The Optimisation Algorithm in Machine Learning

Christofi Elefterios

A thesis presented for the degree of
Master of Science

University of Crete
Department of Mathematics and Applied Mathematics

committee

Prof. Th. Katsaounis(Advisor)
Prof. V. Harmandaris
Prof. M. Plexousakis

June 2020

Abstract

The last two decades have marked a rapid and significant growth of the Artificial Intelligence field. Deep learning using artificial neural networks became an essential tool for a vast number of applications fields. The structure of deep learning relies on basic concepts from several mathematical fields, such as linear algebra, calculus, optimization, statistics. This thesis is an introduction to the mathematical background of deep learning. In particular we focus on the optimization algorithm widely used, namely the stochastic gradient descent. We study and compare the behaviour of different variants of this algorithm under various circumstances and summarize their strengths and weaknesses. The goal of this thesis is to provide the reader with the knowledge to comprehend the training procedure of a neural network.

Acknowledgements

First of all, I would like to thank Professor Theodoros Katsaounis for supervising my research on this project. Our collaboration made me interact with new aspects of science and especially the area of deep learning. Additionally, i would also like to thank my committee members, Professor Michael Plexousakis and Professor Vangelis Harmandaris. Lastly, i thank my family and friends for their love and support.

List of Figures

Contents

Chapter 1

Introduction

Artificial Intelligence(AI) is the capability of a machine to imitate intelligent human behaviour. AI is accomplished by studying how humans learn, decide and work while trying to solve a problem. Outcomes of this study is used as a basis of developing intelligent software and systems. Today's artificial intelligence is powerful and easily accessible. AI has the ability to transform industries and opens up a world of new possibilities.

To achieve Artificial Intelligence we utilize a tool called Machine Learning. Machine learning is a subset of Artificial Intelligence that provides computers with the ability to learn without being explicitly programmed. In contrast with its huge capabilities, machine learning has some limitations. A subset of machine learning called deep learning enables us to overcome them.

Deep learning is a way to extract useful patterns from data in an automated way, with as little human effort involve as possible, hence the automated. The main difference between deep learning and machine learning is the ability to remove the human costly inefficient effort from the whole process, [17]. Deep learning gets us closer to the raw data, without the need of human involvement. Therefore, the automated extraction of features allows us to work with larger datasets. Later in this thesis we will show that with the optimization of neural networks we can achieve that. There are tools easily accessible like TensorFlow [13] and PyTorch

[9], which provide us with the resources needed to implement deep learning. The hard part of deep learning and artificial intelligence in general, is to ask the right questions in order to get useful data. Due to the digitization of information we have the ability to access data easily in a distributed fashion across the world. As a result, all kinds of problems have now a digital form, which they can be accessed by learning algorithms. We live in a time where we have the hardware that enables the efficient and effective large-scale execution of these algorithms. Deep learning has a vast range of applications, from object detection [18], natural language processing [12] and speech recognition [15] to medical diagnosis [3] and drug discovery [6].

The structure of this thesis is organized as follow. In the first part, an introduction to the mathematical background of deep learning is given, in order to define the basic principles that occur during the optimization of the neural networks. This process is described qualitatively and a method for the optimization called gradient descent is introduced along with the numerical methods that were used for approaching this process. Afterwards, a number of gradient descent algorithms is presented, with each approaching the optimization process in a different way. In the last part, we compared the gradient descent algorithms and discuss the numerical results.

Chapter 2

Mathematical formulation of the problem

2.1 Artificial Neural Network

The majority of deep learning methods use neural networks, as a result of which deep learning models described as deep neural networks. The word *deep* frequently applies to the number of hidden layers in the neural network. A neural network can be *shallow*, implies that it contains only one hidden layer, while deep neural networks contain two or more hidden layers as shown in Figure 2.1.

The structure of neural networks consists of the input layer (this layer supplies the information into the system), the hidden layers (execute the calculations and pass the results to the output layer) and the output layer(this layer presents us the information identified by the network).

The idea behind deep learning is to build algorithms that can mimic brain. The artificial neural network implements multiple application of a basic function, called activation function. The main purpose of that function is to imitate the action of the biological neuron, by deciding whether the neuron should fire or not. There are three types of activation functions, binary step functions, linear activation functions and non linear activation functions. Only non linear activation functions

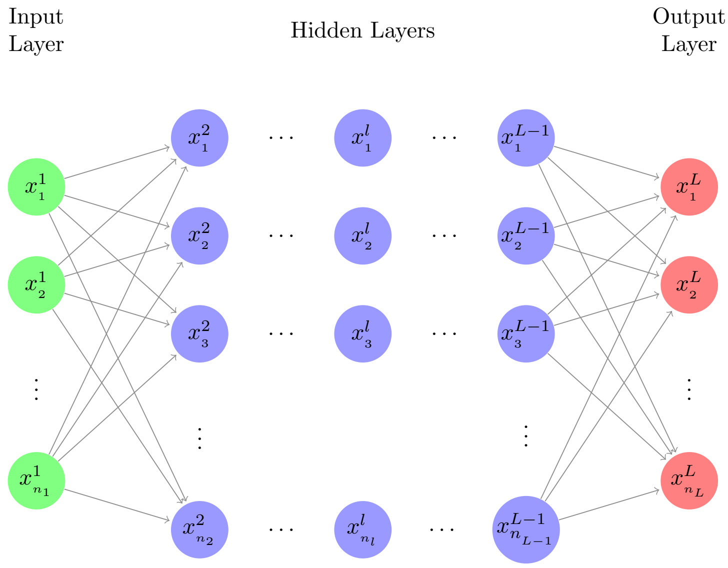

Figure 2.1: Deep Neural Network General Form

have the ability to allow backpropagation because they have a derivative function which is related to the inputs. There is a variety of non linear activation functions such as *Sigmoid, Tanh and ReLU* [8].

The sigmoid function is one of the most commonly used activation functions

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.1}$$

which takes values between 0 and 1 and its graph is depicted in Figure 2.2. The value 1 indicates the neuron is active(*fired*) while 0 indicates neuron inactivity. It has a smooth gradient and its derivative can be expressed in terms of the function itself :

$$\begin{aligned}\sigma'(x) &= (\frac{1}{1+e^{-x}})' = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1-1+e^{-x}}{(1+e^{-x})^2} = -(\frac{1}{1+e^{-x}})^2 + \frac{1}{(1+e^{-x})} \\ &= -(\sigma(x))^2 + \sigma(x) \implies \sigma'(x) = \sigma(x)(1-\sigma(x)).\end{aligned}$$

The shape and location of the curve can be modified by changing the argument to $z = Wx + b$, where W(*weight*) controls the steepness of the function and b(*bias*) is the displacement in x-direction as shown in Figure 2.2.

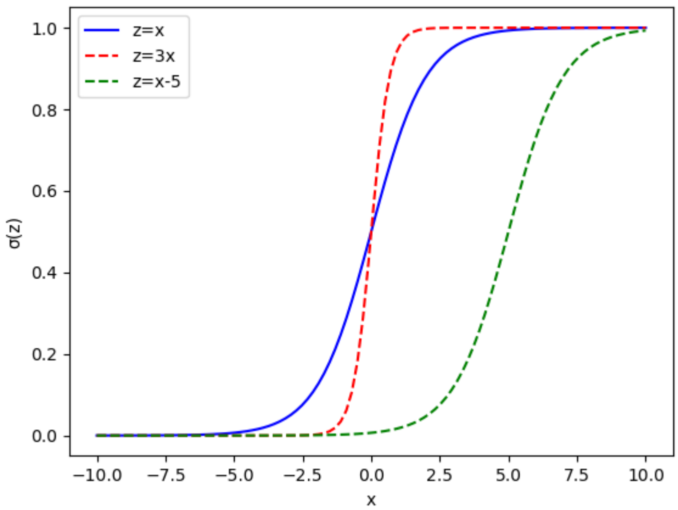

Figure 2.2: Sigmoid function with shifted and scaled input

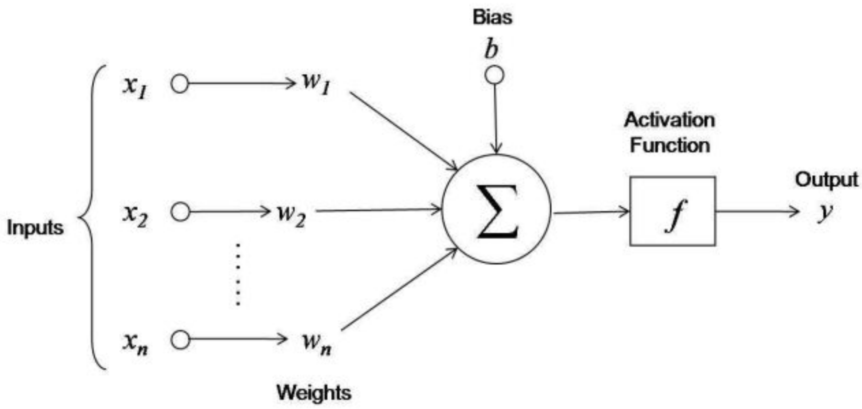

Figure 2.3: Single neuron

Every neuron takes a set of X values as an input and calculate a y value as an output. Each neuron has its own set of parameters, mostly mentioned as W (vector of weights) and b (bias) which change through the learning process. In every iteration the neuron computes a weighted average of the X values, relying on its current weights and adds bias. At last, the computation of that result through the activation function gives us the output.

$$y = \sigma(z) = \sigma(\sum_j W_j X_j + b).$$

The process we just described is presented in Figure 2.3.

The next step is to observe a single layer. We bring together all the outputs from the neurons belonging to the specific layer into a vector called $a^{[i]}$ (a comes from activation) where i is the index of the layer. Now, W takes the shape of a $n_i \times n_{i-1}$ matrix and b the shape of a $n_i \times 1$ vector, where n_i is the number of neurons of the layer i and n_{i-1} the number of neurons of the layer $i-1$. For instance, in Figure 2.1 the layer l has n_l neurons, while layer $l-1$ has n_{l-1} neurons. Therefore, $a^{[l]} \in \mathbb{R}^{n_l \times 1}$ and $a^{[l-1]} \in \mathbb{R}^{n_{l-1} \times 1}$. Since, $z \in \mathbb{R}^m$, $\sigma(z) : \mathbb{R}^m \to \mathbb{R}^m$ then, $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ and $b^{[l]} \in \mathbb{R}^{n_l}$. Moreover, the output of a single neuron and a single layer takes the form of equation (2.2) and (2.3) respectively,

$$(\sigma(z^{[l]}))_j = \sigma(z_j^{[l]}) = \sigma\left(\sum_{i=1}^{n_{l-1}} W_{ji}^{[l]} a_i^{[l-1]} + b_j^{[l]}\right), \tag{2.2}$$

$$a^{[l]} = \sigma(z^{[l]}) = \sigma\left(W^{[l]} a^{[l-1]} + b^{[l]}\right). \tag{2.3}$$

2.2 Cost Function

The cost function measures the error between the values predicted by the model and the actual values. In simple terms, it displays how well the model is performing. Cost function maps that error into a real number. If our data set consists of N points ($x_i \in \mathbb{R}^{n_1}$) and each point has its own target output($y(x_i) \in \mathbb{R}^{n_L}$), then the cost function takes the form of equation (2.4), where $a^{[L]}$ is the output from the last layer:

$$cost = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} \left(\|y(x_i) - a^{[L]}(x_i)\|_2\right)^2. \tag{2.4}$$

The optimization problem we are facing is to find the appropriate weights and biases to minimize the cost function, (2.4). If we collect all the parameters(weights and biases) into a single vector $p \in \mathbb{R}^v$, where v is the number of parameters, then $cost : \mathbb{R}^v \to \mathbb{R}$.

