From Linear and Logistic Regression to Neural Networks

Stian Aannerud, Marius Bjerke Børvind, Dennis Fremstad and Andreas Wetzel

November 18, 2022

In this project we have explored gradient descent methods to fit models to data, using both polynomial regression, logistic regression and a feed-forward neural network (FFNN) in regression and classification cases. We have compared how these models fare against traditional means such as the Ordinary Least Squares and Ridge methods. While the neural network requires more manual adjustment of hyperparameters, we found that it could outperform previous methods with the right configuration. The UCI Breast Cancer Wisconsin (Diagnostic) Data Set[1] served as our basis in classification tests, while we continued analysis of the Franke function to compare results with our previous paper[2]. For regression cases we found that the Mean Squared Error (MSE) cost function gave the best result, while the Cross Entropy cost fit the classification/logistic cases better. All hidden network layers showed best promise with the standard Sigmoid activation function, while the output layer function varied between tests. Regression cases used a linear/identity output, while classification and logistic cases used the hyperbolic tangent and Sigmoid respectively. The learning rate was primarily adjusted through adaptive methods such as Adam. We found that the MSE for our FFNN, performed on the Franke function, was 27% lower for OLS. The FFNN also had an accuracy score of 96% compared to 90% for logistic regression when studying the Wisconsin Breast Cancer data.

1 Introduction

Neural networks are series of algorithms that aim to recognize patterns in data just like a human brain. Whereas traditional methods of regression rely on the user to define features which are used to fit the data, such as polynomial factors or Gaussian curves, neural networks can put together its own features through layers of nodes, taking only the raw data as input. Today neural networks are used to solve problems such as pattern recognition, classification and optimization, and is therefore a valuable tool for anyone trying to pull information from data, such as us.

The main aim of this project is to develop our own feed-forward neural network, study which parameters make it up, and test it out on a couple sets of data. We start by developing a simple gradient descent code to study the various parameters which affects regression. Among others we'll look at the learning rate, with various adaptive methods, and Stochastic Gradient Descent (SGD), where we only use a partial amount of the data with each update.

We will compare our FFNN to already obtained results from regression methods such as ordinary least squares (OLS), Ridge- and logistic regression. Our neural network will be tested in regression on the Franke function, comparing our results to our previous paper, where we used OLS/Ridge matrix inversion methods and polynomial features[2]. For the classification and logistic cases, we will use the FFNN to analyse the UCI Wisconsin Breast Cancer dataset [1] to predict if breast tumors are benign or malignant based on a list of parameters.

We begin by describing background information in the Theory section. Here we introduce the mathematics and ideas for the feed-forward neural network, logistic regression, and gradient decent methods. In the Method section, we cover our choice of numerical implementation and how we have chosen to implement the theory into code. The results and discussion section contains the results we obtained using our methods and shows how we were able to improve our results by using other methods. While the conclusion section contains a summary of our results.

All code used in this project can be found at our GitHub page `https://github.com/frdennis/FYS-STK4155`.

2 Theory

In most machine learning problems one start with some data $\mathbf{y}$, a design matrix $\mathbf{X}$, unknown parameters $\boldsymbol{\beta}$ and a cost function $C(\boldsymbol{\beta})$. The aim is then to find the optimal parameters $\hat{\boldsymbol{\beta}}$ which minimize the cost function. That means the parameters which solve the following equation,

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = 0. \tag{1}$$

Unlike in ordinary least squares (OLS) and Ridge regression, the equation above does not in general have analytical solutions. This is the case in logistic regression since the cost function is not linear in the parameters $\boldsymbol{\beta}$. Before we discuss logistic regression we set up the cost function and its derivative for OLS and Ridge regression,

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(x_{ij}\beta_j - y_i)^2 \tag{2}$$

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_j} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}), \tag{3}$$

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(x_{ij}\beta_j - y_i)^2 + \lambda\sqrt{\sum_{i=0}^{n-1}\beta_i^2}, \tag{4}$$

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_j} = 2(\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + \lambda\boldsymbol{\beta}). \tag{5}$$

Here, x_{ij} are the elements in the design matrix $\mathbf{X}$, n is the number of data points and λ is a regularization parameter. The cost function for OLS is just the mean squared error (MSE) which will be used to evaluate the results we obtain in this project.

2.1 Logistic regression

Logistic regression is used in classification problems where we want the output, here denoted categories, to be discrete variables. For example in a binary classification case we want the categories y_i to be 1 or 0. This can be achieved by using the sigmoid function which in general is written as

$$f(z) = \frac{1}{1+e^{-z}}, \tag{6}$$

for some variable z.

Since there are only two categories in a binary classification problem, we define the probabilities for each of them as

$$p(y_i = 1|x_{ij}, \beta_j) = \frac{\exp(x_{ij}\beta_j)}{1+\exp(x_{ij}\beta_j)}, \tag{7}$$

$$p(y_i = 0|x_{ij}, \beta_j) = 1 - p(y_i = 1|x_{ij}, \beta_j). \tag{8}$$

After these have been computed we set the probability which is larger than 0.5 to 1 and the other to 0.

Next, we want the probability of obtaining the data to be as high as possible. This is done by approximating the total likelihood for all possible outcomes as

$$\begin{aligned} P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} &\left[p(y_i = 1|x_{ij}, \beta_j)\right]^{y_i} \\ &\times \left[1 - p(y_i = 1|x_{ij}, \beta_j))\right]^{1-y_i}, \end{aligned} \tag{9}$$

where $\mathcal{D}$ is the dataset. This expression can be rewritten to give the cross entropy,

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n}\left[y_i(x_{ij}\beta_j) - \log\left(1+\exp\left(x_{ij\beta j}\right)\right)\right]. \tag{10}$$

This expression is used as the cost function in logistic regression. That means we want to find the parameters $\boldsymbol{\beta}$ which minimizes it. Thus we need its derivative which can be expressed on vector form as follows,

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{p}). \tag{11}$$

$\mathbf{y}$ and $\mathbf{p}$ are vectors with the categories y_i and probabilities $P(y_i|x_{ij}\beta_j)$ respectively. Since $\mathbf{p}$ contains exponentials with $\boldsymbol{\beta}$, we are not able to set the derivative to zero and solve the equation analytically. We therefore need other ways of finding the optimal parameters $\hat{\boldsymbol{\beta}}$.

2.2 Gradient descent methods

On way of obtaining the optimal parameters $\hat{\boldsymbol{\beta}}$, when we can not solve Equation 1 analytically, is to use the Newton-Raphson method which can be expressed as

$$\boldsymbol{\beta}^{\text{new}} = \boldsymbol{\beta}^{\text{old}} - \left(\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}\partial\boldsymbol{\beta}^T}\right)^{-1}_{\boldsymbol{\beta}^{\text{old}}} \times \left(\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}\right)_{\boldsymbol{\beta}^{\text{old}}}. \tag{12}$$

We see that this is an iterative method where we need both the first and second derivative of the cost function to update $\boldsymbol{\beta}$. Computing the second derivative of the cost function might involve matrix multiplication of the design matrix $\mathbf{X}$ and its transpose. In addition, the resulting matrix has to be inverted. Thus, for large matrices it is computationally expensive to find the second derivative. We therefore want to avoid performing these computations at each iteration.

