Building a Neural Network

Justina Grønbekk,
Sophia Gaupp,
Mina Piasdatter Walderhaug

UNIVERSITETET
I OSLO

Project 2
of FYS-STK4155 – Applied Data Analysis and Machine Learning.
at the University of Oslo
Oslo, Norway

November 2024

Abstract

Machine learning models, particularly neural networks, have emerged as powerful tools for tackling complex classification and regression tasks, with promising applications in critical fields like medical diagnostics.
However, despite their potential, there is ongoing debate about the necessity and effectiveness of neural networks for tasks that can also be handled by simpler, analytical models such as Ordinary Least Squares (OLS) and Ridge regression.
To address this, we implemented both neural networks and traditional regression methods to evaluate their performance on tasks involving medical data, specifically in diagnosing breast cancer.
Our study showed that a finely tuned neural network provided high classification accuracy for breast cancer diagnosis, on par with or even better than logistic regression. Whereas for linear regression, simpler models were equally effective but required less computational power.
The selection of the right model can have implications of high importance especially in the medical field where time is a scarce resource. A correctly chosen, diagnostically valuable model can aid doctors in making timely, accurate diagnoses, ultimately enhancing patient care.

Contents

Abstract **ii**

1 Introduction **1**

2 Methods **3**

2.1 Neural Network 3

2.1.1 Different problems 4

2.1.2 Cost function 5

2.2 Linear regression methods 5

2.3 Logistic Regression 5

2.4 Gradient Descent 6

2.4.1 Stochastic gradient descent 7

2.4.2 Learning Rate 7

2.4.3 Batches and number of epochs 9

2.5 Activation functions 10

2.5.1 Sigmoid function 10

2.5.2 ReLU 10

2.5.3 Leaky ReLU 11

2.5.4 Softmax 11

2.6 Feed Forward 11

2.7 Back propagation 12

2.8 Data sets 13

2.8.1 Wisconsin Breast Cancer data set 13

2.9 Performance measure 14

2.9.1 MSE and R^2 14

2.9.2 Accuracy 14

2.9.3 Confusion Matrix 14

3 Results **16**

3.1 Linear regression 16
3.2 Classification of Wisconsin Breast Cancer data set 18
3.2.1 Classification using FFNN 18
3.2.2 Classification using Tensorflow 20
3.3 Logistic Regression 21
3.3.1 Generated dataset 21
3.3.2 Wisconsin Breast Cancer dataset 24

4 Discussion **25**
4.1 Linear regression 25
4.1.1 OLS and Ridge regression 25
4.1.2 Neural Network 25
4.2 Logistic regression 27
4.3 Application to Breast cancer dataset 27
4.3.1 Neural Network on the Wisconsin Breast Cancer dataset 27
4.4 FFNN vs Logistic regression 29
4.5 Conclusion 30

A **33**
A.1 GitHub Repository 33

Chapter 1

Introduction

Two of the most common topics in news medias today are artificial intelligence (AI) and Neural Networks (NNs). AI can be used to predict and automate the diagnostic process of many cancers (Zhang, Shi, and Wang 2023). The use of AI in medicine has ”potential to improve the diagnosis, prognosis, and quality of life of patients with various illnesses, not just cancer” (Zhang, Shi, and Wang 2023). Neural Networks can be used in a variety of fields, including health care, technology and education. Neural networks have recently come under some scrutiny, as it is difficult to understand how they really work and reach conclusions. NNs have the potential to save lives yet they draw their conclusions on factors largely unknown to us.

In this report we have implemented a Feed-Forward Neural Network with back propagation. We compare the training of one such network with other regression models, both linear and logistic, to hopefully gain a better understanding of the inner workings of a Neural Network and when to use what method. We hope to see that a neural network can be used to classify cancer, and to show that NNs are possible to use for diagnostics in general.

Firstly, we show linear regression with a second-degree polynomial, and compare this with a neural network. This is to find out if a simpler model is better or worse than a neural network in such cases. Then we perform a classification analysis with logistic regression and compare with a neural network on the Wisconsin Breast Cancer Data Set .

We found that a neural network can give the same results as established linear regression methods when given enough training. However, OLS and Ridge will be preferred for such cases. In contrast, for the case of logistic regression, a Neural Network is preferred, as it can yield slightly better results. We found that a neural network can have a high accuracy in classifying tumors as malignant or benign. And thus we find it to be effective in solving these problems.

In this report we first explain our methods, we then show our results and discuss their implications. Lastly, we have a conclusion of our findings.

Chapter 2

Methods

2.1 Neural Network

Neural networks (NNs) are computational models inspired by the structure and functioning of biological neural networks, particularly those in the human brain. They consist of interconnected units called neurons, organized into layers. Information flows from the input layer to a number of hidden layers, and finally out through the output layer. In this project, we developed a Feed-Forward Neural Network (FFNN) to address both classification and regression tasks.

Neural networks consist of layers of interconnected nodes (or ”neurons”), where each node in one layer is connected to every node in the subsequent layer. Here’s a step-by-step breakdown of how a neural network works:

Input Layer: The input layer receives the input data, which can be in the form of numerical values, images, or text (depending on the problem). Each input node represents one feature of the dataset. These inputs are then passed to the next layer, a hidden layer.

Hidden Layers: Between the input and output layers, there are one or more hidden layers. This is where the core computation happens. Each neuron in the hidden layer receives the output from all neurons in the previous layer. Each input value is weighted differently, meaning it is multiplied by a weight that represents the importance of that feature for the neuron. The weighted inputs are summed up along with a bias term, which allows the model more flexibility in adjusting the output. The result is then passed through an activation function, which introduces non-linearity to the model, allowing it to learn complex patterns.

The output of each layer can be written as $z = W * a + b$, where z is the output, W is the weight, a is the input for the layer and b is the bias.

Output Layer: The final layer (output layer) gives the prediction or classification, depending on the task. For a classification task, the output is often the probability of each class (using activation functions like softmax or sigmoid), while for regression tasks, the output is typically a continuous value (with a linear activation function).

Training a Neural Network consists of two main steps:

Feedforward Step: In the feedforward step, the input data is passed through the network, moving from the input layer through the hidden layers and finally to the output layer. The network makes predictions based on the current weights and biases. Weights and biases are chosen randomly at the beginning of this process. At the end of the feedforward step, the network generates predicted outputs, which are compared to the actual target values.

Backpropagation: The gradient of the cost function with respect to each weight and bias in the network is calculated. This gradient indicates how much the cost function will change with small changes in the weights and biases. The network then uses this information to update its weights and biases in a way that reduces the cost. This process is typically done using optimization algorithms like gradient descent. The goal is to adjust the weights and biases so that, during the next iteration, the cost function decreases, meaning the model's predictions are getting closer to the actual target values.

2.1.1 Different problems

In machine learning, problems are broadly categorized into regression and classification based on the nature of the target variable that needs to be predicted.

Regression problem: Regression involves predicting a continuous value. In these problems, the output (target variable) is a real number, such as predicting temperatures, stock prices, or the height of a person based on their age and weight. The goal is to model the relationship between the input features (independent variables) and the continuous target variable.