2.3 Gradient Descent

The procedure of the cost function minimization is called *learning*. To achieve that, we use the gradient descent method. Gradient descent is a first order iterative optimization algorithm and it's used widely in machine learning for minimizing the cost function. If the current value of the cost function is $cost(p)$, to find the local minimum of the function we need to choose $p+\Delta p$ such that $cost(p) \geq cost(p+\Delta p)$. Suppose, that Δp is very small, then the second and higher order terms of the Taylor series expansion are ignored. Therefore,

$$\begin{aligned} cost(p+\Delta p) &\approx cost(p) + \sum_{i=1}^{v} \frac{dcost(p)}{dp_i} \Delta p_i \implies \\ cost(p+\Delta p) &\approx cost(p) + \nabla cost(p)^T \Delta p. \end{aligned} \tag{2.5}$$

Consequently, from equation (2.5) the key to decrease the value of the cost function is to select properly Δp such that $\nabla cost(p)^T \Delta p$ is as negative as possible. To achieve that we use the Cauchy-Schwarz inequality,

$$|\nabla cost(p)^T \Delta p| \leq \|\nabla cost(p)\|_2 \|\Delta p\|_2 \implies \tag{2.6}$$

$$-\|\nabla cost(p)\|_2 \|\Delta p\|_2 \leq \nabla cost(p)^T \Delta p \leq \|\nabla cost(p)\|_2 \|\Delta p\|_2. \tag{2.7}$$

From equation (2.7) it is obvious that for $\Delta p = -\nabla cost(p)$ we have the desired result. This observation enable us to construct the algorithm called *steepest descent* method. We start with an initial guess p_0 of the parameters we want to optimize and then iterate according to:

$$p_{i+1} = p_i - \eta \nabla cost(p_i), \tag{2.8}$$

where,

$$C_{x_i} = \frac{1}{2}(\|y(x_i) - a^{[L]}(x_i)\|_2)^2, \tag{2.9}$$

$$\nabla cost(p) = \frac{1}{N} \sum_{i=1}^{N} \nabla C_{x_i}(p). \tag{2.10}$$

For an appropriate step-size η (*learning rate*), the sequence $cost(p_0) \geq cost(p_1) \geq ...$ converges to a local minimum.

To detect the local minimum of a function through the gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point as shown in Figure 2.4(right). A disadvantage of gradient descent is the fact that throughout the learning process the algorithm can get trapped in a local minimum of the cost function and never make it to the global minimum, see Figure 2.4(right), except for the convex cost functions.

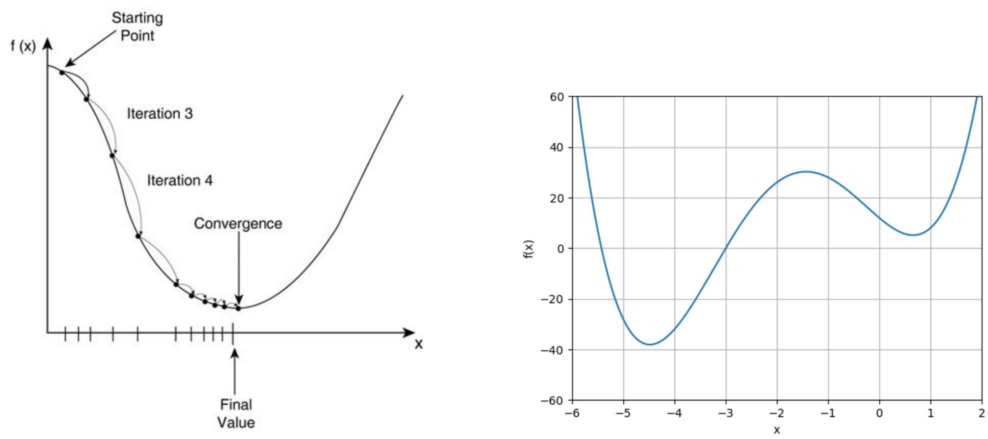

Figure 2.4: Left: The Gradient Descent process in 1D. Right : Graph of the non-convex function $f(x) = x^4 + 7x^3 + 5x^2 - 17x + 12$

The Python code Listing 6.1 in Section 6.1 shows how the gradient descent algorithm works on the non-convex function displayed in Figure 2.4(right). The initial guess of the x value determines at what local minimum the algorithm will converge. For instance, for $x > -1$ the negative gradient would have let us to a local minimum, while if we had started at the left side the negative gradient would have let us to the global minimum.

2.4 Back Propagation

The purpose of back propagation is to calculate the partial derivatives of the cost function with respect to any weight (W) and bias (b). By taking a fixed training point the equation (2.9) takes the form,

$$C = \frac{1}{2}(\|y - a^{[L]}\|_2)^2 = \frac{1}{2}\sum_j (y_j - a_j^{[L]})^2. \tag{2.11}$$

Because of the fact that x is a fixed training point then y is also a fixed parameter. Consequently, C is a function of the output activations ($a^{[L]}$). In order to proceed through the back propagation fundamental equations we need to introduce the Hadamard product notation. Specifically, suppose that the vectors v and u are of the same dimension. Hence, the notation $v \odot u$ indicates the elementwise multiplication product of the two vectors corresponding components. Then the components of $v \odot u$ are exactly $(v \odot u)_i = v_i u_i$. For instance,

$$\begin{bmatrix} a \\ b \end{bmatrix} \odot \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ bd \end{bmatrix}$$

Furthermore, the quantity $\delta^{[l]} \in \mathbb{R}^{n_l}$ is referred to as the error of the l^{th} layer. Thus, $\delta_j^{[l]}$ is called the error of the j^{th} neuron of the l^{th} layer and is defined by

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}}, \tag{2.12}$$

where,

$$z_j^{[l]} = \sum_{i=1}^{n_{l-1}} W_{ji}^{[l]} a_i^{[l-1]} + b_j^{[l]}, \tag{2.13}$$

from (2.2). Back propagation with the help of chain rule will provide us a method to calculate the error $\delta_j^{[l]}$ and link it with $\frac{\partial C}{\partial b_j^{[l]}}$ and $\frac{\partial C}{\partial w_{jk}^{[l]}}$.

Lemma 1. *The four fundamental equations of back propagation are given by*

$$\delta^{[L]} = \sigma'(z^{[L]}) \odot (a^{[L]} - y), \tag{2.14}$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \odot (W^{[l+1]})^T \delta^{[l+1]}, \qquad \textit{for } 2 \le l \le L-1, \tag{2.15}$$

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]}, \qquad \textit{for } 2 \le l \le L, \tag{2.16}$$

$$\frac{\partial C}{\partial W_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]}, \qquad \textit{for } 2 \le l \le L. \tag{2.17}$$

Proof. We start with (2.14), which provides us an expression for the output error δ^L. By applying the chain rule in the relation (2.12) with $l = L$ we have

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \sum_{k=1}^{n_L} \frac{\partial C}{\partial a_k^{[L]}} \frac{\partial a_k^{[L]}}{\partial z_j^{[L]}} = \sum_{k=1}^{n_L} \frac{\partial C}{\partial \sigma(z_k^{[L]})} \frac{\partial \sigma(z_k^{[L]})}{\partial z_j^{[L]}}. \tag{2.18}$$

Since $\sigma(z_k^{[L]})$ depends on $z_j^{[L]}$ only when $k = j$, then (2.18) takes the form

$$\delta_j^{[L]} = \frac{\partial C}{\partial \sigma(z_j^{[L]})} \frac{\partial \sigma(z_j^{[L]})}{\partial z_j^{[L]}}. \tag{2.19}$$

Moreover, from (2.11),

$$\frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \sum_{k=1}^{n_L} \frac{1}{2}(y_k - a_k^{[L]})^2 = -(y_j - a_j^{[L]}) = (a_j^{[L]} - y_j) \tag{2.20}$$

Therefore, combining (2.19) and (2.20), we obtain

$$\delta_j^{[L]} = \frac{\partial C}{\partial \sigma(z_j^{[L]})} \frac{\partial \sigma(z_j^{[L]})}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j) \frac{\partial \sigma(z_j^{[L]})}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j)\sigma'(z_j^{[L]})$$

which is (2.14), in componentwise form. Furthermore, (2.15) provides a relation for $\delta^{[l]}$ in terms of $\delta^{[l+1]}$. In order to achieve that, we apply the chain rule and use (2.12), to get

$$\begin{aligned} \delta_j^{[l]} &= \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} \\ &= \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} \delta_k^{[l+1]} \end{aligned} \tag{2.21}$$

To determine $\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}$, we keep in mind that

$$z_k^{[l+1]} = \sum_{i=1}^{n_l} W_{ki}^{[l+1]} a_i^{[l]} + b_k^{[l+1]} = \sum_{i=1}^{n_l} W_{ki}^{[l+1]} \sigma(z_i^{[l]}) + b_k^{[l+1]}. \tag{2.22}$$

Thus, by differentiating we obtain,

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = W_{kj}^{[l+1]} \sigma'(z_j^{[l]}) = \sigma'(z_j^{[l]}) W_{kj}^{[l+1]}. \tag{2.23}$$

From (2.21) and (2.23) we obtain

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}} \sigma'(z_j^{[l]}) W_{kj}^{[l+1]} \delta_k^{[l+1]}, \tag{2.24}$$

which is the componentwise form of (2.15). Similarly, to prove (2.16) we apply the chain rule

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \delta_j^{[l]} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}}, \tag{2.25}$$

where $z_j^{[l]} = (W^{[l]}\sigma(z^{[l-1]}))_j + b_j^{[l]}$. So, since $z^{[l-1]}$ does not depend on $b_j^{[l]}$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \tag{2.26}$$

Substituting (2.26) back into (2.25) we obtain

$$\frac{\partial C}{\partial b_j^{[l]}} = \delta_j^{[l]},$$

which is (2.16) written in componentwise form.

At last, we use again the chain rule and the definition (2.12) to show (2.17),

$$\frac{\partial C}{\partial w_{ji}^{[l]}} = \sum_{k=1}^{n_l} \frac{\partial C}{\partial z_k^{[l]}} \frac{\partial z_k^{[l]}}{\partial w_{ji}^{[l]}} = \sum_{k=1}^{n_l} \delta_k^{[l]} \frac{\partial z_k^{[l]}}{\partial w_{ji}^{[l]}} \tag{2.27}$$

where the sum is over all neurons of the l^{th} layer. Recall that by definition $z_k^{[l]} = \sum_{i=1}^{n_l} W_{ki}^{[l]} a_i^{[l]} + b_k^{[l]}$. Obviously, $z_k^{[l]}$ depends only on $w_{ji}^{[l]}$ when $k = j$ and vanishes when $k \neq j$. Consequently, we have

$$\frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = a_i^{[l]}. \tag{2.28}$$

Substituting (2.28) back into (2.27) we have

$$\frac{\partial C}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} \frac{\partial z_j^{[l]}}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} a_i^{[l]},$$

which completes the proof of the Lemma. □

To summarise, the above four equations provide us with an algorithm to compute the partial derivatives $\frac{\partial C}{\partial w_{ji}^{[l]}}$ and $\frac{\partial C}{\partial b_j^{[l]}}$. At first, if we set the corresponding activation $a^{[1]}$ for the input layer, then equation (2.3) feed forward through the network to calculate $a^{[L]}$. Furthermore, commencing from the last layer, from relation (2.14) we compute $\delta^{[L]}$. Moreover, from (2.15) we are able to calculate $\delta^{[l]}$ backwards from the $(L-1)^{th}$ layer to the second layer. Finally, using (2.16) and (2.17) we compute the partial derivatives. The backward movement through the network is the reason why it is called back propagation. With the aim of eliminating Hadamard product from the equations, we present a diagonal matrix $D^{[l]} \in \mathbb{R}^{n_l \times n_l}$ where, $D_{ii}^{[l]} = \sigma'(z_i^{[l]})$. Thus, the equations (2.14) and (2.15) take the form of (2.29) and (2.30) respectively

$$\delta^{[L]} = D^{[L]}(a^{[L]} - y), \tag{2.29}$$

$$\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}. \tag{2.30}$$

Recall that the back propagation algorithm calculates the gradient of cost function for a fixed point x. Hence, the gradient of the cost function for the whole training set is the mean of the individual gradients over all training points,

$$C_{x_i} = \frac{1}{2}(\|y(x_i) - a^{[L]}(x_i)\|_2)^2,$$

$$\nabla cost(p) = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{x_i}(p).$$

Now, we are at a point where we are able to construct an iterative method to minimize the value of the cost function. Listing 6.2 in Section 6.2 presents a pseudocode which illustrates this process.