This can be achieved by using the fact that for a small enough $\eta > 0$, we have $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$ [4, ch. 7] if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta_k \nabla F(\mathbf{x}_k). \tag{13}$$

Instead of using Equation 12, we can therefore use

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta_k \nabla_{\beta} \mathcal{C}(\boldsymbol{\beta}_k), \tag{14}$$

where $\nabla_{\boldsymbol{\beta}} = \partial / \partial \boldsymbol{\beta}$, and η is a hyperparameter called the learning rate. It can be seen as the length of one iteration step. This way of updating $\boldsymbol{\beta}$ is part of what is called gradient descent (GD) methods.

The way GD methods works is by first guessing initial values $\boldsymbol{\beta}_0$. Then we use these to compute the derivative and compare this to a chosen convergence criterion. If the derivative is larger than this criterion, we use Equation 14 to update $\boldsymbol{\beta}$ and perform the same procedure. This is done until the convergence criterion is met. It is also possible to set a convergence criterion for the cost function.

An advantage with the hyperparameter η is that we avoid computing the second derivative of the cost function. However, it is not straightforward to determine the optimal value of the learning rate. Since it can be seen as the iteration step size, very small values of η means it will take long for the method to converge. Large values, on the other hand, makes it difficult to find the minimum of the cost function, and the method might not converge at all. The learning rate therefore has to be tuned for the specific problem. Approaches for tuning the learning rate will be discussed later.

For GD methods, as mentioned, we have to choose the initial values $\boldsymbol{\beta}_0$. This choice determines the starting point on the cost function which again affects the time it takes to find the optimal parameters. For example, if we start far away from the minimum of the cost function, the method might converge very slowly, or we could end up in a local minimum.

These problems can be dealt with in several ways. One approach is called gradient descent with momentum (GDM). The parameters $\boldsymbol{\beta}$ are then updated as follows,

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \eta_t \nabla_{\beta} \mathcal{C}(\boldsymbol{\beta}_t) - \gamma \Delta \boldsymbol{\beta}_t. \tag{15}$$

We see that the next $\boldsymbol{\beta}$ now also depends on a momentum term containing the previous change $\Delta\beta_t$ and a momentum parameter, $0 \leq \gamma \leq 1$. The objective is to include a kind of memory of the direction in which we are moving. Since the momentum term will accumulate the previous changes in $\boldsymbol{\beta}$, the iteration steps make the convergence quicker even in parts with small gradient. Another benefit with including momentum is that oscillations in directions with high curvature will be suppressed. That means there is less chance of ending up in a local minimum.

A second way of dealing with slow convergence and the possibility of ending up in a local minimum is called stochastic gradient descent (SGD). In this method, the data is first reshuffled randomly before it is split into mini-batches of size M. If n is the number of data points, we have n/M mini-batches. Then, for each mini-batch, drawn randomly, we compute the gradient and update $\boldsymbol{\beta}$.

It is common to call an iteration over all mini-batches for an epoch. A number of epochs can then be chosen, and after all iterations the result is evaluated. This can be done by computing both the gradient and cost function with all data points.

An advantage with SGD is that computing many gradients for a small number of data points requires less floating point operations than computing the gradient for all data points once. By both distributing the data points and draw the mini-batches randomly, we have greater chance of avoiding local minima. To decrease the convergence time, it is usual to include momentum in SGD.

2.3 Adaptive learning rates

Next, we look at ways of optimizing the search for the optimal parameters $\hat{\boldsymbol{\beta}}$ by considering the learning rate η which has to be tuned for the particular problem. In this project we will look at several ways to tune the learning rate. One possibility is simply to set it to a constant and perform the regression. This is then done for several values of η to find the one that gives the best result.

We will also consider methods with adaptive learning rates. These methods make use of the gradient to update η for each iteration. The aim with adaptive learning rates is to adapt the learning rates to the shape of the cost function. For example in flat regions, we want a large learning rate to make the convergence faster. In steeper regions, in particular close to the minimum, we want a smaller learning rate. How this is achieved is shown in the following sets of equations.

The first set below, 16, shows the algorithm for updating the learning rate with the adaptive gradient method (AdaGrad),

$$\begin{aligned} \mathbf{g}_t &= \nabla_\beta C(\boldsymbol{\beta}) \\ \mathbf{r}_t &= \mathbf{r}_{t-1} + \mathbf{g}_t^2 \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \eta \frac{\mathbf{g}_t}{\delta + \sqrt{\mathbf{r}}}. \end{aligned} \tag{16}$$

δ is a small constant, often between 10^{-6} and 10^{-8}[7], added in the denominator to prevent divergence when $\mathbf{r}$ is small. $\mathbf{r}$ is an accumulation variable which is a sum of all previous gradients squared. This variable therefore always increases which means the learning rate decreases for each iteration. We also notice that the learning rate decreases less when the gradient is small and opposite. However, since the learning rate always decreases, it might become smaller than necessary before we approach the minimum. This is dealt with in the next methods.

The next set of expressions, 17, shows the algorithm for the method called RMS propagation (RMSProp),

$$\begin{aligned} \mathbf{g}_t &= \nabla_\beta C(\boldsymbol{\beta}) \\ \mathbf{r}_t &= \rho \mathbf{r}_{t-1} + (1-\rho)\mathbf{g}_t^2 \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \eta \frac{\mathbf{g}_t}{\sqrt{\mathbf{r}_t + \delta}}. \end{aligned} \tag{17}$$

δ plays the same role here as for AdaGrad. The decay rate ρ should lie between 0 and 1. It often takes values around 0.9 [4, ch. 7]. As for AdaGrad, we see that the learning rate decreases for each iteration due to the accumulation variable $\mathbf{r}$. However, unlike for AdaGrad, the parameter ρ makes sure the learning rate does not decrease too quickly.

The last set of equations describes the algorithm for Adam, which has its name from "adaptive moments",

$$\begin{aligned} \mathbf{g}_t &= \nabla_\beta C(\boldsymbol{\beta}) \\ \mathbf{s}_t &= \rho_1 \mathbf{s}_{t-1} + (1-\rho_1)\mathbf{g}_t \\ \mathbf{r}_t &= \rho_2 \mathbf{r}_{t-1} + (1-\rho_2)\mathbf{g}_t^2 \\ \hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\rho_1^t} \\ \hat{\mathbf{r}}_t &= \frac{\mathbf{r}_t}{1-\rho_2^t} \\ \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \eta \frac{\hat{\mathbf{s}}_t}{\sqrt{\hat{\mathbf{r}}_t} + \delta}. \end{aligned} \tag{18}$$

δ is again a small value like in AdaGrad and RMSProp. ρ_1 and ρ_2 has the same function as ρ in RMSProp. Again they take values between 0 and 1. Typical values are $\rho_1 = 0.9$ and $\rho_2 = 0.999$[7].

