FYS-STK4155

Applied Data Analysis and Machine Learning

Classification and Regression

from Linear and Logistic Regression to Neural Networks

Authors:
Celine Marie Solberg & Ole Petter Maugsten

November, 2022

Table of Contents

List of Figures

Abstract

In this report, neural networks were used and examined for regression- and classification-problems. The networks were compared to linear and logistic regression methods for modelling datasets. Datasets generated with the Franke function was used for the regression case. The linear regression approach gave, for one run, a MSE of 0.00767. The neural network managed to get it down to 0.00672. For the classification problem, the Wisconsin Breast Cancer dataset was studied. The logistic regressor was up to 94.7% accurate, while the neural network was 97.3% accurate. All approaches gave accepted results.

Introduction

There is an approach to artificial intelligence called *neural networks*, and it has gained a lot of attention in recent times. Neural networks have had great success with image processing like facial recognition and cancer detection. They have also proven very useful for forecasting in fields like finance and scientific research. Neural networks mimic the inner workings of the human brain, by sending signals through them and lighting up artificial neurons along the way, eventually making a prediction of some sort. By feeding the networks with data, they can learn from it by seeing which neurons were useful to light up and which who were not when making predictions. The incredible increase in computing power in later years along with the availability of big datasets, have allowed neural networks to be taken to new hights. In 2016, the deep neural network AlphaGo beat the 9-dan professional Lee Sedol in a five-game match of the board game Go. It shocked the world that a machine could be trained to do this.

In project 1, regression analysis was performed on the Franke function. The ordinary least squares method was used to optimise the fitting parameters to a design matrix with polynomial terms. It is interesting to see if the mean squared error could be reduced when approaching this problem with artificial neural networks instead.

The Wisconsin Breast Cancer dataset is made with fine needle aspirate of different breast masses. The dataset describe characteristics of cell nuclei, and categorise the sample as malignant or benign. Neural networks might learn from this data and be able to classify future samples. For comparison, logistic regression will also be performed. We will then see if our neural networks can beat the regression and classification models.

Theory

Neural Networks

The goal of a neural network is to map some input data X to another dataset y. A neural network consists of a collection of artificial neurons often called *nodes*. The nodes are organised in *layers*. There are connections between the nodes of one layer and the next. When a node receives a signal, a real number, it activates and sends new signals to its connected nodes. The strengths of the new signals are determined by a set of weights and biases associated with each layer. Figure 1 shows an example of a neural network.

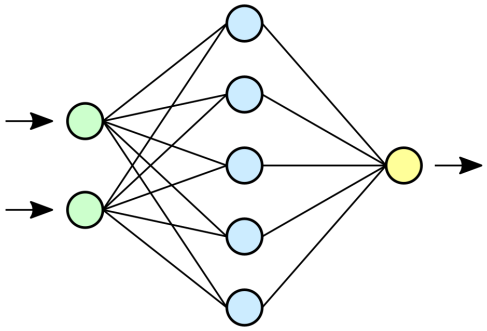

Figure 1: The figure shows a simplified view of a generic neural network with one hidden layer. The input layer (green) has two nodes, the hidden layer (blue) has five, and the output layer (yellow) has one node.

The input layer of a neural network has as many nodes as there are features (columns) in the dataset X. There are as many nodes in the output layers as there are dependent variables to predict. There can be an arbitrary number of hidden layers between the input and output layer. When the dataset X is passed through to the first hidden layer, it is first multiplied by a set of weights $W^{(1)}$ and then a bias $b^{(1)}$ is added

$$z^{(2)} = W^{(1)}X + b^{(1)}. \tag{1}$$

The activation $a^{(2)}$ of the first hidden layer is found when applying an *activation function* f to the signal $z^{(2)}$

$$a^{(2)} = f(z^{(2)}). \tag{2}$$

An activation function is a function that takes a real number roughly to the interval [0,1]. The output number of the function can be interpreted as how much a node 'lights up' or is activated. For an arbitrary hidden layer, the algorithm reads

$$z^{(k+1)} = W^{(k)}a^{(k)} + b^{(k)} \tag{3}$$

$$a^{(k+1)} = f(z^{(k+1)}). \tag{4}$$

After passing through all the hidden layers, a signal is sent to the output layer. This signal can be interpreted for regression or classification.

Before training the network, the output is likely rubbish. By training the network, we mean comparing the output of the network $\hat{y}$ with the real data y and updating the weights and biases in such a way that the error reduces. This calls for a definition of the error, or *cost* C. The best approximation $F(X)$ in the Euclidean space is the one who minimises the residual sum of squares to the data y we are fitting. We can therefore let it be the cost function

$$C = \frac{1}{2}\sum_{i=1}^{n}(y_i - F(X_i))^2, \tag{5}$$

where n is the number of datapoints. The factor of one half is there to make later calculations pretty. It is possible to add a regularisation term $\lambda \sum ||w||$ to the cost function which keeps the weights from possibly blowing up and overfitting the model. It sums the individual weights, and λ is just a coefficient. The mapping $F(X)$ with our neural network is dependent on the weights and biases. Therefore the cost function is dependent on the weights and biases. Knowing how to change these parameters, requires computation of gradients. For the output layer L, the gradients of the cost function with respect to the previous weights $W^L = w^L_{jk}$ and biases $b^L = b^L_j$ are

$$\frac{\partial C}{\partial w^L_{jk}} = \delta^L_j a^{L-1}_k, \tag{6}$$

$$\frac{\partial C}{\partial b^L_j} = \delta^L_j, \tag{7}$$

$$\delta^L_j = f'(z^L_j)\frac{\partial C}{\partial a^L_j}. \tag{8}$$

Calculating the derivative of the cost function with respect to the activation of the last layer is simple with our definition of the cost. With $a^L = F(X)$, it becomes

$$\frac{\partial C}{\partial a^L} = -y + a^L. \tag{9}$$

For an arbitrary layer l, the derivatives look like

$$\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k, \tag{10}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j, \tag{11}$$

$$\delta^l_j = \sum_k \delta^{l+1}_j {}^{l+1}_{kj} f'(z^l_j). \tag{12}$$

This way of working backwards in the network to compute gradients is called back-propagation. With all the gradients computed, one can update the weights and biases with gradient descent methods.

Gradient Descent Methods

Gradient descent (GD) methods are types of optimisation methods to either minimise or maximise a function [Goodfellow et al. 2016]. In our case, the goal is to find the global minima of a cost function. This minima will provide information about how well the neural network performs.

