FYS-STK4155
Applied data analysis and machine learning
Project 2

Martin Krokan Hovden

November 13, 2019

Abstract

In this paper we study machine learning algorithms for classification and regression. Code for neural networks and logistic regression is developed from scratch and the theory behind the algorithms are explained in detail.

For classification we use the credit card data from UCI [6]. The data is preprocessed and cleaned before being fed into the models. We will compare the ability to classify defaults on credit card debt for neural networks and logistic regression. We use the results from I-Cheng Yeh et als. article [5] as a baseline for testing.

For regression we will study the performance of the neural network on the Franke function. The results are compared to variations of the linear regression model: OLS regression, Ridge regression and Lasso regression.

To find the best hyperparameters for the models we do a grid search. For the classification problem we find that a neural network with two hidden layers with sigmoid activation functions and 20 and 10 nodes in each layer performs better than logistic regression. By choosing $\alpha = 0.001$ and $\lambda = 0.1$ we get the best area ratio $= 0.5198$. For the regression problem the lowest MSE was obtained using a neural network with three hidden layers with sigmoid activation functions and 20, 10 and 5 neurons in each layer. By setting $\alpha = 0.01$ and $\lambda = 1e-5$ we obtain the best MSE $= 0.0045$. The self made models worked well, however, the computational time were much higher than for scikit learn's implementation.

1 INTRODUCTION

Machine learning algorithms have become increasingly popular over the last years. The increase in computing power have made it possible to train larger and more complex models and the huge amount of data enables us to make good use of the methods. Their application span from self driving vehicles to classifying cancer cells and the algorithms are used in science and research, as well as in businesses in all kind of sectors.

In this project we will study neural networks for both classification and regression. We will develop our own code from scratch and compare it to the professional implementation in scikit-learn. To get a better understanding of how everything works, we will study the theory behind each algorithm in depth. For the classification part we will use credit card data from an Taiwanese bank. The data needs cleaning and preprocessing, and methods for doing this is presented and implemented. The goal is to predict whether a credit card customer defaults or not given some input. The performance of the neural network and logistic regression is compared for different variations of hyperparameters and the resutls from the article [5] is used as benchmarks to test our code.

We will also test the neural network on a regression problem. The function we want to model is the Franke function. The results using OLS, ridge regression and Lasso regression from project 1 will be used as benchmarks to see if we can improve the predictions using variations of a neural network.

For both the classification and regression problem, we will use grid-searches for hyperparameter optimization.

The report is divided in to three main parts. First we will look at the theory behind the algorithms being implemented. This part will dive deep into the machinery of the algorithms and we will try to get an understanding of how the algorithms actually works. One of the main problems with neural network is that they are often seen as a black box that takes some input and returns some output. By studying how the algorithm actually learn we will be able to understand how it comes up with the output. After that, I will present how the code is structured in the GitHub repository. I found it better to leave the code examples out of the report and refer to the python modules at the GitHub repository instead. I have tried to write extensive doc-strings and comments in the code, so I hope it is quite easy to follow. The python codes of the algorithms are also implemented the same way as they are described in the theory part. Lastly, I will present the results from running the algorithms on the data sets mentioned and compare the different methods.

2 THEORY AND METHODS

The theory and mathods part are based on the lecture notes in FYS-STK4155 written by Morten Hjort-Jensen. For more detailed derviatation of the results and further information, see `https://compphysics.github.io/MachineLearning/doc/web/course.html`. For classification, we will use logistic regression and neural network. The focus will be on binary classification. Neural networks will also be studied for regression problems. The regression results will be compared to linear regression models. For a discussion on the linear regression models, see for example Hastie et. al. [4].

2.1 MEASURES OF PERFORMANCE

Choosing the right model is an important problem when applying machine learning to classification and regression problems. No algorithm is the best for every problem, and there is in general no specific rule for choosing the right one. For each model we can typically also fine-tune parameters, so the space to choose from becomes very large. It is therefore important to have ways of comparing the different models. Finding the best model comes down to testing different models on the data, and comparing their performance to each other. However, one have to restrict the search space to a manageable size. We then have to introduce metrics to determine which one is the best for the specific case.

2.1.1 Measures of performance for classification

Classification often deals with labeling samples into a set of different categories. The goal is to classify as many samples as possible to the actual category of the sample. There are different methods for measuring the classifiers ability to classify points. Some of them are presented in this section. All of the methods assumes that we have lableled test data.

An often used performance measure for classification problems is the accuracy score. The accuracy score is the proportion of correctly classified data points. It is given by

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

where y_i is the predicted value of sample i, t_i is the target value of sample i and n is the number of samples. $I(y_i = t_i)$ is the indicator function,

$$I(y_i = t_i) = \begin{cases} 1 & , \ y_i = t_i \\ 0 & , \ y_i \neq t_i \end{cases} \tag{2}$$

The accuracy is easy to interpret and implement, and works fine if the data is unbiased. However, if the data is biased, there is a problem with using the accuracy. Let's say that the targets consist of 90% one's. A model that only predicts ones will then get an accuracy of 90%. Choosing a model based on the accuracy can therefore lead to choosing a model without the ability to classify data.

An better alternative is to check how many points the model correctly classifies and how many it misclassifies. For a binary classification problem, we can start by introducing some notation [2].

- TP: True positive. The number of 1's classified as 1's.
- FP: False positive. The number of 0's classified as 1's.
- TN: True negative. The number of 0's classified as 0's.
- FN: False negative. The number of 1's classified as 0's.
- TPR: True positive rate. $TPR = TP/P$. Also called sensitivity.
- FPR: False postive rate. $FPR = 1 - TN/N$. TN/N is called the specificity.

A confusion matrix is a way to summarize the measures in a compact way. It gives a quick overview of how well the model performs. In addition, other measures can be calculated by using the values in the matrix. Each column represents the true label of the data, and each row represents the predicted label. A general set-up for a confusion matrix is illustrated in table 1. The ideal would be a case where FP and FN are equal to zero. We want those as small as possible.

	1	0
1	TP	FP
0	FN	TN

Table 1: Illustration of a confusion matrix for binary classification. Columns are the true values, while rows are the predicted values.

Using the previously calculated values, we can also find the roc-curve. It plots the true positive rate (TPR) against the false positive rate (FPR) for different threshold values [2]. An optimal classifier would have the curve hugging the upper left corner. The models can be compared by calculating the area under the roc-curve, often called the AUC-score. The higher the AUC-score, the better the model.

Another measure used is the area ratio. To calculate the area ratio we need to find the cumulative gains curve. The cumulative gains curve plots the percentage of classifications of a given class as a function of number of total samples. An optimal model would have a linear curve from (0,0) to (1, percentage of class in total data). The baseline curve is a straight line from (0,0) to (1,1). When the three curves are calculated, the area ratio is given by

$$\text{Area ratio} = \frac{\text{area between baseline curve and the models curve}}{\text{area between the baseline curve and the optimal curve}} \tag{3}$$

The optimal area ratio is one. In that case the cumulative gains curve of the model is the same as the one for the optimal classifier. For more info about the cumulative gains curve see.[1]

[1] https://www.ibm.com/support/knowledgecenter/de/SSLVMB_24.0.0/$spss/tutorials/mlp_bankloan_outputtype_02.html$

2.1.2 Measures of performance for regression

For regression problems we will use the mean squared error (MSE) for comparison. The MSE is given by

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - t_i)^2. \tag{4}$$

where y_i is the predicted value of sample i, t_i is the target value of sample i, and n is the number of samples. The MSE measures the average of the squared deviations. We want the MSE as low as possible. In this case most of our predictions are close to the target values.

2.2 OPTIMIZATION

When fitting machine learning models the goal is to minimize a cost function with respect to the models parameters. The cost function says something about how wrong our models output is compared to the desired outputs. Cost functions will be discussed in more detail later. Some machine learning models have an analytical expression for finding the optimal values. In general, this is not the case. Finding the optimal parameters then requires numerical optimization. The search for the optimal parameters are typically the bottleneck of machine learning algorithms [2]. Finding effective methods for obtaining minimas is therefore of great interest. With the increase in computational power over the last years, training larger and larger models have become possible.

