Department of physics, University of Oslo

Logistic regression and Neural Networks

Jon A Ottesen

(Dated: 13. november 2019)

For this project, the aim is study logistic regression and neural network in a binary classification example resolving around credit card data and regression analysis of the franke function. In the binary classification example, neural network performed better than logistic regression. Neural network had the highest F1 score with 0.543 and with accuracy of 0.789, whereas a F1 score of 0.527 and accuracy of 0.783 was the best created model for logistic regression. For regression analysis of the franke function, nerual network gave both stable and accurate result with the most accurate having $\mathrm{R}^2 = 0.941$ between the model and the actual franke function for the test data.

I. INTRODUCTION

With an increasing amount of data, both in respect to parameters and quantity, statistical knowledge is essential for any type of analysis. This ranges from scientific work to stock marked prices and medical imaging. The type of statistical analysis possible ranges from regression type models for curve fitting such as OLS, classification cases with logistic regression and neural networks and many more. These examples are just a few types of statistical tools under the broader term of supervised machine learning.

The main focus of this project resolves around the previous mentioned methods of logistic regression and neural network. These methods will be used in conjunction with various sampling techniques, to study a binary classification example resolving around a credit card data set. Unlike most studies it's not the data set which is the focus, rather the methods themselves. However a major goal is of-course to create a good performing model for the data set.

The final aspect studied in this report is the use of neural networks in a curve-fitting regression example on the franke function with added normally distributed noise. Moreover the performance of the neural network is compared to the performance of the more standard curve fitting algorithms: Ols, ridge regression and lasso regression.

II. THEORY

For most of the following theory the $\boldsymbol{X}$ matrix is a matrix with n-samples and p-categories (columns) such that

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & \dots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{np} \end{bmatrix}. \tag{II.1}$$

A. Preprocessing and sampling

There exists a multitude of different preprocessing techniques meant to improve the performance of a model. An example of such a technique would be dimensional reduction where the data consist of an overwhelming amount of parameters. This is however not important for the credit card data. Instead the technique focused on are: one hot encoding and re-scaling techniques such as standardization and normalization.

Lets assume that the column k in $\boldsymbol{X}$ represents a category(parameter) with s categorical features e.g yesterdays dinner. A regular representation for this would be to assign a feature to a specific integer e.g chicken = 0, beef = 1, pasta = 2 etc. This would however assign a weight for each feature, and making one feature more important than another. One hot encoding is a process which extends the number of categories in $\boldsymbol{X}$ by the number of features in a specific category. This means: one hot encoding is a process where categorical features in a column is converted into multiple binary 0,1 columns in $\boldsymbol{X}$. Moreover this would extend the size of $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ to $\boldsymbol{X} \in \mathbb{R}^{n\times p+s-1}$.

The next topic of discussion is the re-scaling technique standardization. Although normalization i.e scaling every element between $[0, 1]$ is a well studied and used technique I will be limiting myself to standardization. Given a sample category $\boldsymbol{x}$ (a column in bmX, the standardized version of the sample is given by

$$\boldsymbol{x}' = \frac{\boldsymbol{x} - \mu_{\boldsymbol{x}}}{\sigma_{\boldsymbol{x}}} \tag{II.2}$$

where μ is the mean of the category and $\sigma_{\boldsymbol{x}}$ is the standard deviation. The category bmx is therefore re-scaled such that bmx' has a mean $\mu = 0$ and standard deviation $\sigma = 1$.

In cases where a data set is highly biased, down sampling are often used to combat over-fitting. For a simple 0/1 binary example, this done by excluding biased data such that the ratio $\frac{S_0}{S_1}$ i.e the ratio of the number of 0 and 1 samples goes toward 1. This does however not mean that the ratio of 0 and 1 samples must equal, rather such that they are more balanced. Lastly, the exclusion of data samples should be carried out randomly or by removing undefined data. However, the removal off samples will result in the loss of potentially useful data for training, and this may affect the model.

B. Gradient decent

The theory in this part is found in or based off [1].

Gradient decent is a group of optimizing algorithms used in finding the ideal parameters for a model that minimizes some function. This function is often an error function e.g the cost function. As mentioned, there exists multiple types of gradient decent algorithms both in regards to the decent itself (finding the minimum) and the implementation.

Steepest decent is one of the simplest gradient methods. The idea behind steepest decent is that for function $C(\boldsymbol{\beta}), \boldsymbol{\beta} = (\beta_1, \beta_2, \beta_3...., \beta_n)$, decreases fastest if $\Delta\boldsymbol{\beta} = \gamma_t\,(-\nabla C(\boldsymbol{\beta}))$ for $\gamma_k > 0$. Thus the minimization of C based $\boldsymbol{\beta}$ is given by

$$\boldsymbol{\beta}_{t+1} = \beta_t - \gamma_t \nabla C(\boldsymbol{\beta}_t) \tag{II.3}$$

where γ_t is chosen such that $C(\boldsymbol{\beta_{t+1}}) \leq C(\boldsymbol{\beta_t})$. Furthermore the parameter γ_t is in most litterateur called the learning rate, as it will be here. Equation II.3 is repeated multiple times such that the F hopefully converges towards a minimum. However whether this minimum is a global or local minimum is difficult to determine even when testing for multiple initial values of $\boldsymbol{\beta}$. Intuitively steepest decent is method were one goes in the opposite direction of the steepest increase.

Another gradient descent method is the ADAM optimizer, and the main difference from steepest descent is that it 'remembers' the previous gradient to adapt the learning rate. ADAM much like steepest descent is a collection of equations used iteratively:

$$\boldsymbol{g}_t = \nabla C(\boldsymbol{\beta}_t) \tag{II.4}$$
$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \tag{II.5}$$
$$\boldsymbol{s}_t = \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \tag{II.6}$$
$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t} \tag{II.7}$$
$$\hat{\boldsymbol{s}}_t = \frac{\boldsymbol{m}_t}{1-\beta_2^t} \tag{II.8}$$
$$\boldsymbol{x}_{t+1} = \boldsymbol{\beta}_t - \gamma_t \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{s}}} + \epsilon} \tag{II.9}$$

where β_1 and β_2 are constants often taken as 0.9 and 0.99, while $\epsilon \sim 10^{-8}$. For the hat notation, this is just a notation quirk for the ADAM algorithm and nothing else.

As mentioned in the beginning of this subsection, there are multiple ways of implementing a gradient descent algorithm. Stochastic gradient descent is one method. Lets assume that the function C is not only a function of β, but also data samples from X and the corresponding y.

Both X and y are measured quantities and cannot be changed, however the number of samples and which samples given in C can be changed. Stochastic gradient descent is a implementation of gradient descent where the used samples in the function C is limited and randomized. The samples passed in C is referred to as a batch. When the total number of samples in the iterative process is equal to or greater than total number of samples in X, an epoch has passed. In stochastic gradient descent and most supervised machine learning, it is normal to state the number of epochs instead of iterations.

A general take on a stochastic gradient descent type of algorithm may look like this:

Algorithm 1 Gradient descent

```
Define cost function derivative and constants
for 1, 2, 3, 4... epochs do
    for t = 1, 2, ... iterations in epoch do
        Randomize new batch from X and y
        Compute $\nabla C(X_{batch}, y, \beta)$
        In-between calculations if ADAM
        Update $\beta$ with $-\gamma_t \nabla C(X_{batch}, y, \beta)$
    end for
end for
```

C. Logistic regression

The theory presented in this subsection can be found at [2].

In this subsection we will consider a design matrix $\boldsymbol{X}$ with p categories that has n-samples i.e the same as the II.1, with the response or outcome $\boldsymbol{y}$ with n-samples ($n \times 1$ vector). This outcome is a binary outcome with two possible responses, either 1 or 0 i.e true or false.

Logistic regression is a so-called 'soft'-classifier. Thus the result of logistic regression is the probability that the data points in the categories for $\boldsymbol{X}_i$ belongs to a specific response in $\boldsymbol{y}_i$. The probability of a specific event is given by the sigmoid function:

$$p(t) = \frac{1}{1 + \mathrm{e}^{-t}} \tag{II.10}$$

where t is just some arbitrary input variable. The goal of logistic regression is therefore to find the input t given the parameters in $\boldsymbol{X}_i$ to classify the correct response in $\boldsymbol{y}_i$. For this I will assume the correct input is given by the linear formula

$$\boldsymbol{t}_i = \boldsymbol{X}_i \boldsymbol{\beta}^T \tag{II.11}$$

where $\boldsymbol{\beta}$ is a row vector such that $\boldsymbol{\beta} \in \mathbb{R}^{p \times 1}$. The probabilities from the sigmoid can therefore be written in the following form:

$$p\left(\boldsymbol{y}_i = 1 | \boldsymbol{X}_i, \boldsymbol{\beta}\right) = \frac{1}{1 + \mathrm{e}^{-\boldsymbol{X}_i \boldsymbol{\beta}^T}} \tag{II.12}$$

$$p\left(\boldsymbol{y}_i = 0 | \boldsymbol{X}_i, \boldsymbol{\beta}\right) = 1 - p\left(\boldsymbol{y}_i = 1 | \boldsymbol{X}_i, \boldsymbol{\beta}^T\right). \tag{II.13}$$