A parameter that needs to be defined in the GD method is the step size or *learning rate* η. It can be updated throughout the iterations by having a so called *learning schedule* or it can stay fixed. This will affect the results. The recursive formula for GD is

$$\theta_{i+1} = \theta_i - \eta \cdot \nabla_{\theta_i} C, \tag{13}$$

where θ_i is the parameters and $\nabla_{\theta_i} C$ is the gradient of the cost function in terms of the parameters. By moving in the negative direction of this gradient, we should lower the function value on the next step as long as we do not take too big of a step.

We can introduce an additional parameter to our algorithm which is called *momentum*. Momentum is a way to possibly speed up the convergence. When adding momentum, the GD algorithm will look like this:

$$\theta_{i+1} = \theta_i - (\eta \cdot \nabla_{\theta_i} C + \text{momentum} \cdot \theta_i). \tag{14}$$

Figure 2 [Goodfellow et al. 2016] shows an illustration of how the gradient descent method operates. The method halts when a minima of a function is reached.

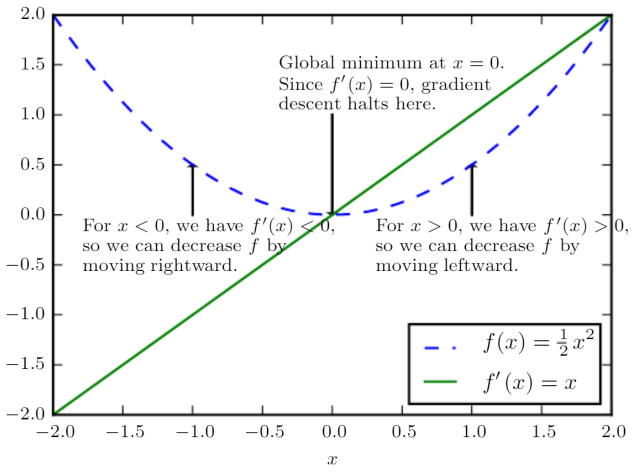

Figure 2: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minima.

In machine learning, one might be dealing with optimising cost functions that have many local minima. In practice, one usually settles for finding a minima that has a small value, even though it might not be the smallest existing value (global minima). Figure 3 [Goodfellow et al. 2016] illustrates three different points in a function: a global minima with the lowest existing cost value, a local minima with a low (but not lowest) cost value and

a local minima with a high cost value. The figure describes which of these points are ideal, which has an acceptable value and which has not.

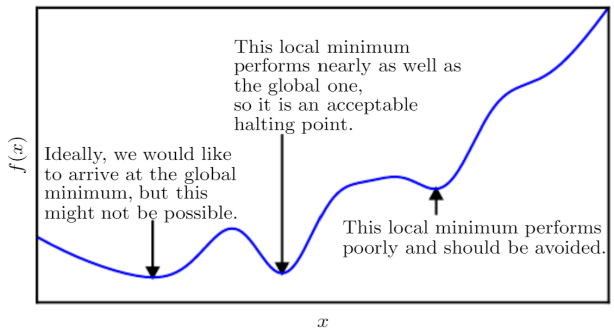

Figure 3: Approximate minimisation. The image illustrates what a function with one global and two local minima can look like. The ideal situation would be to arrive at the global minima (to the left), but it might not be possible. A local minima like the one in the middle could be and acceptable place to stop as it almost has the same value as the global minima. The local minima to the right could possibly have an error that is way bigger than the previous local minima, and would therefore be a bad place to arrive.

If a function contains several local minima, the standard gradient descent method might get stuck in a bad place that makes the algorithm perform poorly. A way to solve this is by introducing the *stochastic gradient descent* method.

Stochastic gradient descent (SGD) is a method that is especially good at two things: coping with huge amounts of data and preventing the algorithm to get stuck in a bad minima, though it can still happen. The fact that the method is stochastic will in most cases bring us closer to finding the true global minima of the cost function. The main difference between GD and SGD is that the latter divides the data into smaller subsets also called *mini-batches*. It is common to pre-define the desired amount of mini-batches. The mini-batch size (number of data points) is then defined as

$$\text{Mini-batch size} = \frac{\text{Total number of data points}}{\text{Number of mini-batches}}.$$

We also define an *epoch* to be one complete pass of the training dataset through the algorithm [Sarangam 2022]. That means that the time it takes to complete one epoch is the time it takes for all the mini-batches to pass through the algorithm or network one time.

Adaptive Learning Rates

A way to have a dynamic learning rate is to introduce *adaptive learning rates*. The idea is to adapt the step size individually for each model parameter. This means that a parameter that has a big influence on the results should get a decrease in learning rate, while a parameter with small influence should get an increase in learning rate. For the sake of simplicity we will from now on refer to adaptive learning rates as *optimizers*.

In the **AdaGrad** algorithm the parameters with the largest partial derivative of the cost function have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate [Goodfellow et al. 2016]. The mathematical expression goes as follows:

$$\delta_i = \delta_{i-1} + \nabla^2\theta_i \tag{15}$$

$$\theta_i = \theta_i - \frac{\eta}{\sqrt{\delta_i + \epsilon}}\nabla\theta_i \tag{16}$$

where δ_i is the cumulative sum of squared gradients [Kocagil 2022], θ is the parameters of (in our case) layer i, $\nabla\theta_i$ is the gradient of the parameters, η is the learning rate and ϵ is a scalar to prevent zero division.

A drawback of the AdaGrad algorithm is that it may shrink the learning rate too fast, which can cause the algorithm to arrive at a local and not optimal minima.

Unlike the AdaGrad method, the **RMSProp** algorithm uses an exponentially weighted moving average to discard history of the cumulated gradient. The method introduces a new hyperparameter ρ which is usually set to have a value of 0.9. We call this hyperparameter the *decaying term*. The mathematical expression goes as follows:

$$\delta_i = \rho\delta_{i-1} + (1-\rho)\nabla^2\theta_i \tag{17}$$

$$\theta_i = \theta_i - \frac{\eta}{\sqrt{\delta_i + \epsilon}}\nabla\theta_i. \tag{18}$$

The **Adam** method is similar to the RMSProp method. The difference is that it includes momentum and bias correction. The "momentum" term is a physical analogy to for instance a ball having momentum. In a simplified way, one can think of the ball having enough speed to be able to get out of a local minima of a function. The mathematical expression is:

$$\delta_{M_i} = \rho_1 \cdot \delta_{M_i} + (1-\rho_1)\nabla C_{\theta_i} \tag{19}$$

$$\delta_{V_i} = \rho_2 \cdot \delta_{V_i} + (1-\rho_2)\nabla^2\theta_i \tag{20}$$

$$\tilde{\delta}_{M_l} = \frac{\delta_{M_i}}{1-\rho_1} \tag{21}$$

$$\tilde{\delta}_{V_l} = \frac{\delta_{V_i}}{1-\rho_2} \tag{22}$$

$$\theta_i = \theta_{i-1} - \frac{\eta}{\sqrt{\tilde{\delta}_{V_l} + \epsilon}}\tilde{\delta}_{M_l} \tag{23}$$

where ρ_1 and ρ_2 are decaying parameters usually set to 0.9 and 0.999, respectively. δ_{M_i} is the first-moment decaying cumulative sum of gradients [Kocagil 2022], δ_{V_i}

is the second-moment decaying cumulative sum of gradients, $\tilde{\delta}_{M_l}$ and $\tilde{\delta}_{V_l}$ are bias-corrected values, θ_i is the parameters and η is the learning rate.

