University of Oslo

FYS-STK4155

Helle Bakke

Classification and regression with neural networks

November 9, 2019

Contents

Abstract

Abstract

1 Introduction

Machine learning is a fairly new concept to most people, but its early history dates back to 1943 when Warren McCulloch and Walter Pitts introduced the first neural network (Mayo et al., 2018). In the following decade, the Turing test was made known, and in 1958, Frank Rosenblatt designed the first artificial neural network called the *Perceptron*. However, it was not until the early 1980s the interest in neural networks gained new momentum. In 1986, the *back propagation* algorithm was developed, an important step in any neural network. Since then, the world has seen a large increase in both data and computing power. These factors have contributed to the rapid progress in machine learning that we see today.

The aim of this project is to study classification and regression problems by developing our own neural network and comparing it to a logistic regression code and the linear regression code developed in Project 1. We will use a credit card data set from UCI for the classification problem. This data set is well studied, which is useful when validating our models. For the linear regression problem we will revisit the Franke function from Project 1.

The structure of the report is as follows. In section 2, we start by explaining the data preprocessing, before taking a deep dive into logistic regression methods and the machine learning algorithms that go into the feed-forward neural network. The results are presented in section 3. Section 4 contains a discussion of our main findings, and section 5 gives the concluding remarks.

2 Method

2.1 Data preprocessing

The studied data set contains information on default payments, credit data, payment history, bill statements and other demographic factors of credit card clients in Taiwan from April 2005 to September 2005 (Yeh and Lien, 2009). The data set contains 25 variables, but if we remove the ID of the client, we are left with 23 explanatory variables and the response variable. Each

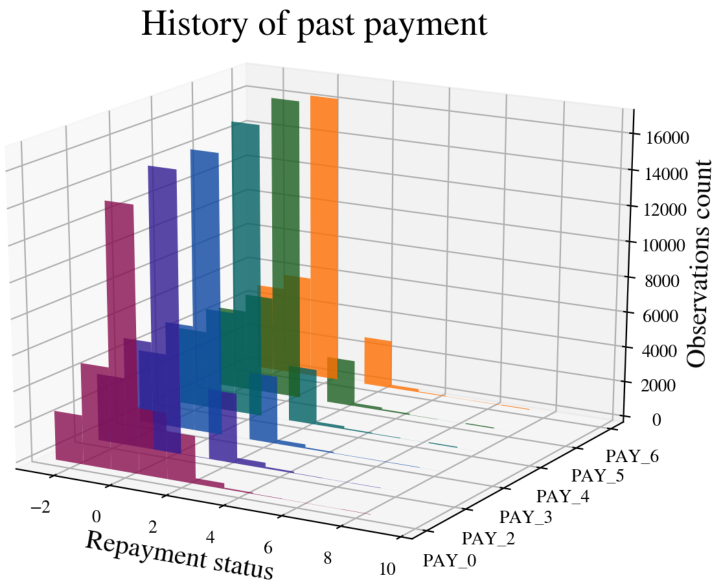

Figure 1: Histograms showing the history of past payment for the credit card holders.

variable contains 30000 observations, but not all of them are relevant. If the amount of bill statements and/or the amount of previous payments are 0, we remove the observations from all features of that client. Additionally, we remove uncategorised observations as long as they are not a major class. A categorical feature is an explanatory varibale with levels. An example is the feature MARRIAGE where 1=married, 2=single and 3=others. In this data set, however, many of the categorical features of the data set include levels that are not specified. Figure 1 shows histograms of the history of past payments, and shows that level 0 is a major class. Normally, this would be fine, but in this data set level 0 is undocumented. Most undocumented values can safely be removed, but in this case that would mean loosing the majority of observations. Hence, we decide to keep the observations as an NA class.

After stripping the data set for observations, we are left with 28000 observations. We define features and targets, where the features are normalised using the standard scaler from scikit-learn. For the classification neural network, we one-hot encode the categorical features and the targets. The advantage of one-hot encoded categorical features is to remove assumed dependencies between categories. The categorical feature MARRIAGE can be represented as

$$\begin{bmatrix} 1 \\ 2 \\ 2 \\ \vdots \\ 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ \vdots & & \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

while the binary targets can then be represented as

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ \vdots & \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

In principle, all categorical features should be one-hot encoded. However, since the history of past payment features have 12 levels each, we find it too extensive to one-hot encode them (this would lengthen the x-dimension of the feature matrix by $12 \times 6!$). The matrix gets sparse, and the computation time inceases.

2.2 Logistic regression

Logistic regression is used when dealing with a categorical dependent variable, or *target*. In classification problems, the targets take the form of discrete variables such as categories. In many cases the target is binary, meaning that it can be either yes or no, true or false, positive or negative etc. The credit card data set that we are exploring has binary targets (0,1). Rewrite??

Logistic regression is an important first step in order to understand the neural network algorithms and how supervised deep learning works. In lo-

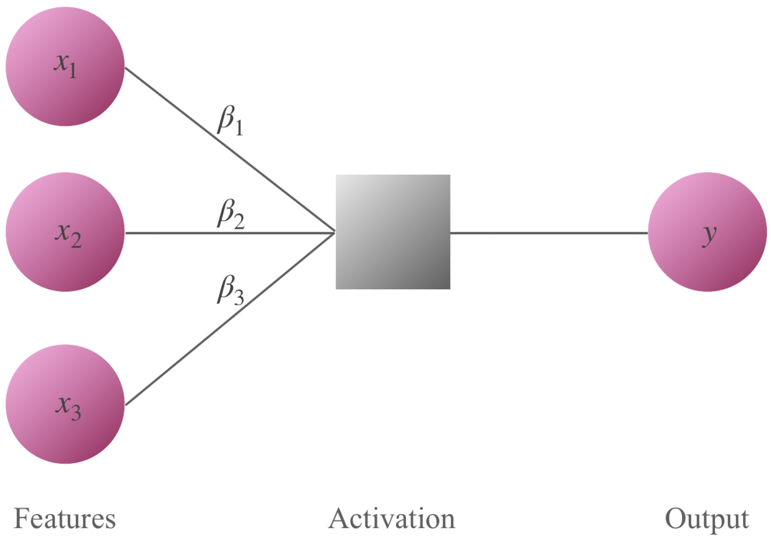

Figure 2: Simple illustration of a logistic regression model.

gistic regression, we want to find β such that the cost function $C(\beta)$ is minimised. This leads to gradient descent methods, which are important parts of nearly all machine learning algorithms. Mention that GD is an optimizer?

Figure 2 shows a simple logistic regression model with three features. The β-values of each feature are passed to an *activation function*, before the output is given. In the following, we go into the details of setting up a logistic regression model.

2.2.1 Cost function

In order to find the β-values that minimise $C(\beta)$, we need to define the cost function. The target of our classification problem is binary, hence we start by defining the probability, or likelihood, for a given event. The probability of a data point belonging to a category $y = \{0, 1\}$ is given by the Sigmoid function

$$p(\theta) = \frac{1}{1+e^{-\theta}} = \frac{e^{\theta}}{1+e^{\theta}} \tag{1}$$

where θ is the prediction of y (in logistic regression, this reduces to the linear equation $\tilde{\hat{y}} = \hat{X}\beta$). Further, the probabilities of y_i being either 0 or 1 can be expressed as

$$p(y_i = 1|\hat{X}^i, \hat{\beta}) = \frac{e^{\beta_0 + x_1^i\beta_1 + x_2^i\beta_2 + \ldots + x_p^i\beta_p}}{1 + e^{\beta_0 + x_1^i\beta_1 + x_2^i\beta_2 + \ldots + x_p^i\beta_p}}$$
$$p(y_i = 0|\hat{X}^i, \hat{\beta}) = 1 - p(y_i = 1|\hat{X}^i, \hat{\beta})$$

where p is the number of predictors. Next, we use the above probabilities to define the log-likelihood

$$C(\hat{\beta}) = -\sum_{i=1}^{n}(y_i \text{log} p(y_i = 1|\hat{X}^i, \hat{\beta}) + (1 - y_i)\text{log}[1 - p(y_i = 1|\hat{X}^i, \hat{\beta})]) \quad (2)$$

which is the cost function. In statistics, this equation is known as the *binary cross-entropy*, where y is the target (either 0 or 1) and $\sigma(y)$ is the predicted probability of the data point being 1. In example, we see that when $y_i = 1$, the cost reduces to

$$C(\hat{\beta}) = -\sum_{i=1}^{n} \text{log} p(y_i = 1|\hat{X}^i, \hat{\beta}),$$

and when $y_i = 0$ the cost reduces to

$$C(\hat{\beta}) = -\sum_{i=1}^{n} \text{log}[1 - p(y_i = 1|\hat{X}^i, \hat{\beta})].$$

