Classification and Regression

from linear and logistic regression to neural networks

Elisabeth Strøm

November 13, 2019

Abstract

Abstract

Contents

1 Introduction

In this project we build a Feed Forward Neural Network (FFNN), and perform linear and logistic regression analyses on two different types of data.

First we perform a logistic regression on credit card data taken from a bank in Taiwan using a stochastic gradient descent. Next use our Feed Forward Neural Network, and perform the logistic regression one more time, and compare our results.

In the next part, we perform a linear regression analysis using our Neural Network on the Franke function (Franke, 1979), and compare our output with that found previously in our project 1 report.

In Section 2, we go through the theory behind the methods we will use, and also show how we will implement them numerically.

Our results can be seen in Section 3, and we will discuss them in Section 4. Finally, we sum up our conclusions in Section 5.

2 Method

In general, we have the explanatory variables $\boldsymbol{x}_i$, in the form of a design matrix $\boldsymbol{X}$, and the response variable $\boldsymbol{y}$. The exact relation between $\boldsymbol{x}_i$, depends on our problem, and is given in more detail in the next few sections.

For our logistic regression analyses, and when using our Neural Network, we split our data into a training and a test set, using 30% of our data for testing.

2.1 Logistic Regression

In our logistic regression analysis, we have as a cost function the cross-entropy function,

$$\mathcal{C}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=0}^{n-1} \left(y_i \log s(\hat{y}_i) + (1 - y_i) \log[1 - s(\hat{y}_i)] \right). \tag{1}$$

Here, y is the actual binary response, $\hat{y}$ is the predicted response, and $sigma(x)$ is the sigmoid function (Equation 15).

We are doing a linear logistic regression, meaning the predicted outcome is defined as

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta}, \tag{2}$$

where $\boldsymbol{\beta}$ are the best-fit regression parameters to be decided by minimizing $\mathcal{C}$, and $\boldsymbol{X}$ is the design matrix. For our classification problem, the columns of $\boldsymbol{X}$ consists of the different explanatory variables of our dataset. Some are numerical, others categorical.

2.1.1 Stochastic Gradient Descent

There are several ways of optimizing the cost function, in order of deciding the best fit $\boldsymbol{\beta}$. We will use the stochastic gradient descent (SGD), in our logistic regression, and in our Neural Network. We verify our results by comparing with the linear `LogisticRegression` class of `sci-kit learn`.

In SGD, we run through several epochs. In each epoch, we randomly shuffle our data, $\boldsymbol{X}$ and $\boldsymbol{y}$, and split it into m mini-batches, $\boldsymbol{X}_k = \boldsymbol{X}_{k\cdot M:(k+1)M,*}$ and $\boldsymbol{y}_k = \boldsymbol{y}_{k\cdot M:(k+1)M}$. Here M is the number of points in each batch m, and k is the k'th batch, $k = 0, 1...m-1$. If we have n datapoints, then $m = n/M$

For each batch, we calculate the derivative of the cost function with respect to $\boldsymbol{\beta}$,

$$\frac{\partial \mathcal{C}(\boldsymbol{y}, \hat{\boldsymbol{y}}_k)}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}_k^T[\boldsymbol{y}_k - \sigma(\hat{\boldsymbol{y}}_k)], \tag{3}$$

where ^{T} means the transposed matrix, $\hat{\boldsymbol{y}}_k = \boldsymbol{X}_k\boldsymbol{\beta}_k$, and $\sigma(x)$ is the sigmoid function (Equation 15).

We use the gradient to update our β parameters,

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta\nabla\mathcal{C}(\hat{\boldsymbol{y}}_k). \tag{4}$$

Our final $\boldsymbol{\beta}$ estimate, is then $\boldsymbol{\beta}_m$. We initialize $\boldsymbol{\beta}$ by drawing random samples from the standard normal distribution $\sim \mathcal{N}(0, 1)$.

η is the so-called learning rate. We implement a decaying learning rate,

$$\eta = \frac{t_0}{t + t_1}, \tag{5}$$

where we put $t_0 = 1$ and $t_1 = 10$, and where $t = \text{epoch} \cdot m + i$, so that η grow smaller as the number of epochs increase.