Unlike the RMSProp method, the Adam method is more robust to the choice of hyperparameters, but the learning rate sometimes needs to be changed [Goodfellow et al. 2016].

Figure 4 [Kocagil 2022] shows an example of how the different adaptive learning rates methods can behave. The point to show here is the pattern of their movement compared to each other.

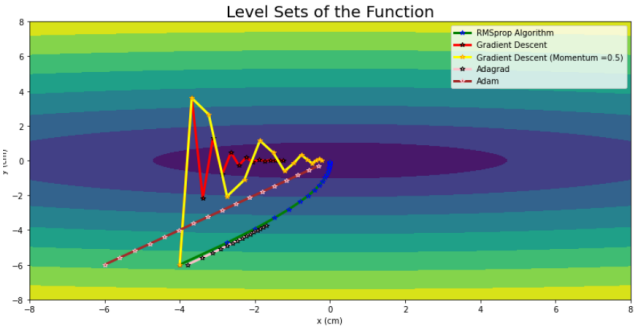

Figure 4: Comparison of adaptive learning rates methods. An example of the behaviour of the different methods. The main point is the pattern of the lines that each represent an adaptive learning rate method.

Regression and Classification

The signal z of the output layer is the prediction $\hat{y} = F(X)$ that the neural network makes based on the input data X. If we try to map input data to a continuous dependent variable y, we call the fitting a regression. In that case, we can leave the output of the network as it is.

If we try to map input data to a binary or multi-class dataset, we are dealing with a classification problem. In this case, we have to map the continuous output of the network to the classes. First, we use sigmoid functions for rescaling. Then, in the binary case of ones and zeros, one can take values lower than 0.5 to 0, and everything else to 1.

Logistic Regression

When the data we try to fit is categorical, linear regression fails because it is unbounded, meaning that the output can take any value. In this case a logistic model has proven to be useful. The idea is to fit a sigmoid curve to the data being studied, and setting a threshold value for rounding up to 1 or down to 0. In the binary case, 1 and 0 can represent *True* and *False* values, respectively. A common approach is to use the logistic function (24) with a threshold of 0.5,

$$f(x) = \frac{1}{1 + e^{-(\beta_1 x + \beta_0)}}. \tag{24}$$

Changing the parameters β_0 and β_1 will slide and stretch the sigmoid curve. These are the parameters that are fitted in the logistic regression model. How do we know which way to nudge these parameters? We need to define a cost function. For logistic regression, it is often set to

$$C(\beta) = -\frac{1}{n}\sum_{i=1}^{n}\Big(y_i \log\left(f(x_i)\right) + (1 - y_i)\log 1 - f(x_i)\Big). \tag{25}$$

This is the function we want to minimise. The gradient of the cost function can be written in vectorized form like such

$$\frac{\partial C}{\partial \beta} = X^T(f(x) - y). \tag{26}$$

Now, we can use gradient descent methods to minimise the cost function.

Evaluation

It is always important, when making predictions, to evaluate the results. It is common practice to split the dataset into a training set and a test set. The training set is only used for training the network, and the test set is only used for evaluating it. A rule of thumb is the split the train and test data into parts of 80% and 20% respectively. Once the training is done, we can pass the test data through the network and compare the output $\hat{y}$ with the measured data y.

When doing regression analysis, two ways of measuring how much the predicted values deviates from the real values are the mean squared error (MSE) and the R^2 score. The MSE is defined as

$$MSE(\boldsymbol{z}, \tilde{\boldsymbol{z}}) = \frac{1}{n}\sum_{i=0}^{n-1}(z_i - \tilde{z}_i)^2 \tag{27}$$

where $\boldsymbol{z}$ is the real values and $\tilde{\boldsymbol{z}}$ is the predicted values. A high MSE is considered bad. If MSE is equal to zero it means that the model is a perfect fit.

The R^2 score is defined as

$$R^2(\boldsymbol{z}, \tilde{\boldsymbol{z}}) = 1 - \frac{\Sigma_{i=0}^{n-1}(z_i - \tilde{z}_i)^2}{\Sigma_{i=0}^{n-1}(z_i - \bar{y})^2} \tag{28}$$

where $\bar{\boldsymbol{z}}$ is the mean value of $\boldsymbol{z}$. R^2-scores usually range from 0 to 1. A high R^2-score is considered good. An R^2-score of 1 indicates that the regression model fit the data perfectly.

For classification problems, one way of evaluating the results is to use accuracy score. The accuracy score is the percentage of correct predictions.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{29}$$

An accuracy score of 100% indicates that the network classifies the data perfectly.

Method

Regression

Regression analysis was done on data generated with the Franke function. The function was evaluated on a 20×20 meshgrid with x and y values equally spaced between 0 and 1. This resulted in a data-array z with 400 datapoints. Gaussian noise with mean 0 and standard deviation 0.1 was added to the dataset. The dataset was split into train data (80%) and test data (20%). The datasets were rescaled with min-max normalization. The goal was to find a good model for this data. Three different approaches were tried.

The first approach was to use the ordinary least squares (OLS) method to find the optimal fitting parameters θ for a given design matrix. The design matrix consisted of polynomial terms as illustrated below

$$X = \begin{pmatrix} 1 & x_1 & y_1 & x_1^2 & y_1^2 & x_1 y_1 & \dots \\ 1 & x_2 & y_2 & x_2^2 & y_2^2 & x_2 y_2 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\ 1 & x_n & y_n & x_n^2 & y_n^2 & x_n y_n & \dots \end{pmatrix}.$$

The code from project 1 was reused for this method. Design matrices with polynomials of different degrees were fitted to the train data. No resampling techniques were used. For each polynomial degree, the MSE was calculated on the test data. This exact method was chosen because it gave one of the best results in project 1.

The second approach was to replace the OLS method with a gradient descent method to find the best fitting parameters for the design matrices. The methods that were tested was standard gradient descent (GD) and stochastic gradient descent (SGD). Both methods were additionally tested with AdaGrad, RMSProp and Adam as optimizers. Momentum and a regularization parameter was also implemented in all methods.

