Design of Deep Neural Networks Formulated as Optimisation Problems

Sushen Zhang

Department of Chemical Engineering and Biotechnology
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Fitzwilliam College October 2021

I would like to dedicate this thesis to my loving parents for their genuine trust and emotional support. I would also like to dedicate this thesis to my supervisor Dr. Vassilios S. Vassiliadis, and my supervisor and advisor Prof. Alexei A. Lapkin, for their guidance throughout the PhD degree.

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee

Sushen Zhang
October 2021

Abstract

Design of Deep Neural Networks Formulated as Optimisation Problems

Sushen Zhang

The design of deep neural networks (DNNs) involves the explicit definition of network architecture as well as the training of the network weights. Each process can be formulated into an optimisation algorithm and can be investigated with regard to optimisation performance. The training of the network weights is defined as a minimisation of the objective function with regard to network parameters. The architecture search is an optimisation of the objective function with regard to the presence/absence of layers or neurons. I draw similarity between the two scenarios, and propose frameworks that define either the training or the architectural optimisation of DNNs, or a combination of both. The contribution of the thesis is six-fold, in which I propose: 1) a quasi-Newton training algorithm based on Truncated Newton and Gradient Flow methods, 2) a lifting scheme to allow network sparsification, 3) a lifting framework to automatically evolve neural architectures, 4) a multi-scale hierarchical search framework involving sensitivity analysis suitable for the training of neural networks, 5) a heuristic search algorithm for architectural optimisation of a dynamic model, and 6) a dynamic cascade learning model solved in the context of *de novo* drug

design. In each contribution, I define the optimisation problem and solve the optimisation problem under different frameworks. The ultimate aim of this research is to facilitate the democratisation of AI, enabling people with less domain expertise to participate in the design of a deep neural network under a guided framework.

Acknowledgements

I would like to thank the Cambridge Overseas Trust and the China Scholarship Council for their funding of this research. I would also like to thank Prof. Leroy Cronin and Dr. Laurie J. Points, School of Chemistry, University of Glasgow, for hosting and letting me collect data for the OILDROPLET dataset. I would like to thank Dr Laurie J. Points for the help with collecting data for the OILDROPLET dataset.

Contents

List of Figures

List of Tables

Nomenclature

Roman Symbols

A	Architecture
a, b, c, d, e, f	Coefficients of a quadratic model
a_k, b_k, c_k, d_k	Coefficients of the linear transformation
B	Second-order derivative of a function
b	Bias term
C	Steady state concentration
c_1, c_2	Small constants
D	Dataset
d	Search direction in the inner loop of HFGF
E	Expectation
E_a	Activation energy
F	Linear transformation result
F_A	Inflow rate
F_x	Underlying distribution of training dataset
f	Arbitrary function
$f'(net_j)$	Derivative of the activation function of the hidden neuron j (Table 6.1)
$f'(net_k)$	Derivative of the activation function of the output neuron k (Table 6.1)

G	Non-linear transformation result
g	Gradient of loss function
H	Hessian matrix
h	Difference between the supposed output of a neuron and product of activated neuron output and attenuation/amplification factor
I	Identity matrix
J	Cost function
$\mathscr{J}$	Reconstruction error
K	Kinetic constant
k	Data point index
L	Lipschitz constant
$\mathscr{L}$	Loss function
M	Model
m_t	First moment in Adam
$\hat{m}_t$	Bias-corrected first moment in Adam
N	Total number
N_d	Total number of data points
N_i	Total number of neurons
N_j	Total number of neurons in the previous layer
N_k	Total number of data points
N_l	Total number of layers
NN_l	Number of neurons in layer l
n_{fev}	Number of function evaluations
n_{gev}	Number of gradient evalutations

n_{iter}	Number of iterations
O	Objective function for architectural optimisation
P	Productivity
p	Search direction in optimisation
p_{data}	Data-generating distribution
$p_{decoder}$	Decoder mapping
$p_{encoder}$	Encoder mapping
$p(z)$	Prior distribution
p_θ	Decoding distribution
Q	Intermediate matrix in HFGF
Q_{ik}	Percentage of influence of the input variable x_i and output y_k (Table 6.1)
q_ϕ	Encoding distribution
q	Quadratic model as a selector at the branching point
R	Universal gas constant
$\mathscr{R}$	Regularisation function
r_x	Value of the randomized selector to determine the side of a branch to go down
S	Intermediate varialbe in L-BFG
S_{left}, S_{right}	Absolute values of the structural sensitivities at the left and right side of the branching point
s	Sensitivity value
s_k,y_k	Intermediate variables in BFGS derivation
T	Temperature
u	Update gate

V	Volume
v	Output of a neuron after activation (multiplied by the attenuation/amplification factor)
v_{jk}	Connection weight between hidden neuron j and output neuron k (Table 6.1)
v_t	Second moment in Adam
$\hat{v_t}$	Bias-corrected second moment in Adam
W, w	Network weights
$W_{l,i,j}$	Neural network weights at layer l neuron i from neuron j in previous layer
$\overline{W}$	Sum of weights
WP_{ik}	Influence of the input variable x_i on the output y_i (Table 6.1)
w_{rj}	Weights between input neuron r and the hidden neuron j (Table 6.1)
X, x	Input to a neural network
$X^{(train)}$	Training dataset
$X^{(val)}$	Validation dataset
Y, y	Neural network output
$\hat{Y}$	Nueral network predicted output
Z	Diagonal matrix
z	Neuron output after activation

Greek Symbols

α	Angle between random and input vector
β_1, β_2	Coefficient of momentum in Adam
δ	A small disturbance in the value of sensitivity
ε	Trust region bound
$\varepsilon_{machine}$	Machine precision

$\varepsilon_{LSQR,target}$	Width of deadband
ε_{task}	Error of task-specific layers
$\varepsilon^{\star}_{task}$	Tolerance on error of task-specific layers
ε_{total}	Error of shared layers
$\varepsilon^{\star}_{total}$	Tolerance on error of shared layers
ε_{small}	Small disturbance
η	Learning rate
γ	Sum of weights before optimisation
κ	Step-length
Λ	Architectural search process
λ	Lagrandian multiplier for equality constraints
$\overline{\lambda}$	Normalised Lagrangian multiplier
λ_h	Hyperparameter
μ	Optimised coefficient for reference angle
ν	Lagrangian multiplier for inequality constraints
ω	Upper bound on sum of weights
ϕ	Minimisation problem
ρ	Architectural parameter
σ	Activation function
σ_γ	Powerball function
τ	Constant
θ	Attenuation/amplification factor
ξ_{sh}	Number of shared layers
ξ_{ts}	Number of task-specific layers

ζ	Intermediate variable in BFGS

Superscripts

$\star$	Optimal point

Subscripts

i	Neuron index
j	Neuron index from previous layer
l	Layer index
n	Arbitrary Index

Acronyms / Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
AutoML	Automated Machine Learning
BFGS	Broyden, Fletcher, Goldfarb and Shanno method
BO	Bayesian optimisation
CG	Conjugate Gradient
CNN	Convolutional neural network
DNN	Deep Neural Network
GAN	Generative adversarial network
GF	Gradient Flow
GP	Gaussian process
GPU	Graphics computing unit
GRU	Gated recurrent unit
HFGF	Hessian-free Gradient Flow
iLQR	Iterative linear-quadratic regulator

$L-BFGS$	Limited BFGS
$LSQR$	Mean squared loss
$LSTM$	Long-short term memory
MSE	Mean squared error
NAS	Neural architectural search
NCG	Nonlinear Conjugate Gradient
NLP	Nonlinear Programming
PCA	Principle component analysis
QED	Quantitative estimation of drug-likeness
RL	Reinforcement learning
RNN	Recurrent neural network
SAS	Synthetic accesibility score
SGD	Stochastic gradient descent
$SISO$	Single-input single-output
$SMILES$	Simplified molecular input line entry system
SSE	Standard Sum of Errors
TN	Truncated Newton
tol	Total tolerance of SSE
VAE	Variational autoencoder

Chapter 1

Introduction

Since the publication of Rina Dechter's article on the concept of "Deep Learning" in 1986 [1], deep learning has revolutionised the field of pattern recognition and machine learning. The process entails the "credit assignment in adaptive systems with long chains of potentially causal links between actions and consequences" [2]. The practical implementation of deep learning consists of computational models that are composed of "multiple processing layers to learn representations of data with multiple levels of abstraction" [3]. There are different types of processing layers, and one of the earliest development is the feedforward neural network, where the processing layers inherit the structure of a multi-layered perceptron (MLP), with each layer consisting of linear transformations and non-linear activation functions. Following from this prototype, a myriad of neural architectures have been developed to deal with different tasks such as computer vision [4] [5] [6] or natural language processing [7] [8] [9].

Within the domain of deep learning is the Deep Neural Network (DNN), one of the most successful Deep Learning algorithms, which entails a learning process. The learning process is achieved through the arrangement of layers of neurons to simulate a particular transformation from input to output by updating parameters.

The general idea behind deep learning and deep neural networks is a learning process, and the learning process is achieved through the process of training. The learning process entails a learning algorithm A aiming to find a function f that minimises expected loss function $\mathscr{L}(x; f)$ over i.i.d samples x from a natural distribution Fx. The learning algorithm A is a function that maps a data set $X^{(train)}$ (a finite set of samples from Fx) to a function f [10]. In essence, the learning process can be formulated as the optimisation of the loss function with regard to parameters that define the mapping function of f, and this process is termed "the training process". The high number of parameters and the non-linearity introduced into the neural network by activation functions render the loss function a highly complicated one, with many local minima and a difficult-to-describe shape. Therefore, the optimisation

problem is usually hard to formulate and requires careful calculation of a large number of network parameters through the process of backpropagation. This is the problem of training of the DNN, and it can be formulated as an optimisation problem.

In the training process, there are parameters that relate to the optimisation process, instead of the network itself. These parameters are termed "hyperparameters", with examples such as learning rate, number of epochs, *etc*. The search for hyperparameters can similarly be formulated as an optimisation problem. The process involves choosing the right hyperparameter λ_h, that achieves a minimal value of the objective function. It is often considered an outer optimisation problem as opposed to inner optimisation of network parameter values.

While hyperparameters only entail information regarding the training of the neural network, a third class of optimisation problems defined in the context of DNN design is the neural architectural search (NAS). Conventional architectural design relies heavily on manual work and human expertise. On the other hand, automated search for architectures is a computationally heavy process involving a comprehensive exploration in a large search space consisting of combinations of architectural parameters, requiring thousands of GPU hours [11]. The ultimate goal in such research areas focuses on how to reduce the computational complexity and how to automate the process. An important consequence of the process is the democratisation of AI techniques, enabling access to AI practices by a layman or by a cross-disciplinary researcher. Current efforts along this line include Google's autoML and Yelp's Metric Optimisation Engine (MOE). However, the commercialised product either lacks transparency or is limited in scope [11]. To find an effective architectural search algorithm is an ongoing research effort, and is of interest to this thesis and to the research community at large.

1.1 Formulation of a Neural Network

The earliest and simplest form of DNN is the feedforward neural network, or a multilayer perceptron (MLP). Its goal is to perform a mapping from a set of inputs x to a category y, defined by the mapping $y=f(x;\ \theta)$, learning the parameter θ along the way to generate predictions for a new set of incoming x. A DNN consists of many layers of calculation units, termed neurons. In each neuron, values output from the previous layer undergo a linear combination with the weights and biases, then they undergo a non-linear transformation, often termed activation. The process is described as:

$$z = w^T x + b \tag{1.1}$$

$$h = \sigma(z) \tag{1.2}$$

where w is the weighting of the network, b is the bias, z is the result of the linear combination, $\sigma(\cdot)$ is the activation function, and h is the result of activation and the input to the next layer of neurons. Common activation functions include ReLU, leaky ReLU, tanh, and sigmoid functions.

By stacking layers of neurons, the DNN performs the basic chained calculation, in the form of $f(x) = f_n \cdots f_3(f_2(f_1(x))$ [12]. As there are linear and non-linear components in each transformation, the overall result is a complicated composite function that, with proper tuning of the parameters and appropriate choice of the activation function, can represent any complicated transformations from an input to an output. Thus, neural networks are often viewed as "Universal Function Approximators".

1.2 Training and Hyperparameters Tuning

In supervised learning, the complete design of a DNN will involve three datasets, each entailing one step in the definition of the network. First, the training dataset is used in a supervised process to find the optimal weights and biases in the network. In particular, one optimises the weights and biases such that the output of the network matches the targeted output defined by the labels in the training dataset. Second, parameters other than the weights and biases are optimised using the performance on the validation set, in a process called "hyperparameters optimisation". Third, the ability of the network to generalise to other dataset is tested on the testing dataset.

The training process is essential as it evaluates the parameters (weights) of the network that directly determine the performance of a regression or a classification task. To find the optimal parameter values, an unconstrained minimisation of the objective function with respect to network parameters is performed. The optimisation algorithm adopted as well as the initial point determines the value of the local minimum the optimiser reaches and thus the performance of the neural network.

To formulate the training process as an optimisation problem:

$$w^\star = \underset{w \in \mathscr{W}}{\operatorname{argmin}} \mathscr{L}(x; w(X^{(train)})) \tag{1.3}$$

where w represents network parameters (weights), and $X^{(train)}$ is the training dataset.

The performance of a neural network depends both on the parametric model and hyperparameters. Hyperparameters tuning is the second step of formalising an appropriate model for

a specific dataset. Traditional machine learning methods select values of hyperparameters arbitrarily, based on human expertise. However, this tuning process is unsystematic and inaccurate [10]. Moreover, tuning of hyperparameters cannot be exhaustive when dataset is large [10] [13]. Therefore, many other methods, including random search [10], grid search [10], Bayesian optimisation [14] [15], reinforcement learning [16] and evolutionary algorithms [13] [17], have been adopted to perform hyperparameters tuning with a clearly defined search space and a well-defined search strategy.

The hyperparameters tuning problem can be defined as an optimisation process. The process is defined as follows:

$$\lambda_h^{(*)} = \underset{\lambda_h \in \Lambda}{\operatorname{argmin}} E_{x \sim F_x}[\mathscr{L}(x; M_{\lambda_h}(X^{(val)}))] \tag{1.4}$$

where λ_h is the hyperparameter, M is the model defined by the hyperparaemters λ_h on the validation dataset X^{val}, $\mathscr{L}$ is the loss function and Fx is the natural distribution of the data.

1.3 Architectural Search Optimisation

Neural Architecture Search (NAS) is the process of automating architecture engineering. With a defined search scheme, the parameters defining the architecture of the neural network are optimised. Current techniques in optimising neural architecture involve random search [18] [19] [20], grid search [10] [19] [21] , Bayesian optimisation [11] [22], reinforcement learning [19] [23] [24] [25], evolutionary algorithm [26] [27] [28] and gradient-based methods [29] [30] [31]. With regard to the search methodology, there are two separate schools of thoughts: one focuses on training a network larger than necessary and then reducing the network size through the process of pruning [32]; the other seeks to optimise the architecture by starting from some minimal initial structure and adding neurons or layers to the network through the process of dynamic construction [33] [34] [35].

In most problems, the search for optimal architecture is separately defined from the training process. However, cutting-edge research often focuses on one-shot models that trains the network and the architectures jointly. This method has an edge in terms of training speed, reducing days of GPU time to hours.

It is possible to define the architectural search problem as an optimisation process. The NAS is defined as:

$$\Lambda : D \times A \rightarrow M \tag{1.5}$$

where Λ is a search process that maps the data D using the architecture A to model M. The objective function for NAS is defined as:

$$\Lambda(x,\rho) = \underset{m_{\rho,\theta}\in M_\rho}{\operatorname{argmin}} \mathscr{L}(m_{\rho,\theta}, X^{(train)}) + R(\theta) \tag{1.6}$$

where ρ represents architectural parameters and R is a regularisation term. The goal is then to find the optimal architectural parameter $\rho^\star$:

$$\rho^\star = \underset{\rho\in\mathscr{A}}{\operatorname{argmin}} O(\Lambda(\rho, X^{(train)}), X^{(val)}) \tag{1.7}$$

where $X^{(val)}$ is the validation dataset, and ρ is the architectural parameter.

1.4 Research Aims and Objectives

This research focuses on the design of deep neural networks by reviewing important optimisation processes involved in building a functional network from scratch: the training and the architectural search. In both processes, the search to the optimal solution is formulated as an optimisation problem.

The first part of the research involves developing an intuition around the training of deep neural networks, formulated as the optimisation of the objective loss function with regard to the weights and biases in the network. Conventional practices in the industry often adopts first-order stochastic optimisers for this process. This research proposes a quasi-Newton method that is capable of performing the same task with generally comparable performance to the current optimisers. In defined conditions, it is shown that the optimiser outperforms the current second-order optimisers in computational speed. I also briefly discuss the tuning of hyperparameters, which is often viewed as an inherent part of the training process, and also an important optimisation process involved in the design of deep neural networks. I motivate this research because an improved optimisation method on the training process will greatly increase the efficiency of neural network tuning and will facilitate the design of deep neural networks.

A second method on the training of DNNs is also proposed based on sensitivity values of neurons. The training method adopts a multi-scale hierarchical search to select the most sensitive layer to optimise in a binary search tree. I motivate this research also to increase the training speed of very large networks. When only selective layers are optimised, the method has the potential to become faster compared to traditional end-to-end backpropagation method and has great potential for parallelisation.

Transitioning from the proposal of training algorithms, the thesis also develops algorithms for architectural search. After reviewing key techniques adopted in architectural search, including random and grid search, Bayesian optimisation, reinforcement learning, evolutionary algorithms and gradient-based methods, the thesis then proceeds to develop an autonomous self-evolving network adopting the lifting scheme. This includes a sparsification scheme that focuses on pruning a larger than necessary network and an evolution scheme based on network sensitivities that can interactively increase or decrease the network size. I motivate this research because an automated method for the evolution of the network is an important step in the design of neural networks. It allows an efficient search to optimal architecture without heavy computations required, and with minimal input from users, it allows a simple-to-use method to adapt neural architectures.

The research then proposes an architectural search method for small-scaled datesets and large-scaled datasets respectively, adopting dynamic models that alter the architecture as training proceeds. The discussion on small-scaled datasets involves a heuristic search scheme that optimises the architecture of a neural network adapted for multi-task learning. The optimisation is tested on industrial datasets in the chemical engineering industry. The large-scaled datasets adopt deep cascade learning to optimise a generative model. The optimisation process is tested on the problem of *de novo* chemical design.

Overall, the design of deep neural networks falls into the "training-architectural search" paradigm, both can be formulated as optimisation problems. I would like to advance research in the design of DNNs because of the wide applications of DNNs in different research fields, including speech recognition [36] [37] [38] [39], autonomous driving [40] [41], natural language processing [7] [8] [42], computer vision [43] [44] [45] [46], image recognition [5] [47] [48], drug design [49] [50] and personalised medicine [51] [52] [53]. Moreover, the current research in deep neural networks relies heavily on domain expertise in order to design a neural network that is best suited to the task. This elevates the entry barrier to research on artificial intelligence and makes it difficult for a layman to participate in research of deep learning [54]. I am motivated by this thesis to facilitate the democratisation of AI, a process empowering people to participate in AI research [54]. With automated ways of training and architecture optimisation, it is thus possible to enable more understanding into the elements of deep neural networks such as architecture and weights, and therefore also encourage more adoption of DNNs in different research areas with less reliance on domain expertise.

1.5 Thesis Overview

This thesis consists of 9 chapters. Chapter 2 demonstrates cutting-edge research that formulates the techniques used in the design of deep neural networks. Different approaches to optimisation involved in training and architectural search have been discussed.

Chapter 3 focuses on how the training process is solved as an optimisation problem and proposes a novel optimisation method for the training of deep neural networks.

Chapter 4 proposes a method to evolve a neural network through sparsification with the network formulated under the lifting scheme.

Chapter 5 develops a method to evolve the neural network dynamically based on sensitivity values with the convergence criterion that the objective falls into a dead-band range.

Chapter 6 returns to the problem of network training and discusses a multi-scale hierarchical search method to act as an alternative tuning method, by searching for the most sensitive layer through a binary search tree.

Chapter 7 discusses a novel architectural search method involving dynamic architecture construction and proposes a heuristic optimisation scheme that optimises the architecture when adapted to multi-task learning.

Chapter 8 proposes a dynamic architectural search optimisation method for deeper models and applied the search method to a generative model of variational autoencoders.

Chapter 9 evaluates the proposed methods in the thesis and discusses the potential for integration into a framework. Moreover, popular topics in the machine learning literature are discussed in relation to methods proposed in this thesis and future works are investigated.

Chapter 2

Literature Review

Research on the design of deep neural networks (DNNs) involves the important processes of training and architectural search. Earlier research defines an arbitrary architecture and finds the network parameters (weights) through the training process. More recent research questions the optimality of an arbitrarily defined architecture and proposes various schemes that search for the most effective architecture through a dynamic process. This is done by introducing flexibility to the architecture and by defining the neural architectures to be able to interact with the search scheme. Both processes are crucial to the development and implementation of a DNN. This thesis aims to define key optimisation problems involved in the design of DNNs, namely the processes of training and architectural search. Following the "training-architectural search" paradigm, I conduct literature review of each process in sequence.

This chapter is organised as follows:

- Section 2.1 reviews key optimisation algorithms used in the training of DNNs including first-order, second-order and quasi-Newton methods.
- Section 2.2 reviews the optimisation algorithms used in the architectural search of DNNs.
- Section 2.3 concludes by discussing the frameworks commonly used in the autoML pipeline.

2.1 Training of a Neural Network

The training of a neural network can be formulated as an optimisation problem. It is defined as the optimisation of the loss function - a large-scale, nonlinear and non-convex function -

formed by the complicated topology of the neural network [12] [55]. The most widely used loss functions include the mean squared error (MSE) loss [56] [57] [58], the cross-entropy loss [59] [60] [61], the hinge loss [62] [63] [64], the Kullback-Leibler loss [65] [66] and the Huber loss [67] [68]. Depending on whether the output to the neural network is discrete (a classification task) [69] [70] or continuous (a regression task) [71] [72], different objective functions are used.

In this thesis, the MSE loss is used for regression tasks. It is defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.1}$$

where n is the sample size, Y_i is the true label, and $\hat{Y}_i$ is the predicted output from the model [73].

2.1.1 Classification of Optimisation Problems, Solutions and Algorithms

Optimisation problems can be classified into discrete optimisation and continuous optimisation. The former involves the variables taking specific values such as integers, binary values or permutations of an ordered set [74]. Typical algorithms include integer programming [75] [76] and mixed integer programming [77] [78]. On the other hand, the optimisation of neural networks is a continuous optimisation problem as the values of weights are uncountably infinite [79]. This simplifies the optimisation process as the smoothness of the functions allows the use of objective functions, derivatives and constraint information to find the solution [74].

Another classification of optimisation problems is constrained versus non-constrained optimisation [74]. Constrained optimisation sets a group of boundary conditions that may or may not lead to a feasible set. The solution is bounded to the feasible set and optimisation is performed in the bounded area [80] [81]. Unconstrained optimisation is unbounded. The objective function can be minimised with no restrictions at all on the values of its variables [82] [83]. The optimisation of DNNs is in itself an unconstrained optimisation problem, where no limits are imposed on the values of weights [84] [85]. In specific problems, the objective function can be defined to be bounded to certain regions [86] [87]. For example, in the problem of *de novo* chemical design, the output of the neural network can be bounded to certain regions that produce the valid molecular formulas [88]. In these cases, the optimisation becomes a constrained process and different optimisation techniques are applied.

A key characteristic of the solution to an optimisation problem is whether it belongs to a global or local optimum [89]. A point $x^\star$ is a global minimizer if $f(x^\star) \leq f(x)$ for all x, where

x ranges over all of $\mathbb{R}^n$ [74]. A point $x^\star$ is a local minimizer if there is a neighborhood N of $x^\star$ such that $f(x^\star) \leq f(x)$ for all $x \in N$ [74]. In the optimisation of DNNs, the ideal is always to find the global optimum since I could minimise the loss function to the greatest extent [90]. However, there is usually no guarantee of global optimality even with cutting-edge algorithms given the highly complicated nonconvex nature of the objective function of DNNs [91] [92]. Meanwhile, the research community often accepts local optima as the solution since they usually produce empirically satisfactory results in tasks such as image analysis [93] or speech recognition [94] [95]. To prove that an algorithm optimises a point to a global minimum has been an on-going research topic [96] [97] [98].

There are salient features of optimisation algorithms that separate them into classes. The first distinction I can draw to classify optimisation algorithms is the difference between stochastic methods and deterministic methods [74]. Stochastic optimisation methods entail within them some degree of uncertainty and possess some inherent randomness [99] [100]. The same set of parameter values and initial conditions will lead to different outputs [99]. On the other hand, in deterministic models, the output of the model is fully determined by the parameter values and the initial conditions [101] [102].

The second distinction to draw is the difference between first-order, quasi-Newton and second-order methods [74]. First-order methods are based on first-order derivatives information only, thus only the Jacobian matrix is used [103] [104]. Second-order methods also make use of second-order derivatives information and a Hessian matrix is often employed [105] [106]. In between the two classes of methods are the quasi-Newton methods where the Hessain matrix is approximated by first-order information [107] [108]. From Taylor theorem,

$$f(x_k + p_k) = f(x_k) + p_k^T \nabla f_k + \frac{1}{2} p_k^T \nabla^2 f(x_k) p_k \tag{2.2}$$

$$x_{k+1} = x_k + \kappa p_k \tag{2.3}$$

where p is the search direction and κ is the step-length. Differentiating with respect to p and equating to 0 gives

$$\nabla f(p_k) = \nabla f_k + \nabla^2 f(x_k) p_k = 0 \tag{2.4}$$

Defining $\nabla^2 f(x_k) = B_k$ and rearranging,

$$p_k = -B_k^{-1} \nabla f_k \tag{2.5}$$

If I let $B_k = I$, I arrive at the first-order method of steepest descent. If I let $B_k = \nabla^2 f(x_k)$, I arrive at the second-order Newton method. If B_k is the approximated Hessian, I arrive at the quasi-Newton method.

2.1.2 An Outline of Stochastic First-order Methods

A number of first-order stochastic methods are present in literature. Here I present some of the common methods adopted in the machine learning literature to optimise a deep neural network.

Gradient Descent Methods

Gradient Descent [109] [110] [111] is the earliest and most basic form of optimisation for deep neural networks. Most important state-of-the-art *Deep Learning Libraries* contain implementations of gradient descent algorithm (e.g. Caffe, Keras, Tensorflow, Pytorch). The gradient descent method entails some lack of transparency and is often viewed as a black-box optimiser [112].

Gradient descent optimises the objective function $\mathscr{L}$ with respect to parameters w. The search direction is defined to be a constant multiple of the negative gradient [109]. The multiple, η, is termed the learning rate and determines the step size towards the minimum. It is a first-order method because it only makes use of the gradient information.

There are three basic variants of gradient descent (GD): batch GD [113] [114], stochastic GD [115] [116], and mini-batch GD [117] [118]. Batch gradient descent computes the gradient using the information provided by the entire training dataset:

$$w = w - \eta \cdot \nabla_w \mathscr{L}(w) \tag{2.6}$$

Batch GD (BGD) cannot perform online calculation and sometimes cannot fit in the local memory when the data size is large [114]. Moreover, the amount of calculation required in each update is high [114]. The advantage is that it is guaranteed to converge to local minima in non-convex cases [112].

Stochastic gradient descent (SGD) performs a parameter update for each training example.

$$w = w - \eta \cdot \nabla_w \mathscr{L}(w; x^{(i)}, y^{(i)}) \tag{2.7}$$

The advantage of SGD compared to BGD is that it reduces the amount of redundant calculations of derivatives [119]. While in BGD, the gradients of the whole batch is calculated after each updates, only one gradient is calculated in SGD. This feature makes SGD highly suitable for online learning [120]. The disadvantage is that SGD tends to zig-zag in its descent and around the optimal point [121]. The optimal point is usually not exactly calculated if the learning rate is high [112].

A compromise is made in the algorithm of mini-batch gradient descent (mini-batch SGD), where a small number of samples are used to update the parameters in each round:

$$w = w - \eta \cdot \nabla_w \mathscr{L}(w; x^{(i:i+n)}, y^{(i:i+n)}) \tag{2.8}$$

Mini-batch SGD combines the advantages of BGD and SGD. On one hand, it reduces the amount of calculations per update round. On the other hand, it enables stable convergence [122]. Therefore, this is the most commonly used form of gradient descent in practice [122].

Adaptive Gradient Methods

In some variants of GD algorithms, the learning rate is adaptive in nature, *i.e.* it is not constant and adjusts to the value of the parameters. Commercially used adaptive gradient methods include Adagrad [123] [124], Adadelta [125] and Adam [126].

One of the earliest algorithm adopting an adaptive learning rate is Adagrad [123]. The key equation governing the formulation of Adagrad is:

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{Z_{t,ii} + \varepsilon_{small}}} \cdot g_{t,i} \tag{2.9}$$

$Z_{t,ii}$ is a diagonal matrix where the i^{th} diagonal element is the sum of the squares of the gradients with respect to w_i up to step t. ε_{small} is an infinitesimal number to prevent division by zero, and g is the gradient from the previous step. To vectorise:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{Z_t + \varepsilon_{small}}} \odot g_t \tag{2.10}$$

Another popular GD variant is Adadelta. While Adagrad makes use of sum of the squares of all previous gradients, Adadelta defines a recursive decaying average of previous gradients:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{2.11}$$

where $E[g^2]_t$ is calculated from the previous iterations and the current gradient, and γ is the controlling coefficient of the decay. The update rule is:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon_{small}}} \odot g_t \tag{2.12}$$

Momentum Methods

The SGD with momentum method is often used when different dimensions of the network have different scales, leading to a search direction that zig-zags towards the local minima. Momentum accelerates downward movement and dampens oscillations [112]. A general expression of the SGD with momentum is expressed as:

$$v_t = \gamma v_{t-1} + \eta \nabla_w \mathscr{L}(w) \tag{2.13}$$

$$w = w - v_t \tag{2.14}$$

where v_t is the momentum term expressed as a decaying sum of gradients.

A highly popular optimisation method is the Adaptive Moment Estimation (Adam) optimiser [126]. Similar to Adadelta, the previous gradient values are stored to determine the learning rate. The first moment m_t and the second moment v_t is calculated:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.15}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{2.16}$$

To find the bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.17}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.18}$$

The update rule is thus:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon_{small}} \hat{m}_t \tag{2.19}$$

2.1.3 An Outline of Deterministic Newton Methods

Second order methods are important due to its higher potential to distributed training in developing neural networks [127]. Moreover, second-order methods make use of curvature information hence can arrive at the minimum point in fewer iterations. Stochastic second-order methods have the advantage of both escaping from local minima and obtaining a faster search process, and can benefit from larger mini-batches [128]. Second-order methods typically have less hyperparameters to tune compared to common variants of SGD [127].

Most second-order method are based on Newton method. The search direction in the Newton method can be formulated as:

$$p_k = -H_k^{-1} \nabla f_k \tag{2.20}$$

2.1.4 Quasi-Newton Methods

Quasi-Newton methods approximate the Hessian information based only on first-order derivatives [107]. Gradient estimates are required to have low variance such that the estimated results have low variance [127]. Thus, there is a trade-off between epoch number and mini-batch size [127], allowing mini-batches to be evaluated in parallel.

The most commonly used quasi-Newton method is the L-BFGS method [129]. It is the limited-memory version of the BFGS algorithm that approximates the inverse Hessian. The advantage is that it is empirically robust to large-scale problems. However, the algorithm is derived in the convex context and hence can become ineffective in large-scale non-convex problems such as optimisation of deep neural networks [74].

BFGS Method

The BFGS method is named after the inventors Broyden, Fletcher, Goldfarb, and Shanno [74]. To derive the search direction, the algorithm is first defined from the quadratic model:

$$m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p \tag{2.21}$$

The minimiser p_k is thus defined as:

$$\nabla m_k(p) = \nabla f_k + B_k p = 0 \tag{2.22}$$

$$p_k = -B_k^{-1} \nabla f_k \tag{2.23}$$

where p_k is the search direction. I define values of next iteration with step length κ.

$$x_{k+1} = x_k + \kappa_k p_k \tag{2.24}$$

The calculation of B_k is an intensive step. Thus, instead of computing B_k in every iteration, it can be updated with the curvature information in the most recent step

$$m_{k+1}(p) = f_{k+1} + \nabla f_{k+1}^T p + \frac{1}{2} p^T B_{k+1} p \tag{2.25}$$

To derive the update rule, the gradient of m_{k+1} should match the gradient of the objective function f at x_k and x_{k+1}. The initial point is

$$\nabla m_{k+1}(0) = \nabla f_{k+1} \tag{2.26}$$

The previous point is

$$\nabla m_{k+1}(-\kappa_k p_k) = \nabla f_{k+1} - \kappa_k B_{k+1} p_k = \nabla f_k \tag{2.27}$$

Rearranging,

$$B_{k+1}\kappa_k p_k = \nabla f_{k+1} - \nabla f_k \tag{2.28}$$

Define the vectors

$$s_k = x_{k+1} - x_k = \kappa_k p_k \tag{2.29}$$

$$y_k = \nabla f_{k+1} - \nabla f_k \tag{2.30}$$

Then,

$$B_{k+1} s_k = y_k \tag{2.31}$$

Imposing the condition that among all symmetric matrices satisfying Equation 2.31, B_{k+1} is closest to B_k

$$\begin{aligned} &\min_B ||B - B_k|| \\ &s.t.\ B = B^T,\ Bs_k = y_k \end{aligned} \tag{2.32}$$

The solution to the above minimisation problem is the Davidon–Fletcher–Powell (DFP) update rule

$$B_{k+1} = (I - \zeta_k y_k s_k^T) B_k (I - \zeta_k s_k y_k^T) + \zeta_k y_k y_k^T \tag{2.33}$$

where $\zeta_k = \frac{1}{y_k^T s_k}$. Define $H_k = B_k^{-1}$, the DFP update rule is calculated to be

$$H_{k+1} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{y_k^T s_k} \tag{2.34}$$

The BFGS rule is derived from the minimisation problem

$$\begin{aligned} &\min_H ||H - H_k|| \\ &s.t.\ H = H^T,\ Hy_k = s_k \end{aligned} \tag{2.35}$$

The solution, similar to Equation 2.33, is defined to be

$$H_{k+1} = (I - \zeta_k s_k y_k^T) H_k (I - \zeta_k y_k s_k^T) + \zeta_k s_k s_k^T \tag{2.36}$$

which is the BFGS update rule.

L-BFGS Method

The Limited Memory BFGS (L-BFGS) is useful in problems with a large dimension becasue instead of saving a $n \times n$ matrix of approximated Hessian, it saves only a few vectors to represent the approximation. If I define $S_k = I - \zeta_k y_k s_k^T$, then from Equation 2.36,

$$H_{k+1} = S_k^T H_k S_k + \zeta_k s_k sk^T \tag{2.37}$$

Since S_k and ζ_k are defined purely from $\{s_k, y_k\}$, it is possible to store a fixed number of pairs of $\{s_k, y_k\}$, instead of the whole approximated Hessian matrix. The update rule is then

$$\begin{aligned} H_k =& (S_{k-1}^T \cdots S_{k-m}^T) H_k^0 (S_{k-m} \cdots S_{k-1}) \\ &+ \zeta_{k-m} (S_{k-1}^T \cdots S_{k-m+1}^T s_{k-m} s_{k-m}^T (S_{k-m+1} \cdots S_{k-1}) \\ &+ \zeta_{k-m+1} (S_{k-1}^T \cdots S_{k-m+1}^T s_{k-m+1} s_{k-m+1}^T (S_{k-m+2} \cdots S_{k-1}) \\ &+ \cdots \\ &+ \zeta_{k-1} s_{k-1} s_{k-1}^T \end{aligned} \tag{2.38}$$

2.1.5 Discussion

One distinct feature of first-order methods is simplicity [130]. Given the increasing computing power of the current computing systems, the first-order methods are advantageous in their robustness and in overcoming saddle points [131] [132]. On the other hand, second-order methods are less well researched. This is due to the increased computational complexity brought about by the introduction of the Hessian matrix. However, quasi-Newton methods approximate the Hessian to reduce computational complexity [133]. Moreover, in theory the quasi-Newton methods take less search steps to reach the optima compared with first-order methods [133]. Thus, research into quasi-Newton methods have great potential to be applied to the optimisation of large-scalee deep neural networks.

In the field of optimisation, there is the "no free lunch" theorem [134]. That is, no single optimiser is better than all other optimisers for all optimisation problems. Thus, it is important to explore the advantages and limitations of each model and apply each optimiser with care to different optimisation conditions.

Another important and integral step to the optimisation of model parameters is hyperparameter optimisation [135] [136]. I briefly discuss hyperparamter optimisation in this section because it should be viewed as an indivisible process to the training of the network. Traditional hyperparameter optimisation is performed through manual work but with the advances in computer cluster and GPU technology, more systematic search methods have been developed [137] [138]. While optimisation of model parameters is a continuous black-box optimisation that often uses first- or second-order derivative information, hyperparamter optimisation can be discrete and entails a few parameters that are optimised combinatorically. Common search methods include random search [10], grid search [10], Bayesian optimisation [14] [15], reinforcement learning [16] and evolutionary algorithms [13] [17]. The choice of hyperparamters plays an essential role in determining the performance of a neural network [139]. It is equally important compared to model parameters in defining the end result of a neural network.

2.2 Architectural Search Optimisation

A simplistic scheme involved in the design of a DNN follows the "training - hyperparameters tuning" process. After training and optimising the hyperparameters of a neural network, however, it is important to query the fundamental question of how effective the network is. Sources of ineffectiveness often arise in the manual component of the network design where the number of layers or the number of neurons are arbitrarily defined. Therefore, further the analysis of optimising hyperparameters, it is important to delve into the area of architectural search.

There is a fuzzy distinction between hyperparameters optimisation and architectural search optimisation [28]. The former focuses more on parameters related to training of the neural network, such as learning rate, momentum, number of epochs, *etc.* The latter calculates parameters that define the topology of the neural network, usually preserving values of connectivity, number of layers, number of neurons, *etc.* The similarity between hyperparameter optimisation and architectural search is that both aim to find a set of metaparameters usually based on the validation set accuracy, *i.e.* both share the validation dataset in defining their values [28]. Moreover, both can be formulated as an optimisation problem with overlapping techniques commonly adopted to achieve an optimum, such as Bayesian optimisation [11] [14] and Evolutionary algorithms [13] [27]. The difference is that architectural search focuses more on building an archetype that defines the wholeness of the network whereas hyperparameters optimisation focuses more on training parameters. Some

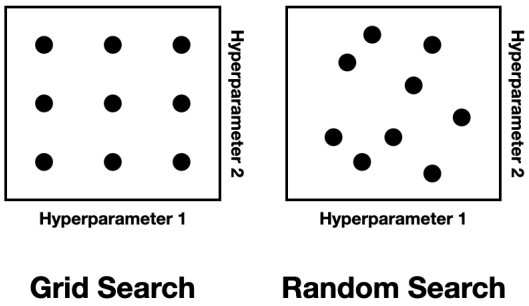


Figure 2.1 Comparison between grid and random search of nine trials for optimising a two-dimensional space function. Random search explores more values in the hyperparameters and is more likely to find the optimal combination than grid search

techniques such as gradient-based methods can only be applicable in architectural search optimisation [31].

The research on architectural search has mostly been focused on defining the search space, the search strategy and the performance estimation strategy [11]. The search space, if properly defined based on contextual knowledge, can greatly reduce the complexity of a search process, but it may introduce unnecessary biases by limiting the search space. The search strategy encapsulates how to find a suitable architecture in the search space and is the most critical step in the definition of a search methodology. The performance estimation strategy, on the other hand, serves as a metric to evaluate the effectiveness of search strategies. Traditional performance estimation is based singly on the validation accuracy but developing other methods is an on-going research process [140]. The performance estimation strategy can also interact with the search strategy in a feedback loop where searches are directed to improve the results of performance estimation [11]. This thesis focuses on the search strategy and a review of different search strategies commonly found in literature is provided in the following sections.

2.2.1 Random Search and Grid Search

In the early years, the most widely used methods in searching for an optimal architecture are random search and grid search [19]. Grid search [18] [20] divides the search space into regular intervals (the grid), and assigns the values of the architectural parameters based on values on the grid. Random search [10], on the other hand, selects values of architectural parameters at random from the search space. In both cases, the best performing set of values are stored as parameter values.

Grid search has the advantage that it is simple to implement in parallel and has the capability to find better parameters than manual search [19]. The disadvantage is that the search points are uniformly placed without emphasis on more important parameters and on more important areas in the search space [19]. This is illustrated in Figure 2.1, where only three values of important parameters are searched for in the grid search, and more values are explored in random search. In [10], it is demonstrated that not all parameters are equally important to search for and grid search explores too much in the unimportant areas. In [21], it has been suggested that a coarse grid search can be performed followed by a random search near the region where grid search has generated a good result. In [141], a contracting-grid search algorithm is proposed where there is a maximum likelihood value for each point in the gird. Grids are successively halved based on the value until convergence to a local minimum.

In both grid search and random search, the effectiveness of results is generally increased with longer search time, with a higher likelihood to find the optimal parameters with more trials. However, there is no guarantee that optimality is found with increasing number of iterations. Thus, there is the trade-off between computational time and optimality [16]. A potential solution is shown in [16] where better performing half of parameters are sought for by successively removing the worse half.

2.2.2 Bayesian Optimisation

The key improvement by adopting the method of Bayesian Optimisation (BO) in architectural search is the successive search for the optimal set of architectural parameters [22] [142]. It is less popular in architectural search compared to hyperparameters tuning because most BO packages are based on Gaussian Processes (GP) and mainly deal with low-dimensional continuous problems [11]. A typical architectural search adopting Bayesian optimisation is performed in [22].

2.2.3 Reinforcement Learning

Reinforcement learning methods have been adopted in neural architectural search since the architectural search process can be formulated as the architectural design agent searching for reward (the validation accuracy) [25]. However, such methods often suffer from high computational complexity and time [19].

The central idea in adopting reinforcement learning to architectural design is the conversion of structural connectivity to a variable-length string [25]. A popular practice is to make use of a recurrent network (RNN) as the controller to generate the string, and train the string with respect to validation accuracy. In the case of [25], the controller is designed to be

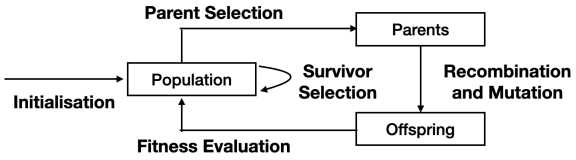


Figure 2.2 A framework for evolutionary algorithms

autoregressive and the key challenge is to optimise a non-differentiable metric of validation accuracy. MetaQNN [143] then makes use of a meta-modelling algorithm using Q-learning with an ε-greedy exploration strategy and experience replay. It takes 10 days and 10 GPUs to search for architectures on different dataset including CIFAR-10, CIFAR-100, SVHN and MNIST. Afterwards, many research developed cell-based search that greatly reduced the amount of time spent, including NASNet [144], BlockQNN [24] and ENAS [23]. ENAS creates a great reduction in GPU time to 10 hours using 1 GPU. It treats each trained network as a sub-graph of the search space so each child network is not trained from scratch.

2.2.4 Evolutionary Algorithms

Evolutionary algorithm (EA) is a generic population-based meta-heuristic optimisation algorithm [19] with high robustness and wide applicability. Compared to reinforcement learning, evolutionary algorithms operate on much larger search spaces. It has been actively researched for three decades, initially used to evolve weights of the neural network [26]. It is a "population-based global optimizer for black-box functions" [28]. The process of optimisation is outlined as follows (also outlined in Figure 2.2):

1. Initialise the first generation of population
2. Select parents from the population for reproduction
3. Recombine and mutate operations to create new individuals
4. Evaluate the fitness of new individuals
5. Select the survivors of the population

An example is the NueroEvolution of Augmenting Topologies (NEAT) algorithm [27]. It is defined as the artificial evolution of neural networks using genetic algorithms. From this algorithm, many modified versions have been developed, including CoDeepNEAT [145] and LEAF [13].

2.2.5 Gradient-based Methods

Unlike other methods that search for the architecture through a black-box optimisation problem over discrete domain, gradient-based methods often seek to convert discrete representations into continuous ones, and optimise the continuous variables together with the network weights [19]. This prevents the need to evaluate a large number of architectures defined by combinations of discrete network parameters [19]. For example, a typical reinforcement learning method requires 2000 GPU days [23] and a typical evolution method requires 3150 GPU days [146]. With a conversion of parameters from discreteness to continuity, I can perform even simple gradient descent in the continuous search space with respect to its validation performance, therefore reducing the reliance on computing power and increasing the search efficiency.

Much of the research on architectural optimisation starts by defining a Super Network that contains a set of layers or blocks of cells connected together in a Direct Acyclic Graph (DAG). This technique has been named differently in different papers such as Deep Sequential Neural Networks [29], Neural Fabrics [31] or PathNet [30]. Architectural search is performed by optimising connectivity between layers of blocks, and the connectivity is usually symbolised by a continuously valued scalar. Optimising the architecture involves calculating the connectivity value, removing connections when the connectivity value is low or choosing the type of connecting operation with the highest value.

Another important element in performing architectural search is on the definition of the loss function. The simplest form of architectural search optimises structure based on validation set accuracy. However, it is possible to redefine the loss function such that other considerations can be included quantitatively in the loss function [147]. The flexibility in defining the loss function has accentuated one of the advantages of performing architectural search through gradient-based methods. Composite objectives can only be easily executed through a gradient-based optimisation method as long as it is differentiable, whereas in other search methods, redefining the objective function may revamp the whole definition of the optimisation problem.

2.2.6 Discussion

In architecture search, there is the intuition that more complicated architectures generate better performance. The intuition is corroborated in the top-performing CNNs in the computer vision community as exemplified by [47], where the performance of AlexNet was improved by making the network deeper. This has been contradicted in [5] [55], that increasing the depth of the network decreased performance, confirmed by experiments in [4]. Moreover,

deeper networks are more difficult to train and more prone to overfitting [148]. Therefore, the hypothesis for an architecture-generating algorithm should not focus on expanding the depth but rather on lower validation loss and higher test accuracy.

The design of the architecture is data-sensitive, relying heavily on human expertise to allocate specific designs to different datasets [10] [135]. Structural parameters are difficult to optimise because it is currently unclear how different architectural parameters interact to generate model performance [13]. In addition, there is no mathematical formula to calculate the appropriate architectural parameters. The choice of architectural parameters is often by trial and error, dependent on the experience of the designer [17]. In areas where there is a lack of expert knowledge, architectural parameters are optimised through random search or grid search, a time-consuming and error-prone process due to the sheer size of available parameter choices [19]. Recent searches of architectural parameters have been formulated as an optimisation problem, producing results exceeding human design [13] [19].

Current Challenges

There are currently three main problems [149] faced by frameworks that conduct architectural search: 1) inadequate baseline for performance gain, 2) high complexity of methods adopted with unclear indication of what achieves competitive empirical results, and 3) lack of reproducibility both in reproducing the same results and generalising to other problems while maintaining the same level of performance. Thus, current frameworks are often ad-hoc in nature with a huge variety of methods adopted to perform neural architectural search.

In current research, it is frequently observed that different research makes use of different baselines to determine improvements of their proposed search strategy. For example, in [23] and [150], the performance of the proposed frameworks are compared to random search with a limited computational budget, giving themselves an unfair advantage to an exaggerated performance gain. Therefore, it is essential to define and adopt an appropriate baseline for performance evaluation. The authors of [149] proposed using random search with early-stopping as a baseline due to its simplicity while being state-of-the-art random search method outperforming leading adaptive search strategies. This baseline is often identified as the benchmark for the evaluation of frameworks for architectural search.

Moreover, the model adopted is usually complex such that the search process is more or less akin to a black-box optimisation process [151]. In many search processes, there is an interplay between training routines, architecture transformations and modelling assumptions, and thus it is unclear what plays a critical role in improving the performance of the architectural optimisation. It is essential to perform ablation studies [149] in order to guide future research into the area.

The third biggest challenge is the lack of reproducibility of current research. It is difficult to 1) reproduce exact results from the models used in literature [152], and 2) the model seldom generalises or stays robust to new datasets [151]. This can be due to the complexity of the model used which takes a high computational cost hence most research report the best performing results based on few runs [19]. Moreover, it can be due to the highly empirical nature of research in this field where search processes are processed and validated by results, instead of by rigorous mathematical proof [152]. The stochastic nature of most optimisation problems, the setting of random seed, the uncertainty involved in hyperparameter tuning all add complexity to the reproducibility, making the current research results difficult to build upon [151].

2.2.7 Architectural Search Methodology

In previous sections I have evaluated different techniques used in the process of architectural search. However, there exists a methodology where the search is not based on a particular technique but more on how to modify a network after the full training scheme. More specifically, the pruning process is often adopted where linkages between neurons are cut-off from the network such that parts of the network completely dissociate from the original network [32]. This process is expected to reduce the degree of freedom of the network, preventing over-fitting and improving generalisation, and in consequence generating a network of minimal size [153].

A most critical rule in obtaining a good generalisable neural network is that one should use the "smallest system that fits the data" [32]. To ensure an effective neural network with good generalisation is produced, the process of pruning is often adopted. Pruning involves training a network that is larger than necessary and then removing parts of it to generate comparable or better results. It is a difficult task because a system too small is faster and cheaper to build but may not be able to handle the given data, while a system too large may slow down performance and become over-sensitive to hyperparameters [32]. Moreover, to determine the part of a network to prune requires careful calculations and sound mathematical underpinnings.

Opposite to pruning is the process of dynamic architecture construction. This methodology starts from a minimal architecture and continuously expands the architecture until a satisfactory objective value is obtained [154] [155] [156]. I will discuss the method more in details in Chapter 7 and Chapter 8.

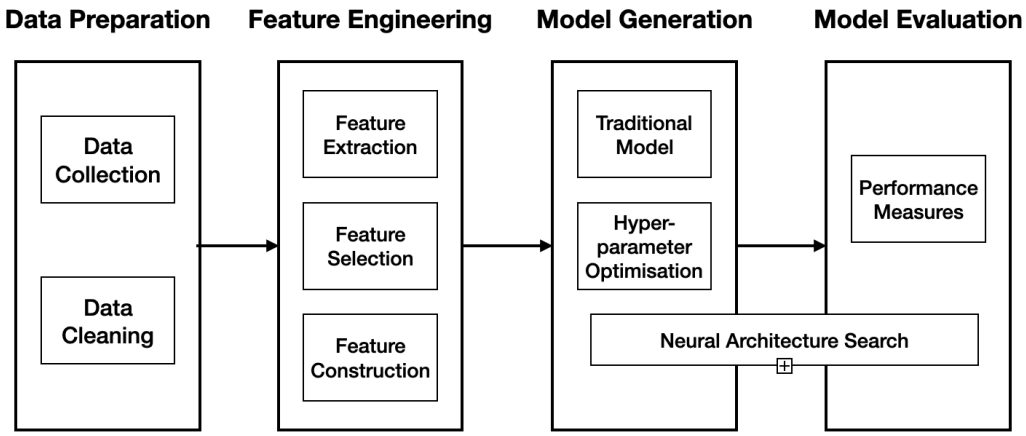


Figure 2.3 An overview of AutoML pipeline covering data preparation, feature engineering, model generation and model evaluation.

2.3 Frameworks on Deep Neural Networks

In this chapter, I have reviewed literature that completes the whole pipeline of developing a deep neural network. The pipeline consists of training, hyperparameter tuning and architectural search, with each process defined as an optimisation problem on its own. It is possible to amalgamate the three processes by defining a single optimisation framework that performs architectural search in the higher level of the framework while performing training in the lower level framework, as exemplified in the research of [150]. There has been a number of literature that proposes frameworks for the whole end-to-end easy-to-use pipeline, often termed Automated Machine Learning (AutoML) [19]. However, AutoML covers the complete process from data preparation, feature engineering, model generation to model evaluation. Depending on the framework, the pipeline may also include definition of constraints, performance evaluation, result analysis, and visualisation [157]. I have put a special focus on model generation and evaluation. Figure 2.3 has illustrated the whole process of AutoML.

AutoML is an important topic in current research because it facilitates the transformation of machine learning from a strictly R&D technology to common enterprise practices, accelerating the democratisation of AI in the process [158]. Democratisation enables businesses and individuals to access and implement AI algorithms with a minimal requirement of expertise, thus promoting the technology in multiple areas of applications while increasing productivity in industrial applications, reducing computational overhead in different research disciplines, unravelling hidden information from data, and allowing customisation of AI models [158].

The current AutoML technology allows the automation of feature engineering and hyperparameter optimisation, the most time-consuming steps in the machine learning pipeline [159]. Freedom is allowed in the constrcution of a machine learning model. Popular AutoML framework in industrial and enterprise practices include Auto_ml, auto-sklearn, TPOT, H2O [159] and Auto-WEKA [160]. A commercialised version is Yelp's Metric Optimisation Engine which is used when objective function is a black box and the derivatives are unavailable.

Auto_ml [1] is one of the popular frameworks widely used in production systems. It contains algorithms for data pre-processing (such as categorical encoding, date processing, numeric scaling) and feature reduction (such as using PCA). For its algorithm implementations, the algorithm utilizes highly optimized libraries such as ScikitLearn, XGBoost, TensorFlow, Keras, and LightGBM [159]. Hyperparameter optimisation is implemented through an evolutionary grid search.

Auto-sklearn [2] is another popular framework that wraps around sklearn for its functionality, thus sklearn estimators are used for modelling processes [159]. The hyperparameter optimisation is performed with Bayesian search techniques. An advantage of the framework is its easy integration with the existing sklearn package. It leverages recent advantages in Bayesian optimization, meta-learning and ensemble construction [161] and introduces these advances to machine learning packages.

TPOT (Tree-based Pipeline Optimisation Tool) [3] makes it easy to optimise hyperparameters. Its hyperparameter optimisation is performed with genetic programming [159]. Unlike auto-sklearn, it defines its own classifier and regressor functions.

H2O [162] is a package for machine learning like the sklearn but contains an automated module. The module automates machine learning with its own internal algorithms. Thus, configuration is limited to algorithm choice, stopping time, and degree of k-fold validation [159]. It also performs an exhaustive search over its feature engineering methods and model hyperparameters to optimize its pipelines. Therefore, it has the key drawback of massive use of computational resources [159].

Auto-WEKA [163] is another optimisation framework that performs hyperparamter optimisation through Bayesian optimisation. The framework is based on a machine learning software package WEKA and builds an automated version in auto-WEKA, containing functionalities of feature selection, ensemble methods, and meta-methods [163].

[1] https://github.com/ClimbsRocks/auto_ml
[2] https://github.com/automl/auto-sklearn
[3] https://github.com/EpistasisLab/tpot

However, these current frameworks extensively make use of traditional machine learning algorithms such as random forest to perform classification and regression tasks. Less utilised is the cutting-edge research into neural architecture search.

The problem of neural architectural search comes into the picture as an important subfield of AutoML. Given the vision that architectural search can be fully automated, the architecture of DNNs can be included in the AutoML framework with benefits in better performance and wider acceptability. Packages that currently contain neural architectural search components are Google's AutoML and AutoKeras.

Google's AutoML [4] is a commercialised cloud development suite that performs automatic neural architecture search based on cutting-edge computer vision technology. However, it is expensive to use and is not open-source.

AutoKeras, on the other hand, is an open-source package that makes use of a variant of ENAS (mentioned in Section 2.2.3) to perform automatic architectural search. It is easy to implement for the community using Keras, and as it is open-source, one can modify the code to implement customised parameters.

There is currently no single algorithm that outperforms all the others. Several pieces of research have attempted to generate a benchmark for the evaluation of the AutoML packages [160] [159]. The benchmarks make use of metrics such as F1 score, MSE or AUC (area under the curve) to make a comparison. In [159], the authors also undertake analysis on data-dependent performance. However, the benchmarks developed in these research overly focus on the final results instead of other aspects such as ease of use, accessibility, interpretability, flexibility to system constraints and reproducibility.

Our research comes into the picture of the AutoML pipeline because the objective is to allow automated search of neural architectures and network parameters. I believe our research will contribute to the methodology of training and neural architecture search and will become an integral part of the AutoML framework.

[4] https://cloud.google.com/automl

Chapter 3

Hessian-free Gradient Flow Method

3.1 Introduction

The development and implementation of a Deep Neural Network (DNN) entails two important aspects: training and architectural search. Both aspects can be defined as optimisation problems and are critical in determining the performance of a DNN. In this chapter, I focus on the optimisation methods used in the training process of DNNs. The training process is formulated as a minimisation of the loss function over a complicated, non-convex function shape defined by the network topology.

I propose a novel optimisation method for the training of DNNs, termed Hessian-free Gradient Flow (HFGF). The algorithm entails the design characteristics of the Truncated Newton, Conjugate Gradient and Gradient Flow method. It employs a finite difference approximation scheme to make the algorithm Hessian-free and makes use of Armijo conditions to determine the descent condition. The method is first tested on standard testing functions with a high optimisation model dimensionality. Performance on the testing functions has demonstrated the potential of the algorithm to be applied to large-scale optimisation problems. The algorithm is then tested on classification and regression tasks using real-world datasets. Comparable performance to conventional optimisers has been obtained in both cases.

This chapter is organised as follows:

- Section 3.2 formulates the training process as an optimisation problem
- Section 3.3 defines the convergence criteria and the conditions for sufficient descent
- Section 3.4 outlines second-order and quasi-Newton methods
- Section 3.5 proposes the novel algorithm of HFGF

- Section 3.6 introduces the standard testing functions commonly used to test the effectiveness of optimisation algorithms
- Section 3.7 tests the algorithm on standard testing functions and hypothetical DNNs. Then it applies the algorithm to real-world datasets of MNIST and OILDROPLET dataset
- Section 3.8 performs further analysis of the proposed algorithm
- Section 3.9 then concludes the chapter

3.2 Formulation of an Optimisation Problem

The training of deep neural networks can be formulated as an optimisation problem. The key difference between training and traditional optimisation is that training is usually indirect [12]. That is, I would like to optimise performance measure P of the test set but could only reduce the cost function $J(w)$ of the training set. The cost function is properly defined as:

$$J(w) = E_{(x,y)\sim\hat{p}_{data}} L(f(x;w),y) \tag{3.1}$$

where E is the expectation, L is the per-example loss function, $f(x;w)$ is the predicted output when the input is x and the output is y. $\hat{p}_{data}$ is the empirical distribution of the input-output pair. More typically, I prefer to train the following cost function:

$$J^*(w) = E_{(x,y)\sim p_{data}} L(f(x;w),y) \tag{3.2}$$

where p_{data} is the data-generating distribution instead of the empirical distribution from the training dataset. Thus, the training process is defined as an unconstrained optimisation process:

$$\min J^*(w) \tag{3.3}$$

At the end of the chapter, hyperparameters tuning is also performed to maximise the performance of the DNNs. I define the hyperparameters tuning process as an optimisation scheme. The optimisation scheme adopted is the sequential heuristic scheme.

3.3 Convergence Criteria and Sufficient Descent

Two assumptions must be made in order to converge to a local minimum of the loss function from any starting point with arbitrary learning rate: 1) the error function is a real-valued

function defined and continuous everywhere in $\mathbb{R}^n$, bounded below in $\mathbb{R}^n$, and 2) for any two points w and $v \in \mathbb{R}^n$, the gradient $g(x)$ of the loss function satisfies the Lipschitz condition $||g(w) - g(v)|| \leq L||w - v||$, where $L > 0$ denotes the Lipschitz constant. Under these assumptions, there is an upper bound on the degree of the nonlinearity of the error function and the first derivatives are continuous [164]. If these assumptions are fulfilled, there is the guaranteed convergence to a minimum with the learning rate appropriately defined by the Wolfe conditions [164].

The *Armijo* condition is a condition for the sufficient descent of an inexact line search method [74]

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k \tag{3.4}$$

where $c_1 \in (0,1)$ is some small constant. A value of α that satisfies the above equation generates a sufficient descent.

The second condition is the *curvature condition* which states

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k \tag{3.5}$$

where $c_2 \in (c_1, 1)$ is some constant. The *Armijo* and the *curvature conditions* are collectively called the *Wolfe conditions* [74]. In the context of neural network optimisation, the x's refer to the weights of the network.

In our proposed algorithm, the *Armijo* condition is adopted as a condition to determine whether a sufficient descent is achieved by a new search direction. It is also used to judge whether the step-size used is sufficiently large. When the *Armijo* condition is satisfied, the step-size increases, allowing a more aggressive search. When the condition is not satisfied, the step-size decreases.

3.4 An Outline of Second-order and Quasi-Newton Optimisation Methods

In Chapter 2, I have discussed the key classifications of neural network training algorithms. Under such classification, the current widely-adopted optimisers are usually first-order and heuristic in nature. However, several pieces of current research target at quasi-Newton deterministic methods to perform optimisation. These methods are not yet directly applied to machine learning problems, but are used in non-linear control problems. However, they demonstrate promises in becoming the state-of-art optimisation algorithms for deep neural networks. The shooting algorithm[165][166], for example, has been applied in control

problems and is second-order in nature. This second-order nature that requires the storage of the Hessian matrix has been modified by different methods of Hessian approximation, generating quasi-Newton algorithms. The *iterative Linear-Quadratic Regulator (iLQR)* method is one of the many ways to approximate a Hessian matrix. Alternatively, Gauss-Newton Hessian approximation has been used in [166] to improve the solution to the nonlinear optimal control problem. Although these algorithms are effective solvers in the optimal control problems, they require much more modifications to be applied to large-scale datasets with a complicated objective function to optimise.

Within the machine learning community, the most popular quasi-Newton method is the L-BFGS method. More specifically, L-BFGS is a great improvement from its predecessor BFGS by using limited amount of computer memory. Unlike BGFS that stores a dense $n \times n$ matrix to approximate the inverse of Hessian, L-BFGS stores a few vectors that implicitly approximate the inverse of Hessian matrix. It retains a limited amount of past gradient and hence is effective in memory storage. This characteristic makes it useful in being applied to large datasets [167]. However, L-BFGS has its own limitations. Although it is highly advantageous in its search speed, it works well with only simple variable bounds and may sometimes misbehave in deep structures [168]. Recent research seeks to combine L-BFGS with stochastic methods to achieve a linear convergence rate [169] [170]. It has been applied with some success in multiple convex and non-convex problems with different step sizes. However, it is has only been experimented on case studies without full mathematical proof.

Although there have been several pieces of research work that investigate second-order methods, these are not the mainstream research direction and the proposed methods often have problems with regard to memory storage and computing power. However, with the developments in computer memory and computing power, it is expected that such limitations will be less of a problem in the near future [171].

A few quasi-Newton deterministic methods are widely used in optimisation. I introduce three of those which also form the basis of our proposed optimiser: the Conjugate Gradient Method, the Truncated Newton Method and the Brown and Bartholomew-Biggs' Method.

3.4.1 Conjugate Gradient Method

The Conjugate Gradient (CG) method has the advantage of using only first-order derivatives but overcomes the limitation of slow convergence rate as opposed to the Gradient Descent method [172]. Like L-BFGS, it does not require the storage of the full Hessian matrix. The method makes use of the gradient at the current iteration, finds its conjugate direction, and searches along this direction for the minimum point. Depending on the problem, it can approach the solution more uniformly with a faster convergence rate compared to Stochastic

Gradient Descent. The solution to a linear equation $Ax = b$ is shown below:

$$\alpha_k = \frac{r_k^T r_k}{d_k^T A d_k} \tag{3.6}$$

$$x_{k+1} = x_k + \alpha_k d_k \tag{3.7}$$

$$r_{k+1} = r_k + \alpha_k A d_k \tag{3.8}$$

$$\beta_{k+1} = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \tag{3.9}$$

$$d_{k+1} = -r_{k+1} + \beta_{k+1} d_k \tag{3.10}$$

where d_k is the conjugate search direction, α and β are coefficients and r_k is a residual of $r_k = b - Ax_k$. The method has been extended to the training of neural networks and has been improved: Fletcher-Reeves' method based on CG modifies the calculation of α_k and r_k [173], whereas Polak-Ribiere's method modifies the β_k [174]. The CG method is simple and memory-effective, therefore it has been adopted by the machine learning community. The method also guarantees convergence within n iterations for a problem of dimension n [175].

3.4.2 Truncated Newton Method

Another popular optimisation method is the Truncated Newton (TN) Method. The method consists of an outer loop and an inner loop. The outer loop solves the optimisation problem and the inner loop computes the update at each iteration by estimating the solution to Newton's Equation. The inner loop is truncated before a Newton solution is reached and often adopts the CG method to solve for the search direction. The method has been shown to be suitable for solving large-scale unconstrained optimisation problems [176], which dictates its applicability to solve ANN problems. It has been adopted by the machine learning community to train neural networks [177].

3.4.3 Brown and Bartholomew-Biggs' Method

The Gradient Flow method calculates the solution of an unconstrained optimisation problem by solving the ODE:

$$\frac{dx(t)}{dt} = -\nabla_x f(x) \tag{3.11}$$

In 1989, Brown and Bartholomew-Biggs proposed an improvement to the Gradient Flow algorithm that forms the basis of our proposed algorithm ([178]). It is built upon the Gradient

Descent method and the Newton's method. The search direction is calculated as follows:

$$\left[H_k + \frac{1}{h_k} I\right] \cdot p_k = -g_k \tag{3.12}$$

where H_k is the Hessian matrix, h_k is the step-size, I is the identity matrix, g_k is the gradient and p_k is the update of $x_{k+1} = x_k + p_k$. It can be observed from the equation that as the step size increases to infinity, the method tends to the second-order Newton's method, and as the step size decreases to zero, the method becomes Gradient Descent method. Constant adjustment of the step sizes occur throughout the process of optimisation and one of the strategy, which is adopted as the default in our proposed algorithm, is to double the step size when a reduction in objective function is achieved and half the step size when a reduction is not achieved.

3.5 A Novel Quasi-Newton Method

This section proposes a novel quasi-Newton method, Hessian-free Gradient Flow (HFGF), for the optimisation of deep neural networks. The method adopts approximated second-order information based on finite difference and is developed from the Gradient Flow method and Truncated Newton method. The descent condition makes use of the Amijo condition. The performance of the algorithm is first tested on standard testing functions and then applied to neural network settings.

3.5.1 Algorithm Development

I propose a method that adopts approximated second-order information to perform an optimisation task. To derive the optimisation method, I first write the general update rule as follows:

$$x_{k+1} = x_k + \Delta t \cdot \Delta x_k. \tag{3.13}$$

The gradient descent method uses the negative gradient vector as search direction:

$$x_{k+1} = x_k - \Delta t \cdot \nabla f(x_k). \tag{3.14}$$

The limit $\Delta t \to 0$ gives the smooth trajectory of the gradient flow method:

$$\frac{dx}{dt} = -\nabla f(x_k). \tag{3.15}$$

Linearizing the right-hand side, I obtain:

$$\frac{dx}{dt} \approx -[\nabla f(x_k) + \nabla^2 f(x_k) \cdot (x_{k+1} - x_k)]. \tag{3.16}$$

Rewriting the gradient vector and the Hessian matrix as g_k and H_k, and applying a linearly implicit Euler scheme gives:

$$x_{k+1} - x_k = -\Delta t \cdot [g_k + H_k \cdot (x_{k+1} - x_k)]. \tag{3.17}$$

The step length Δt_k is replaced with the step-size h_k and Equation 3.17 is rearranged:

$$\left[H_k + \frac{1}{h_k} \cdot I\right](x_{k+1} - x_k) = -g_k. \tag{3.18}$$

The search direction is equal to Δx_k which takes the form p_k, similar to Newton's equation. To give Brown and Bartholomew-Bigg's equation [178] and the iteration update for x_{k+1}:

$$\begin{aligned} \left[H_k + \frac{1}{h_k} \cdot I\right] p_k &= -g_k, \\ x_{k+1} &= x_k + p_k. \end{aligned} \tag{3.19}$$

This equation can be rewritten in the form of a linear equation, where the Hessian matrix and the product of step-size and identity matrix have been grouped together into a new matrix, Q_k:

$$\begin{aligned} Q_k &= H_k + \frac{1}{h_k} \cdot I, \\ Q_k \cdot p_k &= -g_k. \end{aligned} \tag{3.20}$$

To solve for this equation, I follow a schema that is similar to the truncated Newton method, where the outer loop solves for the optimisation problem and the inner loop solves for the search direction p_k [179]. Instead of solving for $Ax = b$ as in Section 3.4.1, I iteratively solve for $Qp = -g$ using the Conjugate Gradient algorithm. The inner loop does not solve for the exact solution. Instead, the number of iterations is truncated, *i.e.* I stop after a finite number of iterations. This reduces the computational complexity of each iteration, generating an algorithm that converges faster to the optimum.

The combination of the truncated Newton and gradient flow methods creates a large-scale solution method for unconstrained optimisation problem with improved properties. These properties include better navigation of non-convex regions through the substitution

of Newton's equation with the gradient flow equation. The manipulation of step-size (h_k) within the gradient flow method also supports a faster and more accurate convergence than Newton's method. This should result in better convergence qualities for the truncated Newton method when combined with gradient flow.

Another important improvement of the algorithm is that it is Hessian-free. The product of the Hessian matrix and the conjugate direction vector product is approximated using finite difference such that the algorithm only evaluates first-order derivatives. The scheme is outlined below in the vector finite difference equation (Equation 3.18), where ε is a user-defined small number to approximate an infinitesimal value.

$$Q_k p_i = \left[\frac{\nabla f(x_k + \varepsilon p_i) - \nabla f(x_k)}{\varepsilon}\right] + \frac{1}{h_k} I p_i. \tag{3.21}$$

3.5.2 Convergence Criteria

The convergence criterion for the outer loop is the infinity norm of the gradient vector resulted from each iteration:

$$\|g_k\|_\infty = \max(|g_1|, ..., |g_n|) \tag{3.22}$$

$$\|g_k\|_\infty < tol \tag{3.23}$$

where tol is a user-defined tolerance parameter that is adapted depending on the problem to solve. Typical values of tolerance are between $10^{-5} - 10^{-9}$.

If this is satisfied, the step size is increased. If not and the maximum CG iterations is reached, the step size is decreased. The total number of CG iterations, theoretically the dimension of the input n, is set as a hyperparameter in the HFGF method. This is because empirical investigation has demonstrated that in large-scale problems, optimality can be reached without arriving at n iterations. The default number of iterations is set at 15 since convergence is usually achieved by this number.

3.5.3 Algorithm Definition

The proposed HFGF algorithm is outlined in Algorithm 1. The value of h_0 is initialised to any value between 0 and 1, and p_0 is the zero vector commonly used in truncated Netwon methods. From the pseudocode, it is observable that the inner loop is effectively a CG iteration and the values of $Q_k p_k$ and $Q_k d_i$ are solved by finite differences.

Algorithm 1 HFGF Algorithm

Initialise: $x_0, p_0 = \overrightarrow{0}$ and $h_0 = 10^{-3}$
while $\|g_k\| > tol$ **do**
  $Q_k p_k = \frac{g(x_k+\varepsilon p_k)-g(x_k)}{\varepsilon} + \frac{1}{h_k} I \cdot p_k$
  Initialise: $r_0 = Q_k p_k + g_k$, $d_0 = -r_0$ and $p_i = p_k$
  for $i = 0, 1, 2, ..., i_{max}$ **do**
    $Q_k d_i = \frac{g(x_k+\varepsilon d_i)-g(x_k)}{\varepsilon} + \frac{1}{h_k} I \cdot d_i$
    $\alpha_i = \frac{r_i^T r_i}{d_i^T Q_k d_i}$
    $p_{i+1} = p_i + \alpha_i d_i$
    $r_{i+1} = r_i + \alpha_i Q_k d_i$
    $\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
    $d_{i+1} = -r_{i+1} + \beta_{i+1} d_i$
    $x_{try} = x_k + p_{i+1}$
    if $f(x_{try}) < f(x_k)$ **then**
      if $f(x_{try}) < f(x_k) + \mu(g_k^T p_k)$ [Armijo First Order Conditions (of descent)] **then**
        $x_{k+1} = x_{try}$ and $p_{k+1} = p_{i+1}$
        $h_{k+1} = 2h_k$
        break [Minor iteration]
      else if $i = i_{max}$ **then**
        $x_{k+1} = x_{try}$ and $p_{k+1} = p_{i+1}$
        $h_{k+1} = \frac{1}{2} h_k$
        break [Minor iteration]
      end if
    else if $i = i_{max}$ **then**
      $h_{k+1} = \frac{1}{2} h_k$ and x_{try} not accepted
      break [Minor iteration]
    end if
  end for
end while

3.5.4 Convergence Analysis

Although the algorithm has been developed in theory, it is important to state its convergence property. This section provides the proof of convergence.

Training ANNs and DNNs can be viewed as the equivalent to minimizing a large-scale optimisation problem of the form:

$$\min f(x), \tag{3.24}$$

where $x \in \mathbb{R}^n$ is a real valued n-dimensional vector of system variables that are to be optimised to minimize the scalar function $f(x) : \mathbb{R}^n \to \mathbb{R}$. I will adopt the inexact Newton Method to derive the proof of convergence.

In this paper, I assume that f has an optimal value $f(x^*)$ at x^*. I will use the following assumption about the objective function for the rest of this article.

Assumption 1 *Assume that f is L-smooth, that is, f is differentiable and the gradient is L-Lipschitz continuous, i.e., $\forall x, y \in \mathbb{R}^n$, $\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$.*

Theorem 1 *Consider HFGF algorithm (equation [3.18]). Under the convex assumption (Assumption 1), when $0 < h_k \leq \frac{1}{L}$, I have*

$$\|x_k - x^*\|^2 \leq \tau^k \|x_0 - x^*\|^2, \tag{3.25}$$

where τ is a constant satisfy that $1 - \frac{1}{L} \leq \tau < 1$. Furthermore,

$$\|f(x(t)) - f(x^*)\| \leq \frac{L}{2}\tau^k \|x_0 - x^*\|^2,$$

holds.

Proof 1 *Using a finite difference scheme to approximate Hessian vector-product, there exists $\varepsilon > 0$, such that:*

$$H_k p_k = \frac{\nabla f(x_k + \varepsilon p_k) - \nabla f(x_k)}{\varepsilon} + \frac{1}{h_k} I p_k. \tag{3.26}$$

Then I get the iterate, as following:

$$\begin{aligned} -\nabla f(x_k) &= \frac{\nabla f(x_k + \varepsilon p_k) - \nabla f(x_k)}{\varepsilon} + \frac{1}{h_k} I p_k, \\ x_{k+1} &= x_k + p_k. \end{aligned} \tag{3.27}$$

Let $\tilde{x}_{k+1} := x_k + \varepsilon p_k$ *and* $p_k = x_{k+1} - x_k$*, then rewrite Equation 3.27 as:*

$$\begin{aligned} x_{k+1} &= x_k - h_k \nabla f(x_k) - \frac{h_k}{\varepsilon}\big(\nabla f(\tilde{x}_{k+1}) - \nabla f(x_k)\big), \\ \tilde{x}_{k+1} &= x_k + \varepsilon(x_{k+1} - x_k). \end{aligned} \tag{3.28}$$

By substituting the second equation in Equation 3.28 into the first equation, the following implicit iterative form can be obtained:

$$x_{k+1} = x_k - h_k \nabla f(x_k) - \frac{h_k}{\varepsilon}\big(\nabla f(x_k + \varepsilon(x_{k+1} - x_k)) - \nabla f(x_k)\big). \tag{3.29}$$

According to the hypothesis, function f *is second-order differentiable, and Hessian matrix* $\nabla^2 f$ *is positive definite. The first-order Taylor expansion of* $\nabla f(x_k + \varepsilon(x_{k+1} - x_k))$ *in the above equation at* x_k *can be obtained as follows:*

$$x_{k+1} = x_k - h_k \nabla f(x_k) - \frac{h_k}{\varepsilon}\Big(\nabla f(x_k) + \nabla^2 f(z_k) \cdot \varepsilon(x_{k+1} - x_k) - \nabla f(x_k)\Big), \tag{3.30}$$

where $z_k \in (x_k, x_k + \varepsilon(x_{k+1} - x_k))$*, i.e., there exists some* $\delta \in (0,1)$ *such that* $z_k = x_k + \varepsilon \cdot \delta(x_{k+1} - x_k)$*. Rewrite Equation 3.30 as follows:*

$$x_{k+1} + h_k \nabla^2 f(z_k)(x_{k+1} - x_k) = x_k - h_k \nabla f(x_k), \tag{3.31}$$

then:

$$\big\|x_{k+1} - x^* + h_k \nabla^2 f(z_k)(x_{k+1} - x_k)\big\|^2 = \big\|x_k - x^* - h_k \nabla f(x_k)\big\|^2. \tag{3.32}$$

For the left side of Equation 3.32:

$$\begin{aligned} & \big\|x_{k+1} - x^* + h_k \nabla^2 f(z_k)(x_{k+1} - x_k)\big\|^2 \\ = \;& \|x_{k+1} - x^*\|^2 + h_k^2 \big\|\nabla^2 f(z_k)\big\|^2 \|x_{k+1} - x_k\|^2 + 2h_k \big\langle x_{k+1} - x^*, \nabla^2 f(z_k)(x_{k+1} - x_k)\big\rangle \\ \geq \;& \|x_{k+1} - x^*\|^2 + h_k^2 \big\|\nabla^2 f(z_k)\big\|^2 \|x_{k+1} - x_k\|^2 \\ & + 2h_k \|x_{k+1} - x^*\| \cdot \big\|\nabla^2 f(z_k)\big\| \cdot \|x_{k+1} - x_k\|. \end{aligned} \tag{3.33}$$

Then:

$$\big\|x_{k+1} - x^* + h_k \nabla^2 f(z_k)(x_{k+1} - x_k)\big\|^2 \geq \|x_{k+1} - x^*\|^2. \tag{3.34}$$

For the right side of Equation 3.32:

$$\begin{aligned} & \left\|x_k - x^* - h_k \nabla f(x_k)\right\|^2 \\ = \; & \|x_k - x^*\|^2 + h_k^2 \left\|\nabla f(x_k)\right\|^2 - 2h_k \left\langle x_k - x^*, \nabla f(x_k)\right\rangle \\ \leq \; & \|x_k - x^*\|^2 - h_k \left(\frac{2}{L} - h_k\right) \left\|\nabla f(x_k)\right\|^2, \end{aligned} \tag{3.35}$$

where I have used the following inequality:

$$\frac{1}{L}\|\nabla f(x) - \nabla f(y)\|^2 \leq \left\langle x - y, \nabla f(x) - \nabla f(y)\right\rangle,$$

for all $x, y \in R^n$, and $\nabla f(x^*) = 0$.

