FYS-STK4155 – Project 2
Neural Networks and Logistic Regression

Julie Thingwall (`juliethi`)
Jonas Gahr Sturtzel Lunde (`jonassl`)
Jakob Borg (`jakobbor`)

also known as the three Js of the apocalypse

Duh - duh - duh ... duh-duh .... duhduh

- Mortens Phone

Abstract

The steady rise in computational power over the last handful of decades has given rise to a large interest in more power hungry machine learning models, like neural networks, to rival more conventional methods. In this report, we study the performance of neural networks in both regression and classification problems, and compare it to the performance of the more traditional methods of OLS regression, Ridge regression, and logistic regression. A large focus was put on the inner workings of the neural network and logistic regression models, which we both built from scratch, in Python. We have gathered, cleaned and studied data from the World Happiness Report for the regression fit, and credit card default data from a Taiwanese bank for the classification case. Hyperparameter optimization was performed over relevant parameters for all models. The neural network obtained an optimal R2 score of 0.765 in the regression case. This put it slightly behind that of Ridge, at 0.782, but head of OLS, at 0.743. In the classification case our neural network achieved an area score of 0.670, slightly tailing logistic regression, at 0.684. We also found that the classification data strongly favored a deep network, with few nodes but several layers, while the regression data seemed to prefer more narrow networks, with fewer layers, but many nodes.

Contents

1 Introduction

Classification and regression are the two most common problems in machine learning, both aiming to tackle the challenge of predicting one or more properties of a data set, given some explanatory variables. While regression aims to predict some continuous quantity, classifications strives to place data points in a series of discrete categories. Both problems have for ages been rigorously studied, and approached by regression and other classical statistical methods. In recent years, the rising popularity of neural networks have brought new approaches to both problems, and, in some cases, vastly improves upon the more old school approaches.

In this report we study both classification and regression using both the more orthodox methods, and developing neural networks for both cases. In the case of classification, we build a classifying neural network and write a code for logistic regression. These two methods are both explored separately, and also compared to study the differences in performance. Likewise for regression, we build a neural network, which is bench marked against the more orthodox method of linear regression, specifically Ridge regression, where we utilize the already implemented methods found in the Python package `scikit-learn`. We will also compare the performance of our self made neural network with the Python package `Keras`. The performance, properties, pros, and cons of each method is studied by applying them to two chosen datasets. For the regression problem, we use data from the World Happiness Report, 2015-2017, trying to predict the happiness score of given countries over the period. For the classification problem, credit card default data from customers of a Taiwan bank, April-September 2005, is used, in an attempt to classify who defaults on their debt.

2 Theory

2.1 Neural Networks

2.1.1 Brief Introduction to Artificial Neural Networks

In a superficial form, artificial neural networks are computational networks of artificial neurons able to learn by an iterative process of considering training data and matching the data features with expected outputs.

Theses networks, or systems, are heavily inspired by the biological decision making network of neurons present in the human brain. A vast connected network of neurons influencing each other by sending electromagnetic signals in between them. This system is responsible for all our rational (and irrational) thinking. For example when we observe some sort of event happening around us; the information obtained is fed into our network of neurons in the brain, which weighs up our thoughts and possible responses based on the event and our prior experiences, eventually leading to some sort of decision or sensation as a result of our observations.

In artificial neural networks this behavior is mimicked in a simplified form. Layers of artificial neurons are connected together and the electromagnetic signals are replaced by mathematical functions. Each connection between neurons are represented by a weight, which acts similarly to the prior knowledge or experiences of a person, and are used to quantify the significance of each connection. The goal of a neural network is to tune these weights of every connection in order to correctly estimate some reaction, or output, given a set of inputs. There are many kinds and families of neural networks, in this report we implement one of the simplest types; a fully connected feed-forward neural network (FFNN) with backward propagation and stochastic gradient decent.

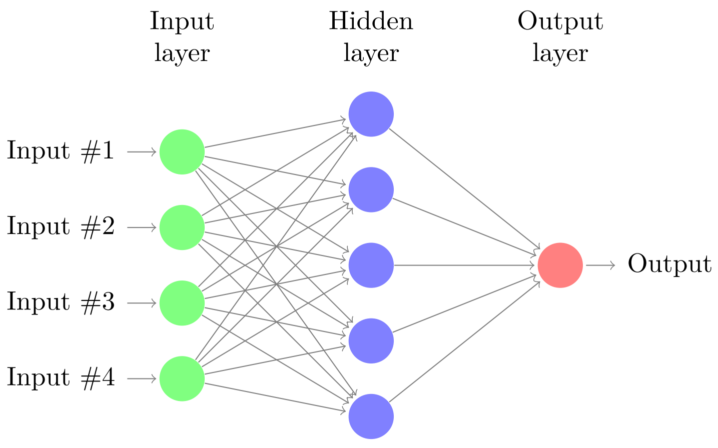


Figure 1 – *Conceptual sketch of the connections in a neural network with one hidden layer (Fauske, 2016). Each neuron in the preceding layers are connected to each neuron in the following layer with a weight and a bias parameter.*

2.1.2 Basic Expressions and Quantities

Let's describe the basic theory in terms of a simple example network, with an input layer, one hidden layer with m neurons and an output layer. Assuming a classification problem with binary outcome and a data set with a single feature; the input layer will then consist of one neuron for each data point in the data set $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$, and the output layer will have two neurons, one for each class. Each of the neurons in the input layer sends its output, which is just the data points, to each neuron in the hidden layer. The output of a layer we call $\boldsymbol{a}^l$ where l indicate the layer number. In the connection between the input layer, $l = 0$, and the hidden layer, $l = 1$, we multiple each output with its corresponding weight variable, and add a small bias term to prevent zero values. The result is the so called weighted input which is used as input to each neuron in the hidden layer

$$z_j^l = \sum_{i=1}^{n} w_{ji}^l a_i^{l-1} + b_j^l \quad \text{for} \quad i \in [1, 2, \dots, m] \tag{1}$$

with $l = 1$ for the weighted input in the hidden layer. This is how the neurons communicate, the output from the previous layer $\boldsymbol{a}^{l-1}$ is used together with the corresponding weights and biases to produce the input to the next layer. The input is then handled with an activation function in each following layer, so that the output from layer l is

$$a_j^l = \sigma\left(z_j^l\right) = \sigma\left(\sum_{i=1}^{n} w_{ji}^l a_i^{l-1} + b_j^l\right) \tag{2}$$

where σ is the activation function.

The activation function is the mathematical way of mimicking the way the biological neurons behave, which can be thought of as a neuron either responding or not responding (firing or not firing) based on the input. The choice of which activation function to use is just one of the many choices for hyperparameters one has to make when creating a neural network. Some of the most popular choices is the hyperbolic tangent, the Rectified linear unit (RELU) and the sigmoid function. The latter may be expressed as

$$\sigma(z) = \frac{1}{1+\exp(-z)}. \tag{3}$$

This function takes the value zero for large negative values of the weighted input, and the value one for large positive z, this we interpret as the neuron not firing or firing. The reason the sigmoid function is often used is because of its smooth behavior near $z = 0$, where it goes from $\sigma = 0$ to $\sigma = 1$ in a well behaved way. As we will see later this is crucial for how the network will learn, where we can infer a small change in the weights and biases in a specific layer which then produces a slightly different output from that layer. In our binary example we can define the resulting output class from the input depending on the value of the activation function in the last layer. For $\sigma(z^L) \geq 0.5$ we can define the output as class type 0, and for $\sigma(z^L) < 0.5$ the output is class type 1. Note that we use $l = L$ to indicate the last layer, the output layer.

The popular and simpler linear activation function, the RELU function, is expressed

$$\sigma(z) = \begin{cases} 0 & \text{for} \quad z < 0 \\ z & \text{for} \quad z \geq 0. \end{cases} \tag{4}$$

This behaves quite differently than the sigmoid function, with no upper limit on the value it can take, but the same lower value, and the discontinuous behavior near $z = 0$. Both functions has strengths and weaknesses, and its not perfectly clear why one seems to work better than the other in different cases. Both functions have simple derivatives, which will be important in the learning process discussed in section 2.1.6.

As mentioned, the choice of activation function is important when creating a neural network. There are a few constraints on this choice in order for the neural network to fulfill the universal approximation theorem which briefly states; a feed forward neural network with one hidden layer with a finite number of neurons are able to approximate any multidimensional continuous function on compact subsets of real functions to arbitrary accuracy given appropriate parameters and hyperparameters, *if* the activation functions in the hidden layers are

- bounded
- differentiable
- non-constant
- monotonically-increasing

This is a really powerful theorem, which gives the feed forward neural networks the potential of being so called universal approximators.

2.1.3 Feed Forward

This is the first part of the neural network, and is the method for moving information through the network from the input layer to the output layer. This step involves iterating through the layers, applying eqs. (1) and (2) in between each layer. Note that this system is easily extended to networks with more than one hidden layer. One would just keep iterating from the input layer through each hidden layer using the same equations, while changing the dimensions of the sums and vectors to reflect the number of neurons in the preceding and current layer. That is changing the m and n in the appropriate equations.

2.1.4 The Cost Function

After one complete feed forward procedure, the next step is to quantify the predicting power of the neural network. This is done using a cost function, which describes how well the network has approximated the output, $\boldsymbol{a}^L$, to the desired output (or target), $\boldsymbol{t}$. Again one are free to choose any cost function, but there are a few properties which are beneficial. The unknowns variables in the eqs. (1) and (2) are the weights and biases, and ultimately these are the variables we want the network to optimize in order to predict the output correctly. As a result, it's preferred that the cost function is a smooth function with nice derivatives with respect to the weights and biases. We'll come back to this in section 2.1.5. Some often used cost functions, which we will stick to in this report, are the mean square error for regression problems and the cross-entropy for classification problems. These are respectively, using $\hat{\theta}$ as a short hand for the appropriate parameters for the specific problem,

$$\mathcal{C}\left(\hat{\theta}\right) = \frac{1}{2n_{\text{output}}} \sum_{i=1}^{n_{\text{output}}} \left(a_i^L - t_i\right)^2 \tag{5}$$

$$\mathcal{C}\left(\hat{\theta}\right) = -\sum_{i=1}^{n_{\text{output}}} \left[a_i^L \log\bigl(p(a_i^L = 1)\bigr) + (1 - a_i^L)\log\bigl(1 - p(a_i^L = 1)\bigr)\right] \tag{6}$$

where the $p(a_i^L = 1)$ is the probability of input x_i being in class 1, and $1 - p(a_i^L = 1)$ is the probability of class 0, for a binary case. In the binary example, we can interpret the sigmoid function as a probability because it ranges from zero to one. But for a more generalized case, we introduce the softmax function as the activation function used in the output layer

$$a_j^L = P\left(\mathrm{class\, j} \mid \mathrm{z_j^L}\right) = \frac{\exp\bigl(z_j^L\bigr)}{\sum_{c=0}^{C-1}\exp(z_c^L)} \tag{7}$$

where C is the total number of categories, or classes or neurons, in the final layer, and the components of $\boldsymbol{z}^L$ is the probability of the output being in each available category. Here the so called one-hot encoding is used. For our binary example the targets are defined as

$$\boldsymbol{t} = \begin{cases} (1,0) & \text{for} \quad t = 0 \\ (0,1) & \text{for} \quad t = 1 \end{cases}$$

which is compared to the output of the last layer $\boldsymbol{a}^L$, where each neuron holds the probability of the input being in each category.