2.5 Simple Deep learning implementation

In this section we utilize the tools we gained from the previous sections in this chapter, to train a neural network for a simple use of deep learning [4]. As shown in Figure 2.5 we have ten labeled points. The points are split in two groups, group A displayed with circles and group B displayed with triangles. Our goal is to create

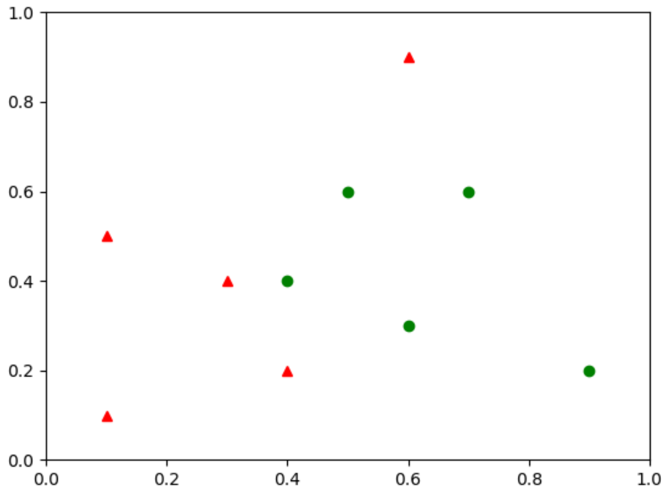

Figure 2.5: Labeled data points in $\mathbb{R}^2$

a model which can predict if a point in $\mathbb{R}^2$ belongs either in group A or in group B. The characterization of the two groups will be presented later in this section.

To construct that model we will use the Neural Network in Figure 2.6. Figure 2.6 depicts a Neural Network with 3 layers. The input layer (layer 1) consists of two neurons because of the fact that our data set points belongs in $\mathbb{R}^2$, therefore, they have two components. We recall from (2.3) that $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$, $b^{[l]} \in \mathbb{R}^{n_l \times 1}$ and $a^{[l]} \in \mathbb{R}^{n_l \times 1}$. The second layer has six neurons, thus $a^{[1]} \in \mathbb{R}^{2\times 1}$, $W^{[2]} \in \mathbb{R}^{6\times 2}$ and $b^{[2]} \in \mathbb{R}^{6\times 1}$. Consequently, the output from layer 2 has the form

$$a^{[1]} = x$$
$$a^{[2]} = \sigma(z^{[2]}) = \sigma(W^{[2]}a^{[1]} + b^{[2]}) \in \mathbb{R}^{6\times 1}$$

Furthermore, the output layer (layer 3) has 2 neurons. Following the same procedure, the weights and biases of the 3^{rd} layer take the shape of $W^{[3]} \in \mathbb{R}^{2\times 6}$ and $b^{[3]} \in \mathbb{R}^{2\times 1}$ respectively. Hence, the output of the network has the form of equation (2.31).

$$\begin{aligned} F(x) &= a^{[3]} = a^{[L]} \\ F(x) &= \sigma(z^{[3]}) = \sigma(W^{[3]}a^{[2]} + b^{[3]}) \\ F(x) &= \sigma(W^{[3]}\sigma(W^{[2]}a^{[1]} + b^{[2]}) + b^{[3]}) \in \mathbb{R}^{2\times 1} \end{aligned} \tag{2.31}$$

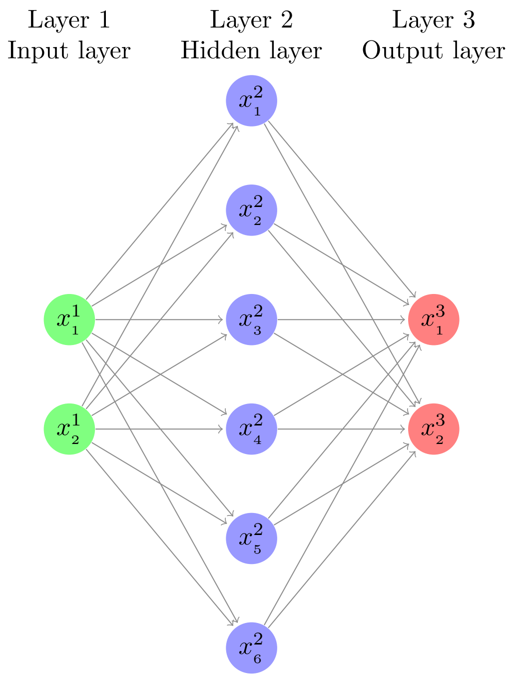

Figure 2.6: A Neural Network with one hidden layer

The number of parameters of the neural network shown in Figure 2.6 are 32. Therefore, the function $F(x) : \mathbb{R}^2 \to \mathbb{R}^2$ of the relation (2.31) except from the input value x is also defined in terms of those 32 parameters. With the purpose of using the cost function (2.4) we have to set target output $(y(x_i))$ for each input data point (x_i). Thus, the target output for data points of group A is $[0, 1]^T$ and the target output for data points of group B is $[1, 0]^T$.

Hence, for every new data point which $F(x)$ is close to the vector $[0, 1]^T$ we categorize it in group A and for each new data point which $F(x)$ is close to the vector $[1, 0]^T$ we categorize it in group B. Consequently, if the function $F(x)$ satisfies the inequality $F_1(x) < F_2(x)$ then, the new data point belongs in group A and if it satisfies the inequality $F_1(x) > F_2(x)$ then, the new data point belongs in group B.

The relations that we have extracted above, allow us to execute the pseudocode mentioned at the end of the previous section. Listing 6.3 in Section 6.3 demonstrates a simple application of the neural network training of the data set shown in

Figure 2.5 using the gradient descent algorithm and the back propagation method.

Figure 2.7 shows the behaviour of the cost function during the learning process. The value of the cost function started close to 5 and throughout the procedure it plummeted to 10^{-2}.

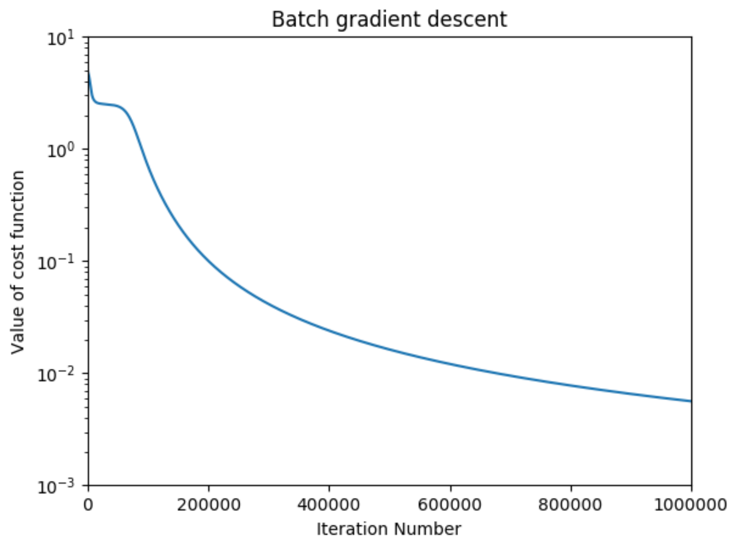

Figure 2.7: The graph above demonstrates the behaviour of the cost function throughout the learning process

Chapter 3

An overview of gradient descent optimization algorithms

3.1 Variations of the basic algorithm

There are three variants of gradient descent algorithm. The major difference between those three is the amount of data they use for each update of the parameters and in particular, to compute the gradient of the cost function. We compromise by making an exchange between the accuracy and the computational cost [14].

3.1.1 Batch gradient descent

Batch gradient descent or Vanilla gradient descent is when on every iteration we sum up the gradient of the cost function for each sample of the training set and then compute the mean of those individual gradients. Hence, for every update we have used the entire training set:

$$
\begin{aligned}
p_{i+1} &= p_i - \eta \nabla cost(p_i), \\
C_{x_i} &= \frac{1}{2}(\|y(x_i) - a^{[L]}(x_i)\|_2)^2, \\
\nabla cost(p) &= \frac{1}{N}\sum_{i=1}^{N} \nabla C_{x_i}(p).
\end{aligned}
$$

The code implementation takes the form:

```
for i in range(epochs):
  Gradient=compute_gradient(cost_function,parameters,data_points)
  parameters=parameters - learning_rate * Gradient
```

There are advantages and dis-advantages in this approach which are now highlight.

- **The main advantages**
  - **Computational efficiency:** This technique is less computationally demanding as all of computer resources are not used to process a single sample but the entire training set.
  - **Stable convergence:** It has less oscillations and noisy steps in the direction of the minimum because of the fact that we use the whole training set rather than a single sample to update the parameters. Furthermore, it is guaranteed to converge to the global minimum if the cost function is convex and to a local minimum if the cost function is not convex.
- **The main disadvantages**
  - **Slower learning:** The learning process of the Batch gradient descent is slow since the entire training set is used to perform an update of the parameters.
  - **Local minimums:** Throughout the learning process we can get trapped in a local minimum of the cost function and never make it to the global minimum. In view of the fact that we lack the noisy steps that will help us escape the local minimum and reach our goal, the global minimum.

3.1.2 Stochastic gradient descent

Stochastic gradient descent performs an update of the parameters for each sample of the training set rather than using all the samples. Suppose that our training set holds N samples. We shuffle our training set and perform N updates, one for every individual sample. When we do that, we complete an *epoch*. Therefore, learning occurs for every sample:

$$p_{i+1} = p_i - \eta \nabla cost(p_i, x_k, y_k)$$

Stochastic gradient descent in code takes the shape of:

```
for i in range(epochs):
  np.random.shuffle(data)
  for j in range(N):
    Gradient=compute_gradient(cost_function ,parameters ,sample)
    parameters=parameters - learning_rate * Gradient
```

- **The main advantages**
  - **Escape from local minimum:** As a result of the frequent updates, the steps taken in the direction of the cost function minimum have oscillations which can help us avoiding local minima of the cost function.
  - **Faster learning:** It can converge faster due to the frequent updates and its computational speed.
- **The main disadvantages**
  - **Computationally expensive:** Stochastic gradient is far more computationally expensive than the batch gradient descent due to using all the resources for processing one sample at a time.
  - **Inability to remain at the global minimum:** The algorithm is unable to remain at the global minimum of the cost function due to the noisiness of the process.

