Final version November 15, 2019

Preprint typeset using LaTeX style emulateapj v. 12/16/11

CLASSIFICATION AND REGRESSION, FROM LINEAR AND LOGISTIC REGRESSION TO NEURAL NETWORKS PROJECT 2

Per-Dimitri B. Sønderland and Vala M. Valsdóttir

Final version November 15, 2019

ABSTRACT

In this article we explore the theory and methods of *Logistic Regression* and a *Feedforward Artificial Network* of the type *Multilayer perceptron* with *Backpropagation*. We apply these methods to a data set containing the attribute information of 30.000 credit card clients with the purpose of assessing the risk of default of any given client outside the data set. Further on we assign our neural network architecture to the task of approximating the *Franke Function* with the objective of evaluating the network's ability to solve regression problems. In both the classification and regression case we found that our neural network outperforms models of Logistic and Linear regression.

Subject headings: Machine Learning — Linear Regression — Logistic Regression — Gradient Descent — Stochastic Gradient Descent — Neural Network — Feed Forwad — Back propogation

1 Introduction

The idea and the theory behind Artificial Neural Networks have been around since the 1940s but it is in the recent years that it has become the word on everybody's mouth. It has now become the most used and best performing Machine Learning methods with its ability to learn without being programmed with task-specific rules. The Artificial Neural Network is inspired by, but not identical to an animal brain. Even so, this makes it able to learn and perform task on a higher level than any other method. It can also be used for different purposes like for example classification problems or fitting linear functions. In classification one has two or more possible outcomes, often a problem with a yes or no answer, for example does the patient have breast cancer? By the data collected about the patient the answer will be yes or no. In classification Logistic Regression is commonly used. Logistic Regression uses the sigmoid function to give a probability and one can set a threshold to classify that probability. For example if the sigmoid gives a probability over 0.5 the outcome is 1 (or yes) and if the probability is under 0.5 the outcome is 0 (or no). In these classification problems one can use many metrics to evaluate if the threshold actually should be 0.5 or higher/lower. These evaluation include metrics like the ROC-curve and AUC score. After evaluating which threshold should be used one can then look at how well the Logistic Regression is by plotting it in a Cumulative Gain Chart. The Logistic Regression is often used with the Gradient Descent method to find its optimal parameters. The Artificial Neural Network can also be used to classify the same problems as the Logistic Regression. One can write the Neural Network with different activation functions, like the sigmoid or others, and a cost function, like the cross entropy function, which makes it classify in the same way. The same goes for Linear Regression problems, here the Neural Network can use both the mean square error cost function and one can use activation functions like ELU or ReLu to get satisfying answers just like one can with regular Linear Regression analysis. In this project we explore the possibilities of a Feed-Forward Neural Network with backpropogation, and compare it to Logistic Regression using Credit Card Data from Taiwan as a classification problem, and then we see how the network performs with a linear problem, more specifically the Franke function, compared to Linear Regression.

p.d.b.sonderland@fys.uio.no,
vala.m.valsdottir@fys.uio.no
[1] Department of Physics, University of Oslo, P.O. Box 1048 Blindern, N-0316 Oslo, Norway

2 Theory

2.1 Logistic Regression

When talking about Logistic Regression one has to know the difference or the similarities between odds and probabilities (11). If you have a coin and you want to find the odds of getting heads you look at the ratio between the probability of getting heads and the probability that you do not get heads

$$ODDS = \frac{P}{1-P}. \tag{1}$$

This is an important part of the understanding of

Logistic Regression, and the function above is usually logged

$$\log\left(\frac{P}{1-P}\right), \tag{2}$$

which we will come back to later. The main point of the Logistic Regression model is that it takes in a classification problem and gives probabilities. For example $Y = 1$ or $Y = 0$ if you have a yes or no answer, then one can use the sigmoid-function, which gives values between 1 and 0 to evaluate the probability of the classification problem. When training our model we look for the value of the sigmoid-function that gives us the best fit to the model at hand. For training we use Gradient Descent that converges to the best solution. The sigmoid-function

$$P = \frac{1}{1+\exp\{-y\}}, \tag{3}$$

outputs the probabilities of something happening, $P = 1$ if it occurred and $P = 0$ if not. y is the equation of a line, and if its value is tending towards negative infinity the sigmoid-function will be equal to zero and if it is tending to positive infinity the sigmoid-function will be equal to one. Instead of optimizing the sigmoid-function we can optimize y to find the best fit, and this is were the similarities between odds and probability comes into play (11). As mentioned before the sigmoid-function will only have values between 0 and 1, but the line y can go from negative to positive infinity, an illustration of what has been written here can be seen in figure 1.

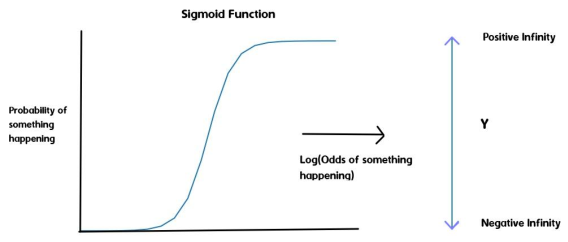

Sigmoid Function
Probability of something happening
Log(Odds of something happening)
Positive Infinity
Y
Negative Infinity

FIG. 1.— Illustration of how we can use probabilities, obtained from the sigmoid-function, to log the odds and get the values on the line y. Figure is greatly inspired from the article from the webpage towardsdatascience.com, written by Cory Maklin, Data Engineer @ Interset

By taking the log of the odds from the sigmoid-function we transform the data points onto our new y-axis. These data points can then again be fitted to a line when we find the y-intercept and slope (11). To find these we need more flesh on the bone, to be precise we need a likelihood function to minimize to be able to optimize the function at hand. On top of this we need parameters to optimize. These are found in our function y which is represented as follows

$$\hat{y} = \hat{X}^T\hat{w} + \hat{\epsilon}. \tag{4}$$

where y is our linear function, $\hat{X}$ is our design matrix and $\hat{w}$ our parameters to optimize, from now on called weights. As talked about before we have our sigmoid-function which gives us the likelihood of $y = 0$ or $y = 1$, mathematically this becomes

$$\begin{array}{c} p\left(y_i = 1|x_i, \hat{w}\right) = \frac{\exp(x_i w_i)}{1+\exp(w_i x_i)} \\ p\left(y_i = 0|x_i, \hat{w}\right) = 1 - p\left(y_i = 1|x_i, \hat{w}\right) \end{array}. \tag{5}$$

The weights w are what we wish to extract from our data. To obtain the total likelihood for all the possible outcomes of our dataset we use the Maximum Likelihood Estimation principle (MLE) (1). Thus we multiply all the probabilities of something happening with all the probabilities of the same not happening, for example all the possibilities that you get heads in a coin toss and all the possibilities that you do not get heads. This gives

$$P(\mathcal{D}|\hat{w}) = \prod_{i=1}^{n}\left[p\left(y_i = 1|x_i, \hat{w}\right)\right]^{y_i}\left[1 - p\left(y_i = 1|x_i, \hat{w}\right)\right]^{1-y_i}. \tag{6}$$

To obtain our cost/loss function or log-likelihood function is simply taking the log of the function above

$$\begin{array}{c} C(\hat{w}) = \sum_{i=1}^{n}\left(y_i \log p\left(y_i = 1|x_i, \hat{w}\right)\right) \\ +\left(1-y_i\right)\log\left[1 - p\left(y_i = 1|x_i, \hat{w}\right)\right) \end{array}. \tag{7}$$

By putting in the values from equation 5 one can calculate this, and it becomes

$$\mathcal{C}(\hat{w}) = \sum_{i=1}^{n}\left(y_i\left(w_i x_i\right) - \log\left(1 + \exp\left(w_i x_i\right)\right)\right). \tag{8}$$

This is the function we are after for optimizing our weights to finally fit our function to the data. This function is commonly called *The Cross Entropy* error function. To minimize the cost-function, which in turn optimizes our weights. We thus need to find the derivative in terms of the weights.

$$\frac{\partial\mathcal{C}(\hat{w})}{\partial w_i} = -\sum_{i=1}^{n}\left(y_i x_i - x_i\frac{\exp\left(w_i x_i\right)}{1+\exp\left(w_i x_i\right)}\right). \tag{9}$$

For the observant eye one sees that the latter function simply adds up to the first function in equation 5. So if we call this term p we can sum up the derivative in of the cost function in matrix form like this

$$\frac{\partial\mathcal{C}(\hat{w})}{\partial\hat{w}} = -\hat{X}^T(\hat{y} - \hat{p}). \tag{10}$$

To obtain the second derivative of this function we need to define a diagonal matrix $\hat{\mathbf{W}}$ which is filled with the likelihood $p\left(y_i|x_i,\hat{w}\right)\left(1-p\left(y_i|x_i,\hat{w}\right)\right.$ on its diagonal. Thus obtaining

$$\frac{\partial^2 \mathcal{C}(\hat{w})}{\partial\hat{w}\partial\hat{w}^T} = \hat{X}^T \hat{W} \hat{X}. \tag{11}$$

To understand why this is important we need to dive into how we train these weights to minimize our cost function, which can be done with many methods, the Newton-Rapson' method, Gradient Descent or the Stochastic Gradient Descent.