The last case comes from the fact that there are only two responses, thus they will be dependent on each-other. Moreover, in equation II.13 it's common practice to add a intercept term β_0 to $\boldsymbol{X}$ by adding a column vector consisting of 1's for centering purposes. As a common trope among regression problems there is of course a cost function to minimize:

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(\boldsymbol{y}_i \left(\boldsymbol{X}_i \boldsymbol{\beta}^T \right) - \log(1 + \mathrm{e}^{\boldsymbol{X}_i \boldsymbol{\beta}^T}) \right). \tag{II.14}$$

The derivative of equation II.14 with respect to $\boldsymbol{\beta}^T$[1] can be written as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}^T} = -\sum_{i=1}^{n} \left(\boldsymbol{y}_i \left(\boldsymbol{X}_i \right) - \boldsymbol{X}_i \frac{\mathrm{e}^{\boldsymbol{X}_i \boldsymbol{\beta}^T}}{1 + \mathrm{e}^{\boldsymbol{X}_i \boldsymbol{\beta}^T}} \right). \tag{II.15}$$

Expression II.15 can be neatly tied up by matrix multiplication resulting in the following expression

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}^T} = -\boldsymbol{X}^T \left(\boldsymbol{y} - \frac{\mathrm{e}^{\boldsymbol{X}\boldsymbol{\beta}^T}}{1 + \mathrm{e}^{\boldsymbol{X}\boldsymbol{\beta}^T}} \right). \tag{II.16}$$

[1] I should just have defined beta as a column vector from the beginning, that would have made the notation easier.

The remaining parts of logistic regression i.e finding the optimal $\boldsymbol{\beta}$ to minimize the cost function is labeled a gradient decent problem. Equation II.16 would than be treated as the ∇C form section II B.

D. Neural Network

The theory for this subsection can be found at [3].

Neural network much like logistic regression is a method that can be trained by gradient decent type algorithms. Unlike logistic regression, neural network can be extended from classification to perform regression analysis among others.

There exists a multitude of different neural networks such as convolution neural networks and recurrent neural networks. Although newer methods have been devised, our focus will be on the simplest type of artificial neural networks; the feed-forward neural network.

1. *Feed forward*

Neural networks are based around the principle of having a given input parameter that is connected to another layer with neurons. In these connections there exist a weight, and in the neuron there exist i bias. This implies that the new value in the connected neuron from that specific input can be written as

$$z_j = w_{ij}x_i + b_j. \tag{II.17}$$

This implies that the value in the j'th next node is given by the i'th node in the previous layer multiplied with the weight of the connection plus the bias of that node. This is not the entire truth, just a small portion. The value of the j'th next node is a sum of all the previous features plus the bias. In matrix multiplication this can be written as:

$$\boldsymbol{z}^l = \boldsymbol{a}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)} \tag{II.18}$$

where l specifies the layer in the neural network. Furthermore $\boldsymbol{a}^{(l)}$ is of shape N_{l-1}, $\boldsymbol{W}^{(l)}$ of shape $N_{l-1}\times N_l$ and $\boldsymbol{b}^{(l)}, \boldsymbol{z}^{(l)}$ of shape N_l. The value given by $\boldsymbol{z}^{(l)}$ is not the output value of a given node. The output value of a node is given by

$$\boldsymbol{a}^{(l)} = f_l\left(\boldsymbol{z}^l\right). \tag{II.19}$$

Merging equation II.18 and II.19, the value for any given node as function of the previous nodes is given by

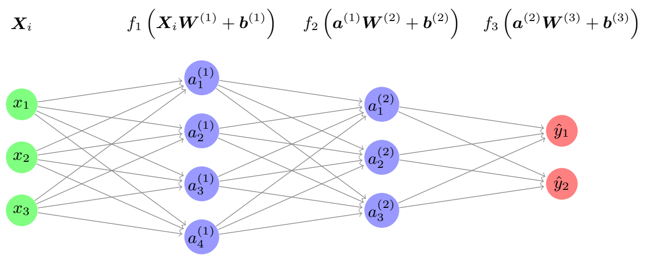
$$\boldsymbol{a}^l = f_l\left(\boldsymbol{a}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}\right) \tag{II.20}$$

where f_l is the so-called activation function. The activation function is function enforced upon every layer except the first initial layer, and usual activation functions include: the sigmoid defined in equation II.10, the tanh function, the relu function and the elu function among others. For more information see the appendix B 2. However, I will include the softmax activation function defined as

$$f(\boldsymbol{z}_j^{(l)}) = \frac{\mathrm{e}^{\boldsymbol{z}_j^{(l)}}}{\sum_{j=1}^{n}\mathrm{e}^{\boldsymbol{z}_j^{(l)}}} \tag{II.21}$$

and is special in the sense that it should only be used in final layer, preferably for a classification case. In a classification case it will give the probability for a specific node being the correct response and sum the probabilities will be 1.

I have tried to illustrate the entire forward process in a feed forward neural network in figure 1.


$\boldsymbol{X}_i$
$f_1\left(\boldsymbol{X}_i\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)$
$f_2\left(\boldsymbol{a}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right)$
$f_3\left(\boldsymbol{a}^{(2)}\boldsymbol{W}^{(3)} + \boldsymbol{b}^{(3)}\right)$
x_1
x_2
x_3
$a_1^{(1)}$
$a_2^{(1)}$
$a_3^{(1)}$
$a_4^{(1)}$
$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
$\hat{y}_1$
$\hat{y}_2$

Figur 1: An illustration of a neural network with the corresponding equation connecting each layer.

An interesting and very important consequence of equation II.21 is that the entire process is just a continuously mathematical mapping from the inputs to the outputs. Therefore, if I so desired the entire process could be written as one large equation.

In the terminology used, the node layers $\boldsymbol{a}^{(l)}, \boldsymbol{z}^{(l)}$ are treated as row vectors instead of the mostly used column vector terminology. This does however not change the end result.

2. *Back propagation*

To create of good preforming neural network model, the wights and biases has to be optimized. This process is called back propagation and is a gradient decent type of algorithm for neural networks. The general idea is to start at the end of the neural network and update the weights and biases backward i.e the name back propagation.

Much like logistic regression, neural network would not be complete without a cost function to minimize. The cost function in this case will remain undefined as $C(\boldsymbol{W}, \boldsymbol{b})$, where the actual cost function is dependent on the function of the neural network. The two used cost functions is given in the appending B 2.

An important aspect of the continuity of equation II.21 implies that the derivative with respect to the cost function can be studied in every layer as

$$\frac{\partial C}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial C}{\partial \boldsymbol{a}^{(l)}} \frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}} \tag{II.22}$$

$$\frac{\partial C}{\partial \boldsymbol{b}^{(l)}} = \frac{\partial C}{\partial \boldsymbol{a}^{(l)}} \frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}}. \tag{II.23}$$

The first term in both equation II.22 and II.23 is the derivative of the cost function based on the input parameter in cost function. This is only interesting for the output layer L. The derivatives of the cross entropy and mean squared error with respect to output layer are:

$$\text{MSE}' = 2(\boldsymbol{a}^{(L)} - \boldsymbol{y}) \tag{II.24}$$

$$\text{C-entropy}' = -\Big(\boldsymbol{y} \ominus \boldsymbol{a}^{(L)} + \Big(1 - \boldsymbol{y} \ominus (1 - \boldsymbol{a}^{(L)})\Big)\Big) \tag{II.25}$$

where I am applying hadamard division by $\ominus$ i.e element-wise division. The second term in equation II.22 and II.23 is just the derivative of the activation function used in the l-layer with respect to the input parameter $\boldsymbol{z}^{(l)}$. This is than dependent on the used activation function and the used activation functions with their corresponding derivatives is given in the appendix B 2.

The third term in equation II.22 are both surprisingly simple:

$$\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{a}^{(l-1)} \tag{II.26}$$

$$\frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{b}^{(l)}} = 1 \tag{II.27}$$

which follows from the linear relation $\boldsymbol{z}^{(l)} = \boldsymbol{a}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}$ defined earlier in the forward propagation.

The next step and a very important step is to define

$$\delta^l = \frac{\partial C}{\partial \boldsymbol{a}^{(l)}} \frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}} = f_l'(\boldsymbol{z}^{(l)}) \circ \frac{\partial C}{\partial \boldsymbol{a}^{(l)}} \tag{II.28}$$

with $\circ$ being the hadamard multiplication.

The layers can be connected by the chain rule to allow for backward propagation:

$$\frac{\partial C}{\partial \boldsymbol{a}^{(l-1)}} = \frac{\partial C}{\partial \boldsymbol{z}^{(l)}} \frac{\partial \boldsymbol{z}^{(l)}}{\partial \boldsymbol{a}^{(l-1)}} \tag{II.29}$$

$$= \frac{\partial C}{\partial \boldsymbol{z}^{(l)}} \frac{\partial}{\partial \boldsymbol{a}^{(l-1)}} \left(\boldsymbol{a}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}\right) \tag{II.30}$$

$$= \frac{\partial C}{\partial \boldsymbol{z}^{(l)}} {\boldsymbol{W}^{(l)}}^T \tag{II.31}$$

$$= \frac{\partial C}{\partial \boldsymbol{a}^{(l)}} \frac{\partial \boldsymbol{a}^{(l)}}{\partial \boldsymbol{z}^{(l)}} {\boldsymbol{W}^{(l)}}^T \tag{II.32}$$

$$= \delta^l {\boldsymbol{W}^{(l)}}^T. \tag{II.33}$$

The final result is **very** important

$$\frac{\partial C}{\partial \boldsymbol{a}^{(l-1)}} = \delta^l {\boldsymbol{W}^{(l)}}^T \tag{II.34}$$

since it allows for the possibility of backwards propagation when the gradient in the outer layer L is calculated. Before giving a explanation on how everything fits together it's worth noting that

$$\frac{\partial C}{\partial \boldsymbol{b}^{(l)}} = \delta^l. \tag{II.35}$$

Another important note is that by combining equation II.28 with II.34 results in the following relation

$$\delta^{l-1} = \delta^l {\boldsymbol{W}^{(l)}}^T \circ f'_{l-1}(\boldsymbol{z}^{(l-1)}) \tag{II.36}$$

where the transpose ensures correct dimensions.