This is also easily extended to problems with arbitrary number of categories. To abbreviate the notation for the cross-entropy the loss function is used, so we can rewrite eq. (6) into

$$\mathcal{C}\left(\hat{\theta}\right) = \sum_{i=1}^{C} \mathcal{L}_i\left(\hat{\theta}\right). \tag{8}$$

2.1.5 Stochastic Gradient Decent

With the feed forward step and cost function defined now remains the optimizing (or learning) of the network. This process center around minimizing the cost function, and for this a stochastic gradient decent algorithm is used. Here the essence of the algorithm is stated briefly, for an excellent and detailed explanation we refer to Nielsen (2015). As mentioned minimizing the cost function is really finding the optimal weights and biases for the network to correctly predict the output. After one feed forward pass the gradient of the cost function with respect to the parameters can be used to adjust the value of the weights and biases in the direction which minimizes the cost function. This is an iterative process which slightly adjust the parameters for each iteration, until the minimum in the cost function is reached. This has to be done for each weight and bias in all the layers of the network according to

$$w_{ji}^l = w_{ji}^l - \eta \frac{\partial \mathcal{C}}{\partial w_{ji}^l} \tag{9}$$

$$b_j^l = b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} \tag{10}$$

where $\eta > 0$ is the learning rate hyperparameter.

In order to introduce stochasticity in the algorithm, the data set is randomly devided into subsets called minibatches, and the gradients are approximated on these minibatches instead. This reduces the chance of our gradients to fall into local minima of the cost function, as well as speeding up the calculations as we don't operate on the entire data set at once. This also introduces another hyperparameter to the network, the size, M, of the minibatches. With n original data points there will be n/M number of minibatches, each set denoted B_k for $k \in [1, 2, \dots, n/M]$. We then perform the gradient decent on each minibatch instead of the hole data set, iterating over the number of minibatches until the data set is exhausted. One full iteration over all the minibatches is called an epoch, which is yet another hyperparameter introduced, the number of epochs to train the network on. The gradients of the cost function is then written as

$$\boldsymbol{\nabla}_\theta \mathcal{C}(\theta) = \sum_{i=1}^{N} \boldsymbol{\nabla}_\theta \mathcal{L}_i(\theta) \rightarrow \sum_{i \in B_k} \boldsymbol{\nabla}_\theta \mathcal{L}_i(\theta) \tag{11}$$

summing over all the points in the respective minibatch B_k.

2.1.6 Backward Propagation Algorithm

Now the last piece is the algorithm for calculating every required gradient of the cost function with respect to the appropriate parameters, as in eqs. (9) and (10). For this the backward propagation algorithm is implemented. Here the need for an appropriate choice of cost function becomes clear, the gradients with respect to the weights and biases of the cost function is obtained through applying the chain rule in a methodical way. Here the final equations are listed, for an in depth explanation we again refer to Nielsen (2015). The propagation is an iterative process to find the error of each layer, defined $\delta_j^l = \frac{\partial C}{\partial z_j^l}$, starting from the output layer and iterating backwards to the first hidden layer. The error from the output layer is then defined as

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma\prime(z_j^L) \tag{12}$$

where $\sigma\prime(z_j^L)$ is the derivative of the activation function. With this quantity the error of the preceding layers are found by iterating backwards from $l = L$ to the first hidden layer $l = 1$ by (note the reversed order in the indices for the weights)

$$\delta_j^l = \sum_i \delta_i^{l+1} w_{ij}^{l+1} \sigma\prime(z_j^l). \tag{13}$$

Using these defined error quantities the gradients with respect to weights and biases may be expressed

$$\frac{\partial C}{\partial w_{ji}^l} = a_i^{l-1} \delta_j^l \tag{14}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{15}$$

for all layers $l \in [1, L]$. As mentioned when discussing the activation functions, its derivatives enter in eqs. (12) and (13) acting as a weight to the error used in the gradients. This introduces a potential problem for the rate of learning in the network, especially when using the sigmoid activation, called the vanishing gradients problem. For large weighted inputs to the sigmoid function, $|z| >> 0$, its derivatives are close to zero. This results in the gradients for the weights and biases being really small, and so the network will require a lot of iterations to optimize the quantities through eqs. (9) and (10). The RELU function behaves in a different way, since its derivative is either 0 or 1, and so the vanishing gradients are less of a problem as long as not all of the weighted inputs become less than zero.

2.2 Logistic regression

As mentioned, neural networks is a relatively new approach to solving classification problems. A more classical way of approaching such problems is logistic regression. While similar in name to linear regression, logistic regression differs in that it aims to predict discrete data points, not continuous. In essence, logistic regression classifies data in different groups or classes based on a set of inputs or predictors. The simplest case is where the output is binary, for example if a person pays their credit card bill or not.

In many ways, one can say that logistic regression is a very specific case of a neural network *(or that a neural network is an extension of logistic regression)*, with no hidden layers and no backward propagation. Thus, much of the theory will be similar.

2.2.1 The sigmoid function

While linear regression aims to find the best fit line through a set of data points, logistic regression tries to categorize data points. Thus, one needs a function where it's relatively straight forward to set a boundary between different classes. A much used example of such a function is the logistic function, or the sigmoid function, which takes the form

$$p(t) = \frac{1}{1+\exp(-t)}. \tag{16}$$

This function, as previously discussed in section 2.1.2, has the important property that it converges to 1 and 0 for large and small t respectively, making it ideal for classifying binary cases. Notice that eq. (16) is denoted as $p(t)$ and not $\sigma(z)$, as it was when speaking of neural networks. This is because, in the case of logistic regression, the sigmoid function represents a probability of a given output, not whether or not a neuron activates, making this notation more natural going forward.

In general, given a case with a binary output $y_i = \{0, 1\}$, a set of predictors $\boldsymbol{x}_i = [1, x_1, x_2, .., x_p]$ corresponding to output y_i, and a set of weights $\boldsymbol{\beta} = [\beta_0, \beta_1, ..., \beta_p]$, the probabilities of either output can be expressed as

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

$$p(y_i = 0|\boldsymbol{x}_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}). \tag{18}$$

2.2.2 Cost function

Just as with linear regression, the final model is controlled by a cost function. Simply put, the cost function $\mathcal{C}(\boldsymbol{\beta})$ is some function of the weights $\boldsymbol{\beta}$. This function has the property that the βs corresponding to the minimum of $\mathcal{C}(\boldsymbol{\beta})$ are the βs yielding the model with least error.

The important properties a cost function should have is thoroughly discussed in section 2.1.4, and it turns out that the same cost function used in a classification neural network can also be used in logistic regression. This cost function is the cross entropy function, as described in eq. (6). Rewriting it with the notation used here yields

$$\begin{aligned}\mathcal{C}(\boldsymbol{\beta}) = -&\left[\boldsymbol{y}^T \log(p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}))\right.\\ &\left.+ (1-\boldsymbol{y})^T \log(1 - p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}))\right],\end{aligned} \tag{19}$$

where $\boldsymbol{y}$ is a vector of all outputs y_i.

2.2.3 Minimizing the cost function

As described, minimizing the cost function is alpha and omega in logistic regression. There are many algorithms of varying complexity designed to do just this, and possibly the simplest one is simply called steepest descent. This is very reminiscent of the stochastic gradient descent described in section 2.1.5, but without the actual stochasticity. Steepest decent is an iterative algorithm where a new set of $\boldsymbol{\beta}$s is calculated using the gradient of the function one wishes to minimize. Simply put, it can be expressed as

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \eta\frac{\partial\mathcal{C}}{\partial\boldsymbol{\beta}}, \tag{20}$$

where $\eta > 0$ is the step size, or more commonly the learning rate of the algorithm, and the gradient $\frac{\partial\mathcal{C}}{\partial\boldsymbol{\beta}}$ is expressed as

$$\frac{\partial\mathcal{C}}{\partial\boldsymbol{\beta}} = -\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{p}), \tag{21}$$

where $\boldsymbol{p}$ is a vector containing the probabilities $p(y_i|, x_i, \boldsymbol{\beta})$ and $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ is the design or feature matrix holding all predictors.

2.3 Error metrics

To judge the proficiency of a model, one can employ some appropriate error metrics, making comparisons of models possible. For regression, the **R2 score** is commonly used. It is expressed as

$$R2(Y,\ Y_{\text{pred}}) = 1 - \frac{MSE(Y,\ Y_{\text{pred}})}{MSE(Y,\ \overline{Y})}, \tag{22}$$

where Y is the true values of some output dataset, $\overline{Y}$ is the average of Y, Y_{pred} is the predicition on Y. The R2 score represents 1 minus the ratio between the mean squared error of the prediction, and the mean squared error one would get from simply predicting the average of the model. In other words, R2 ranges from 0 to 1, with 0 representing the score of a completely naive model, and 1 a perfect model. The main advantage of the R2 score over mean squared error, is that it to some degree scaled by how difficult the model is to predict in the first place. It is also unitless.

For classification, two error metrics can be emoployed, among others. The first one being the **accuracy score**

$$\text{Accuracy} = \frac{\text{Nr. of correct predictions}}{\text{Total nr. of predictions}}, \tag{23}$$

which is simply the fraction of correct predictions. While this error metric is easily interpretable, it has one main drawback: It can be very biased if the dataset is biased. As an example, given a data set where 90% of the outputs are negatives one can get an accuracy of 90% by simply guessing that every output is negative.

In other words, when dealing with biased datasets, a metric which doesn't carry this bias is prefered. The **area score**, or "Area Under the ROC Curve" score, is one such metric, as it takes into account biases in the dataset. The area score will typically be 0.5 for any completely naive model, irregardless of bias in the dataset. The metric is explained in detail in Mishra (2018).

3 Method

3.1 Data

We gathered one dataset which presents a regression problem, and one which presents a classification problem. This sections presents both datasets, as well as the analysis and cleaning done on the data.

3.1.1 World Happiness Report - regression case

For the regression problem we used data from the World Happiness Report (WHR), years 2015, 2016, and 2017. The dataset was found at this link (`https://www.kaggle.com/unsdsn/world-happiness/`), and the data is explained and explored in depth in the World Happiness Reports themselves, found at this link (`https://worldhappiness.report/`).

The data presents 7 columns, explained in table 1. We will attempt to predict the given happiness score given by WHR using the columns 1-6 as explanatory variables. The columns are a combinations of purely objective data (GDP and life expectancy), as well as 4 columns based on questionnaires given by the Gallup World Poll. The full description of the columns, as well as the questions employed by the Gallup World Poll, can be found in the WHR itself.

Col. Nr	Column Description	Value Description
0	Happiness Score	The level of happiness, in the range 0-10, given by the World Happiness Report.
1	Economy	Log GDP per Capita, using PPP (purchasing power parity).
2	Family	Average response to question regarding support from family and friends, range [0,1].
3	Health	Life expectancy, scaled to range [0,1].
4	Freedom	Averaged response to question about freedom in life, range [0,1].
5	Trust	Averaged response to question about level of government corruption, range [0,1].
6	Generosity	Average response to question of personal generosity, range [0,1].

Table 1 – *Table describing the columns of the world happiness dataset. For columns 1, 2, 4, 5, and 6, the numbers are gathered from the Gallup World Poll. Column 1 is gathered from the World Development indicators released by the World Bank, while column 3 is gathered from the World Health Organization.*

The report contained the same columns for all three years, and we simply added the data from the different years together, such that e.g, Norway 2015 and Norway 2016 are two separate data points, with their own happiness score (and other variables).

3.1.2 Credit card defaults - classification case

For our classification problem, we used credit card default data (from now on refered to as the CC data) found on the UCI machine learning repository, found at this link (`https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients`). The data contains 24 columns, explained in table 2, and is gathered from credit card usage of 30'000 customers of a Taiwanese bank, over the span of April through September 2005.

The dataset contains information about the customers gender, educations, marital status, age, as well as their given credit cap in NT dollar (New Taiwan dollar), and information about their bills and repayments the last 6 months (April - September 2005). With all of these as explanatory variables, we will attempt to classify the final column of the set, which says whether the customer defaulted on their credit card debt the following month (October 2005).

Col. Nr	Column Description	Value Description
0	Given Credit Limit	Amount of given credit, in NT dollars.
1	Gender	1 = male 2 = female
2	Education	1 = grad school 2 = university 3 = high school 4 = other
3	Marital Status	1 = Married 2 = Single 3 = Other
4	Age	Age in years.
5-10	Repayment status for a given month.	-1 = paid duly 1 = delayed 1 month ... 9 = delayed 9 months
11-16	Bill amount for a given month.	Bill amount in NT dollar.
17-22	Payment amount for a given month.	Payment amount in NT dollar.
23	Defaulted on payment next month.	1 = Yes 0 = No

Table 2 – *Table describing the columns of the credit card dataset, which contains information of Taiwanese credit card customers. The columns for repayment status, bill amount, and repayment amount have 6 values, representing the months of April through September of 2005, in reverse order. The final column represents whether the individual defaulted on their payment the month of October 2005.*

3.1.3 Data cleaning

The WHR data was found to be extremely well behaved, having values in reasonable ranges with no outlying, missing, or otherwise bad behaving values. Therefore, no additional cleaning was needed for the regression dataset, and it was used as-is.

The credit card dataset have a handful of columns that need some sort of handling. The columns 1 through 10 are all discrete, with no possibility for large outliers. They do, however, have a large amount of undefined values. Columns 5-10, containing the repayment status, are defined for values of -1, and 1-9. However, as much as 53% and 13% of the entries in these columns take on the values of 0 and -2, respectively. A whooping 86% of the data contains 0 or -2 in at least one of these columns. These values are undefined by the provider of the data, but throwing away all this data would be a great sacrifice. Probably, one of these values refer to the data simply being missing or unavailable. Why there are two options for this, is unknown. We figured the value of 0 seemed appropriate for undefined values, and chose to set all values of -2 to 0 instead, while leaving the zeros be.

Columns 1-4 contain *some* values of 0, which again are technically undefined, but given they number in the sub-0.1%, we'll simply leave them be.

Columns 11 through 22, as well as 0, all contain continuous[1] values with units of NT dollars. These columns may have arbitrarily large values, and were observed to contain worrisome outliers.

We employed a common outlier removal technique, outlined in Sharma (2018). It involves dividing the dataset by 3 quantiles, Q1, Q2, and Q3, which, in ascending order, splits the data in 4 chuncks of equally many points, each containing 25% of the data. The *interquartile range* (IQR), which is the range which contains the middle 50% of the data, is defined as

$$IQR = Q3 - Q1$$

These quantities have the advantage of not being affected by outliers, since they only rely on the distribution of the data in the 0.25 - 0.75 percentile range.

It is common to remove data falling outside something like a $1.5 \cdot IQR$ interval outside the first and third quantile. Because of the shape of our data[2], we settle on a $1.0 \cdot IQR$ interval on the lower side, and a $2.0 \cdot IQR$ interval on the upper side. To summarize, for outlier removal of the columns 11 - 22, and 0, we removed all values that did not fall in the range

$$\left[Q1 - 1.0 \cdot IQR,\ Q3 + 2.0 \cdot IQR\right]$$

3.2 Polynomial Regression - OLS and Ridge

To establish a baseline for our neural networks, we implemented OLS and Ridge polynomial regression, with the intention of giving an idea of what kind of performance to expect. A comprehensive walkthrough of these methods can be found in Thingwall u. a. (2019). In summary, OLS serves as a very simple polynomial regression model, while Ridge builds upon it by introducing a suppression parameter λ, which reduces its tendency to overfit.

Using the WHR data, with a held-back testing set of 10%, we ran both OLS and Ridge for polynomial degrees 1 through 8, and Ridge with $\lambda \in [10^{-4},\ 10^{2}]$. The R2 score of the model applied to the testing data were averaged over 1000 runs to give consistent results. The implementation and results can be found in `world_happiness_linreg.ipynb`.

[1] Technically, money is discrete, but the values are so much larger than the discreteness size, making them almost continuous, for practical purposes.

[2] Our data is mostly very shallow on the upper range, and very steep on the lower range. The data contains amounts of money in a population, which is known to often follow a Pareto distribution, or other similarly steep curves.

3.3 Logistic Regression

As with the regression case, we also wanted to establish a baseline for the classification neural network. To do so, we implemented logistic regression, following the theory described in section 2.2, with the goal to study the CC data.

Our implementation was pretty straight forward. We wrote a class `LogReg` which takes in a matrix $\boldsymbol{X}$ of all predictors, a vector $\boldsymbol{Y}$ of all outputs, number of epochs and a learning rate. The number of epochs is simply the number of iterations used in the gradient descent algorithm.

To explore how logistic regression behaves for different parameters, we ran the code for different combinations of epochs and learning rates. Specifically with epochs in the range $100 \cdot 2^n$ for $n \in [0, 6]$ and learning rates 10^n for $n \in [0, -5]$. To make sure we got reliable results, we ran each combination of parameters 40 times and calculated an average of both the accuracy- and area score. This implementation can be found in `credit_card_Logreg.ipynb`.

3.4 Neural Network Design

Our goal for the neural network was to use it for both regression and classification problems, with as high adaptability as possible regarding all the hyperparameters. We therefore implemented an object oriented structure, as to not change the neural network code itself depending on the type of problem.

3.4.1 Code Structure

The code consist of three modules; the main class for the neural network and two slightly different classes to use for either classification problems or regression problems[3]. The neural network class itself is written in a very general way, following the equations from section 2.1 closely without specifying either the cost function or the activation functions used. These are determined in either the classification problem class or the regression class, which are used as input to the neural network, and from now on referred to as the problem classes. When initializing the problem classes the type of activation functions to be used in the hidden layers and the output layer are set, as well as what cost function to use. These classes are written in a way so that it is easy to extend the system to include any activation functions and cost functions in the future.

Also, the equations in section 2.1 are rewritten into matrix vector multiplications to improve the performance of the calculations using the numpy library in python. Thus, for example, eq. (1) and eq. (13) simply becomes

$$\boldsymbol{z}^l = \left(\boldsymbol{w}^l\right)^T \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{24}$$

$$\boldsymbol{\delta}^l = \boldsymbol{\delta}^{l+1} \left(\boldsymbol{w}^{l+1}\right)^T \odot \sigma\prime(\boldsymbol{z}^l). \tag{25}$$

[3] These classes can be found in `neural_network.py`, `classification_problem.py` and `regression_problem.py` respectively.

3.4.2 Network Architecture

All hyperparameters, including the number of layers and the number of neurons in each of them, are set in the initialization of the network.

Input and output layer The number of neurons in the input layer are determined by the input data it self, having as many neurons as there are features in the data. The neurons in the output layer has to be set explicitly, determined by the type of problem and the data to be analyzed. For our cases, we used one-hot encoding for the binary classification problem with two neurons in the output layer. Each neuron then outputs the probability of the input being in each class. For the regression case only one output neuron where used, outputting a vector with values for each input point. Also, the activation function used in the output layer is set by the problem class, and used together with the cost function to determine to output error. Using softmax and cross-entropy for classification results in an output error of

$$\boldsymbol{\delta}^L = \boldsymbol{a}^l - \boldsymbol{t}. \tag{26}$$

Using the mean squared error for regression results in

$$\boldsymbol{\delta}^L = \frac{1}{n_{\text{output}}} \big(\boldsymbol{a}^l - \boldsymbol{t}\big) \odot \sigma\prime(\boldsymbol{z}^L). \tag{27}$$

Hidden layers The number of hidden layers and neurons in them are given as input to the network as a simple list containing the numbers of neurons to be used in each layer. Each layer only communicates with its closest neighbours so the shapes of the matrices used in the hidden layers all are determined by the number of neurons in the current and the previous layers. The activation function used in all the hidden layers are determined again by the problem class.

All the weights in the hidden and output layers are initialized with a small random value from the normal distribution with mean zero and a standard deviation of 0.05, with appropriate shapes as discussed. The biases for each layer is simpler, as its only one bias per neuron, and its initialized to a small nonzero value of 0.01 to remove zero inputs.

3.4.3 Training The Network

The stochastic gradient descent discussed in section 2.1.5 are implemented as the training algorithm. The number of epochs and the size of each minibatch is taken as input when initializing the network. The gradient descent algorithm loops over the number of epochs, and the number of iterations for each epoch which is determined from the batch size and the number of data points to exhaust each epoch. For each iteration a subset of the full data is chosen at random with replacement from iteration to iteration. The network then trains on these subsets instead of the full data set, sending the subset through the feed forward pass and the backward propagation, following the discussion in sections 2.1.5 and 2.1.6. Note that there is a small difference in the way the backward propagation algorithm is implemented compared to the equations listed, which will be explained in section 3.4.4. The descent method is implemented with some keyword arguments for tracking how well the network trains as a function of epochs. This tracking is a major performance hit, and so it is possible to turn off and on as needed. Again the code is implemented in a way that it is easy to specify and extend the metrics which are tracked. For regression we used the popular R2 score, and for classification the accuracy score was used, discussed in section 2.3.

3.4.4 Implementation of Backward Propagation

In order to simplify the implementation, the backward propagation algorithm uses two loops over the layers in the network. This way we found the indexing of the arrays and lists in use to be easiest, as well as dividing the method into two parts making debugging easier. The code is primarily intended for relatively small networks, and so this is not a big performance hit. The algorithm first find the output error according to eq. (12), and uses this to find the errors in all the hidden layers by eq. (13) attractively. Here we utilize the negative indexing of python so that we can establish a list of errors, but where the first element is the output error and the last element is the error of the first hidden layer. Then this list is reversed, and the method loops over the layers again starting from the first hidden layer to the output layer. The gradients are found using the previously found errors and activations from the feed forward pass. Here the code differ from the theory in eq. (14) because of the way each list of activations and errors are defined. As we don't have an error for the input layer, when looping through the layers the list of activations are skewed once relative to the errors. Thus the equation reads like

```
for l in range(L+1):
    ...
    grad_w_list.append(np.matmul(a_list[l].T,error_list[l])
    ...
    weights_list[l] -= eta*grad_w_list[l]
```

3.5 Employing the Neural Network

In addition to our own implementation, we build a neural network in Keras, which is a Python module build on top of the popular machine learning module Tensorflow. Keras offers very high level and simplistic implementations of neural networks, and we used it as a testing ground and comparison tool for our own network.

At first, we implemented an identical network structure both in our own NN class, and in Keras, to the best of our ability. Running with 2 hidden layers of 16 neurons, with the ReLu activation function, a batch size of 32, and a learning rate of 0.001, we ran both implementations, and plotted the train and test accuracy as a function of epochs. This was done both for classification, which can be found in `credit_card_NN_TF.ipynb` for Keras, and in `running_NN_classification.py` for our own implementation, and for regression, which can be found in `world_happiness_NN_TF.ipynb` for Keras, and in `running_NN_regression.py` for our own implementation.

Timing was also performed between the Keras implementation and our own. We ran both implementations for 1000 epochs, with a similar setup as described above, expect the size of the number of neurons per layer, which was varied between 10^1 and $10^{3.5}$. We ran the timings 20 times and averaged them, for increased accuracy. The implementation can be found in `Timing.ipynb`.

3.6 Hyperparameter Optimization

As in most statistical models, we wish to tune the different hyperparameters of the model to optimal values. This involves taking every parameter that can be freely tuned, like learning rate, and trying a lot of different values. When multiple such hyperparameters are present, every combination of them should be tested, as they are often strongly correlated.

Neural networks contain a very large amount of such hyperparameters, unlike simpler methods, like logistic regression. Even in their most basic form, they often contains something like 5 or 10 hyperparameters, ranging from what types of activation functions used, to what kind of optimizer used, to number of layers. This problem presents itself both for classification and regression neural networks, and presents some problems. The first and biggest problem is the runtime of such an optimization. Ideally, we would like to run something like 10 hyperparameters, where we would like each to take on average, perhaps 5 values each. This means $5^{10} \approx 10$ million combinations, where each configuration needs to be averaged over a large number of runs, to ensure consistent results.

The second problem is visualizing what combinations of hyperparameters give the best results. One could of course just state the best combinations of parameters, but the bigger picture is often of interest as well, especially since the performance of the parameters are often strongly correlated (tuning the learning rate often requires tuning the number of epochs, for example).

With this in mind, limited ourselves to four hyperparameters, namely number of layers, number of neurons per layer, number of epochs, and learning rate. After some initial testing, we found a reasonable range of values for each parameter. The chosen hyperparameters are listed in table 3 for regression, and in table 4 for classification. The total combinations number 240 and 108, respectively. Every combination was run 40 times, on a randomly selected held back 10% test data, and the respective test score (R2 or area score) averaged over the runs. Due to time constraints, the hyperparameter optimization was performed using a neural network implemented in Keras, and not our own implementation. The implementations can be found in `world_happiness_NN_TF.ipynb` for regression, and `credit_card_TF_NN.ipynb` for classification.

Number of layers:	1, 2, 4
Neurons per Layer:	4, 8, 16, 32
Epochs:	200, 400, 800, 1600, 3200
Learning Rate:	0.01, 0.001, 0.0001, 0.00001

Table 3 – *Table showing set of hyperparameters used for hyperparameter optimization of WHR regression neural network.*

Number of layers:	1, 2, 4
Neurons per Layer:	4, 8, 16, 32
Epochs:	50, 100, 200
Learning Rate:	0.01, 0.001, 0.0001

Table 4 – *Table showing set of hyperparameters used for hyperparameter optimization of CC classification neural network.*

4 Results

4.1 Data

4.1.1 World Happiness data

As the WHR data was found to be of acceptable quality, it was employed unchanged, with no data cleaning performed. It contained a total of 470 data points, corresponding to over 150 countries over the span of 3 years. The distribution of values by feature is shown in fig. 2. We see that all features have very reasonable ranges on their data, most in the order of unity.

Figure 3 shows the correlation matrix of the WHR data. We observe that all our explanatory variables correlate positively with the happiness score, and most of them quite a lot, with the two best predictors seemingly being economy and health. The generosity score stands out at the only parameter without significant correlation to happiness.

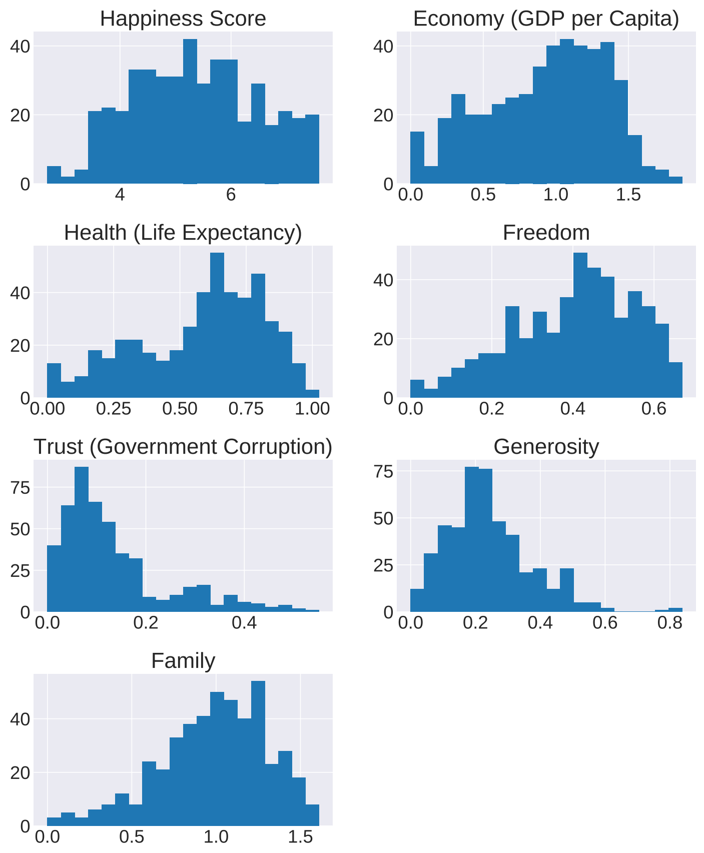


Figure 2 – *Histogram showing the distribution of values in the WHR dataset.*

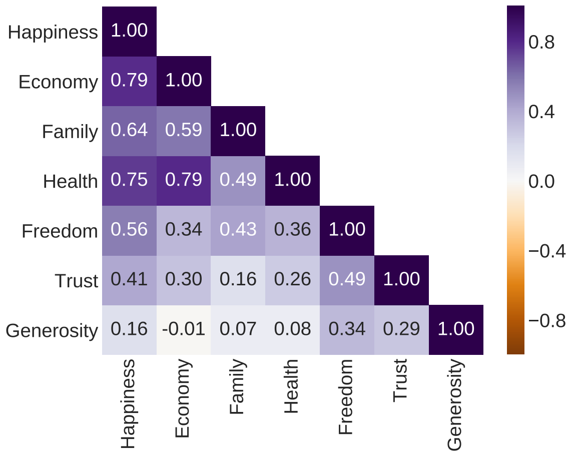


Figure 3 – *Heatmap showing the correlation matrix of the WHR data. The first column shows the correlation with the happiness score, the quantity we wish to predict.*

4.1.2 Credit Card data

Outlier removal, as outlined in section 3.1.3, was performed on the credit card data. This reduced the total number of data points from 30'000 to 21'308. As this is still a very respectable size for a dataset, and in no way a problematic reduction, no further attempt at making smaller and more surgical reductions was done. The reduction also had the fortunate effect of reducing the fraction of defaulting cases from 0.22 to 0.25, making the data slightly less biased towards not defaulting. It is, however important to keep in mind this bias, as any model simply guessing that nobody will default, will be right 75% of the time. In the appendix, figures 14, 15, 16, and 17 can be found, showing the cuts performed, and the reamdining data.

Figure 4 show the correlation between the default variable (valued 1 for defaulting and 0 for not defaulting) and the 23 explanatory variables. The most correlated quantities are the payment status, which measures the delay on the payment, in months (or -1 for no delay). The payment status is positively correlated with defaulting, saying that a delayed payment gives a more likely default. The payback amount on the bills is negatively correlated with default, meaning that those who pay more back each month are less likely to default. The credit limit is also negatively correlated with default, meaning people who have been given a higher credit limit is less likely to default. The remaining variables are very weakly correlated, and probably within the margin of error. Another natural observation is that, for the columns which spans several months, the later months (like PAY_1) are more strongly correlated than earlier months. None of these correlations are unexpected or seem unnatural.

The (very large) full correlation matrix, fig. 18, can be found in the appendix.

4.2 Regression

4.2.1 Establishing a baseline - OLS and Ridge

Figure 5 shows the R2 scores of performing OLS and Ridge regression on the WHR dataset, over a series of polynomial orders and λs.

We observe that OLS performs best at very low order, the best score coming out at 0.743, which is achieved with a simple linear fit. Probably due to the very limited size of the dataset, OLS actually manages to overfit already from a second order polynomial. Pretty impressive stuff.

Ridge outperforms OLS, capping out at an R2 score of 0.782, which is achieved over a span of several parameters. The optimal seem to be around $\lambda = 10^{-1.5}$, and polynomial order 4 or 5. The fact that Ridge outperforms OLS (and the fact that higher complexity Ridge outperforms lower complexity Ridge) shows that there actually exists higher order relations between the parameters, but, due to the limited size of the dataset, overfitting can run rampant. One of the main advantages of Ridge regression, its resistance to overfitting, is very evident.

We will keep in mind both the scores of 0.743 and 0.782, serving as fair baselines for a very simple, and slightly more sophisticated regression fit, respectively.

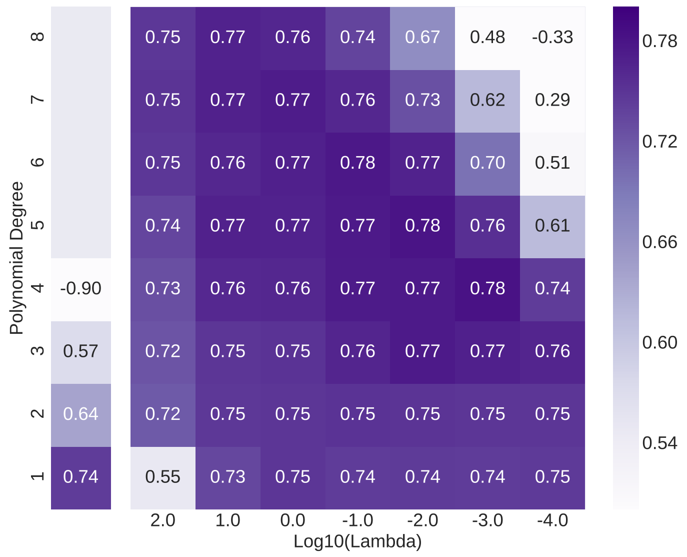


Figure 5 – *Figure showing R2 score of OLS and Ridge regression as function of polynomial degree and lambda, averaged over 1000 runs. R2 score is calculated on a 10% randomly selected test data. The column on the right shows the R2 scores of OLS, which doesn't have a lambda value, and only varies with polynomial order. The matrix on the right shows R2 scores of Ridge, which varies both with lambda and polynomial order. Ridge gives the best R2 score, of 0.783, in the region of $\lambda \approx 10^{-2}$, and polynomial order 4-5. OLS performs best for polynomial order 1, with an R2 score of 0.74.*

4.2.2 Neural Networks

Figure 6 show the R2 score of 10 random runs of our own neural network implementation, and a neural network implementation

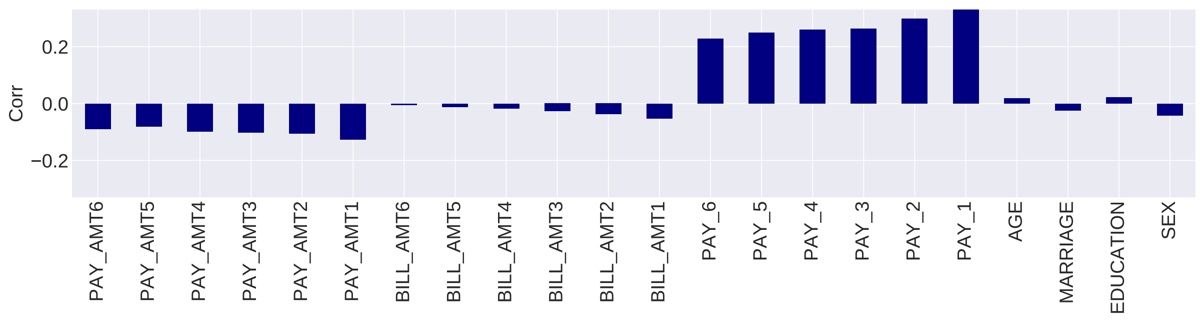


Figure 4 – *Figure showing the correlation between defaulting on credit card depth, and the explanatory variables.*

using Keras, on the WHR regression data. The idea was to confirm that our neural network had expected behavior by comparing it to an identical Keras implementation, before moving forward. The two networks do indeed show very similar behavior, and end up with near identical test data accuracies.[4] However, the Keras implementation of the NN seems to learn much quicker, reaching a steady state for the test data very early. It also seem to have much more volatile training accuracy, while the accuracy of both the test data of Keras, and both test and train data on our own network, show a much smoother behavior. We are unsure of what might be causing this, and suspect it might be some techicality buried in the Keras model. Either way, both networks show quantitatively similar behavior, and give very similar results in both training and testing score, and we conclude that our network works as intended.

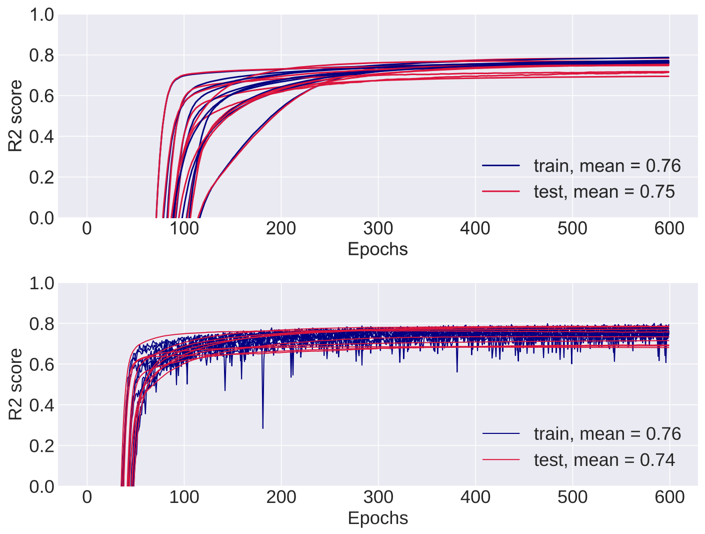


Figure 6 – *Figure showing the R2 scores of 10 random runs, using a self-built NN (top), and a Keras NN (bottom), trained on the WHR regression data. Both training data, and a randomly selected held-back 30% testing data, as function of number of epochs, are shown. Both networks employed 2 hidden layers of 16 neurons each, using the ReLu activation, and binary cross entropy loss. Both networks employ the SGD optimizer with a learning rate of* 10^{-3}.

4.2.3 Hyperparameter Optimization

The hyperparameter optimization on the WHR regression data was performed in Keras, using the parameters outlined in section 3.6. The 5 best performing networks are shown in table 5, giving a maximum R2 score of 0.765. There doesn't appear to be a whole lot of consistence in the types of nerworks that perform the best, as almost every possible value of every hyperparameter shows up in at least one of the five networks.

Figure 7 shows a somewhat clearer picture, where box plots of the R2 scores, as functions of number of layers and neurons, are shown. While we observe the same thing we observed in the table above - that almost every combination of parameters give *some* results which are promising, some parameters give much more consistenly good results. This specific problem seems to heavily favor narrow networks, with 1 layer performing, on average, much better than more layers do. The network does however seem to favor a higher number of neurons, with 16 and 32 both giving consistently good results.

Figure 8 shows the average R2 score of combinations of epochs and learning rate. Since these parameters are expected to be very heavily correlated, they are plotted as a heatmap instead of a boxplot. We observe that the network strongly favors relatively high learning rates, and a high number of epochs. The best results seem to appear with something like 800-1600 epochs, and a learning rate of 0.001.

R2 Score	Nr Layers	Nr Neurons	LR	Epochs
0.765	1	16	0.00100	1600
0.765	4	16	0.01000	3200
0.764	1	8	0.00100	800
0.762	4	32	0.01000	400
0.759	1	16	0.01000	3200

Table 5 – *Table showing the 5 best performing neural networks, measured in by R2 score on a held-back 10% test data, averaged over 40 runs for each combination of hyperparameters.*

[4]We have already established the area score as a superior metric to the accuracy score, and we would have prefered to use it. However, we found no simple way of storing it for each epoch in Keras, while this was no problem for the accuracy score. Since we're only trying to confirm the similar behavior of the networks, we found the accuracy score to be adequate.

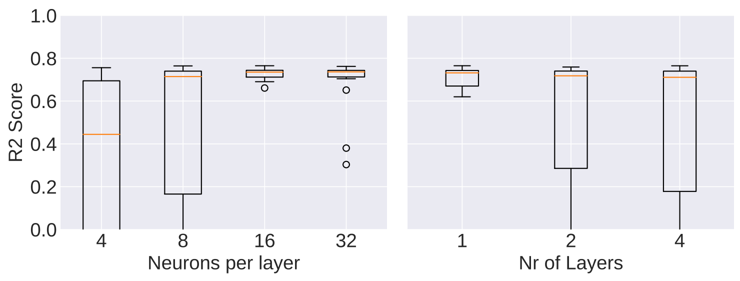


Figure 7 – *Box plots showing the distribution of R2 scores for varying number of neurons per layer, and varying number of layers. Each datapoint represents a unique combinations of hyperparameters, is the average R2 score of a 10% held-back test data, doing 40 repeated runs using the same hyperparameters.*

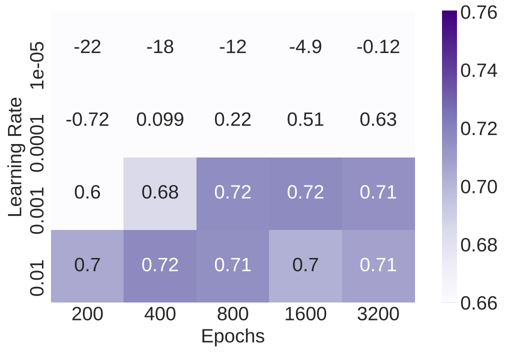


Figure 8 – *Heatmap showing the R2 score of the neural network fit over a series of epochs and learning rates, averaged over a series of other hyperparameters, as described in fig. 7*

4.2.4 Comparing the models

Interestingly, even after a hyperparameter optimization, we are unable to outperform Ridge regression, as seen in table 6. The NN places itself somewhere in the middle of the performance of OLS and Ridge, leaning somewhat more towards the latter.

	OLS	Ridge	NN
R2 Score	0.743	0.782	0.765

Table 6 – *Table showing best case R2 score of OLS, Ridge, and NN on 10% test data.*

4.3 Classification

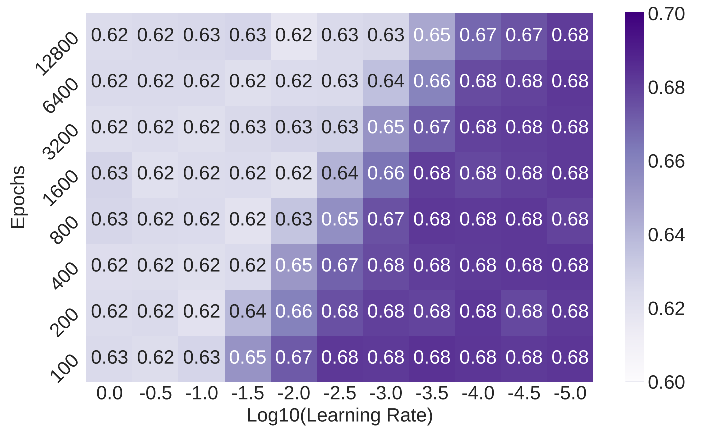
4.3.1 Logistic Regression

Figure 9 show the area score and accuracy score of the logistic regression classification, as a function of learning rate and number of epochs. We see that the best area and accuracy scores are, respectively, 0.684 and 0.790.

Remembering that a naive model guessing nobody will default gives an accuracy of 0.75, our score of 0.79 might not seem awfully impressive. The accuracy score is, however, not a great metric for performance. A completely naive model would have an area score of 0.5, which makes our score of 0.684 look at least like a fair improvement.

The figures also show that the area score performs very well at low learning rates, and low number of epochs, while the accuracy score performs best at high learning rates, and high number of epochs. They actually seem to be somewhat perfectly inverse, one performing well where the other struggles. This is extremely puzzling, as they should both, to some degree, indicate the performance of the classification. Area score is usually considered a better metric than accuracy, so one should think that the logistic regression performs worse the longer it trains. Often, this would indicate overfitting. However, a logistic regression model has extremely low complexity, and we find it unbelievable that is should be capable of overfitting a dataset of 20'000 points with 24 parameters.

An alternative explanation is that the cost function, which is in the end what we're minimizing, is biased towards higher accuracy, at the expense of area score. We do, however, find this somewhat hard to believe, since our cost function, binary cross entropy, seems better aligned with the area score than it does the accuracy score. In the end, we have no good explanation for this odd result.



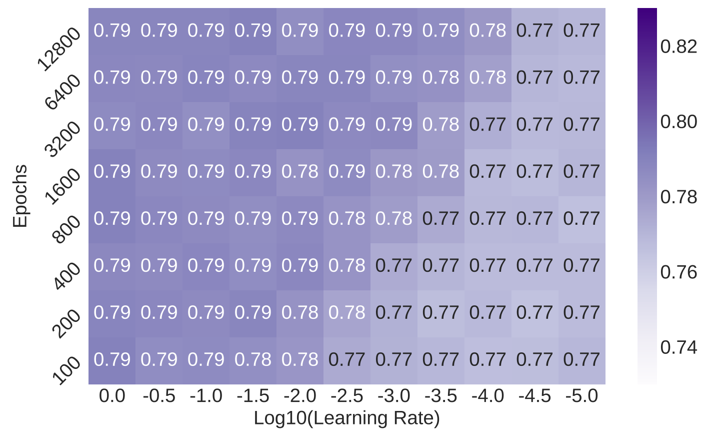


Figure 9 – *Heatmap showing area score (top) and accuracy score (bottom) of the logistic regression fit over combinations of learning rate and number of epochs used in training. The area score is calculated on a held-back 10% test data, and is averaged over 40 runs of randomly split train and test datasets.*

4.3.2 Neural Networks

The results of applying our home-made neural network, as well as a Keras implementation to the credit card classification dataset, is shown in fig. 10. We see that both networks, in all 10 runs shown, converge relatively rapidly to an accuracy somewhere slightly below 0.8. We remember from the logistic regression results that we got an optimal accuracy score of 0.79, which seems in line with what we're seeing here.

The Keras implementation converges quicker than the home-made NN. This is probably due to the usage of a more sophisticated optimizer (Adam vs SGD), because we had problems getting the Keras implementation to play nicely with SGD in the classification case[5]. It also seem to start out at a higher accuracy, which might be because the first point on the graph is calculated *after* the first epoch, after which the Adam optimizer might already have gained a lot of accuracy. It might also be due to different initialization techniques for the weights and biases.

The plots do, however, look qualitatively similar, and our neural network implementation seems to be working fine.

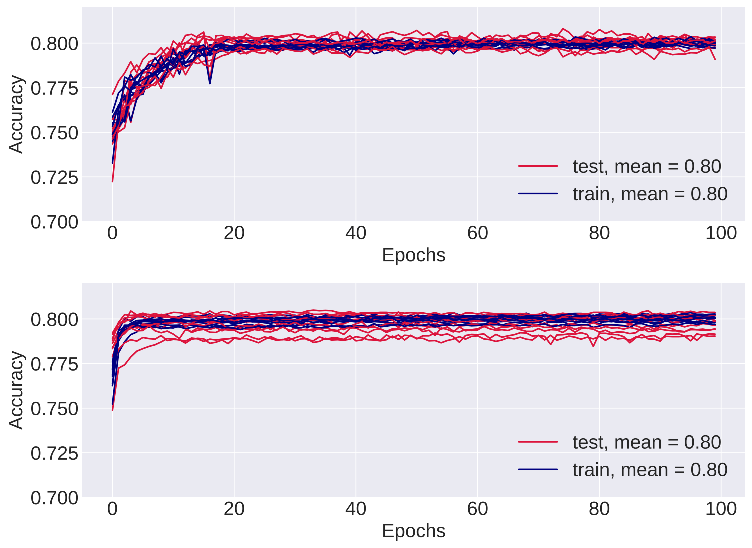


Figure 10 – *Figure showing the accuracy scores of 10 random runs, using a self-built NN (top), and a Keras NN (bottom), trained on the credit card classification data. Both training data, and a randomly selected held-back 30% testing data, as function of number of epochs, are shown. Both networks employed 2 hidden layers of 16 neurons each, using the ReLu activation, binary cross entropy loss, and a learning rate of 10^{-3}. For the self-made NN, the SGD optimizer was used, while the Keras NN employed the more sophisticated Adam optimizer.*

4.3.3 Hyperparameter optimization

The hyperparameter optimization on the CC classification data was performed in Keras, using the parameters outlined in section 3.6. The 5 best performing networks are shown in table 7, where we see that the best network performs an area score of 0.670, and an accuracy score of 0.799. We see that among the 5 best networks, every single one has 4 layers, and seem to prefer 16 nodes per layer. 50 epochs also seem to be the preference in these networks, while the learning rate is a bit all over the place.

Area Score	Acc Score	Nr Layers	Nr Neurons	LR	Epochs
0.670	0.799	4	16	0.0100	50
0.667	0.797	4	16	0.0010	50
0.666	0.799	4	32	0.0001	50
0.666	0.798	4	16	0.0001	50
0.664	0.797	4	8	0.0010	100

Table 7 – *Table showing the 5 best performing classification neural networks, measured by their accuracy and area scores, on a held-back 10% test data, averaged over 40 runs for each combination of hyperparameters.*

Figure 11 contains a box plot of area scores, over a series of different numbers of layers and neurons per layer. As observed in the table, we see that the best results are obtained with 4 layers, although it should be noted that the 2 layer model contains significantly fewer low outliers. Both 8, 16, and 32 neurons per layer seems to perform fine, but 16 outperforms the other two options by a small margin. It is very interesting to observe that, unlike the regression case, which strongly preferred a shallow network of only 1 hidden layer, the classification case prefers deeper networks, of 4 (or perhaps more) layers.

Figure 12 shows a heatmap of the average area scores for different combinations of epochs and learning rates. Interestingly, there seems to be very small preference in the learning rate or number of epochs for the classification model. This is in big contrast to the regression model, which suffered heavily for small number of epochs, or too low learning rate. It corresponds well to what we saw in fig. 10, namely that the classification neural network converges very quickly.

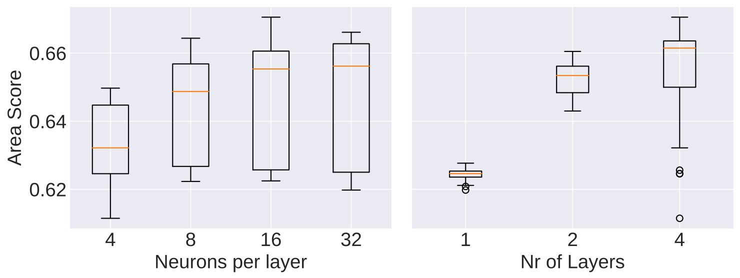


Figure 11 – *Box plots showing the distribution of area scores for varying number of neurons per layer, and varying number of layers. Each individual datapoint represents a unique combinations of hyperparameter, and each datapoint is the average area score of a 10% held-back test data, doing 40 repeated runs using the same hyperparameters.*

[5]It refused to learn properly, and gave terrible results

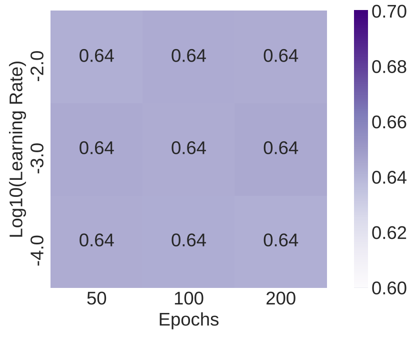


Figure 12 – *Heatmap showing the area scores for combinations of epochs and learning rates, with identical setup as fig. 11*

4.3.4 Comparing the models

Table 8 shows the accuracy and area score performance of the logistic regression and neural network models on a the credit card data. The performance of the models is strikingly similar, with the logistic regression performing somewhat better in area score, and the neural net somewhat better in accuracy score. Since we have declared the area score to be a superior metric, we conclude that the logistic regression outperforms the neural network, although with a very small margin.

	Log.Reg.	NN
Acc Score	0.790	0.799
Area Score	0.684	0.670

Table 8 – *Table showing best case accuracy and area score of logistic regression and NN on 10% test data.*

4.4 Timings

Figure 13 show the time-usage of the self implemented and Keras implemented networks, over a number of different neurons in each layer. For layer sizes smaller than ~ 100, our network outperforms Keras by a rather substantial factor. Keras seem to have some minimum runtime, no matter how small the network is, perhaps caused by compilation, or constant overhead on the propagations. For larger sets of neurons, Keras strongly outperforms our network, and probably employs more effective linear algebra operations for large matrices than we are capable of. The networks have equal performance around ~ 100 neurons per layer.

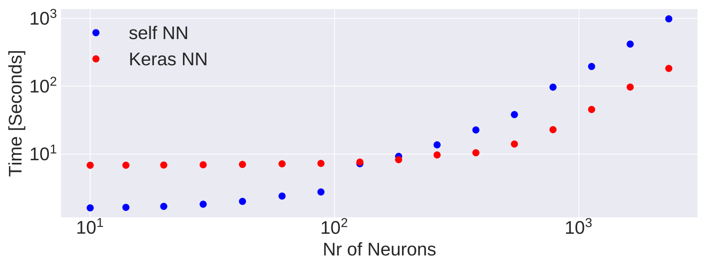


Figure 13 – *Figure showing time usage of training both the self written and Keras implemented regression NNs, using two hidden layers of variable sizes. The WHR dataset is used, with SGD, 1000 epochs and a batch size of 32, and averaged over 20 runs.*

5 Conclusion

In this project we have explored classification and regression problems, mainly focusing on the performance of neural networks, and logistic regression, to do such tasks. The inner workings of a neural net, and a logistic regression model, was of especial focus, and these models were both explored in theoretical detail, and built from the ground up in Python. For comparison, we employed OLS and Ridge regression from the Scikitlearn package, and a Keras implementation of a neural network. For regression, we studied and employed a dataset on international happiness rankings from the World Happiness Report, 2015-2017. For classification, a dataset on credit card defaults from a Taiwanese bank, 2005, was cleaned, studied, and used.

Both regression and classification showed very similar behavior in the final test data results between our self implemented neural network, and the Keras implementation. The runtime performance of the two implementations showed that our own implementation outperformed Keras in speed for narrow networks (< 100 neurons per layer), while Keras won for broader networks.

In regression, the neural network reached a final R2 score of 0.765, placing it in between the score obtained by OLS, of 0.743, and that of Ridge, of 0.782. The scores were all obtained through hyperparameter optimization of relevant variables, for all three models. The regression problem showed a clear preference for shallow, but broad neural networks, with only one hidden layer, but a large number of nodes.

In classification, the neural network again found itself slightly outperformed by more traditional models. The neural network achieved a area score of 0.670, while logistic regression ended up at 0.684. Again, the hyperparameters of both models were optimized. The classification problem favored deeper neural networks, performing its best at our maximum chosen depth of 4, while favoring a relatively narrow number of 16 neurons per layer.

5.1 Future Prospects

In this project we have been given the opportunity to delve deep into the workings of neural networks, both the professional Keras implementation and building our own from the ground up. Unfortunately the development of our own network was slow in the beginning, as we didn't find a good way to benchmark our code before the full network was in place, at which point it obviously was full of bugs. Because of time constraints and us being unsure of when we would get our own network to run, we did the hyperparameter optimization on the network built with Keras. To reduce the amount of results presented in this report we decided to not redo all of this work with our own network, and just compare the networks with the results obtained from the optimization of Keras. We didn't honestly expect our own network to perform inline with the professional tools, and in hindsight it would be really interesting to perform a proper hyperparameter optimization for our own network. Some of the same heatmaps presented here where also produced for our own network, but in a fast and ad-hoc way for fast comparison and benchmarking. In the end these are not included, as we would need a lot more computation time to produce consistent results by averaging over many runs.

As we saw from the timing of our networks in fig. 13, it was a pleasant surprise to note that our network actually outperformed Keras on smaller networks, with similar accuracy. We have a lot of future improvements in mind which would be fun and interesting to implement. First of, it would be interesting to optimize the code even further from a run time perspective, starting by optimizing the backward propagation algorithm for speed. Also, as the code is written in a highly modular and object oriented way, as discussed in section 3.4, further extension to include more available activation functions and cost functions in the problem classes would be easy to implement. It would also be fun to see the effects of having different activation functions in each hidden layer, especially for networks with high number of layers.

We would also very much have liked to explore a larger set of hyperparameters in our optimizations, particularly for the classification problem. The classification data seemed to prefer deeper networks, but we limited ourselves to four layers in our hyperparameter optimization. A drawback of introducing more layers is that it increases the complexity, and thereby the chance of overfitting. As we saw in fig. 11, while 4 layers gave the overall best results, it also had a significantly larger spread in values than the shallower. A common technique to counter overfitting in neural networks is *regularization*, which involves penalizing large activation on a single node with a new hyperparameter, in a similar concept as Ridge regression does with its introduction of the λ hyperparameter. It would have been very interesting to see if we could improve the performance of our classification network by further increasing the number of layers, and in addition, add regularization to the list of optimized hyperparameters, to counter the impact of overfitting, which might become a large problem for even deeper networks.

References

[Fauske 2016] Fauske, Kjell M.: *Example: Neural network*. 12 2016. – URL `http://www.texample.net/tikz/examples/neural-network/`

[Mishra 2018] Mishra, Aditya: *Metrics to Evaluate your Machine Learning Algorithm*. 2018. – URL `https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234`

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

[Sharma 2018] Sharma, Natasha: *Ways to Detect and Remove the Outliers*. 2018. – URL `https://towardsdatascience.com/ways-to-detect-and-remove-the-outliers-404d16608dba`

[Thingwall u. a. 2019] Thingwall, J. ; Lunde, J. ; Borg, J.: FYS-STK4155 Project 1 - Multiple Polynomial Regression. (2019). – URL `https://github.com/asdfbat/FYS-STK4155/blob/master/Project1/report/report.pdf`

Appendices

A Figures that didn't make it

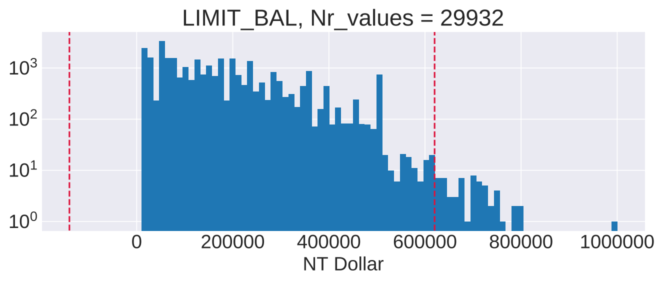


Figure 14 – *Histogram showing the distribution of credit limits, as well as the outlier cuts, as red lines.*

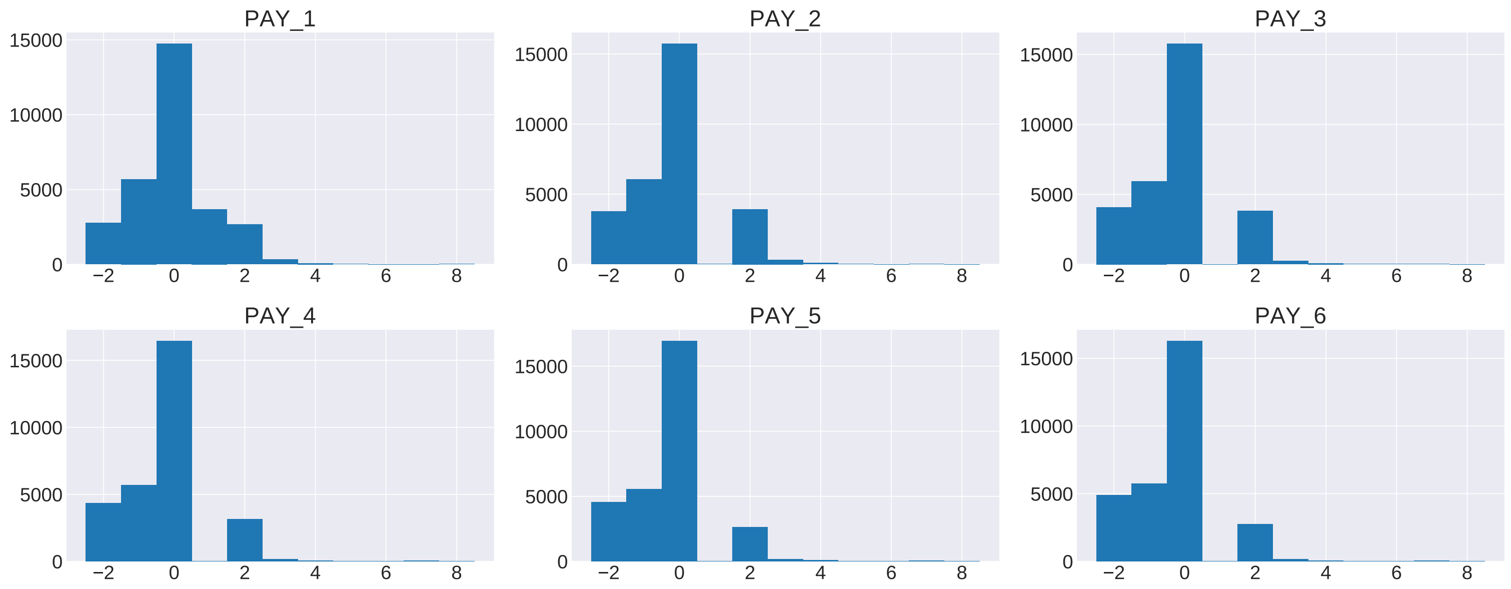


Figure 15 – *Histogram showing the distribution of payment statuses. Oddly, the value of 1 is missing from every column but PAY_1. It is unknown why.*

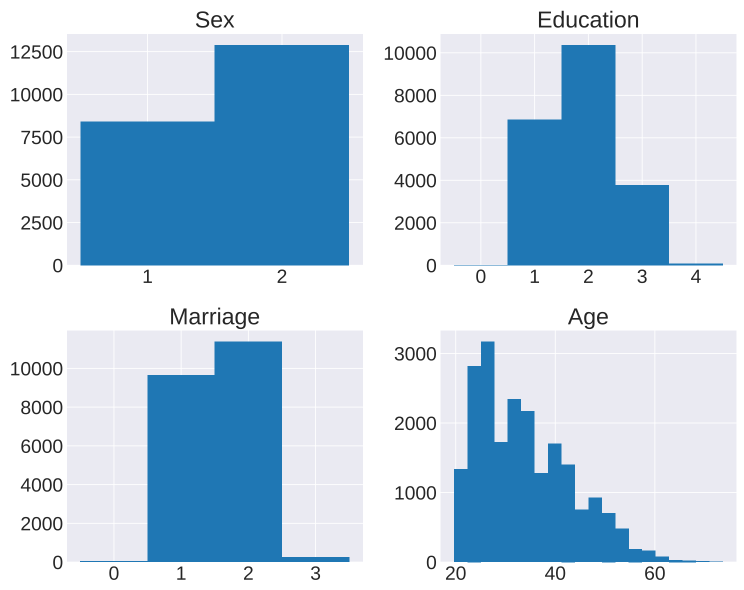


Figure 16 – *Histogram showing the distribution of credit limits, as well as the outlier cuts, as red lines.*

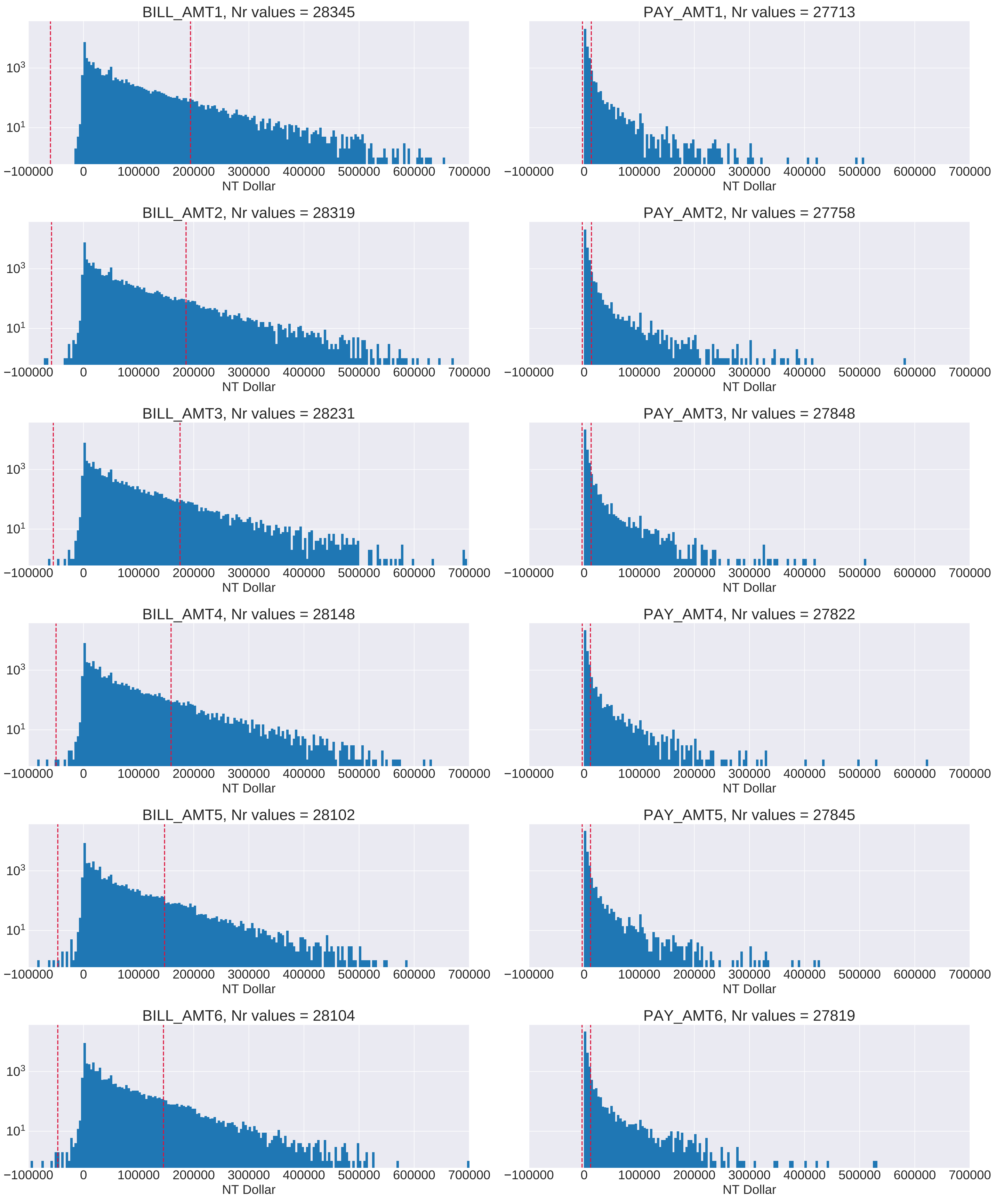


Figure 17 – *Histograms showing the distributions of values for bill amount (left) and payment amount (right) in the credit card data, as well as the outlier cuts, as red lines. The number of remaining values (down from 30'000) is shown in the title. Keep in mind that the y-axis is logarithmic, so the cuts aren't nearly as invasive as they might seem at first.*

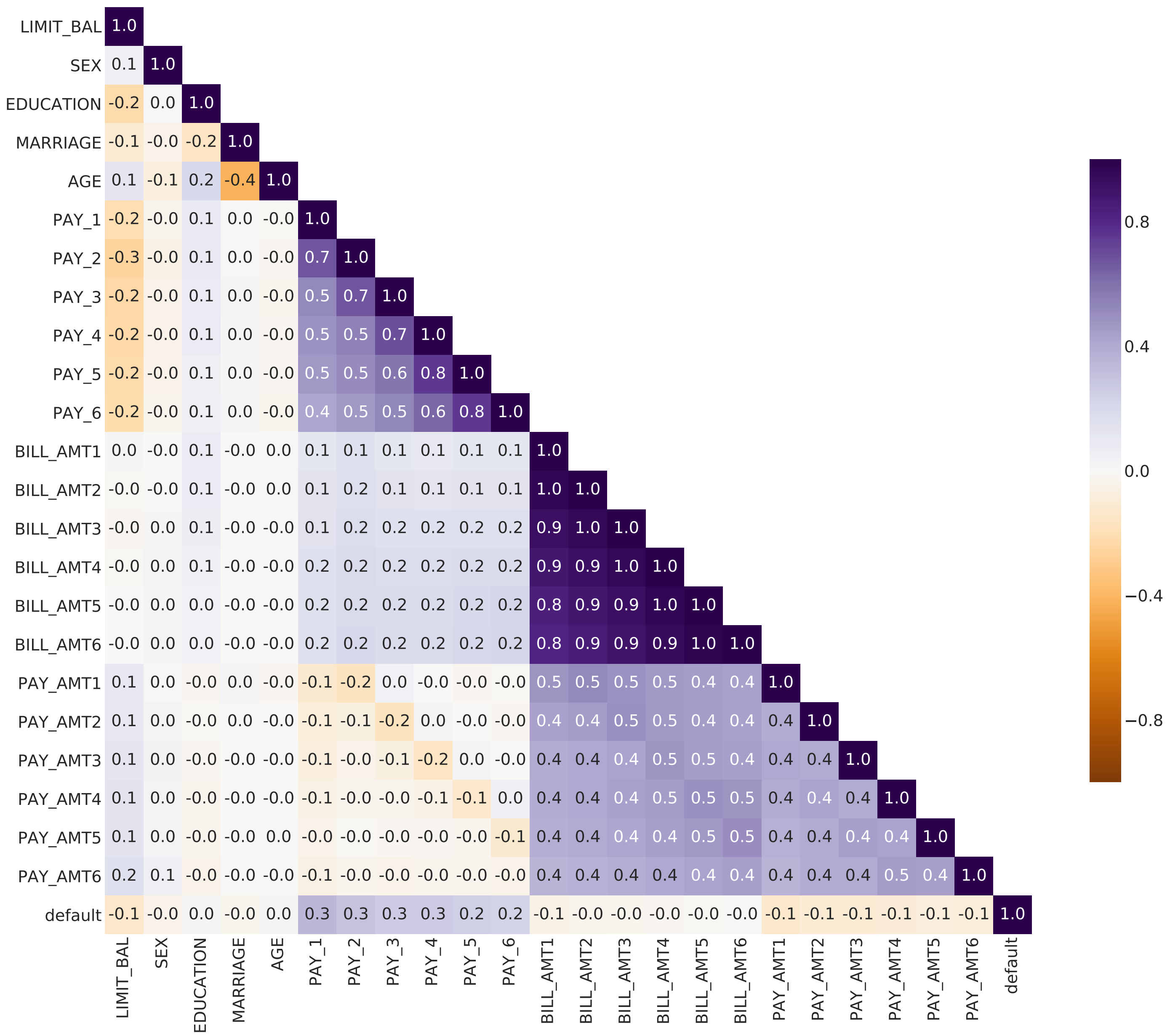


Figure 18 – *Correlation matrix between all the 24 columns of the credit card classification dataset.*