Using Neural Networks and Logistic Regression for Classification and Regression Problems

Eina B. Jørgensen, Anna Lina P. Sjur and Jan-Adrian H. Kallmyr

November 15, 2019

Abstract

We use a Neural Network (NN) and Logistic Regression (LR) to classify the 2005 Taiwan credit card data. There are two categories: non-default and default. A comparison is made between the two methods, and we find that the NN yields the highest accuracy score (HER) compared with (HER) from the LR algorithm. Area ratio was also used as a performance measure, the NN producing the highest score of (HER). In terms of efficiency, the LR algorithm converges quite rapidly compared to the NN, and is overall more stable, possibly due to an implementation of dynamic learning rate. Furthermore, we use the NN to fit the Franke function (with noise), and compare this to linear regression algorithms used in our earlier paper. Our findings suggest that

1 Introduction

Artifical Neural Networks (NNs) offer a flexible way of performing data analysis, and can be used for many different problems in machine learning. In particular, we can use ANNs for classification and regression problems such as determining if a planet is terrestrial or not, or fitting terrain data. It is therefore of interest to measure the performance and quality of NNs compared with other machine learning algorithms.

In this article, we will study the 2005 Taiwan credit card data set with the purpose of classifying risky/non-risky credit card holders. For this, we use two methods: logistic regression and artificial neural networks. We also use a ANN to fit the Franke function (see our earlier paper Jørgensen et al. (2019)). In both cases, we compare our ANN to the respective methods: logistic and linear regression (Jørgensen et al., 2019).

Starting with section 2, we describe the data set and the methods used in detail, as well as our chosen activation, cost, and score functions. Moving on to section 3, we present our most important results such as tuning parameters and accuracy scores for each method. In section 4 we discuss the method comparisons in more detail, and consider our results. Furthermore, possible lacks and improvements are suggested, before concluding our paper in section 5.

2 Theory and methods

2.1 Data sets

In this report, the default of credit card clients data set (Yeh and Lien, 2009) was used to study the performance and compare the logistic regression method and neural networks for classification. A description of the attributes of the data set can be found in Yeh and Lien (2009). Upon inspection, it is notable that the data set contains a considerable amount of

values different from their valid values as described by Yeh and Lien. Mostly, this apply for the categorical variables, where the given value does not correspond to any category. By removing data points with invalid values, as well as entries where the client does not have any history of past payments or bill statements, the data set is reduced from 30000 data points to 3792 points. As this is a considerable reduction of data points, two additional data sets were constructed. In one, all the invalid PAY values were kept. These invalid values were some -2s, but mostly 0s. In the second additional set, all -2s, -1s and 0s were set to 0, since these were assumed to all represent duly payments. Both sets contained 26908 data points, with 78% of the labels equal to zero, i.e. not default payment next month. The last set gave best results on initial runs, and was used for the rest of the project. For a more in-depth inspection of the data set, see the Jupyter notebook `inspect_data.ipynb` in the GitHub repository of janadr.

In addition to the default of credit card clients data set, data produced with Franke's function was applied to neural networks. For details on Franke's function, see Jørgensen et al. (2019).

2.2 Logistic Regression

Classification problems aim to predict the behaviour of a given object, and look for patterns based on discrete variables (i.e categories). Logistic Regression can be used to solve such problems, commonly by the use of variables with binary outcomes such as true/false, positive/negative, success/failure etc., or in the specific credit card case: *risky/non-risky* Hjort-Jensen (2019b)

As opposed to Linear Regression, the equation one gets as a result of minimization of the cost function by $\hat{\beta}$ using Logistic Regression, is non-linear, and is solved using minimization algorithms called *gradient descent methods*.

When predicting the output classes in which an object belongs, the prediction is based on the design matrix $\hat{\mathbf{X}} \in \mathbb{R}^{n\times p}$ that contain n samples that each carry p features.

A distinction is made between *hard classification* - determines the variable to a category deterministically, and *soft classification* - determines the probability that a given variable belongs in a certain category. The latter is favourable in many cases, and logistic regression is the most used example of this type of classifier.