According to Equation 3.32, the following inequality can be obtained from Equation 3.33 and Equation 3.35:

$$\begin{aligned} \|x_{k+1} - x^*\|^2 &\leq \|x_k - x^*\|^2 - h_k \left(\frac{2}{L} - h_k\right) \left\|\nabla f(x_k)\right\|^2 \\ &\leq \|x_k - x^*\|^2 - h_k \left(\frac{2}{L} - h_k\right) \cdot L \|x_k - x^*\|^2 \\ &\leq \left(1 - h_k(2 - h_k L)\right) \|x_k - x^*\|^2. \end{aligned} \tag{3.36}$$

Thus when $0 \leq h_k \leq \dfrac{1}{L}$, there exist some $1 - \frac{1}{L} \leq \tau < 1$, such that

$$\|x_{k+1} - x^*\|^2 \leq \tau \|x_k - x^*\|^2 \leq \tau^k \|x_0 - x^*\|^2.$$

Furthermore, I arrive at the simplest analysis: $f(x_k) - f(x^*) \leq \frac{L}{2}\|x_k - x^*\|^2 \leq \frac{L}{2}\tau^k \|x_0 - x^*\|^2$.

3.6 Test of Optimisation Algorithms

Optimisation algorithms developed in theory should be tested for speed, accuracy, and scalability in practice. The convention is to first test the performance of the optimisation algorithms on a set of functions that are specifically designed to undertake this task. These functions are called standard testing functions.

Standard testing functions have designed topological complexity that mimics the highly non-convex nature of the topology of an ANN/DNN objective function, and are often treated as benchmarks for evaluating the performance of optimisation algorithms. Therefore, the HFGF method is first experimented with the standard functions described in literature concerning optimisation. The testing functions are obtained from [180]. Table 3.1 divides the

testing functions used in this research based on their characteristics. For the purpose of this report, the focus will be on multi-dimensional functions, as the form of the neural networks are rarely two-dimensional. This is especially the case for deep neural networks, where the dimensions are orders of millions in industrial applications.

Table 3.1 Standard testing function types and details

Test Functions	Types of Function	Types of Mimima	Reference
1. Rosenbrock, 2. Chung Reynolds, 3. De Jong	Unimodal, convex, multidimensional	Global	Jamil & Yang [181], Molga & Smutnicki [180]
4. Rastrigin, 5. Ackley	Multimodal, multidimensional	Many local & Global	Jamil & Yang [181], Molga & Smutnicki [180]
6. Booth	Unimodal, convex, two-dimensional	Global	Jamil & Yang [181]
7. Dropwave, 8. Shubert	Multimodal, two-dimensional	Many local & Global	Molga & Smutnicki [180]

In this thesis, I make use of a few standard testing functions to test the performance of our proposed algorithm. For the purpose of completeness, I briefly describe these algorithms below.

3.6.1 Rosenbrock Function

In [180]'s paper, higher order Rosenbrock functions are treated as unimodal functions. In the 2-dimensional case, the global minimum (1,1) is inside a long, narrow, parabolic shaped flat valley. The graph of the function is shown in Figure 3.1.

The function has also been extended to higher orders and have been shown to have more than one minimum by [182]. The multi-dimensional form of the Rosenbrock function is shown in Equation 3.37:

$$f(x) = \sum_{i=1}^{N-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \tag{3.37}$$

where $x = [x_1, ..., x_N] \in \mathbb{R}^N$. The minimum of this function is located at $x_i = 1$, for $i \in [1, ..., N]$.

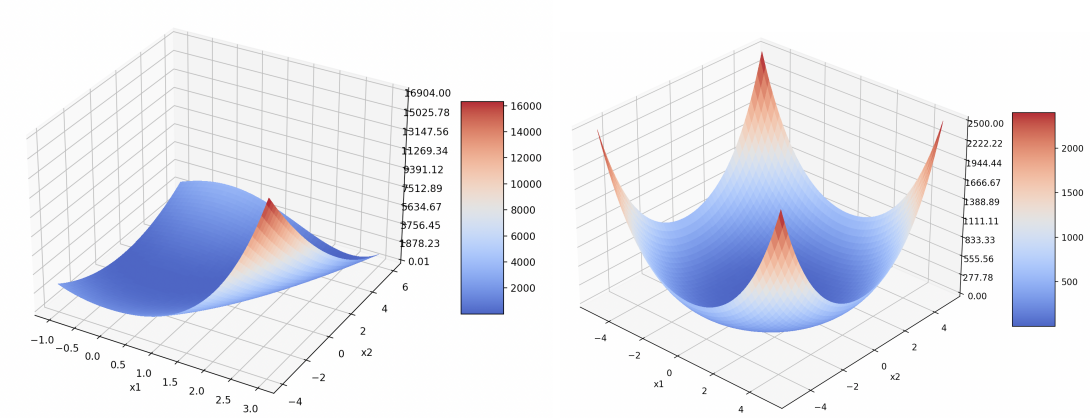


Figure 3.1 The shape of a 2-dimensional (a) Rosenbrock function with a global minimum at (1,1), and (b) Chung Reynolds function with a global minimum at (0,0)

3.6.2 Chung Reynolds Function

The mathematical expression of a Chung Reynolds function is shown in Equation 3.38. It has a minimum value of 0 occurring at $x_i = 0$. A graph of the Chung Reynolds function is shown in Figure 3.1.

$$f(x) = (\sum_{i=1}^{N} x_i^2)^2 \tag{3.38}$$

3.6.3 De Jong Function

The mathematical definition of one of the simplest test function, De Jong function, is shown in Equation 3.39, with a global minimum at $x_i = 0$ with a function value of 0. Its graph is shown in Figure 3.2.

$$f(x) = \sum_{i=1}^{N} x_i^2 \tag{3.39}$$

3.6.4 Rastrigin Function

The Rastrigin function is a non-convex function with a large search space and a large number of local minima. The two-dimensional form is shown in Figure 3.3. The mathematical expression is shown in Equation 3.40.

$$f(x) = 10N + \sum_{i=1}^{N} [x_i^2 - 10\cos(2\pi x_i)] \tag{3.40}$$

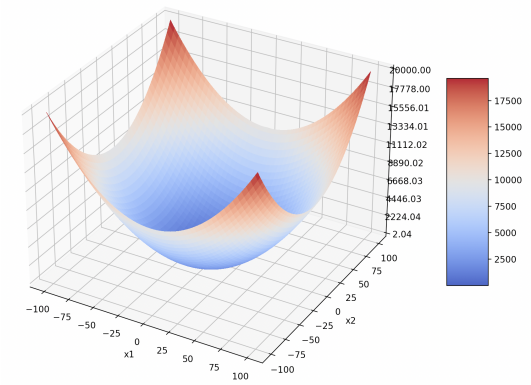


Figure 3.2 The shape of a 2-dimensional De Jong function with global minimum at (0,0)

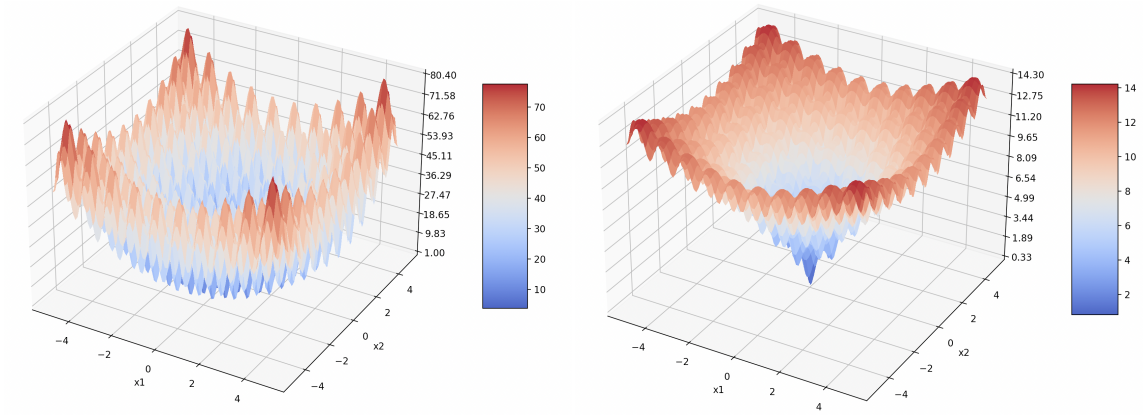


Figure 3.3 A graph of the two-dimensional form of the (a) Rastrigin function with a global minimum at (0, 0) and (b) Ackley function with a global minimum at (0,0)

The range of x_i is [-5.12,5.12]. The function has a global minimum at $x_i = 0$ with a minimum function value of 0.

3.6.5 Ackley Function

The Ackley function is one of the most difficult-to-optimise function with many local minima. The two-dimensional form is shown in Figure 3.3. It has a nearly flat outer region full of local minima with a hole in the center. The mathematical form is shown in Equation 3.41.

$$f(x) = -20exp\left(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N}x_i^2}\right) - exp\left(\frac{1}{d}\sum_{i=1}^{N}\cos(2\pi x_i)\right) + 20 + exp(1) \qquad (3.41)$$

The function has a global minimum at $x_i = 0$ with a function value of 0. The search domain is $x_i \in [-32.768, 32.768]$.

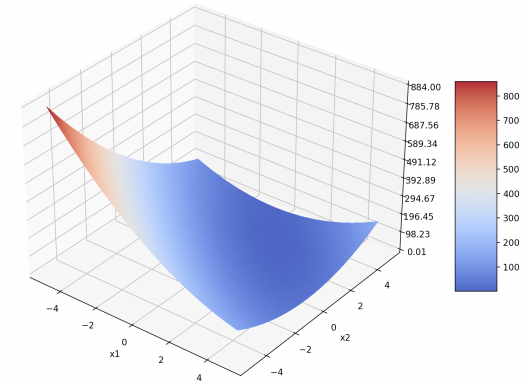


Figure 3.4 The shape of 2-dimensional Booth function with global minimum at (1,3)

3.6.6 Booth Function

The mathematical form is demonstrated in Equation 3.42 and the graphical form is shown in Figure 3.4.

$$f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \tag{3.42}$$

The function has a global minimum at $x_1 = 1$, $x_2 = 3$ with a function value of 0.

3.6.7 Dropwave Function

The mathematical form is demonstrated in Equation 3.43 and the graphical form is shown in Figure 3.5.

$$f(x) = -\frac{1 + \cos\left(12\sqrt{x_1^2 + x_2^2}\right)}{\frac{1}{2}\left(x_1^2 + x_2^2\right) + 2} \tag{3.43}$$

The function has a global minimum at $x_1 = 0$, $x_2 = 0$ with a function value of -1. The search domain is $x_i \in [-5.12, 5.12]$.

3.6.8 Shubert Function

The mathematical form is demonstrated in Equation 3.44 and the graphical form is shown in Figure 3.5.

$$f(x) = \left(\sum_{i=1}^{5} i\cos((i+1)x_1 + i)\right)\left(\sum_{i=1}^{5} i\cos((i+1)x_2 + i)\right) \tag{3.44}$$

The function has many global minima such as $(-0.8121, -0.8121)$ with a minimum function value of -186.7309. The search domain is $x_i \in [-5.12, 5.12]$.

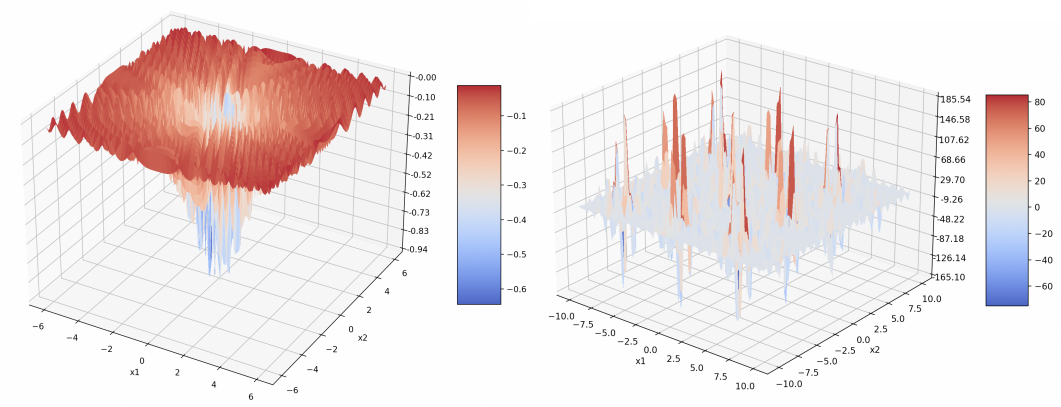


Figure 3.5 A graph of the two-dimensional form of the (a) Dropwave function with a global minimum at (0, 0) and (b) Shubert function with many global minima

3.7 Analysis of the HFGF Algorithm

This section undertakes the analysis of the cost and performance of the HFGF algorithm. To test the performance, I introduce several testing functions that is highly complex and non-convex in nature. These functions are introduced to simulate the highly complicated shape of the objective function of neural networks. Unless specifically mentioned, the results are generated using a 2.3 GHz Intel Core i5 processor with a memory of 8 GB 2133 MHz LPDDR3.

The HFGF algorithm has been coded using Python. In cases where standard optimisers are used, the optimisers are coded in C. It is understood that the speed of C outperforms that of Python. However, I observe the speed advantages of HFGF even in Python. For future work, coding HFGF in Cython can be an optioin to speed up the optimiser performance.

3.7.1 Performance on Standard Testing Functions

I test the performance of the optimisation algorithm on standard testing functions. The functions range from the simplest case of a 2-dimensional unimodal convex function to the more complex case of a multidimensional, multimodal function. The latter is the focus since I expect the objective function of a neural network to be multimodal and multidimensional.

Two-Dimensional Test Functions

The test on 2-dimensional functions serves as a testament to the convergence behaviour of the optimisers. By applying the optimiser on two-dimensional functions I observe that the proposed HFGF method outperforms other traditional optimisers including Truncated

Newton, Newton Conjugate Gradient, and L-BFGS. The results are summarised in Table 3.2. These results are generated from 10 independent runs in each equation.

Table 3.2 Results of performing different classic second-order optimisation algorithms on 2-D Test Functions. The algorithms include Truncated Newton method (TN), Nonlinear Conjugate Gradient (NCG), L-BFGS method and the proposed HFGF method. $f(x)_{final}$ is the final function value obtained after optimisation. n_{iter} is the total number of iterations in the major loop. n_{fev} is the number of function evaluations. n_{gev} is the number of gradient evaluations. $Convergence[\%]$ is the rate of success in optimising each function.

Function	Optimisation Method	$f(x)_{final}$	n_{iter}	n_{fev}	n_{gev}	$Convergence[\%]$
Dropwave	TN	-2.92×10^{-1}	4	17	8	100
	NCG	-2.08×10^{-1}	3	41	9	67
	L-BFGS	-3.23×10^{-1}	3	28	6	67
	HFGF	-1.00×10^{0}	2	32	34	100
Shubert	TN	5.80×10^{-17}	4	13	7	100
	NCG	2.13×10^{-2}	3	39	8	50
	L-BFGS	3.33×10^{-15}	5	25	9	50
	HFGF	2.20×10^{-11}	8	24	32	100
Booth	TN	2.34×10^{-1}	7	21	15	100
	NCG	1.04×10^{-16}	2	20	5	100
	L-BFGS	1.23×10^{-12}	5	20	10	100
	HFGF	1.18×10^{-11}	15	45	60	100

The two-dimensional test functions have been used to test the basic convergence ability of the proposed method. The CPU time required for each method is negligible for all 2-D test cases. The Dropwave and Shubert function require the optimisers to navigate through a highly non-convex region to find the global minima. At the global minima, the Dropwave and Shubert functions have values of $f(x^*) = -1$ and $f(x^*) = 0$ respectively. From the Table 3.2, it is clear that the proposed method performed the best as it was the only method that converged to the global minima with a high success rate.

Multidimensional Test Functions

I perform analysis on the multidimensional testing functions, both unimodal and multimodal. High dimensions of 100,000 and 1,000,000 are selected to simulate the high dimension of data I input to a deep neural network. The results are displayed in Table 3.3–3.4. At this scale, the other optimisers calculated using SciPy are not able to converge in many cases as shown in the next section, except when initialisation of x values are very close to the global optimum.

CPU times extend to many hours and in some cases require days to achieve convergence. Therefore, the values are not reported. The HFGF method is able to handle problems ranging from 50,000 to 200,000 dimensions with less than 5 minutes of CPU times in most cases. This clearly illustrates the strength of HFGF algorithm in large-scale applications compared to NCG, L-BFGS, and TN methods. The final objective function values for the Ackley function are comparably high due to the highly complicated and non-linear shape of the function, but the convergence criteria set for HFGF have been reached when convergence is claimed.

Table 3.3 HFGF results for 100,000-dimensional test functions.

$Function$	$f(x)_{final}$	n_{iter}	n_{fev}	n_{gev}	$CPUtime(s)$	$Convergence[\%]$
Rosenbrock	1.40×10^{-9}	409	105	514	743.47	100
Dejong	6.03×10^{-6}	75	25	100	38.16	100
Chung Reynolds	2.49×10^{-2}	58	19	77	50.77	100
Rastrigin	1.16×10^{6}	22	7	29	25.84	100
Ackley	2.272×10^{1}	22	72	80	72.07	100

Table 3.4 HFGF results for 1,000,000-dimensional test functions.

$Function$	$f(x)_{final}$	n_{iter}	n_{fev}	n_{gev}	$CPUtime(s)$	$Convergence[\%]$
Rosenbrock	2.86×10^{-9}	399	110	509	6112.24	100
Dejong	6.03×10^{-5}	75	25	100	418.40	100
Chung Reynolds	2.98×10^{-1}	69	23	92	543.41	100
Rastrigin	1.16×10^{7}	32	11	43	368.37	100
Ackley	6.560×10^{0}	27	74	81	702.19	100

Comparison with Other Function Optimisers

I have performed a comparison between different quasi-Newton algorithms. The results are tabulated in Table 3.5 for the dimension of 10,000 and in Table 3.6 for the dimension of 100,000. The optimisers of Truncated Newton, Newton Conjugate Gradient and L-BFGS are coded in SciPy [183] and HFGF is coded using the package Casadi [184]. I have only included comparisons for Dejong, Chung Reynolds and Rastrigin functions for the dimensionality of 100,000 because the optimisers other than HFGF fail to converge within a time frame of 100,000 seconds. The operating system used to run the code is Ubuntu 18.04.1

Table 3.5 HFGF results for 10,000-dimensional test functions in comparison to other optimisers of TN, NCG and L-BFGS.

Function	Method	$f(x)_{final}$	n_{iter}	n_{fev}	n_{gev}	CPU Time	Success
Rosenbrock	TN	7.54×10^{3}	433	9219	-	59932.24	No
	NCG	3.99×10^{0}	183	2.83×10^{6}	283	2215.28	Yes
	L-BFGS	6.49×10^{4}	1	2.00×10^{4}	-	13.122	No
	HFGF	2.31×10^{-8}	62	308	278	17.811	Yes
Dejong	TN	1.01×10^{-7}	4	18	-	118.22	Yes
	NCG	2.90×10^{-6}	1	5.00×10^{4}	5	30.64	Yes
	L-BFGS	8.14×10^{2}	1	4.00×10^{4}	-	25.24	No
	HFGF	1.69×10^{-5}	14	28	42	0.84	Yes
Chung Reynolds	TN	2.02×10^{-6}	3	15	-	101.49	Yes
	NCG	6.87×10^{5}	1	1.60×10^{5}	16	101.49	No
	L-BFGS	4.10×10^{6}	1	3.00×10^{4}	-	18.62	No
	HFGF	3.62×10^{-3}	14	28	42	1.13	Yes
Rastrigin	TN	5.97×10^{-6}	4	17	-	121.04	Yes
	NCG	6.42×10^{-4}	1	8.00×10^{4}	8	53.79	Yes
	L-BFGS	2.29×10^{-4}	1	8.00×10^{4}	-	55.22	Yes
	HFGF	4.15×10^{-6}	16	66	43	2.29	Yes
Ackley	TN	6.02×10^{-9}	4	53	-	7387.56	Yes
	NCG	4.25×10^{0}	0	1.00×10^{4}	1	128.38	No
	L-BFGS	4.25×10^{0}	0	1.000×10^{4}	-	127.40	No
	HFGF	2.27×10^{-2}	22	72	80	3.98	Yes

LTS and the CPU used is Intel Core i7-8700K with a memory of 3.70GHz. A few points I have noted from the data:

- The time of convergence for HFGF is much faster compared to other quasi-Newton method.
- HFGF converges successfully in all cases.
- Optimisers other than HFGF succeed and fail in different cases, well demonstrating the "No Free Lunch" Theorem.
- The optimiser L-BFGS fails in most cases, indicating its lack of robustness.

Table 3.6 HFGF results for 100,000-dimensional test functions in comparison to other optimisers of TN, NCG and L-BFGS. Only HFGF results are included for Rosenbrock and Ackley functions because the other optimisers fail to converge within 100,000 seconds of CPU time.

Function	Method	$f(x)_{final}$	n_{iter}	n_{fev}	n_{gev}	CPU Time	Success
Rosenbrock	HFGF	1.40×10^{-9}	409	105	514	743.47	Yes
Dejong	TN	3.35×10^{-12}	13	83	-	53195.35	Yes
	NCG	9.33×10^{-3}	1	6.00×10^{5}	6	9379.80	Yes
	L-BFGS	1.88×10^{4}	1	4.00×10^{5}	-	2678.09	No
	HFGF	6.03×10^{-4}	75	25	100	38.16	Yes
Chung Reynolds	TN	1.95×10^{-4}	12	101	-	66195.31	Yes
	NCG	2.72×10^{7}	1	1.70×10^{6}	17	11539.14	No
	L-BFGS	3.53×10^{8}	1	4.00×10^{5}	-	2688.95	No
	HFGF	2.49×10^{-2}	58	19	77	50.77	Yes
Rastrigin	TN	1.90×10^{-3}	5	28	-	24269.53	Yes
	NCG	1.98×10^{-4}	1	1.10×10^{6}	11	9795.51	Yes
	L-BFGS	3.96×10^{1}	1	1.00×10^{6}	-	8185.25	No
	HFGF	1.16×10^{6}	22	7	29	25.84	Yes
Ackley	HFGF	2.27×10^{1}	22	72	80	72.07	Yes

3.7.2 Time and Memory Analysis

Time Analysis

The HFGF method involves both function evaluations and gradient evaluations. I perform CPU time analysis to identify the most time-consuming steps in the optimiser. Table 3.7 tabulates the CPU time spent on each type of evaluation based on 10 different initial points. The optimiser is run to solve for the optima of a 100-dimensional Rosenbrock function. The average CPU time for one step of function evaluation is 3.97×10^{-5} seconds and the average CPU time for one step of gradient evaluation is 4.16×10^{-4} seconds. In Table 3.7, f represents the final function value, t is the total time spent; n_{ev} is the total number of evaluations; n_{fev} is the number of function evaluations; n_{gev} is the number of gradient evaluations; t_{fev} is the total time spent on function evaluations; t_{gev} is the total time spent on gradient evaluations; and t_{other} is the total time spent on the rest of the evaluations.

From Table 3.7, it is evident that the most CPU time is spent on gradient evaluations. This has demonstrated the importance of selecting an automatic differentiation package that

Table 3.7 CPU time spent on different optimiser steps

Run	f	t	n_{ev}		t_{fev}		t_{gev}		t_{other}	
			n_{fev}	n_{gev}	Time	%	Time	%	Time	%
1	2.35×10^{-5}	0.0274	56	42	0.00200	7.30	0.0169	61.68	0.0085	31.02
2	2.58×10^{-5}	0.0341	56	42	0.00284	8.33	0.0209	61.29	0.0104	30.38
3	7.44×10^{-5}	0.0270	60	45	0.00201	7.44	0.0164	60.74	0.0086	31.81
4	5.41×10^{-5}	0.0333	60	45	0.00245	7.36	0.0209	62.76	0.0100	29.88
5	4.23×10^{-5}	0.0283	59	44	0.00203	7.17	0.0177	62.54	0.0086	30.28
6	3.75×10^{-5}	0.0317	67	50	0.00251	7.92	0.0196	61.83	0.0096	30.25
7	1.32×10^{-5}	0.0277	59	44	0.00237	8.56	0.0168	60.56	0.0085	30.79
8	7.72×10^{-5}	0.0322	64	48	0.00248	7.70	0.0196	60.87	0.0101	31.43
9	4.18×10^{-5}	0.0262	56	42	0.00208	7.94	0.0155	59.16	0.0086	32.90
10	5.41×10^{-5}	0.0365	59	44	0.00289	7.92	0.0213	58.36	0.0123	33.73
Ave*	4.44×10^{-5}	0.0304	60	45	0.00237	7.77	0.0186	60.99	0.0095	31.25

*Average of the time spent on function and gradient evaluations

has a shorter time-frame to evaluate derivatives. Moreover, the optimisation of the algorithm can center on the reduction of the number of gradient evaluations.

Memory Analysis

The proposed method requires the storage of 16 vectors of length n (the vectors x_k, $g(x_k)$, and 14 working vectors). This is similar to the requirement of L-BFGS method, a popular quasi-Newton method, which has a memory storage of $14n$, based on the scheme that has 7 vectors of past information storage. In either case, I do not need to store second-order derivatives information. There is also no storage of matrices; only storage of vectors suffices.

3.7.3 Performance on Deep Neural Networks

To justify the performance of the algorithm on Deep Neural Networks, I define a hypothetical neural network and implement the optimiser on this network. In this case, a deep neural network is arbitrarily defined to have three hidden layers with 10, 50 and 10 neurons respectively. The structure of the neural network is shown in Figure 3.6. There are 1,000 input data points with values randomly generated. 500 are labelled 1 and the other 500 labelled 0. I compare the performance of the designed HFGF optimiser to those commonly used in industrial settings, including L-BFGS, SGD and Adam. The HFGF optimiser is coded

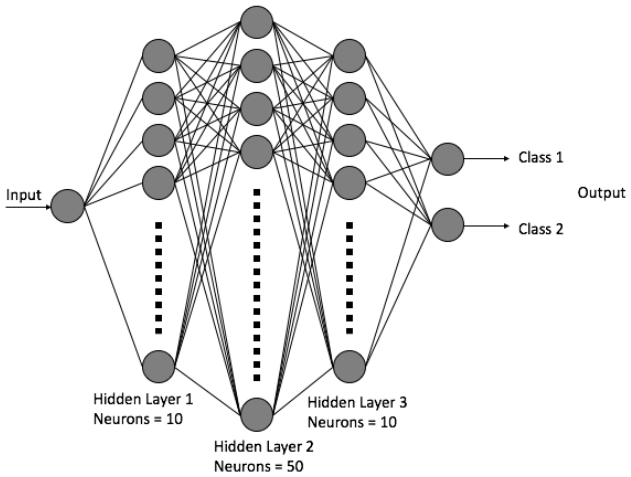


Figure 3.6 The structure of the deep neural network

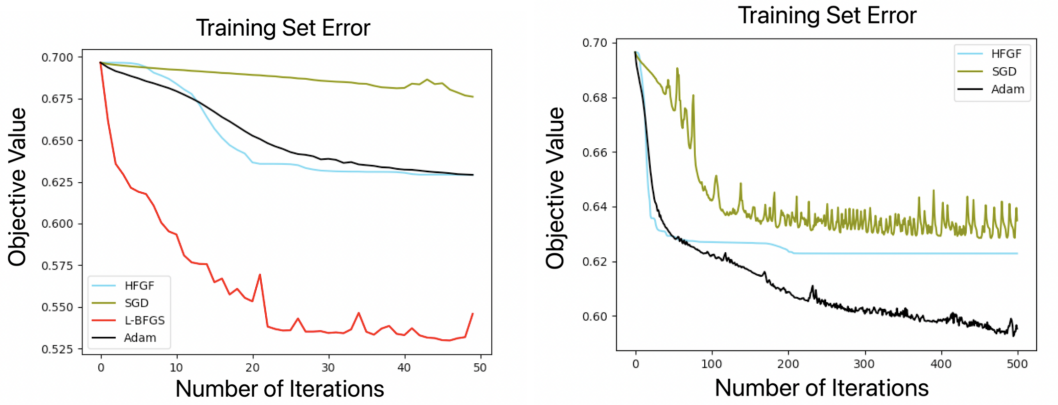


Figure 3.7 Comparison of the performance of different optimisers (a) running 50 epochs to observe the rate of convergence and (b) running 500 epochs to observe the final convergence value

in Python by inheriting from the "Optimiser" class published by Pytorch, with a backend written in C. The resulting of running 50 and 500 epochs are shown in Figure 3.7.

From Figure 3.7, it can be observed that when there are 50 epochs, L-BFGS is the best performing optimiser, followed by the HFGF method. However, the L-BFGS method does not reach the convergence criterion defined by its own and does not produce an optimal value. HFGF, although initially slower than SGD and Adam, later catches up with the other optimisers, generating a lower value of loss function. In this case, none of the optimisers have reached convergence within 50 epochs. When extended to 500 epochs, L-BFGS starts to overflow hence is not included in the Figure 3.7. This corroborates with the general perception in the machine learning community that L-BFGS is weaker for optimisation of

larger neural networks. Among the other three optimisers, Adam is the best performing algorithm, generating the lowest value of the loss function, although the value zig-zags slightly. HFGF produces a rather smooth curve and generates the second lowest loss value, outperforming the industrial convention of SGD optimiser. The SGD optimiser gives a zig-zag output, converging to a value higher than that of HFGF. It is noteworthy that in this case, hyperparameters of each optimiser have been tuned to improve behaviour with Adam having a learning rate of 0.01 and the SGD a learning rate of 0.5. This may explain the zig-zag behaviour produced by SGD, as the learning rate is set too high. However, when the learning rate is low, it will take over 800 epochs for the optimiser to converge. The slow rate of convergence is adjusted by increasing learning rate such that a comparison with other optimisers can be visualised.

From the plot of losses against the number of epochs, it can be seen that the key advantage of our proposed algorithm is that it is a quasi-Newton method, hence it is fast in converging to its final value compared to first-order method such as Adam and SGD. For example, in Figure 3.7, although its final convergence value is not the lowest, it is the first among all three optimisers to reach its final value. It is safe to claim that there is a trade-off of convergence speed and objective value in this case. HFGF is a better choice when convergence speed is important.

Moreover, as a quasi-Newton method, HFGF is more robust towards larger dataset and more complicated architectures compared to the state-of-art quasi-Newton method L-BFGS. When L-BFGS generates overflows for deep neural networks, our HFGF method is still functioning well.

A distinct feature of the optimiser is that it initially converges slowly, generating a plateau in the graph, and then decreases rapidly in a few epochs to a very low value of convergence. This is because during the initial search, the step size is small and doubles slowly. Once the value of the step size is increased to a large value, the search for optimality proceeds rapidly. Then the values of the objective do not decrease further and the step is rapidly decreased, generating smaller decreases in the value of the objective function.

3.7.4 Applications to Real-world Datasets

The analysis above provides a theoretical understanding of the algorithmic performance. In this section, the optimiser is applied to several test cases to determine its real-world performance. The derived algorithm is applied to large-scale optimisation of DNNs to test its speed, robustness and accuracy. In particular, DNNs are optimised to claim the efficacy of the optimiser to large-scale problems.

Datasets

In order to test the performance of our proposed optimiser, I perform one classification test and one regression task on the proposed optimiser adopting real-life dataset. First, I perform classification on the MNIST dataset. The MNIST dataset has a training set of 60,000 examples and a test set of 10,000 examples, consisting of hand-written digits representing the numbers 0-9. It is a commonly used database in the machine learning community to test pattern recognition capability of a designed network with minimal efforts on pre-processing and formatting the images. I adopt the state-of-the-art Convolutional Neural Network (CNN) architecture - LeNet5 [185] - to analyse the MNIST database. The network consists of two convolutional layers separated by maxpooling layers, followed by three fully-connected layers. Here I adopt mini-batches to perform optimisation. The activation function used is ReLU.

Second, I perform a regression task on a research dataset, termed OILDROPLET dataset. Details of the dataset can be found in Appendix. The dataset contains observations generated from dropping an oil droplet to a surface of water. The input features of the dataset include composition of the droplet, environment temperature, oil viscosity, oil surface tension and oil density. The output features are generated by observing the movement and merging of the oil droplet on the water surface, including average movement speed, maximum speed of a single droplet, average number of droplets in the last second, average number of droplets throughout the experiment. I construct an optimised neural network consisting of three hidden layers with 50, 20 and 20 neurons respectively. Between each layer I apply the ReLU activation function. Since the key is to compare performance of the optimiser, the architectural parameters are held constant.

CNN on MNIST Data

The network is run on several optimisers and the comparison of the performance is shown in Table 3.8. I have selected SGD, Adam and L-BFGS as the target of comparison. SGD and Adam are selected because these are the first-order stochastic methods. SGD is the most basic form and Adam is the best-performing counterpart. L-BFGS is adopted because it is a quasi-Newton method. It acts as a benchmark to the current quasi-Newton methods and is known to be very fast. Before comparison, I optimise the performance of the optimisers through hyperparameters tuning. The learning rate for SGD is set at 0.01, the optimal hyperparameter obtained after tuning. The learning rate for Adam is 0.001. The hyperparameter values for HFGF are the set of default values I determined to be optimal in this analysis. The

performance of the three optimisers are compared based on accuracy of prediction on the test set.

Table 3.8 Comparison between the performance of different optimisers on the classification task of MNIST data

Optimiser	Accuracy	Final training loss	Time of convergence (s)
SGD	92.5%	0.0801	55.98
Adam	91.6%	0.2448	65.13
L-BFGS	90.9%	0.3565	2225.21
HFGF	96.4%	0.0643	142.11

From Table 3.8, it can be seen that our proposed method obtained comparable result in terms of accuracy of classification. While the conventional quasi-Newton method of L-BFGS sometimes breaks down with a high time of convergence, our proposed quasi-Netwon method is robust to the problem. Although the final loss achieved is higher than the first-order methods, the classification accuracy is comparable to the other two optimisers. Moreover, although the time of convergence is longer than stochastic first-order methods, it is lower than the benchmark of L-BFGS. For problems that rely on speed of computation, an improvement from 92.5% to 96.4% in accuracy at the expense of computation time is not desired. This can be viewed as a limitation of the algorithm. In cases where accuracy is the key, the results show promises in improving performance. Moreover, compared to other quasi-Newton methods such as L-BFGS, the speed becomes an edge as there is significant improvement in terms of computation time.

DNN on OILDROPLET

I apply the same set of optimisers, SGD, Adam, L-BFGS and the proposed optimisation algorithm, on the OILDROPLET dataset. The result of application is shown in Table 3.9. The learning rate used in SGD is 0.01, and in Adam is 0.05. In HFGF method, I have mostly used the default settings or settings of a similar scale. The settings used include the following: the increase in the value of h after each step is $8\times$, the decrease is $0.125\times$. The hyperparameters defined to have infinitesimal values are all close to zero. The value of ε is 10^{-7} and of tolerance is 10^{-5}. The value of μ is 10^{-4} and of initial h is 10^{-3}. The maximum number of inner iterations is originally 15 but is now tuned to 2. This tuning has greatly increased the processing speed of the algorithm. To evaluate the effectiveness of optimisation, I use the mean squared error (MSE) loss on the test dataset as the metric for performance. Since there are four output feature, I tabulate an MSE for each dataset.

Table 3.9 Comparison between the performance of different optimisers on the classification task of OILDROPLET data. S_{Ave} represents the average speed of droplets. $D_{FinalAve}$ represents the average number of droplets in the last second. S_{Max} represents the maximum average single droplet speed. D_{Ave} represents the average number of droplets.

Optimiser	Training loss	Test error				Time of convergence (s)
		S_{Ave}	$D_{FinalAve}$	S_{Max}	D_{Ave}	
SGD	0.389	0.3782	0.3785	0.4616	0.3365	162.89
Adam	0.224	0.1960	0.2297	0.2589	0.2104	160.76
L-BFGS	0.188	0.1583	0.1938	0.2223	0.1789	4249.79
HFGF	0.290	0.2800	0.2707	0.3478	0.2577	535.89

On the OILDROPLET dataset, the HFGF method achieved better result than SGD in terms of final loss and regression error. It is under-performing compared to Adam in terms of final loss, test error and speed. However, it is faster than the quasi-Newton method of L-BFGS. Considering that L-BFGS sometimes fails for complicated problems, our optimiser is able to perform more complicated tasks and is more robust to datasets of high dimensionality.

3.8 Further Analysis of HFGF

The HFGF optimiser has been successfully adopted to optimise neural networks. However, I would like to perform further analysis on the optimiser to explore its properties. The main problems I would like to explore include:

- How to speed up the initial descent
- What is the effect of changing approximation of Hessian to only make use of function information
- What is the effect of making the search direction adaptive
- How does hyperparameter optimisation affect the optimiser performance

I will discuss each point in detail in the following sections.

3.8.1 Powerball Function

The performance on the hypothetical neural network has demonstrated that one distinct characteristic of the optimiser is its initial slow descent as demonstrated in the graph of loss

against iterations. There is a small plateau before a rapid descent. Therefore, this section seeks to speed up initial descent by adopting current research in optimisation.

To demonstrate the transformations introduced by Powerball function, I first take a look at the Newton's update rule. The Newton's method optimises a function based on the following search direction:

$$x(k+1) = x(k) - H_k^{-1} \nabla f(x(k)) \tag{3.45}$$

where $k = 0, 1, ..., H_k^{-1}$ is the Hessian matrix and $\nabla f(x(k))$ is the gradient of function f at $x(k)$.

In [186], the Powerball function was proposed, which applies a nonlinear element-wise transformation to the gradient by:

$$x(k+1) = x(k) - H_k^{-1} \sigma_\gamma \nabla f(x(k)), for\, k = 0, 1, ... \tag{3.46}$$

where $\sigma_\gamma(z) = sign(z)|z|^\gamma$ for $\gamma \in [0, 1)$. Sign(z) returns the sign of of z if $z \neq 0$, or 0 if $z = 0$. This transformation is believed to speed up initial descent which solves the problem existing in the current optimisation method. Details of proof can be found in [186].

To analyse the effects of adopting the Powerball function, I have added it to the optimiser's update rule. I then perform the optimisation of the neural network on MNIST data. By plotting the training loss against the iterations, I can observe the acceleration of descent brought about by Powerball function, as demonstrated in Figure 3.8.

3.8.2 Hessian Approximation

From Section 3.7.2, I have seen that the most time-consuming step in the optimisation algorithm is the gradient evaluations. To optimise the performance of the algorithm, I replace the gradient evaluations with function evaluations. The original approximation of the Hessian matrix is defined by the first-order derivatives:

$$Q_k p_k = \frac{g(x_k + \varepsilon p_k) - g(x_k)}{\varepsilon} + \frac{1}{h_k} I p_k \tag{3.47}$$

where g is the first-order derivative and H is the Hessain matrix. However, I replace the approximation of the Hessian matrix with function evaluations, *i.e.* replace first-order derivatives with values of the objective function:

$$g(x_k) = \frac{f(x_k + \varepsilon p_k) - f(x_k)}{\varepsilon} \tag{3.48}$$

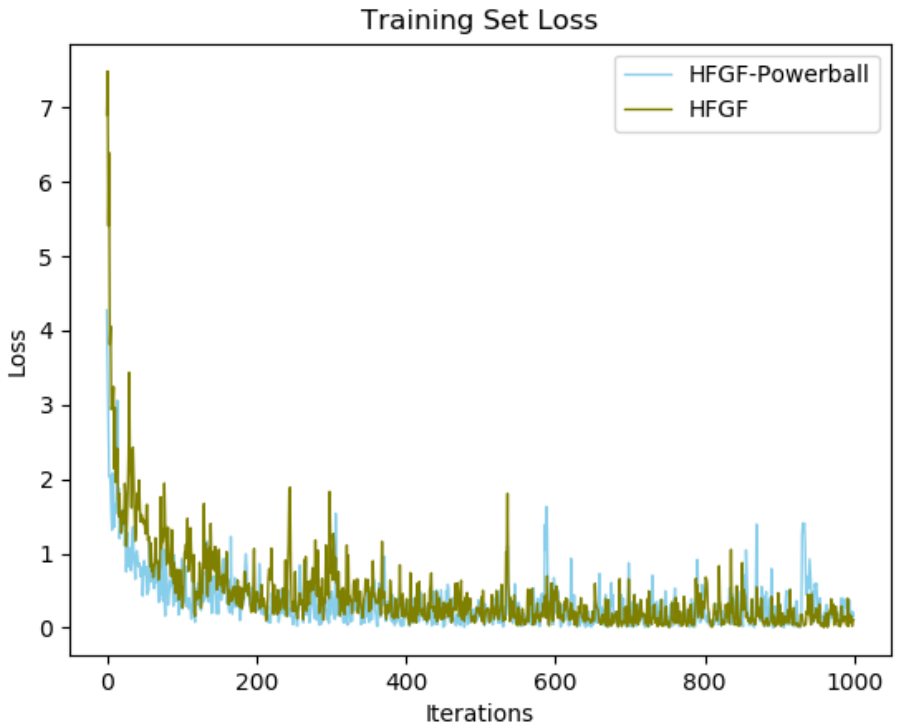


Figure 3.8 Training loss against iterations with application of Powerball function at $\gamma = 0.8$

$$g(x_k + \varepsilon p_k) = \frac{f(x_k + 2\varepsilon p_k) - f(x_k + \varepsilon p_k)}{\varepsilon} \tag{3.49}$$

Substituting Equation 3.48 and 3.49 into Equation 3.47, I obtain an equation that is based purely on function evaluations:

$$Q_k p_k = \frac{f(x_k + 2\varepsilon p_k) - 2f(x_k + \varepsilon p_k) + f(x_k)}{\varepsilon^2} + \frac{1}{h_k} I p_k \tag{3.50}$$

I use Equation 3.50 in the optimisation algorithm, and evaluate the performance keeping all other parameters equal to the original derivative-based evaluation. The results and comparison with the original HFGF algorithm are tabulated in Table 3.10.

Table 3.10 Approximation of Hessian with function evaluations

Evaluation	Loss	Accuracy	Time	Number of Iterations
Gradient	0.472971	93.431%	644.78	120
Function	NaN	10.920%	7323.74	120

From Table 3.10, I observe that the approximation using function evaluations does not generate good approximation results for the application of HFGF. The optimiser diverges, generating a loss value of *NaN*, with a much longer CPU time to optimise to the local minimum. One possible reason for the divergence is the introduction of extra uncertainty brought about by the second-order approximation. The optimisation algorithm requires high accuracy and a more rigorously calculated gradient value to allow effective DNN training.

3.8.3 Adaptive Step Sizes

The optimiser takes adaptive step sizes where the value of step size h is doubled when a significant descent defined by the *Amijo condition* is made and halved when a descent is not made. In certain cases, I can implement a more aggressive search by increasing and decreasing the value of h by a proportionally larger amount.

Table 3.11 tabulates the performance of each value of h and the effect of such a change is observed. It is noteworthy that the product of the increase and decrease in step size is always equal to 1. The values tabulated (except time) are calculated from the last epoch of optimisation. The time refers to the total time of optimisation and is calculated by running the code on a processor that is the 2.3GHz Quad-Core Intel Core i5 with a memory setting of 8GB 2133MHz LPDDR3.

Table 3.11 The effect of adaptive step sizes on the performance of optimisation.

Step Size (Increase)	Step Size (Decrease)	Loss	Accuracy	Time	Number of Iterations
1	1	0.032402	99.730%	387.28	120
2	0.5	0.472971	93.431%	644.78	120
4	0.25	2.429082	8.172%	607.19	1
8	0.125	4.863550	10.067%	488.62	1
16	0.0625	2.324766	97.336%	837.05	1

From the results, it can be observed that the best performing changes in step sizes is not to change the step size (increase=1, decrease=1). The second best performing is to double when the descent condition is met and to halve when the condition is not met. Other types of increases or decreases are too aggressive and they result in high loss and low validation accuracy. The average number of iterations in the last epoch is 1 for those divergent optimisation processes. The loss increases with the step size changes getting more and more aggressive. Thus, a potential improvement of the HFGF algorithm is to remove the adaptive step sizes and keep a constant step size, at a value of $h = 10^{-3}$ in this case.

3.8.4 Hyperparameter Optimisation

One distinct feature of our designed HFGF optimiser is that it has a comparatively high number of hyperparameters. In the design of the optimiser, a number of variables have been intentionally set as the hyperparameters to give more freedom to the training of the neural network. Tuning hyperparameters has been an active research field and there are many cases where hyperparameters are tuned through grid search or random search in order to achieve better optimisation results [187] [188]. In this case, I have micro-tuned the hyperparameters of the HFGF method and several observations have been obtained:

- The HFGF optimiser performs better on the designed CNN when ε is set at 10^{-7}. Performance deteriorates rapidly when ε is 10^{-6}. This is empirically perceived to be the lack of an accurate approximation of the Hessian matrix.
- The parameter that doubles and halves the step size can be changed to higher and lower rates for different networks and this greatly affects the rate of convergence. These two parameters should be the focus of any hyperparameters tuning.
- The value of μ has been performing well at a value of 10^{-4}.

- The value of tolerance is currently set to be 10^{-9}. This is empirically determined from values of the loss function. When the dimension is larger than those experimented in this project, it is expected that the tolerance should be decreased accordingly. Moreover, it is widely accepted that the neural network is often very robust to different levels of optimisation accuracy, which explains the popularity of SGD optimiser. Thus, the tolerance is set as a tunable hyperparameter, increased when in pursuit of accuracy and decreased in pursuit of speed.

- Initial values of step size h has minimal effect on the whole optimiser as it is quickly doubled or halved throughout the optimisation process. However, an appropriate h does generate faster convergence. The current default value in use is empirically designed to be 10^{-3}.

- Further analysis of the hyperparameters can change the number of inner iterations in the HFGF optimiser. Previous investigations on testing functions have indicated that the optimal number of inner iterations is 15. However, when the architecture of the network changes, the optimal number of iterations also changes. A number of inner iterations too large can slow down the optimising process, leading to slower rate of convergence.

3.9 Summary

I have demonstrated the derivation of a novel quasi-Newton optimisation method with a proof of convergence. The method is named Hessian-free Gradient Flow (HFGF) algorithm and has been designed for the optimisation of DNNs. I first tested the HFGF method on standard testing functions and then compared it with the other common optimisation algorithms to test for its convergence. Then I performed time analysis to identify the most time-consuming steps in the proposed algorithm. I also briefly discussed the storage requirement. The HFGF method was then applied to case studies where open-source databases and real-world industrial databases were used to test the effectiveness of the optimisation algorithm.

The aim of the algorithm is not to beat cutting-edge first-order stochastic algorithms as the quasi-Newton method itself carries with it a computational cost. However, our algorithm has demonstrated comparable performance in tasks of classification and regression with regard to popular first-order algorithms. It is also more robust than L-BFGS algorithm if I set L-BFGS as a benchmark for quasi-Newton algorithms.

In summary, although the method is not the fastest optimisation algorithm compared to L-BFGS algorithms in simple cases, and not the simplest compared to SGD or Adam,

it introduces a trade-off between speed and robustness. Moreover, the method has demonstrated advantages in classification accuracy in large datasets compared to most first-order algorithms.

Currently ongoing future research is focused on improving the speed of the new method, particularly by reducing the number of gradient evaluations and a more optimised algorithmic implementation architecture. To prevent convergence to saddle points and local minima, I can also introduce stochastic components to the optimiser. According to the principle of "no free lunch", each optimisation method has its own advantage in certain contexts. I believe our new algorithm may be more suited for the fitting of DNNs, compared to other standard optimisation methods.

Chapter 4

Autonomous Sparsification - *Part I*

4.1 Introduction

Historically, the field of Artificial Intelligence is full of ideas inspired from biomedical research. For example, the structure of Artificial Neural Networks (ANNs) itself is inspired from how neurons are organised in the brain and how they fire electronic signals [189]. To advance the process of neural architectural search, I similarly attempt to gain inspirations from biomedical research. I am motivated to develop "true intelligence" that simulates the functionality and evolution of the brain. In particular, I focus on the concept of "neurogenesis", which states that the neurons in the adult brain are capable of duplicating and differentiating continuously with regard to the environment [190]. I hope to integrate such functionalities in the neural architectural search, by allowing the architecture of the network to constantly evolve until an optimal architecture is obtained. This optimised architecture is minimal in nature and can convey all relevant information particular to the data inputs.

More specifically, I formulate the self-evolution of a neural network mathematically, enabling a systematic simulation of the brain construction and destruction process. During the process, neurons take on new knowledge and correct the previously learnt output. A neuron that has a high importance in the network is duplicated and a neuron with less importance is deleted.

An optimisation scheme for ANNs is proposed. The scheme aims to achieve the following targets, which are previously very hard if not impossible to contemplate systematically:

- Rapid and organic evaluation of the ANN architecture in a completely free and deterministic way, emulating neuronal/brain real tissue growth and learning.

- The very simple steps proposed are a kind of "survival of the fittest" scheme that allows the neurons in an ANN to produce offspring when they are "fit" and kill offspring that are deemed "unfit".

While conceptually simple and clear-cut, the proposal of a framework to evolve neural architecture contains several challenges. First, it is difficult to formulate the ANN into a dynamic system where particular neurons can be easily added or removed from the whole structure. While current literature adopts the dropout method in a deep network to improve performance [191] [192] , this dropout is usually random, with a defined dropout rate. To continuously evolve a network by dropping or adding a specific neuron is less well researched. Moreover, it is difficult to innovate on the formulation of the ANN to allow such a process to be conducted easily. In this chapter, I propose the "lifting" scheme to allow such an evolution process to become easy and fully automated.

Second, there is a lack of a standardised measure to guide how to modify the network. Recent research in the "multiple shooting" method that develops a "lifting" scheme [193] allows the formulation of an ANN network into an equality-bounded optimisation problem. Under this formulation, each neuron, or a unit of calculation, is formulated into an equality constraint. It is then possible to extract sensitivities of individual equalities with regard to the overall objective function. This sensitivity value can act as a guide that indicates the importance of a node or a layer. Thus, I effectively have measures to guide the evolution of the network by formulating the solution to an ANN into an optimisation problem under the lifting scheme.

There are several functionalities I am able to achieve through the lifting scheme. In particular, I achieve L1-regularisation by redefining the optimisation problem with added inequality constraints to allow autonomous sparsification of the network. This method can be fully automated and be guided by the Lagrangian sensitivities. I term the process "Autonomous Sparsification".

In this chapter, I introduce the lifting scheme and develops the process of autonomous sparsification. The chapter consists of the following sections:

- Section 4.2 provides the background on the lifting scheme.
- Section 4.3 illustrates the concept of lifting with a simple example.
- Section 4.4 formulates the solution to an ANN into an optimisation problem adopting the lifting scheme.
- Section 4.5 introduces how to make use of Lagrangian multipliers to perform sparsification. It also describes the optimisation scheme and the relevant subroutines.

- Section 4.6 describes in detail the implementation process of the autonomous sparsification scheme.
- Section 4.7 applies the lifting scheme first to a single point optimisation, and then expands to inputs with higher dimensions and with multiple input data points.
- Section 4.8 introduces a nonlinear chemical process and applies the lifting technique to produce an ANN that solves this problem adopting simulated data. Key results of the sparsification process are presented.
- Section 4.9 then describes the sparsification process and the gradual growth scheme applied to feedback ANNs and ANNs in their most general form.
- Section 4.10 then summarises the chapter.

4.2 Literature Review

The lifting scheme is adopted by [193] to solve nonlinear programming problems (NLPs) that employs intermediate variables for the objective and constraint functions. The method involves transforming an NLP by adding new constraints into an equivalent problem. The article discusses the application to a Newton-type method with no additional computational cost or programming burden. Empirical results demonstrate faster local quadratic convergence adopting the lifting scheme.

The lifting scheme works by "introducing intermediate variables as additional degrees of freedom and corresponding constraints to ensure equivalence to the original problem" [193]. It then solves this augmented equivalent system to achieve optimality, while bringing in benefits in terms of convergence and solution if properly formulated. Although the lifting scheme seems more expensive due to the involvement of intermediate variables, it can be redefined with "structure-exploiting linear algebra" [193] to overcome the complexity.

The lifting scheme has several advantages. First, by transferring a nonlinear root finding problem into a higher-dimensional space, the lifting scheme offers advantages in terms of convergence rates and region of attraction [194]. Second, the lifting scheme provides benefit to classical problems of optimal control in Ordinary Differential Equations [195] and Differential Algebraic Equations [196], where the intermediate variables are system states at different time points. Third, it is demonstrated that a properly designed lifting scheme achieves superior local convergence speed in a Newton type problem [193].

The disadvantage of the lifting scheme is often its computational and programming burden due to the addition of intermediate variables. However, with commercially available optimisers, I expect the burden to be overcome and large-scale application is possible.

4.3 Concept of Lifting

Lifting is defined as a functional mapping of a mathematical problem to higher dimensions. The lifting scheme has the following properties:

- Property 1: Solution of the lifted problem is in some way easier than the original.
- Property 2: Once solved in the lifted space, it is easy to transform the solution to the original space.
- Property 3: The original and the lifted models yield the same solution.
- Property 4: In the case of Nonlinear Programming (NLP), the lifted and the unlifted models share exactly the same set of local minimisers (hence also the global ones).

This can be demonstrated in the following example of a constrained nonlinear optimisation problem solved using lifting.

Suppose I have the optimisation problem with the following objective function:

$$\min_{x1,x2} \quad sin[exp(\frac{x_1}{x_2})] + (x_1 + x_2) \cdot (lnx_1 + lnx_2) \tag{4.1a}$$

$$\text{subject to} \quad x_1^2 + x_2^2 \leq 1 \tag{4.1b}$$

$$0 \leq x_1 \leq 1 \tag{4.1c}$$

$$0 \leq x_2 \leq 1 \tag{4.1d}$$

Lifting on NLP uses the concept of a function evaluation tree to aid the mapping transformation from the original NLP model. For the optimisation problem in this example, the objective function tree is shown as a Directed Acyclic Graph (DAG) (Figure 4.1). The tree demonstrates the intermediate steps in computing $f(x)$ from values of x_1 and x_2.

The constraints are separately described in the constraint function tree (Figure 4.2), where *pow* denotes the following operation:

$$pow(x, \alpha) = x^{\alpha} \tag{4.2}$$

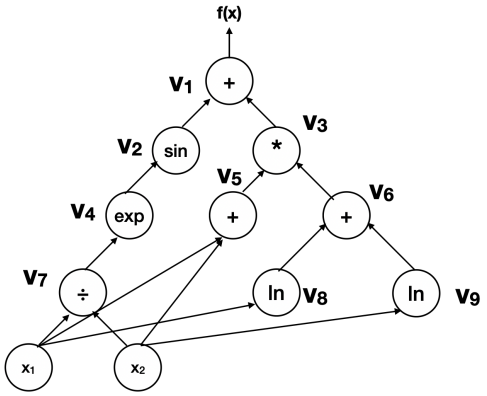


Figure 4.1 Directed Acyclic Graph showing the objective function tree for the example optimisation problem

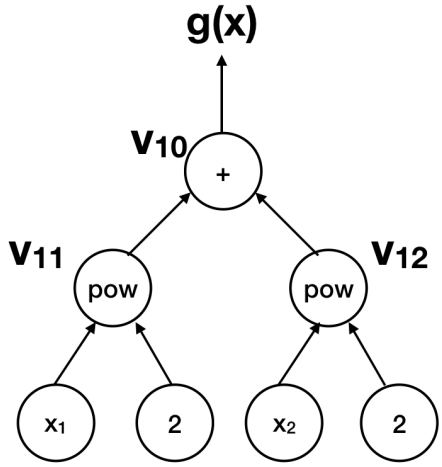


Figure 4.2 Directed Acyclic Graph showing the constraint objective function tree for the example optimisation problem

Thus, the optimisation problem can be adopted in the lifting scheme as follows:

$$\begin{aligned}
\min_{x} \quad & v_1 && \text{(4.3a)}\\
\text{subject to} \quad & v_1 = v_2 + v_3 && \text{(4.3b)}\\
& v_2 = sin(v_4) && \text{(4.3c)}\\
& v_3 = v_5 \times v_6 && \text{(4.3d)}\\
& v_4 = exp(v_7) && \text{(4.3e)}\\
& v_5 = x_1 + x_2 && \text{(4.3f)}\\
& v_6 = v_8 + v_9 && \text{(4.3g)}\\
& v_7 = x_1/x_2 && \text{(4.3h)}\\
& v_8 = ln(x_1) && \text{(4.3i)}\\
& v_9 = ln(x_2) && \text{(4.3j)}\\
& v_{10} = v_{11} + v_{12} && \text{(4.3k)}\\
& v_{11} = x_1^2 && \text{(4.3l)}\\
& v_{12} = x_2^2 && \text{(4.3m)}\\
& 0 \leq x_1 \leq 1 && \text{(4.3n)}\\
& 0 \leq x_2 \leq 1 && \text{(4.3o)}\\
& v_{10} \leq 1 && \text{(4.3p)}
\end{aligned}$$

With this lifting reformulation, the original problem of dimension $x \in \mathbb{R}^2$ can be reformulated into dimension of $z \in \mathbb{R}^{14}$, where $z \in x \cup v$ and z is defined to be $z = (x_1, x_2, v_1, v_2 ... v_{12})^T$. In this way,I effectively lifted the optimisation problem into higher dimensional space and the optimisation problem is easier to solve.

In the case of constrained optimisation to be delivered to a solver, the bounds of x's can be propagated to become the bounds of v's. Interval arithmetic can be used to calculate the bounds for the lifted variables v_i for $i = 0, 1, 2, ...12$. Here I conclude the example demonstrating the lifting scheme.

There are specific advantages of adopting the lifting scheme to the problem of ANN fitting. For one, ANNs are already defined as a DAG which can be easily formulated into the lifting scheme. Further advantages will be discussed in the following section.

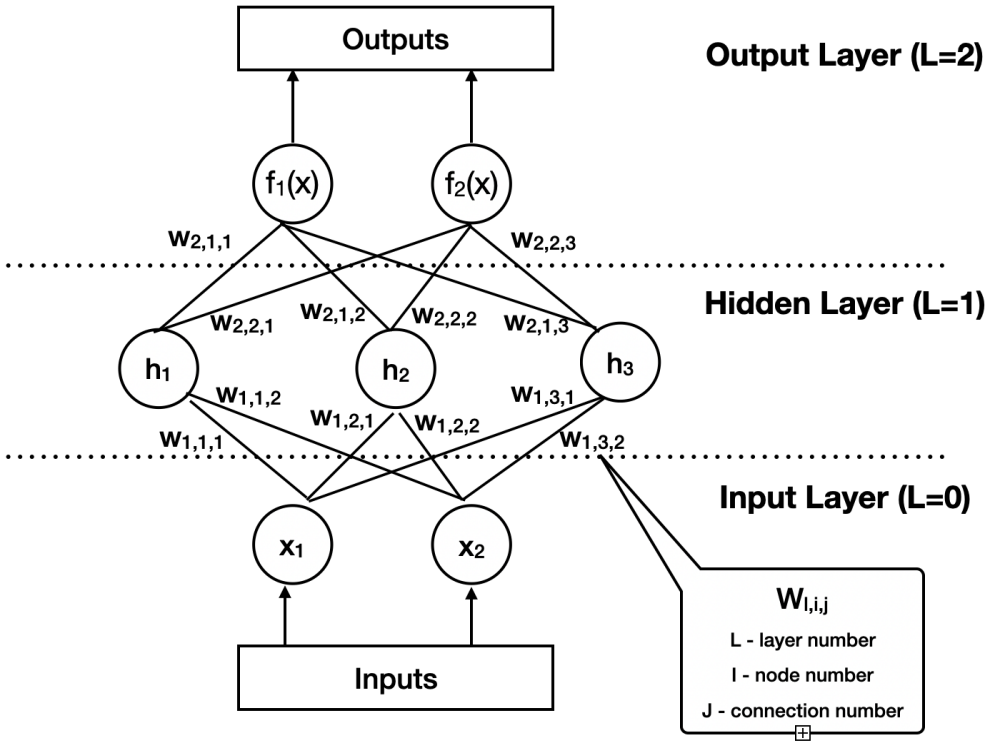


Figure 4.3 A simplified version of ANN with one hidden layer of 3 neurons used for the formulation using the lifting scheme

4.4 Formulation of ANN Fitting Problems

The fitting of an ANN can be formulated into a constrained optimisation problem involving the lifting scheme. In Section 4.3, I have explained how an optimisation problem can be formulated into a Directed Acyclic Graph (DAG). In the case of ANN (and DNN), the network has already been formulated as a DAG. Therefore, a natural extension would be to formulate the fitting problem into an optimisation problem involving lifting.

A simplified version of the formulation of ANN fitting involving 2 inputs and 2 outputs is shown in Figure 4.3. Suppose I have a pair of inputs $x_k \rightarrow \{x_{1,k},\, x_{2,k}\}$. The first layer of transformation can be represented in the following equations:

$$v_{1,1}^{[k]} = W_{1,1,1} \times x_1^{[k]} + W_{1,1,2} \times x_2^{[k]} + W_{1,1,0} \tag{4.4}$$

$$v_{1,2}^{[k]} = W_{1,2,1} \times x_1^{[k]} + W_{1,2,2} \times x_2^{[k]} + W_{1,2,0} \tag{4.5}$$

$$v_{1,3}^{[k]} = W_{1,3,1} \times x_1^{[k]} + W_{1,3,2} \times x_2^{[k]} + W_{1,3,0} \tag{4.6}$$

where $v_{l,i}$ is the result of linear transformation in layer l node i, and $W_{l,i,j}$ represents the weight in layer l node i from input node j in the previous layer. $W_{l,i,0}$ represents the bias term in the linear transformation.

The linear transformation is followed by activation functions that generate the node values in the hidden layer.

$$h_{1,1}^{[k]} = \sigma(v_{1,1}^{[k]}) \tag{4.7}$$

$$h_{1,2}^{[k]} = \sigma(v_{1,2}^{[k]}) \tag{4.8}$$

$$h_{1,3}^{[k]} = \sigma(v_{1,3}^{[k]}) \tag{4.9}$$

The activated values undergo a further linear transformation to generate the outputs.

$$f_1^{[k]} = w_{2,1,1} \times h_{1,1}^{[k]} + w_{2,1,2} \times h_{1,2}^{[k]} + w_{2,1,3} \times h_{1,3}^{[k]} + w_{2,1,0} \tag{4.10}$$

$$f_2^{[k]} = w_{2,2,1} \times h_{1,1}^{[k]} + w_{2,2,2} \times h_{1,2}^{[k]} + w_{2,2,3} \times h_{1,3}^{[k]} + w_{2,2,0} \tag{4.11}$$

Overall, there are 8 equations governing the simple neural network with input $x^{[k]}$ and output $f^{[k]}$.

The values of v and h are separate sets to W, as values of v and h change with each datum point while W remains the same for any input datum point.

Values of Ws are chosen to optimally minimise a LSQR objective function:

$$min_{W,h,v} \sum_{k=1}^{N_{data}} \left[\sum_{i=1}^{outputs=2} [f_i^{[k]} - \hat{f}_i^{[k]}]^2 \right] \tag{4.12}$$

In this simplified version, the problem of neural network fitting has been effectively expressed as a DAG with 8 equations. I term this unconstrained minimisation problem expressed under the lifting scheme as ANN-NLP-1.

The formulation of the ANN fitting problem into a constrained optimisation problem is characterised by the following properties:

- Lifted model will be much larger
- The constraints will contain bilinearities
- The lifted model will contain solitary, single-variable functions giving each neuron's outputs
- The model will be relatively sparse in constraints

The optimisation problem transformed by the lifted scheme can be easily solved by Interior Points method using the commercially available package of IPOPT. IPOPT can effectively solve NLPs with more than 10^6 variables and constraints. As the formulation involves intermediate variables that will make the optimisation problem large, the capability of IPOPT to solve high-dimensional problems makes it very handy to use. Moreover, the Lagrangian sensitivity values are required, which IPOPT can solve with simple calculations. Literature has also demonstrated the efficiency of IPOPT compared to other optimisers [197] [198], making it a very sensible choice. Therefore, I solve the fitting of ANNs effectively by using the NLP solver of IPOPT. The unconstrained optimisation algorithm proposed in Chapter 3 is not applicable in this case as this chapter is dealing with constrained optimisation problems.

A key challenge to the formulation of ANNs adopting the lifting scheme is that the complexity of the optimisation process scales with the number of data points. I expect that the number of iterations and the CPU time taken will increase with increasing data points, leading to problems if the speed is too low. To find out how it scales, I perform several optimisation processes with different number of data points and I record the number of iterations and CPU time taken. The processor used to run this code is the 2.3GHz Quad-Core Intel Core i5 with a memory setting of 8GB 2133MHz LPDDR3. The results are tabulated in Table 4.1.

Table 4.1 The number of iterations and the CPU time taken to optimise the network with different number of data points

Number of Data Points	Number of Iterations	CPU Time (s)
500	115	201.02
600	86	10.80
700	82	12.46
800	539	106.87
900	5284	2256.46

From Table 4.1, I can see that there is no definite pattern between the number of data points and how fast the optimisation process is. I try to increase the number of data points until the CPU Time is over 1 hour. I can observe that the maximum number of data points the system is capable of processing under this CPU time limit is 900, adopting an architecture of [4,5,5,5,3,2], where each number represents the number of neurons in successive layers. It should be noted that the time and structure reflect the idiosyncrasy of the data and the idiosyncrasy comes into the picture as well.

Another feature to evaluate is how the optimisation time changes with the number of neurons in the network. I expand the network depth but not the network width until the CPU time is over 1 hour. I observe that the maximum number of neurons the system is capable of processing under this CPU time limit is 150, with 500 data points, as tabulated in Table 4.2.

Table 4.2 The number of iterations and the CPU time taken to optimise the network with different number of neurons

Number of Neurons	Number of Iterations	CPU Time (s)
50	72	87.484
60	71	142.648
80	147	655.480
110	61	499.441
150	102	1550.635

From practical implementations, I observe that the NLP solver IPOPT is capable of overcoming local minima to achieve a lower optimal point overall. This is demonstrated by the increase in the value of the objective function during the optimisation process followed by a decrease to a lower value.

I also observe from practical implementation that the optimisation processes take very long to run. Thus, in future implementations, it will make the system better to improve the speed of solving NLPs when time is limited.

While I have demonstrated that the ANN fitting can be reformulated into optimisation problems effectively under the lifting scheme, the scheme can also be applied to the problem of ANN design through the process of "ANN Sparsification", which is shown below.

4.5 Sparsification of ANNs in Theory

In Section 2.2.7, I have introduced the concept of pruning to obtain a minimal neural network that is almost as effective as the original one. In the mathematical formulation, I name the term "Sparsification", defined equivalently as the pruning process, to remove the neurons that are less important in generating the outputs. In this section, I describe the lifting transformation and its adoption in neural network design. I introduce the pruning process used in the machine learning for a fixed number of layers and neurons, and the Lagrangian multiplier analysis to evaluate node performance in the ANN and its impact on the quality of fitting (a systematic method for sparsification). I further discuss the performance of sparsification on entire layers of the neural network without losing effectiveness of calculating the outputs.

4.5.1 Mathematical Formulation of Connection Sparsification

This section introduces the method used in network sparsification, which formulates the sparsification problem into a constrained optimisation problem.

To enable sparsification, it is required that I optimise two separate objectives: 1) minimising the prediction error, and 2) sparsifying the network connections. This gives us a bi-objective optimisation problem which generates Pareto optimalities. That is, the minimisation of LSQR value and the minimisation of network density does not dominate over one another. The aim of the bi-objective optimisation is to obtain one of the Pareto optimalities.

In traditional machine learning research, L1- or L2-regularisation is often adopted. From the L1-regularised problem, as an example, it is common to constrain the objective function as such:

$$\min_{w}(1-\lambda)LSQR(w)+\lambda||w||_1 \tag{4.13}$$

λ represents the penalty coefficient. In this way, the bi-objective problem is defined to become an optimisation with penalty. This is effectively:

$$\min_{w} LSQR(w)+\frac{\lambda}{(1-\lambda)}||w||_1 \tag{4.14}$$

where $||w||_1$ is effectively $\sum_{l=1}^{N_l-1}\sum_{i=0}^{N_i}\sum_{j=1}^{N_j}|w_{l,i,j}|$. N_l is the total number of layers including input layer and output layer. N_i is the number of neurons in layer l. N_j is the number of neurons in the previous layer. Therefore, to perform L1-regularisation is effectively to constrain the sum of weights to a particular percentage of this sum. Thus, I obtain the following formulation.

From the fitted ANN, I compute:

$$\overline{W}=\sum_{l=1}^{N_l-1}\sum_{i=0}^{N_i}\sum_{j=1}^{N_j}|w_{l,i,j}| \tag{4.15}$$

Suppose I name the unconstrained model developed in Section 4.4 as ANN-NLP-1, and the new model with sparsification as ANN-NLP-2. The constrained optimisation problem of sparsifying a neural network is formulated as the following:

Model ANN-NLP-2:

$$ANN - NLP - 1 \tag{4.16a}$$

$$\text{subject to} \quad -\varepsilon_{l,i,j} \leq w_{l,i,j} \leq +\varepsilon_{l,i,j} \tag{4.16b}$$

$$\varepsilon_{l,i,j} \geq 0 \tag{4.16c}$$

$$. \quad \sum_{l=1}^{N_l-1} \sum_{i=0}^{N_i} \sum_{j=1}^{N_j} \varepsilon_{l,i,j} \leq \omega \tag{4.16d}$$

where $l = 1,2,...,N_l - 1$, $i = 0,1,...,N_i$ and $j = 1,2,..,N_j$. $\omega \geq 0$ is a user-input constant parameter such that $0 \leq \omega \leq \overline{W}$ which through the above constraint, controls the sum of the absolute values of weightings at the optimal solution. If I set $\omega = \overline{W}$ and resolve the optimisation problem above, I would obtain the result of ANN-NLP-1, where the following equality holds:

$$w_{l,i,j}^{(*,2)} = w_{l,i,j}^{(*,1)} \tag{4.17}$$

where $w_{l,i,j}^{(*,1)}$ represents the optimal solution to the ANN-NLP-1 model, and $w_{l,i,j}^{(*,2)}$ represents the optimal solution to the ANN-NLP-2 model.

The purpose of formulating the ANN fitting problem into constrained optimisation problem is that I would obtain novel, original and extremely useful new information through the Lagrangian multiplier values of ANN-NLP-2.

In the practical implementation of ANN-NLP-2 model, the initial runs define a slightly tighter constraint on the sum of weightings compared to $\overline{W}$, where $\omega = \overline{W} - \delta\overline{w}$. $\delta\overline{w}$ is defined to be $\delta\overline{w} \geq 0$ and $\delta\overline{w} << \overline{W}$.

The value of $\delta\overline{w}$ follows a finite difference heuristic rule-of-thumb:

$$\delta\overline{w} = 100 \times \sqrt{\varepsilon_{machine}} \times max(1, |\overline{W}|) \tag{4.18}$$

where $\varepsilon_{machine}$ is the machine precision, for example, 10^{-6}.

4.5.2 Definition of Lagrangian Multipliers

This subsection introduces the concept of Lagrangian multiplier. I define an optimisation problem as:

$$\min_{x \in \mathbf{R}^n} \phi(x) \tag{4.19a}$$

$$\text{subject to} \quad h_i(x) = 0 \tag{4.19b}$$

$$g_j(x) \leq 0 \tag{4.19c}$$

where $i = 1, ..., N_h$ and $j = 1, ..., N_g$.

The Lagrangian multiplier at a local optimum point $x^\star$ (KKT point) satisfies the following:

$$\frac{\partial \phi(x^\star)}{\partial (h_i)} = -\lambda_i^{[h]} \tag{4.20}$$

The partial derivatives represent the "sensitivity" of the objective function value (at a KKT point $x^\star$) subject to a perturbation.

Similarly for inequalities, the same holds but with sign restrictions:

$$\frac{\partial \phi(x^\star)}{\partial (g_j)} = -\nu_j^{[g]} \tag{4.21}$$

4.5.3 Adoption of Lagrangian Multipliers

I have defined the constraint:

$$\sum_{l=1}^{N_l-1} \sum_{i=0}^{N_i} \sum_{j=1}^{N_j} \varepsilon_{l,i,j} \equiv \sum_{l=1}^{N_l-1} \sum_{i=0}^{N_i} \sum_{j=1}^{N_j} |w_{l,i,j}| \leq \omega \tag{4.22}$$

This constraint is relaxed for $\overline{W}$ in the first optimal solution of ANN-NLP-2. In the following runs, I may start reducing this value by a factor $0 < \gamma < 1$ such that $\omega = \overline{W} \cdot \gamma$. As γ decreases, I enforce a smaller sum of absolute values of the weighting coefficients. Norm-1 type constraints are then similar to the imposition of Norm-0 (or penalty coefficient if added directly to the LSQR objective function as a penalty term) with the appropriate limit upper bound: $||w||_0 =$ count of elements of vector w such that $w_{l,i,j} \neq 0$ ($||w||_1$ is typically used in "Machine Learning" to sparsify models.)

I evaluate new networks with tighter ω to reduce number of the nonzero weight coefficients at new optimal solution of ANN-NLP-2. This generates a sequence of $\overline{W} > w_1 >$

$w_2 > ... > w_M > 0$ to be solved in the constraints. With tighter constraints I can check which coefficients go to zero exactly by enforcing $\varepsilon_{l,i,j} \geq 0$ on its bound. As $\varepsilon_{l,i,j} \to 0$, $w_{l,i,j}$ will also become zero. I may choose any sparsified and automatically obtained architecture by removing the arc joining neuron $(l-1, j)$ to (l, i) iff $w^{*2}_{l,i,j} = 0$ for some ω: $0 < \omega < \overline{W}$.

4.5.4 Adding / Removing Nodes

This section presents an automated method to add / remove neurons in an ANN network by calculating the impact of the presence of a node to the overall fitting (LSQR minimum).

In the added constraints of ANN-NLP-1, I have:

$$h^{[k]}_{l,i} - \sigma(v^{[k]}_{l,i}) = 0 \tag{4.23}$$

where $l = 1, 2, ..., N_l$, $i = 1, 2, ..., N_i$, $k = 1, 2, ..., N_k$. N_k is the number of data points used in fitting the ANN. Each of these constraints gives the output of node i in layer l for measurement k, and has associated Lagrangian multiplier $\lambda^{[h],k}_{l,i}$.

To evaluate the impact the node (l, i) at experimental point (k) has overall on the objective function, I can observe that through Equation 4.23, this is equivalent to perturbing the RHS of Equation 4.23, *i.e.* perturb the value of $h^{[k](*,2)}_{l,i}$ at the optimal solution of ANN-NLP-2 for a given value of ω: $0 < \omega < \overline{W}$. This effectively measures the impact of the node through the Lagrangian multiplier $\lambda^{[h],k,(*,2)}_{l,i}$ for experiment (k). Since Equation 4.23 is an equality constraint, there is no sign restrictions on the value of λ. Thus, to measure the impact, the absolute value of λ is used.

To measure the overall impact of the presence of an entire node in the ANN structure, I calculate the total impact of all nodes to the objective function value:

$$\lambda_w = \sum_{k=1}^{N_k} \sum_{l=1}^{N_l - 1} \sum_{i=1}^{N_i} |\lambda^{[h],k,(*,2)}_{l,i}| \tag{4.24}$$

The overall impact of each node is normalised, sorted and compared. The normalisation process is shown below:

$$\hat{\lambda}^{(*,2)}_{l,i} = \frac{\sum_{k=1}^{N_k} |\lambda^{[h],k,(*,2)}_{l,i}|}{\lambda_w} \tag{4.25}$$

where $l = 1, 2, ...N_l$ and $i = 1, 2, ..., N_i$.

The value of $\hat{\lambda}^{(*,2)}_{l,i}$ dictates the importance of each node in the network. To prune the network, the nodes with a lower importance are dropped since they contribute insignificantly to the overall objective function at the current optimum of the ANN-NLP-2 model.

The removal is performed by forcefully setting the node value to zero:

$$w_{l,i,j} = 0 \tag{4.26}$$

In practical implementation, this is effectively achieved by setting the upper and lower bound of the variable $W_{l,i,j}$ to 0. The network with dropped nodes are re-optimised and the above process of node removal is performed until a satisfactory result is obtained.

4.5.5 Removal of Entire Layer

The entire layer can be removed based on Lagrangian sensitivity analysis. I follow the enlisted consecutive steps to arrive at an automatic and systematic way of removing layers.

- Step 1: Calculate the normalised node contribution using the solution of ANN-NLP-2
- Step 2: Calculate $\hat{\lambda}_{l,i}$ for node (i) in layer (l)
- Step 3: To calculate the overall normalised contribution of a complete layer to the currently optimal objective function value, calculate the following layer-related indices: $\hat{\lambda}_l = \sum_{i=1}^{N_i} \hat{\lambda}_{l,i}$ for $l = 1, 2, ..., N_l$
- Step 4: The layers with the calculated index may be sorted and compared

Following from the previous steps, I modify the neural network according to the following methods:

- Option 1: Increase by a given percentage the number of nodes in the "top sensitivity value" $\hat{\lambda}_l$ in layer l, as investing nodes in them has the greatest impact
- Option 2: Decrease by a given percentage the number of nodes in "low sensitivity value" $\hat{\lambda}_l$ in layer l, as these nodes have lower impact.
- Option 3: Remove an entire layer if its $\hat{\lambda}_l$ value is comparatively very low (for example, an order of magnitude smaller than the next one higher value for $\hat{\lambda}_l$ for some layer l)
- Option 4: Add an entire layer if its $\hat{\lambda}_l$ value is comparatively very high

4.5.6 Optimisation Schemes

The process of adding/removing a node or a layer can be pursued through the steps as outlined. However, there are several options associated with the optimisation schemes as well.

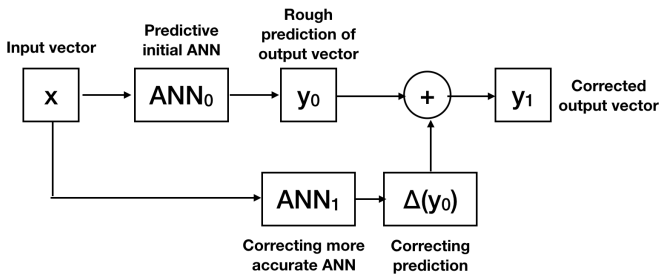


Figure 4.4 Flow diagram denoting one iteration of the scheme to improve ANN structures

- Option 1: Insert/remove the layer/node and re-optimise the whole structure
- Option 2: Insert/remove the layer/node while keeping other weight values unchanged
- Option 3: Insert/remove the layer/node and re-optimise only the weights that are affected by the change in structure

In this chapter, I adopt the first option of the optimisation scheme since I believe changing the architecture of the network may result in a different local minimum, which requires re-optimising. However, in cases where the computational time or complexity is limited, I can adopt the second or the third option.

