Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Gradient-based learning in Deep Neural Networks

Bachelor Thesis

2018
Truc Lam Bui

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Gradient-based learning in Deep Neural Networks

Bachelor Thesis

Study programme:	Informatics
Study field:	2508 Informatics
Department:	Department of Computer Science
Supervisor:	RNDr. Kristína Malinovská, PhD.

Bratislava, 2018

Truc Lam Bui

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Truc Lam Bui
Study programme: Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study: Computer Science, Informatics
Type of Thesis: Bachelor´s thesis
Language of Thesis: English
Secondary language: Slovak

Title: Gradient-based learning in Deep Neural Networks

Annotation: Deep neural networks (DNN) [1] are currently one of the most popular techniques in machine learning. Deep learning models, such as the convolutional neural networks, are very powerful in classification tasks, such as image recognition. DNNs are most commonly trained using backpropagation [2] with various variants of gradient descent, which is very effective, but suffers from problems such as the vanishing and exploding gradient.

Aim: 1. Study the theory on DNN, and the most common problems in deep learning.
2. Based on your studies explore various ways of improving learning in deep networks (through new activation functions, weight adaptation, ...).
3. Experimentally evaluate proposed solutions and compare them with the canonical models.

Literature: [1] LeCun, Y., Bengio, Y. and Hinton, G., 2015. Deep learning. nature, 521(7553), p.436.
[2] Rumelhart, D.E., Hinton, G.E. and Williams, R.J., 1985. Learning internal representations by error propagation (No. ICS-8506). California Univ San Diego La Jolla Inst for Cognitive Science.

Keywords: artificial neural networks, deep learning, machine learning

Supervisor: RNDr. Kristína Malinovská, PhD.
Department: FMFI.KAI - Department of Applied Informatics
Head of department: prof. Ing. Igor Farkaš, Dr.

Assigned: 06.10.2017

Approved: 26.10.2017 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

....

Student Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Truc Lam Bui
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Gradient-based learning in Deep Neural Networks
Gradientové učenie hlbokých neurónových sietí

Anotácia: Hlboké neurónové siete [1] sú v súčasnosti jednou z najpoužívanejších metód v strojovom učení. Hlbokým modelom, akými sú napríklad konvolučné siete, sa veľmi darí v klasifikačných úlohách, ako je napríklad rozpoznávanie obrazu. Najčastejšie sú trénované pomocou algoritmu spätnej propagácie chýb [2] s rôznymi variantmi metódy najstrmšieho spádu (gradient descent). To je efektívne, ale má to svoje problémy, ako napríklad problém miznúceho a vybuchujúceho gradientu.

Cieľ: 1. Naštudovať teóriu hlbokých neurónových sietí a najčastejšie problémy hlbokého učenia, ako napríklad problém miznúceho gradientu.
2. Na základe štúdia preskúmať rôzne spôsoby vylepšenia učenia v hlbokých sieťach (napr. aktivačné funkcie, adaptáciu váh, ...).
3. Experimentálne vyhodnotiť navrhované riešenia a porovnať ich s kanonickými modelmi.

Literatúra:
[1] LeCun, Y., Bengio, Y. and Hinton, G., 2015. Deep learning. nature, 521(7553), p.436.
[2] Rumelhart, D.E., Hinton, G.E. and Williams, R.J., 1985. Learning internal representations by error propagation (No. ICS-8506). California Univ San Diego La Jolla Inst for Cognitive Science.

Kľúčové slová: umelé neurónové siete, hlboké učenie, strojové učenie

Vedúci: RNDr. Kristína Malinovská, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 06.10.2017

Dátum schválenia: 26.10.2017

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

Acknowledgement: I would like to thank my supervisor RNDr. Kristína Malinvská, PhD. for her countless advice, even if it was the same over and over again. I would also like to thank my sister Mgr. Phuong Bui Thi Mai for numerous conversations on the topic of deep learning.. Last but not least, I would like to thank the other family members for taking up the chores for me, as I would not have been able to finish the thesis on time otherwise.

Abstrakt

Hlboké neurónové siete sú aktuálne jeden z najsilnejších nástrojov strojového učenie. Oproti plytkým neurónovým sieťam, ich hĺbka im umožňuje modelovať dáta na rôznych úrovniach abstrakcie. Hlboké neurónové siete majú uplatnenie vo veľa rôznych oblastiach, ako napríklad samojazdiace autá, diagnostické metódy v medicíne, a strojový preklad. Tieto siete ale môžu byť veľmi veľké a ich trénovanie môže trvať aj niekoľko týždňov, v závislosti od hardvéru.

V našej bakalárskej práci skúmame rôzne nepreskúmané spôsoby, akými by sa dal proces učenia v hlbokých neurónových sieťach urýchliť. Navrhneme dve nové typy vrstiev: *posuvnú* vrstvu a *škálovaciu* vrstvu, ktoré už sú v podstate vo väčšine architektúrach neurónových sietí, ale nie na tých správnych miestach. Domnievali sme sa, že tieto dve vrstvy umožnia hlbokým sieťam sa rýchlejšie učiť, a túto hypotézu sme experimentálne overili. Výsledky našich experimentov naznačujú, že tieto vrstvy skutočne dokážu urýchliť učenie v porovnaní so sieťou bez nich.

Kľúčové slová: umelé neurónové siete, hlboké učenie, strojové učenie

Abstract

Deep neural networks are currently one of the most powerful machine learning methods. Unlike shallow neural networks, their depth allows them to model the input data at various levels of abstraction. They find application in diverse fields, such as self-driving cars, medical diagnostics, and machine translation. However, such networks can be very large and training them can take several weeks, depending on the hardware.

In our thesis, we explored various novel ways of improving learning in deep neural networks, without increasing the computational cost of doing so. We propose two novel types of layers, called *shifting* and *scaling* layers, which are essentially already present in most neural network architectures, but not at the right places. We hypothesised that putting them in a different position would enable the network to learn faster, and we evaluated this in a series of experiments. The results of our experiments suggest that our shifting and scaling layers innovations can indeed improve the pace of learning, compared to networks that do not utilize them.

Keywords: artificial neural networks, deep learning, machine learning

Contents

Introduction **1**

1 Machine Learning **3**

1.1 Example 1: regression 3
1.2 Example 2: classification 5
1.3 Fitting a model in general 7

2 Artificial Neural Networks **9**

2.1 History 9
2.2 Multilayer perceptron 10
2.3 Learning with gradient descent 11
2.4 Sigmoid neuron 12
2.5 Backpropagation 13

3 Computational Networks **16**

3.1 Introduction 16
3.2 Feedforward neural networks 17
3.2.1 Multilayer perceptron 17
3.2.2 Convolutional neural networks 19
3.3 Recurrent neural networks 21

4 Learning in Deep Neural Networks **23**

4.1 Optimization algorithms 23
4.1.1 Geometry of the problem 24
4.1.2 Stochastic gradient descent 25
4.1.3 Momentum 26
4.2 Vanishing and exploding gradient problem 26
4.3 Weight initialization 28
4.3.1 Xavier initialization 28
4.3.2 He initialization 29
4.4 Normalization 30

4.4.1 Batch normalization . . . 31
4.5 Regularization . . . 33
4.5.1 Dropout . . . 34
4.5.2 L_2 regularization (weight decay) . . . 34

5 Proposed solutions **36**
5.1 Shifting layer . . . 36
5.2 Scaling layer . . . 37
5.3 Combinations . . . 37

6 Experimental setup **39**
6.1 Datasets . . . 39
6.2 Architectures . . . 41
6.2.1 All convolutional . . . 41
6.2.2 Multilayer perceptron . . . 42
6.3 Methodology . . . 42
6.4 Implementation . . . 44

7 Individual experiments **45**
7.1 Initial experiment . . . 45
7.1.1 Hypothesis 1 . . . 48
7.1.2 Hypothesis 2 . . . 50
7.2 CIFAR10, convolutional with dropout . . . 50
7.3 Experiments with multilayer perceptron . . . 51
7.4 CIFAR100 . . . 52
7.5 Other experiments . . . 56

Conclusion **57**

Appendix A **62**

List of Figures

6.1 CIFAR10 . . . 40

7.1 Initial experiment, all variants . . . 46
7.2 Initial experiment, batch 1 . . . 47
7.3 Initial experiment, batch 2 . . . 47
7.4 Initial experiment, batch 3 . . . 48
7.5 Hypothesis 1, elementwise shifting . . . 49
7.6 Hypothesis 1, channelwise shifting . . . 49
7.7 CIFAR10, fast bias learning . . . 50
7.8 CIFAR10, all convolutional network with dropout . . . 51
7.9 Multilayer perceptron experiment . . . 52
7.10 Multilayer perceptron experiment, no dropout . . . 53
7.11 Multilayer perceptron experiment, with dropout . . . 53
7.12 CIFAR100 experiment . . . 54
7.13 CIFAR100 experiment, no dropout . . . 55
7.14 CIFAR100 experiment, with dropout . . . 55

List of Tables

5.1 Shifting and scaling combinations 38

6.1 Abbreviations of variants 40
6.2 All convolutional network 41
6.3 Multilayer perceptron 42

Introduction

Computers give great power to humans. Unlike previous forms of automation, which were in the form of other (un)willing human beings, industrial robots and complex machinery, a computer can be relatively easily *programmed* to do the task.

With such automation, the question of whether intelligence could be automated had emerged. Thus, the field of artificial intelligence was born. In the early days of computers, researchers were severely limited by the low processing power and the lack of data from which an artificial intelligence could learn. These constraints caused the field to largely abandon the idea of learning from data. Instead, the systems usually obtained knowledge in the form of rules and heuristics programmed into the system by humans.

With the processing power exponentially increasing, the limitation slowly faded away. The field of *machine learning* had been established, with the goal of using advanced statistical methods to make predictions based on data. The attention of AI researchers shifted from symbolic approaches to long-abandoned connectionist ones, most importantly *artificial neural networks*—simplified models of biological neural networks that were capable of learning.

Raw processing power on its own, however, would not be enough to enable these fields to flourish. Appropriate learning algorithms for the models had to be devised. It was not until relatively recently that we became capable of training *deep neural networks*, and this advance came from the discovery of a technique called *unsupervised pretraining*, not from the increase in processing power.

Deep neural networks are currently one of the most powerful tools of machine learning. They find application almost everywhere, especially with the increasing amounts of data that is being tracked, stored and processed. Such applications include self-driving cars, medical diagnostics, machine translation, speech recognition, image recognition and many others.

In our thesis, we explored various ways of increasing the pace at which deep neural networks can learn, without increasing the computational cost of doing so. These systems can be very large and they can take several weeks to train. Improving their learning speed by only a few percent could significantly decrease their training time. We also hope to obtain a better understanding of learning in deep neural networks.

In chapter one, we review some basic concepts from machine learning, which will provide us with a mathematical foundation for what learning from data, is and how it can be formalized.

In chapter two, we start with the history of artificial neural networks. We then present the core ideas behind this computational paradigm, such as how an artificial neural network performs computation, training with *gradient descent* and the *back-propagation* algorithm.

Chapter three takes a more modern approach towards neural networks, explaining them in terms of a more general structure called *computational network*. We look at the building blocks that make up such graphs, and how they can be put together into various architectures.

In chapter four, we review deep neural networks. We first show motivation behind their use by describing their advantages over shallow networks. We then explain the *vanishing* and *exploding gradient* problems, which contribute to difficulties in training them. We conclude with the methods commonly employed for training these networks.

Further chapters are dedicated to our solutions and enhancements. In chapter five, we present the solutions, and we take a look at how they can be integrated into networks. Chapter six contains the general informations about the performed experiments, and explains elements that are common among them. The seventh chapter contains the details and results of individual experiments.

Chapter eights concludes the thesis. It summarizes the results, and gives directions for further work that there is to be done.

Chapter 1

Machine Learning

Machine learning is a field of computer science. It studies and uses methods from various disciplines, such as statistics and mathematical optimization, that enable machines to "learn" from data, as opposed to programming the solution explicitly.

We will focus on the task of *supervised learning*, where we are given example input-output pairs, and we want to generalize to unseen situations.

1.1 Example 1: regression

Consider the following problem: we are given the areas $x_1, \dots, x_t$ (in squared meters) of t houses and their respective prices $y_1, \dots, y_t$ (in euros). Based on this data, which is generally called *training data*, we would like to predict the prices of other houses based on their area.

This is an example of a *regression* task, where the task is to predict some numerical value, and the closer we get, the better.

Suppose we came up with a magical formula for predicting house prices. This formula has the form of a function $f : \mathbb{R}^+ \to \mathbb{R}^+$. How can we determine whether it is good or bad? We can measure its performance on the training data: for each house i, we compare the predicted value $f(x_i)$ with its real price y_i. The closer these two values are, the better.

Formally, we have a *loss function* L that takes two arguments: the predicted value and the real value, and returns a number representing the error. Our choice of the loss function determines what we see as "good" and how good it is. Common choices are the squared error $L(y', y) = (y - y')^2$ and the absolute error $L(y', y) = |y - y'|$.

The *training error* is the sum of errors over the training data:

$$J(f) = \sum_{i=1}^{t} L(f(x_i), y_i). \tag{1.1}$$

However, simply finding a function f that minimizes the training error is not enough. Consider the following formula for price prediction: we check if the input area x is in the provided data. If it is, return the corresponding y. If it is not, return a random value.

Such a solution would achieve 0 training error in the case that all x_i are distinct, and would therefore be considered perfect by the aforementioned metric. However, it is not useful, because it fails to *generalize* to unseen data. A good solution should not have low training error, but it should have low *generalization error*, that is, error on unseen data. It should come with an appropriate *inductive bias* [22]: a set of assumptions that it uses to predict outputs on novel inputs.

Suppose that we visualize the data, and there is an almost linear relationship between house areas and house prices. Then, it would be reasonable to model the relationship as a linear one, and consider only such functions when minimizing the training error. Hence we restrict the set of functions to consider to those of the form $f_{a,b}(x) = ax + b$ for some parameters a, b.

We are thus interested in

$$\arg\min_{a,b} \left(\sum_{i=1}^{t} L(ax_i + b, y_i) \right), \tag{1.2}$$

and this problem can be approached by methods from mathematical optimization. We can expect the solution to generalize well to unseen data, since we have observed that the relationship is almost linear.

Note that we could have easily decided to model the relationship as a quadratic one, that is, we would consider functions of the form $f(x) = ax^2 + bx + c$. This would lead to a larger set of functions we consider. We call this set the *set of hypotheses* and denote it H.

A larger set of hypotheses leads to lower training error, but it could model some of the noise in the data instead of the underlying relationship, leading to higher generalization error. This phenomenon is called *overfitting* [9]. It can be resolved by a smaller set of hypotheses, more training data, or methods specifically designed to combat overfitting, called *regularization.* Common regularization techniques include:

- L_2 regularization, which introduces an additional term to the training error. The term is $\lambda \cdot \|w\|_2^2$, where w is the vector of weights that determines the function, and λ determines the amount of regularization. It uses the L_2 norm, defined as

$$\|w\|_2 = \sqrt{\sum_{i=1}^{|w|} w_i^2} \tag{1.3}$$

  It essentially penalizes large weights.