2.2.1 Gradient descent

Gradient descent is an iterative method for finding the minimum of a function. It is often used in machine learning to find the optimal weights and biases of the model. The idea behind the gradient descent method is to calculate the gradient of a function for each iterative step, and then move a given distance in the opposite direction [3]. The gradient is the direction where the function increases fastest. Moving in the opposite direction will then lead to the largest decrease in function value. By continuously moving in the opposite direction of the gradient, the idea is to end up in the minimum of the function. This can be summarized in the equation

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha_k \nabla \boldsymbol{f}(\boldsymbol{x}^k) \tag{5}$$

where $\boldsymbol{x^k}$ is the parameters we want to change, $\alpha_k > 0$ is the step-length, $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ and $\nabla \boldsymbol{f}(\boldsymbol{x^k})$ is the gradient of the function. The approximation of the minimum is in general sensitive to the choice of initial condition $\boldsymbol{x}^0$. If the initial choice is not very good, the algorithm can converge to local minimums and get stuck there [3]. When running the algorithm the iterations typically stop at some predefined number of iterations or when the norm of the gradient is smaller than some tolerance, ϵ,

$$||\nabla \boldsymbol{f}(\boldsymbol{x}^k)|| \leq \epsilon \tag{6}$$

We will use a predetermined number of iterations. This is to analyse the effects of number of iterations.

In machine learning, the step length is often called the learning rate. It determines how fast the model learn. However, the step-length have to be chosen carefully, and there are different ways of doing this. One method is to use a pre-specified value. Choosing it to low will lead to very slow convergence. Choosing it to large can lead to the approximation bouncing around the actual solution or diverging away from the solution. By using a to large value, we are no longer sure that the next parameteres will lead to a smaller function value. By using a pre-specified value, we therefore have to test for different values and see which one works best for our problem. Alternatively, the step length can be chosen by minimizing the function in the direction of the gradient. The last method is called a line search. In this project we will use a predetermined learning rate and test for different values to see how it effects the performance of the model.

The general gradient descent method can get troubles for ill-behaved functions with many local minimums and "rough terrain". Since the method is deterministic, the method can get stuck in local minimums and in general we will have no idea whether the point is a global or a local minimum. The initial choice of parameters is therefore important. However, if the function is convex and differentiable, every minimum is a global minimum and $\nabla \boldsymbol{f}(\boldsymbol{x}) = 0$.[3]

In machine learning, the function to minimize is the cost function. If the parameters is denoted by $\boldsymbol{\theta}$ and the cost function by $\boldsymbol{C}$, we get that

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_\theta \boldsymbol{C}(\boldsymbol{\theta}^k). \tag{7}$$

where $\nabla_\theta \boldsymbol{C}(\boldsymbol{\theta}^k)$ is the gradient with respect to the parameters. The cost function is the sum of the loss function for each sample

$$C(\boldsymbol{\theta}) = \sum_{i}^{n} \boldsymbol{c}_i(\boldsymbol{x}_i, \theta) \tag{8}$$

This can make the gradient descent step very expensive to compute for large data sets with many iterations. One way to fix this is by only using a small batch of the samples for each calculation of the gradient. Theese batches are called "mini-batches".

2.2.2 Stochastic gradient descent (SGD)

The idea behind stochastic gradient descent is to only use a subset of the data for calculating the gradient at each iterative step. By introducing stochastisity we can try to improve on some of the shortcomings of the general gradient descent method. By using only a fraction of the data points, we will hopefully be able to escape local minimums. In addition, the computations of the gradient will be faster due to the smaller number of data points.

If we have n data samples, we can split the data into K batches. The i'th batch is denoted by B_i and consist of n/K data points. We can then find the gradient for this batch by

$$\nabla_{B_k} \boldsymbol{C}(\boldsymbol{\theta}) = \sum_{i \in B_k} \nabla c_i(x_i, \boldsymbol{\theta}) \tag{9}$$

where c_i is the loss function for the i'th sample. The next step can then be calculated as

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{B_k} \boldsymbol{C}(\theta^k). \tag{10}$$

For each step we choose the k randomly. We have two extra parameters that have to be chosen when moving from gradient descent to stochastic gradient descent. The batch size and the number of epochs. One epoch is finished when we have iterated K steps, so that all the data have had the opportunity to help with upgrading the parameters. We typically run many epochs and the number of epochs needed varies from problem to problem.

2.3 LOGISTIC REGRESSION

Logistic regression is a method for finding the probability of a data point belonging to a certain class. Given a set of data points (y_i, x_i), we want to classify the input, x_i, into at set of K classes, $y_i \in (1, 2, ..., K-1)$. In this report we will look at binary classification problems. This is problems with only two classes, so $y_i \in (0, 1)$. The probabilities of each class is given by

$$p(y = 1|\boldsymbol{x}, \boldsymbol{\beta}) = \frac{1}{1 + \exp(-\boldsymbol{\beta}^T \boldsymbol{x})}, \tag{11}$$

$$p(y = 0|\boldsymbol{x}, \boldsymbol{\beta}) = 1 - p(y = 1|\boldsymbol{x}, \boldsymbol{\beta}) \tag{12}$$

where $\boldsymbol{\beta} = [\beta_0, \beta_1, ..., \beta_p]$ is the parameter vector, and $\boldsymbol{x} = [1, x_1, x_2, ..., x_p]$.[2] The 1 is added as an intercept. Logistic regression is called a soft-classifier. In contrast to a hard classifier that returns the predicted class, a soft classifier returns the probability of belonging to each class. When the probability of the class is calculated, we can classify the points given a threshold. Samples with probability higher than the threshold is classified as 1, and samples with probability lower than the threshold is classified as 0. A typical threshold value is 0.5. The class is then given by

$$y = \begin{cases} 1 & \text{if } p(y = 1|\boldsymbol{x}, \boldsymbol{\beta}) \geq 0.5 \\ 0 & \text{if } p(y = 1|\boldsymbol{x}, \boldsymbol{\beta}) < 0.5 \end{cases} \tag{13}$$

Threshold values can take on values in the interval [0,1]. Different problems require different threshold values. For example, for some problems there is important to not have false negatives. This can be for medical purposes, where we don't want to classify a sick person as a healthy person. Lowering the threshold can then lower the chance of this. However, choosing it to low might lead to classifying a lot of healthy people as sick. There have to be a balance between the two.

The problem is finding the optimal parameter vector $\boldsymbol{\beta}$. A often used method maximum likelihood estimation. The maximum likelihood is given by

$$P(\mathcal{D}|\beta) = \prod_{i=1}^{n} [p(y_i = 1|x_i, \beta)]^{y_i} [1 - p(y_i = 1|x_i, \beta)]^{1-y_i}. \tag{14}$$

[4] We want to maximize this function. By taking the log, we get

$$\sum_{i}^{n} y_i \left(\log[p(y_i = 1|x_i, \beta) + (1 - y_i) \log[1 - p(y_i = 1|x_i, \beta)] \right) \tag{15}$$

Taking the log does not change the maximum of the function. Finding the parameters that maximizes this is the same as finding the parameters that minimize the negative. The methods from the optimization chapter can be used. The cost function can then be expressed as

$$C(\beta) = -\sum_{i}^{n} \left(y_i \log[p(y_i = 1|x_i, \beta)] + (1 - y_i) \log[1 - p(y_i = 1|x_i, \beta)] \right) \tag{16}$$

which can be simplified by noting that

$$p(y = 1|\boldsymbol{x}, \boldsymbol{\beta}) = \frac{\exp(\boldsymbol{\beta}^T \boldsymbol{x})}{1 + \exp(\boldsymbol{\beta}^T \boldsymbol{x})} \tag{17}$$

Inserting this into equation 16 and rearranging, we get

$$C(\beta) = -\sum_{i}^{n} y_i(\beta^T x) - \log[1 + \exp(\beta^T x)]) \tag{18}$$

This is called the cross-entropy and will also be used for the neural network with some slight modifications. The cost function does not have an analytic expression for the minimum. It is therefore necessary to use the numerical optimization methods discussed. However, the cross-entropy is a convex function for the logistic regression problem.[4] This assures us that if we find a minima, this is in fact a global minima.