2.2.2 Gradient descent

Gradient descent is an optimisation algorithm used to find the local minima of a function. The idea is to minimise $C(\hat{\beta})$, where $\hat{\beta}$ is a vector with elements for each feature value (Marsland, 2014). We want to find a sequence of new points $\hat{\beta}(i)$ that move towards a solution. This is done by taking the derivative of the cost function. However, the direction of the derivative is not arbitrary. In steepest gradient descent, we always choose to go downhill as fast as possible for each point. This leaves us with the equation

$$\hat{\beta}_{k+1} = \hat{\beta}_k - \eta \nabla_\beta C(\hat{\beta}) \quad (3)$$

where η is the leraning rate and the derivative of the cost function is

$$\nabla_\beta C(\hat{\beta}) = -\hat{X}^T(\hat{y} - \hat{p}) \quad (4)$$

Here, $\hat{X}$ is the feature matrix, $\hat{y}$ is a vector of the targets and $\hat{p}$ is a vector of the fitted probabilites (Sigmoid function). A drawback of the steepest gradient descent method is that many of the directions that $\hat{\beta}$ travel are directly towards the centre (local minima).

In general, gradient descent methods have limitations, and we can address some of the shortcomings by considering the stochastic gradient descent (SGD) method. We keep equation (3) as it is, but add stochasticity by randomly shuffle the data. Stochastic gradient descent converges faster for larger data sets, but since the method uses one example at a time, the computations can slow down (Patrikar, 2019). By adding mini-batches to the model, we can vectorise the SGD, making the computations faster. Additionally, the gradient is computed against more training samples, which in turn means that it is averaged over more training samples. This may then lead to a smoother convergence.

For each *epoch*[1], we loop over all the mini-batches and calculate $\hat{\beta}_{k+1}$ using equation (3). After we have calculated $\hat{\beta}_{k+1}$ for the training samples in all the mini-batches, we randomly shuffle the data before continuing to the next epoch. The process is repreated until we reach the end of the last epoch.

To summarise and connect the above theory to Figure 2, the logistic regression model for classification problems is as follows:

- Divide the data into features and targets.
- Split the data into training and test samples.
- Feed a randomly generated $\hat{\beta}$-value to the activation layer, where we for each epoch calculate $\hat{\beta}_{k+1}$ for the training samples in all the mini-batches. Randomly shuffle the training data.
- Calculate the prediction of $\hat{y}$.

2.3 Neural network

A neural network is a collection of neurons that can learn to recognise patterns in data. In biological terms, a neuron is a nerve cell. It is the processing units of the brain, and each neuron is a separate processor performing simple tasks (Marsland, 2014). An artificial neuron sums the incoming signals, and

[1] A single iteration over all the data.

an activation function/threshold decides whether or not an output is given. If the threshold is not overcome, the neuron has zero output (Hjorth-Jensen, 2019).

In this project, we are creating a feed-forward neural network. That means that the information moves forward through the layers. Figure 3 shows a Multi-layer Perceptron (MLP) with an input layer consisting of n neurons, one hidden layer consisting of 4 neurons and an output layer consisting of 2 neurons. An MLP is a fully-connected feed-forward neural network with three or more layers, and consists of neurons with non-linear activation functions (such as the Sigmoid function). Training the MLP consists of finding the outputs given the inputs and current weights and biases, and then updating the weigths and biases according to the output error. In the following, we will take a deeper dive into the mathematical model comprising the Multi-layer Perceptron.

2.3.1 Feed-forward pass

The feed-forward pass refers to the process of passing values from the input neurons through the network, layer by layer, until the output layer is reached. The activation values of each neuron in each layer is defined as

$$z_j^l = \sum_{k=1}^{N_{l-1}} w_{jk}^l a_k^{l-1} + b_j^l \tag{5}$$

where $\hat{w}^l$ are the weights, $\hat{a}^{l-1}$ are the forward passes/outputs from the previous layer, $\hat{b}^l$ are the biases and N_{l-1} represents the total number of neurons in layer $l-1$. With the activation values from the current layer, we can define the output $\hat{a}^l$ of the current layer as

$$\hat{a}^l = f(\hat{z}^l) \tag{6}$$

where the function f is the *activation function*. In the logistic regression model, we used the Sigmoid function in equation (1) as our activation function. This is a non-linear activation function, and we can use it in order to obtain the output of each neuron in each layer such as

$$a_j^l = f(z_j^l) = \frac{e^{z_j^l}}{1 + e^{z_j^l}} \tag{7}$$

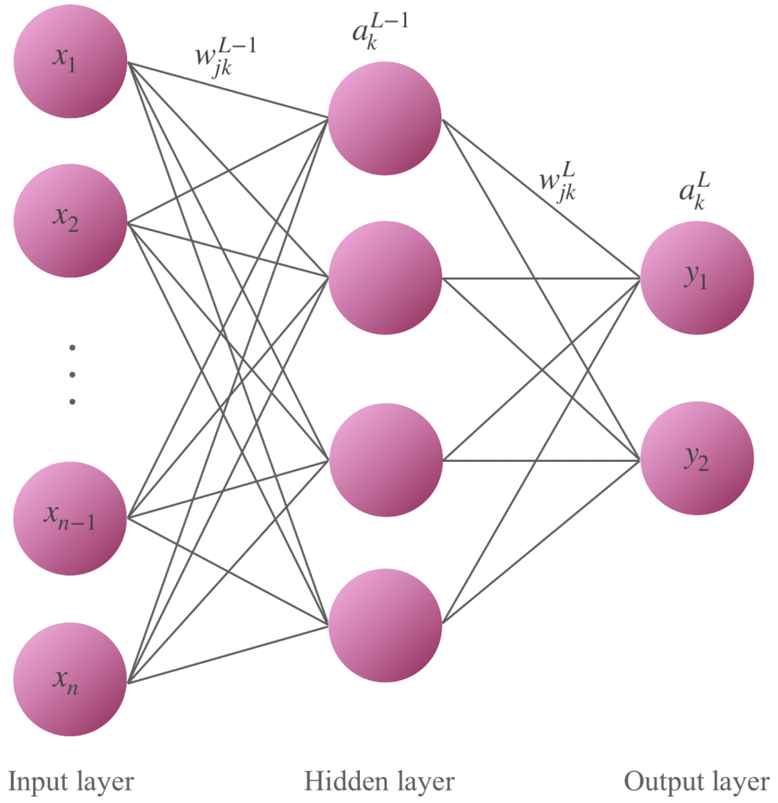

Figure 3: Simple illustration of a feed-forward neural network with one hidden layer.

In the case of linear regression, we change the activation function from a Sigmoid function to a Rectified Linear Unit (ReLU). The ReLU activation function is defined as

$$f(z_j^l) = \max(0, z_j^l), \tag{8}$$

and is linear for all positive values of z_j^l and zero for all negative values of z_j^l. More??

2.3.2 Back propagation

Back propagation refers to the algorithm used to train a feed-forward neural network. Back propagation computes the gradient of the cost function with respect to the weights of the network. In the following section, we derive the back propagation algorithm by first defining the cost function. Then, we compute the gradients and error terms, before updating the weights and biases.