When using Logistic Regression, the probability that a given data point x_i belongs in a category y_i is given by the Sigmoid-function (or logistic function):

$$\begin{aligned} p(t) &= \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t} \\ 1-p(t) &= p(-t) \end{aligned} \tag{1}$$

Assuming a binary classification problem, i.e. y_i can be either 0 or 1, and a set of predictors $\hat{\beta}$ the Sigmoid function (1) gives the probabilities with relation:

$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta})$$

The total likelihood for all possible outcomes $\mathcal{D} = \{(y_i, x_i)\}$ is used in the Maximum Likelihood Estimation (MLE), aiming at maximizing the log/likelihood function (2). The likelihood function can be expressed with $\mathcal{D}$:

$$P(\mathcal{D}|\hat{\beta}) = \prod_{i=1}^{n} \Big[p(y_i = 1|x_i, \hat{\beta})\Big]^{y_i} \Big[1 - p(y_i = 0|x_i, \hat{\beta})\Big]^{1-y_i}$$

And the log/likelihood function is then:

$$\begin{aligned} P_{\log}(\hat{\beta}) = \sum_{i=1}^{n} \Big(y_i \log \Big[p(y_i = 1|x_i, \hat{\beta})\Big] \\ + (1-y_i)\log[1 - p(y_i = 0|x_i, \hat{\beta})]\Big) \end{aligned} \tag{2}$$

The cost/error-function $\mathcal{C}$ (also called cross-entropy in statistics) is the negative of the log/likelihood. Maximizing $P_{\log}$ is thus the same as minimizing the cost function. The cost function is:

$$\mathcal{C}(\hat{\beta}) = -P_{\log}(\hat{\beta}) = \\ -\sum_{i=1}^{n} \Big(y_i \log \Big[p(y_i = 1|x_i, \hat{\beta}) \Big] \\ + (1 - y_i) \log[1 - p(y_i = 0|x_i, \hat{\beta})] \Big) \tag{3}$$

Finding the parameters $\hat{\beta}$ that minimize the cost function is then done through differentiation. Defining the vector $\hat{y}$ containing n elements y_i, the $n \times p$ matrix $\hat{X}$ containing the x_i elements, and the vector $\hat{p}$ that is the fitted probabilities $p(y_i|x_i, \hat{\beta})$, the first derivative of $\mathcal{C}$ is

$$\nabla_\beta \mathcal{C} = \frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T(\hat{y} - \hat{p}) \tag{4}$$

This gives rise to set of linear equations, where the aim is to solve the system for $\hat{\beta}$.

With $\hat{x} = [1, x_1, x_2, ..., x_p]$ and p predictors $\hat{\beta} = [\beta_0, \beta_1, \beta_2, ..., \beta_p]$ the ration between likelihoods of outcome is:

$$\log \frac{p(\hat{\beta}\hat{x})}{1 - p(\hat{\beta}\hat{x})} = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p \tag{5}$$

and $p(\hat{\beta}\hat{x})$ defined by:

$$p(\hat{\beta}\hat{x}) = \frac{e^{\beta_0 + \beta_1 x_1 + ... + \beta_p x_p}}{1 + e^{\beta_0 + \beta_1 x_1 + ... + \beta_p x_p}} \tag{6}$$

2.3 Gradient Descent Methods

The General Idea

With the gradient of $\mathcal{C}$ defined as in (4), we used this to find the minimum of the cost function. The basic idea is that by moving in the direction of the negative gradient of a function, we can move towards the value (in this case the β) that minimizes the function (in this case $\mathcal{C}(\beta)$). Hjort-Jensen (2019a)

This is done by repeating the algorithm

$$\beta_{j+1} = \beta_j - \gamma \nabla_\beta \mathcal{C}(\beta) \quad j = 0, 1, 2, ... \tag{7}$$

When a minimum is approached, $\nabla_\beta \mathcal{C}(\beta) \to 0$, and thus we can set a limit when $\beta_{k+1} \approx \beta_k$ given a certain tolerance, and the β which minimizes the cost function is found. γ is in this case called the *learning rate*, and is a parameter that must be tuned to each specific case in order to optimize the regression.

