FYS-STK4155 Project 2

Jan Ole Åkerholm

November 15, 2019

Abstract

Classification of credit card data and modelling of the Franke function is studied using self-developed methods for logistic regression and a feed-forward multi-layer perceptron neural network capable of both classification and linear regression. A stochastic gradient descent method using minibatches was employed for the purpose of fitting the models. The models are compared against each other, as well as against ordinary linear regression (OLS) used in project 1 [1], and against classification and regression tools from Scikit-Learn. I found that my own logistic regression model performed worse than the tool from Scikit-Learn, and the neural network in general performed slightly better than the other methods. Various values for the parameters of the learning rate η, regularization parameter λ and the minibatch sizes were tested in order to find a set of parameters which provided a good fit. In general a learning rate of 0.1 or 0.01 performed well, depending on the model, with a regularization parameter of 0.0001, and a batch size of 100 showing decent results. This gave an accuracy score for the neural network classifier of 0.824, the logistic regression classifier achieved a score of 0.820. The neural network linear regressor achieved a score of $\sim$0.88 on the Franke function with added noise.

1 Introduction

Machine learning for classification and regression problems can be powerful tools, but knowing which method to use for which problem is not always obvious. There is also a danger of creating a "black-box" model, where the model produces results that appear reasonable, but the reason why might not be clear. This project studies in detail logistic regression for classification purposes by applying it to a set of credit card data from 2005 which describes customers tendency to default on their monthly payments. The data set has previously been studied in detail by I-Cheng Yeh and Che-hui Lien [2]. The goal of this study is to determine which methods work best in the given contexts, and to see what effect the regularization parameter, learning rate and batch size of the stochastic gradient descent has. The credit card data set is a binary classification data set with 23 explanatory variables (predictors), and the target values describe whether or not a customer defaulted on their debt (1) or not (0). Of note is that the data set is very biased towards non-risky customers (87.88%). There are also certain values within the predictors that do not fit the described ranges, some of these were removed from the data set.

The results will also be discussed in comparison with the linear regression model from the previous project [1], where I will now apply a neural network model in an attempt to make a fit of the Franke function.

2 Method

2.1 Credit card data preprocessing

The credit card data set contains a total of 30 000 entries, each with 23 predictors, however not all of the entries contain variables that fit the described range. For example, some of the

predictors describe past payment history, where a value of -1 = paid on time, 1 = paid one month late, 2 = paid two months late, and so on. Some of the entries in the data (in fact, the majority) has a value of 0 for the past payment history, and the meaning of this is not described. However because this is such a large amount of the data, these entries are simply left in the data.

Entries where all of the past bill statements or payments are 0 are removed from the data, as these entries are not relevant.

2.2 Logistic regression

Logistic regression is described in detail in the FYS-STK4155 lecture notes on the subject [3] and *The Elements of Statistical Learning* by Hastie et. al. [4].

Simply put, logistic regression can be used with a set of explanatory variables to provide a simple classification model. In our case, the target variable y is binary, i.e. 1 or 0, and a model can be created where we have a set of weights β, and a set of explanatory variables $\mathbf{X}$, and the probability of the target variable being 1 is described as:

$$p(y = 1|\mathbf{X}, \beta) = \frac{e^{\beta_0 + x_1\beta_1 + x_2\beta_2 + ... + x_p\beta_p}}{1 + e^{\beta_0 + x_1\beta_1 + x_2\beta_2 + ... + x_p\beta_p}}$$

and correspondingly, the probability of the target value being 0 is simply:

$$p(y = 0|\mathbf{X}, \beta) = 1 - p(y = 1|\mathbf{X}, \beta)$$

Here, p is the number of predictors.

The goal then is to find the optimal weights β by minimizing the *cost function*. In this case the cost function is given by the log-likelihood:

$$C(\beta) = -\sum_{i=1}^{n} y_i \log\left[p(y = 1|\mathbf{X^i}, \beta)\right] + (1 - y_i) \log\left[1 - p(y = 1|\mathbf{X^i}, \beta)\right] \tag{1}$$

where n is the number of entries in the data set. This function is also known as the binary cross-entropy. Here y_i is the target variable.

The goal of logistic regression is to minimize the cost function by choosing the ideal values for β. This is done using an iterative *gradient descent* method.

2.3 Gradient descent

Gradient descent is described in detail in the FYS-STK4155 lecture notes on the subject [5].

The gradient descent method is an iterative method, where an initial set of weights β^0 is generated randomly. A new set of weights is then generated by:

$$\beta^{k+1} = \beta^k - \eta \nabla C(\beta^{\mathbf{k}})$$

The gradient of the cost function is given by:

$$\nabla C(\beta^{\mathbf{k}}) = -\hat{\mathbf{X}}^T(\mathbf{y} - \mathbf{p}(y = 1|\hat{\mathbf{X}}, \beta^{\mathbf{k}}))$$

Here, η is the *learning rate* and $\hat{\mathbf{X}}$ is a matrix where each row corresponds to one entry in the data set. The gradient descent method will normally converge towards the closest local minimum, but the learning rate must be chosen such that it is large enough that the method

converges within a reasonable timeframe, but not so large that the gradient descent will "skip over" the minimum and oscillate.