The next step is to find the derivatives of the cross entropy. They are simply given by

$$\frac{\partial C(\beta)}{\partial \beta_i} = -\sum (x_i y_i - x_i p_i) \tag{19}$$

where $p_i = p(y = 1|x_i, \beta)$. The expression can be simplified by introducing some new notation. We often collect all of the samples in a matrix X, where

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & ... & x_{1p} \\ 1 & x_{21} & x_{22} & ... & x_{2p} \\ ... & ... & ... & ... & ... \\ 1 & x_{n1} & x_{n2} & ... & x_{np} \end{bmatrix} \in \mathbf{R}^{n \times p+1}. \tag{20}$$

n is the number of samples and p is the number of features. Here x_{ij} is the j'th feature of the i'th sample. The gradient of the cost function can then be written in matrix notation

$$\nabla \boldsymbol{C}(\boldsymbol{\beta}) = -X^T(\boldsymbol{y} - \boldsymbol{p}) \tag{21}$$

where

$$p = \frac{1}{1 + \exp(-X\boldsymbol{\beta})}. \tag{22}$$

Using gradient descent, the minima can be found by iteratively updating the approximations of the optimal parameters according to

$$\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^k + \alpha X^T(\boldsymbol{y} - \boldsymbol{p}). \tag{23}$$

for k = 0,1,2,...,K.

2.4 NEURAL NETWORKS

A neural network is used for both regression problems and classification problems. The network consists of layers and nodes. Nodes are connected to nodes in the neighbouring layers, and forms a network of nodes. Each connection between each layer is associated with a weight. This lets information flow through the network. When information hits a node, we say that the node is activated.We will study a dense feed forward neural network. In a feed forward neural network, data can only flow in one direction. In a dense neural network, every node in one layer is connected to every node in the next layer. A neural network with 3 input nodes, one hidden layer with 6 nodes and one output layer with 1 node is illustrated in figure 1.

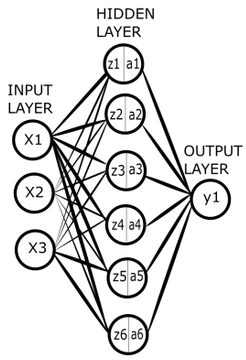


Figure 1: Illustration of a neural network with one hidden layer. This is typically used for regression or binary classification since we only have one output node.Weighted input z comes in to each layer, and then an activation function is applied and gives a.

To derive mathematical expression for the neural network, we will need to introduce some notation. The input will be on the form $X = (n_{\text{inputs}}, n_{\text{features}})$. X is an matrix that contains the features of each sample. $Y = (n_{\text{inputs}})$, is an vector with the target values for each sample. The activated values of a layer is described by the matrix a^l, where l = 1,2,..,L. When input is on the form as described above, a^l is a matrix with dimension $(n_{\text{nodes in layer l}}, n_{\text{nodes in layer l-1}})$. The weight between node j in layer (l-1) and node i in layer l, is denoted by w_{ij}. The weights values going into layer l is collected in the matrix

$$\boldsymbol{w}^l = \begin{pmatrix} w_{11} & w_{12} & ... & w_{1k} \\ ... & ... & ... & ... \\ w_{l1} & ... & ... & w_{lk} \end{pmatrix} \tag{24}$$

Each layer, except the input layer, can have a bias vector $\boldsymbol{b}^l = [b_1^l, .., b_m^l]$.

Flow of information through the network follows a feed forward structure. The input layer consists of the features of a sample. By using the weight matrices, we can efficiently compute the activated values for each layer simultaneously for each sample. The activated values are given by applying the activation function to the weighted sum of the activated values in the previous layer. By using the values in the previous layer and the formula

$$\boldsymbol{a}^l = f(\boldsymbol{a}^{l-1}\boldsymbol{W}^l + \boldsymbol{b}^l) \tag{25}$$

we can calculate the activated values of the current layer. We often denote

$$\boldsymbol{z}^l = \boldsymbol{a}^{l-1}\boldsymbol{W}^l + \boldsymbol{b}^l \tag{26}$$

and it is called the weighted input to layer l.

For the first hidden layer we have that $\boldsymbol{a}^{l-1} = X$. By applying equation 25 recursively from the first hidden layer to the output layer, we can calculate the output by $\boldsymbol{a}^L = f(\boldsymbol{a}^{L-1}\boldsymbol{W}^L + \beta^L)$.

The goal is to adjust the weights and biases so that our model predicts the same values as the target values. This is done via an efficient method called backpropagation. This will be discussed in detail later.

2.4.1 Layers

When setting up a neural network, we are left with numerous choices on the architecture of the network. There are endless combinations of number of neurons, number of layers and activation functions. The optimal combination varies from problem to problem and often it comes down to trial and error.

When choosing activation function for the layers, it is important to use at least one non-linear activation functions for one of the layers. This way we can model non-linear data. In general, there is no specific rule for choosing the right one. However, when switching between regression and classification problems, we have to do certain changes in the output layer. For regression type problems we typically dont need any activation function in the last layer. For classification, we often use an activation function in the last layer that maps the input values into the interval [0,1]. There are some common examples of activation functions that are widely used. One of them is the the sigmoid function

$$f(z) = \frac{1}{1 + \exp(-z)}. \tag{27}$$

The sigmoid is typically used for the output layer in binary classification problems. This is because the function maps values into the interval [0,1]. The sigmoid functions derivatives is defined everywhere. One problem with the sigmoid function is that the gradients tends to vanish for high positive and negative numbers. [2] This becomes a problem when training the model. Another example is the ReLu family of activation functions. The general ReLu function is given by

$$f(z) = \max(z, 0). \tag{28}$$

It works well in most cases and are fast to compute. An variation of the ReLu function is leaky ReLu function

$$f(z) \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases} \tag{29}$$

When choosing the number of nodes in each layer there are some restrictions on the number of nodes in the input and output layer. The input layer have to have the same amount of nodes as features in the input. The output have to have the same number of nodes as targets we want to predict. For the hidden layers, we can choose the number of neurons as we want. Keep in mind that by choosing the number to high might lead to overfitting and very high computational time. One way to avoid overfitting is by using various regularization techniques like early stopping or adding a regularization parameter to the cost function. We will discuss the latter. It is often advised that a layer does not have more nodes than the previous layer [2], but this is not a rule that necessarily have to be followed.

The choice of number of hidden layers also comes down to trial and error. One used method is to start with one layer and then gradually add layers until the networks start to overfit. In this project we will use network with 1 to 6 hidden layers. The universal approximation theorem states that by using a neural network with one hidden layer with a finite number of nodes we can approximate any function we want [2].

2.4.2 Cost function

When training the model, we need to have control over how good our model fit the data. This is done with the cost function. The idea is to measure the deviations between the predicted values and the target values. The cost function is chosen according to the problem one want to solve. We will study a regression problem and a binary classification problem. For the regression problem, we will use the mean squared error. The MSE is given by

$$MSE = \frac{1}{n}\sum_i^n (a_i^L - t_i)^2 \tag{30}$$

where t_i is the target value of the i'th sample. For binary classification problems we typically use the cross entropy introduced for logistic regression. The cross entropy is given by

$$-\sum_{i}^{n}[t_i \log(a_i^L) + (1-t_i)\log(1-a_i^L)] \tag{31}$$

where a_i^L is the output from the network and t_i is the actual repsonse of the samples.

2.4.3 Backpropagation

Backpropagation is a efficient method for computing the gradient of the cost function with respect to weights and biases. By adjusting the weights and biases appropriately, the cost function can be minimized. We initialize the weights and biases with small random numbers. It is important that they are not equal to zero, because the gradient would then be zero, and the algorithm would stop. By finding the gradient of the cost-function, we can use stochastic gradient descent for finding the minimum. The derivation is based on the derivation in Morten Hjort-Jensen's lecture notes linked in the start of this section.

