university of groningen / faculty of science and engineering / bernoulli institute

MASTER'S THESIS

Exploring Improvements for Gradient Descent Optimization Algorithms in Deep Learning

Author:
Richard ELDERMAN

Supervisors:
Dr. Marco WIERING
Prof. Dr. Lambert SCHOMAKER

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Artificial Intelligence

in the

Bernoulli Institute
Faculty of Science and Engineering
UNIVERSITY OF GRONINGEN

March 7, 2019

"There's a way to do it better - find it."

Thomas A. Edison

UNIVERSITY OF GRONINGEN

Abstract

Faculty of Science and Engineering
Bernoulli Institute

Master of Science in Artificial Intelligence

Exploring Improvements for Gradient Descent Optimization Algorithms in Deep Learning

by Richard ELDERMAN

The use of the right optimization algorithm for gradient descent in a deep learning system can have a big influence on the learning performance of the classifier. Currently, there are two families of optimization algorithms commonly used: pure stochastic gradient descent (SGD) with some kind of momentum, and algorithms with adaptive learning rates from which Adam is the most commonly known. In this thesis, it is tried to investigate possible improvements on these state of the art optimization algorithms for gradient descent used in deep learning. Five different new methods are invented that could improve the algorithms. One of them tries to incorporate some braking system in the SGD with momentum algorithm, while the other four use a different gradient history collection (GHC) method for the adaptive learning rate algorithms. These new techniques are compared to the state-of-the-art algorithms in 16 different experiments. One experiment is dedicated to a pure optimization setting on a simple convex optimization function, while all other experiments use either the logistic regression method, a standard multi-layer perceptron, or a convolutional neural network as a classifier. The results of the experiments show in general that the adaptation of the SGD with momentum algorithm turns out to perform worse than the standard SGD with momentum algorithm. Two of the four alternative methods for GHC in adaptive learning rate algorithms also perform worse than the standard version, but the other two methods turn out to perform better. These two alternative GHC methods can be proposed as potential improvements of the state of the art algorithms such as Adam.

Table of contents

Chapter 1

Introduction

Deep learning is the most popular machine learning technique nowadays. It is a powerful tool, loosely based on the internal workings of the brain, and can be used for a variety of applications in multiple research fields. Often it is used as a classifier, but it is also suited for a regression task. Examples of research fields in which deep learning is used are speech recognition (Hinton et al., 2012), object recognition (Socher et al., 2012), natural language processing (Collobert and Weston, 2008), and reinforcement learning (Sutton and Barto, 1998; Wiering and Van Otterlo, 2012; Wolfshaar, Wiering, and Schomaker, 2018).
Deep learning is nowadays also used as the artificial intelligence part of custom applications and products, such as face recognition (Schroff, Kalenichenko, and Philbin, 2015), personal assistants like Apple's Siri and Google Assistant (Jones, 2014), and autonomous driving systems (C. Chen et al., 2015). These products and applications can use deep learning because of one or more of the following reasons: (1) a big company with a lot of money produces it, (2) the classification scope is (highly) limited, and (3) the situations in operation time can be simulated during training time. If neither would be the case, the deep learning system can hardly be trained well enough before operation time, and hence deep learning could not be used.
The underlying problem is that, for training a neural network such that it becomes robust and stable enough for the custom market, it needs a big amount of training examples and learning iterations. Moreover, for real-world applications the deep learning systems are often very big, so it can take days or even months before such a system is trained. In recent years the hardware has of course become much faster and expandable than before, especially since GPUs can be used.
The final performance of the deep learning system is highly dependent on three aspects that all must be chosen at forehand: the type and size of the network used in the system (also called network architecture), how the internal parameters of the system are initialized, and finally the method to train the network which is highly dependent on the chosen optimization algorithm. Considering this final aspect, not only the choice of algorithm is important, but the optimal values of the hyperparameters of the algorithm are different for each application in which neural networks are used. So it may need several tries with different algorithms and parameter settings before a trained system has a performance high enough to use it in custom applications, which often still takes a lot of time.
A deep learning system is typically a supervised learning system, which means that it gets pairs of input and corresponding output. The difference between the output of the system and the desired output, calculated using some error function, can then be used (via a method called backpropagation) to tweak the internal parameters (also called weights) in the system such that the difference would become smaller. Since the network needs to be able to correctly classify as much data as possible, the weights in the network cannot be changed such that the current training examples

will be classified optimally. This is the case because the current training examples will hardly ever reflect the exact (relevant) properties of all the data: it is a stochastic representation of the full data set. Therefore the weights are changed only a small bit into the direction of the weights that would classify the current training examples better. Doing small steps has the disadvantage that it will take a long time until convergence has been reached, i.e. no further improvements can be made (which by the way does not always mean that all data will be classified correctly).

1.1 Current state of optimization algorithms for gradient descent

Optimization algorithms are designed to try to take as much advantage from the information about the weight changes coming from the current examples as possible while preserving the generalization of the network over the entire scope of the data set. They get as input a list of gradients, one for each weight in the network, which are commonly calculated using backpropagation, and try to change the weights accordingly. The most basic and also oldest algorithm that does this is called standard stochastic gradient descent, or SGD (Goodfellow, Bengio, and Courville, 2016). This algorithm simply takes a proportion of the gradient using a parameter called the learning rate. However, this method is rather slow, since it can take small steps and ultimately after a large amount of time reach the minimum error, or take big steps and probably overshoots the minimum (i.e. the big steps make it ascent instead of descent in the error landscape). The addition of (some kind of) momentum to this SGD algorithm (Nesterov, 1983; Polyak, 1964), based on the analogy of a ball rolling down a hill, speeds up this process significantly (Qian, 1999; Sutskever et al., 2013). However, in recent years a new family of algorithms has been developed that can speed up the learning process even more.

1.1.1 Adaptive learning rate methods

The key idea in adaptive learning rate algorithms is to give every weight in the network its own learning rate. In this way, a single weight can take a big step into a direction if needed without dragging the other weights along and making them worse. The first commonly known algorithm with this technique was proposed back in 2011 (Duchi, Hazan, and Singer, 2011). This publication led to a small flood of derivatives and improved versions (Zeiler, 2012; Kingma and Ba, 2014; Dozat, 2016; Reddi, Kale, and Kumar, 2018), which added a more convenient way of determining the importance of an update and incorporated (Nesterov) momentum into the algorithm. The algorithms in this family, and especially the more sophisticated ones (Kingma and Ba, 2014; Dozat, 2016), have proven to get superior results with respect to the standard SGD algorithms, and have therefore become the standard first choice as optimization algorithm in a deep learning system.

1.2 Motivation of the study and research question

The fact that since the introduction of adaptive learning rates optimization algorithms about eight years ago multiple derivations have been proposed (up until one year ago) indicates that the deep learning community is still searching and exploring for the best mechanism for this algorithm. It is thought that the research conducted in this thesis could contribute to the state of this research field, by searching for possible improvements of the state of the art algorithms. Since the deep learning field is very popular, and hence makes a quite rapid progress, the research conducted in this thesis could have some impact and influence on the research field in the case that it includes at least one idea to improve the current state-of-the-art algorithms. The main research question of this thesis can be stated as:

How can the state-of-the-art optimization algorithms for gradient descent in neural networks be adapted, such that the new variant outperforms its standard version in convergence rate and -speed?

It is tried to answer this question by coming up with possible variants of state of the art algorithms, and testing the performance and functionality of these new versions in either a multi-layer perceptron, convolutional neural network, or a pure optimization setting in a total of 16 different experiments. The results of all the conducted experiments will be evaluated, and it will be determined if the novel algorithms outperform the state-of-the-art algorithms. For this purpose, the performance of the algorithm has to meet some requirements. It should:

1. need (significantly) less training epochs than every other algorithm to reach a certain training cost.
2. reach a test cost at least as low as the average of all other algorithms.
3. not need (significantly) more time for a single update than every other algorithm.

The first requirement is the main criterion for this research, since it shows the direct optimization from the supervised learning of the training data. If this would be bad, the algorithm could never reach a low test cost. Criterion 2 is included since the algorithm also needs to preserve the generalization of the entire data set, and thus not overfit on the training data. Criterion 3 is included since an algorithm that requires a much smaller number of updates than other algorithms seems good, but when a single update lasts much longer than for the other algorithms, the algorithm is not faster after all.

1.3 Thesis outline

In the next chapter the theory behind all used state of the art concepts in the field of neural networks are covered, including its biological sources of inspiration, the basic principles of an artificial neuron, and a standard multi-layer perceptron. Moreover, some regularization methods will be covered, as well as the convolutional neural network. However, the biggest part of the next chapter consists of an extensive overview of all state of the art optimization algorithms.
In the third chapter, the new ideas for improvements of the optimization algorithms are described and explained, and some general aspects of the conducted experiments are discussed.
Hereafter are the various experiments described in 5 chapters, being: experiments on the Rosenbrock function, on the XOR problem, and on the MNIST (LeCun, 1998) and CIFAR10 (Krizhevsky and Hinton, 2009) data sets. The experiments on these two data sets are either on a system with logistic regression, on a multi-layer perceptron, or on a convolutional neural network.

After these chapters, a chapter is assigned to the general discussions of the results of all experiments. The final chapter consists of the general conclusions, it is tried to answer the research question, and a list of future work ideas building on the outcomes of this thesis is included.

Chapter 2

Theory

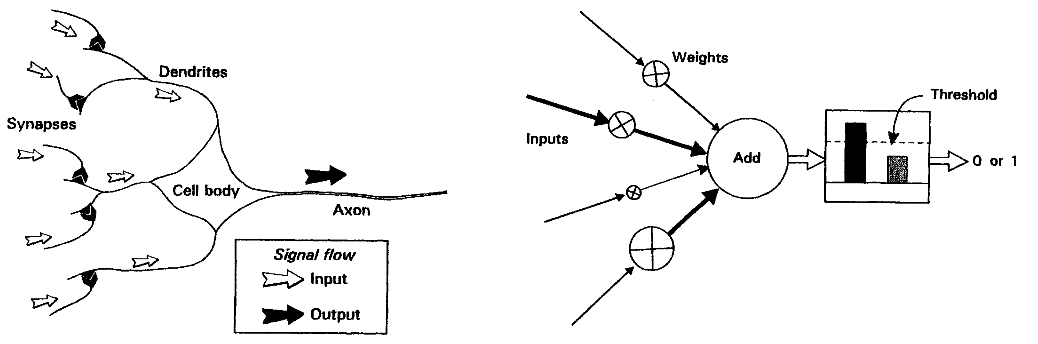
The field of deep learning is quite a broad field, with many aspects and several different kinds of neural networks. In this chapter, all topics in the field of deep learning that are used in the experiments in this thesis are explained and discussed. In the first section, the basic workings of an artificial neuron and an artificial neural network are explained, as well as some regularization methods. In the second section, all state of the art optimization algorithms considered in this thesis are extensively described. In the last section, a special kind of neural network is discussed, namely the convolutional neural network.

2.1 Introduction to neural networks

The basic principles of neural networks are already quite complicated. As with all classifiers, there is a given input describing the data, and the system has to give a classification using this input. Moreover, the system must be able to learn from the past, and since it is a supervised learning system, it can do so by changing its inner workings upon the errors it made in the previous classifications. In this section, it is described how a neural network comes up with a classification based on some input, and how it can learn to improve them. But first, the building blocks of an artificial neural network are discussed, starting with a brief comparison between these artificial neurons and their biological counterpart and source of inspiration.

2.1.1 Biological influence

The brains of every existing living creature can be seen as a very sophisticated biological classifier (Kalat, 2015). The five senses, along with the inner thoughts, serve as inputs, and the output is represented by the activation of all kinds of muscles and also the inner thoughts. In between these "inputs" and "outputs" is a complex cognitive process going on, involving different parts of the brain for different tasks. However, all these different parts of the brain have in common that their cognitive abilities are mainly due to the workings of the same kind of structure: the biological neuron. The brain consists of billions of those neurons, which are "connected" with each other and work together in a big biological neural network.
Each biological neuron has in basic terms the same components. A schematic overview of these essential components can be seen in Figure 2.1a. The dendrites are together the input of a neuron. These dendrites carry outputs of other neurons. These outputs are in the form of electrical signals that last very short. At the location of a synapse, this output becomes via an electrochemical reaction the input of the next neuron. A biological neuron typically receives input from many thousands of connections with other neurons. All the electrical signals come together in the cell body, where they accumulate to a certain level. If this level is high enough, the neuron will "fire", i.e.



(A) Biological neuron (B) Artificial neuron (perceptron)

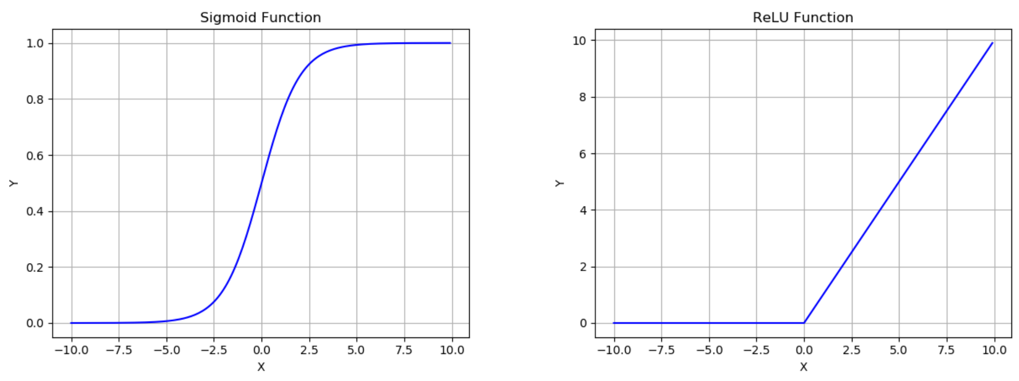
FIGURE 2.1: A schematic overview of the most essential components of a biological and artificial neuron. Images retrieved from (Gurney, 2014).

generates an electrical response by itself. It can do so via its output channel, which is called the "axon". Some incoming signals try to prevent the neuron from firing (inhibitory signals), while others try to make the neuron fire (excitatory signals).
The inputs and outputs of a single biological neuron change over time. For example, if an input from a certain other neuron is never used, it will deteriorate and ultimately be removed. This is a method through which the neurons can adapt to be able to contribute to a bigger effect on the outcomes of certain inputs of the brain, which can lead to the creature to have learned something. The field of neurophysiology has not yet discovered the exact workings of these mechanisms, which also make the brain much more energy-efficient than any artificial neural network implementation.

2.1.2 Perceptron

The workings of a biological neuron, and the learning power that can be achieved by using multiple of them, have been a great source of inspiration for an artificial counterpart that can be used for classification tasks. The perceptron (Rosenblatt, 1958) is one of the earliest inventions that led to the artificial neurons used in modern deep learning systems.
In the case of an artificial neuron, the input would be data from some instance in the data set. The output should be a classification of the input data. Besides classification, a neuron can also be used for a regression task. In that case, the output should be the predicted value of some variable. For example, the task can be, given the current weather conditions (temperature, wind speed etc.), calculate the most probable temperature after one hour.
In any case, the input is represented by numbers, and they must all be able to contribute to the output. The artificial counterpart of the inhibitory and excitatory signals is the use of weights. Every input value of an artificial neuron gets multiplied by a certain weight, such that they are able to either negatively or positively contribute to the total sum of all weighted inputs, and also can have a greater impact than other input values (or no impact at all if this would be desirable). In a formula, the output can thus be calculated in the following way:

$$z = \sum_{x \in X} x \cdot w_x \tag{2.1}$$



(A) Sigmoid function (B) ReLU function

FIGURE 2.2: Plots of the used activation functions. The figures are generated using custom Python scripts.

In this equation, z is the output of this function, X is the set of all input values, and w_x is the weight for input value x. If the input and the corresponding weights are treated as vectors, the equation can be rewritten using a dot product between input vector X and weight vector W:

$$z = X^\intercal \cdot W \tag{2.2}$$

2.1.3 Activation functions

The total sum of all weighted inputs is to be checked to see if it reaches the threshold to "fire" the output of the artificial neuron. This can be done using a hard threshold: output a value of 1 if the threshold has been reached and zero otherwise, but it is far more convenient to use a so-called activation function. Such a function takes as input the total sum of weighted inputs, and a non-linear transformation is applied to it. The use of a function instead of a binary threshold makes it more easy to make the neuron learn to adapt its response, since for that purpose the derivative of the function can be used (more on that in the next subsection).
Multiple functions have been proposed that can be used as an activation function. In this thesis two different functions are used in the experiments.

Logistic Sigmoid function

The first activation function is called the Logistic Sigmoid function, which is the most simple version of the general Sigmoid function. This function is as follows, with x being the input and y the (activated) output:

$$y = \frac{1}{1 + \exp -x} \tag{2.3}$$

The plot of the Sigmoid function can be seen in Figure 2.2a. The output value is (close to) zero for input values smaller than -5, and (close to) one for values bigger than 5. This makes these regions very insensitive for changes in the input. This is a good thing if they reflect an important aspect of the data, but a bad thing if they need to be (re)trained. In between these points the function has a gradual ascend in the form of an S. Parameters in this region are more sensitive to changes in the input: a relatively small change in the input leads to a relatively big change in the output.

ReLU function

The second activation function that is used during the experiments is the Rectified Linear Unit (ReLU) function. This function is more recently taken in use as activation function, and its workings are very simple. If the input value is negative the function returns zero, and otherwise it returns the input value. The plot of this function can be seen in Figure 2.2b.
The ReLU function does not simulate the output of a binary function in a continuous manner, but its output can be seen as a linear function with two linear pieces ($y = 0$ for $x \leq 0$ and $y = x$ for $x > 0$). This makes the ReLU function almost linear, and hence it has many of the properties that linear models have which makes it easy to optimize with methods based on the gradient (Goodfellow, Bengio, and Courville, 2016). Moreover, instead of the Sigmoid function which has two regions in which the gradient is very small, the ReLU function only has one such region. This makes it less likely that a certain neuron becomes saturated. In recent years the ReLU function has become the default recommendation to use as activation function (Nair and Hinton, 2010; Glorot, Bordes, and Bengio, 2011).

Other activation functions

There are quite a few other activation functions than the ones used in this thesis. For example the leaky ReLU (Maas, Hannun, and Ng, 2013), which introduces a small gradient in the part of the ReLU function with the input smaller than zero. Another function based on ReLU is the softplus function (Dugas et al., 2001; Nair and Hinton, 2010), which can be seen as the continuous version of ReLU. Softplus and ReLU were compared in (Glorot, Bordes, and Bengio, 2011) and it turned out the latter performed better. A different kind of function is the hyperbolic tangent function tanh, which is closely related to the Sigmoid function.

2.1.4 Loss functions and adapting weights

The introduction of a weight for each input value of a neuron created a way to make an input value more important for the total value (the sum that will be activated) than others. However, they can also be used as the changeable parameter with which a model can be fitted, i.e. changing the weights can lead to a learning effect on the perceptron.
In basic terms a weight can be increased, decreased, or left the same after an update. The error in the output gives an indication of the direction in which the weights must be changed. This error can be calculated using a so-called loss function. Multiple functions can be used for this purpose. The result of the loss function indicates the error in the output of the perceptron, however this output was generated using a sum of weighted inputs and an activation function. To determine in which direction each weight must be changed, it must first be found out how much a single weight can be "blamed" for the error in the output. This can be done using first derivatives. The total formula for the output of the activation function is as follows:

$$\hat{y} = a(z(X)) \tag{2.4}$$

In this equation, $\hat{y}$ is the output, $a()$ is the activation function, and $z(X)$ is the unactivated sum as calculated in Equation 2.2. In this equation, X consists of input values $x_{1...n}$, and W consists of corresponding weight values $w_{1...n}$. The only value in this equation that can be changed is the weight vector, since the input is data and hence constant and the activation function is also not changeable. The error J in the output can be calculated using the loss function, for example the mean-squared error (MSE) function:

$$J = \frac{1}{2}(y - \hat{y})^2 \tag{2.5}$$

To determine how the weight vector must be changed, the derivative of the error J with respect to the weight vector W can be calculated. The result yields the following equation:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial W} \tag{2.6}$$

As can be seen, the derivative of J with respect to W is calculated using the chain rule, because first the dot product between X and W is determined, which is then used as argument in the activation function $a()$, and finally this outcome is used to determine the error using the loss function J. When the partial derivatives are filled in, the following equation is obtained:

$$\frac{\partial J}{\partial W} = -(y - \hat{y}) \cdot a'(z) \cdot X \tag{2.7}$$

The partial derivative of J with respect to $\hat{y}$ is in the case of the MSE loss function $(y - \hat{y})$, the partial derivative of $\hat{y}$ with respect to z is $a'(z)$, whatever function a exactly is used, and the partial derivative of z with respect to W is the input vector X.

2.1.5 Gradient descent

The vector of $\frac{\partial J}{\partial W}$ consists of a value for each weight in the vector W that can be used to update the values in W such that the obtained error J would become smaller. The values in the vector indicate the direction in which the weights must be changed (which is the opposite direction, since the values must move downhill in the error landscape), but they do not indicate the exact magnitude of the change. They can be seen as the local gradient of each degree of freedom at the current position of the weights in the n-dimensional error space (where n is the number of weights in W). Somewhere in this error space is a position with the smallest error. However, since the current error landscape is an approximation of the one for the entire data set, it cannot be assumed that the minimum point for a single data example is at the same location as the one for all data. The use of batches of data points can reduce this uncertainty since this gives a better indication of the variation between the instances in the set.

It is tried to reach the position with the smallest value in the error space by changing the weights in such a way that the obtained configuration leads to an error as small as possible. There are a couple of methods to do this, such as second-order derivative methods from which Newton's method is most famous (Nocedal and Wright, 2006), which use Hessian matrices and Lagrangian functions to find out the best weight configuration. This method in theory only needs a couple of optimization steps, however one step is (until now) computationally extremely expensive. Therefore the method is hardly ever used in practice.

The method that is used most often is called gradient descent. With this method, the weights are changed by "following the gradient". This means that the weights are changed according to $\frac{dJ}{dW}$ such that a new configuration is found that should lie "downhill" in the n-dimensional error space. However, it is not convenient to change the weights with the values in $\frac{dJ}{dW}$, because these values are only obtained on the current data instance(s). These will hardly ever reflect the exact (relevant) properties of all the data. Since the network needs to be able to correctly classify as much data as possible, the weights in the network cannot be changed such that the current training examples will be classified optimally. Therefore the weights are updated by a small step in the direction of $\frac{dJ}{dW}$. The size of the small step is determined by the learning rate α. This parameter often has a small value, always between 0 and 1. We have obtained the general update rule for gradient descent, which is also the most basic optimization algorithm called Stochastic Gradient Descent (SGD):

$$W = W - \alpha \frac{\partial J}{\partial W}$$

Taking small steps has the disadvantage that it will take a long time until convergence has been reached, i.e. no further improvements can be made (which by the way does not always mean that all data will be classified correctly). There are multiple optimization algorithms that try to speed up this learning process. These algorithms are discussed in the next section.
If all goes well, the weights will ultimately have a configuration that leads to a vector $\frac{dJ}{dW}$ with only values very close to zero. This means that the gradient in every direction is zero, and thus that a local minimum in the error space has been found. However, it is not always guaranteed that this is also the global minimum, i.e. there could exist another configuration of the weights that yields an even smaller error. The local minimum is only guaranteed to be the global minimum too if the error space is convex. This means that there exists only one minimum in the error space, and this minimum can be reached from every other point in the error space. Unfortunately, most error spaces from real-world problems are non-convex, which means that they have multiple local minima and only one of them is the global minimum. Often the initial configuration of the weights (the start position in the error space) has a big influence on the probability to reach the global minimum.

2.1.6 Logistic regression

Until now only a single neuron, the perceptron, has been considered. For a classification task, multiple neurons are needed since with only one neuron only problems that are linearly separable can be solved (Alpaydin, 2009). In a classification task with a neural network, the output layer is normally composed of a number of neurons (also called nodes) equivalent to the number of classes. Often the Softmax method is used to determine the classification: the value of every output node is scaled between 0 and 1, in such a way that the values of all output nodes add up to 1. The chosen output node, which is the one with highest value, represents the classification made by the system.
When the output layer is the only layer that is used, every output neuron bases its output directly on input data. This kind of deep learning system is called a linear network, and along with a cross-entropy loss function the system is equivalent to logistic regression. This is in fact a type of classifier that can be implemented in multiple ways, and using a linear network is one of them. It is a special kind of linear

regression, in which the output is not linear but binary.
Since in a linear network the inputs have a direct connection with the output neurons, it is the case that the output is only formed by contributions from individual input values. However, in some cases this can not provide enough information for the system to learn and perform good classifications. In such a case it is needed to introduce multiple layers of neurons in the network.

2.1.7 Multi-layer perceptron

A neural network with multiple layers of nodes and in which the connections between nodes are always between a node of the current layer and a node of the next layer (i.e. a feed-forward network) is called a multi-layer perceptron (MLP). The MLPs that are used in the experiments in this thesis are all fully-connected. This means that there exists a connection between every node in the current layer and every node in the next layer, and no other connections exist (i.e. a connection between a node and another node multiple layers ahead, or a node connected to itself).
The layers in between the input layer and the output layer are called hidden layers, and the nodes in them are hidden nodes. This term "hidden" is used due to the fact that the output of a node in a hidden layer is not directly accessible and hardly interpretable. It should represent some statistic coming from a combination of multiple values from the previous layer, but the exact meaning is unknown. Some think this is a downside of an MLP, since it is not clear exactly why a certain output is generated upon an input. Research has been conducted trying to understand the meaning of outcomes of hidden nodes, which could then be used in explanations to users about how a network came to its classification. Especially custom applications will be expected to provide such information in the future. This subfield of AI is also known as explainable AI (XAI) (Gunning, 2017) and is yet in an early stage of development.
To come up with an output for a certain input of the system, information travels through the network according to the forward propagation method. In the first layer, the input data gets weighted and summed up for every node, after which the result gets activated by the activation function. These activated outputs of the nodes in the first layer act as the inputs of the second layer. In this layer, the weighted sum of the inputs is again calculated for every node, and the result is activated. This process will be repeated for every layer in the MLP, and in the end the output of the last layer is used to come up with the final classification.

2.1.8 Backpropagation

Learning in a neural network with multiple layers of nodes is in general similar to how a single perceptron learns: by adapting the weights. However, in this case it is a bit more difficult to know how each weight must be changed. In the case of one perceptron (and also the linear network), only one activated dot product between input data and weights is needed to obtain the output of a node. Since in an MLP there are multiple layers, the output of a node in the last layer is determined by taking the (activated) dot product of the weights and the outputs of the nodes in the previous layer. These latter values are each calculated in a same fashion. This means that the error obtained in the output of the final layer can be caused by the weights in the last layer, but also by the weights in the previous layer(s). So the question is which weights from which layers to blame for the obtained error. This is also sometimes called the credit assignment problem. A method that can solve this problem is called

backpropagation, as described in (Rumelhart, Hinton, and Williams, 1986).
The basics are the same as for a single neuron: the influence of a weight in the network on the error in the output can be determined by taking the first derivative of the error J with respect to the weight in W_n. This weight vector consists of all weights in layer n. If the network has three layers of nodes, the partial derivatives for the weights in layer 3 are as follows:

$$\frac{\partial J}{\partial W_3} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial W_3} \tag{2.8}$$

The terms are all the same as in Equation 2.7, and so z_3 is the unactivated output of layer 3 in the network, except for $\frac{\partial z_3}{\partial W_3}$. Since this is not the first layer, this term, which resembles the derivative of the unactivated output of layer 3 with respect to the weights in layer 3, cannot be replaced by the input. Instead, it must be replaced by the output of the previous layer (the second layer in the network), which is called H_2.
The influence of the weights in the second layer on the error in the output layer can be calculated in the same way, but now some more partial derivatives are needed:

$$\frac{\partial J}{\partial W_2} = \left(\sum_j \frac{\partial J_j}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_{3,j}} \cdot \frac{\partial z_{3,j}}{\partial H_2} \right) \cdot \frac{\partial H_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} \tag{2.9}$$

The first two partial derivatives are the same as before, but now they must be summed over all j nodes in the output layer, since a single weight in the second hidden layer has influence on the values of all output nodes in a network with fully-connected layers. The derivative $\frac{\partial z_3}{\partial H_2}$ resembles the derivative of the unactivated output of layer 3 with respect to the input of layer 3 (hence the output of layer 2), which results in the weight matrix W_3 of the third layer. This term "removes" the part of the error due to the weights in W_3 from the rest of the error that is propagated back to the second layer. $\frac{\partial H_2}{\partial z_2}$ is the derivative of the output of the second layer with respect to the unactivated output of the second layer, which results in the derivative of the activation function, this time with z_2 as argument. $\frac{\partial z_2}{\partial W_2}$ is the derivative of the unactivated output of the second layer with respect to the weight matrix W_2 of the second layer, which resembles the output H_1 of the first layer.
Finally, the influence of the weights in the first layer can be calculated using a derivative with even more partial derivatives:

$$\frac{\partial J}{\partial W_1} = \left(\sum_k \left(\sum_j \frac{\partial J_j}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial z_{3,j}} \cdot \frac{\partial z_{3,j}}{\partial H_{2,k}} \right) \cdot \frac{\partial H_{2,k}}{\partial z_{2,k}} \cdot \frac{\partial z_{2,k}}{\partial H_1} \right) \cdot \frac{\partial H_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} \tag{2.10}$$

In this equation, there must be summed over all k nodes in the second hidden layer and per node over all j nodes in the output layer, since a weight in the first hidden layer has influence on all the values from a single node in the first hidden layer to (all) the nodes in the second hidden layer, and thus also on the complete set of connections between the second hidden layer and the output layer. The additional terms for the first layer with respect to Equation 2.9 can be filled in analogously to the ones for the second layer, except for $\frac{\partial z_1}{\partial W_1}$ which can be replaced by the input vector X.
As can be seen, the earlier a layer in the network occurs, the longer the list of partial derivatives becomes. Moreover, the first terms of every equation occur in every derivative. Therefore, they are often stored as a variable δ_n (delta) in the following

way:

$$\delta_3 = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3}, \quad \delta_2 = \left(\sum_j \delta_{3,j} \cdot \frac{\partial z_{3,j}}{\partial H_2} \right) \cdot \frac{\partial H_2}{\partial z_2}, \quad \delta_1 = \left(\sum_k \delta_{2,k} \cdot \frac{\partial z_{2,k}}{\partial H_1} \right) \cdot \frac{\partial H_1}{\partial z_1} \tag{2.11}$$

In this way, δ_n can be determined upon calculating $\frac{\partial J}{\partial W_n}$, and used in the calculation of $\frac{\partial J}{\partial W_{n-1}}$.

2.1.9 Regularization methods

During the training process of a neural network the weights get changed very often. This can lead to unbalances in the network, for example in the case that one weight gets a very high value for some reason, or that all weights to a neuron become (nearly) zero. Such things have a bad influence on the ability and flexibility of training. Regularization methods can be used to prevent these things to occur. In the experiments in this thesis two regularization methods are used.

Dropout layer

The basic principle of a dropout layer (Srivastava et al., 2014) is that it removes some of the activation values from the previous layer. Or in other words it "drops" certain nodes "out" of the network. By doing this, the remaining nodes form a slightly different network, and must react to the absence of the "dropped out" nodes. This regularization method tries to make the network more robust against small differences in the input, without actually receiving these from the training data. This is valid, since real world data will always include some noise with respect to the examples in the data set.

Batch normalization

The input data of a neural network (and also for classifiers in general) must often be normalized, such that features in the data vector with a broader range of possible values do not have a bigger impact than features with a smaller range of possible values. The same can be done for the input of a hidden layer, that comes from a previous layer of nodes (Ioffe and Szegedy, 2015).
Batch normalization will normalize the activated output of the previous layer by subtracting the (mini-)batch mean and dividing by the standard deviation of the (mini-) batch. This prevents the activation values for becoming too large or too small, what prevents overfitting and allows higher learning rates to be used, and hence makes it easier to learn more different features. Including batch normalization in a network can speed up the training process significantly.

2.2 Optimization algorithms for gradient descent

As discussed above, a small part of the obtained gradient is subtracted from the current weights in a network to adapt them such that they perform not only better on the currently evaluated data instance(s), but hopefully also on the other instances in the data set. However, taking too small steps would slow down the learning process unnecessarily. Optimization algorithms for gradient descent try to speed up the learning process as much as possible, without letting go of the generalization to the entire data set. This section discusses all state-of-the-art optimization algorithms considered in this thesis. The algorithms can be divided into two groups: those using a version of standard stochastic gradient descent, and those using adaptive learning rate strategies.

2.2.1 SGD algorithms

Several optimization algorithms for gradient descent have been proposed to find a way to minimize the error in the output layer as fast as possible. The most basic, and also oldest one is stochastic gradient descent (SGD). This method takes a small step in the opposite direction of the gradient, using the learning rate α. This parameter has a fixed value during the entire learning process, often around 0.01, sometimes much higher depending on the data to be learned, but always smaller than 1. See also Equation 2.12, in which Θ_t are the parameters/weights of the model at epoch t, $g_t(\Theta_t)$ are the gradients for each weight in Θ_t at epoch t, and α is the learning rate.

$$\Theta_{t+1} = \Theta_t - \alpha \cdot g_t(\Theta_t) \tag{2.12}$$

Momentum

Standard stochastic gradient descent moves, independently from previous steps, with a fixed step size downhill on the error landscape. The analogy of a ball rolling down a hill introduced the idea to add momentum to the algorithm. The basic idea is that the direction and magnitude of the previous gradients contribute to the step size in the current epoch: when the gradient is still pointing in the same direction, the ball would accelerate according to the magnitude or the slope of the gradient. The moment the direction would change, i.e. the slope starts to go upward, the ball would slow down and so the step size would be decreased. This mechanism can be implemented in the following way (Polyak, 1964):

$$\Theta_{t+1} = \Theta_t - v_t \tag{2.13}$$

$$v_t = \gamma \cdot v_{t-1} + \alpha \cdot g_t(\Theta_t) \tag{2.14}$$

In these equations, γ is the momentum term that determines the influence of previous gradients on the current update. The general idea is to set this term as close to 1 as possible, and to choose for the learning rate α a value as high as possible, while conserving a stable convergence.

Nesterov Accelerated Gradient

The analogy of the ball rolling down the hill, gaining speed through momentum, seems a nice feature to implement. However, a ball would not stop once it is in the valley. It will roll further until no speed is left. If there is a slope uphill on the other side of the valley, this would cause the ball to eventually roll back, but it would take multiple oscillations before the ball stops at the minimum point in the valley. Nesterov (1983) came up with an idea to prevent the ball from going uphill again too much. For the calculation of the step size in the current epoch, it will calculate the gradient at the approximate location that the ball would end up using standard momentum. This gradient is then used to make a correction, which is especially convenient in the case when there is an uphill slope at the approximated location. It will prevent a too big step size that would go to the other side of the valley. Nesterov implements this method in the following manner:

$$\Theta_{t+1} = \Theta_t - v_t \tag{2.15}$$

$$v_t = \gamma \cdot v_{t-1} + \alpha \cdot g_t(\Theta_t - \gamma \cdot v_{t-1}) \tag{2.16}$$

A downside of this method is the calculation of $g_t(\Theta - \gamma \cdot v_{t-1})$, which can be computationally intensive and time consuming for some networks. Sutskever et al. (2013) propose to modify the calculation of v_t in the following way:

$$v_t = \gamma \cdot m_t + \alpha \cdot g_t(\Theta_t) \tag{2.17}$$

$$m_t = \gamma \cdot m_{t-1} + \alpha \cdot g_t(\Theta_t) \tag{2.18}$$

This latter method is also used in the built-in SGD with Nesterov momentum algorithm in the software used in the experiments in this thesis.

2.2.2 Adaptive learning rate algorithms

All versions of SGD, as described above, use for the calculation of the update of every individual parameter in the system the same learning rate. However, the information from the gradient in a single update could be much more important for one parameter than for another. In such a situation, the parameter with the important update should take a bigger step in the direction of the gradient than the parameter with a less important update. In other words: to speed up the learning process every parameter to be updated should have its own learning rate in every epoch. Cases in which the gradients are sparse (i.e. images with a lot of white space around the edges) should particularly benefit from this mechanism, but it also speeds up the learning process in other cases.

Adagrad

The first commonly used algorithm that does this, is Adagrad (Duchi, Hazan, and Singer, 2011). This method calculates a separate learning rate for every parameter in the model, based on the full history of the squared gradients of the parameter. The squared gradient resembles the magnitude of the gradient. They are summed up for each parameter, and the current learning rate is calculated by dividing the current gradient by the square root of the summed squared gradients plus a small value (to prevent division by zero). This means that, the bigger the previously obtained gradients were, the less important the current gradient is, and thus the smaller the learning rate and thus the step size in the current epoch would be. However, the sum of squared gradients will only increase, so later updates must have a larger gradient to obtain the same step size as earlier updates. The formal notation is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\sqrt{G_t} + \epsilon} \tag{2.19}$$

$$G_t = G_{t-1} + g_t(\Theta_t)^2 \tag{2.20}$$

In this algorithm, α is again the learning rate, and ϵ is the smoothing term, often set to $1e - 08$. This to prevent a situation where division by zero would occur. Note that in Equation 2.19, just like for all other optimization algorithms described in this thesis, the calculation of Θ_t term is for a single parameter in the weight matrix.

AdaDelta

The idea behind Adagrad is nice, but the algorithm has some downsides. The biggest one has to do with the behavior described above. After a while, the accumulation of the squared gradients reaches a value so high that the learning rates used in the updates become very small, and hence almost no improvements can be made. The algorithm AdaDelta (Zeiler, 2012) tries to tackle this problem by adapting the Adagrad algorithm such that the window of accumulated past gradients is restricted to a fixed size instead of taking the entire history. For an efficient implementation, the authors chose to use an exponentially decaying average of all past squared gradients, instead of storing the relevant previous gradients.
Apart from this, the authors introduce a method to remove the need of a learning rate from the algorithm. They note that the units of the learning rate and the decaying average of all past squared gradients do not match. They solve this by replacing the learning rate by another exponentially decaying average, of the squared parameter updates $\Delta\Theta_t^2$. This resembles the magnitude of the previous changes of the parameter. By dividing the square root of this by the square root of the resemblance of the magnitude of the previous gradients (both plus a small smoothing term ϵ), we end up with a value that more or less indicates the proportion of the magnitude of the previous gradients that was used to update the parameters, or in other words the influence of the previous gradients on the current value of the parameter. The formal notation is as follows:

$$\Theta_{t+1} = \Theta_t - g_t(\Theta_t) \cdot \frac{\sqrt{E(\Delta\Theta_t^2)_t} + \epsilon}{\sqrt{E(g^2)_t} + \epsilon} \tag{2.21}$$

$$E(\Delta\Theta_t^2)_t = \gamma \cdot E(\Delta\Theta^2)_{t-1} + (1-\gamma)\Delta\Theta_t^2 \tag{2.22}$$

$$E(g^2)_t = \gamma \cdot E(g^2)_{t-1} + (1-\gamma) g_t(\Theta_t)^2 \tag{2.23}$$

The parameters ϵ and γ are again the smoothing term (often set to $1e-08$) and the momentum term (often set to a value close to 1) respectively.

RMSprop

More or less parallel to the AdaDelta algorithm, the RMSprop algorithm was proposed by Geoff Hinton in Lecture 6e of his Coursera class (without an official publication, in collaboration with Tijmen Tieleman, Nitish Srivastava, and Kevin Swersky)[1]. This algorithm aims to solve the same issues in Adagrad as AdaDelta does, but keeps the learning rate. The formal notation is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\sqrt{E(g^2)_t} + \epsilon} \tag{2.24}$$

$$E(g^2)_t = \gamma \cdot E(g^2)_{t-1} + (1-\gamma) g_t(\Theta_t)^2 \tag{2.25}$$

Hinton suggests to use a value of 0.9 for the momentum term γ, while using a learning rate α of 0.001.

[1] `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

Adam

The Adaptive Moment Estimation (Adam) algorithm, proposed by Kingma and Ba (2014), is another algorithm that uses per-parameter learning rates. It is in some sense an extension of RMSprop, although not explicitly stated as such. Like RMSprop, an exponentially decaying average of past squared gradients v_t is used to calculate the step size of each parameter in the update. Additionally, an exponentially decaying average of past gradients m_t is included. This new value resembles the average direction of the past gradients, and it acts in a similar way as the momentum term in standard SGD: the more recently obtained gradients point into the same direction, the bigger the current steo size will be. The standard learning rate α is multiplied by (a bias-corrected) m_t, so the per-parameter learning rate gets higher when m_t gets higher, which is the case when the past gradients are in a more equal direction.
The authors include a bias-correction for both m_t and v_t, such that they are not biased towards zero. The formal notation is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.26}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.27}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t(\Theta_t) \tag{2.28}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.29}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t(\Theta_t)^2 \tag{2.30}$$

New parameters in this algorithm are β_1 and β_2, which are used in the two exponentially decaying averages. The authors propose standard values of 0.9 for β_1, 0.999 for β_2, and $1e-08$ for ϵ.

AdaMax

In the same paper as Adam is proposed, the authors propose a slightly different version called AdaMax (Kingma and Ba, 2014). In this algorithm, the ℓ_2 norm used in the calculation of v_t (see Equation 2.30) is replaced by the ℓ_∞ norm. This variation is proposed, since the ℓ_∞ norm generally exhibits stable behavior too. This change causes the exponentially decaying average of Equation 2.30 to be converted into the maximum term in Equation 2.34.[2] The bias-correction is omitted for u_t. The full formal notation is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\hat{m}_t}{u_t} \tag{2.31}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.32}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t(\Theta_t) \tag{2.33}$$

$$u_t = \max(\beta_2 \cdot u_{t-1}, |g_t(\Theta_t)|) \tag{2.34}$$

[2]In the definition of Equation 2.34 in (Kingma and Ba, 2014) the previous value of v_t is taken instead of u_t. This is an error: the v_t term is not used in AdaMax.

Nadam

The Nadam algorithm (Dozat, 2016) is the Adam algorithm with Nesterov acceleration included. To incorporate Nesterov acceleration into Adam, Dozat first rewrites the NAG update rule (see Equation 2.17 and Equation 2.18). He then rewrites the Adam update rule, and applies the new NAG update rule to the new Adam update rule. The $\hat{m}_t$ term resembles the momentum term, so we can put that in the place of the m_t term in Equation 2.18. The full update rule is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\beta_1 \cdot \hat{m}_t + \frac{(1-\beta_1)\cdot g_t(\Theta_t)}{1-\beta_1^t}}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.35}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.36}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t(\Theta_t) \tag{2.37}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.38}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t(\Theta_t)^2 \tag{2.39}$$

The fraction in the nominator in Equation 2.35 is only applicable if a decay is included for β_1, hence the notation β_1^t in that equation. If the same value is used throughout the learning process, this fraction term can be changed into $g_t(\Theta_t)$.