Classification problem: Classification involves predicting a discrete label or category. In these problems, the output is a class label, which could either be binary (e.g., 0 or 1) or multiclass (e.g., different species of plants). The goal is to assign the input data to one of several predefined categories.

2.1.2 Cost function

A cost function (also known as a loss function or objective function) is a mathematical function that quantifies how well a machine learning model's predictions match the actual data. In essence, it measures the error or difference between the predicted output from the model and the true target values. The goal of training a machine learning model is to minimize this cost function by adjusting the model's parameters (weights and biases in neural networks). We want to find a global minimum dependent on the optimal weights and biases of the neural network which subsequently minimizes the deviation between our model's prediction and the actual data. Depending on the problem that is to be solved, different cost functions are leveraged. For example, in a regression problem, you might use Mean Squared Error (MSE) as the cost function to measure the difference between the predicted and true values. In a classification problem, you might use cross-entropy loss. See Project 1 for detailed formulas and implementations of such cost functions.

2.2 Linear regression methods

We normally use OLS and Ridge regression to fit a continuous function. We explain these methods more in detail in Project 1.

2.3 Logistic Regression

Logistic regression is a method that extends linear regression to classification problems by defining a probability distribution for class membership rather than predicting continuous values (Goodfellow, Bengio, and Courville 2016). It is typically used for binary classification, distinguishing between two classes labeled as 0 and 1.

In linear regression, a normal distribution is often used to model real-valued outputs based on a mean that can take any real value. However, for logistic regression, the output must represent a probability, constrained to the interval [0, 1]. This is achieved by applying the logistic sigmoid function to the output of a linear function, which "squashes" the real-valued predictions into this range, allowing them to be interpreted as probabilities of class membership.

Unlike linear regression, logistic regression does not have a straightforward analytical solution for optimizing model parameters. Instead, parameters are found by maximizing the log-likelihood function, which indicates how well the model explains the observed data. This is typically achieved by minimizing the negative log-likelihood (often referred to as the cross-entropy loss) using gradient descent.

While both logistic regression and FFNNs are classification models, they differ significantly in complexity and architecture. Logistic regression is a simpler model suitable for binary classification tasks where data is approximately linearly separable. It consists of only an input and an output layer, using the sigmoid function as an activation. In contrast, FFNNs contain multiple layers and are better suited for complex, non-linear data patterns.

2.4 Gradient Descent

Gradient Descent, GD, is an optimization method used in neural network training (Goodfellow, Bengio, and Courville 2016). It is an algorithm that takes a basis in the gradient of a function to minimize the Cost-Function of different models. The gradient is the partial derivatives for functions where we have multiple inputs.

The basis of gradient descent is that you have a function

$$\boldsymbol{F(x)},\ \mathbf{x} = (\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n})$$

which decreases the fastest when $\mathbf{x}$ goes in the direction of the negative gradient $-\nabla\mathbf{F(x)}$ (Hjorth-Jensen 2024).

However the challenge is to avoid local minima as you want to find the global one. This can still prove a challenge, as we step-wise move through our function for different values of $\mathbf{x}$. Given the equation

$$x_{k+1} = x_k - \gamma_k \nabla F(x_k)$$

where $\gamma_k > 0$, and by choosing a sufficiently small γ_k we can ensure that our optimization is always moving to a minimum by the condition $F(x_{k+1}) \leq F(x_k)$.

The γ_k is often referred to as the learning rate, or the step length.

We have an initial guess for x_0, and compute to see if we reach a global minimum. However, GD is deterministic and we cannot always know if we have found a global minimum. Therefore it is useful to introduce some randomness.

2.4.1 Stochastic gradient descent

Stochastic gradient descent, SGD, is a common method to diminish the shortcomings of standard gradient descent (Hjorth-Jensen 2024). It is based on the fact that the cost function can almost always be written as a sum over n data points.

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta)$$

This means that the gradient also can be written as a sum over i gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

Randomness is added by taking the gradient on a subset of the data called minibatches, instead of on the entire dataset. The idea is to approximate the gradient by taking the sum over the data points in one minibatch at each gradient descent step. This makes the new gradient step

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \epsilon B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

where B_k is a minibatch.

2.4.2 Learning Rate

The Learning rate defines how large the step should be to account for the error in each step (Wei 2019). A high learning rate speeds up training but may reduce the final accuracy, whereas a lower learning rate extends the training time but offers the potential for higher accuracy.

Momentum The key idea behind momentum is to incorporate the past gradients into the current update, allowing the model to build up velocity in directions where gradients are consistently pointing in the same direction. The aim is to speed up and skip areas where no minimum is to be found and slow down and subsequently increasing accuracy when its more likely to find a minimum ultimately aiming to find the global minimum.

ADAgrad. ADAgrad stands for Adaptive Gradient Algorithm. The key idea behind AdaGrad is to modify the learning rate dynamically for each parameter. In this way the frequently updated parameters get smaller updates, while infrequently updated parameters get larger updates. This adjustment is particularly useful for problems with sparse data or features, as it helps the model make larger updates in directions that require more attention.

Mathematically, AdaGrad scales the learning rate η by dividing it by the square root of the sum of the squares of all previous gradients for each parameter:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

Where:

- θ_t represents the parameters at step t,
- G_t is the sum of the squares of past gradients for each parameter,
- g_t is the gradient at step t,
- ϵ is a small constant added to avoid division by zero.

This adjustment ensures that large gradients in the early stages of training result in smaller steps as learning progresses, preventing overshooting and improving convergence. However, a downside of AdaGrad is that it can cause the learning rate to decay too aggressively, potentially leading to slow convergence later in training.

RMSprop. Root Mean Square Propagation (RMSprop) is designed to resolve the issue of AdaGrad's aggressive learning rate decay. In RMSprop, instead of accumulating all past squared gradients, we use an exponentially decaying average of past squared gradients. This allows the algorithm to maintain a more stable and adaptive learning rate throughout training, preventing the learning rate from shrinking too much.

The update rule for RMSprop is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

Where:

- $E[g^2]_t$ is the exponentially weighted moving average of the squared gradients,
- η is the learning rate.

Adam. The Adaptive Moment Estimation (Adam) optimizer combines the benefits of both AdaGrad and RMSprop by maintaining separate adaptive learning rates for each parameter and incorporating momentum into the update process. Adam computes two different running averages: the first is for the gradient (similar to momentum), and the second is for the squared gradient (similar to RMSprop). The optimizer uses these averages to adjust the learning rates for each parameter adaptively.

The update rules for Adam are given by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

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

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Where:

- m_t and v_t are the running averages of the gradient and squared gradient, respectively,
- β_1 and β_2 are hyperparameters controlling the exponential decay rates for these moving averages, typically set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$,
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected versions of the moment estimates.

Adam adapts the learning rate for each parameter, taking into account both the magnitude and the frequency of the updates. This makes Adam highly effective in scenarios with noisy or sparse data and helps maintain a fast and stable convergence rate.

2.4.3 Batches and number of epochs

For large-scale applications, the training data can have an order of millions of examples, and then it can be a bit wasteful to compute the cost function over all the data points we have (Hjorth-Jensen 2024). It is then common practice to split up the training data in batches. These batches (a selection of data points) is then used to update the parameter before moving on to a new batch.