To summarize everything in four equations:

$$\delta^L = f'_L(\boldsymbol{z}^{(L)}) \circ \frac{\partial C}{\partial \boldsymbol{a}^{(L)}} \tag{II.37}$$

$$\frac{\partial C}{\partial \boldsymbol{W}^{(L)}} = {\boldsymbol{a}^{(l-1)}}^T \delta^L \tag{II.38}$$

$$\frac{\partial C}{\partial \boldsymbol{b}^{(L)}} = \delta^L \tag{II.39}$$

$$\delta^{L-1} = \delta^L {\boldsymbol{W}^{(L)}}^T \circ f'_{L-1}(\boldsymbol{z}^{(L-1)}). \tag{II.40}$$

Algorithm 2 Backpropagation(GD = gradient descent)

```
Define cost function derivative and constants
for epochs and iterations do
    Randomize new batch from X and y
    Compute $\nabla C(X_{batch}, y_{batch})$
    Compute II.37 for last layer
    Compute II.22 and II.23
    Update weights and biases in last layer by GD
    for Layers backwards do
        Update delta by II.40
        Compute II.22 with II.38
        Update weights and biases by GD
    end for
end for
```

E. Model evaluation

To evaluate a binary classification problem it's unusual to use the cross-entropy cross function, although it's comply valid. The problem however, lies in the fact that a low loss in the cost function does not always equate to a good model. Furthermore, intuitively cross entropy is harder to grasp than the accuracy given defined as

$$\text{Accuracy} = \frac{1}{n}\sum_{i=0}^{n} I(y_i = \tilde{y}_i) \tag{II.41}$$

which is the fraction of correct predictions. In equation II.41 y_i is the correct values whereas $\tilde{y}_i$ is the predicted value.

For an unbiased data set, the accuracy of a model is a good estimate for the performance. However, for a biased data set this is not always the the case. For a binary classification problem with either 0 or 1 where 90% is 0, an accuracy of $\sim 90\%$ would not correspond to good model. This is because this accuracy could be achieved by only guessing 0. For a model to good performance it also has to be predictive. To determine this four terms is needed to be known[5]

1. TP: True positive is the number of predicted and correct 1 predictions.

2. TN: True negative is the number of predicated and correct 0 predictions.

3. FP: False positive is the number of predicted but incorrect 1 predictions.

4. FN: False negative is the number of predicted but incorrect 0 predictions.

From this the so-called precision and recall are defined as

$$\text{precision} = \frac{TP}{TP + FP} \tag{II.42}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{II.43}$$

and from this the F1 score can be defined

$$F1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{II.44}$$

The F1 score is a measure often used in unbalanced data and is the harmonic mean of precision and recall[5].

Another possibility model scoring is the area ratio i.e the area under the cumulative gain curve as done in [6]. This can be done using sklearnplot's module `plot_cumulative_gain`[2].

III. METHOD

A. Code implementation

1. Logistic regression

Logistic regression is implemented by first implementing some necessary equations. These include equation II.12 (sigmoid) and equation II.16 (cost function derivative). The next step was to initiate the initial β-values, which I did by setting them all to zero. What's left is just to implement a gradient descent algorithm for β by using the general algorithm 1 for ADAM.

A final, but not short note on how I implemented batches in the gradient descent. The usual way is to ensure that there are no repeated samples in the batch, however this implementation takes time. Instead, I chose batches at random which could lead to overlapping samples. For the direct implementation I would recommend looking at the ADAM method in Gradient in the program `reg_and_nn.py`. The reason for this implementation is that some minor tests showed that it is about 5-7 times faster.

[2] This was however not done due some technical difficulties. I simply not able to download the scikit-plot package, so the F1 scores were used instead.

2. Neural Network

Neural Network, much logistic regression is first implemented by implementing all necessary equations. These are the activation and cost functions in appendix B 2. The next step is to implement the stochasticity in the soon to come gradient descent, by batch implementation. Unlike logistic regression, this was done without overlapping samples, but still random between each iteration in the epochs.

With usable batches implemented, the next step is forward propagation. This is done similarly as in figure 1, but not for a sample at the time. The matrix multiplication in equation II.20 with the entire batch of $\boldsymbol{X}$ acting as $\boldsymbol{a}^{(0)}$. The weights are at first chosen at random in a normal distribution with $\mu = 0$ and $\sigma_w = \sqrt{2/\text{number of nodes}}$, while all biases are set to 0.01.

With forward propagation working the next step is backward propagation. This is done equivalently as in algorithm 2 with a general cost function that is specified when initiating the class. For my gradient descent I did use steepest descent rather than ADAM for simplicity.

B. Preparations

1. Credit card data

The data used can be found on following web page:

https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

with descriptions of the different attributes of the data set. In our case the entire data set is the design matrix $\boldsymbol{X}$ except the last column which is the correct values $\boldsymbol{y}$.

The first preparatory step before the analysis is preprocessing. The following columns: 1 (given credit), 5 (age), 12-17 (Bill statement for different months) and 18-23 (Previous payment) were all standardized by equation II.2 independently. The remaining columns where not standardized nor normalized.

Multiple of the columns in the credit card data was categorical with numerous features. To avoid unfair weighting, column: 3 (education), 4 (martial status) and 6-11 (payment history) were all one hot encoded. However in column 6-11 only the undefined values were one hot encoded and the -1 value although not undefined, and their previous value was set to 0 in the previous column. The same goes for other undefined categorical values in column 3 and 4. Another approach would be to remove rows containing undefined values, however I chose to keep them and instead treat them as another category.

After prepossessing, the data was divided into a test and training set at random with a 40/60 test-training ratio. This was done using sklean's train_test_split function. This split remained unchanged for the entire binary classification example, and every model was made on the training set. When stating that error estimates were calculated, I am impling that they were calculated based on the test set, not the training set unless specified otherwise.

The next preprocessing step is down-sampling. There exists no correct ratio for down-sampling. My choice was to use the ratio which maximized the norm of the accuracy and F1 scores. Furthermore I also chose the score which maximized the F1 score. These ratios are calculated by excluding 100 samples than 200 samples and so forth until the ratio between the samples are $\frac{S_0}{S_1} = 1$. During the exclusion of samples the error estimates; accuracy and F1 are calculated. This process is repeated N=5 times with the excluded samples being randomized after each run, where one run stops when the ratio is 1. When N=5 runs are repeated the mean of the error estimates pr down-sampled ratio is calculated. The ratios which maximized the norm and F1 are than calculated from the mean of the error estimates for the N runs. For this test, the learning rate and number of epochs remained constant. This is done both on logistic regression and neural network, but for neural network the network configuration remained unchanged. However this is done immediately for logistic regression, but not for neural network.

2. *Franke function*

Most of this part is just a redo of what is done in [4] with the same seed, shape i.e $81 times 81$ and the addition of normal noise with a mean of 0 and a standard deviation of 1. Thus ensuring that the data set is exactly the same as in the article. Furthermore, the train-test split is also carried out with the same split percentage and seed i.e 70/30 train/test split with seed 42 using the sklearn module mentioned in the previous sub-subsection. The design matrix used is of 5th degree complexity.

C. Logistic regression

With a down-sampled training set, multiple models were created for a grid of different learning rates and epochs. For all of these models the error estimates: accuracy and F1 scores were calculated. From this grid a good preforming model and the corresponding recall and precision score was calculated.

By using the previously discovered good model from the mesh-grid of learning rates and epoch the evolution of the error estimates was studied for different thresholds. Previously and for all further results a threshold of 0.5 has been used for determining whether the model predicts a 0 or 1. This was done by calculating the different error estimates for different thresholds including the precision and recall. The threshold which maximized the F1 score was also determined. Most of these calculations were carried out by sklearn's function `precision_recall_curve`.

In effort to maximize the predictiveness of a logistic regression model a different down-sampling ratio was chosen. The ratio chosen was S_0/S_1. For this new ratio a grid of models with different learning rates and epochs was made, and the error estimates were calculated for all the models in the grid. Furthermore, the test with varying threshold was also applied with the 1:1 ratio data set.

D. Neural Network Credit card

Before down-sampling it's necessary to find a stable network. For the entire training section of the data set the error estimates; accuracy and F1 was calculated for a meshgrid of different learning rates and epoch for four different activation functions in one hidden layer with constant neurons for all models. From this data, a stable configuration was chosen to use in the down-sampling process.

As in the logistic regression subsection, the down-sampling method is already described previously in subsection III B. The down-sampling ratio found will be used during all the results except when stated otherwise. As a final note, during the entirety of the credit card data analysis for neural network the softmax activation function from equation II.21 is used in the final layer and a batch size of 200.