Ideally, the method is not guaranteed to find the global minimum for the cost function. In fact there may be many local minimums, especially for more complex models, and depending on the initial guess for β, the solution may end up in one of these local minimums.

A solution to this problem is to instead use a so-called stochastic gradient descent with minibatches. This method picks a random subset of the total data set, and calculates the gradient on this subset. This greatly reduces the chance that the method ends up in a local minimum, but may cause the solution to be slightly worse than the normal gradient descent in situations where the method would end up converging to the global minimum anyway.

2.4 MLP (Multilayer perceptron) neural network

The form and function of the MLP neural network is described in detail in the FYS-STK4155 lecture notes by Morten Hjorth-Jensen [6].

A neural network is a collection of *nodes* or *neurons* that is loosely based on models of how the neurons of animal brains function. The MLP neural network consists of layers of neurons, where each neuron has a set of weighted inputs from the neurons of the previous layer. The sum of these inputs is called the *activation* of the neuron. Each neuron also contains a function describing its output, the so-called *activation function* ($\mathbf{z} = \mathbf{f}(\mathbf{a})$), which depends on the activation of the neuron. The activation function determines whether or not the neuron "fires" based on its activation.

Note that I generally use vectors $\mathbf{z}$, $\mathbf{a}$, etc. to describe the output and activation of the neurons in any given layer, although I may be discussing single neurons for simplicity.

The MLP consists of the input layer, at least one hidden layer and an output layer. In every layer (except the output layer), each neuron is connected to every neuron of the next layer with a certain weight. Each layer (except the input layer) has its own weight matrix $\hat{\mathbf{W}}^l$.

2.4.1 Feed-Forward propagation

Feed-Forward propagation describes the method in which the activations are propagated throughout the network. The activations of the first hidden layer, $\mathbf{a}^1$ are first calculated by applying the weights and biases of the first layer to the inputs:

$$\mathbf{a}^1 = \hat{\mathbf{W}}^1\hat{\mathbf{X}} + \mathbf{b}^1$$

Where $\mathbf{b}^1$ are the biases of the first layer, which are added to the activation in order to avoid the weights of the neurons killing certain parts of the network by making the activations zero.

The output of the first hidden layer is then found by applying its activation function:

$$\mathbf{z}^1 = \mathbf{f}^\mathbf{1}(\mathbf{a}^\mathbf{1})$$

Next, the activation of the next layer is found by again applying the weights and biases to the output of the previous layer:

$$\mathbf{a}^2 = \hat{\mathbf{W}}^2\mathbf{z}^1 + \mathbf{b}^2$$

This process continues until the output of the output layer is computed.

2.4.2 Activation functions

There are many activation functions to choose from, and in general a different one can be chosen for every hidden layer, as well as the output layer. For the output layer, it's common to use a so called one-hot encoded output when dealing with classification problems. This means that a value of $y = 1$ would be encoded as [0, 1], and a value of $y = 0$ would be encoded as [1, 0]. In this case it's common to use the *softmax* function for the output layer.

Another common activation function is the *sigmoid* function, which is given by:

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

.

Other common functions include the rectified linear unit (ReLU) and $\tanh x$.

An ideal activation function is easily differentiable, as the derivative is required for minimizing the cost function of the layer outputs, which will be further described in the back propagation section.

2.4.3 The back-propagation algorithm, model evaluation, regularization etc.

Short on time, focusing on results. Will describe more if time. Methods used are exactly as described in the FYS-STK4155 lecture notes [6].

3 Results and discussion

3.1 Credit card data

3.1.1 Logistic regression

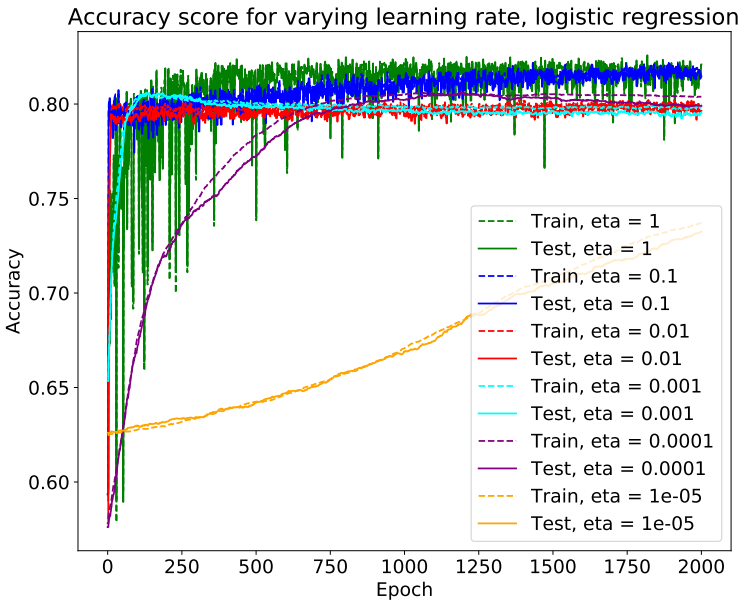

Figure 1: Accuracy score for various learning rates η for the logistic regression classification problem

Figure 1 shows the accuracy score for the logistic regression model with various values of the learning rate η. For the best results, the accuracy ends at approximately 0.82. Of note is that the model appears to work best for a learning rate of 0.1. For a learnign rate of 1, the accuracy has fairly significant downward spikes, even after almost 2000 epochs, while with the learning rate of 0.1 it stabilizes much more towards the end. The smaller learning rates appear to converge to a value slightly lower than the larger learning rates, however this might change with more epochs.