For the classification problem, we use binary cross-entropy defined as

$$C(\hat{W}^L) = -\sum_{k=1}^{N_L} (y_k \text{log} a_k^L + (1 - y_k)\text{log}[1 - a_k^L]) \tag{9}$$

where y_k is the target and a_k^L is the output from neuron k in layer L (the output layer). The derivative of the cost function with respect to the weights can be written as

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \frac{\partial C(\hat{W}^L)}{\partial a_k^L} \frac{\partial a_k^L}{\partial w_{jk}^L} \tag{10}$$

where we have applied the chain rule. We can apply the chain rule to last derivative on the right-hand side as well, obtaining

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \frac{\partial C(\hat{W}^L)}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L} \frac{\partial z_k^L}{\partial w_{jk}^L} \tag{11}$$

Now, we have three derivatives that we can solve. The first derivative on the right-hand side of equation (11) is solved as

$$\begin{aligned}\frac{\partial C(\hat{W}^L)}{\partial a_k^L} &= -\frac{y_k}{a_k^L} + \left(\frac{1-y_k}{1-a_k^L}\right)\\ &= \frac{a_k^L(1-y_k)}{a_k^L(1-a_k^L)} - \frac{y_k(1-a_k^L)}{a_k^L(1-a_k^L)}\\ &= \frac{a_k^L(1-y_k)-y_k(1-a_k^L)}{a_k^L(1-a_k^L)}\\ &= \frac{a_k^L-y_k}{a_k^L(1-a_k^L)}\end{aligned}$$

The second derivative on the right-hand side of equation (11) can be solved as

$$\begin{aligned}\frac{\partial a_k^L}{\partial z_k^L} &= f'(z_k^L)\\ &= a_k^L(1-a_k^L)\end{aligned}$$

The last term on the right-hand side of equation (11) is just

$$\frac{\partial z_k^L}{\partial w_{jk}^L} = a_k^{L-1}$$

Next, we insert the separate solutions of the derivatives into equation (11) and obtain

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = a_k^{L-1}(a_k^L - y_k) \tag{12}$$

We can then define the output error

$$\delta_k^L = f'(z_k^L)\frac{\partial C(\hat{W}^L)}{\partial a_k^L} = a_k^L - y_k \tag{13}$$

This will be the starting equation of the back propagation algorithm. The error terms of the hidden layers now be defined as

$$\delta_j^l = f'(z_j^l)\frac{\partial C(\hat{W}^l)}{\partial a_j^l} = \frac{\partial a_j^L}{\partial z_j^l}\frac{\partial C(\hat{W}^l)}{\partial a_j^l} = \frac{\partial C(\hat{W}^l)}{\partial z_j^l}$$

In addition, the error can also be interpreted in in terms of biases b_k^L, such as

$$\delta_j^l = \frac{\partial C(\hat{W}^l)}{\partial b_j^l}\frac{\partial b_j^l)}{\partial z_j^l} = \frac{\partial C(\hat{W}^l)}{\partial b_j^l}$$

However, we want to express the error in terms of the equation for layer $l+1$. This can be done by employing the chain rule so that

$$\delta_j^l = \sum_k^{N_{l+1}} \frac{C(\hat{W}^l)}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k^{N_{l+1}} \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l}$$

From equation (5), we see that the derivative $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} f'(z_j^l)$. The hidden layer error can finally be written as

$$\delta_j^l = \sum_k^{N_{l+1}} \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{14}$$

After computing the errors of the hidden layers, the weights and biases are finally updated using stochastic gradient descent as

$$w_{jk}^l \leftarrow w_{jk}^l - \eta\delta_j^l a_k^{l-1} \tag{15}$$

and

$$b_j^l \leftarrow b_j^l - \eta\delta_j^l \tag{16}$$

For the linear regression neural network model, we define the cost function as the Mean Squared Error

$$C(\hat{W}^L) = \frac{1}{2}\sum_{k=1}^{N_L}(y_k - a_k^L)^2 \tag{17}$$

where the derivative is given as

$$\frac{\partial C(\hat{W}^L)}{\partial a_k^L} = a_k^L - y_k \tag{18}$$

The derivative of the ReLU activation function is defined as

$$f'(z_k^L) = \begin{cases} 1 & \text{if } z_k^L > 0 \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

2.4 Regularisation

The model is overfitted when the network performs well on training data, but poorly on the test data. The network has a high variance, and can not generalise well the data it has been trained on (Peixeiro, 2019). One way of addressing overfitting is to add regularisation to the cost function. In addition to minimising the cost, we also want to penalise the large weights by adding an L_2 regularisation term. We can redefine any cost function as

$$C(\hat{W}^L) = C(\hat{W}^L) + \frac{\lambda}{2N_L}||\hat{W}^L||^2 \tag{20}$$

where λ is a regularisation hyper-parameter that can be tuned. If the weights are large, they will be penalised more if λ is large as well. On the other hand, if λ is small, the effect of the regularisation decreases. Because the size of the weights are reduced, the effect of the activation function decreases and a less complex function is fitted to the data.

Since we have added regularisation to the cost function, we need to reformulate the update of the weights. The derivative of the cost function then becomes

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^l} = \delta_j^l a_j^{l-1} + \lambda w_{jk}^l \tag{21}$$

where the weights are updated as

$$w_{jk}^l \leftarrow w_{jk}^l - \eta(\delta_j^l a_k^{l-1} + \lambda w_{jk}^l) \tag{22}$$

2.5 Model evaluation

The performance of the classification models are measured using an *accuracy score*. The accurcy score is defined as

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{23}$$

where t_i represents the target and y_i represents the output of either the logistic regression code or the classification neural network. The function I is an indicator function, and is defined as

$$I = \begin{cases} 1 & \text{if } t_i = y_i \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

This means that a classifier that is able to predict all the targets has an accuracy of 100%.

The model performance of the linear regression neural network is measured using the R^2 score given by

$$R^2(t_i, y_i) = 1 - \frac{\sum_{i=1}^{n}(t_i - y_i)^2}{\sum_{i=1}^{n}(t_i - \bar{y}_i)^2} \tag{25}$$

where $\bar{y}_i$ is the mean value of the predicted variable.

Further, we can use the accuracy score to verify that the neural network is learning. By reducing the data set to a few samples and train the neural network on this, the training set should immediately reach an accuracy of 100% (the training set is overfitted). At the same time, the accuracy of the test set should go to 0%. As a second test, we keep the full data set, but shuffle the labels. By doing this, the neural network can only learn by memorising the training set, and the training cost decrease slowly. The test cost will increase quite rapidly, until it reaches the random chance test cost[2].

The above tests are relatively quick and easy to implement while developing the code, and work as a way of benchmarking the neural network. However, the best method of validation is to compare the obtained results against a similar code. In this project, we have tried to replicate our neural network with tensorflow/keras.

2.6 Code

In the previous project, we discussed bootstrapping as a resampling method. In this project, we use k-fold cross-validation as the main assessment method. The training and test data are randomly partitioned into k subsamples. The model is then trained on each of the training subsamples, and tested on the test subsamples. The variance is reduced since we are using different cross-validation training and test sets.

All developed codes in this project utilises k-fold cross-validation. The input features X and targets y are randomly shuffled and split into training and test subsamples. For each k-fold, we loop over epochs, and for each epoch we calculate the new weights ($\hat{\beta}$ for logistic and linear regression, $\hat{w}^l$ for neural networks) using stochastic gradient descent with mini-batches (see

[2] The *Golden Tests* have been found at stackexchange.