In SGD, it is usually assumed that you use mini batches. Mini-batches are smaller and more manageable, and can be viewed as the splitting of one batch. An iteration over the number of mini batches is commonly referred to as an epoch. It is then typical to choose a number of epochs, and iterate over the mini batches for each epoch.

2.5 Activation functions

An activation function is there to calculate the output of a node given an input and its weight. Based on their distinct characteristics different activation functions influence how a model learns and performs.

2.5.1 Sigmoid function

The sigmoid function follows the logistic function and is mathematically defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} = 1 - \sigma(-x).$$

It has a range of 0 to 1, making it particularly useful when modeling probabilities, as the output can be interpreted as the likelihood of a binary outcome. The function is smooth and differentiable. However, one of its major drawbacks is the vanishing gradient problem: for very large or very small input values, the gradient approaches zero, slowing down learning in deep networks. This problem arises because the sigmoid function saturates at both extremes (close to 0 or 1), making weight updates through gradient descent inefficient. Despite this, it remains a standard choice in the output layer for binary classification tasks, where its probabilistic output is essential.

2.5.2 ReLU

ReLU stands for 'rectified linear unit' and is mathematically defined as:

$$f(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}.$$

It ranges from 0 to infinity. This function returns the input directly if it is positive; otherwise, it returns zero. With ReLU a scarce activation is achieved with only about 50% of hidden nodes being activated (not equal zero)

The key advantages of ReLU are its computational efficiency and its ability to mitigate the vanishing gradient problem, which can slow down learning in deep networks. Unlike the sigmoid function, ReLU does not saturate for positive values, allowing for faster convergence in deep models. This property makes it particularly useful in large, deep neural networks.

However, ReLU is not without its drawbacks. One common issue is the dying ReLU problem, where neurons can become inactive during training if they output zero for all inputs, causing them to stop learning. This occurs when the input to a ReLU neuron is negative for all training examples. To address this, variants such as Leaky ReLU have been introduced.

2.5.3 Leaky ReLU

Leaky ReLU is a variant of ReLU, mathematically described as:

$$f(x) = \begin{cases} 0.01x & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases}$$

which allow small negative values when the input is less than zero, keeping the neuron active and allowing for continued gradient flow.

2.5.4 Softmax

The softmax function is used in classification tasks where multiple classes are involved. It transforms the raw output of a model into a probability distribution across several classes, making it particularly suited for multi-class classification problems. Given a vector of real-valued scores, the softmax function calculates the exponential of each score, normalizing them so they sum to one. This results in each value representing the probability that the input belongs to a specific class.

In mathematical terms, for an output vector z of length k, the softmax function for each element z_i is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

where e^{zi} represents the exponentiated score for class i, and the denominator is the sum of exponentiated scores for all classes.

2.6 Feed Forward

A feed-forward neural network (FFNN) is a simple Artificial Neural Network (ANN) (Hjorth-Jensen 2024). In a FFNN all information is passed forward through nodes in layers, the value of each node being calculated using an activation functions and weights. Weights are weighted edges with direction that are the connections between layers in a NN. They can be computed using the back propagation algorithm. Activation functions are different functions used to determine whether or not a single node will be activated.

An important part of a FFNN is the feed-forward loop. This is the process that computes the value for each node, feeding information forwards through the network. The value of each node is calculated using the activation function on the sum of the output from the previous node together with the weight and bias of the new edge and node. The value of a node in the input layer can be calculated as such:

$$\mathbf{a}_j^1 = \alpha(w_j x + bj)$$

Where α is the chosen activation function for the layer, w is the weight, x is the input and b is the bias.
The value a_j^l for a node in a hidden layer l, the value will be calculated by taking the sum of all nodes in the previous layer as input, and evaluating this input using an activation function.

$$\mathbf{a}_j^l = \alpha(\sum_{i=0}^{m}(w_{ij}a_i + b_j))$$

In this instance there are m nodes from the previous layer that has edges leading into a_j^l.

Each layer can use different activation functions. The activation function for each layer is dependent on the type of problem one is solving, and what yields the best results.

The final output will depend on the results of all the previous nodes that leads into the output layer. The nodes together with the chosen activation functions, weights and biases will therefore determine the output from the output nodes. The final output can thus be changed by tuning the different activation functions, weights, and biases as is most appropriate for the problem one is solving.

2.7 Back propagation

Back propagation is a method used for tuning weights and biases in a FFNN according to an output error. In broad strokes back propagation can be described as such:
Compute the output of the NN using the feed-forward loop and random weights and biases. Then calculate the cost for each node in each layer, starting with the output nodes of the last layer and work backwards to the input nodes in the first layer. Using this we recalculate the biases and weights in order to minimize the cost of each node. Repeat until the desired cost is reached.
To minimize the error, we need to calculate the derivative of the cost function with respect to the weights and biases. We do this by combining different derivatives and using the chain rule (Hjorth-Jensen 2024, weekly exercise 43).

To find the derivative of the cost function with respect to the weights, use the chain rule on the expression:

$$\frac{dC}{dW} = \frac{dC}{da}\frac{da}{dz}\frac{dz}{dW}$$

Here, z is an intermediary value defined as the sum of the output from the last node and the weight and bias. a is the activation function applied to z.
To find the derivative of the cost function with respect to the bias, use the chain rule on the expression:

$$\frac{dC}{db} = \frac{dC}{da}\frac{da}{dz}\frac{dz}{db}$$

The expression $\frac{dC}{da}\frac{da}{dz}$ can be reused for both calculations, and needs only be calculated once for each layer. For the output layer $\frac{dC}{da}$ is calculated as the derivative of the cost function with respect to the final activation function. $\frac{dC}{da}\frac{da}{dz}$ can thus be calculated as such:

$$\frac{dC}{da}\frac{da}{dz} = \frac{dC}{dz} = costfunction'(a_{last\_layer}, target) * activationfunction'(z_{last\_layer})$$

For the general layer i $\frac{dC}{da}$ needs to be calculated using the chain rule and the z from the previously calculated layer.

$$\frac{dC}{da_i} = \frac{dC}{dz_{i+1}}\frac{dz_{i+1}}{da_i}$$

$\frac{dC}{da_i}\frac{da_i}{dz_i}$ is then further calculated as

$$\frac{dC}{da_i}\frac{da_i}{dz_i} = \frac{dC}{da_i} * activationfunction'(z_i)$$

When we have $\frac{dC}{da_i}\frac{da_i}{dz_i}$ we can then further calculate $\frac{dC}{dW_i}$ and $\frac{dC}{db_i}$ using $\frac{dz_i}{dW_i}$ and $\frac{dz_i}{db_i}$.

2.8 Data sets

All our models were assessed on generated data sets with standard values chosen to fit the models abilities. A standard number of 100 samples was employed. Random seeds were used in order to make our results reproducible.

