Comparing gradient descent methods, and using neural networks and logistic regression to analyze the Wisconsin Breast Cancer dataset.*

Erling Nupen
Ole Kristian Rustebakke
Brage Andreas Trefjord

November 19, 2023

Abstract

In this research project, a Feed Forward Neural Network was constructed to assess the probability of malignancy or benignity in breast tumors based on diverse features. A comprehensive examination of multiple optimization algorithms was conducted, leading to the identification of the Adam optimization algorithm as the most advantageous for our neural network. Our neural network was used on the two-dimensional Franke Function, but with a maximum R^2-score of 0.87 with RELU as activation function and learning rate $\eta = 0.001$, hyperparameter $\lambda = 10^{-4}$ we found that OLS or Ridge regression is better suited for such a problem. The performance of our neural network was juxtaposed with the logistic regression method, which is equivalent to a FFNN without hidden layers, revealing that a neural network configured a learning rate $\eta = 0.01$ and a regularization parameter $\lambda = 10^{-4}$, as discerned through heatmap analyses, gave the best results across different hidden layer activation functions and network structures. The best achieved accuracy was found with the sigmoid function as both hidden layer and output activation functions, and using one hidden layer of 50 nodes. This configuration gave an accuracy of 96%. In contrast, logistic regression achieved its highest accuracy of 68% under conditions of $\eta = 0$ and a learning rate of 0.

Contents

*GitHub Repository: `https://github.com/Bragit123/FYS-STK3155/tree/main/Project_2`

1 Introduction

This report will present a regression analysis, the various steps needed in constructing a Feed Forward Neural Network (FFNN), and towards the end, a comparative analysis with logistic regression. The FFNN is one of the simplest and first Artificial Neural Networks(ANNs) ever constructed and is characterized by the uni-directional forward flow of information. The main goal is to compare our FFNN to logistic and linear regression and perform a critical evaluation of the various algorithms employed. In our case we look at the Wisconsin Breast Cancer data [1], with a FFNN and comparing with logistic regression, as well as the Franke Function, using a FFNN and comparing with linear regression.

Over time, ANNs have evolved, to have several layers of artificial neurons. These artificial neurons are classified depending on which activation function they use. In this report, three different activation functions will be tested, sigmoid, RELU and LRELU. Neural Networks with multiple layers of neurons with non-linear activation functions are known as multilayer perceptrons (MLPs) neural networks[1].

Section 2 contains various background on the theory behind a FFNN. First the gradient descent (GD) and stochastic gradient descent (SGD) method is described in Section 2.1, along with the introduction of the various cost functions we will use. In Section 2.2 the method of logistic regression, which will be used on the Wisconsin Breast Cancer Dataset, is explained. The structure of the neural network, with the introduction of various activation functions that are used, and the back-propagation algorithm is explained respectively in Sections 2.3 and 2.7. In Sections 2.4 and 2.5 respectively, the Franke Function and the Wisconsin Breast Cancer Dataset, which we will use our neural network to train and predict, is described. The Results and Discussion, Section 3, provide an analysis of the neural network and how it performs on the data which we use. The Conclusion, Section 4, is just a summary of what we deduced from the discussion.

2 Theory

Firstly, we will present the intricacies of optimization algorithms, specifically focusing on gradient descent (GD) and its stochastic variant, stochastic gradient descent (SGD). These algorithms play a pivotal role in training machine learning models, influencing the convergence speed and overall performance. We will also present different learning schedules, and how they interplay with gradient-based optimization. Toward the end, we will cover the backpropagation algorithm, cost functions, and

[1] Confusingly, although we are dealing with non-linear activation functions, these neural networks are still called MLPs, for historical reason[2]

the data sets we would like to fit.

2.1 Gradient Descent

The explanation of Gradient Descent with different tuning algorithms follows below. The implementation of this was mostly taken from Morten Hjorth Jensens lecture notes [3].

2.1.1 Regular Gradient Descent

For a function $F(\vec{x})$ the direction of the steepest descent is that of the negative gradient $-\nabla F(\vec{x})$. This means we can define the next step in the steepest direction as Eq (1)

$$\vec{x}_{k+1} = \vec{x}_k - \gamma_k \nabla F(\vec{x}). \tag{1}$$

After a specific number of iterations, k, and ensuring that $\gamma_k > 0$ is suitably small, we will eventually converge to a local minimum. We aim to minimize a cost function $C(\beta)$. This is done by doing the iterations in (1) for the weights/coefficients β.We aim to minimize the cost function as it serves as an estimate of the error inherent in fitting a given function. For example, if the function we want to fit is y and we have a fit $\tilde{y} = X\beta$, where X is the feature matrix, the cost function for n inputs with the ordinary least squares (OLS) method is the mean squared error given in Eq (2)

$$C(\beta) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y})^2 = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - X\beta)^2. \tag{2}$$

y_i are the datapoints and $\hat{y}_i$ are the predicted values of our model for $i = 0, 1, \ldots, n-1$.

For Ridge regression, we apply a correction term λ to avoid overfitting. The cost function for Ridge is given in Eq (3)

$$C(\beta) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - X\beta)^2 + \lambda||\beta||_2^2. \tag{3}$$

In our study of logistic regression and classification problems we will, instead of minimizing error as in OLS and Ridge, train the model's parameters in a way that maximizes the likelihood of observing the actual outcomes in the training data.

This leads to new cost functions to minimize, namely the cost cross-entropy cost function given in Eq (4).

$$C(\beta) = \sum_i \left[y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i) \right]. \tag{4}$$

GD as seen in equation (1) can be written in our problem as

$$\beta_{i+1} = \beta_i - \eta \nabla C(\beta_i). \tag{5}$$

This iteration method with a constant learning rate η means we will have to choose an η such that the jumps from iteration to iteration are not too big so that we do not jump past the minima. We also need a learning rate that is not too small, or else it would take too many iterations to reach the minimum, which is computationally heavy for each iteration.

2.1.2 Gradient Descent with Momentum

One way of reaching the minimum quickly is to add a momentum term, as in Eq (6), to our GD in order to tune the learning rate.

$$\beta_{i+1} = \beta_i - \eta \nabla C(\beta) - p \cdot (\beta_i - \beta_{i-1}). \tag{6}$$

The momentum, p, is a constant. This momentum term will make the iteration go faster towards the minimum when we are far away, because the descent is steep, and when we are close it slows down so that we do not jump past the minimum. For all the algorithms β_0 is set as a random vector where every element is drawn from a normal distribution with standard deviation $\sigma = 1$ and mean $\mu = 0$.