2.1.1 *Newton-Raphson's Method*

For understanding the Gradient Descent it is wise to first understand Newton-Raphson's method, since the Gradient Descent can be said to be a modified version of this method. For this method to work sufficiently one need to be able to find the derivative and the second derivative of the cost function at hand, as we did in the last section in equation 10 and 11. This gives us a way to updated our weights over a number of iterations so they better optimize the cost function

$$\hat{w}^{\text{new}} = \hat{w}^{\text{old}} - \left(\frac{\partial^2 \mathcal{C}(\hat{w})}{\partial\hat{w}\partial\hat{w}}\right)^{-1}_{\hat{w}^{\text{old}}} \times \left(\frac{\partial \mathcal{C}(\hat{w})}{\partial\hat{w}}\right)_{\hat{w}^{\text{old}}}. \tag{12}$$

In matrix form

$$\hat{w}^{\text{new}} = \hat{w}^{\text{old}} - \left(\hat{x}^T \hat{W} \hat{X}\right)^{-1} \times \left(-\hat{X}^T(\hat{y}-\hat{p})\right)_{\hat{w}^{old}}. \tag{13}$$

Newton-Raphson's method can be computationally expensive because of the second derivative term with the diagonal matrix $\hat{\mathbf{W}}$ that has to compute all the likelihoods a cross its diagonal (2). In some cases a second derivative of the cost function is also hard to obtain, this is when one has more than one parameter and has to compute many derivatives in a matrix called the Jacobian, this becomes very computationally expensive, and makes way for the often used method instead of Newton-Raphson's the Gradient Descent.

2.1.2 *Gradient Descent*

When having more than one parameter doing a Gradient Descent is often the solution, instead of computing the second derivative of all the parameters. The basic idea of the method is that the cost function $C(\hat{w})$ will decrease faster if one goes from $\mathbf{w}$ in the direction of the negative gradient of the cost function $-\nabla C(\hat{w})$ (2).

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_w \nabla F\left(\mathbf{w}_k\right), \tag{14}$$

were $\eta > 0$ is the substitution for the Jacobian, and is a parameter one needs to tune to fit the model at hand often called the learning rate of the model. This is then repeated for wanted number of iterations until the minimization of the cost function has converged. The downside of this method is that it can converge to a local minimum not a global, which is what is wanted. This can mean that one is actually fitting the function with the wrong minimum, and thus resulting in a poorly fitted model (2). On top of this the the Gradient Descent is sensitive to the learning rate and it is therefore important to find the optimal learning rate for the data. If one has a big set of data one can also experience that the Gradient Descent is computationally expensive since one has to compute the parameters, obtained by the gradient, of the cost function for each iteration.

2.1.3 *Stochastic Gradient Descent*

With Stochastic Gradient Descent and the variants of it, some of the problems with Gradient Descent are improved. The cost function can, in almost all cases, be presented as a sum over n data points $\mathbf{w_i}$ (2)

$$C(w) = \sum_{i=1}^{n} c_i\left(\mathbf{x}_i, w\right). \tag{15}$$

Therefore the gradient of the cost function can be presented as

$$\nabla_w C(w) = \sum_{i}^{n} \nabla_w c_i\left(\mathbf{x}_i, w\right). \tag{16}$$

The essence of the method is that one shuffles the data and then iterate over batches, typically divided into the length of the output data, and only use the parameters from each batch to compute the cost function. The gradient step thus becomes

$$w_{j+1} = w_j - \eta_j \sum_{i\in B_k}^{n} \nabla_w c_i\left(\mathbf{x}_i, w\right). \tag{17}$$

This will automatically make the computation of the descent much faster, and since the data is now shuffled it introduces a randomness which decreases the possibility of getting stuck in a local minimum. If one wishes to optimize even further Stochastic Gradient Descent with mini-batches can be applied. The difference is that you divide the shuffled data into batches of a given size and iterate over them. One can therefore, in theory, end up iterating over the same data many times or some of the data not touched, but this will be less time consuming and is often the most used variant of the Stochastic Gradient Descent methods.

2.2 Neural Networks

The basic idea behind Neural Network is to build a code that acts in a way similar to a brain, hence the name Neural Network. In the brain neurons are the basic working unit that transmits information around to the different areas of the brain. This is the design idea behind the Neural Network, though it does not work the same way as a brain. There is an input, or a signal, which has information that we need to make sense of, this will be sent through a neural network where weights are connected from each input to each neuron, the latter often called a hidden layer. the weights have a value which indicates how active the particular hidden neuron should be, an example of Neural Network is shown in figure 2.

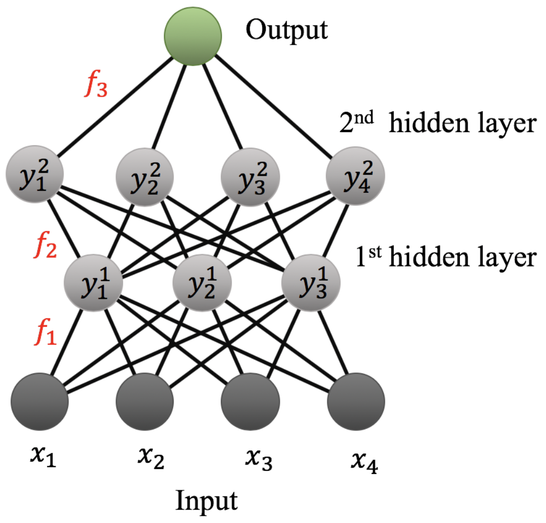


FIG. 2.— Example of a Neural Network with two hidden layers (10)

In the hidden layer the information, given by the weight of the output layer, go through an activation function, for example the sigmoid function, which tells us how active that particular information in that particular neuron is. This again produces output weights which gives information to the output layer, and the value of the output weights tells which part of the output layer should be activated. Now, this is only explained with one hidden layer, but in theory one can have more than one, depending of the complexity of the data. Mathematically this is represented as

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u). \tag{18}$$

Where y is the output at the end, u is its activation function, and w are the weights, and x are the signals.

2.2.1 *Multilayer Perceptrons with Back Propagation*

In this report the Multilayer Percetron (MLP) with Back Propogation was used and in this section the theory of how this network is build up will be further elaborated. An MLP is a fully connected feed forward neural network, and to understand fully what MLPs are, one has to know what a Feed-Forward Neural Network (FFNN) is. Often it is called the simplest type of Neural Networks. The name speaks for itself, the network only moves forward through the layers (3). An MLP, on the other hand, is a fully connected FFNN with three or more layers (input layer, one or more hidden layer and an output layer). The neurons in the hidden layer have non-linear activation functions, whilst the output nodes are assumed to be linear, which gives

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z). \tag{19}$$

An important difference between this function and the function above 18 is that the inputs, or signal x_i are from the output layer of the neuron in the preceding layer. This is what is meant with fully connected. b_i is the bias, which is needed if there is a zero activation on the weights of the inputs. For each node i in the first hidden layer, one can calculate the weighted sum of the output x_j (3)

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1. \tag{20}$$

Where z_i^1 is the argument to the activation function presented above 19 of each node i. The definition of the output y_i^1 of all neurons in layer 1 is

$$y_i^1 = f\left(z_i^1\right) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right). \tag{21}$$

As seen here all the nodes in the same layer has an identical activation function. If one has more than one layer with different activation functions the general form of this becomes

$$y_i^l = f^l\left(u_i^l\right) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right). \tag{22}$$

Where l present number of layer in question. When the output of all the nodes in the first hidden layer is computed, the values of the subsequent layer can be calculated and so forth until the output layer is obtained (3). The output thus becomes, in general form

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_i} w_{ij}^3 f^l \left(\sum_{k=1}^{N_l-1} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\cdots\right) + b_k^l\right) + b_1^3\right] \tag{23}$$

The only independent variable in this equation are the input variables x_n. For the observant eye this shows that the MLP is just an analytical function that maps the real valued vectors $\hat{x} \in R^n \rightarrow \hat{y} \in R^m$ (3). This can then be expressed as a sum of scaled activation functions. By adjusting the parameters, the weights and biases, the activation function can be shifted and change slope or be reduced after once wishes, which is the key to why Neural Network is one of the most used methods in Machine Learning today (3).