4.6 Methodology

4.6.1 Initialisation

The experimentation started with a simple initial ANN to tune, and then the structure is analysed and modified. There is no predefined structure requirement of the ANNs, including cycles and connections between the input and output nodes.

A flow diagram of one step of the ANN design process is shown below in Figure 4.4. It involves having a simplistic ANN_0 predicting with a high error and then designing a second ANN_1 using the output of ANN_0.

Because I am embedding "surgically" nodes within a fixed given structure where they are needed the most, in a way producing minimal changes, and because I am simplifying the structure by not carrying over "useless" outdated nodes within the evolving structure in the successive iterations of our algorithm, this is equivalent to "apoptosis" of cells that are becoming isolated and barely used.

4.6.2 Simulation

Using a simple fitted ANN for example as our architecture generation algorithm will produce, consider a fitted ANN:

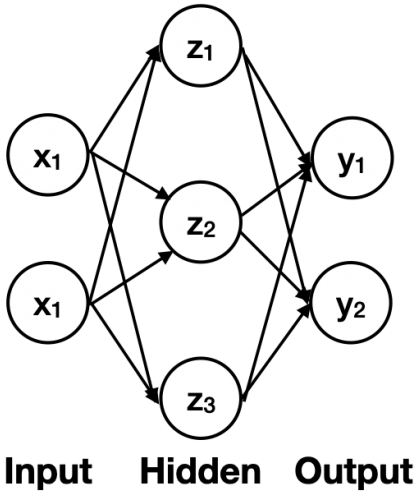


Figure 4.5 Simple fitted ANN as an example

To compute $y_1(x_1,x_2)$, $y_2(x_1,x_2)$, I have to solve the following coupled nonlinear system of equations.

$$v_1 = w_{x_1,1} \times x_1 + w_{x_2,1} \times x_2 \tag{4.27}$$

$$v_2 = w_{x_1,2} \times x_1 + w_{x_2,2} \times x_2 \tag{4.28}$$

$$v_3 = w_{x_1,3} \times x_1 + w_{x_2,3} \times x_2 \tag{4.29}$$

$$z_1 = \sigma(v_1) \tag{4.30}$$

$$z_2 = \sigma(v_2) \tag{4.31}$$

$$z_3 = \sigma(v_3) \tag{4.32}$$

$$y_1 = w_{1,y_1} \times z_1 + w_{2,y_1} \times z_2 + w_{3,y_1} \times z_3 \tag{4.33}$$

$$y_2 = w_{1,y_2} \times z_1 + w_{2,y_2} \times z_2 + w3, y_2 \times z_3 \tag{4.34}$$

There are 8 nonlinear coupled equations to solve. The unknowns are $\{z_1,z_2,z_3,v_1,v_2,v_3\}$, and the inputs are $\{x_1,x_2\}$. The parameters are known and fixed: $\{w_{x_1,1}, w_{x_2,1}, w_{x_1,2}, w_{x_2,2}, w_{x_1,3}, w_{x_2,3}, w_{1,y_1}, w_{2,y_1}, w_{3,y_1}, w_{1,y_2}, w_{2,y_2}, w_{3,y_2}\}$.

The simple ANN as proposed requires the use of Newton's method to give the predicted output. This is practical and feasible with modern Numerical Linear Algebra Algorithms and Newton method for large-scale implementations.

The above formulation serves as an example of how the process of simulation is mathematically modeled. I later also extend from this small network to networks of a much larger scale.

4.6.3 Evaluation of Sensitivities

In the lifting scheme, I have the IPOPT solver to solve both for the values and the Lagrangian multipliers of all variables involved in the definition of the NLP formulation. The Lagrangian multipliers always exist for nodes with either free inlet/outlet weights or fixed ones. Therefore, our Lagrangian sensitivity measure is valid for all nodes and can be used reliably both to add new neurons or to prune the least contributing ones.

4.6.4 Retraining with New Data

When new data are fed into the ANN, a completely novel and unique scheme is adopted, which is unprecedented in the open literature. Start by generating an initial architecture and then removing nodes or connections gradually as it evolves to fit new data and learning conditions. This assumes that even the previous dataset used in training the given ANN may not be even necessary (e.g. after a long time, the old dataset may even be unavailable to include and retrain).

Alternatively, given an ANN fitted to a given dataset of a given process, I can change the architecture in an "evolutionary" way as an entirely novel methodology on ANNs and DNNs.

4.6.5 Nonlinear Approximation

This section serves to perform sigmoid function approximation to enable tunable precision and smoothness in an algebraically easier form for numerical solvers. The approximation method is not a must-to-use but it makes the calculations simpler. Thus, it is recommended to use this form under the lifting scheme. In practice, the approximation is adopted in the nonlinear constraints of the model.

Consider the following approximation of the absolute value function:

$$|x| \simeq \psi(x;\alpha) \triangleq \sqrt{x^2+\alpha^2} \tag{4.35}$$

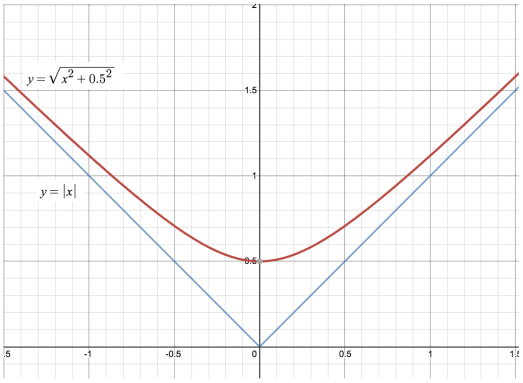


Figure 4.6 Approximation of the absolute value function with $\alpha = 0.5$

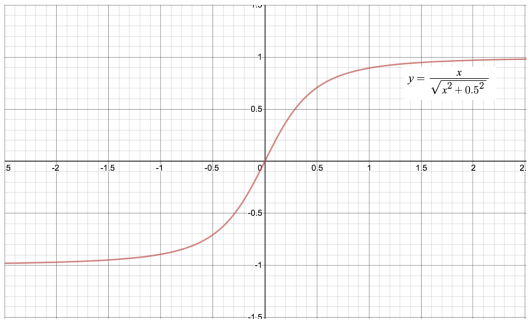


Figure 4.7 Approximation of the sigmoid function with $\alpha = 0.5$

where $\alpha \geq 0$ is a suitably chosen small positive value effecting the smoothing transformation-approximation. The effects of the approximation is demonstrated in Figure 4.6, where a small value of $\alpha = 0.5$ is used.

The implementation of this function approximation in numerical solvers and programming languages works because the function of square root only returns the positive values as the only root of it.

If I differentiate with respect to x once to obtain:

$$\frac{\partial \psi(x;\alpha)}{\partial x} = \frac{1}{2}\frac{2x}{\sqrt{x^2+\alpha^2}} = \frac{x}{\sqrt{x^2+\alpha^2}} \tag{4.36}$$

The equation has the following properties:

- As $x = 0 \Rightarrow \frac{\partial \psi}{\partial x}(0;\alpha) = 0$;
- As $x = \infty \Rightarrow \frac{\partial \psi}{\partial x}(0;\alpha) = +1^{-}$;
- As $x = -\infty \Rightarrow \frac{\partial \psi}{\partial x}(0;\alpha) = -1^{+}$.

Thus, I can directly use this function in ANN / DNN implementations ranging in output from -1 to +1:

$$\sigma_1(x;\alpha) \triangleq \frac{x}{\sqrt{x^2+\alpha^2}} \tag{4.37}$$

If I wish to scale the output between 0 and +1 then I can define:

$$\sigma_2(x;\alpha) \triangleq \frac{\sigma_1(x;\alpha)+1}{2} \tag{4.38}$$

This gives:

$$\sigma_2(x;\alpha) \triangleq \frac{1}{2}[\frac{x}{\sqrt{x^2+\alpha^2}}+1] \tag{4.39}$$

In the ANN/DNN methodology proposed, I have input-output constraints (equality constraints) of the form $z = \sigma(v)$. If I use $\sigma_1(x;\alpha)$ in our formulation, these constraints become:

$$z = \frac{v}{\sqrt{v^2+\alpha^2}} \tag{4.40}$$

As this form is fairly nonlinear and requires the use of the square root function within the NLP, as well as a division I multiply through and square the result, I obtain the following:

$$z^2(v^2+\alpha^2) = v^2 \tag{4.41}$$

One has to be careful with this transformation though: its Lagrangian multiplier may have a different value than the constraint: $z = \frac{v}{\sqrt{v^2+\alpha^2}}$ The last form is the true input-output form for the sensitivity analysis based methodology.

4.7 Computational Results

4.7.1 NLP Formulation of a Simple ANN

In Section 4.4, I have reformulated a simple 2-layered neural network with 2 inputs, 2 outputs, and 3 hidden neurons, adopting a lifting scheme. IPOPT is adopted to solve the lifted formulation and generated accurate results for a single datum point. The model is denominated as ANN-NLP-1. The model contains 25 variables with lower and upper bounds, 8 equality constraints, 34 non-zeros in equality constraint Jacobian, and 11 non-zeros in Lagrangian Hessian.

Although the model is deterministic for convex problems, since the training of ANN is a non-convex optimisation process, the model does not guarantee running to the same optima. The nature of the IPOPT solver also dictates that the initial points will lead to very different

optimisation results. Thus, the value of starting point is important in the performance of the model. Table 4.3 enlisted the weights generated from the optimisation scheme under different starting points. The training data points used have inputs of 0.1 and 0.5, and target outputs of 1 and 2.

Table 4.3 Weights obtained from the lifting scheme with different starting points. Connection 0 represents the bias term.

Layer	Neuron	Connection	Value(Init=0.25)	Value(Init=0)	Value(Init=-0.25)
1	1	1	2.20×10^{-5}	0	-0.000211
	1	2	0.000110	0	-0.00106
	1	0	0.000221	0	-0.00211
	2	1	2.201×10^{-5}	0	-0.000211
	2	2	0.000110	0	-0.00106
	2	0	0.000221	0	-0.000211
	3	1	2.201×10^{-5}	0	-0.00211
	3	2	0,000110	0	-0.00106
	3	0	0.000221	0	-0.00211
2	1	1	0.259	0	-0.250
	1	2	0.250	0	-0.250
	1	3	0.250	0	0.250
	1	0	0.250	1	-0.500
	2	1	0.500	0	-0.500
	2	2	0.500	0	-0.500
	2	3	0.500	0	-0.500
	2	0	0.500	1	0.500

With an initial value of -0.25, the weights are generated through 25 iterations with 27 objective function evaluations, 27 equality constraint evaluations, 26 gradient evaluations, and 25 Hessian evaluations. The total CPU time used is 0.011 seconds with 0.001 seconds used for NLP function evaluations. Similar results are obtained for initial value of 0.25. With an initial value of 0.25, the weights are generated through 14 iterations with 19 objective function evaluations, 19 equality constraint evaluations, 15 gradient evaluations, and 14 Hessian evaluations. The total CPU time used is 0.007 seconds with 0.000 seconds (trivial) used for NLP function evaluations.

The most complicated case to run is the case when the initial values are zeros. With an initial value of 0, the weights are generated through 125 iterations with 126 objective function evaluations, 126 equality constraint evaluations, 126 gradient evaluations, and 125 Hessian evaluations. The total CPU time used is 0.066 seconds with 0.003 seconds (trivial) used for NLP function evaluations. Although the change in CPU time is not significant, it employs

more calculations for the coefficients to deviate from zero through a more complicated fitting scheme.

An important feature of the results shows that the model produced is almost symmetric. This is a feature of the current lifted model. In the model initialised to 0.25, for example, the 2 neurons in the first layer have the same weights. The 3 neurons in the second layer have almost the same weights. This is also observed in models initialised with values of -0.25.

In the case when initial weight values are set to zero, I obtain a highly sparse model. The model effectively negates all input and makes use of a constant node to output the targeted output. This leads to a trivial model. It corroborates with the popular belief that the initial weights of ANNs cannot be zero [199] [200]. On the other hand, this may motivate future research into what the existing connections mean when the initial weights are zero.

To corroborate with the results generated from our formulation, I run the simple 2-layered neural network on Keras adopting Stochastic Gradient Descent (SGD) as the optimiser. The weights generated from the optimisation scheme are listed in Table 4.4. Since SGD employs a random search process, the weights obtained for each run are different. I present the weight values of 3 consecutive runs adopting the SGD optimiser. From Table 4.4, it is observed that the weight values vary significantly from different runs, generating different coefficients at each neuron. This indicate that the search ends at local optima thus generating different and incomparable coefficients.

I introduce Lagrangian multiplier as an indication of the sensitivity of each node to the outputs. In this case, the values found are enlisted in Table 4.5. I have obtained the Lagrangian multiplier for weights generated from different starting points (with initial values in the range {0.25, 0, -0.25}). From these values, I observe symmetry in all three models due to the symmetry observed in the weight coefficients. In particular, the model initialised with 0s exhibits no sensitivity to the inputs, which leads to a trivial model (as corroborated with the weights values). The only non-symmetric sensitivity values are close to the output where the values aligns with the targeted output.

4.7.2 Connection Sparsification through Lagrangian Multipliers

Following the procedure in Section 4.5.1, I have introduced the extra constraints adopting a small value of ε to constrain the weights values. This implements the process of sparsification to the network. The values of weightings produced are more sparse than previously calculated. This section builds up on the model ANN-NLP-1 run in the previous section and formulates the optimisation problem in ANN-NLP-2.

The program takes an iterative process to train the network. First, an ANN-NLP-1 model is implemented to obtain some initial estimation of the weight values. In this case, the

Table 4.4 The weight values obtained from fitting a neural network using Gradient Descent.

Layer	Neuron	Connection	Lifted Value	SGD Value 1	SGD Value 2	SGD Value 3
1	1	1	0.00171	-0.517	0.555	0.166
	1	2	0.00859	-0.00239	0.140	-0.400
	1	0	0.01717	-0.0214	0.00237	-0.00311
	2	1	0.00172	0.547	-0.709	-0.0785
	2	2	0.00859	-0.749	-0.172	0.752
	2	0	0.00172	0.0313	-0.00136	0.00399
	3	1	0.00859	1.072	-0.957	0.998
	3	2	0.01717	-0.108	1.000	0.872
	3	0	0.00172	-0.030	0.00288	-0.00253
2	1	1	0.25014	-0.345	0.365	-0.531
	1	2	0.25014	0.183	-0.914	0.443
	1	3	0.25014	-0.412	1.0150	0.552
	1	0	0.25014	0.348	0.0113	0.0114
	2	1	0.25038	0.620	0.833	-0.335
	2	2	0.50032	0.855	0.744	0.628
	2	3	0.50032	0.527	0.0979	-0.903
	2	0	0.50064	0.110	0.00662	0.0187

weights are initialised to 0.5 and the corresponding weights are calculated from the lifting scheme. Following that, I introduce ANN-NLP-2 which imposes the additional constraints that the value of weights should be between $-\varepsilon$ and ε where the sum of ε's should be smaller than ω.

Accordingly, I calculate the value of $\overline{W}$ with Equation 4.15 and ω with Equation 4.18. I obtain $\overline{W} = 3.0848$ and $\omega = 2.7763$. A more sparse result is produced with this initial constraints.

Following from the ANN-NLP-2 model with $\omega = \overline{W}$, I implement a second version of the ANN-NLP-2 model with $\omega = \gamma \times \overline{W}$, and a third version of the ANN-NLP-2 model with ($\omega = \gamma^2 \times \overline{W}$). This is to iteratively reduce the values the weights thus achieving sparsification of the ANN model. I arbitrarily set the value of γ to be 0.7 but any value in the range $0 < \gamma < 1$ works.

The weights obtained are enlisted in Table 4.6 for a comparison of weights obtained through the four models. It can be observed that the values are increasingly sparsified by lowering the values of ω by a rate γ.

The first model requires 77 iterations to run with a total CPU time of 0.041 seconds. The second model requires 13 iterations as the weight values are initialised to be those obtained in ANN-NLP-1 model. The total CPU time is 0.006 seconds. The third model takes 173

Table 4.5 The Lagrangian multiplier values for each constraint of the lifted problem

Neuron	Sensitivity(Init=0.25)	Sensitivity(Init=0)	Sensitivity(Init=-0.25)
$v_{1,1}$	1.467×10^{-12}	0	-3.843×10^{-12}
$v_{1,2}$	1.467×10^{-12}	0	-3.843×10^{-12}
$v_{1,3}$	1.467×10^{-12}	0	-3.843×10^{-12}
$h_{1,1}$	-2.008×10^{-9}	0	7.273×10^{-10}
$h_{1,2}$	-2.008×10^{-9}	0	7.273×10^{-10}
$h_{1,3}$	-2.008×10^{-9}	0	7.273×10^{-10}
f_1	1.338×10^{-9}	4.263×10^{-5}	4.848×10^{-10}
f_2	3.345×10^{-9}	2	1.212×10^{-9}

iterations to run and the total CPU time is 0.116 seconds. The fourth model takes 81 iterations and the total CPU time is 0.048 seconds. Overall, there is no significant trends in terms of iterations and total CPU time as the model depends more on the initial values rather than on the complexity of the model.

4.7.3 Node and Layer Removal

I further enlist the values of Lagrangian multipliers in Table 4.7 for each sparsified model.

Based on the value of Lagrangian multipliers, I can perform node removal according to the procedure described in Section 4.5.4. The value of λ is used to represent the sensitivity at each node location. To demonstrate with an example, in the simple network I have with the ANN-NLP-2($\omega = \overline{W}$) model,I calculate the following values:

$$\lambda_w = |\lambda_{1,1}| + |\lambda_{1,2}| + |\lambda_{1,3}| = 4.151 \times 10^{-8} \tag{4.42}$$

$$\lambda_{1,1} = \frac{|\lambda_{1,1}|}{\lambda_w} = 0.330 \tag{4.43}$$

$$\lambda_{1,2} = \frac{|\lambda_{1,2}|}{\lambda_w} = 0.335 \tag{4.44}$$

$$\lambda_{1,3} = \frac{|\lambda_{1,3}|}{\lambda_w} = 0.335 \tag{4.45}$$

As λ dictates the importance of a particular node in the whole neural network, the results demonstrate that the least important node is $h_{1,1}$, with a lowest λ value. Therefore, when node removal happens, this node will be preferentially removed. To remove the node, the weight associated with that node is manually set to zero. For implementation, this is achieved

Table 4.6 The weight values obtained from fitting ANN-NLP-1 and ANN-NLP-2 models

Layer	Neuron	Connection	ANN-NLP-1	ANN-NLP-2 ($\omega = \overline{W}$)	ANN-NLP-2 ($\omega = \gamma \times \overline{W}$)	ANN-NLP-2 ($\omega = \gamma^2 \times \overline{W}$)
1	1	1	0.00202	-0.00125	0.00161	-0.00108
	1	2	0.0101	-0.00627	0.00830	-0.00547
	1	0	0.0202	-0.0125	0.0160	-0.0114
	2	1	0.00202	-0.00181	-0.000529	0.00108
	2	2	0.0101	-0.00903	-0.00503	0.00547
	2	0	0.0202	-0.0181	-0.0131	0.0114
	3	1	0.00202	0.00170	0.00161	0.00108
	3	2	0.0101	0.00851	0.00805	0.00547
	3	0	0.0202	0.0170	0.0161	0.0114
2	1	1	0.250	-0.525	0.237	-0.249
	1	2	0.250	-0.158	-0.285	0.249
	1	3	0.250	0.159	0.246	0.249
	1	0	0.250	0.159	0.233	0.256
	2	1	0.500	-0.465	0.506	-0.496
	2	2	0.500	-0.519	-0.488	0.496
	2	3	0.500	0.511	0.503	0.496
	2	0	0.500	0.508	0.505	0.516

Table 4.7 The Lagrangian multiplier values for each constraint of the ANN-NLP-1 and ANN-NLP-2 models

Neuron	Sensitivity (ANN-NLP-1)	Sensitivity ($\omega = \overline{W}$)	Sensitivity ($\omega = \gamma \times \overline{W}$)	Sensitivity ($\omega = \gamma^2 \times \overline{W}$)
$v_{1,1}$	1.055×10^{-10}	-9.990×10^{-10}	4.152×10^{-9}	-2.316×10^{-8}
$v_{1,2}$	1.055×10^{-10}	-1.158×10^{-9}	-3.735×10^{-9}	2.316×10^{-8}
$v_{1,3}$	1.055×10^{-10}	1.147×10^{-9}	4.149×10^{-9}	2.316×10^{-8}
$h_{1,1}$	-2.011×10^{-9}	1.369×10^{-8}	-3.398×10^{-8}	6.695×10^{-8}
$h_{1,2}$	-2.011×10^{-9}	1.391×10^{-8}	3.337×10^{-8}	-6.695×10^{-8}
$h_{1,3}$	-2.011×10^{-9}	-1.391×10^{-8}	-3.382×10^{-8}	-6.695×10^{-8}
f_1	1.339×10^{-9}	1.377×10^{-8}	3.811×10^{-8}	8.714×10^{-8}
f_2	3.349×10^{-9}	2.073×10^{-8}	4.772×10^{-8}	9.134×10^{-8}

by setting the upper and lower bounds of the systematic variable representing the weights related to $h_{1,1}$ to 0s.

In this case, since I am experimenting with a simple network with a highly symmetrical structure, the differences in the Lagrangian multipliers is minimal for a given layer. Therefore, the effect of removal is not significant. However, I obtain the weight values when the node $h_{1,2}$ is removed, tabulated in Table 4.8.

Another example is the ANN-NLP-2 ($\omega = \gamma \times \overline{W}$) model. I calculate the following values:

$$\lambda_w = |\lambda_{1,1}| + |\lambda_{1,2}| + |\lambda_{1,3}| = 1.012 \times 10^{-7} \tag{4.46}$$

$$\lambda_{1,1} = \frac{|\lambda_{1,1}|}{\lambda_w} = 0.336 \tag{4.47}$$

$$\lambda_{1,2} = \frac{|\lambda_{1,2}|}{\lambda_w} = 0.330 \tag{4.48}$$

$$\lambda_{1,3} = \frac{|\lambda_{1,3}|}{\lambda_w} = 0.334 \tag{4.49}$$

The node to remove is $h_{1,2}$ which has the lowest Lagrangian multiplier. Therefore, I manually remove the node by setting related weights to zero. The resulting weights are enlisted in Table 4.8.

From Table 4.8, I can observe that the weight values increase when one of the nodes is removed. This is because less nodes are present to achieve the output prediction hence a larger value is required.

In either case, as this is a simple model, the prediction error (mean squared error) is zero. Therefore, there is no better model in terms of performance. Performance will be evaluated once I scale up the model. Moreover, since this is a model with 1 hidden layer, I do not perform layer removal. It is imperative to construct models of a larger size in order to further examine the effectiveness of the model.

4.7.4 NLP Formulation of a Multi-input Network

To further improve the performance of the network, it is required to incorporate additional data points to the system. This requires the reformulation of the NLP problem, achieved through the addition of constraints. This is illustrated in the following example.

Suppose I have two input data points to the neural network system, each has a dimension of 2. Similarly, I assume that there is one hidden layer of 3 neurons and an output layer of dimension 2. As both data points share the same weights, I separately define a set of variables

Table 4.8 Comparison of weights before and after node removal. L represents the layer number, N represents the neuron number, and C represents the connection number

L	N	C	ANN-NLP-2 ($\omega = \overline{W}$) (before removal)	ANN-NLP-2 ($\omega = \overline{W}$) (after removal)	ANN-NLP-2 ($\omega = \gamma \times \overline{W}$) (before removal)	ANN-NLP-2 ($\omega = \gamma \times \overline{W}$) (after removal)
1	1	1	-0.00125	0	0.00161	0.00172
	1	2	-0.00627	0	0.00830	0.00863
	1	0	-0.0125	0	0.0160	0.0173
	2	1	-0.00181	-0.00201	-0.000529	0
	2	2	-0.00903	-0.0101	-0.00503	0
	2	0	-0.0181	-0.0202	-0.0131	0
	3	1	0.00170	0.00149	0.00161	0.00137
	3	2	0.00851	0.00745	0.00805	0.0685
	3	0	0.0170	0.0149	0.0161	0.0138
2	1	1	-0.525	0	0.237	0.333
	1	2	-0.158	-0.335	-0.285	0
	1	3	0.159	0.333	0.246	0.333
	1	0	0.159	0.334	0.233	0.334
	2	1	-0.465	0	0.506	0.0667
	2	2	-0.519	-0.667	-0.488	0
	2	3	0.511	0.666	0.503	0.666
	2	0	0.508	0.668	0.505	0.669

and a set of constraints to represent the multi-input system. The input variables include $x_1^{[0]}$, $x_2^{[0]}$, $x_1^{[1]}$, and $x_2^{[1]}$. The original constraints are added as such:

$$v_{1,1}^{[0]} = W_{1,1,1} \times x_1^{[0]} + W_{1,1,2} \times x_2^{[0]} + W_{1,1,0} \tag{4.50}$$

$$v_{1,2}^{[0]} = W_{1,2,1} \times x_1^{[0]} + W_{1,2,2} \times x_2^{[0]} + W_{1,2,0} \tag{4.51}$$

$$v_{1,3}^{[0]} = W_{1,3,1} \times x_1^{[0]} + W_{1,3,2} \times x_2^{[0]} + W_{1,3,0} \tag{4.52}$$

The newly introduced constraints represent the second datum point, sharing the same weight variables W's from the previous set of constraints.

$$v_{1,1}^{[1]} = W_{1,1,1} \times x_1^{[1]} + W_{1,1,2} \times x_2^{[1]} + W_{1,1,0} \tag{4.53}$$

$$v_{1,2}^{[1]} = W_{1,2,1} \times x_1^{[1]} + W_{1,2,2} \times x_2^{[1]} + W_{1,2,0} \tag{4.54}$$

$$v_{1,3}^{[1]} = W_{1,3,1} \times x_1^{[1]} + W_{1,2,3} \times x_2^{[1]} + W_{1,3,0} \tag{4.55}$$

I observe that the input constraints double in size with the addition of one datum point. Similarly, the intermediate constraints also double in number. The original constraints are:

$$h_{1,1}^{[0]} = \sigma(v_{1,1}^{[0]}) \tag{4.56}$$

$$h_{1,2}^{[0]} = \sigma(v_{1,2}^{[0]}) \tag{4.57}$$

$$h_{1,3}^{[0]} = \sigma(v_{1,3}^{[0]}) \tag{4.58}$$

The newly added constraints include:

$$h_{1,1}^{[1]} = \sigma(v_{1,1}^{[1]}) \tag{4.59}$$

$$h_{1,2}^{[1]} = \sigma(v_{1,2}^{[1]}) \tag{4.60}$$

$$h_{1,3}^{[1]} = \sigma(v_{1,3}^{[1]}) \tag{4.61}$$

Similarly, the constraints relating to the output doubles in size.

$$f_1^{[0]} = w_{2,1,1} \times h_{1,1}^{[0]} + w_{2,1,2} \times h_{1,2}^{[0]} + w_{2,1,3} \times h_{1,3}^{[0]} + w_{2,1,0} \tag{4.62}$$

$$f_2^{[0]} = w_{2,2,1} \times h_{1,1}^{[0]} + w_{2,2,2} \times h_{1,2}^{[0]} + w_{2,2,3} \times h_{1,3}^{[0]} + w_{2,2,0} \tag{4.63}$$

$$f_1^{[1]} = w_{2,1,1} \times h_{1,1}^{[1]} + w_{2,1,2} \times h_{1,2}^{[1]} + w_{2,1,3} \times h_{1,3}^{[1]} + w_{2,1,0} \tag{4.64}$$

$$f_2^{[1]} = w_{2,2,1} \times h_{1,1}^{[1]} + w_{1,2,2} \times h_{1,2}^{[1]} + w_{2,2,3} \times h_{1,3}^{[1]} + w_{2,2,0} \tag{4.65}$$

The additional output points are reflected in the definition of the objective function.

$$\min_{w,v,h} \sum_{i=1}^{outputs=2} [f_i^{[0]} - \hat{f}_i^{[0]}]^2 + \sum_{i=1}^{outputs=2} [f_i^{[1]} - \hat{f}_i^{[1]}]^2 \tag{4.66}$$

With these newly defined variables, constraints and the objective function, it is possible to perform the calculation within a multi-input system.

I implement a system with 50 input datum points with the same neural architecture as before but without sparsification. This serves to demonstrate the effectiveness of the multi-input system. The resulting weights are tabulated in Table 4.9. I observe from the table that the weights obtained are different for single input (overfitting) and multi-input systems. The former has heavier weights towards the end layer whereas the latter has heavier weights in the second last layer.

Table 4.9 Comparison of the weights for a single-input system and a multi-input system with equivalent data points

Layer	Neuron	Connection	Single-input	Multiple-input (#data points = 50)
1	1	1	0.00171	0.00163
	1	2	0.00859	0.00813
	1	0	0.01717	0.01625
	2	1	0.00172	0.00163
	2	2	0.00859	0.00813
	2	0	0.00172	0.01625
	3	1	0.00859	0.00163
	3	2	0.01717	0.00813
	3	0	0.00172	0.01625
2	1	1	0.25014	0.35023
	1	2	0.25014	0.35023
	1	3	0.25014	0.35021
	1	0	0.25014	0.35056
	2	1	0.25038	-1.838×10^{-17}
	2	2	0.50032	-7.799×10^{-17}
	2	3	0.50032	-5.913×10^{-17}
	2	0	0.50064	-3.869×10^{-17}

4.7.5 NLP Formulation of a Larger Network

To systematically formulate a larger network requires structure and a higher level of abstraction in the definition of the available variables. It is required that the model network should be in a position to run an entire data set comprised of multidimensional vector points. The network will contain a full definition of status vector instances of the entire I/O state of the ANN. The network should have the flexibility to evaluate ever growing networks for a realistically sized dataset of multiple inputs and multiple outputs per data point. This allows the observation of the "en masse" behaviour of the neurons and how to manipulate the architectures with the new measures based on Lagrangian multipliers.

To allow this formulation, I have applied the algorithm described in Algorithm 2.

An important method to ensure that the coded algorithm is performing correctly is to check the number of equality constraints and the number of variables. To achieve this purpose, I have implemented a three layered network with 4, 10, 2 neurons in each layer and implemented it using the lifting framework. Suppose I have 10 input data points. Based on the network structure, the inputs have a dimension of 4 and the outputs have a dimension of 2.

To count the number of equality constraints:

- In Layer 1, the input layer, there are no equality constraints formed.
- In Layer 2, there are 20 equality constraints formed by each node in the layer. 10 from the linear transformation and 10 from the nonlinear transformation.
- In Layer 3, there are 2 equality constraints formed by each node in the layer. There are no nonlinear transformations in this layer as this is the output layer.
- Overall, the model contains 22 equality constraints for each datum point. For 10 data points, there should be 220 equality constraints in total.

To count the number of variables:

- The weights of the network is shared with all data points. Therefore, I only count the number of weights once. In Layer 1, there are 4 input weights and 1 bias value. Thus, there are 5 weights related to each of the 10 neurons in Layer 2. From Layer 1 to Layer 2, there are $5 \times 10 = 50$ weight values. From Layer 2 to Layer 3, there are similarly $11 \times 2 = 22$ weights. The total number of weights is 72.
- The other variables are not shared between data points. There are 10 hidden nodes with 1 additional bias, which gives 22 variables considering variables defined for linear

Algorithm 2 Formulation of ANN with Lifting Scheme

Initialize:
$List(structure\_of\_network)$
$n \rightarrow \#Layers$
$X \rightarrow Input$,
$y \rightarrow Output$,
$N \rightarrow \#DataPoints$,
$X_{dim} \rightarrow Dimension\_of\_Inputs$,
$y_{dim} \rightarrow Dimension\_of\_Outputs$,
and $\alpha \rightarrow 0$

for <l in range(n)> **do**
    for <i in range(#Neurons_in_Layer(l))> **do**
        for <j in range(#Neurons_in_Layer(l-1))> **do**
            $Define\ W_{l,i,j}$
        end for
    end for
end for

for <k in range(N)> **do**
    for <i in range(#Neurons_in_Layer(l))> **do**
        $Define\ f_{i,k}$
    end for
end for

for <k in range(N)> **do**
    for <i in range(#Input_Neurons+1)> **do**
        if <i==0> **then**
            $Define\ X_{i,k} \rightarrow 1$
        else
            $Define\ X_{i,k}$
        end if
    end for
end for

for <k in range(N)> **do**
    for <i in range(#Output_Neurons)> **do**
        $Define\ Y_{true,i,k}$
    end for
end for

41: **for** <k in range(N)> **do**
42: **for** <l in range(n)> **do**
43: **for** <i in range(#Neurons_in_Layer(l))> **do**
44: **if** <i==0> **then**
45: $Define\ X = h_{l,i,k} \rightarrow 1$
46: $Define\ X = v_{l,i,k} \rightarrow 1$
47: **else**
48: $Define\ h_{l,i,k}$
49: $Define\ v_{l,i,k}$
50: **end if**
51: **end for**
52: **end for**
53: **end for**
54:
55: $Objective\ Function = Sum(f_{i,k} - Y_{true,i,k})$
56:
57: **for** <k in range(N)> **do**
58: **for** <l in range(n)> **do**
59: **for** <i in range(#Neurons_in_Layer(l))> **do**
60: **for** <j in range(#Neurons_in_Layer(l-1))> **do**
61: **if** <l==1> **then**
62: $Constraint = Sum_{l,i,j,k}(W_{l,i,j} \times X_{i,k}) - v_{l,i,k}$
63: **else if** <l==n-1> **then**
64: $Constraint = Sum_{l,i,j,k}(W_{l,i,j} \times h_{l,i,k}) - f_{i,k}$
65: **else**
66: $Constraint = Sum_{l,i,j,k}(W_{l,i,j} \times h_{l,i,k}) - v_{l,i,k}$
67: **end if**
68: **end for**
69: **end for**
70: **end for**
71: **end for**
72:
73: **for** <l in range(n)> **do**
74: **for** <i in range(#Neurons_in_Layer(l))> **do**
75: $Constraint = h_{l,i} - v_{l,i}/Sqrt(v_{l,i}^2 + \alpha^2)$
76: **end for**
77: **end for**
78:
79: $xlist = List(Variables)$
80: $glist = List(Constraints)$
81: Input upper and lower bounds
82: Use IPOPT to optimise *xlist* and *glist*
83: Obtain variable values, Lagrangian sensitivities and objective value from optimiser

and nonlinear transformations. There are two output variables. Thus, in total, there are 24 non-weight variables for each datum point. Overall, there are 240 non-weight variables.

- In total, this three-layered network gives a total number of variables of 312.

Therefore, to confirm that the model is set up correctly, I deploy this three-layered model and check the number of variables and constraints accordingly. Once the model is set up, it has the flexibility to fit any fixed structured neural network with any number of data points. The next step is to apply the model to a simulated chemical problem in order to test its effectiveness. This is discussed in the following section.

4.8 Application to a Nonlinear Process Case Study

In previous sections, I have applied the lifting scheme to the optimisation of an arbitrarily defined problem where a 2 dimensional input is trained to give a 2 dimensional output, generating a simplified model to investigate. As the model is simplistic, I produced exact fittings to the targeted output. Moreover, the process is highly linear, giving low significance of the hidden nodes. Therefore, it is imperative to implement the lifting scheme to a highly nonlinear process such that the effectiveness of the scheme is tested. I use simulated data from a nonlinear chemical process as an example.

4.8.1 Simulated Dataset

To simulate the nonlinear process, I adopt the framework of an Arrhenius problem. The problem is formulated as follows. Suppose I have a system with inflow A and product B:

$$A \overset{K(T)}{\rightleftharpoons} B \tag{4.67}$$

I perform steady state material balance as follows:

$$-F_A \cdot C_{A0} + F_A \cdot C_A = -K(T) \cdot C_A \cdot V \tag{4.68}$$

$$-F_A \cdot 0 + F_A \cdot C_B = K(T) \cdot C_A \cdot V \tag{4.69}$$

where F_A is the inflow rate. C_{A0} is the initial concentration of substance A, C_A is the final steady state concentration of A. C_B is the steady state concentration of B. V is the volume of the reactor. The material balances state that the changes in the amount of material is equal to

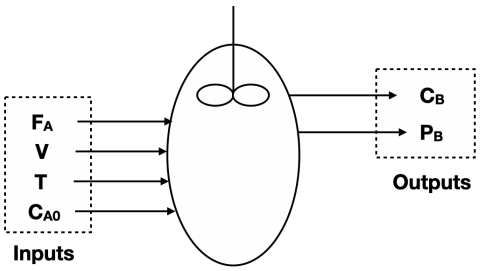


Figure 4.8 The formulation of the Arrhenius problem into a nonlinear transformation with 4 inputs and 2 outputs

those generated / consumed in the reaction. Combining Equation 4.68 and Equation 4.69 gives:

$$C_B = C_{A0} - C_A \tag{4.70}$$

$K(T)$ is the kinetic constant at temperature T, given by:

$$K(T) = K_0 \cdot exp(\frac{-E_a}{RT}) \tag{4.71}$$

Rearrange the material balance equation gives:

$$F_A \cdot C_{A0} = C_A(F_A + K(T) \cdot V) \tag{4.72}$$

Further rearrangement gives:

$$C_A = \frac{F_A \cdot C_{A0}}{F_A + K(T) \cdot V} \tag{4.73}$$

The productivity of B is formulated as:

$$P_B = F_A \cdot C_B \tag{4.74}$$

With this setting of the problem, I have formulated a simulated system with inputs T, C_{A0}, F_A and V. The outputs include C_B and P_B. The nonlinear process is encapsulated in the Arrhenius equation (Equation 4.71) and the material balance (Equation 4.73). This is further demonstrated in Figure 4.8.

As the Arrhenius system is formulated into a system of inputs and outputs undergone a nonlinear transformation, I can effectively model it as an ANN. To enable the approximation of the nonlinear process, I create a dataset that is generated from the system of nonlinear

Table 4.10 The range of variables used in the simulated Arrhenius dataset

Variable	Lower Bound	Upper Bound	Step
C_{A0}	0.05	1	0.05
V	5	50	5
T	500	2000	100
F_A	10	100	10

Table 4.11 The arbitrary constant values adopted in the simulated Arrhenius dataset

Variable	Value
R	$8.314 JK^{-1}mol^{-1}$
K_0	$2 \times 10^9 s^{-1}$
E_a	$127 kJmol^{-1}$

equations. The dataset is further used to construct the ANN model adopting the lifting scheme.

To generate the dataset, I create data points in a grid search as tabulated in Table 4.10. The range enclosed by lower and upper bounds are divided by the value of steps to obtain the grid search values. In total, I produce 43200 data points. With this number of points and given the complexity of the problem defined, it is expected that a much larger neural network is required to solve this highly nonlinear process.

As this is a simulated environment, I first settle on the values tabulated in Table 4.11. The values are arbitrarily chosen to simulate an Arrhenius problem. I generate the data according to these values and construct a network adopting the lifting scheme.

4.8.2 Network Sparsification

In this section, I present a set of methodology to sparsify a trained neural network by removing connections, nodes or layers that are undesirable. The sparsification is guided by the Lagrangian multiplier values (sensitivity) of each connection to the output. The connection, node or layer is removed by setting the boundary values of the connections involved to zero. I adopt the simulated nonlinear dataset presented in Section 4.8 to test for the efficacy of the method of sparsification.

4.8.3 Initialisation

I start from some arbitrarily sized network, which serves as the starting point. The architecture is modified in later iterations.

The initial architecture exhibiting certain degrees of freedom should be limited by the number of data points input into the neural network. In a network defined in Section 4.7.5 as an example, where there are 4, 10, 2 nodes in each layer in a three-layered model, the degrees of freedom is calculated as follows:

- In Layer 1, there are 4 input nodes and a bias. That gives 5 degrees of freedom inputting to each node in Layer 2. This gives 5×10 degrees of freedom.
- In Layer 2, there are 10 nodes and a bias. This gives 11 degrees of freedom inputting to each node in Layer 3. This gives 11×2 degrees of freedom.
- Overall, in this three-layered model, the total degrees of freedom is $50 + 22 = 72$. Therefore, the minimum number of data points should be higher than 72.

For the Arrhenius simulated problem as above, I start with an architecture of [4, 5, 5, 2]. That is, the input has dimension of 4, followed by 5 hidden nodes in the second layer, followed by 5 hidden nodes in the third layer, and lastly outputs of dimension 2. The input and output dimensions are pre-fixed by the dataset but the number of hidden layers and hidden neurons are editable. The degrees of freedom in this architecture is 67, thus requiring a minimum of 67 data points. I will calculate the Lagrangian multipliers to further make modifications to the model structure. The current architecture serves as a starting point.

4.8.4 Node Removal

The second step is to experiment with the node removal process as outlined in Section 4.5.4. I first run the unconstrained optimisation adopting the lifting scheme, and calculate the values of all Lagrangian multipliers $\lambda_{l,j}^{[k]}$ with regard to the constraints that define the nonlinear transformations in the network $h_{l,i}^{[k]} - \sigma(v_{l,i}^{[k]})$. Since there are multiple data points, there will be an equal multiple of the number of constraints related to the nonlinear transformations. I sum up the Lagrangian multipliers for the same neuron across different data points as the value $\lambda_{l,i} = \frac{1}{k}\sum_k [\lambda_{l,i}^{[k]}]$.

The percentage of each multiplier with regard to the sum of the absolute value of Lagrangian multipliers of all neurons ($\lambda_w = \sum_k \sum_l \sum_i [|\lambda_{l,i}^{[k]}|]$) are also calculated. By running the initialised model, I obtain the values of Lagrangian multipliers and percentages of the sum as enlisted in Table 4.12. The values seem equal when rounded up to 4 significant figures

Table 4.12 The values and percentages of Lagrangian multipliers adopting a structure of [4,5,5,2]

Variable	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$h_{1,1}$	1.761×10^{-11}	0.8452
$h_{1,2}$	1.761×10^{-11}	0.8452
$h_{1,3}$	1.761×10^{-11}	0.8452
$h_{1,4}$	1.761×10^{-11}	0.8452
$h_{1,5}$	1.334×10^{-10}	6.404
$h_{2,1}$	1.675×10^{-10}	8.039
$h_{2,2}$	1.675×10^{-10}	8.039
$h_{2,3}$	1.675×10^{-10}	8.039
$h_{2,4}$	1.675×10^{-10}	8.039
$h_{2,5}$	1.209×10^{-9}	58.06

but they are not equal when higher precision is recorded. The objective function for this run has a value of 4.988×10^{-16}, which is a very low value potentially due to over-fitting the network. To prevent over-fitting, I have performed node removal as a follow-up.

From the results, I observe that most nodes have a low value of percentage Lagrangian sensitivities. 4 out of 5 nodes in the first layer has a value smaller than 1. In the second layer, only the last neuron has high significance. The other 4 nodes are equally less important.

Although I can argue that the first layer can be removed completely due to lower sensitivities, I stay conservative by keeping 3 nodes in the first layer and keeping 3 nodes in the second layer to observe the changes in Lagrangian multiplier. Again, I tabulate the raw values and the percentages of the Lagrangian multiplier in Table 4.13. In this run, the value of the objective function is 1.711×10^{-5}. The objective has gone down as I have introduced node removal to cut down on the number of parameters in making a prediction.

From the updated values of Lagrangian multiplier, I observe that 2 nodes in the first layer are significant and the third neuron having a very low sensitivity can be removed. Similarly in the second layer, the values are more balanced but I can remove the first neuron due to its low sensitivity. It is advised not to cut the number of nuerons in the second layer too low to prevent the introduction of bottleneck in the network. The bottleneck will undesirably compress the dimension introducing larger errors. Therefore, as a next step, I keep 2 nodes in the first hidden layer and 2 nodes in the second hidden layer, adopting a structure of [4,2,2,2]. I tabulate the values of Lagrangian multipliers in Table 4.14.

From the values, I can observe that the first layer contains two significant neurons, sharing an almost equally high percentage in terms of sensitivity. The neurons in the second layer are

Table 4.13 The values and percentages of Lagrangian multipliers adopting a structure of [4,3,3,2]

Variable	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$h_{1,1}$	4.717×10^{-9}	37.52
$h_{1,2}$	3.994×10^{-9}	31.77
$h_{1,3}$	2.837×10^{-14}	0.0002257
$h_{2,1}$	8.016×10^{-10}	6.377
$h_{2,2}$	1.936×10^{-9}	15.40
$h_{2,3}$	1.123×10^{-9}	8.923

Table 4.14 The values and percentages of Lagrangian multipliers adopting a structure of [4,2,2,2]

Variable	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$h_{1,1}$	3.646×10^{-9}	38.11
$h_{1,2}$	4.068×10^{-9}	42.52
$h_{2,1}$	8.769×10^{-10}	9.164
$h_{2,2}$	9.771×10^{-10}	10.21

Table 4.15 The values and percentages of Lagrangian multipliers adopting a structure of [4,2,2,2,2]

Layer Number	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
*Layer*1	2.228×10^{-23}	1.984×10^{-9}
*Layer*2	5.146×10^{-16}	0.04581
*Layer*3	1.123×10^{-12}	99.95

less significant but the sensitivity values are acceptably high. Therefore, I have performed node removal by adopting Lagrangian multiplier values. The finalised architecture has a structure of [4,2,2,2] with 22 degrees of freedom, much less compared to the initial network with 72 degrees of freedom.

More importantly, the value of the objective function of the network after neuron removal is even smaller, with a value of 8.320×10^{-18}. This demonstrates that the former models may have been over-parameterised and the network after node removal gives better performance.

4.8.5 Layer Removal

I also experiment with the theories developed in Section 4.5.5, where I decide whether an entire layer shall be removed from the architecture. Suppose I start with a network having layers more than necessary, *i.e.* I start with an architecture of [4,2,2,2,2], having three hidden layers. The number of neurons adopted are carried over from the previous section. To find whether a layer should be removed, I calculate the sum of the absolute values of Lagrangian multipliers for each layer, and delete a layer when the sum is small compared to other layers. By searching in the space of the defined structure, I obtain the results as tabulated in Table 4.15.

The objective function achieved has a value of 1.509×10^{-18}, a very low and satisfactory result. From the results, it is obvious that layer 1 and layer 2 can be removed. Therefore, I train a network with 1 hidden layer and 2 neurons and observe that the value of objective function has increased to 9.242×10^{-11}, which although has increased in value, still preserves acceptable sensitivity.

4.8.6 Connection Sparsification

I further develop the empirical implementation of the theories described in Section 4.5.1, where certain connections within the network are removed to generate a more compact model. This is different from node removal as only certain connections are removed but not the

whole node. Therefore, it entails the network with more flexibility as certain nodes may be connected to some but not all nodes.

The process of connection sparsification is achieved by enforcing an upper (ε) and lower bound ($-\varepsilon$) on the individual network weights. Moreover, the sum of ε's are controlled by an upper limit of the value of ω. The value of ω successively decreases by a factor of γ, enforcing tighter bounds as the process iterates. With the additional constraints, some of the weights are pushed closer to 0. I then define a threshold to decide which connection to remove as weight values smaller than the threshold are forced to 0's. This is achieved by manually setting the upper and lower bound of that weight value to 0's, thus effectively removing the connection associated with that particular weight.

Suppose I start from a model with an initial structure of [4,2,2,2,2]. I calculate the Lagrangian multiplier for each connection. This is tabulated in Table 4.16. The objective value of this network is 1.059×10^{-18}.

Assume that I would like to remove around 45% of the connections (13 connections). Therefore, a suitable threshold in terms of percentages of Lagrangian multipliers is 10^{-6}. To surgically remove the connections, I set the upper and lower bound of the variable values corresponding to the connections to be 0. The results are tabulated in Table 4.17.

In this way, I have successfully removed connections in the network. Although the method is similar to the removal of nodes and layers, it allows a surgical removal at a much lower level, allowing more choices in sparsification.

I also observe from Table 4.17 that the Lagrangian values have equilibrated in the process of connection removal, with more connections having similar Lagrangian multiplier values. The objective value of the sparsified network is 3.952×10^{-16}. Although slightly higher than the unsparsified network, both values are close to zero and the sparsified network contains less parameters to calculate.

4.9 Advanced Formulation of ANNs

The purpose of this section is to demonstrate how an ANN, in its most general form, can be explicitly formulated as an NLP to achieve its fitting process. Lagrangian multipliers are used to identify the most important and the least important nodes in a given fitted ANN using LSQR. The section also introduces auxiliary techniques to sparsify a densely connected ANN defined through a bi-objective optimisation problem. The proposed optimisation scheme is able to dictate any level of complexity of an ANN so as to enhance performance and predictive ability. In this section, I use a simple initial structure to preliminarily fit an ANN, and use its information to add strategically new nodes and connections selectively.

Table 4.16 The values and percentages of Lagrangian multipliers adopting a structure of [4,2,2,2,2] for connection sparsification

Layer	Neuron	Connection	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
1	1	1	1.290×10^{-22}	1.348×10^{-5}
	1	2	5.948×10^{-25}	6.212×10^{-8}
	1	3	-7.867×10^{-23}	8.217×10^{-6}
	1	4	-5.397×10^{-25}	5.637×10^{-8}
	1	0	2.327×10^{-40}	2.430×10^{-23}
	2	1	4.872×10^{-16}	50.885
	2	2	4.629×10^{-17}	4.834
	2	3	-1.288×10^{-24}	1.346×10^{-7}
	2	4	-6.248×10^{-27}	6.526×10^{-10}
	2	0	1.030×10^{-16}	10.756
2	1	1	-7.078×10^{-22}	7.393×10^{-5}
	1	2	-1.049×10^{-18}	0.110
	1	0	7.866×10^{-25}	8.237×10^{-8}
	2	1	8.299×10^{-41}	8.668×10^{-24}
	2	2	1.061×10^{-22}	1.108×10^{-5}
	2	0	-1.008×10^{-20}	0.00105
3	1	1	-1.735×10^{-23}	1.813×10^{-6}
	1	2	2.403×10^{-25}	2.510×10^{-8}
	1	0	3.423×10^{-25}	3.358×10^{-8}
	2	1	1.995×10^{-16}	20.834
	2	2	5.070×10^{-17}	5.295
	2	0	6.903×10^{-17}	7.210
4	1	1	-2.093×10^{-28}	2.187×10^{-11}
	1	2	1.815×10^{-25}	1.895×10^{-8}
	1	0	-1.059×10^{-24}	1.106×10^{-7}
	2	1	-7.030×10^{-23}	0.0734
	2	2	-9.919×10^{-21}	0.00104
	2	0	-4.745×10^{-22}	4.956×10^{-5}

The Lagrangian Multiplier Node Significance criterion (sensitivities) will be used to add / delete nodes and connections. Practically, I am treating the ANN by an unstructured general information flow graph.

Table 4.17 The values and percentages of Lagrangian multipliers adopting a structure of [4,2,2,2,2] for further connection sparsification

Layer	Neuron	Connection	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
1	1	1	-6.839×10^{-15}	8.433×10^{-6}
	1	2	0	0
	1	3	-1.023×10^{-13}	0.000126
	1	4	0	0
	1	0	0	0
	2	1	5.394×10^{-8}	66.512
	2	2	-1.233×10^{-11}	0.0152
	2	3	0	0
	2	4	0	0
	2	0	0	0
2	1	1	8.182×10^{-12}	0.0101
	1	2	6.764×10^{-12}	0.00834
	1	0	0	0
	2	1	0	0
	2	2	-6.839×10^{-15}	8.433×10^{-6}
	2	0	4.836×10^{-11}	0.0596
3	1	1	-1.023×10^{-13}	0.000126
	1	2	0	0
	1	0	0	0
	2	1	2.697×10^{-8}	33.257
	2	2	-1.233×10^{-11}	0.0152
	2	0	-3.581×10^{-11}	0.0442
4	1	1	0	0
	1	2	0	0
	1	0	0	0
	2	1	6.764×10^{-12}	0.00834
	2	2	4.836×10^{-11}	0.0596
	2	0	8.182×10^{-12}	0.0101

4.9.1 Feedback ANNs Example

SISO Example

An example of ANN is shown in Figure 4.9. The task is to build an ANN functional representation, $y = y(x)$ for a single-input, single-output (SISO) system given a usual dataset of measurements: $D = \{(x_i, y_i)|i = 1, 2, ..., N_D\}$. The system as presented has "novel" features:

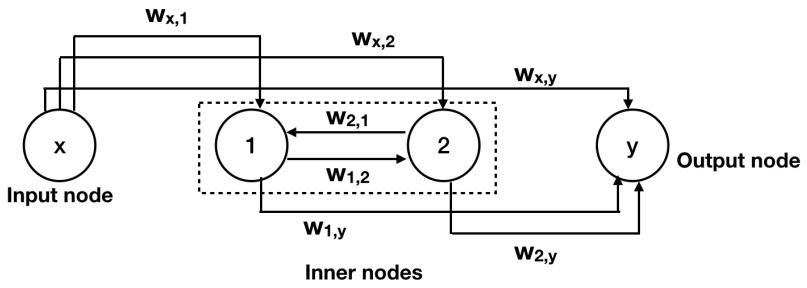


Figure 4.9 The structure of a simple ANN in the SISO Example

- The input nodes can feed into every interior ANN node
- The output nodes can similarly combine the output of every interior neuron (including a direct contribution by the input nodes)
- Feeding a node into itself is not considered

To formulate the NLP model, I note that this is a standard I/O network flow problem, with nonlinear "generation" terms of the "flow" (the information flow). The NLP for this example is the balance of the Input/Output flow along arcs into each node.

Node 1:
Input: $v_1^{[k]} = w_{x,1} \cdot x^{[k]} + w_{2,1} \cdot h_2^{[k]}$
Output: $h_1^{[k]} = \sigma(v_1^{[k]})$
where σ is the activation function such as the Sigmoid function.

Node 2:
Input: $v_2^{[k]} = w_{x,2} \cdot x^{[k]} + w_{1,2} \cdot h_1^{[k]}$
Output: $z_2^{[k]} = \sigma(v_2^{[k]})$

Node y:
$\hat{y}^{[k]} = w_{x,y} \cdot x^{[k]} + w_{1,y} \cdot h_1^{[k]} + w_{2,y} \cdot h_2^{[k]}$

Clearly, a fully connected two-way network, although feasible to model as above, will contain a very dense connectivity with $O(N^2)$ weights $w_{i,j}$ to fit, where N is the number of nodes (neurons).

Circular Feedback Network Example

I then construct another example architecture as shown in Figure 4.10, where each node is connected with neighbours of distance up to 2. The input is at node 1 and the output is at node 6. This can also be easily formulated under the lifting scheme as follows:

Node 1: Input: $v_1^{[k]} = w_{x,1} \cdot x^{[k]} + w_{2,1} \cdot h_2^{[k]} + w_{3,1} \cdot h_3^{[k]} + w_{5,1} \cdot h_5^{[k]} + w_{6,1} \cdot h_6^{[k]}$

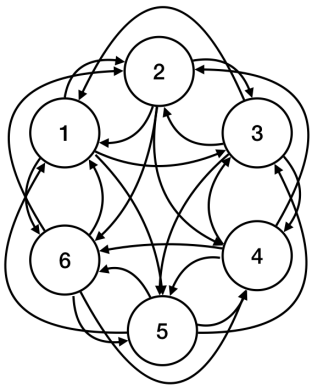


Figure 4.10 An example of the initial ANN structure

Output: $h_1^{[k]} = \sigma(v_1^{[k]})$
Node 2:
Input: $v_2^{[k]} = w_{1,2} \cdot h_1^{[k]} + w_{3,2} \cdot h_3^{[k]} + w_{4,2} \cdot h_4^{[k]} + w_{6,2} \cdot h_6^{[k]}$
Output: $z_2^{[k]} = \sigma(v_2^{[k]})$
Node 3:
Input: $v_3^{[k]} = w_{1,3} \cdot h_1^{[k]} + w_{2,3} \cdot h_2^{[k]} + w_{4,3} \cdot h_4^{[k]} + w_{5,3} \cdot h_5^{[k]}$
Output: $z_3^{([k])} = \sigma(v_3^{[k]})$
Node 4:
Input: $v_4^{[k]} = w_{2,4} \cdot h_2^{[k]} + w_{3,4} \cdot h_3^{[k]} + w_{5,4} \cdot h_5^{[k]} + w_{6,4} \cdot h_6^{[k]}$
Output: $z_4^{[k]} = \sigma(v_4^{[k]})$
Node 5:
Input: $v_5^{[k]} = w_{1,5} \cdot h_1^{[k]} + w_{3,5} \cdot h_3^{[k]} + w_{4,5} \cdot h_4^{[k]} + w_{6,5} \cdot h_6^{[k]}$
Output: $z_5^{[k]} = \sigma(v_5^{[k]})$
Node 6:
Input: $v_2^{[k]} = w_{1,6} \cdot h_1^{[k]} + w_{2,6} \cdot h_2^{[k]} + w_{4,6} \cdot h_4^{[k]} + w_{5,6} \cdot h_5^{[k]}$
Output: $z_6^{[k]} = \sigma(v_6^{[k]})$
Output: $\hat{y}^{[k]} = z_6^{[k]}$

4.9.2 ANN Gradual Evolution Scheme

I propose a gradual evolution scheme where the ANN changes its size adopting the sensitivity measure as the criterion. The ANN here is defined as its most general form, where the traditional layered structure is no longer adopted. Start with a simple structure, noting:

- There is no "official" designation or use of the concept of layers as is traditionally done /necessary with the old restricted approaches.

- There is similarly no input or output "layer" distinction
- It is possible for the input variables to feed into fewer nodes than their number, and the same for the output variables

An example is illustrated in Figure 4.10 where an initial ordered ANN structure is defined. Essentially, each neuron connects to its neighbourhood with a distance of below 2, as illustrated in Figure 4.11. This repeats for all neurons and closes them in a circular interlinked but very sparsely connected graph. I propose the following procedures to optimise

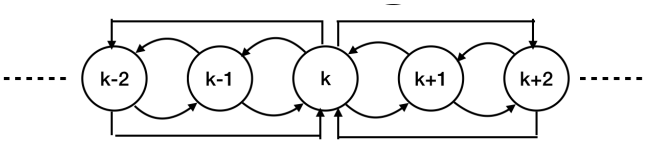


Figure 4.11 Illustration of the connectivity of the neuron k with its neighbourhood

an ANN:

- Step 1: Rank all neurons/connections, after LSQR fitting using the sensitivity measures for them based on Lagrangian multipliers.
- Step 2: Dropping nodes: nodes/connections that are found to be of very low impact sensitivity index can be removed completely.
- Step 3: The highest ranking in sensitivity nodes are selected (e.g. 5%)
- Step 4: Ensure that the subset of the most important in rank nodes are now parallel with full connections between each on of them (unless already connected)
- Step 5: Initialise the new link weights and use for the other connectivity the optimal fitting values for the weights from the previous LSQR.
- Step 6: Introduce alternative new connectivity and new node. Clone the nodes/connections with top ranking by duplicating a node/connection near the original node, sharing the same nodes to connect to but use general feedback for the connections introduced.
- Step 7: Refitting policies: (A) Fix the old connections to their weights from before and only optimise the new model connections. (B) Following Step (A) one could re-optimise and sparsify everything in the given connectivity from (A).

Overall, the scheme can rapidly build up ANNs and in more than one way to produce optimal architectures as "self-evolving" general graphs. The key idea is that after deletion of weak nodes and addition of strong nodes, I may choose to fit the previous architecture and weights (as constants in the next new LSQR problem for the new, revised architecture) and not wasting time refitting old weights.

It is assumed that the loss of predictive ability will more than be compensated and corrected by the new "strong nodes" introduced and fitted.

4.9.3 Results and Analysis

SISO example

I construct the architecture in the SISO Example. This can be simply formulated under the lifting scheme by changing the equality constraints. The initial weights obtained are tabulated in Table 4.18. The initial objective value is 3.610×10^{-19}.

Table 4.18 The values and percentages of Lagrangian multipliers adopting a structure in the SISO Example before connection sparsification

Weights	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$W_{x,1}$	8.651×10^{-12}	1.384
$W_{2,1}$	-1.195×10^{-10}	19.112
$W_{x,2}$	8.658×10^{-12}	1.385
$W_{1,2}$	-1.199×10^{-10}	19.178
$W_{x,y}$	2.403×10^{-10}	38.445
$W_{1,y}$	8.651×10^{-12}	1.384
$W_{2,y}$	-1.195×10^{-10}	19.112

I observe that, since this is a simple network, all the weights are quite important. To further sparsify the network, I remove weights where the percentage values of Lagrangian multiplier are less than 10. However, to keep the connection complete, I retain the connection between x and node 1 to allow a full network. The complete network resulted is demonstrated in Figure 4.12.

The Lagrangian values after sparsification are demonstrated in Table 4.19. The value of the objective function after sparsification is 5.982×10^{-17}. Although slightly higher due to the removal of nodes, the objective is acceptably low.

Then I seek to grow the network by adding nodes that have the highest sensitivity. From Table 4.19, it can be observed that $W_{1,2}$ and $W_{x,y}$ have the highest sensitivity, hence I duplicate

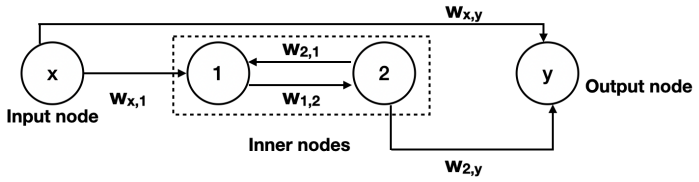


Figure 4.12 The feedback network sparsified from the structure in the SISO Example

Table 4.19 The values and percentages of Lagrangian multipliers adopting a structure in the SISO Example after connection sparsification

Weights	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$W_{x,1}$	-7.057×10^{-31}	1.174×10^{-20}
$W_{2,1}$	7.064×10^{-34}	1.175×10^{-23}
$W_{x,2}$	0	0
$W_{1,2}$	-2.918×10^{-9}	48.541
$W_{x,y}$	3.094×10^{-9}	51.459
$W_{1,y}$	0	0
$W_{2,y}$	4.125×10^{-34}	6.860×10^{-24}

the connections. This gives us the architecture in Figure 4.13. The objective value obtained is 1.847×10^{-17}. There is a slight improvement from the architecture before network growth.

The Lagrangian values after network growth are tabulated in Table 4.20. It can be observed that the addition of weights greatly changes the percentage sensitivities of different nodes.

Overall, I have demonstrated how the gradual growth scheme works. It can be observed that the sparsification scheme is capable of removing connections of low sensitivity and adding connections of high sensitivity, while generating very low objective values.

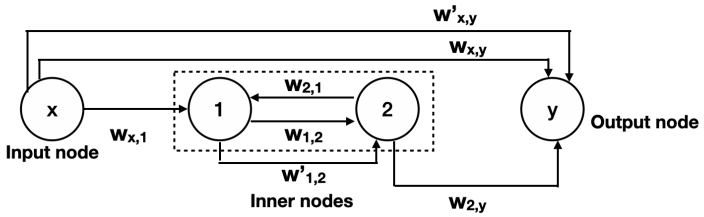


Figure 4.13 The feedback network grown from the sparsified structure in the SISO Example

Table 4.20 The values and percentages of Lagrangian multipliers adopting a structure in the SISO Example after further connection sparsification

Weights	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$W_{x,1}$	2.243×10^{-29}	5.903×10^{-19}
$W_{2,1}$	-2.242×10^{-33}	5.902×10^{-22}
$W_{x,2}$	0	0
$W_{1,2}$	-1.210×10^{-9}	31.859
$W_{x,y}$	1.378×10^{-9}	36.282
$W_{1,y}$	0	0
$W_{2,y}$	-2.243×10^{-32}	5.904×10^{-22}
$W'_{1,2}$	1.210×10^{-15}	3.186×10^{-5}
$W'_{x,y}$	-1.210×10^{-9}	31.859

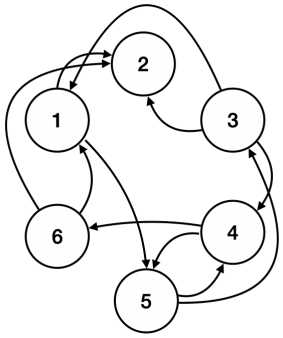


Figure 4.14 Sparsified circular feedback network

Circular Feedback Network

I return to the example of circular feedback network. The initial Lagrangian multiplier values are tabulated in Table 4.21. The value of the objective obtained is 3.20. I aim to drop 50% of the connections (13 connections). The threshold I set is 1% in terms of percentage value of Lagrangian multiplier.

The Lagrangian multipliers of the sparsified circular feedback network is presented in Table 4.22. The new architecture is demonstrated in Figure 4.14. The new objective function value is 3.174, albeit that 50% of the connections are removed. This demonstrates an improvement in terms of the performance of the network after sparsification.

To apply the gradual growth scheme, I aim to duplicate 1 node. The node with the highest value of Lagrangian multiplier by summing up all connection Lagrangian values is node 1. The Lagrangian multipliers after node growth is demonstrated in Table 4.23. The new

Table 4.21 The values and percentages of Lagrangian multipliers adopting a structure of Circular Feedback Network before connection sparsification

Weights	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$W_{x,1}$	3.362×10^{-19}	9.787
$W_{2,1}$	-1.630×10^{-20}	0.474
$W_{3,1}$	-3.477×10^{-19}	10.121
$W_{5,1}$	1.569×10^{-20}	0.457
$W_{6,1}$	6.397×10^{-20}	1.862
$W_{1,2}$	-4.294×10^{-20}	1.245
$W_{3,2}$	-3.477×10^{-19}	10.121
$W_{4,2}$	1.630×10^{-20}	0.474
$W_{6,2}$	3.477×10^{-19}	10.121
$W_{1,3}$	-1.569×10^{-20}	0.457
$W_{2,3}$	-5.141×10^{-23}	0.001
$W_{4,3}$	3.157×10^{-20}	0.919
$W_{5,3}$	-3.240×10^{-19}	9.430
$W_{2,4}$	1.616×10^{-20}	0.470
$W_{3,4}$	3.479×10^{-19}	10.126
$W_{5,4}$	-1.527×10^{-20}	0.445
$W_{6,4}$	3.092×10^{-20}	0.900
$W_{1,5}$	3.480×10^{-19}	10.126
$W_{3,5}$	-1.616×10^{-20}	0.470
$W_{4,5}$	-3.479×10^{-19}	10.126
$W_{6,5}$	-1.527×10^{-20}	0.445
$W_{1,6}$	5.510×10^{-23}	0.002
$W_{2,6}$	-3.131×10^{-20}	0.911
$W_{4,6}$	-3.445×10^{-19}	10.028
$W_{5,6}$	-1.630×10^{-20}	0.474

objective value is 3.172, with a slight improvement. The updated network architecture is demonstrated in Figure 4.15.

In this section, I have performed sparsification and implemented the gradual growth scheme on two example architectures: SISO example and circular feedback network. Although only two example networks are presented, this section demonstrates the unlimited possibility of implementing the lifting scheme on a variety of feedback networks or other ANNs in their most general form. This testifies the flexibility of ANNs to be formulated under the lifting scheme and the effectiveness of the sparsification and gradual growth process to optimise the architecture of ANNs.

Table 4.22 The values and percentages of Lagrangian multipliers adopting a structure of Circular Feedback Network after connection sparsification

Weights	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$W_{x,1}$	-3.020×10^{-6}	13.867
$W_{2,1}$	0	0
$W_{3,1}$	-1.228×10^{-9}	0.006
$W_{5,1}$	0	0
$W_{6,1}$	5.865×10^{-7}	2.693
$W_{1,2}$	-7.328×10^{8}	0.336
$W_{3,2}$	2.531×10^{-8}	0.116
$W_{4,2}$	0	0
$W_{6,2}$	1.421×10^{-9}	0.007
$W_{1,3}$	0	0
$W_{2,3}$	0	0
$W_{4,3}$	0	0
$W_{5,3}$	-1.296×10^{-11}	5.950×10^{-5}
$W_{2,4}$	0	0
$W_{3,4}$	-3.616×10^{-15}	1.660×10^{-8}
$W_{5,4}$	3.616×10^{-18}	1.660×10^{-11}
$W_{6,4}$	0	0
$W_{1,5}$	-1.868×10^{-8}	0.086
$W_{3,5}$	0	0
$W_{4,5}$	-9.536×10^{-10}	0.004
$W_{6,5}$	0	0
$W_{1,6}$	0	0
$W_{2,6}$	0	0
$W_{4,6}$	-1.805×10^{-5}	82.885
$W_{5,6}$	0	0

4.10 Summary

This chapter proposed an NLP-based idea for the fitting of ANN / DNN adopting the lifting scheme. To summarise, the method entails the following advantages:

- The method has been demonstrated to show better speed than other methods.
- The constraints comprising the ANN / DNN model are only satisfied at the solution of NLP.

Table 4.23 The values and percentages of Lagrangian multipliers adopting a structure of Circular Feedback Network after further connection sparsification

Weights	Lagrangian Multiplier (Raw Value)	Lagrangian Multiplier (Percentage %)
$W_{x,1}$	-3.512×10^{-11}	5.342
$W_{2,1}$	0	0
$W_{3,1}$	7.947×10^{-12}	1.209
$W_{5,1}$	0	0
$W_{6,1}$	-7.724×10^{-12}	1.175
$W_{1,2}$	1.142×10^{-11}	1.737
$W_{3,2}$	1.225×10^{-10}	18.639
$W_{4,2}$	0	0
$W_{6,2}$	-6.966×10^{-11}	10.595
$W_{1,3}$	0	0
$W_{2,3}$	0	0
$W_{4,3}$	0	0
$W_{5,3}$	4.189×10^{-11}	6.372
$W_{2,4}$	0	0
$W_{3,4}$	3.294×10^{-12}	0.501
$W_{5,4}$	-1.160×10^{-14}	0.002
$W_{6,4}$	0	0
$W_{1,5}$	1.860×10^{-10}	28.292
$W_{3,5}$	0	0
$W_{4,5}$	-1.042×10^{-10}	15.852
$W_{6,5}$	0	0
$W_{1,6}$	0	0
$W_{2,6}$	0	0
$W_{4,6}$	6.201×10^{-11}	9.431
$W_{5,6}$	0	0
$W'_{2,1}$	0	0
$W'_{3,1}$	8.194×10^{-15}	0.001
$W'_{5,1}$	0	0
$W'_{6,1}$	5.601×10^{-12}	0.852

- Lifting trivialises the non-linearity of all constraints so even symbolic differentiation would be just as efficient for gradient evaluation.
- Interior point methods can deal easily with NLP's of $O(10^6)$ variables and constraints

The second part of the chapter proposed a fitting scheme that can be used to obtain the optimal architecture of ANNs. It achieves the following targets:

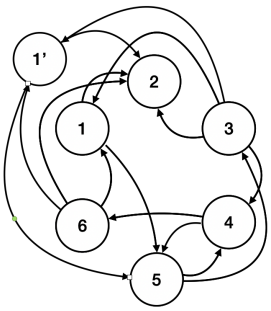


Figure 4.15 The architecture of the circular feedback network after connection sparsification and node growth

- Rapid, novel evolution of the ANN architecture in a completely free and deterministic way, emulating neuronal real tissue growth including pruning and learning.
- The very simple steps proposed are a kind of "survival of the fittest scheme" and producing "offspring" (the cloned strong important nodes identified) to improve the network.
- The end result will be the self-evolution of ANN architectures consisting of successive neuron / connection addition and removal .

In summary, this chapter proposed a novel, unique and quantifiable methodology to rigorously analyse and modify ANN architectures and connectivity. The method is novel and unprecedented in open literature. It has been demonstrated that the method is capable of sparsifying both feedforward neural networks and ANN in its most general form.

For future research, it can be expected that the final ANN exhibits a "Small World Network" structure, as in the arrangement of neurons in the real tissues of living organisms. Therefore, it is possible to explore the relationship between a standard "Small World Network" and our optimised ANN architecture.

Chapter 5

Autonomous Learning ANN - *Part II*

5.1 Introduction

In Chapter 4, I have briefly conducted a preliminary analysis of the process of neural architecture search based on sensitivity values. The analysis involves the removal of a connection, node or layer by setting the boundary values of the equivalent optimisation problem to zero. The process simulates L1-regularisation and is developed to achieve sparsification to arrive at a smallest-possible network. Although this preliminary exploration is effective, this chapter further develops on this idea by allowing both addition and removal of connections, nodes and layers, built as subroutines. This allows a dynamic interaction between the architecture of a network and the value of the objective function. The modification is no longer through the setting of the boundary values. The connections, nodes and layers are added from zero or removed completely instead. With these subroutines, I build the foundation of an automatically-evolving system where the architecture of the system can evolve freely with an understanding of the sensitivity values.

The chapter is organised as follows:

- Section 5.2 describes 6 processes to alter the architecture of the ANN acting as subroutines, including the addition/removal of connections, nodes or layers. I also describe the process of calculating the sensitivity values and discuss the optimisation options.
- Section 5.3 provides a method to find the neuron sensitivities through automatic differentiation.
- Section 5.4 describes the use of finite difference method to evaluate network sensitivities.

- Section 5.5 describes the algorithm to perform autonomous architecture evolution and uses 2 case studies to demonstrate the effectiveness of the algorithm. It further compares the architecture to backpropagation ANN.
- Section 5.6 then concludes the chapter and discusses future work.

5.2 Subroutines for the Auto-optimisation of Network Structure

In this section, I describe 6 subroutines available for the development of an autonomous structure-evolving network, including connection addition/removal, node addition/removal and layer addition/removal. The ideas behind the construction of these subroutines and the pseudocode are presented. These subroutines build up the foundation of an autonomous system where a systematic evolution of the network structure is possible.

5.2.1 Connection Removal

In Section 4.8.6, I have removed connections between nodes by setting the upper and lower bound to zeros. In this way, although the connections are effectively removed, the variables relating to the definition of the neurons are still present. In this section, I describe a method that removes a specific connection in the neural network with the corresponding variables completely removed from the system. This effectively clears out variables to achieve a smaller storage with dynamic memory allocation.

To effectively control the addition and removal of connections, I employ an internal index system where each connection is assigned to an index with value of either 0 or 1. When the value is 1, the connection is still present. When the value is 0, the connection has effectively been removed. The internal indexing system facilitates the management of connections and aids the process of determining whether a node is still present when certain connections to the node has been removed. This is different to the traditional method of architecture optimisation where a node is either dropped out or kept. In our case, I can only remove certain connections but not the whole node. The internal indexing helps to control when a connection can be removed or added.

For connection removal, I follow the sequence of ideas defined below:

- Step 1: Check if this is the last in-coming or out-going connection in the node. If yes, I remove the node by removing node-related variables and all variables related to the out-going and in-coming connections.

- Step 2: If this is not the last in-coming or out-going connection, I only remove the connection itself and parts of the constraint equations that contain the removed connection.

The internal indexing system is useful when checking whether the removed connection is the last in-coming or out-going connection. This is achieved by summing up the indexes of the connections related to the neuron of interest and checking if the sum equals to 1.

Even if the node of interest still contains several out-going connections, if there is no in-coming connections, the node will still be removed. This is because a node without in-coming connections is not going to have a value in the network. Similarly, a node containing no out-going connections will not participate in the production of results hence the node can be removed completely despite there are multiple in-coming connections.

I define the process of removal in more detail in the pseudocode in Algorithm 3.

Algorithm 3 Removal of a Connection

```
procedure CONNECTION_REMOVAL(Layer, Node, Connection, Internal_index, Network_Structure)
    for <all data points> do
        if <Last In-coming/ Out-going Connection> then
            Internal_index[connection]=0
            Remove all connections of the node
            Remove all parts of the constraints containing the node variables
            Remove all variables (W's, h's, and v's) concerning the node
        else
            Internal_index[connection]=0
            Remove all parts of the constraints containing the node variables
            Remove the connection variable (W's)
        end if
    end for
end procedure
```

5.2.2 Connection Addition

In the previous section, I have dealt with the removal of a connection. This section focuses on the addition of a particular connection. Addition is more complicated compared to removal, as removal only requires removing the corresponding connection variables, thus manually removing a particular connection associated with that node. For addition of a connection, however, I follow the following logic: suppose I would like to add connection $W_{l,i,j}$ where l is the layer number, i is the node the connection is connected to (the in-coming node), and j

is the node where the out-going connection is starting from (the out-going node). This is the same definition as in Section 4.4.

- If the in-coming node $v_{l,i}^{[k]}$ already exists, this means that the connection is added on to the original connection with other neurons.
- Next, I check whether the connection and weights associated with the connection already exist.
- If the weight is non-empty, that means there was a connection originally but was removed in the connection removal step, through the imposition of a constraint on the upper and lower bounds. To add this connection, I simply relax that constraint and set the upper and the lower bounds to nonzero values.
- If the weight does not exist, that means there has been a node present in the in-coming layer but this in-coming node is not connected to the out-going node. Therefore, I define the weights associated with the connection and add the product of the weights and the out-going node into the calculation of the constraints of the in-coming node.
- If the in-coming node does not exist before making the connection, I need to define $h_{l,i}^{[k]}$ and $v_{l,i}^{[k]}$ as the new variables associated with the node. I also define the value of $W_{l,i,j}$ once for all. I also add in new constraints that define the values of $v_{l,i}^{[k]}$ and $h_{l,i}^{[k]}$.
- It is noteworthy that the whole connection construction process is repeated for the number of data points. Except the values of Ws which are the same for different data points, the other values (v and h) are idiosyncratic to the data point and thus are separately defined for each data point.
- It is also noteworthy that the structure of the network is changed during the process, hence the variables concerning the network architecture are also changed in the connection addition process.

The pseudocode of the connection addition process is demonstrated in Algorithm 4.

5.2.3 Node Removal

Different to Section 4.8.4, the node removal described in this section completely removes a node and its corresponding constraints and variables. The internal indexing system is important in this case, as it controls which connection has already been removed and which

Algorithm 4 Addition of a Connection

```
procedure CONNECTION_ADDITION(Layer, Node, Connection, Internal_index, Network_Structure)
Internal_index[connection]=1
    for <all data points> do
        if <Node exists> then
            if <Weights already exists> then
                Relax upper and lower bounds.
            else
                Add connection (W's) and relevant constraints
            end if
        else
            Add a new node in Network Structure
            Add corresponding variables (W's, v's, h's)
            Add corresponding constraints
        end if
    end for
end procedure
```

connection is still present. To completely remove the node, I remove all in-coming and out-going connections corresponding to the node.

