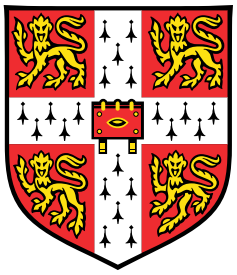
Probabilistic Continual Learning using Neural Networks



Siddharth Swaroop

Department of Engineering
University of Cambridge

This thesis is submitted for the degree of
Doctor of Philosophy

Churchill College January 2022

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis 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 thesis 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 thesis does not exceed the prescribed word limit for the relevant Degree Committee.

Siddharth Swaroop
January 2022

Acknowledgements

This thesis is the result of work over many years, and has been impacted by many people. I would firstly like to thank my supervisor, Richard Turner, for his support and time throughout my PhD, particularly early on when I was discovering what it means to do research. He was always willing to get involved in the technical details, and also provide high-level feedback on my ideas. I learnt a lot from him about the importance of good mentorship, and I hope his passion and ability to teach rubs off on me a little. I was also fortunate to have a fruitful collaboration with Mohammad Emtiyaz Khan, spanning several years, papers, and time zones, complete with the occasional trip to Japan (at least, before COVID-19)! He has spent a lot of time guiding, teaching and supporting me, and I have learnt a lot from him technically. I especially appreciate his openness on the joys and tribulations of research life. Both people have greatly impacted the kind of researcher I aspire to be.

Thanks to Rich and Emti, and their groups' open research cultures, I have been able to work with and learn from very many co-authors and collaborators: Thang D Bui, Cuong V Nguyen, Marcin B Tomczak, Mrinank Sharma, Michael Hutchinson, Antti Honkela, Kazuki Osawa, Anirudh Jain, Runa Eschenhagen, Rio Yokota, Noel Loo, Pingbo Pan, Alexander Immer, Andrew YK Foong, Dharmesh Tailor, Paul E Chang, Voot Tangkaratt, Arno Solin, Erik Daxberger, José Miguel Hernández-Lobato, Matthew Ashman, Stratis Markou, Mikko Heikkilä, Yingzhen Li, Diego Mesquita, Samuel Kaski, Martin Kukla, John Winn, Pavel Myshkov and Tom Minka. Every person brought their unique perspectives in my conversations with them, and I am grateful to all of them, even if our project together was not eventually published. I would also like to thank my examiners Carl Henrik Ek and Roger Grosse for their constructive feedback and for helping improve this thesis.

I have also had many important conversations with various members of the CBL, and I value these immensely. In particular, I would like to mention Adrià Garriga-Alonso, John Bronskill, Talay Cheema, Jonathan Gordon, Marton Havasi, Robert Pinsler and Will Tebbutt, with whom I shared an office and many an interesting conversation.

Finally, I owe a lot to my loved ones, family and friends. They know who they are.

Abstract

Probabilistic Continual Learning using Neural Networks

Neural networks are being increasingly used in society due to their strong performance at a large scale. They excel when they have access to all data at once, requiring multiple passes through the data. However, standard deep-learning techniques are unable to continually adapt as the environment changes: either they forget old data or they fail to sufficiently adapt to new data. This limitation is a major barrier to applications in many real-world settings, where the environment is often changing, and also in stark contrast to humans, who continuously learn over their lifetimes. The study of learning systems in these settings is called continual learning: data examples arrive sequentially and predictions must be made online.

In this thesis we present new algorithms for continual learning using neural networks. We use the probabilistic approach, which maintains a distribution over beliefs, naturally handling continual learning by recursively updating from priors to posteriors. Although previous work has been limited by approximations to this idealised scheme, we scale our probabilistic algorithms to large-data settings and show strong empirical performance. We also theoretically analyse why our algorithms perform well in continual learning.

We start with a variational approximation over neural network weights in Chapter 3. Previous weight-prior algorithms converge slowly, and we speed up convergence by using natural-gradient updates, allowing us to scale to large-data settings such as ImageNet for the first time. However, we find there is still room for improving continual learning performance.

We argue that ultimately we are only interested in model outputs, and this motivates us to view neural networks in function-space and regularise their outputs directly in Chapter 4. We approximate a term in the variational objective with its function-space alternative, leading to FROMP. FROMP identifies and regularises on a few memorable past examples to avoid forgetting, and performs very well on existing continual learning benchmarks.

However, we find that FROMP is not exact in simple settings such as Generalised Linear Models (GLMs). We fix this in Chapter 5 with a method called Knowledge-adaptation priors (K-priors), a generalisation of FROMP and weight-priors that can be exact on GLMs. K-priors achieve quick and accurate adaptation across many adaptation tasks, including adding data (as in continual learning) but also removing data, changing the regulariser, and changing the model. We use K-priors to provide insight into why our previous methods achieve good performance, and we suggest improvements to them. Overall, in this thesis we provide a comprehensive probabilistic framework for continual learning using neural networks, and provide thorough evaluation of instances of this framework.

Table of contents

Chapter 1

Introduction

The real world is an ever-changing environment. This makes deploying machine-learning algorithms very difficult: modelling assumptions and choices we make today can become irrelevant in the future. On the other hand, humans show an ability to quickly adapt to changes around them. This thesis is concerned with designing algorithms that can handle such adaptation.

For example, consider an image classifier for use in autonomous vehicles, trained to classify between objects such as traffic lights, pedestrians, pavements, and other vehicles such as cars or motorbikes or cycles. Such classifiers play a crucial role in autonomous vehicles' decision-making, informing the vehicle which objects are around it.

We train our classifier by showing it many labelled images of objects, and then deploy it on our autonomous vehicle. But the real world is constantly changing, and if we do not account for this, our classifier will slowly perform worse over time, leading to potentially disastrous accidents. This can happen whenever the real world departs from assumptions made during training. For example, perhaps people start using electric scooters commonly on roads, or the design of road vehicles slowly change over time.

To deal with such changes, we can collect new data (as labelled images), and retrain our system. Ideally, our system would quickly adapt to small changes, similar to how humans can quickly adapt to new concepts.

Current state-of-the-art models for such large-scale problems (especially image classification) are deep neural networks, and they are widely used in society already. Unfortunately, neural networks require multiple passes through *all* data to train. Consider collecting a thousand new images (of electric scooters) to update a system previously trained on a million images. In order to perform well on both old and new data, neural networks require retraining from scratch on *all* one million plus one thousand images. This is expensive and time-consuming. Ideally, we would quickly adapt our system by only training on the thousand new

images, but standard deep-learning techniques cannot do this. This is a particular problem when faced with multiple adaptation steps sequentially: either the neural network forgets old information, or it fails to sufficiently adapt to new information.

The field of continual learning deals with this setting. In continual learning (also known as lifelong learning or sequential learning), data examples arrive sequentially and predictions must be made in an online fashion. We are not allowed to store all past data. Developing algorithms for continual learning allows us to avoid retraining-from-scratch every time we want to update our system with new data. We may also satisfy some privacy requirements as we do not store all past data. Standard deep-learning techniques fail catastrophically in this setting, and we therefore need to design new algorithms and techniques.

1.1 The probabilistic approach to continual learning

We use a probabilistic approach to continual learning using neural networks. The probabilistic approach maintains a distribution over beliefs. We start with a distribution summarising our prior beliefs, and when we see new data, we update it into a posterior distribution. We use the Bayesian update, which follows the rules of probability.

This is easily applied to continual learning: every time we see new data, we update our distribution over beliefs, recursively updating from priors to posteriors. As we see more and more data, we slowly refine our distribution over beliefs, slowly becoming more certain. In this way, the probabilistic approach naturally handles continuously-arriving data.[1]

On the other hand, standard deep-learning techniques keep only a single belief (instead of a distribution). This can lead to over-committing to a particular belief early on, and being unable to correct for this later. This leads to failures in continual learning.

Unfortunately, the probabilistic approach is difficult to realise in practice, particularly at a large scale, due to the heavy computation required. Approximations are required to the idealised scheme, potentially leading to worse performance. Our goal is to make approximations that have low computation cost while keeping benefits of the probabilistic framework. Previous work in the field has been limited by the approximations made, and we aim to improve on such previous work in this thesis.

[1] The probabilistic approach also has other potential benefits, like better robustness, better calibration, and better performance in low-data settings, but we do not consider these other benefits in detail in this thesis.

1.2 Thesis outline

In this chapter so far, we have motivated and introduced intuitions about probabilistic continual learning using neural networks. In Chapter 2 we mathematically formalise these intuitions. We show how the Bayesian update is exact for continual learning, and introduce the variational approximation that we will use throughout this thesis. We also summarise previous methods for continual learning, characterising them in terms of three orthogonal approaches. We end Chapter 2 by describing the metrics and benchmarks that we use throughout the thesis to compare methods.

In Chapter 3 we improve variational weight-prior methods for probabilistic continual learning. When we look into why they work, we find that entire units are pruned out, and that this pruning effect can help in continual learning. We use natural-gradient update steps to speed up convergence, allowing us to scale to large-data settings such as ImageNet for the first time. However, we find that there is still room for improving continual learning performance, and we argue that this is due to weight-space approximations we made.

This motivates us to view neural networks in function-space, and regularise their outputs directly. In Chapter 4 we do this in a probabilistic way with a method called Functional Regularisation of Memorable Past (FROMP). We approximate a term in our objective function with its function-space alternative, and use a Gaussian Process formulation of neural networks to identify and regularise on only a few memorable past datapoints. FROMP significantly improves upon previous weight-prior approaches on benchmarks.

In Chapter 5 we introduce Knowledge-adaptation priors (K-priors), which are a generalisation of FROMP and weight-priors. K-priors provide a general probabilistic framework for model adaptation, and unlike our previous methods, they can be exact on Generalised Linear Models. We apply K-priors to many more adaptation tasks than just adding data (like in continual learning), such as removing data, changing the regulariser, and changing the model. We also use the K-priors framework to (i) improve weight-priors with function-regularisation, and (ii) understand why FROMP works well, providing suggestions to improve FROMP. When we apply K-priors to neural networks, we see a link with knowledge distillation (Hinton et al., 2015), and we can use this link to improve both K-priors and knowledge distillation. Experiments show K-priors performing quick and accurate adaptation across different adaptation tasks on a variety of models.

Finally, Chapter 6 provides conclusions, as well as summarising potential avenues for future research.

1.3 List of publications

During my degree, I have co-authored a number of peer-reviewed publications, listed here chronologically regardless of whether I discuss them in this thesis.

Understanding Expectation Propagation. Siddharth Swaroop, Richard E Turner (2017). *Advances in Approximate Bayesian Inference workshop at NIPS 2017.*

Partitioned variational inference: A unified framework encompassing federated and continual learning. Thang D Bui, Cuong V Nguyen, Siddharth Swaroop, Richard E Turner (2018). *arXiv preprint arXiv:1811.11206, Bayesian Deep Learning workshop at NeurIPS 2018 (spotlight).*

Neural network ensembles and variational inference revisited. Marcin B Tomczak, Siddharth Swaroop, Richard E Turner (2018). *Advances in Approximate Bayesian Inference Symposium 2018.*

Improving and Understanding Variational Continual Learning. Siddharth Swaroop, Thang D Bui, Cuong V Nguyen, Richard E Turner (2019). *Continual Learning workshop at NeurIPS 2018 (Oral presentation).*

Differentially Private Federated Variational Inference. Mrinank Sharma*, Michael Hutchinson*, Siddharth Swaroop, Antti Honkela, Richard E Turner (2019). *Privacy in Machine Learning workshop at NeurIPS 2019.*

Practical Deep Learning with Bayesian Principles. Kazuki Osawa, Siddharth Swaroop*, Anirudh Jain*, Runa Eschenhagen, Richard E Turner, Rio Yokota, Mohammad Emtiyaz Khan (2019). *Advances in Neural Information Processing Systems, 2019.*

Combining Variational Continual Learning with FiLM Layers. Noel Loo, Siddharth Swaroop, Richard E Turner (2020). *LifeLongML workshop at ICML 2020 (Oral presentation).*

Efficient Low Rank Gaussian Variational Inference for Neural Networks. Marcin B Tomczak, Siddharth Swaroop, Richard E Turner (2020). *Advances in Neural Information Processing Systems, 2020.*

Continual Deep Learning by Functional Regularisation of Memorable Past. Pingbo Pan*, Siddharth Swaroop*, Alexander Immer, Runa Eschenhagen, Richard E Turner, Mohammad Emtiyaz Khan (2020). *LifeLongML workshop at ICML 2020 (Oral presentation), Advances in Neural Information Processing Systems, 2020 (Oral presentation).*

Generalized Variational Continual Learning. Noel Loo, Siddharth Swaroop, Richard E Turner (2021). *International Conference on Learning Representations, 2021.*

Knowledge-Adaptation Priors. Mohammad Emtiyaz Khan*, Siddharth Swaroop* (2021). *Advances in Neural Information Processing Systems, 2021.*

Collapsed Variational Bounds for Bayesian Neural Networks. Marcin B Tomczak, Siddharth Swaroop, Andrew YK Foong, Richard E Turner (2021). *Advances in Neural Information Processing Systems, 2021.*

Partitioned Variational Inference: A framework for probabilistic federated learning. Matthew Ashman, Thang D Bui, Cuong V Nguyen, Stratis Markou, Adrian Weller, Siddharth Swaroop, Richard E Turner (2018). *arXiv preprint arXiv:2202.12275.*

Publications discussed in this thesis

Material from these publications is used in Chapters 3 to 5.

Chapter 3 is based on work from a few publications. The work on Improving VCL and observing pruning in Section 3.1 is from Swaroop et al. (2019), for which I was the first-author. The variational pruning effect in neural networks has been observed in many papers, and I also draw results from papers for which I was second-author (Loo et al., 2021; Tomczak et al., 2021). Section 3.2 is from Osawa et al. (2019). As joint second-author, I helped lead the project, and particularly contributed to the experiments testing the quality of predictive probabilities. Section 3.2.3 includes some experiments led by Runa Eschenhagen. Section 3.3 is an expansion on an Appendix from Pan et al. (2020), where I am joint first-author.

The work in Chapter 4 is mostly from Pan et al. (2020). As joint-first author, I contributed to all parts of the paper, including conceiving the original idea, deriving mathematical results, coding and running experiments. The results in Section 4.5.2 with OGN-FROMP are new. The Leverage method in Section 4.3 and the results in Section 4.5.1 are new experiments, part of an ongoing project on improving memorable past selection (see Section 6.2 for details).

Most of the work in Chapter 5 is from Khan and Swaroop (2021), for which I am joint first-author. However, Chapter 5 also significantly expands upon Khan and Swaroop (2021), such as by considering the variational setting in detail, comparing in detail to FROMP, and presenting new theory and algorithms for the limited-memory setting (such as Quadratic K-priors in Section 5.5). Some of these additional details are part of follow-up work and are in collaboration with Mohammad Emtiyaz Khan. The experiments applying K-priors to continual learning are part of ongoing work on K-priors for continual learning (see Section 6.2 for details).

Chapter 2

Background

We start this chapter by formally introducing continual learning in Section 2.1. We motivate and explain the probabilistic framework for continual learning in Section 2.2, where we also provide background on variational inference and natural-gradient variational inference. In Section 2.3, we summarise current methods for continual learning, categorising them in terms of three orthogonal approaches. In Section 2.4, we introduce the metrics and benchmarks we will use throughout this thesis to compare continual learning methods.

2.1 Continual learning

Neural networks perform very well in the *batch-setting*, where the full dataset is available at once: the method can sample in an independent and identically distributed fashion from the full dataset, and multiple passes are allowed through the dataset. In contrast, *continual learning* (also known as lifelong learning or sequential learning) considers continuous adaptation of machine learning systems when faced with a stream of changing data. Data examples arrive sequentially to a machine learning system, and we want the system to perform well over all data that has been fed to it. We formalise continual learning through a list of desiderata, similar to some past works (Schwarz et al., 2018; Hadsell et al., 2020; Lange et al., 2021). An ideal continual learning system should have the following properties:

1. Not store all past data (this can be made stricter to not store *any* past data), for example due to memory/computational constraints or due to privacy reasons;

2. Handle online learning, where new data examples are continuously presented to the system, with no assumptions on the structure of the data (for example, no fixed tasks or datasets, and no clear boundaries between tasks);

3. Avoid *catastrophic forgetting* (Robins, 1995; French, 1999) and *interference* (McCloskey and Cohen, 1989; Ratcliff, 1990), meaning learning on new data does not destroy performance on previously seen data;

4. Show *forward transfer*, the ability to leverage past information to perform better on new data;

5. Show *backward transfer*, the ability to improve performance on old data by using information from new data;

6. Model capacity and computation should be bounded (or scale well) as more data examples are seen, such that the system is scalable over long streams of data;

7. Maintain plasticity, meaning the model can continue to learn effectively as new data examples are seen.

The system should show these properties while learning quickly and being accurate. Although we ideally want a machine learning system to fulfil all of these desiderata, this is currently very difficult. In order to eventually reach this goal, current research typically aims to satisfy a subset of these desiderata at a time. This may also be reasonable as different real-world applications may only require different subsets of desiderata.

In this thesis, we aim to satisfy all but Desideratum 2: we do not consider the fully online setting, and we assume that data examples arrive in distinct tasks, where we are provided with knowledge of task boundaries. We also assume that best possible performance is given by a model that would have had access to all data at once. Other scenarios may assume that old data can become irrelevant or stale, but we assume older data examples are as important to remember as newer data examples. In Section 6.2 we discuss ways to move beyond such assumptions in future work.

Additionally, in this thesis, we mainly focus on neural networks (NNs). Neural networks are capable of learning complex relationships and are scalable to large numbers of data examples. We focus on image-recognition tasks, a supervised learning problem that convolutional neural networks (CNNs) perform very well on.

Notation. A deep neural network takes an input $\boldsymbol{x}$ and has a mapping $\boldsymbol{f_w}(\boldsymbol{x})$, with network parameters $\boldsymbol{w}$. We train a neural network by minimising a loss function with respect to the neural network's weights. This loss function measures the error between the neural network's output and the true (provided) label/output. For a supervised multi-class classification problem, we are given a dataset $\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$ of N input-output pairs with inputs

$\boldsymbol{x}_i \in \mathbb{R}^D$ and outputs $\boldsymbol{y}_i$, one-hot encoded vectors of K classes. We optimise the batch loss,

$$\ell^{\text{Batch}}(\boldsymbol{w}) = \underbrace{\sum_{i=1}^{N} \ell(\boldsymbol{y}_i, \boldsymbol{h}(\boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{x}_i)))}_{=N\bar{\ell}(\boldsymbol{w})} + \mathcal{R}(\boldsymbol{w}), \tag{2.1}$$

where $\ell(\boldsymbol{y}_i, \boldsymbol{h}(\boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{x}_i)))$ is the loss for a datapoint $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}$ when passed through a neural network $\boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{x}) \in \mathbb{R}^K$ with weights $\boldsymbol{w} \in \mathbb{R}^P$, and $\mathcal{R}(\boldsymbol{w})$ denotes a regularisation function, usually the L_2-regulariser, $\mathcal{R}(\boldsymbol{w}) = \frac{1}{2}\delta \boldsymbol{w}^\top \boldsymbol{w}$, where δ is the regularisation strength. The loss $\ell(\boldsymbol{y}, \boldsymbol{h}(\boldsymbol{f}))$ is a differentiable loss function (such as cross-entropy) between an output $\boldsymbol{y}$ and the neural network output $\boldsymbol{h}(\boldsymbol{f})$, where $\boldsymbol{h}$ is an inverse link function.

For binary classification with $y \in \{0, 1\}$, the loss function is $\ell(y, h(f)) = -y \log \sigma(f) - (1-y) \log(1 - \sigma(f))$, where $\sigma(f) = 1/\left(1 + e^{-f}\right)$ is the sigmoid function. This extends to multiclass classification with one-hot encoded labels $\boldsymbol{y}$. For a generic loss function derived from a Generalised Linear Model (GLM), the loss becomes $\ell(\boldsymbol{y}, \boldsymbol{h}(\boldsymbol{f})) = -\log p(\, \boldsymbol{y} \mid \boldsymbol{f})$. For binary classification in a GLM, we use a Bernoulli distribution, and $h(f) = \sigma(f)$.

In our continual learning scenario, we assume that T tasks arrive sequentially. Each task t has its own dataset $\mathcal{D}_t = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N_t}$, consisting of N_t datapoints. We assume we are allowed to train for as long as we want on $\mathcal{D}_t$, but we do not have access to all of $\mathcal{D}_{1:t-1}$. Our ideal loss function is the joint loss over all tasks up to and including task T (we earlier made this assumption explicitly),

$$\ell_T^{\text{Joint}}(\boldsymbol{w}) = \sum_{t=1}^{T} \sum_{i \in \mathcal{D}_t} \ell(\boldsymbol{y}_i, \boldsymbol{h}(\boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{x}_i))) + \mathcal{R}(\boldsymbol{w}). \tag{2.2}$$

We call this the Joint Tasks loss, and we could also describe it as retraining-from-scratch on all data, or the full-batch loss over all data. Optimising this loss requires access to all data (from all tasks) at once, and is therefore not possible in continual learning. Note that when summing over a dataset, we should write $(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{D}_t$, but we slightly abuse notation by writing the simpler $i \in \mathcal{D}_t$ instead.

Relationship to other learning scenarios

There are many learning scenarios that aim to transfer information between datasets, particularly in the low-data regime. We now quickly summarise their relationship to continual learning.

Multitask learning aims to learn multiple tasks at the same time, and relates to the Joint Tasks loss described earlier. Often, the tasks will be very different, potentially with different loss functions (such as classification and regression losses). **Transfer learning** and **domain adaptation** leverage information from a source task (the first task) to maximise performance on the new task (the second task). There can potentially be many different source tasks. The aim is to maximise forward transfer, and backward transfer or catastrophic forgetting are not important. **Few-shot learning** aims to quickly adapt to a few examples of new classes or data, and again backward transfer or catastrophic forgetting are not important. A common recent strand of methods approach this by meta-learning on a much larger dataset, instead of achieving adaptation in a purely online fashion. **Curriculum learning** re-orders datapoints when training a model on a dataset, aiming to present easier examples first, so that overall performance on the dataset is improved. In **active learning**, an algorithm chooses a set of datapoints to label from a large unlabelled set. The typical motivation is that obtaining supervised labels can be expensive, and so we should only label the most important datapoints. The algorithm then adapts to the new data, usually using the Joint Tasks loss, before choosing the next set of datapoints to be labelled. **Federated learning** assumes that the dataset is split spatially, and stored across many different clients. Datapoints are not allowed to leave the clients (perhaps due to security considerations), and we want to train a global model over all data in a communication-efficient manner. Each client's datapoints are allowed to be visited many times and in any order. Federated learning can be seen as a generalisation of continual learning (Bui et al., 2018): in continual learning, each client's (/task's) datapoints can only be visited once, and the clients must be seen in a specific order.

2.2 Probabilistic continual learning

Probabilistic machine learning uses probability distributions to quantify belief about objects of interest. In a parametric setting, where the model is defined by parameters $\boldsymbol{w}$, the prior $p(\boldsymbol{w})$ encodes subjective or previous information, the likelihood $p(\mathcal{D}|\boldsymbol{w})$ defines how the parameters are related to the observed data $\mathcal{D}$, and we perform inference to find our posterior

belief $p(\boldsymbol{w}|\mathcal{D})$ using Bayes' rule,

$$p(\boldsymbol{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})}, \tag{2.3}$$

where $p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{w})\, p(\boldsymbol{w})\, d\boldsymbol{w}$ is the marginal likelihood of the model. Note that in supervised learning, where the dataset $\mathcal{D}$ consists of inputs $\boldsymbol{X}$ and labels $\boldsymbol{y}$, we should write the likelihood as $p(\boldsymbol{y}|\boldsymbol{w}, \boldsymbol{X})$, but we slightly abuse notation by writing $p(\mathcal{D}|\boldsymbol{w})$. Throughout this thesis, we will use a turquoise colour to keep track of terms coming from the likelihood $p(\mathcal{D}|\boldsymbol{w})$, and a purple colour to keep track of terms coming from a prior $p(\boldsymbol{w})$. Note that the negative log likelihood of a datapoint is also the loss over the datapoint with a model $\boldsymbol{f}_{\boldsymbol{w}}$, $-\log p(\boldsymbol{y}_i|\boldsymbol{w}, \boldsymbol{x}_i) = \ell(\boldsymbol{y}_i, \boldsymbol{h}(\boldsymbol{f}_{\boldsymbol{w}}(\boldsymbol{x}_i)))$.

The concept of updating the prior into a posterior distribution using new data is ideal for continual learning. The prior distribution holds our knowledge about previous data, and is combined with new data to give us a posterior distribution. Mathematically, we can see this by writing Bayes' rule for just task 1, and then Bayes' rule for both tasks 1 and 2,

$$\begin{aligned}
\text{Task 1:} \qquad & p_1(\boldsymbol{w}|\mathcal{D}_1) && \propto p(\mathcal{D}_1|\boldsymbol{w})\, p(\boldsymbol{w}), \\
\text{Tasks 1 \& 2:} \qquad & p_2(\boldsymbol{w}|\mathcal{D}_1, \mathcal{D}_2) && \propto p(\mathcal{D}_2|\boldsymbol{w}) \underbrace{p(\mathcal{D}_1|\boldsymbol{w})\, p(\boldsymbol{w})}_{\propto p_1(\boldsymbol{w}|\mathcal{D}_1)} \\
& && \propto p(\mathcal{D}_2|\boldsymbol{w})\, p_1(\boldsymbol{w}|\mathcal{D}_1).
\end{aligned} \tag{2.4}$$

The likelihood term over the two datasets $p(\mathcal{D}_1, \mathcal{D}_2|\boldsymbol{w})$ decomposes as we assume datapoints are independently observed. By replacing the last two terms in the second line with the posterior after task 1, we obtain Equation 2.4. This equation is exactly Bayes' rule, and is the same as Equation 2.3 except with the prior $p(\boldsymbol{w})$ replaced with the previous posterior $p_1(\boldsymbol{w}|\mathcal{D}_1)$. By using the previous posterior, we do not directly need to store past data $\mathcal{D}_1$, as all relevant information is stored in the distribution over model parameters $p_1(\boldsymbol{w}|\mathcal{D}_1)$.

This generalises for task $t-1$ and task t,

$$p_t(\boldsymbol{w}|\mathcal{D}_{1:t}) \propto p(\mathcal{D}_t|\boldsymbol{w})\, p_{t-1}(\boldsymbol{w}|\mathcal{D}_{1:t-1}). \tag{2.5}$$

If we could perfectly compute Equation 2.5 for every new dataset $\mathcal{D}_t$, we would have solved continual learning.[1] Additionally, we could also gain many other benefits, such as correctly-calibrated uncertainties, robustness to overfitting, and good performance in the low-data regime (MacKay, 1992; Mackay, 1995; Neal, 1995; Gal, 2016).

[1] This assumes that a Bayesian solution over all data $\mathcal{D}_{1:T}$ is our ideal solution (this is similar to our assumption earlier that the Joint Tasks loss is our ideal loss function).

Unfortunately, Bayes' rule is intractable for most models (especially neural networks), and we need to make approximations. These approximations introduce errors, and these errors can add up as we recursively apply Bayes' rule. Our big challenge in probabilistic continual learning is to make approximations that minimise these errors, and we will discuss this throughout the thesis.

Approximations to Bayes' rule can be broadly split into Monte Carlo approaches and distributional approaches.

Monte Carlo approaches: These sample from the posterior, and have the desirable property that with enough compute power, the samples are from the exact posterior distribution. However, only having access to collections of (potentially weighted) point masses poses a problem in continual learning, where we set our new prior to be the previous posterior. Our number of samples would get smaller and smaller as we recursively approximate a set of point masses with another set of point masses. The closest method to such an idea is Sequential Monte-Carlo (Liu and Chen, 1998; Doucet et al., 2001), but it is not straightforward to apply such techniques outside of time-series models.

Distributional approaches: These approximate the Bayesian posterior by introducing an approximate posterior $q(\boldsymbol{w}) \approx p(\boldsymbol{w}|\mathcal{D})$ that belongs to a simpler variational family, such as Gaussian distributions. The approximate posterior typically has support everywhere (as opposed to Monte Carlo approaches). By dealing with these simpler distributions, we can easily calculate (an approximation to) our model evidence, which previously involved a difficult (often intractable) integration. The Laplace approximation is one such approach. We first train for a Maximum-A-Posteriori (MAP) estimate of model weights (such as by optimising Equation 2.1), and then use a Gaussian approximate posterior, with mean given by the MAP estimate, and covariance given as the Hessian of the loss around the mean (with a prior term). Variational inference is another distributional approach, and we discuss it next.

2.2.1 Variational inference

We now describe Variational Inference (VI) (Blei et al., 2017; Zhang et al., 2019), which we will use extensively in this thesis. VI calculates the parameters of the approximate posterior $q(\boldsymbol{w})$ by minimising the Kullback-Leibler (KL) divergence (Kullback and Leibler, 1951) between the approximate posterior and the true posterior, $\mathcal{KL}\left[q(\boldsymbol{w})||p(\boldsymbol{w}|\mathcal{D})\right]$. We can re-write this optimisation as minimising the negative Evidence Lower Bound (ELBO),

$$\mathcal{L}^{\text{ELBO}}(q) = \underbrace{\mathbb{E}_{q(\boldsymbol{w})}\left[-\log p(\mathcal{D}|\boldsymbol{w})\right]}_{\text{Likelihood term}} + \underbrace{\mathbb{E}_{q(\boldsymbol{w})}\left[\log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w})}\right]}_{\text{KL-to-prior term}}, \tag{2.6}$$

where we note the KL-to-prior term can also be written as $\mathcal{KL}\left[q(w) \,\|\, p(w)\right]$. .

In addition, we assume that $q(\boldsymbol{w})$ belongs to an exponential family distribution in this thesis. Exponential family distributions include Gaussian distributions, which we will use extensively. An exponential family distribution over parameters $\boldsymbol{w}$ with natural parameters $\boldsymbol{\eta}$ takes the following form,

$$q(\boldsymbol{w}|\boldsymbol{\eta}) = q_{\boldsymbol{\eta}}(\boldsymbol{w}) = h(\boldsymbol{w})\, \exp[\langle \boldsymbol{\eta}, \boldsymbol{\phi}(\boldsymbol{w})\rangle - A(\boldsymbol{\eta})], \tag{2.7}$$

where $\boldsymbol{\phi}(\boldsymbol{w})$ is the vector of sufficient statistics, $\langle \cdot, \cdot \rangle$ is an inner product, $A(\boldsymbol{\eta})$ is the log-partition function and $h(\boldsymbol{w})$ is the base measure. We also assume a *minimal* exponential family, meaning the sufficient statistics are linearly independent. This results in a one-to-one mapping between the natural parameters $\boldsymbol{\eta}$ and the mean parameters $\boldsymbol{m} = \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}[\boldsymbol{\phi}(\boldsymbol{w})]$, and also means that $\boldsymbol{m} = \nabla_{\boldsymbol{\eta}} A(\boldsymbol{\eta})$.

When we combine variational inference (Equation 2.6) for exponential family distributions (Equation 2.7) with Bayes' rule for continual learning (Equation 2.5), we get variational inference for continual learning,

$$\mathcal{L}_t(\boldsymbol{\eta}_t) = \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[-\log p(\mathcal{D}_t|\boldsymbol{w})\right] + \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[\log \frac{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}{q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})}\right], \tag{2.8}$$

where we are learning the parameters $\boldsymbol{\eta}_t$ of our current approximate posterior $q_{\boldsymbol{\eta}_t}(\boldsymbol{w})$, and $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$ is the previous approximate posterior, and is our prior in the ELBO.

This thesis extensively uses Equation 2.8. In Chapter 3 we directly optimise this equation in *weight-space* (or parameter-space). This is how Equation 2.8 is currently written, as it is only in terms of $\boldsymbol{\eta}_t$ and $\boldsymbol{w}$. Then in later chapters we consider replacing some terms with *function-space* regularisation.

Variational Continual Learning (VCL) (Nguyen et al., 2018)

Variational Continual Learning (VCL) applies Equation 2.8 on neural networks in weight-space. Running on neural networks requires additional approximations, and VCL optimises the ELBO with the Bayes-By-Backprop (BBB) (Blundell et al., 2015) method, a popular method for running (approximate) VI on neural networks. This involves (i) using a mean-field Gaussian variational approximating family $q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, where $\boldsymbol{\Sigma}_t$ is a diagonal matrix, (ii) using a Monte-Carlo approximation of the likelihood term, and (iii) analytically calculating the KL-to-prior term, which is possible as both terms in the KL divergence are

mean-field multivariate Gaussian distributions. This gives the objective,

$$\mathcal{L}_t^{\mathrm{VCL}}(\boldsymbol{\eta}_t) = \sum_{i \in \mathcal{D}_t} \frac{1}{S} \sum_{s=1}^{S} \left[-\log p(\boldsymbol{y}_i | \boldsymbol{w}^{(s)}, \boldsymbol{x}_i) \right] + \underbrace{\mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})} \left[\log \frac{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}{q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})} \right]}_{\text{Analytically calculated}}, \tag{2.9}$$

where we have used independence of the likelihood term, assumed supervised learning on the dataset $\mathcal{D}_t = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N_t}$, and $\boldsymbol{w}^{(s)} \sim q_{\boldsymbol{\eta}_t}(\boldsymbol{w})$ are samples from the current approximate posterior. VCL is a weight-prior approach as the prior $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$ is always in weight-space.

VCL was found to perform very well on some continual learning benchmarks. Nguyen et al. (2018) applied VCL to both discriminative models and generative models, but in this thesis we only look at discriminative models. In Chapter 3 we start by improving VCL's performance and analysing why it performs well.

VCL + Coreset

Nguyen et al. (2018) also describe the VCL+Coreset method, which stores a subset of datapoints from each task, using them later in order to improve VCL's performance. Here, a coreset refers to a subset of the whole dataset, with datapoints chosen using any method (for example, randomly choosing datapoints).

Before training on data from a new task t, a coreset C_t is produced by selecting datapoints from the current dataset $\mathcal{D}_t$ and the old coreset C_{t-1}. The coreset C_t is set aside, and the VCL+Coreset algorithm trains on $C_{t-1} \cup \mathcal{D}_t \backslash C_t$ to give an approximate posterior $\bar{q}_{\boldsymbol{\eta}_t}(\boldsymbol{w})$, which approximates the posterior $p(\boldsymbol{w}|\mathcal{D}_{1:t}\backslash C_t)$. We have used the notation $\mathcal{D}_{1:t}\backslash C_t$ to indicate removing the points in C_t from $\mathcal{D}_{1:t}$. When it comes to test-time, we first train on just C_t to help mitigate forgetting on datapoints in C_t. This step uses the VCL objective function (Equation 2.9) except with $\bar{q}_{\boldsymbol{\eta}_t}(\boldsymbol{w})$ as the prior and C_t as the dataset.

When training on a new task, a new coreset C_t is chosen, and $\bar{q}_{\boldsymbol{\eta}_t}(\boldsymbol{w})$ is used as the prior. This ensures that, when it comes to making predictions, a correct $q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) \approx p(\boldsymbol{w}|\mathcal{D}_{1:t})$ is always used, with no overcounting (or undercounting) of data.

This way of using a coreset can also be seen as a message-passing scheme where the update for coreset datapoints is scheduled after updating on the other data (Winn and Bishop, 2005; Bui et al., 2018).

2.2.2 Natural-gradient variational inference

In this section we look at performing variational inference with natural-gradient updates. We motivate why we might want to do this, and derive the VOGN algorithm.

Motivation

Bayes-By-Backprop (and VCL) optimise Equation 2.6 directly for the parameters of their variational approximating family, which for mean-field Gaussians is the mean $\boldsymbol{\mu}$ and diagonal covariance matrix $\boldsymbol{\Sigma}$. But this has been very difficult to scale to large neural networks (such as ResNets (He et al., 2016)). This is because optimisation is restrictively slow, requiring many passes through the data. As we will see in Chapter 3, this is especially true in continual learning, where training for a long time is important to improve results.

Natural-gradient (NG) update steps are a principled way of incorporating the information geometry of the distribution being optimised (Amari, 1998). By incorporating the geometry of the distribution, we expect to take gradient steps in much better directions, speeding up optimisation. We can see this by re-writing standard-gradient descent, and comparing to natural-gradient descent.

At every iteration j, standard-gradient descent takes a step in the direction of the gradient, $\boldsymbol{\eta}_{j+1} \leftarrow \boldsymbol{\eta}_j - \rho_j \left[\hat{\nabla}_{\boldsymbol{\eta}} \mathcal{L}(\boldsymbol{\eta}_j)\right]$, where ρ_j is the learning rate, j indexes iteration, and $\hat{\nabla}$ indicates a stochastic estimate of the gradient ∇. When we re-write this update equation, and compare to the NGD update step, we immediately see a difference,

Standard-gradient descent:

$$\boldsymbol{\eta}_{j+1} \leftarrow \arg\min_{\boldsymbol{\eta}} \; \boldsymbol{\eta}^\top \left[\hat{\nabla}_{\boldsymbol{\eta}} \mathcal{L}(\boldsymbol{\eta}_j)\right] + \frac{1}{2\rho_j} ||\boldsymbol{\eta} - \boldsymbol{\eta}_j||^2. \tag{2.10}$$

Natural-gradient descent:

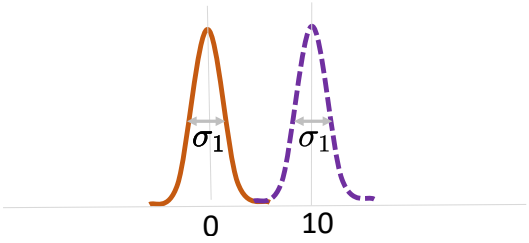
$$\boldsymbol{\eta}_{j+1} \leftarrow \arg\min_{\boldsymbol{\eta}} \; \boldsymbol{\eta}^\top \left[\hat{\nabla}_{\boldsymbol{\eta}} \mathcal{L}(\boldsymbol{\eta}_j)\right] + \frac{1}{2\rho_j} (\boldsymbol{\eta} - \boldsymbol{\eta}_j)^\top \left[\boldsymbol{F}(\boldsymbol{\eta}_j)\right] (\boldsymbol{\eta} - \boldsymbol{\eta}_j), \tag{2.11}$$

where the Fisher Information Matrix (FIM) $\boldsymbol{F}(\boldsymbol{\eta})$ is a Riemannian metric induced by the exponential family $q_{\boldsymbol{\eta}}(\boldsymbol{w})$, defined as $\boldsymbol{F}(\boldsymbol{\eta}) = \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\nabla_{\boldsymbol{\eta}} \log q_{\boldsymbol{\eta}}(\boldsymbol{w}) \, \nabla_{\boldsymbol{\eta}} \log q_{\boldsymbol{\eta}}(\boldsymbol{w})^\top\right]$.

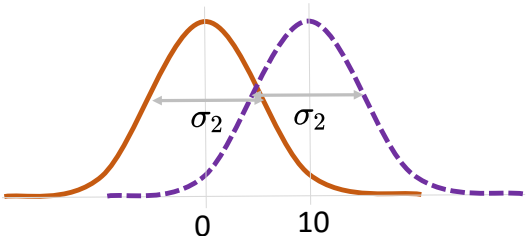
Standard-gradient descent moves in the direction of the gradient while remaining close to the previous parameters $\boldsymbol{\eta}_j$ in *Euclidean space*, while natural-gradient descent remains close by using a Riemannian metric. By incorporating the information geometry in this way, we expect natural-gradient to take steps in better directions. See Figure 2.1 for intuition on how Euclidean distances may not be ideal.

Overall, (stochastic) natural-gradient descent has the update step (consider solving Equation 2.11),

$$\boldsymbol{\eta}_{j+1} \leftarrow \boldsymbol{\eta}_j - \rho_j \, \boldsymbol{F}(\boldsymbol{\eta}_j)^{-1} \left[\hat{\nabla}_{\boldsymbol{\eta}} \mathcal{L}(\boldsymbol{\eta}_j)\right]. \tag{2.12}$$



(a) Two Gaussians with mean 1 and 10 respectively, and variances equal to σ_1, have Euclidean distance = 10.



(b) Same as Figure (a) on the left, except now with variance $\sigma_2 > \sigma_1$. They still have Euclidean distance = 10.

Figure 2.1: This figure is reproduced from Khan and Nielsen (2018), illustrating how Euclidean distances are a poor metric to measure distances between distributions. The distributions in Figure (a) on the left barely overlap, while the distributions in Figure (b) on the right are much closer, and yet the Euclidean distances are the same.

We can simplify this update step by avoiding calculating the Fisher Information Matrix directly, using a property of minimal exponential families (Hoffman et al., 2013; Khan and Lin, 2017),

$$\nabla_{\boldsymbol{\eta}} \mathcal{L}(\boldsymbol{\eta}_j) = [\nabla_{\boldsymbol{\eta}} \boldsymbol{m}_j] \nabla_{\boldsymbol{m}} \mathcal{L}_*(\boldsymbol{m}_j) = \left[\nabla^2_{\boldsymbol{\eta}\boldsymbol{\eta}} A(\boldsymbol{\eta}_j)\right] \nabla_{\boldsymbol{m}} \mathcal{L}_*(\boldsymbol{m}_j) = \boldsymbol{F}(\boldsymbol{\eta}_j) \nabla_{\boldsymbol{m}} \mathcal{L}_*(\boldsymbol{m}_j), \tag{2.13}$$

where $\mathcal{L}_*(\boldsymbol{m}_j)$ is the same function as $\mathcal{L}(\boldsymbol{\eta}_j)$ except written as a function of the mean parameters $\boldsymbol{m}$. We also used the equivalence $\boldsymbol{F}(\boldsymbol{\eta}) = \nabla^2_{\boldsymbol{\eta}\boldsymbol{\eta}} A(\boldsymbol{\eta})$ for minimal exponential families. This leads to our final simplified natural-gradient descent update step,

$$\boldsymbol{\eta}_{j+1} \leftarrow \boldsymbol{\eta}_j - \rho_j \left[\hat{\nabla}_{\boldsymbol{m}} \mathcal{L}_*(\boldsymbol{m}_j)\right]. \tag{2.14}$$

We will use this update to derive algorithms next.

Natural-gradient VI for neural networks

We now derive Variational Online Gauss-Newton (VOGN), a natural-gradient variational inference (NGVI) algorithm for neural networks. We follow the appendices of Khan et al. (2018) to derive VOGN, which builds on Khan and Lin (2017). In Chapter 3 we will scale VOGN to large datasets and architectures, which standard-gradient VI methods struggle to scale to. There is a slightly different derivation to similar algorithms in Zhang et al. (2018), where they compare with natural-gradients for maximum-likelihood estimation. Instead, the derivation presented here starts with the variational objective directly.

We start our derivation by plugging the VI objective function (Equation 2.6) into the natural-gradient update step (Equation 2.14). Let the prior be an exponential family (Equation 2.7) with natural parameters $\boldsymbol{\eta}_0$. We first note that the KL-to-prior term in the ELBO can be simplified,

$$\begin{aligned}
\nabla_{\boldsymbol{m}} \left[\text{KL-to-prior term}\right] &= \nabla_{\boldsymbol{m}} \mathbb{E}_{q_\eta(\boldsymbol{\theta})} \left[\boldsymbol{\phi}(\boldsymbol{\theta})^\top (\boldsymbol{\eta} - \boldsymbol{\eta}_0) - A(\boldsymbol{\eta}) + \text{const}\right] \\
&= \nabla_{\boldsymbol{m}} \left[\boldsymbol{m}^\top (\boldsymbol{\eta} - \boldsymbol{\eta}_0)\right] - \nabla_{\boldsymbol{m}} A(\boldsymbol{\eta}) \\
&= \boldsymbol{\eta} - \boldsymbol{\eta}_0 + \left[\nabla_{\boldsymbol{m}} \boldsymbol{\eta}\right]^\top \boldsymbol{m} - \nabla_{\boldsymbol{m}} A(\boldsymbol{\eta}) \\
&= \boldsymbol{\eta} - \boldsymbol{\eta}_0 + \mathbf{F}(\boldsymbol{\eta})^{-1} \boldsymbol{m} - \mathbf{F}(\boldsymbol{\eta})^{-1} \boldsymbol{m} \\
&= \boldsymbol{\eta} - \boldsymbol{\eta}_0. \qquad (2.15)
\end{aligned}$$

The third line follows using the product rule, and the fourth line uses $\nabla_{\boldsymbol{m}}(\cdot) = \mathbf{F}(\boldsymbol{\eta})^{-1} \nabla_{\boldsymbol{\eta}}(\cdot)$ (Equation 2.13) and the symmetry of the Fisher Information Matrix. Plugging the VI objective function (Equation 2.6) into the natural-gradient update step (Equation 2.14) gives,

$$\begin{aligned}
\boldsymbol{\eta}_{j+1} &\leftarrow \boldsymbol{\eta}_j - \rho_j \left(\nabla_{\boldsymbol{m}} \mathbb{E}_{q_{\boldsymbol{\eta}_j}(\boldsymbol{w})} \left[-\log p(\mathcal{D}|\boldsymbol{w})\right] + (\boldsymbol{\eta}_j - \boldsymbol{\eta}_0) \right) \\
\therefore \boldsymbol{\eta}_{j+1} &\leftarrow (1 - \rho_j) \boldsymbol{\eta}_j + \rho_j \Big(\boldsymbol{\eta}_0 + \nabla_{\boldsymbol{m}} \underbrace{\mathbb{E}_{q_{\boldsymbol{\eta}_j}(\boldsymbol{w})} \left[\log p(\mathcal{D}|\boldsymbol{w})\right]}_{\mathcal{F}_j} \Big). \qquad (2.16)
\end{aligned}$$

We now consider a Gaussian approximating family, $q_{\boldsymbol{\eta}}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$. The minimal representation for a Gaussian family has two components to its natural parameters and mean parameters,

$$\boldsymbol{\eta}^{(1)} = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}, \qquad \boldsymbol{\eta}^{(2)} = -\tfrac{1}{2} \boldsymbol{\Sigma}^{-1}, \qquad (2.17)$$

$$\boldsymbol{m}^{(1)} = \boldsymbol{\mu}, \qquad \boldsymbol{m}^{(2)} = \boldsymbol{\mu}\boldsymbol{\mu}^\top + \boldsymbol{\Sigma}. \qquad (2.18)$$

Let the prior be a Gaussian, $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \Sigma_0)$. We can therefore write the prior natural parameters as $\boldsymbol{\eta}_0^{(1)} = \boldsymbol{\Sigma}_0^{-1} \boldsymbol{\mu}_0$, $\boldsymbol{\eta}_0^{(2)} = -\frac{1}{2} \Sigma_0^{-1}$. We can also simplify the likelihood term $\nabla_{\boldsymbol{m}} \mathcal{F}_j$ to be in terms of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ instead of $\boldsymbol{m}$, using the chain rule (Opper and Archambeau, 2009; Khan and Lin, 2017),[2]

$$\nabla_{\boldsymbol{m}^{(1)}} \mathcal{F} = \nabla_{\boldsymbol{\mu}} \mathcal{F} - 2 \left[\nabla_{\boldsymbol{\Sigma}} \mathcal{F}\right] \boldsymbol{\mu}, \qquad (2.19)$$

$$\nabla_{\boldsymbol{m}^{(2)}} \mathcal{F} = \nabla_{\boldsymbol{\Sigma}} \mathcal{F}. \qquad (2.20)$$

[2]These chain rule equations hold for any function of $\boldsymbol{w}$, and are not specific to $\mathcal{F}$.

This allows us to write down our NGVI updates (Equation 2.16) for the parameters of a Gaussian, in terms of the prior parameters and the data $\mathcal{F}_j$,

$$\boldsymbol{\Sigma}_{j+1}^{-1} \leftarrow (1-\rho_j)\boldsymbol{\Sigma}_j^{-1} + \rho_j(\boldsymbol{\Sigma}_0^{-1} - 2\nabla_{\boldsymbol{\Sigma}}\mathcal{F}_j), \tag{2.21}$$

$$\begin{aligned}\boldsymbol{\Sigma}_{j+1}^{-1}\boldsymbol{\mu}_{j+1} &= (1-\rho_j)\boldsymbol{\Sigma}_j^{-1}\boldsymbol{\mu}_j + \rho_j(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \nabla_{\boldsymbol{\mu}}\mathcal{F}_j - 2[\nabla_{\boldsymbol{\Sigma}}\mathcal{F}_j]\boldsymbol{\mu}_j) \\ &= \underbrace{\left[(1-\rho_j)\boldsymbol{\Sigma}_j^{-1} + \rho_j(\boldsymbol{\Sigma}_0^{-1} - 2\nabla_{\boldsymbol{\Sigma}}\mathcal{F}_j)\right]}_{=\boldsymbol{\Sigma}_{j+1}^{-1}, \text{ by Equation 2.21}}\boldsymbol{\mu}_j + \rho_j(\nabla_{\boldsymbol{\mu}}\mathcal{F}_j - \boldsymbol{\Sigma}_0^{-1}(\boldsymbol{\mu}_j - \boldsymbol{\mu}_0)) \\ \therefore \boldsymbol{\mu}_{j+1} &\leftarrow \boldsymbol{\mu}_j - \rho_j\boldsymbol{\Sigma}_{j+1}(-\nabla_{\boldsymbol{\mu}}\mathcal{F}_j + \boldsymbol{\Sigma}_0^{-1}(\boldsymbol{\mu}_j - \boldsymbol{\mu}_0)).\end{aligned} \tag{2.22}$$

Finally, we use Bonnet's and Price's theorems (Opper and Archambeau, 2009; Rezende et al., 2014) to calculate terms involving the data $\mathcal{F}_j$, allowing us to move the derivatives inside the expectation,

$$\nabla_{\boldsymbol{\mu}}\mathcal{F}_j = \mathbb{E}_{q_{\boldsymbol{\eta}_j}(\boldsymbol{w})}\left[\nabla_{\boldsymbol{w}}\log p(\mathcal{D}|\boldsymbol{w})\right] = -\mathbb{E}_{q_{\boldsymbol{\eta}_j}(\boldsymbol{w})}\left[N\mathbf{g}(\boldsymbol{w})\right], \tag{2.23}$$

$$\nabla_{\boldsymbol{\Sigma}}\mathcal{F}_j = \frac{1}{2}\mathbb{E}_{q_{\boldsymbol{\eta}_j}(\boldsymbol{w})}\left[\nabla^2_{\boldsymbol{ww}}\log p(\mathcal{D}|\boldsymbol{w})\right] = -\frac{1}{2}\mathbb{E}_{q_{\boldsymbol{\eta}_j}(\boldsymbol{w})}\left[N\mathbf{H}(\boldsymbol{w})\right], \tag{2.24}$$

where we define the per-example gradient $\mathbf{g}(\boldsymbol{w}) = -\frac{1}{N}\nabla_{\boldsymbol{w}}\log p(\mathcal{D}|\boldsymbol{w})$ and the per-example Hessian $\mathbf{H}(\boldsymbol{w}) = -\frac{1}{N}\nabla^2_{\boldsymbol{ww}}\log p(\mathcal{D}|\boldsymbol{w})$, where we earlier defined the size of the dataset $N = |\mathcal{D}|$. Like in Bayes-By-Backprop, we calculate these at a single Monte-Carlo sample from the current approximate posterior $\boldsymbol{w}_j \sim q_{\boldsymbol{\eta}_j}(\boldsymbol{w})$. It is possible to average over many Monte-Carlo samples, but we write for a single sample here for ease of notation.

We also use a Gauss-Newton approximation of the Hessian (Schraudolph, 2002; Graves, 2011; Martens, 2020),[3] $\mathbf{H}(\boldsymbol{w}_j) = -\frac{1}{N}\nabla^2_{\boldsymbol{ww}}\log p(\mathcal{D}|\boldsymbol{w}) \approx \frac{1}{N}\sum_{i\in\mathcal{D}}\mathbf{g}_i(\boldsymbol{w}_j)\,\mathbf{g}_i(\boldsymbol{w}_j)^\top$, where $\mathbf{g}_i(\boldsymbol{w}_j) = -\nabla_{\boldsymbol{w}}\log p(\boldsymbol{y}_i|\boldsymbol{w}_j, \boldsymbol{x}_i)$. The Gauss-Newton matrix has some nice properties, such as being positive semi-definite (which we require), and becoming a better approximation of the Hessian as we train for longer and the training error reduces.

Finally, in order to scale this to even small neural networks, we make additional approximations and simplifications: (i) we use a stochastic minibatch $\mathcal{B}_j$ of size B, with average gradient $\hat{\mathbf{g}}(\boldsymbol{w}_j) = \frac{1}{B}\sum_{i\in\mathcal{B}_j}\mathbf{g}_i(\boldsymbol{w}_j)$; (ii) we re-parameterise the update equations to be in terms of $\boldsymbol{S}_j = (\boldsymbol{\Sigma}_j^{-1} - \boldsymbol{\Sigma}_0^{-1})/N$; (iii) like in Bayes-By-Backprop, we use mean-field

[3] A side note on Generalised Gauss-Newton (GGN) approximations: GGN approximations are a general class of techniques for approximating the Hessian. Technically, we are using a GGN approximation with a non-standard parameterisation of the loss function (for example, see Equation 6.17 in Bottou et al. (2018)). This turns out to be mathematically equivalent to the empirical Fisher matrix. We simply call this a Gauss-Newton approximation, but we could also call it a Fisher approximation or a GGN approximation. Later in Chapter 4, we will see a different GGN parameterisation in the VOGGN algorithm (Khan et al., 2019).

Gaussians, $\boldsymbol{S}_j = \text{diag}(\boldsymbol{s}_j)$, where $\text{diag}(\boldsymbol{s})$ denotes a diagonal matrix with $\boldsymbol{s}$ as the diagonal, allowing the diagonal Gauss-Newton matrix to be calculated as $\mathbf{H}(\boldsymbol{w}) \approx \frac{1}{B}\sum_{i\in\mathcal{B}_j}(\mathbf{g}_i(\boldsymbol{w}_j)^2)$; (iv) we use separate learning rates α_j, β_j in the update equations for $\boldsymbol{\mu}_j$, $\boldsymbol{s}_j$, instead of a single shared ρ_j.

These changes lead to the VOGN algorithm (Khan et al., 2018), where to reduce clutter we have also assumed a zero-mean constant-variance Gaussian prior, $\boldsymbol{\mu}_0 = \mathbf{0}$, $\boldsymbol{\Sigma}_0 = \delta^{-1}\mathbf{I}$,

$$\boldsymbol{\mu}_{j+1} \leftarrow \boldsymbol{\mu}_j - \alpha_j \frac{\hat{\mathbf{g}}(\boldsymbol{w}_j) + \tilde{\delta}\boldsymbol{\mu}_j}{\boldsymbol{s}_{j+1} + \tilde{\delta}\mathbf{1}}, \tag{2.25}$$

$$\boldsymbol{s}_{j+1} \leftarrow (1-\beta_j)\boldsymbol{s}_j + \beta_j \frac{1}{B}\sum_{i\in\mathcal{B}_j}\left(\mathbf{g}_i(\boldsymbol{w}_j)^2\right), \tag{2.26}$$

where $\tilde{\delta} = \delta/N$, $\mathbf{1}$ is a vector of 1s, and all operations are element-wise. Khan and Nielsen (2018) showed that this algorithm can converge much quicker than Bayes-By-Backprop to a similar solution on small multi-layer-perceptrons on some UCI data (Dua and Graff, 2017). In Section 3.2 we will scale this to much larger neural networks and datasets.

2.3 Approaches to continual learning

We have introduced continual learning, and have looked the main methods we will use in this thesis to approach the problem: probabilistic continual learning, usually via variational inference. We now briefly summarise other approaches, and how they attempt to tackle the continual learning problem (the reader is also referred to Parisi et al. (2019) for a review).

Current methods for continual learning can be classified into three orthogonal approaches: **regularisation-based approaches** regularise parameter updates by penalising against changes in 'important' parameters for previous tasks, **rehearsal/memory-based approaches** rehearse some past data or pseudo-data, and **architecture-based approaches** change the model architecture or mask model weights. These approaches are complementary and they can be combined, and recent works tend to do so in order to combine advantages of different approaches. We summarise many related works within these three approaches in Figure 2.2, and next discuss a few representative methods in more detail. This thesis discusses regularisation-based approaches in Chapter 3, and then discusses hybrid regularisation and rehearsal-based approaches in Chapters 4 and 5.

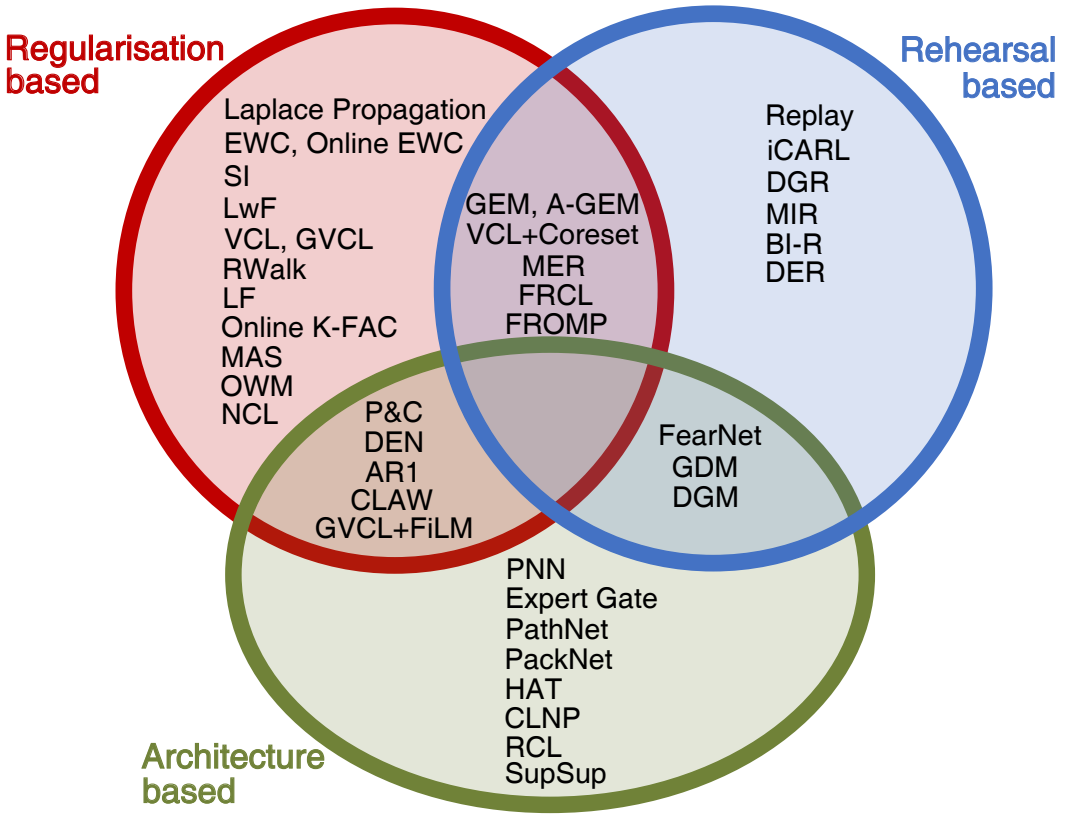


Figure 2.2: Methods for continual learning can be classified into three orthogonal approaches, shown here using a Venn diagram. These approaches are complementary and they can be combined, leading to methods that belong to more than one approach. No approach yet combines all three approaches (the centre of the diagram).

Laplace Propagation (Smola et al., 2004); EWC: Elastic Weight Consolidation (Kirkpatrick et al., 2017); Online EWC (Schwarz et al., 2018); SI: Synaptic Intelligence (Zenke et al., 2017); LwF: Learning without Forgetting (Li and Hoiem, 2016); VCL(+Coreset): Variational Continual Learning (+ Coreset) (Nguyen et al., 2018); GVCL(+FiLM): Generalized Variational Continual Learning (+ FiLM layers) (Loo et al., 2021); RWalk: Riemannian Walk (Chaudhry et al., 2018); LF: Less-Forgetful learning (Jung et al., 2018); Online K-FAC (Ritter et al., 2018); MAS: Memory Aware Synapses (Aljundi et al., 2018); OWM: Orthogonal Weights Modification (Zeng et al., 2019); NCL: Natural Continual Learning (Kao et al., 2021); GEM: Gradient Episodic Memory (Lopez-Paz and Ranzato, 2017); A-GEM: Averaged GEM (Chaudhry et al., 2019); MER: Meta-Experience Replay (Riemer et al., 2019); FRCL: Functional Regularisation for Continual Learning (Titsias et al., 2020); FROMP: Functional Regularisation of Memorable Past (Pan et al., 2020); Replay (or Experience Replay) (Ratcliff, 1990; Robins, 1995; Rolnick et al., 2019); iCARL: Incremental Classifier and Representation Learning (Rebuffi et al., 2017); DGR: Deep Generative Replay (Shin et al., 2017); MIR: Maximal Interfered Retrieval (Aljundi et al., 2019a); BI-R: Brain-Inspired Replay (van de Ven et al., 2020); DER: Dark Experience Replay (Buzzega et al., 2020); P&C: Progress & Compress (Schwarz et al., 2018); DEN: Dynamically Expandable Networks (Yoon et al., 2018); AR1 (Maltoni and Lomonaco, 2019); CLAW: Continual Learning with Adaptive Weights (Adel et al., 2020); FearNet (Kemker and Kanan, 2018); GDM: Growing Dual-Memory architecture (Parisi et al., 2018); DGM: Dynamic Generative Memory (Ostapenko et al., 2019); PNN: Progressive Neural Networks (Rusu et al., 2016); Expert Gate (Aljundi et al., 2017); PathNet (Fernando et al., 2017); PackNet (Mallya and Lazebnik, 2018); HAT: Hard Attention to the Task (Serra et al., 2018); CLNP: Continual Learning via Neural Pruning (Golkar et al., 2019); RCL: Reinforced Continual Learning (Xu and Zhu, 2018); SupSup: Supermasks in Superposition (Wortsman et al., 2020).

Laplace Propagation (Smola et al., 2004), Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017) and Online EWC (Schwarz et al., 2018). These are probabilistic weight-prior approaches that use the Laplace approximation. After training on a task, they approximate the Bayesian posterior using a Laplace approximation, approximating the Hessian of the neural network with the (diagonal) Fisher information matrix. Laplace Propagation (Smola et al., 2004), when applied to continual learning, calculates the new approximate posterior's precision as the sum of the current Hessian and previous precision, $\boldsymbol{\Sigma}_t^{-1} = \mathbf{H}_t + \boldsymbol{\Sigma}_{t-1}^{-1}$, where the initial precision $\boldsymbol{\Sigma}_0^{-1}$ is the prior precision (the L_2-regulariser δ), and $\mathbf{H}_t$ is the approximate Hessian calculated over data $\mathcal{D}_t$ using converged weights $\boldsymbol{w}_t$. When training on a new task for parameters $\boldsymbol{w}$, the regulariser is then $\frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_{t-1})^\top \boldsymbol{\Sigma}_{t-1}^{-1}(\boldsymbol{w} - \boldsymbol{w}_{t-1})$. Smola et al. (2004) did not apply to neural networks.

EWC (Kirkpatrick et al., 2017) achieves strong results on neural networks, using well-known identities of the Fisher information matrix to ease computation of $\mathbf{H}_t$. EWC uses a separate regulariser per task, requiring storing many $\boldsymbol{w}_t$ and $\boldsymbol{\Sigma}_t^{-1} = \mathbf{H}_t$. Additionally, EWC upweights the regularisation term by a hyperparameter λ, which is often very large to obtain good overall performance. By using separate regulariser per task, EWC does not follow the online Bayesian update (Huszár, 2018), unlike Laplace Propagation. Online EWC (Schwarz et al., 2018) uses an online update for $\boldsymbol{\Sigma}_t^{-1}$ like in Laplace Propagation, but they also multiply $\boldsymbol{\Sigma}_t^{-1}$ by a scalar hyperparameter to improve performance.

Synaptic Intelligence (SI) (Zenke et al., 2017). SI uses a different method to compute the importance of each parameter, looking at each parameter's contribution to the decrease in loss during training. This can be seen as a different way of calculating the diagonal $\boldsymbol{\Sigma}_t^{-1}$ from Laplace approximations. Riemannian Walk (Chaudhry et al., 2018) combines Online EWC and SI. Although these methods perform well, the variational Bayesian approach of Nguyen et al. (2018) (discussed in Section 2.2.1) performs better on many benchmarks.

Learning without Forgetting (LwF) (Li and Hoiem, 2016). LwF is a regularisation-based approach that adds a knowledge distillation loss term during training. This term is calculated on inputs from the current task $\mathcal{D}_t$, with soft labels given by passing these inputs through the model trained on the previous task, $\boldsymbol{w}_{t-1}$. Although LwF performs well when tasks are similar, it can perform worse when tasks are very different. We discuss LwF further in Section 5.2.

