Gradient Descent Methods for Training Neural Networks

Simon Slettevold

Spring 2024

Abstract

This thesis explore algorithms to optimize the training process of a neural network for deep learning purposes, where the focus will be on gradient descent algorithms such as SGD ("Stochastic Gradient Descent") and ADAM ("Adaptive Moment Estimation"). The fundamental theory regarding neural networks and its mathematics will be thoroughly explained, and also put into practise through practical examples. Using these examples, a comparison will be made between different gradient descent schemes and algorithms. Through these comparisons, the ADAM algorithm shows most favorable.

Acknowledgements

To start, I would like to express my gratitude for my supervisor, Tore Selland Kleppe, for guiding me throughout this project. I am grateful for his approach to make this work my own, whilst still providing with most helpful feedback and advice.

Contents

1 Introduction **4**

2 Theory **5**
2.1 Gradient Descent 5
2.1.1 Learning rate 5
2.2 Building a Neural Network 8
2.2.1 The Perceptron 8
2.2.2 Single Layer Neural Network 12
2.2.3 Multi Layer Neural Network 15
2.3 Training a Neural Network 17
2.3.1 Loss Function 17
2.3.2 Backpropagation 18

3 Simple Application of Deep Learning **26**
3.1 Classification Problem: MNIST Database 26
3.2 Regression Problem: Wine Quality 30

4 Optimization methods for Neural Networks **33**
4.1 Gradient Descent Methods 33
4.1.1 Stochastic Gradient Descent and Mini-batching 33
4.1.2 ADAM 34
4.2 Comparison 36
4.2.1 Various Batch-size for Classification Problem 37
4.2.2 Various Batch Sizes for Regression Problem 38
4.2.3 Discussion 38

5 Conclusion **40**

1 Introduction

Throughout history, scientists have been fascinated by the human brain. With assistance from our organs, it is able to convert every input into comprehensible outputs that we can perceive. What might be even more fascinating, is that the brain is able to remember all these forms of input data and can *learn* from them based on what we have experienced. For instance, I assume most people could correctly distinguish a dog from a cat, based on their own perceptions and experiences.

Over the last decades, with the rise of computer science, the field of ”deep learning” have been formed. Deep learning is the study of an ”artificial neural network” or a ”neural network”, that have been developed to imitate the intrications of the neurons in the human brain. The motivation is for a computer or a machine to be able to learn and recognize patterns similarly to what the human brain does. Along with the advancements in computational power, machines are capable of handling even larger models of these neural networks. With the added complexity, the necessity for methods to optimize these computations is found to be critical. In models consisting of millions or billions parameters and input data, small improvements in efficiency can result in massive changes in the models performance. Today, digital tools such as ChatGPT, Google Translate or Grammarly are neural network models that are trained on a massive amount of data. These models then heavily rely on being trained as efficiently as possible, while also being accurate.

For this thesis, I will establish the fundamental mathematics relating to how neural networks are built and how they can be trained. When the theory have been settled, I intend on applying the mathematics into practise and train some simple neural network models. Furthermore, these models will make the groundwork for the discussion of alternative training algorithms that can optimize the training process.

2 Theory

2.1 Gradient Descent

Gradient Descent is an optimization method that is used to find a local minimum of a differentiable function. The algorithm operates in an iterative manner, meaning it ”step-by-step” converges towards the optimal solution. The idea of the algorithm is to pick a starting point, x_0, then take a ”step” in the direction where the function value decreases the fastest. The gradient of the function point in the direction of the fastest increase, so the direction of the fastest decrease would be in the opposite direction of the gradient. To find the next step, we use the following formula [3]:

$$\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \lambda_i \nabla f(\boldsymbol{x}^{(i)})$$

Here λ_i is the *learning rate*, and $\boldsymbol{x}_i$ is the input vector at the i^{th}-step for our function. The learning rate could be set constant for all iterations, but one can also utilize an adaptive learning rate for optimizing the algorithm even further.

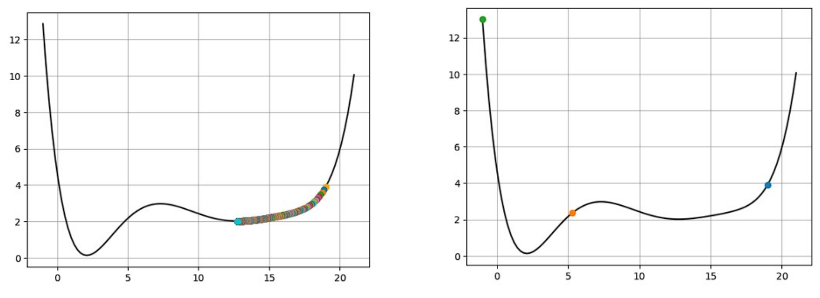
The full algorithm can be expressed as (inspired by [3]):

Algorithm 1 Gradient Descent

```
Require: $\boldsymbol{x}^{(0)}$: initial trial solution
  while $\boldsymbol{x}^{(i)}$ not converged do
      Pick a learning rate $\lambda_i$
      Calculate $\nabla f(\boldsymbol{x}^{(i)})$
      Calculate $\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \lambda_i \nabla f(\boldsymbol{x}^{(i)})$
  end while
```

2.1.1 Learning rate

There are several possible strategies for choosing the learning rate. The simplest is to pick a constant learning rate for all the iteration steps. This may lead to problems, if we choose a too large or too small learning rate.



(a) Small learning rate ($\lambda = 0.1$) (b) Large learning rate ($\lambda = 10$)

Figure 1: Gradient descent process with two different learning rates. The function being optimized is: $f(x) = 0.000017x^6 - 0.001133x^5 + 0.029241x^4 - 0.36691x^3 + 2.234115x^2 - 5.510020x + 4.722197$

In Figure 1, we see that when we apply a too small learning rate, we are able to find a local minimum. Even though we successfully found a local minimum, it takes a lot of steps to do so. We would prefer to use as few iterations as possible; because for larger and more complex functions, we would like for the algorithm to converge as fast as possible to save computation time. In the case of applying a too large learning rate, we notice that it diverges drastically after only two iterations.

Our goal is to find a learning rate that enables us to successfully converge towards a local minimum with as few iterations as possible. This could be achieved in two ways. The first approach involves experimenting with different constant learning rates, and see which learning rates work. However, this method can be quite inefficient as it requires trial and error, and we would like to have a more intelligent way to select our learning rate.

By introducing an *adaptive learning rate*, the learning rate adjusts itself for each iteration to find the most optimal value for that step. There are several algorithms which applies this concept, and we will look more into some of these later. For now, I will just demonstrate a relatively straightforward way to apply adaptive learning rate when implementing the gradient descent algorithm [6]:

Algorithm 2 Adaptive Gradient Descent

```
Require: $\boldsymbol{x}^{(0)}$: initial trial solution
  while $\boldsymbol{x}^{(i)}$ not converged do
      Calculate $\nabla f(\boldsymbol{x}^{(i)})$
      Calculate $\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \lambda_i \nabla f(\boldsymbol{x}^{(i)})$
      Insert the iteration step into the initial function $f(\boldsymbol{x}^{(i+1)})$ to obtain $f(\lambda)$
      Solve $f(\lambda) = \min_{\lambda \geq 0} f(\lambda)$, to obtain the optimal step-size, $\lambda_i^*$
      Calculate $\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \lambda_i^* \nabla f(\boldsymbol{x}^{(i)})$
  end while
```

Example 1. *This example shows an analytic way of adaptive gradient descent. Consider the function of two independent variables:*

$$f(x_1, x_2) = x_1^2 + 2x_2^2 - 2x_1x_2 - 2x_2$$

We can then calculate the gradient of the function $\nabla f(\boldsymbol{x})$:

$$\nabla f(\boldsymbol{x}) = (\tfrac{\partial f}{\partial x_1}, \tfrac{\partial f}{\partial x_2})$$

$$\begin{cases} \frac{\partial f}{\partial x_1} = 2x_1 - 2x_2 \\ \frac{\partial f}{\partial x_2} = 4x_2 - 2x_1 - 2 \end{cases}$$

$$\nabla f(\boldsymbol{x}) = (2x_1 - 2x_2, 4x_2 - 2x_1 - 2)$$

To begin the procedure, we need a starting point $\boldsymbol{x}_0$. We stand freely to choose any starting point. I'll choose $\boldsymbol{x}^{(0)} = (0, 0)$. Calculating the gradient at this point gives:

$$\nabla f(0, 0) = (0, -2)$$

Now we can calculate the first iteration by setting:

$$\boldsymbol{x}^{(1)} = \begin{cases} x_1 = 0 - \lambda(0) = 0 \\ x_2 = 0 - \lambda(-2) = 2\lambda \end{cases}$$

and submitting these expressions into $f(\boldsymbol{x})$ to obtain:

$$\begin{aligned} f(\boldsymbol{x}^{(1)}) &= f(0, 2\lambda) \\ &= (0)^2 + 2(2\lambda)^2 - 2(0)(-2) - 2(2\lambda) \\ &= 8\lambda^2 - 4\lambda \end{aligned}$$

We're interested in which λ gives us the smallest value for $f(0, -2\lambda)$.

$$f(0, 2\lambda^*) = \min_{\lambda \geq 0} f(0, 2\lambda)$$

From:

$$\tfrac{d}{d\lambda}(8\lambda^2 - 4\lambda) = 16t - 4 = 0$$

It follows that:

$$\lambda_1^* = 1/4$$

Piecing it all together, we get after the first iteration:

$$\boldsymbol{x}^{(1)} = (0, 2\lambda_1^*) = (0, 2(\tfrac{1}{4})) = (0, \tfrac{1}{2})$$

Continuing with more iterations, we converge towards the result $\boldsymbol{x}^* = (1, 1)$.

(This example is based on a similar exercise from [6])

2.2 Building a Neural Network

A "neural network" is a computational model inspired by the intricate workings of the biological neurons in our brain. The idea behind a neural network is to replicate and simulate the behaviour of the biological neurons, so that our network structure can process and "learn" from a large enough dataset. The necessity of recognizing patterns is something that multiple fields of science encounter. Neural networks have shown to be extremely useful tools in solving complex problems such as image recognition, medical diagnosis and visualization for autonomous vehicles [5] [11].

In this section, we will discuss how we can build a neural network. We shall begin from the theory of a single neuron, a gradually build up to a full network consisting of multiple neurons connected in layers. (Formulas and notation are mostly inspired by [2] [5] [11])

2.2.1 The Perceptron

In the realm of deep learning, the "building blocks" of a neural network are called *perceptrons.* A perceptron is a single neuron cell, and a full neural network could consist of thousands, millions or even billions of these [2] [11].

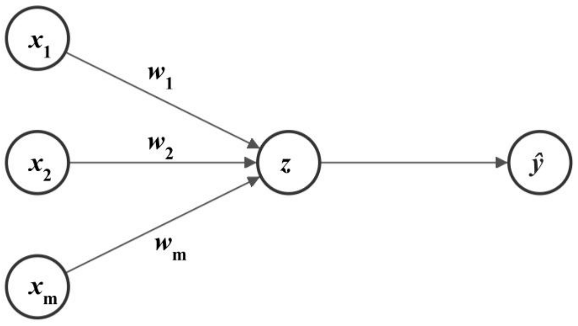


Figure 2: The Perceptron

Figure 2 shows a simple example of a perceptron. It consists of multiple inputs, $x_1, x_2, ..., x_m$, where the inputs have their corresponding weights, $w_1, w_2, ..., w_m$, an activation step, z, and a single output, $\hat{y}$. The way the perceptron combines all these inputs and weights to produce an output is happening in the activation step. There, two things happens:

1. **Summation**: It multiplies and sums up all the weights and inputs, $\sum_{j=1}^{m} x_j w_j$.
2. **Apply a non-linearity**: Apply a non-linear activation function.