Let's say we have a cost-function given by

$$C = \frac{1}{2}\sum_{i}^{n}[t_i - a^L(x_i)]^2 \tag{32}$$

where n is the total number of training samples, and $a^L(x_i)$ is the output vector for sample i and t_i is the target value for sample that we want our model to produce. The first step is then to compute the output of the network. We need to know how close we are to the actual values. We start with the first hidden layer and recursively calculate the next layers according to

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l \tag{33}$$

This can be rewritten as

$$\boldsymbol{z}^l = (W^l)^T \boldsymbol{a}^{l-1} + \boldsymbol{\beta}^l. \tag{34}$$

in vector notation.
For each layer, the activated values are calculated using the activation function of layer l,

$$\boldsymbol{a}^l = f^l(z^l). \tag{35}$$

For the first hidden layer a^{l-1} is simply the input $\boldsymbol{x}$,

$$\boldsymbol{z}^1 = (W^1)^T \boldsymbol{x} + \boldsymbol{\beta}^1. \tag{36}$$

The next step is to calculate the derivatives of the cost function with respect to the weights and biases of the network. Those are the parameters that we want to change to minimize the cost function. We first note that

$$\frac{dz_j^l}{dw_{ji}^l} = a_i^{l-1} \tag{37}$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l. \tag{38}$$

Finding the expression for the derivative of the cost function gives that

$$\frac{\partial C(W^L)}{\partial w_{jk}^L} = \frac{\partial C(W^L)}{\partial a_j^L}\frac{\partial a_j^L}{\partial w_{jk}^L} = (a_j^L - t_j)\frac{\partial a_j^L}{\partial w_{jk}^L} = (a_j^L - t_j)\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L}. \tag{39}$$

where the two last steps comes from the chain rule and that

$$\frac{\partial C(W^L)}{\partial a_j^L} = (a_j^L - t_j). \tag{40}$$

The expression for $\partial a_j^L / \partial z_j^L$ depends on the activation function, and can be written as

$$\frac{\partial a_j^l}{\partial z_j^l} = f'(z_j^l) \tag{41}$$

for an arbitrary layer. By combining this, we get that

$$\frac{\partial C(W^L)}{\partial w_{jk}^L} = (a_j^L - t_j) f'(z_j^L) a_k^{L-1} \tag{42}$$

where $(a_j^L - t_j)$ is the derivative of the cost function with respect to the activated values in the output layer. This will actually be the same when using the cross entropy as cost function instead of the squared error.

Next, we introduce the error of the output layer as

$$\delta_j^L = f'(z_j^L)\frac{\partial C}{\partial a_j^L} \tag{43}$$

and note that

$$\delta_j^L = \frac{\partial C}{db_j^L} \tag{44}$$

Define

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{45}$$

By using the previous results, the error in a layer can be found by using the error in the next layer (We work our way from the end to the start of the network). We can use the chainrule again, to get the expression

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}. \tag{46}$$

and this can be rewritten as

$$\delta_j^l = \sum \delta_k^{l+1} w_{kj} l + 1 f'(z_j^l) \tag{47}$$

by using the previous results.

Calculating all of the derived expression is quite time consuming. Especially in python where the for-loops needed to calculate all the sums and recursive steps would be time consuming. By instead introducing matrix and vector notation, we are able to speed up computations and make the algorithm more efficient. The backpropagation algorithm can be summarized as follows using matrix and vector notation.

- Initialize all the weights for each layers randomly. They should be chosen from a normal distribution with zero mean.
- Feed the input through the network and compute the activated values for each layer, including the output layer.
- Compute the error in the output layer, $\boldsymbol{\delta}^L = (\boldsymbol{t} - \boldsymbol{y}) \circ f'(\boldsymbol{z}^L)$. Find the gradient of the cost function with respect to W^L and b^L.

  $$\frac{\partial C}{\partial w^L} = (\boldsymbol{a}^{L-1})^T \boldsymbol{\delta}^L$$

  and

  $$\frac{\partial C}{db^L} = \sum_i^{n^L} \boldsymbol{\delta}^L$$

  where the sum is over the samples.
- Propagate the error backward through the network with

  $$\boldsymbol{\delta^l} = \boldsymbol{\delta}^{l+1}(W^{l+1})^T \circ f'(\boldsymbol{z}^l) \tag{48}$$

  for l = l-1, l-2, ..., 2. Find the gradient of the cost function with respect to W^l and b^l.

  $$\frac{\partial C}{\partial w^l} = (\boldsymbol{a}^{l-1})^T \boldsymbol{\delta}^L$$

  and

  $$\frac{\partial C}{db^l} = \sum_i^{n^l} \boldsymbol{\delta}^l$$

  where the sum is over the samples.

The weights and biases are updated with respect to the gradients found in the backpropagation algorithm. Using the stochastic gradient descent method discussed previously, the weights can be updated

$$w^l = w^l - \alpha\frac{\partial C}{\partial w^l} \tag{49}$$

and

$$b^l = b^l - \alpha\frac{\partial C}{db^l} \tag{50}$$

for $l = 2, ..., L$

If the error is propagated to a large stack of layers, the backpropagation might encounter vanishing gradients. This makes it impossible or very slow to train the network any further, and the weights will be stuck.

Details on the implementation of the backpropagation algorithm are found in the class `NN` in the python file `nn.py`. The file also contains classes for creating each layer and a class for setting up the whole neural network.

2.4.4 Regularization

One of the main problems with neural networks is overfitting.[2] Especially for networks with a large number of weights and biases. Overfitting the data means to fit the model to noise instead of the actual signals. By doing this, the model fit the traning data very well, but when tested on unseen data it performs poorly. There is different ways to address the problem of overfitting. Regularization is the method used in this project. The idea is that by using regularization, we can maintain the size of the network without overfitting the data. This is done by restricting the size of the weights by adding an extra penalty term to the cost function. [2] This way, the network can not choose the parameters as large as they would like. By changing the cost function to

$$C(\theta) = \sum_i c_i(x_i, \theta_i) \rightarrow \sum_i c_i(x_i, \theta_i) + \lambda \sum_{ij} w_{ij}^2 \tag{51}$$

we can obtain the desired results. λ is the penalty parameter. The neural network class in `nn.py` contains an option to train the network with or without the regularization parameter. Details are found in the docstring of the method.

2.5 PREPROCESSING OF THE DATA

Before applying the different algorithms it is often useful to do some preprocessing of the data. The preprocessing that is used in this report is scaling of the data, and one-hot encoding of the categorical features. In the scaling we simply subtract the mean of the features and divide by the standard deviation. See report 1. It is important to scale the training data after splitting into training and test part, so that the data that is fed into the model for training are actually centralized with standard deviation equal to 1. One-hot encoding of the categorical variables is also done. For a feature with categories 1,2,3 and 4, the one-hot vector will have length 4, with a 1 in the correct index. For example,

$$y = (0, 0, 1, 0) \tag{52}$$

means that the category of y is 3. Scikit-learn have a module that does the one-hot encoding automatically with the python code

```
from sklearn.preprocessing import OneHotEncoder

onehotencoder = OneHotEncoder(categories="auto")

design_matrix = ColumnTransformer(
[("", onehotencoder, [i]),],
remainder="passthrough"
).fit_transform(design_matrix)
```

where [i] is a list of columns that we want to transform. Only the feature matrix for the classification problem on credit card data are one-hot encoded in the following analysis.