2.8.1 Wisconsin Breast Cancer data set

To further evaluate our models, we use the Wisconsin Breast Cancer Dataset, a widely recognized dataset in medical research. This dataset contains a set of features computed from digitized images of fine needle aspirates of breast masses, which help distinguish between malignant and benign tumors. This dataset includes 569 instances, with each instance described by 30 numerical features (e.g., radius, texture, perimeter, area, smoothness), providing a comprehensive basis for classification tasks aimed at breast cancer diagnosis (Street, Wolberg, and Mangasarian 1993).

The use of this particular dataset allows us to benchmark our models against a well-documented dataset. We can then compare our results with previous studies done on these data. The dataset's real-world relevance makes it an ideal test bed to highlight the relevance of assessing our model accuracy and generalization in binary classification tasks.

2.9 Performance measure

2.9.1 MSE and R^2

See in-depth explanation of MSE and R^2 in Project 1.

2.9.2 Accuracy

Accuracy measures the proportion of correct predictions made by a model out of the total number of predictions. This metric is commonly used in classification tasks and provides a straightforward way to assess model effectiveness. Calculated as the ratio of correctly predicted instances to the total amount of instances, accuracy is expressed as:

$$Accuracy = \frac{NumberofCorrectPredictions}{TotalNumberofPredictions}$$

This measure allows us to quantify the success rate of our models and serves as a reliable benchmark for comparing different models. The models with a low accuracy can usually be dismissed outright. When several models have a high accuracy, it might also be necessary to look at the confusion matrices for the models to make a good comparison.

2.9.3 Confusion Matrix

A confusion matrix is a table that provides a detailed breakdown of a model's performance in classification tasks by displaying the true and predicted classifications for each class. It helps in identifying not only how often the model is correct (accuracy) but also the types of errors it makes. This matrix is especially useful when looking at model performance in medical use cases since mistakes in the different classes have widely different medical implications.

The confusion matrix consists of four main components in the case of binary classification:

True Positives (TP): Cases where the model correctly predicted the positive class.

True Negatives (TN): Cases where the model correctly predicted the negative class.

False Positives (FP): Cases where the model incorrectly predicted the positive class (also known as Type I error).

False Negatives (FN): Cases where the model incorrectly predicted the negative class (also known as Type II error).

Chapter 3

Results

3.1 Linear regression

MSE and R2.

	OLS	Ridge	NN (initial guess)	NN (trained)	NN (test data)
MSE	0.000	0.000	2.722	0.019	0.029
R2	1.000	0.999	-0.220	0.991	0.988

Table 3.1: The MSE and R2 of the different methods used for a linear regression problem. For OLS and Ridge, the MSE and R2 is evaluated for the test data. For NN the MSE and R2 is evaluated before and after training and then on the test data.

Network training.

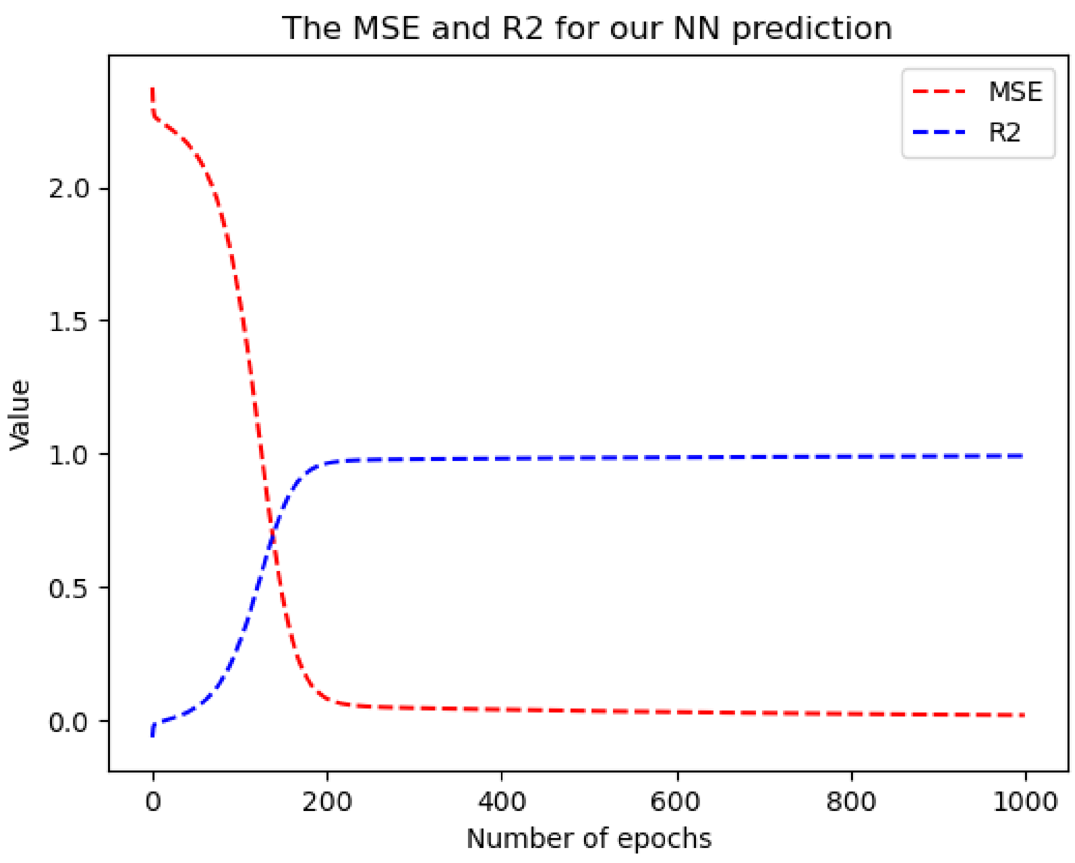

Figure 3.1: Plot of the MSE and R2 of our neural networks prediction using our training data. X-axis shows number of epochs. Y-axis shows value of MSE or R2. This is using Linear regression.

In Figure 3.1 we notice that the R2 converges to a value of 1 and the MSE converges to a value of 0 at around 200 epochs, when training our model.

3.2 Classification of Wisconsin Breast Cancer data set

3.2.1 Classification using FFNN

Number of hl	Number of nodes	Activation functions	Epochs	LR	Accuracy
1	10	Sigmoid, Sigmoid	1000	0.01	0.851
1	5	Sigmoid, Sigmoid	1000	0.01	0.807
1	10	Sigmoid, Sigmoid	1000	0.001	0.465
1	10	Sigmoid, Sigmoid	1000	0.1	0.947
1	10	Sigmoid, Sigmoid	100	0.01	0.465
1	10	Sigmoid, Sigmoid	2000	0.01	0.886
1	10	ReLU, Sigmoid	1000	0.01	0.807
1	10	ReLU, Sigmoid	1000	0.001	0.526
1	10	ReLU, Sigmoid	1000	0.1	0.930
1	10	Leaky ReLU, Sigmoid	1000	0.1	0.930
1	5	ReLU, Sigmoid	1000	0.1	0.965
1	5	Leaky ReLU, Sigmoid	1000	0.1	0.947
2	10, 10	Sigmoid, ReLU, Sigmoid,	1000	0.1	0.965
2	20, 20	Sigmoid, ReLU, Sigmoid,	1000	0.1	0.974
2	20, 20	Sigmoid, ReLU, Sigmoid,	1000	0.4	0.965
2	20, 20	ReLU, ReLU, Sigmoid,	1000	0.1	0.982
2	20, 20	ReLU, ReLU, Sigmoid,	1000	0.2	0.982
2	20, 20	ReLU, ReLU, Sigmoid,	1000	0.4	0.991

Table 3.2: A table showing the accuracy score for our FFNN on the Wisconsin Breast Cancer data set, tuned on different parameters. The parameters are as follows: Number of hidden layers, number of nodes in each hidden layer, activation functions, number of epochs, learning rate. The accuracy is calculated as is mentioned in Section 2.9.2. See below for the confusion matrices of the colored rows.