We can express this as following:

$$\hat{y} = g\left(\sum_{j=1}^{m} x_j w_j\right) \tag{1}$$

Where g is our non-linear activation function. We can write this in term of vectors:

$$\hat{y} = g\left(\boldsymbol{x}^T \boldsymbol{w}\right) \tag{2}$$

Where $\boldsymbol{x} = (x_1, ..., x_m)$ and $\boldsymbol{w} = (w_1, ..., w_m)$.

Activation Function
The main purpose of using an activation function in the perceptron is to introduce non-linearities into the network. If we were to use a linear activation function (or not any activation function at all), the neural network would struggle to handle non-linear data. In the practical world, a majority of the data we obtain are non-linear, meaning it is important for our neural network to be able to handle that [2] [5].

There are several options in choosing a non-linear activation function. It depends on your data, which functions would be the most beneficial. One of the most popular activation functions, is the *sigmoid function.* (Formulas and notation similar to [3] [5]).

$$g(z) = \frac{1}{1 + e^{-az}} \tag{3}$$

$$g'(z) = a\, g(z)\, (1 - g(z))$$

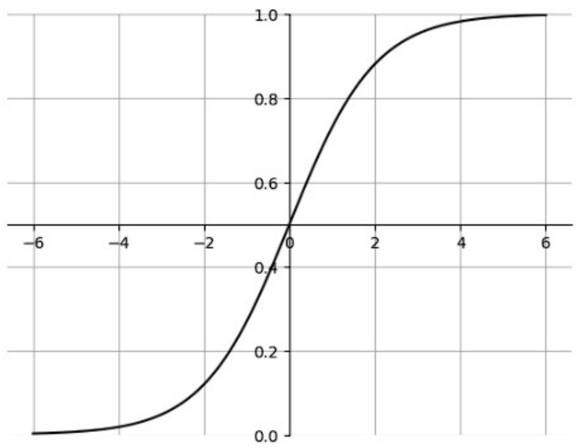


Figure 3: The Sigmoid Function

The sigmoid function will for any input give values between 0 and 1. This is useful when we for example are interested in probabilities or a scale-based result with values from 0 to 1. Another property that is desirable is that the function is continuously differentiable for the entire input space. This enables the use of gradient-based optimization methods. A major disadvantage of the sigmoid function, is the "vanishing gradient problem". The problem is encountered during the training of the neural network, where the gradients of the loss function becomes really small. This leads to little updating of the weights and that the network learns slower [4].

In modern neural networks, the *ReLU* (*"rectified linear unit"*) activation function is the preferred option.

$$g(z) = max(0, z) = \begin{cases} z & if\ z > 0 \\ 0 & otherwise \end{cases}$$

$$g'(z) = \begin{cases} 1 & if\ z > 0 \\ 0 & otherwise \end{cases}$$

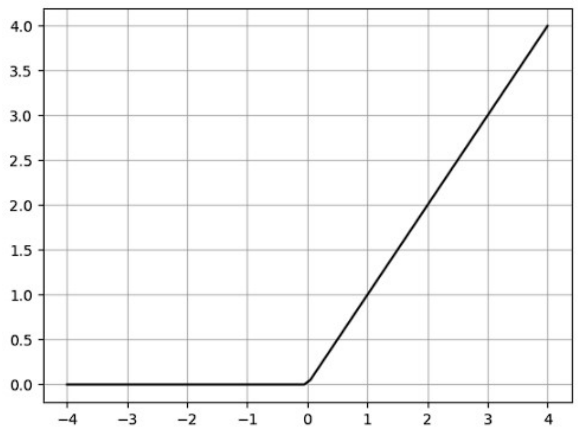


Figure 4: The ReLU Function

A ReLU activation function is very computationally efficient, since its computation of the gradient is always the same. ReLU avoids the vanishing gradient problem due to its' gradient being constant. There are still some disadvantages, such as it is not differentiable at $z = 0$, which can lead to problems. A common quick solution could be to arbitrarily choose the derivative to be either 0 or 1 at $z = 0$. The biggest concern of using ReLU is what is called the "dying ReLU problem". ReLU neurons can sometimes be pushed into states where they will always return 0 on any input value. This condition is known as the "dead state" of the ReLU neurons, and it is difficult to recover because the gradient of 0 is also 0. Despite these imperfections, it is still more preferable than the sigmoid function in modern neural networks. [4].

To compare and summarize the two activation function discussed, ReLU is the more commonly used activation function for large, modern neural networks. This is mostly due to the fact that it is more computationally effective and that it overcomes the major difficulties of the sigmoid function. The sigmoid function is still usable, but is more suited for smaller and less complex networks.

Bias

For now, the expressions formulated are not entirely accurate. To obtain a greater accuracy we add an extra node to our perceptron, which is called the *bias*.

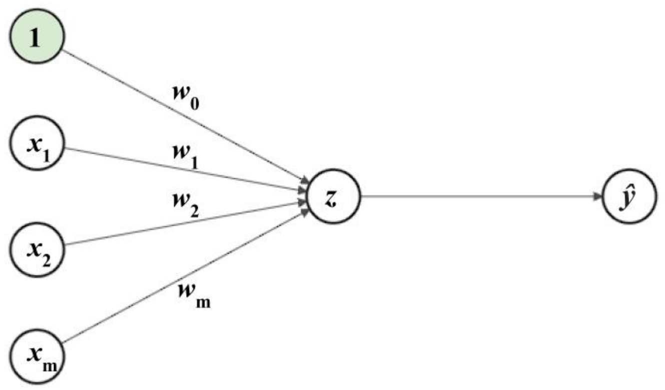


Figure 5: Including the bias in the perceptron from Figure 1

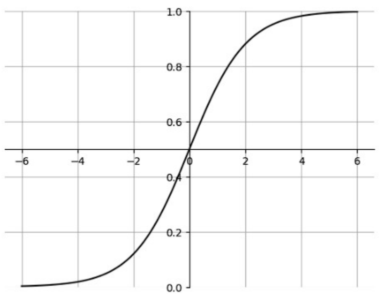
The bias is a scalar added to the weights and inputs to help transform the activation function the way we want. With the addition of the bias, we are able to shift the activation function towards the positive or negative side. With the addition of the bias term, we rewrite Equation 1 and 2 as:

$$\hat{y} = g\left(\sum_{j=0}^{m} x_j w_j\right) = g\left(w_0 + \sum_{j=1}^{m} x_j w_j\right)$$

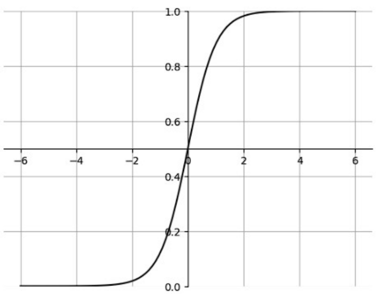
$$\hat{y} = g\left(w_0 + \boldsymbol{x}^T \boldsymbol{w}\right)$$

To show the effects of the bias term, I have plotted various plots of the sigmoid function below, with only one input node. Using Equation 3, this takes the form:

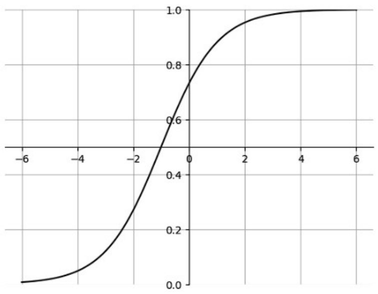
$$g(x) = \frac{1}{1 + e^{-(w_0 + x \cdot w_1)}}$$



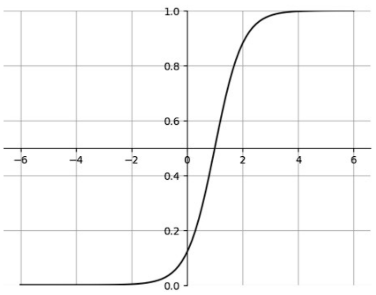
(a) $w_1 = 1$, $w_0 = 0$



(b) $w_1 = 2$, $w_0 = 0$



(c) $w_1 = 1$, $w_0 = 1$



(d) $w_1 = 2$, $w_0 = -2$

In these plots we can identify the effects of the addition of the bias term. We see in plot (c) and (d) that the function shifts towards the positive and the negative side. When we only change the weight, as shown in plot (b), we only change the slope of the function. With the bias added, we gain another parameter that provides the function with a lot more flexibility. This flexibility gives us a lot of power over how the activation step behaves and process the inputs.

2.2.2 Single Layer Neural Network

In the previous section, we got an understanding of what constructs the neural network. Now, we are going to start building a network which consists of multiple of these perceptrons. (Formulas and notation similar to [2] [11]).

For further calculations, we will use the substitution:

$$z = w_0 + \sum_{j=0}^{m} x_j w_j$$

We can think of z as the "state" of the neuron. With this substitution we can write the output as:

$$\hat{y} = g(z)$$

Moving forward, we can start by looking at the *multi output perceptron*. It is similar to the normal perceptron, but can have multiple outputs. For the multi output perceptron, each perceptron is associated with its own output, and uses all of the inputs. In the figure below there is an example of a multi output perceptron with two activation nodes [2].

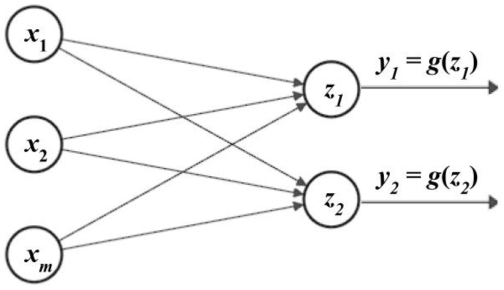


Figure 6: Multi output perceptron

For the notation, we introduce another subscript, j, to denote which perceptron it belongs to:

$$z_j = w_{j,0} + \sum_{k=0}^{m} x_k w_{j,k}$$

Now, we can finally begin creating the *Single Layer Neural Network*. What defines a single layer neural network is that it has only one *hidden layer* of perceptrons between the inputs and the outputs. By increasing the number of perceptrons we increase the number of connections between all the nodes, and thereby the network's complexity. This gives the network a larger amount of weights we can train and adjust to achieve the desired output [5].

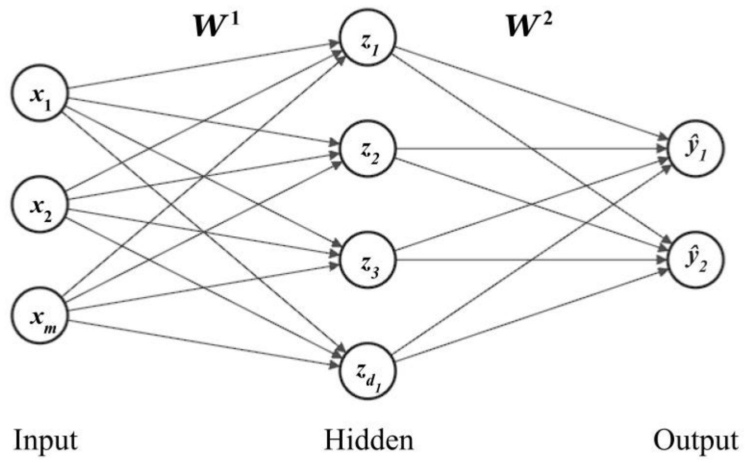


Figure 7: Single Layer Neural Network

By using the same notation for z_j as for the multi output perceptron:

$$z_j = w_{j,0}^1 + \sum_{k=0}^{m} x_k w_{j,k}^1 \tag{4}$$

where the superscript "1" indicates which weight matrix we are gathering our weights from. We can now write for the final output layer:

$$\hat{y}_j = g\left(w_{j,0}^2 + \sum_{k=0}^{d_1} g(z_k) w_{j,k}^2\right) \tag{5}$$

Note that the superscript "2" in Equation 5 refers to the layer we want to calculate, and not as an exponent. To get an intuition on what is going on in these formulas, I will demonstrate in Example 2 how one easily can do these calculations.