I following the logic below to perform node removal:

- Step 1: The validity of the node number to be removed is checked. This excludes nodes that are non-existent in the network or the bias term. Moreover, nodes in input and output layers cannot be removed.
- Step 2: Update the internal indexing system with regard to the removal of a node.
- Step 3: Delete the in-coming and out-going constraints corresponding to the node. The constraints are deleted before the variables are. Before deleting, the existence of the node is double-checked.
- Step 4: Delete the variables corresponding to the node (W's, v's and h's).
- Step 5: Remove the node from the network structure.

One limitation of the code is that it does not completely remove the last neuron in a layer. The removal of the last neuron can be performed in the code to remove a layer, which is described in Section 5.2.5. The removal of the last neuron is effectively the same as removing a entire layer by definition.

The pseudocode for node removal is presented in Algorithm 5.

Algorithm 5 Removal of a Node

procedure NODE_REMOVAL(Layer, Node, Internal_index, Network_Structure)
  Check validity of the node number
  Internal_index[all connections related to the node]=0
  for `<all data points>` **do**
    Remove all out-going constraints from the node.
    Remove all in-coming constraints from the node
    Remove corresponding variables (W's, v's, h's)
  end for
  Remove the node in Network Structure
end procedure

5.2.4 Node Addition

Having defined the process of connection addition, node addition is much simpler. If I would like to add a node $h_{l,i}$, where l is the layer number and i is the node number, I simply add all connections that are linked with the node, both in-coming and out-going connections. I follow the following steps in constructing the network:

- Step 1: Check if the layer and node number is valid. I raise exception in cases where the layer/node number is out of range, the node is existent or the node is added to the input/output layer.
- Step 2: I update the internal indexing system to add a position in the index to represent the node.
- Step 3: Create variables concerning the node addition process (create W's, v's and h's).
- Step 4: Create constraints relating to the in-coming nodes. The internal indexing system is checked for any neuron already removed from the network. If a neuron is absent, the constraints will not contain terms related to that node variable.
- Step 5: Create constraints relating to the out-going nodes. Similarly, the internal indexing system is checked for any absent node not to be contained in the constraint equations.
- Step 6: Update the variable representing the network structure.

One limitation of the function to perform node addition is that it cannot add on to a completely new layer with one neuron, *i.e.* it can only add on to existing layers. If the target is to add a neuron to a new layer, I can use the add layer function instead which will be described in Section 5.2.6.

The pseudocode for node addition is presented in Algorithm 6.

Algorithm 6 Addition of a Node

```
procedure NODE_ADDITION(Layer, Node, Internal_index, Network_Structure)
    Check validity of the node position
    Internal_index[all connections related to the node]=1
    Create corresponding variables (W's, v's, h's)
    for <all data points> do
        Add all out-going constraints from the node.
        Add all in-coming constraints from the node.
    end for
    Add a new node in Network Structure
end procedure
```

5.2.5 Layer Removal

The layer removal is a complicated process. When a layer is removed, I re-establish the connections between layer $l+1$ and layer $l-1$ with full connection between the all the active nodes in the two layers. Suppose I would like to remove layer l. I outline the layer removal process as below:

- Step 1: Check validity of the layer to be removed. Only hidden layers can be removed.
- Step 2: Disconnect in-going and out-coming connections in layer l by deleting the relevant variables (W's, v's, h's) and constraints
- Step 3: Decrease the internal index of all layers beyond layer l ($l-1, l-2, \cdots$) by 1.
- Step 4: Connect the original $l+1$ layer to layer $l-1$ by redefining relevant variables and constraints.

The pseudocode for removing a layer is demonstrated in Algorithm 7

5.2.6 Layer Addition

To add a layer is a very convoluted process. Suppose I would like to add a layer after layer l. There are several key features of the layer addition subroutine. First, since the layer l is added, the original layer l becomes $l+1$ and all layers beyond increase their index by 1. Second, the connections between layer $l-1$ and original l are all broken, which replace the original sparsified connections to fully connected version with the new layer l. Thus, I always

Algorithm 7 Removal of a Layer

procedure LAYER_REMOVAL(Layer, Internal_index, Network_Structure)
Check validity of the layer position
Remove all connections between layer l and layers $l+1$ and $l-1$
Internal_index[all connections beyond layer l]-=1
Re-establish the Internal_index to represent modified connection between layer $l+1$ and $l-1$
Create corresponding variables (W's, v's, h's)
for `<all data points>` **do**
Add all out-going constraints for the layer $l-1$ and all in-coming constraints for layer $l+1$.
end for
Remove a layer in Network Structure
end procedure

obtain the fully connected layers between $l-1$ to l, and l to $l+1$. Third, it is noteworthy that I only add onto the hidden layers, since I cannot add onto the input and output layers.

To summarise the process of layer addition:

- Step 1: Check validity of the layer and the number of neurons to add.
- Step 2: Remove all connections from layer $l-1$ to l.
- Step 3: Increase the index of all layers beyond the layer of interest ($l+1, l+2, \cdots$) by 1.
- Step 4: Add nodes in layer l with the total number of nodes equal the target value.
- Step 5: Connect the in-coming and out-going connections to the added layer.

The algorithm outline is described in Algorithm 8.

5.2.7 Normalisation of Sensitivity Values

After applying the subroutines and obtaining the sensitivity values, I would like to normalise the sensitivity values into a range of -1 to 1 for each layer. In this way, I can create a comparison of the significance of each node and decide which node to remove. The methodology for normalisation is outlined below.

Given λ_i for $i = 1, 2, 3, \cdots, m$. Let:

$$\overline{\lambda} = \max_i \{\lambda_i\} \tag{5.1}$$

Algorithm 8 Addition of a Layer

function LAYER_ADDITION(Layer, Target_Number_of_Node, Internal_index, Network_Structure)
Check validity of the layer position
Remove all connections between layer l and layer $l-1$
Internal_index[all connections beyond layer l]+=1
Re-establish the Internal_index to represent full and modified connection between layer $l-1$ to l and l to $l+1$
Create corresponding variables (W's, v's, h's)
for `<all data points>` **do**
Add all out-going constraints from the layer l.
Add all in-coming constraints to the layer l.
end for
Add a new layer in Network Structure
end function

$$\underline{\lambda} = \min_i \{\lambda_i\} \tag{5.2}$$

I use as a reference value (-1 or +1) the largest of $\overline{\lambda}$ and $\underline{\lambda}$ in magnitude.

$$\Delta\lambda = \max\{|\overline{\lambda}|, |\underline{\lambda}|\} \tag{5.3}$$

Default to range:

$$-\Delta\lambda \leq \lambda \leq +\Delta\lambda \tag{5.4}$$

and normalise all λ_i by:

$$\hat{\lambda}_i = \frac{1}{\Delta\lambda} \cdot \lambda_i \tag{5.5}$$

for $i = 1, 2, 3, \cdots, m$. This creates a normalised range by default, such that:

$$-1 \leq \hat{\lambda}_i \leq +1 \tag{5.6}$$

I use $\hat{\lambda}_i$ for comparison and ranking neurons in terms of importance, where:

$$\lambda_i = \left[\frac{\sum_{j=1}^{ND} \lambda_{i,j}}{ND} \right] \tag{5.7}$$

where ND is the total number of data points. This is effectively finding an average of the sensitivity values on each node across different data points.

5.2.8 Optimisation Options

I have discussed in detail 6 network modification algorithms, including the addition / removal of connections, nodes and layers respectively. Each process entails modification of the network hence an optimisation scheme is required to generate results from the modified network. I have choices of the following optimisation schemes. With regard to node removal:

1. Re-optimise the whole ANN
2. Re-optimise all ANN weights of the remaining nodes
3. Re-optimise only all the weights of the nodes it used to be connected (*i.e.* the node removed)
4. Re-optimise only the in-coming / out-going nodes

With regard to node addition:

1. Re-optimise the whole ANN
2. Keep all other weights $W_{l,i,j}$ fixed and fit only the weights of the new nodes
3. Re-optimise the new nodes and all I/O weights of attached old nodes.

With regard to layer addition:

1. Re-optimise the whole ANN
2. Optimise only the new weights added in this layer
3. Optimise the complete set of weights of the new layer and also the neighbouring layers to the inserted layer

With regard to layer removal:

1. Re-optimise the whole ANN
2. Re-optimise inter-connectivity between layer weights where the missing layer used to be
3. Re-optimise the new inter-connectivity but also the outside layers from the two layers newly linked due to the removal of layer l

With the optimisation schemes above, I could experiment with different schemes and compare the performance of the generated ANN through metrics such as the value of the LSQR objective function.

5.3 Neuron Sensitivity through Automatic Differentiation

In previous sections, I have demonstrated how the modifications of a network can be performed with subroutines to alter the architecture based on sensitivity values. However, it has not been defined when to add/remove neurons and layers. Therefore, I provide the following analysis on the neural architecture with an amplification/attenuation parameter to control the evolution of architecture. The method makes use of an attenuation/amplification factor for each neuron and also collectively for a layer. The sensitivities with regard to the attenuation/amplification factor is used as a guidance to whether a node or a layer should be removed.

5.3.1 Background

The idea of attenuation/amplification factor is inspired from [201]. In this research, the attenuation/amplification factor is adopted to the sensitivity analysis of the predictive modification of biochemical pathways to optimise the selection of reaction steps.

I adopt a similar set of analysis in the ANN network where the sensitivity is calculated from the partial differentials of the objective function with regard to the attenuation/amplification factor I introduced into the ANN to determine the existence of a node or layer. The mathematical details are described in the following section.

5.3.2 Mathematical Formulation

I define the ANN as the following process:

$$z_{l,i,k} = f(y_{l,i,k}) \tag{5.8}$$

where z is the output from a neuron, y is the input to the neuron, $l = 1, \cdots, N_l$ is the layer index, $i = 1, \cdots, N_i$ is the neuron index within layer l, and $k = 1, \cdots, N_k$ is the data point index.

I introduce an attenuation/amplification factor into the formulation of the ANN such that it pre-multiplies the output value of a particular neuron. The factor can effectively serve to represent the neuron's sensitivity in the optimisation process.

$$z_{l,i,k} = \theta_{l,i} \cdot f(y_{l,i,k}) \tag{5.9}$$

where

$$\theta_{l,i} = \begin{cases} 1, & neuron\ exists \\ 0, & neuron\ does\ not\ exists \end{cases}$$

The artificial parameters $\theta_{l,i}$ are attenuation/amplification parameters.

I find sensitivities of the optimal objective function value with respect to the parameters: $\frac{\partial(LSQR^\star)}{\partial\theta_{l,i}^{(neuron)}}$ and $\frac{\partial(LSQR^\star)}{\partial\theta_{l}^{(layer)}}$.

Consider an optimisation problem involving a set of parameters $\theta \in \mathbb{R}^{N_\theta}$. $N_\theta \geq 1$ is the total number of parameters. I define the optimisation problem:

$$\begin{aligned} \phi &= min_{x\in\mathscr{X}} f(x;\theta) \\ &s.t.\ h(x;\theta) = 0 \end{aligned}$$

other constraints, equalities and/or inequalities, that nonetheless do not depend on θ, will be ignored.

Consider the Lagrangian function:

$$\mathscr{L}(x,\lambda;\theta) = f(x;\theta) + \lambda^T h(x;\theta) \tag{5.10}$$

I know that for $(x^\star,\lambda^\star)$ I get:

$$\mathscr{L}(x^\star,\lambda^\star;\theta) = f(x^\star;\theta) + 0 \tag{5.11}$$

The total derivative/gradient of the objective function with respect to θ at the optimal point $(x^\star(\theta),\lambda^\star(\theta))$ for a given value for the vector of parameters θ:

$$\begin{aligned} \left.\frac{Df}{D\theta}\right|_{[x^\star(\theta),\lambda^\star(\theta),\theta]} &= \left.\frac{D\mathscr{L}}{D\theta}\right|_{[x^\star(\theta),\lambda^\star(\theta),\theta]} \\ &= \frac{\partial f}{\partial x}\frac{\partial x}{\partial\theta} + \lambda^T\left(\frac{\partial h}{\partial x}\frac{\partial x}{\partial\theta} + \frac{\partial h}{\partial\theta}\right) \\ &= \frac{\partial f}{\partial x}\frac{\partial x}{\partial\theta} \end{aligned}$$

Consider next the specific form of our ANN neuron fitting constraints:

$$h_{l,i,k} = z_{l,i,k} - \theta_{l,i}\cdot f(y) \tag{5.12}$$

where $z_{l,i,k}$ is the set of neurons outputs for each data point in our dataset, y is the set of variables in other equality constraints of the general form $g(y,z) = 0$. The variables z are the

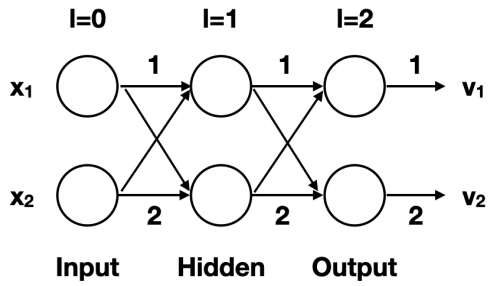


Figure 5.1 The architecture of ANN used for example formulation.

ones I am interested in modifying with the artificially introduced attenuation/amplification parameters θ. The other equations more specifically relate to the impact of the node (l,i) at data point (k):

$$y_{l,i,k} = \sum_{j=1}^{N_{i-1}} z_{l,j,k} \cdot W_{l,j,i}$$

where $l = 2,\cdots,N_l$, $i,j = 1,\cdots,N_i$, and $k = 1,\cdots,N_k$.

Example Formulation

The generic form of these relations is not very easy to construct algebraically. I first proceed by an example to get the required partial derivatives in the Jacobian matrices involved. Figure 5.1 represents an example network architecture.

There are 12 weights and bias terms for the hidden layer $(l = 1)$ and output layer $(l = 2)$. The input layer is $(l = 0)$. The equations involved include:

$$h_1 = z_{1,1,k} - \theta_{1,1} \cdot f(y_{1,1,k}) = 0 \tag{5.13}$$

$$h_2 = y_{1,1,k} - (W_{1,1,0} + W_{1,1,1} \cdot x_{1,k} + W_{1,2,1} \cdot x_{2,k}) = 0 \tag{5.14}$$

$$h_3 = z_{1,2,k} - \theta_{1,2} \cdot f(y_{1,2,k}) = 0 \tag{5.15}$$

$$h_4 = y_{1,2,k} - (W_{1,2,0} + W_{1,2,1} \cdot x_{1,k} + W_{1,2,2} \cdot x_{2,k}) = 0 \tag{5.16}$$

$$h_5 = v_{1,k} - (W_{2,1,0} + W_{2,1,1} \cdot z_{2,1,k} + W_{2,1,2} \cdot z_{2,2,k}) = 0 \tag{5.17}$$

$$h_6 = v_{2,k} - (W_{2,2,0} + W_{2,2,1} \cdot z_{2,1,k} + W_{2,2,2} \cdot z_{2,2,k}) = 0 \tag{5.18}$$

The θs are added only before the outputs of a neuron to control the existence/absence of a neuron. There are 6 equations in total with state variables $z_{1,1,k}$, $z_{1,2,k}$, $y_{1,1,k}$, $y_{1,2,k}$. $v_{1,k}$ and $v_{2,k}$. Thus, the 6 equations determine the 6 state variables.

I want the Jacobian of these 6 equations with respect to the 6 state variables ($\frac{\partial h}{\partial a}$), where a can refer to $z_{l,j,i}$ or $y_{l,j,k}$.

$$\frac{\partial h_1}{\partial z_{1,1,k}} = +1 \tag{5.19}$$

$$\frac{\partial h_1}{\partial y_{1,1,k}} = -\frac{\partial f}{\partial y_{1,1,k}} \cdot \theta_{1,1} \tag{5.20}$$

$$\frac{\partial h_2}{\partial y_{1,1,k}} = +1 \tag{5.21}$$

$$\frac{\partial h_3}{\partial z_{1,2,k}} = +1 \tag{5.22}$$

$$\frac{\partial h_3}{\partial y_{1,2,k}} = -\frac{\partial f}{\partial y_{1,2,k}} \cdot \theta_{1,2} \tag{5.23}$$

$$\frac{\partial h_4}{\partial y_{1,2,k}} = +1 \tag{5.24}$$

All other Jacobian entries are zero. Because I start with an ANN where obviously all nodes/neurons are fully in existence, then clearly that is why I must have $\theta_{2,1} = \theta_{2,2} = +1$.

I also need the Jacobian with respect to the artificially introduced factors $\theta_{l,i}$, where $l = \{1\}, i = \{1,2\}$. I obtain the following equations:

$$\frac{\partial h_1}{\partial \theta_{1,1}} = -f(y_{1,1,k}) \tag{5.25}$$

$$\frac{\partial h_3}{\partial \theta_{1,2}} = -f(y_{1,2,k}) \tag{5.26}$$

All other entries of this Jacobian are zero.

Objective function is:

$$LSQR = \frac{1}{2} \sum_{k=1}^{N_k} \left[(v_{1,k} - v_{target,1,k})^2 + (v_{2,k} - v_{target,2,k})^2 \right] \tag{5.27}$$

where $v_{target,1,k}$ and $v_{target,2,k}$ are constant output level data.

Gradient with respect to v:

$$\frac{\partial (LSQR)}{\partial v_{1,k}} = (v_{1,k} - v_{target,1,k}) \tag{5.28}$$

$$\frac{\partial (LSQR)}{\partial v_{2,k}} = (v_{2,k} - v_{target,2,k}) \tag{5.29}$$

Table 5.1 The Jacobian values of equality constraints with regard to node variables.

$\frac{\partial h}{\partial a}$	$z_{1,1,k}$	$z_{1,2,k}$	$y_{1,1,k}$	$y_{1,2,k}$
h_1	+1	0	$-\frac{\partial f}{\partial y_{1,1,k}} \cdot \theta_{1,1}$	0
h_2	0	0	+1	0
h_3	0	+1	0	$-\frac{\partial f}{\partial y_{1,2,k}} \cdot \theta_{1,2}$
h_4	0	0	0	+1
h_5	$-W_{2,1,1}$	$-W_{2,1,2}$	0	0
h_6	$-W_{2,2,1}$	$-W_{2,2,2}$	0	0

Table 5.2 The Jacobian values of equality constraints with regard to attenuation/amplification variables.

$\frac{\partial h}{\partial \theta}$	$\theta_{1,1,k}$	$\theta_{1,2,k}$
$h_{1,k}$	$-f(y_{1,1,k})$	0
$h_{2,k}$	0	0
$h_{3,k}$	0	$-f(y_{1,2,k})$
$h_{4,k}$	0	0
$h_{5,k}$	0	0
$h_{6,k}$	0	0

The objective does not contain by construction the θ, nor any other part of the model, except form the neuron firing/output equality constraints. The various Jacobians and total sensitivity of the objective formula are given in Table 5.1 and 5.2.

Thus, the total sensitivity gradient is given by:

$$\frac{D(LSQR)}{D\theta} = \frac{\partial(LSQR)}{\partial a}\frac{\partial a}{\partial \theta} \tag{5.30}$$

I already have values of $\frac{\partial(LSQR)}{\partial a}$. For $\frac{\partial a}{\partial \theta}$, I have the sensitivity equation of ANN with respect to θ:

$$\frac{\partial h}{\partial a}\frac{\partial a}{\partial \theta} + \frac{\partial h}{\partial \theta} = 0 \tag{5.31}$$

since $h(a,\theta) = 0$.

Since given: (a) all the weights $W_{l,j,i}$, (b) $\theta_{l,i} \equiv 1$, and (c) all $(v_{target,1,k}, v_{target,2,k})^T$, I can determine uniquely $y_{l,i,j}$ and $z_{l,i,j}$. Therefore, $\frac{\partial h}{\partial a}$ is invertible (factorizable and very sparse). I can solve $\frac{\partial h}{\partial a}$ to get $\frac{\partial a}{\partial \theta}$. Then I can calculate $\frac{D(LSQR)}{D\theta}$, which I can use as a "gradient" to guide addition or removal of nodes and/or entire layers.

There are many sensitivities and they are by construction dense matrices (the "$\frac{\partial a}{\partial \theta}$"). Since I care about the total sensitivity over all dataset points, I can find more efficient ways to compute them, for example, to calculate them independently for all data points and then

sum them up.

$$\left.\frac{D(LSQR)}{D\theta}\right|_{[k]} = \left.\frac{\partial(LSQR)}{\partial a}\right|_{[k]} \cdot \left.\frac{\partial a}{\partial \theta}\right|_{[k]} \tag{5.32}$$

$$\left.\frac{\partial a}{\partial \theta}\right|_{[k]} = -\left(\left.\frac{\partial h}{\partial a}\right|_{[k]}\right)^{(-1)} \cdot \frac{\partial h}{\partial \theta} \tag{5.33}$$

Thus, the total sensitivity of LSQR with respect to θ should be:

$$\left.\frac{D(LSQR)}{D\theta}\right|_{total} = \sum_{k=1}^{N_k} \left.\frac{D(LSQR)}{D\theta}\right|_{[k]} \tag{5.34}$$

where $k = 1, \cdots, N_k$. The back-substitution and matrix factorisation can be done separately and in parallel for all data points. Thus, the calculation of the LSQR total sensitivity/gradient with respect to θ is a fully scalable and parallelisable task. This sensitivity analysis demonstrates the potential of the formulation of ANN with the lifting scheme to be used for large-scale development.

Generalisation of the Example Formulation

I perform layer by layer calculation of output sensitivity of an ANN to its neurons' attenuation/amplification artificial parameters. For $k = 1, \cdots, N_k$ data points, the output of layer l given input form layer $l-1$ is:

$$z_{l,i,k} = \theta_{l,i} \cdot f(y_{l,i,k}) \tag{5.35}$$

where:

$$y_{l,i,k} = \sum_{j=1}^{NN_{(l-1)}} W_{l,i,j} \cdot z_{(l-1),j,k} + W_{l,i,0} \tag{5.36}$$

for $l = 1, 2, \cdots, N_l$, $i = 1, 2, \cdots, N_i$, and $j = 1, 2, \cdots, N_j$. N_l is the total number of layers. N_i is the number of neurons in layer l and N_j is the number of neurons in layer $l-1$. $W_{l,i,0}$ is the bias term.

I wish to compute the following sensitivity:

$$\left[\frac{\partial z_{i,k}}{\partial \theta_{l,i}}\right] \tag{5.37}$$

For layer (l), assume that from the previous layer computation, I have available:

$$\frac{\partial z_{(l-1),i,k}}{\partial \theta_{l',i',k}} \tag{5.38}$$

for $l' \leq (l-1)$, *i.e.* $l' = 1,2,\cdots,(l-1)$, $j = 1,2,\cdots,N_j$, $j' = 1,2,\cdots,N'_j$, and $k = 1,2,\cdots,N_k$.

Therefore, for layer (l) from Equations 5.35 and Equation 5.36:

$$\frac{\partial z_{l,i,k}}{\partial \theta_{l',i'}} = \frac{\partial f}{\partial y_{l,i,k}} \cdot \frac{\partial y_{l,i,k}}{\partial \theta_{l',i'}} \cdot \theta_{l,i} + f(y_{l,i,k}) \cdot C \tag{5.39}$$

where

$$C = \begin{cases} 1, & iff \text{ l'=l, i'=i} \\ 0, & otherwise \end{cases}$$

and

$$\frac{\partial y_{l,i,k}}{\partial \theta_{l',i'}} = \sum_{j=1}^{NN_{(l=1)}} W_{l,i,j} \cdot \frac{\partial z_{(l-1),i,k}}{\partial \theta_{l',i'}} \tag{5.40}$$

where $NN_{(l=1)}$ is the number of neurons in the input layer. This is clearly a nice and transparent iterative scheme. I can encode it into an algorithm. Initialise a $\left[\frac{\partial z_{l,i,k}}{\partial \theta_{l',i'}}\right]$ matrix storage, initialised to zeros. To see how the initial iteration works, I start the computation for layer $l = 1$.

Set $l = 1$:

$$\frac{\partial z_{1,i,k}}{\partial \theta_{l',i'}} = \frac{\partial f}{\partial y_{1,i,k}} \cdot \frac{\partial y_{1,i,k}}{\partial \theta_{l',i'}} \cdot \theta_{1,i} + f(y_{1,i,k}) \cdot C \tag{5.41}$$

where $\theta_{1,i} = 1$ and $\frac{\partial y_{1,i,k}}{\partial \theta_{l',i'}} = 0$ since there is no prior layer dependent on neuron's outputs.

Therefore, for layer $l = 1$ I get:

$$\frac{\partial z_{1,i,k}}{\partial \theta_{l',i'}} = 0 \tag{5.42}$$

for all $l' = 1,2,\cdots,(N_l)$ and $i' = 1,2,\cdots,N'_i$, where $l' \neq l$ and $i' \neq i$.

For $l' = l$ and $i' = i$,

$$\frac{\partial z_{1,i,k}}{\partial \theta_{1,i}} = f(y_{1,i,k}) \tag{5.43}$$

Then, for $l > 1$, Equations 5.39 and 5.40 can be used routinely as derived.

For example, for the second layer which is assumed to be a non-terminating layer, I set $l = 2$:

$$\frac{\partial z_{2,i,k}}{\partial \theta_{l',i'}} = \frac{\partial f}{\partial y_{2,i,k}} \cdot \frac{\partial y_{2,i,k}}{\partial \theta_{l',i'}} \cdot \theta_{2,i} + f(y_{2,i,k}) \cdot C \tag{5.44}$$

where $\frac{\partial f}{\partial y_{2,i,k}}$ is calculated from the derivatives of the activation function, $\frac{\partial y_{2,i,k}}{\partial \theta_{l',i'}}$ is defined in Equation 5.40, $\theta_{2,i} = 1$ and $f(y_{2,i,k})$ is the output of the neuron.

If the activation function used is the Sigmoid function, which has the form:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5.45}$$

The derivative of the Sigmoid function is:

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{5.46}$$

Since f is the transformation from the input to a neuron to the output of a neuron, it is effectively the activation function. $f(y_{l,i,k})$ is the output of a neuron. Thus, I have $\frac{\partial f}{\partial y_{2,i,k}} = f(y_{2,i,k})(1-f(y_{2,i,k}))$.

Therefore,

$$\frac{\partial z_{2,i,k}}{\partial \theta_{l',i'}} = f(y_{2,i,k})(1-f(y_{2,i,k})) \cdot \sum_{j=1}^{NN_{(l=1)}} W_{2,i,j} \cdot \frac{\partial z_{1,i,k}}{\partial \theta_{l',i'}} \cdot \theta_{2,i} + f(y_{2,i,k}) \cdot C \tag{5.47}$$

where $\frac{\partial z_{1,i,k}}{\partial \theta_{l',i'}}$ is calculated in Equation 5.42 for $l' \neq l, i' \neq i$ and in Equation 5.39 for $l' = l, i' = i$. These are in the previous loop for $l = 1$. Another special case I need to look at is the last layer (output layer). In this layer, I do not have the activation function.

For $l' = l, i' = i$, I have the following relationship:

$$\frac{\partial v_{l,k}}{\partial \theta_{l,i}} = 0 \tag{5.48}$$

since the output neurons are always present and are not controlled by the attenuation/amplification factor θ.

For $l' \neq l, i' \neq i$, I have the following:

$$\frac{\partial v_{i,k}}{\partial \theta_{l',i'}} = \frac{\partial v_{i,k}}{\partial z_{(l-1),i}} \cdot \frac{\partial z_{(l-1),i}}{\partial \theta_{l',i'}} \quad = W_{l,j,i} \cdot \frac{\partial z_{(l-1),i}}{\partial \theta_{l',i'}}$$

where $\frac{\partial z_{(l-1),i}}{\partial \theta_{l',i'}}$ is calculated from the previous iteration. Thus, I arrive at having calculated the following for the output layer of the ANN: $\frac{\partial v_{i,k}}{\partial \theta_{l,i}}$, for all $l = 1, 2, \cdots, N_l$. This is the first-order sensitivities of ANN output with respect to neuron existence.

Looking at our objective function:

$$LSQR = \sum_{k=1}^{N_k} \sum_{i=1}^{NN_{l=N_l}} (v_{i,k} - v_{target,i,k})^2 \tag{5.49}$$

where $NN_{l=N_l}$ is the number of neurons in the last layer.

To get a single neuron gradient contribution or sensitivity, I obtain:

$$\frac{\partial(LSQR)}{\partial\theta_{l',i'}} = \frac{\partial}{\partial\theta_{l',i'}}\left[\sum_{k=1}^{N_k}\sum_{i=1}^{NN_{l=N_l}}(v_{i,k}-v_{target,i,k})^2\right] \tag{5.50}$$

This yields:

$$s_{l',i'} = \frac{\partial(LSQR)}{\partial\theta_{l',i'}} = \sum_{k=1}^{N_k}\sum_{i=1}^{NN_{l=N_l}}\frac{\partial z_{l,i,k}}{\partial\theta_{l',i'}} \tag{5.51}$$

for $l' = 1,2,\cdots,N_l$ and $i' = 1,2,\cdots,N_i'$. I term $s_{l',i'}$ the LSQR sensitivity with respect to $\theta_{l',i'}$.

For the entire gradient contribution of an entire layer as a collection of neurons, I get:

$$s_{l'} = \sum_{i'=1}^{N_i'} s_{l',i'} \tag{5.52}$$

This concludes the special sensitivity analysis for efficient computation of ANN sensitivities with respect to the existence of single neuron and entire neuron layers.

Example Iteration

Suppose I have a neural network with architecture of [4,5,3,2,2] neurons in each layer, where the first number represents the neuron number in the input layer and the last number represents the neuron number in the output layer. I arrive at Table 5.3 and Table 5.4.

Table 5.3 The sensitivity of equality constraints with regard to attenuation/amplification variables (Part I).

$\frac{\partial z}{\partial \theta}$	$z_{1,1,k}$	$z_{1,2,k}$	$z_{1,3,k}$	$z_{1,4,k}$	$z_{1,5,k}$	$z_{2,1,k}$	$z_{2,2,k}$	$z_{2,3,k}$
$\theta_{1,1}$	$f(y_{1,1,k})$	0	0	0	0	$z_{2,1,k}\_\theta_{1,1} = f(y_{2,1,k})(1-f(y_{2,1,k})) \cdot (W_{2,1,1} f(y_{1,1,k}) \cdot \theta_{2,1}$	$z_{2,2,k}\_\theta_{1,1} = f(y_{2,2,k})(1-f(y_{2,2,k})) \cdot (W_{2,1,1} f(y_{1,1,k}) \cdot \theta_{2,2}$	$z_{2,3,k}\_\theta_{1,1} = f(y_{2,3,k})(1-f(y_{2,3,k})) \cdot (W_{2,1,1} f(y_{1,1,k}) \cdot \theta_{2,3}$
$\theta_{1,2}$	0	$f(y_{1,2,k})$	0	0	0	$z_{2,1,k}\_\theta_{1,2} = f(y_{2,1,k})(1-f(y_{2,1,k})) \cdot (W_{2,1,2} f(y_{1,2,k}) \cdot \theta_{2,1}$	$z_{2,2,k}\_\theta_{1,2} = f(y_{2,2,k})(1-f(y_{2,2,k})) \cdot (W_{2,2,2} f(y_{1,2,k}) \cdot \theta_{2,2}$	$z_{2,3,k}\_\theta_{1,2} = f(y_{2,3,k})(1-f(y_{2,3,k})) \cdot (W_{2,3,2} f(y_{1,2,k}) \cdot \theta_{2,3}$
$\theta_{1,3}$	0	0	$f(y_{1,3,k})$	0	0	$z_{2,1,k}\_\theta_{1,3} = f(y_{2,1,k})(1-f(y_{2,1,k})) \cdot (W_{2,1,3} f(y_{1,3,k}) \cdot \theta_{2,1}$	$z_{2,2,k}\_\theta_{1,3} = f(y_{2,2,k})(1-f(y_{2,2,k})) \cdot (W_{2,2,3} f(y_{1,3,k}) \cdot \theta_{2,2}$	$z_{2,3,k}\_\theta_{1,3} = f(y_{2,3,k})(1-f(y_{2,3,k})) \cdot (W_{2,3,3} f(y_{1,3,k}) \cdot \theta_{2,3}$
$\theta_{1,4}$	0	0	0	$f(y_{1,4,k})$	0	$z_{2,1,k}\_\theta_{1,4} = f(y_{2,1,k})(1-f(y_{2,1,k})) \cdot (W_{2,1,4} f(y_{1,4,k}) \cdot \theta_{2,1}$	$z_{2,2,k}\_\theta_{1,4} = f(y_{2,2,k})(1-f(y_{2,2,k})) \cdot (W_{2,2,4} f(y_{1,4,k}) \cdot \theta_{2,2}$	$z_{2,3,k}\_\theta_{1,4} = f(y_{2,3,k})(1-f(y_{2,3,k})) \cdot (W_{2,3,4} f(y_{1,4,k}) \cdot \theta_{2,3}$
$\theta_{1,5}$	0	0	0	0	$f(y_{1,5,k})$	$z_{2,1,k}\_\theta_{1,5} = f(y_{2,1,k})(1-f(y_{2,1,k})) \cdot (W_{2,1,5} f(y_{1,5,k}) \cdot \theta_{2,1}$	$z_{2,2,k}\_\theta_{1,5} = f(y_{2,2,k})(1-f(y_{2,2,k})) \cdot (W_{2,2,5} f(y_{1,5,k}) \cdot \theta_{2,2}$	$z_{2,3,k}\_\theta_{1,5} = f(y_{2,3,k})(1-f(y_{2,3,k})) \cdot (W_{2,3,5} f(y_{1,5,k}) \cdot \theta_{2,3}$
$\theta_{2,1}$	0	0	0	0	0	$f_{2,1,0}$	0	0
$\theta_{2,2}$	0	0	0	0	0	0	$f_{2,2,0}$	0
$\theta_{2,3}$	0	0	0	0	0	0	0	$f_{2,3,0}$
$\theta_{3,1}$	0	0	0	0	0	0	0	0
$\theta_{3,2}$	0	0	0	0	0	0	0	0
$\theta_{4,1}$	0	0	0	0	0	0	0	0
$\theta_{4,2}$	0	0	0	0	0	0	0	0

Table 5.4 The sensitivity of equality constraints with regard to attenuation/amplification variables (Part II).

$\frac{\partial z}{\partial \theta}$	$z_{3,1,k}$	$z_{3,2,k}$	$v_{1,k}$	$v_{2,k}$
$\theta_{1,1}$	$z_{3,1,k}\_\theta_{1,1} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,1}z_{2,1,0}\_\theta_{1,1}+$ $+W_{3,1,2}z_{2,2,0}\_\theta_{1,1}+W_{3,1,2}z_{2,2,0}\_\theta_{1,1})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{1,1} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,2,1}z_{2,1,0}\_\theta_{1,1}$ $W_{3,2,2}z_{2,2,0}\_\theta_{1,1}+W_{3,2,3}z_{2,3,0}\_\theta_{1,1})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{1,1})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{1,1})$
$\theta_{1,2}$	$z_{3,1,k}\_\theta_{1,2} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,1}z_{2,1,0}\_\theta_{1,2}+$ $+W_{3,1,2}z_{2,2,0}\_\theta_{1,2}+W_{3,1,3}z_{2,3,0}\_\theta_{1,2})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{1,2} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,2,1}z_{2,1,0}\_\theta_{1,2}$ $W_{3,2,2}z_{2,2,0}\_\theta_{1,2}+W_{3,2,3}z_{2,3,0}\_\theta_{1,2})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{1,2})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{1,2})$
$\theta_{1,3}$	$z_{3,1,k}\_\theta_{1,3} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,1}z_{2,1,0}\_\theta_{1,3}+$ $+W_{3,1,2}z_{2,2,0}\_\theta_{1,3}+W_{3,1,3}z_{2,3,0}\_\theta_{1,3})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{1,3} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,2,1}z_{2,1,0}\_\theta_{1,3}$ $W_{3,2,2}z_{2,2,0}\_\theta_{1,3}+W_{3,2,3}z_{2,3,0}\_\theta_{1,3})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{1,3})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{1,3})$
$\theta_{1,4}$	$z_{3,1,k}\_\theta_{1,4} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,1}z_{2,1,0}\_\theta_{1,4}+$ $+W_{3,1,2}z_{2,2,0}\_\theta_{1,4}+W_{3,1,3}z_{2,3,0}\_\theta_{1,4})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{1,4} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,2,1}z_{2,1,0}\_\theta_{1,4}$ $W_{3,2,2}z_{2,2,0}\_\theta_{1,4}+W_{3,2,3}z_{2,3,0}\_\theta_{1,4})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{1,4})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{1,4})$
$\theta_{1,5}$	$z_{3,1,k}\_\theta_{1,5} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,1}z_{2,1,0}\_\theta_{1,5}+$ $+W_{3,1,2}z_{2,2,k}\_\theta_{1,5}+W_{3,1,3}z_{2,3,k}\_\theta_{1,5})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{1,5} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,2,1}z_{2,1,k}\_\theta_{1,5}$ $W_{3,2,2}z_{2,2,k}\_\theta_{1,5}+W_{3,2,3}z_{2,3,k}\_\theta_{1,5})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{1,5})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{1,5})$
$\theta_{2,1}$	$z_{3,1,k}\_\theta_{2,1} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,2}f_{2,2,k})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{2,1} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,1,2}f_{2,2,k})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{2,1})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{2,1})$
$\theta_{2,2}$	$z_{3,1,k}\_\theta_{2,2} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,2}f_{2,2,k})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{2,2} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,1,2}f_{2,2,k})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{2,2})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{2,2})$
$\theta_{2,3}$	$z_{3,1,k}\_\theta_{2,3} =$ $f(y_{3,1,k})(1-f(y_{3,1,k}))\cdot(W_{3,1,3}f_{2,3,k})\cdot\theta_{3,1}$	$z_{3,2,k}\_\theta_{2,3} =$ $f(y_{3,2,k})(1-f(y_{3,2,k}))\cdot(W_{3,1,3}f_{2,3,k})\cdot\theta_{3,1}$	$W_{4,1,1}(z_{3,1,k}\_\theta_{2,3})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{2,3})$
$\theta_{3,1}$	$f(y_{3,1,0})$	0	$W_{4,1,1}(z_{3,1,k}\_\theta_{3,1})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{3,1})$
$\theta_{3,2}$	0	$f(y_{3,2,0})$	$W_{4,1,1}(z_{3,1,k}\_\theta_{3,2})$	$W_{4,1,2}(z_{3,2,k}\_\theta_{3,2})$
$\theta_{4,1}$	0	0	0	0
$\theta_{4,2}$	0	0	0	0

There is a more sophisticated manner in which I can implement this iterative scheme. If I carefully observe the tables, I can see that the iterative process can be simplified into the following steps:

- Initialisation: Generate a matrix to store differential values as represented in Table 5.3 and Table 5.4 (number of rows equal to the total number of neurons in the network and number of columns equal to the total number of neurons times the number of data points). I divide the matrix into sub-matrices of differential values for each layer.
- Step 1: Define the initialisation of the differential values for the first layer ($\frac{\partial z_{1,i,k}}{\partial \theta_{l',i'}}$). This is effectively setting the diagonal of the sub-matrix representing the first layer to be the output of the neuron ($f(y_{1,i,k})$) while keeping other positions to be 0's.
- Step 2: To arrive at the values for the second layer ($\frac{\partial z_{2,i,k}}{\partial \theta_{l',i'}}$), I perform a scaled matrix multiplication between the weights related to the second layer (matrix of $W_{2,i,j}$'s) and the initialised matrix. The scaled value is defined to be $f(y_{2,i,k})(1-f(y_{2,i,k}))\cdot\theta_{2,i}$.
- Step 3: In the previous step, I have updated the values for $\theta_{l',i'} \neq \theta_{l,i}$. To update for $\theta_{l',i'} = \theta_{l,i}$, I only need to change the diagonal (in part of the matrix relating to $\theta_{l,i}$) to $f(y_{2,i,k})$ while keeping the non-diagonal values to 0's. This finishes the update of the second layer.
- Step 4: For non-terminating layers (hidden layers), I repeat Step 2 and Step 3 except that I replace the initialised sub-matrix of first layer to be the sub-matrix of the layer previous to the layer under current iteration. For example, for layer 3, I take the sub-matrix of layer 2 ($\frac{\partial z_{2,i,k}}{\partial \theta_{l',i'}}$) to be the component for matrix multiplication.
- Step 5: For the terminating layer (output layer), this is effectively a matrix multiplication between the weights matrix for that layer and the sub-matrix of differential values for the previous layer.

The sensitivity of each neuron with regard to the objective function is thus obtained through the sub-matrix of the last layer. The final sensitivity for each neuron is the sum of sensitivity values across the output dimension and across different data points.

This scheme allows the calculation of sensitivities of all neurons to be achieved through a simple iterative matrix multiplication process. This can be easily implemented systematically in code or even in an Excel spreadsheet.

Adopting the dataset described in Section 4.8.1, with the neural architecture of [4,5,3,2,2] as demonstrated in Table 5.3 and Table 5.4, I arrive at the calculated sensitivities for 1,000 data points as tabulated in Table 5.5 and Table 5.6.

Table 5.5 The calculated sensitivities of each neuron in a neural network with structure [4,5,3,2,2] using the simulated dataset - Part I.

	$z_{1,1}$	$z_{1,2}$	$z_{1,3}$	$z_{1,4}$	$z_{1,5}$	$z_{2,1}$	$z_{2,2}$
$\theta_{1,1}$	7.82257×10^{-1}	0	0	0	0	1.09007×10^{-3}	3.69873×10^{-4}
$\theta_{1,2}$	0	7.82257×10^{-1}	0	0	0	1.90063×10^{-3}	3.69870×10^{-4}
$\theta_{1,3}$	0	0	7.82257×10^{-1}	0	0	1.09007×10^{-3}	3.69872×10^{-4}
$\theta_{1,4}$	0	0	0	7.82257×10^{-1}	0	1.09008×10^{-3}	3.69874×10^{-4}
$\theta_{1,5}$	0	0	0	0	7.82257×10^{-1}	1.09007×10^{-3}	3.69872×10^{-4}
$\theta_{2,1}$	0	0	0	0	0	8.17351×10^{-1}	0
$\theta_{2,2}$	0	0	0	0	0	0	8.17351×10^{-1}
$\theta_{2,3}$	0	0	0	0	0	0	0
$\theta_{3,1}$	0	0	0	0	0	0	0
$\theta_{3,2}$	0	0	0	0	0	0	0
v_1	0	0	0	0	0	0	0
v_2	0	0	0	0	0	0	0

Table 5.6 The calculated sensitivities of each neuron in a neural network with structure [4,5,3,2,2] using the simulated dataset - Part II.

	$z_{2,3}$	$z_{3,1}$	$z_{3,2}$	v_1	v_2	Overall Neuron Sensitivity
$\theta_{1,1}$	3.69873×10^{-4}	2.05372×10^{-6}	7.30697×10^{-7}	3.95613×10^{-6}	3.95613×10^{-6}	2.74701×10^{-6}
$\theta_{1,2}$	3.69870×10^{-6}	2.05370×10^{-6}	7.30691×10^{-7}	3.95610×10^{-6}	3.95610×10^{-6}	2.74699×10^{-6}
$\theta_{1,3}$	3.69873×10^{-4}	2.05372×10^{-6}	7.30698×10^{-7}	3.95614×10^{-6}	7.82257×10^{-1}	2.74702×10^{-6}
$\theta_{1,4}$	3.69874×10^{-4}	2.05372×10^{-6}	7.30699×10^{-7}	3.95614×10^{-6}	3.95614×10^{-6}	2.74702×10^{-6}
$\theta_{1,5}$	3.69872×10^{-4}	2.05371×10^{-6}	7.30694×10^{-7}	3.95612×10^{-6}	3.5612×10^{-6}	2.74700×10^{-6}
$\theta_{2,1}$	0	1.64436×10^{-3}	5.37891×10^{-4}	2.91646×10^{-3}	2.91646×10^{-3}	5.83293×10^{-3}
$\theta_{2,2}$	0	1.64436×10^{-3}	5.37891×10^{-4}	2.91646×10^{-3}	2.91646×10^{-3}	5.83293×10^{-3}
$\theta_{2,3}$	8.17351×10^{-1}	1.64436×10^{-3}	5.37890×10^{-4}	2.91646×10^{-3}	2.91646×10^{-3}	5.83292×10^{-3}
$\theta_{3,1}$	0	6.46865×10^{-1}	0	2.06033×10^{-2}	2.06033×10^{-2}	-4.12065×10^{-2}
$\theta_{3,2}$	0	0	6.46865×10^{-1}	3.44434×10^{0}	3.44434×10^{0}	6.88868×10^{0}
v_1	0	0	0	0	0	0
v_2	0	0	0	0	0	0

Table 5.7 The CPU time for the sensitivity calculation of a neural network with architecture [4,5,3,2,2] using 1,000 data points.

Steps	Optimisation	Iterations 1	Iterations 2	Iterations 3	Iterations 4	Total
CPU Time(s)	1.233	0.000709	0.0192	0.00787	0.000728	1.262

From Table 5.5 and Table 5.6, I identify the neuron with the lowest sensitivity to be the neurons in the first layer with equal sensitivity. Therefore, in terms of neuron removal, I could remove any neuron in the first layer. In terms of layer removal, I add up the sensitivities of neurons in each layer and the results demonstrate that I should remove the first layer. If I would like to add a neuron, I should consider neurons in the last hidden layer which has the highest sensitivity. If I would like to add a layer, the last hidden layer is the most sensitive and hence shall be duplicated.

5.3.3 CPU Time and Memory

The complete process of automatic differentiation consists of two steps: 1) NLP optimisation to obtain the weights and the neuron outputs, and 2) iterations to calculate the sensitivities. For a structure of [4,5,3,2,2], the CPU times required for calculation of sensitivities of 1,000 data points are tabulated in 5.7.

The processor used to run this code is the 2.3GHz Quad-Core Intel Core i5 with a memory setting of 8GB 2133MHz LPDDR3. It can be observed that most of the time is spent on the optimisation process (97.7% of CPU time). This is the uncontrollable process as I am using a standard optimiser. The calculation of the sensitivities are fast, taking 0.0290s in total (2.30% of CPU time). Thus, this is quite an effective method as long as the optimiser reaches a good optimum in a small amount of time.

The disadvantage is that the algorithm requires the storage of a matrix of size $(NN, NN \times N_k)$ where NN is the total number of neurons and N_k is the total number of data points.

5.4 Neuron Sensitivity through Finite Differences

In the previous section, I have performed the calculation of sensitivities of all neurons in the network through automatic differentiation, with the aim to identify the least (most) sensitive neuron or layer in order to remove (add). The method of automatic differentiation is able to produce the most rigorously calculated sensitivity values for all neurons. An alternative method to gauge the sensitivity of neurons is through finite difference method. This method is less rigorous and cannot produce a sensitivity value. However, it is easy in terms of

implementation into more sophisticated forms. Moreover, it can serve as a validation of the results obtained from automatic differentiation.

5.4.1 Mathematical Formulation

I adopt central difference in this case to generate more accurate results. Thus, the finite difference is defined to be:

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\frac{1}{2}\varepsilon) - f(x-\frac{1}{2}\varepsilon)}{\varepsilon} \tag{5.53}$$

I start from an example formulation and then extend to generalised cases. Suppose I have a network with the architecture [2,2,2]. The formulation adopting the lifting scheme has been described in Section 4.4, which is listed here fore reference. In order to calculate the sensitivity of each neuron, I first solve the optimisation problem with the following constraints.

$$z_{1,1,k} - f(y_{1,1,k}) = 0 \tag{5.54}$$

$$y_{1,1,k} - (W_{1,1,0} + W_{1,1,1} \cdot x_{1,k} + W_{1,2,1} \cdot x_{2,k}) = 0 \tag{5.55}$$

$$z_{1,2,k} - f(y_{1,2,k}) = 0 \tag{5.56}$$

$$y_{1,2,k} - (W_{1,2,0} + W_{1,2,1} \cdot x_{1,k} + W_{1,2,2} \cdot x_{2,k}) = 0 \tag{5.57}$$

$$v_{1,k} - (W_{2,1,0} + W_{2,1,1} \cdot z_{2,1,k} + W_{2,1,2} \cdot z_{2,2,k}) = 0 \tag{5.58}$$

$$v_{2,k} - (W_{2,2,0} + W_{2,2,1} \cdot z_{2,1,k} + W_{2,2,2} \cdot z_{2,2,k}) = 0 \tag{5.59}$$

After solving the optimisation problem, I will obtain preliminary optimised weight matrices. For Layer 1:

$$\begin{bmatrix} W_{1,1,0} & W_{1,1,1} & W_{1,1,2} \\ W_{1,2,0} & W_{1,2,1} & W_{1,2,2} \end{bmatrix}$$

For Layer 2:

$$\begin{bmatrix} W_{2,1,0} & W_{2,1,1} & W_{2,1,2} \end{bmatrix}$$

I construct matrices of the attenuation/amplification factor for each layer to be of the same size as the weight matrices. For Layer 1:

$$\begin{bmatrix} \theta_{1,1,0} & \theta_{1,1,1} & \theta_{1,1,2} \\ \theta_{1,2,0} & \theta_{1,2,1} & \theta_{1,2,2} \end{bmatrix}$$

For Layer 2:

$$\begin{bmatrix} \theta_{2,1,0} & \theta_{2,1,1} & \theta_{2,1,2} \end{bmatrix}$$

The values of the attenuation/amplification factors θ's controls the impact of weights on the objective function. To find the sensitivity of θ's with regard to the objective function, the values of θ's are individually perturbed ($\pm\frac{1}{2}\varepsilon$) and the changes in the objective function is observed. The perturbation resulting in the largest change in the objective function is the most sensitive neuron and vice versa. To implement the process, initially I start with all $\theta = 1 + \frac{1}{2}\varepsilon$ where ε is a small user-defined value and calculate the value of the objective function. I then change the value of θ's to $1 - \frac{1}{2}\varepsilon$. I then multiply the θ's element-wise with the weights in each layer to gain a new objective value. This is effectively a element-wise matrix optimisation.

To find the sensitivity for blocks of layers, I perturb all the θ's connected with the block of the layers at the same time and evaluate the value of the objective function. In this way, I can identify the sensitivities of any number of layers in combination through finite difference. This is easier compared with automatic differentiation where complicated differentials need to be derived in order to find the sensitivities of blocks of layers.

I now generalise the formulation. For $k = 1, \cdots, N_k$ data points, the output of layer l given input form layer $l-1$ is:

$$z_{l,i,k} = f(y_{l,i,k}) \tag{5.60}$$

where:

$$y_{l,i,k} = \sum_{j=1}^{NN_{(l-1)}} W_{l,i,j} \cdot z_{(l-1),j,k} + W_{l,i,0} \tag{5.61}$$

for $l = 1, 2, \cdots, N_l$, $i = 1, 2, \cdots, NN_l$, and $j = 1, 2, \cdots, NN_{(l-1)}$, where NN_l is the number of neurons in layer l and $NN_{(l-1)}$ is the number of neurons in layer $l-1$. $W_{l,i,0}$ is the bias term. I obtain the weights through the optimisation process of Equation 5.35 and 5.36 and then define the matrices of θ's for each layer of weights. For layer l:

$$\begin{bmatrix} \theta_{l,1,0} & \cdot & \theta_{l,1,NN_{(L-1)}} \\ \vdots & & \vdots \\ \theta_{l,NN_l,0} & \cdot & \theta_{l,NN_l,NN_{(L-1)}} \end{bmatrix}$$

where $\theta = 0$ *or* 1. $\frac{\partial(LSQR)}{\partial\theta_{l',i'}}$ is found by perturbing θ's based on Equation 5.53.

5.4.2 Example Formulation

I similarly adopt the architecture of [4,5,3,2,2] to calculate the sensitivities of each neuron for 1,000 data points. The perturbation ε is set to be 1×10^{-6}. The approximated values of the sensitivities through finite difference method are tabulated in Table 5.8.

From Table 5.8, I find the neuron with the highest sensitivity is neuron (1,1) and the neuron with the lowest sensitivity is neuron (1,3). Therefore, I seek to duplicate neuron (1,1) or remove neuron (1,3) in the automatic search process. The layer with the highest sensitivity is layer 1 and the layer with the lowest sensitivity is layer 3. Therefore, I could duplicate layer 1 or remove layer 3.

I also tabulate the number of iterations and the CPU time for each optimisation process. Similarly, the processor used to run this code is the 2.3GHz Quad-Core Intel Core i5 with a memory setting of 8GB 2133MHz LPDDR3. From the results, I observe that the average CPU time for a single iteration is 2.327s and for the calculation of one sensitivity value is 0.0895s.

If I take the backward or the forward difference instead of the central difference, the total CPU time spent is less since the calculation of $\theta = 1$ is more easily calculated. The approximated values of the sensitivities through backward finite difference method are tabulated in Table 5.9.

Table 5.8 Sensitivity values of neurons with regard to the objective function calculated through central finite difference method.

θ	$f(\theta+\frac{1}{2}\varepsilon)$	Time(s)	$f(\theta-\frac{1}{2}\varepsilon)$	Time(s)	$\frac{\partial(LSQR)}{\partial\theta}$
$\theta_{1,1}$	1.0038671	0.038933	1.0038716	0.034208	-4.5713116×10^{-5}
$\theta_{1,2}$	1.0038468	0.019590	1.0038468	0.019838	2.1725223×10^{-6}
$\theta_{1,3}$	1.0038746	0.103405	1.0038753	0.098813	-5.9615236×10^{-5}
$\theta_{1,4}$	1.0038797	0.099894	1.0038804	0.10045	-7.2581343×10^{-5}
$\theta_{1,5}$	1.0038385	0.020573	1.0038383	0.019912	1.9588717×10^{-5}
θ_1	1.0039174	0.020815	1.0039190	0.020443	-1.5650015×10^{-4}
$\theta_{2,1}$	1.0038174	0.020605	1.0038169	0.020015	6.1821786×10^{-5}
$\theta_{2,2}$	1.0038229	0.019586	1.0038224	0.020965	5.0618858×10^{-5}
$\theta_{2,3}$	1.0040042	0.033760	1.0040074	0.037512	-3.1847834×10^{-4}
θ_2	1.0039489	0.0052710	1.0039510	0.000190	-2.0653320×10^{-4}
$\theta_{3,1}$	1.0042998	0.0202360	1.0043089	0.020118	-9.1393862×10^{-4}
$\theta_{3,2}$	1.0035660	0.0204222	1.0035607	0.019830	5.31730815×10^{-4}
θ_3	1.0040200	0.0079319	1.0040239	0.000275	3.9056441×10^{-4}

Table 5.9 Sensitivity values of neurons with regard to the objective function calculated through backward finite difference method.

θ	$f(\theta+\frac{1}{2}\varepsilon)$	Time(s)	$f(\theta-\frac{1}{2}\varepsilon)$	Time(s)	$\frac{\partial(LSQR)}{\partial\theta}$
$\theta_{1,1}$	1.0038519	0.109680	1.0038479	0.036438	4.0498089×10^{-5}
$\theta_{1,2}$	1.0038478	0.039457	1.0038479	0.020371	-2.146170×10^{-6}
$\theta_{1,3}$	1.0038483	0.196182	1.0038479	0.104112	4.848526×10^{-5}
$\theta_{1,4}$	1.0038484	0.221150	1.0038479	0.101935	5.585575×10^{-5}
$\theta_{1,5}$	1.0038477	0.227941	1.0038479	0.205130	-1.811979×10^{-5}
$\theta_1.$	1.0038490	0.136057	1.0038479	0.020925	1.2397074×10^{-4}
$\theta_{2,1}$	1.0038473	0.256144	1.0038479	0.020091	-6.083805×10^{-5}
$\theta_{2,2}$	1.0038473	0.154538	1.0038479	0.020250	-5.012097×10^{-5}
$\theta_{2,3}$	1.0038509	0.169131	1.0038479	0.020759	3.125831×10^{-4}
θ_2	1.0038499	0.009998	1.0038479	0.000395	2.016290×10^{-4}
$\theta_{3,1}$	1.0038570	0.115276	1.0038479	0.021455	9.121332×10^{-4}
$\theta_{3,2}$	1.0038418	0.073981	1.0038479	0.020555	-6.051310×10^{-4}
$\theta_3.$	1.0385093	0.042776	1.0038479	0.000649	3.070853×10^{-4}

5.4.3 Validation with Automatic Differentiation

The results from finite difference is less rigorously derived and calculated than the results from automatic differentiation. However, both methods serve to produce the same evaluation of sensitivities of individual neurons with respect to the objective. The two methods can serve to validate each other.

In both cases, it is identified that the last layer has the highest sensitivity values, followed by layer 2, then by layer 1. Both results corroborate with each other in terms of neuron importance. The difference is that in layer 2 and last layer, the raw values of the sensitivities calculated are different. For example, sensitivities of layer 2 calculated from automatic differentiation are of the scale 10^{-3} whereas in finite difference, it is calculated to be of the scale $10^{-5} - 10^{-4}$. This can be attributed to the finite difference method, being a less rigorous method, can introduce errors based on the shape of the objective function. I have to assume that the deviation from the original point is minimal such that I can arrive at an accurate estimator of the gradient. It is as effective when used to perform ranking of neurons as the automatic differentiation method.

I further compare the advantages of the two methods. The key advantage of automatic differentiation is:

- It does not rely on assumptions and the calculated results are rigorously proved.
- Not counting the optimisation step, it is faster than finite difference method for a small network.
- All θ values are set to 1 or 0 hence are easier to optimise or calculate.

The key advantage of finite difference method is:

- It does not require a run of optimisation in order to generate the sensitivity values.
- It is easy to formulate, *i.e.* no complicated mathematical derivations are required to find the sensitivity values.
- It is possible, without further derivations, that sensitivity values of a whole layer can be calculated.
- The sensitivity values do not need to be limited to one neuron or one layer. Any block of layers or sub-layers can be evaluated for sensitivity with respect to the objective function.

Overall, the two methods have their own advantages. However, if I would like to go into more advanced applications, the finite difference method is easier to implement and is more flexible to be adopted in different scenarios.

5.5 Autonomous Learning

This section designs an autonomous learning ANN that restructures its topology by automatically evolving its architecture. The autonomous learning system should be minimal in terms of user-input such that only a few values are adopted in arriving at the best architecture. In Section 5.5.4, I have performed two processes of evolving the architecture manually. This section seeks to automate this process.

I identify several aims and objectives to the automation task.

- The algorithm should operate with a bare minimum of user-defined input/control parameters and instructions
- The algorithm should be able to run in automatic mode, adapting to a continuously available, real-time input-output data stream from the process to be modeled with the ANN
- The aim is to have an online, real-time ANN model adaptation algorithm.
- The algorithm should "contact" its user only if significant, unrecoverable failures are encountered during real-time run-time.

The overall aim is to produce the smallest size ANN possible to fit a given dataset, and then on to be able to adapt its size and topology in an autonomous, optimal manner.

5.5.1 User-Input Variables

I enlist the user-input variables to the autonomous system. The number of user-input variables is minimal.

- Define an upper bound on the LSQR error desired: ε_{LSQR} (e.g. 10^{-3}, 10^{-6}, $etc.$)
- Define for ANN:
  (1) the maximum number of layers: NL_{max} ($e.g.$ 100, 500,$etc.$)
  (2) the maximum number of neurons per layer: NN_{max} ($e.g.$ 100, 200, $etc.$)

Note that the minimum number of layers is 1 and the minimum number of neurons per layer is also set to 1.

5.5.2 Initial ANN Structure

I have the following options for initialisation of ANN structure.

- Option A: A user-defined arbitrary structure
- Option B: A user-defined structure, from previous run/fitting the same underlying system or process; *i.e.* having the same inputs and outputs
- Option C: A user-defined "sufficiently large" initial ANN which is to be systematically reduced to satisfy user tolerances
- Option D: A user-defined "minimal size" initial ANN which will be systematically added to or removed from in terms of individual neurons and entire layers as required.

In all cases the network is to be basically adjusted in terms of architecture dynamically (iteratively) so as to find an ANN configuration that has the minimal number of neurons and layers to satisfy user tolerances for a given process dataset.

5.5.3 Algorithm Description

The "autonomous ANN restructuring algorithm" should be tightly focused at first to the clearly defined single aim ("objective function") of the ANN restructuring design algorithm.

The discrete nature of adding or removing a node or an entire layer from an ANN with a given structure, results in a non-differentiable optimisation problem when trying to find the minimal-structure new ANN that satisfies user-specified least-squares fitting tolerances. When modeled as a mathematical programming problem, the above stated ANN-structure optimisation problem would result in a non-convex Mixed-Integer Nonlinear Programming (MINLP) Problem. Although MINLP is more intuitive, the highly non-convex complicated shape of the ANN objective function makes it computationally impossible to solve the problem of architectural search with MINLP.

Our approach cannot be viewed that it can match exactly any real-valued tolerances, not only because of its mildly heuristic nature, but also crucially because of the underlying problem nature, as per the previous analysis.

Therefore, I propose a dead-band, or buffer region, where the algorithm would terminate once in this buffer zone of, say, the LSQR error value:

- $\varepsilon_{LSQR,target} \in (0,1)$
- User specifies a dead-band width around which I terminate: $\delta_{LSQR,target}$

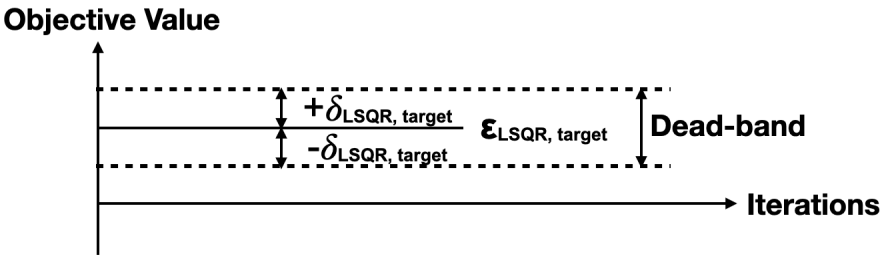


Figure 5.2 Illustration of the concept of dead-band.

I obtain the following properties:

- $0 < \varepsilon_{LSQR,target} < 1$
- $0 < \delta_{LSQR,target} < \varepsilon_{LSQR,target}$

The dead-band serves to fully guarantee the termination of the optimisation process and convergence to user criteria. Figure 5.2 demonstrates the concept of the dead-band in the optimisation process.

The dead-band, or buffer zone of no-action, comes from the very difficult-to-avoid oscillation and "humming" (rapid on-off behaviour) of the online controllers in Control Engineering technology, and from a very old-established practical implementation idea.

5.5.4 Experimentation with Autonomous Learning

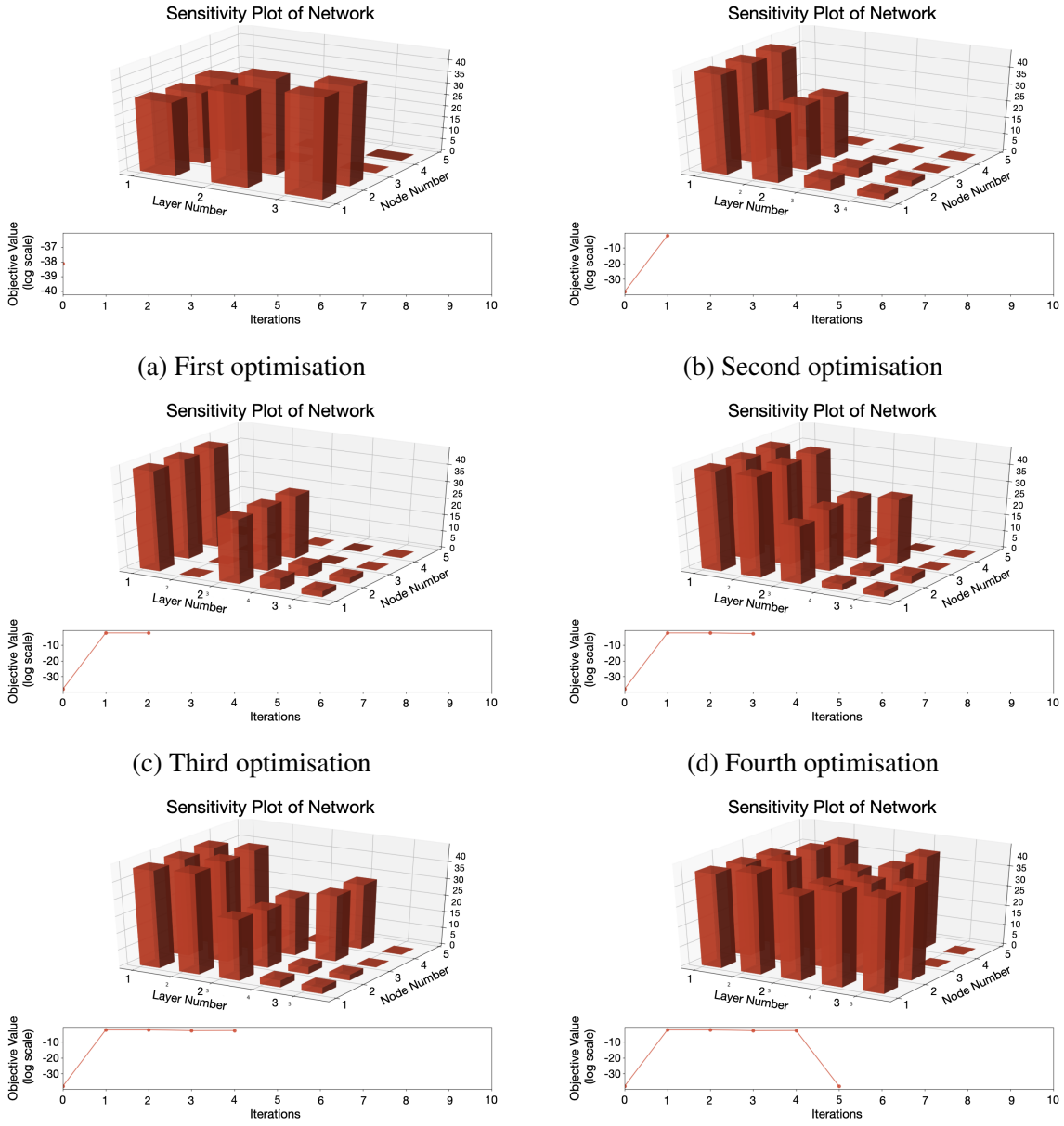
I have applied a series of architectural modifications to an user-defined initial network with 5,000 input data points. The aim is to confirm the functionality of the subroutines created including node addition/removal and layer addition/removal. The process of architectural modification is currently "supervised" and is performed manually. This step lays the foundation for further automation of the architectural search process. The dataset used is still the nonlinear process I described in Section 4.8.1.

I describe two experiment with the construction of the network corresponding to the same dataset : (1) add nodes and layers from a minimally possible network (*Case Study I*), and (2) remove nodes and layers from a larger than necessary network (*Case Study II*).

Case Study I

Suppose I start from a minimally possible network with 2 hidden layers of 3 and 2 neurons each. Figure 5.3a demonstrates the sensitivities of the corresponding neurons. The neuron

sensitivities are plotted on a negative log-scale and the raw numbers of the sensitivities are smaller than 1. Hence, on the negative log-scale, the higher the value of the bar chart, the lower the sensitivity. Therefore, I duplicate neurons with high sensitivity (low bar value) and remove neurons with low sensitivity (high bar value). The series of optimisation performance and changes in sensitivity values are presented in Figure 5.3.



(a) First optimisation

(b) Second optimisation

(c) Third optimisation

(d) Fourth optimisation

(e) Fifth optimisation

(f) Sixth optimisation

Figure 5.3 The sensitivities (negative log-scale) of the network neurons in a series of optimisation processes. The line plot below demonstrates the changes in the value of the objective function (log-scale).

From Figure 5.3a, I can observe that there are three layers in total, two hidden layers and one output layer. The hidden layers consist of 3 and 2 neurons each and the output layer consists of 2 neurons. The positions where the values of sensitivities are zero indicate the absence of neurons at those positions (*e.g.* layer 3 node 3 is empty). I also observe that nodes in the first layer have higher sensitivities compared to other nodes. Since this is a minimally possible architecture, I seek to add another layer and observe the effect on the objective function.

Figure 5.3b demonstrates the effects of adding a third layer with 3 neurons on the sensitivities. It can be observed that the first hidden layer has the lowest sensitivities and the second hidden layer has higher sensitivities, followed by the third hidden layer with highest sensitivities among all hidden layers. Therefore, as a next step, I add the 4^{th} node in the third hidden layer. It can also be observed that the value of the objective function has increased. While this means that network with two hidden layers is more optimal than network with three hidden layers, the former might be a local optimum. Adding one more layer may help to overcome the local optimality.

Figure 5.3c and 5.3d demonstrate the effects of further adding nodes on the layer with the highest sensitivities. It can be observed that during the process of adding the 4^{th} and 5^{th} neurons to the third hidden layer, the value of the objective function has not been significantly reduced. I further perform modifications to the network in order to achieve lower objective values.

I then add a 4^{th} neuron to the second hidden layer. I observe that the value of the objective function is significantly reduced, to a point lower than the first objective value. Moreover, I observe that the sensitivity values have been greatly equilibrated over the process, *i.e.* the sensitivity values are becoming similar. Therefore, the algorithm has equalised the importance of each neuron in arriving at an architecture with almost equally important neurons.

After this node addition step, the termination criterion is triggered. Therefore, the algorithm finishes at this step. In this way, I have finished the optimisation process of starting from a minimal network expanding to a larger network with equilibrated neuron importance. The final objective value arrived is 3.119×10^{-17}. I have enlisted the evolution of objective values in Table 5.10.

Case Study II

In the second case study, I present the optimisation process of a network with initialisation of a larger than necessary network. In this case, I start with an architecture of [4, 10, 10, 10,

Table 5.10 The objective values obtained through successive optimisations in *Case Study I*.

Optimisation Steps	Objective Value
First Optimisation	6.695×10^{-16}
Second Optimisation	0.331
Third Optimisation	0.257
Fourth Optimisation	0.154
Fifth Optimisation	0.0869
Sixth Optimisation	3.119×10^{-17}

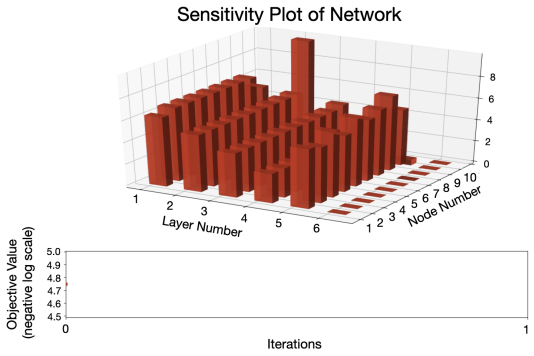
10, 2]. Only node removal and layer removal are involved in this case. The variations in sensitivity values are tabulated in Figure 5.4.

From Figure 5.4, it can be observed that the first and second layer, having the lowest raw sensitivity values (highest negative logarithmic values) are removed gradually, initially with 10 neurons each layer and later are evolved to have 4 and 3 neurons each. Observing the changes in objective value, it can be demonstrated that several ups and downs are present, indicating the possibility of reaching local minima and then overcoming the local minima to arrive at a lower value. Overall, the process is effective in reducing the number of parameters and shrinking the size of the neural network while achieving a comparable and lower objective value. The evolution of objective values are tabultaed in Table 5.11.

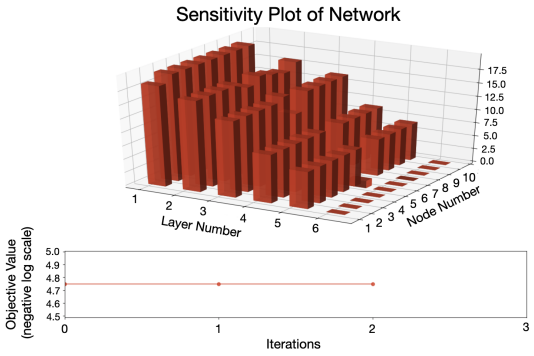
Table 5.11 The objective values obtained through successive optimisations in *Case Study II*.

Optimisation Steps	Objective Value
Iteration 1	8.890×10^{-3}
Iteration 3	9.004×10^{-3}
Iteration 6	9.000×10^{-3}
Iteration 9	8.684×10^{-3}
Iteration 12	8.673×10^{-3}

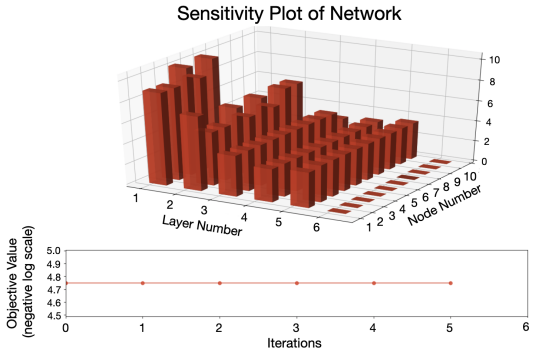
It is also observable that although with the same dataset, the two case studies reach differrent optimised architectures. This is because the nature of the DNN is complex such that different architectures with different weights can reach similar objective values within the deadband. The definition of the deadband and how the architecture is initially defined both affect the final architecture reached. Therefore, it is possible to discuss these effects in future work.



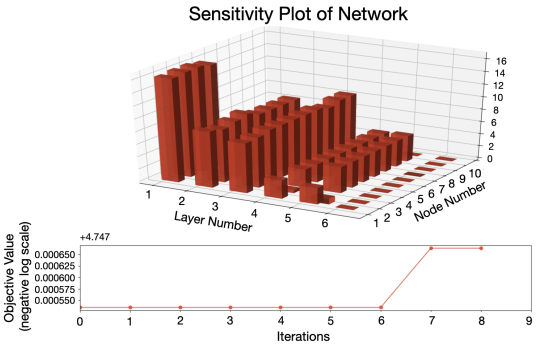
(a) Optimisation at Iteration 1



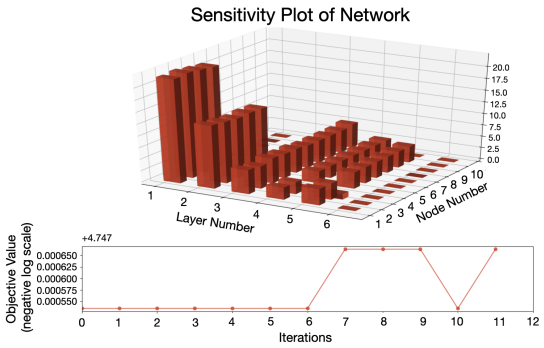
(b) Optimisation at Iteration 3



(c) Optimisation at Iteration 6



(d) Optimisation at Iteration 9



(e) Optimisation at Iteration 12

Figure 5.4 Sensitivity and objective plot of the optimisation process starting from a larger than necessary network.

5.5.5 Comparison with Backpropagation ANN

I employ the following process in generating an NLP formulation and compare it with the traditional backpropagation ANN values:

1. Solve unconstrained ANN with LSQR objective through backpropagation algorithm

2. Store the value of LSQR objective from the previous step
3. Use solution to initialise NLP formulation
4. Solve NLP formulation
5. Obtain objective value and compare it with the results from Step 2

Case Study I

I define the unconstrained ANN to have the architecture [4, 3, 2, 2] as the initial structure solved through backpropagation algorithm. I train through backpropagation on MSE loss with 50 epochs using the Stochastic Gradient Descent (SGD) solver. The objective value obtained from backpropagation is 0.5573. Solving using the NLP formulation, the objective value obtained is 3.119×10^{-17}. I observe that the objective value of both algorithms are close to zero but that obtained from NLP solution is much lower.

I also compare the loss with final architecture of the self-evolving algorithm, *i.e.* the architecture of [4, 3, 4, 3, 4, 2]. The final objective value obtained is 0.5461, which is also higher compared to the NLP formulation.

Case Study II

I define the unconstrained ANN to have the architecture of [4, 10, 10, 10, 10, 10, 2]. I train through backpropagation on MSE loss with 50 epochs using SGD solver. The objective obtained from backpropagation is 0.4316. In comparison, using the NLP solver gives an objective of 8.673×10^{-3}. I observe that both objectives are close to zero but the NLP solution gives a much lower objective value.

I also compare the loss with the final architecture of the self-evolving algorithm, *i.e.* the architecture of [4, 4, 3, 10, 10, 10, 2]. The objective value obtained is 0.4236. This is also higher than the objective value obtained from NLP formulation.

Overall, due to the deterministic nature and the core of the NLP formulation as an optimiser, the proposed algorithm is able to arrive at a solution much lower compared to traditional backpropagation algorithm.

5.6 Summary

In this chapter, I have implemented the subroutines to add/remove a connection, a node or a layer in the feedforward Artificial Neural Network (ANN) adopting the lifting scheme.

Sensitivity values with regard to the attenuation/amplification factors were calculated from either the automatic differentiation method or the finite difference method to represent neuron importance. An autonomous architecture evolving algorithm was introduced where the target objective value is described by a dead-band and if the objective value falls within the band, optimisation stops. Then I presented two case studies to demonstrate how the algorithm evolved the network from a minimal or larger than necessary initial architecture. I then compared the performance with traditional ANNs. Overall, I believe the method is novel and effective in autonomously evolving the architecture of ANNs.

I describe the longer-term aims of the optimisation scheme as possible future work related to the optimisation of ANNs.

- Given an ANN tuned for some purpose, *e.g.* to fit a given dataset:

  (1) Keep the same hidden, prior ANN structure and weights

  (2) Find an appropriate way to add extra layers and nodes so as to fit quickly the new dataset, while preserving the "knowledge" (fitting model) of the prior, initial ANN structure and weights. The goal is to preserve the ability to fit the same process Input/Output data up to that point, and to expand to include a new dataset with either minimal ANN modification, or no modification at all.

  (3) A way also has to be found to include the initial, prior ANN architecture (structure) and weights, while a new ANN of comparable size emerges automatically with this future algorithm evolution.

  (4) All these assume a close relationship between dataset 1 for the original ANN, and the new set, dataset 2, which will be used along with the old initial ANN, to emerge the new ANN that fits both dataset 1 and dataset 2.

- Evolution of non-layer based, but arbitrarily connected ANN network. For example, instead of fitting a standard feedforward neural network, I can design autonomous learning algorithms for feedback neural networks, or other more complicated connection schemes of an ANN in its most general form.

Overall, the proposed framework to self-evolve a neural network builds a good foundation to be further applied to more variants of ANN structure and has great potential to be further developed for different optimisation purposes.

