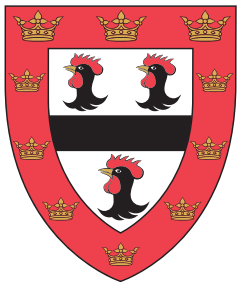
Structural Priors in Deep Neural Networks



Yani Andrew Ioannou

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Jesus College

September 2017

To my mother, Diane, who has tirelessly supported me
for much longer than reasonable without asking:
"When will you graduate?"
At least not very often ...

To Jin-A Lee, who somehow sustained our relationship
from the other side of the world.

Finally, to all my friends, close or far,
especially those in Cambridge/Jesus College,
who have made my time as a PhD student surprisingly enjoyable:

(alphabetically by surname)

Anna Alberio, Bart Andrews, Antonio Criminisi, Demetris Demetriou,
Krittika D'Silva, John Dudley, Frances St George-Hyslop, Simone Hanebaum,
May Hen, Danesh Irani, Jermaine Jiang, Jessica Louise Lindeman, Sebastian Nowozin,
Cai Read, Sohaib Abdul Rehman, Hajime Shinohara, Victoria Tse, Jackson Wo,
Chunwen Xiao, Jenny Yang, and many others!

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Yani Andrew Ioannou
September 2017

Acknowledgements

I have been told the most important single factor in a research career is having the right people as your mentors and, through little doing of my own, I have been extraordinarily fortunate in the people I have had in this role.

I would like to thank my former supervisor at the University of Bath (now Google), Dr. Matthew Brown, for his accepting me into the scholarship that has supported my research, and nurturing the research questions that sparked a fruitful first year of my studies. I thank Dr. Duncan Robertson for somewhat inadvertently continuing in this role, especially for the many discussions we had, in particular his critical analysis, and the resulting insights which are responsible for much of the work I published. I would like to thank Professor Roberto Cipolla for so generously accepting me into his lab at the University of Cambridge, and the support he has shown me ever since. I thank Dr. Richard Turner, also at the University of Cambridge, for agreeing to be my advisor and his time in helping me proofread the first year report. I thank Bart Andrews for his time in proofreading this dissertation. Most of all I would like to thank Dr. Antonio Criminisi at Microsoft Research Cambridge, for his unfailing support beyond the call of a supervisor throughout the course of this degree, and without whom I'm quite sure my PhD would not have been possible.

Much of this work would not have been completed without the collaboration and input of numerous other people I have not listed at Microsoft Research and the University of Cambridge, both of which have more than lived up to their reputations as world class research institutions, but more importantly are immensely friendly environments fostering lasting collaboration and friendships. I'm honoured to have been able to learn in both of these environments.

This dissertation ties together a large body of work completed during my time in Cambridge at both of these institutions, and below I detail collaborators for each of the papers from which this work emanated.

Chapter 2 is based loosely on my contributions to the MSALT4 (Advanced Machine Learning) course report on training deep autoencoders using contemporary deep learning methods, in collaboration with Felix Stahlberg, Moquan Wan and Joseph Zammit.

Chapter 4 presents work presented at the International Conference on Learning Representations (ICLR) 2016, in San Jose, Puerto Rico (Ioannou, Robertson, Shotton, *et al.*, 2016) in collaboration with Duncan Robertson, Jamie Shotton, Roberto Cipolla and Antonio Criminisi.

Chapter 5 presents work presented at the IEEE Conference for Computer Vision and Pattern Recognition (CVPR) 2017, in Honolulu, Hawaii (Ioannou, Robertson, Cipolla, *et al.*, 2017) in collaboration with Duncan Robertson, Roberto Cipolla and Antonio Criminisi.

Chapter 6 is based on work published in a Microsoft Research Technical Report (Ioannou, Robertson, Zikic, *et al.*, 2015) as the result of a 9-month research project, from March–Dec. 2014, at Microsoft Research Cambridge in extensive collaboration with Duncan Robertson, Darko Zikic, Peter Kontschieder, Jamie Shotton, Matthew Brown and Antonio Criminisi. Many of the figures are reproduced here with permission of Antonio Criminisi.

Section 7.1 is based on a research proposal I wrote for a recent fellowship application.

List of Ph.D. Publications

A full list of the papers I published during my Ph.D. is as follows (most recent first):

- Yani Ioannou, Duncan Robertson, Roberto Cipolla, and Antonio Criminisi (2017). "Deep Roots: Improving CNN efficiency with hierarchical filter groups". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Honolulu, HI, USA, July 21–26, 2017), pp. 5977–5986. DOI: 10.1109/CVPR.2017.633. arXiv: 1605.06489

- Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi (2016). "Measuring Neural Net Robustness with Constraints". In: *Advances in Neural Information Processing Systems*. (Barcelona, Spain, Dec. 5–Oct. 12, 2016). Curran Associates, Inc., pp. 2613–2621. arXiv: 1605.07262

- Sukrit Shankar, Duncan Robertson, Yani Ioannou, Antonio Criminisi, and Roberto Cipolla (2016). "Refining Architectures of Deep Convolutional Neural Networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Las Vegas, NV, USA, June 27–30, 2016), pp. 2212–2220. DOI: 10.1109/CVPR.2016.243. arXiv: 1604.06832

- Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi (2016). "Training CNNs with Low-Rank Filters for Efficient Image Classification". In: *International Conference on Learning Representations (ICLR)*. (San Juan, Puerto Rico, May 2–4, 2016). arXiv: 1511.06744

- Yani Ioannou, Duncan Robertson, Darko Zikic, Peter Kontschieder, Jamie Shotton, Matthew Brown, and Antonio Criminisi (2015). *Decision Forests, Convolutional Networks and the Models in-Between*. Tech. rep. MSR-TR-2015-58. Microsoft Research

- Darko Zikic, Yani Ioannou, Antonio Criminisi, and Matthew Brown (2014). "Segmentation of Brain Tumor Tissues with Convolutional Neural Networks". In: *MICCAI workshop on Multimodal Brain Tumor Segmentation Challenge (BRATS)*. (Boston, MA, USA, Sept. 14, 2014)

Abstract

Deep learning has in recent years come to dominate the previously separate fields of research in machine learning, computer vision, natural language understanding and speech recognition. Despite breakthroughs in training deep networks, there remains a lack of understanding of both the optimization and structure of deep networks. The approach advocated by many researchers in the field has been to train monolithic networks with excess complexity, and strong regularization — an approach that leaves much to desire in efficiency. Instead we propose that carefully designing networks in consideration of our prior knowledge of the task and learned representation can improve the memory and compute efficiency of state-of-the art networks, and even improve generalization — what we propose to denote as *structural priors*.

We present two such novel structural priors for convolutional neural networks, and evaluate them in state-of-the-art image classification CNN architectures. The first of these methods proposes to exploit our knowledge of the low-rank nature of most filters learned for natural images by structuring a deep network to learn a collection of mostly small, low-rank, filters. The second addresses the filter/channel extents of convolutional filters, by learning filters with limited channel extents. The size of these channel-wise basis filters increases with the depth of the model, giving a novel sparse connection structure that resembles a tree root. Both methods are found to improve the generalization of these architectures while also decreasing the size and increasing the efficiency of their training and test-time computation.

Finally, we present work towards conditional computation in deep neural networks, moving towards a method of automatically learning structural priors in deep networks. We propose a new discriminative learning model, *conditional networks*, that jointly exploit the accurate representation learning capabilities of deep neural networks with the efficient conditional computation of decision trees. Conditional networks yield smaller models, and offer test-time flexibility in the trade-off of computation *vs.* accuracy.

Table of contents

List of figures

List of tables

Symbols

a net activation of a neuron 11, 12, 13, 14, 15, 16, 21, 25, 27

b bias of a neuron 8, 11, 14, 21

$\mathbf{b}$ vector of biases for a convolutional layer 70, 73

β response/gradient per-layer scaling factor 28, 70

c # (input) channels of a convolutional filter/feature map 23, 24, 66, 70, 73, 98, 137

covar covariance 100, 101

$\circ$ function composition operator 28

$*$ convolution operator 23, 66, 67, 68

d # output channels of a convolutional filter/feature map 70, 71, 73

δ 'local gradient'/'error'/delta 13, 14, 15, 16

e error for an output neuron with a given sample 11, 12, 13, 14, 15, 16

E error function 12, 13, 14, 15, 16, 17, 18, 17, 18, 19, 20, 55, 56

E expected value 28, 71, 100

f activation function of a neuron 8, 11, 12, 13, 14, 15, 16, 20, 21, 27, 70, 71

$\mathbf{F}$ convolutional filter/kernel tensor 23, 24, 25, 66, 67, 68

g group index 71, 73

γ learning rate hyper-parameter 13, 16, 17, 19, 20, 74, 135, 140

h height of a convolutional filter 24, 66, 70, 71, 73

H height of a convolutional feature map 23

$\mathbf{H}$ Hessian matrix of second-order derivatives 18, 56

$\mathcal{L}$ surogate loss function 36, 70

λ weight decay hyper-parameter 74, 135, 140

μ mean 29, 30

O worst case computational complexity 55, 56, 64, 67, 68

σ standard deviation 28, 29, 30, 73

t target label for an output neuron with a given sample 11, 12, 14, 74, 135, 140

t training iteration 19

$\mathbf{v}$ velocity vector, used in momentum 19

Var variance 71

w weight of a neuron 8, 11, 12, 13, 14, 15, 16, 19, 20, 43, 55, 56, 70, 71, 73

$\mathbf{w}$ vector of weights for a neuron, $\mathbf{w} = \{w_0, w_1, \ldots\}$ 8, 11, 19, 21, 43, 55, 56

w width of a convolutional filter 24, 66, 70

W width of a convolutional feature map 23

$\mathbf{W}$ convolutional layer weight matrix 70, 71, 73

x input of a neuron 8, 11, 12, 13, 14, 15, 17, 19, 41, 43, 70, 71

$\mathbf{x}$ vector of inputs to a neuron, $\mathbf{x} = \{x_0, x_1, \ldots\}$ 8, 11, 20, 21, 35, 36, 70, 71, 73, 100

X training set 11, 19, 20, 50, 51, 52, 100, 101

$\mathbf{X}$ incoming convolutional feature map tensor 24, 25, 66, 67, 68

$\mathbf{Y}$ outgoing convolutional feature map tensor 24, 25, 66, 67, 68

$\hat{x}$ normalized response 29, 30

y output of a neuron 8, 11, 12, 13, 14, 15, 16, 17, 21, 30, 70, 71

$\mathbf{y}$ vector of outputs for a single pixel in an output featuremap 70, 71, 73

Acronyms

API Application Programming Interface

BLAS Basic Linear Algebra Subprograms *see Glossary:* BLAS, 74

CIFAR Canadian Institute for Advanced Research *see Glossary:* CIFAR,

CNN Convolutional Neural Network *see Glossary:* CNN, xi, 2, 3, 4, 5, 22, 23, 24, 31, 34, 39, 44, 47, 48, 59, 61, 62, 63, 64, 74, 83, 87, 89, 90, 92, 93, 116, 117, 119, 120, 123, 140, 143, 145, 148, 151, 152

CPU Computer Processing Unit 74, 96, 107, 111, 112, 111, 116, 117, 135, 145

CuBLAS CUDA BLAS *see Glossary:* CuBLAS, 116

CUDA Compute Unified Device Architecture *see Glossary:* CUDA, 116

CVPR Computer Vision and Pattern Recognition 169

DNN Deep Neural Network *see Glossary:* DNN, 1, 3, 4, 5, 7, 17, 18, 19, 25, 27, 30, 31, 48, 52, 53, 57, 89, 93, 94, 107, 117, 143, 146, 147, 148, 149, 154

DOI Digital Object Identifier 167, 169

FLOPS Floating Point Operations 96, 107, 108, 111, 112, 111, 115, 135, 137

GAP Global Average Pooling 33, 32, 76, 123, 133, 137

GMP Global Max Pooling 133, 145

GPU Graphical Processing Unit 25, 32, 74, 75, 90, 96, 107, 111, 112, 111, 116, 117, 133, 145, 147

ICLR International Conference on Learning Representations 169

ILSVRC Imagenet Large-Scale Visual Recognition Challenge 5, 34, 43, 48, 58, 68, 75, 76, 78, 79, 82, 83, 90, 92, 107, 112, 117, 119, 133, 145, 153, 159

LDE Low-Dimensional Embeddings 34

MKL Intel's Matrix Kernel Library *see Glossary:* MKL, 96

MLP Multi-layer Perceptron *see Glossary:* MLP

MNIST Modified National Institute of Standards and Technology *see Glossary:* MNIST, 64, 159

NiN Network in Network *see Glossary:* NiN, 34, 81, 82, 96, 101, 117, 133, 137, 139

OBD Optimal Brain Damage 55

PCA Principled Component Analysis 31, 101

ReLU Rectified Linear Unit 21, 27, 31, 68, 69, 70, 71, 73, 79, 123, 127, 158, 159

RGB Red-Green-Blue 23, 31

RNN Recurrent Neural Network *see Glossary:* RNN, 2, 39

SGD Stochastic Gradient Descent 20, 159, 163

SIFT Scale-Invariant Feature Transform 22, 39, 119

VC Vapnik-Chervonenkis 41, 42, 43, 44

VGG Visual Geometry Group *see Glossary:* VGG, 31, 133, 134, 135

ZCA Zero Component Analysis 96, 100, 101

1

Introduction

> *"A general tabula rasa network is a fine subject for the abstract, formal studies, but one should not try to use it to solve practical problems. ... One should pre-program the network with all available information about the structure of the problem, especially information about the symmetry and topology of the data."*
>
> – John Denker *et al.*, *Large Automatic Learning, Complex Systems, 1987*

Deep learning has in recent years come to dominate the previously separate fields of research in machine learning, computer vision, natural language understanding and speech recognition. The fact that these fields would have been considered relatively distinct less than five years ago belies the power of the theses methods. Deep learning is only the latest in a long history of connectionist learning research, and while the breakthroughs in training DNNs are real, and the research community has continued to discover an increasing number of applications for deep learning, many relatively simple questions of learning in neural networks remain unanswered.

1.1 End-to-End Learning

Classic computer vision approaches depended on hand-designing problem-specific *features* — salient representations of the input — requiring expert knowledge of the domain, thus limiting the scalability and effectiveness of the approach. Early computer vision research, for example, depended on finding prominent edges in images. The often cited appeal of deep learning, as compared to classic feature-driven approaches in computer vision (and speech recognition, etc), is that neural networks are trained 'end-to-end', *i.e.* without needing manual design of internal representations. Whereas in classic computer vision, more salient representations of the input must be hand-

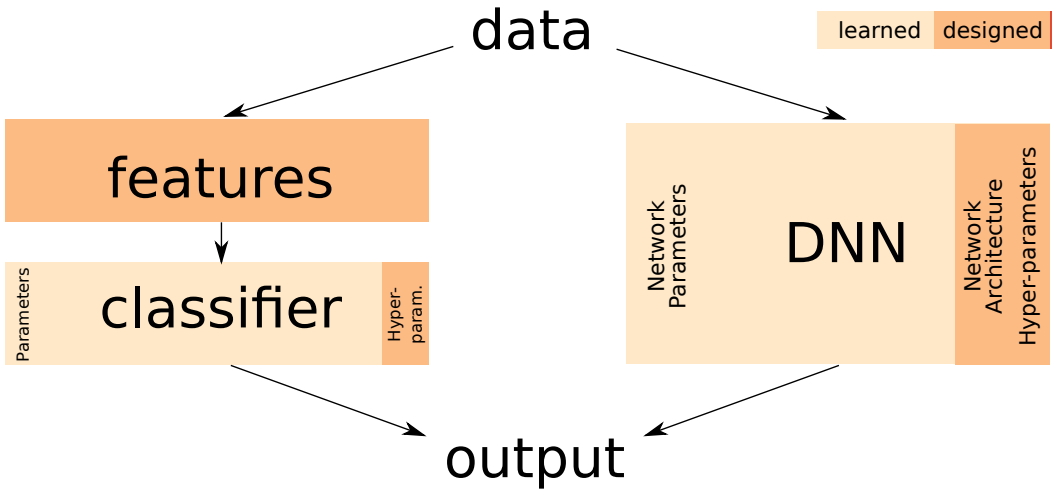


Fig. 1.1 **A comparison of classic feature-based approach to computer vision, and deep learning approaches.** Although both methods require hand-tuned hyper-parameters and design, deep learning methods avoid much of the hand-engineering of classic approaches. Deep networks are far from automatic however, for every application hyper-parameters must be tuned and significant thought has to be put into the design of the structure of the network.

designed for each problem, DNNs can be trained directly on the original data itself, and have been shown to learn better representations of the data.

Glossed over in this description of deep learning, as it is often in the field itself, is that this learning ability does not come for free, and is far from automatic. It relies on very specific assumptions that are mostly encoded in the design of the neural network architecture itself — which, for the sake of brevity, in this dissertation we will denote as *structural priors*[1].

Structural priors require some knowledge of the domain to design but, unlike hand-designed features, are higher level and have relatively broad applicability. In real-world applications, fully-connected neural networks are seldom used. In fact almost every practical application of deep learning has relied on some form of structural prior. Figure 1.1 illustrates, at a high level, the amount of learning *vs.* design in deep learning as compared to traditional feature-based computer vision.

For example, CNNs are a broad class of neural network architectures that encode assumptions based on our prior knowledge about learning with natural image inputs.

[1]I. Goodfellow, Y. Bengio, and Courville (2016) denote this concept "infinitely strong regularization". Unfortunately regularization is often a broadly-used and ill-defined term, in many cases seemingly falling under the definition of 'anything that improves generalization'. Indeed, other recent work (C. Zhang *et al.*, 2017) has struggled with a the general lack of clarity on what regulariation means, given its broad usage. Instead, we define regularization more narrowly (see Glossary).

RNNs are another broad class of architectures that encode our prior knowledge about learning sequence inputs, such as natural language sentences. Neural networks designed with problem-salient structural priors have fewer parameters, are faster, and have better *generalization* — better accuracy on data outside the training set — than the fully-connected equivalent.

1.2 Understanding the Effect of Structure on Learning

Although it has been proven that a neural network with an infinitely wide single hidden layer is a universal approximator (Cybenko, 1989; Hornik, Stinchcombe, and White, 1989), theoretically able to learn any function, in practice the architecture of a neural network has an unreasonably large affect on the generalization of a trained network. For example, it has been well demonstrated that for image classification, a reasonably designed CNN as proposed by LeCun, Bottou, *et al.* (1998), will outperform a fully-connected network, even when that fully-connected network has many more parameters, as will be shown in section 3.2.3. This is despite the fact that, as explained in chapter 2, any CNN can be represented exactly in a (larger) fully-connected network where shared parameters are replaced by duplicated weights and filter connectivity is simply represented by zeroed out weights.

Novel work on optimization algorithms used to train neural networks is considered research progress, while novel structural changes made in DNNs are often dismissed as 'engineering', but this is to fundamentally misunderstand the importance of structure. If a structural change results in lower test loss for the same objective and dataset, then it is moving towards understanding the black box internal representation learned in DNNs.

This lack of understanding in both the optimization and structure of DNNs has meant that contemporary deep network architectures for image classification have high computational and memory complexity. This is a direct result of the inability to identify the optimal architecture for datasets. The approach advocated by many researchers in the field has been to train monolithic networks with excess complexity, and strong regularization — an approach that has found success in practice for accuracy, but leaves much to desire in efficiency.

In this dissertation, we propose that carefully designing networks in consideration of our prior knowledge of the task can improve the memory and computational efficiency of state-of-the art networks, and even increase accuracy through structurally induced

regularization. This is not a completely new idea, and had received focus in an earlier iteration of connectionist research progress, in the late 80́s and early 90́s — CNNs are perhaps the most successful example of this approach.

1.3 Contributions

While this philosophy defines our approach, deep neural networks have a large number of degrees of freedom, and there are many facets of deep neural networks that warrant such analysis. In this dissertation, we will look at several interrelated aspects of DNN, for each of which we have made a novel contribution:

(i) **Spatial connectivity (chapter 4):** a novel spatial structural prior for CNNs, by structuring the network to learn a mixed low/full rank basis for filters,

(ii) **Inter-filter connectivity (chapter 5):** a novel inter-filter structural prior for DNNs, by structuring the network to learn a reduced inter-layer filter/neuron connectivity,

(iii) **Conditional connectivity (chapter 6):** a novel method of learning conditional routing inside a DNN, and using this conditional routing to reduce computation at test time.

So as to not conflate the methods and results, we have presented each of these in isolation in this dissertation. However, the methods address different aspects of sparsity in DNNs, and we believe them to be complementary.

1.4 Organization

This dissertation is organized as follows:

Chapter 2 gives the necessary background on neural networks, and contemporary deep learning architectures with particular emphasis on image classification, in order to understand our work and motivation.

Chapter 3 explores the effect of structure on learning in neural networks, in particular machine learning theory and early empirical results towards understanding the effect of the size and structure of neural networks on their generalization, motivating the focus of this dissertation.

Chapter 4 addresses the spatial extents of convolutional filters, proposing to exploit our knowledge of the low-rank nature of most filters learned for image recognition by structuring a deep network to learn a mixed full and low-rank basis for filters.

Rather than approximating filters in previously-trained networks with more efficient versions, we learn a set of small basis filters from scratch; during training, the network learns to combine these basis filters into more complex filters that are discriminative for image classification. To train such networks, a novel weight initialization scheme is used. This allows effective initialization of connection weights in convolutional layers composed of groups of differently-shaped filters.

We validate our approach by applying it to several existing CNN architectures and training these networks from scratch using the CIFAR, ILSVRC and MIT Places datasets. Our results show similar or higher accuracy than conventional CNNs, while requiring much less computation, and many fewer parameters.

Chapter 5 addresses the filter/channel extents of convolutional filters, by learning filters with limited channel extents.

A new method is proposed for creating computationally efficient and compact CNNs using a novel sparse connection structure that resembles a tree root. This allows a significant reduction in computational cost and number of parameters of state-of-the-art deep CNNs without compromising accuracy. We validate our approach by using it to train more efficient variants of state-of-the-art CNN architectures, evaluated on the CIFAR-10 and ILSVRC datasets. Our results show similar or higher accuracy than the baseline architectures with much less computation and many fewer parameters.

Chapter 6 presents work towards conditional computation in deep neural networks, allowing for faster inference by understanding the connections between two state-of-the-art classifiers: DNNs and random decision forests.

We propose a new discriminative learning model, *conditional networks*, that jointly exploits the accurate *representation learning* capabilities of deep neural networks with the efficient *conditional computation* of decision trees and directed acyclic graphs (DAGs). Conditional networks can be thought of as a way to learn a block-diagonal sparsification of a DNN, and we show how they can be trained to cover the continuous spectrum between DNNs and decision forests/jungles.

In addition to improving test and training efficiency, conditional networks yield smaller models, and offer test-time flexibility. Validation is performed on stan-

dard image classification tasks. Compared to the state-of-the-art, our results demonstrate superior efficiency for at-par accuracy both on the ImageNet and CIFAR datasets.

Chapter 7 summarizes the contributions made in this dissertation, reviews the results, and looks at the research questions that arise from the work presented in this dissertation. In particular the importance of the significant hand-design still present in "end-to-end" learning, the effectiveness of structural priors, and the ineffectiveness of current optimization methods are discussed. Proposals are made for future research directions which are pertinent given our results.

2

Background

"One thing that connectionist networks have in common with brains is that if you open them up and peer inside, all you can see is a big pile of goo."
– Mozer *et al.*, *Using Relevance to Reduce Network Size Automatically, 1989*

Somewhat like their biologically inspiration, artificial neural networks (or simply *neural networks*) are easier to understand at the low-level — the individual neuron itself lacks in complexity — it is in the whole that it becomes hard to understand. While we will focus on the latter in the remainder of this dissertation, here we will present the basics behind the structure and training of neural networks and contemporary DNNs.

2.1 Neural Networks

Neural networks are a broad range of statistical models characterized by consisting of a set of inter-connected nodes with non-linear *activation functions* with learnable parameters, or *weights*. Although initially biologically inspired, neural networks within the field of machine learning have moved away from biologically-plausible models and towards entirely practical models guided by empirical results. Neural networks are now the most popular statistical model used for learning applications in a diverse set of fields including computer vision, speech recognition, and medical imagery.

We will not cover the long and colourful history of neural networks (for this we recommend reading the introduction of I. Goodfellow, Y. Bengio, and Courville (2016)), but attempt to instead provide the foundations, and thereafter an overview of contemporary models and methods directly relevant to this work. For a comprehensive overview of the basics of neural networks we refer the reader to the excellent reference

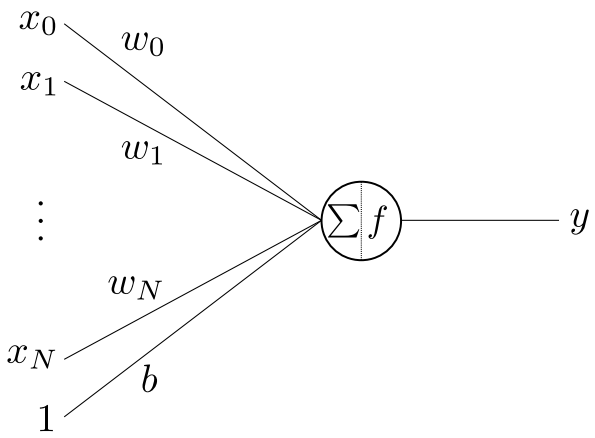


Fig. 2.1 **An illustration of a typical artificial neural network *neuron*.** The neuron is composed of output y, a set of inputs, $\{x_0, \ldots, x_N\}$, input weights $\{w_0, \ldots, w_N\}$, bias b and activation function f. Here the bias b is considered the weight of a fixed unit input.

of Bishop (1995), and for a more recent in-depth overview of the field of *deep learning*, we refer the reader to I. Goodfellow, Y. Bengio, and Courville (2016).

2.1.1 The Neuron

A neuron is a function of the weighted aggregation of its many inputs: $\{x_0, \ldots, x_N\}$,

$$y = f\left(\sum_i^N w_i x_i + b\right), \tag{2.1}$$

where w_i is the weight of the input x_i, f is an *activation function*, and b is the *bias*. This is usually expressed more simply in matrix notation, where each neuron consists of an input vector $\mathbf{x} = (x_0, \ldots, x_N)$, weights $\mathbf{w} = (w_0, \ldots, w_N)$ and a bias b, the output of which is,

$$y = f\left(\mathbf{w}^T\mathbf{x} + b\right). \tag{2.2}$$

If we assume the function f to be a variant of the Heaviside step function,

$$f(x) = \begin{cases} 1 & \text{if } x \geqslant 0 \\ 0 \text{ (or } -1) & \text{if } x < 0, \end{cases} \tag{2.3}$$

then the neuron is also called a *perceptron*, a simple binary classifier, and one of the earliest connectionist learning methods, invented by Rosenblatt (1958) in 1957. A perceptron network is a single-layer neural network (*i.e.* a linear classifier), such as

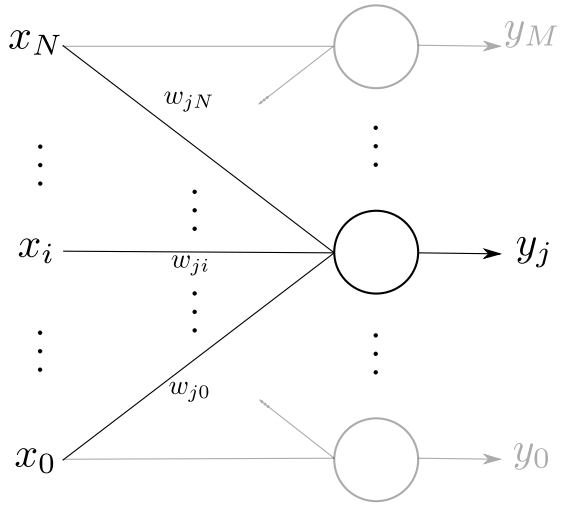


Fig. 2.2 **An illustration of a simple single-layer neural network, such as a perceptron network.**

that illustrated in fig. 2.2, and should not to be confused with a MLP (Multi-layer Perceptron) — an unfortunate, but common, misnomer of any multi-layer neural network. The perceptron also has a geometric interpretation (Bishop, 1995), as shown in fig. 2.3. In 2D, for example, this is equivalent to the equation of the line. Assuming our perceptron only has a single input, *i.e.* $N = 1$, if we define $a \equiv w_0$, $x \equiv x_0$, $c \equiv b$, and $f(x) = x$,

$$y = ax + c. \tag{2.4}$$

In general a perceptron defines a *hyperplane*, a separating manifold of dimension $d - 1$ for an input space of dimension d, a line in two dimensions, or plane in three dimensions. Neural networks are a discriminative classifier, and each neuron can be visualized as a hyperplane functioning as a single decision boundary.

2.1.2 The Limitations of Single-Layer Networks

A single-layer neural network, such as the perceptron, is only a linear classifier, and as such is ineffective at learning a large variety of tasks. Most notably, in the 1969 book *Perceptrons* (Minsky and Papert, 1988), the authors showed that single-layer perceptrons could not learn to model functions as simple as the XOR function, amongst other non-linearly separable classification problems. As shown in fig. 2.4, no single line can separate even a sparse sampling of the XOR function — *i.e. it is not linearly*

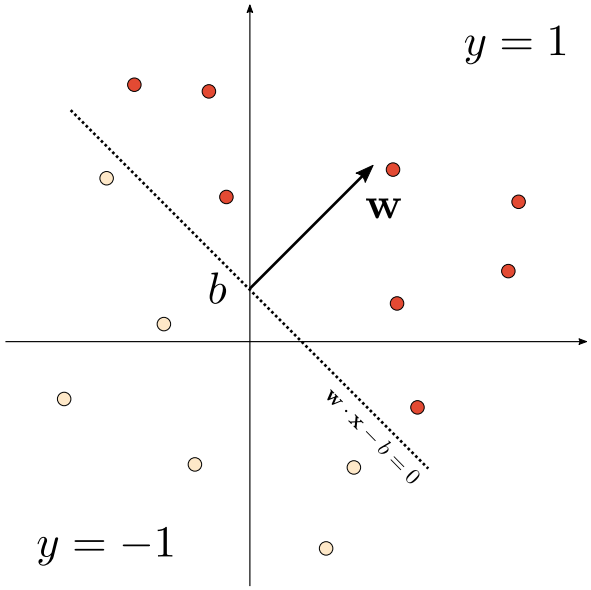


Fig. 2.3 **The interpretation of a perceptron as a oriented hyperplane**, *i.e.* a line, in $\mathbb{R}^2$.

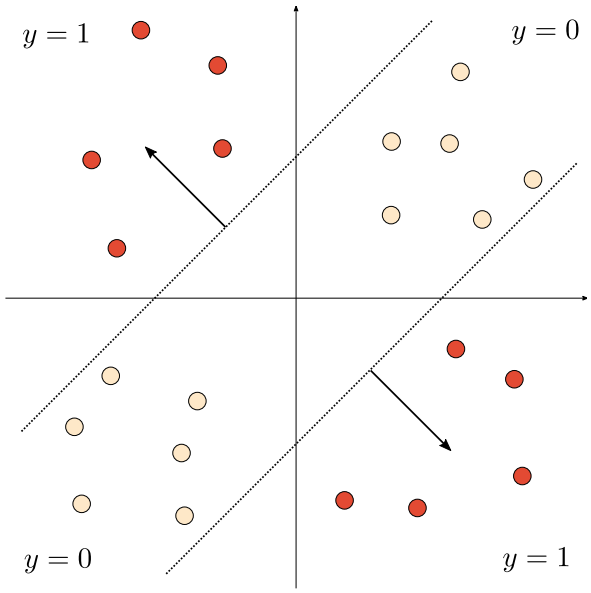


Fig. 2.4 **An illustration of the inability of a single line (*i.e.* a perceptron) to correctly classify the XOR function.** Instead, the composition of two lines is required to correctly separate these samples, *i.e.* multiple layers.

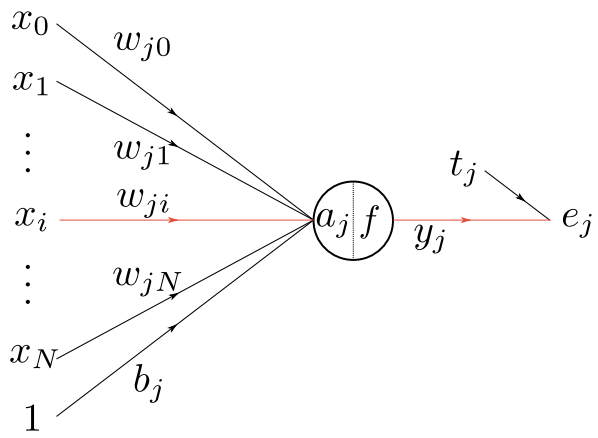


Fig. 2.5 **Detailed illustration of a single-layer neural network trainable with the delta rule.** The input layer consists of a set of inputs, $\{x_0, \ldots, x_N\}$. The layer has weights $\{w_{j0}, \ldots, w_{jN}\}$, bias b_j, net neuron activation $a_j = \sum_i w_{ji}$, activation function f, and output y_j. The error for output y_j, e_j, is calculated using the target label t_j.

separable. Instead, only a composition of lines is able to correctly separate and classify this function, and other non-linearly separable problems.

At the time, it was not obvious how to train networks with more than one layer of neurons, since the methods of learning neuron weights, the *perceptron learning rule* (Rosenblatt, 1961) for perceptrons or the *delta rule* (Widrow and Hoff, 1960) for general neurons, only applied to single-layered networks.

2.1.3 Training Single Layer Networks: The Delta Rule

The delta rule for single-layered neural networks is a gradient descent method, using the derivative of the network's weights with respect to the output error to adjust the weights to better classify training examples.

Training is performed on a training dataset X, where each training sample $\mathbf{x}^n \in X$ is a vector $\mathbf{x}^n = (x_0^n, \ldots, x_N^n)$. Assume that for a given training sample $\mathbf{x}^n$, the ith neuron in our single-layer neural network has output y_j^n, target (desired) output t_j^n, and weights $\mathbf{w} = (w_{j0}, \ldots, w_{jM})$, as shown in fig. 2.5. We can consider the bias to be an extra weight with a unit input, and thus we can omit the explicit bias from the derivation.

We want to know how to change a given weight w_{ji} given the output of node j for a given input data sample $\mathbf{x}^n$,

$$y_j^n = f\left(a_j^n\right), \tag{2.5}$$

where the net activation a_j^n is,

$$a_j^n = \sum_i w_{ji} x_i^n. \tag{2.6}$$

To do so, we must use the error of our prediction for each output y_j and training sample x^n as compared to the known label t_j,

$$e_j^n = y_j^n - t_j^n. \tag{2.7}$$

For this derivation, we assume the error for a single sample is calculated by the sum of squared errors of each output. In fact, the derivation holds as long as our error function is in the form of an average (Bishop, 1995),

$$E^n = \frac{1}{2} \sum_j \left(e_j^n \right)^2. \tag{2.8}$$

The chain rule allows us to calculate the *sensitivity* of the error to each weight w_{ji} in the network,

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial e_j^n} \frac{\partial e_j^n}{\partial y_j^n} \frac{\partial y_j^n}{\partial a_j^n} \frac{\partial a_j^n}{\partial w_{ji}}. \tag{2.9}$$

Differentiating eq. (2.8) with respect to e_j^n,

$$\frac{\partial E^n}{\partial e_j^n} = e_j^n, \tag{2.10}$$

eq. (2.7) with respect to y_j^n,

$$\frac{\partial e_j^n}{\partial y_j^n} = 1, \tag{2.11}$$

eq. (2.5) with respect to a_j^n,

$$\frac{\partial y_j^n}{\partial a_j^n} = f' \left(a_j^n \right), \tag{2.12}$$

and finally eq. (2.6) with respect to w_{ji},

$$\begin{aligned} \frac{\partial a_j^n}{\partial w_{ji}} &= \frac{\partial}{\partial w_{ji}} \left(\sum_i w_{ji} x_i \right) \\ &= x_i, \end{aligned} \tag{2.13}$$

since only one of the terms in the sum is related to the specific weight w_{ji}. Thus the sensitivity is,

$$\frac{\partial E^n}{\partial w_{ji}} = e_j^n f'\left(a_j^n\right) x_i. \tag{2.14}$$

Typically what is variously called the local gradient, error, or simply *delta*, is then defined,

$$\begin{aligned} \delta_j^n &\equiv \frac{\partial E^n}{\partial a_j^n} \\ &= \frac{\partial E^n}{\partial e_j^n}\frac{\partial e_j^n}{\partial y_j^n}\frac{\partial y_j^n}{\partial a_j^n} \\ &= e_j^n f'\left(a_j^n\right), \end{aligned} \tag{2.15}$$

such that eq. (2.14) can be rewritten,

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j^n x_i. \tag{2.16}$$

The delta rule adjusts each weight w_{ji} proportional to the sensitivity,

$$\Delta w_{ji} = -\gamma \frac{\partial E^n}{\partial w_{ji}}, \tag{2.17}$$

where γ is a constant called the *learning rate* or *step size*. Using the delta defined in eq. (2.15), this is simply written,

$$\Delta w_{ji} = -\gamma \delta_j^n x_i. \tag{2.18}$$

2.1.4 Backpropagation

The credit-assignment problem was solved with the discovery of *backpropagation* (also known as the *generalized delta rule*), allowing learning in multi-layer neural networks. It is somewhat controversial as to who first 'discovered' backpropagation, since it is essentially the application of the chain rule to neural networks, however it's generally accepted that it was first demonstrated experimentally by Rumelhart, Geoffrey E. Hinton, and Williams (1986). Although it is "just the chain rule", to dismiss this first demonstration of backpropagation in neural networks is to understate the importance of this discovery to the field, and to dismiss the practical difficulties in first implementing the algorithm — a fact that will be attested to by anyone who has since attempted.

The following is a derivation of backpropagation loosely based on the excellent references of Bishop (1995) and Haykin (1994), although with different notation. This

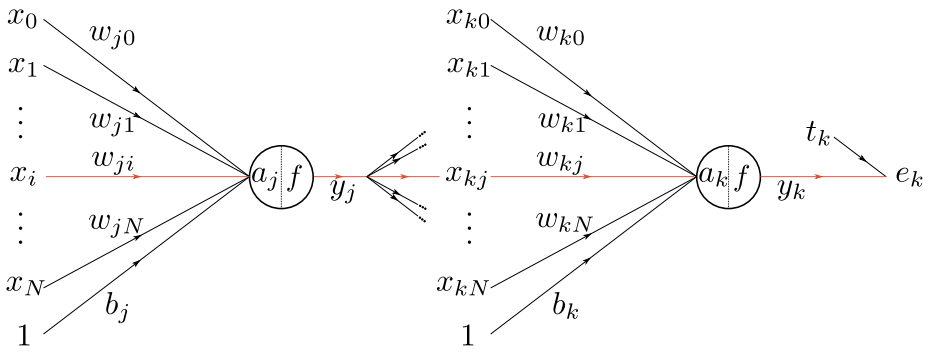


Fig. 2.6 **Detailed illustration of a neural network with a single hidden layer.** The input layer consists of a set of inputs, $\{x_0, \ldots, x_N\}$. The hidden layer has weights $\{w_{i0}, \ldots, w_{iN}\}$, bias b_i, net neuron activity $a_j = \sum_i w_{ji}$ and activation function f. The output layer with output y_k, has weights $\{w_{k0}, \ldots, w_{jN}\}$ and bias b_j. The error for output y_k, e_k, is calculated using the target label t_k. Note that $x^n_{kj} \equiv y^n_j$.

derivation builds upon the derivation for the delta rule in the previous section, although it is important to note that, as shown in fig. 2.6, the indexing we will use to refer to neurons of different layers differs from that in fig. 2.5 for the single-layer case.

We are interested in finding the sensitivity of the error E to a given weight in the network w_{ij}. There are two classes of weights for which we must derive different rules, (**i**) those belonging to *output layer neurons*, *i.e.* neurons lying directly before the output, such as w_{kj} in fig. 2.6, and (**ii**) weights belonging to hidden layer neurons, such as w_{ji} in fig. 2.6.

(i) Output Layer The output weights are relatively easy to find, since they correspond to the same types of weights found in single-layer networks, and have direct access to the error signal, *i.e.* e^n_j. Indeed the derivation in section 2.1.3 also describes the sensitivity of the weights in the output layer of a multi-layer neural network. With some change of notation (now indexing by k rather than j to match fig. 2.6), we can use the sensitivity found in eq. (2.14),

$$\begin{aligned} \frac{\partial E^n}{\partial w_{kj}} &= \frac{\partial E^n}{\partial a^n_k} \frac{\partial a^n_k}{\partial w_{kj}} \\ &= \delta^n_k x^n_{kj} \\ &= \delta^n_k y^n_j. \end{aligned} \tag{2.19}$$

(ii) Hidden Layer We will first derive the partial derivative $\partial E^n / \partial w_{ji}$, for a single hidden layer network, such as that illustrated in fig. 2.6. Unlike in the first case, the weights belonging to hidden neurons have no direct access to the error signal, instead

we must calculate the error signal from all of the neurons that indirectly connect the neuron to the error (*i.e.* every output neuron y_k).

Following from the chain rule we can write the partial derivative of a hidden weight w_{ji} with respect to the error E^n,

$$\frac{\partial E^n}{\partial w_{ji}} = \underbrace{\left(\sum_k \frac{\partial E^n}{\partial e_k^n} \frac{\partial e_k^n}{\partial y_k^n} \frac{\partial y_k^n}{\partial a_k^n} \frac{\partial a_k^n}{\partial y_j^n} \right)}_{\text{output neurons}} \underbrace{\frac{\partial y_j^n}{\partial a_j^n} \frac{\partial a_j^n}{\partial w_{ji}}}_{\text{hidden neuron}}, \tag{2.20}$$

where the sum arises from the fact that, unlike in eq. (2.13) where the weight w_{kj} affects only a single output, the hidden weight w_{ji} affects all neurons in the subsequent layer (see fig. 2.6).

We already know how to calculate the partials for the output layer from the derivation of the delta rule for single-layer networks, and we can substitute these from eq. (2.19) for the output neuron and error partial derivatives,

$$\frac{\partial E^n}{\partial w_{ji}} = \left(\sum_k \delta_k^n y_j^n \frac{\partial a_k^n}{\partial y_j^n} \right) \frac{\partial y_j^n}{\partial a_j^n} \frac{\partial a_j^n}{\partial w_{ji}}. \tag{2.21}$$

Recall from eq. (2.6), the net activation a is a sum of all previous layer weights. Thus,

$$\frac{\partial a_k^n}{\partial y_j^n} = \frac{\partial}{\partial y_j^n} \left(\sum_j w_{kj} y_j^n \right) = w_{kj}, \tag{2.22}$$

and substituting from eq. (2.12) and eq. (2.13) into eq. (2.21),

$$\frac{\partial E^n}{\partial w_{ji}} = \left(\sum_k \delta_k^n y_j^n w_{kj} \right) f' \left(a_j^n \right) x_i. \tag{2.23}$$

This bears some resemblance to the derived expression for a single-layer, and just as in eq. (2.15), we can use our definition of the delta to simplify it. For hidden layers this evaluates as

$$\begin{aligned} \delta_j^n &\equiv \frac{\partial E^n}{\partial a_k^n} \\ &= \left(\sum_k \frac{\partial E^n}{\partial e_k^n} \frac{\partial e_k^n}{\partial y_k^n} \right) \frac{\partial y_k^n}{\partial a_k^n} \\ &= \left(\sum_k \delta_k^n y_j^n w_{kj} \right) f' \left(a_j^n \right). \end{aligned} \tag{2.24}$$

This leaves us with the more convenient expression (as we will see in section 2.1.4),

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j^n x_i. \tag{2.25}$$

Arbitrary Number of Hidden Layers

The derivation above was based on the specific case of a single hidden layer network, but it is trivial to extend this result to multiple hidden layers. There is a recursion in the calculation of the partial derivatives in eq. (2.24) which holds for a network with any number of hidden layers, and which we will now make explicit.

The delta is defined,

$$\delta_i^n = \begin{cases} f'\left(a_j^n\right) e_j^n & \text{when neuron } j \text{ is output} \\ f'\left(a_j^n\right) \left(\sum_j \delta_j^n y_i^n w_{ji}\right) & \text{when neuron } j \text{ is hidden,} \end{cases} \tag{2.26}$$

for any adjacent neural network layers i, j, including the output layer where the outputs are considered to have an index j. The sensitivity is then,

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j^n y_i. \tag{2.27}$$

2.1.5 Learning with Backpropagation

Learning with backpropagation is much like the delta rule; sensitivities are used to correct weights proportional to a constant *learning rate* or *step size* parameter γ. Although the correction is proportional to the sensitivity, we wish to *reduce* the error E^n, and so we move the weight in the opposite direction of the gradient [1]. Formally, the weight change rule is given by,

$$\begin{aligned} \Delta w_{ij}^n &= -\gamma \frac{\partial E^n}{\partial w_{ji}} \\ &= -\gamma\, \delta_j^n y_i, \end{aligned} \tag{2.28}$$

where δ_j^n is as defined in eq. (2.26), and y_i is the output of neuron i. Backpropagation is a method of steepest descent. This is illustrated in fig. 2.7, where the backpropagation learning rule, eq. (2.28), specifies a step size in the form of the *learning rate*. The learning rate parameter scales the step size, or the magnitude of the weight change

[1] Note that, rather than optimizing the error function directly, usually a surrogate loss that is easier to optimize is used, *e.g.* negative log-likelihood for classification.

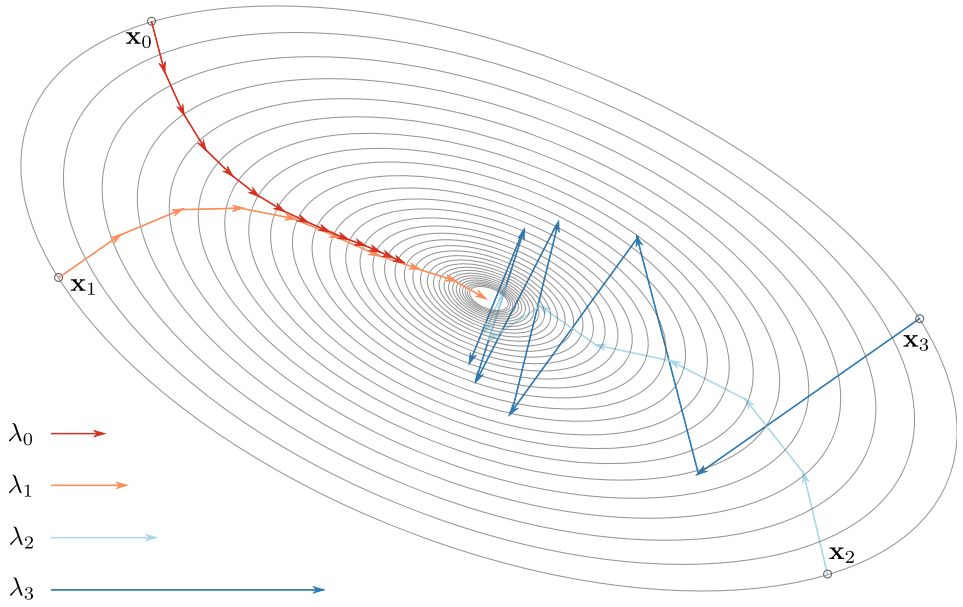


Fig. 2.7 **An illustration of the effect of step size (learning rate) and learning policy on convergence with backpropagation.** This example is of a symmetric 2D error surface, where the parameters are initialized to one of the symmetrically identical surface points x_i where $i = 0 \ldots 4$. For each of the different initial learning rates γ_i, the learning rate is decreased by 10% each iteration.

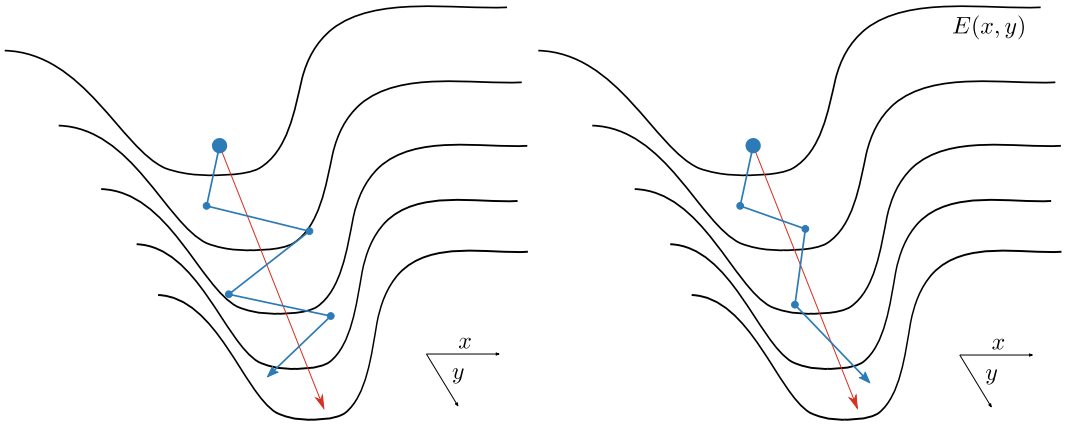
vector. Figure 2.7 also illustrates the effect of learning rate on gradient descent. Too small a learning rate can result in very slow learning such as for γ_0, while too large a step size can result in bouncing around the minima (γ_2, γ_3), or missing it altogether.

In order to settle into a local minima, the learning rate must also be decreased as training progresses. However, too fast a rate of decrease and it may never reach the basin of attraction of the local minima, as with γ_0, while if the rate of decrease is too slow it will take a very long time to enter the basin of attraction, such as with γ_3.

The balance of trying to find an appropriate learning rate and learning policy is unfortunately part of the 'black magic' behind training DNNs which comes from experience, but Bottou (2012) and I. Goodfellow, Y. Bengio, and Courville (2016) are excellent references on some of the common approaches taken to make this task simpler.

2.1.6 The Problem with First-Order Optimization

The underlying reason learning rate and learning policy has such a large effect is that gradient descent is a *first-order* optimization method, and only considers the first-order partial derivatives, *i.e.* for a 2D error surface $E(x, y)$, gradient descent moves in the



(a) Gradient descent (b) Momentum

Fig. 2.8 **Pathological curvature.** An error surface $E(x, y)$ exhibiting a narrow valley, and the optimal path from the starting point to the minima shown by the red arrow. In a pathological error surface such as this, first-order methods cannot use the information provided by the Hessian on the surface curvature to avoid bouncing along the walls of the valley, slowing descent. Momentum alleviates this somewhat in damping the change in direction, by preserving information on previous gradients, allowing a quicker descent. Inspired by a similar diagram by Martens (2010).

opposite direction of the gradient,

$$\nabla E(x, y) = \left(\frac{\partial E}{\partial x}, \frac{\partial E}{\partial y} \right). \tag{2.29}$$

This gradient tells us the direction of maximum increase at a given point on the error surface, but it does not tell us any information about the *curvature* of the surface at that point. The curvature of the surface is described by higher-order derivatives such as the second-order partial derivatives, *e.g.* $\frac{\partial^2 E}{\partial x^2}$, and mixed partial derivatives, *e.g.* $\frac{\partial^2 E}{\partial x^2 \partial y^2}$. These second-order partials give important information about the curvature of the error surface E. For example, in fig. 2.7, the error surface takes on an elliptical shape, which causes problems when we only consider the direction of maximum decrease $-\nabla E$. The classic example of such a pathological error surface for first-order methods is an error surface that looks like a narrow valley, as shown in fig. 2.8(a). With an initialization outside the bottom of the valley, gradient descent will bounce along the walls of the valley, leading to a very slow learning convergence.

For well-behaved surfaces where the scaling of parameters is similar, basins of attraction around a minima are roughly circular, and thus avoid this problem, since the first-order gradients will point almost directly at the minima for any location on the error surface.

There are second-order optimization methods based on Newton's method, however the issue is that they do not scale to the size of any practical DNNs. The matrix of second-order partial derivatives for a scalar-values function, the Hessian $\mathbf{H}$, is required for any full second-order optimization method, however the Hessian is square in the number of parameters in the network. For networks of millions of parameters this means storing the Hessian is infeasible.

There are a whole slew of optimization tricks for gradient descent, often attempting to compensate for the shortcomings of first-order optimization without using the Hessian, or using some approximation to it. We will not cover those here, since none of these were used in our experiments. A full background of the issues of optimization in DNNs is outside the scope of this dissertation, however interested readers should refer to I. Goodfellow, Y. Bengio, and Courville (2016) to learn more about these methods, and Martens (2010) for an excellent introduction to the problems of first and second-order optimization in DNNs.

2.1.7 Momentum

A common improvement to gradient descent is momentum (Polyak, 1964; Rumelhart, Geoffrey E. Hinton, and Williams, 1986), a trick for minimizing the effect of pathological curvature on gradient descent, which also helps with variance in gradients. The name comes from the analogy of the update to physical momentum ρ for a moving particle, $\mathbf{p} = m\mathbf{v}$, where we assume unit mass, $m = 1$.

In momentum, the gradients over multiple iterations are accumulated into a velocity gradient,

$$\begin{aligned} \mathbf{v}_{t+1} &= \alpha \mathbf{v}_t - \gamma \nabla E(\mathbf{w}) \\ \Delta \mathbf{w} &= \mathbf{w}_t + \mathbf{v}_{t+1}, \end{aligned} \tag{2.30}$$

where γ is the learning rate, t is the iteration, ∇E is the gradient of the error surface $E(\mathbf{w})$ being minimized, and $\mathbf{w}$ is the weight vector optimized. Momentum in effect stores some information on the gradients found in past iterations, and uses this to damp the effect of a new gradient on the search direction, as illustrated in fig. 2.8(b). For error surfaces with pathological curvatures, this can dramatically speed up learning.

2.1.8 Batch and Stochastic Gradient Descent

Although the backpropagation weight change rule, eq. (2.28), tells us how to change the weights given a single training sample x^n, in practice this method is rarely used. The reason is simply that the gradients from a single sample are too biased, or noisy, and

they are not representative of the dataset in general; Δw_{ij}^n is only an approximation to the true gradient we want — it is only from one sample, x^n, of the training dataset X.

Batch Gradient Descent

At the opposite end of the spectrum there is *batch* training where the gradient is computed over all data samples in the training set,

$$\Delta w_{ij} = -\gamma \frac{1}{N} \sum_{n=0}^{N} \frac{\partial E^n}{\partial w_{ji}}, \tag{2.31}$$

where N is the number of training samples in X. Batch training gives us the true gradient, however it is also very expensive, since it requires us to perform the forward pass of the network over all training samples for every update.

Stochastic Gradient Descent

Instead of computing the gradient on only one training sample, or over the entire training set, we might instead use a significant subset of the training set — a *mini-batch*. This approach is called SGD.

When using SGD, we randomly sample (without replacement) a subset of the training set $X_{\text{mb}} \subset X$, such that

$$\Delta w_{ij} = -\gamma \frac{1}{|X_{\text{mb}}|} \sum_{\{n|\,\mathbf{x}^n \in X_{\text{mb}}\}} \frac{\partial E^n}{\partial w_{ji}}, \tag{2.32}$$

where the size of the mini-batch $|X_{\text{mb}}|$ should be significant enough to represent the statistics of the training set distribution, *i.e.* for a classification problem the mini-batch should capture a significant number of the classes in the training set. Using a mini-batch size of one, *i.e.* a single sample as shown in eq. (2.28), is a special case of SGD.

It has been observed in practice that adding noise to the gradient by using stochastic gradient descent often helps generalization compared to batch gradient descent, perhaps by preventing overfitting. Note that even if we use the true gradient for the training dataset, the training set X is only a sampling of the population distribution we want our network to generalize to.

2.1.9 Activation Functions

As we have seen with the perceptron, the activation function for a single-layer network can provide a means of pushing the outputs of each neuron towards a binary classification. However the activation function has a much more important function in multi-layer neural networks. Without a non-linear activation function, even a large multi-layer neural network would only have the representational power of a linear classifier — the composition of linear functions is a linear function. For this reason, the *activation function* f is a non-linear function applied to the output of a neuron to allow multi-layer networks to learn complex non-linear functions,

$$y = f\left(\mathbf{w}^T\mathbf{x} + b\right). \tag{2.33}$$

In the field of neural networks, activation functions have classically been chosen to be a *sigmoid* function, *i.e.* a function mapping negative inputs to negative outputs and positive inputs to positive outputs with a smooth transition around $a = 0$. This is a nice property to have, since the function is still pushing the outputs of the network towards a binary classification, the function is non-linear (so composition of functions are non-trivial), and the function has well-defined gradients. Examples of sigmoid functions commonly used include the logistic function,

$$f(a) = \frac{1}{1 + e^{-a}}, \tag{2.34}$$

and they hyperbolic tangent,

$$f(a) = \tanh(a). \tag{2.35}$$

An issue with sigmoidal activation functions however, is that the gradients are very small in a large part of the domain of the function. For this reason, and improved empirical results, modern neural networks tend to use the ReLU activation function, as described in section 2.2.1.