Example 2. *Let us look at z_2 from the single layer neural network in Figure 7.*

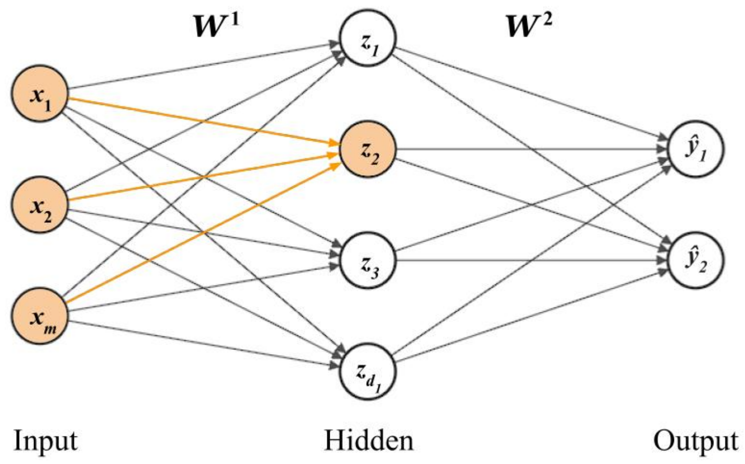


Figure 8: Calculating z_2

We can calculate it using Equation 4.

$$\begin{aligned} z_2 &= w^1_{2,0} + \sum_{k=1}^{m} x_k w^1_{2,k} \\ &= w^1_{2,0} + x_1 w^1_{2,1} + x_2 w^1_{2,2} + ... + x_m w^1_{2,m} \end{aligned}$$

(Example from [2])

2.2.3 Multi Layer Neural Network

Now that we have established an understanding of how to construct a neural network with only a single hidden layer, we can continue adding more such layers. When we have a neural network consisting of multiple hidden layers, we call it a ”Multi Layer Neural Network”.

The main reason for implementing multiple hidden layers to a neural network, is added complexity. With added layers, comes added applications of non-linearities. By having more non-linearities applied, one can find better approximations for more complex data [1].

To make the calculation more clear, I start of by defining some variables (formulas and notation similar to [11]):

- $z_j^r \rightarrow$ element j of the neurons in the r^{th} layer
- $\boldsymbol{z}^r \rightarrow$ output vector of the r^{th} layer (prior to the application of a non-linearity)
- $\boldsymbol{y}^r \rightarrow$ output vector of the r^{th} layer
- $\boldsymbol{y}^L := \hat{\boldsymbol{y}} \rightarrow$ final output vector
- ${\boldsymbol{w}_j^r}^T \rightarrow$ weight vector related to the j^{th} neuron in the r^{th} layer, z_j^r,
  $\boldsymbol{w}_j^r = \left[w_{j,0}^r, w_{j,1}^r, w_{j,2}^r, ..., w_{j,k_{r-1}}^r\right]^T$
- ${\boldsymbol{W}^r}^T \rightarrow$ weight matrix related to the r^{th} layer, $\boldsymbol{W}^r = [\boldsymbol{w}_1^r, \boldsymbol{w}_2^r, ..., \boldsymbol{w}_{kr}^r]^T$

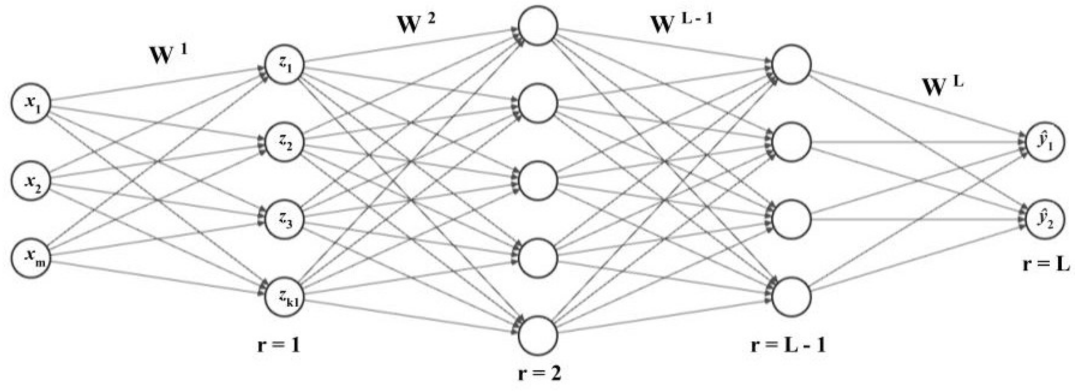


Figure 9: Multi Layer Neural Network

We can define the inner product calculation, z_j^r, at the j^{th} neuron of the r^{th}, where $j = 1, 2, ..., k_r$ and $r = 1, 2, ..., L$, as:

$$z_j^r = {\boldsymbol{w}_j^r}^T \boldsymbol{y}^{r-1} \tag{6}$$

where:

$${\boldsymbol{w}_j^r}^T \boldsymbol{y}^{r-1} = w_{j,0}^r + \sum_{k=0}^{k_{r-1}} y_k^{r-1} w_{j,k}^r = w_{j,0}^r + \sum_{k=0}^{k_{r-1}} g(z_k^{r-1}) w_{j,k}^r \tag{7}$$

Note that the dimension of $\boldsymbol{w}_j^{r^T}$ is $k_{r-1}+1$, where k_{r-1} corresponds to the number of neurons in the previous layer $(r-1)$ and the 1 accounts for the bias. We can collect all these inner product values into a vector:

$$\boldsymbol{z}^r = \left[z_1^r, z_2^r, ..., z_{k_r}^r\right]^T$$

Alternatively, we can get the output vector by computing:

$$\boldsymbol{z}^r = \boldsymbol{W}^{r^T} \boldsymbol{y}^{r-1} \tag{8}$$

Finally, to get the final output vector of the r^{th} layer, we apply an activation function, g, on each components of $\boldsymbol{z}^r$:

$$\boldsymbol{y}^r = \begin{bmatrix} 1 \\ g(\boldsymbol{z}^r) \end{bmatrix} \tag{9}$$

where the 1 represents the bias term, and the notation $g(\boldsymbol{z}^r)$ means that g acts individually on each one of the components of $\boldsymbol{z}^r$. For $\hat{y}_j$, we can express it as:

$$\hat{\boldsymbol{y}} = g(\boldsymbol{z}^L)$$

where (from Equation 6 and 7):

$$\begin{aligned} \hat{y}_j &= g\left(z_j^L\right) \\ &= g\left(w_{j,0}^L + \sum_{k=0}^{k_{L-1}} y^{L-1} w_{j,k}^L\right) \\ &= g\left(w_{j,0}^L + \sum_{k=0}^{k_{L-1}} g(z_k^{L-1}) w_{j,k}^L\right) \end{aligned}$$

2.3 Training a Neural Network

We have explored the construction of neural networks, where multiple ”neurons” are interconnected in layers. Our aim is for the network to be able to recognize patterns, similar to how the human brain operates. This is achieved through *training* our neural network with a large amount of data, so that hopefully over time the network will pick up a pattern.

2.3.1 Loss Function

In order to effectively train and enhance our network, we require a type of measurement on its performance. With such a measurement, we can apply optimization methods to identify an optimal set of weights and biases for our network. We can define a *loss function*, a function that evaluates the disparity between the predicted output (generated by our network) and the desired target value. Typically, the loss function is defined as: (Formulas are similar to [2] and [11])

$$\mathcal{L}\left(f(\boldsymbol{x}_n; \boldsymbol{W}), \boldsymbol{y}_n\right)$$

where $f(\boldsymbol{x}_n;\boldsymbol{W})$ is the predicted output and $\boldsymbol{y}_n$ is the actual value(s). Because we are interested in the error for a full data set and not just one single data point, we sum up and take the average:

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(f(\boldsymbol{x}_n;\boldsymbol{W}),\boldsymbol{y}_n\right)$$

There are different options of loss function one can choose. For cases where you have a classification model with only two possible classes, a common loss function to use is the "cross entropy loss" function.

$$J(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n}\boldsymbol{y}_n log\left(f(\boldsymbol{x}_n;\boldsymbol{W})\right) + (1-\boldsymbol{y}_n)log\left(1-f(\boldsymbol{x}_n;\boldsymbol{W})\right)$$

For other cases, like a regression model, a commonly used loss function is the "mean squared error" loss function:

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{y}_n - f(\boldsymbol{x}_n;\boldsymbol{W})\right)^2$$

A loss function tells us how well the network is able to predict the final output. With high accuracy, the loss function would be a small number, and for low accuracy, the loss function would be a higher number. Our objective for training the neural network is then to find the network that achieves the minimal loss.

$$\boldsymbol{W^*} = \min_{\boldsymbol{W}} J(\boldsymbol{W})$$

Remember that $\boldsymbol{W} = \{\boldsymbol{W^{(1)}}, \boldsymbol{W^{(2)}}, ...\}$, which is a list of all the weight matrices of our neural network. The loss function, $J(\boldsymbol{W})$, will then be of the same dimension as the number of weights in our neural network. To minimize the loss, we use the method of *backpropagation.*

2.3.2 Backpropagation

So far, we have discussed neural networks that "feed forward" all the information, which is passing the input values through the network until we have an output. The idea of backpropagation is to reverse this process. When training our network, we initially feed forward the information, then propagate the data backwards again to pinpoint where to make necessary adjustments to enhance the network's accuracy. By leveraging the loss function as a metric to determine the network's performance, we can better our neural network. Because we aim to minimize our loss and optimize the network, methods like gradient descent can be utilized to guide us towards a minimum of our loss function (see Algorithm 1).

Gradient Descent
Because we are interested to find a minimum to a multidimensional function, we can apply the gradient descent algorithm.

$$\nabla J(\boldsymbol{W}) = \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w}_1^1} \\ \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w}_2^1} \\ \cdots \\ \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w}_k^r} \end{bmatrix}$$

Here, the gradient is a vector containing the derivatives of the loss function with respect to all the weights and biases. This vector encapsulates the changes to the weights required to minimize the loss. To illustrate what is happening, we can imagine processes as shown in Figure 1 (note that the figure just shows a function of one independent variable, and is a very simplified illustration) where the weights are trying to reach a local minima. The gradient descent scheme is found to be extremely useful for optimizing the weights and biases for a neural network. The way we apply the gradient descent algorithm, is through the method of backpropagation.

