JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Pieter-Jan Hoedt

Submitted at
Institute of Bioinformatics

Supervisor
Univ. Prof. Dr. Sepp Hochreiter

Co-Supervisor
Mag. Dr. Günter Klambauer

October 2017

Moment Dynamics in Self-Normalising Neural Networks

Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

In the passed decade, deep learning has achieved state-of-the-art performance for various machine learning tasks. The wide applicability of deep learning mainly arises from the ability of deep neural networks to learn useful features by themselves. By combining multiple layers in a neural network, hierarchical representations can be created with an increasing level of abstraction. However, there is one fundamental difficulty in learning deep neural networks, which is known as the vanishing gradient problem. Today, this issue has been alleviated by new activation functions and better ways to initialise weights. Although less severe, also internal covariate shift slows down learning in deep networks. Several techniques such as batch normalisation have been proposed to counter this issue by normalising the data in each layer. An alternative approach is to construct the layers in a network so that their activations have the same mean and variance as the data in the input. Networks which are capable of doing so are called Self-Normalising neural networks (SNNs) and can be constructed by enforcing certain characteristics onto the mappings that each layer induces on the moments of its input data. In this thesis, we extend the idea of analysing the moments in SNNs to the backward pass, in which we both theoretically and empirically investigate the backward dynamics. Further, we extend SNNs to networks with bias units and show that similar dynamics hold as for the weights. Additionally, we compare the performance and learning behaviour of SNNs with respect to different data normalization techniques, weight distributions for initialization and optimisers on several machine learning benchmarking data sets. We find that (1) the variance of the weights steadily increases with very small steps in networks with random errors, (2) the architecture affects how the error signal propagates back through the network and (3) that an error signal with reduced variance in lower layers can be advantageous for learning. Furthermore, our analysis reveals that SNNs perform best with whitened data, the choice of the initial weight distribution has no significant effect on the learning behaviour and that most adaptive learning rate schedules do help, although they break the conditions for self-normalisation.

Zusammenfassung

Im letzten Jahrzehnt hat deep learning verschiedenste machine learning Probleme geknackt. deep learning verdankt seine vielseitige Anwendbarkeit der Fähigkeit von tiefen neuronalen Netzten eigenständig sinnvolle Eigenschaften von Daten zu finden. Durch die Kombination von verschiedenen Schichten in einem neuronalen Netz können hierarchische Darstellungen in unterschiedlichen Abstraktionsebenen erstellt werden. Das Trainieren von tiefen neuronalen Netzen kann jedoch durch ein fundamentales Problem, dem sogenannten Vanishing Gradient Problem, erschwert werden. Heutzutage wird dieses Problem weitgehend umgangen mithilfe von neuen Aktivierungsfunktionen und besseren Methoden zur Initialisierung von den Gewichten der neuronalen Netze. Ein weiterer Faktor der zur Verlangsamung des Lernvorganges beitragen kann, ist das Problem des Internal Covariate Shift. Verschiedenste Verfahrensweisen wie etwa Batch Normalisation, welche dieses Problem durch die Normalisierung von Daten in jeder Schicht zu lösen versuchen, wurden bereits vorgestellt. Eine alternative Lösung besteht darin, die Schichten so aufzubauen, dass der Erwartungswert und die Varianz der Eingabewerte auch die Momente der Ausgabewerte sind. Netzwerke mit solchen Schichten werden SNNs genannt, und fordern gewisse Eigenschaften für eine Abbildung von Momenten des Inputs jeder Schicht. In dieser Arbeit erweitern wir die Analyse von Momenten in SNNs zu dem Fehlerrückfluss, wobei diese Dynamik sowohl theoretisch als auch empirisch untersucht wird. Zusätzlich erweitern wir SNNs mit Bias-Units und zeigen, dass dennoch eine ähnliche Dynamik aufgewiesen wird wie mit Gewichten. Des Weiteren vergleichen wir die Leistung und Lerngeschwindigkeit von SNNs hinsichtlich verschiedener Methoden für Datennormalisierung, Verteilungen für die Initialisierung von Gewichten sowie unterschiedlicher Optimierungsmöglichkeiten anhand von verschiedenen machine learning Benchmarks. Wir stellen fest, dass (1) in Netzwerken mit zufälligen Fehlern die Varianz der Gewichte in kleinen Schritten ständig zunimmt, (2) die Netzwerkarchitektur einen Einfluss auf den Fehlerrückfluss hat und (3), dass schneller gelernt werden kann wenn die Netzwerke in den ersten Schichten weniger Varianz in dem Fehlersignal aufweisen. Zusätzlich finden wir heraus, dass SNNs besser arbeiten mit nicht-korrelierten und normalisierten Daten, die genaue Verteilung der Anfangsgewichte keinen signifikanten Einfluss auf das Lernen hat und, dass die meisten komplexen Optimierungsmethoden das Lernen positiv beeinflussen, obwohl sie den normalisierenden Effekt von SNNs brechen.

Acknowledgements

First of all, I would like to thank the bioinformatics institute at the JKU that provided me access to GPUs. This saved me quite a bit of time to run all of the experiments. They also provided me with a forum to present my ideas and retrieve valuable feedback. Special thanks to Günter Klambauer who has patiently guided me through this thesis and provided new ideas .

I would also like to thank the people that contribute(d) to open source projects like *Tensorflow* (Google) (Abadi et al., 2015), *Theano* (MILA) (Al-Rfou et al., 2016) and *Keras* (François Chollet) (Chollet, 2015). These deep learning frameworks greatly simplified the process of building and training neural networks in python. Especially Keras, which allows to switch between different lower-level learning frameworks, has been very useful to set up experiments for this thesis. Also matplotlib (Hunter, 2007) has been of great use to make the visualisations of the experiments as they are.

I would also like to explicitly thank Stéfan van der Walt and Nathaniel Smith for developing *virridis*. Their quest for the perfect colour map has made it easier for me to focus on content rather than colour schemes. For more information on their work, I refer to `https://bids.github.io/colormap/`.

The (free) availability of platforms as wolframalpha and the wolframcloud have been extremely useful for the more theoretical parts of this thesis. Being able to double-check most of the equations has solidified the confidence in my research and writing.

Last, but not least, I would like to thank Katharina Prinz for her patience, multiple reviews and valuable feedback for this thesis.

Contents

1 Introduction 1

2 Machine Learning 3
- 2.1 Definition and Taxonomy . . . 3
  - 2.1.1 Taxonomy . . . 4
- 2.2 Supervised Learning . . . 4
  - 2.2.1 Labelled Data . . . 4
  - 2.2.2 Risk and Loss . . . 5
  - 2.2.3 Bayes Classifier . . . 6
- 2.3 Practical Risk Minimisation . . . 6
  - 2.3.1 Empirical . . . 6
  - 2.3.2 Hypothesis Space . . . 7
  - 2.3.3 Structural . . . 8
- 2.4 A Bunch of Errors . . . 8
  - 2.4.1 Excess Error . . . 9
  - 2.4.2 Bias vs Variance . . . 10
- 2.5 Risk assessment . . . 11
  - 2.5.1 Test Set . . . 12
  - 2.5.2 Cross Validation . . . 13

3 Neural Networks 15
- 3.1 Basic Structure . . . 15
  - 3.1.1 Neurons . . . 15
  - 3.1.2 Layers with Bias . . . 17
  - 3.1.3 A Network of Connected Layers . . . 18
- 3.2 Alternative Architectures . . . 18
  - 3.2.1 Convolutional Neural Network . . . 19
  - 3.2.2 Long Short-Term Memory . . . 20
- 3.3 Hyperparameters . . . 20
  - 3.3.1 Activation Functions . . . 21
  - 3.3.2 Weight Initialisation . . . 22
- 3.4 Learning . . . 24
  - 3.4.1 Product of Jacobians . . . 25
  - 3.4.2 Delta propagation . . . 26
- 3.5 Optimisation . . . 27
  - 3.5.1 Variations on Gradient Descent . . . 27
  - 3.5.2 Next-Level Gradient Descent . . . 28

4 Self-Normalisation 29
- 4.1 Internal Covariate shift . . . 29
  - 4.1.1 Regular Normalisation . . . 30
- 4.2 Self-Normalisation . . . 31
  - 4.2.1 Contraction Mapping . . . 31
  - 4.2.2 Self-Normalising Layers . . . 32
- 4.3 Scaled Exponential Linear Units . . . 33
  - 4.3.1 Finding parameters . . . 33
  - 4.3.2 The Role of Initialisation . . . 34
- 4.4 Moment Dynamics . . . 35
  - 4.4.1 Moments of Deltas . . . 35
  - 4.4.2 Backprop with SELUs . . . 36
  - 4.4.3 The Role of Momentum . . . 37

5 Empirical Results 39
- 5.1 Experiment Settings . . . 39
  - 5.1.1 Datasets . . . 39
  - 5.1.2 Base Model . . . 41
  - 5.1.3 Visualisations . . . 42
- 5.2 Pre-processing . . . 43
- 5.3 Weights . . . 45
  - 5.3.1 Initialisation . . . 46
  - 5.3.2 Dynamics . . . 46
  - 5.3.3 Biases . . . 48
- 5.4 Architecture . . . 49
  - 5.4.1 Set-up . . . 49
  - 5.4.2 Observations . . . 51
  - 5.4.3 Learning from the Back . . . 51
- 5.5 Optimisers . . . 51

6 Conclusion 57

List of Figures

2.1 Structural Risk Minimisation 8
2.2 Under- and Overfitting 10
2.3 Bias and Variance of Polynomial Hypothesis Space 11
2.4 Test Set and Cross Validation 12

3.1 Artificial and Biological Neurons 16
3.2 Artificial Neural Network 18
3.3 Activation Functions 23

4.1 Internal Covariate Shift 30
4.2 Scaled Exponential Linear Unit 34

5.1 MNIST images 40
5.2 CIFAR-10 images 41
5.3 CIFAR-100 images 41
5.4 Distributions in the baseline model 42
5.5 Distributions for different pre-processing techniques 44
5.6 Loss for different pre-processing techniques 45
5.7 Distributions of weights before learning 47
5.8 Distributions of biases 49
5.9 Loss for different weight initialisations 50
5.10 Loss for different weight initialisations with biases 50
5.11 Distributions of deltas for first update 52
5.12 Loss for different architectures on MNIST datasets 53
5.13 Loss for different architectures on CIFAR datasets 53
5.14 Distributions of weights for different optimisers 55
5.15 Loss for different optimisers 55

1 Introduction

It has been over a decade since the term deep learning (DL) has been introduced to prelude a new era of neural network (NN) research (G. E. Hinton, Osindero and Teh, 2006). In these ten years, DLs has grown to become one of the most influential machine learning (ML) methods by increasingly improving on state-of-the-art models in the most various fields, e.g. computer vision (He et al., 2016), speech recognition, (Amodei et al., 2016) and natural language processing (Wu et al., 2016).

One of the most interesting characteristics of DL is that they are representation-learning methods (LeCun, Bengio and G. E. Hinton, 2015). This means that NNs are capable of learning useful features from raw data. The time-consuming process of feature engineering can therefore be skipped. Also domain expertise is not a necessity any longer, which makes it easier to get started with learning.

None of these properties of NNs are new since DL, however. It was well known that NNs learn useful representations in hidden layers (Rumelhart, G. E. Hinton and Williams, 1986). Also the universal function approximation theorem (Hornik, 1991), which proves that NNs can approximate arbitrary functions has been well known in the previous century.

The most important aspect that has been changed since DL, is that networks with multiple layers have become possible. On one side, this is thanks to the improved performance of available hardware. Without solutions for the vanishing gradient problem (Hochreiter, 1991), however, the better hardware would not help much. By initialising the weights in a smarter way and using better activation functions, DL got where it is today.

This does not mean that all problems are resolved. Recently, the problem of internal covariate shift has been addressed (Ioffe and Szegedy, 2015). Because the changes in distributions negatively affect the learning of the network, several techniques to counter this problem have been suggested. The general approach of most of these techniques is to normalise the data in each layer to counter the changes in the distribution.

SNNs take a different approach in tackling the internal covariate shift. Rather than normalising the data, the layers in a SNN seek to keep the moments of the data constant. This is done by analysing how layers in the simplest of networks, i.e. multilayer perceptrons (MLPs), affect the moments of the data passing through.

In this thesis, we extend this analysis of moments to the backward pass of SNNs. This allows us to verify that weight updates in a SNN do not break the property of self-normalisation. Additionally, we find that by initialising the weights so that they have a variance of $\frac{1}{n}$, with n the number of inputs to the layer, the architecture influences the back propagation of the error signal. Finally, we also find a correlation between learning speed and the amount of variance in updates for lower layers.

The rest of this thesis is organised as follows. In chapter 2, we give a short overview of machine learning with a focus on supervised techniques. Chapter 3 provides a similar overview for NNs with a focus on MLPs. The problem of internal covariate shift and SNNs are introduced in chapter 4. We also describe our analysis of the moment dynamics in this chapter. We present some experiments to empirically verify our theoretical results in chapter 5. At the end of this thesis, we summarise our conclusions in chapter 6.

2 Machine Learning

Machine learning (ML) is the field that studies algorithms that allow computers to learn. It is generally considered a sub-field of artificial intelligence (AI) where learning has been a fundamental concept since the very beginning (Turing, 1950). Unlike good old-fashioned AI (GOFAI), however, also non-symbolic approaches have been welcomed to machine learning (ML) in the hope of enriching the field (Langley, 2011). This caused a shift from deductive learning in the symbolic paradigm towards approaches that learn inductively from data. Also the quest for intelligence in machines has been left aside. Instead, ML builds upon results from statistics, probability theory and optimization to improve performance on well-defined tasks (Langley, 2011).

The goal of this chapter is to provide some insight in the rich field of machine learning. A quick overview can be found in section 2.1. Subsequent chapters introduce the problem of learning from labelled data (section 2.2.1) and how to solve it in a practical way (section 2.3). The errors that can arise while learning from labelled data are discussed in section 2.4. Finally, techniques to assess the errors that have been made during learning can be found in section 2.5.

2.1 Definition and Taxonomy

Samuel described ML as the study 'concerned with the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning' Samuel, 1959. This describes the purpose of ML precisely, but does not quite tell when a program can be considered to learn. Samuel does, however, talk about programs that learn from experience (Samuel, 1959).

This idea can also be found in a more formal definition, provided by Mitchell:

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. Mitchell, 1997

Note that this defines the learning problem rather than the program. After all, there can be multiple programs that use the same experience for the same tasks and are evaluated with the same performance measure.

ML is thus about finding new and improving upon old algorithms for well-defined learning problems. A generic ML algorithm consists of a useful representation for the experience, some cost function to approximate the performance measure and a model that is able to capture the progress of learning (Goodfellow, Bengio and Courville, 2016). To this end, ML uses ideas and techniques from fields like information theory, optimisation and statistics.

2.1.1 Taxonomy

Learning problems and their algorithms can be grouped according to the experience they learn from. The easiest problems are those for which the experience is a fixed set of data. The two main scenarios in this case are *supervised* and *unsupervised* learning. When the dataset consists of inputs and corresponding output-labels, supervised techniques can be used to retrieve a mapping from inputs to outputs. If no labels are provided in the examples, unsupervised learning can be used to retrieve information from the data. When only a few examples from the dataset are labelled, the term semi-supervised learning can be used.

There are also ML algorithms that do not require a fixed dataset for learning. The most important group of such algorithms is known as *reinforcement learning*. In this case, the experience comes from a function that assigns a reward to the outputs of the system (Russell and Norvig, 2010). Variants of this learning paradigm that are less common include transductive inference as well as on-line and active learning (Mohri, Rostamizadeh and Talwalkar, 2012).

One field that is very closely related to ML is *statistical learning*. This subfield of statistics takes a more formal approach in studying problems and methods that involve learning from data (James et al., 2013). Another field that has a strong connection with ML is *computational learning theory*. It provides theoretical foundations for problems and algorithms in both statistical and machine learning (Russell and Norvig, 2010).

2.2 Supervised Learning

From the different ML settings discussed in section 2.1.1, supervised learning has the most solid theoretical foundations (Mohri, Rostamizadeh and Talwalkar, 2012). For this reason, the rest of this chapter is mainly dedicated to learning from labelled data. This section provides the basic foundation on which most of supervised learning has been built.

2.2.1 Labelled Data

A labelled dataset

$$\mathcal{D} = \{(X_1, Y_1), (X_2, Y_2), \dots, (X_p, Y_p)\}$$

consists of feature vectors $X_i \in \mathcal{X}$ and labels $Y_i \in \mathcal{Y}$. In case that both input and output space consist of vectors, i.e. $\mathcal{X} \subseteq \mathbb{R}^m$ and $\mathcal{Y} \subseteq \mathbb{R}^n$, the dataset can be represented by means of matrices where each row represents one example. Concretely, we consider the matrices

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_p \end{bmatrix} \in \mathbb{R}^{p \times m} \qquad \mathbf{Y} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_p \end{bmatrix} \in \mathbb{R}^{p \times n}$$

for inputs and outputs respectively.

If the labels are numerical, e.g. $\mathcal{Y} = \mathbb{R}^n$, the learning problem is referred to as *regression*. The term *classification* is used for problems for which the output space is a finite set of values or classes, e.g. $\mathcal{Y} = \{-1, 1\}$. Note that although it is easy to generate all possible inputs, collecting labels generally requires human input and is hard to automate (Goodfellow, Bengio and Courville, 2016).

Let us assume that the data has been generated by some $f : \mathbb{R}^m \to \mathcal{Y}$. Given that there is some random, zero-mean noise ϵ in the available data, the relation between inputs and labels can be written as

$$Y = f(\boldsymbol{x}) + \epsilon\,. \tag{2.1}$$

Supervised ML is about finding an estimate — sometimes also called *hypothesis* — h for the hidden function f from the examples in a labelled dataset $\mathcal{D}$ to make predictions on future data, i.e. feature vectors that are not contained in the available dataset (Russell and Norvig, 2010).

2.2.2 Risk and Loss

The *generalisation error* or *risk* for a hypothesis h is defined as

$$R(h) = \mathrm{E}_{\boldsymbol{x}}\left[C(Y, \hat{Y})\right] = \mathrm{E}_{\boldsymbol{x}}\left[C(f(\boldsymbol{x}) + \epsilon, h(\boldsymbol{x}))\right]\,, \tag{2.2}$$

where $Y = f(\boldsymbol{x}) + \epsilon$ is the provided label in the data, $\hat{Y} = h(\boldsymbol{x})$ the predicted label and C is a loss function. In words, the generalisation error is the average loss that is to be expected when using hypothesis h to predict labels.

Considering that the goal of ML is prediction, the optimal estimate h^* for f is the hypothesis that leads to the smallest risk, i.e.

$$h^* = \arg\min_h R(h)\,. \tag{2.3}$$

A *cost* or *loss function* $C : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ is a measure for computing how far apart two predictions are in the output space $\mathcal{Y}$. It is a measure for the quality of a predicted label w.r.t. the true target label for a specific feature vector. A simple, well-known example loss function for classification is the zero-one loss

$$C_{\text{zo}}(Y, \hat{Y}) = \begin{cases} 0 & Y = \hat{Y} \\ 1 & Y \neq \hat{Y} \end{cases}\,. \tag{2.4}$$

The expected value of the zero-one loss equals the number of times the predictor predicted the wrong class and is therefore closely related to the error rate.

In practice, however, classes are often encoded with a one-hot coding, where each entry in the output vector represents the probability for one of the possible classes. This means that $\mathcal{Y} = [0, 1]^n$ if there are n classes. In this setting, the cross-entropy loss

$$C_{\text{ce}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{n} y_i \ln(\hat{y}_i) \tag{2.5}$$

is a popular choice. In case of regression, the squared loss

$$C_{\mathrm{sq}}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.6}$$

generally proves useful.

2.2.3 Bayes Classifier

For a binary classification task, evaluated with zero-one loss (2.4), the risk is minimised by the so-called *Bayes optimal classifier* (James et al., 2013)

$$h^*_{\mathrm{zo}}(\boldsymbol{x}) = \begin{cases} 1 & p(Y = 1 \mid \boldsymbol{x}) > p(Y = -1 \mid \boldsymbol{x}) \\ -1 & p(Y = 1 \mid \boldsymbol{x}) < p(Y = -1 \mid \boldsymbol{x}) \end{cases} . \tag{2.7}$$

The generalisation error for this classifier is given by

$$\begin{aligned} R(h^*_{\mathrm{zo}}) &= \int_{\mathbb{R}^m} p(\boldsymbol{x}, Y \neq h^*_{\mathrm{zo}}(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} \\ &= \int_{\mathbb{R}^m} p(Y \neq h^*_{\mathrm{zo}}(\boldsymbol{x}) \mid \boldsymbol{x}) p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \,. \end{aligned}$$

Here, $p(Y \neq h^*_{\mathrm{zo}}(\boldsymbol{x}) \mid \boldsymbol{x})$ is the minimum of $p(Y = -1 \mid \boldsymbol{x})$ and $p(Y = 1 \mid \boldsymbol{x})$ due to the definition of h^*_{zo} (2.7). Although this solution is provably optimal, it is impossible to compute. After all, this would require knowledge about the distribution $p(Y \mid \boldsymbol{x})$, which is not available in practice (James et al., 2013).

2.3 Practical Risk Minimisation

Although the solution to supervised learning is as simple as minimising the risk, it is generally impossible to express the risk or its minimum in terms of known properties of the data. Therefore, an approximation of the generalisation error is minimised in practice, rather than the true risk. In this section, we list the two most commonly known techniques to minimise the risk practically.