3.1.3 Mini batch gradient descent

Mini batch gradient descent is a mixture of stochastic gradient descent and batch gradient descent. Frequently, is mentioned as the default method to apply the gradient descent algorithm to deep neural networks, as it combines the advantages of the previous two algorithms. Mini batch introduces a new hyper-parameter m, where the term hyper-parameter is defined as the parameter that has to be chosen manually before training. We shuffle and separate the training set into *mini batches* of size m and cycle through every mini batch in a random order. The performing over all the mini batches is referred to as an *epoch* like in the stochastic gradient descent. Hence learning occurs for each mini batch of size m:

$$p_{i+1} = p_i - \eta \nabla cost(p_i, x_{k:k+m}, y_{k:k+m})$$

Mini batch gradient descent in code takes the shape of:

```
for i in range(epochs):
 np.random.shuffle(data)
 for j in range(N/m):
  Gradient=compute_gradient(cost_function,parameters,batch_m)
  parameters=parameters - learning_rate * Gradient
```

- **The main advantages**
  - **Computational efficiency:** In contrast with the stochastic gradient descent, we use all the resources to process a mini batch of samples rather than a single sample to perform an update.
  - **Faster learning:** We perform updates more often than the batch gradient descent, hence the network learns faster.
  - **Stable convergence:** As a result of the fact that we compute an average of the cost function gradient over m samples, that leads to less noise.
- **The main disadvantages**
  - **New parameter:** Compared with the previous two algorithms, mini

batch gradient descent has an additional hyper-parameter, the parameter m. The mini batch size could play a crucial role in the learning process. As a result of that is very important to find the proper m for the network.

- **Local minima:** Stable error gradient can lead us to a local minimum and in contrast with the stochastic gradient descent we do not have the noise that will assist us escape.

3.2 Gradient descent algorithms

The previous methods face some difficulties with the local minima and the speed of learning. The most critical parameter defined above is the learning rate. Some of the following algorithms try to set a different approach to the learning rate, by adapting it during the learning process and provide an alternative way of updating the parameters.

3.2.1 Gradient descent and ravines

Figure 3.1(a) depicts a surface where area A is shallow and area B is steep. The optimal would be if we take large steps when we are in area A and small steps in area B because of the fact that a flat surface may indicate that the optimum is far from reaching. Instead, gradient descent method updates are proportional to the gradient magnitude. Hence, the algorithm has large step size when it has a big gradient and the equivalent for small step size. An approach to solve that problem is to modify the learning rate (η) according to the gradient. On the other hand, Figure 3.1(b) display a tougher issue, where we have different slopes in several directions. The situation where the slope in one direction is steepest than the others is defined as a ravine. The appropriate approach would be to adapt the learning rate in every direction. However, is impossible due to the definition of the gradient descent, which is to follow the direction of the minimum, since by adjusting the learning rate the direction of the step may be changed [16].

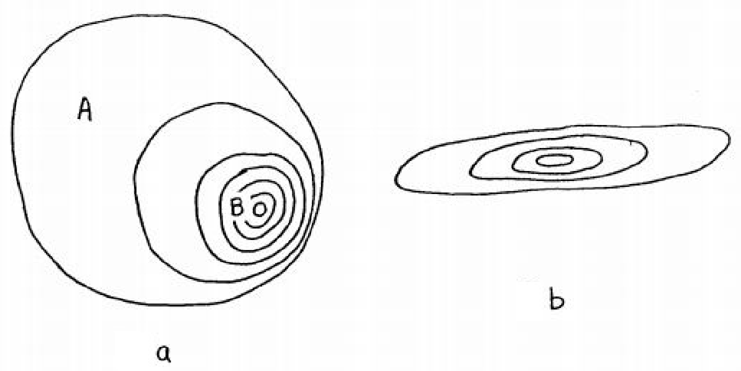

Figure 3.1: a) depicts a surface with different slopes in different areas and b) depicts a surface with slopes in different directions

3.2.2 Momentum

One of Gradient descent variants main difficulty is when they face ravines. Ravines are usually near local minima. In this situation the algorithm oscillates as shown in Figure 3.2(a) across the ravine rather than along the ravine towards the optimum, thus, the learning process slows down. Figure 3.2(b) displays how momentum [10] method overcomes this issue by accelerating the gradients towards the right direction,

$$v_{i+1} = \gamma v_i + \eta \nabla cost(p_i), \quad p_{i+1} = p_i - v_{i+1}$$

The γ term belongs in $[0, 1]$ and frequently set in a value close to 0.9. The momentum constant γ controls the decay of the velocity vector v, and values closer to 1 lead to higher velocities. The momentum term diminishes updates in the dimensions where the gradients direction alters and it amplifies them in the directions where the direction of the gradients remains the same.

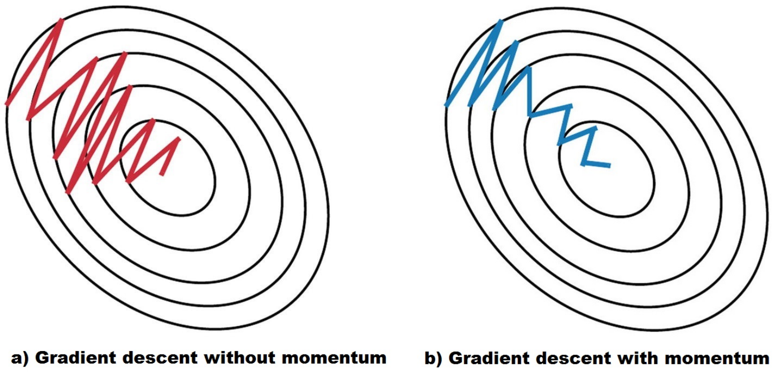

Figure 3.2: Gradient descent method behaviour in a ravine

3.2.3 Nesterov accelerated gradient (NAG)

NAG differs slightly from the momentum method. The main difference between these two algorithms is the computation of the gradient [7].

$$v_{i+1} = \gamma v_i + \eta \nabla cost(p_i - \gamma v_i), \quad p_{i+1} = p_i - v_{i+1}$$

Momentum calculates the gradient prior applying the velocity, while NAG calculates the gradient after applying the velocity. This disparity enables NAG to modify v in a faster and more robust way. The fundamental thought behind the Nesterov accelerated gradient is that when the parameters vector p is at position i, then the first term of the momentum update is about to push p by γv. Hence, we use $p_i - \gamma v_i$ as an approximation of the parameters next position. Therefore, instead of computing the gradient with respect to p_i like momentum does, we use the approximation $p_i - \gamma v_i$ to calculate the gradient. Figure 3.3 shows that with Nesterov accelerated gradient rather than computing the gradient at the current position, it instead uses the approximate future position to calculate the gradient.

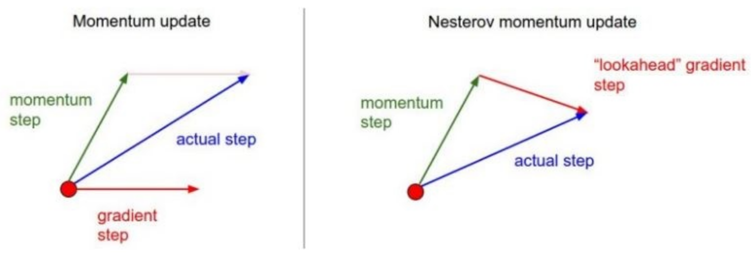

Figure 3.3: One iteration of momentum method (left) and one iteration of Nesterov accelerated gradient method (right)

3.2.4 Adagrad

The common hyper-parameter for all gradient descent algorithms is the learning rate (η). Unfortunately, it is very hard to find its optimal value. As a result of the fact that if we set it very small the learning process would be slow and it will take considerable time to reach an acceptable cost function. On the other hand, if we set it too large, there is a possibility that the parameters will run throughout the cost function and they may be unable to reach a tolerable cost. If our parameter vector $p \in \mathbb{R}^n$, then we are facing a non-convex cost function in $\mathbb{R}^n$, which could get us different sensitivity on each dimension.

The Adagrad algorithm [2] attempts to cope with that problem by adjusting the magnitude of the learning rate in each dimension. Thus, the Adagrad componentwise parameter update takes the form of:

$$p_{i+1,j} = p_{i,j} - \frac{\eta}{\sqrt{G_{i,jj} + \epsilon}} g_{i,j}, \tag{3.1}$$

where η is the learning rate constant, ϵ is a small quantity that prevents the division by 0, g_i is the gradient of the cost function at the time step i which we can compute with the equation:

$$g_i = \nabla cost(p_i), \quad g_{i,j} = \nabla cost(p_{i,j}),$$

and G_i denote a $d \times d$ diagonal matrix with (j, j) entry given by the sum of the squares of the gradients with respect to p_j in the time step i.

$$G_{i,jj} = \sum_{k=1}^{i} (g_{k,j})^2.$$

Therefore, using the Hadamard product in (3.1) gives

$$p_{i+1} = p_i - \frac{\eta}{\sqrt{G_i + \epsilon}} \odot g_i.$$

Adagrad major disadvantage is that the learning rate diminishes pretty fast due to the accumulation of the cost function gradients since the launch of the training. Hence, there comes a time that the model is unable to acquire additional knowledge because the learning rate value is close to zero. This issue is mitigated by the following algorithms.

3.2.5 RMSprop

Root mean square prop or RMSprop is an adaptive learning rate method that attempts to upgrade the Adagrad algorithm. Rather than amass the sum of the past squared gradients in the denominator like Adagrad, RMSprop expresses the sum of the past squared gradients as a decaying average of these gradients. Identical to momentum, this decaying average is the exponential moving average of current and previous gradients.

$$p_{i+1} = p_i - \frac{\eta}{\sqrt{E[g^2]_i + \epsilon}} g_i,$$

where, $\sqrt{E[g^2]_i + \epsilon}$ is defined as the root mean squared (RMS) error. Therefore,

$$p_{i+1} = p_i - \frac{\eta}{RMS[g]_i} g_i$$

with,

$$E[g^2]_i = \gamma E[g^2]_{i-1} + (1 - \gamma) g_i^2.$$

3.2.6 Adadelta

Adadelta as well as RMSprop seek to remedy the aggressive diminish of the Adagrad learning rate. The difference between these two algorithms is that Adadelta eliminates the application of the learning rate term η. We recall that the Stochastic gradient descent update in terms of Δp has the form:

$$\Delta p_i = -\eta g_i,$$
$$p_{i+1} = p_i + \Delta p_i.$$

Therefore the RMSprop update may be rearranged as

$$\Delta p_i = -\frac{\eta}{RMS[g]_i} g_i,$$
$$p_{i+1} = p_i + \Delta p_i.$$

The authors in [19] detected a missmatch in Stochastic gradient descent, Momentum, Adagrad and RMSprop units. They attempted to modified the RMSprop update in an effort to match the units of the parameters. In order to achieve that they replaced the learning rate term (η) in the numerator. Since, Δp at the current time step is unknown, the exponentially decaying average RMS over the previous Δp provides us with an approximation for Δp at the current time step.

$$\Delta p_i = -\frac{RMS[\Delta p]_{i-1}}{RMS[g]_i} g_i$$
$$p_{i+1} = p_i + \Delta p_i$$

where,

$$E[\Delta p^2]_i = \gamma E[\Delta p^2]_{i-1} + (1-\gamma)\Delta p_i^2$$
$$RMS[\Delta p]_i = \sqrt{E[\Delta p^2]_i + \epsilon}$$

3.2.7 Adam

Adaptive moment estimation or Adam is a combination of Momentum and RMSprop [5]. This method uses estimates of first and second moments of the gradients to calculate adaptive learning rates for each parameter. Where, the i^{th} moment of a random variable X is the expected value of that variable in the i^{th} power

$$m_i = E[X^i].$$

We assume that the gradient of the cost function is a random variable due to the fact that it is often computed in a random mini batch of data. Therefore, the first moment ($E[g]_i$) is the mean and the second moment ($E[g^2]_i$) is the uncentered variance, because of the fact that we do not subtract the mean through the variance computation. Adam utilizes exponentially moving averages of past gradients (m_i) and of past squared gradients (v_i), to estimate the first and second moment of the gradients respectively

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1) g_i,$$
$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) g_i^2.$$

Since, m_i and v_i are initialized as vectors of zeros, they are biased towards zero. In order to address that issue the estimators take the following form:

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i},$$
$$\hat{v}_i = \frac{v_i}{1 - \beta_2^i}.$$

Finally, we obtain the Adam update rule:

$$p_{i+1} = p_i - \frac{\eta}{\sqrt{\hat{v}_i} + \epsilon} \hat{m}_i.$$

3.2.8 AdaMax

AdaMax is a variant of Adam algorithm derived from the infinity norm. The Adam update rule term v_i, scales the gradients of each individual parameter inversely proportional to a ℓ^2 norm of their past (v_{i-1}) and current (g_i^2) gradients.