The probability of our prediction belonging in class 1, is given by the sigmoid function, $p(\hat{y} = 1|\boldsymbol{\beta}_m, \boldsymbol{X}) = \sigma(\boldsymbol{X}\boldsymbol{\beta}_m)$, where $\boldsymbol{X}$ is then our test or our training set. To convert into binary output, we use,

$$\hat{\boldsymbol{y}}(p) = \begin{cases} 1, & \text{if } p \geq 0.5 \\ 0, & \text{if } p < 0.5 \end{cases}. \tag{6}$$

Stochastic gradient decent has the benefit of introducing randomness into the calculations, which helps ensure that we do not get stuck in a local minimum. And, as we are averaging over the cost function of small batches of the data, calculating the gradient is faster.

2.1.2 Evaluating classification

We have many ways of evaluating the predicted response in classification and regression problems. For the classification part of this project, the metrics we considered where the Accuracy score, the F_1 score, the Cumulative Gains curve (CG), the area under the CG (AUC_C), the area under the ROC (Receiver operating characteristic) curve(AUC_R). FIX THIS

For a binary response variable y that, for instance, takes the values $0, 1$, True Positive (TP) denotes the correctly guessed response $\hat{y} = y = 1$, False Positive (FP), the incorrectly guessed $\hat{y} = 1 \neq y$, Positive Negative (PN), correctly guessed $\hat{y} = y = 0$, and False Negative (FN), the incorrectly guessed $\hat{y} = 0 \neq y$.

The accuracy score is the percentage of correct predictions,

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}} \tag{7}$$

If our data is very imbalanced, if the majority class is significantly larger than the minority class, it is possible to achieve a very high Accuracy, simply by having our model always predicting the majority class. This is the Null Accuracy, or the Baseline Accuracy, so we want an accuracy score higher than this.

For imbalanced data, the F_1 score is more suitable. It is defined as

$$F_1 = 2\frac{\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}}, \tag{8}$$

where

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{9}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{10}$$

Precision is a measurement of, out of all the cases we labeled as 1, how many of them were correctly labeled? Recall answers the question; of all the members of class 1, how many did we predict correctly?. The F_1 score, is the weighted, or harmonic, mean of Precision and Recall.

2.1.3 AUC

We can calculate the area under the curves of ROC and CG.

2.2 Linear Regression

We performed a linear regression analysis in project 1, using Ordinary Least Squares (OLS), Ridge, and Lasso regression on the Franke function. Details of these methods can be found in the report of project 1. We will compare our results from this linear regression, with that we find when performing a regression analysis using a Neural Network. More on this in Section 2.3

The cost function of choice for the linear regression analysis, is the one half of the Mean Squared Error (MSE),

$$\mathcal{C}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2}\text{MSE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2}\frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2, \tag{11}$$

which is minimized to find the best fit regression parameters $\boldsymbol{\beta}$.

We have already performed a regression analysis using $k = 5$-fold cross validation on the Franke function (Franke, 1979), $f(x, y)$, given by

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \end{aligned} \tag{12}$$

2.2.1 Evaluating regression

As a way of evaluating our regression results, we calculate the Mean Squared Error as shown in Equation 11. We again, also calculate the R^2 score, defined as

$$R^2(\boldsymbol{y}, \hat{\boldsymbol{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2}, \tag{13}$$

where $\bar{y}$ is the mean of the response y.

2.3 Neural Network

A neural network (NN) is a form of supervised learning, where we feed our algorithm with input data, which is our independent variables, $\boldsymbol{X}$, and also the output data that contains the dependent variable, $\boldsymbol{y}$. The output of the neural network is the predicted outcome $\hat{\boldsymbol{y}}$.

The architecture of our NN has three parts: An input layer, $\boldsymbol{X}$ an arbitrary amount of hidden layers, and one output layer, $\hat{\boldsymbol{y}}$. Between each layer, there are weights $\boldsymbol{W}$ and biases, $\boldsymbol{b}$. Each hidden layer has an activation function, and so does the output layer.