Section 2.2.2). Below is a pseudocode of the *k*-fold cross-validation of the neural network.

```python
import numpy as np
from sklearn.model_selection import KFold

X, y = data_preprocessing()

# shuffle X and y
random_index = np.arange(X.shape[0])
np.random.shuffle(random_index)
X = X[random_index,:]
y = y[random_index,:]

kfolds = KFold(n_splits=n_folds)

train_index = []
test_index  = []
for i_train, i_test in kfolds.split(X):
  train_index.append(np.array(i_train))
  test_index.append(np.array(i_test))

for k in range(n_folds):
  # split into training and test data
  X_train = X[train_index[k]]
  y_train = y[train_index[k]]

  X_test = X[test_index[k]]
  y_test = y[test_index[k]]

  # define weights and biases
  weights_biases()


  for j in range(len(epochs)):
    for i in range(0,X_train.shape[0],minibatch_size):
      feed_forward(X_train[i:i+minibatch_size,:])
      backpropagation(y_train[i:i+minibatch_size,:])

  # prediction from training and test data
  feed_forward(X_train)
  feed_forward(X_test)
```

3 Results

3.1 Classification with logistic regression

The performance of our logistic regression model is measured using the accuracy score in equation (23). In order to find the input parameters that give the highest accuracy score, we experiment with different values of the learning rate η. Figure 4 shows the accuracy score for values of $\eta \in [10^{-4}, 10^{0}]$. We have used 10-fold cross-validation, with 100 epochs and 100 minibatches. When the learning rate is small, the model converges slowly. In the figure, we see that in order to obtain a higher accuray score, we need to run the code for more epochs. Oppositely, when the learning rate is large, the model converges faster, and we risk overfitting the model if the number of epochs is large. However, there exists a middle ground where the model learns at a relatively good pace. This happens when $\eta = 0.001$. We observe that the number of epochs can be reduced by approximately half, as the accuracy flattens around 80%.

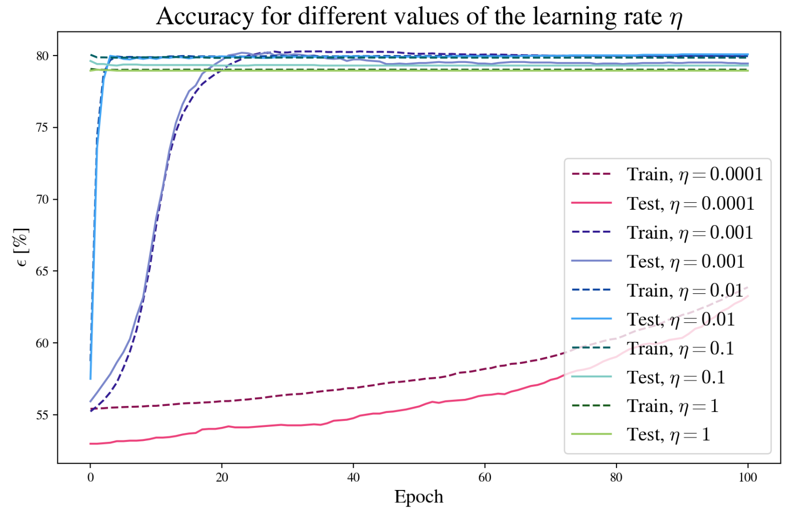

Figure 4: Accuracy of the logistic regression model for different values of the learning rate η.

Results of experiments with different mini-batch sizes are shown in figure 5. When we decrease the size of the mini-batches, we decrease the amount of data that goes into each batch. As a result, we loop over a larger amount of mini-batches every epoch, slowing down the computation time. The figure clearly illustrates that larger mini-batch sizes do not affect the accuracy of the model negatively. For the sake of computation time, we can therefore safely use a mini-batch size of 150.

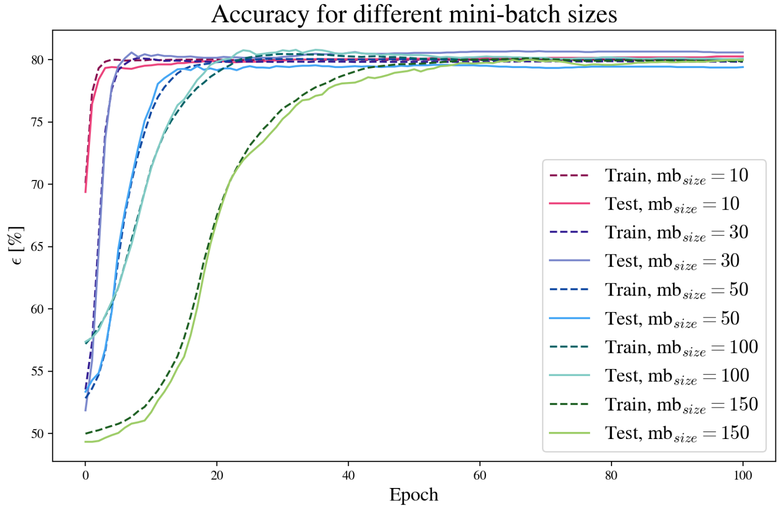

Figure 5: Accuracy of the logistic regression model for different mini-batch sizes.

Based on the results of figures 4 and 5, we run the logistic regression model with $\eta = 0.001$, 60 epochs, 150 mini-batches and 10 folds. For each k-fold, we check the last accuracy score against the previous ones to find the maximum and minimum accuracy from the cross-validation. The results are shows in figures 6 and 7. Figure 6 is a direct illustration of the maximum and minimum accuracy for the training and test data, while figure 7 shows the cost as a function of epochs for the maximum and minimum accuracy. Both figures show that as the number of epochs increase, the accuracy and cost converges to their maxima and minima, respectively.

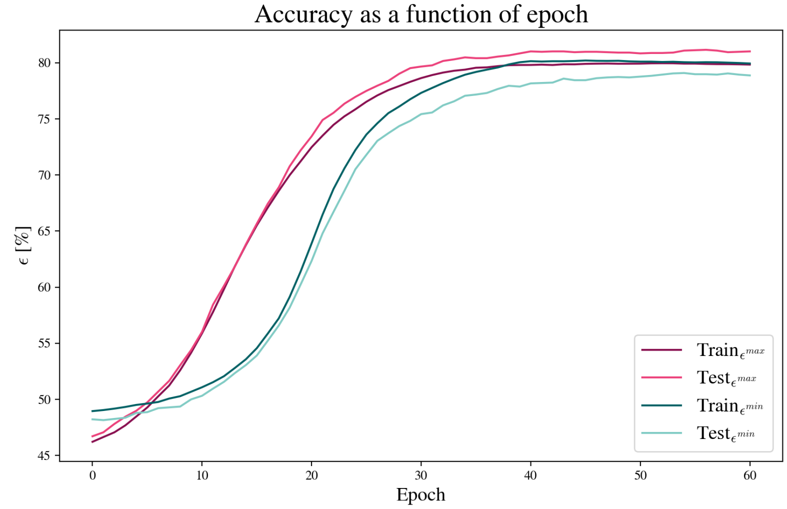

Figure 6: Maximum and minimum accuracy from the k-fold cross-validation of the logistic regression model.

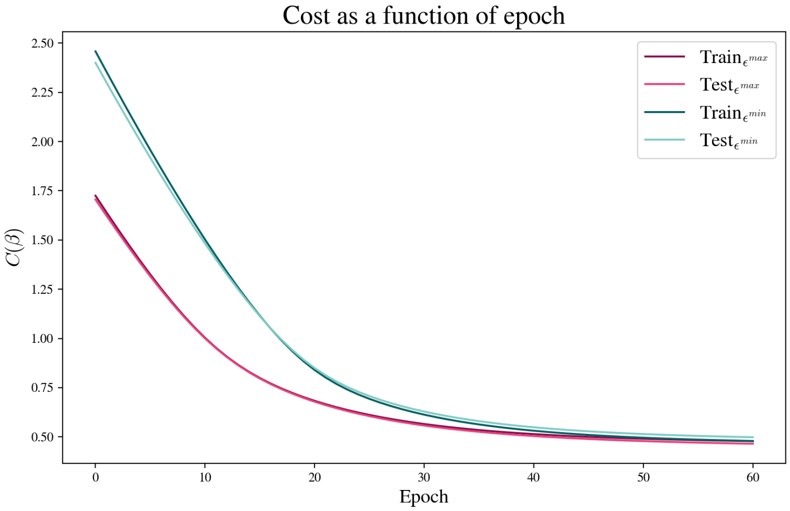

Figure 7: Cost as a function of epochs for the logistic regression model. The cost originates from the same iterations as the maximum and minimum accuracy scores.