The code to the right shows an example of how gradient descent was implemented. Here, the parameters θ were calculated using standard gradient descent with AdaGrad for optimization. Notice that gradient clipping was performed to prevent, or at least delay, the problem of exploding gradients and overflow.

```
# AdaGrad
eta = 0.01  # learning rate
iterations = 1000
delta = 1e-8  # parameter to avoid possible
↪  zero division
tol = 10**4  # tolerance for gradient
↪  clipping
change = 0  # for momentum part

# initial guess for parameters
theta = np.random.randn(len(X[0]),1)

# storing the cumulative gradient
Giter =
↪  np.zeros((np.shape(X)[1],np.shape(X)[1]))

for i in range(iterations):

    # calculate the gradient
    gradient = gradientFunc(X,y,theta,n)

    # gradient clipping
    Giter += np.minimum(gradient @
    ↪  gradient.T, tol)

    # take the diagonal elements
    Ginverse = np.c_[eta / (delta +
    ↪  np.sqrt(np.diag(Giter)))]

    # update parameters
    change = np.multiply(Ginverse,gradient) +
    ↪  momentum * change
    theta -= change

    # calculate MSE
    y_predict = np.dot(X, theta)
    cost[i+1] = MSE(y, y_predict)
```

The third approach was to use the gradient descent methods again, but this time implemented in a neural network. The neural network was not given a design matrix like the previous methods. It was given a 20×2 matrix with the x-values in the first column and the y-values in the second. This dataset was split into train and test data, and they were rescaled. The neural network operated in the following way:

1. Train data was fed forward through the network resulting in a prediction as output.
2. The output was fed into the cost function giving a certain cost. The gradient of the cost function with respect to the different weights and biases was calculated.

3. Gradient descent methods then updated each weight and bias based on how much they influenced the network.

4. The process was repeated a certain number of times in hope of arriving at the global minima of the cost function or at least at a local minima with an accepted cost value.

Different architectures for the neural network was tried. Varying the number of hidden layers, number of nodes in each layer and calculating the error would give us an idea of how to structure the architecture for further tests. We start off with a few nodes and hidden layers, typically one layer with two nodes, and increase the complexity. The challenge is to find a model that is complex enough to give a good fit for the data, but not so complex that it enters the region of overfitting.

Different learning rates were also tested to show their influence, and momentum was varied to see whether there was something to it. Would it change the convergence rate?

One neural network was trained with three different activation functions. The activation functions that were tested were the standard logistic function

$$f(z) = \frac{1}{1+e^{-z}}, \tag{30}$$

the ReLU function

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

and the Leaky ReLU function

$$f(z) = max(0.01z, z). \tag{32}$$

Classification

Neural networks can also be used for classification problems. We studied the Wisconsin Breast Cancer dataset [UCI 2022]. This dataset consisted of 30 features of cancer cell nucleuses such as mean radius and mean area. For each patient, the features and the diagnosis was recorded.

Our neural network was easily repurposed for classification by applying a standard logistic function to the output, giving it a value between 0 and 1 (included). With a threshold of 0.5, the value was either rounded up to 1 or down to 0. We let malignant be 1 and benign 0. With now 30 features, the input layer size was updated to 30. The classification model was evaluated by calculating the accuracy score (29). The neural network was tuned by adjusting the architecture and hyperparameters.

To compare our neural network to another classification method, a logistic regression code was made for the same dataset. This model was optimised with the standard gradient descent method. The learning rate was tuned to find the best predictions.

Results and Discussion

Franke Function without Noise

The Franke function was used to make two datasets; one with noise and one without noise. The mean squared error between these two sets were calculated. This was repeated 1000 times and the average MSE was calculated to 0.00454. This was done to give us an idea of how big the error is with a model that is 100% correct in the sense that it is the model we are looking for.

Regression with OLS

Linear regressions were made for the Franke dataset. A design matrix with polynomial terms were fitted to the data with the ordinary least squares method. No resampling techniques were used. Different models were made by varying the degree of the polynomials fitted from 4 to 10. For each model, the MSE was calculated. This was repeated 1000 times and an average of each MSE was calculated. The results are listed in table 1.

Ordinary Least Squares

Polynomial Degree	4	5	6	7	8	9	10
MSE	0.01041	0.00775	0.00767	0.00780	0.00780	0.00790	0.00834

Table 1: The table shows the polynomial degree of the fitted OLS models and the corresponding mean MSE of 1000 runs.

The polynomial fit of degree 6 was on average the best of the OLS models. Its average MSE was 0.00767 which, for our intents and purposes, is an accepted amount of error. For further analysis of the dataset with neural networks, this is the MSE that we want to beat.

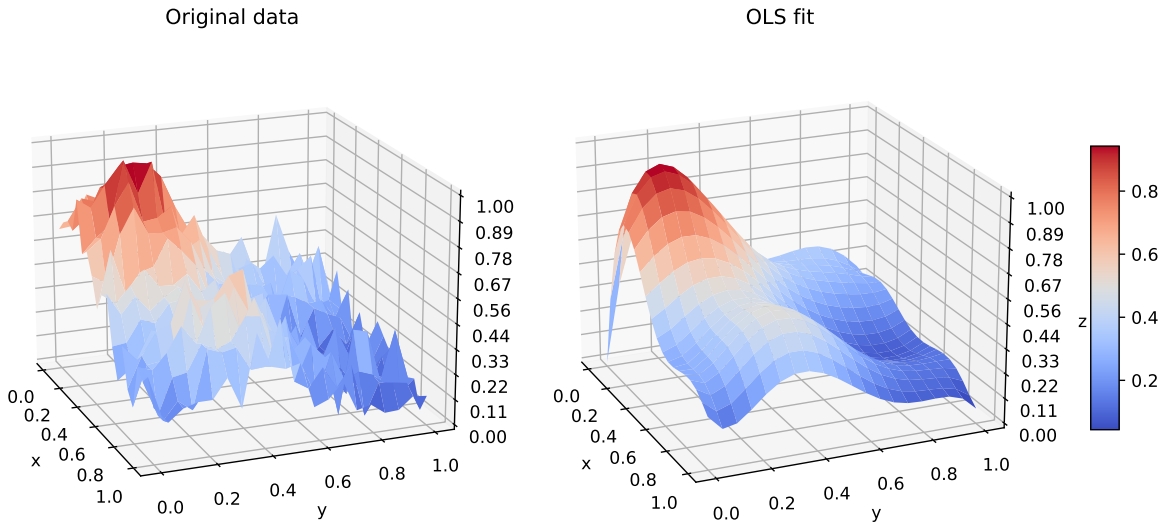

Figure 5: The figure shows a surface plot of the Franke data we are trying to model, and a surface plot of the OLS polynomial fit of degree 6.

