UNIVERSITY
OF OSLO

Project work

A COMPARISON OF GRADIENT DESCENT METHODS USING BACKTRACKING LINE SEARCH

Sophus B. Gullbekk

MAT2000
10 study points

Department of Mathematics

Spring 2022

Sophus B. Gullbekk

A COMPARISON OF GRADIENT DESCENT METHODS USING BACKTRACKING LINE SEARCH

Abstract

We investigate the performance of backtracking line search applied to three different gradient descent optimization methods. The methods in question are 1) standard gradient descent, 2) gradient descent with momentum and 3) Nesterov accelerated gradient descent. To first visualize how the different methods move towards a minimum, we apply them to the Rosenbrock function in $\mathbb{R}^2$. Then we look at the performance when performing image recognition on the CIFAR10 data set using the deep neural network ResNet18. We find that $\alpha = 10^{-4}$ and $\beta = 0.5$ are good stable hyperparameters for backtracking line search. Then we describe an algorithm for applying backtracking to mini-batch learning, where we switch the α-value to 0.5 if the loss or accuracy becomes worse during training. The deep neural network is trained for 75 epochs on each optimization method, and we find that the backtracking methods perform well both in regards to accuracy and loss. The mini-batch two way backtracking with momentum is the most accurate at 93%. Further we perform a comparative measure of the time efficiency of the different methods. The result suggest that the backtracking methods are slower if we do not account for learning rate tuning.

Chapter 1

Introduction

Optimization plays a crucial role in many scientific applications. The core idea of optimization is to find the best solution to a problem. It is therefore natural that the name optimization originates from *optimus*, the Latin word for "best". In our case we will consider unconstrained optimization problems of the form

$$\min f(x), \quad x \in \mathbb{R}^k,$$

where $f \in C^1(\mathbb{R}^k, \mathbb{R})$.

One of the most popular methods to solve such minimization problems is *Gradient Descent* (GD). This method is commonly attributed to the french mathematician Augustin-Louis Cauchy as he used it to find the orbits of heavenly bodies in the 19'th century (Lemaréchal 2012). The idea behind GD is to calculate the gradient of the function we are working with and use it to figure out which direction leads to a better solution. In our case we will follow the gradient downwards as we try to find the global minimum of our function. Analogously the algorithm can be viewed as a ball rolling down a hill. The most basic GD method is referred to as standard GD and gives us the sequence

$$x_{n+1} = x_n - \delta_0 \nabla f(x_n).$$

The generated sequence $\{x_n\}$, is mainly decided by two factors. The search direction ∇f and the step size δ_0. It is crucial in GD methods to find a suitable step size that fits the problem. If the step size is too big the sequence can diverge at worst and be chaotic at best. On the other hand, if the step size is too small we risk getting a slow convergence that will not have time to find a proper local minimum. Therefore most GD methods rely on tuning the learning rate manually. This process can be both frustrating and time consuming. We investigate how the process of selecting a step size can be done using *Backtracking line search* as well as how it compares to conventional methods.

Backtracking line search is a search scheme based on *Armijo's condition* (Armijo 1966) that proposes a rule for how long each step along the search direction should be. It does this by changing the step size for each iteration, or according to a predefined rule. The magnitude of change for each iteration is decided by the two hyperparameters α and β in Armijo's condition. We will take a closer look both theoretically and experimentally on how α and β affect the accuracy and efficiency of the backtracking methods. The specific type

of backtracking line search we will focus on is *Backtracking* GD (BGD) and *two-way* BGD. These methods apply Armijo's condition to standard GD in two different ways: BGD uses Armijo's condition to decrease the step size during the iterations, while two-way BGD lets us both increase and decrease the step size.

It has been 150 years since Cauchy introduced standard GD to work on his equations and during that time many variants of GD has been introduced. Besides standard GD, we want to focus on two other methods: GD with *Momentum* (MMT) (Rumelhart, G. E. Hinton and Williams 1986) and *Nesterov Accelerated* GD (NAG) (Nesterov 1983). These methods introduce a momentum term to standard GD that makes them more robust and accurate. We will will implement versions of all three methods that uses backtracking line search to calculate the step sizes and test them experimentally against both their normal counterparts and some of the most popular optimization algorithms used today. To compare the different optimization algorithms we will apply them to the CIFAR10 data set (Krizhevsky, Nair and G. Hinton 2009) using the Deep Neural Network (DNN) ResNet18 (He et al. 2016). This is an interesting bench mark since it represents a real-world problem with a more complicated cost function. When dealing with DNNs, the term *learning rate* is usually used in place of step size.

In Section 2 we describe the relevant theory behind the different GD methods and backtracking line search. In Section 3 we show the figures and results from our experiments. Next, in Section 4 we discuss the results. Finally, in Section 5 we summarize our findings and make the appropriate conclusions.

Acknowledgements: I would like to thank my supervisor Trung Tuyen Truong for guiding me through this project. I would also like to thank Tov and Jouval for feedback. In addition Terje Kvernes at the IT department has helped me a lot with navigating the different computer clusters at UiO.

Chapter 2

Theory and Methods

2.1 Gradient Descent

As mentioned in the introduction standard GD is the most basic implementation of GD and serves as the foundation that the other GD methods are build upon. We start off with outlining the algorithm behind standard GD. Let $f : \mathbb{R}^k \to \mathbb{R}$ be a continuous function in C^1. Next, choose an appropriate learning rate $\delta_0 > 0$ and a $x_0 \in \mathbb{R}^k$. We can now construct the sequence $\{x_n\}$ given by

$$x_{n+1} = x_n - \delta_0 \nabla f(x_n), \quad n = 1, 2, 3, \ldots, \tag{2.1}$$

which we refer to as standard GD.