- Lasso regularization, which introduces the term $\lambda \cdot \|w\|_1$ to the training error. It uses the L_1 norm, defined as

$$\|w\|_2 = \sum_{i=1}^{|w|} |w_i|. \tag{1.4}$$

A too small set of hypotheses might not contain functions complex enough to capture the relationships in the data. This leads to both high training error and generalization error. The phenomenon is called *underfitting*, and can be resolved by considering a larger set of hypotheses.

1.2 Example 2: classification

Consider the following problem: we are given a grayscale image of a handwritten digit, and we are to determine which digit from 0 to 9 is depicted.

This is an example of a *classification* task, where we want to predict the class the input belongs to.

When is the image that of a zero? When is it that of a nine? We may come up with various descriptions, such as "zero is round and divides the plane in two regions" and "nine is a circle with a leg appended on the bottom right". But what if the image is that of an incomplete zero, which does not separate the plane into two parts? What if the nine is dashed, or even composed of other objects? Such descriptions often fail to capture edge cases that we had not thought of during design. And this all assumes that we have been able to put our intuitions into computer code, but for tasks of this nature, it is rather difficult.

Humans learn to recognize digits differently. Instead of learning from such descriptions, we learn from seeing many examples, and such descriptions are only a byproduct of our experience. This yields a different approach: predict the digit based on past data. We teach a computer program to recognize digits by providing it with many labelled images, that is, image-digit pairs.

The task can formalized as follows.

An image is an $r \times c$ array of pixels, where each pixel consists of one real number from $\langle 0, 1\rangle$: its brightness.

An algorithm for digit recognition is a function f that maps from the set of all possible images $I = \langle 0, 1\rangle^{r \cdot c}$ to the set of all classes $C = \{0, 1, \ldots, 9\}$.

Given are t labelled examples $(x_1, y_1), \ldots, (x_t, y_t)$: the training data. We can measure an algorithm's error on it. The number of incorrect classifications is

$$J(f) = \sum_{i=1}^{t} L(f(x_i), y_i),$$

where L is the 0–1 loss function:

$$L(y', y) = \begin{cases} 1, & \text{if } y' \neq y \\ 0 & \text{otherwise} \end{cases}$$

Find the function f which minimizes the training error while considering only functions from the set of hypotheses H. This can be approached using methods from mathematical optimization.

The 0–1 loss function does not have nice properties. For many optimization methods, it is important for the optimized function to be differentiable with respect to the parameters θ that determine the function f, but this loss function is not even continuous.

However, if the output of the algorithm is only the predicted class, there is no degree of how good or bad our prediction is. It is either correct or incorrect. To illustrate, if the prediction is 3 and the correct answer is 6, we did as badly as if we had predicted 4, thus the associated losses should be the same. Hence it makes no sense to use other loss functions, such as squared error or absolute error.

We will consider a different formalization of an algorithm, where it is possible to say how close its output is to the correct output.

The output of an algorithm will not only be the guessed class. Instead, for each class, it returns the "probability" with which it believes the input image belongs to this class. The guessed class is then the one with the highest probability. Formally, the output of an algorithm is a tuple $y' = (y'[1], \ldots, y'[|C|])$ such that $y[i] \geq 0$ for all $i \in \{1, \ldots, |C|\}$, and

$$\sum_{i=1}^{|C|} y'[i] = 1. \tag{1.5}$$

If the correct class is y, the target that our function should aim at is

$$\text{one-hot}(y) = (\underbrace{0, \ldots, 0}_{y}, 1, \underbrace{0, \ldots, 0}_{|C|-y-1}), \tag{1.6}$$

where one-hot(y) is the so called *one-hot encoding* of y. The training error of our algorithm f is

$$J(f) = \sum_{i=1}^{t} L(f(x_i), \text{one-hot}(y_i)), \tag{1.7}$$

where L is the loss function of our choice. Since we are comparing two real valued vectors, it is possible to use the squared error loss function,

$$L(y', y) = \sum_{i=1}^{|C|} (y[i] - y'[i])^2, \tag{1.8}$$

or the absolute error loss function,

$$L(y, y') = \sum_{i=1}^{|C|} |y[i] - y'[i]|. \tag{1.9}$$

In practice, the best results are obtained using the *cross-entropy* loss [8]:

$$L(y, y') = \sum_{i=1}^{|C|} y'[i] \cdot \log y[i]. \tag{1.10}$$

If we decide to go with the cross entropy loss, we must restrict the output probabilities to be positive, since log does not behave well at 0.

1.3 Fitting a model in general

In supervised learning, we are given t input-output pairs $(x_1, y_1), \ldots, (x_t, y_t)$, and we want to train an algorithm f that will be able to generalize well, that is, perform well on unseen data from the same distribution.

First, we restrict the set of functions to consider, the *set of hypotheses*, to some set H. In our thesis, the set of hypotheses will be a certain class of neural networks.

If the task is a classification task, it may be necessary to transform the data before searching for f. Denote the set of classes C. We can encode the outputs in the data using *one-hot encoding*:

$$\text{one-hot}(y) = (\underbrace{0, \ldots, 0}_{y}, 1, \underbrace{0, \ldots, 0}_{|C|-y-1}). \tag{1.11}$$

Using methods from mathematical optimization, we look for as good $f \in H$ as possible. How "good" a solution is is measured by performance on the *training data*, and performance is measured as the sum of errors over the training data:

$$J(f) = \sum_{i=1}^{t} L(f(x_i), y_i), \tag{1.12}$$

where L is a loss function of our choice. Common choices are the squared loss and absolute loss for regression tasks, and cross entropy loss for classification tasks. In our thesis, we will focus on the latter. The cross entropy loss is defined as

$$L(y, y') = \sum_{i=1}^{|C|} y'[i] \cdot \log y[i].$$

The function J that we are minimizing is in general called the *objective function*. In principle, it can include terms other than error on training data, such as regularization

terms that reduce the amount of *overfitting*, thus helping the model generalize outside of the training data.

This way, we can find the best solution with the chosen *hyperparameters* (set of hypotheses, used regularization technique, optimization algorithm, ...). We are interested in the solution's ability to generalize, that is, its expected error on data that was not used during training (*generalization error*). An estimate can be obtained by calculating the mean error on some data that was not used during training, commonly called *validation data*.

A different choice of hyperparameters could have lead to a better solution, that is, a solution with lower generalization error. Thus, in order to find as good a solution as possible, many hyperparameter choices are tried out. Each choice yields a different solution, and we take the best solution, where "best" is measured by its error on validation data.

Finally, once we are done with tuning the hyperparameters of the model, we can calculate an estimate of the solution's generalization error on some data that was used neither during training, nor during validation. This data is called *test data*, and provides an unbiased estimate of the real generalization error, since there is no way information from it could have leaked into the solution.

Chapter 2

Artificial Neural Networks

In this chapter, we explain *artificial neural networks*, computing systems which are inspired by biological neural networks such as the human brain. We first take a look at their history, explain the computation performed by such networks, and conclude with the basic learning algorithms.

2.1 History

The first attempt at mathematically modeling a biological neuron is the McCulloch–Pitts neuron [20].

The neuron is thought of as a function—it takes several inputs and computes a function of those inputs. To model the synaptic connections between biological neurons, inputs can be either excitatory or inhibitory. The neuron is either firing or not, represented by its output being 1 or 0, respectively.

If any of the inhibitory inputs is 1, the output is 0. Otherwise, the output is determined by the number of excitatory inputs that are set to 1. If this number is below some threshold t, the output is 0, otherwise it is 1.

Such artificial neurons can be connected to form networks. The networks are required to be acyclic, that is, no neuron's output depends on itself. The reason is that the McCulloch–Pitts neuron does not model the biological neuron in time.

By forming networks and setting the thresholds of individual neurons appropriately, it is possible to realize the logical AND, OR and NOT operations. Thus any boolean function can be realized. However, there were no known algorithms for training such networks.

Further research led to the development of Rosenblatt's model of biological neuron, called the *perceptron* [25]. It models inhibitory and excitatory connections through real-valued weights associated with inputs. The output of a neuron is 1 if the sum of its weighted inputs is greater than or equal to the neuron's threshold t, otherwise it is

0. Formally,

$$\text{output} = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq t \\ 0 & \text{otherwise} \end{cases},$$

where n is the number of inputs, $x_1, \ldots, x_n$ are the inputs and $w_1, \ldots, w_n$ are their respective weights.

Note that McCulloch–Pitts neuron is a special case of the perceptron, where excitatory inputs have weight 1 and inhibitory inputs have weight $-\infty$. Thus, networks of perceptrons can realize arbitrary boolean functions, too.

Rosenblatt also came with a learning algorithm for perceptrons. This learning is, however, restricted to the case of a single layer network, where there is only a single layer of inputs, and single layer of perceptrons representing the output of the network. There can be no intermediate neurons.

Also, there are limitations to what a perceptron can compute. One can look at the perceptron as a classifier: based on its output, the input is classified as either 0 or 1. Minsky and Rupert have shown that the perceptron is unable to implement the logical XOR [21]. More generally, it cannot fit data where the two classes of inputs are not linearly separable.

Networks of perceptrons do not have this limitation, but the learning algorithm is not applicable to them. Nevertheless, the perceptron represented a significant advancement in the field of artificial neural networks.

2.2 Multilayer perceptron

It was becoming clear to researchers that the way to go is to go deeper. Thus, they focused on networks with intermediate neurons, where not all the processing is done by the output neurons. The simplest of such networks are called *multilayer perceptrons*.

A multilayer perceptron is organized into multiple layers.

The first layer, called *the input layer*, contained neurons through which input data is fed into the network. At the beginning of computation, their output is determined by input data. These neurons are called *input neurons*, even though they are not really neurons, since they do not perform any processing of their own.

For example, if the input data is a 32×32 grayscale image, we would have $32 \cdot 32$ input neurons in the network, each representing a specific pixel.

In further layers, the inputs to a neuron are all of the outputs from the previous layer. Hence, the layers are called *dense* or *fully connected*, because all the connections between adjacent layers that can be there are there.

The last layer is composed of all the neurons that are parts of the output of the network. The layer is called the *output layer*.

For example, if the task is to determine which digit is depicted in the image, there would be one output neuron for each digit. Their outputs would represent whether the network believes the digit to be present in the image or not.

The layers between the first and the last are called *hidden layers*. If only a single hidden layer is included, the network is called *shallow*, as opposed to *deep networks*, which have more than one hidden layer.

Since the learning algorithm for the perceptron was not applicable to neurons in hidden layers, the parameters of those neurons were usually hand-engineered so that they would detect meaningful features. Only the output layer's neurons were trained with the perceptron learning algorithm. This was, of course, not the desired state of the matter, and researchers were looking for other ways to train multilayer perceptrons.

2.3 Learning with gradient descent

There was the idea of training artificial neural networks using the method of *gradient descent* from mathematical optimization. Gradient descent, also known as the method of steepest descent, works as follows:

Suppose we want to find the vector $w = (w_1, \ldots, w_n)$ which minimizes the function J. In the context of artificial neural networks, w would represent the vector of all the network's parameters, and J is the objective function, usually the error on the training data.

If the function is continuously differentiable, the geometry of J in a sufficiently small neighbourhood of w can be approximated by a hyperplane. This hyperplane's slope is determined by the *gradient* of J:

$$\nabla_J(w) = \left(\frac{\partial J}{\partial w_1}, \ldots, \frac{\partial J}{\partial w_n} \right). \tag{2.1}$$

If we move a small distance Δw from w, the difference between the new value of J and the previous value of J can be approximated as follows:

$$J(w + \Delta w) - J(w) \approx \Delta w \cdot \nabla_J(w) \tag{2.2}$$

We are looking for the direction in which J decreases the fastest in the Euclidean metric space. That is, we want to find some nonzero vector u that minimizes the change per distance:

$$\frac{u \cdot \nabla_J(w)}{\|u\|_2}. \tag{2.3}$$

Note that only the direction of u matters, since rescaling the vector u by scalar k yields

$$\frac{ku \cdot \nabla_J(w)}{\|ku\|_2} = \frac{ku \cdot \nabla_J(w)}{k\|u\|_2} = \frac{u \cdot \nabla_J(w)}{\|u\|_2}. \tag{2.4}$$

By simple application of the Cauchy–Schwarz inequality, we see that the optimal direction is in the direction opposite to the gradient:

$$u = -\nabla_J(w). \tag{2.5}$$

However, since the geometry of J at w can be approximated only on a small neighbourhood, we cannot move arbitrarily far in that direction. A small gradient could mean that w is close to the global minimum, and it could be detrimental to move too far away. Thus, we choose a small *learning rate* α and we adjust w as follows:

$$w := w - \alpha \cdot \nabla_J(w). \tag{2.6}$$

2.4 Sigmoid neuron

There are a few obstacles on the path to training artificial neural networks with gradient descent. The first is: what model of a neuron do we use? The perceptron is not a good choice, because it is not continuous, thus it also is not differentiable with respect to its weights.

We will now define the *sigmoid neuron*, which can be arrived at by searching for a differentiable approximation of the perceptron.

Note that the perceptron can be formulated differently. Let $x_1, \ldots, x_n$ be the inputs to the perceptron and let $w_1, \ldots, w_n$ be their respective weights. Instead of the threshold t, we consider its *bias* $b = -t$, which represents how easy it is to make the neuron fire. Then,

$$\text{output} = H\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{2.7}$$

where H is the Heaviside step function, defined as follows:

$$H(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.8}$$

In this formulation, it is easy to replace the step function with any other function. The used function is in general called the *activation* function. Its role in the artificial neural network is to introduce nonlinearity. Without them, the network would only be able to realize linear functions.

The sigmoid function, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.9}$$

is an approximation to the step function, in the following sense: If we multiply all the incoming weights $w_1, \ldots, w_n$ and the neuron's bias by a positive value c, then in the limit $c \to \infty$, we obtain behaviour identical to that of the step function.

The sigmoid function is also continuously differentiable. Its derivative is

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x) \cdot (1 - \sigma(x)). \tag{2.10}$$

If we compose the network out of neurons using the sigmoid activation, we ensure that the output of the network is continuously differentiable with respect to its weights. The output is then used to compute the training error, and this operation is also continuously differentiable. Thus gradient descent can be, in principle, applied.

2.5 Backpropagation

There is another obstacle. Gradient descent assumes that we are able to *compute* the partial derivatives of the objective J with respect to the weights and biases of the network. So far, we only know that they exist.

This was overcome with the introduction of the *backpropagation* algorithm [26]. Before we get into the details of the algorithm, we first take a look at the notation and conventions we will be using.

First, note that the objective function satisfies

$$J = \sum_{i=1}^{t} E_i, \tag{2.11}$$

where E_i denotes the error in the i-th training example. Then, the partial derivative of J with respect to a network's parameter p is

$$\frac{\partial J}{\partial p} = \sum_{i=1}^{t} \frac{\partial E_i}{\partial p}. \tag{2.12}$$

It is thus enough to compute the derivative of the error on a single training example, and we will focus on this.

We will denote the weighted sum of inputs to the i-th neuron, including bias, as net_i. We will call this sum the *net input* to the neuron. The output of the i-th neuron will be denoted out_i, and its activation function will be denoted f_i. The weight from neuron i to neuron j, in the case that connection from i to j exists, will be denoted $w_{i,j}$. The bias of neuron i will be denoted b_i.