We can rewrite Algorithm 1 in terms of the loss function and the weights:

Algorithm 3 Gradient Descent for Neural Networks

```
Require: $\boldsymbol{W}^{(0)}$: initial trial solution (chosen randomly)
  while $\boldsymbol{W}$ not converged do
    Compute the gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}\Big|_{W=w_m}$
    Update the weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \lambda_i \nabla J(\boldsymbol{W})$
  end while
```

Backpropagation: Establishing the algorithm

Let us establish the backpropagation method. Firstly, let us define $(\boldsymbol{y}_n, \boldsymbol{x}_n), n = 1, 2, ..., N$ to be the set of training samples. To do this generally, assume we have a multi layer neural network with multiple input and output variables. Note that the formulas and notation will follow similarly as discussed in Section 2.2.3. Thus, the network consists of L layers, $L-1$ hidden layers, one output layer, and that each layer have $k_r, r = 1, 2, ..., L$ neurons. From the training samples, we can write the output vectors as:

$$\boldsymbol{y}_n = [y_{n1}, y_{n2}, ..., y_{nk_L}]^T, \qquad n = 1, 2, ..., N \tag{10}$$

We can then address the backpropagation method as a process of two computational steps [11]:

Algorithm 4 Backpropagation Method

1. *Forward Computations*: We compute a forward pass of the input vector, $\boldsymbol{x}_n,\ \ n = 1, 2, ..., N$, using the current weights and biases, to get all the outputs of all the neurons in all layers, y^r_{nj}.
2. *Backward Computations*: With the computed outputs and neurons, we compute the gradient of the loss function. This involves L steps (corresponding to the number of layers). The process of computing the gradient is as following:
   - Compute the derivatives of the loss function with respect to the parameters of the neurons of the last layer, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w}^L_j},\ \ j = 1, 2, ..., k_L$.
   - **for** $r = L-1$ to 1 **do**:
     - Compute the derivatives of the r^{th} layer, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w}^r_k},\ \ k = 1, 2, ..., k_r$, from the derivatives of layer $r+1$, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{w}^{r+1}_j},\ \ k = 1, 2, ..., k_{r+1}$.

     end for

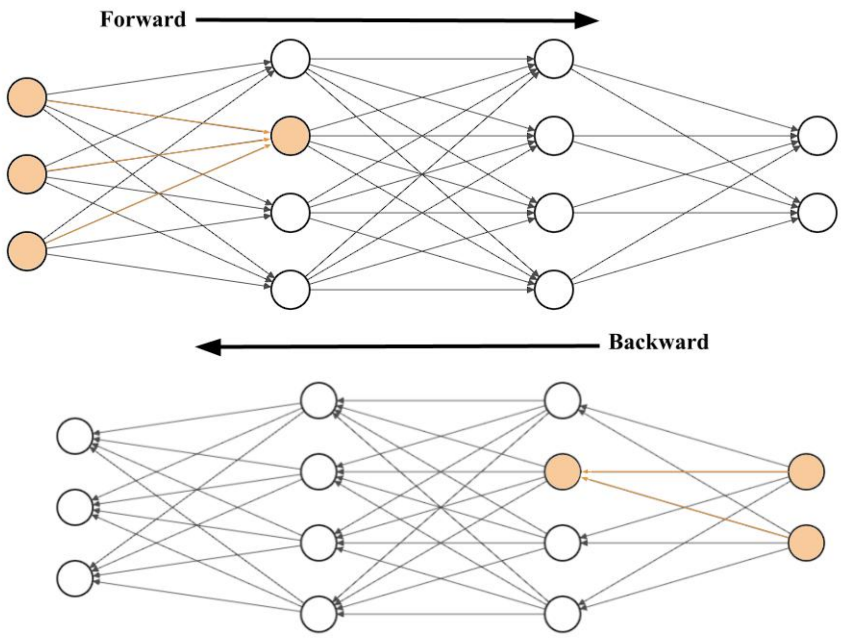


Figure 10: Forward and Backward Pass

From Equation 8 we know that:

$$y_k^r = g\left(\boldsymbol{w}_k^{r^T}\boldsymbol{y}^{r-1}\right), \qquad k = 1, 2, ..., k_r$$

and by Equation 9 that the output of the j^{th} neuron of the next layer becomes:

$$y_j^{r+1} = g\left(\boldsymbol{w}_k^{r+1^T}\boldsymbol{y}^r\right) = g\left(\boldsymbol{w}_k^{r+1^T}\left[\begin{matrix}1\\ g\left(\boldsymbol{W}^{r^T}\boldsymbol{y}^{r-1}\right)\end{matrix}\right]\right)$$

We notice that there arises a "function of a function"-structure where the next layer can be calculated from the previous layer, knowing all the weights and biases. This holds throughout the whole network. This structure provides complications for computing the gradients, as it is a highly non-linear operation. The idea of computing the gradient backwards in the network, comes from the simplicity of computing the derivatives of the output layer:

$$\hat{\boldsymbol{y}} = \boldsymbol{y}^L = g\left(\boldsymbol{W}^{L^T}\boldsymbol{y}^{L-1}\right)$$

where it can be found directly by knowing $\boldsymbol{y}^{L-1}$, which we do from the forward pass. By knowing the derivatives of the last layer, we can continue back in the network to find all the rest of the derivatives in the gradient.

Let us now expand Algorithm 4, and formulate it more mathematically. This derivation will follow similarly from the derivation given in [11].

For this derivation, I will use the squared error loss function:

$$J(\boldsymbol{W}) = \frac{1}{N}\sum_{n=1}^{N} J_n(\boldsymbol{W}) = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{k_L}\sum_{k=1}^{k_L}(\hat{y}_{nk} - y_{nk})^2 = \frac{C}{2}\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{k_L}(\hat{y}_{nk} - y_{nk})^2$$

where $C = 2\frac{1}{k_L}$ and $\hat{y}_{nk},\ k = 1, 2, ..., k_L$, are the estimated values of the output layer, $\hat{\boldsymbol{y}}_n$. Now, let z_{nj}^r denote the result of the linear product of the j^{th} neuron in the r^{th} layer when $\boldsymbol{x}_n$ is applied as the input. We get:

$$z_{nj}^r = w_{j,0}^r + \sum_{m=1}^{k_r - 1} w_{jm}^r y_{nm}^{r-1} = \sum_{m=0}^{k_r - 1} w_{jm}^r y_{nm}^{r-1} = \boldsymbol{w}_j^{r^T} \boldsymbol{y}_n^{r-1}$$

where $\boldsymbol{y}_n^{r-1}$ is defined similarly as in Equation 10:

$$\boldsymbol{y}_n^{r-1} = [1, y_{n1}^{r-1}, y_{n2}^{r-1}, ..., y_{nk_{r-1}}^{r-1}]$$

where we have to include $y_{n0}^{r-1} = 1$ for the bias term. Further, we can apply the chain rule for computing the gradients:

$$\frac{\partial J_n}{\partial w_j^r} = \frac{\partial J_n}{\partial z_{nj}^r}\frac{\partial z_{nj}^r}{\partial w_j^r} = \frac{\partial J_n}{\partial z_{nj}^r}\boldsymbol{y}_n^{r-1}$$

We see that the gradients only depend on the derivatives $\frac{\partial J_n}{\partial z_{nj}^r}$ and the output values of the previous layer (which we know from the forward pass). Therefore, the work of computing the gradients lies within calculating these derivatives. Commonly they are redefined as:

$$\delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r} \tag{11}$$

Finally, we can write the elements of the gradient as:

$$\frac{\partial J(\boldsymbol{W})}{\partial w_j^r} = \sum_{n=1}^{N} \delta_{nj}^r \boldsymbol{y}_n^{r-1} \tag{12}$$

where $r = 1, 2, ..., L$ denote the layer, and $n = 1, 2, ..., N$ refers to a respective training sample, n.

Backpropagation: Finding δ
Up to this point, we have yet to discuss the backwards calculations of the algorithm for training the neural network. In preparation of the derivation, I would like to summarize all the known variables we need for the calculations. These are either known by definition or from the forward pass calculations.

- $J_n = \frac{1}{k_L}\sum_{k=1}^{k_L}(\hat{y}_{nk} - y_{nk})^2 = \frac{C}{2}\sum_{k=1}^{k_L}\left(g(z_{nk}^L) - y_{nk}\right)^2$ (known by definition)
- (Assuming we use the sigmoid activation function)
  $g(z) = \frac{1}{1+e^{-az}}$, $\quad g'(z) = a\, g(z)\,(1 - g(z))$ (known by definition)
- $\delta_{nj}^L = \frac{\partial J_n}{\partial z_{nj}^L} = C(\hat{y}_{nj} - y_{nj})g'(z_{nj}^L)$ (known from forward pass)

- Note that we know all z_{nj}^r values from the forward pass

With these known variables, we want to derive an expression for δ_{nj}^{L-1} and so on for all r. We can do this by generalising for δ_{nj}^{r-1}. As we are familiar with by now, due to the "function-over-function" structure between the layers and the neurons, we can utilize the chain rule for differentiation to find δ_{nj}^{r-1} (derivations from [11]):

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^r} \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^r \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} \tag{13}$$

For $\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}$, we can write it as:

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} w_{km}^r y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}} \tag{14}$$

where

$$y_{nm}^{r-1} = g(z_{nm}^{r-1}) \tag{15}$$

By combining Equation 14 and Equation 15, we get that:

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = w_{kj}^r g'(z_{nj}^{r-1}) \tag{16}$$

Now, from entering Equation 16 into Equation 13, we get the final expression:

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^r w_{kj}^r \right) g'(z_{nj}^{r-1}), \qquad j = 1, 2, ..., k_{r-1}$$

To put this result into words, by knowing δ_{nj}^{r-1} we can find all the elements of the gradient, $\nabla J(\boldsymbol{W})$, from Equation 12. We notice here that we can calculate the derivatives of the early layers from the later layers. This is why we call it backpropagation, because we do a forward pass of the input data and then go backwards again to calculate all the changes to the weights we would like.

Example 3. ***Simple Example of Backpropagation***
Backpropagation can be complicated, so to get a quick insight we can look at a simple example. Let us look at the simplest neural network we could make; a network containing one input, one neuron, and one output (example inspired from [2]). Note that this example is a very simple case of a neural network with very few parameters, and is just to get an easy insight into the formulas discussed previously.