NadaMax

In the same way as Adam can be transformed into AdaMax (Kingma and Ba, 2014), Nadam can be changed into NadaMax. The Nesterov momentum term must be calculated in AdaMax, which can be done analogous to the method in Adam. First AdaMax is rewritten in a more convenient way, and then Nesterov momentum is added. We end up with the following update rule.

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\beta_1 \cdot \hat{m}_t + g_t(\Theta_t)}{u_t} \tag{2.40}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.41}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t(\Theta_t) \tag{2.42}$$

$$u_t = \max(\beta_2 \cdot u_{t-1}, |g_t(\Theta_t)|) \tag{2.43}$$

AMSGrad

In their recent paper "On the convergence of Adam and beyond" (Reddi, Kale, and Kumar, 2018), Google DeepMind employees Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar describe a kind of problem that cannot be solved by standard state-of-the-art optimization algorithms like Adam and RMSprop. They provide a small example optimization problem that shows the limitations of these algorithms. The problem function is stated as follows:

$$f_t(x) = \begin{cases} Cx & \text{for } t \bmod 3 = 1 \\ -x & \text{otherwise} \end{cases} \tag{2.44}$$

where $C > 2$, and $-1 \leq x \leq 1$. In each time step t a value x_t is received, and the goal is to obtain on the longer run (i.e. overall, in total, or cumulative) a value as small as possible. According to the authors, the optimal solution would be $x = -1$. The Adam algorithm however will always converge to $x = 1$. The authors propose a new algorithm called AMSGrad that can solve these kinds of problems, while keeping all the benefits from Adam. Experiments in the paper show that the algorithm even outperforms Adam in experiments on the CIFAR10 data set.
The new algorithm differs at two points from the Adam algorithm. The first difference is the calculation of $\hat{v}_t$, the bias correction for v_t. Instead of dividing this term by $1 - \beta_2$, the maximum is taken of the previous value of $\hat{v}_t$ and the current value of v_t. In this way, a sort of long term memory is incorporated for large gradients. The second difference with respect to Adam is that the bias correction for the m_t term is omitted. The formal notation of the algorithm is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{m_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.45}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t(\Theta_t) \tag{2.46}$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t) \tag{2.47}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t(\Theta_t)^2 \tag{2.48}$$

AMSGrad uses the same parameters as Adam. They propose a value of 0.9 for β_1 and 0.99 or 0.999 for β_2.

NAMSGrad

Analogously to Adam and AdaMax, AMSGrad can also be modified to incorporate Nesterov acceleration. The resulting algorithm NAMSGrad is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\beta_1 \cdot m_t + \frac{(1-\beta_1) \cdot g_t(\Theta_t)}{1-\beta_1^t}}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.49}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t(\Theta_t) \tag{2.50}$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t) \tag{2.51}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t(\Theta_t)^2 \tag{2.52}$$

2.2.3 Differences and similarities between algorithms

The algorithms described in the previous two sections are all adaptations of SGD or Adagrad. This means that some terms/techniques are used in multiple algorithms. To give a good overview of the different algorithms, they are described in terms of these general parts. The notation is loosely based on the generic algorithm described by Reddi, Kale, and Kumar (2018).

Building blocks

Most algorithms can be described in terms of a uniquely used combination of a small number of building blocks:

- Running average of past gradients (direction variation, momentum):

$$m_t = \gamma_1 m_{t-1} + (1 - \gamma_1) g_t(\Theta_t)$$

Algorithm	M_t	V_t
SGD	$g_t(\Theta_t)$	1
Adagrad	$g_t(\Theta_t)$	$V_{t-1} + g_t(\Theta_t)^2$
RMSprop	$g_t(\Theta_t)$	v_t
Adam	$\hat{m}_t$	$\hat{v}_t$
AdaMax	$\hat{m}_t$	$\max(V_{t-1}, \lvert g_t(\Theta_t)\rvert)$
Nadam	$\gamma_1\hat{m}_t + g_t(\Theta_t)$	$\hat{v}_t$
NadaMax	$\gamma_1\hat{m}_t + g_t(\Theta_t)$	$\max(V_{t-1}, \lvert g_t(\Theta_t)\rvert)$
AMSGrad	m_t	$\max(V_{t-1}, v_t)$

TABLE 2.1: The values for M_t and V_t for all algorithms that can be described in terms of the generic algorithm in Equation 2.53.

- Running average of past squared gradients (size variation, importance):

$$v_t = \gamma_2 v_{t-1} + (1-\gamma_2) g_t(\Theta_t)^2$$

- Bias corrections of m_t and v_t:

$$\hat{m}_t = \frac{m_t}{1-\gamma_2} \qquad \hat{v}_t = \frac{v_t}{1-\gamma_1}$$

Algorithmic framework

The general update rule for most algorithms can be described in general terms in this way:

$$\Theta_{t+1} = \Theta_t - \alpha \frac{M_t}{\sqrt{V_t} + \epsilon} \tag{2.53}$$

The M_t and V_t terms differ between algorithms. Table 2.1 shows for every algorithm the values in terms of the building blocks described above and the gradient $g_t(\Theta)$. The SGD algorithms with momentum cannot be described in terms of this generic algorithm, since they use a different notation (without any fraction). AdaDelta cannot be (efficiently) transformed too, since it does not use any learning rate α.

2.3 Convolutional Neural Network (CNN)

The convolutional neural network (LeCun et al., 1989), or CNN, is a special type of neural network that combines the multi-layer perceptron with convolutions. It is often used in a classification algorithm for data with a topological input, such as time-series data in the one-dimensional case, and image data in the two-dimensional case (M. Chen et al., 2017; Noh, Hongsuck Seo, and Han, 2016; Rocco, Arandjelović, and Sivic, 2017; Zbontar and LeCun, 2016). In this research, the CNN is used as one of the classifiers tested on the data sets MNIST and CIFAR10. This section discusses the internal operations of the CNN in general, and explains some reasons for using a CNN instead of a standard neural network.

2.3.1 Convolutional layer

In general, a CNN is a neural network with at least one layer that uses a convolution. Such a layer is called a "convolutional layer". A convolution is an operation on two functions, in this case with values in two-dimensional matrices called tensors. The general equation of the two-dimensional convolution is displayed in Equation 2.54.

$$S(i,j) = (I \cdot K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.54}$$

A convolution is done between tensors I, the input matrix or feature map, and K, the kernel or feature. K is (much) smaller than the feature map, and is slid over I. At each location, the convolution of K with the underlying part of I is calculated. This is done by multiplying the values at the same topological location, and summing the results. The convolution in a convolutional layer is by default a so-called "valid" convolution, which means that K is only convolved with the underlying part of I if there are no regions outside the input in the underlying part. This causes the result of a convolutional layer S being of smaller size than the input I. It is also an option to use a "full" convolution, in which case the convolution of K and a part of I is also performed at places where K only partially covers I.
The trained kernels can be used to detect certain patterns in the image data, as lines and curves. The output from previous convolutional layers can be combined in another convolutional layer to combine the detected shapes into the detection of more complicated shapes, like doors, eyes or even entire faces.

2.3.2 Pooling layer

A convolutional layer often appears in a CNN together with a ReLU layer, which incorporates the ReLU activation function as earlier discussed, and a pooling layer. The pooling layer (Krizhevsky, Sutskever, and Hinton, 2012) comes in different versions, but the general purpose of the pooling layer is to replace the output of the convolutional layer with a summary statistic of the neighboring outputs. The idea behind this is that a relevant feature in the input data might not always occur in the exact same spot, but somewhere in a (bit) wider region. Replacing the outputs of all locations in a certain region with a single value removes the need of the exact location: as long as the detection is made somewhere in the considered region it must be fine.
Multiple summary statistics can be used in a pooling layer. Max-pooling (Zhou and Chellappa, 1988) is a popular version, which replaces the output with the maximal value in the set of nearby values in the output. Other possibilities are average pooling, which takes the average of the set of nearby values, and weighted average pooling, which gives weights to output values based on the topological distance to the current value. In theory, any summary statistic function can be used in the pooling layer. Often the nearby values lie in a rectangular shape around the current value, but depending on the input another shape can be chosen (e.g. a circular shape). The size of the neighborhood, also called window size, is often small. Neighborhoods can overlap, depending on the size of the stride, which is the number of pixels that the window will be shifted. If there is little or no overlap, the output of the pooling layer is smaller than the input.

2.3.3 Architecture of a CNN

A typical CNN starts with a couple of convolutional layers, often with a ReLU and pooling layer in between, while at the end of the network one or more fully connected layers are implemented. In between the layers also a dropout layer can be inserted and/or batch normalization can be applied. The input of the first fully connected layer is a list of summary statistics obtained from the convolutional and pooling layers, based on the various different patterns that are searched for. The fully connected layers use these statistics to come up with a classification for the input of the network.

2.3.4 Learning in a CNN

During training, the weights in every layer in the CNN are adapted, such that the system should come up with better classifications. These weights are adapted using the error in the classification that is back propagated through the layers.
Backpropagation through and error calculation in the pooling layer is somewhat different from the method in a fully connected layer. The pooling layer does not really contribute to the error: it only reshapes and summarizes the output of the convolutional layer. Therefore, the error is only back propagated through the pooling layer, which is done by inverting the operation in the forward pass. For example, in a max-pooling layer only the local maxima are kept. Since only the local maxima are fed to the next layer, only those values contributed to the error. So in back-propagation the input of the pooling layer is reshaped, while only keeping the local maxima (all other values are set zero), and the back-propagated error is mapped on that reconstructed input. If another summary statistic function was used, backpropagation would consist of inverting that operation. I.e. in average pooling the error would be divided among all values that contributed to the local average.
Backpropagation and error calculation in the convolutional layer is also somewhat different from the way it is done in a fully connected layer, due to the shared weights in the kernels. For a single kernel in the layer, the calculation is as follows: the reconstructed output matrix of the convolution layer S', filled with back propagated errors, is convolved with the input matrix of the convolutional layer I, using a "valid" convolution. This results in a matrix of the size of the kernel, K', filled with delta values. This matrix can be piece-wise added to the kernel values in K, to adapt the weights. The error can be back propagated to the layer before the convolutional layer, by doing a "full" convolution between the reconstructed error matrix E and the kernel K with original values, using Equation 2.54. In that way, a matrix of the same shape as the I is reconstructed, filled with back-propagated error. This matrix can then be passed on to the previous layer.

2.3.5 Reasons to use a CNN

There are several reasons to use a CNN instead of a standard multi-layer perceptron for classifying images (Goodfellow, Bengio, and Courville, 2016).

- An image as input often results in very big layers, since every pixel is one input value. If there would only be the ordinary neural network, so with full connections between the layers, a lot of memory must be used in each calculation to store the weights. In a convolutional layer there is no full connection, instead there is "sparse" connectivity. This is due to the kernel being smaller than the input: every node in the layer has only k incoming and outgoing connections, where k is the number of values in the kernel. This sparse connectivity results in fewer connections than in a fully connected layer, and thus fewer weights to store.

- Input from a binary image, as in the MNIST data set, contains quite a lot of zeros (white space), which would not have much influence on the final classification. Therefore it would be redundant to provide input from those locations with separate weights. The CNN due to its sparse weights does not do this.
- In a fully connected layer, every weight is independent of all the others, which results in many values that must be updated and stored. A convolutional layer has "shared" weights, since the weights are the values in the kernel. Every time a convolution is made with a certain area of the input, it is done with the same kernel, and thus the weights are the same. Thus in one convolutional layer there are only k independent weights, instead of the $n \times m$ weights in a fully connected layer from a layer with n nodes to a layer with m nodes.
- Multiple kernels can be used in one convolutional layer. One kernel can be seen as a filter to detect a certain small pattern in the input image. By using multiple kernels, there can be searched for multiple small patterns at the same time. The outputs of those different filters can then be used in the next convolutional layer, in which more complicated shapes can be detected, built from the small patterns.

Chapter 3

New ideas for optimization algorithms

In the previous chapter, many current state-of-the-art optimization algorithms for gradient descent in deep learning have been discussed. In this chapter, the new ideas for the algorithms that possibly lead to better learning performances are explained. It starts with a new idea that can be implemented in the standard SGD algorithm. After this, some ideas are discussed to improve the adaptive learning rate technique in algorithms based on Adagrad and Adam.
Finally, in the last section of this chapter, some general methodology is discussed and explained of the conducted experiments in this thesis.

3.1 SGD adaptation: handbrake momentum

Standard momentum adds a value to the current update step size, based on the current gradient and the previous update, which is (by recursion) based on all previous gradients. In this way, the step size increases (accelerates) when multiple steps towards the same direction are taken, and decreases (slows down) when the gradient starts to point to the opposite direction. Since the optimum value, or the minimum error, must be found, there might in theory be an opportunity for faster convergence, if the added momentum would be zero the moment the current gradient points in another direction than the previous one. In that way, the weight would not unnecessary move in the "old" direction, which results in oscillation around the (local) optimum, but stay at the (local) minimum once it has been found. This technique can be implemented in the standard stochastic gradient descent algorithm with momentum, by setting the momentum term to zero the moment it seems to have reached the (local) minimum. This algorithm can be stated formally as follows:

$$\Theta_{t+1} = \Theta_t - v_t \tag{3.1}$$

$$v_t = \gamma \cdot M_t \cdot v_{t-1} + \alpha \cdot g_t(\Theta_t) \tag{3.2}$$

$$M_t(v_{t-1}, g_t(\Theta_t)) = \begin{cases} 1 & \text{if } v_{t-1} \cdot g_t(\Theta_t) \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

In this algorithm, α is the learning rate, γ is the momentum term, and M_t is the "handbrake" term. This last term acts like a boolean. It is 1 if the gradient still points into the same direction, and 0 if this is not the case. By multiplying the momentum part in v_t with this value, the momentum is reset for a weight at the moment the local minimum of the weight has been reached. To evaluate if the current momentum and the current gradient point in the same direction, they are multiplied with each other. If the result is positive, they do point in the same direction (both are positive or both are negative). If the result is negative, they point into the opposite direction (one is positive, while the other is negative). If the result is zero, it means the current gradient must be zero. In this case, it is not needed to reset the momentum term since this does not automatically indicate a local minimum. Note that, except for the first learning step, in practice the momentum term can hardly ever become zero. This would only be the case if multiple gradients of zero are received after each other, such that the remaining momentum from previous gradients becomes zero too.

3.2 Adaptive learning rate algorithms

Most considered state-of-the-art optimization algorithms for gradient descent that use adaptive learning rates have by design the same method to calculate the "importance" of the current gradient with respect to the previous gradients. The previous gradients are represented by an exponentially decaying average of past squared gradients. By dividing the learning rate (and some momentum term in some algorithms) by the square root of this average, the learning rate for each weight in the current update can be calculated. In this way, weights that received larger gradients in the past will receive a smaller update for the same current gradient than other weights. The key observation is that the magnitude of the previous gradients and the scaling factor for the learning rate are calculated using a method similar to the ℓ_2 norm [1]. In the paper of the first adaptive learning rate algorithm that was proposed, Adagrad, the authors state that they have tested several methods for determining the importance of the gradient and that the ℓ_2-norm had the best results. Therefore they have used this method. However, the more recent algorithms do not use a pure norm anymore, but a sort of decayed version. Moreover, in the AdaMax algorithm it is shown that a different method, in this case the infinity norm, can also have a good performance. So it seems that the results of the different methods as tested on the Adagrad algorithm do not necessarily apply to other algorithms as well. This might indicate that some other method to determine the importance of the gradient could lead to a better result for these algorithms. In fact in theory any method is possible, as long as a higher magnitude of previous gradients leads to a lower scaling factor for the learning rate, which is the core idea of adaptive learning rates.
Taking this into account, it was tried to come up with some alternative **gradient history collection (GHC)** methods that can be implemented in the state of the art optimization algorithms. To compare the standard and the to be introduced alternatives algorithm-wise the RMSprop algorithm is taken as an example, since this is the most basic algorithm that uses gradient history collection in an exponentially decaying average. The standard version of RMSprop is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\sqrt{v_t} + \epsilon} \tag{3.4}$$

$$v_t = \gamma \cdot v_{t-1} + (1-\gamma) g_t(\Theta_t)^2 \tag{3.5}$$

[1]Although a norm is slightly different (except for the method used in Adagrad), since it normally does not include a decaying factor, the methods for gradient history collection used in the algorithms are referred to as "norm" in the rest of this thesis.

In this algorithm, α is the learning rate, ϵ is the smoothing term, and γ is the momentum term. The exponentially decaying average v_t uses squared gradients to collect the gradient history, while in the update rule the square root of v_t is taken.
Four alternative ways to collect information about the gradient history are defined and tested in the experiments. Each alternative way can be implemented in every algorithm that uses the standard method.

3.2.1 Absolute difference

The first alternative is named "Abs", since it takes the absolute gradient instead of the squared gradient in the exponentially decaying average of past gradients. However, it keeps the square root from the standard method. The "Abs" version of RMSprop can be defined as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\sqrt{v_t} + \epsilon} \tag{3.6}$$

$$v_t = \gamma \cdot v_{t-1} + (1-\gamma)|g_t(\Theta_t)| \tag{3.7}$$

3.2.2 Alternative norm

The second method is able to use a very dynamic value for the power and the root, however in a single update the power in the exponentially decaying average and the root in the denominator use the same value, similar to a norm. In this case the absolute value of the gradient must be taken before the power is calculated, to ensure only positive values are added and to make the use of an odd norm possible. The value for the norm can be the same throughout the entire learning process, but it can also be increased or decreased using a fixed schedule during (a part of) the learning process.
This so-called ℓ_{1-4}-version of RMSprop can be defined as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\sqrt[\Lambda_t]{v_t} + \epsilon} \tag{3.8}$$

$$v_t = \gamma \cdot v_{t-1} + (1-\gamma)|g_t(\Theta_t)|^{\Lambda_t} \tag{3.9}$$

$$\Lambda_t = \begin{cases} \min(\Lambda_0 + \lambda \cdot t, \Lambda_{max}) & \text{if } \lambda \geq 0 \\ \max(\Lambda_0 + \lambda \cdot t, \Lambda_{min}) & \text{otherwise} \end{cases} \tag{3.10}$$

As the name of the algorithm already suggests, it is highly recommended to keep the value of Λ_t between 1 and 4, since a smaller value leads generally to bad results and a higher value leads hardly to improvements and therefore only introduces more expensive computations.

3.2.3 Adaptive norm

The different gradient history collection methods give different "rates of importance" to the same previous gradients. The second alternative method described above can be used to illustrate the difference in convergence if a ℓ_1, ℓ_2, or ℓ_3 norm is used. To this end, the ℓ_{1-4}-version of the Adam optimizer is considered. Three parameter settings are defined, in which only the norm differs between ℓ_1, ℓ_2, and ℓ_3. Each variation must find the minimum of the Rosenbrock function from the same starting

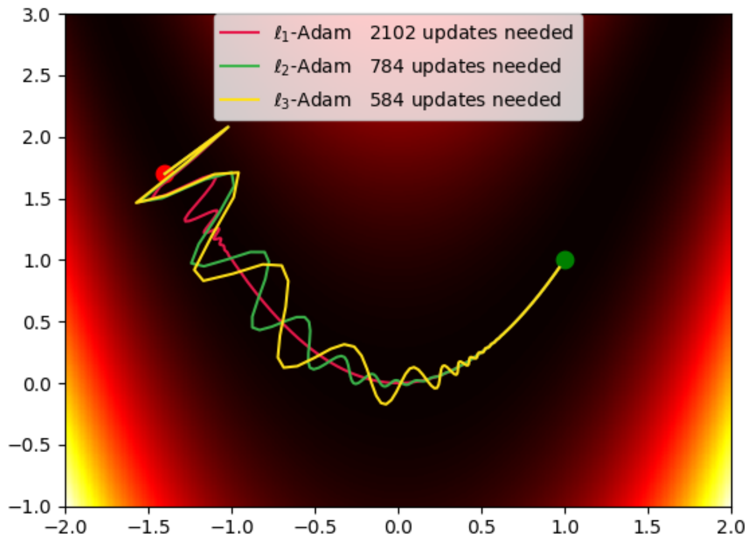


FIGURE 3.1: Comparison of the path towards convergence of the Adam optimizer using a ℓ_1, ℓ_2, and ℓ_3 norm in the gradient history collection.

point. The resulting plot is displayed in Figure 3.1. It can be seen that the higher the power in the algorithm, the more oscillation and hence exploration occurs. Additionally, it is the case that the algorithms with higher powers need less steps to converge to the minimum of the function. The one with ℓ_1 needed 2102 steps, ℓ_2 norm (hence standard Adam) needed 784 steps, while the algorithm with a ℓ_3 norm only needed 584 steps. This observation and also the results in the next chapter indicate that the use of a higher norm leads to more exploration, and also that the use of a norm other than ℓ_2 leads to (slightly) better results in almost all cases. Probably, it would be even better if the algorithm could somehow automatically change the used norm every update, such that it is even more dynamic than the increasing or decreasing norm scheme that the ℓ_{1-4} method allows. To this end, it is tried to come up with an adaptation of the standard adaptive algorithms that changes the used norm every update, based on information in previous updates. The resulting algorithm, called "ℓ_{ada}", includes the third method of gradient history collection, and the version of RMSprop is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\sqrt[\Lambda_t]{v_t} + \epsilon} \tag{3.11}$$

$$v_t = \gamma \cdot v_{t-1} + (1-\gamma)|g_t(\Theta_t)|^{\Lambda_t} \tag{3.12}$$

$$\Lambda_t = \Lambda_0 - \frac{P_t}{Pm_t} \tag{3.13}$$

$$P_t = \gamma_P \cdot P_{t-1} + (1-\gamma_P)|g_t(\Theta_t)| \tag{3.14}$$

$$Pm_t = \max(Pm_{t-1}, |g_t(\Theta_t)|) \tag{3.15}$$

The power value Λ at time step t is in this method defined as Λ_0 minus the ratio of the exponentially decaying average of past absolute gradients and the maximum absolute gradient. This means that the norm can vary between Λ_0 and $\Lambda_0 - 1$ (or even a smaller range, which is also dependent on γ_P), and it becomes lower the steeper (absolute higher) the current gradient is. In this way, it is tried to allow more exploration in the case of a small gradient and less exploration for a big gradient. Several other ratios were tested, such as only the current absolute gradient divided by the maximum gradient (which is still possible in the final algorithm if γ_P is set to zero), and dividing the exponentially decaying average by an exponentially decaying maximum value (e.g. $Pm_t = \max(0.9Pm_{t-1}, |g_t(\Theta)|)$, but these had (far) worse results than the ratio in the final algorithm. The algorithm has two new parameters: γ_P which has a default value of 0.9, and Λ_0 which has a default value of 3. Furthermore, it is needed to set the initial value of Pm_t to a small positive number (like the ϵ parameter in the standard algorithms), to prevent division by zero in case all gradients so far have been zero.

3.2.4 Exponential function

All considered methods, and the standard method, use a method similar to a norm for the gradient history collection. However, since the used function must have the property that a higher input value results in a higher output value, other functions can also be considered. The last method uses a power function in the exponentially decaying average of past gradients. Instead of the squared gradient, e to the power of the gradient is taken. And instead of the root in the update rule, it uses the natural logarithm of the result to compute the scaling factor for the learning rate. For this to work, v_t must be initialized at 1, since the log of a value less than 1 is negative, while the denominator in Equation 3.16 must be positive.
The "Exp" version of RMSprop can be defined as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{g_t(\Theta_t)}{\ln(v_t + \epsilon)} \tag{3.16}$$

$$v_t = \gamma \cdot v_{t-1} + (1 - \gamma) e^{|g_t(\Theta_t)|} \tag{3.17}$$

Note that AdaDelta needs to use the adaptations in the GHC of the algorithm as proposed in the four methods above in both of its exponentially decaying averages, to preserve the reason of the authors for introducing the second average: "keeping the units the same".

3.3 General methodology

The new alternative optimization algorithms, as described in the first sections of this chapter, will be compared to the state of the art optimization algorithms on a number of experiments. These experiments cover the next four chapters, and within each chapter the specific setup of the experiments are described. However, some aspects of the experiments are performed in more than one experiment. These parts of the methodology are described and explained in this section.

3.3.1 Experiments

A total of 16 different experiments have been conducted to find out if the adapted versions of the optimization algorithms outperform the state of the art ones. The next five chapters cover the results of the experiments, grouped by and in order of

SGD	**SGDmom**	**SGDNAG**	**SGDHB**
Adagrad	Absgrad	$\ell_{1-4}-$Adagrad	**AdaMax**
AdaDelta	AbsDelta	$\ell_{1-4}-$AdaDelta	NadaMax
RMSprop	Absprop	$\ell_{1-4}-$RMSprop	**ExpAdam**
Adam	**AbsAdam**	$\ell_{1-4}-$**Adam**	$\ell_{ada}-$**Adam**
Nadam	AbsNadam	$\ell_{1-4}-$Nadam	
AMSGrad	AbsAMSGrad	$\ell_{1-4}-$AMSGrad	
NAMSGrad	AbsNAMSGrad	$\ell_{1-4}-$NAMSGrad	

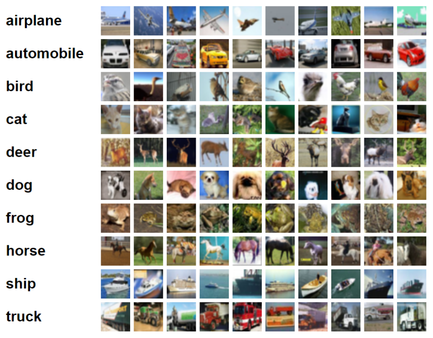
TABLE 3.1: All algorithms considered in the experiments

the complexity of the classifier.
The experiments are on four different problems. The most basic one is the optimization of the Rosenbrock optimization function. For this problem no deep learning system is needed, instead the two input values of the function can be changed using the pure optimization algorithms and the partial derivatives of the function. A bit more difficult is the XOR problem, which does need a (small) neural network to be able to find a solution. For this problem, the algorithms are tested on four different neural networks, where for one network it is easier to find a solution than for another.

Used datasets: MNIST and CIFAR10

The majority of the experiments that are conducted in this thesis are about the classification of images from a data set. Two different data sets are used for this purpose: MNIST and CIFAR10.
The MNIST data set consists of 28x28 binary images of handwritten digits. The data set is constructed by Yann LeCun (LeCun et al., 1998), who also was one of the developers of the convolutional neural network. The data set was designed as an easy to use real-world data set. LeCun used a subset of a data set collected by NIST (National Institute of Standards and Technology) and made some modifications (hence the M) such that no extra preprocessing is required when the data set is used. This makes it easy to use in studies with the main focus on (aspects of) the classifier. Moreover, since there are only 784 input values per example, a classifier will not need a huge amount of memory to process an image from this data set.
The data set used in the experiments was retrieved from (LeCun, 1998). It contains 60000 training examples and 10000 test examples with 10 different classes (the digits 0 to 9). Some example binary images are displayed in Figure 3.2a.
The CIFAR-10 data set (Krizhevsky and Hinton, 2009) consists of 32x32 RGB images of objects from 10 classes: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship' and 'truck'. The images form a subset of the "80 million tiny images" data set (Torralba, Fergus, and Freeman, 2008). Like the images in the MNIST data set, the images in the CIFAR-10 data set have already been preprocessed. They are however more difficult to classify since they are bigger and above all they are RGB images instead of binary images, which means that every pixel has three values (red, green, blue), each in a wider range than binary values.
The used data set was retrieved from `https://www.cs.toronto.edu/~kriz/cifar.html`, and consists of 50000 training images (5000 per class) and 10000 test images (1000 per class). Some example images are displayed in Figure 3.2b.

(A) Some examples of the binary images in the MNIST data set. (Retrieved from `https://en.wikipedia.org/wiki/MNIST_database#/media/File:MnistExamples.png`)



(B) Some example images per class from the CIFAR-10 data set. Image retrieved from `https://www.cs.toronto.edu/~kriz/cifar.html`.

FIGURE 3.2: Example images from the two data sets.

Overview of algorithms used in the experiments

The experiments on the Rosenbrock function, on the XOR problem, and logistic regression on the MNIST data set, are conducted with all considered algorithms. 29 algorithms in total are considered, including state of the art algorithms and adaptations of multiple state of the art algorithms using the new methods described in the previous sections. A full list of all considered algorithms can be seen in Table 3.1. However, for sake of proper use of time, it was decided to reduce the number of algorithms to be considered for the experiments with more complex deep learning systems, and hence a longer duration time, such that more of these kinds of experiments could be conducted. This reduced selected group contains 12 different algorithms, which are put in bolt in Table 3.1. Some of the experiments are conducted on a small group of additional algorithms, which are the newly proposed adaptations of one of the state of the art algorithms derived from Adam. It was decided to include those to find out if the indicated effect of the new adaptation on Adam itself in the results also applies to the relevant derivative.

Software

All experiments are implemented in the Python programming language, version 3.6.4 (Python Core Team, 2018). For the experiment on the Rosenbrock function, the optimization algorithms are customarily implemented using the Numpy package (Walt, Colbert, and Varoquaux, 2011). For all other experiments, the PyTorch package (Paszke et al., 2017) is used for all deep learning techniques and the optimization algorithms. Packages like TensorFlow (Abadi et al., 2016) and Caffe (Jia et al., 2014) could not be used since they do not provide the amount of space for custom implementation as PyTorch does. For the implementation of every new adaptation in every considered state of the art algorithm, an additional module had to be created. Moreover, the state of the art algorithms "Nadam", "NadaMax", and "NAMSGrad" are for some reason not included in the standard PyTorch package, so these had to be custom created too. The experiments and parameter optimization runs of the experiments on the Rosenbrock function and on the XOR problem are run on a standard desktop computer. All runs of the other experiments are run on the Peregrine cluster from the University of Groningen via an SSH protocol and the program MobaXterm.

3.3.2 Scoring system

Since the algorithms will be compared on their performances in the experiments, and since it is quite complicated and extensive to fully analyze every difference in performance between all algorithms in every experiment, a scoring system is introduced to compare the performance of the algorithms in a more succinct manner.
For this scoring system, only those algorithms are considered that are involved in every single experiment conducted in this thesis. This group consists of the 12 algorithms in the selected group.
For each experiment, the considered algorithms are ranked based on their performance on a certain statistic, which is in most of the experiments the training cost (the lower the reached cost the better). It is decided to prefer to use the reached training cost as the criterion over the reached test cost since this gives a better indication of the pure optimization capabilities of the algorithms. In some of the more easy experiments, where every algorithm was able to reach 100% accuracy on the training set, the algorithms are compared on a statistic indicating the optimization speed, i.e. how many iterations over the training set it took before 100% accuracy was reached. The exact relevant statistic for each experiment is indicated by an asterisk (*) behind the name of the considered statistic in the result tables of the experiments.
Each algorithm earns a number of points for its performance on each individual experiment. The best performing algorithm gets 12 points, the second best 11 points etc. The worst performing algorithm gets 1 point. In this way, it is much easier to see which algorithm performs better on which kind of problems, and also the general performance can be determined and individual algorithms can be compared.

3.3.3 Learning feature plots

Apart from a comparison in general performance, so in terms of reached training cost etc, it is also needed to compare the algorithms on their behavior during the optimization process. For a simple problem as the Rosenbrock function (discussed in the next chapter), this can be done by comparing the visited positions in the 2D error space before convergence on the global minimum as displayed on a 2D heat map of the error space. However, with more complicated problems where the error space can have more than a thousand dimensions (weights) this is of course not possible.
For certain problems, certain statistics might be useful to keep track of. For example, the step size per weight update could indicate something about the convergence and the way of exploration. Another aspect is the obtained gradient for each weight update. A smaller gradient would mean that the current parameter configuration lies closer to the (local) minimum than a greater one.
A total of eight so-called features are kept track of during the runs in the experiments on the MNIST or the CIFAR10 data set.

- The average and standard deviation of the gradient, calculated by taking the average over the gradients for all weights in the neural network.
- The average and standard deviation of the absolute gradient, calculated by taking the average over the absolute gradients for all weights in the neural network.
- The average and standard deviation of the delta weight, so the average value added to all weights.

- The average and standard deviation of the absolute delta weight, so the average of the absolute values added to all weights.

For each feature, one value per parameter update is obtained. A plot can be made of the feature value over all parameter updates in the training process. The plots of different algorithms can then be compared to find indications of certain behaviors during the optimization process.

Chapter 4

Experiment on the Rosenbrock optimization function

All considered algorithms as stated in Table 3.1 are compared on their ability to optimize the performance of a deep learning system. However, since quite a few other mechanisms in those systems also affect the performance, the algorithms are also compared on a pure optimization problem. In this way, the optimization power of just the algorithms can be compared. Moreover, this test case served as a good indicator for any new ideas for the optimization functions.
This chapter describes this experiment, by first explaining the problem to be solved. After this, the experiment itself is explained and described and the results are discussed.

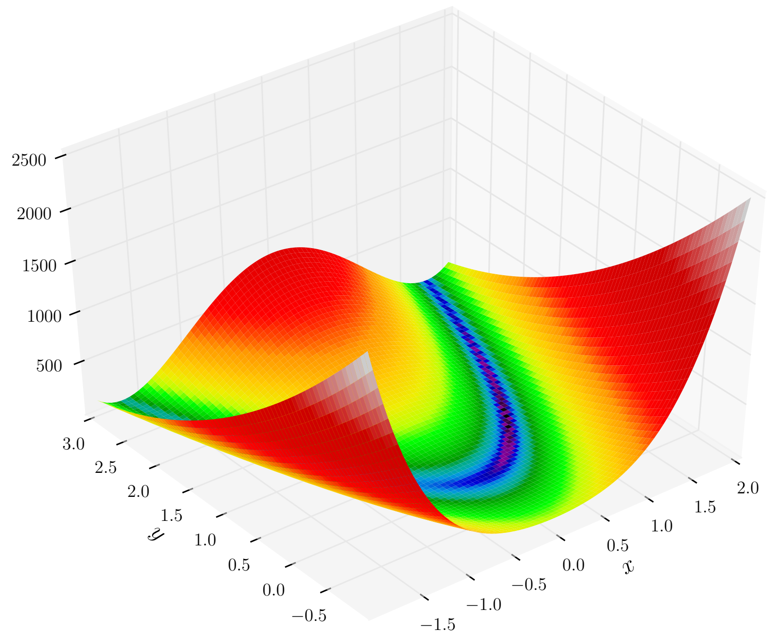


FIGURE 4.1: Plot of the Rosenbrock function. The global minimum of zero is at point (1,1)

4.1 Rosenbrock function

Over the years, several functions have been invented that can be used to test optimization algorithms[1]. Some of these functions are pretty complicated, while others are very simple. Since an optimization function is needed in which the algorithms must be able to find the global minimum, a convex optimization function must be chosen. A highly non-convex function, like the Eggholder function (Whitley et al., 1996), is not suited since it contains many local minima and this is not convenient for gradient descent. For example, genetic algorithms could solve such a problem much easier, but since gradient descent is used in the learning process of the deep learning systems considered in this thesis, it is also used in this case. Next to the function being convex, it must also have only 2 input values. This must be the case such that the optimization "paths" of the algorithms can be plotted in a heat map of the function. In that way, the behavior of the algorithms during optimization can be compared, and algorithms with similar strategies can be identified.
The Rosenbrock function (Rosenbrock, 1960) is convex and has 2 input values, and is therefore chosen to be used as a test case for the optimization algorithms. The general Rosenbrock function is defined as:

$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \tag{4.1}$$

The goal of the algorithms is to minimize the value of $f(x,y)$. The values for a and b determine the exact function but are usually set to 1 and 100 respectively. These values are also chosen in this experiment. The considered ranges are $-2 \leq x \leq 2$ and $-1 \leq y \leq 3$. A plot of the function within this scope can be seen in Figure 4.1. It can be seen that the function landscape is composed of a curved trench with quite steep walls. Moreover, in the trench there is also a little gradient, and the purple area indicates the region around the global minimum. This minimum lies on the exact point $(x,y) = (1,1)$ and has a value of 0.

4.2 Experimental setup

The optimization algorithms have to be able to find the global minimum of the Rosenbrock function from any start point on the function landscape using as few parameter updates as possible. An initial random position in the 2D landscape of the function is chosen, and the algorithm has to find the coordinates of the global minimum of the function, and do so in as few steps as possible.
For each algorithm, the optimal parameter settings must be found. This is done by running each considered parameter setting on 50 randomly chosen start points in within the considered parameter range. The setting resulting in the lowest average number of updates needed to find the minimum of the function is chosen. For each algorithm, the optimal parameter setting is displayed in Table A.1 in Appendix A. Note that for AbsDelta no parameter setting could be found that resulted in convergence behavior. Also, note that for ℓ_{1-4}-RMSprop no better parameter setting could be found than the one of standard RMSprop.
After these optimal values have been found, the main experiment can be conducted. Each algorithm is tested on a set of 400 different starting positions in the Rosenbrock function. These start positions are also plotted on the heat map of the Rosenbrock

[1]A nice overview can be found at `https://en.wikipedia.org/wiki/Test_functions_for_optimization`

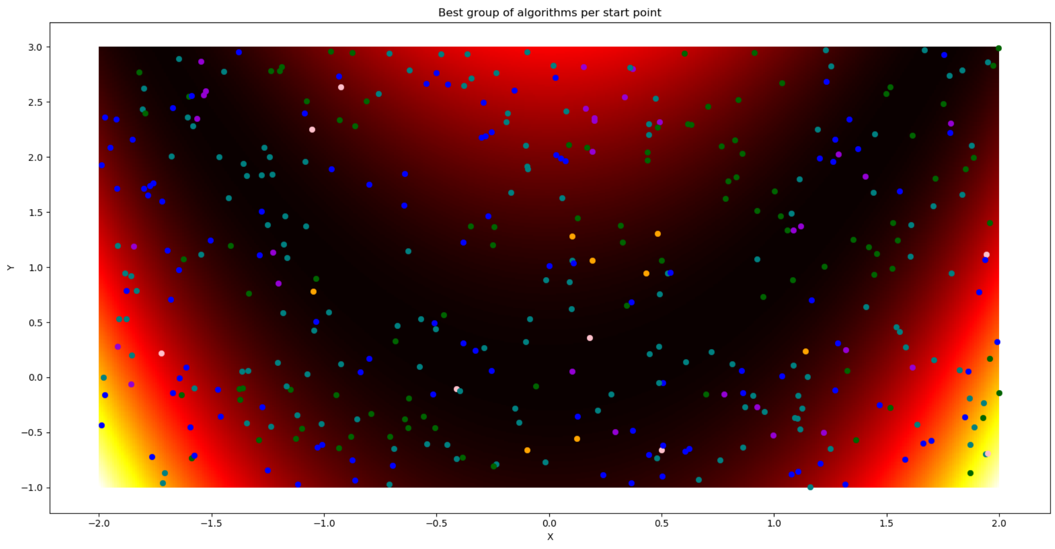


FIGURE 4.2: All the starting points in the Rosenbrock function in the data set. The colors represent the family of algorithms to which the algorithm belongs that had the best result. Purple = SGD, Green = Std Ada, Blue = Abs Ada, Pink = Exp Ada, Orange = ℓ_{ada} Ada, Teal = ℓ_{1-4} Ada

function in Figure 4.2. For each test case, the number of updates that the algorithm needed to converge to the minimum is taken. The maximum number of updates per test case was set on 300000: if an algorithm still did not find the global minimum this maximum number is taken to be the number of updates needed. Since this will turn out to be much more than an algorithm would need if it finds the global minimum, it has a huge impact on the average number of updates needed.

4.3 Results (A)

The results of this experiment are displayed in Table 4.1. The average and standard deviation of the number of updates needed are shown, as well as the convergence rate (i.e. in how many of the test cases was the algorithm able to find the minimum). It turns out that the majority of the algorithms was able to converge to the minimum in all test cases: only SGD, SGDHB, all variations of AdaDelta, NAMSGrad, and ℓ_{1-4}-NAMSGrad had some misses. The algorithm with the smallest average number of updates needed turned out to be ℓ_{1-4}-Adam, while ℓ_{1-4}-AMSGrad needed a similar number of updates. AbsDelta was the worst algorithm since it could not find the minimum of the function from any start position.
Looking at the difference between SGDHB and the other SGD versions, the former took much longer on average to find the optimum and it also had more cases in which it was not able to find the minimum at all.
Comparing the standard, "Abs", and "ℓ_{1-4}" versions of the algorithms with adaptive learning rates, it can be seen that in 4 cases (Adagrad, RMSprop, Nadam, NAMSGrad) the "Abs" version had the best result, in 2 cases (Adam, AMSGrad) the "ℓ_{1-4}" version had the best result, and in 1 case (AdaDelta) the standard version had the best result.
The "Exp" version of Adam scored better than the standard version, but worse than the "Abs" and "ℓ_{1-4}" versions. Finally, the "ℓ_{ada}" version of Adam scored better than

Algorithm	**Avg upd.***	**Std Dev**	**Conv. rate**	**# best res**
SGD	3832	21011	398/400 (99.50%)	0
SGDmom	324	88	400/400 (100.00%)	10
SGDNAG	407	115	400/400 (100.00%)	6
SGDHB	46180	105584	341/400 (85.25%)	14
Adagrad	4510	2103	400/400 (100.00%)	8
AdaDelta	43418	79693	366/400 (91.50%)	2
RMSprop	3273	991	400/400 (100.00%)	0
Adam	726	281	400/400 (100.00%)	4
AdaMax	434	164	400/400 (100.00%)	11
Nadam	771	363	400/400 (100.00%)	19
NadaMax	740	401	400/400 (100.00%)	25
AMSGrad	466	126	400/400 (100.00%)	3
NAMSGrad	13170	58552	384/400 (96.00%)	26
Absgrad	2069	893	400/400 (100.00%)	19
AbsDelta	300000	0	0/400 (0.00%)	0
Absprop	2974	1051	400/400 (100.00%)	3
AbsAdam	333	381	400/400 (100.00%)	8
AbsNadam	440	343	400/400 (100.00%)	**64**
AbsAMSGrad	7793	3061	400/400 (100.00%)	4
AbsNAMSGrad	1870	899	400/400 (100.00%)	11
$\ell_{1-4}-$**Adagrad**	3490	1787	400/400 (100.00%)	3
$\ell_{1-4}-$**AdaDelta**	47513	97833	348/400 (87.00%)	7
$\ell_{1-4}-$**RMSprop**	3273	991	400/400 (100.00%)	0
$\ell_{1-4}-$**Adam**	**228**	66	400/400 (100.00%)	46
$\ell_{1-4}-$**Nadam**	482	260	400/400 (100.00%)	37
$\ell_{1-4}-$**AMSGrad**	231	66	400/400 (100.00%)	38
$\ell_{1-4}-$**NAMSGrad**	21312	74981	373/400 (93.25%)	16
ExpAdam	387	129	400/400 (100.00%)	8
$\ell_{ada}-$**Adam**	445	197	400/400 (100.00%)	8

TABLE 4.1: Results of all optimization algorithms on the Rosenbrock function. The experiment included 400 different randomly chosen starting points, and the maximum number of epochs was set to 300000. The values of the average and the standard deviation of the number of updates needed to reach convergence are rounded.

the standard version, but worse than all other versions.
In Table 4.1 it is also indicated in the rightmost column for each algorithm in how many of the test cases it had the best result, i.e. it managed to find the optimum the fastest of all algorithms. The "Abs" version of Nadam had the best result here with 64 fastest times. If the different versions of the same standard algorithm are compared on this result, it turns out that SGDHB had the best result among the SGD algorithms. This is remarkable, since SGDHB needed on average far more updates to converge than most of the other algorithms. This might indicate that the test cases in which SGDHB was fastest to find the optimum were very hard to solve. For Adagrad, RMSprop, and Nadam the "Abs" version scored best, while for AdaDelta, Adam, and AMSGrad the "ℓ_{1-4}" version had the best result. NAMSGrad is the only algorithm from which the standard version had the best result.
In Figure 4.2 all 400 start positions are drawn in the heat map of the Rosenbrock function. Each start position has the color of the algorithm group to which the fastest algorithm belongs. It was hoped that something could be inferred from this plot about in which cases (high or low initial gradients etc.) which algorithm group performed best. As can be seen, every algorithm group has best performances all over the map, so not much can be inferred here.