The highest accuracy score we got is an accuracy of 0.991. This accuracy is achieved when we have two hidden layers with 20 nodes each and the activation functions [ReLU, ReLU, Sigmoid], 1000 epochs and a learning rate of 0.4.

From the results presented here, we got seven models with a very high accuracy, higher than 0.95. The confusion matrix of some of these results (the rows highlighted in green) are presented below.

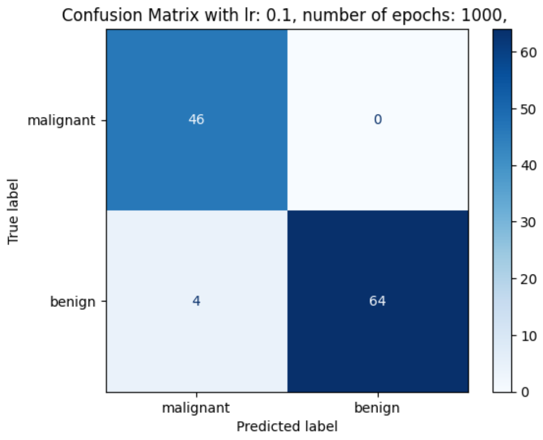

Figure 3.2: The confusion matrix of a FFNN model. The model has one hidden layer with 5 neurons, a learning rate of 0.1, the number of epochs = 1000 and the activation functions used are [ReLU, Sigmoid]. The accuracy of the model is 0.965.

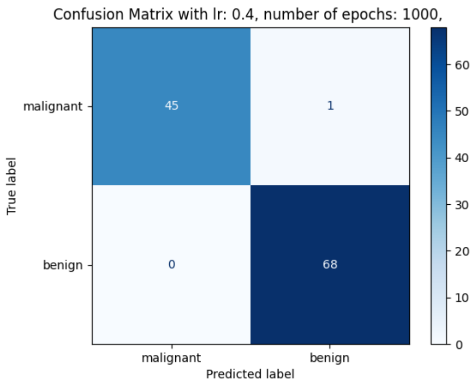

Figure 3.3: The confusion matrix of a FFNN model. The model has two hidden layer with 20 neurons each, a learning rate of 0.4, the number of epochs = 1000 and the activation functions used are [ReLU, ReLU, Sigmoid]. The accuracy of the model is 0.991.

Presented above are two confusion matrises. Figure 4.2 represents a FFNN with the following parameters: one hidden layer with 5 neurons, a learning rate of 0.1, the number of epochs = 1000 and the activation functions used are [ReLU, Sigmoid]. From Figure 4.2 the model correctly classifies 64 cases as benign and 46 cases as malignant. It also incorrectly classifies four cases as malignant, which should have been classified as benign (a false-positive). No cases are falsely classified as benign (false-negative).

Figure 4.3 shows the confusion matrix from a FFNN model with the following parameters: two hidden layer with 20 neurons each, a learning rate of 0.4, the number of epochs = 1000 and the activation functions used are [ReLU, ReLU, Sigmoid]. From Figure 4.3 the model correctly classifies 68 cases as benign and 45 cases as malignant. It also incorrectly classifies one case as benign (false-negative). No cases are incorrectly classified as malignant (false-positive).

3.2.2 Classification using Tensorflow

Number of hl	**Number of nodes**	**Activation functions**	**Epochs**	**LR**	**Accuracy**
1	10	sigmoid, sigmoid	1000	0.001	0.965
1	10	ReLU, sigmoid	1000	0.001	0.991
2	20, 20	ReLU, ReLU, Sigmoid,	1000	0.1	0.974

Table 3.3: A table showing the accuracy score using Tensorflow on the Wisconsin Breast Cancer data set, tuned on different parameters. The parameters are as follows: Number of hidden layers, number of nodes in each hidden layer, activation functions, number of epochs, learning rate. The accuracy is calculated as is mentioned in Section 2.9.2. See below for the confusion matrix of the colored row.

The highest accuracy score we achieved with Tensorflow was an accuracy of 0.991. This accuracy is achieved with one hidden layer with 10 nodes and the activation functions [ReLU, Sigmoid], 1000 epochs and a learning rate of 0.001.

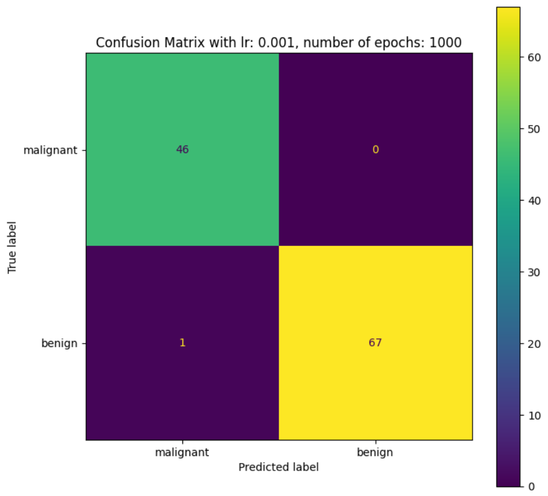

Figure 3.4: The confusion matrix of a FFNN model using Tensorflow. The model has one hidden layer with 10 neurons, a learning rate of 0.001, the number of epochs = 1000 and the activation functions used are [ReLU, Sigmoid]. The accuracy of the model is 0.991.

Figure 4.4 shows the confusion matrix for a FFNN model using Tensorflow with the following parameters: one hidden layer with 10 neurons, a learning rate of 0.001, the number of epochs = 1000 and the activation functions used are [ReLU, Sigmoid]. The model correctly classifies 67 cases as benign and 46 cases as malignant. It falsely classifies one case as malignant. No cases are falsely classified as benign.

3.3 Logistic Regression

We determined the performance of our Logistic Regression first with a generated dataset and then on the Wisconsin Breast Cancer data set.

3.3.1 Generated dataset

A generated dataset with a random seed of 666, 100 samples and two features was created. With a learning rate of 0.1 and 1000 epochs it yields an accuracy on the training set of 1 and an accuracy on the test set of 0.85.

Learning Rate.