2.1.3 Stochastic Gradient Descent

Regular GD requires us to use the whole dataset at every iteration in order to compute the gradient. This quickly becomes computationally heavy when we work with many parameters and many iterations. One way to reduce the computational cost is to use stochastic gradient descent (SGD). This method works by splitting the data into stochastic subsets, called *minibatches*. Instead of computing the gradient from the whole dataset, we now compute the gradient from just one such minibatch, and update the weights based on this gradient using Eq (5).

We then do the same for the next minibatch, and so on. In this way, we heavily reduce the computational cost of the algorithm by reducing the number of operations per iteration. This allows GD to converge faster to optimal weights.

2.1.4 Adagrad

One method of tuning the learning rate is Adagrad (Adaptive Gradient). We set the global learning rate η as a constant. For every iteration we first compute the gradient $\nabla C(\beta_i)$ and the accumulated squared gradient $G_{i+1} = G_i + \nabla C(\beta_i)^2$, starting at $G_0 = 0$. Then we compute the updated β as Eq (7)

$$\beta_{i+1} = \beta_i - \frac{\eta}{\delta + \sqrt{G_{i+1}}} \nabla C(\beta_i), \tag{7}$$

where $\delta = 10^{-8}$ in our case is added to avoid division by zero if the gradient is zero. Dividing by the root of the accumulated squared gradient $\sqrt{G}$ takes the history of the gradient into account so that each step towards the minimum becomes smaller as we converge towards it. It is made to converge rapidly for convex functions, as discussed in chapter 8.5.1 in Goodfellow et. al. [4].

2.1.5 RMS-prop

RMS-prop (Root Mean Square Propagation) is a modification of Adagrad, which uses an exponentially decaying average to get rid of gradients from the earlier iterations, and then converges rapidly once it finds a convex region, this is discussed in chapter 8.5.2 in Goodfellow et. al. [4].

In the algorithm, we set a global learning rate η as well as a decay rate ρ. As usual, we compute the gradient $\nabla C(\beta_i)$. We now use the decay rate to compute the accumulated squared gradient, Eq (8)

$$G_{i+1} = \rho G_i + (1 - \rho) \nabla C(\beta_i)^2. \tag{8}$$

Starting at $G_0 = 0$, then the update is given in Eq (9)

$$\beta_{i+1} = \beta_i - \frac{\eta}{\sqrt{\delta + G_{i+1}}} \nabla C(\beta_i), \tag{9}$$

with $\delta = 10^{-8}$.

2.1.6 Adam

Adam (Adaptive Moments) can be seen as a combination of RMS-prop and momentum. We need two exponential decay rates ρ_1 and ρ_2 and we will calculate moment estimates given in equations (10) and (11)

$$G_{1,i+1} = \rho_1 G_{1,i} + (1-\rho_1)\nabla C(\beta_i), \tag{10}$$
$$G_{2,i+1} = \rho_1 G_{2,i} + (1-\rho_1)\nabla C(\beta_i)^2. \tag{11}$$

Introducing a correction bias to compensate for the initialization of $G_{1,i+1} = 0$, $G_{1,i+1} = 0$, we get equations (12) and (13)

$$\hat{G}_{1,i+1} = \frac{G_{1,i+1}}{1-\rho_1^t}, \tag{12}$$
$$\hat{G}_{2,i+1} = \frac{G_{2,i+1}}{1-\rho_2^t} \tag{13}$$

where t is the current iteration. Subsequently, by applying the update, we are able to estimate the next optimal β value, as shown in Eq (14)

$$\beta_{i+1} = \beta_i - \eta \frac{\hat{G}_{1,i+1}}{\delta + \sqrt{\hat{G}_{2,i+1}}} \nabla C(\beta_i). \tag{14}$$

2.2 Logistic Regression

Logistic regression is most commonly applied to a situation with two possible outcomes. This problem is known as a classification problem, where the outcome is either 0 or 1, true or false, success or failure, etc. Logistic regression as a statistical method used for binary classification is a useful stepping stone to more advanced neural network algorithms and supervised deep learning.

Binary classification revolves around predicting the outcome of a categorical dependent variable with two possible values (0 or 1), or outcomes. The dependent variable represents the outcome, and the independent variables are used to predict this outcome.

Logistic regression essentially predicts the probability that the dependent variable belongs to a particular category. In order to map the dependent variable to a binary output, the most common function to use, and the one we used in this project, is the sigmoid function 18.

Mathematically, the likelihood of a positive outcome in logistic regression can be expressed as Eq (15)

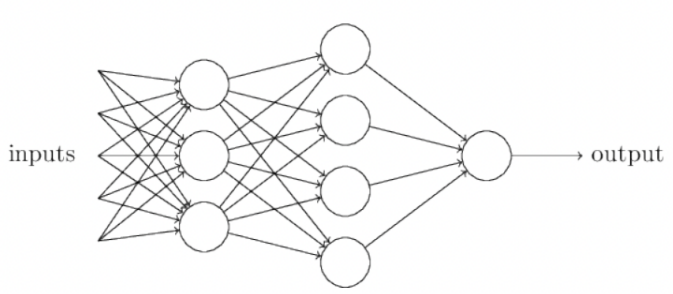


Figure 1: Illustration showing how the different neurons pass information to each other. Notable, input from the previous layer is passed to each neuron in the next layer

$$P(Y=1) = \frac{1}{1+e^{-(b_0+b_1X_1+b_2X_2+\ldots+b_nX_n)}}. \tag{15}$$

Logistic regression is equal to using an FFNN without any hidden layers and with the sigmoid activation function in the output layer[2]. In an FFNN without hidden layers, each input node corresponds to a feature, and due to a single output node with the activation function being a sigmoid function. The weights $(w_0, w_1, \ldots, w_n)$ and inputs $(X_0, X_1, \ldots, X_n)$ are analogous to the parameters in logistic regression.

2.3 Structure of a Multilayer Perceptron Neural Network

An illustration of the multilayer perceptron (MLP) neural network is given in Figure 1 [2]

The first layer of neurons is called the input layer. All layers after the first, not including the last layer, are considered hidden layers of neurons. The last layer is called the output layer. For each neuron in each layer, a weighted sum of the input is calculated, as shown in Eq (16)

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1. \tag{16}$$

x_j is the input, w_{ij} is the weight, and b_i is the bias. The sum goes from the first input, $j = 1$ to the last input M. After the first layer, the weighted output of layer 1, is passed through the activation function as shown in Eq (17)

[2]This realization came to me after spending quite some time trying to make a Logistic Regression code.

$$y_i^1 = f\left(z_i^1\right) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right). \tag{17}$$

where f is the activation function. For our neural network, the activation function will be the sigmoid function, the Rectified Linear Unit (RELU), the Leaky RELU function and the identity function. Each output of the previous layer will be the input of each neuron in the next layer, going all the way through to the last layer. The characteristics of the activation functions are as follows:

The **Sigmoid Function** is given in Eq (18).