2.2.2 *Implementing the Back Propagation for the MLP*

As mentioned above, in this project we make use of an MLP with Back Propagation (BP) and therefore some information of how this works in theory is needed. Until now we have looked at how our output and activation functions work in the FFNN but still unknown are our weights. These are, as mentioned before, something we want to be able to change so we need an algorithm for that and at the same time we wish that our errors are as small as possible, hence the BP algorithm. For evaluation of the equations in the BP we start off with a cost function that we wish to minimize, for this purpose we choose the mean square error cost function

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

Where t_i is the output we wish for, the true values, and y_i are the actual outputs the network has produced. The mean square error cost function is just for illustration, in this project we also make use of the cross entropy cost function for classification, which was talked about in the subsection Logistic Regression 2.1. There are four important equations in the BP algorithm that we need, to make sure that the errors are as small as possible. If we look at the activation of a neuron j of the l-th layer we get, in matrix form

$$\hat{z}^l = \left(\hat{W}^l\right)^T \hat{a}^{l-1} + \hat{b}^l. \tag{25}$$

If we choose the sigmoid as the output activation function $\hat{a}^l = f(\hat{z^l})$

$$a_j^l = f\left(z_j^l\right) = \frac{1}{1+\exp-\left(z_j^l\right)} \tag{26}$$

Derivation of the equation 25 in terms of weights and in terms of $\hat{a}^{l-1}$ gives

$$\begin{aligned}\frac{\partial z_j^l}{\partial w_{ij}^l} &= a_i^{l-1} \\ \frac{\partial z_j^l}{\partial a_i^{l-1}} &= w_{ji}^l\end{aligned}. \tag{27}$$

This will show itself to be important later. The derivative of the activation function in terms of z_j^l is

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l\left(1-a_j^l\right) = f\left(z_j^l\right)\left(1-f\left(z_j^l\right)\right). \tag{28}$$

All of these derivatives are important for the equations for the BP. The derivative of the cost function in terms of the weights is needed so a minimum can be obtained

$$\frac{\partial \mathcal{C}\left(\hat{W}^L\right)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right)\frac{\partial a_j^L}{\partial w_{jk}^L}. \tag{29}$$

The last term can be calculated using the equations given above 27

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L\left(1-a_j^L\right)a_k^{L-1}. \tag{30}$$

Thus,

$$\frac{\partial \mathcal{C}\left(\hat{W}^L\right)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right)a_j^L\left(1-a_j^L\right)a_k^{L-1}, \tag{31}$$

We recognize the term $\left(a_j^L - t_j\right)$ as the derivative of the cost function in terms of a_j^L and the term $a_j^L\left(1-a_j^L\right)$ as the derivative obtained in equation 28. If we define this as the error δ_j^L and write it in vector terms as the Hadamard product

$$\hat{\delta}^L = f'\left(\hat{z}^L\right)\circ\frac{\partial \mathcal{C}}{\partial\left(\hat{a}^L\right)}. \tag{32}$$

This equation is important for the interpretation of the network, the first term measures how fast the activation function f is changing at a given activation value $z_j L$, and the second term measures how fast the cost function is changing as a function of the jth output activation (3). All this results then in

$$\frac{\partial \mathcal{C}\left(\hat{W}^L\right)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}. \tag{33}$$

The term a_k^{L-1}, that is the activation, tells us how the network is learning. If this term is small the derivative of the cost function with respect to the weights will also be small and this means that the system as a whole is learning slowly. The error can be interpreted in many ways, as we already have seen it can be presented in the form of

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}, \tag{34}$$

this can in turn also be in terms of the bias

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L}. \tag{35}$$

Which simply means that the error term is exactly equal to the rate of change of the cost function as a function of the bias (3). We need a last equation to connect the error of the last layer to the error of the previous layer. By replacing L with a general layer l

$$\delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l} \tag{36}$$

using the chain rule and summing over all k's this gives

$$\begin{aligned}\delta_j^l &= \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'\left(z_j^l\right)\end{aligned}. \tag{37}$$

These are all the equation needed to do a proper BP, this gives the opportunity to optimize the cost function at hand according to the weights and biases which is needed to do a Gradient Descent in our Neural Network.

3 Method

This sections covers our approach to a binary classification problem and a regression problem. Various iterative optimization techniques are applied and compared for the classification problem. Next *Artifical Neural Networks* is employed and evaluated for both the classification- and the regression case. All the results, data and illustrations can be found on the GitHub repository referenced in the appendix ??.

3.1 Classification Analysis: Credit Card Data

The basis for this section is the paper *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Expert Systems with Applications*(5). One of the goals of this article was to reproduce some of the results of this paper, in particular those pertaining to *Logistic Regression*, *Gradient Descent* and *Neural Networks*. As in the paper, we look at the same data set displaying defaults(or not) of credit card clients. The data is from Taiwan and examines the payment history of 30.000 people. The payment history ranges from April to September, 2005. In the appendix ?? the different features and what they mean are explained, this data attribute information is taken directly from (6)

3.1.1 *Logistic Regression*

The implementation of logistic regression follows the theory in section 2.1. In the following we lay out step by step the method that was utilized. The procedure was more or less the same for all the different iterative optimization methods, which are all covered in 2.

1. Preparing the data

- We first prepared the data by fixing erroneous inputs in the data. The errors were identified as data points that had a numerical classification outside the range of the given intervals. Such as a person having marital status as 0, when (1 = married; 2 = single; 3 = others) are the only possible statuses.

- The data was first randomly shuffled then it was split into train data, validation data and test data for the purpose of cross-validation. The split was divided in 10% test, 10% validation and 80% train data. The validation data was used to find optimal parameters, and the test data for the final test of the model.

- The data set contained values ranging from the values such as those for marital status, to credit limits of thousands of dollars. The data set was thus scaled $z = (x - \bar{X})/s_x$, where x is the sample, $\bar{X}$ is the sample mean and s_x is the standard deviation of the sample mean. This was done using scikit-learn (7)

2. Setting initial values

- All the iterative methods used needs instructions on how to commence, that is, it needs initial values. The data has 23 features, and is therefore in need of 23 initial weights. The weights, which we in a regression-style fashion have called *beta* in the code, were initialized taking random samples from the standard normal distribution, $N\left(\mu, \sigma^2\right)$.

- The learning rate η, discussed in 2.1.2, can be determined by simply trying out a good number of possible configurations and check which one gives the best score for a given metric. This was done by a loop over learning rates. The size of the interval was determined by some good old ad-hoc intuition obtained by previous test-runs. We did not iterate over how many iterations to use, as we figured that when a minimum was obtained the gradient would not significantly grow again, meaning that if e.g 340 iterations gave a minimum, then 400 would not increase markedly, but fluctuate or create a new minimum. Hence we simply chose a large iteration number for all the learning rates.

- We also tried using a learning schedule(2), where we chose some reasonable initial values and gradually decreased the learning rate size with respect to iterations(epochs) and mini-batches. Naturally, iteration over mini-batches occurred where mini-batches existed. The formula used was $\gamma_j\left(t; t_0, t_1\right) = \frac{t_0}{t+t_1}$, where $t = e \cdot m + i$. Number of mini-batches: m, iteration number over mini-batches: i, iterations(epochs): e. The values of t_0 and t_1 where the arbitrary chosen values.

3. Updating the weights

- Each data example, corresponding to a person, was represented by a row of features. Before updating the weights, the stochastic methods had their rows of data shuffled randomly. The standard gradient descent and Newton's method were kept as they were.