In Adam, the learning rate is also updated by the accumulation variable for the gradient as well as by the accumulation variable for the gradient squared. In addition, this method also corrects for bias in the accumulation variables seen as $\hat{\mathbf{r}}$ and $\hat{\mathbf{s}}$. It should be mentioned that in the above equations, all multiplications and divisions of vectors are performed elementwise.

All methods and algorithms presented so far, contribute to what is called the optimization part of machine learning. They will also be important when moving on to discuss neural networks.

2.4 Neural network

In this project we will consider a feed-forward neural network with back propagation. The main parts of such a network are an input layer, one or more hidden layers and an output layer. The hidden layers consists of what we will here refer to as nodes. The optimal number of hidden layers and nodes in each layer has to be tested for the specific problem.

We here denote the target, meaning the data we want the network to reproduce, as $\mathbf{t}$, the input data as $\mathbf{x}$ and the model $\tilde{\mathbf{y}}$. The input layer then consists of the values in $\mathbf{x}$ which become the input nodes. As always, the data should be split randomly into test and training data such that only parts of the data is used to train the network to avoid overfitting.

Here, we only consider so-called fully connected neural networks. That means all nodes in two adjacent layers are connected. A feed-forward step therefore starts with each input node being connected to each node in the first hidden layer. This is done by computing the following for each node i in the first hidden layer,

$$a_i^1 = f(z_i^1), \tag{19}$$

where f is called the activation function and the weighted sum z_i is found from

$$z_i^1 = \sum_{j=1}^{n} w_{ij}^1\, x_j + b_i^1. \tag{20}$$

Here n is the number of nodes in the first hidden layer. w_{ij} and b_i are weights and biases respectively. The superscript 1 refers to the first hidden layer.

After a_i^1 has been found for all nodes in the first hidden layer, they are used as input to the second hidden layer which means the weighted sum becomes

$$z_i^2 = \sum_{j=1}^{n} w_{ij}^2\, a_j^1 + b_i^2. \tag{21}$$

n is now the number of nodes in the second hidden layer. This continues all the way to the output layer. That means we can write a general expression for the output from each layer,

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l\, a_j^{l-1} + b_i^l\right). \tag{22}$$

l denotes the layer and N_{l-1} is the number of nodes in layer $l-1$. The activation function f can be different for each layer which is the reason why a superscript l has been added. This expression can also be written on vector form,

$$\mathbf{a}^l = f^l(\mathbf{z}^l) = f^l\left(\mathbf{W}\mathbf{a}^{l-1} + \mathbf{b}^l\right). \tag{23}$$

It was mentioned previously that the optimal number of nodes in each hidden layer has to be found by testing. On the other hand, the number of nodes in the output layer should be set so that the final output can be compared to the target t by the use of a cost function. For regression problems, the number of output nodes is the same as the number of input nodes. In classification problems, the number of output nodes should be equal to the number of categories.

After the model has been evaluated, a back propagation step is performed. During this step, the weights and biases are updated based on the error in the different layers δ_i^l. First, we have to find the error in output layer L,

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial a_j^L}, \tag{24}$$

where $f'(z_j^L)$ means the derivative with respect to z_j^L. The error for the other layers can then be found from

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l). \tag{25}$$

It should be noted that in the back propagation algorithm, l runs backwards from $L, L-1, ..., 2, 1$. At each layer the weights and biases are updated as follows,

$$w_{jk}^l = w_{jk}^l - \eta\,\frac{\partial \mathcal{C}(a_k^l)}{\partial w_{jk}^l}, \tag{26}$$

$$b_j^l = b_j^l - \eta\frac{\partial \mathcal{C}(a_j^l)}{\partial b_j^l}, \tag{27}$$

where it can be found that

$$\frac{\partial \mathcal{C}(a_k^l)}{\partial w_{jk}^l} = \delta_j^l\, a_k^{l-1}, \tag{28}$$

$$\frac{\partial \mathcal{C}(a_j^l)}{\partial b_j^l} = \delta_j^l. \tag{29}$$

Here η is again the learning rate discussed earlier in the report. These update expressions resemble the expressions for GD, and it is normal to include both SGD and SGD with momentum in a FFNN with back propagation. Thus, also the various algorithms for adaptive learning rates, discussed previously, are frequently used for optimization.

When we have reached the first hidden layer and updated all the sets of weights and biases, we again perform a feed-forward step as described earlier. The result is then evaluated once more before we again take a back propagation step. Here, we say that one iteration consist of both a feed-forward and a back propagation step. Either a chosen number of iterations can be set, or one run as many iterations as needed till a convergence criterion is met.

Now that the algorithms have been presented, we look closer at the different components. We start with the initialization of the weights and biases. This has to be done for all layers except the input layer. In the method section we present and discuss several ways of initializing the weights.

The biases could be set to the same small value for all nodes in each layer. The reason is that the biases mainly ensure that no nodes get zero activation during the first feed-forward and back propagation step.

Next, there is a large number of possible activation functions f. The requirements for these functions comes from the universal approximation theorem. It states that if a function is non-constant, bounded, monotonically increasing and continuous, it is possible to approximate a continuous function to arbitrary accuracy by using a FFNN with only one hidden layer and a finite number of nodes [4, ch. 13].

In this project we will try out different activation functions. The first is the sigmoid function discussed in the context of logistic regression and shown in Equation 6. Since we also need the derivatives of the activation functions in the back propagation step, we note that the derivative of the sigmoid function can be written

$$f'(z) = f(z)(1 - f(z)). \tag{30}$$

Here $f(z)$ is the sigmoid function from Equation 6. The derivative is taken with respect to z.

Other activation functions are the rectified linear unit (RELU) which is just, along with its derivative,

$$f(z) = \max(0,\, z), \tag{31}$$

$$f'(z) = \begin{cases} 1 & z > 0, \\ 0 & z < 0, \end{cases} \tag{32}$$

$$\tag{33}$$

and leaky RELU, defined as

$$f(z) = \begin{cases} z & z \geq 0, \\ \alpha\, z & z < 0, \end{cases} \tag{34}$$

$$f'(z) = \begin{cases} 1 & z \geq 0, \\ \alpha & z < 0, \end{cases} \tag{35}$$

where α is a small constant often set to 0.01[4, ch. 14]. We will also consider the hyperbolic tangent

$$f(z) = \tanh(z), \tag{36}$$

$$f'(z) = 1 - \tanh^2(z). \tag{37}$$

In regression problems, these activation functions are only used in the hidden layers. In the output layer, we want the activation function to return values that can be compared to the target $\mathbf{t}$. For regression problems, it is therefore common to use the identity matrix I_n, with n the number of nodes in the output layer, as activation function in the output layer. That means the output just becomes the weighted sum z_i^L.