The idea behind Equation (2.1) is that $\{x_n\}$ should converge to a local minimum when following the path of the steepest descent given by $\nabla f(x_n)$. However, as (Ruder 2016) points out, there are a few challenges that needs to be addressed. The challenges mainly revolves around the choice of a appropriate learning rate δ_0. In general, there is no guarantee for convergence. If the learning rate is too small the convergence rate will be exceedingly slow, and if it is too big $\{x_n\}$ might not converge at all. Moreover, if one works with non-smooth functions like $f(x) = |x|^{4/3}$, then there may be no value of learning rate δ_0 for which standard GD converges to the global minimum $x = 0$, see (Truong and Nguyen 2022, p. 11). The challenges mentioned can be addressed by introducing backtracking line search, which modifies the learning rate for each iteration. In the next section we will describe how BGD works and how it can improve upon standard GD (among others).

2.2 Armijo's Condition

In this section we present Armijo's condition (Armijo 1966, p. 2), which is the theoretical background behind backtracking line search methods.

Definition 2.1 (Armijo's condition)**.** Let $f \in C^1(\mathbb{R}^k, \mathbb{R})$, $\alpha \in (0, 1)$ and $x, y \in \mathbb{R}^k$. Armijo's condition is satisfied if

$$f(y) - f(x) \leq \alpha \langle \nabla f(x) \, , \, y - x \rangle, \tag{2.2}$$

where $\langle \cdot\ ,\ \cdot \rangle$ is the standard inner product in $\mathbb{R}^k$.

We want to apply Definition 2.1 to standard GD. To do this we first need to redefine it to the special case where $y = x - \sigma \nabla f(x)$, for a $\sigma > 0$. The value for y is chosen as it represents the next step in the sequence generated by standard GD, shown in Equation (2.1).

Definition 2.2 (Armijo's condition continued)**.** Let $f \in C^1(\mathbb{R}^k, \mathbb{R})$, $x \in \mathbb{R}^k$, $\delta_0 > 0$ and $\alpha, \beta \in (0, 1)$. We define the function $\delta(f, \delta_0, x) : \mathbb{R}^k \to \mathbb{R}$ to be the largest element σ in $\{\beta^n \delta_0 : n \in \mathbb{N}_0\}$ that satisfies

$$f(x - \sigma \nabla f(x)) - f(x) \leq -\alpha \sigma \|\nabla f(x)\|^2. \tag{2.3}$$

Here, $\|\cdot\|$ is the Euclidian norm.

It is known that $\delta(f, \delta_0, x)$ is both well defined and strictly positive, see (Truong and Nguyen 2022). The intuition behind Armijo's condition is that $f(x - \sigma \nabla f(x)) - f(x)$ represents the difference between the function values of two consecutive points in the sequence generated by standard GD, and by forcing the difference to satisfy Armijo's condition we get an adequate decrease each step of the sequence. The condition says that the function $f(x - \sigma \nabla f(x)) - f(x)$ has to be below the line given by $-\alpha \sigma \|\nabla f(x)\|^2$, as shown in Figure 2.1. First we check if δ_0 is below the line and if not we decrease the step size by a factor of β. In Figure 2.1 we see how the condition is satisfied after decreasing it two times ($\delta_0 \to \delta_0 \beta \to \delta_0 \beta^2$). The act of decreasing the step size by multiplying with β is where the name "backtracking" originates, as we start with a high initial step size and move backwards until Armijo's condition is satisfied.

The inequality given in Definition 2.2 is the backbone of BGD, which we introduce in the section below.

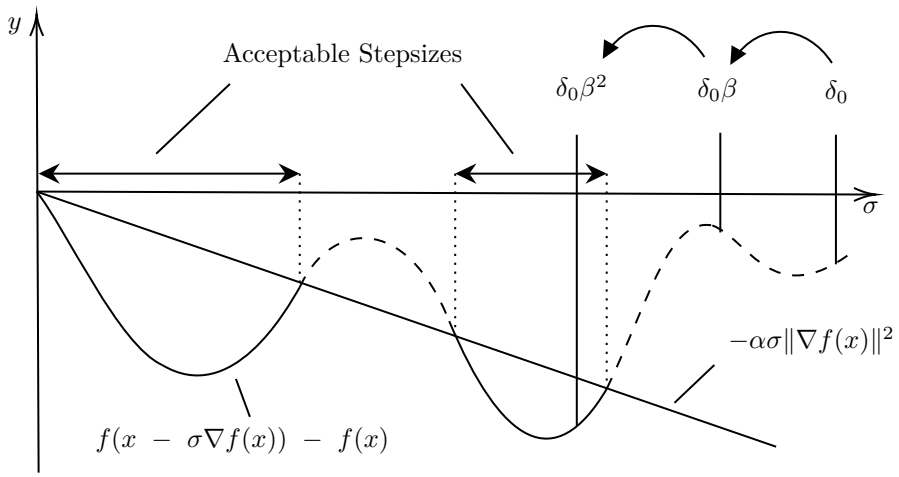

Figure 2.1: An illustration of how Armijo's condition is used to find an acceptable step size σ given by the function $\delta(f, \delta_0, x)$. Inspiration for the figure is taken from (Kopavcka et al. 2010).