- The weights were updated for each iteration. The weight equations that were used for the different optimization techniques are given by the equations *Newton's Method*: 2.1.1, *Gradient Descent*: 2.1.2, *Stochastic Gradient Descent(With or without mini-batches)*: 2.1.3. The gradient terms used to calculate the weights were exactly as in equation 10 for the gradient methods, and as in equation 13 for Newton's method. The computational approach used for the gradient methods were a straight forward application of their weight equation, but for Newton's method the weight equation has a diagonal matrix $\hat{\mathbf{W}}$ that had to be filled up iteratively along the diagonal. Parts of the resulting equation was then inverted by pseudo-inversion.

4. Evaluation

- The cost functions for the respective methods were initially plotted for various values of the learning rate to check if our code was functioning properly. The expectation was simply that our gradient converged, which could be seen by a decreasing cost function. For the stochastic methods the cost function, although initially decreasing, fluctuates a lot. We chose to use the metric *Area Under the Curve(AUC)*, discussed in section 3.5, to evaluate and compare the different configurations in the parameter space.

3.1.2 *Neural Network: Classification*

The theoretical foundation for our Neural Network follows the theory in section 2.2. The Neural Network we constructed is inspired and in part built upon the Python class by H.J. Morten (3). The network consists of several activation functions a which will be covered later in section 3.4,a feed forward function, a backpropagation function, a train function using stochastic gradient descent with variable choice of mini-batches, and the derivative of the cost functions with respect to the output activation a_i^L. For the binary classification case this is the derivative of the *Cross Entropy* in equation 8, with $y_i = a_i^L = f\left(z_i^L\right)$.

$$\frac{\partial \mathcal{C}(\hat{W})}{\partial a_i^L} = \frac{a_i^L - t_i}{a_i^L\left(1 - a_i^L\right)}. \tag{38}$$

Where t_i is the target data. Additionally the network has a function that initiates early stopping and validation, which is used in regression. The network has one input, one output and one hidden -layer of neurons.

1. Preparing the data

- As the data is the same as in logistic regression it was also prepared in the same manner, editing the same values and scaling it, see 3.1.1.

- The data was first randomly shuffled and then divided in the same way as before, 10% test data, 10% validation data and 80% train data.

2. Setting initial values

- Initial values were created by a function that sets the *biases* and *weights*. The weight were set with random samples from $N\left(\mu, \sigma^2\right)$ filling the matrices $(\textit{features}) \times (\textit{hidden-neurons})$ for hidden weights, and $(\textit{features}) \times (\textit{output-neurons})$ for output weights. The bias was set to $b = 0.01$, and fed into two vectors also with the appropriate sizes.

- The learning rate η, regularization parameter λ[1], number of hidden neurons n and batch size m, were chosen by the use of our hyperparameter-tuning function without early stopping. This is discussed later in section 3.3.

3. Feed forward

- The feed forward essentially creates a prediction for the hidden layer and the output layer respectively. It then feeds this to the activation function used. This process is then iterated over all epochs, and for each epoch it is iterated over the number of batches, $M = \frac{Inputs}{BatchSize}$. In between each batch iteration backpropagation is done.

- Our network permits the use of all the activation functions in section 3.4, and for different combinations of hidden activation functions and output activation function. Note that the data and number of categories isn't necessarily suited for all activation functions. We used the *Sigmoid(Sig)* for both hidden and output in the classification case. The subsequent equations follow from equation 22.

$$\begin{aligned} \hat{z}_h &= \hat{X}\hat{W}_h + \hat{b}_h \implies \hat{a}_h = \mathrm{Sig}\left(\hat{z}_h\right) \\ \hat{z}_o &= \hat{a}_h\hat{W}_o + \hat{b}_o \implies \hat{a}_o = \mathrm{Sig}\left(\hat{z}_o\right) \end{aligned} \tag{39}$$

[1] More on regularization in our previous article (9)

4. Backpropagation

- For a given round of iteration the backpropagation function receives the activation values z_o, a_o, a_h described in the feed forward part. The process goes as follows:

  I. Using equation 32, our feed forward 39 results and the cross entropy(CS) gradient 38 to create the *output error*

  $$\hat{\delta}_o = \nabla \text{Sig}(\hat{a}_o) \cdot \nabla \text{CS}(a_o, t)$$

  II. We use equation 37 to create the *hidden error*, which takes the form

  $$\hat{\delta}_h = (\hat{\delta}_o \cdot \hat{W}_o^T) \cdot \nabla \text{Sig}(a_h)$$

  III. Using equation 33 and 36 we compute the hidden and output gradients and apply them together with λ and η to compute our weights and biases for the hidden layer

  $$\hat{W}_h \Leftarrow \hat{W}_h - \eta\lambda\hat{W}_h - \eta\hat{\delta}_h \cdot \hat{X}$$
  $$\hat{b}_h \Leftarrow \hat{b}_h - \eta\hat{\delta}_h$$

  and for the output layer

  $$\hat{W}_o \Leftarrow \hat{W}_o - \eta\lambda\hat{W}_o - \eta\hat{\delta}_o \cdot \hat{a}_o$$
  $$\hat{b}_o \Leftarrow \hat{b}_o - \eta\hat{\delta}_o$$

5. Evaluation

- The parameters of the network were iteratively evaluated using the AUC-score 3.5.1 and updated and changed according to the score.

3.2 Regression Analysis: Franke Function

This section is largely a follow up on our previous article (9), where we applied different forms of regression to fit the *Franke Function*. In this article we wanted to use our neural network structure to predict the Franke Function

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+\frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &-\frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right) \end{aligned} \tag{40}$$

We extract the data in the same way as in our previous article. In short, the interval $[0,1]$ is divided into a a grid of a suiting number of points, n. The number of data points that can be fed to the model is then n^2. The grid is used to create a design matrix of a given polynomial degree($p = 4$ was mainly used) and also values of the Franke Function[2].

3.2.1 *Neural Network: Regression*

The network was trained in the same fashion as with the classification 3.1.2. The main differences were

I The data was split into train-validation-test(64,12.96,23.04). The validation set this was used for hyperparameter tuning with early stopping to find the optimal parameters, see later section 3.3.This involved testing the model, trained on the train-data, for every epoch on the validation set. This was done using MSE as the metric, and checking if the early stopping conditions were met. The best parameters were then used to train the model and check the predictions against the test data set.

II The MSE cost function 24 was used instead of the Cross Entropy function, as the former is most suited for regression.

III *ELU* was used as the output layer activation function, and *Sigmoid* as the hidden layer activation function.

IV The MSE was exclusively used as our metric of choice to evaluate the model.

V The data had to be backshuffled to be plotted, as the order of the data naturally matters when we're basically creating a landscape. We had stored the indicies of our data when we first shuffled. The design matrix of the test data had to be scaled to the size of the original design matrix $\boldsymbol{X}$, this was done by introducing zeros on all places other than the places of the test data.

3.3 Hyperparameter Tuning and Early Stopping

3.3.1 *Hyperparameter tuning*

The hyperparameter tuning function creates a matrix of size $(parameters \times configurations)$. The parameters used were the learning rate η, regularization λ, batch size m, polynomial degree p, hidden neurons n. Each were divided into an equal number of values over large intervals. The limits, though large, had to be set by intuition. The rows were then shuffled independent of each other. Next the function runs a loop that iterates over the columns in the matrix. How many columns to iterate over is simply determined by how much time you have at your disposal. For the classification case the

[2] For more on this, see (9)

function would train a network and check the AUC-score for each parameter, and return the parameters that gave the highest AUC.

3.3.2 *Early Stopping*

For the regression case the Neural Network class would run a *validation and early stopping* function that we used in conjunction with the tuning process. The function would make a prediction on the validation data set and check the MSE score against the true validation data values for each epoch. If the function encountered any of the following conditions the training process would stop and return the last best value before the relevant condition was met

I After an increase in the MSE was encountered, it would store the last MSE value before the increase and then start and add every following MSE for a predetermined number of epochs, and then calculate the average. If the average was larger than than the stored MSE value it would return it as the value for that round of tuning.

II If the MSE was approximately flat for three epochs it would return the MSE value. By approximately we mean that the values compared were rounded to a given decimal point. This eliminated many rounds that consisted of very slow depreciating learning curves.