It's also clear that if the learning rate is chosen to be very small, the model needs a lot of epochs in order to converge to a solution.

The Scikit-Learn model gave an accuracy of approximately 0.822 on this data set, slightly better than the self-developed model, but not enough to seem a significant difference.

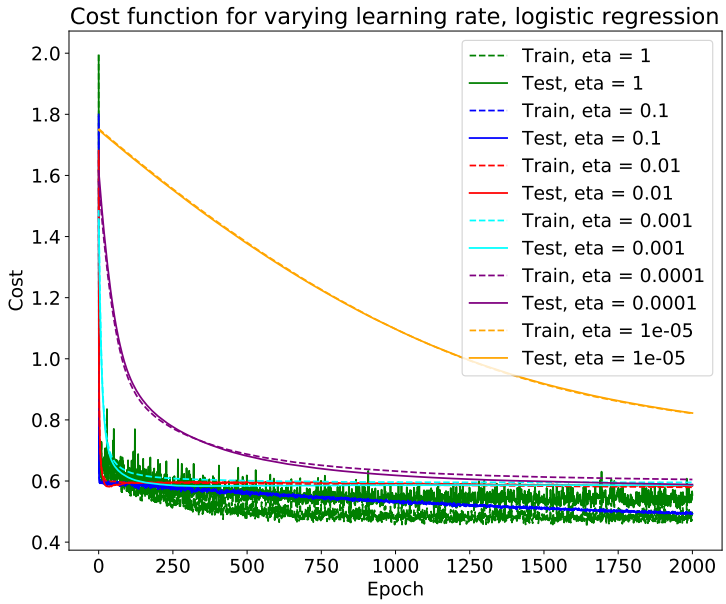

Figure 2: Cost function for various learning rates η for the logistic regression classification problem

Figure 2 shows the same as the previous figure. Again the learning rate of 0.1 appears to be best for the logistic regression model. Of note here is that both the training and testing cost is still dropping after 2000 epochs for the learning rate of 0.1, indicating that there may be additional accuracy to be gained by continuing the computation for futher epochs. However one should also be careful to avoid overfitting by continuing this process too far. In fact it appears that the model using $\eta = 1$ shows signs of overfitting, as the cost function for the test data is clearly larger than the cost function for the training data.

3.1.2 Classification neural network

I used a neural network with two hidden layers, the first with 50 neurons and the second with 20 neurons. Changing the amount of neurons did not appear to give any significant change in the result of this problem.

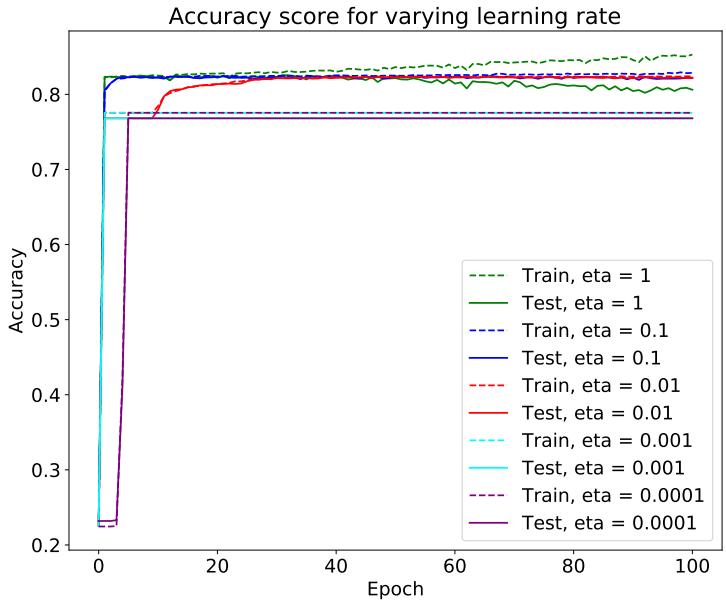

Figure 3: Accuracy score for various learning rates η for the neural network classification problem

Figure 3 shows the accuracy score for various learning rates for the neural network classifier. Note that the reason for the straight lines at approx 0.78 and 0.22 are because the network initially predicts that the outputs of every data input is either 1 or 0 (recall that there is a large bias towards the output showing non-risky customers, so when the network predicts either that all the customers are risky, or all the customers are non-risky, these lines show up at certain values, also note that there is a slight difference in total number of non-risky customers in the test set vs. the training set).

The results are otherwise similar to that of the logistic regression classifier. Overfitting is a clear problem for the model with $\eta = 1$, as we can see that the training accuracy increases, but the test accuracy drops. If one looks very closely, it also appears that overfitting starts becoming a problem also for $\eta = 0.1$ after 100 epochs.