4.4 Discussion

From the results in this experiment it can be concluded that, apart from SGDHB, the new adaptations can result in a faster convergence than the standard versions of the algorithms. The new version ℓ_{1-4} seems to be most promising, but the Exp, ℓ_{ada} and Abs versions performed often better than the standard version too.

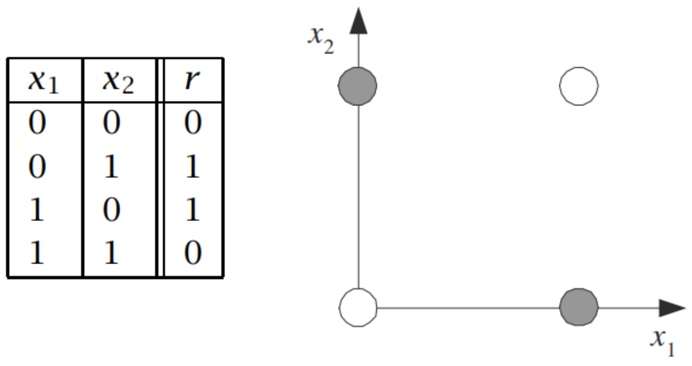
4.4.1 Differences in behavior of algorithms

To gain some insight in the differences in optimization behavior between the algorithms, all versions are tested on a single case, and the trajectory that has been followed by the algorithm in the function heat map is plotted. In Figure B.1, Figure B.2 and Figure B.3 an example trajectory of each optimized algorithm is displayed. Algorithms based on the same state-of-the-art algorithm are placed next to each other, and in every caption the number of updates needed for the displayed run is given. In general, it can be seen that the algorithms that follow the gradient, like RMSprop and AdaDelta, are not the fastest algorithms. ℓ_{1-4}-Adam was the fastest, with only 144 updates needed, and looking at the path in the plot it started with some exploration, then the oscillation dampens, and only in the final stage the pure gradient is followed. Algorithms like ℓ_{ada}-Adam, Adam, AdaMax, and ℓ_{1-4}-Amsgrad have a similar "strategy", but in this case they overshoot the optimum point, and have to make a turn before they converge. Furthermore, most "Abs" versions and SGDmom and SGDNAG are bouncing a lot on the borders of the considered parameter range, which helps them converging. This could mean that, without this help, they would have needed much more updates to converge. Standard SGD and SGDHB have a more gentle path within the borders, but they need almost ten times as much updates. Finally, ExpAdam seems to have a slightly different oscillation pattern, which looks less smooth than the algorithms using a power function for the gradient collection. However, it converges faster than standard Adam in this case.

Chapter 5

Experiments on the XOR problem

The next group of experiments conducted for this thesis is on the XOR problem. Four different small neural networks are trained to solve this problem using the full set of considered optimization functions.



x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 5.1: Plot of the XOR problem. The two classes are clearly not linearly separable: there is no method to draw a single line that separates them. (Modified image from (Alpaydin, 2009).)

5.1 The XOR problem

The XOR ("exclusive or") problem is a classic but very easy problem to solve when compared to training on entire data sets. It is based on the workings of the logical "exclusive or" operator. In the classic XOR problem, there are 2 binary inputs and 1 binary output. The output will only be 1 if either of the two inputs is 1. If the two inputs have the same value, the output will be 0. The XOR problem cannot be solved by a single perceptron, since the problem space is not linearly separable (Russell and Norvig, 2016) (see also Figure 5.1). This caused the field of neural networks (in the time it was called "cybernetics") to become very unpopular. Only after the invention of back-propagation for the multi-layer perceptron the field regained its popularity.

5.2 General methodology

Some of the aspects across the four different neural networks trained on the XOR problem are the same. As activation function the Sigmoid function is used in this experiment. The mean-squared error is chosen as the loss function. The weights in the network are initialized at random values between -0.1 and 0.1, this to encourage

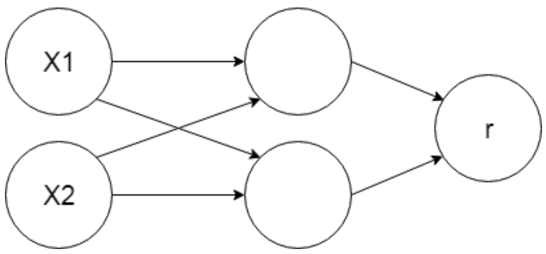


FIGURE 5.2: The neural network with two hidden nodes, used in the experiment on the XOR problem

the learning process. The neural networks are trained in a stochastic or online fashion, i.e. the training examples are presented one by one.

5.2.1 Experimental setup

The procedure in the experimental setup is also the same in the four experiments. To find out the parameter settings for the best performance of each algorithm, multiple parameter settings are compared on 50 test cases. Per test case the number of epochs needed to reach a cost smaller than 0.001 is saved. The maximum number of epochs is set to 20000 and if an algorithm exceeds this number it has failed to solve the XOR problem. The setting that on average needs the smallest amount of epochs is chosen as the optimal one. The optimal parameter settings for all four experiments are displayed in Table A.2 and Table A.3 in Appendix A.
During the main experiments, each optimized algorithm is tested for 200 runs, every time with another begin configuration of the weights. The number of epochs in one run is different for the four experiments. Each epoch consists of a full iteration over the training set, and after each epoch the average loss is saved. Per run a data file is created in which the average loss per epoch is stored.

5.3 Neural network with 2 hidden nodes

The first experiment on the XOR problem is with one of the smallest MLPs possible. The MLP consists of the two input nodes, followed by a hidden layer of 2 nodes, and finally one node in the output layer, and is displayed in Figure 5.2. No biases were included in the network. This setup is known to be the smallest neural network capable of solving the XOR problem. It was first believed that there exist multiple local minima in the error landscape of this neural network (I. G. Sprinkhuizen and E. J. Boers, 1996), but later it was found out that these local minima only exist with infinite parameter values, and do not occur with finite ones (I. Sprinkhuizen and E. Boers, 1999). In practice this means that local minima are possible. This neural network is still the most difficult one to optimize when compared to the three other networks in the other experiments on the XOR problem.

5.3.1 Results (B)

The obtained results for all optimization algorithms can be seen in Table 5.1. The convergence rate indicates in how many of the 200 cases the function was learned (i.e. the final cost was smaller than 0.001). The columns "Avg # upd needed" and

Algorithm	**Convergence rate**	**Avg # upd needed***	**st.dev.**
SGD	179/200 (89.5%)	9732	3275
SGDmom	**197/200 (98.5%)**	1448	1078
SGDNAG	195/200 (97.5%)	1410	1065
SGDHB	**197/200 (98.5%)**	1257	727
Adagrad	121/200 (60.5%)	401	104
Absgrad	165/200 (82.5%)	12459	801
$\ell_{1-4}-$**Adagrad**	126/200 (63.0%)	1783	653
AdaDelta	182/200 (91.0%)	9236	3209
AbsDelta	172/200 (86.0%)	9990	3213
$\ell_{1-4}-$**AdaDelta**	180/200 (90.0%)	9301	3507
Adam	133/200 (66.5%)	746	119
AbsAdam	180/200 (90.0%)	572	1351
$\ell_{1-4}-$**Adam**	128/200 (64.0%)	475	1207
ExpAdam	141/200 (70.5%)	2915	274
$\ell_{ada}-$**Adam**	141/200 (70.5%)	515	334
RMSprop	149/200 (74.5%)	373	36
Absprop	190/200 (95.0%)	834	919
$\ell_{1-4}-$**RMSprop**	148/200 (74.0%)	**321**	49
AMSGrad	141/200 (70.5%)	586	1347
AbsAMSGrad	186/200 (93.0%)	595	1173
$\ell_{1-4}-$**AMSGrad**	123/200 (61.5%)	778	199
Nadam	132/200 (66.0%)	1131	596
AbsNadam	185/200 (92.5%)	783	135
$\ell_{1-4}-$**Nadam**	121/200 (60.5%)	1330	339
NAMSGrad	126/200 (63.0%)	870	1084
AbsNAMSGrad	185/200 (92.5%)	538	85
$\ell_{1-4}-$**NAMSGrad**	123/200 (61.5%)	377	50
AdaMax	115/200 (57.5%)	2559	379
NadaMax	133/200 (66.5%)	531	756

TABLE 5.1: The results of all optimization algorithms on the XOR problem, using a neural network with 2 hidden nodes and no batch normalization. The experiment included 200 different randomly chosen starting points, and the maximum number of epochs was set to 20000.

"st.dev." indicate how many updates were needed on average to reach a cost smaller than 0.001, and the standard deviation of this statistic respectively. Note that if the algorithm was not able to reach such a low cost in a run, the run will not be counted for the statistics about the number of updates needed (this applies also to the results of the other experiments on the XOR problem).

It can be seen that none of the algorithms managed to learn the XOR function in all 200 runs. The highest accuracy was reached by SGD with momentum and SGD with handbrake momentum (98.5%), where the latter needed on average less updates, and also had a smaller standard deviation. Given these statistics, it can be concluded that in this setup the handbrake momentum had a minimally equal convergence rate and a faster convergence speed than all other SGD algorithms.

Taking a look at the adaptive learning rate algorithms and their versions, it can be seen that, except for AdaDelta, the "Abs" version of each algorithm had the highest convergence rate. However, only for Nadam the convergence speed of the "Abs" version was also the highest of all versions. Instead, often the "ℓ_{1-4}" version needed the least amount of updates on average, with "ℓ_{1-4}-RMSprop" being the fastest on average of all algorithms. The results of the "ℓ_{ada}" version of Adam are in terms of the convergence rate better than all other versions except the "Abs" version, and in terms of speed better than all other versions except the "ℓ_{1-4}" version. The "Exp"

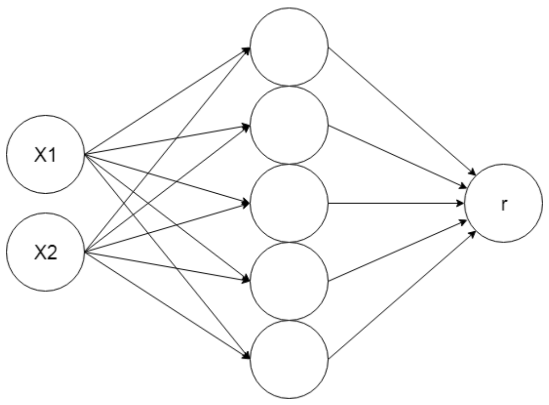


FIGURE 5.3: The neural network with five hidden nodes used in the experiments on the XOR problem.

version of Adam turned out to have a convergence rate which is only worse than the Abs version, but it is much slower in converging than all other Adam versions.

5.4 Neural network with 5 hidden nodes

This neural network differs from the one in the experiment described in the previous section by the number of nodes in the hidden layer. Where the previous network only had 2 hidden nodes, does this one contain 5 hidden nodes. This provides the network with a lot more weights to be optimized, and hence more ways in which the network can be trained to solve the XOR problem. The network is displayed in Figure 5.3.

5.4.1 Results (C)

Table 5.2 shows the results on the network with 5 hidden nodes and no batch normalization. It can be seen that the algorithms in general had fewer problems to find the optimal solution on this network than on the smaller network. Almost half of the algorithms were able to find the optimal solution in all 200 cases, and those that could not reach a convergence rate of 1.0 did reach a higher rate than on the previous network.
All algorithms using SGD with some kind of momentum had a very similar result: they all had a convergence rate of 1.0, and both the average and the standard deviation of the average number of updates needed are very close to each other. However, they are much higher than the ones of most of the adaptive learning rate algorithms. For these algorithms again the "Abs" versions seemed to have the best convergence rate. However, in terms of the number of updates needed they were not always the fastest. ℓ_{1-4}-AMSGrad turned out to be the overall fastest with an average of 128 (std=79), also much faster than the other versions of AMSGrad. The "Exp" version of Adam was again the worst performing algorithm both in convergence rate and speed. The convergence rate of the "ℓ_{ada}" version of Adam is in between the one of the standard version and the ℓ_{1-4} version, and the average speed seems to be higher than both of them: the average number of updates needed of the ℓ_{1-4} version is a bit smaller, but its standard deviation turns out to be much higher.

Algorithm	Convergence rate	Avg # upd needed*	Std.dev.
SGD	199/200 (100.0%)	8874	2403
SGDmom	**200/200 (100.0%)**	895	262
SGDNAG	**200/200 (100.0%)**	936	275
SGDHB	**200/200 (100.0%)**	950	238
Adagrad	191/200 (95.5%)	370	1058
Absgrad	**200/200 (100.0%)**	7576	1582
$\ell_{1-4}-$**Adagrad**	198/200 (99.0%)	341	183
AdaDelta	**200/200 (100.0%)**	8602	2386
AbsDelta	199/200 (99.5%)	8567	2542
$\ell_{1-4}-$**AdaDelta**	198/200 (99.0%)	8584	2175
Adam	197/200 (98.5%)	252	194
AbsAdam	**200/200 (100.0%)**	179	23
$\ell_{1-4}-$**Adam**	196/200 (98.0%)	182	977
ExpAdam	194/200 (97.0%)	776	1390
$\ell_{ada}-$**Adam**	197/200 (98.5%)	224	49
RMSprop	190/200 (95.0%)	234	351
Absprop	**200/200 (100.0%)**	291	148
$\ell_{1-4}-$**RMSprop**	**200/200 (100.0%)**	232	222
AMSGrad	186/200 (93.0%)	226	772
AbsAMSGrad	**200/200 (100.0%)**	182	35
$\ell_{1-4}-$**AMSGrad**	197/200 (98.5%)	**128**	79
Nadam	195/200 (97.5%)	238	983
AbsNadam	**200/200 (100.0%)**	170	25
$\ell_{1-4}-$**Nadam**	198/200 (99.0%)	226	88
NAMSGrad	**200/200 (100.0%)**	147	44
AbsNAMSGrad	**200/200 (100.0%)**	170	21
$\ell_{1-4}-$**NAMSGrad**	195/200 (97.5%)	183	357
AdaMax	196/200 (98.0%)	285	68
NadaMax	**200/200 (100.0%)**	605	1485

TABLE 5.2: The results of all optimization algorithms on the XOR problem, using a neural network with 5 hidden nodes and no batch normalization. The experiment included 200 different randomly chosen starting points, and the maximum number of epochs was set to 20000.

5.5 Neural network with 2 hidden nodes and batch normalization

This experiment is on a similar network as the one shown in Figure 5.2, but now with an additional batch normalization layer applied to the output of the hidden layer. This addition should give the optimization process a boost, according to the influence of batch normalization on experiments in the literature (He et al., 2016; Huang et al., 2017).

5.5.1 Results (D)

The results for this network can be seen in Table 5.3. It is clear that the usage of batch normalization gives a huge boost to both the convergence rate and the convergence speed, when these results are compared with the ones in Table 5.1. All algorithms now manage to find the optimal solution in all 200 cases. However, some algorithms need some more updates than others. Therefore these slower algorithms, which are all algorithms based on AdaDelta, used a maximum number of updates of 10000. All other algorithms had a maximum of 5000 updates.
Since all algorithms reached a convergence rate of 1.0, the different algorithms can

Algorithm	Convergence rate	Avg # upd needed*	Std.dev.
SGD*	200/200 (100.0%)	1100	770
SGDmom	200/200 (100.0%)	216	213
SGDNAG	200/200 (100.0%)	226	324
SGDHB	200/200 (100.0%)	169	155
Adagrad	200/200 (100.0%)	119	183
Absgrad	200/200 (100.0%)	224	51
$\ell_{1-4}-$**Adagrad**	200/200 (100.0%)	132	343
AdaDelta*	200/200 (100.0%)	840	623
AbsDelta*	200/200 (100.0%)	1125	880
$\ell_{1-4}-$**AdaDelta***	200/200 (100.0%)	1089	817
Adam	200/200 (100.0%)	77	16
AbsAdam	200/200 (100.0%)	63	22
$\ell_{1-4}-$**Adam**	200/200 (100.0%)	51	14
ExpAdam	200/200 (100.0%)	96	115
$\ell_{ada}-$**Adam**	200/200 (100.0%)	49	16
RMSprop	200/200 (100.0%)	180	19
Absprop	200/200 (100.0%)	62	25
$\ell_{1-4}-$**RMSprop**	200/200 (100.0%)	123	14
AMSGrad	200/200 (100.0%)	54	12
AbsAMSGrad	200/200 (100.0%)	73	27
$\ell_{1-4}-$**AMSGrad**	200/200 (100.0%)	87	10
Nadam	200/200 (100.0%)	117	12
AbsNadam	200/200 (100.0%)	100	35
$\ell_{1-4}-$**Nadam**	200/200 (100.0%)	147	15
NAMSGrad	200/200 (100.0%)	74	8
AbsNAMSGrad	200/200 (100.0%)	77	27
$\ell_{1-4}-$**NAMSGrad**	200/200 (100.0%)	68	11
AdaMax	200/200 (100.0%)	548	59
NadaMax	200/200 (100.0%)	**43**	20

TABLE 5.3: The results of all optimization algorithms on the XOR problem, using a neural network with 2 hidden nodes and batch normalization. The experiment included 200 different randomly chosen starting points, and the maximum number of epochs was set to 10000 for algorithms with *, and 5000 for the others.

only be compared on the convergence speed. It turns out that the SGD version with handbrake needed the least updates on average, and also had the smallest standard deviation of all SGD versions. So it can be concluded that this new SGD version had the best performance among them. However, also in this case most of the adaptive learning rate algorithms were (much) faster.
For this network, there is no version that has a clearly better performance in all algorithms. For Adagrad, AdaDelta, and Amsgrad it seems the standard version had the best performance, while for RMSprop and Nadam the "Abs" version performed best and for NAMSGrad the "ℓ_{1-4}" version performed best. Moreover, the "ℓ_{ada}" version of Adam outperformed all other versions of Adam. The overall best performing, hence fastest algorithm turned out to be NadaMax with 43 updates needed on average.

5.6 Neural network with 5 hidden nodes and batch normalization

The final network is in theory the easiest to optimize when compared to the three other ones. This network both uses 5 nodes in the hidden layer (similarly to the

Algorithm	Convergence rate	Avg # upd needed*	Std.dev.
SGD	200/200 (100.0%)	514	161
SGDmom	200/200 (100.0%)	77	24
SGDNAG	200/200 (100.0%)	74	26
SGDHB	200/200 (100.0%)	82	26
Adagrad	200/200 (100.0%)	52	12
Absgrad	200/200 (100.0%)	186	23
$\ell_{1-4}-$**Adagrad**	200/200 (100.0%)	42	11
AdaDelta	200/200 (100.0%)	502	123
AbsDelta	200/200 (100.0%)	528	173
$\ell_{1-4}-$**AdaDelta**	200/200 (100.0%)	522	195
Adam	200/200 (100.0%)	40	4
AbsAdam	200/200 (100.0%)	32	6
$\ell_{1-4}-$**Adam**	200/200 (100.0%)	29	3
ExpAdam	200/200 (100.0%)	25	7
$\ell_{ada}-$**Adam**	200/200 (100.0%)	**13**	3
RMSprop	200/200 (100.0%)	54	10
Absprop	200/200 (100.0%)	49	23
$\ell_{1-4}-$**RMSprop**	200/200 (100.0%)	45	12
AMSGrad	200/200 (100.0%)	23	3
AbsAMSGrad	200/200 (100.0%)	32	6
$\ell_{1-4}-$**AMSGrad**	200/200 (100.0%)	31	3
Nadam	200/200 (100.0%)	23	3
AbsNadam	200/200 (100.0%)	27	5
$\ell_{1-4}-$**Nadam**	200/200 (100.0%)	32	4
NAMSGrad	200/200 (100.0%)	26	3
AbsNAMSGrad	200/200 (100.0%)	28	5
$\ell_{1-4}-$**NAMSGrad**	200/200 (100.0%)	26	3
AdaMax	200/200 (100.0%)	33	6
NadaMax	200/200 (100.0%)	28	7

TABLE 5.4: The results of all optimization algorithms on the XOR problem, using a neural network with 5 hidden nodes and batch normalization. The experiment included 200 different randomly chosen starting points, and the maximum number of epochs was set to 5000.

network in Figure 5.3), which provides more ways to optimize the network such that it solves the XOR problem, and it applies batch normalization to the output of the hidden layer, which speeds up the learning process as well. It is therefore expected that all algorithms achieve in this experiment better results than in the three other experiments.

5.6.1 Results (E)

The results for this network can be seen in Table 5.4. In these runs, the maximum number of updates was set at 5000.
Since all algorithms have a convergence rate of 1.0, they can again only be compared in terms of convergence speed. It turns out that again the algorithms based on AdaDelta have the worst performance, together with standard SGD, with more than 500 updates on average needed and a standard deviation between 100 and 200. The SGD versions with momentum are again much faster than standard SGD, and in this case the version with Nesterov momentum slightly outperforms the others.
Most of the algorithms with adaptive learning rate again outperform the ones using SGD. Only the AdaDelta versions and the "Abs" version of Adagrad are slower. The "ℓ_{1-4}" version of the algorithms outperforms the other versions for Adagrad

and RMSprop. For all other algorithms the standard version had the best performance, except for the Adam algorithm, where the "ℓ_{ada}" version had the best result. ℓ_{ada}-Adam also has the best result of all algorithms, with only 13 updates needed on average and a standard deviation of 3.

5.7 Discussion

The results of the algorithms on the four networks differ very much. For all algorithms it is the case that the worst performance is on the network with two hidden nodes, followed by five hidden nodes, two hidden nodes and batch normalization, and finally five hidden nodes and batch normalization. This means that the addition of batch normalization has a greater positive effect than the addition of three extra hidden nodes on the convergence rate.
However, this research is about the influence of the optimization algorithm. The results show that on average the more a network gets help from other mechanisms (like batch normalization or extra nodes) the smaller the influence of the optimization algorithm becomes. In the experiment with five hidden nodes and batch normalization all algorithms were able to reach a perfect accuracy, and especially between all the derivations of standard Adam the difference in speed was small. Certainly when compared to the AdaDelta variants and standard SGD, that needed roughly 20 times the amount of epochs on average. The experiment with 2 hidden nodes and no batch normalization however showed that in these more bare circumstances the use of the right optimization algorithm does make a big difference. The standard SGD algorithm with momentum and the SGD algorithm with handbrake momentum reached an almost perfect score, while most other algorithms were not even able to reach 75%. Moreover, for almost all adaptive learning rate algorithms the Abs variant reached a (much) better score than the standard version, while the other variants often had a more or less equivalent convergence rate with a sometimes higher convergence rate.
These differences between the networks can also be seen in the plots of the average cost over the runs, as shown in Appendix C. In Figure C.1 it can be seen that for the experiment with 2 hidden nodes practically every algorithm converges on its own final cost (and does so in general after only a couple of hundreds epochs, as also indicated by the convergence speeds in Table 5.1). This effect is also present in the plot for the experiment with a network with 5 hidden nodes, which is displayed in Figure C.3, but it has disappeared in Figure C.2 and Figure C.4 which show the plots for the experiments with batch normalization included. In these two latter plots almost all algorithms manage to reach a cost of zero, and only the speed in which they do so differs. Moreover, this difference in speed is smaller when the three extra hidden nodes are used.
From these experiments it can be concluded that the choice of the optimization algorithm can have a big influence on the training performance, but this influence seems to become smaller when a bigger network is used and especially when batch normalization is added. The new introduced variants of the optimization algorithms had some promising results. It seems that SGDHB is at least as fast as the other SGD algorithms with momentum, and regarding the different GHC methods it seems that Abs is more stable, i.e. has a higher convergence rate, while the others seem to have a similar convergence rate but with a higher speed.

Chapter 6

Experiments with logistic regression

The most basic version of the neural network, a network where input and output are directly connected (also called "linear network"), can be used in a setup for logistic regression. This type of classifier has been discussed more elaborately in subsection 2.1.6. It is used here as a test case on image data that is closest to pure optimization (Reddi, Kale, and Kumar, 2018), so without any additional techniques.
In this chapter, the two experiments using a logistic regression setup are discussed. The chapter starts with the experiment on the MNIST data set and continues with the experiment on CIFAR10.

6.1 Logistic regression on MNIST

The classifier in this experiment is not expected to have a very high performance with any optimization algorithm since initial research by LeCun et al. (1998) showed that more complex systems are needed to accomplish that. However, it does serve as a kind of baseline regarding the optimization power of the different algorithms for the MNIST data set. All 29 considered algorithms are tested in this experiment.

6.1.1 Experimental setup

The classifier in this experiment consists of $28 \cdot 28 = 784$ inputs nodes, which are directly (fully) connected to the 10 output nodes, representing the 10 different classes (which are in fact the digital numbers 0 to 9). As loss function the cross-entropy method is used, which incorporates the Softmax functionality.
Every optimization algorithm is optimized by running different parameter settings each for 5 runs during 15 iterations over the entire training set. A batch size of 128 is used, and the training data is shuffled every iteration. The training cost is kept track of, and for every parameter setting the average is calculated of the minimum training cost reached per run. The setting that resulted in the smallest average training cost is taken as the optimal parameter setting. These configurations are displayed in Table A.4 in Appendix A.
The experiment itself is conducted by running every optimization algorithm for 5 runs during 250 iterations over the training set. The average scores for the training cost and -accuracy and test cost and -accuracy over each iteration are calculated and saved in data files.

Algorithm	**Min training cost***	**Max training acc**	**Min test cost**	**Max test acc**
SGD	0.2392	93.55%	0.2714	92.63%
SGDmom	0.2326	93.71%	0.2769	92.56%
SGDNAG	0.2325	93.72%	0.2760	92.57%
SGDHB	0.2393	93.58%	0.2724	92.59%
Adagrad	0.2187	94.09%	0.3073	92.62%
Absgrad	0.2472	93.25%	**0.2674**	**92.74%**
$\ell_{1-4}-$**Adagrad**	0.2195	94.05%	0.3057	92.59%
AdaDelta	0.2405	93.54%	0.2699	92.66%
AbsDelta	0.2409	93.51%	0.2719	92.63%
$\ell_{1-4}-$**AdaDelta**	0.2392	93.58%	0.2711	92.63%
Adam	0.2189	94.03%	0.2772	92.65%
AbsAdam	0.2213	93.97%	0.2775	92.64%
$\ell_{1-4}-$**Adam**	**0.2143**	**94.12%**	0.3111	92.51%
ExpAdam	0.2161	94.06%	0.3261	92.46%
$\ell_{ada}-$**Adam**	0.2147	94.06%	0.3065	92.43%
RMSprop	0.2214	94.01%	0.2824	92.69%
Absprop	0.2235	93.95%	0.2766	92.68%
$\ell_{1-4}-$**RMSprop**	0.2188	94.08%	0.3047	92.56%
AMSGrad	0.2204	94.01%	0.2771	92.68%
AbsAMSGrad	0.2228	93.96%	0.2758	92.66%
$\ell_{1-4}-$**AMSGrad**	0.2147	94.11%	0.3065	92.49%
Nadam	0.2188	94.04%	0.2760	92.65%
AbsNadam	0.2215	93.98%	0.2762	92.64%
$\ell_{1-4}-$**Nadam**	**0.2143**	94.11%	0.2973	92.52%
NAMSGrad	0.2203	94.02%	0.2759	92.68%
AbsNAMSGrad	0.2217	93.98%	0.2776	92.62%
$\ell_{1-4}-$**NAMSGrad**	0.2150	94.10%	0.2964	92.50%
AdaMax	0.2246	93.91%	0.2763	92.65%
NadaMax	0.2245	93.92%	0.2758	92.63%

TABLE 6.1: Results of all algorithms on the MNIST data set using logistic regression. For both the training set and test set the minimal cost and maximal accuracy (after 250 iterations over the entire training set) averaged over 5 runs with randomly shuffled data is displayed, and per column the best value is highlighted.

6.1.2 Results (F)

The obtained results are displayed in Table 6.1. The table shows for every algorithm the minimum training cost reached, the maximum training accuracy, the minimum test cost, and the maximum test accuracy. These values are all averages over the 5 runs per algorithm.
In terms of training cost, the $\ell_{1-4}-$ versions of Adam and Nadam scored the best results, while the former also achieved the best result in training accuracy. All SGD versions, all AdaDelta versions, and Absgrad together make up the group of worst performing algorithms in terms of training cost. The latter had the worst score of all algorithms for both training cost and accuracy. The new algorithm SGDHB performed on a similar level as standard SGD and worse than SGDmom and SGDNAG. When for each adaptive learning rate algorithm the different versions are compared on training cost, it can be concluded that in six out of seven cases the $\ell_{1-4}-$version performed best. Only for Adagrad the standard version reached the lowest training cost. Moreover, the Exp and $\ell_{ada}-$versions of Adam performed better than the standard version. The Abs version on the other hand performed in terms of training cost for all algorithms worst of all versions.
When the results for the minimum test cost reached are analyzed, it turns out that the

algorithm worst performing on training cost (Absgrad), is the best performing algorithm on test cost. Moreover, the worst performing algorithms are the $\ell_{1-4}-$versions of the algorithms, as well as the Exp and $\ell_{ada}-$versions of Adam. All SGD versions perform on a similar level as the average adaptive learning rate algorithm, and standard SGD as well as SGDHB outperform the SGD version with momentum and SGDNAG.

Comparing the versions of the adaptive learning rate algorithms on test cost, it turns out that in four of the seven cases the standard version had the best result, and in the other three cases the Abs version reached the lowest test cost. The results indicate that, in general, the algorithms that achieve lower values on the training cost are among the highest scores on the test cost and vice versa. For example Absgrad, which has the highest training cost, reached the absolute lowest test cost among all algorithms. The plots in Figure D.5 and Figure D.6 in Appendix D show that in general the training cost keeps decreasing, seemingly approaching a convergence point at which it cannot improve anymore. The plots of the test cost on the other hand show a rapid decrease at the start, but after a couple of iterations the cost slowly increases again. The minimum test cost is in this case therefore in general reached after only a couple of (tens of) iterations, while the minimum training cost is almost always reached in the last iteration. The Adagrad and SGD algorithms seem to keep the low test cost until the end (see Figure D.6c and Figure D.6k), where the former also has not much improvement in training cost (see Figure D.5c) but the latter has (Figure D.5k).

6.2 Logistic regression on CIFAR10

Similarly to the logistic regression experiment on the MNIST data set described above, this experiment is conducted to find out the basic optimization power of the various algorithms in a classifier, but now on the classification of images from the CIFAR10 data set. This is a more difficult task, since the images are more complicated and bigger, which is caused by the three channels in the RGB image format. A selection of 14 optimization algorithms is tested in this experiment.

6.2.1 Experimental setup

The input of the linear network consists of 32x32 RGB images. This means that there are $32 \cdot 32 \cdot 3 = 3072$ input nodes since RGB images have three input channels. There are images from ten classes, so the network has ten output nodes.

The algorithms are optimized by selecting the parameter setting that reached the lowest training cost. This was found out by taking the average score over three runs, where each run consisted of five iterations over the entire training set. The data was shuffled before every iteration, and a batch size of 128 was used.

For the experiment, each optimization algorithm is tested on 5 runs, where each run lasted 20 iterations over the entire training set. Again, the training data is shuffled every iteration and a batch size of 128 is used. Since the number of instances in the data set is quite high, the average cost and accuracy are calculated and stored every 20 mini-batches instead of every iteration. Since every time that data is stored the performance on the test set is determined, multiple of these performance measures can be done in this way. The same is done with the current cost and accuracy on the test data. This provides the plots with $((50000/128)/20) * 20 = 391$ data points per run. However, for the plots of the training and test cost average, average values are

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	1.6613	44.21%	**1.7365**	**40.56%**
SGDmom	1.6532	44.43%	1.7463	40.12%
SGDNAG	1.6531	**44.53%**	1.7442	40.18%
SGDHB	1.6602	44.23%	1.7409	40.35%
Adam	1.6577	44.32%	1.7406	40.36%
AbsAdam	1.6561	44.35%	1.7422	40.28%
ℓ_{1-4}**−Adam**	**1.6494**	44.50%	1.7509	39.96%
ExpAdam	1.6530	44.46%	1.7421	40.29%
ℓ_{ada}**−Adam**	1.6505	44.52%	1.7494	39.95%
Nadam	1.6509	44.51%	1.7487	40.01%
AMSGrad	1.6568	44.36%	1.7401	40.38%
AbsAMSGrad	1.6556	44.36%	1.7422	40.26%
ℓ_{1-4}**−AMSGrad**	1.6497	**44.53%**	1.7482	40.07%
AdaMax	1.6560	44.44%	1.7437	40.21%

TABLE 6.2: Results of all considered algorithms on the CIFAR-10 data set using logistic regression. For both the training set and test set the minimal cost and maximal accuracy on the entire set (after 20 iterations over the entire training set) averaged over 5 runs with randomly shuffled data is displayed, and per column the best value is highlighted.

calculated for entire iterations over the training set, to obtain a more representative outcome.

6.2.2 Results (G)

The results of the optimization algorithms on the experiment with logistic regression on the CIFAR10 data set can be seen in Table 6.2.
In terms of training cost the ℓ_{1-4} version of Adam again has the best result, directly followed by the ℓ_{1-4} version of AMSGrad, the ℓ_{ada} version of Adam, and standard Nadam. Among the worst performing algorithms are standard SGD and SGD with handbrake. Furthermore, it turns out that all new versions of Adam outperform the standard version in terms of minimum training cost reached.
The results of the minimum test cost reached show that standard SGD has the best performance here, at some distance followed by SGDHB and standard AMSGrad and Adam. The ℓ_{1-4} and ℓ_{ada} versions of Adam as well as standard Nadam form the group of worst performing algorithms on the test cost. It seems again that algorithms with a better score on the training set reach a worse score on the test set and vice versa. Furthermore, the plots in Figure E.1 and Figure E.2 show a similar course of the training cost and test cost as in the experiment with logistic regression on the MNIST data set. The plots are zoomed in to be able to see the different lines. This was needed due to the enormous drop in both test and training cost that occurred in the first few iterations. After that the training cost keeps decreasing, but the test cost starts to converge after about eight iterations over the training set. A difference from the previous experiment is that the plots do not show a significant increase of test cost after convergence occurs.

Chapter 7

Experiments on a multi-layer perceptron

While in the previous chapter the optimization algorithms were tested on linear networks in a logistic regression setting, in this chapter they will be tested on several multi-layer perceptrons (MLPs). A total of six different experiments are described in this chapter.

7.1 MLP with 1 hidden layer on MNIST

In the first experiment, a selection of the optimization algorithms is tested on the classification of the MNIST data set using a small neural network. This network consists of the 784 input nodes, followed by 100 hidden nodes, and 10 output nodes. The ReLU activation function is used in between the layers, and as loss function cross-entropy is used. It is expected that this network has better results than the linear network in the logistic regression experiment, due to the extra hidden layer which provides multiple extra weights to optimize.

7.1.1 Experimental setup

In this experiment, the 12 selected algorithms are tested, as well as AbsAMSGrad and ℓ_{1-4}-AMSGrad. These two have been added to find out if the performance difference between standard Adam and AMSGrad transits to the new versions of the algorithms.
The optimal parameter setting for each algorithm is determined by running each algorithm five times for 40 iterations on the MNIST data set. The setting that yielded on average the smallest training cost is chosen as the optimal setting. The final parameter settings for all algorithms can be found in Table A.5.
In the experiment itself, the optimization algorithms are tested during 5 runs on the MNIST data set. Each run consists of 100 iterations over the training set. A batch size of 128 is used. After each full iteration, the average cost and accuracy on the training set are calculated, and the performance on the test set is determined.

7.1.2 Results (H)

The results of this experiment are displayed in Table 7.1. The table shows the minimum cost and maximum accuracy reached on the test set and the training set. These values are all calculated from the average result over the 5 runs. Moreover, since most algorithms were able to achieve 100% accuracy on the training set, a result is added to the table indicating how many iterations over the training set were needed

Algorithm	Min training cost*	Max training accuracy	Avg to 100% (st.dev.)	Min test cost	Max test accuracy
SGD	0.000254	100.00%	24.80(0.75)	0.0732	98.08%
SGDmom	0.000133	100.00%	19.80(0.75)	0.0803	**98.17%**
SGDNAG	0.000112	100.00%	19.40(1.50)	0.0793	98.13%
SGDHB	0.000130	100.00%	26.80(0.75)	0.1018	97.99%
Adam	0.000104	100.00%	36.20(3.97)	0.0813	97.94%
AdaMax	**0.000003**	100.00%	26.20(1.47)	0.0761	98.10%
Nadam	0.000015	100.00%	37.00(5.83)	0.0784	97.92%
AMSGrad	0.000107	100.00%	17.40(0.49)	0.0848	98.02%
AbsAdam	0.000014	100.00%	31.80(0.98)	**0.0719**	98.01%
$\ell_{1-4}-$**Adam**	0.000040	100.00%	35.80(1.60)	0.0732	97.94%
ExpAdam	0.010232	99.73%	-(-)	0.0872	97.55%
$\ell_{ada}-$**Adam**	0.000030	100.00%	29.20(1.33)	0.0760	97.99%
AbsAMSGrad	0.000072	100.00%	16.60(0.49)	0.0852	98.07%
$\ell_{1-4}-$**AMSGrad**	0.000050	100.00%	**16.40(1.36)**	0.0858	98.06%

TABLE 7.1: Results of all algorithms on the MNIST data set using an MLP with 1 hidden layer of 100 nodes. For both the training set and test set the minimal cost and maximal accuracy (after 200 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted. Moreover, for the algorithms that on average reached a training accuracy of 100%, it is indicated after how many training iterations this happened.

to reach this accuracy level. This number is also the average over 5 runs, and therefore also the standard deviation is included, which indicates the difference in training speed between the 5 runs for each algorithm.
The results in the table show that all algorithms managed to get a training accuracy of 100%, except for ExpAdam that only reached 99.73%. This is also the only algorithm with a minimum training cost reached above 0.0003, namely slightly over 0.01. The other algorithms have a very small minimum training cost, and they lie quite close to each other. One group has a minimum slightly over 0.0001, while the other group has an even smaller minimum training cost. AdaMax managed to get the smallest value with 0.000003.
The division in groups of the algorithms on the training cost reached, cannot be used on the number of iterations to reach 100% accuracy on that training set. ExpAdam has the highest minimum training cost and is the only algorithm that did not reach 100%. Some members of the group of algorithms with a minimum cost slightly over 0.01 reached 100% accuracy after about 30-40 iterations, but some also already within less than 20 iterations. Moreover, the group of algorithms with the smallest training cost includes also algorithms that needed 30-40 iterations as well as algorithms that needed less than 20 iterations. Most algorithms have a small standard deviation for this statistic, only standard Adam and Nadam have a standard deviation (much) higher than 2. The fastest state of the art algorithm turns out to be AMSGrad, so the Abs and ℓ_{1-4} versions of that algorithm are also tested. It turns out that both new versions outperform the standard AMSGrad algorithm, both in terms of minimum training cost reached and number of iterations to reach 100%. This is also the case for the Adam algorithm, where also the ℓ_{ada} version outperformed the standard version.
The plots in Figure D.1 show for various algorithm groups the average training cost per algorithm for each iteration (over 5 runs). There seem to be two groups of algorithms: those with a smoothly decreasing line, and those with a lot of ups and downs

in their lines. This last group is formed only by adaptive learning rate algorithms, namely Adam, Nadam, AbsAdam, ExpAdam, and ℓ_{ada}-Adam. Figure D.1a shows that ℓ_{1-4}-AMSGrad is the fastest algorithm to reach convergence and that ExpAdam is not even close to the other algorithms. In the first 15 iterations it keeps up with the others, but from that moment it seems that it is not able to improve at all.
In terms of test cost and accuracy reached, most algorithms were able to reach a cost between 0.07 and 0.09 (except for SGDHB), and an accuracy between 97.5% and 98.2%. This means that there seems to be no significant difference in performance on the test set between any of the tested algorithms. AdaMax had the best test cost from the state of the art algorithms, while SGD with standard momentum reached the highest accuracy. All new versions of Adam, except for the Exp version, were able to outperform the standard version of Adam. However, the Abs and ℓ_{1-4} versions of AMSGrad had a slightly worse minimum test cost but a slightly better test accuracy.
Figure D.2 shows for some algorithm groups for every algorithm the average test cost per iteration (over 5 runs). Similarly to the plots in Figure D.6, the curves show a rapid decrease in the first couple of iterations (this time 5-10 iterations), after which a gradual increase happens. All SGD versions and all AMSGrad versions were able to minimize this decrease, while the other state of the art algorithms and ExpAdam have a quite rapid increase in test cost. However, each new Adam version outperforms its AMSGrad counterpart in terms of minimum test cost reached, as can be seen in Figure D.2b and Figure D.2f, and these two also outperform standard Adam as can be seen in Figure D.2d.

7.2 MLP with 1 hidden layer and batch normalization on MNIST

The next experiment is on a neural network very similar to the one used in the previous experiment, so with a hidden layer of 100 nodes, the ReLU activation function, and the cross-entropy loss function. The only difference is the addition of a batch normalization layer. This promises to speed up the learning process of the previous experiment significantly. Batch normalization is applied to the output of the hidden layer. The MNIST data set is again used as the test case.