2.3.1 Empirical

The easiest technique is known as empirical risk minimisation (ERM). The *empirical risk* is nothing more than the risk (2.2), where the expectation is taken over the available training data instead of over the entire input space. Mathematically, the empirical risk for a hypothesis h on a dataset with p independently and identically distributed (i.i.d.) examples is given by

$$\hat{R}(h) = \frac{1}{p}\sum_{i=1}^{p} C(Y_i, h(\boldsymbol{x}_i)) \,. \tag{2.8}$$

It can be shown that for an increasing number of available samples, $p \to \infty$, the empirical risk converges to the generalisation error (Mohri, Rostamizadeh and Talwalkar, 2012). A practical estimate for f can thus be found by minimising this empirical risk to get

$$\hat{h} = \arg\min_{h} \hat{R}(h)\,. \tag{2.9}$$

An estimate that results from (2.9), is not guaranteed to minimise the true risk (James et al., 2013). It would be easy to construct a function that predicts correct labels for all examples in the dataset, but produces wrong results everywhere else. This function would have a zero empirical risk, but probably a very high generalisation error. Apart from the poor prediction capabilities of this classifier, it is infeasible to consider all possible functions $\mathcal{X} \to \mathcal{Y}$ in order to find the estimate $\hat{h}$.

2.3.2 Hypothesis Space

In order to resolve the problems of bad generalisation and infeasibility in empirical risk minimisation (ERM) (see section 2.3.1), it suffices to limit the search space of hypotheses. Concretely, we consider a fixed hypothesis space $\mathcal{H}$ with hypotheses that are expected to generalise well (Goodfellow, Bengio and Courville, 2016). This generalisation can be enforced by choosing a space with hypotheses with a limited complexity.

Infeasibility could be tackled by using a finite hypothesis space, but this is often too much of a constraint. Therefore, spaces with parametrised functions are interesting in practice, since it is generally easier to estimate parameters than to fit an entire function (James et al., 2013). Often, a hypothesis space uses parameters to specify the structural form of its functions that can generally not be learnt. We will denote a hypothesis space with so-called *hyperparameters* θ by $\mathcal{H}_\theta$.

A typical example for a hypothesis space with a hyperparameter is the set of polynomial functions

$$\mathcal{H}_d = \{ f(x\,; \boldsymbol{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d \mid w_i \in \mathbb{R} \}\,.$$

In this case, the coefficients w_i are the parameters to be learnt and the degree of the polynomial d is the only hyperparameter. Note that the complexity of the hypotheses in $\mathcal{H}_d$ can be controlled by the hyperparameter d.

This makes ERM a two-phase process. The first phase is about choosing a hypothesis space and its hyperparameters. This process is commonly referred to as *model selection*. Once a hypothesis space or *model class* has been selected, the parameters for the final hypothesis or *model* can be optimised in the second phase. The search for parameters is known as *training* or *fitting* and comes down to minimising the empirical risk, i.e. the final model is given by

$$\hat{h}_{\mathcal{H}} = \arg\min_{h \in \mathcal{H}_\theta} \hat{R}(h)\,. \tag{2.10}$$

Note that the choice for $\mathcal{H}_\theta$ is crucial to find models that generalise well.

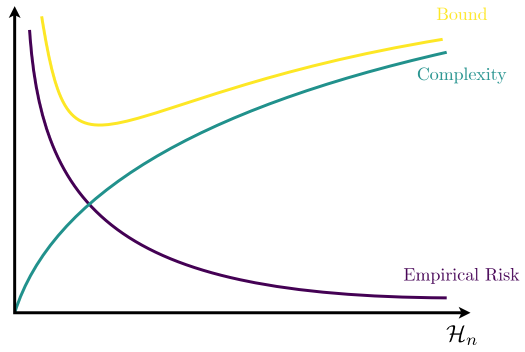


Figure 2.1: Illustration of the error bound that is minimised in SRM (Mohri, Rostamizadeh and Talwalkar, 2012).

2.3.3 Structural

Structural risk minimisation (SRM) is the best known alternative to ERM. It is based on the fact that the sum of the empirical risk and a complexity term for the hypothesis space $\mathcal{H}$ form an upper bound on the generalisation error (2.2) (Mohri, Rostamizadeh and Talwalkar, 2012). By minimising this upper bound over an infinite sequence of increasingly complex hypothesis spaces, i.e.

$$\mathcal{H}_1 \subset \mathcal{H}_2 \subset \ldots \subset \mathcal{H}_n \subset \ldots ,$$

SRM allows to do the model selection together with the training of the model. An illustration of SRM can be found in figure 2.1. The final hypothesis is given by

$$\hat{h} = \arg \min_{\substack{h \in \mathcal{H}_n \\ n \in \mathbb{N}}} \hat{R}(h) + \text{complexity}(\mathcal{H}_n, |\mathcal{D}|) \,. \tag{2.11}$$

Although it is theoretically possible to compute every part in SRM, it is rarely used in practice. After all, SRM requires to solve a possibly large number of ERM problems to get to a solution (Mohri, Rostamizadeh and Talwalkar, 2012). One type of practical ML models that is often considered to be an example of SRM, are support vector machines (SVMs). The reason for this is that a SVM minimises the training error together with a margin, which can be interpreted as a measure for the complexity of the model.

2.4 A Bunch of Errors

The true goal in supervised ML algorithms is to minimise the generalisation error. Because this turns out to be impossible, an approximation of the risk is minimised instead. This and other approximations that are used in the process of ERM form entry points for various distortions. A discussion of the consequences and origins of these errors is the topic of this section.

2.4.1 Excess Error

The first error is introduced by limiting the hypothesis space to some $\mathcal{H}_\theta$ in the first phase of ERM (see section 2.3.2). This limitation causes the best possible hypothesis to be

$$h^*_{\mathcal{H}} = \arg\min_{h \in \mathcal{H}_\theta} R(h), \tag{2.12}$$

rather than the optimal hypothesis h^* (2.3). The discrepancy between the risks of the suboptimal and the optimal hypotheses, i.e.

$$\mathcal{E}_{\text{app}} = \mathrm{E}\left[R(h^*_{\mathcal{H}}) - R(h^*)\right], \tag{2.13}$$

is known as the *approximation error* (Mohri, Rostamizadeh and Talwalkar, 2012).

A second error is introduced by using the empirical risk as an estimate for the generalisation error. The expected difference in performance between the result of ERM (2.10) and the suboptimal (2.12) hypothesis, i.e.

$$\mathcal{E}_{\text{est}} = \mathrm{E}\left[R(\hat{h}_{\mathcal{H}}) - R(h^*_{\mathcal{H}})\right], \tag{2.14}$$

is the *estimation error* (Mohri, Rostamizadeh and Talwalkar, 2012).

The third error comes from the fact that minimisation procedures rarely lead to exact solutions. Instead, the result is rather a sufficiently good approximation $\tilde{h}_{\mathcal{H}}$ so that

$$\hat{R}(\tilde{h}_{\mathcal{H}}) < \hat{R}(\hat{h}_{\mathcal{H}}) + \rho$$

for some tolerance $\rho \geq 0$. The *optimisation error* is then given as

$$\mathcal{E}_{\text{opt}} = \mathrm{E}\left[R(\tilde{h}_{\mathcal{H}}) - R(\hat{h}_{\mathcal{H}})\right], \tag{2.15}$$

which should be comparable to the tolerance ρ in magnitude (Bottou and Bousquet, 2007).

The total error that is made in ERM is called the *excess error*,

$$\mathcal{E} = \mathrm{E}\left[R(\tilde{h}_{\mathcal{H}}) - R(h^*)\right], \tag{2.16}$$

and is simply the sum of the approximation, estimation and optimisation error, i.e.

$$\mathcal{E} = \mathcal{E}_{\text{opt}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{app}}.$$

There exist problems for which it is feasible to do the optimisation exactly. We talk about *small-scale* learning when $\mathcal{E}_{\text{opt}} = 0$. However, *large-scale* problems, for which $\mathcal{E}_{\text{op}} > 0$, are more common in ML (Bottou and Bousquet, 2007).

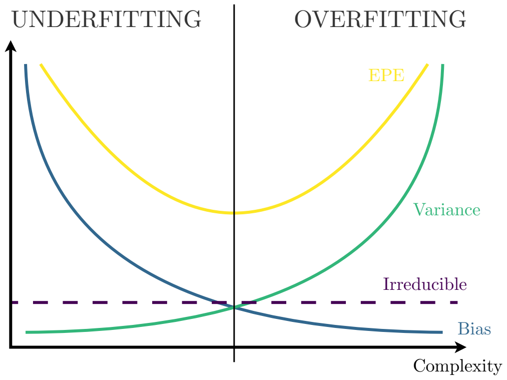


Figure 2.2: Illustration of bias variance decomposition together with the phenomena of under- and overfitting. Inspired by figure in (Goodfellow, Bengio and Courville, 2016)

2.4.2 Bias vs Variance

Another way of looking at the errors that arise in ERM is the decomposition of the expected prediction error (EPE) of the squared loss. The EPE is the expected loss of the prediction in some unseen point, i.e.

$$\mathrm{EPE}(\hat{h}, \boldsymbol{x}_0) = \mathrm{E}\left[C_{\mathrm{sq}}\left(f(\boldsymbol{x}_0) + \epsilon, \hat{h}(\boldsymbol{x}_0)\right)\right] ,$$

in case of the squared loss (2.6).

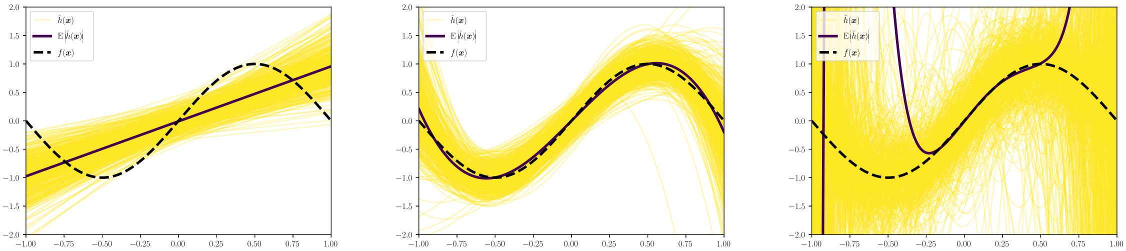
By using simple rules of expectation and the fact that $\mathrm{E}[\epsilon] = 0$ and $\mathrm{Var}[\epsilon] = \sigma_\epsilon^2$, the EPE can be decomposed as follows

$$\begin{aligned}
\mathrm{EPE}(\hat{h}, \boldsymbol{x}_0) &= \mathrm{E}\left[\left(\hat{h}(\boldsymbol{x}_0) - (f(\boldsymbol{x}_0) + \epsilon)\right)^2\right] \\
&= \mathrm{E}\left[\left(\hat{h}(\boldsymbol{x}_0) - \mathrm{E}\left[\hat{h}(\boldsymbol{x}_0)\right] + \mathrm{E}\left[\hat{h}(\boldsymbol{x}_0)\right] - f(\boldsymbol{x}_0)\right)^2\right] + \mathrm{E}\left[\epsilon^2\right] \\
&= \mathrm{E}\left[\left(\hat{h}(\boldsymbol{x}_0) - \mathrm{E}\left[\hat{h}(\boldsymbol{x}_0)\right]\right)^2\right] + \mathrm{E}\left[\left(\mathrm{E}\left[\hat{h}(\boldsymbol{x}_0)\right] - f(\boldsymbol{x}_0)\right)^2\right] + \mathrm{Var}\left[\epsilon\right] \\
&= \mathrm{Var}\left[\hat{h}(\boldsymbol{x}_0)\right] + \left(\mathrm{E}\left[\hat{h}(\boldsymbol{x}_0)\right] - f(\boldsymbol{x}_0)\right)^2 + \sigma_\epsilon^2 ,
\end{aligned}$$

where the expectations are computed over the dataset that was used to train $\hat{h}$.

The EPE is the sum of the variance of the hypotheses, the bias $\mathrm{E}\left[\hat{h}\right]$ of the predictions and the variance of the noise in the data. This formulation is therefore best known as the *bias-variance decomposition.* The bias,

$$\mathrm{E}\left[\hat{h}(\boldsymbol{x}_0)\right] - f(\boldsymbol{x}_0) ,$$



(a) $d = 1$ (b) $d = 3$ (c) $d = 10$

Figure 2.3: Illustration of bias and variance for the polynomial hypothesis space, $\mathcal{H}_d$, from section 2.3.2. The bias for each point is indicated by the distance between the darker, solid line and the black, dashed line. The variance is encoded by the area covered by the lighter, solid lines. The degree, d, of the polynomials is a measure for the complexity of $\mathcal{H}_d$. Inspired by figure in (James et al., 2013)

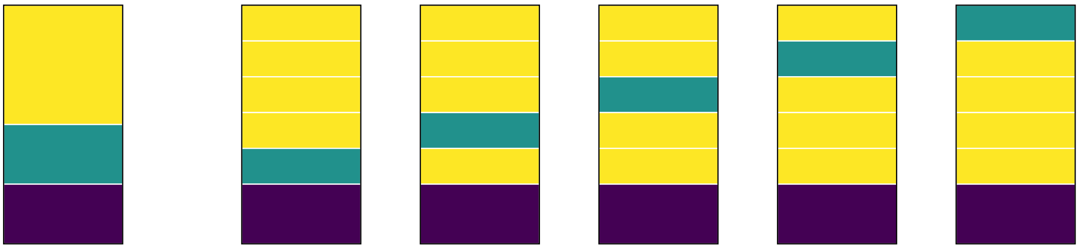
indicates how far predictions of $\hat{h}$ are expected to be from the truth. A high bias indicates that no hypothesis can be found that comes close enough to f. This is generally the case if the hypothesis space is not complex enough, in which case the problem is said to suffer from *underfitting*.

By increasing the complexity, the model is normally able to adapt better to the specific data and therefore the bias decreases. The variance of the hypotheses, on the other hand, increases with the complexity. The reason for this is that complex hypothesis spaces allow to model the smallest changes in the data, which can lead to very different predictions for slightly altered data. When the hypothesis space is so complex that trained hypotheses are very susceptible to noise, the model is said to suffer *overfitting*.

Because the noise can not be changed, the variance of the noise is often referred to as the irreducible error. This variance that is inherent to the data, provides a lower bound on the EPE. In other words, the prediction accuracy can only be improved by reducing both the bias and variance of the hypothesis space up to a certain point. From the diagram in figure 2.2, it should be clear that this is an act of balancing the complexity of the model. An illustration of bias and variance in a practical example is provided in figure 2.3.

2.5 Risk assessment

More important than the knowledge that the process of ERM introduces several errors, is the possibility to get an idea of the magnitude of the excess error. This justifies the addition of a third phase in the process of ERM: risk assessment. In this section, we tackle some well-known techniques to get an idea of how good a hypothesis actually is. The general idea is to withhold some part of the data from training and use this part as unseen data to get an idea of the predictive performance of a model.



(a) test set

(b) 5-fold cross validation

Figure 2.4: Illustration of possible splits when using a test set or cross validation. The part with the brightest colour is the data used for training the model, the darkest colour indicates the test set and the rest make up the validation set. The 5 blocks in the illustration of cross validation represent exactly the same dataset. Note that, in the illustration, cross validation is only used for model selection. For the risk assessment, the test set method is used.

2.5.1 Test Set

The easiest way to do quality assessment is to use the so-called *test set method*, in which the available data $\mathcal{D}$ is split into two parts. The first part of the data is used to train a model and is therefore called the *training set*. The second part is used to test the predictive performance of the trained model and is known as the *test set*. When splitting the data, it is important that both training and test set have approximately the same distribution as the original data (Goodfellow, Bengio and Courville, 2016).

In order to assess the generalisation error by means of the test set, it suffices to compute the empirical risk (2.8) over the test set once learning has finished. This leads to a good estimate for the risk, provided that the model was not influenced by any information from the test set (Goodfellow, Bengio and Courville, 2016). As soon as the slightest bit of information is used for pre-processing, model selection or training, the estimate for the risk will look better than it actually is. Therefore, it is generally a good idea to lock away the test set until the model is fully trained. Note that when applied correctly, the test set is used only once on a single fully-trained model.

During model selection, risk assessment is often a useful idea to find out if the hypotheses in a space $\mathcal{H}_n$ suffer from under- or overfitting. As mentioned above, the test set can not be used for this purpose. Therefore, the training set is often split into two parts once more to get a new, smaller set of training samples and a so-called *validation set*. Again, the empirical risk (2.8) is computed over the validation set to get an idea of how well each of the hypothesis spaces generalise. Rules for how the data should be partitioned into training, validation and test sets depend on the problem and the available data (James et al., 2013).

2.5.2 Cross Validation

One of the main disadvantages of the test set method is that the entire test set is only used once. Also the validation set is used only a limited number of times. On top of that, the choice of how the data is partitioned can affect the resulting risk estimates (James et al., 2013). Both of these issues can be alleviated by using a technique called *cross validation*. k-fold cross validation involves the partitioning of the data $\mathcal{D}$ in k disjoint sets, also known as *folds*. Each fold is used as a test or validation set once while the other $k-1$ folds are used to train the model.

In practice, this requires each model to be trained k times. This leads to k different hypotheses of the form

$$\hat{h}_{\text{CV},i} = \arg\min_{h \in \mathcal{H}_\theta} \frac{1}{p - |\mathcal{D}_i|} \sum_{(\boldsymbol{x},Y) \in \mathcal{D} \setminus \mathcal{D}_i} C(Y, h(\boldsymbol{x})) \,, \tag{2.17}$$

where $p = |\mathcal{D}|$ and $\mathcal{D}_i$ represents fold i of k. The estimate for the risk is then computed by averaging over test or validation errors of the k different hypotheses, i.e.

$$\hat{R}_{\text{CV}}(\mathcal{H}_\theta) = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{|\mathcal{D}_i|} \sum_{(\boldsymbol{x},Y) \in \mathcal{D}_i} C\left(Y, \hat{h}_{\text{CV},i}(\boldsymbol{x})\right) \,. \tag{2.18}$$

Note that cross validation provides an average risk estimate for all hypotheses in $\mathcal{H}_\theta$, rather than for one specific hypothesis. Also, each of the hypotheses $\hat{h}_{\text{CV}}$ that are trained in the process of cross validation are biased towards the folds they have been trained on. This makes these hypotheses unsuited to serve as final models and renders cross-validation impractical for assessing the quality of some final model. For the purpose of model selection, however, cross validation can be a very useful tool to resolve the need for a separate validation set.

3 Neural Networks

Neural networks (NNs) are the result of connecting simple computational units, known as *neurons*. In biology, a neuron is just a specific kind of cell that can be found in the brain (Raven et al., 2014). In order to get a better understanding of the brain, researchers created mathematical abstractions for these neurons and their connections (McCulloch and Pitts, 1943). These ideas found their way into the field of ML, where they became a powerful tool for learning (Schmidhuber, 2015).

Currently, artificial neural networks (ANNs) are one of the most popular and successful ML methods due to state-of-the-art performance on various tasks in e.g. computer vision (He et al., 2016), speech recognition (Amodei et al., 2016) and natural language processing (Wu et al., 2016). Another important reason for the popularity of ANNs is the fact that they are representation learning methods (LeCun, Bengio and G. E. Hinton, 2015). This means that a NN is capable of finding useful features from raw data for the task at hand. This alleviates the need for domain expertise and careful feature engineering, which is one of the main challenges for a lot of other methods.

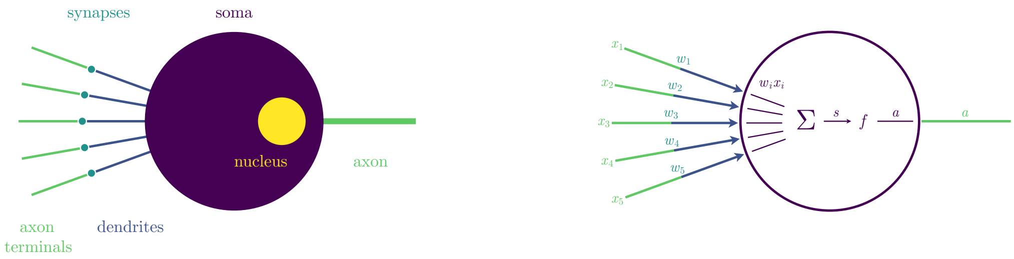
In this chapter, we give an introduction on how ANNs can be useful for supervised learning. The fundamental structure of NNs and the notation for the different entities in these networks are introduced in section 3.1. Some examples of some practical, more advanced networks is provided in 3.2. How these models are able to learn from labelled data is explained in sections 3.4 and 3.5.

3.1 Basic Structure

The basic structure of artificial neural networks (ANNs) is based on the human brain, which is an example of a biological neural network (BNN). The goal of this section is to reveal the similarities as well as the differences in the structure of these two types of NNs. We will especially focus on multilayer perceptrons (MLPs), which is a specific type of ANN, because it is the easiest kind of networks to study. The notation that is introduced here, and used in the rest of this thesis, might therefore not be ideal to describe more complex models.