2.3 Backtracking Gradient Descent

We are now ready to introduce the backtracking line search method BGD. It builds on the framework of standard GD presented in Equation (2.1) and Armijo's condition from Definition 2.2. Let $f : \mathbb{R}^k \to \mathbb{R}$ be a C^1-function. Next choose an initial learning rate $\delta_0 > 0$ and $z_0 \in \mathbb{R}^k$. We can now apply Armijo's condition to standard GD by defining the sequence $\{z_n\}$ given by

$$z_n = z_{n-1} - \delta(f, \delta_0, z_{n-1})\nabla f(z_{n-1}), \quad n \in \mathbb{N}_0. \tag{2.4}$$

This sequence is defined as BGD. The purpose of switching from using the static step size δ_0 to using the more dynamic step size $\delta(\delta_0, f, z_n)$ is to allow us to reduce the learning rate while constructing the sequence. As mentioned before, a large learning rate can make the sequence diverge. To build upon this method, it seems reasonable that we should be able to increase the learning rate as well. The next method we introduce let us do just that, it is called *two-way* BGD.

2.4 Two-way Backtracking Gradient Descent

Two-way BGD is a version of BGD that aims to make the algorithm more efficient by letting us both increase and decrease the learning rate. The idea is that instead of searching for the learning rate from δ_0 as we do in backtracking GD, we instead generate a sequence δ_n of learning rates (with initial condition δ_0) and base it on the learning rate $\sigma = \delta_{n-1}$ from the previous step. This procedure allows us to reduce the time needed to check Armijo's condition.

In (Truong and Nguyen 2021) it is shown how σ can be chosen by the following steps for an arbitrary point n in the sequence:

Definition 2.3 (Two-way BGD)**.** Below is a step by step description of the two-way BGD algorithm.

i. Choose $\sigma = \delta_{n-1}$.

ii. If σ does not satisfy Armijo's condition Definition 2.2, then choose σ to be $\delta(f, \sigma, x)$.

iii. If σ satisfies Definition 2.2, then choose σ to be $\min\{\sigma/\beta^n : \sigma/\beta^n \leq \delta_0,\ n \in \mathbb{N}\}$ such that σ satisfies Definition 2.2. This means that we replace σ with σ/β^n for some $n \in \mathbb{N}$.

iv. Define δ_n to be the final value of σ.

v. Update the sequence using the same formula as in Equation (2.4), but with the new σ.

Two-way BGD addresses the problem of slow convergence if the initial δ_0 is too small. In addition, it makes the procedure of finding the value of σ more efficient. This is because we do not start the search from δ_0 as we do in standard GD, but rather from δ_{n-1}. This means that if $\delta_n \approx \delta_{n-1}$ we need to compute few iterations to acquire a good step size. In cases where there is a large discrepancy between δ_0 and σ the calculations in standard GD can take a lot longer to complete.

Two-way BGD is therefore a stable method in regards to the initial choice of δ_0. There are reasons to expect a faster convergence for two-way BGD than both BGD and standard GD as well as less computational time. Therefore, we will use two-way BGD and not standard BGD in further experiments. In figure 2.2 we show how the different methods (standard GD, BGD and two-way BGD) moves toward the global minimum of the Rosenbrock function. This function is defined as

$$f(x, y) = (1 - x)^2 + b(y - x^2)^2, \quad b \in R, \tag{2.5}$$

which has a global minimum when $(x, y) = (1, 1)$. We use the Rosenbrock function as it is a common test function for optimization when $b = 100$.

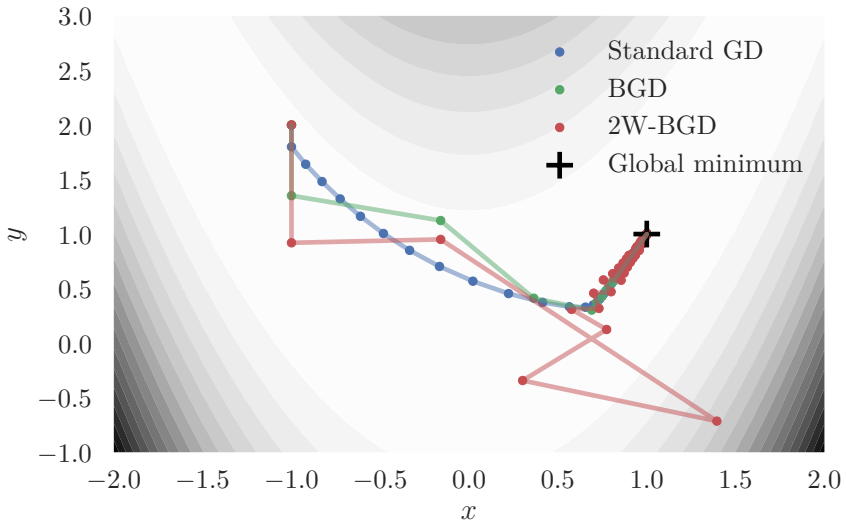

Figure 2.2: This figure shows how the sequences generated by standard GD, BGD and two-way BGD moves towards the minimum from the initial point $(x, y) = (-1, 2)$. The methods are applied to the Rosenbrock function (2.5) with $b = 100$, which has a global minimum at the point (1, 1). The parameters used for standard GD is $\delta_0 = 0.1$, while for BGD and two-way BGD it is $\alpha = 0.5$, $\beta = 0.6$ and $\delta_0 = 0.9$. The margin of error is $\epsilon = 10^{-7}$.

2.5 Momentum- and Nesterov Accelerated Gradient Descent

We have described problems related to standard GD and how the use of backtracking can solve them. In this section we will look at two other common GD methods and how they also can be improved using backtracking line search. The two techniques are called standard GD with momentum (MMT) and standard GD with Nesterov Accelerated Gradient (NAG). They are both based on the idea of generating momentum by considering the past steps of the iteration. This

is done to speed up the convergence of the sequence, as well as to avoid bad local minima.