Chapter 6

Hierarchical Multi-scale Parametric Optimisation of ANNs - *Part III*

6.1 Introduction

In Chapter 5, I developed an autonomous architecture evolving process where the neurons are added and removed based on sensitivity values of each neuron. In particular, two methods of evaluating sensitivities were presented: automatic differentiation and the finite difference method. The former is more rigorously developed but more computationally expensive. The latter is easy to implement and highly flexible, but requires more parameter settings to obtain sensible results. This chapter further shows that with the sensitivity values obtained, it is possible to propose the formulation of a more advanced network training process.

The sensitivity is an important measure when performing analysis on a neural network. This is because it is indicative of component importance for a neuron, a layer, or a block of layers. In particular, sensitivity dictates how much a perturbation in the value of the component will cause a change in the value of the output or the objective function. Neural sensitivity analysis has been widely adopted in the analysis of deep neural networks (DNNs) with the aim to demystify the "black-box" nature and add further metrics to identify how the network reacts to changes in the explanatory variables. Early stages of research focuses exclusively on how the output or the objective value changes with a perturbation in the input [202] [203]. While this is an important step in understanding how a network responds to different datasets, the analysis is limited by only focusing on the sensitivity of the explanatory variables [202]. Moreover, research on neural sensitivity analysis often focuses on the relative values of the weights or inputs, manipulated to become a sensitivity measure [202] [203] [204]. Different to these traditional definitions, our research demonstrates a new definition

of the sensitivity measure, by the adoption of an attenuation/amplification factor into the network, to allow the evaluation of sensitivities throughout the network.

Furthermore, I present a novel neural sensitivity analysis on the neural network which extends to all components in the network. The analysis can achieve the selective tuning of the network. In this case, I propose a novel framework for the training of the DNNs, where the sensitivity values guide a hierarchical multi-scale search down the binary tree representing layer importance. An efficient algorithm to search for the most sensitive layer to tune (in the case of training) is developed. The framework is simple to implement and efficient to run. It has key advantage in optimising large DNNs with a high number of layers to optimise due to its search efficiency of $O(logn)$. The method, relying on sensitivity values, is effective in producing optimised neural networks, and is a novel formulation of the optimisation processes used in the application of DNNs.

This chapter is organised as follows:

- Section 6.2 introduces the background of this research.
- Section 6.3 formulates the hierarchical multi-scale search algorithm.
- Section 6.4 demonstrates how the hierarchical multi-scale search method can be applied to the tuning of ANNs.
- Section 6.5 makes use of second-order information to redefine the hierarchical search process.
- Section 6.6 compares the performance of the adoption of first-order and second-order information.
- Section 6.7 applies the algorithm to large-scale problems and discusses the performance of large-scale applications.
- Section 6.8 then discusses relevant research areas.
- Section 6.9 concludes the chapter.

6.2 Background

Early stages of research in neural sensitivity analysis focus either on the values of the weights of the network [203] [204] [205], or on the sensitivity of the input values [202] [206] [207]. The aim is to observe the influence of the input on the output or the objective function, *i.e.*, the explanatory capacity of the network [207].

There are several key measures that have been used in the context of neural sensitivity analysis, as demonstrated in Table 6.1. While a variety of measures have been proposed, the sensitivities are often based exclusively on the values of the weights. A more practical method is to provide perturbations in the inputs and observe the effects on the outputs [208] [209] [210].

A sensitivity analysis on the weights has been demonstrated to be ineffective in measuring the network's functionality [202]. Sensitivity analysis on the input values is more widely adopted in image recognition [211] and engineering research [202], but is limited in application for cases with discrete inputs [202].

There are in general two ways to perturb the input for sensitivity analysis: 1) to restrict one input value while perturbing the rest of the variables and finding the changes in the objective/output [208]; and 2) to perturb one input value while keeping everything else unchanged [210].

Table 6.1 Sensitivity measures adopted in literature.

Sensitivity Measure	Equation	Reference
Numerical sensitivity measure	$\frac{\partial y_k}{\partial x_i} = f'(net_k)\sum_{j=1}^{l} v_{jk} f'(net_j) w_{ij}$	[202]
Weight product	$WP_{ik} = \frac{x_i}{y_k}\sum_{j=1}^{L} w_{ij} v_{jk}$	[203]
Q factor	$Q_{ik} = \frac{\sum_{j=1}^{L}(\frac{w_{ij}}{\sum_{r=1} w_{rj}} v_{jk})}{\sum_{i=1}^{N}(\sum_{j=1}^{L}(\frac{w_{ij}}{\sum_{r=1}^{N} w_{rj}} v_{jk}))}$	[204]

* Notations are described in Nomenclature

There are several measures of perturbation: 1) to implement a grid search where the inputs are perturbed to be their minimum, 1^{st} quartile, median, 3^{rd} quartile and maximum values [209] [212]; 2) to add an input neuron and evaluate the disturbance observed in the objective function [207] [213] [214]; and 3) to remove an input neuron and observe the change in objective function [209] [207] [214].

More recent work in image processing demystifies the convolutional neural network (CNN) by perturbing a pixel or a small region in an image and observing its effect on the objective [215]. Other studies seek to find partial derivatives of the image classification results with regard to individual pixels and visualise it in a graph, as a measure of input sensitivity [216] [217].

The advantages of the perturbation method is that it is simple to implement and communicates a clear message on how each variable interacts to give the objective value. The advantages of a partial derivatives method is that it represents a more rigorously developed calculative process that can be easily interpreted.

However, while these methods all merit in their own design, the sensitivity analysis is exclusively focused on the input importance. For importance of individual neurons, sensitivity analysis is less used and less well-researched. I present the representation of neuron importance in the next section.

Later work has developed methods to identify the importance of neurons through Neural Interpretation Diagrams (NIDs) [210] [218] [219] [220], which represent the relative magnitude of each connection weight by line thickness. The positive weights are viewed as "excitator signals" while the negative weights are viewed as "inhibitor signals". The diagram assumes that by tracking the path with thicker lines (higher positive weight values), it is possible to find input variables and neurons that are more important. However, the diagram will be difficult to visualise when the amount of connections is large, *i.e.*, with a large number of neurons.

Other studies focus on visualising the importance of neurons through a relevance score [221] [222] [223], calculated from Layer-wise Relevance Propagation [224]. This method is developed because images used as inputs contain a large number of pixels in each entry, thus making it impossible to disturb single pixels for sensitivity.

While these methods focus exclusively on visualisation of neural importance, the method used is either simplistic (by constructing graphs based on raw weight values) or highly complicated (by defining an equation of the relevance score). I propose a method of moderate complexity that allows the neural importance to be evaluated for tuning or for architecture evolution.

6.3 Mathematical Formulation

In this section, I consider the topological modification and design of ANNs in an automated way. The important idea in hierarchical multistage optimisation of underlying systems relies on the identification of structural blocks in the system that are either to be entirely excluded or to be included (or duplicated) in an automated manner using the novel concept of structural sensitivities. These are facilitated by the appropriate inclusion of an artificial continuous parameter "θ" : $\theta \in \{0,1\}$ which is relaxed as $0 \leq \theta \leq 1$. $\theta = 1$ indicates the inclusion of the structural equations/model block of the model while $\theta = 0$ indicates removal/absence of the block in question. It is important to note that in order to take full advantage of the capability of structural sensitivities, I can evaluate not only just individual neurons one-at-a-time, but also entire layers at once, or even entire larger subsets of layers of the ANN's analysed.

6.3.1 User-defined Parameters for Tuning of ANNs

I define the parameters required to initialise the ANN tuning process using the proposed method:

- Architecture: the number of neurons in each layer and the total number of layers of a network
- ε: a user-defined infinitesimal value to define the scale of perturbation in finite difference method
- Learning rate: an empirical parameter used to define the scale of backpropagation
- Maximum number of iterations: the maximum number of times to search for the most sensitive layer to optimise
- Number of epochs: the number of rounds of optimisation for a particular layer
- Tolerance: a small user-defined value to stop searching for another layer to optimise when the value is greater than the maximum gradient of the LSQR against the network weights

6.3.2 Multi-scale Hierarchical Analysis of ANNs

The key idea behind multi-scale hierarchical analysis of mathematical models of any type of system is to start from the highest level of abstraction, *i.e.* the entire system as an Input/Output (I/O) system. Then from this highest view-point of the system, as encapsulated by the equations comprising our model, I effectively employ a set bisection approach to proceed each time downwards in terms of the scale of detail considered at the nodes of an ever-expanding binary search tree.

This continues with bisection to refine the level of details considered to be optimised during the major iterations of our design algorithm.

Refinement of binary search-tree sub-levels (nodes) continues until user-defined criteria are met or the final level mathematical detailed description is reached during the refinement iterations of our binary search (bisection) algorithm.

Example ANN

Let us consider an example ANN with the following set of layers, with an Input layer, an Output layer, and 7 hidden layers: ANN Layer Set = {Input, 1, 2, 3, 4, 5, 6, 7, Output}.

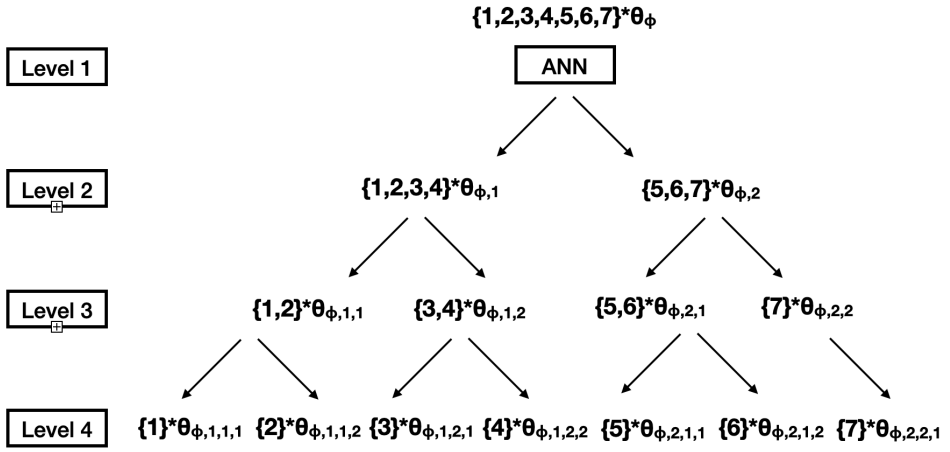


Figure 6.1 The binary tree partitioning the ANN with sensitivity parameter θs.

The structural optimisation of the ANN topology (architecture) is going to be based on evaluating the novel structural sensitivity measures. Furthermore, to facilitate a multi-scale efficient maneuver of identifying rapidly large parts of an ANN that require structural optimisation, it is necessary to introduce an arbitrary and yet a very natural way of viewing the hierarchically partitioning/decomposing decisions within the ANN automated design optimisation algorithm. Binary tree partitioning of ANN and artificial θ-parameters for structural sensitivity calculation is demonstrated in Figure 6.1.

Thus, following the structural sensitivities calculating scheme, in 3 finite difference steps I can identify the most sensitive/important layer in the ANN from the set of the 7 hidden layers.

The associated artificial parameters θ_i with the model of the previous binary tree is given in Figure 6.2.

Generally speaking, an upper bound on the number of steps required to identify the most contributing layer in the ANN, requires at the most $[\log_2(NL)]$ steps where NL is the number of hidden layers in the network.

Similarly then, if each layer has a fixed number of neurons, then to identify the most sensitive neuron in the previously identified hidden layer, I will require at the most $[\log_2(NN)]$ finite difference steps, where NN refers to the number of neurons in a particular layer.

Overall, the effort to identify a single artificial neuron is thus given by:

$$Effort \propto [log_2(NL)] + [log_2(NN)] \tag{6.1}$$

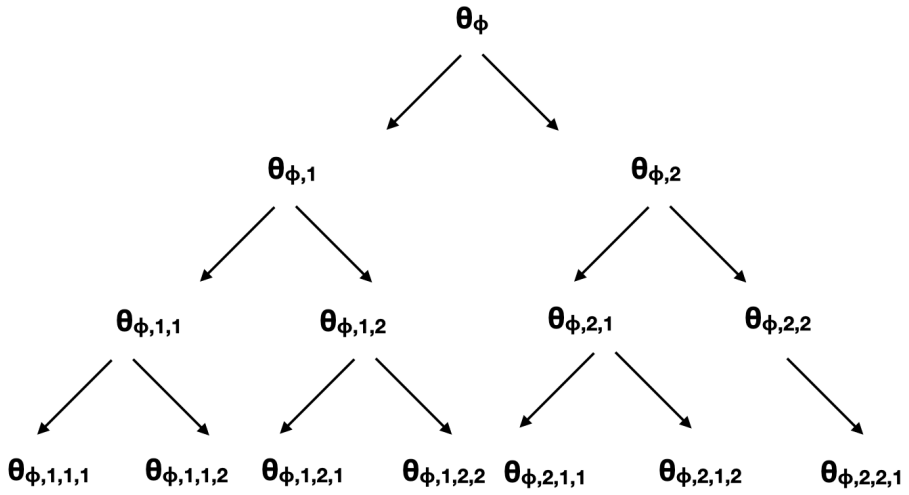


Figure 6.2 The parameters partition associated with the binary tree partitioning of the ANN.

Clearly, the multi-scale hierarchical analysis of the ANN system requires a logarithmic number of steps to reach any level of "scale" of encapsulation based on a binary, set bisection analysis tree.

It is noted that I do not need for very large-scale ANNs to consider descending to the level of individual neurons or even layers. For large ANNs, I can add or remove in fact entire blocks of neurons within a layer as well as entire blocks of hidden layers at a time according to the structural sensitivity values at any level of analysis using the binary tree set partitioning with each level of the binary tree signifying a different block size for removal or addition locally in the topology/architecture of the ANN that is being designed and structurally optimised with the proposed scheme.

With this scheme, the entire process can be fully automated in a real-time implementation of what is in essence a true highly efficient novel ANN design algorithm.

6.4 Network Tuning

One important implementation of this scheme is to tune the weights of the network. I adopt the sensitivity scheme to identify the most sensitive neuron or blocks of neurons that have the greatest impact on the objective value. During tuning, I keep the the value of other weights constant and re-optimise the network by only changing the values of the neurons that have been identified to be "most sensitive". This then enters into an iterative process where the new sensitivities are found and the network is re-turned. I repeat this iterative process until a satisfactory objective value is found.

I start from randomised weights and identify the most sensitive layers. Each time, I only optimise the most sensitive layers. The optimisation used can be anything ranging from backpropagation to derivatives-free algorithms (*e.g.* Genetic Algorithm). Here I use backpropagation as an example. The parameters are updated using an update rule corresponding to the optimisation scheme. For the simplest Stochastic Gradient Descent, the update rule is:

$$W' = W - \alpha \times \frac{d(LSQR)}{dW} \tag{6.2}$$

where W's are the weights, α is the learning rate, and $LSQR$ is the objective function.

The number of iterations used in each optimisation step can be arbitrarily small as empirical implementation has demonstrated that even a small number of iterations can bring good optimisation results by iteratively focusing only on the most sensitive layers. This is demonstrated in Section 6.4.2.

To find the most sensitive layer of neurons to optimise, I implement a binary tree search method. Starting from the first level of division, where the whole network is divided into two parts, I focus on the part of the network that has a higher sensitivity. Then inside that part, I further divide the block of layers into two blocks. I keep dividing until the smallest unit is one layer and sensitivities are calculated along the division process. To identify the most sensitive layer, I conduct a search down the binary tree until reaching the smallest unit, always choosing the higher sensitivity value along the path. In this way, the most sensitive layer can be found efficiently with $O(log_2 N)$. For each iteration, I only optimise that layer using backpropagation. After optimisation, I perform another binary tree search until the convergence criterion is fulfilled.

Initial implementation demonstrates that the selection of layers is often trapped in a loop, where the neuron having the highest sensitivity is always the same and is optimised repeatedly. To overcome this hurdle, I implement a random factor in the selection of the neuron to optimise, similar to the ε-greedy algorithm in reinforcement learning [225]. In fact, this implementation of the random factor is a trade-off between "exploration" and "exploitation" as I allow the neuron with lower sensitivities to have the chance to be optimised.

Another issue that is solved by introducing the random factor is when the left branch and the right branch have roughly the same sensitivity. This makes the selection process difficult. This is also frequently observed and requires a level of randomness to select the layer to optimise.

When I have practically equal sensitivities at a branching point, then I clearly have the same impact indicated for left and right branches, respectively. This means that to choose to branch left or right, I have to somehow take into account the fact that the weight indicated by a branching point structural sensitivity should reflect the "probability" of that branch to be

chosen. Our algorithm overall is deterministic, and will remain so with a judicious choice of a randomization step to break the problem of having to choose effectively numerically equal branching at points.

The algorithm is developed as such: suppose I call "S_left" and "S_right" the respective absolute values of the structural sensitivities at a branching point during the level exploration phase. These sensitivities are used to arrive at the next layer whose tuning weights are to be optimised together, while holding all other weights of all other layers constant, *i.e.*:

$$S\_left = AbsoluteValue(s\_left) \tag{6.3}$$

$$S\_right = AbsoluteValue(s\_right) \tag{6.4}$$

where s_left and s_right are the sensitivity values on the left and right branch at the branching point. Using the absolute values of the structural sensitivities at a branching binary search tree exploration step, choose the left or right branch based on a randomised algorithm which is applied irrespective of the values of the structural sensitivities. I follow the steps:

1. Generate a random real number (the random factor) between 0 and 1: r=RandomReal([0,1))
2. Calculate the probability for left and right branch:

$$branch\ left\ probability = S\_left/(S\_left + S\_right) \tag{6.5}$$

$$branch\ right\ probability = 1 - branch\ left\ probability \tag{6.6}$$

3. Make use of the random factor. If branch-left probability is greater than the random factor, branch left. If not, branch right.

The random factor is used throughout the binary search process and when used for sufficiently long time, it will be able to break "ties" among the branching points in a binary search tree.

The pseudo-code is presented in Algorithm 9.

6.4.1 Convergence Criterion

The convergence criterion adopted in this algorithm is based on the partial derivative of the objective function with regard to the weights. For every 50 rounds of iteration, I check the value of the derivative of objective function with regard to the weights. If the norm of the value is smaller than a user-defined tolerance (usually 10^{-5} to 10^{-6}), I stop the iteration. To

Algorithm 9 Network Tuning Adopting Hierarchical Multi-scale Parametric Optimisation

Read training and testing I/O data points.
Initialise architecture of neural network.
Initialise sensitivity (θ) matrices and weight (W) matrices for each layer
Initialise a user-defined infinitesimal value ε used for perturbation

procedure REFRESH_THETA(Network Structure)
    Define θ matrices based on the Network Structure. Set θ matrices values to be 1.
    return θ matrices
end procedure

procedure PERTURB_THETA(Network Structure, Perturbation, Leftmost Layer To Perturb, Number Layers To Perturb, Perturbation Side)
    Refresh_Theta (Network Structure)
    Choose perturbation side
    Layers to perturb are found by:
    if <Perturbation side on the left> **then**
        Left bound to perturb = Leftmost Layer To Perturb
        Right bound to perturb = Leftmost Layer To Perturb + $\frac{1}{2}$Number Layers To Perturb
    else
        Right bound to perturb = Leftmost Layer To Perturb + $\frac{1}{2}$Number Layers To Perturb + 1
        Right bound to perturb = Leftmost Layer To Perturb + Number Layers To Perturb
    end if
    for `<layers in the left bound to right bound>` **do**
        Set θ matrices values to be $1-$Perturbation
    end for
    return θ matrices
end procedure

procedure WEIGHT_TIMES_THETA(Network Structure, Theta Matrices, Weight Matrices)
    Calculate the element-wise product of θ and W matrices.
    Assign the values to New Weight Matrices
    return New Weight Matrices
end procedure

```
33: procedure CALCULATE_SENSITIVITIES(Theta Matrices, Weight Matrices, Network
    Structure, Leftmost Layer To Calculate, Number Of Layers To Calculate)
34:     Initialise Weight Matrices
35:     Forward propagate the weights in Weight Matrices
36:     Calculate the objective function value, denoted as Original Loss
37:
38:     Initialise θ with Refresh_Theta (Network Structure)
39:     Perturb_Theta(Network Structure, ε, Leftmost Layer To Calculate, Number Of
    Layers To Calculate, Left Side)
40:     New Weight Matrices = Weight_Times_Theta (Network Structure,Theta Matrices,
    Weight Matrices)
41:     Forward propagate the weights in New Weight Matrices
42:     Calculate the objective function value, denoted as Left Side Loss
43:     Left Sensitivity = (Left Side Loss - Original Loss) / ε
44:
45:     Initialise θ with Refresh_Theta(Network Structure)
46:     Perturb_Theta (Network Structure, ε, Leftmost Layer To Calculate, Number Of
    Layers To Calculate, Right Side)
47:     New Weight Matrices = Weight_Times_Theta (Network Structure, Theta Matrices,
    Weight Matrices)
48:     Forward propagate the weights in New Weight Matrices
49:     Calculate the objective function value, denoted as Right Side Loss
50:     Right Sensitivity = (Right Side Loss - Original Loss) / ε
51:
52:     return Left Sensitivity, Right Sensitivity
53: end procedure
54:
55: procedure TRAIN_MODEL(X Train, Y Train, Maximum Number of Epochs, Left
    Bound, Right Bound, Iteration Number)
56:     if <Iteration Number == 1> then
57:         Initialise parameters randomly
58:     else
59:         Populate previously populated parameters
60:     end if
61:     for <Increasing Epochs> do
62:         Forward propagate X Train values in the network
63:         Calculate the objective value
64:         Backpropagate to find the gradients of parameters and update Parameters
65:         for <layers in Left Bound to Right Bound> do
66:             Alter layer parameters by the update rule
67:         end for
68:     end for
69:     return Parameters, Cost
70: end procedure
71:
```

```
72: procedure CALCULATE_PARTIAL_DERIVATIVE(Gradients, Norm=1)
73:     if <Norm == 1> then
74:         Value = Sum of absolute values of Gradients
75:     else if <Norm == 0> then
76:         Value = Maximum value in Gradients
77:     end if
78:     return Value
79: end procedure
80:
81: procedure FIND_LEFT_RIGHT_BOUND(Sensitivity)
82:     Align Sensitivity into Levels
83:     for <each Level> do
84:         Obtain Left Sensitivity and Right Sensitivity
85:         Calculate Left Probability and Right Probability
86:         Generate a Random Number between 0 and 1
87:         if <Left Probability >= Random Number> then
88:             Choose left side
89:         else
90:             Choose right side
91:         end if
92:         Identify Leftmost Layer of interest, and Number of Layers Branch (left and right)
93:         if <Last Level> then
94:             Left Bound = Leftmost Layer
95:             Right Bound = Leftmost Layer + 1
96:         else
97:             Left Bound = Leftmost Layer of interest
98:             Right Bound = Leftmost Layer + Number of Layers in Branch
99:         end if
100:    end for
101:    return Left Bound, Right Bound
102: end procedure
103:
104: Initialise: Parameters, Backpropagation Derivative, Left Bound, Right Bound, Left
     Bound List, Right Bound List
105: Input: X Train, Y Train, Network Architecture, Tolerance, Maximum Number of
     Epochs, Maximum Number of Iterations
106: Count: Iteration Number
107: while <Backpropagation Derivative > Tolerance> do
108:    Left Bound = last element in Left Bound List
109:    Right Bound = last element in Right Bound List
110:    Parameters, Cost = Train_Model (X Train, Y Train, Maximum Number of Epochs,
     Left Bound, Right Bound, Iteration Number)
111:    Left Bound, Right Bound = Find_Left_Right_Bound (Sensitivity)
112:    if <Iteration Number % 50 == 0> then
113:        Backpropagate to obtain gradients of LSQR with respect to weights
114:        Backpropagation Derivative = Calculate_Partial_Derivative (Gradients)
115:    end if
```

if <Iteration Number > Maximum Number of Iterations> **then**
    Break
end if
Append Left Bound to Left Bound List
Append Right Bound to Right Bound List
end while
Output: Cost, Left Bound List, Right Bound List, Parameters

Table 6.2 The hyperparameters used in the example formulation to tune the network.

Hyperparameter	Value
Architecture	[4,5,5,5,5,5,5,5,5,2]
ε	0.001
Learning rate	0.01
Number of Epochs in each Iteration	1000
Total Number of Iterations	100
Tolerance	10^{-5}

formulate mathematically, suppose I adopt the L1-norm, I end the iteration when:

$$\left|\left|\frac{\partial LSQR}{\partial W}\right|\right|_1 < tolerance \tag{6.7}$$

I also limit the total number of iterations in case the algorithm cannot converge to the tolerance. This acts as a break from the loop when the total number of iterations is too large.

6.4.2 Example Tuning

In this section, I run an example of the tuning of the deep neural network. As I only focus on the tuning functionality, no changing of architecture is involved and the architecture adopted is arbitrary. In this case, I experiment with an architecture of [4,5,5,5,5,5,5,5,5,2]. The optimisation is performed with the hyperparameters tabulated in Table 6.2. The total number of data points used is 10,000, obtained through randomly sampling from the kinetics dataset.

I obtain the series of optimisation results from this architecture as demonstrated in Figure 6.3. Figure 6.3 demonstrates the change in sensitivity values over the process of optimising the network one layer at a time. The 3D bar chart on the top of each figure represents how the sensitivity values change for each division along the architecture. Thus, I see 2 values in the first level of division, 4 in the second level of division, *etc.*. This is because the total number of layers are divided by 2 each time. The line plot on the bottom represents the variations in

the value of the objective function. The figures represent the optimised results at iteration 1, 20, 40, 60, 80 and 100 respectively.

From Figure 6.3, I can see that the sensitivities are quickly equilibrated with a minimal number of iterations. In the first iteration, the sensitivity value is quite high for division on the right-hand-sides. I can only observe three bars because the values for these three bars are so high that it suppresses the display of sensitivity values for the other bars. This demonstrates how different the sensitivity values are at initiation.

At iteration 20, I can see that the sensitivity values have been equilibrated, *i.e.* the values are very similar (although in raw numbers they are different). The values of the objective has been decreasing and there are three plateaus in the objective value. The plateaus corresponds to local minima and this result demonstrates that the algorithm is capable of overcoming the local minima. Another local minimum that takes a long time to be overcome is displayed in Figure 6.3d, where a large plateau is observed, which is overcome in Figure 6.3e.

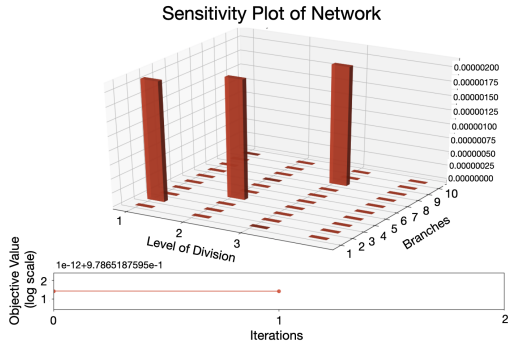
At iteration 100, I observe that the objective value keeps decreasing while the sensitivity values are roughly of the same value. This demonstrates that all neurons are playing an important part in the optimisation process and the objective value is decreasing with further optimisations.

With the introduction of the random factor, the layers that are optimised rotate among available layers instead of looping around a few. I plot the distribution of the layers being optimised during the 100 iterations in Figure 6.4. I observe that there is a reasonable distribution of layers being optimised and the optimisation is no longer stuck in a loop only optimising a single layer with the highest sensitivity.

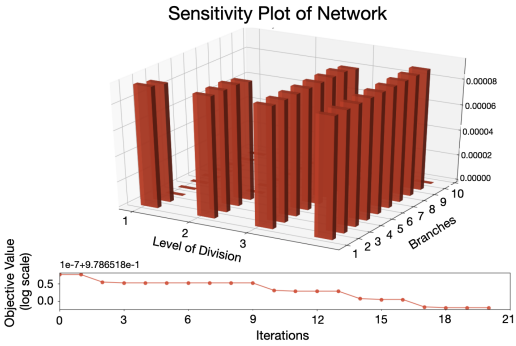
6.4.3 Comparison with End-to-end Backpropagation Algorithm

The conventional method to optimise a network is through backpropagation. Thus, I would like to compare the performance of the proposed algorithm with the end-to-end backpropagation method. I set the end-to-end backpropagation algorithm also running in iterations with the same set of hyperparameters. Each time, the end-to-end backpropagation algorithm optimises all layers instead of one selected layer, as in the case of the proposed algorithm. I run the end-to-end backpropagation algorithm for 1,000 number of epochs in each iteration. The number of epochs is arbitrarily set large to ensure optimisation to the optimal point. By having the same settings for either algorithm, this allows comparison between the end-to-end method and the multi-scale hierarchical optimisation method.

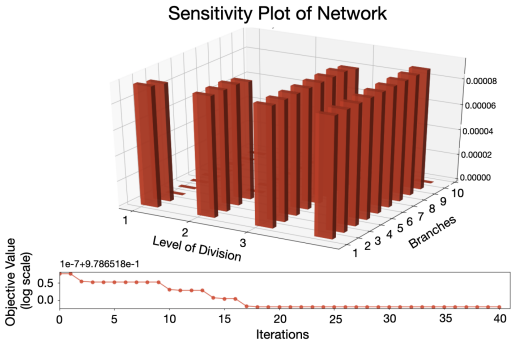
The same convergence criterion is applied to the end-to-end optimisation using backpropagation. In backpropagation, the derivatives of the objective function with regard to the weights are calculated at every epoch. Therefore, the check is performed every round



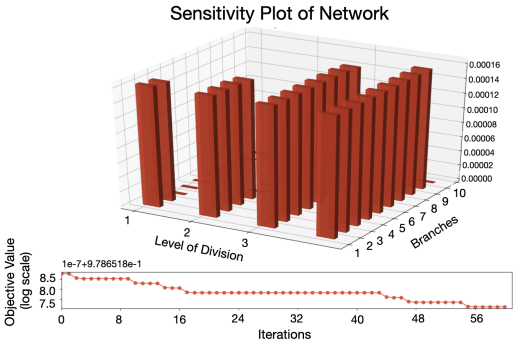
(a) First optimisation



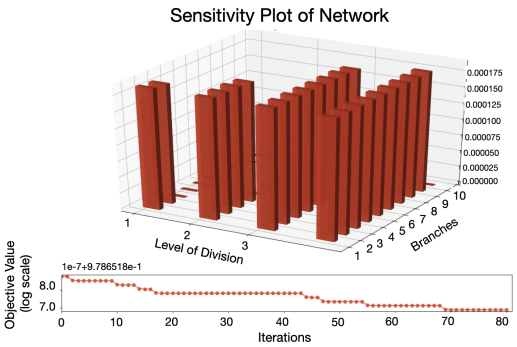
(b) Second optimisation



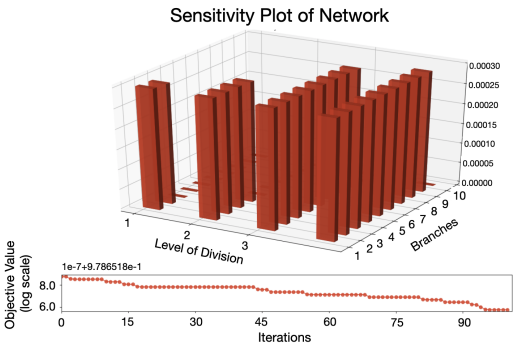
(c) Third optimisation



(d) Fourth optimisation



(e) Fifth optimisation



(f) Sixth optimisation

Figure 6.3 The process of optimisation in the example formulation with neural architecture [4,5,5,5,5,5,5,5,5,2] using 10,000 data points. The 6 figures represents the optimisation results at iteration 1, 20, 40, 60, 80 and 100 respectively.

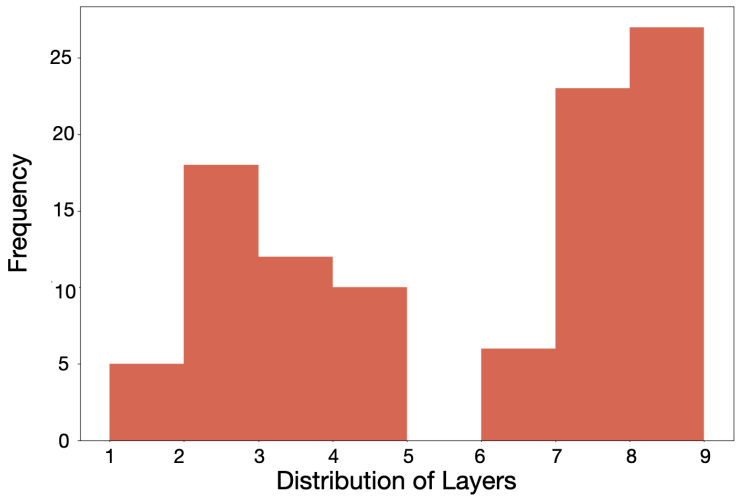


Figure 6.4 The distribution of layers selected in the optimisation process.

Table 6.3 The hyperparemters used in the comparison of performance between end-to-end backpropagation method and the multi-scale hierarchical search method.

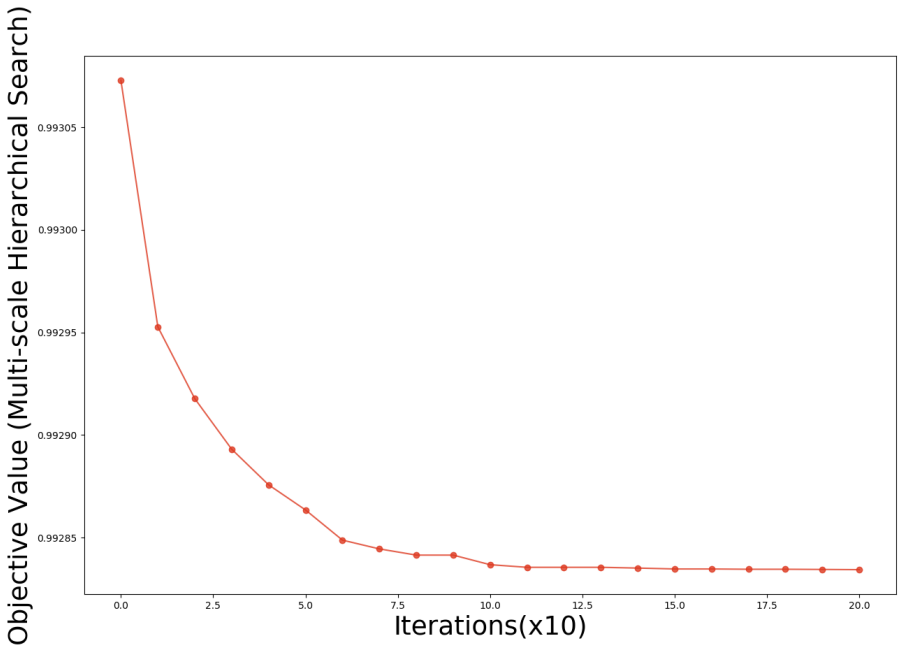
Hyperparameters	Values
Architecture	[4,5,5,5,5,5,5,5,3,2]
ε	0001
Number of data points	10000
Learning rate	0.0001
Number of epochs in each iteration	1000
Upper limit on number of iterations	200
Tolerance	0.001

instead of for every 50 rounds. I adopt the same convergence criterion for the multi-scale hierarchical search algorithm and for the backpropagation algorithm as a comparison.

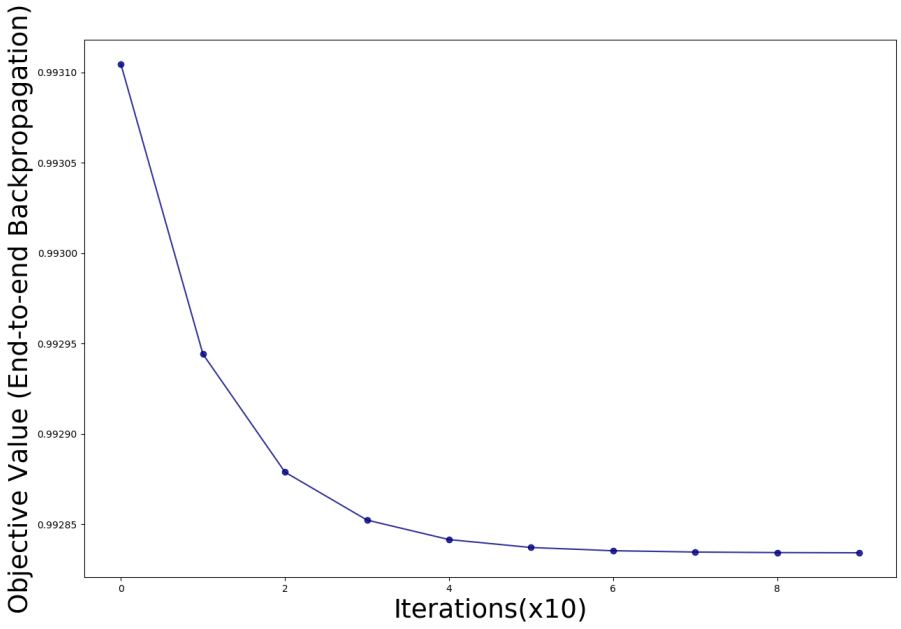
Hyperparameters. I list the hyperparameters used in the comparison of the two method in Table 6.3. The hyperparameters are selected arbitrarily with a random search for optimality.

Objective. The objective value obtained from end-to-end optimisation is 0.992952. The objective value obtained from multi-scale hierarchical method is 0.992834. The values of the objectives are comparable, with the multi-scale hierarchical search method reaching a slightly lower objective value.

I also plot the value of the objective function against iterations. This is demonstrated in Figure 6.5.



(a) Multi-scale Hierarchical Search



(b) End-to-end Backpropagation

Figure 6.5 The comparison between the optimisation performance of the end-to-end back-propagation method and the multi-scale hierarchical search method.

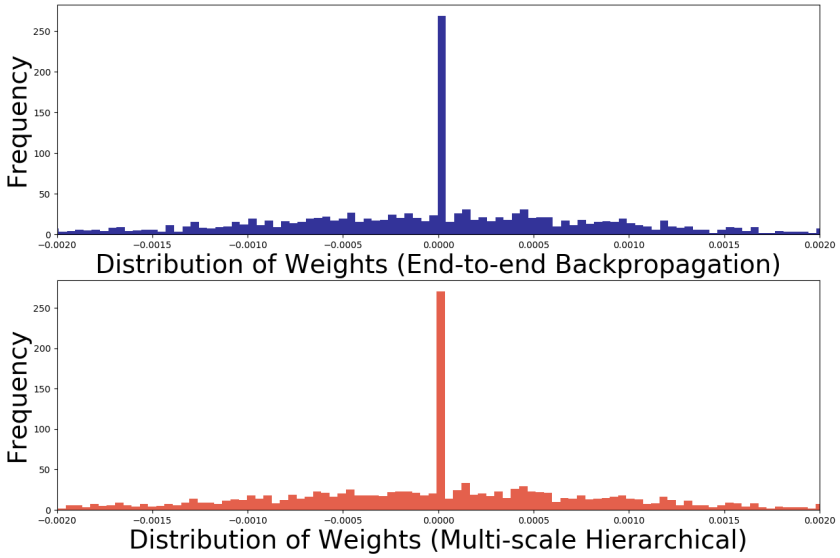


Figure 6.6 The distribution of weights obtained from end-to-end backpropagation and multi-scale hierarchical optimisation.

Optimisation Time. The CPU time taken for the end-to-end optimisation is 111.587 seconds. In comparison, the CPU time taken for the multi-scale hierarchical optimisation method is 2773.410 seconds. Although optimising to a smaller objective value, the multi-scale hierarchical search method is taking longer.

Weight Values. I check the values of the weights to observe whether the two algorithms arrive at the same local minimum. The distribution of the weights are plotted in Figure 6.6 and it is observed that the weights are distributed in a similar way. This indicates that there is a high possibility that the two optimisation methods optimise to the same local minimum, which is further corroborated with the fact that the values of the objective functions are very similar.

6.5 Second-order Information

6.5.1 Mathematical Formulation

I consider the local second order Taylor expansion of function $f(x,y)$:

$$f(x+\Delta x, y+\Delta y) = f(x,y) + \frac{\partial f}{\partial x}(\Delta x) + \frac{\partial f}{\partial y}(\Delta y) + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(\Delta x)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial y^2}(\Delta y)^2 + \frac{\partial^2 f}{\partial x \partial y}(\Delta x)\cdot(\Delta y) + Higher\ Order\ Terms\ (H.O.T) \tag{6.8}$$

If second order information is recorded then at least at the bottom layer of the binary decomposition tree I could modify 2 variables simultaneously if required in simulation and/or optimisation tasks.

Now I have calculated that at every branching point the structural sensitivity model is quadratic, *e.g.* by computing first and second order finite differences to obtain for left and right ("1" or "2" respectively):

$$f(\theta_1+\Delta\theta_1, \theta_2+\Delta\theta_2) = f(\theta_1,\theta_2) + \frac{\partial f}{\partial \theta_1}(\Delta\theta_1) + \frac{\partial f}{\partial \theta_2}(\Delta\theta_2) + \frac{1}{2}\frac{\partial^2 f}{\partial \theta_1^2}(\Delta\theta_1)^2 + \frac{1}{2}\frac{\partial^2 f}{\partial \theta_2^2}(\Delta\theta_2)^2 + \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2}(\Delta\theta_1)\cdot(\Delta\theta_2) + Higher\ Order\ Terms\ (H.O.T) \tag{6.9}$$

I have demonstrated that the second order, quadratic sensitivity model can be computed. Then I demonstrate how to exploit the second order information in the binary tree-based decomposition search for which branch to follow, assuming that I follow only a single branch and only in the last level of decomposition, to only change one variable at a time by consistency.

Given a quadratic model in $\delta_1 \triangleq \Delta\theta_1$,$\delta_2 \triangleq \Delta\theta_2$:

$$f(\theta_1+\delta_1, \theta_2+\delta_2) = a + b\delta_1 + c\delta_2 + d\delta_1\delta_2 + e\delta_1^2 + f\delta_2^2 \tag{6.10}$$

and a trust region:

$$\begin{cases} -\varepsilon_1 \leq \delta_1 \leq +\varepsilon_1 \\ -\varepsilon_2 \leq \delta_2 \leq +\varepsilon_2 \end{cases}$$

I can optimise the above quadratic model (even considering only sufficiently small quantized discrete-size steps: $\delta_1 = \pm\varepsilon_1$, $\delta_2 = \pm\varepsilon_2$) and assign targets to change the next sublevel values by $\delta_1^\star$, $\delta_2^\star$: $\theta_1^{new} = \theta_1^{old} + \delta_1^\star$, $\theta_2^{new} = \theta_2^{old} + \delta_2^\star$.

Whether the optimal steps $\delta_1^\star$, $\delta_2^\star$ are quantised, or continuously valued within the trust region, they require cascading down the model's evaluation tree.

Coming to a point where there's a multiply linked leaf, then deciding on a simple updating criterion will become very challenging due to this coupling.

An alternative way explored is to use the quadratic model to extract further information in computing left-and-right selection probabilities in the non-deterministic, randomised left-or-right selector at the binary tree branching points.

Given:

$$q(\delta_1, \delta_2) = a + b\delta_1 + c\delta_2 + d\delta_1\delta_2 + e\delta_1^2 + f\delta_2^2 \tag{6.11}$$

I obtain as a local approximation model for the two gradient elements:

$$\frac{\partial q}{\partial \delta_1} = b + d\delta_2 + 2e\delta_1 \tag{6.12}$$

$$\frac{\partial q}{\partial \delta_2} = c + d\delta_1 + 2f\delta_2 \tag{6.13}$$

Both expressions are independent of the trust region parameters.

To modify the selector (non-deterministic randomised element) as a first and simple way, I can base the selection on the average value of the above gradient elements within the trust region of interest.

So:

$$\left[\frac{\partial q}{\partial \delta_1}\right] = \frac{1}{4\varepsilon_1\varepsilon_2}\int_{\theta_1-\varepsilon_1}^{\theta_1+\varepsilon_1}\int_{\theta_2-\varepsilon_2}^{\theta_2+\varepsilon_2}(b + d\delta_2 + 2e\delta_1)d\delta_2 d\delta_1 \tag{6.14}$$

where $\frac{1}{4\varepsilon_1\varepsilon_2}$ comes from $\int_A \int 1 \cdot d\delta_2 d\delta_1$ and $A = \{\theta_1 - \varepsilon_1 \leq \delta_1 \leq \theta_1 + \varepsilon_1, \theta_2 - \varepsilon_2 \leq \delta_2 \leq \theta_2 + \varepsilon_2\}$.

$$\begin{aligned}\left[\frac{\partial q}{\partial \delta_1}\right] &= \frac{1}{4\varepsilon_1\varepsilon_2}\int_{\theta_1-\varepsilon_1}^{\theta_1+\varepsilon_1}[b\delta_2 + \frac{1}{2}d\delta_2^2 + 2e\delta_1\delta_2]_{\theta_2-\varepsilon_2}^{\theta_2+\varepsilon_2}d\delta_1 \\ &= \frac{1}{4\varepsilon_1\varepsilon_2}\int_{\theta_1-\varepsilon_1}^{\theta_1+\varepsilon_1}[b(\theta_2+\varepsilon_2) + \frac{1}{2}d(\theta_2+\varepsilon_2)^2 + 2e\delta_1(\theta_2+\varepsilon_2) \\ &- b(\theta_2-\varepsilon_2) - \frac{1}{2}d(\theta_2-\varepsilon_2)^2 - 2e\delta_1(\theta_2-\varepsilon_2)]d\delta_1\end{aligned} \tag{6.15}$$

$$\begin{aligned}\left[\frac{\partial q}{\partial \delta_1}\right] &= \frac{1}{4\varepsilon_1\varepsilon_2}\int_{\theta_1-\varepsilon_1}^{\theta_1+\varepsilon_1}(2b\varepsilon_2+2d\theta_2\varepsilon_2+4e\delta_1\varepsilon_2)d\delta_1 \\ &= \frac{1}{4\varepsilon_1\varepsilon_2}[(2b\varepsilon_2+2d\theta_2\varepsilon_2)\delta_1+2e\varepsilon_2\delta_1^2]_{\theta_1-\varepsilon_1}^{\theta_1+\varepsilon_1} \\ &= \frac{1}{4\varepsilon_1\varepsilon_2}[(2b\varepsilon_2+2d\theta_2\varepsilon_2)\cdot 2\varepsilon_1+2e\varepsilon_2\cdot 4\theta_1\varepsilon_1] \\ &= \frac{1}{4\varepsilon_1\varepsilon_2}[4b\varepsilon_1\varepsilon_2+4d\theta_2\varepsilon_1\varepsilon_2+8e\theta_1\varepsilon_1\varepsilon_2] \\ &= b+d\theta_2+2e\theta_1\end{aligned} \tag{6.16}$$

Thus, the end result is:

$$\left[\frac{\partial q}{\partial \delta_1}\right] = b+d\theta_2+2e\theta_1 \tag{6.17}$$

From $q(\delta_1,\delta_2) = a+b\delta_1+c\delta_2+d\delta_1\delta_2+d\delta_1^2+f\delta_2^2$, by comparison with Equation 6.17, I obtain symmetrically:

$$\left[\frac{\partial q}{\partial \delta_2}\right] = b+d\theta_1+2f\theta_2 \tag{6.18}$$

Both Equation 6.17 and Equation 6.18 are independent of the trust region parameters.

The perturbation of θ's are such that:

$$\theta_1 = \theta_2 \triangleq 1+\tau \tag{6.19}$$

where τ is an infinitesimal value.

From Equation 6.17 and Equation 6.18 I get:

$$\left[\frac{\partial q}{\partial \delta_1}\right] = b+d+2e \tag{6.20}$$

$$\left[\frac{\partial q}{\partial \delta_2}\right] = b+d+2f \tag{6.21}$$

which are independent of the trust region parameters.

These are then to be used instead of the point-wise gradient elements for θ_1 and θ_2 in order to select which branch to follow, left or right.

The randomised selector step is then defined by:

$$r_x = RandomReal([0,1]) \tag{6.22}$$

and

$$r_g = \frac{|[\frac{\partial q}{\partial \delta_1}]|}{|[\frac{\partial q}{\partial \delta_1}]|+|[\frac{\partial q}{\partial \delta_2}]|} \tag{6.23}$$

If $r_x \leq r_g$, I select the left branch (θ_1). If $r_x \geq r_g$, I select the right branch (θ_2).

The previous results on the average value of the gradients using a quadratic model require the calculation of the second and first order derivatives by finite difference.

I can compute the first order derivatives by:

$$\frac{\partial f(\theta_1,\theta_2)}{\partial \theta_1} \approx \frac{f(\theta_1+h_1,\theta_2)-f(\theta_1,\theta_2)}{h_1} + O(h_1) \tag{6.24}$$

or:

$$\frac{\partial f(\theta_1,\theta_2)}{\partial \theta_1} \approx \frac{f(\theta_1+h_1,\theta_2)-f(\theta_1-h_1,\theta_2)}{2h_1} + O(h_1^2) \tag{6.25}$$

and the second order derivative:

$$\begin{aligned} \frac{\partial^2 f}{\partial \theta_1^2} &\approx \frac{\frac{\partial f(\theta_1+h)}{\partial \theta_1} - \frac{f(\theta_1-h)}{\partial \theta_1}}{2h} \\ &\approx \frac{\frac{f(\theta_1+h,\theta_2)-f(\theta_1,\theta_2)}{h} - \frac{f(\theta_1,\theta_2)-f(\theta_1-h1,\theta_2)}{h}}{2h} \\ &= \frac{1}{2h^2}[f(\theta_1+h,\theta_2)-2f(\theta_1,\theta_2)+f(\theta_1-h_1,\theta_2)] \end{aligned} \tag{6.26}$$

In the same way, I get symmetrically the values of $\frac{\partial f}{\partial \theta_2}$ and $\frac{\partial^2 f}{\partial \theta_2^2}$.

The second-order mixed derivatives are:

$$\begin{aligned} \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} &\approx \frac{\frac{\partial f(\theta_1,\theta_2+h_2)}{\partial \theta_1} - \frac{f(\theta_1,\theta_2-h_2)}{\partial \theta_1}}{2h_2} \\ &\approx \frac{\frac{f(\theta_1+h_1,\theta_2+h_2)-f(\theta_1-h_1,\theta_2+h_2)}{2h_1} - \frac{f(\theta_1+h_1,\theta_2-h_2)-f(\theta_1-h_1 1,\theta_2-h_2)}{2h_1}}{2h_2} \end{aligned} \tag{6.27}$$

In summary,

$$\begin{aligned} \frac{\partial^2 f(\theta_1,\theta_2)}{\partial \theta_1 \partial \theta_2} \approx \frac{1}{4h_1h_2}[&f(\theta_1+h_1,\theta_2+h_2)-f(\theta_1-h_1,\theta_2+h_2) \\ &-f(\theta_1+h_1,\theta_2-h_2)+f(\theta_1-h_1,\theta_2-h_2)] \end{aligned} \tag{6.28}$$

For finite difference calculations use the heuristic:

$$\Delta_x = 100 \times \sqrt{MachinePrecision} \times max\{|x|, 1.0\} \tag{6.29}$$

6.5.2 Results and Analysis

I perform an example run of the optimisation adopting the second-order information as described above to select left or right at the branching point. I similarly adopt the architecture of [4,5,5,5,5,5,5,5,5,2] as an example to investigate the effect of incorporating second-order information. The number of data points used is 10,000. The same set of hyperparameters are adopted as enlisted in Table 6.2.

Figure 6.7 demonstrates the sensitivity values and the objective function values over the course of optimisation, at iteration 1, 10, 20, 30, 40 and 50. The optimisation reaches the tolerance value at exactly 50 iterations.

From Figure 6.7, it can be observed that the objective value decreases over time with convergence to a low value within around 20 iterations. Moreover, I observe the quick equilibration of sensitivity values with this tuning scheme. Within 10 iterations, the value of sensitivities become almost equal across layers, and the equilibrium stays over further iterations.

I also plot the distribution of layers selected in the optimisation process in Figure 6.8. It can be observed that the first and last layer are more frequently updated but with the random factor, other layers have a possibility of being selected.

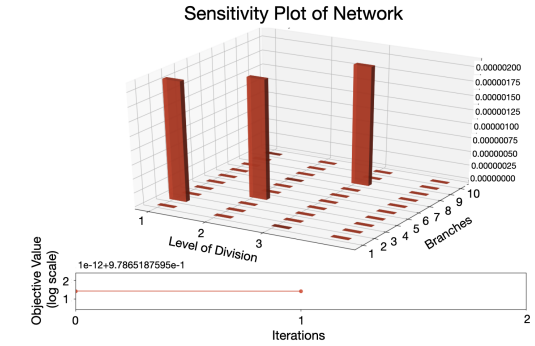
Compared to the optimisation using first-order information, the new optimisation scheme observes the same effect of equilibration and fast convergence but the values of the sensitivities are higher due to the introduction of ε^2 on the denominator. The other difference is that it converges faster within 50 rounds of iterations, indicating a more direct search direction brought about by the second-order information.

6.5.3 Comparison with End-to-end Training

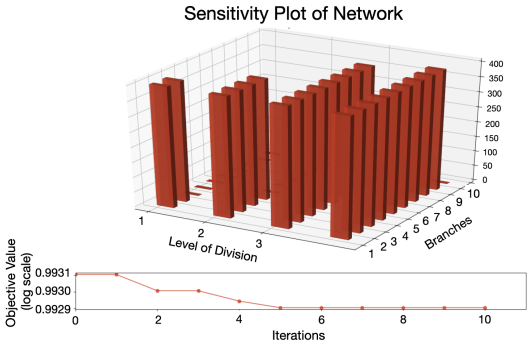
Hyperparameters. The hyperparameters adopted in both end-to-end training and multi-scale hierarchical training are enlisted in Table 6.2. It is the same set of hyperparameters that are used in the case of optimisation using first-order information.

Objective. The objective value obtained form end-to-end optimisation is 0.99283426. The objective value obtained from multi-scale hierarchical search method is 0.99283424. The values of the objectives are comparable, with the multi-scale hierarchical optimisation arriving at a lower objective value.

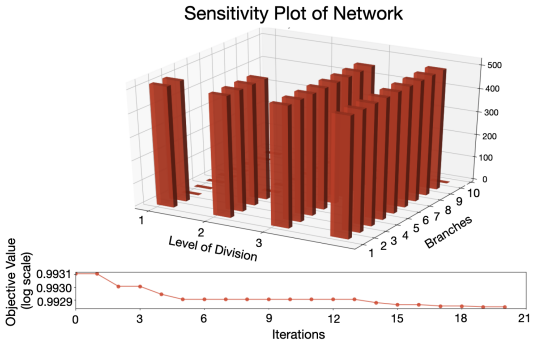
Optimisation Time. The CPU time taken for the end-to-end optimisation is 358.165 seconds. In comparison, the CPU time taken for the multi-scale hierarchical optimisation method is 762.714 seconds. If I compare the number of iterations, the end-to-end optimisation takes 99 iterations to reach the convergence criterion whereas the multi-scale hierarchical



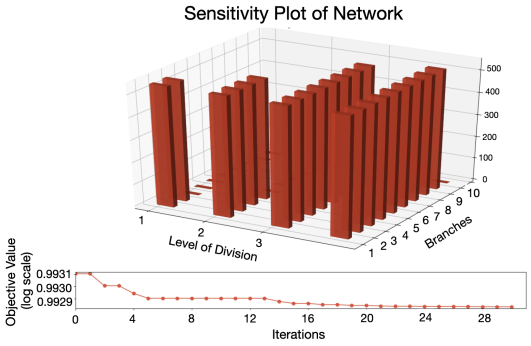
(a) First optimisation



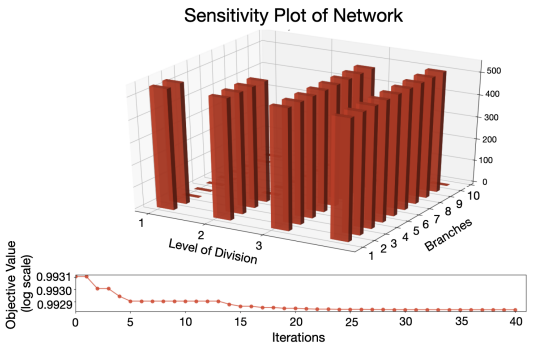
(b) Second optimisation



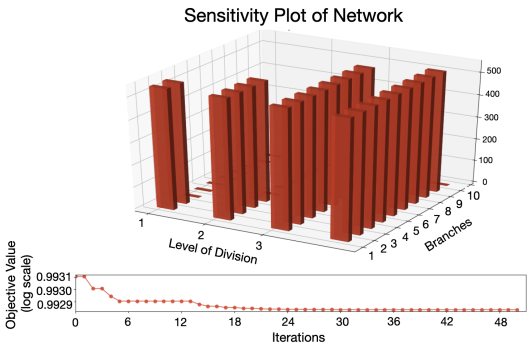
(c) Third optimisation



(d) Fourth optimisation



(e) Fifth optimisation



(f) Sixth optimisation

Figure 6.7 The process of optimisation in the example formulation adopting second-order information with neural architecture [4,5,5,5,5,5,5,5,5,2] using 10,000 data points. The 6 figures represent the optimisation results at iteration 1, 10, 20, 30, 40 and 50 respectively.

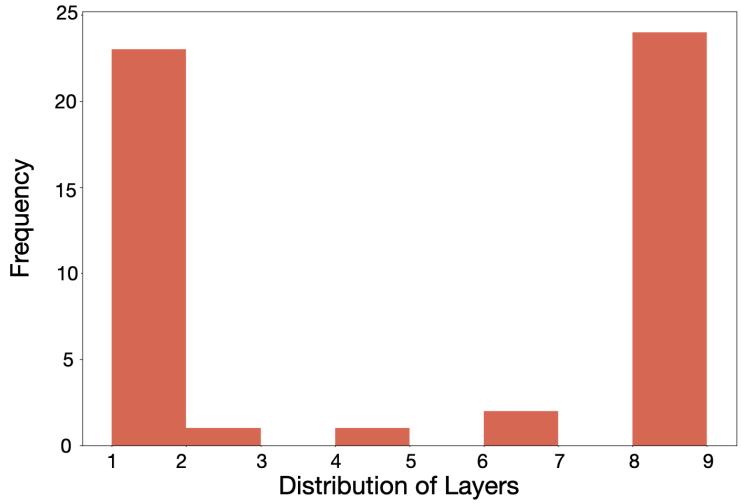


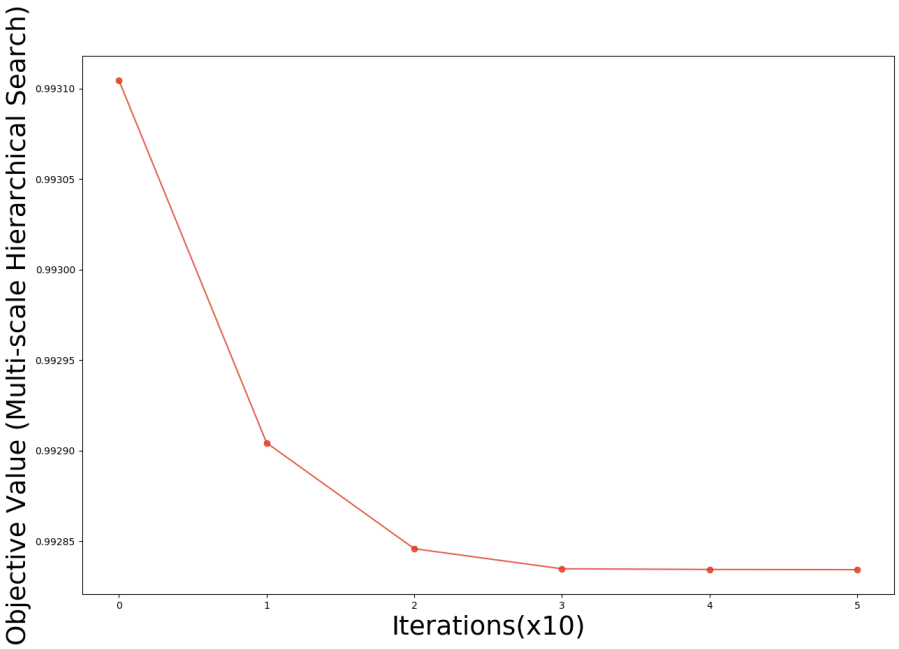
Figure 6.8 The distribution of layers selected in the optimisation process adopting second-order information.

optimisation takes 50 iterations. The plot of the optimisation process is demonstrated in Figure 6.9. Separate graphs are drawn since the number of iterations are different and are not directly comparable. One iteration in end-to-end backpropagation optimisation corresponds to one update of the weights across all layers whereas one iteration in multi-scale hierarchical search corresponds to one update of a single layer obtained from the binary tree search.

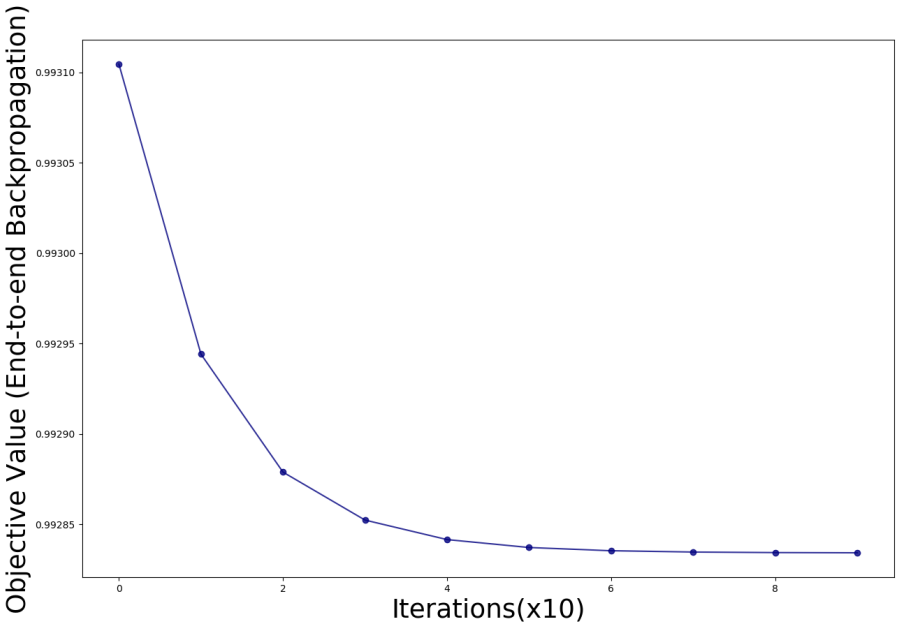
Weight Values. Similarly, I check the weight values to observe whether the two algorithms arrive at the same local minimum. The distribution of the weights are presented in Figure 6.10. From Figure 6.10, it can be observed that the weights are roughly similar and the two algorithm are highly likely to optimise to the same local minium, corroborated by the similarity in the values of the objective function obtained.

6.6 Comparison between First-order Search and Second-order Search

I compare the performance of the binary tree search adopting first-order information against the second-order information. In the analysis above, I adopt the same set of hyperparamters to evaluate the search criterion using first- and second-order information. Thus, the performances are directly comparable. Overall, the optimisation adopting second-order information is faster, with a total CPU time of 762.714 seconds and a total number of iterations of 50. This draws comparison to the optimisation adopting first-order information, with a total CPU time of 2773.410 seconds and a total number of iterations of 200.



(a) Multi-scale Hierarchical Search



(b) End-to-end Backpropagation

Figure 6.9 The comparison between the optimisation performance of the end-to-end back-propagation method and the multi-scale hierarchical search method.

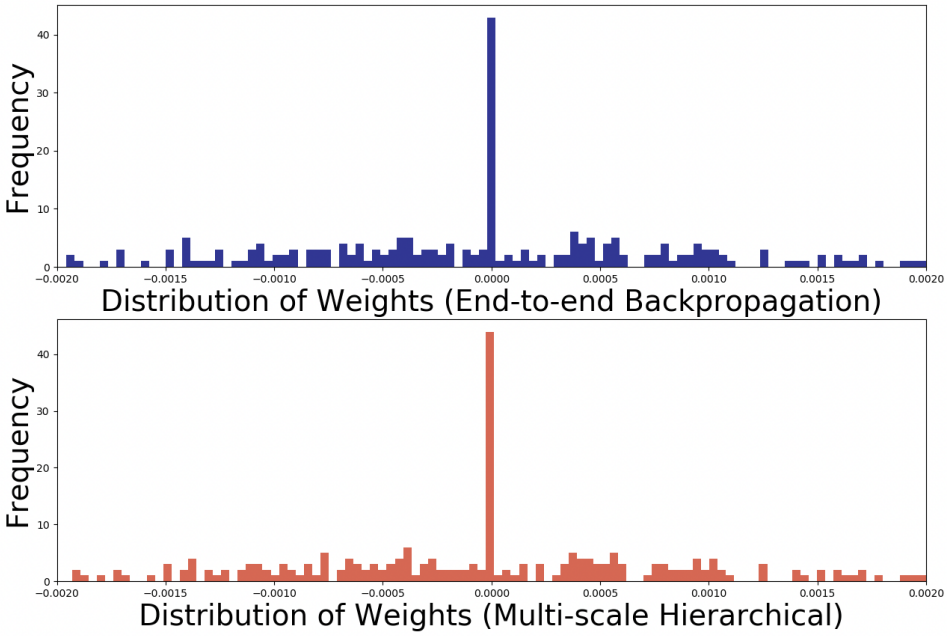


Figure 6.10 The distribution of the weights obtained from end-to-end backpropagation and multi-scale hierarchical search algorithm with second-order information.

In the analysis above, the number of epochs in each iteration is fixed to be 1,000, which means that the number of epochs to optimise each layer is 1,000. To further compare how the incorporation of second-order information improves the search of optimality by reducing the operation time, I insert a convergence criterion in each iteration of optimisation. The convergence criterion for each iteration is defined as follows:

$$\left|\left|\frac{\partial LSQR}{\partial W_{layer}}\right|\right|_{inf} < tolerance \tag{6.30}$$

where W_{layer} refer to the weights in the layer being optimised. The tolerance is a user-defined input value which can be equal to the convergence criterion of the whole optimisation problem. By comparing the infinity norm of the weights in the layer tuned to a tolerance value, I stop further optimisation of the layer if the tolerance is subceeded. Thus, the number of epochs in each iteration will be less than 1,000.

In comparison, the convergence criterion for the whole optimisation problem is:

$$\left|\left|\frac{\partial LSQR}{\partial W_{network}}\right|\right|_{inf} < tolerance \tag{6.31}$$

Table 6.4 The comparison between binary tree search based on first-order and second-order information, against backpropagation. The average values are obtained for each iteration. The total values are obtained for all iterations.

Order	Average CPU Time	Average Epochs	Total CPU Time	Total Iterations
First	5.746	405.7	350.4	51
Second	3.013	203.4	310.9	101
Backpropagation	3.022	200	297.3	99

where $W_{network}$ refer to the all the weights in the network. Practical implementation shows that a better tolerance for each iteration is 10^{-4} and for the overall optimisation problem is 10^{-3}.

To demonstrate how the incorporation of second-order information improves the speed of optimisation, I record the average value of the number of epochs, the CPU time and the infinity norm of the gradient values in each iteration. The comparison of first-order sensitivity selection and second-order sensitivity selection is shown in Table 6.4.

After adding a convergence criterion in each iteration, I observe that the adoption of second-order sensitivity as the selection benchmark takes less operation time than first-order sensitivity. The average CPU time and the average number of epochs in each iteration is smaller when second-order information is adopted. Although the total number of iterations is higher for second-order information, this is compensated with less search time within each iteration.

Comparing with backpropagation, the second-order process takes comparable and even less CPU time per iteration. The total CPU time is also similar with backpropagation. Backpropagation is the fastest algorithm overall with the second least total number of iterations. Although overall, backpropagation outperforms both first and second order processes in CPU time, the first-order multi-scale hierarchical search takes less iterations and the second-order search is faster per iteration.

I also plot the convergence of $\left|\left|\frac{\partial LSQR}{\partial W_{network}}\right|\right|_{inf}$ across iterations, which is demonstrated in Figure 6.11a. It can be observed that both optimisations converge almost roughly at the same time (around 50 iterations). The only reason that the second-order process takes longer is due to the fact that the tolerance has not been reached in exact numerical values.

In Figure 6.11b, I draw the scatter plot of the number of epochs in each iteration against the number of iterations. It can be observed that the values are almost separated into two categories, either reaching the maximum number of epochs allowed in each iteration (1,000), or using only one epoch to reach the tolerance, with the exception of only two points. One

Table 6.5 The comparison between binary tree search based on first-order and second-order information, against backpropagation, for a network with 20 layers. The average values are obtained for each iteration. The total values are obtained for all iterations.

Order	Average CPU Time	Average Epochs	Total CPU Time	Total Iterations
First	18.681	397.26	1022.4	50
Second	18.402	397.26	1153.5	50
Backpropagation	9.376	200	912.7	97

explanation is that either the tolerance is difficult to reach or the layer is already optimised so I have a polarised distribution of number of epochs.

Comparing with Figure 6.11c, the distribution of CPU time is similar to Figure 6.11b, indicating that the CPU time is closely related to the number of epochs in each iteration, with fluctuations due to the speed of performing different calculations.

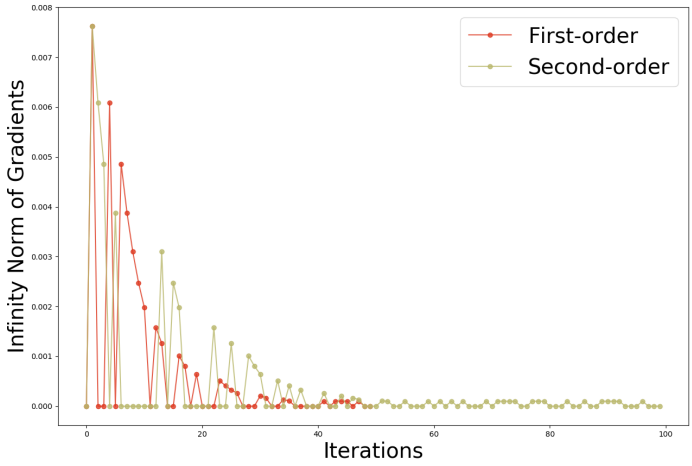
Overall, the second-order information performs better than first-order in terms of total performance as well as performance per iteration. It also has the potential to converge in fewer iterations than first-order processes with a different definition of tolerance.

6.7 Application to Large-scale Problems

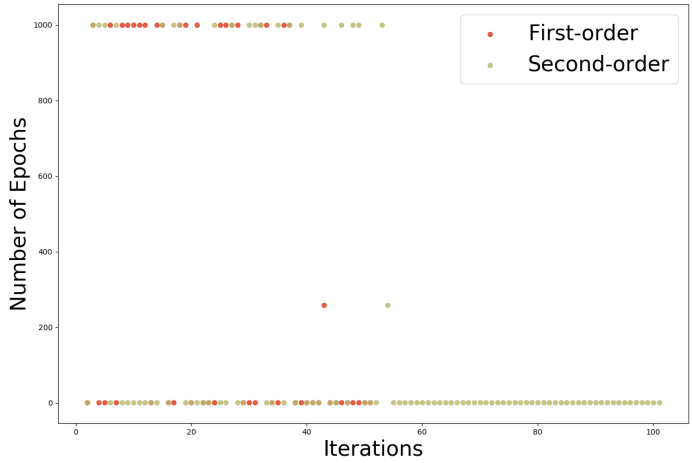
The advantage of the multi-scale hierarchical method is not obvious when the network scale is small, as demonstrated in Section 6.4.3. Therefore, I implement a much larger network and observe its effects.

I first implement the algorithm to a network with 20 hidden layers of 5 neurons each. The performance of implementing first-order process, second-order process and backpropagation is demonstrated in Table 6.5. From the results, it can be observed that the backpropagation is still the fastest algorithm in terms of total CPU time and CPU time per iteration. The second-order method is slightly faster than first-order in terms of average CPU time per iteration. The number of iterations is the same for first- and second-order method in this case, both are higher than the backpropagation method. This demonstrates the effectiveness of the sensitivity-based selection method in optimising the network to a required tolerance.

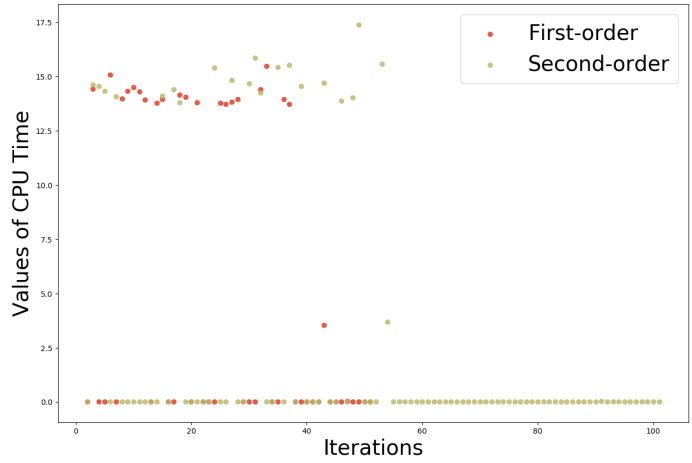
Figure 6.12 demonstrates the performance of the first-order method against the second-order method. From Figure 6.12a, I observe that the rate of convergence is roughly similar for first-order and second-order method. Compared to the smaller network, I similarly observe that the training time and epochs is separated into two categories, either costing the maximum epochs to optimise a particular layer, or immediately reaching convergence criterion with



(a) Gradient Across Iterations



(b) Epochs Across Iterations



(c) CPU Time Across Iterations

Figure 6.11 Comparison of the convergence rate of optimisation using first-order information vs second-order information.

Table 6.6 The comparison between binary tree search based on first-order and second-order information, against backpropagation, for a network with 50 layers. The average values are obtained for each iteration. The total values are obtained for all iterations.

Order	Average CPU Time	Average Epochs	Total CPU Time	Total Iterations
First	24.519	208.83	3271.9	100
Second	48.912	416.67	3746.6	50
Backpropagation	25.206	200	2529.4	100

1 epoch. This is demonstrated in the polarisation of data points in Figure 6.12b and Figure 6.12c.

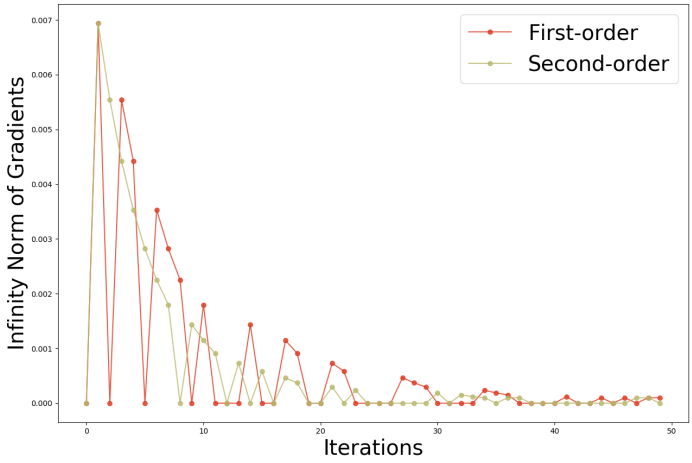
I then increase the number of layers to 50. The results are demonstrated in Table 6.6 and Figure 6.13.

From Table 6.6, I observe that the first-order method is the fastest in terms of average CPU time, followed by the backpropagation method. However, the second-order method has a key advantage with the lower number of iterations. Observing Figure 6.13, I find that the second-order method initially converges more slowly than the first-order method, later reaching similar speed of convergence after several iterations. As the speed of convergence differs from that of a 20-layer model, it can be inferred that there is no fixed dominance of the first-order over the second-order method and vice versa. I similarly conclude that there is a polarisation in terms of data points, indicating either immediate convergence or very slow convergence of a particular layer. I postulate that the slow convergence occurs on layers that contribute significant changes to the overall model performance, indicating the unequal importance of different layers inside the model.

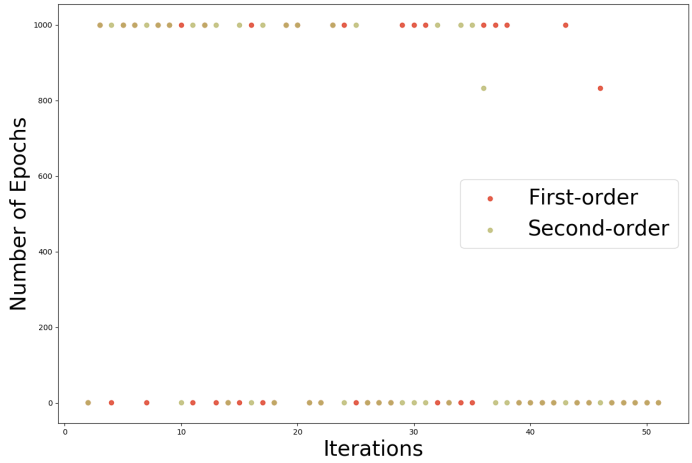
6.8 Discussion

6.8.1 Structural Sensitivity Equilibration

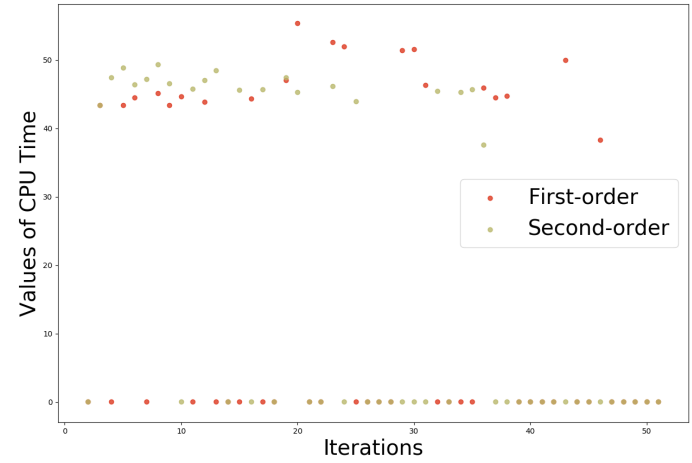
An important aspect of the new multi-scale hierarchical optimisation algorithm is that it is capable of equilibriating the structural sensitivities and converges to at least a local minimiser. Secondly, by the randomised left-or-right non-deterministic selector criterion at the binary multi-scale partitioning tree, and its statistically designed element on favouring the side with the largest locally determined absolute structural sensitivity (derivative) value, the iterations are naturally expected to achieve a path that not only progresses to a local minimum, but also exhibits equilibration of left and right structural sensitivity values at the branching points of the binary partitioning tree.