3.1.1 Neurons

Each neuron in the brain is a cell with a *soma* or cell body, and a nucleus (Raven et al., 2014). *Dendrites* are branched extensions to the soma that allow neurons to receive information from different sources. The surface of the soma integrates the incoming impulses from the dendrites and triggers an impulse by itself if the excitation from the



(a) BNN (b) ANN

Figure 3.1: Schematic illustrations of neurons in a biological neural network and a artificial neural network with an emphasis on the structural similarities.

integrated impulses reaches a certain threshold. This triggered impulse is broadcast to dendrites of other neurons by means of the *axon*. *Synapses* form the connections between *axon terminals* and dendrites and are believed to be the basis of learning and memory (Raven et al., 2014). These components are schematically illustrated in figure 3.1a.

Neurons in a ANN, on the other hand, are rather mathematical functions of the form (Russell and Norvig, 2010)

$$\text{neuron}_f : \mathbb{R}^n \times \mathcal{W} \to \mathbb{R} : \boldsymbol{x}\,; W \mapsto a = \text{neuron}_f(\boldsymbol{x}\,; W)\,.$$

Each of these neurons receives a vector $\boldsymbol{x}$ of inputs x_i and computes a single *activation* a as output. So-called *synaptic weights* $w \in W$ are used to weight the incoming signals from other neurons. f is the so-called *activation function* of the neuron and is responsible for computing the outgoing activations, i.e. f models the threshold logic. A visual representation of the elements in an artificial neuron is given in figure 3.1b.

A simple, concrete example of a neuron is

$$\text{neuron}_f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} : \boldsymbol{x}\,; \boldsymbol{w} \mapsto a = \text{neuron}_f(\boldsymbol{x}\,; \boldsymbol{w})\,. \tag{3.1}$$

In this case, each input has a corresponding weight to indicate the strength of its connection. By computing the dot product

$$s = \boldsymbol{w} \cdot \boldsymbol{x} = \sum_{i=1}^{n} w_i x_i\,,$$

the weighted inputs can be integrated and the activation of the neuron is given by

$$a = \text{neuron}_f(\boldsymbol{x}\,; \boldsymbol{w}) = f(s) = f(\boldsymbol{w} \cdot \boldsymbol{x})\,.$$

This neuron has been used to visualise the ANN in figure 3.1 with $n = 5$.

3.1.2 Layers with Bias

The simple neuron from (3.1) has exactly one weight for each input and is therefore not able to control the activation for zero inputs. A simple work-around is to add a constant input unit $x_0 = 1$ with a special weight $b = w_0$, which is known as the *bias*, i.e.

$$\mathrm{neuron}_f : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R} : \boldsymbol{x}\,; \boldsymbol{w}, b \mapsto a = \mathrm{neuron}_f(\boldsymbol{x}\,; \boldsymbol{w}, b)\,. \tag{3.2}$$

The only difference in the computation of the activations is that the bias b appears in the integration of the weighted inputs:

$$s = \boldsymbol{w} \cdot \boldsymbol{x} + b = \sum_{i=1}^{n} w_i x_i + b\,.$$

This simple modification allows to shift the activation function left and right, which allows for a more flexible threshold.

In ANNs, neurons are typically organised in layers. Each layer can be seen as a vector of neurons with the same activation function. Concretely, given a neuron of the form (3.1.1), we can define a layer of *width* N, i.e. with N neurons, as

$$\mathrm{layer}_{f,N} : \mathbb{R}^n \times \mathcal{W} \to \mathbb{R}^N : \boldsymbol{x}\,; W \mapsto \boldsymbol{a} = \mathrm{layer}_{f,N}(\boldsymbol{x}\,; W)\,.$$

Note that a layer is in fact a parametrised transformation from $\mathbb{R}^n$ to $\mathbb{R}^N$ (Goodfellow, Bengio and Courville, 2016).

A layer with neurons of the form (3.2) would look like

$$\mathrm{layer}_{f,N} : \mathbb{R}^n \times \mathbb{R}^{N \times n} \times \mathbb{R}^N \to \mathbb{R}^N : \boldsymbol{x}\,; \mathbf{W}, \boldsymbol{b} \mapsto \boldsymbol{a} = \mathrm{layer}_{f,N}(\boldsymbol{x}\,; \mathbf{W}, \boldsymbol{b})\,. \tag{3.3}$$

Instead of computing the dot product

$$s_i = \sum_{j=1}^{n} w_{ij} x_j + b_i$$

for each neuron in the layer explicitly, the integrated inputs can be computed at once by using a single dot product

$$\boldsymbol{s} = \mathbf{W} \cdot \boldsymbol{x} + \boldsymbol{b}\,.$$

The final activations are the result of applying the activation function element-wise to $\boldsymbol{s}$, i.e.

$$\boldsymbol{a} = \mathrm{layer}_{f,N}(\boldsymbol{x}\,; \mathbf{W}, \boldsymbol{b}) = f(\boldsymbol{s}) = f(\mathbf{W} \cdot \boldsymbol{x} + \boldsymbol{b})\,,$$

where $a_i = f(s_i)$.

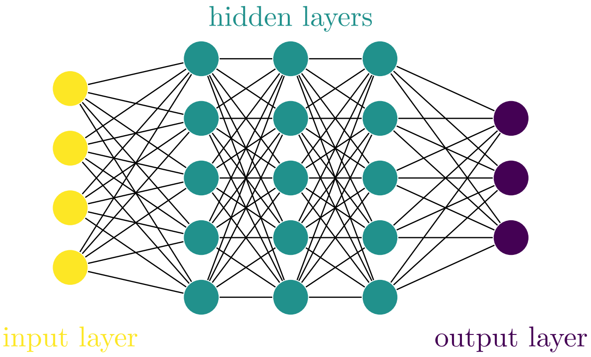


Figure 3.2: Graphical representation of a simple multilayer perceptron with an input, output and hidden layers. According to our notation, this network has a depth of $L = 4$ (three hidden and one output layer). The number of units in each layer are given by $N_0 = 4, N_1 = N_2 = N_3 = 5, N_4 = 3$.

3.1.3 A Network of Connected Layers

The result of stacking different layers on top of each other, is a ANN. Formally, we can write

$$\mathrm{net}_\theta : \mathbb{R}^n \times \mathcal{W} \to \mathbb{R}^m : \boldsymbol{x} \mapsto \boldsymbol{y} = \mathrm{net}_\theta(\boldsymbol{x}\,; W)\,, \tag{3.4}$$

where θ is the set of hyperparameters and W is the set of parameters for the NN.

A network with L layers is said to be of depth L and when using layers of the form (3.3), a concrete example for a ANN is given by

$$\begin{aligned}\mathrm{net}_\theta(\boldsymbol{x}\,; W) &= \mathrm{layer}_{f_L,N_L} \circ \mathrm{layer}_{f_{L-1},N_{L-1}} \circ \ldots \circ \mathrm{layer}_{f_2,N_2} \circ \mathrm{layer}_{f_1,N_1}(\boldsymbol{x}) \\ &= \mathrm{layer}_{f_L,N_L}\left(\ldots\ \mathrm{layer}_{f_1,N_1}\left(\boldsymbol{x}\,; \mathbf{W}^{(1)}, \boldsymbol{b}^{(1)}\right)\ \ldots\,; \mathbf{W}^{(L)}, \boldsymbol{b}^{(L)}\right)\,.\end{aligned}$$

In this case, the set of hyperparameters would be $\theta = \{L, f_1, N_1, f_2, N_2, \ldots, f_L, N_L\}$ and $W = \{\mathbf{W}^{(1)}, \boldsymbol{b}^{(1)}, \mathbf{W}^{(2)}, \boldsymbol{b}^{(2)}, \ldots, \mathbf{W}^{(L)}, \boldsymbol{b}^{(L)}\}$ are the parameters to be learnt. A network with this architecture is illustrated in figure 3.2.

Because the input layer does not apply any transformation to the data, we have ignored it in our formulation. It can be useful to refer to the input layer, however. Therefore, we reserve the layer-index zero to refer to the input layer: e.g. the number of inputs of the network in (3.4) is $N_0 = n$. The last layer (L) in the network is commonly referred to as the output layer. The layers that are not at the input or at the output, i.e. layer 1 to $L - 1$ are called *hidden layers*. These three different kinds of layers in a network are illustrated in figure 3.2.

3.2 Alternative Architectures

Although we depicted NNs as one hypothesis space in the introduction of this chapter, it can also be seen as a collection of different hypothesis spaces. Because neurons and layers can be made arbitrarily complex and combined in numerous ways, there exists a wide variety of possible architectures of ANNs. The architecture with *fully connected layers* (3.3) from section 3.1, corresponds to a multilayer perceptron (MLP) and is one of the easiest possible networks (Goodfellow, Bengio and Courville, 2016).

If there are no connections to previous layers, the data can only flow from the input to the output layer. Networks that have no cycles in their connections, e.g. MLPs, are called feedforward neural networks (FNNs). The class of ANNs where cycles are allowed are known as recurrent neural networks (RNNs). By means of the cycles in a RNN, the neurons that are triggered by one input can influence activations for the next input, exhibiting memory-like behaviour (Graves, 2012).

In this section, we provide some examples of more complex architectures that are very popular for solving practical ML problems. Rather than to provide a thorough discussion of these architectures, the goal of this section is to provide some insights in what the possibilities are when moving away from simple MLPs.

3.2.1 Convolutional Neural Network

A famous example of a FNN that has been designed with a focus on image recognition tasks, is the convolutional neural network (CNN) (LeCun, Bottou, Bengio et al., 1998). On top of the fully connected layers that can be found in a MLP, CNNs make use of *convolutional* and *pooling* layers.

Convolutional layers compute convolutions of the incoming signal with a set of small kernels, instead of using the dot product in (3.3). This greatly reduces the number of parameters that need to be learnt. Additionally, the weights of the kernel are shared for different local patches in the input image, which enables better generalisation (Goodfellow, Bengio and Courville, 2016). Another advantage of the convolution operation is that it is able to take into account information from neighbouring pixels due to its local receptive fields.

Pooling layers reduce the size of the image by combining groups of neighbouring pixels. This is done by taking the maximum or average pixel value, for example. These techniques are known as max pooling and average pooling respectively. Other pooling strategies include the L_2 norm and weighted averaging, but many more exist (Goodfellow, Bengio and Courville, 2016).

Most CNNs are built using convolutional and pooling layers alternately, followed by one or more fully connected layers. In each convolutional layer, the image at the input is convolved with a set of kernels that make up the parameters of that layer. This results in a *feature map* for each kernel, which is the result of the convolution with that kernel. The output of the convolutional layer is the element-wise application of the activation function on that feature map.

By regularly inserting a pooling layer, the images become smaller and therefore the computational costs of the convolution decrease and the network becomes less variant to scaling. An alternative way to reduce the computational effort for the network is to use convolutional layers with larger strides (Springenberg et al., 2015).

3.2.2 Long Short-Term Memory

A popular example of a RNN architecture is the long short-term memory (LSTM) network (Hochreiter and Schmidhuber, 1997). LSTMs have particularly complex neurons, called *memory cells*, that have been designed to alleviate the vanishing gradient problem (see section 3.4.1). The key part of a memory cell is its *cell state*, which explicitly models the memory of a neuron.

Each cell also has a set of *gates* to manipulate the cell state and to control the activation. The most important gates are the input and output gate. The former controls whether the cell state is updated with the current input signal, whereas the latter decides whether its neuron should fire or not. Another useful gate is the forget gate (Gers, Schmidhuber and Cummins, 2000), which allows to reset the cell state and thus clears the memory of the neuron.

On top of the weights $\boldsymbol{w}$ and b that can be found in almost every neuron, e.g. (3.2), a memory cell also requires $\boldsymbol{w}_{\text{gate}}, b_{\text{gate}}$ for each of its gates. This means that the number of parameters is quadrupled in case all three of the above mentioned gates are used. Concretely, the additional parameters are $\boldsymbol{w}_{\text{in}}$ and b_{in}, for the input gate, $\boldsymbol{w}_{\text{out}}$ and b_{out} for the output gate and the forget gate requires $\boldsymbol{w}_{\text{forget}}$ and b_{forget}.

The output of a gate is generally computed by means of a logistic sigmoid (3.6), which is a value between zero and one. By multiplying a signal with the output of a gate, it can be blocked, i.e. set to zero, if the output is close to zero or flow through if the gate output is close to one. The input gate is used to control the flow of the weighted sum of inputs. The output gate decides whether the current cell state C_t makes it to the output of the cell. Finally, the forget gate allows to clear the cell state and therefore the memory of the cell.

Using the index t to denote the time step t of the signal, the activation of a memory cell at each time step can be computed by

$$\begin{aligned}
i_t &= \sigma\left(\boldsymbol{w}_{\text{in}} \cdot \boldsymbol{x}_t + b_{\text{in}}\right) \\
o_t &= \sigma\left(\boldsymbol{w}_{\text{out}} \cdot \boldsymbol{x}_t + b_{\text{out}}\right) \\
f_t &= \sigma\left(\boldsymbol{w}_{\text{forget}} \cdot \boldsymbol{x}_t + b_{\text{forget}}\right) \\
C_t &= f_t C_{t-1} + i_t h(\boldsymbol{w} \cdot \boldsymbol{x}_t + b) \\
a_t &= o_t g(C_t)\,.
\end{aligned}$$

Here, h and g are activation functions that do not act as gates. Note that we did not distinguish between forward and recurrent connections to keep this overview on LSTMs short. For a more in-depth look into LSTMs and RNNs, we refer to (Graves, 2012).

3.3 Hyperparameters

One of the main disadvantages of NNs is the high number of hyperparameters that have to be chosen. Only for the structure of a simple MLP, the number of layers and neurons in each layer, etc. have to be chosen. Additionally, a decision has to be made on which activation function to use in each layer and how to initialise the weights.

In general, a grid search or cross-validation (see section 2.5.2) is too expensive for ANNs. Especially when considering deep networks, these techniques require too much computational resources to be practical. A random hyperparameter search turns out to be one of the most efficient ways to find good hyperparameters (Bergstra and Bengio, 2012). Also Bayesian optimisation is an efficient, but still costly way to find good hyperparameters (Snoek, Larochelle and Adams, 2012).

In order to start a search for hyperparameters, possible values have to be known. In this section, we provide an overview of commonly used activation functions and weight initialisation strategies that can be used for configuring a ANN.

3.3.1 Activation Functions

Although the activation function of a layer could be anything, there are certain choices that are more common than others. A great example of a theoretically sound activation function that is not particularly practical, is the identity, i.e.

$$\mathrm{id}(x) = x \qquad \mathrm{id}'(x) = 1\,. \tag{3.5}$$

The identity function can be interesting to study the learning dynamics of a ANN (Saxe, McClelland and Ganguli, 2014), but the resulting model will always be a linear classifier, irrespective of the number of layers. Therefore, it is important to use non-linear activation functions to construct ANNs that can model more complex relations. Another interesting property for activation functions that the identity does have, is differentiability. This is particularly useful when the weights of the network are updated by some variant of gradient descent (section 3.4).

Based on the firing mechanism of neurons in a BNN, non-differentiable threshold functions were a popular choice in the early days (Russell and Norvig, 2010). Unfortunately these functions are not differentiable and therefore useless in gradient based learning. The family of sigmoid functions provided a set of differentiable, non-linear alternatives. The logistic sigmoid

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \qquad \sigma'(x) = \sigma(x)(1 - \sigma(x))\,, \tag{3.6}$$

which results in activations between zero and one, has been a popular choice for quite some time.

Due to its small derivatives and the fact that it is not zero-centred, the logistic sigmoid does not facilitate fast learning (LeCun, Bottou, Orr et al., 1998). The hyperbolic tangent

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \qquad \tanh'(x) = \left(1 - \tanh(x)^2\right) \tag{3.7}$$

has zero mean outputs and greater derivatives. Therefore, it is to be preferred over logistic sigmoids (LeCun, Bottou, Orr et al., 1998).

An interesting activation function for the purpose of classification is the softmax

$$\boldsymbol{\sigma}(\boldsymbol{x}) = \frac{\exp(\boldsymbol{x})}{\sum_j \exp(x_j)} \tag{3.8}$$

$$\boldsymbol{\sigma}'(\boldsymbol{x}) = \boldsymbol{\sigma}(\boldsymbol{x})(\mathbf{1} - \boldsymbol{\sigma}(\boldsymbol{x}))\,. \tag{3.9}$$

In some sense it is a generalisation of the logistic sigmoid for vectors. The output of a softmax is a vector of probabilities that sum up to one. This makes it especially useful as output activation function to predict one-hot encoded labels.

Another popular family of activation functions has been built around the rectified linear unit (ReLU)

$$\mathrm{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \qquad \mathrm{ReLU}'(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}\,. \tag{3.10}$$

ReLUs are biologically inspired and promote sparsity, which results in faster learning (Glorot, Bordes and Bengio, 2011). The hard zero for negative inputs with its zero derivative, however, can prevent the network from propagating errors (see section 3.4.2) and leads to a rather unfavourable distribution of activations for learning (LeCun, Bottou, Orr et al., 1998).

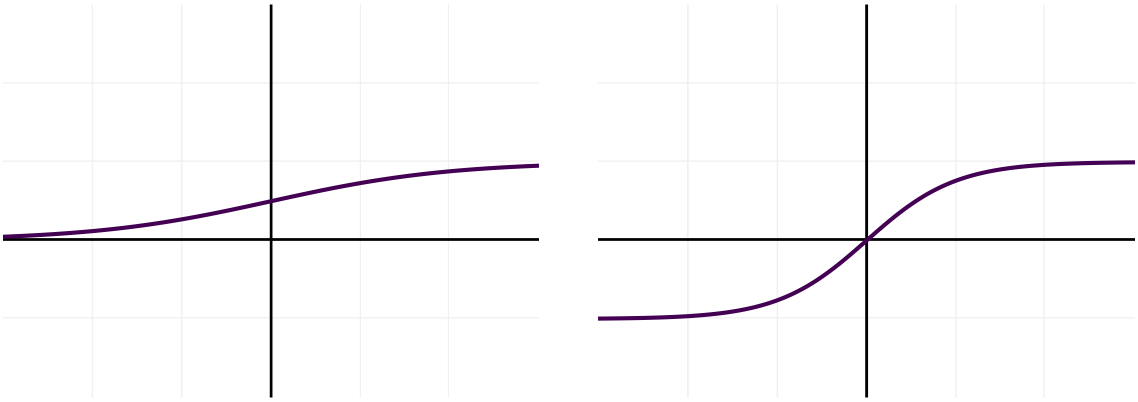
This is why several variants of ReLUs have been constructed. One activation function that allows the network to learn for negative input and has a more centred activation distribution, is the exponential linear unit (ELU) (Clevert, Unterthiner and Hochreiter, 2015)

$$\mathrm{ELU}(x\,;\alpha) = \begin{cases} \alpha(\exp(x) - 1) & x < 0 \\ x & x \geq 0 \end{cases} \quad \mathrm{ELU}'(x\,;\alpha) = \begin{cases} \mathrm{ELU}(x) + \alpha & x < 0 \\ 1 & x \geq 0 \end{cases}\,. \tag{3.11}$$

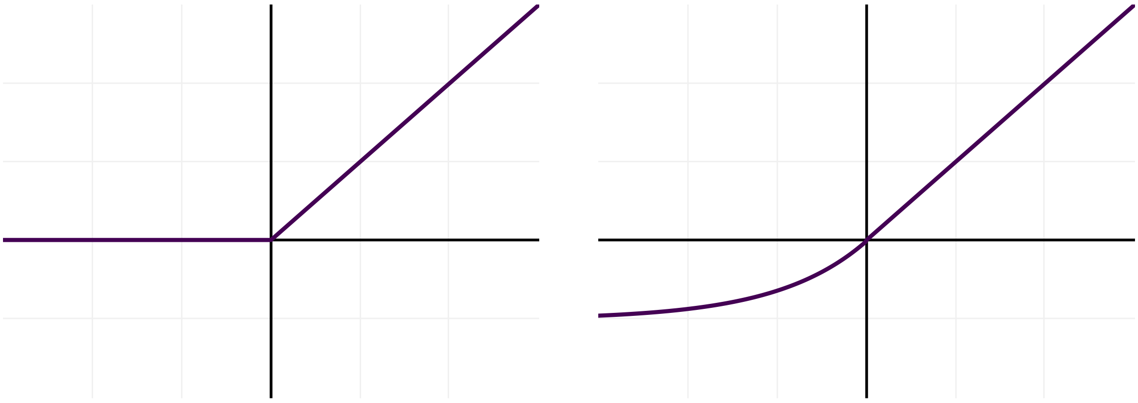
3.3.2 Weight Initialisation

When using an activation function of the sigmoid family, the greatest gradients appear for inputs close to zero. Because a small gradient can have negative effects on learning (see section 3.4.1), it is generally a good idea to initialise weights with values that are close to zero (LeCun, Bottou, Orr et al., 1998). It would not make sense to initialise all values with zero because this is equivalent to ignoring all inputs. Also using the same initial value for all weights does not lead to interesting networks. Therefore, the initial weights are generally small, randomly generated numbers. Although this is a good start, it turns out that careful initialisation is crucial for efficient learning in deep neural networks (DNNs).

The first successful deep architecture was a deep belief network (DBN) (G. E. Hinton, Osindero and Teh, 2006). Before this time, DNNs were often too hard to train. The key to the success of DBNs was the unsupervised pre-training. By stacking restricted Boltzmann machines (RBMs) that have been trained to reproduce their inputs, i.e.



(a) logistic sigmoid (b) hyperbolic tangent



(c) rectified linear unit (ReLU) (d) exponential linear unit (ELU)

Figure 3.3: Plots of some commonly used activation functions in ANNs.

without labels, a network can be built that already captures some of the structure of the data. Unsupervised pre-training can therefore be seen as an expensive, but effective way to initialise the weights in a ANN.