The standard version of MMT was introduced in (Rumelhart, G. E. Hinton and Williams 1986). They proposed that you could create momentum by adding a fraction $\gamma > 0$ of the last step to the current step of the sequence. The algorithm can be defined as follows:

Definition 2.4 (MMT)**.** Choose two initial points $z_0, v_{-1} \in \mathbb{R}^k$ and fix $\gamma, \delta_0 > 0$. MMT is then defined by the following update rule:

$$\begin{aligned} v_n &= \gamma v_{n-1} + \delta_0 \nabla f(z_{n-1}), \\ z_n &= z_{n-1} - v_n. \end{aligned}$$

We have already mentioned the common analogy that GD methods behaves like a ball rolling down a hill. With standard GD this is true, but the ball will not behave like we are used to. With standard GD the ball will start to slow down before it hits the bottom, where it stops abruptly. MMT gives the ball momentum such that the speed increases as it rolls down, leading to a possible overshoot. Ideally we want a ball that increases speed as it goes down the hill, but is able to slow down in time before it hits the bottom. This would be a more "intelligent" ball, that can anticipate changes in the terrain before it. This kind of intelligence is what NAG (Nesterov 1983) gives to the momentum of our ball. It takes the momentum term γv_{n-1} from Definition 2.4 and observes that $z_n - \gamma v_{n-1}$ gives us an approximation to the next step in the sequence. We can use this to effectively "look ahead" one step in the sequence. The definition of NAG can be described as follows:

Definition 2.5 (NAG)**.** Choose two initial points $z_0, v_{-1} \in \mathbb{R}^k$ and fix $\gamma, \delta_0 > 0$. NAG is defined by the following update rule:

$$\begin{aligned} v_n &= \gamma v_{n-1} + \delta_0 \nabla f(z_{n-1} - \gamma v_{n-1}), \\ z_n &= z_{n-1} - v_n. \end{aligned}$$

Notice that we do not calculate the gradient with respect to z_{n-1} as is done in the previous methods, but rather with respect to $z_{n-1} - \gamma v_{n-1}$. It can be shown that NAG is the first order method that has the fastest convergence rate for strongly convex functions whose gradient is Lipschitz continuous (Nesterov 1983).

Another observation we can make about MMT and NAG is that they will start to behave more like standard GD the closer they get to a local minimum. This happens when the value of γv_{n-1} gets sufficiently small. The advantage of the methods are how fast they are able to get to a point close to the local minimum, which is often significantly faster given a similar step size on a well behaving function.

As we have described the theory behind the popular MMT and NAG versions of standard GD, it is time to implement versions of them that use backtracking line search. To apply backtracking line search directly to MMT and NAG would be both complicated and potentially too computationally inefficient. Therefore we use a simpler implementation of MMT and NAG with backtracking. We perform two-way BGD as normal each iteration, and apply the learning rate to MMT and NAG as well. This means that the three methods standard GD,

MMT and NAG use the same backtracking line search algorithm to calculate the learning rate. This way of implementing MMT and NAG with backtracking line search is shown heuristically to be effective. Moreover, a complicated way of incorporating backtracking line search to MMT and NAG has strong theoretical guarantees, see (Truong and Nguyen 2021).

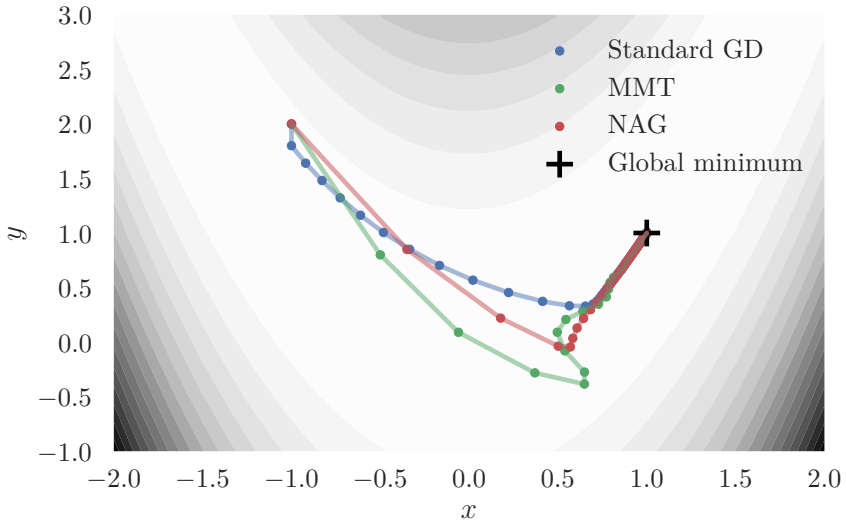

Figure 2.3: This figure shows how the sequences generated by standard GD, MMT and NAG moves towards the minimum from the initial point $(x, y) = (-1, 2)$ on the Rosenbrock function (2.5) with $b = 100$. The global minimum is at the point $(1, 1)$. The initial learning rate is set to $\delta_0 = 0.1$. Both MMT and NAG use the momentum $\gamma = 0.9$. The margin of error is $\epsilon = 10^{-7}$.

2.6 Mini-batch Gradient Descent

We want to apply the network on the CIFAR10 data set consisting of low resolution images of different objects. The goal is to make the network able to recognise the different objects and classify them correctly. To do this we have a loss function that gives each prediction a score, which in turn gives the network the ability to learn from its mistakes. The calculated loss is then used to update the nodes in the network using backpropagation. We call the whole process of going through the data, calculating the total loss, and updating the nodes for an *epoch*. A question we need to answer before the training starts is how much data we go through each epoch. If we go through all the data, we will need a large amount of memory as well as computing power since the data set is considerably large. Instead, we split the data set into several mini-batches. These mini-batches consists of a random sampling of images from the data set, which we will use as a representation for the data set as a whole. The advantage of using mini-batches is that we do not have to store the whole data set in memory and that we have a higher probability of escaping bad local minima. We will implement a version of mini-batch two-way backtracking (MBT) and

get the three new methods: MBT-GD, MBG-MMT and MBT-NAG.