7.2.1 Experimental setup

In this experiment, only the 12 optimization algorithms from the selected group are tested. The optimal parameters for each algorithm are again determined by running several different parameter settings for each algorithm over three runs. Each run lasts 40 iterations over the training set with a batch size of 128. The parameter setting with the on average lowest cost reached is chosen to be the optimal one. The final parameter settings can be found in Table A.6.
The experiment itself is conducted by running each algorithm for 5 runs. Each run lasts 100 iterations over the training set with a batch size of 128. After every iteration the average training cost is calculated and the performance of the current network on the test set is determined.

7.2.2 Results (I)

The results of this experiment are displayed in Table 7.2. The table shows again for each algorithm the minimal average training and test cost reached, as well as

Algorithm	**Min training cost***	**Max training accuracy**	**Avg to 100% (st.dev.)**	**Min test cost**	**Max test accuracy**
SGD	0.00087	99.99%	-	0.0746	97.92%
SGDmom	0.00126	99.97%	-	0.0820	97.91%
SGDNAG	0.00058	99.99%	-	0.0740	97.96%
SGDHB	0.00132	99.97%	-	0.0809	97.72%
Adam	0.00078	99.99%	-	0.0748	97.96%
AdaMax	**0.00028**	99.99%	-	**0.0713**	97.97%
Nadam	0.00075	99.99%	-	0.0761	97.82%
AMSGrad	0.00049	**100.00%**	**57.20(6.82)**	0.0746	**98.03%**
AbsAdam	0.00123	99.97%	-	0.0754	98.00%
$\ell_{1-4}-$**Adam**	0.00047	**100.00%**	82.60(9.99)	0.0759	97.85%
ExpAdam	0.00207	99.94%	-	0.0796	97.89%
$\ell_{ada}-$**Adam**	0.00047	**100.00%**	67.20(4.66)	0.0758	97.89%

TABLE 7.2: Results of all algorithms on the MNIST data set using an MLP with 1 hidden layer of 100 nodes and batch normalization. For both the training set and test set the minimal cost and maximal accuracy (after 200 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted. Moreover, for the algorithms that reached a training accuracy of 100%, it is indicated after how many training iterations this happened.

the maximum average accuracy on the training and test set. Since again some algorithms were able to reach a perfect score on the training set, a column is added indicating the average number of iterations to reach this 100% accuracy and its standard deviation.

The results show that, although all of the algorithms managed to reach a training accuracy above 99.90%, only three algorithms reached the perfect score of 100%. However, the algorithm with the lowest minimum training cost reached, which is just like in the previous experiment AdaMax, is not among these three algorithms. These are AMSGrad, ℓ_{1-4}-Adam and ℓ_{ada}-Adam, and so these latter two have outperformed their standard version on the training set. The Abs and Exp versions however had slightly worse results, with ExpAdam being the overall worst performing algorithm on the training set. From the three algorithms that reached the perfect score, AMSGrad turned out to be the fastest to converge, with on average slightly over 57 iterations needed. ℓ_{ada}-Adam has the second least amount of iterations needed: about 10 more, while ℓ_{1-4}-Adam needs about 83 iterations. The standard deviations are quite high, much higher than in the previous experiment. The minimum training cost and average iterations needed to reach 100% accuracy are also higher, while the reached maximum accuracy is often lower. This means that the addition of the batch normalization layer was in this case in terms of the performance on the training set only beneficial for the ExpAdam algorithm, since this algorithm reached better scores than in the previous experiment.

The influence of the addition of the batch normalization layer on the reached test cost and accuracy is positive for some algorithms, mostly the state of the art adaptive learning rate algorithms, but negative for the other algorithms, including the new versions and the SGD algorithms. However, the scores have not changed very much. AdaMax, which also reached the lowest training score, reached the lowest test score. AMSGrad had the highest test accuracy. The SGD algorithm with handbrake did outperform SGD with standard momentum on the test cost but not on the test accuracy, and had a worse score than the other two SGD algorithms. All new versions of the Adam algorithm were outperformed by the standard version of

Adam in test cost, but AbsAdam managed to reach a slightly higher test accuracy.
The plots in Figure D.3 show for several algorithm groups the average training cost reached per iteration for each algorithm. All plots are zoomed in, such that the differences between the various algorithms are better visible. However, in this case they all have a very similar performance. The various waves in the lines are therefore mainly caused by the far zoom in.
The plots show that AMSGrad is fastest in the first part of the learning process, but after about 70 iterations over the training set it starts to fluctuate more, leading to an increase in training cost. In that phase AdaMax takes over the lead and keeps it until the end. Looking at Figure D.3c it can be seen that ExpAdam is the worst performing version of Adam during the entire run. ℓ_{1-4}-Adam and ℓ_{ada}-Adam seem to take turns in having the lowest training cost. Adam and AbsAdam are in a similar situation, but with a slightly higher cost than the two best-performing algorithms.
Figure D.3d shows the averaged performances of the SGD algorithms. The new SGD algorithm with handbrake momentum seems to be behind for the biggest part of the run, and only at the end it more or less matches the training cost of SGD with momentum. SGD with Nesterov momentum is the best performing algorithm in the first 60 iterations, but then its cost suddenly increases. However, in the end it seems to recover and reaches the lowest training cost of all SGD algorithms. At the very end it seems to give away the lead to standard SGD.
Figure D.4 shows for various algorithm groups the average test cost reached per iteration for each algorithm. These plots have been zoomed in slightly, except for Figure D.4d which has no zoom.
The shapes of the lines are quite similar to each other, and also like the ones in the previous experiments: a big decrease in the first 10 iterations followed by a gradual increase for the rest of the run. However, there seems to be no algorithm that manages to more or less restraint the increase.
At the minimum point of the test cost, it is clearly AdaMax which has the best result, while SGDmom seems to reach the highest minimal test cost. This latter algorithm also has the highest final cost, while AbsAdam, SGD, and AMSGrad have the lowest final test cost. Figure D.4c shows that ExpAdam performs again worse than all other Adam versions during almost the entire run. AbsAdam seems to have the best performance during the phase with the big decrease of test cost, however it is outperformed by Adam in reaching the lowest test cost. At the end of the run AbsAdam regains the smallest test cost, and it also seems that ℓ_{ada}-Adam can match this performance at the end. However, during the first couple of iterations in the run it seems that this algorithm might be the worst performing algorithm together with ExpAdam. In Figure D.4d it can be seen that standard SGD has the best performance during almost the entire run, but SGD with Nesterov momentum reaches the lowest minimum test cost. SGD with handbrake momentum performs worse than both but better than SGD with standard momentum.

7.3 MLP with 1 hidden layer on CIFAR10

This experiment is similar to the first experiment described in this section, but now the CIFAR10 data set has been used. Since it is more difficult to learn to classify this data than the MNIST data set, mainly due to the bigger input caused by the RGB type of the input images, the hidden layer in the neural network has 200 nodes instead of 100.

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	0.632	79.11%	1.419	51.26%
SGDmom	0.527	82.24%	1.429	51.19%
SGDNAG	0.447	84.70%	1.413	51.53%
SGDHB	0.587	80.22%	1.428	51.08%
Adam	0.415	86.09%	1.415	51.82%
AdaMax	0.407	86.43%	1.404	51.94%
Nadam	0.353	88.18%	1.411	51.98%
AMSGrad	0.402	86.75%	1.410	51.87%
AbsAdam	0.474	83.99%	1.425	51.54%
ℓ_{1-4}**−Adam**	0.417	85.82%	1.427	51.57%
ExpAdam	0.429	85.90%	1.405	51.81%
ℓ_{ada}**−Adam**	0.404	86.45%	1.414	51.90%
AbsNadam	0.397	86.50%	1.414	51.58%
ℓ_{1-4}**−Nadam**	0.351	88.21%	1.405	**52.10%**
ℓ_{ada}**−Nadam**	**0.349**	**88.32%**	**1.402**	51.96%
AbsAMSGrad	0.469	84.17%	1.421	51.45%
ℓ_{1-4}**−AMSGrad**	0.384	87.20%	1.417	51.79%

TABLE 7.3: Results of all algorithms on the CIFAR10 data set using an MLP with 1 hidden layer of 200 nodes. For both the training set and test set the minimal cost and maximal accuracy (after 40 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

7.3.1 Experimental setup

Apart from the algorithms in the selected group, this experiment is conducted on the Abs and ℓ_{1-4} versions of AMSGrad and Nadam and the ℓ_{ada} version of Nadam as well. These are included to see if the difference between Adam and its versions also occurs between AMSGrad and Nadam and their versions.
The parameters of all algorithms are optimized by running for each algorithm several parameter settings. Each setting is tested in 3 runs, and the reached cost after 5 full iterations over the training set, with a batch size of 128, is taken as the selection criterion. The setting with the smallest average training cost reached is taken as the optimal one. The final parameter settings for all algorithms are displayed in Table A.9.
The performance of all optimized algorithms on the CIFAR10 data set using a neural network with a single hidden layer is measured by testing each algorithm for 5 runs. Each run lasts 40 iterations with a batch size of 128. Every 20 mini-batches the current average training cost and accuracy (over the mini-batch) are stored, and the performance on the entire test set is determined.

7.3.2 Results (J)

The results are displayed in Table 7.3. The table shows for every tested algorithm the minimum average cost and maximum accuracy reached on the training set and on the test set. It can be seen that all algorithms reach a much lower training cost than test cost, and also the accuracy on the former is much better than on the latter.
There seems to be quite a difference between the algorithms in reached minimum average training cost. The ℓ_{ada} version of Nadam reaches the smallest value, immediately followed by the ℓ_{1-4} and standard versions of Nadam. This same result applies to the maximum training accuracy reached. The worst algorithms in terms of training cost are the SGD algorithms and the Abs versions of AMSGrad and Adam (AbsNadam performs better than most of the other algorithms, but worst of

all Nadam versions). SGDHB does a better job than standard SGD, but far worse than SGDNAG and SGDmom. ExpAdam also has a worse performance than standard Adam, but still better than AbsAdam. So only the new ℓ_{ada} and ℓ_{1-4} versions were all (except ℓ_{1-4}-Adam) able to improve the performance of their standard versions on the training set.
The plots in Figure E.3 show for groups of algorithms the average training cost over the course of the learning process in the 5 runs. It can clearly be seen that the standard Nadam along with the ℓ_{1-4} and ℓ_{ada} versions of Nadam outperform the other algorithms during the entire run. Comparing this difference to the difference between these three versions of Nadam, it seems that the latter is much smaller. The various plots show that the difference in performance between standard Nadam and AMSGrad/Adam seems to transit to the new versions. Figure E.3i shows that during the runs the order in the performance of the SGD algorithms remains the same.
Looking at the test scores in Table 7.3, it seems that the scores from different algorithms lie much closer to each other than for the training scores: all algorithms reach a minimum average cost between 1.40 and 1.43 and a maximum accuracy between 51.0% and 52.1%. ℓ_{ada}-Nadam had the lowest test cost reached (and therefore has the best score in 3 of the 4 statistics), while the ℓ_{1-4} version of Nadam had the highest accuracy on the test set. SGDHB managed to outperform SGDmom on the minimum test cost but reached a worse score than the other two SGD algorithms. For the new versions of Adam, only ℓ_{1-4}- and ExpAdam did better than the standard, in the group of AMSGrad algorithms neither did better than the standard, but for the Nadam algorithms only Abs did not outperform the standard.
Figure E.4 shows the plots of the average test cost during the course of the runs. Again the typical shape of the lines in the plot can be seen: a big decrease of test cost, followed by a gradual increase. The difference between the algorithms in the valley of the plot is not very clear. However, it can be seen that after the valley the ℓ_{1-4} version of Adam is worst in preventing the cost from increasing (at the end together with standard Nadam), while standard SGD does the best job here. From the new adaptive learning rate versions, ExpAdam seems to have the best result. This is a result similar to some cases in the logistic regression experiments, since ExpAdam as well as SGD are among the worst performing algorithms on the training set.

7.4 MLP with 1 hidden layer and batch normalization on CIFAR10

Similarly to the experiments with a neural network with a single hidden layer on the MNIST data set, the addition of batch normalization to the neural network with a single hidden layer for the classification of data from the CIFAR10 data set should increase the performance of the learning system.

7.4.1 Experimental setup

In this experiment, only the algorithms from the selected group have been tested. The parameters of the algorithms are optimized with exactly the same method as in the experiment on the neural network with a single hidden layer and without batch normalization. The optimal parameter settings for the algorithms can be reviewed in Table A.10.
The experiment itself is also conducted in the exact same fashion as the experiment

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	0.622	78.89%	1.396	51.86%
SGDmom	0.554	81.01%	1.392	51.89%
SGDNAG	0.507	82.53%	1.388	52.03%
SGDHB	0.583	80.13%	1.383	52.07%
Adam	0.475	83.85%	1.376	52.40%
AdaMax	0.474	83.99%	**1.361**	**52.66%**
Nadam	**0.432**	**85.25%**	1.370	52.56%
AMSGrad	0.474	83.91%	1.365	52.64%
AbsAdam	0.515	82.32%	1.379	52.21%
ℓ_{1-4}**−Adam**	0.474	83.90%	1.375	52.32%
ExpAdam	0.683	76.63%	1.384	51.73%
ℓ_{ada}**−Adam**	0.481	83.54%	1.375	52.38%

TABLE 7.4: Results of all algorithms on the CIFAR10 data set using an MLP with 1 hidden layer of 200 nodes and batch normalization. For both the training set and test set the minimal cost and maximal accuracy (after 40 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

without batch normalization. So 5 runs per algorithm are conducted, with 40 iterations per run and a batch size of 128. Every 20 mini-batches the average training cost and accuracy are calculated and the performance on the test set is determined.

7.4.2 Results (K)

The results of this experiment can be seen in Table 7.4. Again the minimum training cost is for all algorithms much lower than the minimum test cost, and the maximum training accuracy is much higher than the maximum test accuracy. However, compared to the results of the experiments on the neural network with 1 hidden layer without batch normalization, the training scores are worse in this experiment while the test scores are slightly better.
The training costs show some big differences between algorithms. Nadam had the lowest average training cost and the highest training accuracy, both with quite a distance to the other algorithms. The standard versions of Adam, AdaMax, and AMSGrad, as well as ℓ_{1-4}-Adam, have the closest minimum average training cost, immediately followed by ℓ_{ada}-Adam. AbsAdam and SGD with Nesterov momentum are slightly worse, but the worst performing algorithms are again the other SGD versions (including SGD with handbrake) and ExpAdam, which is the overall worst performing algorithm with almost 1.6 times as high minimum average training cost as Nadam.
Plots of the average training cost over the course of the runs are displayed in Figure E.5. It can be seen that Nadam has the lowest average training cost during the entire run, while ExpAdam has the highest one. Apart from this latter algorithm and AbsAdam, which perform worse than Adam, there is no clear difference in performance visible between the standard Adam algorithm and the new versions. Figure E.5d shows that SGD with handbrake has the usual performance: better than standard SGD, but worse than the other two SGD versions.
The minimum test costs reached by the algorithms are also displayed in Table 7.4. The values of the different algorithms are quite similar, between 1.36 and 1.4. AdaMax had the best result, followed by AMSGrad. The worst results are from the SGD versions and ExpAdam, but SGD with handbrake was able to outperform the other SGD versions this time. The standard Adam version was slightly outperformed by

both the ℓ_{1-4} and the ℓ_{ada} version.
Figure E.4 shows for the different algorithm groups the average test cost per iteration during the 5 runs. The plots are all zoomed in on the part of the plot with the valley. The lines in the full plots were again of the same typical shape encountered in all plots of test cost so far. The lines of the different algorithms are very close to each other, but it can be seen that AdaMax reaches the lowest value. Moreover, it seems that ExpAdam reached the valley in its curve somewhat later than the other algorithms did. Similar to the plots of the training cost of this experiment, in Figure E.6c no clear difference between standard Adam and the ℓ_{1-4} and ℓ_{ada} versions is visible. AbsAdam and ExpAdam however have a somewhat higher minimum average test cost. Figure E.6d shows that SGDHB, although a bit slower than the other algorithms, managed to reach a lower average value on the test cost than the other SGD algorithms.

7.5 MLP with 3 hidden layers on CIFAR10

The neural networks tested on the CIFAR10 data set so far were not able to reach a very high accuracy on the test set, 52% at the highest. In the experiments with neural networks on the MNIST data set however the algorithms reached accuracies far above 90%. This level of performance is not expected on CIFAR10, since already a lot of systems have been trained on it and only much more complicated ones were able to reach such a performance. However, the performance could in theory be better with some additional hidden layers in the neural network. Therefore, in this experiment a network with 3 hidden layers is used to see what the difference in performance would be.

7.5.1 Experimental setup

The neural network has $32 \cdot 32 \cdot 3 = 3072$ input nodes, followed by a first hidden layer of 500 nodes. This layer is followed by a second hidden layer of 200 nodes and a third hidden layer of 50 nodes. Finally, there is the output layer with 10 nodes for the 10 different classes. In every layer, the ReLU activation function is used. As can be seen, the deeper in the network the smaller the size of the layer becomes. This is chosen with the use of the rule of thumb *"It is good to have the size of the hidden layers between the size of the input layer and output layer."* (Russell and Norvig, 2016).
The algorithms tested in this experiment are only the ones from the selected group. They have all been optimized in a similar way as in the previous experiments. Every considered parameter setting is tested for 3 runs of each 5 iterations with a batch size of 128, and the setting with the smallest training cost reached is chosen to be the optimal one. These optimal parameter settings are displayed in Table A.13.
The experiment itself is also conducted in a similar manner as the previous ones. Each algorithm is tested for 5 runs each lasting 40 iterations with a batch size of 128. Every 20 mini-batches the average training cost and accuracy over these mini-batches are calculated and the current performance on the test set is determined.

7.5.2 Results (L)

The results of this experiment can be seen in Table 7.5. In terms of training cost, it is clear that the new SGD algorithm with handbrake is by far the worst performing algorithm. The other algorithms seem to be divided into two groups for the training cost: the other SGD algorithms plus AbsAdam and ExpAdam reach a training cost

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	0.268	90.80%	1.392	52.95%
SGDmom	0.220	92.46%	1.370	53.42%
SGDNAG	0.234	92.07%	1.372	53.48%
SGDHB	0.528	82.59%	1.397	52.82%
Adam	0.131	95.49%	1.353	54.21%
AdaMax	**0.087**	**97.10%**	1.359	54.04%
Nadam	0.131	95.61%	1.345	54.37%
AMSGrad	0.128	95.64%	1.360	54.09%
AbsAdam	0.208	92.86%	1.367	53.82%
ℓ_{1-4}**−Adam**	0.115	96.08%	1.349	**54.50%**
ExpAdam	0.180	94.19%	**1.341**	53.98%
ℓ_{ada}**−Adam**	0.112	96.19%	1.347	54.30%

TABLE 7.5: Results of all algorithms on the CIFAR10 data set using an MLP with 3 hidden layers. For both the training set and test set the minimal cost and maximal accuracy (after 40 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

between 0.18 and 0.25, and the other algorithms (all adaptive learning rate ones) manage to get a value smaller than 0.14, with AdaMax having the very best score of 0.087. The standard Adam algorithm was outperformed on the reached training cost by both the new ℓ_{1-4} variant and the new ℓ_{ada} version.
Plots of the algorithms grouped together about the average training cost reached per iteration over the training set are displayed in Figure E.7. The line of the SGD algorithm with handbrake seems to have a somewhat different course than the other lines. It goes along with the rest until about the 20th iteration, but after this point it reaches a sort of valley from which it also goes up again at the end of the runs. The rest of the algorithms are clearly grouped together as described above, although ExpAdam did seem to make more progress during the second part of the runs than the other algorithms. Moreover, it seems that in the first half of the runs Nadam reached the lowest average training cost, but it was overtaken by AdaMax which then took the lead. In the end, Nadam was also outrun by the ℓ_{1-4} and ℓ_{ada} versions of Adam.
The minimum test costs reached for the algorithms are closer to each other than the minimum training costs. However, SGD with handbrake had also in this case the worst score of 1.397, closely followed by standard SGD with 1.392. All SGD algorithms performed worse than all adaptive learning rate algorithms. ExpAdam was able to reach the lowest minimum test cost with 1.341, but ℓ_{1-4}-Adam took the highest test accuracy. These two versions along with ℓ_{ada}-Adam all outperformed standard Adam on the minimum test cost reached.
The plots in Figure E.8 show the average test cost reached per iteration over the 5 runs. Note that these plots are all zoomed in, fixated on the part of the plot with the valley. This can also be seen by the values on the x-axis, which show that a big part on the right side of the plot has been taken out.
It can clearly be seen that, probably due to the described lag in the training cost of ExpAdam with respect to the other algorithms (see Figure E.7), ExpAdam reaches its minimum test cost two to four iterations over the training set later than the other algorithms. However, it manages to reach a lower minimum value than the others. SGD and SGDHB both seem to have the worst performance. In Figure E.8d it can be seen that SGDNAG and SGDmom reach a more or less equivalent minimum test cost. Figure E.8c shows that standard Adam has only a better result than the Abs

Algorithm	**Min training cost***	**Max training acc**	**Min test cost**	**Max test acc**
SGD	0.178	93.81%	1.337	53.74%
SGDmom	0.168	94.05%	1.334	53.95%
SGDNAG	0.156	94.48%	1.334	54.01%
SGDHB	0.196	93.13%	1.341	53.75%
Adam	0.138	95.31%	1.314	54.34%
AdaMax	0.123	95.74%	1.317	54.35%
Nadam	0.147	94.83%	1.317	**54.63%**
AMSGrad	**0.117**	**95.95%**	1.314	54.55%
AbsAdam	0.155	94.64%	1.327	54.01%
ℓ_{1-4}**−Adam**	0.134	95.45%	1.317	54.21%
ExpAdam	0.145	94.92%	1.328	54.26%
ℓ_{ada}**−Adam**	0.137	95.31%	**1.312**	54.43%

TABLE 7.6: Results of all algorithms on the CIFAR10 data set using an MLP with 3 hidden layers and batch normalization. For both the training set and test set the minimal cost and maximal accuracy (after 40 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

version. Moreover, it can be seen that the better performing versions reach their minima (slightly) later than the standard version.

7.6 MLP with 3 hidden layers and batch normalization on CIFAR10

The addition of batch normalization to the network with 1 hidden layer led to a slightly higher performance on the test set, but a worse minimum cost and maximum accuracy on the training set. In this experiment, it is tried to find out if this effect also occurs for the neural network with 3 hidden layers.

7.6.1 Experimental setup

The general architecture of the neural network in this experiment is the same as in the previous one, but now batch normalization layers are added: one to the output of the first hidden layer and one to the output of the second hidden layer. For this experiment, only the algorithms in the selected group have been tested. The algorithms have been optimized in the usual way: by taking the parameter setting that yielded on average over 3 runs the smallest training cost after 5 iterations over the training set with a batch size of 128. The obtained parameter settings for every algorithm are displayed in Table A.14.
The experiment itself is also conducted in the usual way. Every optimized algorithm is tested on 5 runs each lasting 40 iterations over the entire training set. A batch size of 128 is taken, and after every 20 mini-batches the average training cost and accuracy over those mini-batches are determined and the performance on the test set is calculated.

7.6.2 Results (M)

The results of this experiment can be seen in Table 7.6. The average training cost of all algorithms seems to be a bit lower than in the previous experiment, but the best result (now from AMSGrad) is much higher than the lowest average training cost

reached in the previous experiment. In terms of the test cost however, the scores seem to be somewhat lower and the lowest score (from ℓ_{ada}-Adam) is also smaller than the one in the previous experiment.
SGDHB again has the worst performance regarding the minimum average training cost reached, followed by the other SGD versions from which SGDNAG performed the best. This algorithm is only slightly worse than the worst adaptive learning rate algorithm, which is AbsAdam. The worst performing state of the art adaptive learning rate algorithm is Nadam. This is a bit of a surprise, since in the previous experiments this algorithm was among the best performing, if not the very best. The standard version of Adam was outperformed by both the ℓ_{ada} and ℓ_{1-4} versions. Plots of the average training cost per iteration over the 5 runs for each algorithm can be seen in Figure E.9. The plots are zoomed in on the last couple of iterations to make the difference in performance between the various algorithms more clearly visible. The SGD algorithms are clearly worst performing together with AbsAdam, and SGDHB is the very worst. Figure E.9c shows the various Adam algorithms. ExpAdam has a much higher cost than the other algorithms, but ℓ_{1-4}-Adam, ℓ_{ada}-Adam, and sometimes also standard Adam seem to take turns for the smallest average training cost.
The test costs of the algorithms are very close to each other: all between 1.31 and 1.341. The worst scores are again from the SGD algorithms, with SGDHB having the very worst. From the adaptive learning rate algorithms, the Exp and Abs versions of Adam have the highest minimum costs reached, and the other algorithms are very close to each other: 1.312 to 1.317. Standard Adam was only outperformed by the ℓ_{ada} version. Finally, Nadam reached the highest test accuracy. Plots of the average test cost for each iteration over the 5 runs are displayed in Figure E.10. These plots are again zoomed in on the part of the plot with the valley, since the lines again have the same typical shape as in the plots of the test cost in the previous experiments. It seems that in this case all algorithms reach their minimum value for the test cost at more or less the same iteration. These plots do therefore show the same result as the column of the Min test cost in Table 7.6: SGDHB has the worst score, and ℓ_{ada}-Adam reached the lowest cost on the test set.

Chapter 8

Convolutional neural network experiments

For the image data in the MNIST and CIFAR10 data sets a standard multi-layer perceptron can be a good classifier. However, a convolutional neural network (CNN) is almost always an even better classifier, due to the use of its kernels. In this chapter, three experiments are described each involving a CNN. The first experiment is on the classification of the MNIST data set, while the other two are on the CIFAR10 data set. For each experiment, the network architecture and the experimental setup are described, after which the obtained results are discussed.

8.1 CNN on MNIST

In the experiments with a neural network with 1 hidden layer on the MNIST data set, almost all algorithms were able to reach a perfect score (100% accuracy) on the training set, while having more than 95.0% accuracy on the test set. This means that there is not much room for improvement on the training set, but on the test set the CNN could have a better result. Moreover, since this research is in the first place about the performance of the newly developed optimization algorithms with respect to the state of the art ones, this experiment also gives an indication if the difference in performance in the experiments on MLPs also occurs when a CNN is used.

8.1.1 Network architecture

The convolutional neural network that is used in this experiment is adapted from `https://github.com/vinhkhuc/PyTorch-Mini-Tutorials`. The architecture of the network consists of multiple layers. The network starts with a convolutional layer of 5 kernels, each of size 5x5, followed by a max-pooling layer with a 2x2 window and a ReLU layer. After this a second convolutional layer is included, this time with 20 kernels of size 5x5. After this, a drop out layer is included with the default probability of 0.5. Again a max-pooling layer with a 2x2 window and a ReLU layer follow. After this, the 2D output is converted to 1D using a custom implementation of a "Flatten" module added to the PyTorch package. The obtained vector of 320 values goes through a hidden layer of 50 nodes, and after this the ReLU activation function is applied. Next up is another dropout layer with the default probability of 0.5, and finally the output layer.

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	0.0444	98.61%	0.0589	98.34%
SGDmom	0.0455	98.55%	0.0604	98.29%
SGDNAG	0.0405	98.69%	0.0565	98.41%
SGDHB	0.1386	96.30%	0.1190	96.73%
Adam	0.0410	98.70%	0.0562	98.44%
AdaMax	0.0410	98.74%	0.0555	**98.49%**
Nadam	0.0406	98.72%	0.0545	98.41%
AMSGrad	0.0470	98.50%	0.0607	98.31%
AbsAdam	0.0418	98.69%	0.0572	98.40%
ℓ_{1-4}**−Adam**	0.0424	98.67%	0.0581	98.41%
ExpAdam	0.0803	97.52%	0.0812	97.59%
ℓ_{ada}**−Adam**	**0.0395**	**98.75%**	**0.0544**	98.48%

TABLE 8.1: Results of all algorithms on the MNIST data set using a convolutional neural network with 2 convolutional layers (second with dropout) followed by a fully connected hidden layer of 50 nodes and dropout. For both the training set and test set the minimal cost and maximal accuracy (after 200 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

8.1.2 Experimental setup

In this experiment, only the optimization algorithms from the selected group are tested. Every algorithm is optimized by tuning its parameters. Several configurations per algorithm have been tested during 3 runs of each 40 iterations over the entire training set, using a batch size of 128. The parameter setting that reached the on average smallest training cost is chosen to be the optimal one. The obtained parameters are displayed in Table A.7.
The experiment is conducted by running each optimized algorithm over 5 runs. Each run lasted 100 iterations, and again a batch size of 128 is used. After every iteration, the average training cost and the training accuracy are calculated, and the current performance on the test set is determined.

8.1.3 Results (N)

The results of this experiment can be seen in Table 8.1. The algorithms seem to reach a bigger minimum average training cost and smaller training accuracy, but a smaller minimum average test cost and a higher test accuracy than in the experiments with the MLPs.
The ℓ_{ada}-Adam algorithm turns out to perform the best of all algorithms on the training cost and accuracy. It is the only algorithm with a minimum training cost below 0.04, but most of the other algorithms are not far behind. Only SGD with handbrake momentum and ExpAdam have a much higher minimum average training cost. The state of the art SGD algorithms do not seem to be inferior to the adaptive learning rate algorithms in this experiment: SGDNAG even manages to obtain the second-best score on the minimum average training cost reached. The standard Adam algorithm is only outperformed by the ℓ_{ada}-version.
The plots in Figure D.7 show for this experiment for different algorithm groups the average training cost reached per iteration over the 5 runs. The plots are slightly zoomed in, such that the differences between the algorithms are more clearly visible. Only the plot showing all algorithms is not zoomed in, such that the SGDHB performance can also be seen. This performance is quite bad since it does not only

reach a minimum average training cost nearly as low as the other algorithms but also starts to increase its training cost again after about 30 iterations over the training set. This makes the shape of the plot of SGDHB unique among the plots of all algorithms. Furthermore, the plots show that ExpAdam is also performing worse than most of the algorithms. However, contrary to SGDHB it manages to keep on decreasing its training cost until the last iteration. The other algorithms are quite close to each other, but Figure D.7b shows that AMSGrad is also performing slightly worse than the others, and the gap seems to increase towards the end of the run. Figure D.7c shows that, except for ExpAdam, there is hardly any difference in performance between the Adam versions. At the start of the runs it seems that AbsAdam takes the lead, but at the end it reaches a similar minimum value as standard Adam, ℓ_{1-4}-Adam and ℓ_{ada}-Adam.
Looking at the scores on the minimum test cost reached, it turns out that also here ℓ_{ada}-Adam has the best performance, directly followed by Nadam. It reached the second highest test accuracy reached, just behind AdaMax. It seems that the algorithms with the worst score on the training cost also have the highest values for the minimum test cost reached. Apart from these algorithms, the scores of the various algorithms lie quite close to each other: between 0.054 and 0.061. It seems that the SGD algorithms are somewhat behind the adaptive learning rate algorithms on the test set, but this is also only a narrow gap. The standard version of the Adam algorithm again is only outperformed by the ℓ_{ada} variant.
Figure D.8 shows for the various algorithm groups the average test cost per iteration during the 5 runs. These plots are also zoomed in to enhance the visibility of differences in performance, only Figure D.8 shows the full plot to keep SGDHB in view. This algorithm again has a very bad performance, and this is probably a consequence of the progress in the training cost. Its average test cost plot is somewhat similar to its average training cost plot: decrease until about 30 iterations, and after that slow increase with at the end suddenly a big increase. In general, the plots of the training cost and test cost for all algorithms look quite similar. A major difference is that the group of algorithms with the best performance keeps on decreasing its training cost, however their test costs seem to have converged after about 20 iterations after which it stays at the same level. This is unlike the test cost plots in the experiments on logistic regression or an MLP, where a typical valley occurred somewhere in the first half of the iterations after which the test cost started to slowly increase again. The plots show that the best performing algorithms are very close to each other, and the various Adam algorithms except for ExpAdam do therefore also have a similar performance.

8.2 CNN on CIFAR10

Similarly to the MNIST data set also for the CIFAR10 data set an experiment is conducted with a convolutional neural network. The results of the experiments with an MLP showed that these learning systems were not able to reach a very high test accuracy with any optimization algorithm, also the addition of two extra hidden layers did not really work. The maximum test accuracy obtained in all these experiments was 54.63% by the Nadam algorithm. On the training cost, the results were somewhat better, but an even close to perfect score was never reached: the maximum here was 97.10% by AdaMax. This means that there is quite some room for improvement for a CNN.

8.2.1 Network architecture

The convolutional neural network used in this experiment consists of 2 convolutional layers followed by a fully-connected layer. The first convolutional layer uses six kernels of size 5x5 on the 3D input (2D images from 3 different channels in the RGB data structure), after which the ReLU activation function is applied and max-pooling is performed over 2x2 windows. The result is put into another convolutional layer, this time with 16 kernels of size 5x5. Again the ReLU activation function is applied, and max-pooling with a 2x2 window is included. The result is transformed from the 3D matrix to a vector using the customary implemented module "Flatten". The result, which includes 400 values, is directly fully connected to the output layer of 10 nodes, for the different classes in the data set. Cross-entropy is again used as the loss function.
Note that the order of ReLU and max-pooling is switched around in this network with respect to the CNN used on the MNIST data set. This order does not really matter since max-pooling is only a reduction operation leaving the maximum value from the set. Applying the activation before or after therefore does not matter for the result. The only difference is computation time since max-pooling takes away values to which otherwise also the activation function would be applied. Since the ReLU function, which is a computationally very cheap function, is used, the difference in computation time would also be quite small.

8.2.2 Experimental setup

The group of algorithms tested in this experiment consists of the algorithms from the selected group, plus a single extra algorithm which is ℓ_{1-4}-Nadam. This algorithm is included to find out if the effect of this new variant on the Adam algorithm also manifests for the Nadam algorithms.
The algorithms are all optimized in the usual way. For all algorithms, different parameter settings are tested during 3 runs where each run lasts 20 iterations over the training set. A batch size of 128 is used. The configuration that reached the smallest average training cost was selected as the optimal one. The final parameters for all algorithms can be found in Table A.11.
The experiment itself is conducted by running each optimized algorithm during 5 runs of each 40 iterations. Again a batch size of 128 is used, and after every 20 mini-batches, the average training cost and accuracy are calculated over these last 20 mini-batches. Furthermore, the performance of the current network on the test set is determined.

8.2.3 Results (O)

The results of this experiment are displayed in Table 8.2. It can be seen that the scores on the training set do not meet the scores obtained in the experiments with an MLP, but the scores on the test set are much better.
Looking at the scores on the minimum average training cost reached, it turns out that standard Nadam has reached the smallest value, directly followed by AMSGrad and ℓ_{1-4}-Nadam. This means that the (small) increase in performance on the test set for ℓ_{1-4}-Adam with respect to standard Adam does not apply to the Nadam algorithm. This variant of Adam is also the only one that outperforms the standard version. ExpAdam is the worst performing algorithm of all, while AbsAdam is only slightly better than the SGD algorithms, which make up the rest of the group of

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	0.960	66.69%	1.084	62.96%
SGDmom	0.953	66.77%	1.091	62.71%
SGDNAG	0.950	66.96%	1.093	62.89%
SGDHB	0.966	66.58%	1.095	62.89%
Adam	0.930	67.59%	1.078	63.22%
AdaMax	0.939	67.30%	1.081	63.45%
Nadam	**0.905**	**68.55%**	**1.040**	64.47%
AMSGrad	0.907	68.52%	1.045	**64.59%**
AbsAdam	0.945	67.04%	1.089	62.77%
ℓ_{1-4}**−Adam**	0.927	67.87%	1.065	63.90%
ExpAdam	1.084	62.33%	1.143	60.01%
ℓ_{ada}**−Adam**	0.939	67.31%	1.080	63.10%
ℓ_{1-4}**−Nadam**	0.915	68.30%	1.054	64.00%

TABLE 8.2: Results of all algorithms on the CIFAR10 data set using a convolutional neural network with 2 convolutional layers and one fully connected layer. For both the training set and test set the minimal cost and maximal accuracy (after 40 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

worst performing algorithms. SGD with handbrake momentum could not outperform the other SGD algorithms.
Plots of the average training cost per iteration over the 5 runs of each algorithm can be seen in Figure E.11. ExpAdam is clearly by far the worst performing algorithm, although at the very start standard SGD has the highest score. Furthermore, it looks like SGDNAG has the best score in the first part of the runs, but then the two Nadam versions take over. Figure E.11f shows that during the first 25 iterations the ℓ_{1-4} version performs better and that the standard version only surpasses it near the end of the runs. This is not the case for the Adam algorithms, whose plot in Figure E.11c is slightly zoomed in to make the difference between the versions more clearly visible. These algorithms seem to have an almost equal plot while outperforming the other versions during the entire runs. Figure E.11d shows that SGDHB has the lead over standard SGD for the biggest part of the runs, and only at the end it is surpassed.
The values of the average minimum test costs reached in Table 8.2 show that also here Nadam has the best performance, and once again AMSGrad and ℓ_{1-4}-Nadam follow at a small distance. It seems that for each algorithm the relative performance on the training set carries through in the relative performance on the test set. Again ExpAdam is worst performing, followed by the SGD algorithms (from which SGDHB has the worst score) and AbsAdam. However, standard SGD is the best performing SGD algorithm in this case, while it was, except for SGDHB, the worst performing algorithm on the training set. Standard Adam is again only outperformed by its ℓ_{1-4} variant.
The plots of the various algorithm groups showing the average test cost per iteration over the 5 runs are displayed in Figure E.12. These plots seem to be very similar to the corresponding plots of the training cost. This is different from the test plots in other experiments, where often a typical learning effect on the test set occurred, resulting in a valley in the first part of the plot followed by a slow increase in test cost, but more similar to the plots obtained in the experiment with a CNN on MNIST. In this case however, it seems that after 40 iterations over the training set the point in the learning process where the test cost increases again or at least stabilizes is not yet

reached. This could mean that an even lower test cost could be reached using this convolutional neural network when it is trained for more iterations.

8.3 CNN with batch normalization on CIFAR10

In the experiments with CIFAR10 on the MLPs, the results show that the addition of batch normalization to the deep learning system in general improves the performance on the test set, while slightly worsening the performance on the training set. Since the results of the previous experiment still keep plenty of room for improvement, it is investigated in this experiment if this effect also occurs when a convolutional neural network is used instead of a standard multi-layer perceptron.

8.3.1 Network architecture

The convolutional neural network is built up in a same way as the one used in the previous experiment, but now batch normalization is included. The network starts with a convolutional layer with six kernels of size 5x5 on the 3D input (2D images from 3 different channels in the RGB data structure), after which batch normalization is performed, the ReLU activation function is applied and max-pooling is added over 2x2 windows. The result is put into another convolutional layer, this time with 16 kernels of size 5x5, followed by batch normalization. Again the ReLU activation function is applied, and max-pooling with a 2x2 window is included. The result is transformed from the 3D matrix to a vector using the custom implemented module "Flatten" added to the PyTorch package. The result, which includes 400 values, is directly fully connected to the output layer of 10 nodes, for the different classes in the data set. Cross-entropy is again used as the loss function.

8.3.2 Experimental setup

In this experiment, only the algorithms in the selected group are considered. The parameters of all algorithms are optimized in the usual way, by taking the configuration that reached on average over 3 runs the smallest training cost after 20 iterations over the entire training set with a batch size of 128. The resulting parameters are all displayed in Table A.12.
In the experiment itself, each optimized algorithm is tested during 5 runs each lasting 40 iterations over the training set with a batch size of 128. Every 20 mini-batches the average training cost and accuracy are calculated, and the current performance on the test set is determined.

8.3.3 Results (P)

The results of this experiment are displayed in Table 8.3. It can be seen that for all algorithms the performance on both the training set and the test set has increased compared to the previous experiment.
The algorithm with the smallest minimum average training cost reached turns out to be SGD with Nesterov momentum. The SGD algorithms are not clearly inferior to the others regarding the training cost, although standard SGD is the second worst algorithm on this statistic after ExpAdam. The gap between this latter algorithm and the rest is smaller than in the previous experiment: its minimum average training

Algorithm	Min training cost*	Max training acc	Min test cost	Max test acc
SGD	0.9112	68.44%	1.0321	64.87%
SGDmom	0.8952	69.00%	1.0309	64.71%
SGDNAG	**0.8769**	**69.57%**	1.0224	**65.16%**
SGDHB	0.9082	68.54%	1.0378	64.48%
Adam	0.8985	68.88%	1.0339	64.50%
AdaMax	0.8811	69.46%	1.0241	65.05%
Nadam	0.8782	69.50%	**1.0219**	65.00%
AMSGrad	0.8976	68.78%	1.0415	64.30%
AbsAdam	0.9061	68.55%	1.0378	64.54%
ℓ_{1-4}−**Adam**	0.8879	69.10%	1.0242	65.05%
ExpAdam	0.9267	67.90%	1.0409	64.27%
ℓ_{ada}−**Adam**	0.8972	68.94%	1.0280	64.78%

TABLE 8.3: Results of all algorithms on the CIFAR10 data set using a convolutional neural network with 2 convolutional layers and one fully connected layer with batch normalization. For both the training set and test set the minimal cost and maximal accuracy (after 40 iterations over the entire training set) averaged over 5 runs with randomly shuffled data are displayed, and per column the best value is highlighted.

cost is only 0.05 higher than the one for SGDNAG. SGD with handbrake manages to outperform standard SGD but performs worse than the other two. Looking at the different versions of Adam, it turns out that standard Adam is outperformed by the ℓ_{1-4}- and ℓ_{ada} variants.
Plots of the training cost reached per iteration averaged over 5 runs of the various algorithm groups can be seen in Figure E.13. The plots are all zoomed in on the right side of the plot with the minimum values, such that differences are more clearly visible. This makes only the part of the lines visible after about 12 to 30 iterations (so ExpAdam needed about 18 more iterations to reach the training cost of Nadam at iteration 12). In this part of the plot, it can be seen that Nadam and SGDNAG more or less take turns in the leadership, with AdaMax and ℓ_{1-4}-Adam close behind. This latter algorithm has during the iterations visible in the plot always a better score than standard Adam, but it seems that the size of the gap stays roughly the same which would mean that the lead has been gained in the first 15 iterations of the runs. ℓ_{ada}-Adam seems to compete with standard Adam, while AbsAdam performs somewhat worse. SGDHB has the lead over standard SGD, but it seems that the latter slowly catches up with it.
Looking at the scores on the minimum average test cost reached in Table 8.3 it turns out that Nadam has the best score of 1.0219 which is about 0.018 smaller than on the CNN without batch normalization. SGDNAG however is not far behind and even manages to reach the highest average accuracy on the test set. The scores of the algorithms are very close to each other: the difference between AMSGrad, which has the worst score of 1.0415, and Nadam is less than 0.02. SGDHB is not able to outperform any of the other SGD algorithms on the test set but does outperform several of the adaptive moment algorithms. Standard Adam is again only outperformed by the ℓ_{1-4}- and ℓ_{ada} variants. Figure E.14 shows for the different algorithm groups a plot of the test cost reached per iteration over the training set, averaged over the 5 runs. These plots are also zoomed in on the part of the plot with the minimum values for the algorithms to make the differences more clearly visible. It can be seen that the plots have roughly the same shape as the plots of the training cost, similarly to the plots of the results from the previous experiment. So probably also here the

test cost could be decreased further by optimizing the weights for more iterations on the training set. Figure E.14d shows that SGDNAG takes a big lead in the first half of the runs and that the other algorithms only slowly catch up during the second part. Furthermore, SGDHB is at the end overtaken by standard SGD. Figure E.14b shows that Nadam is the leading adaptive learning rate algorithm during the entire runs, and ℓ_{1-4}-Adam is constantly second. Standard Adam has a reached minimum value for the test cost that the ℓ_{1-4} variant already reached after 22 iterations over the training set, which is about 15 iterations earlier. Furthermore, there seems to be more difference in this test plot between standard Adam and its ℓ_{ada} variant than in the plots of the training cost, with the variant taking a bigger lead. AbsAdam performs slightly worse than the standard, while ExpAdam seems to be the worst performing algorithm during the entire runs. However, ExpAdam seems to almost catch up with AbsAdam and AMSGrad at the end of the runs with quite a steep decrease in the line, so maybe after some additional iterations it could even catch up with standard Adam.