3 IMPLEMENTATION

The details on the implementation and how to use the methods and classes are in the Github repository `https://github.com/MartinKHovden/FYS-STK4155-project-2`. Classes for building neural networks from scratch are in the file `nn.py`. The main class is the nn class, which represents the full network and stores each layer and information about the network. The backpropagation and feed forward algorithms is implemented on matrix form as described in the theory part. The nn class can be used for both classification and regression by choosing the appropriate output layer and cost function. The file also contains a class for creating layers in the neural network. The layers can be customized with activation functions and different numbers of neurons in each layer. After creating an object of the neural network, layers are added to that object. The network is fitted using the back-propagation algorithm and the stochastic gradient descent optimizer. When fitting the network there is possible to choose the learning rate, batch size, number of epochs and the regularization parameter. Making objects for each layers gives the user the freedom to build the network as they want. However, this leads to a large amount of different parameters to tune. We will consider only a small part of this space of parameter combinations. A grid search is performed for optimizing the hyperparameters. The file nn.py also includes function for calculating area ratio, auc score, accuracy and MSE using 5-fold cross validation. Similarly, in the file `library.py` there is class for logistic regression. This class uses the normal gradient descent algorithm when optimizing the beta coefficients. When training the model there is possible to choose the learning rate and number of iterations for the gradient descent optimizer. `library.py` also includes functions for calculating cross-validation area ratio, accuracy and MSE.

Scripts for producing the different results and plots for each part of the project are also in the GitHub repository. There is both jupyter notebooks and normal python scripts that can be run on your own computer. The normal python scripts might require commenting out some of code to avoid running the most time consuming calculations each time if its not wanted. Using the jupyter notebooks will not lead to this problem. In the jupyter notebooks, benchmarks for the different models are shown for varying hyperparameters. In addition, in the plots folder, plots from chosen runs are shown. You can compare your runs of the code to these benchmarks to see if you get the same results.

To run tests on the implemented classes for the logistic regression and the neural network, one can simply run the

command `pytest test.py` in the terminal. Make sure to be in the correct directory. The tests are unit tests on the functions within the classes. They test for fixed cases to avoid the randomness. The tests primarily focus on checking if the main calculations and functions within the classes works as it should.

While developing the code, the models were tested against scikit-learn's implementations. The results were similar, however, the time it took to train the network was significantly faster using scikit-learn compared to the self made code. For a brief comparison, see the jupyter notebook `comparison.jpynb`. In addition, the results from the article found at `https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf` were used as a baseline, to check if we got reasonable results.

4 RESULTS

4.1 Classification of risky credit card customers

4.1.1 Credit card data

For classification we will study the "default of credit card clients" data set from UCI. The data set can be found at `https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)`. It is an extensively studied data set and our results will be compared to the results in the article from the original study of the data. The article can be found at `https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf`. The credit card data set consist of 30000 samples, were each sample have 24 features. The data was collected from a major Taiwanese bank in 2005. The goal of our study is to predict if a credit card customer will default within the next month. The target value is coded as a binary variable where 1 = default and 0 = no default. Out of the 30000 samples, 6636 customers defaulted on their credit card debt. The 23 remaining features will be used to predict defaults. They are a combination of categorical and continuous variables and includes personal information about the credit card customer in addition to information about credit card history. The features are described in the list below.

- X1: Amount of the given credit.
- X2: Gender (1 = male, 0 = female).
- X3: Education (1 = graduate school, 2 = university, 3 = high school, 4 = other).
- Marital status (1 = married, 2 = single, 3 = others).
- X5: Age (years).
- X5-X11: History of past monthly payment records. (from April to September, 2005) as follows: 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.

Description of each sample are copied from the link to the data set.

By studying the data a bit closer, we note that it need some cleaning. First of, some of the samples have unlabeled and undocumented features. These have to be put into appropriate categories. For "education" and "marital status" we put the unlabeled and undocumented into "other". For X5-X11 there is a lot of samples with value equal to -2 and 0. X5-X11 represent the delay (or not) of payment in the previous months. Since they have chosen to code "pay duly" as -1, I think that changing every sample with -2 or 0 in to -1 is the best choice. The continuous variables X12-X23 have some outliers. Those will be handled by scaling the data. Scaling the data often reduces the effects of outliers and increases the performance of the algorithms. Scaling of the data is done as described in the theory part. We also note that the data have a overrepresentation of data points with label 0 = no default. This is important to be aware of when choosing the metric to compare models. The accuracy might not be a good measure of performance. Since we have about 77% of samples with no default, a classifier that only returns 0's will get an accuracy of 77%. When comparing the models we therefore use the area ratio.

For a full description of the data preparation and how to implement it in python, see the jupyter notebook `Credit_card_default_exploration.jpynb`.

4.2 Results on the credit card data

In this section we look at the results on the classification problem using neural networks and logistic regression. The models are tested for different values of hyperparameters and architectures.

4.2.1 Logistic regression

The results from using logistic regression is shown in figure 2. The two parameters we will focus on is the learning rate and the number of iterations in the gradient descent. The best area ratio for logistic regression is 0.5058. This result is obtained using learning rate = 1e-6 and 500 iterations. However, when comparing to the best possible model with an area ratio of 1, this does not look very good.

By looking at the runs with learning rate 0.0001 or lower, the results does not improve significantly after 25 iterations. We can see a slight increase from 10 to 25 iterations. This is as expected because the gradient descent will often not have found the minimum after only 10 iterations. However, after 25 iterations, the gradient descent seems to have converged to the optimal parameters. By using learning rate = 0.001, we see that the area ratio decreases significantly. The accuracy also changes fast in the change from 0.0001 to 0.001. It looks like when using a too high learning rate that the parameters diverges away from the actual minimum, and classifies almost every point as a 0. The proportion of 0's was 77%, which is close to the accuracy when the learning rate was set to $\alpha = 0.001$. The area score is also very low which indicates that the model is not very good at classifying samples in to the right category.

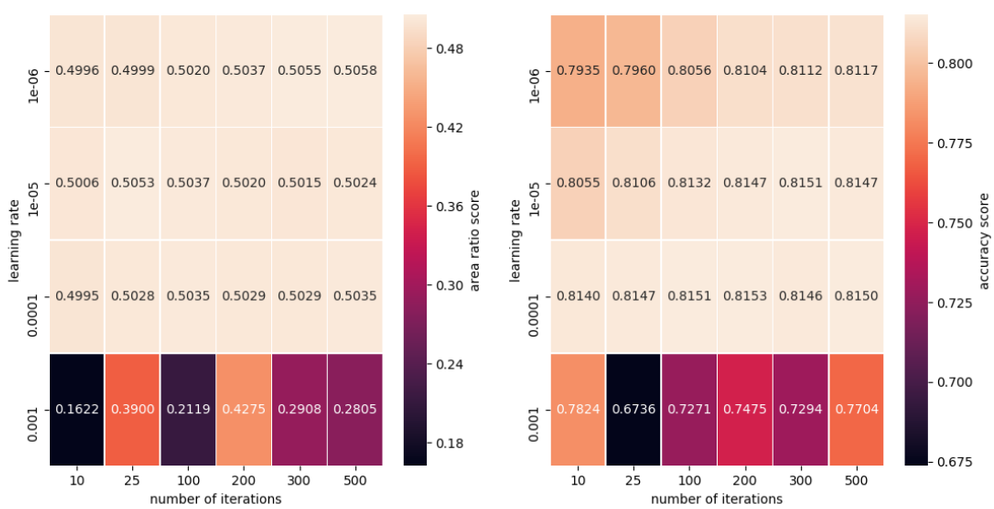
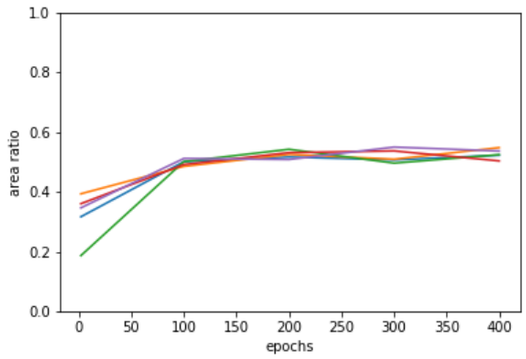


Figure 2: Area ratio-scores and accuracy scores for logistic regression for different variations of parameters on the credit card data. The test scores are estimated using 5-fold cross-validation

