Université de Montréal

Advances in Parameterisation, Optimisation and Pruning of Neural Networks

par

César Laurent

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Informatique

6 octobre 2020

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée

Advances in Parameterisation, Optimisation and Pruning of Neural Networks

présentée par

César Laurent

a été évaluée par un jury composé des personnes suivantes :

Aaron Courville
(président-rapporteur)

Pascal Vincent
(directeur de recherche)

Simon Lacoste-Julien
(membre du jury)

Jimmy Ba
(examinateur externe)

Résumé

Les réseaux de neurones sont une famille de modèles de l'apprentissage automatique qui sont capable d'apprendre des tâches complexes directement des données. Bien que produisant déjà des résultats impressionnants dans beaucoup de domaines tels que la reconnaissance de la parole, la vision par ordinateur ou encore la traduction automatique, il y a encore de nombreux défis dans l'entraînement et dans le déploiement des réseaux de neurones. En particulier, entraîner des réseaux de neurones nécessite typiquement d'énormes ressources computationnelles, et les modèles entraînés sont souvent trop gros ou trop gourmands en ressources pour être déployés sur des appareils dont les ressources sont limitées, tels que les téléphones intelligents ou les puces de faible puissance. Les articles présentés dans cette thèse étudient des solutions à ces différents problèmes.

Les deux premiers articles se concentrent sur l'amélioration de l'entraînement des réseaux de neurones récurrents (RNNs), un type de réseaux de neurones particulier conçu pour traiter des données séquentielles. Les RNNs sont notoirement difficiles à entraîner, donc nous proposons d'améliorer leur paramétrisation en y intégrant la normalisation par lots (BN), qui était jusqu'à lors uniquement appliquée aux réseaux non-récurrents. Dans le premier article, nous appliquons BN aux connections des entrées vers les couches cachées du RNN, ce qui réduit le décalage covariable entre les différentes couches ; et dans le second article, nous montrons comment appliquer BN aux connections des entrées vers les couches cachées et aussi des couches cachée vers les couches cachée des réseau récurrents à mémoire court et long terme (LSTM), une architecture populaire de RNN, ce qui réduit également le décalage covariable entre les pas de temps. Nos expériences montrent que les paramétrisations proposées permettent d'entraîner plus rapidement et plus efficacement les RNNs, et ce sur différents bancs de tests.

Dans le troisième article, nous proposons un nouvel optimiseur pour accélérer l'entraînement des réseaux de neurones. Les optimiseurs diagonaux traditionnels, tels que RMSProp, opèrent dans l'espace des paramètres, ce qui n'est pas optimal lorsque plusieurs paramètres sont mis à jour en même temps. A la place, nous proposons d'appliquer de tels optimiseurs dans une

base dans laquelle l'approximation diagonale est susceptible d'être plus efficace. Nous tirons parti de l'approximation K-FAC pour construire efficacement cette base propre Kronecker-factorisée (KFE). Nos expériences montrent une amélioration en vitesse d'entraînement par rapport à K-FAC, et ce pour différentes architectures de réseaux de neurones profonds.

Le dernier article se concentre sur la taille des réseaux de neurones, i.e. l'action d'enlever des paramètres du réseau, afin de réduire son empreinte mémoire et son coût computationnel. Les méthodes de taille typique se base sur une approximation de Taylor de premier ou de second ordre de la fonction de coût, afin d'identifier quels paramètres peuvent être supprimés. Nous proposons d'étudier l'impact des hypothèses qui se cachent derrière ces approximations. Aussi, nous comparons systématiquement les méthodes basées sur des approximations de premier et de second ordre avec la taille par magnitude (MP), et montrons comment elles fonctionnent à la fois avant, mais aussi après une phase de réapprentissage. Nos expériences montrent que mieux préserver la fonction de coût ne transfère pas forcément à des réseaux qui performent mieux après la phase de réapprentissage, ce qui suggère que considérer uniquement l'impact de la taille sur la fonction de coût ne semble pas être un objectif suffisant pour développer des bon critères de taille.

Mots clés. Réseaux de neurones, apprentissage profond, réseaux de neurones récurrents, normalisation par lots, taille non structurée, gradient naturel, factorisation de Kronecker.

Abstract

Neural networks are a family of Machine Learning models able to learn complex tasks directly from the data. Although already producing impressive results in many areas such as speech recognition, computer vision or machine translation, there are still a lot of challenges in both training and deployment of neural networks. In particular, training neural networks typically requires huge amounts of computational resources, and trained models are often too big or too computationally expensive to be deployed on resource-limited devices, such as smartphones or low-power chips. The articles presented in this thesis investigate solutions to these different issues.

The first couple of articles focus on improving the training of Recurrent Neural Networks (RNNs), networks specially designed to process sequential data. RNNs are notoriously hard to train, so we propose to improve their parameterisation by upgrading them with Batch Normalisation (BN), a very effective parameterisation which was hitherto used only in feed-forward networks. In the first article, we apply BN to the input-to-hidden connections of the RNNs, thereby reducing internal covariate shift between layers. In the second article, we show how to apply it to both input-to-hidden and hidden-to-hidden connections of the Long Short-Term Memory (LSTM), a popular RNN architecture, thus also reducing internal covariate shift between time steps. Our experiments show that these proposed parameterisations allow for faster and better training of RNNs on several benchmarks.

In the third article, we propose a new optimiser to accelerate the training of neural networks. Traditional diagonal optimisers, such as RMSProp, operate in parameters coordinates, which is not optimal when several parameters are updated at the same time. Instead, we propose to apply such optimisers in a basis in which the diagonal approximation is likely to be more effective. We leverage the same approximation used in Kronecker-factored Approximate Curvature (K-FAC) to efficiently build this Kronecker-factored Eigenbasis (KFE). Our experiments show improvements over K-FAC in training speed for several deep network architectures.

The last article focuses on network pruning, the action of removing parameters from the network, in order to reduce its memory footprint and computational cost. Typical pruning methods rely on first or second order Taylor approximations of the loss landscape to identify which parameters can be discarded. We propose to study the impact of the assumptions behind such approximations. Moreover, we systematically compare methods based on first and second order approximations with Magnitude Pruning (MP), showing how they perform both before and after a fine-tuning phase. Our experiments show that better preserving the original network function does not necessarily transfer to better performing networks after fine-tuning, suggesting that only considering the impact of pruning on the loss might not be a sufficient objective to design good pruning criteria.

Keywords. Neural Networks, Deep Learning, Recurrent Neural Networks, Batch Normalisation, Unstructured Pruning, Natural Gradient, Kronecker Factorisation.

Contents

List of Tables

List of Figures

List of Abbreviations

BiRNN	Bidirectional Recurrent Neural Network
BN	Batch Normalisation
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional Neural Network
EKFAC	Eigenvalue-corrected Kronecker Factorisation
FLOPS	Floating-point Operations per Second
GD	Gradient Descent
GGN	Generalised Gauss-Newton
GPU	Graphical Processing Unit
K-FAC	Kronecker-factored Approximate Curvature
KFE	Kronecker-factored Eigenbasis
LM	Linear Model
LN	Layer Normalisation
LSTM	Long Short-Term Memory
ML	Machine Learning
MLP	Multilayer Perceptron
MP	Magnitude Pruning
MSE	Mean Squared Error
OBD	Optimal Brain Damage
OBS	Optimal Brain Surgeon
PTB	Penn Treebank
QM	Quadratic Model
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
TPU	Tensor Processing Unit
WSJ	Wall Street Journal

Acknowledgements

I first would like to thank my family and girlfriend for their unconditional support in this adventure, and especially Arantxa Casanova for her patience and understanding during the stressful deadline times. I would also like to thank Thomas Schweitzer and Anne-Marie Petter for their generous hospitality, which greatly eased my transition to the somewhat hostile environment that is Montréal during the winter.

I would then like to thank Yoshua Bengio, for accepting me at Mila and supervising me the first couple of years of my PhD, and Pascal Vincent, for his supervision during the remainder of my studies. During the interactions I had the chance to have with them, they showed me how to approach challenging problems and properly formulate new and exciting research questions, and I am extremely grateful for that.

I would also like to thank all my other co-authors for their collaborations: Camille Ballas, Nicolas Ballas, Xavier Bouthillier, Philémon Brakel, Tim Cooijmans, Aaron Courville, Thomas George, Çağlar Gülçehre, Gabriel Pereyra and Ying Zhang. I would like to thank Thomas George and Nicolas Ballas, as this thesis would not have happened without their help and all the fruitful interactions we had.

Finally, I would like to thank all the amazing people whose paths I crossed during my PhD, whether at Mila or playing rugby for either the Parc Olympique or for the University of Montréal's Carabins. I would particularly like to thank Adriana Romero, Michał Drożdżal, Francesco Visin, Lluís Castrejón, Faruk Ahmed, Alexandre de Brébisson, Anthony Di Ioia, Christian Dansereau and Francis Grégoire for all the great times we had.

Chapter 1

Introduction

The performance of Artificial Intelligence (AI) systems has tremendously increased over the past decade. Neural networks, which are models that can be trained to automatically solve complex tasks by learning directly from the data, are at the root of this success (Dahl et al., 2011; Krizhevsky et al., 2012). They have proven to be extremely effective in many research area, including for instance speech recognition (Baevski et al., 2020), machine translation (Edunov et al., 2018), image recognition (Xie et al., 2020), as well as in medical applications such as for instance skin cancer classification (Esteva et al., 2017) or automatic analysis of chest radiographs (Nam et al., 2019). They were also at the heart of the *AlphaGo* system, which defeated professional players in the game of Go (Silver et al., 2016). Finally, they could also be useful in the fight against climate change (Rolnick et al., 2019), which is arguably one of the major challenges we will face in the next decades.

However, training these large-scale neural networks requires a huge amount of computational resources. For instance, one of the neural networks used for the speech recognition system of Baevski et al. (2020) was trained for more than 5 days on 128 V100 GPUs, learning from more than 53 thousand hours of audio signal. Similarly, the image recognition system of Xie et al. (2020), which learned from more than 300 million images, was trained for 6 days on a TPU v3 Pod with 2048 core, whose total processing power is greater than 100 petaFLOPS (Google, 2018). On top of these gigantic computational requirements, we should add all the resources used during the development and design of these final neural networks. These massive computations also have a huge impact in terms of carbon emissions (Lacoste et al., 2019). For instance, the carbon emitted by all the networks we trained for the articles presented in this thesis, which required orders of magnitude less computational power than the systems presented above, is estimated to be equivalent to flying across the Atlantic multiple times.

Thus, developing neural networks and algorithms that allow for faster training in an important research direction that can have a drastic impact not only on the computational resources neural networks require, but also on their carbon footprint. Similarly, designing methods to reduce the size of trained neural networks to further reduce their computational cost and memory requirements is also an important research direction, and can allow to generalise the deployment of powerful AI systems on resource-limited devices, such as smartphones (Han et al., 2015a).

All the articles presented in this document have therefore the common objective of reducing the computational cost associated with neural networks. The first couple of articles, Laurent et al. (2016) and Cooijmans et al. (2016), presented in Chapter 4 and 6 respectively, introduce new parameterisations of neural networks used to process sequential data, allowing to train them faster and and reach better performance. The third article, George et al. (2018), presented in Chapter 8, introduces a new optimiser that can increase the training speed of neural networks. Finally the fourth article, Laurent et al. (2020), presented in Chapter 10, studies methods to reduce the size of trained neural networks, which lowers their computational requirements for inference.

1.1. Structure of this Document

This thesis is structured as follows. Chapter 2 introduces the relevant background material on neural networks, including their parameterisation, the algorithms used to train them, as well as methods to reduce their size. Chapters 3 to 10 contain the four articles presented in this thesis. Each article is introduced by a prologue chapter, which details the contribution of the different authors, the context of the article, its contributions, as well as recent developments. Finally, Chapter 11 contains concluding remarks, as well as potential future research directions.

Chapter 2

Background

The goal of this chapter is to provide the background information which is necessary to understand the articles presented in this thesis. We first quickly present some core concepts of Machine Learning. We then formally introduce neural networks, along with the basic algorithm used to train them, and highlight some of the issues that arise during training. Afterwards, we present the typical parameterisations and later optimisers that are used in practice to mitigate these issues. Finally, we briefly introduce the topic of network pruning, whose objective is to reduce the size of neural networks in order to ease their deployment on devices with limited resources.

2.1. Machine Learning Basics

2.1.1. Introduction

To predict the weather for the next day given the current temperature and air pressure, a programmer can design by hand an algorithm in order to automatically perform such prediction. However, as the dimensionality and complexity of the inputs increase, for instance when the inputs are natural images or audio signals, designing such algorithms can rapidly become tedious or even impossible. Machine Learning (ML) studies models and algorithms that are able to automatically solve tasks by *learning* from the data. The different types of tasks that ML algorithms can handle are typically categorised as follows:

- In **supervised learning**, the data is composed of examples with their associated targets, and the goal is to train models that predict the correct target when inputted with its associated example.

- In **unsupervised learning**, there are no targets associated with the examples, so the goal is to learn a representation of the data.
- In **clustering**, the goal is to discover the underlying structure in the data in order to partition it into small groups.
- In **density estimation**, the goal is to learn an estimate of the unknown distribution that generated the data.
- In **reinforcement learning**, an agent learns to interact and behave in an environment, in order to maximise a reward.

This thesis focuses exclusively on supervised learning, and more particularly on how to improve the models and their training and ease their deployment.

2.1.2. Empirical Risk Minimisation

Let's consider a model $m \in \mathcal{M}$ from a family of models $\mathcal{M}$ and a loss function $\ell(m(\mathbf{x}), \mathbf{t})$ that measures how close to the target $\mathbf{t}$ the prediction of the model m is, with $\mathbf{x}$ as input. We can compute the expected risk $\mathcal{R}(m)$ associated with m, i.e. how well the model performs on average on the data:

$$\mathcal{R}(m) = \mathop{\mathbb{E}}_{p(\mathbf{x},\mathbf{t})}[\ell(m(\mathbf{x}), \mathbf{t})] \tag{2.1}$$

where $p(\mathbf{x},\mathbf{t})$ is the true data distribution. This distribution is typically unknown, so it is usually approximated with a training dataset $\mathcal{D} = \{(\mathbf{x}_i,\mathbf{t}_i)\}_{1 \leq i \leq N}$, composed of N (example, target) pairs. We can then use this dataset to empirically estimate the expected risk:

$$\mathcal{R}(m) \approx \mathcal{L}(m) = \frac{1}{N} \sum_{i=1}^{N} \ell(m(\mathbf{x}_i), \mathbf{t}_i) \tag{2.2}$$

where $\mathcal{L}(m)$ is the empirical risk associated with the model m. *Empirical Risk Minimisation* is about finding the model m^* among the family of models $\mathcal{M}$ that minimises the empirical risk:

$$m^* = \arg\min_{m \in \mathcal{M}} \frac{1}{N} \sum_{i=1}^{N} \ell(m(\mathbf{x}_i), \mathbf{t}_i) \tag{2.3}$$

In other words, we want to find the model m^* that best fits the data, and this fitting is measured by the loss function ℓ.

2.1.3. Model Capacity and Generalisation

A core concept in ML is the *capacity* of a model, i.e. its representation power. A model with a small capacity would probably perform badly in modelling a non-linear mapping from

input to targets. This phenomenon is called *under-fitting*. One could think that the bigger the capacity, the better the model. However, when training a model, we actually care about its *generalisation* performances, i.e. how well the model minimises $\mathcal{R}(m)$, which measures the performances on the true data distribution $p(\mathbf{x},\mathbf{t})$, rather than how well it minimises $\mathcal{L}(m)$, which measures the performances on the training data. Indeed, if a model has too much representation power, it can *over-fit*, namely learn "by heart" all the examples of the training data, and will totally fail on unseen data.

To monitor both under- and over-fitting, the dataset $\mathcal{D}$ is split into a training set, used to train the model m, and a test set, used to estimate $\mathcal{R}(m)$. Both the training algorithm and the model family might also have *hyper-parameters*, such as the size of the model, that require manual tuning. In such cases, to avoid over-fitting on the test set, the training set is further split to generate a validation set that will be used to find the best hyper-parameters, before evaluating the performance of the selected model on the test set.

2.2. Neural Networks Basics

2.2.1. Definition

Neural networks are families of models $f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$ that map an input space $\mathcal{X}$ to an output space $\mathcal{Y}$, $\boldsymbol{\theta} \in \mathbb{R}^D$ being the vector containing all the trainable parameters of the network. Neural networks are typically organised in *layers*, each layer eventually containing trainable parameters:

$$f_{\boldsymbol{\theta}} = f^L_{\boldsymbol{\theta}^L} \circ f^{L-1}_{\boldsymbol{\theta}^{L-1}} \circ \cdots \circ f^1_{\boldsymbol{\theta}^1} \tag{2.4}$$

where the exponent indexes the layers. $\boldsymbol{\theta}$ is thus the concatenation of the parameters of each layer $\{\boldsymbol{\theta}^1, \ldots, \boldsymbol{\theta}^L\}$. Usually, neural networks interleave layers containing parameters together with layers that implement non-linear *activation* functions, which allow $f_{\boldsymbol{\theta}}$ to learn a potentially non-linear mapping between $\mathcal{X}$ and $\mathcal{Y}$. The bigger the number of layers L, the *deeper* the network is. Current state-of-the-art networks have hundreds of layers and millions of parameters (He et al., 2016b), hence the name *Deep Learning*.

2.2.2. Training

Gradient Descent. Neural networks are trained by seeking parameters $\boldsymbol{\theta}^*$ that minimise the empirical risk $\mathcal{L}(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ \mathcal{L}(\boldsymbol{\theta}) \tag{2.5}$$

That minimisation is usually performed using *gradient descent* (GD), an iterative procedure which takes small steps in the direction opposite to the gradient of $\mathcal{L}(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{2.6}$$

where η is the learning rate, a hyper-parameter controlling the step size. $\mathcal{L}(\boldsymbol{\theta})$ and its gradient can be computed using either all the N examples of the training set, as in the original GD, only one example at a time, as in *Stochastic Gradient Descent* (SGD), or a small subset of the training set, called a *mini-batch*. One usually rely on mini-batches for training in practice, in order to take full advantage of the computational and parallelisation power that GPUs can offer.

Backpropagation. To train neural networks, we have to compute the gradient of the empirical risk $\mathcal{L}(\boldsymbol{\theta})$ with respect to all the parameters in $\boldsymbol{\theta}$. Taking advantage of both the modularity of neural networks (Equation 2.4) and the chain rule of the derivatives, we can decompose the gradient of $\mathcal{L}(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}^l$ of an arbitrarily layer l as:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^l} = \frac{\partial f^l_{\boldsymbol{\theta}^l}}{\partial \boldsymbol{\theta}^l} \frac{\partial f^{l+1}_{\boldsymbol{\theta}^{l+1}}}{\partial f^l_{\boldsymbol{\theta}^l}} \cdots \frac{\partial f^L_{\boldsymbol{\theta}^L}}{\partial f^{L-1}_{\boldsymbol{\theta}^{L-1}}} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial f^L_{\boldsymbol{\theta}^L}} \tag{2.7}$$

This can be efficiently implemented by computing the gradient layer by layer, starting from the last layer L, and *backpropagating* it through the network up to the first layer. The computation that each layer needs to perform can be summarised in two equations:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^l} = \frac{\partial f^l_{\boldsymbol{\theta}^l}}{\partial \boldsymbol{\theta}^l} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial f^l_{\boldsymbol{\theta}^l}} \tag{2.8}$$

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial f^{l-1}_{\boldsymbol{\theta}^{l-1}}} = \frac{\partial f^l_{\boldsymbol{\theta}^l}}{\partial f^{l-1}_{\boldsymbol{\theta}^{l-1}}} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial f^l_{\boldsymbol{\theta}^l}} \tag{2.9}$$

where Equation 2.8 computes the gradient with respect to the input of the layer and Equation 2.9 computes the gradient with respect to its own parameters. An important point to keep in mind is that the ability to successfully train neural networks is tightly linked with the flow of gradients through the network, as we will see in the following Sections.

Loss functions for training. So far the loss function ℓ in the empirical risk $\mathcal{L}(\boldsymbol{\theta})$ has not been properly defined. Since the gradient of $\mathcal{L}(\boldsymbol{\theta})$ are required to train the network, ℓ must be differentiable. For regression tasks, where targets $\mathbf{t}$ are real-valued, the Mean Squared Error (MSE) loss is typically used:

$$\ell_{MSE}(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{t}_i) = \frac{1}{2} \left\| f_{\boldsymbol{\theta}}(\mathbf{x}_i) - \mathbf{t}_i \right\|_2^2 \tag{2.10}$$

For classification tasks, we can not directly optimise for the error rate, since it is not differentiable. Instead, we optimise the cross-entropy as a surrogate for the error rate:

$$\ell_{CE}(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{t}_i) = -\sum_{k=1}^{K} \mathbf{t}_{ik} \log\left(f_{\boldsymbol{\theta}}(\mathbf{x}_i)_k\right) \tag{2.11}$$

where K is the number of classes. $\mathbf{t}_i$ is here a *one-hot* vector, which represents a multinoulli distribution, and is filled with zeros except at the correct class index where it takes the value 1.

2.2.3. Difficulties in Neural Networks Training

Before going further, we highlight here some of the issues that arise when training and using neural networks; issues that the articles presented in this thesis aim to address. We redirect the reader to Goodfellow et al. (2016) for a more complete overview of all the other issues that might arise during training, and that we omitted here for brevity.

Since there are almost no restrictions on the shape neural networks can take (Equation 2.4), the loss landscape $\mathcal{L}(\boldsymbol{\theta})$ can be arbitrarily complex. In fact, because of the non-linear activations functions that are interleaved in the neural networks, one cannot assume that $\mathcal{L}(\boldsymbol{\theta})$ is convex, so $\mathcal{L}(\boldsymbol{\theta})$ can contain local minima in which Gradient Descent can get stuck. Fortunately this is not an issue in practice, as these local minima are usually of high quality, which means the loss at these points is close to the loss at the global minimum (Choromanska et al., 2015).

However, the Hessian of $\mathcal{L}(\boldsymbol{\theta})$ is also typically ill-conditioned (Sagun et al., 2017; Papyan, 2018; Alain et al., 2019), which translates to a loss landscape with some directions along which the curvature can be extremely high, while the curvature in other directions might be very low. In such cases, the steps taken during Gradient Descent (Equation 2.6) will most likely be too large along the directions with high curvature, and almost zero along the directions with low curvature. Some parameters might thus receive large gradients, while other parameters might not get any gradient at all. These issues make the training of neural networks a challenging task. Figure 2.1 pictures an example of Gradient Descent optimisation on a 2D loss landscape and shows how the optimisation progresses depending on the learning rate η: When η is too large, the optimisation might diverge, and when η is too low, the optimisation might require a huge amount of computationally expensive iterations. Note that Figure 2.1 greatly understates the problem, as the highest curvature of the ravine might be several orders of magnitude larger than the smallest one.

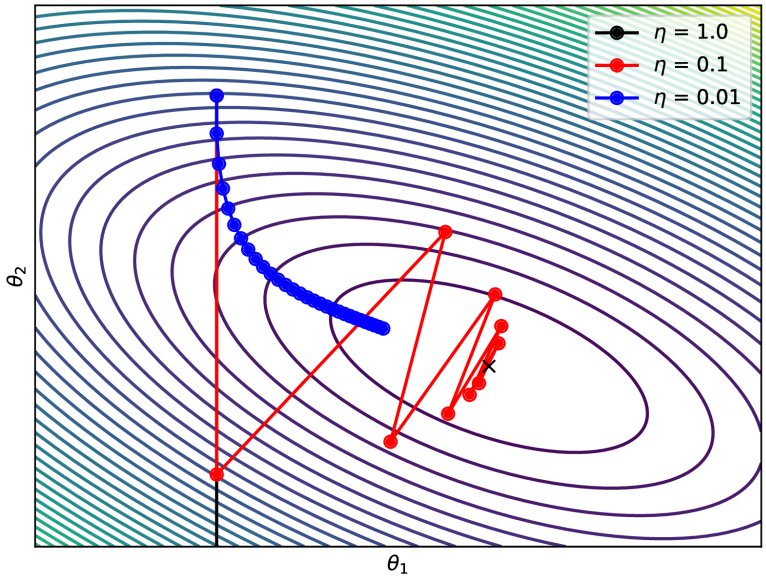


Figure 2.1. Example of iterations of Gradient Descent on a 2D loss landscape, represented by a contour plot with its minimum being denoted by a black cross, starting from the upper left portion of the plot. Each bullet point denotes an iteration of GD, and each colour represents a different η (see Equation 2.6). When η is too large (e.g. 1.0, in black, going out of the plot), GD might produce steps that are too large which can make the learning diverge and miss the minimum. With a smaller η (e.g. 0.1, in red), the optimisation might bounce off walls along steep direction, while progressing more slowly along flatter directions. When η is too small (e.g. 0.01, in blue), the optimisation might require a lot of iterations before reaching the minimum.

Overcoming training difficulties. Two main strategies have been used by ML researchers to help overcome these training issues. The first one, which is the basis for the success of Deep Learning, is to improve the *parameterisation* of neural networks, so that they can be trained more easily or perform better. Indeed, what matters is the function that the network learns, not the exact parameterisation of that function. So except for computational constraints, which we detail more in Section 2.5, there are virtually no constraints on the exact parameterisation of $f_{\boldsymbol{\theta}}$. Current practices in network parameterisation are presented in Section 2.3.

The second strategy is to develop *optimisers* that can navigate more easily than Gradient Descent on loss landscapes with pathological curvature, or that can provide better updates by adapting the step size according to the curvature. Section 2.4 presents the optimisers that are used in this thesis.

2.3. Parameterisation of Neural Networks

We now detail the different layers $f^l_{\theta^l}$ that are typically used to build neural networks and are employed throughout this thesis. Again, we redirect the reader to Goodfellow et al. (2016) for a more complete overview.

2.3.1. Basic Layers

Linear layer. The simplest building block for neural networks is the linear, or fully-connected layer. It performs an affine transformation using a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$:

$$f_{\{\mathbf{W},\mathbf{b}\}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.12}$$

A neural network composed solely of a linear layer is performing linear regression. A Multilayer Perceptron (MLP) is a neural network composed of several linear layers, interleaved with activation functions. An Auto-Encoder is a special kind of MLP composed of two parts: one Encoder, which maps the input to a smaller-sized code, and a Decoder, which maps the code back to the original input space.

Convolution layer. One parameter-efficient way of processing images or sequences is to perform a convolution operation with a kernel of parameters $\mathbf{W}$:

$$f_{\{\mathbf{W},\mathbf{b}\}}(\mathbf{x}) = \mathbf{W} * \mathbf{x} + \mathbf{b} \tag{2.13}$$

where $*$ denotes the convolution operation. See Dumoulin & Visin (2016) for an excellent guide on convolutions for deep learning. Neural network containing convolution layers are called Convolutional Neural Networks (CNNs), and are used to process images (LeCun et al., 1989), and other sequential data such as natural language (Collobert et al., 2011) or speech signal (e.g. Zhang et al. (2017)).

Activation Functions. Activation functions are inserted between convolution and linear layers to allow the network to model non-linear functions, thus increasing its representation power. The activations used in this thesis are listed in Table 2.1. Older neural networks were based on the sigmoid and hyperbolic tangent function, while the more recent ones typically use Rectified Linear Units (ReLUs) (Nair & Hinton, 2010), or one of its many variants (Maas et al., 2013; He et al., 2015; Clevert et al., 2015). The softmax activation function transforms its inputs such that they become non-negative and sum to 1. The last layer of networks that are trained to perform classification tasks is typically a softmax activation function, so that its output can be interpreted as a multinouilli distribution.

Table 2.1. Activation functions used in this thesis.

Name	Function
Sigmoid	$f(x) = \frac{1}{1+\exp(-x)}$
Hyperbolic tangent (tanh)	$f(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
Rectified Linear Unit (ReLU)	$f(x) = \max(0, x)$
Softmax	$f(\mathbf{x})_i = \frac{\exp(\mathbf{x}_i)}{\sum_j \exp(\mathbf{x}_j)}$

Initialisation schemes. As stated earlier, we need to ensure good gradient propagation in order to be able to train neural networks properly. However, activation functions can saturate, so they actually reduce the amplitude of the signal propagated through the network, which has harmful effects on gradient propagation. To ensure good gradient propagation, initialisation schemes for parameters in linear and convolution layers have been proposed in the literature: Glorot & Bengio (2010) derived such initialisation scheme for hyperbolic tangent and sigmoid activations, and He et al. (2015) did the same for ReLU activations.

2.3.2. Batch Normalisation

In a neural network, one can think of an arbitrary layer as receiving samples from a distribution that is shaped by the previous layer. This distribution changes during the course of training, making any layer but the first responsible not only for learning a good representation but also for adapting to a changing input distribution. To reduce this variation in distribution, called *Internal Covariate Shift*, Batch Normalisation (BN) (Ioffe & Szegedy, 2015) uses mini-batch statistics to standardise each feature. Given a mini-batch of data $\mathbf{x} \in \mathbb{R}^{M \times K}$ composed of M samples and K features, the sample mean and sample variance of each feature k can be computed along the mini-batch axis:

$$\mu_k = \frac{1}{M}\sum_{i=1}^{M} \mathbf{x}_{i,k} \qquad \text{and} \qquad \sigma_k^2 = \frac{1}{M}\sum_{i=1}^{M}(\mathbf{x}_{i,k} - \mu_k)^2 \tag{2.14}$$

Using these statistics, one can standardise each intermediate activation of the network. However, this reduces the representation power of the network, as all the features are constrained to have zero mean and unit variance. To account for this issue, BN introduces two additional trainable parameters γ and β, which respectively scale and shift the data, leading to the following BN transformation:

$$BN_{\{\gamma_k,\beta_k\}}(\mathbf{x}_k) = \gamma_k \frac{\mathbf{x}_k - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}} + \beta_k \tag{2.15}$$

where ϵ is a small positive constant to improve numerical stability. By setting γ_k to σ_k and β_k to μ_k, the network could recover the original layer representation, if it was the optimal thing to do. This normalisation is part of the network, so the gradients are backpropagated through the means and variance computations.

BN is applied in between the linear or convolution layers and the activation functions, which we denote φ here:

$$\mathbf{y} = \varphi(BN(\mathbf{W}\mathbf{x})) \tag{2.16}$$

The bias vector in the linear or convolution layer can be removed, since its effect is cancelled by the mean removal of BN, and is replaced by β after the normalisation.

Benefits of BN. BN reduces the Internal Covariate Shift by making the forward pass invariant to the scale of the weight matrix (Ioffe & Szegedy, 2015). Indeed, for a rescaling parameter α, we have:

$$BN(\alpha\mathbf{W}\mathbf{x}) = BN(\mathbf{W}\mathbf{x}) \tag{2.17}$$

Moreover, the scale of the weight matrix does not affect the Jacobian of the transformation (Ioffe & Szegedy, 2015) since:

$$\frac{\partial BN(\alpha\mathbf{W}\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial BN(\mathbf{W}\mathbf{x})}{\partial \mathbf{x}} \tag{2.18}$$

Also, BN introduces a regularisation effect by stabilising the growth of the parameters (Ioffe & Szegedy, 2015). Indeed, the gradient of the output with respect to the parameters has the following property:

$$\frac{\partial BN(\alpha\mathbf{W}\mathbf{x})}{\partial \alpha\mathbf{W}} = \frac{1}{\alpha}\frac{\partial BN(\mathbf{W}\mathbf{x})}{\partial \mathbf{W}} \tag{2.19}$$

So bigger weights, which are typically associated with over-fitting (Bishop, 2006), will receive smaller gradients. The downside of BN is that it only works with mini-batch SGD, with a mini-batch size large enough to produce good estimates of the statistics. Otherwise, these statistics might be too noisy, which could hurt the training procedure.

Inference with BN. During inference, one can not rely on the statistics of the mini-batch, because the prediction of the network for a given example should not depend on the other examples in the mini-batch. Instead, population statistics should be estimated by either forwarding several training mini-batches through the network, or by maintaining a running average calculated over each mini-batch seen during training, which is typically used in practice.

2.3.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) (Rumelhart et al., 1985) are a special kind of neural networks designed to process sequential data, such as natural language or speech signal. Contrarily to CNNs, which have a relatively narrow receptive field and are designed to extract localised information, RNNs can learn *long-term dependencies*, i.e. extract information that can potentially be scattered throughout the sequence: In natural language for instance, neural networks might have to understand relationships between words that can be separated by long and irrelevant subordinate clauses.

Formulation. RNNs take an input sequence of T vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ and produce, by applying recursively the same function $f_{\boldsymbol{\theta}}$, a sequence of states $(\mathbf{h}_1, \ldots, \mathbf{h}_T)$. At each time step, $f_{\boldsymbol{\theta}}$ usually relies on the current input $\mathbf{x}_t$ and the previous state $\mathbf{h}_{t-1}$ to compute the next state, as follows:

$$\mathbf{h}_t = f_{\boldsymbol{\theta}}(\mathbf{h}_{t-1}, \mathbf{x}_t) \tag{2.20}$$

where $\boldsymbol{\theta}$ contains the parameters of $f_{\boldsymbol{\theta}}$, which includes the initial hidden state $\mathbf{h}_0$. To form deeper architectures, RNNs can be stacked on top of each other, using the sequence of states produced by the previous RNN as inputs for the next one (Graves et al., 2013b):

$$\mathbf{h}_t^l = f_{\boldsymbol{\theta}^l}^l(\mathbf{h}_{t-1}^l, \mathbf{h}_t^{l-1}) \tag{2.21}$$

Figure 2.2 pictures an example of a stack composed of 2 RNNs, unrolled on 3 time steps. The simplest RNN structure is the Tanh-RNN, defined as:

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}) \tag{2.22}$$

where $\mathbf{W}_h$, $\mathbf{W}_x$ are the hidden-to-hidden and input-to-hidden weight matrices, respectively, and $\mathbf{b}$ is a bias vector.

Training RNNs. RNNs are usually trained using Backpropagation Through Time (Rumelhart et al., 1985), i.e. the backpropagation is applied on the time-unrolled model (pictured in Figure 2.2). The loss function can be computed using either parts or the whole sequence outputted by the RNN, depending on the task at hand. In *sequence classification* for instance, where the goal is to classify the whole sequence, one can use a RNN to encode the whole sequence, and feed the last hidden state produced by the RNN to a final MLP that will perform the classification. On the other hand, in *frame-wise classification*, there is one target for each time step in the sequence, and the goal is to correctly classify each time step of the sequence. In that case, the total training loss is simply a sum or average of the losses $(\mathcal{L}_1(\boldsymbol{\theta}), \ldots, \mathcal{L}_T(\boldsymbol{\theta}))$ computed at each time step. Finally, in the more complex general case where the size of the target sequence does not match the size of the input sequence, one need

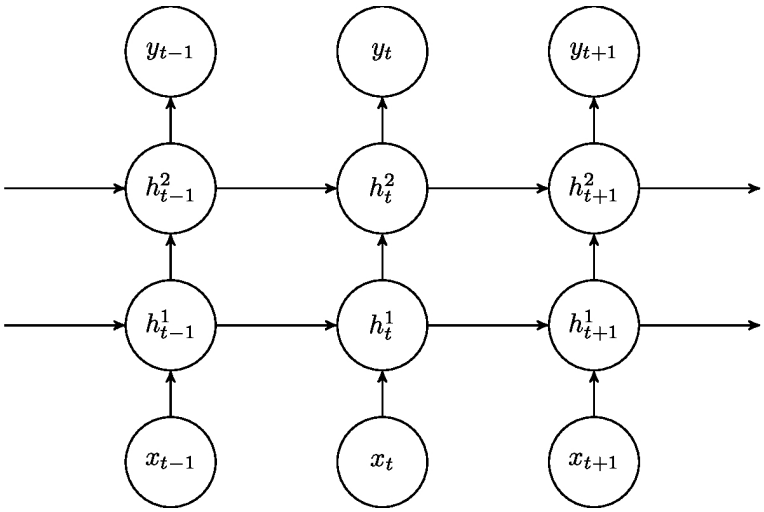


Figure 2.2. 3 time steps of a stack of 2 RNNs.

to rely on more advanced adaptations, such as Connectionist Temporal Classification (Graves et al., 2006) or Attention Mechanisms (Bahdanau et al., 2015) to perform the alignment.