An issue with mini-batch training is the fact that the gradient calculated using a mini-batch is stochastic in nature. Since standard GD is sensitive to the choice of learning rate, this poses the problem that a learning rate (step size) that works well for the given mini-batch does not generalize well to the data set as a whole. Therefore (Truong and Nguyen 2022) propose an alternative approach to do MBT-GD. This method will use backtracking on a mini-batch to find an initial learning rate, and then switch back to MBT-GD until the loss starts to increase for a given amount of epochs. Technical details can be found in (Truong and Nguyen 2022).

2.7 Rescaling Learning Rate for Mini-batch

There are some issues that needs to be considered when applying the different GD methods to DNNs using mini-batch learning. One of those issues is created by the fact that the cost functions obtained from the mini-batches are not necessarily accurate in relation to the whole data set. This means that a learning rate generated by backtracking is only optimal for the corresponding mini-batch. To prevent this problem, we introduce a scaling of the learning rates as done in e.g. (Krizhevsky 2014) and (Truong and Nguyen 2022).

The proposed method is to calculate $\rho = k/N$ where N is the size of the full batch and k is the size of a mini-batch. Then we scale the learning rate by dividing it by ρ such that $\delta_n = \delta_n/\rho$. In (Truong and Nguyen 2022), they show heuristically that the larger learning rate $\delta_n = \delta_n/\sqrt{\rho}$ can be used as well.

2.8 Choice of Hyperparameters

We define the algorithm behind two-way BGD in Definition 2.3. The algorithm includes two hyperparameters α and β in addition to the learning rate δ_0. Before applying the algorithm we need a good understanding of how those parameters change the optimizer. We would also like to establish good and "safe" values that we can assume works in a broad spectrum of applications.

In general we do not expect two-way BGD to be very sensitive to the choice of hyperparameters. We do, however, expect the hyperparameters to affect the speed of the algorithms significantly. Intuitively, you can think of β as how much we change the learning rate each time we perform an iteration of BGD, while α decides which changes are accepted. If β is large, we get a small change in learning rate during each iteration. This means that we have to perform more iterations, conversely if β is small we need fewer iterations. If α is small, we will accept many solutions and vice versa. Therefore, it will be most computationally efficient to choose a large value for β and a small value for α. However, we need to be careful that the hyperparameters still gives us accurate results.

Chapter 3

Figures and Results

In Figure 3.1 we look at how the tuning of the hyperparameters α and β affect the performance of the MBT algorithms after 50 epochs on the CIFAR10 data set using ResNet18. In Figure 3.2 we compare the performance of nine different optimizers including MBT-GD, MBT-MMT and MBT-NAG. It is done by plotting the validation accuracy and validation loss over a period of 75 epochs using ResNet on CIFAR10. Next, we have Table 3.1a and Table 3.1b which show the validation accuracy and validation loss for different learning rates for the same nine optimizers. Finally, we have Figure 3.3, which compares the time spent per epoch for the nine relevant optimizers.

The code used for generating Figure 2.2 and Figure 2.3 can be found at (Gullbekk 2022). The rest of the results are created by code from (Truong and Nguyen 2020).

α \ β	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1e-06	75	79	81	83	83	84	85	85	86
1e-05	71	79	82	82	83	85	85	85	85
0.0001	69	79	83	80	83	84	84	86	84
0.001	65	79	80	83	84	85	85	85	85
0.01	68	75	81	82	82	84	83	85	84
0.1	63	72	77	81	80	82	82	82	83
0.25	63	72	77	72	76	77	77	78	79
0.5	63	51	60	65	66	68	68	69	68
0.75	37	51	46	54	53	54	55	56	55

Figure 3.1: Comparison between different α- and β-values. The value shown is the validation accuracy after 50 epochs on CIFAR10 with ResNet18 (rounded to the nearest digit) using the optimizer MBT-GD. The initial learning rate was set to an arbitrary $\delta_0 = 100$.