2.1.10 Deep *Vs.* Shallow Neural Networks

Neural networks with at least one (infinitely wide) *hidden* layer have been proven to be a universal approximator — *i.e.* such a neural network can theoretically represent any function (Cybenko, 1989; Hornik, Stinchcombe, and White, 1989). This is in stark contrast to the limitations of neural networks without hidden layers, as explained in section 2.1.2.

In practice however we do not find that a network with only a single hidden layer, of even a very large width, can learn to represent complex functions as well as networks with many hidden layers. Indeed the observation that networks with many hidden layers, or *deep* networks, empirically achieve better accuracy than networks with few hidden layers, or *shallow* networks, represents some of the progress made in learning with neural networks in recent years.

2.1.11 Convolutional Neural Networks

The earliest work on what are now termed CNNs was by Fukushima, on the *Neocognitron* (Fukushima, 1980, 2013). The Neocognitron was a biologically motivated architecture, motivated by what are typically called simple and complex cells in the primary visual cortex (V1). To model simple cells; cells whose response correlated with simple oriented edges in a translation invariant manner, the Neocognitron used shared weights which were connected to local image patches of the input image (and were not simply described as convolution of a filter). Complex cells were modelled by a "blurring" operation, what we now term more generally as *pooling*. The Neocognitron network consisted of alternating layers of simple and complex cells, *i.e.* alternating convolution and pooling layers, much as seen in state-of-the-art convolutional networks. In an era in which fully-connected networks were used to learn any input type Fukushima showed that for structured inputs a drastically different architecture could make a big difference in generalization.

Despite the pioneering novelty of the work on Neocognitron, it was only following the simplification and improvement of LeCun, Bottou, *et al.* (1998) in both the description of the network and its operation that it gained wider acknowledgement as a breakthrough for image recognition. In their work the local shared weights of the Neocognitron are put in the context of convolution, and the averaging operation replaced with max-pooling. The application to handwritten digit recognition gave state-of-the-art results, and would result in the *LeNet5* network, still used today in some commercial applications.

The application of the LeNet-style CNN architecture to more complex problems, however, proved infeasible (I. Goodfellow, Y. Bengio, and Courville, 2016). These problems required a deeper hierarchy of representation, which implied a large number of layers. Networks with a large number of layers proved to be un-trainable due to numerous issues with the model itself, notably vanishing gradients (Hochreiter, 1991), and the lack of large datasets and computational power at the time. Convolutional neural networks fell out of favour, and were passed over in favour of the more successful

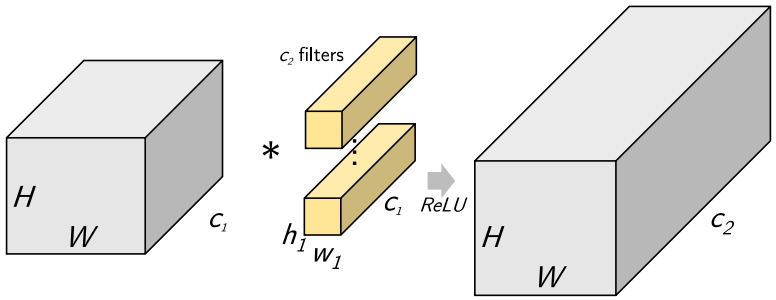


Fig. 2.9 **Convolution with c_2 filters of shape $h_1 \times w_1 \times c_1$.** Convolutional filters (yellow) have the same channel dimension c_1 as the input feature maps (gray) on which they operate, while the feature map spatial dimensions H and W are typically larger. Each filter is convolved across the entire set of input feature maps to produce a single output feature map. With c_2 filters, this gives an output feature map with c_2 channels. In this illustration we assume padding appropriate to preserve the spatial dimensions.

paradigm of using hand-crafted local features, such as SIFT (Lowe, 2004) for many tasks, and in particular the problem of object instance recognition was well addressed by such solutions. Meanwhile object class recognition remained a difficult problem, for which the best solutions were deformable parts models, also based on local features.

Convolutional Layers

Figure 2.9 illustrates a typical convolutional layer. If we denote the d^{th} feature map for the given layer as h^d, where the associated filter has weights $\mathbf{F}^d$, bias b_d, and activation function f, then the single *pixel* of the feature map h^d at spatial location i, j is given by,

$$h^d_{i,j} = f\left(\left(\mathbf{F}^d * x\right)_{i,j} + b_d\right), \tag{2.36}$$

where $*$ is the convolution operator. The discrete convolution operator $(f*g)$ is defined (Damelin and Miller Jr, 2012) for two 1D sequences f, g as,

$$(f*g)[n] = \sum_{i=-\infty}^{\infty} f[i]\, g[n-i]. \tag{2.37}$$

This can be extended to 2D sequences,

$$(f*g)[m,\, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,\, j]\, g[m-i][n-j]. \tag{2.38}$$

Convolution in Practice

While we have described the mathematical convolution operator, when performing convolution on feature maps/images in neural networks, we do not exactly adhere to this definition. In practice the input images or feature maps are multi-dimensional finite arrays, *i.e.* tensors; 3D for RGB colour images inputs (2 spatial + 1 colour dimension). CNN layers perform a 2D convolution[2] with a 3D filter over each channel of the input image, and stack the response images into an output tensor, where the number of output channels is the same as the number of convolutional filters in the layer. Each of the channels of the input and output tensors we will call an *image*.

We will here adopt a similar notation to I. Goodfellow, Y. Bengio, and Courville (2016, chapter 9)[3], and denote the incoming feature map $\mathbf{X}$, outgoing feature map $\mathbf{Y}$, and convolutional filter, or kernel $\mathbf{F}$. The scalar elements of each feature map are $\mathbf{X}_{i,j,k}$, $\mathbf{Y}_{i,j,k}$, where $i = \{0, \ldots, c\}$ is the feature maps channel (*i.e.* colour for an input image), and $j = \{0, \ldots, h\}$, $k = \{0, \ldots, w\}$ are the spatial coordinates, rows and columns respectively, of the channel i image. The filter's scalar elements are $\mathbf{F}_{i,j,k,l}$, where i is the filter's index in the convolutional layer's filter bank and the output channel in $\mathbf{Y}$ to which the filter's result is written, j is the input channel in $\mathbf{X}$ over which the filter's spatial elements are convolved, and (k, l) are the row and column offset between the output and input images.

A convolutional layer then convolves across the layer such that,

$$\mathbf{Y}_{i,j,k} = \sum_{l,m,n} \mathbf{X}_{l,j+m,k+n}\, \mathbf{F}_{i,l,m,n}, \tag{2.39}$$

for all valid indices l, m, n, depending on the padding of the input image. We only use zero padding as detailed in our experiments, please see Szeliski (2011, §3.2) for a more in-depth discussion on alternative forms of padding.

Pooling Layers

Another key aspect of convolutional architectures is pooling, a form of non-linear spatial sub-sampling of the feature maps of a given layer. Pooling layers were designed to add translation invariance to CNNs by making the network less sensitive to small local changes in the spatial location of input pixels/convolutional responses, and also to reduce feature map spatial sizes with network depth. Reducing the feature map

[2] this is typically called a 2D convolution rather than 3D, since there is no 'sliding' of the filter in the channel dimension.

[3] note we use zero-based indices, unlike I. Goodfellow, Y. Bengio, and Courville.

spatial size serves both to save computation, and to increase the size of the receptive field of successive convolutional layers. This allows successive convolutional layers to operate on progressively larger scales.

Pooling layers divide the image into non-overlapping pooling regions, in which the spatial extents of the input image/feature map are aggregated into a single scalar. LeNet used the average to aggregate the pooling regions, *i.e.* average pooling (LeCun, Bottou, *et al.*, 1998). More modern networks have typically used max to aggregate pooling regions, *i.e.* max-pooling, which empirically has been found to work better. Some even more recent networks do not use pooling at all, but simply use a strided convolution to reduce the feature map sizes (He *et al.*, 2016a).

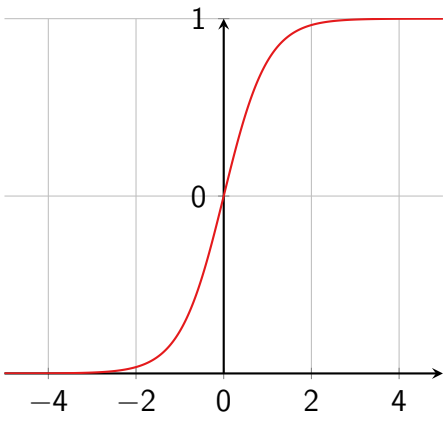
Strided Convolution

Convolutional layers are memory intensive since they must store the output feature maps and, during training, the backpropagated gradients. The largest feature map is typically that of the first convolutional layer, since the input image has relatively large spatial dimensions, and pooling reduces the spatial size of feature maps exponentially with depth. Due to the memory limitations of current GPUs, this means that many contemporary network architectures cannot be trained on even modestly sized input images without using strided convolution. When using strided convolution, a given number of input feature map/image pixels are skipped in both the row and column directions, producing a smaller output feature map and reducing computation, at the sacrifice of a coarser output feature map, and scale. For a stride of s pixels in both the row and column directions, the strided convolution operation can be defined,

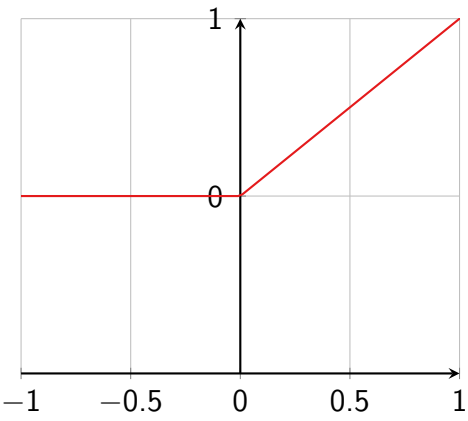
$$\mathbf{Y}_{i,j,k} = \sum_{l,m,n} \mathbf{X}_{l,sj+m,sk+n} \, \mathbf{F}_{i,l,m,n}. \tag{2.40}$$

2.2 Contemporary Methods of Training Neural Networks

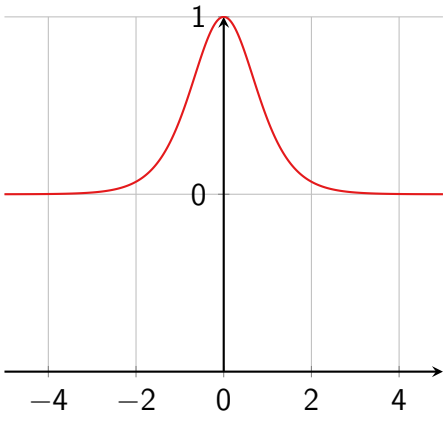
Here we will outline the most relevant differences in training contemporary DNNs as compared to before the work of Krizhevsky, Sutskever, and Geoffrey E. Hinton (2012), less than 10 years ago.



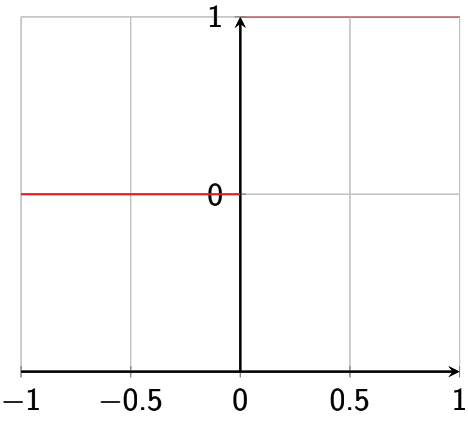
(a) Hyperbolic Tangent $y = \tanh(a)$



(b) ReLU activation function $y = \max(0, a)$



(c) Derivative $\frac{\mathrm{d}}{\mathrm{d}a}\left(\tanh(a)\right)$



(d) Derivative $\frac{\mathrm{d}}{\mathrm{d}a}\left(\max(0, a)\right)$

Fig. 2.10 **Common activation functions used in neural networks.**

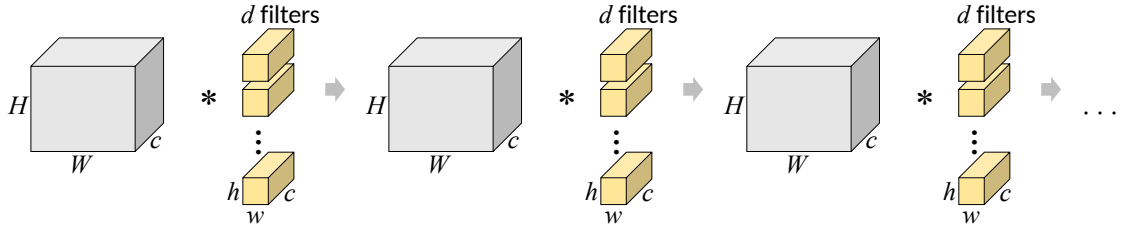


Fig. 2.11 **Vanishing gradients.** For networks with many layers, even small deviations from a unit gradient are quickly geometrically magnified by propagation through all layers.

2.2.1 Rectified Linear Activation Function

An integral part of any useful neuron in a neural network is a non-linear activation function. With a linear activation function, even the deepest network would only be able to represent a linear function. Historically, neural networks have used sigmoidal activation functions, as explained in section 2.1.9.

A major issue with sigmoidal activation functions however, is that gradients outside of a relatively narrow region of the function domain (close to $a = 0$) are very small. When training with backpropagation, this means that most gradients are of very small magnitude, and training can take a very long time, or even stall altogether — a situation that is often called the *vanishing gradient* problem, first identified by Hochreiter (1991). This term has also been conflated with numerical precision issues caused by incorrect initialization, as identified by Glorot and Y. Bengio (2010).

ReLUs were proposed as a solution, first for restricted Boltzmann machines (Nair and Geoffrey E. Hinton, 2010), and later for neural networks (Glorot and Y. Bengio, 2010), where empirically they were shown to allow easier training with backpropagation. These neurons have a piece-wise activation function, plotted in fig. 2.10(b),

$$f(a) = \max(0, a). \tag{2.41}$$

ReLUs do not exhibit the 'saturation' of sigmoidal functions, always giving a gradient of either 0 or 1. In practice this can greatly speed up training with backpropagation, or even allow training networks that are not otherwise trainable in practice with sigmoidal activation functions, such as the deep network of Krizhevsky, Sutskever, and Geoffrey E. Hinton (2012).

2.2.2 Methods of Network Initialization

Until relatively recently, pre-training was considered necessary for the feasibility of training deep neural networks (Geoffrey E. Hinton and Salakhutdinov, 2006). The vanishing gradient problem was first addressed through better methods of random initialization which considered the geometric effect of propagating gradients through very DNNs. Without careful initialization, gradients can either surpass numerical representation (exploding gradients) or be reduced to close to zero (vanishing gradients). This effect is further exacerbated by the use of the softmax function at the end of the network, containing the exponential function.

For example, consider a deep network consisting of L identical layers as shown in fig. 2.11, and assume that our initialization results in the first pass through the network scaling the signal (i.e gradients) by a factor of β for each layer.

After propagating through L layers, this becomes a scaling of β^L, exponentially magnifying the effect of the discrepancy. For example, the output of a trivial deep network, where each layer l only maps the identity function, $f_l(x) = x$, with L layers, and each layer is initialized such that the output response is scaled by β, will be:

$$\begin{aligned} f_L(x) &= (f_1 \circ f_2 \ldots \circ f_L)(x) \\ &= x \prod_{l}^{L} \beta = x\, \beta^L, \end{aligned} \tag{2.42}$$

where $(f \circ g)(x)$ is the composition $f(g(x))$. This problem has two distinct outcomes determined by the effective scaling of each layer's initialization β:

$$\lim_{L \to \infty} f_L(x) = \begin{cases} \infty & \text{if } \beta > 1, \text{ training loss } \textit{diverges} \\ 0 & \text{if } \beta < 1, \text{ training loss } \textit{stalls}. \end{cases}$$

Thus we want a random initialization of layers such that $\beta \approx 1$ to minimize the 'vanishing gradient' effect. For sigmoidal activation functions, such an initialization was proposed by Glorot and Y. Bengio (2010). For random Gaussian initialization, and given the number of outgoing/incoming connections to each neuron, we can carefully choose the standard deviation σ such that the expected value,

$$\mathrm{E}\left[\beta\right] = 1. \tag{2.43}$$

In practice however, most networks have layers of different numbers of neurons. In this case, since there are different numbers of incoming connections and outgoing

connections for each neuron, there are two possible initializations, one for the expected forward pass (response) scaling, and one for the backwards pass (gradient). As a compromise, Glorot and Y. Bengio (2010) proposed to use the average number of outgoing and incoming connections to the neuron:

$$\begin{aligned} \sigma_{\text{forwards}} &= \frac{1}{\sqrt{n_{\text{out}}}} \\ \sigma_{\text{backwards}} &= \frac{1}{\sqrt{n_{\text{in}}}} \\ \sigma_{\text{average}} &= \frac{1}{\sqrt{(n_{\text{out}} + n_{\text{in}})/2}}. \end{aligned} \tag{2.44}$$

For the more typically used rectified linear unit, a variation of this initialization was proposed by He *et al.* (2015):

$$\begin{aligned} \sigma_{\text{forwards}} &= \frac{2}{\sqrt{n_{\text{out}}}} \\ \sigma_{\text{backwards}} &= \frac{2}{\sqrt{n_{\text{in}}}} \\ \sigma_{\text{average}} &= \frac{2}{\sqrt{(n_{\text{out}} + n_{\text{in}})/2}}. \end{aligned} \tag{2.45}$$

2.2.3 Batch Normalization

Some network architectures are sufficiently complex, *i.e.* networks with neurons with drastically different number of outgoing/incoming connections, that even careful initialization will not prevent exploding/vanishing gradients. Instead, Ioffe and Szegedy (2015) proposed a more direct approach of maintaining the desired zero-mean, unit Gaussian response distribution. Batch normalization proposes to use batch statistics to whiten the responses of layers it is applied to during training.

Full whitening (*i.e.* de-correlating the responses) is prohibitively expensive however, instead batch normalization calculates the mini-batch mean and variance, and prevents vanishing gradients by normalizing responses/gradients according to the batch statistics. Using batch-normalization during training can make a dramatic difference, in many cases networks that would previously not converge, will converge, and it can sometimes even speed up training.

The batch statistics calculated by batch normalization are,

$$\begin{aligned} \mu_b &= \frac{1}{M}\sum_{i=0}^{M} x_i \\ \sigma_b^2 &= \frac{1}{M}\sum_{i=0}^{M} \\ \hat{x}_i &= \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}, \end{aligned} \tag{2.46}$$

where for mini-batch b, μ_b is the mean, σ_b^2 is the variance, ϵ is for numerical stability, $\{x_0, \ldots, x_i, \ldots x_M\}$ are the mini-batch responses for a particular parameter of the layer, and $\hat{x}_i$ is the normalized response for that parameter.

Batch normalization then uses these statistics, along with two parameters learned across mini-batches, γ and β, to scale and shift the responses,

$$y_i = \gamma\hat{x}_i + \beta. \tag{2.47}$$

Most state-of-the-art supervised DNNs now use batch normalization along with the initialization, as presented in section 2.2.2, to train.

2.2.4 Dropout

Geoffrey E. Hinton, Srivastava, *et al.* (2012a) and Srivastava *et al.* (2014) introduced *dropout*, a method of preventing overfitting in large networks during training. The implementation of dropout is to, during training, effectively zero out a set of neurons, sampled randomly from each layer with a fixed probability p. At test time all the neurons are active, and to maintain the expected responses, a multiplicative factor of p is used.

The mechanism of the effect of dropout is explained several different ways, and notably different explanations are given by Geoffrey E. Hinton, Srivastava, *et al.* (2012a) and Srivastava *et al.* (2014)[4]. The explanation by Geoffrey E. Hinton, Srivastava, *et al.* (2012a) is that dropout is a form of regularization by noise, preventing 'co-adaption' of neurons (see chapter B). The main explanation given by Srivastava *et al.* (2014) is that dropout is a form of model integration, averaging over a large number of random 'thinner' model architectures at training time in order to improve generalization. At training time however, averaging over all the models considered during training would

[4]we have found some empirical evidence to suggest it is an optimization trick rather than a form of regularization, see section B.3.

be extremely computationally expensive, since there are an exponential (2^n) number of possible models considered during training.

Empirically dropout improves generalization of networks with very large layers, in particular the large fully-connected layers to be found at the end of the AlexNet and VGG architectures. It has less effect on models train with batch normalization however, as observed by the authors (Ioffe and Szegedy, 2015), and in practice is not used in more recent deep network architectures such as ResNet.

2.3 Deep Neural Network Architectures

An exhaustive list of every novel deep learning architecture would be infeasible, and outside the scope of this dissertation, however here we have made an effort to cover the recent architectures which have both inspired our work and formed the basis of many of our results.

2.3.1 AlexNet

Training DNNs, that is neural networks with many (*i.e.* two or more) hidden layers, had proven difficult due to the high computational complexity, and the so called 'vanishing gradient' problem (Y. Bengio, Simard, and Frasconi, 1994). Krizhevsky, Sutskever, and Geoffrey E. Hinton (2012) showed that a deep CNN (the specific architecture since referred to as AlexNet) trained on a very large dataset (Russakovsky *et al.*, 2015), with the appropriate initialization (Sutskever *et al.*, 2013), weight decay ReLU activation functions (Nair and Geoffrey E. Hinton, 2010) and dropout (Geoffrey E. Hinton, Srivastava, *et al.*, 2012b) could beat state-of-the-art methods on large scale object class recognition methods, based on hand-crafted features, by a large margin. This single paper introduced or motivated many of the recent advances in training neural networks, as covered in section 2.2.

AlexNet notably used training-time and test-time augmentation to achieve its state-of-the-art accuracy. During training random 224×224 crops of a 256×256 image are used, along with random mirroring of these crops. In addition *relighting augmentation* is used, where the PCA components over all RGB pixels in the image are used to perturb the "brightness" of the image, and give some robustness to photometric variations in the test images. At test time "10× oversampling" is used, that is for each 256×256 test image, and its mirrored image, four corner and one centre crop are pushed through the network, and the prediction is simply the averaged over these 10

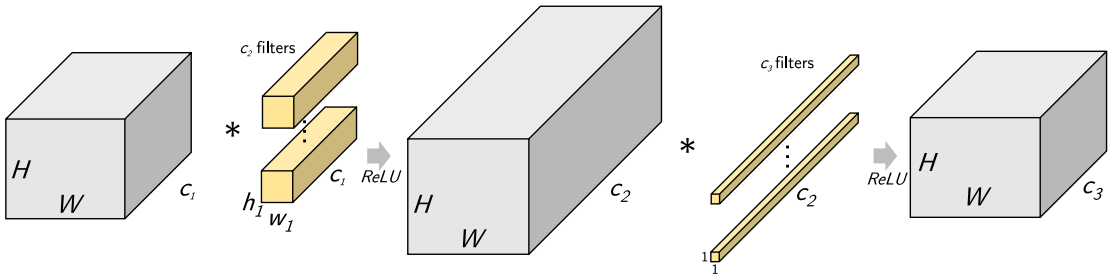


Fig. 2.12 **LDE.** Introduced in the NiN architecture, a LDE consists of learning a 1×1 convolutional layer after a normal convolutional layer. The pairing of 1×1 filters and a non-linearity (*i.e.* ReLU) can effectively learn a non-linear transformation into a different space. If $c_3 < c_2$, then a transformation into a lower-dimensional space is learned, and potentially a more compact embedding of the learned representation.

crops. Finally, for the best results reported (Top-5 error of 15.4%), an *ensemble* of 7 models is used, where the prediction is the average of all of these models.

AlexNet uses two filter groups throughout most of the layers of the model in order to split computation and model parameters across two GPUs, the motivation being that at the time GPUs did not have enough memory to fit such a large model. The authors observed that the filters on each GPU appeared to specialize to learn fundamentally different features regardless of initialization (Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012). This interesting observation has mostly been ignored in subsequent networks where GPU memory has increased enough that such a split of the network is not required, but the original observation is a fundamental motivation of our work.

2.3.2 Network in Network

Lin, Q. Chen, and Yan (2014) introduced NiN (Network in Network), in which the main contribution was the use of so-called 'micro networks', consisting of increased non-linearity between convolutions using 1×1 convolutions. The authors claimed the extra non-linearities introduced allow the network to capture more complex functions. These 1×1 convolutions, illustrated in fig. 2.12, have since been referred to as LDE (Low-Dimensional Embeddings). If the number of 1×1 filters is lower than the number of normal convolutional filters, then the 1×1 layer learns a non-linear transformation of the input feature map into a smaller space, *i.e.* a reduction in the number of filters by a mapping of a high-dimensional feature map onto a lower-dimensional feature map. This can be used to reduce the computation and parameters of convolutional layers significantly, while potentially learning a more compact and efficient representation. The full NiN architecture is shown in fig. 2.13.

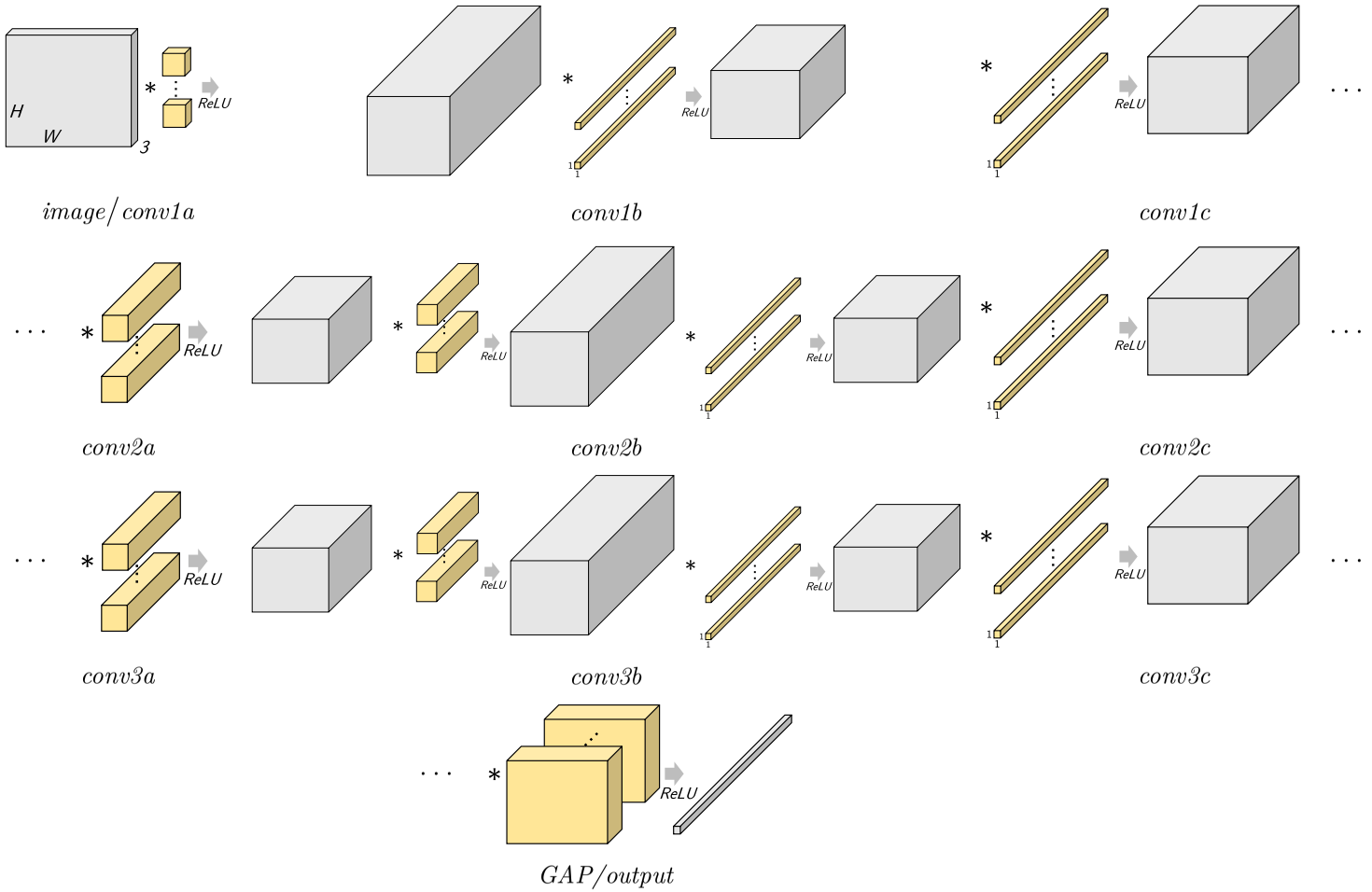


Fig. 2.13 **NiN Architecture.** Coloured blocks represent the filters of each layer, grey blocks the intermediate feature maps over which a layer's filters operate. GAP is used at the end of the network.

Lin, Q. Chen, and Yan (2014) also introduced *GAP*, in which the spatial extents at the end of the convolutional layers (*i.e.* pool5 for NiN/AlexNet) are aggregated such that there is only a single scalar output response for each filter in the pooled layer. After GAP of a layer with f filters/feature maps, the input to the classification layer is simply a vector of f responses, as illustrated in fig. 2.13. This reduces the parameters of the network dramatically since the majority of the parameters in the network are typically between the last convolutional layer and the fully-connected classification layer. Lin *et al.* showed that on CIFAR-10 GAP by itself achieved a lower error than having a fully-connected layer with dropout.

2.3.3 VGG

Since AlexNet, there have been many improvements to the state of the art on the ILSVRC challenge, every one of which has been an improved CNN architecture. One particular architecture that has lent itself to both high accuracy and being a natural extension of the original network has been that proposed by Simonyan and Zisserman (2015) of the Visual Geometry Group (VGG) at Oxford. The primary contributions of the VGG network are (**i**) showing that very deep networks improve generalization, and (**ii**) learning stacked small filters, *i.e.* three 3×3 convolutional layers is more computationally efficient than learning a single convolutional layer of 7×7 filters, and also improves generalization.

VGG is an evolution of the AlexNet models, with the same number of max-pooling layers, however using very small convolutional filters (3×3) in the convolutional layers, and many more of these convolutional layers between pooling, instead of the relatively large single-layers of convolutional filters in AlexNet (7×7). In addition VGG uses small non-overlapping max-pooling (2×2), and the *fully convolutional trick* introduced by Sermanet *et al.* (2014) to do test-time oversampling more efficiently. VGG uses extensive training augmentation, extending the augmentation used in AlexNet (Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012) by adding scale augmentation, where crops are taken from images of different rescaled sizes.

2.3.4 Inception

The winner of the ILSVRC2014 challenge, as measured by classification accuracy, was the *Inception architecture*, or GoogLeNet (Szegedy, Liu, *et al.*, 2015). The Inception architecture is particularly interesting, in that it was created explicitly to minimize computation and learn a more efficient representation. Although it uses the LDE of

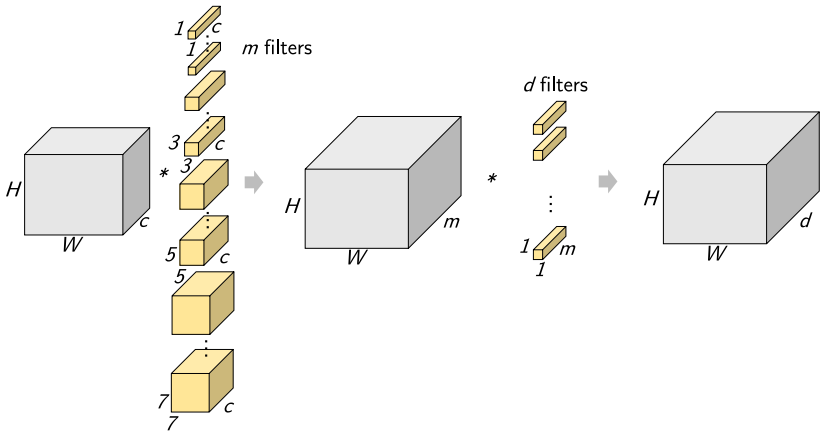


Fig. 2.14 **Inception unit.** The building block of the Inception/GoogLeNet architecture. Learns a limited number of large filters (7×7, 5×5), and a great number of small filters (3×3, 1×1), followed by a LDE (1×1) layer.

NiN, this is combined with a novel combination of filters of different spatial size, within what is called an *Inception unit*, illustrated in fig. 2.14.

The motivation of the architecture is that most of the important correlations in natural images are very localized, so much so that even 3×3 filters can learn most of the important features, for example image gradients and edges — as demonstrated by the VGG networks (Simonyan and Zisserman, 2015). However, a few of the correlations are less localized, more complex, and better captured by 5×5 or even 7×7 filters. Instead of learning a lot of large and computationally expensive 7×7 filters, the Inception unit learns mostly 1×1, and 3×3 filters, with fewer 5×5 and even fewer 7×7 filters. This represents a balance between representation and efficiency.

The authors explain this as learning 'factorized' filters, however we disagree, and understand this architecture instead as learning a *basis* for filters. Ignoring the non-linearity between the two layers in fig. 2.14, heterogeneous filters on the same layer are concatenated into a single feature-map, which is then *linearly combined* by the 1×1 filters of the subsequent layer. This linear combination of smaller filters in order to represent a minimally parameterized, but effective filter of full-size (7×7), is similar to representing a complex function as a parameterization of simpler basis functions.

2.3.5 Residual Networks

He *et al.* (2016a) introduced residual networks, which provide an important insight on a problem with the training of very deep networks. While deeper networks have been found to improve generalization, especially with large datasets; at a sufficiently

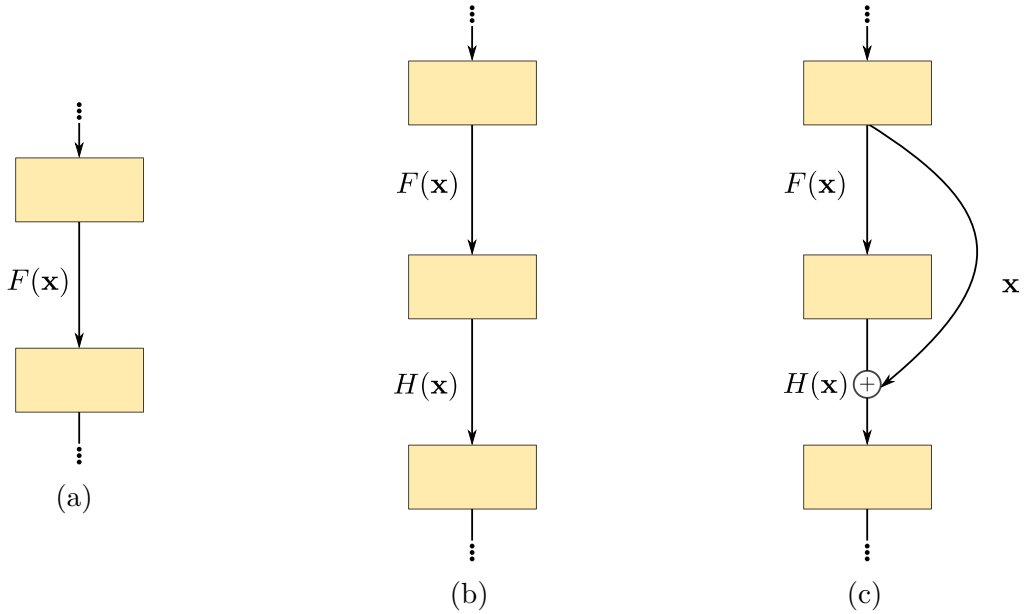


Fig. 2.15 **Residual networks.** (a) A convolutional network, where the mapping between the final two layers is $F(\mathbf{x})$, (b) learning an additional layer with the mapping $H(\mathbf{x})$, and (c) learning an additional residual layer with the mapping $H(\mathbf{x}) + \mathbf{x}$.

large depth training becomes difficult, even with batch normalization and the correct initialization, and generalization begins to level off, or even decline.

The important insight of He *et al.* (2016a) into this problem can be summarized in fig. 2.15. Having trained a deep network with good generalization (*i.e.* fig. 2.15(a)), with N layers, a training loss of $\mathcal{L}_1$ is observed. On adding a single-layer to the otherwise identical deep network architecture (*i.e.* fig. 2.15(b)), and re-training from random initialization, the new training loss of the network with $N + 1$ layers is found to be $\mathcal{L}_2 > \mathcal{L}_1$, *i.e.* the training loss has increased.

Yet from an optimization standpoint it is not clear why this should be so. We can observe that there is a trivial set of parameters defining a transformation that will maintain the training loss of the shallower network, *i.e.* $\mathcal{L}_1 = \mathcal{L}_1$ — that is the identity transformation $H(\mathbf{x}) = x$.

He *et al.* (2016a) proposed that in order to aid the optimization, a *residual connection* (as in fig. 2.15(c)) is added to the convolutional layers, allowing the trivial identity solution to be easily learned. This residual connection can be thought of as a shortcut, bypassing the previous layer. Assuming our desired, but difficult to optimize, mapping from one layer to the next is $H(x)$, the residual function learned is simply:

$$H(x) + x. \tag{2.48}$$

In practice these residual layers greatly help the training of very deep networks, and have pushed state-of-the-art accuracy in many datasets. All current state-of-the-art models for image classification use residual layers.

3

The Effect of Structure on Learning

"Everything, but the data itself, is an assumption."
– Zoubin Ghahramani, *Microsoft Research AI Summer School, 2017*

It is well known that the design of a neural network architecture can have a large effect on the generalization of a learned model; and yet network design itself remains poorly understood, with intuition and experience being the cited motivation behind most common architectures, rather than theory. This, more than perhaps any other factor, has been a barrier to access for the practical use of neural networks by people who are not experts in the field.

Beyond hyper-parameters used for tuning the optimization method, such as learning rate, momentum and weight decay, the architecture of a network has a profound effect on the learning. Nowhere is this effect more pronounced than in the case of using neural networks with highly structured inputs, such as natural images. Although neural networks are usually posed as general learning machines, time and again it has been demonstrated that neural networks only truly stand out as a learning method when we encode our prior knowledge of the task in the architecture itself — a concept that we will, throughout this work, refer to as *structural priors*. Examples of structural priors include common network architectures for images (CNNs), and sequences (RNNs).

Neural Networks with structural priors still differ significantly from hand-tuned local features, as popularized in computer vision in the early 2000s, such as SIFT (Lowe, 2004). As compared with neural networks, such local features are rigidly defined in terms of structure and weights, and the learning system is restricted to finding and cataloguing the pre-determined features in images. Neural networks with structural priors on the other hand, while restricting the structure of the network somewhat,

still allow the network to learn more fine-grained structure, and have no effect on the latitude given to learning weights.

The history of understanding the role of neural network architecture in learning is long, arguably going back to the Hebbian rule of learning (Hebb, 1949), and yet our understanding is still far from complete. In this section we will review a select number of the most important previous works relevant to understanding the role that structural priors play, and how they emerged to dominate the practical use of neural networks today.

3.1 Network Architecture

A persistent question in training artificial neural networks has been in the design of the networks. Specifically the question of how many parameters should be learned, and in what way they should be connected, so as to be suitable for good generalization from a given size dataset. Notable steps in the theoretical answers to this question include findings showing the limitations of single-layer networks (Minsky and Papert, 1988), information-theoretic measures of the representational capacity of a network (Vapnik and Chervonenkis, 2015), the proof that single hidden-layer networks are universal approximators (Hornik, Stinchcombe, and White, 1989), and the theoretical number of nodes required for generalization from a dataset of given size (Baum and Haussler, 1988).

Empirical results have, however, shown that generalization of neural networks is not fully explained by our current theory. Deep networks of many hidden layers have been shown time and again to out-perform shallow networks (He *et al.*, 2016a,b; Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012; Simonyan and Zisserman, 2015), perhaps due to our limited method of optimization (Ba and Caruana, 2014). Networks with many more parameters than training samples[1], that use early-stopping or are regularized strongly, generalize better in practice than networks with the theoretically sufficient capacity (Caruana, Lawrence, and Giles, 2000; Geoffery E. Hinton, 2015; Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012). Networks designed with a specialized connectivity structure closer reflecting the underlying representation being learned have consistently generalized better than fully-connected networks with higher learning capacity (He *et al.*, 2016b; LeCun, Boser, *et al.*, 1989). In fact these design strategies, so poorly explained by theory, can claim to have been responsible for

[1] although it should be noted, these training samples often have high dimensionality, *i.e.* 256×256 images in Imagenet.

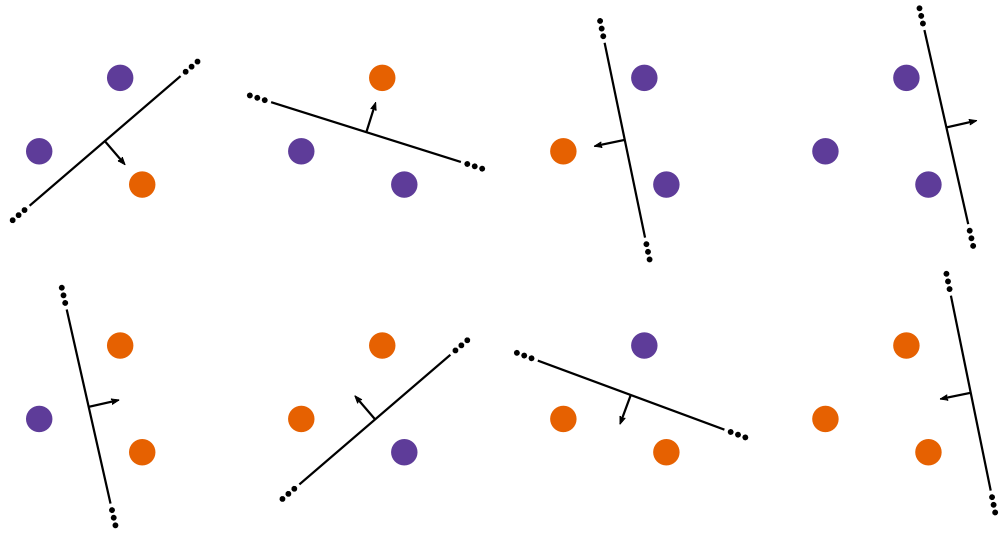


Fig. 3.1 **Possible labellings of 3 points in $\mathbb{R}^2$.** All possible labellings of 3 points in $\mathbb{R}^2$ can be separated by a oriented line (2D hyperplane). This is not possible for all labellings of 4 points in $\mathbb{R}^2$ however, and thus the VC dimension of oriented hyperplanes in $\mathbb{R}^2$ is 3. Inspired by figure in Burges (1998).

recent breakthroughs in generalization on previously difficult tasks such as image class recognition (Geoffery E. Hinton, 2015; Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012).

3.2 Model Capacity and Representational Power

The information-theoretic notion of capacity, that is the expressive power of a classification algorithm, gives important insights to the learning ability of a classification algorithm. Analysis is typically based on the Vapnik–Chervonenkis (VC) dimension (Vapnik and Chervonenkis, 2015) of the class of functions used as discriminators, *e.g.* hyperplanes in the case of neural networks. Intuitively, for a discriminative classifier, the VC dimension measures the largest number of points that can be classified without error. In such a case, the set of points is said to be *shattered* by the classifier. A good overview of VC dimension is given by Burges (1998).

3.2.1 Vapnik-Chervonenkis Dimension

More formally, a classification model $f(\theta)$, parametrized by θ is said to *shatter* a set of data points $(x_0, x_1, \ldots, x_h)$ if for all possible labellings of the points, the classification model can perfectly learn the points. The VC dimension is the largest number of (any)

points that can be shattered by such a classifier. For a classifier of VC dimension h, it is sufficient that there exists a *single* set of h points which can be shattered. It is important to note that in general a classifier with a VC dimension of h will not necessarily shatter every possible set of h points.

For example, in fig. 3.1, the function class of oriented hyperplanes, *i.e.* lines in 2D, can separate all possible labellings of 3 points in $\mathbb{R}^2$ — oriented hyperplanes in $\mathbb{R}^2$ shatter 3 points. However, for 4 points, this is no longer true. It can be proven (Burges, 1998) that in general for $\mathbb{R}^n$, the set of oriented hyperplanes shatters any set of $n + 1$ points.

The VC dimension gives us a measure of the theoretical learning capacity of a classifier, however it can also be somewhat counter-intuitive. While models with large numbers of parameters usually will have a higher VC dimension, there are examples of small single parameter models with infinite VC dimension for more specific sets of points. For example, if we have a set of evenly spaced points in 2D, a simple sinusoidal curve with the appropriate phase can shatter any labelling of an infinite number of such points. However, such a classifier would be poor at classifying more general sets of points, despite the impressive theoretical VC dimension.

3.2.2 VC Dimension of Neural Networks

In the case of neural networks, early work showed the capacity of neural networks to be quite large (Baum and Haussler, 1988; Hornik, Stinchcombe, and White, 1989). Baum and Haussler (1988) looked at feed-forward networks of *threshold units*, *i.e.* perceptrons.

The authors prove a lower bound on the VC dimension for a single hidden layer network of k units and n inputs,

$$d_{\mathrm{VC}} \geqslant 2\lfloor k/2 \rfloor n, \tag{3.1}$$

where $\lfloor \ \rfloor$ is the floor operation, *i.e.* largest integer less than or equal to the operand, and d is the number of inputs. For a single hidden layer network with a large number of n inputs and k units, the authors make the assumption that $kn \approx w$, *i.e.* the number of weights in the first layer alone is approximately that of the whole network w,

$$d_{\mathrm{VC}} \geqslant w. \tag{3.2}$$

Baum and Haussler (1988) use this lower bound on the VC dimension of a hidden layer to bound the number of training samples required to achieve an error rate of ϵ, showing that for a network with w weights, and a desired error rate ϵ the minimum number of training samples required is given by,

$$N_{\min} \approx w/\epsilon. \tag{3.3}$$

For an error rate of $\epsilon = 0.1$, this gives the rule of thumb that for a network with a total of w weights, approximately $10 \times w$, or 10 times the number of training samples as weights in the network, are required to guarantee an error rate of 10%.

At first glance this work may seem to have solved a major problem in the design and training of neural networks, however the work of Baum and Haussler (1988) is to show a worst-case lower bound on the number of training samples required — in practice this can be far from what is empirically required. Baum and Haussler (1988) themselves point out that this is likely be far more than necessary in networks where the learning algorithm seeks to minimize the number of non-zero weights (such as networks using weight decay, pruning, etc). Indeed, in practice neural networks were found to generalize much better than this worst case bound would indicate, to the point where modern deep neural networks are trained with far fewer samples than weights, in the case of AlexNet (Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012) approximately 250× *fewer* training samples than weights allows good generalization on the ILSVRC (Russakovsky *et al.*, 2015) dataset. It should be noted however, that each of these samples have a high dimensionality, each image has around 65,000 pixels.

Bartlett (1996) later showed that rather than capacity being based solely on the number of weights, a bound more in-line with empirical results could be found by using the number of *large weights*. To show this, they moved to a scale sensitive form of the VC dimension: the *fat-shattering* dimension. The author shows that the error rate for an ℓ-layer sigmoidal (rather than threshold unit) neural network with n inputs and m training samples,

$$\epsilon \approx (cA)^{\ell(\ell+1)/2}\sqrt{(\log n)/m}, \tag{3.4}$$

and c is a constant factor, and the ℓ_1 norm of each unit's weight vector w is bounded by A,

$$\|\mathbf{w}\|_1 = \sum_i |w_i| \leqslant A. \tag{3.5}$$

Surprisingly there is no term for the number of units for any layer in this equation, but rather it is the bounds on the weights themselves that determine ability of the network

to generalize. In general, for a network where the inputs $\mathbf{x}$ are also bounded,

$$\|\mathbf{x}\|_\infty = \max(|x_0|, |x_1|, \ldots, |x_n|) \leqslant B, \tag{3.6}$$

Bartlett (1996) show that for a given error ϵ, the number of training samples m required grows roughly as,

$$m \approx \frac{B^2 A^{\ell^2}}{\epsilon^2}. \tag{3.7}$$

For a single hidden layer network, *i.e.* $\ell = 2$, and an error rate of $\epsilon = 0.05$,

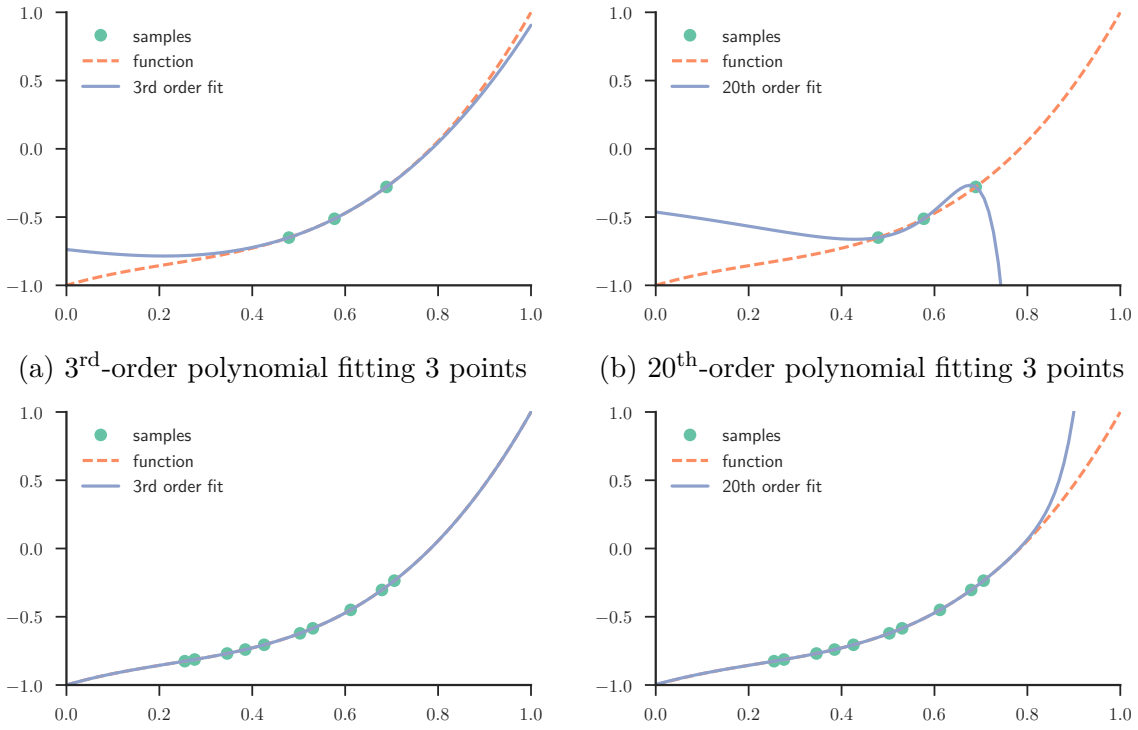
$$m \approx 400\, B^2 A^6. \tag{3.8}$$

This result, while surprising given the analysis based on the VC dimension of neural networks, supports empirical results in using contemporary training methods such as weight decay (Geoffery E. Hinton, 1987), early stopping (Bishop, 1995), and even more recently batch normalization (Ioffe and Szegedy, 2015), all of which can keep weight magnitudes low.

3.2.3 Model Size

J. Denker *et al.* (1987) explored the relationship of network architecture to generalization in a more empirical manner. The work was particularly motivating in the later design of CNNs (LeCun, 1989; LeCun, Boser, *et al.*, 1989). The authors make the intuitive analogy between the effect of the size of a neural network on its generalization, and a simple least-squares polynomial fit. Figure 3.2 shows various polynomial fits to samples from a 3$^{\text{rd}}$-order polynomial function. When using a 3$^{\text{rd}}$-order polynomial to fit even a small number of samples (fig. 3.2(a)), the fit extrapolates, *i.e.* is closer to the desired function outside the range of training samples, better than when we use a 20$^{\text{th}}$-order polynomial to fit the same data (fig. 3.2(b)). While the number of samples can help the fit of the higher-order function, even with a large number of samples the 20$^{\text{th}}$-order polynomial fit (fig. 3.2(d)) will not extrapolate as well as the polynomial with a more appropriate lower number of parameters (fig. 3.2(c)). Similarly, a neural network with a large number of parameters may not generalize as well as a neural network with fewer, more salient, parameters.

More recently Caruana, Lawrence, and Giles (2000) further explored the analogy by training neural networks to fit polynomials, showing that overfitting in neural



(a) 3^{rd}-order polynomial fitting 3 points

(b) 20^{th}-order polynomial fitting 3 points

(c) 3^{rd}-order polynomial fitting 10 points

(d) 20^{th}-order polynomial fitting 10 points

Fig. 3.2 **Polynomial fits of samples from a 3rd order function.** Polynomials of high order, like neural networks of many parameters, easily overfit a small number of samples as compared to polynomials of a more suitable order for the sampled function. While generalization is helped by more data, the higher-order polynomial still tends to overfit.

00**1111**00	00**11**0**11**00
00**111**0000	00**1**0**1**0**1**00
0000**1**0000	00**111**0**1**00

(a) Binary sequences with one clump

(b) Binary sequences with two-or-more clumps

Fig. 3.3 **Two-or-more clumps predicate.** The two-or-more clumps predicate asks for the network to classify (padded) binary input sequences as having one or two-or-more contiguous strings of ones.

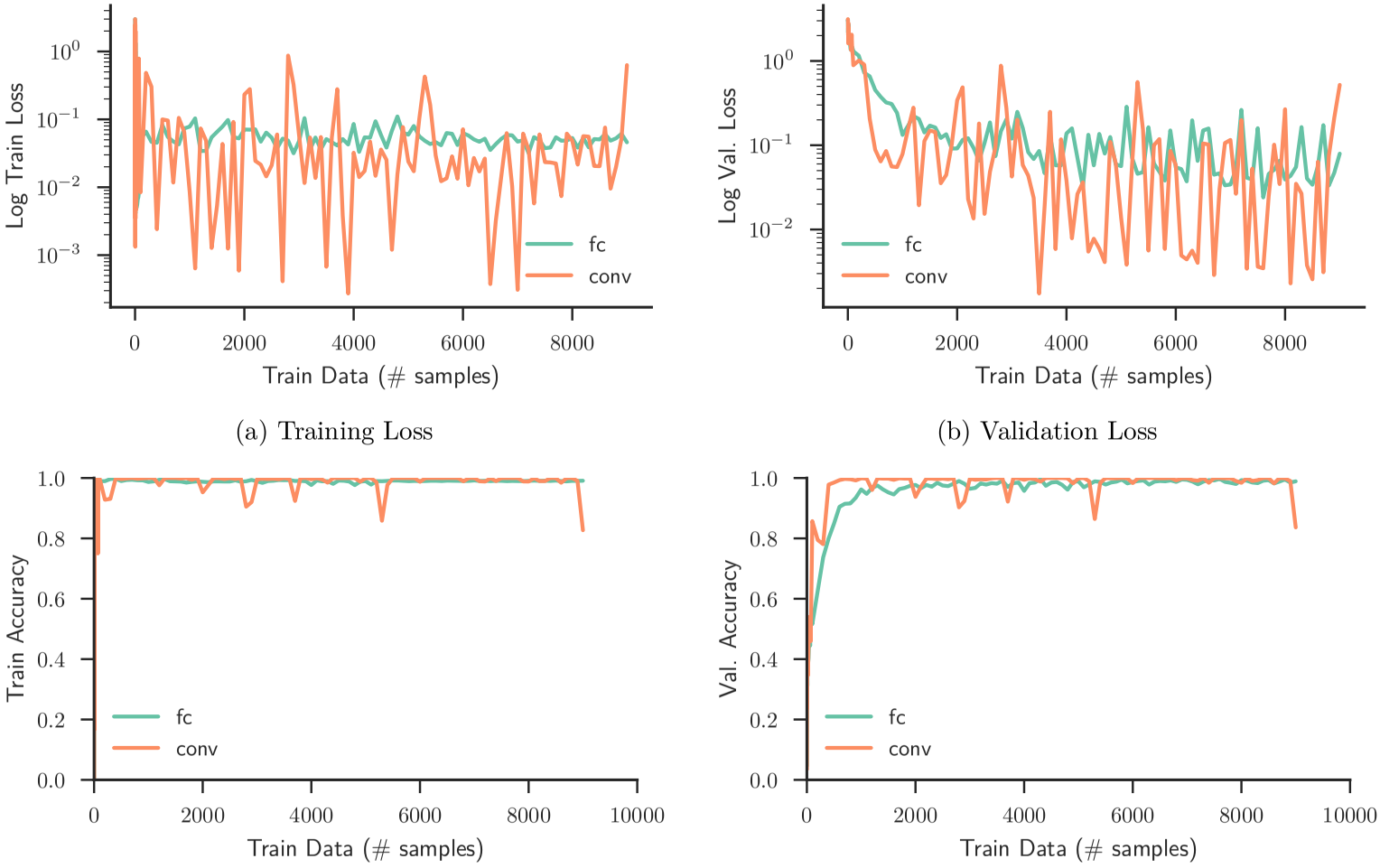
networks does not seem to be as serious a problem as in polynomials. The greatly over-parametrized neural networks still found relatively good fits. The authors suggest that neural networks trained with backpropagation may be biased towards "smoother approximations".