Exploding and vanishing gradients. Tanh-RNN are notoriously hard to train, because of exploding and vanishing gradients (Hochreiter et al., 2001; Pascanu et al., 2013b). Using the equation of the Tanh-RNN for reference (Equation 2.22), the gradient of the loss at time step $t+K$, $\mathcal{L}_{t+K}(\boldsymbol{\theta})$, with respect to the input $\mathbf{x}_t$ at time step t can be computed using the chain rule:

$$\frac{\partial \mathcal{L}_{t+K}(\boldsymbol{\theta})}{\partial \mathbf{x}_t} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{x}_t} \left(\prod_{k=t}^{t+K-1} \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \right) \frac{\partial \mathcal{L}_{t+K}(\boldsymbol{\theta})}{\partial \mathbf{h}_{t+K}} \tag{2.23}$$

The Jacobian $\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$ controls the propagation of the gradients through time. Neglecting the effect of the hyperbolic tangent, one can see that the Jacobian of the Tanh-RNN is simply the weight matrix $\mathbf{W}_h$. So if one of its eigenvalues λ_i is different than 1, the gradients along the corresponding eigenvector will either vanish (when $\lambda_i < 1$) or explode (when $\lambda_i > 1$) (Pascanu et al., 2013b).

Mitigating exploding and vanishing gradients. A simple solution to exploding gradients is to clip the gradients if they exceed a given threshold (Pascanu et al., 2013b). To solve both exploding and vanishing gradient problems, one can use an orthogonal matrix for the hidden-to-hidden transition. However, maintaining the orthogonality of the matrix is, in general, too expensive computationally ($\mathcal{O}(n^3)$, where n is the size of the matrix). Also, Vorontsov et al. (2017) showed that strict orthogonality might not be desirable, as it can hurt the training. Another solution is to construct the matrix by composing unitary transformations in the complex domain (Arjovsky et al., 2016; Wisdom et al., 2016). Finally, an old but extremely effective solution is to use the Long Short-Term Memory (LSTM)

architecture (Hochreiter & Schmidhuber, 1997), which is a special RNN structure designed to alleviate the vanishing gradient problem thanks to gating mechanisms and memory cells. It will be presented in more details in Chapter 4.

2.3.4. Improving the Parameterisation of RNNs

Since RNNs are widely used to process sequential data, improving the parameterisation of RNNs is an active area of research. The first two articles presented in this thesis line up in that direction: We first show in Chapter 4 how to apply BN to any kind of RNN architecture, hitherto only used in feed-forward networks, and then propose an improved version of the LSTM architecture that better leverages BN in Chapter 6.

2.4. Optimisers for Neural Networks

Although great care is taken to parameterise and initialise neural networks to ease gradient propagation, it might still be difficult to move around the loss landscape $\mathcal{L}(\boldsymbol{\theta})$ following only stochastic gradient information. To alleviate this issue, one can use the more fancy optimisers presented below to train neural networks more efficiently.

2.4.1. First-order Optimisers

Because of their simplicity and good performance, first-order optimisers such as SGD are by far the most popular choices of optimisers used to train neural networks. These optimisers do not consider interactions between parameters, and thus update each $\boldsymbol{\theta}_i$ independently. The most popular ones, which are the ones used in this thesis, are listed below.

Momentum. Inspired from physics, the momentum method adds a velocity term $\mathbf{v}$ to the parameter update, increasing the update in directions that the gradient consistently indicates:

$$\mathbf{v} \leftarrow \mu\mathbf{v} + \frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{2.24}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\mathbf{v} \tag{2.25}$$

where μ is a hyper-parameter, usually set to 0.9. The momentum method works extremely well in practice, and is the typical choice to optimise feed-forward networks, such as MLPs and CNNs.

RMSProp. The idea behind RMSProp (Tieleman & Hinton, 2012), which is based on AdaGrad (Duchi et al., 2011), is to have one learning rate per parameter, instead of having only one single learning rate controlling the step size of all parameters. AdaGrad proposes to divide the current gradient of each parameter by the square root of the sum of all the squared gradients that each parameter received so far. RMSProp improves on this method by only maintaining a running average $\mathbf{r}$ of the squared gradients, instead of the whole gradient history:

$$\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\left(\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right)^2 \tag{2.26}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\frac{1}{\sqrt{\mathbf{r}+\epsilon}} \odot \frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{2.27}$$

where the hyper-parameter ρ (usually set to 0.9) is the forgetting factor of the running average, and ϵ is a small positive constant that ensures numerical stability. RMSProp is typically used to optimise networks whose gradients are badly conditioned, such as RNNs.

Adam. Kingma & Ba (2015) further improves RMSProp by maintaining a running average not only of the second moment of the gradients, but also of the first moment. It also corrects for the bias associated with the initialisation of the running averages:

$$t \leftarrow t+1 \tag{2.28}$$

$$\mathbf{v} \leftarrow \beta_1\mathbf{v} + (1-\beta_1)\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \tag{2.29}$$

$$\mathbf{r} \leftarrow \beta_2\mathbf{r} + (1-\beta_2)\left(\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\right)^2 \tag{2.30}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\frac{\sqrt{(1-\beta_2^t)}}{(1-\beta_1^t)}\frac{1}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{v} \tag{2.31}$$

where the hyper-parameters β_1 and β_2 are the forgetting factors of the running average of the first and second moments $\mathbf{v}$ and $\mathbf{r}$, respectively, and ϵ is a small positive constant that ensures numerical stability. The default values of 0.9 for β_1 and 0.999 for β_2 originally proposed by Kingma & Ba (2015) work extremely well in practice.

2.4.2. Higher-order Optimisers

As it can be observed in Figure 2.3, another interesting issue when following the direction given by the gradient is that it might not be indicating the direction one should follow. Indeed, the steps taken by Gradient Descent point towards the steepest direction, which is only aligned with the direction of the minimum if the loss landscape is a circular bowl. We

present here two methods that have been developed to alleviate this issue: Newton's Methods, pictured in Figure 2.3, which corrects the direction of the gradient by taking into account the curvature of the loss landscape, and Natural Gradient (Amari, 1998), which computes the steps in the function space rather than in the parameter space.

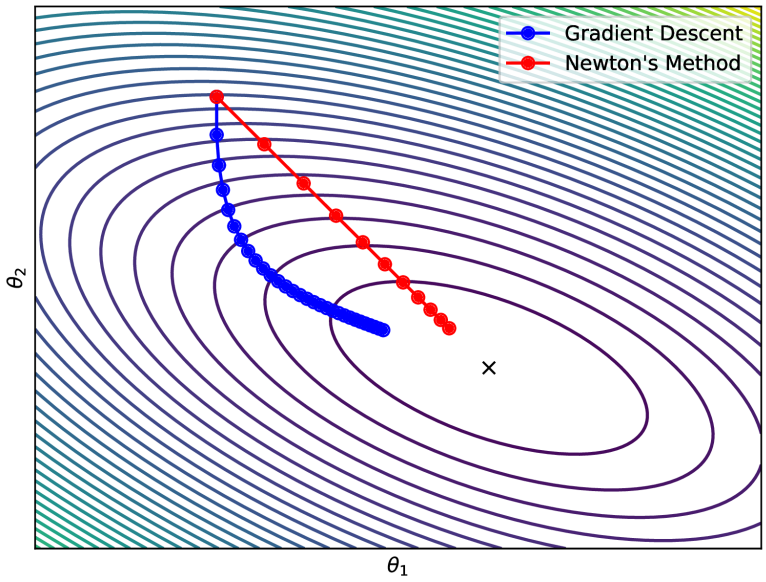


Figure 2.3. Same as Figure 2.1, but comparing the updates of Gradient Descent (blue), with the updates of Newton's method (red). The direction followed by Gradient Descent, which is the steepest direction, is not aligned with the direction of the minimum. By taking into account the curvature of the loss landscape, Newton's method corrects for this issue, and the steps it takes are directed towards the minimum.

Newton's Method. To take the curvature of the loss landscape into account when updating the parameters of the network, Newton's Method relies on a second-order Taylor expansion of $\mathcal{L}(\boldsymbol{\theta})$, rather than a first-order one like in Gradient Descent (Equation 2.6):

$$\mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \approx \mathcal{L}(\boldsymbol{\theta}) + \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}^{\top} \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^{\top}\mathbf{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} \tag{2.32}$$

where $\mathbf{H}(\boldsymbol{\theta}) \in \mathbb{R}^{D\times D}$ is the Hessian of $\mathcal{L}(\boldsymbol{\theta})$. Assuming that $\mathbf{H}(\boldsymbol{\theta})$ is invertible, one can derive the update rule of Newton's Method:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\mathbf{H}^{-1}(\boldsymbol{\theta})\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{2.33}$$

where η is an optional learning rate. This update rule is similar to the GD update rule (Equation 2.6), with an extra preconditioning of the gradient by the inverse of the Hessian.

For quadratic $\mathcal{L}(\boldsymbol{\theta})$ and with $\eta = 1$, Newton's Method reaches the minimum in a single update (Nocedal & Wright, 2006).

Generalised Gauss-Newton. Despite the nice convergence properties of Newton's Method (Nocedal & Wright, 2006), $\mathbf{H}(\boldsymbol{\theta})$ might not be necessarily positive definite, which means Newton's Method might take steps in the wrong direction. To alleviate this issue, Schraudolph (2002) proposed to rely on $\mathbf{G}(\boldsymbol{\theta})$, the Generalised Gauss-Newton (GGN) approximation of $\mathbf{H}(\boldsymbol{\theta})$ instead:

$$\mathbf{H}(\boldsymbol{\theta}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} \frac{\partial f_{\boldsymbol{\theta}}(x_i)}{\partial \boldsymbol{\theta}}^{\top} \nabla^2_{u=f_{\boldsymbol{\theta}}(x_i)} \ell(u, t_i) \frac{\partial f_{\boldsymbol{\theta}}(x_i)}{\partial \boldsymbol{\theta}}}_{\mathbf{G}(\boldsymbol{\theta}),\ \text{the Generalised Gauss-Newton}} + \underbrace{\sum_{k}^{K} \frac{\partial \ell(u, t_i)}{\partial u_k}\Big|_{u=f_{\boldsymbol{\theta}}(x_i)} \frac{\partial^2 f_{\boldsymbol{\theta}}(x_i)_k}{\partial \boldsymbol{\theta}^2}}_{\approx 0} \tag{2.34}$$

$$\approx \mathbf{G}(\boldsymbol{\theta}) \tag{2.35}$$

where K is the number of outputs of the network. The advantage of $\mathbf{G}(\boldsymbol{\theta})$ over $\mathbf{H}(\boldsymbol{\theta})$ is that it is positive semi-definite by construction (Schraudolph, 2002).

Natural Gradient. The update rule in GD (Equation 2.6) can be seen as minimising the following problem (for small enough η and regular enough $\mathcal{L}(\boldsymbol{\theta})$):

$$\Delta\boldsymbol{\theta}^* = \underset{\Delta\boldsymbol{\theta}}{\operatorname{argmin}}\ \mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) + \frac{1}{2\eta}\|\Delta\boldsymbol{\theta}\|_2^2 \tag{2.36}$$

In other words, we search for a step vector $\Delta\boldsymbol{\theta}$ that minimises $\mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$, and this $\Delta\boldsymbol{\theta}$ should be small, in terms of Euclidean distance, so that we do not move too far from the current parameter vector $\boldsymbol{\theta}$. This constraint on the parameter space is rather odd, because we do care about the function that the network $f_{\boldsymbol{\theta}}$ computes, not about the exact values that $\boldsymbol{\theta}$ takes. So instead of measuring how the network changes in the parameter space, we could measure how the function computed by the network changes. This is the idea behind Natural Gradient (Amari, 1998): For networks $f_{\boldsymbol{\theta}}$ that output probability distributions, e.g. when equipped with a softmax or a sigmoid layer at the output, one can measure the functional change using the Kullback-Leibler divergence D_{KL}, leading to the following objective:

$$\Delta\boldsymbol{\theta}^* = \underset{\Delta\boldsymbol{\theta}}{\operatorname{argmin}}\ \mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) + \frac{1}{\eta} D_{KL}(f_{\boldsymbol{\theta}}(y \mid x) \mid\mid f_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(y \mid x)) \tag{2.37}$$

D_{KL} can then be approximated using a second-order Taylor expansion around $\boldsymbol{\theta}$:

$$D_{KL}(f_{\boldsymbol{\theta}}(y \mid x) \mid\mid f_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(y \mid x)) \approx \frac{1}{2}\Delta\boldsymbol{\theta}^{\top}\mathbf{F}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} \tag{2.38}$$

Note that the first-order term in the Taylor expansion of D_{KL} is zero by construction. $\mathbf{F}(\boldsymbol{\theta}) \in \mathbb{R}^{D \times D}$, the Hessian of D_{KL}, is the Fisher Information Matrix:

$$\mathbf{F}(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{x,y \sim p(x,y)} \left[\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial \boldsymbol{\theta}} \left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial \boldsymbol{\theta}} \right)^{\top} \right] \tag{2.39}$$

Injecting this approximation into Equation 2.37, we have:

$$\Delta \boldsymbol{\theta}^* \approx \mathop{\text{argmin}}_{\Delta \boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta} + \Delta \boldsymbol{\theta}) + \frac{1}{2\eta} \Delta \boldsymbol{\theta}^{\top} \mathbf{F}(\boldsymbol{\theta}) \Delta \boldsymbol{\theta} \tag{2.40}$$

As for Gradient Descent, we can now derive the Natural Gradient Descent update rule using a first-order Taylor approximation of $\mathcal{L}(\boldsymbol{\theta} + \Delta \boldsymbol{\theta})$ around $\boldsymbol{\theta}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \mathbf{F}^{-1}(\boldsymbol{\theta}) \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{2.41}$$

This update rule is similar to the update rule of Newton's Method (Equation 2.33), but with the Hessian $\mathbf{H}(\boldsymbol{\theta})$ replaced by the Fisher Information Matrix $\mathbf{F}(\boldsymbol{\theta})$.

Links between $\mathbf{F}(\boldsymbol{\theta})$ and $\mathbf{G}(\boldsymbol{\theta})$. Interestingly, there are strong links between $\mathbf{F}(\boldsymbol{\theta})$ and $\mathbf{G}(\boldsymbol{\theta})$. Schraudolph (2002) showed that for networks with linear outputs trained to minimise MSE, $\mathbf{F}(\boldsymbol{\theta})$ and $\mathbf{G}(\boldsymbol{\theta})$ are equal, and Pascanu & Bengio (2013) demonstrated similar equality for networks that output probability distributions and trained to minimise cross-entropy, such as in typical classification setups.

Block-diagonal approximation. Unfortunately, since $\mathbf{F}(\boldsymbol{\theta}) \in \mathbb{R}^{D \times D}$, computing its inverse has a computational complexity of $\mathcal{O}(D^3)$, which quickly becomes intractable, even for tiny network. Further approximations are thus required to be able to use Natural Gradient Descent or Newton's Method to train modern neural networks. A typical first approximation is to assume this matrix is block-diagonal, each block $\mathbf{F}^l(\boldsymbol{\theta})$ containing only the parameters of a single layer l, thus assuming there is no correlation between the gradients on the parameters of one layer and the gradients of any other layer. Block-diagonal matrices are cheaper to inverse, since they can be inverted block by block.

Kronecker-factored Approximate Curvature. Sadly, a layer l can still contain several million parameters, so $\mathbf{F}^{l^{-1}}(\boldsymbol{\theta})$ might still be intractable. Heskes (2000) and then later Martens & Grosse (2015) proposed to further approximate the blocks of linear layers, artfully leveraging the outer product structure of the gradient on the weight matrix $\mathbf{W}^l$. To ease the notations, we use homogeneous coordinates and incorporate the bias vector $\mathbf{b}^l$ into $\mathbf{W}^l$. So for a linear layer l in the network:

$$f^l_{\mathbf{W}^l}(\mathbf{h}^l) = \mathbf{W}^l \mathbf{h}^l \tag{2.42}$$

we can derive the approximation which is at the heart of Kronecker-factored Approximate Curvature (K-FAC) (Martens & Grosse, 2015):

$$\mathbf{F}^l(\boldsymbol{\theta}) = \mathop{\mathbb{E}}_{x,y\sim p(x,y)} \left[\operatorname{vec}\left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial \mathbf{W}^l} \right) \operatorname{vec}\left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial \mathbf{W}^l} \right)^\top \right] \tag{2.43}$$

$$= \mathop{\mathbb{E}}_{x,y\sim p(x,y)} \left[\operatorname{vec}\left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial f^l_{\mathbf{W}^l}} \mathbf{h}^{l\top} \right) \operatorname{vec}\left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial f^l_{\mathbf{W}^l}} \mathbf{h}^{l\top} \right)^\top \right] \tag{2.44}$$

$$= \mathop{\mathbb{E}}_{x,y\sim p(x,y)} \left[\left(\mathbf{h}^l \mathbf{h}^{l\top} \right) \otimes \left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial f^l_{\mathbf{W}^l}} \left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial f^l_{\mathbf{W}^l}} \right)^\top \right) \right] \tag{2.45}$$

$$\approx \underbrace{\mathop{\mathbb{E}}_{x,y\sim p(x,y)} \left[\mathbf{h}^l \mathbf{h}^{l\top} \right]}_{\mathbf{A}} \otimes \underbrace{\mathop{\mathbb{E}}_{x,y\sim p(x,y)} \left[\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial f^l_{\mathbf{W}^l}} \left(\frac{\partial \log(f_{\boldsymbol{\theta}}(y \mid x))}{\partial f^l_{\mathbf{W}^l}} \right)^\top \right]}_{\mathbf{B}} \tag{2.46}$$

where $\operatorname{vec}(\cdot)$ denotes the vectorisation operation, and $\otimes$ denotes the Kronecker product. Equation 2.46 is the K-FAC approximation of $\mathbb{F}^l(\boldsymbol{\theta})$, where the backpropagated signal on the output of the layer $f^l_{\mathbf{W}^l}$ is considered independent of the input of the layer $\mathbf{h}^l$. For $\mathbf{W}^l \in \mathbb{R}^{M\times N}$, K-FAC approximates $\mathbf{F}^l(\boldsymbol{\theta}) \in \mathbb{R}^{MN\times MN}$ with the two smaller factors $\mathbf{A} \in \mathbb{R}^{N\times N}$ and $\mathbf{B} \in \mathbb{R}^{M\times M}$ which are both tractable. Moreover, one can compute cheaply the inverse of $\mathbf{F}^l(\boldsymbol{\theta})$, thank to the following Kronecker products property:

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{2.47}$$

Finally, we can incorporate this approximation into Newton's Method (Equation 2.33) or Natural Gradient (Equation 2.41) update rules, leading to the following K-FAC update rule:

$$\mathbf{W}^l \leftarrow \mathbf{W}^l - \eta \mathbf{B}^{-1} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \mathbf{W}^l} \mathbf{A}^{-1} \tag{2.48}$$

To further reduce the computational cost of K-FAC, the computation and inversion of $\mathbf{A}$ and $\mathbf{B}$ is typically amortised over several updates, assuming the geometry of the loss landscape does not vary too much between successive updates.

Extensions of K-FAC. K-FAC have been extended to other types of layers: Grosse & Martens (2016) extended K-FAC for convolution layers; Ba et al. (2017) proposed another approximation for linear layers that follow the flattening operation in CNNs; Martens et al. (2018) extended K-FAC for RNNs; finally we showed in Laurent et al. (2018) that the factorisation for convolutions can be approximated even further, thus reducing even more its computational cost.

2.4.3. Improving Neural Networks Optimisers

As training neural networks is computationally intensive, designing more efficient optimisers to allow faster or better training is an active area of research. In the article presented in Chapter 8, we investigate the combination of using running averages, as in first-order optimisers, but applied in a basis constructed from the K-FAC approximation.

2.5. Neural Network Pruning

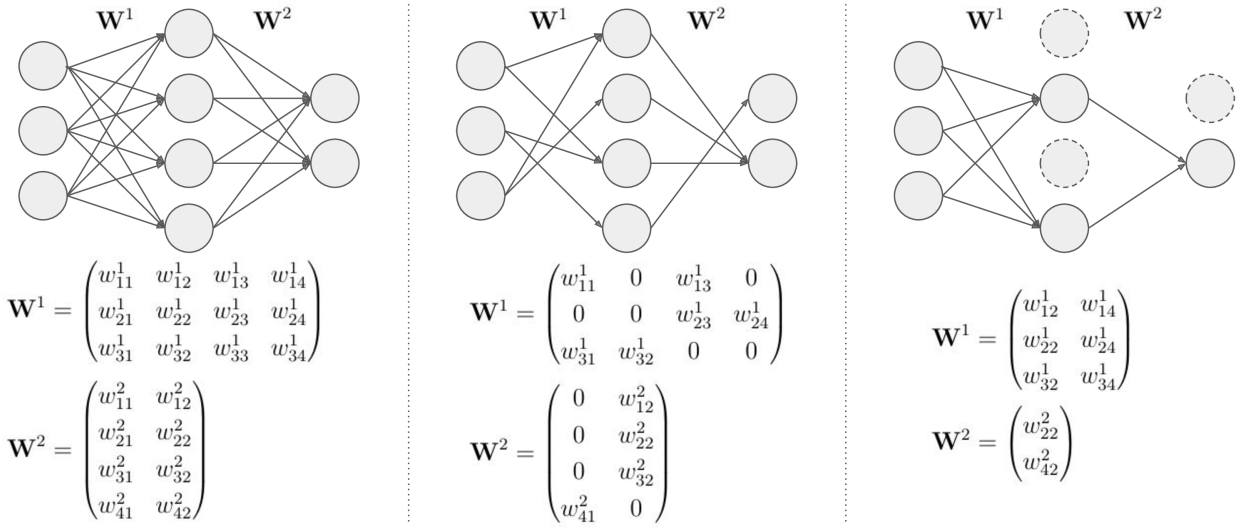
Since neural networks are getting bigger and bigger, thus requiring more memory and computational power, it is becoming harder and harder to deploy them on limited-resources devices, such as smartphones or embedded systems. And since it is typically easier to train larger models than smaller ones (Hinton et al., 2012), several strategies have been developed to reduce the size and computational requirements of trained neural networks. One can use for instance approximate decomposition of the weight matrices and convolution kernels (e.g. Denton et al. (2014); Moczulski et al. (2015)). One can also distil the knowledge learned in one or several big network into a smaller one (Romero et al., 2014; Hinton et al., 2015). Another strategy consists of quantising the parameters of the model to reduce their memory footprint (e.g. Courbariaux et al. (2016); Wu et al. (2016)). Finally, network pruning, the focus of this Section, proposes to reduce the size of networks by removing some parameters, i.e. setting them to zero (e.g. LeCun et al. (1990); Han et al. (2015b)).

2.5.1. Structured and Unstructured Pruning

Network pruning can be done in an *unstructured* way, allowing to remove any parameter of the network, or in a *structured* way, removing entire rows and columns of the matrices and kernels in linear and convolution layers. Both these variants are pictured in Figure 2.4. Unstructured pruning (Figure 2.4 (b)) leads to sparse weight matrices, which require very high levels of sparsity in order to be able to leverage sparse implementations to obtain computational speedups and memory savings (Han et al., 2016).

On the other hand, structured pruning (Figure 2.4 (c)) leads to matrices with smaller dimensions, so models pruned that way are directly ready to be deployed. However, it also reduces the dimensions of the intermediate representations computed by the network, so reducing the size of the network might be more challenging with structured pruning. Also, there is some evidence that a network pruned in a structured way can achieve a similar level

of performance than a network of the same reduced size trained from scratch, and it does not seems to be the case for unstructured pruning (Liu et al., 2019). The remainder of this Section focuses on unstructured pruning.



(a) Original network **(b)** Unstructured pruning **(c)** Structured pruning

Figure 2.4. Pruning variants. **(a)** Picture of a small network composed of 2 linear layers, with their corresponding weight matrices $\mathbf{W}^1$ and $\mathbf{W}^2$ displayed below. **(b)** Illustration of unstructured pruning, which removes connections from the network, leading to sparser weight matrices. **(c)** Illustration of structured pruning, which removes units from the network, leading to matrices with smaller dimensions.

2.5.2. Selecting which Parameters to Prune

The question is now to decide which parameters to prune and which parameters to keep. This is a combinatorial problem which is intractable even for tiny networks, so one needs to rely on heuristics to find approximate solutions. We separate here two families of methods: the *penalty-based*, and the *ranking-based* methods.

Penalty-based methods. One first family of pruning methods relies on penalties that are applied during the training of the network, slowly pushing some parameters to zero. Such penalties can be for instance variational dropout (Molchanov et al., 2017), L_0 regularisation (Louizos et al., 2017), L_1 regularisation on the γ parameters of BN (Liu et al., 2017; Ye et al., 2018), or soft thresholding (Kusupati et al., 2020). Ding et al. (2019) proposed to skip gradient updates on the parameters that receive the smallest gradients from $\mathcal{L}(\boldsymbol{\theta})$, and only apply L_2 regularisation to these parameters. Finally, Lin et al. (2020) uses a feedback signal to reactivate weights that might have been pruned prematurely. The main drawback

of penalty-based methods is that it can be hard to balance the penalty with the training loss: A stronger penalty might hinder the training of the network, while a weaker one might yield networks that are not sparse enough.

Ranking-based methods. The oldest family of methods aims at ranking the parameters in order of importance, according to a pruning *criterion*, so that the least important parameters can be safely removed from the network, while the most important ones shall be kept (LeCun et al., 1990). To compute the importance of each parameter one can use a criterion that approximate how much removing it would impact $\mathcal{L}(\boldsymbol{\theta})$, relying on either first-order (Lee et al., 2019b) or second-order (LeCun et al., 1990; Hassibi & Stork, 1993; Zeng & Urtasun, 2019; Wang et al., 2019; Singh & Alistarh, 2020) Taylor approximations, or one can approximate its impact on the gradient flow (Lee et al., 2019a; Wang et al., 2020). Finally, one can simply use the magnitude of each parameter as importance criterion (Han et al., 2015b).

2.5.3. Pruning Frameworks

Contrarily to penalty-based methods, which are applied during training, ranking-based methods can be applied at any time. For instance, pruning phases can be interleaved with training phases to iteratively reduce the size of the network. We present below some of these different pruning *frameworks*.

Foresight pruning. In foresight pruning, the pruning is performed before the training of the network, and thus the training of the model will be performed on an already sparse architecture. For instance, Lee et al. (2019b) proposes to use a first-order Taylor approximation of $\mathcal{L}(\boldsymbol{\theta})$ before the initial training. However, as noted by Lee et al. (2019a) and Wang et al. (2020), removing parameters with low impact on $\mathcal{L}(\boldsymbol{\theta})$ does not make much sense for untrained networks which essentially produce random predictions. Instead, they both propose to keep the parameters that will ensure a good gradient flow, and thus an hopefully smooth training, even with a highly sparse neural network.

Iterative pruning and fine-tuning. As proposed by LeCun et al. (1990), pruning can be performed in an iterative manner, where the network is first trained, and then iterations of pruning and fine-tuning are performed until the desired sparsity level is reached. At the time of writing this thesis, such iterative scheme, coupled with the magnitude of the parameters as importance criterion (Han et al., 2015b), is still the state-of-the-art in network pruning (Renda et al., 2020). The downside of pruning and fine-tuning iteratively is its computational cost, since several iterations of training are required.

One-shot pruning. One-shot pruning is a special case of iterative pruning, where only one iteration of pruning and fine-tuning is performed (e.g. Zeng & Urtasun (2019); Wang et al. (2019)). This has obvious computational advantages over performing several iterations of fine-tuning. It can also be used to compare different pruning criteria before fine-tuning, which has the advantage of removing all the sources of variations that are associated with fine-tuning. Unfortunately, this is something that is usually not reported in practice (with the exception of Wang et al. (2019) and Singh & Alistarh (2020)). Another situation where one-shot pruning can be used is when pruning a pre-trained model whose training data is not available for an eventual fine-tuning phase.

Lottery Ticket. Finally, one last framework to mention is the Lottery Ticket framework. Based on the hypothesis that networks contain smaller sub-networks (the "winning lottery tickets") that can reach similar performance than the original network, the Lottery Ticket framework is essentially an iterative pruning framework where, after each pruning iterations, the parameters that have not been pruned are reset to their value at initialisation (Frankle & Carbin, 2018). However, Renda et al. (2020) showed that not resetting the parameters to their initial value, i.e. performing a classical iterative pruning, works better than the Lottery Ticket framework, so there is no reason to use the Lottery Ticket framework for pruning in practice.

2.5.4. Towards better Understanding Loss-based Pruning Criteria

As the reader probably noticed, there are endless possible combinations of pruning criteria and pruning frameworks, which can hinder proper comparison between pruning methods (Gale et al., 2019; Blalock et al., 2020). Moreover, and somewhat surprisingly, magnitude-based criteria work extremely well in practice (Renda et al., 2020), and have comparable performance to criteria based on Taylor approximations of the loss (Blalock et al., 2020). One possible hypothesis is that the assumptions behind using Taylor approximations, which are often overlooked in practice, could be the cause of the somewhat mitigated performance of loss-based criteria. The last article of this thesis, presented in Chapter 10, investigates the impact of these assumptions when using Taylor approximations for pruning.

Chapter 3

Prologue to the First Article

3.1. Article Details

Batch Normalized Recurrent Neural Networks. César Laurent [1], Gabriel Pereyra[1], Philémon Brakel, Ying Zhang and Yoshua Bengio, In *Proceedings of the 41st IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2016).*

Authors contributions. I developed the method presented in this article, its implementation, ran the experiment of speech alignment prediction and wrote most of the article. Yoshua Bengio discovered that Gabriel Pereyra, a student at the University of Southern Carolina, was working on a similar idea, so we decided to add his language modelling experiment to the article. Philémon Brakel was the direct supervisor of the project. He and Ying Zhang were the author of the code base used in the speech alignment prediction experiment. Yoshua Bengio was the supervisor of the project.

3.2. Context

Ioffe & Szegedy (2015) introduced BN, a parameterisation that drastically improved the training speed and performance of state-of-the-art neural networks. Originally applied to CNNs, BN set new standards of performance on ImageNet (Deng et al., 2012). BN could be applied out-of-the-box on any feed-forward architecture, i.e. CNNs, MLPs and Auto-Encoders, and was thus quickly adopted by the computer vision community.

1. Equal contribution

However when working with sequential data, such as in speech recognition and natural language processing, researchers could not benefit from the advantages of BN: It was indeed unclear how to properly apply BN to RNNs, which is the type of neural networks that was typically used to process sequential data (e.g. Graves et al. (2013b); Bahdanau et al. (2015)).

3.3. Contributions

This work is the first step towards applying BN to RNNs. Although this article does not show great generalisation performance, it still highlights the fact that, while applying BN simultaneously on both input and recurrent connections is detrimental to the performance of the RNNs, only normalising the input connections already helped the optimisation procedure.

It also presents two possible ways of computing the statistics for the normalisation, and explains in what context one or the other should be used.

3.4. Recent Developments

Although producing somewhat mitigated generalisation performance on our benchmarks, the parameterisation presented in this article worked extremely well in the speech recognition system of Baidu, *Deep Speech 2*, which achieved performance levels competitive with human transcriptions on standard datasets of Mandarin and English (Amodei et al., 2016).

In Cooijmans et al. (2016), we built on this work to propose a better application of BN when the architecture of the RNN is a LSTM. That "BN-LSTM" is the version that most people were later using and building on. This article is presented in Chapter 6.

Ravanelli et al. (2018) used our parameterisation in their Gated Recurrent Units based models for speech recognition, and their implementation is available in the wildly-used *Pytorch-kaldi Speech Recognition Toolkit* (Ravanelli et al., 2019). Finally, Ledinauskas et al. (2020) showed that our parameterisation greatly improves the training of deep Spiking Neural Networks, a brain inspired variant of neural networks.

Chapter 4

First Article: Batch Normalised Recurrent Neural Networks

Abstract. Recurrent Neural Networks (RNNs) are powerful models for sequential data that have the potential to learn long-term dependencies. However, they are computationally expensive to train and difficult to parallelise. Recent work has shown that normalising intermediate representations of neural networks can significantly improve convergence rates in feed-forward neural networks (Ioffe & Szegedy, 2015). In particular, batch normalisation, which uses mini-batch statistics to standardise features, was shown to significantly reduce training time. In this paper, we show that applying batch normalisation to the hidden-to-hidden transitions of our RNNs doesn't help the training procedure. We also show that when applied to the input-to-hidden transitions, batch normalisation can lead to a faster convergence of the training criterion but doesn't seem to improve the generalisation performance on both our language modelling and speech recognition tasks. All in all, applying batch normalisation to RNNs turns out to be more challenging than applying it to feed-forward networks, but certain variants of it can still be beneficial.

4.1. Introduction

Recurrent Neural Networks (RNNs) have received renewed interest due to their recent success in various domains, including speech recognition (Graves et al., 2013b), machine translation (Sutskever et al., 2014; Bahdanau et al., 2015) and language modelling (Mikolov, 2012). The so-called Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) type RNN has been particularly successful. Often, it seems beneficial to train deep architectures in which multiple RNNs are stacked on top of each other (Graves et al., 2013b). Unfortunately, the training cost for large datasets and deep architectures of stacked RNNs can be prohibitively

high, often times an order of magnitude greater than simpler models like n-grams (Williams et al., 2015). Because of this, recent work has explored methods for parallelising RNNs across multiple graphics cards (GPUs). In (Sutskever et al., 2014), an LSTM type RNN was distributed layer-wise across multiple GPUs and in (Hannun et al., 2014) a bidirectional RNN was distributed across time. However, due to the sequential nature of RNNs, it is difficult to achieve linear speed ups relative to the number of GPUs.

Another way to reduce training times is through a better conditioned optimisation procedure. Standardising or whitening of input data has long been known to improve the convergence of gradient-based optimisation methods (LeCun et al., 2012). Extending this idea to multi-layered networks suggests that normalising or whitening intermediate representations can similarly improve convergence. However, applying these transforms would be extremely costly. In Ioffe & Szegedy (2015), batch normalisation was used to standardise intermediate representations by approximating the population statistics using sample-based approximations obtained from small subsets of the data, often called mini-batches, that are also used to obtain gradient approximations for stochastic gradient descent, the most commonly used optimisation method for neural network training. It has also been shown that convergence can be improved even more by whitening intermediate representations instead of simply standardising them (Desjardins et al., 2015). These methods reduced the training time of Convolutional Neural Networks (CNNs) by an order of magnitude and additionally provided a regularisation effect, leading to state-of-the-art results in object recognition on the ImageNet dataset (Russakovsky et al., 2015). In this paper, we explore how to leverage normalisation in RNNs and show that training time can be reduced.

4.2. Batch Normalisation

In optimisation, feature standardisation or whitening is a common procedure that has been shown to reduce convergence rates (LeCun et al., 2012). Extending the idea to deep neural networks, one can think of an arbitrary layer as receiving samples from a distribution that is shaped by the layer below. This distribution changes during the course of training, making any layer but the first responsible not only for learning a good representation but also for adapting to a changing input distribution. This distribution variation is termed *Internal Covariate Shift*, and reducing it is hypothesised to help the training procedure (Ioffe & Szegedy, 2015).

To reduce this internal covariate shift, we could whiten each layer of the network. However, this often turns out to be too computationally demanding. Batch normalisation (Ioffe & Szegedy, 2015) approximates the whitening by standardising the intermediate representations

using the statistics of the current mini-batch. Given a mini-batch $\mathbf{x}$, we can calculate the sample mean and sample variance of each feature k along the mini-batch axis

$$\bar{\mathbf{x}}_k = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_{i,k}, \tag{4.1}$$

$$\sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_{i,k} - \bar{\mathbf{x}}_k)^2, \tag{4.2}$$

where m is the size of the mini-batch. Using these statistics, we can standardise each feature as follows

$$\hat{\mathbf{x}}_k = \frac{\mathbf{x}_k - \bar{\mathbf{x}}_k}{\sqrt{\sigma_k^2 + \epsilon}}, \tag{4.3}$$

where ϵ is a small positive constant to improve numerical stability.

However, standardising the intermediate activations reduces the representational power of the layer. To account for this, batch normalisation introduces additional learnable parameters γ and β, which respectively scale and shift the data, leading to a layer of the form

$$BN(\mathbf{x}_k) = \gamma_k\hat{\mathbf{x}}_k + \beta_k. \tag{4.4}$$

By setting γ_k to σ_k and β_k to $\bar{x}_k$, the network can recover the original layer representation. So, for a standard feed-forward layer in a neural network

$$\mathbf{y} = \varphi(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{4.5}$$

where $\mathbf{W}$ is the weights matrix, $\mathbf{b}$ is the bias vector, $\mathbf{x}$ is the input of the layer and φ is an arbitrary activation function, batch normalisation is applied as follows

$$\mathbf{y} = \varphi(BN(\mathbf{W}\mathbf{x})). \tag{4.6}$$

Note that the bias vector has been removed, since its effect is cancelled by the standardisation. Since the normalisation is now part of the network, the back propagation procedure needs to be adapted to propagate gradients through the mean and variance computations as well.

At test time, we can't use the statistics of the mini-batch. Instead, we can estimate them by either forwarding several training mini-batches through the network and averaging their statistics, or by maintaining a running average calculated over each mini-batch seen during training.