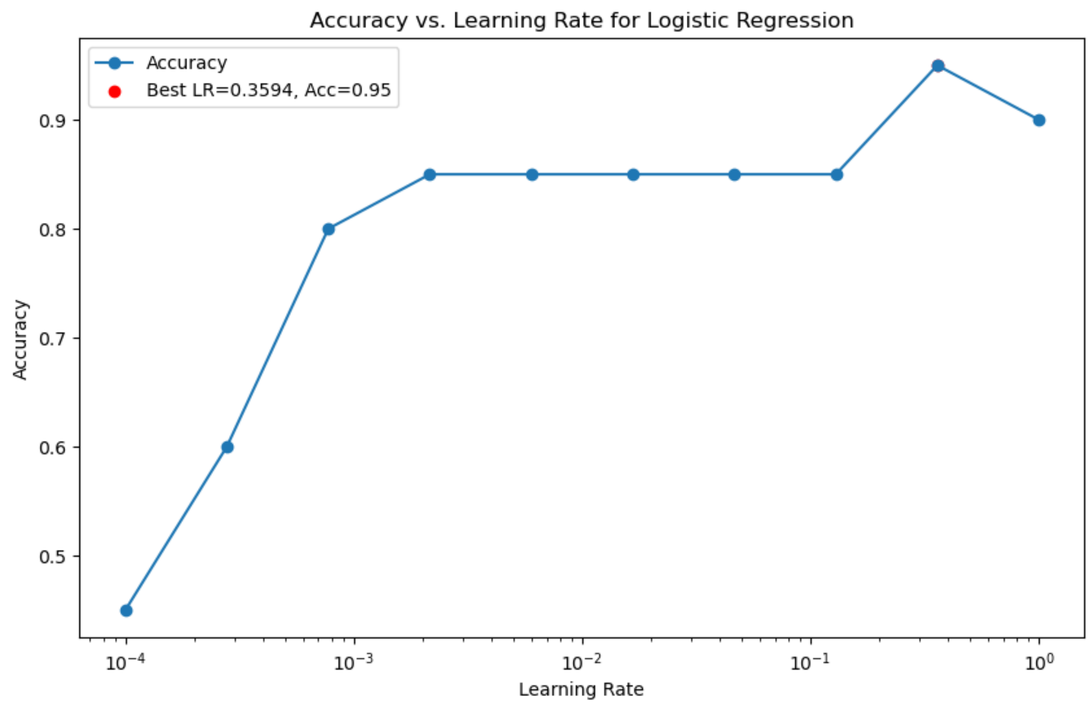

Figure 3.5: Applied method is Logistic Regression on test set. X-axis depicts the Learning Rate. Y-axis shows the Accuracy.

We can see the test set accuracy increasing with an increasing learning rate first very steeply then being quite stable. Accuracy increases to a peak of 0.95 at an optimal learning rate of 0.3594.

Number of epochs.

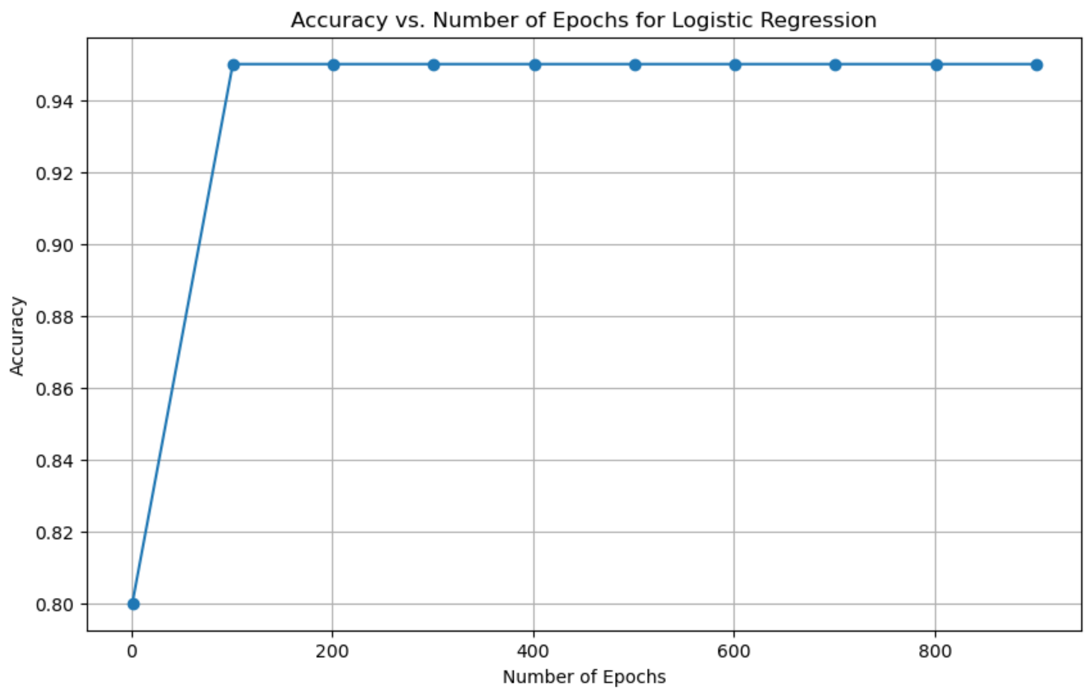

Figure 3.6: Applied method is Logistic Regression with an optimal learning rate of 0.3594. X-axis depicts number of epochs. Y-axis shows the Accuracy. Step size for increasing epochs is 100 starting with 1.

Accuracy increases from 0.8 to 0.95 from 1 to 100 epochs. After 100 epochs accuracy is very stable at a level of 0.95.

SciKit Learn.

With the SciKit Learn logistic regression Pedregosa et al. 2011 we get an accuracy of 1 with a data set generated in the same way and the same random seed.