Xavier- or *Glorot-initialisation* offers a more efficient alternative to weight initialisation. It is the result of studying the moments of the activations and deltas (see section 3.4.2) in a network with symmetric activation functions that have derivative one at zero (Glorot and Bengio, 2010). The key of Xavier-initialisation is to generate weights $W^{(l)}$ from a distribution with zero mean and a variance inversely related to the number of inputs N_{l-1} and neurons in a layer N_l, i.e. so that

$$\mathrm{E}\left[W^{(l)}\right] = 0 \qquad\qquad \mathrm{Var}\left[W^{(l)}\right] = \frac{2}{N_{l-1} + N_l}\,.$$

By initialising the networks in this way, the variance of both activations and back-propagated gradients can be kept approximately constant.

He-initialisation offers an alternative initialisation method that is better suited for ReLUs. After all, a ReLU (3.10) is not symmetric and therefore it does not work well with Glorot-initialisation (He et al., 2015). More specifically, ReLUs annihilate the variance due to negative values, which is about half the variance of symmetrically distributed inputs. Therefore, He-initialisation suggests to initialise the weights so that

$$\mathrm{E}\left[W^{(l)}\right] = 0 \qquad\qquad \mathrm{Var}\left[W^{(l)}\right] = \frac{2}{N_{l-1}}\,.$$

Additionally, the biases $\boldsymbol{b}^{(l)}$ are initialised with zeros.

For both Xavier- en He-initialisation it does not matter what the actual distribution is, as long as it has the appropriate mean and variance. Generally, the weights are generated from a uniform or truncated normal distribution in order to prevent outliers for the initial values. Another interesting initialisation technique is to use orthogonal weight matrices. This has the interesting property that its transformation as well as its Jacobian preserve norm (Saxe, McClelland and Ganguli, 2014).

3.4 Learning

Although the structure of a NN forms the foundation of these models, it would not be interesting for the purpose of ML without a learning algorithm. The number one technique for training a network is backpropagation (BP) (Werbos, 1974), which is a form of gradient descent. It thanks its popularity to a study (Rumelhart, G. E. Hinton and Williams, 1986) that experimentally showed that meaningful patterns emerge in the hidden layers of a backpropagation (BP)-trained ANN (Schmidhuber, 2015). Although it can be used in unsupervised settings, (Baldi, 2012), BP is predominantly a supervised learning algorithm (see section 2.2).

The BP algorithm consists of two steps, or passes. In the *forward pass*, the network predictions $\hat{\boldsymbol{y}} = \mathrm{net}_\theta(\boldsymbol{x}\,; W)$ for each feature vector $\boldsymbol{x}$ are computed. In the *backward pass*, the gradients of the empirical risk (2.8) with respect to the weights are computed to update each weight. The forward pass is as simple as applying (3.4) on the data. In this section, we give an overview of the backward pass.

3.4.1 Product of Jacobians

An important part of BP is thus to compute the gradients

$$\frac{\partial C(\boldsymbol{y}, \hat{\boldsymbol{y}})}{w},$$

for all weights $w \in W$ in the network. Because $\hat{\boldsymbol{y}} = \mathrm{net}_\theta(\boldsymbol{x}\,; W)$, the chain rule gives

$$\frac{\partial C(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w} = \frac{\partial C(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial w},$$

which leaves us with the computation of the gradients of the network w.r.t. w. Because we defined a ANN to be a composition of layer functions (3.4), the chain rule applies again and leads to

$$\frac{\partial\, \mathrm{net}_\theta(\boldsymbol{x}\,; W)}{\partial w} = \frac{\partial \mathrm{layer}_{\theta_L}(\boldsymbol{x}^{(L)}\,; W_L)}{\partial \boldsymbol{x}^{(L)}} \frac{\partial \mathrm{layer}_{\theta_{L-1}}(\boldsymbol{x}^{(L-1)}\,; W_{L-1})}{\partial \boldsymbol{x}^{(L-1)}} \cdots \frac{\partial \mathrm{layer}_{\theta_l}(\boldsymbol{x}^{(l)}\,; W_l)}{\partial w},$$

assuming that w is a weight in layer l. This reveals that the gradient of the entire network w.r.t. a weight w is simply the product of derivatives of a set of layers.

There are different gradients to be considered for a fully-connected layer in a MLP (3.3). The first gradient is the Jacobian of the layer and is given by

$$\frac{\partial\, \mathrm{layer}_{f,N}(\boldsymbol{x}\,; \mathbf{W}, \boldsymbol{b})}{\partial \boldsymbol{x}} = \frac{\partial f(\boldsymbol{s})}{\partial \boldsymbol{s}} \frac{\partial (\mathbf{W} \cdot \boldsymbol{x} + \boldsymbol{b})}{\partial \boldsymbol{x}} = \mathrm{diag}(f'(\boldsymbol{s})) \cdot \mathbf{W}\,, \tag{3.12}$$

where $\cdot$ is the inner product and $\mathrm{diag}(\boldsymbol{x})$ is a diagonal matrix from the values in $\boldsymbol{x}$.

The gradients that are necessary to compute the weight updates in a MLP are given by

$$\frac{\partial\, \mathrm{layer}_{f,N}(\boldsymbol{x}\,; \mathbf{W}, \boldsymbol{b})}{\partial \mathbf{W}} = \frac{\partial f(\boldsymbol{s})}{\partial \boldsymbol{s}} \frac{\partial (\mathbf{W} \cdot \boldsymbol{x} + \boldsymbol{b})}{\partial \mathbf{W}} = f'(\boldsymbol{s}) \otimes \boldsymbol{x} \tag{3.13}$$

$$\frac{\partial\, \mathrm{layer}_{f,N}(\boldsymbol{x}\,; \mathbf{W}, \boldsymbol{b})}{\partial \boldsymbol{b}} = \frac{\partial f(\boldsymbol{s})}{\partial \boldsymbol{s}} \frac{\partial (\mathbf{W} \cdot \boldsymbol{x} + \boldsymbol{b})}{\partial \boldsymbol{b}} = f'(\boldsymbol{s})\,, \tag{3.14}$$

where $\otimes$ denotes the outer product.

The derivative of the entire MLP with respect to some weight w in layer l can then be written as a product of the Jacobians of layers $l+1$ up to L and the corresponding weight update

$$\frac{\partial\, \mathrm{net}_\theta(\boldsymbol{x}\,; W)}{\partial w} = \left(\prod_{i \in (L, L-1, \ldots, l+1)} \mathrm{diag}(f_i'(\boldsymbol{s}^{(i)})) \cdot \mathbf{W}^{(i)} \right) \cdot \frac{\partial \boldsymbol{a}^{(l)}}{\partial w}\,. \tag{3.15}$$

This formula reveals one fundamental problem of BP.

Rewriting (3.15) for a single entry at position i, j in the result of the matrix product above, cf. (Hochreiter, 1998), gives

$$\sum_{y_{L-1}=1}^{N_{L-1}} \cdots \sum_{y_{l+1}=1}^{N_{l+1}} \left[f'_L\left(s_i^{(L)}\right) w_{i,y_{L-1}}^{(L)} \left(\prod_{x=l+2}^{L-1} f'_x\left(s_{y_x}^{(x)}\right) w_{y_x,y_{x-1}}^{(x)} \right) f'_{l+1}\left(s_{y_{l+1}}^{(l+1)}\right) w_{y_{l+1},j} \right] .$$

This clearly shows that if $|f'(s_i)w_{ij}| < 1$ for all i and j in each layer, the entries in the result of the product can become very small. For DNN, where L can be very large, this causes the updates for lower layers to become approximately zero, hence the term *vanishing gradient problem* (Hochreiter, 1991). It is also possible that weights become too large and cause so-called *exploding gradients*.

3.4.2 Delta propagation

The equations above could be used to implement the backward pass in BP, but that would be rather inefficient. Instead of computing a product of Jacobians, the backward pass is generally implemented by means of the derivatives

$$\boldsymbol{\delta}^{(l)} = \frac{\partial C(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{s}^{(l)}} . \tag{3.16}$$

The first interesting aspect of this approach is that for certain pairs $(f_L(\cdot), C(\cdot, \cdot))$ of output activations and cost functions, the delta for the last layer can be simply computed as

$$\boldsymbol{\delta}^{(L)} = \hat{\boldsymbol{y}} - \boldsymbol{y} .$$

For the squared loss (2.6), this requires a linear output activation (3.5) and the cross-entropy loss (2.5) plays well with the logistic sigmoid (3.6) and softmax (3.8) activations. If the output activation and cost function do not make up such a pair, $\boldsymbol{\delta}^{(L)}$ is just the product of the derivatives of the loss and the output activation.

Starting from the last layer, the deltas in a MLP can be computed recursively, using

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)} \cdot \frac{\partial \boldsymbol{s}^{(l+1)}}{\partial \boldsymbol{s}^{(l)}} = \boldsymbol{\delta}^{(l+1)} \cdot \mathbf{W}^{(l+1)} \cdot f'(\boldsymbol{s}^{(l)}) , \tag{3.17}$$

which is once more a result that is derived from the chain rule. In order to compute the weight updates from the deltas, we can simply use

$$\frac{\partial \operatorname{layer}_{f_l,N_l}(\boldsymbol{x} \,; \mathbf{W}^{(l)}, \boldsymbol{b}^{(l)})}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \otimes \boldsymbol{x} \tag{3.18}$$

$$\frac{\partial \operatorname{layer}_{f_l,N_l}(\boldsymbol{x} \,; \mathbf{W}^{(l)}, \boldsymbol{b}^{(l)})}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)} . \tag{3.19}$$

3.5 Optimisation

There are several ways to use the gradients of the loss for updating the weights. The most straightforward way is to do gradient descent on the weights with a fixed step size or *learning rate* η, i.e.

$$w \leftarrow w - \eta \frac{\partial \hat{R}(\mathrm{net}_\theta)}{\partial w} \,. \tag{3.20}$$

This can also be considered a form of stochastic gradient descent (SGD) given that the empirical risk is already an estimate for the risk, which is generally the true goal. We will refer to this update as gradient descent, however, in order to distinguish between cases where not all examples of the training data are used to compute the gradient.

3.5.1 Variations on Gradient Descent

When there are too much examples, it is often computationally too costly to consider all samples at once. In this case, smaller groups of examples $\mathcal{B}_i \subseteq \mathcal{D}$ called *mini-batches*, are used to estimate the gradient of the empirical risk so that the update becomes

$$w \leftarrow w - \eta \nabla_{\mathcal{B}} C(\boldsymbol{y}, \hat{\boldsymbol{y}}) \,. \tag{3.21}$$

Here, we used $\nabla_{\mathcal{B}} C(\boldsymbol{y}, \hat{\boldsymbol{y}})$ to denote a gradient computed over a batch $\mathcal{B}$, i.e.

$$\nabla_{\mathcal{B}} C(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{|\mathcal{B}|} \sum_{(\boldsymbol{x}, \hat{\boldsymbol{y}}) \in \mathcal{B}} \frac{\partial C(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial w} \,.$$

When all mini-batches contain exactly one example, the algorithm is said to learn *on-line*. The other extreme, when there is only one mini-batch that contains all examples, i.e. $\mathcal{B} = \mathcal{D}$, is known as *batch* learning. Note that what we refer to as gradient descent is the same as batch learning, because $\hat{R}(\mathrm{net}_\theta) = \nabla_{\mathcal{D}} C(\boldsymbol{y}, \hat{\boldsymbol{y}})$.

Instead of only using the gradient of the loss function, learning can be accelerated by taking into account old gradients as well. In *momentum*-based SGD, this is done by computing an exponential moving average over the past gradients with coefficient μ (Goodfellow, Bengio and Courville, 2016). The momentum update is then computed as follows

$$\begin{aligned} v &\leftarrow \mu v - \eta \nabla_{\mathcal{B}} C(\boldsymbol{y}, \hat{\boldsymbol{y}}) \\ w &\leftarrow w + v \,, \end{aligned} \tag{3.22}$$

where v represents the average. This simple upgrade of SGD amplifies changes in low-curvature directions, which happens to have the same effect as second-order methods (Sutskever et al., 2013).

Nesterov momentum is an alternative to the method described above that performs the update in two steps. In the first step, the weights are updated with the velocity that has been gathered thus far. These updated weights are then used to compute a lookahead gradient that makes up the update in the second step. This makes it possible to make up for wrong directions faster (Sutskever et al., 2013). The updates are given by

$$\begin{aligned} w &\leftarrow w + \mu v \\ v &\leftarrow \mu v - \eta \nabla_{\mathcal{B}} C(\boldsymbol{y}, \hat{\boldsymbol{y}}) \\ w &\leftarrow w - \eta \nabla_{\mathcal{B}} C(\boldsymbol{y}, \hat{\boldsymbol{y}}) \,, \end{aligned} \tag{3.23}$$

where the gradients are computed w.r.t. the weights after the first update. Nesterov momentum has slightly better convergence properties than the classical method and works with greater values for μ (Sutskever et al., 2013).

3.5.2 Next-Level Gradient Descent

Another popular family of SGD variants use per-parameter, adaptive learning rates instead of the global and fixed variants discussed thus far. A first example of such an algorithm is AdaGrad (Duchi, Hazan and Singer, 2011), which gradually decreases the learning rate for each weight individually as the model learns. One main disadvantages of AdaGrad is that the learning rate converges to zero. Adadelta is a method that tries to improve on the pitfalls of AdaGrad (Zeiler, 2012). This is done by replacing the sum of squared gradients by an exponential moving average. This introduces an extra hyperparameter ρ, but keeps the learning rate from disappearing. Another famous algorithm in this family is Adam (Diederik P. Kingma and J. Ba, 2015), which combines exponential moving averages to estimate both the first and the second raw moments of the gradients. It is often seen as a combination of Adadelta and momentum.

All of the above algorithms use only first order derivatives to update the weights. There are a several second order methods that require the Hessian to be either computed or estimated, e.g. Newton's method, CG, BFGS, etc. Because this is often a costly computation and does not provide significant advantages over the methods above, they are not commonly used in practice. For a more thorough and complete discussion on the different optimisation algorithms, we refer to (Goodfellow, Bengio and Courville, 2016).

4 Self-Normalisation

It has long been known that it is easier for layers in a ANN to learn from whitened, i.e. decorrelated, zero mean and unit variance, features (LeCun, Bottou, Orr et al., 1998). Therefore, a common strategy to facilitate learning in ANNs, is to apply a whitening transform on the data during pre-processing. Unfortunately, the benefits of whitened input features get lost as the data passes through the learning network. Different techniques have recently been proposed to assure normalised inputs at each layer to speed up learning.

In this chapter, we will focus on a technique that is known as Self-Normalising neural networks (SNNs) (Klambauer et al., 2017). In section 4.1, we discuss the negative impact of the changes in the distributions of the data in a ANN. We also touch upon a set of normalisation techniques that alleviate this negative impact. The basic ideas of self-normalisation are introduced in section 4.2 as well as an extension of SNNs that takes into account the effect of bias units. How scaled exponential linear units (SELUs) help to create SNNs is discussed in section 4.3. Finally, our analysis of the backward pass in BP is described in section 4.4.

4.1 Internal Covariate shift

In order to accelerate learning in a ANN, it is considered good practice to normalize or whiten the data during pre-processing (LeCun, Bottou, Orr et al., 1998). As the data propagates through the different layers of the network, however, it undergoes various layer transformations. On top of that, the weights, which form the linear part of these layer transformations, are updated repeatedly. This causes each transformation to alter the data differently after each update. As a consequence, the beneficial properties of whitened data at the input of a ANN gets lost as it propagates through the network.

The actual problem is not that the data is transformed, but rather that each transform generally affects the distribution of the data. The term *internal covariate shift* has been defined as ‘change in the distribution of network activations due to the change in network parameters during training’ Ioffe and Szegedy, 2015. Here, we extend this definition to also incorporate the changes in distribution that are a consequence of applying layer transformations on the data (see figure 4.1). We will refer to these latter changes as static internal covariate shift and those due to parameter updates will be referred to as dynamic internal covariate shift.

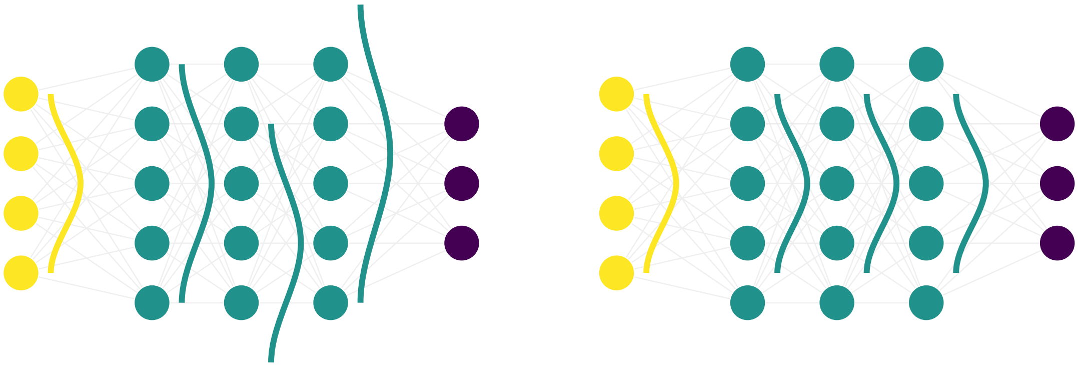

(a) with (b) without

Figure 4.1: Visualisation of (static) internal covariate shift in a ANN. The curve after each layer sketches the probability density function of each layer. Ideally, the distribution stays about the same (right), but in practice, the distribution changes (left). The dynamic internal covariate shift would denote the changes due to parameter updates.

4.1.1 Regular Normalisation

To preserve the moments of data with zero mean and unit variance input, LeCun et al. suggest to use a scaled version of the hyperbolic tangent (LeCun, Bottou, Orr et al., 1998)

$$\begin{aligned}\operatorname{stanh}(x) &= 1.7159 \cdot \tanh\left(\frac{2}{3}x\right)\\ \operatorname{stanh}'(x) &= 1.7159 \cdot \frac{2}{3}\left(1 - \tanh\left(\frac{2}{3}x\right)^2\right).\end{aligned}$$

By initialising the weights so that

$$\mathrm{E}\left[W^{(l)}\right] = 0 \qquad\qquad \mathrm{Var}\left[W^{(l)}\right] = \frac{1}{N_{l-1}},$$

the inputs to the scaled tanh have unit variance and the outputs in each layer should be normalised. It turns out, however, that the scaling merely gets the variance of its outputs closer to one than the original tanh. Instead of the promised unit variance, it can easily be verified experimentally that it is rather around 0.75.

A more recent normalisation technique that has gained plenty of attention, is *batch normalisation* (Ioffe and Szegedy, 2015). By adding a per-batch, per-feature normalising transformation

$$\mathrm{BN} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} : s_i\,; \gamma, \beta \mapsto \gamma \frac{s_i - \mathrm{E}_{\boldsymbol{x} \in \mathcal{B}}\left[s_i\right]}{\sqrt{\mathrm{E}_{\boldsymbol{x} \in \mathcal{B}}\left[\left(s_i - \mathrm{E}_{\boldsymbol{x} \in \mathcal{B}}\left[s_i\right]\right)^2\right]}} + \beta$$

in front of the activation function in each layer, the internal covariate shift can be effectively countered. In order to prevent the normalisation from damaging the representational power of the NN, parameters γ and β are introduced. γ and β explicitly allow the network to undo the normalisation, which is necessary to allow neurons to saturate. Both γ and β can be learnt with BP because the normalising transformation is entirely differentiable.

Layer normalisation is a technique that applies the basic ideas of batch normalisation in a different way to make it suitable for RNNs and also more efficient (L. J. Ba, Kiros and G. E. Hinton, 2016). Instead of computing the statistics for each neuron individually, one mean and standard deviation are computed for each layer. Another important difference with batch normalisation is that mean and variance are computed for each sample individually, rather than for the entire batch. Mathematically, the layer normalisation transform is given by

$$\mathrm{LN} : \mathbb{R}^N \times \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N : \boldsymbol{s}\,; \boldsymbol{\gamma}, \boldsymbol{\beta} \mapsto \boldsymbol{\gamma} \odot \frac{\boldsymbol{s} - \mathrm{E}_{s_i \in \boldsymbol{s}}\left[s_i\right]}{\sqrt{\mathrm{E}_{s_i \in \boldsymbol{s}}\left[\left(s_i - \mathrm{E}_{s_i \in \boldsymbol{s}}\left[s_i\right]\right)\right]}} + \boldsymbol{\beta}$$

A completely different approach is taken in *weight normalisation* (Salimans and Diederik P Kingma, 2016). By re-parametrising the weights in each neuron using

$$\boldsymbol{w} = \frac{g}{\|\boldsymbol{v}\|}\boldsymbol{v}\,,$$

the norm g and direction $\boldsymbol{v}$ of each weight vector are decoupled. By carefully initialising parameters g and the bias terms b with mini-batch statistics, the inputs to the activation function $s = \boldsymbol{w} \cdot \boldsymbol{x} + b$ can be efficiently normalised. This technique turns out to be especially effective when combined with mean-only batch normalisation (Salimans and Diederik P Kingma, 2016).