4.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) extend Neural Networks to sequential data. Given an input sequence of vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$, they produce a sequence of hidden states $(\mathbf{h}_1, \ldots, \mathbf{h}_T)$,

which are computed at time step t as follows

$$\mathbf{h}_t = \varphi(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t), \tag{4.7}$$

where $\mathbf{W}_h$ is the recurrent weight matrix, $\mathbf{W}_x$ is the input-to-hidden weight matrix, and φ is an arbitrary activation function.

If we have access to the whole input sequence, we can use information not only from the past time steps, but also from the future ones, allowing for bidirectional RNNs (Schuster & Paliwal, 1997)

$$\overrightarrow{\mathbf{h}}_t = \varphi(\overrightarrow{\mathbf{W}}_h \overrightarrow{\mathbf{h}}_{t-1} + \overrightarrow{\mathbf{W}}_x \mathbf{x}_t), \tag{4.8}$$

$$\overleftarrow{\mathbf{h}}_t = \varphi(\overleftarrow{\mathbf{W}}_h \overleftarrow{\mathbf{h}}_{t+1} + \overleftarrow{\mathbf{W}}_x \mathbf{x}_t), \tag{4.9}$$

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t : \overleftarrow{\mathbf{h}}_t], \tag{4.10}$$

where $[\mathbf{x} : \mathbf{y}]$ denotes the concatenation of $\mathbf{x}$ and $\mathbf{y}$. Finally, we can stack RNNs by using $\mathbf{h}$ as the input to another RNN, creating deeper architectures (Pascanu et al., 2013a)

$$\mathbf{h}_t^l = \varphi(\mathbf{W}_h \mathbf{h}_{t-1}^l + \mathbf{W}_x \mathbf{h}_t^{l-1}). \tag{4.11}$$

In vanilla RNNs, the activation function φ is usually a sigmoid function, such as the hyperbolic tangent. Training such networks is known to be particularly difficult, because of vanishing and exploding gradients (Pascanu et al., 2013b).

4.3.1. Long Short-Term Memory

A commonly used recurrent structure is the Long Short-Term Memory (LSTM). It addresses the vanishing gradient problem commonly found in vanilla RNNs by incorporating gating functions into its state dynamics (Hochreiter & Schmidhuber, 1997). At each time step, an LSTM maintains a hidden vector $\mathbf{h}$ and a cell vector $\mathbf{c}$ responsible for controlling state updates and outputs. More concretely, we define the computation at time step t as follows (Gers et al., 2003):

$$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_{hi} \mathbf{h}_{t-1} + \mathbf{W}_{xi} \mathbf{x}_t) \tag{4.12}$$

$$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_{hf} \mathbf{h}_{t-1} + \mathbf{W}_{hf} \mathbf{x}_t) \tag{4.13}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{hc} \mathbf{h}_{t-1} + \mathbf{W}_{xc} \mathbf{x}_t) \tag{4.14}$$

$$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_{ho} \mathbf{h}_{t-1} + \mathbf{W}_{hx} \mathbf{x}_t + \mathbf{W}_{co} \mathbf{c}_t) \tag{4.15}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{4.16}$$

where sigmoid$(\cdot)$ is the logistic sigmoid function, tanh is the hyperbolic tangent function, $\mathbf{W}_{h\cdot}$ are the recurrent weight matrices and $\mathbf{W}_{x\cdot}$ are the input-to-hidden weight matrices. $\mathbf{i}_t$, $\mathbf{f}_t$ and $\mathbf{o}_t$ are respectively the input, forget and output gates, and $\mathbf{c}_t$ is the cell.

4.4. Batch Normalisation for RNNs

From equation 4.6, an analogous way to apply batch normalisation to an RNN would be as follows:

$$\mathbf{h}_t = \varphi(BN(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_x\mathbf{x}_t)). \tag{4.17}$$

However, in our experiments, when batch normalisation was applied in this fashion, it didn't help the training procedure (see appendix 4.A for more details). Instead we propose to apply batch normalisation only to the input-to-hidden transition ($\mathbf{W}_x\mathbf{x}_t$), i.e. as follows:

$$\mathbf{h}_t = \varphi(\mathbf{W}_h\mathbf{h}_{t-1} + BN(\mathbf{W}_x\mathbf{x}_t)). \tag{4.18}$$

This idea is similar to the way dropout (Srivastava et al., 2014) can be applied to RNNs (Zaremba et al., 2014): batch normalisation is applied only on the vertical connections (i.e. from one layer to another) and not on the horizontal connections (i.e. within the recurrent layer). We use the same principle for LSTMs: batch normalisation is only applied after multiplication with the input-to-hidden weight matrices $\mathbf{W}_{x\cdot}$.

4.4.1. Frame-wise and Sequence-wise Normalisation

In experiments where we don't have access to the future frames, like in language modelling where the goal is to predict the next character, we are forced to compute the normalisation a each time step

$$\hat{\mathbf{x}}_{k,t} = \frac{\mathbf{x}_{k,t} - \bar{\mathbf{x}}_{k,t}}{\sqrt{\sigma^2_{k,t} + \epsilon}}. \tag{4.19}$$

We'll refer to this as *frame-wise normalisation*.

In applications like speech recognition, we usually have access to the entire sequences. However, those sequences may have variable length. Usually, when using mini-batches, the smaller sequences are padded with zeroes to match the size of the longest sequence of the mini-batch. In such setups we can't use frame-wise normalisation, because the number of un-padded frames decreases along the time axis, leading to increasingly poorer statistics estimates. To solve this problem, we apply a sequence-wise normalisation, where we compute the mean and

variance of each feature along both the time and batch axis using

$$\bar{\mathbf{x}}_k = \frac{1}{n}\sum_{i=1}^{m}\sum_{t=1}^{T}\mathbf{x}_{i,t,k}, \tag{4.20}$$

$$\sigma_k^2 = \frac{1}{n}\sum_{i=1}^{m}\sum_{t=1}^{T}(\mathbf{x}_{i,t,k} - \bar{\mathbf{x}}_k)^2, \tag{4.21}$$

where T is the length of each sequence and n is the total number of un-padded frames in the mini-batch. We'll refer to this type of normalisation as *sequence-wise normalisation*.

4.5. Experiments

We ran experiments on a speech recognition task and a language modelling task. The models were implemented using Theano (Bastien et al., 2012) and Blocks (Van Merriënboer et al., 2015).

4.5.1. Speech Alignment Prediction

For the speech task, we used the Wall Street Journal (WSJ) (Paul & Baker, 1992) speech corpus. We used the si284 split as training set and evaluated our models on the dev93 development set. The raw audio was transformed into 40 dimensional log mel filter-banks (plus energy), with deltas and delta-deltas. As in Graves et al. (2013a), the forced alignments were generated from the Kaldi recipe tri4b, leading to 3546 clustered triphone states. Because of memory issues, we removed from the training set the sequences that were longer than 1300 frames (4.6% of the set), leading to a training set of 35746 sequences.

The baseline model (BL) is a stack of 5 bidirectional LSTM layers with 250 hidden units each, followed by a size 3546 softmax output layer. All the weights were initialised using the Glorot (Glorot & Bengio, 2010) scheme and all the biases were set to zero. For the batch normalised model (BN) we applied sequence-wise normalisation to each LSTM of the baseline model. Both networks were trained using standard SGD with momentum, with a fixed learning rate of 1e-4 and a fixed momentum factor of 0.9. The mini-batch size is 24.

4.5.2. Language Modelling

We used the Penn Treebank (PTB) (Marcus et al., 1993) corpus for our language modelling experiments. We use the standard split (929k training words, 73k validation words, and 82k

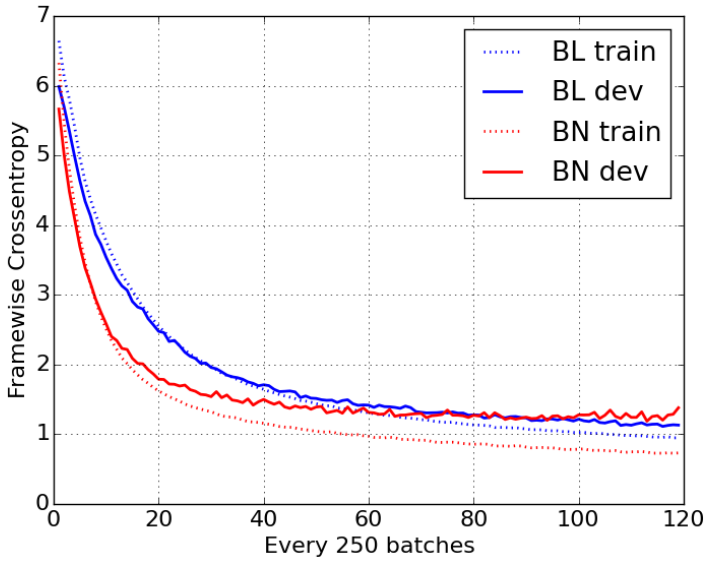


Figure 4.1. Frame-wise cross entropy on WSJ for the baseline (blue) and batch normalised (red) networks. The dotted lines are the training curves and the solid lines are the validation curves.

Table 4.1. Best frame-wise cross-entropy (FCE) and frame error rate (FER) on the training and development sets for both networks.

Model	**Train**		**Dev**	
	FCE	FER	FCE	FER
BiRNN	0.95	0.28	**1.11**	**0.33**
BiRNN (BN)	**0.73**	**0.22**	1.19	0.34

test words) and vocabulary of 10k words. We train a small, medium and large LSTM as described in Zaremba et al. (2014). All models consist of two stacked LSTM layers and are trained with stochastic gradient descent (SGD) with a learning rate of 1 and a mini-batch size of 32.

The small LSTM has two layers of 200 memory cells, with parameters being initialised from a uniform distribution with range [-0.1, 0.1]. We back propagate across 20 time steps and the gradients are scaled according to the maximum norm of the gradients whenever the norm is greater than 10. We train for 15 epochs and halve the learning rate every epoch after the 6th.

The medium LSTM has a hidden size of 650 for both layers, with parameters being initialised from a uniform distribution with range [-0.05, 0.05]. We apply dropout with probability of 50% between all layers. We back propagate across 35 time steps and gradients are scaled

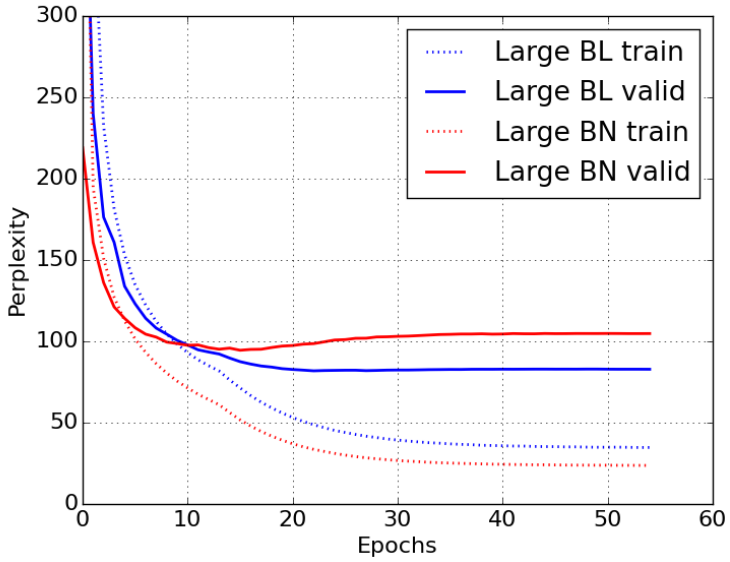


Figure 4.2. Large LSTM on Penn Treebank for the baseline (blue) and the batch normalised (red) networks. The dotted lines are the training curves and the solid lines are the validation curves.

Table 4.2. Best perplexity on training and development sets for LSTMs on Penn Treebank.

Model	**Train**	**Valid**
Small LSTM	78.5	119.2
Small LSTM (BN)	62.5	120.9
Medium LSTM	49.1	89.0
Medium LSTM (BN)	41.0	90.6
Large LSTM	49.3	**81.8**
Large LSTM (BN)	**35.0**	97.4

according to the maximum norm of the gradients whenever the norm is greater than 5. We train for 40 epochs and divide the learning rate by 1.2 every epoch after the 6th.

The Large LSTM has two layers of 1500 memory cells, with parameters being initialised from a uniform distribution with range [-0.04, 0.04]. We apply dropout between all layers. We back propagate across 35 time steps and gradients are scaled according to the maximum norm of the gradients whenever the norm is greater than 5. We train for 55 epochs and divide the learning rate by 1.15 every epoch after the 15th.

4.6. Results and Discussion

Figure 4.1 shows the training and development frame-wise cross-entropy curves for both networks of the speech experiments. As we can see, the batch normalised networks trains faster (at some points about twice as fast as the baseline), but over-fits more. The best results, reported in table 4.1, are comparable to the ones obtained in Graves et al. (2013a).

Figure 4.2 shows the training and validation perplexity for the large LSTM network of the language experiment. We can also observe that the training is faster when we apply batch normalisation to the network. However, it also over-fits more than the baseline version. The best results are reported in table 4.2.

For both experiments we observed a faster training and a greater overfitting when using our version of batch normalisation. This last effect is less prevalent in the speech experiment, perhaps because the training set is way bigger, or perhaps because the frame-wise normalisation is less effective than the sequence-wise one. It can also be caused by the experimental setup: in the language modelling task we predict one character at a time, whereas we predict the whole sequence in the speech experiment.

Batch normalisation also allows for higher learning rates in feed-forward networks, however since we only applied batch normalisation to parts of the network, higher learning rates didn't work well because they affected un-normalised parts as well.

Our experiments suggest that applying batch normalisation to the input-to-hidden connections in RNNs can improve the conditioning of the optimisation problem. Future directions include whitening input-to-hidden connections (Desjardins et al., 2015) and normalising the hidden state instead of just a portion of the network.

Acknowledgments

Part of this work was funded by Samsung. We also want to thank Nervana Systems for providing GPUs.

4.A. Experimentation with Normalisation Inside the Recurrence

In our first experiments we investigated if batch normalisation can be applied in the same way as in a feed-forward network (equation 4.17). We tried it on a language modelling task on the Penn Treebank dataset, where the goal was to predict the next characters of a fixed length sequence of 100 symbols.

The network is composed of a lookup table of dimension 250 followed by 3 layers of simple recurrent networks with 250 hidden units each. A dimension 50 softmax layer is added on the top. In the batch normalised networks, we apply batch normalisation to the hidden-to-hidden transition, as in equation 4.17, meaning that we compute one mean and one variance for each of the 251 features at each time step. For inference, we also keep track of the statistics for each time step. However, we used the same γ and β for each time step.

The lookup table is randomly initialised using an isotropic Gaussian with zero mean and unit variance. All the other matrices of the network are initialised using the Glorot scheme (Glorot & Bengio, 2010) and all the bias are set to zero. We used SGD with momentum. We performed a random search over the learning rate (distributed in the range [0.0001, 1]), the momentum (with possible values of 0.5, 0.8, 0.9, 0.95, 0.995), and the batch size (32, 64 or 128). We let the experiment run for 20 epochs. A total of 52 experiments were performed.

In every experiment that we ran, the performances of batch normalised networks were always slightly worse than (or at best equivalent to) the baseline networks, except for the ones where the learning rate is too high and the baseline diverges while the batch normalised one is still able to train. Figure 4.3 shows an example of a working experiment. We observed that in practically all the experiments that converged, the normalisation was actually harming the performance. Table 4.3 shows the results of the best baseline and batch normalised networks. We can observe that both best networks have similar performances. The settings for the best baseline are: learning rate 0.42, momentum 0.95, batch size 32. The settings for the best batch normalised network are: learning rate 3.71e-4, momentum 0.995, batch size 128.

Those results suggest that this way of applying batch normalisation in the recurrent networks is not optimal. It seems that batch normalisation hurts the training procedure. It may be due to the fact that we estimate new statistics at each time step, or because of the repeated application of γ and β during the recurrent procedure, which could lead to exploding or vanishing gradients. We will investigate more in depth what happens in the batch normalised networks, especially during the back-propagation.

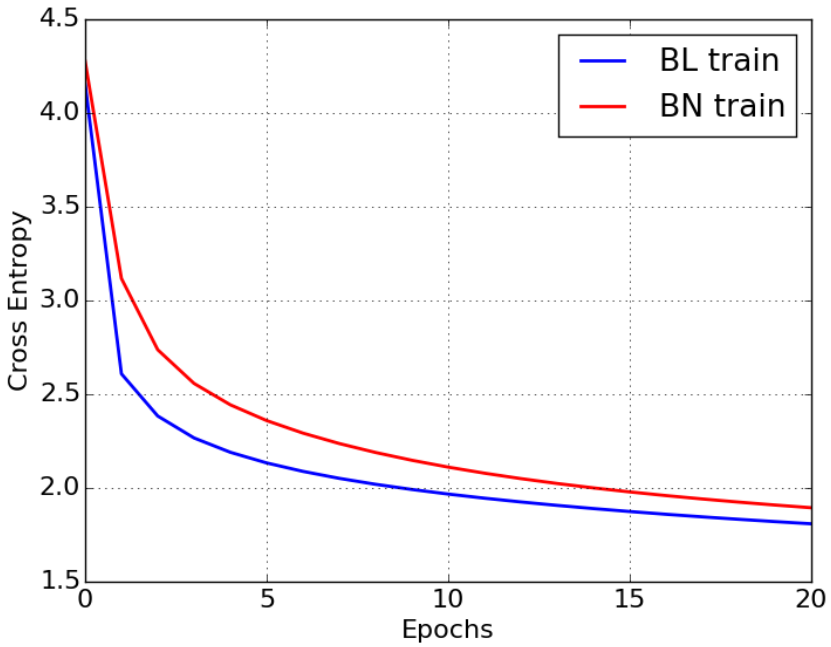


Figure 4.3. Typical training curves obtained during the grid search. The baseline network is in blue and batch normalised one in red. For this experiment, the hyper-parameters are: learning rate 7.8e-4, momentum 0.5, batch size 64.

Table 4.3. Best frame-wise cross-entropy for the best baseline network and for the best batch normalised one.

Model	**Train**	**Valid**
Best Baseline	1.05	1.10
Best Batch Norm	1.07	1.11

Chapter 5

Prologue to the Second Article

5.1. Article Details

Recurrent Batch Normalisation. Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre and Aaron Courville, In *Proceedings of the 5^{th} International Conference on Learning Representations (ICLR 2017).*

Note. Because we were not able to apply normalisation to all the parameters of the model, the previous article presented in this document left me with a feeling of unfinished business for which this next article is providing closure, since it presents how to successfully apply normalisation to all the parameters of the model. This is the reason why, although I am not the first author of the article, I nevertheless decided to include it in my thesis for completeness.

Authors contributions. Tim Cooijmans designed the proposed BN-LSTM parameterisation, and came up with the idea of initialising the γ parameters to ensure good gradient propagation. Nicolas Ballas provided the Theano implementation of the BN function and ran the MNIST experiments. I ran the lager scale *Teaching Machines to Read and Comprehend* experiment. By properly handling the padding of variable-length sequences in the bidirectional LSTMs, I was able to drastically improve the performance on that benchmark. I also ran speech recognition experiments, which did not make the cut in the final article. The *Teaching Machines to Read and Comprehend* experiment used a code base developed by Çağlar Gülçehre. Aaron Courville was the supervisor of the project.

5.2. Context

After the first article on BN for RNNs (Laurent et al., 2016) and despite Baidu's successful use of that implementation in their system (Amodei et al., 2016), there was still a need for better RNN parameterisations. The fact that the recurrent connections were not normalised, and thus preventing the use of higher learning rates, clearly indicated that there was room for improvement.

5.3. Contributions

This paper presents the BN-LSTM, an extension of the LSTM architecture that applies BN not only to the input connections, as in Laurent et al. (2016), but also to the recurrent connections, allowing for faster optimisation.

In particular, it highlights the importance of normalising both input and recurrent connections separately, which is something I had not investigated previously. Also, it shows that the γ parameter has a drastic impact on the back-propagation of the gradients, and that an initial value of 0.1, instead using 1.0 as in the original BN formulation (Ioffe & Szegedy, 2015), leads to better convergence of the training procedure.

5.4. Recent Developments

Since the LSTM was at the heart of almost every deep learning system processing sequences, the BN-LSTM has been used for various applications, including e.g. action recognition (Carreira & Zisserman, 2017), mobile sensing data processing (Yao et al., 2017) or sleep staging (Phan et al., 2019).

However, the main drawback of BN, and thus of the BN-LSTM as well, is having to keep track of estimates of population statistics for inference. To alleviate this issue, Ba et al. (2016) proposed Layer Normalisation (LN), a modification of BN which computes the statistics along the feature axis rather than the mini-batch axis, thus removing the need to track populations statistics. So, replacing BN with LN in the BN-LSTM greatly simplifies the implementation and usage of normalised LSTMs. LN became the *de facto* normalisation strategy for LSTMs, and is still widely used at the time of writing of this document (Dai et al., 2019; Yang et al., 2019; Sun et al., 2020). Yet, we might maybe see a comeback of BN in recurrent setups (Shen et al., 2020).

Chapter 6

Second Article: Recurrent Batch Normalisation

Abstract. We propose a reparameterisation of LSTM that brings the benefits of batch normalisation to recurrent neural networks. Whereas previous works only apply batch normalisation to the input-to-hidden transformation of RNNs, we demonstrate that it is both possible and beneficial to batch-normalise the hidden-to-hidden transition, thereby reducing internal covariate shift between time steps. We evaluate our proposal on various sequential problems such as sequence classification, language modelling and question answering. Our empirical results show that our batch-normalised LSTM consistently leads to faster convergence and improved generalisation.

6.1. Introduction

Recurrent neural network architectures such as LSTM (Hochreiter & Schmidhuber, 1997) and GRU (Cho et al., 2014) have recently exhibited state-of-the-art performance on a wide range of complex sequential problems including speech recognition (Amodei et al., 2016), machine translation (Bahdanau et al., 2015) and image and video captioning (Xu et al., 2015; Yao et al., 2015). Top-performing models, however, are based on very high-capacity networks that are computationally intensive and costly to train. Effective optimisation of recurrent neural networks is thus an active area of study (Pascanu et al., 2013b; Martens & Sutskever, 2011; Ollivier, 2013).

It is well-known that for deep feed-forward neural networks, covariate shift (Shimodaira, 2000; Ioffe & Szegedy, 2015) degrades the efficiency of training. Covariate shift is a change in the distribution of the inputs to a model. This occurs continuously during training of feed-forward

neural networks, where changing the parameters of a layer affects the distribution of the inputs to all layers above it. As a result, the upper layers are continually adapting to the shifting input distribution and unable to learn effectively. This *internal* covariate shift (Ioffe & Szegedy, 2015) may play an especially important role in recurrent neural networks, which resemble very deep feed-forward networks.

Batch normalisation (Ioffe & Szegedy, 2015) is a recently proposed technique for controlling the distributions of feed-forward neural network activations, thereby reducing internal covariate shift. It involves standardising the activations going into each layer, enforcing their means and variances to be invariant to changes in the parameters of the underlying layers. This effectively decouples each layer's parameters from those of other layers, leading to a better-conditioned optimisation problem. Indeed, deep neural networks trained with batch normalisation converge significantly faster and generalise better.

Although batch normalisation has demonstrated significant training speed-ups and generalisation benefits in feed-forward networks, it is proven to be difficult to apply in recurrent architectures (Laurent et al., 2016; Amodei et al., 2016). It has found limited use in stacked RNNs, where the normalisation is applied "vertically", i.e. to the input of each RNN, but not "horizontally" between timesteps. RNNs are deeper in the time direction, and as such batch normalisation would be most beneficial when applied horizontally. However, Laurent et al. (2016) hypothesised that applying batch normalisation in this way hurts training because of exploding gradients due to repeated rescaling.

Our findings run counter to this hypothesis. We show that it is both possible and highly beneficial to apply batch normalisation in the hidden-to-hidden transition of recurrent models. In particular, we describe a reparameterisation of LSTM (Section 6.3) that involves batch normalisation and demonstrate that it is easier to optimise and generalises better. In addition, we empirically analyse the gradient backpropagation and show that proper initialisation of the batch normalisation parameters is crucial to avoiding vanishing gradient (Section 6.4). We evaluate our proposal on several sequential problems and show (Section 6.5) that our LSTM reparameterisation consistently outperforms the LSTM baseline across tasks, in terms of both time to convergence and performance.

Liao & Poggio (2016) simultaneously investigated batch normalisation in recurrent neural networks, albeit only for very short sequences (10 steps). Ba et al. (2016) independently developed a variant of batch normalisation that is also applicable to recurrent neural networks and delivers similar improvements as our method.

6.2. Prerequisites

6.2.1. LSTM

Long Short-Term Memory (LSTM) networks are an instance of a more general class of recurrent neural networks (RNNs), which we review briefly in this paper. Given an input sequence $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$, an RNN defines a sequence of hidden states $\mathbf{h}_t$ according to

$$\mathbf{h}_t = \phi(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}), \tag{6.1}$$

where $\mathbf{W}_h \in \mathbb{R}^{d_h \times d_h}, \mathbf{W}_x \in \mathbb{R}^{d_x \times d_h}, \mathbf{b} \in \mathbb{R}^{d_h}$ and the initial state $\mathbf{h}_0 \in \mathbb{R}^{d_h}$ are model parameters. A popular choice for the activation function $\phi(\,\cdot\,)$ is tanh.

RNNs are popular in sequence modelling thanks to their natural ability to process variable-length sequences. However, training RNNs using first-order stochastic gradient descent (SGD) is notoriously difficult due to the well-known problem of exploding/vanishing gradients (Bengio et al., 1994; Hochreiter, 1991; Pascanu et al., 2013b). Gradient vanishing occurs when states $\mathbf{h}_t$ are not influenced by small changes in much earlier states $\mathbf{h}_\tau$, $t \ll \tau$, preventing learning of long-term dependencies in the input data. Although learning long-term dependencies is fundamentally difficult (Bengio et al., 1994), its effects can be mitigated through architectural variations such as LSTM (Hochreiter & Schmidhuber, 1997), GRU (Cho et al., 2014) and iRNN/uRNN (Le et al., 2015; Arjovsky et al., 2016).

In what follows, we focus on the LSTM architecture (Hochreiter & Schmidhuber, 1997) with recurrent transition given by

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b} \tag{6.2}$$

$$\mathbf{c}_t = \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t) \tag{6.3}$$

$$\mathbf{h}_t = \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathbf{c}_t), \tag{6.4}$$

where $\mathbf{W}_h \in \mathbb{R}^{d_h \times 4d_h}, \mathbf{W}_x \mathbb{R}^{d_x \times 4d_h}, \mathbf{b} \in \mathbb{R}^{4d_h}$ and the initial states $\mathbf{h}_0 \in \mathbb{R}^{d_h}, \mathbf{c}_0 \in \mathbb{R}^{d_h}$ are model parameters. σ is the logistic sigmoid function, and the $\odot$ operator denotes the Hadamard product.

The LSTM differs from simple RNNs in that it has an additional memory *cell* $\mathbf{c}_t$ whose update is nearly linear which allows the gradient to flow back through time more easily. In addition, unlike the RNN which overwrites its content at each time step, the update of the

LSTM cell is regulated by a set of gates. The forget gate $\mathbf{f}_t$ determines the extent to which information is carried over from the previous time step, and the input gate $\mathbf{i}_t$ controls the flow of information from the current input $\mathbf{x}_t$. The output gate $\mathbf{o}_t$ allows the model to read from the cell. This carefully controlled interaction with the cell is what allows the LSTM to robustly retain information for long periods of time.

6.2.2. Batch Normalisation

Covariate shift (Shimodaira, 2000) is a phenomenon in machine learning where the features presented to a model change in distribution. In order for learning to succeed in the presence of covariate shift, the model's parameters must be adjusted not just to learn the concept at hand but also to adapt to the changing distribution of the inputs. In deep neural networks, this problem manifests as *internal covariate shift* (Ioffe & Szegedy, 2015), where changing the parameters of a layer affects the distribution of the inputs to all layers above it.

Batch Normalisation (Ioffe & Szegedy, 2015) is a recently proposed network reparameterisation which aims to reduce internal covariate shift. It does so by standardising the activations using empirical estimates of their means and standard deviations. However, it does not decorrelate the activations due to the computationally costly matrix inversion. The batch normalising transform is as follows:

$$\mathrm{BN}(\mathbf{h}; \gamma, \beta) = \beta + \gamma \odot \frac{\mathbf{h} - \widehat{\mathbb{E}}[\mathbf{h}]}{\sqrt{\widehat{\mathrm{Var}}[\mathbf{h}] + \epsilon}} \tag{6.5}$$

where $\mathbf{h} \in \mathbb{R}^d$ is the vector of (pre)activations to be normalised, $\gamma \in \mathbb{R}^d, \beta \in \mathbb{R}^d$ are model parameters that determine the mean and standard deviation of the normalised activation, and $\epsilon \in \mathbb{R}$ is a regularisation hyper-parameter. The division should be understood to proceed element-wise.

At training time, the statistics $\mathbb{E}[\mathbf{h}]$ and $\mathrm{Var}[\mathbf{h}]$ are estimated by the sample mean and sample variance of the current mini-batch. This allows for backpropagation through the statistics, preserving the convergence properties of stochastic gradient descent. During inference, the statistics are typically estimated based on the entire training set, so as to produce a deterministic prediction.

6.3. Batch-Normalised LSTM

This section introduces a reparameterisation of LSTM that takes advantage of batch normalisation. Contrary to Laurent et al. (2016) and Amodei et al. (2016), we leverage batch normalisation in both the input-to-hidden *and* the hidden-to-hidden transformations. We introduce the batch-normalising transform $\mathrm{BN}(\,\cdot\,;\gamma,\beta)$ into the LSTM as follows:

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathrm{BN}(\mathbf{W}_h\mathbf{h}_{t-1};\gamma_h,\beta_h) + \mathrm{BN}(\mathbf{W}_x\mathbf{x}_t;\gamma_x,\beta_x) + \mathbf{b} \tag{6.6}$$

$$\mathbf{c}_t = \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t) \tag{6.7}$$

$$\mathbf{h}_t = \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathrm{BN}(\mathbf{c}_t;\gamma_c,\beta_c)) \tag{6.8}$$

In our formulation, we normalise the recurrent term $\mathbf{W}_h\mathbf{h}_{t-1}$ and the input term $\mathbf{W}_x\mathbf{x}_t$ separately. Normalising these terms individually gives the model better control over the relative contribution of the terms using the γ_h and γ_x parameters. We set $\beta_h = \beta_x = \mathbf{0}$ to avoid unnecessary redundancy, instead relying on the pre-existing parameter vector $\mathbf{b}$ to account for both biases. In order to leave the LSTM dynamics intact and preserve the gradient flow through $\mathbf{c}_t$, we do not apply batch normalisation in the cell update.

The batch normalisation transform relies on batch statistics to standardise the LSTM activations. It would seem natural to share the statistics that are used for normalisation across time, just as recurrent neural networks share their parameters over time. However, we find that simply averaging statistics over time severely degrades performance. Although LSTM activations do converge to a stationary distribution, we observe that their statistics during the initial transient differ significantly (see Figure 6.5 in Appendix 6.A). Consequently, we recommend using separate statistics for each time step to preserve information of the initial transient phase in the activations. [1]

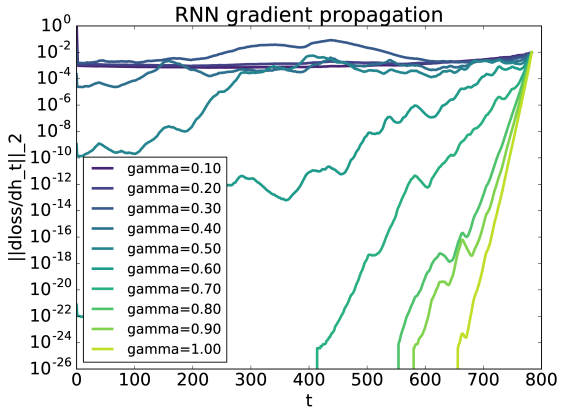
Generalising the model to sequences longer than those seen during training is straightforward thanks to the rapid convergence of the activations to their steady-state distributions (cf. Figure 6.5). For our experiments we estimate the population statistics separately for each time step $1,\ldots,T_{max}$ where T_{max} is the length of the longest training sequence. When at test time we need to generalise beyond T_{max}, we use the population statistic of time T_{max} for all time steps beyond it.

1. Note that we separate *only* the statistics over time and not the γ and β parameters.

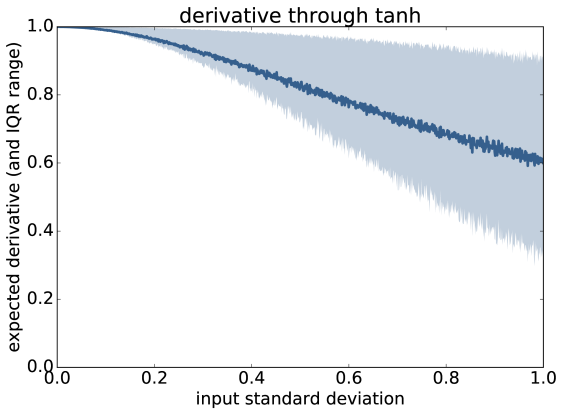
During training we estimate the statistics across the mini-batch, independently for each time step. At test time we use estimates obtained by averaging the mini-batch estimates over the training set.

6.4. Initialising γ for Gradient Flow

Although batch normalisation allows for easy control of the pre-activation variance through the γ parameters, common practice is to normalise to unit variance. We suspect that the previous difficulties with recurrent batch normalisation reported in Laurent et al. (2016) and in Amodei et al. (2016) are largely due to improper initialisation of the batch normalisation parameters, and γ in particular. In this section we demonstrate the impact of γ on gradient flow.



(a) We visualize the gradient flow through a batch-normalised tanh RNN as a function of γ. High variance causes vanishing gradient.



(b) We show the empirical expected derivative and interquartile range of tanh nonlinearity as a function of input variance. High variance causes saturation, which decreases the expected derivative.

Figure 6.1. Influence of pre-activation variance on gradient propagation.

In Figure 6.1a, we show how the pre-activation variance impacts gradient propagation in a simple RNN on the sequential MNIST task described in Section 6.5.1. Since backpropagation operates in reverse, the plot is best read from right to left. The quantity plotted is the norm of the gradient of the loss with respect to the hidden state at different time steps. For large values of γ, the norm quickly goes to zero as gradient is propagated back in time. For small values of γ the norm is nearly constant.

To demonstrate what we think is the cause of this vanishing, we drew samples x from a set of centred Gaussian distributions with standard deviation ranging from 0 to 1, and computed the derivative $\tanh'(x) = 1 - \tanh^2(x) \in [0, 1]$ for each. Figure 6.1b shows the empirical

distribution of the derivative as a function of standard deviation. When the input standard deviation is low, the input tends to be close to the origin where the derivative is close to 1. As the standard deviation increases, the expected derivative decreases as the input is more likely to be in the saturation regime. At unit standard deviation, the expected derivative is much smaller than 1.

We conjecture that this is what causes the gradient to vanish, and recommend initialising γ to a small value. In our trials we found that values of 0.01 or lower caused instabilities during training. Our choice of 0.1 seems to work well across different tasks.

6.5. Experiments

This section presents an empirical evaluation of the proposed batch-normalised LSTM on four different tasks. Note that for all the experiments, we initialise the batch normalisation scale and shift parameters γ and β to 0.1 and 0 respectively.

6.5.1. Sequential MNIST

We evaluate our batch-normalised LSTM on a sequential version of the MNIST classification task (Le et al., 2015). The model processes each image one pixel at a time and finally predicts the label. We consider both sequential MNIST tasks, MNIST and permuted MNIST (pMNIST). In MNIST, the pixels are processed in scan line order. In pMNIST the pixels are processed in a fixed random order.

Our baseline consists of an LSTM with 100 hidden units, with a softmax classifier to produce a prediction from the final hidden state. We use orthogonal initialisation for all weight matrices, except for the hidden-to-hidden weight matrix which we initialise to be the identity matrix, as this yields better generalisation performance on this task for both models. The model is trained using RMSProp (Tieleman & Hinton, 2012) with learning rate of 10^{-3} and 0.9 momentum. We apply gradient clipping at 1 to avoid exploding gradients.