Suppose we want to calculate the partial derivative of the error E with respect to $w_{i,j}$, the weight from neuron i to neuron j. The only value that directly depends on this weight is the net input of neuron j:

$$\text{net}_j = w_{i,j}\text{out}_i + \sum_{k \in I - \{i\}} w_{k,j}\text{out}_k, \tag{2.13}$$

where I is the set of neurons that are inputs to neuron j. By application of the chain rule, we obtain

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial \mathrm{net}_j}{\partial w_{i,j}} \cdot \frac{\partial E}{\partial \mathrm{net}_j} = \mathrm{out}_i \cdot \frac{\partial E}{\partial \mathrm{net}_j}. \tag{2.14}$$

A similar equation can be derived for derivative with respect to bias b_j:

$$\frac{\partial E}{\partial b_j} = \frac{\partial \mathrm{net}_j}{\partial b_j} \cdot \frac{\partial E}{\partial \mathrm{net}_j} = \frac{\partial E}{\partial \mathrm{net}_j} \tag{2.15}$$

In both cases, it is enough to know the partial derivative of J with respect to net_j. For that, we again use the chain rule. The only value that directly depends on net_j is $\mathrm{out}_j = f_j(\mathrm{net}_j)$. Thus,

$$\frac{\partial E}{\partial \mathrm{net}_j} = \frac{\partial \mathrm{out}_j}{\partial \mathrm{net}_j} \cdot \frac{\partial E}{\partial \mathrm{out}_j} = f_j'(\mathrm{net}_j) \cdot \frac{\partial E}{\partial \mathrm{out}_j}. \tag{2.16}$$

Now, we would like to express the partial derivative of E with respect to out_j. There are two cases, based on whether the neuron is an output neuron or not.

If it is, the error E directly depends on out_j, thus we can compute the partial derivative based on the exact form of E. For example, if we are minimizing the sum of squared errors on the training data, the partial derivative would be

$$\frac{\partial E}{\partial \mathrm{out}_j} = 2 \cdot (d_j - \mathrm{out}_j), \tag{2.17}$$

where d_j is the desired value of the neuron.

If there are other neurons that directly depend on this neuron's output, denote the set of those neurons O. For each neuron $i \in O$,

$$\mathrm{net}_i = w_{j,i}\mathrm{out}_j + \sum_{k \in I - \{j\}} w_{k,i}\mathrm{out}_k. \tag{2.18}$$

where I is the set of inputs to neuron i. Then, by the chain rule,

$$\frac{\partial E}{\partial \mathrm{out}_j} = \sum_{i \in O} \left(\frac{\partial \mathrm{net}_i}{\partial \mathrm{out}_j} \cdot \frac{\partial E}{\partial \mathrm{net}_i} \right) = \sum_{i \in O} \left(w_{j,i} \cdot \frac{\partial E}{\partial \mathrm{net}_i} \right). \tag{2.19}$$

In general, in order to calculate the derivative with respect to some value, we need to calculate derivatives with respect to some other values that occur later in the network. Occasionally, we also need the net input and output of a neuron.

We will compute the latter two in the *forward pass* of the algorithm, where we simply compute all the net inputs and outputs of all the neurons. We do so in the *topological ordering* of the network, so that whenever we are processing a neuron, the neurons it depends on have already been processed. We store both the net inputs and outputs for later use.

After the forward pass, we compute the partial derivatives in the *backward pass*. For each neuron, we first calculate the partial derivative with respect to its output, using equations 2.17 and 2.19. Then, we do the same with respect to its net input, using equation 2.16. We conclude with the neuron's bias, using equation 2.15, and all weights that lead to the neuron, using equation 2.14.

We process the neurons in the *reverse topological ordering* of the network. This ensures that whenever we are processing a neuron, the required partial derivatives have already been computed.

The time complexity of the algorithm on a single example is $O(n + m)$, where n is the number of neurons in the network and m is the number of connections between them. Thus, it is linear with respect to the size of the network. Taking into account all the t training examples, the time complexity is $O(t \cdot (n + m))$.

The space complexity is $O(n)$, because for each neuron, we have to remember its net input and output from the forward pass, so that we can use this information during the backward pass.

The algorithm is called backpropagation because whenever we are processing a neuron, we "propagate" its gradient back to its inputs. Based on the exact relationship between the input and the neuron, the gradient is adjusted prior to being propagated backward. These gradients are in literature commonly called *errors*, and we will also adopt this terminology.

Chapter 3

Computational Networks

This chapter is dedicated to *computational networks*, general structures that can be used to implement artificial neural networks. They can be seen as a generalization of artificial neural networks. First, we define them. Then we take a look at commonly used neural network architectures, how they can be expressed in terms of computational networks and what building blocks they are made of.

3.1 Introduction

A computational network consists of nodes, where each node represents a function application on its inputs. These functions can be parametrized, in the same way that artificial neurons have weights and biases. Note that nodes can represent very simple computations, such as adding its inputs or multiplying them. The output of a node can then be used in further computation.

Some nodes are designated as *input nodes*. Through these nodes, we provide input data to the network. They have no inputs. Nodes can also be designated as *output nodes*. The output nodes make up the output of the network.

Computational networks can be trained in a similar fashion to artificial neural networks. Gradient descent is used, with the backpropagation algorithm to calculate the partial derivatives with respect to the network's parameters. The only prerequisite is that the network output must be differentiable with respect to all the parameters. Thus if an input to a node is affected by any network parameter, the node must be differentiable with respect to that input, so that we can propagate the error back to that input. This is crucial for backpropagation to be applicable.

In general, nodes in the network need not work with single real-valued variables only, but they can also work with vectors, matrices and tensors. This has the benefit that many operations on these more complex objects, such as matrix multiplication, can be performed faster than their counterparts working with only single value at a

time. Operations on tensors are a standard part of most neural network frameworks, such as PyTorch [24] and TensorFlow [2].

An important application of tensors is evaluation of a network on multiple inputs. Instead of running the network t times, the inputs are stacked along a new dimension, called *batch dimension*, into a single tensor. For example, if the inputs are the (column) vectors $\vec{x_1}, \ldots, \vec{x_t}$, the resulting tensor is the matrix

$$\begin{pmatrix} \vec{x_1}^T \\ \vdots \\ \vec{x_t}^T \end{pmatrix}. \tag{3.1}$$

In further mathematical text, we will also assume that all vectors are column vectors.

3.2 Feedforward neural networks

In feedforward networks, the connections between nodes do not form a cycle. There is no notion of time, the output of each node is calculated only once.

The nodes of a feedforward network are usually arranged in multiple layers, in fashion similar to that of a multilayer perceptron. The first layer is the input layer, through which we provide input data to the network. Nodes in further layers can have inputs only among outputs from the previous layer. Thus, a layer is essentially a function that transforms the vector of previous layer's outputs. The last layer is the output layer, which contains all the network's outputs.

3.2.1 Multilayer perceptron

(Note that the name "multilayer perceptron" is a misnomer, since in general, the neurons in the network need not be perceptrons.)

In a multilayer perceptron, each neuron takes input from all neurons from the previous layer, performs a linear combination of them, and finally passes them through an activation function. This can be implemented as an alternating series of two types of layers: *dense layers* and *activation layers*. The former performs the linear combination, and the latter performs the activation. We now describe both of these layers in greater detail.

Dense layer

The input to a dense layer is a vector $\vec{x} = (x_1, \ldots, x_n)$ of size n. The i-th component of the vector corresponds to the output of the i-th neuron from the previous layer.

The output is a vector $\vec{y} = (y_1, \ldots, y_m)$ of size m, where the i-th component of the vector corresponds to the net input of the i-th neuron in this layer. The outputs are related to the inputs in the following way:

$$y_i = b_i + \sum_{j=1}^{n} w_{j,i} x_i, \tag{3.2}$$

where b_i is the bias of neuron i in this layer, and $w_{j,i}$ is the weight between the previous layer's neuron j and this neuron. In terms of vectors and matrices,

$$\vec{y} = \vec{b} + \mathbf{W}\vec{x}, \tag{3.3}$$

where $\vec{b} = (b_1, \ldots, b_m)$ is the vector of biases and $\mathbf{W}$ is the matrix of weights, both of which are learnable parameters of the layer. The weight matrix is given by

$$\mathbf{W}_{i,j} = w_{j,i}. \tag{3.4}$$

Activation layer

The activation layer is parametrized by a function f. The input to the layer is a vector $\vec{x} = (x_1, \ldots, x_n)$, and it represents the net inputs to the neurons on this layer. The output vector $\vec{y} = (y_1, \ldots, y_n)$ has the same size, and is obtained by applying f to the input.

We will now go over some common choices for the function f.

First, we take a look at functions f that perform an elementwise transformation on the input, that is, where each input dimension is transformed separately by the same function g. Common choices for g are:

- The sigmoid function:
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The hyperbolic tangent function:
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
Note that $\tanh(x) = 2\sigma(2x) - 1$, thus the two functions are related, and networks using one can be transformed into networks using the other. However, tanh has better properties when it comes to learning, and is preferred over the sigmoid.

- The rectified linear unit, ReLU:
$$\mathrm{ReLU}(x) = \max(0, x)$$
Even though its derivative is not continuous, in practice, networks utilizing ReLU learn many times faster than networks utilizing any of the above two activation functions. It is the current go-to activation function for most applications.

From among other activation functions, we mention the *softmax* activation. It is defined as follows:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{3.5}$$

The outputs are all from the interval $(0, 1)$ and can be interpreted as probabilities of some events occuring. Moreover, the sum of outputs is equal to 1, which corresponds to exactly one event occuring. Each y_i is increasing with respect to the aligned input x_i, and decreasing with respect to other inputs x_j. Thus, the larger the input, the more likely the corresponding event becomes.

Due to its unique properties, softmax is used mainly in classification tasks as the last layer of a network. The outputs from the softmax layer are interpreted as the "probabilities" of the input data belonging to the respective classes.

3.2.2 Convolutional neural networks

Convolutional neural networks are inspired by the way cells are organized in the animal visual cortex. Individual neurons respond only to a local region of the visual field, called their *receptive field*.

They are widely used in image processing tasks. One of their earliest applications was recognizing handwritten characters in documents [19]. Modern applications include self-driving cars, medical diagnostics, artificial intelligence for playing Go [28], and many others.

The input to a convolutional neural network is a tensor of shape $c \times w \times h$, where c is the number of channels, w is the width of the image and h is its height. The channel dimension is referred to as the *depth* of the input, and each individual channel can also be referred to as the *depth slice*.

Two types of layers are introduced: *convolution layers* and *pooling layers*.

Convolution layer

Convolution layers serve as filters that detect specific features in the input. They are based on two principles: locality and weight sharing.

First, locality means that small connected patches of the input generally make sense on their own, without having to look at the entire input. Thus, connecting a neuron only to a small patch of the input could lead to better generalization, since the neuron would not try to make sense out of distant, possibly unrelated, parts of the image.

Second, if it is worth detecting something in a specific region of the image, it may be useful to detect it in other regions as well. Objects can appear anywhere on the image, they are not constrained to a specific position, such as "in the lower left corner".

We thus force all neurons in the layer to detect the same feature, by restricting their weights to be the same. The weights are thus shared among all neurons in the layer.

Let us formalize it. Denote the input tensor $\mathbf{X}$. Each neuron is connected to a rectangular region of the input along its width and height, and is connected to all cells located there along the depth.

Each neuron in the layer thus sits on top of a small rectangular region of the input. Denote its upper left corner's coordinates i_0, j_0, and its width and height as w_k, h_k. The neuron performs a linear combination of values in this region:

$$y = b + \sum_{i=0}^{w_k-1} \sum_{j=0}^{h_k-1} \sum_{k=1}^{c} w_{i,j,k} \mathbf{X}_{i_0+i, j_0+j, k}, \tag{3.6}$$

where $w_{i,j,k}$ is the weight towards the input value at (i, j, k), and b is the bias of the neuron.

The matrix of weights, defined by $\mathbf{W}_{i,j,k} = w_{i,j,k}$, is called the *kernel* and the size of the rectangular region is called the *kernel size*.

Only rectangular regions which are completely inside the input are well defined. This forces the output to be smaller than the input, which may be undesirable. It can be resolved by *padding* the input beforehand, by expanding the input at the borders. The new input dimensions are usually set to zeros.

The neurons are usually laid on a grid. The amount of horizontal and vertical space between adjacent neurons, called *stride*, determines the sparsity of the convolution operation. With larger stride, the rectangular regions overlap less. This shrinks the output width and height to

$$w_o = \left\lceil \frac{w + 2w_p - w_k + 1}{w_s} \right\rceil, h_o = \left\lceil \frac{h + 2h_p - h_k + 1}{h_s} \right\rceil, \tag{3.7}$$

where w_p and h_p are the amounts of padding along the width and height, respectively. (The input is padded by these amounts on both borders.)

Commonly, convolution layers with the same stride, padding and kernel sizes are stacked on top of each other. Each sublayer detects a different feature and collectively, they create an output of shape $k \times w_o \times h_o$, where k is the number of kernels. This stack of layers is then usually called the convolution layer, instead of individual sublayers.

Pooling layer

Sometimes, the exact position of a certain feature in the image is not important. For example, to detect whether the image contains a boy playing frisbee, it is enough to detect the boy and the frisbee. The relative locations of one to another are not important.

This is accomplished by a pooling layer. The pooling layer considers each input feature individually, thus it operates on depth slices. Similar to a convolution layer, it also consider rectangular regions, parametrized by the kernel size, stride and padding. The difference is in the operation that is performed.

Pooling layer computes a predetermined function f on all values in the region. Common choices for the function f include the average of the region, called *average pooling*, and the maximum of the region, called *max pooling*. The latter been shown to perform better than average pooling [27], and is most commonly used.

Nowadays, pooling layers can be replaced by convolution layers with the same stride, kernel size and padding, as this has been shown to perform equally well if not better [29].

3.3 Recurrent neural networks

In a recurrent neural network, the neuron outputs evolve in time. Due to this, they are widely used in applications dealing with arbitrary sequences, such as time series prediction, natural language processing, speech recognition, and many others.

The values for input neurons are supplied from outside of the network at each time step. The remaining neurons are called *hidden*, and any hidden neuron can be designated as an output of the network. Such a network can output a value at each time step, or it can first process the entire input sequence and then output some final answer.

In the simplest recurrent neural networks, the output of a hidden node at time step t depends on the state of the network at the previous time step $t-1$ and the current input to the network, for $t > 0$. If there is not previous time step ($t = 0$), an initialization procedure sets the outputs of the nodes.

For example, consider a simple recurrent network with n hidden nodes and m input nodes. Let U be the set of input nodes, V be the set of hidden nodes, and denote $y_i(t)$ the output of node i at time t. Then,

$$y_i(t) = b_i + \sum_{u \in U} w_{u,i} y_u(t) + \sum_{v \in V} w_{v,i} y_v(t-1),$$

where b_i is the bias of node i, and $w_{j,i}$ is the weight of connection from node j to node i.

We will call a connection between two nodes Δt*-delayed* if the second node's value at time t directly depends on the first node's value at some earlier time $t - \Delta t$. In general, a recurrent neural network can include connections of arbitrary time delay.

The only condition that must hold is that connections with no delay must form an acyclic graph, that is, no value must depend on itself.