Stochastic Gradient Descent

In this project we use a stochastic version of gradient descent, which is an improvement upon the regular gradient descent. This is done by expressing the cost function (and thus also its gradient) as a sum

$$\nabla_\beta \mathcal{C}(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\boldsymbol{x_i}, \beta), \tag{8}$$

and by only calculating the gradient of a subset of the data at the time. These subsets, called *minibatches* are of size $\boldsymbol{M}$, and the total amount is $\frac{n}{\boldsymbol{M}}$ where n is the amount of data points. The minibatches are denoted $\boldsymbol{B}_k$, with k = 1,2,...,$\frac{n}{\boldsymbol{M}}$.

Instead of a sum over all the the data points $i \in [1, n]$ we now sum over all the data points in the given minibatch $i \in \boldsymbol{B}_k$ where k is picked randomly with uniform probability from $[1, \frac{n}{\boldsymbol{M}}]$.

The stochastic and final version of (7) is therefore given by the algorithm

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in \boldsymbol{B}_k} \nabla_\beta c_i(\boldsymbol{x_i}, \beta) \tag{9}$$

An iteration over the total number of minibatches is commonly referred to as an *epoch.*

By using the stochastic gradient descent method (9) to minimize the cost function (3) we can find the β values that give the most

accurate classification, by doing *logistic regression.*

Dynamic learning rate

Convergence rate might be slow, and a possible solution to this is changing learning rate as the gradient descent gets closer to a local minima. Instead of a static learning rate, we have a learning schedule. As we expect convergence as iterations increase, it is natural that the learning schedule be a function of epochs and minibatches. Specifically, the learning rate should decrease as it approaches a minima. A possible function for the learning rate is then

$$\Gamma_i = \frac{\Gamma_0}{nN + i}, \tag{10}$$

where Γ_0 is the initial learning rate, n the current epoch, i the current minibatch, and N the minibatch size.

2.4 Neural Networks

In this section, the equations used are based off the book by Nielsen (2015), unless otherwise specified.

The structure of a network

Neural Networks, as the name suggests, are inspired by our understanding of how networks of neurons function in the brain. As can be seen in the example network in Figure 1, neurons are structured in layers. We always have a input and an output layer, in addition to a varying number of hidden layers. The input layer has as many neurons as there are input variables, while the output layer has one neuron for each output. How many neurons you have in the output layer depends on the specific problem. The number of neurons in each hidden layer, on the other hand, is not directly related to inputs or outputs, and must be decided in some other way.

As the diagram in Figure 1 suggests, the neurons in each layer are not connected with each other, but takes in inputs from the previous layer and passes on an output to the neurons in the next layer, as illustrated with arrows. This way, the inputs are fed through the network and processed, resulting in an output.

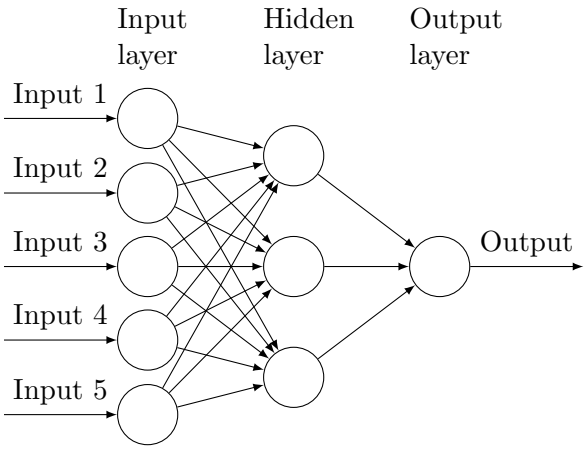

Figure 1: Schematic diagram of a neural network with five input neurons in the input layer, one hidden layer with tree neurons and a single output neuron in the output layer.

Forward feeding

Each neuron has one or multiple inputs, as illustrated with arrows in Figure 1. Each of these inputs has a weight associated with it. To clarify the notation used, let's take a look at the jth neuron in the lth layer. The weight associated with the input coming from the kth neuron in the previous layer is denoted as w^l_{jk}. In addition, each neuron has a bias associated with it, for the neuron in question denoted as b^l_j. Summing the weighted inputs and the bias, and feeding this to a function f, gives the ac-

tivation a_j^l:

$$a_j^l = f\left(\left(\sum_k w_{jk}^l a_k^{l-1}\right) + b_j^l\right)$$

This activation is then fed forward as input to all the neuron in the next layer.