3.2.4 Generalization and Parameters in Neural Networks

There has been a lot of research into the relationship between the number of parameters of a network and its generalization, aside from theoretical work presented in section 3.2.2, there were many empirical studies, especially in the earlier years of connectionist research (Ahmad and Tesauro, 1988; J. Denker *et al.*, 1987; Giles and Maxwell, 1987; Hanson and Pratt, 1988; Geoffery E. Hinton, 1987; LeCun, 1989).

J. Denker *et al.* (1987) and Giles and Maxwell (1987) explore the relationship between network architecture and generalization by evaluating networks for solving the *two-or-more clumps* predicate (truth statement). The two-or-more clumps predicate asks for the network to classify binary input sequences as having one or two-or-more contiguous strings of ones, some examples of which are shown in fig. 3.3. The predicate is largely based on the more general predicate of *connectedness* explored by Minsky and Papert (1988), and is shown to be a problem not linearly separable, or more specifically, solvable by a locally connected perceptron.

The authors illustrate some surprising properties of the generalization of fully-connected neural networks learned with backpropagation. First, a human-preferred 'geometric' solution is manually hard-coded into the weights of a fully-connected network. While this weight configuration is a valid solution, and is intuitive to humans, the authors show that it is not a solution that the network would ever settle upon when trained with backpropagation. By using the geometric solution as an initialization,



(a) Training Loss

(b) Validation Loss

(c) Training Accuracy

(d) Validation Accuracy

Fig. 3.4 **Two-or-more clumps problem and structural priors.** Our results for of the learning curves for the experiment of J. Denker *et al.* (1987). A CNN converges quicker than a much larger fully-connected network sized such that it can fully represent the sparser CNN, *i.e.* the CNN would be fully recovered if the trained fully-connected network learned zero weights for weights not present in the CNN, and learned duplicate weights for shared parameters in the CNN.

and training the network further, the authors show that the error-surface around the region is not stable.

In fig. 3.4 we show the learning curves for this problem, using contemporary training methods. The small CNN converges quicker than a much larger, fully-connected network that contains the super-set of weights not present in the sparser CNN due to weight sharing and missing weights. Something not observed in the original paper however is that the class imbalance due to the numbers of samples for the two classes in a fixed-length binary string seems to cause instability in the convergence of both networks.

Work on trying to understand generalization of neural networks is not the domain of the past, and is still an active area of research. One of the more interesting recent papers on generalization in DNN highlights that DNNs may not always be learning the type of representations we assume. C. Zhang *et al.* (2017) looked at the effect of training DNNs where they replaced the labels of the ILSVRC dataset with random labels. Surprisingly, DNNs can learn the randomly labelled datasets perfectly (*i.e.* zero training error), despite the intrinsic relationship between the labels and images being destroyed. This flies in the face of the commonly accepted explanation that the network is learning to represent the natural data in a low-dimensional manifold in a higher dimension, random labels are unlikely to be learned this way. Instead it seems that DNNs are doing a lot more memorization of training data than previously thought. The authors also show that many forms of regularization currently used, including weight decay and dropout, are less effective than expected.

On the subject of the importance of structure for learning DNNs, these results show that structural priors may be more important than previously thought for generalization than even strong forms of regularization.

3.2.5 No Free Lunch Theorem

The *No Free Lunch* theorem (NFL) (Wolpert, 1996) states that no algorithm performs better than any other when averaged over all possible problems (*i.e.* all possible data-generating distributions) of a particular type, as each algorithm makes assumptions which will bias it towards different types of data distributions. Intuitively, if you are tasked with predicting the unobserved members of a large set, only from a few samples, and without using any assumptions whatsoever, you cannot do better than random on average. Any assumptions you do make on the pattern or relationship between members of the set will give you better performance for some data types, and worse for others. The NFL theorem simply formalizes the notion that we cannot infer a rule

describing an entire set without either information or assumptions on every member of the set.

Formal Definition

Here we present a formal definition of the theorem specialized for supervised learning, as described by Lattimore and Hutter (2013): Let X and Y be the input and label sets representing the input and outputs space of the classification problem, respectively. The classification itself is defined as a mapping $f : X \to Y$, where $y = f(x)$ is the desired class label of input x. Let the training set X_m be the subset of the input space X, on which our classification algorithm is trained, and the unseen input space, $X_u = X - X_m$. Let the classification algorithm be defined $A(f_{X_m}, x)$, where for $x \in X_u$, $A(f_{X_m}, x)$ is the guess for the class label of input x, and $f_{X_m}(x) = f(x)$ where $x \in X_m$. Let the loss function be defined,

$$L_A(f, X_m) = \frac{1}{X_u} \sum_{x \in X_u} 1_{A(f_{X_m}, x) \neq f(x)}, \tag{3.9}$$

where 1_t is the indicator function, returning 1 when t is true, and 0 otherwise. The loss function measures the number of misclassifications on the training data of the classification algorithm. The expected loss on all functions $\mathcal{M} : X \to Y$ is then,

$$E\left[L_A\left(P, X_m\right)\right] = \sum_{f \in \mathcal{M}} P(f) L_A(f, X_m) \tag{3.10}$$

where P is a probability distribution on $\mathcal{M}$.

Theorem 1 (No Free Lunch)**.** *Let P be a uniform probability distribution on $\mathcal{M}$. For any algorithm A and training data $X_m \subseteq X$,*

$$E\left[L_A\left(P, X_m\right)\right] = |Y - 1|/|Y|, \tag{3.11}$$

i.e. no algorithm can do better than random on all possible problems, given only the training data with no other biases.

The NFL theorem leads many to claim that there is no such thing as a universal learner, however it is important to note that in practice the data distributions that we are interested in represent a small number of all possible data distributions, so it's not inconceivable that a learning algorithm with assumptions may be able to do well on many different real world datasets. Formalizing this argument, Lattimore and Hutter (2013) claim that without contradicting NFL, biased learning algorithms may exist

that do well on universal problems. Their main insight is that the assumption of a uniform probability function P on data distributions may be unrealistic, and under other distributions, a biased learner may do well even averaged over all distributions.

No Free Lunch and Structural Priors

The NFL theorem highlights the need to focus on designing learning algorithms that work well for the real-world data distributions of interest. At first this may be seem to be the *raison d'être* of structural priors, however this discounts the generality of the theorem. All neural networks make implicit assumptions even without structural priors. For example, using a single hidden layer neural network assumes that the problem being learned is not linear. If the problem is linear, instead a single-layer network would suffice, and likely generalize better. The NFL is, however, important in pointing out that structural priors in neural networks are not a 'hack'. In fact well-informed assumptions are the reason machine learning works in practice.

3.3 Bayesian Model Selection

Model selection is the problem of choosing, with generalization in mind, between several model architectures with different numbers of layers, neurons, etc. MacKay (1991, 1992, 1995) proposed a Bayesian approach to model selection. MacKay points to the relationship of Occam's razor to Bayesian approaches and, rather than performing model selection by using cross-validation, MacKay proposes to compute the likelihood of models given only the training data. Assume we have several trained models $\mathcal{M}_i$, and a training dataset X. In the problem of model selection we are interested in solving:

$$\underset{i}{\operatorname{argmax}}\, p(\mathcal{M}_i|X), \tag{3.12}$$

where $p(\mathcal{M}_i|X)$ is the probability of the model conditioned on the training data.

The basic principle behind Bayesian model selection is explained by Bayes' rule. For a neural network model $\mathcal{M}_i$ with parameters θ,

$$p(\theta|\mathcal{M}_i, X) = \frac{p(X|\theta, \mathcal{M}_i)\, p(\theta|\mathcal{M}_i)}{p(X|\mathcal{M}_i)}, \tag{3.13}$$

where $p(\theta|\mathcal{M}_i, X)$, or the *posterior*, is what we commonly attempt to solve for using gradient descent when training a neural network. The *prior*, $p(\theta|\mathcal{M}_i)$, are the probabilities on what values we expect the weights of the model to be. The normalization

term, $p(X|\mathcal{M}_i)$, is the probability of the data given the model, or 'evidence', for the model.

In the case of model selection, we are interested in,

$$p(\mathcal{M}_i|X) = \frac{p(X|\mathcal{M}_i)\, p(\mathcal{M}_i)}{p(X)}, \tag{3.14}$$

where $p(\mathcal{M}_i)$ is the prior (subjective) probability we assign to model $\mathcal{M}_i$. If all models are considered to be of equal probability, then inference is based wholly on the evidence.

The evidence itself is given by the marginalization,

$$p(X|\mathcal{M}_i) = \int p(X|\theta, \mathcal{M}_i)\, p(\theta|\mathcal{M}_i)\, d\theta. \tag{3.15}$$

MacKay (1992) proposes to use the Hessian to evaluate the curvature of the error surface around the point on the error surface represented by the parameters learned in the model, in order to gain knowledge of the uncertainty of the parameters, *i.e.* $p\left(\theta|X, \mathcal{M}_i\right)$, however as explained in section 2.1.6, the computation or storage of the Hessian for contemporary deep networks is infeasible.

3.3.1 Occam's Razor

Of particular interest to the question of the effect of structure and regularization on generalization, are MacKay's empirical results showing that Occam's razor is very much a principle on which neural network design should be structured. MacKay (1991, §3.4), shows that regularization is not enough to make an over-parameterized network generalize as well as a network with a more appropriate parameterization.

In fact, MacKay makes the argument that the Bayesian approach naturally encodes the Occam's razor approach, since given two models that predict the training data, the simpler model with fewer parameters will have less flexibility. This means that the model will explain only a more narrow range of data points (and probably only those), as compared to a more complex model which, with more parameters, will be able to explain a wider range of data. Given that both of these models are assigning probabilities, the simpler model will necessarily assign higher probability to the narrow range in which the data of interest lies[2].

[2]Figure 1 in MacKay (1995) explains this concept particularly well

3.3.2 Practical Implementation

The practical implementation of this method is rather more difficult (Chipman, George, and McCulloch, 2001) than might be expected. In particular, like all Bayesian approaches, having the correct prior probabilities for both the models and weights is important, and yet this is practically difficult in large neural networks.

Besides the difficulty of assigning correct priors, MacKay (1992) proposes to use the Hessian to evaluate the curvature of the error surface around the point on the error surface represented by the parameters learned in the model, in order to gain knowledge of the uncertainty of the parameters, *i.e.* $p(\theta|X, \mathcal{M}_i$. As explained in section 2.1.6, the Hessian is difficult to represent for even reasonably-sized contemporary DNNs.

3.4 Constructive Neural Network Algorithms

Given the effect of structure and network design on generalization, what if instead of designing a neural network before training, we could build networks from the ground up based on data? This appealing direction of research was prominent in the late 80's and early 90's.

Constructive approaches to binary classification neural network architectures were the approach taken by Fahlman and Lebiere (1989), Frean (1990), and Mezard and Nadal (1989) among others. A survey of such methods is presented by Parekh, Yang, and Honavar (2000). These methods share one issue however, they present algorithms to build neural networks that can classify a binary training set *perfectly*, as pointed out by Bishop (1995). Rather, our interest is usually in a classifier that generalizes well — neural networks that classify the training set perfectly are likely instead to have overfit the training set.

Tiling Algorithm

The earliest proposals for building neural network structures from the data itself were proposed for *boolean classification* networks, *i.e.* classifying binary patterns and returning a binary result. The *tiling algorithm* (Mezard and Nadal, 1989) is guaranteed to build a network that can achieve perfect classification of the training set. For each layer in the network, a master neuron is first used to provide the best linear separation of the data (trained using the *pocket algorithm* (Gallant, 1986)), and then ancillary neurons are added until perfect classification of the training data is achieved.

Cascade Correlation

Cascade correlation (Fahlman and Lebiere, 1989) is a constructive algorithm for building general neural networks from only a basic two-layer network of input and output nodes. It is based on two key ideas, a *cascade* where hidden units are added to the network one at a time, and *correlation*[3] where the output of the new hidden unit is learned such that it is highly correlated with the error residual being minimized in the existing network.

At the beginning of training, only the input and output layers exist, and since this is a single-layer network with no hidden units, any of the simpler perceptron training rules can be used. After sufficient training, which is heuristically determined, the network is expanded by a single new hidden unit at a time. Each new (candidate) hidden unit that is added to the network is connected to all of the network inputs, in addition to all of the outputs of the previous hidden units. The candidate unit is treated as a new hidden layer, and since all other connections are frozen, can be trained as if it was a single layer — *i.e.* with the delta rule, or other similar single-layer training algorithm, rather than backpropagation. The candidate unit's input weights are adjusted so as to maximize the covariance of the candidate unit's output with the error output of the network:

$$C = \sum_{o} \left| \sum_{p} (y_{po} - \overline{y_o})(E_{po} - \overline{E_o}) \right|, \tag{3.16}$$

where y_{po} is the output of the output unit o with input pattern p, and E_{po} is the error. The trained unit's input weights are then frozen after training, and if an error threshold hasn't been met, a new layer is again added.

Since each new neuron produces a new layer, cascade correlation leads to very large DNNs with low throughput, since each layer is only one neuron wide, making them very inefficient computationally. Due to their greedy training strategy, cascade correlation networks are trained to perfectly fit the training data, and thus typically overfit. This means they are not likely to generalize as well as standard neural networks trained with backpropagation, and where no weights are frozen during training.

[3] In fact it is covariance that is used rather than correlation, since no normalizing factors are used, as noted by Fahlman and Lebiere (1989).

Upstart Algorithm

The *upstart algorithm* is a method for building binary classification networks, and is guaranteed to find an architecture with perfect training classification. At a high level it sounds like the tiling algorithm of section 3.4, but instead of building up layers from the input, in the upstart algorithm all of the neurons in the network are connected directly to the inputs, and new 'child' neurons are added to compensate for the misclassification errors of existing 'parent' neurons.

Initially the network consists of one neuron that is trained with the *pocket algorithm* (Gallant, 1986). This 'parent' neuron, now frozen, will misclassify some of the training samples and, to correct these, two 'child' neurons are added. Since the neurons in the upstart algorithm are binary threshold units, the 'parent' neuron will either misclassify samples as negative or positive. The two 'child' neurons are then trained (also by the pocket algorithm) to provide enough of a negative/positive signal to cause the parent neuron to give correct classifications for one or more of the misclassified samples. The 'child' neurons become 'parents', and their offspring also learn to correct their mistakes. Empirical results showed that the upstart algorithm produces networks with fewer neurons than the tiling algorithm.

3.5 Pruning Algorithms

Rather than the progressive construction of networks, an alternative approach to learning efficient networks is *pruning*, the removal of unimportant network weights after training.

Amongst the earliest work[4] on pruning was by Sietsma and Dow (1988), who investigated the effect of pruning weights in very small perceptron networks. The pruning rules were summarized as "If the output of a unit does not change for any input pattern that unit is not contributing to the solution. If the outputs of any two units are the same or opposite across all patterns the two units duplicate and one can be removed." (Sietsma and Dow, 1988). Even in these small networks, the complexity of pruning emerges. The authors propose removing weights close to zero, or weights that similarly do not affect the output of the neuron, and removing complementary but opposite (*i.e.* duplicate) weights.

There have been many proposals of methods of pruning network connections since (Castellano, Fanelli, and Pelillo, 1997; Gorodkin *et al.*, 1993; Han, H. Mao, and

[4]Rumelhart is noted by Hanson and Pratt (1988) to have worked on a similar, albeit unpublished, method.

Dally, 2016; Han, Pool, Narang, *et al.*, 2017; Han, Pool, Tran, *et al.*, 2015; Hanson and Pratt, 1988; LeCun, J. S. Denker, and Solla, 1989; Mozer and Smolensky, 1988, 1989; Setiono, 1997; Ullrich, Meeds, and Welling, 2017) differing mostly on the method of evaluating the saliency of weights, *i.e.* how important each weight is to maintaining generalization. Here we will cover only a small selection of the most relevant methods.

As demonstrated even in the early approach of Sietsma and Dow, a naive saliency measure, such as weight magnitude, is overly simplistic in practice since it does not account for the larger distributed representation learned in neural networks and can lead to a large reduction in accuracy if used for pruning.

In a typical pruning algorithm, after having trained the network, a saliency measure for each network weight is calculated. The weights are then sorted by saliency, and the lowest saliency parameters are deleted. After this pruning step, the network must again be re-trained from scratch. This iterative re-training means that such pruning methods do not scale well to contemporary neural networks where training times can be measured in weeks.

The simplest saliency measure would be to directly measure the effect of removing a weight on the training error of the network. For each weight in the network, the training error can be evaluated, and the difference in error used as the saliency measure for that weight. In practice this straight-forward method is infeasible however, since we must evaluate over the training set for each weight of the network. For a network with n weights, and p training samples, calculating the saliencies for the whole network in this manner is $O(np)$ (Hanson and Pratt, 1988).

3.5.1 Optimal Brain Damage

LeCun, J. S. Denker, and Solla (1989) proposed OBD to iteratively remove neurons after training based on a saliency measure that judges the parameters that have the least effect on training error. The authors point out that simply using the magnitude of the weights themselves is equivalent in the limit to having trained with a form of non-proportional weight decay. Instead the authors propose a more theoretically justified saliency measure — they propose measuring the change in the objective function caused by removing a parameter. Rather than performing the computationally intensive task of re-evaluating the objective for every possible parameter deletion, the second derivative of the objective with respect to the weights is instead used. The authors approximate the derivative of an objective function with respect to the weights

using the 2nd-order Taylor series expansion, $f(a) = \sum_{n=0}^{2} \frac{f^{(n)}(a)}{n!}(x-a)^n$, of the error:

$$\delta E = \sum_i \frac{\partial E}{\partial w_i}\delta w_i + \frac{1}{2}\sum_i \frac{\partial^2 E}{\partial w_i^2}\delta w_i^2 + \frac{1}{2}\sum_{i\neq j} \frac{\partial^2 E}{\partial w_i \partial w_j}\delta w_i \delta w_j + O(\|\delta \mathbf{w}\|^3), \tag{3.17}$$

where E is the objective function, w_i is the i^{th} weight, and here δ represents a finite difference rather the 'delta' defined in section 2.1.3. We can write this more compactly in matrix notation,

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \delta \mathbf{w} + \frac{1}{2}\delta \mathbf{w}^T \mathbf{H}\, \delta \mathbf{w} + O(\|\delta \mathbf{w}\|^3), \tag{3.18}$$

where $\mathbf{w}$ is the vector of network weights, and $\mathbf{H}$ is the Hessian matrix of 2nd-order derivatives.

This approximation is infeasible to calculate in practice, mostly due to the size of the Hessian matrix which is square in the number of weights. The authors use several assumptions to approximate this in a more computationally efficient manner. Most importantly they use the "diagonal approximation" of the Hessian, using only the diagonal terms, $h_{i,i} = \partial^2 E/\partial w_i^2$. Further, the "extremal approximation" assumes that pruning is only done once the training has converged, and so it can be assumed that the first term is zero. Finally they assume that the function is approximately quadratic, and thus higher-order terms (*i.e.* $O(\|\delta \mathbf{w}\|^3)$) can be ignored. In all, the final approximation of the objective function's change is simply:

$$\delta E \approx \frac{1}{2}\sum_i \frac{\partial^2 E}{\partial w_i^2}\delta w_i^2. \tag{3.19}$$

To calculate the second derivative, the chain rule is used, and the second derivatives are back-propagated through the network. This approximation is used directly as the saliency measure for each network weight w_i, and after having trained the network, the weights are sorted by saliency, and the lowest saliency parameters are deleted.

3.5.2 Learning both Weights and Connections

Han, Pool, Tran, *et al.* (2015) begins by pruning a trained model using a naive weight magnitude saliency, and then re-training the network in a layer-wise manner. Although the novelty of this method is arguable given the past work on pruning, it presents results on contemporary deep networks. Despite using a naive weight magnitude-based

saliency the results are reasonable with a 9-fold decrease in parameters, with only a small loss in accuracy. However, the experiments were performed on AlexNet which had a very large fully-connected layer composing most of its parameters.

3.6 Compression & Quantization

In order to use DNN on embedded devices, and in many applications, first a more compact representation and perhaps even faster inference is often required. Although mostly a matter of engineering, compression of neural networks also tells us something about the information-theoretic capacity of the models we train, and therefore gives us hints about the internal representation learned in DNNs. There have been many papers on the topic (Han, H. Mao, and Dally, 2016; Han, Pool, Narang, *et al.*, 2017; Han, Pool, Tran, *et al.*, 2015; Kim *et al.*, 2016; Rastegari *et al.*, 2016; Ullrich, Meeds, and Welling, 2017), here we will only cover a few of the most relevant papers.

3.6.1 Deep Compression

Han, H. Mao, and Dally (2016) extends the method of Han, Pool, Tran, *et al.* (2015) with both quantization and Huffman coding. The quantization uses a look-up table to store a limited number of weight values (within each layer) which are indexed by multiple weights — in effect a form of weight sharing. Interestingly the model is further trained after the quantization, using backpropagation with the quantized weights. Finally, the resulting quantized model is compressed losslessly using a Huffman encoding. The results are impressive, a large VGG-16 model of 552MB parameters is reduced in size to 11.3MB, a decrease by a factor of 40.

3.6.2 Deep-Sparse-Dense Training

One of the most interesting recent papers on pruning is that of Han, Pool, Narang, *et al.* (2017) who do something quite different. The authors propose to train a model in distinct initial, sparse and dense steps:

Intial Train a DNN as normal,

Sparse Next prune the network, as in Han, Pool, Tran, *et al.* (2015), with the difference that a different threshold is learned for each layer. The network is then retrained with the pruned connections removed, recovering the original accuracy of the initial network. The authors claim this is akin to a 'regularization' of the network,

Dense Finally the pruned connections are re-introduced and initialzed to zero[5], and the network is finetuned at 1/10 the original learning rate.

The results of this training method are that, even in an efficient large-scale deep network trained on ILSVRC such as GoogLeNet, the top-5 error is decreased by almost 1%, which is significant.

3.6.3 XOR-Net: Binary Convolutional Neural Networks

Rastegari *et al.* (2016) proposed two methods, one in which filter parameters were quantized to binary values, from the typical 16-bit floating point representation. Surprisingly even when the method was applied to models trained on ILSVRC, such as AlexNet, the accuracy was little effected.

A further method, XNOR networks, is proposed where both the filter parameters and feature maps are reduced binary representations. XNOR networks claim to be 58× faster and have 32× smaller convolutional layers. The evaluation for ILSVRC is based on the AlexNet architecture however, which does not represent a good accuracy/model size tradeoff compared to more recent deep network architectures.

3.7 Structural Priors

Chapters 2 and 3 have shown that the concept we have denoted 'structural priors' is not new, and has varied interpretations. Here we will attempt to summarize the concepts related to regularization, network architecture, and generalization covered by this large body of work in a compact nomenclature, much like C. Zhang *et al.* (2017) attempted to define types of regularization. While, like with any compact nomenclature, there is a danger of overly simplifying the details, it can also help in seeing the big picture.

To return to the example of fitting a polynomial curve presented in section 3.2.3, in practice when fitting such a curve, we have little idea of which order polynomial would best fit the data. Necessarily, we must use a relatively higher-order curve to fit the data. Similarly, with a neural network, we rarely have knowledge of the underlying structure of the solution (but when we do, we should use it to parametrize our models appropriately (Jain *et al.*, 2016)). Instead we must use networks with more parameters than necessary to ensure that there is enough capacity to learn the underlying, but

[5]This is considered to be a bad initialization when training neural networks from scratch due to the requirement to break symmetry, yet the authors do not explain the use here. Presumably, because there are already non-zero weights in the network however, symmetry is not as big of an issue.

likely sparse solution. The problem with this approach is that over-parametrization of a model generally leads to poor generalization due to overfitting. To prevent this, there are two general types of methods in which we can relate our prior knowledge that the model is over-parametrized to the optimization:

Weak Structural Prior: Regularization Knowing only that our model is over-parametrized is a relatively weak prior, however we can encode this into the fit by using a regularization penalty. This restricts the model to effectively use only a small number of the parameters by adding a penalty, for example on the ℓ_2 norm of the model weights. For polynomial regression, this is called *ridge regression*, while for neural networks it is called *weight decay* (Geoffery E. Hinton, 1987). In neural networks, early stopping during training is another method for doing this.

Strong Structural Prior: Restricted Connectivity With more prior information on the task, *i.e.* when fitting a polynomial, we may ascertain that a certain order polynomial is more appropriate from the convexity of the polynomial, and restrict learning to that order. For example, given that samples from a polynomial appear to be convex, we can surmise that the polynomial is likely to be of an even or 2^{nd}-order, and restrict our fit to be of that order.

In neural networks, as we have seen in section 2.1.11, a similar effect can be achieved by *removing parameters* that we know are not needed (*i.e.* in CNNs, using filters with local connectivity), or *sharing parameters* we know are redundant (*i.e.* in CNNs, using the same filters for all pixels).

Although neural networks are usually posed as general learning machines, time and again it has been demonstrated that they only truly stand out as a learning method when we use strong structural priors, encoding our prior knowledge of the task in the architecture itself. As observed by J. Denker *et al.* (1987) this may be considered closer to modifying the problem to be solved, rather than changing the learning method. For example, by asking a CNN to learn to classify a dataset, we are asking the network to "classify these images", whereas by asking a fully-connected network to classify the same dataset, we are asking the network "classify this data". The first task is inherently easier because of the assumptions it makes.

4

Spatial Connectivity

> *"Classical work in visual pattern recognition has demonstrated the advantage of extracting local features and combining them to form higher-order features. Such knowledge can be easily built into the network by forcing the hidden units to combine only local sources of information. Distinctive features of an object can appear at various location on the input image. Therefore it seems judicious to have a set of feature detectors that can detect a particular instance of a feature anywhere on the input plane."*
> – Yann LeCun, *Backprop. Applied to Handwritten Zip Code Recognition*, 1989

CNNs (see section 2.1.11) are a highly specialized form of neural network for learning image representations. Their use of *convolutional filters* allows CNNs to learn much more efficient representations, from a memory and computational efficiency standpoint, than a full connected network. Such filters usually have limited spatial extents (*i.e.* width and height, as opposed to channels) and their learned weights are shared across the image's spatial domain to provide translation invariance (Fukushima, 1980; LeCun, Bottou, *et al.*, 1998). Thus, as illustrated in fig. 4.1, in comparison with fully-connected network layers (fig. 4.1(a)), convolutional layers have a much sparser connection structure and use fewer parameters (fig. 4.1(b)). This leads to faster training and inference, better generalization, and higher accuracy.

This chapter focuses on reducing the computational complexity of the convolutional layers of CNNs by further sparsifying their spatial connection structures. Specifically, we show that by representing convolutional filters using a basis space comprising groups of filters of different spatial dimensions (examples shown in fig. 4.1(c, d)), we can significantly reduce the computational complexity of existing state-of-the-art CNNs without compromising classification accuracy.

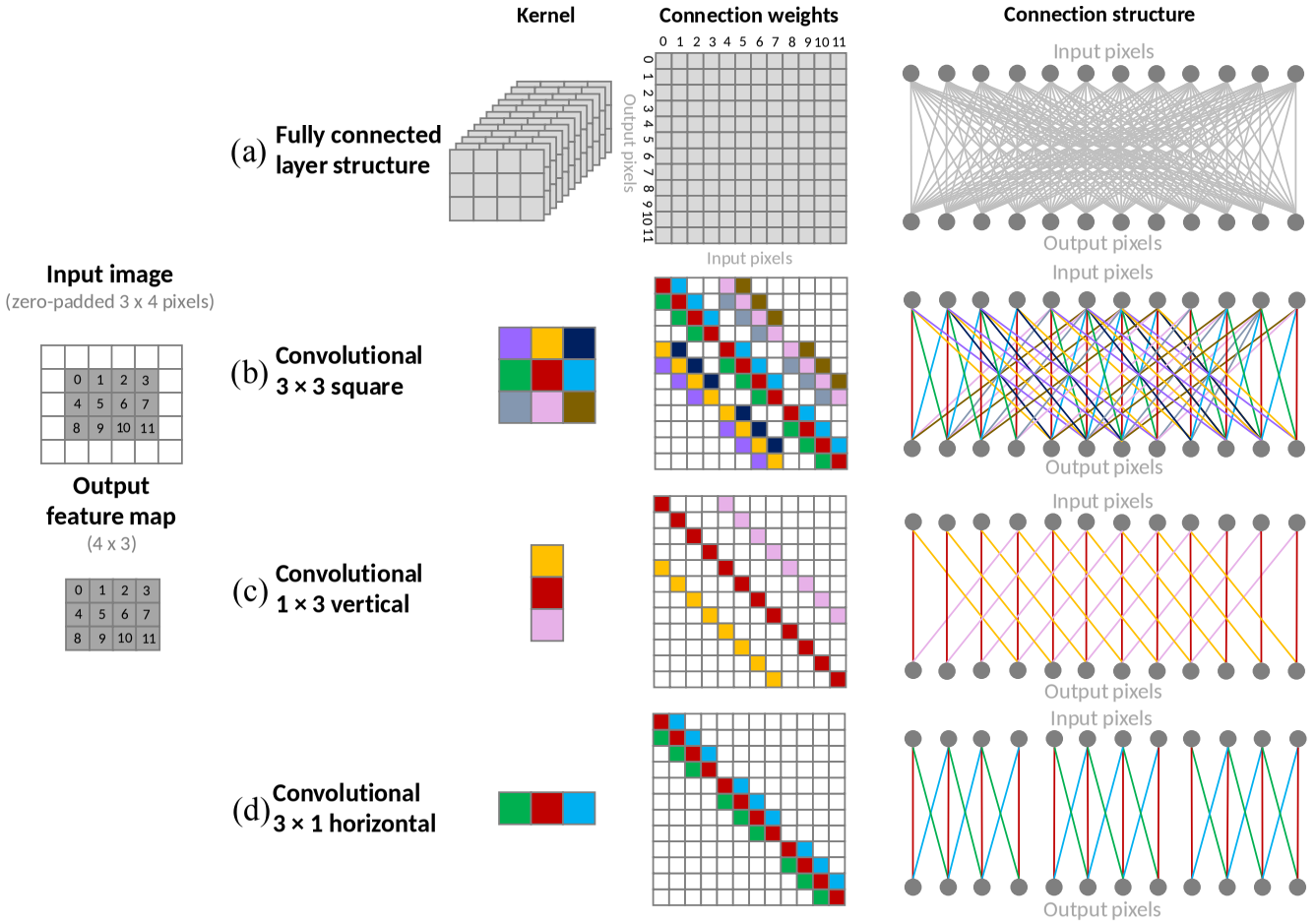


Fig. 4.1 **Network connection structure for convolutional layers.** For a single-layer neural network, the sparsity of a convolutional filter as compared to a fully connected network is illustrated. Connection weight maps (centre) show the pairwise dependencies between input and output pixels. In a fully-connected network (a), each output node is connected to all input pixels. For a CNN (b,c,d), the output pixels depend only on a sparse subset of input pixels, where shared weights are represented by repeated unique colours, and white pixels represent pixels with no connection. Note that sparsity increases from (a) to (d), opening up potentially more efficient implementation.

Our contributions include a novel method of learning a set of small basis filters that are combined to represent larger filters efficiently. Rather than approximating previously trained networks, we train networks *from scratch* and show that our convolutional layer representation can improve both efficiency and classification accuracy. Unlike methods that approximate previously-trained models (as listed in sections 4.1 and 4.1) this allows us to reduce training time, and even increase accuracy over the original model. We further describe how to initialize connection weights effectively for training networks with composite convolutional layers containing groups of differently-shaped filters, which we found to be of critical importance to our training method[1].

4.1 Related Work

There has been much previous work on increasing the test-time efficiency of CNNs. Some promising approaches work by making use of more hardware-efficient representations. For example Gupta *et al.* (2015) and Vanhoucke, Senior, and M. Z. Mao (2011) achieve training- and test-time compute savings by further quantization of network weights that were originally represented as 32-bit floating point numbers. However, more relevant to our work are approaches that depend on new network connection structures, efficient approximations of previously trained networks, and learning low rank filters.

Efficient Network Connection Structures There has been shown to be significant redundancy in the trained weights of CNNs (Denil *et al.*, 2013). LeCun, J. S. Denker, and Solla (1989) suggest a method of pruning unimportant connections within networks. However this requires repeated network re-training and may be infeasible for modern, state-of-the-art CNNs requiring weeks of training time. Lin, Q. Chen, and Yan (2014) show that the geometric increase in the number and dimensions of filters with deeper networks can be managed using low-dimensional embeddings. The same authors show that global average-pooling may be used to decrease model size in networks with fully-connected layers. Simonyan and Zisserman (2015) show that stacked filters with small spatial dimensions (*e.g.* 3×3), can operate on the effective receptive field of larger filters (*e.g.* 5×5) with less computational complexity.

Low-Rank Filter Approximations Rigamonti *et al.* (2013) approximate *previously trained* CNNs with low-rank filters for the semantic segmentation of curvilinear

[1] note that much of this work was done before the widespread use of batch normalization, however inititalization still plays an important role.

structures within volumetric medical imagery. They discuss two approaches: enforcing an ℓ_1-based regularization to learn approximately low rank filters, which are later truncated to enforce a strict rank, and approximating a set of pre-learned filters with a tensor decomposition into many rank-1 filters. Neither approach learns low rank filters directly, and indeed the second approach proved the more successful.

The work of Jaderberg, Vedaldi, and Zisserman (2014) also approximates the existing filters of previously trained networks. They find separable 1D filters through an optimization minimizing the reconstruction error of the already learned full rank filters. They achieve a 4.5× speed-up with a loss of accuracy of 1% in a text recognition problem. However since the method is demonstrated only on text recognition, it is not clear how well it would scale to larger data sets or more challenging problems. A key insight of the paper is that filters can be represented by low rank approximations not only in the spatial domain but also in the channel domain.

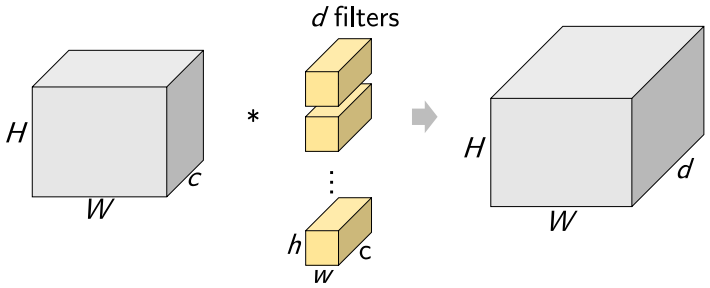
Both of these methods show that, at least for their respective applications, low rank approximations of full-rank filters learned in convolutional networks can increase test-time efficiency significantly. However, being approximations of pre-trained networks, they are unlikely to improve test accuracy, and can only increase the computational requirements during training.

Learning Separable (Factorized) Filters Mamalet and Garcia (2012) propose training networks with separable filters on the task of digit recognition with the MNIST dataset. They train networks with *sequential* convolutional layers of horizontal and vertical 1D filters, achieving a speed-up factor of 1.6×, but with a relative increase in test error of 13% (1.45% *vs.* 1.28%). Our approach is different than this, allowing both horizontal and vertical 1D filters (and other shapes too) on the same layer and avoiding issues with ordering. We also demonstrate a decrease in error, and validate on more challenging datasets.

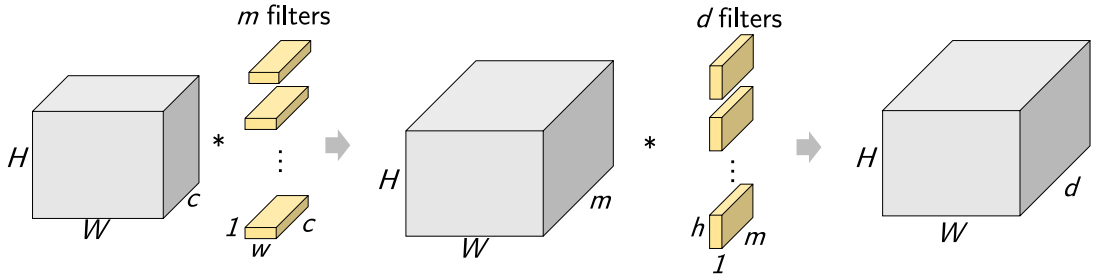
4.2 Using Low-Rank Filters in CNNs

4.2.1 Convolutional Filters

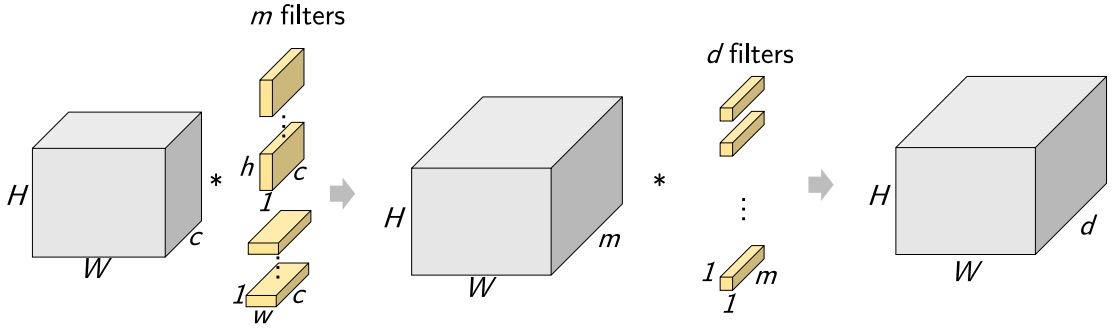
The convolutional layers of a CNN produce output 'images' (usually called *feature maps*) by convolving input images with one or more learned filters. In a typical convolutional layer, as illustrated in fig. 4.2(a), a c-channel input image of size $H \times W$ pixels is convolved with d filters of size $h \times w \times c$ to create a d-channel output image.



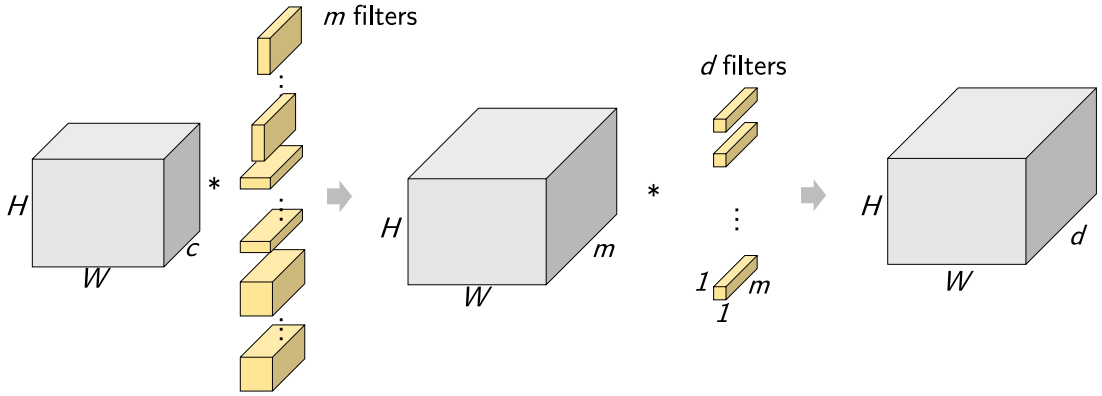
(a) A standard convolutional layer.



(b) Sequential separable filters (Jaderberg, Vedaldi, and Zisserman, 2014).



(c) Our method, a learned basis space of filters that are rectangular in the spatial domain and oriented horizontally and vertically.



(d) Our method, a learned basis space of vertical/horizontal rectangular filters and square filters. Filters of other shapes are also possible.

Fig. 4.2 **Methods of using low-rank filters in CNNs**. Methods from literature and our proposed methods for learning low rank filters. The activation function is not shown, coming after the last layer in each configuration.

Each filter is represented by hwc independent weights. Therefore the computational complexity for the convolution of the filter with a c-channel input image is $O(dwhc)$ (per pixel in the output feature map).

Here we will use our existing mathematical description of convolution from section 2.1.11, where the incoming feature map is denoted by $\mathbf{X}$, outgoing feature map $\mathbf{Y}$, and convolutional filter, or kernel $\mathbf{F}$. The scalar elements of each feature map are $\mathbf{X}_{i,j,k}$, $\mathbf{Y}_{i,j,k}$, where $i = \{0, \ldots, c\}$ is the feature maps channel (*i.e.* colour for an input image), and $j = \{0, \ldots, h\}$, $k = \{0, \ldots, w\}$ are the spatial coordinates, rows and columns respectively, of the channel i image. The filter's scalar elements are $\mathbf{F}_{i,j,k,l}$, where i is the filter's index in the convolutional layer's filter bank and the output channel in $\mathbf{Y}$ to which the filter's result is written, j is the input channel in $\mathbf{X}$ over which the filter's spatial elements are convolved, and (k, l) are the row and column offset between the output and input images.

A convolutional layer, as illustrated in fig. 4.2(a), then convolves across the layer such that,

$$\mathbf{Y}_{i,j,k} = \sum_{l,m,n} \mathbf{X}_{l,j+m,k+n}\, \mathbf{F}_{i,l,m,n}, \tag{4.1}$$

for all valid indices l, m, n, depending on the padding of the input image. We will express this in shorter terms using the convolution operator $*$ and allowing considering only a single pixel (*i.e.* fixed j, k) output spatially for simplicity,

$$\mathbf{Y}_i = \sum_l \mathbf{X}_l * \mathbf{F}_{il}. \tag{4.2}$$

In what follows, we describe schemes for modifying the architecture of the convolutional layers so as to reduce computational complexity. The idea is to replace expensive, full-rank spatial convolutional filters, with modified versions that represent the same number of effective filters by a linear combinations of smaller basis vectors.

4.2.2 Sequential Separable Filters

An existing scheme for reducing the computational complexity of convolutional layers (Jaderberg, Vedaldi, and Zisserman, 2014) is to replace each one with a sequence of two regular convolutional layers but with filters that are rectangular in the spatial domain, as shown in fig. 4.2(b).

The first convolutional layer has m horizontal filters $\mathbf{F}_{i,\,l=0,\ldots,m}$ of size $w \times 1 \times c$, producing an output feature map with m channels. The second convolutional layer

has d vertical filters $\mathbf{F}_{i,\,l=0,\ldots,d}$ of size $1\times h\times m$, producing an output feature map with d channels.

Mathematically, the seperable convolution illustrated in fig. 4.2(b) can be expressed,

$$\begin{aligned}\mathbf{Y}_i &= \sum_l \mathbf{Y}_l^{\mathrm{h}} * \mathbf{F}_{il}^{\mathrm{v}} \\ &= \sum_l \left(\mathbf{X}_l^{\mathrm{h}} * \mathbf{F}_{il}^{\mathrm{h}}\right) * \mathbf{F}_{il}^{\mathrm{v}},\end{aligned} \tag{4.3}$$

where $\mathbf{X}^{\mathrm{h}}$ and $\mathbf{X}^{\mathrm{v}}$ are the input feature maps convolved with the horizontal and vertical filters $\mathbf{F}^{\mathrm{h}}$ and $\mathbf{F}^{\mathrm{v}}$ respectively.

By these means the full rank original convolutional filter bank is represented by a low rank approximation formed from a linear combination of a set of separable $w\times h$ basis filters. The computational complexity of this scheme is $O(mcw)$ for the first layer of horizontal filters and $O(dmh)$ for the second layer of vertical filters, with a total of $O(m(cw + dh))$.

Note that Jaderberg, Vedaldi, and Zisserman (2014) use this scheme to approximate existing full rank filters belonging to previously trained networks using a retrospective fitting step. In this work, by contrast, we *train* networks containing convolutional layers with this architecture from scratch. In effect, we learn the separable basis filters and their combination weights simultaneously during network training.

4.2.3 Filters as Linear Combinations of a Basis

We introduce a novel method for reducing convolutional layer complexity by training with low-rank filters. This works by representing convolutional filters as linear combinations of basis filters as illustrated in fig. 4.2(c). This scheme uses *composite layers* comprising several sets of filters where the filters in each set have different spatial dimensions (see fig. 4.4). The outputs of these basis filters may be combined in a subsequent layer containing filters with spatial dimensions 1×1.

This configuration is illustrated in fig. 4.2(c), and can be expressed as,

$$\begin{aligned}\mathbf{Y}_i &= \sum_l \mathbf{Y}_l^{\text{basis}} * \mathbf{F}_{il}^{\text{weights}} \\ &= \sum_l f_{il}^{\text{weights}} \mathbf{Y}_l^{\text{basis}} \quad \text{(since } \mathbf{F}_{il}^{\text{weights}} \text{ is a scalar)} \\ &= \sum_{l=0}^{m/2} f_{il}^{\text{weights}} \mathbf{X}_l * \mathbf{F}_{il}^{\mathrm{h}} + \sum_{l=m/2}^{m} f_{il}^{\text{weights}} \mathbf{X}_l * \mathbf{F}_{il}^{\mathrm{v}},\end{aligned} \tag{4.4}$$

where $\mathbf{F}^{\mathrm{h}}$ and $\mathbf{F}^{\mathrm{v}}$ are the horizontal and vertical filters respectively.

Here, our composite layer contains horizontal $w\times1$ and vertical $1\times h$ filters, the outputs of which are concatenated in the channel dimension, resulting in an intermediate m-channel feature map. These filter responses are then linearly combined by the next layer of d 1×1 filters to give a d-channel output feature map. In this case, the filters are applied on the input feature map with c channels and followed by a set of m 1×1 filters over the m output channels of the basis filters. If the number of horizontal and vertical filters is the same, the computational complexity is $O(m(wc/2 + hc/2 + d))$.

The effective filters learned in our models are low-rank in that, although we only learn mostly basis filters much smaller than the original networks (*e.g.* $1\times h$, $w\times1$), the effective filter size when also using only a few full $w\times h$ basis filters is still a full 3×3. Necessarily, some of the parameters in our effective filters are linear combinations of others, since the effective filter is a learned linear combination of the low-rank basis filters.

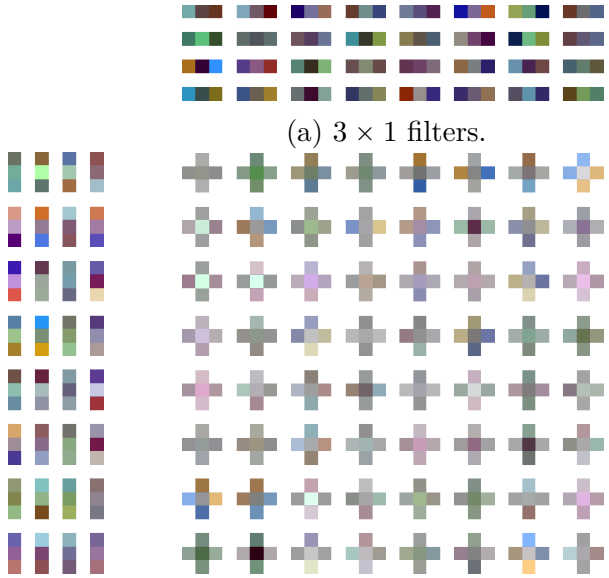
Interestingly, the configuration of fig. 4.2(c), where we only use horizontal and vertical basis filters, gives rise to linear combinations of horizontal and vertical filters that are cross-shaped in the spatial domain. This is illustrated in fig. 4.3 for filters learned in the first convolutional layer of the 'vgg-gmp-lr-join' model that is described in section 4.4, where it is trained using the ILSVRC dataset.

Note that, in general, more than two different sizes of basis filter might be used in the composite layer. In the more general case, for a set of heterogeneous filter groups $\mathbf{F}^{g=0,\ldots,G}$, we can express this as

$$\mathbf{Y}_i = \sum_l f_{il}^{\text{weights}} \sum_g \mathbf{X}_l * \mathbf{F}_{il}^g. \tag{4.5}$$

For example, fig. 4.2(d) shows a combination of three sets of filters with spatial dimensions $w\times1$, $1\times h$, and $w\times h$. Also note that an interesting option is to omit the 1×1 linear combination layer and instead allow the connection weights in a subsequent network layer to learn to combine the basis filters of the preceding layer (despite any intermediate non-linearity, *e.g.* ReLUs). This possibility is explored empirically in the section 4.4.

In that our method uses a combination of filters in a composite layer, it is similar to the 'GoogLeNet' of Szegedy, Liu, *et al.* (2015) which uses Inception modules comprising several (square) filters of different sizes ranging from 1×1 to 5×5. In our case, however, we are implicitly learning linear combinations of less computationally expensive filters with different orientations (*e.g.* 3×1 and 1×3 filters), rather than combinations of



(a) 3×1 filters.

(b) 1×3 filters.

(c) Learned linear combinations.

Fig. 4.3 **Learned Cross-Shaped Filters**. The cross-shaped filters (c) learned as weighted linear combination of (b) 1×3 and (c) 3×1 basis filters in the first convolutional layer of the the 'vgg-gmp-lr-join' model trained using the ILSVRC dataset.

filters of different sizes. Amongst networks with similar computational requirements, GoogLeNet is one of the most accurate for large scale image classification tasks (see fig. 4.6), partly due to the use of heterogeneous filters in the Inception modules, but also the use of low-dimensional embeddings and global pooling.

4.3 Training CNNs with Mixed-Shape Low-Rank Filters

To determine the standard deviations to be used for weight initialization, we use an approach similar to that described by Glorot and Y. Bengio (2010) (with the adaptation described by He *et al.* (2015) for layers followed by a ReLU). In section 4.3.1, we show the details of our derivation, generalizing the approach of He *et al.* (2015) to the initialization of composite layers comprising several groups of filters of different spatial dimensions (see section 4.3.1, fig. 4.4).

At the start of training, network weights are initialized at random using samples drawn from a Gaussian distribution with a standard deviation parameter specified separately for each layer. We found that the setting of these parameters was critical to

the success of network training and difficult to get right, particularly because published parameter settings used elsewhere were not suitable for our new network architectures. With unsuitable weight initialization, training may fail due to *exploding gradients*, where back-propagated gradients grow so large as to cause numeric overflow, or *vanishing gradients* where back-propagated gradients diminish such that their effect is dwarfed by that of weight decay such that loss does not decrease during training (Hochreiter *et al.*, 2001).

The approach of Glorot and Y. Bengio (2010) works by ensuring that the magnitudes of back-propagated gradients remain approximately the same throughout the network. Otherwise, if the gradients were inappropriately scaled by some factor (*e.g.* β) then the final back-propagated signal would be scaled by a potentially much larger factor (β^L after L layers) (see section 2.2.2).

4.3.1 Derivation of the Initialization for Composite layers

In what follows, we adopt notation similar to that of He *et al.* (2015), and follow their derivation of the appropriate standard deviation for weight initialization. However, we also generalize their approach to the initialization of composite layers comprising several groups of filters of different spatial dimensions (see fig. 4.4).

Forward Propagation The response of the l^{th} convolutional layer can be represented as,

$$\mathbf{y}_l = \mathbf{W}_l \mathbf{x}_l + \mathbf{b}_l, \tag{4.6}$$

where $\mathbf{y}_l$ is a $d{\times}1$ vector representing a pixel in the output feature map, and $\mathbf{x}_l$ is a $whc \times 1$ vector that represents a $w{\times}h$ sub-region of the c-channel input feature map. $\mathbf{W}_l$ is the $d{\times}n$ weight matrix, where d is the number of filters and n is the size of a filter, *i.e.* $n = whc$ for a filter with spatial dimensions $w{\times}h$ operating on an input feature map of c channels, and $\mathbf{b}_l$ is the bias. Finally $\mathbf{x}_l = f(\mathbf{y}_{l-1})$ is the output of the previous layer passed through an activation function f (*e.g.* the application of a ReLU to each element of $\mathbf{y}_{l-1}$).

Backward Propagation During backpropagation, the gradient of a convolutional layer is computed as,

$$\Delta\mathbf{x}_l = \hat{\mathbf{W}}_l \Delta\mathbf{y}_l, \tag{4.7}$$

where $\Delta\mathbf{x}_l$ and $\Delta\mathbf{y}_l$ denote the derivatives of loss $\mathcal{L}$ with respect to input and output pixels. $\Delta\mathbf{x}_l$ is a $c{\times}1$ vector of gradients with respect to the channels of a single pixel

in the input feature map and $\Delta\mathbf{y}$ represents $h \times w$ pixels in d channels of the output feature map. $\hat{\mathbf{W}}_l$ is a $c \times \hat{n}$ matrix, and $\hat{n} = whd$. Note that $\hat{\mathbf{W}}_l$ can be simply reshaped from $\mathbf{W}_l^\top$. Also note that the elements of $\Delta\mathbf{y}_l$ correspond to pixels in the output image that had a forward dependency on the input image pixel corresponding to $\Delta\mathbf{x}$. In backpropagation, each element Δy_l of $\Delta\mathbf{y}_l$ is related to an element Δx_{l+1} of some $\Delta\mathbf{x}_{l+1}$ (*i.e.* a back-propagated gradient in the next layer) by the derivative of the activation function f:

$$\Delta y_l = f'(y_l)\Delta x_{l+1}, \tag{4.8}$$

where f' is the derivative of the activation function.

Weight Initialization Now let Δy_l, Δx_l and w_l be scalar random variables that describe the distribution of elements in $\Delta\mathbf{y}_l$, $\Delta\mathbf{x}_l$ and $\hat{\mathbf{W}}_l$ respectively. Then, assuming $f'(y_l)$ and Δx_{l+1} are independent,

$$\mathrm{E}[\Delta y_l] = \mathrm{E}[f'(y_l)]\,\mathrm{E}[\Delta x_{l+1}]. \tag{4.9}$$

For the ReLU case, $f'(y_l)$ is zero or one with equal probability. Like Glorot and Y. Bengio (2010), we assume that w_l and Δy_l are independent. Thus, eq. (4.7) implies that Δx_l has zero mean for all layers l, when w_l is initialized by a distribution that is symmetric around zero. Thus we have $\mathrm{E}[\Delta y_l] = \frac{1}{2}\mathrm{E}[\Delta x_{l+1}] = 0$ and also $\mathrm{E}[(\Delta y_l)^2] = \mathrm{Var}[\Delta y_l] = \frac{1}{2}\mathrm{Var}[\Delta x_{l+1}]$. Now, since each element of $\Delta\mathbf{x}_l$ is a summation of $\hat{n}$ products of elements of $\hat{\mathbf{W}}_l$ and elements of $\Delta\mathbf{y}_l$, we can compute the variance of the gradients in eq. (4.7):

$$\begin{aligned}\mathrm{Var}[\Delta x_l] &= \hat{n}\mathrm{Var}[w_l]\mathrm{Var}[\Delta y_l] \\ &= \frac{1}{2}\hat{n}\mathrm{Var}[w_l]\mathrm{Var}[\Delta x_{l+1}].\end{aligned} \tag{4.10}$$

To avoid scaling the gradients in the convolutional layers (and so avoid exploding or vanishing gradients), we set the ratio between these variances to 1:

$$\frac{1}{2}\hat{n}\mathrm{Var}[w_l] = 1. \tag{4.11}$$

This leads to the result of He *et al.* (2015), in that a layer with $\hat{n}_l$ connections followed by a ReLU activation function should be initialized with a zero-mean Gaussian distribution with standard deviation $\sqrt{2/\hat{n}_l}$.