Additionally of note is that much fewer epochs are required before the accuracy is similar to that of the logistic regression model. This does not necessarily mean that the model is less computationally expensive however, as the neural network most of the time will require more weights and biases to be calculated for each epoch.

For the very small learning rates, the network never seems to stop predicting only non-risky or risky customers within the 100 epochs. By letting the network run for a longer time, this could potentially change.

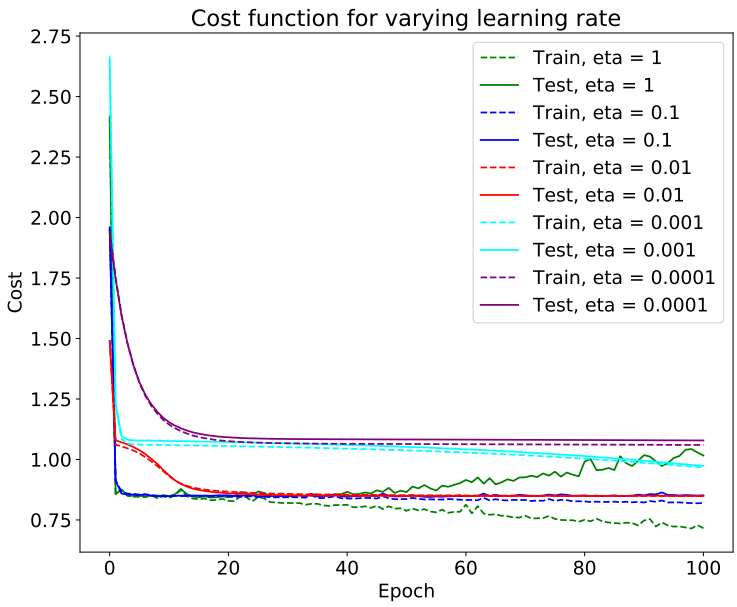

Figure 4: Cost function for various learning rates η for the neural network classification problem

Figure 4 again clearly shows much of the same as in the figure showing the accuracy score. Here the overfitting when using $\eta = 1$ is obvious, as the training cost drops rapidly while the test cost begins growing significantly.

In this case it is easier to tell that the models with very low learning rates might also converge eventually, as it's clear that the cost begins dropping towards the end of the epochs, even for $\eta = 0.0001$.

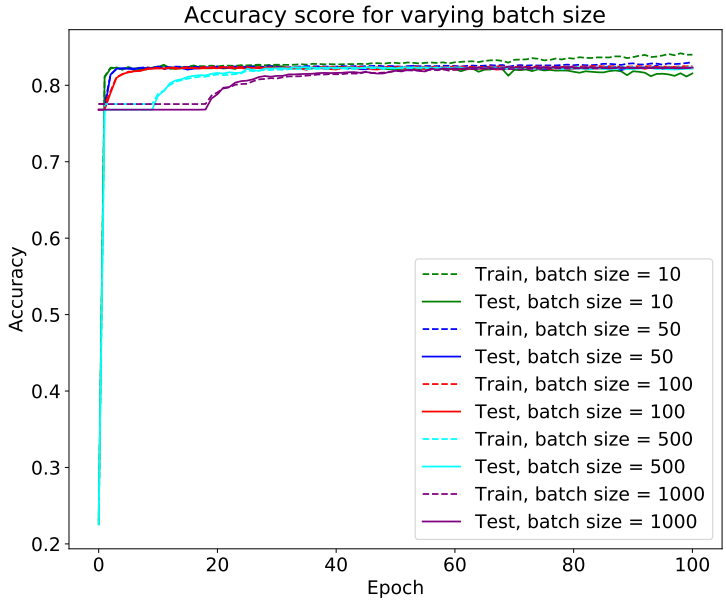

Figure 5: Accuracy score for various batch sizes for the neural network classification problem

Changing the batch sizes did not appear to have a very significant difference in this problem, as can be seen in figure 5. A smaller batch size does appear to give a more rapid convergence, but again this comes at the cost of potential overfitting. With this in mind, a batch size of approximately 100 seems appropriate for this problem.

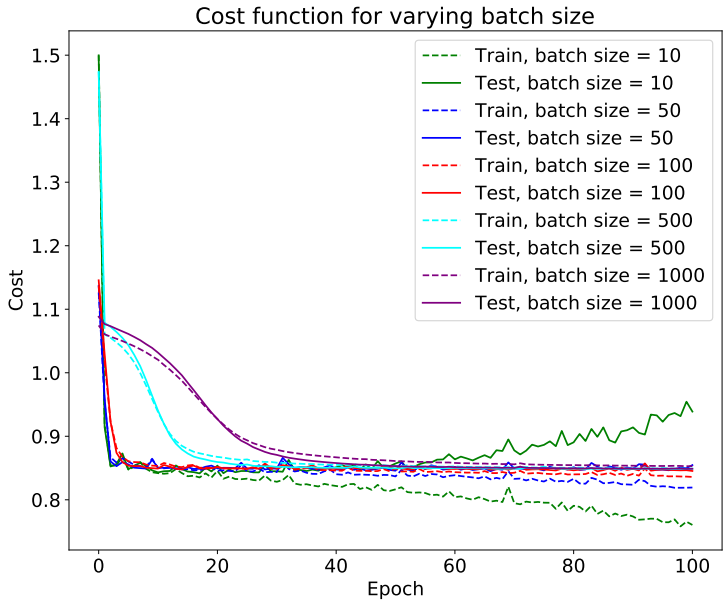

Figure 6: Cost function for various batch sizes for the neural network classification problem

In figure 6 the batch size appears to have a larger impact than what was visible in figure 5. The overfitting is once again clearly visible for a batch size of 10. A batch size of 100 still seems to be appropriate however.