The in-order MNIST task poses a unique problem for our model: the input for the first hundred or so time steps is constant across examples since the upper pixels are almost always black. This causes the variance of the hidden states to be exactly zero for a long period of time. Normalising these zero-variance activations involves dividing zero by a small number at many time steps, which does not affect the forward-propagated activations but causes the back-propagated gradient to explode. We work around this by adding Gaussian noise to

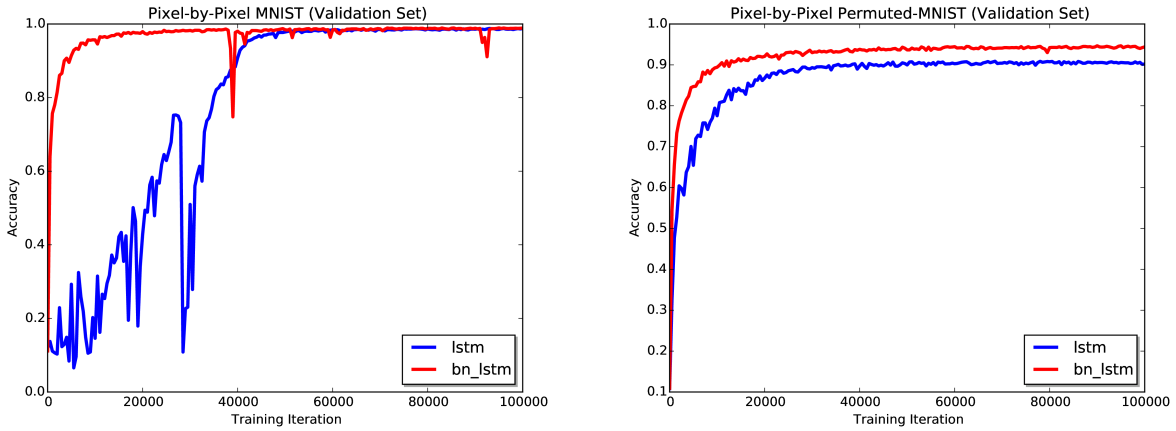


Figure 6.2. Accuracy on the validation set for the pixel by pixel MNIST classification tasks. The batch-normalised LSTM is able to converge faster relatively to a baseline LSTM. Batch-normalised LSTM also shows some improve generalisation on the permuted sequential MNIST that require to preserve long-term memory information.

the initial hidden states. Although the normalisation amplifies the noise to signal level, we find that it does not hurt performance compared to data-dependent ways of initialising the hidden states.

Table 6.1. Accuracy obtained on the test set for the pixel by pixel MNIST classification tasks

Model	**MNIST**	***p*MNIST**
TANH-RNN (Le et al., 2015)	35.0	35.0
*i*RNN (Le et al., 2015)	97.0	82.0
*u*RNN (Arjovsky et al., 2016)	95.1	91.4
*s*TANH-RNN (Zhang et al., 2016)	98.1	94.0
LSTM (ours)	98.9	90.2
BN-LSTM (ours)	**99.0**	**95.4**

In Figure 6.2 we show the validation accuracy while training for both LSTM and batch-normalised LSTM (BN-LSTM). BN-LSTM converges faster than LSTM on both tasks. Additionally, we observe that BN-LSTM generalises significantly better on *p*MNIST. It has been highlighted in Arjovsky et al. (2016) that *p*MNIST contains many longer term dependencies across pixels than in the original pixel ordering, where a lot of structure is local. A recurrent network therefore needs to characterise dependencies across varying time scales in order to solve this task. Our results suggest that BN-LSTM is better able to capture these long-term dependencies.

Table 6.1 reports the test set accuracy of the early stop model for LSTM and BN-LSTM using the population statistics. Recurrent batch normalisation leads to a better test score, especially for pMNIST where models have to leverage long-term temporal dependencies. In addition, Table 6.1 shows that our batch-normalised LSTM achieves state of the art on both MNIST and pMNIST.

6.5.2. Character-level Penn Treebank

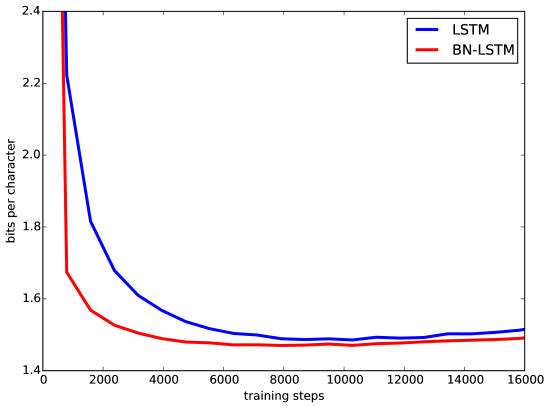
We evaluate our model on the task of character-level language modelling on the Penn Treebank corpus (Marcus et al., 1993) according to the train/valid/test partition of Mikolov et al. (2012). For training, we segment the training sequence into examples of length 100. The training sequence does not cleanly divide by 100, so for each epoch we randomly crop a sub-sequence that does and segment that instead.

Our baseline is an LSTM with 1000 units, trained to predict the next character using a softmax classifier on the hidden state $\mathbf{h}_t$. We use stochastic gradient descent on mini-batches of size 64, with gradient clipping at 1.0 and step rule determined by Adam (Kingma & Ba, 2015) with learning rate 0.002. We use orthogonal initialisation for all weight matrices. The setup for the batch-normalised LSTM is the same in all respects except for the introduction of batch normalisation as detailed in 6.3.

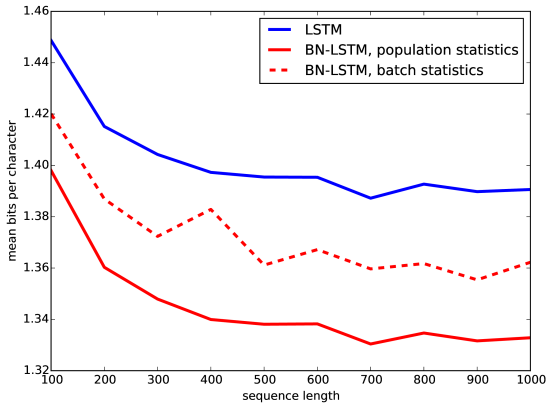
We show the learning curves in Figure 6.3a. BN-LSTM converges faster and generalises better than the LSTM baseline. Figure 6.3b shows the generalisation of our model to longer sequences. We observe that using the population statistics improves generalisation performance, which confirms that repeating the last population statistic (cf. Section 6.3) is a viable strategy. In table 6.2 we report the performance of our best models (early-stopped on validation performance) on the Penn Treebank test sequence. Follow up works havd since improved the state of the art (Krueger et al., 2016; Chung et al., 2016; Ha et al., 2016).

6.5.3. Text8

We evaluate our model on a second character-level language modelling task on the much larger text8 dataset (Mahoney, 2009). This dataset is derived from Wikipedia and consists of a sequence of 100M characters including only alphabetical characters and spaces. We follow Mikolov et al. (2012); Zhang et al. (2016) and use the first 90M characters for training, the next 5M for validation and the final 5M characters for testing. We train on non-overlapping sequences of length 180.



(a) Performance in bits-per-character on length-100 sub-sequences of the Penn Treebank validation sequence during training.



(b) Generalisation to longer sub-sequences of Penn Treebank using population statistics. The sub-sequences are taken from the test sequence.

Figure 6.3. Penn Treebank evaluation

Both our baseline and batch-normalised models are LSTMs with 2000 units, trained to predict the next character using a softmax classifier on the hidden state $\mathbf{h}_t$. We use stochastic gradient descent on mini-batches of size 128, with gradient clipping at 1.0 and step rule determined by Adam (Kingma & Ba, 2015) with learning rate 0.001. All weight matrices were initialised to be orthogonal.

We early-stop on validation performance and report the test performance of the resulting model in table 6.3. We observe that BN-LSTM obtains a significant performance improvement over the LSTM baseline. Chung et al. (2016) has since improved on our performance.

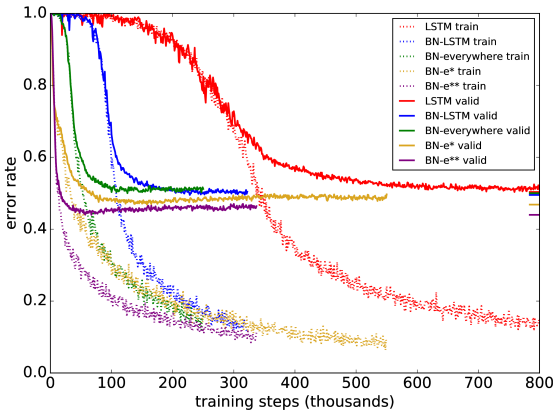
Table 6.2. Bits-per-character on the Penn Treebank test sequence.

Model	**Penn Treebank**
LSTM (Graves, 2013)	1.26 [2]
HF-MRNN (Mikolov et al., 2012)	1.41
Norm-stabilised LSTM (Krueger & Memisevic, 2015)	1.39
ME n-gram (Mikolov et al., 2012)	1.37
LSTM (ours)	1.38
BN-LSTM (ours)	1.32
Zoneout (Krueger et al., 2016)	1.27
HM-LSTM (Chung et al., 2016)	1.24
HyperNetworks (Ha et al., 2016)	**1.22**

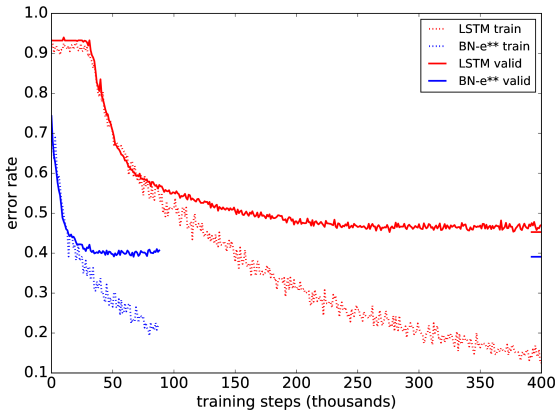
6.5.4. Teaching Machines to Read and Comprehend

Recently, Hermann et al. (2015) introduced a set of challenging benchmarks for natural language processing, along with neural network architectures to address them. The tasks involve reading real news articles and answering questions about their content. Their principal model, the Attentive Reader, is a recurrent neural network that invokes an attention mechanism to locate relevant information in the document. Such models are notoriously hard to optimise and yet increasingly popular.

To demonstrate the generality and practical applicability of our proposal, we apply batch normalisation in the Attentive Reader model and show that this drastically improves training.



(a) Error rate on the validation set for the Attentive Reader models on a variant of the CNN QA task (Hermann et al., 2015). As detailed in Appendix 6.C, the theoretical lower bound on the error rate on this task is 43%.



(b) Error rate on the validation set on the full CNN QA task from Hermann et al. (2015).

Figure 6.4. Training curves on the CNN question-answering tasks.

We evaluate several variants. The first variant, referred to as BN-LSTM, consists of the vanilla Attentive Reader model with the LSTM simply replaced by our BN-LSTM reparameterisation.

Table 6.3. Bits-per-character on the text8 test sequence.

Model	**text8**
td-LSTM (Zhang et al., 2016)	1.63
HF-MRNN (Mikolov et al., 2012)	1.54
skipping RNN (Pachitariu & Sahani, 2013)	1.48
LSTM (ours)	1.43
BN-LSTM (ours)	1.36
HM-LSTM (Chung et al., 2016)	**1.29**

The second variant, termed BN-everywhere, is exactly like the first, except that we also introduce batch normalisation into the attention computations, normalising each term going into the tanh non-linearities.

Our third variant, BN-e*, is like BN-everywhere, but improved to more carefully handle variable-length sequences. Throughout this experiment we followed the common practice of padding each batch of variable-length data with zeros. However, this biases the batch mean and variance of $\mathbf{x}_t$ toward zero. We address this effect using *sequence-wise* normalisation of the inputs as proposed by Laurent et al. (2016) and Amodei et al. (2016). That is, we share statistics over time for normalisation of the input terms $\mathbf{W}_x\mathbf{x}_t$, but *not* for the recurrent terms $\mathbf{W}_h\mathbf{h}_t$ or the cell output $\mathbf{c}_t$. Doing so avoids many issues involving degenerate statistics due to input sequence padding.

Our fourth and final variant BN-e** is like BN-e* but bidirectional. The main difficulty in adapting to bidirectional models also involves padding. Padding poses no problem as long as it is properly ignored (by not updating the hidden states based on padded regions of the input). However to perform the reverse application of a bidirectional model, it is common to simply reverse the padded sequences, thus moving the padding to the front. This causes similar problems as were observed on the sequential MNIST task (Section 6.5.1): the hidden states will not diverge during the initial time steps and hence their variance will be severely underestimated. To get around this, we reverse only the un-padded portion of the input sequences and leave the padding in place. See Appendix 6.C for hyper-parameters and task details.

Figure 6.4a shows the learning curves for the different variants of the attentive reader. BN-LSTM trains dramatically faster than the LSTM baseline. BN-everywhere in turn shows a significant improvement over BN-LSTM. In addition, both BN-LSTM and BN-everywhere show a generalisation benefit over the baseline. The validation curves have minima of 50.3%, 49.5% and 50.0% for the baseline, BN-LSTM and BN-everywhere respectively. We emphasise that these results were obtained without any tweaking – all we did was to introduce batch normalisation. BN-e* and BN-e** converge faster yet, and reach lower minima: 47.1% and 43.9% respectively.

Table 6.4. Error rates on the CNN question-answering task (Hermann et al., 2015).

Model	**CNN valid**	**CNN test**
Attentive Reader (Hermann et al., 2015)	38.4	37.0
LSTM (ours)	45.5	45.0
BN-e** (ours)	**37.9**	**36.3**

We train and evaluate our best model, BN-e**, on the full task from (Hermann et al., 2015). On this dataset we had to reduce the number of hidden units to 120 to avoid severe overfitting. Training curves for BN-e** and a vanilla LSTM are shown in Figure 6.4b. Table 6.4 reports performances of the early-stopped models.

6.6. Conclusion

Contrary to previous findings by Laurent et al. (2016) and Amodei et al. (2016), we have demonstrated that batch-normalising the hidden states of recurrent neural networks greatly improves optimisation. Indeed, doing so yields benefits similar to those of batch normalisation in feed-forward neural networks: our proposed BN-LSTM trains faster and generalises better on a variety of tasks including language modelling and question-answering. We have argued that proper initialisation of the batch normalisation parameters is crucial, and suggest that previous difficulties (Laurent et al., 2016; Amodei et al., 2016) were due in large part to improper initialisation. Finally, we have shown our model to apply to complex settings involving variable-length data, bidirectionallity and highly nonlinear attention mechanisms.

Acknowledgements

The authors would like to acknowledge the following agencies for research funding and computing support: the Nuance Foundation, Samsung, NSERC, Calcul Québec, Compute Canada, the Canada Research Chairs and CIFAR. Experiments were carried out using the Theano (Team et al., 2016) and the Blocks and Fuel (Van Merriënboer et al., 2015) libraries for scientific computing. We thank David Krueger, Saizheng Zhang, Ishmael Belghazi and Yoshua Bengio for discussions and suggestions.

6.A. Convergence of population statistics

Figure 6.5 shows how the population statistics converge to stationary distribution on the Penn Treebank task.

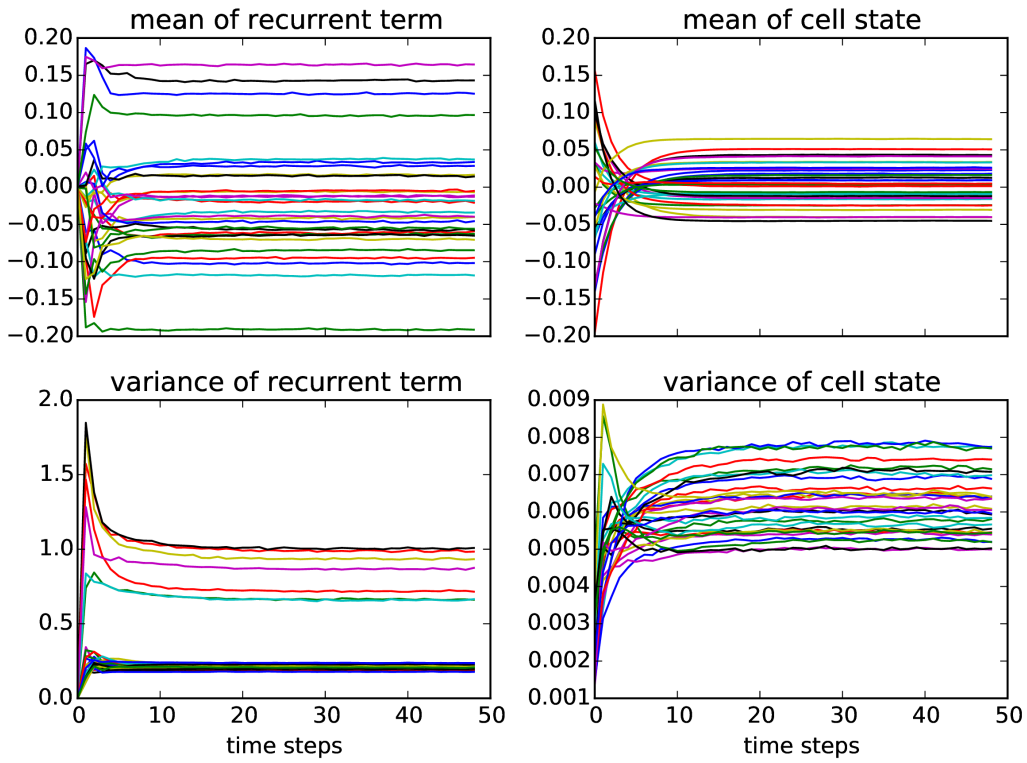


Figure 6.5. Convergence of population statistics to stationary distributions on the Penn Treebank task. The horizontal axis denotes RNN time. Each curve corresponds to a single hidden unit. Only a random subset of units is shown. See Section 6.3 for discussion.

6.B. Sensitivity to initialisation of γ

In Section 6.4 we investigated the effect of initial γ on gradient flow. To show the practical implications of this, we performed several experiments on the *p*MNIST and Penn Treebank benchmarks. The resulting performances are shown in Figure 6.6.

The *p*MNIST training curves confirm that higher initial values of γ are detrimental to the optimisation of the model. For the Penn Treebank task however, the effect is gone.

We believe this is explained by the difference in the nature of the two tasks. For *p*MNIST, the model absorbs the input sequence and only at the end of the sequence does it make a prediction on which it receives feedback. Learning from this feedback requires propagating the gradient all the way back through the sequence.

In the Penn Treebank task on the other hand, the model makes a prediction at each time step. At each step of the backward pass, a fresh learning signal is added to the backpropagated gradient. Essentially, the model is able to get off the ground by picking up short-term

dependencies. This fails on pMNIST wich is dominated by long-term dependencies (Arjovsky et al., 2016).

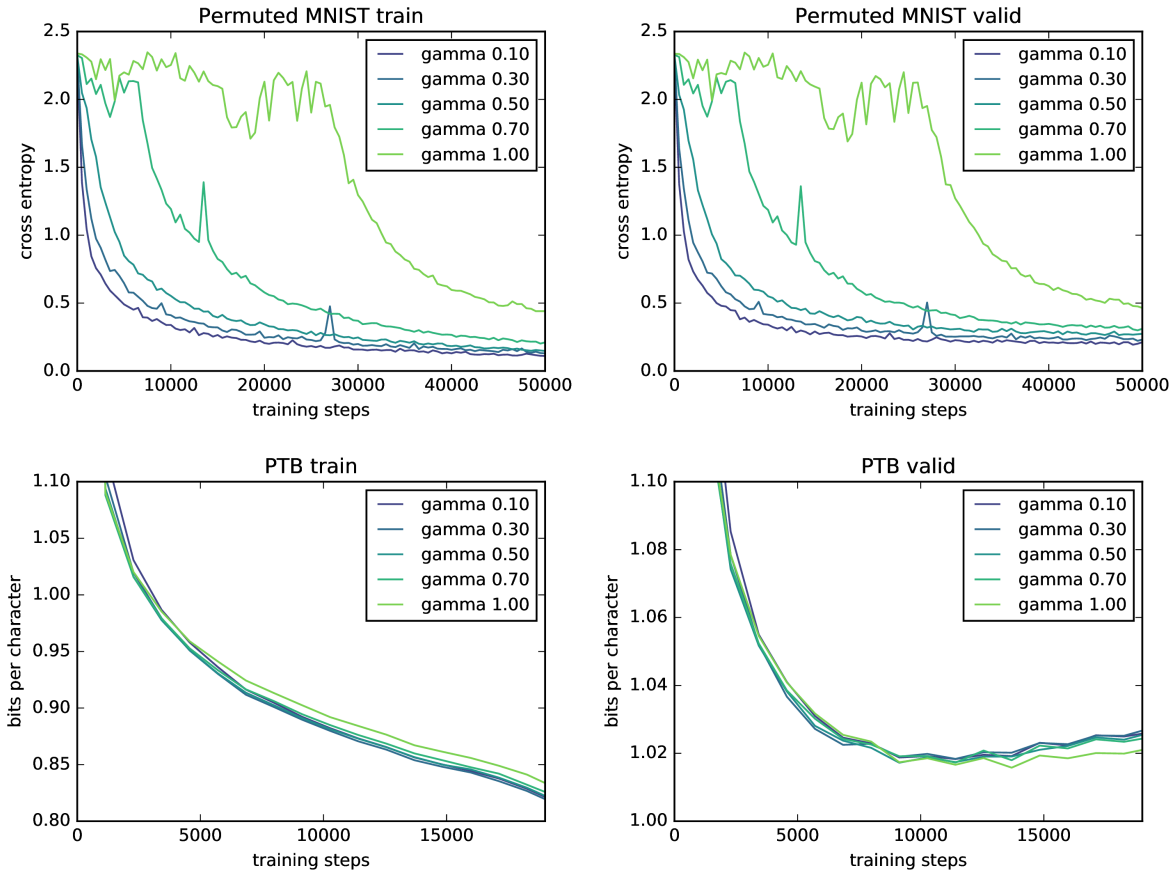


Figure 6.6. Training curves on pMNIST and Penn Treebank for various initialisations of γ.

6.C. Teaching Machines to Read and Comprehend: Task setup

We evaluate the models on the question answering task using the CNN corpus (Hermann et al., 2015), with placeholders for the named entities. We follow a similar preprocessing pipeline as Hermann et al. (2015). During training, we randomly sample the examples with replacement and shuffle the order of the placeholders in each text inside the mini-batch. We use a vocabulary of 65829 words.

We deviate from Hermann et al. (2015) in order to save computation: we use only the 4 most relevant sentences from the description, as identified by a string matching procedure. Both the training and validation sets are preprocessed in this way. Due to imprecision this

heuristic sometimes strips the answers from the passage, putting an upper bound of 57% on the validation accuracy that can be achieved.

For the reported performances, the first three models (LSTM, BN-LSTM and BN-everywhere) are trained using the exact same hyper-parameters, which were chosen because they work well for the baseline. The hidden state is composed of 240 units. We use stochastic gradient descent on mini-batches of size 64, with gradient clipping at 10 and step rule determined by Adam (Kingma & Ba, 2015) with learning rate 8×10^{-5}.

For BN-e* and BN-e**, we use the same hyper-parameters except that we reduce the learning rate to 8×10^{-4} and the mini-batch size to 40.

6.D. Hyper-parameter Searches

Table 6.5 reports hyper-parameter values that were tried in the experiments. For MNIST and *p*MNIST, the hyper-parameters were varied independently. For Penn Treebank, we performed a full grid search on learning rate and hidden state size, and later performed a sensitivity analysis on the batch size and initial γ. For the text8 task and the experiments with the Attentive Reader, we carried out a grid search on the learning rate and hidden state size.

The same values were tried for both the baseline and our BN-LSTM. In each case, our reported results are those of the model with the best validation performance.

Table 6.5. Hyper-parameter values that have been explored in the experiments.

MNIST and pMNIST	
Learning rate:	1e-2, 1e-3, 1e-4
RMSProp momentum:	0.5, 0.9
Hidden state size:	100, 200, 400
Initial γ:	1e-1, 3e-1, 5e-1, 7e-1, 1.0
Penn Treebank	
Learning rate:	1e-1, 1e-2, 2e-2, 1e-3
Hidden state size:	800, 1000, 1200, 1500, 2000
Batch size:	32, 64, 100, 128
Initial γ:	1e-1, 3e-1, 5e-1, 7e-1, 1.0
Text8	
Learning rate:	1e-1, 1e-2, 1e-3
Hidden state size:	500, 1000, 2000, 4000
Attentive Reader	
Learning rate:	8e-3, 8e-4, 8e-5, 8e-6
Hidden state size:	60, 120, 240, 280

Chapter 7

Prologue to the Third Article

7.1. Article Details

Fast Approximate Natural Gradient Descent in a Kronecker-factored Eigenbasis. Thomas George [1], César Laurent[1], Xavier Bouthillier, Nicolas Ballas and Pascal Vincent, In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018).*

Authors contributions. The EKFAC method presented in this paper is the result of an extensive work in terms of development and implementation, of which I am responsible for a large part. I also ran all the K-FAC baselines. Thomas George had the original idea for the EKFAC method. Xavier Bouthillier helped us by doing punctual analysis and Nicolas Ballas helped us scale our experiments on CIFAR10. Pascal Vincent was the supervisor of the project, is the author of the different theorems in the article, and also contributed to small parts of the code.

7.2. Context

After working on BN, which standardises the features, I wanted to move to the next natural step: decorrelating the features. Indeed, there was already evidence that decorrelation of the input features could improve neural networks training (Krizhevsky & Hinton, 2009), so it made sense to try to apply such transformation at every layer of the network.

1. Equal contribution

It turned out that such idea was already explored in Natural Neural Networks (Desjardins et al., 2015), and was strongly related to natural gradient under the K-FAC approximation (Heskes, 2000; Martens & Grosse, 2015).

7.3. Contributions

To lower the computational cost of K-FAC, Martens & Grosse (2015) propose to amortise the computation of the factors by re-estimating them every few updates, assuming the geometry of the loss landscape evolves relatively slowly along the gradient steps taken during optimisation. Instead of taking approximate natural gradient steps, we proposed to use eigenbasis of the factors to project the gradients to and from an approximate Kronecker-factored Eigenbasis (KFE). These projections allow the use of traditional diagonal methods such as RMSProp, in a basis where their inherent diagonal approximation is likely to be more accurate.

7.4. Recent Developments

Although the EKFAC optimiser itself has not been used much in practice, Bae et al. (2018) proposed to leverage the KFE to improve noisy natural gradient, an algorithm used to train variational Bayesian neural networks. Lee & Triebel (2019) used the KFE to improve the Laplace Approximation used for modelling the uncertainty in the predictions of neural networks. Finally, Wang et al. (2019) leveraged the KFE in their criterion for structured pruning.

Chapter 8

Third Article: Fast Approximate Natural Gradient Descent in a Kronecker-factored Eigenbasis

Abstract. Optimisation algorithms that leverage gradient covariance information, such as variants of natural gradient descent (Amari, 1998), offer the prospect of yielding more effective descent directions. For models with many parameters, the covariance matrix they are based on becomes gigantic, making them inapplicable in their original form. This has motivated research into both simple diagonal approximations and more sophisticated factored approximations such as K-FAC (Heskes, 2000; Martens & Grosse, 2015; Grosse & Martens, 2016). In the present work we draw inspiration from both to propose a novel approximation that is provably better than K-FAC and amendable to cheap partial updates. It consists in tracking a diagonal variance, not in parameter coordinates, but in a Kronecker-factored eigenbasis, in which the diagonal approximation is likely to be more effective. Experiments show improvements over K-FAC in optimisation speed for several deep network architectures.

8.1. Introduction

Deep networks have exhibited state-of-the-art performance in many application areas, including image recognition (He et al., 2016a) and natural language processing (Gehring et al., 2017). However top-performing systems often require days of training time and a large amount of computational power, so there is a need for efficient training methods.

Stochastic Gradient Descent (SGD) and its variants are the current workhorse for training neural networks. Training consists in optimising the network parameters θ (of size n_θ) to

minimise a regularised empirical risk $R(\theta)$, through gradient descent. The negative loss gradient is approximated based on a small subset of training examples (a mini-batch). The loss functions of neural networks are highly non-convex functions of the parameters, and the loss surface is known to have highly imbalanced curvature which limits the efficiency of 1^{st} optimisation methods such as SGD.

Methods that employ 2^{nd} order information have the potential to speed up 1^{st} order gradient descent by correcting for imbalanced curvature. The parameters are then updated as: $\theta \leftarrow \theta - \eta G^{-1} \nabla_\theta R(\theta)$, where η is a positive learning-rate and G is a preconditioning matrix capturing the local curvature or related information such as the Hessian matrix in Newton's method or the Fisher Information Matrix in Natural Gradient (Amari, 1998). Matrix G has a gigantic size $n_\theta \times n_\theta$ which makes it too large to compute and invert in the context of modern deep neural networks with millions of parameters. For practical applications, it is necessary to trade-off quality of curvature information for efficiency.

A long family of algorithms used for optimising neural networks can be viewed as approximating the diagonal of a large preconditioning matrix. Diagonal approximate of the Hessian (Becker et al., 1988) are proven to be efficient, and algorithms that use the diagonal of the covariance matrix of the gradients are widely used among neural networks practitioners, such as Adagrad (Duchi et al., 2011), Adadelta (Zeiler, 2012), RMSProp (Tieleman & Hinton, 2012), Adam (Kingma & Ba, 2015). We refer the reader to Bottou et al. (2018) for an informative review of optimisation methods for deep networks, including diagonal rescalings, and connections with the Batch Normalisation (BN) (Ioffe & Szegedy, 2015) technique.

More elaborate algorithms do not restrict to diagonal approximations, but instead aim at accounting for some correlations between different parameters (as encoded by non-diagonal elements of the preconditioning matrix). These methods range from Ollivier (2015) who introduces a rank 1 update that accounts for the cross correlations between the biases and the weight matrices, to quasi Newton methods (Liu & Nocedal, 1989) that build a running estimate of the exact non-diagonal preconditioning matrix, and also include block diagonals approaches with blocks corresponding to entire layers (Heskes, 2000; Desjardins et al., 2015; Martens & Grosse, 2015; Fujimoto & Ohira, 2018). Factored approximations such as K-FAC (Martens & Grosse, 2015; Ba et al., 2017) approximate each block as a Kronecker product of two much smaller matrices, both of which can be estimated and inverted more efficiently than the full block matrix, since the inverse of a Kronecker product of two matrices is the Kronecker product of their inverses.

In the present work, we draw inspiration from both diagonal and factored approximations. We introduce an Eigenvalue-corrected Kronecker Factorisation (EKFAC) that consists in tracking

a diagonal variance, not in parameter coordinates, but in a Kronecker-factored eigenbasis. We show that EKFAC is a provably better approximation of the Fisher Information Matrix than K-FAC. In addition, while computing Kronecker-factored eigenbasis is an expensive operation that needs to be amortised, tracking of the diagonal variance is a cheap operation. EKFAC therefore allows to perform partial updates of our curvature estimate G at the iteration level. We conduct an empirical evaluation of EKFAC on the deep auto-encoder optimisation task using fully-connected networks and CIFAR10 relying on deep convolutional neural networks where EKFAC shows improvements over K-FAC in optimisation.

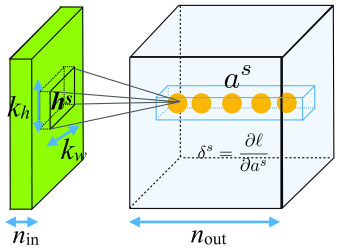
8.2. Background and notations

We are given a dataset $\mathcal{D}_{\text{train}}$ containing (input, target) examples (x,y), and a neural network $f_\theta(x)$ with parameter vector θ of size n_θ. We want to find a value of θ that minimises an empirical risk $R(\theta)$ expressed as an average of a loss ℓ incurred f_θ over the training set: $R(\theta) = \mathbb{E}_{(x,y)\in\mathcal{D}_{\text{train}}}\left[\ell(f_\theta(x),y)\right]$. We will use $\mathbb{E}$ to denote both expectations w.r.t. a distribution or, as here, averages over finite sets, as made clear by the subscript and context. Considered algorithms for optimising $R(\theta)$ use stochastic gradients $\nabla_\theta = \nabla_\theta(x,y) = \frac{\partial \ell(f_\theta(x),y)}{\partial \theta}$, or their average over a mini-batch of examples $\mathcal{D}_{\text{mini}}$ sampled from $\mathcal{D}_{\text{train}}$. Stochastic gradient descent (SGD) does a 1^{st} order update: $\theta \leftarrow \theta - \eta\nabla_\theta$ where η is a positive learning rate. 2^{nd} order methods first multiply ∇_θ by a preconditioning matrix G^{-1} yielding the update: $\theta \leftarrow \theta - \eta G^{-1}\nabla_\theta$. Preconditioning matrices for Natural Gradient (Amari, 1998) / Generalised Gauss-Newton (Schraudolph, 2001) / TONGA (Le Roux et al., 2008) can all be expressed as either (centred) covariance or (un-centred) second moment of ∇_θ, computed over slightly different distributions of (x,y). Thus natural gradient uses the Fisher Information Matrix, which for a probabilistic classifier can be expressed as $G = \mathbb{E}_{x\in\mathcal{D}_{\text{train}},y\sim p_\theta(\mathbf{y}|x)}\left[\nabla_\theta \nabla_\theta^\top\right]$ where the expectation is taken over targets sampled form the *model* $p_\theta = f_\theta$. Whereas the "empirical Fisher" approximation or generalised Gauss-Newton uses $G = \mathbb{E}_{(x,y)\in\mathcal{D}_{\text{train}}}\left[\nabla_\theta \nabla_\theta^\top\right]$. Our discussion and development applies regardless of the precise distribution over (x,y) used to estimate a G so we will from here on use $\mathbb{E}$ without a subscript.

Matrix G has a gigantic size $n_\theta \times n_\theta$, which makes it too big to compute and invert. In order to get a practical algorithm, we must find approximations of G that keep some of the relevant 2^{nd} order information while removing the unnecessary and computationally costly parts. A first simplification, adopted by nearly all prior approaches, consists in treating each layer of the neural network separately, ignoring cross-layer terms. This amounts to a first block-diagonal approximation of G: each block $G^{(l)}$ caters for the parameters of a single layer l. Now $G^{(l)}$ can typically still be extremely large.

A cheap but very crude approximation consists in using a diagonal $G^{(l)}$, i.e. taking into account the variance in each parameter dimension, but ignoring all covariance structure. A less stringent approximation was proposed by Heskes (2000) and later Martens & Grosse (2015). They propose to approximate $G^{(l)}$ as a Kronecker product $G^{(l)} \approx A \otimes B$ which involves two smaller matrices, making it much cheaper to store, compute and invert [1]. Specifically for a layer l that receives input h of size d_{in} and computes linear pre-activations $a = W^\top h$ of size d_{out} (biases omitted for simplicity) followed by some non-linear activation function, let the backpropagated gradient on a be $\delta = \frac{\partial \ell}{\partial a}$. The gradients on parameters $\theta^{(l)} = W$ will be $\nabla_W = \frac{\partial \ell}{\partial W} = \text{vec}(h\delta^\top)$. The Kronecker factored approximation of corresponding $G^{(l)} = \mathbb{E}\left[\nabla_W \nabla_W^\top\right]$ will use $A = \mathbb{E}\left[hh^\top\right]$ and $B = \mathbb{E}\left[\delta\delta^\top\right]$ i.e. matrices of size $d_{\text{in}} \times d_{\text{in}}$ and $d_{\text{out}} \times d_{\text{out}}$, whereas the full $G^{(l)}$ would be of size $d_{\text{in}}d_{\text{out}} \times d_{\text{in}}d_{\text{out}}$. Using this Kronecker approximation (known as K-FAC) corresponds to approximating entries of $G^{(l)}$ as follows: $G^{(l)}_{ij,i'j'} = \mathbb{E}\left[\nabla_{W_{ij}} \nabla_{W_{i'j'}}^\top\right] = \mathbb{E}\left[(h_i\delta_j)(h_{i'}\delta_{j'})\right] \approx \mathbb{E}\left[h_i h_{i'}\right] \mathbb{E}\left[\delta_j \delta_{j'}\right]$.

A similar principle can be applied to obtain a Kronecker-factored expression for the covariance of the gradients of the parameters of a convolutional layer (Grosse & Martens, 2016). To obtain matrices A and B one then needs to also sum over spatial locations and corresponding receptive fields, as illustrated in Figure 8.1.



$$\underset{(n_{in}\, n_{out}\, k_w\, k_h)^2}{G^{(l)}} \approx \underset{(n_{in}\, k_w\, k_h)^2}{A} \otimes \underset{(n_{out})^2}{B}$$

$A = \mathbb{E}\left[\sum_{(s,s')} h^s h^{s'\top}\right]$, $B = \mathbb{E}\left[\sum_{(s,s')} \delta^s \delta^{s'\top}\right]$

$s \in \mathcal{S}$, $s' \in \mathcal{S}$: spatial positions iterated over by the filter

h^s: flattened input subtensor (receptive field) at position s

δ^s: gradient of ℓ w.r.t. output of filter at position s

Figure 8.1. K-FAC for convolutional layer with $n_{\text{out}}n_{\text{in}}k_w k_h$ parameters.