In classification problems however, we want the output as a probability for the different categories, as seen in the discussion on logistic regression. An activation function that is often used in such problems is the softmax function. However, in this project, we only study a binary classification problem, which means there are only two categories. We therefore have to use a function returning values in the range from 0 to 1. Hence, the sigmoid function or the hyperbolic tangent are possible activation functions for the output layer.

We also have to choose cost function for evaluating the results. For regression cases, this can be the MSE in Equation 2. However, in the case of neural networks we do not have a design matrix. Therefore, the expression for the MSE is written

$$\mathcal{C}(\tilde{y}_i) = \frac{1}{n}\sum_{i=1}^{n}(\tilde{y}_i - y_i)^2, \tag{38}$$

$\tilde{y}_i$ are the predicted values from the FFNN. That means it is the a_i^L from the output layer.

In the case of classification problems, we use the cross entropy from Equation 10, but as function of the predicted values $\tilde{y}_i$ instead of β_i,

$$\mathcal{C}(\tilde{y}_i) = -\sum_{i=1}^{n}\left[y_i\tilde{y}_i - \log\left(1 + \exp\left(\tilde{y}_i\right)\right)\right], \tag{39}$$

Last, the regularization parameter λ can also be included in the cost function in question. This is done by adding a term of the form $\lambda \sum_{ij} w_{ij}^2$. The expression for the derivative of the cost function in Equation 28 then get an additional term and becomes

$$\frac{\partial \mathcal{C}(a_k^l)}{\partial w_{jk}^l} = \delta_j^l\, a_k^{l-1} + 2\lambda w_{jk}^l. \tag{40}$$

We thus get an additional term in the expression for updating the weights. The expression for updating the biases, on the other hand, does not change with the introduction of λ.

In the next section we describe how the gradient descent methods, logistic regression and the FFNN was implemented.

3 Method

In this section we start by describing how the gradient descent methods and the adaptive learning rate algorithms were implemented. Then we show how our feed-forward neural network was developed before we consider the implementation of logistic regression.

3.1 Gradient descent

We implement plain gradient descent instead of matrix inversion in OLS and Ridge regression. This is done by using Equation 14. The initial values of the parameters $\boldsymbol{\beta}$ are set to zero. Since we know the expressions for the derivatives of the cost functions in OLS and Ridge regression, from Equation 3 and 5, we simply use these to compute the gradient $\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta})$.

Next, we add momentum to the GD method as shown in equation 15 before we implement SGD as described in the Theory section. Then we also include the different algorithms for adaptive learning rates described in 16, 17 and 18. In each case we initialize the accumulation variables as $\mathbf{r} = 0$ and $\mathbf{s} = 0$. We set the different constants to $\rho = \rho_1 = 0.9$ and $\rho_2 = 0.999$. δ is a small value between 10^{-6} and 10^{-8}.

We will investigate the use of MSE as a cost function for different parameters and see how the parameters affect the results. The parameters we will study are the learning rate η, the regularization parameter λ, the number of epochs N_{epochs}, the number of minibatches N_B and finally the momentum parameter γ.

To find the lowest MSE, there are several methods. One solution could be to test each parameter 10 times. However, that means running the algorithm 10^5 times, which means our algorithm would take a very long time. Another, better and faster way is to perform a grid search for one parameter at a time, while setting the other parameters constant. After performing the grid search, we can see how every single parameter affects the results while the other parameters remain constant.

3.2 Assembling the FFNN

We design our FFNN to work similarly to SGD with momentum. The neural network has a set of layers with a number of nodes that communicate with the adjacent layer through a set of weights and biases. Like in the case of SGD with momentum we update these weights and biases, which serves as $\boldsymbol{\beta}$ in SGD, in order to minimize a cost function. We also include a term that serves as a momentum term, remembering what direction in the parameter space we updated the weights last.

Like with $\boldsymbol{\beta}$, the more nodes and layers we include, the more complex our model can be. Similarly to SGD we also shuffle the data randomly and use minibatches in order to try to avoid local minima. After we train the data over several epochs we are able to input a point and get a prediction as an output. As an example, if we train the FFNN to the Franke function, we will be able to input an x and a y coordinate and get a predicted altitude, z, as an output.

The way the neural network is trained is through two sets of algorithms. Firstly the feed-forward algorithm is used, in which the FFNN takes an input and feeds it through all the hidden layers by doing matrix multiplications with the weights and adding the biases. This is done until we reach the output layer. At this stage we have the output value of our neural network along with the output from each hidden layer. Then we are able to start from the output layer and go backwards to the input layer and adjust the weights and biases.

We create a FFNN with flexible learning rate, hyper-parameter λ, cost function, activation function in the hidden and output layers, initialization of weights, number of layers and number of nodes in each layer, in order to explore how we can use this network to create models of a given dataset.

There are several ways that we can initialize the weights in our neural network. The most direct one is to simply use a normal distribution with mean 0 and variance 1. A more careful approach to this initialization is to make the distribution thinner by decreasing the variance. Setting the variance to $2/n$, where n is the number of nodes in the previous layer, results in the He initialization [6].

Another way to initialize the weights is to use a uniform distribution. This is known as the Xavier distribution U [5], where the weights follow the distribution

$$w \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]. \tag{41}$$

Again n is the number of nodes in the previous layer. The biases are set to 0.01 in all cases.

When evaluating the FFNN for regression problems, we use the MSE, shown in Equation 2, and the

R^2 score defined as

$$R^2(\mathbf{z},\tilde{\mathbf{z}}) = 1 - \frac{\sum_{i=0}^{n-1}(z_i - \tilde{z}_i)^2}{\sum_{i=0}^{n-1}(z_i - \bar{z})^2}, \tag{42}$$

where $\mathbf{z}$ is the target value, $\tilde{\mathbf{z}}$ is the prediction value and $\bar{z} = \frac{1}{n}\Sigma_{i=0}^{n-1} z_i$. The reason why we also look at the R^2 score is because it gives us the relative performance with a scaled data.

3.3 Wisconsin Breast Cancer Data

We also test our FFNN on a classification problem. The data set we use is the Winsconsin Breast Cancer Data. This data set includes data on 569 patients that have taken 30 different measurements along with a binary output showing if the tumor is cancerous or not. We will use this data set to predict if the tumor is benign or malignant, represented by a 0 or a 1 respectively. We will determine the quality of our model by using the accuracy score which measures in percent how many correct predictions we have made;

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{43}$$

where I is the indicator function, t_i represents the target, y_i is the outputs from the FFNN and n is the number of targets.

Last, we want to perform logistic regression on the Wisconsin Breast Cancer data as well, and compare the results to those obtained with the neural network. We noticed that instead of implementing logistic regression separately, we could use the same code as for the FFNN. The only difference is that when performing logistic regression, we do not use any hidden layers, only the input and output layer. The activation function and cost function in the output layer is now the sigmoid function, Equation 6, and cross entropy, Equation 10, respectively.

In order to avoid exploding gradients, meaning the gradients diverges in the back propagation step giving overflow in the computations, we scaled the data when studying logistic regression. This was done by dividing all values by the maximum value in the data set. Thus, all components in the data set had values between 0 and 1.