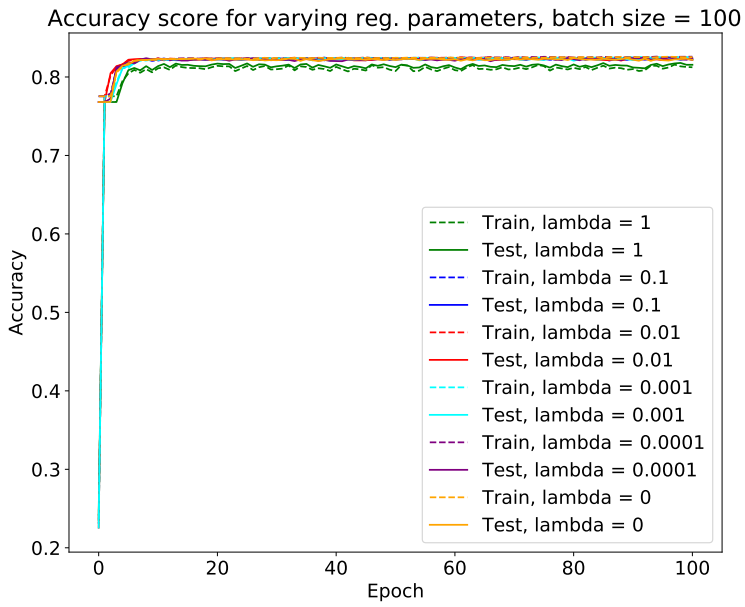

Figure 7: Accuracy score for various regularization parameters λ for the neural network classification problem

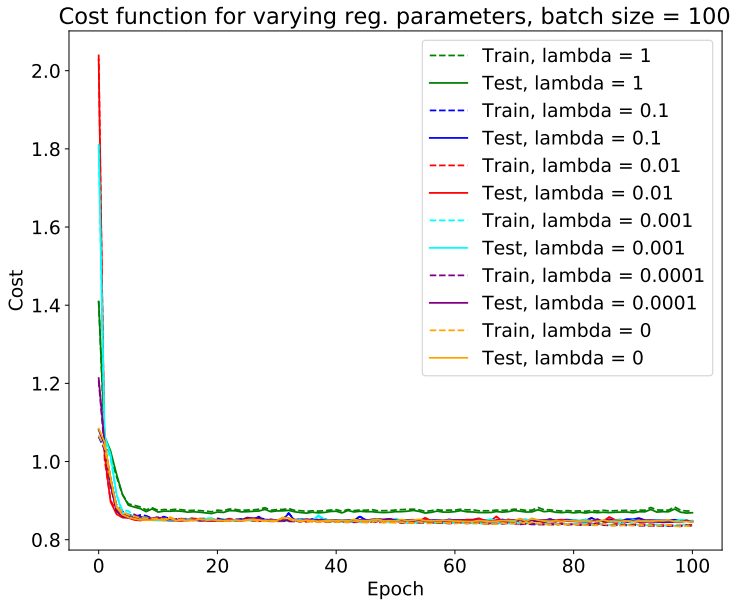

Figure 8: Cost function for various regularization parameters λ for the neural network classification problem

Figures 7 and 8 show that the regularization parameter does not affect the performance of the network significantly, but the difference between the test results and the training results seem somewhat smaller. As one of the primary goals of regularization is to reduce overfitting by penalizing the largest weights, this is expected and good to see.

The best result from the classification neural network was an accuracy score of 0.824 for the test

data. This is not much better than the logistic regression model, which is somewhat unexpected and disappointing. This may be due to problems with the neural network code, or simply poor choice of parameters, including the number of neurons and types of activation functions (for example, I did not have time to try implementing other activation functions for the hidden layers, so I only tried with the sigmoid function). The result is also much worse than the neural network result from Yeh and Lien [2], where they achieve a result of 0.965.

3.2 Franke function

The Franke function and the linear regression fit of this function is described in detail in project 1 [1]. The Franke function is generated using 100x100 points, and a small amount of noise is added to it. Here I attempt to perform the same fit, but this time by using a neural network instead, with the coordinates as inputs and the height as outputs. The neural network once again used two hidden layers with 50 neurons in the first, and 20 neurons in the second. This appeared to give a good fit without significant overfitting after 100 epochs. I suspect the lack of significant overfitting might be because a relatively small number of 20 neurons in the final hidden layer have to describe a total of 1000 output values (heights).