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) |g_i|^2.$$

The authors of the AdaMax method [5] generalize the ℓ^2 update rule to a ℓ^t based update rule.

$$v_i = \beta_2^t v_{i-1} + (1 - \beta_2^t) |g_i|^t,$$
$$v_i = (1 - \beta_2^t) \sum_{j=1}^{i} \beta_2^{t(i-j)} |g_i|^t.$$

Moreover, the norms for big t are numerically unstable. Despite that, for $t \to \infty$ the norm illustrate a stable behaviour. Because of that, the authors of the AdaMax proved that v_i with ℓ^∞ converges to the following value.

$$u_i = \beta_2^\infty v_{i-1} + (1 - \beta_2^\infty) |g_i|^\infty,$$
$$u_i = max(\beta_2 v_{i-1}, |g_i|).$$

We replace v_i with u_i to prevent the confusion with Adam. Therefore, u_i takes the place of $\sqrt{\hat{v}_i} + \epsilon$ and yield the AdaMax update rule:

$$p_{i+1} = p_i - \frac{\eta}{u_i}\hat{m}_i,$$

where,

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i}.$$

3.2.9 Nadam

Nesterov-accelerated adaptive moment estimation or Nadam [1], is a combination of Nesterov accelerated gradient (NAG) and Adam. We recall that Adam is a combination of RMSprop and Momentum. The Momentum and NAG update rules, are the relations (3.2) and (3.3) respectively.

$$\begin{aligned} g_i &= \nabla cost(p_i), \\ m_i &= \gamma m_{i-1} + \eta g_i, \\ p_{i+1} &= p_i - m_i, \end{aligned} \tag{3.2}$$

$$\begin{aligned} g_i &= \nabla cost(p_i - \gamma m_{i-1}), \\ m_i &= \gamma m_{i-1} + \eta g_i, \\ p_{i+1} &= p_i - m_i. \end{aligned} \tag{3.3}$$

Moreover, in (3.2) we replace the symbol m_i in the parameter update with the definition for m_i

$$p_{i+1} = p_i - (\gamma m_{i-1} + \eta g_i), \tag{3.4}$$

The authors of Nadam modified the NAG update rule, in order to use it instead of the momentum in the Adam update rule.

$$\begin{aligned} g_i &= \nabla cost(p_i), \\ m_i &= \gamma m_{i-1} + \eta g_i, \\ \hat{m}_i &= \gamma m_i + \eta g_i, \\ p_{i+1} &= p_i - (\gamma m_i + \eta g_i). \end{aligned} \tag{3.5}$$

In (3.5) $\hat{m}_i$ consists of the gradient at the current time step (ηg_i) in addition to the momentum vector at the current time step (γm_i). Thus, in the NAG update rule (3.5) we use the current momentum vector to update the parameters, instead of the previous momentum vector as in the momentum update rule (3.4). Now, we recall the Adam update rule

$$\begin{aligned} m_i &= \beta_1 m_{i-1} + (1-\beta_1) g_i, \\ \hat{m}_i &= \frac{m_i}{1-\beta_1^i}, \\ p_{i+1} &= p_i - \frac{\eta}{\sqrt{\hat{v}_i}+\epsilon}\hat{m}_i, \end{aligned} \tag{3.6}$$

which may be rearranged as

$$\begin{aligned} p_{i+1} &= p_i - \frac{\eta}{\sqrt{\hat{v}_i}+\epsilon}\left(\frac{m_i}{1-\beta_1^i}\right), \\ p_{i+1} &= p_i - \frac{\eta}{\sqrt{\hat{v}_i}+\epsilon}\left(\frac{\beta_1 m_{i-1} + (1-\beta_1) g_i}{1-\beta_1^i}\right), \\ p_{i+1} &= p_i - \frac{\eta}{\sqrt{\hat{v}_i}+\epsilon}\left(\frac{\beta_1 m_{i-1}}{1-\beta_1^i} + \frac{(1-\beta_1) g_i}{1-\beta_1^i}\right). \end{aligned} \tag{3.7}$$

We note that $\beta_1^i \approx \beta_1^{i-1}$ because of the fact that β_1 value is close to 1. Thus, $\frac{m_{i-1}}{1-\beta_1^i} \approx \frac{m_{i-1}}{1-\beta_1^{i-1}}$, which is equal to $\hat{m}_{i-1}$. Hence,

$$p_{i+1} = p_i - \frac{\eta}{\sqrt{\hat{v}_i}+\epsilon}\left(\beta_1 \hat{m}_{i-1} + \frac{(1-\beta_1) g_i}{1-\beta_1^i}\right). \tag{3.8}$$

Finally, to obtain the Nadam update rule we apply the NAG update rule in (3.8) simply by converting the bias corrected estimate momentum vector from $\hat{m}_{i-1}$ to $\hat{m}_i$

$$p_{i+1} = p_i - \frac{\eta}{\sqrt{\hat{v}_i}+\epsilon}\left(\beta_1 \hat{m}_i + \frac{(1-\beta_1) g_i}{1-\beta_1^i}\right).$$

In Table 3.1 we summarize the gradient descent algorithms and the components they act upon. Furthermore, in Table 3.2 we present typical values of the parameters involved in these variations of the gradient descent algorithm.

Algorithm	**Learning Rate**	**Gradient**
Momentum		✓
NAG		✓
Adagrad	✓	
RMSprop	✓	
Adadelta	✓	
Adam	✓	✓
AdaMax	✓	✓
Nadam	✓	✓

Table 3.1: Gradient descent algorithms and the components they act upon

Algorithm	**Default Hyper-Parameter Values**				
	η	γ	ϵ	β_1	β_2
Momentum	0.01	0.9			
NAG	0.01	0.9			
Adagrad	0.01		10^{-7}		
RMSprop	0.001	0.9	10^{-6}		
Adadelta		0.95	10^{-6}		
Adam	0.001		10^{-8}	0.9	0.999
AdaMax	0.002			0.9	0.999
Nadam	0.002		10^{-7}	0.9	0.999

Table 3.2: Proposed default hyper-parameter values for each gradient descent algorithm

Chapter 4

A comparison of gradient descent algorithms

4.1 Numerical Results

The objective of this chapter is to display the numerical results of a comparative study of the aforementioned gradient descent optimization algorithms. They were implemented through the three basic gradient descent variants. These are, the batch gradient descent, the stochastic gradient descent and the mini-batch gradient descent. The comparative results were acquired using three different neural networks, which differed in terms of hidden layers and hidden neurons. Optimization was performed for the problem we discussed in Section 2.5. Numerical comparison is based on reliability and efficiency, evaluated by the computational time and the number of iterations required to obtain a certain accuracy level. The numerical codes were implemented in Python, using the time library to execute the aforementioned task.

4.1.1 Neural Network 1

The six following tables emerged using the neural network shown in Figure 4.1, which consists of two hidden layers, of two and three neurons each. The Tables

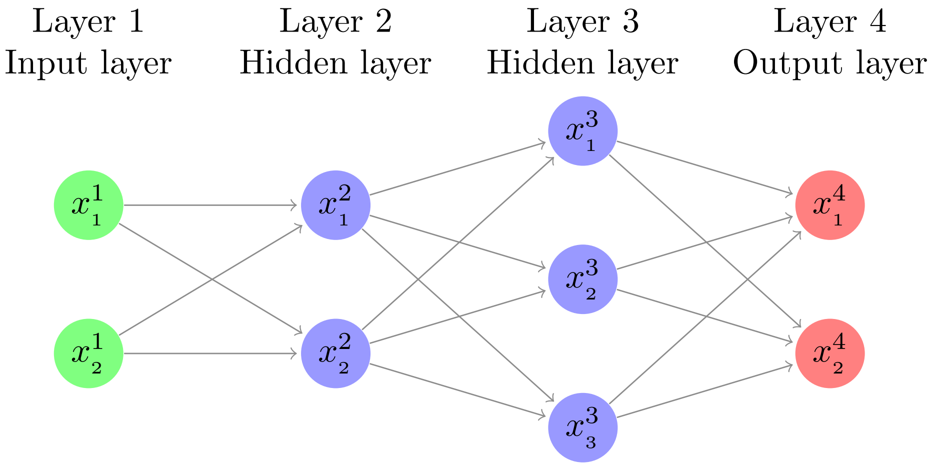

Figure 4.1: A neural network with four layers

4.1, 4.3 and 4.5 represent the number of iterations needed for the cost function to achieve the value of 10^{-3} through the methods of stochastic gradient descent, batch gradient descent and mini-batch gradient descent respectively. At the first glance, it is clear that, the RMSprop is the fastest algorithm to reach the requested cost function value in all three tables. Making the batch gradient descent the first method to obtain that goal at just 4.23 seconds and 759 iterations. On the other hand, as it is reasonable, the slowest algorithms of each table are the basic methods. Furthermore, Adadelta is the second slowest algorithm for the stochastic gradient descent and mini-batch gradient descent methods, while NAG is the second slowest algorithm for the batch gradient descent method. Moreover, the Tables 4.2, 4.4 and 4.6 display the accuracy achieved after 5×10^5 iterations for each algorithm. As can be seen, the algorithm that performed the best in all three basic methods is AdaMax, which managed to reach a value below 10^{-33} for the cost function. Also, apart from the basic methods, the algorithms of Momentum, NAG and Adagrad take the last three places, showing a mild improvement for the batch and mini-batch gradient descent methods compared with the stochastic gradient descent method.