(a) Infinity Norm of Gradient Across Iterations

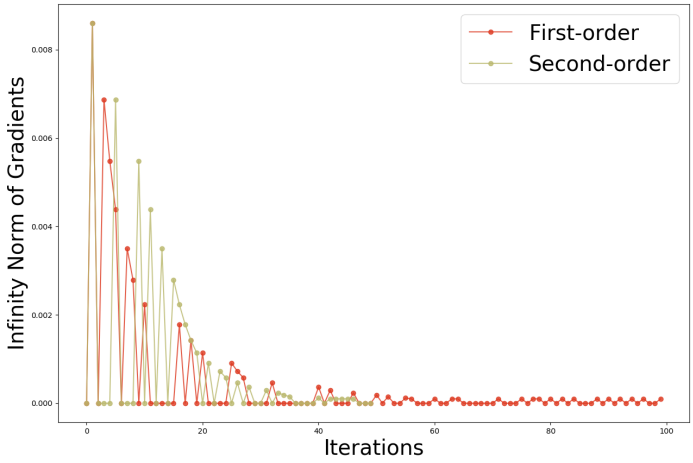


(b) Epochs Across Iterations

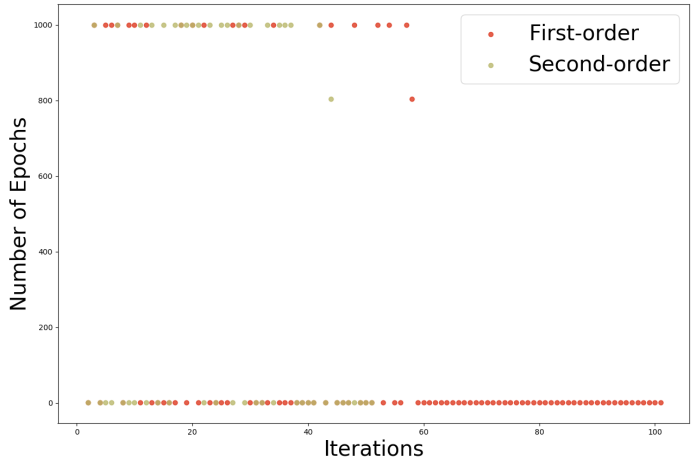


(c) CPU Time Across Iterations

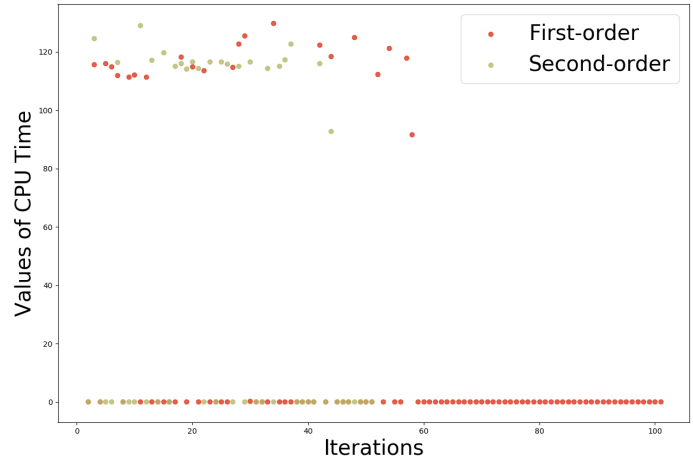
Figure 6.12 Comparison of the convergence rate of optimisation using first-order information vs second-order information in a network with 20 layers.



(a) Infinity Norm of Gradient Across Iterations



(b) Epochs Across Iterations



(c) CPU Time Across Iterations

Figure 6.13 Comparison of the convergence rate of optimisation using first-order information vs second-order information in a network with 50 layers.

This poses an interesting first question (a) as to the meaning and potential physical interpretation, further form their mathematical definition, with regards to implications for the underlying mathematical model, and hence (b) physical interpretations of the physical system being modeled within the novel multi-scale hierarchical modeling and solution computation framework proposed.

6.8.2 Online Application

Compartments of the arbitrary partitioning generates equilibrated values in terms of importance/impact on the overall defined performance index criterion of the entire system being modeled, or even controlled online in real-time by the novel optimization framework introduced here. This is a very interesting property which allows further novel considerations: if a system is thus structural sensitivity-wise equilibrated, any small deviation will be easily detectable at the top-level of the binary tree and easily identifiable at the finest structure of the embedded model and underlying physical system in log_2 steps of the total number of the finest structure components of the system.

6.8.3 Degree of Coupling Implications with Second-order Sensitivities

In view of Section 6.5, there is now a new metric of analysing, evaluating, designing and eventually controlling fully, very large-to-enormous scale systems, both in the mathematical and in the more abstract sense. Effectively, a huge-impact topic highlighted at the end of Section 6.5 can be enhanced by:

1. First-order structural sensitivities revealed a "linear behaviour" approximation of our system from a hierarchical multi-scale point of view.

2. The second-order structural sensitivities measure quantitatively the degree of local non-linearity of our system considering its interacting parts form the top level of abstraction, all the way down to its finer/finest modeling scale.

3. The mixed second-order sensitivities reveal the degree of coupling of the underlying level sub-compartments of our model, as I descend down the multi-scale hierarchical structure/framework levels.

Such a numerically quantitative system metric will:

1. Allow the exploration of new, self-adaptive local information, simulation and system optimisation algorithms within the currently proposed framework

2. Allow a self-adaptive parallelisation of nested computations such that loosely interacting (or loosely connected) components can be mathematically reliably assigned to different parallel processors, thus paving the way both for new parallel algorithms with existing hardware, as well as the design of new dedicated computational architectures that can be exploited more fully by the novel proposed computational modeling and solution framework.

6.8.4 Training and Testing

It is noteworthy that performance results in this chapter are mostly based on the training process. The testing results are not included since the focus is on how fast the model converges and whether the same results are arrived compared to end-to-end models. As future work, it is possible to add generalisation as a performance evaluation process and demonstrate that there is no overfitting from our proposed multiscale hierarchical search model.

6.9 Summary

In this chapter, a novel multi-scale hierarchical search algorithm is developed to achieve the tuning of artificial neural networks (ANNs). The algorithm takes a leveled approach where within each level, the network is divided into left and right side, and the sensitivity of each side is evaluated through first-order or second-order finite difference method. The algorithm then successively selects one side to further divide until only one layer is left, obtaining overall a binary tree search. The selected layer is optimised in one iteration until a convergence criterion for one iteration is reached. Then the selection process repeats until an overall tolerance is reached, and until the tolerance is reached, the tuning of the network terminates.

A key feature of the algorithm is the introduction of a randomisation process in the binary selector where a sensitivity-based probability value is calculated and used as the probability to select left or right. The introduction of the random factor improves performance as it ensures less repetition in the optimisation process of a single layer, thus achieving more efficient optimisation.

The key novelty of the algorithm is the binary search process implemented to only optimise one layer at a time. In theory, it is applicable to a large network where only a subsets of layers of key importance are optimised. If the requirement on tolerance is not high, the method can be in theory faster than the optimisation of all layers.

Overall, a novel hierarchical multi-scale search method is proposed for the tuning of deep neural networks. The introduction of selective tuning allows fast convergence of DNNs and quick equilibration of sensitivity values. As sensitivity values of the attenuation/amplification factor also corresponds to neuron importance, ongoing research efforts can focus on how to manipulate this method to be applied to the structural evolution of DNNs.

Chapter 7

Dynamic Neural Architecture Construction

7.1 Introduction

In Section 2.2.7, I have reviewed literature that searched for the architecture of a neural network through the process of pruning, *i.e.* removing some branches in the network while achieving comparable or even better performance. It is in the mainstream literature that such a method is used for the construction of a neural network with reduced parameters storage. However, another school of literature worked counter-wise to design a neural network that is built constructively by extending from a small network. Therefore, I developed the intuition that a neural network can be reversely produced from some basic structure, and I term the process "Dynamic Neural Architecture Construction".

There are some weaknesses of pruning a network in comparison to constructing a dynamic one. In pruning, training time has been spent on optimising a network larger than necessary, generating wasteful computations [226]. Moreover, the optimal architecture found is often stuck in one of the intermediately sized solutions, and the smallest network is not found [227].

The dynamic construction of a neural network often involves a series of optimisation processes, as each time the architecture is transformed, an optimisation with regard to the output error is performed. Within literature, the objective function of the optimisation process in a dynamic architecture often includes common metrics such as training or testing losses. However, different architectures entail different optimisation processes and different variations of the objective functions.

There are several advantages of a dynamic architecture. First, it allows lifelong learning where the responses from a network evolve with new feed of input data [228]. Traditional neural networks adapt to changing inputs by varying the parameters of the network, but the overall architecture stays unchanged. With a dynamic structure, it is possible to alter architectural hyperparameters in order to respond to a learning question. Second, while some have the intuition that dynamic networks trained with local adaptation methods may not work so well as a complete neural network trained end-to-end on the whole training dataset, this can be a countered by empirical evidence [229]. Both methods are found equally effective as the advantage of eliminating unnecessary parameters outweighs the disadvantage of local incremental learning [229].

The controversy surrounding the dynamic methods is that it can be quite time-consuming to find an architecture and the process requires a lot of computation. This is accentuated in a large and deep network such as the ResNet [230] where there are 152 layers in total and the addition layer by layer can be a slow process. Each time the architectural parameters are altered, an optimisation algorithm is run to ensure that the structure is optimal. However, this process is capable of producing a minimally structured network, therefore reducing the needs of performing unnecessary computations in the testing and application stages. Moreover, the problems in many industrial settings have a small dataset with a less complicated nonlinear relationship to model, which allows a dynamic method to be economical to use compared to problems in computer vision [231], natural language processing [232], speech recognition [233], *etc*.

This chapter discusses the application and modification of one dynamic algorithm, DAN2, with a focus on the chemical engineering datasets. The architecture incrementally adds layers to the initial structure but with a fixed number of neurons in each layer. The architecture has been demonstrated to perform well in predicting non-linear processes [234] and time-series [235]. Moreover, it has an edge in the prediction of dimensionally small datasets, typical of what most chemical engineering datasets are. The original model is created for single-output predictions only. I introduce the model to a few chemical datasets and adapt the model to multi-task learning to allow multi-variate output. Different structures of multi-task models are proposed and the performance is compared. I also instigate a heuristic search scheme to find the optimal architecture of the multi-task learning network. The proposed architecture search method is well-suited to the multi-task learning network and allows the introduction of multi-task learning processes to DAN2 network.

I motivate a search of the optimal architecture of the multi-task version of the DAN2 network since industrial datasets from the chemical process industries usually contain multiple highly-correlated outputs relating to the same operating conditions [236] [237]. Since

the correlation between the outputs are high, there is huge potential in applying multi-task learning in chemical engineering. A design method that identifies the optimal architecture of this multi-task learning network is thus beneficial to the research community.

Moreover, the adoption of DAN2 serves as a novel alternative to the current ANNs or mathematical models commonly used in the industries to simulate processes. Mathematical models require heavy computations such as in the case of fluid dynamics. DAN2 and ANNs are data-based models that make predictions with less complicated modeling techniques. In comparison with ANNs, DAN2 tends to have less parameters and is more suited to time series modeling.

There are two optimisation processes involved in optimising DAN2 network: 1) the optimisation of the total number of layers with respect to the test set performance evaluated by metrics such as Mean Squared Error, and 2) the optimisation of the value of coefficient μ with respect to the function approximated by one layer of the network. A more comprehensive discussion can be found in Section 7.6. I focus on the first optimisation process as this is the neural architecture search step. By optimising the total number of layers for each task and for layers shared by each task, I can successfully construct a multi-task learning DAN2 network.

This chapter is outlined as follows:

- Section 7.2 describes past and current literature adopting a dynamic architectural search method to construct neural networks.
- Section 7.3 describes the model of DAN2 and the optimisation processes involved. It also performs a primitive analysis of the architecture.
- Section 7.4 outlines the methodology involved in data processing and model construction.
- Section 7.5 gives an overview of the multi-task learning and the advantages of applying DAN2 to multi-task learning problems.
- Section 7.6 defines key mathematical concepts in optimisations of multi-task learning problems.
- Section 7.5 introduces the optimisation of a multi-head structure. Both the methods of grid search and the heuristic search are proposed and compared.
- Section 7.8 introduces the serial structure adopting similar optimisation schemes. The performance is compared.
- Section 7.9 discusses alternative optimisation methods and their pros and cons. In addition, possible future works are investigated.

- Section 7.10 then concludes the chapter.

7.2 Review on Dynamic Architectures in Literature

The idea of dynamically extending the network is inspired from the classic Adaptive Resonance Theory (ART) [238] and Grow and Learn (GAL) Theory for networks [154]. Similarly closely related is the concept of "Lifelong Learning", where the network is allowed to evolve incrementally with new inputs such that the response function changes over time [228], and this process often involves dynamically changing the architecture. This concept is sometimes also termed "Continual learning", which refers to the ability of the system to learn from a continuous stream of input information without catastrophic forgetting [239]. "Incremental Learning", which refers to effective model adaptation with a continuous feed of streaming data, is often frequent in research of dynamic architectures.

7.2.1 Primitive Dynamic Methods

The primitive dynamic architecture construction consists of three mainstream methods. They are mainly related to evolving the basic structure of an ANN and are some of the earliest work to address the dynamic architectural construction problem:

- Adding neurons to a fixed architecture [35] [227] [240] [241]
- Adding layers to a fixed architecture [155]
- Altering connections between neurons [33]

Research in the first category has been focusing on finding the minimal topology of a neural network by adding neurons one by one. One of the earliest work in this area is [227], recognising the trade-off between a large overfitting network and a small inaccurate model. It introduces a Dynamic Node Creation (DNC) method that adds a node to the hidden layer and perform backpropagation to the network until the flattening of the average squared error curve has been detected.

In [240], the authors propose the Growing Neural Gas model, a popular method well-referenced in many literature. However, it adds neurons to the network only at fixed, pre-defined intervals.

In [241], two neural networks are generated and neurons are added to the hidden layers of each network. It works under the knowledge that as the network grows and starts to overfit, the hidden layers will diverge. Thus, when divergence happens, the two networks are amalgamated with each other.

A method to add neurons to the position that generates the maximum reduction in error is proposed in [35]. Since it only creates links when absolutely necessary, the method often produces a sparse and non-uniform network.

The design of a framework that can be used to grow neural networks based on the most primitive changes such as adding neurons or adding skip connections is proposed in [33]. This is summarised below:

- Reduce the error by expanding the capabilities of the network: (a) Change the number of neurons in a certain layer (b) Add sideway growth
- Optimise the network for a particular class of problem: (a) Add asymmetry (b) Add sideway connections (c) Skip layers
- If the fitness of network is not increasing, attempt strategies completely changing the performance: (a) Add new layer (b) Add feedback loops
- Improve the efficiency of the network: (a) Add bias (b) Prune connections

7.2.2 Modern Dynamic Methods

The Grow When Required (GWR) network [34] is also a popular model. While most growing networks add new neurons to support the node that has produced the highest error or to support topological structure, this method adds a new node when the input cannot be sufficiently modeled by the network. Moreover, previous methods add a neuron after a number of iterations to identify the necessity of neuron addition, and optimise a few times until next neuron is added, this method adds a neuron whenever the input distribution is changing. The network has been used in protein folding classification [242].

An alternative terminology is neural network self-organisation, where the network evolves to fit to non-stationary inputs by adopting a dynamic architecture. More recent literature that focuses on neural network self-organisation dynamically allocate or remove neurons in response to sensory experience with applications in human-robot interaction (HRI) [228] [239]. In [228], a self-organising recurrent model is constructed to process human actions in video sequences, and it has achieved state-of-the-art performance. In [239], a self-organising convolutional neural network is proposed to recognise body and emotional expressions. In both cases, the models are robust to non-stationary input with the self-evolving nature of the network. The authors in [243] apply self-organising neural networks to novelty detection, the process that robots recognise unexpected data in their sensory field, with applications in surveillance, reconnaissance, self-monitoring, *etc*. The model used is based on Grow

When Required Neural Network (GWRNN). In [244], a novel model is produced to analyse stationary and non-stationary noisy streaming data.

The popularity of self-organising neural networks in applications of robotics is that the input is usually a continuous stream with non-stationary distribution. Chemical processes, similar to robotics, contain streaming data of temperature, pressure, composition, *etc*. Therefore, it is natural to find applications of a self-organising network, or a dynamic architecture neural network, in the field of chemical engineering.

The following sections review in detail one of the dynamic architectural construction methods, the Dynamic Architecture for Artificial Neural Networks (DAN2) method. The method adds to the input layer incrementally layer by layer and defines the non-linear activation through trigonometric functions. It defines a novel architecture compared to traditional ANNs. It also serves as a typical example of methods often adopted in dynamic architecture construction.

7.3 Dynamic Architecture for Artificial Neural Networks (DAN2)

A dynamic feedforward architecture, DAN2, is proposed in [234], which is characteristic of common dynamic models where the number of layers increase incrementally and a better definition of each layer is encapsulated in its design [234]. The objective of the network is to model nonlinear processes or time series data. The model achieves this objective by learning and accumulating knowledge at each layer, propagating and adjusting this knowledge forward to the next layer, repeating until the desired network performance criteria are reached [234].

The advantage of DAN2 architecture is three-fold: 1) the model trains on the entire in-sample dataset collectively, allowing more effective capture of data patterns; 2) the model has high scalability, allowing the addition of layers by continuously and dynamically updating only five parameters; 3) the model removes the "black-box" notion that has been associated with neural network models by using a closed form set of equations that are obtained at the end of each step of the training process. With such advantages, it is imperative that I can expand the potential areas of applications through modifications of the network, such as making it multi-task.

In the DAN2 model, the input layer is defined by external data with normalisation. The hidden layers are sequentially and dynamically generated until the stopping criteria are met. The novelty of the algorithm is that there is a fixed number of hidden neurons in each hidden layer. The architecture is shown in Figure 7.1.

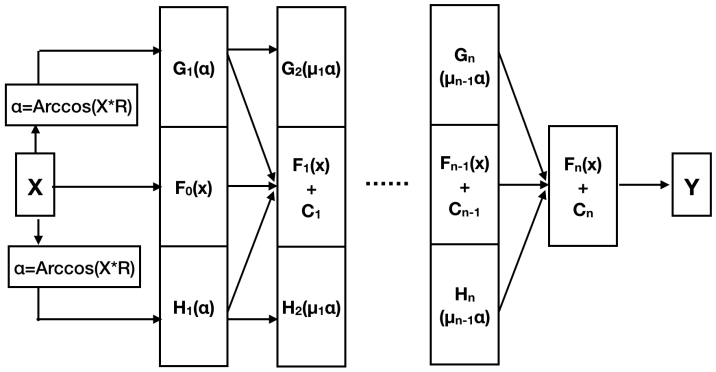


Figure 7.1 DAN2 network architecture

In each layer there are four hidden nodes. The constant (C) node adds a constant to the corresponding node. The "current accumulated knowledge element (CAKE, labelled as F)" node is a linear transformation from all nodes in previous layers. There rest nodes are "current residual nonlinear element (CURNOLE, labelled as G and H)" nodes, which perform a nonlinear transformation from the weighted average of previous CURNOLE nodes. a_k, b_k, c_k, d_k and μ_k are the weights of each nodes inputting to the next layer.

The training process involves the addition of the network layer by layer. In the first layer, the CAKE node is calculated as a linear combination of input variables and the constant (C) node. The weights of the linear combination are obtained through linear regression. If accuracy criteria are met at this step, the process to be modeled is effectively linear. If not, the subsequent CAKE nodes take input from previous CAKE, CURNOLE and C nodes.

The nonlinear transformation brought about by CURNOLE nodes adopts a vector project method. A reference vector is defined and the input vectors are projected onto this vector. The angles, α_i, between each input samples and the reference vector are recorded. The trigonometric function $Cosine(\mu_k\alpha_i + \theta_k)$ captures the nonlinear transformation equivalent to a rotation of μ_k and a shift of θ_k. $Cosine(\mu_k\alpha_i + \theta_k)$ is equivalent to $AConsine(\mu_k\alpha_i) + BSine(\mu_k\alpha_i)$. Therefore, two CURNOLE nodes are set up giving a nonlinear transformation of *Sine* and *Cosine* each.

The overall relationship is outlined as:

$$F_k(X_i) = a_k + b_k F_{k-1}(X_i) + c_k G_k(X_i) + d_k H_k(X_i) \tag{7.1}$$

where X_i represents the n independent input records, $F_k(X_i)$ represents the output value at layer k, $G_k(X_i) = Cosine(\mu_k\alpha_i)$, and $H_k(X_i) = -Sine(\mu_k\alpha_i)$ represent the transferred nonlinear components, and a_k, b_k, c_k, d_k and μ_k are parameter values at iteration k. The objective is

to minimize the total error, $SSE_k = \sum_i [F_k(X_i) - \hat{F}(X_i)]^2$, where $\hat{F}(X_i)$ is the observed output value. Substituting $F_k(X_i)$, I have:

$$SSE_k = \sum_i [a_k + b_k F_{k-1}(X_i) + c_k Cos(\mu_k \alpha_i) + d_k Sin(\mu_k \alpha_i) - \hat{F}(X_i)]^2 \tag{7.2}$$

I optimise this equation with respect to parameters a_k, b_k, c_k, d_k and μ_k. The process of layer addition and parameters optimisation alternates until a satisfying stopping condition evaluated by *SSE* is obtained.

7.3.1 Applications of DAN2 in Literature

The dynamic architecture neural network (DAN2) is proposed as a novel alternative structure to traditional feedforward backpropagation (FFBP) algorithm [234]. Research has demonstrated that the network is effective in predicting nonlinear processes [245] and in predicting time-series data [246]. From then on, mainstream research has been focusing on the prediction of time-series based on this model [156] [247]. A variety of cross-disciplinary problems have been treated based on this model, including automated text classification [248], movie revenue prediction [249], twitter sentiment analysis [250], urban water demand forecasting [251], medium term electrical load forecasting [252], stock market index and stock price prediction [253] [254].

Meanwhile, it has been demonstrated that the model is effective in performing classification tasks [255] [256]. In [255], the author developed a hierarchical model in which the model is compared to traditional machine learning algorithms such as linear discriminant analysis, quadratic discriminant analysis, k-nearest neighbor algorithms, support vector machines, and traditional artificial neural networks. The ability of the network to perform classification is corroborated in [256] where the model is used to classify Camellia (Theaceae) species based on leaf characteristics.

While most literature focuses on the application of the algorithm to different context, an important revision of the algorithm is performed in [246] where modification to the algorithm is performed with regard to time series forecasting. The modification reformulates the network as an additive model and proposes a novel method to optimise the network where the network parameter μ is fixed to a few selections and linear regression is performed with regard to the additive model. Applications to datasets have demonstrated positive results.

7.3.2 The Optimisation Problem

The original algorithm entails two optimisation processes [154]. The first process is the determination of the number of layers with regard to training and validation accuracy. There are two criteria of accuracy proposed for this model:

$$\varepsilon_1 = (SSE_k - SSE_{k+1})/SSE_k \leq \varepsilon_1^\star \tag{7.3}$$

$$\varepsilon_2 = |MSE_T - MSE_v|/MSE_T \leq \varepsilon_2^\star \tag{7.4}$$

The first criterion, as enlisted in Equation 7.3, defines the training loss. When the fractional changes in sum of squares error (SSE) with an addition of a layer is smaller than a benchmark ($\varepsilon_1^\star$), the training stops and the network layers are fixed. The second criterion, as enlisted in Equation 7.4, defines the testing loss. When the MSE of training dataset converges to MSE of validation dataset and the difference is smaller than a benchmark ($\varepsilon_2^\star$), the training process is stopped.

The second training problem aims to find the value of μ. This is achieved by optimising SSE relative to the μ based on Equation 7.2. To optimise SSE with respect to μ, I can adopt first-order methods such as the bisection method to find the minimum SSE. The derivation of the derivative values can be found in [234] and is formulated in Equation 7.5.

$$\begin{aligned}\delta(SSE_k)/\delta\mu_k = \sum_i [&(\hat{F}(X_i) - a_k - b_k F_{k-1}(X_i)) \times (c_k \times \alpha_1 \times \sin(\mu_k \times \alpha_i) \\ &- d_k \times \alpha_1 \times \cos(\mu_k \times \alpha_i)) \\ &+ (d_k^2 - c_k^2) \times \alpha_1 \times \sin(2 \times \mu_k \times \alpha_i)/2 \\ &+ c_k \times d_k \times \alpha_1 \times \cos(2 \times \mu_k \times \alpha_i)] = 0\end{aligned} \tag{7.5}$$

7.3.3 Analysis on Dynamic Architectural Methods

One distinct advantage of the DAN2 architecture is that it only has one free parameter to optimise: the number of hidden layers. This draws comparison to the traditional neural network where the number of hidden layers and the number of hidden neurons in each layer are all parameters to optimise. This results in a number of literature reviewing on optimising these numbers, such as grid search, Bayesian Optimisation, *etc*. Therefore, DAN2 surpasses these architectures by having a simpler optimisation regime. However, it also limits the flexibility of this network in comparison to other dynamic models such as cascade learning, rendering it a bit rigid to variations in the data. If I draw comparison to traditional neural network, on the other hand, the model is flexible due to its dynamic nature, adapting to the datasets by changing the total number of layers.

Another advantage is that the model encompasses layer-wise training. Layer-wise training enables applications such as larger models under memory constraints, model prototyping, joint model compression and training, parallelised training, and more stable training for challenging scenarios [257]. Moreover, layer-wise training has been shown to be able to deal with large dataset such as ImageNet [257], contrary to common beliefs.

One characteristic of the DAN2 network is that it freezes its parameters in each layer as more layers are added. There are three advantages associated with this characteristic: (1) it reduces computational complexity hence requires less computational time; (2) there is no need to store gradient information hence the memory storage is low, unlike in neural networks optimised through gradient based methods; and (3) it circumvents the vanishing gradient problem.

The vanishing gradient problem manifests itself in traditional neural networks when the automatic differentiation dictated by the chain rule generates very small numbers in the initial layers of the network, making the updates of parameters in the initial layers very slow. The DAN2 solves this problem by training from bottom-to-top, *i.e.* starting from the input layers, and freezes the coefficients for each layer along the process. Therefore, one can expect better-parsed parameters for the initial layers.

7.4 Methodology

In the previous section, I have reviewed the algorithm of DAN2 and discussed its characteristics. In this section, the methodology behind the application of the DAN2 network is described. I apply algorithmic modifications to achieve multi-task learning and implement the architectural search.

7.4.1 Dataset

There are two datasets to which I have applied our algorithm. Both datasets are used for regression tasks. I developed a first dataset to simulate a dynamic process of pressure swing adsorption (PSA). The PSA dataset contains 6 set of continuous input features and 3 continuous output values, *i.e.* the recovery rate, the purity and the energy consumption. Details of the data collection process and the data pre-processing can be found in Appendix. This dataset is used because (1) it contains highly correlated outputs obtained under the same operating conditions which are suitable for multi-task learning, and (2) the process is highly non-linear in nature and thus requires a good non-linear approximator such as DAN2 to model.

To ensure that the dataset is applicable to multi-task learning, I calculate the correlation matrix of the output values consisting of recovery rate, purity and energy consumption. The correlation matrix is shown in Table 7.1. From the matrix, I see correlations between the three values, indicating the applicability of multi-task learning.

Table 7.1 Correlation matrix for the output values of recovery rate, purity and energy consumption in the PSA dataset

	Recovery Rate	Purity	Energy Consumption
Recovery Rate	1	-0.672	0.652
Purity	-0.672	1	-0.499
Energy Consumption	0.652	-0.499	1

The second dataset used is the OILDROPLET dataset. The dataset contains 24,422 experimental entries, and within each entry, there are 7 input variables and the 4 output variables (*i.e.* average movement speed, maximum speed of a single droplet, average number of droplets in the last second, average number of droplets throughout the experiment). Details of the dataset can be found in Appendix. I make use of this dataset because it also involves an implicitly non-linear process under which DAN2 can be a suitable model.

Similarly, I calculate the correlation matrix of the output values to ensure a multi-task learning architecture is suitable. Table 7.2 tabulates the correlation matrix, demonstrating a strong correlation between output values of droplet sizes and between outputs of droplet speeds.

Table 7.2 Correlation matrix for the output values of the OILDROPLET dataset. S_{Ave} represents the average speed of droplets. $D_{FinalAve}$ represents the average number of droplets in the last second. S_{Max} represents the maximum average single droplet speed. D_{Ave} represents the average number of droplets.

	S_{Ave}	$D_{FinalAve}$	S_{Max}	D_{Ave}
S_{Ave}	1	0.157	0.871	0.0513
$D_{FinalAve}$	0.157	1	0.295	0.864
S_{Max}	0.871	0.295	1	0.244
D_{Ave}	0.0513	0.864	0.244	1

7.4.2 Data Pre-processing

I pre-process the input data through normalisation and standardisation. Empirical comparison of the two pre-processing methods demonstrate that normalisation tends to generate more accurate results. Therefore, I have adopted normalisation in our experimentation. Moreover, I split the data into training (49%), validation (21%) and testing (30%) dataset. I have put a larger weighting on validation set because I would like to have more data to determine the architectural parameter of the the network. Cross-validation is not performed under the scope of this research but as future work, cross-validation can be performed to ensure the generalisation of the proposed algorithm.

7.4.3 Multi-task Learning

The dynamic model is then modified for multi-task learning. Two architectures are proposed adopting a multi-head structure and a serial structure. In each proposed architecture, I propose two different methods to learn the architecture of the network: (a) the grid search and (b) an iterative heuristic search scheme. In the method of (a), the dynamic architecture is constructed with the free parameters representing the total number of shared or task-specific layers. The optimal architecture is found by altering the values of the free parameters. In the method of (b), the dynamic architecture is constructed such that each addition of layers, either shared or task-specific, is written into a subroutine. To search for the optimal number of layers requires the iterative running of the subroutines until convergence to a tolerance.

In either method, the predicting power of the architectures are evaluated and compared to traditional ANNs. Different datasets have been adopted to evaluate effectiveness of the developed network.

7.4.4 Evaluation

To evaluate the effectiveness of each model, I introduce the evaluative metrics of the Sum of Squared Error (SSE) (Equation 7.6) and the Mean Squared Error (MSE) (Equation 7.7). The metrics are further classified into errors of the training, testing and validation datasets.

$$SSE = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{7.6}$$

$$MSE = \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n} \tag{7.7}$$

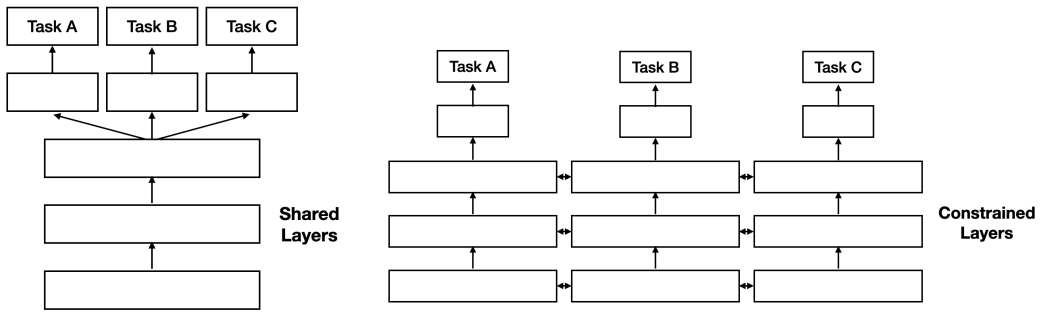


Figure 7.2 Structure of multi-task learning with (a) hard parameter sharing and (b) soft parameter sharing

7.5 Applications to Multi-task Learning

A novel architectural search algorithm is proposed where I inherit the gist of dynamical extension of networks of the DAN2 network. I introduce multi-task learning to this network and experiment with the results. The PSA dataset and the OILDROPLET dataset are adopted to train and validate the multi-task learning network.

In the PSA dataset, I define the three tasks to be performed in parallel as predicting recovery rate, purity and energy consumption respectively. I believe our data is a good fit for multi-tasking because there is strong correlation between the tasks - all three predictions are obtained under the same operating condition. Therefore, there is shared hidden representation between the different tasks.

Similarly in the OILDROPLET dataset, the tasks are defined to be the prediction of average speed of droplets, average number of droplets in the last second, maximum average single droplet speed and average number of droplets throughout recording. The multi-task learning model is a suitable model also becuase there is correlation between the tasks, and thus in the networks, layers can be shared between the tasks.

Multi-task learning has wide applications in many fields including drug discovery [258] [259], computer vision [260] [261], speech recognition [262] [263] and natural language processing [264] [265]. Techniques involving multi-task learning can be separated into hard parameter sharing and soft parameter sharing. In hard parameter sharing, the shared layers for different tasks have equal weights while a few output layers are designed for each specific task. The structure is demonstrated in Figure 7.2(a). In soft parameter sharing, separate networks are developed for each task but the weights are regularised to be close to each other. The structure of soft parameter sharing is demonstrated in Figure 7.2(b).

Our model adopts the hard parameter sharing scheme where the same weights are shared and fixed for different tasks. The advantage of a hard parameter sharing is that it greatly

reduces over-fitting, allowing the network to generalise better between tasks [122]. Soft parameter sharing is not adopted since it in essence trains separate networks for different tasks, taking up more computation and memory to develop and store.

There are several advantages of multi-task learning [122]. The first is implicit data augmentation, where sample size is increased with data from different tasks and the model trained will better represent the hidden distribution of a more general representation. The second is attention focusing, where the model puts more emphasis on important features with relevance confirmed by auxiliary tasks. The third is eavesdropping, where a difficult to learn task can be learned effectively through auxiliary tasks. The fourth is representation bias, as the model is biased to learn similar problems since it generalises well within our defined problems. The last is regularisation, as sharing parameters prevents over-fitting to a specific task.

A technical issue of applying multi-task learning is whether to freeze the weights with the addition of a new layer or to retrain the whole network. While the former introduces less re-computation, the latter finds the more optimal solution. This is because holding other weights constant only allows search along the affine subspace of the weight space [227]. However, the design of the DAN2 network is such that weights are frozen in each iteration, as the addition of each layer is a deterministic step with no free parameter. Therefore, it is reasonable to freeze the weights without the need to retrain the whole network, saving computations as an added benefit.

The advantage of the DAN2 network to be applied to multi-task learning is that it freezes the parameters in each layer after training is performed and adds in new layers without affecting the parameters of the previous layers. This greatly simplifies the calculation of the shared layers. Our dataset is suitable for multi-task learning since the output values are generated based on the same operating conditions as inputs. Therefore, I expect strong correlation between different outputs and a shared representation is suitable for the data at hand.

7.6 Mathematical Formulation

The multi-task learning problem can be formulated as a multi-objective optimisation problem [266]. Here I summarise key concepts discussed in [266].

Suppose the input space is represented as $\mathscr{X}$ and the output space $\{\mathscr{Y}^t\}_{t\in[T]}$. The dataset is a set of i.i.d. data points represented as $\{x_i, y_i^1, \cdots, y_i^T\}_{i\in[N]}$ where x_i is a particular input that is related to outputs y_i of tasks 1 to T, and N is the total number of data points. The learning problem for a particular task is defined as $f^t(x;\theta^{sh},\theta^t) : \mathscr{X} \to \mathscr{Y}^t$, where θ^{sh} are

the shared parameters and θ^t are task-specific parameters. The task-specific loss function is defined as $\mathscr{L}^t(\cdot,\cdot) : \mathscr{Y}^t \times \mathscr{Y}^t \to \mathbb{R}^+$.

In most empirical cases, the overall loss function is defined as a weighted average of losses, where the weights c^t can be static or dynamically computed. $\hat{\mathscr{L}}^t(\theta^{sh}, \theta^t)$ is the empirical loss of a particular task t, which is defined in Equation 7.9.

$$\min_{\theta^{sh},\theta^1,\cdots,\theta^T} \sum_{t=1}^{T} c^t \hat{\mathscr{L}}^t(\theta^{sh}, \theta^t) \tag{7.8}$$

$$\hat{\mathscr{L}}^t(\theta^{sh}, \theta^t) = \frac{1}{N} \sum_i \mathscr{L}(f^t(x_i; \theta^{sh}, \theta^t), y_i^t) \tag{7.9}$$

To formulate the problem as a multi-objective optimisation problem, I define loss $\mathbf{L}$:

$$\min_{\theta^{sh},\theta^1,\cdots,\theta^T} \mathbf{L}(\theta^{sh}, \theta^1, \cdots, \theta^T) = \min_{\theta^{sh},\theta^1,\cdots,\theta^T} (\hat{\mathscr{L}}^1(\theta^{sh}, \theta^1), \cdots, \hat{\mathscr{L}}^T(\theta^{sh}, \theta^T))^T \tag{7.10}$$

The goal of multi-objective optimisation is to achieve Pareto optimality.
Definition (Pareto optimality)

(a) A solution θ dominates a solution $\bar{\theta}$ if $\hat{\mathscr{L}}^t(\theta^{sh}, \theta^t) \leq \hat{\mathscr{L}}^t(\bar{\theta}^{sh}, \bar{\theta}^t)$ for all tasks t and $\mathbf{L}(\theta^{sh}, \theta^1, \cdots, \theta^T) \neq \mathbf{L}(\bar{\theta}^{sh}, \bar{\theta}^1, \cdots, \bar{\theta}^T)$

(b) A solution $\theta^\star$ is called Pareto optimal if there exists no solution θ that dominates $\theta^\star$.

In the context of DAN2, the loss function is defined as *MSE* of the testing dataset. The shared parameter θ^{sh}, is the free hyperparameter of the total number of shared layers. The task-specific parameter θ^t is the free hyperparameter of the total number of task-specific layers. To find Pareto optimality, a grid search of the shared and task-specific parameters are performed where condition (a) holds. The best performing parameters derived from the grid search are assumed to be non-dominant by other parameter values, fulfilling condition (b). The multi-objective optimisation problem is thus simplified to optimising two hyperparameters regarding the architecture, while minimising the vector-formed loss function $\mathbf{L}(\theta^{sh}, \theta^1, \cdots, \theta^T)$.

7.7 Optimisation of Multi-head Architecture

I propose a multi-task learning network where the first few hidden layers are kept the same for different learning tasks. I adopt the hard parameter sharing scheme and due to the fixation of most parameters of the model, the only free hyperparameter is the number of layers. A

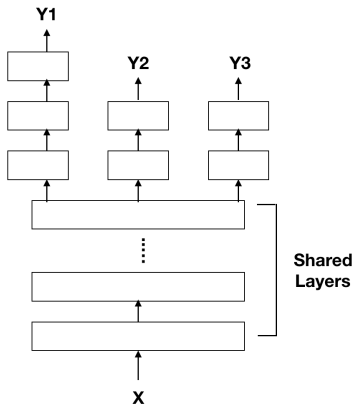


Figure 7.3 A multi-head architecture adopted with different number of task-specific layers for each task.

hyperparameter ξ_{sh} is introduced to represent the number of shared hidden layers. After ξ_{sh} layers, the rest of the network layers are trained separately for different learning tasks. The structure of the proposed model is demonstrated in Figure 7.3.

In the first ξ_{sh} layers, the network is trained with regard to the whole dataset. I convert the three dimensions of the output into single one-dimensional output through concatenation. During training of the shared layers, the data are randomly sampled from these one-dimensional output. During the training of the task-specific layers, only task-specific outputs are sampled to train the layers.

7.7.1 Grid Search

It is important to determine the architectural parameters of the network. The first method to find the architectural parameters, *i.e.* the value of ξ_{sh} and the total number of layers, n, is through a grid search. $n - \xi_{sh}$ represents the number of task-specific layers. The searched architectural parameters range from 3 to 8 for both shared and task-specific layers. I have enlisted the results of grid search adopting 3-8 layers in Table 7.3.

From Tables 7.3, it can be observed that the training error is close to zero in all cases searched. This demonstrates that the model is good at fitting training data with the possibility of overfitting. Therefore, Table 7.4 tabulates the validation error from a grid search. This serves to check the presence of overfitting.

From the results, it is evident that although no significant signs of overfitting is observed, the values start to diverge when the total number is becoming higher. For example, at $(\xi = 8, n - \xi = 8)$ for Energy Consumption, the values of validation error becomes higher, indicating possible overfitting starting from that point. To testify the presence of overfitting, I

Table 7.3 Training MSE of the grid search for multi-task DAN2 network: tasks of predicting recovery rate, purity and energy consumption. $n-\xi_{sh}$ represents the number of task-specific layers and ξ_{sh} represents the number of shared layers.

Task	Number of Layers		ξ_{sh}		
			3	5	8
Recovery rate	$n-\xi_{sh}$	3	5.213×10^{-8}	3.992×10^{-8}	4.791×10^{-8}
		5	5.013×10^{-8}	5.358×10^{-8}	4.700×10^{-8}
		8	4.764×10^{-8}	5.034×10^{-8}	5.011×10^{-8}
Purity	$n-\xi_{sh}$	3	1.088×10^{-7}	1.623×10^{-7}	$1.589\times10-7$
		5	1.447×10^{-7}	9.919×10^{-8}	1.223×10^{-7}
		8	1.380×10^{-7}	1.272×10^{-7}	9.909×10^{-8}
Energy Consumption	$n-\xi_{sh}$	3	4.495×10^{-7}	4.375×10^{-7}	4.616×10^{-7}
		5	4.413×10^{-7}	4.350×10^{-7}	2.396×10^{-6}
		8	4.711×10^{-7}	4.332×10^{-7}	4.381×10^{-7}

have performed several runs on the point at $(\xi=8, n-\xi=8)$ for Energy Consumption values, and large values (>1.0) are obtained in most cases, indicating the presence of overfitting.

Similarly, I apply the multi-task DAN2 network to the dataset of OILDROPLET. The results are tabulated in Table 7.5 and Table 7.6.

The results demonstrate that the multi-head DAN2 network is able to arrive at low training MSEs since all training MSEs are in the range $10^{-3}-10^{-2}$. However, the validation losses demonstrate that the algorithm is unstable, sometimes generating high values of validation losses. The situation varies with different selections of the random reference vector. The instability can be attributed to the different definition of the random reference vector in the fitting and the predicting process. With a different random vector, the values of the angle α as input to the trigonometric function are different. On the other hand, the parameters are optimised using the random reference vector in the fitting process, thus generating unstable prediction performance.

The challenge of using grid search is that a high number of points are needed to be searched for in order to arrive at the optimal architecture. For example, in the case of the PSA dataset, if I set the choices of the number of task-specific layers to be only 3, 5, 8, but I allow the possibility of having different number of task-specific layers, I need to optimise the network for "$3\times3\times3\times3$" times to observe over-fitting or under-fitting. This requires in total a search of 81 points in space. Moreover, if the number of layers to search for is high, *i.e.* the number is over 100, the grid search will be computationally tedious and expensive. There is also no proof that the search is exhaustive. Therefore, the second method, the heuristic search scheme, is proposed to manage this problem.

Table 7.4 Validation MSE of the grid search for multi-task DAN2 network: tasks of predicting recovery rate, purity and energy consumption. $n - \xi_{sh}$ represents the number of task-specific layers and ξ_{sh} represents the number of shared layers.

Task	Number of Layers		ξ_{sh}		
			3	5	8
Recovery rate	$n - \xi_{sh}$	3	3.950×10^{-5}	9.225×10^{-6}	6.579×10^{-5}
		5	2.783×10^{-5}	8.316×10^{-5}	1.858×10^{-5}
		8	1.436×10^{-5}	6.459×10^{-3}	5.104×10^{-5}
Purity	$n - \xi_{sh}$	3	3.559×10^{-2}	5.749×10^{-5}	$9.900 \times 10-3$
		5	9.624×10^{-3}	3.144×10^{-6}	1.163×10^{-4}
		8	8.298×10^{-6}	3.753×10^{-6}	1.640×10^{-6}
Energy Consumption	$n - \xi_{sh}$	3	7.320×10^{-6}	3.027×10^{-6}	4.784×10^{-7}
		5	4.413×10^{-7}	4.079×10^{-5}	4.211×10^{-6}
		8	1.694×10^{-2}	3.303×10^{-2}	1.086

7.7.2 Heuristic Search Scheme

The grid search, although systematic, is time-consuming and limited in scope. Finding each set of optimal architectural parameters requires the rerunning of the whole network. Thus, only a small number of architectures can be evaluated in a limited number of runs. To determine the architectural parameters to evaluate based on a grid is also a difficult task. The size of the grid, the number of points to evaluate, the spread of the points on the grid, are all free parameters to be determined and they may significantly affect the performance of the network.

Therefore, I propose a heuristic search scheme for the optimisation of multi-task DAN2 network. The optimisation scheme is heuristic in nature where a sequential method is adopted. In the original network where only one output is calculated, the total number of layers is the only free parameter to adjust and is settled when the percentage difference between two consecutive SSEs is smaller than a user-defined value $\varepsilon^{\star}$. In the case of multi-task learning, the total number of layers consists of the number of shared layers ξ_{sh} and the number of task-specific layers ξ_{ts}. The heuristics become more complicated when the number of tasks increases. For example, when there are three tasks to regress, the total number of parameters to optimise increases to 4, including ξ_{sh} and ξ_1, ξ_2, ξ_3 for each task.

Traditional methods to optimise a high number of architectural parameters include random search, grid search, evolutionary algorithms, Bayesian optimisation or reinforcement learning. Random search is unsystematic and grid search covers a small space to optimise. The other methods are complicated in nature and require specific implementation of packages.

Table 7.5 Training MSE of the grid search for multi-task DAN2 network for the OIL-DROPLET dataset. $n-\xi_{sh}$ represents the number of task-specific layers and ξ_{sh} represents the number of shared layers.

Task	Number of Layers		ξ_{sh}		
			1	2	3
S_Ave	$n-\xi_{sh}$	1	3.812×10^{-3}	3.795×10^{-3}	3.794×10^{-3}
		2	3.781×10^{-3}	3.914×10^{-3}	3.776×10^{-3}
		3	3.766×10^{-3}	3.841×10^{-3}	3.735×10^{-3}
$D_{FinalAve}$	$n-\xi_{sh}$	1	5.032×10^{-3}	5.162×10^{-3}	4.938×10^{-3}
		2	5.160×10^{-3}	5.197×10^{-3}	5.073×10^{-3}
		3	4.946×10^{-3}	5.031×10^{-3}	5.029×10^{-3}
S_{Max}	$n-\xi_{sh}$	1	1.534×10^{-2}	1.530×10^{-2}	1.548×10^{-2}
		2	1.535×10^{-2}	1.578×10^{-2}	1.534×10^{-2}
		3	1.540×10^{-2}	1.528×10^{-2}	1.531×10^{-2}
D_{Ave}	$n-\xi_{sh}$	1	3.984×10^{-3}	3.978×10^{-3}	3.979×10^{-3}
		2	3.987×10^{-3}	4.105×10^{-3}	4.005×10^{-3}
		3	3.991×10^{-3}	4.000×10^{-3}	3.998×10^{-3}

This section proposes a novel heuristic method to sequentially optimise the number of shared layers and the number of task-specific layers. The method provides a systematic framework in which the optimal multi-head structure can be found. Although there is no proof of optimality in the structure generated, the dynamic nature of the network dictates that a relatively minimal structure extended from one basic initial layer is obtained.

The optimisation scheme consists of two steps: (a) increasing the number of shared layers by 1, and (b) increasing the number of task-specific layers by 1. The stopping criterion for Step (a) is defined as $\varepsilon_{total} \geq \varepsilon^{\star}_{total}$ where $\varepsilon^{\star}_{total}$ is a user-defined small value and $\varepsilon_{total} = (SSE_{total} - SSE_{total,prev})/SSE_{total,prev}$. Since layers are shared between tasks, the total SSE for all tasks is used in the calculation of the stopping criterion. The stopping criterion for Step (b) is defined as $\varepsilon_{task} \geq \varepsilon^{\star}_{task}$ where $\varepsilon^{\star}_{task}$ is another user-defined small value and $\varepsilon_{task} = (SSE_{task} - SSE_{task,prev})/SSE_{task,prev}$. The task-specific SSE is used because I would like to train the layers such that the number of layers differs for each task.

I employ an iterative procedure where Step (a) and Step (b) alternates until the overall SSE is smaller than a tolerance value. The tolerance value is arbitrarily small and determines the structure of the network. The pseudocode for the optimisation of the multi-task learning network is shown in Algorithm 10.

I apply the multi-head structure to both datasets. In the PSA dataset, the optimal architectural parameters combined with training and testing MSE are listed in Table 7.7. It is evident

Algorithm 10 Optimisation of DAN2 Network

Initialize: $R \rightarrow Random$, $X \rightarrow Input$, $y \rightarrow Output$ and $SSE \rightarrow \infty$
$\alpha = \arccos(R * X)$
$F_0 = LinearRegressionFit(X, y)$
$G_1 = \cos(\alpha)$ and $H_1 = \sin(\alpha)$
$F_1 = LinearRegressionFit(F_0, G_1, H_1)$
$a_1, b_1, c_1, d_1 \rightarrow$ Coefficients of F_1
$\xi_{sh} = 0$ and $\xi_{ts} = 0$
function CALCULATE(a, b, c, d, F_{prev})
    $F = a + bF_{prev} + c\cos(\mu\alpha) + d\sin(\mu\alpha)$
end function
while $SSE \geq tol$ **do**
    while $\varepsilon_{total} \geq \varepsilon^{\star}_{total}$ **do**
        $\xi_{sh} = \xi_{sh} + 1$
        $\mu_{\xi_{sh}} \rightarrow \arg\max_{\mu} F$
        $G_{\xi_{sh}} = \cos(\mu_{\xi_{sh}}\alpha)$ and $H_{\xi_{sh}} = \sin(\mu_{\xi_{sh}}\alpha)$
        $F_{\xi_{sh}} = LinearRegressionFit(F_{\xi_{sh}-1}, G_{\xi_{sh}}, H_{\xi_{sh}})$
        $a_{\xi_{sh}}, b_{\xi_{sh}}, c_{\xi_{sh}}, d_{\xi_{sh}} \rightarrow$ Coefficients of $F_{\xi_{sh}+1}$
        $SSE = \sum(F_{\xi_{sh}} - y)^2$
        $\varepsilon_{total} = (SSE - SSE_{total,prev})/SSE_{total,prev}$
    end while
    while $\varepsilon_{task} \geq \varepsilon^{\star}_{task}$ **do**
        $\xi_{ts} = \xi_{ts} + 1$
        $\mu_{\xi_{ts}} \rightarrow \arg\max_{\mu} F$
        $G_{\xi_{ts}} = \cos(\mu_{\xi_{ts}}\alpha)$ and $H_{\xi_{ts}} = \sin(\mu_{\xi_{ts}}\alpha)$
        if $\xi_{ts} == 1$ **then**:
            $F_{\xi_{ts}} = LinearRegressionFit(F_{\xi_{sh}}, G_{\xi_{ts}}, H_{\xi_{ts}})$
        else:
            $F_{\xi_{ts}} = LinearRegressionFit(F_{\xi_{ts}-1}, G_{\xi_{ts}}, H_{\xi_{ts}})$
        end if
        $a_{\xi_{ts}}, b_{\xi_{ts}}, c_{\xi_{ts}}, d_{\xi_{ts}} \rightarrow$ Coefficients of $F_{\xi_{ts}+1}$
        $SSE = \sum(F_{\xi_{ts}} - y)^2$
        $\varepsilon_{task} = (SSE - SSE_{ts,prev})/SSE_{ts,prev}$
    end while
end while

Table 7.6 Validation MSE of the grid search for multi-task DAN2 network for the OIL-DROPLET dataset. $n - \xi_{sh}$ represents the number of task-specific layers and ξ_{sh} represents the number of shared layers.

Task	Number of Layers		ξ_{sh}		
			1	2	3
S_{Ave}	$n - \xi_{sh}$	1	1.081×10^{2}	8.361×10^{1}	1.316×10^{0}
		2	6.422×10^{-1}	5.305×10^{-3}	4.472×10^{1}
		3	2.738×10^{2}	9.753×10^{-1}	1.392×10^{1}
$D_{FinalAve}$	$n - \xi_{sh}$	1	3.546×10^{2}	3.473×10^{-2}	7.950×10^{0}
		2	7.634×10^{-2}	5.529×10^{-3}	1.794×10^{1}
		3	6.729×10^{1}	2.705×10^{1}	7.596×10^{1}
S_{Max}	$n - \xi_{sh}$	1	1.148×10^{2}	4.376×10^{-1}	6.486×10^{1}
		2	1.554×10^{1}	2.607×10^{-2}	1.653×10^{1}
		3	4.211×10^{0}	4.921×10^{1}	8.850×10^{2}
D_{Ave}	$n - \xi_{sh}$	1	2.801×10^{2}	5.279×10^{-1}	5.144×10^{-1}
		2	2.557×10^{2}	4.029×10^{-3}	9.362×10^{-1}
		3	3.125×10^{2}	1.591×10^{1}	2.344×10^{1}

that the multi-head structure is capable of producing a network generating small training and validation losses. The value for Energy Consumption diverges because of outlier values in the dataset. Since in each layer there are 5 network parameters and there is a total of 23 layers, the total number of network parameters is 115.

Table 7.7 Training and validation results of applications of multi-head DAN2 to the PSA dataset. The training loss is reported as SSE and the validation loss is reported as MSE. The average training loss is defined as the sum of individual SSEs for each task-specific layer. The average validation loss is defined as the average of MSEs for each task-specific layer.

Layer	Number	Training Loss	Validation Loss
Shared layers	8	6.272×10^{-3}	1.302×10^{-6}
Task-specific layers (Recovery Rate)	6	8.620×10^{-5}	2.690×10^{-6}
Task-specific layers (Purity)	4	2.861×10^{-4}	1.335×10^{-6}
Task-specific layers (Energy Consumption)	5	7.507×10^{-4}	3.937
Average	-	3.743×10^{-4}	1.312

The application to OILDROPLET dataset is shown in Table 7.8. In this case, no divergence is observed and both the training and validation losses are small and the number of layers used is very limited. This demonstrates the effectiveness of the network acting as an approximator to nonlinear processes. Adapting to a multi-head structure does not limit the

effectiveness of the network in generating small training and validation losses. Since there are 5 network parameters in each layer and there are 10 layers in total, the total number of parameters used is 50.

Table 7.8 Training and validation results of applications of multi-head DAN2 to the OIL-DROPLET dataset. The training loss is reported as SSE and the validation loss is reported as MSE. The average training loss is defined as the sum of individual SSEs for each task-specific layer. The average validation loss is defined as the average of MSEs for each task-specific layer.

Layer	Number	Training Loss	Validation Loss
Shared layers	2	8.019×10^{-3}	8.047×10^{-3}
Task-specific layers (S_{Ave})	2	3.914×10^{-3}	5.305×10^{-3}
Task-specific layers ($D_{FinalAve}$)	2	5.197×10^{-3}	5.529×10^{-3}
Task-specific layers (S_{Max})	2	1.578×10^{-2}	2.607×10^{-2}
Task-specific layers (D_{Ave})	2	4.105×10^{-3}	4.029×10^{-3}
Average	-	7.248×10^{-3}	1.364×10^{-2}

7.7.3 Comparison with Traditional Neural Network

I construct artificial neural networks to compare the performance of the DAN2 network with traditional artificial neural network. The hyperparameters of the traditional deep neural network is optimised manually. I control the total number of network parameters in the ANN to be the same as that in DAN2 model.

The DAN2 network for PSA dataset has 115 parameters. A comparable ANN with one hidden layer of 11 neurons entails 113 network parameters, since the input dimension is 6 and the output dimension is 3. I train the network using Adam optimiser with MSE loss. Empirical experimentation has demonstrated that both networks generate comparable results as enlisted in Table 7.9, with the exception of divergence. From Table 7.9, it can be observed that the DAN2 training MSE is of a similar order of scale to that from a traditional ANN. For validation MSE, values are of similar order of scale except the divergent value in DAN2 network. The divergence is possibly due to the initial shared structure in the multi-head structure which limits the performance of prediction in each head. Overall, traditional ANN performs slightly better but rather comparable to the multi-DAN2 network.

The DAN2 network for OILDROPLET dataset has 50 parameters. A comparable ANN with a similar number of parameters will entail a network with 52 parameters. The structure

Table 7.9 Comparison between DAN2 and ANN with applications to the PSA dataset.

	ANN Training MSE	ANN Validation MSE
Recovery Rate	1.647×10^{-4}	1.042×10^{-7}
Purity	2.879×10^{-4}	1.965×10^{-7}
Energy Consumption	1.896×10^{-3}	1.069×10^{-6}
Average	7.828×10^{-4}	4.566×10^{-7}

	DAN2 Training MSE	DAN2 Validation MSE
Recovery Rate	8.620×10^{-5}	2.690×10^{-6}
Purity	2.861×10^{-4}	1.355×10^{-6}
Energy Consumption	7.507×10^{-4}	3.937
Average	1.123×10^{-3}	1.312

consists of one hidden layer with 4 neurons. The comparison between the ANN and the DAN2 network is enlisted in Table 7.10.

Table 7.10 Comparison between DAN2 and ANN with applications to the OILDROPLET dataset.

	ANN Training MSE	ANN Validation MSE
S_{Ave}	4.340×10^{-3}	4.700×10^{-3}
$D_{FinalAve}$	7.090×10^{-3}	7.593×10^{-3}
S_{Max}	1.970×10^{-2}	2.021×10^{-2}
D_{Ave}	5.542×10^{-3}	5.868×10^{-3}
Average	9.168×10^{-3}	9.593×10^{-3}

	DAN2 Training MSE	DAN2 Validation MSE
S_{Ave}	3.914×10^{-3}	5.305×10^{-3}
$D_{FinalAve}$	5.197×10^{-3}	5.529×10^{-3}
S_{Max}	1.578×10^{-2}	2.607×10^{-2}
D_{Ave}	4.105×10^{-3}	4.029×10^{-3}
Average	7.248×10^{-3}	1.364×10^{-2}

From Table 7.10, all training MSEs are of the same order of scale for the traditional ANN and multi-DAN2 network. This is similarly observed in validation MSE with some values of multi-DAN2 over-performing and some values under-performing the traditional ANN network.

7.7.4 Time and Memory Analysis

CPU Time Analysis

A comparison between the CPU time required for the training of each network is performed. The results are tabulated in Table 7.11. The architecture of the networks are the same as the ones used in the analysis above. From the results, it is evident that the networks of ANN and DAN2 spend similar amount of time on the tasks of regressing multi-dimensional output using multi-task learning architecture.

Table 7.11 Comparison between the CPU time of DAN2 and ANN with applications to the PSA and OILDROPLET datasets.

	ANN CPU Time (s)	DAN2 CPU Time (s)
PSA	0.503	0.0853
OILDROPLET	0.495	0.595

Memory Storage

The method does not require the storage of matrices. The coefficients in each layer are stored as separate vectors. There is no need of matrix multiplication and only manipulation of vectors is required. The total number of vectors to store depends on the number of layers used in the network hence this number varies. Overall, the memory storage is of $O(n)$.

7.8 Optimisation of Serial Architecture

I propose a serial structure of multi-task learning as shown in Figure 7.4. The input undergoes transformations from shared layers and the task-specific layers are arranged sequentially such that the number of layers an output undergoes differ for different outputs. This structure is adopted because I observe that the outputs are correlated and some outputs need to undergo a higher number of layers in order to generate accurate results.

The structure is motivated by the correlated nature of the potential outputs to the system. This structure can also be adapted for time-series data where a number of outputs into the future can be predicted from each input.

In this case, there are several free architectural parameters to optimise. I use ξ_1, ξ_2, ξ_3, *etc.* to represent the different number of task-specific layers, and ξ_{sh} as the number of shared layers. Thus, the architectural search problem is converted to an optimisation problem with

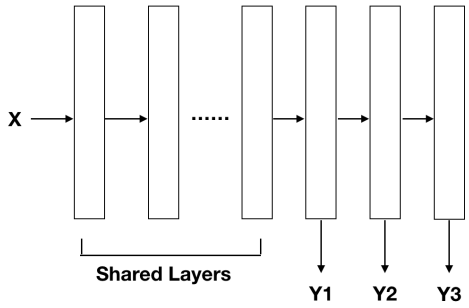


Figure 7.4 A serial structure of the multi-task learning network

more than one free parameters to optimise. The objective function will be the training or validation SSE of the whole network, which is a complicated, non-linear and non-convex function relative to the free parameters.

7.8.1 Grid Search

Common optimisation practices include random search, grid search, Bayesian optimisation, reinforcement learning and evolutionary algorithms. I optimise the architectural parameters using a grid search. The results for training are tabulated in Table 7.12 and the results for validation are tabulated in Table 7.13.

Table 7.12 Training MSE generated from a grid search adopting the serial structure for the implementation of the PSA dataset.

ξ_{sh}	ξ_1	ξ_2	ξ_3	MSE_1	MSE_2	MSE_3	*Average MSE*
2	1	1	1	2.868×10^{-8}	1.038×10^{-7}	6.057×10^{-7}	7.382×10^{-7}
	2	2	2	2.826×10^{-8}	1.013×10^{-7}	6.367×10^{-7}	7.662×10^{-7}
	3	3	3	2.769×10^{-8}	1.018×10^{-7}	6.514×10^{-7}	7.809×10^{-7}
	4	4	4	2.860×10^{-8}	1.042×10^{-7}	6.060×10^{-7}	7.388×10^{-7}
3	1	1	1	2.865×10^{-8}	1.066×10^{-7}	5.904×10^{-7}	7.257×10^{-7}
	2	2	2	2.840×10^{-8}	1.025×10^{-7}	6.254×10^{-7}	7.563×10^{-7}
	3	3	3	2.852×10^{-8}	1.044×10^{-7}	6.050×10^{-7}	7.379×10^{-7}
	4	4	4	2.810×10^{-8}	1.039×10^{-7}	6.370×10^{-7}	7.690×10^{-7}
4	1	1	1	2.856×10^{-8}	1.048×10^{-7}	6.090×10^{-7}	7.424×10^{-7}
	2	2	2	2.842×10^{-8}	1.066×10^{-7}	6.132×10^{-7}	7.482×10^{-7}
	3	3	3	2.854×10^{-8}	1.060×10^{-7}	5.995×10^{-7}	7.340×10^{-7}
	4	4	4	2.859×10^{-8}	1.058×10^{-7}	5.914×10^{-7}	7.258×10^{-7}

Table 7.13 Validation MSE generated from a grid search adopting the serial structure for the implementation of the PSA dataset.

ξ_{sh}	ξ_1	ξ_2	ξ_3	MSE_1	MSE_2	MSE_3	*Average MSE*
2	1	1	1	2.904×10^{-8}	4.067×10^{-7}	4.602×10^{-6}	1.679×10^{-6}
	2	2	2	9.897×10^{-8}	4.680×10^{-7}	5.183×10^{-6}	1.917×10^{-6}
	3	3	3	1.176×10^{-7}	4.841×10^{-7}	4.942×10^{-6}	1.848×10^{-6}
	4	4	4	2.933×10^{-8}	3.866×10^{-7}	4.591×10^{-6}	1.669×10^{-6}
3	1	1	1	2.906×10^{-8}	3.962×10^{-7}	4.575×10^{-6}	1.667×10^{-6}
	2	2	2	5.479×10^{-8}	3.818×10^{-7}	5.501×10^{-6}	1.979×10^{-6}
	3	3	3	2.921×10^{-8}	3.950×10^{-7}	4.638×10^{-6}	1.687×10^{-6}
	4	4	4	1.826×10^{-7}	5.317×10^{-7}	6.310×10^{-6}	2.341×10^{-6}
4	1	1	1	3.376×10^{-8}	4.141×10^{-7}	4.919×10^{-6}	1.789×10^{-6}
	2	2	2	3.613×10^{-8}	4.244×10^{-7}	4.743×10^{-6}	1.735×10^{-6}
	3	3	3	2.976×10^{-8}	3.971×10^{-7}	4.554×10^{-6}	1.660×10^{-6}
	4	4	4	2.922×10^{-8}	4.190×10^{-7}	4.569×10^{-6}	1.672×10^{-6}

From Table 7.12 and Table 7.13, it can be observed that in the PSA dataset, a minimal structure of 2 shared layers and 1 task-specific layer for each task is capable of producing a small training and validation MSE.

I similarly apply the grid search on the multi-task DAN2 network to the OILDROPLET dataset. The results are tabulated in Table 7.14 and Table 7.15. From the results, it can be observed that the network is able to arrive at a low training and validation MSE, indicating the validity and effectiveness of the serial model.

The grid search is effective in finding the optimal architecture of the DAN2 network. However, the grid search is rigid, in the sense that it is difficult to determine a particular number of layers for task-specific layers. There is also no guidance with regard to how many layers to adopt for shared layers. Therefore, the heuristic scheme is proposed to solve these problems.

7.8.2 Heuristic Search Scheme

In Section 7.7.2, I proposed a novel algorithm for the optimisation of a multi-head structure. In this algorithm, I similarly adopt a heuristic scheme where the number of shared layers ξ_{sh} and the number of task-specific layers ξ_{ts} are optimised sequentially in an iterative manner. The pseudocode is the same as Algorithm 10 since both methods share the overall structure and the subroutines.

Table 7.14 Training MSE generated from a grid search adopting the serial structure for the implementation of the OILDROPLET dataset.

ξ_{sh}	ξ_1	ξ_2	ξ_3	ξ_4	MSE_1	MSE_2	MSE_3
2	1	1	1	1	3.629×10^{-3}	1.533×10^{-2}	5.254×10^{-3}
	2	2	2	2	3.600×10^{-3}	1.511×10^{-2}	5.217×10^{-3}
	3	3	3	3	3.621×10^{-3}	1.527×10^{-2}	5.244×10^{-3}
	4	4	4	4	3.631×10^{-3}	1.535×10^{-2}	5.258×10^{-3}
3	1	1	1	1	3.595×10^{-3}	1.508×10^{-2}	5.205×10^{-3}
	2	2	2	2	3.612×10^{-3}	1.519×10^{-2}	5.250×10^{-3}
	3	3	3	3	3.616×10^{-3}	1.527×10^{-2}	5.254×10^{-3}
	4	4	4	4	3.624×10^{-3}	1.530×10^{-2}	5.233×10^{-3}
4	1	1	1	1	3.614×10^{-3}	1.526×10^{-2}	5.237×10^{-3}
	2	2	2	2	3.617×10^{-3}	1.519×10^{-2}	5.203×10^{-3}
	3	3	3	3	3.615×10^{-3}	1.527×10^{-2}	5.262×10^{-3}
	4	4	4	4	3.622×10^{-3}	1.524×10^{-2}	5.187×10^{-3}

ξ_{sh}	ξ_1	ξ_2	ξ_3	ξ_4	MSE_4	*Average MSE*
2	1	1	1	1	6.681×10^{-3}	3.089×10^{-2}
	2	2	2	2	6.649×10^{-3}	3.057×10^{-2}
	3	3	3	3	6.671×10^{-3}	3.080×10^{-2}
	4	4	4	4	6.686×10^{-3}	3.092×10^{-2}
3	1	1	1	1	6.633×10^{-3}	3.052×10^{-2}
	2	2	2	2	6.674×10^{-3}	3.073×10^{-2}
	3	3	3	3	6.680×10^{-3}	3.082×10^{-2}
	4	4	4	4	6.670×10^{-3}	3.083×10^{-2}
4	1	1	1	1	6.651×10^{-3}	3.076×10^{-2}
	2	2	2	2	6.579×10^{-3}	3.059×10^{-2}
	3	3	3	3	6.688×10^{-3}	3.084×10^{-2}
	4	4	4	4	6.584×10^{-3}	3.063×10^{-2}

I apply the serial structure to the regression problem adopting the PSA and the OIL-DROPLET dataset. The results of the performance are presented in Table 7.16 and Table 7.17.

From the results in Table 7.17, it is evident that serial structure is capable of reaching low values of training and validations losses, demonstrating the effectiveness of the network structure. It is also evident that the number of layers used are comparable, though slightly higher, compared to that generated from a multi-head structure. The training and validation losses are also comparable, and slightly higher, compared to the losses in the multi-head

Table 7.15 Validation MSE generated from a grid search adopting the serial structure for the implementation of the OILDROPLET dataset.

ξ_{sh}	ξ_1	ξ_2	ξ_3	ξ_4	MSE_1	MSE_2	MSE_3
2	1	1	1	1	4.187×10^{-3}	2.437×10^{-1}	1.285×10^{-2}
	2	2	2	2	3.818×10^{-1}	2.153×10^{-1}	5.164×10^{-1}
	3	3	3	3	4.564×10^{-3}	3.234×10^{-2}	2.308×10^{-2}
	4	4	4	4	7.579×10^{-3}	2.942×10^{-2}	4.000×10^{-2}
3	1	1	1	1	3.638×10^{-3}	1.568×10^{-1}	1.373×10^{-2}
	2	2	2	2	2.607×10^{-2}	3.782×10^{-2}	7.058×10^{-1}
	3	3	3	3	1.055×10^{-2}	2.056×10^{-2}	5.908×10^{-2}
	4	4	4	4	2.594×10^{-2}	2.182×10^{-1}	6.176×10^{-2}
4	1	1	1	1	5.824×10^{-1}	1.845×10^{-1}	1.609×10^{-1}
	2	2	2	2	5.275×10^{-3}	3.045×10^{-2}	1.516×10^{-2}
	3	3	3	3	3.734×10^{-3}	2.295×10^{-2}	2.256×10^{-2}
	4	4	4	4	1.648×10^{-1}	1.466×10^{-1}	1.632×10^{-2}

ξ_{sh}	ξ_1	ξ_2	ξ_3	ξ_4	MSE_4	*Average MSE*
2	1	1	1	1	1.024×10^{-2}	6.774×10^{-2}
	2	2	2	2	7.297×10^{-1}	4.608×10^{-1}
	3	3	3	3	1.1534×10^{-2}	1.788×10^{-2}
	4	4	4	4	1.486×10^{-2}	2.297×10^{-2}
3	1	1	1	1	1.431×10^{-2}	4.712×10^{-2}
	2	2	2	2	1.115×10^{-1}	2.203×10^{-1}
	3	3	3	3	5.031×10^{-2}	3.513×10^{-2}
	4	4	4	4	6.693×10^{-1}	2.438×10^{-1}
4	1	1	1	1	1.409×10^{-1}	2.672×10^{-1}
	2	2	2	2	1.173×10^{-2}	1.566×10^{-2}
	3	3	3	3	9.458×10^{-3}	1.468×10^{-2}
	4	4	4	4	6.532×10^{-2}	9.825×10^{-2}

structure. In either case, the number of layers used is minimal, requiring a small number of parameters. Thus, they serve as effective alternative to artificial neural networks.

In the PSA dataset in particular, there is no divergence of the output in the prediction of energy consumption. This is a prominent improvement from the multi-head DAN2 network where the algorithm can be unstable and diverge in the prediction of energy consumption.

In comparison with the results generated from grid search, it can be observed that more freedom and flexibility is given in the search of an optimal architecture with the heuristic search method. It is possible to obtain different number of task-specific layers. Moreover,

Table 7.16 Training and validation results of the applications of the serial DAN2 to the PSA dataset. The training loss is reported as SSE and the validation loss is reported as MSE. The average training loss is defined as the sum of individual SSEs for each task-specific layer. The average validation loss is defined as the average of MSEs for each task-specific layer.

Layer	Number	Training Loss	Validation Loss
Shared layers	2	2.857×10^{-8}	2.968×10^{-8}
Task-specific layers (Recovery Rate)	4	2.846×10^{-8}	3.033×10^{-8}
Task-specific layers (Purity)	2	1.047×10^{-7}	4.089×10^{-7}
Task-specific layers (Energy Consumption)	3	5.979×10^{-7}	4.545×10^{-6}
Average	-	1.899×10^{-7}	1.254×10^{-6}

Table 7.17 Training and validation results of the application of the serial DAN2 to the OILDROPLET dataset. The training loss is reported as SSE and the validation loss is reported as MSE. The average training loss is defined as the sum of individual SSEs for each task-specific layer. The average validation loss is defined as the average of MSEs for each task-specific layer.

Layer	Number	Training Loss	Validation Loss
Shared layers	2	3.626×10^{-3}	1.173×10^{-2}
Task-specific layers (S_{Ave})	3	3.626×10^{-3}	3.987×10^{-3}
Task-specific layers ($D_{FinalAve}$)	2	1.533×10^{-2}	2.613×10^{-2}
Task-specific layers (S_{Max})	3	5.255×10^{-2}	1.205×10^{-2}
Task-specific layers (D_{Ave})	2	6.686×10^{-3}	1.053×10^{-2}
Average	-	1.636×10^{-2}	1.077×10^{-2}

both the grid search and the heuristic search arrive at a minimal architecture, although the grid search is more of a systematic guess of architectural optimality whereas the heuristic search is more guided.

7.8.3 Comparison with Traditional Neural Network

I perform comparison between the serial architecture and the ANNs with a similar number of parameters. The results are shown in Table 7.18 and 7.19.

In the PSA dataset, the serial structure has 11 layers with 55 parameters in total. A comparable ANN has one hidden layer of 5 neurons entailing 53 parameters. I train the network using Adam optimiser with regard to MSE loss. The results of running the ANN in comparison to the serial DAN2 network is shown in Table 7.18.

Table 7.18 Comparison between DAN2 and ANN with applications to the PSA dataset.

	ANN Training MSE	ANN Validation MSE
Recovery Rate	1.647×10^{-4}	1.042×10^{-7}
Purity	2.879×10^{-4}	1.965×10^{-7}
Energy Consumption	1.896×10^{-3}	1.068×10^{-6}
Average	7.829×10^{-4}	4.562×10^{-7}

	DAN2 Training MSE	DAN2 Validation MSE
Recovery Rate	2.846×10^{-8}	3.033×10^{-8}
Purity	1.047×10^{-7}	4.089×10^{-7}
Energy Consumption	5.979×10^{-7}	4.545×10^{-6}
Average	2.436×10^{-7}	1.661×10^{-6}

The second dataset of OILDROPLET makes use of 12 layers with 60 parameters. This is equivalent of an ANN with one hidden layer of 5 neurons with a total number of 59 parameters. I run the comparalbe ANN and report the results in Table 7.19.

Table 7.19 Comparison between DAN2 and ANN with applications to the OILDROPLET dataset.

	ANN Training MSE	ANN Validation MSE
S_{Ave}	4.340×10^{-3}	4.670×10^{-3}
$D_{FinalAve}$	7.090×10^{-3}	7.593×10^{-3}
S_{Max}	1.969×10^{-2}	2.020×10^{-2}
D_{Ave}	5.542×10^{-3}	5.868×10^{-3}
Average	9.166×10^{-3}	9.583×10^{-3}

	DAN2 Training MSE	DAN2 Validation MSE
S_{Ave}	3.626×10^{-3}	3.987×10^{-2}
$D_{FinalAve}$	1.533×10^{-2}	2.613×10^{-2}
S_{Max}	5.255×10^{-2}	1.205×10^{-2}
D_{Ave}	6.686×10^{-3}	1.053×10^{-2}
Average	1.955×10^{-2}	2.215×10^{-2}

In either table, the performance of DAN2 is comparable to that of ANNs with some entries having a better performance than ANNs. As the MSE values are close to zero, it follows that both networks are effective in generating accurate predictions based on the

results. Therefore, DAN2 network with a serial architecture can act as a good alternative to ANNs.

7.8.4 Time and Memory Analysis

CPU Time Analysis

I perform a comparison between the CPU time spent on optimising DAN2 and ANN, and the results are tabulated in Table 7.20. The architectures of DAN2 and ANN used are reported in the previous section. From the results, it is evident that DAN2 network with a serial structure spends a lower amount of time on optimising the network to generate comparable MSE. This demonstrates the advantage of a DAN2 network which is the faster processing time.

Table 7.20 Comparison between the CPU time of DAN2 and ANN with applications to the PSA and OILDROPLET datasets.

	ANN CPU Time (s)	DAN2 CPU Time (s)
PSA	0.461	0.0444
OILDROPLET	0.992	0.422

Memory Storage

Similar to the multi-head structure, the algorithm only requires the storage of vectors, and there is no storage of matrices. The number of vectors stored depends on the total number of layers. Overall, the memory storage is of $O(n)$.

7.8.5 Comparison between Multi-head and Serial Structure

I perform an empirical comparison of the two architectures adopting heuristic search - the multi-head and the serial architecture. The results are tabulated in Table 7.21 and Table 7.22.

From the results, it is evident that in the PSA dataset, serial structure outperforms multi-head structure in terms of both training and validation MSE. Moreover, in the serial dataset, the divergence of the Energy Consumption data is not observed. However, in the OILDROPLET dataset, the serial structure underperforms the multi-head structure, generating a higher MSE in both the training and the validation dataset.

I postulate that the serial structure is more suitable to the PSA dataset because there is stronger inter-correlation between the outputs. The stronger correlation dictates that

Table 7.21 Comparison between DAN2 with a multi-head and serial structure with applications to the PSA dataset.

	Multi-head structure			Serial structure		
	ξ	MSE_{Train}	MSE_{Val}	ξ	MSE_{Train}	MSE_{Val}
Shared	8	6.272×10^{-3}	1.302×10^{-6}	2	2.857×10^{-8}	2.968×10^{-8}
Recovery Rate	6	8.620×10^{-5}	2.690×10^{-6}	4	2.846×10^{-8}	3.033×10^{-8}
Purity	4	2.681×10^{-4}	1.335×10^{-6}	2	1.047×10^{-7}	4.089×10^{-7}
Energy Consumption	5	7.507×10^{-4}	3.937	3	5.979×10^{-7}	4.545×10^{-6}
Average	-	1.123×10^{-3}	1.312	-	1.899×10^{-7}	1.254×10^{-6}

Table 7.22 Comparison between DAN2 with a multi-head and serial structure with applications to the OILDROPLET dataset.

	Multi-head structure			Serial structure		
	ξ	MSE_{Train}	MSE_{Val}	ξ	MSE_{Train}	MSE_{Val}
Shared	2	8.019×10^{-3}	8.047×10^{-3}	2	3.626×10^{-3}	1.173×10^{-2}
S_{Ave}	2	3.914×10^{-3}	5.305×10^{-3}	3	3.626×10^{-3}	3.987×10^{-2}
$D_{FinalAve}$	2	5.197×10^{-3}	5.529×10^{-3}	2	1.533×10^{-2}	2.613×10^{-2}
S_{Max}	2	1.578×10^{-2}	2.607×10^{-2}	3	5.255×10^{-2}	1.205×10^{-2}
D_{Ave}	2	4.015×10^{-3}	4.029×10^{-3}	2	6.686×10^{-3}	1.053×10^{-2}
Average	-	7.248×10^{-3}	1.364×10^{-2}	-	1.955×10^{-2}	2.215×10^{-2}

processing of one output may extract features related to the processing of another output. The serial structure reduces the total number of layers by processing simultaneously multiple outputs, thus achieving better performance in both time and accuracy.