The backpropagation algorithm is extended to such networks by "unfolding them in time"—we decouple node outputs at individual time steps into separate nodes, and use backpropagation on the resulting computational network. This variant of backpropagation is known as *backpropagation through time.*

Formally, let $M_{\Delta t}$ be the set of all nodes such that there is a Δt-delayed connection from node i to that node. Then, we can calculate the error for neuron i at time t as follows:

$$\frac{\partial J}{\partial y_i(t)} = \sum_{\Delta t=0}^{\infty} \sum_{j \in M_{\Delta t}} \frac{\partial y_j(t+\Delta t)}{\partial y_i(t)} \cdot \frac{\partial J}{\partial y_j(t+\Delta t)} \tag{3.8}$$

The first factor in each summand can be calculated based on the exact relationship between $y_j t + \Delta t$ and $y_i(t)$. It is determined by the function that y_j computes and the values of its other inputs. Also, in the case that neuron i is an output neuron, there is an additional term that measures the direct impact on the objective J.

Recurrent networks can be implemented simply by reusing layers, that is, by applying the same layer with the same set of weights multiple times. For example, consider the simple recurrent network from the beginning of the section. Denote $\vec{y_U}(t)$ and $\vec{y_V}(t)$ outputs of input and hidden nodes, respectively, at time t. Then,

$$\vec{y_V}(t) = \vec{b} + \mathbf{W}_U \vec{y_U}(t) + \mathbf{W}_V \vec{y_V}(t-1), \tag{3.9}$$

where $\vec{b}$ is the vector of biases of hidden nodes, $\mathbf{W}_U$ is the matrix of weights between input nodes and hidden nodes, and $\mathbf{W}_V$ is the matrix of weights between hidden nodes. Both of the matrix multiplications and adding the bias can be carried out by a dense layer.

Chapter 4

Learning in Deep Neural Networks

This chapter is about deep neural networks. We present the advantages they have over shallow networks and the obstacles on the path towards their efficient training. We then take a look at some of the state of the art methods for efficient deep learning.

Deep networks are networks that contain more than one hidden layer. They may contain tens, hundreds or even thousands of layers. However, even shallow networks are able to approximate any continuous function to arbitrary precision [13]. What is the benefit of introducing additional hidden layers? Intuitively, their depth allows them to learn features at various levels of abstraction. For example, in an image processing application, an early layer could detect the presence of an edge oriented in a specific direciton, the next layer would be able to detect simple shapes, ... From a mathematical point of view, it turns out that for most functions, deep networks are able to approximate them using exponentially fewer nodes than shallow networks [23] [5].

Recurrent networks can be considered the deepest type of networks. One can see why once they are unfolded in time. The depth of the resulting network is proportional to the number of time steps, thus it can be much larger than the depth of any feedforward network.

4.1 Optimization algorithms

In this section, we take a closer look at the algorithm of gradient descent. We then mention other optimization algorithms that are, in some way, better performing variants of gradient descent. We go into greater detail for two specific variants that we will use during our experiments: *momentum* and *stochastic gradient descent.*

4.1.1 Geometry of the problem

Recall that in gradient descent, in each step, we calculate the gradient of the objective function J with respect to the network's weights. Then, for each weight w, we update its value based on the formula

$$w := w - \alpha \cdot \frac{\partial J}{\partial w}, \tag{4.1}$$

where α is the learning rate.

The idea behind gradient descent is that we move a small amount in the direction in which J decreases the fastest. This direction is opposite to the direction of the gradient.

However, this assumes that the Euclidean metric space, where distance is measured by the L_2-norm, is the right choice for the metric. In such a metric, all parameters have the same scale. This may not be the case, for example, consider the two following cases.

In the first, the task is to find x_1, y_1 that minimize $f_1(x_1, y_1) = x_1^2 + y_1^2$. The gradient of f_1 with respect to x_1, y_1 is

$$\frac{\partial f_1}{\partial x_1} = 2x_1,\ \frac{\partial f_1}{\partial y_1} = 2y_1. \tag{4.2}$$

In the second case, the task is to find x_2, y_2 that minimize $f_2(x_2, y_2) = x_2^2 + (ay_2)^2$, for some fixed a.

This is related to the first case by a simple reparametrization: $x_2 = x_1$ and $y_2 = \frac{y_1}{a}$. The scale of y_2 is a times smaller than the scale of y_1. Thus if we change y_1 by ε in the first case, the same effect is achieved in the second case by changing y_2 by $\frac{\varepsilon}{a}$.

However, the change made to y_2 by gradient descent is a times larger, not a times smaller:

$$\frac{\partial f_2}{\partial y_2} = 2y_2 = 2ay_1 = a \cdot \frac{\partial f_1}{\partial y_1}. \tag{4.3}$$

This is because we have not altered the metric in the second case, even though y was made a times smaller. To compensate for this, we should have considered the metric where distance between two points $\vec{p} = (p_1, p_2)$ and $\vec{q} = (q_1, q_2)$ is measured as

$$d(\vec{p}, \vec{q}) = \sqrt{(p_1 - p_2)^2 + \left(\frac{q_1 - q_2}{a}\right)^2}. \tag{4.4}$$

With this metric, we are looking for the vector $\vec{u} = (\Delta x_2, \Delta y_2)$ that minimizes the change per distance, where distance is measured by d. Thus we are minimizing

$$\frac{\vec{u} \cdot \nabla_{f_2}(x_2, y_2)}{\sqrt{\Delta x_2^2 + \left(\frac{\Delta y_2}{a}\right)^2}} \tag{4.5}$$

which, by the Cauchy–Schwarz inequality, yields

$$u = \left(-2x_2, -\frac{2y_2}{a}\right) = -\nabla_{f_2} \cdot \left(1, a^{-1}\right). \tag{4.6}$$

In general, the effect of a different metric is that prior to adjusting the weights, we rescale the components of the gradient. This is implemented by setting different learning rates for different network weights.

The chosen metric can have a very large impact on learning performance. Various algorithms for determining the "optimal" metric for learning have been devised, such as Adadelta [33], RMSProp [31] and Adam [16]. We only mention them, as we did not utilize them in our thesis.

4.1.2 Stochastic gradient descent

In machine learning applications, the objective function J consists of the training error E and possibly some other terms, such as regularization terms.

The training error is the mean of errors on individual examples, $E_1, \ldots, E_t$. Thus, if we took only a smaller set of training errors and computed the mean error on that set, we would obtain an approximation of the true training error. The benefit is that calculating the gradient on a smaller set of examples takes less time to compute. This is the idea behind *stochastic gradient descent*.

Instead of considering all training data, we consider only a small randomly chosen sample of the data, and calculate the gradient based on this sample. Denote the set of the chosen examples as M. The mean error on this sample is

$$J = \frac{1}{|M|} \cdot \sum_{i \in M} E_i. \tag{4.7}$$

The calculated derivative with respect to some network parameter p is then

$$\frac{\partial J}{\partial p} = \frac{1}{|M|} \cdot \sum_{i \in M} \frac{\partial E_i}{\partial p}. \tag{4.8}$$

The gradient calculated this way is an unbiased estimator of the true gradient.

To ensure even contribution of all training data, the following approach is usually taken. The training proceeds in *epochs*. At the beginning of each epoch, the training data are randomly shuffled and divided into minibatches of size b (except for the last minibatch, which may be smaller). Then, we proceed in steps. In each step, we consider the next minibatch. We approximate the gradient based on it, and adjust the network parameters using this estimate.

The time complexity of one step is thus reduced to $O(b \cdot (n + m))$.

The case when $b = 1$ is called *online learning*, where the training examples are provided one at a time. At first, it may seem best to choose $b = 1$ to provide the

maximum speed-up. However, to make use of fast operations on tensors, parallel processing and GPUs, the batch size is usually set to a larger value, suc as 32.

4.1.3 Momentum

Momentum is an enhancement that can be coupled with any gradient-based optimization method. It was first mentioned in the paper on backpropagation [26].

It works as follows. We do not adjust network parameters directly based on the gradient. Instead, for each parameter, we maintain its *velocity*, which is recalculated at each step based on the gradient. Afterwards, each parameter is adjusted by its velocity.

Formally, denote v the velocity of the parameter and g the gradient with respect to it. At each step, we first adjust the velocity based on the formula

$$v := \rho v - g, \tag{4.9}$$

where ρ is the momentum coefficient. The coefficient is the same for all parameters. Then, we adjust the parameter p:

$$p := p + \alpha v, \tag{4.10}$$

where α is the learning rate of the parameter.

With momentum, directions in which the derivative has consistently one sign will have high velocity, whereas directions with fluctuating derivative will not build up high velocity. This accelerates the passage through saddle points, where the gradient is almost zero, but there is a direction in which the objective slowly but steadily decreases. Another benefit is that with high velocity, it is possible to overcome small local minima (small "valleys" in the objective landscape), though these do not occur as often as saddle points [6].

4.2 Vanishing and exploding gradient problem

With great depth comes great difficulty with learning. The most common problems are the *vanishing gradient* and the *exploding gradient* problems. They were first observed by Bengio et al. [4] in recurrent neural networks, but the problems apply to deep feedforward networks as well.

If a network experiences the former, it means that earlier layers of the network are not learning as effectively as the later layers, because gradients in those earlier layers are smaller than the gradients in later layers. In other words, changes in weights in the later layers have a much bigger impact on the output than changes in the earlier layers.

If a network experiences the latter, the effect is reversed. Later layers are learning slower than the earlier layers, because gradients in earlier layers are larger.

Note that in the case of feedforward neural networks, the problem might be tied to some gradient descent variants only. If the scales of parameters were such that earlier weights had a larger scale than later weights, we could afford to change the earlier weights more. The two effects, the vanishing gradient and different scales, would cancel each other, and the layers would be learning evenly.

Determining the scales is a nontrivial task though, and even if we could determine them, there are no guarantees they would actually compensate for the vanishing gradient.

In the case of recurrent neural networks, changing a weight influences both the early and the later layers. Thus it is impossible to have a larger scale for parameters "only in the early layers".

Let us take a closer look at the problem. The following analysis is based on Hochreiter et al. [12]

Consider a multilayer perceptron with layers $L_1, L_2, \ldots, L_k$. Each L_i is a set of neurons in that layer. Denote the output of neuron i as out_i and the net input to neuron i as net_i. Denote the weight from neuron i to neuron j as $w_{i,j}$. For simplicity, we will assume that all neurons use the same activation function f.

Let us take a neuron a_0 from the layer l. The gradient with respect to its output is

$$\frac{\partial J}{\partial \mathrm{out}_{a_0}} = \sum_{a_1 \in L_{l+1}} \frac{\partial \mathrm{out}_{a_1}}{\partial \mathrm{out}_{a_0}} \cdot \frac{\partial J}{\partial \mathrm{out}_{a_1}} \tag{4.11}$$

$$= \sum_{a_1 \in L_{l+1}} w_{a_0,a_1} f'(\mathrm{net}_{a_1}) \cdot \frac{\partial J}{\partial \mathrm{out}_{a_1}} \tag{4.12}$$

$$= \sum_{a_1 \in L_{l+1}} \cdots \sum_{a_{k-l} \in L_k} \left(\frac{\partial J}{\partial \mathrm{out}_{a_{k-l}}} \cdot \prod_{i=1}^{k-l} \left(w_{a_{i-1},a_i} f'(\mathrm{net}_{a_i}) \right) \right) \tag{4.13}$$

We can thus see that, assuming the same set of weights, activation functions with smaller derivatives will tend to experience vanishing gradient more often. On the other hand, activation functions with larger derivatives will tend to experience exploding gradient more often.

During training, a node can become *saturated*, meaning that inputs to the node are mostly such that the derivative is small. This causes training to slow down, because no errors are flowing back through the node. ReLU alleviates the problem a bit, because its derivative is small only on one side.

4.3 Weight initialization

One aspect of training a network is often overlooked, and that is, how do we initialize the weights of the network?

From the aforementioned analysis, we can see that large weights lead to exploding gradients and small weights lead to vanishing gradients. Thus weights should be initialized in such a way that these two aspects are balanced.

The initialization procedure should introduce some variability into the network. To illustrate why, consider the extreme case of a multilayer perceptron where all weights are set to the same value. Weights going out from the same neuron will be receiving the same errors during backpropagation, thus they will stay the same forever, which severely limits the capabilities of the network.

4.3.1 Xavier initialization

In this section, we describe the Xavier initialization [7], also known as the Glorot initialization.

We will consider a multilayer perceptron for simplicity, and assume that the activations in the network can be approximated by the identity, $f(x) = x$. This is true for some activation functions when x is sufficiently close to 0, such as for tanh. For other functions, the reasoning has to be refined.

Let us take the perspective of a single neuron. We can measure the information that comes to us during backpropagation as the variance of the backpropagated error. To ensure that different layers are learning evenly, we want to ensure that this variance is the same for all neurons.

Suppose we have m outgoing connections with weights $w_1, \ldots, w_m$. We will sample each of these weights from the normal distribution with mean 0 and variance σ^2, for some σ. Our goal is to choose appropriate σ.

Let us take a look at the errors of neurons in the next layer:

$$\frac{\partial J}{\partial \text{out}_1}, \ \ldots, \ \frac{\partial J}{\partial \text{out}_m}. \tag{4.14}$$

Consider the probability distribution of each of them, and assume that they are independent and with variance 1. The error flowing to us is

$$\frac{\partial J}{\partial \text{out}} = \sum_{i=1}^{m} w_i \cdot \frac{\partial \text{out}_i}{\partial \text{net}_i} \cdot \frac{\partial J}{\partial \text{out}_i} = \sum_{i=1}^{m} w_i \cdot f'(\text{net}_i) \cdot \frac{\partial J}{\partial \text{out}_i}. \tag{4.15}$$

Since the neuron is in the linear regime of the activation function, we can approximate $f'(\text{net}_i)$ as 1, obtaining

$$\frac{\partial J}{\partial \text{out}} = \sum_{i=1}^{m} w_i \cdot \frac{\partial J}{\partial \text{out}_i}. \tag{4.16}$$

Each summand is the product of two independent random variables with mean 0, this the variance of each summand is simply the product of its factors variances. When we also take into account that these m summands are independent of each other, we can calculate the variance of our error as

$$\frac{\partial J}{\partial \text{out}} = \sum_{i=1}^{m} \sigma^2 = m \cdot \sigma^2.$$

Thus, the optimal choice for σ is

$$\sigma = \sqrt{\frac{1}{m}}.$$

Alternatively, we can measure the information contained in a neuron's output instead of information contained in errors that arrive at it. In this case, our goal is to ensure that the variance of all neurons is the same. The reasoning is very similar, and yields

$$\sigma = \sqrt{\frac{1}{n}},$$

where n is the number of inputs to the node. By combining these two intializations, we obtain the *Xavier intialization* (also known as Glorot initialization), where a neuron with n inputs and m outputs is initialized from the normal distribution with mean 0 and variance

$$\sigma = \sqrt{\frac{2}{n+m}}.$$

Even though the derivation makes many assumptions which are not generally true, the Xavier initialization works well in practice.

4.3.2 He initialization

The derivation of Xavier initialization is only sound for activations that approximate the identity near 0. The rectified linear unit does not have this property. He et al. [10] have subsequently adapted the Xavier initialization for the ReLU, Leaky ReLU, Parametric ReLU and similar activations.

Here, we will derive the initialization only for the ReLU activations. We will follow the same path as in the derivation of Xavier initialization, except for a few differences.