We verify our model by comparing it to a Stochastic Gradient Descent Claasifier (SGDClassifier) from the scikit-learn python library. Figure 8 shows the training and test accuracy from both models, where the last training and test accuracies from each k-fold have been stored. There are relatively small differences between the two models, but according to the figure, the SGDClassifier from scikit-learn is able to produce better accuracies.

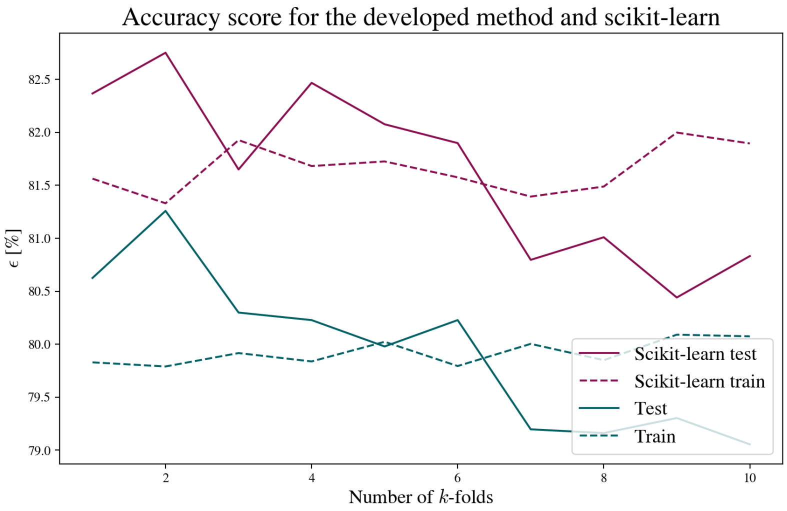

Figure 8: Comparison of the developed logistic regression model and scikit-learn.

3.2 Verification of the neural network

We verify that the neural network is able to learn by training the network with few samples. The result is seen in figure 10, where the neural nwtwork immediately overfits the training set. The training accuracy goes to 100% immediately, while the test accuracy stays at 0%. Figure 10 shows the accuracy of the neural network for a larger portion of the data set. The labels are shuffled, and the only way for the neural network to learn is by memorising the training set. The test set should reach the target ratio, which is ±70% (since we are shuffling the labels, the target ratio varies from run to run). From the figure, we see that this indeed happens, and we are confident that our neural network works properly.

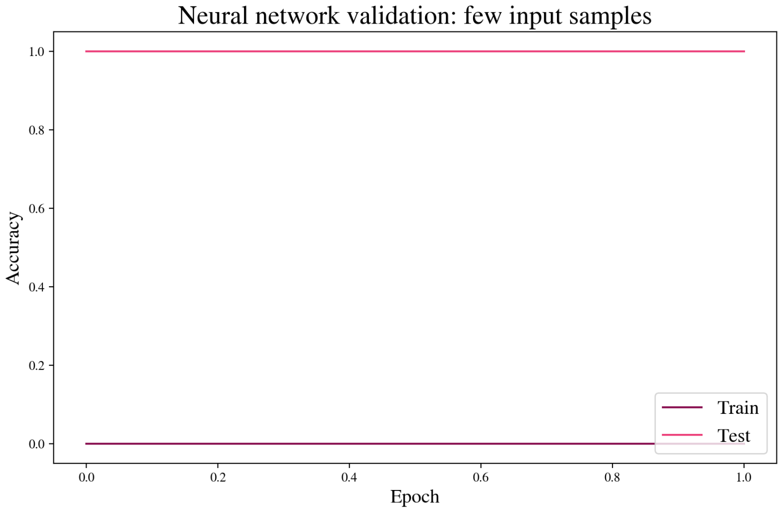

Figure 9: Accuracy of neural network with few samples.

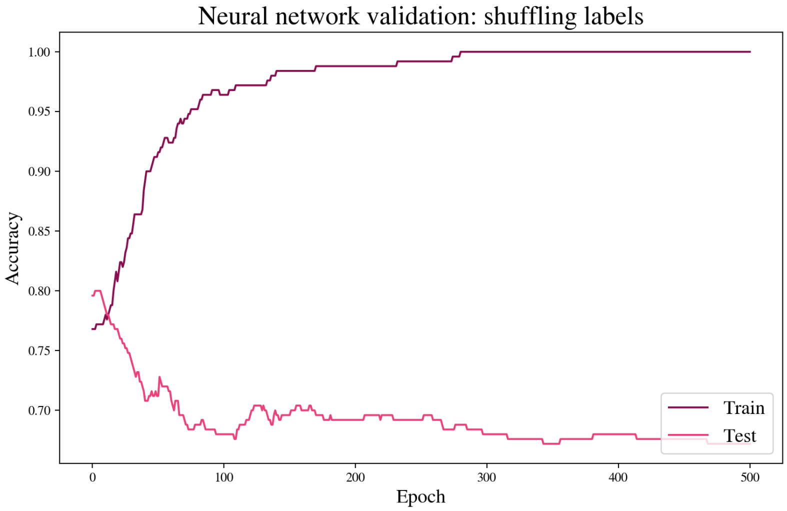

Figure 10: Accuracy of neural network with shuffling of the labels.

3.3 Classification with neural network

Again, we use the accuracy score in (23) to measure the performance of our neural network. As with the logistic regression model, we want to find the input parameters that gave the highest accuracy score. Figure 11 shows the accuracy scores for different values of λ. The model was run with a constant learning rate $\eta = 0.01$, 100 minibatches, 50 epochs and 10 folds for the cross-validation. The figure shows that increasing values of λ decreases the accuracy, and vice versa. When $\lambda = 0.0001$, we obtain the largest accuracy score. As a matter of fact, the number of epochs does not appear to be sufficient enough in order to obtain the highest accuracy score.

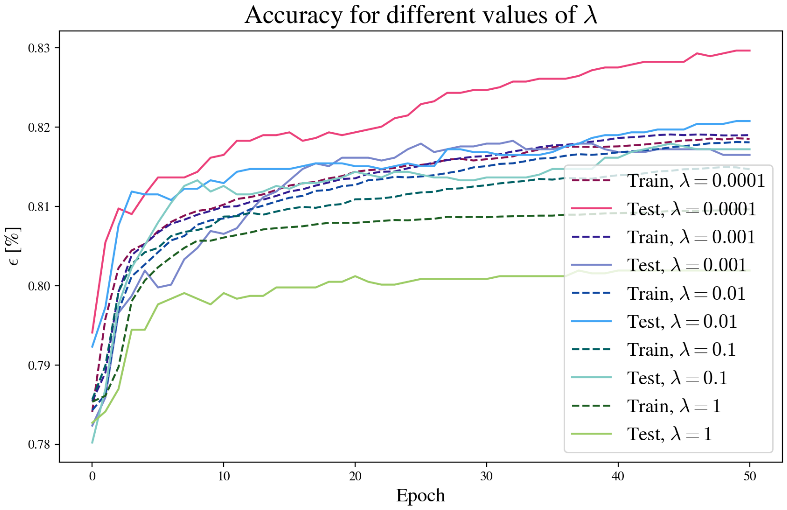

Figure 11: Accuracy score of the neural network for different regularisatiomn hyper-parameters λ.

Further, we explore the effects of changing the learning rate η. We exclude the regularisation hyper-parameter, and set mini-batch size to 100, number of epochs to 50 and use 10-fold cross-validation. The result is seen in figure 12. By employing the smallest learning rate, the accuracy it kept more or less constant for the first $8-9$ epochs. Additionally, smaller learning rates increase the computation time, hence we dismiss this value. Larger learning rates produce higher accuracies, but we also note that they accuracy is nearly constant over all epochs. Based on the figure, it seems safe to settle at a middle ground with $\eta = 0.01$.