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{18}$$

This is a logistic function, a smooth S-shaped curve that maps input values to a range between 0 and 1. It is useful for classification problems since its output is restricted to the range $(0, 1)$. It is notorious for struggling with vanishing gradient issues in deep neural networks[5].

One potential solution to a vanishing gradient is the **Rectified Linear Unit** (RELU) function given in Eq (19).

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}. \tag{19}$$

RELU is a simple and widely used activation function that replaces negative input values with zero and keeps positive values unchanged. Disregarding negative input values effectively addresses the vanishing gradient problem making it suitable for training deep neural networks[6][7].

However, since RELU sets negative input values to zero, you could run into another problem, where your gradient ends up essentially dying out, i.e. becoming zero. In machine learning terminology, this is called a dead neuron[8]. To prevent dead neurons, a function called **Leaky RELU** (LRELU) was developed. Leaky RELU can be mathematically summarized as Eq (20)

$$f(x) = x \text{ if } x > 0, \text{ and } f(x) = 0.01x \text{ if } x <= 0. \tag{20}$$

Leaky RELU allows for small negative inputs, which prevents the neuron from dying.

The last activation function we will use is the **Identity Function** given in Eq 2.3, which simply returns the input

$$f(x) = x..$$

This activation function essentially implies the absence of any activation function.

2.4 Franke Function

We want to use our neural network to fit the Franke function, given in Eq (21)

$$\begin{aligned} f(x,y) = 0.75\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + 0.75\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)^2}{10}\right) \\ + 0.5\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - 0.2\exp\left(-\frac{(9x-4)^2}{4} - \frac{(9y-7)^2}{4}\right). \end{aligned} \tag{21}$$

We will be using different amounts of nodes and hidden layers to find the best fit. Adam with momentum, (14), is the method we will use to tune the learning rate. Sigmoid, RELU, and LRELU are the activation functions we will use for the hidden layer, and since we want the output to be some function value, we will use the identity function on the output layer. We use 20x20 uniformly distributed datapoints $x, y \in (0, 1)$, i.e. 400 datapoints in total, with an added stochastic noise of $0.1N(0, 1)$ using numpy.random with seed 200. We split the data into a 80% train set and 20% test set. We also use scikits MLPRegressor which includes a function called fit, this is what is called to train the network. This is used merely to have a different network so we can compare our results.

2.5 Wisconsin Breast Cancer Dataset

The Wisconsin Breast Cancer dataset[1] is a widely used dataset in machine learning and medical research. It comprises features extracted from digitized images of breast cancer biopsies, such as the mean radius, texture, and smoothness, among others. The dataset's primary purpose is to facilitate the classification of tumors as either benign or malignant based on these features.

We acquire the dataset using the scikit-learn package [9]. The dataset consists of 30 features of tumors of 569 patients. We split the dataset into a training set (80%) and a validation set (20%). We then use the training set to train a neural network, and the validation set to check the accuracy of the model. We also use this dataset in

our study of logistic regression, but in this case, it is the coefficients using gradient descent we train, not the weights of a neural network.

2.6 Learning vs optimization

The objective of optimization algorithms is to determine the optimal model parameters that either minimize or maximize an objective function, denoted as the cost or loss function, as defined in equations (22) and (23). To achieve this objective, we will employ gradient-based optimization techniques, which will be utilized to minimize the cost functions described in equations (22) and (23).

$$\mathcal{C}(\hat{W}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 \tag{22}$$

$$L(y, p(y=1)) = -[y \cdot \log(p(y=1)) + (1-y) \cdot \log(1 - p(y=1))] \tag{23}$$

Mean-squared error, Eq (22) is mainly used for regression problems where the target variable is continuous, whilst Cost-Cross entropy, Eq (23) is useful for classification problems. Since the main goal of this project is to study classification and regression problems using our own FFNN, a key point is to choose the optimal cost function for our neural network.

2.7 Back-propagation algorithm

The Back-propagation algorithm is a fundamental part of the training of a neural network. Using the optimization algorithms in Section 2.1, we are able to calculate the gradient of the cost function backward from the final layer in order to obtain the optimal weights of the network. Using Eq (22) we can reformulate it in terms of the final layer, as in Eq (24)

$$\mathcal{C}\left(\hat{W}^L\right) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(a_i^L - t_i\right)^2. \tag{24}$$

Where a_i is the activation function defined as f in Eq (17). The goal of the back-propagation algorithm is to fine-tune the weights and learn from the errors in order to tune the weights and biases. We initialize the weights by the normal distribution, $N(0,1)$, and the biases by $0.1N(0,1)$. In order to do this error correction or self-learning, we need to derive the derivative of the cost function in terms of the weights

of each neuron. The derivative of the cost function with respect to the weights can be written as in Eq (25)

$$\frac{\partial \mathcal{C}\left(\hat{W}^L\right)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right) \frac{\partial a_j^L}{\partial w_{jk}^L} \tag{25}$$

Using the chain rule, Eq (25) can be written in terms of the activation function of the final layer and the second to last layer, as presented in Eq (26)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L \left(1 - a_j^L\right) a_k^{L-1}. \tag{26}$$

This can be more compactly written as Eq (27)

$$\frac{\partial \mathcal{C}\left(\hat{W}^L\right)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}. \tag{27}$$

These results were obtained from [10].

3 Results and Discussion

3.1 Gradient Descent and stochastic gradient descent

To see which values for the learning rate η and the hyperparameter λ from Equations (5) and (3) gives good results we have plotted in Figure 2 the R^2-scores which we get after applying stochastic gradient descent with momentum on the function $f(x) = 5x^2 + 3x + 1$. The input is 100 uniformly distributed $x \in [0, 1]$, and we use a 80-20 train-test split. The added stochastic noise is $0.1N(0, 1)$ with key 123 from jax random library, momentum $p = 0.001$, and we used 100 epochs with a batch size of 20. Ridge is the cost function that is used. The figure shows that we get the best fits for $\eta = 0.01$ and $\lambda = 0.01, 0.1$, which gives $R^2 = 0.98$, which is close to the desired result $R^2 = 1$. The results are not good for $\eta = 10^{-4}$, with a negative R^2-score. This learning rate is likely too small, so that the cost function does not reach its minima in 100 epochs.