Let us, again, take the perspective of a single neuron. The variance of the error that arrives at us is

$$\frac{\partial J}{\partial \text{out}} = \sum_{i=1}^{m} w_i \cdot f'(\text{net}_i) \cdot \frac{\partial J}{\partial \text{out}_i}. \tag{4.17}$$

This time, we cannot assume that $f'(\text{net}_i) = 1$, since f is the rectified linear unit. We will have to proceed differently.

Let us take a look at each summand. Suppose that net_i and out_i are independent, and that the probability of net_i being positive is $\frac{1}{2}$. Then,

$$w_i \cdot f'(\text{net}_i) \cdot \frac{\partial J}{\partial \text{out}_i} = \begin{cases} w_i \cdot \frac{\partial J}{\partial \text{out}_i}, & \text{with probability } \frac{1}{2} \\ 0, & \text{otherwise} \end{cases} \tag{4.18}$$

Note that the first value is centered around 0 (its expected value is 0), which is exactly the value of the second option. Thus, the effect of the ReLU activation is essentially that with probability 0.5, it replaces the error with the mean. Since the definition of variance is

$$\text{Var}[X] = \text{E}\left[(X - \text{E}[X])^2\right], \tag{4.19}$$

the result is that in half of the cases, the value under expectation is 0, and in the other half of the cases, it is left unchanged. Thus, the variance is halved:

$$\text{Var}\left[w_i \cdot f'(\text{net}_i) \cdot \frac{\partial J}{\partial \text{out}_i}\right] = \frac{1}{2} \cdot \text{Var}\left[w_i \cdot \frac{\partial J}{\partial \text{out}_i}\right] = \frac{1}{2} \cdot m \cdot \sigma^2. \tag{4.20}$$

By setting it to 1, we obtain the *He initialization* (also known as Kaiming initialization) for ReLU:

$$\sigma = \sqrt{\frac{1}{m}} \cdot \sqrt{2}, \tag{4.21}$$

where m is the number of output connections from this neuron.

We could also consider the forward propagation case, where information is measured by the variance of the neuron outputs. In that case, we would obtain

$$\sigma = \sqrt{\frac{1}{n}} \cdot \sqrt{2}, \tag{4.22}$$

where n is the number of input connections to this neuron. Taking a compromise between the forward and backward propagation cases, we obtain

$$\sigma = \sqrt{\frac{2}{n+m}} \cdot \sqrt{2}. \tag{4.23}$$

Thus, we obtain He initialization simply by appropriately rescaling the weights initialized by Xavier initialization.

4.4 Normalization

Suppose we are given a probability distribution. We sample one value from it, and obtain 100 000. Is that a large value, or a small value? Generally, it is impossible to tell without having further information about the distribution, such as its mean and variance.

In a similar way, machine learning algorithm have to first learn the rough distribution of the input data. However, this can be done prior to employing the algorithm, in a process called *normalization*, which significantly increases the pace of learning.

In this process, we consider the attributes of the input data independently. We adjust the distribution of each attribute so that the values are relative and have well defined meaning. Commonly, the input data is first shifted and then rescaled, so that the resulting distribution has mean 0 and variance 1. This is done in two steps:

1. First, we calculate the mean of the attribute. We subtract this mean from all datapoints to obtain data centered around 0.

2. Second, we calculate the variance v of the attribute. We divide all datapoints by $\sqrt{v}$ to obtain data with variance 1. To ensure numerical stability, it is common to instead divide by $\sqrt{v + \epsilon}$ for some small $\epsilon > 0$.

In the resulting distribution, the sign of a value represents whether the (original) value is greater than or less than the average, and the magnitude tells us how much larger it is.

This can be taken one step further. Note that when the input features are correlated, having information about one attribute can change our expectations of other attributes. To illustrate, suppose we have two attributes: price and quality, with means 1 000 and 4.83. A high quality is indicative of high price and vice versa. Thus, if we already know that the quality is 8.74, a price of 1 000 would be very much below the expectation.

Thus, it is common practice to decorrelate the input data, in addition to ensuring mean 0 and variance 1. The process through which this is done is called *whitening*. We do not describe it here, as we do not utilize it in our thesis.

The idea behind normalization—bringing inputs into a more specific distribution—can also be used in the intermediate layers of a network, though it is not trivial to see how. This yielded various normalization layers, such as *batch normalization* [15], layer normalization [3] and local response normalization.

We take a more detailed look at batch normalization only.

4.4.1 Batch normalization

In a feedforward neural network, the network is consists of a sequence of layers. Each layer, except the input layer, takes its inputs from the previous layer. Thus, if we separated the network into the early stages and the later stages, the outputs from the early stages are essentially the input data for the later stages. Normalizing this data could improve learning.

However, it changes whenever any parameter of the network is adjusted, which happens after each optimization step. Computing its mean and variance over all train-

ing data after each step would be computationally infeasible: we would first have to compute the data by running the early stages of the network, and then calculate the means and variances on this large dataset.

There is also another problem. Gradient descent does not take into account that the data are being continually normalized. Thus, its estimates of how parameter adjustments affect the output would be generally incorrect, and would need to be adjusted.

Both of these issues can be resolved as follows. A new layer, called *batch normalization layer*, is introduced, whose goal is to normalize its inputs. Unlike other layers in a neural network, it takes advantage of the fact that we are operating on a batch of training data, instead of considering them one by one.

For simplicity, consider the case where there is a single input to the layer. In the general case, when the input is an arbitrary tensor, the transformation is carried out independently for each input dimension.

Let $x_1, \ldots, x_b$ be its values on the training cases in the batch, where b is the batch size. They are stacked along the batch dimension and together make up the *input tensor* to this layer:

$$\begin{pmatrix} x_1 \\ \vdots \\ x_b \end{pmatrix}. \tag{4.24}$$

We can obtain an estimate of the real mean of the input by calculating the mean on the current set of training cases:

$$\mu_{\mathcal{B}} = \frac{1}{b} \left(\sum_{i=1}^{b} x_i \right) \tag{4.25}$$

Similarly, we can estimate the real variance by calculating the variance on this batch:

$$\sigma_{\mathcal{B}}^2 = \frac{1}{b} \left(\sum_{i=1}^{b} (x_i - \mu_{\mathcal{B}})^2 \right) \tag{4.26}$$

This way, we can calculate an estimate of the mean and variance during the forward pass of backpropagation. The estimate is much faster to compute than the real values, which would require going over the entire training data. This resolves the first problem.

The output of the layer on the i-th training case is then

$$\frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}. \tag{4.27}$$

Note that all performed operations (calculating the mean, the variance, shifting the inputs and rescaling them) are differentiable with respect to the input tensor. Thus, the layer is compatible with backpropagation, and by propagating errors properly through all these operations, the second problem is resolved.

The normalized values are then rescaled and shifted. This is so that the introduction of the layer does not restrict the network's representational abilities. Thus, there are two learnable parameters per input dimension: β that shifts, and γ that scales. The output from the layer is then

$$\gamma \cdot \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta. \tag{4.28}$$

During evaluation, when we are no longer training the network, we use the real means and variances instead of the batch estimates. Hence, batch normalization degenerates to a simple rescale and shift operation.

The procedure is slightly altered when the preceding layer (the layer to be normalized) is a convolution layer. We want the normalization to normalize neurons in the same channel equally, so that they still detect the same feature, but at a different position. Hence, instead of estimating the mean and variance for each neuron individually, we estimate the mean and variance for each channel, and these are then used for all neurons in the channel. We also utilize per channel β and γ, instead of per neuron.

When we apply batch normalization to a layer, we essentially decouple some of the layer characteristics from the underlying process that calculates the layer's output. It causes the distribution of the layer to be more stable, since no matter what happens during the earlier stages of the network, the resulting value will have mean β and variance γ^2. This stability enables the next layer to learn faster, since it does not have to adapt as much to the changing distribution of the previous layer.

With batch normalization, one can usually set much larger learning rates during training, and the learning is significantly faster. Other than improving the learning speed, it also makes the network more robust to initialization, and has a small regularizing effect. The latter is due to the random nature of the estimates of batch mean and batch variance. Thus, batch normalization is commonly not used with other regularization techniques, or when it is used, they are reduced in strength.

There are however cases where batch normalization is not applicable. In online learning, the batch size is 1, making it impossible to estimate the means and variances. In recurrent neural networks, the distribution of the preceding layer changes after each step, and finding a single β and γ that would normalize the input dimension is generally impossible.

4.5 Regularization

The role of reularization in neural networks is to decrease overfitting. Since the networks are usually very large and have a large number of parameters, overfitting is

a serious problem. We present two regularization methods that are commonly used alongside neural networks.

4.5.1 Dropout

Neurons in neural networks often adapt together. In such adaptations, one neuron depends on a very specific set of other neurons. Such adaptations are very precise, thus they allow the network to detect very specific features. This allows them to remember the training data very well while not being able to generalize outside of it.

Dropout [11] [30] effectively solves this problem by randomly "dropping" neurons from the network during training. A neuron that is dropped has its output set to zero. The probability p of dropping a neuron is usually set to 0.5.

During testing, all neurons are active. This causes the input to each neuron to be roughly $\frac{1}{1-p}$ times larger than the corresponding input during training. To compensate for this, either the output of each neuron is scaled down during testing, or the outputs are scaled up during training. The latter approach is usually taken.

The introduction of dropout slows down learning, but can improve the final generalization error.

It was noted by Tompson et al. [32] that in the setting of convolutional networks, standard dropout, where each neuron is considered individually, does not decrease overfitting. They have formulated a better suited variant of dropout, called *spatial dropout*, where instead of considering neurons individually, entire channels of the input are being dropped.

4.5.2 L_2 regularization (weight decay)

L_2 regularization introduces an additional term to the objective function, which penalizes large weights. The intuition behind it is that large weights enable individual neurons to have large impact on other neurons. By introducing a penalty, we force the adaptation to take multiple neurons, as a single neuron is unable to do the job. The exact formula is based on the L_2 norm:

$$\lambda \cdot \|\vec{w}\|_2^2 = \sqrt{\sum_{i=1}^{|\vec{w}|} w_i^2}, \tag{4.29}$$

where λ is the regularization coefficient, and $\vec{w}$ is the vector of all the network's parameters.

When the optimization algorithm used is gradient descent, the effect of L_2 regularization on a weight w is that at each step, the weight is also adjusted by the value $-\lambda \cdot w$. Thus, if we consider the regularization term only, the update to weight w is

$$w := (1 - \lambda) \cdot w \tag{4.30}$$

Hence, in the context of neural networks where variants of gradient descent are the standard, the regularization method is also called *weight decay*.

Chapter 5

Proposed solutions

The goal of this thesis was to explore various ways of improving the pace at which deep networks learn. To this end, we devised and experimented with various optimization algorithms, activation functions, ... This chapter is dedicated to the most promising of our inventions. We start with the motivation.

Batch normalization can be decomposed into two sublayers: normalization of inputs, and rescaling and shifting. We contemplated whether both sublayers are necessary. Clearly, the second sublayer is necessary for the network to retain its representational ability. However, maybe the explicit normalization could be left out, and the layer would learn to normalize the inputs on its own.

Thus, we considered various layers composed of two building blocks: the *shifting layer* and the *scaling layer*, which we define in the next section.

5.1 Shifting layer

The shifting layer shifts the input tensor. The amount of shift for each dimension is a learnable parameter of the layer. In line with standard ANN terminology we will call this parameter *bias*.

Essentially, the layer is already present as a sublayer in many other neural network constructions. It is present in batch normalization, where it plays the role of the mean of the distribution, and in dense and convolution layers.

Explicitly decoupling the shifting layer from the aforementioned constructions allows us to place it at arbitrary positions in the network. We will most commonly place it after the activation layer. The reason is that before the activation, there is usually a dense or convolution layer, which already include a shifting layer.

Another improvement can come from sharing biases. By considering various sharing schemes, we obtain several simple variants of the layer:

- The *elementwise* variant, where each input dimension has its own bias. This is

the case for dense layers.

- The *channelwise* variant, where each input channel has a separate bias that is used by all input dimensions in that channel. This is the case for convolution layers.

- We also consider a variant that, to our knowledge, has not been utilized before, called the *layerwise* variant. All input dimensions share the same bias.

We will initialize the biases of all shifting layers to 0, as is common practice for biases in standard constructs. Such an initialization has the benefit that the shifting layer does not affect the initial network.

5.2 Scaling layer

The scaling layer rescales the input tensor, that is, each input dimension is multiplied by a certain factor. These factors are a learnable parameter of the layer. We will call this parameter *weight*, akin to weights in dense and convolution layers. Other than these two, the scaling layer is also a part of batch normalization and other normalization constructs.

We will mostly place the layer after the activation layer. The reason is that we will be utilizing ReLU, which is invariant to rescaling of its input in the following sense:

$$\mathrm{ReLU}(w \cdot x) = w \cdot \mathrm{ReLU}(x) \tag{5.1}$$

for nonnegative w. The initial weights will be positive, and we do not believe that their sign is likely to change during training, especially when the weights are shared. Such a change would require too large "coordination" between errors flowing from separate applications of the weight. By this line of thought, it makes no difference whether we place the layer before or after an activation. We choose the latter option.

In a similar fashion to shifting layer, we will consider three variants: the *elementwise*, the *channelwise*, and the *layerwise* variant. They have separate weights per input dimension, per channel, and per layer, respectively.

We initialize the weights to 1, so that the initial network is unaffected by the inclusion of the shifting layer.

5.3 Combinations

The shifting and scaling layers and their variants can be composed into complex combinations. The number of combinations is large. This stems from the fact that each of the layers can be either before or after an activation, each type of layer has three

		Scaling: None	Scaling: Elementwise	Scaling: Channelwise	Scaling: Layerwise
	None	✓	✓	✓	✓
Shifting	Elementwise	✓	✓	✗	✗
	Channelwise	✓	✗	✓	✗
	Layerwise	✓	✗	✗	✓

Table 5.1: The reduced set of combinations of shifting and scaling layers, and whether we will consider them or not.

variants, the various layer variants can be included multiple times, and then there is the order in which they are applied.

We thus took heuristic steps toward minimizing the number of combinations considered. Some of the steps and the intuitions behind them have already been explained in previous sections, such as that we will mostly place the layers after the activation. We now take further steps toward reducing the number of combinations.

Experimentally, we found out that when both the shifting and scaling layers are present, the order in which they are applied does not matter. Thus, we will place the shifting layers first and the scaling layers second by convention.

We will also set a limit of at most one shifting layer, and at most one scaling layer. This comes from the intuition than multiple layers of the same type, even if they are different variants, fill the same role in the network.

This reduces the number of combinations to 16, of which 10 we will consider. They are depicted in table 5.1.

Each activation, except for the final softmax, is turned into a sequence of layers. First, the activation layer is applied, then the output is shifted if there is a shifting layer, and finally it is rescaled if there is a scaling layer. Other than after the activation layers, we also put the shifting and scaling layers immediately after the input layer.

There is one special case. If we are using a channelwise variant, it may not always be applicable to the output of the previous layer, such as in the case when the preceding layer is a dense layer, not a convolution layer. In such a case, we apply the elementwise variant as a fallback.

Chapter 6

Experimental setup

In this chapter, we describe the common elements among experiments that were used to evaluate the proposed solutions. The details of individual experiments are provided in the next chapter.

Our goal was to evaluate how well the considered combinations perform compared to a plain ReLU activation with no enhancements, and compared to batch normalized networks, where each dense or linear layer is followed by batch normalization. Other than measuring performance, we were interested in why some combinations perform and others do not.