In matrix notation, the activation for the whole layer l can be written as

$$\boldsymbol{a}^l = f\left(\boldsymbol{w}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l\right) \tag{11}$$

Here, $\boldsymbol{a}^l$ and $\boldsymbol{b}^l$ are vertical vectors containing the activations and biases of the lth layer, while $\boldsymbol{w}^l$ is a matrix with elements w_{jk}^l, i.e. the jth column contains the weights of the inputs reaching the jth neuron.

Let's look at the activation function in Eq. (11) denoted with a f. In the case of classification, the sigmoid function stated in Eq. (1) is often used in introductory texts. As we will see in the backpropagation algorithm, the sigmoid is a good choice for activation function, since a small change in the output can be propagated backwards, resulting in small changes in the weights and biases through the network.

Another activation function is the so-called rectified linear unit (ReLU) function, given as $f(z) = \max(0, z)$ This gives an activation function which only fires when z is positive. In the case of regression, a variant of ReLU, called leaky ReLU, was used in this project. This function is given as

$$f(z) = \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases} \tag{12}$$

This form of ReLU has a small positive gradient for negative values, which allows for gradient based learning even when the z-value is negative (Wang et al., 2018).

With a basis in Eq. (11), the algorithm for forward feeding is given in Algorithm 1. Here L is the total number of layers.

Set $\boldsymbol{a}^1$ = input;
foreach *l=2:L* **do**
  Compute $\boldsymbol{z} = \boldsymbol{w}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$;
  Compute $\boldsymbol{a}^l = f(\boldsymbol{z})$;
Set output to $\boldsymbol{a}^L$;

Algorithm 1: The forward feeding algorithm.

Note that the output will have values between 0 and 1, when the sigmoid function is used to compute the activations of all the layers. In a classification problem, this corresponds to the likelihood of an outcome. For example, in a classification problem with five classes, the network would have five output neurons, each representing a class. The final classification of an input would then be the class with the highest probability. In the regression case, we see that when using the leaky ReLU activation function from Eq. (12), the output can be any positive value (strictly also negative values, due to the leakage, but this would call for large negative z-values).

Backpropagation

When training the network, the goal is to find the weights and biases that minimize the cost function C. In this project, the cross-entropy cost function was used for classification. This is given as

$$C = -\sum_{i=1}^{n} \left[y_i \ln a_i^L + (1 - y_i) \ln\left(1 - a_i^L\right)\right] \tag{13}$$

Here, we are summing over n points of training data, where a_i^L is the output with the corresponding correct value y_i.

In the regression case, the quadratic cost function, given as

$$C = \frac{1}{2} \sum_{i=1}^{n} ||a_i^L - y_i||^2 \tag{14}$$

is used.

To find the weights and biases that minimize Eq.(13) and (14), one can use Stochastic Gradient Decent, as described previously. But in order to use SGD, the derivatives of C with respect to all the weights and biases must be computed, and it is here that backpropagation comes in. It can be shown, by repeating the chain rule, that the derivatives are given as in Eq. (15).

$$\begin{aligned} \delta^L &= \nabla_a C \odot f'(z^L) \\ \delta^l &= ((\boldsymbol{w}^{l+1})^T \delta^{l+1}) \odot f'(\boldsymbol{z}^l) \\ \frac{\partial C}{\partial b_j^l} &= \delta_j^l \\ \frac{\partial C}{\partial w_{jk}^l} &= a_k^{l-1} \delta_j^l \end{aligned} \tag{15}$$

Taking a look at the expression for δ^L, one can show that this reduces to $\delta^L = \boldsymbol{a} - \boldsymbol{y}$, when the cross-entropy cost function is combined with the sigmoid activation function, and similarly for the quadratic cost and ReLU when we have positive output values.

Equation (15) are the basis for the backpropagating algorithm, described in Algorithm 2.