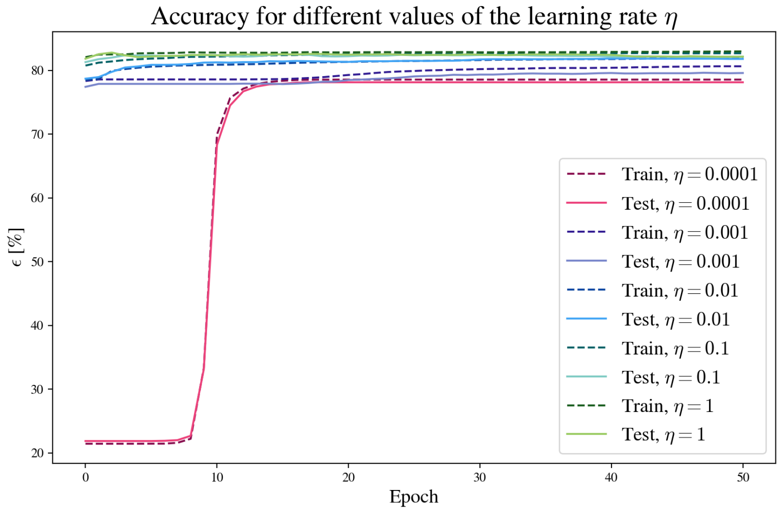

Figure 12: Accuracy of the neural network for different values of the learning rate η.

Finally, we observe what happens when we vary the size of the mini-batches, using a constant learning rate $\eta = 0.01$, zero regularisation, 50 epochs and 10-fold cross-validation. Figure 13 shows that there are larger differences in the accuracy scores compared to figure 5. We should aim at using smaller mini-batch sizes, but still keep in mind that this increases the computation time. According to the figure, a mini-batch size of 50 should yield a relatively good accuracy score.

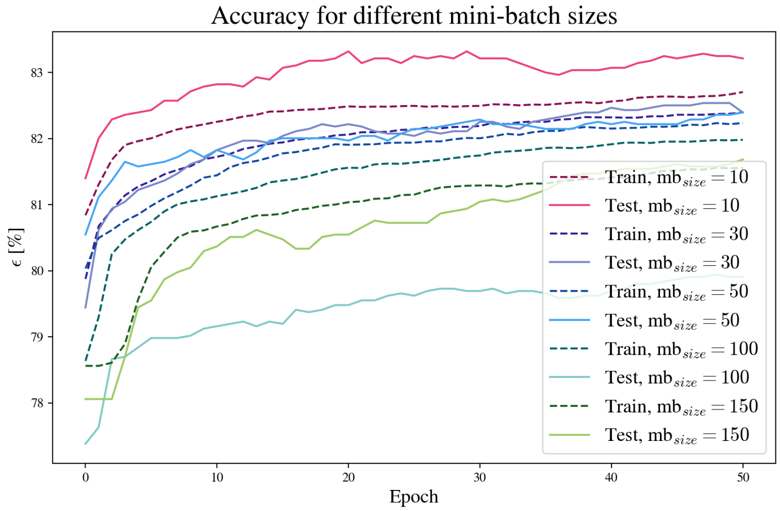

Figure 13: Accuracy of the neural network for different mini-batch sizes.

Lastly, the number of hidden layers and neurons have been tested using trial and error, For the classification neural network, we have use one hidden layer with 50 neurons. Based on the above assessment of the input parameters, we run our network with $\eta = 0.01$, $\lambda = 0.0001$, a mini-batch size of 50, 100 epochs and 10-fold cross-validation. Figures 14 illustrates the maximum and minimum accuracy from the cross-validation, and figure 15 shows the cost from the same iterations as the accuracy.

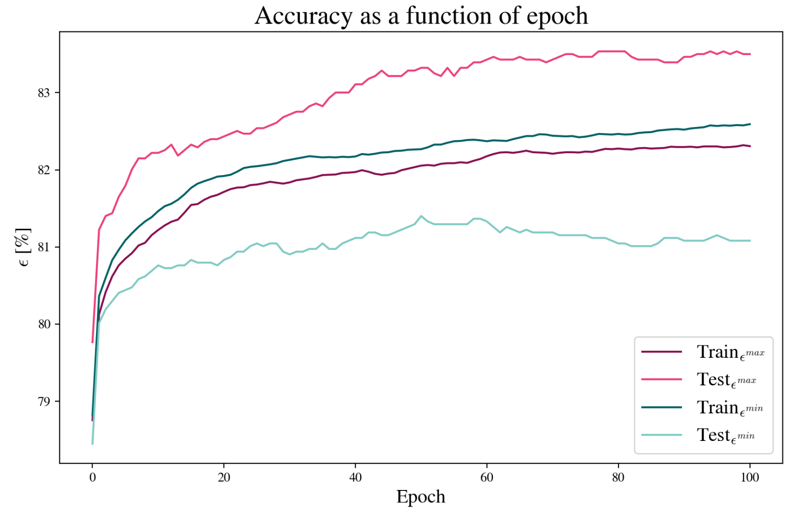

Figure 14: Maximum and minimum accuracy as a function of epochs.

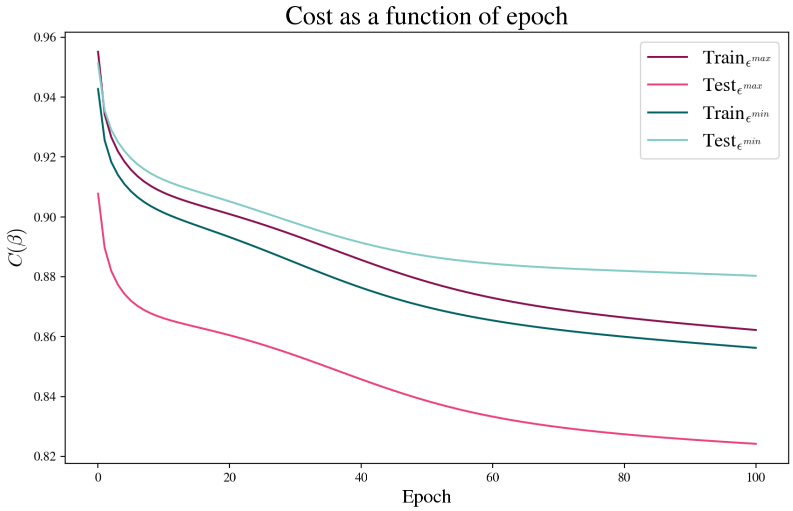

Figure 15: Cost for the maximum and minimum accuracy as a function of epochs.

We compare out developed neural network with a neural network from tensorflow/keras. The result is seen in figure 16, where the performance of both networks are equally good.

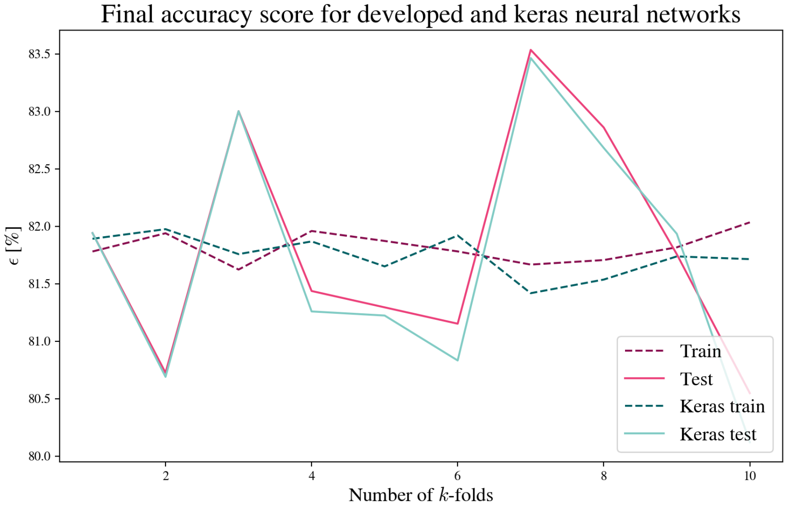

Figure 16: Final accuracy output of the developed and tensorflow/keras neural networks.