With a down-sampled data, the next topic of study was how the performance of a neural network model changes depending on the learning rate, number of epochs and different activation functions in a hidden layer. Thus the accuracy and F1 scores were calculated for a multitude of models in a meshgrid of learning rates and epochs. Moreover a total of four grids were calculated, one for each activation function; the sigmoid, tanh, elu and relu. For all the models created, the number of neurons in the hidden layer remained constant.

The next step is to create a good preforming model. For this part I will be restrict myself to 'only' one or two hidden layers. During this entire test for both one and two layers the learning rate and number of epochs remained constant. For one hidden layer: all possible combinations of activation functions and nodes between 1 and 100 was tested. For these 400 models the error estimates: accuracy and F1 was calculated, and the models with the best individual performance for each estimate and norm was chosen. For two layers, models from a meshgrid with any combination of $[4, 14, 24, 34, 44, 54, 64, 74, 84, 94, 104]$ nodes in both layers with all possible activation function combination was created. For all models the error estimates were calculated, and the models with the best accuracy, F1 and norm between the error estimates was chose.

Choosing the best performing layer combination for two layers with regards to the F1 score, this combination was recreated a total of 300 times. For each of these recreations the error estimate F1 was calculated and all the F1 scores was plotted in a histogram.

The next step was to choose the models with the highest F1 score from both 1 and 2 layers. These models was than recreated for varying λ-values and the F1 score was calculated. The F1 scores was than plotted as a function of the λ-values, and the overflow models was emitted from the plot. Furthermore, the initial hidden weights were kept constant.

The last object of study is to maximize the F1 score. For this I estimated the ideal down-sampling ratio from a previous plot. With this down-sampling ratio I used layer combination with the best F1 score and recreated this model 300 times with the new down-sampling. For each recreation the F1 score was calculated and thereafter plotted in a histogram.

E. Neural Network Franke

The first step in this part is to change the cost function from the cross entropy used in the classification case to the MSE in the neural network. Furthermore changing the output nodes from two too one. As a final note before starting on the actual method: during this entire analysis I will only be using the relu as the activation function with a batch size of 500. As a final final note, when stating that the error estimates are calculated etc, I mean that both MSE and R2 are calculated with respect to the test section of the data for both the data set and the actual Franke function without noise.

The first topic at hand is to showcase the instability I am experiencing. For this, a relu model for all activation functions was recreated a total of 100 times. Each time the model predicted only zeros was counted and every time the model predicted something else was counted. Note however that the model itself was stable i.e not overflow, but rather it only predicted zeros. The instability is important to be aware of such that countermeasures can be taken, to avoid missing a possible good model.

Without any knowledge of a good combination of epochs, learning rate and activation function for the hidden layer, exploring this was the first task at hand. Using a constant number of neurons (50) the error estimates for the R2 and MSE for the test data (both franke and data set) was calculated for every possible combination of $[1, 10, 50, 100, 200, 500]$ epochs with $\eta = [10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}]$ and sigmoid , tanh, elu and relu as the hidden activation functions. The best score was chosen for each activation function.

So far the design matrix X has only been of 5th order polynomial, this will now change. By choosing the best combination of activation function, epoch and learning rate from the previous paragraph, over-fitting with neural network will be studied. This is done on the entire data and k-fold cross validation is used with 4-folds for polynomial complexity from 1st degree to 20th degree. For each excluded fold in k-fold cross validation the error estimates is calculated. This process is repeated $N = 10$ times and the average of the error estimates is calculated. Without much specification, a test was implemented in the case of instability, and for unstable cases (predicting only zeros) the same model was re-created a maximum of n-times until stability. The error estimates was than finally plotted against the polynomial complexity.

The final step, albeit large and complex is to create a good performing model by utilizing multiple layers if necessary. Before starting I will impose a restriction, and that is that the only activation function used in the hidden layers are the elu activation function. Furthermore I will only be using a design matrix of 5th degree complexity, except when stated otherwise.

The first step was to study one hidden layer. Multiple models with the following $[10, 20, 40, 60, 80, 100, 140, 180, 250, 300]$ nodes were created. For each node a model was created a total of 5 times and the best error estimates for each node was kept. The best preforming number of neurons was than chosen to be used as the first layer in a two hidden layered model. The same step as with a one layered model was repeated with the same nodes. This run was also repeated 5 times with the best performing model for each node being kept. The best combination so far was than used as basis for a third hidden layer. The method used in the one and second hidden layer was than repeated for the third layer with again 5 repeated runs and the best performing models for each node being kept. In the cases the model was unstable for all 5 repeated runs, that node for that layer was omitted. Furthermore the addition of a third layer was repeated once for a different number of neurons in the second layer, the method however in the third layer remained unchanged.

Finally the very best model found regardless of number of hidden layers and number of neurons was used to plot a 3D plot of the Franke function based on the model.

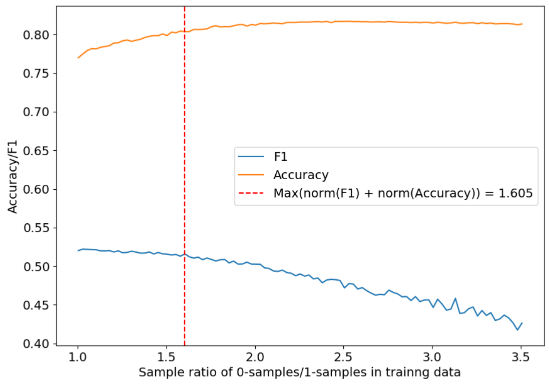
IV. RESULTS

A. Logistic regression

Figure 2 shows the accuracy and F1 scores as a function of the ratio of 0-samples/1-samples in the training data. When down-sampling: the batch size is 200, learning rate 0.001 and 1000 epochs. The plot is the mean of $n = 5$ random runs. The ratio which maximizes the norm of the accuracy and F1 scores is

$$\frac{S_0}{S_1} \approx 1.6 \tag{IV.1}$$

which is the same as 7600 excluded 0-samples. This is the value for down-sampling which will be used in the rest of the logistic regression analysis.



Figur 2: The accuracy and F1 scores of logistic regression model as a function of the ratio $\frac{S_0}{S_1}$ i.e the ratio between 0-samples and 1-samples in the training data. The batch size is 200, a learning rate 0.001 and 1000 epochs.

The test accuracy of the logistic regression models for different leaning rates and number of iterations with constant batch size of 200 is shown in table I. All these models are trained using the down-sampled and normalized training set. In table II the F1 scores for different learning rates and epochs are given. In all of these cases the data is down-sampled to the ratio in IV.1.

Tabell I: The accuracy scores of the logistic regression model for different learning rates and epoch, with a constant batch size of 200 with the down-sampling ratio given in IV.1.

	0.1	0.01	0.001	0.0001	1e-05	1e-06
10	0.789	0.798	0.808	0.806	0.804	0.804
100	0.779	0.811	0.806	0.808	0.806	0.804
1000	0.779	0.811	0.809	0.803	0.807	0.806
10000	0.778	0.808	0.802	0.803	0.804	0.806
100000	0.78	0.788	0.803	0.804	0.805	0.804

Tabell II: The F1 scores of the logistic regression model for different learning rates and epoch, with a constant batch size of 200 with the down-sampling ratio given in IV.1.

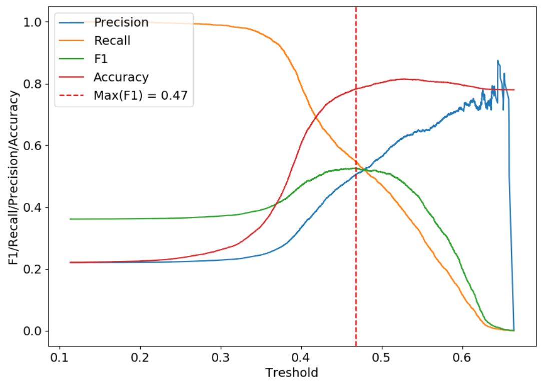
	0.1	0.01	0.001	0.0001	1e-05	1e-06
10	0.506	0.514	0.496	0.422	0.404	0.404
100	0.0307	0.505	0.512	0.501	0.419	0.404
1000	0.00747	0.474	0.506	0.515	0.505	0.416
10000	0.0852	0.506	0.512	0.512	0.514	0.504
100000	0.0337	0.51	0.509	0.513	0.512	0.513

From table I and II a good model seem to be the 1000 epochs with learning rate 0.0001[3], with accuracy of 0.803 and F1 score of 0.515. The precision and recall score of this model is

$$\text{Recall} = 0.471 \tag{IV.2}$$
$$\text{Precision} = 0.563 \tag{IV.3}$$

with a threshold of 0.5 for rounding up to the binary 1. In figure 3 the F1, precision, recall and accuracy is plotted for different thresholds for binary rounding. The maximum for F1 is found at 0.47 being $F1 = 0.527$ with an accuracy of 0.783.

[3] Many of the other models are equally valid choices, but with 1000 epochs the calculation speed won't suffer too much.



Figur 3: The accuracy, F1, precision, recall as a function of the threshold used in determining whether a probability corresponds to a 0 or 1 sample. This is for a 0.0001 learning rate, 1000 epochs, 200 batch size model with a down-sampling ratio of $\frac{S_0}{S_1} \approx 1.6$.