Each layer consists of a number of nodes, or neurons. Neurons takes a group of weighted inputs, applies the activation function, and returns the output. The input to a neuron can be the features from the input data, or output from other neurons in the previous layer. So what happens between layers, is that data that travels between neurons get weights applied to them, and the neuron applies the activation function.

In addition to weights, we also add a bias term to the input. The bias allows the neural network to shift the input left and right.

Our neural network consists of two phases, the Feed Forward phase, and the Back Propagation phase. These are expanded upon in the Section 2.3.4 and 2.3.5

2.3.1 Number of layers and neurons

For the input layer, the number of neurons are equal to the number of independent variables, that is, the number of columns in the design matrix. For the output layer, the number of nodes depends on the analysis we are doing. For a linear regression, we have only one output node. In the case of logistic regression, we have as many nodes as there are separate classes in $\boldsymbol{y}$. For binary classes, we have one node if we pass on $\boldsymbol{y}$ as being the output for class 1, or two nodes if $\boldsymbol{y}$ is one-hot encoded, so that $\boldsymbol{y}$ is two dimensional.

There are no set rules on the number of hidden layers and hidden neurons to use. However using too few neurons leads to underfitting, while too many neurons can lead to overfitting. There are many rules of thumb, for instance

- The number of hidden neurons should lie between the number of input and output neurons
- The number of hidden neurons should be 2/3 that of the input neurons plus the output neurons
- The number of hidden neurons should be less than twice that of the size of the input layer,

which was all taken from Heaton (2015).

Other rules of thumbs: As the complexity between the input data and the desired output increases, more neurons may be needed. The following formula also provides an upper bound estimate on the number of hidden neurons, N_h, (H1; H2, 2012)

$$N_h = \frac{N_s}{\alpha(N_o + N_i)}, \tag{14}$$

where N_s is the size of the input data, N_o is the output neurons, and N_i are the input neurons. α should have a value between 5-10, where the less noisy the data is, the higher α can be. We put $\alpha = 6$.

One hidden layer is enough in many cases, but we will also try using two and three. In the end, its a matter of trial and error.

2.3.2 Activation functions

For our hidden layers, we will be using either the sigmoid function, $\sigma(x)$, or the ReLU function as activation.

The sigmoid is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{15}$$

with the derivative with respect to x as

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)). \tag{16}$$

The ReLU is defined as

$$R(x) = max(0, x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \geq 0 \end{cases}, \tag{17}$$

and its derivative with respect to x is

$$R'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \geq 0 \end{cases}. \tag{18}$$

If we are performing a regression analysis, the output activation we will use, is the linear one,

$$l(x) = x. \tag{19}$$

In the case of classification, we use the sigmoid as an output activation if we have a binary response, or the softmax function if we have a multiclass response, of if the binary response is one-hot encoded. The softmax is defined as

$$softmax(x) = \frac{e^x}{\sum_{i=0}^{n-1} e^{x_i}}. \tag{20}$$

2.3.3 Weight initialization

Before we can begin the Feed Forward phase, the weights and biases needs to be initialized.

There are many ways of initializing the weights, $\boldsymbol{W}$, and biases $\boldsymbol{b}$, for the different layers. We implement two of them: Random initialization,

$$\boldsymbol{W}^i = \mathcal{N}(0,1), \tag{21}$$

$$\boldsymbol{b}^i = \mathcal{N}(0,1) + 0.01, \tag{22}$$

and Xavier initialization,

$$\boldsymbol{W}^i = \mathcal{N}(0,1) \cdot \sqrt{\frac{1}{N^{i-1}}}, \tag{23}$$

$$\boldsymbol{b}^i = \mathcal{N}(0,1) + 0.01 \tag{24}$$

The $+0.01$ term in the bias equations, are there to ensure that all neurons have some output which can be back propagated. N^i is the number of neurons in the ith layer.

The weights are matrices with shape (N^{i-1}, N^i), that is, they depend on the number of neurons in the previous layer,$i-1$, and the number of neurons in this layer, i.

2.3.4 Feed Forward

In the Feed Forward phase, we calculate the output of each layer in the NN, that is, weight is applied to the input, and the bias is added. Then the activation function is applied.