The multi-head structure is more suitable to the OILDROPLET dataset because the outputs are more independent compared to the PSA dataset, and thus the parallel structure outperforms a serial structure. The serial structure may confound the processing of different outputs. This may also lead to the increase in the number of layers required for processing. Therefore, multi-head structure allows better performance in the OILDROPLET dataset under comparable CPU time.

7.9 Discussion and Future Work

7.9.1 Comparison with Other Machine Learning Models

I have also applied other regressors commonly used in machine learning literature and compare the performance with the DAN2 network. The regressors include Support Vector Regression (SVR), K-nearest neighbours regression (KNN), Ridge regression (RR) and Decision Tree regression (DT). The performance are tabulated in Table 7.23 and Table 7.24. The MSEs in the tables refer to the validation MSEs. From the tables, I observe that both the multi-head DAN2 and the serial DAN2 obtain comparable performance to the current machine learning regression models in the literature.

Table 7.23 Comparison between DAN2 and other machine learning models with applications to the PSA dataset.

	Recovery Rate	Purity	Energy Consumption	Average
SVR MSE	2.564×10^{-7}	3.394×10^{-7}	1.706×10^{-7}	$1.916 \times ^{-7}$
KNN MSE	3.482×10^{-8}	6.220×10^{-8}	1.260×10^{-7}	5.575×10^{-8}
RR MSE	2.921×10^{-8}	5.722×10^{-8}	1.185×10^{-7}	5.124×10^{-8}
DT MSE	7.394×10^{-8}	9.515×10^{-8}	$1.415] \times 10^{-7}$	7.764×10^{-8}
Multi-head DAN2 MSE	2.690×10^{-6}	1.355×10^{-6}	3.937	1.312
Serial DAN2 MSE	3.033×10^{-8}	4.089×10^{-7}	4.545×10^{-6}	1.661×10^{-6}

Table 7.24 Comparison between DAN2 and other machine learning models with applications to the OILDROPLET dataset.

	S_{Ave}	$D_{FinalAve}$	S_{Max}	D_{Ave}	Average
SVR MSE	6.652×10^{-3}	5.032×10^{-3}	1.527×10^{-2}	4.094×10^{-3}	7.762×10^{-3}
KNN MSE	2.921×10^{-3}	3.995×10^{-3}	1.294×10^{-2}	2.732×10^{-3}	5.644×10^{-3}
RR MSE	3.946×10^{-3}	6.595×10^{-3}	1.739×10^{-2}	5.060×10^{-3}	8.249×10^{-3}
DT MSE	1.208×10^{-3}	3.029×10^{-3}	8.224×10^{-3}	2.021×10^{-3}	3.621×10^{-3}
Multi-head DAN2 MSE	5.305×10^{-3}	5.529×10^{-3}	2.607×10^{-2}	4.029×10^{-3}	1.364×10^{-2}
Serial DAN2 MSE	3.987×10^{-2}	2.613×10^{-2}	1.205×10^{-2}	1.053×10^{-2}	2.215×10^{-2}

7.9.2 Data Pre-processing

The DAN2 network structure can be adapted for the task of pre-processing multi-variate regression data. The structure of the model is shown in Figure 7.5, where I have a separate

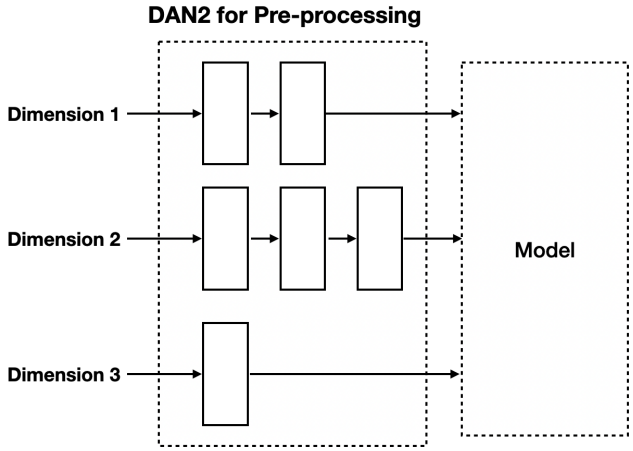


Figure 7.5 Adoption of DAN2 for data pre-processing

input head for each dimension of the input. The DAN2 network effectively acts to compress the data into a three-dimensional value (two CURLONE outputs and one CAKE output). As future work, it is possible to adopt the DAN2 network for data pre-processing, allowing a good summary of the data before inputting into other models for further analysis.

7.9.3 Multi-labelled Classification

The proposed network has been applied to regression tasks. Another important area of application is classification, especially multi-labelled classification. The multi-head DAN2 and the serial DAN2 can be used to perform this task by combining different categories of labels into pseudo-labels, which effectively converts a multi-labelled classification problem into a single-labelled case. Alternatively, in the multi-head architecture, it is possible to make each head into a classifier for each label, hence the proposed algorithm can be converted to a serch for an effective multi-labelled classifier.

7.10 Summary

In this chapter, I develop neural architectural search algorithms for multi-task learning structures adapted from DAN2. Two structures of multi-task learning are proposed, a multi-head structure and a serial structure. In each case, a grid search method and a heuristic search method are adopted to find the optimal architecture. Comparison between the two methods demonstrates that the heuristic method is capable of finding smaller structures that gives good performance. Comparison with traditional neural networks also demonstrates comparable performance.

As future work, it could be possible to apply alternative methods to optimise the network, to adopt the network for data pre-processing and to implement multi-labelled classification. Overall, the multi-task DAN2 network has great potential to be applied to the field of engineering with a efficient network of a minimal number of parameters. The effective heuristic architecture search scheme enables this application by finding an effective method to arrive at an optimal architecture through the process of "Dynamic Neural Architecture Construction".

Chapter 8

Deep Cascade Generative Model

8.1 Introduction

In this chapter, I address the problem of architectural search through the adoption of deep cascade network towards the fitting of generative models in the context of the problem of *de novo* molecular design. The research is motivated by recent advances in the application of deep cascade network to the problem of Human Activity Detection (HAD) [267] [268]. While the deep cascade network is a fully connected neural network in nature, the model proposed in this chapter is a generative model with an automatic search of its architecture.

Deep cascade network adopts a dynamic architecture search where the topology of the network changes as the training process progresses. The weights are frozen such that the extraction of more general but coarser features is stored and the vanishing gradient problem is avoided. I describe in detail the architecture in Section 8.2.1. Different to the dynamic architecture I defined in Chapter 7 that is more focused on solving problems of a smaller scale, the deep cascade generative models are well-suited to large-scale problems with key advantages over networks trained end-to-end.

The advantage of adopting a deep cascade network is that the additional layer is always close to the output layer thus it avoids the problem of vanishing gradient. It is also very adept at extracting higher-level information compared to end-to-end learning. Moreover, it is computationally efficient, and requires less memory and computational resources [155].

I would like to focus on the *de novo* molecular design problem because it is currently a challenging large-scale problem in the chemical and drug industry. The context is defined by the problem of finding chemical molecules that are active towards a particular bio-target. Automated approaches for designing compounds with the desired properties *de novo* have been actively researched [269]. This area of research entails many state-of-the-art machine learning algorithms with much novelty in the approaches to decipher a chemical

structure [270]. Moreover, with developments in hardware, computational algorithms and high-throughput screening technologies have enabled the expansion of virtual libraries and high-performance algorithms to run searches on a fully developed database [269].

The challenges faced by *de novo* molecular design are two-folds: 1) how to structure the search space into regions of molecules with desired or undesired properties, and 2) how to design a systematic search strategy [271]. To effectively solve the problem, a number of machine learning methods have been developed. These methods are described in Section 8.2.2.

I believe that cascading layers in the generative model is suited to the *de novo* molecular design problem because the proposed algorithm is capable of extracting higher-level representations that, in the context of molecular design, can be functional groups or related groups of molecules that define bio-chemical properties. The network can extract bioactivity information based on latent features in the molecule that are less observant by simply viewing the molecular structure by naked eyes.

This chapter is organised as follows:

- Section 8.2 describes the current landscape in the cascade learning research and in the application of chemical design, with also an introduction of various deep neural network structures used in the research.
- Section 8.3 describes the motivation behind this research project.
- Section 8.4 describes the optimisation problem behind this research project.
- Section 8.5 summarises the deep cascade learning algorithm used to optimise architectures.
- Section 8.6 demonstrates the results of the training of an autoencoder using cascade learning versus end-to-end training.
- Section 8.7 describes the results obtained from the training of a variational autoencoder using cascade and end-to-end method.
- Section 8.8 discusses the results obtained and explores the outlook of adopting the generative model.
- Section 8.9 explores the potential application of the problem in the area of transfer learning.

8.2 Literature Review

8.2.1 Literature Review - Architecture Search

In 1990, the concept of Cascade Correlation Network was developed where the network was formed by incrementally adding neurons to some initial structure with the parameters of the neuron fixed upon addition [154]. Key properties such as generalisation of the network have also been discussed in literature [272]. Performance on evaluating functions was researched in [273]. From then on, a few structures have developed based on Cascade Correlation Network. For example, the authors of [274] proposed a network to predict stock prices with an architecture consisting of two blocks. The first block was a multi-layer perceptron network connected to the second block which was a Cascade Correlation Network. Empirical results demonstrated that the hybrid model had good accuracy of prediction. In [275], modular units instead of neurons were cascaded in an autonomous robot context.

There has also been research into modifying the structure of Cascade Correlation Network. In [276], it has been proposed to generate networks with restricted node creation and small depth by controlling connectivity. The results have demonstrated a trade-off between connectivity and performance attributes such as learning time. Cascade learning has also acted as a benchmark of comparison in architecture search problems together with pruning methods in [277][278].

The deep cascade learning method is built upon the idea of cascade learning where layers instead of neurons are cascaded [155]. The model is similar to a constructive structure termed Resource Allocation Network found in Platt's work [279]. The scalability of the deep cascade network to large-scale problems is demonstrated in [257]. A convergence study of layerwise training can be found in [280].

In the following sections, I review in detail the algorithm of Cascade-correlation learning and the deep cascade network.

Cascade-correlation Learning Architecture

The Cascade Correlation Network is essentially an architectural search algorithm that is dynamic in nature. Traditional neural network is trained with a fixed topology a priori, which limits the flexibility of the structure and often leads to excessive structural components requiring pruning to optimise. Some earliest work has been proposing a dynamically additive architecture, termed Cascade-correlation Learning Architecture [154]. Starting from some minimal network, the algorithm automatically trains and adds new hidden units one by one, until a multi-layered structure is created. The key characteristic of the network is that once a

neuron is added, its input-side weights are frozen. The neuron becomes a permanent feature detector that will not change its parameters albeit changes in input data.

The network entails three advantages: 1) it is fast in training as parameters are frozen for incremental neurons, 2) it is flexible and self-determinant in size and topology, and 3) no back-propagation of errors is required.

Deep Cascade Network

Deep Cascade Network [155] is motivated by the dynamically changing architecture of the Cascade Correlation Network [154], but is extended to solve a large data problem. Unlike Cascade Correlation Network that adds hidden neurons one by one, Deep Cascade Network adopts a bottom-up approach while performing layer additions.

The deep cascade network is trained with the following steps [155]:

- Connect first layer to be trained to the "output block": several dense layers or an average pooling layer followed by an activation function to the output layer
- Backpropagate through the first layer to learn the parameters for the first layer
- Connect second layer to the first layer in the front and to the "output block" at the end and perform forward propagation through the first layer
- Backpropagate the second layer with parameters in the first layer fixed.
- Repeat until all layers are learned. Hyperparameters can be changed in the process

The advantage to train a network in a cascade way is that it solves the vanishing gradient problem by ensuring that the added layer is always close to the output. The vanishing gradient problem occurs when the total number of layers to be trained is high [155]. During backpropagation, the multiplicative effect of the chain rule leads to larger gradients in layers near the output while initial layers have weight updates much smaller. Deep cascade network solves this problem by fixing the parameters in each layer trained such that backpropagation only occurs when the trained layer is close to the output layer.

Moreover, in [155], it has been empirically shown that in deep cascade network, better, more domain-specific representations can be learned in early layers, in comparison to end-to-end learning. Thirdly, the algorithm has computational and memory advantages over end-to-end learning [281] [282] [283], and can act as a pre-training step [155]. Lastly, the algorithm does not have static depth and remains flexible to the size of the training data [155] [283].

8.2.2 Literature Review - Chemical Design Problem

The emergence of new experimental techniques such as High-Throughput Survey (HTS), parallel synthesis, *etc.* [284] have led to the wide accessibility and the increasing size of accumulated chemical data. This amplification of available data size encourages the application of data-driven analysis tools. Typically, techniques of machine learning have been widely adopted in the chemical industry for purposes such as *de novo* molecular design [270] [285], chemical properties prediction [286] [287] [288], prediction of protein structures [289] [290], synthetic route analysis [291] [292] and retro-synthesis [293]. Commonly used algorithms in these categories include support vector machines (SVM) [294] [295], artificial neural networks (ANNs) [296] [297] [298], random forests (RF) [299] [300] and more recently, deep neural networks (DNNs) [301] [302] [303] [304]. Deep neural networks have especially gained attention due to its flexibility and ability to handle large dataset [302]. This review focuses on the *de novo* molecular design problem applied in drug discovery.

The common databases used for chemical design problems include PubChem [305], ZINC [306], ChEMBL [307], ChEBI [308], DrugBank [309], GDB [310], EPA CompTox [311] and nmrshiftdb [312]. The most widely used is the PubChem database and the ZINC database. The PubChem Database [305] contains over 253 million substances, 103 million compounds, and 268 million BioAssay records. The ZINC Database [306] contains over 750 million commercially available compounds. The DrugBank [309] is also particularly useful for drug design and contains items such as drugs, drug actions and drug targets. These databases serve as the foundation for research on molecular design problems.

The greatest challenge behind using *de novo* molecular design is the sheer size of the search space, making it a hard job to screen and filter the most relevant molecule. It has been estimated that there are 10^{60} molecules available for drug discovery [313]. To find a particular drug molecule active towards a bio-target, high-throughput screening is often used in traditional search methods to screen molecules at a scale of 10^6 per run [314]. Although a drastic improvement from traditional *in vitro* studies, this method does not satisfy the need to screen large amounts of molecules and the cost increases rapidly with larger scale [315].

Machine learning acts as a solution to this challenge by performing virtual screening of molecules before bio-assay. Common methods of searching are based on similarity-metrics to find molecules with potentially similar chemical properties [316]. In the *de novo* design problem, the aim is to screen the search space and create new molecules with desired bio-activity.

The key assumption behind the use of machine learning to perform *de novo* molecular design is the Quantitative Structural-Activity Relationship (QSAR). The most common

practice is to use SMILES strings as representations of structural data [270]. Common classes of machine learning methods used include:

- Recurrent Neural Network (RNN) [317] [318]
- Convolutional Neural Network (CNN) [88] [319]
- Reinforcement Learning (RL) [320]
- Evolutionary Algorithm [271]
- Autoencoder [88]

The most commonly used model in *de novo* design of molecules is the autoencoders [321]. This is because autoencoders are capable of converting discrete representations to continuous values, allowing gradient-based applications to run on the latent representations leading to the location of new molecules decoding into discrete molecular structures. In [88], the authors proposed the adoption of variational autoencoders (VAE) to generate new molecules based on the latent representation extracted from the autoencoder network. The inputs used are SMILES (Simplified Molecular Input Line Entry System) strings from the ZINC database. The encoder-decoder network enables transformations between the discrete representations of the molecules and the continuous latent representations. New molecules with desired properties are generated by optimising the latent space through techniques such as Bayesian optimisation. The authors in [322] used a VAE in combination with a Generative Adversarial Network (GAN) to generate new structures with desired anti-cancer properties. VAE was also used in [270] to generate molecules active towards dopamine receptor type 2.

Overall, a variety of deep neural networks have been applied in the *de novo* drug design industry. With the new functionality brought about by novel machine learning techniques, the difficulty and complexity of searching for a valid molecule has been greatly reduced.

8.2.3 Literature Review - Deep Neural Network and its Variants

In this section, I introduce the variants of DNNs I have implemented in this chapter for the problem of *de novo* drug design. A number of structures have been derived from the feedforward DNN archetype. These structures inherit similar computational procedures from the fully-connected feedforward DNNs but each is specialised to perform specific tasks. For example, Convolutional Neural Networks are often adopted in image processing due to its ability to extract higher representational details from 2-D data. Recurrent Neural Networks, on the other hand, are particularly effective in dealing with time-series data, and hence are

common in speech analysis. This section describes some of the structures that are adopted in this thesis.

Convolutional Neural Network

Convolutional Neural Network (CNN) is a deep learning neural network designed for processing structured arrays of data [323]. The theoretical operation behind one layer of convolution is shown in Figure 8.1. Convolutional layers achieve feature extraction through convoluting with the kernel, an operation that replaces matrix multiplication in feedforward DNNs. Maxpooling layers usually follow convolutional layers to achieve size reduction and further feature extraction. The operation behind maxpooling layer is shown in Figure 8.2.

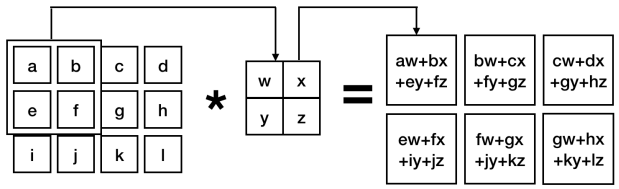


Figure 8.1 The computation of a convolutional layer

Recurrent Neural Network

Recurrent Neural Networks (RNNs) are often used to process a sequence of inputs. Compared to networks without sequence based specialisation, RNNs are capable of processing sequences of large scale with variable lengths [12]. A key feature of RNNs is parameter sharing across time, under which the model is capable of generalising to variable sequence length without specifying rules to dissect the sequence [12]. A typical structure of an RNN is shown in Figure 8.3.

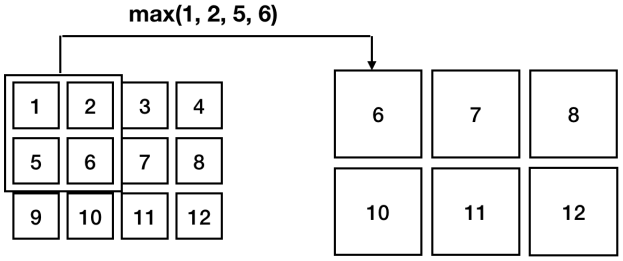


Figure 8.2 The operation of maxpooling

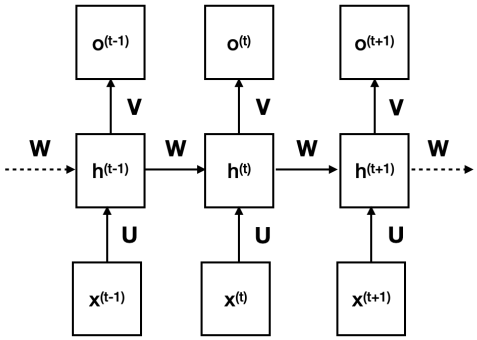


Figure 8.3 The structure of RNN.

To formulate an RNN, the recurrent property is represented:

$$s^{(t)} = f(s^{(t-1)}, \theta) \tag{8.1}$$

This is recurrent because information from a previous point in time $s^{(t-1)}$ is used to calculate the current point $s^{(t)}$. In an RNN, I define a hidden unit:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \tag{8.2}$$

where the recurrent nature of the hidden values can be observed. More specifically, an RNN performs the following set of computations in generating an output:

$$h^{(t)} = tanh(b + Wh^{(t-1)} + Ux^{(t)}) \tag{8.3}$$

$$\hat{y}^{(t)} = \text{softmax}(c + Vh^{(t)}) \tag{8.4}$$

The hidden values $h^{(t)}$ can be seen as a linear combination of the previous hidden values $h^{(t-1)}$ and the current input $x^{(t)}$ with coefficients b, W, U, transformed by an activation function $tanh$ ($tanh$ is often used for RNNs). The output is then the linear combination of $h^{(t)}$ followed by activation with softmax.

The training of an RNN involves optimising a loss function that are commonly defined to minimise the difference between $\hat{y}$ and the label y.

There are many variants of RNNs, but the widely recognised ones include long-short term memory (LSTM) and gated recurrent unit (GRU). They are collectively called gated RNNs and are empirically effective in performing sequence processing [12].

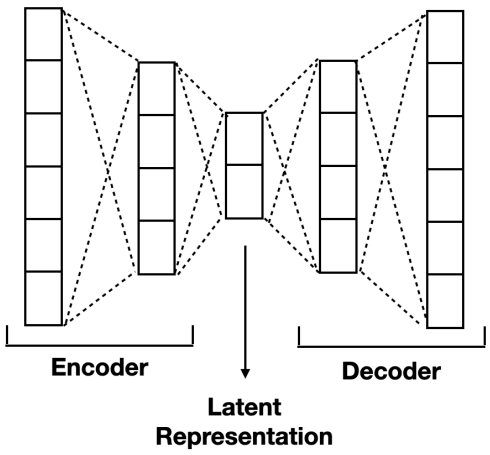


Figure 8.4 The architecture of an autoencoder consisting of an encoder and a decoder network.

In GRUs, the update equation is:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)} \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)})) \tag{8.5}$$

where U and W are coefficients, σ is the activation function, u is the "update" gate and r is the "reset" gate. The values of the gates are defined by:

$$u_i^{(t)} = \sigma(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)}) \tag{8.6}$$

$$r_i^{(t)} = \sigma(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)}) \tag{8.7}$$

Although other variants are present, the currently most optimal architectures are the LSTM and GRU [324].

Autoencoders

One typical form of deep neural networks is the autoencoder network. The network seeks to reproduce the input from its output. The network consists of two parts: an encoder function $h = f(x)$ and a decoder function $r = g(h)$, where h is the hidden layer that is often extracted as latent representation. Conceptually, I set $g(f(x)) = x$, but practically, it is often restricted to copy the inputs approximately and to copy only inputs that resemble the data [12]. Some modern autoencoders are stochastic in its mappings $p_{encoder}(h|x)$ and $p_{decoder}(x|h)$. The structure of an autoencoder is shown in Figure 8.4.

From the Figure 8.4, I can see that the structure is similar to that of a feedforward neural network except that there is a bottleneck between the encoder and the decoder. Autoencoders are commonly used for dimensionality reduction and feature learning. This is because the bottleneck layer, or the latent representation, is often set to a lower dimension than the inputs and outputs. This layer is often used in further analysis to extract properties of the input data. It is often perceived as a lower level representation of the input data.

Another common application of autoencoders is to convert discrete values into continuous ones. The discrete values will be input to the network and the latent representation is extracted with continuous values. This is a common technique used in many research [88] [325].

The training of the autoencoder is simply to minimise the loss function $\mathscr{L}(x, g(f(x))$. Common optimisation techniques work similarly in autoencoders such as gradient descent or Adam. To extract the latent representation, the input is transformed through the trained encoder and the result is stored. Variants of the autoencoder include sparse autoencoders, denoising autoencoders, autoencoders with regularisation by penalising derivatives and stochastic autoencoders [12].

A special type of autoencoders is the variational autoencoder (VAE) [326]. It provides a formulation in which the continuous representation, termed z, is interpreted as a latent variable in a probabilistic generative model. Suppose $p(z)$ is the prior distribution imposed on the continuous representation, then $q_\phi(z|X)$ is the encoding distribution and $p_\theta(X|z)$ is the decoding distribution. The training of a VAE is essentially obtaining parameters for $q_\phi(z|X)$ and $p_\theta(X|z)$. The decoder is optimised by maximising the log-likelihood $p_\theta(X|z)$. The encoder is regularised to approximate $p(z)$ by minimising Kullback-Leibler divergence.

The advantage of VAE is that it entails a stochastic generation process. This stochastic generation means, that even for the same input with the mean and standard deviations remaining the same, the actual encoding will vary on every single pass simply due to the process of sampling. This is what characterises VAE as a generative model.

8.3 Motivation, Problem Definition and Methodology

In this chapter, I aim to develop an effective model to perform the task of automated *de novo* chemical design through the dynamic architectural search of a neural network. The dynamic architectural search allows a minimal number of layers to be adopted in a neural network and I test the effectiveness of the architectural search process in the context of automated *de novo* chemical design. The problem I aim to address in this research is how to adopt a dynamic architectural search strategy to produce neural networks that solve the problem of *de novo* chemical design.

Automated *de novo* chemical design is defined as a computational process whereby structural coordinate data of the target, together with a design strategy, forms the input to a system that outputs optimal molecular structures to fit the a search problem optimally without further human intervention [327]. The aim is to produce molecular structures with desired pharmacological properties and to complement high-throughput screening [328]. The molecular structures generated from the design process are usually delivered for post-processes such as analysing synthetic tractability and drug-like properties [327].

There are three challenges related to *de novo* chemical design: 1) how to assemble candidate compounds, 2) how to evaluate potential quality, and 3) how to sample the search space [328]. In our design paradigm, the first and third challenges are achieved through an open-ended search in the continuous latent space. This reduces the tasks of selecting candidate compounds and finding sample search spaces to a computational problem of unconstrained optimisation. As the continuous latent space is divisible and contains infinitely many representations, it can effectively be claimed that the search space is very large. With the functionality of autoencoders converting molecular representations back and forth between continuous and discrete forms, it can be safely claimed that a large number of potential candidates are assembled with a well-defined search space. The second challenge is currently controlled by commercially available packages to ensure that the output from the algorithm is valid. It has been widely accepted that automatic *de novo* design may generate invalid molecules and only few transforms into a prospective new lead series with desired properties for further analysis [328].

Machine learning techniques such as deep learning have been suitable to solve this problem because the total number of molecules to search for is combinatorially large. Given the different types of elements and the linkages, it is difficult to perform exhaustive search for all structural combinations [328]. Machine learning techniques are able to solve this problem by learning the correlations between the structural information and the properties of the molecules, thus generating an informed search direction and a few molecules from the guided search. Therefore, algorithm-driven *de novo* design often involves machine learning processes.

The key obstacle of designing an algorithm for chemical design is the stochasticity inherent to the model that produces novel molecules [328]. A direct ramification is the generation of different molecules in different runs of the algorithm. To prevent such cases, constraining the search to certain regions in the latent space has been popular in literature. There are two types of constraints: positive design restricts the search to certain regions more likely to produce drug-like molecules and negative design forbids regions where molecules in those regions have unwanted properties. However, the stochasticity in the model can

be treated as a positive trait that introduces flexibility into the model [88], allowing the generation of valid molecules that otherwise might fall into "dead region" in the search space in a deterministic model. Moreover, to maximise flexibility and functionality, unconstrained optimisation is performed.

The database I use is the ZINC database. The ZINC database is selected because it is open-source and contains a large amount of commercially available molecules to train and test our algorithm. As the database is large, I employ the High Performance Computing Services to perform the optimisation of the algorithm.

The representation of molecules I adopt in this research is the SMILES representation. Although there are other representations available such as the chemical fingerprint [329], convolutional neural networks on graphs [330], similar graph-convolutions [331], and Coulomb matrices [332], I adopt SMILES strings because it is easily and readily convertible to a molecule. There are several advantages of adopting SMILES representation, especially when applied to *de novo* chemical design. One key advantage of adopting SMILES in comparison to a molecular graph is its representations of important molecular features, such as the presence of cycles, the cis-/trans- isomerism, and the presence of sp^2 and sp^3 atoms [333]. Secondly, with a SMILES string representation rather than a 2-D or 3-D representation, the problem of analysing and generating new molecules can be undertaken effectively by 1D-convolutional layers and GRUs respectively. The complication brought about by a high dimensional input is avoided. Thirdly, the SMILES representation allows easy breakdown of molecules into its atomic or group components that entail key structural information. The breakdown allows one-hot representations of the molecules which are formations easily analysed by machine learning techniques. Although there are key advantages of molecular graphs as well [333], the SMILES representations are a simple and elegant form that I adopt in my research.

The key machine learning algorithm adopted is a set of autoencoders. The distinct feature of the algorithm is the dynamic addition of encoder layers to allow the extraction of more general features. I initially experiment with autoenocders and then with the generative variational autoencoders.

8.4 The Problem of Optimisation

The training of the autoencoders can be formulated as an optimisation process, where the objective function is defined such that the network reconstructs the original input. For a

two-layer network, the calculative process is summarised in Equation 8.8 and 8.9 [334]:

$$h = f_e(x) = s_e(W_e x + b_e) \tag{8.8}$$

$$x_r = f_d(x) = s_d(W_d h + b_d) \tag{8.9}$$

where $x \in \mathbb{R}^d$, $f_e : \mathbb{R}^d \to \mathbb{R}^h$ and $f_d : \mathbb{R}^h \to \mathbb{R}^d$. W and b are the weights corresponding to the neural network architecture. h is the latent representation and x_r is the reconstructed input. Common practices is to keep x_r as a copy or a noise added copy to the input x. The training of autoencoders is thus an optimisation over the reconstruction error:

$$\mathscr{J}_{AE}(\Theta) = \sum_{x \in \mathscr{D}} \mathscr{L}(x, x_r) \tag{8.10}$$

where $\mathscr{L}$ is the loss function of reconstruction error and $\mathscr{D}$ is the training dataset.

More specifically, the training in a dynamic manner with addition of layers after weights are frozen in the previously trained layers can be formulated as optimisation processes. A typical analysis of layer-wise training in autoencoders is summarised below [334].

The loss function for a single layer autoencoder is defined as:

$$\mathscr{J}_{GAE}(\Theta) = \sum_{x \in \mathscr{D}} \mathbb{E}_{x_c \in Q(x_c|x)} \left[\mathscr{L}(x, f_d \circ f_e(x_c)) \right] + \lambda \mathscr{R}(\Theta) \tag{8.11}$$

where $\lambda \geq 0$, and $\mathscr{R}(\Theta)$ is an arbitrary regularisation function over parameters.

In [334], two methods are proposed to train the autoencoder layerwise:

- Greedy layerwise training: define $\mathscr{J}(\cdot)$ as the single layer autoencoder objective and Θ^n the parameters for layer n. Suppose there is a two-layer autoencoder. The training process optimises $\mathscr{J}(\Theta_1)$ first as a single layer using original input. Then the output is treated as latent representation to input into the second layer. The second layer is trained with objective $\mathscr{J}(\Theta_2)$.
- Training with overall objective: define the overall objective $\sum_{x \in \mathscr{D}} \mathscr{L}(x, x_r)$. All layers are trained based on this overall objective function.

Deep cascade learning is in essence layer-wise training of the autoencoders. In this research, I would like to apply layer-wise training to the autoencoders following the second scheme, *i.e.* optimising layers with respect to the overall objective. This method is selected because I would like to oversee the performance of the whole architecture instead of focusing on a particular layer. I believe with the selected optimisation scheme, it is possible to arrive at a better solution through the layer-wise optimisation process.

8.5 Algorithm and Optimisation

8.5.1 Autoencoders for *De Novo* Chemical Design

A set of autoencoders is adopted in this research for the problem of *de novo* chemical design. The input to the autoencoders consists of structural representations of the molecules converted to one-hot representation. The structural representations adopt the SMILES system to encode each input into short ASCII strings. It effectively converts 3-D representations of the molecules to 1-D. Atoms, bonds, rings, aromaticity, branching, stereo-chemistry and isotopes are all represented with ASCII symbols. In my model, I recognise 120 sets of ASCII symbols as a breakdown of each molecular input, registering inputs into one-hot representations of dimension 120.

The encoder and the decoder are composed of simple standard neural layers. The encoder consists of 1D-convolutional layers followed by a set of fully connected layers. Optionally, the convolutional layers can be followed by batch-normalisation layers. Similarly, the few fully-connected layers can be separated by dropout layers and batch-normalisation layers. The decoder layers consist of GRU layers to output sequences of molecular representations. It has been mentioned in the original paper [88] that due to the character-by-character nature of SMILES representation and the fragility of its internal syntax, the decoder may produce invalid syntax. The package RDKit has been adopted to screen out invalid molecules, allowing maximised flexibility with the autoencoder network.

After the implementation of autoencoders, I also construct a variational autoencoder to perform the task of *de novo* drug design. This is because the variational autoencoder is a generative model that is capable to introducing flexibility to the results of the network with less rigid and more innovative solutions. Both the autoencoder and the variational autoenocder are similarly analysed and the results are compared to that obtained from end-to-end training.

8.5.2 Deep Cascade Learning on Autoencoders

In the original setting of deep cascade network [155], the optimised network is the convolutional network. This allows simple stacking of layers below each other during the training process to enable layer-wise training. The layer-wise training process becomes more difficult in the case of autoencoders. This is because the model consists of an encoder and a decoder where the output is only directly connected to the decoder. I cannot train an encoder separately from a decoder. Therefore, the layers of the encoder are cascaded in a layer-wise manner whereas the layers of the decoder act as a constant transformation between the latent

representation and the output in my proposed algorithm. The process of training is illustrated in Figure 8.5.

I employ an optimisation scheme that is heuristic in nature. I sequentially add one encoder block to the network until a satisfactory result based on loss criteria is obtained. The optimisation scheme is implemented with the following steps:

(a) Define the model input layer

(b) Add one encoder block (a convolutional layer and potentially a batch-normalisation layer)

(c) Connect to fully connected layers in the encoder

(d) Connect to fully connected layers in the decoder

(e) Add all decoder blocks (GRUs)

(f) Train the model with respect to the output

(g) Store the weights to the added encoder block and remove the fully connected layers and the GRUs in the decoder

(h) Repeat from Step (b), freezing network parameters that have been trained, until a satisfactory number of encoder blocks (defined by the training or validation error) are added.

I have implemented fully-connected layers in all the training process because they act as bridges between the encoder and the decoder. The main feature extraction occurs at the encoder blocks thus only encoder block layers are cascaded.

I cascade new layers from outside to inside. This is to ensure that initially trained layers are closer to the output hence they extract coarser features from data. The later layers capture the finer details of the data as they are farther away from the input and output layers. As I freeze the weights of the trained layers, the structure is able to store essential information regarding coarser and finer feature characteristics of the data. The weights of the fully-connected layers are not frozen, since I expect these layers to act like bridges and will buffer any change made in the encoder and decoder blocks.

The advantage of the method is that it is simple to implement, as the process involves a simple heuristic to optimise the network in a layer-wise manner. Moreover, it employs a systematic search scheme that allows efficient layer-wise training of the encoder and decoder simultaneously, allowing the production of a symmetric autoencoder network. Thirdly, it

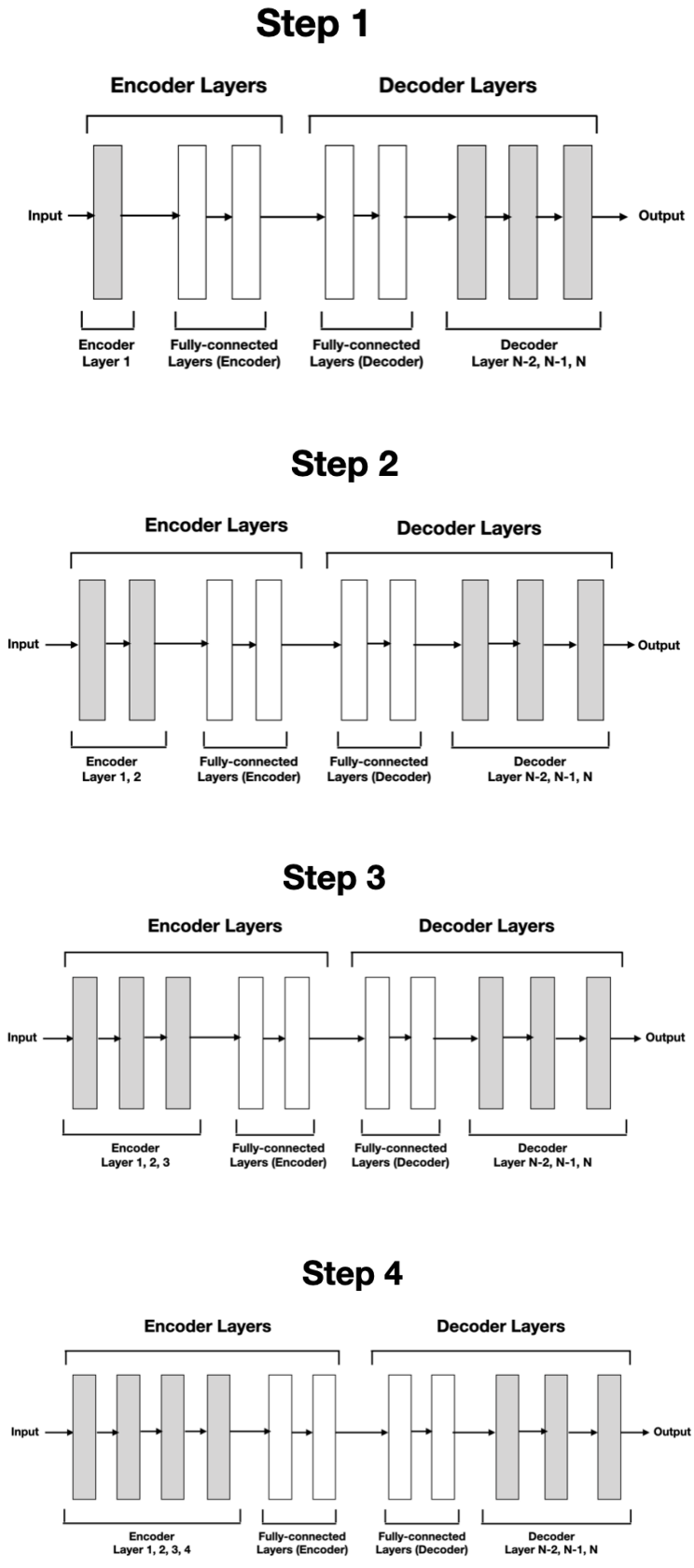


Figure 8.5 The process of cascading autoencoders.

inherits all the advantages of a deep cascade network, where weights are frozen in each layer thus better capture the general features of the network.

A limitation would be that the autoencoder trained under this scheme will have more encoder layers than decoder layers.A solution to this would be to adopt the heuristic scheme as in Chapter 7, where I iterate on the addition of encoder blocks and decoders blocks in an alternating sequence. However, empirically this leads to unnecessarily deep encoder layers or decoder layers. Therefore, the method is not adopted in this optimisation setting.

Another limitation is that it is less effective in solving the vanishing gradient problem compared to the cascading process in a simple feedforward network. Although it employs layer-wise training, the training layers are away from the outputs by the length of the decoder, with each consecutive layer farther away from the input and output. However, due to the freezing of weights, the vanishing gradient problem is less prominent compared to end-to-end training.

8.5.3 Flowchart and Pseudocode

To further illustrate the logic behind the autoencoder or variational autoencoder used in the chemical problem, a flowchart is drawn in Figure 8.6. The key functionality of the algorithm is to perform property predictions based on the latent representation obtained from the training of autoencoders or variational autoencoders. Two models work independently in this flowchart. The autoencoder finds the latent representation and the property predictor is a separate multi-layered perceptron network that acts on the latent representation. The flow terminates when the predicted properties of the concerned molecules are obtained.

The pseudocode for the end-to-end optimisation process is shown in Algorithm 11. The pseudocode for a cascaded network is shown in Algorithm 12.

8.6 Training Results of Autoencoders

I first investigate the performance of cascade learning using the the slightly modified version of the optimal structure suggested in [88]. The details of the structure is enlisted in Table 8.1. I adopt the structure from literature as a starting point because this will lead to an optimal point where the most effective model adopting cascade learning is compared to end-to-end training. However, I acknowledge that the model hyperparameters from the literature is found through random search, which may not be the most optimal. I perform hyperparamters tuning after a preliminary comparison of the two models. The original model is obtained by performing random search over hyperparameters. I attempt to perform random searches on

Algorithm 11 Autoencoders with end-to-end learning.

function ENCODER_LAYERS(parameters)
    Add Input Layer
    for <number of encoder_blocks> **do**
        Add Conv1D Layer
        Add BatchNormalisation Layer
    end for
    return Encoder_Model
end function

function DECODER_BLOCK(parameters)
    Add encoder Dense Layers
    Add decoder Dense Layers
    Add Flatten Layer
    Add RepeatVector Layer
    for <number of GRU_layers> **do**
        Add GRU Layer
    end for
    return Decoder_Model
end function

function END-TO-END(parameters, encoder_blocks, decoder_block, X_train, Y_train, X_test, Y_test, epochs, loss_function, batch_size)
    Initialise: optimiser and learning rate
    Compile Encoder_Model and Decoder_Model
    Fit Encoder_Model and Decoder_Model using X_train, Y_train, X_test, Y_test, epochs, batch_size
    Store optimisation history
end function

Algorithm 12 Autoencoders from Cascade Learning

```
function CASCADE_LEARNING(parameters, encoder_blocks, decoder_block, X_train,
Y_train, X_test, Y_test, epochs, loss_function, batch_size)
    for <i in number of encoder_blocks> do
        Append first i encoder blocks to Current_Model
        Append decoder block to Current_Model
        Define Sequential_Model
        if <i==0> then
            Add layers in Current_Model to Sequential_Model
        else
            Remove decoder block
            Set encoder block to be not trainable
            Add newly added encoder block to the Sequential_Model
            Add decoder block
        end if

        Initialise: optimiser and learning rate
        Compile Sequential_Model using loss_function
        Fit Sequential_Model using X_train, Y_train, X_test, Y_test, epochs, batch_size
        Store optimisation history
    end for
end function
```

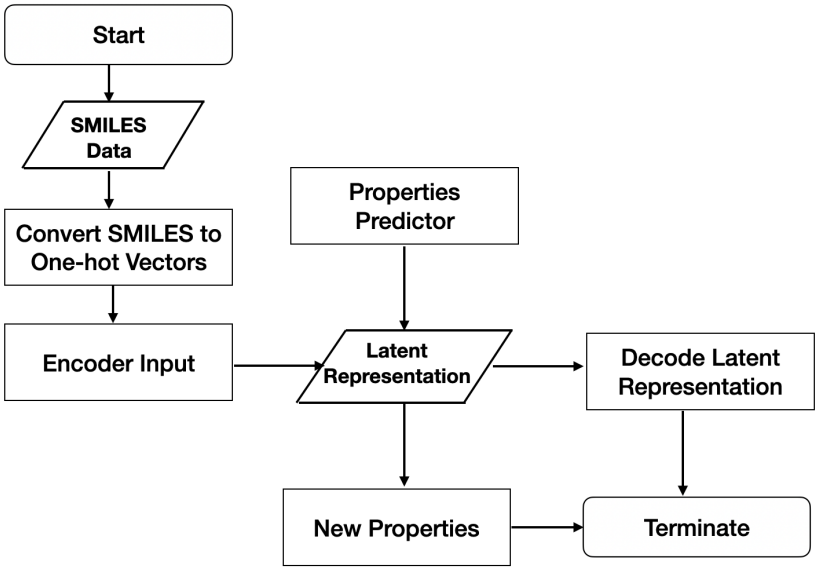


Figure 8.6 Flowchart of the logic behind autoencoders in the problem of *de novo* chemical design.

the hyperparamters as well and the parameters are listed in Table 8.2. The total number of epochs is set to the value of 70 which is found in the original model.

The dataset I use contains 250,000 chemical formulas described by SMILES representation and their corresponding properties including the water-octanol partition coefficient (logP), the synthetic accessibility score (SAS) and the Quantitative Estimation of Drug-likeness (QED), which ranges in values between 0 and 1, with higher values indicating that the molecule is more drug-like.

I perform two sets of training: 1) end-to-end training, and 2) cascade learning. This is to gauge the effectiveness of the cascade learning on this particular problem. The objective is to compare the values of weights obtained by cascade learning versus the values obtained by end-to-end learning.

8.6.1 Weights Comparison

I perform a comparison of the values of weights on the two models. I illustrate the comparison in Figure 8.7.

There are two observations I can generate from Figure 8.7: 1) the two models contain weights that are very differently distributed; the end-to-end weights have a higher kurtosis (kurtosis = 0.549, platykurtic) whereas the cascaded weights have a lower kurtosis (kurtosis

Table 8.1 Structure of a simple and small network adopting cascade learning.

Layer	Output Shape	#Parameters
Input	(120, 25)	0
Convolution1D	(113, 8)	2248
BatchNormalisation	(113, 8)	32
Convolution1D	(106, 8)	520
BatchNormalisation	(106, 8)	32
Convolution1D	(99, 8)	520
BatchNormalisation	(99, 8)	32
Convolution1D	(92, 8)	520
BatchNormalisation	(92, 8)	32
Flatten	(736,)	0
Dense	(100,)	73700
Dropout	(100,)	0
BatchNormalisation	(100,)	400
Dense	(100,)	10100
Dropout	(100,)	0
BatchNormalisation	(100,)	400
Dense	(100,)	300
Dropout	(100,)	0
BatchNormalisation	(100,)	400
Dense	(100,)	10100
Dropout	(100,)	0
BatchNormalisation	(100,)	400
RepeatVector	(120,100)	0
GRU	(120, 35)	14385
GRU	(120, 35)	7560
GRU	(120, 35)	7560
GRU	(120, 35)	7560

Table 8.2 Hyperparameters of a simple and small network adopting cascade learning.

Parameters	Value
Epochs	70
Dropout rate	0.0828
Batch size	126
Encoder layers	4
Decoder layers	4
Middle layers	2

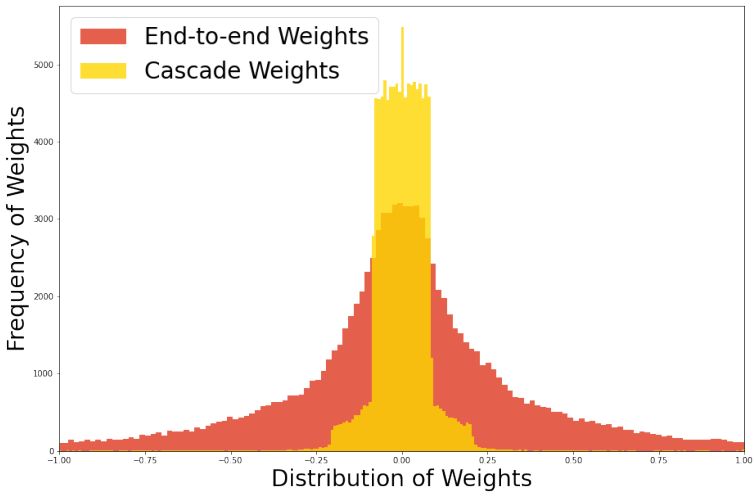


Figure 8.7 The distribution of weight values between the trained weights of the end-to-end model and the cascade model in an autoencoder.

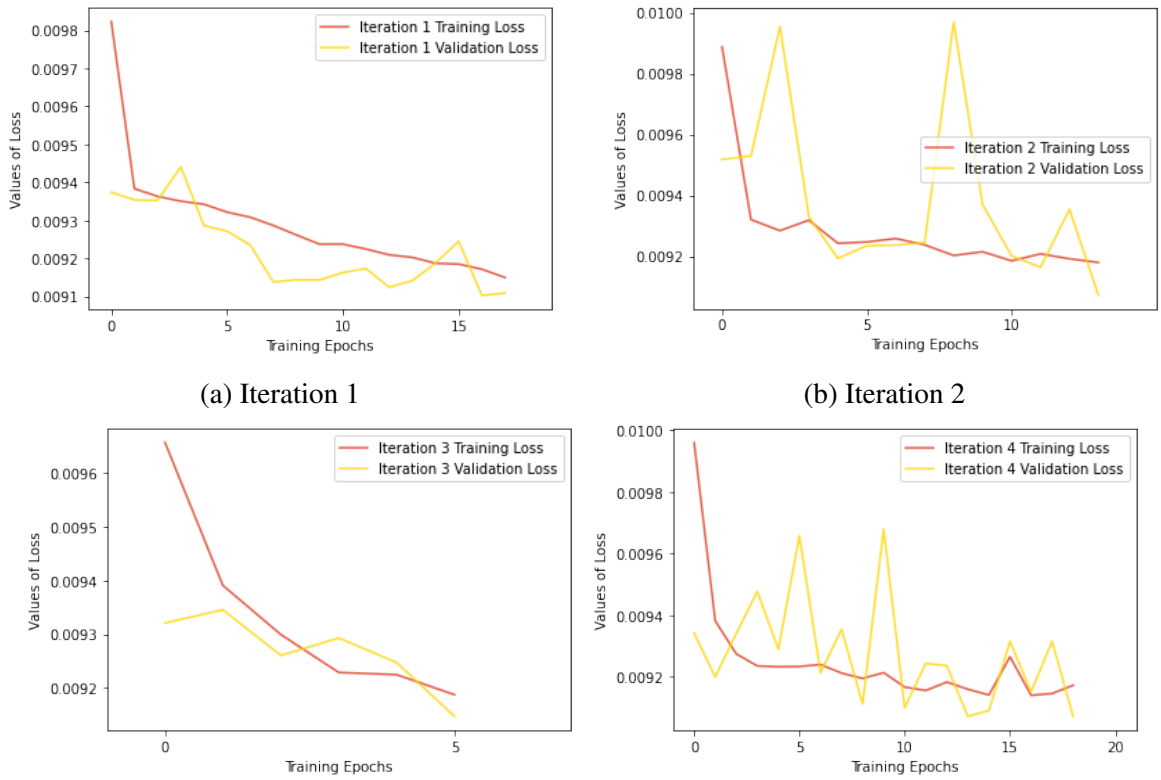
=-3.0, highly platykurtic); and 2) the end-to-end weights (skewness = -0.385) are more skewed to the left compared to the cascaded weights (skewness = 0.0).

The reason I look at the distribution of the weights is two-fold: 1) I would like to observe the effect of cascade learning on the values of weights; as the weights are not identically distributed, it is reasonable to conclude that there is a strong effect of cascade learning on the weights obtained, and 2) the weights of cascade learning is more clustered around the mean, indicating that the values are more similar; this is possibly due to the fact that trained layers are always close to the output, hence there is not the problem of diminishing gradient.

8.6.2 Training Performance

I also investigate the effects on training and validation losses. Within models adopting cascade learning, I compare the effects of the minimal model and the layers added to the model. The learning curve is shown in Figure 8.8. For example, curve labelled with Iteration 1 represents the minimal model with one Conv1D layer and one Batchnormalisation layer whereas Iteration 2 represents the model with 2 Conv1D layers and 2 Batchnormalisation layers alternating. The number of epochs in each iteration differs since I choose to save the "best" model with the lowest validation losses in the first 20 epochs. I only present the training performance up until the best model since this is the model that is fed to the next iteration.

From Figure 8.8, I observe that overall, the training losses decrease with increasing number of epochs. The validation losses trace the decrease in training losses but fluctuate



(a) Iteration 1

(b) Iteration 2

(c) Iteration 3

(d) Iteration 4

Figure 8.8 The learning curve for each step of optimisation in the cascade learning in an autoencoder.

up and down throughout the process. This is expected because the model will demonstrate inaccuracies after being generalised to a different dataset. The overall decreasing trend of the validation losses demonstrate the effectiveness of the training process.

8.6.3 Optimisation Performance

Previous analysis has based the cascade learning model on the optimal model produced in [88]. However, there is no guarantee that such a model is the optimal. Therefore, I perform a search of optimal structure by adopting a heuristic search scheme of architectural parameters combined with random search on other hyperparameters. The heuristic search allows the addition of layers until a sufficiently small validation error is obtained. To observe the effect of optimising the network with regard to the addition of layers, I obtain the optimisation results in the form of reconstruction MSE layerwise. Since the shape of the input is 3 dimensional (consisting of batch size each with a two dimensional one-hot encoding), I define the MSE in this case to be the mean of the reconstruction error for all entries in the

one-hot encoding. From Table 8.3, it can be observed that the cascade model outperforms the end-to-end model in both training MSE and validation MSE, indicating the effectiveness of architecture search to find the most optimal model.

Table 8.3 The reconstruction error of the autoenocder trained through cascade learning *vs* end-to-end learning.

Layer	Training MSE	Validation MSE
1	0.00915	0.00911
2	0.00918	0.00907
3	0.00919	0.00915
4	0.00917	0.00907
End-to-end	0.00919	0.00909

8.6.4 Property Prediction

The effectiveness of each model is evaluated by making property predictions in the latent space. A multi-layered perceptron network is employed to act on the latent space to perform property prediction. The hyperparameters adopted for both models are enlisted in Table 8.4. For activation function, I have adopted *tanh* as this is the original activation function used in [88], which is the literature my research is based on.

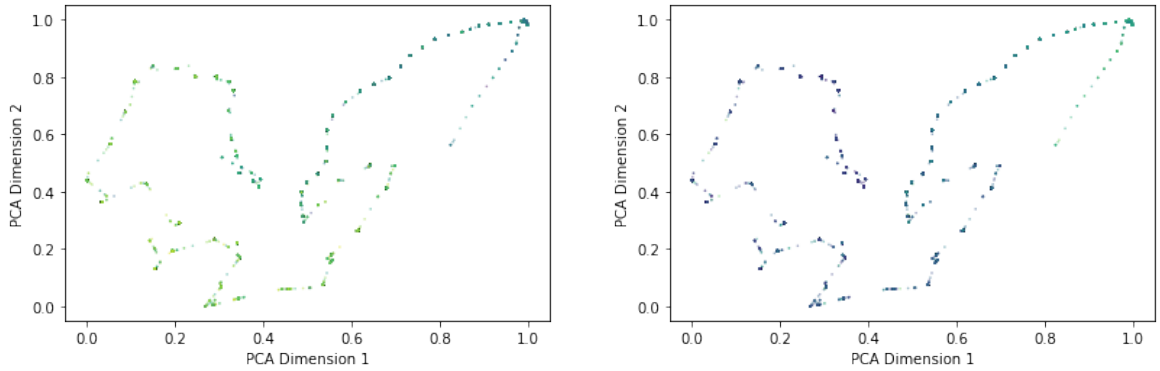
Table 8.4 Hyperparameters of the network solving the problem of property prediction.

Parameters	Value
Training Epochs	20
Batch Size	126
Network Depth	3
Network Width	36
Activation Function	Tanh
Objective Function	Mean Squared Error
Batch-normalisation Layers	Included

Adopting the latent space obtained from the cascade model, I obtain a mean squared error of 0.91065 in the testing dataset. This draws comparison to the latent space obtained from the end-to-end model which has a mean squared error of 0.93953. The cascade model is outperforming the end-to-end model in this setting.

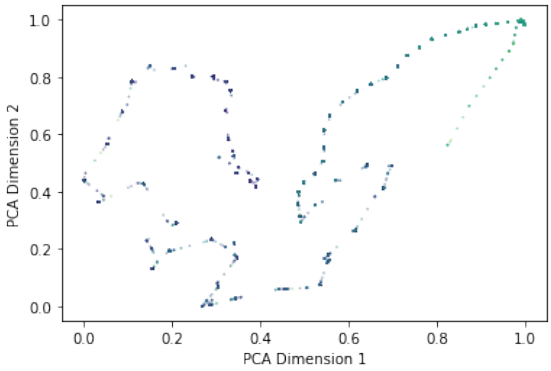
I also perform principle component anlaysis (PCA) on the latent space, compressing the latent space into 2 dimensions. The compressed results for latent space from the cascaded

model are demonstrated in Figure 8.9. The compressed results for latent space from the end-to-end model are demonstrated in Figure 8.10. It can be observed from Figure 8.9 and Figure 8.10 that the latent space from cascade learning takes a curved line shape with some discontinuity due to the discrete nature of the input molecules. That is, only a subset of the molecules are considered in the training dataset thus generating a discontinuous graph. The highly different shape of the compressed latent space demonstrate the effect of cascade learning in generating a highly different model from end-to-end training.



(a) PCA of logP

(b) PCA of QED

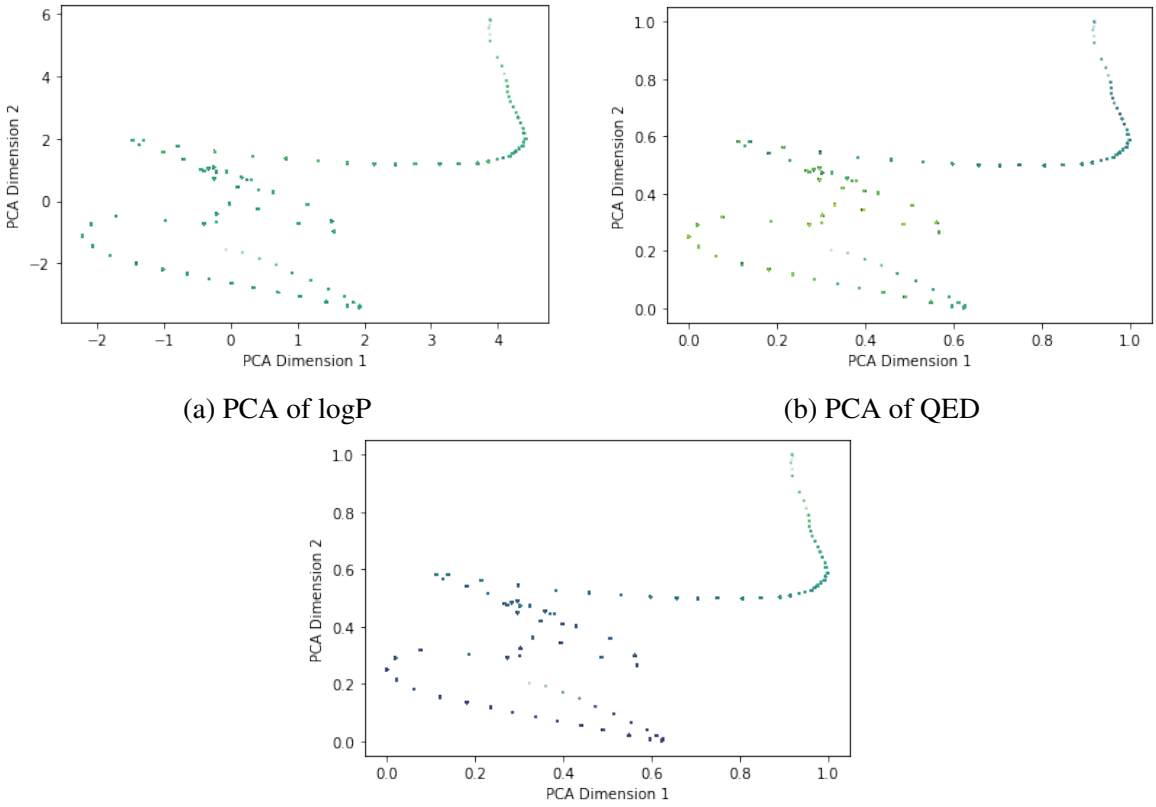


(c) PCA of SAS

Figure 8.9 The principle component analysis of the embedding obtained form cascade training the autoencoder model.

8.6.5 Comparison of Filters

I compare the filter values obtained for each convolutional layer. The convolutional layer is 1D hence the filter obtained is 1D as well. The filters have a shape of 1×8 since this is defined in the hyperparameters. The comparison of filters obtained from end-to-end learning versus the deep cascade learning is tabulated in Figure 8.11. In Figure 8.11, the value of



(a) PCA of logP

(b) PCA of QED

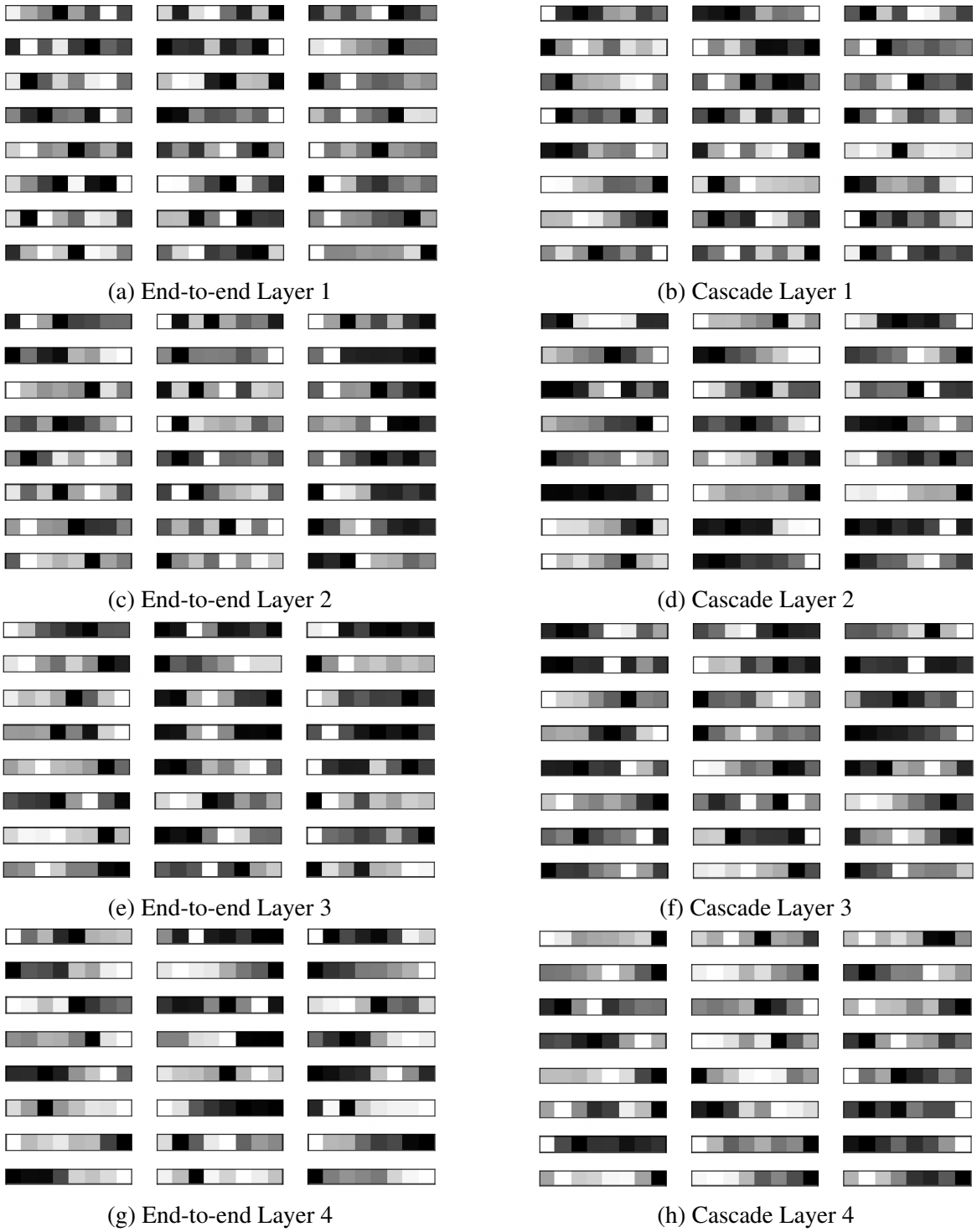
(c) PCA of SAS

Figure 8.10 The principle component analysis of the embedding obtained form end-to-end training the autoencoder model.

the filter used in the CNN is converted to grey-scale images to better visualise the value difference.

From Figure 8.11, I observe that the filters obtained from the two models are very different, indicating the effects of cascade learning on the model parameters. The filters of the first layer are more similar since the weights are trained in a similar environment. The filters of later layers are more different due to the freezing of weights in the previous layers. This almost results in completely opposite filters in the last layer. This indicates that the optimisers arrive at different local minima in the process of optimisation.

Overall, it can be claimed that the two optimisation processes generate very different local minima. With the optimisation performance of the cascade model outperforming the end-to-end model, the cascade model is capable of reaching a lower minimum compared to the end-to-end model.



(a) End-to-end Layer 1

(b) Cascade Layer 1

(c) End-to-end Layer 2

(d) Cascade Layer 2

(e) End-to-end Layer 3

(f) Cascade Layer 3

(g) End-to-end Layer 4

(h) Cascade Layer 4

Figure 8.11 The images of filter values for each layer in the end-to-end and the cascade learning model adopting an autoencoder.

8.7 Implementation of Variational Autoencoder

I explore the effects of generative models used for the problem of *de novo* chemical design. A typical model is the variaitonal autoencoder. The theory behind a variational autoencoder has been discussed in Section 8.2. Since the variational autoencoder entails a probabilistic generative process, the advantage of using a variational autoencoder in this problem is that newly generated molecules have some degree of flexibility compared to molecules generated from only an autoencoder. This means that more freedom is allowed in the generation of new molecules.

8.7.1 Weights Comparison

I perform a comparison of the values of the weights in the end-to-end model against that in the cascade model. The comparison is illustrated in Figure 8.12.

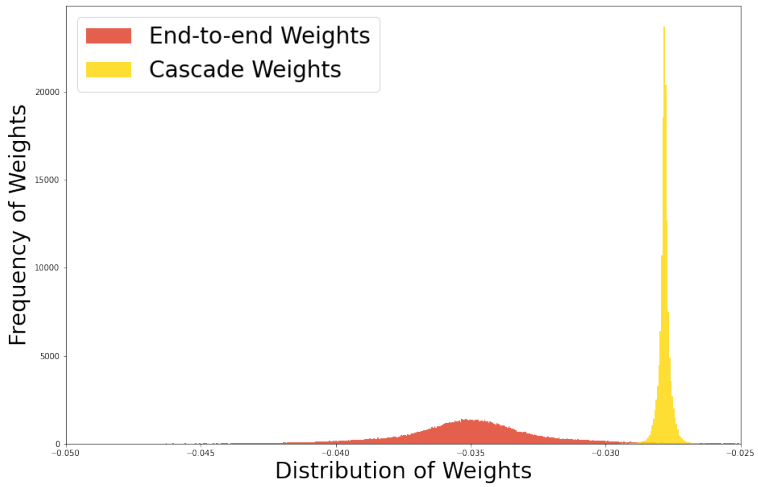
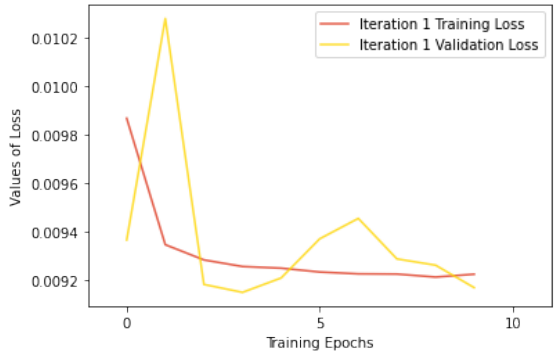


Figure 8.12 The distribution of weight values between the trained weights of the end-to-end model and the cascade model in a variational autoencoder.

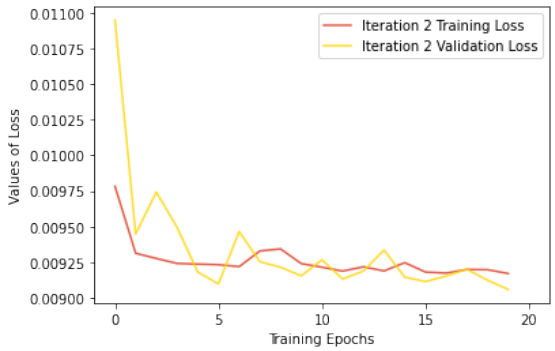
From Figure 8.12, it can be observed that the two models generate very different weights: the end-to-end weights have a kurtosis of 0.927 whereas the cascade weights have a kurtosis of 3.003; the end-to-end weights have a skewness of -0.416 whereas the cascade weights have a skewness of -0.223. This demonstrates that the two models arrive at different local optima and the cascade learning model tends to have more clustered weights, indicating that the weights are not diminishing across layers. This is the effect of training and freezing each layer during the optimisation process, such that the weights calculated are always close to the output, and are taking values closer to each other.

8.7.2 Training Performance

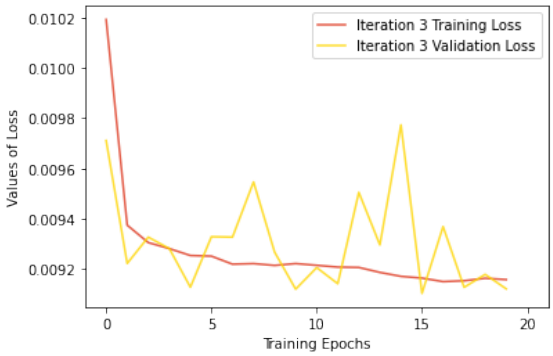
I plot the learning curve of the variational autoencoder throughout the process of adding layers in cascade learning (delimited by "Iterations"). The number of epochs in each iteration differs since I choose to save the "best" model with the lowest validation losses in the first 20 epochs. I only present the training performance up until the best model since this is the model that is fed to the next iteration. In iteration 1 and 2 where there are 1 and 2 convolutional layers respectively, the training and validation losses trace each other. In Iteration 3 and 4 where there are 3 and 4 convolutional layers respectively, there training losses decrease gradually whereas the validation losses fluctuate up and down. This is attributed to the generative element in the network that brings flexibility and uncertainty to the results obtained. Moreover, the fluctuations observed in Iteration 4 is largely due to the smaller y-coordinate scale being used. The fluctuations are indeed much smaller in raw values. The decrease of training loss is less obvious in Iteration 4 because the model is already very well-trained with a very low training loss overall.



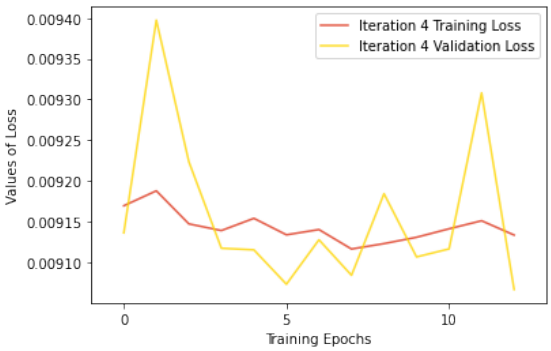
(a) Iteration 1



(b) Iteration 2



(c) Iteration 3



(d) Iteration 4

Figure 8.13 The learning curve for each step of optimisation in the cascade learning in a variational autoencoder.

8.7.3 Optimisation Performance

Since I am performing cascade learning on the encoder, the encoder is divided into 4 levels, with each level consisting of a Convolutional Layer and a Batchnormalisation layer. The reconstruction error is compared for different levels of network produced from cascade learning. The results are demonstrated in Table 8.5. The definition of the MSE is the same as outlined in Section 8.6.3.

Table 8.5 The reconstruction error of the variational autoencoder trained through cascade learning *vs* end-to-end learning.

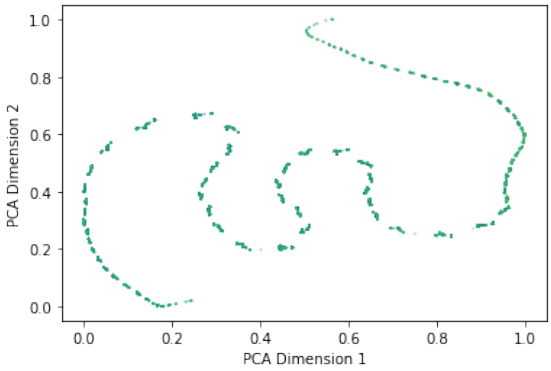
Level	Training MSE	Validation MSE
1	0.00922	0.00917
2	0.00917	0.00906
3	0.00916	0.00912
4	0.00913	0.00907
End-to-end	0.00952	0.00919

From Table 8.5, it can be observed that the training errors are decreasing with levels added through cascade learning. This demonstrates that the addition of layers improves model performance. However, the values of the validation losses fluctuate up and down throughout the cascade learning process, indicating the lack of generalisation of the model developed.

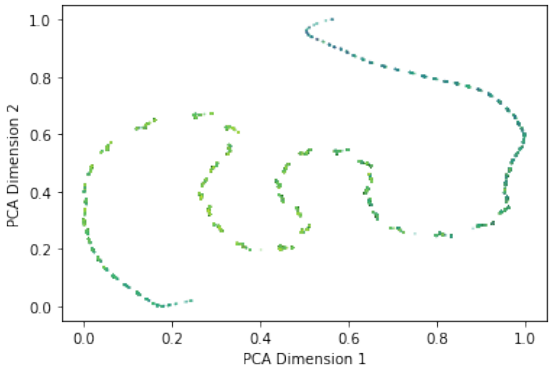
Comparing to the end-to-end training, the cascade network produces a model with lower training MSE after 4 iterations of cascade learning. Moreover, validation errors of iteration 2 and 3 outperform that of the the end-to-end model.

8.7.4 Principle Component Analysis

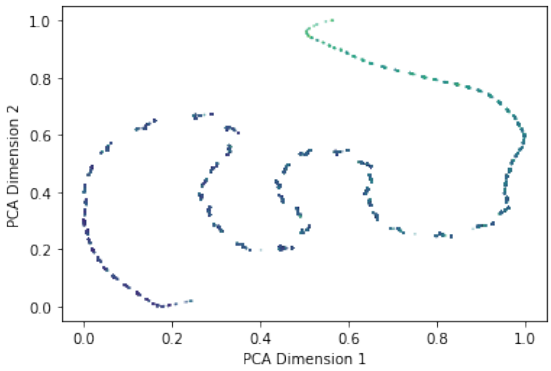
I employ Principle Component Analysis (PCA) of the latent embedding to observe the effect of the generative process on the autoencoder. The PCA technique converts high-dimensional data into lower-dimensions such that it can be visualised. I plot the embedding transformed by PCA and the results for the cascade training and the end-to-end training are shown in Figure 8.14 and Figure 8.15 respectively. In either case, the latent space reduces to a two-dimensional matrix is a curved line. The continuity of the latent space representation demonstrates that the model encodes the molecular representation into a small defined latent space. The difference in the latent space representation demonstrates the effectiveness of the cascade learning, generating a highly different model and a different latent representation.



(a) PCA of logP

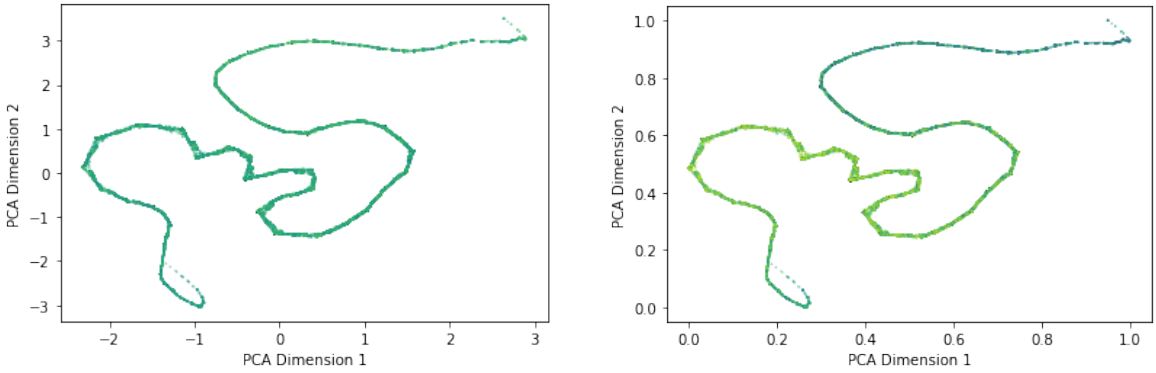


(b) PCA of QED



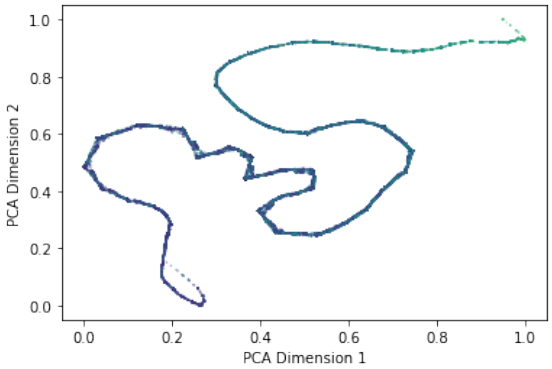
(c) PCA of SAS

Figure 8.14 The principle component analysis of the embedding obtained from the cascade training of the variational autoencoder model.



(a) PCA of logP

(b) PCA of QED



(c) PCA of SAS

Figure 8.15 The principle component analysis of the embedding obtained from the end-to-end training of the variational autoencoder model.

8.7.5 Property Prediction

Similarly, I employ a multi-layered perceptron on the latent space to perform the property prediction task. The same set of hyperparameters are adopted for the cascade model and the end-to-end model, as enlisted in Table 8.4. Performing property prediction on the latent space obtained from the cascade model, I obtain a mean squared error of 0.67003. Performing the same task on the latent space obtained from the end-to-end model, I obtain a mean squared error of 0.88205. The cascade model is outperforming the end-to-end model in the property prediction task.

8.7.6 Comparison of Filters

I compare the filters obtained from each model, as demonstrated in Figure 8.16. From Figure 8.16, it can be observed that the cascade model and the end-to-end model arrive at very different filters, which demonstrates the impact of applying cascade learning on the problem. It can also be observed that the filters for the first few layers are similar for the cascade model and the end-to-end model. However, as optimisation progresses, the filters at the last few layers are becoming more different. This is due to the accumulated difference obtained from the freezing of the initial layers which leads to escalated differences in later stages.

Overall, I have demonstrated that the two models arrive at very different optimised results, indicating the great potential of cascade model to be used as an alternative optimisation method to generate results for further analysis.

8.8 Discussion

8.8.1 Model Validity and Quality

The autoencoder and the VAE model described are capable of producing results with low reconstruction errors and low errors on the property prediction task. However, the performance is not robust and is unstable when conditions change. In the VAE, for example, the quality of results generated largely depends on the generative process. The uncertainty entailed within the VAE model implies varied performance over different runs and different initialisations. Therefore, it can be claimed that the model is valid due to the production of positive results, but it is difficult to assert that the model has good robustness. A slight change in initialisation can lead to very different results.

I believe that the cascaded model has better performance based on both the reconstruction error and the error on the property prediction task. I believe this is due to the capability of

the cascaded model to capture coarser features from the training data and the avoidance of the vanishing gradient problem. However, it is noteworthy that hyperparameters tuning and randomised initiation play important roles in achieving such results.

8.8.2 Model Variants

The model structure is solely based on the literature from [88] where the encoder consists of CNN layers and the decoder consists of GRU layers. However, other variants of the model structure is possible. For example, the input to the model is highly sparse (one-hot in nature), allowing the use of fully-connected layers as a pre-processing step or as the encoder layers.

8.8.3 Challenges of Adopting Deep Cascade Generative Models

Although generative models have been used as a popular research tool in augmenting the understanding of large data, there are key challenges existent within the adoption of generative models and the application of deep cascade learning on those models.

First, since the underlying distribution of the data is unknown, it is difficult to compare the performance of the end-to-end model [335] and the cascaded model. I can only "embed intuition in the machine's understanding" through the application of deep cascade learning on generative models [335]. Thus, it is possible to use common metrics to compare performances but it is difficult to arrive at which model truly represents the underlying distribution of the data.