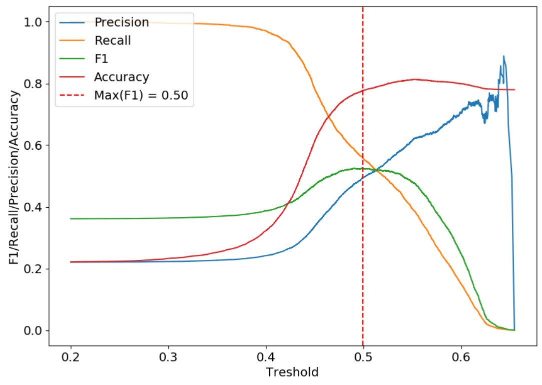
For a down sampling ratio of $\frac{S_0}{S_1} = 1$, the accuracy and F1 scores for different learning rates and epochs are shown in table III and IV respectively. The maximum F1 score being 0.525. For a 1000 epoch, 0.0001 learning rate model, the threshold that maximizes the F1 score was found at 0.5 with unchanged F1 as seen in figure 3, and a accuracy of 0.777.

Tabell III: The Accuracy scores of the logistic regression model for different learning rates and epoch, with a constant batch size of 200 with the down-sampling ratio of 1.

	0.1	0.01	0.001	0.0001	1e-05	1e-06
10	0.624	0.776	0.779	0.786	0.777	0.753
100	0.773	0.772	0.778	0.782	0.787	0.77
1000	0.751	0.778	0.776	0.776	0.782	0.786
10000	0.722	0.787	0.776	0.775	0.775	0.78
100000	0.775	0.768	0.771	0.775	0.775	0.775

Tabell IV: The F1 scores of the logistic regression model for different learning rates and epoch, with a constant batch size of 200 with the down-sampling ratio of 1.

	0.1	0.01	0.001	0.0001	1e-05	1e-06
10	0.454	0.516	0.524	0.512	0.505	0.5
100	0.508	0.515	0.524	0.524	0.51	0.503
1000	0.499	0.525	0.525	0.524	0.524	0.512
10000	0.478	0.52	0.523	0.522	0.525	0.523
100000	0.0139	0.525	0.522	0.523	0.524	0.524



Figur 4: The accuracy, F1, precision, recall as a function of the threshold used in determining whether a probability corresponds to a 0 or 1 sample. This is for a 0.0001 learning rate, 1000 epochs, 200 batch size model with a down-sampling ratio of $\frac{S_0}{S_1} = 1$.

B. Neural Network

For 'just' one hidden layer with a tanh hidden activation layer function and softmax activation function the accuracy and F1 scores for a multitude of learning rates and epochs for 64 hidden nodes is shown in table V and VI. Furthermore the accuracy and F1 scores for the same learning rates and epochs with the softmax activation function, but different hidden layer functions is given in table X to XV.

Tabell V: The accuracy for a NN with tanh activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.78	0.816	0.796	0.585	0.416	0.566
10	0.78	0.818	0.816	0.793	0.677	0.693
50	0.78	0.812	0.817	0.807	0.783	0.658
100	0.78	0.811	0.818	0.816	0.797	0.634
500	0.78	0.754	0.816	0.819	0.812	0.787

Tabell VI: The F1 for a NN with tanh activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.0	0.464	0.35	0.234	0.368	0.174
10	0.0	0.495	0.462	0.34	0.285	0.255
50	0.0	0.478	0.471	0.426	0.206	0.326
100	0.0	0.485	0.475	0.466	0.386	0.215
500	0.0	0.455	0.482	0.474	0.439	0.266

From table V and VI the combination of 100 epochs and a learning rate of 0.0001 gave the following scores

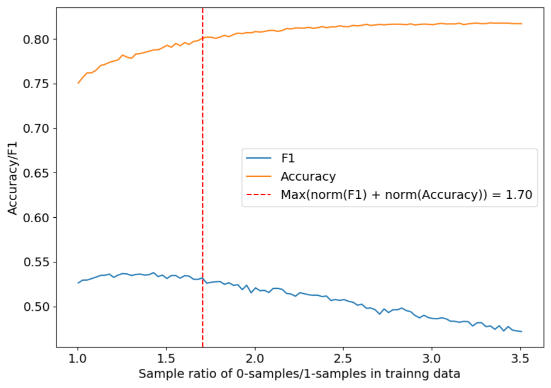
$$\text{Accuracy} = 0.818 \tag{IV.4}$$
$$\text{F1} = 0.475. \tag{IV.5}$$

This combination of epochs, learning rate and hidden activation function is used in figure 5 where the accuracy and F1 scores are plotted against the ratio of 0-samples over 1-samples in the training data. The ratio where the sum of the norms of the accuracy and F1 scores are at its highest is at

$$\frac{S_0}{S_1} \approx 1.70 \tag{IV.6}$$

which is about the same as 7200 excluded samples from the training set. The performance scores for this model with down-sampling is

$$\text{Accuracy} = 0.801 \tag{IV.7}$$
$$\text{F1} = 0.531. \tag{IV.8}$$

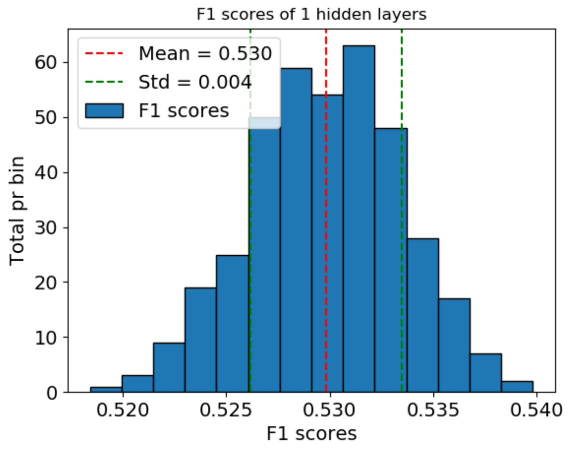


Figur 5: The accuracy and F1 scores for a NN with a hidden layer with 64 nodes and tanh activation as a function of the ratio $\frac{S_0}{S_1}$ i.e the ratio between 0-samples and 1-samples in the training data. The batch size is 200, a learning rate of 0.0001 and 100 epochs.

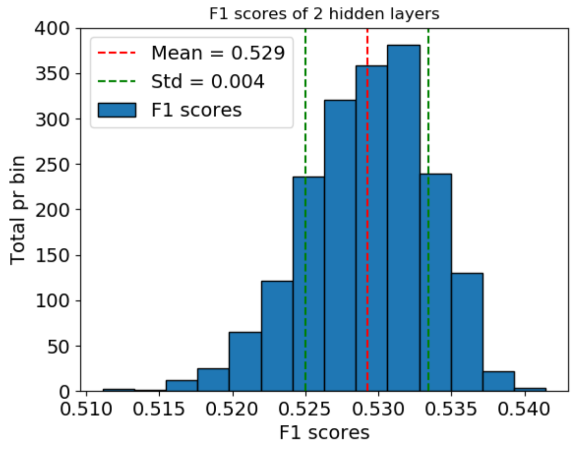
Using the down-sampling from figure 5 with many, many and many combinations for 1 and 2 hidden layers with 100 epochs and a learning rate of 0.0001. The combinations which yielded the best accuracy, F1 and the norm is given in table VII. Furthermore in figure 6 and 7, a histogram distribution of the F1 scores for one and two hidden layers are plotted with their mean and standard deviation.

Tabell VII: The best accuracy, F1 and norm between them for both 1 and 2 hidden layers for the given activation function and number of nodes with 100 epochs, a learning rate of 0.0001 and a batch size of 200. The ratio of the down-sampling used is given in IV.6.

Layers	Nodes	Accuacy	F1
Elu	7	0.808	0.523
Elu	13	0.801	0.540
Elu	13	0.801	0.540
Sigmoid - Elu	54 - 74	0.812	0.517
Tanh - Sigmoid	74 - 104	0.793	0.541
Elu - Sigmoid	24 - 94	0.803	0.538

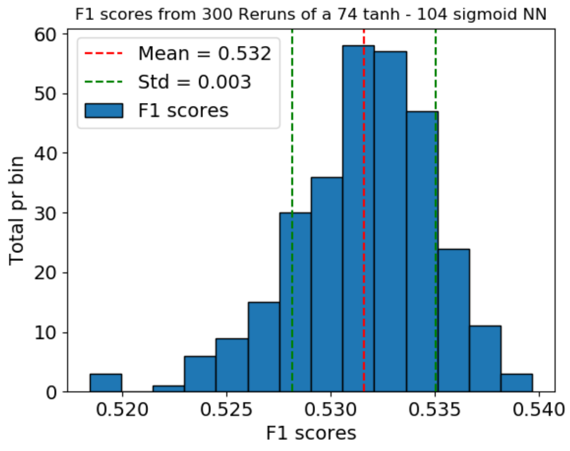


Figur 6: A histogram of all the F1 scores for all the models with different activation functions and nodes with one hidden layer for 100 epochs, batch size of 200 and 0.0001 learning rate. The ratio of the down-sampling used is given in IV.6.