We want to compare the different methods of tuning the learning rate while doing gradient and stochastic gradient descent for the function. To avoid doing very many

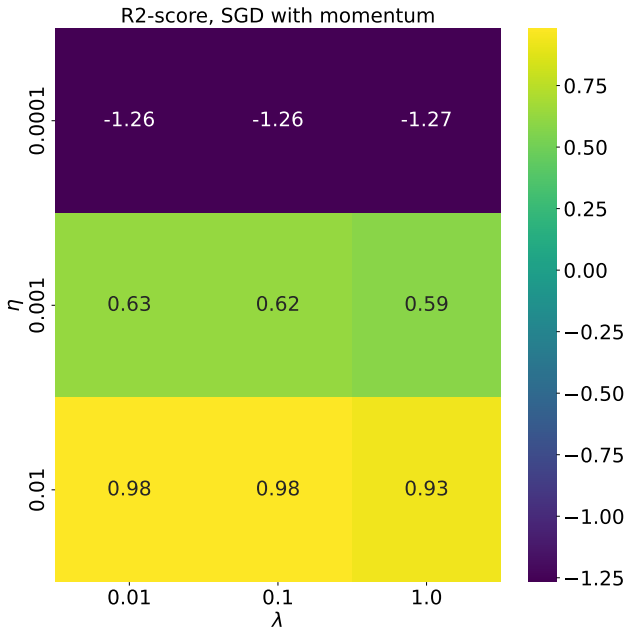
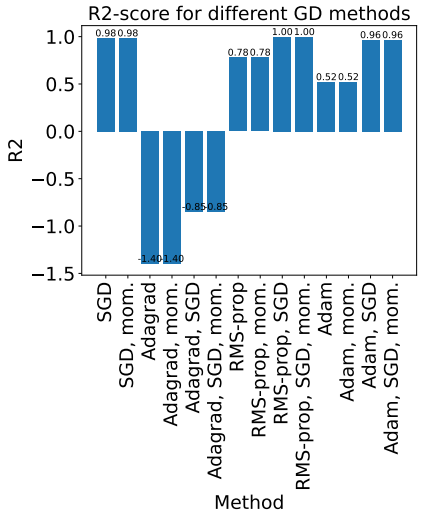


Figure 2: R^2-score for different learning rates η and hyperparameters λ with Ridge as cost function and SGD with momentum on a polynomial of degree 2. Batch size is 20 and we use 100 epochs

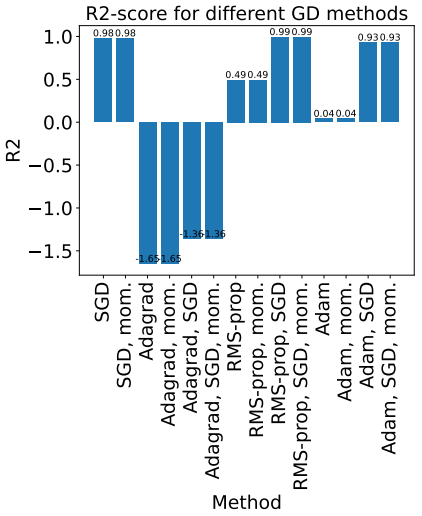
computations we chose the best parameters from Figure 2, i.e. $\eta = 0.01$ and $\lambda = 0.1$. For RMS-prop we use decay rate $\rho = 0.99$ and for Adam decay rates $\rho_1 = 0.9$ and $\rho_2 = 0.999$. We used 100 iterations for the gradient descent methods, and 100 epochs with batch size 20 for the stochastic gradient descent methods. Figure 3a shows the R^2-score for the different methods with OLS as cost-function, and figure 3b shows the R^2-scores but with Ridge as cost-function. The R^2-score is almost unaffected by which cost function we use for stochastic gradient descent (slightly worse for Ridge), but using Ridge gives a much lower score for the gradient descent methods. We can see that there is a big difference in R^2-score for the different learning algorithms. Gradient descent with and without momentum gave respectively $\beta = (-1.2, -0.6, -0.4) \cdot 10^9$ and $\beta = (-1.0, -0.6, -0.4) \cdot 10^9$ with OLS as cost function and respectively $\beta = (-1.4, -0.7, -0.5) \cdot 10^9$ and $(6.9, 3.7, 2.6) \cdot 10^{14}$ and is therefore not plotted. Note that GD and GD with momentum gave better results for a different learning rate, but this still shows how quickly you get bad results if you do not choose the learning rate very precisely. Stochastic gradient descent consistently gives R^2-scores closer to 1. Adagrad gives negative R^2-scores, and it is probably because it never reaches β, so it would need more iterations. RMS-prop and Adam gives the best results, with RMS-prop actually giving $R^2 = 1.00$ with SGD. Still we will use Adam in our neural network simulation, since it has given solid R^2-scores of $R^2 = 0.96$ and $R^2 = 0.93$ for OLS and Ridge respectively, and is regarded as more robust for choice of hyperparameters, as discussed in Chapter 8.5.3 in Goodfellow et.al [4]. Note that even though the R^2-scores are close to 1, the coefficients β is off. For Adam with SGD and momentum and OLS as cost function, we get $\beta = (1.5, 2.7, 3.9)$ and we know that the function should have $\beta \approx (1, 3, 5)$, which makes it probable that we would get a lower R^2-score for a different test set.

3.2 Franke fit

First, we train the neural network to find a fit for the Franke function in equation (21). As discussed we have 400 datapoints and use a 80-20 train-test split. We use stochastic gradient descent and Adam with momentum to tune the learning rate, we used descent rates $\rho_1 = 0.9$, $\rho_2 = 0.999$ and momentum $p = 0.01$, the batch size is 5 and 100 epochs are used. We use OLS as cost-function. The weights and biases are what we train now, by using backpropagation. We trained the network for different amounts of hidden neurons and hidden layers, but found 1 hidden layer with 50 hidden neurons to give the best results. In respectively Figure 4a, 4b and 4c we have used sigmoid, RELU and LRELU as activation functions for the hidden layer. For the output layer we use the identity function as the activation function, since we want the ouput to be a function value, and because sigmoid, RELU and LRELU is more applicable when we want the output to be for example true or false, two different options. This is because they force the output to take a value close to 0 and 1 in large regions, and they can therefore affect the output negatively. In general all the



(a) R^2-score with OLS as cost function.