We test the solutions in image recognition tasks, due to their accesability and common use as benchmarks. The input is an image that belongs to one of finitely many classes, and the task is to determine which of the classes it is.

We try many different settings. Each setting is determined by the dataset and the network architecture. For each architecture, we try many of the aforementioned variants (plain network, our combinations, batch normalized).

For individual variants of networks, we adopt the abbreviations in table 6.1. We use these abbreviations when plotting the results from experiments.

6.1 Datasets

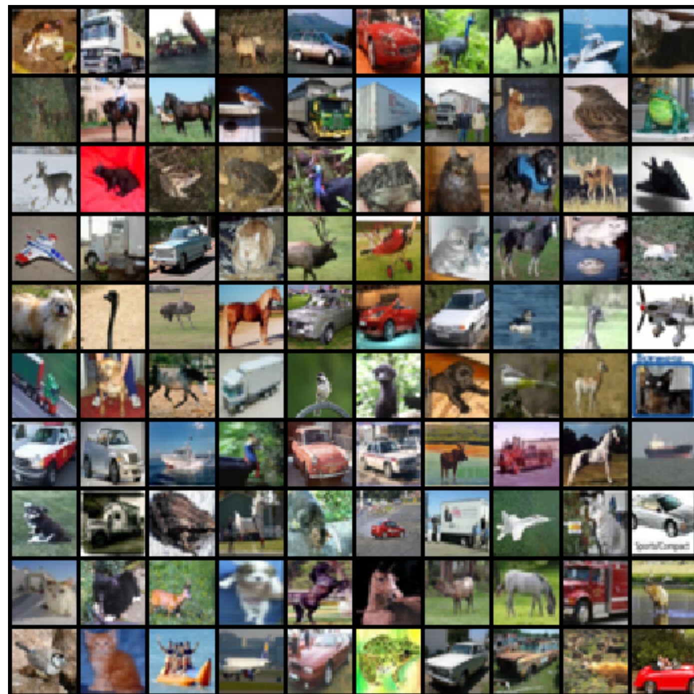
We were primarily working with the CIFAR10 and CIFAR100 datasets [17], which are commonly used as benchmarks for image classification models. Each of the datasets consists of 32×32 colored images and their labels. The data is split into 50 000 training cases and 10 000 test cases in both datasets.

In CIFAR10, each of these images depicts an object from 10 possible classes: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship* and *truck*. The classes are represented equally both in the training data and test data, thus there are 5 000 instances of each class in the training data and 1 000 instances of each in the test data.

variant	abbreviation
plain	plain
batch normalized	bn
elementwise shift	sh
elementwise scale	sc
elementwise shift and scale	sh_sc
channelwise shift	csh
channelwise scale	csc
channelwise shift and scale	csh_csc
layerwise shift	lsh
layerwise scale	lsc
layerwise shift and scale	lsh_lsc

Table 6.1: Abbreviations of all tested variants.

Figure 6.1: Some images from the CIFAR10 dataset.



input size	kernel size	num. kernels	padding	stride
$3 \times 32 \times 32$	3×3	96	1	1
$96 \times 32 \times 32$	3×3	96	1	1
$96 \times 32 \times 32$	3×3	96	1	2
$96 \times 16 \times 16$	(spatial dropout with $p = 0.5$)			
$96 \times 16 \times 16$	3×3	192	1	1
$192 \times 16 \times 16$	3×3	192	1	1
$192 \times 16 \times 16$	3×3	192	1	2
$192 \times 8 \times 8$	(spatial dropout with $p = 0.5$)			
$192 \times 8 \times 8$	3×3	192	0	1
$192 \times 6 \times 6$	1×1	192	0	1
$192 \times 6 \times 6$	1×1	10	0	1
$10 \times 6 \times 6$	global averaging over 6×6 spatial dimensions			
10	softmax			

Table 6.2: The architecture of the all convolutional network. Computation proceeds from top to bottom. Each convolution layer (including the last one) is followed by a ReLU activation layer.

CIFAR100 is essentially a more difficult version of CIFAR10, with 100 classes. Each class has 500 instances in the training data and 100 instances in the test data.

6.2 Architectures

In this section, we describe the various architectures used throughout our experiments.

6.2.1 All convolutional

The first architecture is a convolutional network by Springenberg et al. that introduces strided convolution [29]. The description of the architecture can be found in table 6.2. We will be using two versions of the architecture: one without dropout, and one with dropout. Note that we use *spatial* dropout, whereas Springenberg et al. employed standard dropout (since at that time, the former had not been conceived yet).

The weights in convolution layers are initialized with He initialization, since each of those layers is followed by a ReLU nonlinearity. The biases of the layers are set to 0.

input layer, size $3 \times 32 \times 32$	
flatten to size 3072	
1	dense 3072×3000
(dropout with $p = 0.5$)	
2	dense 3000×1000
3 ⋮ 12	dense 1000×1000 ⋮ dense 1000×1000
(dropout with $p = 0.5$)	
13	dense 1000×300
14 ⋮ 23	dense 300×300 ⋮ dense 300×300
24	dense 300×10
softmax	

Table 6.3: The multilayer perceptron. Computation proceeds from top to bottom. Each dense layer is followed by a ReLU activation, except for the last dense layer.

6.2.2 Multilayer perceptron

The second architecture is a simple multilayer perceptron. The input to the network is first flattened into a vector and then, 24 dense layers are applied. After each dense layer, there is an activation layer, and all activations except for the last one are ReLU. The last activation is softmax.

The details of the architecture are listed in table 6.3. We will be using both a version without dropout, and a version with dropout.

Weights in all dense layers are initialized with the He initialization, except for the last dense layer, where Xavier initialization is used instead. All biases are set to 0.

6.3 Methodology

In this section, we describe the employed methods and techniques that are common to most of our experiments, unless stated otherwise in the experiment's description.

Loss function We choose cross entropy as the loss function to evaluate the model's performance. No weight decay (L_2 regularization) was used, thus the objective function is simply the mean cross entropy over the batch of training cases.

Data preprocessing The training data are normalized (not whitened). The same transformation is applied to the test data. Note that this is not the same as normalizing the test data—we shift it and scale it using the mean and variance of training data, not test data.

One epoch of training At the beginning of each epoch, we randomly shuffle the training data. Then, at each step, we consider the next batch of training cases. We evaluate the model on this batch, calculate the mean cross entropy and compute the gradients with backpropagation. We update the parameters of the model using stochastic gradient descent with momentum 0.9. At the end of each step, we log the mean cross entropy and the accuracy of the model on the batch.

After each epoch, we calculate the validation error of the model by evaluating its performance on the test data that was not used during training.

Batch size We set the batch size to 32. We also experimented with larger batch sizes, concretely 128 and 512. With batch size of 128, training was faster, but at the cost of universally higher validation error. Batch size of 512 caused an even higher validation error.

Learning rate In each experiment, we first determine an appropriate learning rate for the plain network with no enhancements. A learning rate where the network actually learns can be found by trying learning rates of the form 10^{-i}. We call this learning rate the *anchor*. Afterwards, we adjust the learning rate in the neighbourhood of the anchor in smaller amounts, and observe whether this leads to faster learning.

If the only difference in the network architecture is the presence of dropout, we use the same anchor as in the version without dropout. However, we still search in the small neighbourhood of the anchor.

The same learning rate that is used for the plain networks is then used for all our combinations. Thus, if a combination consistently performs better than the plain network, we can expect it to perform even better if the learning rate is adjusted specifically for that combination.

For batch normalization, we use a slightly higher anchor than what was found for plain networks. This is because batch normalization allows the network to learn effectively even with higher learning rates.

Initialization Once all hyperparameters are set, we train the network variant. Since the networks are initialized randomly, to obtain more representative results, we train it multiple times to obtain multiple *runs*. To ensure that these runs are comparable

between different variants, the sequence of seeds used to initialize the random number generator is the same for each variant.

6.4 Implementation

We estimated that we would have to try a large number of experiments before arriving at an interesting solution, and we considered implementing each experiment individually infeasible. We thus created a library for fast prototyping, where large portions of the code between different experiments could be reused.

The library is programmed in Python 3.5.2, utilizing PyTorch 0.4 [24] for computational networks and optimization. For plotting various graphs which were later exported and included in the thesis, we used Matplotlib [14]. The source code of the library can be found at: `https://github.com/buj/bakalarka_public`

Chapter 7

Individual experiments

7.1 Initial experiment

With this experiment, our goal was to determine which of the combinations are worth further investigation, and which are not. We compared all of them with the batch normalized variant and the plain variant on the CIFAR10 dataset. We chose the all convolutional network (ch. 6.2.1) without dropout as the architecture.

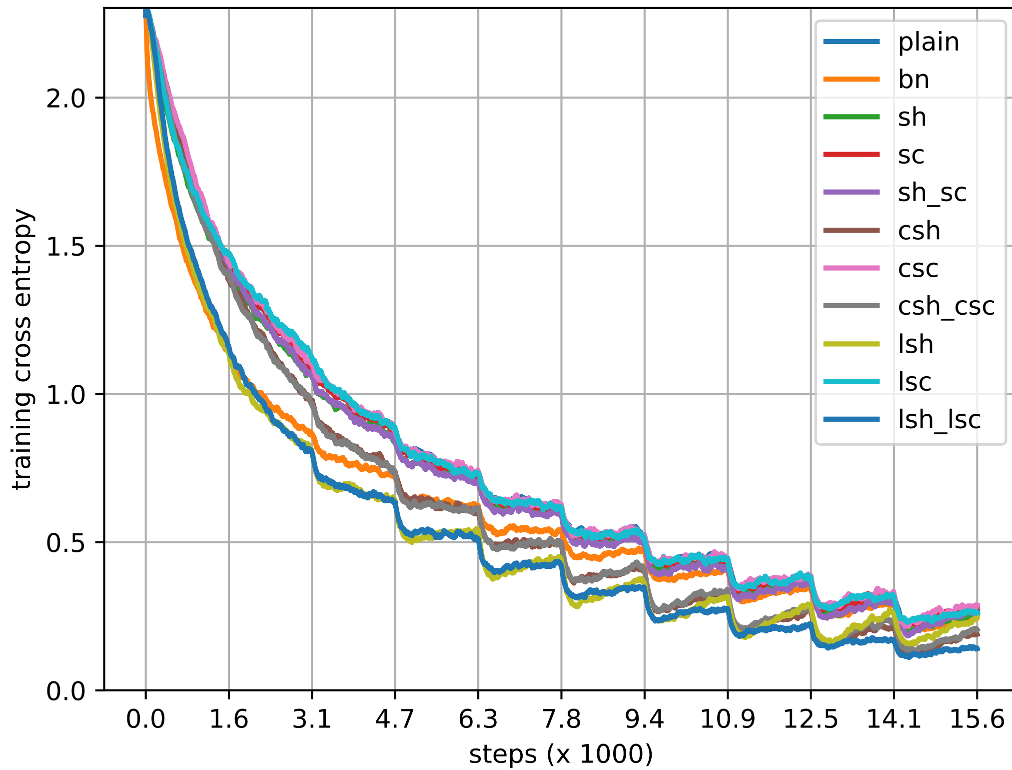
After determining the anchor, the learning rate was selected from among 0.005, 0.01 and 0.02. With each of them, we trained the plain variant for 10 epochs. The learning rate is picked based on the results.

For the batch normalized variant, we try the learning rates from among 0.02, 0.04 and 0.08. With each, we train the model for 10 epochs to see which learning rate is the best.

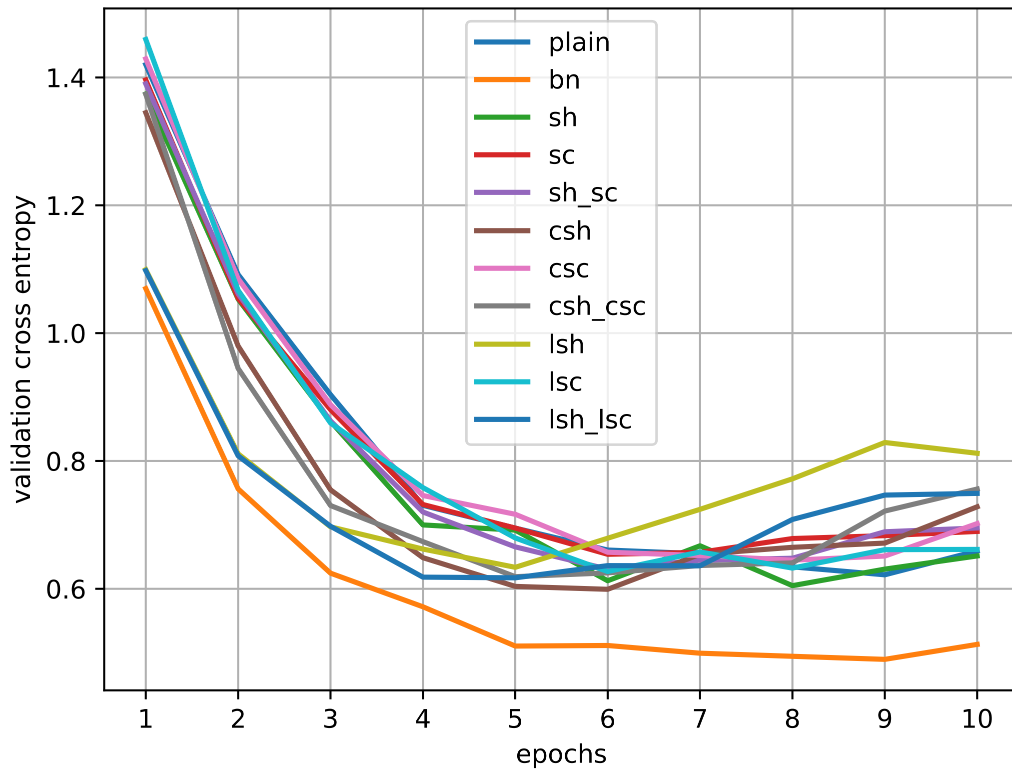
Once the learning rates are determined, we train all the variants in 4 training runs.

Results and discussion The results of the experiment can be found in the plot 7.1. For clarity, we also include plots where similarly performing variants are grouped together: 7.2, 7.3 and 7.4. In each plot, we include both the plain variant and the batch normalized variant for comparison.

The scaling-only variants all performed similarly to the plain variant. The variants which included the shifting layer performed better, and surprisingly, the less parameters a shifting layer had, the better. The elementwise shifting performed the worst from among them, then the channelwise shifting, and layerwise shifting performed the best. We came up with a possible explanation, which is explained and tested in the next subsection.

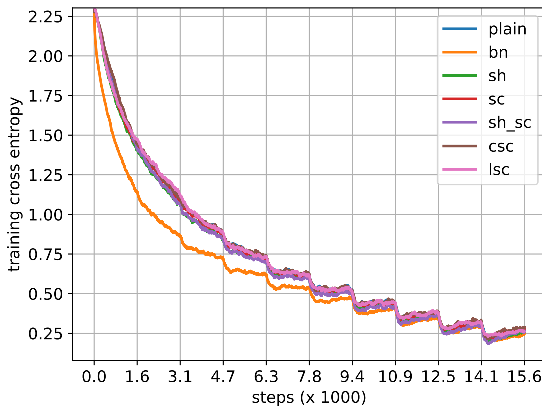


(a) Exponential moving average of training errors after each step, with coefficient 0.99.