3.4 Linear regression with neural network

For the linear regression neural network, we use the Franke function as input data. Compared to the credit card data, preprocessing of the Franke function is relatively easy. We define a meshgrid that is (100×100), and calculate the function. It is important to note that the entire Franke function acts as our target, while the meshgrid are the features. The network performance is measured using the R^2 score, but in the following this is referred to as accuracy.

In order to find the best input parameters to our network, we test different values of the regularisation hyper-parameter λ, learning rate η and size of the mini-batches. Figure 17 shows the accuracy as a function of epochs for different values of λ. The network was run with $\eta = 0.01$, a mini-batch size of 100, 300 epochs and 10 folds for cross-validation. For the largest value of λ, the accuracy is zero for all epochs. While this might seem like a mistake, we recall from project 1 that the prediction of the Franke function using Lasso regression gave a flat surface for large hyper-parameters. If we were to plot the prediction using $\lambda = 1$, this is indeed what we would obtain. Further, we note that smaller λ-values lead to higher accuracy scores.

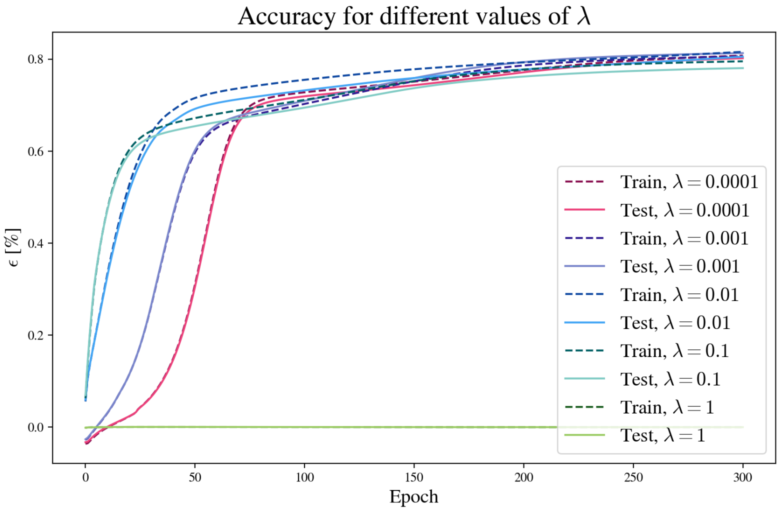

Figure 17: Accuracy score of the linear regression neural network for different regularisatiomn hyper-parameters λ.

Figure 18 shows the accuracy when varying the learning rate, η. The network was run without regularisation, a mini-batch size of 100, 300 epochs and 10 folds. We see that the smallest learning rate stays constant just below 0, and will most likey not increase. As the learning rate increase, the accuracy converges to a maximum. Based on the figure, a learning rate of $\eta = 0.01$ should provide a a sufficiently good accuracy score.

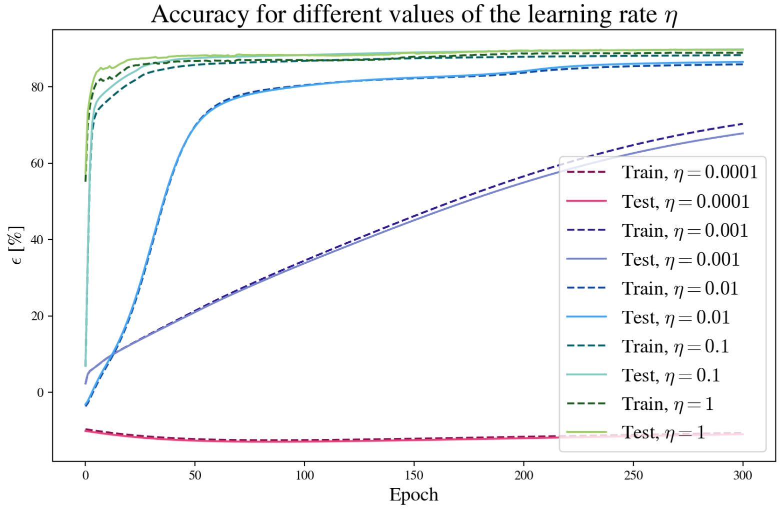

Figure 18: Accuracy of the neural linear regression network for different values of the learning rate η.

Figure 19 shows the accuracy when varying the mini-batch size. The network was run with a learning rate of $\eta = 0.01$, zero regularisation, 300 epochs and 10 folds. The figure shows that smaller mini-batch sizes yield higher accuracy, but the difference in accuracy between the sizes 10 and 150 is not large. Increasing the mini-batch size speeds up the computation time, hence in this case it is advantagous to choose a larger size.

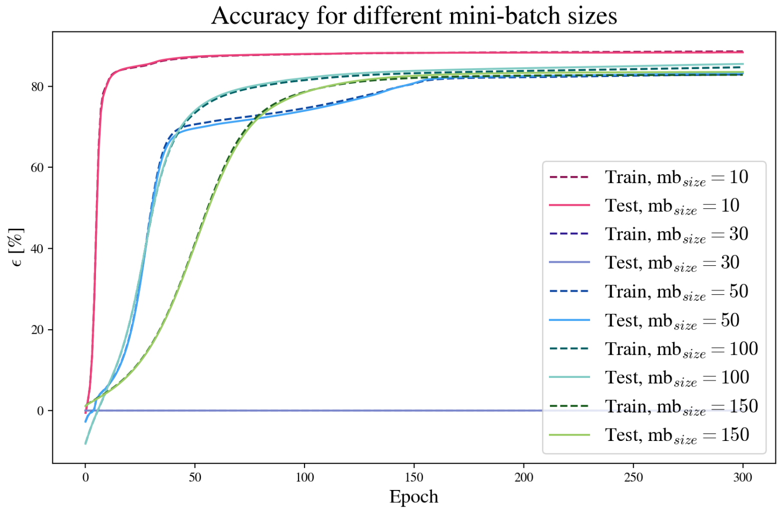

Figure 19: Accuracy of the linear regression neural network for different mini-batch sizes.

Based on the above assessment of the input parameters, we run our network with $\eta = 0.01$, $\lambda = 0.001$, mini-batch size of 150, 200 epochs and 10 folds. The results are displayed in figures 20, 21 and 22, showing the accuracy, cost and prediction, respectively. Compared to the previous models of this project, the cost and maximum and minimum accuracy from the k-fold cross-validation takes on the same behaviour. While the accuracy increase and converges to a maximum, the cost function decrease and diverges to a minimum. The most interesting result of the linear regression is seen in figure 22. The figure shows the Franke function together with the prediction from the neural network, where the input Franke function includes noise. The network is able to predict the most prominent features, such as large peaks. However, it is not able to predict the smaller features, such as the small dip at the bottom of the Franke function.

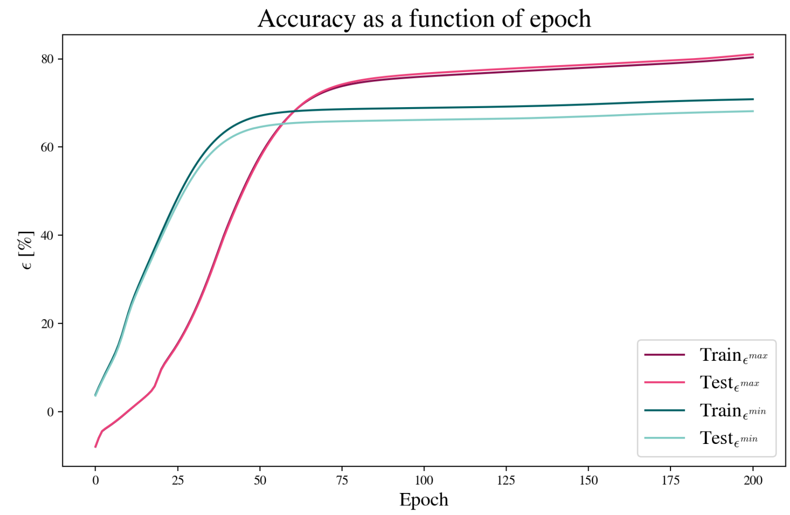

Figure 20: Maximum and minimum accuracy as a function of epochs.