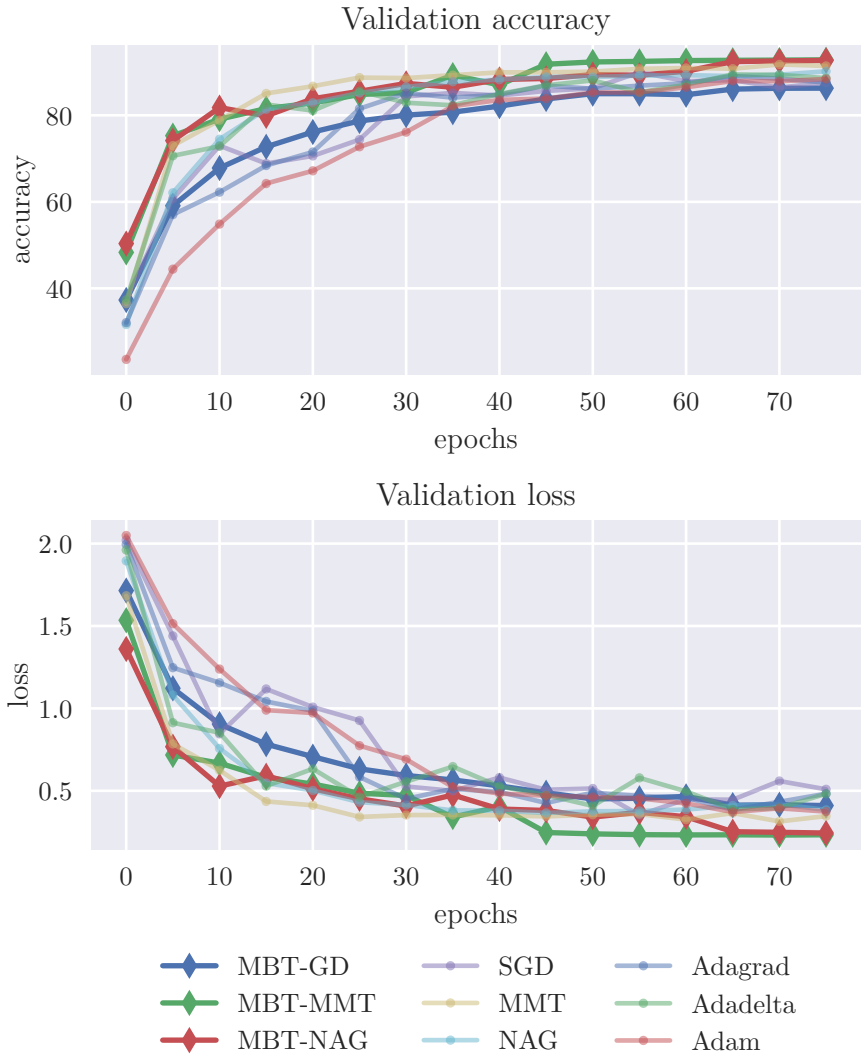

Figure 3.2: An comparison of different accuracies and loss values achieved by DNN using different optimization algorithms. The values shown is the validation accuracy and validation loss for 75 epochs on CIFAR10 with ResNet18. The top figure shows the accuracies, while the bottom figure shows the loss. All methods that do not use backtracking have an initial learning rate $\delta_0 = 0.1$ while the backtracking methods use an arbitrary $\delta_0 = 100$. We have used 0.9 as the momentum for both MMT and MBT-MMT. The backtracking parameters are $\alpha = 10^{-4}$ and $\beta = 0.5$.

Table 3.1: The two subplots below show the validation accuracy and validation loss from nine optimization algorithms using different learning rates. The DNN used is ResNet18 and it is trained for 75 epochs on the CIFAR10 data set. The parameters used for the backtracking methods are $\alpha = 10^{-4}$ and $\beta = 0.5$, with 100 as an arbitrary initial learning rate. For MMT and MBT-MMT we have used a momentum of 0.9. The lowest loss achieved is marked in **bold**. The best accuracy for each non-backtracking method is marked with an underline.

(a) Validation accuracy for seven different learning rates.

learning rates	10	1	0.1	0.01	0.001	0.0001	1e-05
standard GD	66.04	84.53	87.55	83.36	69.9	41.99	23.04
MMT	10.0	85.87	91.41	90.87	87.79	71.97	41.8
NAG	9.94	85.93	90.17	91.16	87.52	70.49	41.66
Adagrad	29.94	82.91	87.56	89.6	82.96	52.95	29.05
Adadelta	88.23	89.34	88.6	83.93	62.48	38.1	19.36
Adam	10.0	10.0	88.38	88.91	90.04	90.25	81.33
MBT-GD				86.27			
MBT-MMT				**92.82**			
MBT-NAG				92.69			

(b) Validation loss for seven different learning rates.

learning rates	10	1	0.1	0.01	0.001	0.0001	1e-05
standard GD	1.14	0.61	0.51	0.56	0.85	1.55	2.08
MMT	2.59	0.43	0.35	0.37	0.40	0.79	1.55
NAG	6.00	0.44	0.38	0.36	0.39	0.86	1.55
Adagrad	2.04	0.59	0.48	0.39	0.50	1.30	1.92
Adadelta	0.50	0.47	0.48	0.50	1.06	1.66	2.19
Adam	2.79	2.38	0.37	0.43	0.40	0.36	0.55
MBT-GD				0.41			
MBT-MMT				**0.23**			
MBT-NAG				0.24			