8.4 Comparison of CNN and MLP

Given the results of the several experiments on MNIST and CIFAR10, a small comparison can be made of the performances on multi-layer perceptrons and convolutional neural networks.
The results suggest that the CNN can reach higher scores on the test set than the MLP. In the experiments on an MLP, the minimum test cost was always reached somewhere in the first part of the runs, and after that it often increases again while the training cost kept decreasing. This is a sign of overfitting the network on the data in the training set, which made the MLP reach a lower minimum training cost than the CNN. The results of the CNN's on the CIFAR10 data set however indicate that at the end of the runs the network might not be fully optimized such that the minimum test cost has been reached since this value is still decreasing. Nonetheless, it already reached a test cost lower than the MLPs managed to reach at their minimum, so the CNN performed much better. This applies also to the MNIST data set, in which the CNN did reach the minimum test cost for most optimization algorithms in the first part of the runs, but after that it managed to keep it at that level.
Another observation of the results is that the addition of batch normalization seems to have less influence on the performance of the CNN than for the MLP. This might suggest that the use of shared weights in the kernels, and of course in the case of the CNN on MNIST also the inclusion of dropout in the network, already leads to a significant amount of regularization in the network.

Chapter 9

General discussion

In the previous five chapters, several experiments are described in which the existing optimization algorithms for gradient descent in deep learning systems are compared. All these experiments together make up for a big pile of results left for interpretation. In this chapter, it is tried to highlight the most important or noteworthy aspects in this pile of results, and to find out the reasons behind them.

9.1 Summary of results

As already noted in section 3.3 which describes the general methodology of the experiments conducted in this thesis, it is not convenient for a final performance comparison to compare the algorithms on each individual experiment. Instead, the scoring system also described in that section is used for this purpose. The results of this are displayed in Table 9.1. This table shows for each algorithm the points obtained from the performance ranking on each individual experiment. These experiments are named A to P such that the table would fit on the page, and the corresponding experiment is indicated by the letter between brackets in the titles of the result section of every experiment in this thesis. As also indicated in the section which includes the description of this scoring system: the ranking criterion of each experiment is indicated by an asterisk (*) behind the name of the considered statistic in their respective result tables.
It is nice to see that the state of the art algorithms appear in the ranking in the table more or less in the chronological order of proposal: the only thing here is that the AMSGrad algorithm should be higher than Nadam, but apart from this is (from worst to best) SGD-SGDmom-SGDNAG-Adam-AdaMax-AMSGrad-Nadam a perfect reflection of the historical development of optimization algorithms for gradient descent in neural networks. Since AMSGrad was proposed as an improvement of Adam and not of Nadam (they can even be combined into NAMSGrad, which by the way does not automatically make up for a superior algorithm, as shown in the first couple of experiments in this thesis), these results indicate that every step in the developmental process is indeed an improvement of the previous one. It therefore also shows that the adaptive learning rate algorithms are superior to the standard SGD algorithms.
In that light, the results show a possible next step in the progress towards the optimal algorithm, since both ℓ_{1-4}-Adam as well as ℓ_{ada}-Adam appear on the very top of the ranking. Their score differs by only a single point, but it differs quite significantly from Nadam, which has a steady third place. The standard version of Adam only appears as sixth, just before AbsAdam. ExpAdam has a bad score since it is the worst among the adaptive learning rate algorithms and also below SGD with Nesterov momentum. SGD with handbrake momentum also has a bad score. It was intended as an improvement for SGDmom, but the results show that it is hardly an

ALG	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**tot**
$\ell_{1-4}-$Adam	12	12	11	11	8	12	12	8	10	7	9	10	10	6	10	9	157
$\ell_{ada}-$Adam	6	11	10	12	12	11	11	9	11	10	7	11	8	12	8	7	156
Nadam	3	7	8	6	10	9	10	10	7	12	12	8	6	10	12	11	141
AMSGrad	5	9	9	10	11	7	4	6	9	11	10	9	12	3	11	6	132
AdaMax	7	3	6	2	6	5	6	12	12	9	8	12	11	8	7	10	124
Adam	4	8	7	8	5	8	3	7	6	8	11	7	9	9	9	5	114
AbsAdam	10	10	12	9	7	6	5	11	4	4	5	5	5	7	6	4	110
SGDNAG	8	5	3	3	4	4	8	5	8	5	6	3	4	11	5	12	94
ExpAdam	9	2	5	7	9	10	9	1	1	6	1	6	7	2	1	1	77
SGDmom	11	4	4	4	3	3	7	3	3	3	4	4	3	4	4	8	72
SGDHB	1	6	2	5	2	1	2	4	2	2	3	1	1	1	2	3	38
SGD	2	1	1	1	1	2	1	2	5	1	2	2	2	5	3	2	33

TABLE 9.1: The score for each algorithm from the selected group obtained in every experiment conducted in this thesis. The best performing algorithm among this group in a single experiment gets 12 points, second best 11, etc., and the worst performing gets 1 point. The points for every algorithm are accumulated in the rightmost column, and the algorithms are in descending order of this final score.

improvement of standard SGD.
It must be said that the attempt made in Table 9.1 to summarize the differences in performances between the algorithms in the experiments ignores how much the results differed. However, the small analyses of the results of each experiment showed that often the differences between the best scoring algorithms were very small, while in most of the cases the worst algorithms reached a much higher minimal cost than average. This might indicate that the more recent a development in optimization techniques is proposed, the smaller the improvement in performance gets. The scores also ignore the difficulty of the task. For example, the Rosenbrock function is much easier to optimize than a CNN for CIFAR10. It can also be seen that, when the experiments on Rosenbrock and XOR are removed, suddenly Nadam performs best, but Adam is still outperformed by ℓ_{1-4}-Adam and ℓ_{ada}-Adam. The results on the experiments showed that the more complicated the used classifier becomes, the smaller the differences in performance between the optimization algorithms becomes. This might indicate that for bigger systems the optimization algorithm has less influence, probably due to the many weights to optimize and thus several ways to reduce the cost.

9.2 Analysis of algorithms

In this section, it is tried to analyze and interpret the results of the five different new optimization algorithms, and to find out for each of them if it is an improvement. This is done using the results in Table 9.1 and some of the feature plots in Appendix C. These kinds of plots are available for all experiments with MNIST or CIFAR10 (experiments **F** to **P** in Table 9.1), and since there are eight different features (as described in section 3.3) for at least four considered groups per experiment, it is at least impractical to include them all in this thesis. Moreover, for some statistics the plots from different experiments turn out to be quite similar. Therefore it is decided to use the feature plots from only two of the experiments as a reference for each algorithm group. As first experiment the one with the most similar order of considered algorithms when compared to the final order is chosen, and as second experiment the one with the most unsimilar order. This turns out to be experiments **J** (CIFAR10 on MLP with 1 hidden layer) and **N** (MNIST on CNN) for the SGD algorithms, and experiments **P** (CIFAR10 on CNN with batch normalization) and **H** (MNIST on MLP with 1 hidden layer) for the Adam variants.

9.2.1 SGD with handbrake

As already indicated, this algorithm turned out to be a failure, since the purpose was to at least level the performance of SGDmom and SGDNAG, but the results showed it could barely outperform standard SGD. Table 9.1 shows that the algorithm is often among the two worst performing algorithms, and its highest scores are sixth and seventh on two of the experiments on XOR.
In the introduction of this algorithm in Chapter 3, it was reasoned that the handbrake would prevent the weights to go uphill (due to the momentum term) after a (long) downhill movement, so a bit like the idea behind Nesterov accelerated gradient momentum. In this way, the weights can make use of the momentum term when it is still safe to do so, and get rid of it the moment the gradient starts to point in the opposite direction. This would in theory improve the performance of SGD with standard momentum. Except for three incidental better scores, the results show clearly no improvement but retrogression with respect to SGDmom.
To get an idea about the reasons why the algorithm does not work as expected, the feature plots of the SGD algorithms are considered for experiments **J** (most similar to final performance results) and **N** (most unsimilar). The feature plots of experiment **J** are displayed in Figure F.1. In this experiment SGDNAG performed best, followed by SGDmom, SGDHB, and finally standard SGD. Figure F.1c shows that the better an algorithm performed, the bigger the obtained gradients on average were. This could also be derived from Figure F.1a, which shows the average gradient: the value for SGD lies constantly very close to zero, while SGDHB's values are somewhat bigger (and on average almost everywhere positive), and SGDNAG and SGDmom reach much bigger values both on the positive and negative side. The plots of the standard deviations of the gradient and absolute gradient are very similar to the plot of the average absolute gradient, but not exactly the same. So from this it can be concluded that averaged over the five runs in the experiment, the standard SGD algorithm got on average (over the gradients in the mini-batches per iteration) much smaller gradients over the entire run than the other algorithms. So this is a remarkable difference, but it is more important what the algorithm does with the gradients. Figure F.1b and Figure F.1d try to give some answers by showing the average delta weight and the average absolute delta weight per iteration averaged over the five runs in the experiment. It is clearly visible that for this statistic standard SGD has the highest value over the entire runs, while SGDmom and SGDNAG are close to each other and have the smallest values and SGDHB is somewhere in between. This means more or less that the algorithms with on average small absolute gradients perform big absolute weight updates. The reason for this is the size of the learning rate of each algorithm: Table A.9 shows that SGD uses a much higher learning rate than the others (0.18), where SGDHB has 0.05 and the other two both use 0.02.
When the feature plots of the other considered experiment, CNN on MNIST, are analyzed, one of the first things that are noticed is the very big positive average gradient and delta weight of SGDHB at the end of the runs, while the ones for the other algorithms stay much closer to zero. This can also be seen in the plot of the average absolute gradient in which SGDHB has much higher values than the other algorithms, and in the plot of the average absolute delta weight which shows that SGDHB performs on average hardly any weight updates except for the very end.

The standard deviation plots are again very similar to their respective average absolute feature plots.
So the differences between these plots and the ones analyzed above are the big positive values at the end of the runs for SGDHB on all statistics, and the higher average absolute gradient during the entire runs. This can indicate that the weights in the system keep on getting bigger gradients in both directions (average gradient close to zero) for multiple dimensions (higher absolute gradient) but somehow no momentum is built up, which is probably due to the handbrake mechanism that keeps on slowing the learning process down. At the end suddenly a very high average gradient occurs, and also the average absolute gradient becomes bigger, which could indicate that suddenly one or a couple of very high gradients are encountered, or that most gradients point into the same direction. In either way would this be a good opportunity to enhance the performance of the network, but it is strange that this occurs so late in the training process. Therefore it can also be caused by some bad previous updates which moved the system out of the local minimum. This is probably the reason that SGDHB performed in this experiment worst of all algorithms. SGD did slightly better, followed by SGDmom and SGDNAG. This last algorithm also reached the second best score among all tested algorithms.
So the reason that SGDHB has such a bad performance might be that the handbrake is used too much, i.e. hardly any momentum is built up probably mainly due to the occasional bumps that are present in areas with in total a steep downhill gradient. Moreover, this leads to unstable behavior in which the weights can move out of a (local) minimum in the error space.
A possibly better version of this algorithm would remove the momentum term before the gradient starts to point into another direction instead of afterwards when the weights have already gone "uphill" a bit. This could be implemented using some technique similar to Nesterov momentum, which also looks ahead to see where the current update with momentum would shift the weights to. This idea could be researched in the future.

9.2.2 Different GHC methods

The different gradient history collection methods for adaptive learning rate algorithms are first discussed separately in terms of the general results in Table 9.1. At the end, they are all analyzed using the feature plots from the considered experiments.

Absolute gradient

The Adam algorithm with the GHC method that uses the absolute value of the gradient in the exponentially decaying average performed in general slightly worse than Adam with the standard GHC method. In the experiments on the Rosenbrock function and the XOR problem it performed better than the state of the art variant, but in the experiments on MNIST and CIFAR10 it managed to outperform standard Adam in only 3 of the 11 cases. Therefore it reached at the end a score just worse than Adam. In the experiments where the method was also implemented in and compared to other state of the art algorithms (like RMSprop and Adagrad), the majority of the Abs variants managed to outperform their standard counterparts in the Rosenbrock and XOR experiments as well (especially on the convergence rate on the XOR networks without batch normalization), but also performed worse on the experiment with logistic regression on MNIST. So it seems that for easy problems this method is an improvement of the standard, but for the harder problems it becomes worse. It can also be the case that the method gets worse when the neural networks are bigger.

Changeable norm

The ℓ_{1-4}-Adam algorithm, in which the used norm in the GHC can be changed from 2 to a value between 1 and 4, and in which if needed also a certain increase or decrease over iterations can be included, shows very good results in the vast majority of the experiments. It turns out that the algorithm gets on 13 of the 16 experiments a higher amount of points than the standard version, as can be seen in Table 9.1. Moreover, it has the highest number of points of all algorithms, making it the best performing algorithm. When the implementations of this method in other state of the art algorithms, as tested in the first couple of experiments in this thesis, are considered it turns out that in general the standard versions are here outperformed as well. Note that in the majority of the experiments this variant uses the same learning rate as the standard Adam algorithm, for which it was optimized. Also note that the option to increase or decrease the norm over epochs or iterations is omitted most of the times. The use of the different norm, constant throughout the entire optimization process, alone causes the increase in performance of the algorithms, which made ℓ_{1-4}-Adam the very best overall scoring algorithm.
However, also for the ℓ_{1-4}-Adam algorithm there seems to be a slight decay in performance when the networks get bigger and/or the problems get more complicated. In the experiment on Rosenbrock and XOR it is in 4 of the 5 cases among the top 2 best-performing algorithms, while in the other 11 experiments it manages to reach the top 2 only twice. In the experiments that it reaches a lower rank, it is often outperformed by Nadam and ℓ_{ada}-Adam, and sometimes also by AdaMax, AMSGrad, or even SGDNAG. This indicates that the GHC method might be an improvement of Adam, but it is not always superior.

Adaptive norm

The algorithm with an adaptive norm, ℓ_{ada}-Adam, was proposed as a more dynamical variant of the ℓ_{1-4}-Adam algorithm. This GHC method used a different norm for each weight in each update, calculated using a basic norm (between 1 and 4) minus the ratio of the exponentially decaying average of absolute gradients divided by the maximum obtained absolute gradient. The idea behind this was to decrease the norm the moment a bigger gradient is encountered, and in this way it was tried to allow more exploration in the case of a small gradient and less exploration for a big gradient.
The results of the ℓ_{ada}-Adam algorithm are of a similar level as from ℓ_{1-4}-Adam, so very good and much better than the standard version of Adam. It reached the second place, 1 point behind the first place. It was able to be the best performing algorithm in 3 experiments, from which 2 on the XOR problem. It managed to outperform standard Adam in 13 of the 16 experiments. The effect of better performance on smaller networks and/or easier problems, as Abs and ℓ_{1-4} had, is also here present, but it seems to be somewhat less.

Exponential function

The final new GHC method tested used, instead of a norm, an exponential function to collect the gradients. The exponential decaying average used exponential gradients, and the logarithmic function of the result was taken. This method was implemented to see if something other than the use of norms would also work.
The results in Table 9.1 clearly show that this is not the case. The ExpAdam algorithm is among the worst performing algorithms, even behind SGDNAG. Moreover, in 5 cases the algorithm performed the very worst. On the other hand, it managed to reach four top 4 positions (two of them on Rosenbrock or XOR), and it outperformed standard Adam in four experiments, making the algorithm not as bad as SGDHB. Nonetheless, the results seem to indicate that not every function is suitable to use in a GHC method.

Analysis of feature plots

To have some insight in the difference in optimization strategies between the different GHC methods, the feature plots are considered of experiments **P** (CIFAR10 on CNN with batch normalization) and **H** (MNIST on MLP with 1 hidden layer).
The plots of experiment **P**, which is the one with the most similar, in this case even the exact same, result as the final order in Table 9.1 considering the Adam algorithms, are displayed in Figure F.4. The plot of the average delta weight does not give much information: all algorithms seem to have a plot that is decreasing during the first couple of iterations, after which it gets more or less stable at the same value. All algorithms also have a slightly oscillating line such that it becomes quite difficult to view them separately, except for ExpAdam which seems to have much less oscillation than the other algorithms. The value of this algorithm also approaches quicker the value of zero. The plot of the average gradient gives some more information. Again all lines are heavily oscillating (especially ExpAdam and ℓ_{ada}-Adam), but it is clearly visible that this former algorithm at the start has a much lower average gradient than the others. These other algorithms all have their plot similarly shaped, again first decreasing and then more or less stable around the same value.
The plots of the average absolute gradient and average absolute delta weight give more information about the differences between the algorithms. In the first plot, Adam and ℓ_{ada}-Adam seem to have more or less the same plot, which slowly decreases. AbsAdam constantly has a smaller value, and is also decreasing, as well as ℓ_{1-4}-Adam but this algorithm has constantly a slightly bigger value. ExpAdam has a plot that is much different since it starts at a small value and rapidly increases and finally slows down again. It is a bit like a logarithmic function, which is probably a coincident with the use of a logarithmic function in the GHC method of ExpAdam. The plot of the average absolute delta weight shows for all algorithms a relatively high value in the first iteration and a rapid decrease direct afterwards. For the rest of the runs, ExpAdam has a very low value, the value of AbsAdam is somewhat higher, and the other three algorithms have even higher values. These three algorithms have a slowly decreasing plot towards the end, while the line of AbsAdam slowly decreases. Standard Adam has the highest value, directly followed by ℓ_{ada}-Adam and ℓ_{1-4}-Adam.
The second experiment considered in this analysis is with the neural network with 1 hidden layer on MNIST (experiment **H**). In this experiment the AbsAdam algorithm performed best of the Adam algorithms, followed by ℓ_{ada}-Adam, ℓ_{1-4}-Adam, Adam, and lastly ExpAdam. The plots of this experiment are displayed in Figure F.3. Both

the plot of the average gradient and the plot of the average delta weight do not show big differences between the algorithms, except for the very first (couple of) iterations. It seems in this early phase that the ℓ_{1-4}-Adam algorithm gets much bigger gradients on average than the other algorithms, both in the initial negative gradients as in the hill towards the positive side afterwards. However, the ExpAdam algorithm performs by far the biggest weight updates on average in this phase. The figures with the average absolute gradient and average absolute delta weight do show more differences between the algorithms. In both plots, it can be seen that ExpAdam reached on average higher absolute delta weights over the entire run, and also higher absolute gradients but these only in the second half of the runs. This could indicate that this algorithm keeps on oscillating around a (local) minimum in the error landscape, only slowly moving towards it (indicated by the slow decrease in absolute gradient). This would explain the slow progress made by ExpAdam, which was also noticed in the plots of training cost in multiple experiments, and the bad performance of this algorithm.
During the first couple of iterations the other algorithms reach higher absolute gradients on average, with Abs and ℓ_{1-4} reaching the highest. These two algorithms however have the smallest absolute delta weight at the start. It is also remarkable that during the second half the particularly AbsAdam and to some extent also ℓ_{1-4}-Adam seem to receive only very small gradients, and hence only perform very small updates on the weights. Standard Adam and ℓ_{ada}-Adam keep on receiving bigger gradients and performing bigger delta weights on average more or less all the way to the end of the runs. This could mean that the latter two are still busy converging towards a minimum in the error landscape, while the former two already found their particular (locally) optimal configuration. Especially AbsAdam shows this behavior, and this can be a reason for the result that this algorithm performed best in this experiment.

Conclusions per GHC method

From these analyses it can be concluded that the bad performance of the ExpAdam is probably due to its disability to properly translate the size of the gradient to the importance of it: the feature plots of one of the experiments showed that even in the case that a high amount of big gradients is encountered, the algorithm only performs very tiny weight updates, while those of the other considered experiment showed that even when the algorithm performs bigger updates the obtained gradients remain high. These are both examples of misinterpreting the importance of a certain gradient, which makes the algorithm much slower in convergence/optimization than the standard algorithm. This GHC method is clearly not the way to go for improvement of the standard method.
The Abs GHC method uses the absolute value of the gradient in the exponentially decaying average instead of the squared value. As the feature plots indicated, the gradients always have very small values, so it can be said safely that they are always between -1 and 1. This means that the absolute value of a gradient is always bigger than the squared value, and hence the square root of it is also always bigger. This square root is in the denominator in the scaling factor of the update rule, and hence this factor should become smaller when Abs is used. This would cause the algorithm to take smaller steps. This can of course be compensated by a higher learning rate for the Abs algorithm, and according to the optimal parameter values as displayed in the tables in Appendix A this is indeed the case: AbsAdam often uses a learning

rate 10 or more times higher than standard Adam. However, it seems that this does not compensate entirely. The proof in Appendix G shows that this is true since it is proven that the parameter configuration of Adam cannot be changed such that it completely mimics the behavior of the Adam algorithm with the Abs GHC method. As the feature plots showed, the AbsAdam algorithm received on average smaller gradients and performed also smaller weight updates. This second fact can be due to the working of the Abs method as described above, while the smaller gradients could be a consequence of this. This is probably the main reason that AbsAdam generally performed slightly worse than standard Adam. The superior performance of AbsAdam on the Rosenbrock function and three of the experiments on the XOR problem could also be explained by this, since during the optimization process of these problems much bigger gradients are encountered (in Rosenbrock even much higher than 1, which would turn around the reasoning above and make the scaling factor bigger when Abs is used, which makes it take bigger steps and in this problem this leads to faster convergence). So it seems that Abs might be a good method to use in a problem where the error landscape is full of big gradients, but a less good idea to use in other cases, which unfortunately also include more realistic data like MNIST and CIFAR10.
ℓ_{ada}-Adam and ℓ_{1-4}-Adam both are able to change the norm used in the exponentially decaying average and in the scaling factor, and by doing this try to change the rate of importance of a gradient. The key factor here is the decay parameter in the exponentially decaying average. Since the gradient, after taken to a power, is first multiplied by this decaying factor before the root is taken, the norm that is used does make a difference. Again it can be assumed that the gradients are (very) close to zero, or at least between -1 and 1. When the same decay factor is used, a bigger norm will lead to a bigger value in the denominator, as can be seen in the following three calculations of v_t and $\sqrt{v_t}$ in the first optimization step, in which respectively a norm of 1, 2 and 3 are used and a gradient of 0.002 and momentum of 0.999 (as recommended by the authors (Kingma and Ba, 2014)), are assumed:

$$v_0 = 0.999 \cdot 0 + (1 - 0.999) \cdot (|0.002|)^1 = 2e - 06; \quad \sqrt[1]{v_0} = 2e - 06$$

$$v_0 = 0.999 \cdot 0 + (1 - 0.999) \cdot (|0.002|)^2 = 4e - 09; \quad \sqrt[2]{v_0} \approx 6e - 05$$

$$v_0 = 0.999 \cdot 0 + (1 - 0.999) \cdot (|0.002|)^3 = 8e - 12; \quad \sqrt[3]{v_0} = 2e - 04$$

This means that when a bigger norm is used, the value in the denominator of the scaling factor becomes bigger and thus when everything else remains the same, smaller steps will be made. In Appendix G it is proven that this effect cannot be imitated by the standard Adam algorithms when it uses some different parameter configuration. The proof shows that in the first optimization step the norm does not matter (which can also be seen in Figure 3.1), but in later optimization steps no parameter configuration can compensate for this effect. However, the shown effects do not match the observations in Figure 3.1. In this plot, a higher norm leads to more exploration and bigger steps. It is tried to explain this difference by plotting four different measurements over the first 50 iterations of the optimization processes as displayed in Figure 3.1. These plots can be seen in Figure 9.1, and it seems to be the case that the main cause is the denominator of the scaling factor in the update rule, which gets higher for lower norm values, and especially very high for the ℓ_1-norm. This can only be the case when the v_{t-1} term in the exponentially decaying average becomes much bigger for lower norms since the plot of the absolute gradient shows that the obtained gradients do not differ very much. Looking at the values for v_0 in

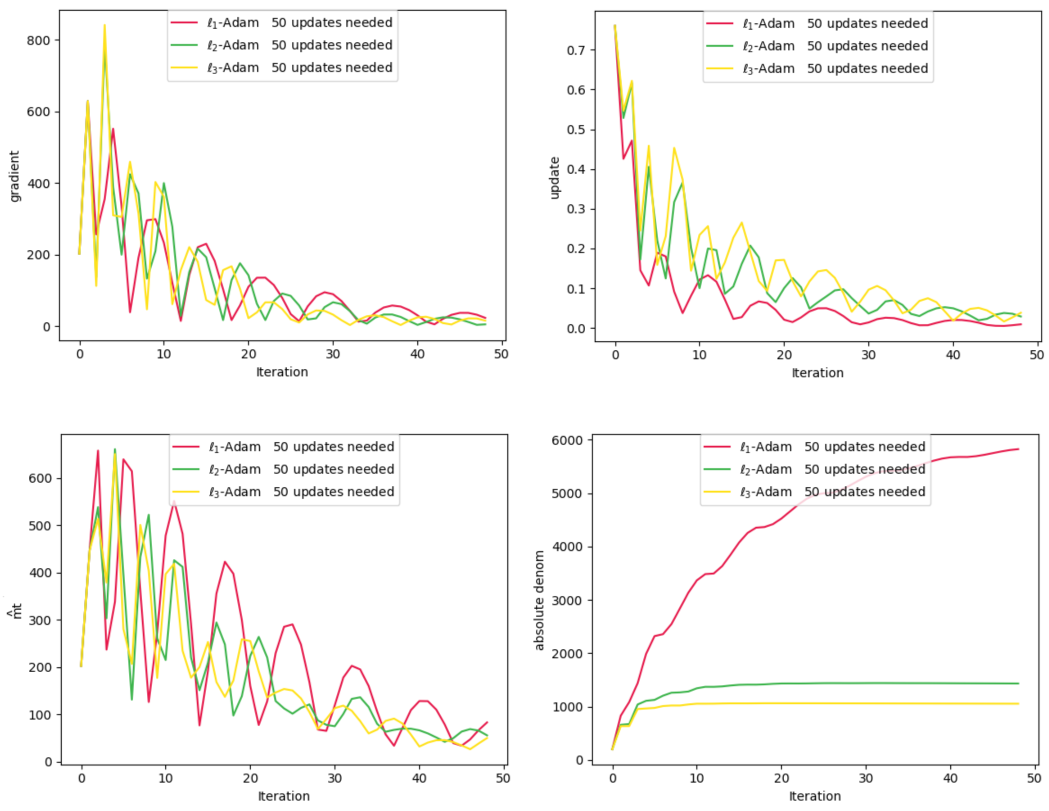


FIGURE 9.1: Plots of four measurements during the first 50 iterations of the runs as displayed in Figure 3.1: average absolute gradient, average absolute weight update, average absolute $\hat{m}_t$ and average absolute $\sqrt[n]{\hat{v}_t}$.

the equations above, this seems indeed to be the case: when the norm is increased by one, v_0 gets (in this case) almost thousand times smaller. In later values of v the v_{t-1} term is multiplied by the same value for all algorithms, so the difference remains. The lower values for later v_t terms for higher norms lead to bigger step sizes, and hence (somewhat) bigger gradients, but these can apparently not counterweight the difference in the v_{t-1} term.

When the parameter settings of the ℓ_{1-4}-Adam and ℓ_{ada}-Adam algorithms are observed in the tables in Appendix A, it can be seen that in 11 of the 16 experiments a norm higher than 2 is chosen for ℓ_{1-4}-Adam and ℓ_{ada}-Adam. These include the experiments on Rosenbrock and XOR, as well as the experiments with a single hidden layer neural network on MNIST, and CNN on CIFAR10. In the rest of the cases, only one of the two algorithms used a norm higher than 2. In other cases, especially the two experiments with logistic regression, a norm close to 1 turned out to be the best choice. However, as can be seen in Table 9.1, in both cases the standard Adam algorithm was outperformed. This seems to show that for different problems a different norm needs to be used. The choice of ℓ_2 in Adam and other state of the art adaptive learning rate algorithms is a good one, and also has a strong background in the variation theory, but clearly in almost all cases considered in this thesis the use of some other norm outperforms the ℓ_2 norm in standard Adam. Looking at the comparison and consequences of the different norms (as explained in the equations above), this could mean that for different problems other methods to determine the importance of a gradient must be used. How this mechanism exactly works is not completely clear yet, but according to the results, the adaptive norm as proposed in the ℓ_{ada} variant, which slightly lowers the norm the moment a big gradient is encountered to encourage more exploration the moment low gradients are obtained, seems to be at least a step in the right, and hopefully optimal, direction.

Chapter 10

Conclusions

In this thesis, some newly invented techniques for optimization algorithms for gradient descent in deep learning systems have been investigated. A total of five different methods were described and considered. The first one tries to incorporate a sort of handbrake in the existing SGD algorithm with momentum, resulting in the SGDHB algorithm. The other four all tried to improve the existing state of the art algorithms that use adaptive learning rates, especially Adam which is currently the most popular algorithm in the field. All possible improvements were variants on the mechanism in Adam and other adaptive learning rate algorithms which collects information about the history of the gradients, in this thesis named the gradient history collection (GHC) method. The standard method uses a ℓ_2 norm, and in the possible improvements this is changed into a method that uses absolute values in the exponentially decaying average of past gradients (Abs), a method that can use a different norm between ℓ_1 and ℓ_4 and can increase or decrease this value according to a fixed scheme over time (ℓ_{1-4}), a method related to the previous one that uses an adaptive norm based on the previously encountered gradients (ℓ_{ada}), and a method that does not really use a norm, but instead tries to collect information of the gradient history using an exponential function in combination with its inverse, the natural logarithm (Exp).

These newly invented GHC methods and the SGDHB algorithm have been tested in 16 different experiments on several optimization problems, including the MNIST and CIFAR10 data set, in different deep learning systems, either an MLP or CNN, or in a pure optimization setting. In the experiments on the more easy problems, some of the new GHC methods are tested in all state of the art algorithms for which they are suited. In the other experiments only a subset of the optimization algorithms, consisting of all SGD versions as well as Adam and all its derived versions, has been tested.

The results on all the experiments show in general that the SGDHB algorithm is not an improved version of the SGD algorithm with momentum at all: it barely outperforms the standard SGD algorithm. The Exp GHC method also has very bad results, while the Abs GHC method is slightly worse than the standard. The ℓ_{1-4} and ℓ_{ada} variants however have a much better performance, and even outperform on average the state of the art algorithms based on Adam, like Nadam, AdaMax, and AMSGrad.

10.1 Answers to the research question

The research question in the introduction of this thesis was stated as follows:

How can the state-of-the-art optimization algorithms for gradient descent in neural networks be adapted, such that the new variant outperforms its standard version in convergence rate and -speed?

Looking at the requirements for the research question mentioned in the first chapter, which an algorithm needs to meet to be a good candidate to be proposed as an improvement of the state of the art algorithms, it is clear that SGDHB, ExpAdam, and AbsAdam all are not suited since they do not meet the most important requirement which is performing better on the training data. The remaining two algorithms, ℓ_{1-4}-Adam and ℓ_{ada}-Adam do meet this first requirement, as can be seen in Table 9.1. Looking at the results on the test data in the experiments on MNIST and CIFAR10 (this is not applicable to the experiments on XOR and Rosenbrock), these variants have in 8 of the 12 experiments a better result than standard Adam, so it can be said that they also meet this requirement. The final requirement is about the time it takes for the optimization process. This might be a downside of these two variants since they take more time than standard Adam. They both include the calculation of the gradient to the power of some value, which takes more time than the squared operation in Adam. Additionally for ℓ_{ada}-Adam, extra time is needed to calculate the norm used in the update for every parameter, in which the calculation of the new exponentially decaying average of past absolute gradients takes the biggest part. All this must be done for every weight parameter in the network in every parameter update. Unfortunately, it was not exactly measured how much extra time this took, but looking at the file names of the runs (which include end times, and the previous run's end time is the current run's start time) this can be estimated. Taking the computationally most intensive experiment, which is CIFAR10 on the convolutional neural network with batch normalization, as the most extreme example, a single run with Adam took on the Peregrine cluster about one hour and 35 minutes, while it took with ℓ_{1-4}-Adam and ℓ_{ada}-Adam about 1 hour and 40 minutes. So this is slightly longer, but to put things in perspective: using the SGDNAG algorithm, that does not need to calculate a learning rate for every individual parameter at all, it took about one hour and 30 minutes. Moreover, since the methods perform better than the standard algorithm, it needs fewer epochs to reach a similar performance level and hence also less time than indicated above. So this means that the extra time needed is not significant enough to be a bottleneck for the newly proposed variants. Moreover, since they were of course customary implemented in PyTorch and since the implementations of the algorithms that are already in the package are heavily optimized, it is likely that these new versions can also be implemented in a more efficient way.
Looking at all the results, it can be concluded that the ℓ_{1-4} and ℓ_{ada} GHC methods do fulfill all requirements to be a candidate algorithm to be proposed as an improvement of the state of the art algorithms. These methods provide therefore good answers to the research question in this thesis.

10.2 Influence on the research field

The deep learning field is currently one of the most popular fields within artificial intelligence, and therefore also one of the fastest developing fields. If the new methods as proposed and described in this thesis that improve the general performance of the Adam algorithm and other adaptive learning rate algorithms would be published in some scientific journal, it could be picked up by the community as an interesting new method to use in their learning systems. However, there is no real guarantee for this. For example, the Adam algorithm became quickly after publishment very popular, while improved versions of Adam, like Nadam, did not really take over the role as the most used optimization algorithm. However, the ℓ_{1-4} and ℓ_{ada} GHC methods proposed in this thesis might open some new doors and lead to even better performing and more sophisticated versions.

10.2.1 Future work

This is also one of the main aspects of the ideas for future work coming from the research conducted in this thesis: it was tried to understand and explain the internal workings of the new GHC methods to come up with reasons why they perform better or worse than the standard method, but not all questions could be answered. Future research can be done to find out the exact reasons for the difference in performance, and investigate the interaction between the GHC method and the error landscape of the data. A better understanding of this could lead to a better GHC method, which would probably use an adaptive norm.

Other research could be done about GHC methods other than with a norm. The Exp method showed that the use of an exponential function in combination with the natural logarithm was no good idea, but maybe the use of some other function and its inverse will have good results. In theory, every function could be used for this purpose, as long as its output value increases for bigger inputs. An example method that could work is an alternative version of the Exp method described in this thesis, which takes the natural logarithm of the multiplication of the e-powers of the previously obtained absolute gradients. In that way, due to the mathematical rules behind the use of a logarithm ($\ln(e^x \cdot e^y) = x + y$), the magnitudes of the gradients could possibly be considered in a better fashion. It would need some research to incorporate the decay factor in the method, an option could be to put this factor in the power term.

The methods proposed in this thesis, especially the ones with good results, could also be tested on different deep learning systems, such as a recurrent neural network as the LSTM (Hochreiter and Schmidhuber, 1997) or a capsule network (Sabour, Frosst, and Hinton, 2018), to find out if the improved performance also applies in these kinds of systems. They could also be tested in systems with a different, and possibly more advanced, method than backpropagation for the calculation of the gradients.

Finally, the adapted version of SGDHB as already described in the discussion section of this algorithm could be tested to see if it would be an improvement of SGD with standard momentum.