3.3.2 Wisconsin Breast Cancer dataset

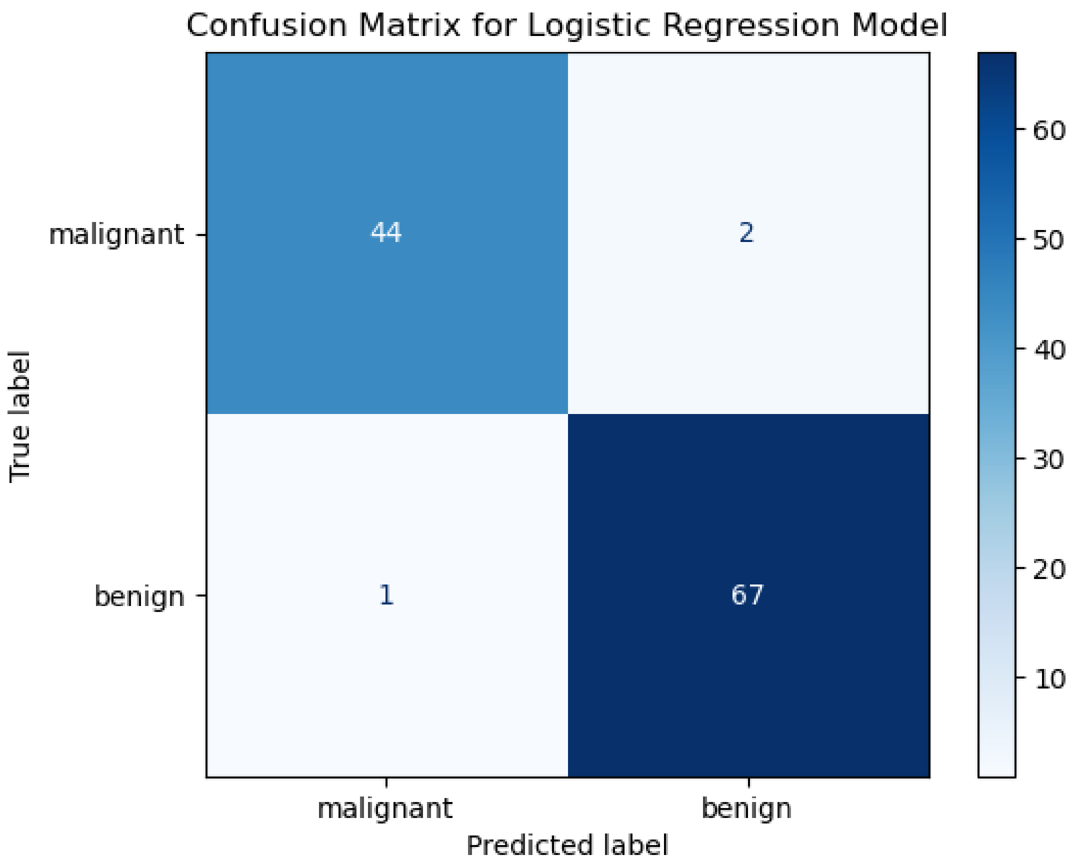

Figure 3.7: Confusion Matrix with applied method being Logistic Regression. Data set used is the Wisconsin Breast Cancer data set. 570 cases divided in training test split in a ratio of 8:2 results in a total number of 114 test cases distributed in the Confusion Matrix.

Applying our Logistic Regression to the Wisconsin Breast Cancer dataset yields a test accuracy of 0.9737. See the Confusion matrix above for a more in-detail distribution of the different cases to the different classes resulting in the whole composition of the matrix.

Chapter 4

Discussion

4.1 Linear regression

4.1.1 OLS and Ridge regression

In our implementation of OLS and Ridge regression, we trained our β values on a training set and then made a prediction using these values and our test data. We chose a simple second-degree polynomial to compare with a neural network to demonstrate the basic findings in these cases.

OLS. Our OLS clearly gives the best results for our choice of polynomial. From Table 4.1 we see that the MSE is zero and R2 is 1. This means that we have a perfect model, and this is not uncommon for a simple second-degree polynomial (with no added noise).

Ridge. We chose the optimal λ value of 0.1 for Ridge regression by plotting the MSE and R2 for different λ values. Using this value, we see in Table 4.1 that the MSE is zero and the R2 is 0.999. This is also a very good prediction.

This goes to show that our nice analytical solutions for linear regression using OLS and Ridge regression will yield a prediction that is ideal for our test data.

4.1.2 Neural Network

In our implementation for a FFNN for a linear regression problem, there are a few important details we looked at.

Weights and biases. For initializing our weights and biases, we normal distributed them both. We found that this gave better results in our training earlier on. If we did not normal distribute our biases, we had an even worse initial guess (or predicted β

values) for our FFNN.

Cost function. We chose to use MSE as our cost function for the back propagation. We decided this was the best choice as OLS and Ridge regression is based on the MSE as well.

Activation functions. We used the sigmoid function to evaluate our output for each hidden layer. However, it is very important to not use such a function for our last output layer. This is due to its nature giving us values to fit the range of a logistic regression problem. We therefore want a function that gives us all possible values as the last output. In linear regression, all values are considered valid. Therefore we had the identity function as the last activation function. This is just a function the returns the same output as input. Therefore we get all possible values.

Number of nodes and layers. The ideal number of nodes and layers is something you cannot necessarily predict before training your model. We therefore tested different number of nodes and layers to see what would give better result quicker. We always want to lessen the computational time and load for a better result. We saw that two hidden layers improved our prediction from one hidden layer for our particular problem. We also noted that increasing the number of layers further gave worse results. There was therefore a compromise between too few and too many layers. This same trend was seen for the number of nodes. However, it also depends on what number of nodes you have for each layer. The number did not have that much of an effect on the end result when in the first hidden layer as in the second hidden layer.

Regularization parameter λ. For Ridge regression you have the regularization parameter. We assessed the best choice of λ by plotting lamdas against the MSE. We then chose the value that gave the lowest MSE.

Learning rate. For the learning rate, it is important to choose a step-size that is neither too large or too small. We found that a value of 0.1 gives the desired result for MSE and R2 very quickly in our case and without needing much more training. Anything lower will cause longer training. And if it becomes too high, we see that the prediction will actually become much worse. It forces the NN away from the minimum.

Number of epochs. When choosing the number of epochs, it is important to have a number of iterations that can allow for a convergence of values (prevent underfitting). Also, it cannot be too high as this will result in overfitting. We found that a number of 1000 epochs gave the best results.

In our FFNN we saw that the initial guess gave very poor results. This makes sense as it does not have any analytical expression for the β values to assess. However, after training the model we notice that the FFNN will give improved MSE and R^2 values for our training dta. This means that a neural network can be used for linear regression as well. However, it is not an ideal choice for solving linear regression problems. Our simple example shows us that it clearly takes longer for the NN to get the same optimal values as OLS gets straight away. For small problems like this, the computational load is very low but in the case of large datasets, the training will take even longer and be computationally heavier. We can then say that OLS and Ridge would be preferred in such cases as it does not have the same computational load as a NN has.

4.2 Logistic regression

Considering Logistic Regression offers a more simple model architecture an accuracy of 0.95 when parameters such as learning rate are optimally chosen is quite good. Choosing the right learning rate is extremely important which becomes obvious when compared to the accuracy received with a default learning rate of 0.1.

4.3 Application to Breast cancer dataset

In medical applications, relying solely on accuracy as a performance metric is inadequate. The consequences of classification errors can have drastic implications in a clinical context. For example, failing to detect a tumor (a false negative) can lead to severe health consequences that are a result of a sick person not getting treatment because the disease was not correctly detected.

In our analysis of the Wisconsin Breast Cancer dataset using logistic regression, we identified two out of 114 cases incorrectly categorized as benign when they were actually malignant. Additionally, one case was misclassified as malignant despite being benign (a false positive).

While our logistic regression model demonstrated strong performance, the implications of these misclassifications underscore the necessity of carefully selecting the appropriate parameters and improving model performance. This approach is vital to maximize the effectiveness of such models in clinical settings while minimizing the risk of potentially fatal misclassifications.

4.3.1 Neural Network on the Wisconsin Breast Cancer dataset

Activation function

We tried to use ReLU as the last activation function, but realised soon that this does'nt work well. Several of the models we tried that had ReLU as the last activation function had results that classified data as 2 (instead of the expected 0 or 1). This makes no sense for a binary classification problem, as the output values necessarily needs to be binary. This result does make some sense, as the ReLU function is not built for binary classification. The ReLU function does not always return a number between 0 and 1 (as Sigmoid does). It would seem that because of this the ReLU function will sometimes return values larger than 1 even in binary classification problems, making it unfit to be the last function used for binary classification. As we round the output to the nearest number, some values where thus rounded to 2. We had this issue when using Tensorflow as well.