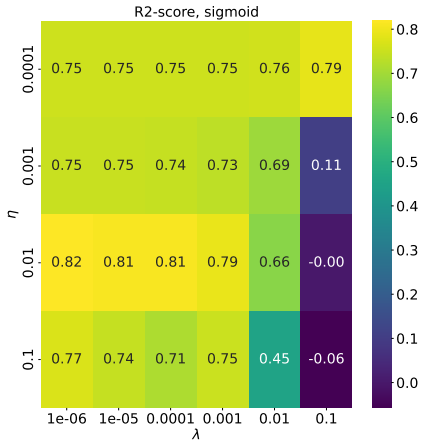


(b) R^2-score with Ridge as cost function. The hyperparameter $\lambda = 0.1$

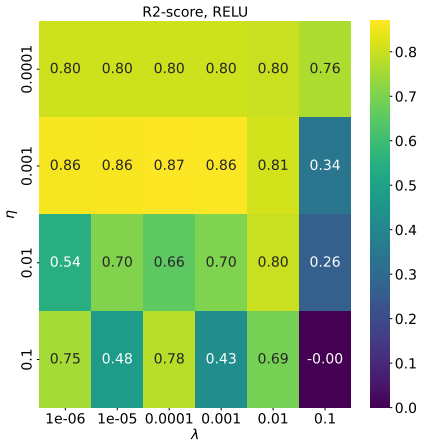
Figure 3: R^2-score for different methods of tuning the learning rate, using η=0.01, momentum $p = 0.001$, $\rho = 0.99$ for RMS-prop, $\rho_1 = 0.9$ and $\rho_2 = 0.999$ for Adam. Batch size is 20 with 100 epochs where SGD is used.

functions seem to give large regions where the R^2-score is moderately high (>0.75). Sigmoid has a maximum of $R^2 = 0.82$ for $\eta = 0.01$ and $\lambda = 10^{-6}$, while RELU gives the best performance, with a maximum of $R^2 = 0.87$ for $\eta = 0.01$ and $\lambda = 10^{-4}$, and LRELU has a maximum of $R^2 = 0.85$ for $\eta = 0.001$ and $\lambda = 0.001$. There are generally large areas for all the different activation functions where the R^2-score is similar, but we see that for $\lambda = 0.1$ the R^2-score is mostly low. This is probably because it will make the regularization term in the update for the weights dominate, making the gradient term almost negligible. Obviously we do not want this, as the gradient is the basis for the update of the weights. With our own implementation of OLS and Ridge regression for the Franke function, our maximum R^2-score was $R^2 \approx 0.95$ for both methods, even with only 100 datapoints [11], and generally they are more consistent for different parameters. Therefore it can probably be said that the regression methods perform better when it comes to fitting a function.

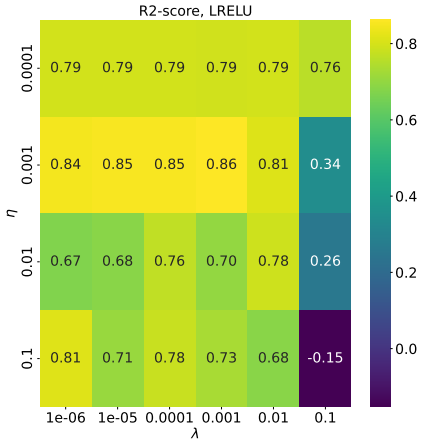
In Figure 5a and 5b we have plotted the R^2-score with respectively sigmoid and RELU as activation functions using scikits neural network code MLPRegressor. The parameters are exactly the same as we used in our own neural network, but we can see that the R^2-scores are different. With sigmoid as activation function the R^2-scores seem to be worse for scikit, with a maximum only of $R^2 = 0.77$ for $\eta = 0.001$ with $\lambda = 10^{-6}$, and $\eta = 0.001$ with $\lambda = 10^{-5}$. With RELU as activation function



(a) R^2-score using sigmoid as activation function for the hidden layer.



(b) R^2-score using RELU as activation function for the hidden layer.

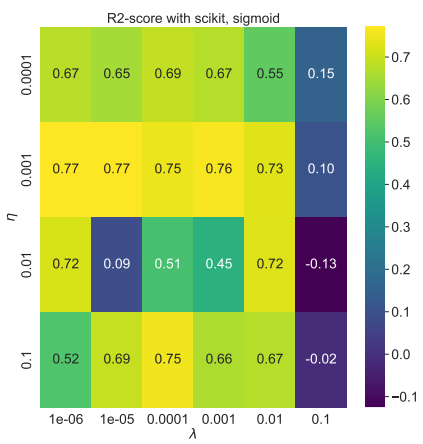


(c) R^2-score using LRELU as activation function for the hidden layer.

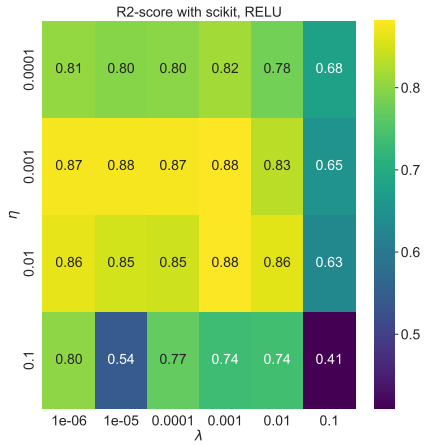
Figure 4: R^2-score for the fit of the Franke function when using the activation functions sigmoid, RELU and LRELU for the hidden layer. We have 2 inputs (x,y), 50 hidden neurons in 1 hidden layer and 1 output value (z). Learning rate η is on the y-axis and hyperparameter λ on the x-axis. Adam with momentum is used to tune the learning rate, with $\rho_1 = 0.9$, $\rho_2 = 0.999$ and momentum $p = 0.01$. The batch size is 5 and we use 100 epochs.

the results are better for scikit, with a maximum of $R^2 = 0.88$ for $\eta = 0.001$ and $\lambda = 0.0001$ or $\lambda = 10^{-5}$, and for $\eta = 0.01$ with $\lambda = 0.001$. The reason why scikit perform differently may lie in the way the weights and biases are initialisied, since everything else is equal, but the difference in quality of the fits are within reason, and we cannot say scikit performs better or worse than our own code. It should

also be noted that Scikits program runs much quicker than ours, but we did not focus much on making the code efficient, and we have a lot of if-statements which are generally less efficient.



(a) R^2-score for scikits neural network using sigmoid as activation function.



(b) R^2-score for scikits neural network using RELU as activation function.