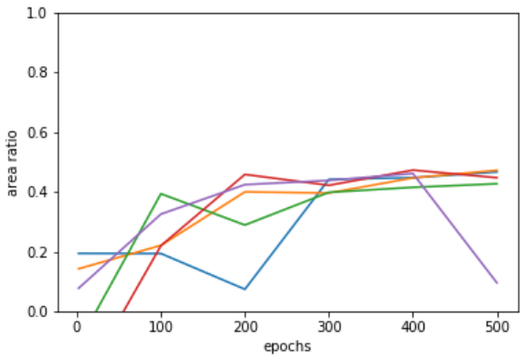
4.2.2 Neural networks

For the neural networks there is used one node in the output layer. All test were ran with batch-size = 10000 for the stochastic gradient descent algorithm. The neural networks have a larger set of parameters to tune compared to logistic regression, so an extra step in the analysis is needed. We will start by finding the number of epochs to run when fitting the neural network. In figure 3 we see how the area ratio changes as a function of epochs. It is tested using neural networks with 2 and 4 hidden layers. Each architecture is also tested with two different learning rates. The test are ran 5 times to get a more reliable picture of how the area ratio behaves as function of number of epochs. Each time the data are randomly split into a training and test part. The model is trained on the training part and the value shown are calculated on the test part. The first thing to note is that the convergence of the area ratio is much more stable with learning rate $\alpha = 0.001$. The converge is also faster, and after 100 epochs the results don't improve that much. When using a learning rate α= 0.00001, the area ratio fluctuates much more. After around 300 epochs it looks like the values stabilizes for the network with 2 hidden layers. For the network with 4 hidden layers and learning rate $\alpha = 0.00001$ the area ratio varies a lot, but after 300 epochs the improvement in the area ratio stops. Overall, it seems like two hidden layer is the best alternative. For the rest of the analysis we will do 300 epochs for a neural network with 2 hidden layers with sigmoid function on all layers. The batch size is set to 10000 after after some trial and error. Trying different variations of layers, neurons and epochs, this seemed to give good results without unreasonably high running times.

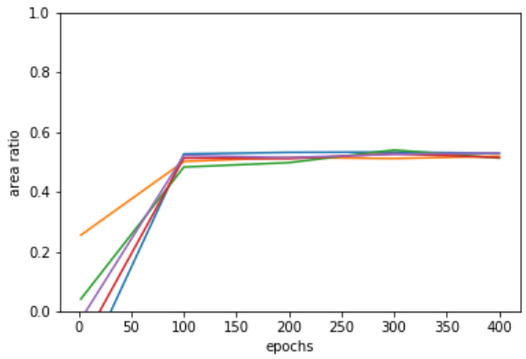
It is interesting to see if the results improves when introducing a regularization parameter to the cost function as discussed in the theory. The area ratio and the accuracy score for the neural network with two hidden layers are presented in figure 4 as a function of learning rate and regularization parameter. The values in the heatmap were found using 5-fold cross validation. For info about cross-validation see the report from project 1. From the two heatmaps, we can find the best combinations of parameters. The best accuracy is 0.8169 for learning rate set to $\alpha = 0.001$ and regularization paramater set to λ =1e-5. The accuracy score is however not that interesting as dicussed before since a model that only predicts 0's gets an accuracy of 0.78. However, the corresponding area ratio is 0.5147, and the model actually predict both 1's and 0's. The highest area ratio is 0.5198 for $\alpha = 0.001$ and $\lambda = 0.1$. From the heatmap is looks like the neural network is more sensitive to the choice of learning rate than logistic regression. By choosing a learning rate other than 0.001 or 0.0001, the area ratio and and the accuracy score drops significantly. The AUC-score drops more than the accuracy and even gets negative. This is because the model classifies almost every sample into 0, so the area ratio becomes very bad but the accuracy stays at around 0.78.



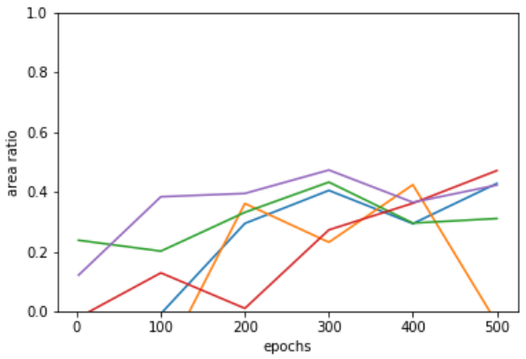
(a) Neural network with 2 hidden layers with 20 and 10 neurons. Every layer have the sigmoid activation function. No regularization parameter. Learning rate = 0.001. Batch size = 10000. Best area ratio = 0.5492.



(b) Neural network with 2 hidden layers with 20 and 10 neurons. Every layer have the sigmoid activation function. No regularization parameter. Learning rate = 0.00001. Batch size = 10000. Best area ratio = 0.477



(c) Neural network with 4 hidden layers with 30, 20, 20 and 10 neurons. Every layer have the sigmoid activation function. No regularization parameter. Learning rate = 0.001. Batch size = 10000. Best area ratio = 0.5391



(d) Neural network with 4 hidden layers with 30,20, 20 and 10 neurons. Every layer have the sigmoid activation function. No regularization parameter. Learning rate = 0.00001. Batch size = 10000. Best area ratio = 0.463

Figure 3: Area ratio

The best results are summarized in table 2. The best accuracy score is quite similar. However, the are ratio for the neural network is significantly higher than the area ratio for logistic regression. Even though the logistic regression obtained a bit poorer results, the training of the model was a lot faster. Compared to the results obtained in the article `https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf`, the accuracy is quite similar. They got an error 0.83 for the neural network and 0.82 for logistic regression. For both models their results were slightly higher than the accuracy we obtained. For the area ratio they got 0.54 for the neural network and 0.44 for logistic regression. Compared to 0.52 and 0.51, we see that their neural network performed better than ours. However, the self made logistic regression outperformed their by quite a bit. It would be interesting to how the self made neural network compared by creating more complex architectures.

Classifier	Test accuracy	Test area ratio
Neural network	0.8164	0.5198
Logistic regression	0.8153	0.5058

Table 2: Test accuracy and test AUC for classifying defaults on the credit card data using neural network and logistic regression. Results are for the best parameters found for each classifier and are found using 5-fold cross validation.

Using the parameters that had the highest area ratio score, we can compare the confusion matrix and the roc-curve of the two classifiers. The data was randomly split into a training part and a test part. The test part was 20% of the original data, and the model was fitted on the training data. From the test data, we computed the confusion matrix and the roc-curve. This was done 5 times and the average of the results are presented.

In the confusion matrices in figure 5, the neural network seems to be better at classifying 1's, compared to logistic regression. The neural network classifies 496 1's correctly. Compared to 309 for logistic regression. However, when looking at the the number of 0's predicted as 1's, the neural network comes out worse. It looks like the neural network classifies more samples as 1's while logistic regression classify more samples as 0's. By looking at the roc-curves and the AUC in figure the curves is very similar, which we would expect when they perform so similarly. The AUC is the area under the roc curve.

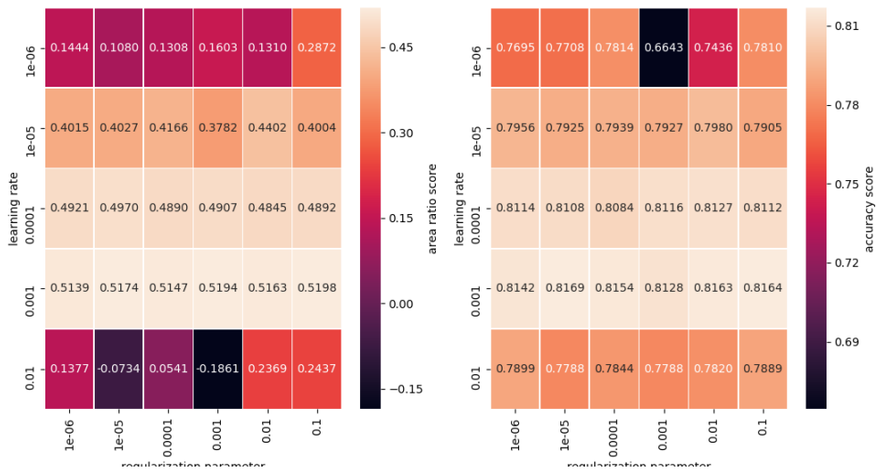
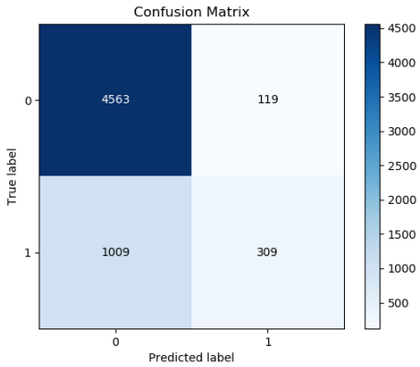
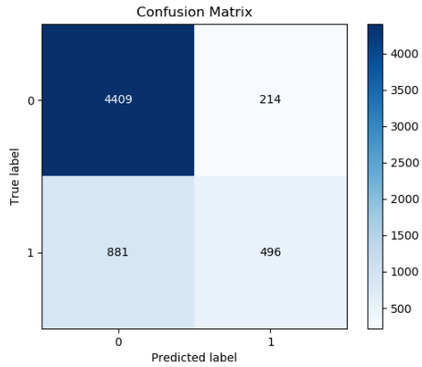