Appendices

Appendix A

Optimal parameter settings in the experiments

This appendix contains for each experiment in this thesis a table with the optimal parameter settings for each algorithm. The method to obtain these values differs per experiment, and is described in detail in the corresponding section in the thesis.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.0021	-	-
SGDmom	0.002	0.9	-
SGDNAG	0.0017	0.9	-
SGDHB	0.002	0.01	-
Adagrad	0.95	-	-
AdaDelta	1.0	0.999999	-
RMSprop	0.0015	0.9	-
Adam	0.85	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
AdaMax	0.85	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
Nadam	0.43	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
NadaMax	0.23	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
AMSGrad	0.97	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
NAMSGrad	0.15	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
Absgrad	0.15	-	-
AbsDelta	1.0	0.9	-
Absprop	0.0015	0.9	-
AbsAdam	0.35	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
AbsNadam	0.08	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
AbsAMSGrad	0.007	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
AbsNAMSGrad	0.0035	$\gamma_1 = 0.9, \gamma_2 = 0.999$	-
$\ell_{1-4}-$Adagrad	0.2	-	$\Lambda_0 = 2, \lambda = 0.01, \Lambda_{max} = 3$
$\ell_{1-4}-$AdaDelta	1.0	0.999999	$\Lambda_0 = 3, \lambda = 0, \Lambda_{max} = 3$
$\ell_{1-4}-$RMSprop	0.0015	0.9	$\Lambda_0 = 2, \lambda = 0, \Lambda_{max} = 2$
$\ell_{1-4}-$Adam	0.38	$\gamma_1 = 0.9, \gamma_2 = 0.999$	$\Lambda_0 = 2, \lambda = 0.01, \Lambda_{max} = 3$
$\ell_{1-4}-$Nadam	0.33	$\gamma_1 = 0.9, \gamma_2 = 0.999$	$\Lambda_0 = 2, \lambda = 0.1, \Lambda_{max} = 3$
$\ell_{1-4}-$AMSGrad	0.27	$\gamma_1 = 0.9, \gamma_2 = 0.999$	$\Lambda_0 = 2, \lambda = 0.01, \Lambda_{max} = 3$
$\ell_{1-4}-$NAMSGrad	0.2	$\gamma_1 = 0.9, \gamma_2 = 0.999$	$\Lambda_0 = 1, \lambda = 0.01, \Lambda_{max} = 2$
ExpAdam	0.99	-	-
$\ell_{ada}-$Adam	0.93	-	$\Lambda_0 = 3, \gamma_P = 0.9$

TABLE A.1: Optimal parameter settings for all algorithms on the Rosenbrock function. Note that the smoothing parameter η used by all adaptive methods is constantly set to $1e - 08$. Also note that for AbsDelta, no parameter setting resulted in a convergence towards the minimum.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.99	-	-
SGDmom	0.85	0.9	-
SGDNAG	0.95	0.9	-
SGDHB	0.96 (0.9)	0.9	-
Adagrad	0.84 (0.2)	-	-
AdaDelta	1.0	0.9	-
RMSprop	0.04 (0.01)	0.99	-
Adam	0.03	-	-
AdaMax	0.01	-	-
Nadam	0.02	-	-
NadaMax	0.23	-	-
AMSGrad	0.05	-	-
NAMSGrad	0.03	-	-
Absgrad	0.95	-	-
AbsDelta	1.0	0.96 (0.9)	-
Absprop	0.25	0.99	-
AbsAdam	0.7	-	-
AbsNadam	0.28	-	-
AbsAMSGrad	0.55	-	-
AbsNAMSGrad	0.41	-	-
$\ell_{1-4}-$Adagrad	0.13	-	Λ_0 = 3, λ = -1E-06, Λ_{min} = 2
$\ell_{1-4}-$AdaDelta	1.0	0.9	Λ_0 = 3, λ = 0, Λ_{max} = 3
$\ell_{1-4}-$RMSprop	0.075 (0.02)	0.99	Λ_0 = 3, λ = 0, Λ_{max} = 3
$\ell_{1-4}-$Adam	0.12	-	Λ_0 = 3, λ = 0, Λ_{max} = 3
$\ell_{1-4}-$Nadam	0.02	-	Λ_0 = 3, λ = 0, Λ_{max} = 3
$\ell_{1-4}-$AMSGrad	0.05	-	Λ_0 = 3, λ = 0, Λ_{max} = 3
$\ell_{1-4}-$NAMSGrad	0.04	-	Λ_0 = 1 (3), λ = 1E-06 (0), Λ_{max} = 2 (0)
ExpAdam	0.003 (0.005)	-	-
$\ell_{ada}-$Adam	0.11 (0.12)	-	Λ_0 = 3

TABLE A.2: Optimal parameter settings for all algorithms on a network with 2 nodes in the hidden layer for the XOR problem. A few algorithms use a different learning rate in the networks with batch normalization, and $\ell_{1-4}-$NAMSGrad also uses a different power schedule. These are given between parentheses behind the parameter values for the network without batch normalization.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.98	-	-
SGDmom	0.99	0.9	-
SGDNAG	0.99	0.9	-
SGDHB	0.95	0.9	-
Adagrad	0.75 (0.35)	-	-
AdaDelta	1.0	0.9	-
RMSprop	0.12 (0.03)	0.99	-
Adam	0.05	-	-
AdaMax	0.16	-	-
Nadam	0.09	-	-
NadaMax	0.3 (0.15)	-	-
AMSGrad	0.11	-	-
NAMSGrad	0.08	-	-
Absgrad	0.99	-	-
AbsDelta	1.0	0.97	-
Absprop	0.85 (0.27)	0.99	-
AbsAdam	0.99	-	-
AbsNadam	0.99	-	-
AbsAMSGrad	0.99	-	-
AbsNAMSGrad	0.99	-	-
$\ell_{1-4}-$Adagrad	0.38	-	$\Lambda_0 = 3$, λ = -1E-06, $\Lambda_{min} = 2$
$\ell_{1-4}-$AdaDelta	1.0	0.9	$\Lambda_0 = 3$, $\lambda = 0$, $\Lambda_{max} = 3$
$\ell_{1-4}-$RMSprop	0.021 (0.07)	0.99	$\Lambda_0 = 1$, λ = 1E-06, $\Lambda_{max} = 2$
$\ell_{1-4}-$Adam	0.19	-	$\Lambda_0 = 3$, $\lambda = 0$, $\Lambda_{max} = 3$
$\ell_{1-4}-$Nadam	0.07	-	$\Lambda_0 = 3$, $\lambda = 0$, $\Lambda_{max} = 3$
$\ell_{1-4}-$AMSGrad	0.18	-	$\Lambda_0 = 3$, $\lambda = 0$, $\Lambda_{max} = 3$
$\ell_{1-4}-$NAMSGrad	0.08	-	$\Lambda_0 = 2$, $\lambda = 0$, $\Lambda_{max} = 2$
ExpAdam	0.01	-	-
$\ell_{ada}-$Adam	0.13 (0.9)	-	$\Lambda_0 = 3$

TABLE A.3: Optimal parameter settings for all algorithms on a network with 5 nodes in the hidden layer for the XOR problem. A few algorithms use a different learning rate in the networks with batch normalization. These are given between parentheses behind the parameter values for the network without batch normalization.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.99	-	-
SGDmom	0.25	0.9	-
SGDNAG	0.25	0.9	-
SGDHB	0.99	0.2	-
Adagrad	0.99	-	-
AdaDelta	1.0	0.99	-
RMSprop	0.01	0.99	-
Adam	0.01	-	-
AdaMax	0.04	-	-
Nadam	0.01	-	-
NadaMax	0.04	-	-
AMSGrad	0.01	-	-
NAMSGrad	0.01	-	-
Absgrad	0.99	-	-
AbsDelta	1.0	0.99	-
Absprop	0.12	0.99	-
AbsAdam	0.14	-	-
AbsNadam	0.13	-	-
AbsAMSGrad	0.12	-	-
AbsNAMSGrad	0.15	-	-
$\ell_{1-4}-$Adagrad	0.99	-	$\Lambda_0 = 1.9, \lambda = 0, \Lambda_{max} = 1.9$
$\ell_{1-4}-$AdaDelta	1.0	0.99	$\Lambda_0 = 2.5, \lambda = 0, \Lambda_{max} = 2.5$
$\ell_{1-4}-$RMSprop	0.01	0.99	$\Lambda_0 = 1.2, \lambda = 0, \Lambda_{max} = 1.2$
$\ell_{1-4}-$Adam	0.01	-	$\Lambda_0 = 1.1, \lambda = 0, \Lambda_{max} = 1.1$
$\ell_{1-4}-$Nadam	0.01	-	$\Lambda_0 = 1.1, \lambda = 0, \Lambda_{max} = 1.1$
$\ell_{1-4}-$AMSGrad	0.01	-	$\Lambda_0 = 1.1, \lambda = 0, \Lambda_{max} = 1.1$
$\ell_{1-4}-$NAMSGrad	0.01	-	$\Lambda_0 = 1.1, \lambda = 0, \Lambda_{max} = 1.1$
ExpAdam	0.01	-	-
$\ell_{ada}-$Adam	0.01	-	$\Lambda_0 = 1.2, \gamma_P = 0.9$

TABLE A.4: Optimal parameter settings for all algorithms using logistic regression on the MNIST data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.7	-	-
SGDmom	0.11	0.9	-
SGDNAG	0.12	0.9	-
SGDHB	0.4	0.9	-
Adam	0.002	-	-
AdaMax	0.015	-	-
Nadam	0.002	-	-
AMSGrad	0.004	-	-
AbsAdam	0.02	-	-
AbsAMSGrad	0.06	-	-
$\ell_{1-4}-$Adam	0.002	-	$\Lambda_0 = 4.0, \lambda = 0, \Lambda_{max} = 4.0$
$\ell_{1-4}-$AMSGrad	0.004	-	$\Lambda_0 = 1.5, \lambda = 0, \Lambda_{max} = 1.5$
ExpAdam	0.001	-	-
$\ell_{ada}-$Adam	0.002	-	$\Lambda_0 = 2.3$

TABLE A.5: Optimal parameter settings for all algorithms using an MLP with 1 hidden layer of 100 nodes on the MNIST data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.2	-	-
SGDmom	0.06	0.9	-
SGDNAG	0.04	0.9	-
SGDHB	0.065	0.9	-
Adam	0.0004	-	-
AdaMax	0.003	-	-
Nadam	0.0003	-	-
AMSGrad	0.002	-	-
AbsAdam	0.0065	-	-
$\ell_{1-4}-$Adam	0.0004	-	$\Lambda_0 = 3.1$, $\lambda = 0$, $\Lambda_{max} = 3.1$
ExpAdam	0.0002	-	-
$\ell_{ada}-$Adam	0.0004	-	$\Lambda_0 = 3.0$

TABLE A.6: Optimal parameter settings for all algorithms using an MLP with 1 hidden layer of 100 nodes and batch normalization on the MNIST data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.2	-	-
SGDmom	0.02	0.9	-
SGDNAG	0.015	0.9	-
SGDHB	0.01	0.9	-
Adam	0.001	-	-
AdaMax	0.005	-	-
Nadam	0.001	-	-
AMSGrad	0.002	-	-
AbsAdam	0.01	-	-
$\ell_{1-4}-$Adam	0.001	-	$\Lambda_0 = 1.9$, $\lambda = 0$, $\Lambda_{max} = 1.9$
ExpAdam	0.0001	-	-
$\ell_{ada}-$Adam	0.001	-	$\Lambda_0 = 2.8$

TABLE A.7: Optimal parameter settings for all algorithms using a convolutional neural network with 2 convolutional layers (second with dropout) followed by a fully connected hidden layer of 50 nodes and dropout on the MNIST data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.045	-	-
SGDmom	0.005	0.9	-
SGDNAG	0.007	0.9	-
SGDHB	0.02	0.9	-
Adam	0.0007	-	-
AdaMax	0.0022	-	-
Nadam	0.001	-	-
AMSGrad	0.0007	-	-
AbsAdam	0.006	-	-
AbsAMSGrad	0.006	-	-
$\ell_{1-4}-$Adam	0.0007	-	$\Lambda_0 = 1$, $\lambda = 0$, $\Lambda_{max} = 1$
$\ell_{1-4}-$AMSGrad	0.0007	-	$\Lambda_0 = 1.1$, $\lambda = 0$, $\Lambda_{max} = 1.1$
ExpAdam	0.0005	-	-
$\ell_{ada}-$Adam	0.0007	-	$\Lambda_0 = 1.3$, $\gamma_P = 0.9$

TABLE A.8: Optimal parameter settings for all algorithms using logistic regression on the CIFAR10 data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.1	-	-
SGDmom	0.015	0.9	-
SGDNAG	0.015	0.9	-
SGDHB	0.045	0.9	-
Adam	0.0009	-	-
AbsAdam	0.011	-	-
$\ell_{1-4}-$Adam	0.0009	-	Λ_0 = 1.2, λ = 0, Λ_{max} = 1.2
ExpAdam	0.0005	-	-
$\ell_{ada}-$Adam	0.0009	-	Λ_0 = 2.0, $\gamma_P = 0.9$
AdaMax	0.0025	-	-
Nadam	0.001	-	-
AbsNadam	0.011	-	-
$\ell_{1-4}-$Nadam	0.001	-	Λ_0 = 2.5, λ = 0, Λ_{max} = 2.5
$\ell_{ada}-$Nadam	0.001	-	Λ_0 = 2.5, $\gamma_P = 0.9$
AMSGrad	0.0008	-	-
AbsAMSGrad	0.011	-	-
$\ell_{1-4}-$AMSGrad	0.0008	-	Λ_0 = 1.3, λ = 0, Λ_{max} = 1.3

TABLE A.9: Optimal parameter settings for all algorithms using an MLP with 1 hidden layer of 200 nodes on the CIFAR10 data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.18	-	-
SGDmom	0.02	0.9	-
SGDNAG	0.02	0.9	-
SGDHB	0.05	0.9	-
Adam	0.0015	-	-
AbsAdam	0.015	-	-
$\ell_{1-4}-$Adam	0.0015	-	Λ_0 = 1.6, λ = 0, Λ_{max} = 1.6
ExpAdam	0.001	-	-
$\ell_{ada}-$Adam	0.0015	-	Λ_0 = 2.2, $\gamma_P = 0.9$
AdaMax	0.0025	-	-
Nadam	0.0015	-	-
AMSGrad	0.001	-	-

TABLE A.10: Optimal parameter settings for all algorithms using an MLP with 1 hidden layer of 200 nodes and batch normalization on the CIFAR10 data set.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.06	-	-
SGDmom	0.01	0.9	-
SGDNAG	0.02	0.9	-
SGDHB	0.04	0.9	-
Adam	0.003	-	-
AbsAdam	0.02	-	-
$\ell_{1-4}-$Adam	0.003	-	Λ_0 = 2.7, λ = 0, Λ_{max} = 2.7
ExpAdam	0.0003	-	-
$\ell_{ada}-$Adam	0.003	-	Λ_0 = 2.0, $\gamma_P = 0.9$
AdaMax	0.011	-	-
Nadam	0.002	-	-
AMSGrad	0.002	-	-

TABLE A.11: Optimal parameter settings for all algorithms using a convolutional neural network on the CIFAR10 data set. The CNN consists of a convolutional layer with 6 5x5 filters with ReLU and max-pooling, followed by a second convolutional layer with 16 5x5 filters with ReLU and max-pooling, and finally fully connected layer leading towards the ten output nodes.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.08	-	-
SGDmom	0.01	0.9	-
SGDNAG	0.01	0.9	-
SGDHB	0.05	0.9	-
Adam	0.004	-	-
AbsAdam	0.03	-	-
ℓ_{1-4}−Adam	0.004	-	$\Lambda_0 = 2.3$, $\lambda = 0$, $\Lambda_{max} = 2.3$
ExpAdam	0.0006	-	-
ℓ_{ada}−Adam	0.004	-	$\Lambda_0 = 2.3$, $\gamma_P = 0.9$
AdaMax	0.007	-	-
Nadam	0.003	-	-
AMSGrad	0.006	-	-

TABLE A.12: Optimal parameter settings for all algorithms using a convolutional neural network with batch normalization on the CIFAR10 data set. The CNN consists of a convolutional layer with 6 5x5 filters with ReLU and max-pooling, followed by a second convolutional layer with 16 5x5 filters with ReLU and max-pooling, and finally fully connected layer leading towards the ten output nodes.

Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.22	-	-
SGDmom	0.02	0.9	-
SGDNAG	0.025	0.9	-
SGDHB	0.085	0.9	-
Adam	0.0007	-	-
AbsAdam	0.012	-	-
ℓ_{1-4}−Adam	0.0007	-	$\Lambda_0 = 2.6$, $\lambda = 0$, $\Lambda_{max} = 2.6$
ExpAdam	0.0002	-	-
ℓ_{ada}−Adam	0.0007	-	$\Lambda_0 = 2.15$, $\gamma_P = 0.9$
AdaMax	0.0029	-	-
Nadam	0.0008	-	-
AMSGrad	0.0009	-	-

TABLE A.13: Optimal parameter settings for all algorithms using an MLP with 3 hidden layers of 500, 200, and 50 nodes on the CIFAR10 data set.

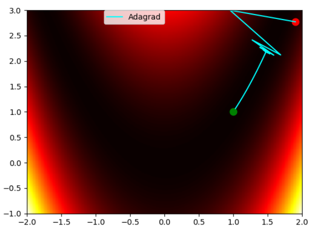
Algorithm	**Learning rate (α)**	**Momentum (γ)**	**Other parameters**
SGD	0.06	-	-
SGDmom	0.01	0.9	-
SGDNAG	0.01	0.9	-
SGDHB	0.02	0.9	-
Adam	0.0002	-	-
AbsAdam	0.0036	-	-
ℓ_{1-4}−Adam	0.0002	-	$\Lambda_0 = 2.6$, $\lambda = 0$, $\Lambda_{max} = 2.6$
ExpAdam	0.0005	-	-
ℓ_{ada}−Adam	0.0002	-	$\Lambda_0 = 1.7$, $\gamma_P = 0.9$
AdaMax	0.0009	-	-
Nadam	0.001	-	-
AMSGrad	0.0005	-	-

TABLE A.14: Optimal parameter settings for all algorithms using an MLP with 3 hidden layers of 500, 200, and 50 nodes and batch normalization on the CIFAR10 data set.

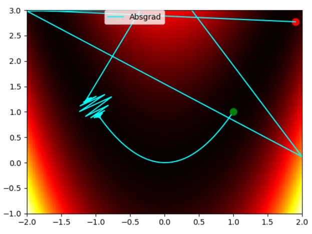
Appendix B

Performance plots on the Rosenbrock function

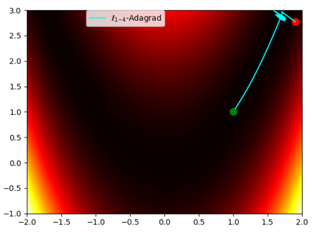
This appendix includes for every algorithm considered in the experiment on the Rosenbrock function a plot with the trajectory of the algorithm from the point (1.9,2.8) towards the end point, which is hopefully the minimum point in the plot at location (1,1).



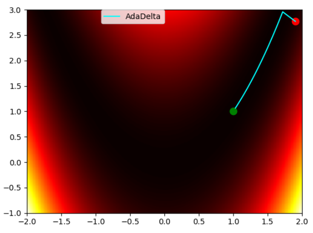
(A) Adagrad (5851)



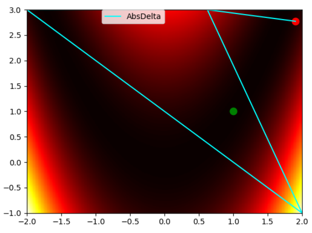
(B) Absgrad (3180)



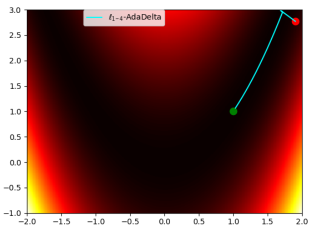
(C) $\ell_{1-4}-$Adagrad (7559)



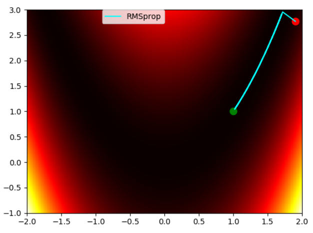
(D) AdaDelta (18855)



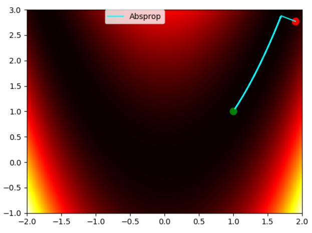
(E) AbsDelta ($\geq$30000)



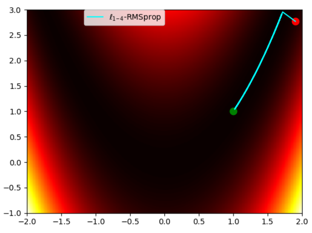
(F) $\ell_{1-4}-$AdaDelta (9691)



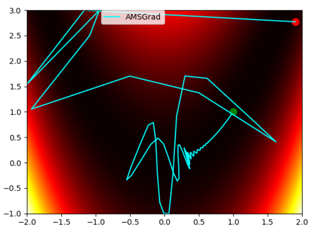
(G) RMSprop (5569)



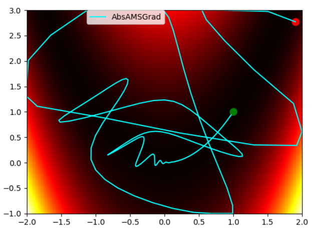
(H) Absprop (6282)



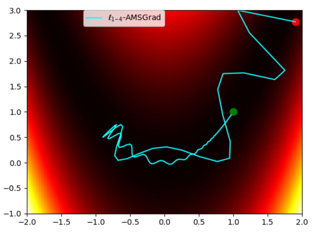
(I) $\ell_{1-4}-$RMSprop (5569)



(J) AMSGrad (380)

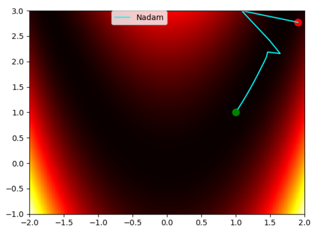


(K) AbsAMSGrad (4382)

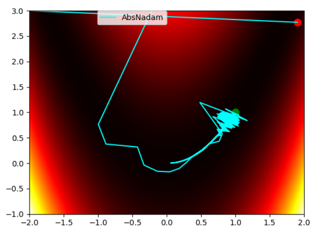


(L) $\ell_{1-4}-$AMSGrad (253)

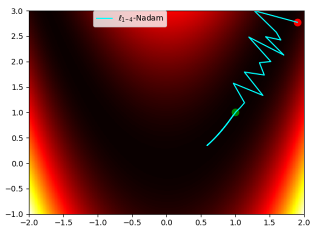
FIGURE B.1: Trajectories of the different algorithms on the Rosenbrock function (part 1). Note that they all start at the same point, at around (1.9, 2.8). The number of steps for each algorithm is between brackets in the captions.



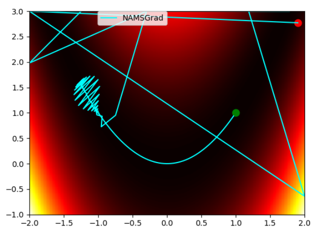
(A) Nadam (516)



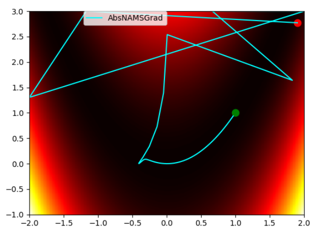
(B) AbsNadam (2251)



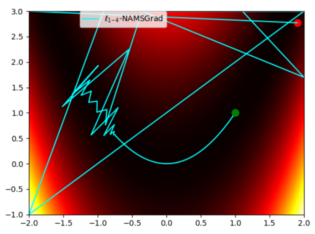
(C) ℓ_{1-4}—Nadam (418)



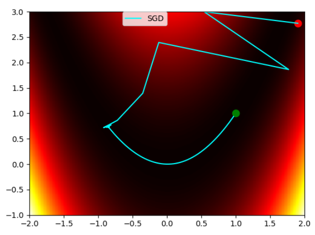
(D) NAMSGrad (1684)



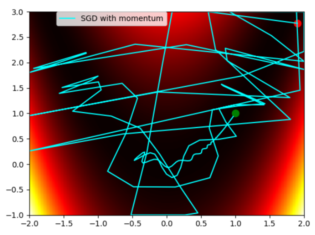
(E) AbsNAMSGrad (1818)



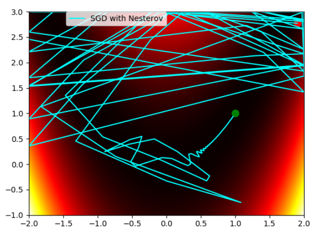
(F) ℓ_{1-4}—NAMSGrad (1356)



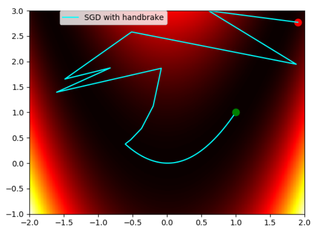
(G) SGD (3027)



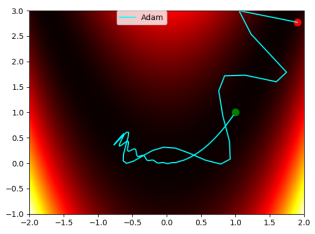
(H) SGD with momentum (344)



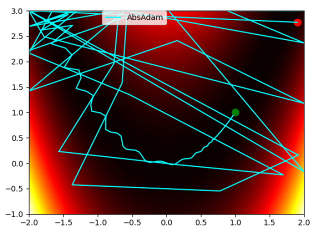
(I) SGD with Nesterov (358)



(J) SGD handbrake mom. (3037)

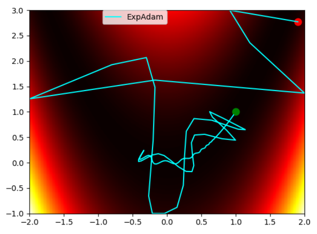


(K) Adam (620)

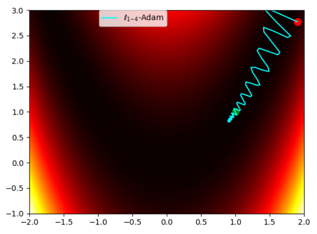


(L) AbsAdam (297)

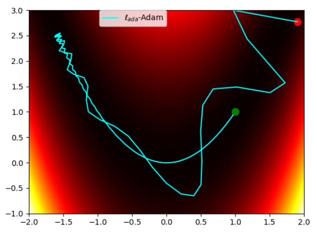
FIGURE B.2: Trajectories of the different algorithms on the Rosenbrock function (part 2). Note that they all start at the same point, at around (1.9, 2.8). The number of steps for each algorithm is between brackets in the captions.



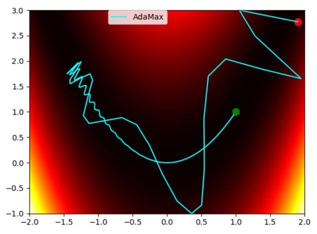
(A) ExpAdam (366)



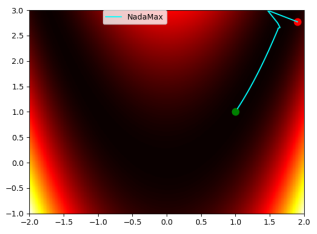
(B) $\ell_{1-4}-$Adam (144)



(C) $\ell_{ada}-$Adam (379)



(D) AdaMax (500)



(E) NadaMax (676)

FIGURE B.3: Trajectories of the different algorithms on the Rosenbrock function (part 3). Note that they all start at the same point, at around (1.9, 2.8). The number of steps for each algorithm is between brackets in the captions.

Appendix C

Cost plots of XOR experiments

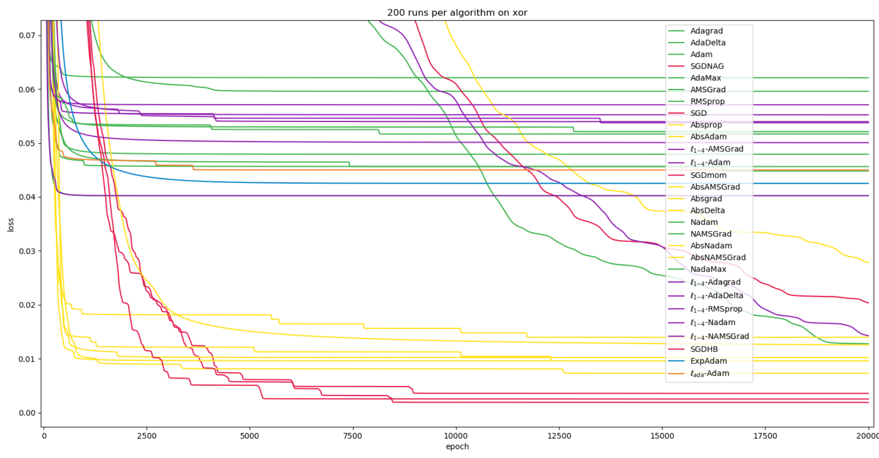


FIGURE C.1: The average cost reached per iteration over the training set for all optimization algorithms on the network with 2 nodes in the hidden layer.

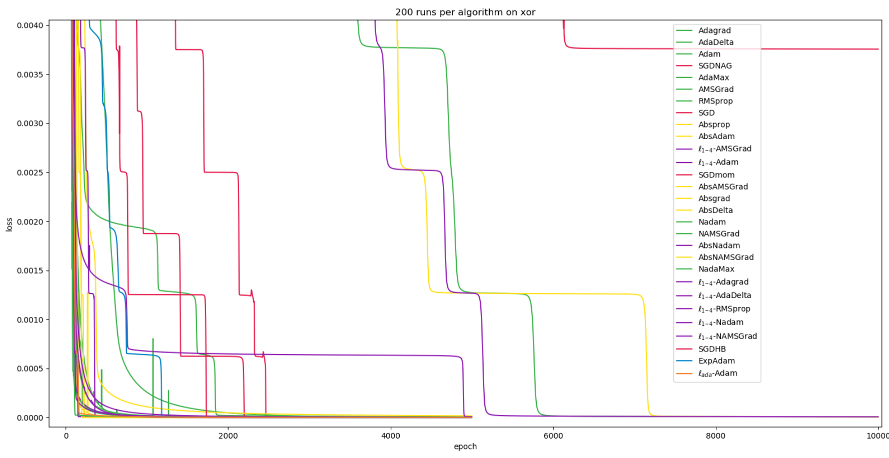


FIGURE C.2: The average cost reached per iteration over the training set for all optimization algorithms on the network with 2 nodes in the hidden layer and batch normalization.

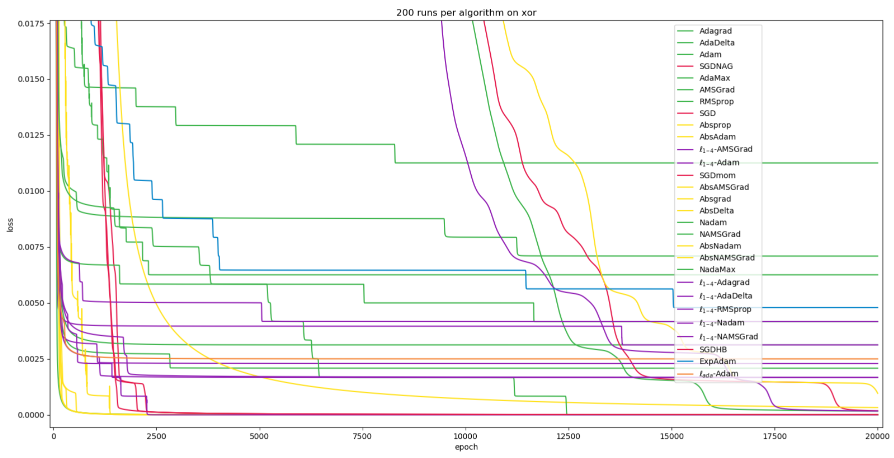


FIGURE C.3: The average cost reached per iteration over the training set for all optimization algorithms on the network with 5 nodes in the hidden layer.

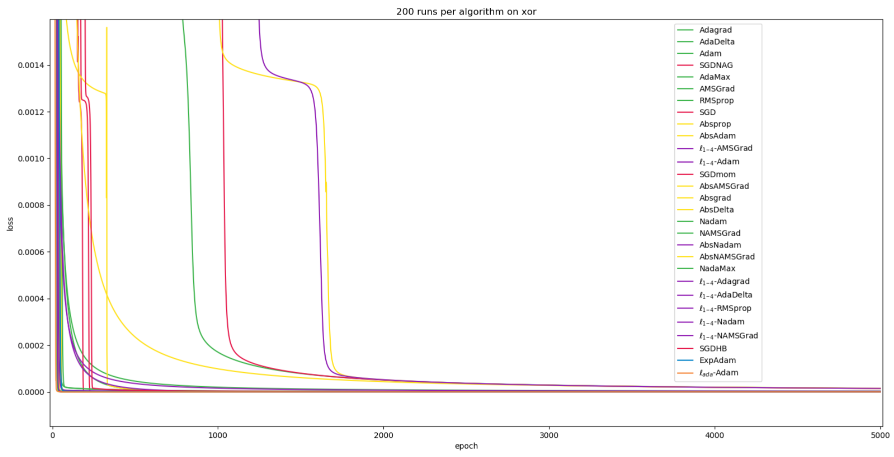
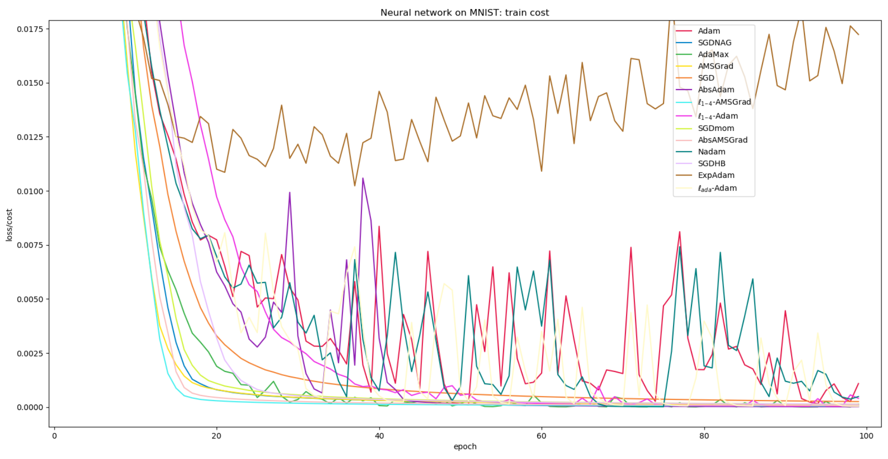


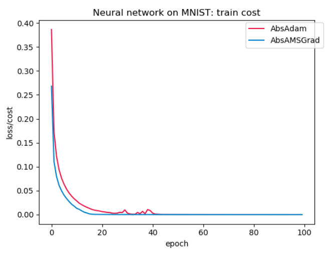
FIGURE C.4: The average cost reached per iteration over the training set for all optimization algorithms on the network with 5 nodes in the hidden layer and batch normalization.

Appendix D

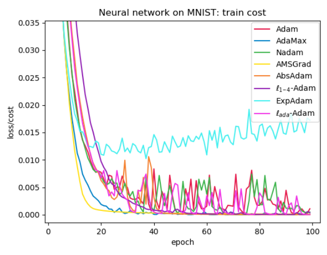
Cost plots of MNIST experiments



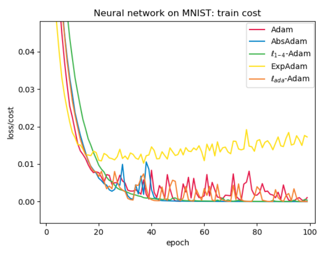
(A) All algorithms



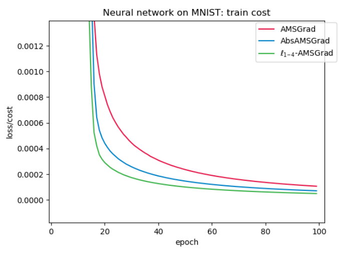
(B) Abs algorithms



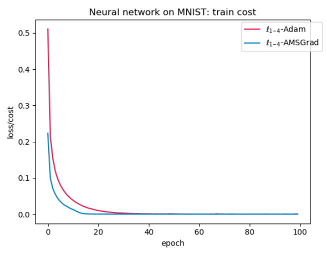
(C) Adaptive learning rate algorithms



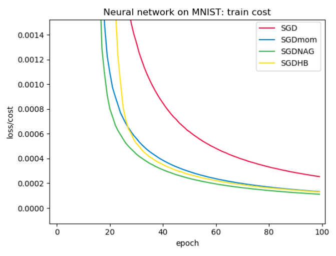
(D) Adam algorithms



(E) AMSGrad algorithms

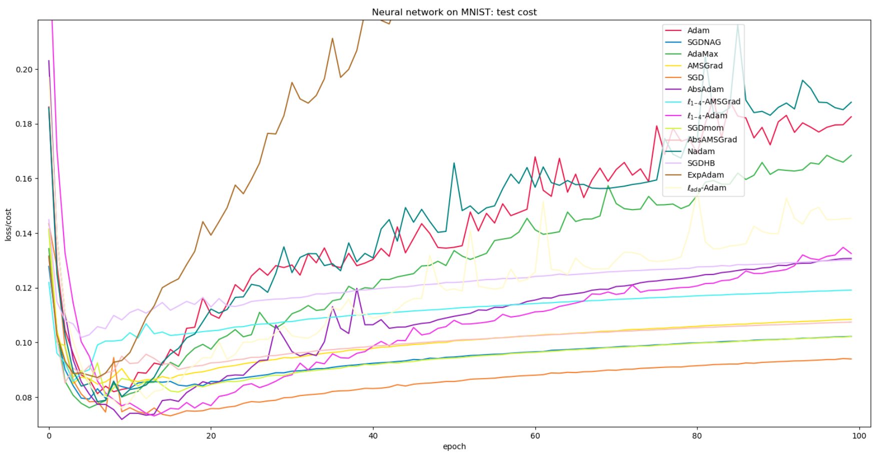


(F) ℓ_{1-4}-algorithms

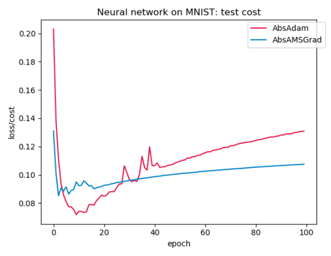


(G) SGD algorithms

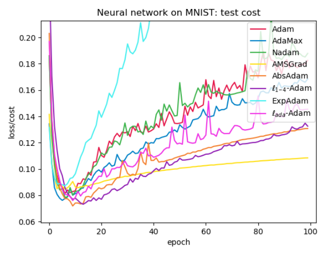
FIGURE D.1: Plots of the performance of algorithms in various groups on the experiment with an MLP with 1 hidden layer on MNIST. The plots show the average training cost per iteration, averaged over 5 runs.



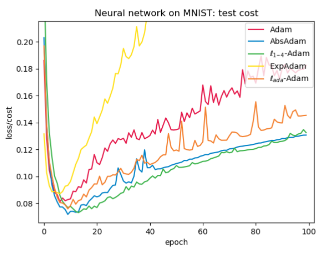
(A) All algorithms



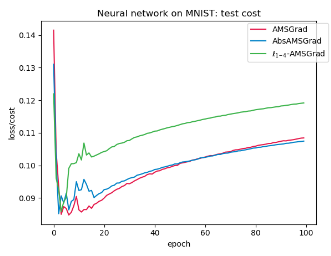
(B) Abs algorithms



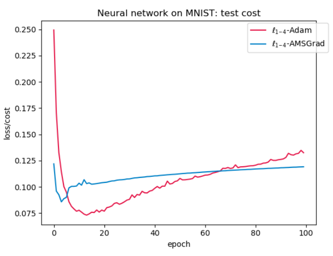
(C) Adaptive learning rate algorithms



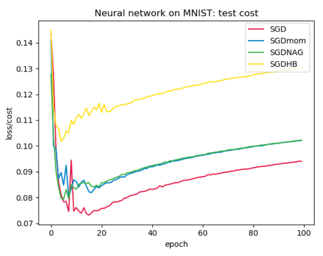
(D) Adam algorithms



(E) AMSGrad algorithms

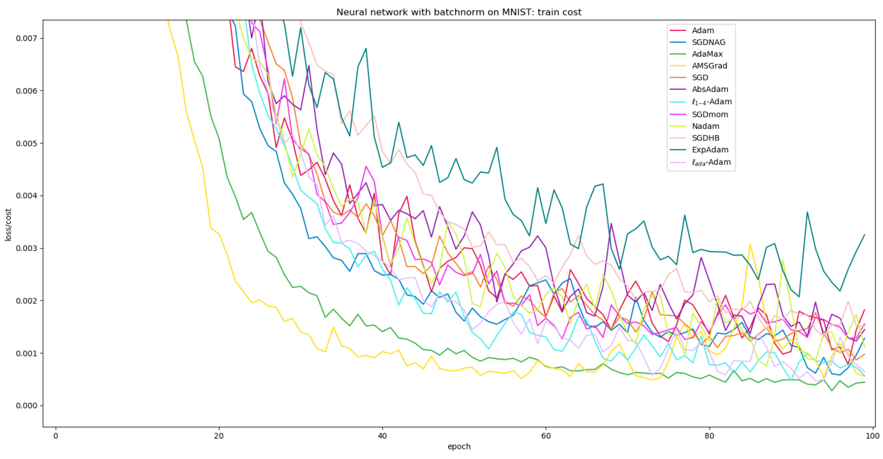


(F) ℓ_{1-4}-algorithms

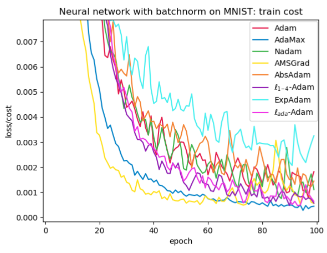


(G) SGD algorithms

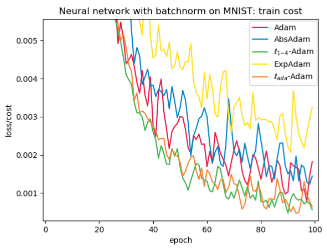
FIGURE D.2: Plots of the performance of algorithms in various groups on the experiment with an MLP with 1 hidden layer on MNIST. The plots show the average test cost per iteration, averaged over 5 runs.



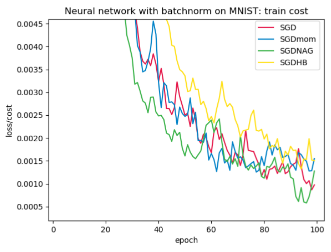
(A) All algorithms



(B) Adaptive learning rate algorithms

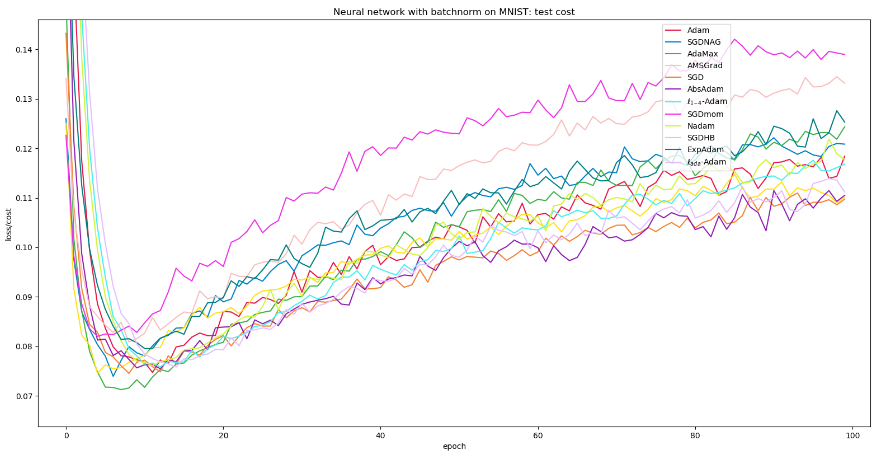


(C) Adam algorithms

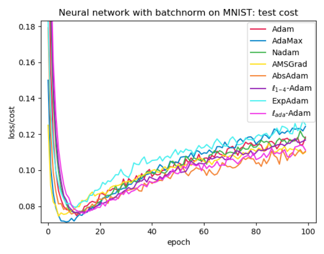


(D) SGD algorithms

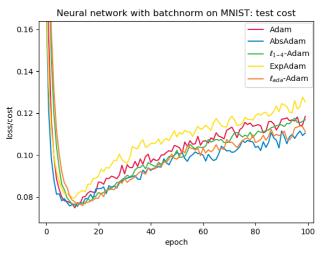
FIGURE D.3: Plots of the performance of algorithms in various groups on the experiment with an MLP with 1 hidden layer and batch normalization on MNIST. The plots show the average training cost per iteration, averaged over 5 runs.



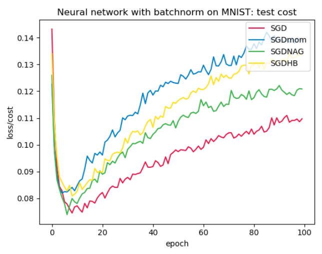
(A) All algorithms



(B) Adaptive learning rate algorithms

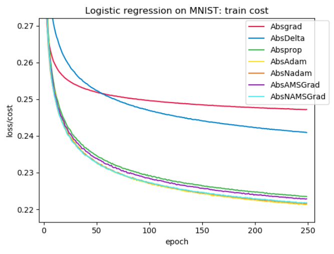


(C) Adam algorithms

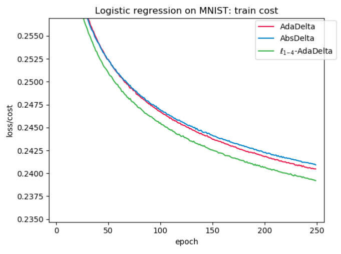


(D) SGD algorithms

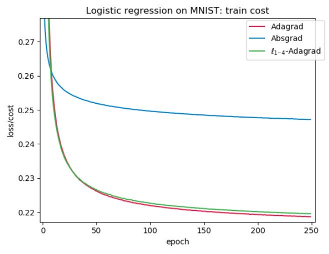
FIGURE D.4: Plots of the performance of algorithms in various groups on the experiment with an MLP with 1 hidden layer and batch normalization on MNIST. The plots show the average test cost per iteration, averaged over 5 runs.



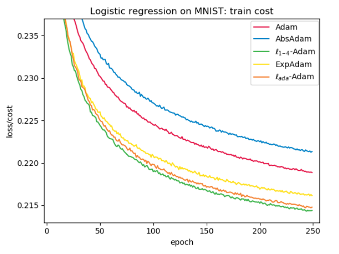
(A) Abs algorithms



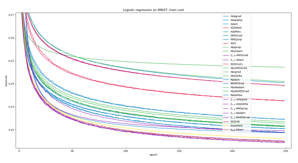
(B) AdaDelta algorithms

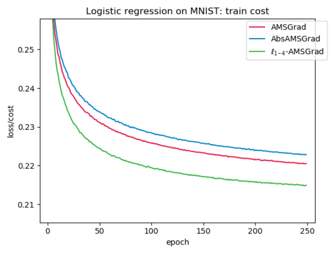


(C) Adagrad algorithms

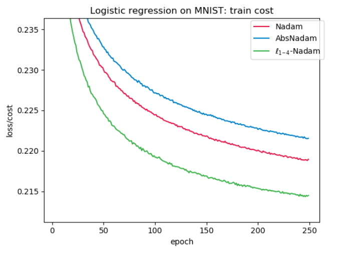


(D) Adam algorithms

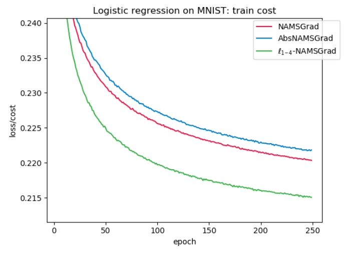

(E) All algorithms



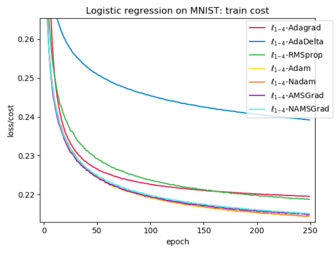
(F) AMSGrad algorithms



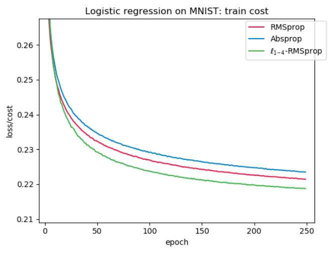
(G) Nadam algorithms



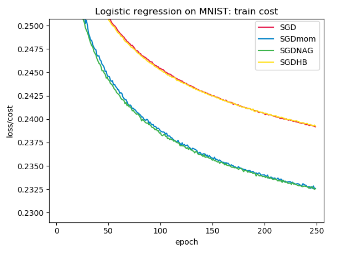
(H) NAMSGrad algorithms



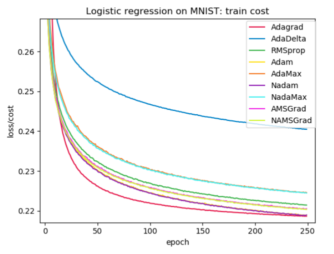
(I) ℓ_{1-4} algorithms



(J) RMSprop algorithms

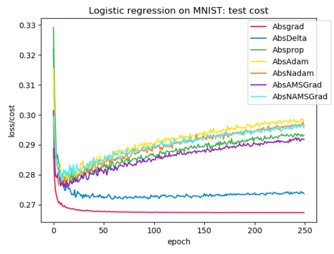


(K) SGD algorithms

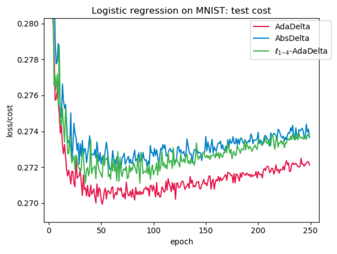


(L) Std adaptive algorithms

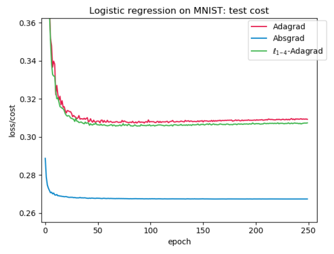
FIGURE D.5: Plots of the performance of algorithms in various groups on the logistic regression experiment on MNIST. The plots show the average training cost per iteration, averaged over 5 runs.