III If the MSE had a spike, defined as $MSE_{i+1} = 10 \times MSE_i$, it would return the last MSE value. These spiking learning rates were never seen to behave properly for later epochs and were thus stopped.

IV The conditions above work in conjunction with the condition that the network had to have been running for a certain number of epochs first. This catalyst criterion, together with the parameter deciding number of epochs for the average MSE, were the "trade-off" parameters of the tuning. Setting the catalyst too low could hinder slow starters from reaching their minimum. Setting it too high would lower efficiency by expending computational power on calculating divergent graphs and ones that reached their possible permanent minimum early and started growing.

3.4 Activation Functions

In the calculations performed in this article the Sigmoid and ELU were mainly used. Leaky Relu are all available for use in our code. In figure 3 one can see how the activation functions behave.

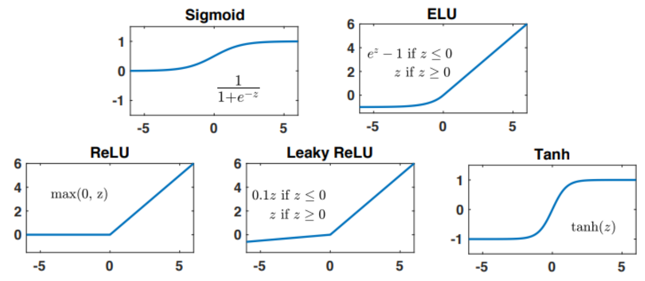


FIG. 3.— Activation functions. Image borrowed from (4, p. 48) and edited slightly

3.5 Metrics

3.5.1 *ROC and AUC*

The *Receiver operating characteristic*(ROC)(8) is a curve often used for graphical illustration to asses the quality of a model and to find the optimal classification, see figure 10 for example. In the case of credit card default one might classify a prediction(a probability), as 'probably will *default*(1)' or 'probably won't default(0)'. To be able classify the data one must define a threshold, say 50%. Meaning that those with over 50% default chance will be classified as *default*. In this case there might be those that did not default, but were incorrectly classified as *default*(above 50%), this is called *False Positives*. There might also be those who did default, but were incorrectly classified as *default*, these cases are called *False Negatives*. The two other possibilities is when we classify correctly: *True Positives* and *True Negatives*. If we change the threshold we will get different numbers of the four scores above, and this is exactly what is done with the ROC curve. A value on the horizontal axis, for a given threshold is calculated by

$$\text{True Positive Rate} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{41}$$

and the vertical axis by

$$\text{False Positive Rate} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}. \tag{42}$$

A diagonal line is also usually drawn to graphically show where $TruePositiveRate = False$. The ROC is simply put a graph that plots the *hit rate* against the *false alarm rate*

AUC

The *Area Under the Curve*(AUC) is exactly that, the area under the curve of the ROC curve. This metric can be used to compare one ROC curve with another ROC curve to assess which one has the greatest area under the curve, and thus is the best model.

3.5.2 *Cumulative Gain Chart*

A *Cumulative Gain Chart* is, as with the ROC curve, a way to evaluate a model, see figure 8 for example. After we obtain a set of probability predictions from our model they are sorted from highest probability to lowest. Meanwhile we keep track of the actual binary predictions that correspond to each probability. Table 1 shows an example. We can now imagine that we take the e.g. the 10% highest probabilities of the sorted predictions. This would correspond to the fist row in table 1. We then check what percentage of the total number of defaults(1) this corresponds to. In our example this would be $\frac{1}{5} = 20\%$. Hence our first point for the gain chart would be $(x_1, y_1) = (10\%, 20\%)$, the next percentage of 20% would include the top two clients, so $\frac{1+0}{5} = 20\%$ which is the same because the second client did not default. So $(x_2, y_2) = (20\%, 20\%)$. We continue doing this until we have gone through all the data, and then we have the *Cumulative Gain Chart*(12). Now the ideal case would be if all the defaults(1) were on the top rows such that for every percentile step we took we would include only defaults until all of them had been counted, after that the curve would stay flat. This can be seen in figure 8. The line along the diagonal is what would happen if our model was no better than flipping a coin. This is what we would expect if instead of sorting the data we shuffle it and pick out data for every percentile. If for every percentile batch we take out of the sorted data we get more or less the same cumulative percentage along the vertical axis(y) as the diagonal line, then our model is no better than chance(13).

TABLE 1
Example of a sorted table used for calculating a gain chart.

Default(actual)	Predicted probability
1	0.97
0	0.8
1	0.7
1	0.66
1	0.55
0	0.4
0	0.2
0	0.2
1	0.1
0	0.1

Area Ratio

This metric is defined as

$$\text{Area Ratio} = \frac{\text{Area under Model Curve} - \text{Area under Random Curve}}{\text{Area under Ideal Curve} - \text{Area under Random Curve}} \qquad (43)$$

The higher the area ratio score is the better the model. This is intuitive as it means it's approaching the ideal model. If it was as good as the ideal model then the Area Ratio would be equal to unity(5).

4 Results and discussion

4.1 Comparison with Scikit-learn

To compare our different model-structures we used the corresponding packages from Scikit-learn (7). The results for logistic regression(LG) with stochastic gradient descent(SGD) and neural networks can be seen in table 2. Practically the same neural network structure, but with different parameters was used for the Franke function, thus a comparison with Scikit-learn just for the credit card data should be sufficient to asses our code. For neural networks(ANN) we got similar values to Scikit-learn, but not identical. To the best of our knowledge we used the same parameters and initial conditions. We couldn't pinpoint the exact reason for the small difference in results, but as they were similar up to the third decimal point we treat the difference as negligible and consider the similarity as confirmation that our neural network code is well functioning.

TABLE 2
Comparison between our models' architecture with Scikit-learn's.

Model	AUC-score
Our Neural Network	0.7777715333367413
Scikit-learn Neural Network	0.7764693389254856
Our Logistic Regression(SGD)	0.703020235909391
Scikit-learn Logistic Regression	0.708169471550586

Model	epochs	η[a]	λ	batch-size	neurons$_{hid.}$
ANN	20	0.0008[b]	0.01	25	41
LG-SGD	20	0.002069	0	1	

[a]Learning rate: η, Regularization term: λ
[b]This gives the same results as $\eta \approx 0.1$. They differ in AUC-score at the 4th decimal point

4.2 Credit Card data

4.2.1 *Correlation Matrix*

The correlation matrix for the different features are shown in figure 4. While we did not tune our data beforehand to account for linear dependency, the correlation matrix gives an idea of which features would be good candidates to possibly trim. Having two or more correlated variables can potentially produce highly fluctuating coefficients/weights, that in turn, if significant, can reduce our ability to asses the strength of a single coefficient and its corresponding feature on the risk of default. The system can as such become a black box, where the total model with all the features has good predictive power, but we can't pinpoint which features that mostly predict the chance of default. This phenomenon is called *multicollinearity*, and is discussed at length in our first project (9). Correlated features can also create redundancies and more local minima for the gradient to transverse and in turn unnecessarily increase the complexity of the network.