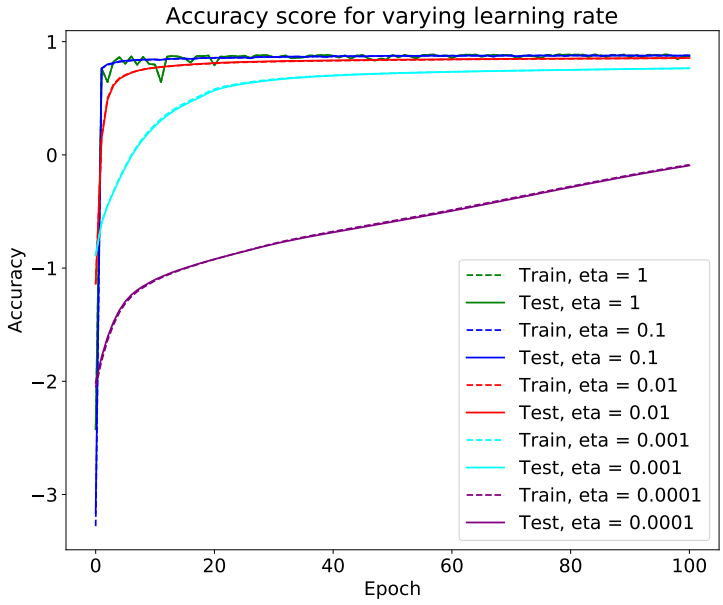

Figure 9: Accuracy score for various learning rates η for the neural network regression problem

From figures 9 and 10 we can see that for most learning rates (everything larger than 0.001) the model converges towards a decent fit of approximately very rapidly. There does not appear to be any significant overfitting for any of the models. A model with a batch size of 100, learning rate η of 0.1 and a regularization parameter λ of 0.0001 gave a model with a test accuracy of approximately 0.88, which is better than the linear regression model from project 1 when applying the same amount of noise to the Franke function [1]. The results from the fit can be seen in figures 15 – 17.

From figures 11 – 14, we can see that changing the batch size or regularization parameter does not appear to give a significant change to the result, other than when $\lambda = 1$, in which case the result is visibly worse for both the test and training data.

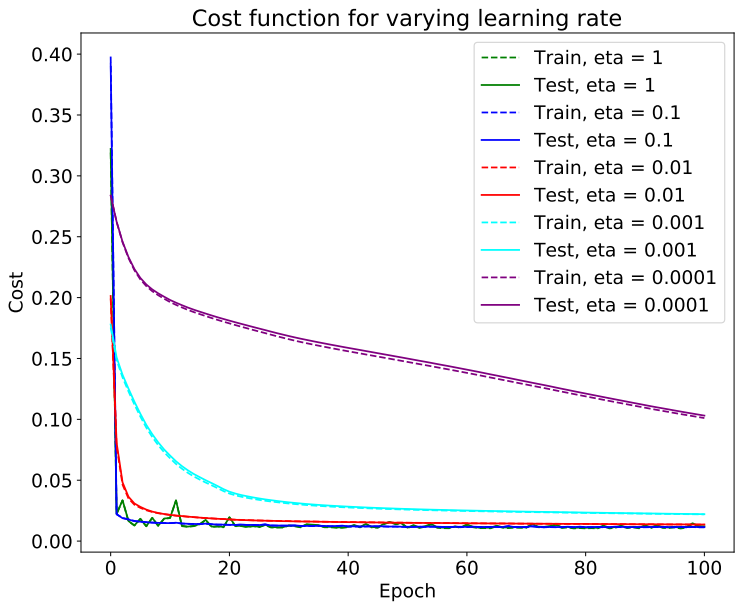

Figure 10: Cost function for various learning rates η for the neural network regression problem

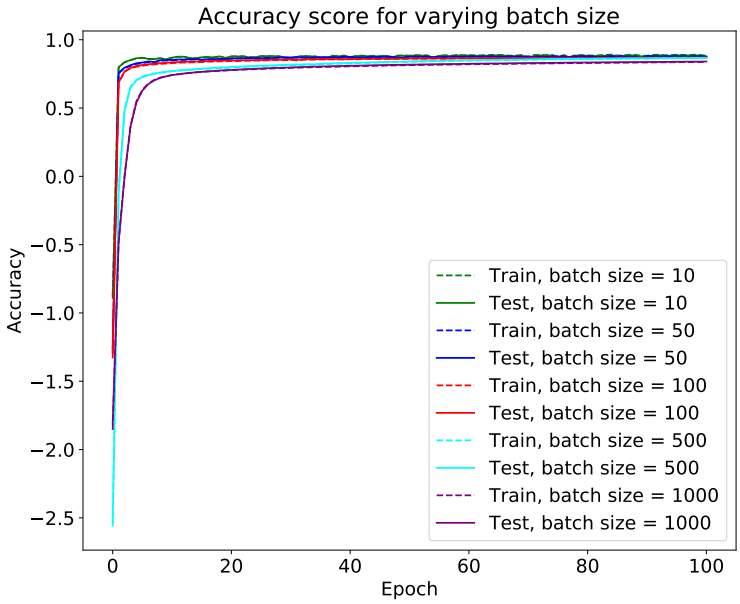

Figure 11: Accuracy score for various batch sizes for the neural network regression problem