Figure 4: (Supposed to say "regularization parameter" on the lower axis. Did not notice until to late, and the script took over an hour to run...)Area ratio scores and accuracy scores for neural network for different variations of parameters on the credit card data. The test-scores were estimated using 5-fold cross-validation. The neural network had two hidden layers with 20 and 10 nodes and sigmoid activation function on all layers. Number of epochs = 300.

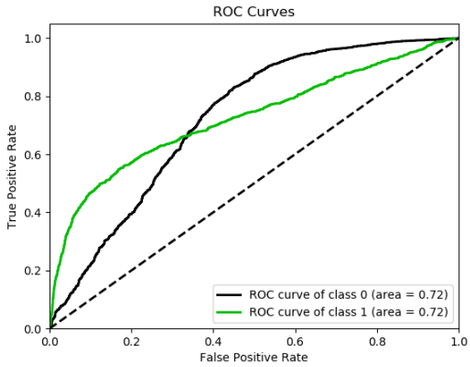


(a) Confusion matrix for logistic regression with learning rate = 0.0001 and iterations = 300. The values are calculated as the average over 5 runs.

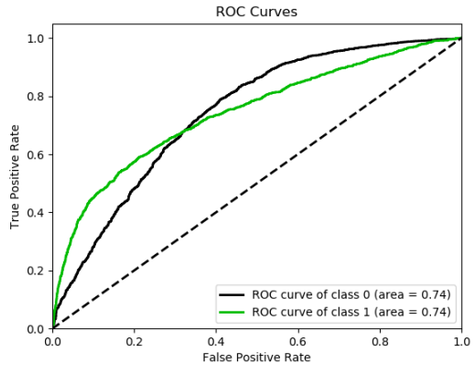


(b) Confusion matrix for neural network with two hidden layers with sigmoid activation function on each layer and epochs = 300 , batch size = 10000,learning rate = 0.001 and reg parameter = 0.01. Values are calculated as the average over 5 runs.

Figure 5: Confusion matrices for the best set of parameters on test data



(a) ROC-curve for logistic regression with learning rate = 0.0001 and iterations = 300.



(b) ROC-curve for neural network with two hidden layers with sigmoid activation function on each layer and epochs = 300 , batch size = 10000,learning rate = 0.001 and reg parameter = 0.01.

Figure 6: ROC-curves for the best set of parameters on test data

4.3 Regression on the Franke function

4.3.1 Description of the Franke function

For the regression part we will study the Franke function. The Franke function is given by

$$\begin{aligned} f(x,y) =& \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4}-\frac{(9y-2)^2}{4}\right)+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49}-\frac{(9y+1)^2}{10}\right)+ \\ & \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4}-\frac{(9y-3)^2}{4}\right)-\frac{1}{5}\exp\left(-(9x-4)^2-(9y-7)^2\right) \end{aligned} \tag{53}$$

The data set to be used will consist of calculated values of the franke function on a 20x20 grid in domain [0,1]x[0,1].

The results using neural networks will be compared to the results from project 1. The resutls from project 1 is used as benchmarks. In project 1 we studied ordinary least squares regression, Ridge regression and Lasso regression on the Franke function.

4.3.2 Results using neural networks

In figure 7 the test MSE is calculated for different values of learning rates, epochs and architectures of the network. The epochs refers to the number of epochs in the stochastic gradient descent. The batch size for the stochastic gradient descent are chosen to 50. This seemed to give the best results for this data set. The plots are generated by randomly splitting the data into a training part and a test part. The model is fitted on the training part, and the MSE are calculated using the test part. This was repeated 10 times for each set of parameters to get a better view of the test MSE of the model.

The difference between the two architectures had some effect on the MSE. For the architecture with 3 layers, the best test-MSE obtained was 0.0701 with learning rate = 0.001. For the architecture with 6 layers, the best test MSE was 0.0769 with learning rate = 0.001. The main characteristics to extract from figure 9 is that after 400-500 epochs, the MSE flattens out. I have tried running for larger values, but no significant increase in MSE can be seen. Using number of epochs higher than 500 does not seems to be any points. Timing of the fitting of the different networks are shown in the jupyter notebook `part_d.ipynb`. The time is a linear function of epochs, so choosing number of epochs = 500 seems to be a good trade-off between time and accuracy. In addition, it seems like that with the learning rate set to 0.001, the results is more stable. They converge faster and more consistently. The faster convergence comes from the larger step length. However, choosing even larger step-lengths might lead to overshooting the best solution and we might end up oscillating around the optimal solution. Since the network with three hidden layers seems to be the best, we will continue the analysis using this architecture.

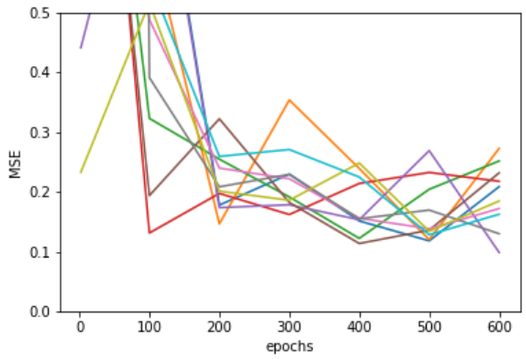
We can now see if we are able to improve these results. A neural network with 3 hidden layers with 20,10 and 5 neurons is used. Each hidden layer have the sigmoid activation function and the output layer does not have any activation function. SGD is used for optimizing the weights and biases with batch size set to 50. We will now try to add a regularization parameter as described in the theory part. We will do a grid search over variations of the learning rate and the regularization parameter. A heatmap of MSE values for different combinations are shown in figure 8. The values are calculated using 5-fold cross validation. By choosing $\alpha = 0.01$ and $\lambda = 1e-5$ we obtain the best MSE = 0.0045. From the heatmap it seems like by decreasing the learning rate the results gets worse. For the regularization parameter it does not seem to be any specific trend, and the results as a function of the regularization parameters varies a lot.

We can also try to fit a more complex polynomial by sending in a polynomial data set of the original features. According to the universal approximation theorem, a neural network with one hidden layer should be able to approximate any function. It is then interesting to see if we can get a better fit by sending in polynomial data. In figure 9 the MSE as a function of polynomial degree in the input are plotted for different values of the regularization parameter. The MSE increase with increasing polynomial degree, so it seems like the network are able to capture the non linearity's well by itself without sending in polynomial data. The optimal values are obtained with a polynomial degree equal to 1. Even though this is not enough to test the universal approximation theorem, we can see that the network are able to model the Franke function quite well on the original inout matrix.