Generalised VCL + FiLM layers (GVCL+FiLM) (Loo et al., 2021). As discussed in Section 2.2.1, VCL is a weight-prior approach that uses the variational objective. Generalised

VCL introduces a hyperparameter in front of the KL-to-prior term in VCL (see Equation 2.9). They show that this modification to VCL recovers Online EWC (Schwarz et al., 2018) as a limiting case, allowing for interpolation between the two methods. This leads to increased performance on several benchmarks. Loo et al. (2021) also introduce task-specific FiLM layers (Perez et al., 2018) to take advantage of and reduce pruning in variational Bayesian neural networks, finding that this also leads to improved performance at the cost of a slightly-increased memory. By introducing FiLM layers, GVCL+FiLM is a hybrid regularisation and architecture-based approach.

Progressive neural networks (PNN) (Rusu et al., 2016). PNN is an architecture-based approach. It trains a new neural network for each new task, using connections between old tasks' networks and the current task's network to incorporate forward transfer. However, the overall model grows linearly in size with task, and PNN is therefore not efficient in terms of model capacity. It also does not have potential for backward transfer. It does, however, obtain good results on many benchmarks.

PathNet (Fernando et al., 2017) and PackNet (Mallya and Lazebnik, 2018). PathNet and PackNet can be seen as a significantly more capacity-efficient alternatives to PNN, as they use a single neural network only. They learn a mask over network weights during training, and then fix these weights for training on future tasks. They therefore do not have potential for backward transfer (like PNN), and this hurts how generally-applicable they are to different problems. They differ from each other in their training procedures for learning masks.

Progress & Compress (P&C) (Schwarz et al., 2018). P&C uses concepts from Progressive Neural Networks and EWC, and is a hybrid regularisation and architecture-based approach. They train an active column on each new task (with some layerwise connections to the 'knowledge base'), which is then distilled into a knowledge base. This distillation step uses Online EWC to prevent catastrophic forgetting.

Replay (or experience replay) (Ratcliff, 1990; Robins, 1995). Replay is the simplest rehearsal-based approach. It simply stores a subset of past data in memory, and rehearses that data during training on future tasks. It is often used as a baseline when comparing rehearsal-based approaches. We discuss Replay and variants of Replay in Section 4.1.

Deep Generative Replay (DGR) (Shin et al., 2017). Inspired by the generative nature of the hippocampus in the brain, DGR trains a generative model on previous tasks' data, and

uses it to generate pseudo-datapoints that are replayed when training future tasks. However, using generative models limits the maximum size of datasets this method can scale to. As DGR uses rehearsal of (pseudo-)data, it is a rehearsal-based approach to continual learning.

Gradient Episodic Memory (GEM) (Lopez-Paz and Ranzato, 2017). GEM uses a subset of past memory to constrain optimisation when training on new data. Specifically, they constrain optimisation such that the loss over past tasks (approximated by using a subset of past data) does not worsen during optimisation. GEM performs especially well in the setting where there is a stream of data, with each datapoint only seen once. GEM is a hybrid regularisation and rehearsal-based approach.

2.4 Measuring performance in continual learning

In this section, we introduce the continual learning metrics and benchmarks that we will use to compare different algorithms throughout the thesis. We start by looking at non-continual baselines and metrics. We then introduce the benchmarks we will use.

2.4.1 Non-continual baselines

Throughout this thesis, we will compare to two non-continual baselines: 'Joint Tasks' and 'Separate Tasks'. These baselines provide insight on the difficulty of our benchmarks and how well our continual learning methods are doing.

Joint Tasks baseline. This baseline was previously introduced in Equation 2.2. It assumes we have access to data from all tasks at once (hence breaking the desiderata of continual learning), and requires training a specific method on all data at once, with just a single network. We assume this is the upper-bound on performance on our benchmarks, and our aim in this thesis is to perform close to this baseline while not storing all the past data or being as expensive to train. We can also call this the retraining-from-scratch method.

Separate Tasks baseline. This baseline trains separate networks on each task in the continual learning benchmark. There is therefore no potential for forward or backward transfer of information between tasks, as we train on each task's data separately. We will see that, on some continual learning benchmarks, Joint Tasks and Separate Tasks perform similarly, indicating that there is no real potential for transfer between tasks. On other continual learning benchmarks, Joint Tasks outperforms Separate Tasks. In such cases, we also hope

that our continual learning algorithms will outperform the Separate Tasks baseline, by using forward and/or backward transfer.[4]

Other offline baselines. There are other possible non-continual learning baselines that can be useful to compare against. We will not consider other baselines in this thesis as the two we have introduced are the most important. However, we could also consider more fine-grained baselines, such as one that trains a separate network for each task t on all data up to and including the latest task (each network is trained on $\mathcal{D}_{1:t}$). This baseline would tell us the maximum possible transfer possible for each task when compared with the Separate Tasks baseline.

2.4.2 Metrics

We will use three metrics to analyse methods' performance on benchmarks: average accuracy, forward transfer, and backward transfer. Average accuracy measures overall performance, and forward/backward transfer provide insight into how the method achieves its performance.

Average accuracy (ACC). The most important metric that we consider in this thesis is the average accuracy across all tasks in the continual learning benchmark, calculated after training on the last task. It is defined in Equation 2.27, and higher is better.

Backward Transfer (BWT). We use the BWT metric defined in Lopez-Paz and Ranzato (2017), which captures the difference in accuracy obtained when a task is first trained and its accuracy after the final task. Higher is better and quantifies performance gain from backward transfer versus performance loss due to forgetting or interference. BWT is defined in Equation 2.28. When methods catastrophically forget information, we expect a large negative BWT.

Forward transfer (FWT). We define a forward transfer metric as the average improvement in accuracy on a new task over a separate model trained *only* on that task. This measures how well the method uses previously seen knowledge to improve classification accuracy on newly seen tasks. FWT is defined in Equation 2.29, and a higher value is better. Note that different algorithms can get different accuracies when separately trained on a single task, and this can complicate comparing FWT between different algorithms.

[4] It is possible that Joint Tasks has lower performance if model capacity is not large enough, as it is trained on significantly more data than a single Separate Tasks model. However, this is not the case for the benchmarks we consider in this thesis.

Other works have used different definitions of forward transfer, for example Lopez-Paz and Ranzato (2017) defined forward transfer in terms of zero-shot performance. It might also be interesting to define forward transfer as the ability of a method to leverage previous information to train quicker on a new task, measuring the speed of convergence on a new task when compared with a model trained only on that task. However, this depends on hyperparameters (such as the learning rate) and can be hard to define. We therefore do not further consider this alternate definition.

Mathematical definitions of metrics. Let $R_{i,j}$ be the classification accuracy of the model on task j after training on task i. Let R_j^{sep} be the classification accuracy of a separate model trained only on task j. Then, our metrics are defined as,

$$\text{Average accuracy, ACC} = \frac{1}{T}\sum_{j=1}^{T} R_{T,j} \tag{2.27}$$

$$\text{Backward Transfer, BWT} = \frac{1}{T-1}\sum_{j=1}^{T-1} R_{T,j} - R_{j,j}, \tag{2.28}$$

$$\text{Forward Transfer, FWT} = \frac{1}{T-1}\sum_{j=2}^{T} R_{j,j} - R_j^{\text{sep}}. \tag{2.29}$$

2.4.3 Benchmarks

This section discusses the benchmarks we use to test our continual learning algorithms throughout this thesis. There have been many benchmarks proposed in the community, especially recently. We focus on image classification benchmarks with MNIST (LeCun and Cortes, 2010) and CIFAR (Krizhevsky and Hinton, 2009). Some recent works have scaled to larger datasets such as OmniGlot (Lake et al., 2015; Schwarz et al., 2018), TinyImageNet (Stanford, 2021; Lange et al., 2021), CUB (Wah et al., 2011; Chaudhry et al., 2019), Core50 (Lomonaco and Maltoni, 2017) and ImageNet (Deng et al., 2009). Other works focus on continual learning in Reinforcement Learning (RL), sometimes split in separate tasks (such as with Atari games (Bellemare et al., 2015)), or more generally as continual RL (see Khetarpal et al. (2020) for a review). We do not consider RL in this thesis, although it would be interesting to apply our continual learning methods in model-based RL for continual learning.

We now describe our four benchmarks. We use a toy benchmark called Toy-Gaussians, and three common image classification continual learning benchmarks: Split MNIST, Permuted MNIST and Split CIFAR.

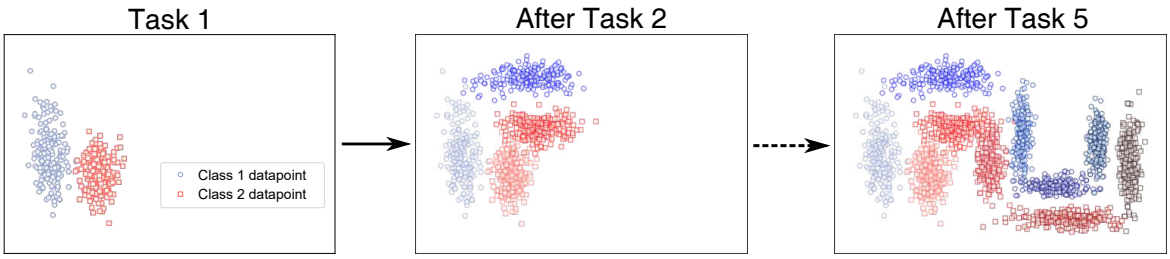


Figure 2.3: Each task in Toy-Gaussians introduces data in a new region of input space. We are always performing binary classification between the blue circles and red squares. The left plot shows the data in Task 1, where we need to learn a simple classification boundary. The middle plot shows data from Task 1 and also new data from Task 2; the new datapoints are in a different region of input space. The right plot shows data from all 5 tasks. At every new task, the decision boundary needs to change and curve around in order to follow the new data.

Toy-Gaussians. This is a toy 2D binary classification benchmark, useful for visualising the performance of continual learning methods. There are 5 tasks, and always two classes. Each task consists of samples from a pair of Gaussians (with 2000 datapoints per class), and the model has to solve the binary classification problem. Each new task adds data in a different region of input-space, and the difficulty is in remembering previous tasks while learning the new task. As we see more tasks, the decision boundary should curve around in order to follow new data. Figure 2.3 visualises the tasks. We will usually use train accuracy (after all 5 tasks) as a way to quantitatively measure performance on this benchmark.

Split MNIST. In this benchmark, we have to sequentially solve five binary classification tasks from the MNIST dataset (LeCun and Cortes, 2010): {0v1}, {2v3}, {4v5}, {6v7}, {8v9}. The challenge in Split MNIST is to obtain good performance on new tasks while retaining performance on old ones. As previously discussed, we assume a multi-head (or task-incremental (van de Ven and Tolias, 2019)) setting, where the algorithm is told which task an image belongs to both when training and testing. We usually use a multi-layer perceptron with two hidden layers, each with 256 hidden units, and with ReLU activation functions, as in previous work (Zenke et al., 2017; Nguyen et al., 2018). In Chapter 3 we will also consider a model with one hidden layer with 200 hidden units. This benchmark does not have much potential for forward or backward transfer, with both Joint Tasks and Separate Tasks achieving extremely high performance (99.7% accuracy using the Adam optimiser (Kingma and Ba, 2015)).

Permuted MNIST. This benchmark consists of ten tasks received sequentially, each of which is the standard (10-way) MNIST classification task, with the pixels having undergone a fixed permutation randomly selected for each task (Goodfellow et al., 2014; Kirkpatrick et al., 2017). Ideally, a network with two or more hidden layers would use lower layer(s) to 'de-permute' the images and higher layer(s) to solve MNIST, which is then constant between tasks. We always use a two-hidden-layer model with 100 units in each layer and ReLU activation functions, and use a single-head setup (also known as the domain-incremental setting (van de Ven and Tolias, 2019)). Past works have used different sizes of networks for this task, and a comprehensive summary is provided in Table 2 of Swaroop et al. (2019). This benchmark does not have potential for forward or backward transfer, as all tasks only consist of permutations of previous task's images. Separate Tasks and Joint Tasks perform similarly well, achieving 98% using the Adam optimiser.

Split CIFAR. This benchmark consists of six tasks. The first task is the full CIFAR-10 dataset (Krizhevsky and Hinton, 2009), which has $50,000$ training datapoints and 10 classes. The following 5 tasks consist of 10 classes each from CIFAR-100 (Krizhevsky and Hinton, 2009), corresponding to $5,000$ training datapoints per task. We always use the same CifarNet model architecture as Zenke et al. (2017): a multi-head CNN with 4 convolutional layers, then 2 dense layers with dropout. We assume a multi-head (or task-incremental) setting, as in Split MNIST. This benchmark is of larger size than the other benchmarks, and is the only one for which we use convolutional neural networks.

In Figure 2.4 we show the performance of our two non-continual learning baselines, Joint Tasks and Separate Tasks, on Split CIFAR, along with performance of some continual learning methods (EWC (Kirkpatrick et al., 2017) and SI (Zenke et al., 2017)). We use the Adam optimiser for Separate Tasks and Joint Tasks performance (other optimisers might get slightly different results). The final column shows the final average accuracy (ACC), while other columns show performance on each task after training on the final task. The Joint Tasks baseline outperforms Separate Tasks, indicating that there is potential for significant forward and/or backward transfer on this benchmark. EWC does not perform even as well as Separate Tasks, while SI performs within error (although note that continual learning methods, like EWC and SI, only use a single network, while Separate Tasks uses a separate network per task, and therefore has higher memory cost). We want our methods to beat the Separate Tasks baseline and approach Joint Tasks. We will return to this figure throughout this thesis in order to compare new methods' performance.

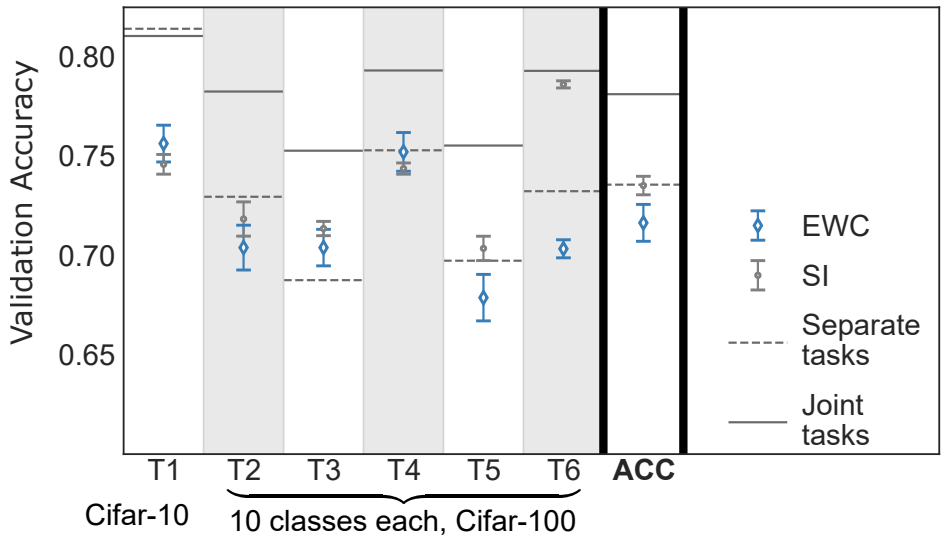


Figure 2.4: This figure plots performance of the Separate Tasks and Joint Tasks baselines on Split CIFAR, along with some continual learning baselines introduced in Section 2.3: EWC (Kirkpatrick et al., 2017) and SI (Zenke et al., 2017). 'Tx' refers to performance on Task x after training on the final task. The ACC column plots average accuracy. We see that the Joint Tasks baseline outperforms Separate Tasks, indicating that there is potential for forward and/or backward transfer on this benchmark. EWC does not perform even as well as Separate Tasks, while SI performs within error (we plot mean performance and standard deviation over 5 runs). We want our methods to beat the Separate Tasks performance and approach Joint Tasks. We will return to this figure throughout this thesis in order to compare new methods' performance.

Chapter 3

Weight-space variational continual learning

Our focus in this chapter is on weight-space techniques for variational continual learning. As discussed in Chapter 2, such weight-prior regularisation-based approaches to continual learning have shown success. Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017) uses (an approximation to) probabilistic continual learning for neural networks, and further work improved such Laplace weight-priors (Schwarz et al., 2018; Ritter et al., 2018). In this chapter we will use Variational Inference (VI) instead of Laplace approximations. VI is is a more global approximation as it averages over distributions instead of using local solutions, and is hence expected to be more robust (Opper and Archambeau, 2009; Khan and Rue, 2021). We therefore hope that our work will further improve on previous methods.

Nguyen et al. (2018) showed how a fully variational approach to continual learning can lead to improved results. We summarised their algorithm, Variational Continual Learning (VCL), in Section 2.2.1, and we start this chapter by improving its performance. We find that performance does not monotonically increase as we train for longer. We improve the algorithm's extremely slow convergence rate, and increase VCL's performance by training for sufficiently long. When we explore VCL's solutions, we find that optimising for longer led to pruning out entire units in the neural network. We summarise this pruning effect in Section 3.1.2, and use these insights to understand how this can help in continual learning.

However, despite these improvements to VCL, obtaining reasonable performance on larger neural networks remains difficult. We therefore consider a natural-gradient VI algorithm, Variational Online Gauss-Newton (VOGN) (Khan et al., 2018). Natural-gradient update steps promise to converge quicker than standard-gradient update steps (for intuition about this see Section 2.2.2). In Section 3.2, we scale VOGN up to large datasets and architectures in the full-batch setting, such as ImageNet/ResNet-scale for the first time. We see that

VOGN achieves similar accuracy to Adam/SGD while keeping some benefits of Bayesian principles such as better uncertainty calibration and out-of-distribution performance. We apply VOGN to continual learning in Section 3.2.3, observing an ability to scale to larger settings than previously possible with VCL.

However, despite these improvements to *weight-space* variational continual learning, we still find fundamental problems remain. We return to a toy benchmark in Section 3.3 to visualise these problems. This motivates us to move to *function-space* continual learning for the rest of this thesis.

3.1 Variational Continual Learning (VCL)

As introduced in Section 2.2, we can sequentially optimise the variational objective function, using the old posterior as our new prior whenever we see new data (Equation 2.5). When we combine this with Variational Inference (VI), and use Bayes-By-Backprop (Blundell et al., 2015) to optimise for a mean-field Gaussian approximating family, we get Variational Continual Learning (VCL). In VCL, we minimise the following variational objective function (this is repeated from Equation 2.9),

$$\mathcal{L}_t^{\text{VCL}}(\boldsymbol{\eta}_t) = \sum_{i \in \mathcal{D}_t} \frac{1}{S} \sum_{s=1}^{S} \left[-\log p(\boldsymbol{y}_i | \boldsymbol{w}^{(s)}, \boldsymbol{x}_i) \right] + \underbrace{\mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})} \left[\log \frac{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}{q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})} \right]}_{\text{Analytically calculated}}. \tag{3.1}$$

We now briefly recap the definitions of these terms, although the reader can also look at Chapter 2 for a full description. We are optimising for the parameters $\boldsymbol{\eta}_t$ of the mean-field Gaussian approximate posterior $q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, where $\boldsymbol{w}$ are the weights in our model (a neural network). Our prior over weights is the previous approximate posterior $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$. The first term (the likelihood term) is the negative log-likelihood of the data given the model $-\log p(\boldsymbol{y}_i | \boldsymbol{w}, \boldsymbol{x}_i)$, where our dataset $\mathcal{D}_t$ consists of inputs $\boldsymbol{x}_i$ and labels $\boldsymbol{y}_i$. We approximate the likelihood term using Monte-Carlo sampling, drawing S samples from our approximate posterior $\boldsymbol{w}^{(s)} \sim q_{\boldsymbol{\eta}_t}(\boldsymbol{w})$. The second term (the KL-to-prior) term can be analytically calculated as all distributions are Gaussian distributions. VCL separately optimises Equation 3.1 for the parameters $\{\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t\}$ using Adam and automatic differentiation.

Nguyen et al. (2018) applied VCL to common benchmarks in continual learning, such as Split MNIST and Permuted MNIST, and showed strong results (for the time), outperforming baselines such as EWC (Kirkpatrick et al., 2017) and SI (Zenke et al., 2017). They ran VCL with and without coresets (see Section 2.2.1). Nguyen et al. (2018) also applied VCL to deep generative models, however we do not consider generative models in this thesis.

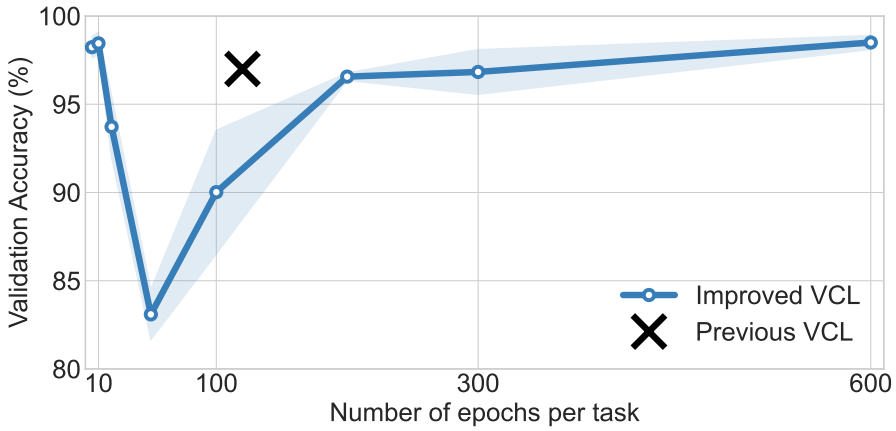


Figure 3.1: VCL's average performance (after training on all tasks) on Split MNIST as we increase the number of epochs per task does not monotonically increase. Previous VCL did not use our improvements to convergence rate, and so required 120 epochs to reach an early-stopped solution. Our improvements reach this performance after just 10 epochs, but we can improve performance further by training for 600 epochs.

3.1.1 Improving VCL

Our first contribution is to improve VCL's performance by being more careful about the optimisation process, finding that we also gain some understanding as to why it works so well. Specifically, we improve results by training until (close to) convergence of an optimum of Equation 3.1. We do this by training for much longer (more optimisation steps), and also improving the algorithm's convergence rate by using the local reparameterisation trick (Kingma et al., 2015) and improving initialisation.

Previously, Nguyen et al. (2018) used early-stopping to achieve good results with VCL. We plot performance (final average accuracy after training on all tasks) as a function of epochs on the Split MNIST benchmark in Figure 3.1, where we see that VCL performs well at very few epochs, but performs better when trained for significantly longer. The Previous VCL algorithm from Nguyen et al. (2018) did not use our improvements to convergence rate. It therefore takes more epochs to reach the high performance of early-stopped solutions (120 epochs instead of 10). By using our improvements and training for significantly longer, we see that after 600 epochs, our Improved VCL algorithm performs the best. Crucially, although progress can sometimes appear to stall during optimisation of the objective function, in reality progress is just extremely slow.

We use two techniques to improve the convergence rate:

1. We employ the local reparameterisation trick (Kingma et al., 2015) during Monte-Carlo sampling of the likelihood term. Instead of sampling each (Gaussian) weight

independently, we sample the pre-activation latent variables just before each neuron's non-linearity (this latent variable is a linear combination of the neuron's input weights). This leads to two improvements, (i) it reduces the variance of stochastic gradients during sampling, and (ii) it is marginally quicker as we sample fewer random variables. The first improvement is particularly important, as it speeds up convergence drastically, reducing the number of epochs until convergence.

2. We also improve the initialisation of the weights of our neural network. VCL (Nguyen et al., 2018) initialised the mean-field Gaussian distribution over weights by setting the means at the maximum-likelihood solution of a deterministic neural network, and setting the variances to be small. We find that initialising the means to be small and random improves convergence speed and leads to more consistently well-performing results across random seeds. Intuitively, this is because initialising randomly allows the network to quickly learn the best trade-off between the new task's data (the likelihood term) and information from previous tasks (the KL-to-prior term).

We summarise performance improvements on continual learning benchmarks in Table 3.1. In Section 3.1.2 we will see that training for such a long time leads to pruning in our variational BNN, but we first focus on the quantitative improved results. Details on the continual learning metrics and benchmarks are in Section 2.4, and code is available at `https://github.com/nvcuong/variational-continual-learning`.

For Split MNIST, we run a one-hidden-layer model with 200 units for 600 epochs (with 256 batch size), sharing the lower level weights between tasks, and report the mean performance and standard deviation over 10 runs. Note that this is different to the two-hidden-layer model described in Section 2.4, but we do this to visualise pruning later (Section 3.1.2). Without coresets, we achieve a final validation accuracy of $98.5 \pm 0.4\%$. 'Previous VCL' (Nguyen et al., 2018) ran for 120 epochs and reported 97.0% accuracy. With coresets, we achieve $98.2 \pm 0.4\%$, similar to the 98.4% reported previously. We find that our Improved VCL has similar forward transfer (FWT) as Previous VCL, but has significantly better backward transfer (BWT), indicating it is forgetting less over many tasks.

For Permuted MNIST, we run a two-hidden-layer model with 100 units in each hidden layer (as described in Section 2.4) for 800 epochs (with 1024 batch size), and report the mean performance and standard deviation over 5 runs. Without coresets, this achieves a final average validation accuracy of $93 \pm 1\%$. 'Previous VCL' (Nguyen et al., 2018) ran for 100 epochs and reported 90% accuracy. With coresets, we achieve a final average validation accuracy of $94.6 \pm 0.3\%$, an improvement from the previously reported 93%. We see that Improved VCL has better FWT and BWT than Previous VCL both with and without coresets.

Benchmark	Metric	Improved VCL	Previous VCL	EWC	SI
Split MNIST	ACC (%)	**98.5±0.4**	97.0	63.1	**98.9**
	FWT (%)	-1.3±0.5	-0.9	–	–
	BWT (%)	-0.1±0.2	-2.2	–	–
Split MNIST +40 coreset/task	ACC (%)	98.2±0.4	98.4	–	–
	FWT (%)	-1.5±0.6	-1.0	–	–
	BWT (%)	-0.3±0.3	-0.7	–	–
Permuted MNIST	ACC (%)	**93±1**	90	84	86
	FWT (%)	-0.2±0.1	-2	–	–
	BWT (%)	-4±1	-6	–	–
Permuted MNIST +200 coreset/task	ACC (%)	**94.6±0.3**	93	–	–
	FWT (%)	-0.2±0.1	-2	–	–
	BWT (%)	-2.3±0.3	-4	–	–
Split CIFAR	ACC (%)	48.8±2.2	–	71.6±0.9	**73.5±0.5**
	FWT (%)	0.8±2.0	–	0.17±0.9	–
	BWT (%)	-29±4	–	-2.3±1.4	–
Split CIFAR +200 coreset/task	ACC (%)	67.4±1.4	–	–	–
	FWT (%)	1.8±3.1	–	–	–
	BWT (%)	-9.2±1.8	–	–	–

Table 3.1: Final average validation accuracy (ACC), forward transfer (FWT) and backward transfer (BWT) metrics on Split MNIST, Permuted MNIST and Split CIFAR, with and without random coresets. We report mean performance and standard deviation over 5 runs (10 runs for Split MNIST). Our improvements lead to significant increases in performance on the MNIST benchmarks (except for remaining within error on Split MNIST + Coreset). On the larger Split CIFAR benchmark, even our improvements to VCL do not help enough: performance is still poor compared to other methods. Note that Previous VCL fails to get any reasonable accuracies in a reasonable amount of time on Split CIFAR. Baselines are Previous VCL (Nguyen et al., 2018), Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017), and Synaptic Intelligence (SI) (Zenke et al., 2017). Results for EWC and SI for Split MNIST and Permuted MNIST are taken from Nguyen et al. (2018). Results for SI for Split CIFAR is taken from Zenke et al. (2017). All coresets are chosen randomly. We do not have results for EWC and SI with coresets, and also do not have FWT/BWT transfer metrics when results are taken from other papers.

As we now run VCL for considerably longer than in Nguyen et al. (2018), Improved VCL is now very slow to run, even on the MNIST benchmarks. This is particularly true when we compare to optimisers such as SGD or Adam, which are also considerably quicker per optimisation step. This means that on the larger Split CIFAR benchmark, we are unable to run VCL for long enough to reach a converged solution, and VCL performs very poorly. To obtain the results shown for Split CIFAR in Table 3.1, we run for 500 epochs on task 1, and 5000 epochs on tasks 2-6 (note that task 1 has 10 times more data than the other tasks), getting a very poor $48.8 \pm 2.2\%$ final average accuracy. We see that VCL has very large negative BWT, indicating it is catastrophically forgetting.

We now turn our attention to why running VCL for longer led to improved results. We will see that training for longer leads to increased pruning in the neural network, and that this is important for performance increases in continual learning. Interestingly, this behaviour is in contrast to what we might expect with the true Bayesian solution, which would remain uncertain over all weights when we see little data, using all model capacity instead of pruning out parts of the model.

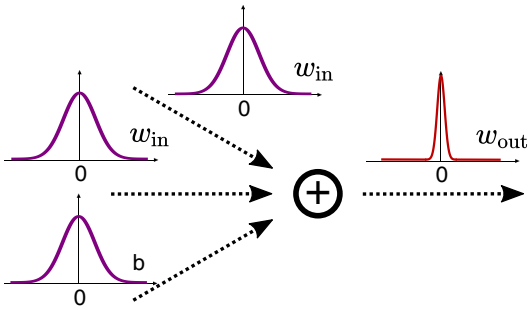
3.1.2 Pruning

We previously found we can significantly increase VCL's performance by running VCL for longer, and by speeding up convergence using some techniques (such as clever initialisation and gradient-variance reduction tricks). In this section, we look at the weights of the learned neural network to determine why training for longer was important, and we find that pruning of entire units in the neural network plays a major role. We start by describing what pruning is in the full-batch setting, and why it happens in variational BNNs. We then look at the weights of learned neural networks on Split MNIST and Permuted MNIST, discussing how pruning is important for continual learning.

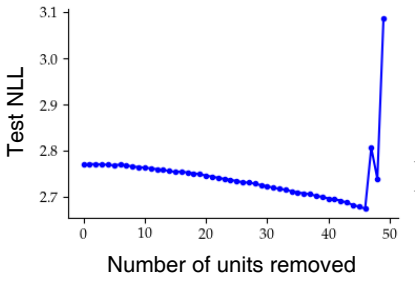
Overpruning in variational Bayesian NNs

Training variational Bayesian neural networks (BNNs) with Gaussian approximating families has been found to prune out entire units (Trippe and Turner, 2018; Swaroop et al., 2019; Tomczak et al., 2020, 2021). In the full-batch setting, the network prunes a very large number of units, causing *underfitting* of the data (Turner and Sahani, 2011), and this effect has been termed *overpruning* in variational BNNs.

This pruning effect seems to be due to the choice of inference scheme. Intuitively, we can explain it by looking at our objective function (Equation 3.1). By reducing the effect of a unit on the output prediction (achieved by setting the output weights to have zero mean and



(a) Pruned units have input weights at the prior, and output weights with zero-mean small-variance.



(b) Removing units can initially improve performance due to noise leaking through pruned units (lower is better).

Figure 3.2: (a) Pruned units set their input weights to the prior (zero-mean Gaussians with fixed variance), and their output weights to a zero-mean small-variance distribution to minimise effect on output distribution. If the network has ReLU activation functions, a unit can alternatively be pruned by setting its bias weight to be negative (Tomczak et al., 2021). Regardless of pruning mechanism, pruned weights can still contribute noise to predictions by leaking through the unit. (b) Because of this leaked noise, removing pruned weights can improve performance, although only slightly, and often this performance improvement is only visible if plotting negative log-likelihoods (instead of accuracies). This experiment is on 200 random Boston regression datapoints with a one-hidden layer network with 50 hidden units, ReLU activations, and zero-mean Gaussian prior. Figure from Tomczak et al. (2021).

small variance), the input weights to the unit can be set to their prior. The small variance of the output weights increases the KL-to-prior term, but this is offset by the reduction in the KL-to-prior term from the more numerous input weights. Provided the likelihood term does not change too much, a pruned solution is therefore more optimal.

We draw a cartoon representation of a pruned unit in Figure 3.2(a). In Appendix A.1 we see how units are entirely pruned out by plotting input and output weights for each unit in a single-hidden layer model trained to classify the digits {0v1} in MNIST. Even though there are 200 units in the hidden layer, only one unit is being used for this binary classification task, and the remaining units are pruned out as part of the optimisation process.

Pruning entire units can sometimes be beneficial, even though underfitting behaviour is undesirable in general. It can be useful in model compression (Louizos et al., 2017), and as we will see later in this section, it can be useful in continual learning. However, in Tomczak et al. (2021) we also find that pruned units can leak noise into predictions (as the output weights are not exactly delta functions). This means that as we remove pruned units from the network, prediction quality can improve. We see this in Figure 3.2(b).

It is possible to avoid pruned solutions by early-stopping training. However, this means we no longer optimise the lower bound to the model evidence completely, which can raise

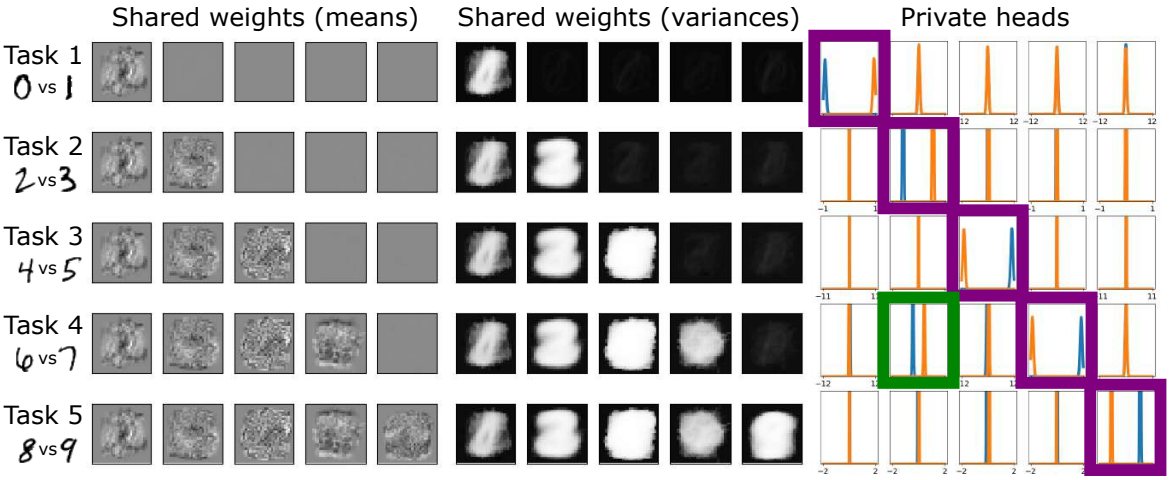


Figure 3.3: Only one unit is active per task in this one-hidden-layer model trained on Split MNIST, even though there are 200 units in total. Each row corresponds to a different task in Split MNIST. Left: means of input weights to each of the five units, Centre: variances of input weights, Right: output weights for each task's two classes. The top row shows that only a single unit is used to classify task 1 ({0v1}). The second row shows that a second unit, different from the first unit, is used to classify task 2 ({2v3}). Only the output head weights for the second task's unit is at not at 0, indicating only this unit is used. On the fourth row, we see that the output head weights are also non-zero for the second active unit (highlighted in green). This indicates that the model is using forward transfer: it is using information learnt earlier (the unit from task 2) to help classify task 4.

theoretical questions regarding the motivation for VI. Early-stopping also does not lead to better results in all scenarios. We see this in continual learning: the original VCL paper (Nguyen et al., 2018) were effectively using early-stopping. They observed good results, but by training for longer, we were able to improve results (Table 3.1). We next look at how variational pruning interacts with continual learning.

Split MNIST

We start with a visualisation of pruning in Split MNIST. We consider the model trained without coresets, although exactly the same effect is also observed for the model trained with coresets. In Appendix A.1 we show the weights into and out of each unit for the first task (binary classification, {0v1}). Although Appendix A.1 only plots weights after training on the first task, we find this trend continues for all five tasks in Split MNIST, with exactly one unit used per task. We collect these five active (un-pruned) units and plot them in Figure 3.3.

Although this pruning effect leads to underfitting and can be detrimental in some settings (Ghosh et al., 2019; Tomczak et al., 2021), we find it can also help in continual learning. Pruning ensures that the algorithm uses very little of its network capacity, while still achieving

high validation accuracies. Remaining, unused units can be used for other tasks that we may see in the future. Ideally, in a probabilistic framework we would achieve such beneficial pruning through the prior, but surprisingly, the inference procedure is causing this here.

Additionally, the pruning effect allows us to see some forward and backward transfer (see Figure 3.3), both important qualities in a good continual learning solution (see continual learning Desiderata 4 and 5 in Section 2.1). Forward transfer is visible when previous tasks' active units have non-zero weights for subsequent tasks. For example, unit 2, which was learnt after task 2 (classifying digits {2v3}), has non-zero output weights after task 4 (classifying {6v7}). The model therefore uses some information about task 2 in solving task 4. This is highlighted in green in the right plot in Figure 3.3. Although less visible in the plots, there is also backward transfer in the same units: unit 2's input weights change slightly after training on task 4 ({6v7}), potentially changing accuracy on task 2 ({2v3}). In this case however, any backward transfer does not result in a change in test accuracies; this could be because there is no potential for improvement given the high accuracies involved.

Permuted MNIST

We now look into pruning when VCL is trained on Permuted MNIST. Figure 3.4 plots the numbers of active (un-pruned) units after training on each task. There are more active units in Permuted MNIST than there were in Split MNIST, likely due to the more difficult nature of Permuted MNIST (classifying between 10 digits, as opposed to between 2). However, only 11 units are used in the second hidden layer, with the remaining 89 units pruned out. Additionally, the output weights on these 11 units do not change between tasks. This indicates that the purpose of the hidden layers remains constant: the hidden layers de-permute the images, allowing the output weights to always simply classify between 10 digits.

Beyond re-using the upper level weights, there is not much evidence of forward or backward transfer. We should expect this from Permuted MNIST because the network trains on all MNIST digits on the first task itself, hence already learning the 'best' way to classify between MNIST digits. Any subsequent permuted images cannot improve this. Instead, the remaining focus of Permuted MNIST seems to be on ensuring we use available model capacity as efficiently as possible. Increasing the model capacity improves results: training a network with 250 units in the lower hidden layer (instead of 100) improves final average accuracy to 95.5% (this was also for 10 tasks).

Incorporating coresets also improves results. However, the number of active units (plotted in Figure 3.4) is about the same. Instead, training VCL with coresets appears to reinforce previous tasks' images. Incorporating coresets can be viewed as changing the order in which the model trains on data, or, changing the schedule with which we visit training data.

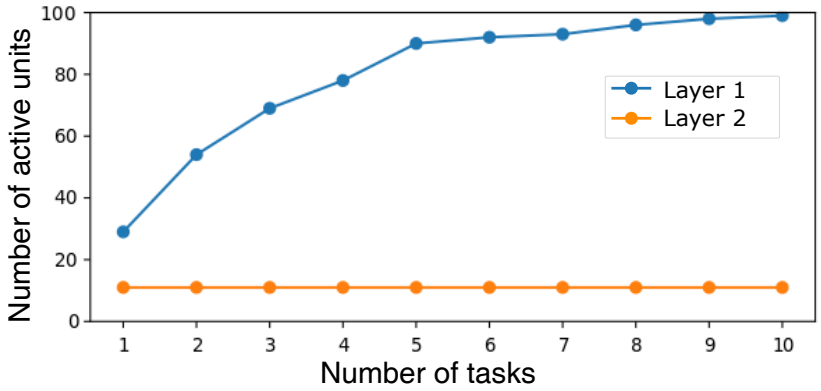


Figure 3.4: The number of active units in each of the layers in a two-hidden-layer MLP trained on Permuted MNIST. The x-axis shows the number of tasks trained so far. After the first task, only 30 units are active in the first hidden layer, and 11 units are active in the second hidden layer (both layers consist of 100 units). The number of units active in the second hidden layer remains constant as we train on more tasks, while the number of active units in the first hidden layer slowly increases.

Discussion: pruning for continual learning

We have seen how VCL's results were improved by training for longer, as we could exploit variational pruning of units in neural networks. This was important on the MNIST benchmarks, allowing the network to use model capacity efficiently, and providing some explainable plots for forward and backward transfer in Split MNIST. However, in order to exploit pruning, we require the algorithm to be run for an extremely long time. Our Improved VCL results required many hundreds of epochs.

In Loo et al. (2021), we further exploited this pruning in continual learning to combine VCL with FiLM layers. The additional task-specific FiLM layers (Perez et al., 2018) modulate each feature between layers, allowing for easier pruning of units. We found that this significantly helps VCL, as the FiLM layer parameters help the network to automatically assign certain units to certain tasks.

To scale VCL (without FiLM layers) to significantly larger datasets, we will need a different approach to significantly speed up optimisation. A promising direction is to use natural-gradient variational inference (NGVI) to optimise Equation 3.1. Natural-gradients have the potential to speed up convergence by using the information geometry of the distribution being optimised. We next describe work to scale NGVI to large neural networks, which we then apply to continual learning. We will see significantly faster convergence.

3.2 Variational Online Gauss-Newton (VOGN)

So far, our variational BNNs were optimised by using Bayes-By-Backprop (Blundell et al., 2015), which optimises the means and standard deviations of our mean-field Gaussian separately. In Section 2.2.2, we looked at why such standard-gradient methods may not be ideal. We used this to motivate natural-gradient VI (NGVI), which incorporates the information geometry of the distribution being optimised, therefore potentially dramatically speeding up convergence. This will be important as we scale to larger architectures and datasets, and especially important in continual learning, where we found that training for longer was important to improve results.

Previous work (Khan et al., 2018) derived natural-gradient VI algorithms that can run on small neural networks. In Section 2.2.2 we derived one of these algorithms, Variational Online Gauss-Newton (VOGN), explaining the assumptions and approximations required to reach the final VOGN algorithm in Equations 2.25 and 2.26.

Although there are different NGVI algorithms that we can choose from, each with their own approximations, we choose to focus on VOGN. This is because other algorithms, such as vAdam (Khan et al., 2018), make more restrictive assumptions and therefore might lose more of the benefits arising from using Bayesian principles. VOGN makes fewer approximations, and is therefore slower than vAdam, but as we will see, it can still be scaled up due to its similarities with Adam.

In Section 3.2.1, we compare the VOGN and Adam update equations, and use their similarities to motivate borrowing tricks that the community has developed for Adam over many years. As a result, in Section 3.2.2 we see that VOGN obtains similar performance in about the same number of epochs as Adam when training on many popular deep networks (such as LeNet, AlexNet, and ResNets) on datasets such as CIFAR-10 and ImageNet. We also see that, despite using an approximate posterior, VOGN preserves many benefits coming from Bayesian principles. Compared to standard deep-learning methods, predictive probabilities are well-calibrated, and uncertainties on out-of-distribution inputs are improved.

We then turn our attention to continual learning in Section 3.2.3. We see that our efforts in scaling VOGN up for batch-learning also results in significantly quicker convergence in continual learning. This allows us to scale VOGN beyond MNIST-based benchmarks, and we see good performance on Split CIFAR.

3.2.1 Practical deep learning with variational inference

In this section, we scale VOGN up to large datasets and architectures. We do this by comparing VOGN's update equations to Adam's update equations, noticing similarities between them. This motivates us to borrow techniques that the community has developed for Adam over many years, such as data augmentation, batch normalisation, and momentum.

We start by repeating the VI objective function, which we are optimising with respect to the natural parameters $\boldsymbol{\eta}_t$ of our (mean-field Gaussian) approximate posterior $q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, where $\boldsymbol{w}$ are the parameters of a neural network,

$$\mathcal{L}(\boldsymbol{\eta}_t) = \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[-\log p(\mathcal{D}_t|\boldsymbol{w})\right] + \tau\, \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[\log \frac{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}{p(\boldsymbol{w})}\right], \tag{3.2}$$

where we now include a tempering parameter τ in front of the KL-to-prior term. We can set $\tau \neq 1$ when we expect model misspecification and/or adversarial examples (Vovk, 1990; Ghosal and Van der Vaart, 2017). Setting $\tau = 1$ recovers standard variational Bayesian inference.

VOGN optimises Equation 3.2 using natural-gradient updates (derivation in Section 2.2.2),

$$\boldsymbol{\mu}_{j+1} \leftarrow \boldsymbol{\mu}_j - \alpha_j \frac{\hat{\mathbf{g}}(\boldsymbol{w}_j) + \tilde{\delta}\boldsymbol{\mu}_j}{\boldsymbol{s}_{j+1} + \tilde{\delta}}, \tag{3.3}$$

$$\boldsymbol{s}_{j+1} \leftarrow (1 - \tau\beta_j)\boldsymbol{s}_j + \beta_j \frac{1}{B}\sum_{i\in\mathcal{B}_j}\left(\mathbf{g}_i(\boldsymbol{w}_j)^2\right), \tag{3.4}$$

which is the same as Equations 2.25 and 2.26, except with the tempering parameter τ included.

We now quickly recap the definitions of each of these terms, although the reader can also look at Chapter 2 for a full description. We are iteratively updating two vectors, $\boldsymbol{\mu}_j$ and $\boldsymbol{s}_j$, where j indexes the iteration. We have a zero-mean prior $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \mathbf{0}, \delta^{-1}\mathbf{I})$, and $\tilde{\delta} = \tau\delta/N$. Our dataset consists of N data examples, and we are taking per-example gradients of the negative log-likelihood $\mathbf{g}_i(\boldsymbol{w}_j)$ at a sample from our current approximate posterior, $\boldsymbol{w}_j \sim q_j(\boldsymbol{w})$. For a randomly-sampled minibatch $\mathcal{B}_j$ of size B, we have defined the average gradient $\hat{\mathbf{g}}(\boldsymbol{w}_j) = \frac{1}{B}\sum_{i\in\mathcal{B}_j}\mathbf{g}_i(\boldsymbol{w}_j)$. There is a simple relation between $\boldsymbol{\Sigma}_j$ and $\boldsymbol{s}_j$, $\boldsymbol{\Sigma}_j^{-1} = N\boldsymbol{s}_j + \delta\mathbf{I}$. Finally, $\alpha_j > 0$ and $0 < \beta_j < 1$ are learning rates, and all operations are element-wise.

Comparison with Adam equations

A key benefit of using VOGN to update Equation 3.2, instead of other methods such as Bayes-By-Backprop, is the similarity of VOGN update equations to Adam update equations. To see this, we write down the form that commonly-used optimisers take, such as SGD, RMSProp (Tieleman and Hinton, 2012), and Adam (Kingma and Ba, 2015),

$$\boldsymbol{\mu}_{j+1} \leftarrow \boldsymbol{\mu}_j - \alpha_j \frac{\hat{\mathbf{g}}(\boldsymbol{\mu}_j) + \delta \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{s}_{j+1}} + \epsilon}, \tag{3.5}$$

$$\boldsymbol{s}_{j+1} \leftarrow (1-\beta_j)\boldsymbol{s}_j + \beta_j \left(\frac{1}{B} \sum_{i \in \mathcal{B}_j} \mathbf{g}_i(\boldsymbol{\mu}_j) + \delta \boldsymbol{\mu}_j \right)^2, \tag{3.6}$$

where $\delta > 0$ is our weight-decay regulariser, $\epsilon > 0$ is a small scalar constant, and all operations are element-wise. Alternative versions with weight-decay and momentum differ from these equations (Loshchilov and Hutter, 2019), but we present a form here that is useful to establish a connection to VOGN.

We now summarise the key similarities and differences between VOGN's equations (Equations 3.3 and 3.4) and Adam's equations (Equations 3.5 and 3.6):

1. *Similarity*: Both $\boldsymbol{\mu}_j$ updates take the form $\boldsymbol{\mu}_{j+1} \leftarrow \boldsymbol{\mu}_j - \alpha_j(\hat{\mathbf{g}} + \delta\boldsymbol{\mu}_j)/\text{function}(\boldsymbol{s}_{j+1})$. The vector $\boldsymbol{s}_{j+1}$ adapts the learning rate in both cases.

2. *Difference*: The denominator in the update for means $\boldsymbol{\mu}_j$ is slightly different. VOGN uses $(\boldsymbol{s}_{j+1} + \tilde{\delta})$, while Adam uses $\sqrt{\boldsymbol{s}_{j+1}}$.

3. *Difference*: VOGN calculates gradients at a sample $\boldsymbol{w}_j \sim q_j(\boldsymbol{w})$, while Adam calculates gradients just at the mean $\boldsymbol{\mu}_j$.

4. *Similarity*: Both updates for $\boldsymbol{s}_{j+1}$ take the form of a moving average update.

5. *Difference*: VOGN uses a Gauss-Newton approximation, requiring $\frac{1}{B}\sum_i (\mathbf{g}_i)^2$, while Adam uses a gradient-magnitude, $\left(\frac{1}{B}\sum_i \mathbf{g}_i + \delta\boldsymbol{\mu}_j\right)^2$. Note that in VOGN, the sum is *outside* the square, while in Adam, the sum is *inside* the square.

A major difference is Difference 5 above: VOGN uses a Gauss-Newton approximation (the empirical Fisher matrix). This is fundamentally different to the gradient-magnitude approach in Adam. VOGN is better approximating second-order information (approximating the Hessian) (Schraudolph, 2002; Graves, 2011; Martens, 2020). The Gauss-Newton matrix also has some nice properties such as being positive semi-definite, which we require. We expect it to become a better approximation to the Hessian as we train for longer.

The gradient-magnitude approach, however, has no such nice guarantees when viewed as an approximation to second-order information: it was heuristically developed in RMSProp and Adam as it was found to perform well. In fact, the gradient-magnitude approach introduces a bias as an approximator to the Gauss-Newton matrix, and this bias increases with minibatch size (see Theorem 1 in Khan et al. (2018)). Therefore, unlike the gradient-magnitude approaches, VOGN is a better second-order method similar to Newton's method, and therefore does not require a square root over $\boldsymbol{s}_{j+1}$ (Difference 2).[1]

However, calculating the Gauss-Newton matrix requires additional computation in modern deep-learning frameworks like PyTorch (Paszke et al., 2019), which makes VOGN slightly slower than Adam. We trade-off this computation cost in order to obtain better variance estimates.

There are many similarities between VOGN and Adam too. These similarities indicate that we might be able to take techniques developed by the community for Adam, and apply similar ideas to scale VOGN up while maintaining good performance. We now describe these techniques in detail. Pseudo-code for the final VOGN algorithm is shown in Algorithm 1.

Techniques to scale VOGN up

We now describe techniques to scale VOGN up to large architectures and datasets.

1. **Batch normalisation:** Batch normalisation (Ioffe and Szegedy, 2015) has been found to significantly speed up and stabilise training of neural networks, and is widely used in deep learning. BatchNorm layers are inserted between neural network layers. They help stabilise each layer's input distribution by normalising using the running average of the inputs' mean and variance. In our VOGN implementation, we simply use existing implementations of BatchNorm with default hyperparameter settings. We do not apply L_2-regularisation or weight decay to BatchNorm parameters (following Goyal et al. (2017)), or maintain uncertainty over BatchNorm parameters. This straightforward application of batch normalisation works for VOGN. We see how batch normalisation improves performance in Figure 3.5 (results are for ResNet-18 on CIFAR-10).

2. **Momentum:** It is well-known that momentum can speed up convergence significantly (Sutskever et al., 2013). Since VOGN is similar to Adam, we can implement momentum in a similar way. We use a momentum rate β_1, shown in Step 17 in Algorithm 1. In Figure 3.5, we see how both momentum and batch normalisation work together to improve VOGN's performance.

[1] We note that other work has attempted to explain the square root using ideas from Bayesian filtering methods (Aitchison, 2018), which is interesting, but a different approach as it uses the temporal dynamics of all other parameters during optimisation.

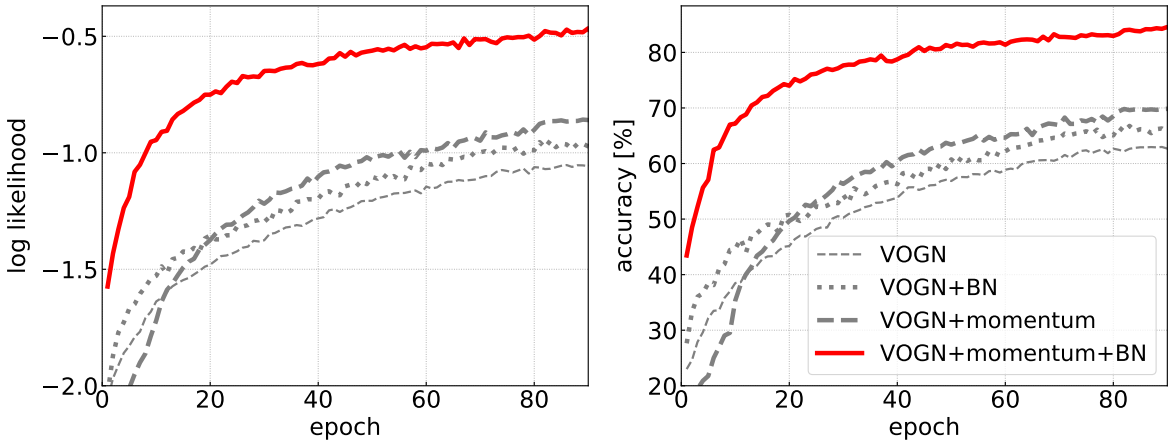


Figure 3.5: Both momentum and BatchNorm (BN) are important to improve VOGN's convergence rate and performance, here shown for ResNet-18 on CIFAR-10. Having only one, or neither, of momentum and BatchNorm results in significantly worse performance. Left figure plots validation log-likelihood, and right figure shows validation accuracy.

3. **Initialisation:** We want VOGN to closely follow Adam's training curves at the beginning of training. We therefore initialise VOGN to be close to Adam for the first few optimisation steps. This significantly increases VOGN's convergence rate.

   We initialise the means $\boldsymbol{\mu}$ in the same way as in Adam (`init.xavier_normal` in PyTorch (Glorot and Bengio, 2010)), and also initialise the momentum term $\mathbf{m}$ to be $\mathbf{0}$, as it is in Adam. VOGN requires an additional initialisation for the variance $\boldsymbol{\Sigma}$: we run a forward pass through the first minibatch, calculate the average of the squared gradients, and initialise the scale $\boldsymbol{s}_0$ with it. This implies that the variance is initialised to $\boldsymbol{\Sigma}_0 = \tau/(N(\boldsymbol{s}_0 + \tilde{\delta}))$. For the tempering parameter τ, we use a schedule where it is increased from a small value (such as 0.1) to 1 in the first few optimisation steps. Note that as $\tau \to 0$, VOGN gets more similar to deterministic algorithms like Adam, as the algorithm is biased to learn smaller variances $\boldsymbol{\Sigma}$.

   With these initialisation protocols, VOGN is able to mimic the convergence behaviour of Adam in the beginning, and quickly converge to a good solution.

4. **Learning rate scheduling:** A common approach to quickly achieve high validation accuracies is to use a specific learning rate schedule (Goyal et al., 2017). The learning rate α is regularly decayed by a factor, typically a factor of 10. The frequency and timings of this decay are usually pre-specified. In VOGN, we use the same schedule used for Adam, and this works well.

5. **Data Augmentation:** When training on image datasets, Data Augmentation (DA) techniques can improve performance drastically (Goyal et al., 2017). For VOGN, we consider two common real-time data augmentation techniques: random cropping and horizontal flipping. After randomly selecting a minibatch at each iteration, we use a randomly selected cropped version of all images. Each image in the minibatch also has a 50% chance of being horizontally flipped.

   We find that directly applying DA gives slightly worse performance than expected, and also affects the calibration of the resulting uncertainty. But we note that DA can be viewed as increasing the effective size of our dataset. We therefore modify the dataset size to be ρN where $\rho \geq 1$, and this improves performance (see Step 2 in Algorithm 1). The reason for this performance boost might be due to the complex relationship between the regularisation δ and dataset size N. For a regularised loss such as with Adam and SGD, the two are unidentifiable, as we can multiply δ by a constant and reduce N by the same constant without changing the location of minima. However, in a Bayesian setting (like with VOGN), the two quantities are separate, and therefore changing the dataset size might also change the optimal prior variance hyperparameter in a complicated way. This needs further theoretical investigations, but our simple fix of scaling N works well in our experiments.

   This method for handling DA is also closely related to KL-annealing in variational inference, as well as the recently-termed 'cold posterior effect' (Wenzel et al., 2020; Loo et al., 2021; Aitchison, 2021). Wenzel et al. (2020) report that tempering the posterior by a temperature improves results (for MCMC sampling). Here, we are effectively scaling the dataset size, and hence just the likelihood term in Bayes' rule, by a constant.

   We set ρ by considering the specific DA techniques used. When training on CIFAR-10, the random cropping DA step involves first padding the 32x32 images to become of size 40x40, and then taking randomly selected 28x28 cropped images. We consider this as effectively increasing the dataset size by a factor of 5 (4 images for each corner, and one central image). The horizontal flipping DA step doubles the dataset size (one dataset of unflipped images, one for flipped images). Combined, this gives $\rho = 10$. Similar arguments for ImageNet DA techniques give $\rho = 5$. Even though ρ is another hyperparameter to set, we find that its precise value does not matter much. Typically, after setting an estimate for ρ, tuning δ a little seems to work well (see Appendix B.3).

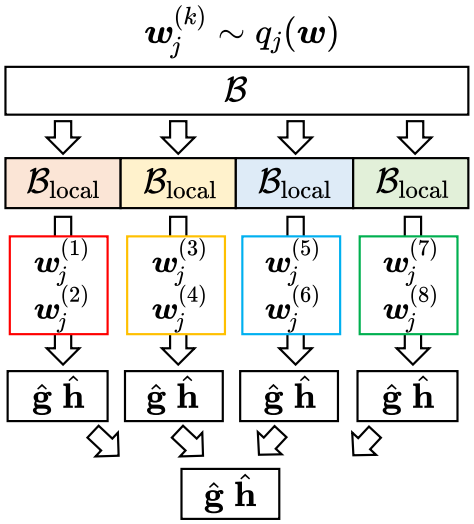


Figure 3.6: Data and Monte-Carlo sample parallelism for VOGN. In this illustration, there are four GPUs. The minibatch $\mathcal{B}$ is split into four, with one $\mathcal{B}_{\text{local}}$ per GPU (data parallelism). Each GPU samples two parameter vectors $\boldsymbol{w}_j^{(k)}$ (Monte-Carlo sample parallelism). Each GPU calculates the average gradient $\hat{\mathbf{g}}$ and Gauss-Newton approximation $\hat{\mathbf{h}}$ on its own data, averaged across its own Monte-Carlo samples. These vectors are then averaged across GPUs at the end.

6. **Distributed training:** We can use distributed training for VOGN to perform large experiments quickly. Typically, we would only parallelise over data, splitting up large minibatch sizes by sending different datapoints to different GPUs. With VOGN, we can also parallelise over Monte-Carlo samples $\boldsymbol{w}_j \sim q_j(\boldsymbol{w})$, where every GPU samples different $\boldsymbol{w}_j^{(k)}$.

   We use a combination of these two parallelism techniques, with different MC samples for different inputs, summarised in Figure 3.6. This reduces variance during training (see Equation 5 in Kingma et al. (2015)). This is particularly important at the beginning of training, which may sometimes require averaging over multiple MC samples to get a sufficiently low variance. Overall, we find that this type of distributed training is essential for fast training on large problems such as ImageNet.

7. **Implementation of the Gauss-Newton update:** As discussed earlier, VOGN uses a Gauss-Newton approximation, which is fundamentally different from the gradient-magnitude approach in Adam. In this approximation, the gradients on individual data examples are first squared and then averaged (see Steps 12 and 18 in Algorithm 1, which implement the update for $\boldsymbol{s}_j$ shown in Equation 3.4). We need extra computation to get access to individual gradients, which results in VOGN being slower than Adam

or SGD. However, this is not a theoretical limitation, and this can be improved if a framework enables an easy computation of the individual gradients. Details of our implementation are described in Appendix B.1. Our implementation is much more efficient than a naive one, where gradients over examples are stored and the sum over the square is computed sequentially. Our implementation usually brings the running time of VOGN to within a factor of two of the time that Adam takes (although VOGN can take longer if we average over multiple MC samples).

8. **External damping factor:** We introduce an external damping factor γ, added to $\boldsymbol{s}_{j+1}$ in the denominator of Equation 3.3 (Zhang et al., 2018). This increases the lower bound of the eigenvalues of the diagonal covariance $\boldsymbol{\Sigma}$, preventing the step size and noise from becoming too large.

9. **Tuning VOGN:** The full list of hyperparameters in VOGN are summarised in Table 3.2. Currently, there is no common recipe for tuning the algorithmic hyperparameters for VI, especially for large-scale tasks like ImageNet classification. It is therefore important to record how we tuned these hyperparameters, so the community can use (and potentially improve upon) these methods. The key idea we use is to start with Adam hyperparameters and then make sure that VOGN training closely follows an Adam-like trajectory in the beginning of training. To achieve this, we divide the tuning into an *optimisation* part and a *regularisation* part.

   In the *optimisation* part, we tune the hyperparameters of a deterministic version of VOGN, called the Online Gauss-Newton (OGN) method. This method does not Monte-Carlo sample $\boldsymbol{w}_j \sim q_j(\boldsymbol{w})$, and instead sets $\boldsymbol{w}_j = \boldsymbol{\mu}_j$. OGN is therefore more stable than VOGN, and a convenient stepping stone when going from Adam/SGD to VOGN. OGN converges to a local minimum of the loss function (like Adam/SGD), and obtains a Laplace approximation instead of a variational approximation. We initialise OGN's parameter values at Adam's values, and tune until OGN is competitive with Adam/SGD.

   We then move to the *regularisation* part, where we tune the prior precision δ, warm-start the tempering parameter τ, and tune the number of MC samples K for VOGN.

Learning rate	α
Momentum rate	β_1
Exp. moving average rate	β_2
Prior precision	δ
External damping factor	γ
Tempering parameter	τ
# MC samples for training	K
Data augmentation factor	ρ

Table 3.2: The full list of hyperparameters in VOGN. The first four are shared with Adam, and the last four are specific to VOGN.

3.2.2 VOGN full-batch performance

In this section, we present experiments fitting several deep networks on CIFAR-10 and ImageNet. Our experiments demonstrate practical training using VOGN on these benchmarks and show performance that is competitive with Adam and SGD. We also assess the quality of the posterior approximation, finding that benefits of Bayesian principles are preserved.

CIFAR-10 (Krizhevsky and Hinton, 2009) contains 10 classes with 50,000 images for training and 10,000 images for validation. For ImageNet, we train with 1.28 million training examples and validate on 50,000 examples, classifying between 1,000 classes. We used a large minibatch size $M = 4{,}096$ and parallelise across 128 GPUs (NVIDIA Tesla P100). On CIFAR-10, we compare VOGN to Adam and MC-dropout (Gal and Ghahramani, 2016). On ImageNet, we also compare to SGD, K-FAC (Martens and Grosse, 2015; Osawa et al., 2018), and Noisy K-FAC (Zhang et al., 2018). We do not consider Noisy K-FAC for other comparisons since tuning is difficult.

We compare 3 architectures: LeNet-5, AlexNet and ResNet-18. We only compare to Bayes-by-Backprop (Blundell et al., 2015) for LeNet-5 on CIFAR-10 since it is very slow to converge for larger-scale experiments. We carefully set the hyperparameters of all methods, following the best practice of large distributed training (Goyal et al., 2017) as the initial point of our hyperparameter tuning. The full set of hyperparameters is in Appendix B.2, and code is available at `https://github.com/team-approx-bayes/dl-with-bayes`.