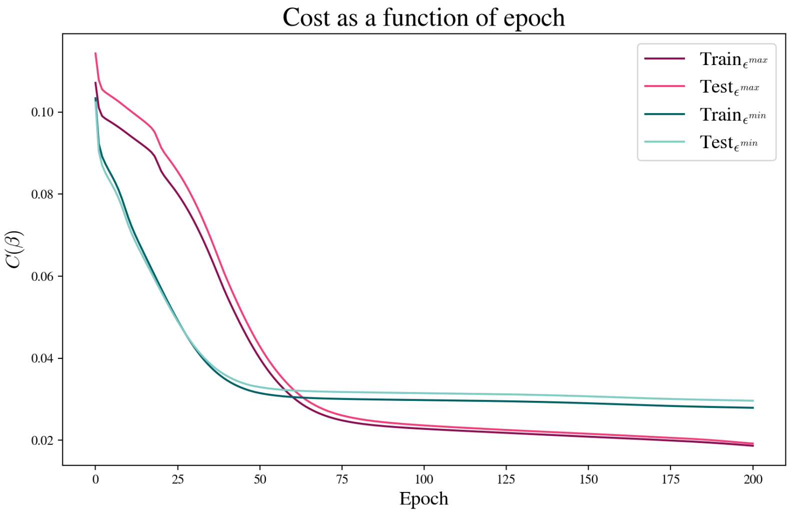

Figure 21: Cost for the maximum and minimum accuracy as a function of epochs.

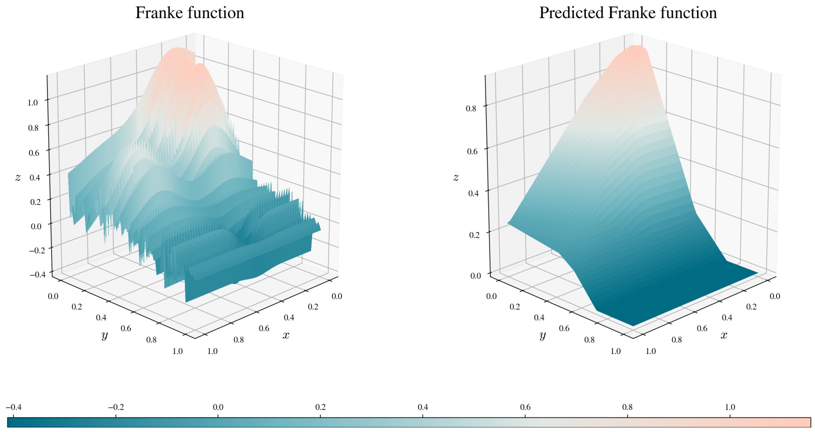

Figure 22: Target and predicted surface from the linear regression neural network.

We compare the developed neural network for linear regression with a neural network from scikit-learn. The result is seen in figure 23, where we have stored the final output accuracy for each k-fold. The figure shows that for most k-folds, both networks perform equally well. At the 4^{th} k-fold, the scikit-learn neural network performance decreases rapidly. At the 9^{th} k-fold, the developed neural network does the same. The accuracy, or R^2 score, becomes 0 since the mean value of the predicted output is equal to the predicted output.

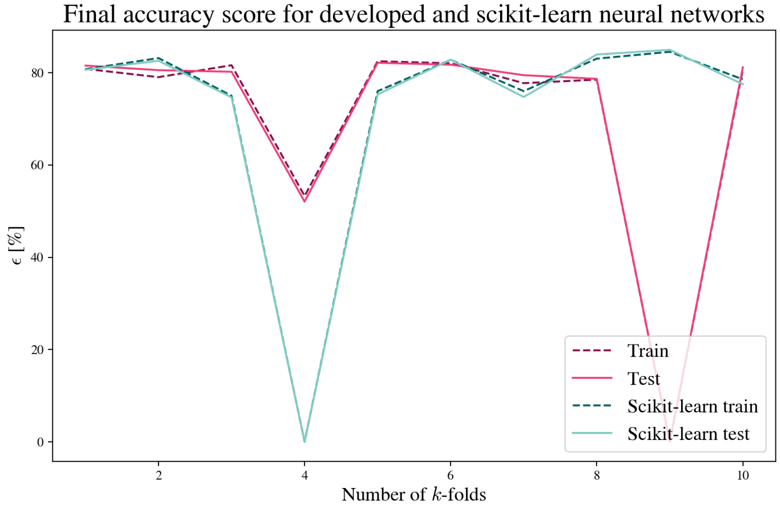

Figure 23: Final accuracy of the developed linear regression neural network and scikit-learn.

Finally, we note that 3 hidden layers with 8, 4 and 3 neurons produce the best results. By increasing the number of neurons, we get a decrease in the R^2 score. The same happens if we reduce the number of hidden layers to one. **Why?? Thoughts..**

4 Discussion

After performing logistic regression on the credit card data, we have obtained an accuracy score of 80%. This is comparable to the R^2 score of 79.4% obtained by Yeh and Lien (2009) from their logistic regression model. Generally, a comparison of our model to the models in the cited paper is difficult. There is a drawback of the credit card data set, as it consists of uncategorised data comprising major classes in 6 of the features. This is an issue that is not addressed by the authors of the paper. We have also noted another difference with the data set in the paper and our data set. The authors describe the data as having 25000 observations, while our data sets contains 30000 observations before preprocessing. While the authors might have reduced the data set without explicitly mentioning it, we have not found any combination of data removal that leads to a reduction of 5000 observations. Regardless, the paper is not about the actual data, but rather data mining techniques.

The developed neural network for classification is not nearly as good as the feed-forward neural network employed in the paper. They obtain an accuracy of 96.5%, while we get an accuracy of 83.5%. When developing neural networks, there are always parameters that can be tuned. When testing the networks for different input parameters, we changed them one by one. By testing them simultaniously (i.e. in a tripe for-loop), we might have found a better combination. Alternatively, we could have implemented a dynamic learning rate for better performance.

By comparing the linear regression neural network to the results of project 1, we see that the predicted Franke function is similar to the prediction made with Lasso regression. As we increase the value of λ, the R^2 score decreases. This is seen in both the current and the previous project. Additionally, the R^2 scores obtained in this project are comparable to those of project 1. In the previous project, we obtained an R^2 score from Lasso regression of 80% for a 5th degree polynomial with $\lambda = 0.001$. In this project, we obtain an R^2 score that is approximately the same, but with a value of λ that is one order of magnitude larger. We might have obtained a better score by exploring other activation functions.

Write about hidden layers and neurons? Does it make sense tha lin reg NN needs more hidden layers? Why do we obtain the best results with one hidden layer for the classification NN?

5 Conclusion

Summarise the method, finding etc. What have we learned?

6 Appendix

Relevant programs developed to solve this project can be found at the GitHub address

https://github.com/hellmb/FYS-STK4155/tree/master/Project_2

References

Hjorth-Jensen, M.
2019. Data analysis and machine learning: Neural networks, from the simple perceptron to deep learning. https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html. Accessed: 2019-07-11.

Marsland, S.
2014. *Machine Learning: An Algorithmic Perspective*. Chapman and Hall/CRC.

Mayo, H., H. Punchihewa, J. Emile, and J. Morrison
2018. History of machine learning. https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html. Accessed: 2019-07-11.

Patrikar, S.
2019. Batch, mini batch & stochastic gradient descent. https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a. Accessed: 2019-07-11.

Peixeiro, M.
2019. How to improve a neural network with regularization. https://towardsdatascience.com/how-to-improve-a-neural-network-with-regularization-8a18ecda9fe3. Accessed: 2019-07-11.

Yeh, I.-C. and C.-h. Lien
2009. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2, Part 1):2473 – 2480.