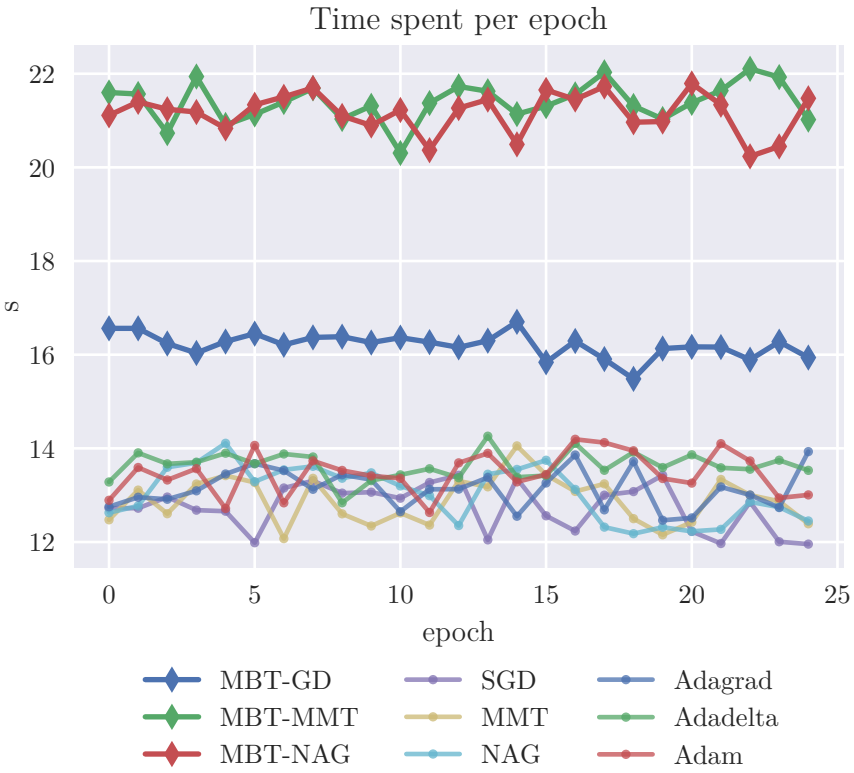

Figure 3.3: The time spent during the first 25 epochs for different optimizers on the CIFAR10 data set using Resnet18. The MBT methods use an initial learning rate $\delta_0 = 100$ and the hyperparameters $\alpha = 10^{-4}$, $\beta = 0.5$.
The non-MBT methods use the learning rate $\delta_0 = 0.1$. For MMT and NAG we have used the momentum factor $\gamma = 0.9$.

Chapter 4

Discussion

4.1 Hyperparameter Tuning

The first step towards an effective optimizer is to understand how to tune the hyperparameters to tailor the optimizer to our problem, as well as establishing some good ground values that should work for a wide variety of problems. All the MBT methods share two common hyperparameters: α and β. The learning rate is not important to consider when using backtracking line search.

In Figure 3.1 we show a heatmap of different combinations of alpha and beta impact the accuracy of the model after 50 epochs. The network used is ResNet18. The values tested are

$$\alpha \in [10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 0.25, 0.5, 0.75] \quad \text{and}$$
$$\beta \in [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9].$$

In Section 2.8 we claimed that we do not expect backtracking line search methods to be very sensitive to the choice of hyperparameters. This is for the deterministic case. However, for the stochastic case, e.g. for the mini-batch setting used in Figure 3.1, we get the impression that this is only partially true, as there is a significant variance in accuracies achieved. The figure shows a bias against large values of α and small values of β. For instance, the accuracy attained using $= 0.75$ and $\beta = 0.1$ is only 37%. It seems that the α values plays the biggest role as the gradient is stronger in the α-direction than in the β-direction. It must be noted that the experiment were only performed for 50 epochs, and we might see a change in accuracies if we trained the DNN for a longer period.

3.1 shows us that a small α gives a good accuracy, but the same can not be said for β. This might be because if the value of β is too small, we get a higher variance in learning rates which might not be optimal for learning. Therefore we will try to choose a sweet spot where β is large enough for a good accuracy, but small enough to be computationally efficient. The best value attained was 86% when $\alpha = 10^{-4}$ and $\beta = 0.8$. This is a good value for α, but we would like a smaller value for β that is more computationally efficient. In Truong and Nguyen (2021) they use $\alpha = 10^{-4}$ and $\beta = 0.5$. This seems reasonable, as the accuracy in 3.1 is still a respectable 83%. Therefore, we set $\alpha = 10^{-4}$ and $\beta = 0.5$ for all the other experiments.

4.2 Comparison of Accuracy and Loss

When comparing the different optimizers, we are mainly interested in two things: How backtracking affects the performance against the non-backtracking counterparts and how they compare against some of the most common optimizers. In Figure 3.2 we see how the accuracy and loss of nine different optimizers change through 75 epochs, while in Table 3.1a and Table 3.1b we show the accuracy and loss for different learning rates.

The first thing we observe in Figure 3.2 is that after around 40 epochs, they all perform similarly. This is reflected both in the loss and accuracy. The models that converges slower are Adam, Adagrad and standard GD. Among the top performers we have both MBT-MMT and MBT-NAG, this is perhaps easiest to see in the loss values. They both seem to have found a good local minimum to converge to, you can observe that MBT-MMT have already converged after around 45 epoch while MBT-NAG follows after around 65 epochs. This might point to the fact that introducing backtracking lets the methods escape bad local minima easier.

In Table 3.1a and Table 3.1b we see that the non-backtracking optimizers are sensitive to the learning rate chosen. For instance does Adamax perform very well with an accuracy of 92.23% when the learning rate is 0.01, but when the learning rate is 10 the accuracy is only 9.93 % which is equivalent to random guessing. This is why the normal GD methods needs to tune the learning rate prior to learning. The process of finding a good learning rate is potentially time consuming and one of the main problems that are addressed by backtracking line methods. In contrast, all the MBT methods were trained with an initial learning rate of 100, since it does not affect the performance.

The best accuracy achieved in both Figure 3.2 and Table 3.1a is 92.82% by the MBT-MMT optimizer, as a close second we have the MBT-NAG optimizer with an accuracy of 92.69%. This is promising, and shows that the two methods perform better than the non-backtracking methods even with some tuning. We see that MBT-GD attains an accuracy of 86.27 % and a loss of 0.41. This is still a relatively good result, as we do not have to adjust the learning rate.

One observation in favour of MBT-MMT and MBT-NAG from 3.2 is that they acheive a good accuracy and loss quickly. We see that both methods have achieved an accuracy of around 80% after just 10 epochs. In comparison, Adam use around 35 epochs to achieve the same accuracy. This means that if you have limited computing power and need fast learning over a small amount of epochs, then both MBT-MMT and MBT-NAG seems like good options.