Regression with Gradient Descent Methods

Table 2 and 3 shows the MSE of the test Franke data as a result of regression performed with gradient descent and stochastic gradient descent with different optimizers. The smallest MSE was approximately 0.0180 and was obtained by SGD with a fixed learning rate of 0.001 and the Adam optimizer. The Adam optimizer also gave the best result for the GD method, but the MSE was almost three times bigger than with SGD. For all cases tested, the best results were obtained with the regularization parameter and momentum equal to zero. The reason might be the fact that the data didn't get overfitted. If it had, the regularization could have given a better result as it introduces a penalty to the sum of all weights.

Gradient Descent, $\eta = 0.001$

Optimizer	**MSE**
None	0.397
AdaGrad	0.303
RMSProp	0.261
Adam	0.0450

Table 2: The table shows the MSE obtained with gradient descent and different optimizers. Adam was the method that gave the smallest MSE.

Stochastic Gradient Descent, $\eta = 0.001$

Optimizer	**MSE**
None	0.0283
AdaGrad	0.0226
RMSProp	0.0215
Adam	0.0180

Table 3: The table shows the MSE obtained with stochastic gradient descent and different optimizers. Adam was the method that gave the smallest MSE.

In our case we would say that an MSE of 0.0180 is ok, but we would still like to see this MSE be reduced. Even though it is an accepted MSE, it is almost three times as big as the MSE obtained with OLS. Remember that the type of MSE that is accepted depends on the purpose of the model. If a model predicts the chances of surviving a medical procedure, the model should have a MSE close to 0. If the model is used to predict the grade you will get in terms of how much you read, then the accepted MSE would most likely be higher.

Figure 6 and 7 shows how the training MSE changes in terms of epochs when using different optimizers and a fixed learning rate η that equals 0.01 and 0.001, respectively. Notice the change in the Adam optimizer (red graph). With $\eta = 0.01$, the MSE starts increasing with epochs at some point. With $\eta = 0.001$, the MSE for the Adam optimizer decreases for each epoch just like the other optimizers.

It is therefore important to test different learning rates when training a neural network. If the step size is too big, one might skip past a good minima of the cost function. On the other hand, if the step size is too small, one might get stuck in a bad minima, or it might take a lot of iterations and computation time to reach the minima.

Note that the plots are slightly zoomed in. All the optimizers start off with the same MSE.

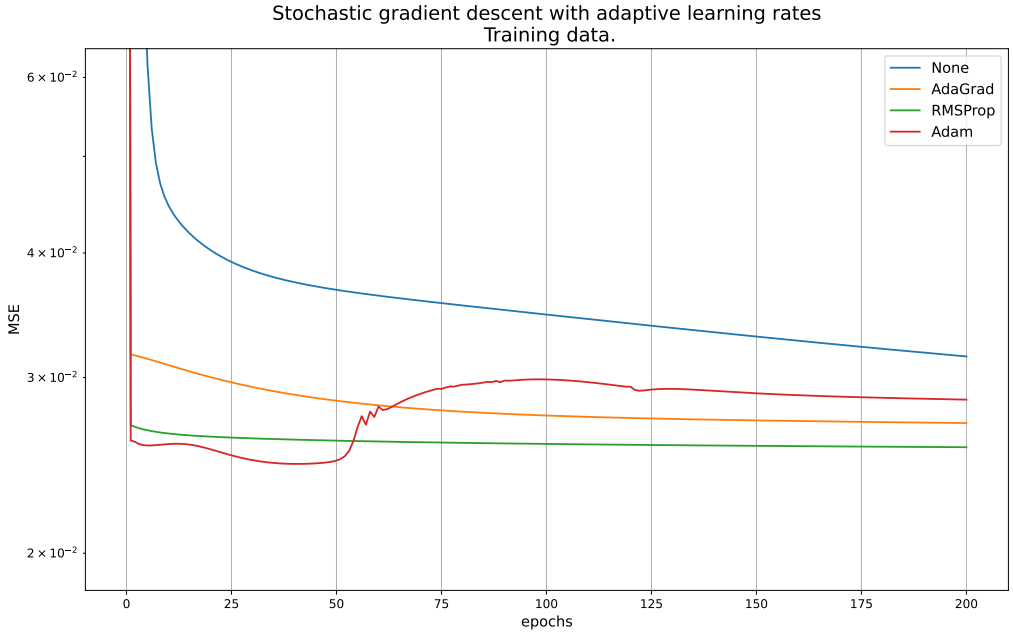

Figure 6: The figure shows the MSE in terms of epochs for different optimizers used with the SGD method with a fixed learning rate $\eta = 0.01$. The MSE of all optimizers decreases with epochs except for the Adam optimizer.

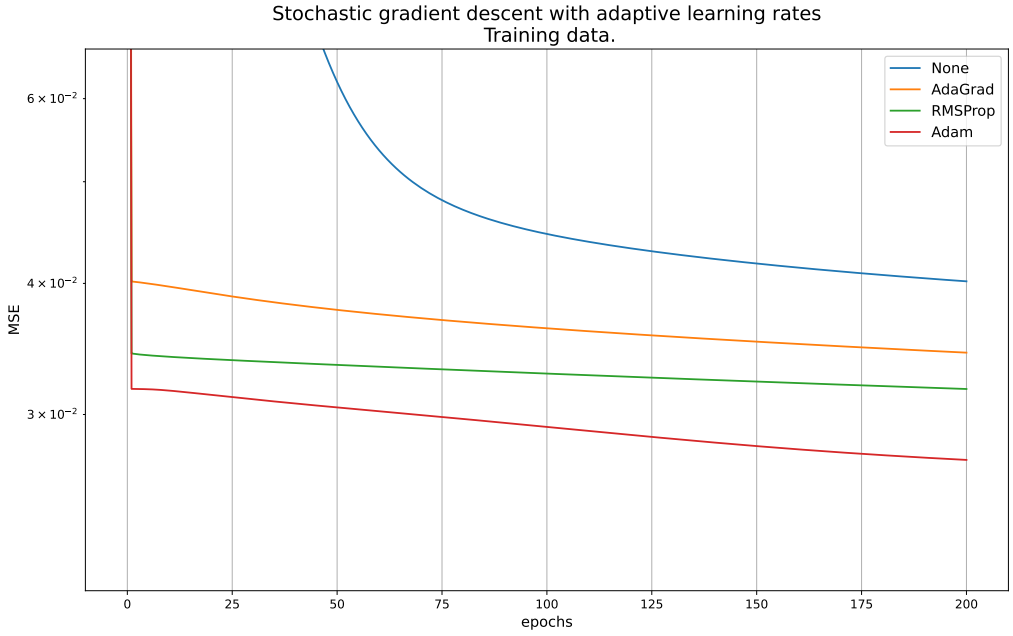

Figure 7: The figure shows the MSE in terms of epochs for different optimizers used with the SGD method with a fixed learning rate $\eta = 0.001$. The MSE of all the optimizers decreases with epochs.