In the next section we present and discuss the results from the various analysis performed in this project.

4 Results & Discussion

4.1 Gradient Descent

In the analysis of the gradient descent methods we used a one dimensional second order polynomial, on the form $f(x) = \beta_0 + \beta_1 x + \beta_2 x^2$, as the data set. We also added some stochastic noise. The design matrix X therefore consists of one column with only 1, since we keep the intercept, one for x, one for x^2 and so on, depending on the order of the polynomial used. The values for x are taken randomly from a uniform distribution.

We first performed OLS regression with plain GD and a fixed learning rate of 0.01. In Figure 1 we see that the model resembles the data points well. We will later discuss this result in term of the MSE.

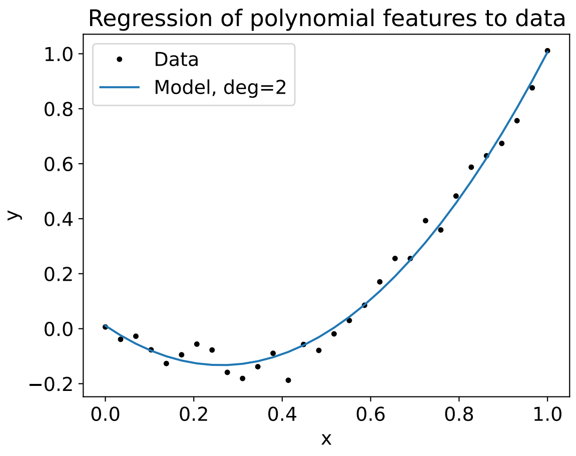

Figure 1: Plot showing how the model from OLS with plain gradient descent follows the data points. A constant learning rate 0.01 was used. To obtain these results, we needed 424505 iterations.

Figure 2 shows how the parameters β_i vary with the number of epochs. We see that the parameters in general converge faster, meaning the curves become flat for a lower number of epochs, in the case with momentum. The momentum serves as a sort of friction, removing a percentage of the size of the previous step we took (recall equation 15). This means that we are approaching the optimal parameters in a more careful way, leading to a faster convergence rate. We can see a visualisation of this in figure 2 where the model with momentum has a much sharper peak, which is due to the momentum term correcting the coefficient by slowing down the evolution of the coefficient.

We then studied how the different parameters and adaptive learning rate algorithms affected the results. This is shown in Table 1 and 2. We have here set some base values for the various parameters while changing one at a time.

In Table 1 we see, as expected, that the MSE decreases for a larger number of iterations. In addition we can see the effect of varying the learning rate and momentum. We can see that for smaller learning rates the MSE increases. This makes sense since the learning rate is essentially the size of each step our algorithm takes; if the step size is too small it will not converge. We also see that for learning rate $\eta = 1$ we get that the MSE is infinite. In this case we observe the opposite in the case of very small learning rate; when the learning rate is too large, the size of each step our algorithm takes is so large that our model explodes, essentially diverging.

In order to increase the convergence rate of our algorithm we can introduce momentum. In table 1 we can see the how different values of the momentum parameter γ, given in terms of a percentage, affects the MSE. Essentially this serves as a friction to our algorithm, making sure that we do not take too large steps when trying to minimize the cost function. We can see that with $\gamma = 1$ we get an MSE of 0.0148 which is 85% lower than the MSE of 0.105 we get with $\gamma = 0$.

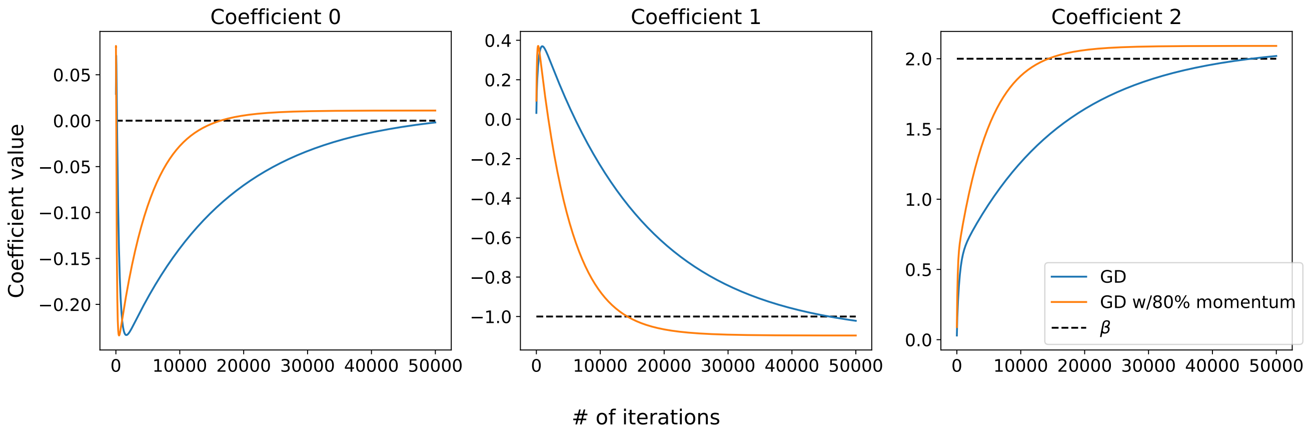

Figure 2: Plots showing how the parameters β_i evolves with number of iterations when performing OLS regression on a second order polynomial with gradient descent with and without momentum. The learning rate had a constant value 0.01.

Pol. deg.		Iterations		Learning rate		Momentum	
Value	MSE	Value	MSE	Value	MSE	Value	MSE
1	0.112	10^1	0.129	10^{-5}	0.126	0 %	0.105
2	0.105	10^2	0.126	10^{-4}	0.105	20 %	0.100
3	0.0995	10^3	0.105	10^{-3}	0.0543	40 %	0.0942
4	0.0954	10^4	0.0543	10^{-2}	0.0115	60 %	0.0847
5	0.0922	10^5	0.0115	10^{-1}	0.00451	80 %	0.0691
6	0.0896	10^6	0.00451	1	inf	100 %	0.0148

Table 1: Gradient descent varying each parameter. We give our parameters some base values, $\{n_{order} = 2, \lambda = 0,$ Max iterations $= 1000, \eta = 0.0001, \gamma = 0\}$, and for each column in the table we only change that parameter, leaving the other parameters as the base value.

Adaptive method		Ridge parameter		Batch size		Epochs	
Method	MSE	Value	MSE	Value	MSE	Value	MSE
None (constant)	0.105	10^{-15}	0.105	3	0.0645	10^1	0.129
AdaGrad	0.124	10^{-10}	0.105	5	0.0843	10^2	0.128
RMSProp	0.0722	10^{-5}	0.105	15	0.118	10^3	0.118
Adam	0.0762	1	0.106	25	0.127	10^4	0.0741

Table 2: Stochastic gradient descent, varying 'Ridge parameters' while keeping the other parameters as the base parameters as in 1. When we do not vary the batch size we let the batch size be the size of the dataset, meaning we do not divide the data into minibatches. The base number of epochs is 1000.