Stochastic Gradient Descent		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Stochastic Gradient	711150	2038.06
Adadelta	130015	443.56
Adagrad	119327	362.77
NAG	72787	216.61
Momentum	72710	216.05
Adam	40592	154.49
Nadam	20601	95.33
AdaMax	10712	47.39
RMSprop	4865	17.29

Table 4.1: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Stochastic gradient descent using the neural network in Figure 4.1

Stochastic Gradient Descent		
Iterations$=5 \times 10^{5}$		
Algorithm	**Accuracy**	**Time(s)**
AdaMax	8.44478×10^{-35}	1929.41
Adam	1.98911×10^{-9}	1857.94
Adadelta	2.73713×10^{-5}	1624.31
Adagrad	1.24239×10^{-4}	1576.78
Momentum	1.02594×10^{-4}	1563.54
RMSprop	1.65404×10^{-8}	1523.25
NAG	1.02986×10^{-4}	1500.33
Nadam	1.21417×10^{-9}	1478.25
Stochastic Gradient	1.58721×10^{-3}	1436.04

Table 4.2: Accuracy each algorithm achieves by optimizing Stochastic gradient descent using the neural network in Figure 4.1 after 5×10^{5} iterations

Batch Gradient Descent		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Batch Gradient	705304	3914.01
NAG	70569	371.62
Momentum	70572	357.45
Adadelta	35293	190.87
AdaMax	6433	35.13
Adam	3257	17.46
Adagrad	2509	13.34
Nadam	2392	12.86
RMSprop	759	4.23

Table 4.3: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Batch gradient descent using the neural network in Figure 4.1

Batch Gradient Descent		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
AdaMax	1.09075×10^{-34}	3641.45
Adam	3.11041×10^{-9}	3406.66
Momentum	9.82960×10^{-5}	2937.08
Batch Gradient	1.57756×10^{-3}	2884.64
Nadam	2.03792×10^{-9}	2823.54
RMSprop	2.43892×10^{-8}	2734.92
Adadelta	4.55304×10^{-5}	2720.77
Adagrad	2.71766×10^{-6}	2676.37
NAG	9.82918×10^{-5}	2650.60

Table 4.4: Accuracy each algorithm achieves by optimizing Batch gradient descent using the neural network in Figure 4.1 after 5×10^5 iterations

Mini-Batch Gradient Descent ($m = 3$)		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Mini-Batch Gradient	704838	2474.19
Adadelta	71319	270.62
NAG	71695	257.93
Momentum	70899	250.61
Adagrad	25008	89.64
Adam	21161	74.97
Nadam	13876	49.13
AdaMax	5782	20.32
RMSprop	1895	6.79

Table 4.5: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Mini-Batch gradient descent using the neural network in Figure 4.1

Mini-Batch Gradient Descent ($m = 3$)		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
NAG	9.97117×10^{-5}	2186.36
AdaMax	7.53399×10^{-34}	2168.82
Nadam	1.57062×10^{-9}	2126.13
Mini-Batch Gradient	1.57653×10^{-3}	1923.97
Adagrad	3.77075×10^{-5}	1832.39
Adadelta	3.69710×10^{-5}	1771.98
Adam	2.06351×10^{-9}	1752.29
RMSprop	2.18096×10^{-8}	1743.32
Momentum	1.00218×10^{-4}	1742.57

Table 4.6: Accuracy each algorithm achieves by optimizing Mini-Batch gradient descent using the neural network in Figure 4.1 after 5×10^5 iterations

4.1.2 Neural Network 2

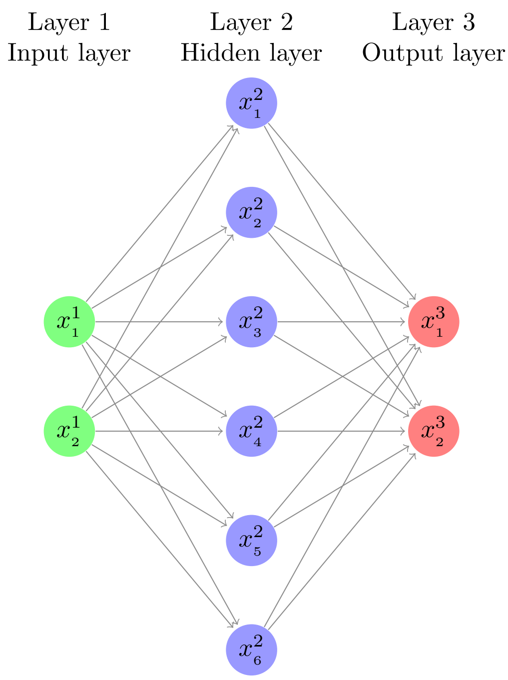

Figure 4.2: A neural network with a six neurons hidden layer

The second neural network we used is shown in Figure 4.2 and contains only one hidden layer, compared to the neural network in Figure 4.1. In the same way as before, the Tables 4.7, 4.9 and 4.11 illustrate a race between the algorithms in order to reach the desired accuracy. Again, the RMSprop is the most descent algorithm, while the algorithms acting upon only on the gradient (Momentum, NAG) take the last spots. Also, Tables 4.8, 4.10 and 4.12 illustrate that after 5×10^5 iterations the algorithm that perform better is the AdaMax. In addition, same as before, Momentum and NAG are the least reliable algorithms. The neural network in Figure 4.2 contains 32 parameters, 9 more than the neural network in Figure 4.1. The number of the parameters is proportional to the computational cost, but instead of improving the algorithms performance, the numerical results indicate that they perform the same or even worse compare to the previous neural network. Consequently, the number of hidden layers plays a more vital role to the performance of the network than the number of neurons.

Stochastic Gradient Descent		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Stochastic Gradient	3178639	10567.91
Momentum	334829	1469.91
NAG	335352	1463.18
Adagrad	256014	1227.89
Adadelta	288045	1091.79
Adam	25513	76.20
Nadam	19357	58.42
AdaMax	8132	24.79
RMSprop	5696	20.09

Table 4.7: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Stochastic gradient descent using the neural network in Figure 4.2

Stochastic Gradient Descent		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
Adagrad	2.82006×10^{-4}	2440.43
Adadelta	7.23279×10^{-4}	2386.75
Momentum	6.29698×10^{-4}	2041.05
Stochastic Gradient	1.29995×10^{-2}	1572.44
Nadam	8.24029×10^{-9}	1564.31
AdaMax	2.00121×10^{-32}	1509.36
Adam	1.24174×10^{-8}	1488.67
RMSprop	1.02048×10^{-7}	1457.14
NAG	6.49063×10^{-4}	1383.23

Table 4.8: Accuracy each algorithm achieves by optimizing Stochastic gradient descent using the neural network in Figure 4.2 after 5×10^5 iterations

Batch Gradient Descent		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Batch Gradient	3822124	17467.84
NAG	382755	2797.03
Momentum	382817	2563.48
Adadelta	190727	1115.83
AdaMax	9030	53.08
Adagrad	6388	36.87
Adam	6255	34.34
Nadam	5471	32.22
RMSprop	1044	5.81

Table 4.9: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Batch gradient descent using the neural network in Figure 4.2

Batch Gradient Descent		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
NAG	7.28562×10^{-4}	3872.95
AdaMax	4.96527×10^{-34}	3550.16
Adam	1.82539×10^{-8}	3302.43
Adadelta	3.28114×10^{-4}	3180.89
RMSprop	1.49884×10^{-7}	3040.29
Momentum	7.28562×10^{-4}	3039.78
Adagrad	7.74168×10^{-6}	3018.01
Nadam	1.76741×10^{-8}	2842.28
Batch Gradient	1.62925×10^{-2}	2387.31

Table 4.10: Accuracy each algorithm achieves by optimizing Batch gradient descent using the neural network in Figure 4.2 after 5×10^5 iterations

Mini-Batch Gradient Descent $(m = 3)$		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Mini-Batch Gradient	3818547	14552.97
Momentum	331308	1073.32
NAG	324718	1065.01
Adadelta	174094	639.71
Adagrad	41162	155.37
Adam	16707	60.52
Nadam	13268	50.02
AdaMax	5373	19.60
RMSprop	2289	8.30

Table 4.11: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Mini-Batch gradient descent using the neural network in Figure 4.2

Mini-Batch Gradient Descent $(m = 3)$		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
Adagrad	5.83938×10^{-5}	2297.35
Adadelta	2.98317×10^{-4}	2285.73
RMSprop	1.24599×10^{-7}	2241.79
AdaMax	3.97643×10^{-33}	2233.41
Adam	1.36019×10^{-8}	2047.73
NAG	6.08158×10^{-4}	1964.46
Nadam	8.76897×10^{-9}	1917.39
Momentum	6.11061×10^{-4}	1861.09
Mini-Batch Gradient	1.64631×10^{-2}	1826.25

Table 4.12: Accuracy each algorithm achieves by optimizing Mini-Batch gradient descent using the neural network in Figure 4.2 after 5×10^5 iterations

4.1.3 Neural Network 3

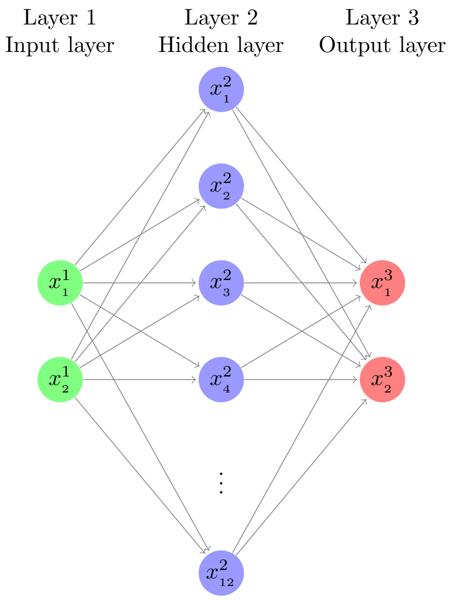

Figure 4.3: A neural network with a twelve neurons hidden layer

The last neural network we utilize is shown in Figure 4.3. The difference from the neural network in Figure 4.2 is the number of neurons in the hidden layer. Like the previous two networks, the algorithm that prevails in the Tables related to the convergence speed (4.13, 4.15 and 4.17) is the RMSprop, while Momentum and NAG are the slowest algorithms. Moreover, once again after 5×10^5 iterations, AdaMax is the most reliable algorithm ,but its accuracy shows a significant decrease compare to the previous networks. Furthermore, the neural network in Figure 4.3 has 62 parameters, almost twice the parameters neural network in Figure 4.2 has. While we were expecting this to have a positive impact on the algorithms, some of them perform worse than the previous network. It can be clearly seen that, depending on which of the three gradient descent methods we chose, we make a trade-off between the accuracy and the computational cost. Taking that into consideration, the Mini-Batch method seems to be the most reliable.

Stochastic Gradient Descent		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Stochastic Gradient	3093835	12217.18
Momentum	315148	1691.33
NAG	320733	1511.33
Adadelta	176626	953.61
Adagrad	73544	367.32
Adam	21487	117.02
Nadam	16227	84.04
AdaMax	6861	37.82
RMSprop	4524	20.78

Table 4.13: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Stochastic gradient descent using the neural network in Figure 4.3

Stochastic Gradient Descent		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
Adam	1.03454×10^{-8}	3224.27
Nadam	6.91298×10^{-9}	2876.20
Stochastic Gradient	1.14957×10^{-2}	2735.02
RMSprop	1.13768×10^{-7}	2583.66
Adadelta	2.34165×10^{-4}	2265.17
AdaMax	6.42469×10^{-9}	2245.44
Adagrad	7.68123×10^{-5}	1979.66
NAG	5.79956×10^{-4}	1907.72
Momentum	5.38247×10^{-4}	1900.26

Table 4.14: Accuracy each algorithm achieves by optimizing Stochastic gradient descent using the neural network in Figure 4.3 after 5×10^5 iterations

Batch Gradient Descent		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Batch Gradient	3087344	21440.17
Momentum	308760	2092.41
NAG	308771	2082.33
Adadelta	152411	1367.66
AdaMax	22020	187.26
Adagrad	4467	40.99
Adam	5044	40.65
Nadam	3884	32.53
RMSprop	846	6.83

Table 4.15: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Batch gradient descent using the neural network in Figure 4.3

Batch Gradient Descent		
Iterations$=5 \times 10^5$		
Algorithm	**Accuracy**	**Time(s)**
Adam	1.15351×10^{-8}	5068.83
RMSprop	1.23825×10^{-7}	4958.70
AdaMax	3.25996×10^{-10}	4544.40
Adagrad	5.53320×10^{-6}	4279.17
Adadelta	2.51489×10^{-4}	4204.81
Nadam	7.62408×10^{-9}	3940.99
Batch Gradient	1.15511×10^{-2}	3873.48
Momentum	5.64308×10^{-4}	3719.04
NAG	5.64351×10^{-4}	3528.93

Table 4.16: Accuracy each algorithm achieves by optimizing Batch gradient descent using the neural network in Figure 4.3 after 5×10^5 iterations

Mini-Batch Gradient Descent $(m = 3)$		
Tolerance$=10^{-3}$		
Algorithm	**Iterations**	**Time(s)**
Mini-Batch Gradient	3084494	14550.75
Momentum	306620	2253.78
NAG	307190	2041.29
Adadelta	179383	918.25
Adagrad	30817	234.41
Adam	15357	109.78
Nadam	12559	104.49
AdaMax	6007	50.24
RMSprop	1988	10.73

Table 4.17: Number of iterations needed to reach the accuracy of 10^{-3} for each algorithm by optimizing the Batch gradient descent using the neural network in Figure 4.3

Mini-Batch Gradient Descent $(m = 3)$		
Iterations$=5 \times 10^{5}$		
Algorithm	**Accuracy**	**Time(s)**
Adam	1.15634×10^{-8}	2808.75
Adadelta	2.58372×10^{-4}	2729.45
Nadam	7.86205×10^{-9}	2705.95
AdaMax	4.70701×10^{-9}	2693.65
NAG	5.60687×10^{-4}	2618.59
RMSprop	1.16606×10^{-7}	2564.99
Mini-Batch Gradient	1.15549×10^{-2}	2556.35
Adagrad	1.74143×10^{-5}	2437.54
Momentum	5.59736×10^{-4}	2028.79

Table 4.18: Accuracy each algorithm achieves by optimizing Mini-Batch gradient descent using the neural network in Figure 4.3 after 5×10^{5} iterations

4.2 Visualization of the gradient descent algorithms

The following animations from Alec Radford [11] illustrate the behaviour of the algorithms under certain circumstances. Unfortunately, the Adam algorithm and its variants (Nadam, AdaMax) are missing. The animation in Figure 4.5 depicts how the algorithms interact with the Beale's lost function. As a result of the large initial gradient, the algorithms based on the gradient are unstable at first. While, the algorithms that act upon the learning rate managed to handle the large gradient with more stability. Furthermore, the animation in Figure 4.4 reveals the behaviour of the algorithms when they face a long valley. Momentum and NAG oscillate until they finally break symmetry, while Stochastic gradient descent did not manage to escape. On the other hand, Adagrad, Adadelta and RMSprop immediately break symmetry and directed towards the negative slope. Finally, the animation in Figure 4.6 shows a saddle point, which as we defined above is a point where the curvature along different direction has different signs. The behaviour of the algorithms is similar to the previous animation. Momentum and NAG like to investigate the area before finding the right path, while RMSprop, Adagrad and Adadelta quickly proceed.

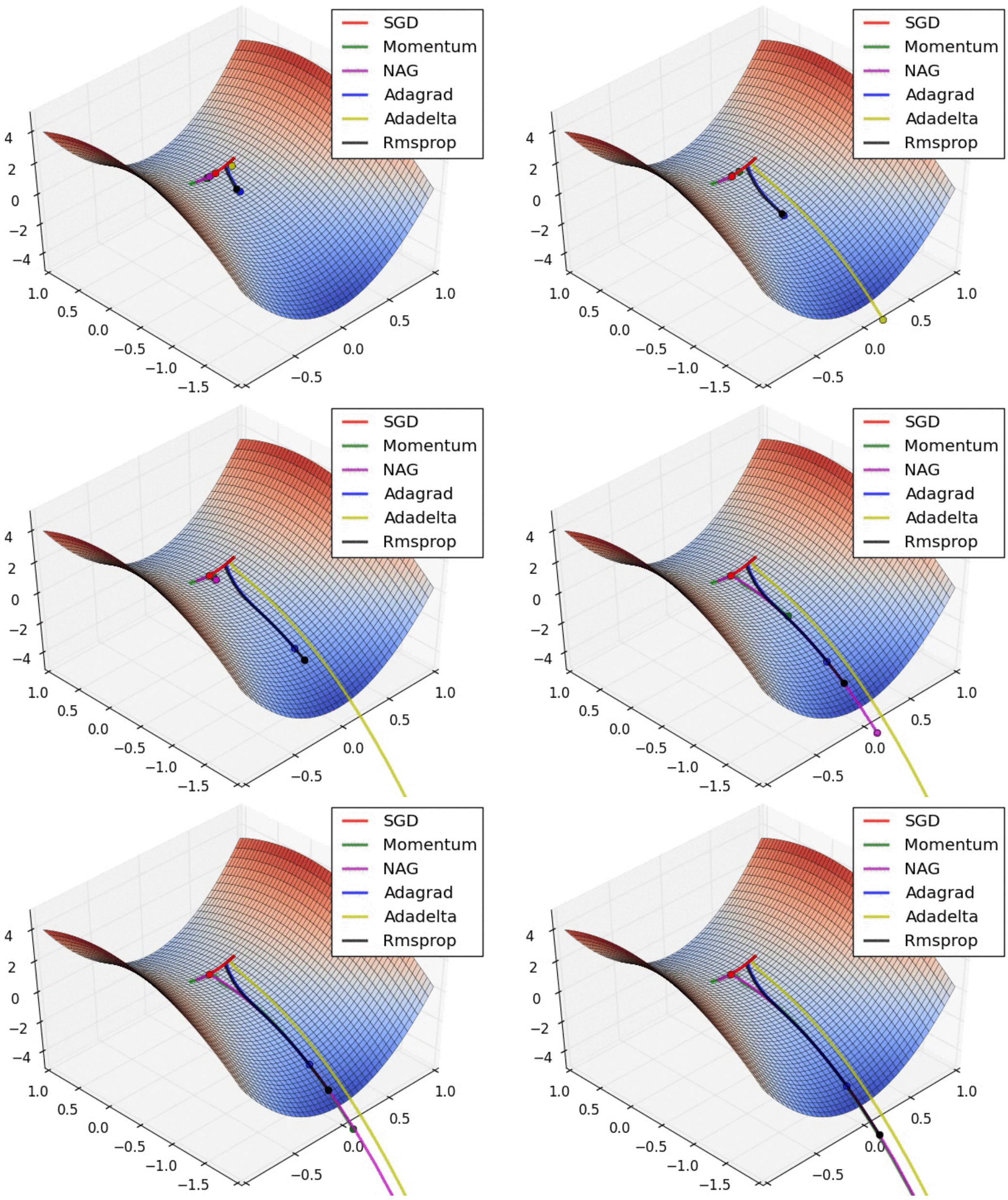

Figure 4.4: Algorithms iterative progress from top left to bottom right frame on a "long valley"

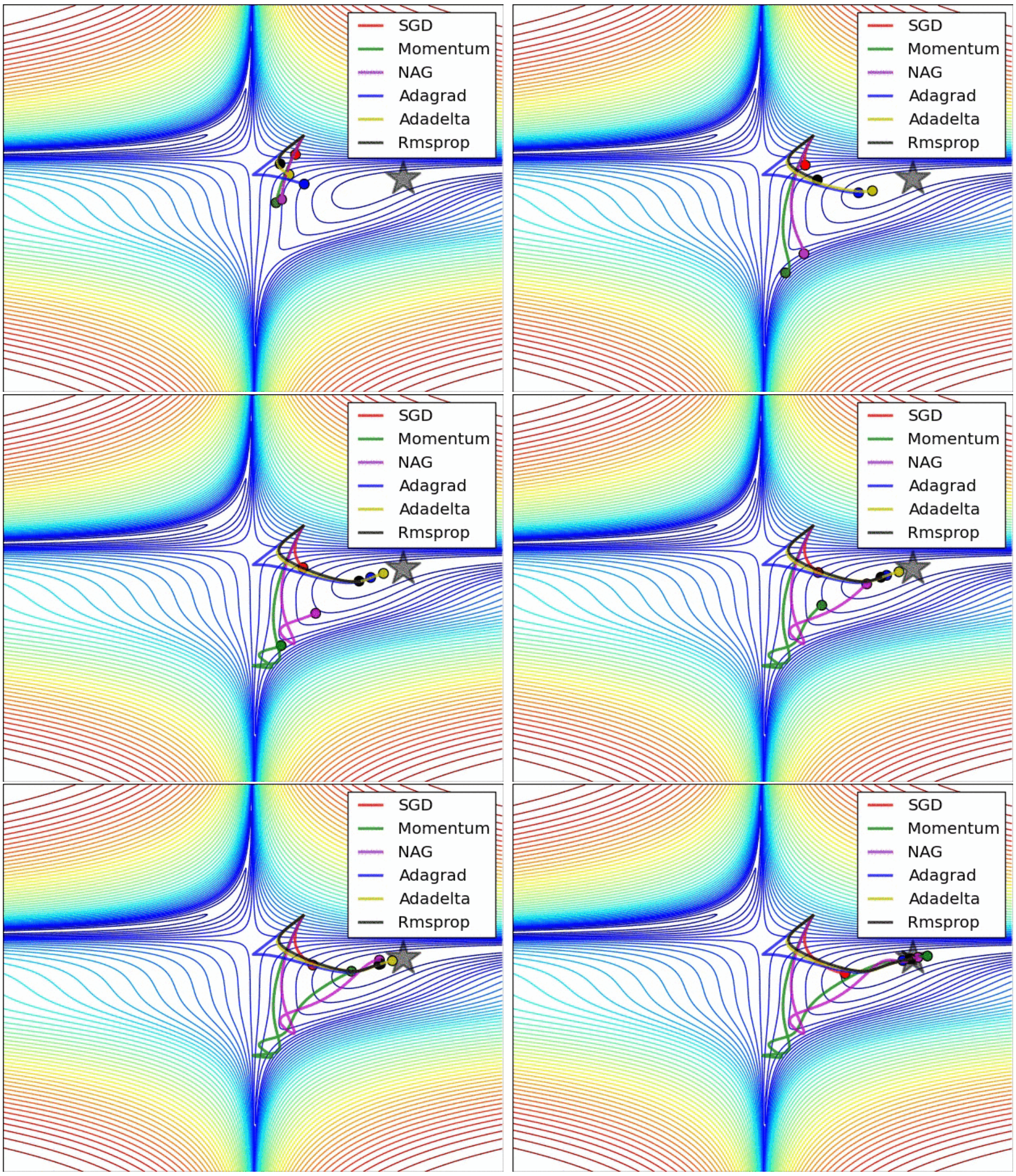

Figure 4.5: Algorithms iterative progress from top left to bottom right frame on Beale's function surface contours