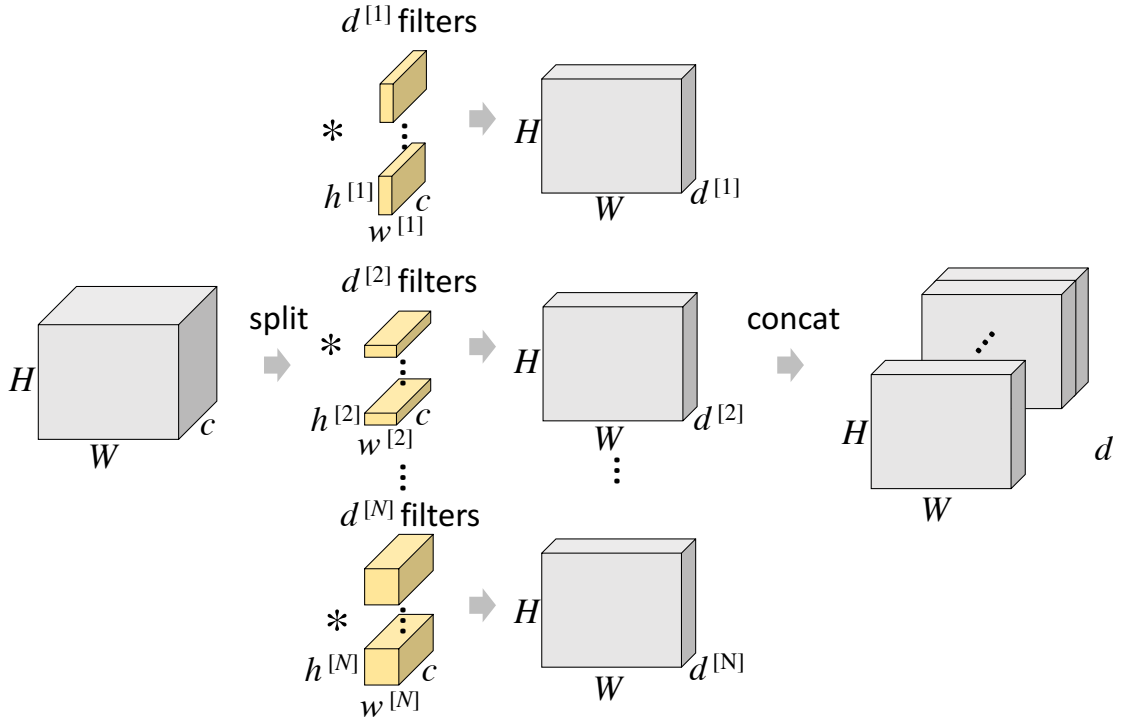


Fig. 4.4 **A composite convolutional layer**. Composite layers convolve an input feature map with N groups of convolutional filters of several different spatial dimensions. Here the i^{th} group has $d^{[g]}$ filters with spatial dimension $w^{[g]} \times h^{[g]}$. The outputs are concatenated to create a d channel output feature map. Composite layers require careful weight initialization to avoid vanishing/exploding gradients during training.

Weight Initialization in Composite layers The initialization scheme described above assumes that the layer comprises filters of spatial dimension $w \times h$. Now we extend this scheme to composite convolutional layers containing N groups of filters of different spatial dimensions $w^{[g]} \times h^{[g]}$ (where superscript $[g]$ denotes the group index and with $g \in \{1, \ldots, N\}$). Now the layer response is the concatenation of the responses of each group of filters:

$$\mathbf{y}_l = \begin{bmatrix} \mathbf{W}_l^{[1]} \mathbf{x}_l^{[1]} \\ \mathbf{W}_l^{[2]} \mathbf{x}_l^{[2]} \\ \cdots \\ \mathbf{W}_l^{[N]} \mathbf{x}_l^{[N]} \end{bmatrix} + \mathbf{b}_l. \tag{4.12}$$

As before $\mathbf{y}_l$ is a $d \times 1$ vector representing the response at one pixel of the output feature map. Now each $\mathbf{x}^{[g]}$ is a $w^{[g]} h^{[g]} c \times 1$ vector that represents a different shaped $w^{[g]} \times h^{[g]}$ sub-region of the input feature map. Each $\mathbf{W}_l^{[g]}$ is the $c_l^{[g]} \times \hat{n}^{[g]}$ weight matrix, where d is the number of filters and $\hat{n}^{[g]}$ is the size of a filter, *i.e.* $\hat{n}^{[g]} = w^{[g]} h^{[g]} c^{[g]}$ for a filter of spatial dimension $w^{[g]} \times h^{[g]}$ operating on an input feature map of $c_l = d_{l-1}$ channels.

During backpropagation, the gradient of the composite convolutional layer is computed as a summation of the contributions from each group of filters:

$$\Delta \mathbf{x}_l = \hat{\mathbf{W}}_l^{[1]} \Delta \mathbf{y}_l^{[1]} + \hat{\mathbf{W}}_l^{[2]} \Delta \mathbf{y}_l^{[2]} + \cdots + \hat{\mathbf{W}}_l^{[N]} \Delta \mathbf{y}_l^{[N]}, \tag{4.13}$$

where now $\Delta \mathbf{y}^{[g]}$ represents $w^{[g]} \times h^{[g]}$ pixels in $d^{[g]}$ channels of the output feature map. Each $\hat{\mathbf{W}}_l^{[g]}$ is a $c_l \times \hat{n}^{[g]}$ matrix of weights arranged appropriately for backpropagation. Again, note that each $\hat{\mathbf{W}}_l^{[g]}$ can be simply reshaped from $\mathbf{W}_l^{[g]}$.

As before, each element of $\Delta \mathbf{y}_l$ is a sum over $\hat{n}$ products between elements of $\hat{\mathbf{W}}_l^{[g]}$ and elements of $\Delta \mathbf{y}_l^{[g]}$ and here $\hat{n}$ is given by:

$$\hat{n} = \sum w^{[g]} h^{[g]} d^{[g]}. \tag{4.14}$$

In the case of a ReLU non-linearity, this leads to a zero-mean Gaussian distribution with standard deviation:

$$\sigma = \sqrt{\frac{2}{\sum w^{[g]} h^{[g]} d^{[g]}}}. \tag{4.15}$$

In conclusion, a composite layer of heterogeneously-shaped filter groups, where each filter group i has $w^{[g]} h^{[g]} d^{[g]}$ outgoing connections should be initialized as if it is a single-layer with $\hat{n} = \sum w^{[g]} h^{[g]} d^{[g]}$. Thus in the case of a ReLU non-linearity,

we find that such a composite layer should be initialized with a zero-mean Gaussian distribution with standard deviation given in eq. (4.15).

4.4 Results

To validate our approach, we show that we can replace the filters used in existing state-of-the-art network architectures with low-rank representations as described above to reduce computational complexity without reducing accuracy. Here we characterize the computational complexity of a CNN using the number of multiply-accumulate operations required for a forward pass (which depends on the size of the filters in each convolutional layer as well as the input image size and stride).

4.4.1 Multiply-Accumulate Operations and Caffe CPU/GPU Timings

We have characterized the computational complexity of a CNN using the number of multiply-accumulate operations required for a forward pass (which depends on the size of the filters in each convolutional layer as well as the input image size and stride), to give as close as possible to a hardware and implementation independent evaluation the computational complexity of our method. However, we have observed strong correlation between multiply-accumulate counts and run-time for both CPU and GPU implementations of the networks described here (as shown in fig. 4.5). Note that the Caffe timings differ more for the initial convolutional layers where the input sizes are much smaller (3-channels), and BLAS is less efficient for the relatively small matrices being multiplied.

4.4.2 Methodology

We augment our training set with randomly cropped and mirrored images, but do not use any scale or photometric augmentation, or over-sampling. This allows us to compare the efficiency of different network architectures without having to factor in the computational cost of the various augmentation methods used elsewhere. During training, for every model except GoogLeNet, we adjust the learning rate according to the schedule,

$$\gamma_t = \gamma_0(1 + \gamma_0 \lambda t)^{-1}, \tag{4.16}$$

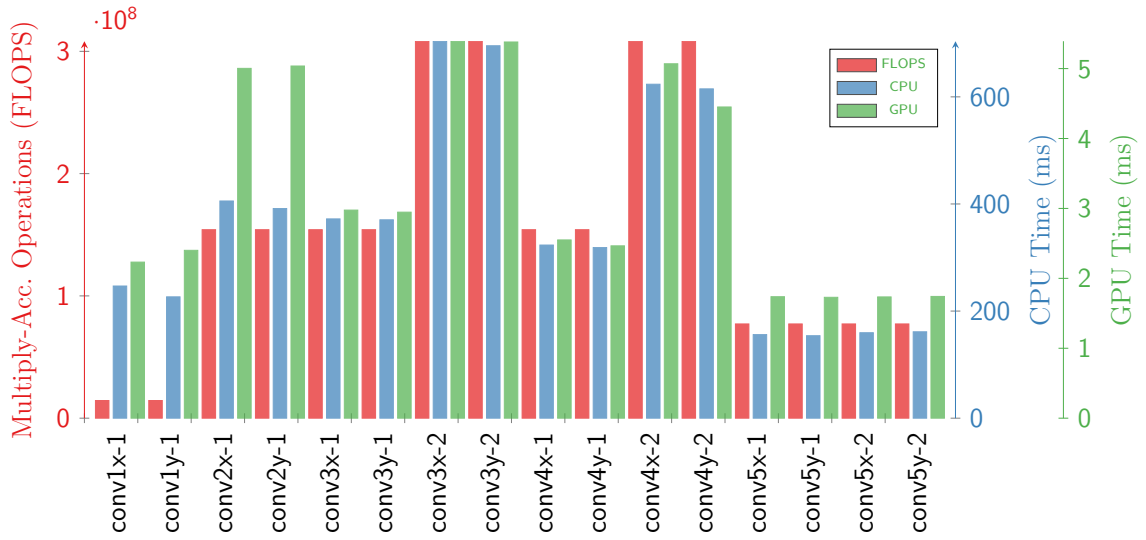


Fig. 4.5 **Multiply-Accumulate Operations and Caffe CPU/GPU Timings.** For the forward pass of each convolutional layer in the 'vgg-gmp-lr' network. Caffe CPU timings were well correlated with multiply-accumulate operations for most layers. GPU timings in some cases are relatively slower however. Please see section 5.4 for an explanation of this discrepancy.

where γ_0, γ_t and λ are the initial learning rate, learning rate at iteration t, and weight decay respectively (Bottou, 2012). When the validation accuracy levels off we manually reduce the learning rate by further factors of 10 until the validation accuracy no longer increases. Unless otherwise indicated, aside from changing the standard deviation of the normally distributed weight initialization, as explained in section 4.3, we used the standard hyper-parameters for each given model. Our results use no test-time augmentation.

4.4.3 VGG-11 Architectures for ILSVRC Object Classification and MIT Places Scene Classification

We evaluated classification accuracy of the VGG-11 based architectures using two datasets, ILSVRC (Jia *et al.*, 2014) and MIT Places (Zhou *et al.*, 2014). The ILSVRC dataset comprises 1.2M training images of 1000 object classes, commonly evaluated by top-1 and top-5 accuracy on the 50K image validation set. The MIT Places dataset comprises 2.4M training images from 205 scene classes, evaluated with top-1 and top-5 accuracy on the 20K image validation set.

VGG-11 ('VGG-A') is an 11-layer convolutional network introduced by Simonyan and Zisserman (2015). It is in the same family of network architectures used by He

Table 4.1 **VGG ILSVRC Results.** Accuracy, multiply-accumulate count, and number of parameters for the baseline VGG-11 network (both with and without GAP) and more efficient versions created by the methods described in this chapter.

Network	Stride	FLOPS $\times 10^9$	Param. $\times 10^7$	Top-1 Acc.	Top-5 Acc.
vgg-11	1	7.61	13.29	0.649	0.862
gmp	1	7.51	3.22	0.685	0.887
gmp-sf	1	6.53	2.97	0.673	0.879
gmp-lr-join-wfull	1	6.34	3.72	0.704	0.897
gmp-lr-join	1	3.85	2.73	0.675	0.880
gmp-lr-2x	1	3.14	3.13	0.693	0.889
gmp-lr	1	2.52	2.61	0.676	0.880
gmp-lr-lde	2	1.02	2.64	0.667	0.875

Table 4.2 **MIT Places Results.** Accuracy, multiply-accumulate operations, and number of parameters for the baseline 'vgg-11-gmp' network, separable filter network as described by Jaderberg, Vedaldi, and Zisserman (2014), and more efficient models created by the methods described in this chapter. All networks were trained at stride 2 for the MIT Places dataset.

Network	Stride	FLOPS $\times 10^8$	Param. $\times 10^7$	Top-1 Acc.	Top-5 Acc.
gmp	2	18.77	3.22	0.526	0.830
gmp-sf	2	16.57	13.03	0.517	0.824
gmp-lr-join	2	9.64	2.73	0.512	0.821
gmp-lr	2	6.30	2.61	0.520	0.825

et al. (2015) and Simonyan and Zisserman (2015) to obtain the state-of-the-art accuracy for ILSVRC, but uses fewer convolutional layers and therefore fits on a single GPU during training. During training of our VGG-11 based models, we used the standard hyperparameters detailed by Simonyan and Zisserman (2015) and the initialization of He *et al.* (2015).

VGG-derived Model Table

Table 4.3 shows the architectural details of the VGG-11-derived models used in section 4.4.3. In what follows, we compare the accuracy of a number of different network architectures detailed in table 4.3. Results for ILSVRC are given in table 4.1, and plotted in fig. 4.6. Results for MIT Places are given in table 4.2, and plotted in fig. 4.7.

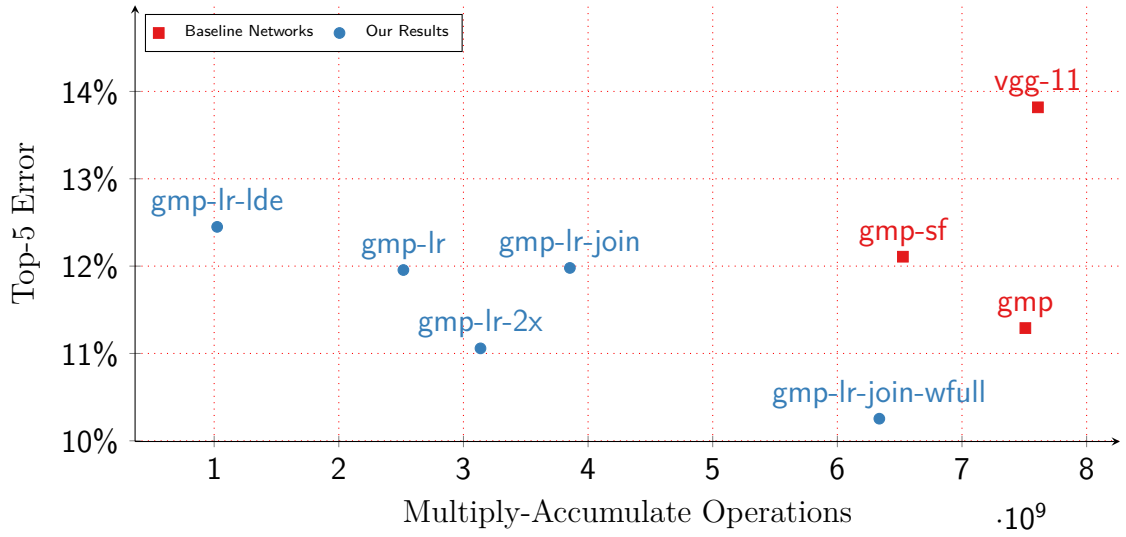


Fig. 4.6 **VGG ILSVRC Results.** Multiply-accumulate operations *vs.* top-5 error for VGG-derived models on ILSVRC object classification dataset, the most efficient networks are closer to the origin. Our models are significantly faster than the baseline network, in the case of 'gmp-lr-2x' by a factor of almost 60%, while slightly lowering error. Note that the 'gmp-lr' and 'gmp-lr-join' networks have the same accuracy, showing that an explicit linear combination layer may be unnecessary.

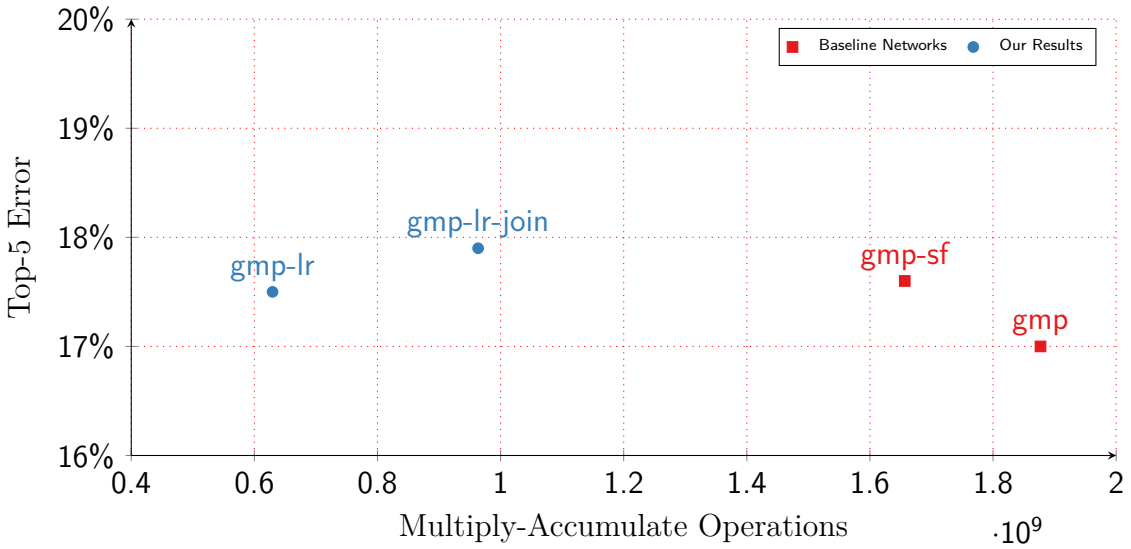


Fig. 4.7 **MIT Places Results.** Multiply-accumulate operations *vs.* top-5 error for VGG-derived models on MIT Places scene classification dataset.

Baseline (Global Max Pooling) Compared to the version of the network described by Simonyan and Zisserman (2015), we use a variant that replaces the final 2×2 max pooling layer before the first fully-connected layer with a global max pooling operation, similar to the global average pooling used by Lin, Q. Chen, and Yan (2014) and Szegedy, Liu, *et al.* (2015). We evaluated the accuracy of the baseline VGG-11 network with global max-pooling (**vgg-gmp**) and without (**vgg-11**) on the two datasets. We trained these networks at stride 1 on the ILSVRC dataset and at stride 2 on the larger MIT Places dataset. This globally max-pooled variant of VGG-11 uses over 75% fewer parameters than the original network and gives consistently better accuracy – almost 3 percentage points lower top-5 error on ILSVRC than the baseline VGG-11 network on ILSVRC (see table 4.1). We used this network as the baseline for the rest of our experiments.

Separable Filters To evaluate the separable filter approach described in section 4.2.2 (illustrated in fig. 4.2(b), we replaced each convolutional layer in VGG-11 with a sequence of two layers, the first containing horizontally-oriented 1×3 filters and the second containing vertically-oriented 3×1 filters (**vgg-gmp-sf**). These filters applied in sequence represent 3×3 kernels using a low-dimensional basis space. Unlike Jaderberg, Vedaldi, and Zisserman (2014), we trained this network from scratch instead of approximating the full-rank filters in a previously trained network. Compared to the original VGG-11 network, the separable filter version requires approximately 14% less computation. Results are shown in table 4.1 for ILSVRC and table 4.2 for MIT Places. Accuracy for this network is approx. 0.8% lower than that of the baseline vgg-11-gmp network for ILSVRC and broadly comparable for MIT Places. This approach does not give such a significant reduction in computational complexity as what follows, but it is nonetheless interesting that separable filters are capable of achieving quite high classification accuracy on such challenging tasks.

Simple Horizontal/Vertical Basis To demonstrate the efficacy of the simple low rank filter representation illustrated in fig. 4.2c, we created a new network architecture (**vgg-gmp-lr-join**) by replacing each of the convolutional layers in VGG-11 (original filter dimensions were 3×3) with a sequence of two layers. The first layer comprises half 1×3 filters and half 3×1 filters whilst the second layer comprises the same number of 1×1 filters. The resulting network is approximately 49% faster than the original and yet it gives broadly comparable accuracy (within 1 percentage point) for both the ILSVRC and MIT Places datasets.

Full-Rank Mixture An interesting question concerns the impact on accuracy of combining a small proportion of 3×3 filters with the 1×3 and 3×1 filters used in 'vgg-gmp-lr-join'. To answer this question, we trained a network, **vgg-gmp-lr-join-wfull**, with a mixture of 25% 3×3 and 75% 1×3 and 3×1 filters, while preserving the total number of filters of the baseline network (as illustrated in fig. 4.2(d)). This network was significantly more accurate than both 'vgg-gmp-lr-join' and the baseline, with a top-5 center crop accuracy of 89.7% on ILSVRC, with a computational saving of approximately 16% over our baseline. We note that the accuracy is approx. 1 percentage point higher than GoogLeNet.

Implicitly Learned Combinations In addition, we try a network similar to vgg-gmp-lr-join but without the 1×1 convolutional layer (as shown in fig. 4.2(c)) used to sum the contributions of 3×1 and 1×3 filters (**vgg-gmp-lr**). Interestingly, because of the elimination of the extra 1×1 layers, this gives an additional computational saving such that this model is only 1/3 of the computation of our baseline, with no reduction in accuracy. This seems to be a consequence of the fact that the subsequent convolutional layer is itself capable of learning effective combinations of filter responses even after the intermediate ReLU non-linearity.

We also trained such a network with double the number of convolutional filters (**vgg-gmp-lr-2x**), *i.e.* with an equal number of 1×3 and 3×1 filters, or $2c$ filters as shown in fig. 4.2(c). We found this to increase accuracy further (88.9% Top-5 on ILSVRC) while still being approximately 58% faster than our baseline network.

Low-Dimensional Embeddings We attempted to reduce the computational complexity of our 'gmp-lr' network further in the **vgg-gmp-lr-lde** network by using a stride of 2 in the first convolutional layer, and adding low-dimensional embeddings, as in Lin, Q. Chen, and Yan (2014) and Szegedy, Liu, *et al.* (2015). We reduced the number of output channels by half after each convolutional layer using 1×1 convolutional layers, as detailed in tables 4.3 and 4.3. While this reduces computation significantly, by approx. 86% compared to our baseline, we saw a decrease in top-5 accuracy on ILSVRC of 1.2 percentage points. We do note however, that this network remains 2.5 percentage points more accurate than the original VGG-11 network, but is 87% faster.

4.4.4 GoogLeNet for ILSVRC Object Classification

GoogLeNet, introduced by Szegedy, Liu, *et al.* (2015), is the most efficient network for ILSVRC, getting close to state-of-the-art results with a fraction of the computation

Table 4.3 **VGG Model Architectures**. Here "3×3, 32" denotes 32 3×3 filters, "/2" denotes stride 2, "fc" denotes fully-connected, and "‖" denotes a concatenation within a composite layer.

Layer	VGG-11	**GMP**	**GMP-SF**	**GMP-LR**	**GMP-LR-2X**	**GMP-LR-JOIN**	**GMP-LR-LDE**	**GMP-LR-JOIN-WFULL**
conv1	3×3, 64		1×3, 64	3×1, 32 ‖ 1×3, 32	3×1, 64 ‖ 1×3, 64	3×1, 32 ‖ 1×3, 32		3×1, 24 ‖ 1×3, 24 ‖ 3×3, 16
			3×1, 64			1×1, 64	1×1, 32	1×1, 64
	ReLU							
	2×2 maxpool, /2							
conv2	3×3, 128		1×3, 128	3×1, 64 ‖ 1×3, 64	3×1, 128 ‖ 1×3, 128	3×1, 64 ‖ 1×3, 64		3×1, 48 ‖ 1×3, 48 ‖ 3×3, 32
			3×1, 128			1×1, 128	1×1, 64	1×1, 128
	ReLU							
	2×2 maxpool, /2							
conv3	3×3, 256		1×3, 256	3×1, 128 ‖ 1×3, 128	3×1, 256 ‖ 1×3, 256	3×1, 128 ‖ 1×3, 128		3×1, 96 ‖ 1×3, 96 ‖ 3×3, 64
			3×1, 256			1×1, 256	1×1, 128	1×1, 256
	ReLU							
	3×3, 256		1×3, 256	3×1, 128 ‖ 1×3, 128	3×1, 256 ‖ 1×3, 256	3×1, 128 ‖ 1×3, 128		3×1, 96 ‖ 1×3, 96 ‖ 3×3, 64
			3×1, 256			1×1, 256	1×1, 128	1×1, 256
	ReLU							
	2×2 maxpool, /2							
conv4	3×3, 512		1×3, 512	3×1, 256 ‖ 1×3, 256	3×1, 512 ‖ 1×3, 512	3×1, 256 ‖ 1×3, 256		3×1, 192 ‖ 1×3, 192 ‖ 3×3, 128
			3×1, 512			1×1, 512	1×1, 256	1×1, 512
	ReLU							
	3×3, 512		1×3, 512	3×1, 256 ‖ 1×3, 256	3×1, 512 ‖ 1×3, 512	3×1, 256 ‖ 1×3, 256		3×1, 192 ‖ 1×3, 192 ‖ 3×3, 128
			3×1, 512			1×1, 512	1×1, 256	1×1, 512
	ReLU							
	2×2 maxpool, /2							
conv5	3×3, 512		1×3, 512	3×1, 256 ‖ 1×3, 256	3×1, 512 ‖ 1×3, 512	3×1, 256 ‖ 1×3, 256		3×1, 192 ‖ 1×3, 192 ‖ 3×3, 128
			3×1, 512			1×1, 512	1×1, 256	1×1, 512
	ReLU							
	3×3, 512		1×3, 512	3×1, 256 ‖ 1×3, 256	3×1, 512 ‖ 1×3, 512	3×1, 256 ‖ 1×3, 256		3×1, 192 ‖ 1×3, 192 ‖ 3×3, 128
			3×1, 512			1×1, 512	1×1, 256	1×1, 512
	ReLU							
	2×2 maxpool, /2		global maxpool					
fc6	$7^2 \times 512 \times 4096$		512×4096					
	ReLU							
fc7	4096×4096							
	ReLU							
fc8	4096×1000							
	softmax							

Table 4.4 **GoogLeNet ILSVRC Results.** Accuracy, multiply-accumulate count, and number of parameters for the baseline GoogLeNet network and more efficient versions created by the methods described in this chapter.

Network	FLOPS $\times 10^9$	Test Param. $\times 10^6$	Top-1 Acc.	Top-5 Acc.
GoogLeNet	1.59	5.97	0.677	0.883
lr	1.18	3.50	0.673	0.880
lr-conv1	0.84	3.42	0.659	0.870

and model size of even VGG-11. The GoogLeNet Inception module is a composite layer of 5 homogeneously-shaped filters, 1×1, 3×3, 5×5, and the output of a 3×3 average pooling operations. All of these are concatenated and used as input for successive layers (see section 2.3.4).

For the **googlenet-lr** network, within only the Inception modules we replaced each the 3×3 filters with low-rank 3×1 and 1×3 filters, and replaced the layer of 5×5 filters with a set of low-rank 5×1 and 1×5 filters. For the **googlenet-lr-conv1** network, we similarly replaced the first and second layer convolutional layers with 7×1 / 1×7 and 3×1 / 1×3 layers respectively.

Results are shown in table 4.4, and fig. 4.8. GoogLeNet uses intermediate losses and fully connected layers only at training time, at test time these are removed. Test-time model size is thus significantly smaller than training time model size. Table 4.4 also reports test-time model size. The low-rank network delivers comparable classification accuracy using 26% less compute. No other networks produce comparable accuracy within an order of magnitude of compute. We note that although the Caffe pre-trained GoogLeNet model (Jia *et al.*, 2014) has a top-5 accuracy of 0.889, our training of the same network using the given model definition, including the hyper-parameters and training schedule, but a different random initialization had a top-5 accuracy of 0.883.

4.4.5 NiN for CIFAR-10 Object Classification

The CIFAR-10 dataset consists of 60,000 32 × 32 images in 10 classes, with 6000 images per class. This is split into standard sets of 50,000 training images, and 10,000 test images (Krizhevsky, 2009). As a baseline for the CIFAR-10 dataset, we used the NiN architecture (Lin, Q. Chen, and Yan, 2014), which has a published test-set error of 8.81%. We also used random crops during training, with which the network has an error of 8.1%. Like most state-of-the-art CIFAR results, this was with ZCA pre-processed training and test data (I. J. Goodfellow *et al.*, 2013), training time mirror

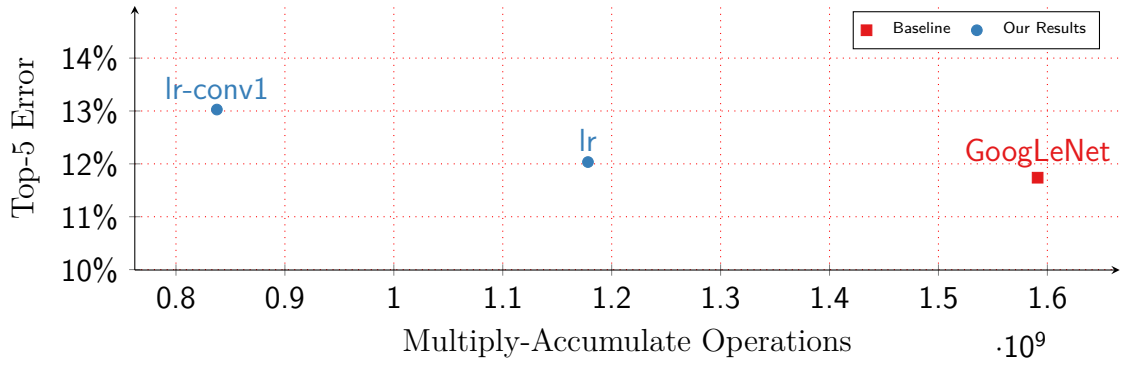


Fig. 4.8 **GoogLeNet ILSVRC Results.** Multiply-accumulate operations *vs.* top-5 error for GoogLeNet-derived models on ILSVRC object classification dataset.

Table 4.5 **NiN CIFAR-10 Results.** Accuracy, multiply-accumulate operations, and number of parameters for the baseline NiN model and more efficient versions created by the methods described in this chapter.

Network	FLOPS $\times 10^8$	Param. $\times 10^5$	Accuracy
NiN	1.93	9.67	0.9188
nin-c3	1.43	7.74	0.9186
nin-c3-lr	1.04	4.38	0.9178

augmentation and random sub-crops. The results of our CIFAR experiments are listed in table 4.5 and plotted in fig. 4.9.

This architecture uses 5×5 filters in some layers. We found that we could replace all of these with 3×3 filters, with comparable accuracy. As suggested by Simonyan and Zisserman (2015), stacked 3×3 filters have the effective receptive field of larger filters with less computational complexity. In this **nin-c3** network, we replaced the first convolutional layer with one 3×3 layer, and the second convolutional layer with two 3×3 layers. This network is 26% faster than the standard NiN model, with only 54% of the model parameters. Using our low-rank filters in this network, we trained the **nin-c3-lr** network, which is of similar accuracy (91.8% *vs.* 91.9%) but is approximately 54% of the original network's computational complexity, with only 45% of the model parameters.

4.4.6 Comparing with ILSVRC State-of-the-Art Networks

Figures 4.10 and 4.11 compare published top-5 ILSVRC validation error *vs.* multiply-accumulate operations and number of model parameters (respectively) for several

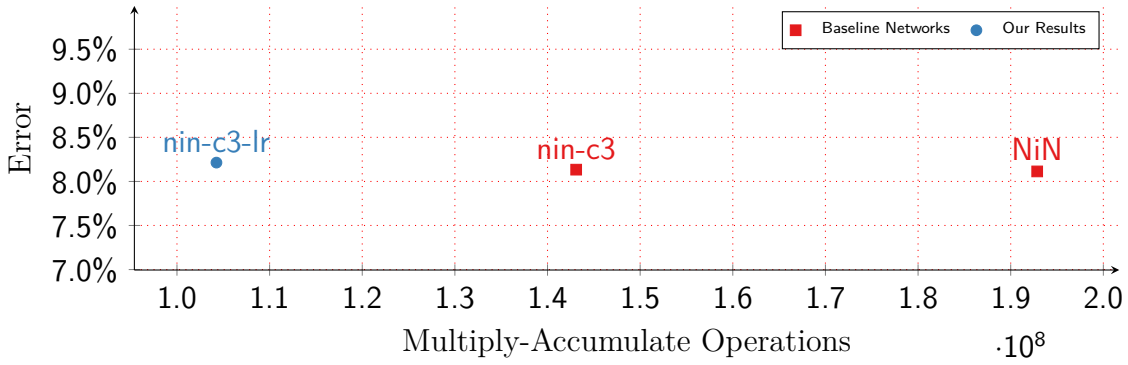


Fig. 4.9 **NiN CIFAR-10 Results.** Multiply-accumulate operations *vs.* error for NiN derived models on CIFAR-10 object classification dataset.

state-of-the-art networks (He *et al.*, 2015; Simonyan and Zisserman, 2015; Szegedy, Liu, *et al.*, 2015). The error rates for these networks are only reported as obtained with different combinations of computationally expensive training and test-time augmentation methods, including scale, photometric, ensembles (multi-model), and multi-view/dense oversampling. This can make it difficult to compare model architectures, especially with respect to computational requirements.

State-of-the-art networks, such as MSRA-C[2], VGG-19 and oversampled GoogLeNet are orders of magnitude larger in computational complexity than our networks. From fig. 4.10, where the multiply-accumulate operations are plotted on a log scale, increasing the model size and/or computational complexity of test-time augmentation of CNNs appears to have diminishing returns for decreasing validation error. Our models *without* training or test-time augmentation show comparable accuracy to networks such as VGG-13 *with* training and test-time augmentation, while having far less computational complexity and a smaller model size. In particular, the 'googlenet-lr' model has a much smaller test-time model size than any network of comparable accuracy.

4.5 Discussion

This chapter has presented a method to train CNN from scratch using low-rank filters. This is made possible by a new way of initializing the network's weights which takes into consideration the presence of differently shaped filters in composite layers. Validation on image classification in three popular datasets confirms similar or higher accuracy than the state-of-the-art models, with much greater computational efficiency.

[2]at the time of these experiments.

Table 4.6 **State-of-the-Art Single Models with Extra Augmentation.** Top-5 ILSVRC validation accuracy, single view and augmented test-time FLOPS (multiply-accumulate) count, and number of parameters for various state-of-the-art models *with* various training and test-time augmentation methods. A multi-model ensemble of MSRA-C is the current state-of-the-art network.

Real Name	FLOPS $\times 10^9$	FLOPS w/ Aug. $\times 10^9$	Param. $\times 10^7$	Top-5 Acc.
MSRA-C	53.46	107.17	33.06	0.943
MSRA-B	23.22	46.54	18.33	0.937
MSRA-A	19.06	38.20	17.80	0.935
VGG-E	19.63	39.30	14.37	0.910
VGG-D	15.47	30.97	13.84	0.912
VGG-C	11.77	23.57	13.36	0.906
VGG-B	11.31	22.64	13.30	0.901
VGG-A	7.61	15.24	13.29	0.895
GoogLeNet	1.59	15.91	1.34	0.909
GoogLeNet	1.59	229.11	1.34	0.921
ResNet-50	3.86	3.86	2.55	0.916

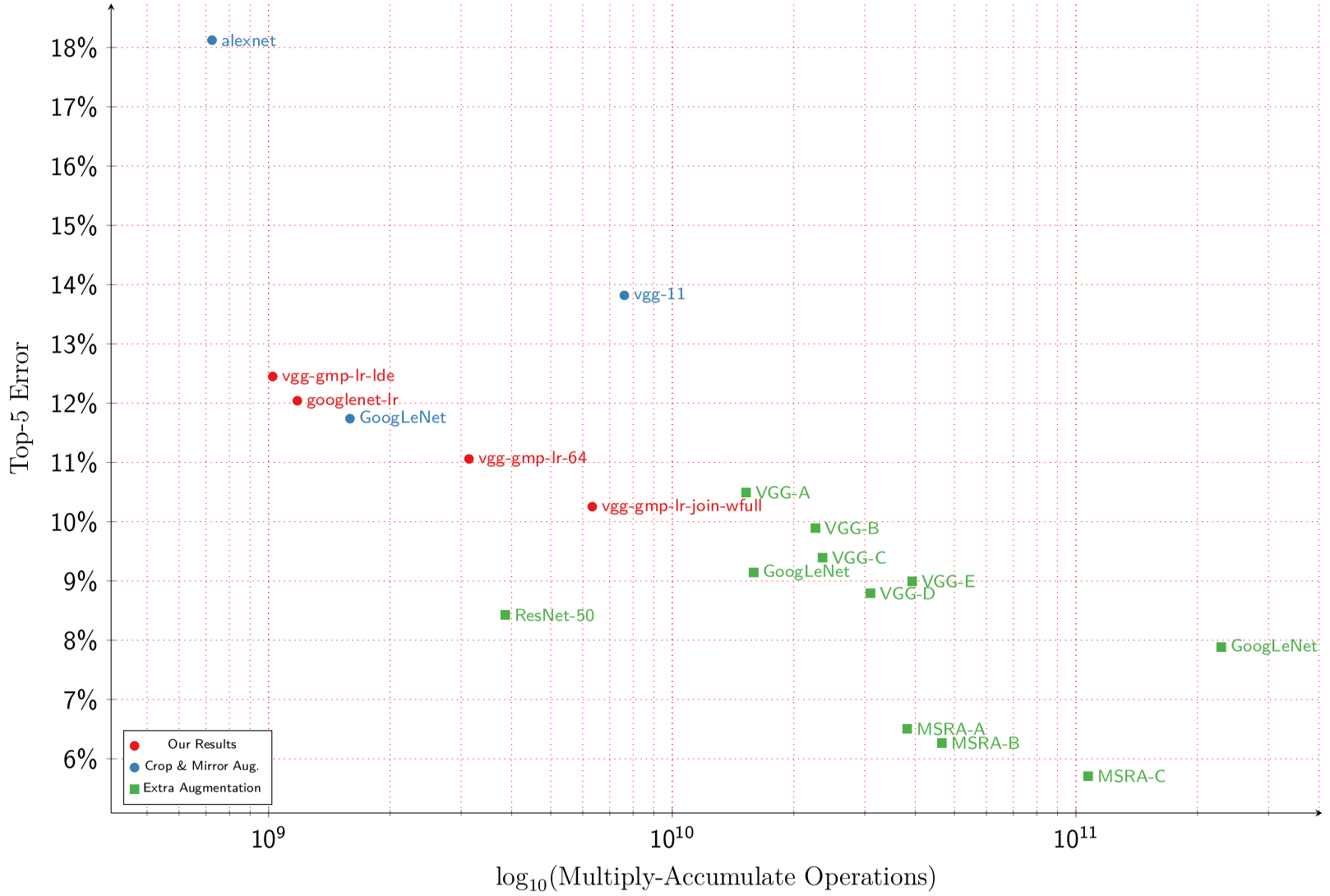


Fig. 4.10 **Computational complexity of state-of-the-art ILSVRC models.** Test-time multiply-accumulate operations *vs.* top-5 error on state-of-the-art networks using a *single* model. Note the difference in accuracy and computational complexity for VGG-11 model with/without extra augmentation. Our 'vgg-gmp-lr-join-wfull' model *without* extra augmentation is more accurate than VGG-11 *with* extra augmentation, and is much less computationally complex.

Fig. 4.11 **Number of Parameters of State-of-the-Art ILSVRC Models.** Test-time parameters *vs.* top-5 error for state-of-the-art models. The main factor in reduced model size is the use of global pooling or lack of fully-connected layers. Note that our 'googlenet-lr' model is almost an order of magnitude smaller than any other network of comparable accuracy.

It is somewhat surprising that networks based on learning filters with less representational ability are able to do as well, or better, than CNNs with full $k \times k$ filters on the task of image classification. However, a lot of interesting small-scale image structure is well-characterized by low-rank filters, *e.g.* edges and gradients. Our experiments training a separable (rank-1) model ('vgg-gmp-sf') on ILSVRC and MIT Places show surprisingly high accuracy on what are considered challenging problems — approx. 88% top-5 accuracy on ILSVRC — but not enough to obtain comparable accuracies to the models on which they are based.

Given that most discriminative filters learned for image classification appear to be low-rank, we instead structure our architectures with a set of basis filters in the way illustrated in fig. 4.2(d). This allows our networks to learn the most effective combinations of complex (*e.g.* $k \times k$) and simple (*e.g.* $1 \times k$, $k \times 1$) filters. Furthermore, in restricting how many complex spatial filters may be learned, this architecture prevents overfitting, and helps improve generalization. Even in our models where we do not use square $k \times k$ filters, we obtain comparable accuracies to the baseline model, since the rank-2 cross-shaped filters effectively learned as a combination of 3×1 and 1×3 filters are capable of representing more complex local pixel relations than rank-1 filters.

Recent advances in state-of-the-art accuracy with CNNs for image classification have come at the cost of increasingly large and computationally complex models. We believe our results to show that learning computationally efficient models with fewer, more relevant parameters, can prevent overfitting, increase generalization and thus also increase accuracy.

5

Inter-Filter Connectivity

"The marvelous powers of the brain emerge not from any single, uniformly structured connectionist network but from highly evolved arrangements of smaller, specialized networks which are interconnected in very specific ways."

– Marvin Minsky, *Prologue: A View from 1988, Perceptrons*

With few exceptions, state-of-the-art CNNs for image recognition are largely monolithic, with each filter operating on the feature maps of all filters on a previous layer. Interestingly, this is in stark contrast to what we understand of biological neural networks, where we see "highly evolved arrangements of smaller, specialized networks which are interconnected in very specific ways" (Minsky and Papert, 1988).

Yet it has been shown that a large proportion of the learned weights in DNNs are redundant (Denil *et al.*, 2013), a property that has been widely exploited to make neural networks smaller and more computationally efficient (Denton *et al.*, 2014; Szegedy, Liu, *et al.*, 2015). A carefully designed sparse network connection structure can have a regularizing effect. CNNs (Fukushima, 1980; LeCun, Bottou, *et al.*, 1998) embody this idea, using a sparse convolutional connection structure to exploit the locality of natural image structure. In consequence, they are easier to train.

In chapter 4 learning a low-rank spatial basis for filters was found to improve generalization while reducing the computational complexity and model size of a CNN with only full rank filters. However, this work addressed only the spatial extents of the convolutional filters (*i.e.* h and w in fig. 5.1(a)). In this work we will show that a similar idea can be applied to the channel extents — *i.e.* filter inter-connectivity — by using *filter groups* (Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012).

In this chapter we show that simple alterations to the architecture of state-of-the-art CNNs for image recognition can drastically reduce computational cost and model

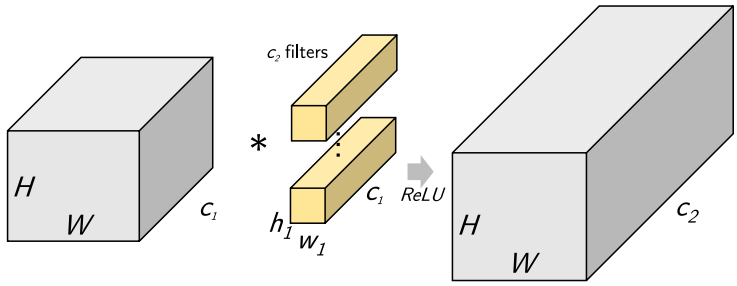
size while maintaining (or even increasing) accuracy, through a novel structural prior reducing the connectivity in monolithic networks to reflect more closely the sparse, localized filter co-dependencies within a trained network.

5.1 Related Work

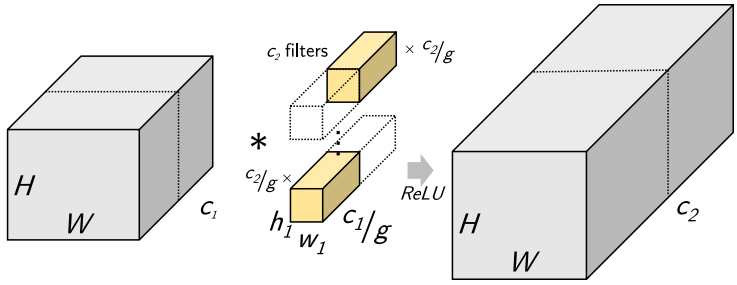
Most previous work on reducing the computational complexity of CNNs has focused on approximating convolutional filters in the spatial (as opposed to the channel) domain, either by using low-rank approximations (Jaderberg, Vedaldi, and Zisserman, 2014; Lebedev *et al.*, 2015; Mamalet and Garcia, 2012; Rigamonti *et al.*, 2013), or Fourier transform based convolution (Mathieu, Henaff, and LeCun, 2014; Rippel, Snoek, and Adams, 2015). More general methods have used reduced precision number representations (Gupta *et al.*, 2015) or compression of previously trained models (W. Chen *et al.*, 2015; Kim *et al.*, 2016). Here we explore methods that reduce the computational impact of the large number of filter channels within state-of-the art networks. Specifically, we consider decreasing the number of incoming connections to neurons.

AlexNet Filter groups Amongst the seminal contributions made by Krizhevsky, Sutskever, and Geoffrey E. Hinton (2012) is the use of 'filter groups' in the convolutional layers of a CNN (see fig. 5.1). While their use of filter groups was necessitated by the practical need to sub-divide the work of training a large network across multiple GPUs, the side effects are somewhat surprising. Specifically, the authors observe that independent filter groups learn a separation of responsibility (colour features *vs.* texture features) that is consistent over different random initializations. Also surprising, and not explicitly stated by Krizhevsky, Sutskever, and Geoffrey E. Hinton (2012), is the fact that the AlexNet network has approximately 57% fewer connection weights than the corresponding network without filter groups. This is due to the reduction in the input channel dimension of the grouped convolution filters (see fig. 5.2). Despite the large difference in the number of parameters between the models, both achieve comparable accuracy on ILSVRC — in fact the smaller grouped network gets $\approx 1\%$ lower top-5 validation error. This paper builds upon these findings and extends them to state-of-the-art networks.

Low-dimensional Embeddings Lin, Q. Chen, and Yan (2014) proposed a method to reduce the dimensionality of convolutional feature maps. By using relatively cheap



(a) Convolution



(b) Convolution with filter groups

Fig. 5.1 **Convolutional Filter groups.** (a) Convolutional filters (yellow) typically have the same channel dimension c_1 as the input feature maps (gray) on which they operate. However, (b) with filter grouping, g independent groups of c_2/g filters operate on a fraction c_1/g of the input feature map channels, reducing filter dimensions from $h \times w \times c_1$ to $h \times w \times c_1/g$. This change does not affect the dimensions of the input and output feature maps but significantly reduces computational complexity and the number of model parameters.

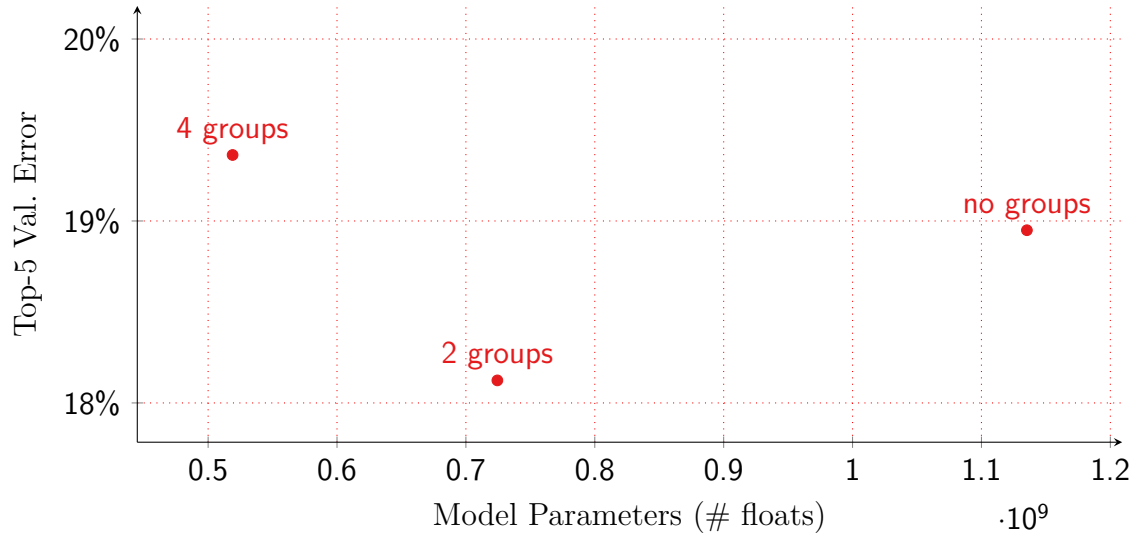


Fig. 5.2 **AlexNet Performance and Filter Groups.** Model Parameters *vs.* top-5 error for variants of the AlexNet model on ILSVRC image classification dataset. Models with moderate numbers of filter groups have far fewer parameters, yet surprisingly maintain comparable error.

'1×1' convolutional layers (*i.e.* layers comprising d filters of size $1\times1\times c$, where $d < c$), they learn to map feature maps into lower-dimensional spaces, *i.e.* to new feature maps with fewer channels. Subsequent spatial filters operating on this lower dimensional input space require significantly less computation. This method is used in most state-of-the-art networks for image classification to reduce computation (He *et al.*, 2016a; Szegedy, Liu, *et al.*, 2015). Our method is complementary.

GoogLeNet In contrast to much other work, Szegedy, Liu, *et al.* (2015) propose a CNN architecture that is highly optimized for computational efficiency. GoogLeNet uses, as a basic building block, a mixture of low-dimensional embeddings (Lin, Q. Chen, and Yan, 2014) and heterogeneously-sized spatial filters — collectively an Inception module. There are two distinct forms of convolutional layers in the Inception module, low-dimensional embeddings (1×1) and spatial (3×3, 5×5). GoogLeNet keeps large, expensive spatial convolutions (*i.e.* 5×5) to a minimum by using few of these filters, using more 3×3 convolutions, and even more 1×1 filters. The motivation is that most of the convolutional filters respond to localized patterns in a small receptive field, with few requiring a larger receptive field. The number of filters in each successive Inception module increases slowly with decreasing feature map size, in order to maintain computational performance. GoogLeNet is by far the most efficient state-of-the-art network for ILSVRC, achieving near state-of-the-art accuracy with the

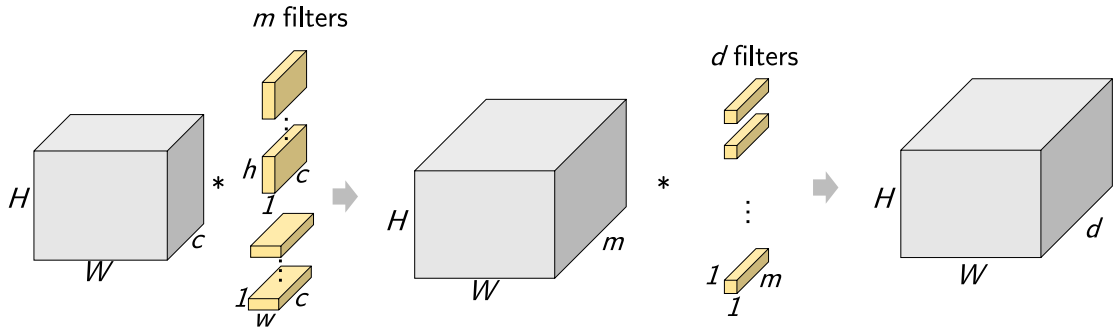


Fig. 5.3 **Learning a spatial basis for filters.** Learning a linear combination of mostly small, heterogeneously-sized spatial filters, as proposed in chapter 4. Note that all filters operate on all c channels of the input feature map.

lowest computation/model size. However, we will show that even such an efficient and optimized network architecture benefits from our method.

Low-Rank Approximations Various authors have suggested approximating learned convolutional filters using tensor decomposition (Jaderberg, Vedaldi, and Zisserman, 2014; Kim *et al.*, 2016; Lebedev *et al.*, 2015). For example, Jaderberg, Vedaldi, and Zisserman (2014) propose approximating the convolutional filters in a trained network with representations that are low-rank both in the spatial and the channel domains. This approach significantly decreases computational complexity, albeit at the expense of a small amount of accuracy. In this chapter we are not approximating an existing model's weights but creating a new network architecture with explicit structural sparsity, which is then trained from scratch.

Learning a Basis for Filters Our approach is connected with that presented in chapter 4, where we showed that replacing 3×3×c filters with linear combinations of filters with smaller spatial extent (*e.g.* 1×3×c, 3×1×c filters, see fig. 5.3) could reduce the model size and computational complexity of state-of-the-art CNNs, while maintaining or even increasing accuracy. However, that work did not address the channel extent of the filters.

5.2 Root Architectures

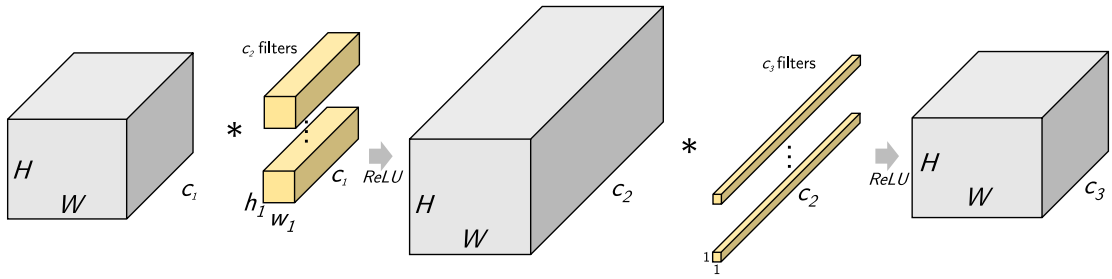
In this section we present the main contribution of our work: the use of novel sparsely-connected architectures resembling tree roots — to decrease computational complexity and model size compared to state-of-the-art DNNs for image recognition.

Learning a Basis for Filter Dependencies It is unlikely that every filter (or neuron) in a deep neural network needs to depend on the output of all the filters in the previous layer. In fact, reducing filter co-dependence in DNNs has been shown to benefit generalization. For example, Geoffrey E. Hinton, Srivastava, *et al.* (2012b) introduced *dropout* for regularization of DNNs. When training a network layer with dropout, a random subset of neurons is excluded from both the forward and backward pass for each mini-batch. Furthermore, Cogswell *et al.* (2016) observe a correlation between the covariance of hidden unit activations and overfitting. To explicitly reduce the covariance of hidden activations, they train networks with a loss function, based on the covariance matrix of the activations in a hidden layer.

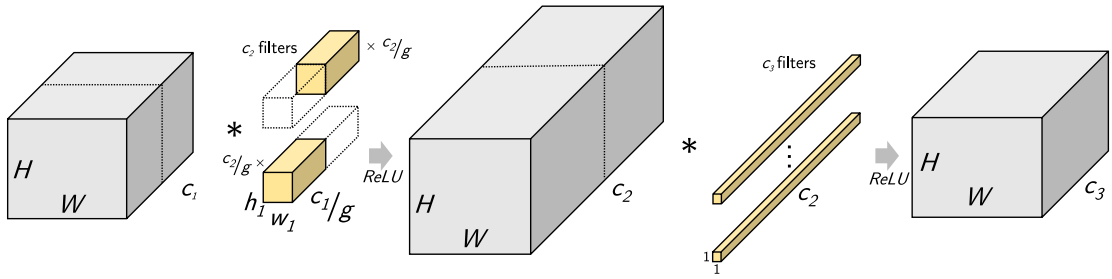
Instead of using a modified loss, regularization penalty, or randomized network connectivity during training to prevent co-adaption of features, we take a much more direct approach. We use filter groups (see fig. 5.1) to force the network to learn filters with only limited dependence on previous layers. Each of the filters in the filter groups is smaller in the channel extent, since it operates on only a subset of the channels of the input feature map.

This reduced connectivity also reduces computational complexity and model size since the size of filters in filter groups are reduced drastically, as is evident in fig. 5.4. Unlike methods for increasing the efficiency of DNNs by approximating pre-trained existing networks (see section 5.1), our models are trained from random initialization using stochastic gradient descent. This means that our method can also speed up training and, since we are not merely approximating an existing model's weights, the accuracy of the existing model is not an upper bound on accuracy of the modified model.