4.2 Self-Normalisation

Self-Normalising neural networks (SNNs) make up a collection of networks in which each layer intrinsically transforms the data so that its moments are preserved (Klambauer et al., 2017). Unlike the techniques from section 4.1.1, where the inputs to the activation function s_i are normalised, SNNs focus on normalising the activations a_i of each layer. On top of providing inputs in the non-saturating region of the activation function, this avoids the direction of weight updates to be biased, which should improve learning even more (LeCun, Bottou, Orr et al., 1998).

4.2.1 Contraction Mapping

In order to understand self-normalisation, we need to take a look at the moment mapping

$$M_F : [a, b] \times [c, d] \to [a, b] \times [c, d] : \left(\mu_X, \sigma_X^2\right) \mapsto \left(\mu_A, \sigma_A^2\right)\,,$$

that is induced by each layer transformation. Here, X and A are random variables representing one of the inputs x_i and one of the activations a_i in one layer and $[a, b] \subseteq \mathbb{R}$ and $[c, d] \subseteq \mathbb{R}^+$.

SNNs are capable of preserving the moments of the data by using layers that induce a *contraction mapping* on the moments (Klambauer et al., 2017). This means that for the layers in a SNN, the Jacobian of M_F,

$$\mathcal{J}_F = \begin{pmatrix} \frac{\partial \mu_A}{\partial \mu_X} & \frac{\partial \mu_A}{\partial \sigma_X^2} \\ \frac{\partial \sigma_A^2}{\partial \mu_X} & \frac{\partial \sigma_A^2}{\partial \sigma_X^2} \end{pmatrix} ,$$

does not have eigenvalues with an absolute value larger than one, i.e.

$$\left| \frac{1}{2} \left(\mathcal{J}_{F11} + \mathcal{J}_{F22} \pm \sqrt{\left(\mathcal{J}_{F11} + \mathcal{J}_{F22}\right)^2 - 4\left(\mathcal{J}_{F11}\mathcal{J}_{F22} - \mathcal{J}_{F12}\mathcal{J}_{F21}\right)} \right) \right| < 1 , \tag{4.1}$$

for any point in the domain of the mapping.

Such a contraction mapping is known to have a unique fixed point and converges to this fixed point when applied iteratively (Banach, 1922). This means that the moments of activations in the end of a SNN for which all layers induce the same contraction mapping, will converge to the fixed point of that contraction mapping. Additionally, SNNs can effectively normalise the data to have zero mean and unit variance by constructing the layers so that the fixed point of their moment mapping is $(0, 1)$.

4.2.2 Self-Normalising Layers

Assuming that inputs X, weights W and biases B are all independent random variables, the moments of the weighted inputs S in a fully connected layer (3.3) are given by

$$\begin{aligned} \mu_S &= \mathrm{E}\left[B + \sum_{i=1}^{n} WX\right] = \mu_B + n\mu_W\mu_X \\ \sigma_S^2 &= \mathrm{E}\left[\left(B + \sum_{i=1}^{n} WX\right)^2\right] - \mu_S^2 = \sigma_B^2 + n\left(\sigma_W^2\sigma_X^2 + \mu_W^2\sigma_X^2 + \sigma_W^2\mu_X^2\right) . \end{aligned} \tag{4.2}$$

If n, the number of inputs to a layer, is large, the central limit theorem applies and therefore S is approximately normally distributed.

This information allows us to compute the moments of the activations as

$$\begin{aligned} \mu_A &= \mathrm{E}\left[f(S)\right] = \int_{-\infty}^{\infty} f(s) p_{\mathcal{N}}(s\,; \mu_S, \sigma_S)\,\mathrm{d}s \\ \sigma_A^2 &= \mathrm{E}\left[f(S)^2\right] - \mu_A^2 = \int_{-\infty}^{\infty} f(s)^2 p_{\mathcal{N}}(s\,; \mu_S, \sigma_S)\,\mathrm{d}s - \mu_A^2 , \end{aligned} \tag{4.3}$$

where $p_{\mathcal{N}}(x\,; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu^2)}{2\sigma^2}\right)$.

Given expressions for the moments of the activations, a specific fixed point (μ, σ^2) can be enforced by solving the fixed point equations

$$\begin{cases} \mu_X = \mu_A & = \mu \\ \sigma_X^2 = \sigma_A^2 & = \sigma^2 \end{cases} . \tag{4.4}$$

For a fully connected layer (3.3), the unknowns in these equations are the moments of weights and biases as well as possible parameters for the activation function. If the fixed point equations can be solved for one or two of these unknowns, the fixed point of M_F will be (μ, σ^2). By setting the fixed point to $(\mu, \sigma^2) = (0, 1)$, SNNs effectively alleviate what we refer to as the static internal covariate shift.

Once the equation system (4.4) is solved in some fixed point (μ, σ^2), the unknowns that are left in the fixed point equations can be assigned any possible value. In order for M_F to be a contraction mapping, however, the stability conditions (4.1) have to be taken into account. If these are satisfied in the fixed point (μ, σ^2), M_F is a contraction mapping in the area around the fixed point for which the stability conditions hold. Otherwise, the moments might diverge away from the fixed point.

4.3 Scaled Exponential Linear Units

The SELU

$$\mathrm{SELU}(x\,; \alpha, \gamma) = \gamma\, \mathrm{ELU}(x\,; \alpha) \qquad \mathrm{SELU}'(x\,; \alpha, \gamma) = \gamma\, \mathrm{ELU}'(x\,; \alpha)\,, \tag{4.5}$$

is one possible activation function that allows to construct self-normalising layers (Klambauer et al., 2017). The parameters α and γ allow to control the changes to both mean and variance in a fully connected layer with SELU activations. By choosing appropriate values for these parameters and the initial weights, a fully connected layer induces a contraction mapping on the moments, which leads to self-normalisation.

4.3.1 Finding parameters

The moments of SELU activations in a MLP are given by (Klambauer et al., 2017)

$$\mu_A = \gamma \left(\alpha(E_1 - E_0) + \frac{\sigma_S}{\sqrt{2\pi}} e^{-\frac{\mu_S^2}{2\sigma_S^2}} + \mu_S(1 - E_0) \right) \tag{4.6}$$

$$\sigma_A^2 = \gamma^2 \left(\alpha^2(E_2 - 2E_1 + E_0) + \frac{\sigma_S \mu_S}{\sqrt{2\pi}} e^{-\frac{\mu_S^2}{2\sigma_S^2}} + (\mu_S^2 + \sigma_S^2)(1 - E_0) \right) - \mu_A^2\,, \tag{4.7}$$

where, for notational brevity, we use

$$E_k = \mathrm{E}_{X<0}\left[e^{kX}\right] = \int_{-\infty}^{0} e^{kx} \frac{1}{\sqrt{2\pi\sigma_S^2}} e^{-\frac{(x-\mu_S)^2}{2\sigma_S^2}}\, \mathrm{d}x = \frac{1}{2} e^{k(\mu_S + \frac{k}{2}\sigma_S^2)}\, \mathrm{erfc}\left(\frac{\mu_S + k\sigma_S}{\sqrt{2\sigma_S^2}} \right) .$$

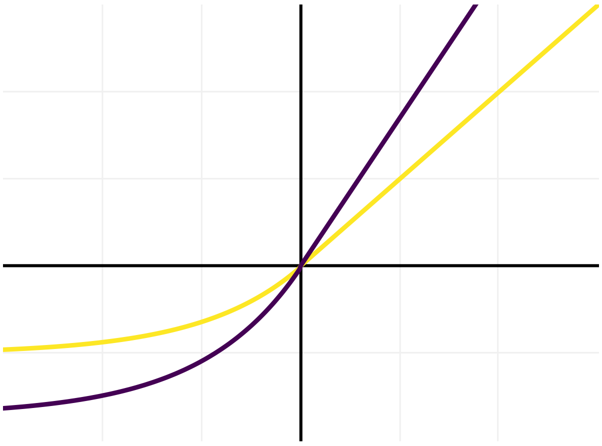


Figure 4.2: Plot of the scaled exponential linear unit (SELU) activation function. The brighter line is the plot of the exponential linear unit (ELU), i.e. without scaling.

For a fixed point at $(\mu, \sigma^2) = (0, 1)$, the fixed point equations (4.4) lead to the following expressions for the SELU hyperparameters

$$\alpha = -\left(\sqrt{\frac{\omega}{2\pi}} e^{-\frac{\mu_B^2}{2\omega}} + \mu_B(1 - E_0')\right)(E_1' - E_0')^{-1} \tag{4.8}$$

$$\gamma^2 = \left(\alpha^2(E_2' - 2E_1' + E_0') + \mu_B \sqrt{\frac{\omega}{2\pi}} e^{-\frac{\mu_B^2}{2\omega}} + (\mu_B^2 + \omega)(1 - E_0')\right)^{-1}, \tag{4.9}$$

with $E_k' = \frac{1}{2} e^{k(\mu_B + \frac{k}{2}\omega)} \operatorname{erfc}\left(\frac{\mu_B + k\sqrt{\omega}}{\sqrt{2\omega}}\right)$ and $\omega = \sigma_B^2 + n(\sigma_W^2 + \mu_W^2)$.

4.3.2 The Role of Initialisation

The variables in the expressions for α (4.8) and γ (4.9) that lead to a fixed point at $(\mu, \sigma^2) = (0, 1)$, are the moments of the parameters in the layer. If the biases are initialised to zeros, i.e. $(\mu_B, \sigma_B^2) = (0, 0)$, and the initial weights satisfy $(\mu_W, \sigma_W^2) = (0, \frac{1}{n})$, these variables are known and the SELU parameters are given by

$$\begin{aligned} \alpha &= -\sqrt{\frac{2}{\pi}}\left(\sqrt{e}\operatorname{erfc}\left(\frac{1}{\sqrt{2}}\right) - 1\right)^{-1} && \approx 1.67326 \\ \gamma &= \pm\sqrt{2}\left(\alpha^2\left(e^2\operatorname{erfc}\left(\sqrt{2}\right) - 2\sqrt{e}\operatorname{erfc}\left(\frac{1}{\sqrt{2}}\right) + 1\right) + 1\right)^{-\frac{1}{2}} && \approx \pm 1.05070\,. \end{aligned} \tag{4.10}$$

These are the same values as presented in (Klambauer et al., 2017), where the biases are not explicitly taken into account.

In (Klambauer et al., 2017) an extensive proof is provided that the moment mapping of a fully connected layer with SELU activations and the aforementioned parameters, leads to a moment contraction mapping

$$M_F : [-0.1, 0.1] \times [0.8, 1.5] \to [-0.1, 0.1] \times [0.8, 1.5] : (\mu_X, \sigma_X^2) \mapsto (\mu_A, \sigma_A^2)\,,$$

with fixed point at $(\mu_X, \sigma_X^2) = (0, 1)$. The domain of this function makes up the region for which a SELU is guaranteed to pull activations towards its fixed point. An important condition for this result to hold is that the mean of the weighted inputs is close to zero and the variance stays close to one, i.e. $\mu_S \in [-0.01, 0.01]$ and $\sigma_S^2 \in [0.76, 1.65]$ (Klambauer et al., 2017). In case the biases are ignored, this is satisfied if $\mu_W \in [-0.1, 0.1]$ and $\sigma_W^2 \in [\frac{0.95}{n}, \frac{1.1}{n}]$ (Klambauer et al., 2017). Otherwise, there is a trade-off between the moments of biases and weights.

Note that it is also possible to find parameters for non-contraction mappings with SELUs. If the biases would have been initialised so that $\sigma_B^2 = 1$ or $\sigma_W^2 = \frac{2}{n}$ holds for the weights, the SELU parameters would be

$$\begin{aligned}
\alpha &= -\frac{2}{\sqrt{\pi}} \left(e \operatorname{erfc}\left(\frac{1}{\sqrt{2}}\right) - 1 \right)^{-1} && \approx 8.20875 \\
\gamma &= \pm\sqrt{2} \left(\alpha^2 \left(e^4 \operatorname{erfc}\left(\sqrt{2}\right) - 2e \operatorname{erfc}\left(\frac{1}{\sqrt{2}}\right) + 1 \right) + 2 \right)^{-\frac{1}{2}} && \approx \pm 0.12881 \,.
\end{aligned}$$

However, this is not a contraction mapping, because the stability conditions (4.1) are not satisfied. This can also be derived from the fact that σ_S^2 would be $\sigma_B^2 + \sigma_X^2 \sigma_W^2 = 2 \notin [0.76, 1.65]$ in both cases.

4.4 Moment Dynamics

After initialisation, the moments of weights and biases are known and it can be assured that layers induce a moment mapping. As the network is learning, however, the weights and biases are updated and their distributions change. If the moments of these parameters stay within a certain range, e.g. as discussed in section 4.3.2, there is generally no problem. However, if they get out of that range, the fixed point become unstable, which might cause the moments to diverge.

In this section, we analyse the moments in the BP algorithm to get an idea of how the moments of the weights change during learning. We apply our analysis to SNNs with a SELU activation function and parameters as discussed in section 4.3. Finally, also the influence of momentum to the changes is discussed.

4.4.1 Moments of Deltas

In order to get an idea how the learning updates influence the moments of weights and biases, we analyse the deltas (3.16) and their distributions in the course of BP. Concretely, we will consider the backward moment mapping

$$M_B : [a, b] \times [c, d] \to [a, b] \times [c, d] : \left(\mu_D, \sigma_D^2\right) \mapsto \left(\mu_{D_-}, \sigma_{D_-}^2\right) ,$$

which maps the moments of deltas D in layer l onto the mean and variance of deltas D_- in layer $l - 1$.

For a fully connected layer, the deltas are propagated according to equation (3.17). Under the assumption that deltas, weights and the weighted inputs of the previous layer are independent, the moments are mapped onto

$$\begin{aligned}\mu_{D_-} &= \mathrm{E}\left[f'(S_-)\cdot\sum_{i=1}^{N} WD\right]\\ &= N\mu_W\mu_D\int_{-\infty}^{\infty} f'(s)p_{\mathcal{N}}(s\,;\mu_{S_-},\sigma_{S_-})\,\mathrm{d}s\\ \sigma^2_{D_-} &= \mathrm{E}\left[f'(S_-)^2\cdot\left(\sum_{i=1}^{N} WD\right)^2\right] - \mu^2_{D_-}\\ &= N\left(\sigma^2_W\sigma^2_D + \mu^2_W\sigma^2_D + \sigma^2_W\mu^2_D + N\mu^2_W\mu^2_D\right)\int_{-\infty}^{\infty} f'(s)^2 p_{\mathcal{N}}(s\,;\mu_{S_-},\sigma_{S_-})\,\mathrm{d}s - \mu^2_{D_-}\,.\end{aligned} \tag{4.11}$$

Here, S_- represents the summed inputs of layer $l-1$ and N is the number of units in layer l. From these expressions, it becomes immediately clear that the errors are expected to have zero mean if the weights are centred.

When using simple gradient descent (3.20), the moments of the weights after one update are

$$\begin{aligned}{\mu_W}' &= \mu_W - \eta\mu_D\mu_X\\ {\sigma^2_W}' &= \sigma^2_W + \eta^2(\sigma^2_D\sigma^2_X + \mu^2_D\sigma^2_X + \sigma^2_D\mu^2_X)\,.\end{aligned} \tag{4.12}$$

For the biases, we find

$$\begin{aligned}{\mu_B}' &= \mu_B - \eta\mu_D\\ {\sigma_B}' &= \sigma^2_B + \eta^2\sigma^2_D\,.\end{aligned} \tag{4.13}$$

Note that in (4.12), the dependency between the weights and the deltas are ignored for simplicity.

The most important observations at this point are that the weights stay approximately centred if the inputs have zero mean. Also if the errors have zero mean, the biases and weights will not undergo a shift. The variance, on the other hand, tends to increase with every update for both weights and biases. This seems to form the most important threat to a moment mapping with a stable fixed point. Also note that the variance increases faster in a network with bias units than for a ANN without biases.

4.4.2 Backprop with SELUs

In order to get an idea of the magnitude of the variance increase due to updates, we can take a look at what happens in a fully connected layer with SELU (4.5) for an activation function. In this case, the moments of the deltas (4.11) are

$$\begin{aligned}\mu_{D_-} &= N\mu_W\mu_D\gamma(\alpha E_{1_-} - E_{0_-} + 1)\\ \sigma^2_{D_-} &= N\left(\sigma^2_W\sigma^2_D + \mu^2_W\sigma^2_D + \sigma^2_W\mu^2_D + N\mu^2_W\mu^2_D\right)\gamma^2(\alpha^2 E_{2_-} - E_{0_-} + 1) - \mu^2_{D_-}\,.\end{aligned}$$

Here, $E_{k_-} = \frac{1}{2} e^{k(\mu_{S_-} + \frac{k}{2}\sigma_{S_-}^2)} \operatorname{erfc}\left(\frac{\mu_{S_-} + k\sigma_S}{\sqrt{2\sigma_{S_-}^2}}\right)$, cf. section 4.3.1.

Using the settings that lead to a contraction mapping with fixed point in $(\mu_X, \sigma_X^2) = (0, 1)$ from section 4.3.2, i.e. $(\mu_W, \sigma_W^2) = (0, \frac{1}{n})$, $(\mu_B, \sigma_B) = (0, 0)$ with parameters $\alpha \approx 1.67326$ and $\gamma \approx 1.05070$, we find

$$\begin{aligned} \mu_{D_-} &= 0 \\ \sigma_{D_-}^2 &\approx 1.07157 \frac{N}{n} \cdot (\sigma_{D_+}^2 + \mu_{D_+}^2)\,. \end{aligned} \tag{4.14}$$

With the recurrence for computing the deltas (3.17), the variance can also be reformulated to

$$\sigma_{D_{L-l}}^2 = 1.07157^l \cdot (\sigma_{D_L}^2 + \mu_{D_L}^2) \prod_{i=L-l}^{L-1} \frac{N_{i+1}}{N_i}\,,$$

where μ_{D_L} and σ_{D_L} are the moments of the error in the last layer.

Both formulations indicate that each layer scales the variance by a factor $1.07157\frac{N}{n}$ with n, the inputs dimension and N, the number of units in that layer. On top of that, this scaling is exponential in the number of layers L in the network. The variance of the back-propagated error will thus generally increase, unless the number of units per layer gradually decreases towards the end of the network so that $N \leq 1.07157n$.

4.4.3 The Role of Momentum

There is more than just gradient descent to do the optimisation in BP. One easy extension to standard gradient descent is to use momentum (3.22). This introduces an extra random variable V for the velocity of the gradient that changes with every update as well. Assuming that V is independent from the inputs and the deltas, the changes to the moments of the weights are given by

$$\begin{aligned} {\mu_W}' &= \mu_W + {\mu_V}' & {\mu_V}' &= \mu\mu_V - \eta\mu_D\mu_X \\ {\sigma_W^2}' &= \sigma_W^2 + {\sigma_V^2}' & {\sigma_V^2}' &= \mu^2\sigma_V^2 + \eta^2(\sigma_D^2\sigma_X^2 + \sigma_D^2\mu_X^2 + \mu_D^2\sigma_X^2)\,. \end{aligned}$$

Mean and variance of the biases are altered according to

$$\begin{aligned} {\mu_B}' &= \mu_B + {\mu_V}' & {\mu_V}' &= \mu\mu_V - \eta\mu_D \\ {\sigma_B^2}' &= \sigma_B^2 + {\sigma_V^2}' & {\sigma_V^2}' &= \mu^2\sigma_V^2 + \eta^2\sigma_D^2\,. \end{aligned}$$

Assuming that the initial velocity is zero, i.e. $(\mu_V, \sigma_V^2) = (0, 0)$, the first update with momentum will lead to the same changes as if it would be trained with plain gradient descent. However, the changes due to each subsequent update will be amplified by a factor μ of previous changes. Because common choices for μ are values close to one, the use of momentum can significantly speed up the variance increase and thus cause unstable fixed points sooner.

Similar observations can be made in case Nesterov momentum (3.23) has been used. In this case, the updates lead to the following moment changes for the weights

$$\begin{aligned} {\mu_W}' &= \mu_W + \mu\mu_V - \eta{\mu_D}'\mu_X & {\mu_V}' &= \mu\mu_V - \eta{\mu_D}'\mu_X \\ {\sigma_W^2}' &= \sigma_W^2 + \mu^2\sigma_V^2 + \eta^2\sigma_{D'X}^2 & {\sigma_V^2}' &= \mu^2\sigma_V^2 + \eta^2\sigma_{D'X}^2\,, \end{aligned}$$

where $\sigma_{D'X}^2 = ({\sigma_D^2}'\sigma_X^2 + {\sigma_D^2}'\mu_X^2 + {\mu_D^2}'\sigma_X^2)$. For the biases, it can be found that

$$\begin{aligned} {\mu_B}' &= \mu_B + \mu\mu_V - \eta{\mu_D}' & {\mu_V}' &= \mu\mu_V - \eta{\mu_D}' \\ {\sigma_B^2}' &= \sigma_B^2 + \mu^2\sigma_V^2 + \eta^2{\sigma_D^2}' & {\sigma_V^2}' &= \mu^2\sigma_V^2 + \eta^2{\sigma_D^2}'\,. \end{aligned}$$

In the above expressions, ${\mu_D}'$, ${\sigma_D^2}'$ are the moments of the deltas D' computed with the parameters after the first update step, i.e. $w + \mu v$ and $b + \mu v$.