In table 2 we explore the result of varying the Ridge parameter λ, the batch size and the number of epochs we run over. We can see that we get little to no change when varying the Ridge parameter. This seems reasonable since the effect of λ is to suppress the parameters β_i which are less important. Here, we only use a second order polynomial which means all parameters β_i will contribute.

We observe that for larger batch sizes we get larger MSE. Since there are more steps taken, and therefore more times we update β_i, when considering a smaller batch size it makes sense that the MSE is smaller. Imagine we have a dataset with 100 entries. With a batch size of 25 we would update β_i 4 times for each epoch, while for a batch size of 5 we would update it 20 times. It should come as no surprise that increasing the number of epochs decreases the MSE, just like in the case of running more iterations.

In addition to exploring how different parameters affect our MSE we also explore how different approaches for adaptive learning rate affect the MSE. In our case we found that AdaGrad performed worse than with a constant learning rate. AdaGrad got a MSE of 0.124 which is 18% higher than the MSE with constant learning rate, which was 0.105. The adaptive method that performed best in our case was RMSProp which had a MSE of 0.0722 which is about 31% less than with constant learning rate.

Generally the Adam approach to adaptive learning rate performs the best, even though it fell just short of RMSProp in our test. Having this in mind we explore the co-dependence of the learning rate and the Ridge parameter with Adam adaptive learning rate as shown in figure 3. We have here used the cost function for Ridge regression.

Figure 3 shows that we get the best results for larger learning rates and smaller values of λ. This seems reasonable since we know, as discussed previously, that too small learning rates could mean we do not get convergence.

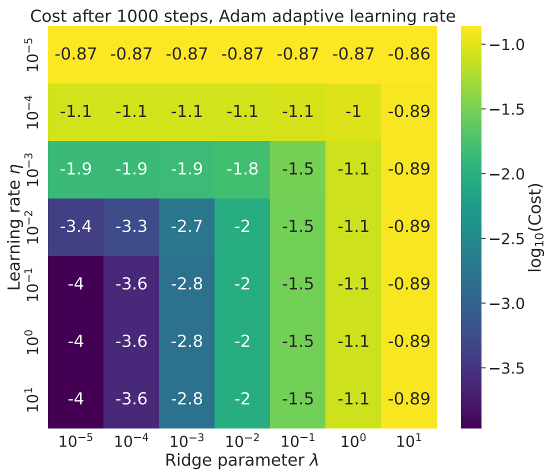

Figure 3: The co-dependence of the learning rate and the ridge parameter on the cost of our model after 1000 steps with Adam adaptive learning rate.

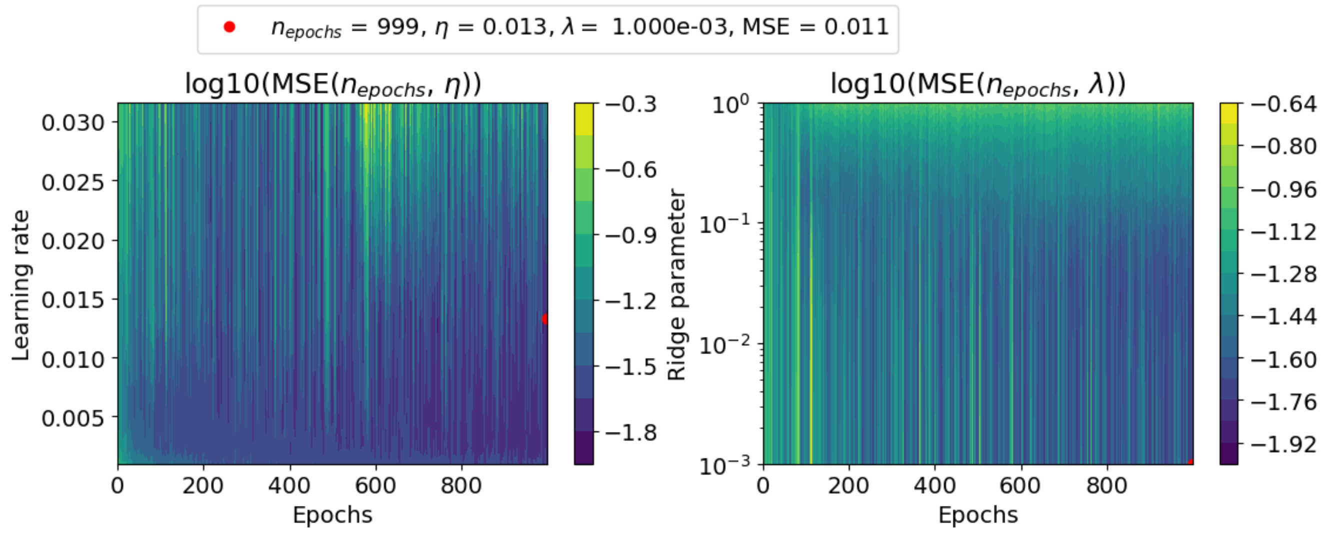

Figure 4: Plots of the MSE for various values of learning rate η, momentum parameter λ and number of epochs. We found the lowest MSE to be 0.011. The corresponding values for the parameters are shown above the plots.

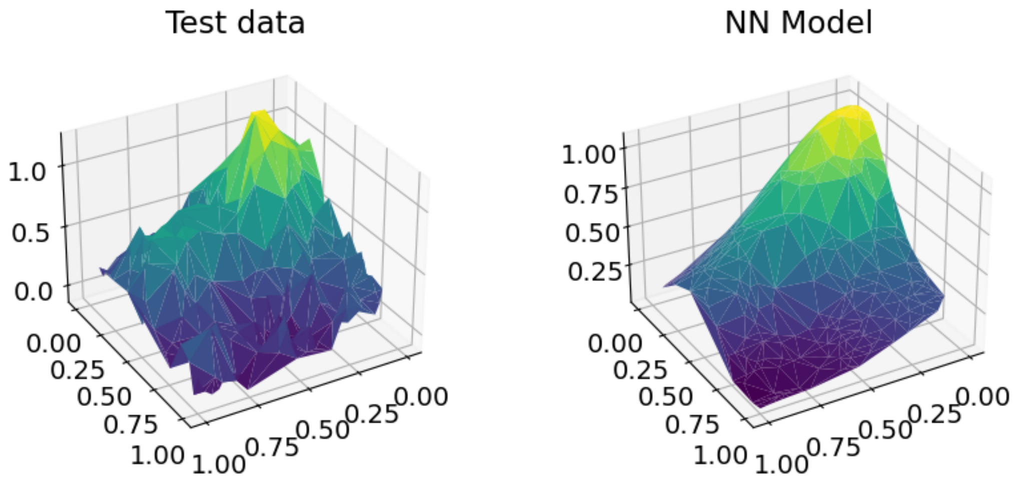

Figure 5: Test data of the Franke function (left) and our neural network model that gave the smallest MSE with parameters; $n_{epochs} = 999$, $\eta = 0.013$ and $\lambda = 10^{-3}$.