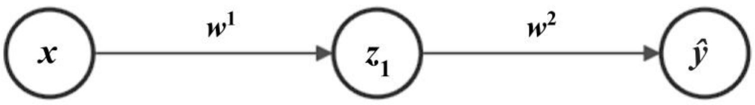


Figure 11

Here we only have to weights, so the gradient has two components:

$$\nabla J(\boldsymbol{W}) = \left(\frac{\partial J(\boldsymbol{W})}{\partial w^1}, \frac{\partial J(\boldsymbol{W})}{\partial w^2}\right)$$

For notation, we can make the following definitions:

$$z^r = w_0^r + y^{r-1}w^r = w_0^r + g(z^{r-1})w^r$$

$$\hat{y} := y^L = g(z^L)$$

By using these two formulas, we can express the variables in our example as:

$$\begin{aligned} z^1 &= xw^1 \\ z^2 &= g(z^1)w^2 \\ \hat{y} &= g(z^2) \end{aligned}$$

If we use the mean squared error loss function, we can write for any training example (from the third equation in the list of known variables (based on Equation 11) and Equation 13):

$$\delta_n^L = \frac{\partial J_n}{z_n^L} = C(\hat{y}_n - y_n)g'(z_n^L)$$
$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^r w_{kj}^r\right) g'(z_{nj}^{r-1}), \qquad j = 1, 2, ..., k_{r-1}$$

By applying these formulas to our example (and setting $C = 2\frac{1}{k_L} = 2$), we get:

$$\delta_n^2 = \frac{\partial J_n}{z_n^2} = 2(\hat{y}_n - y_n)g'(z_n^2)$$
$$\delta_n^1 = \left(\sum_{k=1}^{k_2} \delta_{nk}^2 w_k^2\right) g'(z_n^1) = (\delta_n^2 w^2)g'(z_n^1) = \left(\left[2(\hat{y}_n - y_n)g'(z_n^2)\right] w^2\right) g'(z_n^1)$$

Now, from Equation 12, we can write the gradient as:

$$\frac{\partial J(\boldsymbol{W})}{\partial w^1} = \sum_{n=1}^{N} \delta_n^1 x_n = \sum_{n=1}^{N} \left(\left[2(\hat{y}_n - y_n)g'(z_n^2)\right] w^2\right) g'(z_n^1)x_n \tag{Ex.1}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w^2} = \sum_{n=1}^{N} \delta_n^2 g(z_n^1) = \sum_{n=1}^{N} 2(\hat{y}_n - y_n)g'(z_n^2)g(z_n^1) \tag{Ex.2}$$

Having established all the expressions needed, we can look on what happens through an epoch (a forward pass and a backward pass). Let us say we have an input $x = 2$ and that the desired output for that input was $y = 0.25$. The weights are randomized, so we put them as $w^1 = 0.75$ and $w^2 = 0.5$. We also utilize the sigmoid activation function for g. For the forward pass, we would then get:

$$z^1 = 2 \cdot 0.75 = 1.5$$
$$g(z^1) = \frac{1}{1 + e^{-z^1}} \approx 0.81757$$
$$z^2 = 0.81757 \cdot 0.5 \approx 0.40879$$
$$\hat{y} = g(z^2) = \frac{1}{1 + e^{-z_2}} \approx 0.60080$$

We clearly see that this approximation is very poor. The deviation from our approximation and the desired value is $(\hat{y} - y) = (0.60080 - 0.25) = 0.35080$ which is a fairly large difference. The reason for this is because our network has not been trained yet, all the weights were just randomized. To improve the accuracy of the network, we need to train it. We do this using backpropagation. We calculate the gradient from Equation Ex.1 and Equation Ex.2:

$$\frac{\partial J(\boldsymbol{W})}{\partial w^1} = ([2 \cdot 0.35080 \cdot (0.60080 \cdot (1 - 0.60080))] \cdot 0.5) \cdot (0.81757 \cdot (1 - 0.81757)) \cdot 2 \approx 0.02510$$
$$\frac{\partial J(\boldsymbol{W})}{\partial w^2} = 2 \cdot 0.35080 \cdot (0.60080 \cdot (1 - 0.60080)) \cdot 0.81757 \approx 0.13757$$

As we now have computed the gradient, we can use the gradient descent algorithm to update our weights. For simplicity, let us set the learning rate constant and equal to 1*:*

$$\begin{aligned}\boldsymbol{W} &= \boldsymbol{W} - \lambda \nabla J(\boldsymbol{W}) \\ &= (w_1, w_2) - \nabla J(w_1, w_2) \\ &= (0.75, 0.5) - (0.02510, 0.13757) \\ &= (0.72490, 0.36243)\end{aligned}$$

With our new weights, we can now check if the accuracy has increased by doing another forward pass:

$$\begin{aligned}z^1 &= 2 \cdot 0.72490 = 1.44980 \\ g(z^1) &= \frac{1}{1+e^{-z^1}} \approx 0.80997 \\ z^2 &= 0.80997 \cdot 0.36243 \approx 0.29356 \\ \hat{y} = g(z^2) &= \frac{1}{1+e^{-z^2}} \approx 0.57287\end{aligned}$$

We see that the deviation from our desired value has now decreased, to $(\hat{y} - y) = (0.57287 - 0.25) = 0.32287$ *from* 0.35080 *with our previous weights. Even though it is still not accurate at all, we can notice that the accuracy has increased. By doing more epochs, the accuracy would improve further.*

3 Simple Application of Deep Learning

While we have established the mathematical principles underpinning the construction and training of a neural network, we are not really interested in computing all the calculations by hand. One can tell from Example 11, that the process of computing all the new weights from each epoch increases significantly with the expansion of our network, incorporating additional layers and neurons. Fortunately, the computations are repetitive and not very intricate, making them well-suited to be done on a computer. Today, there exist open-source software libraries such as "TensorFlow" by Google and "PyTorch" by Meta that incorporates all the complicated mathematics and computer science involved in deep learning. This accessibility makes it very easy to apply and utilize deep learning techniques.

In this section, I will embark on practical exploration by tackling both a classification task and a regression task. By applying the theoretical concepts we have discussed, these real-world problems will serve as tangible examples. These examples will lay the groundwork for a subsequent discussion in the following section, where I will consider various optimization methods for training neural networks.

3.1 Classification Problem: MNIST Database

The MNIST (*"Modified National Institute of Standards and Technology"*) database is a large collection of handwritten digits sourced from high school students and employees of the United States Census Bureau [12]. Comprising 60,000 training samples and 10,000 test samples, each image in the dataset is 28x28 pixels, depicting the numbers from 0 to 9. Due to its ease of access and simplicity, MNIST has become a very popular dataset to use for image recognition in deep learning. I will leverage this dataset to train a simple network capable to classifying images to their corresponding digits.

For this purpose, I will utilize "TensorFlow", a powerful library in Python tailored for machine learning and neural network development. I will also import some other packages that be useful for plotting the data.

```
import tensorflow as tf
import random
import time
import datetime
import matplotlib.pyplot as plt
```

TensorFlow conveniently provides a module dedicated to importing the MNIST database.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Whenever we work with a dataset, it can be beneficial to get an idea of what the data looks like. We can visualise the data by using the following code to generate random samples from the training set:

```
images = []
titles = []
for i in range(4):
  r = random.randint(1, 60000)
  images.append(x_train[r])
  titles.append(f"training images [{r}] = {y_train[r]}")
```

```
8
9   plt.figure(figsize=(16,16))
10  for i in range(len(images)):
11    plt.subplot(1, len(images), i+1)
12    plt.imshow(images[i], cmap=plt.cm.gray)
13    plt.title(titles[i])
```

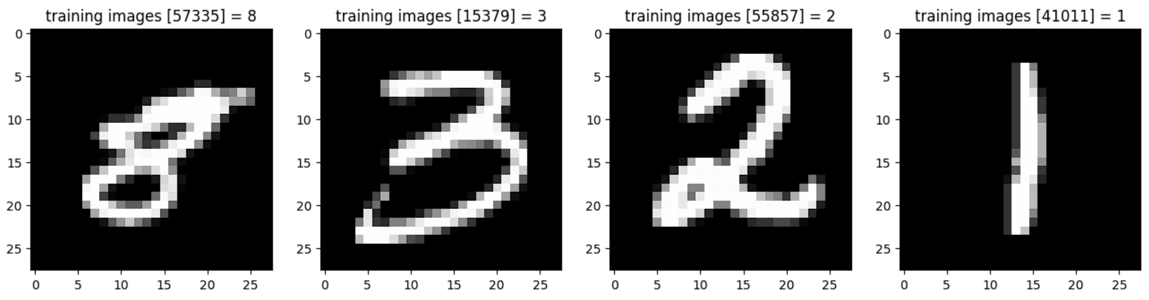

training images [57335] = 8
training images [15379] = 3
training images [55857] = 2
training images [41011] = 1

Figure 12: Random sample from MNIST (output from line 2-13)

Now, before we go on to train a neural network with the MNIST data, it is a common practise to normalize the data. Each picture is made up of 28x28 = 784 pixels, where every pixel holds a value for its grey scale value ranging from 0-255. The function is used in line 14 and 15 from Tensorflow utilize the Euclidean norm to normalize the data. This results in the data now having values from zero to one. The reason for normalizing the data is for all the data to live in the same range. In our example, the data are already all in the same interval, but if we had data inputs where the data had widely different ranges it would be more important to implement some sort of normalization [10] [11].

```
14  x_train = tf.keras.utils.normalize(x_train, axis=1)
15  x_test = tf.keras.utils.normalize(x_test, axis=1)
```

We can finally begin with the process of training our neural network. I use the following code to train the neural network:

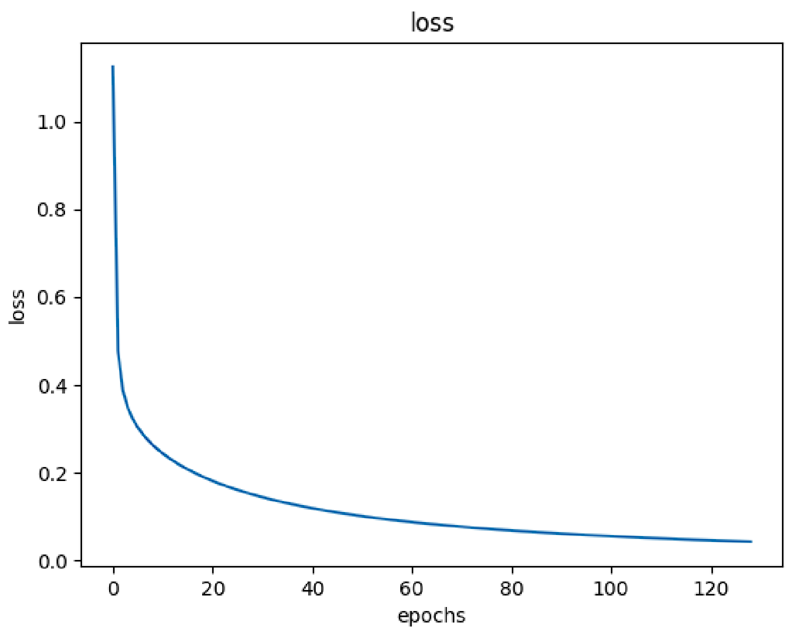
```
16  model = tf.keras.models.Sequential()
17  model.add(tf.keras.layers.Flatten())
18  model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
19  model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
20
21  model.compile(tf.keras.optimizers.SGD(),
22                tf.keras.losses.SparseCategoricalCrossentropy(),
23                metrics = ['accuracy'])
24
25  callback = tf.keras.callbacks.EarlyStopping(monitor='accuracy', min_delta=0.0001,
       patience=3)
26
27  class TimeHistory(tf.keras.callbacks.Callback):
28     def on_train_begin(self, logs={}):
29         self.times = []
30
31     def on_epoch_begin(self, batch, logs={}):
32         self.epoch_time_start = time.time()
33
34     def on_epoch_end(self, batch, logs={}):
        self.times.append(time.time() - self.epoch_time_start)
time_callback = TimeHistory()
```

Here is a quick rundown of the code:

- "*tf.keras.models.Sequential()*" $\rightarrow$ Constructs the feed-forward neural network.
- "*model.add()*" $\rightarrow$ Method to add more elements to the model.
- "*tf.keras.layers.Flatten()*" $\rightarrow$ Flattens the input data from a 28x28 array to one dimensional array of 784 inputs.
- "*tf.keras.layers.Dense(units, activation)*" $\rightarrow$ Creates a hidden layer where you can specify number of neurons (units) and an activation function to that layer. You also use this to make the output layer.
- "*model.compile*" $\rightarrow$ Method to implement a choice of training optimizer and loss function. For classification, we add "metrics=['accuracy']" to denote that we are interested in how accurately the neural network classifies an image correctly.
- "*tf.keras.optimizers.SGD()*" $\rightarrow$ Choose the optimizer "SGD" ("Stochastic Gradient Descent").
- "*tf.keras.losses.SparseCategoricalCrossentropy()*" $\rightarrow$ Choose the loss function "Sparse Categorial Cross-entropy".
- "*tf.keras.callbacks.EarlyStopping()*" $\rightarrow$ Set a stopping condition. If the network does not improve that much, it will stop early. We judge the improvement by a set "min delta", which is the minimum change required from the previous epoch to be categorized as improvement (can be expressed as: Improvement if $|x_t - x_{t-1}| > \Delta_{min}$). The "patience" is how many epochs we are willing to wait to see if there is any improvement.
- The class "*TimeHistory*" is just an additional object so that we can monitor how much time the training process takes.