Regression with Neural Network

Different neural networks with varying number of nodes and hidden layers were trained to model the Franke data. The number of hidden layers ranged from 1 to 3. The number of nodes in each layer ranged from 5 to 30. The number of nodes in each layer were the same for runs with more than one hidden layer. Figure 8 shows a heatmap of the MSE of each trained neural network. All the networks for this figure used stochastic gradient descent with 100 epochs and a mini-batch size of 5 datapoints. The learning rate was set to 0.01.

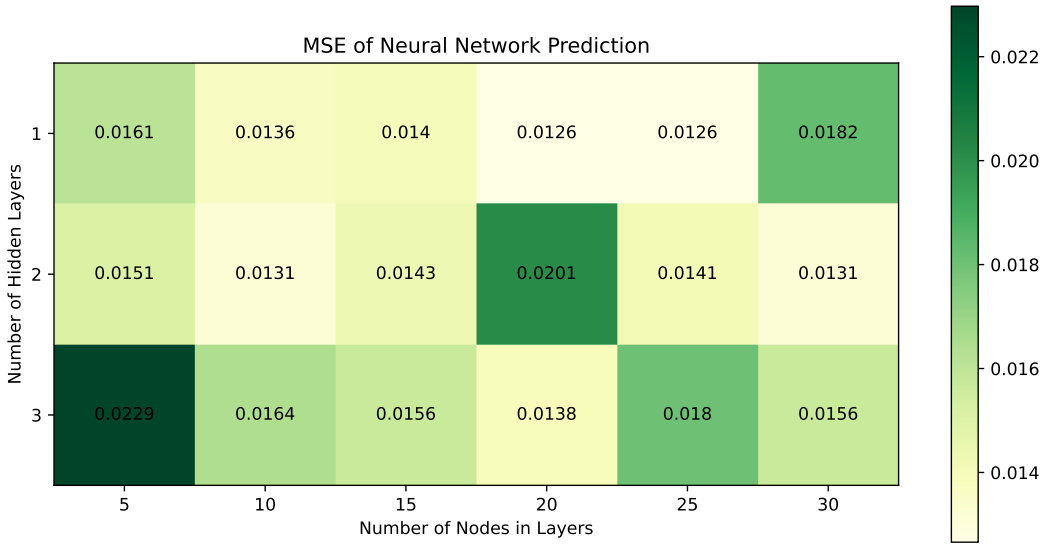

Figure 8: The figure shows the MSE of trained neural networks with different number of nodes and hidden layers. In the case of multiple hidden layers, all layers have the same amount of nodes. The lowest MSEs had a value of 0.0126 and was obtained with 1 hidden layer consisting of 20 and 25 nodes. The highest MSE was obtained with 3 hidden layers and 5 nodes.

From figure 8, we can see that the neural networks performed well for almost all cases. For 4 out of 6 cases with 1 hidden layer the MSE was 0.014 or lower. For the neural networks with 2 hidden layers, only 2 out of the 6 had a MSE of 0.014 or below. For neural networks with 3 hidden layers, it was only 1 out of the 6. This shows that neural networks with only one hidden layer worked best. It seems that increasing the number of hidden layers does not necessarily improve the performance of neural networks. In one way, this is fortunate as upping the number of hidden layers increases computation time.

The networks with 1 hidden layer and 20 and 25 nodes performed best with a MSE of 0.0126. This gives us some idea of how many nodes to use moving forward with training the networks. With 1 hidden layer and 5 nodes, the network gave a MSE of 0.0161. That is 28% worse than the networks with 20 to 25 nodes. By designing a neural network with few nodes, one might limit the networks ability to capture the features real influence on the data we are trying to model. On the other hand if the network has too many nodes, overfitting might become a problem.

Neural networks with different learning rates were trained on the Franke data. These networks had 1 hidden layer with 15 nodes. The results are shown in figure 9.

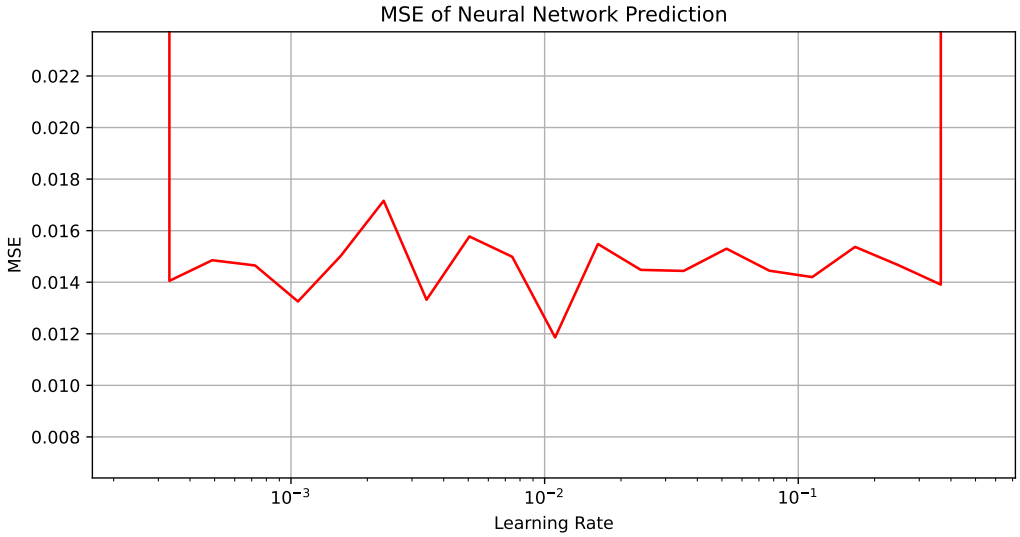

Figure 9: The figure shows the MSE of a neural network with 1 hidden layer and 15 nodes, when varying the learning rate η. The network used standard gradient descent with Adam for optimisation.

From figure 9, we can see the interval of learning rates that gives the best MSE for one type of network. If the learning rate is too large, the network keeps overshooting when trying to converge to local minima of the cost function. This causes the MSE to go through the roof. If the learning rate is to small, the network cannot converge fast enough when searching for the minima. From the figure, we can see that we should keep our learning rate between 10^{-3} and 10^{-1} to be safe. Note that we only know that this result applies to this certain network, but it can gives us some idea of which scale to chose the learning rate from later on.