Figure 5: R^2-score for the Franke fit using scikits MLPRegressor to train the program. Sigmoid and RELU are used as activation functions, Adam is used to tune the learning rate, with momentum $p = 0.01$, $\rho_1 = 0.9$, $\rho_2 = 0.999$. Again we have 50 hidden neurons in the hidden layer.

To visualise how well our data is fitted we plotted our fit together with the Franke function in Figure 6. We used sigmoid as activation function with learning rate $\eta = 0.1$ and hyperparameter $\lambda = 10^{-5}$, and Adam with descent rates $\rho_1 = 0.9$, $\rho_2 = 0.999$ and momentum $p = 0.01$. This gave $R^2 = 0.74$, and we can see that the shape of the functions are similar, although there is some distance between them, as the R^2-score would suggest.

3.3 Wisconsin breast cancer data

3.3.1 Neural Network

We have created a neural network, and trained it on the Wisconsin Breast Cancer dataset [1]. In order to evaluate the neural network, we have tested different activation functions for the hidden layers of our network, and different dimensions (number of hidden layers and nodes). We have also used different regularization parameters λ and gradient descent learning rates η. In Figure 7 you can see the accuracy of the

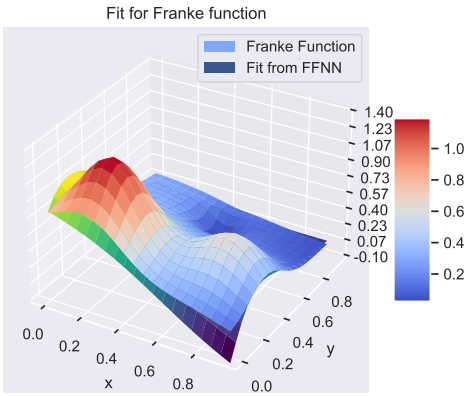


Figure 6: A visualisation of the fitted function our neural network produces to the Franke function. 50 hidden neurons in one layer is used, and Adam is used as tuning algorithm with $\eta = 0.1$, $\rho_1 = 0.9$, $\rho_2 = 0.999$, momentum $p = 0.01$ and $\lambda = 10^{-5}$

validation data for different dimensions. For each dimension, we have plotted the accuracy for different regularization parameters and learning rates.

The highest accuracy obtained is 0.96, which can be seen in Figure 7d with one hidden layer of 50 nodes. This accuracy is obtained by setting $\lambda = 10^{-4}$ and $\eta = 10^{-2}$. We can see, however, that the model is very sensitive to small changes in the learning rate η, as multiplying or dividing it by 10 gives much lower accuracies of 0.39 and 0.30 respectively. It makes sense that small changes in the learning rate changes the accuracy significantly. If the learning rate is too large the weights of the neural network will change change very much at every iteration of the training process, which leads to inaccurate tuning of the weights. If, however, the learning rate is too small the weights will hardly change at each iteration, leading to slow convergence.

Using one hidden layer with 100 nodes also gives high accuracies as can be seen in Figure 7e. If we again choose $\lambda = 10^{-4}$ and $\eta = 10^{-2}$ we get an accuracy of 0.95, which is quite close to the accuracy of 0.96 that we got from 50 nodes. This network is not as sensitive to small changes in λ and η. Multiplying or dividing λ by 10 does not give a difference for the accuracy at this point, but remains stable at 0.95. Multiplying or dividing η by 10 does, however, give significant changes in the accuracy, as it did for 50 nodes. The changes are not as large this time though, as the accuracy drops to 0.61 and 0.52 when multiplying and dividing η by 10 respectively, opposed to 0.39 and 0.30 for 50 nodes.

The other dimensions we tested are two hidden layers of 10 nodes (Figure 7a), which

gave a maximum accuracy of 0.88; two hidden layers of 50 nodes (Figure 7b), which gave a maximum accuracy of 0.89; and one hidden layer of 10 nodes (Figure 7c), which gave a maximum accuracy of 0.85.

One might expect more layers and more nodes to automatically give better results, but it is apparent that this is not the case, as the two best models had only one layer. This might come from a sense of overfitting, where too many layers gives the model too much freedom to optimize the weights to the training data, without preserving the ability to predict new data.

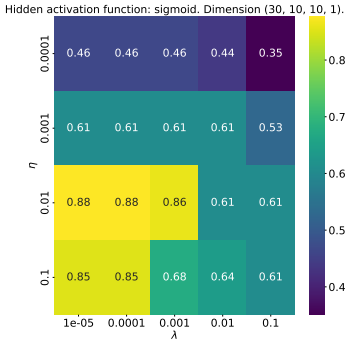
In Figure 8 we have looked at different activation functions for the hidden layers. We have fixed the dimensions to one layer of 100 nodes, and varied the activation function for the hidden layers. We made this choice of dimensions since one layer of 100 nodes gave nearly as good results as one layer of 50 nodes, but with more stability. Since this is a classification problem where we want the output values to be 0 or 1, we decided to fix the activation function for the output layer to be the sigmoid function, as this guarantees results in the range $(0, 1)$.

The highest accuracy obtained here is 0.95, which comes from the sigmoid function (Figure 8a. This is the same plot that we saw in Figure 7e). This accuracy comes when $\eta = 10^{-2}$, and $\lambda \in \{10^{-5}, 10^{-4}, 10^{-3}\}$. The sigmoid function behaves nicely as it makes sure the output of any layer does not diverge by limiting it to the range $(0, 1)$. When using this model we also scaled the input to the same range, which means we restrict the values throughout the complete network, making sure the results never diverge.

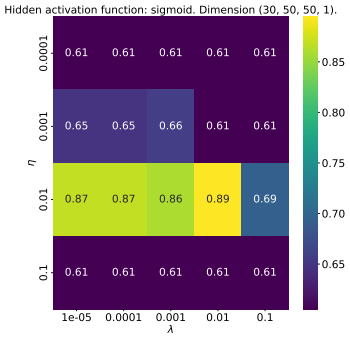
The other activation functions we tested were the identity function (Figure 8b), the RELU function (Figure 8c) and the LRELU function (Figure 8d). The RELU function allows the output to be any positive number, and the identity and LRELU functions allow the output to take the negative values as well. This means none of these functions restrict the output to the range $(0, 1)$, but allow very large (or small) values. This could result in erroneous results since we are still using the sigmoid function as activation function for the output layer, and for very high or low values the gradient gets close to zero. If the gradient approaches zero the weights will barely update at each iteration, which means the network converges slowly to the optimal weights, and we might not reach good weights. This is likely the reason why the identity, RELU and LRELU functions do not give as high accuracies as the sigmoid function.

3.3.2 Logistic Regression

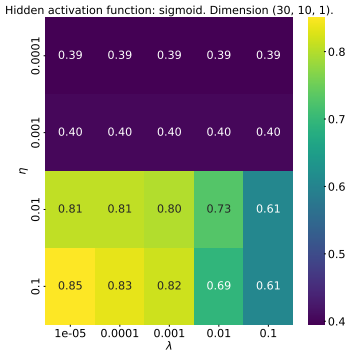
A heatmap plot illustrating the confusion matrix for logistic regression using our own neural network code with no hidden layers is presented in Figure 9. Diverse hyper-parameter values are depicted along the x-axis, while various learning rates are represented along the y-axis. We also fit the data using scikit-learn's implemen-



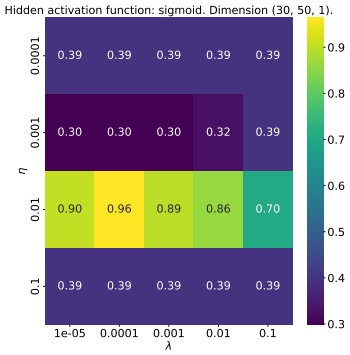
(a) Accuracies for the model with two hidden layers, both with 10 nodes.



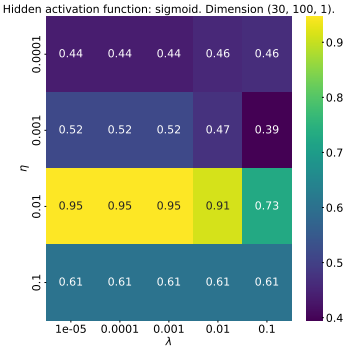
(b) Accuracies for the model with two hidden layers, both with 50 nodes.



(c) Accuracies for the model with one hidden layer with 10 nodes.

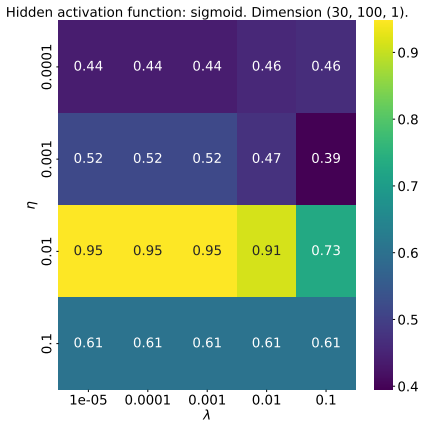


(d) Accuracies for the model with one hidden layer with 50 nodes.

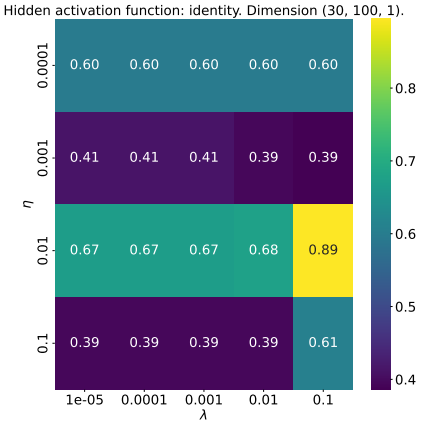


(e) Accuracies for the model with one hidden layer with 100 nodes.

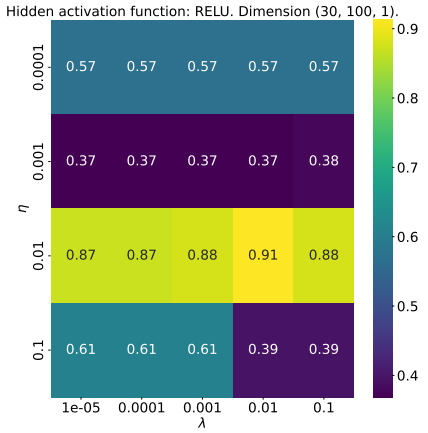
Figure 7: Validation accuracy of our neural network on the Wisconsin Breast Cancer dataset plotted against the regularization parameter λ and the learning rate η. Here we have used the sigmoid function as activation function for both hidden layers and the output layer. The only difference between the networks producing these plots is the dimension of the network.



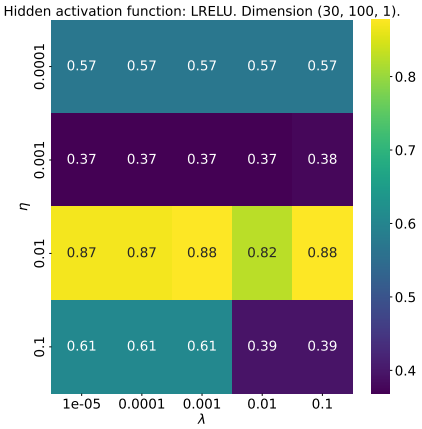
(a) Accuracies for the model with the sigmoid function as activation function for the hidden layers.



(b) Accuracies for the model with the identity function as activation function for the hidden layers.



(c) Accuracies for the model with the RELU function as activation function for the hidden layers.



(d) Accuracies for the model with the LRELU function as activation function for the hidden layers.

Figure 8: Validation accuracy of our neural network on the Wisconsin Breast Cancer dataset plotted against the regularization parameter λ and the learning rate η. Here we have used one hidden layer of 100 nodes. The only difference between the networks producing these plots is the activation function of the hidden layer. In order to make sure that the output did not diverge, we consistently used the sigmoid function as activation function for the output layers in all the networks.