8.3. Proposed method

8.3.1. Motivation: reflexion on diagonal rescaling in different coordinate bases

It is instructive to contrast the type of "exact" natural gradient preconditioning of the gradient that uses the full Fisher Information Matrix would yield, to what we do when approximating

1. Since $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$.

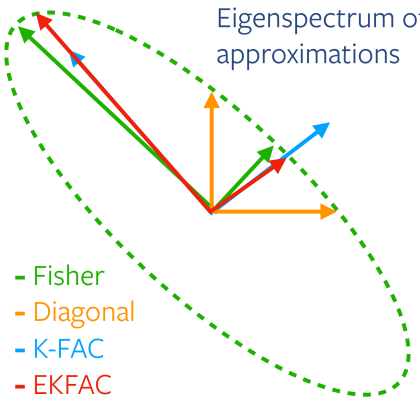
this by using a diagonal matrix only. Using the full matrix $G = \mathbb{E}[\nabla_\theta \nabla_\theta^\top]$ yields the natural gradient update: $\theta \leftarrow \theta - \eta G^{-1} \nabla_\theta$. When resorting to a diagonal approximation we instead use $G_{\text{diag}} = \text{diag}(\sigma_1^2, \ldots, \sigma_{n_\theta}^2)$ where $\sigma_i^2 = G_{i,i} = \mathbb{E}[(\nabla_\theta)_i^2]$. So that update $\theta \leftarrow \theta - \eta G_{\text{diag}}^{-1} \nabla_\theta$ amounts to preconditioning the gradient vector ∇_θ by dividing each of its coordinates by an estimated second moment σ_i^2. This diagonal rescaling happens in the initial basis of parameters θ. By contrast, a full natural gradient update can be seen to do a similar diagonal rescaling, not along the initial parameter basis axes, but along the axes of the *eigenbasis* of the matrix G. Let $G = USU^\top$ be the eigendecomposition of G. The operations that yield the full natural gradient update $G^{-1}\nabla_\theta = US^{-1}U^\top \nabla_\theta$ correspond to the sequence of a) multiplying gradient vector ∇_θ by $U^\top$ which corresponds to switching to the eigenbasis: $U^\top \nabla_\theta$ yields the coordinates of the gradient vector expressed in that basis b) multiplying by a diagonal matrix S^{-1}, which rescales each coordinate i (in that eigenbasis) by S_{ii}^{-1} c) multiplying by U, which switches the rescaled vector back to the initial basis of parameters. It is easy to show that $S_{ii} = \mathbb{E}[(U^\top \nabla_\theta)_i^2]$ (proof is given in Appendix 8.A.2). So similarly to what we do when using a diagonal approximation, we are rescaling by the second moment of gradient vector components, but rather than doing this in the initial parameter basis, we do this in the eigenbasis of G. Note that the variance measured along the leading eigenvector can be much larger than the variance along the axes of the initial parameter basis, so the effects of the rescaling by using either the full G or its diagonal approximation can be very different.

Now what happens when we use the less crude K-FAC approximation instead? We approximate[2] $G \approx A \otimes B$ yielding the update $\theta \leftarrow \theta - \eta(A \otimes B)^{-1}\nabla_\theta$. Let us similarly look at it through its eigendecomposition. The eigendecomposition of the Kronecker product $A \otimes B$ of two real symmetric positive semi-definite matrices can be expressed using their own eigendecomposition $A = U_A S_A U_A^\top$ and $B = U_B S_B U_B^\top$, yielding $A \otimes B = (U_A S_A U_A^\top) \otimes (U_B S_B U_B^\top) = (U_A \otimes U_B)(S_A \otimes S_B)(U_A \otimes U_B)^\top$. $U_A \otimes U_B$ gives the orthogonal eigenbasis of the Kronecker product, we call it the Kronecker-Factored Eigenbasis (KFE). $S_A \otimes S_B$ is the diagonal matrix containing the associated eigenvalues. Note that each such eigenvalue will be a product of an eigenvalue of A stored in S_A and an eigenvalue of B stored in S_B. We can view the action of the resulting Kronecker-factored preconditioning in the same way as we viewed the preconditioning by the full matrix: it consists in a) expressing gradient vector ∇_θ in a different basis $U_A \otimes U_B$ which can be thought of as approximating the directions of U, b) doing a diagonal rescaling by $S_A \otimes S_B$ *in that basis,* c) switching back to the initial parameter space. Here however the rescaling factor $(S_A \otimes S_B)_{ii}$ is *not* guaranteed to match the second moment along the associated eigenvector $\mathbb{E}[((U_A \otimes U_B)^\top \nabla_\theta)_i^2]$.

2. This approximation is done separately for each block $G^{(l)}$, we dropped the superscript to simplify notations.

In summary (see Figure 8.2):

- Full matrix G preconditioning will scale by variances estimated along the eigenbasis of G.
- Diagonal preconditioning will scale by variances properly estimated, but along the initial parameter basis, which can be very far from the eigenbasis of G.
- K-FAC preconditioning will scale the gradient along the KFE basis that will likely be closer to the eigenbasis of G, but doesn't use properly estimated variances along these axes for this scaling (the scales being themselves constrained to being a Kronecker product $S_A \otimes S_B$).



Rescaling of the gradient is done along a specific basis; length of vectors indicate (square root of) amount of down-scaling. Exact Fisher Information Matrix rescales according to eigenvectors/values of exact covariance structure (green ellipse). Diagonal approximation uses parameter coordinate basis, scaling by actual variance measured along these axes (indicated by horizontal and vertical orange arrows touching exactly the ellipse), K-FAC uses directions that approximate Fisher Information Matrix eigenvectors, but uses approximate scaling (blue arrows *not* touching the ellipse). EKFAC corrects this.

Figure 8.2. Cartoon illustration of rescaling achieved by different preconditioning strategies

8.3.2. Eigenvalue-corrected Kronecker Factorisation (EKFAC)

To correct for the potentially inexact rescaling of K-FAC, and obtain a better but still computationally efficient approximation, instead of $G_{\text{KFAC}} = A \otimes B = (U_A \otimes U_B)(S_A \otimes S_B)(U_A \otimes U_B)^\top$ we propose to use an *Eigenvalue-corrected Kronecker Factorisation*: $G_{\text{EKFAC}} = (U_A \otimes U_B)S^*(U_A \otimes U_B)^\top$ where S^* is the diagonal matrix defined by $S^*_{ii} = s^*_i = \mathbb{E}[((U_A \otimes U_B)^\top \nabla_\theta)^2_i]$. Vector s^* is the vector of second moments of the gradient vector coordinates expressed in the approximate basis $U_A \otimes U_B$ and can be efficiently estimated and stored.

In Appendix 8.A.1 we prove that this S^* is the optimal diagonal rescaling in that basis, in the sense that $S^* = \arg\min_S \|G - (U_A \otimes U_B)S(U_A \otimes U_B)^\top\|_F$ s.t. S is diagonal: it minimises the approximation error to G as measured by Frobenius norm (denoted $\|\cdot\|_F$), which K-FAC's corresponding $S = S_A \otimes S_B$ cannot generally achieve. A corollary of this is that we will always have $\|G - G_{\text{EKFAC}}\|_F \leq \|G - G_{\text{KFAC}}\|_F$ i.e. EKFAC yields a better approximation of G than K-FAC (Theorem 8.1 proven in Appendix). Figure 8.2 illustrates the different rescaling strategies, including EKFAC.

Potential benefits. Since EKFAC is a better approximation of G than K-FAC (in the limited sense of Frobenius norm of the residual) it could yield a better preconditioning of the gradient for optimising neural networks [3]. Another potential benefit is linked to computational efficiency: even if K-FAC yielded a reasonably good approximation, it is costly to re-estimate and invert matrices A and B, so this has to be amortised over many updates: re-estimation of the preconditioning is thus typically done at a much lower frequency than the parameter updates, and may lag behind, no longer accurately reflecting the local 2^{nd} order information. Re-estimating the Kronecker-factored Eigenbasis for EKFAC is similarly costly and must be similarly amortised. But re-estimating the diagonal scaling s^* in that basis is cheap, doable with every mini-batch, so we can hope to reactively track and leverage the changes in 2^{nd} order information along these directions.

Algorithm. Using Eigenvalue-corrected Kronecker factorisation (EKFAC) for neural network optimisation involves: a) periodically (every n mini-batches) computing the Kronecker-factored Eigenbasis by doing an eigendecomposition of the same A and B matrices as K-FAC; b) estimating scaling vector s^* as second moments of gradient coordinates in that implied basis; c) preconditioning gradients accordingly prior to updating model parameters. Algorithm 1 provides a high level pseudo-code for the case of fully-connected layers [4], and when using EKFAC to approximate the "empirical Fisher". In this version, we re-estimate s^* from scratch on each mini-batch. An alternative is to update a running average estimate of the variance (of either individual gradients or mini-batch averaged gradients), denoted by EKFAC-ra (for running average) in section 8.4.

Dual view by working in the KFE. Instead of thinking of this new method as an improved factorisation of G that we use as a preconditioning matrix, we can adopt the alternate view of applying a diagonal method, but in a different basis where the diagonal approximation is more accurate (an assumption we empirically confirm in Figure 8.3). This can be viewed by reinterpreting the update given by EKFAC as a 3 step process: project the gradient in the KFE (–), apply natural gradient in this basis (–), then project back to the parameter space (–):

$$G_{\text{EKFAC}}^{-1}\nabla_\theta = (U_A \otimes U_B)S^{*-1}(U_A \otimes U_B)^\top \nabla_\theta \tag{8.1}$$

Note that by writing $\tilde{\nabla}_\theta = (U_A \otimes U_B)^\top \nabla_\theta$ the projected gradient in KFE, the computation of the coefficients s_i^* simplifies in $s_i^* = \mathbb{E}[(\tilde{\nabla}_\theta)_i^2]$. Figure 8.3 shows gradient correlation matrices in both the initial parameter basis and in the KFE. Gradient components appear far less

3. Although there is no guarantee. In particular G_{EKFAC} being a better approximation of G does not guarantee that $G_{\text{EKFAC}}^{-1}\nabla_\theta$ will be closer to the natural gradient update direction $G^{-1}\nabla_\theta$.

4. EKFAC for convolutional layers follows the same structure, but require a more *convoluted* notation.

Algorithm 1 EKFAC for fully connected networks

Require: n: recompute eigenbasis every n mini-batches
Require: η: learning rate
Require: ϵ: damping parameter

procedure EKFAC($\mathcal{D}_{\text{train}}$)
  while convergence is not reached, iteration i **do**
    sample a mini-batch $\mathcal{D}$ from $\mathcal{D}_{\text{train}}$
    Do forward and backprop pass as needed to obtain h and δ
    for all layer l **do**
      if $i \,\%\, n = 0$ **then** ▷ Amortise eigendecomposition
        ComputeEigenBasis($\mathcal{D}$, l)
      ComputeScalings($\mathcal{D}$, l)
      $\nabla^{\text{mini}} \leftarrow \mathbb{E}_{(x,y)\in\mathcal{D}}\left[\nabla_\theta^{(l)}(x,y)\right]$
      UpdateParameters(∇^{mini}, l)

procedure ComputeEigenBasis($\mathcal{D}$, l)
  $U_A^{(l)}, S_A^{(l)} \leftarrow$ eigendecomposition $\left(\mathbb{E}_{\mathcal{D}}\left[h^{(l)}h^{(l)\top}\right]\right)$
  $U_B^{(l)}, S_B^{(l)} \leftarrow$ eigendecomposition $\left(\mathbb{E}_{\mathcal{D}}\left[\delta^{(l)}\delta^{(l)\top}\right]\right)$

procedure ComputeScalings($\mathcal{D}$, l)
  $s^{*(l)} \leftarrow \mathbb{E}_{\mathcal{D}}\left[\left(\left(U_A^{(l)} \otimes U_B^{(l)}\right)^\top \nabla_\theta^{(l)}\right)^2\right]$ ▷ Project gradient in eigenbasis[1]

procedure UpdateParameters(∇^{mini}, l)
  $\tilde{\nabla} \leftarrow \left(U_A^{(l)} \otimes U_B^{(l)}\right)^\top \nabla^{\text{mini}}$ ▷ Project gradients in eigenbasis[1]
  $\tilde{\nabla} \leftarrow \tilde{\nabla} / \left(s^{*(l)} + \epsilon\right)$ ▷ Element-wise scaling
  $\nabla^{\text{precond}} \leftarrow \left(U_A^{(l)} \otimes U_B^{(l)}\right) \tilde{\nabla}$ ▷ Project back in parameter basis[1]
  $\theta^{(l)} \leftarrow \theta^{(l)} - \eta \nabla^{\text{precond}}$ ▷ Update parameters

[1]Can be efficiently computed using the following identity: $(A \otimes B)\text{vec}(C) = B^\top C A$

correlated when expressed in the KFE, which justifies the usage of a diagonal method *in that basis*.

This viewpoint brings us close to network reparametrisation approaches such as Fujimoto & Ohira (2018), whose proposal – that was already hinted towards by Desjardins et al. (2015) – amounts to a reparametrisation equivalent of K-FAC. More precisely, while Desjardins et al. (2015) empirically explored a reparametrisation that uses only input covariance A (and thus corresponds only to "half of" K-FAC), Fujimoto & Ohira (2018) extend this to use also

backpropagated gradient covariance B, making it essentially equivalent to K-FAC (with a few extra twists). Our approach differs in that moving to the KFE corresponds to a change of *orthonormal basis*, and that we cheaply track and perform a more optimal *full diagonal* rescaling in that basis, rather than the constrained factored $S_A \otimes S_B$ diagonal that these other approaches are implicitly using.

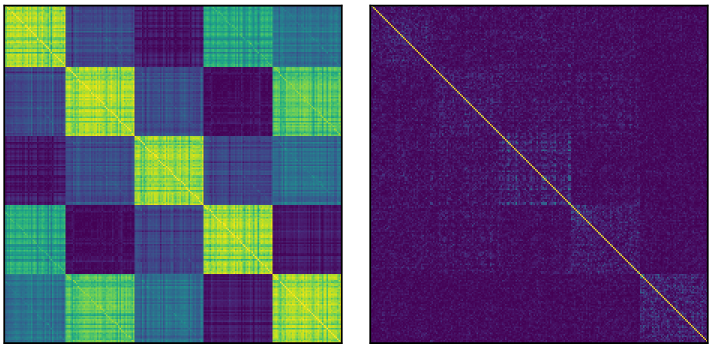


Figure 8.3. Gradient *correlation* matrices measured in the initial parameter basis and in the Kronecker-factored Eigenbasis (KFE), computed from a small 4 sigmoid layer MLP trained on MNIST. We only plotted the block corresponding to 250 parameters in the 2nd layer.

8.4. Experiments

This section presents an empirical evaluation of our proposed Eigenvalue Corrected K-FAC (EKFAC) algorithm in two variants: EKFAC estimates scalings s^* as second moment of intra-batch gradients (in KFE coordinates) as in Algorithm 1, whereas EKFAC-ra estimates s^* as a running average of squared mini-batch gradient (in KFE coordinates). We compare them with K-FAC and other baselines, primarily SGD with momentum, with and without batch-normalisation (BN). For all our experiments K-FAC and EKFAC approximate the "empirical Fisher" G. This research focuses on improving optimisation techniques, so except when specified otherwise, we performed model and hyper-parameter selection based on the performance of the optimisation objective, i.e. on training loss.

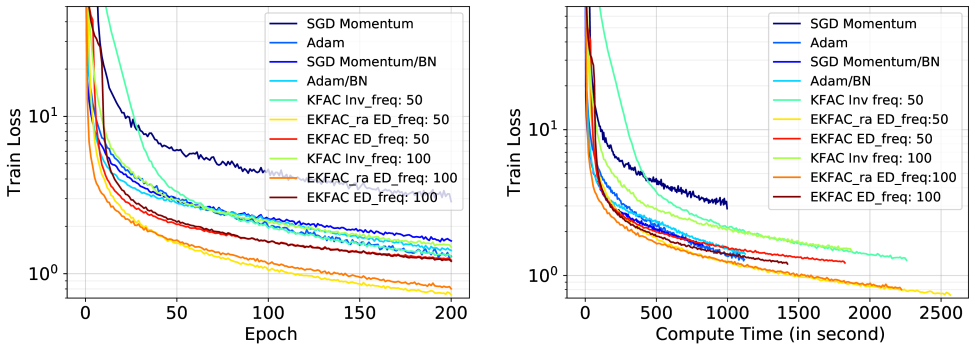
8.4.1. Deep auto-encoder

We consider the task of minimising the reconstruction error of an 8-layer auto-encoder on the MNIST dataset, a standard task used to benchmark optimisation algorithms (Hinton & Salakhutdinov, 2006; Martens & Grosse, 2015; Desjardins et al., 2015). The model consists of an encoder composed of 4 fully-connected sigmoid layers, with a number of hidden units per layer of respectively 1000, 500, 250, 30, and a symmetric decoder (with untied weights).

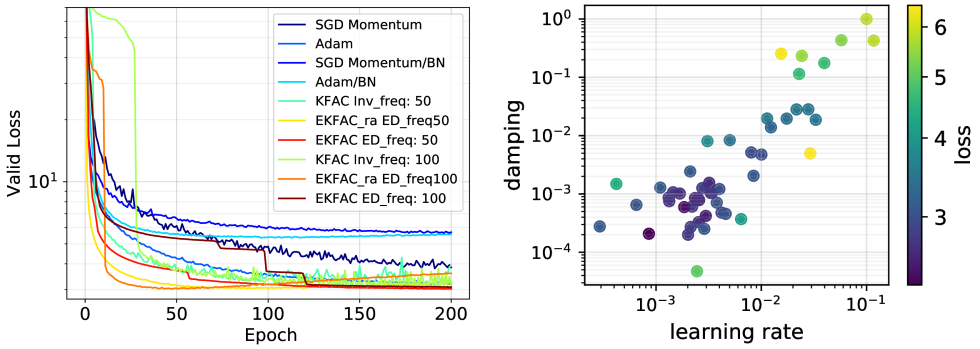
We compare our EKFAC, computing the second moment statistics through its mini-batch, and EKFAC-ra, its running average variant, with different baselines (K-FAC, SGD, SGD with BN, ADAM and ADAM with BN). For each algorithm, best hyper-parameters were selected using a mix of grid and random search based on training error. Grid values for hyper-parameters are: learning rate η and damping ϵ in $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$, mini-batch size in {200, 500}, frequency of reparametrisation (i.e. recomputing the inverse or eigendecomposition) for K-FAC and EKFAC: either every 50 or 100 updates. In addition we explored 20 values for (η, ϵ) by random search around each grid points. We found that an extra care must be taken when choosing the values of the learning rate and damping parameter ϵ in order to get good performances, as often observed when working with algorithms that leverage curvature information (see Figure 8.4 (d)). The learning rate and the damping parameters are kept constant during training.

Figure 8.4 (a) reports the training loss through training and shows that EKFAC and EKFAC-ra both minimise faster the training loss per epoch than K-FAC and the other baselines. In addition, Figure 8.4 (b) shows that an efficient estimation of diagonal scaling vector s^*, as done by EKFAC, allows to achieve faster training in wall-clock time. The use of running average in EKFAC-ra leads to a faster training than K-FAC, while EKFAC is on par with the latter. Finally, EKFAC and EKFAC-ra achieve better optimisation on this task while maintaining their generalisation performances (Figure 8.4 (c)).

Next we investigate how the frequency of the reparametrisations affects the optimisation. In Figure 8.5, we compare K-FAC/EKFAC with different reparametrisation frequencies to a strong K-FAC baseline where we reestimate and compute the inverse at each update. This baseline outperforms the amortised version (as a function of number of of epochs), and is likely to leverage a better approximation of G as it recomputes the approximated eigenbasis at each update. However it comes at a high computational cost, as shown in Figure 8.5 (b). Amortising the eigendecomposition allows to strongly decrease the computational cost while slightly degrading the optimisation performances. In addition, Figure 8.5 (a) shows that the amortised EKFAC preserves better the optimisation performances than its K-FAC counterpart. EKFAC re-estimates at each update, the diagonal second moments in the KFE basis, which correspond to the eigenvalues of the EKFAC approximation of G. This could reduce its estimation error, as the approximation can better match the true curvature of the loss function. To verify this hypothesis, we investigate how the eigenspectrum of the true empirical Fisher G changes compared to the eigenspectrum of its approximations as G_{KFAC} and G_{EKFAC}. In Figure 8.5 (c), we track their eigenspectra and report the l_2 distance between them during training. We compute the KFE once at the beginning and then keep it fixed during training. We focus on the 4^{th} layer of the auto-encoder, since it is small which



(a) Training loss **(b)** Wall-Clock Time



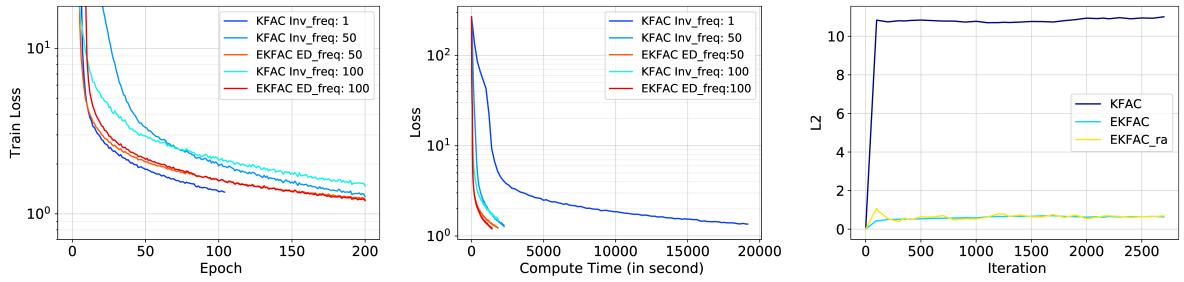
(c) Validation Loss **(d)** Hyperparameters

Figure 8.4. MNIST Deep Auto-Encoder task. Models are selected based on the best loss achieved during training. An ED_freq (Inv_freq) of 50 means eigendecomposition (inverse) are recomputed every 50 updates. **(a)** Training loss vs epochs. Both EKFAC and EKFAC-ra show an optimisation benefit compared to amortised K-FAC and the other baselines. **(b)** Training loss vs wall-clock time. Optimisation benefits transfer to faster training for EKFAC-ra. **(c)** Validation loss. K-FAC, EKFAC and BN achieve the same validation performances. **(d)** Sensitivity to hyper-parameters values. Colour corresponds to final loss reached after 20 epochs for batch size 200.

allows estimating the corresponding G and computing its eigenspectrum at a reasonable cost. We observe that the spectrum of G_{KFAC} quickly diverges from the spectrum of G, whereas the cheap frequent reestimation of the diagonal scaling for G_{EKFAC} allows the spectrum of G_{EKFAC} to stay much closer to that of G. This is true for both the running average and intra-batch versions of EKFAC.

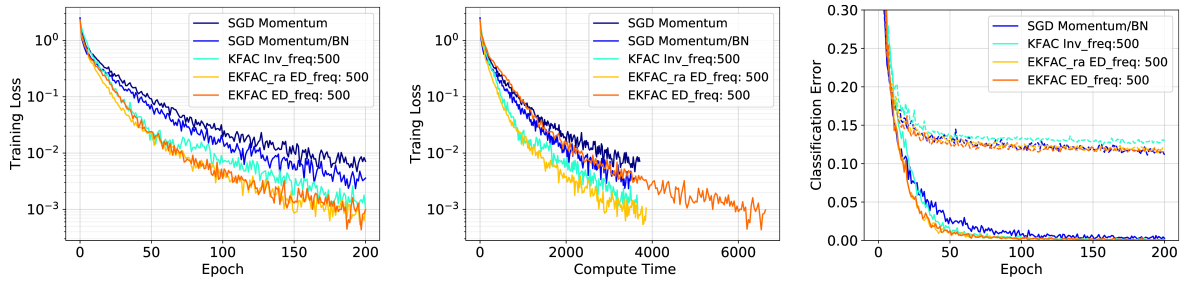
8.4.2. CIFAR10

In this section, we evaluate our proposed algorithm on the CIFAR10 dataset using a VGG11 convolutional neural network (Simonyan & Zisserman, 2015) and a Resnet34 (He et al., 2016a). To implement K-FAC/EKFAC in a convolutional neural network, we rely on the



(a) Training loss **(b)** Wall-clock time **(c)** l_2 distance to spectrum of G

Figure 8.5. Impact of frequency of inverse/eigendecomposition recomputation for K-FAC/EKFAC. A freq. of 50 indicates a recomputation every 50 updates. **(a)(b)** Training loss v.s. epochs and wall-clock time. We see that EKFAC preserves better its optimisation performances as the frequency of eigendecomposition is decreased. **(c)**. Evolution of l_2 distance between the eigenspectrum of empirical Fisher G and eigenspectra of approximations G_{KFAC} and G_{EKFAC}. We see that the spectrum of G_{KFAC} quickly diverges from the spectrum of G, whereas the EKFAC variants, thanks to their frequent diagonal reestimation, manage to much better track G.



(a) Training loss **(b)** Wall-clock time **(c)** Accuracy (solid is train, dash is validation)

Figure 8.6. VGG11 on CIFAR10. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. In **(a)** and **(b)**, we report the performance of the hyper-parameters reaching the lowest training loss for each epoch (to highlight which optimizers perform best given a fixed epoch budget). In **(c)** models are selected according to the best overall validation error. When the inverse/eigendecomposition is amortised on 500 iterations, EKFAC-ra shows an optimisation benefit while maintaining its generalisation capability.

SUA approximation (Grosse & Martens, 2016) which has been shown to be competitive in practice (Laurent et al., 2018). We highlight that we do not use BN in our model when they are trained using K-FAC/EKFAC.

As in the previous experiments, a grid search is performed to select the hyper-parameters. Around each grid point, learning rate and damping values are further explored through random search. We experiment with constant learning rate in this section, but explore learning rate schedule with K-FAC/EKFAC in Appendix 8.C.2. In figures reporting the model training loss

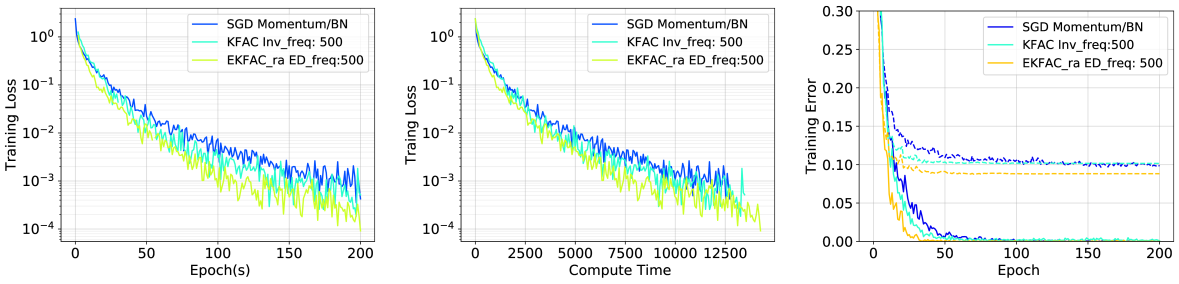
per epoch or wall-clock-time, we report the performance of the hyper-parameters attaining the lowest training loss for each epoch. This per-epoch model selection allows to show which model reach the lowest cost during training and also which model optimises best given any "epoch budget".

In Figure 8.6, we compare EKFAC/EKFAC-ra to K-FAC and SGD Momentum with or without BN when training a VGG-11 network. We use a batch size of 500 for the K-FAC based approaches and 200 for the SGD baselines. Figure 8.6 (a) show that EKFAC yields better optimisation than the SGD baselines and K-FAC in training loss per epoch when the computation of the KFE is amortised. Figure 8.6 (c) also shows that models trained with EKFAC maintain good generalisation. EKFAC-ra shows some wall-clock time improvements over the baselines in that setting (Figure 8.6 (b)). However, we observe that using K-FAC with a batch size of 200 can catch-up EKFAC in wall-clock time despite being outperformed in term of optimisation per iteration (see Figure 8.9, in Appendix). VGG11 is a relatively small network by modern standard and the K-FAC (with SUA approximation) remains computationally bearable in this model. We hypothesise that using smaller batches, K-FAC can be updated often enough per epoch to have a reasonable estimation error while not paying too high a computational cost .

In Figure 8.7, we report similar results on the Resnet34. We compare EKFAC with running averages with K-FAC and SGD-Momentum (with and without BN). In order to train the Resnet34 without BN, we need to rely on a careful initialisation scheme in order to ensure good signal propagation during the forward and backward passes (see Appendix 8.B for details). EKFAC outperforms both K-FAC (when amortised) and SGD-Momentum in term of optimisation per epochs, and compute time. This gain is robust across different batch sizes as shown in Figure 8.10.

8.5. Conclusion and future work

In this work, we introduced the Eigenvalue-corrected Kronecker factorisation (EKFAC), an approximate factorisation of the (empirical) Fisher Information Matrix that is computationally manageable while still being accurate. We formally proved (in Appendix) that EKFAC yields a more accurate estimate than its closest competitor K-FAC, in the sense of the Frobenius norm. In addition, we showed that our algorithm allows to cheaply perform partial updates of our curvature estimate, maintaining an up-to-date estimate of its eigenvalues while keeping the estimate of its eigenbasis fixed. This partial updating proves competitive when applied to standard optimisation tasks, both with respect to the number of iterations and wall-clock time.



(a) Training loss **(b)** Wall-clock time **(c)** Accuracy (solid is train, dash is validation)

Figure 8.7. CIFAR10 with a Resnet Network with 34 layers. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. In **(a)** and **(b)**, we report the performance of the hyper-parameters reaching the lowest training loss for each epoch (to highlight which optimizers perform best given a fixed epoch budget). In **(c)** we select model according to the best overall validation error. When the inverse/eigen decomposition is amortised on 500 iterations, EKFAC-ra shows optimisation and computational time benefits while still maintaining its generalisation capability.

Our approach amounts to normalising the gradient by its 2^{nd} moment component-wise in a Kronecker-factored Eigenbasis (KFE). But one could apply other component-wise (diagonal) adaptive algorithms such as Adagrad (Duchi et al., 2011), RMSProp (Tieleman & Hinton, 2012) or Adam (Kingma & Ba, 2015), *in the KFE* where the diagonal approximation is much more accurate. This is left for future work. We also intend to explore alternative strategies for obtaining the approximate eigenbasis and investigate how to increase the robustness of the algorithm with respect to the damping hyperparameter. We also want to explore novel regularisation strategies, so that the advantage of efficient optimisation algorithms can more reliably be translated to generalisation error.

Acknowledgements

The experiments were conducted using PyTorch (Paszke et al. (2017)). The authors would like to acknowledge the support of Calcul Quebec, Compute Canada, CIFAR and Facebook for research funding and computational resources.

8.A. Proofs

8.A.1. Proof that EKFAC does optimal diagonal rescaling in the KFE

Lemma 8.1. *Let $G = \mathbb{E}\left[\nabla_\theta \nabla_\theta^\top\right]$ a real positive semi-definite matrix. And let Q a given orthogonal matrix. Among diagonal matrices, diagonal matrix D with diagonal entries $D_{ii} = \mathbb{E}\left[\left(Q^\top \nabla_\theta\right)_i^2\right]$ minimise approximation error $e = \left\|G - QDQ^\top\right\|_F$ (measured as Frobenius norm).*

Proof. Since the Frobenius norm remains unchanged through multiplication by an orthogonal matrix we can write

$$\begin{aligned}
e^2 &= \left\|G - QDQ^\top\right\|_F^2 \\
&= \left\|Q^\top\left(G - QDQ^\top\right)Q\right\|_F^2 \\
&= \left\|Q^\top G Q - D\right\|_F^2 \\
&= \underbrace{\sum_i (Q^\top G Q - D)_{ii}^2}_{\text{diagonal}} + \underbrace{\sum_i \sum_{j \neq i} (Q^\top G Q)_{ij}^2}_{\text{off-diagonal}}
\end{aligned}$$

Since D is diagonal, it does not affect the off-diagonal terms.

The squared diagonal terms all reach their minimum value 0 by setting $D_{ii} = \left(Q^\top G Q\right)_{ii}$ for all i:

$$\begin{aligned}
D_{ii} &= \left(Q^\top G Q\right)_{ii} \\
&= \left(Q^\top \mathbb{E}\left[\nabla_\theta \nabla_\theta^\top\right] Q\right)_{ii} \\
&= \left(\mathbb{E}\left[Q^\top \nabla_\theta \nabla_\theta^\top Q\right]\right)_{ii} \\
&= \left(\mathbb{E}\left[Q^\top \nabla_\theta \left(Q^\top \nabla_\theta\right)^\top\right]\right)_{ii} \\
&= \mathbb{E}\left[\left(Q^\top \nabla_\theta\right)_i^2\right] \text{ since } Q^\top \nabla_\theta \text{ is a vector}
\end{aligned}$$

We have thus shown that diagonal matrix D with diagonal entries $D_{ii} = \mathbb{E}\left[\left(Q^\top \nabla_\theta\right)_i^2\right]$ minimise e^2. Since Frobenius norm e is non-negative this implies that D also minimises e. □

Theorem 8.1. *Let $G = \mathbb{E}\left[\nabla_\theta \nabla_\theta^\top\right]$ the matrix we want to approximate. Let $G_{\text{KFAC}} = A \otimes B$ the approximation of G obtained by K-FAC. Let $A = U_A S_A U_A^\top$ and $B = U_B S_B U_B^\top$*

eigendecomposition of A and B. The diagonal rescaling that EKFAC does in the Kronecker-factored Eigenbasis (KFE) $U_A \otimes U_B$ is optimal in the sense that it minimises the Frobenius norm of the approximation error: among diagonal matrices D, approximation error $e = \left\|G - (U_A \otimes U_B)D(U_A \otimes U_B)^\top\right\|_F$ is minimised by the matrix with with diagonal entries $D_{ii} = s_i^* = \mathbb{E}\left[\left((U_A \otimes U_B)^\top \nabla_\theta\right)_i^2\right]$.

PROOF. This follows directly by setting $Q = U_A \otimes U_B$ in Lemma 8.1. Note that the Kronecker product of two orthogonal matrices yields an orthogonal matrix. □

Theorem 8.2. *Let G_{KFAC} the K-FAC approximation of G and G_{EKFAC} the EKFAC approximation of G, we always have $\|G - G_{\text{EKFAC}}\|_F \leq \|G - G_{\text{KFAC}}\|_F$.*

PROOF. This follows trivially from Theorem 8.1 on the optimality of the EKFAC diagonal rescaling.

Since $D = S^*$, with the $S^* = \text{diag}(s^*)$ of EKFAC, minimizes $\left\|G - (U_A \otimes U_B)D(U_A \otimes U_B)^\top\right\|_F$, it implies that:

$$\begin{aligned}
\|G - (U_A \otimes U_B)S^*(U_A \otimes U_B)^\top\|_F &\leq \|G - (U_A \otimes U_B)(S_A \otimes S_B)(U_A \otimes U_B)^\top\|_F \\
\|G - G_{\text{EKFAC}}\|_F &\leq \|G - (U_A S_A U_A^\top) \otimes (U_B S_B U_B^\top)\|_F \\
\|G - G_{\text{EKFAC}}\|_F &\leq \|G - A \otimes B)\|_F \\
\|G - G_{\text{EKFAC}}\|_F &\leq \|G - G_{\text{KFAC}}\|_F
\end{aligned}$$

□

We have thus demonstrated that EKFAC yields a better approximation (more precisely: at least as good in Frobenius norm error) of G than K-FAC.

8.A.2. Proof that $S_{ii} = \mathbb{E}\left[\left(U^\top \nabla_\theta\right)_i^2\right]$

Theorem 8.3. *Let $G = \mathbb{E}\left[\nabla_\theta \nabla_\theta^\top\right]$ and $G = USU^\top$ its eigendecomposition.*
Then $S_{ii} = \mathbb{E}\left[\left(U^\top \nabla_\theta\right)_i^2\right]$.

PROOF. Starting from eigendecomposition $G = USU^\top$ and the fact that U is orthogonal so that $U^\top U = I$ we can write

$$\begin{aligned} G &= USU^\top \\ U^\top G U &= U^\top U S U^\top U \\ U^\top \underbrace{G}_{\mathbb{E}\left[\nabla_\theta \nabla_\theta^\top\right]} U &= S \end{aligned}$$

so that

$$\begin{aligned} S_{ii} &= \left(U^\top \mathbb{E}\left[\nabla_\theta \nabla_\theta^\top\right] U\right)_{ii} \\ &= \left(\mathbb{E}\left[U^\top \nabla_\theta \nabla_\theta^\top U\right]\right)_{ii} \\ &= \left(\mathbb{E}\left[U^\top \nabla_\theta \left(U^\top \nabla_\theta\right)\right]\right)_{ii} \\ &= \mathbb{E}\left[\left(U^\top \nabla_\theta \left(U^\top \nabla_\theta\right)\right)_{ii}\right] \\ &= \mathbb{E}\left[\left(U^\top \nabla_\theta\right)_i^2\right] \end{aligned}$$

where we obtained the last equality by observing that $U^\top \nabla_\theta$ is a vector and that the diagonal entries of the matrix $aa^\top$ for any vector a are given by a^2 where the square operation is element-wise. □

8.B. Residual network initialisation

To train residual networks without using BN, one need to initialise them carefully, so we used the following procedure, denoting n the fan-in of the layer:

(1) We use the He initialisation for each layer directly preceded by a ReLU (He et al., 2015): $W \sim \mathcal{N}(0, 2/n)$, $b = 0$.