Neural networks with different momentums were trained on the Franke data. These networks had 1 hidden layer with 6 nodes. Learning rate was 0.01. The results are shown in figure 10. The same neural network was trained without momentum for comparison.

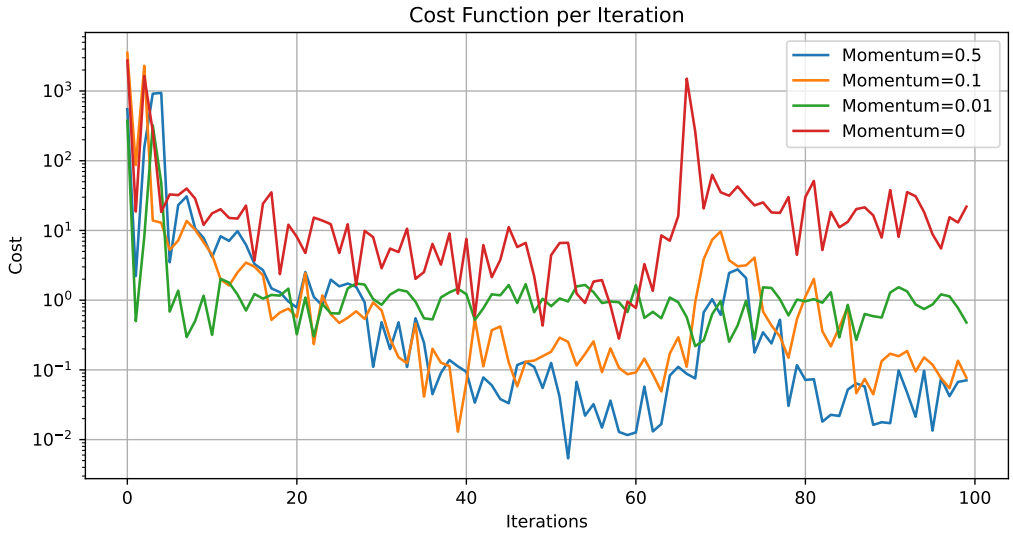

Figure 10: The figure shows the train cost between at each iteration. Four networks were trained, each with a different momentum. The figure shows only the 100 first iterations.

From figure 10, we can see that adding momentum to the algorithm can help the network by converging to a local minimum faster. We see that increasing the momentum, speeds up the convergence.

Up until now, the networks have only used the logistic function for activating the layers. Here, neural networks were trained with different activation functions. The functions tested were the standard logistic function, ReLU and leaky ReLU. All these neural networks had the same architecture with 1 hidden layer and 6 nodes. The networks used stochastic gradient descent with a mini-batch size of 5 and 100 epochs. The learning rate was set to 0.001. The results are listed in table 4.

Activation Function	**MSE**
Logistic	0.0117
ReLU	0.00841
Leaky ReLU	0.00584

Table 4: The table shows the MSE of trained neural networks with different activation functions.

From table 4 we can see that switching activation function from the logistic function to ReLU dramatically reduces the MSE of the fitted model. The MSE was reduced 28% by going from 0.0117 to 0.00841 with ReLU. It is now apparent that the choice of activation function have deep impact on the network's performance. By implementing the leaky ReLU, the MSE was reduces even more to 0.00584. The network with leaky ReLU performed a lot better than the polynomial fit with linear regression with MSE of 0.00767.

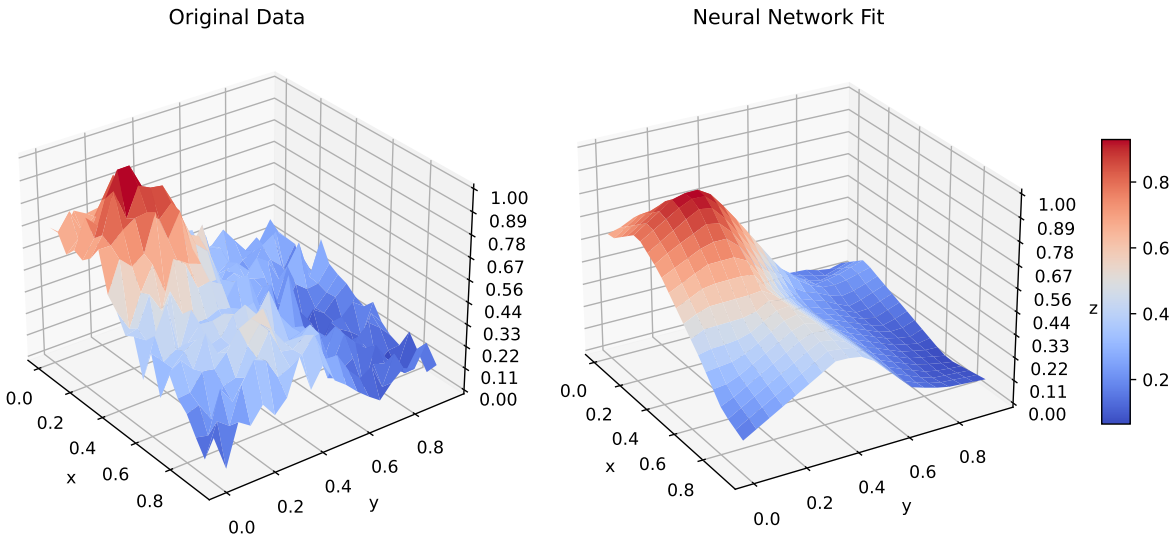

Figure 11: The figure shows a surface plot of the Franke data, and a surface plot of the neural network fit with leaky ReLU as activation function. The MSE of the prediction was 0.00584.

We have now seen how the three modelling approaches worked. Both the OLS method and the neural networks managed to find solutions that gave acceptable mean squared errors. The gradient descent method for finding the fitting parameters of a design matrix started to approach the OLS solutions when done stochastic with ADAM for optimising, but still lagged behind. We could never expect them to beat the OLS fit of course as this is the best approximation to the train data.

Classification with Neural Network

The neural network was repurposed for classification. The accuracy of the predictions were calculated for neural networks with different architecture. The number of hidden layers and number of nodes in each layer were varied. The learning rate was set to 0.01, and it used stochastic gradient descent with mini-batch size 10 and 100 epochs. The results are shown in figure 12.