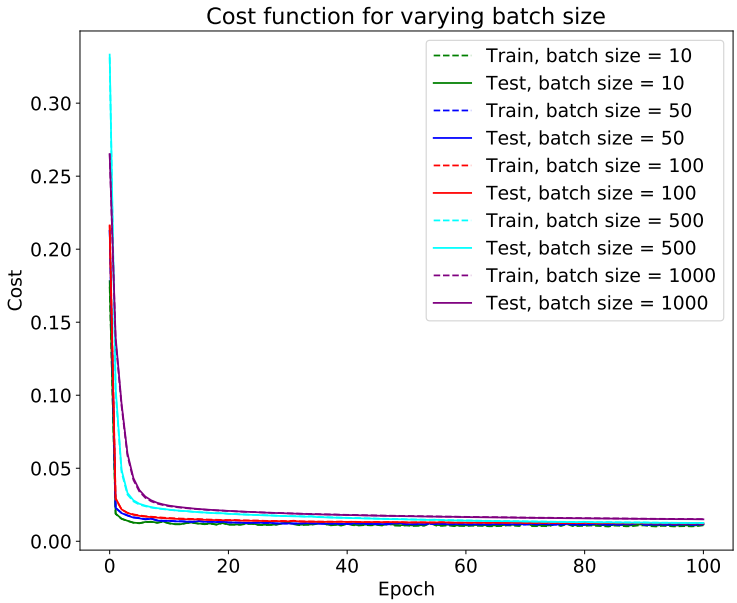

Figure 12: Cost function for various batch sizes for the neural network regression problem

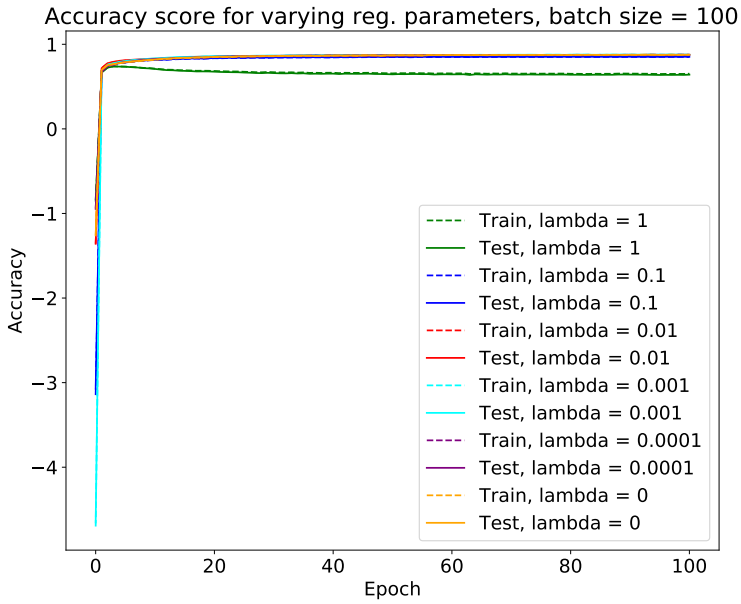

Figure 13: Accuracy score for various regularization parameters λ for the neural network regression problem

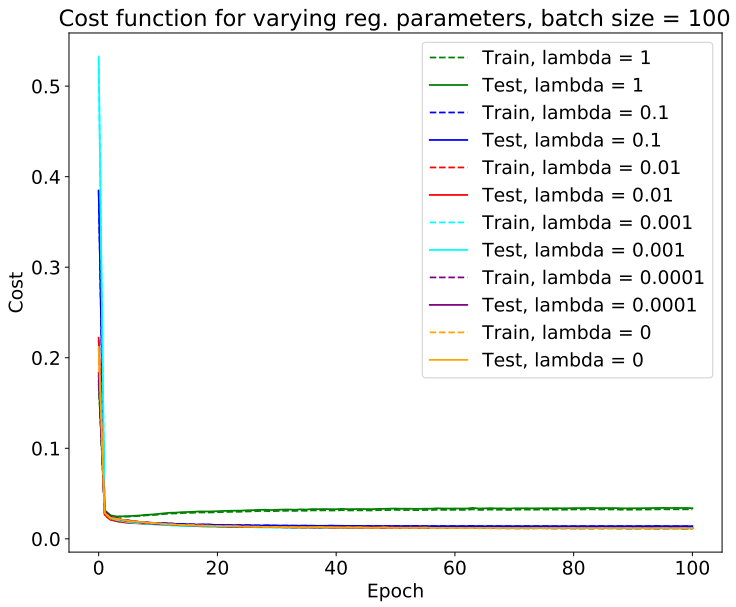

Figure 14: Cost function for various regularization parameters λ for the neural network regression problem