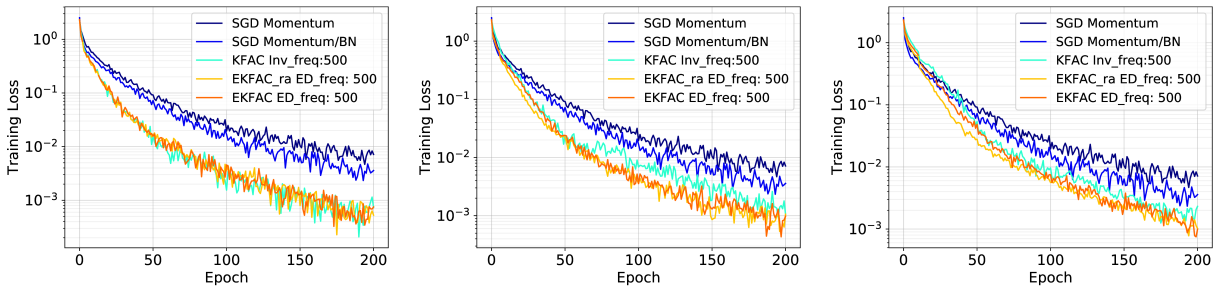
(2) Each layer not directly preceded by an activation function (for example the convolution in a skip connection) is initialised as: $W \sim \mathcal{N}(0, 1/n)$, $b = 0$. This can be derived from the He initialisation, using the Identity as activation function.

(3) Inspired from Goyal et al. (2017), we divide the scale of the last convolution in each residual block by a factor 10: $W \sim \mathcal{N}(0, 0.2/n)$, $b = 0$. This not only helps preserving the variance through the network but also eases the optimisation at the beginning of the training.

8.C. Additional empirical results

8.C.1. Impact of batch size

In this section, we evaluate the impact of the batch size on the optimisation performances for K-FAC and EKFAC. In Figure 8.8, we reports the training loss performance per epochs for different batch sizes for VGG11. We observe that the optimisation gain of EKFAC with respect of K-FAC diminishes as the batch size gets smaller.

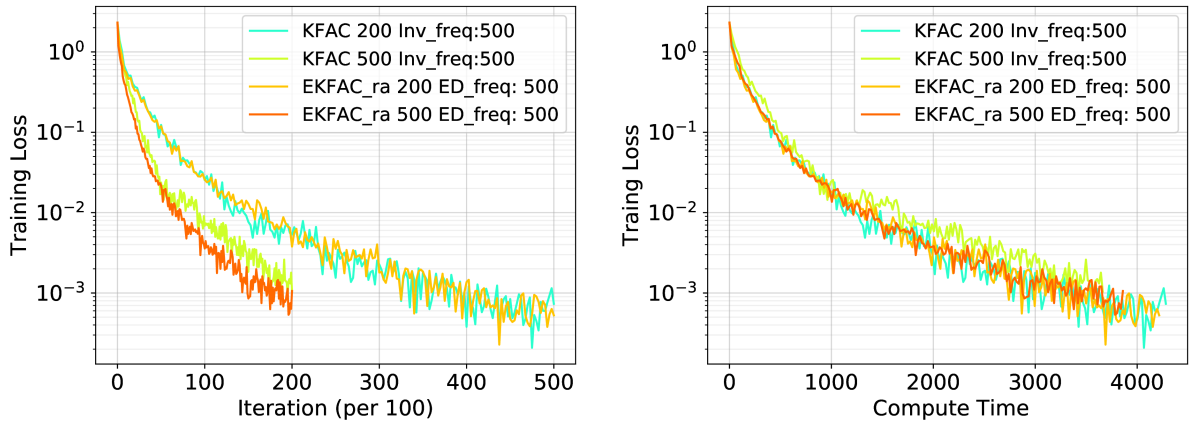


(a) Batch size = 200 **(b)** Batch size = 500 **(c)** Batch size = 1000

Figure 8.8. VGG11 on CIFAR10. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. We perform model selections according to best training loss at each epoch. On this setting, we observe that the optimisation gain of EKFAC with respect of K-FAC diminishes as the batch size reduces.

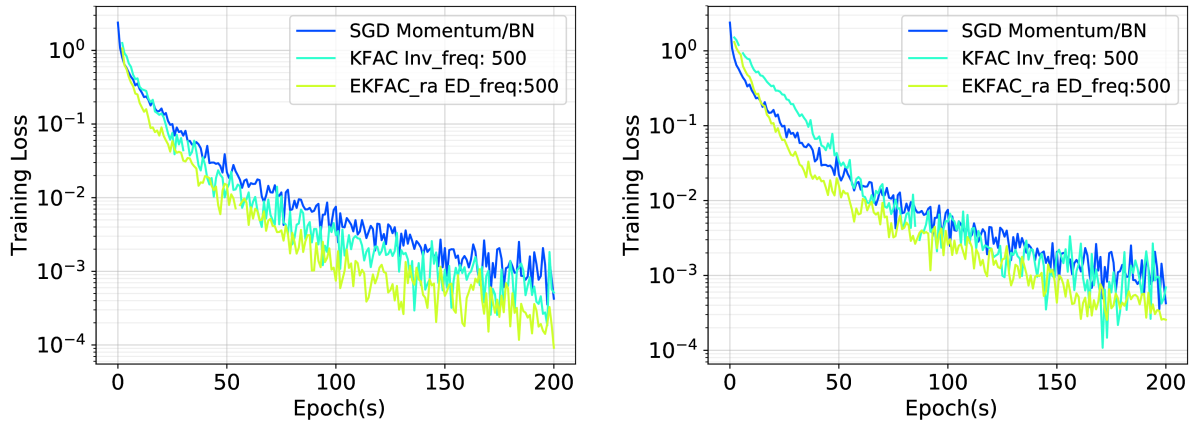
In Figure 8.9, we look at the training loss per iterations and the training loss per computation time for different batch sizes, again on VGG11. EKFAC shows optimisation benefits over K-FAC as we increase the batch size (thus reducing the number of inverse/eigendecomposition per epoch). This gain does not translate in faster training in term of computation time in that setting. VGG11 is a relatively small network by modern standard and the SUA approximation remains computationally bearable on this model. We hypothesise that using smaller batches, K-FAC can be updated often enough per epoch to have a reasonable estimation error while not paying a computational price too high.

In Figure 8.10, we perform a similar experiment on the Resnet34. In this setting, we observe that the optimisation gain of EKFAC with respect of K-FAC remains consistent across batch sizes.



(a) Iterations **(b)** Wall-clock time

Figure 8.9. VGG11 on CIFAR10. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. We perform model selections according to best training loss at each epoch. **(a)** Training loss per iterations for different batch sizes. **(b)** Training loss per computation time for different batch sizes. EKFAC shows optimisation benefits over K-FAC as we increases the batch size (thus reducing the number of inverse/eigendecomposition per epoch). This gain does not translate in faster training in terms of wall-clock time in that setting.



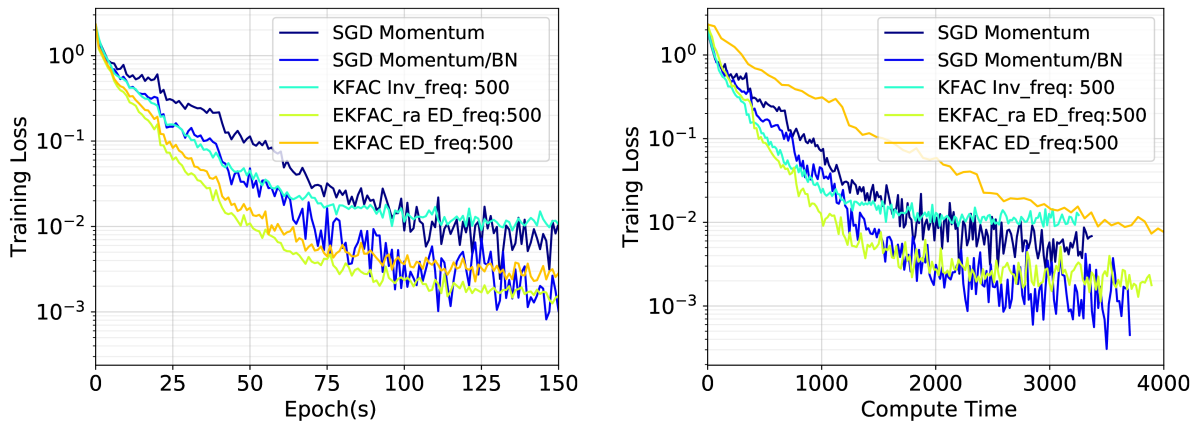
(a) Batch size = 200 **(b)** Batch size = 500

Figure 8.10. Resnet34 on CIFAR10. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. We perform model selections according to best training loss at each epoch. In this setting, we observe that the optimisation gain of EKFAC with respect of K-FAC remains consistent across batch sizes.

8.C.2. Learning rate schedule

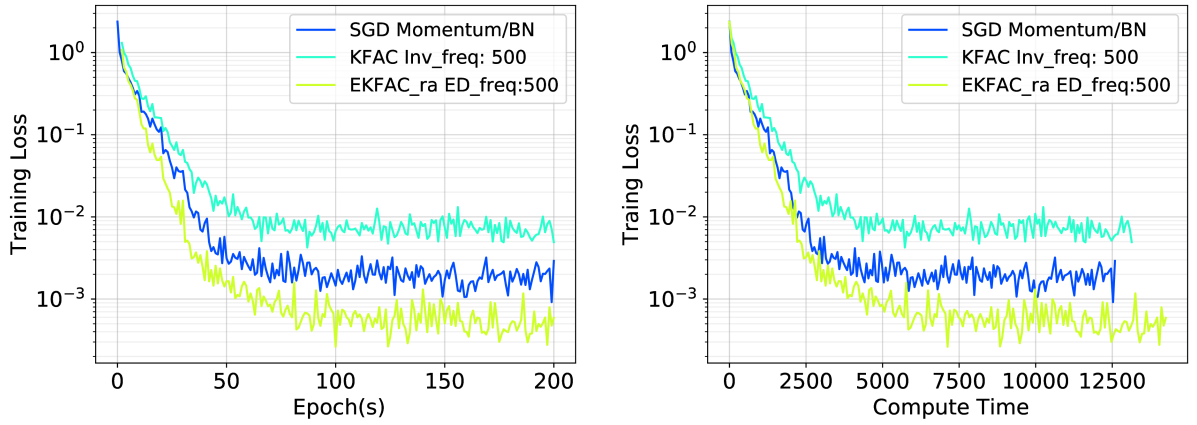
In this section we investigate the impact of a learning rate schedule on the optimisation of EKFAC. We use a similar setting than the CIFAR10 experiment, except that we decay the

learning by 2 every 20 epochs. Figure 8.11 and 8.12 show that EKFAC still highlight some optimisation benefit, relatively to the baseline, when combined with a learning rate schedule.



(a) Training loss **(b)** Wall-clock time

Figure 8.11. VGG11 on CIFAR10 using a learning rate schedule. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. In **(a)** and **(b)**, we report the performance of the hyper-parameters reaching the lowest training loss for each epoch (to highlight which optimizers perform best given a fixed epoch budget).



(a) Training loss **(b)** Wall-clock time

Figure 8.12. Resnet34 on CIFAR10 using a learning rate schedule. ED_freq (Inv_freq) corresponds to eigendecomposition (inverse) frequency. In **(a)** and **(b)**, we report the performance of the hyper-parameters reaching the lowest training loss for each epoch (to highlight which optimizers perform best given a fixed epoch budget).

Chapter 9

Prologue to the Fourth Article

9.1. Article Details

Revisiting Loss Modelling for Unstructured Pruning. César Laurent, Camille Ballas, Thomas George, Pascal Vincent and Nicolas Ballas, *arXiv preprint*, 2020.

Authors contributions. I came up with the idea of analysing the impact of the assumptions behind loss models in the context of pruning. I am responsible for the majority of the article, as well as a large part of the code and experiments. Camille Ballas contributed to the code and ran many of the complementary experiments. Thomas George helped me in the elaboration of the theoretical part of the article. Nicolas Ballas was the direct supervisor of the project, helped us scale our CIFAR10 experiments, and ran the ImageNet experiments. Pascal Vincent was the supervisor of the project.

9.2. Context

After working on EKFAC, I wanted to leverage the knowledge and code base that we build for modelling loss functions and KL Divergences with second-order Taylor expansions, and use it in a more applied setup. Such models have been used in the literature to design pruning methods since more than 30 years: The best known are Optimal Brain Damage (OBD) (LeCun et al., 1990) and Optimal Brain Surgeon (OBS) (Hassibi & Stork, 1993). These two methods rely on a second-order Taylor expansion of the loss to select which parameters to keep and which parameters prune. However, these methods typically under-perform against simpler pruning heuristics, such as Magnitude Pruning (MP) (Han et al., 2015b), which simply prunes the parameter with the smallest absolute value.

We hypothesise that one reason for the poor performance of methods based on loss modelling could be attributed to the violation of the locality and convergence assumptions behind using such models, so we propose to explore this hypothesis in this article.

9.3. Contributions

There are two sides to this article. The first one focuses on the assumptions behind using a quadratic model: We show that proper care needs to be taken to ensure these assumptions are properly satisfied. We also show the impact of violating these locality and convergence assumptions, which is something often overlooked in the literature.

The second side of the article questions the adequacy of the objective one aims to optimise when using these methods. Indeed, our experiments showed very little correlation between how well our methods were performing and the final performances after fine-tuning of the network. We believe this observation to be a good insight for the community, and that it could help develop better pruning methods.

9.4. Recent Developments

As this article has been submitted just before the time of writing this thesis, it is too early to measure its impact. We hope that our findings will help practitioners to design better pruning methods and that they will better respect the assumptions behind their pruning methods.

Chapter 10

Fourth Article: Revisiting Loss Modelling for Unstructured Pruning

Abstract. By removing parameters from deep neural networks, unstructured pruning methods aim at cutting down memory footprint and computational cost, while maintaining prediction accuracy. In order to tackle this otherwise intractable problem, many of these methods model the loss landscape using first or second order Taylor expansions to identify which parameters can be discarded. We revisit loss modelling for unstructured pruning: we show the importance of ensuring locality of the pruning steps, and systematically compare first and second order Taylor expansions. Finally, we show that better preserving the original network function does not necessarily transfer to better performing networks after fine-tuning, suggesting that only considering the impact of pruning on the loss might not be a sufficient objective to design good pruning criteria.

10.1. Introduction

Neural networks are getting bigger, requiring more and more computational resources not only for training, but also when used for inference. However, resources are sometimes limited, especially on mobile devices and low-power chips. In unstructured pruning, the goal is to remove some parameters (i.e. setting them to zeros), while still maintaining good prediction performances. This is fundamentally a combinatorial optimisation problem which is intractable even for small scale neural networks, and thus various heuristics have been developed to prune the model either before training (Lee et al., 2019b; Wang et al., 2020), during training (Louizos et al., 2017; Molchanov et al., 2017; Ding et al., 2019), or in an iterative training/fine-tuning fashion (LeCun et al., 1990; Hassibi & Stork, 1993; Han et al., 2015b; Frankle & Carbin, 2018; Renda et al., 2020).

Early pruning work Optimal Brain Damage (OBD) (LeCun et al., 1990), and later Optimal Brain Surgeon (OBS) (Hassibi & Stork, 1993), proposed to estimate the importance of each parameter by approximating the effect of removing it, using the second order term of a Taylor expansion of the loss function around converged parameters. This type of approach involves computing the Hessian, which is challenging to compute since it scales quadratically with the number of parameters in the network. Several approximations have thus been explored in the literature (LeCun et al., 1990; Hassibi & Stork, 1993; Heskes, 2000; Zeng & Urtasun, 2019; Wang et al., 2019). However, state-of-the-art unstructured pruning methods typically rely on Magnitude Pruning (MP) (Han et al., 2015b), a simple and computationally cheap criterion based on weight magnitude, that works extremely well in practice (Renda et al., 2020).

This paper revisits linear and diagonal quadratic models of the local loss landscape for unstructured pruning. In particular, since these models are *local* approximations and thus assume that pruning steps correspond to small vectors in parameter space, we propose to investigate how this locality assumption affects their performance. Moreover, we show that the *convergence* assumption behind OBD and OBS, which is overlooked and violated in current methods, can be relaxed by maintaining the gradient term in the quadratic model. Finally, to prevent having to compute second order information, we propose to compare diagonal quadratic models to simpler linear models.

While our empirical study demonstrates that pruning criteria based on linear and quadratic loss models are good at preserving the training loss, it also shows that this benefit does not necessarily transfer to better networks after fine-tuning, suggesting that preserving the loss might not be the best objective to optimise for. Our contributions can be summarised as follows:

(1) We present pruning criteria based on both linear and diagonal quadratic models of the loss, and show how they compare at preserving training loss compared to OBD and MP.

(2) We study two strategies to better enforce locality in the pruning steps, pruning in several stages and regularising the step size, and show how they improve the quality of the criteria.

(3) We show that using pruning criteria that are better at preserving the loss does not necessarily transfer to better fine-tuned networks, raising questions about the adequacy of such criteria.

10.2. Background: Unstructured Pruning

10.2.1. Unstructured Pruning Problem Formulation

For a given architecture, neural networks are a family of functions $f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}$ from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$, where $\boldsymbol{\theta} \in \mathbb{R}^D$ is the vector that contains all the parameters of the network. Neural networks are usually trained by seeking parameters $\boldsymbol{\theta}$ that minimise the empirical risk $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_i \ell\left(f_{\boldsymbol{\theta}}\left(x_i\right), t_i\right)$ of a loss function ℓ on a training dataset $\mathcal{D} = \{(x_i,t_i)\}_{1\leq i\leq N}$, composed of N (example, target) pairs.

The goal of unstructured pruning is to find a step $\Delta\boldsymbol{\theta}$ to add to the current parameters $\boldsymbol{\theta}$ such that $\|\boldsymbol{\theta}+\Delta\boldsymbol{\theta}\|_0 = (1-\kappa)D$, i.e. the parameter vector after pruning is of desired sparsity $\kappa \in [0,1]$. While doing so, the performance of the pruned network should be maintained, so $\mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$ should not differ much from $\mathcal{L}(\boldsymbol{\theta})$. Unstructured pruning thus amounts to the following minimisation problem:

$$\underset{\Delta\boldsymbol{\theta}}{\text{minimise}} \quad \Delta\mathcal{L}(\boldsymbol{\theta},\Delta\boldsymbol{\theta}) \stackrel{\text{def}}{=} \left|\mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) - \mathcal{L}(\boldsymbol{\theta})\right| \qquad \text{s.t.} \quad \|\boldsymbol{\theta}+\Delta\boldsymbol{\theta}\|_0 = (1-\kappa)D \tag{10.1}$$

Directly solving this problem would require evaluating $\mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$ for all possible values of $\Delta\boldsymbol{\theta}$, which is prohibitively expensive, so one needs to rely on heuristics to find good solutions.

10.2.2. Optimal Brain Damage Criterion

Optimal Brain Damage (OBD) (LeCun et al., 1990) proposes to use a quadratic modelling of $\mathcal{L}(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$, leading to the following approximation of $\Delta\mathcal{L}(\boldsymbol{\theta},\Delta\boldsymbol{\theta})$:

$$\Delta\mathcal{L}^{QM}(\boldsymbol{\theta},\Delta\boldsymbol{\theta}) = \left|\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}^\top \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^\top\mathbf{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}\right| \tag{10.2}$$

where $\mathbf{H}(\boldsymbol{\theta})$ is the Hessian of $\mathcal{L}(\boldsymbol{\theta})$. $\mathbf{H}(\boldsymbol{\theta})$ being intractable, even for small-scale networks, its Generalised Gauss-Newton approximation $\mathbf{G}(\boldsymbol{\theta})$ (Schraudolph, 2002) is used in practice, as detailed in Appendix 10.A.[1] Then, two more approximations are made: first, it assumes the training of the network has converged, thus the gradient of the loss with respect to $\boldsymbol{\theta}$ is 0, which makes the linear term vanish. Then, it neglects the interactions between parameters, which corresponds to a diagonal approximation of $\mathbf{G}(\boldsymbol{\theta})$, leading to the following model:

$$\Delta\mathcal{L}^{OBD}(\boldsymbol{\theta},\Delta\boldsymbol{\theta}_k) \approx \frac{1}{2}\mathbf{G}_{kk}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}_k^2 \qquad \Rightarrow \qquad s_k^{\text{OBD}} = \frac{1}{2}\mathbf{G}_{kk}(\boldsymbol{\theta})\boldsymbol{\theta}_k^2 \tag{10.3}$$

1. Although LeCun et al. (1990) uses $\mathbf{H}(\boldsymbol{\theta})$ in the equations of OBD, it is actually $\mathbf{G}(\boldsymbol{\theta})$ which was used in practice (LeCun, 2007).

s_k^{OBD} is the *saliency* of each parameter, estimating how much the loss will change if that parameter is pruned, so if $\Delta\boldsymbol{\theta}_k = -\boldsymbol{\theta}_k$. Parameters can thus be ranked by order of importance, and the ones with the smallest saliencies (i.e. the least influence on the loss) are pruned, while the ones with the biggest saliencies are kept unchanged. This can be interpreted as finding and applying a binary mask $\mathbf{m} \in \{0,1\}^D$ to the parameters such that $\boldsymbol{\theta} + \Delta\boldsymbol{\theta} = \boldsymbol{\theta} \odot \mathbf{m}$, where $\odot$ is the element-wise product.

10.2.3. Magnitude Pruning Criterion

Magnitude Pruning (MP) (Han et al., 2015b), is a popular pruning criterion in which the saliency is simply based on the norm of the parameter:

$$s_k^{\text{MP}} = \boldsymbol{\theta}_k^2 \tag{10.4}$$

Despite its simplicity, MP works extremely well in practice (Gale et al., 2019), and is used in current state-of-the-art methods (Renda et al., 2020). We use global MP as baseline in all our experiments.

10.2.4. Optimal Brain Surgeon

Optimal Brain Surgeon (OBS) (Hassibi & Stork, 1993) also relies on the quadratic model in Equation 10.2 to solve the minimisation problem given in Equation 10.1, but uses the Lagrangian formulation to include the constraint to the solution of the minimisation problem. Since OBS requires to compute the inverse of $\mathbf{H}(\boldsymbol{\theta})$, several approximations have been explored in the literature, including diagonal, as in the original OBS, Kronecker-factored (Martens & Grosse, 2015) as in ML-Prune (Zeng & Urtasun, 2019), or diagonal, but in an Kronecker-factored Eigenbasis (George et al., 2018), as in EigenDamage (Wang et al., 2019). While we use OBD in our demonstrations and experimental setup, everything presented in this paper can also be used in OBS-based methods. We leave that for future work.

10.3. Revisiting Loss Modelling for Unstructured Pruning

In this work, we investigate linear and diagonal quadratic models of the loss function and their performance when used for pruning neural networks. In our empirical study, we aim at answering the following questions:

(1) How do criteria based on weight magnitude, or linear or quadratic models compare at preserving training loss (i.e. at solving the minimisation problem in Equation 10.1)?

(2) How does the locality assumption behind criteria based on linear and quadratic models affect their performances?

(3) Do pruning criteria that are better at preserving the loss lead to better fine-tuned networks?

We now describe the linear and quadratic models we use, as well as the strategies to enforce locality of the pruning steps.

10.3.1. Linear and Quadratic Models

In current training strategies, regularisation techniques such as early stopping or dropout (Srivastava et al., 2014) are often used to counteract overfitting. In these setups, there is no reason to assume that the training has converged, implying that the linear term in the Taylor expansion should not be neglected. Thus, one can build a pruning criterion similar to OBD that includes the gradient term in the quadratic model from Equation 10.2, leading to the following saliencies: [2]

$$\Delta\mathcal{L}^{QM}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}_k) \approx \left| \frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_k}^\top \Delta\boldsymbol{\theta}_k + \frac{1}{2}\mathbf{G}_{kk}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}_k^2 \right| \Rightarrow s_k^{\mathrm{QM}} = \left| -\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_k}\boldsymbol{\theta}_k + \frac{1}{2}\mathbf{G}_{kk}(\boldsymbol{\theta})\boldsymbol{\theta}_k^2 \right| \tag{10.5}$$

Recall the constraint $\Delta\boldsymbol{\theta}_k \in \{-\boldsymbol{\theta}_k, 0\}$, hence the saliencies. This criterion generalises OBD for networks that are not at convergence, and provides similar saliencies for networks that have converged.

To avoid the computational cost associated with computing second order information, which is prohibitive for large scale neural networks, one can use a simpler linear model (LM) instead of a quadratic one to approximate $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$, leading to the following approximation and saliencies:

$$\Delta\mathcal{L}^{LM}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = \left| \frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}^\top \Delta\boldsymbol{\theta} \right| \quad \Rightarrow \quad s_k^{\mathrm{LM}} = \left| \frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}_k}\boldsymbol{\theta}_k \right| \tag{10.6}$$

The saliencies of the linear model are very related to the criterion used in Single-shot Network Pruning (Lee et al., 2019b), as demonstrated by Wang et al. (2020).

2. Note that this idea has been explored for OBS (Singh & Alistarh, 2020), as well as for structured pruning (Molchanov et al., 2019)

10.3.2. Enforcing Locality

One important point to keep in mind is that linear and quadratic models (whether diagonal or not) are *local* approximations, and are generally only faithful in a small neighbourhood of the current parameters. Explicitly showing the terms that are neglected, we have:

$$\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = \Delta\mathcal{L}^{LM}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) + \mathcal{O}(\|\Delta\boldsymbol{\theta}\|_2^2) = \Delta\mathcal{L}^{QM}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) + \mathcal{O}(\|\Delta\boldsymbol{\theta}\|_2^3) \tag{10.7}$$

So when approximating $\Delta\mathcal{L}$ with $\Delta\mathcal{L}^{LM}$ we neglect the terms in $\mathcal{O}(\|\Delta\boldsymbol{\theta}\|_2^2)$, and when approximating $\Delta\mathcal{L}$ with $\Delta\mathcal{L}^{QM}$ we neglect the terms in $\mathcal{O}(\|\Delta\boldsymbol{\theta}\|_2^3)$. Both approximations are thus only valid in a small neighbourhood of $\boldsymbol{\theta}$, and are extremely likely to be wrong when $\|\Delta\boldsymbol{\theta}\|_2$ is large. We list here different tricks to prevent this from happening.

Performing the Pruning in several Stages. $\|\Delta\boldsymbol{\theta}\|_2$ can be large when a large portion of the parameters is pruned at once. An easy fix typically used to mitigate this issue is to perform the pruning in several stages, re-estimating the model at each stage. The number of stages, which we denote by π, is typically overlooked (e.g. both Zeng & Urtasun (2019) and Wang et al. (2019) use only 6 stages of pruning). Our experiments, in upcoming Section 10.5, show that it has a drastic impact on the performances. Note that, without fine-tuning phases between the different pruning stages, this strategy violates the *convergence* assumption behind OBD and OBS, since after the first stage of pruning the network is no more at convergence.

The sparsity at each stage can be increased following either a linear schedule, where each step prunes the same number of parameter, or an exponential schedule, where the number of parameters pruned at each stage gets smaller and smaller. The later is typically used in the literature (Zeng & Urtasun, 2019; Wang et al., 2019; Frankle & Carbin, 2018; Renda et al., 2020). We compare them in Section 10.5.

Constraining the Step Size. As is often done when using quadratic models (e.g. Nocedal & Wright (2006)), one can penalise the model when it decides to take steps that are too large, in order to stay in a region where we can trust the model. This can be done by simply adding the norm penalty $\frac{\lambda}{2}\|\Delta\boldsymbol{\theta}_k\|_2^2$ to the saliencies computed by any criterion (Equations 10.3, 10.5 or 10.6), where λ is a hyper-parameter that controls the strength of the constraint: a small value of λ leaves the saliencies unchanged, and a large value of λ transforms the pruning criterion into MP (Equation 10.4).

Other Considerations. $\|\Delta\boldsymbol{\theta}\|_2$ can be large if $\boldsymbol{\theta}$ is large itself. This is dependent on the training procedure of the network, but can be easily mitigated by constraining the norm of the weights, which can be done using L_2 regularisation or weight decay. Since nowadays

weight decay is almost systematically used by default when training networks (e.g. He et al. (2016b); Xie et al. (2017); Devlin et al. (2018)), we do not investigate this further.

10.4. Methodology

We follow the main recommendations from Blalock et al. (2020). For fair comparison between criteria, all experiments are from our own PyTorch (Paszke et al., 2017) re-implementation, and ran on V100 GPUs.We use 5 different random seeds, and both mean and standard deviations are reported. We experiment with a MLP on MNIST, and with both VGG11 (Simonyan & Zisserman, 2015) and a pre-activation residual network 18 (He et al., 2016b) on CIFAR10 (Krizhevsky & Hinton, 2009), to have variability in architectures, while using networks with good performance to number of parameters ratio. We further validate our findings on ImageNet (Deng et al., 2009) using a residual network 50 (He et al., 2016a). Although MNIST is not considered a good benchmark for pruning (Blalock et al., 2020), it can still be used to compare the ability of different criteria to solve the minimisation problem in Equation 10.1. See Appendix 10.B for details about splits, data augmentation and hyper-parameters.

Pruning Framework.

Algorithm 2 presents the pruning framework used in this work: we first train the network, then perform several stages of pruning, and finally perform a single phase of fine-tuning, using the same hyper-parameters as for the original training. *Global* pruning is used for all the criteria. Note that because of their convergence assumption, OBD and OBS advocate for fine-tuning after each stage of pruning. Since LM and QM are not based on this assumption, they should perform well in this proposed framework. While the fine tuning-phase would require hyper-parameters optimisation, Renda et al. (2020) showed that using the same ones as for the original training usually leads to good results. The hyper-parameters used in our experiments are provided in Appendix 10.B.

Performance Metrics. The performances of the pruning criteria are measured using two metrics: First, we use $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = |\mathcal{L}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) - \mathcal{L}(\boldsymbol{\theta})|$, which is the quantity that the pruning criteria are designed to minimise (recall Equation 10.1). Second, we use the validation error gap before/after fine-tuning, which is the metric we ultimately care about when designing pruning methods.

Algorithm 2 Pruning Framework

Require: Network $f_{\boldsymbol{\theta}}$ with $\boldsymbol{\theta} \in \mathbb{R}^D$, dataset $\mathcal{D}$, number of pruning iterations π, and sparsity κ.

$f_{\boldsymbol{\theta}} \leftarrow \text{Training}(f_{\boldsymbol{\theta}}, \mathcal{D})$
$\kappa_0 \leftarrow 0$
$\mathbf{m} \leftarrow \mathbf{1}^D$
for $i = 1$ to π **do**
    $\kappa_i \leftarrow \kappa_{i-1} + \frac{(\kappa - \kappa_0)}{\pi}$ **or** $\kappa_i \leftarrow \kappa_{i-1} + (\kappa - \kappa_0)^{i/\pi}$ ▷ Compute sparsity for iteration i
    $\boldsymbol{s} \leftarrow \text{Saliencies}(f_{\boldsymbol{\theta} \odot \mathbf{m}}, \mathcal{D})$ ▷ Compute saliencies (Equation 10.3, 10.4, 10.5 or 10.6).
    $\mathbf{m}[\text{argsort}(\boldsymbol{s})[: \kappa_i D]] \leftarrow 0$ ▷ Mask the parameters with smallest saliencies.
$f_{\boldsymbol{\theta} \odot \mathbf{m}} \leftarrow \text{Training}(f_{\boldsymbol{\theta} \odot \mathbf{m}}, \mathcal{D})$ ▷ Optional fine-tuning
return $f_{\boldsymbol{\theta} \odot \mathbf{m}}, \mathbf{m}$

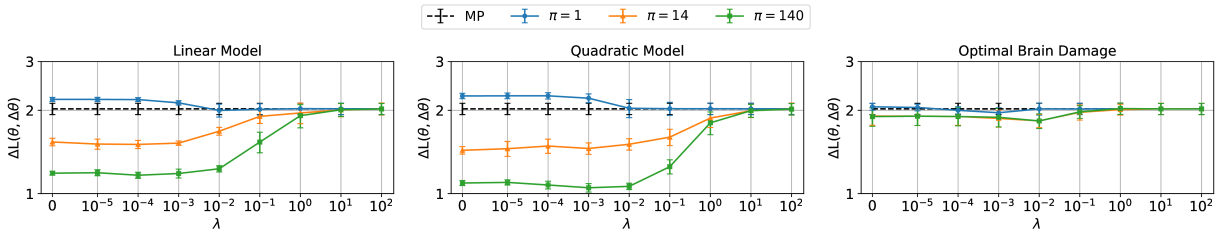
10.5. Performances before Fine-tuning

We evaluate the impact of enforcing locality in the LM, QM and OBS criteria. For each criterion, Figure 10.1 reports $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ as a function of λ, for different number of pruning stages π, using the exponential pruning schedule, and Figure 10.5 in Appendix show the same results for the linear pruning schedule. A typical usage of these criteria would be with a regularisation strength $\lambda = 0$ and a number of pruning stages $\pi \approx 1$. MP, the baseline, which is invariant to both λ and π, is also reported in dashed black. For reference, the networks reached a validation error rate before pruning of 1.47 ± 0.04 % for the MLP, 10.16 ± 0.29 % for VGG11 and 4.87 ± 0.04 % for the PreActResNet18.

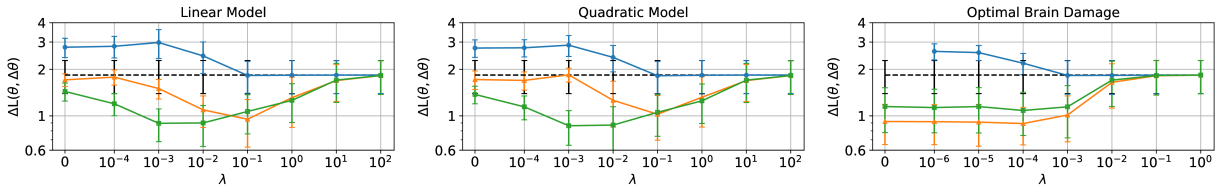
10.5.1. Impact of the Assumptions behind the Different Criteria

Locality Assumption. Figure 10.1 shows that increasing the number of pruning stages can drastically reduce $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ when using LM, QM and OBS criteria. It demonstrates the importance of applying local steps when pruning. Constraining the steps size through $\frac{\lambda}{2} \|\Delta\boldsymbol{\theta}_k\|_2^2$ can also reduce $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$, on CIFAR10 in particular. The trend, however, is less pronounced on MNIST. We hypothesise that it is due to the pruning step size: the MLP contains 260k parameters, vs 9.7M for VGG11, so the number of parameters pruned at each stage in VGG11 is still large, even with $\pi = 140$. This translates to a bigger $\|\Delta\boldsymbol{\theta}\|_2$ that needs to be controlled by the regularisation constraint.

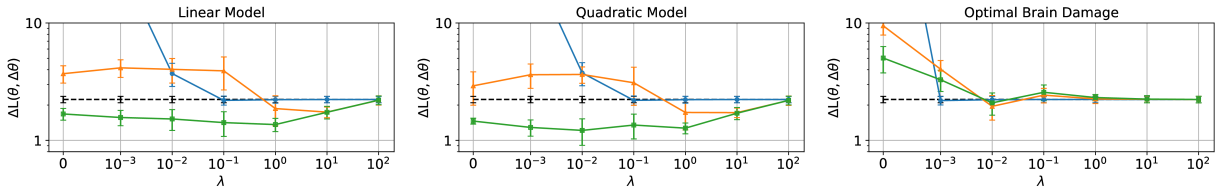
Convergence Assumption. When performing the pruning in several stages, we also observe that LM and QM can reach better performances than OBD. Without retraining phases between pruning stages, we violate the convergence assumption of OBD. This is however not the case for LM and QM, since they are not based on this assumption. Note



(a) MLP on MNIST with 98.8% sparsity.



(b) VGG11 on CIFAR10 with 95.6% sparsity.



(c) PreActResNet18 on CIFAR10 with 95.6% sparsity.