Now that we have constructed and formulated our neural network model, we can begin the training. TensorFlow makes it really simple for us, the whole backpropagation process is expressed through the code below.

```
mdl = model.fit(x_train, y_train, epochs=1000, callbacks=[callback, time_callback],
   batch_size=32)

plt.plot(mdl.history['loss'])
plt.title('loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()

print(f"Number of Epochs: {len(mdl.history['loss'])}, Training loss: {round(mdl.
   history['loss'][-1], 3)}, Training accuracy: {round(mdl.history['accuracy
   '][-1]*100, 2)}%")
```



```
number of epochs: 129, training time: 0:12:14, training loss: 0.042, training accuracy: 98.97%
```

Figure 13: Training result (output from line 37-45)

As we see from the output in Figure 13, the neural network needed 129 epochs to converge to a 98.97% accuracy in correctly classifying the training data. Note, we can freely set the amount of epochs. I chose 1000 epochs just to be sure it would converge, where I assume it will stop earlier due to the stopping condition we set in line 25. Also note that the ”batch size” refers to how many training samples the network should pass through before it updates the weights.

After having the neural network processing through all the data points, we are interested in how well the network performs. We can test or evaluate the performance of the network by letting the network look at the test samples in the dataset, and measure the percentage of how many they correctly classify.

```python
46 val_loss , val_acc = model.evaluate(x_test , y_test)
47 print(f"Evaluation loss: {round(val_loss , 3)}, Evaluation accuracy: {round(val_acc
      *100 , 2)}%")
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0838 - accuracy: 0.9739
Evaluation loss: 0.084, Evaluation accuracy: 97.39%
```

Figure 14: Evaluation of the neural network (output from lines 46-47)

From Figure 14, the neural network correctly classifies 97.39% of the test data. It is slightly less than the training accuracy, but is to be expected as the network have been trained to specifically identify the training data. When the network then is introduced to some new, unknown data, one may anticipate some deviation. We will discuss later how this accuracy can be improved.

3.2 Regression Problem: Wine Quality

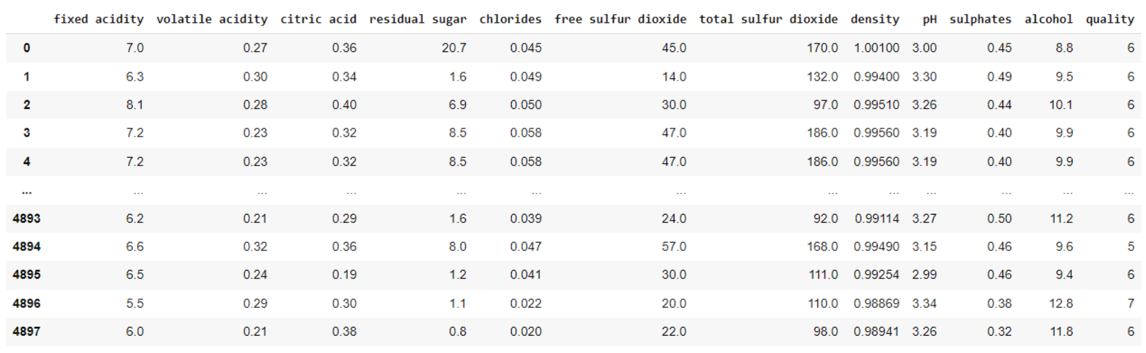
Neural networks can be very useful when dealing with regression problems. While classification tasks aim to correctly identify certain objects or data, regression tasks can be useful to estimate relationships of multiple variables. As a demonstration of a regression problem, I will look at a dataset containing factors to determine the quality of a given white wine [8] (Note that this particular example could also be done as a classification task).

The dataset has 4898 data points, eleven input variables and one output variable. The input variables are different factor that one take into consideration when determining the quality of the wine.

```
import numpy as np
import pandas as pd
import tensorflow as tf
import random
import time
import datetime
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error as mse
```

I start of by using the Pandas-library to import the data. The dataset is downloaded from UC Irvine Machine Learning Repository [8].

```
data = pd.read_csv("winequality-white.csv", sep=";")
data
```



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...	...
4893	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
4894	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
4895	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
4896	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
4897	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

Figure 15: Overview of the dataset

After having imported the dataset, it is important to manipulate it in such a way that it can be used properly. Lines 3-11 show a simple way to separate the data into batches of training data and test data. In line 3, I convert the data into a NumPy-array to make the data easier to deal with. Note also in lines 7 and 8, I split the 75% of the data into the training data and 25% into the test data. For the MNIST example, the training set was $60000/70000 \approx 85,7\%$ which is a higher percentage than I chose here. The reason for this is because the wine quality dataset is much smaller, and we would still like a fair amount of test data.

```
x_train = data.to_numpy()
y_train = np.array(data["quality"])
data.pop("quality")

train_size = int(len(x_train) * 0.75)
test_size = len(x_train) - train_size

x_train, x_test = x_train[0:train_size,:], x_train[train_size:len(x_train),:]
y_train, y_test = y_train[0:train_size], y_train[train_size:len(y_train)]
```

For the MNIST example, we said that we do not really need to normalize the data, because all the data is of the same type. In this case, all the input variables have different values that live in different ranges. From Figure 15, we can observe that ”fixed acidity” typically have values ranging from five to eight, and ”volatile acidity” have values around 0.2 and 0.3. Now it is much more crucial to normalize the data, so that all the inputs have the same range [10] [11].

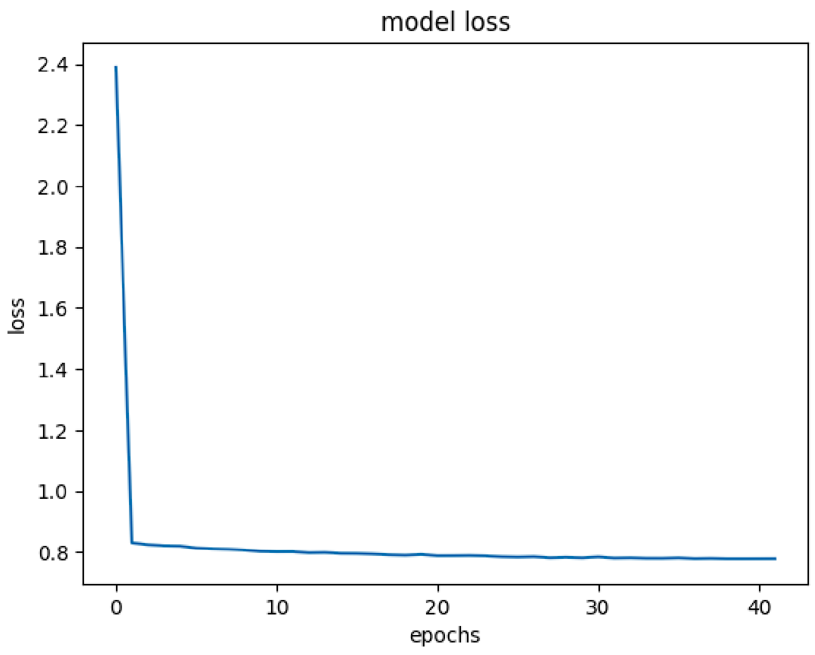
```
x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test = tf.keras.utils.normalize(x_test, axis=1)
```

Now that the data is ready to be used for training, we set up the neural network very similarly to the MNIST example.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.InputLayer( (11,) ))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(1, activation=tf.nn.relu))

model.compile(tf.keras.optimizers.SGD(),
             tf.keras.losses.MeanSquaredError())

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', min_delta=0.0001, patience=3)

class TimeHistory(tf.keras.callbacks.Callback):
  def on_train_begin(self, logs={}):
      self.times = []
  def on_epoch_begin(self, batch, logs={}):
      self.epoch_time_start = time.time()
  def on_epoch_end(self, batch, logs={}):
      self.times.append(time.time() - self.epoch_time_start)
time_callback = TimeHistory()
```

After having established the neural network, we start the training process:

```
mdl = model.fit(x_train, y_train, epochs=1000, batch_size=32, callbacks = [callback, time_callback])

num_epochs = range(0, len(mdl.history['loss']))
plt.plot(num_epochs, mdl.history['loss'])
plt.title('model loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()

print(f"number of epochs: {len(mdl.history['loss'])}, training time: {datetime.timedelta(seconds=int(np.sum(time_callback.times)))}, training loss: {round(mdl.history['loss'][-1], 3)}")
```



```
number of epochs: 42, training time: 0:00:15, training loss: 0.778
```

Figure 16: Output Lines 33-42

From Figure 16, we notice that the plot quickly converges, and result in a training loss of 0.778.

With our trained neural network, we can test its' ability to correctly predict both the training and test data. We do this by letting all the data pass forwards through the network, and calculate the mean squared error from the correct values. I use the function "*mse*" from the scikit-learn library to calculate the mean squared loss.

```
mse(model.predict(x_train, batch_size=32), y_train, squared=False),  mse(model.
    predict(x_test, batch_size=32), y_test, squared=False)
```

```
115/115 [==============================] - 1s 8ms/step
39/39 [==============================] - 0s 5ms/step
(0.8801356159723733, 0.7789493289659192)
```

Figure 17: Output Line 43

Judging from Figure 17, our trained neural network gives a low loss when trying to predict the data it has already seen. It seem to give a lower loss for the test data, which the network has not been trained on.

4 Optimization methods for Neural Networks

When working on training neural networks, we would like for the training process to be as good and as efficient as possible. The goal is to reach an optimal solution in the least amount of computations. Over time, advances has been made and people have come up with better strategies to achieve this. As mentioned, there are two parameters we want to optimize: accuracy and training efficiency.

4.1 Gradient Descent Methods

The standard gradient descent method has low computational demand per iteration, but has a slow convergence rate [11]. The motivation for a better alternative to the standard method has been to try to maintain the low computational demand, but increase the speed of convergence. We will discuss different strategies and methods one can implement to achieve this goal.

4.1.1 Stochastic Gradient Descent and Mini-batching

The method discussed in the Section 2 about the theory of backpropagation is what is called a *batch* scheme of gradient descent. That means that it goes through all the training data before it updates the weights and biases. Note that it also calculates all the data in order, from the first data point to the last one gathered.

Algorithm 5 Batch Schemed Gradient Descent

```
for number of epochs do:
    Compute the gradient, $\nabla J(\boldsymbol{W})$
    Update the weights, $\boldsymbol{W} \leftarrow \lambda_i \nabla J(\boldsymbol{W})$
end for
```

The almost opposite method of this is what is called "*stochastic gradient descent*". This method involves updating all the weights after every single data input, and also shuffling the data between each epoch to remove sequential trends. By shuffling the data, we make the approximation that there are no data reuse, and imagine a steady flow of random data getting processed. Because we make changes due to single data inputs, the updates of the weights happen at a higher variance, which can prevent the algorithm from getting stuck in a local minima. This fluctuation can also lead to overshooting any potential global minima, which can be disadvantageous. Another negative feature is that due to the extensive updating of parameters, it can take longer to train than the batch scheme [9] [11].

Algorithm 6 Stochastic Gradient Descent