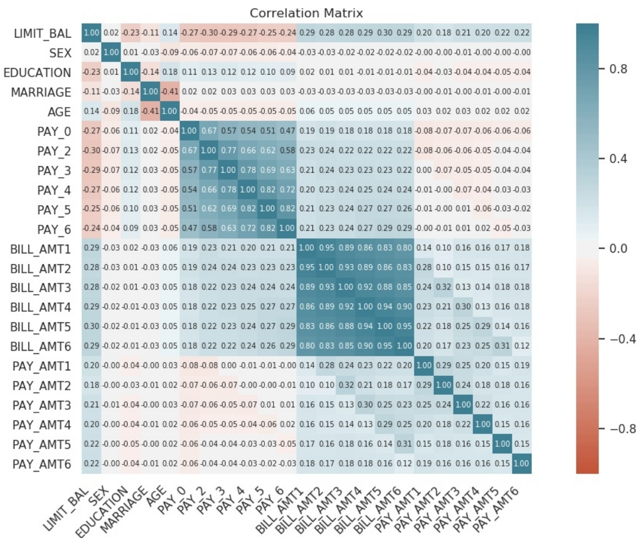


FIG. 4.— Correlation Matrix

4.2.2 *Logistic Regression*

Our goal for the credit card data was to create models based on logistic regression and neural network to predict the chance of default of credit card clients and compare their predictive power by the use of several metrics 3.5. For the logistic regression we used three different types of gradient descent covered in 2, the *Gradient Descent*(GD), *Stochastic Gradient Descent*(SGD) and *Mini-batch Stochastic Gradient Descent*(MBSGD). We also applied Newton's method, but the results were omitted from this article as it lacked consistency[3]. It was highly dependent on the initial condition, specifically the initial coefficient which we chose randomly from the normal distribution for all methods. Delving into methods for picking an initial coefficient with local convergence(as is needed with Newton's method) is beyond the scope of this article. The results for the logistic regression for different optimization methods can be seen in table 3, with the metric AUC. In figure 6 we plotted ROC curves with optimal learning rates and with number of epochs sufficient for convergence. In figure 7 we did the same but now with the same number of epochs(20) for each model. The two plots illustrate how the non-stochastic method of GD is dependent on running a sufficient amount of epochs before it converges to its minimum, while the stochastic ones will have a representative sample of the data-landscape early on because it pulls out random examples from the data and calculates the gradient. The gradient calculated by picking examples at random will resemble the gradient calculated for the total data for that given epoch long before all the data has been iterated over. After running through the data a sufficient amount of times the non-stochastic GD can be seen in our case to converge to the stochastic methods 7.

In our analysis we did not see any significant difference in metric performance between the SGD and MBSGD see 10 and table 3, where the former can be seen as a special case of the latter but with batch size $\equiv 1$. We chose for our logistic regression a MBSGD with batch size of $= 32$. What we would expect is the MBSGD giving a lower AUC than the SGD for low number of epochs, this did not come to fruition first time around. To illustrate the difference between different batch-sizes and between those models and non-stochastic GD we represent a figure with four different batch-sizes: 1, 5, 1000 and 6000. The results can be seen in 5. Note that the horizontal axis shows $epochs(0) = [0], epochs(1) = [0,1], epochs(2) = [0,1,2], epochs(3) = [0,1,2,3]$ etc. The figure shows that the stochastic methods with smaller batch-sizes reaches their maximum(minimum weights) after only a few epochs. When the batch-size is increased to 6000 the convergence suffers. Time-wise the SGD is the most expensive. We also see that there is a simply relationship between time and batch-size. Increasing the batch-size only just a little, such as from batch-size $= 1$ to batch-size $= 5$ in the plot, has a relatively big effect on the time cost. To be specific it reduces the time taken by a factor equal to the ratio of the batch-sizes. Just looking at the plot 5 the relationship becomes very clear: At epoch 20 we see that $\approx \frac{t_{MB1}}{t_{MB5}} = \frac{50}{10} = \frac{Size(MB5)}{Size(MB1)} = 5$. Hence with a batch-size of 1000 the time taken is $\approx \frac{1}{1000}$ of SGD. Depending on how you look at it though, the SGD is not necessarily more time expensive than MBSGD_{6000} if we take convergence into account. As mentioned earlier, stochastic methods will have a representative sample of the data before it has gone through all the data, and more so for SGD than MBSGD_{6000}

For our data batch-size 5 and 1000 has little effect on the AUC-score. We see that they reach their maximum faster(epoch-wise) than SGD, which is interesting and might be data specific. Different initial values(random seeds) were tried, also giving the same result. MBSGD with the rightly chosen batch-size seems to be superior. The rightly chose batch-size can be chosen from considering the time-convergence trade-off at play. The MBSGD_{1000} will time-wise converge much faster than the lower batch alternatives, but the epoch-wise(also implying iteration within epochs) convergence seems to suffer a bit. Lastly it is worth mentioning that all the stochastic methods will never really reach its optimal value since they are stochastic the gradient will fluctuate around its minimum, which can also be seen on the AUC in 5. This will not be the case for the non-stochastic GD, but it is usually a small price to pay for the stochastic methods. We also saw that the GD was very sensitive to its initial conditions(initial weight and learning rate), and that with a learning rate just slightly off its optimal it would not converge.

4.2.3 *Neural Network*

We found that a model created with neural network outperforms logistic regression models, where both the former and the latter has optimal parameters. This can be seen in table 3, and figure 6. This is in line with expectation. If we had a neural network with one hidden node, and one output node(with sigmoid activation), and

[3] But it is included in the code

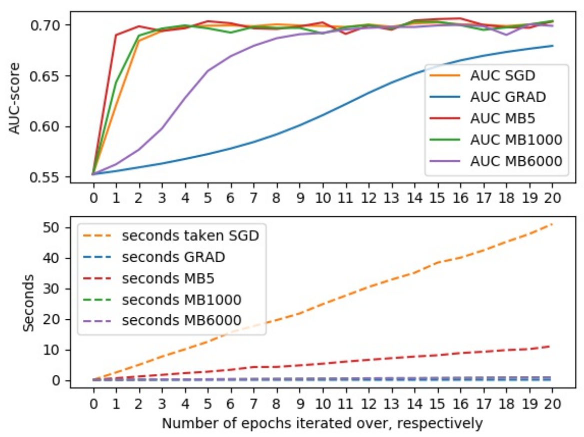


FIG. 5.— Top chart shows AUC scores for different epochs. For example for epoch = 3, the AUC is calculated for models which correspond to iterating over epoch $(0), (0,1), (0,1,2), (0,1,2,3)$. The bottom chart shows seconds used for each epochs calculation for the different optimisers

no bias, we would essentially have something similar to the logistic regression. In a sense, the logistic regression is a special case of the neural network with sigmoid. So it's reasonable to assume that our neural network should perform equally good or better if we increase the degree of complexity. Our model, with one hidden layer, could probably be optimized further. We did not run more than 1000 iterations on our hyperparameter-tuning function. The intervals of the parameters were divided up into 1000 steps, so there's $1 \cdot 10^{12}$ different combinations of the four parameters. Most would probably be killed off by early stopping though, but the point stands.

We compared our neural network model with the results in the paper (5). In the paper they achieved an $area - ratioscore$ of 0.55 for the training set and a score of 0.54 for the validation set. These scores are comparable to ours which were 0.55871 for the training set, and 0.54262 for the test[4] set. In figure 8 and 9 one can see our neural network model results for train and test sets in the form of gain charts. Gain charts are discussed in 3.5.2. The gain charts are also relatively similar to the ones in the paper (5).

4.3 Franke Data

In this section the results for the Franke functions will be presented and discussed. In project 1 we used Linear Regression to analyse the Franke function and in this Project we will further our work by applying Neural Network on Franke and then compare to our results from Project 1. After applying our hypertuning function 3.3 to find optimal parameters for the network we then used these parameters to obtain most of the results presented below with the exception of figure 13. The values

[4] In the scheme of comparing different predictive structures such as logistic regression and neural network to figure out which to use our test set would be a validation set nr 2. Our validation-test split in this articles relates to the validation process of picking ANN-parameters and then test the choice on a hold-out-set.

TABLE 3
RESULTS FOR DIFFERENT MODELS ON CREDIT CARD TEST DATA. THE ANN LISTS THE RESULT OF TWO HYPERPARAMETER RUNS, WHERE ANN_1 AND ANN_2 WERE RESPECTIVELY ITERATED OVER 500 AND 2000 CONFIGURATIONS. ANN: ARTIFICAL NEURAL NETWORK, LG: LOGISTIC REGRESSION. η, λ, B, n_h ARE RESPECTIVELY THE LEARNING RATE, REGULARIZATION PARAMETER, BATCH-SIZE, HIDDEN NEURONS.