One problem we had with using ReLU, was that sometimes the results would classify all cases as malignant. This means that all output data was rounded down to zero, and that these models were useless. This again makes sense, as the ReLU function is known for having dead neurons. For these specific instances leaky ReLU would probably have been a better choice of activation function. In the best models however, we did not see an improvement in using leaky ReLU compared to using ReLU. In fact, some of the models had a lower accuracy than similar models using ReLU.

Of the activation functions we tried, the best models used a combination of sigmoid and ReLU. But as mentioned, we need to be careful when using ReLU as the last activation function. As sigmoid always only gives an output between 0 and 1, and ReLU does not, it is specifically suited to be the last activation function for binary classification. This is because we round the output to the nearest number, and when the output is between 0 and 1, it will always be rounded to either 0 or 1.

Learning rate

From our results, we saw the highest accuracy from a model with a learning rate of 0.4. A small accuracy of 0.001 yielded relatively low accuracy, and an even lower learning rate yielded lower accuracy still. Though our highest accuracy came from having a learning rate of 0.4, this might not be the best learning rate in all scenarios. For example, our most preferable confusion matrix came from a model with a learning rate of 0.1. In our experience, having a learning rate higher than 0.1 generally gave results equal to or worse than the results when the learning rate was 0.1. Thus, 0.1 seems to be the best learning rate for our classification problem, as with linear regression.

Number of Nodes and Layers

Having 2 hidden layers generally yielded an accuracy close to 0.9, with the best results stemming from having 20 nodes in each layer. When increasing the number of nodes to 30, the accuracy decreased again to 0.404. Models with fewer hidden layers could yield high accuracy scores up to 0.9 but generally had a lower accuracy score than the models with two hidden layers. When adding even more hidden layers, the accuracy again dwindled, leaving two hidden layers as the best number of hidden layers for our model.

Epochs

The number of epochs we found to give a desired accuracy was 1000. A higher number of epoch took more time to give a result, and thus was more computationally demanding, but did not necessarily yield a high accuracy. A smaller number of epochs generally gave a low accuracy, and thus gave us a worse model.

Accuracy vs. Confusion Matrix

It is very important in medicine to reduce the amount of false-negative diagnoses, to ensure that no one who needs treatment goes without. Therefore it is not necessarily the FFNN who gives the highest accuracy that is the best.

Using our own FFNN, the model that had the highest accuracy had an accuracy of 0.991. This model yielded only one false-benign classification, and no false-malignant cases. This is a pretty good model, but it does not mean that it is our best model.

We could only find one model using our own FFNN that had a high accuracy and zero false-benign classifications. It has, however, a lower accuracy than our most accurate model, at 0.965, and four false-malignant classifications. As it is important to have as few false-benign classifications as possible this might be our best model, even though other models had a higher accuracy.

Using Tensorflow, we found a model with zero false-benign cases, and only one false-malignant case. It had an accuracy of 0.991, the same as our most accurate model. As it has both a high accuracy and no false-benign cases, it seems to be the preferable model to use to solve this problem. It has a learning rate of 0.001, which was not found to be preferable with our models. The number of epochs = 1000, which is consistent with our findings that this was the best number of epochs. It only had one hidden layer, and not two as our best models had. Though our results were better with two hidden layers, this is not inconsistent with the results from our FFNN.

4.4 FFNN vs Logistic regression

Fine-tuning a FFNN by choosing all the optimal parameters such as activation function and numbers of nodes in each hidden layer proved to be challenging. The accuracy of 0.95 that was achieved through Logistic Regression was surpassed with eight different models. The highest accuracy was reached with a FFNN with two hidden layers with 20 nodes each, 1000 epochs, a learning rate of 0.4 and the activation functions [ReLU, ReLU, Sigmoid].

4.5 Conclusion

In this paper, we have made our own Feed Forward Neural Network and used it in various classification and regression analyses. The results from this has been compared to that of other models and methods.

Using our FFNN on a linear regression problem, we found that it was not the best model for this problem. Instead we found that using OLS or Ridge would be preferable. This is because of the computational load that a large amount of training for a NN requires. To get a comparable result from a NN as with OLS or Ridge, the network would have to go through substantial amounts of training.

We used our FFNN on the Wisconsin Breast Cancer dataset, and compared it to our logistic regression model and Python's Tensorflow model. We found that our FFNN model underperformed compared to the model from Python's Tensorflow, but held up against our logistic regression model. The results were rather similar, with logistic regression being ever so slightly worse. All methods worked well on the classification problem, but using Tensorflow seems to be the best solution in this instance.

Tuning our FFNN to find the best parameters has been strenuous and not always intuitive. From this process, we have found that correctly choosing parameters is vital for the success of the model. Even though a FFNN can be tuned to perfection, it is not always worth the workload and computational cost compared to simpler solutions. For more complex tasks, a NN can be the best solution.

Implications

Tuning a neural network is time consuming, and the computational load can be quite large. Using a neural network to solve simple problems that can be solved by other means is therefore not always practical. There are however some problems that can only be solved using NNs.

As the choosing of parameters is not always intuitive, it is difficult to know exactly how a NN works. This means that the process by which a NN makes decisions is largely unknown and the results it outputs can have unintended negative consequences. The NN will be affected by the data on which it trains, and is thus not unencumbered by human flaws. Errors in the way the data is handled or classified can lead to said errors being replicated throughout the network, with no one being the wiser.

Future directions

Neural Networks are a vital piece of the current paradigm shift in platforms like health care and technology. As we have explored in our paper, it is possible to use NNs to classify tumors, which can potentially save countless lives. Future work using NNs can for example be used for earlier detection of diseases like Alzheimer's disease or types of cancer, which can help start treatment at a much sooner stage. That being said, it is generally wise to treat the results from a neural network with some scepticism, as we don't always have a clear understanding of how the NN came to its conclusions.

Bibliography

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hjorth-Jensen, M. (2024). *FYS-STK4155 – Applied Data Analysis and Machine Learning*. https://github.com/CompPhysics/MachineLearning/tree/master/doc. Lecture Notes, University of Oslo.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Street, W. N., W. H. Wolberg, and O. L. Mangasarian (1993). "Nuclear feature extraction for breast tumor diagnosis". In: *Biomedical Image Processing and Biomedical Visualization*. Vol. 1905. International Society for Optics and Photonics, pp. 861–870.

Wei, J. (2019). "Forget the Learning Rate, Decay Loss". In: *CoRR* abs/1905.00094. arXiv: 1905.00094. URL: http://arxiv.org/abs/1905.00094.

Zhang, B., H. Shi, and H. Wang (June 2023). "Machine Learning and AI in Cancer Prognosis, Prediction, and Treatment Selection: A Critical Approach". In: *Journal of Multidisciplinary Healthcare* Volume 16, pp. 1779–1791. ISSN: 1178-2390. DOI: 10.2147/jmdh.s410301. URL: http://dx.doi.org/10.2147/JMDH.S410301.

Appendix A

A.1 GitHub Repository

We have all our source code and test runs in our repository on GitHub at this link:

https://github.com/sphia-g/ML24UiO/tree/main/Project2