Second, the current generative models adopted in drug design is only at a stage of infancy, thus the application of deep cascade learning is also nascent. While models as described in literature (Section 8.2) have demonstrated some effectiveness, it is still difficult to produce an all-encompassing end-to-end pipeline to automate the process of *de novo* drug design. A high level of domain expertise coupled with deep understanding of the generative models are often required to produce a truly performing model. Therefore, to apply deep cascade learning on such models relies highly on expert knowledge as well and should not be simply interpreted as a blind application of layer-wise optimisation. More understanding of the chemical search space will be critical in generating a functional cascaded generative model.

Moreover, both the end-to-end and the cascaded model produced often have limited generalisation. Practical implementation has demonstrated that a change in the training and validation dateset often leads to large changes in the resulting model. Changes in the hyperparameters and the data size also vary the performance of the models greatly. Therefore, the application of deep cascade learning on generative models for the purpose of *de novo*

drug design needs to overcome the hurdle of proper model development in order to become integrated in an automated process.

The third challenge is how to make use of the generative models to augment human understanding. While the models are effective in generating novel molecules, the process is still largely "black-box" in nature. How to make use the the model results, for example the latent representation in the case of autoencoders, to assist human's understanding into the pharmacology of the designed drugs is still a great challenge today. To understand how deep cascade learning improves model performance in a more fundamental sense is also a difficult task. More dive into the nature of end-to-end learning versus the deep cascade learning is required to truly improve our understanding of a guided search in the chemical space.

While the challenges of adopting deep cascade learning on generative models in the context of *de novo* drug design exist, there are also advantages of such a methodology. For one, the application of deep cascade learning is novel. The current models in literature exclusively focus on modifications of the model to improve performance [336], and few investigates the method from the perspective of an architecture search. In the past, deep cascade learning is often adopted in Human Activity Recognition tasks [267]. To apply it in the drug design space is a novel idea.

8.8.4 Outlook of Deep Cascade Learning

The key reason that I have applied deep cascade learning on the generative model in the area of *de novo* drug design is that I believe the layer-wise optimisation is capable of extracting coarser information from the dataset, in this case, the properties related to molecular structures at a larger scale. The positive results obtained demonstrate the potential validity of this hypothesis.

However, to fully prove such a hypothesis requires much more research into the fundamental aspect of the model, which is out of scope of the current research project. It is noteworthy that to fully prove such a hypothesis, the new molecules generated from the latent space should be examined and compared with the molecules generated from the end-to-end model. The properties of the generated molecules should be compared for both models in order to fully assert that one model outperforms the other. Since research into this area is still at a nascent stage, and due to the sheer size of the chemical space being researched into, such research entails much difficulty and requires a large amount of efforts.

There are several advantages of adopting deep cascade learning on the generative model. For one, the model freezes the trained parameters in each run, reducing the computational expense in each iteration, allowing less computations compared to the training of an end-to-end model. Second, the model has demonstrated good performance in transfer learning

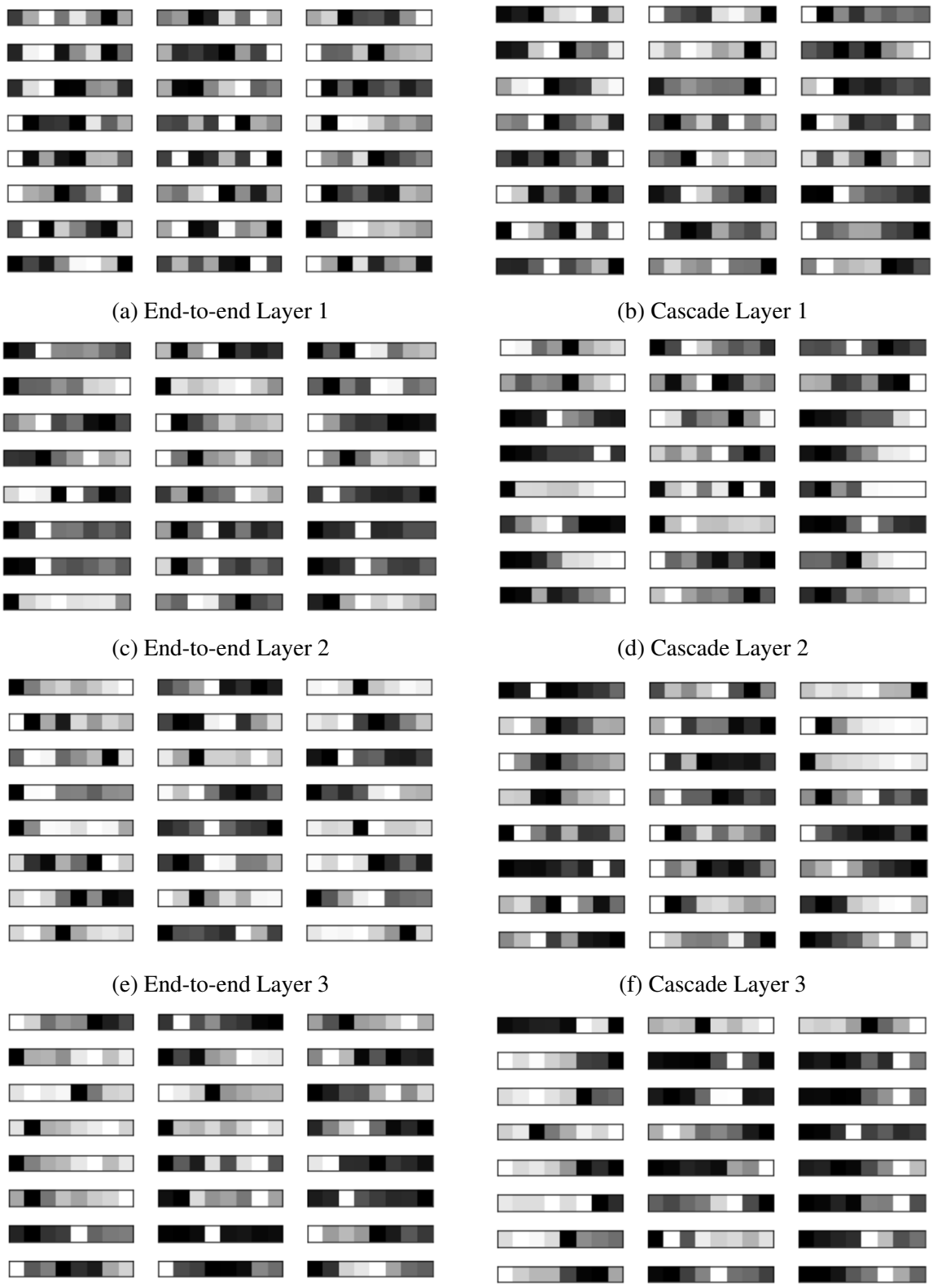
[267], thus guiding research directions into applying the model to smaller datasets. For example, data on drug properties less well-researched can be calculated in the model to predict such properties from molecular structures. Third, the different latent space obtained from the end-to-end model may indicate a future research direction in comparing whether the cascaded latent space is a better alternative in drug design research. Overall, I believe research into deep cascade learning to be applied to *de novo* drug design has great potential and I should encourage on-going research into this area.

8.9 Summary

In this chapter, I have described a dynamic method of searching for the optimal architecture through deep cascade learning. An autoencoder and a variational autoencoder are developed to solve the problem of *de novo* drug design. Similar to Chapter 7 where layers are added one by one, the deep cascade learning method also adds encoder layers layer-wise to train for network parameters that is capable of extracting the coarser features from the input data. This is because during optimisation, the encoder layer is always close to the output data points. Compared to the model in Chapter 7, this model is applicable to dataset of a larger scale and is capable of producing variant forms of ANNs.

In terms of future work, the key advantage of deep cascaded network is that it is better at extracting coarser features from the input data, thus preserving more information in the initial layers of the network. Therefore, it is applicable to transfer learning, where similar problems are solved with the same initial layers. I believe our network is suitable for transfer learning in the *de novo* chemical design problem as I could predict chemical properties based on the features extracted from the same set of molecules.

Another application is in drug repositioning, alternatively called drug repurposing. Drug repositioning is the process of developing new functionality of an already commercialised drug or a drug under investigation. The aim is to find uses outside the scope of original indications. Our approach can be adopted to this field by utilising the latent representations of the trained molecules and finding the drugs with similar latent representations. It is believed that if two molecules are geometrically close in the latent space, they may exhibit similar properties. A commercialised drug close to a certain group of molecules in the latent space is expected to exhibit similar properties to that group. Therefore, observing the latent space is applicable to the process of drug repositioning.



(a) End-to-end Layer 1

(b) Cascade Layer 1

(c) End-to-end Layer 2

(d) Cascade Layer 2

(e) End-to-end Layer 3

(f) Cascade Layer 3

(g) End-to-end Layer 4

(h) Cascade Layer 4

Figure 8.16 The images of filter values for each layer in the end-to-end and the cascade learning model adopting a variational autoencoder.

Chapter 9

Discussion, Conclusion and Future Work

9.1 Discussion and Conclusion

In this chapter, I conclude the thesis by summarising and evaluating key contributions in the research and identifying possible areas of future research. To evaluate the key contributions, I define a series of *desiderata* for an optimisation process. I also discuss the possibility of integrating the optimisation processes into a complete framework. Lastly, I discuss popular topics in the area of machine learning research and how this research is impacted by those topics. Future areas of research are also investigated in relation to the popular topics.

9.1.1 Overview of Thesis

The contributions of thesis is six-fold. In each contribution, I define and develop an optimisation process involved in the design of Deep Neural Networks (DNNs), including the training or the neural architecture search process. I summarise the contributions as follows:

- Chapter 3 proposed a novel quasi-Newton optimisation method for the training of DNNs.
- Chapter 4 proposed a lifting framework for the automated search of optimal architecture of DNNs in a pruning process that focused on network sparsification.
- Chapter 5 investigated a method to evolve the architecture of the DNNs dynamically under the lifting scheme.
- Chapter 6 proposed a multi-scale hierarchical search algorithm to define a novel training process. A binary search tree was adopted to identify the most sensitive layer to train.

- Chapter 7 introduced multitask learning to the Dynamic Architecture Neural Network and proposed a heuristic search scheme to optimise the architecture.
- Chapter 8 performed dynamic architecture search in the context of *de novo* chemical design.

9.1.2 Evaluation of Contributions

I define the following *desiderata* for each optimisation process in the design of DNNs. The *desiderata* include robustness, efficiency, accuracy, computational simplicity, speed and memory advantage.

In Chapter 3, I developed the HFGF method to optimise the training process. Considering the *desiderata*, the algorithm is robust with regard to changes in the network input, fairly efficient and accurate in comparison to first-order training algorithms. Although the computational simplicity is higher than the state-of-the-art and the speed is slower in some cases, the algorithm has an advantage in reducing the number of iterations optimised to the local minimum. The storage memory is also comparable to the state-of-the-art algorithms.

In Chapter 4, the architecture optimisation method based on Lagrangian multipliers is robust, efficient, accurate and conceptually simple. However, due to the increased number of variables involved in the lifting scheme, the method is disadvantaged in memory storage and speed compared to other sparsification algorithms. However, it is a novel idea in terms of how sparsification is performed and it introduces the possibility to regularise a layer or a neuron instead of a particular connection.

In Chapter 5, the adoption of attenuation/amplification factor generates an algorithm that is robust, efficient and accurate. The calculation is less computationally complex compared to other network-evolving algorithms as outlined in Chapter 2. In terms of speed and memory, the algorithm is advantaged similarly due to its simple concept compared to other network-evolving algorithms, although the speed is reduced when a larger network or a higher number of data points are introduced.

In Chapter 6, the proposed method of multi-scale hierarchical tuning is robust and accurate. It is computationally simpler as only one layer is tuned in each iteration. The speed and efficiency is less of an advantage but there is the potential of parallelisation. The memeory storage is the same as the end-to-end backpropagation algorithm.

In Chapter 7, I developed a heuristic search algorithm to optimise a multi-task DAN2 network. The key advantage of the algorithm is its simplicity, with a data-driven convergence criterion. The algorithm is not as robust as traditional DNNs due to the divergence in some cases. However, it is more efficient and computationally simple with a limited number of

parameters in each layer. The accuracy and speed is comparable to traditional DNNs, but the memory storage is much lower.

In Chapter 8, the deep cascade learning method is robust and accurate. It is less efficient and takes longer to train compared to traditional end-to-end learning due to its calculation of weights layer-wise. However, it has the advantage of optimising to a better optimum. It is conceptually and computationally simple. In each step, memory storage is required for weights in only the layer optimised.

A summary of each contribution evaluated based on the *desiderata* is shown in Table 9.1. Overall, each method has its own advantages and disadvantages compared to traditional or state-of-the-art algorithms. As the "no free lunch" theorem dictates, there is no single algorithm that wins over all other algorithms in all optimisation tasks. Thus, I believe our proposed algorithms are novel and ingenious, and they provide alternative ways of thinking to approach the problem of the design of deep neural networks.

Table 9.1 Summary of contributions evaluated based on defined *desiderata*

Chapters	3	4	5	6	7	8
Robustness	+++	+++	++	+++	+	++
Efficiency	++	+++	++	+	+++	+
Accuracy	++	+++	++	+++	++	++
Computational Simplicity	+	+++	+++	+++	+++	+++
Speed	+	+	+	+	++	+
Memory Advantage	++	+	+	+	+++	+++

9.1.3 Integration of Framework

The thesis discusses separately several optimisation processes involved in the design of a deep neural network (DNN). In particular, Chapter 3 and Chapter 6 discuss methods to train a network, and the other chapters discuss methods to evolve a network in a dynamic and interactive manner. This completes the design process of a deep neural network which involves training and architecture search. Although the methods have been introduced separately, I discuss the possibility of integrating the ideas into a complete optimisation framework.

The problem of neural network design is essentially a bilevel optimisation problem, where on the outer level is the optimisation of the network architecture and on the inner level is the optimisation of the weights (training). This is defined as follows:

$$\min_{\rho} \quad \mathscr{L}_{val}(w^{\star}(\rho), \rho) \tag{9.1a}$$

$$\text{subject to} \quad w^{\star}(\rho) = \underset{w}{\operatorname{argmin}}\, \mathscr{L}_{train}(w, \rho) \tag{9.1b}$$

where ρ refers to the architectural parameters and w refers to the weights of the network.

To best balance between the two optimisation levels, several methods have been developed. In [150], it achieves the balance by optimising the outer level and the inner level each for one iteration in turn. This provides a simple solution to the originally complicated bilevel optimisation problem. Adopting this method, it is possible to integrate the training and the architecture search process, by alternating between the two optimisation processes.

For example, the training method proposed in Chapter 6 and the architecture search method proposed in Chapter 5 both makes use of the sensitivity values of the attenuation/amplification factor obtained through finite difference method. Thus, it is possible to integrate the two processes by calculating the sensitivity values and optimise the architecture and the weights in turn, with an unequal number of iterations in each case defined by a hyperparameter. Therefore, there is great potential of integration into a complete framework for the design of deep neural networks.

The architecture optimisation processes proposed in Chapter 7 and Chapter 8 automatically involves the training in the architecture process. They are viewed as an integrated framework because the training is naturally involved in the architecture search process. I believe the integration into a framework can greatly simplify the design of deep neural networks and facilitate the democratisation of AI.

The practical end of AI democratisation is the development of a complete data-driven framework performing tasks of AutoML. Although I have reviewed commercial applications of AutoML in Section 2.3, an open-source and easy-to-use framework designed particularly for Neural Architecture Search is yet designed. Although there are much current research interest in the area with cutting-edge research being implemented in practical applications, a framework with high performance results and good functionality is widely desired.

Our proposed method has the potential to contribute to the development of AutoML, by introducing optimisation processes that can complete the framework of automatically search for the optimal architecture and weights. As future research, it is possible to enable

the AutoML function of the proposed algorithms and make contributions to the AutoML literature.

9.2 Future Work

In this section, I discuss several important aspects of the proposal of a framework that automate the optimisation processes used in deep neural networks. I also discuss topics related to the automation of deep neural network design which consist of the current and future research directions. I explore the correlation of each topic with different frameworks proposed in this thesis and examine the implications of each framework in each topic.

9.2.1 Democratisation of AI

The ultimate purpose of instigating research into training and architecture search is not only about enhancing the understanding of how neural networks operate. More importantly, the research aims to facilitate AI democratisation and the adoption of AI in the chemical industry.

Democratisation of AI is currently an important topic of research since an automated system where less domain expertise is required can unleash the hidden power AI exhibits. By simplifying AI system operations and allowing participation with a lower hurdle, more industrial practitioners can make use of AI in their respective field. Traditional AI research on neural networks require domain-specific expertise. Since the design of neural networks is conventionally manual and arbitrary, a researcher in computer vision cannot transfer his or her expertise in designing neural networks to other fields such as speech recognition. By allowing automated search for the architecture, the problem of neural network design becomes a data-driven problem. Expertise from various fields does not input to the system that generates the optimal design.

I discuss how our developed algorithms deliver a push towards the democratisation of AI. In the Chapter 3, I proposed a training method that enables more efficient search towards the optimal point, which provides a handy tool allowing the automatic search of optimality in a neural network. In the following three chapters, I allow democratisation by fully automate the architecture search process that allows the network to evolve with regard to the data, to fully define a data-driven process. The autonomous system requires minimal input from the user, enabling even a freshman to be able to use the AI system and design the optimal architecture. In Chapter 7, an easily designed heuristic scheme allows the search for an optimal structure to become a simple concept. This allows democratisation by simplifying the optimisation process with an alternative novel architecture. In Chapter 8, the architectural search process

is also automatic, allowing a layman to be able to solve a complicated chemical problem with easily defined deep neural networks.

More specifically, this thesis demonstrates how democratised AI can be used in the context of chemical industry. In particular, I have developed solutions to the prediction problems in the chemical engineering space in Chapter 3 and 7. In Chapter 4-6, I provide solutions to the ANN control problems in both the chemical industry as an example and possibly also in other industries. In Chapter 8, I choose the *de novo* chemical design problem that is more efficiently solved by the automatic architectural search framework. In all chapters, I have demonstrated that the democratised AI can have profound effects when applied to the chemical industry.

The authors of [54] pointed out the key qualities of frameworks that aim to democratise AI. These include accessibility, affordability, explainability, credibility and fairness. I demonstrate that our designed frameworks ace at such criteria. In particular, the algorithms are accessible in that the concepts behind the implementation is simple and easily reproducible. They are affordable because the programs are scalable, and depending on the data size, the algorithms can be run on simple CPU processors or scaled up to GPU processors. They are explainable, all backed up with clear-cut mathematical principles. For example, the autonomous learning system developed in Chapter 4-6 gives a sound explanation that the evolution of the network is guided by sensitivities. In the same sense, the algorithms are credible and reproducible. They provide fairness given that the algorithms are open to application from anyone.

9.2.2 Reproducibility

Reproducibility is a big problem because in most current research of neural architectural search, the results are often not well-reproducible due to randomness in the search process. Attempts seeking to reproduce the search often arrive at less optimal architecture and the results reported in literature cannot always been reproduced (since most processes require random initialisation and require a long processing time) [337]. Thus, this poses a challenge to current research in whether an optimal architecture can always be produced.

Our results in Chapter 4-6 generate an almost deterministic model to guide the search of optimal architecture, transforming the stochastic architectural search process into a process with high certainty, thus generating optimality that is more credible. Therefore, the architectural search process is advanced in the sense that it overcomes the lack of reproducibility in different runs, outperforming many counterparts in current literature in terms of reproducibility.

The results from Chapter 7 contains a high level of certainty in comparison to many other models. Since the core of the operation relies on linear regression and trigonometric operations, the model is almost deterministic. The only randomness comes from the reference vector used in evaluating the trigonometric functions. If this reference vector is fixed, the model is highly reproducible.

The cascade learning model in Chapter 8 is less deterministic in nature. Due to the increased complexity in the model obtained, the random initialisation and the random dropout makes the model less reproducible but comparable performance can be obtained with different randomised values.

Reproducibility is an important topic because it widely acts as a criterion to assess the model validity [337]. In the area of neural architecture search, there are recent advances in coming up with a framework to assess models [152] [151] [338], and reproducibility is an important criterion. While the current measure to improve reproducibility is to increase the number of runs and obtain aggregated results, a model with a higher level of inherent certainty is desirable in the machine learning community.

9.2.3 Continual Learning

Continual learning refers to the machine learning technique that captures changes in the distribution of input data, altering its learning with new data input [339] [340] [341]. The key is to accumulate a set of knowledge learned sequentially and extract essential information from past experience without too much storage and without "catastrophic forgetting" [342] [343] [344]. Catastrophic forgetting occurs when the model forgets previously learnt experiences when adapting to new data input.

The architecture in Chapter 7 is particularly suited to continual learning tasks due to its dynamic accumulative nature of the architecture. This draws parallel to research literature in continual learning where the network is expanded dynamically with the addition of new data while freezing previously learnt weights [345] [346]. To allow continual learning in the DAN2 network, it is possible to expand layers of the network while freezing the trained layer. Since the number of parameters required in each layer is small, the continual learning model will not be as expensive in terms of storage as traditional neural networks. Moreover, the proposed heuristic search process still works and the continual learning model can be combined with the multi-task model to allow full functionality of the network. However, how many weights to freeze in order to maximise continual learning performance is an ongoing research topic, particularly applied to DAN2 network.

The cascade network proposed in Chapter 8 has a similar mechanism that dynamically expands the network while freezing previously trained weights. While this is not applicable

in the case of *de novo* chemical design, the principle is applicable to continual learning in areas such as computer vision [346] and robotics [339].

I would like to discuss continual learning because it is a growing area of research that entails elements of architectural design. Dynamic networks are particularly designed for continual learning which are also well-researched in neural architecture search. The design for an ordinary dynamic architecture can easily be converted to a continual learning design by defining the optimisation target to be that of the new data. Therefore, it is identified as a potential research area extended from neural architectural search.

9.2.4 Explainable AI

Unlike highly transparent models such as decision trees, most of the current state-of-the-art DNN models lack transparency and interpretability due to their black-box nature. There is an increasing need to provide interpretable results from these models, leading to the research area of "explainable artificial intelligence (XAI)" [347] [348] [349].

In Chapter 4-6, the proposed architectural search method explores the idea of explainability by introducing sensitivity measures to explain the significance of each neuron.

The models in Chapter 7 and Chapter 8 lack transparency and explanability because the method overall is the evaluation of a function of the objective function with regard to each layer. Although I can empirically prove the significance of the method, the model is less advanced in terms of explainability.

9.2.5 AutoML

Automatic Machine Learning, or AutoML, is a popular research area [350] [351]. In Chapter 2, I have reviewed the current research and commercial applications of AutoML. Our research is in essence a research on AutoML with the proposal of automated architecture evolving and training algorithms. I believe our research has the potential to be applied in AutoML to allow a more deterministic search for architectures with minimal user input and minimal involvement of complicated architecture. The key advantages of our methods are the simplicity and transparency. I believe our methods are highly integratable to research in the area of AutoML. As future work, I could compare the performance of our proposed AutoML algorithms and the performance of current commercialised products mentioned in Chapter 2. This will evaluate our proposed algorithms from a different angle.

9.3 Summary

In this chapter, I have evaluated the contributions of the thesis with a series of *desiderata*, including robustness, efficiency, accuracy, computational simplicity, speed and memory advantage. The possibility of integrating the contributions into a complete framework for the design of the deep neural networks is also investigated. I have also discussed the potential of our research to be applied to many popular research topics in the current machine learning research literature. Most importantly, our research has the potential to advance the democratisation of AI. By proposing an automated framework to search for the optimal weights and architecture, the research allows optimisation of the neural network requiring less domain expertise. The network is inherently deterministic in terms of network design and can self-evolve based on the value of the objective function. Moreover, the research results are highly reproducible and the models developed can contribute to many research areas including continual learning, explainable AI and AutoML.

As the final chapter of the thesis, I would like to reiterate the importance of the research on automating the process of the design of deep neural networks. As Artificial Intelligence infiltrates every industry and research field, the framework to automate a search of optimal weights and architectures will gain importance. This thesis contributes to such areas of training and architecture search, and thus is an essential process in achieving the democratisation of AI.

Bibliography

[1] Rina Dechter. *Learning while searching in constraint-satisfaction problems*. University of California, Computer Science Department, Cognitive Systems . . . , 1986.

[2] J. Schmidhuber. Deep Learning. *Scholarpedia*, 10(11):32832, 2015. revision #184887.

[3] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[4] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5353–5360, 2015.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[6] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[7] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[8] Alexis Conneau, Holger Schwenk, Loıc Barrault, and Yann Lecun. Very deep convolutional networks for natural language processing. *arXiv preprint arXiv:1606.01781*, 2, 2016.

[9] Rada Mihalcea, Hugo Liu, and Henry Lieberman. Nlp (natural language processing) for nlp (natural language programming). In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 319–330. Springer, 2006.

[10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[11] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Jason Liang, Elliot Meyerson, Babak Hodjat, Dan Fink, Karl Mutch, and Risto Miikkulainen. Evolutionary neural automl for deep learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 401–409, 2019.

[14] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[15] JB Mockus and LJ Mockus. Bayesian approach to global optimization and application to multiobjective and constrained problems. *Journal of Optimization Theory and Applications*, 70(1):157–172, 1991.

[16] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[17] Ilya Loshchilov and Frank Hutter. Cma-es for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269*, 2016.

[18] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480, 2007.

[19] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*, 2019.

[20] Holger H Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous search*, pages 37–71. Springer, 2011.

[21] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification, 2003.

[22] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems*, pages 2016–2025, 2018.

[23] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[24] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2423–2432, 2018.

[25] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[26] Emmanuel Dufourq and Bruce A Bassett. Eden: Evolutionary deep networks for efficient machine learning. In *2017 Pattern Recognition Association of South Africa and Robotics and Mechatronics (PRASA-RobMech)*, pages 110–115. IEEE, 2017.

[27] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[28] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*, 2019.

[29] Ludovic Denoyer and Patrick Gallinari. Deep sequential neural network. *arXiv preprint arXiv:1410.0510*, 2014.

[30] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[31] Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. In *Advances in Neural Information Processing Systems*, pages 4053–4061, 2016.

[32] Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

[33] Christopher MacLeod and Grant M Maxwell. Incremental evolution in anns: Neural nets which grow. *Artificial Intelligence Review*, 16(3):201–224, 2001.

[34] Stephen Marsland, Jonathan Shapiro, and Ulrich Nehmzow. A self-organising network that grows when required. *Neural networks*, 15(8-9):1041–1058, 2002.

[35] VV Vinod and Sujoy Ghose. Growing nonuniform feedforward networks for continuous mappings. *Neurocomputing*, 10(1):55–69, 1996.

[36] Li Deng, Geoffrey Hinton, and Brian Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603. IEEE, 2013.

[37] Mohammed Khasawneh, Khaled Assaleh, Wesam Sweidan, and Monther Haddad. The application of polynomial discriminant function classifiers to isolated arabic speech recognition. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No. 04CH37541)*, volume 4, pages 3077–3081. IEEE, 2004.

[38] Daniel Povey, Lukáš Burget, Mohit Agarwal, Pinar Akyazi, Feng Kai, Arnab Ghoshal, Ondřej Glembek, Nagendra Goel, Martin Karafiát, Ariya Rastrow, et al. The subspace gaussian mixture model—a structured model for speech recognition. *Computer Speech & Language*, 25(2):404–439, 2011.

[39] Preeti Saini and Parneet Kaur. Automatic speech recognition: A review. *International Journal of Engineering Trends and Technology*, 4(2):132–136, 2013.

[40] Michael G Bechtel, Elise McEllhiney, Minje Kim, and Heechul Yun. Deeppicar: A low-cost deep neural network-based autonomous car. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 11–21. IEEE, 2018.

[41] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[42] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[43] Sean Bell and Kavita Bala. Learning visual similarity for product design with convolutional neural networks. *ACM transactions on graphics (TOG)*, 34(4):1–10, 2015.

[44] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[45] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[46] Rahul Mohan. Deep deconvolutional networks for scene parsing. *arXiv preprint arXiv:1411.4101*, 2014.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[48] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[49] I Ralph Edwards and Jeffrey K Aronson. Adverse drug reactions: definitions, diagnosis, and management. *The lancet*, 356(9237):1255–1259, 2000.

[50] Nicholas P Tatonetti, P Ye Patrick, Roxana Daneshjou, and Russ B Altman. Data-driven prediction of drug effects and interactions. *Science translational medicine*, 4(125):125ra31–125ra31, 2012.

[51] G Canetti, S Froman, J al Grosset, P Hauduroy, Miloslava Langerova, HT Mahler, Gertrud Meissner, DA Mitchison, and L Šula. Mycobacteria: laboratory methods for testing drug sensitivity and resistance. *Bulletin of the World Health Organization*, 29(5):565, 1963.

[52] James C Costello, Laura M Heiser, Elisabeth Georgii, Mehmet Gönen, Michael P Menden, Nicholas J Wang, Mukesh Bansal, Petteri Hintsanen, Suleiman A Khan, John-Patrick Mpindi, et al. A community effort to assess and improve drug sensitivity prediction algorithms. *Nature biotechnology*, 32(12):1202, 2014.

[53] Sushen Zhang, Seyed Mojtaba Hosseini Bamakan, Qiang Qu, and Sha Li. Learning for personalized medicine: A comprehensive review from a deep learning perspective. *IEEE reviews in biomedical engineering*, 12:194–208, 2018.

[54] Shakkeel Ahmed, Ravi S Mula, and Soma S Dhavala. A framework for democratizing ai. *arXiv preprint arXiv:2001.00818*, 2020.

[55] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[56] Anna Sergeevna Bosman, Andries Engelbrecht, and Mardé Helbig. Visualising basins of attraction for the cross-entropy and the squared error neural network loss functions. *Neurocomputing*, 400:113–136, 2020.

[57] Yinjin Ma, Peng Feng, Peng He, Zourong Long, and Biao Wei. Low-dose ct with a deep convolutional neural network blocks model using mean squared error loss and structural similar loss. In *Eleventh International Conference on Information Optics and Photonics (CIOP 2019)*, volume 11209, page 112090I. International Society for Optics and Photonics, 2019.

[58] Yangyang Xia, Sebastian Braun, Chandan KA Reddy, Harishchandra Dubey, Ross Cutler, and Ivan Tashev. Weighted speech distortion losses for neural-network-based real-time speech enhancement. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 871–875. IEEE, 2020.

[59] Kai Hu, Zhenzhen Zhang, Xiaorui Niu, Yuan Zhang, Chunhong Cao, Fen Xiao, and Xieping Gao. Retinal vessel segmentation of color fundus images using multiscale convolutional neural network with an improved cross-entropy loss function. *Neurocomputing*, 309:179–191, 2018.

[60] Zhilu Zhang and Mert R Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. In *32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[61] Yangfan Zhou, Xin Wang, Mingchuan Zhang, Junlong Zhu, Ruijuan Zheng, and Qingtao Wu. Mpce: a maximum probability based cross entropy loss function for neural network classification. *IEEE Access*, 7:146331–146341, 2019.

[62] Junqi Jin, Kun Fu, and Changshui Zhang. Traffic sign recognition with hinge loss trained convolutional neural networks. *IEEE Transactions on Intelligent Transportation Systems*, 15(5):1991–2000, 2014.

[63] Buse Melis Ozyildirim and Mariam Kiran. Levenberg–marquardt multi-classification using hinge loss function. *Neural Networks*, 143:564–571, 2021.

[64] Franco Pellegrini and Giulio Biroli. An analytic theory of shallow networks dynamics for hinge loss classification. *Advances in Neural Information Processing Systems*, 33, 2020.

[65] Taehyeon Kim, Jaehoon Oh, NakYil Kim, Sangwook Cho, and Se-Young Yun. Comparing kullback-leibler divergence and mean squared error loss in knowledge distillation. *arXiv preprint arXiv:2105.08919*, 2021.

[66] Masahito Togami, Yoshiki Masuyama, Tatsuya Komatsu, and Yu Nakagome. Unsupervised training for deep speech source separation with kullback-leibler divergence based probabilistic loss function. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 56–60. IEEE, 2020.

[67] Deepak Gupta, Barenya Bikash Hazarika, and Mohanadhas Berlin. Robust regularized extreme learning machine with asymmetric huber loss function. *Neural Computing and Applications*, 32(16):12971–12998, 2020.

[68] Gregory P Meyer. An alternative probabilistic interpretation of the huber loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5261–5269, 2021.

[69] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.

[70] Abdulhamit Subasi and Ergun Ercelebi. Classification of eeg signals using neural network and logistic regression. *Computer methods and programs in biomedicine*, 78(2):87–99, 2005.

[71] Hikmet Kerem Cigizoglu. Generalized regression neural network in monthly flow forecasting. *Civil Engineering and Environmental Systems*, 22(2):71–81, 2005.

[72] Donald F Specht et al. A general regression neural network. *IEEE transactions on neural networks*, 2(6):568–576, 1991.

[73] Shaul K Bar-Lev, Benzion Boukai, and Peter Enis. On the mean squared error, the mean absolute error and the like. *Communications in Statistics-Theory and Methods*, 28(8):1813–1822, 1999.

[74] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[75] Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*, volume 271. Springer, 2014.

[76] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[77] Arthur Richards and Jonathan How. Mixed-integer programming for control. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 2676–2683. IEEE, 2005.

[78] Laurence A Wolsey. Mixed integer programming. *Wiley Encyclopedia of Computer Science and Engineering*, pages 1–10, 2007.

[79] Kate A Smith and Jatinder ND Gupta. Continuous function optimisation via gradient descent on a neural network approxmiation function. In *International Work-Conference on Artificial Neural Networks*, pages 741–748. Springer, 2001.

[80] Dimitri P Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[81] Michael JD Powell. A fast algorithm for nonlinearly constrained optimization calculations. In *Numerical analysis*, pages 144–157. Springer, 1978.

[82] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM, 1996.

[83] Jorge Nocedal. Theory of algorithms for unconstrained optimization. *Acta numerica*, 1:199–242, 1992.

[84] Laurence CW Dixon. Neural networks and unconstrained optimization. In *Algorithms for Continuous Optimization*, pages 513–530. Springer, 1994.

[85] Luigi Grippo. A class of unconstrained minimization methods for neural network training. *Optimization Methods and Software*, 4(2):135–150, 1994.

[86] Youshen Xia, Henry Leung, and Jun Wang. A projection neural network and its application to constrained optimization problems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(4):447–458, 2002.

[87] Li Zhang, Fulin Wang, Ting Sun, and Bing Xu. A constrained optimization method based on bp neural network. *Neural Computing and Applications*, 29(2):413–421, 2018.

[88] Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

[89] AA Goldstein and JF Price. On descent from local minima. *Mathematics of Computation*, 25(115):569–574, 1971.

[90] Kenji Kawaguchi. Deep learning without poor local minima. *arXiv preprint arXiv:1605.07110*, 2016.

[91] Marco Gori and Alberto Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, 1992.

[92] Grzegorz Swirszcz, Wojciech Marian Czarnecki, and Razvan Pascanu. Local minima in training of neural networks. *arXiv preprint arXiv:1611.06310*, 2016.

[93] Nyamlkhagva Sengee, Chinzorig Radnaabazar, Suvdaa Batsuuri, Khurel-Ochir Tsedendamba, and Berekjan Telue. Hierarchical cluster analysis histogram thresholding with local minima. *Journal of Multimedia Information System*, 4(4):189–194, 2017.

[94] Herve A Bourlard and Nelson Morgan. *Connectionist speech recognition: a hybrid approach*, volume 247. Springer Science & Business Media, 2012.

[95] Juraj Kacur and Gregor Rozinaj. Adding voicing features into speech recognition based on hmm in slovak. In *2009 16th International Conference on Systems, Signals and Image Processing*, pages 1–4. IEEE, 2009.

[96] Yaim Cooper. Global minima of overparameterized neural networks. *SIAM Journal on Mathematics of Data Science*, 3(2):676–691, 2021.

[97] Simon Du, Jason Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. In *International Conference on Machine Learning*, pages 1675–1685. PMLR, 2019.

[98] Benjamin D Haeffele and René Vidal. Global optimality in neural network training. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7331–7339, 2017.

[99] Yu M Ermoliev and RJ-B Wets. *Numerical techniques for stochastic optimization*. Springer-Verlag, 1988.

[100] Johannes Schneider and Scott Kirkpatrick. *Stochastic optimization*. Springer Science & Business Media, 2007.

[101] Marco Cavazzuti. Deterministic optimization. In *Optimization methods*, pages 77–102. Springer, 2013.

[102] Ming-Hua Lin, Jung-Fa Tsai, and Chian-Son Yu. A review of deterministic optimization methods in engineering and management. *Mathematical Problems in Engineering*, 2012, 2012.

[103] Amir Beck. *First-order methods in optimization*. SIAM, 2017.

[104] Hongzhou Lin, Julien Mairal, and Zaid Harchaoui. A universal catalyst for first-order optimization. *arXiv preprint arXiv:1506.02186*, 2015.

[105] James Martens. *Second-order optimization for neural networks*. University of Toronto (Canada), 2016.

[106] Peng Xu, Fred Roosta, and Michael W Mahoney. Second-order optimization for non-convex machine learning: An empirical study. In *Proceedings of the 2020 SIAM International Conference on Data Mining*, pages 199–207. SIAM, 2020.

[107] Philip E Gill and Walter Murray. Quasi-newton methods for unconstrained optimization. *IMA Journal of Applied Mathematics*, 9(1):91–108, 1972.

[108] Adrian S Lewis and Michael L Overton. Nonsmooth optimization via quasi-newton methods. *Mathematical Programming*, 141(1):135–163, 2013.

[109] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[110] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[111] Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, volume 4, page 4. Citeseer, 2010.

[112] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[113] Zhongwei Si, Shaoguo Wen, and Bing Dong. Noma codebook optimization by batch gradient descent. *IEEE Access*, 7:117274–117281, 2019.

[114] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.

[115] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[116] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. *Advances in neural information processing systems*, 26:315–323, 2013.

[117] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14, 2012.

[118] Sarit Khirirat, Hamid Reza Feyzmahdavian, and Mikael Johansson. Mini-batch gradient descent: Faster convergence under data sparsity. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2880–2887. IEEE, 2017.

[119] Ekaba Bisong. Optimization for machine learning: Gradient descent. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pages 203–207. Springer, 2019.

[120] Tomaso Poggio, Stephen Voinea, and Lorenzo Rosasco. Online learning, stability, and stochastic gradient descent. *arXiv preprint arXiv:1105.4701*, 2011.

[121] Bob Carpenter. Lazy sparse stochastic gradient descent for regularized multinomial logistic regression. *Alias-i, Inc., Tech. Rep*, pages 1–20, 2008.

[122] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[123] Agnes Lydia and Sagayaraj Francis. Adagrad—an optimizer for stochastic gradient descent. *Int. J. Inf. Comput. Sci*, 6(5), 2019.

[124] Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: Sharp convergence over nonconvex landscapes. In *International Conference on Machine Learning*, pages 6677–6686. PMLR, 2019.

[125] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[126] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[127] Vivek Ramamurthy and Nigel Duffy. L-sr1: A second order optimization method for deep learning. 2016.

[128] Quoc V Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. 2011.

[129] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

[130] Marc Teboulle. A simplified view of first order methods for optimization. *Mathematical Programming*, 170(1):67–96, 2018.

[131] Chi Jin, Rong Ge, Praneeth Netrapalli, Sham M Kakade, and Michael I Jordan. How to escape saddle points efficiently. In *International Conference on Machine Learning*, pages 1724–1732. PMLR, 2017.

[132] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient descent only converges to minimizers. In *Conference on learning theory*, pages 1246–1257. PMLR, 2016.

[133] Syed Muhammad Aqil Burney, Tahseen Ahmed Jilani, and C Ardil. A comparison of first and second order training algorithms for artificial neural networks. *International Journal of Computer and Information Engineering*, 1(1):145–151, 2007.

[134] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[135] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[136] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated machine learning*, pages 3–33. Springer, Cham, 2019.

[137] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[138] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

[139] Jan N Van Rijn and Frank Hutter. Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2367–2376, 2018.

[140] Michael Green and Mattias Ohlsson. Comparison of standard resampling methods for performance estimation of artificial neural network ensembles. In *Third International Conference on Computational Intelligence in Medicine and Healthcare*, 2007.

[141] Jacob Y Hesterman, Luca Caucci, Matthew A Kupinski, Harrison H Barrett, and Lars R Furenlid. Maximum-likelihood estimation with a contracting-grid search algorithm. *IEEE transactions on nuclear science*, 57(3):1077–1084, 2010.

[142] Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. *arXiv preprint arXiv:1910.11858*, 1(2), 2019.

[143] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[144] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

[145] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier, 2019.

[146] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[147] Tom Veniat and Ludovic Denoyer. Learning time/memory-efficient deep architectures with budgeted super networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3492–3500, 2018.

[148] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[149] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[150] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[151] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[152] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.

[153] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[154] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In *Advances in neural information processing systems*, pages 524–532, 1990.

[155] Enrique S Marquez, Jonathon S Hare, and Mahesan Niranjan. Deep cascade learning. *IEEE transactions on neural networks and learning systems*, 29(11):5475–5485, 2018.

[156] Juan David Velásquez-Henao, Carlos Jaime Franco-Cardona, and Yris Olaya-Morales. A review of dan2 (dynamic architecture for artificial neural networks) model in time series forecasting. *Ingeniería y universidad*, 16(1):135–146, 2012.

[157] F Hutter, R Caruana, R Bardenet, M Bilenko, I Guyon, B Kegl, and H Larochelle. Automl 2014@ icml. In *AutoML 2014 Workshop@ ICML*, 2014.

[158] Oleg Bezrukavnikov and Rhema Linder. A neophyte with automl: Evaluating the promises of automatic machine learning tools. *arXiv preprint arXiv:2101.05840*, 2021.

[159] Adithya Balaji and Alexander Allen. Benchmarking automatic machine learning frameworks. *arXiv preprint arXiv:1808.06492*, 2018.

[160] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An open source automl benchmark. *arXiv preprint arXiv:1907.00909*, 2019.

[161] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015.

[162] H2O.ai. *H2O AutoML*, June 2017. H2O version 3.30.0.1.

[163] Chris Thornton. *Auto-WEKA: combined selection and hyperparameter optimization of supervised machine learning algorithms*. PhD thesis, University of British Columbia, 2014.

[164] George D Magoulas and Michael N Vrahatis. Adaptive algorithms for neural network supervised learning: a deterministic optimization approach. *International Journal of Bifurcation and Chaos*, 16(07):1929–1950, 2006.

[165] Riccardo Bonalli, Bruno Hérissé, and Emmanuel Trélat. Solving nonlinear optimal control problems with state and control delays by shooting methods combined with numerical continuation on the delays. *arXiv preprint arXiv:1709.04383*, 2017.

[166] Markus Giftthaler, Michael Neunert, Markus Stäuble, Jonas Buchli, and Moritz Diehl. A family of iterative gauss-newton shooting methods for nonlinear optimal control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

[167] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[168] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.

[169] Daqing Chang, Shiliang Sun, and Changshui Zhang. An accelerated linearly convergent stochastic l-bfgs algorithm. *IEEE transactions on neural networks and learning systems*, 30(11):3338–3346, 2019.

[170] Philipp Moritz, Robert Nishihara, and Michael Jordan. A linearly-convergent stochastic l-bfgs algorithm. In *Artificial Intelligence and Statistics*, pages 249–258, 2016.

[171] Samuel H Fuller and Lynette I Millett. *The Future of Computing Performance: Game Over or Next Level?* National Academy Press, 2011.

[172] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[173] Ioannis E Livieris and Panagiotis Pintelas. A new conjugate gradient algorithm for training neural networks based on a modified secant equation. *Applied Mathematics and Computation*, 221:491–502, 2013.

[174] Luigi Grippo and Stefano Lucidi. A globally convergent version of the polak-ribiere conjugate gradient method. *Mathematical Programming*, 78(3):375–391, 1997.

[175] H Adeli and SL Hung. An adaptive conjugate gradient learning algorithm for efficient training of neural networks. *Applied Mathematics and Computation*, 62(1):81–102, 1994.

[176] Stephen G Nash. A survey of truncated-newton methods. *Journal of computational and applied mathematics*, 124(1-2):45–59, 2000.

[177] Yi Shang and Benjamin W Wah. Global optimization for neural network training. *Computer*, 29(3):45–54, 1996.

[178] AA Brown and Michael C Bartholomew-Biggs. Some effective methods for unconstrained optimization based on the solution of systems of ordinary differential equations. *Journal of Optimization Theory and Applications*, 62(2):211–224, 1989.

[179] David M Young and Kang C Jea. Generalized conjugate-gradient acceleration of nonsymmetrizable iterative methods. *Linear Algebra and its applications*, 34:159–194, 1980.

[180] Marcin Molga and Czesław Smutnicki. Test functions for optimization needs. *Test functions for optimization needs*, 101, 2005.

[181] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimization problems. *arXiv preprint arXiv:1308.4008*, 2013.

[182] Yun-Wei Shang and Yu-Huang Qiu. A note on the extended rosenbrock function. *Evolutionary Computation*, 14(1):119–126, 2006.

[183] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[184] Joel Andersson, Johan Åkesson, and Moritz Diehl. Casadi: A symbolic package for automatic differentiation and optimal control. In *Recent advances in algorithmic differentiation*, pages 297–307. Springer, 2012.

[185] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20(5):14, 2015.

[186] Ye Yuan, Mu Li, Jun Liu, and Claire Tomlin. On the powerball method: Variants of descent methods for accelerated optimization. *IEEE Control Systems Letters*, 3:601–606, 07 2019.

[187] Rémi Bardenet, Mátyás Brendel, Balázs Kégl, and Michele Sebag. Collaborative hyperparameter tuning. In *International conference on machine learning*, pages 199–207, 2013.

[188] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[189] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[190] Fred H Gage. Neurogenesis in the adult brain. *Journal of Neuroscience*, 22(3):612–613, 2002.

[191] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.

[192] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[193] Jan Albersmeyer and Moritz Diehl. The lifted newton method and its application in optimization. *SIAM Journal on Optimization*, 20(3):1655–1684, 2010.

[194] Mike R Osborne. On shooting methods for boundary value problems. *Journal of mathematical analysis and applications*, 27(2):417–433, 1969.

[195] Boris Kramer and Karen E Willcox. Nonlinear model order reduction via lifting transformations and proper orthogonal decomposition. *AIAA Journal*, 57(6):2297–2307, 2019.

[196] P Khodabakhshi and K Willcox. Non-intrusive data-driven model reduction for differential algebraic equations derived from lifting transformations. *Oden Institute Report*, pages 21–08, 2021.

[197] Wei Wan and Lorenz T Biegler. Structured regularization for barrier nlp solvers. *Computational Optimization and Applications*, 66(3):401–424, 2017.

[198] Neculai Andrei. Performances of descon, l-bfgs, l-cg-descent and of conopt, knitro, minos, snopt, ipopt for solving the problem palmer1c. *Parameters*, 1:35, 2019.

[199] A Pavelka and A Procházka. Algorithms for initialization of neural network weights. In *In Proceedings of the 12th Annual Conference, MATLAB*, pages 453–459, 2004.

[200] Yat-Fung Yam and Tommy WS Chow. Determining initial weights of feedforward neural networks based on least squares method. *Neural Processing Letters*, 2(2):13–17, 1995.

[201] Raúl Conejeros and Vassilios S Vassiliadis. Dynamic biochemical reaction process analysis and pathway modification predictions. *Biotechnology and bioengineering*, 68(3):285–297, 2000.

[202] JJ Montano and Alfonso Palmer. Numeric sensitivity analysis applied to feedforward neural networks. *Neural Computing & Applications*, 12(2):119–125, 2003.

[203] T Tchaban, MJ Taylor, and JP Griffin. Establishing impacts of the inputs in a feedforward neural network. *Neural Computing & Applications*, 7(4):309–317, 1998.

[204] David G Garson. Interpreting neural network connection weights. 1991.

[205] Uchenna Oparaji, Rong-Jiun Sheu, Mark Bankhead, Jonathan Austin, and Edoardo Patelli. Robust artificial neural network for reliability and sensitivity analyses of complex non-linear systems. *Neural Networks*, 96:80–90, 2017.

[206] Peter P.Groumpos. Deep learning vs. wise learning: A critical and challenging overview. *17th IFAC Conference on International Stability, Technology and Culture TECIS 2016: Durrës, Albania, 26—28 October 2016*, 49(29):180–189, 2016.

[207] Muriel Gevrey, Ioannis Dimopoulos, and Sovan Lek. Review and comparison of methods to study the contribution of variables in artificial neural network models. *Ecological modelling*, 160(3):249–264, 2003.

[208] Wenjia Wang, Phillis Jones, and Derek Partridge. Assessing the impact of input features in a feedforward neural network. *Neural Computing & Applications*, 9(2):101–112, 2000.

[209] Yu Zhang, Peter Tiňo, Aleš Leonardis, and Ke Tang. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2021.

[210] Stacy L Özesmi and Uygar Özesmi. An artificial neural network approach to spatial habitat modelling with interspecific interaction. *Ecological modelling*, 116(1):15–31, 1999.

[211] Hiroshi Takenaga, Shigeo Abe, Masao Takatoo, Masahiro Kayama, Tadaaki Kitamura, and Yosiyuki Okuyama. Input layer optimization of neural networks by sensitivity analysis and its application to recognition of numerals. *IEEJ Transactions on Industry Applications*, 111(1):36–44, 1991.

[212] Sovan Lek, Marc Delacoste, Philippe Baran, Ioannis Dimopoulos, Jacques Lauga, and Stéphane Aulagnier. Application of neural networks to modelling nonlinear relationships in ecology. *Ecological modelling*, 90(1):39–52, 1996.

[213] Tamás D Gedeon. Data mining of inputs: analysing magnitude and functional measures. *International Journal of Neural Systems*, 8(02):209–218, 1997.

[214] Andrew Hunter, Lee Kennedy, Jenny Henry, and Ian Ferguson. Application of neural networks and sensitivity analysis to improved prediction of trauma survival. *Computer methods and programs in biomedicine*, 62(1):11–19, 2000.

[215] Wojciech Samek, Alexander Binder, Grégoire Montavon, Sebastian Lapuschkin, and Klaus-Robert Müller. Evaluating the visualization of what a deep neural network has learned. *IEEE transactions on neural networks and learning systems*, 28(11):2660–2673, 2016.

[216] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[217] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*, 2017.

[218] EC Shin, JJ Park, J Yu, and CR Patra. Prediction of grouting efficiency by injection of cement milk into sandy soil using an artificial neural network. *Soil Mechanics and Foundation Engineering*, 55(5):305–311, 2018.

[219] Ruhul Amin Mozumder, Aminul Islam Laskar, and Monowar Hussain. Penetrability prediction of microfine cement grout in granular soil using artificial intelligence techniques. *Tunnelling and Underground Space Technology*, 72:131–144, 2018.

[220] Ram Chandra Chaurasia, Dejalin Sahu, and Nikkam Suresh. Prediction of ash content and yield percent of clean coal in multi gravity separator using artificial neural networks. *International Journal of Coal Preparation and Utilization*, 41(5):362–369, 2021.

[221] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[222] Yinchong Yang, Volker Tresp, Marius Wunderle, and Peter A Fasching. Explaining therapy predictions with layer-wise relevance propagation in neural networks. In *2018 IEEE International Conference on Healthcare Informatics (ICHI)*, pages 152–162. IEEE, 2018.

[223] John Grezmak, Jianjing Zhang, Peng Wang, Kenneth A Loparo, and Robert X Gao. Interpretable convolutional neural network through layer-wise relevance propagation for machine fault diagnosis. *IEEE Sensors Journal*, 20(6):3172–3181, 2019.

[224] Grégoire Montavon, Alexander Binder, Sebastian Lapuschkin, Wojciech Samek, and Klaus-Robert Müller. Layer-wise relevance propagation: an overview. *Explainable AI: interpreting, explaining and visualizing deep learning*, pages 193–209, 2019.

[225] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[226] Jocelyn Sietsma and Robert JF Dow. Creating artificial neural networks that generalize. *Neural networks*, 4(1):67–79, 1991.

[227] Timur Ash. Dynamic node creation in backpropagation networks. *Connection science*, 1(4):365–375, 1989.

[228] German I Parisi, Jun Tani, Cornelius Weber, and Stefan Wermter. Lifelong learning of human actions with deep neural network self-organization. *Neural Networks*, 96:137–149, 2017.

[229] Dietmar Heinke and Fred H Hamker. Comparing neural networks: a benchmark on growing neural gas, growing cell structures, and fuzzy artmap. *IEEE transactions on neural networks*, 9(6):1279–1291, 1998.

[230] Sihan Li, Jiantao Jiao, Yanjun Han, and Tsachy Weissman. Demystifying resnet. *arXiv preprint arXiv:1611.01186*, 2016.

[231] Piotr Olaszek. Investigation of the dynamic characteristic of bridge structures using a computer vision method. *Measurement*, 25(3):227–236, 1999.

[232] Rick Dale, Nicholas D Duran, and Moreno Coco. Dynamic natural language processing with recurrence quantification analysis. *arXiv preprint arXiv:1803.07136*, 2018.

[233] Frederick Jelinek, Bernard Merialdo, Salim Roukos, and Martin Strauss. A dynamic language model for speech recognition. In *Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*, 1991.

[234] M Ghiassi and H Saidane. A dynamic architecture for artificial neural networks. *Neurocomputing*, 63:397–413, 2005.

[235] Gecynalda Soares S Gomes, André Luis Santiago Maia, Teresa Bernarda Ludermir, F de AT de Carvalho, and Aluizio FR Araujo. Hybrid model with dynamic architecture for forecasting time series. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 3742–3747. IEEE, 2006.

[236] John S Bulmer, Adarsh Kaniyoor, Thurid Gspann, Jenifer Mizen, James Ryley, Patrick Kiley, Gijs Ratering, Wouter Sparreboom, Gerhard Bauhuis, Troy Stehr, et al. Forecasting continuous carbon nanotube production in the floating catalyst environment. *Chemical Engineering Journal*, 390:124497, 2020.

[237] Claudia N Okonkwo, Jason J Lee, Anton De Vylder, Yadong Chiang, Joris W Thybaut, and Christopher W Jones. Selective removal of hydrogen sulfide from simulated biogas streams using sterically hindered amine adsorbents. *Chemical Engineering Journal*, 379:122349, 2020.

[238] Stephen Grossberg. Adaptive resonance theory. *Scholarpedia*, 8(5):1569, 2013.

[239] German I Parisi. Human action recognition and assessment via deep neural network self-organization. *arXiv preprint arXiv:2001.05837*, 2020.

[240] Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in neural information processing systems*, pages 625–632, 1995.

[241] Goutam Chakraborty, Mitsuru Murakami, Norio Shiratori, and Shoichi Noguchi. A growing network that optimizes between undertraining and overtraining. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 2, pages 1116–1120. IEEE, 1995.

[242] ÖZLEM POLAT and Zümray Dokur. Protein fold classification with grow-and-learn network. *Turkish Journal of Electrical Engineering & Computer Sciences*, 25(2):1184–1196, 2017.

[243] Lenka Pitonakova and Seth Bullock. The robustness-fidelity trade-off in grow when required neural networks performing continuous novelty detection. *Neural Networks*, 122:183–195, 2020.

[244] Chayut Wiwatcharakoses and Daniel Berrar. Soinn+, a self-organizing incremental neural network for unsupervised learning from noisy data streams. *Expert Systems with Applications*, 143:113069, 2020.

[245] M Ghiassi and Stanley Nangoy. A dynamic artificial neural network model for forecasting nonlinear processes. *Computers & Industrial Engineering*, 57(1):287–297, 2009.

[246] M Ghiassi, H Saidane, and DK Zimbra. A dynamic artificial neural network model for forecasting time series events. *International Journal of Forecasting*, 21(2):341–362, 2005.

[247] Lin Wang, Zhigang Wang, Hui Qu, and Shan Liu. Optimal forecast combination based on neural networks for time series forecasting. *Applied soft computing*, 66:1–17, 2018.

[248] Manoochehr Ghiassi, Michael Olschimke, Brian Moon, and Paul Arnaudo. Automated text classification using a dynamic artificial neural network model. *Expert Systems with Applications*, 39(12):10967–10976, 2012.

[249] M Ghiassi, David Lio, and Brian Moon. Pre-production forecasting of movie revenues with a dynamic artificial neural network. *Expert Systems with Applications*, 42(6):3176–3193, 2015.

[250] David Zimbra, Manoochehr Ghiassi, and Sean Lee. Brand-related twitter sentiment analysis using feature engineering and the dynamic architecture for artificial neural networks. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 1930–1938. IEEE, 2016.

[251] Manoochehr Ghiassi, David K Zimbra, and Hassine Saidane. Urban water demand forecasting with a dynamic artificial neural network model. *Journal of Water Resources Planning and Management*, 134(2):138–146, 2008.

[252] MDKZ Ghiassi, David K Zimbra, and H Saidane. Medium term system load forecasting with a dynamic artificial neural network model. *Electric power systems research*, 76(5):302–316, 2006.

[253] Erkam Güreşen and Gülgün Kayakutlu. Forecasting stock exchange movements using artificial neural network models and hybrid models. In *International Conference on Intelligent Information Processing*, pages 129–137. Springer, 2008.

[254] Erkam Guresen, Gulgun Kayakutlu, and Tugrul U Daim. Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, 38(8):10389–10397, 2011.

[255] M Ghiassi and C Burnley. Measuring effectiveness of a dynamic artificial neural network algorithm for classification problems. *Expert Systems with Applications*, 37(4):3118–3128, 2010.

[256] Hongfei Lu, Wu Jiang, M Ghiassi, Sean Lee, and Mantri Nitin. Classification of camellia (theaceae) species using leaf architecture variations and pattern recognition techniques. *PloS one*, 7(1), 2012.

[257] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Greedy layerwise learning can scale to imagenet. *arXiv preprint arXiv:1812.11446*, 2018.

[258] Bharath Ramsundar, Steven Kearnes, Patrick Riley, Dale Webster, David Konerding, and Vijay Pande. Massively multitask networks for drug discovery. *arXiv preprint arXiv:1502.02072*, 2015.

[259] Mehmet Gönen and Adam A Margolin. Drug susceptibility prediction against a panel of drugs using kernelized bayesian multitask learning. *Bioinformatics*, 30(17):i556–i563, 2014.

[260] Xiaoqiang Lu, Xuelong Li, and Lichao Mou. Semi-supervised multitask learning for scene recognition. *IEEE transactions on cybernetics*, 45(9):1967–1976, 2014.

[261] Zhou Yu, Yijun Song, Jun Yu, Meng Wang, and Qingming Huang. Intra-and inter-modal multilinear pooling with multitask learning for video grounding. *Neural Processing Letters*, pages 1–17, 2020.

[262] Fei Tao and Carlos Busso. End-to-end audiovisual speech recognition system with multitask learning. *IEEE Transactions on Multimedia*, 2020.

[263] Jakob Poncelet et al. Multitask learning with capsule networks for speech-to-intent applications. *arXiv preprint arXiv:2002.07450*, 2020.

[264] Jingye Li, Meishan Zhang, Donghong Ji, and Yijiang Liu. Multi-task learning with auxiliary speaker identification for conversational emotion recognition. *arXiv*, pages arXiv–2003, 2020.

[265] Hao Fei, Yafeng Ren, and Donghong Ji. Dispatched attention with multi-task learning for nested mention recognition. *Information Sciences*, 513:241–251, 2020.

[266] Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. In *Advances in Neural Information Processing Systems*, pages 527–538, 2018.

[267] Xin Du, Katayoun Farrahi, and Mahesan Niranjan. Transfer learning across human activities using a cascade neural network architecture. In *Proceedings of the 23rd international symposium on wearable computers*, pages 35–44, 2019.

[268] Shoujiang Xu, Qingfeng Tang, Linpeng Jin, and Zhigeng Pan. A cascade ensemble learning model for human activity recognition with smartphones. *Sensors*, 19(10):2307, 2019.

[269] Mariya Popova, Olexandr Isayev, and Alexander Tropsha. Deep reinforcement learning for de novo drug design. *Science advances*, 4(7):eaap7885, 2018.

[270] Thomas Blaschke, Marcus Olivecrona, Ola Engkvist, Jürgen Bajorath, and Hongming Chen. Application of generative autoencoder in de novo molecular design. *Molecular informatics*, 37(1-2):1700123, 2018.

[271] Gisbert Schneider, Man-Ling Lee, Martin Stahl, and Petra Schneider. De novo design of molecular architectures by evolutionary assembly of drug-derived building blocks. *Journal of computer-aided molecular design*, 14(5):487–494, 2000.

[272] Enno Littmann and Helge Ritter. Generalization abilities of cascade network architecture. In *Advances in neural information processing systems*, pages 188–195, 1993.

[273] A Adams and Sam Waugh. Function evaluation and the cascade-correlation architecture. In *IEEE International Conference on Neural Networks*, pages 942–946. Citeseer, 1995.

[274] Giuliano Armano, Michele Marchesi, and Andrea Murru. A hybrid genetic-neural architecture for stock indexes forecasting. *Information Sciences*, 170(1):3–33, 2005.

[275] Harold Henry Chaput. *The constructivist learning architecture: A model of cognitive development for robust autonomous robots*. PhD thesis, 2004.

[276] Dhananjay S Phatak and Israel Koren. Connectivity and performance tradeoffs in the cascade correlation learning architecture. *IEEE Transactions on Neural Networks*, 5(6):930–935, 1994.

[277] Ke Chen, Liping Yang, Xiang Yu, and Huisheng Chi. A self-generating modular neural network architecture for supervised learning. *Neurocomputing*, 16(1):33–48, 1997.

[278] Randall S Sexton, Robert E Dorsey, and Naheel A Sikander. Simultaneous optimization of neural network function and architecture algorithm. *Decision Support Systems*, 36(3):283–296, 2004.

[279] John Platt. A resource-allocating network for function interpolation. *Neural computation*, 3(2):213–225, 1991.

[280] Yeonjong Shin. Effects of depth, width, and initialization: A convergence analysis of layer-wise training for deep linear neural networks. *arXiv preprint arXiv:1910.05874*, 2019.

[281] Yunhao Gao, Feng Gao, Junyu Dong, and Shengke Wang. Change detection from synthetic aperture radar images based on channel weighting-based deep cascade network. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(11):4517–4529, 2019.

[282] Sam Leroux, Steven Bohez, Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. The cascading neural network: building the internet of smart things. *Knowledge and Information Systems*, 52(3):791–814, 2017.

[283] Mohammad Sabokrou, Mohsen Fayyaz, Mahmood Fathy, and Reinhard Klette. Deep-cascade: Cascading 3d deep neural networks for fast anomaly detection and localization in crowded scenes. *IEEE Transactions on Image Processing*, 26(4):1992–2004, 2017.

[284] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. The rise of deep learning in drug discovery. *Drug discovery today*, 23(6):1241–1250, 2018.

[285] Evgeny Putin, Arip Asadulaev, Yan Ivanenkov, Vladimir Aladinskiy, Benjamin Sanchez-Lengeling, Alán Aspuru-Guzik, and Alex Zhavoronkov. Reinforced adversarial neural computer for de novo molecular design. *Journal of chemical information and modeling*, 58(6):1194–1204, 2018.

[286] Katja Hansen, Franziska Biegler, Raghunathan Ramakrishnan, Wiktor Pronobis, O Anatole Von Lilienfeld, Klaus-Robert Müller, and Alexandre Tkatchenko. Machine learning predictions of molecular properties: Accurate many-body potentials and nonlocality in chemical space. *The journal of physical chemistry letters*, 6(12):2326–2331, 2015.

[287] Grégoire Montavon, Matthias Rupp, Vivekanand Gobre, Alvaro Vazquez-Mayagoitia, Katja Hansen, Alexandre Tkatchenko, Klaus-Robert Müller, and O Anatole Von Lilienfeld. Machine learning of molecular electronic properties in chemical compound space. *New Journal of Physics*, 15(9):095003, 2013.

[288] Feixiong Cheng and Zhongming Zhao. Machine learning-based prediction of drug–drug interactions by integrating drug phenotypic, therapeutic, chemical, and genomic properties. *Journal of the American Medical Informatics Association*, 21(e2):e278–e286, 2014.

[289] Ross D King and Michael JE Sternberg. Machine learning approach for the prediction of protein secondary structure. *Journal of molecular biology*, 216(2):441–457, 1990.

[290] Stephen Muggleton, Ross D King, and Michael JE Stenberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering, Design and Selection*, 5(7):647–657, 1992.

[291] Connor W Coley, William H Green, and Klavs F Jensen. Machine learning in computer-aided synthesis planning. *Accounts of chemical research*, 51(5):1281–1289, 2018.

[292] Jun Li and Martin D Eastgate. Making better decisions during synthetic route design: leveraging prediction to achieve greenness-by-design. *Reaction Chemistry & Engineering*, 4(9):1595–1607, 2019.

[293] Marwin HS Segler and Mark P Waller. Neural-symbolic machine learning for retrosynthesis and reaction prediction. *Chemistry–A European Journal*, 23(25):5966–5971, 2017.

[294] Jean-Pierre Doucet, Florent Barbault, Hairong Xia, Annick Panaye, and Botao Fan. Nonlinear svm approaches to qspr/qsar studies and drug design. *Current Computer-Aided Drug Design*, 3(4):263–289, 2007.

[295] Michael Fernandez, Julio Caballero, Leyden Fernandez, and Akinori Sarai. Genetic algorithm optimization in drug design qsar: Bayesian-regularized genetic neural networks (brgnn) and genetic algorithm-optimized support vectors machines (ga-svm). *Molecular diversity*, 15(1):269–289, 2011.

[296] James Devillers. *Neural networks in QSAR and drug design*. Academic Press, 1996.

[297] Gisbert Schneider. Neural networks are useful tools for drug design. *Neural Networks*, 13(1):15–16, 2000.

[298] Lothar Terfloth and Johann Gasteiger. Neural networks and genetic algorithms in drug design. *Drug Discovery Today*, 6:102–108, 2001.

[299] Gerhard Hessler and Karl-Heinz Baringhaus. Artificial intelligence in drug design. *Molecules*, 23(10):2520, 2018.

[300] Galina Samigulina and Samigulina Zarina. Immune network technology on the basis of random forest algorithm for computer-aided drug design. In *International Conference on Bioinformatics and Biomedical Engineering*, pages 50–61. Springer, 2017.

[301] Fahimeh Ghasemi, Alireza Mehridehnavi, Afshin Fassihi, and Horacio Pérez-Sánchez. Deep neural network in qsar studies using deep belief network. *Applied soft computing*, 62:251–258, 2018.

[302] Yankang Jing, Yuemin Bian, Ziheng Hu, Lirong Wang, and Xiang-Qun Sean Xie. Deep learning for drug design: an artificial intelligence paradigm for drug discovery in the big data era. *The AAPS journal*, 20(3):1–10, 2018.

[303] Shan-Ju Yeh, Jin-Fu Lin, and Bor-Sen Chen. Multiple-molecule drug design based on systems biology approaches and deep neural network to mitigate human skin aging. *Molecules*, 26(11):3178, 2021.

[304] Feisheng Zhong, Jing Xing, Xutong Li, Xiaohong Liu, Zunyun Fu, Zhaoping Xiong, Dong Lu, Xiaolong Wu, Jihui Zhao, Xiaoqin Tan, et al. Artificial intelligence in drug design. *Science China Life Sciences*, 61(10):1191–1204, 2018.

[305] Sunghwan Kim, Paul A Thiessen, Evan E Bolton, Jie Chen, Gang Fu, Asta Gindulyte, Lianyi Han, Jane He, Siqian He, Benjamin A Shoemaker, et al. Pubchem substance and compound databases. *Nucleic acids research*, 44(D1):D1202–D1213, 2016.

[306] Teague Sterling and John J Irwin. Zinc 15–ligand discovery for everyone. *Journal of chemical information and modeling*, 55(11):2324–2337, 2015.

[307] David Mendez, Anna Gaulton, A Patrícia Bento, Jon Chambers, Marleen De Veij, Eloy Félix, María Paula Magariños, Juan F Mosquera, Prudence Mutowo, Michał Nowotka, et al. Chembl: towards direct deposition of bioassay data. *Nucleic acids research*, 47(D1):D930–D940, 2019.

[308] Kirill Degtyarenko, Paula De Matos, Marcus Ennis, Janna Hastings, Martin Zbinden, Alan McNaught, Rafael Alcántara, Michael Darsow, Mickaël Guedj, and Michael Ashburner. Chebi: a database and ontology for chemical entities of biological interest. *Nucleic acids research*, 36(suppl_1):D344–D350, 2007.

[309] David S Wishart, Craig Knox, An Chi Guo, Dean Cheng, Savita Shrivastava, Dan Tzur, Bijaya Gautam, and Murtaza Hassanali. Drugbank: a knowledgebase for drugs, drug actions and drug targets. *Nucleic acids research*, 36(suppl_1):D901–D906, 2008.

[310] Stanley I Letovsky, Robert W Cottingham, Christopher J Porter, and Peter WD Li. Gdb: the human genome database. *Nucleic Acids Research*, 26(1):94–99, 1998.

[311] Antony J Williams, Christopher M Grulke, Jeff Edwards, Andrew D McEachran, Kamel Mansouri, Nancy C Baker, Grace Patlewicz, Imran Shah, John F Wambaugh, Richard S Judson, et al. The comptox chemistry dashboard: a community data resource for environmental chemistry. *Journal of cheminformatics*, 9(1):61, 2017.

[312] Christoph Steinbeck, Stefan Krause, and Stefan Kuhn. Nmrshiftdb constructing a free chemical information system with open-source components. *Journal of chemical information and computer sciences*, 43(6):1733–1739, 2003.

[313] Jean-Louis Reymond, Lars Ruddigkeit, Lorenz Blum, and Ruud van Deursen. The enumeration of chemical space. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2(5):717–733, 2012.

[314] Gisbert Schneider and Karl-Heinz Baringhaus. *Molecular design: concepts and applications*. John Wiley & Sons, 2008.

[315] Lorenz M Mayr and Dejan Bojanic. Novel trends in high-throughput screening. *Current opinion in pharmacology*, 9(5):580–588, 2009.

[316] Jiayu Gong, Chaoqian Cai, Xiaofeng Liu, Xin Ku, Hualiang Jiang, Daqi Gao, and Honglin Li. Chemmapper: a versatile web server for exploring pharmacology and chemical structure association based on molecular 3d similarity method. *Bioinformatics*, 29(14):1827–1829, 2013.

[317] Marwin HS Segler, Thierry Kogej, Christian Tyrchan, and Mark P Waller. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, 4(1):120–131, 2018.

[318] William Yuan, Dadi Jiang, Dhanya K Nambiar, Lydia P Liew, Michael P Hay, Joshua Bloomstein, Peter Lu, Brandon Turner, Quynh-Thu Le, Robert Tibshirani, et al. Chemical space mimicry for drug discovery. *Journal of chemical information and modeling*, 57(4):875–882, 2017.

[319] Matthew Ragoza, Joshua Hochuli, Elisa Idrobo, Jocelyn Sunseri, and David Ryan Koes. Protein–ligand scoring with convolutional neural networks. *Journal of chemical information and modeling*, 57(4):942–957, 2017.

[320] Natasha Jaques, Shixiang Gu, Dzmitry Bahdanau, José Miguel Hernández-Lobato, Richard E Turner, and Douglas Eck. Sequence tutor: Conservative fine-tuning of sequence generation models with kl-control. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1645–1654. JMLR. org, 2017.

[321] Junhai Zhai, Sufang Zhang, Junfen Chen, and Qiang He. Autoencoder and its various variants. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 415–419. IEEE, 2018.

[322] Artur Kadurin, Sergey Nikolenko, Kuzma Khrabrov, Alex Aliper, and Alex Zhavoronkov. drugan: an advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico. *Molecular pharmaceutics*, 14(9):3098–3104, 2017.

[323] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[324] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350, 2015.

[325] Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. D-vae: A variational autoencoder for directed acyclic graphs. *arXiv preprint arXiv:1904.11088*, 2019.

[326] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[327] NP Todorov, IL Alberts, and PM Dean. De novo design. 2007.

[328] Gisbert Schneider and Uli Fechner. Computer-based de novo design of drug-like molecules. *Nature Reviews Drug Discovery*, 4(8):649–663, 2005.

[329] David Rogers and Mathew Hahn. Extended-connectivity fingerprints. *Journal of chemical information and modeling*, 50(5):742–754, 2010.

[330] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems*, 28:2224–2232, 2015.

[331] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

[332] Matthias Rupp, Alexandre Tkatchenko, Klaus-Robert Müller, and O Anatole Von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Physical review letters*, 108(5):058301, 2012.

[333] AA Toropov, AP Toropova, SE Martyanov, E Benfenati, Giuseppina Gini, D Leszczynska, and J Leszczynski. Comparison of smiles and molecular graphs as the representation of the molecular structure for qsar analysis for mutagenic potential of polyaromatic amines. *Chemometrics and Intelligent Laboratory Systems*, 109(1):94–100, 2011.

[334] Yingbo Zhou and Venu Govindaraju. Learning deep autoencoders without layer-wise training. *stat*, 1050:14, 2014.

[335] Daniel Schwalbe-Koda and Rafael Gómez-Bombarelli. Generative models for automatic chemical design. In *Machine Learning Meets Quantum Physics*, pages 445–467. Springer, 2020.

[336] Ryan-Rhys Griffiths and José Miguel Hernández-Lobato. Constrained bayesian optimization for automatic chemical design using variational autoencoders. *Chemical science*, 11(2):577–586, 2020.

[337] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR, 2020.

[338] Nikita Klyuchnikov, Ilya Trofimov, Ekaterina Artemova, Mikhail Salnikov, Maxim Fedorov, and Evgeny Burnaev. Nas-bench-nlp: neural architecture search benchmark for natural language processing. *arXiv preprint arXiv:2006.07116*, 2020.

[339] Timoth'ee Lesort. Continual learning: Tackling catastrophic forgetting in deep neural networks with replay processes. *arXiv preprint arXiv:2007.00487*, 2020.

[340] Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

[341] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR, 2017.

[342] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[343] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[344] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[345] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

[346] Yu-Xiong Wang, Deva Ramanan, and Martial Hebert. Growing a brain: Fine-tuning by increasing model capacity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2471–2480, 2017.

[347] Derek Doran, Sarah Schulz, and Tarek R Besold. What does explainable ai really mean? a new conceptualization of perspectives. *arXiv preprint arXiv:1710.00794*, 2017.

[348] Randy Goebel, Ajay Chander, Katharina Holzinger, Freddy Lecue, Zeynep Akata, Simone Stumpf, Peter Kieseberg, and Andreas Holzinger. Explainable ai: the new 42? In *International cross-domain conference for machine learning and knowledge extraction*, pages 295–303. Springer, 2018.

[349] Andreas Holzinger. From machine learning to explainable ai. In *2018 world symposium on digital intelligence for systems and machines (DISA)*, pages 55–66. IEEE, 2018.

[350] Isabelle Guyon, Lisheng Sun-Hosoya, Marc Boullé, Hugo Jair Escalante, Sergio Escalera, Zhengying Liu, Damir Jajetic, Bisakha Ray, Mehreen Saeed, Michèle Sebag, et al. Analysis of the automl challenge series. *Automated Machine Learning*, page 177, 2019.

[351] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[352] Reza Haghpanah, Aniruddha Majumder, Ricky Nilam, Arvind Rajendran, Shamsuzzaman Farooq, Iftekhar A Karimi, and Mohammad Amanullah. Multiobjective optimization of a four-step adsorption process for postcombustion co2 capture via finite volume simulation. *Industrial & Engineering Chemistry Research*, 52(11):4249–4265, 2013.

[353] Peter Fritzson and Peter Bunus. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings 35th Annual Simulation Symposium. SS 2002*, pages 365–380. IEEE, 2002.

[354] Jonathan Grizou, Laurie J Points, Abhishek Sharma, and Leroy Cronin. A curious formulation robot enables the discovery of a novel protocell behavior. *Science advances*, 6(5):eaay4237, 2020.

Appendix A

Dataset Description

This section describes the relevant details regarding the generation and collection of the data for the applications of the multitask DAN2 algorithm.

A.1 The PSA Dataset

Process Description. A pressure swing adsorption (PSA) process is an energy-efficient technology for gas separation. It achieves gas separation by operating a cyclic process where the gas species is absorbed at a higher pressure and released at a lower pressure. The data used in this research adopted a four-stage PSA process model for CO_2 capture as reported in Haghpanah's work [352].

Data generation method. The PSA dataset is a dataset calculated from the software Dymola to simulate the PSA process through a set of differential equations [352] programmed in Modelica [353]. The PSA cycle is simulated iteratively under a series of defined operating conditions. I evaluate the system parameters operating under these conditions until cyclic steady state (CSS) is reached. At cyclic steady state, I evaluate the recovery, purity and energy consumption of the system.

Data description. The PSA dataset contains 6 sets of continuous input features and 3 continuous output values. The inputs are operating conditions (e.g. set points for pressure, duration for adsorption or desorption stages, and inlet flow rate). The outputs are the recovery rate, purity and energy consumption of the system.

Data preprocessing. I apply the normalisation of the data before inputting to the multi-task DAN2 and the ANN. Normalisation is required because the ranges of the inputs are of different scales. Normalisation converts the numeric values in columns to their own column scale, thus equalising the ranges of different inputs without affecting their relative value.

Data Accuracy. The collected data comes from a simulation process which means that the data points collected are highly accurate and to-the-point. Therefore, there is no concern that there might be anomalies in the dataset.

A.2 The OILDROPLET Dataset

Date collection method. The dataset is generated by a high-throughput droplet-generating robot which can execute and record a 90s droplet experiment every 111 seconds, including mixing, syringe-driven droplet placement, recording, cleaning and drying. Further details of the experimental setup can be found in [354].

Data description. The dataset includes 24,422 experimental entries, and within each entry, there are 7 input variables which are the ratios of four oils (diethyl phthalate, 1-octanol, octanoic acid and 1-pentanol) in the droplet, the viscosity, the surface tension and the density of the mixture. The observation space are generated by observing the movement and merging of the oil droplet on the water surface, including average movement speed, maximum speed of a single droplet, average number of droplets in the last second, average number of droplets throughout the experiment.

Data pre-processing. Similar to the prepreocessing step in the PSA dataset, I have applied normalisation to the data before inputting into the neural networks.

Data Accuracy. The dataset comes from experiments which means that there might be collection errors in the process. However, I have not identified anomalies in the datast that might be troublesome to deal with. The data points are accurate to the point that the collection process is carefully conducted and the data are carefully pre-processed.