4.2 Feed Forward Neural Network

We then move on to discuss the results from our FFNN. In order to put these results into perspective, we perform a simple OLS regression on the Franke function with 5 polynomial degrees. From this we get an MSE of 0.014 and an R^2 score of 0.8423.

We first run our FFNN with the Franke function as our data set. We would like to see how our neural network fares against OLS for different parameters and as a function of the number of epochs. Here, the sigmoid function has been used as activation function for all hidden layers. For the output layer we use the identity matrix as activation function. The reason is that we here work with a regression problem and therefore want the output to be real values not values within a particular range. We use 3 hidden layers with 50, 40, 30 nodes in each layer respectively. We initialize the weights with a standard normal distribution, meaning with mean zero and variance one.

We run our neural network over a grid of η and λ values and record the value from the cost function (MSE) for each epoch. The result is a grid of MSE values for each combination of epochs, η and λ.

In figure 4 we plot the MSE for increasing number of epochs ran and with different values for the learning rate (left figure) and with different values for λ (right figure). We can see that for larger learning rates there is much more noise in the MSE across the number of epochs ran. This is because our model can overshoot the global minimum with a learning rate that is too high. With much smaller learning rates there is significantly less noise, which is an indication that we are not overshooting the minimum, but rather slowly approaching it with each epoch. The model with the lowest MSE is found at $n_{epochs} = 999$ with $\eta = 0.013$

From the right plot in figure 4 we can see that our model performs best for lower λ. For values close to 1 there is a dramatic increase in the MSE. The smallest MSE is found at $\lambda = 10^{-3}$, $n_{epochs} = 999$ meaning that to get the best model we should keep λ very small while having the number of epochs in the order of 10^3. The lowest MSE found in the network is 0.011 which is about 27% lower than for OLS.

We also evaluated the results by considering the R^2 score. The maximum value was found to be 0.863 for $\eta = 0.013$, $\lambda = 10^{-3}$ and 735 epochs. That means the R^2 score is 2.5% higher for the FFNN than for OLS.

We plot our test data and our neural network model fitted to the test data in figure 5. We can see that the model does a decent job at fitting, as the model closely resembles the data but with no noise. This model is fitted with the optimal parameters found by the MSE grid search ($n_{epochs} = 999$, $\eta = 0.013$, $\lambda = 10^{-3}$).

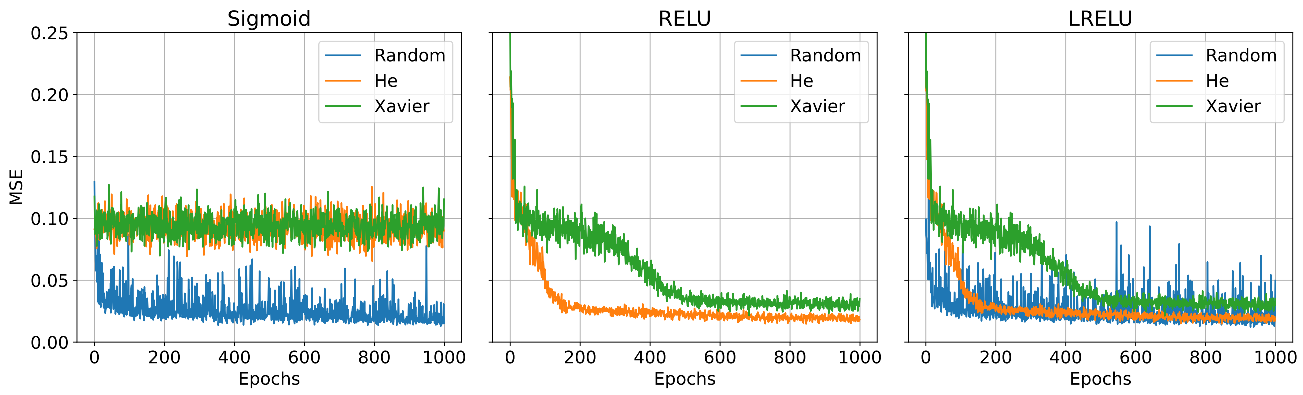

Figure 6: MSE as a function of epochs for different activation functions for the hidden layer with different initializations. The parameters used to get these are $\eta = 0.01$, $\gamma = 10^{-4}$ with 3 hidden layers with 50, 40, 30 nodes.

	Sigmoid	RELU	leaky-RELU
Random	0.013	-	0.0121
He	0.0654	0.0143	0.0143
Xavier	0.0698	0.023	0.023

Table 3: The minimum MSE reached for each activation function and initialization.

In figure 6 we show the result of using different activation functions and initializations on the MSE. We summarize the minimum MSE for each combination in table 3. We can see that in general, the sigmoid activation function results in more noisy MSE as a function of epochs. The same is true for leaky-RELU with random and Xavier initialization. Looking past the noise, we can see that the convergence rate with the random initialization seems to be the fastest among the options. He initialization with both RELU and leaky-RELU results in a more stable MSE and quickly converges after about 200 epochs. With the parameters used in this case we were not able to get a MSE using RELU with random initialization as the backpropagation algorithm resulted in exploding gradient. Even though the MSE is very noisy in the leaky-RELU case with random initializaton, this still results in the smallest MSE of 0.0121, which is about 15% less than what we get with OLS with 5 orders.

4.3 Wisconsin Breast Cancer data

We are now ready to test our neural network on a classification problem with a binary data set. For our choice of data set we use the Wisconsin Breast Cancer data described earlier. Our task now is to make a combination of parameters for our neural network that works well with this dataset.

For our choice of nodes and number of layers we use two hidden layers with 100 nodes in each. Since the dataset has 30 input values for each entry we have to make sure that our neural network is "smart enough to understand" the correlations between these inputs and the target. As an analogy one can think of the nodes and number of layers as the number of polynomial degrees included in OLS. The more polynomials we include, the better fit we will get, and the more complex our dataset is the more polynomial degrees it would be beneficial to include. This is also true for the number of layers and number of nodes in each layer.

We split the data set into a test and training set, keeping 1/4 of the data for testing. For the learning rate we select the value $\eta = 10^{-4}$. We tested different values for the various parameters and ended up with the following giving the best results.

For tuning the learning rate we used the Adam algorithm. For the output layer with used the hyperbolic tangent as activation function and for the hidden layers we used the sigmoid function. The weights were initialized with a normal distribution with mean 0 and variance 1. Cross entropy was used as cost function. With this setup, by training the network for only 10 epochs, we got an accuracy score of 96%.

In order to understand how accurate our neural network is on the Wisconsin Breast Cancer dataset, we look at how many correct predictions it makes. This is summarized in figure 7 where we can see how many times our network predicts a 1 and is correct and how many times it guesses a 0 and is correct. By proxy, the figure also shows how many times it gets it wrong. This is called a confusion matrix. We can see that among the cases where the true value is 0 we guess 93% correct, while in the cases where the true value is 1 we get an accuracy of 98%.