```
for number of epochs do:
    Shuffle the data randomly
    for number of data points do:
        Compute the gradient, $\nabla J(\boldsymbol{W})$
        Update the weights, $\boldsymbol{W} \leftarrow \lambda_i \nabla J(\boldsymbol{W})$
    end for
end for
```

Between these two extremes, there is a method that comprises the two previously mentioned methods, called a "*mini-batch*" scheme. It effectively is a less extreme version of the stochastic gradient descent, because we introduce the scheme of updating the parameters after $n < N$ samples, instead of every data point. This leads to the variance getting decreased, and computations getting more effective.

Algorithm 7 Mini-batch Stochastic Gradient Descent

```
for number of epochs do:
    Shuffle the data randomly
    for number of batches do:
        for number of data points in batch do:
            Compute the gradient, $\nabla J(\boldsymbol{W})$
            Update the weights, $\boldsymbol{W} \leftarrow \lambda_i \nabla J(\boldsymbol{W})$
        end for
    end for
end for
```

Note that in the Python-code from the previous section, "*tf.keras.optimizers.SGD()*" applies the stochastic gradient descent scheme, and in "*model.fit*" we define how large these mini-batch sizes should be. Mini-batches can be applied to not only the stochastic gradient descent algorithm, but also to other backpropagation methods.

4.1.2 ADAM

At the time of writing this, ADAM (*"Adaptive Moment Estimation"*) [7] is the most preferred gradient descent method [11]. The method is a variant of the basic gradient descent, but involves strategies to converge faster and also "jump" over local minima if it suspects a better solution further on.

ADAM applies what is called a "*momentum term*", which works as an adaptive learning rate. It utilize the gradients from previous iterations to make the algorithm converge faster. We remember the standard gradient descent has the updating rule:

$$\boldsymbol{W}^{(i+1)} = \boldsymbol{W}^{(i)} - \lambda \nabla J(\boldsymbol{W}^{(i)})$$

With the addition of a momentum term, the updating rule now becomes:

$$\begin{aligned}\Delta\boldsymbol{W}^{(i+1)} &= \alpha \cdot \Delta\boldsymbol{W}^{(i)} - \lambda \nabla J(\boldsymbol{W}^{(i)}) \\ \boldsymbol{W}^{(i+1)} &= \boldsymbol{W}^{(i)} + \Delta\boldsymbol{W}^{(i+1)}\end{aligned}$$

Here α is the *momentum factor*, and $\Delta\boldsymbol{W}$ is introduced as the new gradient term that implements a momentum term. To get an insight into how the momentum term functions, let us look at a function similar to Figure 1 and start iterating from the right. From the figure with constant learning rate, it converges very slowly with a constant learning rate. If we were to implement a momentum term, the algorithm would pick up on if the previous gradients move in the same direction.

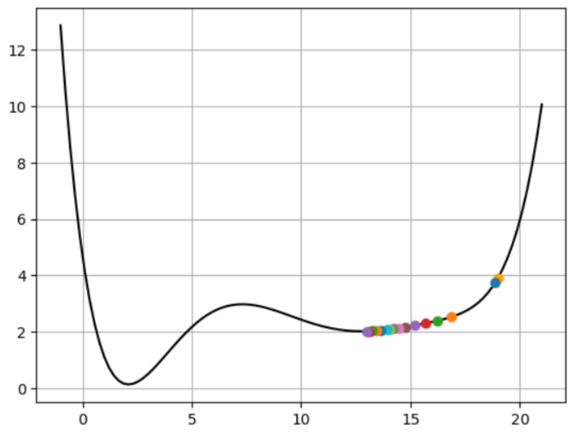


Figure 18: Gradient Descent with Momentum Term ($\alpha = 40, \lambda = 0.1$)

From Figure 18 we see that it converges much faster than in Figure 1 with the same learning rate. We see from the right in the plot that the algorithm has picked up on the curvature, and takes a larger "jump" towards the minima.

Clearly, there is a lot of power in these momentum terms. Note that what we have discussed is the basic idea of momentum terms, and that there exist multiple gradient descent algorithms that use different kinds of momentum terms. Let us now look at the ADAM algorithm and how it utilizes momentum terms [7] [11]:

Algorithm 8 ADAM

```
Require: $\lambda$: Step-size
Require: $\beta_1, \beta_2 \in [0, 1)$: Momentum factors
Require: $J(\boldsymbol{W})$: Loss function
Require: $W^0$: Initial weights
  $m_0 \leftarrow 0$    (Initialize $1^{st}$ moment vector)
  $v_0 \leftarrow 0$    (Initialize $2^{nd}$ moment vector)
  while $\boldsymbol{W}$ not converged do
      $g^{(i)} = \nabla J(\boldsymbol{W}^{(i-1)})$
      $m^{(i)} = \beta_1 \cdot m^{(i-1)} + (1-\beta_1) \cdot g^{(i)}$
      $v^{(i)} = \beta_2 \cdot v^{(i-1)} + (1-\beta_2) \cdot [g^{(i)}]^2$
      $\hat{m}^{(i)} = m^{(i)}/(1-\beta_1^i)$
      $\hat{v}^{(i)} = v^{(i)}/(1-\beta_2^i)$
      $W^{(i)} = W^{(i-1)} - \lambda \cdot \hat{m}^{(i)}/(\sqrt{\hat{v}^{(i)}} + \epsilon)$
  end while
```

Note that $[g^{(i)}]^2$ refers to element-wise multiplication. All operations on vector are element-wise. The creators of ADAM recommend $\alpha = 0.001, \beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as good default values [7]. These are also the default values for the ADAM function used in TensorFlow.

As we can see from the Algorithm 8, ADAM uses two momentum terms: m and v. The terms $\hat{m}$ and $\hat{v}$ are adjusted values of the obtained m and v. When i is low, it makes the denominator closer to 0, such that $\hat{m}$ and $\hat{v}$ blow more up early on in the iterations. Later on, the exponents of the momentum factors will become more dominant and go towards 0, and the denominator will go towards 1. The consequence of this is that early on in the iterations, the algorithm will take bigger leaps in the beginning of the training process, and have the opportunity to overshoot local minima.

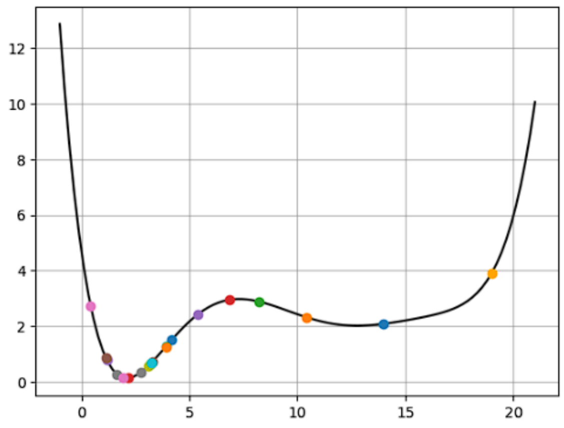


Figure 19: ADAM ($\lambda = 5$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$)

Figure 19 demonstrates that the ADAM algorithm is capable of finding a better minima of a function.

4.2 Comparison

Having discussed some variations of the gradient descent algorithm, we would like to see how their performance may differ. To check their performance, I will train multiple neural networks with both SGD ("Stochastic Gradient Descent") and ADAM for different mini-batch sizes. I will use the examples from Section 3 as my neural networks.

I will approach this comparison by running the same lines of code as in Section 3, but with both SGD and ADAM with various mini-batch sizes as optimizers. For the mini-batch sizes, I choose to use powers of two. Since a batch size of 30 is probably not that different than for 32 or 34, it can be more interesting to look at larger differences in mini-batch sizes. By using powers of two, we can get a good idea of what ranges of mini-batch sizes that seems more favorable. I will let the classification example run for 100 epochs, and the regression example run for 250 epochs. In the code in Section 3, we applied a stopping condition for the training process, which for this comparison I will abandon. It will be easier to compare the convergence of the different batch sizes when they all do the same amount of epochs. The numbers 100 and 250 could be chosen differently, but from trail and error it seemed to a number that was well fitted for these comparisons.

Let us now have a look at the result, both from the MNIST classification example and the wine quality regression example. Note that for batch size = 1 refers to the stochastic scheme of gradient descent, batch size = 4898 or 60000 refers to the batch scheme, and all the others are mini-batch schemes.

4.2.1 Various Batch-size for Classification Problem

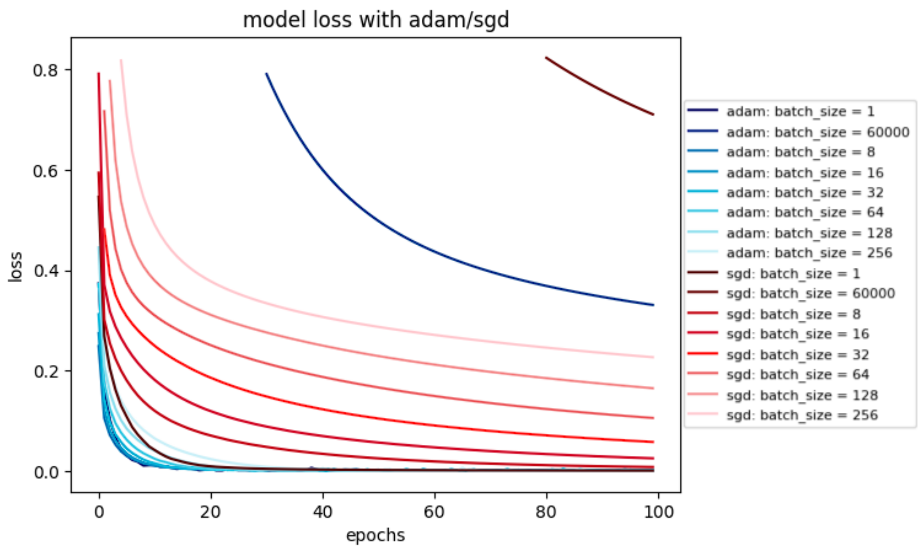
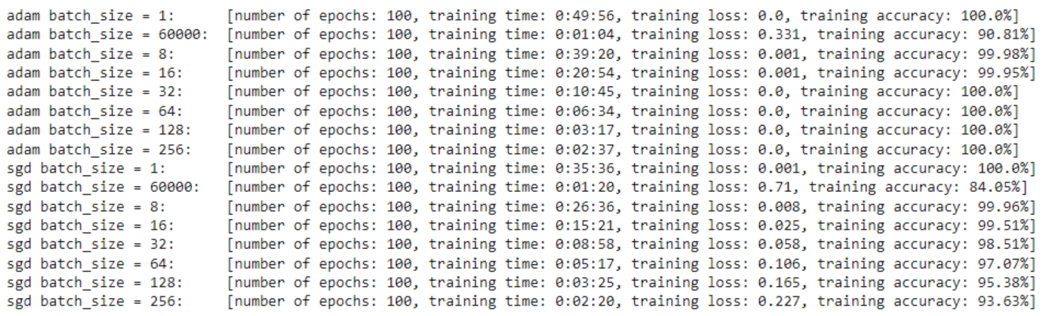


Figure 20: Classification Model: Loss with ADAM and SGD