It should be noted that the results in 3.2, 3.1a and 3.1b are computed using only 75 epochs. This is done because the training of the DNN takes a lot of computation power. As a consequence, the results should be regarded as mostly indicative. If we trained the models for a longer time, for instance 200 epochs, we would expect the final accuracies to be higher as the optimization algorithms finds better local minima. The results do however show that during a smaller training period we see a good performance and stability from the backtracking line search methods, especially from MBT-MMT and MBT-NAG.

4.3 Comparison of times/efficiency

In Figure 3.3 we see how the different optimizers affect the time taken to train each epoch. We clearly see that the time usage can be divided into three separate groups: MBT-MMT and MBT-NAG, MBT-GD, and the non-backtracking methods. The non-backtracking methods are the most time-efficient, staying between 12 and 14 seconds per epoch. The second most efficient is MBT-GD, which uses approximately 16 seconds per epoch. Lastly MBT-MMT and MBT-NAG uses between 20 and 22 seconds per epoch. The reason why the backtracking methods use additional time is because we have to calculate an appropriate learning rate. However, since the MBT-GD method only calculates the new learning rate occasionally, it is not penalized as much as the other two.

If you say that the non-backtracking methods perform at 13 seconds per epoch, while MBT-MMT and MBT-NAG performs at 21 seconds per epoch, we see an increase in time per epoch of around 60%. This is an significant increase, but it does not reflect the fact that you do not need to tune the learning rate when using backtracking. Therefore, in reality, the time spent by the normal GD methods can be significantly higher, depending how much tuning you perform.

Chapter 5

Conclusions

We have described the theory behind backtracking line search and how it can be used to improve GD methods. We focused on backtracking versions of standard GD, MMT and NAG, and showed experimentally how they compare to some of the most commonly used optimization algorithms. We looked at how they move towards the global minimum of the Rosenbrock function and compared the performance of the algorithms when performing image recognition on CIFAR10 using ResNet18.

We trained the DNN using MBT-GD with different values of the hyperparameters and found that a good middle ground between accuracy and performance was $\alpha = 10^{-4}$ and $\beta = 0.5$. All later experiments used those values. Next we compared the accuracy and loss of the backtracking line search methods and compared them to some of the most common gradient descent methods. For the non-backtracking methods we used the learning rates $\delta_0 \in \{10^1, 10^0, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. As the initial learning rate is not important for the MBT methods we set $\delta_0 = 100$ for those. The network were trained for 75 epochs. The results showed that out of all the optimizers MBT-MMT and MBT-NAG performed the best with accuracies of 92.82% and 92.69% respectively. They also had the lowest loss: MBT-MMT got 0.23 and MBT-NAG got 0.24. The best non-backtracking method was standard NAG with an accuracy of 91.16% and a loss of 0.36 for $\delta_0 = 10^{-2}$. The accuracy and loss for $\delta_0 = 0.1$ were visualized in Figure 3.2. Lastly, in Figure 3.3 we investigated the efficiency of each method by measuring the time used to train the network each epoch. There we found that the methods can be put into three different groups: The non-backtracking methods used between 12 and 14 seconds, MBT-GD used approximately 16 seconds and MBT-MMT and MBT-NAG used between 20 and 22 seconds. We argue that the fact that the MBT methods do not need to tune the learning rate makes up for the decrease in efficiency.

There are three main questions that would be interesting to investigate in future work. 1) It would be insightful to do a more rigorous test of the influence of the hyperparameters α and β in backtracking line search, which would also include an analysis of their impact on performance. 2) It would be exciting to investigate if other methods could be improved by backtracking line search. A good candidate for this would be Adam. 3) You could test backtracking line search methods for a larger class of problems and DNNs for more comprehensive evaluations on their performances in realistic and stochastic settings.

Bibliography

Armijo, L. (1966). 'Minimization of functions having lipschitz continuous first partial derivatives'. eng. In: *Pacific journal of mathematics* vol. 16, no. 1, pp. 1–3.

Gullbekk, S. (Feb. 2022). *Implementation of Gradient Descent Methods*. URL: https://github.com/sophus0505/Implementation-of-Gradient-Descent-Methods.

He, K. et al. (2016). 'Deep Residual Learning for Image Recognition'. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Kopavcka, J. et al. (May 2010). 'Utilising Optimization Methods for Computing of Normal Vector to Contact Surface'. In.

Krizhevsky, A. (2014). 'One weird trick for parallelizing convolutional neural networks'. In: *ArXiv* vol. abs/1404.5997.

Krizhevsky, A., Nair, V. and Hinton, G. (2009). 'CIFAR-10 (Canadian Institute for Advanced Research)'. In: URL: http://www.cs.toronto.edu/~kriz/cifar.html.

Lemaréchal, C. (2012). *Cauchy and the Gradient Method*.

Nesterov, Y. (1983). 'A method for solving the convex programming problem with convergence rate $O(1/k^2)$'. In: *Proceedings of the USSR Academy of Sciences* vol. 269, pp. 543–547.

Ruder, S. (2016). 'An overview of gradient descent optimization algorithms'. In: *CoRR* vol. abs/1609.04747. arXiv: 1609.04747.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986). 'Learning representations by back-propagating errors'. eng. In: *Nature (London)* vol. 323, no. 6088, pp. 533–536.

Truong, T. T. and Nguyen, T. H. (2020). *MBT-optimizer*. URL: https://github.com/hank-nguyen/MBT-optimizer.

— (2021). 'Backtracking Gradient Descent Method and Some Applications in Large Scale Optimisation. Part 2: Algorithms and Experiments'. In: *Applied Mathematics and Optimization* vol. 84, pp. 2557–2586.

— (2022). 'Backtracking gradient descent method and some applications in large scale optimisation. Part1: Theory'. In: *Minimax Theory and its Applications* vol. 7, pp. 79–108.