Figure 10.1. $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ for different number of pruning stages π, as a function of λ, the step size constraint strength, using either (left) LM, (middle) QM or (right) OBD criteria. MP, which is invariant to λ and to the number of pruning stages, is displayed in dashed black. The curves are the mean and the error bars the standard deviation over 5 random seeds. OBD with $\pi = 1$ and $\lambda = 0$ diverged for all of the 5 seeds. Increasing the number of pruning stages drastically reduces $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. A $\lambda > 0$ can also help improving performances. Figure 10.6 in Appendix contains the same plots, but displaying the validation gap before/after pruning.

that OBD still works reasonably well on VGG11. This could be related to the depth of VGG11: VGG11 is deeper than the MLP, but not equipped with residual connections like the PreActResNet18.

10.5.2. Loss-preservation Capabilities of the Different Criteria

Table 10.1 contains the best $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ for each of the networks and pruning criteria. Our main observation is that the criteria that model the loss (LM and QM in particular) are better at preserving the loss than MP. Similarly to Table 10.1, Table 10.3 in Appendix contains the best validation error gap before/after pruning, where we can observe similar tendencies.

Table 10.1. Summary of the best $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ across values of λ for different networks and pruning criteria, with $\pi = 140$. QM achieves better loss-preservation than other criteria. OBD performs worse than QM, since we violate its convergence assumption when pruning in several stages.

Network	$\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$			
	MP	**OBD**	**LM**	**QM**
MLP on MNIST	2.02 ± 0.10	1.83 ± 0.11	1.17 ± 0.03	$\mathbf{1.05 \pm 0.04}$
VGG11 on CIFAR10	1.84 ± 0.44	$\mathbf{0.89 \pm 0.24}$	$\mathbf{0.90 \pm 0.21}$	$\mathbf{0.86 \pm 0.22}$
PreActResNet18 on CIFAR10	2.23 ± 0.14	1.95 ± 0.46	1.36 ± 0.18	$\mathbf{1.22 \pm 0.31}$

10.5.3. Linear vs Exponential Pruning Schedule

Figure 10.2 compares the impact of $\|\Delta\boldsymbol{\theta}\|_2$ and reports the training error gap when pruning VGG11 on CIFAR10 in several stages, using either the linear or the exponential pruning schedule. We also compare against one-shot pruning, as reference. The exponential schedule allows to maintain a more constant $\|\Delta\boldsymbol{\theta}\|_2$ throughout the pruning procedure, which limits the maximum size of $\|\Delta\boldsymbol{\theta}\|_2$, and thus better satisfies the locality assumption.

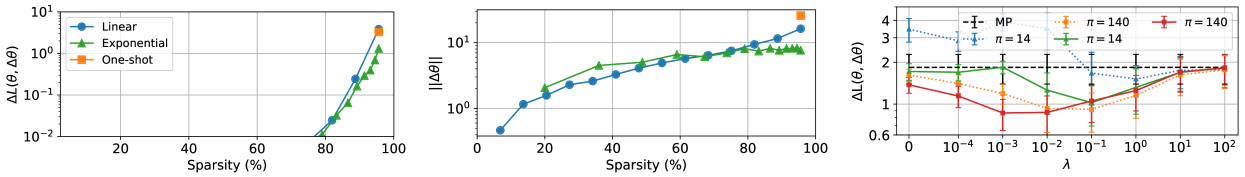


Figure 10.2. Linear vs exponential schedule using QM on VGG11. Left: $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ vs sparsity, zoomed on the end. Markers denote the 14 pruning stages. Middle: $\|\Delta\boldsymbol{\theta}\|_2$ at each stage. Right: Same as Figure 10.1, but comparing exponential (solid) and linear (dotted) schedules at 95.6% sparsity, with $\pi \in \{14, 140\}$. We get smaller $\|\Delta\boldsymbol{\theta}\|_2$ per pruning stage when using exponential instead of linear schedule, resulting in a smaller $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. It is advantageous to use that schedule when the pruning budged is limited, i.e. when π is small. This advantage vanishes for larger values of π.

10.6. Performances after Fine-tuning

We now fine-tune the pruned networks using the same hyper-parameters and number of epochs than for the original training. Table 10.2 shows the validation error gap between the non-pruned networks and the pruned networks after fine-tuning, for all considered criteria. LM performs better than MP on both the MLP and VGG11 (0.5% difference), but all criteria perform similarly on the PreActResNet18. These results are consistent with the observations of Blalock et al. (2020). As reference, global random pruning resulted in validation error rate

of 47.18 ± 6.8 % for the MLP, and resulted in non-retrainable networks on CIFAR10 (with 90 % error rate).

Table 10.2. Best validation error gap of the fine-tuned networks (lower is better), for different pruning criteria, across values of λ and π. LM is better than MP on the MLP and VGG11. All the methods reach similar levels of performance on the PreActResNet18.

Network	Gap of Validation Error (%)			
	MP	OBD	LM	QM
MLP on MNIST	2.4 ± 0.3	2.0 ± 0.1	$\mathbf{1.9 \pm 0.3}$	$\mathbf{1.9 \pm 0.2}$
VGG11 on CIFAR10	0.2 ± 0.2	-0.1 ± 0.2	$\mathbf{-0.3 \pm 0.1}$	-0.1 ± 0.1
PreActResNet18 on CIFAR10	0.2 ± 0.2	0.2 ± 0.2	$\mathbf{0.1 \pm 0.1}$	0.2 ± 0.2

10.6.1. Correlation between Loss-preservation and Performances after Fine-tuning

An important observation is that the hyper-parameters λ and π that give the best performing criteria in terms of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ in Table 10.1 are not the same as the ones that give the best performing criteria after fine-tuning in Table 10.2. We display in Figure 10.3 scatter plots of all the experiments we ran, to observe how well loss-preservation correlates with performance after fine-tuning.

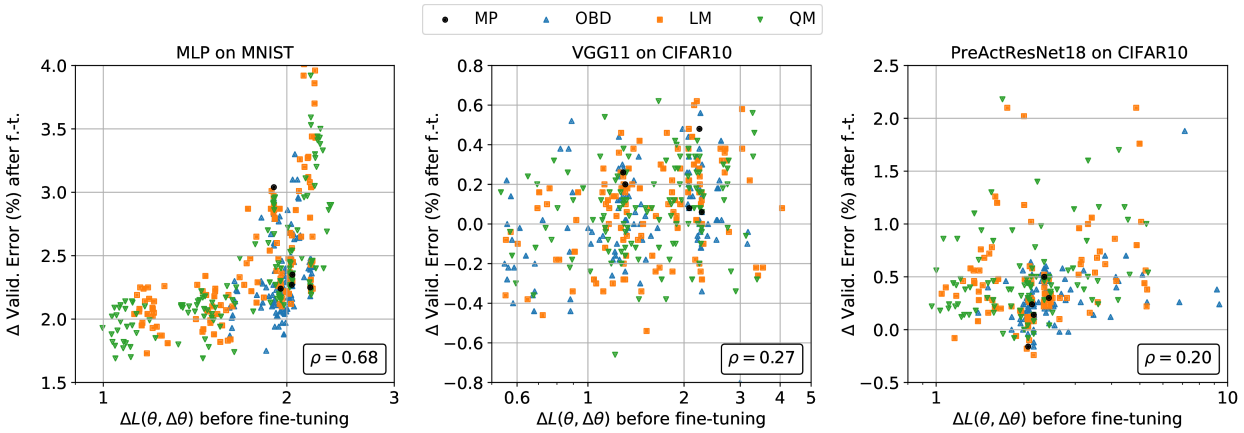


Figure 10.3. Gap of validation error after fine-tuning as a function of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. Each point is one experiment, i.e. one one random seed, one π and one λ. ρ is the Spearman's rank correlation coefficient computed on all the data points. Except for the MLP on MNIST, there is only a weak correlation between $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ and the gap of validation after fine-tuning. Thus, the performance after pruning cannot be explained solely by the loss-preserving abilities of the pruning criteria.

Quite surprisingly, although we are able to obtain networks with smaller $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$, and thus better performing networks right after pruning, the performances after fine-tuning do not

correlate significantly with the gap. Except for the MLP on MNIST, whose Spearman's rank correlation coefficient is $\rho = 0.67$, there is only a weak correlation between $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ and the validation error gap after fine-tuning ($\rho = 0.27$ for VGG11 and $\rho = 0.20$ for PreActResNet18). Figure 10.10 in Appendix contains the same scatter plots, but showing $\mathcal{L}(\boldsymbol{\theta} \odot \mathbf{m})$ after fine-tuning instead of the validation error gap, and similar trends can be observed. Figure 10.11, also in Appendix, shows similar scatter plots, but for different sparsity levels on VGG11. Finally, Figure 10.9 in Appendix contains the same scatter plots but displaying the validation error gap before fine-tuning versus the validation error gap before fine-tuning.

To verify that these observations are not due to a specific choice of fine-tuning hyper-parameters, we perform a hyper-parameter grid search and report similar results in Appendix 10.C.2. Also, we show in Figure 10.15 in Appendix 10.C.3 the fine-tuning curves of networks pruned using MP and our best QM criteria. We observe that, except for MNIST, the difference in training loss right after pruning disappears after only one epoch of fine-tuning, erasing the advantage of QM over MP.

10.6.2. Discussion

These results highlight an important issue: minimising $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$, no matter what model is used, might be used to design better pruning criteria, but it does not necessarily transfer to a better pruning method when fine-tuning is involved. The performance after fine-tuning cannot be explained solely by the local loss-preserving abilities of the criteria, and other mechanisms might be at play. Thus, the effect of fine-tuning should also be taken into account when designing pruning criteria.

For instance, Lee et al. (2019a) and Wang et al. (2020) proposed different heuristics to take into account gradient propagation in the context of *foresight* pruning, i.e. pruning untrained networks right after initialisation. Wang et al. (2020) argues that minimising $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ in that context makes little sense, since the network is producing random predictions. In Appendix 10.C.4 we compare our results to two pruning methods based on preserving the gradient flow, GraSP (Wang et al., 2020) and SynFlow (Tanaka et al., 2020), and show that they comply with our observations above.

Finally, several recent articles are looking further into the impact of various pruning criteria on subsequent training or fine-tuning (Lubana & Dick, 2020; Evci et al., 2020; Frankle et al., 2020).

10.7. Scaling up to ImageNet

To investigate whether our observations also hold on larger datasets, we perform similar experiments with LM, QM and OBD on the ResNet50 on ImageNet. Before pruning, the network reached 76.41% validation accuracy. Figure 10.4 presents results at 70 % sparsity, in a similar fashion as Figure 10.1 and Figure 10.3. We observe a similar trend: The best loss-preserving models are not necessarily the best models after fine-tuning. See Appendix 10.B for the detailed experimental setting, and see Figure 10.17 for results at 90 % sparsity.

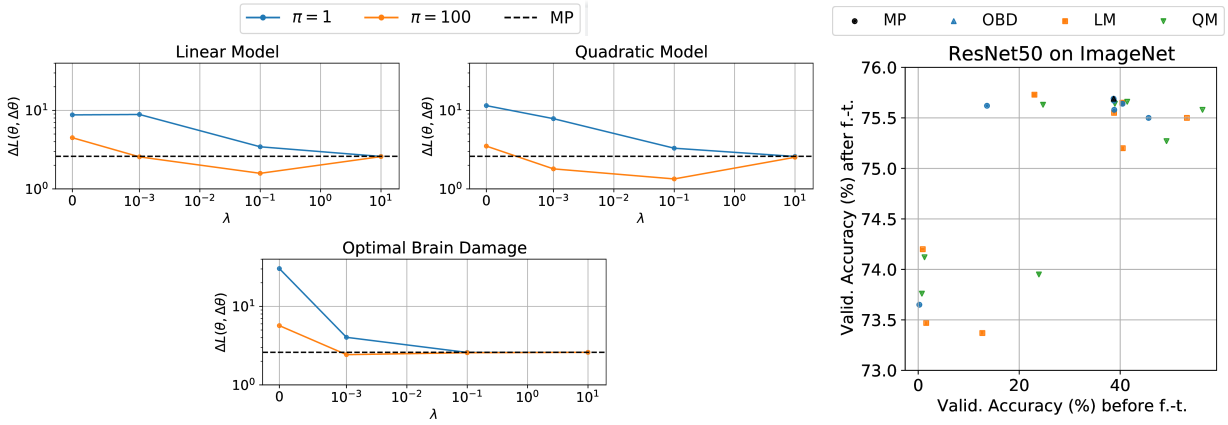


Figure 10.4. Same as Figure 10.1 and Figure 10.3, for the ResNet50 on ImageNet, with a sparsity of 70 %. Increasing the number of pruning stages and constraining the step size reduce $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. However, the best-loss preserving criteria, which maximise the validation accuracy right after pruning, do not produce better networks after fine-tuning. They perform similarly if their validation accuracy after pruning is $> 20\%$. Criteria that outperform MP right after pruning do not achieve better performance after fine-tuning.

10.8. Conclusion

In this paper, we revisited loss modelling for unstructured pruning. We showed that keeping the gradient term in the diagonal quadratic model allows to relax the convergence assumption behind OBS and OBD. We also showed the importance of locality when using loss models for pruning: increasing the number of pruning stages and constraining the step size are two improvements that produce better loss-preserving pruning criteria and that should be added to the recommendation list of Blalock et al. (2020). Finally we observed that the loss right after pruning does not always correlate with the performances after fine-tuning, suggesting that a better loss before fine-tuning is not solely responsible for the performances after fine-tuning. Thus, future research should focus on ways to model the actual effect of subsequent fine-tuning when designing pruning criteria.

Acknowledgements

We thank Facebook for computational and financial resources. This research was also enabled in part by support provided by Calcul Québec and Compute Canada, and Science Foundation Ireland (SFI) under Grant Number 12/RC/2289_P2 and 16/SP/3804 (Insight Centre for Data Analytics). We also wish to thank Aristide Baratin for insightful discussions.

10.A. Generalised Gauss-Newton

Having to compute $\mathbf{H}(\boldsymbol{\theta})$ is an obvious drawback of quadratic models, and thus a common first step is to approximate $\mathbf{H}(\boldsymbol{\theta})$ using the Generalised Gauss-Newton approximation (Schraudolph, 2002):

$$\mathbf{H}(\boldsymbol{\theta}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} \frac{\partial f_{\boldsymbol{\theta}}(x_i)}{\partial \boldsymbol{\theta}}^{\top} \nabla^2_{u=f_{\boldsymbol{\theta}}(x_i)} \ell(u,t_i) \frac{\partial f_{\boldsymbol{\theta}}(x_i)}{\partial \boldsymbol{\theta}}}_{\mathbf{G}(\boldsymbol{\theta}),\ \text{the Generalised Gauss-Newton}} + \underbrace{\sum_{k}^{K} \frac{\partial \ell(u,t_i)}{\partial u_k}\Big|_{u=f_{\boldsymbol{\theta}}(x_i)} \frac{\partial^2 f_{\boldsymbol{\theta}}(x_i)_k}{\partial \boldsymbol{\theta}^2}}_{\approx 0} \tag{10.8}$$

$$\approx \mathbf{G}(\boldsymbol{\theta}) \tag{10.9}$$

where K is the number of outputs of the network. $\mathbf{G}(\boldsymbol{\theta})$ has the advantage of being easier to compute and is also positive semi-definite by construction.

10.B. Details on the Experimental Setup

10.B.1. Setup

Datasets. We use the MNIST dataset (LeCun et al., 1998), and hold-out 10000 examples randomly sampled from the training set for validation. We also use CIFAR10 (Krizhevsky & Hinton, 2009), where the last 5000 examples of the training set are used for validation, and we apply standard data augmentation (random cropping and flipping, as in He et al. (2016b)) during training phases. For ImageNet (Deng et al., 2009), we follow the experimental setting of Goyal et al. (2017).

Network Architectures. On MNIST, we use a MLP of dimensions 784-300-100-10, with Tanh activation functions. On CIFAR10, we use both: a VGG11 (Simonyan & Zisserman, 2015), equipped with ReLUs (Nair & Hinton, 2010), but no Batch Normalisation (Ioffe & Szegedy, 2015); and the PreActResNet18, which is the 18-layer pre-activation variant of residual networks (He et al., 2016b). MLP leverages Glorot & Bengio (2010) as initialisation

while the the weights of VGG11 and PreActResNet18 are initialised following He et al. (2015), and the biases are initialised to 0. On ImageNet (Deng et al., 2009), we use a ResNet-50 (He et al., 2016a) with Batch Normalisation, and follow the initialisation strategy described in (Goyal et al., 2017).

10.B.2. Experiments

For the MNIST and CIFAR10 experiments, the network is first trained for a fixed number of epochs, using early stopping on the validation set to select the best performing network.The hyper-parameters used for training are selected via grid search (before even considering pruning). Then we prune a large fraction of the parameters. For OBD, LM and QM, we randomly select, at each iteration of pruning, 1000 examples (10 mini-batches) from the training set to compute the gradients and second order terms of the models. [3] Finally, we retrain the network using exactly the same hyper-parameters as for the initial training.

For ImageNet, we uses the exact same hyper-parameters than Goyal et al. (2017).

MLP on MNIST. We train the network for 400 epochs, using SGD with learning rate of 0.01, momentum factor of 0.9, l2 regularisation of 0.0005 and a mini-batch size of 100. We prune 98.85% of the parameters.

VGG11 on CIFAR10. We train the network for 300 epoch, using SGD with a learning rate of 0.01, momentum factor of 0.9, a l2 regularisation of 0.0005 and a mini-batch size of 100. The learning rate is divided by 10 every 60 epochs. We prune 95.6% of the parameters.

PreActResNet18 on CIFAR10. We train the network for 200 epochs, using SGD with a learning rate of 0.1, momentum factor of 0.9, a l2 regularisation of 0.0005 and a mini-batch size of 100. The learning rate is divided by 10 every 70 epochs. We prune 95.6% of the parameters.

ResNet50 on ImageNet. For ImageNet, we train a ResNet50 using 8 V100 GPUs. The total mini-batch size is 256, and we train our baseline network for 90 epochs. The learning rate schedule is identical to Goyal et al. (2017): a linear warm-up in the first 5 epochs and decay by a factor of 10 at epochs 30, 60 and 80. We then prune 70% of the parameters. After pruning, we fine-tune the models for 90 epochs using a learning rate of $1e^{-3}$. For LM, QM, OBD, we investigates the following hyper-parameter values: $\pi \in \{1, 100\}$, $\lambda \in \{1e^{-3}, 1e^{-1}, 0, 10,\}$. 1600

3. Using 1000 examples or the whole training set made no difference in our experiments. Using less examples started to degrade the performances, which concord with the observations of Lee et al. (2019b)

examples are used to compute the first and second order terms of the linear and quadratic models.

10.C. Supplementary Results

10.C.1. Performances before Fine-tuning

Validation error Table. Table 10.3 is the same as Table 10.1, but containing the best validation error gap before/after pruning instead of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. We can observe a similar trend as in Table 10.1: LM and QM give better performances than MP, and OBD performs poorly, since the convergence assumption is not respected.

Table 10.3. Best validation error gap before/after pruning for different networks and pruning criteria.

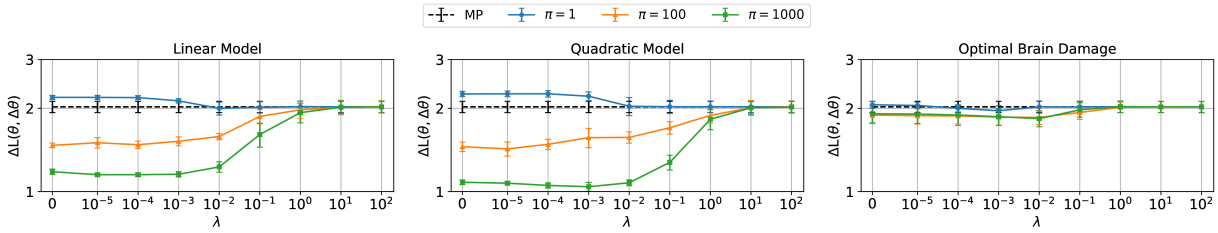
Network	Gap of Validation Error (%)			
	MP	**OBD**	**LM**	**QM**
MLP on MNIST	72.09 ± 3.72	64.89 ± 5.74	**16.35 ± 0.77**	**15.22 ± 0.62**
VGG11 on CIFAR10	56.19 ± 17.9	18.84 ± 5.54	**5.89 ± 1.52**	**5.92 ± 2.14**
PreActResNet18 on CIFAR10	74.13 ± 4.59	49.08 ± 8.18	26.79 ± 8.61	**21.48 ± 5.96**

Linear pruning schedule. Figure 10.5 contains the same experiments than Figure 10.5, but using the linear schedule instead of the exponential one. There is a drastic difference in performances: One need roughly 10x more stages of pruning with the linear schedule to reach the training gap of the exponential schedule.

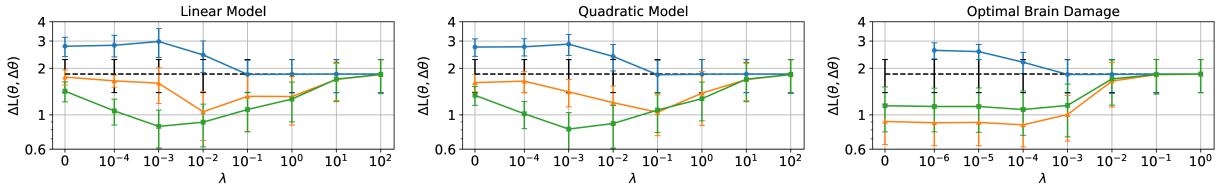
10.C.2. Performances after Fine-tuning

Validation error figures. Figures 10.8 and 10.7 contain the same experiments than Figure 10.5, but displaying the validation error gap, for linear and exponential schedules, respectively. For completeness, Figure 10.6 shows the validation error gap before fine-tuning.

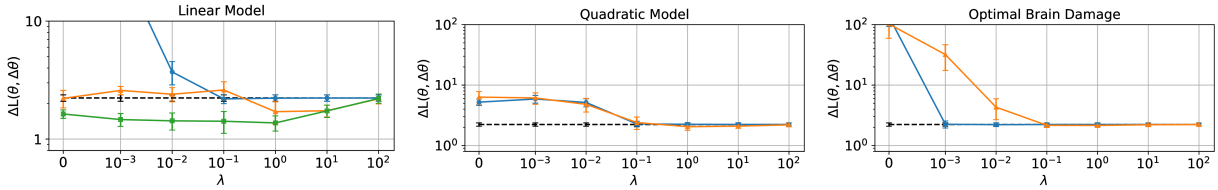
Validation gap before and after fine-tuning. Figure 10.9 is the same as Figure 10.3, but showing the validation error gap after fine-tuning as a function of the validation error gap before fine-tuning. As for Figure 10.3, we do not observe much correlation between the validation error before and after the fine-tuning.



(a) MLP on MNIST with 98.8% sparsity.



(b) VGG11 on CIFAR10 with 95.6% sparsity.



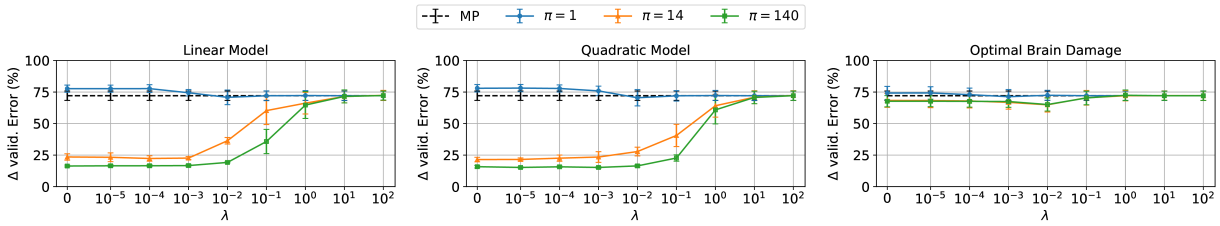
(c) PreActResNet18 on CIFAR10 with 95.6% sparsity.

Figure 10.5. Same as Figure 10.1, but using equally spaced pruning steps. Note the difference in number of pruning iterations.

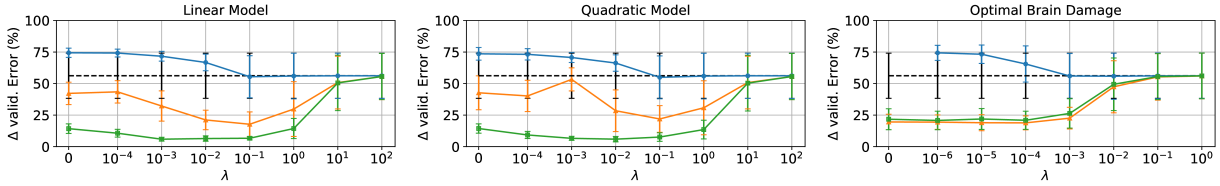
Training loss after fine-tuning. Figure 10.10 is the same as Figure 10.3 but showing $\mathcal{L}(\boldsymbol{\theta} \odot \mathbf{m})$ after fine-tuning as a function of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. It has a similar trend as Figure 10.3: there is not much correlation between the loss before and after fine-tuning, except on MNIST.

Different sparsity levels. Figure 10.11 shows the performances of different criteria on VGG11 on CIFAR10, for different sparsity levels. When the sparsity is low (89.3 %), the network has enough capacity to return to its original performances after fine-tuning. When the sparsity is too high (98.6 %), then all criteria produce networks with random predictions. There might be a sweet spot in between, but one would require more powerful model to verify this supposition.

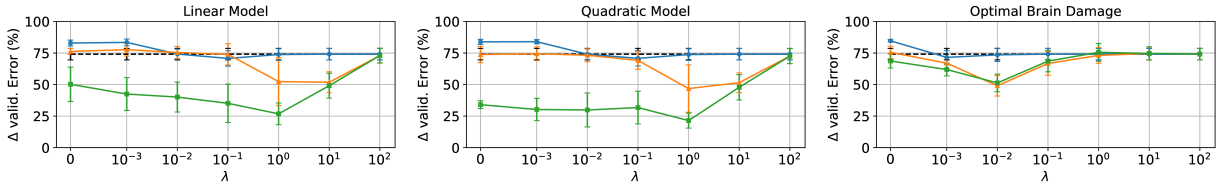
Hyper-parameters optimisation. Figure 10.13 shows the impact of hyper-parameter optimisation for the fine-tuning phase. We performed a grid search with three different learning rate (0.1, 0.01, 0.03) and three different l2-regularisation (0, 5e-4, 5e-5). All 9 sets of hyper-parameters were tested on LM, QM and MP on 5 different random seeds. In this set of experiments, we used $\lambda \in \{0, 0.01, 0.1, 1\}$ and $\pi \in \{14, 140\}$. Optimising hyper-parameters for fine-tuning can lead to better performance after fine-tuning, but does not increases the



(a) MLP on MNIST with 98.8% sparsity.



(b) VGG11 on CIFAR10 with 95.6% sparsity.



(c) PreActResNet18 on CIFAR10 with 95.6% sparsity.

Figure 10.6. Same as Figure 10.1, but displaying the validation error gap before fine-tuning. With proper number of pruning stages and step size regularisation, LM and QM can produce pruned networks that are drastically better than the ones pruned using MP.

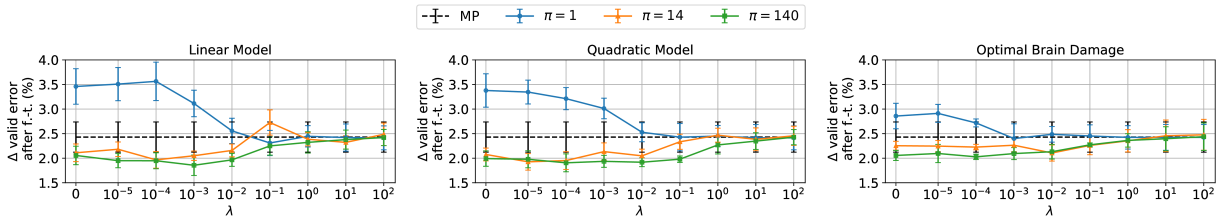
correlation between the performances after fine-tuning and $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. The lack of correlation can thus not be explained by bad fine-tuning hyper-parameters.

10.C.3. Fine-tuning Curves

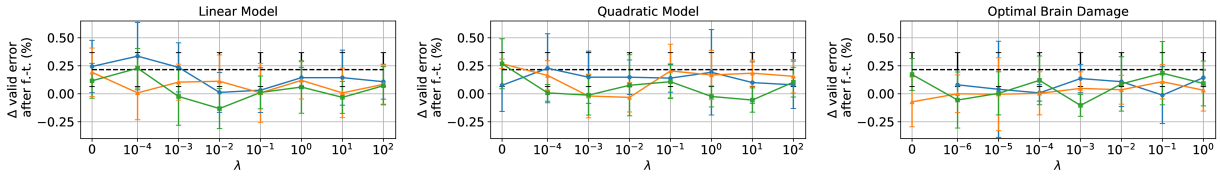
To investigate whether one of the networks is suffering from optimisation issues during fine-tuning, we show in Figure 10.15 the fine-tuning curves of networks pruned using MP and our best QM criteria. We observe that, except for MNIST, the difference in training loss right after pruning disappears after only one epoch of fine-tuning, erasing the advantage of QM over MP.

10.C.4. Results using GraSP and SynFlow

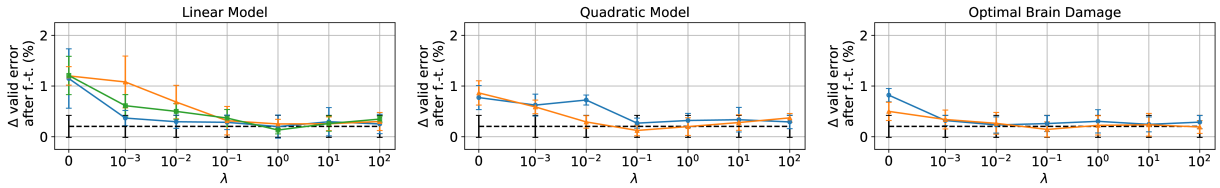
We compare our results with two additional pruning methods that focus on preserving the flow of the gradient in the network instead of preserving the loss: GraSP (Wang et al., 2020), a *data-dependant* method, and SynFlow (Tanaka et al., 2020), a *data-agnostic* one. Both



(a) MLP on MNIST with 98.8% sparsity.



(b) VGG11 on CIFAR10 with 95.6% sparsity.



(c) PreActResNet18 on CIFAR10 with 95.6% sparsity.

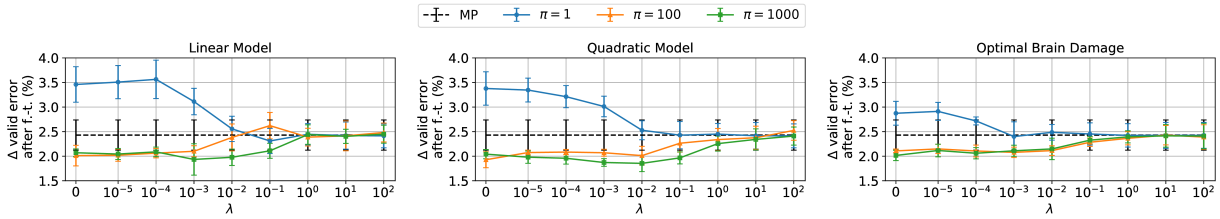
Figure 10.7. Same as Figure 10.1, but displaying the validation error gap after fine-tuning.

methods were design to be applied at initialisation, so we investigate here their use on trained networks. We use $\pi \in \{1, 100, 1000\}$, and added our proposed step size constraint λ to the pruning criteria as well.

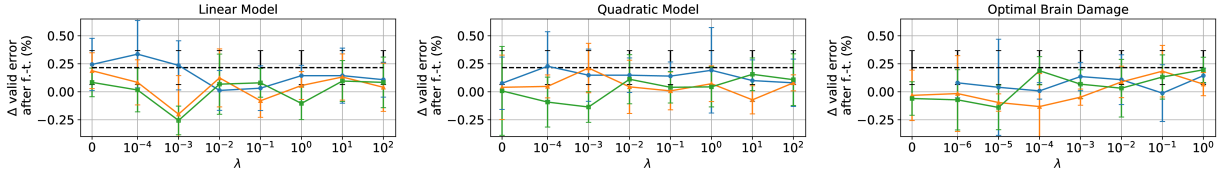
Figure 10.16 shows the scatter plot of the preservation of the loss vs the performance after fine-tuning. Similarly to what we observed before, there is no clear evidence that better preserving the loss lead to better performance after fine-tuning.

10.C.5. Results on ImageNet

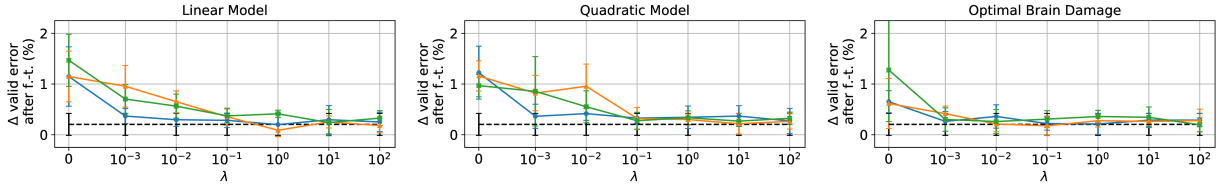
Figure 10.17 is the same as Figure 10.4, but with 90 % sparsity. At that sparsity level, the validation accuracy right after pruning is close to random for all the pruning criteria. There is however quite a big variation in performances after fine-tuning: at equal performance before fine-tuning, some models achieve 70 % validation accuracy after fine-tuning, while others only reach 60 %.



(a) MLP on MNIST with 98.8% sparsity.



(b) VGG11 on CIFAR10 with 95.6% sparsity.



(c) PreActResNet18 on CIFAR10 with 95.6% sparsity.

Figure 10.8. Same as Figure 10.7, but using equally spaced pruning steps. Note the difference in number of pruning stages.

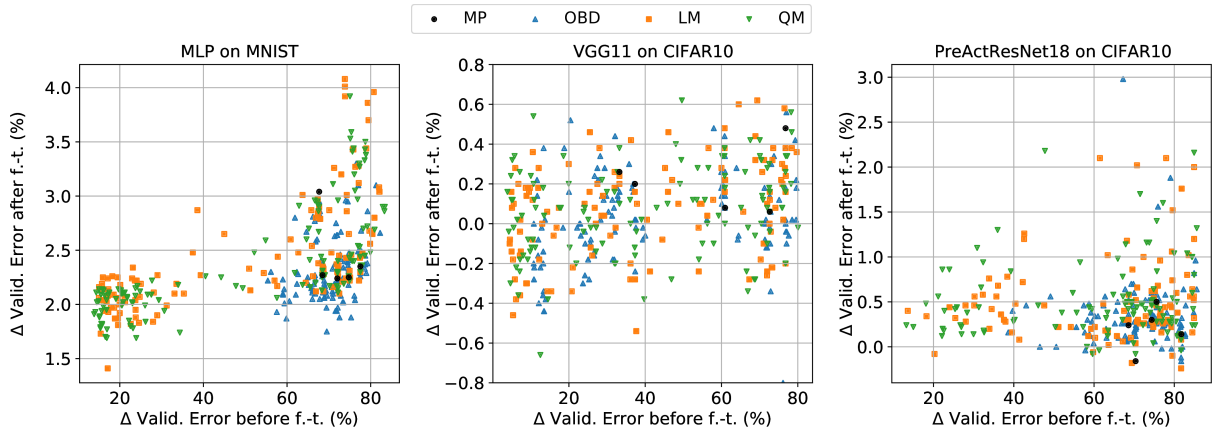


Figure 10.9. Same as Figure 10.3, but showing the validation error gap after fine-tuning as a function of the validation error gap before fine-tuning. Networks with drastically different performance before fine-tuning can still produce similar performances after fine-tuning.

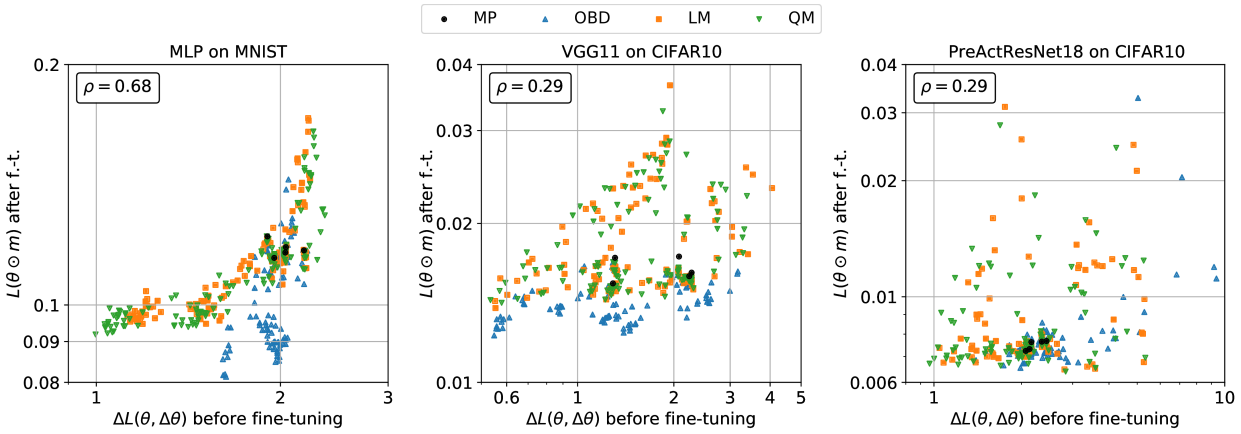


Figure 10.10. Same as Figure 10.3, but showing $\mathcal{L}(\boldsymbol{\theta} \odot \mathbf{m})$ after fine-tuning as a function of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. Except for the MLP on MNIST, there is only a weak correlation between $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ and $\mathcal{L}(\boldsymbol{\theta} \odot \mathbf{m})$ after fine-tuning.