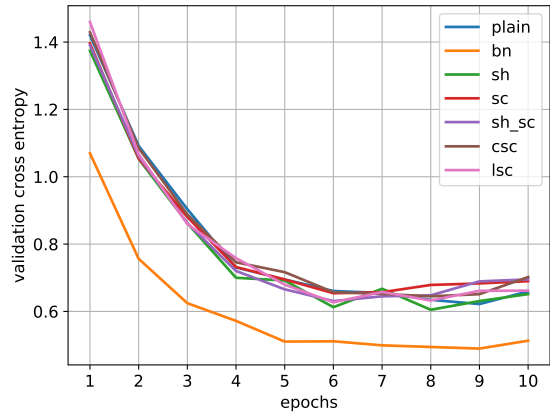


(b) Validation error after each epoch.

Figure 7.1: Initial experiment (CIFAR10, all convolutional network without dropout). All variants, see table 6.1 for abbreviations.

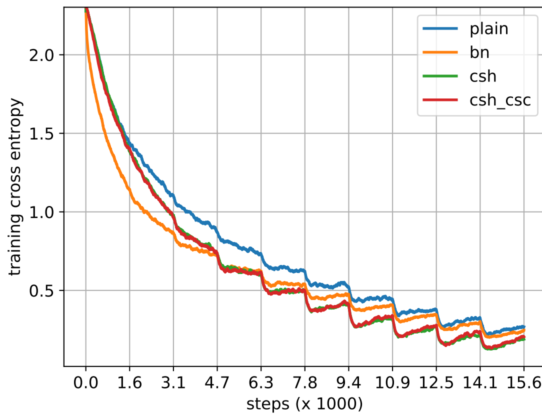


(a) Exponential moving average of training errors after each step, with coefficient 0.99.

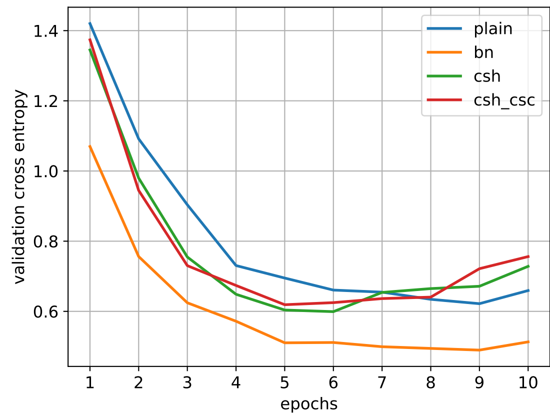


(b) Validation error after each epoch.

Figure 7.2: Initial experiment (CIFAR10, all convolutional network without dropout). Variants that did not perform significantly better than the plain variant: the element-wise variants and the scaling-only variants.

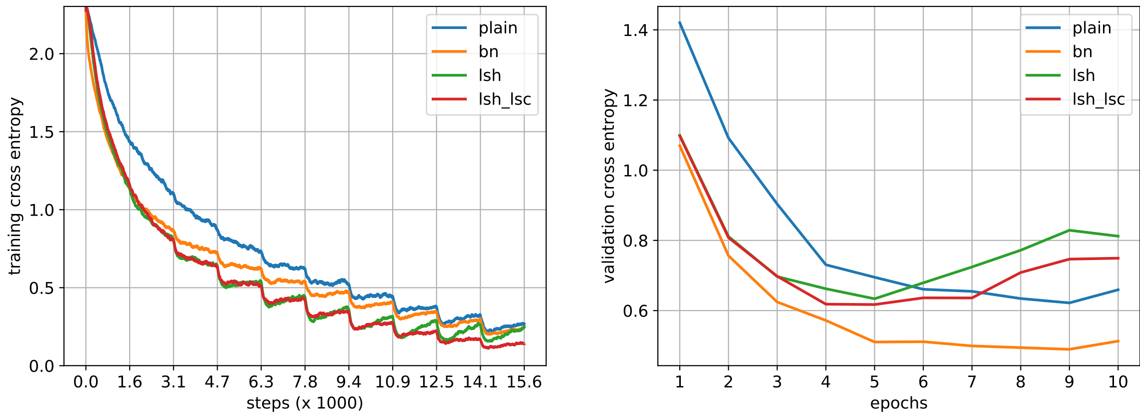


(a) Exponential moving average of training errors after each step, with coefficient 0.99.



(b) Validation error after each epoch.

Figure 7.3: Initial experiment (CIFAR10, all convolutional network without dropout). Variants that performed better than the plain variant: channelwise shift and channel-wise shift + scale.



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.4: Initial experiment (CIFAR10, all convolutional network without dropout). The best performing variants: layerwise shift and layerwise shift + scale.

7.1.1 Hypothesis 1

Suppose that all instances of the biases would often receive similar errors. Since they are all shared in the layerwise variants, they would be accumulated, and each instance would then adapt based on this accumulated error. Thus, they would change much faster than if the error was local for each instance, and not global. This would explain why the channelwise variant did not perform as well, and why the elementwise variant performed the worst.

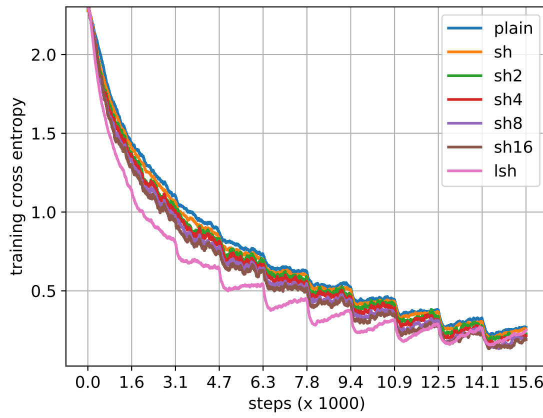
If this explanation were true, we would obtain similar performance with channelwise and elementwise shifts by increasing the learning speed of their biases. This was our next experiment.

We tried the elementwise shift variant, with the learning rate of its biases multiplied by 2, 4, 8 and 16. We give these variants the abbreviations *sh2*, *sh4*, *sh8* and *sh16*, respectively.

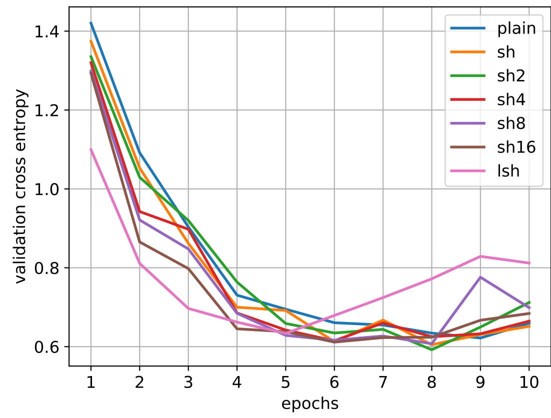
For the channelwise shift, we tried multiplying the learning rate of its biases by 2, 4 and 8. We did not try 16, as we have found that the network was already unable to learn when the multiplier was 8. The abbreviations for the variants are *csh2*, *csh4* and *csh8*, respectively.

The global learning rate is set to 0.01. We trained each variant a single time.

Results and discussion The results can be found in the plot 7.5 and 7.6. We observe that by increasing the learning rates for biases in the elementwise shifting layers, we obtained better results. We did not observe this effect for the channelwise shifting layers. Further investigation needs to be done.

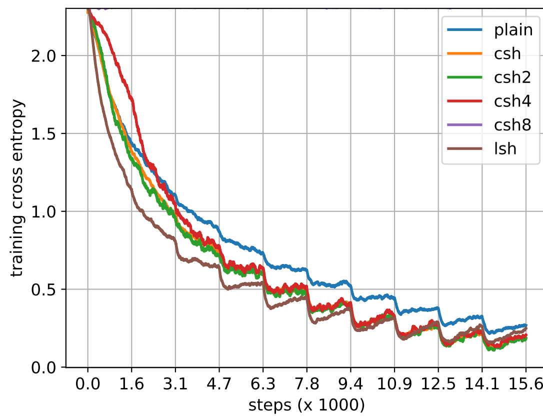


(a) Exponential moving average of training errors after each step, with coefficient 0.99.

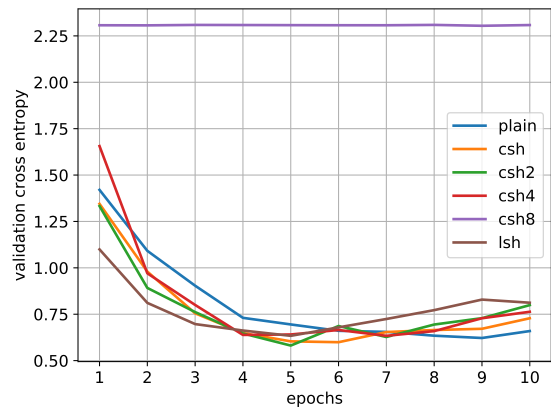


(b) Validation error after each epoch.

Figure 7.5: Data from hypothesis 1, the elementwise shifting layer.

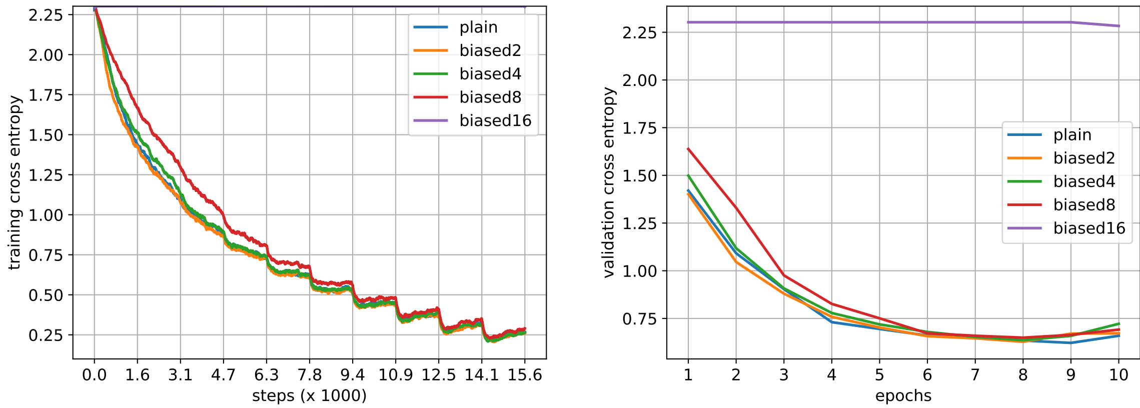


(a) Exponential moving average of training errors after each step, with coefficient 0.99.



(b) Validation error after each epoch.

Figure 7.6: Data from hypothesis 1, the channelwise shifting layer.



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.7: Experiment with all convolutional network without dropout, variants with no enhancements, but where biases in dense and convolution layers learn many times faster than weights.

7.1.2 Hypothesis 2

We were also interested whether the shifting layers were really that important. Maybe it would be enough to increase the learning rate for all biases in the dense and convolutional layers, and we would observe similar effects.

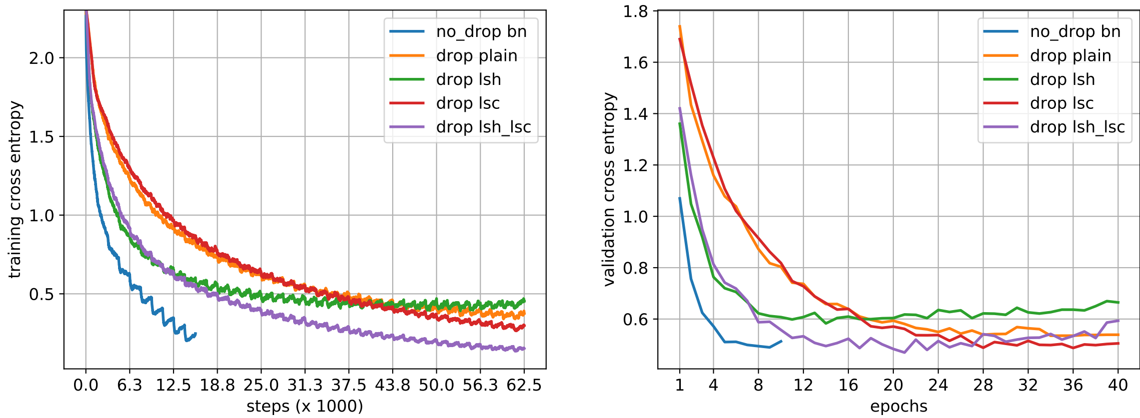
Thus in this experiment, we tested a network with no enhancements, but we multiplied the learning rates for biases in all convolutional and dense layers by a certain coefficient. We considered multiple variants based on this coefficient, the values were 2, 4, 8 and 16. We will call these variants *biased2*, *biased4*, *biased8* and *biased16*, respectively.

We set the global learning rate to 0.01 (the same as for previously mentioned variants) and trained for 10 epochs. We trained for 4 runs, except for the biased16 variant, which was trained once only, as it was unable to learn.

Results and discussion The results can be found in the plot 7.7. None of the variants performed better than the plain variant. Thus, when it comes to learning with gradient descent, the shifting layers have different properties than biases in dense and convolution layers.

7.2 CIFAR10, convolutional with dropout

In this experiment, we were interested in how the best performing variants would do with dropout. Fast learning would be useless if it would lead to overfitting only. We tested the all layerwise variants, including the scaling-only variant, and the plain



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.8: Experiment with all convolutional network with dropout. For comparison, we include the batch normalized variant without dropout.

variant. Since batch normalization has its own regularization effect, we do not evaluate it.

We set the learning rate to 0.01, the same as for the version of the network without dropout. There are 4 runs, and in each run, the network is trained for 40 epochs.

Results and discussion The results can be found in the plot 7.8. We also include the batch normalized variant (without dropout) for comparison. The layerwise shift + scale achieves comparable validation error, although it takes longer to achieve this error, due to dropout.

This experiment demonstrates that it is possible to “tame” the learning speed-up, so that it does not only lead to overfitting.

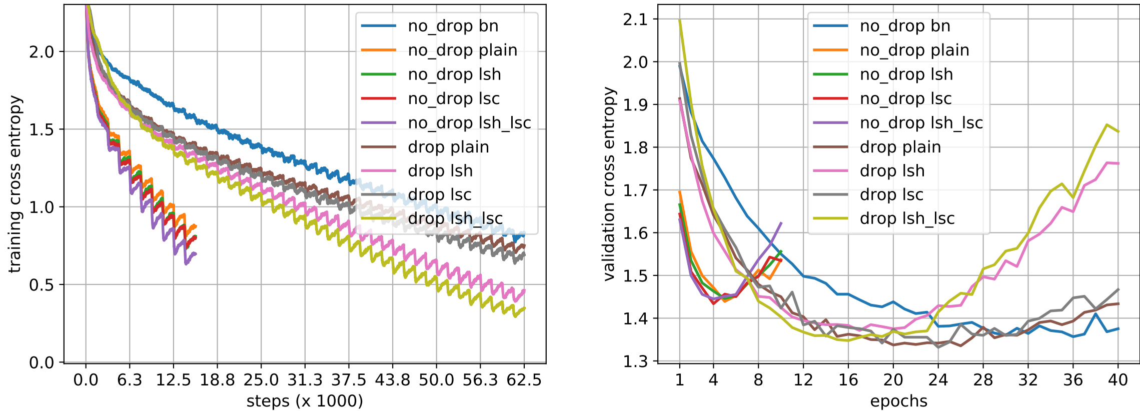
We see that the variants with the layerwise shifting layer are learning significantly faster during the initial stages. However, without the layerwise scaling layer, the learning slows down, and the plain variant eventually gets ahead of the shifting-only variant. The layerwise scaling layer thus somehow mitigates this effect, since the variant that includes both of them performs the best.