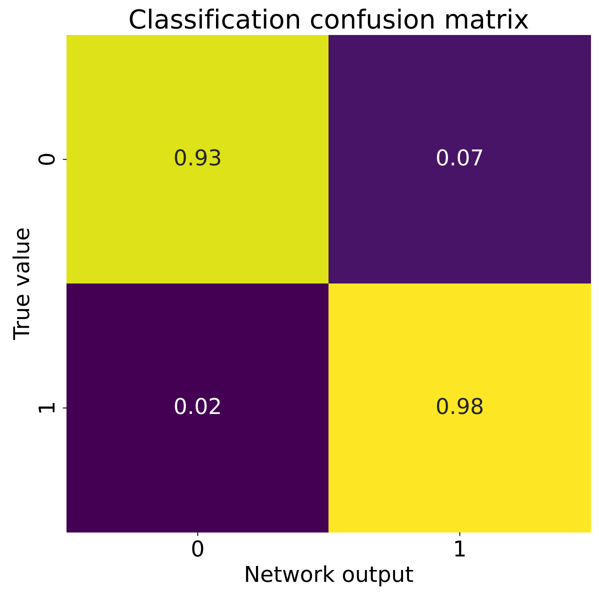

Figure 7: Confusion matrix showing as percentages the amount of times we classify correct and wrong for both the positive, 1, and negative, 0, category. These results were obtained with the neural network.

4.4 Logistic regression

Last, we also analyse the Wisconsin Breast Cancer data using logistic regression. In this case we do not use and hidden layers, meaning our neural network simply consists of one weight matrix and one set of biases that takes our input directly to the output layer. We used a learning rate $\eta = 10^{-4}$, cross entropy as cost function and the sigmoid function as activation function. Compared to the FFNN, we now had to use 1000 iterations to get the best results. We then got an accuracy of 90%.

The confusion matrix for this case is plotted in figure 8. We can see that in this case we get a lot more incorrect guesses, as the accuracy of guessing 0 when the true value is 0 is 76%. In other words, we have discovered that we get better results with our neural network than for logistic regression.

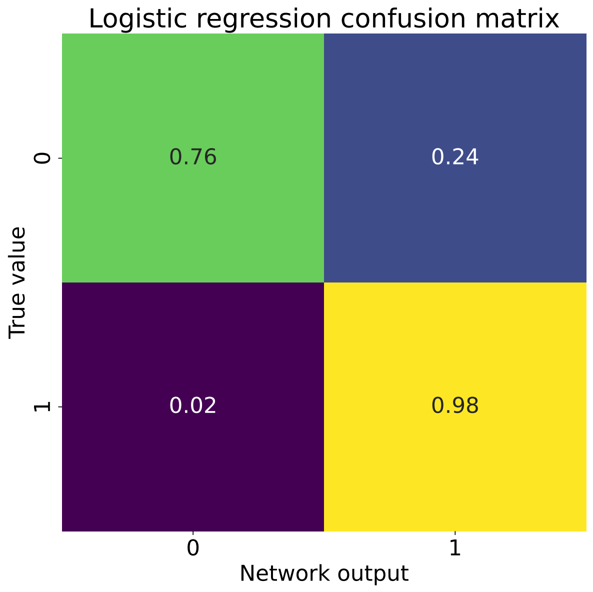

Figure 8: Confusion matrix in the case of logistic regression, showing the amount of times we classify correct and wrong for both the positive, 1, and negative, 0, category. These results were obtained with logistic regression.

5 Conclusion

We have found that a feed-forward neural network is a sensible approach to both linear and logistic regression. With several hyper-parameters and a free choice of cost functions and activation functions we get a tool for fine tuning a model to a given data set. We have seen that with the right setup we can get a model that beats OLS. When it comes to logistic regression we have seen that we can get a neural network that reaches an accuracy of up to 95.8%. This shows that with the correct approach a neural network could easily be used in medical applications to quickly predict if a patient has cancer or not given a set of measurements.

We used the FFNN to fit the Franke function and compared the results to the ones obtained with OLS regression. We found that the MSE for the FFNN was 27% lower than the MSE for OLS with 999 epochs, learning rate $\eta = 0.013$ and Ridge parameter $\lambda = 10^{-3}$. We used 3 hidden layers with where the first, second and third hidden layer had 50, 40 and 30 nodes respectively. The weights were initialized with a normal distribution with mean zero and variance one while all biases were set to 0.01. The activation function for the hidden layers was sigmoid while for the output layer we just used the identity matrix.

We also used our FFNN on a classification problem using the Wisconsin Breast Cancer data as the data set. We here compared the results to those obtained with logistic regression. Again we got best results with our FFNN which got and accuracy of 96% compared to 90% for logistic regression. These results were achieved using a sigmoid activation function for the hidden layers with 2 layers containing 100 nodes. We used the hyperbolic tangent as the activation function for the output layer and the Cross-entropy cost function. The adaptive learning rate used was Adam, with an initial learning rate of 10^{-4}. The FFNN was trained for 10 epochs.

For random initialization of the weights, we found that the neural network tends to explode. A better approach to initialization is the He initialization which uses a sharper normal distribution or a Xavier initialization which uses a uniform distribution.

We have seen that we need to fine tune the learning rate η and Ridge parameter λ, to get a combination that can give a model with an MSE less than OLS.

When it comes to activation functions, we found that the sigmoid function and the hyperbolic tangent performs very similarly which makes sense given their similar shapes.

We have seen that there are many parameters that has to be tuned in order to obtain the best possible results with our FFNN. This can be seen as a disadvantage with neural networks compared to simpler methods like OLS and logistic regression. However, we have also seen that a neural network can outperform both OLS and logistic regression if the parameters are chosen and tuned in an appropriate way.

References

[1] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. *Breast Cancer Wisconsin (Diagnostic) Data Set*: `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29`

[2] Aannerud, S., Børvind, M. B., Fremstad, D. $\hat{H}$., Wetzel, A. (2022). Deep learning methods: *Regression Analysis and Resampling Methods*: Oslo, Norway.

[3] Hjorth-Jensen, M. (2022). Project 2: *Machine Learning*: `https://compphysics.github.io/MachineLearning/doc/Projects/2022/Project2/pdf/Project2.pdf`

[4] Hjorth-Jensen, M. (2021). Deep learning methods: *Applied Data Analysis and Machine Learning Resampling Methods*: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html`

[5] Glorot, X., Bengio, Y. (2010) Weight initialization: *Understanding the difficulty of training deep feedforward neural networks* : `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`

[6] He, K., Zhang, X., Ren, S. (2015) Weight initialization: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*: `https://arxiv.org/pdf/1502.01852.pdf`

[7] Ian Goodfellow and Yoshua Bengio and Aaron Courville. MIT Press, 2016. *Deep Learning*: `http://www.deeplearningbook.org`