The two equations needed, are

$$\boldsymbol{z}^{i+1} = \boldsymbol{a}^i \boldsymbol{W}^{i+1} + \boldsymbol{b}^{i+1}, \tag{25}$$

$$\boldsymbol{a}^{i+1} = A^{i+1}(\boldsymbol{z}^{i+1}), \tag{26}$$

where $i = 0...L$, where $i = 0$ is the input layer, and $L = L_h + 1$ is the output layer, with L_h as the number of hidden layers.

$\boldsymbol{z}^i$ is then the output of applying weights and adding the bias to the layers. $\boldsymbol{a}^i$, is the output from applying the activation function A^i. The input from the input layer is $\boldsymbol{a}^0 = \boldsymbol{X}$.

The final output, $\boldsymbol{a}_o = \boldsymbol{a}^L$, is passed on to the Back Propagation phase

2.3.5 Back Propagation

The main thing that happens during Back Propagation, is that we update the weights and the biases. To do this we need to compute the derivative of the Cost function $\mathcal{C}$ with respect to the weights $\boldsymbol{W}$, the biases $\boldsymbol{b}$, and $\boldsymbol{z}$.

By using the chain rule, we have that

$$\frac{\partial \mathcal{C}(\boldsymbol{y}, \boldsymbol{a})}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{C}(\boldsymbol{y}, \boldsymbol{a})}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}}, \tag{27}$$

$$\frac{\partial \mathcal{C}(\boldsymbol{y}, \boldsymbol{a})}{\partial \boldsymbol{W}} = \frac{\partial \mathcal{C}(\boldsymbol{y}, \boldsymbol{a})}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}, \tag{28}$$

$$\frac{\partial \mathcal{C}(\boldsymbol{y}, \boldsymbol{a})}{\partial \boldsymbol{b}} = \frac{\partial \mathcal{C}(\boldsymbol{y}, \boldsymbol{a})}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}. \tag{29}$$

Remember that $\boldsymbol{a^i} = A^i(\boldsymbol{z^i})$ and $\boldsymbol{z}^i = \boldsymbol{a}^{i-1}\boldsymbol{W}^i + \boldsymbol{b}^i$. The full derivation of these expressions can be found at David (2019).

Our algorithm then looks like this: For the output layer, the gradient of a MSE cost function with a linear output activation, or a cross-entropy cost function with sigmoid, or softmax output acitvation, iwith respect to the weighted input, is

$$\mathrm{d}\boldsymbol{z}^L = \boldsymbol{a}^L - \boldsymbol{y}. \tag{30}$$

This is also the error in our prediction $\boldsymbol{a}^L = \hat{\boldsymbol{y}}$.

For the remaining layers,

$$\mathrm{d}\boldsymbol{z}^{i-1} = \mathrm{d}\boldsymbol{z}^i \boldsymbol{W}^{i^T} A'^i(\boldsymbol{z}^{i-1}). \tag{31}$$

The gradients of the weights and the biases, irregardless of layer, are

$$\mathrm{d}\boldsymbol{W}^i = \boldsymbol{a}^{i^T}\mathrm{d}\boldsymbol{z}^i + \lambda \boldsymbol{W}^i, \tag{32}$$

$$\mathrm{d}\boldsymbol{b}^i = \sum_j^{N^i} \mathrm{d}\boldsymbol{z}^i_{j,*}, \tag{33}$$

which we loop though recursively, $i = L, L-1, ..., 1$. Here, λ is a regularization parameter.

Now we just have to apply the changes to the weights and biases,

$$\boldsymbol{W}^i_{\text{new}} = \boldsymbol{W}^i - \eta \cdot \mathrm{d}\boldsymbol{W}^i, \tag{34}$$

$$\boldsymbol{b}^i_{\text{new}} = \boldsymbol{b}^i - \eta \cdot \mathrm{d}\boldsymbol{b}^i. \tag{35}$$

2.4 Training

We split our data into training and test data, where 0.3 of the data is used for testing.

For the basic logistic regression case with the SGD, we train the model with the training set, and then fit to the test set to get out a prediction.