Compute $\{\boldsymbol{a}^l\}_{l=1}^L$ with feed forward;
Compute δ^L;
Set $\frac{\partial C}{\partial \boldsymbol{b}^L} = \delta^L$;
Compute $\frac{\partial C}{\partial \boldsymbol{w}^L} = \delta^L (\boldsymbol{a}^{L-1})^T$;
foreach *l=L-1:2* **do**
  Compute δ^l;
  Set $\frac{\partial C}{\partial \boldsymbol{b}^l} = \delta^l$;
  Compute $\frac{\partial C}{\partial \boldsymbol{w}^l} = \delta^l (\boldsymbol{a}^{l-1})^T$;

Algorithm 2: The backpropagation algorithm.

Lastly, let's revisit the cost functions stated in Eq. (13) and (14). These equations were chosen to accompany the ReLU and sigmoid activation function, respectively, because when computing δ^L factors cancel out nicely, and we end up with $\delta^L = a^L - y$. This means that we get rid of the $f'(z^L)$-term, which can slow down learning significantly if the derivative is close to 0.

Overfitting and Regularization

A problem that can arise when fitting a regression model, as discussed in Jørgensen et al. (2019), is overfitting to data. This is a major problem also in neural networks, where the network can fit too heavily to outliers in the training set, especially when you have a large number of neurons. Apart from increasing the training data set and decreasing the number of neurons, a third approach is to use a regularization technique. Here, we will use the weight decay or L2 regularization technique. The idea is to add a term to the cost function. For both cross-entropy and quadratic cost, the new cost function can be written as

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

where C_0 is the original cost function and $\lambda > 0$ is the regularization parameter. The algorithm for updating the weights, when using stochastic gradient descent, then becomes

$$w \to \left(1 - \frac{\gamma\lambda}{n}\right) w - \frac{\gamma}{m} \sum_{i \in \boldsymbol{B}_k} \frac{\partial C_i}{\partial w} \tag{16}$$

Here, m is the mini batch size. The algorithm for updating the biases are unchanged:

$$b \to b - \frac{\gamma}{m} \sum_{i \in \boldsymbol{B}_k} \frac{\partial C_i}{\partial b} \tag{17}$$

Note the factor m^{-1} in Eq. (16) and (17) when comparing to Eq. (9). This factor can be understood as taking the mean of all the gradients, instead of the sum. In this project,

the sum was used for logistic regression, while the mean was used in the neural network. In practise, this corresponds to a scaling of the learning rate.

2.5 Quality of Measurements

To measure how good the different methods were at classifying the credit card data, the data was split into a training set and a test set. After training the model on the test set, it was applied on the training set, and a performance score was calculated. Both accuracy, given in Eq. (18) and the area under the receiver operating characteristic curve, denoted AUC, was applied.

$$\text{Accuracy} = \frac{\text{Correct classifications}}{\text{Total \# of classifications}} \tag{18}$$

For calculations of AUC, `sklearn.metrics.roc_auc_score` was used. AUC can be interpreted as the probability that a random true positive sample is ranked higher than a random true negative. This score is not dependent on the chosen classification threshold, i.e. the value that must be exceed to classify an input as one instead of zero.

For data generated with Franke's function, the same test-train-split was made, and the mean squared error was calculated, as in Jørgensen et al. (2019).

2.6 Implementation

All our algorithms were implemented in python3.6 using the numpy, matplotlib, pandas, seaborn, and scikit-learn packages. We wrote our own class "Regression" for linear and logistic regression, as well as class "Network" for neural network. In the way our neural network was implemented, we could use the same code for classification and regression by simply replacing activation functions, while at the same time ensuring that the appropriate cost function was used. Scikit-learn was used to verify our implementation by comparing accuracy and AUC scores. Code along with data and figures can be found on our github repository under "Project 2".[1]

3 Results

3.1 Tuning Learning Rate and Minibatch Size

Logistic Regression

Figure 2 shows a heatmap of the AUC of the Logistic Regression model used on the classification problem for different minibatch sizes and initial learning rates, where the variables are distributed logarithmically.