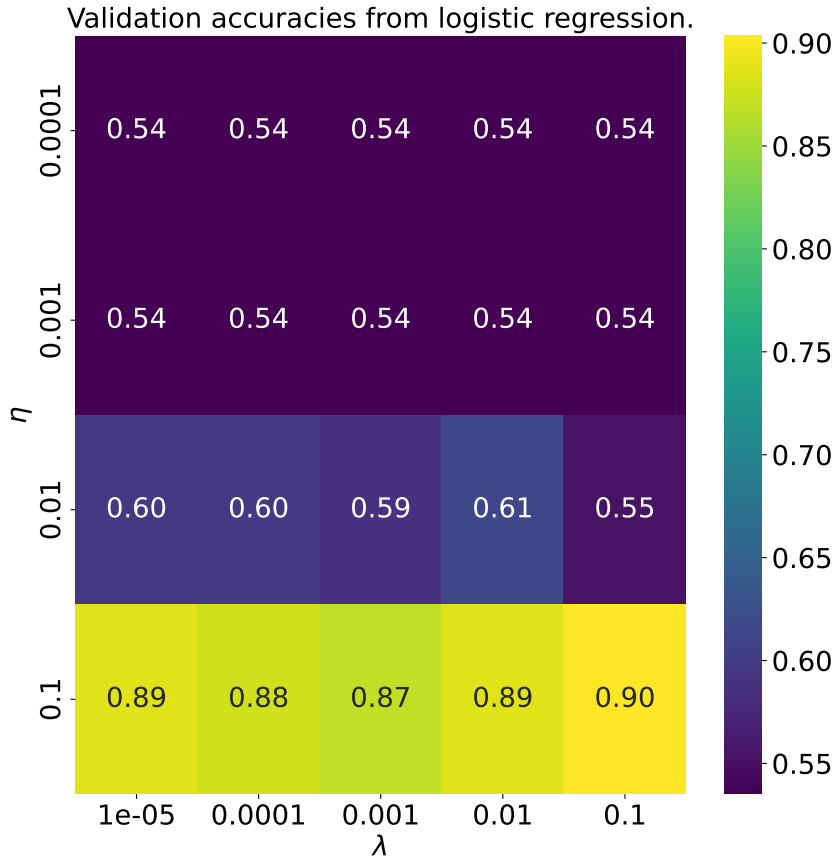


Figure 9: Validation accuracy from logistic regression on the Wisconsin Breast Cancer dataset plotted against the regularization parameter λ and the learning rate η. The logistic regression here is simply a neural network with no hidden layers, and with the sigmoid function as output function.

tation, which gave an accuracy of 96%. This accuracy is better than the accuracies obtained from our own logistic regression code, which reaches it's maximum at 90% accuracy.

Logistic regression, being a linear model struggles to capture non-linearity in Data. The Wisonsin breast cancer data in particular is derived from images, where some features could be non-trivial. Logistic regression relies on the provided features, which may not adapt well to complex hierarchical representations that could be beneficial for breast cancer classification. Since logistic regression is essentially a simplified FFNN without hidden layers, it lacks the depth to iteratively optimize the gradient through multiple layers to best adapt and learn from the optimization algorithm. The absence of these intermediary layers diminishes its capacity to

capture intricate patterns and hierarchical representations in the data. In contrast, including several hidden layers in an FFNN facilitates a more nuanced learning process, enabling the model to discern and adapt to complex relationships, ultimately enhancing its ability to discern optimal features and improve predictive performance. The neural network has the benefit of fine-tuning the weighths and biases, using the backward propagation alogrithm

4 Conclusion

4.1 Fit of Franke Function

We studied how well our Feed Forward Neural Network code worked on fitting the two-dimensional Franke Function. We found that one hidden layer with 50 hidden neurons got the best fits. RELU as activation function gave the best accuracy, measured by the maximum obtained R^2-score, which was $R^2 = 87$ for learning rate $\eta = 0.01$ and hyperparameter $\lambda = 10^{-4}$. LRELU as activation function gave a maximum accuracy of $R^2 = 0.85$ for $\eta = 0.001$ and $\lambda = 0.001$, while sigmoid as activation function gave a maximum of $R^2 = 0.82$ for $\eta = 0.01$ and $\lambda = 10^{-6}$. In comparison with OLS and Ridge regression, these accuracies are low, and we conclude that the neural network is not best applicable for regression.

4.2 Wisconsin Breast Cancer Set

In this report we have studied the Wisconsin Breast Cancer dataset, which consists of 30 features and 569 samples. From this dataset we trained a neural network, where we varied the number of hidden layers and nodes, the hidden layer activation function, the learning rate η and the regularization parameter λ. From our study we found that the best learning rate and regularization parameter were $\eta = 0.01$ and $\lambda = 10^{-4}$, which gave the highest accuracy of 96% for one hidden layer of 50 nodes with the sigmoid activation function. For this choice of network dimensions, the accuracy varied drastically with η and λ, while increasing the number of nodes to 100 gave more stable accuracies when η and λ varied. The highest accuracy achieved with this model was 95%, still with $\eta = 0.01$ and $\lambda = 10^{-4}$.

4.2.1 FFNN compared to Logistic Regression

There are several factors as to why the neural network performs better than logistic regression. The structure of a neural network, with its many hidden layers can capture complex non-linear relationships in the data. Logistic regression struggles to capture intricate non-linear patterns that could be present in the breast cancer dataset. One big advantage a neural network has is the ability to learn feature interactions, whereas logistic regression assumes independence between the various features. Other factors that contribute to the success of the neural network are the

high dimensionality of the data, representation learning, and adaptive learning rates. These three factors allow the neural network to automatically extract hierarchical features, learn new and relevant features, and adjust learning rates for each parameter, effectively capturing more nuanced underlying patterns in the data.

References

[1] W. Wolberg, O. Mangasarian, N. Street, and W. Street, *Breast Cancer Wisconsin (Diagnostic)*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5DW2B, 1995.

[2] M. A. Nielsen, *Neural networks and deep learning*, misc, 2018. [Online]. Available: `http://neuralnetworksanddeeplearning.com/`.

[3] M. Hjorth-Jensen, *Exercises weeks 43 and 44*, University of Oslo, 2023. [Online]. Available: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/exercisesweek43.html`.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[5] L. Ven and J. Lederer, *Regularization and reparameterization avoid vanishing gradients in sigmoid-type networks*, 2021. arXiv: `2106.02260 [cs.LG]`.

[6] A. F. Agarap, *Deep learning using rectified linear units (relu)*, 2019. arXiv: `1803.08375 [cs.NE]`.

[7] S. Basodi, C. Ji, H. Zhang, and Y. Pan, "Gradient amplification: An efficient way to train deep neural networks," *Big Data Mining and Analytics*, vol. 3, no. 3, pp. 196–207, 2020. DOI: `10.26599/BDMA.2020.9020004`.

[8] T. Whitaker and D. Whitley, *Synaptic stripping: How pruning can bring dead neurons back to life*, 2023. arXiv: `2302.05818 [cs.LG]`.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[10] M. Hjorth-Jensen, *Neural networks and constructing a neural network code*, University of Oslo, 2023. [Online]. Available: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week41.html#`.

[11] Nupen, Erling and Rustebakke, Ole Kristian and Trefjord, Brage Andreas, *Applying Regression and Resampling Techniques to Norwegian Terrain Data with Franke's Function as Test Function*, University of Oslo, 2023.