5 Empirical Results

This chapter provides the results for a set of experiments with the main purpose to empirically justify our analysis. On top of that, we try to gain a better understanding of the distributions of data and how these change in SNNs with SELU activations. The basic settings of the experiments are sketched in section 5.1. In section 5.2, we take a look at how pre-processing affects the performance of SNNs. The effect of weights and biases is discussed in section 5.3 with the empirical justification of our analysis of the moment dynamics. Section 5.4 provides results that lead to some interesting insights concerning the influence of the architecture on the backward pass in BP. Finally, we compare the outcomes of using different optimisers in section 5.5.

5.1 Experiment Settings

For our experiments, we chose to test our theory on image classification problems because it is a well-known task with a wide variety of easily accessible datasets. The models that we use to solve these tasks, are simple SNNs with SELU activation functions. For the implementation of the different networks, we chose to use the Keras framework (Chollet, 2015) in python. Training was done on a NVIDIA Tesla K40C GPU using the tensorflow (Abadi et al., 2015) back-end of Keras with 32-bit precision.

In this section, we introduce the different datasets that we used for our experiments. We also introduce the baseline model for each dataset. The settings of these networks are the same for the models that are compared against this baseline. Finally, we also provide some details on the visualisations that are used to visualise the changes of moments in the network.

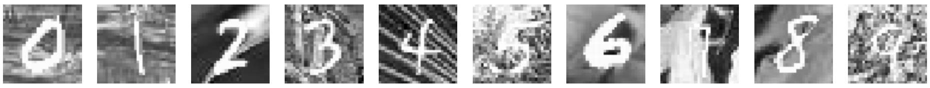
5.1.1 Datasets

We used two families of datasets for our experiments: grey-scale images of handwritten digits (MNIST) and pictures of objects with RGB colours (CIFAR). The raw pixel values for the images in all datasets are unsigned integers in the range $[0, 255]$. For feeding the data to the networks, the 2D images were rearranged to create a 1D feature vector of concatenated pixel rows.

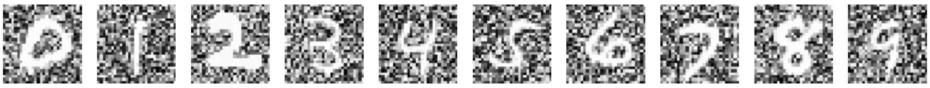
MNIST

The first dataset that we used is MNIST (LeCun, Cortes and Burges, n.d.). It consists of 60 000 training and 10 000 test samples, where each sample is a 28×28 image representing

(a) MNIST



(b) MNIST-BG-IMG



(c) MNIST-BG-RAND

Figure 5.1: Examples from the training sets of different datasets in the MNIST family: standard MNIST, MNIST with background images and MNIST with a random background.

some digit between zero and nine. In our experiments, we used the entire training set for learning and ignored the test set. One problem with the MNIST dataset is that it has some pixels with zero variance over the entire dataset. This makes it impossible to rescale the data so that the variance of all pixels is one.

In addition to the traditional MNIST dataset, we also ran our experiments on variations of MNIST with backgrounds (Larochelle, Dimitru Erhan and Aaron, n.d.). These alternatives to MNIST contain the same images of digits, but the plain black background is replaced with images in one dataset (MNIST-BG-IMG) and white noise in the other (MNIST-BG-RAND). This assures that each pixel has some variance and therefore it is easier to satisfy the assumptions on which SNNs are built. Another remarkable difference is that there are 12 000 samples for training and 50 000 for testing in these variations. For the experiments, we chose to use the 50 000 test samples for learning and ignored the training set for both datasets.

CIFAR

Finally, we also used the CIFAR datasets (Krizhevsky, Nair and G. Hinton, 2009). Each CIFAR dataset consists of 32×32 images with three colour channels: red, green and blue. CIFAR-10 contains images representing ten classes and CIFAR-100 has hundred different labels, arranged in twenty super-classes. For both datasets, the training set holds 50 000 samples and the test set counts 10 000 images. Again, we used the training sets for fitting the model and ignored the test sets.

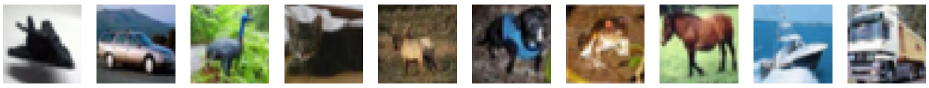


Figure 5.2: Examples from the training set of the CIFAR-10 dataset. The classes are *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship* and *truck*.

Figure 5.3: Examples from the training set of the CIFAR-100 dataset for ten of the hundred classes. These examples represent the classes *apple*, *aquarium fish*, *baby*, *bear*, *beaver*, *bed*, *bee*, *beetle*, *bicycle* and *bottle*.

5.1.2 Base Model

The basic model that has been used as a baseline throughout the different experiments, is a simple MLP with $L = 6$ layers, i.e. five hidden layers. We chose to use a rectangular architecture, where the number of neurons in each hidden layer is the same. The exact numbers depend on the family of images from which the model has to learn.

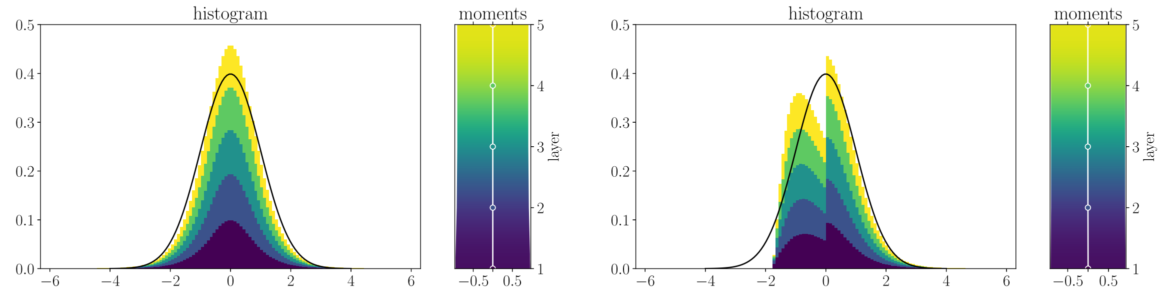
The networks that learn from a MNIST-like dataset have 784 input units, 10 outputs and 500 neurons in each hidden layer. For the CIFAR datasets, we used models with 3072 inputs and hidden layers of 1000 neurons. The number of outputs depends on the class count of the dataset and is thus either 10 or 100 for CIFAR-10 and CIFAR-100 respectively.

The activation function for the networks is a SELU with the parameters from (4.10) and the weights are initialised by drawing samples from a uniform distribution so that $(\mu_W, \sigma_W^2) = (0, \frac{1}{n})$. The baseline network does not have biases and by whitening the data during pre-processing, the layers in the network push moments towards the fixed point $(\mu, \sigma^2) = (0, 1)$.

The only loss that has been used in all of the experiments is the cross-entropy loss (2.5). Therefore, the labels of the data are one-hot encoded and the networks have as many outputs as there are classes. To exclude influence from other updating dynamics, we used plain SGD (3.21) as the default optimiser. The data has been processed in batches of 64 samples with a default learning rate of $\eta = 0.001$.

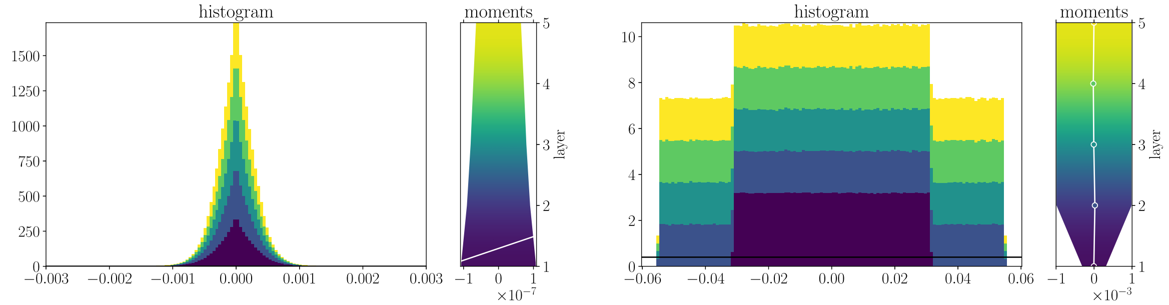
Rationale

The general idea behind these architectural choices was to have networks that are deep and wide enough without requiring too much computational resources. Especially memory was a valuable resource during the experiments because for each network, several internal values were monitored during learning. Concretely, we have tracked the distributions of weighted inputs s and activations a in the forward pass and deltas δ in the backward pass. Also the values of the weights in each layer have been monitored



(a) s

(b) a



(c) δ

(d) w

Figure 5.4: Distributions of the weighted inputs s, activations a, deltas δ and weights w in the CIFAR-10 baseline model before the first update.

during learning. This led to the visualisations in figure 5.4.

Some of the theory from chapter 4 has been taken into account when making these design choices as well. In order for the central limit theorem to hold (see section 4.2.2), it is important that the number of neurons in each layer are large enough. The rectangular architecture has been chosen because of the amount by which the variance of the deltas increases in each layer. By enforcing that $N = n$, the influence of the factor $\frac{N}{n}$ in (4.14), is cancelled out.

5.1.3 Visualisations

In order to limit the amount of concrete numbers for interpretation, we chose to visualise the distributions of the different entities that have been tracked during these experiments. In order to make these visualisations comprehensible, we shortly introduce the semantics of our figures. We use the distributions for different entities in the baseline models as an example.

In figure 5.4, we find the visualisations for the activations a, the weighted inputs s, the deltas δ and the weights w in the baseline model. Each colour in these visualisations represents one layer in the network. Brighter colours correspond to layers closer to the output and a layer that is situated in the beginning of the network has a darker colour.

The left side panel in each visualisation represents the probability density function of the entity as a histogram. These are constructed by stacking the density function of the entity from all layers. The contribution of each layer to the histogram is indicated by the colour coding. The black line that is drawn over these histograms represents the probability density function of a standard Gaussian distribution, i.e. with zero mean and unit variance, and serves simply as a reference.

The panel on the right visualises the moments of the entity in each layer. The white circles that are connected by a white line, represent the mean and the width of the coloured bar is an indication for the amount of variance. The coloured bar is scaled so that it fits within the available region. This has the advantage that the changes in variance throughout different layers are clearly visible, but makes it hard to compare the moments from different figures. This also causes the means that are greater than the variance (in absolute value) to be outside of the plot. This phenomena can be seen in figure 5.4c.

The plots with loss curves only show training losses because our analysis concerns learning behaviour rather than generalisation. Each curve in the plot is the average of three individual runs, where one run embodies the building, initialising and training of the network. The differences between the different runs are indicated by a semi-transparent region around the curve. The minima and maxima of the training loss in each epoch were used to construct this region.

5.2 Pre-processing

The idea behind normalisation is to bring moments of data towards a certain point that defines normalisation. For SNNs, this is only guaranteed to work if the input data is already normalised. In section 4.3.2, we provided the particular parameters to push the moments towards the fixed point $(\mu, \sigma^2) = (0, 1)$, which has also been used to configure our baseline model. In order to get an idea of what happens in a SNN when the moments of the input data do not lie in the domain of the contraction mapping, we tried out different pre-processing strategies in this experiment.

The optimal pre-processing would be a whitening transform, which assures zero mean, unit variance and decorrelated features. Also normalisation, i.e. centring and scaling, provides moments within the domain of the contraction mapping, but in this case, the features are not decorrelated. A more negligent way to pre-process the data is to ignore the variance and only centre the data. Finally, we also provided the network with the raw data, which has pixel values in the range $[0, 255]$ and a more standardised form, where pixels are scaled to be in the range $[0, 1]$. How these various pre-processing strategies affect the moments of the datasets that we consider, is summarised in table 5.1.

In figure 5.5, it can be seen that for normalisation and whitening, the network preserves the moments $(\mu, \sigma^2) = (0, 1)$ of the data as it is propagated through the different layers. The variance for both the centred and standardised, i.e. $\in [0, 1]$, data is much smaller than the ideal $\sigma^2 = 1$ and thus is not in the domain of the contraction mapping. Nonetheless, the variance seems to increase towards one as the data is propagated through the network in both cases. Also for the raw data, which has very high variance,

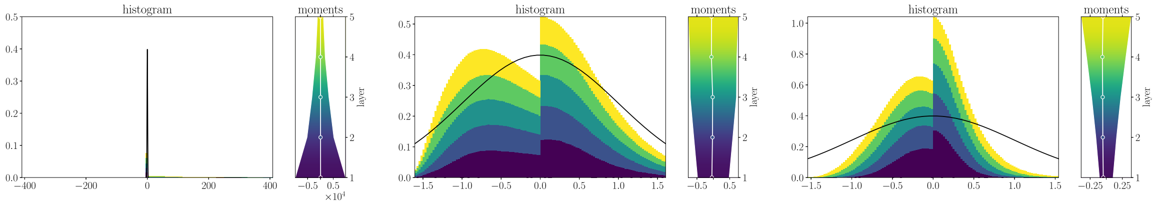
pre-processing	raw	standardised	centred	normalised	whitened
MNIST	33	0.131	-2.18e-6	-5.34e-6	-1.48e-6
MNIST-BG-IMG	139	0.545	-1.69e-5	-6.24e-5	-5.89e-6
MNIST-BG-RAND	142	0.556	-9.55e-6	-3.18e-5	-1.20e-5
CIFAR-10	121	0.473	-3.07e-7	-6.18e-7	2.42e-8
CIFAR-100	122	0.478	-1.11e-7	-3.55e-7	-4.31e-8

(a) Mean

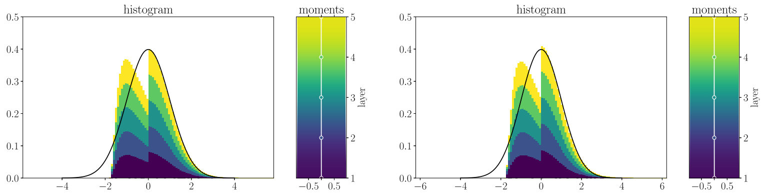
pre-processing	raw	standardised	centred	normalised	whitened
MNIST	6173	0.10	0.07	0.91	0.88
MNIST-BG-IMG	4878	0.08	0.07	1.00	1.00
MNIST-BG-RAND	5992	0.09	0.08	1.00	1.00
CIFAR-10	4115	0.06	0.06	1.00	0.97
CIFAR-100	4677	0.07	0.07	1.00	0.98

(b) Variance

Table 5.1: Moments of the different datasets with the various sorts of pre-processing.

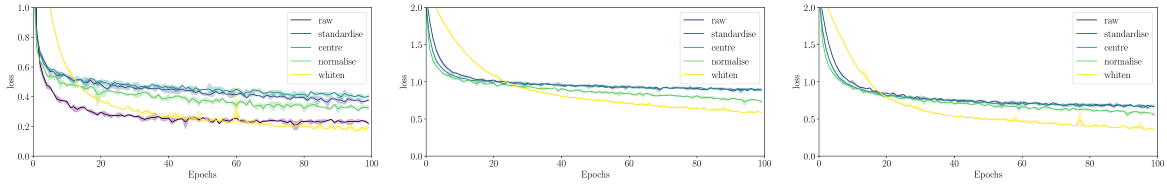


(a) raw (b) standardised (c) centred

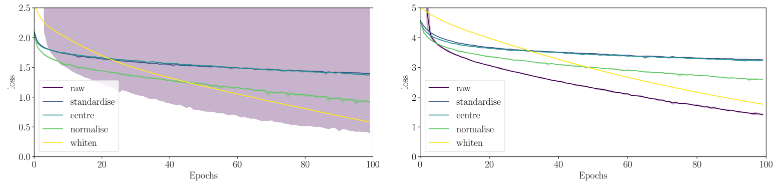


(d) normalised (e) whitened

Figure 5.5: Distributions of activations a in the MNIST-BG-IMG baseline model for different pre-processing strategies. Note how the variance in- or decreases if it is respectively too small or too large.



(a) MNIST (b) MNIST-BG-IMG (c) MNIST-BG-RAND



(d) CIFAR-10 (e) CIFAR-100

Figure 5.6: Training loss for models trained on differently pre-processed data. The pre-processing techniques that have been used are (from dark to bright): *raw*, *standardise*, *centre*, *normalise*, *whiten*. Note that for MNIST background datasets, the network did not start learning from the raw data and for CIFAR-10 it learnt only once from three runs.

the moments seem to move towards zero mean and unit variance.

These observations make clear that normalised and whitened data do have a theoretical advantage. When looking at the learning behaviour for our different datasets in figure 5.6, this theoretical advantage turns out to be a practical one. Also the decorrelation from the whitening transform does help learning compared to simple normalisation. Centring the features seems to provide little to no advantage over the standardised data.

SNNs do seem to learn quite well from raw data, although the raw data has moments that are significantly different from zero and one. Although it sometimes learns nothing at all, e.g. for MNIST with backgrounds and two out of three runs for CIFAR-10, it almost seems to lead to learn faster than any other method when it does. This is likely due to the high input values, which causes the updates to be greater and thus leads to the faster learning.

5.3 Weights

One of the main contributions of this thesis is the analysis of the moment dynamics in SNNs (see section 4.4). The most important conclusion from that analysis is that the variance of the weights will steadily increase with every update (4.12). In this experiment, we try to empirically verify that the moments of the weights do increase and by how much. Additionally, we used different distributions to draw initial weights from to verify that our analysis does indeed only depend on the moments of the weights.

5.3.1 Initialisation

In section 4.2, we described what makes a network self-normalising. To do this, we only needed assumptions on the first two moments of the data. Also the distribution of the weights has been summarised by mean and variance. To show that the exact distribution from which the initial weights are drawn, does not really matter, we initialised our baseline models in four different ways.

The most popular distributions for random number generation are probably the uniform and the normal distribution. Apart from these standard distributions, we also initialised the weights with values drawn from a truncated normal. This is practically the same as sampling from a regular normal distribution where each value outside a certain range has been re-sampled. This forces the weights to be in a well-defined range and prevents outliers from leading to saturation. Lastly, we also initialised the weights by using random orthogonal matrices. This should lead to better starting conditions, similar to what can be achieved by unsupervised pre-training (Saxe, McClelland and Ganguli, 2014).

For the experiment, we initialised all weights so that $(\mu_W, \sigma_W^2) = (0, \frac{1}{n})$, as discussed in section 4.3.2. A visualisation of the weight distributions in the resulting networks before the first update, can be found on the left side in figure 5.7. Note that the input layer has more units than the hidden layers in the baseline model and therefore the variance is smaller in the first layer of the network. Another noteworthy observation is that the variance of the truncated normal is not quite $\frac{1}{n}$. This is due to a faulty implementation in Keras and/or tensorflow.

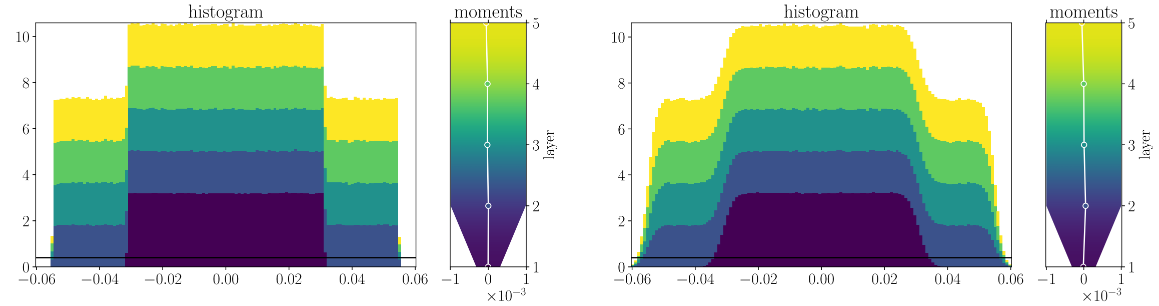
From the training loss curves in figure 5.9, there seem to be no significant differences between the various distributions. Only the truncated normal distribution seems to perform worse, especially for the CIFAR datasets, but this can be explained by the implementation issue discussed earlier. This merely illustrates the effect of having weights outside of the range to get a stable fixed point with SELUs. Orthogonalisation does seem to exhibit slightly different behaviour, but does not consistently perform better or worse than the models that are initialised with different distributions.

5.3.2 Dynamics

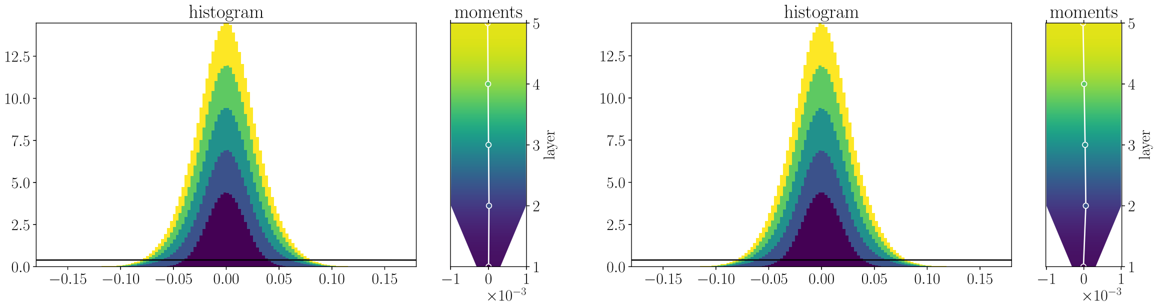
The conclusion that can be drawn from section 4.4 is that the mean of the weights stays roughly the same for centred inputs, but that their variance will increase steadily. The amount by which it increases in every update is defined mainly by the moments of the deltas (4.11). To get an idea of the changes, we compare the distributions of the weights at initialisation time to those after hundred epochs of learning (figure 5.7).