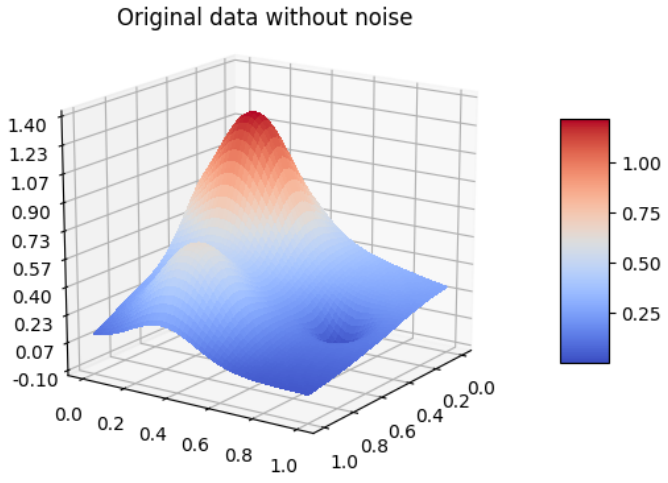

Figure 15: Generated data points for the Franke function

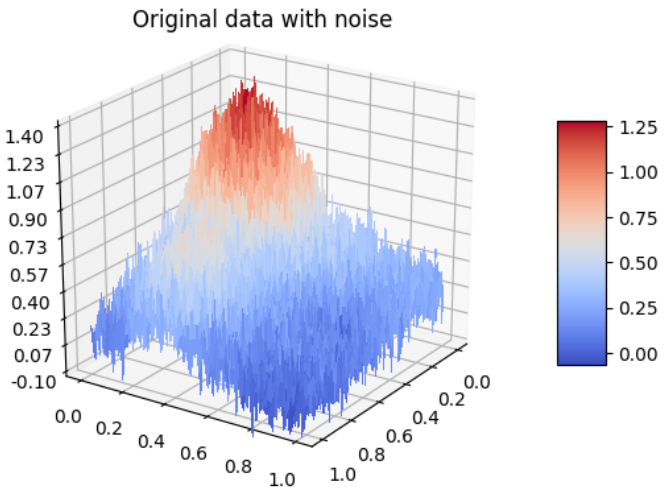

Figure 16: Franke function with added noise. Standard deviation is 0.1.

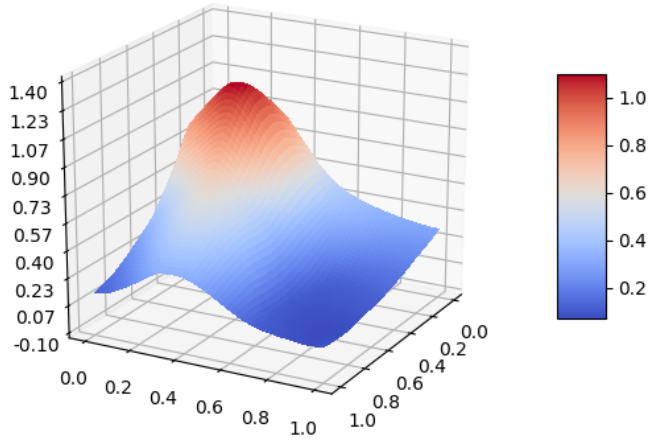

Figure 17: Franke function fit generated by the neural network. $\eta = 0.1, \lambda = 0.0001$ batch size $= 100$

4 Conclusion

Due to illness and time constraints I was unfortunately not able to complete all the parts of the project. Particularly lacking is a comparison between the developed neural network and a neural network using one of the standard libraries (Sckikit-Learn or Keras/Tensorflow). Some parts on the developed methods are also missing, like proper regularization for the logistic regression, as well as k-fold cross validation or bootstrapping.

In general, the neural network appears to work best out of the methods developed here for both

the classification problem, and the linear regression problem, however it's worth noting that the performance is lower than expected, especially for the classification problem, compared to the results from Yeh and Lien [2].

The credit card data set contains many undocumented values, and ideally would have been cleaned up more before the classification analysis. This might also have some impact on the somewhat disappointing results.

Also worth noting is that the difference between the logistic regression classification, and neural network classification is rather small (only about 5%). This is once again unexpectedly bad performance from the neural network. A comparison with a Scikit-Learn network would shed some light on whether or not this is a problem with the network code or architecture, or something else, but unfortunately I did not have time to do that analysis, so that would have to be somethign to look into in the future.

In general it's clear that the neural network performs best in the classification case, but also decent for the linear regression case. Additional exploration with other activation functions would be of interest in the future to explore the effects this has on the result.

References

[1] Jan Ole Åkerholm. "FYS-STK4155 Project 1, Linear regression for terrain data". In: (2019). URL: https://github.com/JanOleA/FYS-STK4155/blob/master/project1.

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

[3] Morten Hjorth-Jensen. *Data Analysis and Machine Learning: Logistic Regression.* https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html. Accessed: 2019-11-15.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition (Springer Series in Statistics).* Springer, 2016. ISBN: 0387848576.

[5] Morten Hjorth-Jensen. *Data Analysis and Machine Learning Lectures: Optimization and Gradient Methods.* https://compphysics.github.io/MachineLearning/doc/pub/Splines/html/Splines.html. Accessed: 2019-11-15.

[6] Morten Hjorth-Jensen. *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: 2019-11-15.