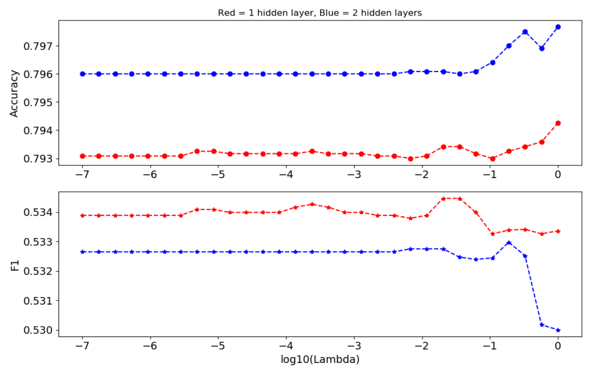
Figur 7: A histogram of all the F1 scores for all the models with different activation functions and nodes with two hidden layers for 100 epochs, batch size of 200 and 0.0001 learning rate. The ratio of the down-sampling used is given in IV.6.

By recreating the model with the highest F1 from table VII, but with random weights 300 times, multiple F1 scores arose. These are plotted in the histogram in figure 8. The best F1 score from this was $F1 = 0.540$ and the mean was $\mu = 0.532$ with a standard deviation of $\sigma = 0.002$.

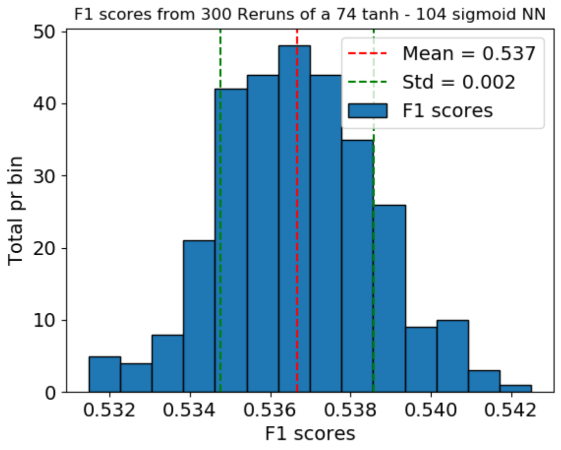


Figur 8: A histogram of the F1 scores for the same model but with different initial random weights with the mean and standard deviation for a 74 tanh - 104 sigmoid model (the same as in table VII).

In figure 9 the two models with the best F1 scores from table VII is plotted against various λ-values in the interval $\lambda \in [10^{-7}, 1]$. In this plot the stochasticity in the initialization of the initial weights are removed by applying a seed before the initialization.



Figur 9: The accuracy of a one hidden and two hidden layer models (the best scoring F1 models from table VII) with different λ-values. The initial weights are the same for all λ-values.



Figur 10: A histogram of the F1 scores for the same model but with different initial random weights with the mean and standard deviation for a 74 tanh - 104 sigmoid model (the same as in table VII). The only difference is that this is for a down-sampling ratio of $\frac{S_0}{S_1} \approx 1.3$.

Using the same method as in figure 4, but with a down-sampling fraction of $\frac{S_0}{S_1} \approx 1.3$ which equal removing 8816 0-samples. The performance scores for the model with the highest F1 score was

$$\text{Accuracy} = 0.789 \tag{IV.9}$$
$$\text{F1} = 0.543. \tag{IV.10}$$

and the corresponding histogram for the F1 scores is shown in figure 10.

C. Regression analysis Neural Network

I will start by illustrating the instability in the creation of the models. For a model with one hidden layer, relu as all activation functions, 50 nodes in the hidden layer, 200 epochs and $\eta = 10^{-4}$ the number of stable models was $N_{stable} = 78$ and $N_{unstable} = 22$ unstable models. Note however that none of these models caused overflow, but rather predicted all zeros.

The start of this result section will resolve around finding the ideal activation functions. Thus with one hidden layer and relu end activation function for 50 neurons in the hidden layer, the R^2 and MSE errors is shown in table VIII. The shown values are the best from a grid of $\eta \in [10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}]$ and epochs $\in [1, 10, 50, 100, 200, 500]$. The best combination for all numbers in table VIII was with 500 epochs with $\eta = 10^{-4}$, except for the tanh function and the MSE/R2 data error where 200 epochs minimized the error. In table IX the error estimates for the elu model with different η-values for 500 epochs is given.

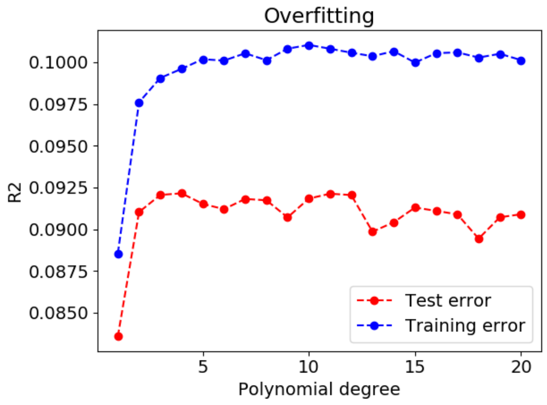
Tabell VIII: The MSE and R2 error estimates for test section of the data for a learning rate of $\eta = 10^{-4}$ and 500 epochs, except for tanh with 200 epochs. The design matrix used is for 5th order polynomial complexity.

	Relu	Elu	Sigmoid	Tanh
MSE Franke	0.00743	0.00734	0.021	0.0117
R2 Franke	0.921	0.922	0.776	0.865
MSE data	0.977	0.978	0.998	0.986
R2 data	0.105	0.103	0.0856	0.0965

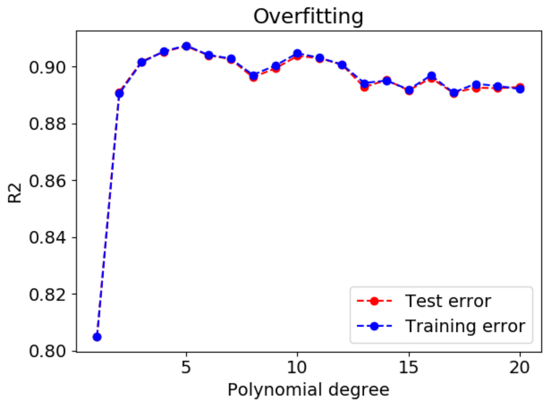
Tabell IX: The MSE and R2 error estimates for test section of the data with different learning rates and 500 epochs for the elu activation function. The design matrix used is for 5th order polynomial complexity.

	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
MSE Franke	0.3	0.0116	0.00734	0.0134	0.0308
R2 Franke	0.0	0.877	0.922	0.858	0.672
MSE data	0.0	0.985	0.98	0.988	0.0
R2 data	0.0	0.0976	0.102	0.0947	0.0771

So far I have only used a 5th order polynomial. In figure 11 the R-squared error for the test and training data is plotted against the polynomial complexity using the best model from table VIII i.e the 50 layer elu model. The shown plot is the averaged of N = 10 random k-fold cross validation runs with 4 folds. Furthermore in figure 12 the same plot is shown, but the error is calculated based on the real franke function.



Figur 11: The R2 test and training error as a function of polynomial complexity for a 500 epochs, 0.0001 learning rate model with 50 neurons and elu in the hidden layer and relu as the activation function in the last layer. The error is with regards to the data set with noise.

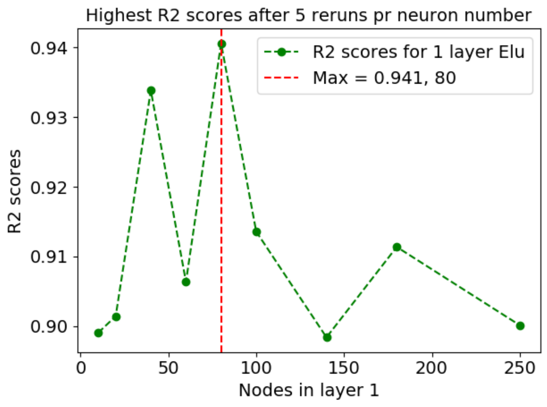


Figur 12: The R2 test and training error as a function of polynomial complexity for a 500 epochs, 0.0001 learning rate model with 50 neurons and elu in the hidden layer and relu as the activation function in the last layer. The error is with regards to the Franke function without noise.