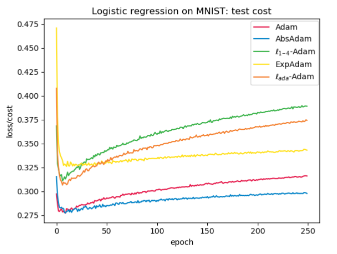
(A) Abs algorithms



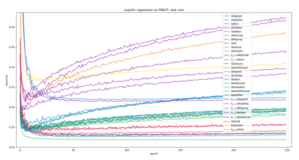
(B) AdaDelta algorithms



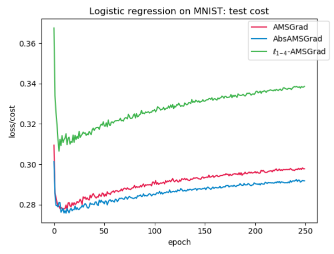
(C) Adagrad algorithms



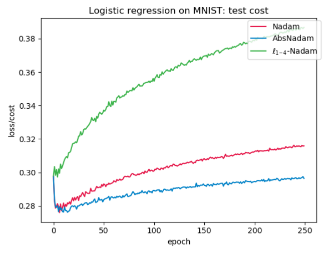
(D) Adam algorithms



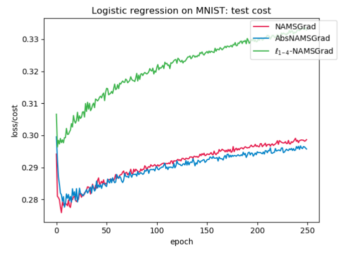
(E) All algorithms



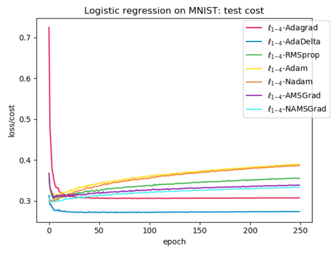
(F) AMSGrad algorithms



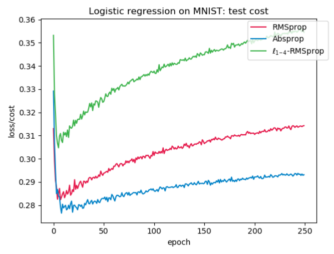
(G) Nadam algorithms



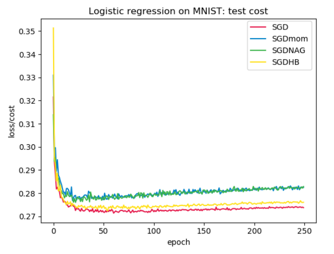
(H) NAMSGrad algorithms



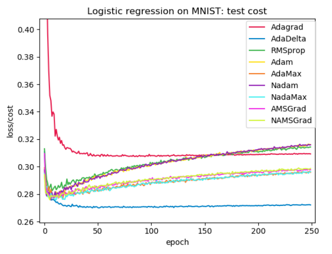
(I) ℓ_{1-4} algorithms



(J) RMSprop algorithms

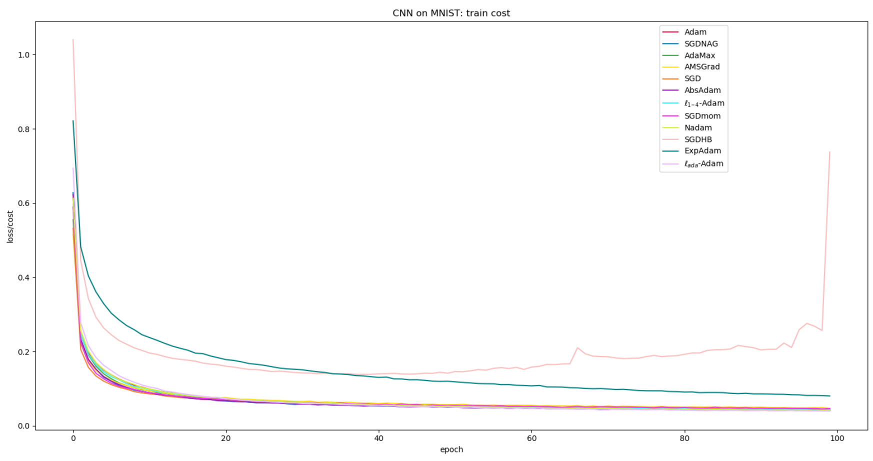


(K) SGD algorithms

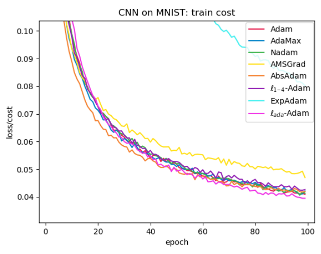


(L) Std adaptive algorithms

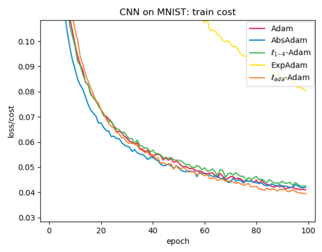
FIGURE D.6: Plots of the performance of algorithms in various groups on the logistic regression experiment on MNIST. The plots show the average test cost per iteration, averaged over 5 runs.



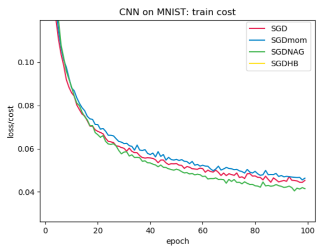
(A) All algorithms



(B) Adaptive learning rate algorithms

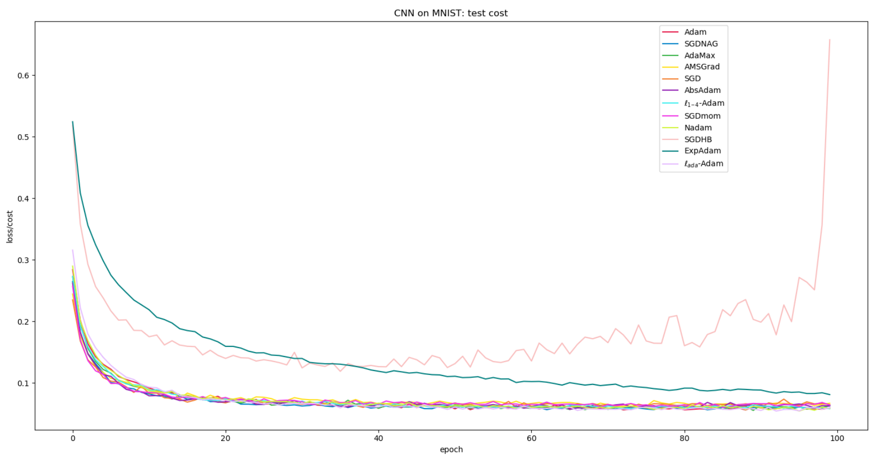


(C) Adam algorithms

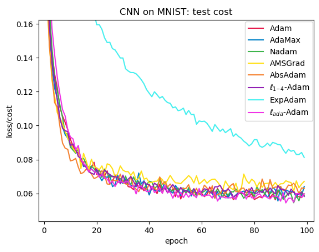


(D) SGD algorithms

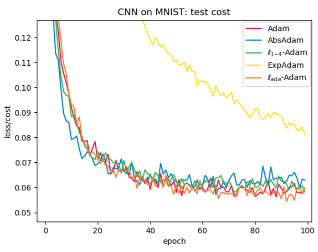
FIGURE D.7: Plots of the performance of algorithms in various groups on the experiment with a convolutional neural network on MNIST. The plots show the average training cost per iteration, averaged over 5 runs.



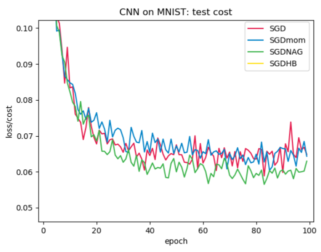
(A) All algorithms



(B) Adaptive learning rate algorithms



(C) Adam algorithms

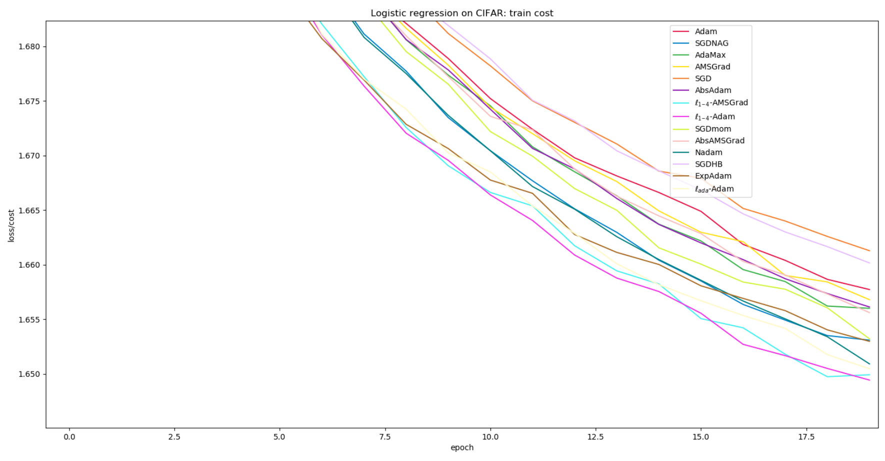


(D) SGD algorithms

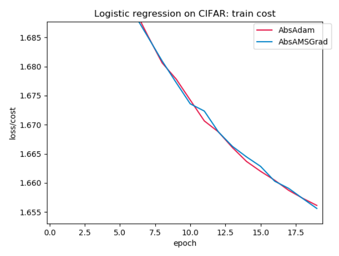
FIGURE D.8: Plots of the performance of algorithms in various groups on the experiment with a convolutional neural network on MNIST. The plots show the average test cost per iteration, averaged over 5 runs.

Appendix E

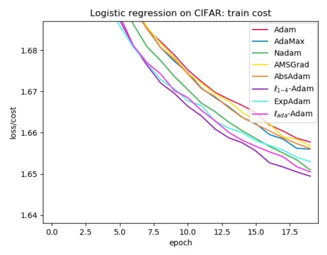
Cost plots of CIFAR10 experiments



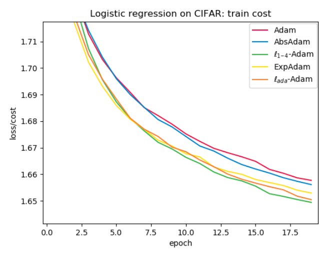
(A) All algorithms



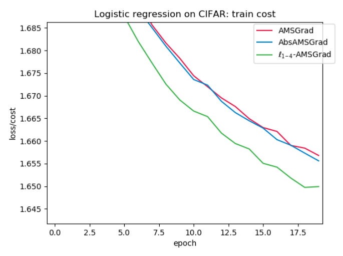
(B) Abs algorithms



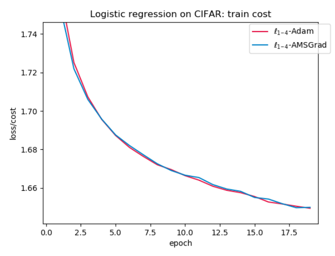
(C) Adaptive learning rate algorithms



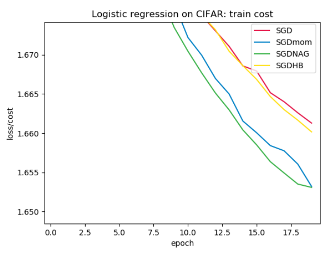
(D) Adam algorithms



(E) AMSGrad algorithms

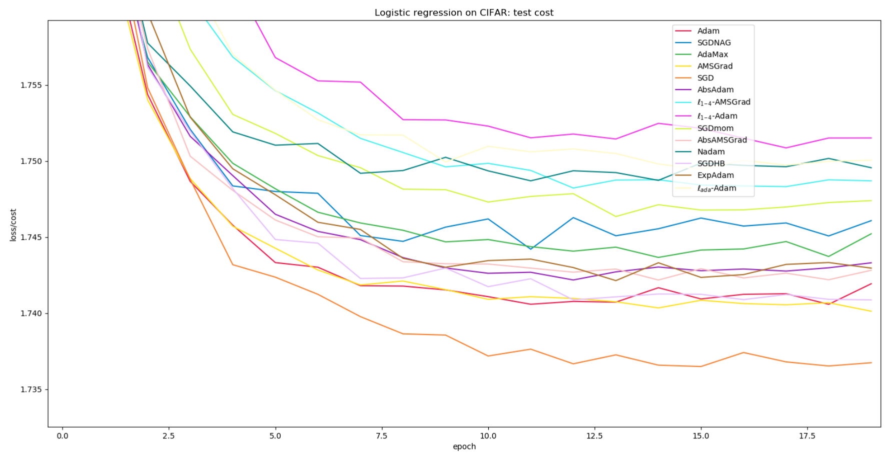


(F) ℓ_{1-4}-algorithms

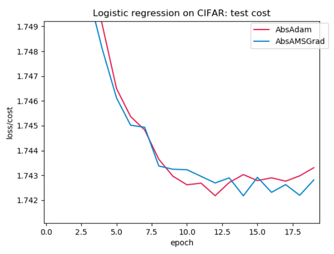


(G) SGD algorithms

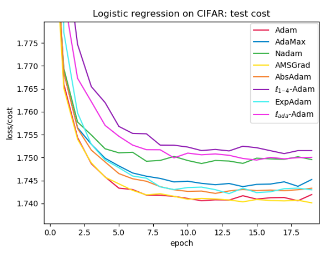
FIGURE E.1: Plots of the performance of algorithms in various groups on the logistic regression experiment on CIFAR10. The plots show the average training cost per iteration, averaged over 5 runs.



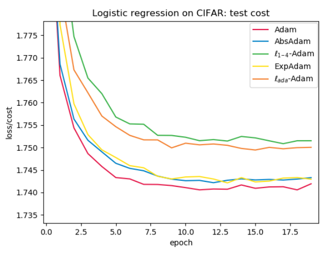
(A) All algorithms



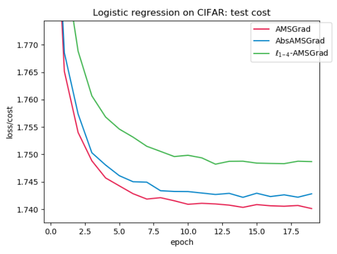
(B) Abs algorithms



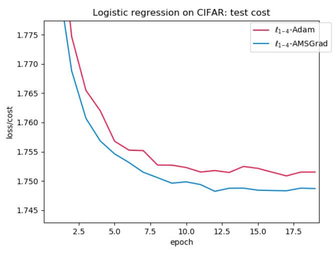
(C) Adaptive learning rate algorithms



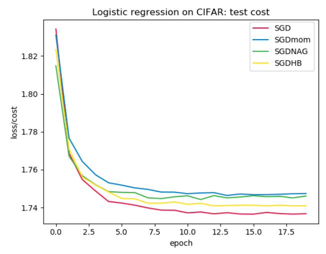
(D) Adam algorithms



(E) AMSGrad algorithms

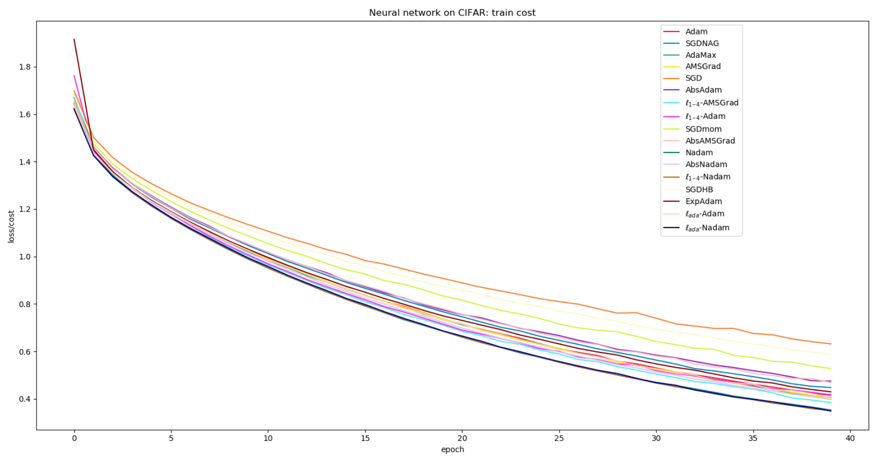


(F) ℓ_{1-4}-algorithms

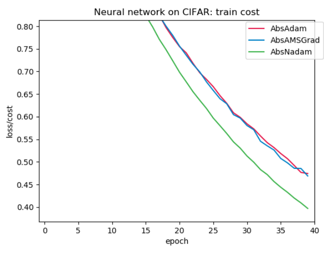


(G) SGD algorithms

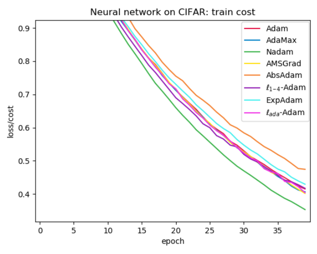
FIGURE E.2: Plots of the performance of algorithms in various groups on the logistic regression experiment on CIFAR10. The plots show the average test cost per iteration, averaged over 5 runs.



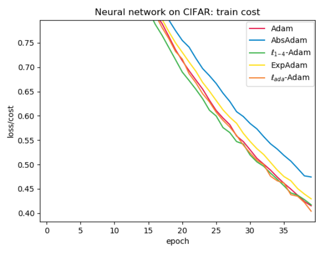
(A) All algorithms



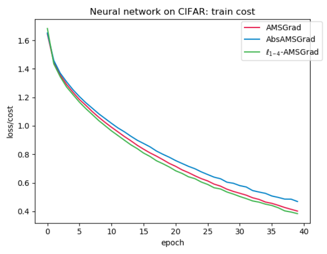
(B) Abs algorithms



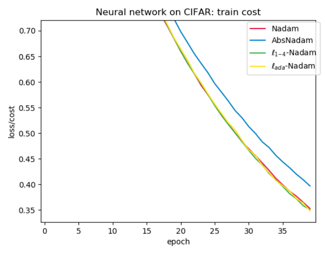
(C) Adaptive learning rate algorithms



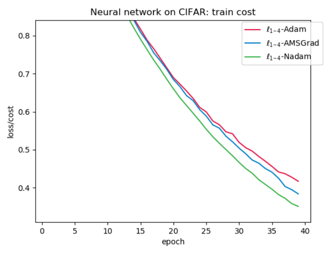
(D) Adam algorithms



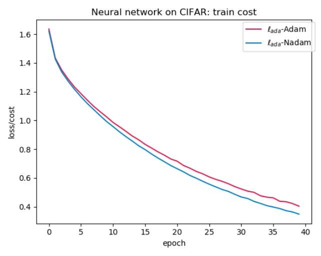
(E) AMSGrad algorithms



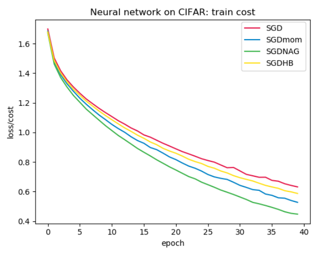
(F) Nadam algorithms



(G) ℓ_{1-4} algorithms

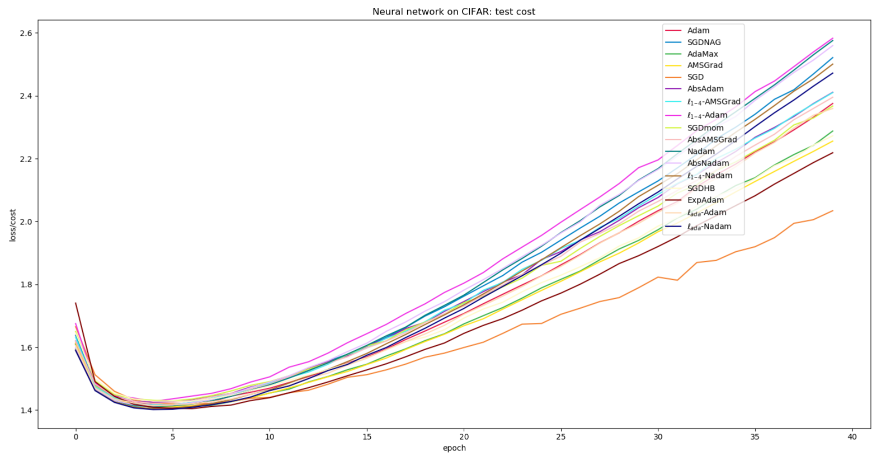


(H) ℓ_{ada} algorithms

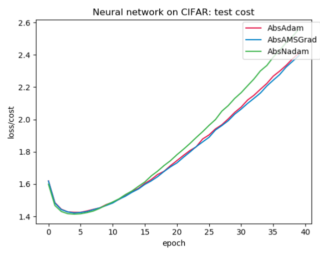


(I) SGD algorithms

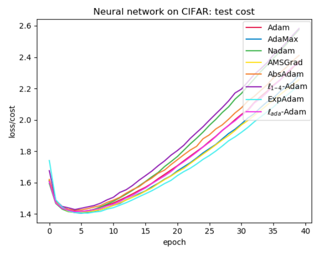
FIGURE E.3: Plots of the performance of algorithms in various groups on the experiment with an MLP containing a single hidden layer on the CIFAR10 data set. The plots show the average training cost per iteration, averaged over 5 runs.



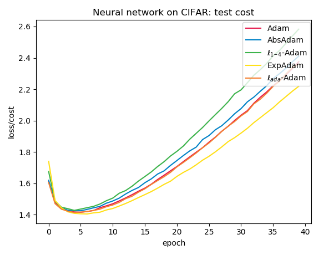
(A) All algorithms



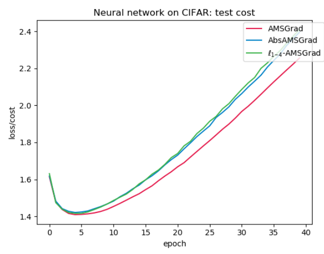
(B) Abs algorithms



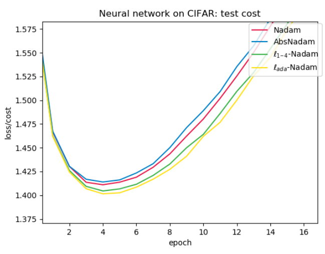
(C) Adaptive learning rate algorithms



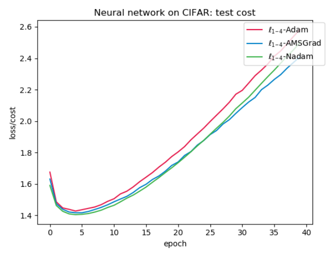
(D) Adam algorithms



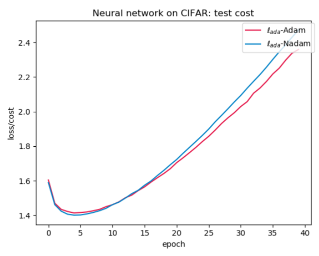
(E) AMSGrad algorithms



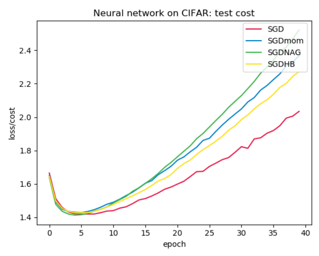
(F) Nadam algorithms



(G) ℓ_{1-4} algorithms

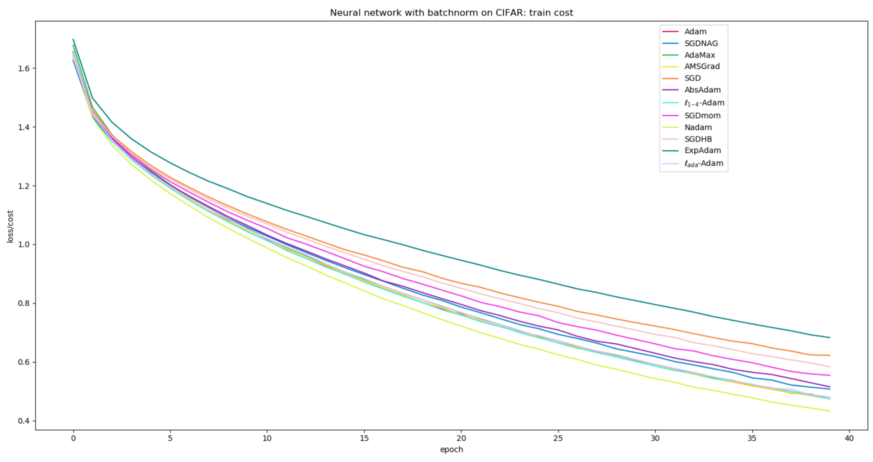


(H) ℓ_{ada} algorithms

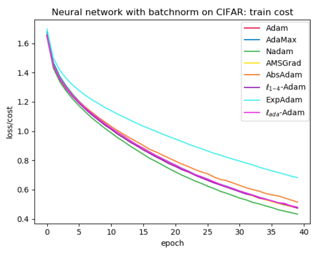


(I) SGD algorithms

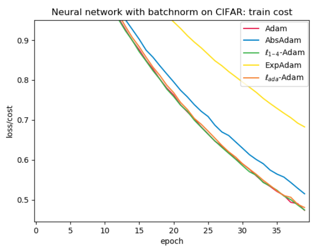
FIGURE E.4: Plots of the performance of algorithms in various groups on the experiment with an MLP containing a single hidden layer on the CIFAR10 data set. The plots show the average test cost per iteration, averaged over 5 runs.



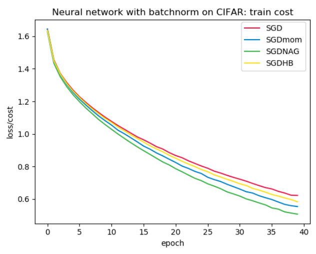
(A) All algorithms



(B) Adaptive learning rate algorithms

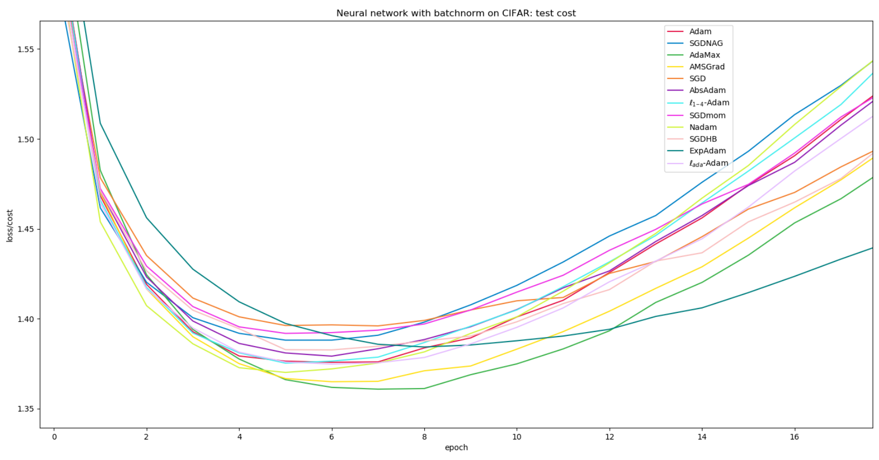


(C) Adam algorithms

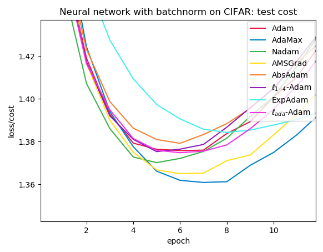


(D) SGD algorithms

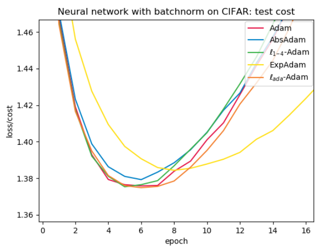
FIGURE E.5: Plots of the performance of algorithms in various groups on the experiment with an MLP with 1 hidden layer and batch normalization on CIFAR10. The plots show the average training cost per iteration, averaged over 5 runs.



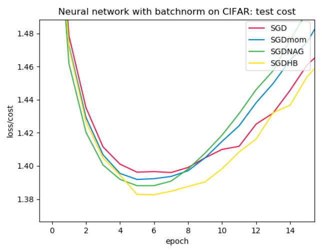
(A) All algorithms



(B) Adaptive learning rate algorithms

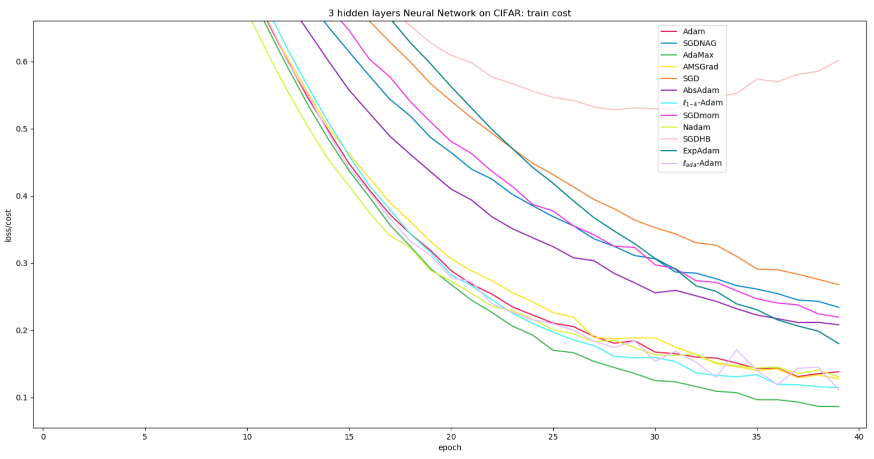


(C) Adam algorithms

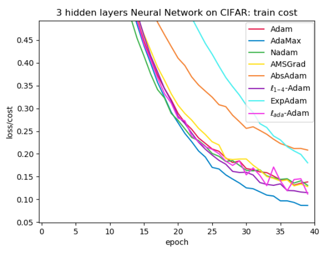


(D) SGD algorithms

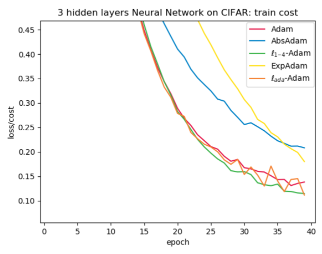
FIGURE E.6: Plots of the performance of algorithms in various groups on the experiment with an MLP with 1 hidden layer and batch normalization on CIFAR10. The plots show the average test cost per iteration, averaged over 5 runs.



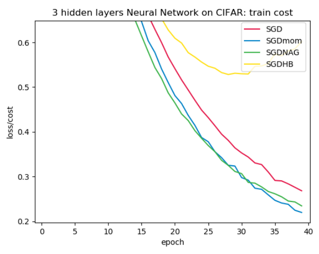
(A) All algorithms



(B) Adaptive learning rate algorithms

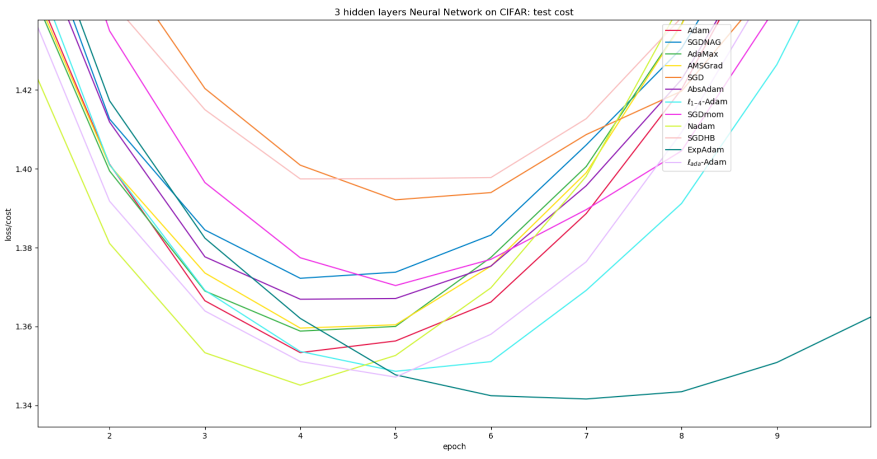


(C) Adam algorithms

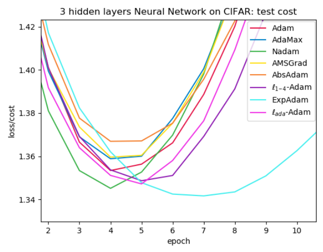


(D) SGD algorithms

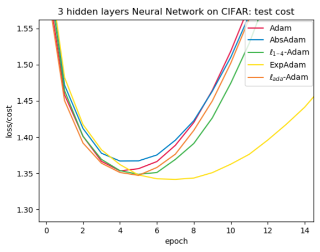
FIGURE E.7: Plots of the performance of algorithms in various groups on the experiment with an MLP with 3 hidden layers on CIFAR10. The plots show the average training cost per iteration, averaged over 5 runs.



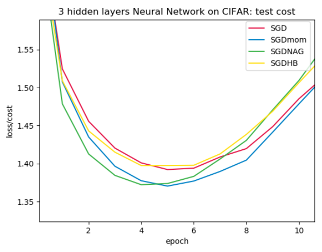
(A) All algorithms



(B) Adaptive learning rate algorithms

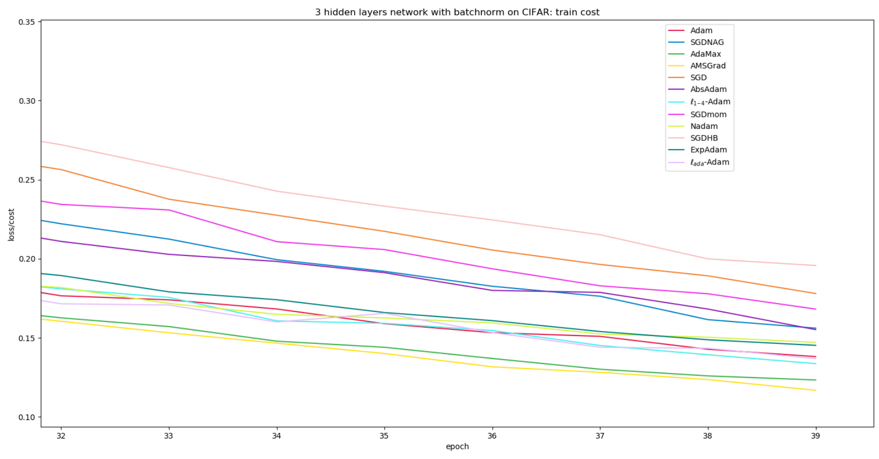


(C) Adam algorithms

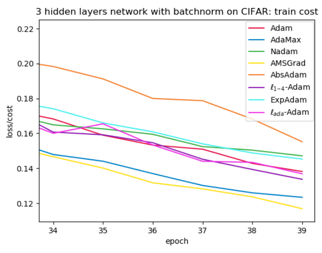


(D) SGD algorithms

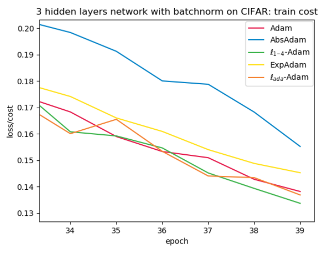
FIGURE E.8: Plots of the performance of algorithms in various groups on the experiment with an MLP with 3 hidden layers on CIFAR10. The plots show the average test cost per iteration, averaged over 5 runs.



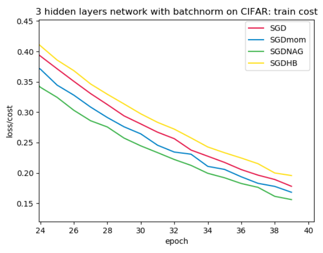
(A) All algorithms



(B) Adaptive learning rate algorithms

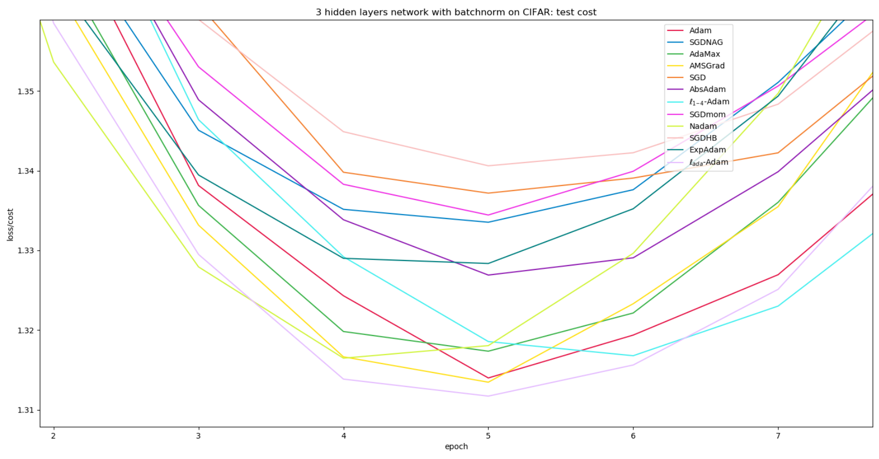


(C) Adam algorithms

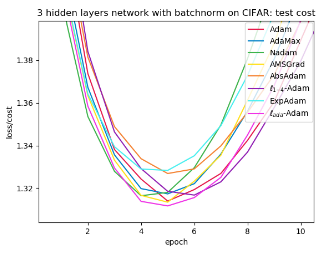


(D) SGD algorithms

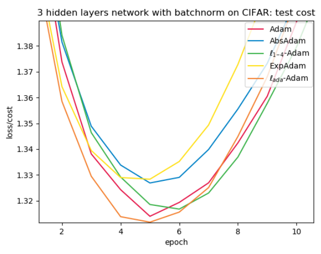
FIGURE E.9: Plots of the performance of algorithms in various groups on the experiment with an MLP with 3 hidden layers and batch normalization on CIFAR10. The plots show the average training cost per iteration, averaged over 5 runs.



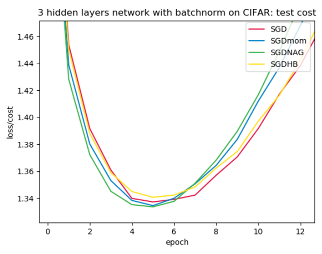
(A) All algorithms



(B) Adaptive learning rate algorithms

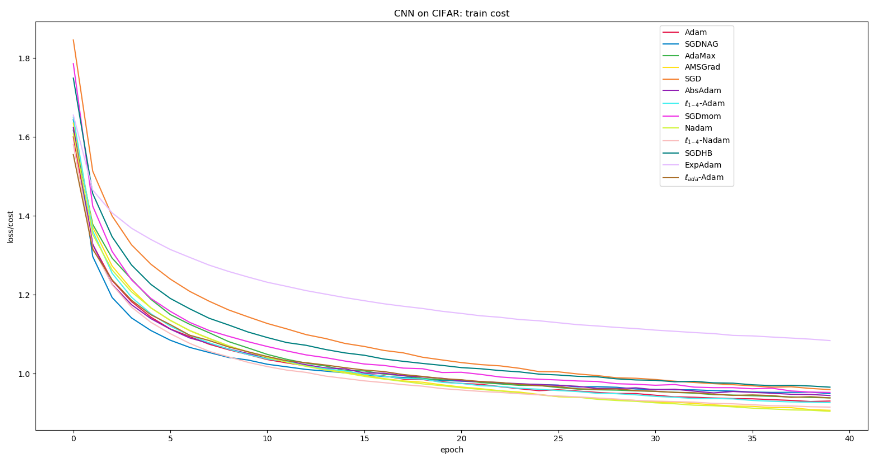


(C) Adam algorithms

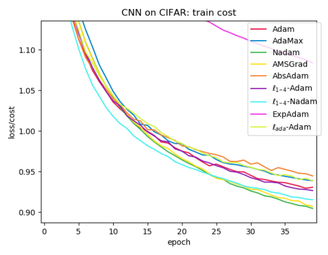


(D) SGD algorithms

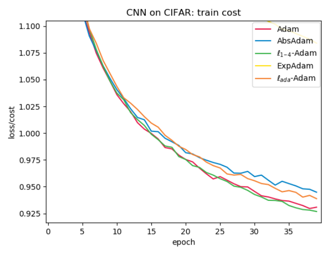
FIGURE E.10: Plots of the performance of algorithms in various groups on the experiment with a neural network with 3 hidden layers and batch normalization on CIFAR10. The plots show the average test cost per iteration, averaged over 5 runs.