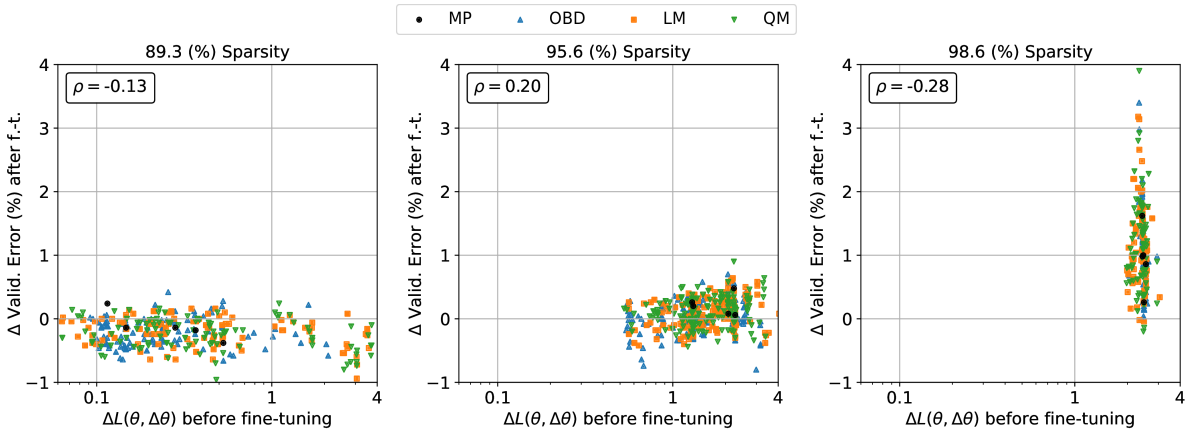


Figure 10.11. Same as Figure 10.3, but for different sparsity levels on the VGG11 on CIFAR10. When the sparsity is low, the network has enough capacity to return to its original performances after fine-tuning. When the sparsity is too high, then all criteria produce networks with random predictions.

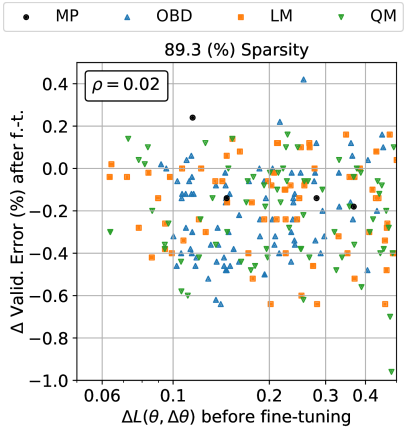


Figure 10.12. Same as Figure 10.11 (left), but zoomed on smaller values of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$.

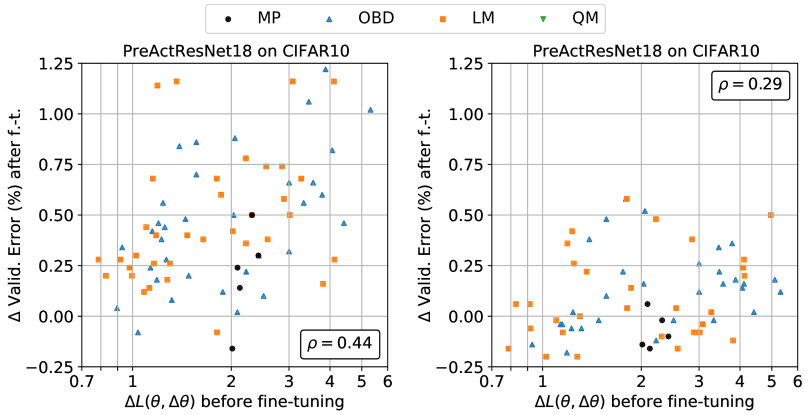


Figure 10.13. Left: Using the same hyper-parameters for fine-tuning as the ones of the original training. Right: Performing hyper-parameters optimisation for the fine-tuning. This figure shows that optimising the hyper-parameters for fine-tuning can improve the performances of the network after pruning. However, it reduces the correlation between the performances after fine-tuning and $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. The lack of correlation can thus not be explained by poor fine-tuning hyper-parameters.

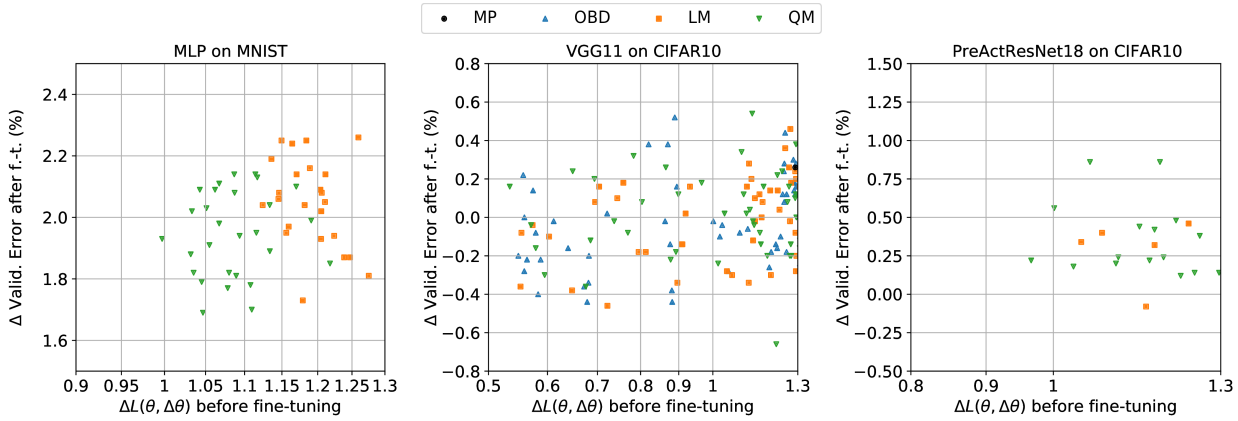


Figure 10.14. Same as Figure 10.3, but zooming on the best performing networks in terms of $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$.

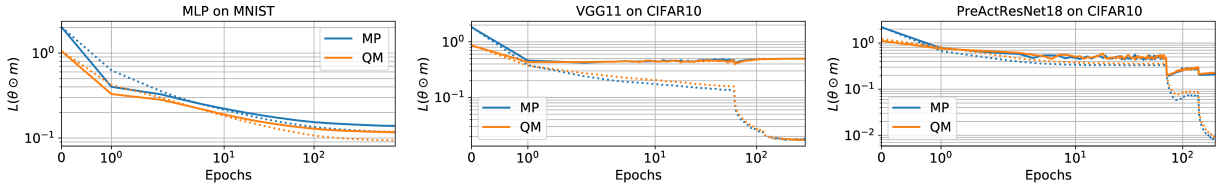


Figure 10.15. Fine-tuning losses (dotted is training, solid is validation) of networks pruned using MP and QM criteria. All the curves are the average over the 5 seeds. We do not show the standard deviation for clarity. Left: MLP, middle: VGG11 and right: PreActResNet18. Except for MNIST, the difference in loss right after pruning (i.e. at epoch 0) disappears after one epoch of fine-tuning.

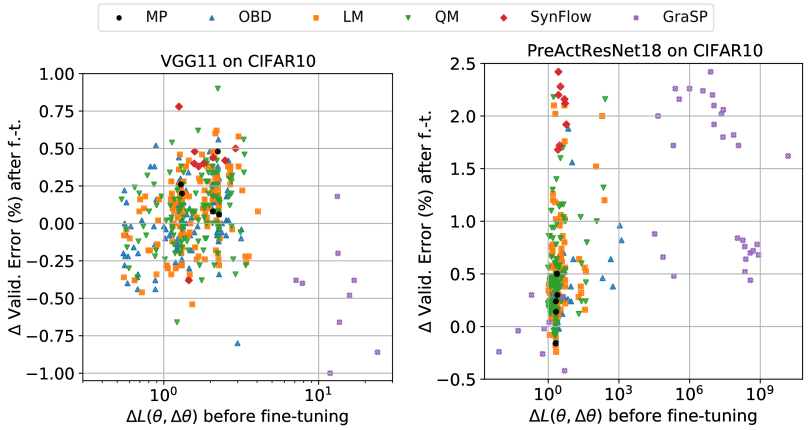


Figure 10.16. Same as Figure 10.3 showing GraSP and SynFlow on VGG11 (left) and the PreActResnet18 (right). This Figure shows that one can observe a large $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$ and yet obtain very good performance after fine-tuning. This is especially true in the case of GraSP for VGG11. Furthermore, we can observe similar behaviour on PreActResNet18 where two different methods can lead to similar performance after fine-tuning while having completely different $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$: GraSP with $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) \approx 10^8$ has fine-tuning performance similar to MP with $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) < 10^1$.

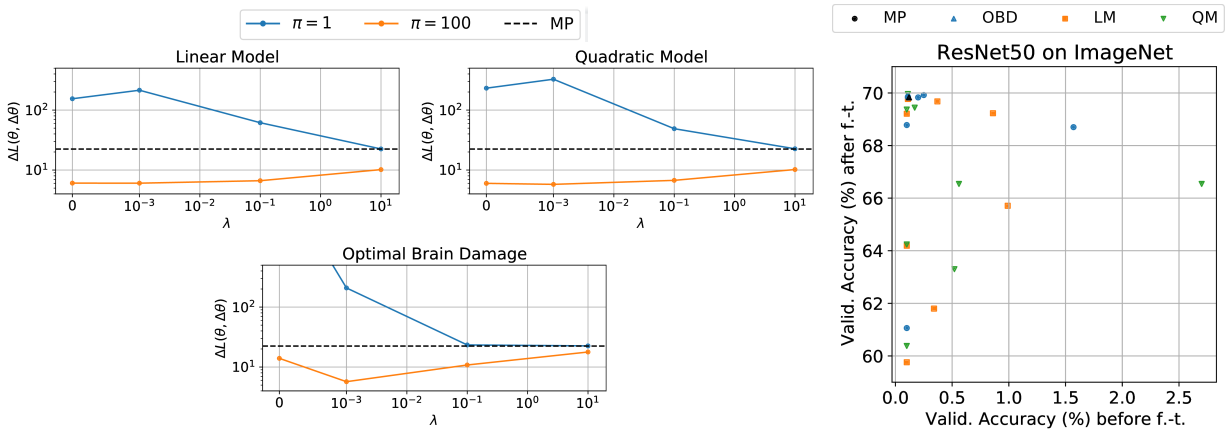


Figure 10.17. Same as Figure 10.4, but with 90 % sparsity. Increasing the number of pruning stages and constraining the step size reduce $\Delta\mathcal{L}(\boldsymbol{\theta}, \Delta\boldsymbol{\theta})$. However, the best-loss preserving criteria, which maximise the validation accuracy right after pruning, do not produce better networks after fine-tuning.

Chapter 11

Discussion

Although the articles presented in this thesis seem to treat different subjects, their common objective was to reduce the computational cost associated with neural networks, whether during training, through better parameterisation and optimisers, or for deployment, through better pruning methods.

I first focused on re-parameterisation techniques that enable faster and better training of RNNs. While the first application of BN to RNNs produced mitigated generalisation performances, it nonetheless proved successful in Baidu's speech recognition system *Deep Speech 2* (Amodei et al., 2016), and, as of this writing, it is still used in a few particular situations. Later next year, I helped develop a better parameterisation of the LSTM with BN, the BN-LSTM, that outperformed all our benchmarks by a significant margin. Although BN has been replaced by LN (Ba et al., 2016) in most of the RNNs that are used nowadays, it is great to see that normalised RNNs are still widely used in practice, and are at the core of many successful systems, particularly in NLP (Dai et al., 2019; Yang et al., 2019; Sun et al., 2020).

The next article was about designing an optimiser that leverages the K-FAC approximation to project the gradients to and from the KFE, allowing one to use traditional diagonal methods, such as RMSProp, in a basis where the diagonal approximation is likely to be more accurate. Just like any other second order method, the optimiser presented in this article has not been used much to train neural networks in practice. However, it is great to see that the idea of projecting to and from the KFE in various interesting applications, including training Bayesian networks (Bae et al., 2018), uncertainty modelling (Lee & Triebel, 2019) and structured pruning (Wang et al., 2019). As a future research direction, it would be nice to see how EKFAC scales compares to K-FAC in a large-scale distributed setup similar to Ueno et al. (2020), as well as better understand the practical impact of using the second

moment of the gradients rather than the Fisher in larger-scale experiments (Kunstner et al., 2019; Thomas et al., 2020). Finally, it would also be great to see it applied to problems where optimisation, rather than generalisation, is of primary concern, such as for training models on a continuous stream of data.

In the last article I decided to leverage the knowledge and code-base that we build to a more applied project: unstructured network pruning. We showed that respecting the assumptions behind the loss models that are used in pruning methods helps improve performance of networks right after pruning. However, we also showed that this did not correlate with the performances after fine-tuning, raising questions about the adequacy the objective used when designing in pruning methods. We hope that our findings will help to conceive better pruning methods. Future research directions should try to further analyse the re-trainability of pruned networks, to better understand why some networks can reach good performances after fine-tuning, while others obtain mitigated performance after fine-tuning. Another interesting direction would be to further analyse the unexpected behaviour of OBD on VGG11 and on the ResNet50 when violating its convergence assumption.

References

Alain, Guillaume, Roux, Nicolas Le, and Manzagol, Pierre-Antoine. Negative eigenvalues of the hessian in deep neural networks. *arXiv preprint*, 2019.

Amari, Shun-Ichi. Natural gradient works efficiently in learning. *Neural computation*, 1998.

Amodei, Dario, Ananthanarayanan, Sundaram, Anubhai, Rishita, Bai, Jingliang, Battenberg, Eric, Case, Carl, Casper, Jared, Catanzaro, Bryan, Cheng, Qiang, Chen, Guoliang, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, 2016.

Arjovsky, Martin, Shah, Amar, and Bengio, Yoshua. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, 2016.

Ba, Jimmy, Grosse, Roger, and Martens, James. Distributed second-order optimization using kronecker-factored approximations. In *ICLR*, 2017.

Ba, Jimmy Lei, Kiros, Jamie Ryan, and Hinton, Geoffrey E. Layer normalization. *arXiv preprint*, 2016.

Bae, Juhan, Zhang, Guodong, and Grosse, Roger. Eigenvalue corrected noisy natural gradient. *arXiv preprint*, 2018.

Baevski, Alexei, Zhou, Henry, Mohamed, Abdelrahman, and Auli, Michael. wav2vec 2.0: A framework for self-supervised learning of speech representations. *arXiv preprint*, 2020.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

Bastien, Frédéric, Lamblin, Pascal, Pascanu, Razvan, Bergstra, James, Goodfellow, Ian J., Bergeron, Arnaud, Bouchard, Nicolas, and Bengio, Yoshua. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

Becker, Sue, Le Cun, Yann, et al. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 connectionist models summer school*. San Matteo, CA: Morgan Kaufmann, 1988.

Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 1994.

Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006.

Blalock, Davis, Ortiz, Jose Javier Gonzalez, Frankle, Jonathan, and Guttag, John. What is the state of neural network pruning? *arXiv preprint*, 2020.

Bottou, Léon, Curtis, Frank E, and Nocedal, Jorge. Optimization methods for large-scale machine learning. *Siam Review*, 2018.

Carreira, Joao and Zisserman, Andrew. Quo vadis, action recognition? a new model and the kinetics dataset. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Cho, Kyunghyun, Van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint*, 2014.

Choromanska, Anna, Henaff, Mikael, Mathieu, Michael, Arous, Gérard Ben, and LeCun, Yann. The loss surfaces of multilayer networks. In *Artificial intelligence and statistics*, 2015.

Chung, Junyoung, Ahn, Sungjin, and Bengio, Yoshua. Hierarchical multiscale recurrent neural networks. *arXiv preprint*, 2016.

Clevert, Djork-Arné, Unterthiner, Thomas, and Hochreiter, Sepp. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint*, 2015.

Collobert, Ronan, Weston, Jason, Bottou, Léon, Karlen, Michael, Kavukcuoglu, Koray, and Kuksa, Pavel. Natural language processing (almost) from scratch. *Journal of machine learning research*, 2011.

Cooijmans, Tim, Ballas, Nicolas, Laurent, César, Gülçehre, Çağlar, and Courville, Aaron. Recurrent batch normalization. *International Conference on Learning Representations*, 2016.

Courbariaux, Matthieu, Hubara, Itay, Soudry, Daniel, El-Yaniv, Ran, and Bengio, Yoshua. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint*, 2016.

Dahl, George E, Yu, Dong, Deng, Li, and Acero, Alex. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 2011.

Dai, Zihang, Yang, Zhilin, Yang, Yiming, Carbonell, Jaime, Le, Quoc V, and Salakhutdinov, Ruslan. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint*, 2019.

Deng, Jia, Dong, Wei, Socher, Richard, Li, Li-Jia, Li, Kai, and Fei-Fei, Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009.

Deng, Jia, Berg, Alex, Satheesh, Sanjeev, Su, H, Khosla, Aditya, and Fei-Fei, L. Imagenet large scale visual recognition competition 2012 (ilsvrc2012). 2012.

Denton, Emily L, Zaremba, Wojciech, Bruna, Joan, LeCun, Yann, and Fergus, Rob. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, 2014.

Desjardins, Guillaume, Simonyan, Karen, Pascanu, Razvan, et al. Natural neural networks. In *NIPS*, 2015.

Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint*, 2018.

Ding, Xiaohan, Zhou, Xiangxin, Guo, Yuchen, Han, Jungong, Liu, Ji, et al. Global sparse momentum sgd for pruning very deep neural networks. In *NeurIPS*, 2019.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

Dumoulin, Vincent and Visin, Francesco. A guide to convolution arithmetic for deep learning, 2016.

Edunov, Sergey, Ott, Myle, Auli, Michael, and Grangier, David. Understanding back-translation at scale. *arXiv preprint*, 2018.

Esteva, Andre, Kuprel, Brett, Novoa, Roberto A, Ko, Justin, Swetter, Susan M, Blau, Helen M, and Thrun, Sebastian. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 2017.

Evci, Utku, Ioannou, Yani A, Keskin, Cem, and Dauphin, Yann. Gradient flow in sparse neural networks and how lottery tickets win. *arXiv preprint*, 2020.

Frankle, Jonathan and Carbin, Michael. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*, 2018.

Frankle, Jonathan, Dziugaite, Gintare Karolina, Roy, Daniel M, and Carbin, Michael. Pruning neural networks at initialization: Why are we missing the mark? *arXiv preprint*, 2020.

Fujimoto, Yuki and Ohira, Toru. A neural network model with bidirectional whitening. In *International Conference on Artificial Intelligence and Soft Computing*. Springer, 2018.

Gale, Trevor, Elsen, Erich, and Hooker, Sara. The state of sparsity in deep neural networks. *arXiv preprint*, 2019.

Gehring, Jonas, Auli, Michael, Grangier, David, Yarats, Denis, and Dauphin, Yann N. Convolutional sequence to sequence learning. In *ICLR*, 2017.

George, Thomas, Laurent, César, Bouthillier, Xavier, Ballas, Nicolas, and Vincent, Pascal. Fast approximate natural gradient descent in a kronecker factored eigenbasis. In *NIPS*, 2018.

Gers, Felix A, Schraudolph, Nicol N, and Schmidhuber, Jürgen. Learning precise timing with lstm recurrent networks. *The Journal of Machine Learning Research*, 2003.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep Learning*. MIT Press, 2016.

Google. Cloud TPU. `https://cloud.google.com/tpu`, 2018. Accessed on October 1^{st} 2020.

Goyal, Priya, Dollár, Piotr, Girshick, Ross, Noordhuis, Pieter, Wesolowski, Lukasz, Kyrola, Aapo, Tulloch, Andrew, Jia, Yangqing, and He, Kaiming. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint*, 2017.

Graves, Alan, Jaitly, Navdeep, and Mohamed, Abdel-rahman. Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE, 2013a.

Graves, Alan, Mohamed, Abdel-rahman, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013b.

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint*, 2013.

Graves, Alex, Fernández, Santiago, Gomez, Faustino, and Schmidhuber, Jürgen. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, 2006.

Grosse, Roger and Martens, James. A kronecker-factored approximate fisher matrix for convolution layers. In *ICML*, 2016.

Ha, David, Dai, Andrew, and Le, Quoc V. Hypernetworks. *arXiv preprint*, 2016.

Han, Song, Mao, Huizi, and Dally, William J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint*, 2015a.

Han, Song, Pool, Jeff, Tran, John, and Dally, William. Learning both weights and connections for efficient neural network. In *NIPS*. 2015b.

Han, Song, Liu, Xingyu, Mao, Huizi, Pu, Jing, Pedram, Ardavan, Horowitz, Mark A, and Dally, William J. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 2016.

Hannun, Awni, Case, Carl, Casper, Jared, Catanzaro, Bryan, Diamos, Greg, Elsen, Erich, Prenger, Ryan, Satheesh, Sanjeev, Sengupta, Shubho, Coates, Adam, et al. Deepspeech: Scaling up end-to-end speech recognition. *arXiv preprint*, 2014.

Hassibi, Babak and Stork, David G. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, 1993.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. In *CVPR*, 2016a.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Identity mappings in deep residual networks. In *ECCV*, 2016b.

Hermann, Karl Moritz, Kocisky, Tomas, Grefenstette, Edward, Espeholt, Lasse, Kay, Will, Suleyman, Mustafa, and Blunsom, Phil. Teaching machines to read and comprehend. In *Advances in neural information processing systems*, 2015.

Heskes, Tom. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 2000.

Hinton, Geoffrey, Vinyals, Oriol, and Dean, Jeff. Distilling the knowledge in a neural network. *arXiv preprint*, 2015.

Hinton, Geoffrey E and Salakhutdinov, Ruslan R. Reducing the dimensionality of data with neural networks. *science*, 2006.

Hinton, Geoffrey E, Srivastava, Nitish, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint*, 2012.

Hochreiter, Sepp. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 1991.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 1997.

Hochreiter, Sepp, Bengio, Yoshua, Frasconi, Paolo, Schmidhuber, Jürgen, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Kingma, Diederik P and Ba, Jimmy. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Krizhevsky, Alex and Hinton, Geoffrey. Learning multiple layers of features from tiny images. 2009.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

Krueger, David and Memisevic, Roland. Regularizing rnns by stabilizing activations. *arXiv preprint*, 2015.

Krueger, David, Maharaj, Tegan, Kramár, János, Pezeshki, Mohammad, Ballas, Nicolas, Ke, Nan Rosemary, Goyal, Anirudh, Bengio, Yoshua, Larochelle, Hugo, and Courville, Aaron. Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint*, 2016.

Kunstner, Frederik, Hennig, Philipp, and Balles, Lukas. Limitations of the empirical fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems*, 2019.

Kusupati, Aditya, Ramanujan, Vivek, Somani, Raghav, Wortsman, Mitchell, Jain, Prateek, Kakade, Sham, and Farhadi, Ali. Soft threshold weight reparameterization for learnable sparsity. *arXiv preprint*, 2020.

Lacoste, Alexandre, Luccioni, Alexandra, Schmidt, Victor, and Dandres, Thomas. Quantifying the carbon emissions of machine learning. *arXiv preprint*, 2019.

Laurent, César, Pereyra, Gabriel, Brakel, Philémon, Zhang, Ying, and Bengio, Yoshua. Batch normalized recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016.

Laurent, César, George, Thomas, Bouthillier, Xavier, Ballas, Nicolas, and Vincent, Pascal. An evaluation of fisher approximations beyond kronecker factorization. *ICLR Workshop*, 2018.

Laurent, César, Ballas, Camille, George, Thomas, Ballas, Nicolas, and Vincent, Pascal. Revisiting loss modelling for unstructured pruning. *arXiv preprint*, 2020.

Le, Quoc V, Jaitly, N., and Hinton, G. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint*, 2015.

Le Roux, Nicolas, Manzagol, Pierre-Antoine, and Bengio, Yoshua. Topmoumoute online natural gradient algorithm. In *NIPS*, 2008.

LeCun, Yann. Who is afraid of convex optimization? NIPS - Workshop on Efficient Learning, 2007. URL `https://cs.nyu.edu/~yann/talks/lecun-20071207-nonconvex.pdf`.

LeCun, Yann, Boser, Bernhard, Denker, John S, Henderson, Donnie, Howard, Richard E, Hubbard, Wayne, and Jackel, Lawrence D. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1989.

LeCun, Yann, Denker, John S, and Solla, Sara A. Optimal brain damage. In *NIPS*, 1990.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

LeCun, Yann A, Bottou, Léon, Orr, Genevieve B, and Müller, Klaus-Robert. Efficient backprop. In *Neural networks: Tricks of the trade*. 2012.

Ledinauskas, Eimantas, Ruseckas, Julius, Juršėnas, Alfonsas, and Buračas, Giedrius. Training deep spiking neural networks. *arXiv preprint*, 2020.

Lee, Jongseok and Triebel, Rudolph. Representing model uncertainty of neural networks in sparse information form. 2019.

Lee, Namhoon, Ajanthan, Thalaiyasingam, Gould, Stephen, and Torr, Philip HS. A signal propagation perspective for pruning neural networks at initialization. *arXiv preprint*, 2019a.

Lee, Namhoon, Ajanthan, Thalaiyasingam, and Torr, Philip. SNIP: Single-Shot Network Pruning based on Connection Sensitivity. In *ICLR*, 2019b.

Liao, Qianli and Poggio, Tomaso. Bridging the gaps between residual learning, recurrent neural networks and visual cortex. *arXiv preprint*, 2016.

Lin, Tao, Stich, Sebastian U., Barba, Luis, Dmitriev, Daniil, and Jaggi, Martin. Dynamic model pruning with feedback. In *International Conference on Learning Representations*, 2020.

Liu, Dong C and Nocedal, Jorge. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 1989.

Liu, Zhuang, Li, Jianguo, Shen, Zhiqiang, Huang, Gao, Yan, Shoumeng, and Zhang, Changshui. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, 2017.

Liu, Zhuang, Sun, Mingjie, Zhou, Tinghui, Huang, Gao, and Darrell, Trevor. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019.

Louizos, Christos, Welling, Max, and Kingma, Diederik P. Learning sparse neural networks through l_0 regularization. *arXiv preprint*, 2017.

Lubana, Ekdeep Singh and Dick, Robert P. A gradient flow framework for analyzing network pruning, 2020.

Maas, Andrew L, Hannun, Awni Y, and Ng, Andrew Y. Rectifier nonlinearities improve neural network acoustic models. In *ICML*, 2013.

Mahoney, Matt. Large text compression benchmark. 2009.

Marcus, Mitchell, Santorini, Beatrice, and Marcinkiewicz, Mary Ann. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 1993.

Martens, James and Grosse, Roger. Optimizing neural networks with kronecker-factored approximate curvature. In *ICML*, 2015.

Martens, James and Sutskever, Ilya. Learning recurrent neural networks with hessian-free optimization. In *ICML*, 2011.

Martens, James, Ba, Jimmy, and Johnson, Matt. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

Mikolov, Tomáš. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 2012.

Mikolov, Tomáš, Sutskever, Ilya, Deoras, Anoop, Le, Hai-Son, Kombrink, Stefan, and Cernocky, Jan. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 2012.

Moczulski, Marcin, Denil, Misha, Appleyard, Jeremy, and de Freitas, Nando. Acdc: A structured efficient linear layer. *arXiv preprint*, 2015.

Molchanov, Dmitry, Ashukha, Arsenii, and Vetrov, Dmitry. Variational dropout sparsifies deep neural networks. In *ICML*, 2017.

Molchanov, Pavlo, Mallya, Arun, Tyree, Stephen, Frosio, Iuri, and Kautz, Jan. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

Nam, Ju Gang, Park, Sunggyun, Hwang, Eui Jin, Lee, Jong Hyuk, Jin, Kwang-Nam, Lim, Kun Young, Vu, Thienkai Huy, Sohn, Jae Ho, Hwang, Sangheum, Goo, Jin Mo, et al. Development and validation of deep learning–based automatic detection algorithm for

malignant pulmonary nodules on chest radiographs. *Radiology*, 2019.

Nocedal, Jorge and Wright, Stephen. *Numerical optimization*. Springer Science & Business Media, 2006.

Ollivier, Yann. Persistent contextual neural networks for learning symbolic data sequences. *CoRR*, abs/1306.0514, 2013.

Ollivier, Yann. Riemannian metrics for neural networks i: feedforward networks. *Information and Inference: A Journal of the IMA*, 2015.

Pachitariu, Marius and Sahani, Maneesh. Regularization and nonlinearities for neural language models: when are they needed? *arXiv preprint*, 2013.

Papyan, Vardan. The full spectrum of deepnet hessians at scale: Dynamics with sgd training and sample size. *arXiv preprint*, 2018.

Pascanu, Razvan and Bengio, Yoshua. Revisiting natural gradient for deep networks. *arXiv preprint*, 2013.

Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, and Bengio, Yoshua. How to construct deep recurrent neural networks. *arXiv preprint*, 2013a.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, 2013b.

Paszke, Adam, Gross, Sam, Chintala, Soumith, Chanan, Gregory, Yang, Edward, DeVito, Zachary, Lin, Zeming, Desmaison, Alban, Antiga, Luca, and Lerer, Adam. Automatic differentiation in pytorch. 2017.

Paul, Douglas B and Baker, Janet M. The design for the wall street journal-based csr corpus. In *Proceedings of the workshop on Speech and Natural Language*. Association for Computational Linguistics, 1992.

Phan, Huy, Andreotti, Fernando, Cooray, Navin, Chén, Oliver Y, and De Vos, Maarten. Seqsleepnet: end-to-end hierarchical recurrent neural network for sequence-to-sequence automatic sleep staging. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 2019.

Ravanelli, Mirco, Brakel, Philemon, Omologo, Maurizio, and Bengio, Yoshua. Light gated recurrent units for speech recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2018.

Ravanelli, Mirco, Parcollet, Titouan, and Bengio, Yoshua. The pytorch-kaldi speech recognition toolkit. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019.

Renda, Alex, Frankle, Jonathan, and Carbin, Michael. Comparing rewinding and fine-tuning in neural network pruning. In *ICLR*, 2020.

Rolnick, David, Donti, Priya L, Kaack, Lynn H, Kochanski, Kelly, Lacoste, Alexandre, Sankaran, Kris, Ross, Andrew Slavin, Milojevic-Dupont, Nikola, Jaques, Natasha, Waldman-Brown, Anna, et al. Tackling climate change with machine learning. *arXiv preprint*, 2019.

Romero, Adriana, Ballas, Nicolas, Kahou, Samira Ebrahimi, Chassang, Antoine, Gatta, Carlo, and Bengio, Yoshua. Fitnets: Hints for thin deep nets. *arXiv preprint*, 2014.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 2015.

Sagun, Levent, Evci, Utku, Guney, V Ugur, Dauphin, Yann, and Bottou, Leon. Empirical analysis of the hessian of over-parametrized neural networks. *arXiv preprint*, 2017.

Schraudolph, Nicol N. Fast curvature matrix-vector products. In *International Conference on Artificial Neural Networks*. Springer, 2001.

Schraudolph, Nicol N. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 2002.

Schuster, Mike and Paliwal, Kuldip K. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 1997.

Shen, Sheng, Yao, Zhewei, Gholami, Amir, Mahoney, Michael, and Keutzer, Kurt. Rethinking batch normalization in transformers. *arXiv preprint*, 2020.

Shimodaira, Hidetoshi. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 2000.

Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 2016.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Singh, Sidak Pal and Alistarh, Dan. Woodfisher: Efficient second-order approximations for model compression. *arXiv preprint*, 2020.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 2014.

Sun, Yu, Wang, Shuohuan, Li, Yu-Kun, Feng, Shikun, Tian, Hao, Wu, Hua, and Wang, Haifeng. Ernie 2.0: A continual pre-training framework for language understanding. In *AAAI*, 2020.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

Tanaka, Hidenori, Kunin, Daniel, Yamins, Daniel LK, and Ganguli, Surya. Pruning neural networks without any data by iteratively conserving synaptic flow. *arXiv preprint*, 2020.

Team, The Theano Development et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint*, May 2016.

Thomas, Valentin, Pedregosa, Fabian, Merriënboer, Bart, Manzagol, Pierre-Antoine, Bengio, Yoshua, and Le Roux, Nicolas. On the interplay between noise and curvature and its effect on optimization and generalization. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2020.

Tieleman, Tijmen and Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 2012.

Ueno, Yuichiro, Osawa, Kazuki, Tsuji, Yohei, Naruse, Akira, and Yokota, Rio. Rich information is affordable: A systematic performance analysis of second-order optimization using k-fac. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

Van Merriënboer, Bart, Bahdanau, Dzmitry, Dumoulin, Vincent, Serdyuk, Dmitriy, Warde-Farley, David, Chorowski, Jan, and Bengio, Yoshua. Blocks and fuel: Frameworks for deep learning. *arXiv preprint*, 2015.

Vorontsov, Eugene, Trabelsi, Chiheb, Kadoury, Samuel, and Pal, Chris. On orthogonality and learning recurrent networks with long term dependencies. In *International Conference on Machine Learning*, 2017.

Wang, Chaoqi, Grosse, Roger, Fidler, Sanja, and Zhang, Guodong. Eigendamage: Structured pruning in the kronecker-factored eigenbasis. *arXiv preprint*, 2019.

Wang, Chaoqi, Zhang, Guodong, and Grosse, Roger. Picking winning tickets before training by preserving gradient flow. In *ICLR*, 2020.

Williams, Will, Prasad, Niranjani, Mrva, David, Ash, Tom, and Robinson, Tony. Scaling recurrent neural network language models. *arXiv preprint*, 2015.

Wisdom, Scott, Powers, Thomas, Hershey, John, Le Roux, Jonathan, and Atlas, Les. Full-capacity unitary recurrent neural networks. In *Advances in neural information processing systems*, 2016.

Wu, Jiaxiang, Leng, Cong, Wang, Yuhang, Hu, Qinghao, and Cheng, Jian. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Xie, Qizhe, Luong, Minh-Thang, Hovy, Eduard, and Le, Quoc V. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Xie, Saining, Girshick, Ross, Dollár, Piotr, Tu, Zhuowen, and He, Kaiming. Aggregated residual transformations for deep neural networks. In *CVPR*, 2017.

Xu, Kelvin, Ba, Jimmy, Kiros, Ryan, Cho, Kyunghyun, Courville, Aaron, Salakhudinov, Ruslan, Zemel, Rich, and Bengio, Yoshua. Show, attend and tell: Neural image caption

generation with visual attention. In *International conference on machine learning*, 2015.

Yang, Zhilin, Dai, Zihang, Yang, Yiming, Carbonell, Jaime, Salakhutdinov, Russ R, and Le, Quoc V. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, 2019.

Yao, Li, Torabi, Atousa, Cho, Kyunghyun, Ballas, Nicolas, Pal, Christopher, Larochelle, Hugo, and Courville, Aaron. Describing videos by exploiting temporal structure. In *Proceedings of the IEEE international conference on computer vision*, 2015.

Yao, Shuochao, Hu, Shaohan, Zhao, Yiran, Zhang, Aston, and Abdelzaher, Tarek. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web*, 2017.

Ye, Jianbo, Lu, Xin, Lin, Zhe, and Wang, James Z. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. *arXiv preprint*, 2018.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint*, 2014.

Zeiler, Matthew D. Adadelta: an adaptive learning rate method. *arXiv preprint*, 2012.

Zeng, Wenyuan and Urtasun, Raquel. MLPrune: Multi-layer pruning for automated neural network compression, 2019. URL `https://openreview.net/forum?id=r1g5b2RcKm`.

Zhang, Saizheng, Wu, Yuhuai, Che, Tong, Lin, Zhouhan, Memisevic, Roland, Salakhutdinov, Russ R, and Bengio, Yoshua. Architectural complexity measures of recurrent neural networks. In *Advances in neural information processing systems*, 2016.

Zhang, Ying, Pezeshki, Mohammad, Brakel, Philémon, Zhang, Saizheng, Bengio, César Laurent Yoshua, and Courville, Aaron. Towards end-to-end speech recognition with deep convolutional neural networks. *arXiv preprint*, 2017.