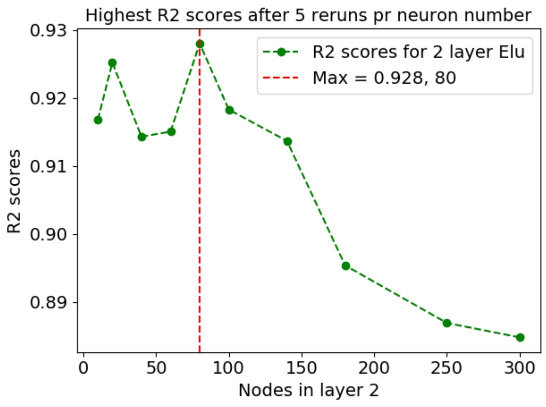
The next step in regression analysis utilizing neural networks concerns the matter of creating a optimal model. In figure 13 the R2 error estimate for test section of the actual Franke function is plotted against the number of neurons in the hidden layer. The activation function for hidden layer is elu, while the end activation function is relu. The best model was found at 80 nodes in the hidden layer with the following error estimates

$$\mathrm{R}^2_{Franke} = 0.941 \tag{IV.11}$$
$$\mathrm{MSE}_{Franke} = 0.00558. \tag{IV.12}$$

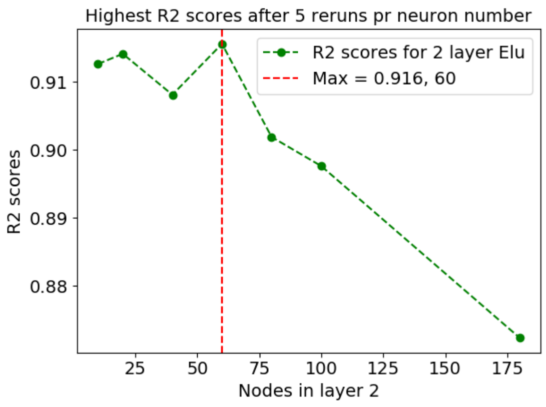
Both of these error estimates are for the test section of the actual Franke function. Furthermore in figure 14 the R2 error estimates for the test section between the model and franke function is plotted against the number of neurons in the added second layer. The second layer consists of the elu activation function added upon the best performing layer from figure 13. This model had the following error estimates $\mathrm{R}^2_{Franke} = 0.928$ and $\mathrm{MSE}_{Franke} = 0.00645$. Lastly figure 15 shows the R2 for the third added layer as a function of the number of neurons in that layer with the following error estimates $\mathrm{R}^2_{Franke} = 0.916$ and $\mathrm{MSE}_{Franke} = 0.00793$.



Figur 13: The best models for each neuron number for N=5 re-randomization of the initial weights. The model uses 500 epochs with a learning rate of 0.0001 and elu hidden activation function, and relu in the last layer.

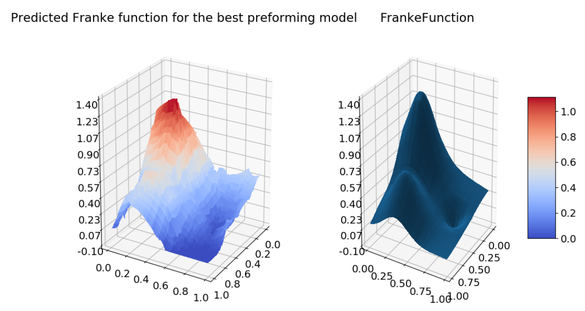


Figur 14: The best models for each neuron number in the second layer for N=5 re-randomization of the initial weights. The model uses 500 epochs with a learning rate of 0.0001 and all elu activation functions except relu in the last layer. The first layer is the best layer from figure 13.



Figur 15: The best models for each neuron number in the second layer for N=5 re-randomization of the initial weights. The model uses 500 epochs with a learning rate of 0.0001 and all elu activation functions except relu in the last layer. The first layer is the best layer from figure 13 and second layer is the best from figure 14.

In figure 16 a 3D plot of the predicted best model from figure 11 beside the actual Franke function.



Figur 16: A 3-dimensional plot of the best model from figure 11 beside the actual Franke function.

V. DISCUSSION

The first topic of discussion will be about bad terminology. In the beginning of subsection IV C I bluntly call the model unstable. This is however not the case. As stated there, none of the so-called unstable models caused overflow. Instead they all predicted zeros. This is a consequence of gradient descent mixed with random initial weights. Gradient descent does not necessarily find the global minimum. Instead, and often, the found minimum will be a local minimum. For around 20-25% cases with the used model, one such local minimum was to predict all zeros. If I were to use a different standard deviation than $\sqrt{2/\text{Number of nodes}}$ this may have been avoided. This also showcase the stochastic behavior in the model creation, but more about that later.

Another point I will discuss is one problem in my down-sampling. When down-sampling, I throw away many data-points never too be seen again. Doing this, I may loose a great deal of potentially critical information regarding the model. One way to avoid this would be to randomize the omitted data-points multiple times and taken the average for my results. However, any methods involving re-running my results multiple times would take too much computational power.

The last point out concerns my implementation of the back propagation algorithm. As mentioned in the section III, I used steepest descent for updating the weights. A better opting may be ADAM since it is capable of adjusting the learning rate automatically. This is of-course just speculation without any results proving my hypothesis.

A. Credit card data

1. *Logistic regression*

Figure 2 showcases an important aspect for biased data, and data analysis in general. It clearly shows improved predictability for a model created with a smaller training set more evenly distributed between the possible sample values (0 or 1 samples). So much that the lose in accuracy is less than the increase in the F1 score. Furthermore, it can clearly be seen that the down-sampling that maximizes the F1 score is at a evenly distributed data set.

Funny enough, the best F1 score was not achieved at a sample ratio of $S_0/S_1 = 1$, but rather when the norm of accuracy and F1 was at its maximum. The very best F1 score ended up being $F1 = 0.527$ and a accuracy of 0.783 with a 0.47 threshold instead of a 0.5 threshold. Whether the change of threshold is a more efficient way of improving the predictability and thus the F1 score than down-sampling is difficult to say, but the results may indicate something along those lines. Another interesting aspect is that the accuracy of the changed threshold model was greater than the accuracy of the 1-ratio down-sampled model. It would have been interesting to change threshold for a non down-sampled model, and see whether the F1 score could improve and compete with the down-sampled versions. Based on the result however it seems a good mixture of down-sampling together with a minor tweak in the threshold would prove the best predictability of the model.

Unlike for neural network, the initial β-values for logistic regression was not randomized. This may have hindered the logistic regression model such that it was 'stuck' in a possible bad local minimum, and could not 'break' free.

2. *Neural Network*

I will begin by directing you gaze to figure 8 and 10. In these figure, all the models created in these histograms have the same parameters. Except all have different initial weights. This makes it perfectly clear just how many different local minima there exists. This makes it difficult to determine whether a model combination is good or just pure luck. Furthermore, it also makes it difficult to determine whether the hyper parameter λ would have any noticeable effect.

In figure 9 the error estimates is plotted for multiple λ-values with the same initial weights. It's clear that the addition of the λ parameter may improve the performance of the model, but not by a large margin. It would require more testing to figure out how much exactly, I did not do this because of the sheer amount of adjustable parameters in a neural network.

For a one layered neural network the combinations of epochs and learning rates differs depending on the activation function as seen for table VI and XI for tanh and sigmoid respectively. The sigmoid function seems to be more dependent on a correct combination of learning rate and epochs. The same analysis can be applied to the other activation functions as-well.

In the subject of down-sampling, a similar trend as seen in logistic regression arises in figure 5. However, unlike logistic regression the down-sampling that maximizes the F1 score is not at a 1:1 ratio, but around $S_0/S_1 \approx 1.3$. This can be seen in figure 8 and 10 where the mean of the F1 score is larger for the $S_0/S_1 \approx 1.3$ down-sampling ratio. The maximum achieved F1 score was for two layers with $F1 = 0.543$ and accuracy of 0.789. It is also notable from table VII that two layers seem to have slightly better performance than with one hidden layer. This may just arise from the fact that there were preformed more test for two layers than one.

B. Comparison

The most important subject is how logistic regression and neural network performed in comparison. From all the results, neural network outperformed logistic regression in both accuracy and F1 scores. The maximum F1 score for logistic regression was 0.527 whereas, the highest F1 score for neural network was 0.543. The corresponding accuracy to the best F1 score models also showed that neural network had a better performance.

A comparison of the highest accuracy is however more difficult. This is because I did not include the accuracy for a non down-samples logistic regression model. They do however seem to be about the same when studying the number in table I, VI and taking figure 2 into account.

It seem neural network outperformed logistic regression in this analysis. As mentioned in the discussion about logistic regression, this may be caused by being 'stuck' in a bad local minima. However, the neural network used in this analysis is not optimal. Both in regard to the implementation and the network itself. If a professional neural network package was used there may have been significant jumps in the performance.

C. Regression Neural network

Much like earlier with the classification example, the randomization of the hidden weights does make it difficult to find a good model. However there are still some clear trends in the results.

From table VIII it's obvious that the best functions for the regression analysis is either the elu or relu activation function. Furthermore it's evident from table IX that a learning rate of 10^{-4} is the best choice.

An important aspect of regression analysis with neural network is the lack of overfitting. Figure 11 barley shows any sign of overfitting and in comparison to OLS from [4] there is no point in worrying. However, it's is clear that the best polynomial lies somewhere around 5th degree complexity, much like in [4] for OLS and ridge.

In comparison to rige and lasso regression, neural network doesn't need the necessary fine tuning in regards to finding the optimal λ penalty parameter. This is an advantage for neural network in comparison to ridge and lasso where the optimal λ-parameter differs depending on the polynomial complexity. This is evident from [4] in the appendix in figure 28. By finding a good combination of epochs and learning rate, the fear of potentially overfitting won't be a problem. However, the need of choosing a good activation function requires some analysis. The same goes for choosing a total number of layers and nodes,

The second to last issue for discussion is the addition of multiple hidden layers. A trend in figure 13, 14 and 15 is that the addition of multiple hidden layers worsen the error estimate. Especially when reaching three hidden layers, where multiple values for some neuron numbers had to be emitted because they predicted zeros or overflow. Although one hidden layer had the best error estimate, it seems that for a small number of nodes in two hidden layers were stable in the error estimate prediction. A clear trend in all three figure is that when the neuron number increases for the last layer before the output the performance of the model goes down.