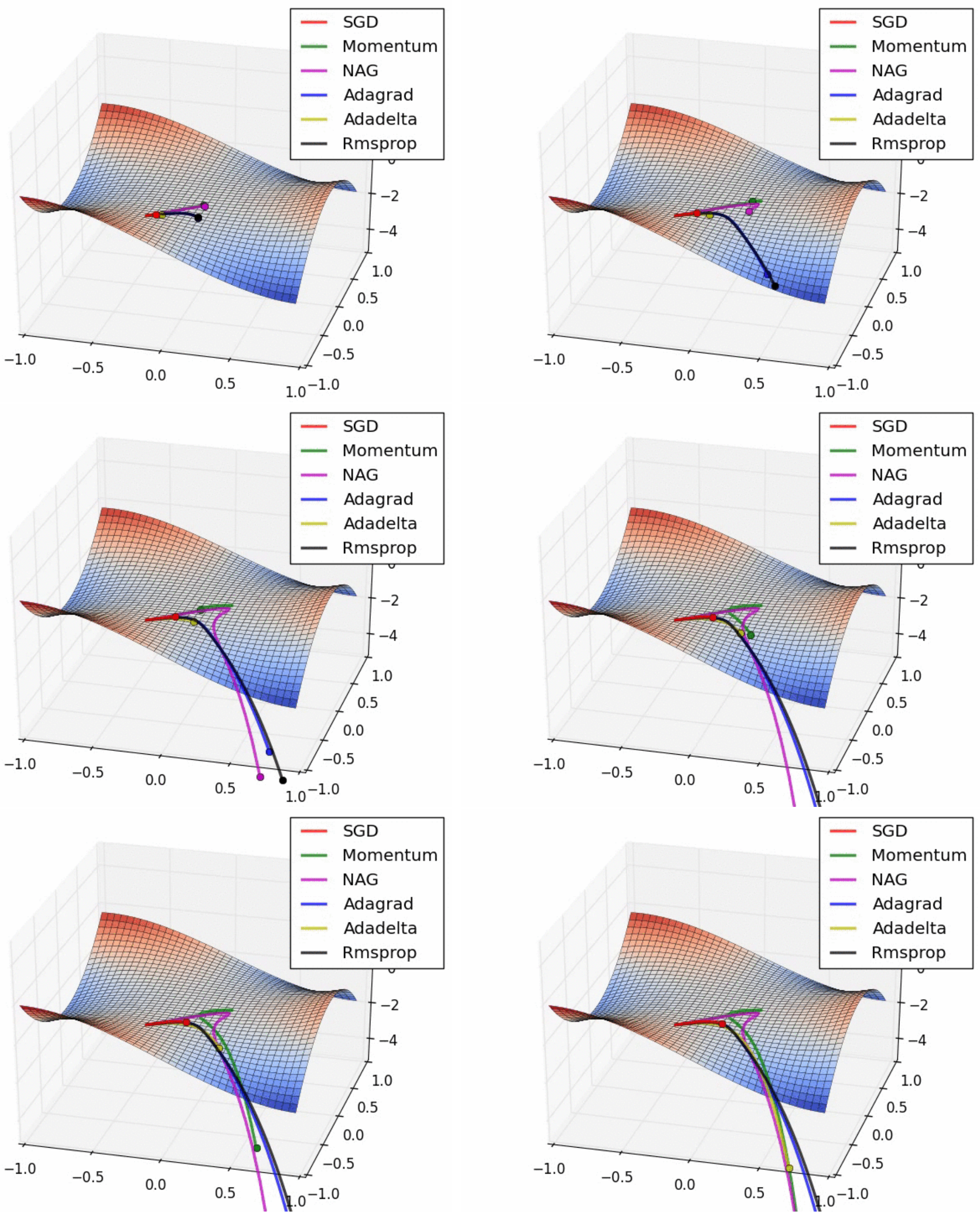

Figure 4.6: Algorithms iterative progress from top left to bottom right frame on a saddle point

Chapter 5

Conclusions

The goal of this thesis is to present a detailed theoretical framework that describes the structure of deep learning. Specifically, a mathematical formulation for the gradient descent optimization method through back propagation using neural networks is given.

Furthermore, we have examined the three variants of the gradient descent method, from which the mini-batch method seems to be the most reliable. In addition, we have looked a number of gradient descent methods. Which, some of them act upon the gradient like momentum and NAG, some of them act upon the learning rate like Adagrad and RMSprop, and some of them act upon both like Adam.

Before drawing any conclusions we have to note an important observation. The gradient descent optimization algorithms are executed with one constant set of parameters for all three networks. An experienced user trying to approach a solution step by step, will definitely be able to tune the parameters depending on the model and the data, and could, thus, achieve better results.

To summarize the most important conclusions, we distinguish between the gradient descent optimization algorithms we introduced in Chapter 3. The conclusions are drawn from the numerical results obtained in Section 4.1. The numerical results indicate that the algorithms perform better in the neural network shown in Figure 4.1, while the number of the parameters is not proportional with the performance of the algorithms. Finally, the adaptive learning rate methods outperform the adaptive gradient ones.

Chapter 6

Appendices

6.1 Appendix 1

Listing 6.1 applies the gradient descent method using a Python code. The f function calculates the value of the function we want to minimize at a given point x. While, *df* computes the derivative of that function. Finally, we acquire the minimum by following the negative slope with the *gradient descent* function .

```
import numpy as np
def f(x):
    return x**4 + 7*x**3 + 5*x**2 -17*x +12
def df(x): # calculate the derivative of f(x)
    return 4*x**3 + 21*x**2 + 10*x -17
def gradient_descent(xi):
        step_size=1
        eta=0.01 # learning rate
        TOL=1e-10 # precision of the algorithm
        n=0
        niter=1e4 #maximum number of iterations
        while step_size > TOL and n<niter:
            xc=xi- eta*df(xi) # gradient descent iteration
            step_size=abs(xi-xc) # calculate the step size
```

```
        xi=xc
        n+=1 #counts the iterations
        print "number of iterations %i" %n
        print "the current value of X is %.3f" %xc
    return f(xc),xc

xi=float(raw_input("initial value of x:"))
minimum,xmin=gradient_descent(xi)
print"The local minimum is","%.3f"%minimum,
print"and occurs at","%.3f"%xmin
```

Listing 6.1: Python example of Gradient Descent

6.2 Appendix 2

Listing 6.2 illustrate a pseudocode that implements a gradient descent algorithm utilizing the back propagation method. Suppose that our training set consists of N training points. First, we start by defining the number of iterations (*niter*). Then, we use back propagation to compute the partial derivatives of the parameters for each data point. Furthermore, we add them together, in order to calculate the mean of every partial derivative over the whole training set. Finally, we update the parameters.

```
For k=1 up till niter
  For i=1 up till N
    x_i = a^[1]
    For l=2 up till L
      z^[l] = W^[l] a^[l-1] + b^[l]
      a^[1] = σ(z^[l])
      D^[l] = diag(σ'(z^[l]))
    δ^[L] = D^[L](a^[L] - y(x_i))
    For l=L-1 down to 2
      δ^[l] = D^[l](W^[l+1])^T δ^[l+1]
    For l=2 up till L
```

$dW^{[l]} = dW^{[l]} + \delta^{[l]}(a^{[l-1]})^T$
$db^{[l]} = db^{[l]} + \delta^{[l]}$
For l=2 up till L
$W^{[l]} = W^{[l]} - \frac{\eta}{N}dW^{[l]}$
$b^{[l]} = b^{[l]} - \frac{\eta}{N}db^{[l]}$

Listing 6.2: Gradient descent pseudocode

6.3 Appendix 3

The following Python code implements the pseudocode in Listing 6.2 on the example shown in Section 2.5. Listing 6.3 includes four functions. *Activate* function evaluate $a^{[L]}$ using the sigmoid function from a forward pass through the network, calculating $a^{[1]}$, $a^{[2]}$ and $a^{[3]}$ in that order. Furthermore, *dactivate* function compute the diagonal matrix $D^{[l]}$ for each layer which allows us to avoid Hadamard product notation. Then, the cost function is called at every iteration to supervise the procedure, by displaying its value on the screen. At last, *netlearning* function contains the gradient descent algorithm. We commence by setting the input data and the target output for each point. Next, the Numpy random library sets an initial value for the parameters in the closed interval $[-2, 2]$, in order to advance through the learning process using the gradient descent method and the back propagation algorithm.

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(5000)
#compute the cost function
def cost(w2,w3,b2,b3):
    x1=np.array([0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7])
    x2=np.array([0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6])
    y=np.array([[1,1,1,1,1,0,0,0,0,0],[0,0,0,0,0,1,1,1,1,1]])
    costvec=np.zeros((10,1))
    for i in range(10):
        x= np.array([[x1[i]],[x2[i]]])
```

```
        a2=activate(x,w2,b2)
        a3=activate(a2,w3,b3)
        costvec[i]=np.linalg.norm((y[:,i].reshape(2,1) -a3),2)
    return (1./10.)*(1./2.)*(np.linalg.norm(costvec,2))**2
#training of the algorithm
def netlearning(epochs):
   #set the data points and the target outputs
   x1=np.array([0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7])
   x2=np.array([0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6])
   y=np.array([[1,1,1,1,1,0,0,0,0,0],[0,0,0,0,0,1,1,1,1,1]])
   #set random initial values for the parameters
   w2=0.5*np.random.uniform(-2,2,(6,2))
   w3=0.5*np.random.uniform(-2,2,(2,6))
   b2=0.5*np.random.uniform(-2,2,(6,1))
   b3=0.5*np.random.uniform(-2,2,(2,1))
   eta=0.05 #set the learning rate value
   #save the value of the cost function for each iteration
   savecost=np.zeros((niter,1))
   n=0 #iteration counter
   while n<epochs:
      # calulate the partial derivatives with back propagation
      w2d=np.zeros((6,2))
      w3d=np.zeros((2,6))
      b2d=np.zeros((6,1))
      b3d=np.zeros((2,1))
      for k in range(10):
        x= np.array([[x1[k]],[x2[k]]])
        a2=activate(x,w2,b2)
        a3=activate(a2,w3,b3)
        delta3=np.dot(dactivate(a3),(a3-y[:,k].reshape(2,1)))
        delta2=np.dot(dactivate(a2),np.dot(w3.T,delta3))
        w2d+=np.dot(delta2,x.T)
        w3d+=np.dot(delta3,a2.T)
        b2d+=delta2
        b3d+=delta3
```

```
            # perform an update of the parameters with gradient descent
            w2=w2-eta*(1./10.)*w2d
            w3=w3-eta*(1./10.)*w3d
            b2=b2-eta*(1./10.)*b2d
            b3=b3-eta*(1./10.)*b3d
            newcost=cost(w2,w3,b2,b3)
            savecost[n]=newcost
            n+=1
            print newcost #display the cost function on the screen
    return savecost
#compute the sigmoid function
def activate(x,w,b):
    z=np.dot(w,x) + b
    s=z.shape[0]
    y=np.zeros((s,1))
    for i in range(s):
     y[i]=1./(1.+np.exp(-z[i]))
    return y
#compute the D matrix
def dactivate(a):
    s=a.shape[0]
    y=np.zeros(s)
    for i in range(s):
        y[i]=a[i]*(1.0-a[i])
    D=np.diag(y)
    return D
epochs=int(1e6)# number of epochs
#plot the cost function - iterations graph
plt.plot(range(epochs),netlearning(epochs))
plt.title('Batch gradient descent')
plt.ylabel('Value of cost function')
plt.xlabel('Iteration Number')
plt.yscale('log')
plt.legend()
```

```python
plt.show()
```

Listing 6.3: Python implementation of network training

Chapter 7

Bibliography

[1] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(Jul):2121–2159, 2011.

[3] Mohammad Havaei, Axel Davy, David Warde-Farley, Antoine Biard, Aaron Courville, Yoshua Bengio, Chris Pal, Pierre-Marc Jodoin, and Hugo Larochelle. Brain tumor segmentation with deep neural networks. *Medical image analysis*, 35:18–31, 2017.

[4] Catherine F. Higham and Desmond J. Higham. Deep learning: An introduction for applied mathematicians, 2018.

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

[6] Junying Li, Deng Cai, and Xiaofei He. Learning graph-level representation for drug discovery. *arXiv preprint arXiv:1709.03741*, 2017.

[7] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/k^ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[8] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[10] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[11] Alec Radford. *Visualizing Optimization Algorithms*, 2014. `https://imgur.com/a/Hqolp`.

[12] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[13] Bharath Ramsundar and Reza Bosagh Zadeh. *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. O'Reilly Media, Inc., 1st edition, 2018.

[14] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[15] George Saon, Gakuto Kurata, Tom Sercu, Kartik Audhkhasi, Samuel Thomas, Dimitrios Dimitriadis, Xiaodong Cui, Bhuvana Ramabhadran, Michael Picheny, Lynn-Li Lim, et al. English conversational telephone speech recognition by humans and machines. *arXiv preprint arXiv:1703.02136*, 2017.

[16] Richard Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.

[17] Hans-Dieter Wehle. Machine learning, deep learning, and ai: What's the difference? In *International Conference on Data scientist innovation day, Bruxelles, Belgium*, 2017.

[18] Nicholas Westlake, Hongping Cai, and Peter Hall. Detecting people in artwork with cnns. In *European Conference on Computer Vision*, pages 825–841. Springer, 2016.

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