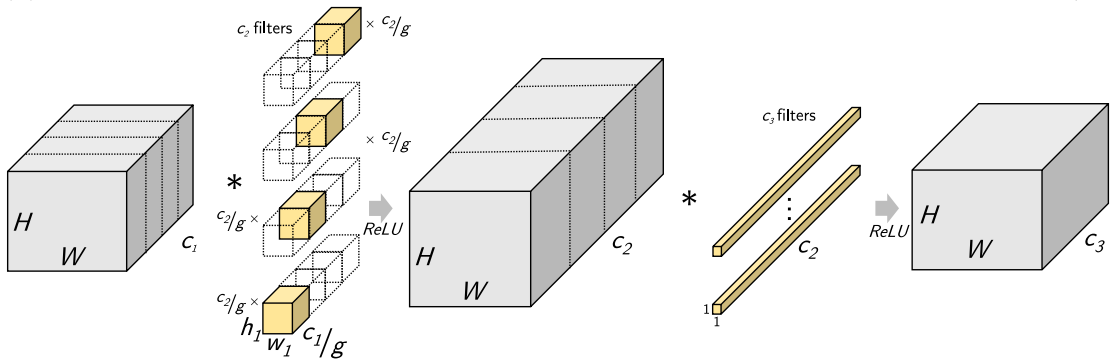
Root Module The basic element of our network architecture, a *root module*, is shown in fig. 5.4. A root module has a given number of filter groups, the more filter groups, the fewer the number of connections to the previous layer's outputs. Each spatial convolutional layer is followed by a low-dimensional embedding (1×1 convolution). Like in chapter 4, this configuration learns a linear combination of the basis filters (filter groups), implicitly representing a filter of full channel depth, but with limited filter dependence.



(a) Convolution with d filters of shape $h \times w \times c$.



(b) Root-2 Module: Convolution with d filters in $g = 2$ filter groups, of shape $h \times w \times c/2$.



(c) Root-4 Module: Convolution with d filters in $g = 4$ filter groups, of shape $h \times w \times c/4$.

Fig. 5.4 **Root Modules.** Root modules (b), (c) compared to a typical set of convolutional layers (a) found in ResNet and other modern architectures. Grey blocks represent the feature maps over which a layer's filters operate, while colored blocks represent the filters of each layer.

Table 5.1 **NiN Root Architectures**. Filter groups in each convolutional layer.

Model	conv1			conv2			conv3		
	a	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*
	5×5	*1×1*	*1×1*	*5×5*	*1×1*	*1×1*	*3×3*	*1×1*	*1×1*
Orig.	1	1	1	1	1	1	1	1	1
root-2	1	1	1	2	1	1	1	1	1
root-4	1	1	1	4	1	1	2	1	1
root-8	1	1	1	8	1	1	4	1	1
root-16	1	1	1	16	1	1	8	1	1

5.3 Results

Here we present image classification results obtained by replacing spatial convolutional layers within existing state-of-the-art network architectures with root modules (described in section 5.2) .

5.3.1 Improving NiN on CIFAR-10

NiN (Lin, Q. Chen, and Yan, 2014) is a near state-of-the-art network for CIFAR-10 (Krizhevsky, 2009). It is composed of 3 spatial (5×5, 3×3) convolutional layers with a large number of filters (192), interspersed with pairs of low-dimensional embedding (1×1) layers. As a baseline, we replicated the standard NiN network architecture as described by Lin, Q. Chen, and Yan (2014) but used state-of-the-art training methods. We trained using random 32×32 cropped and mirrored images from 4-pixel zero-padded mean-subtracted images, as used by I. J. Goodfellow *et al.* (2013) and He *et al.* (2016a). We also used the initialization of He *et al.* (2015) and batch normalization (Ioffe and Szegedy, 2015). With this configuration, ZCA whitening was not required to reproduce validation accuracies obtained in (Lin, Q. Chen, and Yan, 2014). We also did not use dropout, having found it to have little effect, presumably due to our use of batch normalization, as suggested by Ioffe and Szegedy (2015).

To assess the efficacy of our method, we replaced the spatial convolutional layers of the original NiN network with root modules (as described in section 5.2). We preserved the original number of filters per layer but subdivided them into groups as shown in table 5.1. We considered the first of the pair of existing 1×1 layers to be part of our root modules. We did not group filters in the first convolutional layer — since it operates on the three-channel image space, it is of limited computational impact compared to other layers. Results are shown in table 5.2 and fig. 5.5 for various network

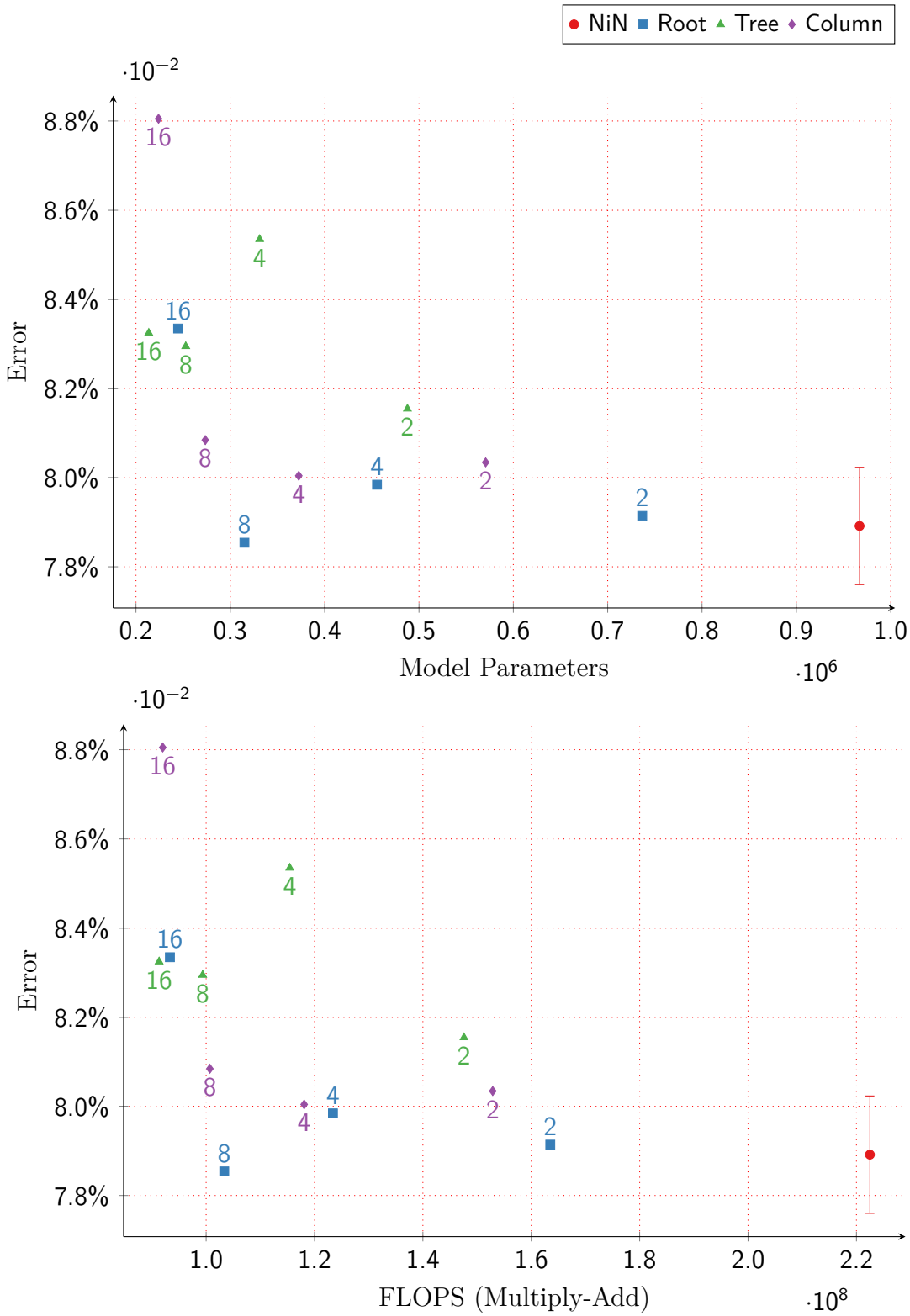


Fig. 5.5 **NiN CIFAR-10 Results.** Spatial filters (3×3, 5×5) are grouped hierarchically. The best models are closest to the origin. For the standard network, the mean and standard deviation (error bars) are shown over 5 different random initializations.

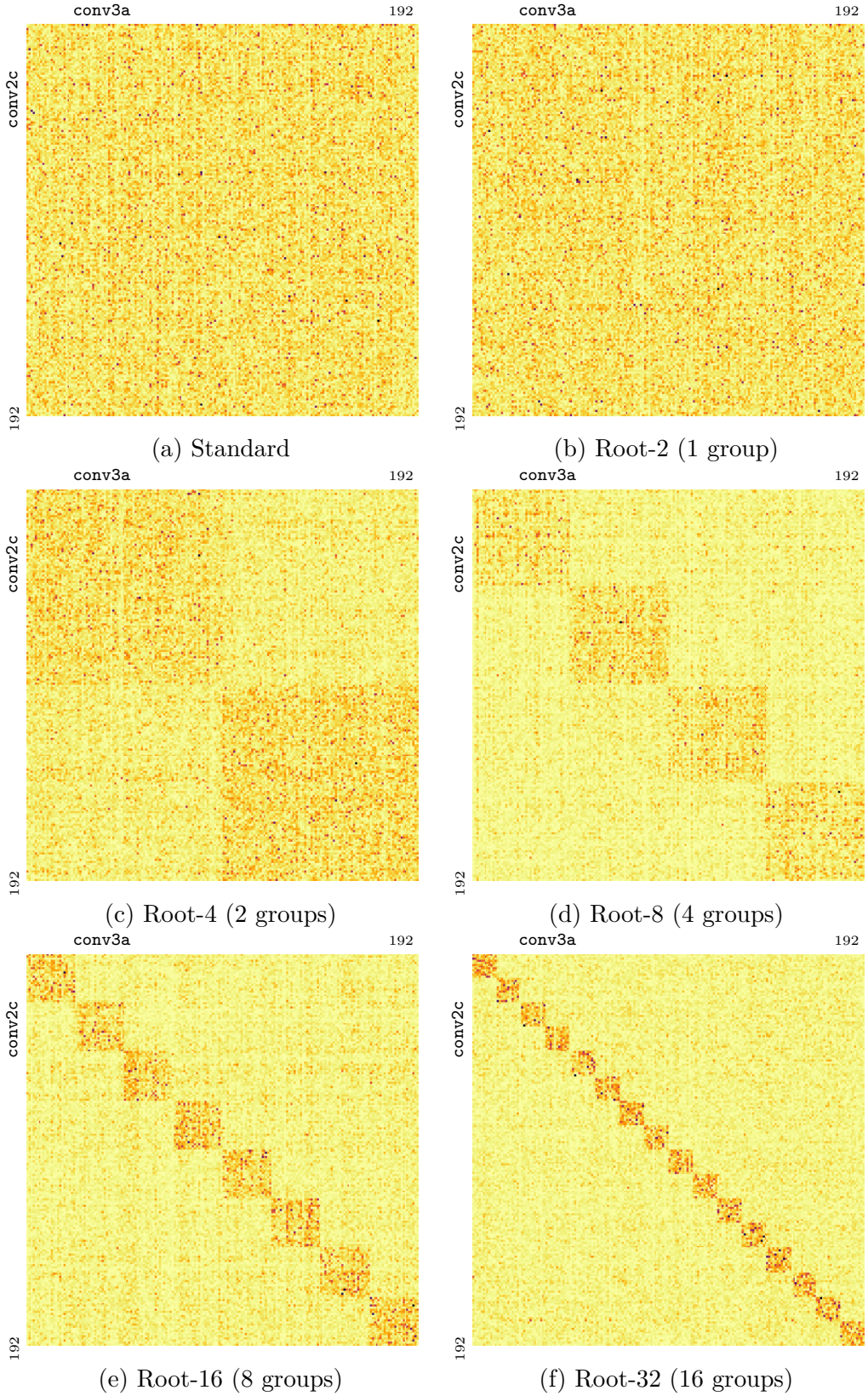
Table 5.2 **NiN CIFAR-10 Results**

Model	FLOPS $\times 10^8$	Param. $\times 10^5$	Accuracy	CPU (ms)	GPU (ms)
Orig.	2.22	9.67	0.9211	39.0	0.623
root-2	1.64	7.37	0.9209	31.2	0.551
root-4	1.23	4.55	0.9202	27.6	0.480
root-8	1.03	3.15	0.9215	24.4	0.482
root-16	0.93	2.45	0.9167	23.0	0.475
tree-2	1.48	4.88	0.9185	31.4	0.541
tree-4	1.15	3.31	0.9147	29.1	0.535
tree-8	0.99	2.53	0.9171	25.7	0.500
tree-16	0.91	2.14	0.9168	20.6	0.512
col-2	1.53	5.71	0.9197	28.8	0.568
col-4	1.18	3.73	0.9200	26.1	0.536
col-8	1.01	2.73	0.9192	23.0	0.475
col-16	0.92	2.24	0.9120	22.8	0.494

architectures[1]. Compared to the baseline architecture, the root variants achieve a significant reduction in computation and model size without a significant reduction in accuracy. For example, the root-8 architecture gives equivalent accuracy with only 46% of the FLOPS, 33% of the model parameters of the original network, and approximately 37% and 23% faster CPU and GPU timings (see section 5.4 for an explanation of the GPU timing disparity).

Figure 5.6 shows the inter-layer covariance between the adjacent filter layers `conv2c` and `conv3a` in the network architectures outlined in table 5.1 as evaluated on the CIFAR training set. The block-diagonalization enforced by the filter group structure (as illustrated in fig. 5.1) is visible, more so with larger number of filter groups. This shows that the network learns an organization of filters such that the sparsely distributed strong filter relations, visible in fig. 5.6(a) as brighter pixels, are grouped into a denser block-diagonal structure, leaving a visibly darker, low-correlated background.

[1]Here (and subsequently unless stated otherwise) timings are per image for a forward pass computed on a large batch. Networks were implemented using Caffe (with CuDNN v2 and MKL) and run on an Nvidia Titan Z GPU and 2 10-core Intel Xeon E5-2680 v2 CPUs.

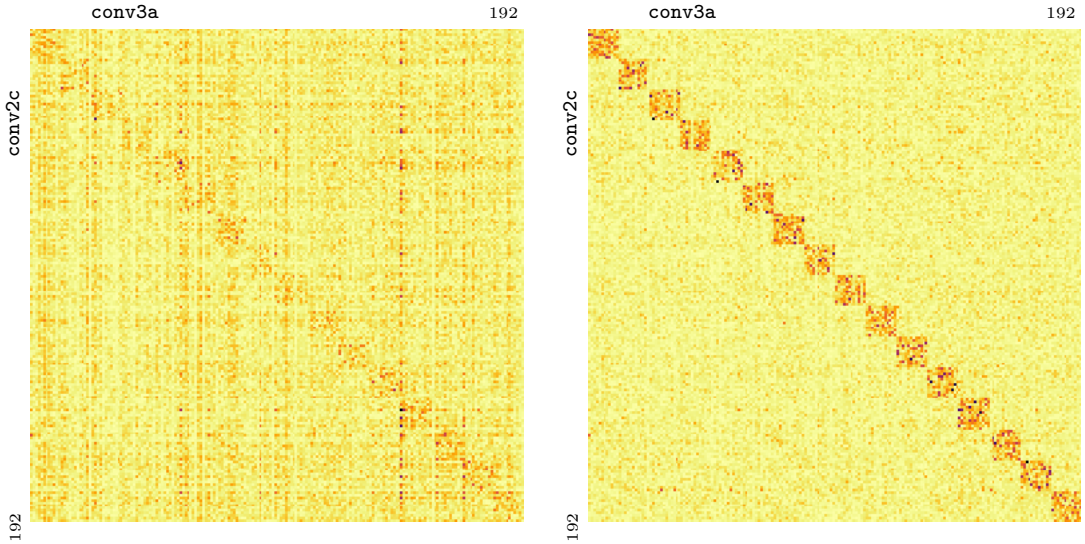


(a) Standard (b) Root-2 (1 group)

(c) Root-4 (2 groups) (d) Root-8 (4 groups)

(e) Root-16 (8 groups) (f) Root-32 (16 groups)

Fig. 5.6 **Inter-layer filter covariance conv2c–conv3a.** The block-diagonal sparsity learned by a root-unit is visible in the correlation of filters on layers `conv3a` and `conv2c` in the NiN network as observed on the CIFAR-10 training data.



(a) Non-whitened responses (b) Whitened responses

Fig. 5.7 Covariance for between two layers in the root-32 NiN model with and without whitened responses.

5.3.2 Inter-Layer Covariance

To show the relationships between filters between adjacent convolutional layers, as illustrated in fig. 5.1(a), we calculate the covariance of the responses from two adjacent feature maps, the outputs of convolutional layers with c_1 and c_2 filters.

Let $X_i = [\mathbf{x}_{i,1}; \mathbf{x}_{i,2}; \ldots; \mathbf{x}_{i,N}]$ be the matrix of N samples $\mathbf{x}_{i,n}$ from the c_i dimensional feature map for layer i. We consider each pixel across the two feature maps to be a sample, and thus each vector $\mathbf{x}_{i,n}$ is a single pixel filter response of dimension c_i. If two feature maps have different spatial dimensions, due to pooling, we up-sample the smaller feature map (with nearest neighbor interpolation) such that there are the same number of pixels (and thus samples) in each feature map.

Given two samples X_1, X_2 with zero mean (*i.e.* mean subtracted) for two adjacent feature maps, we calculate the inter-layer covariance,

$$\operatorname{covar}(X_1, X_2) = \mathrm{E}\left[X_1\, X_2^{\mathrm{T}}\right], \tag{5.1}$$

$$= \frac{1}{N-1} X_1\, X_2^{\mathrm{T}}. \tag{5.2}$$

While this shows the covariance between layers, it is conflated with the inherent covariances within X_1 and X_2 from the data (as shown in fig. 5.7(a)). We can more

clearly show the covariance between layers by first whitening (using ZCA (Krizhevsky, 2009)) the samples in X_1 and X_2. For a covariance matrix,

$$\text{covar}(X, X) = \frac{1}{N-1} X X^{\mathrm{T}}, \tag{5.3}$$

The ZCA whitening transformation is given by,

$$W = \sqrt{N-1} \left(X X^{\mathrm{T}} \right)^{-\frac{1}{2}}. \tag{5.4}$$

Since the covariance matrix is symmetric, it is easily diagonalizable (*i.e.* PCA),

$$\text{covar}(X, X) = \frac{1}{N-1} P D P^{\mathrm{T}}, \tag{5.5}$$

$$\tag{5.6}$$

where P is a orthogonal matrix and D a diagonal matrix. This diagonalization allows a simplified calculation of the whitening transformation (see the derivation in Krizhevsky (2009, Appendix A)),

$$W = \sqrt{N-1} P D^{\circ -\frac{1}{2}} P^{\mathrm{T}}, \tag{5.7}$$

where $D^{\circ -\frac{1}{2}}$ is simply D with an element-wise power of $-\frac{1}{2}$.

The covariance between the whitened feature map responses is then,

$$\text{covar}(W_1 X_1, W_2 X_2) = E\left[(W_1 X_1) (W_2 X_2)^{\mathrm{T}} \right]. \tag{5.8}$$

Figure 5.8 shows the per-layer (intra-layer) filter correlation. This shows the correlation of filters is more structured in root-networks, filters are learned to be linearly combined into useful filters by the root module, and thus filters are often grouped together with other filters with which they correlate strongly.

Figure 5.6 shows the inter-layer filter covariances between layers `conv3a` and `conv2c`. Figure 5.10 shows the full set of inter-layer covariances between all convolutional layers in the NiN models. Block-diagonal sparsity is visible on the layers with filter groups, `conv2a` and `conv3a`. This block-diagonal is shown for all variants in more detail in fig. 5.10.

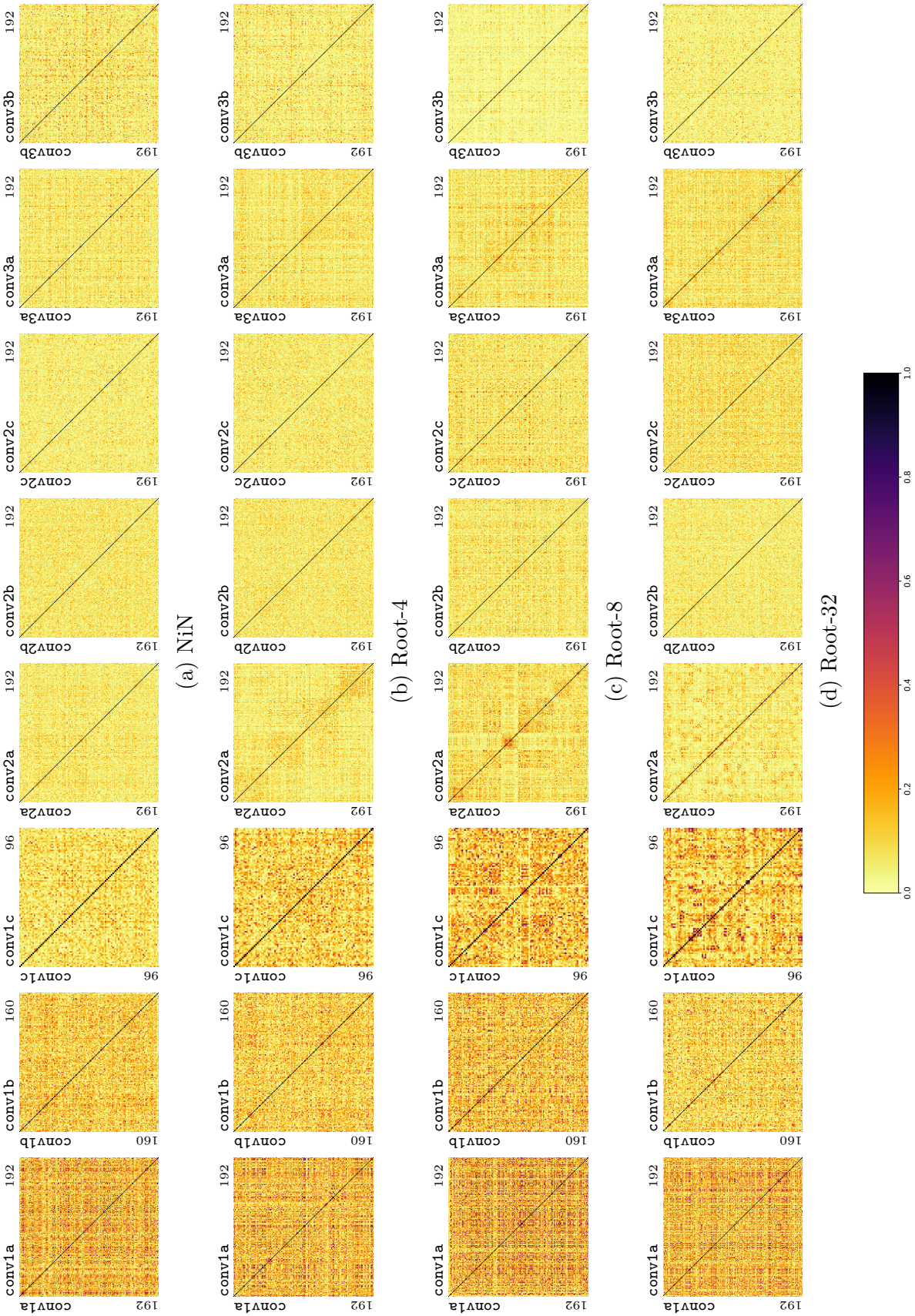


Fig. 5.8 **NiN Intra-Layer Correlation (Train).** Absolute correlation of filters within each layer of a NiN model variant on the training data.

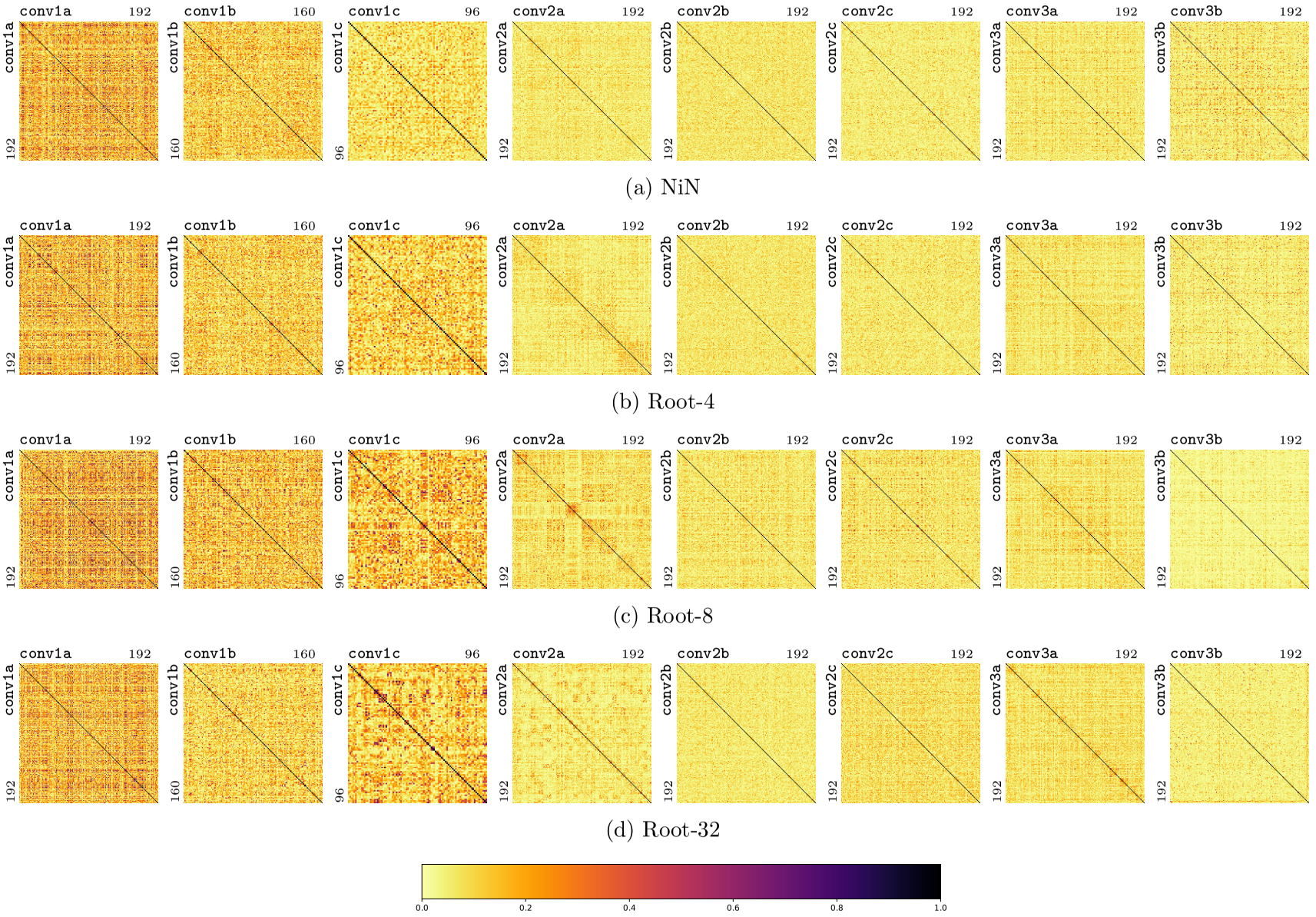


Fig. 5.9 **NiN Intra-Layer Correlation (Test).** Absolute correlation of filters within each layer of a NiN model variant on the test data.

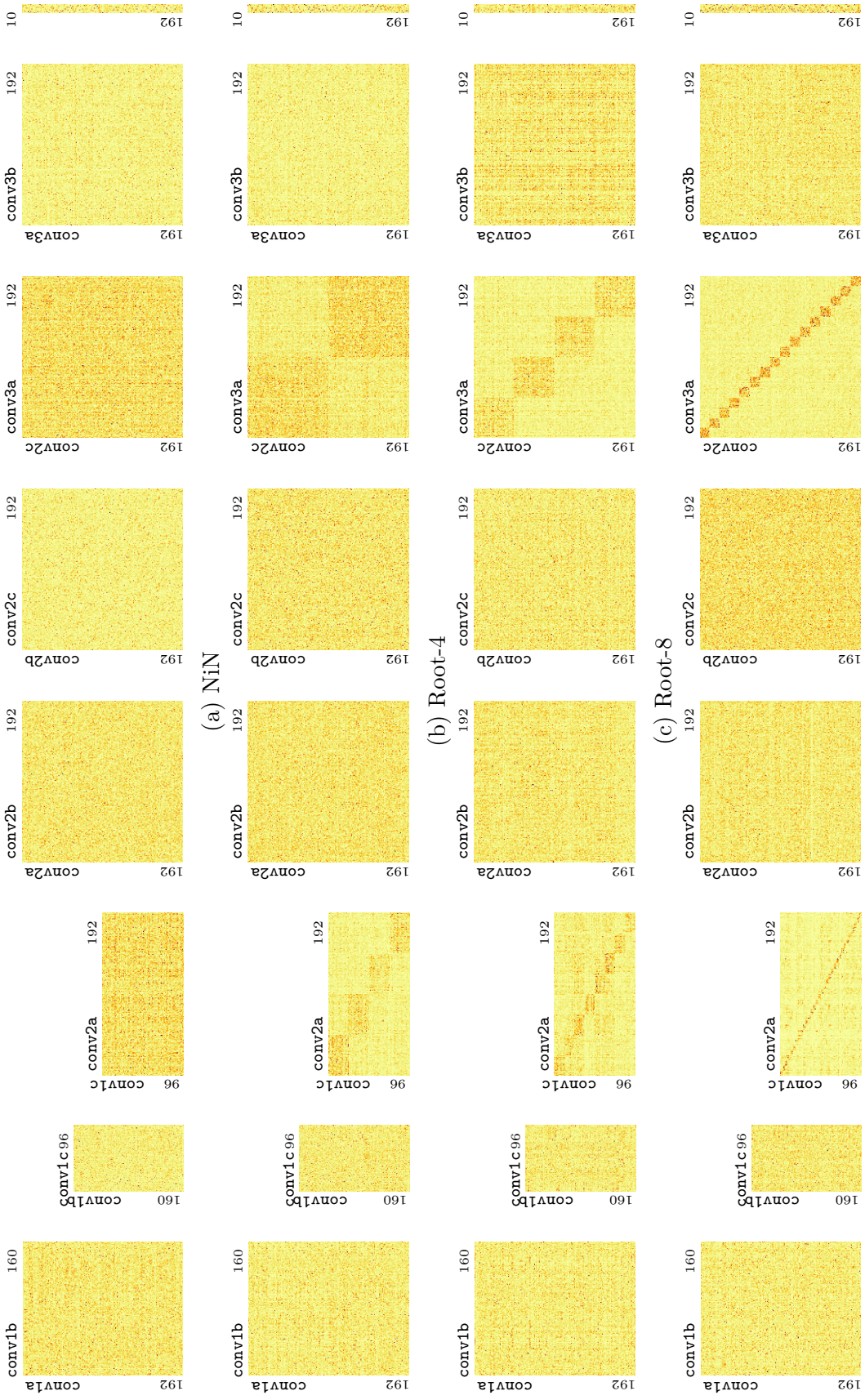


Fig. 5.10 **NiN Inter-layer Covariance (Train).** The inter-layer covariance for all layers in variants of the NiN network

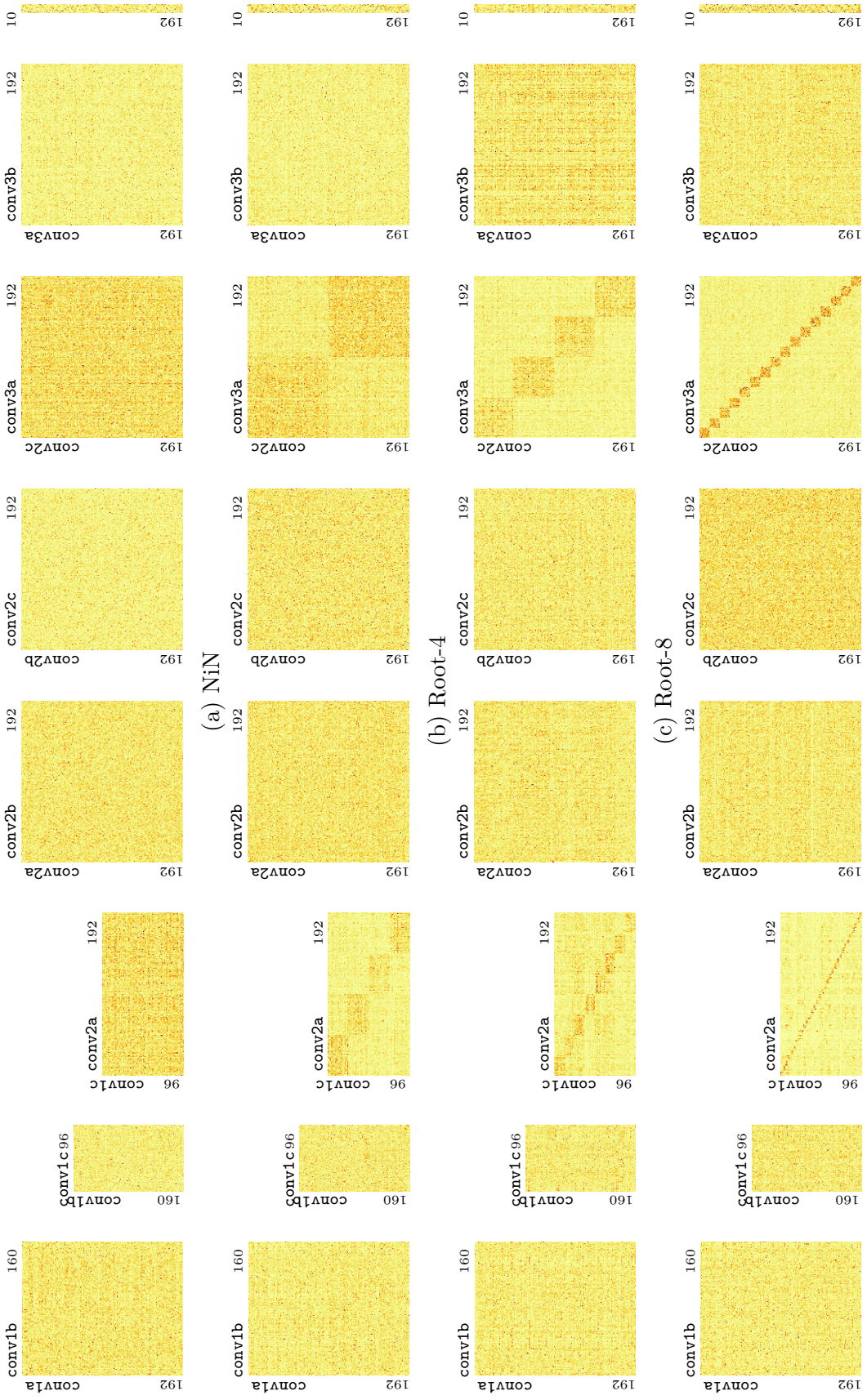
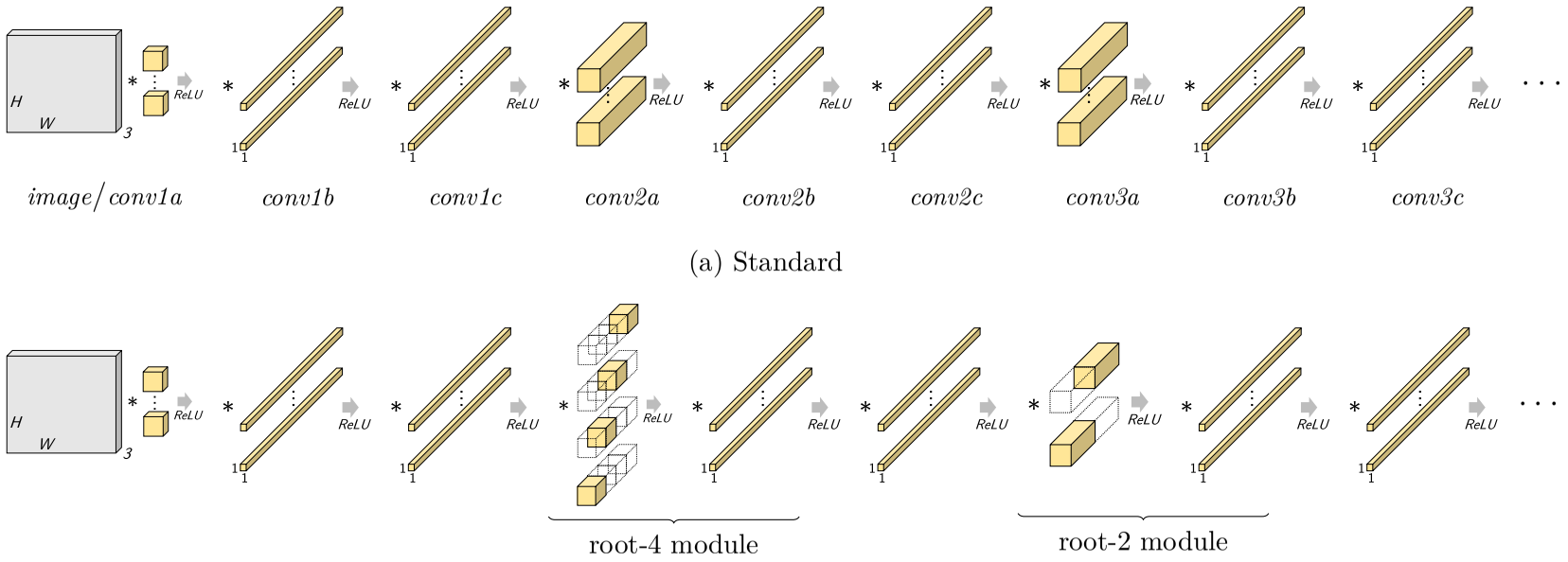


Fig. 5.11 **NiN Inter-layer Covariance (Test).** The inter-layer covariance for all layers in variants of the NiN network



(a) Standard

(b) Root-4 Architecture

Fig. 5.12 **NiN Root Architecture.** The Root-4 architecture as compared to the original architecture for all the convolutional layers. Colored blocks represent the filters of each layer. Here we don't show the intermediate feature maps over which a layer's filters operate, or the final fully-connected layer, out of space considerations (see figs. 2.13 and 5.4). The decreasing degree of grouping in successive root modules means that our network architectures somewhat resemble tree roots, hence the name root.

5.3.3 Grouping Degree with Network Depth

An interesting question concerns how the degree of grouping in our root modules should be varied as a function of depth in the network. For the NiN-like architectures described earlier, we might consider having the degree of grouping: (**i**) decrease with depth after the first convolutional layer, *e.g.* 1–8–4 ('root'); (**ii**) remain constant with depth after the first convolutional layer, *e.g.* 1–4–4 ('column'); or (**iii**) increase with depth, *e.g.* 1–4–8 ('tree').

To determine which approach is best, we created variants of the NiN architecture with different degrees of grouping per layer. Results are shown in fig. 5.5. The results show that the so-called root topology (illustrated in fig. 5.12) gives the best performance, providing the smallest reduction in accuracy for a given reduction in model size and computational complexity. Similar experiments with deeper network architectures have delivered similar results and so we have reported results for root topologies. This aligns with the intuition of DNNs for image recognition subsuming the deformable parts model. If we assume that filter responses identify parts (or more elemental features), then there should be more filter dependence with depth, as more parts (filter responses) are assembled into complex concepts.

5.3.4 Improving Residual Networks on ILSVRC

Residual networks (ResNets) (He *et al.*, 2016a) are the state-of-the-art network for ILSVRC. ResNets are more computationally efficient than the VGG architecture (Simonyan and Zisserman, 2015) on which they are based, due to the use of low-dimensional embeddings (Lin, Q. Chen, and Yan, 2014). ResNets are also more accurate and quicker to converge due to the use of identity mappings.

ResNet 50

As a baseline, we used the ResNet 50 model (He *et al.*, 2016a) (the largest residual network model to fit onto 8 GPUs with Caffe). ResNet 50 has 50 convolutional layers, of which one-third are spatial convolutions (non-1×1). We did not use any training augmentation aside from random cropping and mirroring. For training, we used the initialization scheme described by (He *et al.*, 2015) modified for compound layers, as presented in section 4.3, and batch normalization (Ioffe and Szegedy, 2015). To assess the efficacy of our method, we replaced the spatial convolutional layers of the original network with root modules (as described in section 5.2). We preserved the original number of filters per layer but subdivided them into groups as shown in table 5.3. We

Table 5.3 **ResNet 50.** Filter groups in each conv. layer.

Model	conv1	res2{a–c}		res3{a–d}		res4{a–f}		res5{a–c}	
	7×7	*1×1*	*3×3*	*1×1*	*3×3*	*1×1*	*3×3*	*1×1*	*3×3*
Orig.	1	1	1	1	1	1	1	1	1
root-2	1	1	2	1	1	1	1	1	1
root-4	1	1	4	1	2	1	1	1	1
root-8	1	1	8	1	4	1	2	1	1
root-16	1	1	16	1	8	1	4	1	2
root-32	1	1	32	1	16	1	8	1	4
root-64	1	1	64	1	32	1	16	1	8

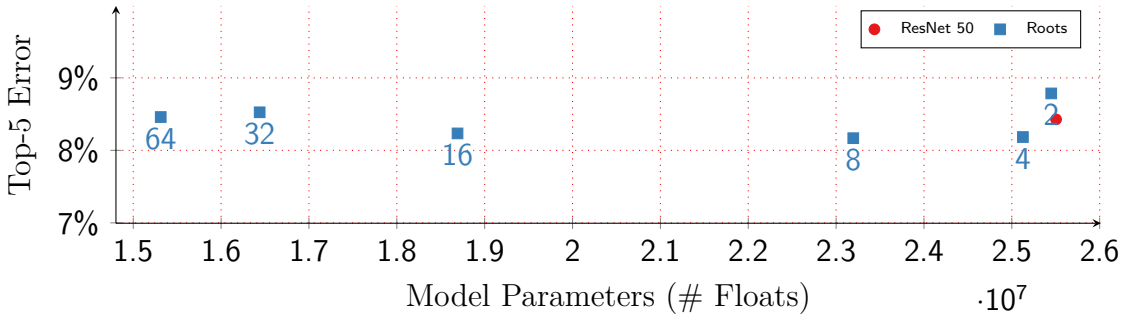
Table 5.4 **ResNet 50 Results.**

Model	FLOPS $\times 10^9$	Param. $\times 10^7$	Top-1 Acc.	Top-5 Acc.	CPU (ms)	GPU (ms)
Orig.	3.86	2.55	0.730	0.916	621	11.6
root-2	3.68	2.54	0.727	0.912	520	11.1
root-4	3.37	2.51	0.734	0.918	566	11.3
root-8	2.86	2.32	0.734	0.918	519	10.7
root-16	2.43	1.87	0.732	0.918	479	10.1
root-32	2.22	1.64	0.729	0.915	469	10.1
root-64	2.11	1.53	0.732	0.915	426	10.2

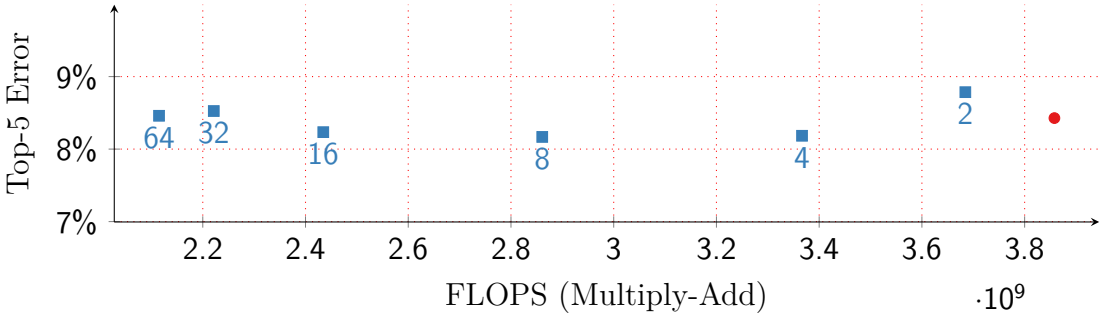
considered the first of the existing 1×1 layers subsequent to each spatial convolution to be part of our root modules. Results are shown in table 5.4 and fig. 5.13 for various network architectures. Compared to the baseline architecture, the root variants achieve a significant reduction in computation and model size without a significant reduction in accuracy. For example, the best result (root-16) exceeds the baseline accuracy by 0.2% while reducing the model size by 27% and floating-point operations (multiply-add) by 37%. CPU timings were 23% faster, while GPU timings were 13% faster. With a drop in accuracy of only 0.1% however, the root-64 model reduces the model size by 40%, and reduces the floating point operations by 45%. CPU timings were 31% faster, while GPU timings were 12% faster.

ResNet 200

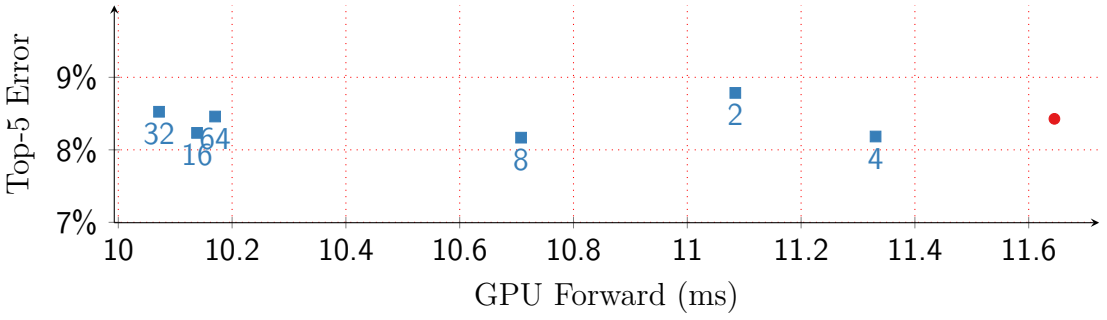
To show that the method applies to deeper architectures, we also applied our method to ResNet 200, the deepest network for ILSVRC 2012. To provide a baseline we used code



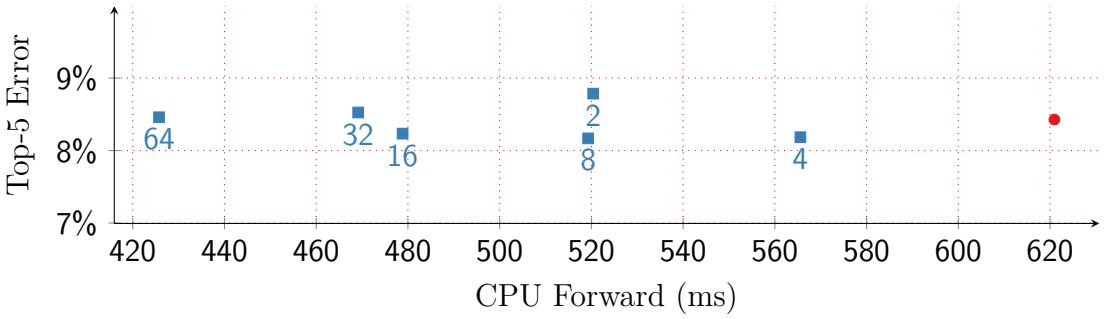
(a) Model Parameters *vs.* Top-5 Error



(b) FLOPS (Multiply-Add) *vs.* Top-5 Error



(c) GPU Forward Time *vs.* Top-5 Error

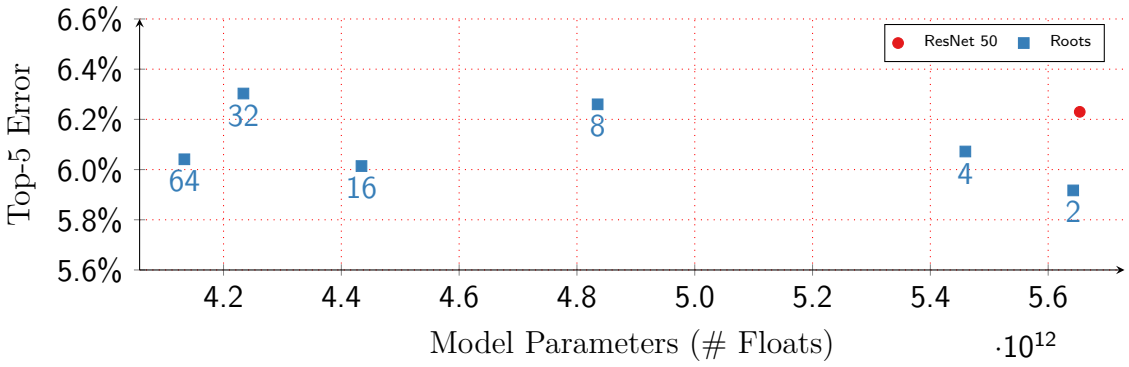


(d) CPU Forward Time *vs.* Top-5 Error

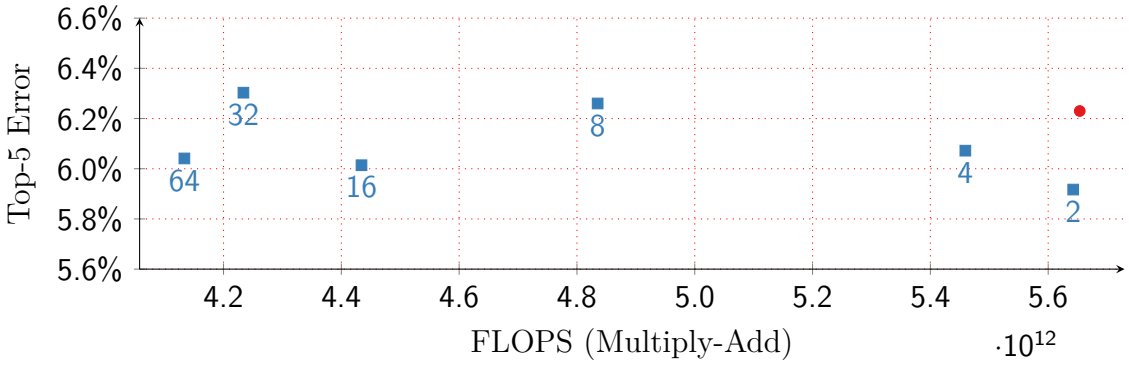
Fig. 5.13 **ResNet-50 ILSVRC Results.** Models with filter groups have fewer parameters, and less floating point operations, while maintaining error comparable to the baseline.

Table 5.5 **ResNet-200 ILSVRC Results**

Model	FLOPS $\times 10^{12}$	Param. $\times 10^{7}$	Top-1 Err.	Top-5 Err.
Orig.	5.65	6.25	0.2196	0.0623
root-2	5.64	6.24	0.2168	0.0592
root-4	5.46	6.06	0.2194	0.0607
root-8	4.84	4.91	0.2205	0.0626
root-16	4.43	3.98	0.2187	0.0601
root-32	4.23	3.51	0.2207	0.0630
root-64	4.13	3.28	0.2210	0.0604



(a) Model Parameters *vs.* Top-5 Error



(b) FLOPS (Multiply-Add) *vs.* Top-5 Error.

Fig. 5.14 **ResNet-200 ILSVRC Results.** Models with filter groups have fewer parameters, and less floating point operations, while maintaining error comparable to the baseline.

Table 5.6 **GoogLeNet ILSVRC Results.**

Model	FLOPS $\times 10^9$	Param. $\times 10^7$	Top-1 Acc.	Top-5 Acc.	CPU (ms)	GPU (ms)
Orig.	1.72	1.88	0.694	0.894	315	4.39
root-2	1.54	1.88	0.695	0.893	285	4.37
root-4	1.29	1.85	0.693	0.892	273	4.10
root-8	0.96	1.75	0.691	0.891	246	3.72
root-16	0.76	1.63	0.683	0.886	207	3.59

Table 5.7 **GoogLeNet**. Filter groups in each conv. layer and Inception module (*incp.*)

Model	conv1	conv2		incp. 3{a,b}			incp. 4{a–e}			incp. 5{a,b}		
	7×7	*1×1*	*3×3*	*1×1*	*3×3*	*5×5*	*1×1*	*3×3*	*5×5*	*1×1*	*3×3*	*5×5*
Orig.	1	1	1	1	1	1	1	1	1	1	1	1
root-2	1	1	2	1	1	1	1	1	1	1	1	1
root-4	1	1	4	1	2	2	1	1	1	1	1	1
root-8	1	1	8	1	4	4	1	2	2	1	1	1
root-16	1	1	16	1	8	8	1	4	4	1	2	2

implementing full training augmentation to achieve state-of-the-art results[2]. Table 5.5 and fig. 5.14 show the results, top-1 and top-5 error are for center cropped images. The models trained with roots have comparable or lower error, with fewer parameters and less computation. The root-64 model has 27% fewer FLOPS and 48% fewer parameters than ResNet 200.

5.3.5 Improving GoogLeNet on ILSVRC

We replicated the network as described by Szegedy, Liu, *et al.* (2015), with the exception of not using any training augmentation aside from random crops and mirroring, as supported by Caffe (Jia *et al.*, 2014)). To train we used the initialization of (He *et al.*, 2015) modified for compound layers, as described in section 4.3 and batch normalization without the scale and bias (Ioffe and Szegedy, 2015). At test time we only evaluate the center crop image.

While preserving the original number of filters per layer, we trained networks with various degrees of filter grouping, as described in table 5.7. While the Inception architecture is relatively complex, for simplicity, we always use the same number of

[2]https://github.com/facebook/fb.resnet.torch

groups within each of the groups of different filter sizes, despite them having different cardinality. For all of the networks, we only grouped filters within each of the 'spatial' convolutions (3×3, 5×5).

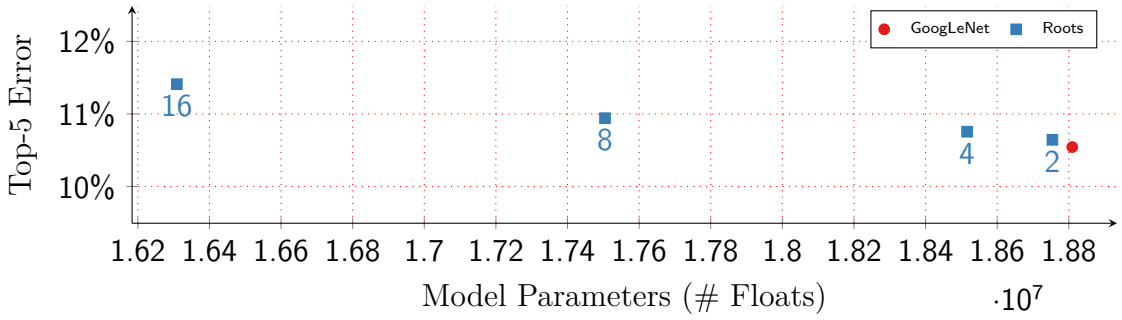
As shown in table 5.6, and plotted in fig. 5.15, our method shows significant reduction in computational complexity — as measured in FLOPS (multiply-adds), CPU and GPU timings — and model size, as measured in the number of floating point parameters. For many of the configurations the top-5 accuracy remains within 0.5% of the baseline model. The highest accuracy result, is 0.1% off the top-5 accuracy of the baseline model, but has a 0.1% higher top-1 accuracy — within the error bounds resulting from training with different random initializations. While maintaining the same accuracy, this network has 9% faster CPU and GPU timings. However, a model with only 0.3% lower top-5 accuracy than the baseline has much higher gains in computational efficiency — 44% fewer floating point operations (multiply-add), 7% fewer model parameters, 21% faster CPU and 16% faster GPU timings.

While these results may seem modest compared to the results for ResNet, GoogLeNet is by far the smallest and fastest near state-of-the-art model ILSVRC model. We believe that more experimentation in using different cardinalities of filter grouping in the heterogeneously-sized filter groups within each Inception module would improve results further.

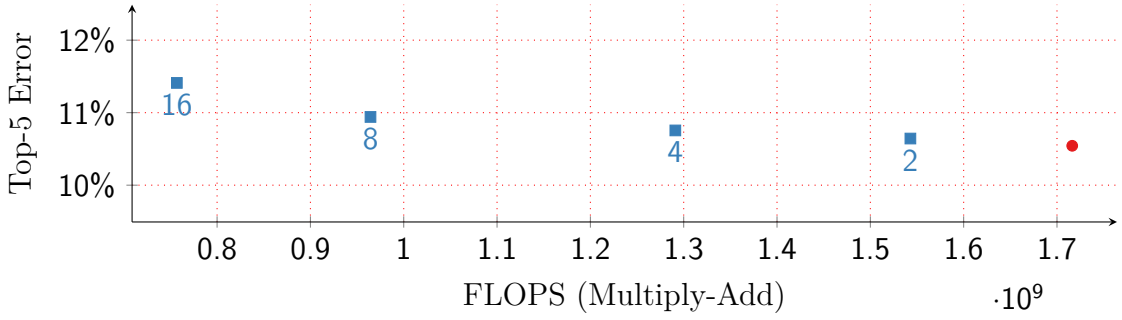
5.3.6 The Effect on Image-level Filters of Root Modules

In the ResNet root models, filter groups are used in `conv2`, directly after the image level filters of `conv1` some of the organization of filters can be directly observed, and give us intuition as to what is happening in root networks. Figure 5.16 shows the `conv0` filters learned for each of the ResNet 50 models. It is apparent that the filters learned in these networks are very similar to those learned in the original model, although sometimes inverted or with a different ordering. This ordering is somewhat consistent in models with filter groups however, even with different random initializations. This is because filter groups cause filters with strong mutual information to be grouped adjacent to each other.

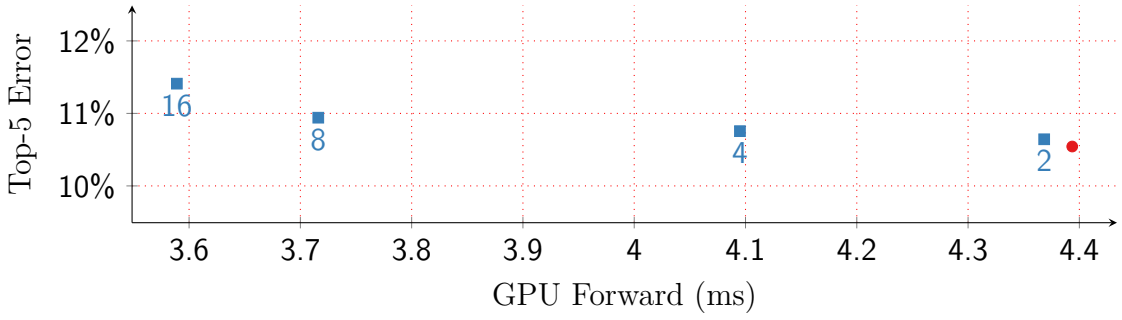
For example, in the root-8 network (fig. 5.16(d)), each row of filters corresponds to the input of an independent filter group in `conv2`. We can see that the first row primarily is composed of filters giving various directions of the same color gradient. These filters can be combined in the next layer to produce color edges easily. Due to the shortcut layer and the learned combinations of filters however, not all filter groupings are so obvious.



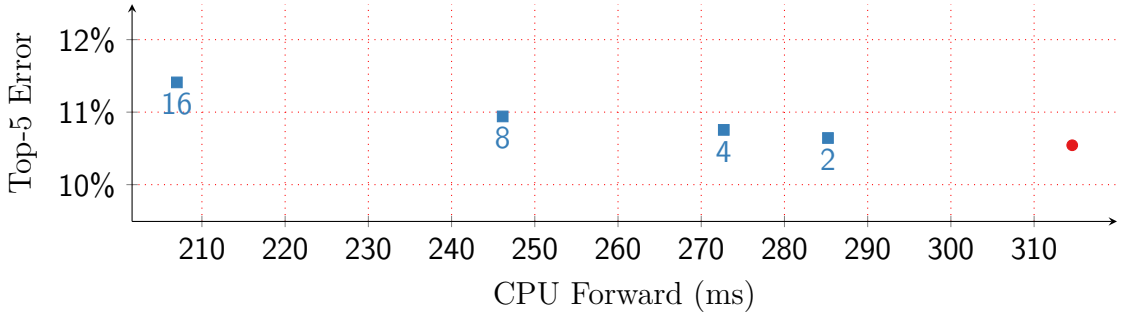
(a) **Model Parameters *vs.* Top-5 Error.**



(b) **FLOPS (Multiply-Add) *vs.* Top-5 Error.**

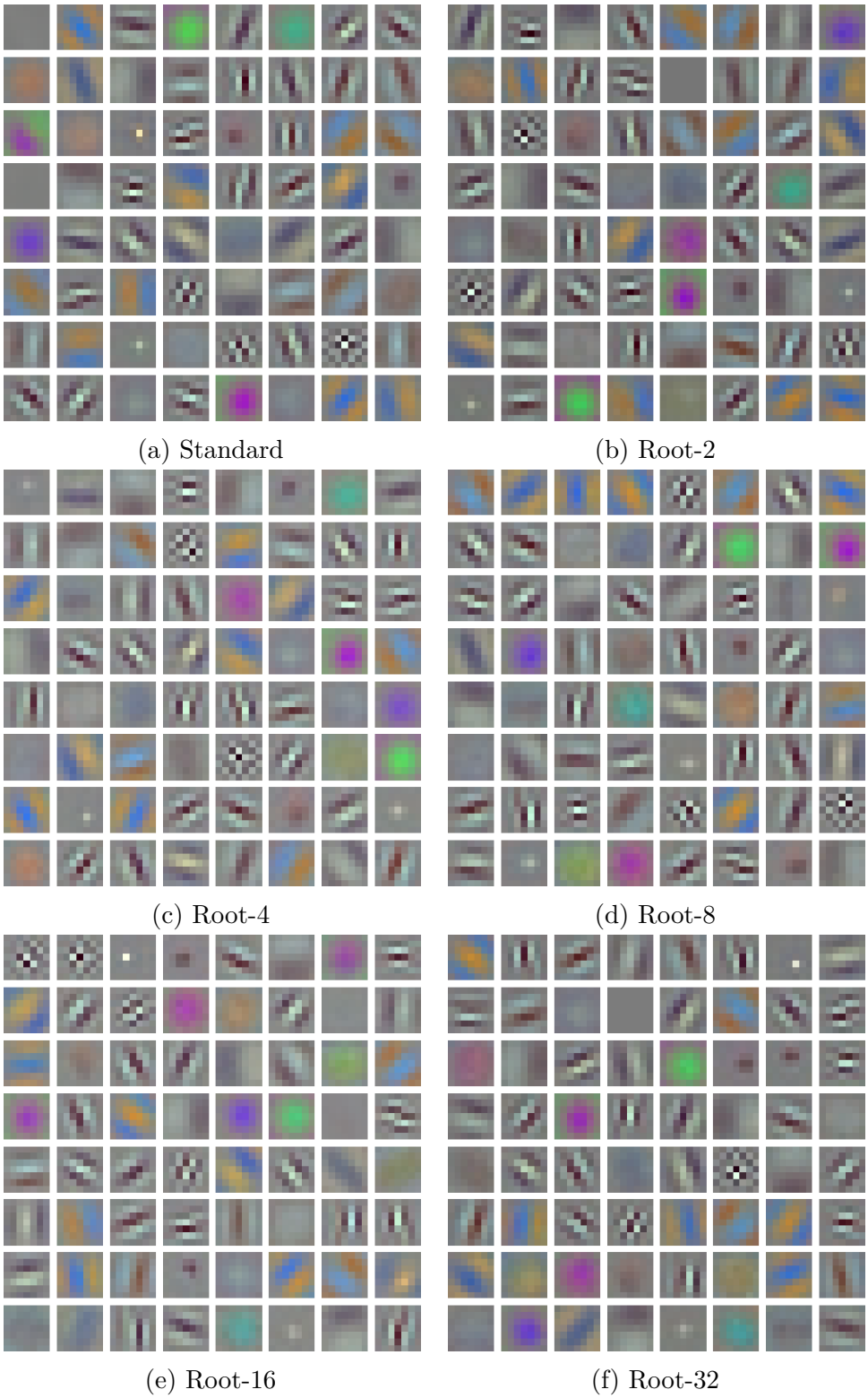


(c) **GPU Forward Time *vs.* Top-5 Error.**



(d) **CPU Forward Time *vs.* Top-5 Error.**

Fig. 5.15 **GoogLeNet ILSVRC Results.** Models with filter groups have fewer parameters, and less floating point operations, while maintaining error comparable to the baseline.



(a) Standard (b) Root-2

(c) Root-4 (d) Root-8

(e) Root-16 (f) Root-32

Fig. 5.16 **ResNet 50 `conv1` filters.** With filter groups directly after `conv1`, in `conv2`, some of the organization of filters can be directly observed, and give us intuition as to what is happening in root networks.

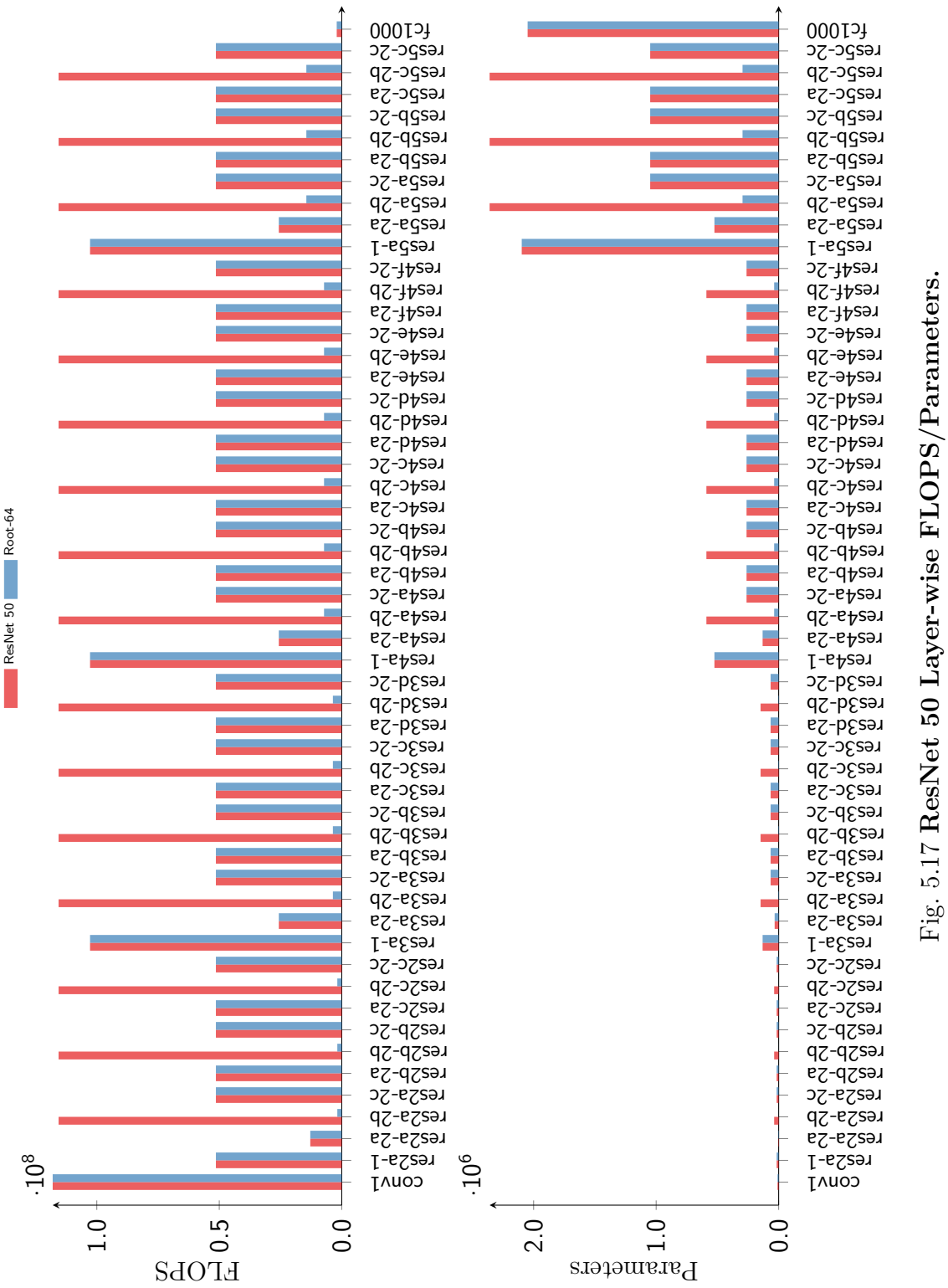


Fig. 5.17 **ResNet 50 Layer-wise FLOPS/Parameters.**

5.3.7 Layer-wise Compute/Parameter Savings

Figure 5.17 shows the difference in compute and parameters for each layer in a standard ResNet 50 model and a root-64 variant. The layers in the original networks with the highest computational complexity are clearly the spatial convolutional layers, *i.e.* layers with 3×3 spatial filters. When instead a root-module is used, the computational complexity of these layers is reduced dramatically. While the low dimensional embedding layers (1×1) are not changed, these have less than half the compute of the spatial convolution layers. The number of parameters in spatial convolution layers with large numbers of input channels, which increase towards the end of the network, are similarly reduced.

5.4 GPU Implementation

Our experiments show that our method can achieve a significant reduction in CPU and GPU runtimes for state-of-the-art CNNs without compromising accuracy. However, the reductions in GPU runtime were smaller than might have been expected based on theoretical predictions of computational complexity (FLOPs). We believe this is largely a consequence of the optimization of Caffe for existing network architectures (particularly AlexNet and GoogLeNet) that do not use a high degree of filter grouping.

Caffe presently parallelizes over filter groups by using multiple CUDA streams to run multiple CuBLAS matrix multiplications simultaneously. However, with a large degree of filter grouping, and hence more, smaller matrix multiplications, the overhead associated with calling CuBLAS from the host can take approximately as long as the matrix computation itself. To avoid this overhead, CuBLAS provides batched methods (*e.g.* `cublasXgemmBatched`), where many small matrix multiplications can be batched together in one call. Jhurani and Mullowney (2015) explore in depth the problem of using GPUs to accelerate the multiplication of very small matrices (smaller than 16×16), and show it is possible to achieve high throughput with large batches, by implementing a more efficient interface than that used in the CuBLAS batched calls. We have modified Caffe to use CuBLAS batched calls, and achieved significant speedups for our root-like network architectures compared to vanilla Caffe without CuDNN, e.g. a 25% speed up on our root-16 modified version of the GoogleNet architecture. However, our optimized implementation still is not as fast as Caffe with CuDNN (which was used to generate the results in this chapter), presumably because of other unrelated optimizations in the (proprietary) CuDNN library. Therefore we

suggest that direct integration of CuBLAS-style batching into CuDNN[3] could improve the performance of filter groups significantly.

5.5 Discussion

We explored the effect of using complex hierarchical arrangements of filter groups in CNNs and show that imposing a structured decrease in the degree of filter grouping with depth — a 'root' (inverse tree) topology — can allow us to obtain more efficient variants of state-of-the-art networks without compromising accuracy. Our method appears to be complementary to existing methods, such as low-dimensional embeddings, and can be used more efficiently to train DNNs than methods that only approximate a pre-trained model's weights.

We validated our method by using it to create more efficient variants of state-of-the-art NiN, Googlenet, and ResNet architectures, which were evaluated on the CIFAR-10 and ILSVRC datasets. Our results show comparable accuracy with the baseline architecture with fewer parameters and much less compute (as measured by CPU and GPU timings). For NiN on CIFAR-10, our model has 47% of the parameters of the original network, and approximately 22% faster CPU and GPU timings. For ResNet 50, our model has 27% fewer parameters, and was 24% (11%) faster on a CPU (GPU). For ResNet 200 our model has 27% fewer FLOPS and 48% fewer parameters. Even for the most efficient of the near state-of-the-art ILSVRC network, GoogLeNet, our model uses 7% fewer parameters and is 21% (16%) faster on a CPU (GPU). Even for the most efficient of the near state-of-the-art ILSVRC network, GoogLeNet, our model uses 7% fewer parameters and is 21% (16%) faster on a CPU (GPU).

[3]note that in August 2017, approximately a month before the submission of this disseration, the latest version of CuDNN, version 7, now supports the acceleration of filter groups.

6

Conditional Connectivity

> *"... the simplest most robust network which accounts for a data set will, on average, lead to the best generalization to the population from which the training set has been drawn"*
> – David Rumelhart, *in personal communication with Hanson and Pratt, 1987*

The ideal discriminative model desired for most tasks would have all the advantages of both neural networks and decision forests, and none of the weaknesses. It would have good generalization with computational efficiency, lend itself to semantic understanding and yet have sufficient functional complexity to solve complex problems. Different tasks require different assumptions however, and hence different model types, but it is rare that either one of these models in itself exhibits a clear and distinct advangtage in all aspects over the other for a particular task — yet we are limited to choosing one or the other in practice.

In this chapter we intend to explore the continuum of discriminative models that exist between decision forests and neural networks, to try to find such a balance. We will explore the theory and applications behind such models, with a focus on contemporary problems such as object class recognition, *i.e.* the ImageNet ILSVRC challenge which has been the focus of much of the recent work on deep learning.

6.1 On Methods of Discriminative Classification

Two methods of discriminative classification, *Neural Networks* and *Decision Forests*, have recently dominated the field of Computer Vision. Deep neural networks have even replaced the research in local features (*e.g.* SIFT), providing end-to-end learning from pixels to output (Yi *et al.*, 2016). Much work has been done on improving

both methods and exploring their applications — with academic and even commercial success. For example, decision forests are used to find the body pose of a person with the Kinect (Shotton *et al.*, 2011) used in game consoles. Deep CNNs on the other hand have recently surpassed human accuracy on what was considered one of the most challenging outstanding problems in computer vision, object class recognition (He *et al.*, 2015), and are already finding their ways into applications such as Google Photos[1]. However, the important fact that these two methods are related often seems to be all but forgotten. Sethi (1990) showed that any decision tree can be represented as a neural network with one hidden layer, however the converse does not necessarily hold true.

Despite this fundamental relationship, decision forests and neural networks have such distinct and mutually exclusive strengths and weaknesses that it is not surprising that they are themselves usually considered to be distinct. Decision forests require vast amounts of labelled data, proportional to the number of classes and tree depth, since samples are "diluted" down the tree, while, with appropriate regularization, neural networks can be trained with far more parameters than actual samples. At test time, neural networks are opaque giving little understanding, while decision forests are more intuitive — each node having an explicit decision on the input data and even describe per-class statistics. The routing of decision forests makes it easy to distribute computation, while the high connectivity of neural networks makes model parallelism difficult and inefficient. Decision forests are extremely fast at test time due to sample routing, only a small part of a tree need be computed, *i.e.* conditional computation. Neural networks must, on the other hand, computation the response at every node, even if many of these responses are approximately zero or not useful to the final output.

6.2 Generalizing DNNs and Decision Trees

Here we explore the continuum of models between decision forests, with the objective of reducing the connectivity of deep neural networks trained with backpropagation, specifically CNNs, while retaining some of the efficiency and understanding which arise from the conditional computation in decision forests.

Towards this objective, we generalize neural networks and decision trees intuitively by using a new graphical notation for representing both. This notation isolates the

[1] http://photos.google.com

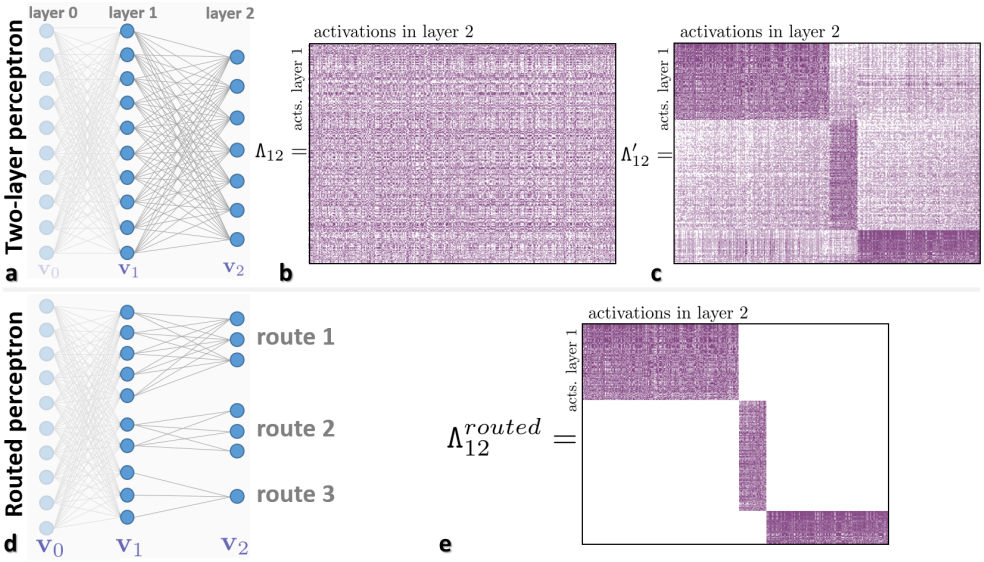


Fig. 6.1 **Block-diagonal correlation of activations, and data routing. (a)** An example 2-layer preceptron with ReLU activations. This is a portion of the 'VGG' model Simonyan and Zisserman, 2015 trained on Imagenet. **(b)** The correlation matrix Λ_{12} shows *unstructured* activation correlation between unit pairs. **(c)** Reordering the units reveals a noisy, block-diagonal structure. **(e)** Zeroing-out the off-diagonal elements is equivalent to removing connections between unit pairs. This corrsponds to the sparser, *routed* perceptron in **(d)**.

differences between the two models, such that we can represent a hybrid model, *i.e.* a *Conditional Network*, compactly[2].

6.3 Structured Sparsity and Data Routing

The seminal work in Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012 demonstrated how introducing rectified linear unit activations (ReLUs) allows *deep* CNNs to be trained effectively. Given a scalar input v_j, its ReLU activation is $\sigma(v_j) = \max(0, v_j)$. Thus, this type of non-linearity *switches off* a large number of feature responses within a CNN. ReLU activations induce a data-dependent sparsity; but this sparsity does not tend to have much structure in it. Enforcing a special type of *structured* sparsity is at the basis of the efficiency gain attained by conditional networks. We illustrate this concept with a toy example.

The output of the exemplar multi-layer perceptron (MLP) of fig. 6.1a is computed as $\mathbf{v}_2 = \sigma(\mathtt{P}_{12}\mathbf{v}_1) = \sigma(\mathtt{P}_{12}\sigma(\mathtt{P}_{01}\mathbf{v}_0))$. Given a trained MLP we can look at the average

[2]This notation itself was created by Dr. Antonio Criminisi, and is not a contribution of this dissertation. Some figures are used with the permission of Dr. Antonio Criminisi/Microsoft Research.

correlation of activations between pairs of units in two successive layers, over all training data. For example, the matrix Λ_{12} (fig. 6.1b) shows the joint correlations of activations in layers 1 and 2 in a perceptron trained on the Imagenet classification task.[3] Here we use the final two layers of the deep CNN model of Simonyan and Zisserman, 2015 with a reduced number of features (250) and classes (350) to aid visualization.

Thanks to the ReLUs, the correlation matrix Λ_{12} has many zero-valued elements (in white in fig. 6.1b), and these are distributed in an *unstructured* way. Reordering the rows and columns of Λ_{12} reveals an underlying, noisy block-diagonal pattern (fig. 6.1c). This operation corresponds to finding groups of layer-1 features which are highly active for certain subsets of classes (indexed in layer-2). Thus, the darker blocks in Fig. fig. 6.1c correspond to three super-classes (sets of 'related' classes). Zeroing out the off-diagonal elements (Fig. fig. 6.1e) corresponds to removing connections between corresponding unit pairs. This yields the sparse architecture in Fig. fig. 6.1d, where selected subsets of the layer-1 features are sent (after transformation) to the corresponding subsets of layer-2 units; thus giving rise to data routing.

We have shown how imposing a block-diagonal pattern of sparsity to the joint activation correlation in a neural network corresponds to equipping the network with a tree-like, routed architecture. Next section will formalize this intuition further and show the benefits of sparse architectures.

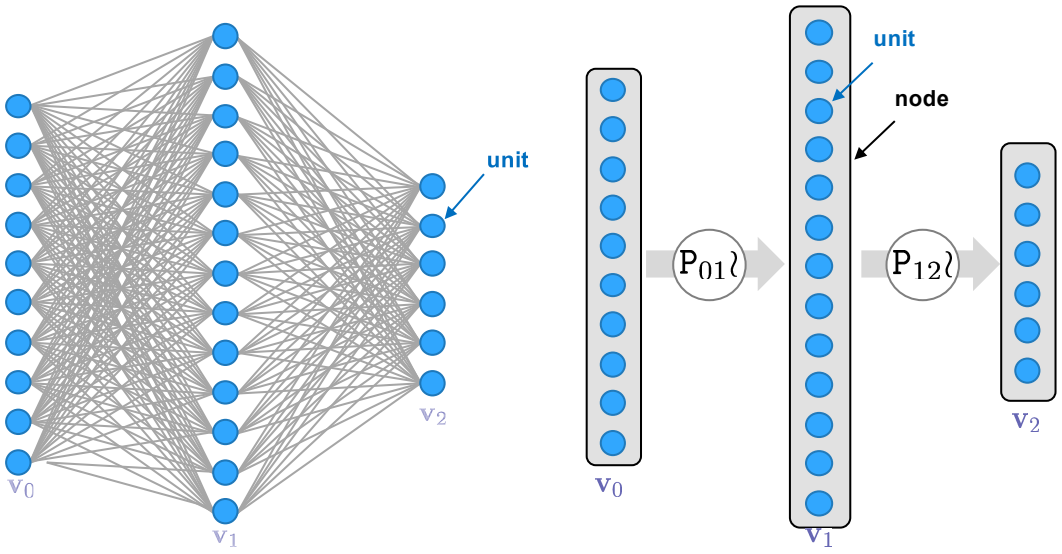
6.4 A New Graphical Notation

The proposals we will make require a re-interpretation of existing classification models, that is neural networks and decision trees, but standard graphical diagrams for both of these models hides the implicit functional similarities on which we will build our models, and are instead connection-centric — focused on showing the connectivity of the models rather than the underlying data transformations. As such, before we are able to explain the concept of a conditional network, a new graphical language is proposed.

6.4.1 Neural Networks

The standard depiction of a neural network with one hidden layer is shown in fig. 6.2(a), where each of the layers is fully-connected, and these connecting weights are illustrated as lines between the neurons represented as circles. While this image illustrates the

[3]The correlation matrix Λ_{12} is not the same as the weight matrix $\mathtt{P}_{12}$.



(a) Standard diagram of a neural network with one hidden layer.

(b) New notation showing transformation between layers explicitly.

Fig. 6.2 **The proposed compact graphical notation for neural networks.** Non-linear transformations in a standard neural network with one hidden layer are indicated by the projection matrix $\mathtt{P}$ between the two layers, followed by a generic non-linearity, represented with the symbol ≀. ©Antonio Criminisi, used with permission.

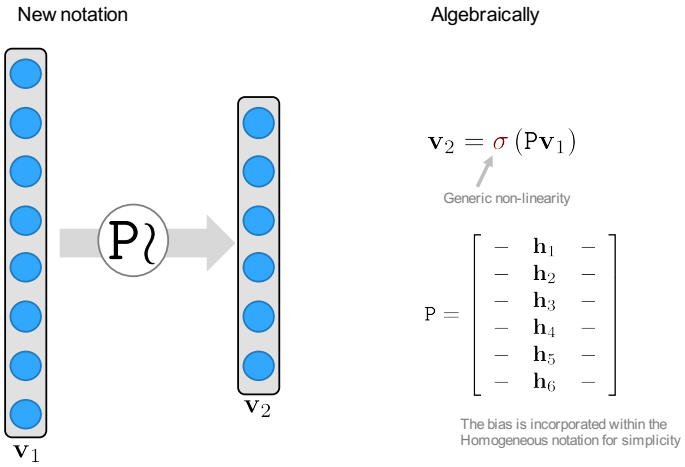
connectivity of the model, it assumes the function of the neurons themselves to be known or otherwise described. In fig. 6.2(b) we use a different notation to show both connectivity and function of each layer, with the assumption that all nodes on a particular layer have the same function.

In this simple example of a fully-connected neural network, between layers i and j, every node outputs the non-linear transformation, $\mathbf{v}_j = \sigma(\mathtt{P}\mathbf{v}_i)$, a composition of the non-linear function σ (*e.g.* a ReLU or sigmoid) and the projection of the input units with a projection matrix $\mathtt{P}$, which includes the bias term in homogeneous coordinates. We denote this operation explicitly as $\mathtt{P}_{ij}≀$, where $\mathtt{P}_{ij}$ is the projection matrix and ≀ represents a non-linearity. In short $\mathtt{P}_{ij}≀$ denotes the standard neural net layer's non-linear transformation $\mathbf{v}_j = \sigma(\mathtt{P}\mathbf{v}_i)$.

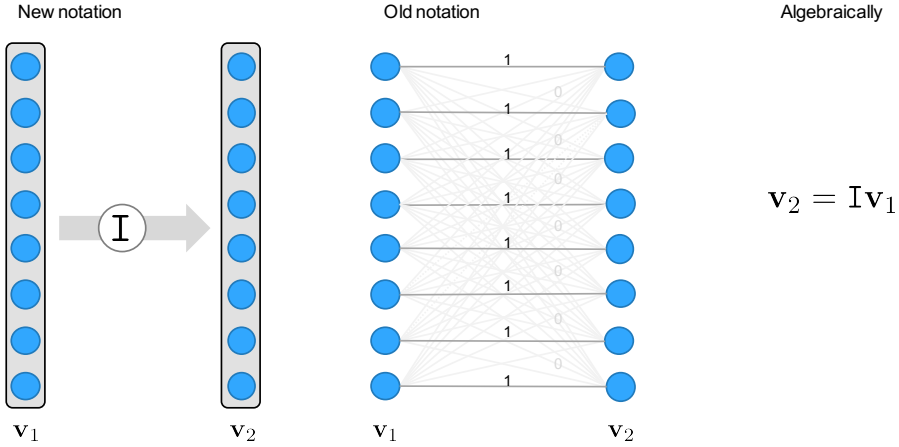
CNNs typically include layers with pooling operations (*e.g.* GAP), or local response normalization. Any of these operations may also be represented by the function σ.

6.4.2 Decision Trees and Random Forests

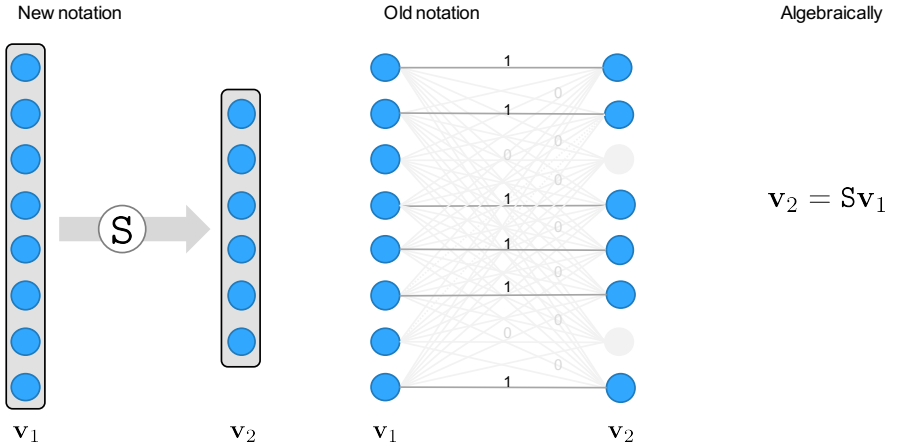
This graphical language may also represent decision trees, also typically depicted in a connection-centric graphical diagram as shown in fig. 6.5(a). Decision trees typically



(a) Generic projection with non-linearity, typically found in neural networks.



(b) Identity projection with identity function, typically found in decision trees.



(c) Selection projection, found in random forests.

Fig. 6.3 **The proposed compact graphical notation for various types of projection in a conditional network.** ©Antonio Criminisi, used with permission.

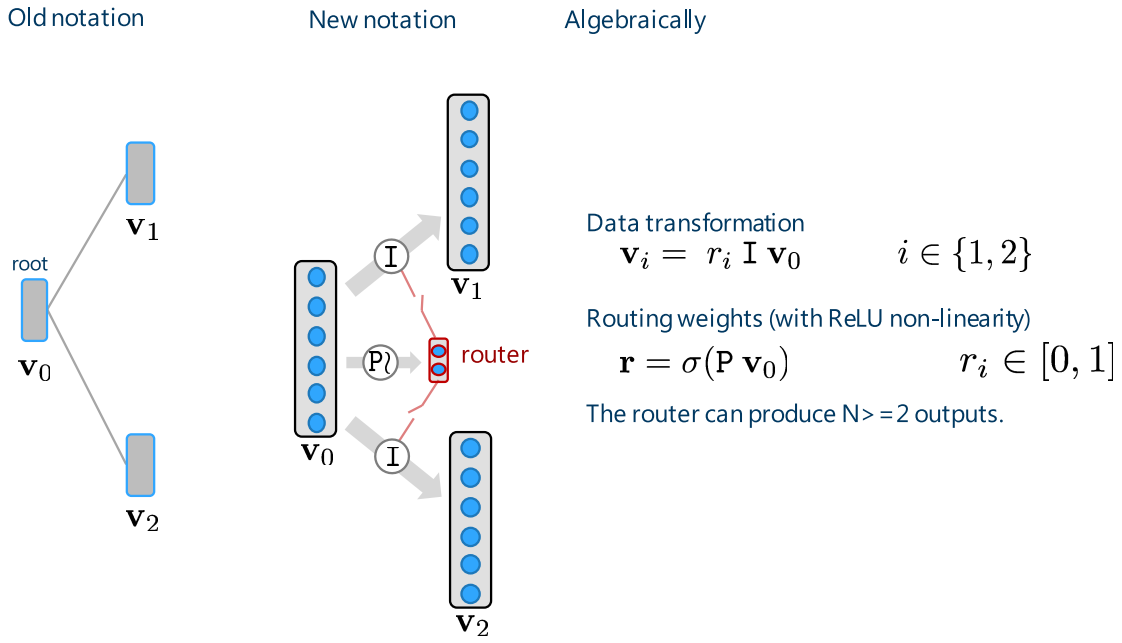


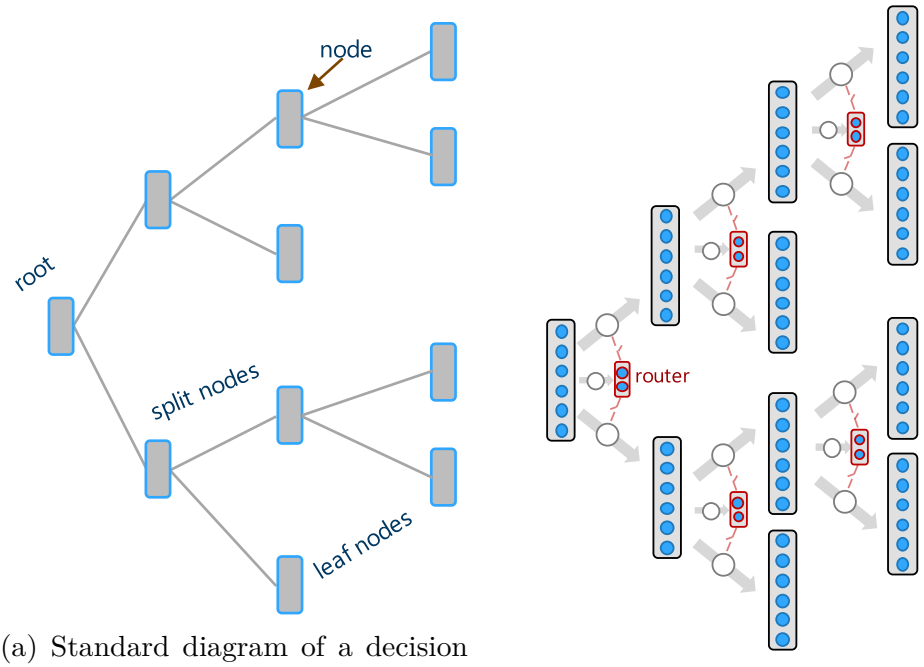
Fig. 6.4 **The proposed compact graphical notation for a "decision stump".**

copy, or reference, samples from the root of the tree down to the leaf (or leaves) without transformation. This is notably in contrast with representation learning approaches, such as neural networks, which try to learn optimal data transformations during training with a full projection matrix, as illustrated in fig. 6.3(a). There have, however, been attempts to incorporate representation learning within decision forests (Bulò and Kontschieder, 2014; Montillo *et al.*, 2011). The copying of the sample may also be considered as a special case of the transformation $\mathbf{v}_j = \sigma(\mathtt{P}\mathbf{v}_i)$, where the projection is the identity matrix $\mathtt{P}_{ij} = \mathtt{I}$, and the function σ is the identity function $\sigma(\mathbf{v}_i) = \mathbf{v}_i$. As such, we use the identity $\mathtt{I}$ in our graphical language to denote the routing between each tree level. This is explained graphically in fig. 6.3(b).

Random forests consist of a number of decision trees, each of which is applied to a restricted number of the input feature dimensionality. It may not be immediately obvious how this is represented in our new graphical language, but in fact a simple extension of the above theme represents selection — *i.e.* an identity matrix of reduced rank, as illustrated in fig. 6.3(c).

6.4.3 Explicit Routing

We are still missing the method of conditional computation found in decision trees in our graphical language however, *i.e.* how the decision is made to route each sample at



(a) Standard diagram of a decision tree.

(b) New explicit graphical notation.

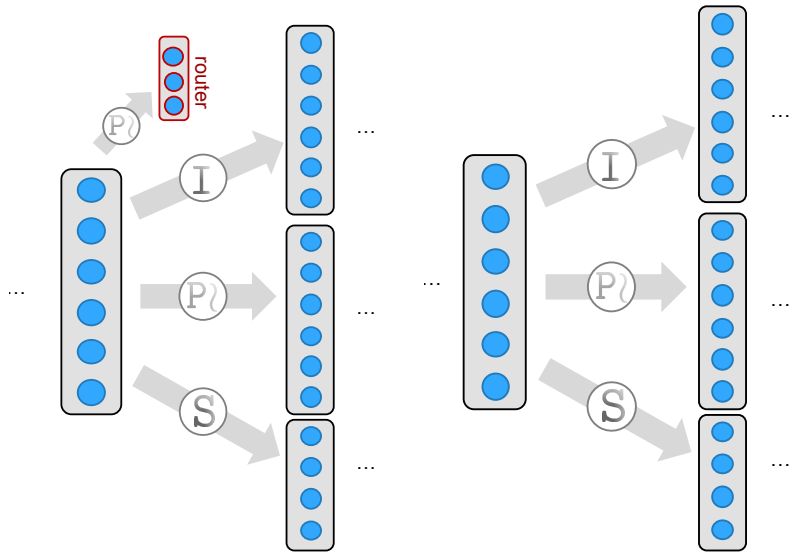
Fig. 6.5 **The proposed compact graphical notation for a full decision tree.** ©Antonio Criminisi, used with permission.

a node. We must generalize two forms of routing found in decision trees, *hard routing* where samples are only routed to one node in the next layer, and *soft routing* where a weighted sample is potentially sent to every node of the next layer.

We achieve this with the minor addition of a new set of nodes we call *routers*. A router consists of K weights for a K-ary node or tree, as shown in fig. 6.4. These router weights themselves are determined in a way more reminiscent of a neuron's activation function, typically a non-linear transformation of the sample. Thus this is represented in the same graphical notation as the mapping between neural network layers, *i.e.* as $\mathtt{P}_{ij}\wr$. Figure 6.5(b) shows the same tree as shown in fig. 6.5(a), notably with the routers highlighted in red. Typically a router will have a number of non-zero weights and perform soft routing, however if only one of the router weights is non-zero, the router effects hard routing.

6.4.4 Implicit Routing

It is in fact possible for networks to learn a conditional routing of data without an explicit router. We call this form of conditional routing *implicit routing vs. explicit routing*, where a router is used.



(a) Explicitly routed network. (b) Implcitly routed network.

Fig. 6.6 **Explicit *vs.* implicitly routed networks.** ©Antonio Criminisi, used with permission.

In an explicitly routed network, the routes are trained by combining the routes before the training loss. For example, in fig. 6.6(a) the input to layer v_2, y_1, is a linear combination of the routes weighted by the router weights,

$$y_1 = \sum_j r_j \sigma \left(P^j \mathbf{v}_1^j \right). \tag{6.1}$$

For an implicitly routed network instead a (non-weighted) linear combination is followed by a single fully-connected layer, *i.e.* inner product and ReLU, *i.e.* for fig. 6.6(b), the output of $\mathbf{v}_2$, is simply,

$$y_1 = \max \left(0, \sum_j v_1^j \sigma(P^j \mathbf{v}_0) \right). \tag{6.2}$$

6.4.5 Conditional Networks

The generalization, a conditional network, mixes elements of both of these models. Conditional networks may perform arbitrary projections of samples, and use arbitrary non-linear functions. Conditional networks may route samples with a soft or hard router in some or all of the layers, they may select some of all of the input feature space.

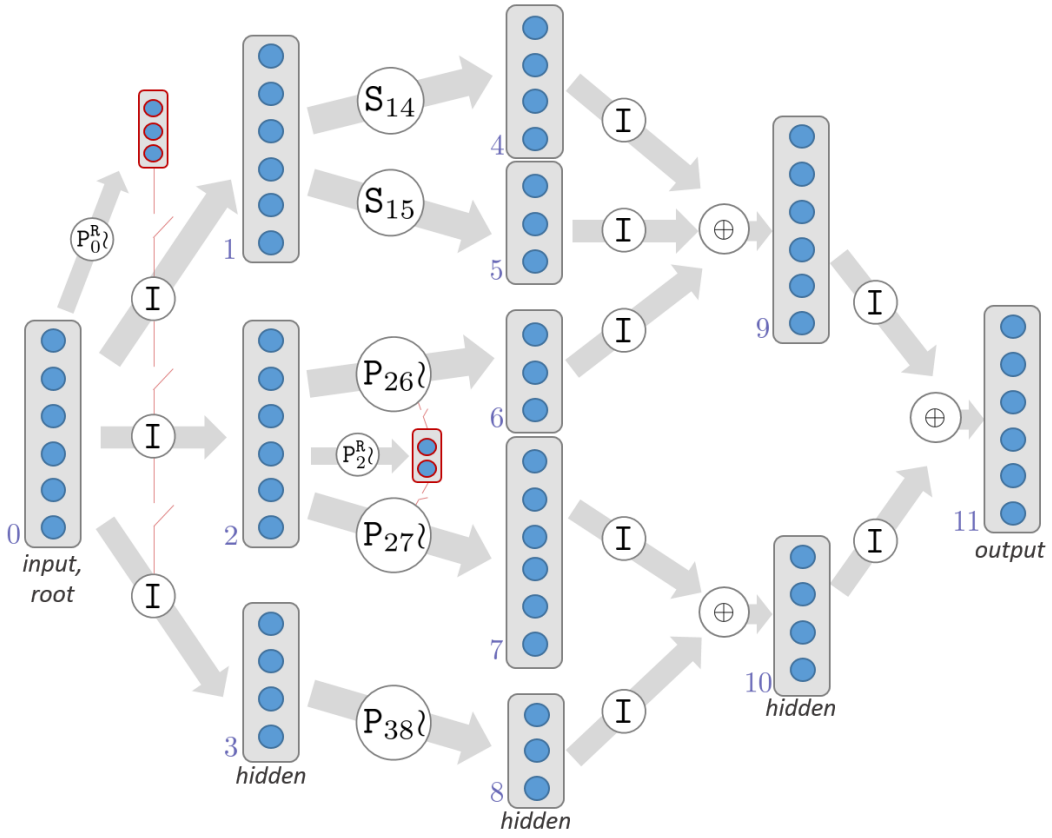


Fig. 6.7 **A generic conditional network.** Conditional networks fuse efficient data routing with accurate data transformation in a single model. Vector concatenations are denoted with ⊕.

6.4.6 Computational Efficiency

Efficiency through explicit data routing. Split nodes can have *explicit* routers where data is *conditionally* sent to the children according to the output of a routing function (*e.g.* node 2 in fig. 6.7), or have *implicit* routers where the data is unconditionally but selectively sent to the children using selection matrices $\mathtt{S}$ (*e.g.* node 1). If the routing is explicit and hard (like in trees), then successive operations will be applied to ever smaller subsets of incoming data, with the associated compute savings. Next we show how implicit conditional networks can also yield efficiency.

Efficiency of implicit routed networks. Figure 6.8 compares a standard CNN with a 2-routed architecture. The total numbers of filters at each layer is fixed for both to c_1, c_2 and c_3. The number of multiplications necessary in the first convolution is $c_2 \times c_1 k_x k_y W H$, with W, H the size of the feature map and k_x, k_y the kernel size (for simplicity here we ignore max-pooling operations). This is the same for both

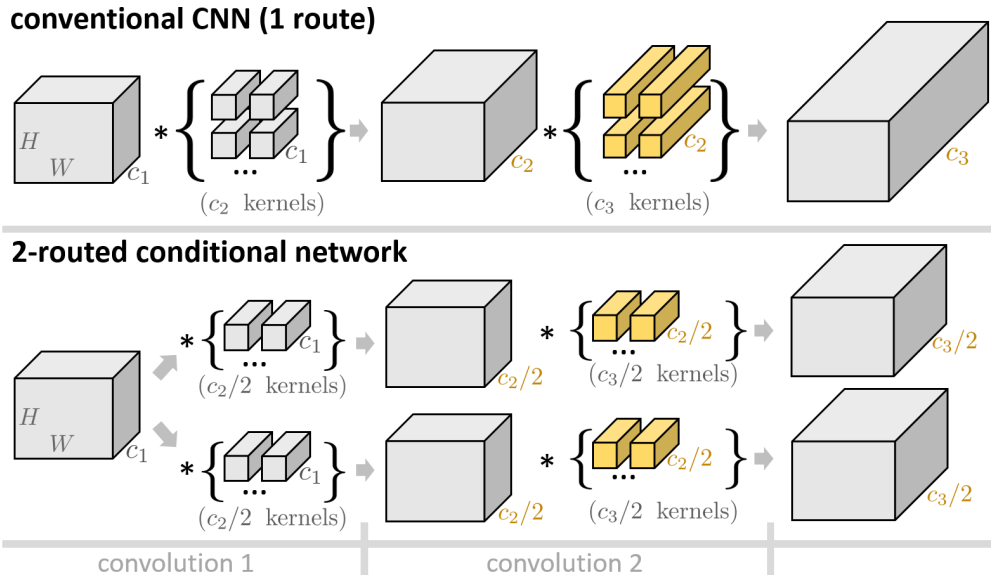


Fig. 6.8 **Computational efficiency of implicit conditional networks. (top)** A standard CNN (one route). **(bottom)** A two-routed architecture with no explicit routers. The larger boxes denote feature maps, the smaller ones the filters. Due to branching, the depth of the second set of kernels (in yellow) changes between the two architectures. The reduction in kernel size yields fewer computations and thus higher efficiency in the branched network.

architectures. However, due to routing, the depth of the second set of filters is different between the two architectures. Therefore, for the conventional CNN the cost of the second convolution is $c_3 \times c_2 k_x k_y W H$, while for the branched architecture the cost is $c_3 \times \left(\frac{c_2}{2}\right) k_x k_y W H$, *i.e.* half the cost of the standard CNN. The increased efficiency is due *only* to the fact that shallower kernels are convolved with shallower feature maps. Simultaneous processing of parallel routes may yield additional time savings.[4]

6.4.7 Back-propagation Training

Implicitly-routed conditional networks can be trained with the standard back-propagation algorithm Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012; Szegedy, Liu, *et al.*, 2015. The selection functions `S` become extra parameters to optimize over, and their gradients can be derived straightforwardly. Now we show that *explicitly*-routed networks can also be trained using back-propagation. To do so we need to compute partial derivatives with respect to the router's parameters (all other differentiation operations are as in conventional CNNs). We illustrate this using the small network in fig. 6.9.

[4]Feature not yet implemented in Caffe Jia *et al.*, 2014.

Here subscripts index layers and superscripts index routes (instead, in fig. 6.7 the subscripts indexed the input and output nodes). The training loss to be minimized is

$$L(\boldsymbol{\theta}) = \frac{1}{2}\left(\mathbf{v}_2(\boldsymbol{\theta}) - \mathbf{v}_2^*\right)^\top \left(\mathbf{v}_2(\boldsymbol{\theta}) - \mathbf{v}_2^*\right), \tag{6.3}$$

with $\boldsymbol{\theta} = \{\{\mathtt{P}^j\}, \mathtt{P}^\mathtt{R}\}$ denoting the parameters of the network, and $\mathbf{v}_2^*$ the ground-truth assignments to the output units. We define this energy for a single training data point, though the extension to a full dataset is a trivial outer summation. The network's forward mapping is

$$\mathbf{v}_1^j = \sigma\left(\mathtt{P}^j \mathbf{v}_0\right) \quad \text{and} \quad \mathbf{v}_2(\boldsymbol{\theta}) = \mathbf{r}(\boldsymbol{\theta})\ \mathtt{V}_1(\boldsymbol{\theta}), \tag{6.4}$$

with $\mathbf{r} = \sigma\left(\mathtt{P}^\mathtt{R}\mathbf{v}_0\right)$ the output of the router. In general: i) the routing weights $\mathbf{r}$ are *continuous*, $r(i) \in [0, 1]$, and ii) multiple routes can be "on" at the same time. $\mathtt{V}_1$ is a matrix whose j-th row is $(\mathbf{v}_1^j)^\top$. The update rule is $\Delta\boldsymbol{\theta}_{t+1} := -\rho \left.\frac{\partial E}{\partial \boldsymbol{\theta}}\right|_t$, with t indexing iterations. We compute the partial derivatives through the chain rule as follows:

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \mathbf{v}_2}\frac{\partial \mathbf{v}_2}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \mathbf{v}_2}\left(\frac{\partial \mathbf{r}}{\partial \mathtt{P}^\mathtt{R}}\mathtt{V}_1 + \sum_{j=1}^{R} r(j)\,\frac{\partial \mathbf{v}_1^j}{\partial \boldsymbol{\phi}^j}\frac{\partial \boldsymbol{\phi}^j}{\partial \mathtt{P}^j}\right), \tag{6.5}$$

with $\boldsymbol{\phi}^j := \mathtt{P}^j\mathbf{v}_0$, and R the number of routes. eq. (6.5) shows the influence of the soft routing weights on the back-propagated gradients, for each route. Thus, explicit routers can be trained as part of the overall back-propagation procedure. Since trees and DAGs are special instances of conditional networks, now we have a recipe for training them via back-propagation (*cf*. Kontschieder *et al.*, 2015; Schulter *et al.*, 2013; Suárez and Lutsko, 1999).

In summary, conditional networks may be thought of as:

1. Decision trees/DAGs which have been enriched with (learned) data transformation operations, or as

2. CNNs with rich, DAG-shaped architectures and trainable data routing functions. Next, we show efficiency advantages of such branched models with comparative experiments.

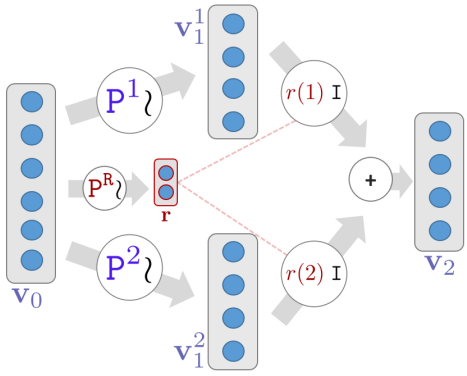


Fig. 6.9 **Training a network's routers via back-propagation.** A toy conditional network used to illustrate how to train the router's parameters $\mathtt{P}^{\mathtt{R}}$ via gradient descent back-propagation.

6.5 Results

6.5.1 Conditional Sparsification of a Perceptron

We begin with a toy experiment, designed to illustrate potential advandages of using explicit routes within a neural network. We take a perceptron (the last layer of "VGG11" Simonyan and Zisserman, 2015) and train it on the 1,000 Imagenet classes, with no scale or relighting augmentation Jia *et al.*, 2014. Then we turn the perceptron into a small tree, with R routes and an additional, compact perceptron as a router (see fig. 6.10a). The router $\mathtt{P}_8^{\mathtt{R}}$ and the projection matrices $\mathtt{P}_8^i$ are trained to minimize the overall classification loss (section 6.4.7).

Interpolating between trees and CNNs. Given a test image we apply the convolutional layers until the beginning of the tree. Then we apply the router, and its R outputs are soft-max normalized and treated as probabilities for deciding which route/s to send the image to. We can send the image only to the highest probability route only (as done in trees) or we could send it to multiple routes, *e.g.* the τ most probable ones. For $\tau = 1$ we reproduce the behaviour of a tree. This corresponds to the left-most point in the curves in fig. 6.10b (lowest cost and higher error). Setting $\tau = R$ corresponds to sending the image to *all* routes. The latter reproduces the same behaviour as the CNN, with nearly the same cost (lowest error and highest compute cost point in the curves). Different values of $\tau \in \{1, \ldots, R\}$ correspond to different points along the error-cost curves.

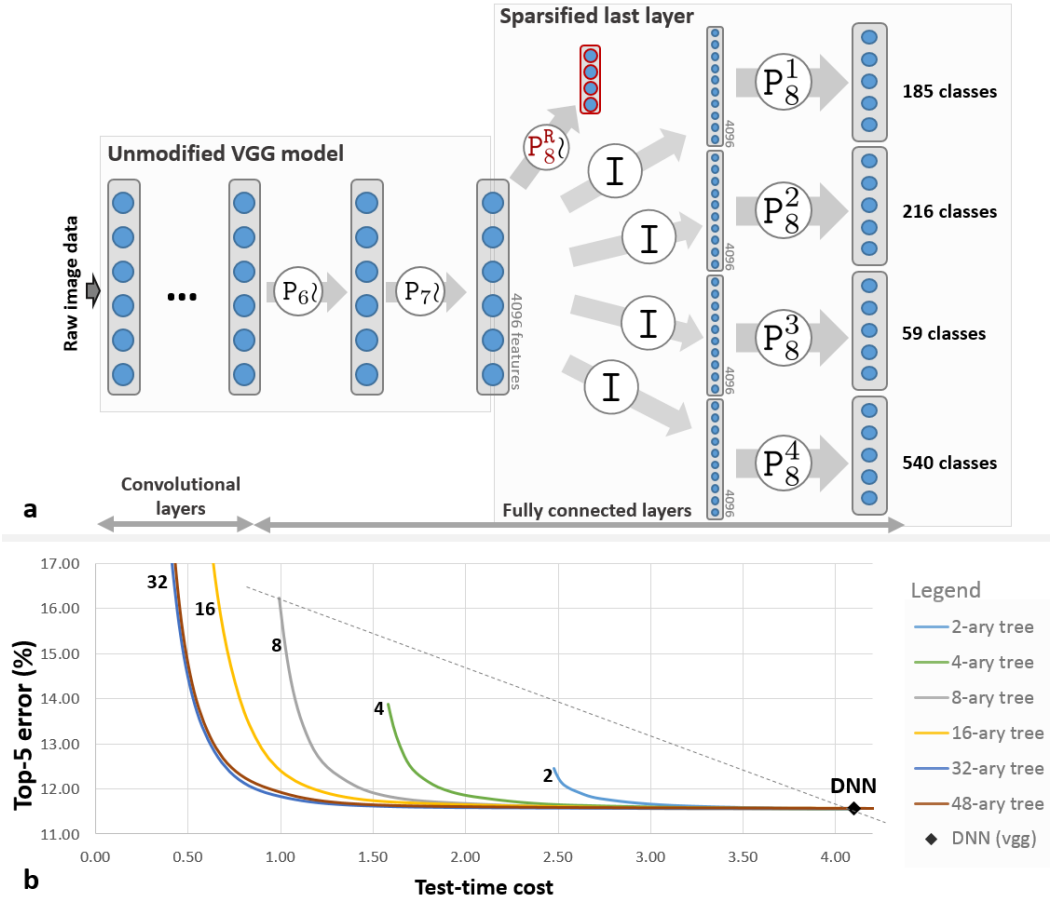


Fig. 6.10 **Conditional sparsification of a single-layer perceptron. (a)** We take the deep CNN model in Simonyan and Zisserman, 2015 ('VGG11') and turn the last fully connected layer (size 4095×1000) into a tree with R routes ($R = 4$ shown in figure). **(b)** The top-5-error *vs.* test-time-cost curves for six conditional networks trained with different values of $R \in \{2, 4, 6, 8, 16, 24, 32\}$. Test-time cost is computed as number of floating point operations per image, and is hardware-independent. The strong sub-linear shape of the curves indicates a net gain in the trade-off between accuracy and efficiency.

Dynamic accuracy-efficiency trade-off. The ability to select the desired accuracy-efficiency operating point at *run-time* allows *e.g.* better battery management in mobile applications. In contrast, a CNN corresponds to a *single* point in the accuracy-efficiency space (see the black point in fig. 6.10b). The pronounced sub-linear behaviour of the curves in fig. 6.10b suggests that we can increase the efficiency considerably with little accuracy reduction (in the figure a 4-fold efficiency increase yields an increase in error of less than 1%).

Why care about the amount of computation? Modern parallel architectures (such as GPUs) yield high classification accuracy in little time. But parallelism is not the only way of increasing efficiency. Here we focus on reducing the total amount of computations while maintaining high accuracy. Computation affects power consumption, which is of huge practical importance in mobile applications (to increase battery life on a smartphone) as well as in cloud services (the biggest costs in data centres are due to their cooling). Next we extend conditional processing also to the expensive convolutional layers of a deep CNN.

6.5.2 ILSVRC

We first validate the use of conditional networks for image classification on the ILSVRC dataset (Russakovsky *et al.*, 2015), a large dataset consisting of 1.2M training images for 1000 classes, and 50,000 validation images.

As discussed in section 5.1, AlexNet uses two filter groups throughout most of the layers of the model in order to split computation across two GPUs. The authors observed that the filters on each GPU appeared to specialize to learn fundamentally different features regardless of initialization (Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012). This interesting observation has mostly been ignored in subsequent networks where GPU memory has increased enought that such a split of the network is not required, but the original observation is a fundamental motivation of our work.

We based our experiments on the VGG network (Simonyan and Zisserman, 2015) on which the current state-of-the-art models are also based (He *et al.*, 2015). Specifically, we focus on the VGG-11 model as it is deep (11 layers) and relatively memory efficient (trains with Caffe (Jia *et al.*, 2014) on a single Nvidia K40 GPU). It notably does not have any filter grouping, as found in AlexNet, or low-dimensional embeddings, as found in NiN. It therefore suffers from an explosion in the number of filters required at each layer, and represents the ideal network on which to demonstrate the efficiency savings brought about by these simple modifications.

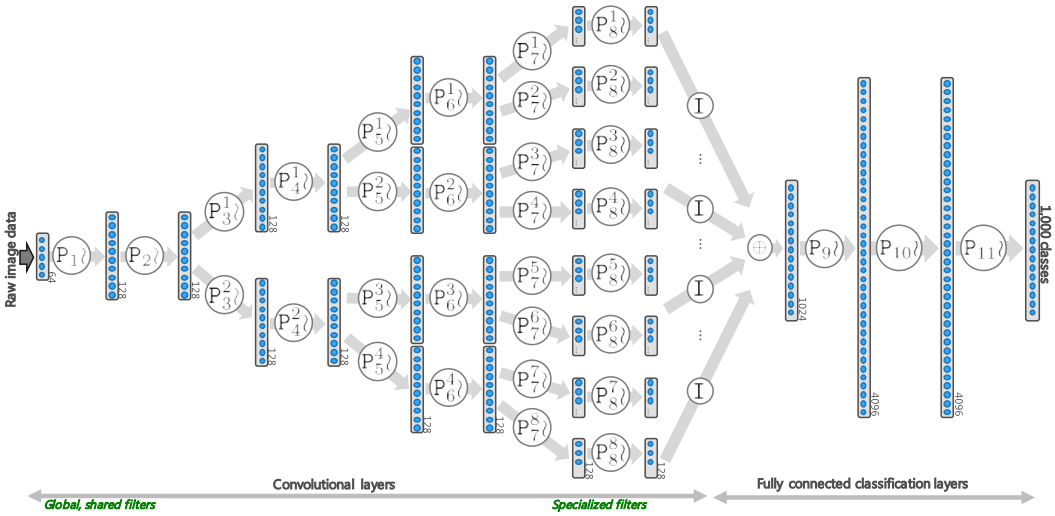


Fig. 6.11 **Conditional network used with Imagenet experiments.** The network employs implicit data routing in the (expensive) convolutional layers to yield higher computational efficiency than the corresponding, non-routed DNNs.

Global Max-Pooling to Reduce Model Size

We found that GMP, like GAP (as used by Lin, Q. Chen, and Yan (2014) and Szegedy, Liu, *et al.* (2015)), after the last convolutional layer is effective in reducing model parameters while maintaining, if not improving, accuracy. This suggests that preserving spatial information after the convolutional layers may not be as important as previously thought. We trained a new network ('VGG-11-GMP') with such pooling, and achieved lower top-5 error than the baseline VGG-11 network (13.3% *vs.* 13.8%), with a decrease in the number of parameters of over 72% (see fig. 6.13).

Designing an Efficient Conditional Architecture

Starting from the already improved, non-routed VGG-11-GMP architecture, we designed the conditional network in fig. 6.11. Since most of the computational cost in VGG-11-GMP is in the convolutional layers, our conditional variant introduces a DAG-like routed structure to split the filters in the convolutional section. The assumption here is that each filter should only need to be applied to a small number of channels in the input feature map.

Data routing is implemented via 'filter groups', as originally used in (Krizhevsky, Sutskever, and Geoffrey E. Hinton, 2012). Thus, at each level conv_n_$\{1,2\}, n = 3\ldots5$, the convolutional filters of VGG-11-GMP are divided into $2^{(n-2)}$ groups. Each group depends only on the results of exactly 128 previous filters. The feature maps of

the last convolutional layer are concatenated together, and globally max-pooled before the fully-connected layers, which remain the same as those in VGG-11-GMP.

Training

We trained our conditional network with the same hyperparameters as in (Simonyan and Zisserman, 2015), except for using the initialization strategy suggested of (He *et al.*, 2015), and a learning schedule,

$$\gamma_t = \gamma_0(1 + \gamma_0\lambda t)^{-1}, \tag{6.6}$$

where γ_0, γ_t and λ are the initial learning rate, learning rate at iteration t, and weight decay respectively (Bottou, 2012). When the validation accuracy of the network levelled out, the learning rate was further decreased by a factor of 10, twice. The conditional network took twice as many epochs to train than VGG-11, however this equates to a comparable training time given its higher efficiency.

Accuracy *vs.* Efficiency

In order to compare different network architectures as fairly as possible, here we did not use any training augmentation aside from that supported by Caffe (Jia *et al.*, 2014) (mirroring/random crops). Similarly we report test-time accuracy based only on centre-cropped images, without potentially expensive data oversampling. This reduces the overall accuracy (w.r.t. to state of the art), but constitutes a fairer test bed for teasing out the effects of different network architectures. This is because each architecture uses a different method of augmentation and has a different affect on the increase in inference time. Applying the same oversampling to all networks produced a nearly identical accuracy improvement in all models, without changing their ranking.

Table 6.1 summarize the results, and fig. 6.12 shows top-5 error as a function of test-time cost[5] and model size. The best network by these measures (*i.e.* closest to the origin) is GoogLeNet (Szegedy, Liu, *et al.*, 2015). In both networks much of the computational saving is obtained by routing subsets of features to different branches of the network. GoogLeNet learns low-dimensional embeddings, has multiple intermediate training losses, and a very different training schedule. These differences, along with its deeper DAG structure, may explain its superior performance.

[5]Measured here as number of multiply-accumulate operations. We have observed this measure to correlate very well with CPU time.

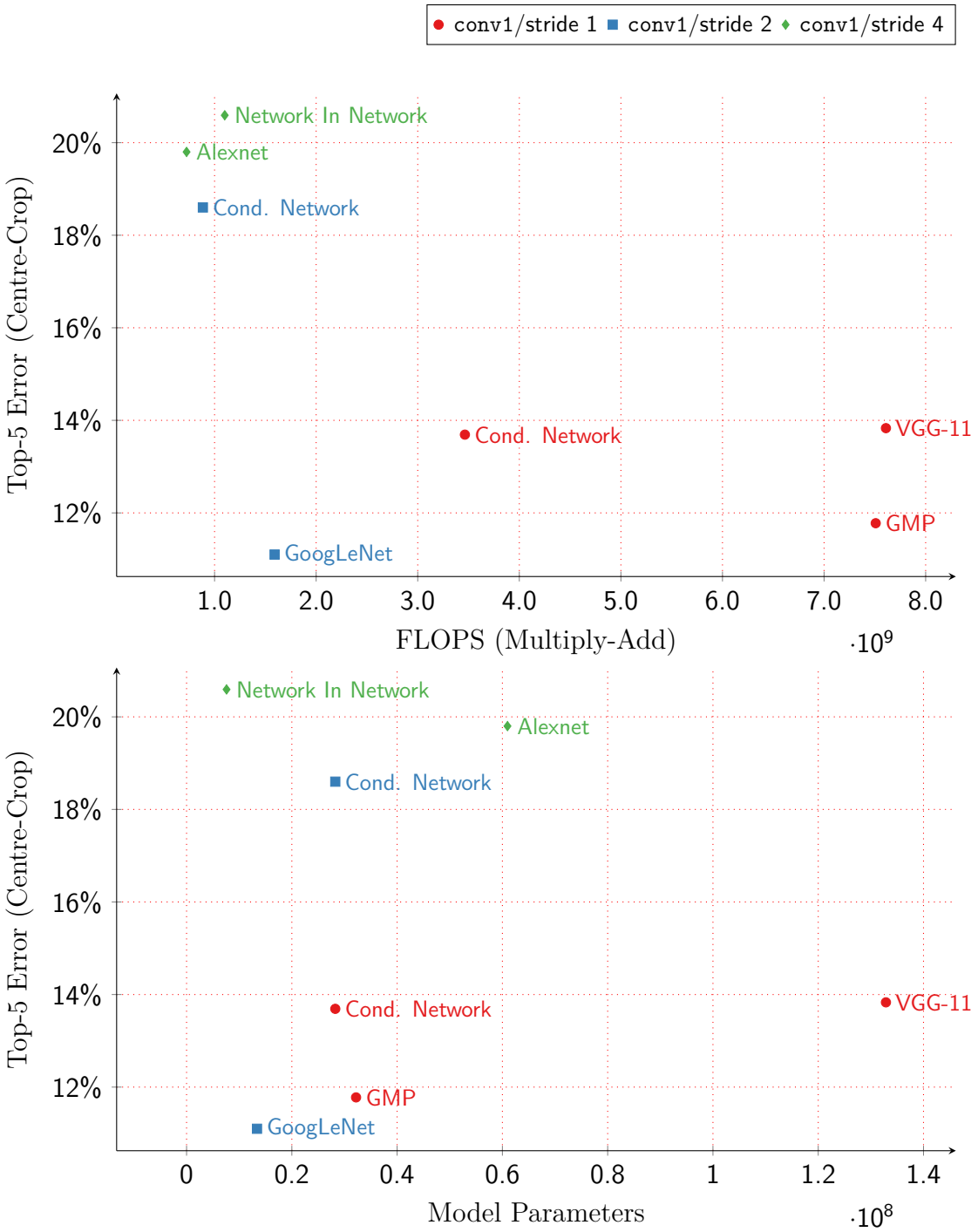


Fig. 6.12 **Efficiency of conditional networks on ILSVRC relative to state-of-the-art models.** Our VGG-11-GMP (GMP) reduces model size significantly, and is the baseline network. VGG-11 conditional networks (Cond. Network) yield points closer to the origin for networks with the same `conv1` stride, noted as (`conv1`/stride) in the legend.

Table 6.1 **Conditional network ILSVRC Results.** Accuracy, multiply-accumulate count, and number of parameters for the baseline networks and more efficient versions created by the methods described in this chapter. Results with test-time oversampling (OS) are also shown.

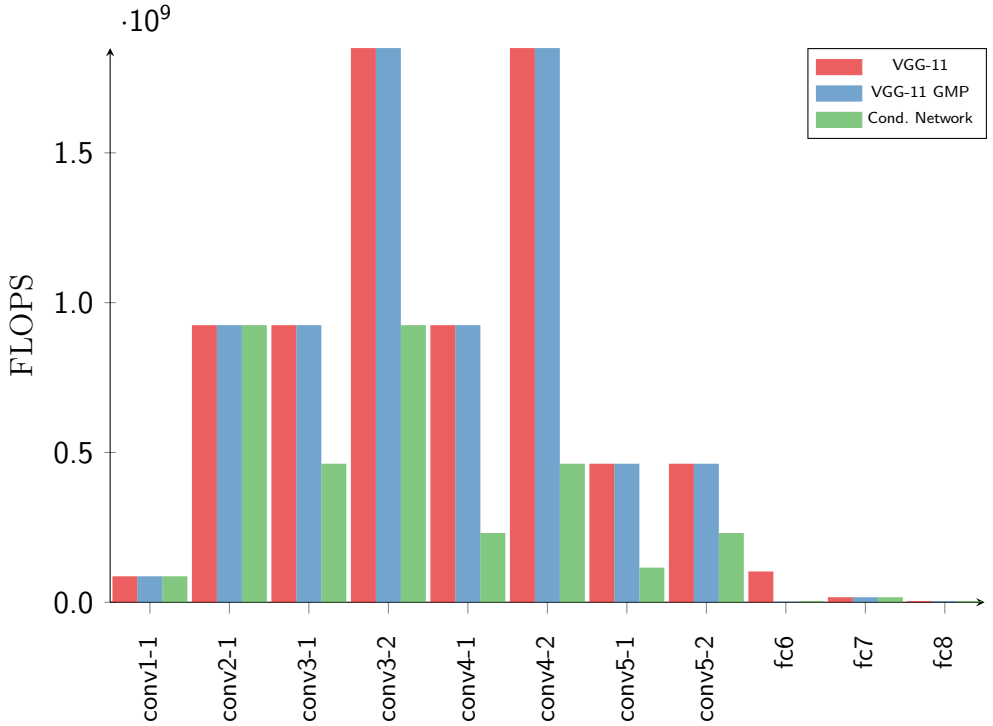
Network Name	Stride	Top-1 Err.	Top-1 Err. OS	Top-1 Err.	Top-5 Err. OS	FLOPS $\times 10^9$	Param. $\times 10^7$
VGG-11	1	0.351	0.328	0.138	0.124	7.61	13.29
GMP	1	0.325	0.309	0.118	0.108	7.51	3.22
Cond. Network	1	0.350	0.336	0.137	0.129	3.46	2.83
Cond. Network	2	0.421	1.000	0.186	1.000	0.88	2.83
GoogLeNet	2	0.313	0.298	0.111	0.101	1.59	1.34
Alexnet	4	0.427	0.414	0.198	0.188	0.72	6.10
Network In Network	4	0.437	0.424	0.206	0.200	1.10	0.76

The conditional network of fig. 6.11 corresponds to the green circle in fig. 6.12. It achieves a top-5, centre-crop error of 13.9% compared to 13.8% for the VGG-11 network, while requiring less than half of the computation (45%), and almost one-fifth (21%) of the parameters. Although it is the second closest to the origin (after GoogLeNet), we believe that better results can be achieved by using routes of different (learned) cardinality, as well as incorporating low-dimensional embedding and multiple-loss training. This is left for future work. Note that the most efficient model in (He *et al.*, 2015) uses 1.9×10^{10} flops which is just outside the plot. More accurate versions of (He *et al.*, 2015) are more expensive still. Finally, fig. 6.13 shows efficiency improvements achieved by our conditional network, layer by layer.

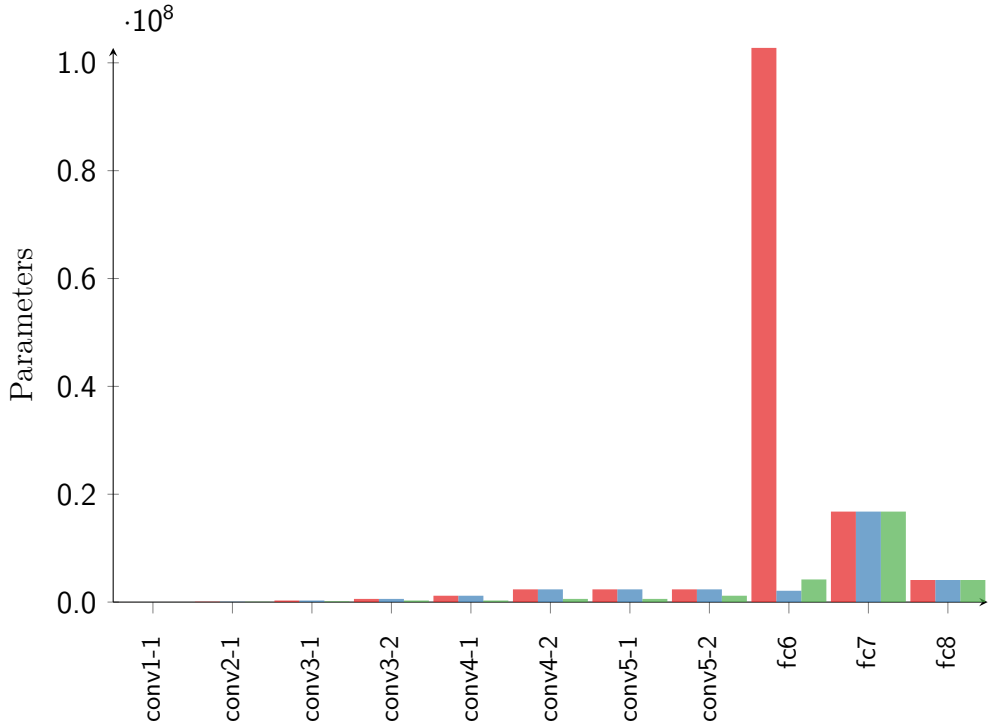
GoogLeNet divides filters within each Inception module into 4 groups of various filter sizes, giving the overall network a DAG-like structure. GoogLeNet's usage of low-dimensional embeddings within these routes reduces computatation and parameters of the network even further.

The most efficient of the state-of-the-art networks used for Imagenet classification, by a wide margin, is GoogLeNet (see fig. 6.12). NiN and GoogLeNet pioneered the use of semi-dense weight matrices in the form of learning dimensionality reductions, reducing the explosion of filters usually found in deep network architectures.

Another unique feature of these two networks is GAP, where the spatial dimensions of the last convolutional feature map are reduced to a single compact feature vector of length c, where c is the number of filters in the layer. This greatly reduces the parameters of the network by reducing the weights in the first fully-connected layer,



(a) # FLOPS for each layer



(b) # Parameters for each layer

Fig. 6.13 **Computational cost of VGG-11 based networks per layer.** Number of parameters and number of multiply-accumulate operations for all (convolutional and fully-connected) layers of VGG-11, VGG-11-GMP, and our conditional network. Our two networks reduce both the memory use and the computational cost.

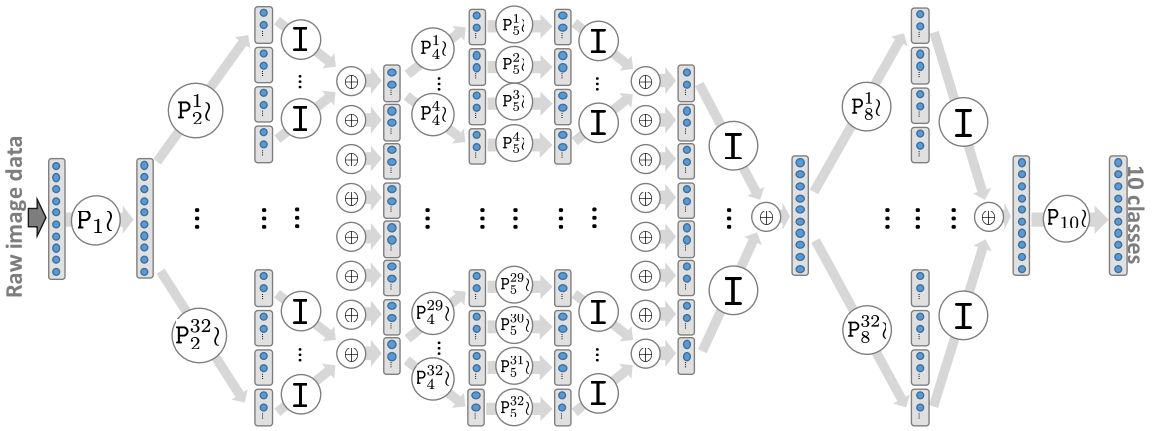


Fig. 6.14 **Automatically learned conditional architecture for image classification in CIFAR-10.** Both structure and parameters of this conditional network have been learned automatically via Bayesian optimization. ©Antonio Criminisi, used with permission.

which contains the majority of weights in the network (see fig. 6.13), and is the single largest factor in the space efficiency of NiN and GoogLeNet.

6.5.3 CIFAR-10

Here we validate conditional networks for the classification of images in the CIFAR-10 dataset (Krizhevsky, 2009). The dataset contains 60,000 images of 10 classes, typically divided into 50,000 training images and 10,000 test images. We take the state-of-the-art NiN model as a reference (Lin, Q. Chen, and Yan, 2014), and we build a conditional version of it to produce the architecture in fig. 6.14. Then we go on to show its increased efficiency compared to the original NiN model.

Designing a Family of Conditional Networks

The NiN model has a large number (192) of initial image-level filters in the first convolutional layer ('conv1'), representing a sizable amount of the overall computation.[6] We build a variant ('NiN-64') that prepends a layer of 64 filters to the NiN model. While this variant is more complex than NiN, when routed (as described later) it allows us to split the larger layers into many routes and increase the efficiency. In fact, for a convolutional layer with N groups of filters, each filter operates on $1/N$ channels of the input feature map, thus yielding reduced test-time computation and learning filters with fewer channels. Additionally, training routed convolutional layers means

[6] Most Imagenet networks typically use $64 - 96$ conv1 filters.

that filters are exposed only to the training subset of data that flows through that route, thus allowing for a potentially higher degree of specialization. By changing the number of routes at each level of the NiN-64 model (from `conv2`) we can generate a whole family of possible conditional architectures.

Learning the Optimal Network Architecture

In this experiment we have optimized the network structure *automatically*, by using Bayesian optimization (Snoek, Larochelle, and Adams, 2012), as available in WhetLab.[7] This has allowed us to search the large joint space of parameters and structures in a principled manner, and come up with multiple reasonable networks to be tested.

In the optimization we maximized the *size-normalized* accuracy $\alpha = \frac{\text{validation accuracy}}{\text{model size}}$ with respect to the parameters $R_l = 2^i, \{i \in \mathbb{N} : 0 \leqslant i \leqslant 5\}$, where R_l is the number of nodes at layer l in the conditional network. Figure 6.14 shows the architecture which maximizes α in CIFAR-10. It is a DAG-structured conditional network with 10 layers. To our knowledge this is the first attempt at learning automatically the architecture of a deep CNN for image classification.

Accuracy *vs.* efficiency

For comparison, we also reduce the complexity of the unrouted NiN-64 network by learning a reduction in the number of per-layer filters, *i.e.* we maximize α over $F_l = F_{\text{orig}}/2^i, \{i \in \mathbb{N} : 0 \leqslant i \leqslant 4\}$, where F_{orig} is the number of filters in layer l in NiN-64.

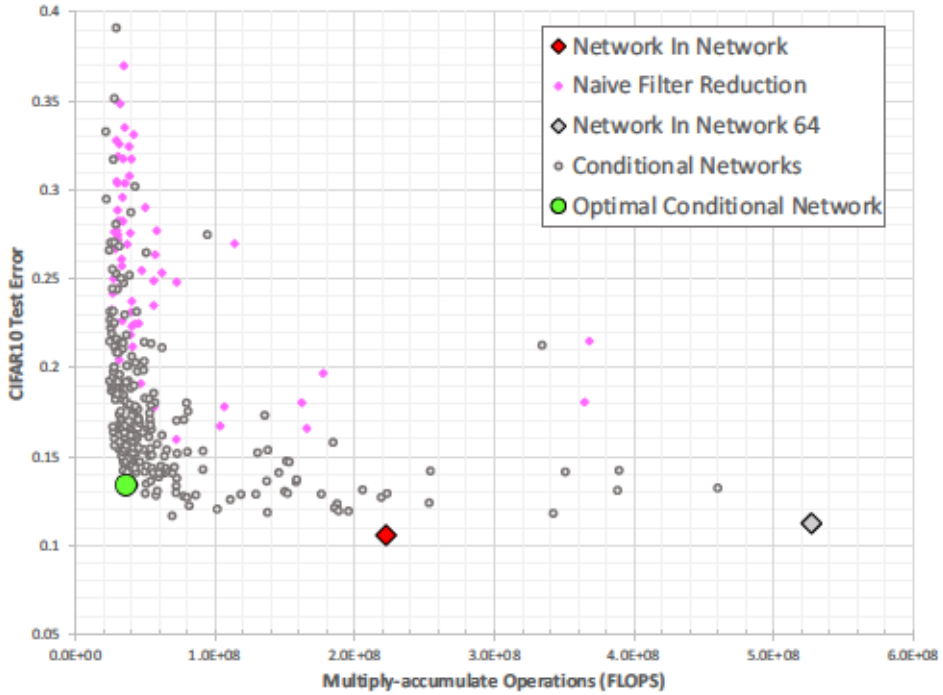
All networks were trained with the same hyperparameters as (Lin, Q. Chen, and Yan, 2014), except for using the initialization strategy of (He *et al.*, 2015), and a learning schedule,

$$\gamma_t = \gamma_0(1 + \gamma_0\lambda t)^{-1}, \tag{6.7}$$

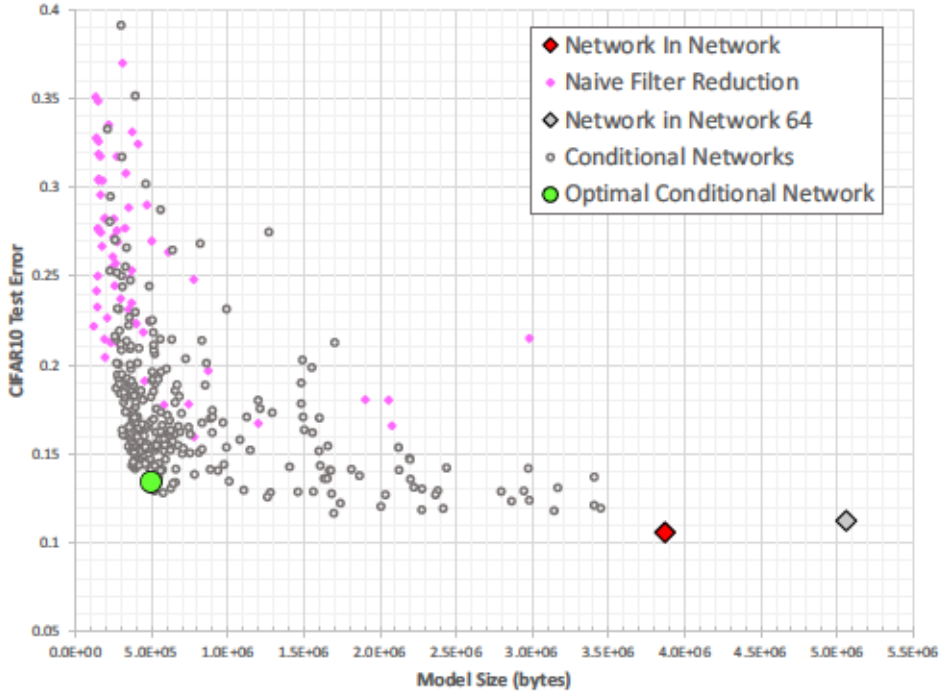
where γ_0, γ_t and λ are the initial learning rate, learning rate at iteration t, and weight decay respectively (Bottou, 2012). Training was run for a maximum of 400 epochs, or until the maximum validation accuracy had not changed in 10,000 iterations. We split the original CIFAR-10 training set into 40,000 training images and 10,000 validation images. The standard 10,000 held-out images are used for testing.

Figure 6.15 shows test errors with respect to test-time cost and model size for multiple networks. Diamonds denote unrouted networks and circles denote conditional networks. The original NiN is shown in red, and our NiN-64 variant is shown as a grey diamond. A sample of 300 models explored during the Bayesian optimization

[7]Formerly at https://www.whetlab.com, defunct since its acquisition by Twitter in June 2015.



(a) Test error *vs.* test-time computation for various architectures used in the CIFAR experiments.



(b) Test error *vs.* model size.

Fig. 6.15 **Comparing network architectures on CIFAR-10.** Conditional networks (denoted with circles) yield the points closest to the origin, corresponding to the best accuracy-efficiency trade-off. Our best conditional network is denoted with a green circle.

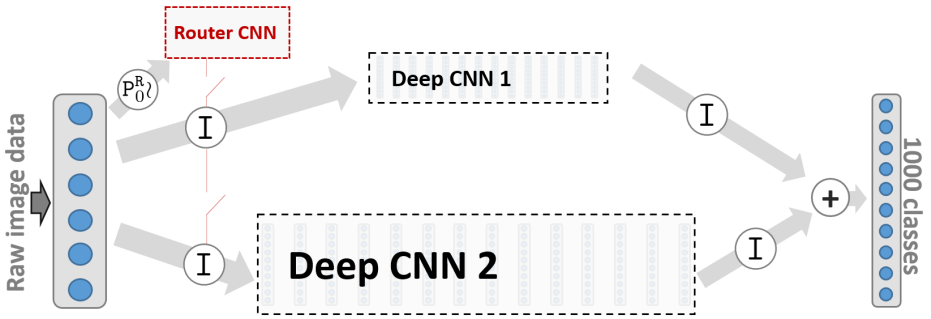


Fig. 6.16 **Explicit data routing for conditional ensembles.** An explicitly-routed conditional network that mixes existing deep CNNs in a learned, data-dependent fashion.

are shown as grey circles. The green circle denotes the conditional network closest to the origin of the 3D space (test-error, test-cost, model-size). Most of the conditional networks proposed by the Bayesian search procedure are distributed along a curve characterized by either high accuracy, or low model size, or both. Reducing the NiN model by filter reduction (pink diamonds in the figure) does not yield the same gains as data routing. Despite NiN achieving the highest accuracy (it has been optimized for accuracy alone), the optimal conditional network is much closer to the origin of the 3D space, thus indicating much higher efficiency (in terms of memory and computation) for a small loss in accuracy.

6.5.4 Conditional Ensembles of CNNs

A key difference between CNNs and conditional networks is that the latter may include (trainable) data routers. Here we use an *explicitly*-routed architecture to create an *ensemble* of CNNs where the data traverses only selected, component CNNs (and not necessarily all of them), thus saving computation.

As an example, the branched network in fig. 6.16 is applied to the ILSVRC2012 image classification task. The network has $R = 2$ routes, each of which is itself a deep CNN. Here, we use GoogLeNet (Szegedy, Liu, *et al.*, 2015) as the basis of each component route, although other architectures may be used. Generalizing to $R > 2$ is straightforward. The routes have different compute cost (denoted by different-sized rectangles), arising from differing degrees of test-time oversampling. We use no oversampling for the first route and 10X oversampling for the second route.

The router determines which image should be sent to which route (or both). The router is trained together with the rest of the network via back-propagation (section 6.4.7) to predict the accuracy of each route for each image. The router is

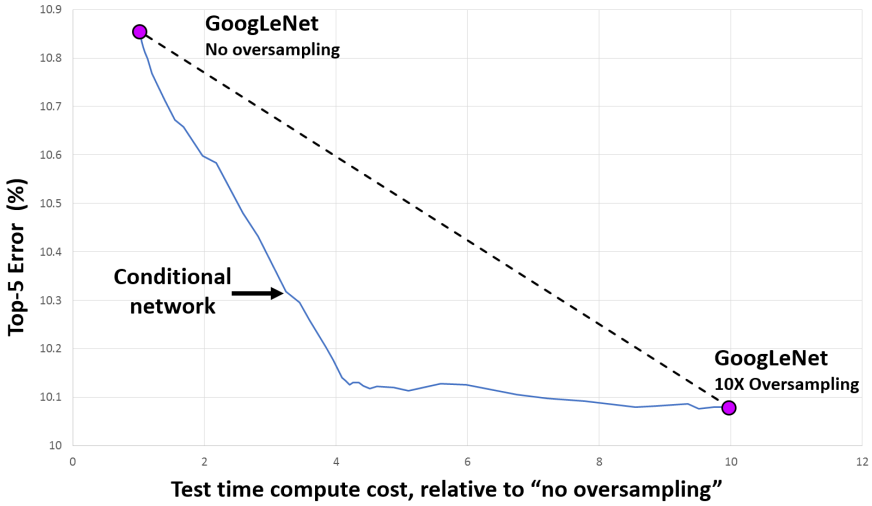


Fig. 6.17 **Error-accuracy results for conditional ensembles of CNNs.** Error-accuracy results for the two GoogLeNet base networks are shown in purple. The *dynamic* error-cost curve for our conditional ensemble is in green. In the green circle we achieve same accuracy as the most accurate GoogLeNet with half its cost.

itself a deep CNN, based on CNN1; This allows computation reuse for extra efficiency. At test time, a (dynamic) trade off can be made between predicted accuracy and computational cost.

Figure 6.17 shows the resulting error-cost curve. All costs, including the cost of applying the router are taken into consideration here. Given our trained conditional network, we use dynamic, multi-way data routing (section 6.5.1) to generate a curve in the error-compute space. Each point on the curve shows the top-5 error on the validation set at a given compute cost, which is an amortized average over the validation set.

The dashed line corresponds to the trivial error vs. compute trade-off that could be made by selecting one or other base network at random, with a probability chosen so as to achieve a required average compute cost. The fact that the green curve lies significantly below this straight line confirms the much improved trade-off achieved by the conditional network. In the operating point indicated by the green circle we achieve nearly the same accuracy as the 10× oversampled GoogLeNet with less than half its compute cost. A conventional CNN ensemble would incur a higher cost since all routes are used for all images.

6.6 Discussion

This chapter has investigated the similarities and differences between decision trees/-forests and DNN. This has led us to introduce a hybrid model (namely conditional networks) which can be thought both as: (**i**) trees which have been augmented with representation learning capabilities, and (**ii**) CNNs which have been augmented with explicit data routers and a rich, branched architecture. Experiments on image classification have shown that highly branched architectures yield improved accuracy-efficiency trade-off as compared to trees or CNNs. The desired accuracy-efficiency ratio can be selected *at run time*, without the need to train a new network.

7

Conclusion and Future Work

In this dissertation, we have proposed that carefully designing networks in consideration of our prior knowledge of the task can improve the memory and computational efficiency of state-of-the art networks, and even increase accuracy through structurally induced regularization. While this philosophy defines our approach, deep neural networks have a large number of degrees of freedom, and there are many facets of deep neural networks that warrant such analysis. We have attempted to present each of these in isolation:

Chapter 4 proposed to exploit our knowledge of the low-rank nature of most filters learned for natural images by structuring a deep network to learn a collection of mostly small 1×h and w×1 basis filters, while only learning a few full w×h filters. Our results showed similar or higher accuracy than conventional CNNs requiring much less computation. Applying our method to an improved version of VGG-11 network using GMP, we achieve comparable validation accuracy using 41% less computation and only 24% of the original VGG-11 model parameters; another variant of our method gives a 1 percentage point *increase* in accuracy over our improved VGG-11 model, giving a top-5 *center-crop* validation accuracy of 89.7% while reducing computation by 16% relative to the original VGG-11 model. Applying our method to the GoogLeNet architecture for ILSVRC, we achieved comparable accuracy with 26% less computation and 41% fewer model parameters. Applying our method to a near state-of-the-art network for CIFAR-10, we achieved comparable accuracy with 46% less computation and 55% fewer parameters.

Chapter 5 addresses the filter/channel extents of convolutional filters, by learning filters with limited channel extents. When followed by a 1×1 convolution, these can also be interpreted as learning a set of basis filters, but in the channel extents. Unlike in chapter 4, the size of these channel-wise basis filters increased with the depth of the model, giving a novel sparse connection structure that resembles a tree root. This

allows a significant reduction in computational cost and number of parameters of state-of-the-art deep CNNs without compromising accuracy. Our results showed similar or higher accuracy than the baseline architectures with much less computation, as measured by CPU and GPU timings. For example, for ResNet 50, our model has 40% fewer parameters, 45% fewer floating point operations, and is 31% (12%) faster on a CPU (GPU). For the deeper ResNet 200 our model has 25% fewer floating point operations and 44% fewer parameters, while maintaining state-of-the-art accuracy. For GoogLeNet, our model has 7% fewer parameters and is 21% (16%) faster on a CPU (GPU).

Chapters 4 and 5 proposed similar methods for reducing the computation and number of parameters in the spatial and channel (filter-wise) extents of convolutional filters respectively. Rather than approximating filters in previously-trained networks with more efficient versions, we learn a set of smaller basis filters from scratch; during training, the network learns to combine these basis filters into more complex filters that are discriminative for image classification. This means that at both training and test time our models are more efficient. Overall, the approach of learning a set of basis filters was not only effective for reducing both computation and model complexity (parameters), but in many of the results in both chapters 4 and 5, the models trained with this approach generalized better than the original state-of-the-art models they were based on.

Chapter 6 presented work towards conditional computation in deep neural networks. We proposed a new discriminative learning model, *conditional networks*, that jointly exploits the accurate *representation learning* capabilities of deep neural networks with the efficient *conditional computation* of decision trees and directed acyclic graphs (DAGs). In addition to allowing for faster inference, conditional networks yield smaller models, and offer test-time flexibility in the trade-off of computation *vs.* accuracy.

7.1 Future Work

Research outcomes are often better evaluated by the questions borne rather than the questions answered. In this section we'll address the main research questions that this dissertation has highlighted, and propose future directions for research which we believe would have the most impact on the field.

7.1.1 Learning Structural Priors

The move towards "end-to-end" learning has made great strides in making learning more automatic, notably in learning complex representations rather than experts designing inferior representations themselves. There still exists however, a significant amount of hand design and manual tuning that is key to the success of any deep learning approach. We hope our work will motivate the field towards a research direction that aims to minimize this further, by working on methods of automatically structuring neural networks, in a move towards a truly "end-to-end" learning of DNN structure itself.

The lack of understanding or concrete rules for structuring DNNs means that in practical applications deep learning is often restricted to experts in the field, who have an intuition in network design formed from years of experience, and know which structural priors to use. The effect on deep learning research is no less profound, with a lack of understanding of the basic interplay between structure and learning in DNNs, we have little chance of understanding the limitations of deep learning or the representations learned by the networks.

The benefits of automatically structuring DNNs go further than these considerations even, as the research presented in this dissertation has shown, better structured DNNs are more computationally efficient (use fewer parameters and are faster to compute), and generalize better. Currently, training state-of-the-art DNNs for image classification requires a prohibitive amount of time and computational resources — 3 weeks of training on 8 high-end and expensive GPUs — and yet we know that trained DNNs are very sparse representations and have been shown to be highly compressible. It is because we cannot appropriately understand this sparse structure well enough to fully exploit it that our current DNNs are so inefficient.

With automatic methods of learning the structure, DNNs will become markedly more efficient to train, leading to faster experimental results for research, and also allow easier deployment to embedded devices, such as mobile phones, drones and robots. It would also allow for research strides in learning networks for multiple modalities, for example a self-driving car needs to process input data from normal camera sensors, along with depth maps or point clouds, and even radar. One of the stumbling blocks in doing this is understanding how to best structure a network to deal with multiple inputs which require different structural priors.

Research on finding automatic methods of structuring neural networks is not a completely new avenue of research, with a substantial effort put towards it 30 years ago when neural networks, and datasets, were much smaller. This is covered in chapter 3,

but suffice to state that there were two main approaches: (**i**) greedily building networks from scratch, and (**ii**) pruning (removing parameters) large networks. The proposals made for both building networks from scratch , such as that of Fahlman and Lebiere (1989), and pruning full networks, such as that of LeCun, J. S. Denker, and Solla (1989), suffer drawbacks which make them unsuitable in the modern deep network of hundreds of millions of parameters. Even in neural networks of contemporary size, the greedy approach of Fahlman and Lebiere (1989) meant that learned networks were suboptimal. This proposal should also not be confused with 'universal learning', or violating the no free lunch theorem (section 3.2.5), since we are interested in learning methods for the specific set of problems we as humans are interested in solving, rather than all possible input patterns.

At least three factors prevented this line of research from being successful historically, that we believe have now been overcome. Recent breakthroughs in training DNNs have given us a better understanding of how to train very large, arbitrarily structured networks, notably avoiding the so-called 'vanishing gradient' (He *et al.*, 2016b; Ioffe and Szegedy, 2015), and a better understanding of initialization (He *et al.*, 2015). Extremely large and diverse datasets are now prevalent, such as ImageNet (Russakovsky *et al.*, 2015), whereas historically datasets were prohibitively small to be useful for automatically structuring DNNs. And finally computational resources have increased dramatically. In fact these are most of the reasons the field of deep learning itself has been more successful now than neural networks were 30 years ago.

7.1.2 Jointly Learning a Basis for Spatial and Channel Extents of Filters

In the shorter term, there is an obvious question arising from the work presented in chapters 4 and 5 that explore learning more efficiently by reducing the learned parameters in the spatial and channel extents of convolutional filters respectively. These naturally lend themselves to being merged into a single effective method for training with low-rank basis filters. We plan to submit a journal article in which both methods are merged and explored in new results on state-of-the-art DNNs.

7.1.3 Optimization and Structural Priors

It is notable that many structural priors can be viewed as enforcing sparsity on fully-connected networks. For example, in the case of a CNN, any learned CNN is representable in a fully-connected network, since a CNN can be viewed as a fully-

connected network with a specific arrangement of zeroed connect weights, and some duplicated weights (shared weights) as illustrated in fig. 4.1.

The question arises then, why can we not learn these in fully-connected networks? Structural priors give lower training loss, and yet when we optimize fully-connected networks with an appropriate structure and capacity to learn the sparse structural priors, they do not. Another, more recent example, is that of ResNets, as explained in section 2.3.5, these are motivated by the observation that in very deep networks the optimization fails to learn even the identity function, when it can be shown to give a lower loss.

In many ways the need for structural priors can be seen as the result of a problem with the current methods of optimization of DNNs. As discussed in section 2.1.6, higher order optimization might help solve this, but is not practical given the size of contemporary DNNs.

7.1.4 Parting Note

In my PhD, I have focused on experiments which I believed would shed light on the representations being learned in DNNs. Although the overt motivation of much of the work in its publication has been efficiency, my personal motivation has always been to better understand the learned internal representation of state-of-the-art DNNs for image classification, and explain why they are so over-parameterized. Structural priors, such as those demonstrated in this dissertation do not only improve the effectiveness of a deep network, but are *necessary* for good generalization.

Appendix A

Bibliographic Epilogue

Research in deep learning has taken on a new rapidity since the adoption of pre-prints, and the explosion of interest in the field. Rather than being bound to the annual conference schedule, new research is released on a weekly basis. Presenting a paper at a contemporary conference, one is now in the odd situation of having to relate the 'new' research being presented to the 6–12 months of follow-up research in the field. Compare this to even a few years ago, when publication of new research was withheld until a conference paper acceptance, perhaps a couple of months before the conference.

This dissertation represents the ultimate presentation of the research we have undertaken and, just as in a conference, it must also be presented in the context and timeline of the follow-up research and applications that it has already inspired.

In this section, we will briefly outline the significant derivative papers published after the original pre-print publication of the research we have presented here, along with their pre-print dates. We also present new research, published after ours that has extended the field towards learning structural priors automatically.

A.1 Pre-print Publication Dates

For reference, the initial public release of the papers behind the work presented in this dissertation are outlined below:

- Decision Forests, Convolutional Networks and the Models in-Between (Ioannou, Robertson, Zikic, *et al.*, 2015)

  MSR internal technical report Apr. 2015

  Pre-print 3 Mar. 2016

- Training CNNs with Low-Rank Filters for Efficient Image Classification (Ioannou, Robertson, Shotton, *et al.*, 2016)

  Pre-print arXiv:1511.06744 (30 Nov. 2015)

  Peer-reviewed publication date May 2, 2016

- Deep Roots: Improving CNN Efficiency with Hierarchical Filter Groups (Ioannou, Robertson, Cipolla, *et al.*, 2017)

  Pre-print arXiv:1605.06489 (20 May 2016)

  Peer-reviewed publication date CVPR 2017 (21 July, 2017)

A.2 Recent Research Related to Chapter 4

The method presented in this chapter directly applied to many deep learning vision problems, and we have heard directly from developers of it's use in embedded devices, in applications as diverse as some of the most popular mobile phones to autonomous drones.

Unfortunately even citing authors assume the paper present a low-rank approximation of the filters, despite our explicit and frequent statements that our models are trained from scratch and not approximated. This is likely due to the title and its similarity to the titles of a large amount of literature proceeding presenting methods of approximation.

Rethinking the Inception Architecture for Computer Vision

Pre-print arxiv:1512.00567 (2 Dec. 2015)

Peer-reviewed publication date CVPR 2016 (28 June, 2017)

Szegedy, Vanhoucke, *et al.* (2016) published an update to the Inception architecture around 3 weeks after our pre-print publication, making it likely that their work was independent. Nevertheless, the method they present is identical to our proposal (Ioannou, Robertson, Shotton, *et al.*, 2016) — the training of the Inception architecture with low-rank (*i.e.* 1×3 and 3×1) filters to reduce computation and improve generalization. They call these 'factorized filters' which we disagree with, since being simply concatenated they are not a factorization (*i.e.* separation of a multiplication), rather we argue they are linearly combined, and so represent a basis.

The method proposed in this chapter (and identical to our proposed method) forms the basis of all current Inception architectures, and are now used in all Google deep learning backed products, for example Google Photos™.

A.3 Recent Research Related to Chapter 5

Unfortunately, likely due to the prominence of Google and Facebook in the research community, the Xception and ResNeXt papers below have received most of the citations and credit for root units in recent work, despite the pre-prints coming five and six months after our pre-print publication, respectively. On the other hand, this exposure has brought a lot of attention to the method, making the use of filter groups, and root modules specifically, for efficiency and generalization increasingly common.

Xception: Deep Learning with Depthwise Separable Convolutions

Pre-print arXiv:1610.02357 (7 Oct. 2016)

Peer-reviewed publication date CVPR 2017 (21 July, 2017)

The author proposes the 'Xception' module, which is a special case of our proposed root modules (Ioannou, Robertson, Cipolla, *et al.*, 2017), where, if c_1 is the number of input feature map channels, and g is the number of groups (as in fig. 5.4), an 'Xception' module is the extreme case where $g \equiv c_1$. In our experiments, this level of sparsity is clearly adverse. In the 'Xception' experiments the number of filters is greatly increased as compared to the original architecture compensating for this. However, by using convolutional groups of the same size as input channels, any of the computational advantages of root modules are lost, for all the reasons described in section 5.4. We should note that in personal correspondence with the author he has denied any relationship between this work and ours, and so does not cite our work.

Aggregated Residual Transformations for Deep Neural Networks

Pre-print arXiv:1611.05431 (16 Nov. 2016)

Peer-reviewed publication date CVPR 2017 (21 July, 2017)

The aggregated residual units proposed by Xie *et al.* (2017) are technically identical to our root modules as implemented in ResNet (the paper cites our work), however they explore a different compute-generalization trade-off with their ResNeXt architecture.

Rather than using the more efficient representation learned by the root units to save parameters and computation as in our work, the authors propose to maintain the original computational footprint of the model, and instead increase the number of filters learned. The result is a much improved network, so much so that their model won second place in the 2016 ILSVRC competition.

Interestingly, the authors claim that root units are even more effective on much larger datasets, and more effective than further increasing depth or width of the network. Xie *et al.* (2017) state that "...increasing cardinality is more effective than going deeper or wider when we increase the capacity.", where they denote the number of filter groups used as cardinality.

ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices

Pre-print arXiv:1707.01083 (4 Jul. 2017)

X. Zhang *et al.* (2017)

Interleaved Group Convolutions for Deep Neural Networks

Pre-print arXiv:1707.02725 (7 Oct. 2016)

Peer-reviewed publication date October 22, 2017

T. Zhang *et al.* (2017)

The Power of Sparsity in Convolutional Neural Networks

Pre-print arXiv:1702.06257 (21 Feb. 2017)

Changpinyo, Sandler, and Zhmoginov (2017)

Convolution with Logarithmic Filter Groups for Efficient Shallow CNN

Pre-print arXiv:1707.09855 (31 Jul. 2017)

Lee *et al.* (2017)

A.4 Recent Research Related to Chapter 6

While being a very interesting topic, little has been published in relation to this paper since its pre-print release, aside from the work of E. Bengio *et al.* (2015) who used a reinforcement learning framework to add conditional computation to a DNN.

Conditional Computation in Neural Networks for Faster Models

Pre-print arXiv:1511.06297 (19 Nov. 2015)

E. Bengio *et al.* (2015)

A.5 Automatically Learning Network Architectures

Since the papers this dissertation is based on were published, the research community has been moving towards learning to create neural network architectures. In particular, the following papers have shown significant progress. Baker *et al.* (2017) and Zoph and Le (2017) both presented methods of learning neural network architectures using reinforcement learning.

Appendix B

Co-adaption in Deep Neural Networks

As discussed in section 7.1.3, there is a link between structural priors and a failure in our current optimization methods. Dropout in particular seems like it may have a link to structural priors given that it is claimed to be randomly sampling 'thinner' network architectures composed of subsets of the neurons from the model. It can also be argued, as we will demonstrate, that dropout is an optimization trick rather than a form of regularization. Here we will present work that, while not being substantial enough for publication, may provide an interesting insight into the problem of 'co-adaption' as discussed by Geoffrey E. Hinton, Srivastava, *et al.* (2012b) and the mechanism of dropout.

B.1 The Limitations of First Order Optimization

With the increasing number of practical applications of deep learning, the optimization of deep neural networks is of critical importance, with generalization, accuracy and training time all direct consequences. Due to practical considerations of training large state-of-the-art models with limited computational resources, network optimization is restricted to first order methods in practice – typically stochastic gradient descent with momentum. With breakthroughs in initialization (Glorot and Y. Bengio, 2010; He *et al.*, 2015) and maintenance of numerical precision during training (Ioffe and Szegedy, 2015) alleviating the 'vanishing gradient' problem, such methods have surpassed human accuracy on large scale image recognition datasets, amongst other breakthrough results. This success has overshadowed any weakness of the current methods of training deep networks.

There has been much evidence to suggest that the optimization of deep networks remains a concern however. Ba and Caruana (2014) showed that shallow networks could be regressed from deep networks, and claim that the success of deep networks could be explained by our inability to properly train shallow networks from scratch. More recently, He *et al.* (2016a,b) in particular have shown that the optimization of very deep networks can expose fundamental optimization issues where training loss *increases* for deeper networks, whereas even the trivial solution to maintain training loss — the identity mapping — is not discovered by the optimization. They suggest a work-around for this problem is to utilize residual layers, incorporating the identity explicitly. Why the optimization fails so spectacularly without identity connections remains unexplained however.

Martens (2010) suggests that 'pathological curvature (see section 2.1.6) is a possible explanation for the difficulty of training deep networks. For some networks, the error surface can have a complex curvature and the solution is to use a second order optimization, proposing a more efficient method, 'Hessian-free' optimization.

In this chapter we will demonstrate that some of the contemporary issues in training deep networks may be explained as being caused by this pathological curvature in the high dimensional error surfaces, and our use of first-order optimization methods. We will show that in particular dropout, which has empirically been shown to improve generalization in deep networks can also be explained in this light, and why this improvement diminishes in the presence of batch normalization, as observed by Ioffe and Szegedy (2015).

B.2 Co-adpation of Hidden Units in Deep Networks

Geoffrey E. Hinton, Srivastava, *et al.* (2012a) and Srivastava *et al.* (2014) proposed dropout as a regularization method for neural networks. In randomly dropping out hidden units — zeroing out a random subset on each layer — it was claimed that complex co-adaptions of these hidden units on training data, which do not generalize to the test set, are prevented. In practice dropout has seen remarkable success in improving the generalization of large neural networks, especially in the context of large fully-connected layers. Several follow-up methods have similarly suggested alternative methods of preventing this co-adaption (Cogswell *et al.*, 2016).

Although empirically dropout works well, the claim that hidden units learn to co-adapt has not itself been well demonstrated, and seemingly straightforward methods of doing have serious drawbacks. Showing the covariance/correlation between pairs of

hidden units doesn't give the full story, since covariance shows the linear relationship between the units, but in a deep network this relationship is likely to be highly non-linear. Mutual information cannot be used either since most modern networks need to use unbounded activation functions, such as ReLUs, in order to train effectively.

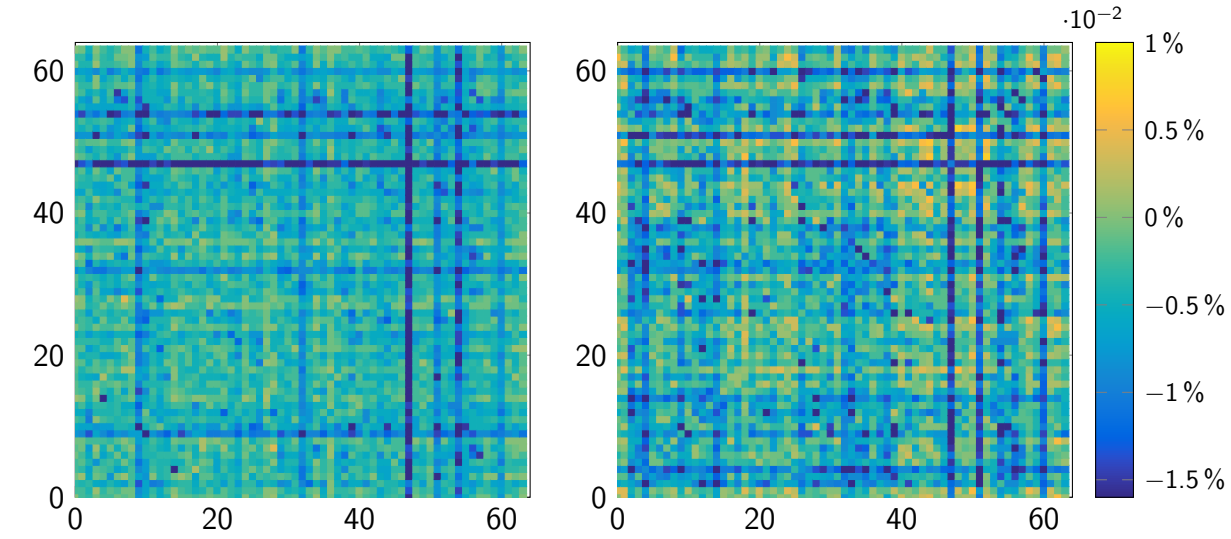
A crude but effective method of analyzing the importance of hidden units in neural networks is ablation – zeroing out certain units in a trained network of interest, and observing the effect on the training and test loss. This method can be extended to empirically evaluate the importance of pairs of hidden units in trained neural networks. For each pair of hidden units, zero out the parameters of both units, and observe the effect on the loss calculated over the training/test set: pairwise ablation. This is however very expensive since each pair of hidden units must be evaluated over the entire dataset.

We will focus on layer-wise filter co-dependence, and thus only needed to evaluate the pairwise ablation within each layer. In addition for large datasets, such as ILSVRC2012 (Russakovsky *et al.*, 2015), a random subset of the dataset was evaluated.

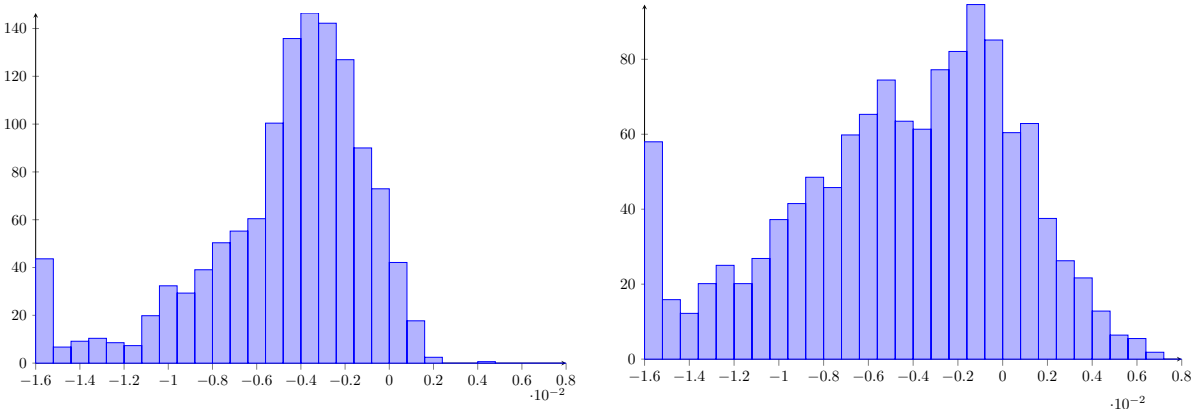
ResNet-50 Figure B.1 shows the results of pairwise ablation on a ResNet-50 He *et al.* (2016a) network trained on ILSVRC2012, on both the training set (fig. B.1(c)) and validation set (fig. B.1(d)).

While, as expected, most pairwise ablations result in a decrease in accuracy, a small but significant number of pairwise ablations result in an *increase* in accuracy (and decrease in loss). For the validation set this seems to provide clear evidence of hidden units co-adapting, and hence overfitting to the training data. Surprisingly however, this effect is also evident when evaluating on the training set, and by definition cannot be explained by overfitting. We observed similar effects on all other layers of the network.

MNIST This effect is not limited to large state-of-the-art deep networks, or indeed even deep networks. Surprisingly this effect is reproducible with a minimal single hidden-layer MNIST network. With a fully-connected network with one hidden layer, ReLU activation functions, and a variety of optimization methods, we find that this co-adaption is still present above a minimal number of hidden units. Figure B.2 shows the minimum and maximum increase in training loss/accuracy and test loss/accuracy for the MNIST network with different numbers of hidden units, when pairwise filters are ablated, as evaluated on the entire MNIST training/test sets. The effect of training with weight decay dropout and momentum are also compared with vanilla SGD. Weight decay and dropout are considered to be regularization methods, while momentum

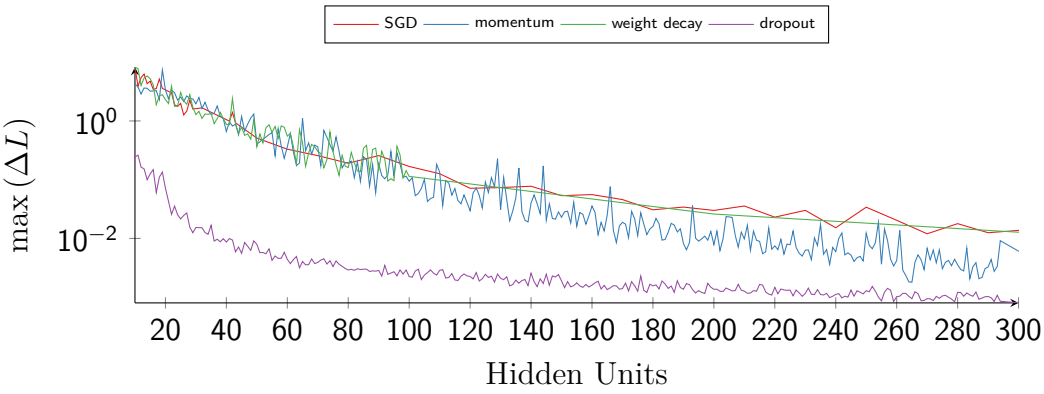


(a) Top-5 training acc. diff. for filter pairs (b) Top-5 validation acc. diff. for filter pairs

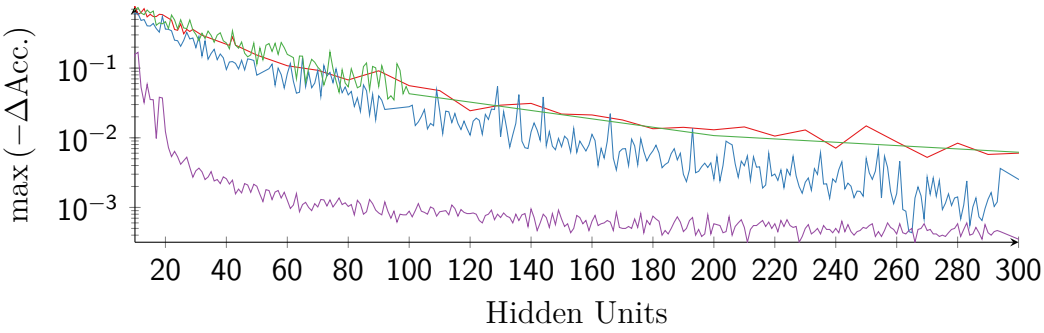


(c) Histogram of change in top-5 train. acc. (d) Histogram of change in top-5 val. acc.

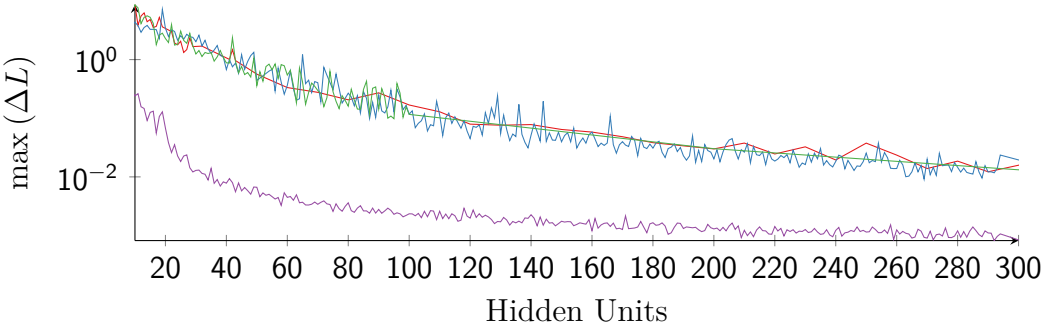
Fig. B.1 Histogram of the change in top-5 accuracy for all pairwise filter ablations of a 2500 randomly sampled images from the ILSVRC training/validation set in `conv1` of ResNet-50.



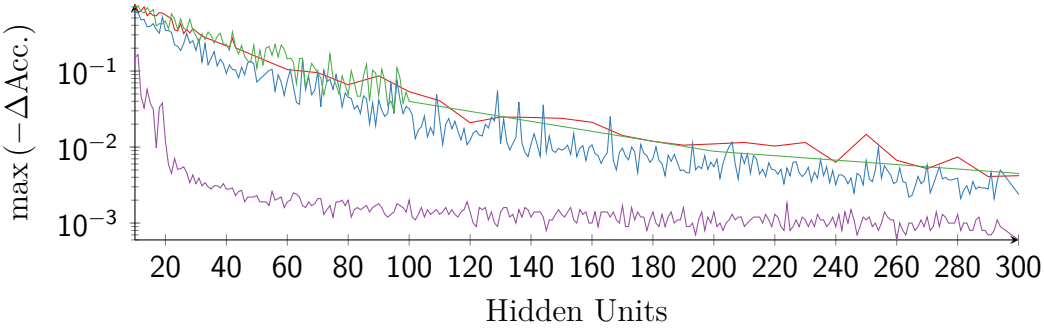
(a) Training loss change: maximum increase



(b) Training accuracy: maximum decrease

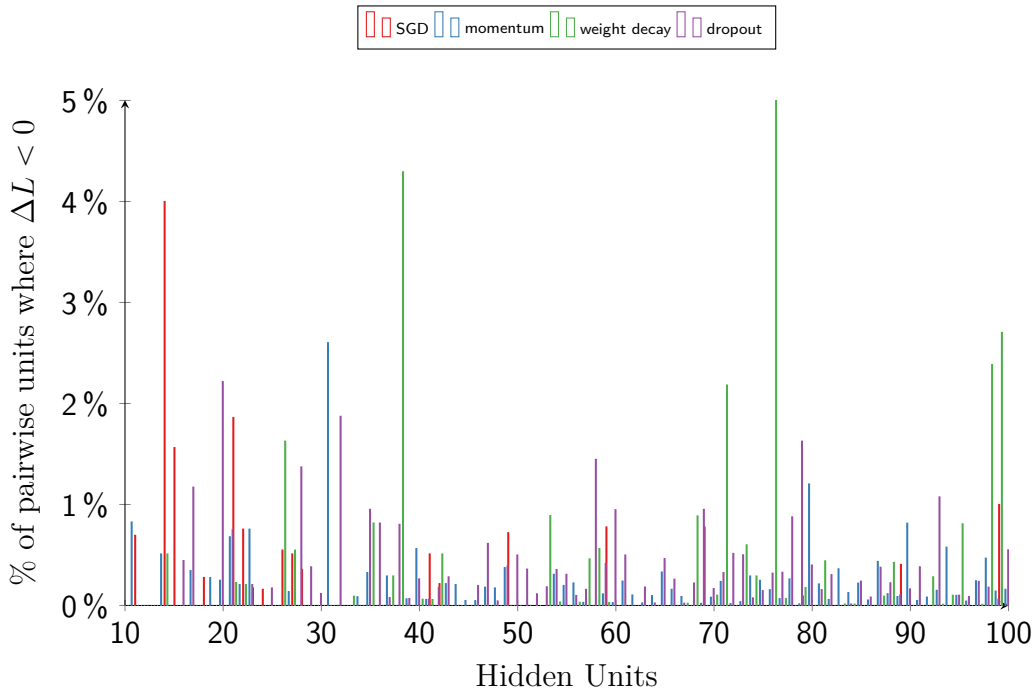


(c) Test loss: maximum increase

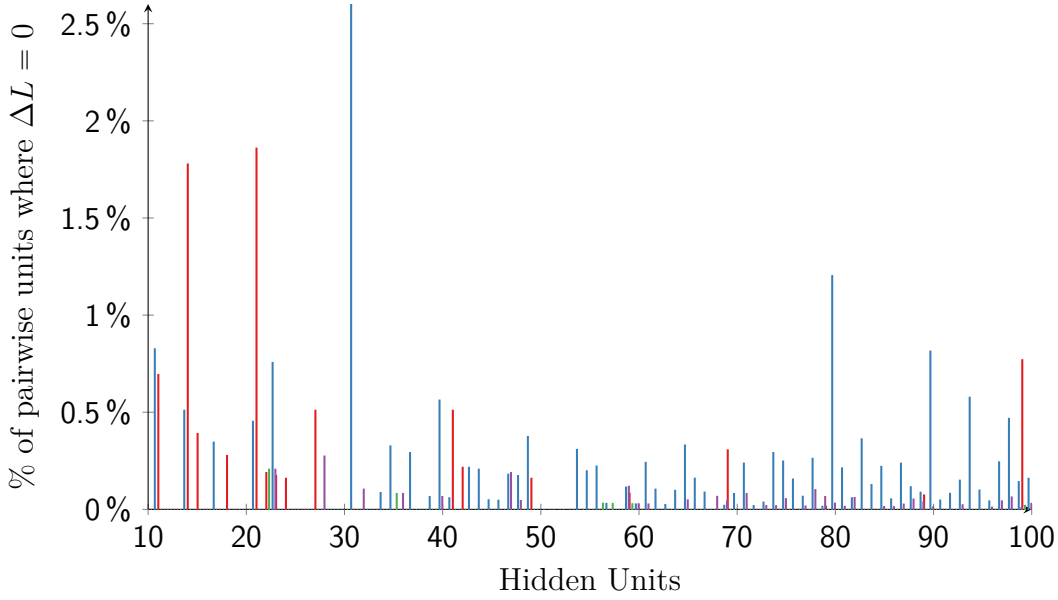


(d) Test accuracy: maximum decrease

Fig. B.2 Maximum increase and decrease in training/test loss/accuracy for a single-layer hidden MNIST classification network under pairwise ablation of the hidden units. The plots show the maximum increase in loss or decrease in accuracy. Both are measures of the level of neural co-adpation in the trained networks.



(a) Training Loss: Percentage of pairs of hidden units that exhibit adverse co-adaption.



(b) Training Loss: Percentage of pairs of hidden units that are independent.

Fig. B.3 Number of pairs of hidden units in a single-layer hidden MNIST classification network which under ablation, are adversely dependent ($\Delta L < 0$) or independent ($\Delta L = 0$) for the training set.

is an optimization trick that is intended to help avoid some issues of using a first order optimization — it can help speed up learning in the presence of some types of pathological curvature that would otherwise lead to slow optimization or a poor local minima with vanilla SGD.

In fig. B.2 momentum clearly helps avoid co-adaption as compared to vanilla SGD, where co-adaption is significant. Weight decay on the other hand, does not seem to have a helpful effect on co-adaption, giving almost identical results to vanilla SGD. As a form of regularization, this might be expected, as it should help generalization, not training fit.

Taken by itself, this observation suggests that co-adaption may be a symptom of an optimization problem, and thus an optimization trick like momentum helps while regularization does not. On the other hand, dropout seems to reduce the number of co-adapted units significantly, and is even effective at reducing co-adaption at *training time*. If dropout is a regularization method, then this seems to conflict with our findings with weight decay and momentum.

B.3 Dropout as an Optimization Trick

Dropout can also be thought of as a orthogonal projection of the error surface onto a random lower-dimensional subspace, in which the curvature of the error surface may no longer exhibit pathological issues, and optimization may be easier. For example, if in a subset of the dimensions, a deep valley exists (as illustrated in fig. 2.8), and these dimensions are dropped-out, first-order optimization will be substantially easier. Random projection is a well established method for dimensionality reduction of high dimensional spaces (Fodor, 2002; Kaski, 1998). If a layer has N nodes, and a width matrix W, and input vector x, then dropout on the layer of K/N nodes may be defined as the transformation:

$$\text{dropout}(\mathbf{W}) = \mathbf{D}_i\mathbf{W}, \tag{B.1}$$

where D is a diagonal binary matrix, with rank K, defining an orthogonal projection onto a K-dimensional subspace.

To demonstrate that it is this projection, rather than the zeroing out of neurons itself, that is responsible for performance improvements with dropout, we can instead perform a random projection in a different orthogonal co-ordinate basis, which does not dropout (zero) any neurons. To do this we can first rotate the parameters with random rotation matrix into a non-axis aligned co-ordinate basis, and perform dropout (orthogonal projection) in the rotated space, and then rotate back into the original

co-ordinate basis:

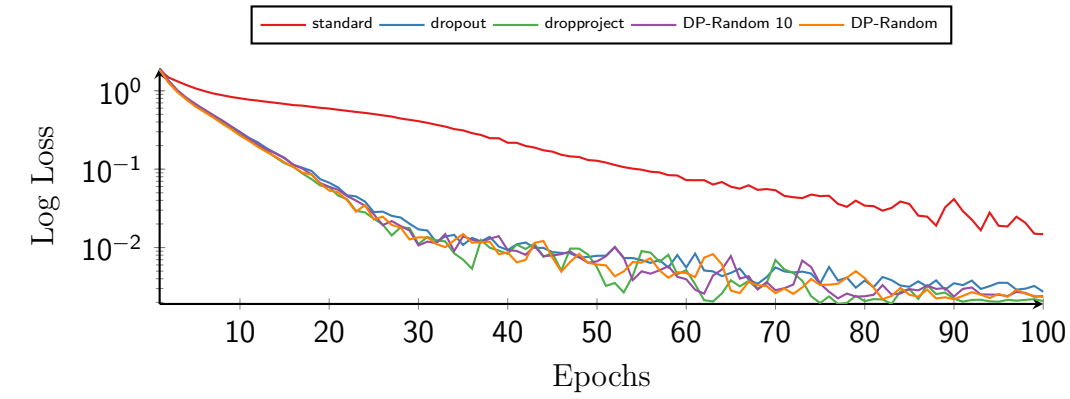
$$\text{dropproject}(\mathbf{W}_l) = \mathbf{R}^{-1}\text{dropout}(\mathbf{R}\mathbf{W}_l), \tag{B.2}$$

for a random rotation matrix $\mathbf{R} \in \text{SO}_N$. This "dropproject" method avoids zeroing out any of the units, while still performing an equivalent projection as that in dropout.

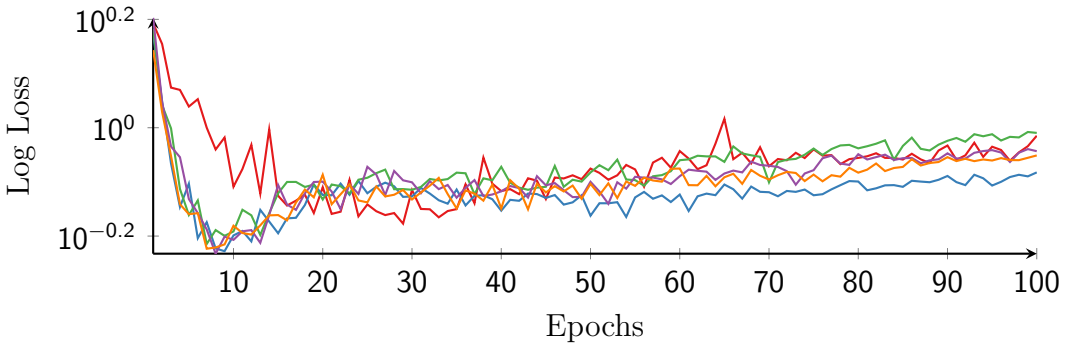
Figure B.4 compares the effect of dropout, and various forms of dropproject, on a large VGG model trained on the CIFAR-10 (Krizhevsky, 2009) dataset. Both dropout and dropproject are only applied to the two large fully-connected layers of the VGG network. With **dropproject** (DP), one random rotation matrix is generated and used for the duration of training. For **DP-Random10**, 10 random rotation matrices are generated and a single rotation matrix is randomly chosen from these for each mini-batch during training. Finally, **DP-Random** generates a random rotation matrix for each mini-batch.

Both methods have close to identical effect on training loss/error, and are drastically different than the plots of the standard network without dropout/dropproject. At test time, both methods also achieve comparable minimum error, but in the loss curve it can be seen that dropproject appears to start overfitting earlier than dropout. While the projection (and not regularization) appears to be responsible for the increased generalization and speed of training, the zeroing out of units for dropout also has a small regularization effect not seen in dropproject. None of the variants of dropproject seem to be different, indicating that the random projection itself rather than the random rotation into a different co-ordinate basis is important for the effect.

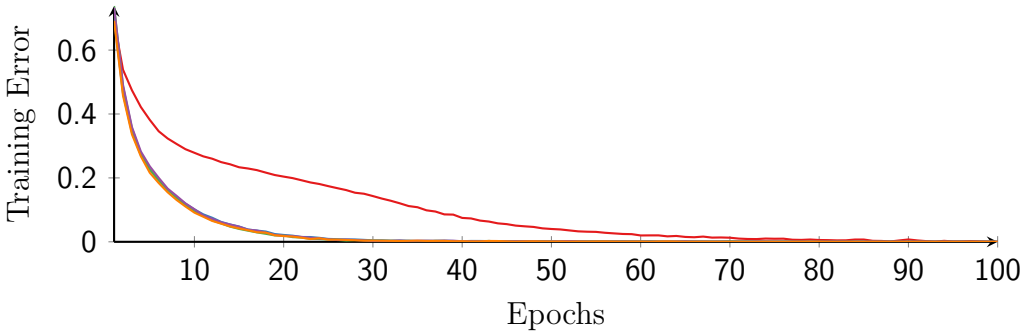
Dropout and Batch Normalization It has been observed empirically by Ioffe and Szegedy (2015) and others that when used with batch normalization, dropout is not as effective. In light of our understanding of dropout as being an optimization method for error surfaces with highly complex curvature, we can explain this. As explained by (Martens, 2010), an important property of second order optimization methods is 'scale invariance' — robustness to any linear rescaling of the model parameters. For example, if we are in an elliptically shaped local minima of the error surface, ideally we would want different a higher learning rate for parameters in the direction of the major axis, as compared to those in the direction of the minor axis, as RMSprop (Tieleman and G. Hinton, 2012) attempts. When using batch normalization, the layer-wise error surface is whitened, reducing the importance of scale-invariance in optimization, and any related optimization tricks.



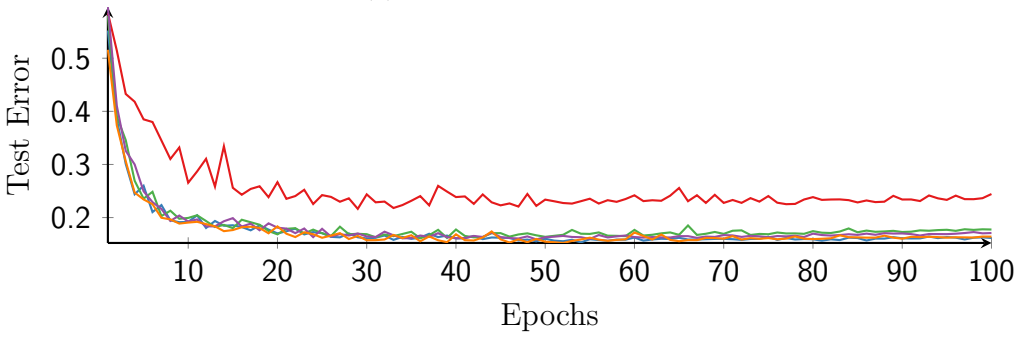
(a) CIFAR Training Log Loss



(b) CIFAR Test Log Loss



(c) CIFAR-10 Training Error



(d) CIFAR-10 Test Error

Fig. B.4 Training and test curves for a VGG network on CIFAR-10 comparing dropout to dropproject.

Appendix C

Trained Models

Although all the details for reproducing the experiments presented in this thesis are detailed in the relevant chapters, we have endevoured to make most of the models open-access, and have archived these with permanent Digital Object Identifiers.

Chapter 4

Trained models for chapter 4 are archived under the identifier:

`doi:10.5281/zenodo.53189`

Chapter 5

Trained models for chapter 5 are archived under the identifier:

`doi:10.5281/zenodo.116680`

Chapter 6

Unfortunately not all the models and code for the experiments in chapter 6 will be publicly released. The research was joint work partly pursued while on a research internship at Microsoft Research. The trained models/code we can release may be found archived under the identifier:

`doi:10.5281/zenodo.988423`

Appendix D

Conference Posters

Sometimes posters achieve what a paper cannot in explaining a concept, and although we hope this is not the case in this dissertation, for completeness we have included the links to relevant conference posters, and have archived these with permanent Digital Object Identifiers.

Chapter 4: ICLR 2016

The poster presented at ICLR 2016 is archived under the identifier:

doi:10.5281/zenodo.53187

Chapter 5: CVPR 2017

The poster presented at CVPR 2017 is archived under the identifier:

doi:10.5281/zenodo.831418

Glossary

AlexNet A neural network architecture proposed by Krizhevsky, Sutskever, and Geoffrey E. Hinton (2012) that revolutionized computer vision, and renewed interest in neural networks 31, 58

BLAS BLAS, a common API for accelerating linear algebra operations, notably matrix multiplication, on hardware. Typically a heavily optimized version is provided by the hardware company. *see* API,

CIFAR CIFAR, government agency behind the funding of several prominent researchers in Canada, notably the lab of Geoffery Hinton who released two popular datasets, CIFAR-10 and CIFAR-100 *see* CIFAR-10,

CIFAR-10 An image recognition dataset funded by CIFAR and created by Krizhevsky (2009) consisting of 60,000 32×32 colour images of 10 classes of objects *see* CIFAR, xxi, 5, 32, 81, 82, 96, 117, 139, 140, 145, 164

CNN CNN, a neural network designed for learning representations of image inputs, with shared parameters in the form of a set of convolutional filters

composite layer a DNN layer effectively composed of several potentially heterogeneous layers, *e.g.* the Inception module *see* Inception, 67, 68, 69, 70, 73, 83

CuBLAS CuBLAS, Nvidia's implementation of BLAS for the CUDA *see* BLAS & CUDA,

CUDA CUDA, Nvidia's GPU programming API *see* API & GPU,

CuDNN Nvidia's Deep Neural Network acceleration library *see* CUDA, 96, 116

DNN DNN, a neural network with two-or-more hidden layers

explicit routing in a conditional network, routing caused by a routing node 126

feature map The input/output of a convolutional layer, a 3D tensor with two spatial dimensions and a third dimension corresponding to the output image from a single convolutional filter 23, 24, 25, 32, 58, 64, 66, 68, 70, 71, 73, 89, 90, 92, 93, 94, 98, 100, 101, 106, 134, 137, 139, 153

filter A convolutional filter, or kernel, of spatial dimensions $w \times h$, and depth c, where c is the number of channels in the input *see* feature map

finetuned A method of continuing the training of a pre-trained network, with varied definitions. Typically a subset or all of the layers of a pre-trained DNN are trained at a greatly reduced learning rate 57

GoogLeNet A neural network architecture proposed by Szegedy, Liu, *et al.* (2015) and since extended in the Inception v1–4 refinements *see* Inception, 34, 58, 112

implicit routing in a conditional network, routing caused by the network's structure 126

Inception A building-block of the GoogLeNet neural network architecture designed for efficient state-of-the-art image recognition *see* GoogLeNet, 34, 35, 68, 79, 81, 92, 111, 112, 137, 152

MKL MKL, BLAS implementation for Intel CPU *see* BLAS & CPU,

MNIST MNIST, dataset of handwritten numerical digits commonly used as a 60,000 image training/10,000 image testing dataset for machine learning algorithms

NiN NiN, a neural network architecture proposed by Lin, Q. Chen, and Yan (2014) which introduced GAP and LDE *see* GAP & LDE,

object class recognition the problem of recognizing a general object class, *e.g.* recognizing a car *vs.* bicycle 22

object instance recognition the problem of recognizing a specific instance of an object class, *e.g.* recognizing a specific car model 22

Occam's razor A general principle in hypothesis selection; given several hypothesis that match the evidence, the simplest hypothesis, *i.e.* the one with the least assumptions, should be selected 50, 51

padding padding of the input feature map/image for convolution, pads the outer edge of the image with (typically zero) dummy values to allow the convolutional filter to be applied to every input pixel *see* feature map, 24, 66

regularization a broadly-used, but relatively ill-defined term — often its usage implies a definition of 'anything that improves generalization', instead we define regularization as any modification of the training algorithm that modifies, explicitly or implicitly, the error surface such that it is smoother, the prototypical method being weight decay (Geoffery E. Hinton, 1987) 2

ResNet Residual network, a network architecture proposed by He *et al.* (2016a) that uses residual connections to improve generalization and training of very deep architectures. 31, 107, 108, 112, 117, 145, 149, 159

RNN RNN, a neural network designed for sequences, with shared parameters in the form of a recurrence

stride number of rows/columns of the input feature map/image to skip during convolution *see* feature map, 25

Structural Prior The encoding of prior knowledge into a neural network by architecture design, *e.g.* a CNN, what some might call "infinitely strong regularization" (I. Goodfellow, Y. Bengio, and Courville, 2016) *see* CNN

VGG VGG, a research group at the University of Oxford from which the popular VGG network architecture was proposed by Simonyan and Zisserman (2015)

References

Ahmad, Subutai and Gerald Tesauro (1988). "Scaling and generalization in neural networks: a case study". In: *Proceedings of the 1st International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, United States). Morgan-Kaufmann, pp. 160–168 (cit. on p. 46).

Ba, Jimmy and Rich Caruana (2014). "Do Deep Nets Really Need to be Deep?" In: *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS)*. (Montréal, QC, Canada, Dec. 8–13, 2014). Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, pp. 2654–2662 (cit. on pp. 40, 158).

Baker, Bowen, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar (2017). "Designing Neural Network Architectures using Reinforcement Learning". In: *International Conference on Learning Representations (ICLR)*. (Toulon, France, Apr. 24–26, 2017). arXiv: 1611.02167 (cs-LG) (cit. on p. 155).

Bartlett, Peter L. (1996). "For Valid Generalization the Size of the Weights is More Important than the Size of the Network". In: *Proceedings of the 9th International Conference on Neural Information Processing Systems (NIPS)*. MIT Press, pp. 134–140 (cit. on pp. 43, 44).

Bastani, Osbert, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi (2016). "Measuring Neural Net Robustness with Constraints". In: *Advances in Neural Information Processing Systems*. (Barcelona, Spain, Dec. 5–Oct. 12, 2016). Curran Associates, Inc., pp. 2613–2621. arXiv: 1605.07262 (cit. on p. ix).

Baum, Eric B. and David Haussler (1988). "What Size Net Gives Valid Generalization?" In: *Proceedings of the 1st International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, United States). Morgan-Kaufmann, pp. 81–90 (cit. on pp. 40, 42, 43).

Bengio, Emmanuel, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup (2015). "Conditional Computation in Neural Networks for Faster Models" (cit. on p. 155).

Bengio, Yoshua, Patrick Simard, and Paolo Frasconi (1994). "Learning Long-Term Dependencies With Gradient Descent Is Difficult". In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166. DOI: 10.1109/72.279181 (cit. on p. 31).

Bishop, Christopher M. (1995). *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press (cit. on pp. 8, 9, 12, 13, 44, 52).

Bottou, Léon (2012). "Stochastic gradient descent tricks". In: *Neural Networks: Tricks of the Trade*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Second. Vol. 7700. Lecture Notes in Computer Science. Springer, pp. 421–436. DOI: 10.1007/978-3-642-35289-8_25 (cit. on pp. 17, 75, 135, 140).

Bulò, Samuel Rota and Peter Kontschieder (2014). "Neural Decision Forests for Semantic Image Labelling". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Columbus, OH, USA, June 24–27, 2014), pp. 81–88. DOI: 10.1109/CVPR.2014.18 (cit. on p. 125).

Burges, Christopher J.C. (1998). "A Tutorial on Support Vector Machines for Pattern Recognition". In: *Data Mining and Knowledge Discovery* 2.2, pp. 121–167. DOI: 10.1023/A:1009715923555 (cit. on pp. 41, 42).

Caruana, Rich, Steve Lawrence, and C. Lee Giles (2000). "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping". In: *Proceedings of the 13th International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, USA, Nov. 27–Dec. 2, 2000). Cambridge, MA, USA: MIT Press, pp. 381–387 (cit. on pp. 40, 44).

Castellano, Giovanna, Anna Maria Fanelli, and Marcello Pelillo (1997). "An iterative pruning algorithm for feedforward neural networks". In: *IEEE Transactions on Neural Networks* 8.3, pp. 519–531. DOI: 10.1109/72.572092 (cit. on p. 54).

Changpinyo, Soravit, Mark Sandler, and Andrey Zhmoginov (2017). "The power of sparsity in convolutional neural networks" (cit. on p. 154).

Chen, Wenlin, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen (2015). "Compressing Neural Networks with the Hashing Trick". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. (Lille, France, July 7–9, 2015). Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR, pp. 2285–2294. arXiv: 1504.04788 (cit. on p. 90).

Chipman, Hugh, Edward I. George, and Robert E. McCulloch (2001). "The Practical Implementation of Bayesian Model Selection". In: *Lecture Notes-Monograph Series* 38, pp. 65–134 (cit. on p. 52).

Cogswell, Michael, Faruk Ahmed, Ross B. Girshick, Larry Zitnick, and Dhruv Batra (2016). "Reducing Overfitting in Deep Networks by Decorrelating Representations."

In: *International Conference on Learning Representations (ICLR)*. (San Juan, Puerto Rico, May 2–4, 2016). arXiv: 1511.06068 (cit. on pp. 94, 158).

Cybenko, George (1989). "Approximation by superpositions of a sigmoid function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4, pp. 303–314 (cit. on pp. 3, 21).

Damelin, Steven B. and Willard Miller Jr (2012). *The Mathematics of Signal Processing*. Cambridge: Cambridge University Press, p. 462. DOI: 10.1017/CBO9781139003896 (cit. on p. 23).

Denil, Misha, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas (2013). "Predicting Parameters in Deep Learning". In: *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS)*. (Lake Tahoe, NV, USA, Dec. 5–10, 2012). Curran Associates, Inc., pp. 2148–2156. arXiv: 1306.0543 (cit. on pp. 63, 89).

Denker, John, Daniel Schwartz, Ben Wittner, Sara Solla, Richard Howard, Lawrence Jackel, and John Hopfield (1987). "Large automatic learning, rule extraction, and generalization". In: *Complex systems* 1.5, pp. 877–922 (cit. on pp. 44, 46, 47, 59).

Denton, Emily L., Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus (2014). "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS)*. (Montréal, QC, Canada, Dec. 8–13, 2014). Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., pp. 1269–1277. arXiv: 1404.0736 (cit. on p. 89).

Fahlman, Scott E. and Christian Lebiere (1989). "The Cascade-Correlation Learning Architecture". In: *Proceedings of the 2nd International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, USA, Nov. 27–30, 1989). Ed. by David S Touretzky. Morgan Kaufmann, pp. 524–532 (cit. on pp. 52, 53, 148).

Fodor, Imola K. (2002). *A survey of dimension reduction techniques*. Tech. rep. UCRL-ID-148494. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory. DOI: 10.2172/15002155 (cit. on p. 163).

Frean, Marcus (1990). "The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks". In: *Neural Computation* 2.2, pp. 198–209. DOI: 10.1162/neco.1990.2.2.198 (cit. on p. 52).

Fukushima, Kunihiko (1980). "Neocognitron: A self-organizing neural network model for a mechanish of pattern recognition unaffected by shifts in position". In: *Biological Cybernetics* 36, pp. 193–202 (cit. on pp. 22, 61, 89).

Fukushima, Kunihiko (2013). "Artificial vision by multi-layered neural networks: Neocognitron and its advances". In: *Neural Networks* 37, pp. 103–119. DOI: 10.1016/j.neunet.2012.09.016 (cit. on p. 22).

Gallant, Stephen I. (1986). "Optimal linear discriminants". In: *Eighth International Conference on Pattern Recognition (ICPR)*. (Paris, France, Oct. 27–31, 1986). IAPR, pp. 849–852 (cit. on pp. 52, 54).

Giles, C. Lee and Tom Maxwell (1987). "Learning, invariance, and generalization in high-order neural networks". In: *Applied Optics* 26.23, pp. 4972–4978. DOI: 10.1364/AO.26.004972 (cit. on p. 46).

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*. (Chia Laguna Resort, Sardinia, Italy, May 13–15, 2010). Vol. 9, pp. 249–256 (cit. on pp. 27–29, 69–71, 157).

Goodfellow, Ian J., David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio (2013). "Maxout Networks". In: *Proceedings of the 30th International Conference on Machine Learning (ICML)*. (Atlanta, GA, USA, June 17–19, 2013). Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. 3, pp. 1319–1327. arXiv: 1302.4389 (cit. on pp. 81, 96).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press (cit. on pp. 2, 7, 8, 17, 19, 22, 24, 173).

Gorodkin, Jan, Lars Kai Hansen, Anders Krogh, Claus Svarer, and Ole Winther (1993). "A quantitative study of pruning by optimal brain damage". In: *International Journal of Neural Systems* 4.02, pp. 159–169. DOI: 10.1142/S0129065793000146 (cit. on p. 54).

Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan (2015). *Deep Learning with Limited Numerical Precision*. Ed. by Francis R. Bach and David M. Blei. arXiv: 1502.02551 (cit. on pp. 63, 90).

Han, Song, Huizi Mao, and William J. Dally (2016). "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: arXiv: 1510.00149 (cit. on pp. 54, 57).

Han, Song, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, Bryan Catanzaro, and William J. Dally (2017). "DSD: Dense-Sparse-Dense Training for Deep Neural Networks". In: *International Conference on Learning Representations (ICLR)*. (Toulon, France, Apr. 24–26, 2017). arXiv: 1607.04381 (cit. on pp. 55, 57).

Han, Song, Jeff Pool, John Tran, and William J. Dally (2015). "Learning both weights and connections for efficient neural network". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*. (Montréal, QC, Canada, Dec. 7–12, 2015). Curran Associates, Inc., pp. 1135–1143. arXiv: 1506.02626 (cit. on pp. 55–57).

Hanson, Stephen José and Lorien Y. Pratt (1988). "Comparing biases for minimal network construction with back-propagation". In: *Proceedings of the 1st International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, United States). Morgan Kaufmann, pp. 177–185 (cit. on pp. 46, 54, 55, 119).

Haykin, Simon (1994). *Neural Networks: A Comprehensive Foundation*. Prentice Hall (cit. on p. 13).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *IEEE International Conference on Computer Vision (ICCV)*. (Santiago, Chile, Dec. 7–13, 2015). IEEE, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123. arXiv: 1502.01852 (cit. on pp. 29, 69–71, 75, 76, 83, 96, 107, 111, 120, 133, 135, 137, 140, 148, 157).

— (2016a). "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Las Vegas, NV, USA, June 27–30, 2016), pp. 770–778. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385 (cit. on pp. 25, 35, 36, 40, 92, 96, 107, 158, 159, 173).

— (2016b). "Identity Mappings in Deep Residual Networks". In: *14th European Conference on Computer Vision (ECCV)*. (Amsterdam, The Netherlands, Oct. 11–14, 2016). Vol. 5. Springer, pp. 630–645. DOI: 10.1007/978-3-319-46493-0_38. arXiv: 1603.05027 (cit. on pp. 40, 148, 158).

Hebb, Donald Olding (1949). *The organization of behavior: A neuropsychological approach*. John Wiley & Sons (cit. on p. 40).

Hinton, Geoffery E. (1987). "Learning translation invariant recognition in a massively parallel networks". In: *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE)*. (Eindhoven, The Netherlands, June 15–19, 1987). Vol. 1. Springer, pp. 1–13. DOI: 10.1007/3-540-17943-7_117 (cit. on pp. 44, 46, 59, 173).

— (2015). *Deep Learning*. Public Lecture. University of Cambridge (cit. on pp. 40, 41).

Hinton, Geoffrey E. and Ruslan R. Salakhutdinov (2006). "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786, pp. 504–507. DOI: 10.1126/science.1127647 (cit. on p. 28).

Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov (2012a). "Improving neural networks by preventing co-adaptation of feature detectors" (cit. on pp. 30, 158).

— (2012b). *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv: 1207.0580 (cit. on pp. 31, 94, 157).

Hochreiter, Sepp (1991). "Untersuchungen zu dynamischen neuronalen Netzen". PhD thesis. Munich, Germany: Technische University of Munich (cit. on pp. 22, 27).

Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber (2001). "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies". In: *A Field Guide to Dynamical Recurrent Networks*. Ed. by John F. Kolen and Stefan C. Kremer. Wiley-IEEE Press. Chap. 14, pp. 237–243. DOI: 10.1109/9780470544037.ch14. arXiv: arXiv:1011.1669 (cit. on p. 70).

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 356–366. DOI: 10.1016/0893-6080(89)90020-8 (cit. on pp. 3, 21, 40, 42).

Ioannou, Yani, Duncan Robertson, Roberto Cipolla, and Antonio Criminisi (2017). "Deep Roots: Improving CNN efficiency with hierarchical filter groups". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Honolulu, HI, USA, July 21–26, 2017), pp. 5977–5986. DOI: 10.1109/CVPR.2017.633. arXiv: 1605.06489 (cit. on pp. viii, ix, 152, 153).

Ioannou, Yani, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi (2016). "Training CNNs with Low-Rank Filters for Efficient Image Classification". In: *International Conference on Learning Representations (ICLR)*. (San Juan, Puerto Rico, May 2–4, 2016). arXiv: 1511.06744 (cit. on pp. viii, ix, 152).

Ioannou, Yani, Duncan Robertson, Darko Zikic, Peter Kontschieder, Jamie Shotton, Matthew Brown, and Antonio Criminisi (2015). *Decision Forests, Convolutional Networks and the Models in-Between*. Tech. rep. MSR-TR-2015-58. Microsoft Research (cit. on pp. viii, ix, 151).

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. (Lille, France, July 7–9, 2015). Ed. by Francis R. Bach and David M. Blei. Vol. 37, pp. 448–456. arXiv: 1502.03167 (cit. on pp. 29, 31, 44, 96, 107, 111, 148, 157, 158, 164).

Jaderberg, Max, Andrea Vedaldi, and Andrew Zisserman (2014). "Speeding up Convolutional Neural Networks with Low Rank Expansions". In: *Proceedings of the*

British Machine Vision Conference. (Nottingham, UK, Sept. 1–4, 2014). BMVA Press. DOI: 10.5244/C.28.88. arXiv: 1405.3866 (cit. on pp. 64–67, 76, 78, 90, 93).

Jain, Ashesh, Amir R. Zamir, Silvio Savarese, and Ashutosh Saxena (2016). "Structural-RNN: Deep Learning on Spatio-Temporal Graphs". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Las Vegas, NV, USA, June 27–30, 2016), pp. 5308–5317. DOI: 10.1109/CVPR.2016.573 (cit. on p. 58).

Jhurani, Chetan and Paul Mullowney (2015). "A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices". In: *Journal of Parallel and Distributed Computing* 75, pp. 133–140. DOI: 10.1016/j.jpdc.2014.09.003 (cit. on p. 116).

Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell (2014). "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. (Orlando, FL, USA, Nov. 3–7, 2014), pp. 675–678. DOI: 10.1145/2647868.2654889. arXiv: 1408.5093 (cit. on pp. 75, 81, 111, 129, 131, 133, 135).

Kaski, Samuel (1998). "Dimensionality reduction by random mapping: Fast similarity computation for clustering". In: *IEEE International Joint Conference on Neural Networks Proceedings*. (Anchorage, AK, USA, USA, May 4–9, 1998). Vol. 1. IEEE, pp. 413–418. DOI: 10.1109/IJCNN.1998.682302 (cit. on p. 163).

Kim, Yong-Deok, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin (2016). "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications". In: *International Conference on Learning Representations (ICLR)*. (San Juan, Puerto Rico, May 2–4, 2016), pp. 1–16. arXiv: 1511.06530 (cit. on pp. 57, 90, 93).

Kontschieder, Peter, Madelina Fiterau, Antonio Criminisi, and Samuel Rota Bulò (2015). "Deep Neural Decision Forests". In: *IEEE International Conference on Computer Vision (ICCV)*. (Santiago, Chile, Dec. 7–13, 2015). IEEE, pp. 1467–1475. DOI: 10.1109/ICCV.2015.172 (cit. on p. 130).

Krizhevsky, Alex (2009). *Learning Multiple Layers of Features from Tiny Images*. Technical Report. Univ. Toronto (cit. on pp. 81, 96, 101, 139, 164, 171).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*. (Lake Tahoe, NV, USA, Dec. 3–8, 2012), pp. 1097–1105. arXiv: 1102.0183 (cit. on pp. 25, 27, 31, 32, 34, 40, 41, 43, 89, 90, 121, 129, 133, 134, 171).

Lattimore, Tor and Marcus Hutter (2013). "No Free Lunch versus Occam's Razor in Supervised Learning". In: *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence*. Ed. by David L. Dowe. Springer, pp. 223–235. DOI: 10.1007/978-3-642-44958-1_17 (cit. on p. 49).

Lebedev, Vadim, Yaroslav Ganin, Maksim Rakhuba1, Ivan Oseledets, and Victor Lempitsky (2015). "Speeding-Up Convolutional Neural Networks Using Fine-tuned CP-Decomposition". In: *International Conference on Learning Representations (ICLR)*. (San Diego, CA, USA, May 7–9, 2015). arXiv: 1412.6553 (cit. on pp. 90, 93).

LeCun, Yann (1989). "Generalization and network design strategies". In: ed. by R. Pfeifer, Z. Schreter, F. Fogelman-Soulié, and L. Steels. First. Zurich, Switzerland: Elsevier. Chap. 9, pp. 143–155 (cit. on pp. 44, 46).

LeCun, Yann, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural Computation* 1.4, pp. 541–551. DOI: 10.1162/neco.1989.1.4.541 (cit. on pp. 40, 44).

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on pp. 3, 22, 25, 61, 89).

LeCun, Yann, John S. Denker, and Sara A. Solla (1989). "Optimal Brain Damage". In: *Proceedings of the 2nd International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, USA, Nov. 27, 1989–Nov. 30, 2012). Vol. 2. 1, pp. 598–605 (cit. on pp. 55, 63, 148).

Lee, Tae Kwan, Wissam J. Baddar, Seong Tae Kim, and Yong Man Ro (2017). "Convolution with Logarithmic Filter Groups for Efficient Shallow CNN" (cit. on p. 154).

Lin, Min, Qiang Chen, and Shuicheng Yan (2014). "Network In Network". In: *International Conference on Learning Representations (ICLR)*. (Rimrock Resort, Banff, AB, Canada, Apr. 14–16, 2014). arXiv: 1312.4400 (cit. on pp. 32, 34, 63, 78, 79, 81, 90, 92, 96, 107, 134, 139, 140, 172).

Lowe, David G (2004). "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2, pp. 91–110. DOI: 10.1023/B:VISI.0000029664.99615.94 (cit. on pp. 23, 39).

MacKay, David J.C. (1991). "Bayesian Methods for Adaptive Models". PhD thesis. Pasadena, CA, USA: California Institute of Technology (cit. on pp. 50, 51).

— (1992). "A Practical Bayesian Framework for Backpropagation Networks". In: *Neural Computation* 4.3, pp. 448–472. DOI: 10.1162/neco.1992.4.3.448 (cit. on pp. 50–52).

— (1995). "Probable networks and plausible predictions — a review of practical Bayesian methods for supervised neural networks". In: *Network: Computation in Neural Systems* 6.3, pp. 469–505. DOI: 10.1088/0954-898X_6_3_011 (cit. on pp. 50, 51).

Mamalet, Franck and Christophe Garcia (2012). "Simplifying ConvNets for Fast Learning". In: *22nd International Conference on Artificial Neural Networks (ICANN)*. Ed. by Alessandro E. P. Villa, Włodzisław Duch, Péter Érdi, Francesco Masulli, and Günther Palm. 2. Springer, pp. 58–65. DOI: 10.1007/978-3-642-33266-1_8 (cit. on pp. 64, 90).

Martens, James (2010). "Deep learning via Hessian-free optimization". In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*. (Haifa, Israel, June 21–24, 2010), pp. 735–742 (cit. on pp. 18, 19, 158, 164).

Mathieu, Michael, Mikael Henaff, and Yann LeCun (2014). "Fast Training of Convolutional Networks through FFTs". In: *International Conference on Learning Representations (ICLR)*. (Rimrock Resort, Banff, AB, Canada, Apr. 14–16, 2014). arXiv: 1312.5851v5 (cit. on p. 90).

Mezard, Marc and Jean-P Nadal (1989). "Learning in feedforward layered networks: The tiling algorithm". In: *Journal of Physics A: Mathematical and General* 22.12, pp. 2191–2203. DOI: 10.1088/0305-4470/22/12/019 (cit. on p. 52).

Minsky, Marvin and Seymour Papert (1988). *Perceptrons*. Second, Expanded Edition. MIT press (cit. on pp. 9, 40, 46, 89).

Montillo, Albert, Jamie Shotton, John Winn, Juan Eugenio Iglesias, Dimitri Metaxas, and Antonio Criminisi (2011). "Entangled decision forests and their application for semantic segmentation of CT images". In: *22nd International Conference on Information Processing in Medical Imaging (IPMI)*. (Kloster Irsee, Germany, July 3–8, 2011). Ed. by Gábor Székely and Horst K. Hahn. Springer, pp. 184–196. DOI: 10.1007/978-3-642-22092-0_16 (cit. on p. 125).

Mozer, Michael C. and Paul Smolensky (1988). "Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment". In: *Proceedings of the 1st International Conference on Neural Information Processing Systems (NIPS)*. (Denver, CO, United States). Morgan-Kaufmann, pp. 107–115 (cit. on p. 55).

— (1989). "Using Relevance to Reduce Network Size Automatically". In: *Connection Science* 1.1, pp. 3–16. DOI: 10.1080/09540098908915626 (cit. on p. 55).

Nair, Vinod and Geoffrey E. Hinton (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning*. (Haifa, Israel, June 21–24, 2010). 3, pp. 807–814 (cit. on pp. 27, 31).

Parekh, Rajesh, Jihoon Yang, and Vasant Honavar (2000). "Constructive neural-network learning algorithms for pattern classification". In: *IEEE Transactions on Neural Networks* 11.2, pp. 436–451. DOI: 10.1109/72.839013 (cit. on p. 52).

Polyak, Boris T. (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17 (cit. on p. 19).

Rastegari, Mohammad, Vicente Ordonez, Joseph Redmon, and Ali Farhadi (2016). "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *14th European Conference on Computer Vision (ECCV)*. (Amsterdam, The Netherlands, Oct. 11–14, 2016). Springer, pp. 525–542. DOI: 10.1007/978-3-319-46493-0_32. arXiv: 1603.05279 (cit. on pp. 57, 58).

Rigamonti, Roberto, Amos Sironi, Vincent Lepetit, and Pascal Fua (2013). "Learning Separable Filters". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Portland, OR, USA, June 23–28, 2013). IEEE, pp. 2754–2761. DOI: 10.1109/CVPR.2013.355 (cit. on pp. 63, 90).

Rippel, Oren, Jasper Snoek, and Ryan Prescott Adams (2015). "Spectral Representations for Convolutional Neural Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*. (Montréal, QC, Canada, Dec. 7–12, 2015). Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., pp. 2440–2448. arXiv: 1506.03767 (cit. on p. 90).

Rosenblatt, Frank (1958). "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological review* 65.6, p. 386 (cit. on p. 8).

— (1961). *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. VG-1196-G-8. Cornell Aeronautical Laboratory Ltd., Cornell University, Buffalo, NY (cit. on p. 11).

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088, pp. 533–536. DOI: 10.1038/323533a0 (cit. on pp. 13, 19).

Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C.

Berg, and Fei-Fei Li (2015). "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3, pp. 211–252. DOI: 10.1007/s11263-015-0816-y (cit. on pp. 31, 43, 133, 148, 159).

Schulter, S., P. Wohlhart, C. Leistner, A. Saffari, P. M. Roth, and H. Bischof (2013). "Alternating Decision Forests". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Portland, OR, USA, June 23–28, 2013). IEEE, pp. 508–515. DOI: 10.1109/CVPR.2013.72 (cit. on p. 130).

Sermanet, Pierre, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann LeCun (2014). "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks". In: *International Conference on Learning Representations (ICLR)*. (Rimrock Resort, Banff, AB, Canada, Apr. 14–16, 2014). arXiv: 1312.6229 (cit. on p. 34).

Sethi, I. K. (1990). "Entropy Nets: From Decison Trees to Neural Networks". In: *Proceedings of the IEEE* 78.10, pp. 1605–1613. DOI: 10.1109/5.58346 (cit. on p. 120).

Setiono, Rudy (1997). "A Penalty-Function Approach for Pruning Feedforward Neural Networks". In: *Neural Computation* 9.1, pp. 185–204. DOI: 10.1162/neco.1997.9.1.185 (cit. on p. 55).

Shankar, Sukrit, Duncan Robertson, Yani Ioannou, Antonio Criminisi, and Roberto Cipolla (2016). "Refining Architectures of Deep Convolutional Neural Networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Las Vegas, NV, USA, June 27–30, 2016), pp. 2212–2220. DOI: 10.1109/CVPR.2016.243. arXiv: 1604.06832 (cit. on p. ix).

Shotton, Jamie, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake (2011). "Real-time human pose recognition in parts from single depth images". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Colorado Springs, CO, USA, June 20–25, 2011). Vol. 2. 3. IEEE, pp. 1297–1304. DOI: 10.1109/CVPR.2011.5995316 (cit. on p. 120).

Sietsma, Jocelyn and Robert J.F. Dow (1988). "Neural net pruning-why and how". In: *IEEE International Conference on Neural Networks*. Vol. 1. IEEE San Diego, pp. 325–333. DOI: 10.1109/ICNN.1988.23864 (cit. on pp. 54, 55).

Simonyan, Karen and Andrew Zisserman (2015). "Very deep convolutional networks for large-scale image recognition". In: *International Conference on Learning Representations (ICLR)*. (San Diego, CA, USA, May 7–9, 2015). arXiv: 1409.1556 (cit. on pp. 34, 35, 40, 63, 75, 76, 78, 82, 83, 107, 121, 122, 131–133, 135, 173).

Snoek, Jasper, Hugo Larochelle, and Ryan Prescott Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*. (Lake Tahoe, NV, USA, Dec. 3–8, 2012). Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, pp. 2951–2959. arXiv: 1206.2944 (cit. on p. 140).

Srivastava, Nitish, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov (2014). "Dropout : A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research (JMLR)* 15.1, pp. 1929–1958 (cit. on pp. 30, 158).

Suárez, Alberto and James F Lutsko (1999). "Globally optimal fuzzy decision trees for classification and regression". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21.12, pp. 1297–1311. DOI: 10.1109/34.817409 (cit. on p. 130).

Sutskever, Ilya, James Martens, George E. Dahl, and Geoffrey E. Hinton (2013). "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. (Atlanta, GA, USA, June 17–19, 2013). Vol. 28. 3, pp. 1139–1147 (cit. on p. 31).

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich (2015). "Going Deeper with Convolutions". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Boston, MA, USA, June 7–12, 2015). DOI: 10.1109/CVPR.2015.7298594 (cit. on pp. 34, 68, 78, 79, 83, 89, 92, 111, 129, 134, 135, 142, 172).

Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna (2016). "Rethinking the Inception Architecture for Computer Vision". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Las Vegas, NV, USA, June 27–30, 2016), pp. 2818–2826. DOI: 10.1109/CVPR.2016.308. arXiv: 1512.00567 (cit. on p. 152).

Szeliski, Richard (2011). *Computer Vision: Algorithms and Applications*. 1st. New York, NY, USA: Springer-Verlag New York, Inc. DOI: 10.1007/978-1-84882-935-0 (cit. on p. 24).

Tieleman, Tijmen and Geoffrey Hinton (2012). *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*. Public (Online) Lecture (cit. on p. 164).

Ullrich, Karen, Edward Meeds, and Max Welling (2017). "Soft Weight-Sharing for Neural Network Compression". In: *International Conference on Learning Representations (ICLR)*. (Toulon, France, Apr. 24–26, 2017). arXiv: 1702.04008 (cit. on pp. 55, 57).

Vanhoucke, Vincent, Andrew Senior, and Mark Z. Mao (2011). "Improving the speed of neural networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. (Granada, Spain, Dec. 16, 2011) (cit. on p. 63).

Vapnik, Vladimir N. and Alexey Ya. Chervonenkis (2015). "On the uniform convergence of relative frequencies of events to their probabilities". In: *Measures of Complexity*. Ed. by B. Seckler. Springer, pp. 11–30. DOI: 10.1007/978-3-319-21852-6_3 (cit. on pp. 40, 41).

Widrow, Bernard and Marcian E. Hoff (1960). *Adaptive switching circuits*. Tech. rep. 1553-1. Solid State Electronics Laboratory, Stanford University, Stanford, CA (cit. on p. 11).

Wolpert, David H. (1996). "The Lack of A Priori Distinctions Between Learning Algorithms". In: *Neural Computation* 8.7, pp. 1341–1390. DOI: 10.1162/neco.1996.8.7.1341 (cit. on p. 48).

Xie, Saining, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He (2017). "Aggregated Residual Transformations for Deep Neural Networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Honolulu, HI, USA, July 21–26, 2017). arXiv: 1611.05431 (cit. on pp. 153, 154).

Yi, Kwang Moo, Eduard Trulls, Vincent Lepetit, and Pascal Fua (2016). "LIFT: Learned Invariant Feature Transform". In: *14th European Conference on Computer Vision (ECCV)*. (Amsterdam, The Netherlands, Oct. 11–14, 2016). 6. Springer, pp. 467–483. DOI: 10.1007/978-3-319-46466-4_28. arXiv: 1603.09114 (cit. on p. 119).

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals (2017). "Understanding deep learning requires rethinking generalization". In: *International Conference on Learning Representations (ICLR)*. (Toulon, France, Apr. 24–26, 2017). arXiv: 1611.03530 (cit. on pp. 2, 48, 58).

Zhang, Ting, Guo-Jun Qi, Bin Xiao, and Jingdong Wang (2017). "Interleaved Group Convolutions for Deep Neural Networks". In: *IEEE International Conference on Computer Vision (ICCV)*. (Venice, Italy, Oct. 22–29, 2017). arXiv: 1707.02725 (cit. on p. 154).

Zhang, Xiangyu, Xinyu Zhou, Mengxiao Lin, and Jian Sun (2017). "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices" (cit. on p. 154).

Zhou, Bolei, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva (2014). "Learning Deep Features for Scene Recognition using Places Database". In: *Proceedings of the 27th International Conference on Neural Information Processing*

Systems (NIPS). (Montréal, QC, Canada, Dec. 8–13, 2014). Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., pp. 487–495 (cit. on p. 75).

Zikic, Darko, Yani Ioannou, Antonio Criminisi, and Matthew Brown (2014). "Segmentation of Brain Tumor Tissues with Convolutional Neural Networks". In: *MICCAI workshop on Multimodal Brain Tumor Segmentation Challenge (BRATS)*. (Boston, MA, USA, Sept. 14, 2014) (cit. on p. ix).

Zoph, Barret and Quoc V. Le (2017). "Neural Architecture Search with Reinforcement Learning". In: *International Conference on Learning Representations (ICLR)*. (Toulon, France, Apr. 24–26, 2017). arXiv: 1611.01578 (cit. on p. 155).

Index