In our neural network, we also implement the stochastic gradient descent. Just as for the basic logistic regression, we specify the number of epochs, and the mini batch size M. For each epoch, and for each batch in the epoch, we call first on the Feed Forward algorithm, then on the Back propagation.

Once the network is trained, we give it the test data. Forward feeding is then done one final time, and the final output is $\hat{\boldsymbol{y}} = \boldsymbol{a}^L$ for the linear regression case, and

$$\hat{\boldsymbol{y}} = \begin{cases} 1 & , \text{if } \boldsymbol{a}^L \geq 0.5 \\ 0 & , \text{if } \boldsymbol{a}^L < 0.5 \end{cases}$$

for the logistic regression case.

2.5 Implementation

The first thing we do is look at our dataset, imported with `pandas.dataframe`, and establish what are numerical data, and what are categorical variables.

The categorical variables are one-hot encoded with `sci-kit learn`'s `OneHotEncoder`. Then we split our data into training and test sets. We scale the numerical training data with `sci-kit learn`'s `StandardScaler`, and then apply that transformation onto the test set.

2.5.1 Credit card data

The credit card data was collected in October of 2005, from a bank in Taiwan. The response variable is binary, either 0 or 1, indicating if the client had default payment (1). For our Neural Network, we one-hot encode the response, and our output activation is then the softmax function.

In the case of the credit card data, some of the categorical classes are unaccounted for, in that, we do not know what they are.

The features of the data are explained in Yeh and Lien (2009)'s article, where they perform various analyses on the data.

- X1: Amount of given credit in NT dollars.
- X2 Geneder (1=male; 2=female).
- X3: Education (1=graduate school; 2=university; 3=high school; 4=others).
- X4: Martial status (1=married, 2=single, 3=others).
- X5: Age (years)
- X6-X11: Past payment history. X6 = repayment status is September 2005;...; X11= repayment status in April 2005. (-1=pay duly; 1=payment delay of one month; ...; 9=payment delay of 9 months and above).
- X12-X17: Amount of bill statement. X12=amount of bill statement in September 2005; X17=amount of bill statement in April 2005.
- X18-X23: Amount of previous payment. X18=amount paid in September, 2005;...; X23=Amount paid in April 2005.

The classes that appear in the data, that is not mentioned in their article, are X6-X11=0 and -2, X3=5, 6, and X4=0.

We therefore run two analyses, one while removing the unknowns, and one where we leave X6-X11=0 in, as removing it removes nearly 80 % (DOUBLECHECK) of the data. We also remove all X18-X23 that are equal to 0, and where all X12-X17 are equal to 0. We also removes all instances where just one of the parameters are less than 0.

2.5.2 Linear regression on Franke

As in project 1, the first thing we do is create the $\boldsymbol{x}$ and $\boldsymbol{y}$. We make two arrays with evenly spaced points, for a total of a 100 each, between 0 and 1. Then we use the `numpy` command `meshgrid(x,y)` to create a meshgrid. $\boldsymbol{x}$ and $\boldsymbol{y}$ are then used to construct the Franke function $\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y})$. Next we flatten the arrays, and copy $\boldsymbol{z}$ and add the normally distributed noise $\boldsymbol{\epsilon}$. We know have the true function, `z1_true` and the noisy function `z1_noise`,

```
z = FrankeFunction(x, y)
z1_true = np.ravel(z)
# adding noise
z1_noise = np.ravel(z) + np.random.normal(0, 1, size=z1_true.shape)
```

Next thing to do is create the design matrix $\boldsymbol{X}$. It is made so that the polynomial degree increases with the columns. Say we have a 2nd order polynomial, the first row then looks like

$$x_{0,*} = [1, x_0, y_0, x_0^2, x_0 y_0, y_0^2].$$

We found in project 1 that a polynomial degree of 5 was the most appropriate, so we will continue using that. We reran the analyses of project 1, and the best fit data was found using $\lambda = 10^{-2}$ for Ridge regression, and $\lambda = 10^{-6}$ for Lasso Regression. Note that λ here, is not the same as the regularization parameter in the NN.

3 Results

Here we show the results of our regression analyses on the Credit card data???? and the Franke function.

3.1 Results from credit card data analyses

We ran two analyses on the credit card data, one where we removed X6-X11=0, and one where we did not.