Performance on CIFAR-10 and ImageNet

Figures 3.7 and 3.8 compare the convergence of VOGN to Adam (for all experiments), SGD (on ImageNet), and MC-dropout (on the rest). VOGN shows similar convergence and its performance is competitive with these baselines. We also try Bayes-By-Backprop on

Algorithm 1 Variational Online Gauss Newton (VOGN)

Initialise $\boldsymbol{\mu}_0$, $\boldsymbol{s}_0$, $\boldsymbol{m}_0$.
$N \leftarrow \rho N$, $\tilde{\delta} \leftarrow \tau\delta/N$.
repeat
  Sample a minibatch $\mathcal{B}$ of size B.
  Split $\mathcal{B}$ into each GPU (local minibatch $\mathcal{B}_{\text{local}}$).
  for each GPU in parallel **do**
    for $k = 1, 2, \ldots, K$ **do**
      Sample $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
      $\boldsymbol{w}^{(k)} \leftarrow \boldsymbol{\mu} + \epsilon\boldsymbol{\sigma}$ with $\boldsymbol{\sigma} \leftarrow (1/(N(\boldsymbol{s} + \tilde{\delta} + \gamma)))^{1/2}$.
      Compute $\mathbf{g}_i^{(k)} \leftarrow \nabla_w \ell(\boldsymbol{y}_i, \sigma(\boldsymbol{f}_{w^{(k)}}(\boldsymbol{x}_i))), \forall i \in \mathcal{B}_{\text{local}}$ using the method described in Appendix B.1.
      $\hat{\mathbf{g}}_k \leftarrow \frac{1}{B}\sum_{i\in\mathcal{B}_{\text{local}}} \mathbf{g}_i^{(k)}$.
      $\hat{\boldsymbol{h}}_k \leftarrow \frac{1}{B}\sum_{i\in\mathcal{B}_{\text{local}}} \left(\mathbf{g}_i^{(k)}\right)^2$.
    end for
    $\hat{\mathbf{g}} \leftarrow \frac{1}{K}\sum_{k=1}^{K} \hat{\mathbf{g}}_k$ and $\hat{\boldsymbol{h}} \leftarrow \frac{1}{K}\sum_{k=1}^{K} \hat{\boldsymbol{h}}_k$.
  end for
  AllReduce $\hat{\mathbf{g}}$, $\hat{\boldsymbol{h}}$.
  $\mathbf{m} \leftarrow \beta_1\mathbf{m} + (\hat{\mathbf{g}} + \tilde{\delta}\boldsymbol{\mu})$.
  $\boldsymbol{s} \leftarrow (1 - \tau\beta_2)\boldsymbol{s} + \beta_2\hat{\boldsymbol{h}}$.
  $\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \alpha\mathbf{m}/(\boldsymbol{s} + \tilde{\delta} + \gamma)$.
until stopping criterion is met

LeNet-5, where it converges prohibitively slowly, performing poorly. We found it far simpler to tune VOGN because we can borrow all the techniques used for Adam. Figure 3.7 also shows the importance of DA in improving performance.

Table 3.3 gives a final comparison of train/validation accuracies, negative log-likelihoods, epochs required for convergence, and run-time per epoch. Table B.1 (in Appendix B) also includes standard deviations across many runs. We can see that the accuracy, log-likelihoods, and the number of epochs are comparable. VOGN is 2-5 times slower per epoch than Adam and SGD (we train for the same number of epochs). This is mainly due to the computation of the sum of squared gradients required in VOGN, as well as multiple Monte-Carlo samples in some cases. Overall, we clearly see that by using deep-learning techniques on VOGN, we can perform practical deep learning. This is not possible with Bayes-By-Backprop.

We see that Adam regularly overfits the training set in most settings, with large train-test differences in both validation accuracy and log-likelihood. LeNet-5 is an exception, and this is most likely due to the small architecture resulting in underfitting (this is consistent with the low validation accuracies obtained). In contrast to Adam, MC-dropout has small train-test gap, usually smaller than VOGN's. However, we will see later that this is because

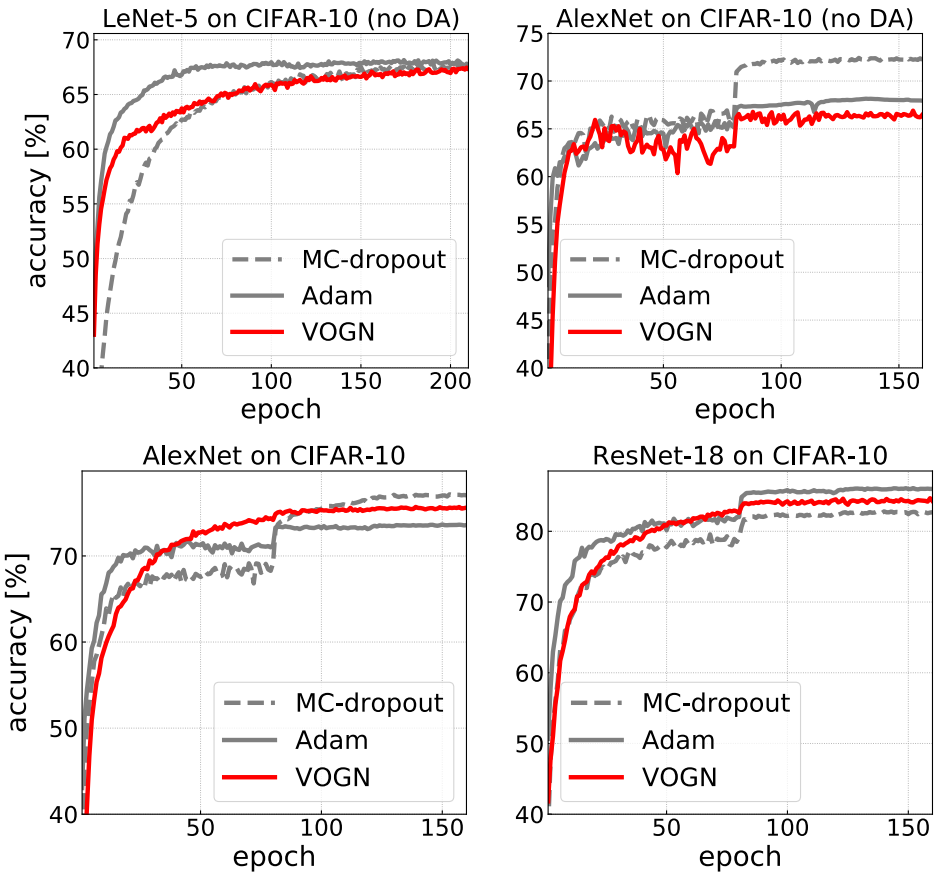


Figure 3.7: Validation accuracy for various architectures trained on CIFAR-10. Top row has no Data Augmentation (DA), bottom row has data augmentation. VOGN's convergence and validation accuracies are comparable to Adam and MC-dropout, except for on AlexNet without DA, highlighting the benefits of DA for VOGN.

of underfitting. Moreover, the performance of MC-dropout is highly sensitive to the dropout rate (see Appendix B.4 for a comparison of different dropout rates). On ImageNet, Noisy K-FAC performs well too. It is slower per epoch than VOGN, but it takes fewer epochs. Overall, wall clock time is about the same as VOGN.

Due to the Bayesian nature of VOGN, there are some trade-offs to consider:

1. Reducing the prior precision δ results in higher validation accuracy, but also larger train-test gap (more overfitting). As an example, training VOGN on ResNet-18 on ImageNet with a prior variance of $7.5e-4$ has train-test accuracy and log-likelihood gaps of 2.29 and 0.12 respectively. When the prior variance is increased to $7.5e-3$ (prior precision is decreased), the respective train-test gaps increase to 6.38 and 0.34 (validation accuracy and validation log-likelihood also increase, see Figure 3.9). As

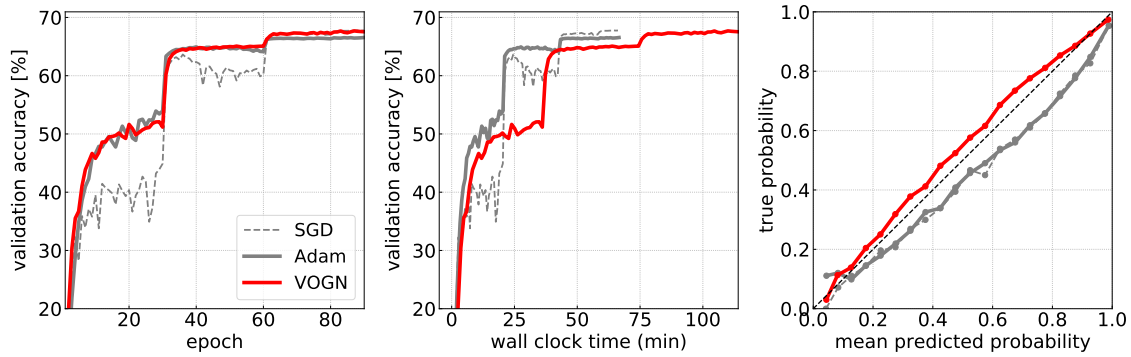


Figure 3.8: The two left plots show that VOGN performs well on ImageNet, reaching a similar validation accuracy to SGD in the same number of epochs. However, it is slower per epoch due to the additional computation required to calculate the Gauss-Newton matrix approximation, making it slightly slower in wall-clock time. The right plot shows the calibration curve, and we see that VOGN gives calibrated probabilities (the diagonal represents perfect calibration).

expected, when the prior precision is small, VOGN reaches converged solutions more like non-Bayesian methods, where overfitting is an issue.

We also show the effect of changing the effective dataset size ρN in Appendix B.3: note that, since we are going to tune the prior precision δ anyway, it is sufficient to set ρ to its correct order of magnitude.

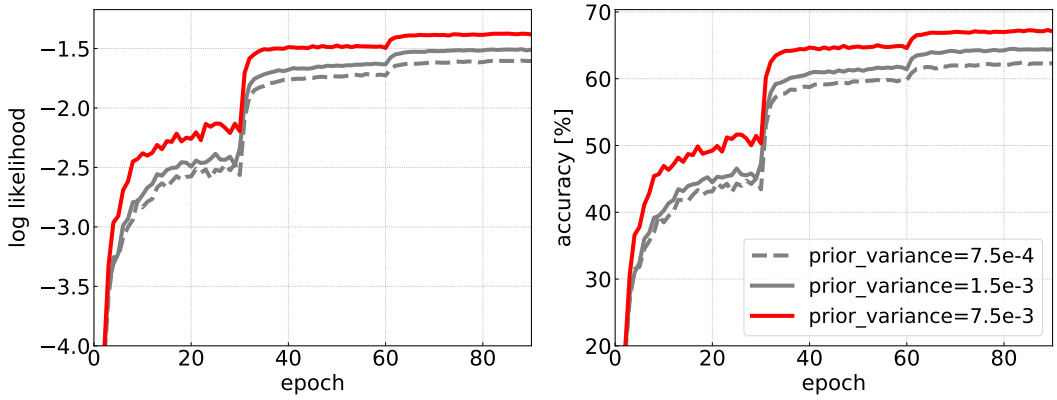


Figure 3.9: Effect of prior variance on VOGN training ResNet-18 on ImageNet. As we increase the prior variance (decrease the prior precision δ), validation performance improves, as measured by validation log-likelihood and validation accuracy. However, increasing the prior variance also leads to a larger train-test gap (see main text), indicating more overfitting.

2. Increasing the number of training Monte-Carlo samples (up to a limit) improves VOGN's convergence rate and stability, but also increases the computation. Increasing the number of Monte-Carlo samples during testing improves generalisation, as expected due to averaging. This trade-off is shown in Figure 3.10.

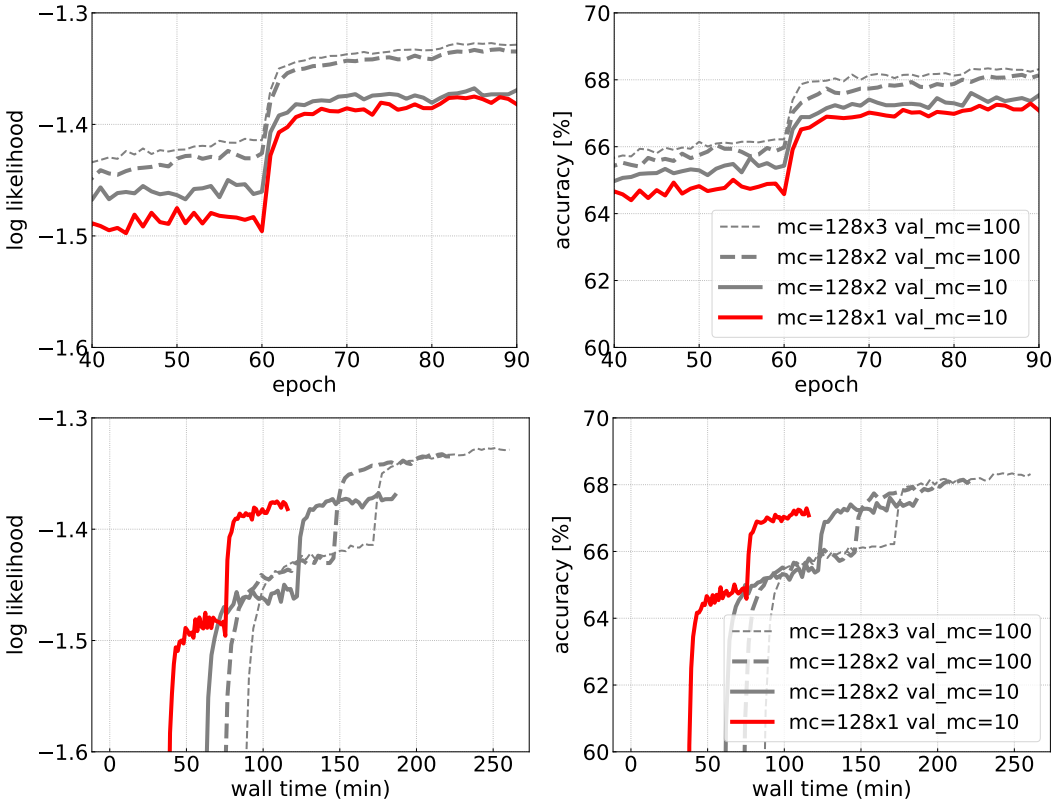


Figure 3.10: Effect of changing the number of training and testing Monte-Carlo samples on VOGN training ResNet-18 on ImageNet. Left plots are validation log likelihood, and right plots are accuracy. '`mc`' refers to the number of training Monte-Carlo samples (there are 128 GPUs, and we are using Monte-Carlo sample parallelism, so '`mc=128x2`' indicates two training Monte-Carlo samples per GPU). '`val_mc`' refers to the number of validation Monte-Carlo samples. As we increase the number of training Monte-Carlo samples, computation cost (and hence wall-clock time) increases, but we get better convergence rate and stability (reduced number of epochs). As we increase the number of testing Monte-Carlo samples, computation again increases (wall-clock time is increasing here because we calculate validation accuracy every epoch), but generalisation improves.

Dataset/ Architecture	Optimiser	Train/Validation Accuracy (%)	Validation NLL	Epochs	Time/ epoch (s)	ECE	AUROC
CIFAR-10/ LeNet-5 (no DA)	Adam	71.98 / **67.67**	**0.937**	210	6.96	**0.021**	0.794
	BBB	66.84 / 64.61	1.018	800	11.43†	0.045	0.784
	MC-dropout	68.41 / **67.65**	0.99	210	6.95	0.087	**0.797**
	VOGN	70.79 / **67.32**	**0.938**	210	18.33	0.046	**0.8**
CIFAR-10/ AlexNet (no DA)	Adam	100.0 / 67.94	2.83	161	3.12	0.262	0.793
	MC-dropout	97.56 / **72.20**	1.077	160	3.25	0.140	**0.818**
	VOGN	79.07 / 69.03	**0.93**	160	9.98	**0.024**	0.796
CIFAR-10/ AlexNet	Adam	97.92 / 73.59	1.480	161	3.08	0.262	0.793
	MC-dropout	80.65 / **77.04**	**0.667**	160	3.20	0.114	0.828
	VOGN	81.15 / 75.48	0.703	160	10.02	**0.016**	**0.832**
CIFAR-10/ ResNet-18	Adam	97.74 / **86.00**	0.55	160	11.97	0.082	**0.877**
	MC-dropout	88.23 / 82.85	0.51	161	12.51	0.166	0.768
	VOGN	91.62 / 84.27	**0.477**	161	53.14	**0.040**	**0.876**
ImageNet/ ResNet-18	SGD	82.63 / **67.79**	**1.38**	90	44.13	0.067	0.856
	Adam	80.96 / 66.39	1.44	90	44.40	0.064	0.855
	MC-dropout	72.96 / 65.64	1.43	90	45.86	**0.012**	0.856
	OGN	85.33 / 65.76	1.60	90	63.13	0.128	0.854
	VOGN	73.87 / **67.38**	**1.37**	90	76.04	0.029	0.854
	K-FAC	83.73 / 66.58	1.493	60	133.69	0.158	0.842
	Noisy K-FAC	72.28 / 66.44	1.44	60	179.27	0.080	0.852

Table 3.3: Performance comparisons on different dataset/architecture combinations. Out of the 15 metrics (NLL, ECE, and AUROC on 5 dataset/architecture combinations), VOGN performs best or tied best on 10, and is second-best on the other 5. Here DA means 'Data Augmentation', NLL refers to 'Negative Log-Likelihood' (lower is better), ECE refers to 'Expected Calibration Error' (lower is better), AUROC refers to 'Area Under ROC curve' (higher is better), with further explanations of these metrics in Appendix B.5. BBB is Bayes-By-Backprop (Blundell et al., 2015). For ImageNet, the reported accuracy and negative log-likelihood are the median value from the final 5 epochs. All hyperparameter settings are in Appendix B.2. See Table B.1 for standard deviations across many runs. † BBB is not parallelised (other methods have 4 processes), and uses 1 MC sample during training of the convolutional layers (VOGN uses 6 samples per process).

Quality of predictive probabilities

We now compare the quality of predictive probabilities for the various methods. For Bayesian methods, we compute these probabilities by averaging over samples from the posterior approximations. For non-Bayesian methods, these are obtained using the point estimates of the weights. We compare predictive probabilities using the following metrics: validation Negative Log-Likelihood (NLL), Area Under ROC curve (AUROC) and Expected Calibration Error (ECE) (Naeini et al., 2015; Guo et al., 2017). For the first and third metric, a lower number is better, while for the second, a higher number is better. See Appendix B.5 for a detailed explanation of each of these metrics. Results are summarised in Table 3.3.

VOGN's uncertainty performance is more consistent and marginally better than the other methods, as expected from a more principled Bayesian method. Out of the 15 metrics (NLL, ECE and AUROC on 5 dataset/architecture combinations), VOGN performs the best or tied best on 10, and is second-best on the other 5. In contrast, both MC-dropout's and Adam's performance varies significantly, sometimes performing poorly, sometimes performing decently. MC-dropout is best on 4, and Adam is best on 1 (on LeNet-5; as argued earlier, the small architecture may result in underfitting).

We also show calibration curves (DeGroot and Fienberg, 1983) in Figures 3.8 and 3.11. Adam is consistently over-confident, with its calibration curve below the diagonal. Conversely, MC-dropout is usually under-confident, with its curve above the diagonal. On ImageNet, MC-dropout performs well on ECE (all methods are very similar on AUROC), but this required an excessively tuned dropout rate (see Appendix B.4).

We also compare performance on out-of-distribution (OOD) datasets. We want methods to make more confident predictions on in-distribution data (such as the validation set from the dataset it was trained on), and more uncertain predictions when testing on images that are different from the training datasets. We use experimental protocol from the literature (Hendrycks and Gimpel, 2017; Lee et al., 2018; DeVries and Taylor, 2018; Liang et al., 2018) to compare VOGN, Adam and MC-dropout. Using trained architectures (LeNet-5, AlexNet and ResNet-18) on CIFAR-10, we test on SVHN, LSUN (crop) and LSUN (re-size) as out-of-distribution datasets, with the in-distribution data given by the validation set of CIFAR-10 (10,000 images). The entire training set of SVHN (73,257 examples, 10 classes) (Netzer et al., 2011) is used. The test set of LSUN (Large-scale Scene UNderstanding dataset (Yu et al., 2015), 10,000 images from 10 different scenes) is randomly cropped to obtain LSUN (crop), and is down-sampled to obtain LSUN (re-size). These out-of-distribution datasets have no overlapping classes with CIFAR-10.

We also borrow metrics from other works (Hendrycks and Gimpel, 2017; Lakshminarayanan et al., 2017). We plot predictive entropy histograms in Figure 3.12 and Ap-

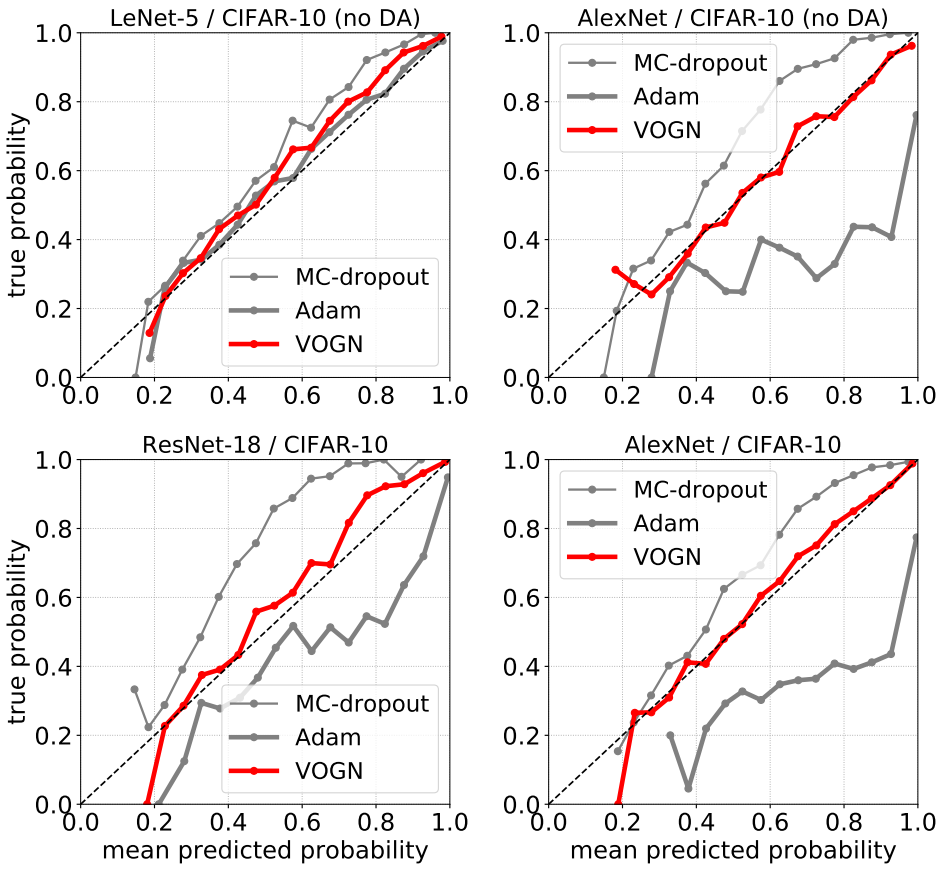


Figure 3.11: Calibration curves for various architectures trained on CIFAR-10. The top row has no data augmentation (DA), while the bottom row has data augmentation. VOGN is extremely well-calibrated compared to the other two methods (except for LeNet-5, where all methods perform well). The calibration curve for ResNet-18 on ImageNet is in Figure 3.8.

pendix B.6, where the predictive entropy per input image is the entropy of the distribution over predicted labels. Ideally, on out-of-distribution data, a model would have high predictive entropy, indicating it is unsure of which class the input image belongs to. In contrast, for in-distribution data, good models should have many examples with low entropy, as they should be confident of many input examples' (correct) class. As in the literature, we also report AUROC and FPR at 95% TPR (this is a different AUROC to that in Table 3.3). By thresholding the most likely class' softmax output, we assign high uncertainty images to belong to an unknown class. This allows us to calculate the FPR and TPR, allowing the ROC curve to be plotted, and the AUROC to be calculated.

On ResNet-18 and AlexNet, VOGN's predictive entropy histograms show the desired behaviour: a spread of entropies for the in-distribution data, and high entropies for out-of-distribution data (see Figure 3.12 and Appendix B.6). Adam has many predictive entropies at

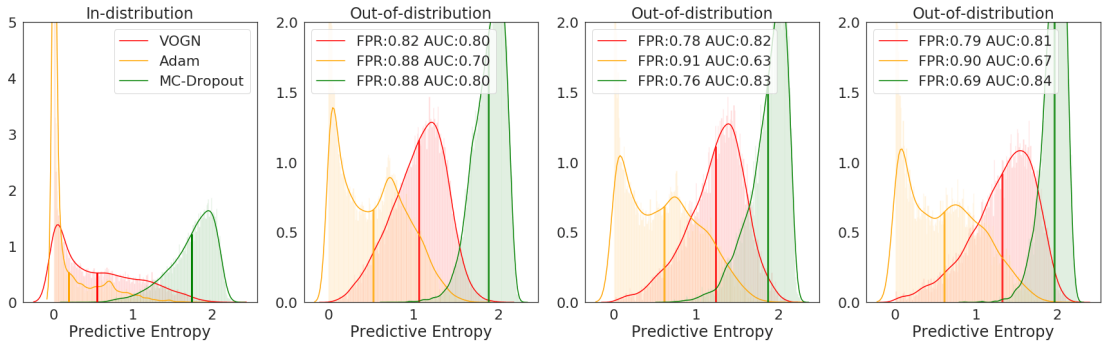


Figure 3.12: Histograms of predictive entropy for out-of-distribution tests for ResNet-18 on CIFAR-10. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are FPR at 95% TPR (lower is better) and AUROC (higher is better), averaged over 3 runs (standard deviations are very small). We clearly see that VOGN's predictive entropy is generally low for in-distribution and high for out-of-distribution data, but this is not the case for other methods. Solid vertical lines indicate the mean predictive entropy.

zero, indicating that Adam tends to classify out-of-distribution data too confidently. Conversely, MC-dropout's predictive entropies are generally high (particularly in-distribution), indicating MC-dropout has too much noise. On LeNet-5, we observe the same result as before: Adam and MC-dropout both perform well. The metrics (AUROC and FPR at 95% TPR) do not provide a clear story across architectures.

Conclusions

We successfully used VOGN to train architectures at large-scale, and analysed performance on a variety of architectures on CIFAR-10 and ImageNet. We used VOGN's similarity with Adam to borrow deep-learning techniques from the community to scale to this size. This was particularly difficult for Bayes-By-Backprop. VOGN's accuracies and convergence rates are similar to SGD and Adam, but VOGN also retains some benefits of Bayesian principles, with well-calibrated uncertainty, good performance on out-of-distribution data, and as we next explore, good continual learning performance. We hope VOGN's quicker convergence will lead to improvements over VCL from Section 3.1 (Tseran et al., 2018; Eschenhagen, 2019).

Benchmark	Metric	Improved VCL	VOGN
Split MNIST	ACC (%)	98.5±0.4	98.8±0.1
	Epochs	600	**100**
Permuted MNIST	ACC (%)	93±1	94.0±0.8
	FWT (%)	-0.2±0.1	-0.6±0.1
	BWT (%)	-4±1	-4±1
	Epochs	800	**100**†
Split CIFAR	ACC (%)	48.8±2.2	**74.4±0.4**
	FWT (%)	0.8±2.0	1.8±0.3
	BWT (%)	-29±4	-0.7±0.5
	Epochs	5000	**600**

Table 3.4: Final average test accuracy of VCL and VOGN on Split MNIST, and accuracy and forward/backward metrics on Permuted MNIST and Split CIFAR. Mean performance and standard deviation over 5 runs. Metrics and benchmarks are defined in Section 2.4. VOGN converges to the same performance as VCL on the MNIST benchmarks, requiring fewer epochs. VOGN also scales well to the larger Split CIFAR tasks, now outperforming previous methods (the best-performing method from Table 3.1 was SI, with 73.5±0.5% accuracy). The number of epochs per task reported for Split CIFAR is for tasks 2-6. The number of epochs for task 1 is 1/10th of this number, as there is 10 times more datapoints in task 1. † Improved VCL required a minibatch size of 1024 to perform well on Permuted MNIST, whereas VOGN has a minibatch size of 256. VOGN's hyperparameters are listed in Appendix A.2 (Eschenhagen, 2019).

3.2.3 VOGN continual learning performance

Having scaled variational inference to large scales in the batch-setting, we now run our algorithm VOGN on continual learning benchmarks. We run VOGN on the same benchmarks as VCL: Split MNIST, Permuted MNIST and Split CIFAR. As VOGN is optimising the same variational objective as VCL, and we are running both to convergence, we hope that both methods would obtain the same performance on the MNIST benchmarks. But we also hope that by using the techniques from Section 3.2.1, VOGN will require fewer epochs to converge to this performance. We also note that every epoch is quicker for VOGN than in VCL, as VCL uses the slower Bayes-By-Backprop method to individually optimise for means and variances of the Gaussian distribution.

We summarise results in Table 3.4. Hyperparameters for VOGN's results are in Appendix A.2 (Eschenhagen, 2019). We find that, as desired, VOGN reaches the same average accuracy on the MNIST-based continual learning benchmarks as VCL, while requiring far fewer epochs to do so. When we look at forward and backward transfer metrics, we see that VOGN has the same BWT as VCL, but slightly worse FWT. This is because VOGN has a

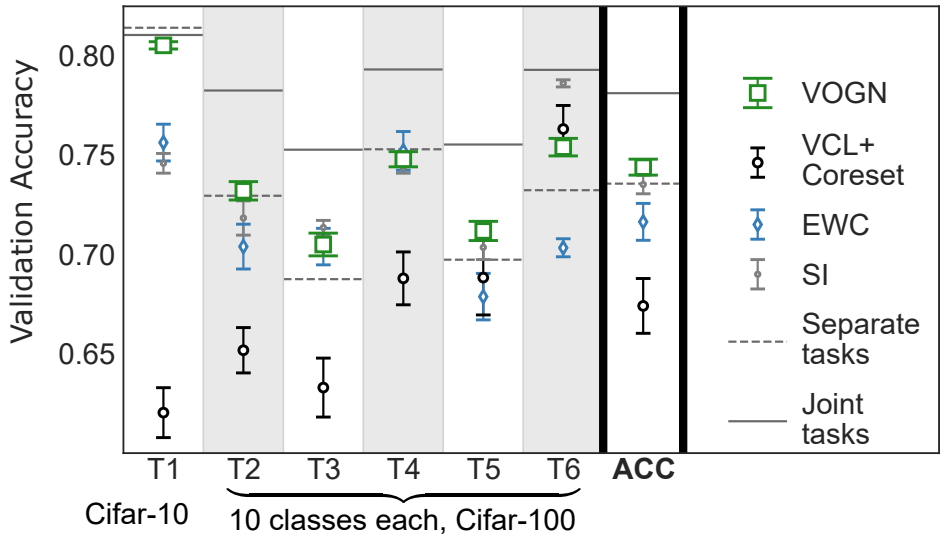


Figure 3.13: Individual task accuracy after training on the final task, and average task accuracy (ACC). VOGN outperforms other weight-regularisation techniques on this benchmark, but there is still potential for improvement as VOGN's performance is still far from the Joint Tasks baseline. We do not plot VCL results as VCL performs very poorly, and instead only plot VCL+Coreset, which also requires storing some datapoints in memory.

higher single task accuracy than VCL, and this makes comparing the FWT metric between methods difficult (see Section 2.4 for definitions of these metrics). Despite this, VOGN has comparable average accuracy, which is the most important metric when comparing algorithms.

This faster convergence also allows us to successfully scale VOGN to larger benchmarks such as the Split CIFAR benchmark, where VOGN performs well, and better than our weight-regularisation baselines, as seen in Figure 3.13. We also see that VOGN has better forward transfer and backward transfer metrics than EWC and VCL on this benchmark (see Table 3.1 for metrics for EWC). However, there is still a sizeable gap between VOGN's final accuracy and the Joint Tasks upper-bound on performance, indicating that there is still potential for improvement on this benchmark.

Finally, we comment that it is possible to incorporate a coreset with VOGN in exactly the same way as with VCL. We would expect this to improve results similar to how it improves results for VCL, in particular on larger-scale benchmarks. However, this has not yet been implemented.

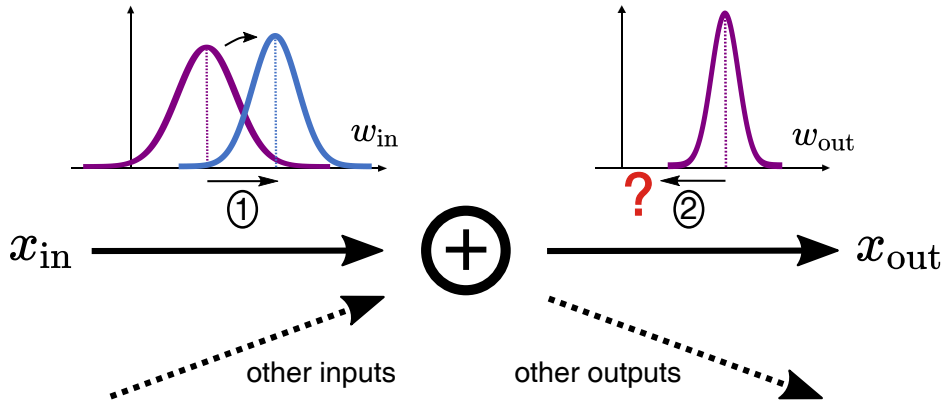


Figure 3.14: An illustration to show that independence between weights of different layers can lead to incorrect regularisation, meaning that our predictions x_{out} are not maintained. We consider a single unit in our neural network, and a single input weight w_{in} and output weight w_{out}. After training on previous tasks, we obtain independent Gaussian distributions over these weights, shown in purple. When we see new data, these distributions act as the prior which we regularise towards. Step 1 (top left): During training, the distribution over input weight w_{in} moves to the right. Step 2 (top right): Ideally, we would regularise towards maintaining predictions x_{out}, leading to a signal to move w_{out}. However, as the two weights are independently regularised, the distribution over w_{out} may not move sufficiently. For example, if w_{out} has small uncertainty, then w_{out} may be overly regularised, and our predictions x_{out} are not maintained.

3.3 Failures of weight-space continual learning

We have seen how weight-space variational neural networks use pruning to obtain good performance in continual learning, and how we can speed up convergence by employing natural-gradient updates. Although we obtained reasonable performance on the larger Split CIFAR benchmark, and outperformed other methods, there is still potential for improvement: VOGN is still significantly worse than the Joint Tasks baseline.

In this section, we argue that this is because of fundamental problems with approximations we had to make in order to train weight-priors on neural networks. Specifically, we had to assume independence across weights of our neural network, an approximation that was necessary in order to scale to larger neural network architectures. This independence means that when lower-layer weights move, upper-layer weights may not move correctly to maintain an output given an input. For example, if upper-layer weights have small uncertainty, they may be overly regularised, meaning there is insufficient ability to update to counteract a lower-layer weight changing. This only happens as the weights are *independently regularised*. We explain this argument schematically in Figure 3.14.

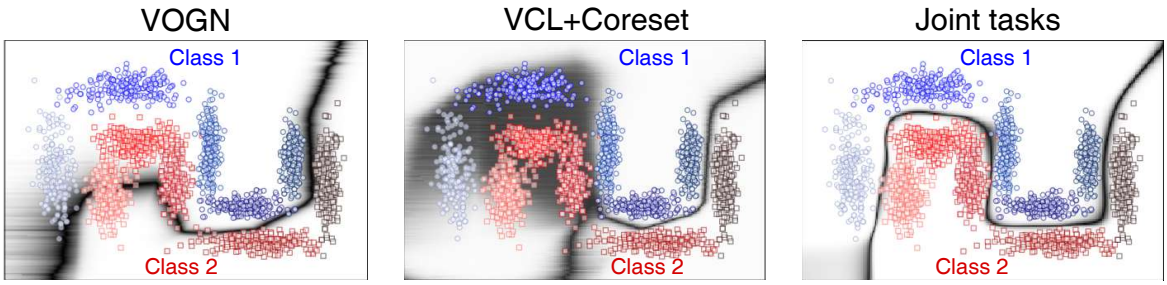


Figure 3.15: VOGN and VCL+Coreset perform badly when continually trained over the 5 tasks of the Toy-Gaussians benchmark (we do not show VCL, which performs even worse). When compared to the Joint Tasks upper-bound baseline, there are undesirable artefacts appearing on old task data (older tasks are on the left of the plots). We plot the middle-performing run of 5 runs for each method. Performance is summarised in Table 3.5.

	VCL	VOGN	VCL+Coreset	Joint Tasks
Train accuracy (%)	68±8	79±11	92±10	99.70±0.03

Table 3.5: The train accuracy (averaged over all 5 tasks) of VCL, VOGN and VCL+Coreset (randomly chosen 20 points per task) is worse than the highly accurate Joint Tasks upper-bound baseline on the Toy-Gaussians benchmark. This illustrates how weight-space methods can be brittle even on simple problems, because weights in the neural network are independently regularised. Hyperparameters for all methods are in Appendix A.3, and visualisations of the middle-performing runs are in Figure 3.15.

We can show the negative effects of this problem on a simple toy example, specifically the Toy-Gaussians example from Section 2.4. We use a two-hidden-layer network with 20 hidden units per layer. As the Toy-Gaussians benchmark introduces new datapoints in a different region of input-space, it requires predictions on old task data to be maintained, and as weights change for new tasks, this may affect previous regions of input-space if not correctly regularised. In Figure 3.15 we train our methods on the five tasks sequentially, and plot the final decision boundary after training on the last task. Hyperparameters are in Appendix A.3. We see how VOGN and VCL+Coreset perform badly across many random seeds, with undesirable artefacts appearing on old task data (older tasks are on the left of the plots). Performance is summarised in Table 3.5, where we report mean performance and standard deviation over 5 runs. Joint Tasks performance is very high, and we would want our continual learning method to be very close to this performance on such a simple benchmark. This is not the case for any of our weight-regularisation methods.

One way of overcoming this problem is to introduce correlations between weights in the neural network, instead of using a mean-field approximation. K-FAC methods (Martens and

Grosse, 2015; Ritter et al., 2018) introduced Kronecker-factored correlations between weights in the same layer, and they observed improved results on continual learning benchmarks. However, there is still no method that introduces correlations between weights of different layers while scaling to larger neural networks.

A different way to overcome this problem is to directly regularise the output of the neural network instead of only regularising in weight-space. We will still have to make approximations to scale any such method, but by making these approximations in function-space instead of weight-space, we hope for better continual learning performance. We will look at methods for doing this in Chapter 4.

3.4 Summary

We started this chapter with Variational Continual Learning (VCL) (Nguyen et al., 2018), a variational weight-prior method, and improved its performance by running for much longer and using a few tricks to speed up convergence rate. We improved performance on Split MNIST and Permuted MNIST, but this came at the cost of long training times. When we analysed why performance increased, we found that running for longer led to increasingly pruned solutions in our variational Bayesian neural network. This pruning has been shown to lead to performance decreases in other work due to underfitting. However, we saw how it can help in continual learning.

We then focussed on speeding up convergence rate for our weight-space variational continual learning method. We did this by using natural-gradient update steps. We scaled up an algorithm called Variational Online Gauss-Newton (VOGN), finding that we could get competitive performance on a large-scale (ImageNet/ResNets) for the first time, while preserving some benefits of Bayesian principles. When we applied VOGN to continual learning benchmarks, we found that VOGN performed as well as our Improved VCL, but required far fewer epochs to converge (and is also quicker per epoch). This allowed it to scale to the larger Split CIFAR benchmark, where it performs competitively with other weight-regularisation baselines.

However, there is still potential for performance improvement on Split CIFAR. We argued that weight-space regularisation has problems: when we regularise weights independently (especially across layers), this leads to brittleness and forgetting over tasks. We showed this happening on a simple two-dimensional continual learning problem (Toy-Gaussians).

There has been follow-up work not discussed in detail in this chapter. In Loo et al. (2021), we introduced Generalised VCL (GVCL), which tempers the KL-to-prior term in the variational objective function. This modification to VCL recovers Online EWC (Schwarz

et al., 2018) as a limiting case, allowing for interpolation between the two approaches. We also introduced task-specific FiLM layers to take advantage of and reduce pruning in variational Bayesian neural networks, finding that this also leads to improved performance.

In Chapter 4 we next focus on function-space regularisation (instead of weight-space regularisation as in this chapter). By regularising in function-space, we hope that we will avoid problems stemming from independently-regularised weights. We will start from the same variational objective function, but approximate a term with one in function-space. We will see improved performance on our benchmarks, but we will also store (a few) datapoints from past tasks, similar to the VCL+Coreset method. Such methods will therefore be a combination of regularisation-based and rehearsal-based approaches to continual learning. In Chapter 5 we will then combine weight and function-regularisation approaches, theoretically analysing when and how they work. Such analysis will also allow us to propose simple ways to improve weight-prior algorithms with functional regularisation (see Section 5.5).

Chapter 4

Functional regularisation of memorable past

In this chapter, we tackle continual learning by performing *functional regularisation* of neural networks, directly considering the outputs or functions of neural networks. We will derive an algorithm called Functional Regularisation of Memorable Past (FROMP). This functional regularisation is in contrast to the *weight regularisation* we performed in Chapter 3, where we improved on previous probabilistic continual learning algorithms, but found that there were still problems.

In Section 3.3 we argued that weight regularisation's problems in continual learning might stem from restrictive independence assumptions that we had to make. Weight-priors make current weights closer to the previous ones, but this may not always ensure that the predictions on the past tasks also remain unchanged. A better approach is to directly regularise the outputs, because what ultimately matters is the network output, not the values of the weights. We therefore now consider such approaches, but still within a probabilistic framework for continual learning.

In order to regularise in function-space, we choose to store some datapoints from the past. By only storing very few datapoints from the past, our memory cost will be significantly lower than if we had stored all past data. However, even this may not always be allowed in a continual learning setting, such as if there are strict privacy constraints (see Section 2.1). In these stricter cases, we could potentially store pseudo-inputs instead of real data, and our theory allows for this. However, we will not explicitly consider algorithms using pseudo-inputs in this chapter.

We start in Section 4.1 with simple ideas for functional regularisation, summarising some previous work in this area. We then introduce our method, and derive detailed equations in

Sections 4.2 to 4.4. We present experimental results in Section 4.5, where we see our method perform very well, and conclude in Section 4.6.

In Chapter 5 we will introduce a framework which will allow us to further theoretically analyse why our methods in this chapter work so well in practice, and also allow us to suggest improvements (see Section 5.6).

4.1 Functional regularisation of neural networks

In Section 3.3 we saw problems with weight regularisation, and motivated functional regularisation of neural networks for continual learning. In this section, we consider various ideas for doing this at a high-level, starting with simple ideas and ending with recent works. We will assume scalar outputs to ease notation.

We start by returning to our assumption that the best possible performance is given by a model trained on all data at once (we explicitly made this assumption in Section 2.1). In probabilistic continual learning with variational inference, this corresponds to the Joint Tasks loss,

$$\mathcal{L}_t^{\text{Joint}}(\boldsymbol{\eta}_t) = \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[-\log p(\mathcal{D}_t|\boldsymbol{w}) - \log p(\mathcal{D}_{1:t-1}|\boldsymbol{w})\right] + \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[\log \frac{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}{p(\boldsymbol{w})}\right]. \quad (4.1)$$

We now quickly recap the definitions of each of these terms, although the reader can also look at Chapter 2 for a full description. We are optimising the variational bound (the negative ELBO) $\mathcal{L}_t^{\text{Joint}}$ at task t with respect to $\boldsymbol{\eta}_t$, which are the parameters of our approximating family distribution $q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, where $\boldsymbol{w} \in \mathbb{R}^P$ are the weights in our model. We have a prior $p(\boldsymbol{w})$, and each task s has data $\mathcal{D}_s = \{\boldsymbol{x}_i, y_i\}_{i=1}^{N_s}$. The negative log-likelihood of dataset s is $-\log p(\mathcal{D}_s|\boldsymbol{w}) = \sum_{i\in\mathcal{D}_s} \ell(y_i, h(f_{\boldsymbol{w}}(\boldsymbol{x}_i)))$, where we have model $f_{\boldsymbol{w}}(\boldsymbol{x})$, and the loss $\ell(y, h(f))$ is a differentiable loss function (such as cross-entropy) between a label y and a neural network output $h(f)$, where $h(\cdot)$ is an inverse link function.

This Joint Tasks loss is our ideal objective, but we cannot directly optimise it as we are not allowed to store all past data $\mathcal{D}_{1:t-1}$. We now make the assumption that we are allowed to store a small subset of previous data (like in the VCL+Coreset method from Section 2.2.1 and Chapter 3). This corresponds to rehearsal-based approaches to continual learning (see discussion in Section 2.3).

Experience replay

We have written the Joint Tasks loss in a variational framework in Equation 4.1, but many works only consider the deterministic version, which we wrote in Equation 2.2. We describe some of these works in this subsection.

The simplest scheme to approximate $\mathcal{L}_t^{\text{Joint}}$ would simply store some data from each previous task's data $\mathcal{D}_s$ for $s = 1, 2, ..., t-1$, and approximate each log-likelihood term as,

$$-\log p(\mathcal{D}_s|\boldsymbol{w}) = \sum_{i \in \mathcal{D}_s} \ell(y_i, h(f_{\boldsymbol{w}}(\boldsymbol{x}_i))) \approx (N_s/M_s) \sum_{i \in \mathcal{M}_s} \ell(y_i, h(f_{\boldsymbol{w}}(\boldsymbol{x}_i))), \tag{4.2}$$

where we store a small subset of the full past-task dataset $\mathcal{D}_s$ (the size of $|\mathcal{D}_s| = N_s$) in a memory set $\mathcal{M}_s$ (the size of $|\mathcal{M}_s| = M_s$). This simple approach is known as Replay (or experience replay) (Ratcliff, 1990; Robins, 1995; Rolnick et al., 2019), usually applied to the loss in Equation 2.2.

We would like to perform better than this simple approach, and many works have tried to do so. One idea is to replace the label y_i in Equation 4.2 with the output using the previous model $f_{\boldsymbol{w}_{t-1}}(\boldsymbol{x}_i)$, like in knowledge distillation (Hinton et al., 2015; Rebuffi et al., 2017). Another idea (Lopez-Paz and Ranzato, 2017; Chaudhry et al., 2019) is to add a constraint during training such that the loss over previous data does not worsen. We leave a detailed discussion of such methods to Chapter 5, as it is closely related to concepts introduced there.

Other works employ a different approach, directly using an L_2-regulariser over the function values from past tasks,

$$\tfrac{1}{2}\tau \sum_{s=1}^{t-1} (\boldsymbol{f}_{t,s} - \boldsymbol{f}_{t-1,s})^\top (\boldsymbol{f}_{t,s} - \boldsymbol{f}_{t-1,s}), \tag{4.3}$$

where $\boldsymbol{f}_{t,s}$ and $\boldsymbol{f}_{t-1,s}$ are vectors of function values $f_{\boldsymbol{w}}(\boldsymbol{x}_i)$ and $f_{\boldsymbol{w}_{t-1}}(\boldsymbol{x}_i)$ respectively for all $\boldsymbol{x}_i \in \mathcal{D}_s$, where $\mathcal{D}_s$ is the dataset for previous task s, and τ is a constant scaling factor. Like before, we cannot store all past data, and so a simple idea is to randomly sample past data to approximate the regulariser terms. This is the key idea behind Dark Experience Replay (Buzzega et al., 2020). They apply their algorithm to the no-task-boundary setting, and so need to maintain a subset of memory that can be updated online as new datapoints are seen, and this is achieved through the reservoir sampling algorithm (Vitter, 1985). Dark Experience Replay++ (Buzzega et al., 2020) further includes the experience replay term (Equation 4.2) in their objective function, thereby hoping to get benefits of both L_2-regularisation and experience replay.

Other works (such as Benjamin et al. (2019)) match the output of the neural network, after the inverse link function,

$$\frac{1}{2}\tau \sum_{s=1}^{t-1} (\boldsymbol{h}_{t,s} - \boldsymbol{h}_{t-1,s})^\top (\boldsymbol{h}_{t,s} - \boldsymbol{h}_{t-1,s}), \tag{4.4}$$

where $\boldsymbol{h}_{t,s}$ and $\boldsymbol{h}_{t-1,s}$ are vectors of values $h(f_{\boldsymbol{w}}(\boldsymbol{x}_i))$ and $h(f_{\boldsymbol{w}_{t-1}}(\boldsymbol{x}_i))$ respectively for all $\boldsymbol{x}_i \in \mathcal{M}_s$. Note that for logistic regression, $h(f) = \sigma(f)$ is the sigmoid function, and for linear regression, $h(f)$ is a constant value, meaning that Equation 4.4 is the same as Equation 4.3 up to a constant. Benjamin et al. (2019) also store randomly-sampled past data (which they call a 'working memory'). Our work in this chapter will improve upon such methods.

Variational functional regularisation

We are interested using functional regularisation within the probabilistic framework, for example applying to the variational objective in Equation 4.1. Rather than using the techniques discussed so far to approximate the log-likelihood of past data, a better approach might be to frame functional regularisation directly using the variational objective. This is what we will do in this chapter.

Titsias et al. (2020) also frame functional regularisation in a variational framework, and derive an algorithm called Functional Regularisation for Continual Learning (FRCL). They employ key ideas from Gaussian Processes (GPs) and sparse inducing point GPs (Csató and Opper, 2002; Titsias, 2009). This is arguably the most related work to our approach in this chapter, and also performs well in practice, but we will see several key theoretical differences throughout this chapter (which also lead to our approach performing better on benchmarks).

FRCL treat the last (output) layer of the neural network in a Bayesian way, considering it in function-space as a GP over function values $f(\boldsymbol{x})$. The remaining layers in the neural network are treated as a feature extractor $\boldsymbol{\phi}_\theta(\boldsymbol{x})$, and the kernel of the final-layer GP is a dot product of these features. Using a sparse GP framework, they can write down the ELBO in terms of the feature extractor parameters θ and the mean and variance of the output-layer Gaussian-distributed weights $q(\boldsymbol{w}_{\text{out}}) = \mathcal{N}(\boldsymbol{w}_{\text{out}}; \boldsymbol{\mu}_{\text{out}}, \boldsymbol{\Sigma}_{\text{out}})$. They optimise the sum of

ELBOs of each task, which simplifies to optimising,

$$\mathcal{L}_t^{\text{FRCL}}(\theta, \boldsymbol{\mu}_{\text{out}}, \boldsymbol{\Sigma}_{\text{out}}) = \sum_{i \in \mathcal{D}_t} \mathbb{E}_{q(\boldsymbol{w}_{\text{out}})} \left[-\log p(y_i | \boldsymbol{w}_{\text{out}}^\top \boldsymbol{\phi}_\theta(\boldsymbol{x}_i)) \right] + \mathcal{KL}\left[q(\boldsymbol{w}_{\text{out}}) \| p(\boldsymbol{w}_{\text{out}}) \right] + \sum_{s=1}^{t-1} \mathcal{KL}\left[q(\boldsymbol{u}_s) \| p_\theta(\boldsymbol{u}_s) \right], \quad (4.5)$$

where $p(\boldsymbol{w}_{\text{out}})$ is the prior over output-layer weights, and $\boldsymbol{u}_s$ are inducing points that summarise previous task s, with distribution $q(\boldsymbol{u}_s)$. They can also use this ELBO to select inducing points after training for the parameters of the neural network, although in practice they select a subset of data by discrete optimisation of the trace of the covariance matrix of the prior GP conditional (this appears in the variational sparse GP ELBO (Titsias, 2009)).

They obtain decent performance on a variety of tasks including Split MNIST and Permuted MNIST. However, there is still potential for improving upon FRCL, and we realise some of this potential in this chapter. First, FRCL uses a kernel only over the output-layer network weights, whereas we consider a kernel over *all* network weights. This is especially important in the early stages of learning, when all the weights are changing and uncertainties are larger (see Appendix C.5 for an example). Second, FRCL's functional prior (the last term in Equation 4.5) does not regularise the mean of the current network to be close to the previous network's mean, instead only encouraging the kernel to remain close to the inducing points' kernel. This is less interpretable and can result in the (mean of the) network's output changing unpredictably. Third, the choice of inducing points in FRCL involves solving a discrete optimisation problem. Instead, we will use memorable past examples that are much cheaper to obtain, and have an intuitive interpretation (we discuss this in Section 4.3). Due to these differences, our method outperforms the method of Titsias et al. (2020), which, unlike ours, performs worse than our Improved VCL results from Chapter 3.

Another more recent work, called Variational Auto-Regressive Gaussian Processes (VAR-GP) (Kapoor et al., 2021), also uses sparse inducing point approximations for continual learning with GPs. They consider distributions over the hyperparameters θ (unlike in FRCL), and learn inducing points instead of using a subset of past data. Their loss function is of a similar form to FRCL's Equation 4.5, except (i) with an additional KL-to-prior term for the hyperparameters, and (ii) with some differences due to an auto-regressive parameterisation of the distribution over inducing points. They show good performance on Split MNIST and Permuted MNIST, however, they may have difficulty scaling up due to computation costs. Future work reducing computation costs in VAR-GP would be an interesting research direction. Instead, in this chapter we take a slightly different approach. We show how we can

make approximations to our initially-expensive method, reducing computation costs when necessary.

There are other ideas for performing function-space inference in Bayesian neural networks, but these are only applied in the batch-setting, and so are not directly applicable to the continual learning setting. For example, fBNNs (Sun et al., 2019) specify GP priors and maximise a variational objective defined on stochastic processes. There is also work discussing how to define variational objectives directly on stochastic processes, arguing that it is not straightforward and that there are some important points that must be considered (Burt et al., 2020).

Our approach: Functional Regularisation of Memorable Past (FROMP)

We take a slightly different approach to the other works discussed so far. We start with the weight-space objective for variational inference (VI) for continual learning (Equation 2.8), but approximate the expectation of the log-prior in weight-space with a term in function-space,

$$\mathcal{L}_t(\boldsymbol{\eta}_t) = \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[-\log p(\mathcal{D}_t|\boldsymbol{w}) + \log q_{\boldsymbol{\eta}_t}(\boldsymbol{w})\right] - \underbrace{\mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[\log q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})\right]}_{\approx \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})\right]}, \tag{4.6}$$

where $\boldsymbol{f}$ is is the vector of function values $f(\boldsymbol{x}_i)$ defined over a set of points, and $\tilde{q}_{w_t}(\boldsymbol{f})$ is the function-space distribution over that set of points induced by the weight-space distribution $q_{\boldsymbol{\eta}_t}(\boldsymbol{w})$.

We only consider approximating the log-prior term in function-space. We could also approximate the entropy term (the $\mathbb{E}_q\left[\log q_{\boldsymbol{\eta}_t}(\boldsymbol{w})\right]$ term) in function-space, meaning we would be approximating the entire KL-to-prior term in function-space. This could be an interesting avenue to explore in the future. We only consider the log-prior term as this is the term which incorporates prior information, and we only need to treat prior information in function-space in order to avoid forgetting.

We note that replacing the term in weight-space with one in function-space is an approximation, and is usually not exact. We analyse this in detail later in this thesis (in Section 5.6 and Appendix D.2), but for now, we note that this is an approximation because $q_{\boldsymbol{\eta}_t}(\boldsymbol{w})$ and $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$ are different distributions, meaning that they induce different mappings to function-space distributions on neural networks.[1]

Overall, there are three steps to our method FROMP, summarised in Figure 4.1, and the following sections in this chapter go over each step in turn:

[1] As we will see in Section 4.2, the mappings to function-space distributions depend on the Jacobian, and the Jacobian depends on the value of the weights in the neural network. Therefore the two mappings are different.

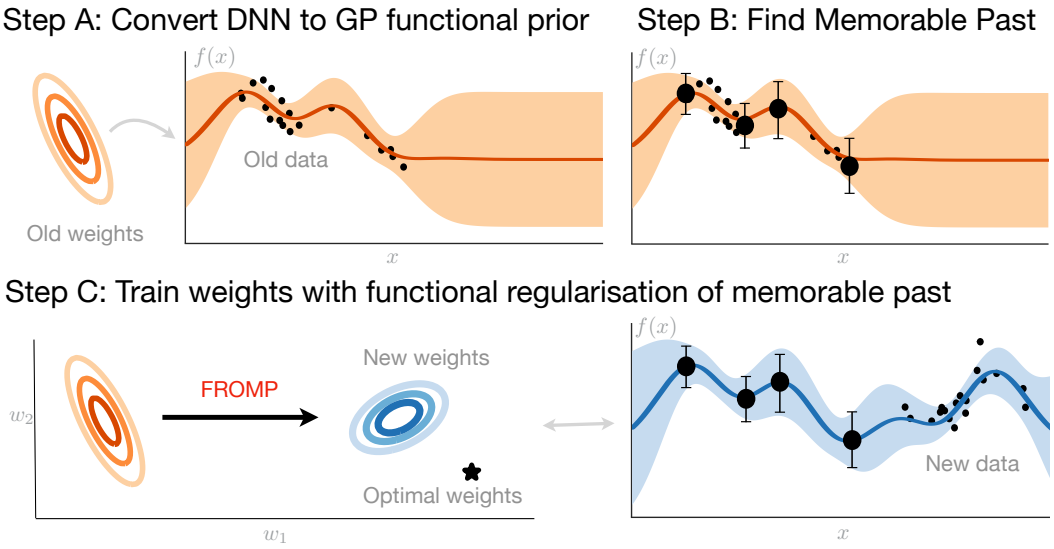


Figure 4.1: There are three steps to our method FROMP. In Step A, we view the deep neural network in function-space, converting a previously-trained model into a Gaussian Process (GP) functional prior for the next task. In Step B, we find a few memorable past examples that will be crucial to avoid forgetting for future tasks (four points in this example). In Step C, when we see new data, we can train the weights of our neural network (left) with functional regularisation of the memorable past examples. We eventually reach a solution that performs well on both the new data and old data (right).

Step A (Section 4.2): we view a distribution over network *weights* as a distribution over the network *parameters*, approximating it with a Gaussian Process (Khan et al., 2019).

Step B (Section 4.3): we choose a few memorable past examples that we will store and regularise on in the future. These are the datapoints that are most important to avoid forgetting information.

Step C (Section 4.4): When we see new data from the next task, we train the weights of our neural network while functionally regularising over the memorable past points that we stored. In this last step, we start from Equation 4.6 and make some approximations to keep computation costs down. We discuss the effects of various approximations and introduce two algorithms based on slightly different approximations.

All steps in FROMP are derived using a single framework, based on natural-gradient variational inference and a linear model view of the update steps. By basing all steps in a single framework, we do not have many separate moving parts that each use different motivation and approximations.

4.2 From deep networks to functional priors

In Step A of FROMP, we convert from a distribution over weights of a neural network to a distribution over functions of the neural network. We assume we have a Gaussian distribution over weights, $q_{\boldsymbol{\eta}}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$, and by the end of this section, we will see how this approximately induces a Gaussian distribution over functions, $\tilde{q}_{\boldsymbol{w}}(\boldsymbol{f}) = \mathcal{N}(\boldsymbol{f}; \boldsymbol{m}, \boldsymbol{K})$, where $\boldsymbol{f}$ is the vector of function values defined over a set of inputs. These distributions over functions are the distributions we will use in FROMP's objective (Equation 4.6), where we need to calculate this mapping twice: once for the functional prior $\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$, and once for the current function $\tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f})$.

We use an approach called DNN2GP from Khan et al. (2019) to convert deep networks to Gaussian Processes (GPs). We now describe this approach, which is also in Appendix B of Khan et al. (2019), but not written out or explained in the detail we require for FROMP. We assume scalar outputs, and the extension to multiple outputs is in Appendix C.2.

The DNN2GP approach is very similar to the standard weight-space to function-space conversion for linear basis-function models (Rasmussen and Williams, 2006). For example, consider a linear regression model on a scalar output $y_i = f_{\boldsymbol{w}}(\boldsymbol{x}_i) + \epsilon_i$ with a function output $f_{\boldsymbol{w}}(\boldsymbol{x}_i) = \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}$ using a feature map $\boldsymbol{\phi}(\boldsymbol{x})$. Assume Gaussian noise $\mathcal{N}(\epsilon_i; 0, \Lambda^{-1})$ and a Gaussian prior $\mathcal{N}(\boldsymbol{w}; 0, \delta^{-1}\mathbf{I})$ where $\mathbf{I}$ is an identity matrix. It can then be shown that the posterior distribution (after seeing dataset $\mathcal{D}$) of this linear model, denoted by $\mathcal{N}(\boldsymbol{w}; \boldsymbol{w}_{\text{lin}}, \boldsymbol{\Sigma}_{\text{lin}})$, induces a GP posterior on function $f_{\boldsymbol{w}}(\boldsymbol{x})$ whose mean and covariance functions are given by (see Appendix C.1 or Chapter 2 in Rasmussen and Williams (2006)),

$$m_{\text{lin}}(\boldsymbol{x}) = f_{\boldsymbol{w}_{\text{lin}}}(\boldsymbol{x}), \quad \kappa_{\text{lin}}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\Sigma}_{\text{lin}}\, \boldsymbol{\phi}(\boldsymbol{x}'), \tag{4.7}$$

where $\boldsymbol{w}_{\text{lin}}$ is simply the Maximum-A-Posteriori (MAP) estimate of the linear model, and $\boldsymbol{\Sigma}_{\text{lin}}^{-1} = \sum_{i\in\mathcal{D}} \boldsymbol{\phi}(\boldsymbol{x}_i)\, \Lambda\, \boldsymbol{\phi}(\boldsymbol{x}_i)^\top + \delta\mathbf{I}$. We can also write the predictive distribution of the observation $y = f(\boldsymbol{x}) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \Lambda^{-1})$ as,

$$p(y|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(y\,;\, \underbrace{f_{\boldsymbol{w}_{\text{lin}}}(\boldsymbol{x})}_{m_{\text{lin}}(\boldsymbol{x})},\ \underbrace{\boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\Sigma}_{\text{lin}} \boldsymbol{\phi}(\boldsymbol{x})}_{\kappa_{\text{lin}}(\boldsymbol{x},\boldsymbol{x})} + \Lambda^{-1}),$$
$$\text{where } \boldsymbol{\Sigma}_{\text{lin}}^{-1} = \sum_{i\in\mathcal{D}} \boldsymbol{\phi}(\boldsymbol{x}_i)\, \Lambda\, \boldsymbol{\phi}(\boldsymbol{x}_i)^\top + \delta\mathbf{I}. \tag{4.8}$$

DNN2GP computes a similar GP posterior but for a *neural network* whose weight-space posterior is approximated by a Gaussian. The key steps are (i) use a Gaussian approximation on the neural network weights $\boldsymbol{w}$ (such as given by the Laplace approximation or during variational inference with a Gaussian approximating family), (ii) write down a linear model

that has the same posterior over weights, (iii) find the equivalent GP predictions using this linear model. We will skip the details in step (ii), instead referring the reader to Khan et al. (2019). We focus on writing down the form of the equivalent GP (step (iii)). When we evaluate this GP at a set of input points, we get our distribution over functions $\tilde{q}_{\boldsymbol{w}}(\boldsymbol{f})$.

We next go over the various approximations and loss forms that are important for FROMP. We start with a Laplace approximation over weights. We then consider a fully variational approximation, where we optimise the variational objective using natural-gradient variational inference algorithms (see Section 2.2.2 or Chapter 3). This leads to a GP posterior at every iteration during training, and this will define both the functional prior $\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$ and the current function $\tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f})$ in Equation 4.6. After making some approximations in Section 4.4, we will also consider using the Laplace approximation for the functional prior $\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$. The linear models we use in this section will also guide how we choose memorable past points in Section 4.3.

GP Posteriors from the minimiser of neural networks

We start with GP posteriors in the case where we use a (variant of the) Laplace approximation to the weights of a neural network. In our Laplace approximation, we obtain the minimiser $\boldsymbol{w}_*$ after optimising the loss $N\bar{\ell}(\boldsymbol{w}) + \frac{1}{2}\delta \boldsymbol{w}^\top \boldsymbol{w}$. We set the mean $\boldsymbol{\mu}_* = \boldsymbol{w}_*$ and covariance,

$$\boldsymbol{\Sigma}_*^{-1} = \sum_{i \in \mathcal{D}} \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)^\top \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i) + \delta \mathbf{I}, \tag{4.9}$$

where $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}, y) = \nabla^2_{ff}\ell(y, h(f))$ is the scalar Hessian of the loss function and $\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}) = \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x})^\top$ is the $1 \times P$ Jacobian, all evaluated at $\boldsymbol{w} = \boldsymbol{w}_*$. Essentially, this variant of the Laplace approximation uses a Generalised Gauss-Newton approximation for the covariance instead of the Hessian (Schraudolph, 2002; Graves, 2011; Martens, 2020).

Comparing $\boldsymbol{\Sigma}_{\text{lin}}$ in Equation 4.8 with Equation 4.9, we can interpret $\boldsymbol{\Sigma}_*$ as the covariance of a linear model with a feature map $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})^\top$ and noise precision $\Lambda = \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}, y)$.