The scaling-only variant does not initially lead to faster learning, but in the long term, it gets ahead of the plain variant.

7.3 Experiments with multilayer perceptron

With the multilayer perceptron (ch. 6.2.2), we ran experiments only on the CIFAR10 dataset, as we believed it would not learn well on the more difficult CIFAR100 dataset.

First, we tried the version without dropout. We experimented with the plain,



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.9: Experiment with the multilayer perceptron, both with and without dropout.

batch normalized, and the layerwise variants. The learning rate is selected from among 0.0005, 0.001, 0.002, and 0.004. We train for 10 epochs, and the best model from each variant is trained in 8 runs.

For the batch normalized variant, the learning rate is selected from among 0.001, 0.002, 0.004. We train for 40 epochs, and with the best learning rate, we train 8 times.

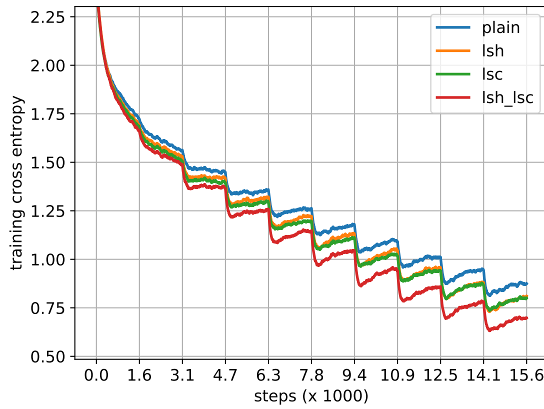
For the version with dropout, the learning rate is selected from among 0.001, 0.002, 0.004. With each of these values, we train for 20 epochs to determine the best learning rate. With the best one, we train for further 20 epochs for a total of 40 epochs. The batch normalized variant is not considered. There are 8 training runs.

Results and discussion The results can be found in the plot 7.9. The results on the architecture without dropout are in plot 7.10, except for the batch normalized version, which behaved much differently. The results with dropout are in plot 7.11, and we include the batch normalized variant here.

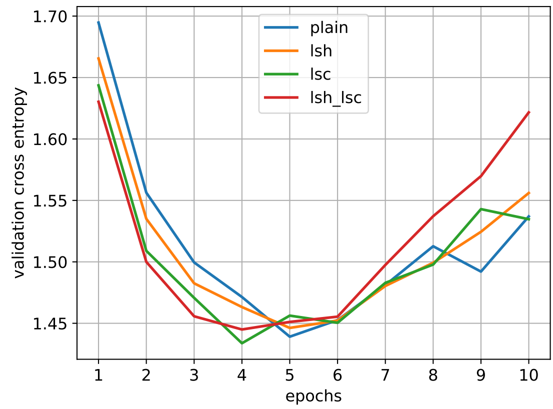
Without dropout, we can see that our variants, mainly those that include a shifting layer, learn faster than the plain variant, both in terms of training error and validation error. With dropout, the variants with shifting layer descend quickly in training error, but overfit much more than the scaling-only variant and the plain variant, even in the face of dropout. None of the variants performed better in terms of validation error. Thus, stronger regularization needs to be employed with our variants.

7.4 CIFAR100

We tried another dataset, to see whether the observed speed-up in learning could be translated to a more difficult task. On this dataset, we trained only with the convolu-

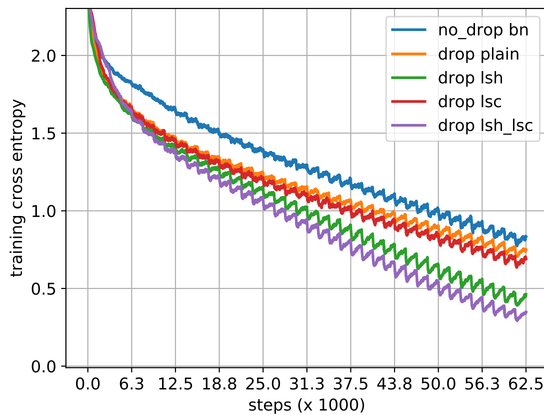


(a) Exponential moving average of training errors after each step, with coefficient 0.99.

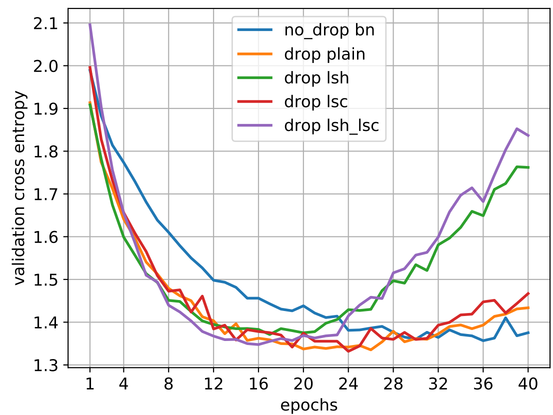


(b) Validation error after each epoch.

Figure 7.10: Experiment with the multilayer perceptron, no dropout.

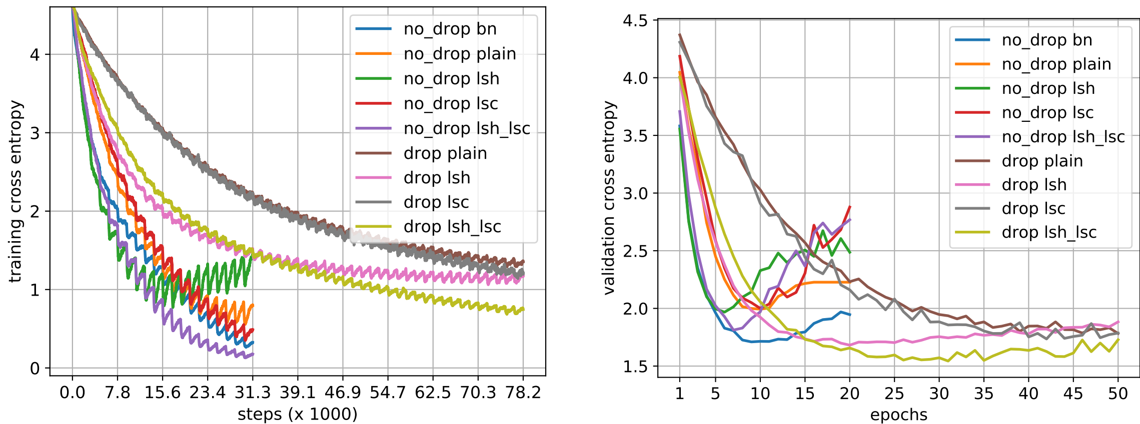


(a) Exponential moving average of training errors after each step, with coefficient 0.99.



(b) Validation error after each epoch.

Figure 7.11: Experiment with the multilayer perceptron with dropout.



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.12: Experiment with the CIFAR100 dataset. The architecture is all convolutional, both version with and without dropout are included.

tional architecture (ch. 6.2.1). Without dropout, we test the plain, batch normalized, and all three layerwise variants. With dropout, we omit the batch normalized variant.

The learning rate is selected from among 0.005, 0.01, 0.02. We train for 15 epochs, and the with the best learning rate, we do at least 2 training runs for each variant.

For the batch normalized variant, we selected the learning rate from among 0.01, 0.02, 0.04 and 0.08. We train for 20 epochs, and there are 4 training runs.

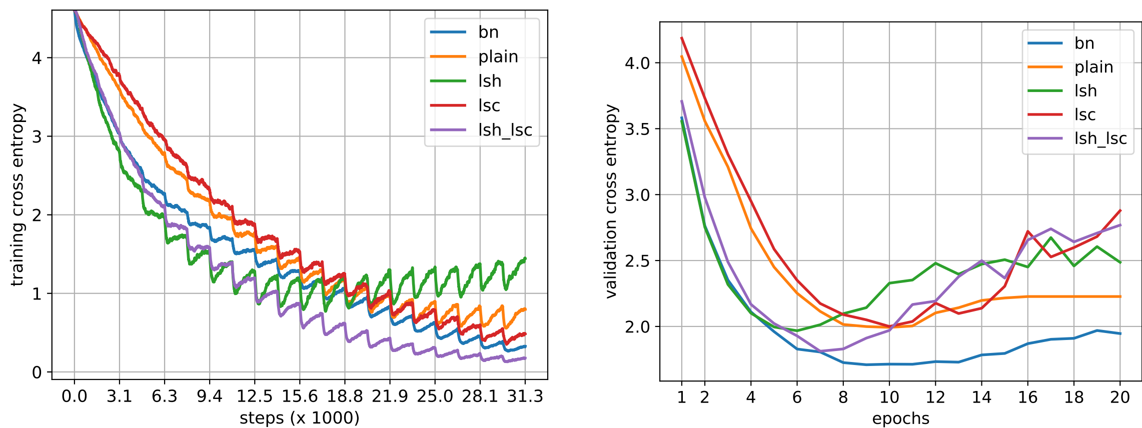
For the version with dropout, the learning rate is chosen from among 0.005, 0.01 and 0.02. We train for 50 epochs. There is at least a single training run.

Results The results for all the variants can be found in the plot 7.12. Variants without dropout are plotted in the plot 7.13. Variants with dropout and batch normalization (without dropout) are plotted in the figure 7.14.

We observe effects similar to the experiment with CIFAR10, all convolutional network with dropout. The layerwise shifting layer causes the network to learn quickly in the initial stages, but it eventually stagnates. In fact, the training error starts to increase, suggesting that the learning rate is too high in these stages of learning.

The layerwise scaling-only initially performs worse than the plain variant, but performs better in the long term. The combination of these two layers learns quickly both in the short term, and in the long term.

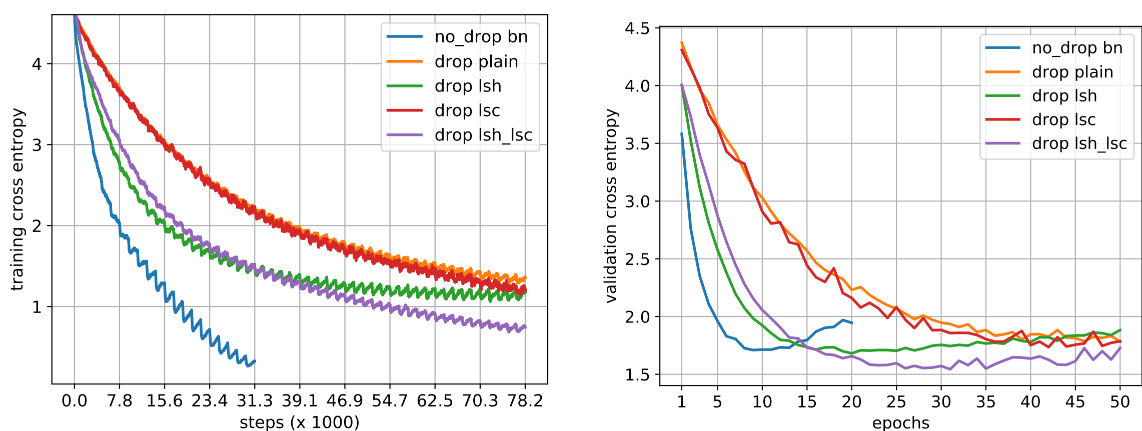
With dropout, we were in able to achieve lower validation error than the batch normalized variant. This again shows that with appropriate regularization, the method can achieve good results in shorter amount of time.



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.13: Experiment with the CIFAR100 dataset. The architecture is all convolutional, no dropout.



(a) Exponential moving average of training errors after each step, with coefficient 0.99.

(b) Validation error after each epoch.

Figure 7.14: Experiment with the CIFAR100 dataset. The architecture is all convolutional with dropout.

7.5 Other experiments

We tested variants utilizing both batch normalization and our innovations. However, we found that the inclusion of the scaling and shifting layers did not affect the training error, and actually led to worse performance as measured by the validation error.

We also experimented with a recurrent neural network architecture, similar to the one described by Le et al. [18] We considered the same task: we trained the recurrent neural network to classify handwritten digits from the MNIST [1] dataset, where the input is fed in pixel by pixel. However, in our experiments, none of the combinations from the table 5.1 yielded better performance than the original network.

We believe this is a difficulty similar to that of batch norm—the distribution of the state changes after each time step, and it is impossible to find one weight and one bias that would fit in all time steps. Further research would need to be done to confirm or disconfirm this hypothesis.

Conclusion

The goal of our thesis was to explore various ways of improving learning in deep neural networks. To this end, we created a library for fast prototyping of models and various other utilities which increased the pace at which we could explore new ideas.

We introduced the *shifting* and *scaling* layers. For each of these type of layers, we introduced three variants based on parameter sharing. These variants could then be put together into various combinations. We were interested whether their introduction into a network would improve the network's learning pace, and generally what effect do these layers have on the learning dynamics.

We tested these combinations on image recognition tasks. The combinations were tested in various settings to evaluate whether their effect is robust enough, or whether it is specific to a certain setting. Each setting was determined by the dataset and the network architecture.

In our initial experiment, we tested all combinations to see which of them were worth further experimenting. We found out that the *layerwise shift* and *layerwise shift and scale* performed well, even though they added to the network only a small number of parameters per layer. Those were then experimented with further. We also experimented with the *layerwise scale*.

We found that in all feedforward architectures, the enhancement significantly improved the pace at which the network learns, compared to the network with no enhancement. However, it also showed a tendency to overfit the data. Thus, stronger regularization has to be employed.

We were unable to match the batch normalized variants in terms of learning pace, as measured on the validation set. Also, combining the layers with batch normalization yielded to the same training errors, and the validation errors were actually worse. However, our method can be employed in settings in which batch normalization cannot be used, such as online learning, where the batch size is constrained to 1.

We were unable to make any of the combinations work well in the recurrent neural network. We believe this is due to the implicit weight sharing in recurrent networks, which constrains the transformation to be the same for each time step.

We find the layerwise shifting and scaling layers to be promising, but further work needs to be done to understand them. We performed experiments only on relatively

small networks, and it remains to be seen how the effects of these layers can translate to larger networks.

Bibliography

[1] The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2018-05-16.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[5] Nadav Cohen, Or Sharir, and Amnon Shashua. On the expressive power of deep learning: A tensor analysis. In *Conference on Learning Theory*, pages 698–728, 2016.

[6] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[8] Pavel Golik, Patrick Doetsch, and Hermann Ney. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*, volume 13, pages 1756–1760, 2013.

[9] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[11] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

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

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[14] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[18] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[21] Marvin Minsky and Seymour Papert. Perceptrons: AnIntroduction to computational geometry, 1969.

[22] Tom M. Mitchell. The need for biases in learning generalizations. Technical report, 1980.

[23] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

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

[25] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[26] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

[27] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer, 2010.

[28] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[29] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[31] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[32] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.

[33] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Appendix A: source code of library

The source code of the library for rapis prototyping of experiments can be found on the attached CD. Alternatively, it can be found on this webpage: `https://github.com/buj/bakalarka_public`