The best performing model with neural network had $R^2 = 0.942$ as the best error estimate and the 3D plot is given in figure 16. This is better than the best performing model in [4] with the best error estimate being $R^2 = 0.932$ when the data is split between test and training. The best neural network model only fall slightly the best model from [4] when the entire data set is used as training with $R^2 = 0.944$. However, the best created model may outperform event that since the error is only taken with respect to the test section of the data not the entire data set. When taking the error estimate on the entire data set with model created on the training set, it may have even better performance.

Before ending, I would like to quickly discuss the uneven behavior of figure 16. This is because I used the elu activation function in the hidden layer. When testing with relu, this did not happen.

VI. CONCLUSION

Both logistic regression and neural network showed clear evidence that down-sampling of data improved model performance. The highest F1 score for a logistic regression model was $F1 = 0.527$ with an accuracy of 0.783. On the same data set, the highest F1 score of a neural network model was $F1 = 0.543$ with an accuracy of 0.789. In general: the neural network had better performance than the logistic regression model in all aspects, although which method had the highest accuracy is unclear.

The best performing neural network model for regression analysis was a one hidden layer model with elu in the hidden layer and relu as the end activation function. This model had an error estimate of $R^2 = 0.941$ which outperformed most models created using linear regression methods. Furthermore,the usage of neural network showed only a slight tendency for overfitting in comparison to OLS.

Tillegg A: Tables NN

Tabell X: The accuracy for a NN with sigmoid activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.78	0.805	0.78	0.78	0.779	0.755
10	0.78	0.815	0.806	0.787	0.76	0.779
50	0.78	0.819	0.813	0.801	0.781	0.763
100	0.78	0.819	0.817	0.806	0.78	0.775
500	0.78	0.81	0.818	0.814	0.804	0.78

Tabell XI: The F1 for a NN with sigmoid activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.0	0.382	0.0127	0.0193	0.00301	0.0367
10	0.0	0.507	0.385	0.0909	0.0217	0.0
50	0.0	0.474	0.433	0.31	0.0128	0.018
100	0.0	0.456	0.457	0.371	0.0	0.0124
500	0.0	0.442	0.474	0.436	0.362	0.0

Tabell XII: The accuracy for a NN with relu activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.78	0.811	0.801	0.728	0.653	0.549
10	0.78	0.817	0.815	0.78	0.754	0.454
50	0.78	0.812	0.818	0.78	0.792	0.473
100	0.78	0.78	0.817	0.78	0.799	0.78
500	0.78	0.78	0.78	0.78	0.78	0.793

Tabell XIII: The F1 for a NN with relu activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.0	0.424	0.359	0.145	0.166	0.276
10	0.0	0.471	0.443	0.0	0.118	0.208
50	0.0	0.49	0.476	0.0	0.205	0.359
100	0.0	0.0	0.471	0.0	0.312	0.0
500	0.0	0.0	0.0	0.0	0.0	0.342

Tabell XIV: The accuracy for a NN with elu activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.78	0.78	0.78	0.756	0.78	0.55
10	0.78	0.816	0.813	0.78	0.78	0.588
50	0.78	0.78	0.78	0.813	0.779	0.78
100	0.78	0.78	0.816	0.813	0.799	0.78
500	0.78	0.78	0.78	0.816	0.803	0.78

Tabell XV: The F1 for a NN with elu activation function in the hidden layer with 64 nodes and softmax for the final activation function. The batch size used is 200.

	0.01	0.001	0.0001	1e-05	1e-06	1e-07
1	0.0	0.0	0.0	0.0832	0.0	0.337
10	0.0	0.438	0.436	0.0	0.0	0.293
50	0.0	0.0	0.0	0.436	0.313	0.0
100	0.0	0.0	0.472	0.454	0.305	0.0
500	0.0	0.0	0.0	0.467	0.381	0.0

Tillegg B: Activation and cost functions

1. Cost function

There are a total of two cost function that will be used in this project: cross entropy and mean square error(MSE). The cross entropy cost function takes the following form for a binary classification problem

$$C(\boldsymbol{p}_i) = -\left(\boldsymbol{y}_i \cdot \ln(\boldsymbol{p}_i) + (1-\boldsymbol{y}_i)\cdot \ln(1-\boldsymbol{p}_i)\right). \tag{B.1}$$

Notice how I did not include either a sum nor divided on the total number. This is because the matrix notation used for the logistic regression and neural network handles multiple samples without any modifications by innate summation. Furthermore the division by the number of samples N is unnecessary because of the multiplication with the learning rate. The derivative of the cross entropy is given by

$$\frac{\partial C(\boldsymbol{p}_i)}{\partial \boldsymbol{p}_i} = -\left(\boldsymbol{y}_i \ominus \boldsymbol{p}_i + (1-\boldsymbol{y}_i \ominus (1-\boldsymbol{p}_i))\right) \tag{B.2}$$

and $\boldsymbol{y}_i$ is the correct value e.g $[1,0]$ whereas $\boldsymbol{p}_i$ is the probability for predicting either. Furthermore, the sign $\ominus$ is meant as hadamard division.

The next cost function is the MSE given by

$$C(\boldsymbol{p}_i) = (\boldsymbol{y}_i - \boldsymbol{p}_i)^2, \tag{B.3}$$

and again notice the lack of summation and division by sample number N. The derivative of the MSE is

$$\frac{\partial C(\boldsymbol{p}_i)}{\partial \boldsymbol{p}_i} = -2\,(\boldsymbol{y}_i - \boldsymbol{p}_i) \tag{B.4}$$

where the number 2 can be omitted because of the learning rate multiplication.

2. Activation functions

There are a total of five activation function utlized within project: softmax, sigmoid (logit), tanh, relu and elu.

The softmax activation function is defined as followed:

$$f_{\text{soft}}(\boldsymbol{p}_i) = \frac{\mathrm{e}^{\boldsymbol{p}_i}}{\sum_{i=1}^{n} \mathrm{e}^{\boldsymbol{p}_i}} \tag{B.5}$$

with the derivative being

$$\frac{\partial f_{\text{soft}}(\boldsymbol{p}_i)}{\partial \boldsymbol{p}_i} = f_{\text{soft}}(\boldsymbol{p}_i) \circ (1 - f_{\text{soft}}(\boldsymbol{p}_i)) \tag{B.6}$$

where $\circ$ is hadamard multiplication.

The sigmoid activation function is defined as followed:

$$f_{\text{sig}}(\boldsymbol{p}_i) = \frac{1}{1-\mathrm{e}^{\boldsymbol{p}_i}} \tag{B.7}$$

with the derivative being

$$\frac{\partial f_{\text{sig}}(\boldsymbol{p}_i)}{\partial \boldsymbol{p}_i} = f_{\text{sig}}(\boldsymbol{p}_i) \circ (1 - f_{\text{sig}}(\boldsymbol{p}_i)) \tag{B.8}$$

where $\circ$ is hadamard multiplication.

The relu activation function is defined as followed:

$$f_{\text{relu}}(\boldsymbol{p}_i) = \begin{cases} \boldsymbol{p}_i & \boldsymbol{p}_i > 0 \\ 0 & \boldsymbol{p}_i \leq 0 \end{cases} \tag{B.9}$$

with the derivative begin

$$\frac{\partial f_{\text{relu}}(\boldsymbol{p}_i)}{\partial \boldsymbol{p}_i} = \begin{cases} 1 & \boldsymbol{p}_i > 0 \\ 0 & \boldsymbol{p}_i \leq 0 \end{cases}. \tag{B.10}$$

The elu activation function is defined as followed:

$$f_{\text{elu}}(\boldsymbol{p}_i) = \begin{cases} \boldsymbol{p}_i & \boldsymbol{p}_i > 0 \\ \alpha \mathrm{e}^{\boldsymbol{p}_i} & \boldsymbol{p}_i \leq 0 \end{cases} \tag{B.11}$$

with the derivative begin

$$\frac{\partial f_{\text{elu}}(\boldsymbol{p}_i)}{\partial \boldsymbol{p}_i} = \begin{cases} 1 & \boldsymbol{p}_i > 0 \\ \alpha \mathrm{e}^{\boldsymbol{p}_i} & \boldsymbol{p}_i \leq 0 \end{cases} \tag{B.12}$$

where $\alpha = 0.1$ and just a parameter.

[1] Morten Hjorth-Jensen. Lectures notes in fys-stk4155. data analysis and machine learning: Linear regression and more advanced regression analysis. `https://compphysics.github.io/MachineLearning/doc/pub/Splines/html/Splines.html`, 2019. [Online; accessed 06-November-2019].

[2] Morten Hjorth-Jensen. Lectures notes in fys-stk4155. data analysis and machine learning: Linear regression and more advanced regression analysis. `https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html`, 2019. [Online; accessed 06-November-2019].

[3] Morten Hjorth-Jensen. Lectures notes in fys-stk4155. data analysis and machine learning: Linear regression and more advanced regression analysis. `https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html`, 2019. [Online; accessed 06-November-2019].

[4] Jon Andre Ottesen. Fys-stk4155 – applied data analysis and machine learning project 1 - regression analysis and resampling methods. `https://github.com/JonOttesen/FYS-STK-Projects/tree/master/Project%201`, 2019. [Online; accessed 06-November-2019].

[5] Wikipedia. F1 score — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=F1%20score&oldid=924681114`, 2019. [Online; accessed 07-November-2019].

[6] 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):2473–2480, 2009.