We first ran a logistic regression with stochastic gradient decent. The baseline accuracy while removing the unknown variables, is 0.63. The accuracy of the test data, is

Baseline accuracy w unknowns:
Baseline accuracy w/o unknowns:

With unknowns: 0.786
Knowns only: 0.661

Table 1: The $F1$, *Accuracy*, CGC AUC, and area ratio scores for the credit card data, using logistic regression using SGD, and by using scikit-learns logistic regressor. Baseline Accuracy is 0.63 without unkowns, and 0.78 with unknowns REWIRTE. M=40 without unknowns, M= with

Model	Test Accuracy	Train Accuracy	Test F_1	Train F_1	AUC	Area ratio
SGD	0.76	0.78	0.65	0.66	0.71	0.65
scikit-learn	0.78	0.77	0.64	0.68	0.70	0.62
			With unknowns			
SGD	0.81	0.	0.45	0.70		0.53
scikit-learn	0.82		0.46	0.72		0.55

Table 2: The $F1$, *Accuracy*, CGC AUC, and area ratio scores for the credit card data, using logistic regression using SGD, and by using scikit-learns logistic regressor

Model	Layers	Neurons	η	λ	Accuracy	F_1	AUC	Area ratio
NN	2	$(18, 18)$	0.01	10^{-6}	0.70	0.60		
sklearn	2	$(18, 18)$	0.78	0.66	0.70	0.60		
NN	2	$(18, 9)$	0.01	10^{-6}	0.70	0.60		
sklearn	2	$(18, 9)$	0.78	0.66	0.70	0.60		

3.2 Results from the linear regression

In Table 3, are the results from our linear regression using the methods of project 1, which we ran again. All methods have very similar results,

Table 3: The MSE, and R^2 scores for the Franke function, using OLS, Ridge, and Lasso regression and our Neural Network

Model	λ	Train MSE	Test MSE	Train R^2	Test R^2
OLS	-	$4.52 \cdot 10^{-3}$	$4.50 \cdot 10^{-3}$	0.945	0.945
Ridge	10^{-2}	$4.49 \cdot 10^{-3}$	$4.44 \cdot 10^{-3}$	0.946	0.946
Lasso	10^{-6}	$4.50 \cdot 10^{-3}$	$4.47 \cdot 10^{-3}$	0.945	0.946
NN					

Table 4: The MSE, and R^2r2 scores for the Franke function using our Neural Network. The number of epochs, the size of each batch M, the learning rate and teh regularization parameter, are also shown

Neurons	MSE test	MSE train	R^2	Epochs	M	η	λ
(14)	0.013	0.013	0.83	300	300	0.001	10^{-4}

4 Discussion

For the linear regression, it seems that doing OLS provides us with the best fit. The MSE scored for test and training data was one order higher than that we found in project 1. Our R^2 score was also smaller than for OLS, at 0.83. We conclude that an ordinary linear regression provides the better fit. Seeing as the Franke function provides a relatively easy problem, it is not surprising that a more simple algorithm is still able to produce good scores.

5 Conclusion

All code can be found at https://github.com/sefthus/FYS-STK4155/tree/master/project2

References

model selection - How to choose the number of hidden layers and nodes in a feedforward neural network? URL https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw.

Training an Artificial Neural Network - Intro, March 2012. URL https://www.solver.com/training-artificial-neural-network-intro.

Patrick David. All the Backpropagation derivatives, March 2019. URL https://medium.com/@pdquant/all-the-backpropagation-derivatives-d5275f727f60.

Richard Franke. A Critical Comparison of Some Methods for Interpolation of Scattered Data, 1979. URL https://calhoun.nps.edu/handle/10945/35052.

Jeff Heaton. *Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks*. CreateSpace Independent Publishing Platform, October 2015. ISBN 978-1-5057-1434-0. Google-Books-ID: q9mijgEACAAJ.

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. *Expert Systems with Applications*, 36 (2, Part 1):2473–2480, March 2009. ISSN 0957-4174. doi: 10.1016/j.eswa.2007.12.020. URL http://www.sciencedirect.com/science/article/pii/S0957417407006719.