Model	AUC-validation	AUC-test	Area ratio-test
ANN_1	0.78428	0.77110	0.54262
ANN_2	0.78331	0.75749	0.51538
LG-GD	0.70058	0.69316	
LG-SGD	0.73024	0.70302	
LG-MBSGD	0.73072	0.70326	

Model	epochs	η	λ	b	n_h
ANN_1	20	0.0008	0.01	25	41
ANN_2	20	3.1423	0.01254	26	66
LG-GD	30	0.000068	0		
LG-SGD	20	0.000792	0	1	
LG-MBSGD	20	0.003862	0	32	

FIG. 6.— ROC curve for the different methods with different epochs. The number of epochs are such that it is more than sufficient for convergence. Epochs in order of the labels: 300, 20, 20, 20

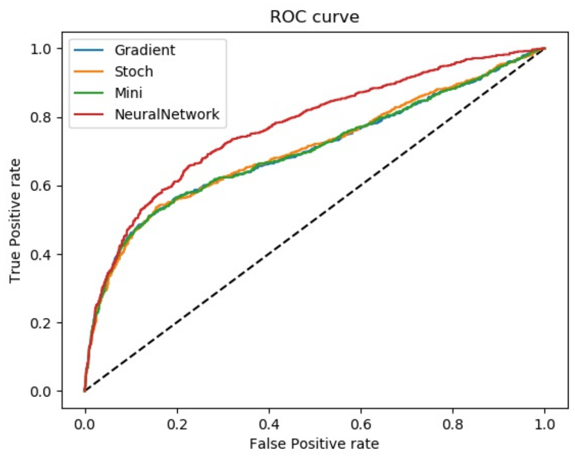
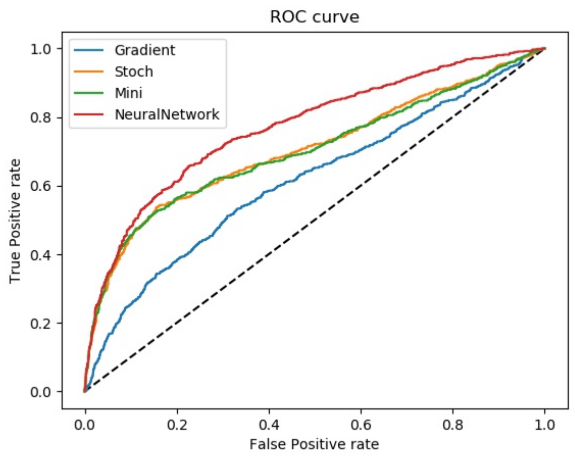


FIG. 7.— ROC curve for the different methods with same number of epochs for each. $Epochs = 20$.



used where $\eta = 3.16227766e-01$, $\lambda = 2.68269580e-08$,

FIG. 8.— Cumulative Gain Chart for Neural Network for the test data

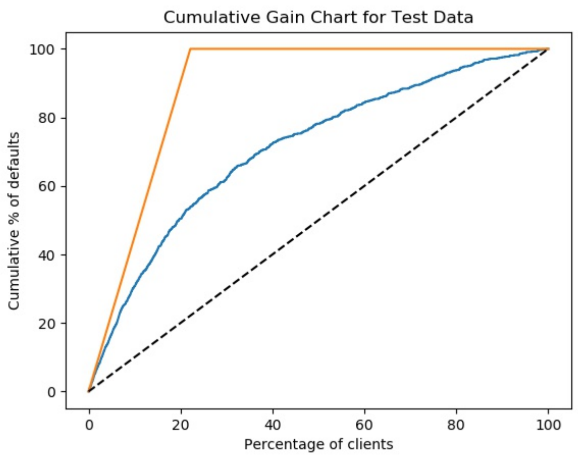
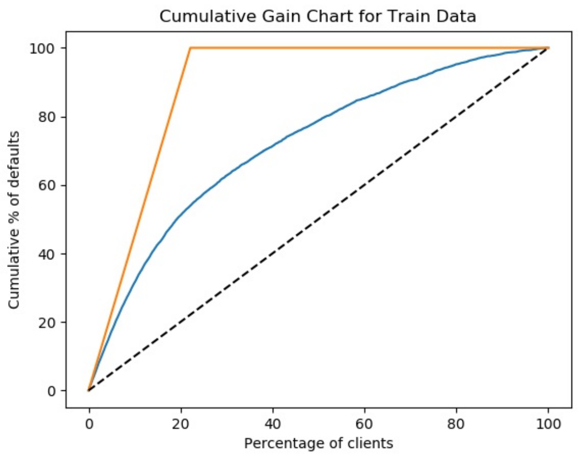


FIG. 9.— Cumulative Gain Chart for Neural Network for the train data



batchsize = 1, hidden neurons = 57, categories = 1, number of polynomials = 4 and epochs = 91. In figure 10 and 11 one can see the real Franke function compared to the fitted test data and the fitted train data without noise.

The Franke was 10000 data points with the training data being 64%, test being 23.04% and validation data being 12.96%. The large number of points(compared to the 400 used in (9)) was mainly chosen to be able to visualise the 3D plots better. The data splits were chosen in descending size order, but the exact percentages somewhat arbitrary except that the train set size had to be large enough[5]. With smaller data sets it was harder to visualise how well the model fitted to the data. In figure 11 one sees the scattered plot of the train and test data. As mentioned in the Method section 3 the data was first shuffled than the network trained with the training data and tested against the test data. To visualize how well the the train and test data perform we used a back shuffle method to order the data back to right indecencies and

[5] Fun fact: The sizes are all perfect squares $80^2 + 48^2 + 36^2 = 100^2$

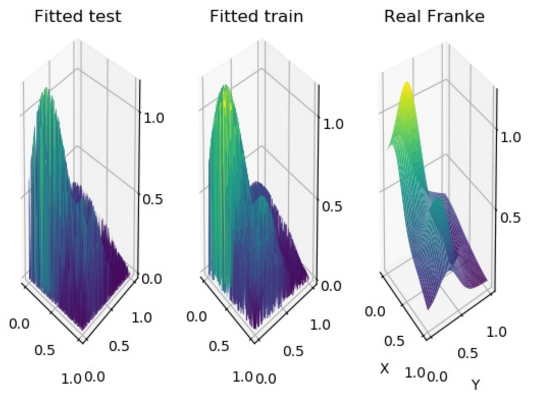


FIG. 10.— Fitted Franke with test and train data with no noise. Surface plot

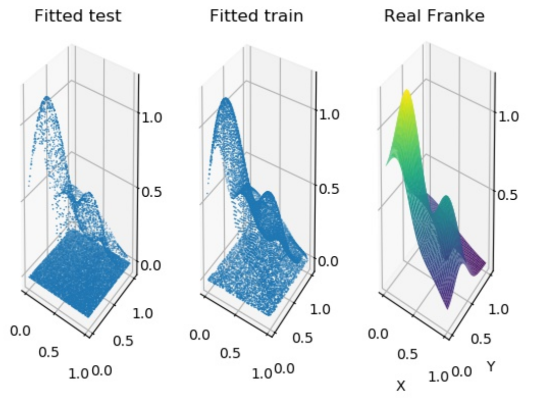


FIG. 11.— Fitted Franke with test and train data with no noise. Scatter plot.

make it as big as the design matrix originally was. Since the test data is only 23.04% of the real design matrix the values missing where chosen to be zero. This is why one sees some of the values being zero in the plots, and this is most apparent in the scattered version.

As one can see from tables 4 and 5 the MSE for both train and test are fairly low and R^2-score is well over 0.90. The Neural Network performs well without noise. When noise is added the network MSE goes up and R^2-score goes down as is expected. This is the same as was found in project 1. Noise = 0.1 is a high noise-term and in project 1 for Ridge Regression it gave an MSE around 0.02 for $\lambda = 0.00016$ (9) see figure 17, and R^2-score just under 0.6 for polynomial degree 4. The Neural Network seem to perform just as well with this noise term, and has a significantly higher R^2-score. This said, the Neural Network has several parameters at play which makes it more complex so for the purpose of fitting the Franke function, Linear Regression analysis is sufficient enough.

In figure 11 one can see the 3D plot of the Franke function with noise 0.1. The plot is not as smooth as

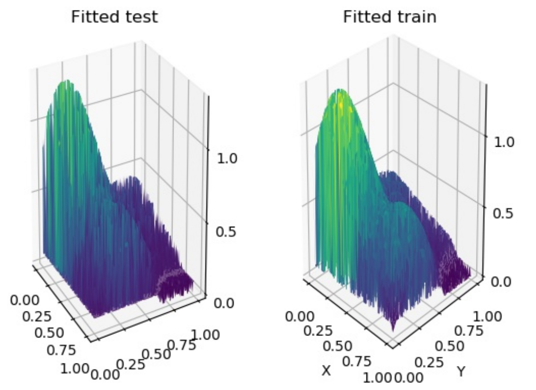


FIG. 12.— Fitted Franke with test and train data with noise $n = 0.1$

the plot without noise and this results in a higher MSE and lower R^2-score. This illustrates the reason for many data points, one an then see more clearly where the network is fitting to the noise-term in comparison to the plot without noise.