```
adam batch_size = 1:      [number of epochs: 100, training time: 0:49:56, training loss: 0.0, training accuracy: 100.0%]
adam batch_size = 60000:  [number of epochs: 100, training time: 0:01:04, training loss: 0.331, training accuracy: 90.81%]
adam batch_size = 8:      [number of epochs: 100, training time: 0:39:20, training loss: 0.001, training accuracy: 99.98%]
adam batch_size = 16:     [number of epochs: 100, training time: 0:20:54, training loss: 0.001, training accuracy: 99.95%]
adam batch_size = 32:     [number of epochs: 100, training time: 0:10:45, training loss: 0.0, training accuracy: 100.0%]
adam batch_size = 64:     [number of epochs: 100, training time: 0:06:34, training loss: 0.0, training accuracy: 100.0%]
adam batch_size = 128:    [number of epochs: 100, training time: 0:03:17, training loss: 0.0, training accuracy: 100.0%]
adam batch_size = 256:    [number of epochs: 100, training time: 0:02:37, training loss: 0.0, training accuracy: 100.0%]
sgd batch_size = 1:       [number of epochs: 100, training time: 0:35:36, training loss: 0.001, training accuracy: 100.0%]
sgd batch_size = 60000:   [number of epochs: 100, training time: 0:01:20, training loss: 0.71, training accuracy: 84.05%]
sgd batch_size = 8:       [number of epochs: 100, training time: 0:26:36, training loss: 0.008, training accuracy: 99.96%]
sgd batch_size = 16:      [number of epochs: 100, training time: 0:15:21, training loss: 0.025, training accuracy: 99.51%]
sgd batch_size = 32:      [number of epochs: 100, training time: 0:08:58, training loss: 0.058, training accuracy: 98.51%]
sgd batch_size = 64:      [number of epochs: 100, training time: 0:05:17, training loss: 0.106, training accuracy: 97.07%]
sgd batch_size = 128:     [number of epochs: 100, training time: 0:03:25, training loss: 0.165, training accuracy: 95.38%]
sgd batch_size = 256:     [number of epochs: 100, training time: 0:02:20, training loss: 0.227, training accuracy: 93.63%]
```

Figure 21: Classification Model: Training Data

Note: batch size = 1 for ADAM and SGD have been simplified to train on a random sample of size N = 10000 from the original dataset. The reason for this is to save time of computation. This simplification does not affect the story of the figure.

4.2.2 Various Batch Sizes for Regression Problem

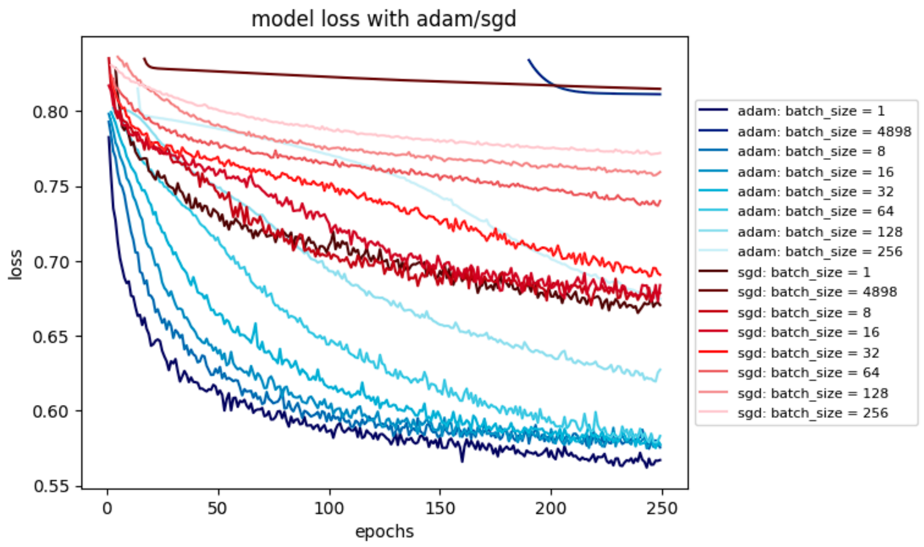
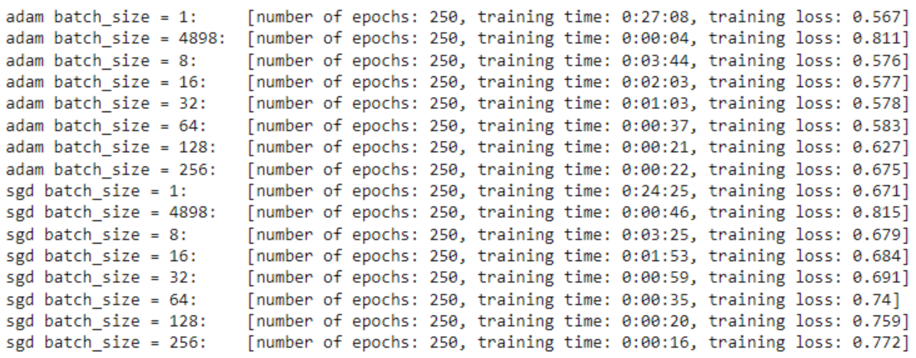


Figure 22: Regression Model: Loss with ADAM and SGD



```
adam batch_size = 1:     [number of epochs: 250, training time: 0:27:08, training loss: 0.567]
adam batch_size = 4898:  [number of epochs: 250, training time: 0:00:04, training loss: 0.811]
adam batch_size = 8:     [number of epochs: 250, training time: 0:03:44, training loss: 0.576]
adam batch_size = 16:    [number of epochs: 250, training time: 0:02:03, training loss: 0.577]
adam batch_size = 32:    [number of epochs: 250, training time: 0:01:03, training loss: 0.578]
adam batch_size = 64:    [number of epochs: 250, training time: 0:00:37, training loss: 0.583]
adam batch_size = 128:   [number of epochs: 250, training time: 0:00:21, training loss: 0.627]
adam batch_size = 256:   [number of epochs: 250, training time: 0:00:22, training loss: 0.675]
sgd batch_size = 1:      [number of epochs: 250, training time: 0:24:25, training loss: 0.671]
sgd batch_size = 4898:   [number of epochs: 250, training time: 0:00:46, training loss: 0.815]
sgd batch_size = 8:      [number of epochs: 250, training time: 0:03:25, training loss: 0.679]
sgd batch_size = 16:     [number of epochs: 250, training time: 0:01:53, training loss: 0.684]
sgd batch_size = 32:     [number of epochs: 250, training time: 0:00:59, training loss: 0.691]
sgd batch_size = 64:     [number of epochs: 250, training time: 0:00:35, training loss: 0.74]
sgd batch_size = 128:    [number of epochs: 250, training time: 0:00:20, training loss: 0.759]
sgd batch_size = 256:    [number of epochs: 250, training time: 0:00:16, training loss: 0.772]
```

Figure 23: Regression Model: Training Data

4.2.3 Discussion

From Figures 20-23 it seems that ADAM is the favorable algorithm. In both plots (Figures 20 and 22), ADAM is capable of finding better minima for the loss function, resulting in a lower overall loss, and also do so more efficiently by converging faster.

Figures 20 and 22 demonstrates the batch and stochastic scheme of gradient descent, and also why they are not advantageous compared to the mini-batch method. The stochastic method (batch size = 1) converges towards the lowest loss, but it has substantially longer computation times because it has to update the weights for every data point. The batch method (batch size = 4898 / 60000) takes shorter amount of time, but do not converge to the proper solution.

From Figure 20 and 22, we notice the difference between the different mini-batch sizes. In Figures 22 and 23, the ADAM mini-batches converge at a slower rate the higher the mini-batch size is, but we see that the mini-batch sizes 8-64 result in a similar training loss. From the same figure, the SGD mini-batch sizes show larger differences in the loss, and do not seem to converge to a similar solution. This same story can be noted from Figures 20 and 21 as well.

For the regression example, the best choice of a mini-batch size, with respect to having the lowest loss and also lowest training time, seems to be a batch size of 8-64. Having a small mini-batch size is what one might expect to the optimal, as it is closer in size to the stochastic scheme than the batch scheme. Here, with batch sizes 128 and 256, the training time is slightly reduced, but tends to increase too much in the training loss. On the other hand, the classification example shows to have the lowest loss for batch sizes 64-256 for the ADAM algorithm, but notice that the loss increases with larger batch sizes for the SGD algorithm. Because this dataset is much larger than the one in the regression example, the training times also become an issue for the smallest batch sizes, and might want to increase the batch size to save time without losing accuracy.

To conclude, ADAM would be the preferred optimization method for training a neural network with these two datasets. For the regression example, a mini-batch size of 8-64 shows to be the most favorable. For the classification example, because the dataset is larger, a slightly larger mini-batch size can be applied, for instance 64-256.

5 Conclusion

In today's rapidly evolving landscape of industry and research, with increasing amount of data, neural networks and deep learning have become powerful tools to help recognize patterns in the data. As datasets and models grow in size and complexity, it is essential that the training processes go as efficiently as possible. Throughout this thesis, there have been established a solid foundation and understanding of how these training processes work, and how gradient descent algorithms are further developed to enhance their efficiency. There have been considered specific versions of standard gradient descent like "Batch Gradient Descent", "Stochastic Gradient Descent" and "Mini-Batch Gradient Descent", alongside an improved and more advanced variant known as the ADAM algorithm. By comparing these different algorithms on two practical examples, ADAM has clearly shown its superiority in optimizing performance. Furthermore, our comparison reveals that the mini-batch approach to gradient descent is favored for its remarkable reduction in computation time, without sacrificing accuracy.

References

[1] Berke Akkaya and Nurdan Çolakoğlu. "Comparison of Multi-class Classification Algorithms on Early Diagnosis of Heart Diseases". In: Sept. 2019, pp. 25–26.

[2] Alexander Amini. *MIT Introduction to Deep Learning — 6.S191*. `https://www.youtube.com/watch?v=QDX-1M5Nj7s&t=1105s`. 2023.

[3] Nicolas Boumal. *MATH-329 Continuous Optimization*. Lecture Notes. 2023.

[4] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. `https://arxiv.org/abs/2109.14545`. 2022. arXiv: `2109.14545 [cs.LG]`.

[5] Gareth James; Daniela Witten; Trevor Hastie and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Second Edition. Springer, 2021. ISBN: 978-1071614181.

[6] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. Eleventh Edition. McGraw-Hill Education, 2021, pp. 542–545. ISBN: 978-1-260-57587-3.

[7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`.

[8] Paulo Cortez; Antonio Cerdeira; Fernando Almeida; Telmo Matos and José Reis. *Wine Quality*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C56S3T. 2009.

[9] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: `1609.04747`.

[10] Dalwinder Singh and Birmohan Singh. "Investigating the impact of data normalization on classification performance". In: *Applied Soft Computing* 97 (2020), p. 105524. ISSN: 1568-4946. DOI: `https://doi.org/10.1016/j.asoc.2019.105524`. URL: `https://www.sciencedirect.com/science/article/pii/S1568494619302947`.

[11] Sergios Theodoridis. "Chapter 18 - Neural Networks and Deep Learning". In: *Machine Learning (Second Edition)*. Ed. by Sergios Theodoridis. Second Edition. Academic Press, 2020, pp. 901–1038. ISBN: 978-0-12-818803-3. DOI: `https://doi.org/10.1016/B978-0-12-818803-3.00030-1`. URL: `https://www.sciencedirect.com/science/article/pii/B9780128188033000301`.

[12] Corinna Cortes Yann LeCun and Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. `https://web.archive.org/web/20220331130319/https://yann.lecun.com/exdb/mnist/`.