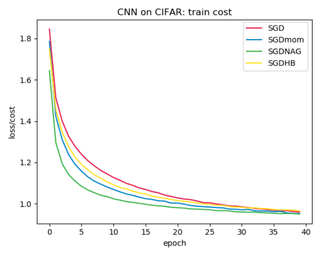
(A) All algorithms



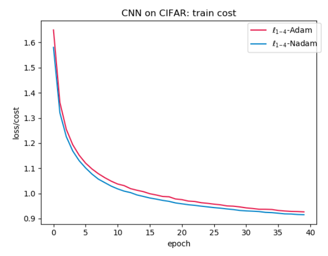
(B) Adaptive learning rate algorithms



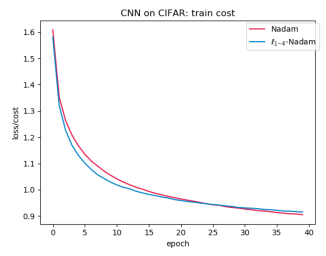
(C) Adam algorithms



(D) SGD algorithms

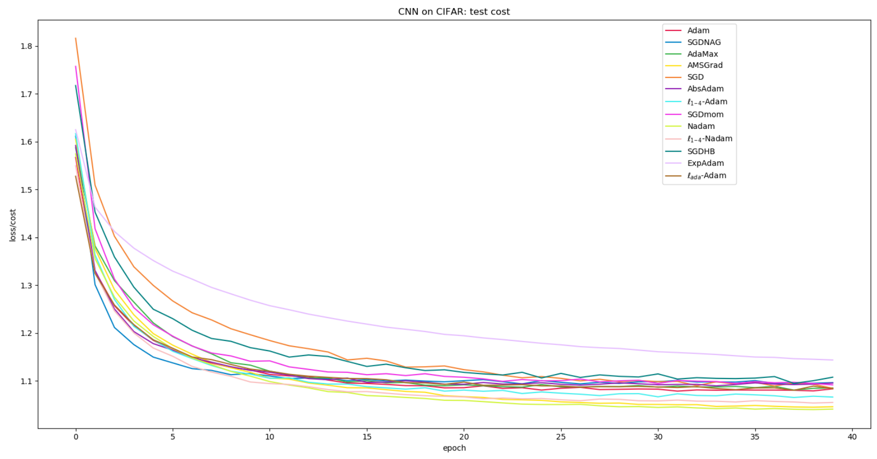


(E) ℓ_{1-4}-algorithms

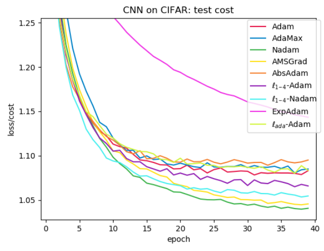


(F) Nadam algorithms

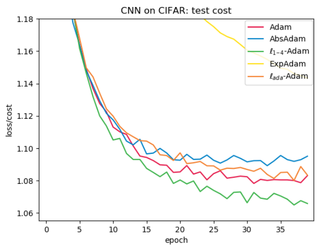
FIGURE E.11: Plots of the performance of algorithms in various groups on the experiment with a convolutional neural network on CIFAR10. The plots show the average training cost per iteration, averaged over 5 runs.



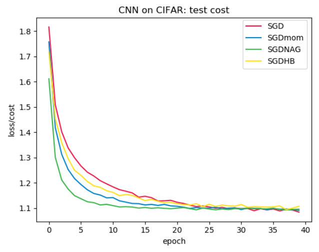
(A) All algorithms



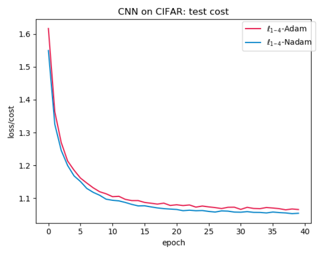
(B) Adaptive learning rate algorithms



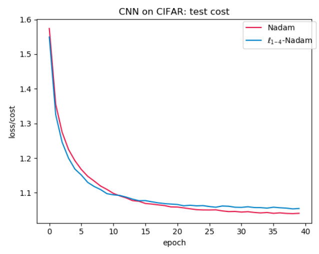
(C) Adam algorithms



(D) SGD algorithms

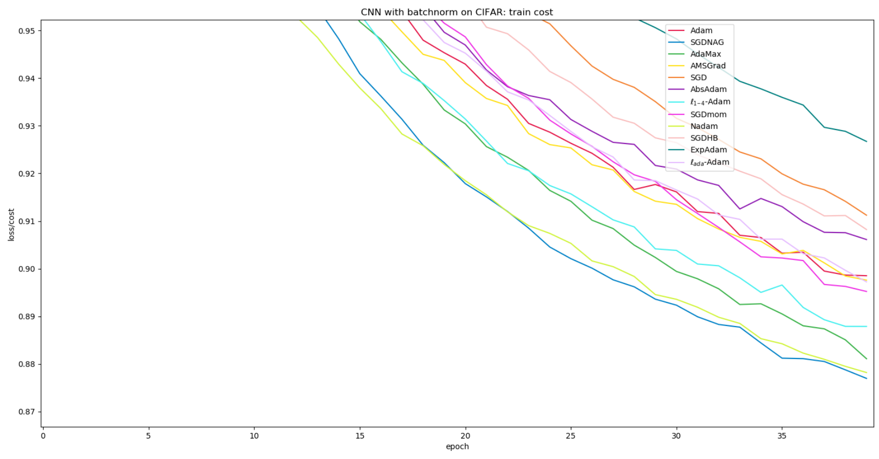


(E) ℓ_{1-4}-algorithms

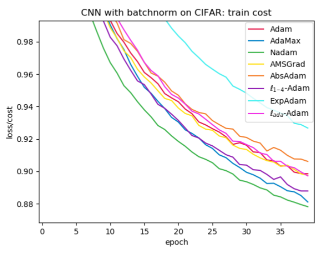


(F) Nadam algorithms

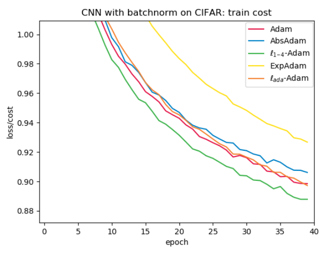
FIGURE E.12: Plots of the performance of algorithms in various groups on the experiment with a convolutional neural network on CIFAR10. The plots show the average test cost per iteration, averaged over 5 runs.



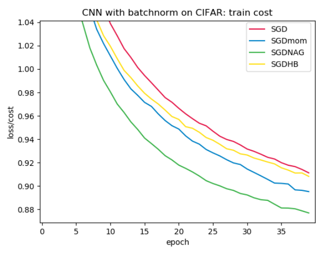
(A) All algorithms



(B) Adaptive learning rate algorithms

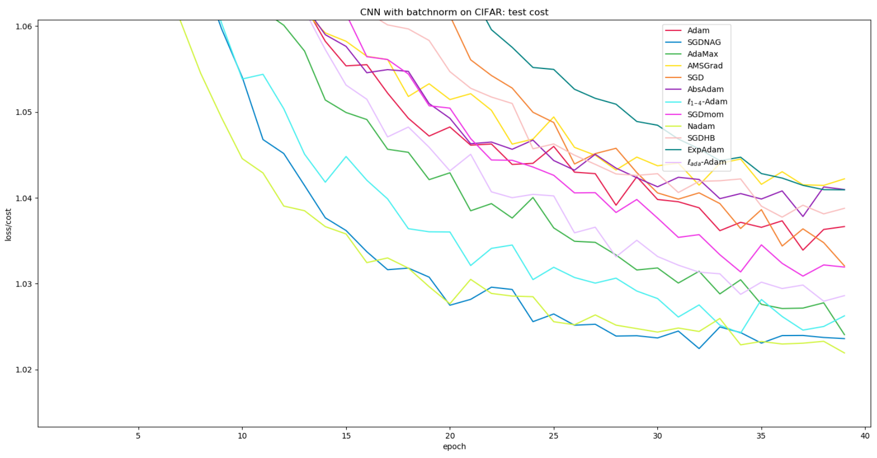


(C) Adam algorithms

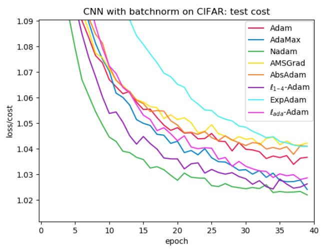


(D) SGD algorithms

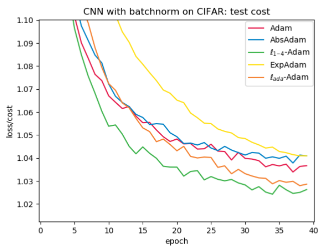
FIGURE E.13: Plots of the performance of algorithms in various groups on the experiment with a convolutional neural network with batch normalization on CIFAR10. The plots show the average training cost per iteration, averaged over 5 runs.



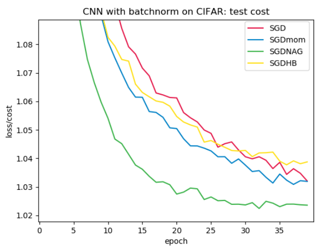
(A) All algorithms



(B) Adaptive learning rate algorithms



(C) Adam algorithms



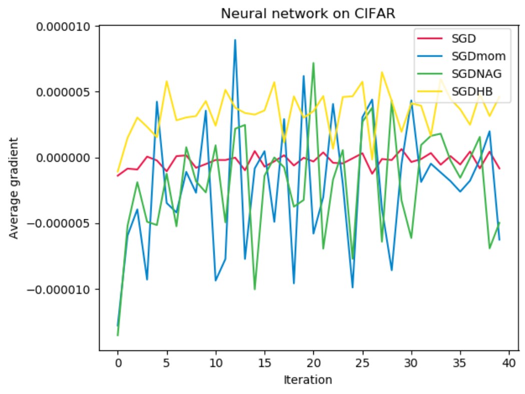
(D) SGD algorithms

FIGURE E.14: Plots of the performance of algorithms in various groups on the experiment with a convolutional neural network with batch normalization on CIFAR10. The plots show the average test cost per iteration, averaged over 5 runs.

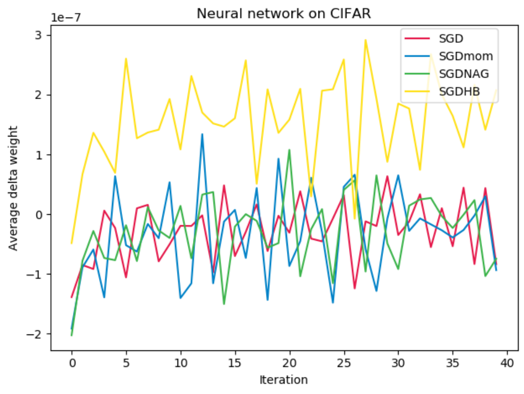
Appendix F

Feature plots

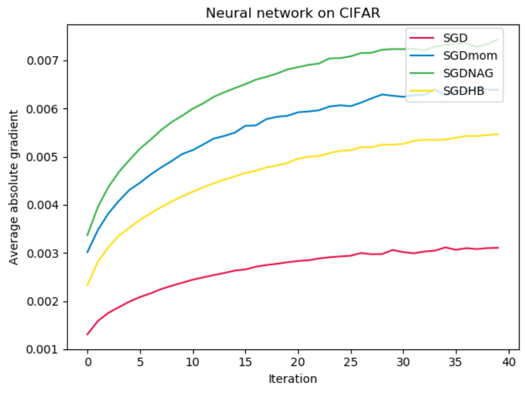
In this Appendix the feature plots that are considered in the analysis in the General Discussion chapter are displayed. For each group of figures, the feature plots showing average values are plotted in a bigger size, while the ones showing a standard deviation are displayed smaller.



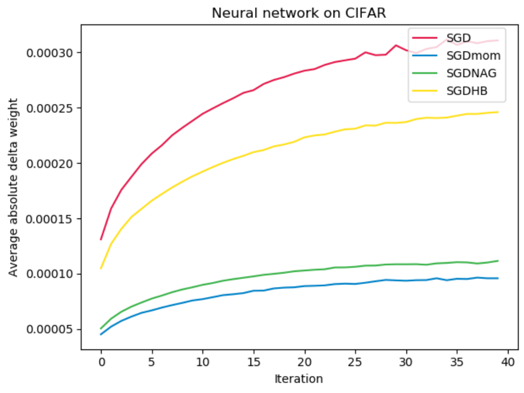
(A) Average gradient



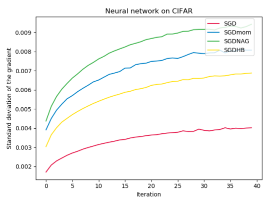
(B) Average delta weight



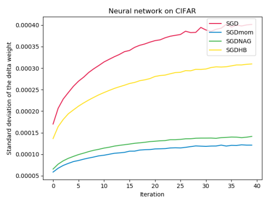
(C) Average absolute gradient



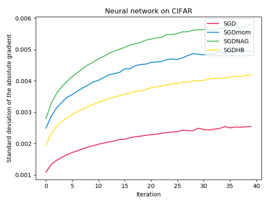
(D) Average absolute delta weight



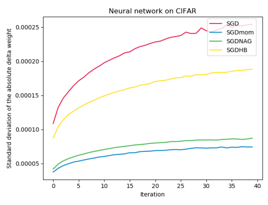
(E) STD gradient



(F) STD dW

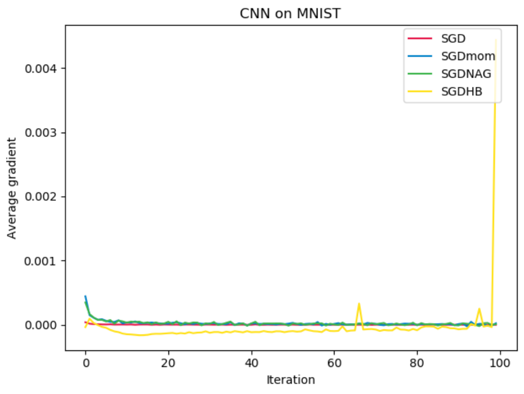


(G) STD abs gradient

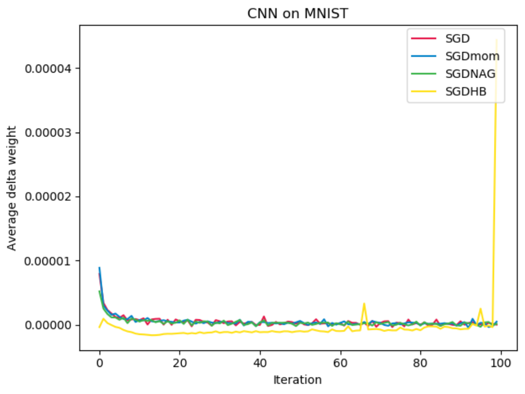


(H) STD abs dW

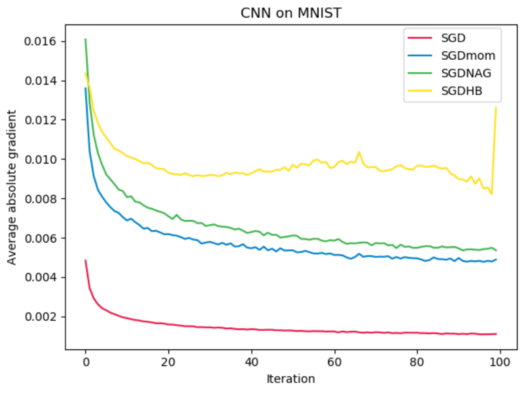
FIGURE F.1: Feature plots of the SGD algorithms on the experiment with an MLP with 1 hidden layer on CIFAR10.



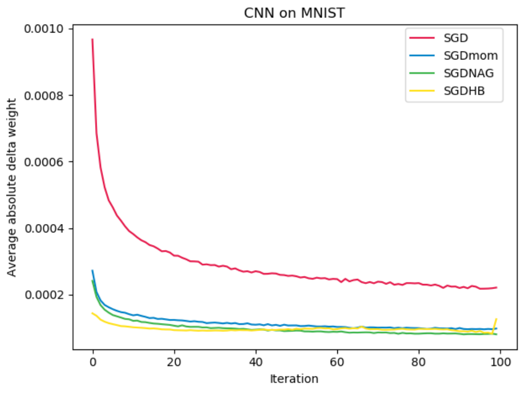
(A) Average gradient



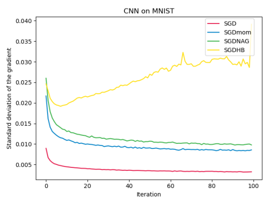
(B) Average delta weight



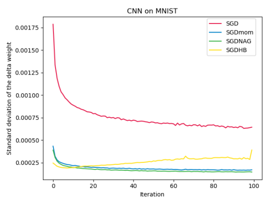
(C) Average absolute gradient



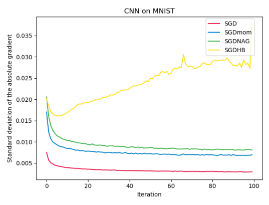
(D) Average absolute delta weight



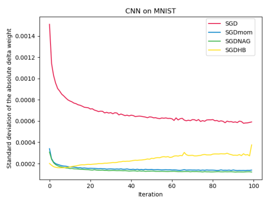
(E) STD gradient



(F) STD dW

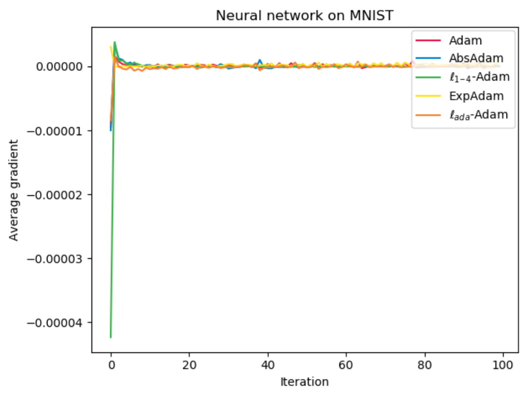


(G) STD abs gradient

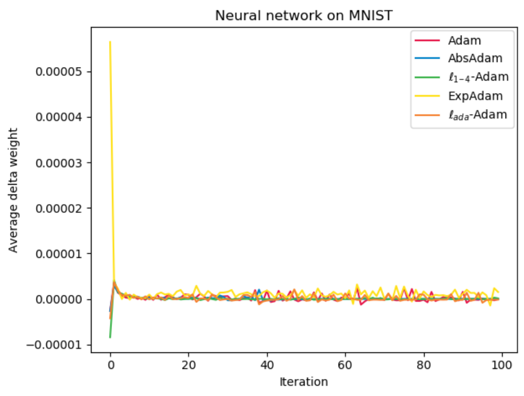


(H) STD dW

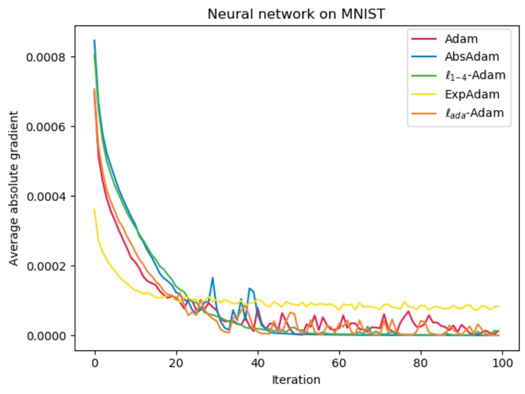
FIGURE F.2: Feature plots of the SGD algorithms on the experiment with a convolutional neural network on MNIST.



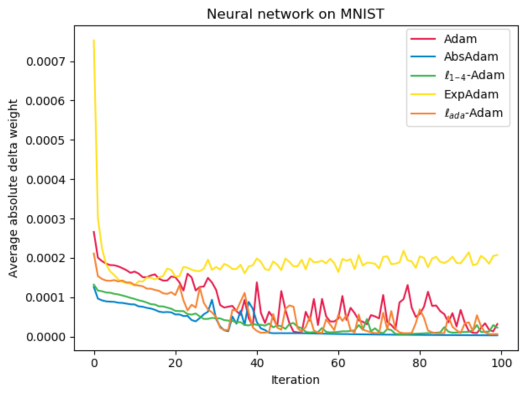
(A) Average gradient



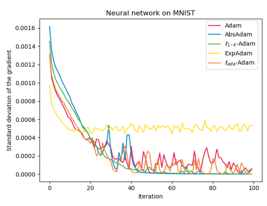
(B) Average delta weight



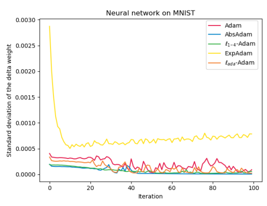
(C) Average absolute gradient



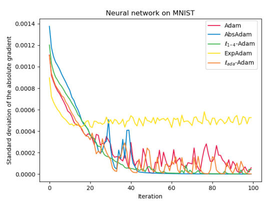
(D) Average absolute delta weight



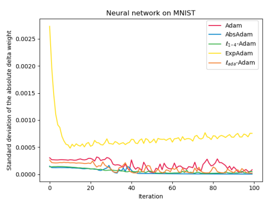
(E) STD gradient



(F) STD dW

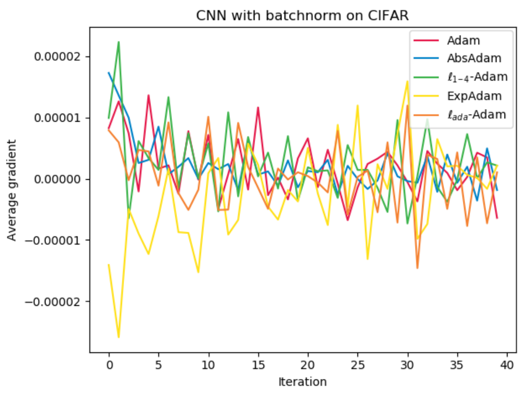


(G) STD abs gradient

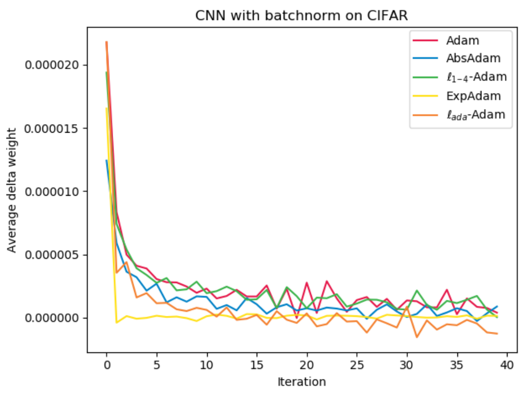


(H) STD dW

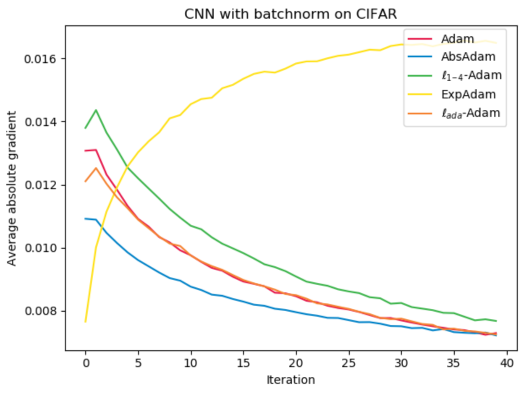
FIGURE F.3: Feature plots of the Adam algorithms on the experiment with an MLP with 1 hidden layer on MNIST.



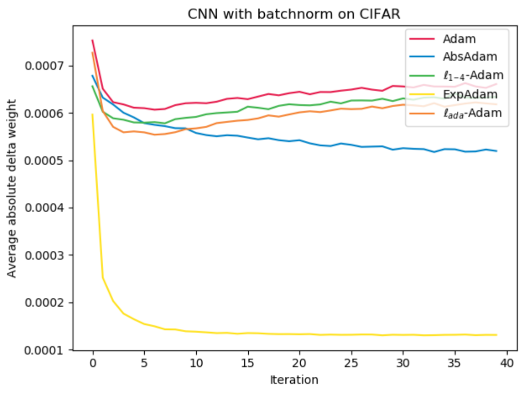
(A) Average gradient



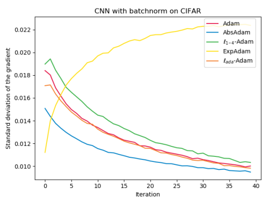
(B) Average delta weight



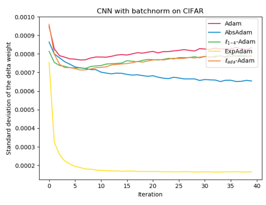
(C) Average absolute gradient



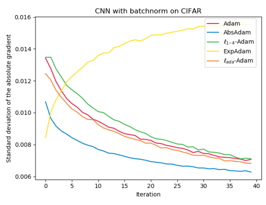
(D) Average absolute delta weight



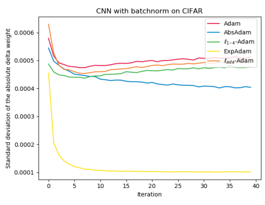
(E) STD gradient



(F) STD dW



(G) STD abs gradient



(H) STD dW

FIGURE F.4: Feature plots of the Adam algorithms on the experiment with a convolutional neural network on MNIST.

Appendix G

Mathematical proofs 'Abs', ℓ_{1-4}, and ℓ_{ada} variants

In this appendix it is tried to prove that the Adam algorithm cannot use a parameter setting that results in the same behavior as ℓ_{1-4}-Adam or ℓ_{ada}-Adam using a norm different than ℓ_2, or AbsAdam.

G.1 Rewrite Adam update rule

First the Adam update rule must be rewritten to be able to compare it properly to the new variants. The standard Adam algorithm is as follows:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{G.1}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{G.2}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t(\Theta_t) \tag{G.3}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{G.4}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t(\Theta_t)^2 \tag{G.5}$$

Equation G.3 can be rewritten as:

$$m_t = (1 - \beta_1) \cdot g_t(\Theta_t) + \sum_{a=0}^{t-1} (1 - \beta_1) \cdot g_a(\Theta_a) \cdot \beta_1^{t-a}$$

This is the same as:

$$m_t = \sum_{a=0}^{t} (1 - \beta_1) \cdot g_a(\Theta_a) \cdot \beta_1^{t-a}$$

Taking a constant term outside, we obtain:

$$m_t = (1 - \beta_1) \cdot \sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_1^{t-a}$$

Since β_1 is constant, we can now rewrite Equation G.2 as follows:

$$\hat{m}_t = \frac{(1 - \beta_1) \cdot \sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_1^{t-a}}{1 - \beta_1} = \sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_1^{t-a}$$

Equivalently, Equation G.5 can be rewritten as:

$$v_t = (1-\beta_2) \cdot \sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_2^{t-a}$$

and Equation G.4 becomes:

$$\hat{v}_t = \frac{(1-\beta_2) \cdot \sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_2^{t-a}}{1-\beta_2} = \sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_2^{t-a}$$

The full update term of Adam in Equation G.1 then becomes:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_1^{t-a}}{\sqrt{\sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_2^{t-a}} + \epsilon}$$

G.2 AbsAdam

The Abs variant only differs from the standard algorithm by the use of the absolute value of the gradient in the exponential decaying average instead of the squared gradient. So the Abs update rule can be rewritten as:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_1^{t-a}}{\sqrt{\sum_{a=0}^{t} |g_a(\Theta_a)| \cdot \beta_2^{t-a}} + \epsilon}$$

It is tried to prove that this update rule is different from the update rule of standard Adam, so there should be no parameter configuration of standard Adam that would resemble the Abs version. To prove this, it is tried to set the two update rules equal, under the conditions that both algorithms start at the exact same weight values and receive the exact same training examples such that the gradients are the same in the first optimization step, and $\alpha, \beta_1, \beta_2, \epsilon$ all have values between 0 and 1 (exclusive). The obtained equation is:

$$\Theta_t - \alpha_x \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,x}^{t-a}}{\sqrt{\sum_{a=0}^{t} |g_a(\Theta_a)| \cdot \beta_{2,x}^{t-a}} + \epsilon_x} \stackrel{?}{=} \Theta_t - \alpha_y \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,y}^{t-a}}{\sqrt{\sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_{2,y}^{t-a}} + \epsilon_y}$$

In this equation both update rules have their own parameters, therefore the momentum, smoothing term and learning rate parameters have the subscripts x (for AbsAdam) and y (for Adam).
The base case in this proof is when $t = 0$, so in the first optimization step of the learning process. In that step, the equation can be rewritten as:

$$\Theta_0 - \alpha_x \cdot \frac{g_0(\Theta_0) \cdot \beta_{1,x}^0}{\sqrt{|g_0(\Theta_0)| \cdot \beta_{2,x}^0} + \epsilon_x} \stackrel{?}{=} \Theta_0 - \alpha_y \cdot \frac{g_0(\Theta_0) \cdot \beta_{1,y}^0}{\sqrt{g_0(\Theta_0)^2 \cdot \beta_{2,y}^0} + \epsilon_y}$$

This can be simplified as:

$$\frac{\alpha_x}{\sqrt{|g_0(\Theta_0)|} + \epsilon_x} \stackrel{?}{=} \frac{\alpha_y}{|g_0(\Theta_0)| + \epsilon_y}$$

After this is rewritten using the rule that $\frac{a}{b} = \frac{c}{d}$ equals $a \cdot d = b \cdot c$, we obtain:

$$\alpha_x \cdot (|g_0(\Theta_0)| + \epsilon_y) \stackrel{?}{=} \alpha_y \cdot (\sqrt{|g_0(\Theta_0)|} + \epsilon_x)$$

The momentum parameters are all vanished by now, due to the fact that $t = 0$ and no momentum has been built up yet, which leaves only the smoothing term and the learning rate. Let's assume $\epsilon_x = \epsilon_y$ (after all it is the smoothing term). We obtain after dividing both terms by $(\sqrt{|g_0(\Theta_0)|} + \epsilon)$ (which cannot be zero due to the smoothing term), and some rewriting:

$$\alpha_x \cdot \frac{(|g_0(\Theta_0)| + \epsilon)}{\sqrt{|g_0(\Theta_0)|} + \epsilon} \stackrel{?}{=} \alpha_y$$

This equation shows that to obtain the same results when $t = 0$ for AbsAdam and Adam, at least the learning rate must be different. The learning rate of AbsAdam, α_x must be scaled by a factor $\frac{(|g_0(\Theta_0)|+\epsilon)}{\sqrt{|g_0(\Theta_0)|}+\epsilon}$ to obtain the right learning rate for Adam α_y. This means that this correct learning rate is dependent on the current gradient, which makes it impossible to configure the parameters of Adam such that it has the same behavior as AbsAdam in the first optimization step. This makes it also impossible to do this for the entire optimization process. Therefore the Abs GHC method makes an optimization algorithm different from the standard version.

G.3 ℓ_{1-4}-Adam and ℓ_{ada}-Adam

To prove that these algorithms are different from the standard Adam, it must be proven that the use of a different norm than ℓ_2 cannot be compensated by a different configuration of the other parameters. For sake of easiness, the ℓ_{1-4}-Adam algorithm is used in this proof.
The update term of ℓ_{1-4}-Adam is not much different from the one of standard Adam: only the square root and the power of 2 in the denominator are substituted for a parameter value Λ which is for sake of easiness left constant over time:

$$\Theta_{t+1} = \Theta_t - \alpha \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_1^{t-a}}{\sqrt[\Lambda]{\sum_{a=0}^{t} |g_a(\Theta_a)|^\Lambda \cdot \beta_2^{t-a}} + \epsilon}$$

Now we can set these two equal (for sake of clarity the 2 in the root is added in the Adam update, and since both algorithms have their own parameters, subscripts of x (for ℓ_{1-4}-Adam) and y (for Adam) are added to the learning rates, smoothing terms and momentum parameters). It is assumed that both systems have the same initial weights and are presented the exact same training data, such that the same gradients are obtained in the first optimization step. Furthermore it is assumed that $\alpha, \beta_1, \beta_2, \epsilon$ all have values between 0 and 1 (exclusive), and Λ has a value between 1 and 4 but not 2. The obtained equation is:

$$\Theta_t - \alpha_x \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,x}^{t-a}}{\sqrt[\Lambda]{\sum_{a=0}^{t} |g_a(\Theta_a)|^\Lambda \cdot \beta_{2,x}^{t-a}} + \epsilon_x} \stackrel{?}{=} \Theta_t - \alpha_y \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,y}^{t-a}}{\sqrt[2]{\sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_{2,y}^{t-a}} + \epsilon_y}$$

Starting with the case where $t = 0$, this becomes (after some simplification):

$$\alpha_x \cdot \frac{g_0(\Theta_0)}{\sqrt[\Lambda]{|g_0(\Theta_0)|^\Lambda} + \epsilon_x} \stackrel{?}{=} \alpha_y \cdot \frac{g_0(\Theta_0)}{\sqrt[2]{g_0(\Theta_0)^2} + \epsilon_y}$$

This is indeed the case if $\alpha_x = \alpha_y$ and $\epsilon_x = \epsilon_y$, or in other words if the same values are used for the learning rate and the smoothing term.
Now for the recursive step, where $t > 0$. Since we have proven that the same update can be made at $t = 0$, we can now assume that the values in Θ_a and hence $g_t(\Theta_t)$ are the same for both algorithms. However, we have to keep using the values for the parameters in the case where it did hold for $t = 0$, so the learning rates and smoothing terms have the same values. Let's start with the full update rules, where subscripts are omitted for the learning rates and smoothing terms:

$$\Theta_t - \alpha \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,x}^{t-a}}{\sqrt[\Lambda]{\sum_{a=0}^{t} |g_a(\Theta_a)|^\Lambda \cdot \beta_{2,x}^{t-a}} + \epsilon} \stackrel{?}{=} \Theta_t - \alpha \cdot \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,y}^{t-a}}{\sqrt[2]{\sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_{2,y}^{t-a}} + \epsilon}$$

This can be simplified to:

$$\frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,x}^{t-a}}{\sqrt[\Lambda]{\sum_{a=0}^{t} |g_a(\Theta_a)|^\Lambda \cdot \beta_{2,x}^{t-a}} + \epsilon} \stackrel{?}{=} \frac{\sum_{a=0}^{t} g_a(\Theta_a) \cdot \beta_{1,y}^{t-a}}{\sqrt[2]{\sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_{2,y}^{t-a}} + \epsilon}$$

When $\beta_{1,x}$ and $\beta_{1,y}$ have the same value, the two numerators cancel each other out, leaving us with the two denominators. Since ϵ must have the same value in both terms, this can also be removed. This results in the following:

$$\sqrt[\Lambda]{\sum_{a=0}^{t} |g_a(\Theta_a)|^\Lambda \cdot \beta_{2,x}^{t-a}} \stackrel{?}{=} \sqrt[2]{\sum_{a=0}^{t} g_a(\Theta_a)^2 \cdot \beta_{2,y}^{t-a}}$$

Can this be true under the circumstances that $\Lambda \neq 2$ and $t > 0$? Let's try for an easy case, in which $\Lambda = 1$ and $t = 1$:

$$\sum_{a=0}^{1} |g_a(\Theta_a)| \cdot \beta_{2,x}^{t-a} \stackrel{?}{=} \sqrt[2]{\sum_{a=0}^{1} g_a(\Theta_a)^2 \cdot \beta_{2,y}^{t-a}}$$

After writing the summation terms out, it becomes:

$$|g_0(\Theta_0)| \cdot \beta_{2,x}^{1-0} + |g_1(\Theta_1)| \cdot \beta_{2,x}^{1-1} \stackrel{?}{=} \sqrt[2]{g_0(\Theta_0)^2 \cdot \beta_{2,y}^{1-0} + g_1(\Theta_1)^2 \cdot \beta_{2,y}^{1-1}}$$

This can be rewritten as:

$$|g_0(\Theta_0)| \cdot \beta_{2,x} + |g_1(\Theta_1)| \stackrel{?}{=} \sqrt[2]{g_0(\Theta_0)^2 \cdot \beta_{2,y} + g_1(\Theta_1)^2}$$

Squaring both sides, we obtain after writing the first expression out:

$$|g_0(\Theta_0)|^2 \cdot \beta_{2,X}^2 + 2 \cdot |g_0(\Theta_0)| \cdot \beta_{2,X} \cdot |g_1(\Theta_1)| + |g_1(\Theta_1)|^2 \stackrel{?}{=} g_0(\Theta_0)^2 \cdot \beta_{2,y} + g_1(\Theta_1)^2$$

Since a squared value has the same result as the square of the absolute of that same value, the two last terms in the expressions are the same and can be cancelled out:

$$|g_0(\Theta_0)|^2 \cdot \beta_{2,X}^2 + 2 \cdot |g_0(\Theta_0)| \cdot \beta_{2,X} \cdot |g_1(\Theta_1)| \overset{?}{=} g_0(\Theta_0)^2 \cdot \beta_{2,Y}$$

Dividing both sides by $g_0(\Theta_0)^2$, we obtain:

$$\beta_{2,X}^2 + \frac{2 \cdot \beta_{2,X} \cdot |g_1(\Theta_1)|}{g_0(\Theta_0)} \overset{?}{=} \beta_{2,Y}$$

So this means that to let the Adam algorithm behave like the ℓ_{1-4}-Adam algorithm that uses $\Lambda = 1$, it needs to have a value for β_2 that is equal to the value for β_2 used by ℓ_{1-4}-Adam squared plus the (scaled) second obtained gradient divided by the first obtained gradient. This means that the right parameter configuration is dependent on the obtained gradients. Therefore it can be concluded that the Adam algorithm cannot use a parameter configuration such that it behaves the same as ℓ_{1-4}-Adam and ℓ_{ada}-Adam that use a value for Λ other than 2.

Bibliography

Abadi, Martın et al. (2016). "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16, pp. 265–283.

Alpaydin, Ethem (2009). *Introduction to machine learning*. MIT press.

Chen, Chenyi et al. (2015). "Deepdriving: Learning affordance for direct perception in autonomous driving". In: *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE, pp. 2722–2730.

Chen, Mo et al. (2017). "JPEG-phase-aware convolutional neural network for steganalysis of JPEG images". In: *Proceedings of the 5th ACM Workshop on Information Hiding and Multimedia Security*. ACM, pp. 75–84.

Collobert, Ronan and Jason Weston (2008). "A unified architecture for natural language processing: Deep neural networks with multitask learning". In: *Proceedings of the 25th international conference on Machine learning*. ACM, pp. 160–167.

Dozat, Timothy (2016). "Incorporating Nesterov momentum into Adam". In: *Citado na*, p. 53.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159.

Dugas, Charles et al. (2001). "Incorporating second-order functional knowledge for better option pricing". In: *Advances in neural information processing systems*, pp. 472–478.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. Vol. 1. MIT press Cambridge.

Gunning, David (2017). "Explainable artificial intelligence (XAI)". In: *Defense Advanced Research Projects Agency (DARPA), nd Web*.

Gurney, Kevin (2014). *An introduction to neural networks*. CRC press.

He, Kaiming et al. (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Hinton, Geoffrey et al. (2012). "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal Processing Magazine* 29.6, pp. 82–97.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Huang, Gao et al. (2017). "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708.

Ioffe, Sergey and Christian Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167*.

Jia, Yangqing et al. (2014). "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, pp. 675–678.

Jones, Nicola (2014). “The learning machines”. In: *Nature* 505.7482, p. 146.

Kalat, James W (2015). *Biological psychology*. Nelson Education.

Kingma, Diederik P and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*.

Krizhevsky, Alex and Geoffrey Hinton (2009). *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*, pp. 1097–1105.

LeCun, Yann (1998). “The MNIST database of handwritten digits”. In: *http://yann. lecun. com/exdb/mnist/*.

LeCun, Yann et al. (1989). “Generalization and network design strategies”. In: *Connectionism in perspective*, pp. 143–155.

LeCun, Yann et al. (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013). “Rectifier nonlinearities improve neural network acoustic models”. In: *Proc. ICML*. Vol. 30. 1, p. 3.

Nair, Vinod and Geoffrey Hinton (2010). “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814.

Nesterov, Yurii (1983). “A method for unconstrained convex minimization problem with the rate of convergence O (1/kˆ 2)”. In: *Doklady AN USSR*. Vol. 269, pp. 543–547.

Nocedal, Jorge and Stephen J Wright (2006). *Numerical optimization*. Springer.

Noh, Hyeonwoo, Paul Hongsuck Seo, and Bohyung Han (2016). “Image question answering using convolutional neural network with dynamic parameter prediction”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 30–38.

Paszke, Adam et al. (2017). *Pytorch: Tensors and dynamic neural networks in Python with strong GPU acceleration, May 2017*.

Polyak, Boris T (1964). “Some methods of speeding up the convergence of iteration methods”. In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17.

Python Core Team (2018). *Python: A dynamic, open source programming language*. Version 3.6.4. Python Software Foundation. Vienna, Austria. URL: `https://www.python.org/`.

Qian, Ning (1999). “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1, pp. 145–151.

Reddi, Sashank J, Satyen Kale, and Sanjiv Kumar (2018). “On the convergence of Adam and beyond”. In: *International Conference on Learning Representations*.

Rocco, Ignacio, Relja Arandjelović, and Josef Sivic (2017). “Convolutional neural network architecture for geometric matching”. In: *arXiv preprint arXiv:1703.05593*.

Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6, p. 386.

Rosenbrock, Howard H (1960). “An automatic method for finding the greatest or least value of a function”. In: *The Computer Journal* 3.3, pp. 175–184.

Rumelhart, David E, Geoffrey Hinton, and Ronald J Williams (1986). “Learning representations by back-propagating errors”. In: *nature* 323.6088, p. 533.

Russell, Stuart J and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited.

Sabour, Sara, Nicholas Frosst, and Geoffrey Hinton (2018). "Matrix Capsules with EM Routing". In: *6th International Conference on Learning Representations, ICLR*.

Schroff, Florian, Dmitry Kalenichenko, and James Philbin (2015). "Facenet: A unified embedding for face recognition and clustering". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823.

Socher, Richard et al. (2012). "Convolutional-recursive deep learning for 3d object classification". In: *Advances in Neural Information Processing Systems*, pp. 656–664.

Sprinkhuizen, Ida G and Egbert JW Boers (1996). *The error surface of the 2-2-1 XOR network: stationary points with infinite weights*.

Sprinkhuizen, Ida and Egbert Boers (1999). "The local minima of the error surface of the 2-2-1 XOR network". In: *Annals of Mathematics and Artificial Intelligence* 25.1-2, p. 107.

Srivastava, Nitish et al. (2014). "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Sutskever, Ilya et al. (2013). "On the importance of initialization and momentum in deep learning." In: *ICML (3)* 28.1139-1147, p. 5.

Sutton, Richard S and Andrew G Barto (1998). *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge.

Torralba, Antonio, Rob Fergus, and William T Freeman (2008). "80 million tiny images: A large data set for nonparametric object and scene recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 30.11, pp. 1958–1970.

Walt, Stéfan van der, S Chris Colbert, and Gael Varoquaux (2011). "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2, pp. 22–30.

Whitley, Darrell et al. (1996). "Evaluating evolutionary algorithms". In: *Artificial intelligence* 85.1-2, pp. 245–276.

Wiering, Marco and Martijn Van Otterlo (2012). *Reinforcement learning: State-of-the-Art*. Springer.

Wolfshaar, Jos van de, Marco Wiering, and Lambert Schomaker (2018). "Deep Learning Policy Quantization." In: *ICAART (2)*, pp. 122–130.

Zbontar, Jure and Yann LeCun (2016). "Stereo matching by training a convolutional neural network to compare image patches". In: *Journal of Machine Learning Research* 17.1-32, p. 2.

Zeiler, Matthew D (2012). "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701*.

Zhou, YT and R Chellappa (1988). "Computation of optical flow using a neural network". In: *IEEE International Conference on Neural Networks*. Vol. 1998, pp. 71–78.