A regression loss: For a regression loss function $\ell(y, h(f)) = \frac{1}{2}\Lambda(y - f)^2$, we get the following expression for the predictive distribution for the observations y (see Equation 44, Appendix B.2 in Khan et al. (2019)):

$$\hat{p}(y|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(y\,; f_{\boldsymbol{w}_*}(\boldsymbol{x}),\ \ \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\boldsymbol{\Sigma}_*\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})^\top + \Lambda^{-1}),$$
$$\text{where } \boldsymbol{\Sigma}_*^{-1} = \sum_{i \in \mathcal{D}} \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)^\top \Lambda\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i) + \delta \mathbf{I}. \tag{4.10}$$

We use $\hat{p}(y|\boldsymbol{x}, \mathcal{D})$ since this predictive distribution is not exact and is obtained using a type of Laplace approximation. Comparing this to Equation 4.8, we can write the mean and covariance functions in a similar fashion as Equation 4.7,

$$m_{\boldsymbol{w}_*}(\boldsymbol{x}) = f_{\boldsymbol{w}_*}(\boldsymbol{x}), \quad \kappa_{\boldsymbol{w}_*}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{\Sigma}_*\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}')^\top. \tag{4.11}$$

A binary classification loss: A similar expression is available for binary classification with $y \in \{0, 1\}$, considering the loss $\ell(y, h(f)) = -y \log \sigma(f) - (1-y)\log(1-\sigma(f)) = -yf + \log(1+e^f)$ where $\sigma(f) = 1/(1+e^{-f})$ is the sigmoid function. The predictive distribution is given as,

$$\begin{aligned}\hat{p}(y|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(y\,; \sigma(f_{\boldsymbol{w}_*}(\boldsymbol{x})), \;\; & \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{\Sigma}_*\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})^\top \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}) + \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x})), \\ & \text{where } \boldsymbol{\Sigma}_*^{-1} = \sum_{i \in \mathcal{D}} \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)^\top\, \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i)\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i) + \delta \mathbf{I}.\end{aligned} \tag{4.12}$$

where $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}) = \sigma\left(f_{\boldsymbol{w}_*}(\boldsymbol{x})\right)\left[1 - \sigma\left(f_{\boldsymbol{w}_*}(\boldsymbol{x})\right)\right]$. The predictive distribution does not respect the fact that y is binary and treats it like a Gaussian, making it comparable to Equation 4.8. Comparing the two, we can conclude that Equation 4.12 corresponds to the predictive posterior distribution of a GP regression model with $y = f(\boldsymbol{x}) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}))$ with mean and covariance functions as shown below,

$$m_{\boldsymbol{w}_*}(\boldsymbol{x}) = \sigma(f_{\boldsymbol{w}_*}(\boldsymbol{x})), \quad \kappa_{\boldsymbol{w}_*}(\boldsymbol{x}, \boldsymbol{x}') = \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{\Sigma}_*\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}')^\top \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}'). \tag{4.13}$$

Two differences from the regression case is that the mean function is passed through the sigmoid function, and the covariance function has $\Lambda_{\boldsymbol{w}_*}$ multiplied on the both sides. These changes appear because of the non-linearity in the loss function introduced due to the sigmoid function. The multiclass classification loss can also be written in a similar form, and is in Appendix C.2.

Later in Section 4.4, we will make approximations that result in the functional prior in FROMP ($\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$ in Equation 4.6) being given by Equation 4.13. However, when we do not make these approximations, we instead use the GP posterior obtained during iterations of our algorithm during optimisation. We derive this next, and as we will see, the expressions will be very similar to Equation 4.13.

GP Posterior from the iterations of a neural network optimisers

The results so far hold only at a minimiser $\boldsymbol{w}_*$. Khan et al. (2019) also generalise this to iterations of optimisers. They did this for a (natural-gradient) variational inference algorithm and also for its deterministic versions. We will see that the resulting equations are very similar to the Laplace case, but defined at the current $\boldsymbol{w}$ as it is being optimised, instead of just at the minimiser $\boldsymbol{w}_*$.

Given a Gaussian variational approximation $q_j(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$ at iteration j, we can use natural-gradient variational inference (NGVI) algorithms such as Variational Online Newton (VON) or Variational Online Gauss-Newton (VOGN) to optimise for $\{\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\}$. The update steps for these algorithms are given in Section 2.2.2 (for example, VOGN's update steps are in Equations 2.25 and 2.26). Given $q_j(\boldsymbol{w})$, these algorithms proceed by first sampling $\boldsymbol{w}_j \sim q_j(\boldsymbol{w})$, and then updating the variational distribution. Surprisingly, the procedure used to derive a GP predictive distribution for the Laplace approximation generalises to this update too. An expression for the predictive distribution is,

$$\hat{p}_{j+1}(y|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(y\,;\, \sigma(f_{\boldsymbol{w}_j}(\boldsymbol{x})),\;\; \Lambda_{\boldsymbol{w}_j}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_j}(\boldsymbol{x})\, \boldsymbol{\Sigma}_j\, \boldsymbol{J}_{\boldsymbol{w}_j}(\boldsymbol{x})^\top \Lambda_{\boldsymbol{w}_j}(\boldsymbol{x}) + \Lambda_{\boldsymbol{w}_j}(\boldsymbol{x})^{-1}), \tag{4.14}$$

where $\boldsymbol{\Sigma}_{j+1}$ and $\boldsymbol{\mu}_{j+1}$ are updated according to an NGVI algorithm's update equations. Comparing to Equation 4.8, we can write down the mean and covariance functions,

$$m_{\boldsymbol{w}_j}(\boldsymbol{x}) = \sigma(f_{\boldsymbol{w}_j}(\boldsymbol{x})), \quad \kappa_{\boldsymbol{w}_j}(\boldsymbol{x}, \boldsymbol{x}') = \Lambda_{\boldsymbol{w}_j}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_j}(\boldsymbol{x})\, \boldsymbol{\Sigma}_j\, \boldsymbol{J}_{\boldsymbol{w}_j}(\boldsymbol{x}')^\top \Lambda_{\boldsymbol{w}_j}(\boldsymbol{x}'). \tag{4.15}$$

The predictive distribution takes the same form as in the Laplace case, but now the covariance and mean are updated according to the NGVI updates. If desired, we can average over many samples $\boldsymbol{w}_j$. We will use Equation 4.15 as our current function $\tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f})$ in Equation 4.6. We can also use this for our functional prior $\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$, where $\boldsymbol{w}_{t-1}$ is sampled from the distribution obtained after training on the previous task.

However, using NGVI updates might lead to additional variance during training, making optimisation difficult. We may not need all the benefits of NGVI during training, and we can use cheaper algorithms too. For example, we can derive similar GP posteriors when using a deterministic of VOGN, called Online Gauss-Newton (OGN), which was also previously introduced in Section 3.2. In OGN, we do not sample $\boldsymbol{w}_j \sim q_j(\boldsymbol{w})$, and instead use $\boldsymbol{w}_j = \boldsymbol{\mu}_j$. The variational approximation is still defined as $q_j(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{w}_j, \boldsymbol{\Sigma}_j)$. The form of the predictive distribution remains the same as Equation 4.14.

We next describe how we choose our memorable past points to calculate these function-space distributions over.

4.3 Identifying memorable past

In the previous section, we discussed how to go from a distribution over weights of a neural network to a distribution over functions. We want to evaluate this distribution over functions at a set of input examples. In this section, we identify this set of memorable past examples (this is Step B in Figure 4.1). These will be the datapoints that are the most important to avoid forgetting information. By only using a few examples, we reduce the computation cost of functional regularisation.

We exploit a property of linear models to identify our memorable past examples. Consider a linear model where different noise precision Λ_i is assigned to each pair $\{\boldsymbol{x}_i, y_i\}$. For MAP estimation, the examples with high value of Λ_i contribute more, as is clear from the objective,

$$\boldsymbol{w}_{\text{MAP}} = \arg\min_{\boldsymbol{w}} \ \tfrac{1}{2} \sum_{i=1}^{N} \Lambda_i (y_i - \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w})^2 + \tfrac{1}{2} \delta \, \boldsymbol{w}^\top \boldsymbol{w}. \tag{4.16}$$

The noise precision Λ_i can therefore be interpreted as the relevance of the data example i. Such relevant examples are crucial to ensure that the solution stays at $\boldsymbol{w}_{\text{MAP}}$ or close to it. These ideas are widely used in the theory of leverage-score sampling (Alaoui and Mahoney, 2015; Ma et al., 2015) to identify the most *influential* examples. However, computation using such methods is difficult since they require inverting a large matrix, and cheaper solutions may be more desirable. Titsias et al. (2020) use an approximation by inverting smaller matrices, but they require solving a discrete optimisation problem to select examples. We propose a method which is not only cheap and effective, but also yields intuitive results. We also compare against the leverage-score sampling method.

We use the same key idea as in Section 4.2, using the linear model from which we obtained our GP posterior. In this way, we use the same framework from Step A of FROMP (converting to function-space) to guide our choice of memorable past points. The linear mode assigns different noise precision to each data example. This is clear when we compare Equation 4.8 with Equation 4.9, where we see the quantity $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)$ playing the same role as the noise precision Λ. Additionally, the linear model in Equation 4.9 uses Jacobians $\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)$ as features (instead of $\boldsymbol{\phi}_i$).

Therefore, $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)$ can be used as a relevance measure, and a simple approach to pick influential examples is to sort $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)$ over all examples, and pick the top few examples. We refer to this method as the Lambda method for choosing examples.

Alternatively, we can use techniques from the leverage-score sampling literature (Alaoui and Mahoney, 2015; Ma et al., 2015), but with $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)$ instead of Λ. Specifically, we

assign a score to each datapoint given by,

$$\text{Leverage score for datapoint } i = \left(\mathbf{\Phi}\mathbf{\Phi}^\top \left[\mathbf{\Phi}\mathbf{\Phi}^\top + \mathbf{\Lambda}^{-1}\right]^{-1}\right)_{ii}, \tag{4.17}$$

where $\mathbf{\Phi}$ is an $N \times P$ matrix with features $\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)$ as rows (for all N datapoints $\boldsymbol{x}_i$), $\mathbf{\Lambda}$ is an $N \times N$ diagonal matrix with $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)$ as the i'th diagonal element, and $\boldsymbol{A}_{ii}$ denotes taking the ii'th element of the matrix $\boldsymbol{A}$. The leverage score is difficult to calculate as it requires inverting a large $NK \times NK$ matrix, where N is the number of datapoints and K is the number of classes. In reality, we can make computation cheaper by approximating this matrix with a block-diagonal one with K lots of $N \times N$ matrices, or even with K lots of $N_k \times N_k$ matrices, where N_k is the number of datapoints in each class $k \in \{1, 2, ..., K\}$ (if there are an equal number of points per class, then $N_k = N/K \ \ \forall k$).

Having obtained a leverage score for each point, we then sample the desired number of points with probability proportional to their leverage score, and store this set of points. We call this method the Leverage method for choosing examples. In Section 5.4, we will see a slightly different motivation for the Leverage method (and also for the Lambda method), based on theory we introduce in Chapter 5.

Due to the computation cost of the Leverage method, we normally use the significantly cheaper Lambda method. The Lambda method only requires a forward pass to get $\ell(y_i, h(f_{\boldsymbol{w}_*}(\boldsymbol{x}_i)))$, followed by double differentiation. For example, in binary classification, this is equal to $\sigma(f_{\boldsymbol{w}_*}(\boldsymbol{x}_i))\left[1 - \sigma(f_{\boldsymbol{w}_*}(\boldsymbol{x}_i))\right]$. For binary classification, the Lambda method is equivalent to the "Confidence Sampling" approaches used in the Active Learning literature (Wang and Shang, 2014; Ash et al., 2020), although in general it differs from them.

Regardless of which method we use, we refer to the set of chosen examples as the *memorable past* examples. After training on task t, we select a set of few memorable examples from $\mathcal{D}_t$, and we denote this memorable past set as $\mathcal{M}_t$.

An example of points chosen by the Lambda method is shown in Figure 4.2, where we pick many examples that are difficult to classify. The Lambda method can be intuitively thought of as choosing examples close to the decision boundary. On the other hand, the Leverage method samples points that are spread out in input-space, with a strong bias towards points that are close to the decision boundary.

We next describe how we functionally regularise over the memorable past examples.

(a) Most (left) vs least (right) memorable, sorted by the Lambda method on MNIST.

(b) Most (left) vs least (right) memorable, sorted by the Lambda method on CIFAR-10.

Figure 4.2: We plot the five most memorable and five least memorable datapoints per class by the Lambda method on (a) MNIST and (b) CIFAR-10. To obtain these, we train a model on the dataset, and sort all datapoints by their noise precision $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i, y_i)$, as described in Section 4.3. The memorable points are the more atypical examples from each class, and more difficult to classify as they may be closer to the decision boundaries.

4.4 Training in weight-space with a functional prior

We will now describe the final step for training in weight-space with regularisation in function-space (Step C in Figure 4.1). As discussed in Section 4.1, our objective function is the variational objective in weight-space, except with the expectation of the log-prior in function-space instead of weight-space,

$$\mathcal{L}_t^{\text{FROMP}}(\boldsymbol{\eta}_t) = \mathbb{E}_{q_{\boldsymbol{\eta}_t}(\boldsymbol{w})}\left[-(1/\tau)\,\log p(\mathcal{D}_t|\boldsymbol{w}) + \log q_{\boldsymbol{\eta}_t}(\boldsymbol{w})\right] - \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})\right], \tag{4.18}$$

where we have highlighted the function-space term in red, and we have introduced a tempering parameter $\tau > 0$ like in Equation 3.2.

We now are armed with all the mathematical tools we need to evaluate this objective and optimise it. The first two terms in Equation 4.18 (the likelihood term and the entropy term) are unchanged from the weight-space variational objective function, and calculating them is unchanged from earlier (we can use the same methods as in Section 2.2). The new function-space term consists of two distributions over $\boldsymbol{f}$, $\tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f})$ and $\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$, where

$\boldsymbol{f}$ is the vector of function values defined over a set of points. In Section 4.2 we derived expressions for these distributions. These distributions are functions of the parameters $\boldsymbol{\eta}_t$ and $\boldsymbol{\eta}_{t-1}$ respectively.

We could directly optimise Equation 4.18 for the parameters $\boldsymbol{\eta}_t$ using standard-gradient techniques, however, we expect natural-gradient update steps to converge quicker. This was motivated in Section 2.2.2, and we saw that this led to improved convergence in Chapter 3. A natural-gradient update (specifically, the VOGGN update[2] (Khan et al., 2019)) to optimise Equation 4.18 takes the following form,

$$\boldsymbol{\Sigma}^{-1} \leftarrow (1-\beta)\boldsymbol{\Sigma}^{-1} + \beta \left[\frac{1}{\tau} \sum_{i \in \mathcal{D}_t} \boldsymbol{J}_{\boldsymbol{w}_t}(\boldsymbol{x}_i)^\top \Lambda_{\boldsymbol{w}_t}(\boldsymbol{x}_i)\, \boldsymbol{J}_{\boldsymbol{w}_t}(\boldsymbol{x}_i) - 2\nabla_{\boldsymbol{\Sigma}}\, \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})} \left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f}) \right] \right], \tag{4.19}$$

$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \beta\, \boldsymbol{\Sigma} \left[\frac{1}{\tau} \sum_{i \in \mathcal{D}_t} \mathbf{g}_i(\boldsymbol{w}_t) - \nabla_{\boldsymbol{\mu}}\, \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})} \left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f}) \right] \right], \tag{4.20}$$

where we have used the per-example gradients of the negative log-likelihood $\mathbf{g}_i(\boldsymbol{w}_t)$ at a sample from our current approximate posterior $\boldsymbol{w}_t \sim q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$, and we have ignored the iteration subscript to simplify notation.

Using the $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ obtained with this iteration, we can use the reasoning in Section 4.2 to define the following GP predictive posterior at a sample $\boldsymbol{w}_t$, with a mean and covariance function as follows (see Equation 4.15),

$$m_{\boldsymbol{w}_t}(\boldsymbol{x}) = \sigma(f_{\boldsymbol{w}_t}(\boldsymbol{x})), \quad \kappa_{\boldsymbol{w}_t}(\boldsymbol{x}, \boldsymbol{x}') = \Lambda_{\boldsymbol{w}_t}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_t}(\boldsymbol{x})\, \boldsymbol{\Sigma}\, \boldsymbol{J}_{\boldsymbol{w}_t}(\boldsymbol{x}')^\top \Lambda_{\boldsymbol{w}_t}(\boldsymbol{x}). \tag{4.21}$$

We use these equations for $\tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f}) = \mathcal{N}(\boldsymbol{f}; \boldsymbol{m}_t(\boldsymbol{w}), \boldsymbol{K}_t(\boldsymbol{w}))$, where $\boldsymbol{m}_t(\boldsymbol{w})$ and $\boldsymbol{K}_t(\boldsymbol{w})$ respectively denote the mean vector and kernel matrix obtained by evaluating Equation 4.21 at the memorable past examples (all examples in $\mathcal{M}_s$ for all tasks $s < t$). The mean vector $\boldsymbol{m}_t$ is obtained by concatenating $m_{\boldsymbol{w}_t}(\boldsymbol{x})$ for all memorable past examples, and the covariance matrix $\boldsymbol{K}_t$ is defined as the matrix with ij'th entry as $\kappa_{\boldsymbol{w}_t}(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

We can use the same reasoning to obtain corresponding mean $\boldsymbol{m}_{t-1}$ and covariance $\boldsymbol{K}_{t-1}$ from $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$. This gives $\tilde{q}_{w_{t-1}}(\boldsymbol{f}) = \mathcal{N}(\boldsymbol{f}; \boldsymbol{m}_{t-1}, \boldsymbol{K}_{t-1})$, where the mean vector and covariance matrix are obtained in exactly the same way as earlier, except evaluated at a sample $\boldsymbol{w}_{t-1} \sim q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$ instead of a sample from $q_{\boldsymbol{\eta}_t}(\boldsymbol{w})$. Note that we can take many samples and average over them. The evaluations are over the same memorable past

[2]The VOGGN update uses the standard Generalised Gauss-Newton approximation to the Hessian (Martens, 2020; Kunstner et al., 2019), instead of the slightly different parameterisation used in VOGN in Section 2.2.2.

examples. Alternatively, if we are using a Laplace approximation for $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$, we can use Equation 4.13.

Given these quantities, the functional regularisation term has an analytical expression,

$$\mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})\right] = -\tfrac{1}{2}\left[\mathrm{Tr}(\boldsymbol{K}_{t-1}^{-1}\boldsymbol{K}_t) + (\boldsymbol{m}_t - \boldsymbol{m}_{t-1})^\top \boldsymbol{K}_{t-1}^{-1}(\boldsymbol{m}_t - \boldsymbol{m}_{t-1})\right] + \text{const.} \tag{4.22}$$

Our goal is to obtain the derivative of this term with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Both $\boldsymbol{m}_t$ and $\boldsymbol{K}_t$ are functions of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ through the sample $\boldsymbol{w}_t = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$. Therefore, we can compute these derivatives using the chain rule. We call this algorithm variational-FROMP (or var-FROMP).

We note that if we had decided to compute the entire KL-to-prior term in function-space, then we would also have to compute the entropy in function-space $\mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f})\right]$, and this would simply add a $-\frac{1}{2}\log|\boldsymbol{K}_t|$ term to Equation 4.22. This would also affect the VOGGN updates in Equations 4.19 and 4.20 (we can follow Section 2.2.2 to derive the new updates).

Unfortunately, var-FROMP is costly for large problems, as it requires Monte-Carlo sampling (for conversion to function-space distributions), large matrix inversions (such as of $\boldsymbol{K}_{t-1}$), as well as higher-order derivatives (the Jacobians in Equation 4.21 depend on the current variational parameters, and so we need to take their derivative with respect to the variational parameters). We therefore now propose approximations to reduce computation cost. We will look at two algorithms. OGN-FROMP will make a few approximations to reduce computation cost, but FROMP will make even more approximations, allowing it to scale easily.

4.4.1 OGN-FROMP

In this subsection, we consider an algorithm that maintains a covariance matrix during optimisation by using the Online Gauss Newton (OGN) algorithm (which we have also previously introduced in Sections 3.2 and 4.2). We make four approximations to var-FROMP to derive an algorithm that we call OGN-FROMP:

Approximation 1: Instead of sampling $\boldsymbol{w}_{t-1}$, we set $\boldsymbol{w}_{t-1} = \boldsymbol{\mu}_{t-1}$, which is the mean of the posterior approximation $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$ until task $t-1$. Therefore, we replace $\mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})\right]$ by $\mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{\boldsymbol{\mu}_{t-1}}(\boldsymbol{f})\right]$. This affects $\boldsymbol{m}_{t-1}$ and $\boldsymbol{K}_{t-1}$ in Equation 4.22, and is similar to the Laplace approximation in Section 4.2.

Approximation 2: We reduce the cost of calculating derivatives of the functional regulariser, avoiding calculating complex derivatives (such as derivatives of Jacobians). For the derivative with respect to $\boldsymbol{\mu}$, we ignore the derivative with respect to $\boldsymbol{K}_t$ and only consider $\boldsymbol{m}_t$. For the derivative with respect to $\boldsymbol{\Sigma}$, we leave the log-prior term in weight-space, and do not use the function-space approximation. This minimises the differences between our function-space algorithm and the weight-space algorithms that we know converge quickly.[3] Future work could consider not making this approximation.

Therefore, the derivative in the update in Equation 4.20 can be approximated as,

$$\nabla_{\boldsymbol{\mu}} \mathbb{E}_{\tilde{q}_{\boldsymbol{w}_t}(\boldsymbol{f})} \left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})\right] \approx -\left[\nabla_{\boldsymbol{\mu}} \boldsymbol{m}_t\right] \boldsymbol{K}_{t-1}^{-1}(\boldsymbol{m}_t - \boldsymbol{m}_{t-1}). \tag{4.23}$$

Approximation 3: Instead of using the full kernel matrix $\boldsymbol{K}_{t-1}$, we factorise it across tasks. We use a block-diagonal matrix containing the kernel matrix $\boldsymbol{K}_{t-1,s}$ for all past tasks s as the diagonal. This makes the cost of inversion linear (instead of cubic) in the number of past tasks.

Approximation 4: We use a deterministic version of the NGVI update. We set $\boldsymbol{w}_t = \boldsymbol{\mu}$, which corresponds to setting the random noise ϵ to zero in $\boldsymbol{w}_t = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\boldsymbol{\epsilon}$, and we use a diagonal covariance $\boldsymbol{\Sigma}$ instead of full-covariance. The gradient of $\boldsymbol{m}_t$ with respect to $\boldsymbol{\mu}$ is given as follows using the chain rule (here $\boldsymbol{m}_{t,s}$ is the sub-vector of $\boldsymbol{m}_t$ corresponding to the task s),

$$\nabla_{\boldsymbol{\mu}} \boldsymbol{m}_{t,s}[i] = \nabla_{\boldsymbol{\mu}} \left[\sigma\left(f_{\boldsymbol{\mu}}(\boldsymbol{x}_i)\right)\right] = \Lambda_{\boldsymbol{\mu}}(\boldsymbol{x}_i)\, \boldsymbol{J}_{\boldsymbol{\mu}}(\boldsymbol{x}_i)^\top, \tag{4.24}$$

where $\boldsymbol{x}_i \in \mathcal{M}_s$, and where the second equality holds for canonical link functions. We can also let automatic differentiation handle this computation.

With these approximations, we can write the OGN-FROMP update as follows,[4]

$$\boldsymbol{\Sigma}^{-1} \leftarrow (1-\beta)\boldsymbol{\Sigma}^{-1} + \beta \left[\sum_{i \in \mathcal{D}_t} \operatorname{diag}\left(\boldsymbol{J}_{\boldsymbol{\mu}}(\boldsymbol{x}_i)^\top \boldsymbol{J}_{\boldsymbol{\mu}}(\boldsymbol{x}_i)\right) + \boldsymbol{\Sigma}_{t-1}^{-1}\right], \tag{4.25}$$

$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \beta\boldsymbol{\Sigma}\left[\frac{1}{\tau}\sum_{i \in \mathcal{D}_t} \mathbf{g}_i(\boldsymbol{\mu}) + \sum_{s=1}^{t-1} \left[\nabla_{\boldsymbol{\mu}} \boldsymbol{m}_{t,s}\right] \boldsymbol{K}_{t-1,s}^{-1}(\boldsymbol{m}_{t,s} - \boldsymbol{m}_{t-1,s})\right], \tag{4.26}$$

[3] As the $\boldsymbol{\Sigma}$ update is now only in weight-space, we do not need a tempering factor τ in its update, but we keep τ in the $\boldsymbol{\mu}$ update. This is especially important when we use a small number of datapoints, and so need to upweight the function-space term.

[4] Note that we are using the OGN update, which uses a slightly different parameterisation of the Generalised Gauss-Newton matrix than in the earlier (V)OGGN update in Equation 4.19.

where $\text{diag}(\boldsymbol{A})$ is the diagonal of matrix $\boldsymbol{A}$.

Overall, OGN-FROMP runs the updates Equations 4.25 and 4.26 on a task (along with minibatching of the current dataset $\mathcal{D}_t$). Then, we use the converged approximate posterior $q_{\boldsymbol{\eta}_t}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ to find a new set of memorable points. When we see a new task, we set the new prior to be the previous posterior, and convert to function-space $\tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f})$.

We will see results for OGN-FROMP in Section 4.5.2. We will see that it performs well on a smaller-scale benchmark, but further work is required to assess its suitability for larger-scale benchmarks. Instead, we can make further approximations, and still get an algorithm that works very well on both small and large-scale benchmarks. We describe these further approximations next.

4.4.2 FROMP

We now describe further approximations that we can make on top of OGN-FROMP in order to get a different algorithm, which we call FROMP. FROMP performs very well in practice, despite making many approximations to var-FROMP. FROMP uses Approximations 1, 3 and 4 from OGN-FROMP. Approximation 2 changes slightly for FROMP, and we introduce another approximation, Approximation 5.

Approximation 2: We ignore the derivative with respect to $\boldsymbol{K}_t$ and only consider $\boldsymbol{m}_t$. For OGN-FROMP, we only did this for the derivative with respect to $\boldsymbol{\mu}$, but now we also do this for the derivative with respect to $\boldsymbol{\Sigma}$. This leads to the following (on top of Equation 4.23),

$$\nabla_{\boldsymbol{\Sigma}} \, \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})} \left[\log \tilde{q}_{\boldsymbol{w}_{t-1}}(\boldsymbol{f}) \right] \approx - \left[\nabla_{\boldsymbol{\Sigma}} \boldsymbol{m}_t \right] \boldsymbol{K}_{t-1}^{-1} (\boldsymbol{m}_t - \boldsymbol{m}_{t-1}). \tag{4.27}$$

Note that when combined with Approximation 4, we effectively ignore the term $\nabla_{\boldsymbol{\Sigma}} \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})} \left[\log \tilde{q}_{w_{t-1}}(\boldsymbol{f}) \right]$, as the gradient with respect to $\boldsymbol{\Sigma}$ is always $\boldsymbol{0}$.

Approximation 5: Our final approximation is to replace the natural-gradient update by an RMSprop-like update where we denote $\boldsymbol{\mu}$ by $\boldsymbol{w}$, giving updates,

$$\boldsymbol{s} \leftarrow (1-\beta)\boldsymbol{s} + \beta \left[\sum_{i \in \mathcal{D}_t} \text{diag} \left(\boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_i)^\top \Lambda_{\boldsymbol{w}}(\boldsymbol{x}_i) \, \boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_i) \right) \right], \tag{4.28}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \beta \frac{1}{\boldsymbol{s} + \delta \boldsymbol{1}} \left[\frac{1}{\tau} \sum_{i \in \mathcal{D}_t} \mathbf{g}_i(\boldsymbol{w}_t) + \sum_{s=1}^{t-1} \left[\nabla_{\boldsymbol{w}} \boldsymbol{m}_{t,s} \right] \boldsymbol{K}_{t-1,s}^{-1} \left(\boldsymbol{m}_{t,s} - \boldsymbol{m}_{t-1,s} \right) \right], \tag{4.29}$$

where we have added a regulariser δ to $\boldsymbol{s}$ in the second line to avoid dividing by zero. Previously, this regulariser was the prior precision. Ideally, when using a functional prior, we would replace this by another term that calculates covariance information in function-space. However, this term was ignored by making Approximations 2 and 4, and we use δ instead. The final Gaussian approximation is obtained with the mean equal to $\boldsymbol{w}$ and covariance equal to a diagonal matrix with $1/(\boldsymbol{s} + \delta\mathbf{1})$ as its diagonal.

Optimising Equations 4.28 and 4.29 gives the same solutions for $\boldsymbol{w}$ as the objective,

$$\min_{w} \quad N\bar{\ell}_t(\boldsymbol{w}) + \tfrac{1}{2}\tau \sum_{s=1}^{t-1} (\boldsymbol{m}_{t,s}(\boldsymbol{w}) - \boldsymbol{m}_{t-1,s})^\top \boldsymbol{K}_{t-1,s}^{-1} (\boldsymbol{m}_{t,s}(\boldsymbol{w}) - \boldsymbol{m}_{t-1,s}). \tag{4.30}$$

The above is a computationally-cheap approximation of the objective in Equation 4.18. We see that the function-space regulariser takes the form of a quadratic regulariser. It has two nice properties: (i) it forces the output of the current network, $\boldsymbol{m}_t$, to be close to the output of the previous network, $\boldsymbol{m}_{t-1}$ (over all memorable points), and (ii) it has a kernel $\boldsymbol{K}_{t-1}^{-1}$ that automatically weighs the stored examples (more important examples get a higher weight, less important examples get a lower weight, and a pair of examples that are in similar regions of input-space have a large off-diagonal term that reduces their contribution). This function-regulariser is able to exploit correlations between memorable examples due to a full kernel matrix $\boldsymbol{K}_{t-1}^{-1}$. Property (i) above is a key difference to Titsias et al. (2020), and property (ii) is an improvement to Equation 4.4 (Benjamin et al., 2019).

We follow the recommendations of Khan et al. (2018) and use Adam to optimise the objective in Equation 4.30. This means that the estimate of the covariance is not accurate, although the fixed-points of the objective are not changed. We correct the approximation after convergence of the algorithm by recomputing the diagonal of the covariance according to Equation 4.28, like in a Laplace approximation.

Our approach therefore provides a cheap weight-space training method while exploiting correlations in function-space. We can expect further improvements by relaxing our approximations. For example, we can use a full kernel matrix or employ a block-diagonal weight-covariance matrix. We leave such comparisons as future work since they require sophisticated implementation to scale, although we provide a single experiment with OGN-FROMP in Section 4.5.2 to show the potential of relaxing approximations. We also note that we could treat the entire KL-to-prior term in function-space, instead of just the log-prior term (see Equation 4.6), without affecting the final FROMP objective in Equation 4.30. This is because of the approximations we made.

Algorithm 2 FROMP for binary classification on task t given memorable past sets $\mathcal{M}_{1:t-1}$ and $q_{t-1}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{\mu}_{t-1}, \text{diag}(\boldsymbol{v}_{t-1}))$. Additional computations on top of Adam are highlighted in red. In this algorithm, we calculate the memorable past using the Lambda method, but we could use other methods such as the Leverage method (see Section 4.3).

function FROMP ($\mathcal{D}_t, \boldsymbol{\mu}_{t-1}, \boldsymbol{v}_{t-1}, \mathcal{M}_{1:t-1}$)
  Compute $\boldsymbol{m}_{t-1,s}, \boldsymbol{K}^{-1}_{t-1,s}, \forall$ tasks $s < t$ (Equations 4.32 and 4.33).
  Initialise $\boldsymbol{w} \leftarrow \boldsymbol{\mu}_{t-1}$.
  repeat
    Sample a minibatch $\mathcal{B}$ of size B.
    $\hat{\mathbf{g}} \leftarrow (1/B) \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{w}} \ell(y_i, \sigma(f_{\boldsymbol{w}}(\boldsymbol{x}_i)))$.
    $\mathbf{g}_f \leftarrow$ g_funcreg $(\boldsymbol{w}, \boldsymbol{m}_{t-1}, \boldsymbol{K}^{-1}_{t-1}, \mathcal{M}_{1:t-1})$.
    $\boldsymbol{w} \leftarrow$ Adam update with gradient $N\hat{\mathbf{g}} + \tau \mathbf{g}_f$.
  until converged
  Set $\boldsymbol{\mu}_t \leftarrow \boldsymbol{w}$.
  Compute $\boldsymbol{v}_t$ (Equation 4.31).
  $\mathcal{M}_t \leftarrow$ memorable_past_Lambda$(\mathcal{D}_t, \boldsymbol{w})$.
  return $\boldsymbol{\mu}_t, \boldsymbol{v}_t, \mathcal{M}_t$

function g_funcreg $(\boldsymbol{w}, \boldsymbol{m}_{t-1}, \boldsymbol{K}^{-1}_{t-1}, \mathcal{M}_{1:t-1})$
  Initialise $\mathbf{g}_f \leftarrow \mathbf{0}$.
  for task $s = 1, 2, ..., t-1$ **do**
    Compute $\boldsymbol{m}_{t,s}$ (Equation 4.32).
    $\boldsymbol{h}_i \leftarrow \Lambda_{\boldsymbol{w}}(\boldsymbol{x}_i)\, \boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_i)^\top, \forall \boldsymbol{x}_i \in \mathcal{M}_s$, and form matrix $\mathbf{H}$ with $\boldsymbol{h}_i$ as columns.
    $\mathbf{g}_f \leftarrow \mathbf{g}_f + \mathbf{H}\boldsymbol{K}^{-1}_{t-1,s}(\boldsymbol{m}_{t,s} - \boldsymbol{m}_{t-1,s})$.
  end for
  return $\mathbf{g}_f$

function memorable_past_Lambda $(\mathcal{D}_t, \boldsymbol{w})$
  Calculate $\Lambda_{\boldsymbol{w}}(\boldsymbol{x}_i), \ \forall \boldsymbol{x}_i \in \mathcal{D}_t$.
  return M examples with highest $\Lambda_{\boldsymbol{w}}(\boldsymbol{x}_i)$.

The final FROMP algorithm and computational complexity

The resulting algorithm, FROMP, is shown in Algorithm 2 for binary classification (the extension to multiclass classification is in Appendix C.2). For binary classification, we assume a sigmoid $\sigma(f_{\boldsymbol{w}}(\boldsymbol{x}))$ function and cross-entropy loss. As shown in Section 4.2, the Jacobian (of size $1 \times P$) and noise precision (a scalar) are as follows: $\boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}) = \nabla_{\boldsymbol{w}} f_w(\boldsymbol{x})^\top$ and $\Lambda_{\boldsymbol{w}}(\boldsymbol{x}) = \sigma\left(f_{\boldsymbol{w}}(\boldsymbol{x})\right)\left[1 - \sigma\left(f_{\boldsymbol{w}}(\boldsymbol{x})\right)\right]$. To compute the mean and kernel, we need the diagonal of the covariance, which we denote by $\boldsymbol{v}_t$. This can be obtained using Equation 4.9

but with the sum over $\mathcal{D}_{1:t}$. The update below computes this recursively,

$$\frac{\mathbf{1}}{\boldsymbol{v}_t} = \left[\frac{\mathbf{1}}{\boldsymbol{v}_{t-1}} + \sum_{i\in\mathcal{D}_t} \text{diag}\left(\boldsymbol{J}_w(\boldsymbol{x}_i)^\top \Lambda_w(\boldsymbol{x}_i)\, \boldsymbol{J}_w(\boldsymbol{x}_i)\right)\right], \tag{4.31}$$

where '/' denotes element-wise division and $\text{diag}(\boldsymbol{A})$ is the diagonal of $\boldsymbol{A}$. We can compute the mean and kernel matrix as follows (see Section 4.2 for details),

$$\boldsymbol{m}_{t,s}(\boldsymbol{w})[i] = \sigma\left(f_{\boldsymbol{w}}(\boldsymbol{x}_i)\right), \tag{4.32}$$

$$\boldsymbol{K}_{t,s}(\boldsymbol{w})[i,j] = \Lambda_{\boldsymbol{w}}(\boldsymbol{x}_i)\left[\boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_i)\,\text{Diag}\left(\boldsymbol{v}_t\right)\boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_j)^\top\right]\Lambda_{\boldsymbol{w}}(\boldsymbol{x}_j), \tag{4.33}$$

over all memorable examples $\boldsymbol{x}_i, \boldsymbol{x}_j$, where $\text{Diag}(\boldsymbol{a})$ denotes a diagonal matrix with the vector $\boldsymbol{a}$ as the diagonal. Using these, we can write the gradient of Equation 4.30, where the gradient of the functional regulariser is added to the gradient of the loss,

$$N\nabla_{\boldsymbol{w}}\bar{\ell}_t(\boldsymbol{w}) + \tau \sum_{s=1}^{t-1} \left[\nabla_{\boldsymbol{w}}\boldsymbol{m}_{t,s}(\boldsymbol{w})\right] \boldsymbol{K}_{t-1,s}^{-1}\left(\boldsymbol{m}_{t,s}(\boldsymbol{w}) - \boldsymbol{m}_{t-1,s}\right), \tag{4.34}$$

where $\nabla_{\boldsymbol{w}}\boldsymbol{m}_{t,s}(\boldsymbol{w})[i] = \nabla_{\boldsymbol{w}}\left[\sigma\left(f_{\boldsymbol{w}}(\boldsymbol{x}_i)\right)\right] = \Lambda_{\boldsymbol{w}}(\boldsymbol{x}_i)\boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_i)^\top$. The regulariser is computed in subroutine `g_funcreg` in Algorithm 2, although we could also let automatic differentiation calculate this derivative.

The additional computations on top of Adam are highlighted in red in Algorithm 2. Every iteration requires functional gradients (in `g_funcreg`) whose cost is dominated by the computation of $\boldsymbol{J}_{\boldsymbol{w}}(\boldsymbol{x}_i)$ at all $\boldsymbol{x}_i \in \mathcal{M}_s, \forall s < t$. Assuming the size of the memorable past is M per task, this adds an additional $\mathcal{O}(MPt)$ computation, where P is the number of parameters and t is the task number. This increases only linearly with the size of the memorable past. We need three additional computations but they are required only *once per task*. First, inversion of $\boldsymbol{K}_s, \forall s < t$, which has cost $O(M^3 t)$. This is linear in number of tasks and is feasible when M is not too large. Second, computation of $\boldsymbol{v}_t$ in Equation 4.31 requires a full pass through the dataset $\mathcal{D}_t$, with cost $O(NP)$ where N is the dataset size. This cost can be reduced by estimating $\boldsymbol{v}_t$ using a minibatch of data (as is common in EWC (Kirkpatrick et al., 2017)). Finally, we need to find the memorable past $\mathcal{M}_t$, requiring a forward pass followed by picking the top M examples for the Lambda method (see `memorable_past_Lambda` in Algorithm 2).

4.5 Experiments

In this section, we perform experiments on continual learning benchmarks with FROMP and OGN-FROMP. We start with the cheaper FROMP algorithm. We see how it is consistent across many runs on the Toy-Gaussians benchmark, indicating its suitability for fixing the brittleness/inconsistent behaviour of weight-regularisation from Section 3.3. We then apply FROMP to our other benchmarks: Split MNIST, Permuted MNIST and Split CIFAR. We see FROMP performs very well on all benchmarks.

We then try the more expensive OGN-FROMP, only running on Split MNIST as a proof-of-concept. We see extremely good performance, but OGN-FROMP is more expensive, and this is a price we pay for the convenience of having better covariance information during training. Future work could look at scaling OGN-FROMP to larger benchmarks while not being too computationally expensive.

Details of benchmarks and metrics (we use average accuracy, forward transfer and backward transfer) are in Section 2.4. We use the same network architectures and protocol for benchmarks as described in Section 2.4, and as used in Chapter 3. Hyperparameter settings for all experiments is in Appendix C.3.

To identify the benefits of the functional prior (Step A) and memorable past (Step B), we compare FROMP to three variants:

1. FROMP-L_2 where we replace the kernel in Equation 4.33 by the identity matrix, similar to Equation 4.4,
2. FRO*R*P where memorable examples selected randomly ("R" stands for random),
3. FRO*R*P-L_2 which is same as FRO*R*P, but the kernel in Equation 4.33 is replaced by the identity matrix.

For most experiments, FROMP is run with the Lambda method for choosing memorable points (Section 4.3). We only use the Leverage method in one subsection.

4.5.1 Experiments with FROMP

We start by looking at FROMP's performance (see Algorithm 2) on four benchmarks: Toy-Gaussians, Split MNIST, Permuted MNIST, and Split CIFAR. We compare with VCL (Nguyen et al., 2018), EWC (Kirkpatrick et al., 2017), VCL+Coreset, VOGN and FRCL (Titsias et al., 2020). We find that FROMP empirically improves upon these previous methods, performing extremely well. Code is available at `https://github.com/team-approx-bayes/fromp`.

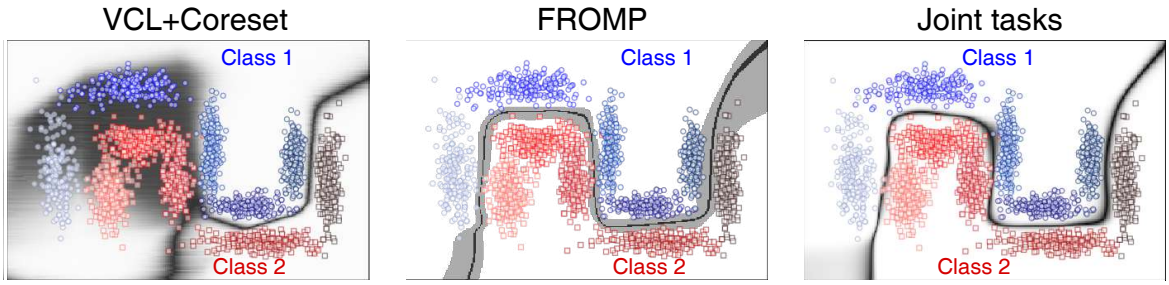


Figure 4.3: FROMP performs very well when continually trained over the 5 tasks of the Toy-Gaussians benchmark, performing similar to the Joint Tasks upper-bound baseline. VCL+Coreset, on the other hand, performs poorly, although it also stores a coreset of memory. We plot the middle-performing run of 5 runs for each method. Performance is summarised in Table 4.1.

FROMP	FRORP	VCL+RP	VCL+MP	VOGN	Joint Tasks
$99.6 \pm 0.2\%$	$98.5 \pm 0.6\%$	$92 \pm 10\%$	$85 \pm 14\%$	$79 \pm 11\%$	$99.70 \pm 0.03\%$

Table 4.1: Train accuracy of FROMP, FRORP, VCL+Coreset, VOGN and Joint Tasks (an upper-bound on performance) on Toy-Gaussians, with mean performance and standard deviation over 5 runs for VCL, VOGN and Joint Tasks, and 10 runs for FROMP and FRORP. VCL+RP and FRORP have the same (random) coreset selections. VCL+MP is provided with 'ideal' coreset points as chosen by an independent run of FROMP. VOGN and VCL+Coreset are brittle with high standard deviations, while FROMP and FRORP are stable, with FROMP slightly better and closer to Joint Tasks performance.

Toy-Gaussians

We previously saw in Section 3.3 (see Figure 3.15 and Table 3.5) how weight-regularisation such as VCL can be brittle and inconsistent on the simple 2D binary classification Toy-Gaussians benchmark. In Figure 4.3 we now see how FROMP performs well on this dataset, unlike the previous method VCL+Coreset. We summarise results in Table 4.1, where FROMP is within error of Joint Tasks performance. Even FRORP, which uses a random memory, is close behind, and significantly better than weight-regularisation methods. There are two versions of VCL+Coreset: (i) VCL+RP uses random points in the coreset, and (ii) VCL+MP uses the same memorable points as FROMP. We see that the choice of coreset points do not significantly affect VCL+Coreset's performance, and VCL+Coreset is always extremely brittle: it can perform well sometimes (1 run out of 5), but usually does not (4 runs out of 5).

We also look at the performance of FROMP with different dataset variations of Toy-Gaussians in Table 4.2. We see that FROMP performs consistently well, indicating that it

Dataset variation	FROMP	Joint Tasks
Original Toy-Gaussians dataset	$99.6 \pm 0.2\%$	$99.7 \pm 0.0\%$
10x less data (400 per task)	$99.9 \pm 0.0\%$	$99.7 \pm 0.2\%$
10x more data (40000 per task)	$96.9 \pm 3.0\%$	$99.7 \pm 0.0\%$
Introduced 6th task	$97.8 \pm 3.3\%$	$99.6 \pm 0.1\%$
Increased std dev of each class distribution	$96.0 \pm 2.4\%$	$96.9 \pm 0.4\%$
2 tasks have overlapping data	$90.1 \pm 0.8\%$	$91.1 \pm 0.3\%$

Table 4.2: Train accuracy of FROMP and Joint Tasks (upper-bound on performance) on variations of Toy-Gaussians, with mean performance and standard deviations over 10 runs for FROMP and 3 runs for Joint Tasks. FROMP performs well across variations. See Appendix C.4 for visualisations of these dataset variations.

is robust. This indicates it might also perform well on larger benchmarks. We visualise the different dataset variations in Appendix C.4.

Finally, in Appendix C.5 we show the importance of the kernel being over all weights on Toy-Gaussians. FROMP's kernel is over all weights, as opposed to other methods such as FRCL (Titsias et al., 2020). We argue that this is especially important earlier in training (on earlier tasks), when all the weights in the neural network are still changing significantly.

Split MNIST and Permuted MNIST

We have seen how FROMP performs consistently well on Toy-Gaussians, showing robustness to dataset variations, and we hope that this performance translates over to larger benchmarks with images. We start by looking at performance on Split MNIST and Permuted MNIST, using the same experimental protocol as described in Section 2.4 and used in Chapter 3. For Permuted MNIST, we set the number of memorable examples in the range 10–200 per task. For Split MNIST, we select 40 points per task.

The final average accuracy is shown in Table 4.3, where FROMP achieves better performance than weight-regularisation methods (EWC, SI, VCL, VOGN, VCL+Coreset) as well as FRCL (Titsias et al., 2020). FROMP also improves over FRORP-L_2 and FROMP-L_2, demonstrating the effectiveness of the kernel. The improvement compared to FRORP is not significant on Split MNIST. We believe this is because a random memorable past is already close to the highest achievable performance with our method, and we see no further improvement by choosing the examples carefully. However, as shown in Figure 4.4(a) on Permuted MNIST, we see larger improvements when the number of memorable examples is smaller (compare FROMP vs FRORP). Finally, Figure 4.2(a) shows the most and least memorable examples chosen by sorting $\Lambda_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{y})$. The most memorable examples appear to

Method	Split MNIST ACC (%)	Permuted MNIST ACC (%)	Permuted MNIST FWT (%)	Permuted MNIST BWT (%)
EWC	63.1	84	–	–
Improved VCL	98.5±0.4	93±1	-0.2±0.1	-4±1
+ Random Coreset	98.2±0.4	**94.6**±0.3	-0.2±0.1	-2.3±0.3
VOGN	**98.8**±0.1	**94.0**±0.8	-0.6±0.1	-4±1
FRCL-RND	97.1±0.7	94.2±0.1	–	–
FRCL-TR	97.8±0.7	94.3±0.2	–	–
FRORP-L_2	98.5±0.2	87.9±0.7	–	–
FROMP-L_2	98.7±0.1	94.6±0.1	–	–
FRORP	**99.0**±0.1	94.6±0.1	–	–
FROMP	**99.0**±0.1	**94.9**±0.1	-1.9±0.1	-1.0±0.1

Table 4.3: Comparing average accuracy (ACC) of various methods on Permuted MNIST (10 tasks) and Split MNIST (5 tasks). FROMP performs the best overall, outperforming previous approaches such as EWC (Kirkpatrick et al., 2017) and also FRCL (Titsias et al., 2020). Improved VCL and VOGN results are from Chapter 3 (and Eschenhagen (2019)). We also compare forward transfer (FWT) and backward transfer (BWT) on Permuted MNIST for FROMP, VOGN and VCL (definitions of metrics are in Section 2.4). FROMP has lower FWT, indicating it does not learn as good an initial performance on new tasks, but has better BWT, indicating it does not forget as much as other methods. For Permuted MNIST, we use 200 examples per task as the memorable past / coreset / inducing points. For Split MNIST, we use 40 examples per task.

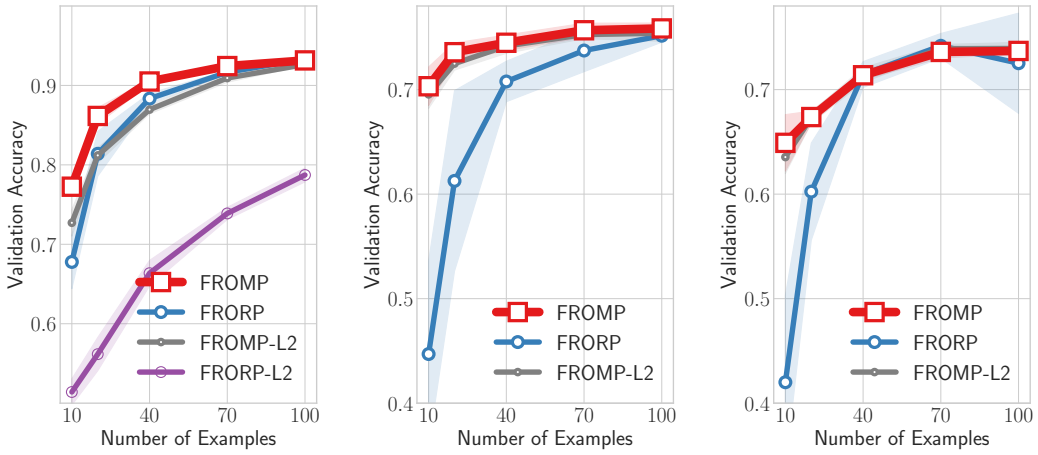
be more difficult to classify than the least memorable examples, which suggests that they may lie closer to the decision boundary.

We also look at forward transfer (FWT) and backward transfer (BWT) metrics for Permuted MNIST in Table 4.3 (see Chapter 3 for details on Improved VCL, VCL+Coreset and VOGN). We see that FROMP achieves its better performance by having worse FWT but better BWT. This indicates that FROMP does not perform as well as other methods immediately as a new task is learnt, but it offsets this by not forgetting past tasks as much as other methods.

We also run FROMP on Split MNIST on the smaller network architecture of VCL, with one hidden layer of 200 units (instead of two hidden layers), obtaining $99.2 \pm 0.1\%$.

Split CIFAR

We also run FROMP on the larger Split CIFAR task, using the same experimental protocol as previously described in Section 2.4 and used in Chapter 3. We run on the CifarNet architecture from Zenke et al. (2017). The number of memorable past points is set in the range 10–200, and we run each method 5 times.



(a) Permuted MNIST **(b)** Split CIFAR **(c)** Split CIFAR (11 tasks)

Figure 4.4: All figures show the average accuracy (after training on all tasks) with respect to the number of memorable examples per task. (a) On Permuted MNIST, we see that FROMP outperforms FRORP and L_2 variants at small numbers of past examples. (b) On Split CIFAR, FROMP outperforms FRORP (note that FRORP-L_2 performs significantly worse), but FROMP-L_2 is usually within error of FROMP, even at small memory sizes. (c) A similar story is seen for Split CIFAR with 11 tasks instead of 6 tasks.

The results are summarised in Figure 4.5, where we see that FROMP is close to the upper limit while outperforming all the other methods (see the final ACC column for average accuracy). The weight-regularisation methods EWC and SI do not perform well on later tasks while VCL(+Coreset) forgets earlier tasks. VOGN performs well but not as good as FROMP, and has not learnt more recent tasks very well. FROMP performs consistently better than VOGN across all but the first task. FROMP also improves over the Separate Tasks baseline by a large margin. In fact, on tasks 4-6, FROMP matches the performance to the network trained jointly on all tasks, which implies that it completely avoids forgetting on these tasks.

Figure 4.4(b) shows the performance with respect to the number of memorable past examples. Similarly to Figure 4.4(a), carefully selecting memorable example improves performance, especially when the number of memorable examples is small. For example, with 10 memorable examples, a careful selection in FROMP increases the average accuracy to 70% from 45% obtained by FRORP. Including the kernel in FROMP here does not improve significantly over FROMP-L_2, unlike in Permuted MNIST. We also look at 11 tasks of Split CIFAR instead of 6 tasks (the first task is CIFAR-10, and the following 10 tasks are 10 classes each from CIFAR-100), and Figure 4.4(c) shows a very similar result to the 6-task version. Figure 4.2(b) shows the most and least memorable past examples in CIFAR-10, where we again see that the most memorable might be more difficult to classify.

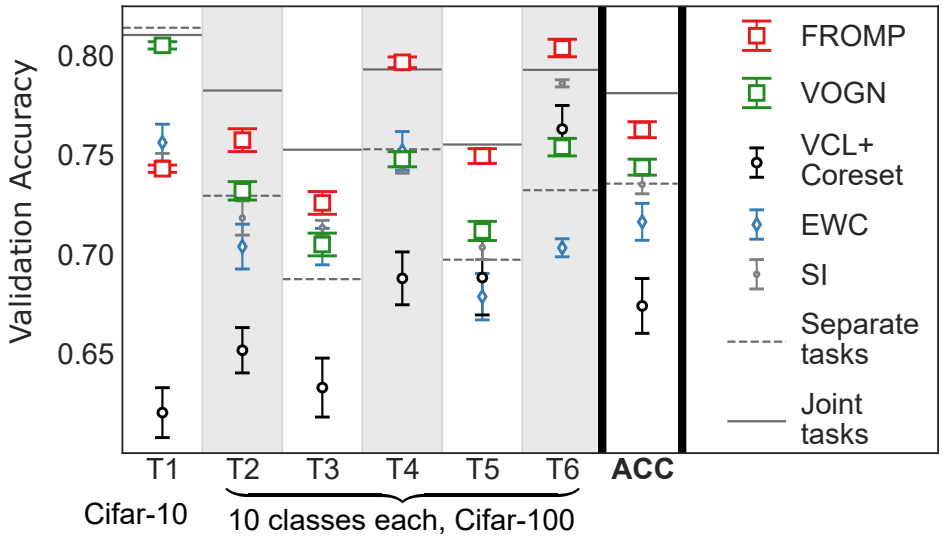
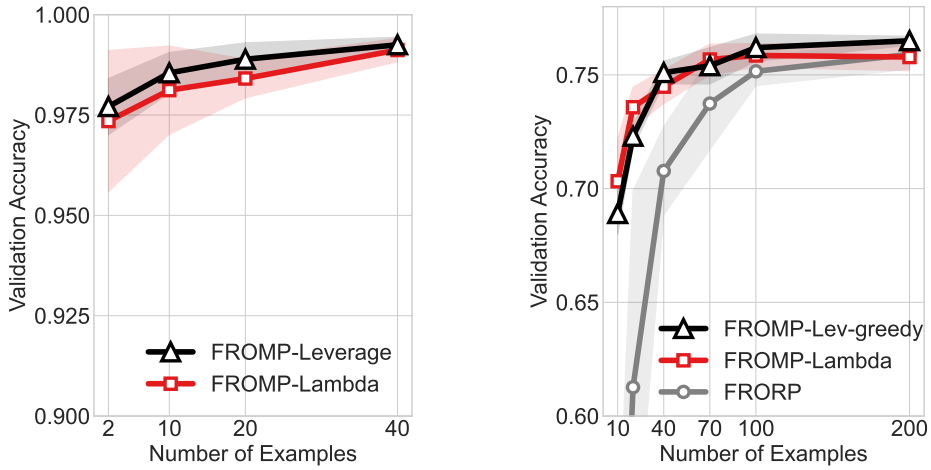


Figure 4.5: Individual task accuracy after training on the final task, and average task accuracy (ACC). FROMP outperforms VOGN (and all other weight-regularisation baselines), as seen in the ACC column. Additionally, on tasks 4-6, FROMP matches the performance of the network trained jointly on all tasks, which implies that it avoids forgetting on these tasks. For these results, FROMP and VCL+Coreset store 200 points per task.

Finally, we analyse the forward and backward transfer obtained by FROMP, summarised in Table 4.4. We find that FROMP's forward transfer is much better than VCL and EWC, while its backward transfer is comparable to EWC. FROMP achieves a forward transfer of $6.1 \pm 0.7\%$, a much higher value compared to $0.17 \pm 0.9\%$ obtained with EWC, $1.8 \pm 3.1\%$ with VCL+coresets and $0.8 \pm 2.0\%$ with VOGN. For backward transfer, FROMP has a score of $-2.6 \pm 0.9\%$, which is comparable to EWC's score of $-2.3 \pm 1.4\%$ but better than VCL+Coreset which obtains $-9.2 \pm 1.8\%$. VOGN has the best backward transfer, with $-0.7 \pm 0.5\%$.

Benchmark	Metric	EWC	VCL+Coreset	VOGN	FROMP
Split CIFAR	ACC (%)	71.6±0.9	48.8±2.2	74.4±0.4	**76.2**±0.4
	FWT (%)	0.2±0.9	0.8±2.0	1.8±0.3	6.1±0.7
	BWT (%)	-2.3±1.4	-29±4	-0.7±0.5	-2.6±0.9

Table 4.4: Final average test accuracy, forward transfer and backward transfer on Split CIFAR for various methods. Mean performance and standard deviation over 5 runs. We see that FROMP has the best average accuracy, as well as the best forward transfer. FROMP's backward transfer matches EWC's, but is worse than VOGN's. For these results, FROMP and VCL+Coreset store 200 points per task.



(a) Split MNIST **(b)** Split CIFAR

Figure 4.6: Both figures show the average accuracy (after training on all tasks) with respect to the number of memorable examples per task. On Split MNIST, we see that FROMP-Leverage (which uses the Leverage method for choosing memorable points) is consistently marginally better than FROMP-Lambda (which uses the significantly cheaper Lambda method for choosing points). On Split CIFAR, we use FROMP-Lev-greedy, which greedily chooses the points with top leverage score, instead of sampling points according to their leverage score. FROMP-Lev-greedy is always within error of FROMP-Lambda (both methods significantly outperform random sampling with FRORP at smaller memories). Using FROMP-Leverage (without greedily choosing points) performs slightly worse on Split CIFAR when only storing a few points, likely because of the additional randomness due to sampling. We use the Lambda method for all our other experiments with FROMP.

FROMP-Leverage

All results so far with FROMP have been with the Lambda method for choosing memorable past points. As discussed in Section 4.3, we can also use different methods for choosing memorable points. We now consider the Leverage method, which was described in Section 4.3. We denote this method FROMP-Leverage, and run it on Split MNIST and Split CIFAR, with all hyperparameters the same as used in the earlier experiments (we now call the previous method FROMP-Lambda as it uses the Lambda method for choosing memorable past points).

We look at results on Split MNIST with reducing number of memorable past points in Figure 4.6(a), and results for Split CIFAR in Figure 4.6(b). On Split MNIST, across different numbers of memorable past points, we find that FROMP-Leverage consistently marginally outperforms FROMP-Lambda: FROMP-Leverage has higher mean performance but is within a standard deviation (each method is run 5 times). For example, on Split MNIST with 40

memorable past examples per task, FROMP-Leverage achieves an average accuracy after all tasks of $99.3 \pm 0.2\%$, while FROMP-Lambda gets $99.0 \pm 0.1\%$.

On Split CIFAR, we use FROMP-Leverage-greedy (or 'FROMP-Lev-greedy'), which calculates the leverage score per datapoint, but then greedily chooses the points with top leverage score instead of sampling points with probability proportional to their score. FROMP-Leverage-greedy performs well, and again is always within error of FROMP-Lambda. At larger memories the leverage method performs marginally better, for example, with 200 memorable past examples per task, FROMP-Leverage-greedy gets $76.5 \pm 0.2\%$, while FROMP-Lambda gets $76.2 \pm 0.4\%$. However, at smaller memories, FROMP-Leverage-greedy does not always (marginally) outperform FROMP-Lambda.

On Split CIFAR with small memories, FROMP-Leverage (without greedy sampling) performs worse than FROMP-Leverage-greedy and FROMP-Lambda. For example, with 10 memorable past examples per task (this corresponds to 1 example per class), FROMP-Leverage gets $65.4 \pm 3.5\%$, while FROMP-Leverage-greedy gets $68.9 \pm 0.9\%$. This worse performance is surprising as, theoretically, sampling according the leverage score should be better than greedily sorting points (Alaoui and Mahoney, 2015; Ma et al., 2015). However, in continual learning, it appears that the additional randomness from sampling reduces performance over many tasks. In such small-memory settings, it appears beneficial to be more exploitative by using a greedy method.

The Leverage method could also be performing worse than expected due to approximations we made to reduce computation costs on Split CIFAR. We made a block-diagonal approximation to reduce the cost of inversion (see Equation 4.17 and discussion around it) due to the large number of datapoints.

Overall, our experiments in this section indicate that choosing memorable past datapoints using the Leverage method does not significantly improve performance over the Lambda method. This is despite the Leverage method being more expensive than the Lambda method, as it requires inversion of large matrices. The Lambda method is already very good for finding memorable past points on our benchmarks.

This is an interesting area of future research, and the subject of an ongoing project. Potential ideas are discussed in more detail in Section 6.2.

4.5.2 Experiments with OGN-FROMP

We now look at how well OGN-FROMP performs in practice. OGN-FROMP makes fewer approximations to variational-FROMP than FROMP, and is described in Section 4.4.1. A key difference to FROMP is that OGN-FROMP maintains a covariance matrix $\mathbf{\Sigma}$ during training. It is therefore more computationally expensive than FROMP, and requires tuning of some

more hyperparameters (similarly to how VOGN has more hyperparameters than Adam, as discussed in Section 3.2.1).

We only run OGN-FROMP on Split MNIST as a proof-of-concept, and find that OGN-FROMP performs very well (hyperparameter values are in Appendix C.3). We run with the same two-hidden-layer network with 256 hidden units per layer as in previous work and as described in Section 2.4. OGN-FROMP obtains a final average accuracy of $99.6 \pm 0.1\%$, which is very close to Joint Tasks performance, and significantly better than all other methods, including FROMP, which achieved $99.0 \pm 0.1\%$. This shows that OGN-FROMP has the potential to perform very well.

OGN-FROMP also has additional benefits stemming from maintaining a covariance matrix during training. As we now maintain a full Gaussian distribution over weights during training, we can convert our distribution over weights to a distribution over functions at any point during training. This is important if we move to the setting where task boundaries are not provided to us (see discussion in Section 2.1), as we are now able to calculate the kernel matrix $\boldsymbol{K}_{t-1}$ in a more online fashion, instead of only calculating it at defined task boundaries.

4.6 Summary

We began this chapter by looking at how we might perform function-regularisation for continual learning instead of just weight-regularisation, which we argued in Section 3.3 made restrictive independence assumptions leading to poor performance. Section 4.1 discussed simple ways to perform function-regularisation, discussed previous works, and introduced our method FROMP. By functionally regularising over stored datapoints, FROMP is a combination of regularisation-based and rehearsal-based approaches to continual learning.

FROMP has three steps (see also Figure 4.1). All three steps are performed in a single framework. Step A (detailed in Section 4.2) converts a distribution over weights of a neural network into a distribution over functions (a Gaussian Process functional prior). We use the DNN2GP method (Khan et al., 2019), which uses a linear model view during optimisation with natural-gradient variational inference algorithms. In Step B (detailed in Section 4.3), we use the linear model view to motivate methods for choosing points to store in memory. We only store a few points from the dataset, choosing points that are crucial to avoid forgetting.

In Step C (detailed in Section 4.4), when we see new data, we optimise the weights of our neural network while functionally regularising over the memorable past datapoints. We do this by approximating the log-prior term in weight-space with one in function-space, calculated over our memorable past points. We derived var-FROMP, which runs a natural-

gradient variational inference algorithm on our objective function. However, var-FROMP is computationally expensive, and we considered how to approximate it, leading to two different algorithms.

Our first algorithm, OGN-FROMP, maintains a covariance matrix during training (like the VOGN algorithm in Section 3.2). We provide initial results with OGN-FROMP in Section 4.5.2, where we see potential to perform very well. Our second algorithm, FROMP (see Algorithm 2), makes more approximations than OGN-FROMP, and is much cheaper. Crucially, we see in Section 4.5.1 that FROMP performs very well on many continual learning benchmarks, outperforming our previous weight-prior methods VCL and VOGN from Chapter 3.

Future work could consider improving various aspects of FROMP and var-FROMP:

1. **Improving memorable past datapoints selection:** We saw that the Leverage method did not empirically improve upon the cheaper Lambda method, and we could investigate why this is the case (theoretically, the Leverage method should perform better). In Section 5.4 we show a different motivation resulting in the same Leverage and Lambda methods, and in Section 6.2 we describe ongoing work on improving on these methods.

2. **Analysing the objective function:** In this chapter, we simply approximated the log-prior term in the variational objective with one in function-space. We did not analyse this approximation theoretically. We make some steps regarding theoretical analysis in the next chapter, where we introduce a framework for general adaptation. This allows us to gain some insight into why FROMP performs well, and we provide suggestions on how to improve FROMP further.

3. **Experimental evaluation of OGN-FROMP:** We could run OGN-FROMP on larger benchmarks, and also try OGN-FROMP in the no-task-boundary setting. This would require more implementation effort due to additional hyperparameters in OGN-FROMP compared to FROMP, but there are many potential benefits as discussed in Section 4.5.2. Other future work would also consider relaxing some of the approximations we make in FROMP and OGN-FROMP (see Approximations 1-5 in Section 4.4).