When looking at the learning curves in figure 5.9, it is clear that the parameters in the network must have changed. These changes do not seem to be reflected in the distributions of the weights before and after training (figure 5.7), however. Only for the uniform weights and the truncated normal, there is a visible difference between the distributions. The moments even look exactly the same.

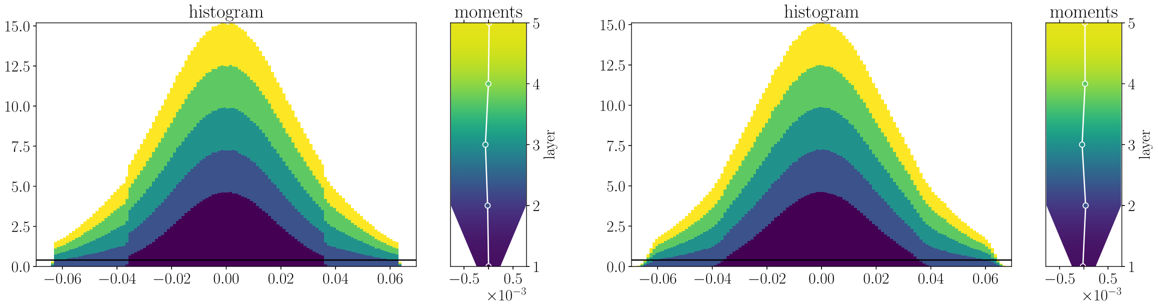
Although it seems as if nothing has changed, the variance has most certainly increased.



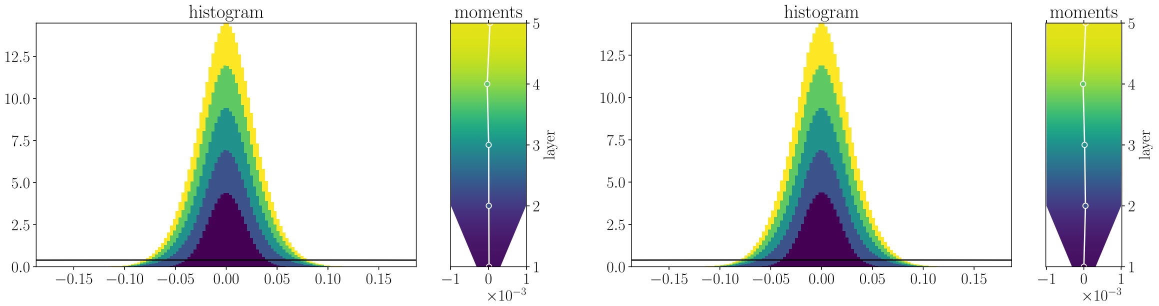
(a) uniform



(b) normal



(c) truncated



(d) orthogonal

Figure 5.7: Distributions of weights w in the CIFAR-100 baseline model for different weight initialisation strategies. The left-hand side shows the distributions before learning and the right-hand side represents the weights after 100 epochs.

layer	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
μ_W	0.269e-5	1.326e-5	0.119e-5	-1.126e-5	-2.135e-5
$\mu_W{}'$	-0.992e-5	5.370e-5	3.512e-5	0.762e-5	-2.125e-5
σ_W^2	0.326e-3	1.001e-3	0.997e-3	1.003e-3	1.000e-3
$\sigma_W^2{}'$	0.329e-3	1.012e-3	1.009e-3	1.017e-3	1.016e-3

Table 5.2: Values for moments of the normally distributed weights w depicted in figure 5.7, before (μ_W, σ_W^2) and after $(\mu_W{}', \sigma_W^2{}')$ learning for hundred epochs.

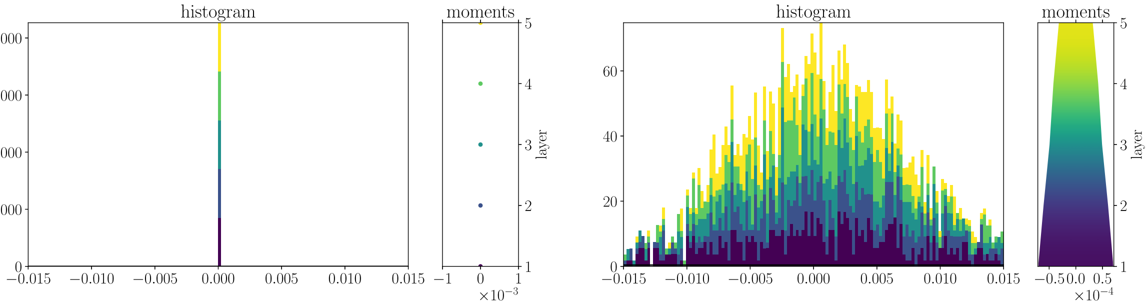
To see this, we need to take a look at the exact values of the moments before and after learning. Let us take the normally distributed weights, which seem to have changed the least, as an example. Table 5.2 lists the moments before and after training for this example. Note that we trained on CIFAR-100, which has 50 000 training samples, for 100 epochs in batches of 64. This means that the network has been trained in 78 200 updates.

The first thing that can be noted is that the mean of the weights does change, but the changes do not seem to be consistent and could therefore be explained by the fact that the deltas and/or means do not have exactly zero mean. The more interesting part is that the variance does seem to increase consistently, as we have shown in theory. Because the variance of the deltas happens to be very small, i.e. $\approx$ 1e-7, the variance increase in every update is very limited. This makes that a MLP with SELUs keeps its self-normalising properties even after a large number of updates.

5.3.3 Biases

We also described the effect of adding bias units to the original SNNs, from Klambauer et al., 2017. In section 4.3.2, we found that the standard SELU parameters (4.10) can be used if the biases are initialised to be all zeros. Just as for the weights, the variance of biases steadily increases as the network is updated. Because the moments of the weighted inputs (4.2) have to stay in the range $[0.76, 1.65]$, this additional variance increase could cause the network to get out of its self-normalising regime.

To see how this has an impact on the network, we extended the baseline model with bias units. If we look at the distributions of the biases in figure 5.8, we see that the variance does increase significantly. The updates are small enough again, however, not to hurt the self-normalising properties of the network. The learning curves for the networks with bias are practically exact to those in figure 5.9.



(a) before learning

(b) after learning

Figure 5.8: Distributions of the biases b in the MNIST baseline network with bias units before and after learning.

5.4 Architecture

In section 4.4.2, we found an expression for the factor by which the variance of the deltas is scaled as it is propagated backwards through some layer (4.14):

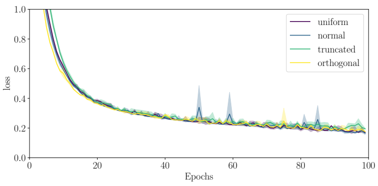
$$1.07\frac{N}{n} \cdot (\sigma_{D_+}^2 + \mu_{D_+}^2) \, .$$

This indicates that the architecture of the network, more specifically the ratio of widths, i.e. the number of neurons, $\frac{N}{n}$ in neighbouring layers, has a certain impact on the back propagation of the deltas. In order to get an idea of how this factor influences learning, we have trained several networks with a different width ratio.

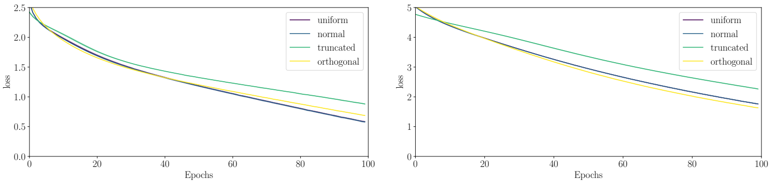
5.4.1 Set-up

For our baseline model, we explicitly chose to have a rectangular network, i.e. $n = N$, to cancel out the effect of the architecture on the propagation of the deltas (see section 5.1.2). In this case, the variance of the deltas increases as it is propagated through the network. To get a comparison with a model for which the variance of the deltas does not increase, we also considered a conic network with a decreasing number of neurons in each layer to get a ratio $\frac{N}{n} = \frac{1}{1.07} = 0.93$. We also trained the opposite network, with an increasing number of neurons so that $\frac{N}{n} = 1.07$ to get an idea of what happens when the factor is amplified. The actual number of neurons that has been used for each factor can be found in table 5.3.

The models discussed thus far, have steady or increasing variance as deltas propagate back through the network. Intuitively, this should help lower layers to learn faster. Because abstract features that are learnt by layers later in the network build upon the low-level features in the lower layers, this should make learning easier. Despite the intuitive disadvantages, we also considered networks for which the variance would decrease. Concretely, we tested two decreasingly conic networks with factors $\frac{N}{n} = \frac{1}{1.5} = 0.67$ and $\frac{N}{n} = \frac{1}{2} = 0.5$ on the MNIST family. For the CIFAR datasets, we also used a network with $\frac{N}{n} = \frac{1}{2} = 0.5$ and the opposite cone where $\frac{N}{n} = 2$.

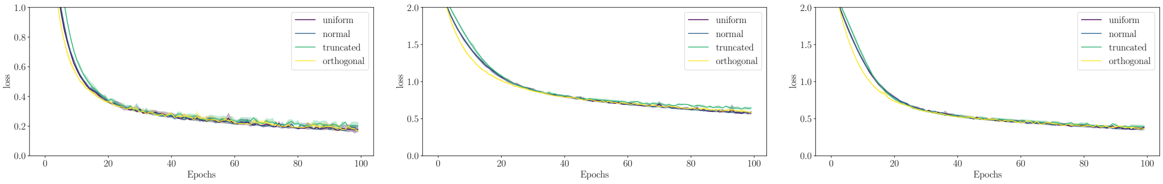


(a) MNIST (b) MNIST-BG-IMG (c) MNIST-BG-RAND

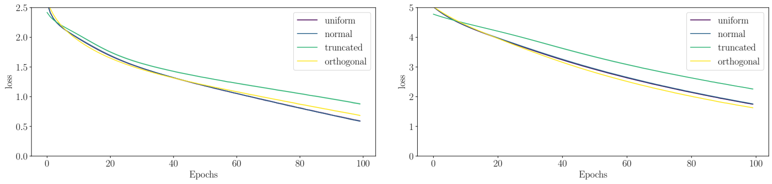


(d) CIFAR-10 (e) CIFAR-100

Figure 5.9: Training loss for networks with initial weights drawn from different distributions. The considered distributions are (from dark to bright): *uniform*, *normal*, *truncated normal* and *orthogonal*.



(a) MNIST (b) MNIST-BG-IMG (c) MNIST-BG-RAND



(d) CIFAR-10 (e) CIFAR-100

Figure 5.10: Training loss for networks with biases. This is the same experiment as in figure 5.9 with bias units, which do not seem to make a difference.

factor	N_1	N_2	N_3	N_4	N_5
1.00	500	500	500	500	500
0.93	572	535	500	467	436
1.07	436	467	500	535	572
0.67	1125	750	500	333	222
0.50	2000	1000	500	250	125

(a) MNIST

factor	N_1	N_2	N_3	N_4	N_5
1.00	1000	1000	1000	1000	1000
0.93	1144	1070	100	934	873
1.07	873	934	1000	1070	1144
0.50	4000	2000	1000	500	250
2.00	250	500	1000	2000	4000

(b) CIFAR

Table 5.3: Number of neurons in each hidden layer for the different factors in the architecture experiment.

5.4.2 Observations

The left-hand side of figure 5.11 shows that the deltas propagate through the network as theorised with the initial parameters. For a rectangular SNN, the variance of the deltas increases. Also a stronger increase can be found for the conic network with factor 1.07. The model with a slowly decreasing number of neurons (factor 0.93), preserves the variance through the network and the other networks lead to a decreasing variance. As the network is learning, the factor increases due to the variance increase of the weights, causing the delta propagation to change. How the deltas look after hundred epochs of learning is illustrated in the right-hand side of figure 5.11. Note that in all cases the variance propagation towards lower layers has increased.

From the learning curves in figures 5.12 and 5.13, we can draw some interesting conclusions. Although we expected networks with a steady or increasing variance towards the lower layers to be better, we find that exactly those networks with lower variance in the first layers outperform all of the other models. It could even be stated that it is better to have low-variance deltas in lower layers. This statement is backed by the observation that for models where the variance in lower layers is even more amplified (factor 2), training is consistently slower than for the other models in our experiments.

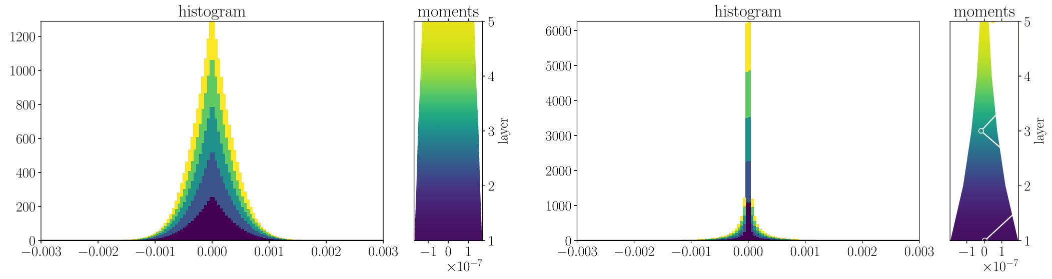
5.4.3 Learning from the Back

These observations suggest that having lower variance deltas in the first layers enables faster learning. One possible explanation for this phenomenon could be that it is easier for the network to start learning from the back. By reducing the variance of the error signal towards the first layers, the network can focus on updating the last layer. As training proceeds, the variance of the weights increases and thus also the factor (4.14) by which the variance of the deltas (4.11) changes. This way, the variance dampening disappears and the lower layers of the network can be updated as well.

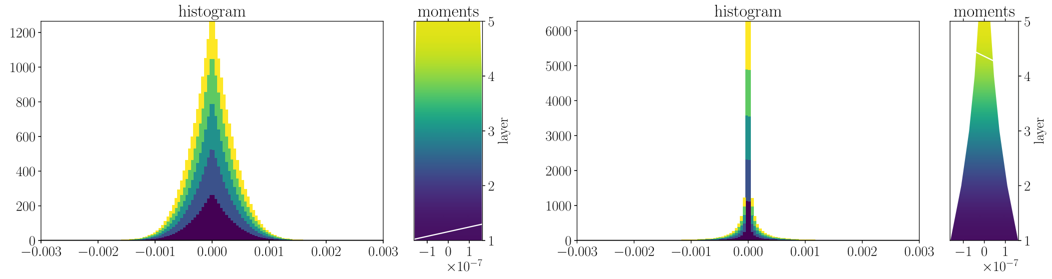
Ideally, the network would be able to start learning by focusing on the first layer, so that it gets a chance to build solid, low-level features. However, in a MLP, the back propagation of large deltas to the first layer, forces the network to learn in all layers at once. This is due to the fact that there is no other way to get the information from an error, which is made in the last layer, to the first layer without propagating it through the other layers of the network.

5.5 Optimisers

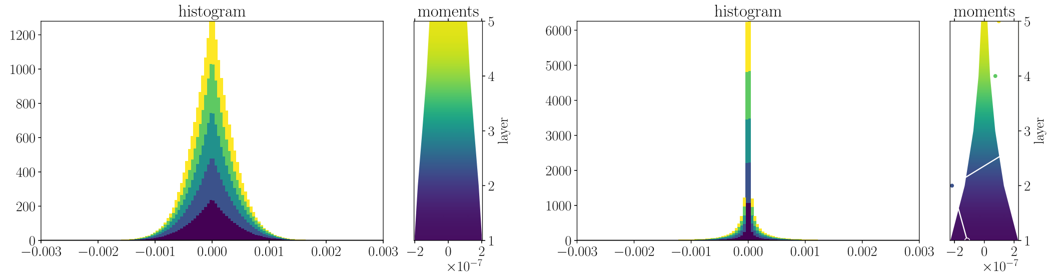
We shortly discussed the influence of momentum on the moment changes due to weight updates in section 4.4.3. Theoretically, we found that the variance of the weights should increase faster when using momentum. In table 5.4, we listed the variances of weights before and after learning for a network trained with plain SGD, regular momentum (3.20) and the variant from Nesterov (3.23) with $\mu = 0.9$. From the numbers in the table, we can conclude that the variance of the weights does indeed increase faster. However,



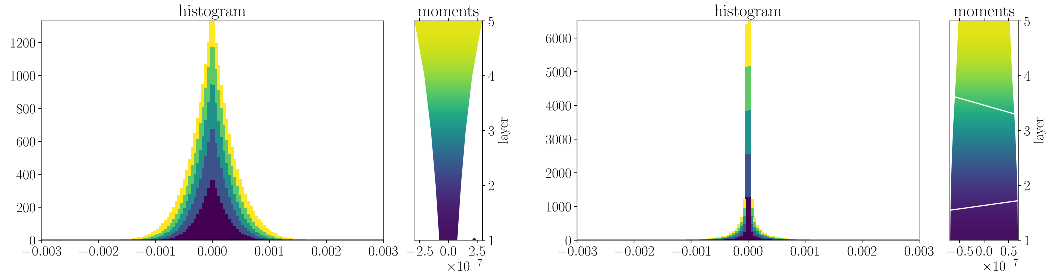
(a) rectangular



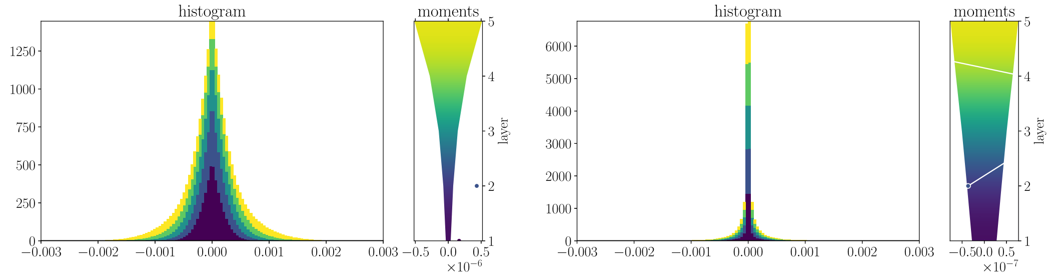
(b) conic 0.93



(c) conic 1.07

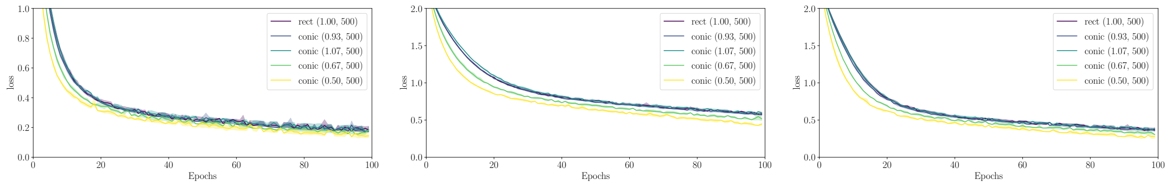


(d) conic 0.67



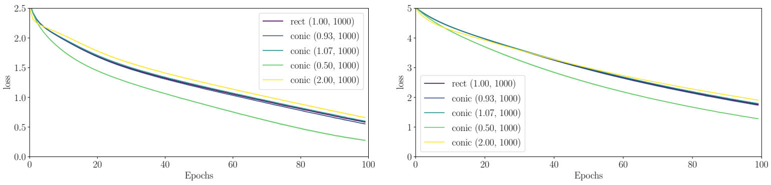
(e) conic 0.5

Figure 5.11: Distributions of deltas δ in the MNIST-BG-RAND baseline model with different architectures. The left-hand side shows the deltas for the first update and the right-hand side represents the deltas after 100 epochs.



(a) MNIST (b) MNIST-BG-IMG (c) MNIST-BG-RAND

Figure 5.12: Training loss for different network architectures. The factors for the architectures are (from dark to bright): *factor 1*, *factor 0.93*, *factor 1.07*, *factor 0.67* and *factor 0.50*. For the concrete number of neurons in the hidden layers for each factor, see table 5.3



(a) CIFAR-10 (b) CIFAR-100

Figure 5.13: Training loss for different network architectures. The factors for the architectures are (from dark to bright): *factor 1*, *factor 0.93*, *factor 1.07*, *factor 0.50* and *factor 2.00*. For the concrete number of neurons in the hidden layers for each factor, see table 5.3

layer		$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
SGD	σ_W^2	0.326e-3	1.001e-3	1.000e-3	1.000e-3	0.999e-3
	${\sigma_W^2}'$	0.328e-3	1.007e-3	1.007e-3	1.007e-3	1.007e-3
momentum	σ_W^2	0.325e-3	1.001e-3	1.001e-3	1.000e-3	1.000e-3
	${\sigma_W^2}'$	0.342e-3	1.044e-3	1.041e-3	1.032e-3	1.028e-3
Nesterov	σ_W^2	0.326e-3	0.999e-3	1.001e-3	1.000e-3	0.999e-3
	${\sigma_W^2}'$	0.342e-3	1.042e-3	1.041e-3	1.032e-3	1.027e-3

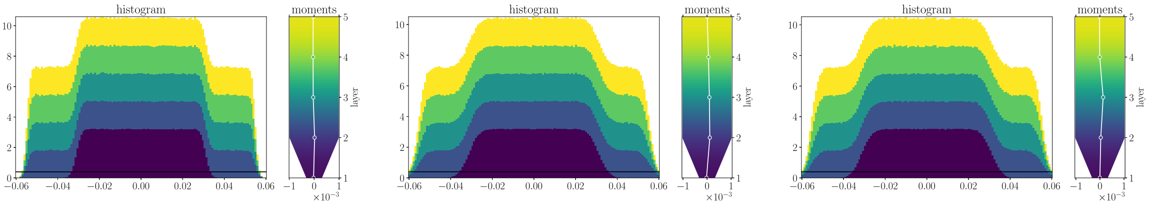
Table 5.4: Values for variance of weights w in networks trained with *SGD*, *momentum* and *Nesterov momentum*, before (σ_W^2) and after (${\sigma_W^2}'$) learning for hundred epochs.

both regular and Nesterov momentum still do not break the self-normalisation for quite a number of updates.