TABLE 4
THE MSE AND R^2-SCORE FOR THE TEST DATA WITH NEURAL NETWORK WITHOUT NOISE AND WITH NOISE 0.1.

Noise	MSE_{test}	R^2_{test}
Neural Network		
0	0.00125	0.985
0.1	0.0272	0.711

TABLE 5
THE MSE AND R^2-SCORE FOR THE TRAIN DATA WITH NEURAL NETWORK WITHOUT NOISE AND WITH NOISE 0.1.

Noise	MSE_{train}	R^2_{train}
Neural Network		
0	0.00123	0.985
0.1	0.0272	0.711

To see if our model overfitted to the noise term we plotted the MSE for the train data and the validation data in 12. The Network does not overfit according to theory[6] even after running with 1000 epochs. Both the training and the validation data becomes more noisy, and this is what we also see in the 3D plot in figure 11. According to theory the validation term should start overfitting to the data and then rise. This may be because we did the hypertuning and then found good values for our Neural Network, and therefore the λ value found does its work and regularize the weights like it is supposed to. Therefore different values for the parameters where tried to see if we could get our validation data to overfit to the noise term.

[6] Bias-Variance Trade-off analysis(9)

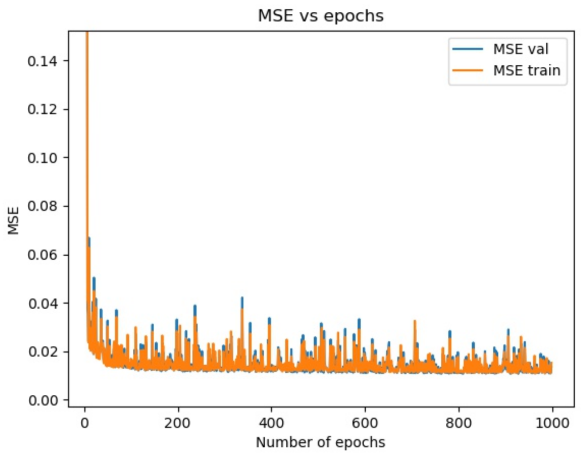


FIG. 13.— MSE for Franke train data vs. validate data for 1000 epocs with noise $n = 0.1$

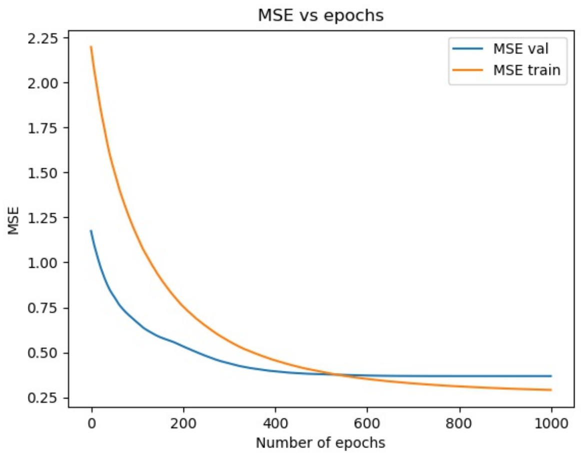


FIG. 14.— Overfitting for Franke train data vs. validate data for 1000 epocs with noise $n = 0.1$.

In figure 13 the parameters where the following: $\eta = 6.95192796e-05$, $\lambda = 1.27427499e-12$, batch size $= 42$, hidden neurons $= 60$ for 1000 epochs. The data points where also lowered to 400 since smaller data sets increase the possibility of overfitting. The validation set was also reduced to 5% and the training set was 90%. Here we see the validation set overfitting to the noise $= 0.1$. This was also the general rule for other plots with this configuration.

5 Perspective for future improvements

Firstly and mainly we like to improve upon our neural network to see if we can push the AUC-values and Area ratios higher. We were under the impression from what we heard from our peers and student-teachers that one layer for our neural network would be more than sufficient. This is also implied from the *universal approximation theorem*(4, p. 47). But we would still like to try it out, and see what we could do with that extra layer.

Further on there is a long range of activation function we could try out to see if we get any better results. Also, to be frank, there's probably a lot of theory and models that we simply haven't got the chance to get the knowledge of just yet that could possibly improve our results.

Secondly we would like to improve the time efficiency of our model. Simply make it run faster. We have the impression from running Scikit-learn that much could be done in terms of improving the code.

Thirdly we would like to improve our hyper-tuning function so that it catches more of the right cases, and discards more of the wrong ones. This would also let us do the random grid search over more configurations as it would be faster.

6 Conclusion

While computationally more expensive, we have seen that a Neural Network performs better than Logistic Regression for the classification case with the Credit Card Data in terms of the metrics AUC and Area ratio. Likewise, our Neural Network outperforms our Linear Regression results from our last project **??** on the Franke data in terms of the R^2-score, which fared significantly better. While it did not perform markedly better for the MSE score for Ridge Regression from the last project, we feel confident that the models we created with Artificial Neural Networks are the champions of both our data results. Apart from the prediction results, this article also reflects our learning journey into many of the concepts related to data analysis. The different types of metrics, activation functions, the general intricacies of convergence and the complexities surrounding the structure of neural networks contains a plethora of information and knowledge we feel elated to have been able to delve into.

7 Appendix A: Table

8 Appendix B: Link to programs

Link to the project in Github

REFERENCES

[1]H.J. Morten. Data Analysis and Machine Learning: Logistic Regression - lecture notes fall 2019,(2019).
[2]H.J. Morten. Data Analysis and Machine Learning: Optimization and Gradient Methods - lecture notes fall 2019,(2019).
[3]H.J. Morten. Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning - lecture notes fall 2019,(2019).
[4]Mehta et. al. A high-bias, low-variance introduction to Machine Learning for physicists, May 29, 2019
[5]Yeh, Ivy & Lien, Che-Hui. (2009). The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. Expert Systems with Applications. 36. 2473-2480. 10.1016/j.eswa.2007.12.020.
[6]Yeh, Ivy & Lien, Che-Hui. (2009). https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients
[7]http://scikit-learn.sourceforge.net, scikit-learn, Scikit-learn: Machine Learning in Python, author: Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E.
[8]Fawcett, Tom (2006). "An Introduction to ROC Analysis" https://people.inf.elte.hu/kiss/11dwhdm/roc.pdf
[9]Per-Dimitri B. Sønderland and Vala M. Valsdóttir. Regression analysis and resampling methods, https://github.com/valamaria89/FYS-STK4155-Project1
[10]M. Markova & V. M. Valsdottir Computational Physics II: Machine Learning Approach Applied to Solutions of Quantum Mechanical Many Body System
[11]Cory Maklin Logistic Regression in Python - Towards Data Science
[12]IBM® Knowledge Center, https://www.ibm.com/support/knowledgecenter/de/SSLVMB_24.0.0/spss/tutorials/mlp_bankloan_outputtype_02.html
[13]Galit Shmueli (2019), Lift Up and Act! Classifier Performance in Resource-Constrained Applications

TABLE 6
Explanation of the different data features in our data set. These explanations are quotations from I-Cheng Yeh and UCI's website (6)

Data features(Client attributes) explanation	
X1	'Amount of the given credit (NT dollar): It includes both the individual consumer credit and his/her family (supplementary) credit.'
X2	'Gender (1 = male; 2 = female)'
X3	'Education (1 = graduate school; 2 = university; 3 = high school; 4 = others)'
X4	'Marital status (1 = married; 2 = single; 3 = others)'
X5	'Age (year)'
X6 -X11	'History of past payment. X6 = the repayment status in September, 2005; X7 = the repayment status in August, 2005; . . .; X11 = the repayment status in April, 2005. The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; . . .; 8 = payment delay for eight months; 9 = payment delay for nine months and above.'
X12 -X17	'Amount of bill statement (NT dollar). X12 = amount of bill statement in September, 2005; X13 = amount of bill statement in August, 2005; . . .; X17 = amount of bill statement in April, 2005.'
X18 -X23	'Amount of previous payment (NT dollar). X18 = amount paid in September, 2005; X19 = amount paid in August, 2005; . . .; X23 = amount paid in April, 2005.'