In the next chapter, we describe a framework to perform general adaptation, and consider algorithms that perfectly remember all past information with sufficient memory. We will see that FROMP is not always exact even on simple problems (such as linear regression). Our framework will allow us to theoretically analyse FROMP, seeing that it tries to maintain second-order information from previous data. This will lead to potential ways to improve FROMP (see Section 5.6).

Chapter 5

Knowledge-adaptation priors

In this chapter, we present Knowledge-adaptation priors (K-priors) for quick and accurate adaptation for a wide variety of tasks and models. K-priors achieve such adaptation by combining weight-space and function-space divergences to reconstruct the gradient of past information, and are a generalisation of many adaptation methods, including weight-priors from Chapter 3 and FROMP from Chapter 4. Previous methods are limited as they can only be applied to a single adaptation task, such as adding data in continual learning. Previous methods also do not recover the exact model trained on all data when choosing a sufficiently large memory of past data.

By generalising previous methods, K-priors are immediately applicable to a wide range of adaptation tasks: we will focus on the case of adding new data (like in continual learning), but will also consider removing data, changing the regulariser, and changing the model class or architecture (see the left of Figure 5.1). K-priors also recover the exact model trained on all data, obtaining the same solution as retraining-from-scratch on all past data when choosing a sufficiently large memory. When considering smaller memory sizes, our theory in this chapter leads to natural adaptation-mechanisms where model predictions need to be readjusted only at a handful of past experiences (a toy example is shown on the right of Figure 5.1).

The key idea in K-priors is to reconstruct the gradient of past information by combining weight-space and function-space divergences. We apply K-priors to both the variational setting and the MAP/Laplace setting. We will see how K-priors in the variational setting relate to variational methods like VCL (Section 3.1) and var-FROMP (Chapter 4), while the K-priors in the MAP/Laplace setting link to methods like Online EWC (Kirkpatrick et al., 2017) and FROMP (Chapter 4).

In general, our theory with K-priors recovers and generalises many existing adaptation strategies, including some existing Bayesian algorithms with weight-priors (Cesa-Bianchi

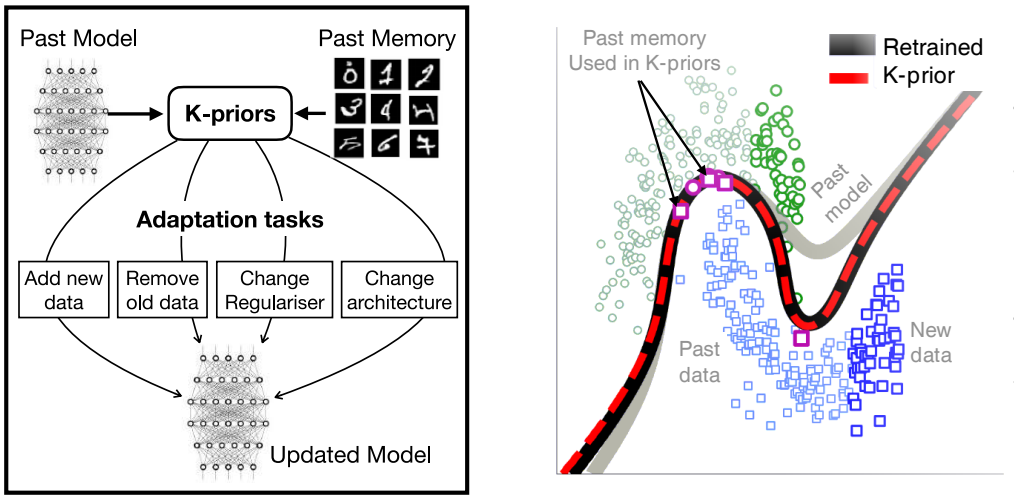


Figure 5.1: Left: K-priors can handle a wide variety of adaptation tasks by using the past model and a small memory of the past data, while being quicker than retraining-from-scratch. The task of adding new data is the same as continual learning, but K-priors can also handle more adaptation tasks. **Right:** We show how K-priors can achieve adaptation with little memory in a toy setting. In this figure, the past model (grey line) is trained on past data, before new datapoints (on the right of the plot) are added. K-priors only need to store a handful of past examples (dark purple markers) to get the new, updated model (dashed red line). This updated model is very close to the retraining-from-scratch (solid black line) solution.

and Lugosi, 2006; Kirkpatrick et al., 2017; Nguyen et al., 2018; Ritter et al., 2018; Schwarz et al., 2018), sparse online Gaussian Processes (Csató and Opper, 2002), algorithms for support vector machines (Cauwenberghs and Poggio, 2001; Liang and Li, 2009; Tsai et al., 2014), continual learning algorithms (Li and Hoiem, 2016; Rebuffi et al., 2017; Pan et al., 2020; Buzzega et al., 2020), and knowledge distillation (Hinton et al., 2015; Lopez-Paz et al., 2016). Previous works apply to narrow, specific adaptation tasks, while K-priors apply to a wide range of adaptation tasks. We call our approach Knowledge-adaptation priors as they provide a principled way of adapting knowledge across many adaptation tasks within a probabilistic framework.

We start in Section 5.1 by considering the adaptation task of adding new data to an already-trained base model, and present vanilla K-priors for adaptation on Generalised Linear Models (GLMs). Vanilla K-priors can be exact when we choose a sufficiently large memory of past data. We then apply vanilla K-priors to neural networks in Section 5.2. There is now an error term, unlike with GLMs, but we draw comparisons with knowledge distillation (Hinton et al., 2015), which also has this term and performs *better* than retraining-from-scratch. Our K-priors theory provides some insights into why knowledge distillation works

well, and we exploit the link between the methods to improve both K-priors and knowledge distillation. Section 5.3 then applies K-priors to adaptation tasks other than just adding data, such as removing data, changing the regulariser, and changing the model class or architecture.

Vanilla K-priors provide guarantees with sufficiently large memory, but we are interested in limited-memory settings too, and we analyse this in Section 5.4. For example, we present the optimal K-prior, which uses a decomposition of the feature matrix to reconstruct the gradient of past information using the optimal number of points in memory, but is difficult to realise in practice. We also consider storing a subset of past inputs in memory. By analysing the error term in K-priors, we motivate algorithms for choosing these memory points, including the Lambda method and Leverage method from Section 4.3.

We also present ways to design divergences in K-priors that efficiently use available memory. In Section 5.5 we see the relationship between K-priors and weight-priors (such as VCL and Online EWC), and present Quadratic K-priors for combining the strengths of weight-priors with functional regularisation from vanilla K-priors. In Section 5.6 we consider FROMP as being in the K-priors framework, finding that we can view FROMP as matching second-order information. This provides insight into why FROMP has strong empirical performance (see results in Section 4.5), and we further suggest improvements to FROMP.

We provide experiments in Section 5.7. We show that K-priors can achieve good performance on a wide variety of tasks and models with small memory. We run vanilla K-priors on GLMs and neural networks with different datasets on all our four adaptation tasks, finding that K-priors are consistently quicker than retraining-from-scratch while achieving good performance. We also run Quadratic K-priors on the Split MNIST benchmark for continual learning, seeing strong performance. Finally, in Section 5.8 we briefly explore the link between K-priors and other adaptation mechanisms for Support Vector Machines and online Gaussian Processes, seeing that K-priors unify and generalise many of the existing ideas.

5.1 Reconstructing the gradient of the past

Our key idea in this chapter is to faithfully reconstruct the gradient of past information using K-priors. In this section, we see how we can do this on Generalised Linear Models. We focus on the adaptation task of adding data, as in continual learning.

We consider two settings: the MAP/Laplace setting and the variational setting. The MAP/Laplace setting calculates the (deterministic) solution of a Maximum-A-Posteriori (MAP) problem, and calculates the K-prior around this solution, similarly to the Laplace method (although we do not necessarily use a Gaussian approximation). The variational setting uses the variational objective function (for further details see Section 2.2.1), and in this chapter, we only consider a Gaussian approximating family distribution.

5.1.1 Adding new data

We want to quickly and accurately adapt an already trained model to incremental changes in its training framework. We focus on the adaptation task of adding data, and call this the 'Add Data' task. We now introduce the 'Add Data' task in both the MAP/Laplace and variational settings mathematically, and then in Section 5.1.2 we see how vanilla K-priors can reconstruct the gradient of past information on Generalised Linear Models. We will refer to the model trained on old data $\mathcal{D}_{\text{old}}$ as the *base model*.

The MAP/Laplace setting

First we consider a Laplace-style setting, where we are training for a deterministic setting of the weights $\boldsymbol{w}_* \in \mathcal{W} \subset \mathbb{R}^P$, obtained by solving the following problem,

$$\begin{aligned} \boldsymbol{w}_* &= \arg\min_{\boldsymbol{w}} \ \mathcal{L}^{\text{MAP}}(\boldsymbol{w}), \\ \text{where } \mathcal{L}^{\text{MAP}}(\boldsymbol{w}) &= \sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) + \mathcal{R}(\boldsymbol{w}). \end{aligned} \tag{5.1}$$

Here, $\mathcal{R}(\boldsymbol{w})$ is a regulariser, and $\ell_i(\boldsymbol{w})$ is the loss function on the i'th data example. When viewing this as a MAP solution, $\mathcal{R}(\boldsymbol{w}) = -\log p(w)$ comes from the prior, and $\sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) = -\log p(\mathcal{D}_{\text{old}}|\boldsymbol{w})$ is the negative log-likelihood of training data.

In the 'Add Data' task, we add some new data $\mathcal{D}_{\text{new}}$ to the model. Ideally, if we could retrain on all data, then we can get the retrained-from-scratch solution $\boldsymbol{w}_+$ by optimising,

$$\boldsymbol{w}_+ = \arg\min_{\boldsymbol{w}} \sum_{j \in \mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}) + \sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) + \mathcal{R}(\boldsymbol{w}). \tag{5.2}$$

With K-priors, we would like to reconstruct the gradient of past information, which refers to the latter two terms in the equation above (these are the terms repeated from Equation 5.1). Note that this 'Add Data' task can be viewed as continual learning with two tasks only. We focus on the simple two-task setup as it allows us to precisely characterise any errors. We could then expand to multiple tasks, where errors will build up over many tasks.

The variational setting

In the variational setting, we minimise the variational objective (Equation 2.6) for the parameters $\boldsymbol{\eta}$ of our approximating distribution $q_{\boldsymbol{\eta}}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$, where we use a Gaussian approximating family. Our variational objective for the base model is,

$$\begin{aligned}
\boldsymbol{\eta}_* &= \arg\min_{\boldsymbol{\eta}} \; \mathcal{L}^{\text{var}}(\boldsymbol{\eta}), \\
\text{where } \mathcal{L}^{\text{var}}(\boldsymbol{\eta}) &= \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[-\log p(\mathcal{D}_{\text{old}}|\boldsymbol{w})\right] + \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[\log \frac{q_{\boldsymbol{\eta}}(\boldsymbol{w})}{p(\boldsymbol{w})}\right] \\
&= \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[\sum_{i\in\mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w})\right]. \qquad (5.3)
\end{aligned}$$

Here, $p(\boldsymbol{w})$ is our prior over parameters, and as before, $-\log p(\mathcal{D}_{\text{old}}|\boldsymbol{w}) = \sum_{i\in\mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w})$. The base model is $q_{\boldsymbol{\eta}_*}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$. We can write the following fixed-point solutions that $q_{\boldsymbol{\eta}_*}(\boldsymbol{w})$ obeys (see Section 3 in Khan and Rue (2021) for an expression),

$$0 = \nabla_{\boldsymbol{\mu}} \mathbb{E}_q[\mathcal{L}(\boldsymbol{w})]\big|_{\boldsymbol{\mu}=\boldsymbol{\mu}_*, \boldsymbol{\Sigma}=\boldsymbol{\Sigma}_*} = \mathbb{E}_q[\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})]\big|_{\boldsymbol{\mu}=\boldsymbol{\mu}_*, \boldsymbol{\Sigma}=\boldsymbol{\Sigma}_*}, \qquad (5.4)$$

$$\boldsymbol{\Sigma}_*^{-1} = \nabla_{\boldsymbol{\Sigma}} \mathbb{E}_q[\mathcal{L}(\boldsymbol{w})]\big|_{\boldsymbol{\mu}=\boldsymbol{\mu}_*, \boldsymbol{\Sigma}=\boldsymbol{\Sigma}_*} = \mathbb{E}_q[\nabla^2_{\boldsymbol{w}\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})]\big|_{\boldsymbol{\mu}=\boldsymbol{\mu}_*, \boldsymbol{\Sigma}=\boldsymbol{\Sigma}_*}, \qquad (5.5)$$

where $\mathcal{L}(\boldsymbol{w}) = \sum_{i\in\mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w})$. For the second equality in both expressions, we have used Bonnet's and Price's theorems (Opper and Archambeau, 2009; Rezende et al., 2014) (see also Equations 2.23 and 2.24). We will use these fixed-point solutions later.

The variational setting (Equation 5.3) and the MAP/Laplace setting (Equation 5.1) are closely related, which we can note in many ways. One way is to notice that the expression for $\mathcal{L}^{\text{var}}$ contains $\mathcal{L}^{\text{MAP}}$, with two differences overall: (i) it takes the expectation of $\mathcal{L}^{\text{MAP}}(\boldsymbol{w})$ with respect to $q_{\boldsymbol{\eta}}(\boldsymbol{w})$, and (ii) it also has a (negative) entropy term $\mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}[\log q_{\boldsymbol{\eta}}(\boldsymbol{w})]$. Another way to note the relationship is to view the MAP/Laplace setting as making two approximations to the variational setting (Khan and Rue, 2021): (i) set our Gaussian approximating family to be $q_{\boldsymbol{\eta}}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \mathbf{I})$ with unknown mean $\boldsymbol{\mu}$ but known and constant identity variance $\mathbf{I}$, (ii) use a first-order delta method to approximate the expectation term, $\mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}[g(\boldsymbol{w})] \approx g(\boldsymbol{\mu})$.

We will use this relationship between the two settings throughout this chapter. This allows us to derive results for the MAP/Laplace setting, and then straightforwardly apply similar results to the variational setting.

In the 'Add Data' task, we add some new data $\mathcal{D}_{\text{new}}$, and the ideal retrained-from-scratch solution gives $\boldsymbol{\eta}_+$ by optimising,

$$\boldsymbol{\eta}_+ = \arg\min_{\boldsymbol{\eta}} \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{j \in \mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}) + \sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w}) \right]. \tag{5.6}$$

As before, we would like to reconstruct the gradient of past information, which refers to $\mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_i \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w})\right]$ in the equation above.

Notation and problem setting

We now briefly recap some additional notation which we will use in this chapter (these definitions are also in Chapter 2). Throughout, we will use a supervised problem where the loss is specified by an exponential family,

$$\ell(y, h(f)) = -\log p(y|f) = -\langle y, f \rangle + A(f), \tag{5.7}$$

where $y \in \mathcal{Y}$ denotes the scalar observation output, $f \in \mathcal{F}$ is the canonical natural parameter, $A(f)$ is the log-partition function, and $h(f) = \mathbb{E}(y) = \nabla_f A(f)$ is the expectation parameter (also the inverse link function). Note that we have assumed that the base measure (see Equation 2.7) is constant and so can be ignored. A typical example is the cross-entropy loss for binary outcomes $y \in \{0, 1\}$ where $A(f) = \log(1 + e^f)$ and $h(f) = \sigma(f)$ is the sigmoid function. For ease of notation, we restrict our equations to consider the scalar output case, but it is straightforward to extend our method to a vector observation and model outputs. We briefly consider extensions to other types of learning frameworks in Section 5.3.4.

In this chapter, we use a shorthand for the model outputs, using $f^i_{\boldsymbol{w}} = f_{\boldsymbol{w}}(\boldsymbol{x}_i)$. We will often refer to $h'(f) = \nabla_f h(f)$, which is usually easy to calculate: for example, for the cross-entropy loss, $h'(f) = \sigma(f)\,[1 - \sigma(f)]$. This was also referred to as $\Lambda_{\boldsymbol{w}}(\boldsymbol{x}, y)$ in Chapter 4. We will also repeatedly make use of the following expression for the derivative of the loss,

$$\begin{aligned} \nabla_{\boldsymbol{w}} \ell(y_i, h(f^i_{\boldsymbol{w}})) &= -\nabla_{\boldsymbol{w}} \log p(y_i|f^i_{\boldsymbol{w}}) \\ &= \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}}\ [\nabla_f A(f^i_{\boldsymbol{w}}) - y_i] \\ &= \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}}\ [h(f^i_{\boldsymbol{w}}) - y_i]. \end{aligned} \tag{5.8}$$

Knowledge-adaptation priors (K-priors)

We present Knowledge-adaptation priors (K-priors) to quickly and accurately adapt a model's knowledge to a wide variety of changes in its training framework. K-priors refer to a class of priors that use both weight and function-space regularisers. We refer to the variational K-prior (var-K-prior) as $q_{\mathcal{K}}(\boldsymbol{w})$ and the Laplace K-prior (Lap-K-prior) as $\mathcal{K}(\boldsymbol{w})$. As we will often do throughout this chapter, we start with the expression for the Lap-K-prior. We can use the relationship between the Laplace setting and variational setting to write down the var-K-prior, and we show this with an example in Section 5.1.2.

We explicitly write the Lap-K-prior's dependence on the base model parameters $\boldsymbol{w}_*$ and a memory set $\mathcal{M}$,

$$\mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}) = \mathbb{D}_f \left(\boldsymbol{f}(\boldsymbol{w}) \| \boldsymbol{f}(\boldsymbol{w}_*) \right) + \tau \, \mathbb{D}_w \left(\boldsymbol{w} \| \boldsymbol{w}_* \right), \tag{5.9}$$

where $\boldsymbol{f}(\boldsymbol{w})$ is a vector of $f_{\boldsymbol{w}}(\boldsymbol{u}_m)$, defined at inputs $\boldsymbol{u}_m$ in $\mathcal{M} = (\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_M)$. The divergence $\mathbb{D}_f(\cdot\|\cdot)$ measures the discrepancies in the function space $\mathcal{F}$, while $\mathbb{D}_w(\cdot\|\cdot)$ measures the same in the weight space $\mathcal{W}$. Throughout, we will use Bregman divergences $\mathcal{B}_\psi(p_1, p_2) = \psi(p_1) - \psi(p_2) - \nabla\psi(p_2)^\top(p_1 - p_2)$, specified using a strictly-convex Bregman function $\psi(\cdot)$.

K-priors are defined using the base model $\boldsymbol{w}_*$ (for var-K-priors this will be $\boldsymbol{\eta}_*$), the memory set $\mathcal{M}$, and a trade-off parameter $\tau > 0$. We keep $\tau = 1$ unless otherwise specified. K-priors might also use other parameters required to define the divergence functions. We will sometimes omit the dependency on parameters and refer to just $\mathcal{K}(\boldsymbol{w})$ (or $q_{\mathcal{K}}(\boldsymbol{w})$).

Our general principle of adaptation is to use $\mathcal{K}(\boldsymbol{w})$ to faithfully reconstruct the gradient of the past training objective. K-priors therefore match the objective function of the past training objective (up to a constant). This is possible due to the combination of weight and function-space divergences. Next, we illustrate this point for supervised learning with Generalised Linear Models.

5.1.2 Generalised Linear Models

Previously, we introduced K-priors and their form. We now look at how K-priors can exactly reconstruct the gradient of past information on Generalised Linear Models (GLMs). We call these *vanilla* K-priors as they require storing a large memory (we consider K-priors for smaller memory settings later in this chapter in Sections 5.4 to 5.6). We start with the MAP/Laplace setting, and then expand to the variational setting. We extend to neural networks in Section 5.2, and to adaptation tasks other than the 'Add Data' task in Section 5.3.

The MAP/Laplace setting

GLMs include models such as logistic and Poisson regression, and have a linear model $f^i_{\boldsymbol{w}} = \boldsymbol{\phi}_i^\top \boldsymbol{w}$, with feature vectors $\boldsymbol{\phi}_i = \boldsymbol{\phi}(\boldsymbol{x}_i) \in \mathbb{R}^P$. In the MAP/Laplace setting, the base model is obtained as follows,

$$\boldsymbol{w}_* = \arg\min_{\boldsymbol{w}} \sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f^i_{\boldsymbol{w}})) + \mathcal{R}(\boldsymbol{w}). \tag{5.10}$$

In what follows, for simplicity, we use an L_2-regulariser $\mathcal{R}(\boldsymbol{w}) = \frac{1}{2}\delta\|\boldsymbol{w}\|^2$, with $\delta > 0$.

We will now discuss a Lap-K-prior that *exactly* reconstructs the gradients of this objective. For this vanilla Lap-K-prior, we choose $\mathbb{D}_w(\cdot\|\cdot)$ to be the Bregman divergence with $\mathcal{R}(\boldsymbol{w})$ as the Bregman function,

$$\mathbb{D}_w\left(\boldsymbol{w}\|\boldsymbol{w}_*\right) = \mathcal{B}_{\mathcal{R}}(\boldsymbol{w}\|\boldsymbol{w}_*) = \tfrac{1}{2}\delta\|\boldsymbol{w} - \boldsymbol{w}_*\|^2. \tag{5.11}$$

We set memory $\mathcal{M} = \mathcal{X}_{\text{old}}$, where $\mathcal{X}_{\text{old}}$ is the set of all inputs $\boldsymbol{x}_i$ for $i \in \mathcal{D}_{\text{old}}$. We regularise each example using separate divergences whose Bregman function is equal to the log-partition $A(f)$ (defined in Equation 5.7),

$$\mathbb{D}_f\left(\boldsymbol{f}(\boldsymbol{w})\|\boldsymbol{f}(\boldsymbol{w}_*)\right) = \sum_{i \in \mathcal{X}_{\text{old}}} \mathcal{B}_A(f^i_{\boldsymbol{w}}\|f^i_{\boldsymbol{w}_*}) = \sum_{i \in \mathcal{X}_{\text{old}}} \ell\big(h(f^i_{\boldsymbol{w}_*}), h(f^i_{\boldsymbol{w}})\big) + \text{constant}. \tag{5.12}$$

Setting $\tau = 1$, we get the following vanilla Lap-K-prior,

$$\mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{X}_{\text{old}}) = \sum_{i \in \mathcal{X}_{\text{old}}} \ell\big(h(f^i_{\boldsymbol{w}_*}), h(f^i_{\boldsymbol{w}})\big) + \tfrac{1}{2}\delta\|\boldsymbol{w} - \boldsymbol{w}_*\|^2, \tag{5.13}$$

which has a similar form to Equation 5.10, but the outputs y_i are now replaced by the predictions $h(f^i_{\boldsymbol{w}_*})$, and the base model $\boldsymbol{w}_*$ serves as the mean of a Gaussian weight-prior.

We can show that the gradient of the above Lap-K-prior is equal to that of the base model objective in Equation 5.10,

$$\nabla_{\boldsymbol{w}}\mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{X}_{\text{old}}) = \sum_{i \in \mathcal{X}_{\text{old}}} \boldsymbol{\phi}_i \left[h(f^i_{\boldsymbol{w}}) - h(f^i_{\boldsymbol{w}_*})\right] + \delta(\boldsymbol{w} - \boldsymbol{w}_*), \tag{5.14}$$

$$= \underbrace{\sum_{i \in \mathcal{D}_{\text{old}}} \boldsymbol{\phi}_i \left[h(f^i_{\boldsymbol{w}}) - y_i\right] + \delta\boldsymbol{w}}_{=\nabla_{\boldsymbol{w}}\mathcal{L}^{\text{MAP}}(\boldsymbol{w}).} - \underbrace{\sum_{i \in \mathcal{D}_{\text{old}}} \boldsymbol{\phi}_i \left[h(f^i_{\boldsymbol{w}_*}) - y_i\right] - \delta\boldsymbol{w}_*}_{=\nabla_{\boldsymbol{w}}\mathcal{L}^{\text{MAP}}(\boldsymbol{w}_*)=0.}, \tag{5.15}$$

where the first line is obtained by using Equation 5.8 and noting that $\nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} = \boldsymbol{\phi}_i$ for GLMs, and the second line is obtained by adding and subtracting outputs y_i in the first term. The second term is equal to 0 because $\boldsymbol{w}_*$ is a minimiser of Equation 5.10, and therefore $\nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w}_*) = \nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_*} = 0$. In this case, the vanilla Lap-K-prior with $\mathcal{M} = \mathcal{X}_{\text{old}}$ exactly reconstructs the gradient of the past training objective. Note that we have not used labels y_i in our K-prior.

We are able to reconstruct exact gradients because the structure of the vanilla Lap-K-prior closely follows the structure of Equation 5.10: the gradient of each term in Equation 5.10 is recovered by a corresponding divergence in the K-prior. The gradient recovery is due to the property that the gradient of a Bregman divergence is the difference between the *dual* parameters $\nabla\psi(p)$,

$$\nabla_{p_1} \mathcal{B}_\psi(p_1, p_2) = \nabla\psi(p_1) - \nabla\psi(p_2). \tag{5.16}$$

This leads to Equation 5.14. For the function-space divergence term, $h(f^i_{\boldsymbol{w}}) - h(f^i_{\boldsymbol{w}_*})$ are the differences in the (dual) expectation parameters. For the weight-space divergence term, we note that the dual space is equal to the original parameter space $\mathcal{W}$ for the L_2 regulariser, leading to $\boldsymbol{w} - \boldsymbol{w}_*$. Lastly, we find that terms cancel out by using the optimality of $\boldsymbol{w}_*$, giving us the exact gradients.

When we store all inputs in our vanilla Lap-K-prior memory $\mathcal{M} = \mathcal{X}_{\text{old}}$, training is still expensive (as expensive as retraining-from-scratch), and moreover this is not allowed in continual learning. However, we can use insights provided by our K-prior framework to reduce the memory required. We discuss reducing memory in Section 5.4.

Because the vanilla Lap-K-prior can reconstruct the gradient of $\mathcal{L}^{\text{MAP}}(\boldsymbol{w})$, we can use it to adapt instead of retraining-from-scratch. For the 'Add Data' task discussed earlier, we can use the following Lap-K-prior-regularised objective,

$$\hat{\boldsymbol{w}}_+ = \arg\min_{\boldsymbol{w}} \sum_{j \in \mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}) + \mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}). \tag{5.17}$$

Using Equation 5.15, we can easily show that this recovers the exact solution when the memory includes all past inputs. Mathematically, $\hat{\boldsymbol{w}}_+ = \boldsymbol{w}_+$ when $\mathcal{M} = \mathcal{X}_{\text{old}}$, where $\boldsymbol{w}_+$ is from Equation 5.2. In Section 5.3, we will show similar results for many more adaptation tasks, such as removing data, changing the regulariser, and changing the model class or architecture.

The variational setting

We can use similar ideas to reconstruct the gradient of past information in the variational setting. For GLMs, the base model parameters $\boldsymbol{\eta}_*$ are given by optimising Equation 5.3, where the loss $\ell_i(\boldsymbol{w}) = \ell(y_i, h(f_{\boldsymbol{w}}^i))$. For simplicity, we will assume a zero-mean Gaussian prior $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \mathbf{0}, \delta^{-1}\mathbf{I})$. Note that this prior becomes the same L_2-regulariser from the MAP/Laplace setting earlier, $\mathcal{R}(\boldsymbol{w}) = -\log p(\boldsymbol{w}) = \frac{1}{2}\delta||\boldsymbol{w}||^2 + \text{const}$. This gives us the following objective for the base model,

$$\boldsymbol{\eta}_* = \arg\min_{\boldsymbol{\eta}} \; \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f_{\boldsymbol{w}}^i)) - \log p(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w}) \right]. \tag{5.18}$$

To reconstruct the gradient of past information, we use a var-K-prior that has the same properties as the previous vanilla Lap-K-prior. Our vanilla var-K-prior is chosen as,

$$q_{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{M}) \propto \exp\left(-\tilde{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{M})\right), \tag{5.19}$$

$$\text{where } \nabla_{\boldsymbol{w}} \tilde{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{M}) = \mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})} \left[\nabla_{\boldsymbol{w}} \mathcal{K}(\boldsymbol{w}; \hat{\boldsymbol{w}}, \mathcal{M})\right]. \tag{5.20}$$

We have taken an expectation of the (gradient of the) Lap-K-prior with respect to the distribution over the base model parameters in order to effectively replace $\boldsymbol{w}_*$ with $\boldsymbol{\eta}_*$ ($\hat{\boldsymbol{w}}$ is a different parameter to $\boldsymbol{w}$). We will use often this expression to derive a var-K-prior from a Lap-K-prior while keeping the Lap-K-prior's properties (in this case, we want our vanilla var-K-prior to also reconstruct the gradient of past information). We will not need the normalising constant of $q_{\mathcal{K}}(\boldsymbol{w})$.

For the vanilla var-K-prior, our Lap-K-prior $\mathcal{K}(\boldsymbol{w})$ is from Equation 5.13 (with $\mathcal{M} = \mathcal{X}_{\text{old}}$), and we get,

$$q_{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{X}_{\text{old}}) \propto \exp\left(- \left[\sum_{i \in \mathcal{X}_{\text{old}}} \ell\left(\mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})}\left[h(f_{\hat{\boldsymbol{w}}}^i)\right], h(f_{\boldsymbol{w}}^i)\right) + \tfrac{1}{2}\delta \|\boldsymbol{w} - \boldsymbol{\mu}_*\|^2 \right] \right), \tag{5.21}$$

where we take an expectation with respect to $\hat{\boldsymbol{w}}$, a different parameter to $\boldsymbol{w}$, and $q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}}) = \mathcal{N}(\hat{\boldsymbol{w}}; \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$. Our soft label is now $\mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})}\left[h(f_{\hat{\boldsymbol{w}}}^i)\right]$, and we can use Monte-Carlo sampling to estimate it.

We now show that the gradient of the objective using the above vanilla var-K-prior is equal to that of the base model objective used in Equation 5.18, where the gradient is taken with respect to the parameters of $q_{\boldsymbol{\eta}}(\boldsymbol{w})$. As we have a Gaussian approximating family, the

parameters of $q_{\boldsymbol{\eta}}(\boldsymbol{w})$ only depend on the mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. It will suffice to show,

$$\nabla \, \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[-\log q_{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{X}_{\text{old}}) \right] = \nabla \, \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f^i_{\boldsymbol{w}})) - \log p(\boldsymbol{w}) \right], \tag{5.22}$$

where we take gradients with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. We will make repeated use of Bonnet's and Price's theorems from Equations 2.23 and 2.24 (Opper and Archambeau, 2009; Rezende et al., 2014) to show this.

We start by showing Equation 5.22 holds for the gradient with respect to the mean $\boldsymbol{\mu}$,

$$\begin{aligned}
&\nabla_{\boldsymbol{\mu}} \, \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[-\log q_{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{X}_{\text{old}}) \right] \\
&\quad = \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\nabla_{\boldsymbol{w}} \left(\sum_{i \in \mathcal{X}_{\text{old}}} \ell\left(\mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})} \left[h(f^i_{\hat{\boldsymbol{w}}}) \right], h(f^i_{\boldsymbol{w}}) \right) + \tfrac{1}{2} \delta \|\boldsymbol{w} - \boldsymbol{\mu}_*\|^2 \right) \right] && (5.23) \\
&\quad = \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{X}_{\text{old}}} \boldsymbol{\phi}_i \left[h(f^i_{\boldsymbol{w}}) - \mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})} \left[h(f^i_{\hat{\boldsymbol{w}}}) \right] \right] + \delta(\boldsymbol{w} - \boldsymbol{\mu}_*) \right] && (5.24) \\
&\quad = \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{X}_{\text{old}}} \boldsymbol{\phi}_i \left[h(f^i_{\boldsymbol{w}}) - y_i \right] + \delta \boldsymbol{w} \right] - \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \underbrace{\left[\sum_{i \in \mathcal{X}_{\text{old}}} \boldsymbol{\phi}_i \left[\mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})} \left[h(f^i_{\hat{\boldsymbol{w}}}) \right] - y_i \right] + \delta \boldsymbol{\mu}_* \right]}_{= \mathbb{E}_{q_{\boldsymbol{\eta}_*}} \left[\nabla_w \left(\sum_i \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w}) \right) \right] = 0, \text{ by Equation 5.4}} && (5.25) \\
&\quad = \nabla_{\boldsymbol{\mu}} \, \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w}) \right]. && (5.26)
\end{aligned}$$

In the first equality we used Bonnet's and Price's theorems (specifically, Equation 2.23) to move the gradient inside the expectation, and so the gradient is now with respect to $\boldsymbol{w}$. For the third equality, we added and subtracted the labels y_i and then noted that the final two terms are the fixed-point for the base model parameters $\boldsymbol{\eta}_*$ given in Equation 5.4. The final equality uses Bonnet's and Price's theorems again.

We now show that Equation 5.22 also holds for the gradient with respect to $\boldsymbol{\Sigma}$,

$$\nabla_{\boldsymbol{\Sigma}} \, \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[-\log q_{\mathcal{K}}(\boldsymbol{w}; \boldsymbol{\eta}_*, \mathcal{X}_{\text{old}}) \right]$$

$$= \tfrac{1}{2} \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\nabla^2_{\boldsymbol{w}\boldsymbol{w}} \left(\sum_{i \in \mathcal{X}_{\text{old}}} \ell\left(\mathbb{E}_{q_{\boldsymbol{\eta}_*}(\hat{\boldsymbol{w}})} \left[h(f^i_{\hat{\boldsymbol{w}}}) \right], h(f^i_{\boldsymbol{w}}) \right) + \tfrac{1}{2} \delta \|\boldsymbol{w} - \boldsymbol{\mu}_*\|^2 \right) \right] \tag{5.27}$$

$$= \tfrac{1}{2} \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{X}_{\text{old}}} \boldsymbol{\phi}_i \, h'(f^i_{\boldsymbol{w}}) \, \boldsymbol{\phi}^\top + \delta \mathbf{I} \right] \tag{5.28}$$

$$= \nabla_{\boldsymbol{\Sigma}} \, \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) - \log p(\boldsymbol{w}) \right]. \tag{5.29}$$

As before, for the first and last equalities we used Bonnet's and Price's theorems (specifically, Equation 2.24).

We have therefore shown that Equation 5.22 holds. Therefore, the gradient of the vanilla var-K-prior is the same as the gradient of the base model objective. We can use this property for adaptation, similar to what we did with the vanilla Lap-K-prior. For the 'Add Data' task, we use this vanilla var-K-prior as our prior when seeing new data (we use $q_{\mathcal{K}}(\boldsymbol{w})$ instead of $q_{\boldsymbol{\eta}_{t-1}}(\boldsymbol{w})$ in the variational objective in Equation 2.8), giving,

$$\hat{\boldsymbol{\eta}}_+ = \arg\min_{\boldsymbol{\eta}} \; \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})} \left[\sum_{j \in \mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}) - \log q_{\mathcal{K}}(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w}) \right]. \tag{5.30}$$

We can use Equation 5.22 to show that the gradient of this objective is equal to the retrained-from-scratch objective in Equation 5.18 when we optimise for the parameters of a Gaussian approximating family. Therefore, when $\mathcal{M} = \mathcal{X}_{\text{old}}$, we get $\hat{\boldsymbol{\eta}}_+ = \boldsymbol{\eta}_+$. Note that we can use natural-gradient updates to optimise these objectives, but by the chain-rule (Equations 2.19 and 2.20), these gradients only depend on the gradients with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

This concludes our section on showing how K-priors can exactly reconstruct the gradient of past information on GLMs. We showed this on the 'Add Data' task only, and in Section 5.3 we will consider other adaptation tasks. In Section 5.6 we will see that this is not the case for the FROMP algorithm from Chapter 4. In the next section, we apply vanilla K-priors to neural networks, and look at the relationship between K-priors and knowledge distillation (Hinton et al., 2015).

5.2 Neural networks and connections with knowledge distillation

We have seen how K-priors can reconstruct the exact gradient of past information on GLMs. In this section, we now apply K-priors to neural networks. Unlike with GLMs, vanilla K-priors do not reconstruct the exact gradient of past information on neural networks, even when we store all past inputs. We also see links between vanilla K-priors and knowledge distillation (Hinton et al., 2015). We analyse the similarities between K-priors and knowledge distillation, arguing that the error term in K-priors may even be beneficial, similarly to how knowledge distillation can improve a model's performance. We also use the links between K-priors and knowledge distillation to suggest ways of improving both algorithms.

We apply vanilla K-priors to neural networks using the same Bregman divergences and expressions as for GLMs (see Equations 5.11 to 5.13). Our analysis in this section will be based on Lap-K-priors, but can also be applied to var-K-priors using the relationships discussed in Section 5.1. The only difference from Section 5.1 is that the model $f_{\boldsymbol{w}}$ is no longer linear, and is instead parameterised by a neural network with parameters $\boldsymbol{w}$. The gradient of the model $\nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}}$ now depends on the weights $\boldsymbol{w}$, while in GLMs the gradient was constant (and equal to the feature vector $\boldsymbol{\phi}_i$).

The gradient of the vanilla Lap-K-prior is obtained similarly to Equation 5.15, where we add and subtract y_i in the first term in the first line below,

$$\nabla_{\boldsymbol{w}} \mathcal{K}(\boldsymbol{w}) = \sum_{i \in \mathcal{X}_{\text{old}}} \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} \left[h(f^i_{\boldsymbol{w}}) - h(f^i_{\boldsymbol{w}_*}) \right] + \delta(\boldsymbol{w} - \boldsymbol{w}_*), \tag{5.31}$$

$$= \underbrace{\sum_{i \in \mathcal{D}_{\text{old}}} \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} \left[h(f^i_{\boldsymbol{w}}) - y_i \right] + \delta \boldsymbol{w}}_{= \nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w})} - \underbrace{\sum_{i \in \mathcal{D}_{\text{old}}} \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} [h(f^i_{\boldsymbol{w}_*}) - y_i] - \delta \boldsymbol{w}_*}_{\neq \nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w}_*), \text{ because } \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} \neq \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}_*}}. \tag{5.32}$$

Unlike with GLMs, the second term is now not zero because $\nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} \neq \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}_*}$, and this term can be viewed as an error term. This error term is larger for points in the old data $\mathcal{D}_{\text{old}}$ that have high residual $r^i_{\boldsymbol{w}_*} = h(f^i_{\boldsymbol{w}_*}) - y_i$, which are the points that the base model did not classify well. This error term also becomes larger when the gradient at the new parameters $\boldsymbol{w}$ is further away from the gradient at the base model's parameters $\boldsymbol{w}_*$.

We next analyse knowledge distillation (Hinton et al., 2015), where we see a similar error term. Empirically, knowledge distillation has been shown to improve upon standard training, and this indicates that our error term may not necessarily always result in lower performance.

Knowledge distillation

Knowledge distillation (KD) (Hinton et al., 2015) is a popular approach for model compression in deep learning for classification problems using a softmax function. In the usual setup, a large teacher network is trained on data $\mathcal{D}$ to give $\boldsymbol{w}_*$ (such as by optimising Equation 5.1, often with no regulariser $\mathcal{R}(\boldsymbol{w})$). We then 'distil' the teacher's knowledge into a smaller student network $\boldsymbol{w}$ using the following objective,

$$\mathcal{L}^{\text{KD}}(\boldsymbol{w}) = \lambda \sum_{i\in\mathcal{D}} \ell\left(y_i, h(f^i_{\boldsymbol{w}})\right) + (1-\lambda) \sum_{i\in\mathcal{D}} \ell\left(h(f^i_{\boldsymbol{w}_*}/T),\, h(f^i_{\boldsymbol{w}}/T)\right). \tag{5.33}$$

Model predictions in the second term are often scaled with a temperature parameter $T > 1$,[1] and $\lambda \in [0, 1]$. KD can be seen as a special case of K-priors without the weight-space term ($\tau = 0$).

KD often yields solutions that are better than training just the student model $\boldsymbol{w}$ from scratch. This has even been found to be the case when the teacher model architecture is the same as the student model. Theoretically the reasons behind the improved performance are not understood well. However, our K-prior framework can provide some insights when we view KD as a mechanism to reconstruct past gradients. To see this, we write out the gradient of the KD objective for $T = 1$,

$$\nabla_{\boldsymbol{w}}\mathcal{L}^{\text{KD}}(\boldsymbol{w}) = \underbrace{\sum_{i\in\mathcal{D}} \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}} \left[h(f^i_{\boldsymbol{w}}) - y_i\right]}_{\text{Training student model from scratch}} \underbrace{- (1-\lambda) \sum_{i\in\mathcal{D}} \nabla_{\boldsymbol{w}} f^i_{\boldsymbol{w}}\, r^i_{\boldsymbol{w}_*}}_{\text{Additional term}}. \tag{5.34}$$

The first term is the same term as training just the student model from scratch, with no teacher model. The additional second term adds large gradients to push away from the high residual examples (the examples the teacher did not fit well), and is the same as the error term in vanilla K-priors in Equation 5.32 (with $\delta = 0$). These high-residual examples are the toughest to classify and may even include mislabelled points. This is similar to Similarity-Control for Support Vector Machines (SVMs) from Vapnik and Izmailov (2015), where "slack"-variables are used in a dual formulation to improve the student. Vapnik and Izmailov (2015) argue the improvement is because the student could now be solving a simpler, separable, classification problem.

In our case, the residuals $r^i_{\boldsymbol{w}_*}$ above play a similar role as the slack variables, but they do not require a dual formulation. Instead, they arise due to the K-prior in a primal formulation. In this sense, K-priors can be seen as an easy-to-implement scheme for Similarity Control

[1] Some works only scale the base model predictions by T, and not the student model predictions too (Lopez-Paz et al., 2016).

that could potentially be useful for student-teacher learning. We provide further comparisons to SVMs in Section 5.8.

Lopez-Paz et al. (2016) use this idea further to generalise distillation and interpret residuals from the teachers as corrections for the student (see Equation 6 in their paper). In general, it is desirable to trust the knowledge of the base model and use it to improve the adapted model. These previous ideas are now unified in K-priors: we can provide the information about the decision boundary to the student in a more accessible form than the original data (with true labels) could.

K-priors extend KD in three ways, by (i) adding the weight-space term, (ii) allowing general link functions or divergence functions, and (iii) using a potentially small number of examples in $\mathcal{M}$ instead of the whole dataset (we discuss smaller memories in Section 5.4). With these extensions, K-priors can handle adaptation tasks other than model compression. Due to their similarity, it is also possible to borrow tricks used in KD to improve the performance of K-priors. For example, we can use a temperature $T > 1$ in K-priors, and in Section 5.7 we see that this improves performance on larger architectures. We can also improve KD by using a smaller memory than the full dataset while still achieving good performance (we do this in Figure 5.4).

Some previous works use a knowledge distillation-style loss for continual learning, and are therefore closely related to vanilla K-priors. iCARL (Rebuffi et al., 2017) uses a knowledge distillation loss over *exemplars* from previous tasks, although they used a nearest-mean-of-exemplars rule for classification, and did not look at why this loss might be beneficial. Learning without Forgetting (LwF) (Li and Hoiem, 2016) uses a knowledge distillation loss but only over inputs in the current task, and they do not store any past data. Our theory with vanilla K-priors indicates that LwF will perform well when the current task is in similar regions of input-space as past tasks, and we expect LwF to perform poorly when the current task is in very different regions of input-space.

Another related continual learning approach is Gradient Episodic Memory (GEM) (Lopez-Paz and Ranzato, 2017), where the goal is to ensure that the loss over past datapoints does not worsen when training on a new task, $\sum_{i \in \mathcal{M}} \left[\ell(y_i, f^i_{\boldsymbol{w}}) - \ell(y_i, f^i_{\boldsymbol{w}_*})\right] < 0$. However, GEM enforces this through constrained optimisation, leading to a different algorithm. Instead, K-priors can be viewed as relaxing this optimisation problem, leading to more general results.

Having analysed K-priors on GLMs and neural networks for the 'Add Data' task, we now turn our attention to other adaptation tasks that are relevant in machine learning. Specifically, we will look at removing data, changing the regulariser, and changing the model class or architecture.

5.3 Many adaptation tasks in machine learning

So far, we have applied K-priors to GLMs and NNs, but have mostly limited ourselves to the 'Add Data' adaptation task, which is closely related to continual learning (we also briefly considered model compression when comparing with knowledge distillation). However, as K-priors reconstruct the gradient of past information, they can also be applied to many more adaptation tasks. In this section, we apply to three more adaptation tasks: (i) removing data, (ii) changing the regulariser, and (iii) changing the model class or architecture. We consider each of these in turn, applying Lap-K-priors and seeing how, similarly to the 'Add Data' task, vanilla K-priors can reconstruct the exact gradient of past information if we store all past inputs. These results all hold for the vanilla var-K-prior too, using exactly the same reasoning as presented in Section 5.1.2 for the 'Add Data' task. We end this section by considering K-priors for general learning problems in Section 5.3.4, where we give a more general form of K-priors. We will use this more general form to derive K-priors that can be better when we have a limited memory size in Sections 5.4 to 5.6.

Although we consider each adaptation task separately, K-priors can be straightforwardly applied to any combination of these adaptation tasks (such as adding and removing data at the same time). Such adaptation can be useful to reduce the cost of model updating in a continuously-evolving ML pipeline. For example, in k-fold cross-validation, the model is usually retrained-from-scratch for every data-fold and hyperparameter setting. Such retraining can be made cheaper and faster by reusing models trained on previous folds and adapting them for new folds and hyperparameters. Model reuse can also be useful in active learning, where we can add new examples using a quick 'Add Data' adaptation.

5.3.1 Removing old data

The 'Remove Data' task is very similar to the 'Add Data' task from Section 5.1. Having trained a base model $\boldsymbol{w}_*$ according to Equation 5.1, we now want to remove data $\mathcal{D}_{\text{rem}}$, where $\mathcal{D}_{\text{rem}} \subset \mathcal{D}_{\text{old}}$. Our new ideal objective for $\boldsymbol{w}_-$ is,

$$\begin{aligned}\boldsymbol{w}_- &= \arg\min_{\boldsymbol{w}} \sum_{i \in \{\mathcal{D}_{\text{old}} \setminus \mathcal{D}_{\text{rem}}\}} \ell_i(\boldsymbol{w}) + \mathcal{R}(\boldsymbol{w}) \\ &= \arg\min_{\boldsymbol{w}} -\sum_{k \in \mathcal{D}_{\text{rem}}} \ell_k(\boldsymbol{w}) + \sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) + \mathcal{R}(\boldsymbol{w}). \end{aligned} \tag{5.35}$$

We can use the following objective with our vanilla Lap-K-prior from Equation 5.13,

$$\hat{\boldsymbol{w}}_{-} = \arg\min_{\boldsymbol{w}} \; -\sum_{k \in \mathcal{D}_{\text{rem}}} \ell_k(\boldsymbol{w}) + \mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}). \tag{5.36}$$

When we use all past inputs $\mathcal{M} = \mathcal{X}_{\text{old}}$, the gradients of Equations 5.35 and 5.36 are the same, just like in the 'Add Data' case. We therefore recover the exact solution, $\hat{\boldsymbol{w}}_{-} = \boldsymbol{w}_{-}$.

5.3.2 Changing regulariser

In the 'Change Regulariser' task, we change the regulariser $\mathcal{R}(\boldsymbol{w})$ in Equation 5.1 to a new regulariser $\mathcal{G}(\boldsymbol{w})$. Specifically, our new objective is,

$$\boldsymbol{w}_{\mathcal{G}} = \arg\min_{\boldsymbol{w}} \; \sum_{i \in \mathcal{D}_{\text{old}}} \ell_i(\boldsymbol{w}) + \mathcal{G}(\boldsymbol{w}). \tag{5.37}$$

For this 'Change Regulariser' task, we slightly modify our vanilla K-prior. We replace the weight-space divergence in Equation 5.13 with a Bregman divergence defined using two different regularisers (see Proposition 5 in Nielsen (2021)),

$$\mathcal{B}_{\mathcal{G}\mathcal{R}}(\boldsymbol{w} \| \boldsymbol{w}_*) = \mathcal{G}(\boldsymbol{w}) + \mathcal{R}^*(\boldsymbol{\eta}_*) - \boldsymbol{w}^\top \boldsymbol{\eta}_*, \tag{5.38}$$

where $\boldsymbol{\eta}_* = \nabla_{\boldsymbol{w}} \mathcal{R}(\boldsymbol{w}_*)$ is the dual parameter and $\mathcal{R}^*$ is the convex-conjugate of $\mathcal{R}$. This is similar to the standard Bregman divergence but uses two different convex functions.

To get an intuition, consider tuning the hyperparameter in the L_2-regulariser $\mathcal{R}(\boldsymbol{w}) = \frac{1}{2}\delta\|\boldsymbol{w}\|^2$, where our new regulariser $\mathcal{G}(\boldsymbol{w}) = \frac{1}{2}\gamma\|\boldsymbol{w}\|^2$ uses a hyperparameter $\gamma \neq \delta$. Since the conjugate $\mathcal{R}^*(\boldsymbol{\eta}) = \frac{1}{2}\|\boldsymbol{\eta}\|^2/\delta$ and $\boldsymbol{\eta}_* = \nabla_{\boldsymbol{w}} \mathcal{R}(\boldsymbol{w}_*) = \delta \boldsymbol{w}_*$, we get,

$$\mathcal{B}_{\mathcal{G}\mathcal{R}}(\boldsymbol{w} \| \boldsymbol{w}_*) = \tfrac{1}{2}(\gamma\|\boldsymbol{w}\|^2 + \delta\|\boldsymbol{w}_*\|^2 - 2\delta \boldsymbol{w}^\top \boldsymbol{w}_*). \tag{5.39}$$

When $\gamma = \delta$, this reduces to the divergence used in Equation 5.13. We define the following K-prior, and use it to obtain $\hat{\boldsymbol{w}}_{\mathcal{G}}$,

$$\mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}) = \sum_{i \in \mathcal{M}} \ell\left(h(f^i_{\boldsymbol{w}_*}), h(f^i_{\boldsymbol{w}})\right) + \mathcal{B}_{\mathcal{G}\mathcal{R}}(\boldsymbol{w} \| \boldsymbol{w}_*), \tag{5.40}$$

$$\hat{\boldsymbol{w}}_{\mathcal{G}} = \arg\min_{\boldsymbol{w} \in \mathcal{W}} \; \mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}). \tag{5.41}$$

For $\mathcal{M} = \mathcal{X}_{\text{old}}$ and strictly-convex regularisers, we have $\hat{\boldsymbol{w}}_{\mathcal{G}} = \boldsymbol{w}_{\mathcal{G}}$. The derivation of this result is very similar to Equation 5.15, where $\delta(\boldsymbol{w} - \boldsymbol{w}_*)$ is replaced by $\nabla_{\boldsymbol{w}} \mathcal{G}(\boldsymbol{w}) - \nabla_{\boldsymbol{w}} \mathcal{R}(\boldsymbol{w}_*)$.

5.3.3 Changing model class or architecture

In the 'Change Model Class/Architecture' task, the model class or architecture is changed, with the following objective (compare with Equation 5.1),

$$\boldsymbol{\theta}_* = \arg\min_{\boldsymbol{\theta}} \sum_{i \in \mathcal{D}_{\text{old}}} \tilde{\ell}_i(\boldsymbol{\theta}) + \tilde{\mathcal{R}}(\boldsymbol{\theta}), \tag{5.42}$$

where we assume the loss $\tilde{\ell}_i(\boldsymbol{\theta})$ has the same form as $\ell(\boldsymbol{w})$ but using the new model $\tilde{f}_{\boldsymbol{\theta}}(\boldsymbol{x})$ for prediction, with $\boldsymbol{\theta}$ as the new parameter. $\boldsymbol{\theta}$ can be of a different dimension to $\boldsymbol{w}$, and the new regulariser $\tilde{\mathcal{R}}(\boldsymbol{\theta})$ can be chosen accordingly.

K-priors can also be applied to this adaptation task, and we discuss this through an example. Suppose we want to remove the last feature from $\boldsymbol{\phi}_i$ so that $\boldsymbol{w} \in \mathbb{R}^P$ is replaced by a smaller weight-vector $\boldsymbol{\theta} \in \mathbb{R}^{P-1}$. Assuming no change in the regularisation hyperparameter, we can simply use a weighting matrix to 'kill' the last element of $\boldsymbol{w}_*$. We define the matrix $\boldsymbol{A} = \mathbf{I}_{P-1 \times P}$ whose last column is $\mathbf{0}$ and the rest is the identity matrix of size $P-1$. With this, we can use the following training procedure,

$$\begin{aligned} \mathcal{K}(\boldsymbol{\theta}) &= \sum_{i \in \mathcal{M}} \ell\big(h(f^i_{\boldsymbol{w}_*}), h(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i))\big) + \mathcal{B}_{\mathcal{R}}(\boldsymbol{\theta} \| \boldsymbol{A}\boldsymbol{w}_*), \\ \hat{\boldsymbol{\theta}}_* &= \arg\min_{\boldsymbol{\theta}} \ \mathcal{K}(\boldsymbol{\theta}). \end{aligned} \tag{5.43}$$

If the hyperparameters or regulariser are different for the new problem, then the Bregman divergence shown in Equation 5.38 can be used, with an appropriate weighting matrix.

Model compression is a specific instance of the 'Change Model Class' task, where the architecture is entirely changed. For neural networks, this also changes the meaning of the weights and the regularisation term may not make sense. In such cases, we can simply use the functional-divergence term in K-priors,

$$\mathcal{K}(\boldsymbol{\theta}) = \sum_{i \in \mathcal{M}} \ell\big(h(f^i_{\boldsymbol{w}_*}), h(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i))\big)\,, \qquad \hat{\boldsymbol{\theta}}_* = \arg\min_{\boldsymbol{\theta}} \ \mathcal{K}(\boldsymbol{\theta}). \tag{5.44}$$

This is equivalent to knowledge distillation (KD) in Equation 5.33 with $\lambda = 0$, $T = 1$ and $\mathcal{M} = \mathcal{X}_{\text{old}}$. Since KD performs well in practice, it is possible to use a similar strategy to boost K-priors. For example, we can define the following,

$$\hat{\boldsymbol{\theta}}_* = \arg\min_{\boldsymbol{\theta}} \ \lambda \sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f^i_{\boldsymbol{\theta}})) + (1-\lambda)\,\mathcal{K}(\boldsymbol{\theta}), \tag{5.45}$$

where we use a trade-off term λ. We can also use a temperature $T > 1$ in the K-priors term.

5.3.4 K-priors for general learning problems

We now consider K-priors for more general learning problems. The main principle behind the design of K-priors is to construct it such that gradients can faithfully be reconstructed. As discussed earlier, this is often possible by exploiting the structure of the learning problem. For example, to replace an old objective such as Equation 5.1, with loss $\ell_i^{\text{old}}(f)$ and regulariser $\mathcal{R}^{\text{old}}(\boldsymbol{w})$, with a new objective with loss $\ell_i^{\text{new}}(f)$ and regulariser $\mathcal{R}^{\text{new}}(\boldsymbol{w})$, the divergences should be chosen such that they have the following gradients,

$$\nabla_{\boldsymbol{w}} \mathbb{D}_w(\boldsymbol{w}\|\boldsymbol{w}_*) = \nabla_{\boldsymbol{w}}\mathcal{R}^{\text{new}}(\boldsymbol{w}) - \nabla_{\boldsymbol{w}}\mathcal{R}^{\text{old}}(\boldsymbol{w}), \tag{5.46}$$

$$\nabla_{\boldsymbol{w}} \mathbb{D}_f(\boldsymbol{f}(\boldsymbol{w})\|\boldsymbol{f}(\boldsymbol{w}_*)) = [\nabla_{\boldsymbol{w}}\boldsymbol{f}(\boldsymbol{w})]^\top \mathbf{B}\, \boldsymbol{d}_u, \tag{5.47}$$

where $\boldsymbol{d}_u$ is an M-length vector with the discrepancy $\nabla_f \ell_m^{\text{new}}(f_{\boldsymbol{w}}) - \nabla_f \ell_m^{\text{old}}(f_{\boldsymbol{w}_*})$ as the m'th entry, for $m \in \mathcal{M}$. The matrix $\mathbf{B}$ is added to counter the mismatch between $\mathcal{D}_{\text{old}}$ and $\mathcal{M}$. For vanilla K-priors on GLMs with $\mathcal{M} = \mathcal{X}_{\text{old}}$, $\mathbf{B} = \mathbf{I}$ is the identity matrix, $\nabla_{\boldsymbol{w}}\boldsymbol{f}(\boldsymbol{w})$ gives features (each feature $\boldsymbol{\phi}_i$ is a row in this matrix), and when we have the same loss function $\ell_i^{\text{new}}(f) = \ell_i^{\text{old}}(f) = \ell(y_i, h(f))$, the elements of $\boldsymbol{d}_u$ are given by $h(f_{\boldsymbol{w}}^m) - h(f_{\boldsymbol{w}_*}^m)$.

Similar constructions to Equation 5.47 can be used for other learning objectives. For non-differentiable functions, the variational version can be used with the Kullback-Leibler (KL) divergence. We can use exponential-family distributions which implies a Bregman divergence through the KL divergence (Banerjee et al., 2005). Information in the dual-space is automatically computed using natural-gradients (Khan and Rue, 2021).

Since the gradient of such divergences is equal to the difference in the dual-parameters, the general principle in K-priors is to use divergences with an appropriate dual space to swap old information with new information. In fact, we may be interested in retaining information that is not simply first-order information. Although we have focussed so far on reconstructing the first-order derivative of past information, we may instead be interested in reconstructing the Hessian (or higher order derivatives) of the past.

Such considerations become important when we store a limited memory, and we design K-priors to preserve information. We discuss this in the next section. We will see optimal K-priors, which preserve first-order information using a matrix $\mathbf{B}$ as in Equation 5.47. We will also compare with FROMP (from Chapter 4) in Section 5.6. We will see that FROMP can be seen as having a specific $\mathbf{B}$, and that FROMP attempts to preserve second-order information using it.

5.4 Limited memory in K-priors

So far, we have only considered the case where the memory in K-priors is all past inputs, $\mathcal{M} = \mathcal{X}_{\text{old}}$. In this section, we consider how to use a limited memory in K-priors while still achieving good performance. This is because, in practice, we may not need all past data (an illustrative example is shown on the right of Figure 5.1). Limited memory is enforced in continual learning, where we are not allowed to store all past data. This is also necessary to reduce the computation cost compared to retraining-from-scratch. As before, we will present results for Lap-K-priors, but these can be extended to var-K-priors.

We start with discussing the *optimal* K-prior, which can theoretically obtain perfect performance by using singular vectors. This is possible because the memory in K-priors can lie anywhere in input space as K-priors do not need the true labels y. However, the optimal K-prior is difficult to realise in practice, and we also consider the simpler case of storing a subset of past data. We will see a link to the theory from Section 4.3, where we chose memorable past examples in FROMP.

The optimal K-prior

We now present theoretical results to show that K-priors with limited memory can achieve low gradient-reconstruction error on GLMs. We will discuss the *optimal* K-prior which can theoretically achieve perfect reconstruction error. This prior is difficult to realise in practice since it requires all past training-data inputs $\mathcal{X}_{\text{old}}$. Our goal here is to establish a theoretical limit, not to give practical choices.

Our key idea is to choose a few input locations that provide a good representation of the N_{old} training-data inputs in $\mathcal{X}_{\text{old}}$. We will make use of the singular-value decomposition (SVD) of the feature matrix $\boldsymbol{\Phi}$, which is an $N_{\text{old}} \times P$ matrix concatenating the N_{old} input features $\boldsymbol{\phi}_i$,

$$\boldsymbol{\Phi}^\top = \boldsymbol{U}^*_{1:K} \boldsymbol{S}^*_{1:K} (\boldsymbol{V}^*_{1:K})^\top, \tag{5.48}$$

where $K \leq \min(N_{\text{old}}, P)$ is the rank, $\boldsymbol{U}^*_{1:K}$ is $P \times K$ matrix of left-singular vectors $\boldsymbol{u}^*_i$, $\boldsymbol{V}^*_{1:K}$ is $N_{\text{old}} \times K$ matrix of right-singular vectors $\boldsymbol{v}^*_i$, and $\boldsymbol{S}^*_{1:K}$ is a diagonal matrix with singular values s_i as the i'th diagonal entry.

We define $\mathcal{M}^* = \{\boldsymbol{u}^*_1, \boldsymbol{u}^*_2, \ldots, \boldsymbol{u}^*_K\}$, and the following K-prior,

$$\mathcal{K}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}^*) = \sum_{j=1}^{K} \beta^*_j \, \ell\big(h(f_{\boldsymbol{w}_*}(\boldsymbol{u}^*_j)), h(f_{\boldsymbol{w}}(\boldsymbol{u}^*_j))\big) + \tfrac{1}{2}\delta \|\boldsymbol{w} - \boldsymbol{w}_*\|^2. \tag{5.49}$$

Here, each functional divergence is weighted by β_j^*, which refers to the elements of

$$\boldsymbol{\beta}^* = \boldsymbol{D}_u^{-1} \boldsymbol{S}_{1:K}^* \boldsymbol{V}_{1:K}^\top \boldsymbol{d}_x, \tag{5.50}$$

where $\boldsymbol{d}_x$ is an N_{old}-length vector with entries $h(f_{\boldsymbol{w}}^i) - h(f_{\boldsymbol{w}_*}^i)$ for all $i \in \mathcal{X}_{\text{old}}$, while $\boldsymbol{D}_u$ is a $K \times K$ diagonal matrix with diagonal entries $h(f_{\boldsymbol{w}}(\boldsymbol{u}_j^*)) - h(f_{\boldsymbol{w}_*}(\boldsymbol{u}_j^*))$ for all $j = 1, 2, \ldots, K$. The above definition uses the matrix $\mathbf{B}$ from Section 5.3.4 (in this case, $\mathbf{B}$ is a $K \times K$ diagonal matrix with diagonal entries given by $\boldsymbol{\beta}^*$). The weights β_j^* depend on $\mathcal{X}_{\text{old}}$, so it is difficult to compute them in practice when the memory is limited. However, it might be possible to estimate them for some problems.

Nevertheless, with β_j^*, the above K-prior can achieve perfect reconstruction. The proof is very similar to the one given in Equations 5.14 and 5.15, and is shown below,

$$\begin{aligned}
\nabla_{\boldsymbol{w}} \mathcal{K}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}^*) &= \sum_{j=1}^{K} \beta_j^* \boldsymbol{u}_j^* \left[h(f_{\boldsymbol{w}}(\boldsymbol{u}_j^*)) - h(f_{\boldsymbol{w}_*}(\boldsymbol{u}_j^*)) \right] + \delta(\boldsymbol{w} - \boldsymbol{w}_*) \\
&= \boldsymbol{U}_{1:K}^* \boldsymbol{D}_u \boldsymbol{\beta}_* + \delta(\boldsymbol{w} - \boldsymbol{w}_*) \\
&= \boldsymbol{U}_{1:K}^* \boldsymbol{S}_{1:K}^* \boldsymbol{V}_{1:K}^\top \boldsymbol{d}_x + \delta(\boldsymbol{w} - \boldsymbol{w}_*) \\
&= \boldsymbol{\Phi}^\top \boldsymbol{d}_x + \delta(\boldsymbol{w} - \boldsymbol{w}_*) \\
&= \nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w}).
\end{aligned} \tag{5.51}$$

The first line is simply the gradient, which is then rearranged in the matrix-vector product in the second line. The third line uses the definition of $\boldsymbol{\beta}_*$, and the fourth line uses the SVD of $\boldsymbol{\Phi}$. This is the same as vanilla K-priors in Equation 5.14, and the rest follows as before.

Due to their perfect gradient-reconstruction property, we call the prior in Equation 5.49 the *optimal* K-prior. In Appendix D.1.1 we derive a very similar form when we optimise $\mathbf{B}$ for reconstructing first-order information when given a fixed memory that we can not choose.

When only the top-M singular vectors are chosen, the gradient reconstruction error grows according to the leftover singular values. We show this below where we have chosen $\mathcal{M}_M^* = \{\boldsymbol{u}_1^*, \boldsymbol{u}_2^*, \ldots, \boldsymbol{u}_M^*\}$ as the set of top-M singular vectors,

$$\begin{aligned}
\boldsymbol{e}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}, \mathcal{M}_M^*) &= \nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w}) - \nabla_{\boldsymbol{w}} \mathcal{K}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}_M^*) \\
&= \nabla_{\boldsymbol{w}} \mathcal{K}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}^*) - \nabla_{\boldsymbol{w}} \mathcal{K}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}_M^*) \\
&= \sum_{j=M+1}^{K} \beta_j^* \boldsymbol{u}_j^* \left[h(f_{\boldsymbol{w}}(\boldsymbol{u}_j^*)) - h(f_{\boldsymbol{w}_*}(\boldsymbol{u}_j^*)) \right] \\
&= \boldsymbol{U}_{M+1:K}^* \, \boldsymbol{S}_{M+1:K}^* \, \boldsymbol{V}_{M+1:K}^\top \, \boldsymbol{d}_x.
\end{aligned} \tag{5.52}$$

The first line is simply the definition of the error, and in the second line we use the result in Equation 5.51 for the optimal K-prior with memory $\mathcal{M}^*$. The next few lines use the definition of the optimal K-prior and rearrange terms. This gives us the following error,

$$\|\boldsymbol{e}^{\text{opt}}(\boldsymbol{w}; \boldsymbol{w}, \mathcal{M}_M^*)\| = \sqrt{\Sigma_{j=M+1}^K s_j^2 (a_j^x)^2}, \tag{5.53}$$

where a_j^x is the j'th entry of a vector $\boldsymbol{a} = \boldsymbol{V}_{1:K}^\top \boldsymbol{d}_x$. This error depends on the leftover singular values. The error is likely to be the optimal error achievable by any memory of size M, and establishes a theoretical bound on the best possible performance achievable by any K-prior.