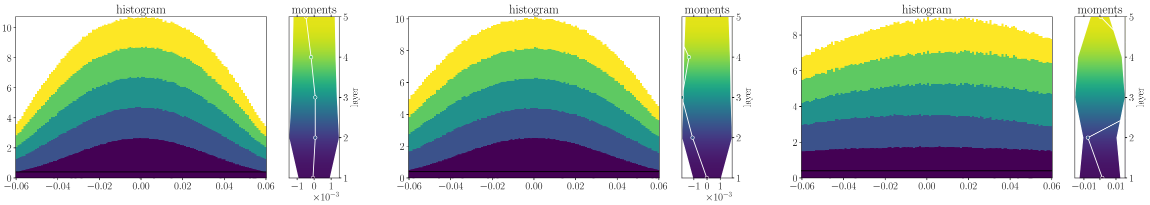
The effect of the momentum terms and other optimisers on learning speed is shown in figure 5.15. First of all, it makes clear that momentum can improve the learning speed of self-normalising networks significantly. Also, there seems to be no significant difference between regular and Nesterov momentum in our experiments.

Apart from the momentum, we also considered some commonly used adaptive learning rate schedules. For the MNIST family, these more complex forms of gradient descent seem to outperform SGD and both variants of momentum. However, for the CIFAR models, interesting things seem to be happening. First thing to be noted is that Adagrad did not learn at all. Also RMSProp and Adam seem to lead to degeneration, although they do seem to learn pretty well in the beginning. The only optimisers that seem to learn fast and steady are Adadelta and Adamax.

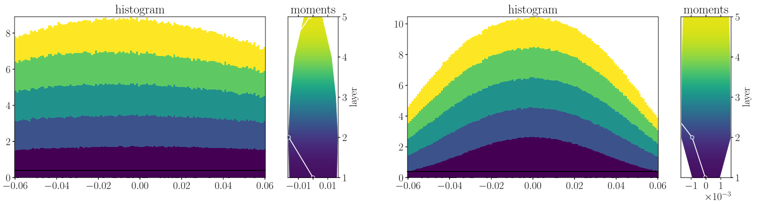
When taking a look at figure 5.14, we find that the variance of the weights increases significantly for adaptive learning rate schedules. Especially for Adam and RMSProp, the variance of the weights has grown by a factor of ten. However, also for the other adaptive learning rate schedules, the weights grow outside of the range that is required for self-normalisation (see section 4.3.2).



(a) SGD (b) momentum (c) Nesterov

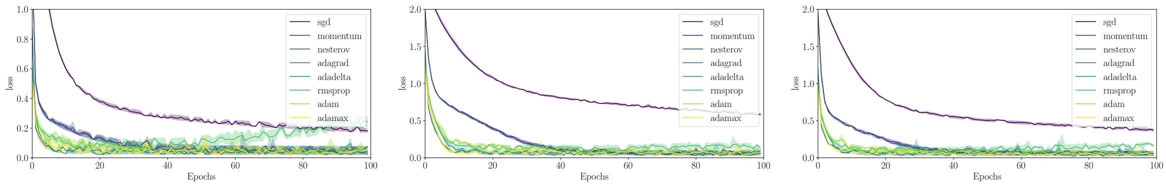


(d) Adagrad (e) Adadelta (f) RMSProp

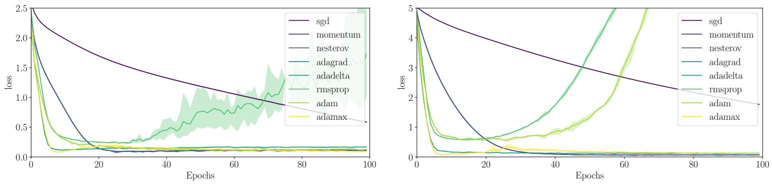


(g) Adam (h) Adamax

Figure 5.14: Distributions of weights w in the CIFAR-10 baseline model with different optimisers after 100 epochs of learning.



(a) MNIST (b) MNIST-BG-IMG (c) MNIST-BG-RAND



(d) CIFAR-10 (e) CIFAR-100

Figure 5.15: Training losses for models that have been trained using different optimisers. The optimisers are (from dark to bright): *SGD*, *SGD with momentum*, *SGD with Nesterov momentum*, *Adagrad*, *Adadelta*, *RMSProp*, *Adam* and *Adamax*.

6 Conclusion

In this thesis, we have analysed the moment dynamics in SNNs. By making assumptions on the moments of the error signal in neural networks, we were able to express how the moments change in a MLP. One of the main conclusions of this analysis, is that the variance of the weights steadily increases. We empirically justified these results for SNNs with SELU activations. We also found that due to the little variability in the error signal, the variance of the weights increases very slowly. This allows SNNs to stay in their self-normalising regime for a massive amount of updates.

We also extended the SNNs with bias units and included these in our analysis. We came to the conclusion that they introduce another source of variance increments. This means that a SNN with bias units generally gets out of its self-normalising regime faster than an equivalent network without biases. Although adding bias units does not seem to harm learning in our experiments, there is no indication that it could provide any benefits for training.

In the analysis of the dynamics, we also considered the effect of momentum. It turns out that both regular and Nesterov momentum increase the variance of the weights faster than SGD. We empirically verified these results and considered a collection of adaptive learning rate schedules. Whereas the momentum terms keep SNNs in their self-normalising regime, the effectiveness of adaptive learning rate schedules causes the weights to get outside of the self-normalising range. Although we did not include a theoretical analysis and only observed this empirically for RMSProp and Adadelta, it seems that there is a chance that adaptive learning rate schedules degenerate during training. Adagrad, on the other hand, does not seem to learn anything at all for certain tasks.

The architecture of the network, more specifically the ratio of neuron counts in neighbouring layers, was also found to have an impact on the variance of the back propagated errors in SELU networks. In our experiments, we found that conic architectures with a decreasing number of neurons towards the outputs, perform significantly better. We also came to the remarkable decision that exactly this kind of architecture causes the variance of the deltas to be dampened in lower layers. Networks for which the error variance was amplified in the first layers, also learned slightly slower. This seems to indicate that learning benefits from low-variance updates in lower layers.

Acronyms

AI artificial intelligence
ANN artificial neural network

BNN biological neural network
BP backpropagation

CNN convolutional neural network

DBN deep belief network
DL deep learning
DNN deep neural network

ELU exponential linear unit
EPE expected prediction error
ERM empirical risk minimisation

FNN feedforward neural network

GOFAI good old-fashioned AI

i.i.d. independently and identically distributed

LSTM long short-term memory

ML machine learning
MLP multilayer perceptron

NN neural network

RBM restricted Boltzmann machine
ReLU rectified linear unit
RNN recurrent neural network

SELU scaled exponential linear unit
SGD stochastic gradient descent
SNN Self-Normalising neural network
SRM structural risk minimisation
SVM support vector machine

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

Amodei, Dario et al. (2016). 'Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin'. In: *Proceedings of the 33rd ICML, New York, New York, USA.* Vol. 48. Proceedings of Machine Learning Research. PMLR, pp. 173–182. URL: `http://proceedings.mlr.press/v48/amodei16.html`.

Ba, Lei Jimmy, Ryan Kiros and Geoffrey E. Hinton (2016). 'Layer Normalization'. In: *CoRR* abs/1607.06450. URL: `http://arxiv.org/abs/1607.06450`.

Baldi, Pierre (2012). 'Autoencoders, Unsupervised Learning, and Deep Architectures'. In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning, Bellevue, Washington, USA.* Vol. 27. Proceedings of Machine Learning Research. PMLR, pp. 37–49. URL: `http://proceedings.mlr.press/v27/baldi12a.html`.

Banach, Stefan (1922). 'Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales'. In: *Fundamenta Mathematicae* 3.1, pp. 133–181. ISSN: 0016-2736.

Bergstra, James and Yoshua Bengio (2012). 'Random Search for Hyper-Parameter Optimization'. In: *Journal of Machine Learning Research* 13, pp. 281–305. ISSN: 1532-4435. URL: `http://dl.acm.org/citation.cfm?id=2188395`.

Bottou, Léon and Olivier Bousquet (2007). 'The Tradeoffs of Large Scale Learning'. In: *Advances in NIPS 20, Vancouver, British Columbia, Canada.* Curran Associates, Inc., pp. 161–168. URL: `http://papers.nips.cc/paper/3323-the-tradeoffs-of-large-scale-learning`.

Chollet, François et al. (2015). *Keras.* `https://github.com/fchollet/keras`.

Clevert, Djork-Arné, Thomas Unterthiner and Sepp Hochreiter (2015). 'Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)'. In: *Conference Proceedings of the 4th ICLR, San Juan, Puerto Rico, USA.* URL: `http://arxiv.org/abs/1511.07289`.

Duchi, John C., Elad Hazan and Yoram Singer (2011). 'Adaptive Subgradient Methods for Online Learning and Stochastic Optimization'. In: *Journal of Machine Learning Research* 12, pp. 2121–2159. ISSN: 1532-4435. URL: `http://dl.acm.org/citation.cfm?id=2021068`.

Gers, Felix A., Jürgen Schmidhuber and Fred A. Cummins (2000). 'Learning to Forget: Continual Prediction with LSTM'. In: *Neural Computation* 12.10, pp. 2451–2471. ISSN: 0899-7667. DOI: `10.1162/089976600300015015`. URL: `https://doi.org/10.1162/089976600300015015`.

Glorot, Xavier and Yoshua Bengio (2010). ‘Understanding the difficulty of training deep feedforward neural networks’. In: *Proceedings of the 13th AISTATS, Sardinia, Italy.* Vol. 9. Proceedings of Machine Learning Research. PMLR, pp. 249–256. URL: `http://proceedings.mlr.press/v9/glorot10a.html`.

Glorot, Xavier, Antoine Bordes and Yoshua Bengio (2011). ‘Deep Sparse Rectifier Neural Networks’. In: *Proceedings of the 14th AISTATS, Fort Lauderdale, Florida, USA.* Vol. 15. Proceedings of Machine Learning Research. PMLR, pp. 315–323. URL: `http://proceedings.mlr.press/v15/glorot11a.html`.

Goodfellow, Ian J., Yoshua Bengio and Aaron C. Courville (2016). *Deep Learning.* Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-03561-3. URL: `http://www.deeplearningbook.org/`.

Graves, Alex (2012). *Supervised Sequence Labelling with Recurrent Neural Networks.* Studies in Computational Intelligence. Springer. ISBN: 978-3-642-24796-5. DOI: `10.1007/978-3-642-24797-2`. URL: `https://doi.org/10.1007/978-3-642-24797-2`.

He, Kaiming et al. (2015). ‘Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification’. In: *Proceedings of the IEEE ICCV, Santiago, Chile.* IEEE, pp. 1026–1034. DOI: `10.1109/ICCV.2015.123`. URL: `http://ieeexplore.ieee.org/document/7410480/`.

– (2016). ‘Deep Residual Learning for Image Recognition’. In: *Conference Proceedings of the 29th CVPR, Las Vegas, Nevada, USA.* IEEE Computer Society, pp. 770–778. DOI: `10.1109/CVPR.2016.90`. URL: `https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html`.

Hinton, Geoffrey E., Simon Osindero and Yee-Whye Teh (2006). ‘A Fast Learning Algorithm for Deep Belief Nets’. In: *Neural Computation* 18.7, pp. 1527–1554. ISSN: 0899-7667. DOI: `10.1162/neco.2006.18.7.1527`. URL: `https://doi.org/10.1162/neco.2006.18.7.1527`.

Hochreiter, Sepp (1991). ‘Untersuchungen zu Dynamischen Neuronalen Netzen’. PhD thesis. Technische Universität München.

– (1998). ‘The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions’. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.2, pp. 107–116. ISSN: 0218-4885. DOI: `10.1142/S0218488598000094`. URL: `https://doi.org/10.1142/S0218488598000094`.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). ‘Long Short-Term Memory’. In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

Hornik, Kurt (1991). ‘Approximation capabilities of multilayer feedforward networks’. In: *Neural Networks* 4.2, pp. 251–257. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/0893-6080(91)90009-T`. URL: `http://www.sciencedirect.com/science/article/pii/089360809190009T`.

Hunter, J. D. (2007). ‘Matplotlib: A 2D graphics environment’. In: *Computing In Science & Engineering* 9.3, pp. 90–95. DOI: `10.1109/MCSE.2007.55`. URL: `https://matplotlib.org/index.html`.

Ioffe, Sergey and Christian Szegedy (2015). ‘Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift’. In: *Proceedings of the 32nd*

ICML, Lille, France. Vol. 37. Proceedings of Machine Learning Research. PMLR, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

James, Gareth et al. (2013). *An Introduction to Statistical Learning*. Springer-Verlag New York. ISBN: 978-1-4614-7138-7. DOI: 10.1007/978-1-4614-7138-7. URL: http://www.springer.com/gp/book/9781461471370.

Kingma, Diederik P. and Jimmy Ba (2015). 'Adam: A Method for Stochastic Optimization'. In: *Conference Proceedings of the 3rd ICLR, San Diego, California, USA*. URL: http://arxiv.org/abs/1412.6980.

Klambauer, Günter et al. (2017). 'Self-Normalizing Neural Networks'. In: *CoRR* abs/1706.02515. Accepted for publication at NIPS 2017. URL: http://arxiv.org/abs/1706.02515.

Krizhevsky, Alex, Vinod Nair and Geoffrey Hinton (2009). *Learning Multiple Layers of Features from Tiny Images*. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

Langley, Pat (2011). 'The changing science of machine learning'. In: *Machine Learning* 82.3, pp. 275–279. ISSN: 0885-6125. DOI: 10.1007/s10994-011-5242-y. URL: http://dx.doi.org/10.1007/s10994-011-5242-y.

Larochelle, Hugo, Dimitru Erhan and Courville Aaron. *Variations on the MNIST digits*. This dataset was introduced in Larochelle, Dumitru Erhan et al., 2007. URL: http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/MnistVariations.

Larochelle, Hugo, Dumitru Erhan et al. (2007). 'An empirical evaluation of deep architectures on problems with many factors of variation'. In: *Proceedings of the 24th ICML, Corvallis, Oregon, USA*. ACM, pp. 473–480. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273556. URL: http://doi.acm.org/10.1145/1273496.1273556.

LeCun, Yann, Yoshua Bengio and Geoffrey E. Hinton (2015). 'Deep learning'. In: *Nature* 521.7553, pp. 436–444. ISSN: 0028-0836. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

LeCun, Yann, Léon Bottou, Yoshua Bengio et al. (1998). 'Gradient-based learning applied to document recognition'. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791. URL: http://ieeexplore.ieee.org/document/726791/.

LeCun, Yann, Léon Bottou, Genevieve B. Orr et al. (1998). 'Efficient BackProp'. In: *Neural Networks: Tricks of the Trade*. Vol. 1524. Lecture Notes in Computer Science. Springer, pp. 9–50. DOI: 10.1007/3-540-49430-8_2. URL: https://doi.org/10.1007/3-540-49430-8_2.

LeCun, Yann, Corinna Cortes and Christopher J. S. Burges. *The MNIST database of handwritten images*. This dataset was introduced in LeCun, Bottou, Bengio et al., 1998. URL: http://yann.lecun.com/exdb/mnist/.

McCulloch, Warren S. and Walter H. Pitts (1943). 'A logical calculus of the ideas immanent in nervous activity'. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. ISSN: 0092-8240. DOI: 10.1007/BF02478259. URL: http://dx.doi.org/10.1007/BF02478259.

Mitchell, Tom M. (1997). *Machine learning*. McGraw Hill series in computer science. McGraw-Hill. ISBN: 978-0-07-042807-2. URL: http://www.worldcat.org/oclc/61321007.

Mohri, Mehryar, Afshin Rostamizadeh and Ameet Talwalkar (2012). *Foundations of Machine Learning.* Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-01825-8. URL: `http://mitpress.mit.edu/books/foundations-machine-learning-0`.

Raven, Peter H. et al. (2014). *Biology.* McGraw-Hill. ISBN: 978-1-259-08081-4.

Al-Rfou, Rami et al. (2016). 'Theano: A Python framework for fast computation of mathematical expressions'. In: *CoRR* abs/1605.02688. Source code available on GitHub. URL: `http://arxiv.org/abs/1605.02688`.

Rumelhart, David E., Geoffrey E. Hinton and Ronald J. Williams (1986). 'Learning Internal Representations by Error Propagation'. In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Vol. 1. MIT Press, pp. 318–362. ISBN: 0-262-68053-X. URL: `http://dl.acm.org/citation.cfm?id=104279.104293`.

Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)* Pearson Education. ISBN: 978-0-13-207148-2. URL: `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html`.

Salimans, Tim and Diederik P Kingma (2016). 'Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks'. In: *Advances in NIPS 29, Barcelona, Spain.* Curran Associates, Inc., pp. 901–909. URL: `http://papers.nips.cc/paper/6114-weight-normalization-a-simple-reparameterization-to-accelerate-training-of-deep-neural-networks`.

Samuel, Arthur L. (1959). 'Some Studies in Machine Learning Using the Game of Checkers'. In: *IBM Journal of Research and Development* 3.3, pp. 210–229. ISSN: 0018-8646. DOI: `10.1147/rd.33.0210`. URL: `https://doi.org/10.1147/rd.33.0210`.

Saxe, Andrew M., James L. McClelland and Surya Ganguli (2014). 'Exact solutions to the nonlinear dynamics of learning in deep linear neural networks'. In: *Conference Proceedings of the 2nd ICLR, Banff, Alberta, Canada.* URL: `https://arxiv.org/abs/1312.6120`.

Schmidhuber, Jürgen (2015). 'Deep learning in neural networks: An overview'. In: *Neural Networks* 61, pp. 85–117. ISSN: 0893-6080. DOI: `10.1016/j.neunet.2014.09.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608014002135`.

Snoek, Jasper, Hugo Larochelle and Ryan P. Adams (2012). 'Practical Bayesian Optimization of Machine Learning Algorithms'. In: *Advances in NIPS 25, Lake Tahoe, Nevada, USA*. Curran Associates, Inc., pp. 2960–2968. URL: `http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms`.

Springenberg, Jost Tobias et al. (2015). 'Striving for Simplicity: The All Convolutional Net'. In: *Workshop Proceedings of the 3rd ICLR, San Diego, California, USA*. URL: `http://arxiv.org/abs/1412.6806`.

Sutskever, Ilya et al. (2013). 'On the importance of initialization and momentum in deep learning'. In: *Proceedings of the 30th ICML, Atlanta, Georgia, USA*. Vol. 28. Proceedings of Machine Learning Research 3. PMLR, pp. 1139–1147. URL: `http://proceedings.mlr.press/v28/sutskever13.html`.

Turing, Alan M. (1950). 'Computing machinery and intelligence'. In: *Mind* LIX.236, pp. 433–460. ISSN: 0026-4423. DOI: `10.1093/mind/LIX.236.433`. URL: `http://dx.doi.org/10.1093/mind/LIX.236.433`.

Werbos, Paul J. (1974). 'Beyond regression: New tools for predictions and analysis in the behavioral science'. PhD thesis. Harvard University.

Wu, Yonghui et al. (2016). 'Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation'. In: *CoRR* abs/1609.08144. URL: `http://arxiv.org/abs/1609.08144`.

Zeiler, Matthew D. (2012). 'ADADELTA: An Adaptive Learning Rate Method'. In: *CoRR* abs/1212.5701. URL: `http://arxiv.org/abs/1212.5701`.

Pieter-Jan Hoedt

suadentque cadentia sidera somnos - Aeneas

Education

2015–now	**Master of Science**, *Johannes Kepler University*, Linz, Austria. Computer Science
2012–2015	**Bachelor of Science**, *KU Leuven*, Kortrijk, Belgium, *Cum Laude*. Informatics
2006–2012	**High School**, *Sint-Jan Berchmanscollege*, Avelgem, Belgium. Latin and Mathematics

Working Experience

Relevant Student Jobs

2017	**Tutor Computer Graphics Institute**, *CG Institute at JKU*, Linz. Assist lecturer during CG labs
2015	**Research Assistant**, *KULAK*, Kortrijk. Continuation of my bachelor thesis on Dedekind numbers

Languages

Dutch	**Native**	
English	**Effective operational proficiency**	
German	**Vantage or upper intermediate**	*learned by doing*
French	**Threshold or intermediate**	*used to be better*

Interests

Music	I play the guitar and learnt myself a bit piano afterwards.
Sports	I play basketball in competition, but I like almost any kind of sport.
Social	I have been a leader in the Chiro, a Belgian youth movement.

Outrijvestraat 16 – 8551 Heestert – Belgium
☎ *+32 479/85 13 65* • ✉ *hoedt.pieterjan@gmail.com*
mrTsjolder

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Linz, October 2017

Pieter-Jan Hoedt