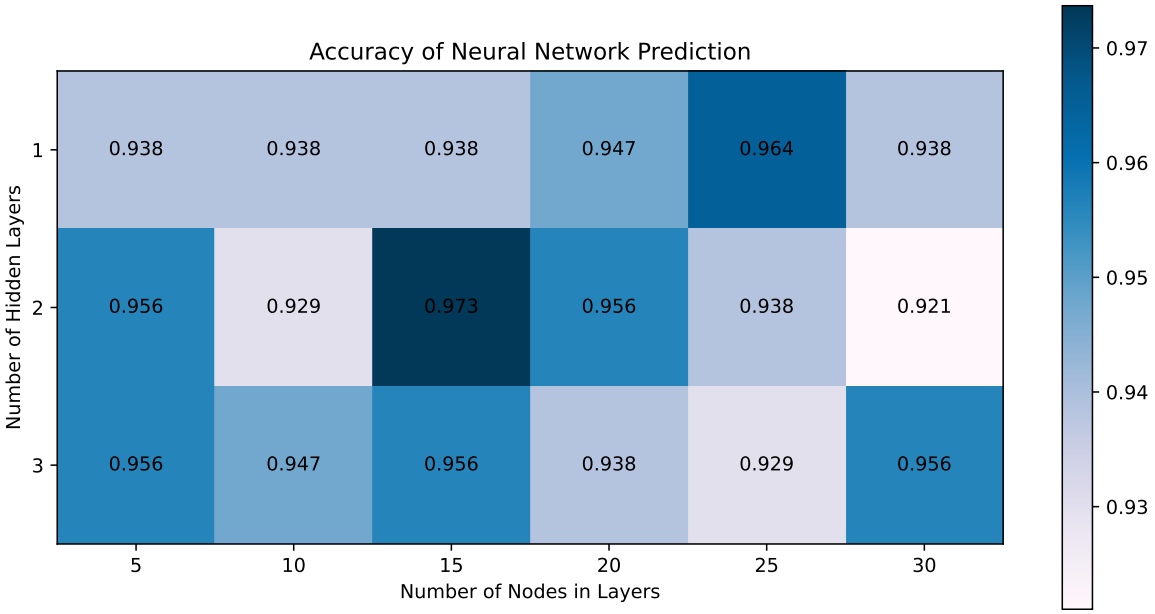

Figure 12: The figure shows the accuracy score of the neural network with different architectures. The number of hidden layers and the number of nodes in each hidden layer was varied.

From figure 12, we can see that the neural network with the highest accuracy had 2 hidden layers and 15 nodes in each layer. The accuracy was over 97% which we are very happy with. This network could work as a pointer for doctors. All of the 18 networks tested gave an accuracy of at least 92% which is very good. This shows that the neural network is a robust method of classifying the dataset. The model could probably be improved

Classification with Logistic Regression

A logistic regression fit was made to the dataset. The fit was optimised with the standard gradient descent with a varying learning rate. The results are shown in figure 13.

The best results with logistic regression were obtained with a learning rate in the region around 10^{-1}. Here, the accuracy score was consistently 94.5%. This is also a good accuracy score. We would happily use either of the methods for classification studied. But the neural network is the winner with 97.3% accuracy.

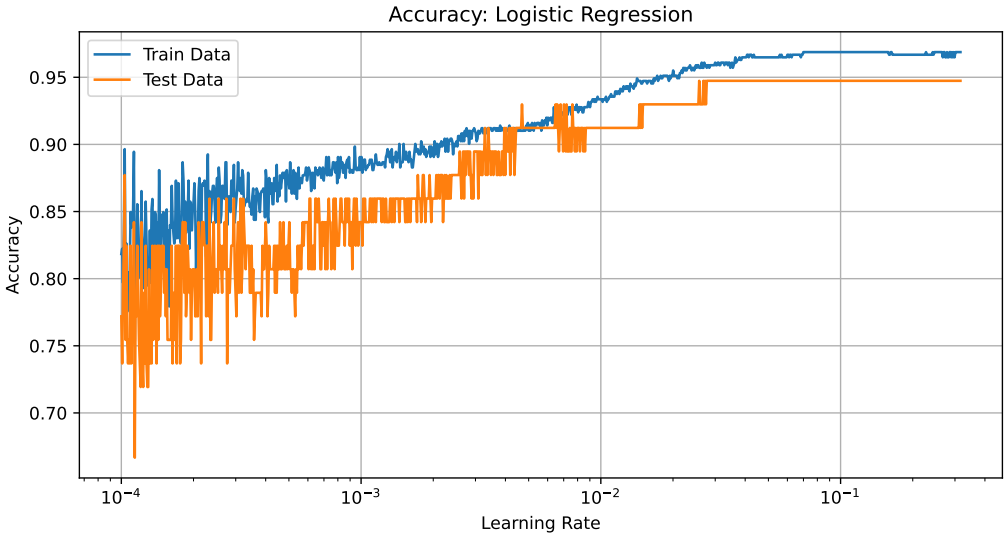

Figure 13: The figure shows the accuracy score of the logistic regression fit for different values of learning rate.

Conclusion

In this project, we wanted to see if a neural networks could accurately model a dataset generated with the Franke function. Both the ordinary least squares method and the neural network gave an accepted mean square error for our regression analysis. There were loads of different parameters to tune and methods to choose, but the very best result was obtained by using the neural network with the leaky ReLU activation function, stochastic gradient descent, learning rate at 0.001 and the Adam optimizer. The MSE was 0.00584. The polynomial of degree 6 was the best of the OLS models with a MSE of 0.00767. The gradient descent method for finding the best fitting parameters to a design matrix approached the OLS solution, but not enough to be comparable to the other methods. With the best GD model with an MSE=0.0180, gradient descent for fitting design matrices did not yield good enough results.

We also wanted to study the binary Wisconsin Breast Cancer dataset. For this classification problem, both the neural network and the logistic regression method managed to predict outcomes very accurately. Their best models were 97.3% and 94.5% accurate, respectively. The neural network classifier could probably be improved by doing the same testing as for the regressor; varying learning rate, momentum, optimizers, etc.

With the acquired results in this project, it is clear that neural networks are a good approach to modelling these kind of regression and classification problems.

Bibliography

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Kocagil, Can (2022). *Neural Network Optimizers from Scratch in Python*. URL: https://towardsdatascience.com/neural-network-optimizers-from-scratch-in-python-af76ee087aab (visited on 7th Nov. 2022).

Sarangam, Ajay (2022). *Epoch in Machine Learning: A Simple Introduction*. URL: https://www.jigsawacademy.com/blogs/ai-ml/epoch-in-machine-learning (visited on 7th Nov. 2022).

UCI (2022). *Breast Cancer Wisconsin (Diagnostic) Data Set*. URL: https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data?resource=download (visited on 13th Nov. 2022).