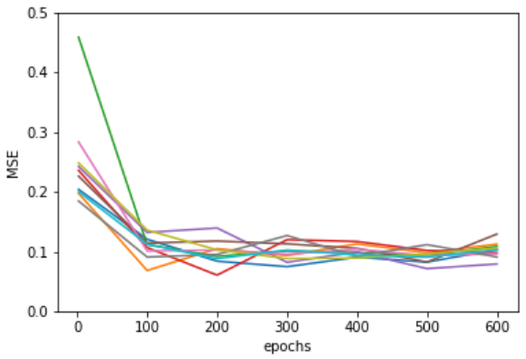
We can now compare the best results for the neural network with the results for least squares regression methods. In table 3 the results are summarized and the best results for each method are shown. We see that the neural network performed best on the Franke function. The optimal value was obtained with $\alpha = 0.01$ and $\lambda = 1e-5$ for a neural network with two hidden layers with 20 and 10 nodes in the layers and sigmoid activation functions.

Method	MSE
Neural network	4.5e-3
Linear regression	7.3e-3
Ridge regression	3.2e-2
Lasso regression	2.1e-1

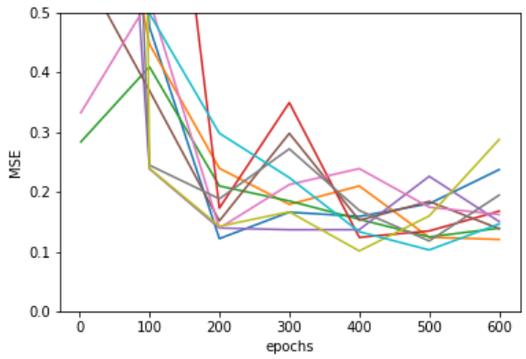
Table 3: MSE found with 5-fold cross-validation for each method. The best MSE are presented.



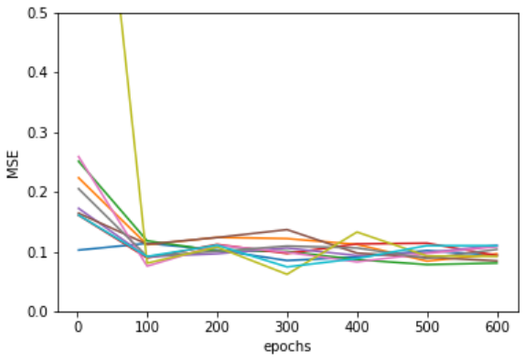
(a) Neural network with 3 hidden layers with 20, 10 and 5 neurons in each layer. Learning rate = 0.00001. All hidden layers have the sigmoid activation function. Batch size = 50. Best test-MSE = 0.0605



(b) Neural network with 3 hidden layers with 20, 10 and 5 neurons. Learning rate = 0.001. All hidden layers have the sigmoid activation function. Batch size = 50. Best test-MSE = 0.0983



(c) Neural network with 6 hidden layers with 50, 40, 30, 20, 10 and 5 neurons in the layers. All hidden layers have the sigmoid activation. Learning rate = 0.00001. Batch size = 50. Best test-MSE = 0.0625



(d) Neural network with 6 hidden layers with 50, 40, 30, 20, 10 and 5 neurons in the layers. All hidden layers have the sigmoid activation. Learning rate = 0.001. Batch size = 50. Best test-MSE = 0.1011

Figure 7: Test MSE for different network architectures

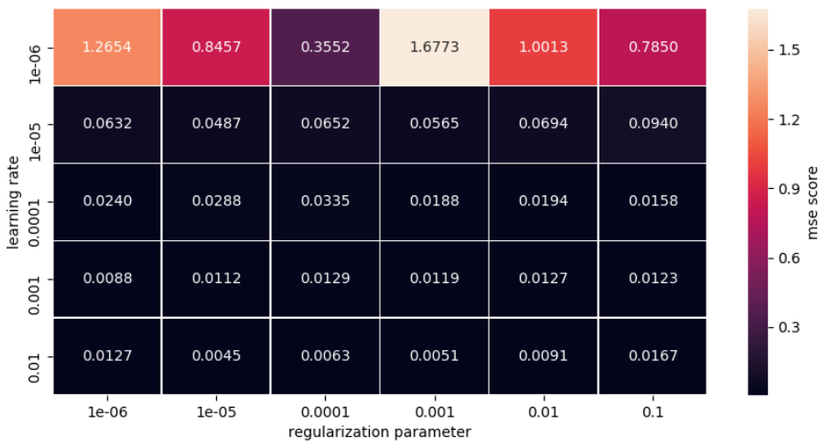
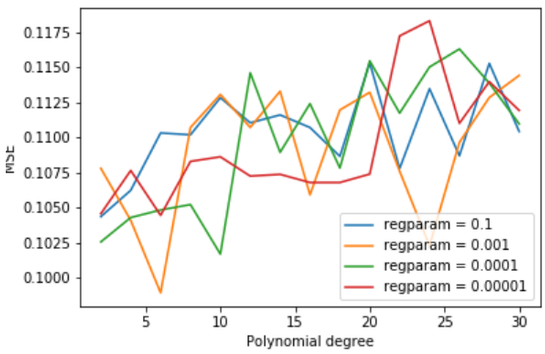


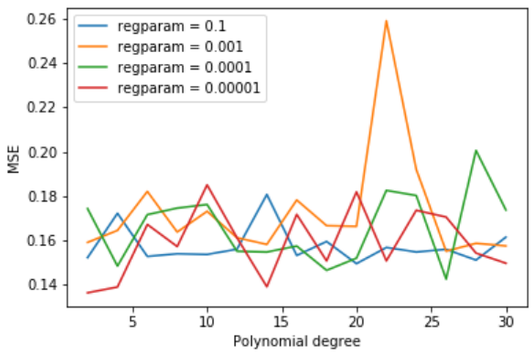
Figure 8: MSE values for different parameters in the neural network. The network consists of three hidden layers with sigmoid activation function on all hidden layers and 20, 10 and 5 neurons in the layers. No activation function on input and output layer.Batch size in SGD = 50 and number of epochs = 500. Values found with 5-fold cross-validation

5 CONCLUSION

In this paper we have studied different machine learning algorithms for classification and regression and studied the theory behind them in detail. The algorithms have been implemented in python from scratch. We have compared the algorithms for a wide variety of combinations of hyperparameters. For classification we used the credit card data set from UCI [6]. The results from logistic regression and the neural network was quite similar,but the neural network performed slightly better for both the area ratio and accuracy. By using the neural network we were able to get an area ratio of 0.5198 while we got 0.5085 using logistic regression. For regression we studied the Franke function. The results from using the neural network were compared to linear regression methods like OLS, Ridge regression and Lasso regression. Again, the neural network performed best and obtained an MSE of 0.0045 compared to 0.0073 for linear regression.



(a) Neural network with 3 hidden layers with 20, 10 and 5 neurons in each layer. Learning rate = 0.001. All hidden layers have the sigmoid activation function. Batch size = 50.



(b) Neural network with 3 hidden layers with 20, 10 and 5 neurons. Learning rate = 0.00001. All hidden layers have the sigmoid activation function. Batch size = 50.

Figure 9: MSE for different polynomial degrees

From this we can conclude that the neural network was the best model for both classification and regression on the data credit card data and on the Franke function.

One of the main problems I came across during the analysis of the data was the time it took fitting the neural networks. When doing 5-fold cross-validation for a large grid-search, it took a lot of time. An idea might be to try more complex architectures on a more powerful computer to see if it is possible to get results closer to the article [5] on the credit card data. A larger part of the parameter space might be investigated if more computing power were available. As can be seen in the jupyter notebook comparison.jpynb, the self made code is extremely slow compared to scikit-learn's implementation. I didn't have the time to optimize my code, so this might be a good idea for future work. It is probably the combination of a slow implementation of the backpropagation algorithm, and the gradient descent algorithm that causes the long running times. On the other hand, the neural network got better results compared to the logistic regression and the linear regression models. The question is then if it worth the large amount of extra time for a small amount of extra area score. For this problem, I dont think so. However, for more sensitive problems where predictive performance is crucial, it might be.

References

[1] Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow.* O'REILLY, 2017.

[3] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer Series in Operating Research and Financial Engineering, 2006.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics, Second Edition, 2017.

[5] I-Cheng Yeh, Che-hui Lien. *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients.* `https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf`

[6] I-Cheng Yeh. *default of credit card clients Data Set* `https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients`