FYS-STK4155 Project #2 - Classification and Regression

Evaluation of Project number: 1
Name: Lennart Lehmann (ERASMUS Student - UiO Code lennarl)

All the Code and data can be found in my Github Repository (https://github.com/lenlehm/Classification-and-Regression).

Abstract ¶

Neural Networks could outperform Logistic Regression on the Classification of credit card defaults with respect to accuracy.
However, this benefit of accuracy comes to the cost of time as well as adds more complexity to search for optimal hyperparamters for the Neural Network.
Additionally, the accuracy could just slightly be improved compared to Logistic Regression.
Hence, the computationally more expensive and more complex models of Neural Networks are not superior for the credit card default data set.
When applying Neural Networks to Regression for terrain data of a region in Norway, they achieved slightly inferior results - but still great results - compared to traditional Regression methods such as Ordinary Least Squares (OLS) and Ridge Regression.
Again, Neural Networks are an overkill for this low-dimensional terrain Regression problem and should not be used for this underlying dataset due to their complexity and time consumption.

1.) Introduction

Neural networks have often been in the news and literally experienced a hype because of that. Many companies and people fear that Artificial Intelligence (AI) will eventually lead to machines taking over the world. This scenario is called *technological singularity* [1] and many famous people like Elon Musk or the deceased Stephen Hawkins warn about AI. [2] There also have been many popular movies depicting this machine dominating future such as *I Robot, Terminator, Ex Machina* and co. However, the most frequent use-case of neural networks are still classification and regression problems.
The rise of Deep Neural Networks emerged in 2012, where the winner of 2012 scored tremendous results on the *ImageNet Large Scale Visual Recognition Challenge (ILSVRC) (http://image-net.org)* by almost halving the Top-5-Error. [3]
Keep in mind that the Error for this challenge almost remained constant over the last years. This challenge focuses on object classification and localization with over 200 distinct objects, in fact an image recognition task.
This breakthrough in 2012 lead to rapid advances in AI, especially Image Classification, where only 3 years later, in 2015, human performance was already achieved. [3]
This rapid progress was utilized for similiar fields such as Regression or classification of non-images across every industry branch. [3]
Nowadays, these algorithms are used to detect cancer in medical images, or to perceive the environment of various agents, such as Autonomous Cars/ Drones or even robots. [3]
Classification is different from regression problems in the sense that algorithm's result can only take values across the classes that means to be classified. Hence, Linear Regression is not suitable for classification tasks, that is why this work focuses on Logistic Regression as well as Neural Networks (NN) for the classification of

Credit Card data.
Neural networks are a subfield of Artifical Intelligence and these algorithms model a complex function to represent the dataset that was given during training. During training the model learns the correlation of the input and its corresponding output. These trained models are then used to predict new, unknown datapoints that need to be classified.
Another advantage of Neural Networks is that they are also neat for Regression problems, thus this work will make use of Neural Networks as well Logistic Regression for Classification and Regression.
Both of these methods are supervised learning techniques, where a dataset comprising inputs and its corresponding targets/ outputs are necessary. The algorithm learns from the input corresponding target for each of the datapoints given. The targets differentiate for classification problems and regression problems, i.e. classification targets are discrete variables like the class names, such as *Cat* or *Dog*, when classifying images for cat or dog images repectively. Targets for regression problems are numerical values such as Stock market prices or housing prices.

2.) Theory

Since this work will make us of *Logistic Regression* as well as *Neural Networks*, the mathematical foundations for each of this algorithms is explained in the following section.
Furthermore, the second part of this project deals with Regression, where the two above mentioned algorithms are tested against the common Regression methods such as *Ordinary Least Squares, Ridge* and *Lasso*. I explained those in my first project (https://github.com/lenlehm/Regression-and-Resampling/blob/master/RegressionAnalysisAndResampling.ipynb), so feel free to refresh those if you feel like it.

2.1) Logistic Regression

Logistic regression can be considered a special case of linear regression with the neat benefit of simple probabilistic results for classification. Such a model specifies that an appropriate function of the fitted probability of the event is a linear function of the observed values X of the available predictors p. Unfortunately, this simple model cannot properly deal with the problems of non-linear and interactive effects of the predictor variables, such as normalization of the data. [4]
Assuming we have a dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y_i)\}_{i=1}^{n}$, where we have p predictors for each data sample $\boldsymbol{x}^{\boldsymbol{i}} = \{x_1^{(i)}, \ldots, x_p^{(i)}\}$.
The output y_i are discrete values and can take values from $k = 0, 1, \ldots, K-1$, for K classes.
Classification problems try to predict the output classes k_i for the given n samples comprising the p predictors.
Logistic Regression usually handles *binary classification* problems by assigning a probability to each of the two classes, thus meaning there are only two possible outcomes/ classes $y_i \in \{0, 1\}$ with each a probability of $p(y|x) \in \{0, \ldots, 1\}$. The following of this section will be according to the *binary classification* problem. Let $p(y|x)$ denote the probabilty of the outcome y given x, then the logistic model reads as follows:

$$p(y = 1|\boldsymbol{x}, \beta) = \frac{1}{1 + e^{-\beta \cdot \boldsymbol{x}}},$$

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

Note that $1 - p(x) = p(-x)$.
The paramters of the mode are denoted with $\beta = (\beta_0, \beta_1, \ldots, \beta_p)$. The term $\beta \cdot \boldsymbol{x} = \beta_0 + \sum_{k=1}^{p} \beta_k x_k$ is known as the log-odds and the following function is called the *sigmoid* of x:

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

Now the logistic model can predict a class $\hat{y}_i$ by utilizing the estimated probabilities $p(y|x)$:

$$\hat{y}_i = \begin{cases} 1 & if \quad p(y = 1|\boldsymbol{x}^{(i)} \geq 0.5 \\ 0 & if \quad p(y = 1|\boldsymbol{x}^{(i)} < 0.5. \end{cases}$$

For training of the logistic model, *maximum likelihood* is used. Under the i.i.d. assumption (identically independent distributed) the likelohood is given by: [6]

$$L(\beta) = \prod_{i:y_i=1} p(y_i = 1 | \boldsymbol{x}^{(i)}) \quad \prod_{i:y_i=0} p(y_i = 0 | \boldsymbol{x}^{(i)})$$

$$= \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i},$$

where $p_i = p(y_i = 1 | \boldsymbol{x}^{(i)}) = \sigma(\beta \boldsymbol{x}^{(i)})$.

The parameters β are chosen to maximize the likelihood. For the sake of simplicity, the maximum likelihood is often rewritten to the *log-likelihood* to turn the productions into summations:

$$\ell(\beta) = log(L(\beta)) = \sum_{i=1}^{n} y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i).$$

Since the logarithmus function is monotonous, maximizing the logarithm of a function is equivalent to maximizing the function itself. Hence β maximizes both, the log-likelihood along with the likelihood itself.
Lastly, the loss function for logistic regression is defined by the *binary cross-entropy* which is denoted as follows:

$$C(\beta) = -\sum_{i=1}^{n} y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i).$$

Minimizing the binary cross-entropy loss (equation (8)) yield the optimal paramters β. One can extend the binary cross-entropy equation by regularizing it with the L^1, L^2 or L^∞ - Norm. Considering the most common L^2 Regularization, (8) can be rewritten as:

$$C_{L^2}(\beta) = -\sum_{i=1}^{n} y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i) + \frac{\lambda}{2} ||\beta||^2,$$

where λ is the regularizing parameter and needs to follow: $\lambda > 0$. L^2 Regularization is more stable than its counterpart L^1, since it has a continuous derivative. However, since we square the differences, outliers are more sensitive (https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0) in the L^2 regularization. [12]
However, finding an analytical solution for the minimization problem is not possible. Making use of numerical optimization algorihtms like the notorious *(stochastic) gradient descent* eradicates this problem.
Looking at the gradient of the binary cross-entropy:

$$\frac{\partial C(\beta)}{\partial \beta_j} = -\sum_{i=1}^{n} \boldsymbol{x}_j^{(i)} \cdot (y^{(i)} - p_i),$$

which can be written in matrix form:

$$\nabla_\beta C(\beta) = -X^T \cdot (\boldsymbol{y} - \boldsymbol{p}). \qquad X \in \mathrm{R}^{n \times (p+1)}$$

X being the design-matrix containing $\boldsymbol{x}^{(i)}$ as its i-th row.

Convexity

A neat feature of convex function is that any local minimm is also a global minimum. Hence, if this function exhibits a minimum, one can say that this minimum is global and thus resulting in the optimal solution. Convexity is guaranteed (for multivariate functions) if the corresponding Hessian matrix of the second partial derivatives is PSD (positive semi-definite).
Showing that binary cross-entropy's (8) second partial derivatives is PSD, proves that this cost function is convex:

$$\frac{\partial^2 C(\beta)}{\partial \beta_k \partial \beta_j} = -\sum_{i=1}^{n} \boldsymbol{x}_j^{(i)} \cdot \frac{\partial (y^{(i)} - p_i)}{\partial \beta_k},$$

$$= -\sum_{i=1}^{n} \boldsymbol{x}_j^{(i)} \cdot \boldsymbol{x}_k^{(i)} \cdot p_i(p_i - 1).$$

Again (13) can be rewritten in matrix form:

$$\nabla_\beta^2 C(\beta) = -\sum_{i=1}^{n} \boldsymbol{x}^{(i)} \cdot (\boldsymbol{x}^{(i)})^T \cdot p_i(p_i - 1).$$

A matrix $M \in \mathrm{R}^{n \times n}$ is PSD, iff $\boldsymbol{z}^T A z \geq 0, \quad \forall z \in \mathrm{R}^n$.

Thus, we get following expression:

$$\boldsymbol{z}^T \nabla_\beta^2 C(\beta) \boldsymbol{z} = -\sum_{i=1}^{n} \boldsymbol{z}^T \cdot \boldsymbol{x}^{(i)} \cdot (\boldsymbol{x}^{(i)})^T \cdot \boldsymbol{z} \cdot p_i(p_i - 1)$$

$$= \sum_{i=1}^{n} ||(\boldsymbol{x}^{(i)})^T \boldsymbol{z}||^2 \cdot p_i(1 - p_i) \geq 0.$$

The inequality follwos due to the sum of non-negative terms. Hence, the Hessian of the binary cross-entropy is a convex function meaning that a local minimum is also a global minimum.

2.2) Neural Networks

Neural networks, as the name already hint, are supposed to mimic the human brain. Recent advances in Neural networks, especially Deep Neural Networks (DNN), was a breakthrough of these algorithms. These networks consist of an input layer, where the data is fed in, arbitrary many hidden layers as well as an output layer (see Fig. 1).
DNN Architecture
Fig. 1 Systematic architecture of a DNN with three hidden layers (blue rectangles) and four neurons or units per hidden layer (white circles inside blue rectangle). (Source: 7 (https://github.com/lenlehm/Classification-and-Regression/blob/master/E04-Deep_Learning.pdf))
A neural net is considered *deep*, when it has multiple hidden layers, thus the depicted network is considered a *deep neural network* since it utilizes three hidden layers.
Each of the inputs and neurons respectively have so-called *weights* $\boldsymbol{W}$ along with biases b per neuron that it counts to optimize. In Fig. 1 each of the edges you see from one neuron to another has a specific weight $w_{l,i}$, where $l \in (0, 1, \cdots, L)$ denotes the current layer of maximum layer size L and $i \in (0, 1, \cdots, N)$ describes the respective neuron in that layer, where N is the last Neuron in that specified layer l. The biases are not depicted in Fig. 1, but there is one bias value per neuron, thus resulting in $b \in (0, 1, \cdots, N)$ Biases per layer l with N neurons.
This architecture is also called *Mulit-layered Perceptrons (MLP)* where a MLP is build from layers of connected neurons. The input of the network is propagated through the layers and processed by each neuron in the network. That is also the reason why these networks are called *feed forward neural networks (FFNN)*, because the information flows through the network in forward direction (from input through layers to output - see Fig. 1).
The network itself outputs a value for a single neuron output i.e. binary classification or regression, or a vector for multi-class classification i.e. 200 classes in the ImageNet challenge (ILSRVC, see Introduction).
As you probably could already tell, the parameters of these network explode, the deeper (more layers) and wider (more neurons per layer) we get. For instance the DNN in Fig. 1 has a 3D input (x_1, x_2, x_3) and a 2D output (y_1, y_2). In between it has 3 hidden layers with 4 neurons each layer, thus resulting in 64 parameters:
$(3 \times 4)_{\boldsymbol{W_0}} \cdot (4 \times 4)_{\boldsymbol{W_1}} \cdot (4 \times 4)_{\boldsymbol{W_2}} \cdot (4 \times 2)_{\boldsymbol{W_3}} + (3 \times 4)_{\boldsymbol{B}} = 64.$

Note, that the addition at the end depicts the Bias vector, each neuron has a single bias value. Since we have 3 (layers) $\times$ 4 (neurons in each layer), we have to add 12 parameters for the biases.
This parameter space blows up pretty fast, especially considering complex tasks, where the input is higher dimensional such as 100-D input, which is not uncommon for real-world applications.
In order to optimize for the desired weights and biases an efficient algorithm for calculating the gradient of the entire function with repsect to the parameters is necessary.
Here comes the *backpropagation* very handy.

Backpropagation

As the name already states, there is not only a forward propagation, but also a *backpropagation*. This can olny be applied when a forward pass was done previously, since the calculated outut is necessary to start the backward propagation. During the forward pass the network calculates the outputs of each layer with respect to the activation function A:

$$O_1 = A(\boldsymbol{W_0}^T \cdot I), \qquad where \quad I = Input.$$

The activation function "activates" the neurons, which can also be rewritten with:

$$z_j^l = \sum_{i=1}^{n} w_{ij}^l \cdot x_i + b_j^l$$

as

$$a_j^l = f_l(z_j^l) = f_l(\sum_{i=1}^{n} w_{ij}^l \cdot x_i + b_j^l).$$

This is done for each layer, until arriving at the output layer. After the forward pass is done, the cost function is calculated along with its derivative w.r.t. the weights and biases in the output layer W^L.
Luckily, through the Chain rule of the gradients this allows us to chain the following gradients in the following manner:

$$\frac{\partial C(W^L)}{\partial w_{jk}^L} = \frac{\partial C(W^L)}{\partial a_j^L} \cdot \frac{\partial a_j^L)}{\partial w_{jk}^L} \quad = \frac{\partial C(W^L)}{\partial a_j^L} \cdot \frac{\partial a_j^L)}{\partial z_j^L} \cdot \frac{\partial z_j^L)}{\partial w_{jk}^L} \quad = \frac{\partial C(W^L)}{\partial a_j^L} \cdot f_L^{'}(z_j^L)a_k^{L-1},$$

where

$$\frac{\partial C(W^L)}{\partial a_j^L} = \frac{\partial}{\partial a_j^L} \cdot \left[\frac{1}{2}\sum_{i=1}^{N}(a_j^L - t_j)^2\right] = a_j^L - t_j,$$

and

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = \frac{\partial}{\partial w_{jk}^L} \cdot \left[\sum_{p=1}^{N} w_{jp}^L \cdot a_j^{L-1} + b_j^L\right] = a_j^{L-1}.$$

Defining everything except a_k^{L-1} in (20) as δ_j^L we end up in :

$$\frac{\partial C(W^L)}{\partial w_{jk}^L} = \delta_j^L \cdot a_k^{L-1}.$$

Guess what's up next: the lovely Chain rule ... again (applied to δ_j^L).

$$\delta_j^L = \frac{\partial C(W^L)}{\partial a_j^L}\frac{\partial f^L}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial b_j^L},$$

where making use of:

$$\frac{\partial b_j^L}{\partial z_j^L} = \left[\frac{\partial z_j^l}{\partial b_j^L}\right]^{-1} = \left[\frac{\partial}{\partial b_j^L}\sum_{i=1}^{N} L - 1 w_{ij}^L \cdot a_i^{L-1} + b_j^L\right]^{-1} = 1.$$

These are the derivatives of the cost function w.r.t. both weights (20) and biases (25) in the output layer W^L and $\boldsymbol{b}^L$.
The following equation holds for any layer, except the output layer:

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

Connecting this to the derivatives w.r.t. the next layer $l + 1$:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \cdot w_{kj}^{l+1} \cdot \frac{\partial f^l}{\partial z_j^l},$$

with

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} a_k^l + b_k^{l+1}\right] = \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} f^l(z_k^l) + b_k^{l+1}\right] = w_{jk}^{l+1} \cdot f^l(z_j^l).$$

Backpropagation is usually iterating Equation (27) and computing the gradients $\partial C/\partial w_{ij}^l$ and $\partial C/\partial b_i^l$ for each layer.

Cost Functions

All the above mentioned formulas share the same cost function C. The cost function is really important in the learning step since it is directly correlated with the accuracy. Finding an optimal cost function is often not easy. The cost function is an indicator of how well the network is doing. High loss indicates that the model is far away from the true values. Likewise, a low loss means that the model can fit well on the data and thus a low loss is desirable.
Most popular and common cost functions are *Mean-Squarred Error*, which is defined as: [6]

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2.$$

MSE is used in Regression tasks, since it gives a great score of how far away the predicted value from the true value is.
Beside MSE, there is also the popular Loss function called *Cross Entropy*, which is used in classification, because there is a known set of possible outcomes i.e. the classes. *Cross Entropy* is defined as: [8]

$$CE = -\sum_{x \in \mathcal{X}} p(x) log(q(x)),$$

where p and q are discrete probability distribution with the same $\mathcal{X}$.
When dealing with a binary classification problem, where only two classes are present (thus $y_i \in \{0, 1\}$) one could simplify (30) to:

$$CE_{binary} = -\frac{1}{N}\sum_{i=1}^{N} y_i log(p(y_i)) + (1 - y_i) log(1 - p(y_i)).$$

Gradient Descent

When the gradients are known, the paramters can be updated by gradient descent according to following formula:

$$\theta^{(t+1)} = \theta^{(t)} - \tau \nabla C(\theta^{(t)}), \quad \tau > 0$$

for timestep t and τ being the well-known and headache provoking *learning-rate*. Too small values for τ (< 0.000001) will take years until convergence is reached. However, too big values for τ (> 0.01) and again no convergence is reached, since the updates are too big and the algorithm jitters around. [3] Since the aim is to minimize and go towards the negative gradient (descent) we have to put a "minus" in the update formula (32). Gradient descent is the most common optimization algorithm in machine learning, due to its performance and simplicity. A model $m(\theta)$ is fit on a dataset X with a cost function $C(X, m(\theta))$, which evaluates the model on the underlying observations X. The model is fit by calculating the gradients and thus finding the optimal parameters θ that minimize the cost function $C(X, m(\theta)$.
As everything in life this method also has its drawbacks. There are usually more non-convex, high-dimensional cost functions that often result in local minima instead of a global minimum. Here the inital guess (timestep $t = 0$) of θ^0 is crucial for the performance of the Gradient descent and thus sensitive to the inital guess.
As mentioned earlier, Gradient Descent is very sensitive for the learning rate τ. There is a lot of research to find the optimal learning rate. Most of the recent published work, utilizes an adaptive learning rate, where the initial learning rate is high, but with each epoch the learning rate is reduced until it gets very small.
Additionally the gradient is a function of $\boldsymbol{x} = (x_1, \ldots, x_n)$, which makes it expensive to compute numerically.
One can alleviate the shortcomings by introducing randomness, i.e. when training in batches such as the Stochastic Gradient Descent (SGD).

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) address the drawbacks of vanilla gradient descent. The idea behind SGD is that the cost function can mostly be rewritten as a sum over datapoints:

$$C(\theta) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i, \theta).$$

Since the gradient is the working horse, this can also be computed as the sum over i-gradients:

$$\nabla_\theta C(\theta) = \sum_{i}^{n} \nabla_\theta c_i(\boldsymbol{x}_i, \theta).$$

Randomness is added by only taking the gradient on a subset of the data, often referred to *minibatches*. Assuming n datapoints and the size of minibatches M, there will be $\frac{n}{M}$ minibatches. In the following of this report, the minibatches are denoted by B_k where $k = 1, \ldots, \frac{n}{M}$. For instance one chooses $M = n$, yieldig a single datapoint in the minibatch: $B_k = \boldsymbol{x}_k$, or $M = 1$, then there is only one (Mini-)Batch B_1 containing all datapoints.
Approximating the gradient by replacing the sum over all datapoints by the sum over a randomly picked minibatch in each gradient descent step:

$$\nabla_\theta C(\theta) = \sum_{i=1}^{n} \nabla_\theta c_i(\boldsymbol{x}_i, \theta) \to \sum_{i \in B_k}^{n} \nabla_\theta c_i(\boldsymbol{x}_i, \theta).$$

Accordingly an update step now looks as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \tau \sum_{i \in B_k}^{n} \nabla_\theta c_i(\boldsymbol{x}_i, \theta),$$

where each minibatch B_k is picked randomly with equal probability from the interval $[1, \frac{n}{M}]$. One iteration over all minibatches is known as *epoch*. Hence, it is common to choose a number of epochs instead of iterating over minibatches.
Taking the gradient on a subset of the data introduces not only radomness, which decreases the chances to get stuck in a local minimum, but also has some computational benefits, if the minibatch size are relatively small to the number of datapoints. Common sizes of Minibatches start from [16, 32, 64, 128, 256, 512], depending on the dataset at hand.

Adaptive Moment Estimation (ADAM)

ADAM (https://arxiv.org/abs/1412.6980) is another optimization algorithm, that was introduced in 2015 on the International Conference on Learning Representations (ICLR). It was a huge achievement, by boosting the performance of the optimization problems.
SGD maintains a single learning rate τ for all update steps and thus the learning rate does not change during training. ADAM instead computes adaptive learning rates for different parameters from estimates of first and second moments (*momentum term*) of the gradients.
By keeping a part of the change at the previous timestep, thus giving the optimization a momentum to accelerate the minimization in parameter space directions in which the gradient is not steep, but consistently has a small value steadily in one direction. Each minibatch changes to: [9]

$$\theta^{(t+1)} = \theta^{(t)} - \left[\eta \nabla_\theta c_i(\boldsymbol{x}_i, \theta^{(t-1)}\right] - \tau \nabla_\theta c_i(\boldsymbol{x}_i, \theta^{(t)}),$$

with the momentum parameter η, usually close to 1.0, i.e. $\eta = 0.95$. In general, using past moments of the previously calculated iterations as a guide for the current gradient step to enhance the performance.
ADAM uses a exponentially decaying average of the first and second memoments of the gradient to compute individual adaptive learning rates for each parameter independently. [9]

Activation functions

The previous Section dealt with the Backpropagation and Gradient descent. However, to calculate a proper gradient, we utilize an activation function $f_l(x)$. In order to introduce non-linearities so that the model can also represent complex datasets, employing a non-linear activation function is essential.
It is required that these functions are continous and differentiable in order for the backpropagation to work. [8]
Since the feedforward is just a nested function of the inputs times the activation functions, it can be written as:

$$\hat{y} = \sum_{j=1}^{M} w_{1j}^{L} f_L\left(\sum_{i=1}^{M} w_{ji}^{L-1} f_{L-1}\left(\sum_{k=1}^{M} w_{ik}^{L-2} f_{L-2}\left(\ldots f_1(w_{m1}^{1} x_1 + b_m^1)\ldots\right) + b_k^{L-2}\right) + b_i^{L-1}\right) + b_1^L.$$

The simplest activation function would be the identity transformation $f_I(x) = x$, which is often used for regression networks in the output layer. Common activation functions comprise the *sigmoid, ReLU, tanh, leaky ReLU* and ELU function. The sigmoid function is commonly used as hidden layer activations, or as the output layer for binary classification tasks, since this function squeezes its input values to a range from $[0, \ldots, 1]$:

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-x}}.$$

Next in line is the tangent hyperbolicus, that squeezes the values to a range from $[-1, \ldots, 1]$:

$$f_{tanh}(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Simpler than both of the other activation functions and usually achieving better results are the Rectified Linear Units (ReLU). It consists of a piecewise linear function, when it reaches the origin of a coordinate system:

$$f_{ReLU}(x) = max(0, x).$$

Many variants of ReLU exist, which are not 0 in the negative space, such as the leaky ReLU:

$$f_{leakyReLU}(x, \alpha) = \begin{cases} x & if \quad x \geq 0 \\ \alpha x & if \quad x < 0. \end{cases},$$

and the exponential linear unit (ELU):

$$f_{ELU}(x, \alpha) = \begin{cases} x & if \quad x \geq 0 \\ \alpha(e^x - 1) & if \quad x < 0. \end{cases}.$$

Fig. 2 visualizes each of these activation functions.

Activation Functions
Fig. 2 Different, common Activation functions used in Neural Networks. Retrieved from Hackernoon (https://hackernoon.com/how-to-debug-neural-networks-manual-dc2a200f10f2) on the 8th October, 2019.

Exploding and vanishing gradients

Looking at Fig. 2, one can see that the functions on the left hand-side (Sigmoid, tanh and ReLU) are only constant on a specific interval. For instance ReLU is 0 before the origin and then linearly increasing, tanh and sigmoid are only differentiable in an interval from approximately [-3, 3].
Thus, those activation functions could squeeze the input to 0 beyond the intervals, i.e. a fully saturated neuron with input $z_j >> 1$, or a dead neuron with input $z_j << -1$ will exhibit very small gradients or none at all. Resulting in wasted neurons, since they cannot learn anything. [3] To avoid this problem, it is important to initialize the weights and biases correctly. One method to initialize the weights and biases is the Xavier initialization (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf). Initialize the weights in such a way that the variance remains the same for x and y as we pass thorugh the layer. [10]
Picking the weights from a Gaussian distribution, $\mathcal{N}(0, \frac{1}{N})$, with zero mean and a variance of $1/N$, where N specifies the number of input neurons:

$$var(w_i) = \frac{1}{N}.$$

Dataset

The following work will focus on the *UCI Credit Card dataset (https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients)*, which consists of 24 columns, where 23 of them are the predictor variables p and one refers to the target y. The dataset has 30.000 entries, or individuals respectively, thus resulting in a dataset matrix $X \in \mathbb{R}^{30000 \times 23}$ along with the targets $\boldsymbol{y} = \{y_1, y_2, \ldots, y_{30000}\}$.
This dataset represents the defaults of Taiwanese credit card clients of the early 2000's, based on the aforementioned 23 predictor variables like education, sex, payment history, age along with 19 others (below is a small excerpt plotted of the dataset). During this time there prevailed a cash and credit card debt crisis. An insanely smart decision [IRONY] from Taiwanese banks was to issue cash and credit cards even to unqualified applicants just to increase their market share.
When taking a closer look to the dataset at hand, one can notice that there are only numerical columns, so there are no written words in any of the columns, and every column is filled with a proper value, which is usually never the case for other datasets.
Furthermore, the numeric values vastly range from binary values like *sex (1 = male, 2 = female)* over *age ([21, ..., 79])* until *payment amount ([0, ..., 1.684.259])*. Hence there is a need of normalizing the entire dataset to decrease the variance in our model as well as being suitable for our activation functions (see Chapter "Vanishing and Exploding Gradients").
There are mostly two common techniques: **Min-Max Normalization**, as well as **Standardization**. However, when the minimum and maximum values are not known in the features, it is obviously not possible to apply the first technique, which reads as follows:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}.$$

Equation (42) squeezes all the values in the respective column to a range from [0, ..., 1]. Additionally to the aforementioned problem, considerable outliers will affect this method greatly, where the outlier takes either the value 1 when being the biggest value or 0 when it is super small.
That is the reason why this work makes use of *Standardization*:

$$x' = \frac{x - \bar{x}}{\sigma_x},$$

where $\bar{x}$ is the mean/ average value of the feature vector and σx is its standard deviation.

Additional to the findings above, this dataset is also heavily unbalanced: It has almost by a factor of 5 more samples labeled as 0, which means no default. Only 22.12% (6.636 out of the 30.000) samples are default samples. In order to not have a biased classifier, this work uses *undersampling* as technique of balancing the dataset, i.e. picking randomly the same amount of the minority class (in our case 6.636). Since using *oversampling* would not work due to the complexity and variances in the dataset, there is no way of upsampling the minority class to the size of the majority class.

Moreover, when comparing the documentation of the dataset with the dataset itself, one can identify mismatches in the documentation and the actual dataset. The documentation states that the features *Education* as well as *Marriage* have ranges from $[1, 2, 3, 4]$ and $[1, 2, 3]$, respectively.
However, the dataset yields values of $0, 5$ and 6 as well. Meaning that these values are not documented and probably have been wrongly labeled. One common way is to declare these non-documented values as *NaN* values and deleting these entries. Unfortunately, we would only keep 13.5% of the initial dataset (4.061 out of 30.000). Obviously this is too much data, which is being lost when applying this method. I decided to declare all

the non-documented values as a sepearate class. Hence, *Marriage* is not going to change since there was only one different, non-documented value. *Education* however, is getting another label, 5, representing the new label *unknown*.

3.) Code and Implementation

In [1]:
```
# import the lovely libraries that saved my ass here
from sklearn.metrics import roc_curve, accuracy_score, roc_auc_score, r2_score
from sklearn.linear_model import LogisticRegression as LR
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

import matplotlib.pyplot as plt
from tabulate import tabulate
import pandas as pd
import numpy as np
import os

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

In [2]:
```
# set up a plot directory
if os.path.isdir(os.path.join(os.getcwd(), 'plots')):
    plot_dir = os.path.join(os.getcwd(), 'plots')
else :
    os.mkdir(os.path.join(os.getcwd(), 'plots'))
    plot_dir = os.path.join(os.getcwd(), 'plots')
```

In [3]:
```
# import my libraries
from LogisticRegression import get_data, LogisticRegression
from neuralNet import NeuralNetwork

# read the data
filename = "default of credit card clients.xls"
datapath = os.getcwd() + '\\data'
filePath = os.path.join(datapath, filename)
X, y = get_data(filePath, standardized=False, normalized=False)
stdX, stdY = get_data(filePath, standardized=True, normalized=False)
normX, normY = get_data(filePath, standardized=False, normalized=True)
print(X.shape)
```

```
(30000, 23)
```

In [4]:
```
# Check for NaN values and target distribution
print(X.isnull().values.any()) # there are no NaN values and only numerical va
lues - AMAZING
X.head(3)
```

```
False
```

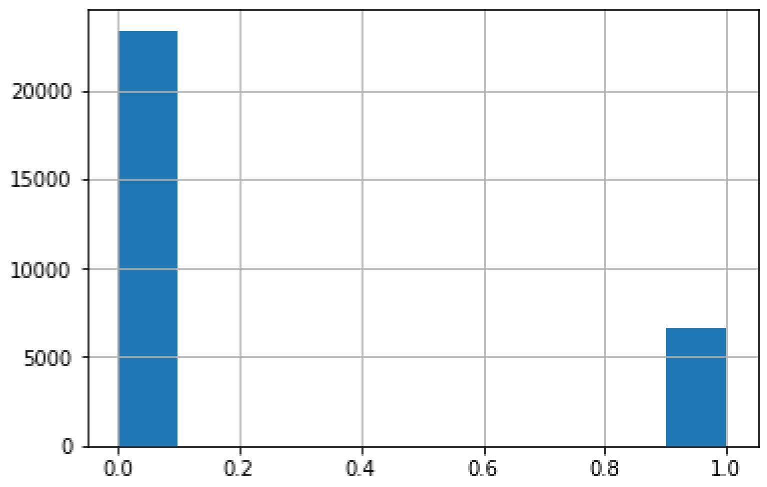
Out[4]:

	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_
0	20000	2	2	1	24	2	2	-1	-1	-2
1	120000	2	2	2	26	-1	2	0	0	0
2	90000	2	2	2	34	0	0	0	0	0

3 rows × 23 columns

In [5]:
```
y.hist()
print("Label 0 has {} recordings\nLabel 1 has {} recordings\nPercentage: {:.2
f}%".format(y.value_counts()[0], y.value_counts()[1], (100* y.value_counts()[1
] / ( y.value_counts()[0] + y.value_counts()[1]))))
None
```

```
Label 0 has 23364 recordings
Label 1 has 6636 recordings
Percentage: 22.12%
```



As you can see in the plot above, the dataset is heavily unbalanced.
Label 1 accounts for only 22% of the entire dataset.

In [6]:
```
print(tabulate( [ [name, mini, maxi, stdmini, stdmaxi, normmin, normmax] for n
ame, mini, maxi, stdmini, stdmaxi, normmin, normmax in zip(X.columns, X.min(ax
is=0), X.max(axis=0), stdX.min(axis=0), stdX.max(axis=0), normX.min(axis=0), n
ormX.max(axis=0))], headers=['Feature Name', 'Min Value', 'Max Value', 'Standa
rdized Min', 'Standardized Max', 'Normalized Min', 'Normalized Max']))
```

```
Feature Name      Min Value    Max Value    Standardized Min    Standardized
Max    Normalized Min    Normalized Max
--------------  -----------  -----------  ------------------  ---------------
---  ----------------  ----------------
LIMIT_BAL             10000      1000000            -1.21377             6.416
42                   0                 1
SEX                       1            2            -1.2343              0.810
147                  0                 1
EDUCATION                 0            6            -2.3447              5.246
88                   0                 1
MARRIAGE                  0            3            -2.9731              2.774
36                   0                 1
AGE                      21           79            -1.57145             4.720
65                   0                 1
PAY_0                    -2            8            -1.76481             7.133
55                   0                 1
PAY_2                    -2            8            -1.55885             6.794
07                   0                 1
PAY_3                    -2            8            -1.53217             6.822
98                   0                 1
PAY_4                    -2            8            -1.52192             7.031
39                   0                 1
PAY_5                    -2            8            -1.53002             7.294
65                   0                 1
PAY_6                    -2            8            -1.48602             7.209
73                   0                 1
BILL_AMT1           -165580       964511            -2.94426            12.402
8                    0                 1
BILL_AMT2            -69777       983931            -1.67135            13.133
4                    0                 1
BILL_AMT3           -157264      1664089            -2.94562            23.317
8                    0                 1
BILL_AMT4           -170000       891586            -3.31499            13.186
5                    0                 1
BILL_AMT5            -81334       927171            -2.00084            14.587
2                    0                 1
BILL_AMT6           -339603       961664            -6.35514            15.495
0                  1
PAY_AMT1                  0       873552            -0.341936           52.398
3                    0                 1
PAY_AMT2                  0      1684259            -0.256985           72.841
8                    0                 1
PAY_AMT3                  0       896040            -0.296796           50.594
4                    0                 1
PAY_AMT4                  0       621000            -0.308057           39.331
5                    0                 1
PAY_AMT5                  0       426529            -0.314131           27.603
2                    0                 1
PAY_AMT6                  0       528666            -0.293377           29.444
6                    0                 1
```

In [14]:
```
# initialize paramters
epochs = 100 # after 150 runs already at maximum, so let's keep it rather shor
t for time reasons
batches = 64

# run Logistic Regression on it and my neural network implementation
print("\nNow the Original     Dataset:")
logistic = LogisticRegression(X, y, test_size=0.2)
logistic.optimize(batch_size=batches, regularization='l2', epochs=epochs, lamd
a=0.001, plot_training=False)

print("\nNow the Standardized Dataset:")
stdLogistic = LogisticRegression(stdX, stdY, test_size=0.2)
stdLogistic.optimize(batch_size=batches, regularization='l2', epochs=epochs, l
amda=0.001, plot_training=False)

print("\nNow the Normalized   Dataset:")
normLogistic = LogisticRegression(normX, normY, test_size=0.2)
normLogistic.optimize(batch_size=batches, regularization='l2', epochs=epochs,
lamda=0.001, plot_training=False)
```

```
Now the Original     Dataset:
Scikit Accuracy on Test Set: 0.779, lambda: 0.001
My     Accuracy on Test Set: 0.779, lambda: 0.001

Now the Standardized Dataset:
Scikit Accuracy on Test Set: 0.806, lambda: 0.001
My     Accuracy on Test Set: 0.524, lambda: 0.001

Now the Normalized   Dataset:
Scikit Accuracy on Test Set: 0.779, lambda: 0.001
My     Accuracy on Test Set: 0.779, lambda: 0.001
```

In [8]:
```
# get the best regularization stregth on raw data (non-standardized and non-no
rmalized)
lambdas = np.logspace(-4,4,9) # array([0.0001, 0.001, 0.01, 0.1, 1., 10., 10
0., 1000., 10000.]
test_acc = []
train_acc = []
for lamda in lambdas:
    logistic.optimize(batch_size=batches, regularization='l2', epochs=epochs,
lamda=lamda, plot_training=False)
    test_acc.append(logistic.test_accuracy)
    train_acc.append(logistic.train_accuracy)
```

```
Scikit Accuracy on Test Set: 0.779, lambda: 0.0001
My     Accuracy on Test Set: 0.224, lambda: 0.0001
Scikit Accuracy on Test Set: 0.779, lambda: 0.001
My     Accuracy on Test Set: 0.779, lambda: 0.001
Scikit Accuracy on Test Set: 0.779, lambda: 0.01
My     Accuracy on Test Set: 0.779, lambda: 0.01
Scikit Accuracy on Test Set: 0.779, lambda: 0.1
My     Accuracy on Test Set: 0.779, lambda: 0.1
Scikit Accuracy on Test Set: 0.779, lambda: 1.0
My     Accuracy on Test Set: 0.617, lambda: 1.0
Scikit Accuracy on Test Set: 0.779, lambda: 10.0
My     Accuracy on Test Set: 0.779, lambda: 10.0
Scikit Accuracy on Test Set: 0.779, lambda: 100.0
My     Accuracy on Test Set: 0.779, lambda: 100.0
Scikit Accuracy on Test Set: 0.779, lambda: 1000.0
My     Accuracy on Test Set: 0.779, lambda: 1000.0
Scikit Accuracy on Test Set: 0.779, lambda: 10000.0
My     Accuracy on Test Set: 0.779, lambda: 10000.0
```

Note that the Scikit Learn Logistic Regression also utilized the different regularization Strengths values. They mostly seem to match up, thus Logistic Regression is not too much affected of λ parametrization. At least not when making use of the L^2 regularization in the cost function.

In [9]:
```
print(tabulate( [ [lmds, test, train] for lmds, test, train in zip(lambdas, te
st_acc[len(test_acc) - 1], train_acc[len(train_acc) - 1])], headers=['λ Value
s', 'Test Accuracy', 'Train Accuracy']))
```

```
  λ Values    Test Accuracy    Train Accuracy
----------  ---------------  ----------------
    0.0001         0.769333          0.770583
    0.001          0.221333          0.222083
    0.01           0.779             0.77875
    0.1            0.571833          0.522333
    1              0.7785            0.778
   10              0.779167          0.778708
  100              0.779             0.778583
 1000              0.779667          0.778583
10000              0.462333          0.43875
```

In [12]:
```python
# plot the lovely accuracies
txt = "Fig. 3 Effects of regularization parameter λ on Logistic Regression."
fig = plt.figure(figsize=(9,7))
plt.plot(lambdas, np.mean(np.array(test_acc), axis=1), label="Test Accuracy")
plt.plot(lambdas, np.mean(np.array(train_acc), axis=1), label="Train Accuracy"
)
plt.title("Accuracies for Logistic Regression for different λ - values")
plt.ylabel("Accuracy Score")
plt.xlabel("λ parametrization values")
plt.xticks(lambdas)
plt.xscale('log')
plt.legend()
fig.text(.1, 0,txt)
plt.savefig(os.path.join(plot_dir, 'Logistic Regression Lambdas.png'), transpa
rent=True, bbox_inches='tight')
plt.show()
```

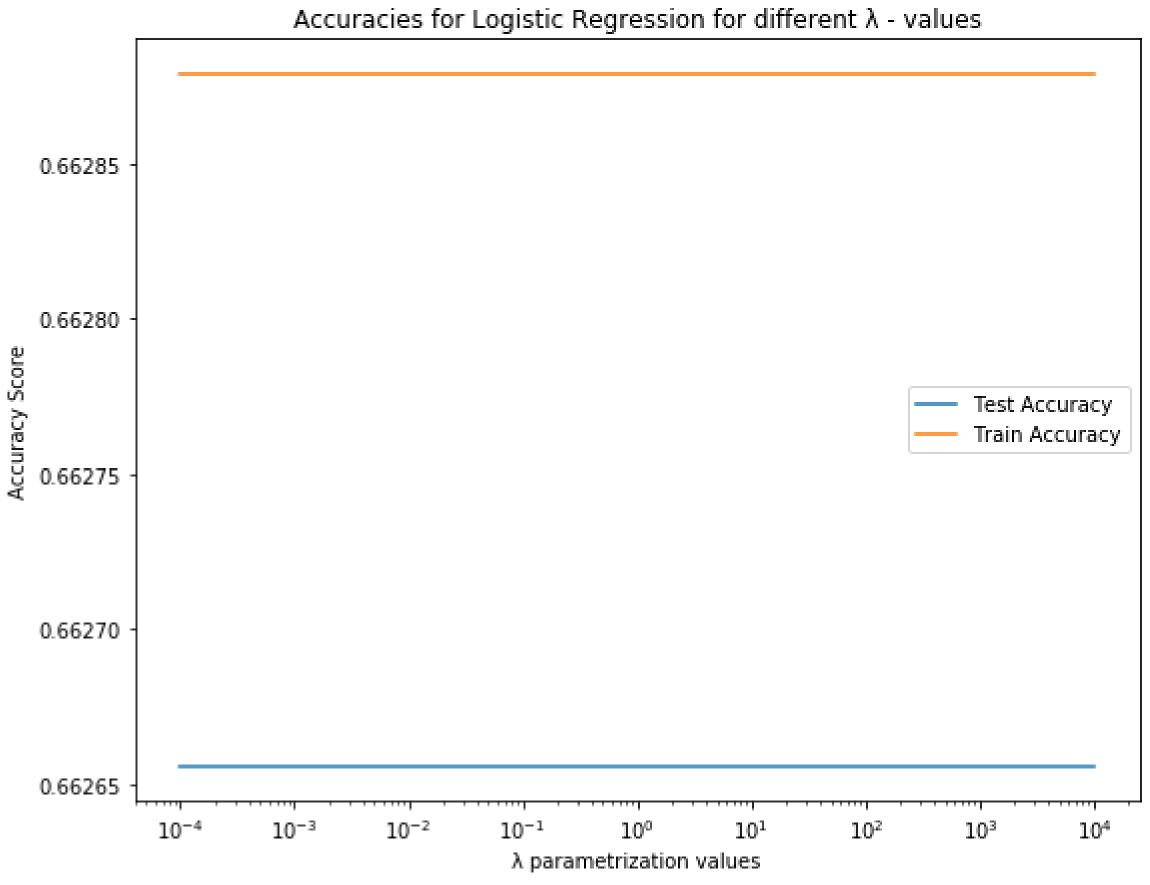


Fig. 3 Effects of regularization parameter λ on Logistic Regression.

As Fig. 3 shows the regularization parameter λ is not making any difference on neither the Scikit Learn Logistic Regression model, nor on my implementation.
They are constant for a wide range of λ and thus insensitive to this parameter change. Keep in mind that this could also be due to this given dataset. That the bias-variance tradeoff are not such a big problem for the default credit card dataset and thus the regularization parameter does not play a role.
We can conclude that Logistic Regression is not sensitive for λ for this dataset und thus makes it even more robust and stable.

Receiver Operating Characteristic Curve (ROC)

In order to compare the results from one algorithm to another, this work utilizes the ROC curve. A ROC curve is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: the *True Positive Rate (TPR)*, and the *False Positive Rate (FPR)*, which are defined as follows:

$$TPR = \frac{TP}{TP + FN}, \quad \text{where TP stands for True Positives and FN for False Negatives,}$$

$$FPR = \frac{FP}{FP + TN}, \quad \text{where FP are False Positives and TN are True Negatives.}$$

.

A ROC curve plots TPR (y-axis) vs. FPR (x-axis) at different classification thresholds. Lowering the classification threshold, classifies more items as positive, thus increasing both False Positives and True Positives.
Moreover, there is an addition to the ROC curve, calles *Area under ROC Curve (AUC)*, which represents the area underneath the 2D ROC curve (can be thought of as the integral of the ROC curve).
AUC provides an aggregate measure of performance across all possible classification thresholds.
AUC is a commonly used metrics to evaluate a classification algorithm due to its scale - and classification threshold invariance.
It measures how well predictions are ranked rather than their absolut values (scale-invariant) and the quality of the model's predictions irrespective of what classification threshold was chosen (classification-threshold-invariant).
The bigger AUC, the better the model.

In [34]:
```
## Setup new Training data and Test data if wanted, but then keep in mind that
we can't compare LR to NN
new_data = False

## Get new data potentially otherwise use the versions from the Log Reg
if new_data:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r
andom_state=2912)
    X_train_std,  X_test_std,  y_train_std,  y_test_std  = train_test_split(st
dX, stdY, test_size=0.2, random_state=2912)
    X_train_norm, X_test_norm, y_train_norm, y_test_norm = train_test_split(no
rmX, normY, test_size=0.2, random_state=2912)
else:
    X_train, X_test, y_train, y_test = logistic.X_train, logistic.X_test, logi
stic.y_train, logistic.y_test
    X_train_std,  X_test_std,  y_train_std,  y_test_std  = stdLogistic.X_train
, stdLogistic.X_test, stdLogistic.y_train, stdLogistic.y_test
    X_train_norm, X_test_norm, y_train_norm, y_test_norm = normLogistic.X_trai
n, normLogistic.X_test, normLogistic.y_train, normLogistic.y_test
```

```
In [38]: ## Claculate and Plot the ROC curve
         model = LR(solver='lbfgs').fit(X_train, y_train)
         probs = model.predict_proba(X_test)
         probs = probs[:, 1]
         FPR, TPR, thresholds = roc_curve(y_test, probs)

         ## ROC Function is all the way down - you have to run this code first

         plot_roc_curve(FPR, TPR, 4, " ROC curve of Logistic Regression on the original
         dataset")
         print("AUC = {:.3f}".format(roc_auc_score(y_test, probs)))
         #print("My R2 Score on Data: {}".format(logistic.R2))
```

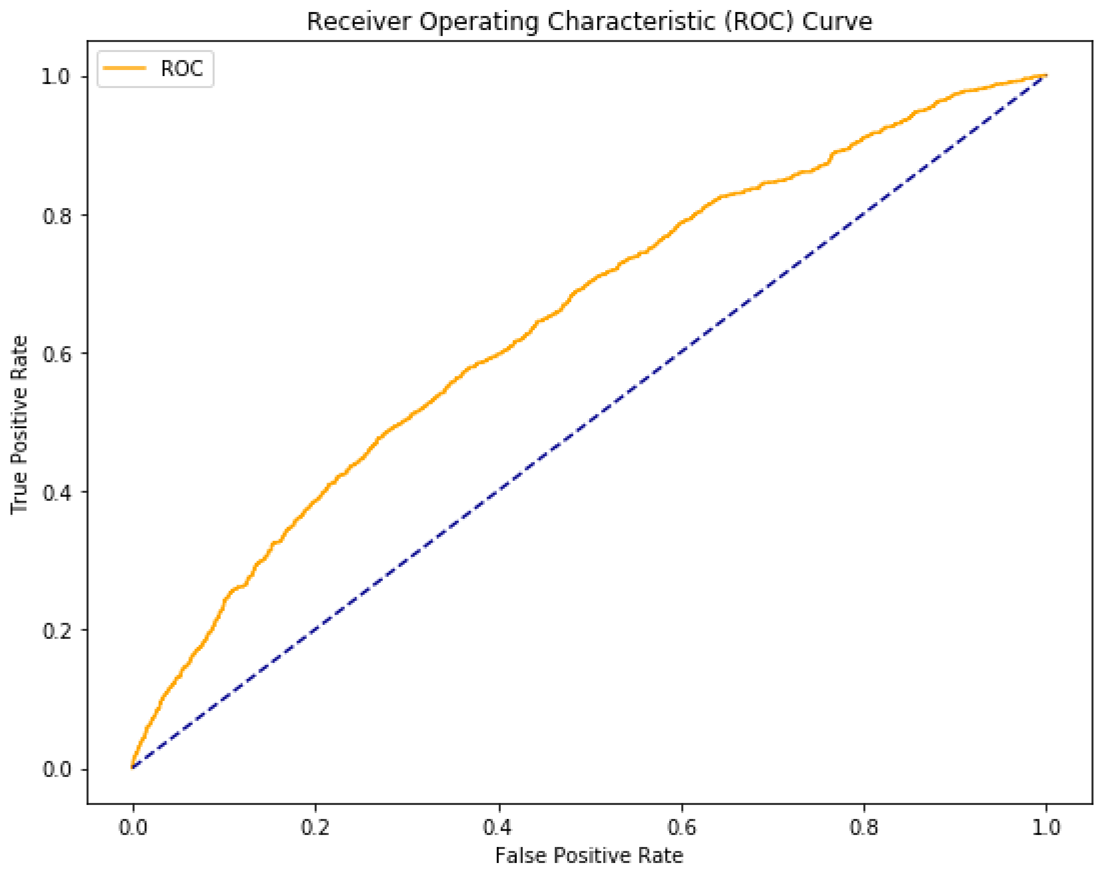


Fig. 4 ROC curve of Logistic Regression on the original dataset

```
AUC = 0.645
```

These results seems to be similiar to the one in the paper of [Yeh, et al. (https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf)](https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf) [4]
However, since they did not explicity mention how they transformed their data, there might be some deviations compared to my results. Moreover, they also do not mention how big their split between Training and Validation set is, which makes it really hard to reproduce their results. Considering a 20% split, there might still be other samples in the training and testing data, which in turn could also change the models parameters.
One could say that we got similiar results, but obtaining the exact same results will most probably not going to happen, since they didn't state their data preparation and also didn't mention their train test split size.

Time to get our hands dirty with the Neural Network now. The following section will train a Neural Network to fit the Credit Card data set and classifies it. Afterwards a comparison among Logistic Regression and Neural Network will take pace.

In [14]:
```
%%time
# One run with one classifier takes about 1m 20s
clf      = MLPClassifier(hidden_layer_sizes=[128, 128, 128, 64], learning_rate
_init=0.001).fit(X_train, y_train)
clf_std  = MLPClassifier(hidden_layer_sizes=[128, 128, 128, 64], learning_rate
_init=0.001).fit(X_train_std, y_train_std)
clf_norm = MLPClassifier(hidden_layer_sizes=[128, 128, 128, 64], learning_rate
_init=0.001).fit(X_train_norm, y_train_norm)
print("SKLearn Accuracy on Original   Dataset: {}".format(clf.score(X_test, y_
test)))
print("SKLearn Accuracy on Stdized    Dataset: {}".format(clf_std.score(X_test
_std, y_test_std)))
print("SKLearn Accuracy on Normalized Dataset: {}".format(clf_norm.score(X_tes
t_norm, y_test_norm)))
```

```
SKLearn Accuracy on Original   Dataset: 0.7461666666666666
SKLearn Accuracy on Stdized    Dataset: 0.7573333333333333
SKLearn Accuracy on Normalized Dataset: 0.8176666666666667
Wall time: 4min 52s
```

In [33]:
```
## Claculate and Plot the ROC curve
probs = clf_norm.predict_proba(X_test_norm)
probs = probs[:, 1]
FPR, TPR, thresholds = roc_curve(y_test_norm, probs)
plot_roc_curve(FPR, TPR, 5, " ROC curve of MLP on normalized data")
print("AUC = {:.3f}".format(roc_auc_score(y_test, probs)))
#print("My R2 Score on Neural Net: {}".format(nn.R2)) # need to run Neural net
2 cells below first
```

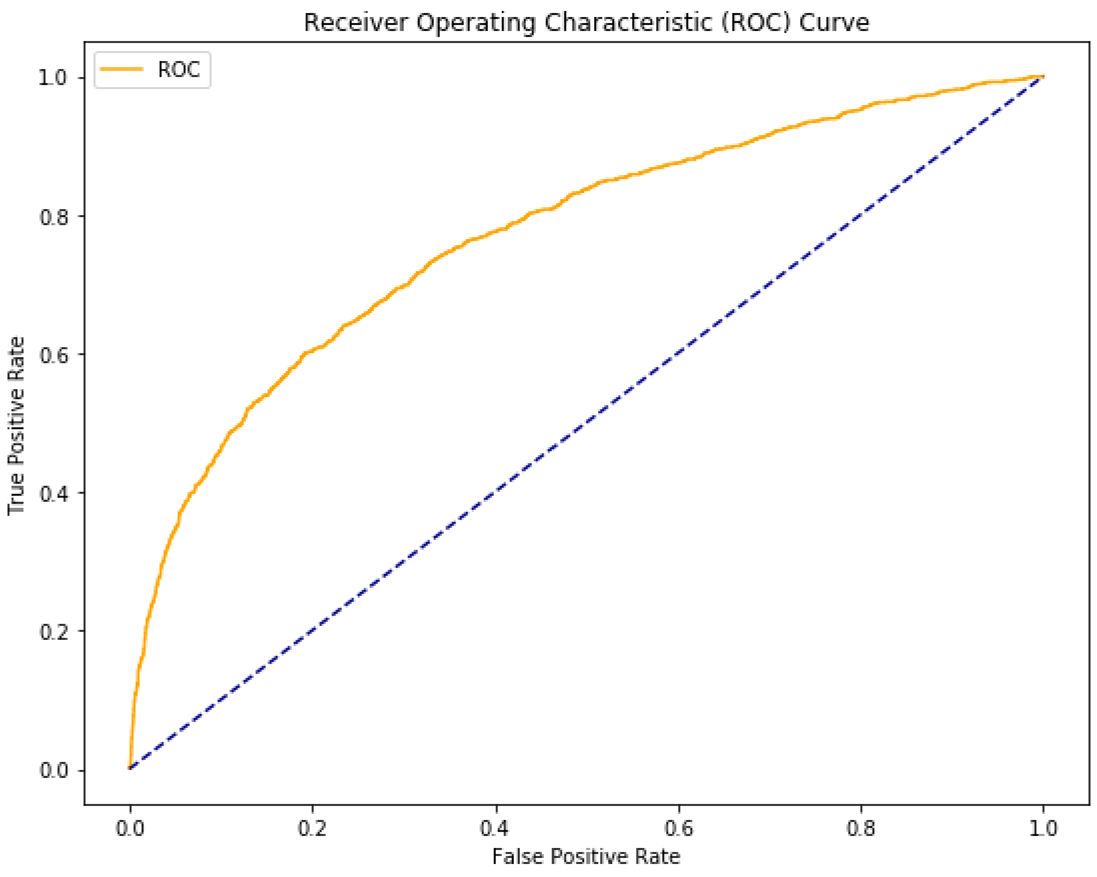


Fig. 5 ROC curve of MLP on normalized data

```
AUC = 0.770
```

In [16]:
```
## Init the neural networks Neural Network
number_of_classes = 2
epochs            = 70 # usually converged at around 100, but for the sake of
 time please allow me to run less epochs
batch_size        = 64
learning_rate     = 0.0001
reg_strength      = 0.0 # lambda parameter for the cost function
```

In [17]:
```
%%time
nn = NeuralNetwork(inputs             = X_train.shape[1],
                   outputs            = number_of_classes,
                   cost_function_name = 'cross_entropy')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'relu', neurons = 128)
nn.addLayer(activations = 'relu', neurons = 128)
nn.addLayer(activations = 'relu', neurons = 64)
nn.addLayer(activations = 'softmax', neurons = number_of_classes, output = Tru
e)

nn.train_network(X_train.T, y_train.T,
       batch_size           = batch_size,
       learning_rate        = learning_rate,
       epochs               = epochs,
       val_size             = 0.2,
       val_stepwidth        = 10,
       optimizer            = 'adam',
       lmbda                = reg_strength)

print("\nNeural Network Accuracy on Original Test data: {}\n".format(nn.accura
cy(X_test, y_test)))
```

```
Adding input layer with 128 neurons, using relu
Adding layer with 128 neurons using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779

Wall time: 4min 44s
```

In [19]:
```
%%time

nn = NeuralNetwork(inputs             = X_train.shape[1],
                   outputs            = number_of_classes,
                   cost_function_name = 'cross_entropy')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'leakyrelu', neurons = 256)
nn.addLayer(activations = 'leakyrelu', neurons = 512)
nn.addLayer(activations = 'leakyrelu', neurons = 256)
nn.addLayer(activations = 'leakyrelu', neurons = 128)
nn.addLayer(activations = 'leakyrelu', neurons = 128)
nn.addLayer(activations = 'leakyrelu', neurons = 64)
nn.addLayer(activations = 'softmax', neurons = number_of_classes, output = Tru
e)

nn.train_network(X_train.T, y_train.T,
       batch_size           = batch_size,
       learning_rate        = learning_rate,
       epochs               = epochs + 40,
       val_size             = 0.2,
       val_stepwidth        = 10,
       optimizer            = 'adam',
       lmbda                = reg_strength)

print("\nNeural Network Accuracy on Original Test data: {}\n".format(nn.accura
cy(X_test, y_test)))
```

```
Adding input layer with 256 neurons, using leakyrelu
Adding layer with 512 neurons using leakyrelu
Adding layer with 256 neurons using leakyrelu
Adding layer with 128 neurons using leakyrelu
Adding layer with 128 neurons using leakyrelu
Adding layer with 64 neurons using leakyrelu
Adding output layer with 2 outputs and softmax
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779

Wall time: 39min 1s
```

The above two cell shows that there was no difference in the accuracy, when adding more layers and also training longer. However, it took waaaaay longer than the previous version.
Here you see the computaional complexity of such models. Since one could not really anticipate the outcomes, when changing some parameters, you have to wait for quite a significant time until there will be a proper result. Now when changing even more parameters, you can say that there will be a long time to evaluate all of those parameters.
For the sake of my (precious) time and also the electricity bill (P.S. I am a super poor student), I rather prefer not blowing up the network too much and just focus on the effects of parametrization of the neural networks.

In [53]:
```
## STANDARDIZED VERSION
nn = NeuralNetwork(inputs             = X_train_std.shape[1],
                   outputs            = number_of_classes,
                   cost_function_name = 'cross_entropy')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'relu', neurons = 128)
nn.addLayer(activations = 'relu', neurons = 64)
nn.addLayer(activations = 'softmax', neurons = number_of_classes, output = Tru
e)

nn.train_network(X_train_std.T, y_train_std.T,
       batch_size          = batch_size,
       learning_rate       = learning_rate,
       epochs              = epochs,
       val_size            = 0.2,
       val_stepwidth       = 10,
       optimizer           = 'adam',
       lmbda               = reg_strength)

print("\nNeural Network Accuracy on Standardized Test data: {}\n".format(nn.ac
curacy(X_test_std, y_test_std)))
```

```
Adding input layer with 128 neurons, using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax

Neural Network Accuracy on Standardized Test data: 0.5245
```

In [52]:
```
## NORMALIZED DATASET
nn = NeuralNetwork(inputs             = X_train_norm.shape[1],
                   outputs            = number_of_classes,
                   cost_function_name = 'cross_entropy')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'relu', neurons = 128)
nn.addLayer(activations = 'relu', neurons = 64)
nn.addLayer(activations = 'softmax', neurons = number_of_classes, output = Tru
e)

nn.train_network(X_train_norm.T, y_train_norm.T,
       batch_size          = batch_size,
       learning_rate       = learning_rate,
       epochs              = epochs,
       val_size            = 0.2,
       val_stepwidth       = 10,
       optimizer           = 'adam',
       lmbda               = reg_strength)

print("\nNeural Network Accuracy on Normalized Test data: {}\n".format(nn.accu
racy(X_test_norm, y_test_norm)))
```

```
Adding input layer with 128 neurons, using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax

Neural Network Accuracy on Normalized Test data: 0.48
```

In [24]:
```
## NOW TEST THE DIFFERENT LEARNING RATES
learning_rates_to_test = [0.000001, 0.001, 1, 1000, 1000000]
epochs = 50 # lowering the epochs to speed up that stuff here
accuracies = []

nn = NeuralNetwork(inputs             = X_train.shape[1],
                   outputs            = number_of_classes,
                   cost_function_name = 'cross_entropy')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'relu', neurons = 128)
nn.addLayer(activations = 'relu', neurons = 128)
nn.addLayer(activations = 'relu', neurons = 64)
nn.addLayer(activations = 'softmax', neurons = number_of_classes, output = Tru
e)

for tau in learning_rates_to_test:
    ## Train the network
    nn.train_network(X_train.T, y_train.T,
           batch_size           = batch_size,
           learning_rate        = tau,
           epochs               = epochs,
           val_size             = 0.2,
           val_stepwidth        = 10,
           optimizer            = 'adam',
           lmbda                = reg_strength)

    acc    = nn.accuracy(X_test, y_test)

    print("\nNeural Network Accuracy on Original Test data: {}, tau: {}".forma
t(acc, tau))
    accuracies.append(acc)
```

```
Adding input layer with 128 neurons, using relu
Adding layer with 128 neurons using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779, tau: 1e-06
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779, tau: 0.001
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779, tau: 1
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779, tau: 1000
There was always the label: 0 predicted

Neural Network Accuracy on Original Test data: 0.779, tau: 1000000
```

In [26]:
```
# plot the lovely accuracies
txt = "Fig. 6 Effects of learning rate on Neural Network using SGD as optimize
r."
fig = plt.figure(figsize=(9,7))
plt.plot(learning_rates_to_test, accuracies, label="Test Accuracy")
plt.title("Accuracies for Neural Network for different learning rates")
plt.ylabel("Accuracy Score")
plt.xlabel("λ parametrization values")
plt.xticks(learning_rates_to_test)
plt.xscale('log')
plt.legend()
fig.text(.1, 0,txt)
plt.savefig(os.path.join(plot_dir, 'Neural Network learning rate.png'), transp
arent=True, bbox_inches='tight')
plt.show()
```

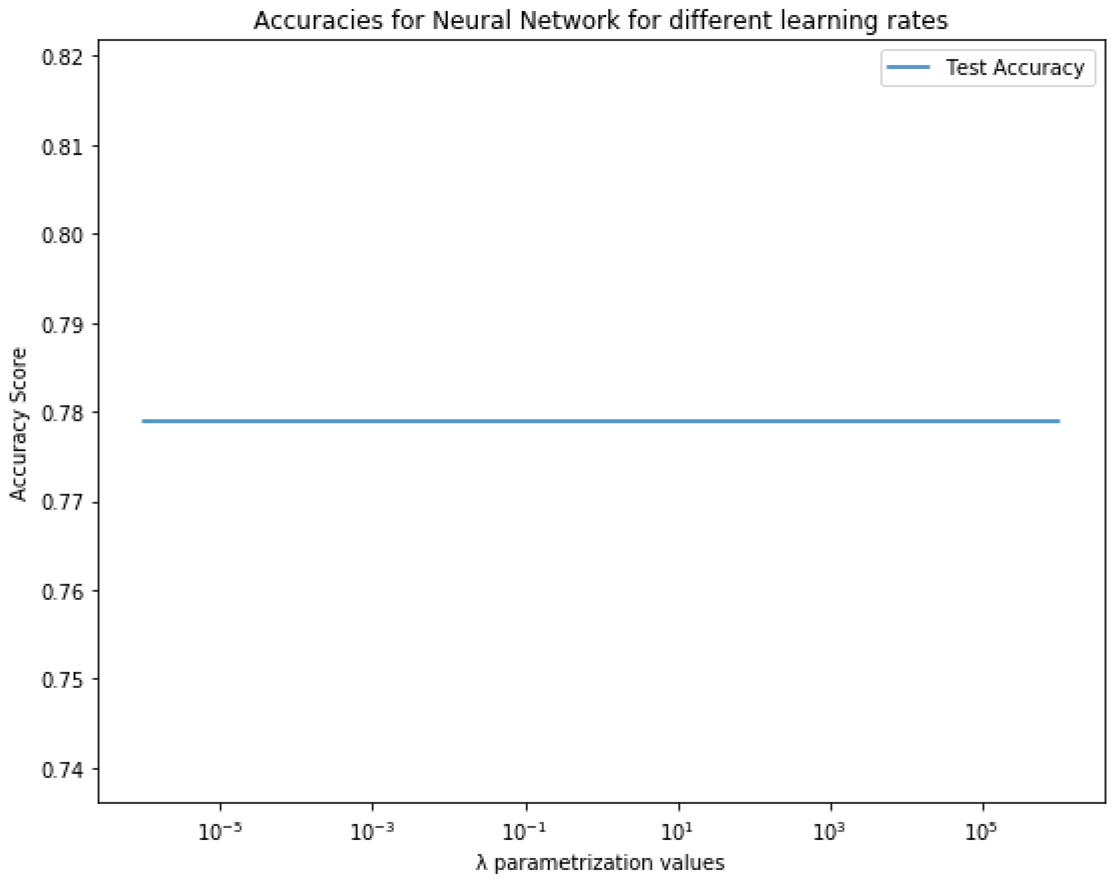


Fig. 6 Effects of learning rate on Neural Network using SGD as optimizer.

This should not be the case here.
Usually the learning rate is a crucial factor for convergence of the algorithm. In the Analysis part below there are some words about the learning rate parametrization and also the effects that should occur in those tests.

In [27]:
```
# just going with super extreme values to show the effects
scores = []
epochs = 60 # reduce the computational amount and also time, otherwise I will
 not finish in time
lambdas_to_test = [0.000001, 0.001, 100, 100000]

for lmda in lambdas_to_test:

    nn = NeuralNetwork(inputs             = X_train_std.shape[1],
                       outputs            = number_of_classes,
                       cost_function_name = 'cross_entropy')

    ## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neuron
s as output layer (binary classification)
    ## All the layers are activated with the ReLU activation function, which m
itigates the vanishing gradient problem
    nn.addLayer(activations = 'relu', neurons = 128)
    nn.addLayer(activations = 'relu', neurons = 128)
    nn.addLayer(activations = 'relu', neurons = 64)
    nn.addLayer(activations = 'softmax', neurons = number_of_classes, output =
True)

    nn.train_network(X_train_std.T, y_train_std.T,
           batch_size          = batch_size,
           learning_rate       = learning_rate,
           epochs              = epochs,
           val_size            = 0.2,
           val_stepwidth       = 10,
           optimizer           = 'adam',
           lmbda               = lmda)

    acc    = nn.accuracy(X_test_std, y_test_std)
    scores.append(acc)
    print("\nNeural Network Accuracy on Standardized Test data: {}, lambda: {}
".format(acc, lmda))
```

```
Adding input layer with 128 neurons, using relu
Adding layer with 128 neurons using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax
There was always the label: 0 predicted

Neural Network Accuracy on Standardized Test data: 0.779, lambda: 1e-06
Adding input layer with 128 neurons, using relu
Adding layer with 128 neurons using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax

Neural Network Accuracy on Standardized Test data: 0.5053333333333333, lambd
a: 0.001
Adding input layer with 128 neurons, using relu
Adding layer with 128 neurons using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax
There was always the label: 0 predicted

Neural Network Accuracy on Standardized Test data: 0.779, lambda: 100
Adding input layer with 128 neurons, using relu
Adding layer with 128 neurons using relu
Adding layer with 64 neurons using relu
Adding output layer with 2 outputs and softmax
There was always the label: 1 predicted

Neural Network Accuracy on Standardized Test data: 0.221, lambda: 100000
```

In [28]:
```
# plot the lovely accuracies
txt = "Fig. 7 Effects of regularization parameter λ on Neural Network using SG
D as optimizer."
fig = plt.figure(figsize=(9,7))
plt.plot(lambdas_to_test, scores, label="Test Accuracy")
plt.title("Accuracies for Neural Network for different λ - values")
plt.ylabel("Accuracy Score")
plt.xlabel("λ parametrization values")
plt.xticks(lambdas_to_test)
plt.xscale('log')
plt.legend()
fig.text(.1, 0,txt)
plt.savefig(os.path.join(plot_dir, 'Neural Network Lambdas.png'), transparent=
True, bbox_inches='tight')
plt.show()
```

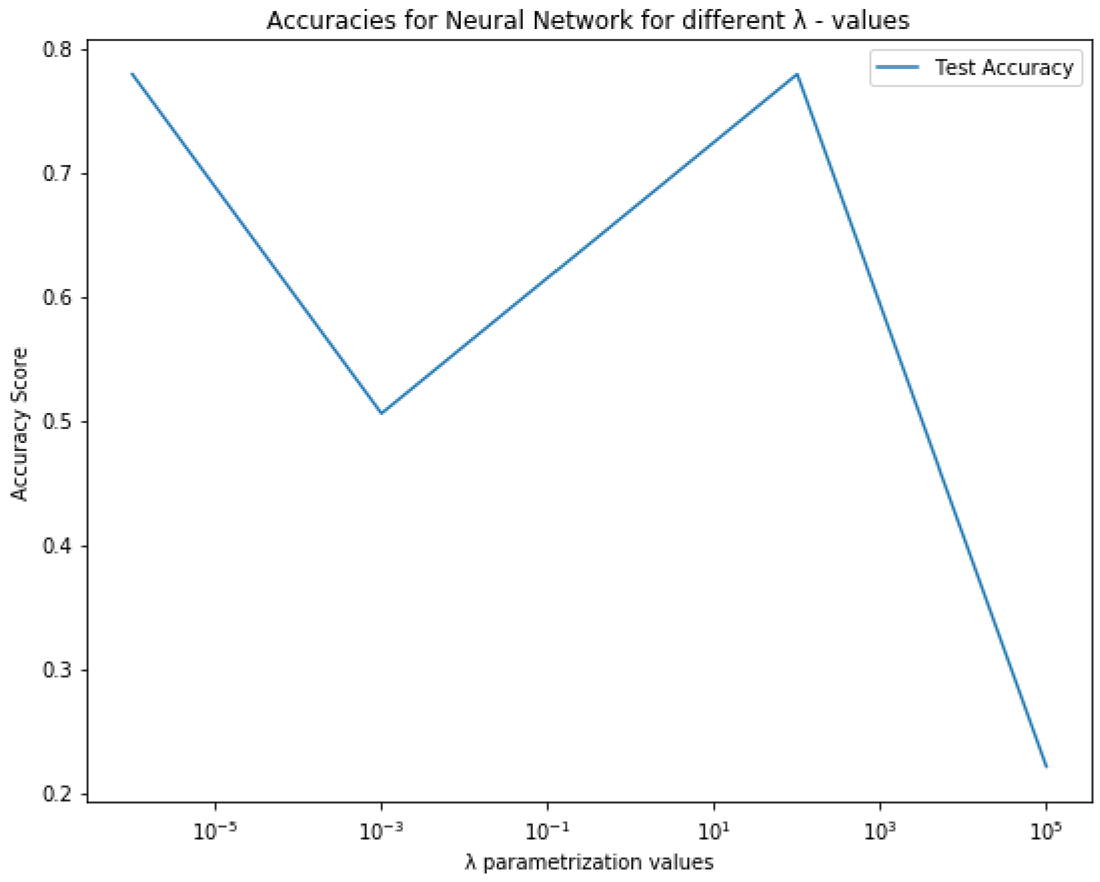


Fig. 7 Effects of regularization parameter λ on Neural Network using SGD as optimizer.

In [37]:
```
%%time
nn = NeuralNetwork(inputs             = X_train_std.shape[1],
                   outputs            = number_of_classes,
                   cost_function_name = 'cross_entropy')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'tanh', neurons = 128)
nn.addLayer(activations = 'tanh', neurons = 128)
nn.addLayer(activations = 'tanh', neurons = 64)
nn.addLayer(activations = 'softmax', neurons = number_of_classes, output = Tru
e)

nn.train_network(X_train_std.T, y_train_std.T,
       batch_size          = batch_size,
       learning_rate       = learning_rate,
       epochs              = epochs,
       val_size            = 0.2,
       val_stepwidth       = 10,
       optimizer           = 'adam',
       lmbda               = reg_strength)

print("\nNeural Network Accuracy on Original Test data: {}\n".format(nn.accura
cy(X_test_std, y_test_std)))
```

```
Adding input layer with 128 neurons, using tanh
Adding layer with 128 neurons using tanh
Adding layer with 64 neurons using tanh
Adding output layer with 2 outputs and softmax

Neural Network Accuracy on Original Test data: 0.20933333333333334

Wall time: 4min 34s
```

Analysis of the classification task

One thing that straight catches the eye is how sensitive the neural network is to small parameter changes. All the parameters that have been changed in the above cells are called *hyperparameters*.
Those are all the parameters that determine the succesand performance of the neural network like not only learning rate, regularization strength, activation functions, but also batch size, number of epochs and so on, which have not greatly been modified above due to time restrictions.
Furthermore, looking across all the different datasets one can see that the data preparation also takes a (big) stake for the model's performance, depending on which activation functions are being used (some could end up in vanishing or exploding gradients).

Unfortunately, Fig. 6, which shows the effects of a changing leraning rates τ did not end up being the desired end result. Usually τ is very critical for the success of learning and also convergence.
If the learning rate τ is too small, it takes years until it converges (if enough epochs are set). If the learning rate τ is too high, the algotihm can't converge at all, since it is only bouncing around and not resulting in a (global) optimum. [8]
Fig. 7 displays the effects of the regularization strength λ. Whe λ is too high, the algorithm is penalzing too much ending up in a bad model and thus bad accuracy. Obviously, if λ is too low it is like not penalzing at all and thus also getting a bad model with a bad accuracy score, if there are inconsistencies in the dataset, where regularization would be needed.

Moving on to the last cell, where only the activation function was changed to a *tanh* activation instead of *ReLU*. The accuracy experienced a massive drop of almost 57%. Coming from almost 78% for a *ReLU* network with the same shape and size to 21%. Due to vanishing and exploding gradients, the network ends up not learning anything.

Those are just some few out of many parameters in a neural networks to tune. There are multiple more, like batch size, number of epochs, number of hidden layers, number of neurons, choice of optimizer and so on. Hence, one can conclude that neural networks are really sensitive to parameter changes along with a tedious search for optimal hyperparameters.

When considering Logistic Regression in the beginning of this work, there were similiar results achieved without having to tune some critical parameters.
Moreover, Logistic Regression's accuracy basically stayed the same even when changing its regularization parameter λ, which makes it even more robust compared to the sensitive neural networks.
Due to its simplicity in implementation, faster convergence and no tedious hyperparameter search Logistic Regression should prefered over Neural Networks for this dataset.

However, there is to mention that when properly tuning the Neural Network hyperparameters with Bayesian Analysis [3], it is possible to further boost the Neural Network's accuracy, however this comes to the cost of a more complex implementation as well as computational power and time (= both of which I do not have).

Lastly, when comparing the results to Yeh et al. [4] it is great to see that they achieved similiar results. Since they do not state, which kind of data preprocessing they utilized for their work and also did not clarify their train and test data split, some deviations are naturally, especially when considering that they most probably had different samples compared to mine in their test dataset even if they utilized a 20% split.
Nevertheless, the results are quite similiar with the same conclusion that Neural Networks are in terms of accuracy slightly better but not with respect to the implementation and searching complexity of these beasts.
(Happy troubleshooting by the way - cost me like 2 days to find a small little bug)

Regression

As mentioned in the theoretical part of Neural Networks, they can either be used for Regression or Classification (as done above). The Classification utilizes the cross entropy, equation (30), as a loss function. The neat benefit of the cross-entropy loss is that it is a convex function, meaning a local minimum is also automatically a global minimum. This is awesome for our minimization problem, since it is sure to find the global minimum of our function and thus the best result.
In classification there is a very particular set of possible output values, the possibles classes. Hence, Mean-Squared Error (MSE), equation (29), is badly defined, as it does not have this kind of knowledge and in turn penalizes errors in an incompatible way. However, in Regression where the goal is to predict a continuous value, MSE gives a great indicator how far apart the predicted value to the true value is.

Moving on to Regression Analysis with the neural network. As mentioned perviously, the cost function changed from Cross Entropy Loss (for classification) to Mean Squared Error for Regression.

In [44]:
```
# import the image libraries we need
from PIL import Image
from matplotlib import cm
from imageio import imread
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FormatStrFormatter

def plot_terrain(file_number=2):
    # caption the plot
    if file_number == 2:
        area_name = 'the Møsvatn Austfjell area'
        plot_number = 8
    else:
        raise ValueError("There is only the Møsvatn area, which is the second
 file, so only call it with '2'.")
    txt = 'Fig. ' + str(plot_number) + ' Terrain data used within this projec
t. The above figure illustrates ' + area_name + ' in Norway. Data taken from U
SGS EarthExplorer [1].'
    # read the file and generate the data
    fileName = os.path.join(os.getcwd(), 'SRTM_data_Norway_' + str(file_number
) + '.tif')
    image = Image.open(fileName, mode='r')
    image.mode = 'I'
    #print(image.size) # width, height
    x = np.linspace(0, 1, image.size[0])
    y = np.linspace(0, 1, image.size[1])
    X,Y = np.meshgrid(x,y)
    Z = np.array(image)
    Z = Z - np.min(Z)
    Z = Z / np.max(Z)

    # plot the figure
    fig = plt.figure(figsize=(9,7))
    ax = fig.gca(projection='3d')
    ax.plot_surface(X,Y,Z,cmap=cm.coolwarm,linewidth=0, antialiased=False)
    ax.set_zlim(-0.10, 1.20)
    ax.set_title("Terrain Data of " + area_name + ", Norway")
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
    ax.view_init(30, 45+90)
    fig.text(.1,.1,txt)
    plt.savefig(os.path.join(plot_dir, 'terrain_' + area_name + '.png'), trans
parent=True, bbox_inches='tight')
    plt.show()
plot_terrain(2)
```

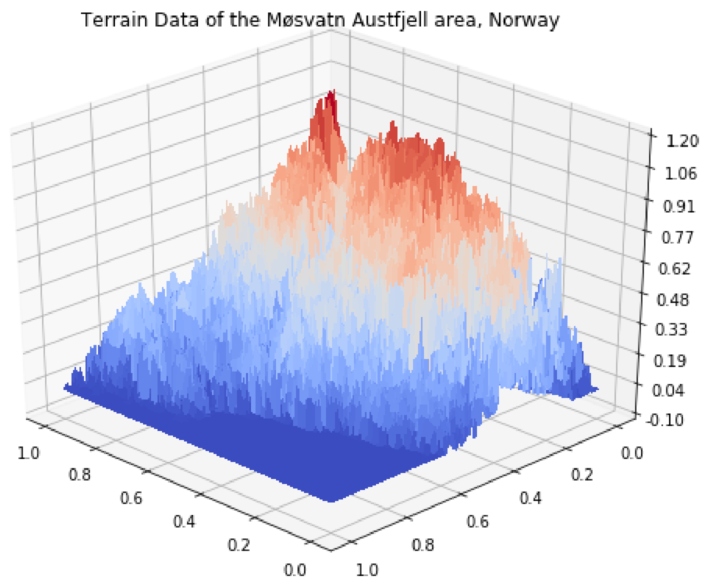


Fig. 8 Terrain data used within this project. The above figure illustrates the Møsvatn Austfjell area in Norway. Data taken from USGS EarthExplorer [1].

In [45]:
```
def terrain_to_x_y_z(self, filenumber=2):
    #setting up data points for real data
    z = imread('SRTM_data_Norway_' + str(filenumber) + '.tif')
    x = np.linspace(0, 1, len(z[1])).reshape(len(z[1]), 1)
    y = np.linspace(0, 1, len(z)).reshape(len(z),1) # normalized data from 0 t
o 1
    terrain_x, terrain_y = np.meshgrid(x,y)
    terrain_z = z/np.max(z) # normalize
    return terrain_x, terrain_y, terrain_z
```

In [46]:
```
## massaging the Terrain data to the right shape
mosvatn_x, mosvatn_y, mosvatn_z = terrain_to_x_y_z(2)
X = np.vstack([mosvatn_x.ravel(), mosvatn_y.ravel()]).T
y = np.reshape(mosvatn_z, X.shape[0])

## for our split we will have to enable the shuffling, since it could just lea
rn the position by time.
## to get random samples from the entire space and not spatial correlated once
we just enable the shuffling.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rando
m_state=42, shuffle=True)

# bring both training data to same shape
# y_train2 = np.vstack([y_train, y_train]).T
# y_test2 =  np.vstack([y_test, y_test]).T

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(5188320, 2) (5188320,)
(1297081, 2) (1297081,)
```

The following code is from project 1 (https://github.com/lenlehm/Regression-and-Resampling). The function used is in the Appendix, but is the exact copy of project 1.

In [47]:
```
%%time
lamda = 0.001
max_poly_degree = 7
mse_OLS_train, mse_OLS_test, r2_OLS_train, r2_OLS_test, mse_ridge_train, mse_r
idge_test, r2_ridge_train, r2_ridge_test, mse_lasso_train, mse_lasso_test, r2_
lasso_train, r2_lasso_test = Regression_Methods(X, y, max_poly_degree=max_poly
_degree, lamda=lamda)
```

```
Evaluate degree, now at: 0 out of: 6
Evaluate degree, now at: 1 out of: 6
Evaluate degree, now at: 2 out of: 6
Evaluate degree, now at: 3 out of: 6
Evaluate degree, now at: 4 out of: 6
Evaluate degree, now at: 5 out of: 6
Evaluate degree, now at: 6 out of: 6
Wall time: 8min 11s
```

In [48]:
```
print("MSE Scores for the Regression methods, Lasso and Ridge used a λ = {}\n"
.format(lamda))
print(tabulate([ [i, olsTest, olsTrain, ridgeTest, ridgeTrain, lassoTest, lass
oTrain] for i, olsTest, olsTrain, ridgeTest, ridgeTrain, lassoTest, lassoTrain
in zip(np.arange(len(mse_OLS_test)), mse_OLS_test, mse_OLS_train, mse_ridge_te
st, mse_ridge_train, mse_lasso_test, mse_lasso_train)], headers=['Degree', 'OL
S Test', 'OLS Train','RIDGE Test','RIDGE Train','LASSO Test', 'LASSO Train'],
tablefmt="github"))
```

```
MSE Scores for the Regression methods, Lasso and Ridge used a λ = 0.001

|   Degree |    OLS Test |   OLS Train |   RIDGE Test |   RIDGE Train |   LAS
SO Test |   LASSO Train |
|----------|-------------|-------------|--------------|---------------|------
--------|---------------|
|        0 | 0.0604247   | 0.06044     |  0.0604247   |   0.06044     |  0.06
04247   |   0.06044     |
|        1 | 2.56953e-29 | 2.57137e-29 |  1.6215e-19  |   1.62231e-19 |  1.65
412e-05 |   1.65453e-05 |
|        2 | 2.79996e-28 | 2.80209e-28 |  2.32916e-18 |   2.33249e-18 |  1.65
412e-05 |   1.65453e-05 |
|        3 | 1.26622e-26 | 1.26865e-26 |  1.41153e-17 |   1.41558e-17 |  1.65
412e-05 |   1.65453e-05 |
|        4 | 8.30612e-25 | 8.31962e-25 |  5.16924e-17 |   5.19319e-17 |  1.65
412e-05 |   1.65453e-05 |
|        5 | 1.13085e-22 | 1.13255e-22 |  1.49742e-16 |   1.50455e-16 |  1.65
412e-05 |   1.65453e-05 |
|        6 | 4.21588e-20 | 4.21981e-20 |  3.96656e-16 |   3.97967e-16 |  1.65
412e-05 |   1.65453e-05 |
```

In [49]:
```
print("R2 Scores for the Regression methods, Lasso and Ridge used a λ = {}\n".
format(lamda))
print(tabulate([ [i, olsTest, olsTrain, ridgeTest, ridgeTrain, lassoTest, lass
oTrain] for i, olsTest, olsTrain, ridgeTest, ridgeTrain, lassoTest, lassoTrain
in zip(np.arange(len(r2_OLS_test)), r2_OLS_test, r2_OLS_train, r2_ridge_test,
r2_ridge_train, r2_lasso_test, r2_lasso_train)], headers=['Degree', 'OLS Test'
, 'OLS Train','RIDGE Test','RIDGE Train','LASSO Test', 'LASSO Train'], tablefm
t="github"))
```

```
R2 Scores for the Regression methods, Lasso and Ridge used a λ = 0.001

|   Degree |     OLS Test |   OLS Train |   RIDGE Test |   RIDGE Train |   LA
SSO Test |   LASSO Train |
|----------|--------------|-------------|--------------|---------------|-----
---------|---------------|
|        0 | -5.17956e-07 |           0 | -5.17956e-07 |  -8.88178e-16 | -5.1
7956e-07 |      0        |
|        1 |  1           |           1 |  1           |   1           |  0.9
99726    |      0.999726 |
|        2 |  1           |           1 |  1           |   1           |  0.9
99726    |      0.999726 |
|        3 |  1           |           1 |  1           |   1           |  0.9
99726    |      0.999726 |
|        4 |  1           |           1 |  1           |   1           |  0.9
99726    |      0.999726 |
|        5 |  1           |           1 |  1           |   1           |  0.9
99726    |      0.999726 |
|        6 |  1           |           1 |  1           |   1           |  0.9
99726    |      0.999726 |
```

In [51]:
```
# plot that stuff
txt = "Fig. 9 Different Regression methods and their respective MSE score on t
he Terrain data set displayed\n Ridge and Lasso utilized a regularization para
meter λ = " + str(lamda)
fig = plt.figure(figsize=(9,7))
plt.plot(np.arange(max_poly_degree)[1:], mse_OLS_test[1:], label="Test Accurac
y OLS")
plt.plot(np.arange(max_poly_degree)[1:], mse_ridge_test[1:], label="Test Accur
acy RIDGE")
plt.plot(np.arange(max_poly_degree)[1:], mse_lasso_test[1: ], label="Test Accu
racy LASSO")
plt.title("MSE Accuracies across the different Regression techniques")
plt.ylabel("Accuracy Score (MSE)")
plt.xlabel("Polynomial degree")
plt.xticks(np.arange(max_poly_degree))
plt.legend()
fig.text(.1, 0, txt)
plt.savefig(os.path.join(plot_dir, 'Regression on Terrain.png'), transparent=T
rue, bbox_inches='tight')
plt.show()
```

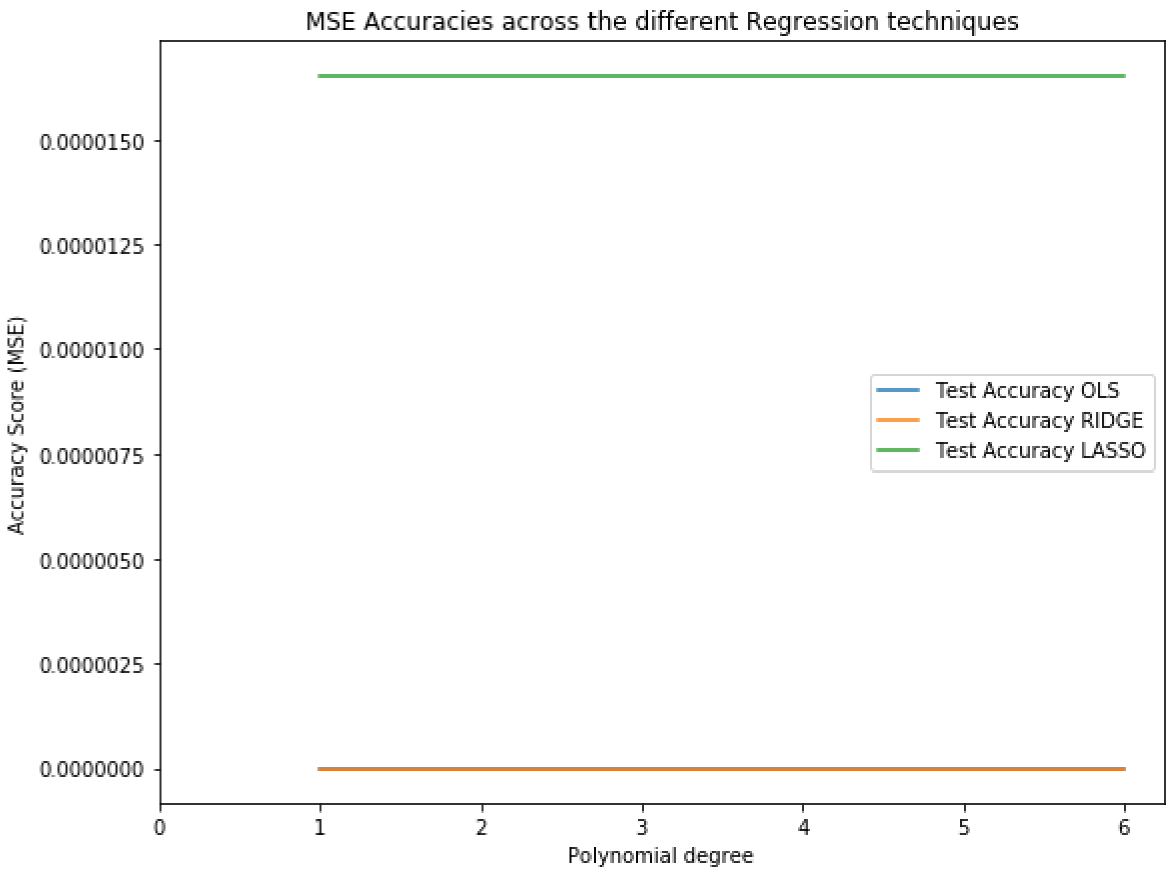


Fig. 9 Different Regression methods and their respective MSE score on the Terrain data set displayed
Ridge and Lasso utilized a regularization parameter λ = 0.001

These results are picture perfect. There are basically no better results to be achieved. This happens if one does not add random noise to the targets, Ridge and OLS can perfectly predict the target values.
So there is no chance for the neural network to top it, but let's see how it will perform.

In [39]:
```
## STANDARDIZED VERSION
nn = NeuralNetwork(inputs             = X_train.shape[1],
                   outputs            = 1,
                   cost_function_name = 'mse')

## NN Setup of: 128, 128, 128, 64 Neurons in the hidden layer and 2 neurons as
output layer (binary classification)
## All the layers are activated with the ReLU activation function, which mitig
ates the vanishing gradient problem
nn.addLayer(activations = 'relu', neurons = 64)
nn.addLayer(activations = 'relu', neurons = 32)
nn.addLayer(activations = 'identity', neurons = 1, output = True)

nn.train_network(X_train.T, y_train.T,
       batch_size          = 256,
       learning_rate       = 0.001,
       epochs              = 5,
       val_size            = 0.2,
       val_stepwidth       = 10,
       optimizer           = 'adam',
       lmbda               = 0.0001)

print("\nNeural Network Accuracy on Test data (20% split): {}\n".format(nn.acc
uracy(X_test, y_test)))
```

```
Adding input layer with 64 neurons, using relu
Adding layer with 32 neurons using relu
Adding output layer with 1 outputs and identity
There was always the label: 0 predicted

Neural Network Accuracy on Test data (20% split): 0.9999992290381249
```

HALLELULJA! This literally took over 10h to compute - never would I have thought this works out!
I am really sorry, but I will not tune anything here, due to obvious time reasons.
However the Accuracy score is quite impressive with 99.99% it did an amazing job and is definitely comparable with the OLS and Ridge Regression method.
To think a little further for multi-dimensional data, when even OLS and Ridge would have a hard time, I guess that Neural Networks will be superior in high-dimensional data, since they can accurately model the strictly non linear data by the cost of a lot computation time.
Considering that the above net only has 2 hidden layers and was trained for 5 epochs, adding more layers and more non-linearities can definitely boost the accuracy even more.
Hence, I dare to say that Neural Networks are definitely comparable in terms of accuracy for this Terrain Dataset, however not really efficient in terms of computational power and time consumption.

In [29]:
```
%%time
from keras.models import Sequential
from keras.layers import Dense

# Create the model
model = Sequential()
model.add(Dense(256, input_dim=2, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='linear'))

# Configure the model and start training
model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['mean_squ
ared_error'])
model.fit(X_train, y_train, epochs=10, batch_size=256, verbose=1, validation_s
plit=0.2)
```

```
Train on 4150656 samples, validate on 1037664 samples
Epoch 1/10
4150656/4150656 [==============================] - 666s 161us/step - loss: 0.
0587 - mean_squared_error: 0.0080 - val_loss: 0.0563 - val_mean_squared_erro
r: 0.0071
Epoch 2/10
4150656/4150656 [==============================] - 852s 205us/step - loss: 0.
0536 - mean_squared_error: 0.0068 - val_loss: 0.0524 - val_mean_squared_erro
r: 0.0064
Epoch 3/10
4150656/4150656 [==============================] - 728s 175us/step - loss: 0.
0521 - mean_squared_error: 0.0065 - val_loss: 0.0533 - val_mean_squared_erro
r: 0.0069
Epoch 4/10
4150656/4150656 [==============================] - 714s 172us/step - loss: 0.
0511 - mean_squared_error: 0.0063 - val_loss: 0.0505 - val_mean_squared_erro
r: 0.0061
Epoch 5/10
4150656/4150656 [==============================] - 724s 175us/step - loss: 0.
0504 - mean_squared_error: 0.0062 - val_loss: 0.0496 - val_mean_squared_erro
r: 0.0058
Epoch 6/10
4150656/4150656 [==============================] - 728s 176us/step - loss: 0.
0499 - mean_squared_error: 0.0060 - val_loss: 0.0494 - val_mean_squared_erro
r: 0.0060
Epoch 7/10
4150656/4150656 [==============================] - 740s 178us/step - loss: 0.
0495 - mean_squared_error: 0.0060 - val_loss: 0.0502 - val_mean_squared_erro
r: 0.0059
Epoch 8/10
4150656/4150656 [==============================] - 734s 177us/step - loss: 0.
0491 - mean_squared_error: 0.0059 - val_loss: 0.0491 - val_mean_squared_erro
r: 0.0059
Epoch 9/10
4150656/4150656 [==============================] - 739s 178us/step - loss: 0.
0489 - mean_squared_error: 0.0058 - val_loss: 0.0485 - val_mean_squared_erro
r: 0.0057
Epoch 10/10
4150656/4150656 [==============================] - 738s 178us/step - loss: 0.
0487 - mean_squared_error: 0.0058 - val_loss: 0.0487 - val_mean_squared_erro
r: 0.0058
```

```
Out[29]: <keras.callbacks.History at 0x21a9a22ecc0>
```

The above model in Keras took at least 5h to compute - I was not expecting that! So no parameter changes here, sorry for that.
However, the results are pretty bad for the long computaion time.

```
In [33]: predictions = model.predict(X_test)
         score = metrics.mean_squared_error(y_test, predictions)
         print("Keras MSE Score: {:.4f}".format(score))
```

```
0.005858260986117952
```

```
In [36]: r2 = metrics.r2_score(y_test, predictions)
         print("Keras R2  Score: {:.4f}".format(r2))
```

```
0.9030485945064795
```

As you can see in the table above, OLS outperforms not only the other common Regression methods like Ridge and Lasso, but also the Neural Network.
All "traditional" regression methods share their effect on the polynomial degree: The lower the degree the better the MSE score (excluding the 0th degree).
OLS achieves the best results, however Ridge also gets great results and Lasso Regression is not doing excellent on this dataset. Hence this dataset does not need any regularization.
Since OLS and Ridge are doing a quite comparable job and Lasso is doing worse, the figure below (Fig. 9) seems almost constant over the degrees, but considering the table above, each degree added will reduce the MSE score by approximately a tenth.

All of the traditional Regression methods outperformed the Neural Network on the Regression tasks, especially when considering time and computational effort involved.
This might be due to the network not having finished training, since it only had 5 or 10 epochs to train, however the training on these epochs already took forever (literally over night). So please understand my feelings with tuning any parameters in that model to even take longer and I either have the time or money to aff(j)ord it.

Analysis of the work

This work expanded on Logistic Regression and Neural Networks, especially Multi-Layered Perceptrons (MLP). As seen in the first part of this work, where credit card data has been successfully classified, MLP slightly outperformed Logistic Regression, but took longer to converge along with a tedious hyperparameter search. Taken into account that my Notebook has limited computing power - Intel Core i7, 7th generation, 16GB RAM and 512GB SSD - there was the bare minimum of hyperparameter search done for the MLP.

Hyperparameters are essential parameters to this algorihtm, which need to be tuned to end up in an optimal solution. These hyperparameters include the learning rate τ, batch sizes, regularization parameter λ, amount of hidden layers, amount of neurons within each layer, activation functions, epochs and optimizer (SGD or Adam). Hence, there is even more potential for MLP with the given credit card dataset, when performing a hyperparameter search by using either random search or Bayesian Analysis. [3]

In conclusion Logistic Regression is to be prefered for this classification dataset due to its simplicity and time benefits by almost the same accuracy.

The neat benefit of neural networks is that they are also useful for Regression tasks, as shown in the second part of the project. By only changing the Loss function from Cross Entropy to MSE, the same algorithm can now predict a continuous value as seen in the terrain dataset above.

Comparing the Neural Network as a regressor to the common regression methods such as Ordinary Least Squares, Ridge and Lasso Regression, one can tell that the common regression methods, especially OLS and Ridge Regression achieved remarkably good results (see the Table four cells above).
Nevertheless, the neural network still outperformed both of those Regression methods, since it can model even more complex shapes.

To put it in a nutshell, neural networks are flexible and adaptive tools, which in turn need a lot of fine tuning and hyperparameter search. For both tasks, Regression and Classification, it challenged the performance of its competitors such as OLS, Ridge and Logistic Regression.
These benefits however, come at the cost of computational power and complexity for searching optimal parametrization values.

5.) Conclusions

This work shows that it is possible to get similiar results for Logistic Regression and Artificial Neural Network of Yeh and Lien. [4]
When it comes to classify the credit card data, the neural networks were slightly better in accuracy than its rival, Logistic Regression, only with the drawback of a more complex implementation as well as sensitive hyperparameters.
Keep in mind that this work did not focus on searching for optimal hyperparameters to optimally classify the credit card default dataset, but rather focuses on the implementation and effects of some parameter changes. Hence, when performing proper hyperparameter search it is most probable that the Neural Networks will outperform Logistic Regression.

Considering Regression on the terrain dataset, Neural Networks are also comparable to the OLS and Ridge Regression in terms of accuracy. Again, its complexity and sensitivity are more of a problem for these easier, low-dimensional problems in Regression.
Having a multi-dimensional Regression problem at hand I dare to say that Neural Networks are more flexible and adaptive to this dataset than the common Regression methods such as OLS and Ridge Regression.

Albeit all of the time and complexity drawbacks of the Neural Network, they are a wonderful tool for both tasks, Classification and Regression. They scored seemingly well on both datasets and tasks and definitely do have more potential when properly tuning the parameters.

6.) Further Work

As the section about the Classification dataset in the Introduction already states, the dataset itself has some values that are not documented. Manually checking and cleaning the dataset that it follows its documentation would most probably lead to better results.
Going further, one could perform feature engineering to enhance the dataset by adding some more predictor variables to it and thus boosting the accuracies for the classification algorithms at hand.
Leveraging this technique along with utilizing other classification algorithms such as Gradient Boosting Machines, i.e. AdaBoost, it would be interesting to compare those methods in terms of accuracies and performance. According to the authors [4], these methods revealed promising results and digging deeper into these techniques could yield even better results.
Another interesting approach would be combining this dataset with another credit card default dataset, such as the German one from before 2000 (https://github.com/olethrosdc/ml-society-science/tree/master/data/credit) to end up in one huge dataset, or comparing the different nations to one another and deriving some insights with respect to the banks handing out credits.

When it comes to Regression, it would be more interesting to compare the common Regression methods to the neural networks for higher dimensional datasets such as the Boston House prices (https://www.kaggle.com/vikrishnan/boston-house-prices). Since the common Regression methods could pretty much perfectly predict the elevation of the terrain data, having a multi-dimensional dataset would definitely reveal some more interesting properties for both, Neural Networks and Ridge Regression.

References

[1] Shanahan, M. (2015). *The Techonological Singularity*. Camebridge: The MIT Press
[2] Saintano, M. (2015). *Stephen Hawking, Elon Musk, and Bill Gates Warn About Artificial Intelligence*. Retrieved from Observer (https://observer.com/2015/08/stephen-hawking-elon-musk-and-bill-gates-warn-about-artificial-intelligence/) on 8th October 2019
[3] Baeuml, B. (2019). *Advanced Deep Learning for Robotics*. Munich: Lecture at Technical University Munich (TUM) (https://github.com/bbaeuml/ss19-advanced-dl-for-robotics/blob/master/docs/adlr-2-advanced-networks.pdf), Summer Term 2019 (password: TUM19ADLR)
[4] Yeh, C. et al. (2009). *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients*. ELSEVIER, Source (https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf) Retrieved from ELSEVIER on 8th October 2019
[5] Hjorth-Jensen, M. (2019). *Applied Data Analysis and Machine Learning*. Oslo: Lecture hold at University of Oslo (UiO), Source (https://compphysics.github.io/MachineLearning/doc/web/course.html)
[6] Hastie, T. et al. (2009). *The Elements of Statistical Learning*. Camebridge: Springer
[7] Knoll, A. (2019). *Cognitive Systems*. Lecture hold at Technical University Munich (TUM), Source (https://github.com/lenlehm/Classification-and-Regression/blob/master/E04-Deep_Learning.pdf) included in Github Repository
[8] Murphy, K. P. et al. (2007). *Machine Learning: A probabilistic Perspective*. Camebridge: The MIT Press.
[9] Diederik K. et al. (2015). *Adam: A Method for Stochastic Optimization*. Cornell University, ArXiv: https://arxiv.org/abs/1412.6980 (https://arxiv.org/abs/1412.6980), retrieved 8th October, 2019
[10] Glorot, X. et al. (2010). *Understanding the difficulty of training deep feedforward neural networks*. Cornell University, ArXiv: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf), retrieved 8th October, 2019.
[10] Bishop, C.M. et al. (2011). *Pattern Recognition and Machine Learning*. Cambridge: Springer.
[11] EarthExplorer website: https://earthexplorer.usgs.gov/ (https://earthexplorer.usgs.gov/), Used Dataset: Norway, last visited 05.09.2019
[12] Grover P. (2018): *5 Regression Loss Functions All Machine Learners Should Know*. Medium: https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0 (https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0), retrieved 28th October, 2019.

Appendix

Some more plots for the terrain data λ parametrization.
All of the plots below are showing the Mean Squared Error (MSE) of Ridge or Lasso, respectively.

The following is the code, which has been used in project 1 only without noisy targets.
This code was used for the Terrain data Regression Analysis, which was shown towards the end of this work.

In [8]:
```
from sklearn.preprocessing import PolynomialFeatures
from RegressionMethods import CreateDesignMatrix_X
from sklearn import linear_model, metrics

def Regression_Methods(X, y, max_poly_degree=7, lamda=0.001):
    '''
    INPUT:
    ----------
    X: numpy ndarray or pandas Dataframe
        entire input data which it counts to analyse using regression
    y: numpy array or pandas Series
        corresponding targets for the input data rows
    max_ploy_degree: integer (default = 7 - too long computation time and no a
ccuracy benefit)
        indicator of which polynomial degree to fit to the dataset
    lamda: float (default = 0.001 - empirical test have proven this to be quit
e good)
        regularization strength parameter for Ridge and Lasso Regression

    OUTPUT:
    ----------
    train/test_metric: numpy ndarray
        Metrics (either MSE or R2) for the corresponding regression method
        There are in total 12 return values with train and test accuracy for a
ll the 3 methods
    '''
    # OLS train and test scores
    mse_OLS_train = np.zeros(max_poly_degree)
    mse_OLS_test  = np.zeros(max_poly_degree)
    r2_OLS_train  = np.zeros(max_poly_degree)
    r2_OLS_test   = np.zeros(max_poly_degree)

    # Ridge train and test scores
    mse_ridge_train = np.zeros(max_poly_degree)
    mse_ridge_test  = np.zeros(max_poly_degree)
    r2_ridge_train  = np.zeros(max_poly_degree)
    r2_ridge_test   = np.zeros(max_poly_degree)

    # Lasso train and test scores
    mse_lasso_train = np.zeros(max_poly_degree)
    mse_lasso_test  = np.zeros(max_poly_degree)
    r2_lasso_train  = np.zeros(max_poly_degree)
    r2_lasso_test   = np.zeros(max_poly_degree)

    # Split the Data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, r
andom_state=42, shuffle=True)

    # bring both training data to same shape
    y_train = np.vstack([y_train, y_train]).T
    y_test =  np.vstack([y_test, y_test]).T

    # now check the proper parametrization and benchmark the algorithms agains
t each other
    for degree in range(max_poly_degree):
        # put everything that DOES NOT depend on lamda in this loop --> entire
```

```
OLS & design Matrices
        print("Evaluate degree, now at: " + str(degree) + ' out of: ' + str(ma
x_poly_degree-1))
        # Train design Matrix to fit our regression model
        X = CreateDesignMatrix_X(X_train.ravel(), y_train.ravel(), degree)
        # Test design Matrix to evaluate the Test Set
        designX_test = CreateDesignMatrix_X(X_test.ravel(), y_test.ravel(), de
gree)

        beta_OLS = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y_train.ravel())#res
hape(-1,1))
        beta_ridge = np.dot(np.linalg.inv(np.dot(np.transpose(X),X) + lamda *
np.eye(X.shape[1])), np.dot(np.transpose(X), y_train.ravel()))#reshape(-1,
 1)))

        ## OLS prediction
        train_pred_OLS = X.dot(beta_OLS)
        test_pred_OLS  = designX_test.dot(beta_OLS)

        ## Ridge Prediction
        train_pred_ridge = X.dot(beta_ridge)
        test_pred_ridge = designX_test.dot(beta_ridge)

        ## LASSO
        polynom    = PolynomialFeatures(degree=degree)
        XY         = polynom.fit_transform(np.array([X_train.ravel(), y_train.
ravel()]).T)
        lasso      = linear_model.Lasso(fit_intercept=True, alpha=lamda)
        lasso.fit(XY, y_train.reshape(-1, 1))
        test_feed  = polynom.fit_transform(np.array([X_test.ravel(), y_test.ra
vel()]).T)
        train_feed = polynom.fit_transform(np.array([X_train.ravel(), y_train.
ravel()]).T)
        ## prediction
        train_pred_lasso = lasso.predict(train_feed)
        test_pred_lasso  = lasso.predict(test_feed)


        ## GET THE SCORES - MSE AND R2 ---------------- OLS
        mse_OLS_train[degree] = metrics.mean_squared_error(y_train.ravel(), tr
ain_pred_OLS)
        ## OLS is doing this one:  np.mean( np.mean( (train_target.ravel() - t
rain_pred_OLS)**2, axis=1, keepdims=True) )
        mse_OLS_test[degree]  = metrics.mean_squared_error(y_test.ravel(), tes
t_pred_OLS)

        r2_OLS_train[degree]  = metrics.r2_score(y_train.ravel(), train_pred_O
LS)
        r2_OLS_test[degree]   = metrics.r2_score(y_test.ravel(), test_pred_OLS
)

        ## RIDGE
        mse_ridge_train[degree] = metrics.mean_squared_error(y_train.ravel(),
train_pred_ridge)
        mse_ridge_test[degree]  = metrics.mean_squared_error(y_test.ravel(), t
est_pred_ridge)
        r2_ridge_train[degree]  = metrics.r2_score(y_train.ravel(), train_pred
_ridge)
        r2_ridge_test[degree]   = metrics.r2_score(y_test.ravel(), test_pred_r
idge)

        ## LASSO
        mse_lasso_train[degree] = metrics.mean_squared_error(y_train.ravel(),
train_pred_lasso)
        mse_lasso_test[degree]  = metrics.mean_squared_error(y_test.ravel(), t
est_pred_lasso)
        r2_lasso_train[degree]  = metrics.r2_score(y_train.ravel(), train_pred
_lasso)
        r2_lasso_test[degree]   = metrics.r2_score(y_test.ravel(), test_pred_l
asso)

    return mse_OLS_train, mse_OLS_test, r2_OLS_train, r2_OLS_test, mse_ridge_t
rain, mse_ridge_test, r2_ridge_train, r2_ridge_test, mse_lasso_train, mse_lass
o_test, r2_lasso_train, r2_lasso_test
```

In [11]:

```
# plot ROC Curve
def plot_roc_curve(FPR, TPR, plot_number, text="ROC curve"):
    txt = "Fig. " + str(plot_number) + text
    fig = plt.figure(figsize=(9, 7))
    plt.plot(FPR, TPR, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    fig.text(.1, 0,txt)
    plt.savefig(os.path.join(plot_dir, 'ROC ' + str(plot_number) + '.png'), tr
ansparent=True, bbox_inches='tight')
    plt.show()
```