Subset of past data

We previously looked at an optimal K-prior that can reconstruct the exact gradient of past information, but is difficult to realise in practice. We now look at how to choose the most important datapoints from past data. We choose inputs to store by looking at the gradient-reconstruction error $\boldsymbol{e}$ for vanilla K-priors, shown below for $\mathcal{M} \subset \mathcal{X}_{\text{old}}$,

$$\boldsymbol{e}(\boldsymbol{w}; \mathcal{M}) = \nabla_{\boldsymbol{w}} \mathcal{L}^{\text{MAP}}(\boldsymbol{w}) - \nabla_{\boldsymbol{w}} \mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}) = \sum_{i \in \mathcal{X}_{\text{old}} \setminus \mathcal{M}} \boldsymbol{\phi}_i \left[h(f_{\boldsymbol{w}}^i) - h(f_{\boldsymbol{w}_*}^i) \right]. \tag{5.54}$$

The error depends on the "leftover" $\boldsymbol{\phi}_i$ for $i \in \mathcal{X}_{\text{old}} \setminus \mathcal{M}$, and their discrepancies $h(f_{\boldsymbol{w}}^i) - h(f_{\boldsymbol{w}_*}^i)$. A simple idea could be to include the inputs where predictions disagree the most, but this is not feasible because the candidates $\boldsymbol{w}$ are not known beforehand. Instead, we can use a first-order approximation,

$$h(f_{\boldsymbol{w}}^i) \approx h(f_{\boldsymbol{w}_*}^i) + h'(f_{\boldsymbol{w}_*}^i) \boldsymbol{\phi}_i^\top (\boldsymbol{w} - \boldsymbol{w}_*). \tag{5.55}$$

When we apply this to Equation 5.54, we get,

$$\boldsymbol{e}(\boldsymbol{w}; \mathcal{M}) \approx \sum_{i \in \mathcal{X}_{\text{old}} \setminus \mathcal{M}} \left[\boldsymbol{\phi}_i \, h'(f_{\boldsymbol{w}_*}^i) \, \boldsymbol{\phi}_i^\top \right] (\boldsymbol{w} - \boldsymbol{w}_*) = \boldsymbol{G}_*(\mathcal{X}_{\text{old}} \setminus \mathcal{M}) \, (\boldsymbol{w} - \boldsymbol{w}_*), \tag{5.56}$$

$$\text{where } \boldsymbol{G}_*(\mathcal{X}_{\text{old}} \setminus \mathcal{M}) = \sum_{i \in \mathcal{X}_{\text{old}} \setminus \mathcal{M}} \boldsymbol{\phi}_i \, h'(f_{\boldsymbol{w}_*}^i) \, \boldsymbol{\phi}_i^\top. \tag{5.57}$$

The approximation is conveniently expressed in terms of the Generalised Gauss-Newton (GGN) matrix (Schraudolph, 2002; Graves, 2011; Martens, 2020), denoted by $\boldsymbol{G}_*(\cdot)$ (see also Section 2.2.2 for details on the GGN matrix). The approximation suggests that $\mathcal{M}$ should be chosen to keep the *leftover* GGN matrix $\boldsymbol{G}_*(\mathcal{X}_{\text{old}} \setminus \mathcal{M})$ orthogonal to $\boldsymbol{w} - \boldsymbol{w}_*$. Since $\boldsymbol{w}$ changes during training, a reasonable approximation is to choose examples that keep the top

eigenvalues of the GGN matrix. This can be done by forming a low-rank approximation by using sketching methods, such as the leverage score (Cook, 1977; Alaoui and Mahoney, 2015; Cohen et al., 2015; Calandriello et al., 2019). This leads to the Leverage method that we introduced in Section 4.3.

A cheaper alternative is to choose the examples with highest $h'(f^i_{\boldsymbol{w}_*})$. The quantity is cheap to compute in general. For example, in deep networks it is obtained with just a forward pass. This was the Lambda method in Section 4.3, and we found in Section 4.5 that it worked well for our classification experiments in continual learning. Due to its simplicity, we will use this method in most of our experiments in Section 5.7.

Rehearsal-based approaches to continual learning replay a subset of memory, and we discussed some of these in Sections 2.3 and 4.1. However, inputs are often randomly sampled from past data (Lopez-Paz and Ranzato, 2017; Benjamin et al., 2019; Buzzega et al., 2020), and this can be improved. More recent works look at more expensive methods, such as maximising the diversity of inputs (Aljundi et al., 2019b), which can be seen as related to the leverage score. Other work (Aljundi et al., 2019a) chooses which examples to replay (from a larger set of stored memory), choosing examples that maximally interfere with current examples. Similarly, Chaudhry et al. (2021) use a concept of hindsight, but they use this to learn pseudo-input anchor points that lie close to the decision boundary. Titsias et al. (2020) use inducing inputs, which is closely connected to the online GP update. Another line of work summarises a large dataset into a small synthetic dataset, referred to as Dataset Distillation (Wang et al., 2018) or Dataset Condensation (Zhao et al., 2021), however they train for their synthetic data by optimising a more complicated objective function. The method we propose in this section does not contradict with these, but gives a more direct way to choose points where gradient errors are taken into consideration.

Rather than using vanilla K-priors with a subset of memory, we can also design K-priors that better use the memory available to us. For example, we can change the function-space divergence by designing specific matrices $\mathbf{B}$ as discussed in Section 5.3.4. We look at doing this in Appendix D.1. In Appendix D.1.1 we derive a K-prior that optimally maintains first-order gradient information from the base model (Equation 5.1), and find that the form for this K-prior's $\mathbf{B}^{*,\text{1ord}}$ is very similar to the optimal K-prior discussed earlier. In Appendix D.1.2 we look at how we might maintain second-order information. The form of $\mathbf{B}^{*,\text{2ord}}$ is very similar to what we do in FROMP (Chapter 4), and we discuss this further in Section 5.6.

We can also design the weight-space divergence in K-priors to maintain information. In the next section, we look at how we can do this by using links to weight-priors, and this leads to a new K-prior. In general, designing the divergences in K-priors to better maintain information is an interesting research direction.

5.5 Improving weight-priors with function-regularisation

In this section, we explore the link between quadratic (or Gaussian) weight-priors and K-priors, and derive new K-priors that can be viewed as improving weight-priors by adding functional regularisation. We call these 'Quadratic K-priors' as they use additional quadratic regularisation when compared with vanilla K-priors from Section 5.1.

We start by considering how weight-priors can be seen as specialised cases of K-priors. This will be related to the limited-memory discussion in Section 5.4. There are two ways we can view weight-priors: (i) as a restrictive (first-order) approximation to the function-space term, or (ii) as only a weight-divergence term where the Bregman divergence is a (squared) Mahalanobis distance (Mahalanobis, 1936). We will use this intuition to design Quadratic K-priors, where we combine vanilla K-priors with weight-priors in order to improve both algorithms. We can apply Quadratic K-priors to both GLMs and neural networks, although with neural networks there is an additional term (see discussion in Section 5.2). We start by focussing on Lap-K-priors, but also derive the Quadratic var-K-prior using the relationship discussed in Section 5.1.

Weight-priors. Quadratic weight-priors are often used in online and continual learning (Kirkpatrick et al., 2017; Nguyen et al., 2018; Schwarz et al., 2018; Ritter et al., 2018) (see also Chapter 3). In the Laplace case, they take the form,

$$\mathcal{R}_{\text{weight}}(\boldsymbol{w}; \boldsymbol{w}_*) = \tfrac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_*)^\top \left[\boldsymbol{G}_*(\mathcal{X}_{\text{old}}) + \delta \mathbf{I}\right](\boldsymbol{w} - \boldsymbol{w}_*), \tag{5.58}$$

where we have used the GGN matrix from Equation 5.57. These can be seen as a first-order approximation to the vanilla Lap-K-prior in Equation 5.13. This follows by approximating the K-prior gradient in Equation 5.13 with the first-order approximation used in Equation 5.56,

$$\begin{aligned}\nabla_{\boldsymbol{w}} \mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{X}_{\text{old}}) &\approx \sum_{i \in \mathcal{X}_{\text{old}}} \boldsymbol{\phi}_i \left[h'(f^i_{\boldsymbol{w}_*})\, \boldsymbol{\phi}_i^\top (\boldsymbol{w} - \boldsymbol{w}_*)\right] + \delta(\boldsymbol{w} - \boldsymbol{w}_*) \\ &= \nabla_{\boldsymbol{w}} \mathcal{R}_{\text{weight}}(\boldsymbol{w}; \boldsymbol{w}_*). \end{aligned} \tag{5.59}$$

A similar expression holds for weight-priors in the variational setting (such as VCL (Section 2.2.1)), where they are also a first-order approximation of the var-K-prior. The key difference in the variational setting is that we take an expectation of the GGN matrix with respect to the base model $q_{\boldsymbol{\eta}_*}(\boldsymbol{w})$.

In this way, weight-priors can be seen as using the old $h'(f^i_{\boldsymbol{w}_*})$ inside the GGN when optimising, while K-priors correct this by using the most recent $h'(f^i_{\boldsymbol{w}})$, leading to being

more accurate (see Figure 5.5(a) for more intuition). Vanilla K-priors require storing points in memory to achieve this. However, we expect that we do not need to store all past points in memory to perform well, as the memory requirements should grow according to the rank of the feature matrix (see discussion on the optimal K-prior in Section 5.4), which may still be manageable. If not, we can apply sketching methods, as discussed in Section 5.4.

We next consider how we can combine vanilla K-priors with weight-priors to give Quadratic K-priors. These use a slightly different formulation to vanilla K-priors as they attempt to correct for inputs not stored in memory.

Quadratic K-priors

We now combine vanilla K-priors from Equation 5.13 with weight-priors, leading to an algorithm that reduces error for both. We end up with a slightly different K-prior, which has a modified weight-divergence. We call this the 'Quadratic K-prior', and give the form for both the Quadratic Lap-K-prior and Quadratic var-K-prior.

Quadratic K-priors use the intuition and results we have built so far in this section and in Section 5.4. When applied to the setting where there are many adaptation tasks in sequence, Quadratic K-priors are related to Online EWC (Schwarz et al., 2018) and VCL/VOGN (Chapter 3), and can be viewed as improving these weight-prior approaches by using some functional regularisation. This is particularly obvious in continual learning, where there are many 'Add Data' tasks in sequence.

Quadratic K-priors reduce the error from just storing a subset of past points $\mathcal{M} \subset \mathcal{X}_{\text{old}}$ when compared with vanilla K-priors. When we choose a subset of points to store in memory for vanilla K-priors, we have an error term given by Equation 5.54, and we can approximate this using a first-order approximation as in Equation 5.56. The key idea in Quadratic K-priors is to add this approximated error term to the vanilla K-prior. This does not require storing any more points in memory, but requires calculating and storing the GGN matrix over points not stored, $\boldsymbol{G}_*(\mathcal{X}_{\text{old}} \backslash \mathcal{M})$. This is also closely related to weight-priors as the same first-order approximation leads to weight-priors from K-priors, as shown in Equation 5.59.

In the MAP/Laplace setting, this leads to combining the vanilla Lap-K-prior from Equation 5.13 with weight-regularisation over the points that are not in memory,

$$\mathcal{K}^{\text{quad}}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}) = \mathcal{K}(\boldsymbol{w}; \boldsymbol{w}_*, \mathcal{M}) + \tfrac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_*)^\top \left[\boldsymbol{G}_*(\mathcal{X}_{\text{old}} \backslash \mathcal{M})\right](\boldsymbol{w} - \boldsymbol{w}_*) \tag{5.60}$$

$$= \underbrace{\sum_{i \in \mathcal{M}} \ell\big(h(f^i_{\boldsymbol{w}_*}), h(f^i_{\boldsymbol{w}})\big)}_{\text{Function-divergence over } \mathcal{M}} + \underbrace{\tfrac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_*)^\top \left[\boldsymbol{G}_*(\mathcal{X}_{\text{old}} \backslash \mathcal{M}) + \delta \mathbf{I}\right](\boldsymbol{w} - \boldsymbol{w}_*)}_{\text{Weight-divergence term, using } \mathcal{X}_{\text{old}} \backslash \mathcal{M}}, \tag{5.61}$$

where the second line expands out the vanilla K-priors term and combines the weight-regularisation terms. We call this the Quadratic Lap-K-prior. The weight-divergence term in the Quadratic Lap-K-prior uses a different weight-divergence than previously used in the vanilla K-prior in Equation 5.11. Specifically, we now use the (squared) Mahalanobis distance (Mahalanobis, 1936), which is also a Bregman divergence but defined with the convex function $\frac{1}{2}\boldsymbol{w}^\top \left(\boldsymbol{G}_*(\mathcal{X}_{\text{old}} \backslash \mathcal{M}) + \delta \mathbf{I}\right) \boldsymbol{w}$ instead of $\frac{1}{2}\delta \boldsymbol{w}^\top \boldsymbol{w}$.

To get some intuition about Quadratic K-priors, we can view Equation 5.61 as improving upon both the vanilla K-prior and weight-priors:

1. When $\mathcal{M} = \mathcal{X}_{\text{old}}$, we recover the vanilla K-prior, which we showed reconstructs the exact gradient of past information. When the memory is a subset of past inputs $\mathcal{M} \subset \mathcal{X}_{\text{old}}$, then the additional weight-divergence term corrects for the vanilla K-prior error using a first-order approximation as in Equation 5.56. This therefore improves upon vanilla K-priors.

2. When $\mathcal{M} = \emptyset$ is the empty set, we recover weight-priors from Equation 5.58. By including some inputs in $\mathcal{M}$, we are improving predictions at these inputs. If we pick the inputs that are the most important, we may only have to pick a few examples in order to correct for the error that weight-priors induce.

We can use Quadratic K-priors on all of our adaptation tasks instead of just vanilla K-priors. When we use the Quadratic Lap-K-prior in continual learning with many sequential tasks, we update the weight-divergence term after every task, and this is similar to weight-prior algorithms for continual learning like Online EWC (Kirkpatrick et al., 2017; Schwarz et al., 2018). We can update the GGN matrix in an online way, just like in Online EWC. We can also use a factorised GGN matrix to reduce computation cost (such as a diagonal or block-diagonal matrix). In this way, the Quadratic Lap-K-prior can be seen as combining the vanilla Lap-K-prior with Online EWC. We provide an experiment with this Quadratic K-prior in Section 5.7, where we see it performs very well in continual learning, improving upon both vanilla K-priors and weight-priors.

The Quadratic var-K-prior. We can also write Quadratic K-priors in the variational setting, where we will see links to weight-priors in variational inference (such as Variational Continual Learning (Chapter 3)). We derive the Quadratic var-K-prior from the Quadratic Lap-K-prior using Equations 5.19 and 5.20. We have $q_{\mathcal{K}}^{\text{quad}}(\boldsymbol{w}) \propto \exp\left(-\tilde{\mathcal{K}}^{\text{quad}}(\boldsymbol{w})\right)$, where $\tilde{\mathcal{K}}^{\text{quad}}(\boldsymbol{w})$ is similar to $\mathcal{K}^{\text{quad}}(\boldsymbol{w})$ from Equation 5.61 except with (i) an expectation of the soft labels in the function-divergence term $\mathbb{E}_{q_{\boldsymbol{\eta}_*}}\left[h(f_{\boldsymbol{w}}^i)\right]$, (ii) an expectation of the GGN matrix, $\tilde{\boldsymbol{G}}_*(\cdot) = \mathbb{E}_{q_{\boldsymbol{\eta}_*}}\left[\boldsymbol{G}(\cdot)\right]$, and (iii) $\boldsymbol{\mu}_*$ instead of $\boldsymbol{w}_*$ in the quadratic weight-divergence

term. Using the 'Add Data' task as an example, this gives us the following loss with the Quadratic var-K-prior,

$$\mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[\sum_{j\in\mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}) - \log q_{\mathcal{K}}^{\text{quad}}(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w})\right]$$
$$= \mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[\sum_{j\in\mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}) + \sum_{i\in\mathcal{M}} \ell\big(h(f^i_{\boldsymbol{w}_*}), h(f^i_{\boldsymbol{w}})\big) - \log q_{\text{weight}}(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w})\right], \tag{5.62}$$

where we have $q_{\text{weight}}(\boldsymbol{w}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w}-\boldsymbol{\mu}_*)^\top \left[\tilde{\boldsymbol{G}}_*(\mathcal{X}_{\text{old}}\backslash\mathcal{M}) + \delta\mathbf{I}\right](\boldsymbol{w}-\boldsymbol{\mu}_*)\right)$.

In the Quadratic Lap-K-prior, we calculated $\boldsymbol{G}_*(\mathcal{X}_{\text{old}}\backslash\mathcal{M})$ using the base model. In the Quadratic var-K-prior, we can use already-computed second-order information in $\boldsymbol{\Sigma}_*$ to save computation, as $\boldsymbol{\Sigma}_*^{-1} = \tilde{\boldsymbol{G}}(\mathcal{X}_{\text{old}}) + \delta\mathbf{I}$ at the fixed-point of the variational objective (see Equation 5.5, where we now also use the GGN approximation to the Hessian). We can therefore write,

$$\tilde{\boldsymbol{G}}_*(\mathcal{X}_{\text{old}}\backslash\mathcal{M}) + \delta\mathbf{I} = \boldsymbol{\Sigma}_*^{-1} - \mathbb{E}_{q_{\boldsymbol{\eta}_*}(\boldsymbol{w})}\left[\,\boldsymbol{G}_*(\mathcal{M})\,\right], \tag{5.63}$$

and we can use Monte-Carlo sampling of the final term. This calculation is quick when the memory $\mathcal{M}$ is small.

Overall, the Quadratic var-K-prior first calculates $\tilde{\boldsymbol{G}}_*(\mathcal{X}_{\text{old}}\backslash\mathcal{M}) + \delta\mathbf{I}$ using Equation 5.63 and the base model, and then optimises the objective in Equation 5.62 (written for the 'Add Data' task, although similar expressions hold for other adaptation tasks). In the continual learning setting, over many sequential tasks, this can be viewed as combining var-K-priors with VCL, with a correction step where we remove datapoints from the covariance estimate (using Equation 5.63). We can use natural-gradient updates to get a version that is more similar to VOGN (see Section 3.2). When going from one task to the next, we can add and remove datapoints in the memory $\mathcal{M}$ by respectively adding and subtracting their contributions to $\tilde{\boldsymbol{G}}_*$ in Equation 5.63.

This concludes our section on improving weight-priors with Quadratic K-priors, which use functional regularisation as well as weight-prior quadratic regularisation. We expect Quadratic K-priors to improve on both weight-prior algorithms and vanilla K-priors when there is limited memory. Having considered how to improve weight-priors using K-priors, we next view FROMP (from Chapter 4) in the K-priors framework, and potential ways we could improve FROMP.

5.6 FROMP in the K-priors framework

We now take a closer look at continual learning by Functional Regularisation of Memorable Past (FROMP), which was described in Chapter 4. FROMP was introduced as a way to functionally regularise over memorable past points for continual learning. Variational-FROMP (var-FROMP) was derived by replacing the log-prior term in the (weight-space) variational objective with a function-space term (see Chapter 4 for more details). Approximations were then made to the var-FROMP algorithm, leading to two cheaper variants: OGN-FROMP and FROMP (these are described in Section 4.4).

In Appendix D.2, we show that even on linear regression, (var-)FROMP does not necessarily reconstruct the gradients of past information, and is therefore not always exact. This is despite storing all past data. This is concerning as linear regression is a very simple setting, where even weight-priors are exact. However, we know that FROMP performs very well on neural networks (see experiments in Section 4.5), and we want to understand why.

In this section, we view FROMP as approximating the gradient of past information in the K-priors framework. By considering FROMP on classification problems on neural networks, we consider the relationship between FROMP and Lap-K-priors. We see that FROMP approximates both the weight-space and function-space divergences in a specific way, attempting to maintain second-order information about the base model solution. This view also provides some suggestions on how to improve the FROMP algorithm.

We show relationships between FROMP and Lap-K-priors for the case when there is a single past task for simplicity, although we can straightforwardly extend to multiple past tasks. We focus this section on comparing FROMP with Lap-K-priors, but we can also draw very similar results for var-FROMP and var-K-priors due to the relationship between the two settings (as discussed in Section 5.1).

In order to write FROMP in the K-priors framework, we take the gradient of the regulariser in FROMP (the right-hand term in Equation 4.34, noting there is just a single past task),

$$\nabla_{\boldsymbol{w}} \mathcal{R}^{\text{FROMP}}(\boldsymbol{w}) = \tau \boldsymbol{J}_u^\top \boldsymbol{\Lambda}_u \left[\boldsymbol{\Lambda}_{u,*} \boldsymbol{J}_{u,*} \left[\boldsymbol{G}_*(\mathcal{X}_{\text{old}}) + \delta \mathbf{I}\right]^{-1} \boldsymbol{J}_{u,*}^\top \boldsymbol{\Lambda}_{u,*}\right]^{-1} \boldsymbol{d}_u, \tag{5.64}$$

where the vector $\boldsymbol{d}_u$ has its m'th entry as $h(f_{\boldsymbol{w}}(\boldsymbol{u}_m)) - h(f_{\boldsymbol{w}_*}(\boldsymbol{u}_m))$, $\boldsymbol{\Lambda}_u$ is a diagonal matrix with $h'(f_{\boldsymbol{w}}(\boldsymbol{u}_m))$ as the m'th diagonal entry, $\boldsymbol{\Lambda}_{u,*}$ is a diagonal matrix with $h'(f_{\boldsymbol{w}_*}(\boldsymbol{u}_m))$ as the m'th diagonal entry, $\boldsymbol{J}_u$ is an $M \times P$ matrix with rows given by $\nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{u}_m)$, and $\boldsymbol{J}_{u,*}$ is an $M \times P$ matrix with rows given by $\nabla_{\boldsymbol{w}} f_{\boldsymbol{w}_*}(\boldsymbol{u}_m)$. These are calculated over all $\boldsymbol{u}_m \in \mathcal{M}$, with $|\mathcal{M}| = M$, and our model has P parameters, $\boldsymbol{w} \in \mathbb{R}^P$. Additionally, $\boldsymbol{G}_*(\mathcal{X}_{\text{old}})$ is the GGN matrix at the base model $\boldsymbol{w}_*$ (defined in Equation 5.57). Note that the

previous covariance $\boldsymbol{\Sigma}_* = [\boldsymbol{G}_*(\mathcal{X}_{\text{old}}) + \delta \mathbf{I}]^{-1}$. When the memory $\mathcal{M}$ contains all past inputs, $\boldsymbol{G}_*(\mathcal{X}_{\text{old}}) = \boldsymbol{J}_{u,*}^\top \boldsymbol{\Lambda}_{u,*} \boldsymbol{J}_{u,*}$.

We can simplify Equation 5.64 by assuming that a pseudo-inverse of $\boldsymbol{J}_{u,*}$ exists. This will almost always be the *right* pseudo-inverse $\boldsymbol{J}_{u,*}^+ = \boldsymbol{J}_{u,*}^\top \left(\boldsymbol{J}_{u,*} \boldsymbol{J}_{u,*}^\top\right)^{-1}$, as we expect $P >> M$ for overparameterised neural networks (and hence $\text{rank}(\boldsymbol{J}_{u,*}) = M$). The FROMP regulariser then simplifies to,

$$\begin{aligned}
&\nabla_{\boldsymbol{w}} \mathcal{R}^{\text{FROMP}}(\boldsymbol{w}) \\
&\quad = \underbrace{\tau\delta \left[\boldsymbol{J}_u^\top \left[\boldsymbol{\Lambda}_u \boldsymbol{\Lambda}_{u,*}^{-1}\right] \boldsymbol{J}_{u,*}^{+\top}\right] \boldsymbol{J}_{u,*}^+ \boldsymbol{\Lambda}_{u,*}^{-1} \boldsymbol{d}_u}_{\text{Weight-divergence term}} + \underbrace{\tau \boldsymbol{J}_u^\top \left[\boldsymbol{\Lambda}_u \boldsymbol{\Lambda}_{u,*}^{-1}\right] \boldsymbol{J}_{u,*}^{+\top} \boldsymbol{G}_*(\mathcal{X}_{\text{old}}) \boldsymbol{J}_{u,*}^+ \boldsymbol{\Lambda}_{u,*}^{-1} \boldsymbol{d}_u}_{\text{Function-divergence term}}.
\end{aligned} \tag{5.65}$$

We now look at these two terms in more detail.

Weight-divergence term: The vanilla Lap-K-prior in Equation 5.13, which we know reconstructs the exact gradient when it stores all past inputs (on GLMs), has a weight-divergence term with gradient $\delta(\boldsymbol{w} - \boldsymbol{w}_*)$. FROMP's weight-divergence term recovers this gradient when (i) we make the first-order approximation to $\boldsymbol{d}_u$ from Equation 5.55, (ii) $\boldsymbol{\Lambda}_{u,*} = \boldsymbol{\Lambda}_u$, and (iii) $\boldsymbol{J}_{u,*} = \boldsymbol{J}_u$. FROMP's weight-divergence term can therefore be seen as approximating the vanilla Lap-K-prior's in this specific way.

Function-divergence term: When comparing the function-divergence term in Equation 5.65 with general K-priors in Equation 5.47, we see that FROMP corresponds to having,

$$\mathbf{B}^{\text{FROMP}} = \left[\boldsymbol{\Lambda}_u \boldsymbol{\Lambda}_{u,*}^{-1}\right] \boldsymbol{J}_{u,*}^{+\top} \boldsymbol{G}_*(\mathcal{X}_{\text{old}}) \, \boldsymbol{J}_{u,*}^+ \boldsymbol{\Lambda}_{u,*}^{-1}. \tag{5.66}$$

This is closely related to the Nyström approximation.

In Appendix D.1.2 we derive a K-prior that best matches second-order information from the trained base model using a limited memory, giving us (see Equation D.10),

$$\mathbf{B}^{*,\text{2ord}} = \boldsymbol{J}_u^{+\top} \boldsymbol{G}_*(\mathcal{X}_{\text{old}}) \, \boldsymbol{J}_u^+ \boldsymbol{\Lambda}_u^{-1}. \tag{5.67}$$

As we can see, $\mathbf{B}^{*,\text{2ord}}$ has a very similar form to $\mathbf{B}^{\text{FROMP}}$. Specifically, $\mathbf{B}^{\text{FROMP}}$ recovers this optimal $\mathbf{B}^{*,\text{2ord}}$ when (i) $\boldsymbol{\Lambda}_{u,*} = \boldsymbol{\Lambda}_u$, and (ii) $\boldsymbol{J}_{u,*} = \boldsymbol{J}_u$.

Using this view, FROMP uses previous task information via $\boldsymbol{G}_*$ to (approximately) preserve second-order information. This is useful as it ensures that our eventual solution is still at a minimum of the objective function. The strong empirical results with FROMP on neural networks provide further evidence that this is sensible to do.

This analysis provides some specific ways to improve the FROMP algorithm. Some simple changes are to (i) use the 'correct' weight-divergence term such as by using Equation 5.11, and/or (ii) recalculate the previous task kernel during optimisation with the new weights $\boldsymbol{w}$ instead of $\boldsymbol{w}_*$ (perhaps every few iterations to reduce computation cost), so that $\boldsymbol{\Lambda}_{u,*} = \boldsymbol{\Lambda}_u$ and $\boldsymbol{J}_{u,*} = \boldsymbol{J}_u$ in FROMP's objective. If we did both of these changes, we would get a K-prior that best matches second-order information, as derived in Appendix D.1.2.

Exploring such insights and connections between FROMP and K-priors more will be very interesting future work, and we discuss this further in Section 6.2.

5.7 Experiments

In this section, we look at how well K-priors perform experimentally. Our goal is to show that K-priors can lead to *quick, accurate and wide* adaptation. K-priors are quick because they are computationally cheaper than retraining-from-scratch, while still being accurate (performing almost as well as retraining-from-scratch). K-priors are computationally cheaper when we use small memories, and in our experiments, we will see how K-priors perform almost as well as retraining-from-scratch even at small memories (often $2-10\%$ of the full data). K-priors achieve wide adaptation as they can be applied to many adaptation tasks and models. We focus only on Lap-K-priors, and leave experiments with var-K-priors as future work. We also mostly focus on vanilla K-priors, and see strong performance with them. Throughout the chapter we derived K-priors that we expect to perform better in the limited memory setting, such as Quadratic K-priors in Section 5.5 and improvements to FROMP in Section 5.6. We run one experiment with the Quadratic Lap-K-prior on continual learning (on the Split MNIST benchmark), where we see very strong performance, but leave a detailed empirical investigation of these better K-priors as future work.

For most of this section, we experimentally test on four adaptation tasks: the 'Add Data' task, the 'Remove Data' task, the 'Change Regulariser' task, and the 'Change Model Class/Architecture' task (these are defined in Sections 5.1 and 5.3). We use Generalised Linear Models (GLMs) and neural networks. We show promising results on larger deep-learning problems, with potential for improvement when we borrow tricks such as a temperature from knowledge distillation (see discussion in Section 5.2).

We compare the performance of vanilla K-priors to retraining-from-scratch on all data (which we call 'Retrained', previously also referred to as 'Joint Tasks' in Chapter 2) and a retraining with replay from a small memory ('Replay'), and we always use $\tau = 1$. For a fair comparison, we use the same memory for Replay and K-priors obtained by choosing points with highest $h'(f^i_{\boldsymbol{w}_*})$ (or the Lambda method, see Section 5.4 for details). Memory chosen

randomly often gives much worse results and we omit these results. Replay uses true labels while K-priors use model predictions, and also has a different weight-divergence term. We will see that Replay performs much worse at small memory sizes.

When we compare with weight-priors, we see that they generally perform well, but they are worse when the adaptation involves a drastic change in inputs. In Table 5.1 we show that training K-priors with limited memory is quicker than both Retrained and Replay.

Details for all experiments are in Appendix D.4, such as hyperparameters and more details on experimental setups. Appendix D.4.3 looks at how well K-priors perform without the weight-divergence term, finding that the weight-divergence term is usually crucial for good performance. Appendix D.4.4 studies the effect of random initialisation, and we find that it performs similarly to initialising at the base model parameters $\boldsymbol{w}_*$. Code is available at `https://github.com/team-approx-bayes/kpriors`.

Generalised Linear Models

We start with two datasets on Generalised Linear Models (GLMs): logistic regression with the UCI Adult dataset, and the 'USPS odd vs even' dataset. Both are binary classification tasks, and we compare the three methods (Retrained, Replay and vanilla K-priors) on all four adaptation tasks. Results are shown in Figure 5.2.

Logistic Regression on the UCI Adult dataset. This is a binary classification problem consisting of $16,100$ examples to predict income of individuals. We randomly sample 10% of the training data (1610 examples), and report mean performance and standard deviation over 10 such splits. For training, we use the L-BFGS optimiser (default PyTorch implementation (Paszke et al., 2019)) for logistic regression with polynomial basis, using a learning rate of 0.01 and running until convergence. Results are summarised in Figure 5.2(a). For the 'Add Data' task, the base model uses 9% of the data and we add 1% new data. For 'Remove Data', we remove 100 of the most important data examples (6% of the training set) picked by sorting $h'(f^i_{\boldsymbol{w}_*})$. For the 'Change Regulariser' task, we change the L_2-regulariser strength from $\delta = 50$ to 5, and for 'Change Model Class', we reduce the polynomial degree from 2 to 1. Note that for all but the 'Change Model Class' task, we use polynomial degree 1, and for all but the 'Change Regulariser' task, we use $\delta = 5$.

As we see in Figure 5.2(a), K-priors perform very well on the first three tasks, remaining very close to Retrained, even when the memory sizes are down to 2%. The 'Changing Model Class' task is slightly more challenging, but K-priors still significantly out-perform Replay. In Appendix D.4.1 we provide an ablation study for Replay with different strategies on this dataset, testing different τ and randomly choosing the points to store.

Figure 5.2: We compare vanilla K-priors (red squares) with the expensive Retrained baseline (grey) and Replay (blue circles) on our four adaptation tasks (columns) with (a) logistic regression on UCI Adult dataset, and (b) logistic regression on the 'USPS odd vs even' dataset. We find that K-priors match Retrained while mostly using $2-5\%$ of the data (for only 3 tasks a larger fraction is required). K-priors always outperform Replay. Replay uses the true labels, while K-priors replace the labels by the model predictions, and have a slightly different weight-divergence term. Details on experimental setup is in the main text.

Logistic Regression on the 'USPS odd vs even' dataset. The USPS dataset (Hull, 1994) consists of 10 classes (one for each digit), and has $7,291$ training images of size 16×16. We split the digits into two classes: odd and even digits. Results are in Figure 5.2(b). For the 'Add Data' task, we add all examples of the digit 9 to the rest of the dataset, and for 'Remove Data' we remove the digit 8 from the whole dataset. By adding/removing an entire digit, we enforce a *heterogeneous* data split, making the tasks more challenging. The 'Change Regulariser' and 'Change Model Class' tasks are the same as the UCI Adult dataset. Note that for all but the 'Change Model Class' task, we use polynomial degree 1, and for all but the 'Change Regulariser' task, we use $\delta = 50$. We optimise using L-BFGS until convergence (with learning rate 0.1).

K-priors perform very well on the 'Add Data' and 'Change Regulariser' tasks, always achieving close to Retrained performance even with small memories. For the 'Remove Data' task, which is a challenging task due to heterogeneity, K-priors still only need to store 5% of past data to maintain close to 90% accuracy, whereas Replay requires 10% of past data.

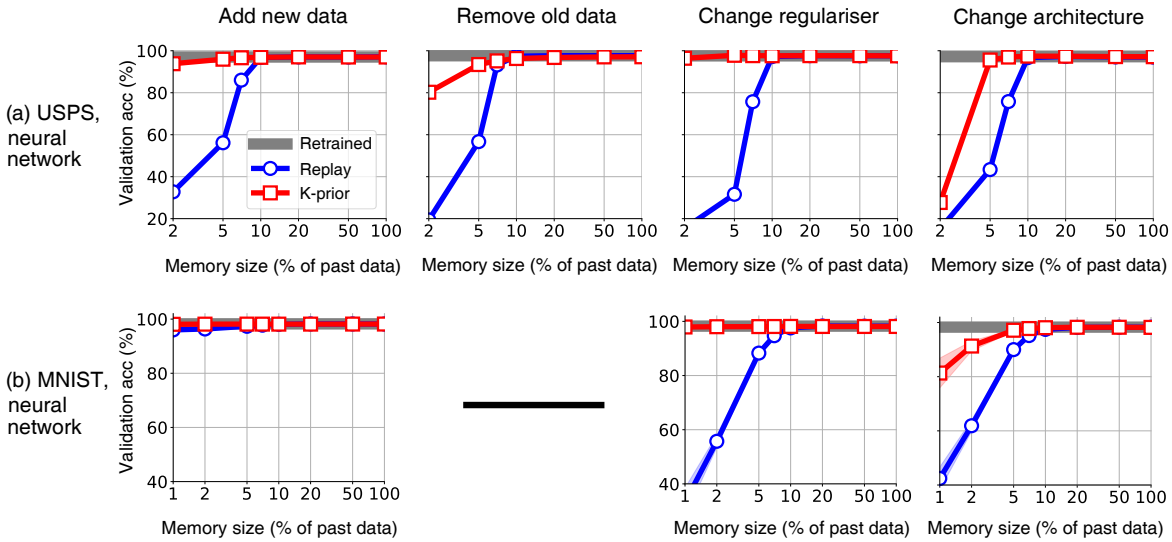


Figure 5.3: We compare vanilla K-priors (red squares) with the expensive Retrained baseline (grey) and Replay (blue circles) with neural networks on (a) four adaptation tasks on the 'USPS odd vs even' dataset, and (b) three adaptation tasks on the MNIST dataset. We find similar results to Figure 5.2, where we used GLMs instead of neural networks: K-priors match Retrained while mostly using $2-5\%$ of the data. Additionally, K-priors always outperform Replay. Replay uses the true labels, while K-priors replace the labels by the model predictions, and have a slightly different weight-divergence term.

Neural networks

We now run on neural networks instead of GLMs. On neural networks, there is an additional error term when using vanilla K-priors, but we hope this might be beneficial like in knowledge distillation (a more detailed discussion is in Section 5.2). We compare to the same methods as before (Retrained and Replay), and run on a small multi-layer perceptron (MLP) with 'USPS odd vs even', a larger MLP with MNIST, and CNNs with CIFAR-10. We also find that we can improve knowledge distillation by using a smaller memory.

Neural Networks on the 'USPS odd vs even' dataset. This is a repeat of the previous experiment with USPS data, but now with a neural network (a one hidden-layer MLP with 100 units) instead of a GLM. Results are in Figure 5.3(a). The 'Change Regulariser' task now changes $\delta = 5$ to 10, and the 'Change Architecture' task compresses the architecture from a two hidden-layer MLP (100 units per layer) to a one hidden-layer MLP with 100 units. For all but the 'Change Regulariser' task, we use $\delta = 5$. We optimise using Adam with a learning rate of 0.005 for 1000 epochs (which is long enough to reach convergence). We see that even with neural networks, K-priors perform very well, similarly out-performing Replay and remaining close to the Retrained solution at small memory sizes.

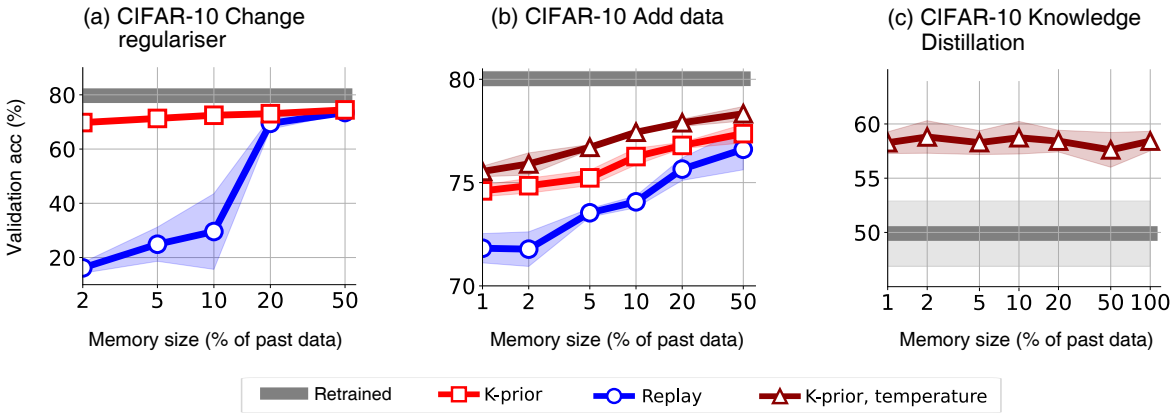


Figure 5.4: (a+b) We compare K-priors (red squares) with the expensive Retrained baseline (grey) and Replay (blue circles) on the 'Change Regulariser' and 'Add Data' tasks in CIFAR-10. We see that although K-priors outperform Replay, there is a gap to Retrained. This gap is reduced when we use a temperature parameter in K-priors (dark-red triangles). (c) The same is true for knowledge distillation (Hinton et al., 2015), and we see that we can reduce memory size while still performing better than the student model, with as little as 1% of total memory for the distillation term. Details on experimental setup is in the main text.

Neural Networks on the MNIST dataset. We show results on 10-way classification with MNIST (LeCun and Cortes, 2010) in Figure 5.3(b), which has $60,000$ training images across 10 classes (handwritten digits), with each image of size 28×28. We use a two hidden-layer MLP with 100 units per layer, and report mean performance and standard deviation over 3 runs. For the 'Add Data' task, we start with a random 90% of the dataset and add 10%. For the 'Change Regulariser' task, we change $\delta = 1$ to 5 (we use $\delta = 1$ for all other tasks). For the 'Change Architecture' task, we compress to a single hidden layer with 100 hidden units. We did not run on the 'Remove Data' task. We optimise using Adam with a learning rate of 0.001 for 250 epochs, using a minibatch size of 512.

We see that K-priors perform well on MNIST on our three adaptation tasks, just like in all other experiments, outperforming Replay at low memory sizes and remaining close to the Retrained solution.

Neural Networks on the CIFAR-10 dataset. We also provide results for CIFAR-10 (Krizhevsky and Hinton, 2009), using 10-way classification, in Figure 5.4. CIFAR-10 has $60,000$ images ($50,000$ for training), and each image has 3 channels, each of size 32×32. We report mean performance and standard deviation over 3 runs. We use the CifarNet architecture from Zenke et al. (2017), which we also used previously in our CIFAR experiments in Chapters 2 to 4, and which we described in Section 2.4. We optimise using Adam with a learning rate of 0.001 for 100 epochs, using a batch size of 128.

In Figure 5.4(a) we provide results on the 'Change Regulariser' task, where we change $\delta = 1$ to 0.5 (we use $\delta = 1$ for all the other tasks). In Figure 5.4(b) we show the 'Add Data' task with CIFAR-10, where we add a random 10% of CIFAR-10 training data to the other 90%. Although vanilla K-priors outperform Replay, there is now a bigger gap between K-priors and Retrained even with 50% past data stored. However, performance improves when we use a temperature (similar to knowledge distillation).

A similar result is shown in Figure 5.4(c) for knowledge distillation, where we compress from a CifarNet teacher to a LeNet5-style student (details in Appendix D.4). Here, K-priors with 100% data is equivalent to knowledge distillation, but when we reduce the memory size using our method, we still outperform Retrained (which trains the student model from scratch on all data). As in all other experiments, we use the Lambda method to choose the memory points, but it might also be interesting in future work to use the residual to choose points (see Equation 5.34, where the error depends on the residuals). Overall, our initial effort here suggests that K-priors can do better than Replay, and have potential to give better results with more hyperparameter tuning.

K-priors converge cheaply

We now show that K-priors with limited memory converge to the final solution cheaply, converging in far fewer passes through data than the Retrained solution. This is because we use a limited memory, only touching important past datapoints.

Accuracy achieved	**Method**	**Add new data**	**Remove old data**	**Change regulariser**	**Change architecture**
90%	Retrained	87	94	94	86
90%	Replay (10% memory)	350	110	240	75
90%	**K-prior (10% memory)**	**73**	**53**	**13**	**22**
97%	Retrained	1,900	1,800	2,700	3,124
97%	Replay (10% memory)	–	340	–	–
97%	**K-prior (10% memory)**	**330**	**120**	**54**	**68**

Table 5.1: Number of backpropagations required to achieve specified accuracies (90% and 97%) with a neural network on the 'USPS odd vs even' dataset (1000s of backprops). K-priors with 10% past memory require significantly fewer backprops to achieve the same accuracy as Retrained. Although Replay uses the same memory as K-priors, it still requires more backprops to reach 90% accuracy, as it uses true labels instead of model predictions. Additionally, Replay with with 10% memory cannot achieve higher accuracies (like 97% accuracy) in many adaptation tasks. We use the same experimental settings as in Figure 5.3(a) and as described in the main text.

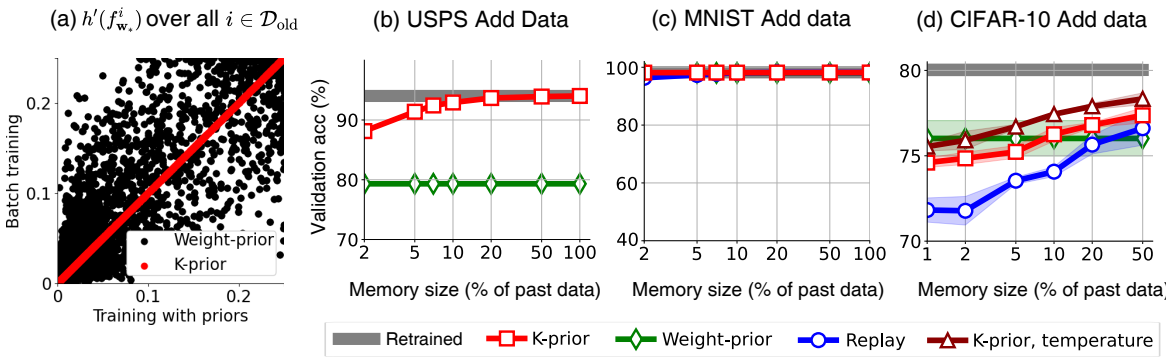


Figure 5.5: (a) When compared at the Retrained solution for the 'Add Data' task on USPS, weight-priors give incorrect values of $h'(f^i_{\boldsymbol{w}})$ (shown with black dots, each dot corresponds to a data example). Points on the diagonal indicate a perfect match, which is the case for K-priors (shown with red dots). (b) Due to this, weight-priors (green diamonds) perform worse than K-priors (red squares). This is a heterogeneous split and is therefore particularly difficult for weight-priors. (c+d) Homogeneous data splits for the 'Add Data' task on MNIST and CIFAR-10 result in weight-priors performing better. On MNIST, all methods perform extremely well. In Appendix D.4.2 we provide similar results for the remaining 'Add Data' tasks: logistic regression on UCI Adult and neural networks on the 'USPS odd vs even' dataset, where we see the same story.

Table 5.1 shows the "number of backpropagations" until reaching specific accuracies (90% and 97%) with a neural network on the 'USPS odd vs even' dataset (using the same settings as in Figure 5.3(a) and previously described in the text). This is one way of measuring the "time taken", as backprops through the model are the time-limiting step. For K-priors and Replay, we use 10% of past memory. All methods use random initialisation when starting training on a new task.

We see that K-priors with 10% of past data stored are quicker to converge than Retrained, even though both eventually converge to the same accuracy (as seen in Figure 5.3(a)). For example, to reach 97% accuracy for the 'Change Regulariser' task, K-priors only need $54,000$ backward passes, while Retrained requires $2,700,000$ backward passes. We also see that Replay is usually very slow to converge. This is because it does not use the same information as K-priors (as Replay uses true labels), and therefore requires significantly more passes through data to achieve the same accuracy. In addition, Replay with 10% of past data cannot achieve high accuracies (such as 97% accuracy), as seen in Figure 5.3(a).

Weight-priors vs vanilla K-priors

As discussed in the main text in Section 5.5, weight-priors can be seen as an approximation of K-priors where $h'(f^i_{\boldsymbol{w}})$ are replaced by 'stale' $h'(f^i_{\boldsymbol{w}_*})$ evaluated at the old $\boldsymbol{w}_*$. In Figure 5.5(a), we visualise these 'stale' $h'(f^i_{\boldsymbol{w}_*})$ and compare them to K-priors. Points on the diagonal indicate a perfect match to the Retrained $h'(f^i_{\boldsymbol{w}})$, and we see that this is the case for K-priors but not for weight-priors. For this experiment, we use logistic regression on the 'USPS odd vs even' dataset (the 'Add Data' task). This heterogeneous data split is difficult for weight-priors, and we show in Figure 5.5(b) that weight-priors do not perform well. Weight-priors perform better with homogeneous data splits (such as in Figure 5.5(c+d)). This indicates that weight-priors do not have a mechanism to fix mistakes made in the past, while in K-priors, we can always change $\mathcal{M}$ to improve performance. In Appendix D.4.2 we provide similar results for the remaining 'Add Data' tasks (logistic regression on UCI Adult and neural networks on the 'USPS odd vs even' dataset), where we see the same story.

Quadratic K-priors for continual learning

We have seen how vanilla K-priors perform well on many adaptation tasks on a variety of datasets on both GLMs and neural networks, and we now consider continual learning (which consists of multiple sequential 'Add Data' tasks). Throughout this chapter, we have seen how K-priors relate to continual learning algorithms such as Online EWC (Kirkpatrick et al., 2017; Schwarz et al., 2018), VCL (Nguyen et al., 2018) (Chapter 3) and FROMP (Chapter 4). We now use the Quadratic Lap-K-prior from Section 5.5, and see how it improves on both weight-priors and vanilla K-priors on the Split MNIST benchmark. All hyperparameters are reported in Appendix D.4.

We start by running Online EWC on the Split MNIST benchmark, using the same experimental protocol as described in Section 2.4, in particular using a two-hidden layer MLP with 256 hidden units in each layer. After optimising for hyperparameters, we get an accuracy of $98.8 \pm 0.1\%$, which is significantly better than just EWC or Laplace Propagation (see results in Chapters 3 and 4, which we had taken from Nguyen et al. (2018)). We then add functional regularisation over some points with Quadratic K-priors, choosing points by the Lambda method. We store 2–40 points per task (2 points per task corresponds to 1 point per class), and we see significant improvement in performance over weight-priors. Results are in Figure 5.6, where we plot mean performance and standard deviation over 5 runs.

We also compare to vanilla K-priors, finding a bigger improvement when there are fewer points (this is due to weight-regularisation helping more when there are very few datapoints in memory). With 2 memory points per task, Quadratic K-priors get $99.15 \pm 0.17\%$, while

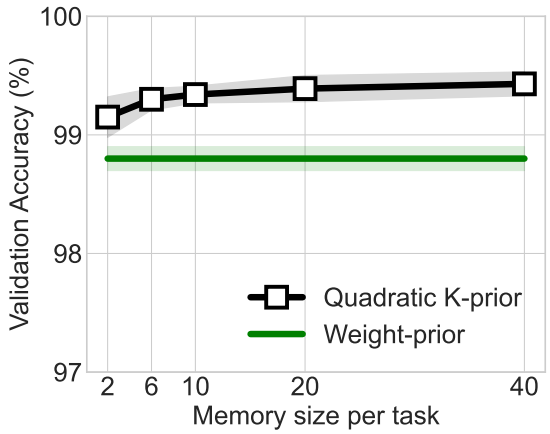


Figure 5.6: Quadratic K-priors achieve extremely good results at all memory sizes on Split MNIST, a continual learning benchmark consisting of 5 binary classification tasks presented sequentially. We compare with a weight-prior method, Online EWC (Schwarz et al., 2018). We plot mean performance and standard deviation over 5 runs. Our Quadratic Lap-K-prior improves upon the weight-prior even with 2 memory points per task (this corresponds to 1 image per class in Split MNIST). With 40 datapoints per task, we achieve $99.4 \pm 0.1\%$, which significantly outperforms other continual learning methods considered in this thesis, including FROMP from Chapter 4 (see Table 4.3). Quadratic K-priors also outperform vanilla K-priors, see main text for results.

vanilla K-priors get $96.8 \pm 0.4\%$. With 40 memory points per task, Quadratic K-priors gets $99.4 \pm 0.1\%$, while vanilla K-priors get $99.26 \pm 0.08\%$. This is significantly better than FROMP from Chapter 4, which achieved $99.0 \pm 0.1\%$ (see Table 4.3).

These results are extremely promising for Quadratic K-priors in continual learning. However, Split MNIST is a small-scale benchmark, and it would be very interesting to see how well Quadratic K-priors do on larger benchmarks such as Split CIFAR, where FROMP performs very well. We leave this for future work.

5.8 Links to Support Vector Machines and Gaussian Processes

In this section, we briefly explain links between (i) K-priors and algorithms for Support Vector Machines (SVMs) (Cauwenberghs and Poggio, 2001; Tveit et al., 2003; Laskov et al., 2006; Duan et al., 2007; Romero et al., 2007; Liang and Li, 2009; Karasuyama and Takeuchi, 2010; Tsai et al., 2014), and (ii) K-priors and online Gaussian Processes (GPs) (Csató and Opper, 2002; Särkkä et al., 2013; Solin et al., 2018). Previous works usually focus on either the 'Add Data' task or the 'Remove Data' task, and we will see that K-priors are very closely related to these adaptation mechanisms in both settings. K-priors also link to many other works too, such as knowledge distillation (Section 5.2), weight-priors (Section 5.5), and rehearsal-based continual learning (Section 5.6), providing a way to unify and generalise many of these previous concepts.

Adding/removing data for Support Vector Machines (SVMs)

K-prior-regularised training yields equivalent solutions to the adaptation strategies used in SVMs to add/remove data examples. K-priors can be shown to be equivalent to the primal formulation of such strategies (Liang and Li, 2009). The key trick to show the equivalence is to use the representer theorem, which we will now illustrate for the 'Add Data' task (from Equation 5.17). Let $\boldsymbol{\Phi}_+$ be the $(N_{\text{old}}+1) \times P$ feature matrix obtained on the dataset $\mathcal{D}_{\text{old}} \cup \mathcal{D}_{\text{new}}$, where $\mathcal{D}_{\text{new}}$ consist of 1 new example. By the representer theorem we know that there exists a $\boldsymbol{\beta} \in \mathbb{R}^{N+1}$ such that $\boldsymbol{w}_+ = \boldsymbol{\Phi}_+^\top \boldsymbol{\beta}$. Taking the gradient of Equation 5.17, and multiplying by $\boldsymbol{\Phi}_+$, we can write the optimality condition as,

$$0 = \boldsymbol{\Phi}_+ \nabla \left[\sum_{j \in \mathcal{D}_{\text{new}}} \ell_j(\boldsymbol{w}_+) + \mathcal{K}(\boldsymbol{w}_+) \right] = \sum_{i \in \mathcal{D}_{\text{old}} \cup \mathcal{D}_{\text{new}}} \left(\nabla_f \ell(y_i, h(f))|_{f=\boldsymbol{\beta}_i^\top \boldsymbol{k}_{i,+}} \right) \boldsymbol{k}_{i,+} + \delta \boldsymbol{K}_+ \boldsymbol{\beta}, \tag{5.68}$$

where $\boldsymbol{K}_+ = \boldsymbol{\Phi}_+ \boldsymbol{\Phi}_+^\top$ and its i'th column is denoted by $\boldsymbol{k}_{i,+}$. This is exactly the gradient of the primal objective in the function-space defined over the full batch $\mathcal{D}_{\text{old}} \cup \mathcal{D}_{\text{new}}$ (see Equation 3.6 in Chapelle (2007)). The primal strategy is equivalent to the more common dual formulations (Cauwenberghs and Poggio, 2001; Tveit et al., 2003; Laskov et al., 2006; Duan et al., 2007; Romero et al., 2007; Karasuyama and Takeuchi, 2010; Tsai et al., 2014). The function-space formulations could be computationally expensive, but speed-ups can be obtained by using support vectors. This is similar to the idea of using limited memory in K-priors in Section 5.4.

Another type of adaptation is to add *privileged* information, originally proposed by Vapnik and Izmailov (2015). The goal is to include different types of data to improve the performance of the model. This is combined with knowledge distillation by Lopez-Paz et al. (2016) and has been applied to domain adaptation in deep learning (Ao et al., 2017; Ruder et al., 2017; Sarafianos et al., 2017). As we discussed in Section 5.2, K-priors can be seen as an easy-to-implement scheme for this Similarity Control, and could similarly be useful for student-teacher learning. This is because K-prior-regularisation occurs in a primal formulation, instead of using slack variables, which require a dual formulation.

Connections to Gaussian Processes

When variational K-priors are written in function-space similarly to Equation 5.68, they are related to the online updates used in Gaussian Processes (GPs) (Csató and Opper, 2002). When $q_{\mathcal{K}}$ is built with limited memory, as described in Section 5.4, the application is similar to sparse variational GPs, but now data examples are used as inducing inputs. These connections are discussed in more detail in Appendix D.3. Our K-prior formulation operates in weight-space and can be easily trained with first-order methods, however an equivalent formulation in function-space can also be employed, as is clear from these connections. The above extensions can be extended to handle arbitrary exponential-family approximations by appropriately defining K-priors using KL divergences, instead of just restricting to Gaussian approximating families. Future work would look at such extensions.

5.9 Summary

In this chapter, we presented Knowledge-adaptation priors (K-priors). K-priors use weight-space and function-space divergences to reconstruct gradients of past information. We started in Section 5.1 by showing how vanilla K-priors can perfectly reconstruct gradients of past information with sufficiently large memory on the 'Add Data' task in two settings: the MAP/Laplace setting and the variational setting. We then looked at vanilla K-priors on neural networks, finding close links with knowledge distillation. We found that we could exploit these links to improve both K-priors and knowledge distillation.

In Section 5.3 we then applied K-priors to other adaptation tasks which are important in machine learning, such as removing data, changing the regulariser and changing the model class or architecture. We then looked at K-priors for general learning problems. This allowed us to transition to settings where we have a limited memory budget and want to achieve optimal knowledge transfer. In Section 5.4 we considered optimal K-priors, which reconstruct gradients by using singular vectors, but are difficult to realise in practice. We

also looked at how to choose memory points by looking at the error when we store a subset of data in vanilla K-priors.

By comparing with weight-priors, in Section 5.5 we introduced Quadratic K-priors, which improve weight-priors with functional regularisation. They can also be viewed as improving vanilla K-priors with quadratic weight-regularisation. In Section 5.6 we placed FROMP (from Chapter 4) in the K-priors framework, and this provided insights into why FROMP performs so well: FROMP uses its memory to maintain second-order information from old data. We derived an optimal way to maintain second-order information, and used this to suggest improvements to FROMP.

Experiments with vanilla K-priors in Section 5.7 showed strong results on many different datasets with GLMs and neural networks on our four adaptation tasks. K-priors are quicker than retraining-from-scratch, while still being accurate and performing significantly better than baselines such as Replay and weight-priors. We also ran Quadratic K-priors on the continual learning Split MNIST benchmark, seeing it performs better than methods from previous chapters.

We ended the chapter by briefly exploring links between K-priors and Support Vector Machines, and K-priors and online Gaussian Processes in Section 5.8.

There are many directions for future work based on K-priors. We took some steps towards deriving K-priors that efficiently use memory in this chapter, and we can do more both theoretically and empirically. We can also consider using these insights to improve methods for memory selection. Another strand of work can look at links between K-priors and adaptation mechanisms in machine-learning models, expanding on Section 5.8 and finding more theoretical relationships. There is also work we can do empirically with var-K-priors, including combining with natural-gradient algorithms (see Section 2.2.2) to derive new probabilistic algorithms for knowledge transfer. We discuss some of these ideas in more detail in Section 6.2.

Chapter 6

Conclusions and future work

This chapter concludes and summarises this thesis. We start with a summary in Section 6.1. We then provide a discussion in Section 6.2, considering potential avenues for future research.

6.1 Summary

We started this thesis by motivating probabilistic continual learning using neural networks in Chapter 1. Chapter 2 formalised these concepts, defining continual learning, the probabilistic approach, and introducing metrics and benchmarks. We also categorised methods for continual learning into three orthogonal complementary approaches: regularisation-based, rehearsal-based and architecture-based approaches.

Weight-space variational continual learning. We considered regularisation-based approaches in Chapter 3. We started with a variational weight-prior algorithm, Variational Continual Learning (Nguyen et al., 2018), and improved the algorithm's convergence rate and performance. We analysed why our changes led to improvements, finding that entire units were pruned out, and that this was beneficial in continual learning (although it can be detrimental in other settings due to underfitting). We then used natural-gradient updates to significantly improve the rate at which learning convergences, accelerating training and allowing larger models and datasets to be handled. We scaled the Variational Online Gauss-Newton (VOGN) algorithm (Khan et al., 2018) to large datasets/architectures such as ImageNet/ResNets for the first time, finding it converged in as many epochs as SGD and Adam with similar accuracy, while adding some benefits of Bayesian principles. VOGN also improved convergence rate in continual learning, allowing us to scale to Split CIFAR. However, it still did not perform as well as we desired, and we argued in Section 3.3 that this is due to restrictive independence approximations we made in weight-space regularisation.

Functional regularisation of memorable past. This motivated us to regularise network outputs or functions directly in Chapter 4, thereby combining regularisation-based approaches with rehearsal-based approaches. We replaced the log-prior term in weight-priors with its function-space alternative, using a Gaussian Process formulation of neural networks to identify and regularise on a few memorable past datapoints. The fully variational algorithm was computationally expensive, and we proposed a series of approximations. Our final algorithm, FROMP, performed extremely strongly on our benchmarks, outperforming our previous regularisation-based approaches. We looked at relaxing some of our approximations in OGN-FROMP, which performed very well on Split MNIST, and also looked at different ways of choosing memorable past datapoints.

Knowledge-adaptation priors. We introduced Knowledge-adaptation priors (K-priors) in Chapter 5. K-priors combine a weight-divergence term and a function-divergence term to reconstruct the gradient of past information, and are a generalisation of FROMP and weight-priors. We saw how vanilla K-priors can perfectly reconstruct gradients in both the MAP/Laplace and variational settings on Generalised Linear Models (GLMs). When applying to neural networks, we saw a link to knowledge distillation (Hinton et al., 2015), and we used this to improve both K-priors and knowledge distillation. We also saw how K-priors can be applied to multiple adaptation tasks such as removing data, changing the regulariser, and changing the model class or architecture.

We then looked at how to apply K-priors to the limited-memory setting. We presented ways of choosing points to store, and ways to design the divergence terms in K-priors to optimally use our stored points. When we compared K-priors with weight-priors, we saw we could improve weight-priors with functional regularisation, calling the algorithm Quadratic K-priors. When we placed FROMP in the K-priors framework, we saw that FROMP does not perfectly reconstruct the gradient of past information, even on GLMs. However, FROMP approximately maintains second-order information from previous tasks' objectives, and we used this insight to propose ways to improve FROMP. Results on GLMs and neural networks showed K-priors perform well on a variety of datasets and adaptation tasks. We also saw Quadratic K-priors perform well on the Split MNIST continual learning benchmark, outperforming weight-priors, vanilla K-priors and FROMP. We ended the chapter by comparing K-priors to algorithms for adaptation in Support Vector Machines and online Gaussian Processes, finding that there are many fundamental theoretical links unified by the K-priors framework.

6.2 Discussion and future work

There are many open questions and potential avenues for future work, and we summarise some of them in this section, including mentioning some ongoing work.

The many benefits of being probabilistic. We start by briefly taking a step back from continual learning. The probabilistic framework has potential to provide benefits beyond just naturally handling continual learning, and we have mostly ignored this potential in this thesis. Our motivating example in Chapter 1 with image classifiers for autonomous vehicles focussed on the importance of performing fast sequential updates, but we may also have other requirements for this image classifier, such as robustness to overfitting, good uncertainty calibration, and good performance in the low-data regime. The probabilistic framework promises to perform well in all of these settings (MacKay, 1992; Mackay, 1995; Neal, 1995; Gal, 2016). One strand of future work would evaluate the probabilistic algorithms developed in this thesis on such problems. Applying our algorithms to the batch-setting is straightforward for weight-space algorithms (and we did this in Section 3.2.2) but more difficult for function-space algorithms.

Improving weight-space variational algorithms further. In Chapter 3 we used VOGN, which converged significantly better than Bayes-By-Backprop (Blundell et al., 2015), allowing us to scale to larger problems. But we could further improve our natural-gradient variational inference (NGVI) algorithms to get even faster convergence and better performance. For example, we could use the VOGGN algorithm (which we introduced in Chapter 4 and is from Khan et al. (2019)), which uses the standard Generalised Gauss-Newton approximation to the Hessian, which we expect is a better approximation than that used in VOGN. We could alternatively use the Improved Bayesian learning rule (iBLR) algorithm (Lin et al., 2020), which uses another way to approximate the Hessian in NGVI, and recently outperformed all other methods in a competition (Wilson et al., 2021). We could also straightforwardly combine NGVI algorithms for continual learning with coresets, like in VCL+Coreset (Nguyen et al., 2018). This should improve performance similar to how VCL+Coreset improves on VCL in Section 3.1.

Relaxing approximations in variational-FROMP. In Section 4.4 we detailed five approximations that we made to variational-FROMP to derive our FROMP algorithm. We could relax many of these approximations to see if they lead to improved performance. Relaxing approximations can lead to more expensive algorithms, but has the potential to improve results. We showed this potential with OGN-FROMP in Section 4.5.2, seeing significantly

improved performance on Split MNIST. Future work can apply OGN-FROMP to larger benchmarks, and also consider relaxing some of the other five approximations. Alternatively, we could use better NGVI algorithms (such as VOGGN or iBLR instead of OGN). As discussed earlier, these are expected to be better approximations than OGN, and therefore should lead to better performance.

Moving beyond the task-boundary assumption in continual learning. In Section 2.1 we defined continual learning through a list of desiderata, and said we would aim to satisfy all but one desideratum in this thesis. Specifically, we relaxed the fully-online setting in continual learning: we assumed that data examples arrive in defined tasks, and we are informed when data from a new task starts arriving. This assumption may not be realistic in some real world scenarios, and we can relax it in various ways, building on work presented throughout this thesis. In Appendix H of Pan et al. (2020) we showed a method of automatically detecting task boundaries with FROMP (using inspiration from Titsias et al. (2020)), but this still assumes that our data examples are separated into tasks. More generally, we discussed in Section 4.4.1 how the OGN-FROMP algorithm (and also var-FROMP) allows for moving to general online learning. OGN-FROMP maintains a covariance matrix throughout learning, therefore allowing us to view the network in function-space at any point during training instead of waiting until the task changes. We would still need to choose memorable past points in some way, but there are simple algorithms for doing this that we can build from, such as reservoir sampling (Vitter, 1985). We could also consider using K-priors for general online learning in a similar way, potentially using the links between K-priors and FROMP to suggest new methods.

Memorable experiences of machine-learning models. In ongoing work, we look at generalising the concept of memorable experiences. We derive criteria (such as the Lambda method and Leverage method from Sections 4.3 and 5.4) directly from the probabilistic framework, perturbing the posterior's moments to reveal datapoints that are the most important. This turns out to be closely related to many existing methods for choosing a subset of important points, and can also be applied to unsupervised learning and reinforcement learning. This project is in collaboration with Dharmesh Tailor, Paul E Chang, Arno Solin and Mohammad Emtiyaz Khan.

In general, improving memorable past points selection should directly improve our methods' performance in continual learning. We can use the K-priors framework to guide our choice of points, and we provided an initial straightforward attempt in Section 5.4. We could also look further into why the Leverage method did not outperform the Lambda method

significantly in FROMP in Section 4.5.1, as this was an unexpected result (theoretically, the Leverage method should perform better). It is possible that the Leverage method will outperform the Lambda method on more difficult benchmarks. For example, when we move beyond the task-boundary assumption in continual learning, we may benefit from the additional randomness during sampling in the Leverage method.

Using the K-priors framework to design better algorithms. Chapter 5 used the K-priors framework to design better algorithms such as Quadratic K-priors (which improve weight-priors using functional regularisation) and improving FROMP. These new algorithms are theoretically better than previous methods, and there is potential to design even better algorithms using the K-priors framework. For example, Appendix D.1 looked at designing the divergences in K-priors based on maintaining first-order and second-order information, and future work can explore further possibilities.

We could also combine variational K-priors with natural-gradient variational inference algorithms (such as VOGN or iBLR). We hope that variational algorithms perform better than the Laplace variants as they use a more global approximation (Opper and Archambeau, 2009), and natural-gradient updates should improve convergence rate and scalability, similar to the improvements we obtained with VOGN in Section 3.2.

Future work can also apply these improved algorithms to continual learning benchmarks. We saw promising results with Quadratic K-priors on Split MNIST in Section 5.7, and we are running on larger benchmarks in ongoing work with Erik Daxberger, Kazuki Osawa, Runa Eschenhagen, Rio Yokota, José Miguel Hernández-Lobato, Richard E Turner and Mohammad Emtiyaz Khan.

Exploring theoretical links between K-priors and other adaptation algorithms. In Section 5.8 we explored some links between K-priors and Support Vector Machines, and K-priors and Gaussian Processes, seeing that K-priors can be shown to be equivalent to various adaptation algorithms. This leaves some interesting questions regarding further theoretical links between K-priors and other adaptation algorithms, and if more adaptation algorithms can be shown to be related to K-priors. We believe that this is possible, and that the K-priors framework is a unification and generalisation of existing algorithms previously thought to be unconnected.

Miscellaneous future work.

Combining with architecture-based approaches: In Section 2.3 we categorised continual learning methods into three orthogonal approaches, and in this thesis we focussed on regularisation and rehearsal-based approaches. It would be very interesting to combine our work with probabilistic architecture-based approaches, such as designing priors that enforce pruning, or priors that automatically grow the network size (we could use the Indian Buffet Process (Doshi et al., 2009; Nalisnick and Smyth, 2017; Kessler et al., 2021)). Growing model size efficiently is particularly important when faced with very long streams of data.

Differential Privacy (Dwork et al., 2006): We could use differentially-private pseudo-inputs in FROMP or K-priors to design differentially-private algorithms for sequential learning. This is becoming increasingly relevant as privacy becomes more important.

Federated learning: In Bui et al. (2018) we showed links between continual learning and federated learning. We can use these links to suggest new algorithms for federated learning that also use functional regularisation, based on FROMP or K-priors. However, to maintain security of datapoints, we may want to use pseudo-inputs (this depends on the exact setup in federated learning).

References

Adel, T., Zhao, H., and Turner, R. E. (2020). Continual learning with adaptive weights (claw). In *International Conference on Learning Representations*.

Aitchison, L. (2018). Bayesian filtering unifies adaptive and non-adaptive neural network optimization methods. *arXiv preprint arXiv:1807.07540*.

Aitchison, L. (2021). A statistical theory of cold posteriors in deep neural networks. In *International Conference on Learning Representations*.

Alaoui, A. and Mahoney, M. W. (2015). Fast randomized kernel ridge regression with statistical guarantees. In *Advances in Neural Information Processing Systems*, pages 775–783.

Aljundi, R., Babiloni, F., Elhoseiny, M., Rohrbach, M., and Tuytelaars, T. (2018). Memory aware synapses: Learning what (not) to forget. In *Computer Vision – ECCV 2018*, pages 144–161. Springer International Publishing.

Aljundi, R., Belilovsky, E., Tuytelaars, T., Charlin, L., Caccia, M., Lin, M., and Page-Caccia, L. (2019a). Online continual learning with maximal interfered retrieval. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Aljundi, R., Chakravarty, P., and Tuytelaars, T. (2017). Expert gate: Lifelong learning with a network of experts. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7120–7129.

Aljundi, R., Lin, M., Goujaud, B., and Bengio, Y. (2019b). Gradient based sample selection for online continual learning. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276.

Ao, S., Li, X., and Ling, C. (2017). Fast generalized distillation for semi-supervised domain adaptation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31.

Ash, J. T., Zhang, C., Krishnamurthy, A., Langford, J., and Agarwal, A. (2020). Deep batch active learning by diverse, uncertain gradient lower bounds. *International Conference on Learning Representation*.

Banerjee, A., Merugu, S., Dhillon, I. S., and Ghosh, J. (2005). Clustering with Bregman divergences. *Journal of machine learning research*, 6(Oct):1705–1749.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2015). The arcade learning environment: An evaluation platform for general agents (extended abstract). In *Journal of Artificial Intelligence Research*.

Benjamin, A., Rolnick, D., and Kording, K. (2019). Measuring and regularizing networks in function space. In *International Conference on Learning Representations*.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112:859 – 877.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. In *International Conference on Machine Learning*, pages 1613–1622.

Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223—311.

Bui, T. D., Nguyen, C. V., Swaroop, S., and Turner, R. E. (2018). Partitioned variational inference: A unified framework encompassing federated and continual learning. *arXiv preprint arXiv:1811.11206*.

Burt, D. R., Ober, S. W., Garriga-Alonso, A., and van der Wilk, M. (2020). Understanding variational inference in function-space. *arXiv:2011.09421*. Comment: Presented at the Advances in Approximate Bayesian Inference workshop 2020.

Buzzega, P., Boschini, M., Porrello, A., Abati, D., and Calderara, S. (2020). Dark experience for general continual learning: a strong, simple baseline. In *Advances in Neural Information Processing Systems*, volume 33, pages 15920–15930.

Calandriello, D., Carratino, L., Lazaric, A., Valko, M., and Rosasco, L. (2019). Gaussian process optimization with adaptive sketching: Scalable and no regret. In *Conference on Learning Theory*, pages 533–557. PMLR.

Cauwenberghs, G. and Poggio, T. (2001). Incremental and decremental support vector machine learning. In *Advances in Neural Information Processing Systems*, volume 13. MIT Press.

Cesa-Bianchi, N. and Lugosi, G. (2006). *Prediction, learning, and games*. Cambridge university press.

Chapelle, O. (2007). Training a support vector machine in the primal. *Neural Computation*, 19(5):1155–1178.

Chaudhry, A., Dokania, P. K., Ajanthan, T., and Torr, P. H. S. (2018). Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

Chaudhry, A., Gordo, A., Dokania, P., Torr, P., and Lopez-Paz, D. (2021). Using hindsight to anchor past knowledge in continual learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(8):6993–7001.

Chaudhry, A., Ranzato, M., Rohrbach, M., and Elhoseiny, M. (2019). Efficient lifelong learning with A-GEM. In *International Conference on Learning Representations*.

Cohen, M. B., Musco, C., and Musco, C. (2015). Ridge leverage scores for low-rank approximation. *arXiv preprint arXiv:1511.07263*, 6.

Cook, R. D. (1977). Detection of influential observation in linear regression. *Technometrics*, 19(1):15–18.

Csató, L. and Opper, M. (2002). Sparse on-line Gaussian processes. *Neural computation*, 14(3):641–668.

DeGroot, M. H. and Fienberg, S. E. (1983). The comparison and evaluation of forecasters. *The Statistician: Journal of the Institute of Statisticians*, 32:12–22.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

DeVries, T. and Taylor, G. W. (2018). Learning confidence for out-of-distribution detection in neural networks. *arXiv preprint arXiv:1802.04865*.

Doshi, F., Miller, K., Gael, J. V., and Teh, Y. W. (2009). Variational inference for the indian buffet process. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 137–144. PMLR.

Doucet, A., de Freitas, N., and Gordon, N. J., editors (2001). *Sequential Monte Carlo Methods in Practice*. Statistics for Engineering and Information Science. Springer.

Dua, D. and Graff, C. (2017). UCI machine learning repository.

Duan, H., Li, H., He, G., and Zeng, Q. (2007). Decremental learning algorithms for nonlinear langrangian and least squares support vector machines. In *Proceedings of the First International Symposium on Optimization and Systems Biology (OSB'07)*, pages 358–366. Citeseer.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography*, pages 265–284. Springer Berlin Heidelberg.

Eschenhagen, R. (2019). Natural gradient variational inference for continual learning in deep neural networks. Technical report, University of Osnabruck.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A., Pritzel, A., and Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks. *arXiv:1701.08734*.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135.

Gal, Y. (2016). *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pages 1050–1059.

Ghosal, S. and Van der Vaart, A. (2017). *Fundamentals of nonparametric Bayesian inference*, volume 44. Cambridge University Press.

Ghosh, S., Yao, J., and Doshi-Velez, F. (2019). Model selection in bayesian neural networks via horseshoe priors. *Journal of Machine Learning Research*, 20(182):1–46.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. PMLR.

Golkar, S., Kagan, M., and Cho, K. (2019). Continual learning via neural pruning. In *Neuro-AI Workshop @ NeurIPS*.

Goodfellow, I. (2015). Efficient Per-Example Gradient Computations. *ArXiv e-prints*.

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. (2014). An empirical investigation of catastrophic forgetting in gradient-based neural networks. *International Conference on Learning Representation*.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677.

Graves, A. (2011). Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356.

Guo, C., Pleiss, G., Sun, Y., and Weinberger, K. Q. (2017). On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1321–1330. PMLR.

Hadsell, R., Rao, D., Rusu, A. A., and Pascanu, R. (2020). Embracing Change: Continual Learning in Deep Neural Networks. *Trends in Cognitive Sciences*, 24(12):1028–1040.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hendrycks, D. and Gimpel, K. (2017). A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *International Conference on Learning Representations*.

Herbrich, R., Lawrence, N., and Seeger, M. (2003). Fast sparse gaussian process methods: The informative vector machine. In *Advances in Neural Information Processing Systems*, volume 15. MIT Press.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. (2013). Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347.

Hull, J. (1994). A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):550–554.

Huszár, F. (2018). Note on the quadratic penalties in elastic weight consolidation. *Proceedings of the National Academy of Sciences*, 115(11):E2496–E2497.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

Jung, H., Ju, J., Jung, M., and Kim, J. (2018). Less-forgetful learning for domain expansion in deep neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1).

Kao, T.-C., Jensen, K., Bernacchia, A., and Hennequin, G. (2021). Natural continual learning: success is a journey, not (just) a destination. In *Advances in Neural Information Processing Systems*, volume 34.

Kapoor, S., Karaletsos, T., and Bui, T. D. (2021). Variational auto-regressive gaussian processes for continual learning. In *Proceedings of the 38th International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR.

Karasuyama, M. and Takeuchi, I. (2010). Multiple incremental decremental learning of support vector machines. *IEEE Transactions on Neural Networks*, 21(7):1048–1059.

Kemker, R. and Kanan, C. (2018). Fearnet: Brain-inspired model for incremental learning. In *International Conference on Learning Representations*.

Kessler, S., Nguyen, V., Zohren, S., and Roberts, S. J. (2021). Hierarchical indian buffet neural networks for bayesian continual learning. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*, pages 749–759. PMLR.

Khan, M. E. (2014). Decoupled variational Gaussian inference. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

Khan, M. E., Aravkin, A., Friedlander, M., and Seeger, M. (2013). Fast dual variational inference for non-conjugate latent Gaussian models. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, pages 951–959, Atlanta, Georgia, USA. PMLR.

Khan, M. E., Immer, A., Abedi, E., and Korzepa, M. (2019). Approximate inference turns deep networks into Gaussian processes. *Advances in neural information processing systems*.

Khan, M. E. and Lin, W. (2017). Conjugate-computation variational inference: converting variational inference in non-conjugate models to inferences in conjugate models. In *International Conference on Artificial Intelligence and Statistics*, pages 878–887.

Khan, M. E. and Nielsen, D. (2018). Fast yet simple natural-gradient descent for variational inference in complex models. In *2018 International Symposium on Information Theory and Its Applications (ISITA)*, pages 31–35. IEEE.

Khan, M. E., Nielsen, D., Tangkaratt, V., Lin, W., Gal, Y., and Srivastava, A. (2018). Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2611–2620. PMLR.

Khan, M. E. and Rue, H. (2021). The bayesian learning rule. *arXiv:2107.04562*.

Khan, M. E. and Swaroop, S. (2021). Knowledge-adaptation priors. In *Advances in Neural Information Processing Systems*, volume 34. Curran Associates, Inc.

Khetarpal, K., Riemer, M., Rish, I., and Precup, D. (2020). Towards continual reinforcement learning: A review and perspectives. *arXiv: arXiv:2012.13490*.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Kingma, D. P., Salimans, T., and Welling, M. (2015). Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

Kullback, S. and Leibler, R. A. (1951). On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86.

Kunstner, F., Hennig, P., and Balles, L. (2019). Limitations of the empirical fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems*, volume 32.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338.

Lakshminarayanan, B., Pritzel, A., and Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems 30*, pages 6402–6413. Curran Associates, Inc.

Lange, M. D., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G. G., and Tuytelaars, T. (2021). A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, PP.

Laskov, P., Gehl, C., Krüger, S., Müller, K.-R., Bennett, K. P., and Parrado-Hernández, E. (2006). Incremental support vector learning: Analysis, implementation and applications. *Journal of machine learning research*, 7(9).

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

Lee, K., Lee, H., Lee, K., and Shin, J. (2018). Training confidence-calibrated classifiers for detecting out-of-distribution samples. In *International Conference on Learning Representations*.

Li, Z. and Hoiem, D. (2016). Learning without forgetting. In *Computer Vision - 14th European Conference, ECCV 2016, Proceedings*, pages 614–629. Springer.

Liang, S., Li, Y., and Srikant, R. (2018). Enhancing the reliability of out-of-distribution image detection in neural networks. In *International Conference on Learning Representations*.

Liang, Z. and Li, Y. (2009). Incremental support vector machine learning in the primal and applications. *Neurocomputing*, 72(10):2249–2258.

Lin, W., Schmidt, M., and Khan, M. E. (2020). Handling the positive-definite constraint in the Bayesian learning rule. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 6116–6126. PMLR.

Liu, J. S. and Chen, R. (1998). Sequential monte carlo methods for dynamic systems. *Journal of the American Statistical Association*, 93(443):1032–1044.

Lomonaco, V. and Maltoni, D. (2017). Core50: a new dataset and benchmark for continuous object recognition. In *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 17–26. PMLR.

Loo, N., Swaroop, S., and Turner, R. E. (2020). Combining variational continual learning with FiLM layers. In *LifeLong Learning workshop at ICML 2020*.

Loo, N., Swaroop, S., and Turner, R. E. (2021). Generalized variational continual learning. In *International Conference on Learning Representations*.

Lopez-Paz, D., Bottou, L., Schölkopf, B., and Vapnik, V. (2016). Unifying distillation and privileged information. *arXiv preprint arXiv:1511.03643*.

Lopez-Paz, D. and Ranzato, M. (2017). Gradient episodic memory for continual learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6470–6479, Red Hook, NY, USA. Curran Associates Inc.

Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Louizos, C., Ullrich, K., and Welling, M. (2017). Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Ma, P., Mahoney, M. W., and Yu, B. (2015). A statistical perspective on algorithmic leveraging. *Journal of Machine Learning Research*, 16(27):861–911.

MacKay, D. J. C. (1992). A Practical Bayesian Framework for Backpropagation Networks. *Neural Computation*, 4(3):448–472.

Mackay, D. J. C. (1995). Probable networks and plausible predictions — a review of practical bayesian methods for supervised neural networks. *Network: Computation in Neural Systems*, 6(3):469–505.

Mahalanobis, P. C. (1936). On the generalized distance in statistics. *Proceedings of the National Institute of Sciences (Calcutta)*, 2:49–55.

Mallya, A. and Lazebnik, S. (2018). Packnet: Adding multiple tasks to a single network by iterative pruning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7765–7773.

Maltoni, D. and Lomonaco, V. (2019). Continuous learning in single-incremental-task scenarios. *Neural Networks*, 116.

Martens, J. (2020). New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, 21(146):1–76.

Martens, J. and Grosse, R. (2015). Optimizing neural networks with Kronecker-factored approximate curvature. In *International Conference on Machine Learning*, pages 2408–2417.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press.

Naeini, M. P., Cooper, G. F., and Hauskrecht, M. (2015). Obtaining well calibrated probabilities using bayesian binning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2901–2907. AAAI Press.

Nalisnick, E. and Smyth, P. (2017). Stick-breaking variational autoencoders. In *International Conference on Learning Representations*.

Neal, R. M. (1995). *Bayesian learning for neural networks*. PhD thesis, University of Toronto.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*.

Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. (2018). Variational continual learning. In *International Conference on Learning Representations*.

Nielsen, F. (2021). On a variational definition for the jensen-shannon symmetrization of distances based on the information radius. *Entropy*, 23(4):464.

Opper, M. and Archambeau, C. (2009). The variational Gaussian approximation revisited. *Neural Computation*, 21(3):786–792.

Osawa, K., Swaroop, S., Khan, M. E., Jain, A., Eschenhagen, R., Turner, R. E., and Yokota, R. (2019). Practical deep learning with Bayesian principles. *Advances in neural information processing systems*.

Osawa, K., Tsuji, Y., Ueno, Y., Naruse, A., Yokota, R., and Matsuoka, S. (2018). Second-order optimization method for large mini-batch: Training ResNet-50 on ImageNet in 35 epochs. *CoRR*, abs/1811.12019.

Ostapenko, O., Puscas, M. M., Klein, T., Jähnichen, P., and Nabi, M. (2019). Learning to remember: A synaptic plasticity driven framework for continual learning. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11313–11321.

Pan, P., Swaroop, S., Immer, A., Eschenhagen, R., Turner, R., and Khan, M. E. (2020). Continual deep learning by functional regularisation of memorable past. In *Advances in Neural Information Processing Systems*, volume 33, pages 4453–4464. Curran Associates, Inc.

Parisi, G., Kemker, R., Part, J., Kanan, C., and Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71.

Parisi, G., Tani, J., Weber, C., and Wermter, S. (2018). Lifelong learning of spatiotemporal representations with dual-memory recurrent self-organization. *Frontiers in Neurorobotics*, 12.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Perez, E., Strub, F., de Vries, H., Dumoulin, V., and Courville, A. C. (2018). Film: Visual reasoning with a general conditioning layer. In *AAAI*.

Rasmussen, C. and Williams, C. (2006). *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA.

Ratcliff, R. (1990). Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, pages 285–308.

Rebuffi, S.-A., Kolesnikov, A., Sperl, G., and Lampert, C. H. (2017). iCaRL: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2001–2010.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286.

Riemer, M., Cases, I., Ajemian, R., Liu, M., Rish, I., Tu, Y., , and Tesauro, G. (2019). Learning to learn without forgetting by maximizing transfer and minimizing interference. In *International Conference on Learning Representations*.

Ritter, H., Botev, A., and Barber, D. (2018). Online structured laplace approximations for overcoming catastrophic forgetting. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Robins, A. (1995). Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146.

Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T., and Wayne, G. (2019). Experience replay for continual learning. In *Advances in Neural Information Processing Systems*, volume 32.

Romero, E., Barrio, I., and Belanche, L. (2007). Incremental and decremental learning for linear support vector machines. In *International Conference on Artificial Neural Networks*, pages 209–218. Springer.

Ruder, S., Ghaffari, P., and Breslin, J. G. (2017). Knowledge adaptation: Teaching to adapt. *arXiv preprint arXiv:1702.02052*.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv:1606.04671*.

Sarafianos, N., Vrigkas, M., and Kakadiaris, I. A. (2017). Adaptive SVM+: Learning with privileged information for domain adaptation. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 2637–2644.

Särkkä, S., Solin, A., and Hartikainen, J. (2013). Spatiotemporal learning via infinite-dimensional Bayesian filtering and smoothing: A look at Gaussian process regression through Kalman filtering. *IEEE Signal Processing Magazine*, 30(4):51–61.

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738.

Schwarz, J., Czarnecki, W., Luketina, J., Grabska-Barwinska, A., Teh, Y. W., Pascanu, R., and Hadsell, R. (2018). Progress & compress: A scalable framework for continual learning. In *International Conference on Machine Learning*, pages 4528–4537. PMLR.

Serra, J., Suris, D., Miron, M., and Karatzoglou, A. (2018). Overcoming catastrophic forgetting with hard attention to the task. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4548–4557. PMLR.

Sharma, M., Hutchinson, M., Siddharth Swaroop, A. H., and Turner, R. E. (2019). Differentially private federated variational inference. *arXiv:1911.10563*. Presented at PriML at NeurIPS 2019.

Shin, H., Lee, J. K., Kim, J., and Kim, J. (2017). Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems*, volume 30.

Smola, A. J., Vishwanathan, S., and Eskin, E. (2004). Laplace propagation. In *Advances in Neural Information Processing Systems*.

Solin, A., Hensman, J., and Turner, R. E. (2018). Infinite-horizon Gaussian processes. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Stanford (2021). Cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/. Accessed: Jan. 1, 2022.

Sun, S., Zhang, G., Shi, J., and Grosse, R. (2019). Functional variational bayesian neural networks. *International Conference on Learning Representation*.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147.

Swaroop, S., Nguyen, C. V., Bui, T. D., and Turner, R. E. (2019). Improving and understanding variational continual learning. *arXiv preprint arXiv:1905.02099*.

Swaroop, S. and Turner, R. E. (2017). Understanding Expectation Propagation. In *Advances in Approximate Bayesian Inference workshop at NIPS 2017*.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-RMSprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning 4*.

Titsias, M. K. (2009). Variational learning of inducing variables in sparse gaussian processes. In *Artificial intelligence and statistics*, pages 567–574. PMLR.

Titsias, M. K., Schwarz, J., de G. Matthews, A. G., Pascanu, R., and Teh, Y. W. (2020). Functional regularisation for continual learning with gaussian processes. In *International Conference on Machine Learning*.

Tomczak, M., Swaroop, S., Foong, A., and Turner, R. (2021). Collapsed variational bounds for bayesian neural networks. In *Advances in Neural Information Processing Systems*, volume 34. Curran Associates, Inc.

Tomczak, M., Swaroop, S., and Turner, R. (2020). Efficient low rank gaussian variational inference for neural networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 4610–4622. Curran Associates, Inc.

Tomczak, M. B., Swaroop, S., and Turner, R. E. (2018). Neural network ensembles and variational inference revisited. In *Advances in Approximate Bayesian Inference Symposium*.

Train, K. E. (2009). *Discrete choice methods with simulation*. Cambridge university press.

Trippe, B. and Turner, R. E. (2018). Overpruning in Variational Bayesian Neural Networks. *arXiv:1801.06230 [stat]*. Presented at the Advances in Approximate Bayesian Inference workshop at NIPS 2017.

Tsai, C.-H., Lin, C.-Y., and Lin, C.-J. (2014). Incremental and decremental training for linear classification. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 343–352. ACM.

Tseran, H., Khan, M. E., Bui, T., and Harada, T. (2018). Natural variational continual learning. In *NeurIPS Workshop on Continual Learning*.

Turner, R. E. and Sahani, M. (2011). Two problems with variational expectation maximisation for time-series models. *Bayesian Time series models*, pages 115–138.

Tveit, A., Hetland, M. L., and Engum, H. (2003). Incremental and decremental proximal support vector classification using decay coefficients. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 422–429. Springer.

van de Ven, G. M., Siegelmann, H. T., and Tolias, A. S. (2020). Brain-inspired replay for continual learning with artificial neural networks. *Nature Communications*, 11(1):4069.

van de Ven, G. M. and Tolias, A. S. (2019). Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*.

Vapnik, V. and Izmailov, R. (2015). Learning using privileged information: similarity control and knowledge transfer. *Journal of Machine Learning Research*, 16(1):2023–2049.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57.

Vovk, V. G. (1990). Aggregating strategies. *Proc. of Computational Learning Theory, 1990*.

Wah, C., Branson, S., Welinder, P., Perona, P., and Belongie, S. (2011). The Caltech-UCSD Birds-200-2011 Dataset. Technical Report CNS-TR-2011-001, California Institute of Technology.

Wang, D. and Shang, Y. (2014). A new active labeling method for deep learning. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 112–119.

Wang, T., Zhu, J.-Y., Torralba, A., and Efros, A. A. (2018). Dataset distillation. *arXiv preprint arXiv:1811.10959*.

Wenzel, F., Roth, K., Veeling, B., Swiatkowski, J., Tran, L., Mandt, S., Snoek, J., Salimans, T., Jenatton, R., and Nowozin, S. (2020). How good is the Bayes posterior in deep neural networks really? In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10248–10259. PMLR.

Wilson, A. G., Izmailov, P., Hoffman, M., Gal, Y., Li, Y., Pradier, M. F., Vikram, S., Foong, A., Lotfi, S., and Farquhar, S. (2021). Approximate inference in bayesian deep learning. https://izmailovpavel.github.io/neurips_bdl_competition/. Accessed: Jan. 1, 2022.

Winn, J. and Bishop, C. M. (2005). Variational message passing. *Journal of Machine Learning Research*, 6(Apr):661–694.

Wortsman, M., Ramanujan, V., Liu, R., Kembhavi, A., Rastegari, M., Yosinski, J., and Farhadi, A. (2020). Supermasks in superposition. In *Advances in Neural Information Processing Systems*, volume 33, pages 15173–15184. Curran Associates, Inc.

Xu, J. and Zhu, Z. (2018). Reinforced continual learning. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Yoon, J., Yang, E., Lee, J., and Hwang, S. J. (2018). Lifelong learning with dynamically expandable networks. In *International Conference on Learning Representations*.

Yu, F., Zhang, Y., Song, S., Seff, A., and Xiao, J. (2015). LSUN: construction of a large-scale image dataset using deep learning with humans in the loop. *CoRR*, abs/1506.03365.

Zeng, G., Chen, Y., Cui, B., and Yu, S. (2019). Continual learning of context-dependent processing in neural networks. *Nature Machine Intelligence*, 1:364–372.

Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995. PMLR.

Zhang, C., Bütepage, J., Kjellström, H., and Mandt, S. (2019). Advances in variational inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(8):2008–2026.

Zhang, G., Sun, S., Duvenaud, D. K., and Grosse, R. B. (2018). Noisy natural gradient as variational inference. *arXiv preprint arXiv:1712.02390*.

Zhao, B., Mopuri, K. R., and Bilen, H. (2021). Dataset condensation with gradient matching. *International Conference on Learning Representation*.

Appendix A

Details on weight-space variational continual learning experiments

In this Appendix, we provide further details on the weight-space variational continual learning experiments in Chapter 3. In Appendix A.1 (and Figure A.1) we provide plots of weights showing pruning in variational Bayesian neural networks. In Appendix A.2 we give hyperparameters for the VOGN results in Section 3.2.3 (from Eschenhagen (2019)). In Appendix A.3 we give hyperparameters for our Toy-Gaussian experiments in Section 3.3.

A.1 Pruning on MNIST

Figure A.1 shows that only one unit is active, with all other 199 units pruned out, when training a one-hidden-layer variational Bayesian neural network to classify between 0 and 1 digits in MNIST. As discussed in Section 3.1.2, such pruning can be detrimental in some settings, but it can also be useful in continual learning. The first column of Figure 3.3 plots this single active unit.

A.2 Hyperparameters for VOGN continual learning experiments

This section gives the hyperparameters for the VOGN continual learning experiments in Section 3.2.3. These are taken from Eschenhagen (2019).

Split MNIST. We use a one-hidden layer multi-layer perceptron (MLP) with 200 hidden units and ReLU activation functions, like the experiment with Improved VCL in Section 3.1.1. VOGN is run for 100 epochs per task (compared to Improved VCL's 600 epochs per task).

Parameters are initialised before training with the default PyTorch initialisation for linear layers. The initial precision is $1e6$. A standard normal initial prior is used, like in VCL. Between tasks, the mean and precision are initialised in the same way as for the first task. We use learning rate $\alpha = 0.001$, batch size $M = 256$, $\beta_1 = 0$ (no momentum), $\beta_2 = 0.001$, and 10 Monte-Carlo (MC) samples are used during training and 100 for testing. We do not use tempering ($\tau = 1$ always), data augmentation ($\rho = 1$) or external damping factor ($\gamma = 0$).

Permuted MNIST (10 tasks). We use a two-hidden layer MLP with 100 hidden units in each layer and ReLU activation functions (as described in Section 2.4). All hyperparameters are the same as the Split MNIST experiment, including number of epochs, learning rate, initialisation, and so on. We only need 100 epochs instead of Improved VCL's 800 epochs per task.

Split CIFAR. We use the same CifarNet architecture from Zenke et al. (2017) and described in Section 2.4. We run the first task for 60 epochs, and remaining tasks for 600 epochs, as the first task has ten times more data than the other tasks (compared to Improved VCL's 500 epochs for the first task and 5000 for the remaining tasks). We use an initial precision of $4e3$ for the first task, and $4e4$ for the other tasks. In contrast to the other experiments, we do not reset the means of the weights between tasks, and initialise at the previous posterior's means. We use a zero-mean prior with prior precision $\delta = 120$. We use learning rate $\alpha = 0.001$, batch size $M = 256$, momentum rate $\beta_1 = 0.9$, $\beta_2 = 1e-4$, and 10 MC samples are used during training and 100 for testing. We do not use tempering ($\tau = 1$ always), data augmentation ($\rho = 1$) or external damping factor ($\gamma = 0$).

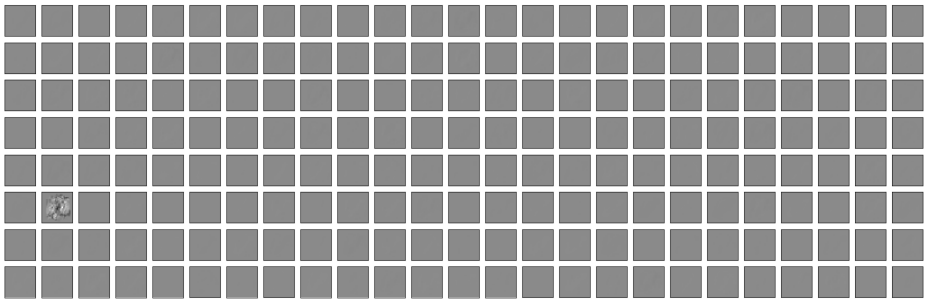
A.3 Hyperparameters for Toy-Gaussians experiments

This section gives the hyperparameters for the Toy-Gaussians experiments in Section 3.3. We found these values by running for 5 runs and picking the values with largest average train accuracy.

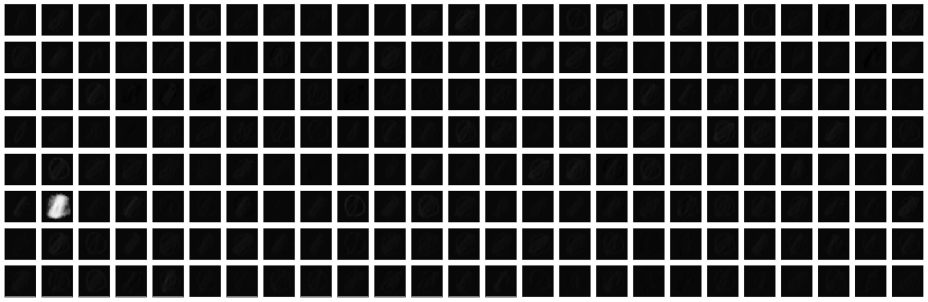
VCL(+Coreset). We use number of epochs $= 200$, number of coreset epochs $= 200$, a standard normal prior (precision $= 1$), batch size $= 40$ and (Adam) learning rate$= 0.01$.

Joint Tasks. We use number of epochs $= 50$, batch size $= 20$, learning rate $= 0.01$.

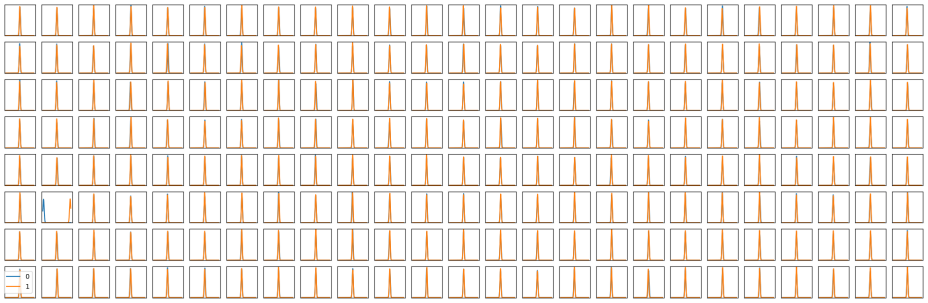
VOGN. We use number of epochs $= 200$, a standard normal prior (precision $= 1$), batch size $= 40$, learning rate $\alpha = 0.01$, $\beta_2 = 0.001$, and initial precision $1e6$.



(a) Hidden layer input means to each of the 200 hidden units.



(b) Hidden layer input variances to each of the 200 hidden units.



(c) Hidden layer output weights from each of the 200 hidden units.

Figure A.1: These plots show the input means and variances of weights into each node, as well as output weights, for a single-hidden layer multi-layer perceptron with 200 hidden units trained to classify between the digits 0 and 1 in MNIST (this is the first task in Split MNIST). We use the Bayes-By-Backprop method (Blundell et al., 2015) to train mean-field Gaussians over each weight, with our techniques to improve convergence rate (described in Section 3.1.1). We see that only one hidden unit is active, and all other 199 units are pruned out, with output weights at zero-mean small-variance Gaussians, and input weights at the prior.

Appendix B

Details on batch VOGN experiments

In this Appendix, we provide further details on VOGN and batch VOGN experiments in Section 3.2.2. This includes details on implementation of VOGN and hyperparameters for non-continual learning experiments (hyperparameters for VOGN continual learning experiments are in Appendix A). Table B.1 provides standard deviations for VOGN batch experiments in Table 3.3. In Appendix B.1 we provide details on how we implement the Gauss-Newton approximation in PyTorch in a computationally-efficient manner. In Appendix B.2 we provide hyperparameters for all batch VOGN experiments in Section 3.2.2. In Appendix B.3 we provide results of how the prior variance δ and dataset size reweighting factor ρ interact with each other. Appendix B.4 shows that MC-dropout is highly sensitive to dropout rate. Appendix B.5 provides further details on the uncertainty metrics we use to compare methods in Table 3.3. Appendix B.6 provides more out-of-distribution results with VOGN, further to the single result reported in Section 3.2.2.

Dataset/ Architecture	Optimiser	Train Acc (%)	Train NLL	Validation Acc (%)	Validation NLL	ECE	AUROC	Epochs	Time (s) / epoch
CIFAR-10/ LeNet-5 (no DA)	Adam	71.98 ± 0.117	0.733 ± 0.021	**67.67 ± 0.513**	**0.937 ± 0.012**	**0.021 ± 0.002**	0.794 ± 0.001	210	6.96
	BBB	66.84 ± 0.003	0.957 ± 0.006	64.61 ± 0.331	1.018 ± 0.006	0.045 ± 0.005	0.784 ± 0.003	800	11.43
	MC-dropout	68.41 ± 0.581	0.870 ± 0.101	**67.65 ± 1.317**	0.99 ± 0.026	0.087 ± 0.009	**0.797 ± 0.006**	210	6.95
	VOGN	70.79 ± 0.763	0.880 ± 0.02	**67.32 ± 1.310**	**0.938 ± 0.024**	0.046 ± 0.002	**0.8 ± 0.002**	210	18.33
CIFAR-10/ AlexNet (no DA)	Adam	100.0 ± 0	0.001 ± 0	67.94 ± 0.537	2.83 ± 0.02	0.262 ± 0.005	0.793 ± 0.001	161	3.12
	MC-dropout	97.56 ± 0.278	0.058 ± 0.014	**72.20 ± 0.177**	1.077 ± 0.012	0.140 ± 0.004	**0.818 ± 0.002**	160	3.25
	VOGN	79.07 ± 0.248	0.696 ± 0.020	69.03 ± 0.419	**0.931 ± 0.017**	**0.024 ± 0.010**	0.796 ± 0	160	9.98
CIFAR-10/ AlexNet	Adam	97.92 ± 0.140	0.057 ± 0.006	73.59 ± 0.296	1.480 ± 0.015	0.262 ± 0.005	0.793 ± 0.001	161	3.08
	MC-dropout	80.65 ± 0.615	0.47 ± 0.052	**77.04 ± 0.343**	**0.667 ± 0.012**	0.114 ± 0.002	0.828 ± 0.002	160	3.20
	VOGN	81.15 ± 0.259	0.511 ± 0.039	75.48 ± 0.478	0.703 ± 0.006	**0.016 ± 0.001**	**0.832 ± 0.002**	160	10.02
CIFAR-10/ ResNet-18	Adam	97.74 ± 0.140	0.059 ± 0.012	**86.00 ± 0.257**	0.55 ± 0.01	0.082 ± 0.002	**0.877 ± 0.001**	160	11.97
	MC-dropout	88.23 ± 0.243	0.317 ± 0.045	82.85 ± 0.208	0.51 ± 0	0.166 ± 0.025	0.768 ± 0.004	161	12.51
	VOGN	91.62 ± 0.07	0.263 ± 0.051	84.27 ± 0.195	**0.477 ± 0.006**	**0.040 ± 0.002**	**0.876 ± 0.002**	161	53.14
ImageNet/ ResNet-18	SGD	82.63 ± 0.058	0.675 ± 0.017	**67.79 ± 0.017**	**1.38 ± 0**	0.067	0.856	90	44.13
	Adam	80.96 ± 0.098	0.723 ± 0.015	66.39 ± 0.168	1.44 ± 0.01	0.064	0.855	90	44.40
	MC-dropout	72.96	1.12	65.64	1.43	**0.012**	0.856	90	45.86
	OGN	85.33 ± 0.057	0.526 ± 0.005	65.76 ± 0.115	1.60 ± 0.00	0.128 ± 0.004	0.8543 ± 0.001	90	63.13
	VOGN	73.87 ± 0.061	1.02 ± 0.01	**67.38 ± 0.263**	**1.37 ± 0.01**	0.0293 ± 0.001	0.8543 ± 0	90	76.04
	K-FAC	83.73 ± 0.058	0.571 ± 0.016	66.58 ± 0.176	1.493 ± 0.006	0.158 ± 0.005	0.842 ± 0.005	60	133.69
	Noisy K-FAC	72.28	1.075	66.44	1.44	0.080	0.852	60	179.27

Table B.1: Comparing optimisers on different dataset/architecture combinations. Mean performance and standard deviation over three runs. This is a repeat of Table 3.3 except with standard deviations included. DA: Data Augmentation, Acc: Accuracy (higher is better), NLL: Negative Log-Likelihood (lower is better), ECE: Expected Calibration Error (lower is better), AUROC: Area Under ROC curve (higher is better), BBB: Bayes By Backprop. For ImageNet results, the reported accuracy and negative log-likelihood are the median value from the final 5 epochs.

B.1 Details on fast implementation of the Gauss-Newton approximation

In this section we show how we efficiently compute the Gauss-Newton (GN) approximation in current codebases such as PyTorch, which are only optimised to directly return the average of gradients over the minibatch. In order to efficiently compute the Gauss-Newton approximation, we modify the backward-pass to efficiently calculate the gradient per example in the minibatch, and extend the solution in Goodfellow (2015) to both convolutional and batch normalisation layers.

Convolutional layer

Consider a convolutional layer with a weight matrix $\boldsymbol{W} \in \mathbb{R}^{C_{out} \times C_{in} k^2}$ (ignore bias for simplicity) and an input tensor $\boldsymbol{A} \in \mathbb{R}^{C_{in} \times H_{in} \times W_{in}}$, where C_{out}, C_{in} are the number of output and input channels respectively, H_{in}, W_{in} are the spatial dimensions, and k is the kernel size. For any stride and padding, by applying `torch.nn.functional.unfold` function in PyTorch[1], we get the extended matrix $\boldsymbol{M}_A \in \mathbb{R}^{C_{in} k^2 \times H_{out} W_{out}}$ so that the output tensor $\boldsymbol{S}$ is calculated by a matrix multiplication,

$$\boldsymbol{M}_A \leftarrow \text{unfold}\left(\boldsymbol{A}\right) \in \mathbb{R}^{C_{in} k^2 \times H_{out} W_{out}}, \tag{B.1}$$

$$\boldsymbol{M}_S \leftarrow \boldsymbol{W} \boldsymbol{M}_A \in \mathbb{R}^{C_{out} \times H_{out} W_{out}}, \tag{B.2}$$

$$\boldsymbol{S} \leftarrow \text{reshape}\left(\boldsymbol{M}_S\right) \in \mathbb{R}^{C_{out} \times H_{out} \times W_{out}}, \tag{B.3}$$

where H_{out}, W_{out} are the spatial dimensions of the output feature map. Using the matrix $\boldsymbol{M}_A$, we can also get the gradient per example by a matrix multiplication,

$$\nabla_{M_S} \ell(y_i, f_W(\boldsymbol{x}_i)) \leftarrow \text{reshape}\left(\nabla_S \ell(y_i, f_W(\boldsymbol{x}_i))\right) \in \mathbb{R}^{C_{out} \times H_{out} W_{out}}, \tag{B.4}$$

$$\nabla_W \ell(y_i, f_W(\boldsymbol{x}_i)) \leftarrow \nabla_{M_S} \ell(y_i, f_W(\boldsymbol{x}_i)) \boldsymbol{M}_A^\top \in \mathbb{R}^{C_{out} \times C_{in} k^2}. \tag{B.5}$$

Note that in PyTorch, we have access to the inputs $\boldsymbol{A}$ and the gradient $\nabla_S \ell(y_i, f_W(\boldsymbol{x}_i))$ per example in the computation graph during a forward-pass and a backward-pass respectively, by using the `Function Hooks`[2]. Hence, to get the gradient $\nabla_W \ell(y_i, f_W(\boldsymbol{x}_i))$ per example, we only need to perform Equations B.1, B.4 and B.5 after the backward-pass for this layer.

[1] https://pytorch.org/docs/stable/nn.functional.html#torch.nn.functional.unfold

[2] https://pytorch.org/tutorials/beginner/former_torchies/nnft_tutorial.html#forward-and-backward-function-hooks

Batch normalisation layer

Consider a batch normalisation layer that follows a fully-connected layer, whose activation is $\boldsymbol{a} \in \mathbb{R}^d$, with scale parameter $\boldsymbol{\gamma} \in \mathbb{R}^d$ and shift parameter $\boldsymbol{\beta} \in \mathbb{R}^d$. We get the output of this batch normalisation layer $\boldsymbol{s} \in \mathbb{R}^d$ by,

$$\boldsymbol{\mu} \leftarrow E\left[\boldsymbol{a}\right] \in \mathbb{R}^d\,, \tag{B.6}$$

$$\boldsymbol{\sigma}^2 \leftarrow E\left[(\boldsymbol{a}-\boldsymbol{\mu})^2\right] \in \mathbb{R}^d\,, \tag{B.7}$$

$$\hat{\boldsymbol{a}} \leftarrow \frac{\boldsymbol{a}-\boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2}} \in \mathbb{R}^d\,, \tag{B.8}$$

$$\boldsymbol{s} \leftarrow \boldsymbol{\gamma}\hat{\boldsymbol{a}} + \boldsymbol{\beta} \in \mathbb{R}^d\,, \tag{B.9}$$

where $E\left[\cdot\right]$ is the average over the minibatch and $\hat{\boldsymbol{a}}$ is the normalised input. We can find the gradient with respect to parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ per example by,

$$\nabla_\gamma \ell(y_i, f_W(\boldsymbol{x}_i) \leftarrow \nabla_s \ell(y_i, f_W(\boldsymbol{x}_i) \circ \hat{\boldsymbol{a}}\,, \tag{B.10}$$

$$\nabla_\beta \ell(y_i, f_W(\boldsymbol{x}_i) \leftarrow \nabla_s \ell(y_i, f_W(\boldsymbol{x}_i)\,. \tag{B.11}$$

We can obtain the input $\boldsymbol{a}$ and the gradient $\nabla_s \ell(y_i, f_W(\boldsymbol{x}_i)$ per example from the computation graph in PyTorch in the same way as a convolutional layer.

Layer-wise block-diagonal Gauss-Newton approximation

Despite using the method above, it is still intractable to compute the Gauss-Newton matrix (and its inverse) with respect to the weights of large-scale deep neural networks. We therefore apply two further approximations (summarised in Figure B.1). First, we view the Gauss-Newton matrix as a layer-wise block-diagonal matrix. This corresponds to ignoring the correlation between the weights of different layers. Hence for a network with L layers, there are L diagonal blocks, and $\mathbf{H}_\ell$ is the diagonal block corresponding to the ℓ-th layer ($\ell = 1, \ldots, L$). Second, we approximate each diagonal block $\mathbf{H}_\ell$ with $\tilde{\mathbf{H}}_\ell$, which is either a Kronecker-factored or diagonal matrix. Using a Kronecker-factored matrix as $\tilde{\mathbf{H}}_\ell$ corresponds to K-FAC, and a diagonal matrix corresponds to a mean-field approximation in that layer. By applying these two approximations, the update rule of the Gauss-Newton method can be written in a layer-wise fashion,

$$\boldsymbol{W}_{\ell,t+1} = \boldsymbol{W}_{\ell,t} - \alpha_t \tilde{\mathbf{H}}_\ell(\boldsymbol{\theta}_t)^{-1} \mathbf{g}_\ell(\boldsymbol{\theta}_t) \;\; (\ell = 1, \ldots, L)\,, \tag{B.12}$$

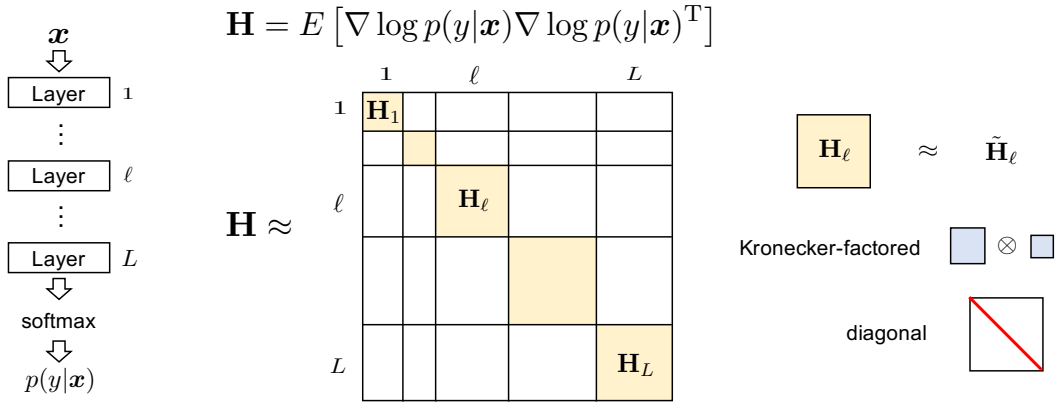


Figure B.1: Layer-wise block-diagonal Gauss-Newton approximation

where $\boldsymbol{W}_\ell$ is the weights in ℓ-th layer, and

$$\boldsymbol{\theta} = \left(\begin{array}{ccccc} \mathrm{vec}(\mathrm{W}_1)^{\mathrm{T}} & \cdots & \mathrm{vec}(\mathrm{W}_\ell)^{\mathrm{T}} & \cdots & \mathrm{vec}(\mathrm{W}_\mathrm{L})^{\mathrm{T}} \end{array} \right)^{\mathrm{T}}. \tag{B.13}$$

Since the cost of computing $\tilde{\mathbf{H}}_\ell^{-1}$ is much cheaper than computing $\mathbf{H}^{-1}$, our approximations make Gauss-Newton much more practical in deep learning.

In the distributed setting (see Figure 3.6), each parallel process (corresponding to 1 GPU) calculates the GN matrix for its local minibatch. Then, one GPU adds them together and calculates the inverse. This inversion step can also be parallelised after making the block-diagonal approximation to the GN matrix. After inverting the GN matrix, the standard deviation $\boldsymbol{\sigma}$ is updated (line 9 in Algorithm 1), and sent to each parallel process, allowing each process to draw independently from the posterior.

In the Noisy K-FAC case, a similar distributed scheme is used. When using K-FAC approximations to the Gauss-Newton blocks for other layers, Osawa et al. (2018) empirically showed that the BatchNorm layer can be approximated with a diagonal matrix without loss of accuracy, and we find the same. We therefore use diagonal $\tilde{\mathbf{H}}_\ell$ with K-FAC and Noisy K-FAC in BatchNorm layers (see Table B.2). For further details on how to efficiently parallelise K-FAC in the distributed setting, please see Osawa et al. (2018).

B.2 Hyperparameter values for batch VOGN experiments

In this section, we give hyperparameter values for batch VOGN experiments. Hyperparameters for all results shown in Table B.1 are given in Table B.3. The settings for distributed VI training are given in Table B.4. Please see Goyal et al. (2017) and Osawa et al. (2018) for best practice on these hyperparameter values.

Optimiser	convolution	fully-connected	Batch Normalisation
OGN	diagonal	diagonal	diagonal
VOGN	diagonal	diagonal	diagonal
K-FAC	Kronecker-factored	Kronecker-factored	diagonal
Noisy K-FAC	Kronecker-factored	Kronecker-factored	diagonal

Table B.2: The approximation used for each layer type's diagonal block $\tilde{\mathbf{H}}_\ell$ for the different optimisers tested this paper.

Bayes by Backprop for CIFAR-10/LeNet-5 training

We use the training procedure and hyperparameter settings for Bayes by Backprop (BBB) (Blundell et al., 2015) as suggested in our Improved VCL experiments in Section 3.1.1. This includes using the local reparameterisation trick, initialising means and variances at small values, using 10 MC samples per minibatch during training for linear layers (1 MC sample per minibatch for convolutional layers) and 100 MC samples per minibatch during testing for linear layers (10 MC samples per minibatch for convolutional layers). Note that BBB has twice as many parameters to optimise than Adam or SGD (it separately optimises means and variances of each weight in the deep neural network). The fewer MC samples per minibatch for convolutional layers speed up training time per epoch while empirically not reducing convergence rate.

B.3 Effect of prior variance and dataset size reweighting factor

Figure B.2 shows the combined effect of the dataset reweighting factor ρ and prior precision δ when training VOGN on ResNet-18 on ImageNet. When ρ is set to a value in the correct order of magnitude, it does not affect performance much; instead, we should tune δ. This is our methodology when dealing with ρ. Note that we set ρ for ImageNet to be smaller than that for CIFAR-10 because the data augmentation cropping step uses a higher portion of the initial image than in CIFAR-10: we crop images of size 224×224 from images of size 256×256.

Dataset/ Architecture	Optimiser	α_{init}	α	Epochs to decay α	β_1	β_2	Weight decay	L_2 reg
CIFAR-10/ LeNet-5 (no DA)	Adam	-	1e-3	-	0.1	0.001	1e-2	-
	BBB	-	1e-3	-	0.1	0.001	-	-
	MC-dropout	-	1e-3	-	0.9	-	-	1e-4
	VOGN	-	1e-2	-	0.9	0.999	-	-
CIFAR-10/ AlexNet (no DA)	Adam	-	1e-3	[80, 120]	0.1	0.001	1e-4	-
	MC-dropout	-	1e-1	[80, 120]	0.9	-	-	1e-4
	VOGN	-	1e-4	[80, 120]	0.9	0.999	-	-
CIFAR-10/ AlexNet	Adam	-	1e-3	[80, 120]	0.1	0.001	1e-4	-
	MC-dropout	-	1e-1	[80, 120]	0.9	-	-	1e-4
	VOGN	-	1e-4	[80, 120]	0.9	0.999	-	-
CIFAR-10/ ResNet-18	Adam	-	1e-3	[80, 120]	0.1	0.001	5e-4	-
	MC-dropout	-	1e-1	[80, 120]	0.9	-	-	1e-4
	VOGN	-	1e-4	[80, 120]	0.9	0.999	-	-
ImageNet/ ResNet-18	SGD	1.25e-2	1.6	[30, 60, 80]	0.9	-	-	1e-4
	Adam	1.25e-5	1.6e-3	[30, 60, 80]	0.1	0.001	1e-4	-
	MC-dropout	1.25e-2	1.6	[30, 60, 80]	0.9	-	-	1e-4
	OGN	1.25e-5	1.6e-3	[30, 60, 80]	0.9	0.9	-	1e-5
	VOGN	1.25e-5	1.6e-3	[30, 60, 80]	0.9	0.999	-	-
	K-FAC	1.25e-5	1.6e-3	[15, 30, 45]	0.9	0.9	-	1e-4
	Noisy K-FAC	1.25e-5	1.6e-3	[15, 30, 45]	0.9	0.9	-	-

Table B.3: This table gives hyperparameters for all results in Table B.1 for all methods. See Table 3.2 and Algorithm 1 for definitions of the terms.

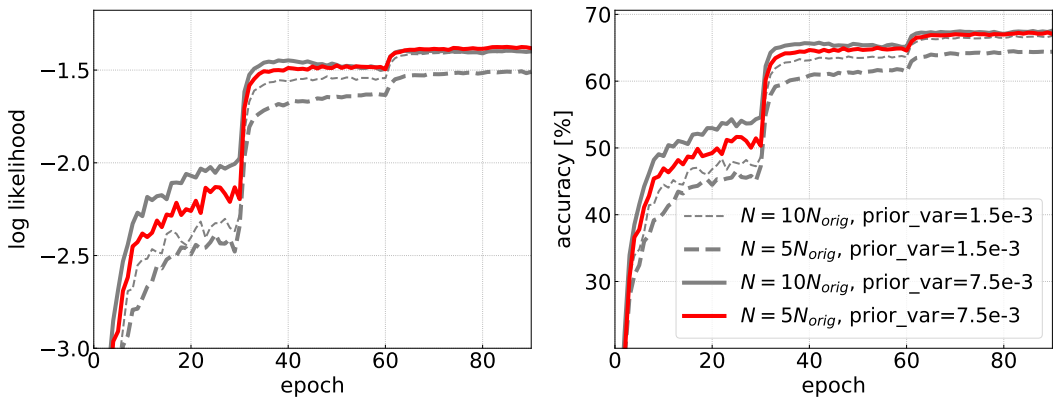


Figure B.2: Effect of changing the dataset size reweighting factor ρ and prior variance δ when training VOGN on ResNet-18 on ImageNet. It suffices to set ρ to the correct order of magnitude, and then tune δ.

Optimiser	Dataset/ Architecture	M	# GPUs	K	τ	ρ	N_{orig}	δ	$\tilde{\delta}$	γ
VOGN	CIFAR-10/ LeNet-5 (no DA)	128	4	6	0.1 $\rightarrow$ 1	1	50,000	100	2e-4 $\rightarrow$ 2e-3	1e-3
	CIFAR-10/ AlexNet (no DA)	128	8	3	0.05 $\rightarrow$ 1	1	50,000	0.5	5e-7 $\rightarrow$ 1e-5	1e-3
	CIFAR-10/ AlexNet	128	8	3	0.5 $\rightarrow$ 1	10	50,000	0.5	5e-7 $\rightarrow$ 1e-5	1e-3
	CIFAR-10/ ResNet-18	256	8	5	1	10	50,000	50	1e-3	1e-3
	ImageNet/ ResNet-18	4096	128	1	1	5	1,281,167	133.3	2e-5	1e-4
Noisy K-FAC	ImageNet/ ResNet-18	4096	128	1	1	5	1,281,167	133.3	2e-5	1e-4

Table B.4: Hyperparameter settings for distributed settings. See Table 3.2 and Algorithm 1 for definitions of the terms.

B.4 MC-dropout's sensitivity to dropout rate

In this section, we show MC-dropout's sensitivity to dropout rate p. We tune MC-dropout as best as we can, finding that $p = 0.1$ works best for all architectures trained on CIFAR-10 (see Figure B.3 for the dropout rate's sensitivity on LeNet-5 as an example). On ResNet-18 trained on ImageNet, we find that MC-dropout is extremely sensitive to dropout rate, with even $p = 0.1$ performing poorly (see Figure B.4). We therefore use $p = 0.05$ for MC-dropout experiments on ImageNet. This high sensitivity to dropout rate is an issue with MC-dropout as a method.

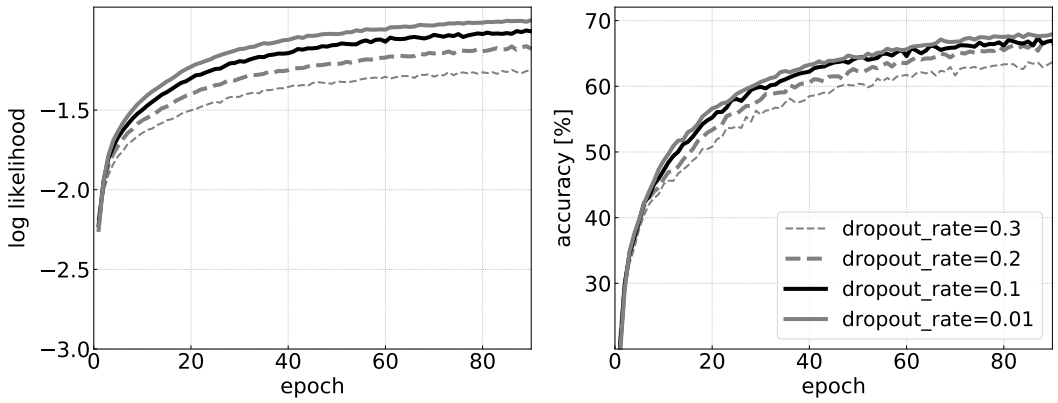


Figure B.3: Effect of changing the dropout rate in MC-dropout, training LeNet-5 on CIFAR-10. When $p = 0.01$, the train-test gap on accuracy and log-likelihood is very high (10.3% and 0.34 respectively). When $p = 0.1$, gaps are 1.4% and 0.04 respectively. When $p = 0.2$, the gaps are -7.71% and -0.02 respectively. We therefore choose $p = 0.1$ as it has high accuracy and log-likelihood, and small train-test gap.

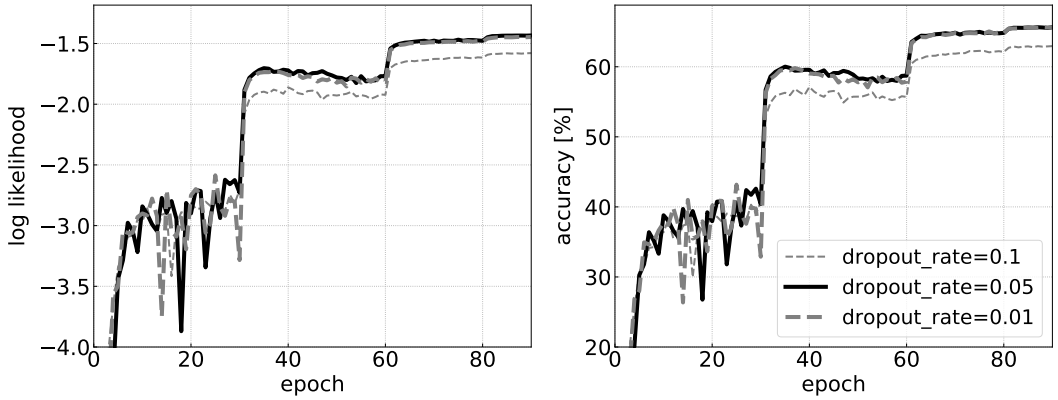


Figure B.4: Effect of changing the dropout rate in MC-dropout, training Resnet-18 on ImageNet. We use $p = 0.05$ for our results.

B.5 Details on uncertainty metrics

We use several approaches to compare uncertainty estimates obtained by each optimiser. We follow the same methodology for all optimisers: first, tune hyperparameters to obtain good accuracy on the validation set. Then, test on uncertainty metrics. For multi-class classification problems, all of these are based on the predictive probabilities. For non-Bayesian approaches, we compute the probability of class k for a validation input $\boldsymbol{x}_i$ as $\hat{p}_{ik} = p(y_i = k|\boldsymbol{x}_i, \boldsymbol{w}_*)$, where $\boldsymbol{w}_*$ is the weight vector of the DNN whose uncertainty we are estimating. For Bayesian

methods, we can compute the predictive probabilities for each validation example $\boldsymbol{x}_i$ as,

$$\hat{p}_{ik} = \int p(y_i = k|\boldsymbol{x}_i, \boldsymbol{w}) p(\boldsymbol{w}|\mathcal{D}) d\boldsymbol{w} \approx \int p(y_i = k|\boldsymbol{x}_i, \boldsymbol{w}) q(\boldsymbol{w}) d\boldsymbol{w}$$
$$\approx \frac{1}{S} \sum_{s=1}^{S} p(y_i = k|\boldsymbol{x}_i, \boldsymbol{w}^{(s)}), \tag{B.14}$$

where $\boldsymbol{w}^{(s)} \sim q(\boldsymbol{w})$ are samples from the (usually Gaussian) approximation returned by a variational method. We use 10 MC samples at validation-time for VOGN and MC-dropout (the effect of changing number of validation MC samples is shown in Figure 3.10). This increases the computational cost during testing for these methods when compared to Adam or SGD.

Using the estimates $\hat{p}_{ik}$, we use three methods to compare uncertainties: validation log-likelihood, AUROC and calibration curves. We also compare uncertainty performance by looking at model outputs when exposed to out-of-distribution data.

Validation log-likelihood. Log-likelihood (or log-loss) is a common uncertainty metric. We consider a validation set of N_{Va} examples. For an input $\boldsymbol{x}_i$, denote the true label by $\boldsymbol{y}_i$, a 1-of-K encoded vector with 1 at the true label and 0 elsewhere. Denote the full vector of all validation outputs by $\boldsymbol{y}$. Similarly, denote the vector of all probabilities $\hat{p}_{ik}$ by $\boldsymbol{p}$, where $k \in \{1, ..., K\}$. The validation log-likelihood is defined as $\ell(\boldsymbol{y}, \hat{\boldsymbol{p}}) = \frac{1}{N_{Va}} \sum_{i=1}^{N_{Va}} \sum_{k=1}^{K} y_{ik} \log \hat{p}_{ik}$.

Area Under ROC curves (AUROC). We consider Receiver Operating Characteristic (ROC) curves for our multi-way classification tasks. A potential way that we may care about uncertainty measurements would be to discard uncertain examples by thresholding each validation input's predicted class' softmax output, marking them as too ambiguous to belong to a class. We can then consider the remaining validation inputs to either be correctly or incorrectly classified, and calculate the True Positive Rate (TPR) and False Positive Rate (FPR) accordingly. The ROC curve is summarised by its Area Under Curve (AUROC), reported in Table 3.3 and Table B.1. This metric is useful to compare uncertainty performance in conjunction with the other metrics we use. The AUROC results are very similar between optimisers, particularly on ImageNet, although MC-dropout performs marginally better than the others, including VOGN. On all but one CIFAR-10 experiment (AlexNet, without DA), VOGN performs the best, or tied best. Adam performs the worst, but is surprisingly good on CIFAR-10/ResNet-18.

Calibration Curves and Expected Calibration Error. Calibration curves (DeGroot and Fienberg, 1983) test how well-calibrated a model is by plotting true accuracy as a function of the model's predicted accuracy $\hat{p}_{ik}$ (we only consider the predicted class' $\hat{p}_{ik}$). Perfectly calibrated models would follow the $y = x$ diagonal line on a calibration curve.

We approximate this curve by binning the model's predictions into $M = 20$ bins, as is often done. We show calibration curves in Figures 3.8 and 3.11. We can also consider the **Expected Calibration Error (ECE)** metric (Naeini et al., 2015; Guo et al., 2017), reported in Table 3.3 and Table B.1. ECE calculates the expected error between the true accuracy and the model's predicted accuracy, averaged over all validation examples, again approximated by using M bins.

B.6 Further out-of-distribution experiments with VOGN

In this section, we provide further out-of-distribution (OOD) experiments with VOGN. We explained the setup and metrics in Section 3.2.2, particularly around Figure 3.12, where we showed OOD results for ResNet-18 trained on CIFAR-10. We now provide similar figures for AlexNet in Figures B.5 and B.6 (trained on CIFAR-10 with DA and without DA respectively) and on LeNet-5 in Figure B.7. These results are discussed in Section 3.2.2.

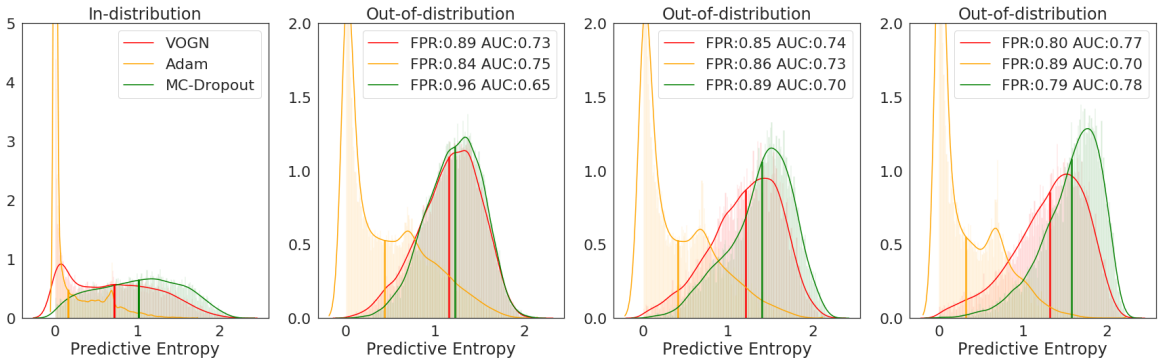


Figure B.5: Histograms of predictive entropy for out-of-distribution tests for AlexNet trained on CIFAR-10 with data augmentation. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the AUROC metric (higher is better) and FPR at 95% TPR metric (lower is better), averaged over 3 runs. The standard deviations are very small and so not reported here.

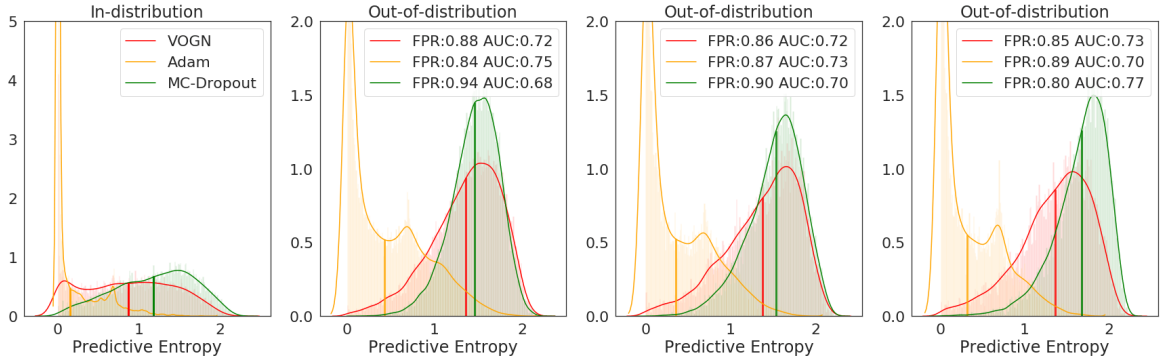


Figure B.6: Histograms of predictive entropy for out-of-distribution tests for AlexNet trained on CIFAR-10 without data augmentation. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the AUROC metric (higher is better) and FPR at 95% TPR metric (lower is better), averaged over 3 runs. The standard deviations are very small and so not reported here.

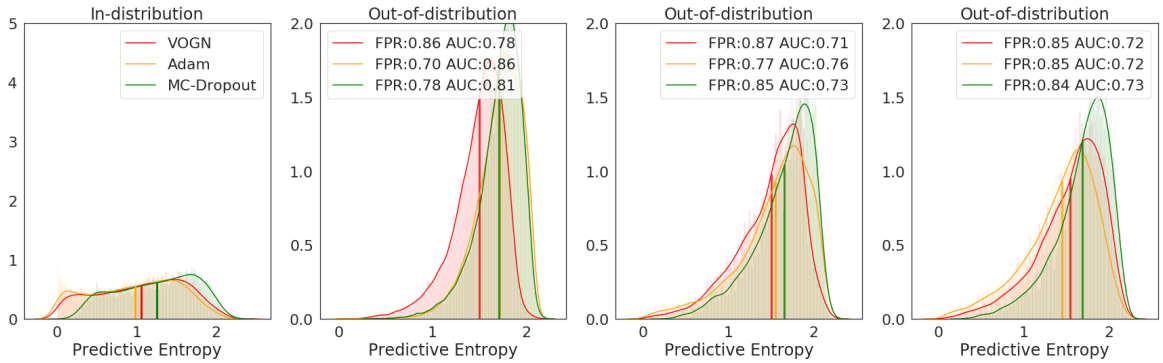


Figure B.7: Histograms of predictive entropy for out-of-distribution tests for LeNet-5 trained on CIFAR-10 without data augmentation. Going from left to right, the inputs are: the in-distribution dataset (CIFAR-10), followed by out-of-distribution data: SVHN, LSUN (crop), LSUN (resize). Also shown are the AUROC metric (higher is better) and FPR at 95% TPR metric (lower is better), averaged over 3 runs. The standard deviations are very small and so not reported here.

Appendix C

Details on FROMP

In this Appendix, we provide further details on our derivations and experiments in Chapter 4, for the Functional Regularisation of Memorable Past (FROMP) algorithm. Appendix C.1 shows how the posterior distribution of a linear model induces a GP posterior, from Rasmussen and Williams (2006). Appendix C.2 discusses how we apply FROMP to the multi-class setting, discussing the changes in equations and additional approximations we make (Chapter 4 only discusses the binary classification case). Appendix C.3 gives hyperparameters for all experiments with FROMP and OGN-FROMP in Section 4.5. Appendix C.4 shows visualisations for the variations on the Toy-Gaussians benchmark, providing more information on the results in Table 4.2. Finally, Appendix C.5 illustrates the importance of the kernel being over all weights instead of just the final layer weights.

C.1 Gaussian Process posteriors from the minimiser of a linear model

The posterior distribution of a linear model induces a GP posterior as shown by Rasmussen and Williams (2006). We discuss this in detail now for the following linear model discussed in Section 4.2,

$$y_i = f_w(\boldsymbol{x}_i) + \epsilon_i, \quad \text{where } f_w(\boldsymbol{x}_i) = \boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{w}, \ \epsilon_i \sim \mathcal{N}(\epsilon_i; 0, \Lambda^{-1}), \\ \text{and } \boldsymbol{w} \sim \mathcal{N}(\boldsymbol{w}; 0, \delta^{-1}\mathbf{I}_P), \tag{C.1}$$

with a feature map $\boldsymbol{\phi}(\boldsymbol{x})$. Rasmussen and Williams (2006) show that the predictive distribution for a test input $\boldsymbol{x}$ takes the following form (see Equation 2.11 in their book),

$$p(f(\boldsymbol{x})|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(f(\boldsymbol{x})\,; \Lambda \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{A}^{-1} \boldsymbol{\Phi} \boldsymbol{y},\ \ \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{A}^{-1} \boldsymbol{\phi}(\boldsymbol{x})), \\ \text{where } \boldsymbol{A} = \sum_i \boldsymbol{\phi}(\boldsymbol{x}_i)\, \Lambda\, \boldsymbol{\phi}(\boldsymbol{x}_i)^\top + \delta \mathbf{I}_P, \tag{C.2}$$

where $\mathcal{D}$ is the set of N training points $\{\boldsymbol{x}_i, y_i\}$, and $\boldsymbol{\Phi}$ is a matrix with $\boldsymbol{\phi}(\boldsymbol{x}_i)$ as columns.

Rasmussen and Williams (2006) derive the above predictive distribution by using the weight-space posterior $\mathcal{N}(\boldsymbol{w}; \boldsymbol{w}_{\text{lin}}, \boldsymbol{\Sigma}_{\text{lin}})$ with mean and covariance defined as,

$$\boldsymbol{w}_{\text{lin}} = \Lambda \boldsymbol{A}^{-1} \boldsymbol{\Phi} \boldsymbol{y}, \quad \boldsymbol{\Sigma}_{\text{lin}} = \boldsymbol{A}^{-1}. \tag{C.3}$$

The mean $\boldsymbol{w}_{lin}$ is also the minimiser of the least-squares loss and $\boldsymbol{A}$ is the hessian at that solution.

Rasmussen and Williams (2006) show that the predictive distribution in Equation C.2 corresponds to a GP posterior with the following mean and covariance functions,

$$m_{\text{lin}}(\boldsymbol{x}) = \Lambda \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{A}^{-1} \boldsymbol{\Phi} \boldsymbol{y} = \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{w}_{\text{lin}} = f_{w_{\text{lin}}}(\boldsymbol{x}), \tag{C.4}$$

$$\kappa_{\text{lin}}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^\top\, \boldsymbol{\Sigma}_{\text{lin}}\, \boldsymbol{\phi}(\boldsymbol{x}'). \tag{C.5}$$

This is the result shown in Equation 4.7 in Section 4.2. We can also write the predictive distribution of the observation $y = f(\boldsymbol{x}) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \Lambda^{-1})$ as follows,

$$p(y|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(y\,; \underbrace{f_{w_{\text{lin}}}(\boldsymbol{x})}_{m_{\text{lin}}(\mathbf{x})},\ \ \underbrace{\boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\Sigma}_{\text{lin}} \boldsymbol{\phi}(\boldsymbol{x})}_{\kappa_{\text{lin}}(\mathbf{x},\mathbf{x})} + \Lambda^{-1}), \\ \text{where } \boldsymbol{\Sigma}_{\text{lin}}^{-1} = \sum_i \boldsymbol{\phi}(\boldsymbol{x}_i)\, \Lambda\, \boldsymbol{\phi}(\boldsymbol{x}_i)^\top + \delta \mathbf{I}_P. \tag{C.6}$$

This is the same as Equation 4.8, and in Section 4.2 we make use of Equations C.4 to C.6 to write the mean and covariance function of the posterior approximation for neural networks.

C.2 Multiclass setting for FROMP

All our derivations and equations in Chapter 4 are for scalar outputs. In this section, we discuss multiclass FROMP, where there are more than two classes. We start with the DNN2GP (Khan et al., 2019) result for multiclass classification losses, which is an extension of the binary classification case in Section 4.2. Using this, we can convert from a distribution over weights to a distribution over functions. We then discuss how we reduce complexity in the multiclass setting.

From deep networks to functional priors: a multiclass classification loss

The result in Section 4.2 for a binary classification loss straightforwardly extends to the multiclass classification case by using a multinomial-logit likelihood (or softmax function). The loss is now,

$$\ell(\boldsymbol{y}, \boldsymbol{f}) = -\boldsymbol{y}^\top \mathcal{S}(\boldsymbol{f}) + \log\left(1 + \sum_{k=1}^{K-1} e^{f_k}\right), \text{ where } k\text{'th element of } \mathcal{S}(\boldsymbol{f}) \text{ is } \frac{e^{f_k}}{1 + \sum_{c=1}^{K-1} e^{f_c}}, \tag{C.7}$$

where the number of categories is equal to K, $\boldsymbol{y}$ is a one-hot-encoding vector of size $K-1$, $\boldsymbol{f}$ is the $K-1$ length output of the neural network, and $\mathcal{S}(\boldsymbol{f})$ is the softmax operation which maps a $K-1$ length real vector to a $K-1$ dimensional vector with entries in the open interval $(0, 1)$. The encoding in $K-1$ length vectors ignores the last category, ensuring identifiability (Train, 2009). In a similar fashion to the binary case, the predictive distribution of the $K-1$ length output $\boldsymbol{y}$ for an input $\boldsymbol{x}$ can be written as,

$$\hat{p}(\boldsymbol{y}|\boldsymbol{x}, \mathcal{D}) = \mathcal{N}(\boldsymbol{y}\,; \mathcal{S}(\boldsymbol{f}_{\boldsymbol{w}_*}(\boldsymbol{x})),\ \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{\Sigma}_*\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})^\top \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})^\top + \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})),$$
$$\text{where } \boldsymbol{\Sigma}_*^{-1} = \sum_i \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)^\top\, \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}_i) + \delta \mathbf{I}_P, \tag{C.8}$$

where $\boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x}) = \mathcal{S}\left(\boldsymbol{f}_{\boldsymbol{w}_*}(\boldsymbol{x})\right)\left[1 - \mathcal{S}\left(\boldsymbol{f}_{\boldsymbol{w}_*}(\boldsymbol{x})\right)\right]^\top$ is a $(K-1) \times (K-1)$ matrix and $\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})$ is the $(K-1) \times P$ Jacobian matrix. The mean function in this case is a $K-1$ length matrix and the covariance function is a square matrix of size $K-1$. Their expressions are,

$$\boldsymbol{m}_{\boldsymbol{w}_*}(\boldsymbol{x}) = \mathcal{S}(\boldsymbol{f}_{\boldsymbol{w}_*}(\boldsymbol{x})), \quad \boldsymbol{K}_{\boldsymbol{w}_*}(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\, \boldsymbol{\Sigma}_*\, \boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x}')^\top \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x}'). \tag{C.9}$$

These expressions are very similar to Equation 4.13, except now with multiple outputs.

Reducing complexity in the multiclass setting

We could use the full multiclass version in Equation C.8, but this is expensive. To keep computational complexity low, we employ an individual Gaussian Process (GP) over each of the K classes seen in a previous task, and treat the GPs as independent.

We have K separate GPs. Let $y^{(k)}$ be the k-th item of $\boldsymbol{y}$. Then the predictive distribution over each $y^{(k)}$ for an input $\boldsymbol{x}$ is,

$$\hat{p}(y^{(k)}|\boldsymbol{x},\mathcal{D}) = \mathcal{N}\big(y^{(k)}\,;\,\mathcal{S}(\boldsymbol{f}_{\boldsymbol{w}_*}(\boldsymbol{x}))^{(k)},\;\; \boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})^{(k)}\,\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})\,\boldsymbol{\Sigma}_*\,\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})^\top\boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})^{(k)\top} + \Lambda_{\boldsymbol{w}_*}(\boldsymbol{x})^{(k,k)}\big), \quad \text{(C.10)}$$

where $\mathcal{S}(\boldsymbol{f}_{\boldsymbol{w}_*}(\boldsymbol{x}))^{(k)}$ is the k-th output of the softmax function, $\boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x})^{(k)}$ is the k-th row of the Hessian matrix and $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x})^{(k,k)}$ is the k,k-th element of the Hessian matrix. The Jacobians $\boldsymbol{J}_{\boldsymbol{w}_*}(\boldsymbol{x})$ are now of size $K \times P$. Note that we have allowed $\mathcal{S}$ and $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x})$ to be of size K instead of $K-1$. This is because we are treating the K GPs independently.

The kernel matrix $\boldsymbol{K}_{t-1}$ is now a block-diagonal matrix for each previous task's classes. This allows us to only compute inverses of each block diagonal (size $M \times M$), repeated for each class in each past task ($K(t-1)$ times), where M is the number of memorable past examples in each task. This changes computational complexity to be linear in the number of classes per task, K, compared to Section 4.4.2 (which only has analysis for binary classification in each task).

When choosing a memorable past (the subset of points to regularise function values over) for the logistic regression case, we can simply sort the $\Lambda_{\boldsymbol{w}_*}(\boldsymbol{x}_i)$'s for all $\{\boldsymbol{x}_i\} \in \mathcal{D}_t$ and pick the largest, as explained in Section 4.3. In the multiclass case, these are now $K \times K$ matrices $\boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x}_i)$. We instead sort by $\text{Tr}(\boldsymbol{\Lambda}_{\boldsymbol{w}_*}(\boldsymbol{x}_i))$ to select the memorable past examples.

FROMP for multiclass classification: The solutions found by the multiclass algorithm is the fixed point of this objective (compare with Equation 4.30),

$$\min_{w} N\bar{\ell}_t(\boldsymbol{w}) + \tfrac{1}{2}\tau \sum_{s=1}^{t-1}\sum_{k\in K_s}(\boldsymbol{m}_{t,s,k} - \boldsymbol{m}_{t-1,s,k})^\top \boldsymbol{K}_{t-1,s,k}^{-1}(\boldsymbol{m}_{t,s,k} - \boldsymbol{m}_{t-1,s,k}), \quad \text{(C.11)}$$

where we define K_s as the set of classes k seen in previous task s, $\boldsymbol{m}_{t,s,k}$ is the vector of $m_{\boldsymbol{w}_t}(\boldsymbol{x})$ for class k, $\boldsymbol{m}_{t-1,s,k}$ is the vector of $m_{\boldsymbol{w}_{t-1}}(\boldsymbol{x})$ for class k, and $\boldsymbol{K}_{t-1,s,k}$ is the kernel matrix from the previous task just for class k, always evaluated over just the memorable points from previous task s. By decomposing the last term over individual outputs and over the memorable past from each task, we have reduced the computational complexity per update.

C.3 Hyperparameters for FROMP experiments

In this section, we provide hyperparameter values for all experiments in Section 4.5.

FROMP on Toy-Gaussians. For our result on the standard Toy-Gaussians benchmark (Table 4.1), we use number of epochs $= 50$, batch size $= 20$, and learning rate $= 0.01$, similarly to the Joint Tasks hyperparameter values. We also use $\tau = 1$.

FROMP on Permuted MNIST. We use the Adam optimiser (Kingma and Ba, 2015) with Adam learning rate set to 0.001 and parameter $\beta_1 = 0.99$, and also employ gradient clipping. The minibatch size is 128, and we learn each task for 10 epochs. We use $\tau = 0.5N$ when there are 200 memorable points. We use a fully connected single-head network with two hidden layers, each consisting of 100 hidden units with ReLU activation functions. We report performance after 10 tasks.

FROMP on Split MNIST. We use the Adam optimiser with Adam learning rate set to 0.0001 and parameter $\beta_1 = 0.99$, and also employ gradient clipping. The minibatch size is 128, and we learn each task for 15 epochs. We use $\tau = 10N$ when there are 40 memorable points. We use a fully connected multi-head network with two hidden layers, each with 256 hidden units and ReLU activation functions. Our FROMP-Leverage experiments in Section 4.5.1 use the same hyperparameter values.

Sensitivity to the value of τ. We tested FROMP and FROMP-L_2 with different values of the hyperparameter τ on Split MNIST. We found that τ can change by an order of magnitude without significantly affecting final average accuracy. Larger changes in τ led to greater than 0.1% loss in accuracy.

FROMP on Split CIFAR. We use the Adam optimiser with Adam learning rate set to 0.001 and parameter $\beta_1 = 0.99$, and also employ gradient clipping. The minibatch size is 256, and we learn each task for 80 epochs. We use $\tau = 10N$ when there are 200 memorable points. Our FROMP-Leverage experiments in Section 4.5.1 use the same hyperparameter values, as does our experiment with 11 tasks instead of the standard 6 tasks.

OGN-FROMP on Split MNIST. We use the same hyperparameters as in the FROMP experiment (values given above), with some slight differences due to a different optimiser (OGN not Adam): $\beta_1 = 0.001$, $\alpha = 0.001$, $\beta_2 = 0.9$, $\delta = 0.001$. See Table 3.2 for definitions of these hyperparameters.

Fewer memorable past examples

When we have fewer memorable past examples, we increase τ to compensate for the fewer datapoints. For example, for Split CIFAR, when we have 40 memorable past examples per task (instead of 200), we use $\tau = (200/40) \times 10N = 50N$ (instead of $\tau = 10N$ for 200 memorable past points). We do this for all experiments with fewer memorable past examples, and do not tune τ in any other way when the number of examples is decreased.

C.4 Variations on Toy-Gaussians benchmark

In Table 4.2 we showed results for FROMP on variations of the Toy-Gaussians benchmark, showing that FROMP is robust to changes in this benchmark. Figures C.1 to C.5 visualise the different dataset variations. We pick the middle performing FROMP run (out of 5) and Joint Tasks run to show. The hyperparameters are in Appendix C.3, except we scale the number of epochs appropriately by 10 when the dataset size is scaled by 10.

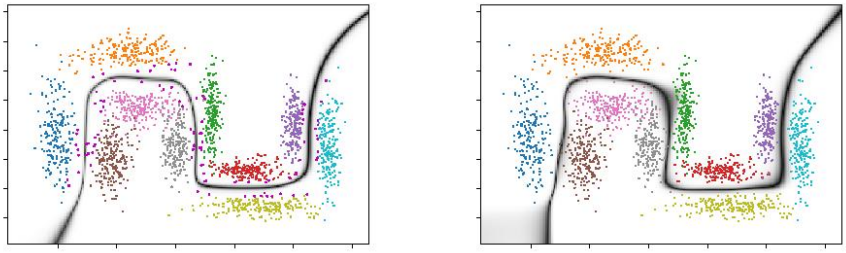


Figure C.1: FROMP (middle performing of 5 runs), left, and Joint Tasks, right, on a dataset 10x smaller (400 points per task).

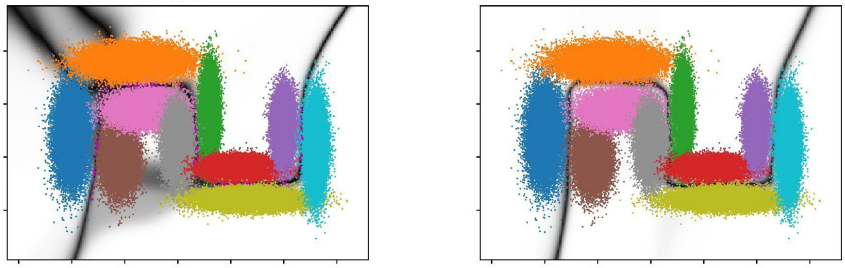


Figure C.2: FROMP (middle performing of 5 runs), left, and Joint Tasks, right, on a dataset 10x larger (40,000 points per task).

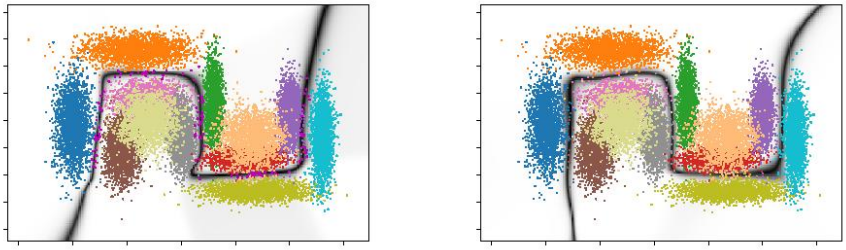


Figure C.3: FROMP (middle performing of 5 runs), left, and Joint Tasks, right, on a dataset with a new, easy, 6th task.

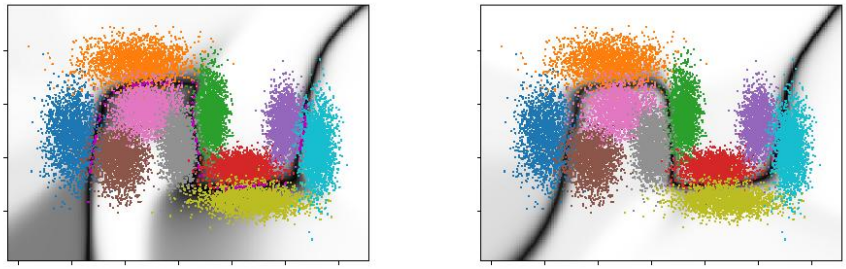


Figure C.4: FROMP (middle performing of 5 runs), left, and Joint Tasks, right, on a dataset with increased standard deviations of each class' points, making classification tougher.

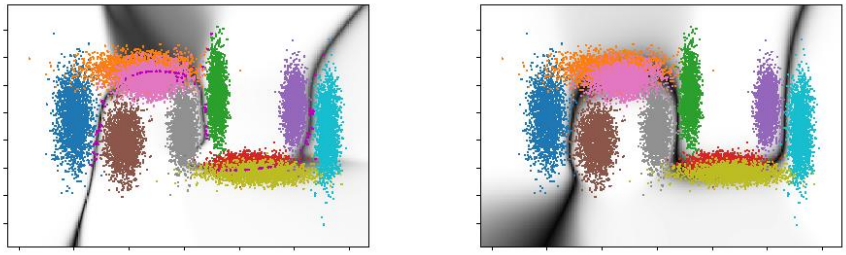


Figure C.5: FROMP (middle performing of 5 runs), left, and Joint Tasks, right, on a dataset with 2 tasks having overlapping data.

C.5 Importance of kernel being over all weights

In this section, we show the importance of using a kernel over all weights in the neural network, instead of just the last layer. We run on the Toy-Gaussians benchmark, and consider the entropies of the Gaussian distributions for weights in each layer. We plot the histogram of these entropies in Figure C.6. As can be seen, all layers have weights with high uncertainty (high entropy), especially for the first few tasks. Note that as we train for more tasks, we expect the uncertainties to reduce as our network parameters become more certain having seen more data.

Therefore, by considering uncertainties across weights in all layers, instead of just the last layer, we might expect better performance.

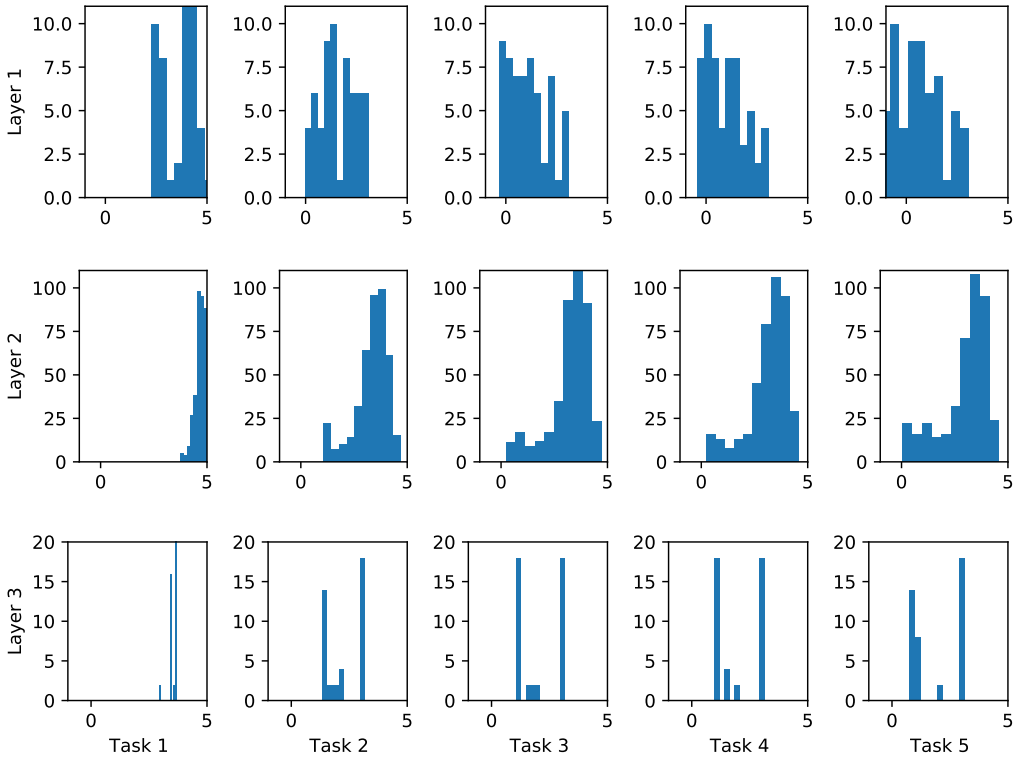


Figure C.6: Histogram of entropy of distribution the distribution of weights for different layers (row) and task (columns). For each layer, we take all weights and plot the histogram of their entropies. Left-most is after the first task, and right-most is after the last task. We see that the entopy is high across layers, implying that there is significant uncertainties about the weights in all layers, not only the last layer (layer 3 in this case).

Appendix D

Details on K-priors

In this Appendix we provide further theoretical results and experimental results using Knowledge-adaptation priors (K-priors) from Chapter 5. In Appendix D.1, we design K-priors that best use a fixed memory, looking at how to best transfer first-order and second-order gradient information from the base model. Appendix D.2 considers FROMP on linear regression, showing that FROMP is not always exact even in this simple setting (while K-priors are, see Section 5.1). Appendix D.3 shows equivalence between K-priors and Gaussian Processes. Finally, in Appendix D.4 we provide hyperparameters for all our K-priors experiments in Section 5.7, and provide more experiments.

D.1 K-priors that optimally preserve information with limited memory

In this section, we look at different ways of designing K-priors to best use a set of stored inputs. We assume we are given a base model with parameters $\boldsymbol{w}_*$, and have chosen our memory using some method, $\mathcal{M} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, ..., \boldsymbol{u}_M\}$. We assume this memory is of small size and limited in some way. Section 5.4 provides an initial discussion on how we might choose points to store in memory. We focus on the MAP/Laplace case in this section.

We specifically use the general K-prior form in Section 5.3.4, which we now repeat. To replace an old objective such as Equation 5.1, with loss $\ell_i^{\text{old}}(f)$ and regulariser $\mathcal{R}^{\text{old}}(\boldsymbol{w})$, with a new objective with loss $\ell_i^{\text{new}}(f)$ and regulariser $\mathcal{R}^{\text{new}}(\boldsymbol{w})$, the divergences should be chosen such that they have the following gradients,

$$\nabla_{\boldsymbol{w}} \mathbb{D}_w(\boldsymbol{w} \| \boldsymbol{w}_*) = \nabla \mathcal{R}^{\text{new}}(\boldsymbol{w}) - \nabla \mathcal{R}^{\text{old}}(\boldsymbol{w}), \tag{D.1}$$

$$\nabla_{\boldsymbol{w}} \mathbb{D}_f(\boldsymbol{f}(\boldsymbol{w}) \| \boldsymbol{f}(\boldsymbol{w}_*)) = [\nabla_{\boldsymbol{w}} \boldsymbol{f}(\boldsymbol{w})]^\top \mathbf{B}\, \boldsymbol{d}_u, \tag{D.2}$$

where $\boldsymbol{d}_u$ is an M-length vector with the discrepancy $\nabla_f \ell_m^{\text{new}}(f_{\boldsymbol{w}}) - \nabla_f \ell_m^{\text{old}}(f_{\boldsymbol{w}_*})$ as the m'th entry, for $m \in \mathcal{M}$. The matrix $\mathbf{B}$ is added to counter the mismatch between $\mathcal{D}_{\text{old}}$ and $\mathcal{M}$, and in this section we look at deriving expressions for $\mathbf{B}$.

As we discussed in Chapter 5, the function-divergence term in K-priors has a specific role: we want it to match the likelihood term in our overall objective. In this section we consider the MAP objective over the base model (Equation 5.1) for Generalised Linear Models or neural networks,

$$\sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f_{\boldsymbol{w}}^i)). \tag{D.3}$$

In Appendix D.1.1 we minimise the L_2-distance between the gradient of past information (the gradient of Equation D.3 vs Equation D.2).

In Appendix D.1.2 we focus on designing $\mathbf{B}$ to match second-order information of Equation D.3. We again minimise the L_2-distance. We will see that the $\mathbf{B}^{*,\text{2ord}}$ we derive is of a similar form to $\mathbf{B}^{\text{FROMP}}$ in Section 5.6. This will also provide some ways to improve the FROMP algorithm (FROMP was introduced and discussed in Chapter 4). We will also see expressions that are related to the Nyström approximation.

D.1.1 Preserving first-order information

In this section, we want to match first-order information from Equation D.3 using the function-divergence term in K-priors. Specifically, we want the gradient of Equation D.3 to be close to Equation D.2 in some way, where we are given a memory set $\mathcal{M}$ in our K-prior.

For simplicity, we assume the same loss in the base model (this could be a GLM or neural network, for example), $\ell_i^{\text{new}}(f) = \ell_i^{\text{old}}(f) = \ell(y_i, h(f))$. The gradient of Equation D.3 is then given by,

$$\nabla_{\boldsymbol{w}} \sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f_{\boldsymbol{w}}^i)) = \boldsymbol{J}_x^\top \boldsymbol{d}_x, \tag{D.4}$$

where $\boldsymbol{J}_x$ is an $N_{\text{old}} \times P$ matrix with rows given by $\nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_i)$, and $\boldsymbol{d}_x$ is an N_{old}-length vector with the discrepancy $\nabla_f \ell(y_i, h(f_{\boldsymbol{w}}^i)) - \nabla_f \ell(y_i, h(f_{\boldsymbol{w}_*}^i)) = h(f_{\boldsymbol{w}}^i) - h(f_{\boldsymbol{w}_*}^i)$ as the i'th entry, for all $i \in \mathcal{D}_{\text{old}}$, and $|\mathcal{D}_{\text{old}}| = N_{\text{old}}$. With GLMs, the Jacobian $\boldsymbol{J}_x$ is constant (and given by the input features), while for neural networks, the Jacobian depends on the value of the parameter $\boldsymbol{w}$.

We choose to minimise the L_2-distance between the two gradients (although we may be able to derive other interesting expressions by minimising different distances),

$$\mathbf{B}^{*,1\text{ord}} = \arg\min_{\mathbf{B}} \ \tfrac{1}{2}\|\boldsymbol{J}_x^\top \boldsymbol{d}_x - \boldsymbol{J}_u^\top \mathbf{B}\boldsymbol{d}_u\|^2, \tag{D.5}$$

where $\boldsymbol{J}_u$ is an $M \times P$ matrix with rows given by $\nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{u}_m)$, $\boldsymbol{d}_u$ is an M-length vector with the discrepancy $h(f_{\boldsymbol{w}}(\boldsymbol{u}_m)) - h(f_{\boldsymbol{w}_*}(\boldsymbol{u}_m))$ as the m'th entry, for all $m \in \mathcal{M}$, and $|\mathcal{M}| = M$.

Taking the gradient of the objective in Equation D.5 and setting equal to zero, we get,

$$\mathbf{B}^{*,1\text{ord}} \boldsymbol{d}_u \boldsymbol{d}_u^\top = \left(\boldsymbol{J}_u \boldsymbol{J}_u^\top\right)^{-1} \left(\boldsymbol{J}_u \boldsymbol{J}_x^\top\right) \boldsymbol{d}_x \boldsymbol{d}_u^\top. \tag{D.6}$$

This first-order optimal $\mathbf{B}^{*,1\text{ord}}$ is the best we can do to reconstruct the gradient of past information given a fixed memory that we did not choose (for an L_2 objective).

We also note two interesting relationships to other K-priors in Chapter 5:

1. When $\mathcal{M} = \mathcal{X}_{\text{old}}$ and on GLMs, we get $\mathbf{B}^{*,1\text{ord}} = \mathbf{I}$ and recover the vanilla K-prior from Section 5.1, which reconstructs the exact gradient with GLMs (the loss in Equation D.5 is zero). We can show this by taking the SVD of $\boldsymbol{J}_u = \boldsymbol{J}_x$ to lead to cancellations in Equation D.6.

2. The form of $\mathbf{B}^{*,1\text{ord}}$ is very similar to the optimal K-prior discussed in Section 5.4. Specifically, we can recover Equation 5.50 by taking the SVD of $\boldsymbol{J}_u$ and $\boldsymbol{J}_x$ and plugging in to Equation D.6. The difference here is that we assume we are given a fixed memory that we did not choose, while the optimal K-prior required the SVD of $\boldsymbol{J}_x$ to choose inputs to store in memory.

It would be very interesting to further explore the properties of $\mathbf{B}^{*,1\text{ord}}$, and test it experimentally. We note that this is difficult to do in practice as $\boldsymbol{d}_x$ in Equation D.6 requires storing all past memory, but we may be able to approximate it in certain settings.

D.1.2 Preserving second-order information

In this section, we match second-order information from Equation D.3 with the function-divergence term in K-priors. We follow the same process as in Appendix D.1.1, except with second-order information. We assume we are given a memory $\mathcal{M}$, and want to use the matrix $\mathbf{B}$ in Equation D.2. We will see that the optimal $\mathbf{B}^{*,2\text{ord}}$ is related to FROMP (see Section 5.6).

We use the same definitions as in Appendix D.1.1. The second-order derivative of Equation D.3 is then given by,

$$\nabla^2_{\boldsymbol{ww}} \sum_{i \in \mathcal{D}_{\text{old}}} \ell(y_i, h(f^i_{\boldsymbol{w}})) = \boldsymbol{J}_x^\top \boldsymbol{\Lambda}_x \boldsymbol{J}_x, \tag{D.7}$$

where $\boldsymbol{\Lambda}_x$ is an $N_{\text{old}} \times N_{\text{old}}$ diagonal matrix with $h'(f^i_{\boldsymbol{w}})$ as its i'th diagonal entry, over all $i \in \mathcal{D}_{\text{old}}$. The second-order derivative of the function-divergence term in K-priors is,

$$\nabla^2_{\boldsymbol{ww}} \mathbb{D}_f(\boldsymbol{f}(\boldsymbol{w}) \| \boldsymbol{f}(\boldsymbol{w}_*)) = \boldsymbol{J}_u^\top \mathbf{B} \boldsymbol{\Lambda}_u \boldsymbol{J}_u, \tag{D.8}$$

where $\boldsymbol{\Lambda}_u$ is an $M \times M$ diagonal matrix with $h'(f^m_{\boldsymbol{w}})$ as its m'th diagonal entry, over all $m \in \mathcal{M}$.

Again, we minimise the L_2-distance between these,

$$\mathbf{B}^{*,\text{2ord}} = \arg\min_{\mathbf{B}} \ \tfrac{1}{2} \| \boldsymbol{J}_x^\top \boldsymbol{\Lambda}_x \boldsymbol{J}_x - \boldsymbol{J}_u^\top \mathbf{B} \boldsymbol{\Lambda}_u \boldsymbol{J}_u \|^2. \tag{D.9}$$

We now explicitly assume that a pseudo-inverse of $\boldsymbol{J}_u$ exists, which we denote as $\boldsymbol{J}_u^+$. We specifically use the right-pseudo inverse $\boldsymbol{J}_u^+ = \boldsymbol{J}_u^\top \left(\boldsymbol{J}_u \boldsymbol{J}_u^\top \right)^{-1}$. We could compute this, for example, via an SVD of $\boldsymbol{J}_u$ (although this can get very expensive for large M or large P). It may be possible to approximate the pseudo-inverse through approximate decompositions of $\boldsymbol{J}_u$.

Taking the gradient of the objective in Equation D.9 and setting to zero, we get,

$$\mathbf{B}^{*,\text{2ord}} = \boldsymbol{J}_u^{+\top} \left(\boldsymbol{J}_x^\top \boldsymbol{\Lambda}_x \boldsymbol{J}_x \right) \boldsymbol{J}_u^+ \boldsymbol{\Lambda}_u^{-1}. \tag{D.10}$$

Like in Appendix D.1.1, when the memory $\mathcal{M} = \mathcal{X}_{\text{old}}$ is all the past inputs, then $\mathbf{B}^{*,\text{2ord}}$ becomes the identity matrix, and we recover the vanilla K-prior from Section 5.1.

We find that FROMP (from Chapter 4) effectively approximates this second-order optimal $\mathbf{B}^{*,\text{2ord}}$. In Section 5.6 we viewed FROMP as being in the K-priors framework, and derived FROMP's $\mathbf{B}^{\text{FROMP}}$ (see Equation 5.66). In fact, the only difference between $\mathbf{B}^{\text{FROMP}}$ and $\mathbf{B}^{*,\text{2ord}}$ lies in FROMP calculating its kernel using the base model $\boldsymbol{w}_*$: this difference leads to $\boldsymbol{J}_{u,*}$ instead of $\boldsymbol{J}_u$, as well as an additional $\boldsymbol{\Lambda}_u \boldsymbol{\Lambda}_{u,*}^{-1}$ term.

This leads to a very easy way to improve FROMP: we could recalculate the kernel at the current model parameters. This could be every few iterations to save computation cost (calculating and inverting the kernel in FROMP can be expensive).

D.2 FROMP is not exact on linear regression

In this section, we consider var-FROMP on linear regression on the 'Add Data' task (this is continual learning with just two tasks), and we store all past datapoints in memory. In this simple setting, weight-priors are known to recover the Retrained solution. We also know that vanilla K-priors are exact from Section 5.1.2. However, we will see that var-FROMP (and by extension, FROMP) is not always exact even in this simple setting.

The notation we use is the same as in Section 5.1. We assume a scalar function output $f^i_{\boldsymbol{w}} = \boldsymbol{\phi}_i^\top \boldsymbol{w}$ using a feature map $\boldsymbol{\phi}_i$. For linear regression, we have a scalar output $y_i = f^i_{\boldsymbol{w}} + \epsilon_i$ with Gaussian noise $\mathcal{N}(\epsilon_i; 0, \Lambda^{-1})$. We have a Gaussian prior $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \mathbf{0}, \delta^{-1}\mathbf{I})$. After seeing $\mathcal{D}_{\text{old}}$, the posterior distribution we have is $q_{\boldsymbol{\eta}_*}(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$, where $\boldsymbol{\mu}_*$ is the MAP solution and $\boldsymbol{\Sigma}_* = \sum_{i \in \mathcal{D}_{\text{old}}} \boldsymbol{\phi}_i \, \Lambda \, \boldsymbol{\phi}_i^\top + \delta \mathbf{I}$. When we see new data $\mathcal{D}_{\text{new}}$, we optimise for the parameters $\{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ in the distribution $q(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$, and use $q_{\boldsymbol{\eta}_*}(\boldsymbol{w})$ as a prior.

Weight-space. The weight-space solution is exact in this simple setting (with $\tau = 1$). The objective is,

$$\mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[(N/\tau)\,\bar{\ell}_t(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w}) - \log q_{\boldsymbol{\eta}_*}(\boldsymbol{w})\right],$$
$$\text{where } \mathbb{E}_{q_{\boldsymbol{\eta}}(w)}[\log q_{\boldsymbol{\eta}_*}(\boldsymbol{w})] = -\tfrac{1}{2}\left(\text{Tr}(\boldsymbol{\Sigma}_*^{-1}\boldsymbol{\Sigma}) + (\boldsymbol{\mu} - \boldsymbol{\mu}_*)^\top \boldsymbol{\Sigma}_*^{-1}(\boldsymbol{\mu} - \boldsymbol{\mu}_*)\right) + \text{constant}. \tag{D.11}$$

var-FROMP. As explained in Chapter 4, var-FROMP optimises a slightly different objective, where the expectation of the log-prior term in weight-space is replaced with one in function-space. To simplify analysis in this section, we use Approximations 1 and 4 from Section 4.4, where we do not sample weights when converting to function-space, and set the weights at the previous means (although we do use a full-covariance matrix $\boldsymbol{\Sigma}$). For this simple model, the feature map $\boldsymbol{\phi}_i$ is fixed and constant. We calculate the functional prior $\tilde{q}_{w_*}(\boldsymbol{f}) = \mathcal{N}(\boldsymbol{f}; \boldsymbol{m}_*, \boldsymbol{K}_*)$, where $\boldsymbol{m}_*$ is the concatenation of $\boldsymbol{\phi}_i^\top \boldsymbol{\mu}_*$ over all datapoints $i \in \mathcal{D}_{\text{old}}$ (because we see/store all memorable points), and $\boldsymbol{K}_*$ has elements $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}_i^\top \boldsymbol{\Sigma}_* \boldsymbol{\phi}_j$. Letting the size of $\mathcal{D}_{\text{old}}$ be N_{old}, we therefore have $\boldsymbol{m}_* = \boldsymbol{\Phi}_* \boldsymbol{\mu}_*$, where $\boldsymbol{\Phi}_*$ stacks the N_{old} lots of feature vectors $\boldsymbol{\phi}_i$ into an $N_{\text{old}} \times P$ feature matrix. We also have an $N_{\text{old}} \times N_{\text{old}}$ matrix $\boldsymbol{K}_* = \boldsymbol{\Phi}_* \boldsymbol{\Sigma}_* \boldsymbol{\Phi}_*^\top$. Similarly, the current function-space distribution $\tilde{q}_{w_t}(\boldsymbol{f}) = \mathcal{N}(\boldsymbol{f}; \boldsymbol{m}, \boldsymbol{K})$, where we now calculate over $q_{\boldsymbol{\eta}}(\boldsymbol{w})$. Note that $\boldsymbol{m}$ and $\boldsymbol{K}$ are also calculated over past datapoints $\boldsymbol{x}_i \in \mathcal{D}_{\text{old}}$. Therefore $\boldsymbol{m} = \boldsymbol{\Phi}_* \boldsymbol{\mu}$ and $\boldsymbol{K} = \boldsymbol{\Phi}_* \boldsymbol{\Sigma} \boldsymbol{\Phi}_*^\top$.

Given this, var-FROMP optimises the following objective,

$$\mathbb{E}_{q_{\boldsymbol{\eta}}(\boldsymbol{w})}\left[(N/\tau)\,\bar{\ell}_t(\boldsymbol{w}) + \log q_{\boldsymbol{\eta}}(\boldsymbol{w})\right] - \mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{w_*}(\boldsymbol{f})\right],$$

where,

$$\begin{aligned}\mathbb{E}_{\tilde{q}_{w_t}(\boldsymbol{f})}\left[\log \tilde{q}_{w*}(\boldsymbol{f})\right] &= -\tfrac{1}{2}\left(\mathrm{Tr}(\boldsymbol{K}_*^{-1}\boldsymbol{K}) + (\boldsymbol{m}-\boldsymbol{m}_*)^\top \boldsymbol{K}_*^{-1}(\boldsymbol{m}-\boldsymbol{m}_*)\right) + c \\ &= -\tfrac{1}{2}\left[\mathrm{Tr}\big(\boldsymbol{\Phi}_*^\top[\boldsymbol{\Phi}_*\boldsymbol{\Sigma}_*\boldsymbol{\Phi}_*^\top]^{-1}\boldsymbol{\Phi}_*\,\boldsymbol{\Sigma}\big) + (\boldsymbol{\mu}-\boldsymbol{\mu}_*)^\top\boldsymbol{\Phi}_*^\top[\boldsymbol{\Phi}_*\boldsymbol{\Sigma}_*\boldsymbol{\Phi}_*^\top]^{-1}\boldsymbol{\Phi}_*\,(\boldsymbol{\mu}-\boldsymbol{\mu}_*)\right] + c.\end{aligned} \tag{D.12}$$

Comparing Equations D.11 and D.12, we can see the two are equal when,

$$\boldsymbol{\Sigma}_*^{-1} = \boldsymbol{\Phi}_*^\top\left[\boldsymbol{\Phi}_*\boldsymbol{\Sigma}_*\boldsymbol{\Phi}_*^\top\right]^{-1}\boldsymbol{\Phi}_*. \tag{D.13}$$

We can simplify this further by using the definition of $\boldsymbol{\Sigma}_*$ and applying the matrix inversion lemma twice, leading to equality when $\boldsymbol{\Phi}_*^\top(\boldsymbol{\Phi}_*\boldsymbol{\Phi}_*^\top)^{-1}\boldsymbol{\Phi}_* = \mathbf{I}$.

This indicates equality when the *left* pseudo-inverse of $\boldsymbol{\Phi}_*$ exists, $\boldsymbol{\Phi}_*^+ = \left(\boldsymbol{\Phi}_*^\top\boldsymbol{\Phi}_*\right)^{-1}\boldsymbol{\Phi}_*^\top$. This only exists when $\mathrm{rank}(\boldsymbol{\Phi}_*) = P$, which requires that the number of datapoints $N_{\text{old}} \geq P$. This will not be the case for large models with many parameters P, such as with (large) neural networks. Therefore var-FROMP does not always recover the exact solution, unlike weight-priors, even on linear regression. The same holds after making additional approximations to get FROMP (instead of var-FROMP). Note that with neural networks, the features $\boldsymbol{\phi}$ will no longer be the same for the current model and base model, as they also depend on the weight parameters (see Section 4.2), and this will introduce further inaccuracies in var-FROMP. Additionally, moving from linear regression to Generalised Linear Models will also only introduce further inaccuracies.

D.3 K-priors and equivalence to Gaussian Processes

In this section, we look at connections between variational K-priors and Gaussian Processes (expanding on Section 5.8), showing equivalence to online Gaussian Processes (GPs). We consider the 'Add Data' task, using the same notation as in Chapter 5 and specifically notation from the discussion regarding Support Vector Machines in Section 5.8.

We start by noting that, similarly to the representer theorem, the mean and covariance of $q_+(\boldsymbol{w})$ can be expressed in terms of the two N-length vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\lambda}$ (Opper and Archambeau, 2009; Khan et al., 2013; Khan, 2014),

$$\boldsymbol{\mu}_+ = \boldsymbol{\Phi}_+^\top\boldsymbol{\alpha}, \qquad \boldsymbol{\Sigma}_+ = (\boldsymbol{\Phi}_+^\top\boldsymbol{\Lambda}\boldsymbol{\Phi}_+ + \delta\mathbf{I})^{-1}, \tag{D.14}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix with $\boldsymbol{\lambda}$ as the diagonal. Using this, we can define a marginal $q(f_i) = \mathcal{N}(f_i; m_i, v_i)$, where $f_i = \boldsymbol{\phi}_i^\top \boldsymbol{w}$, with the mean and variance defined as follows,

$$m_i = \boldsymbol{\phi}_i^\top \boldsymbol{\mu}_+ \qquad = \boldsymbol{k}_{i,+}^\top \boldsymbol{\alpha}, \tag{D.15}$$

$$v_i = \boldsymbol{\phi}_i^\top \boldsymbol{\Sigma}_+ \boldsymbol{\phi}_i \qquad = k_{ii,+} - \boldsymbol{k}_{i,+}^\top \left(\boldsymbol{\Lambda}^{-1} + \delta \boldsymbol{K}_+\right)^{-1} \boldsymbol{k}_{i,+}, \tag{D.16}$$

where $k_{ii,+} = \boldsymbol{\phi}_i^\top \boldsymbol{\phi}_i$. Using these, we can now re-write the optimality conditions in the function-space to show equivalence to GPs (compare the following with Equation 5.68).

We show this for the first optimality condition (Equation 5.4),

$$\nabla_{\boldsymbol{\mu}} \mathbb{E}_q[\mathcal{L}(\boldsymbol{w})]|_{\boldsymbol{\mu}=\boldsymbol{\mu}_+, \boldsymbol{\Sigma}=\boldsymbol{\Sigma}_+} = \sum_{i \in \mathcal{D} \cup j} \mathbb{E}_{\mathcal{N}(\epsilon_i; 0, 1)} \left[\nabla_f \ell(y_i, h(f))|_{f=\boldsymbol{\phi}_i^\top \boldsymbol{\mu}_+ + \left(\boldsymbol{\phi}_i^\top \boldsymbol{\Sigma}_+ \boldsymbol{\phi}_i\right)^{1/2} \epsilon_i} \right] \boldsymbol{\phi}_i + \delta \boldsymbol{\mu}_+ \tag{D.17}$$

Multiplying by $\boldsymbol{\Phi}_+$, we can rewrite the gradient in the function space,

$$0 = \sum_{i \in \mathcal{D} \cup j} \mathbb{E}_{\mathcal{N}(\epsilon_i; 0, 1)} \left[\nabla_f \ell(y_i, h(f))|_{f=m_i + v_i^{1/2} \epsilon_i} \right] \boldsymbol{k}_{i,+} + \delta \boldsymbol{K}_+ \boldsymbol{\alpha} \tag{D.18}$$

$$= \sum_{i \in \mathcal{D} \cup j} \nabla_{m_i} \mathbb{E}_{q(f_i)} \left[\ell(y_i, h(f_i)) \right] \boldsymbol{k}_{i,+} + \delta \boldsymbol{K}_+ \boldsymbol{\alpha} \tag{D.19}$$

where $\boldsymbol{m}$ is the vector of m_i. Setting this to 0 gives us the first-order condition for a GP with respect to the mean (for example, see Equations 3.6 and 4.1 in Chapelle (2007)). It is easy to check this for GP regression, where $\ell(y_i, h(f_i)) = (y_i - f_i)^2$, and the equation becomes,

$$0 = \sum_{i \in \mathcal{D} \cup j} (m_i - y_i) \boldsymbol{k}_{i,+} + \delta \boldsymbol{K}_+ \boldsymbol{\alpha} \quad \Rightarrow \boldsymbol{\alpha} = (\boldsymbol{K}_+ + \delta \mathbf{I})^{-1} \boldsymbol{y}, \tag{D.20}$$

which is the quantity which gives us the posterior mean. A similar condition condition for the covariance can be written as well.

When we use a limited memory, some of the data examples are removed, and we get a sparse approximation similar to approaches like the informative vector machine, which uses a subset of data to build a sparse approximation (Herbrich et al., 2003). Better sparse approximations can be built by carefully designing the function-divergence term, and this is further discussed in Chapter 5 (see Equation 5.47) and Appendix D.1.

D.4 Further K-priors experiments and hyperparameters

This section starts by providing hyperparameters for experiments in Section 5.7. Most details are in Section 5.7 already, and in this section we provide hyperparameters for the moons visualisation on the right of Figure 5.1, the knowledge distillation experiment in Figure 5.4 and the continual learning experiment in Figure 5.6. Appendix D.4.1 performs an ablation study for Replay with different values for τ and randomly choosing points to store (instead of using the Lambda method). Appendix D.4.2 provides further results for weight-priors on the 'Add Data' tasks. Appendix D.4.3 studies the importance of the weight-divergence term in K-priors, where we see that this term is important to get good results. Appendix D.4.4 initialises weights randomly before training on a new task, instead of initialising at the base model values, and we obtain the same results as in Section 5.7.

Further details on the moons dataset on the right of Figure 5.1

To create this visualisation, we took 500 samples from the moons dataset, and split them into 5 splits of 100 datapoints each, with each split having 50 datapoints from each class. Additionally, the splits were ordered according to the x-axis, meaning the 1^{st} split was the left-most points, and the 5^{th} split had the right-most points. In the provided visualisations, we show transfer from 'past data' consisting of the first 3 splits (so, 300 datapoints) and the 'new data' consisting of the 4^{th} split (100 new datapoints). We store 3% of past data as past memory in K-priors, chosen as the points with the highest $h'(f^i_{\boldsymbol{w}_*})$ (the Lambda method).

Hyperparameters for knowledge distillation experiment in Figure 5.4

For the knowledge distillation task in Figure 5.4(c), we used K-priors with a temperature, similar to the temperature commonly used in knowledge distillation (Hinton et al., 2015). In our experiments, we use a temperature T on both the student and teacher logits, as written in the final term of Equation 5.33. We also multiply the final term by T^2 so that the gradient has the same magnitude as the other data term (as is common in knowledge distillation). We used $\lambda = 0.5$ in the experiment. We performed a hyperparameter sweep for the temperature (across $T = [1, 5, 10, 20]$), and used $T = 5$. For K-priors in this experiment, we optimise for 10 epochs instead of 100 epochs, and use $\tau = 1$.

We change from the CifarNet architecture to a LeNet5-style architecture. The CifarNet architecture is described in Section 2.4 and is from Zenke et al. (2017). The smaller LeNet5-style architecture has two convolution layers followed by two fully-connected layers: the first convolution layer has 6 output channels and kernel size 5, followed by the ReLU activation, followed by a Max Pool layer with kernel size 2 (and stride 2), followed by the second

convolution layer with 16 output channels and kernel size 5, followed by the ReLU activation, followed by another Max Pool layer with kernel size 2 (and stride 2), followed by a fully-connected layer with 120 hidden units, followed by the last fully-connected layer with 84 hidden units. We use ReLU activation functions in the fully-connected layers.

In Figure 5.4(b) we also showed initial results using a temperature on the 'Add Data' task on CIFAR-10. We used the same temperature from the knowledge distillation experiment ($T = 5$), but did not perform an additional hyperparameter sweep. We find that using a temperature improved results on CNNs, and we expect increased improvements if we perform further hyperparameter tuning. Note that many papers that use knowledge distillation perform more extensive hyperparameter sweeps than we have here, particularly over λ.

Hyperparameters for Quadratic K-priors on Split MNIST

For the Split MNIST experiment with Quadratic K-priors (see Figure 5.6), we used the same hyperparameters as the Split MNIST experiments with FROMP (values are in Appendix C.3). Specifically, we use the Adam optimiser with Adam learning rate set to 0.0001 and parameter $\beta_1 = 0.99$, and also employ gradient clipping. The minibatch size is 128, and we learn each task for 15 epochs. We use a fully connected multi-head network with two hidden layers, each with 256 hidden units and ReLU activation functions.

We perform a hyperparameter sweep for τ, which weights the weight-divergence term in Quadratic K-priors (and in weight-priors, or Online EWC (Schwarz et al., 2018)), sweeping over orders of magnitude ($\tau = [0.1, 1, 10, 100, 1000, 10000, ...]$). We also introduce a separate hyperparameter in front of the function-divergence term in K-priors, calling this ρ. For the weight-prior experiment, $\tau = 1000$, and for all Quadratic K-prior experiments, $\tau = 100$. When we store 40 past datapoints, $\rho = 50$, and we multiply this term by an appropriate factor when we reduce the memory size (for example, when we store 2 datapoints, $\rho = (40/2) \times 50 = 1000$). For vanilla K-priors with 40 past datapoints, $\rho = 1000$, and with 2 past datapoints, $\rho = 100,000$.

D.4.1 Replay with different τ and random memory

In Figure D.1 we provide an ablation study for Replay with different strategies: (i) we choose points by $h'(f^i_{\boldsymbol{w}_*})$ and use $\tau = N_{\text{old}}/M$, (ii) we choose points randomly and use $\tau = 1$, (iii) we choose points randomly and use $\tau = N_{\text{old}}/M$. Recall that N_{old} is the past dataset size (the size of $\mathcal{D}_{\text{old}}$) and M is the number of datapoints stored in memory (the size of $\mathcal{M}$). We see that choosing points by $h'(f^i_{\boldsymbol{w}_*})$ and using $\tau = 1$ performs very well, and we therefore choose this for all our experiments. This is then consistent with the memory in K-priors.

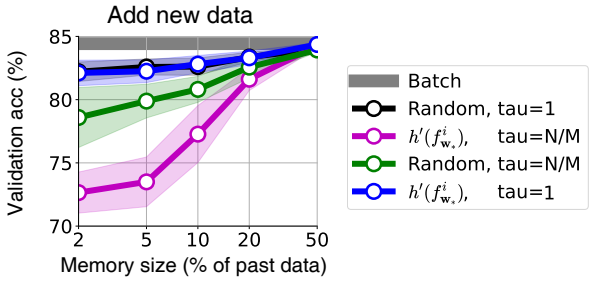


Figure D.1: This figure shows using $\tau = 1$ works well for Replay, both for random selection of memory and choosing memory by sorting $h'(f^i_{\boldsymbol{w}_*})$. We compare different methods for Replay on the UCI Adult 'Add Data' task. 'Random' means the points in memory are chosen randomly as opposed to choosing the points with highest $h'(f^i_{\boldsymbol{w}_*})$. We also consider using $\tau = N_{\text{old}}/M$ instead of $\tau = 1$. We choose to report memory chosen by $h'(f^i_{\boldsymbol{w}_*})$ in all experiments (this is then consistent with the memory in K-priors).

D.4.2 Further experiments with weight-priors

In Figure D.2 we provide results comparing with weight-priors for the 'Add Data' tasks for logistic regression on UCI Adult and neural networks on the 'USPS odd vs even' dataset. Other weight-prior results are in Figure 5.5. We see that for homogeneous data splits (such as UCI Adult, MNIST and CIFAR), weight-priors perform well. For heterogeneous data splits (the 'USPS odd vs even' dataset), weight-priors perform worse.

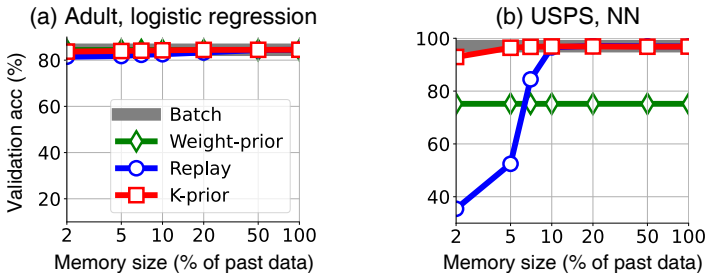


Figure D.2: Results on the 'Add Data' task, with a comparison to weight-priors. (a) Logistic regression on UCI Adult: on this homogeneous data split, weight-priors perform well. (b) Neural networks on the 'USPS odd vs even' dataset: on this heterogeneous data split, weight-priors perform worse.

D.4.3 K-priors ablation with weight-term

In this section we perform an ablation study on the importance of the weight-term in Equation 5.13. In Figure D.3 we show results on logistic regression on the 'USPS odd vs even' dataset, both with the correct K-prior term $\frac{1}{2}\delta\|\boldsymbol{w} - \boldsymbol{w}_*\|^2$, and with the incorrect $\frac{1}{2}\delta\|\boldsymbol{w}\|^2$. We see that using the correct weight-term always improves performance.

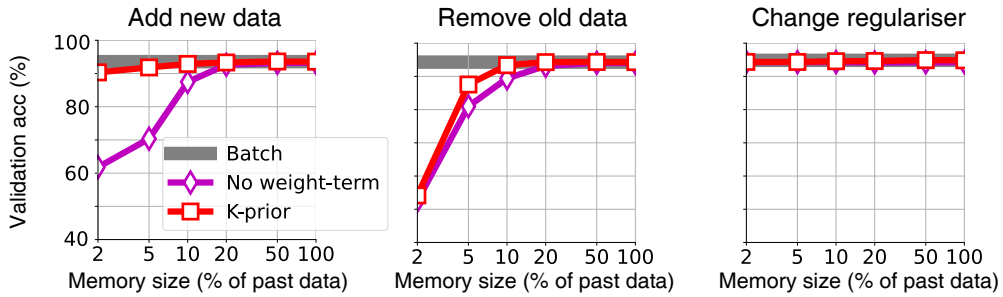


Figure D.3: Comparing K-priors with a version of K-priors without the correct weight-term on logistic regression on the 'USPS odd vs even' dataset. We see that using the correct weight-term is important, especially on the 'Add Data' task.

D.4.4 K-priors with random initialisation

In most of our experiments in Section 5.7, when we train on a new task, we initialise the parameters at the base model parameters $\boldsymbol{w}_*$. Note that this is not possible in the 'Change model class/architecture' task, where weights were initialised randomly. We also initialised randomly for our results in Table 5.1. In this section, we show that our results are independent of initialisation strategy: we get the same results whether we use random initialisation or initialise at previous values. The only difference is that random initialisation can sometimes take longer to converge (for all methods: Batch, Replay and K-priors).

For Generalised Linear Models, where we always train until convergence and there is a single optimum, it is clear that the exact same solution will always be reached. We now also provide the result for neural networks on the 'USPS odd vs even' dataset with random initialisation in Figure D.4, for the 3 tasks where we had earlier initialised at previous values $\boldsymbol{w}_*$ (compare with Figure 5.3(a)). We use exactly the same hyperparameters and settings as in Figure 5.3(a), aside from initialisation method, and find we obtain the same results.

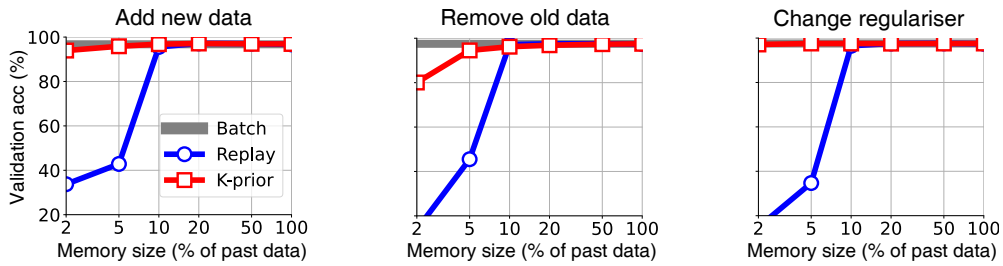


Figure D.4: K-priors obtain the same results when randomly initialising the weights for the 'Add new data', 'Remove old data' and 'Change regulariser' tasks on neural networks on the 'USPS odd vs even' dataset. Previous results, including Figure 5.3(a), initialised parameters at previously learnt values $\boldsymbol{w}_*$. The 'Change model class/architecture' task originally used random initialisation and so is not repeated here.