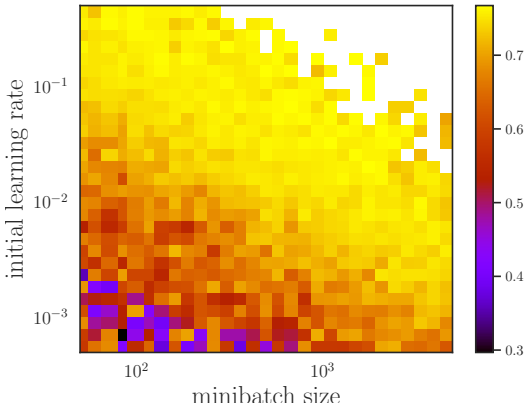

Figure 2: Heatmap showing the AUC of the Logistic Regression for different values of the minibatch sizes and initial learning rates.

Figure 3 is similar to Figure 2, except that the heatmap shows the accuracy, and not the AUC, for the Logistic Regression.

In both Figure 3 and 2, the monochromatic field to the top right represents parameters that gave overflow.

[1] https://github.com/janadr/FYS-STK4155.git

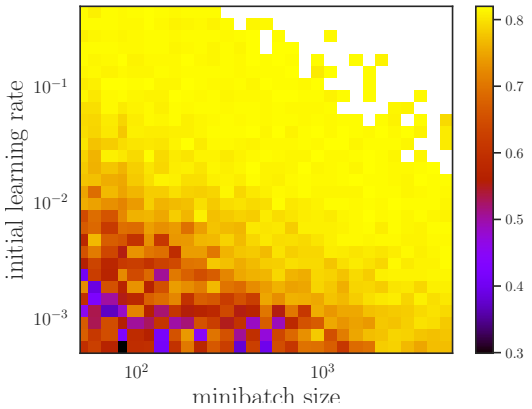

Figure 3: Heatmap showing the accuracy of the Logistic Regression for different values of the minibatch sizes and initial learning rates.

We see that both AUC and accuracy are highest in a band spanning form relatively high initial learning rates and relatively small batch sizes to smaller initial learning rates and larger batch sizes.

Neural network

Figure 5 and 2 shows heatmaps of the accuracy and AUC of the Neural Network.

Optimal Parameters and Associated Results

Table 1 spells out the values that gave the highest AUC in Figure 2 and 4.

Similarly, Table 2 shows the parameters that gave the highest accuracy in Figure 3 and 5.

In Table 3, we can see the number of true and false positives and negatives, when the parameters stated in Table 1 are used, i.e. the parameters that gave the highest AUC.

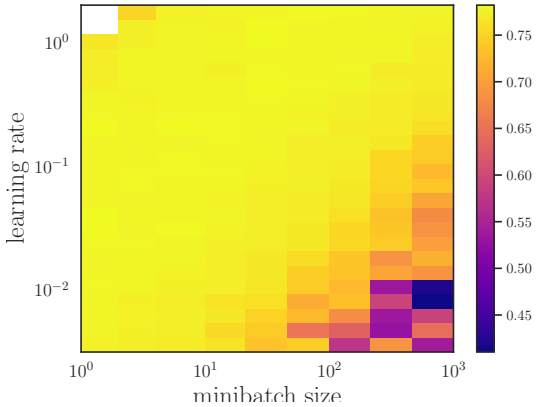

Figure 4: Heatmap showing the AUC of the Neural Network for different values of the minibatch sizes and learning rates.

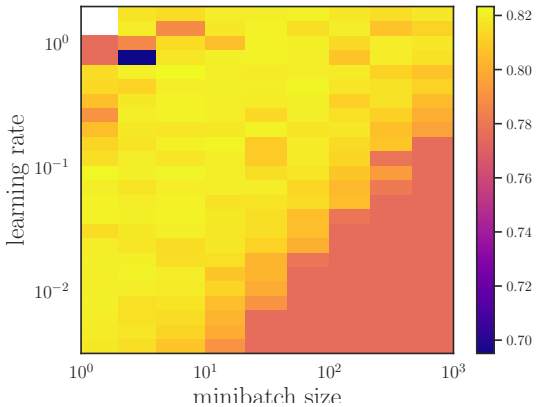

Figure 5: Heatmap showing the accuracy of the Neural Network for different values of the minibatch sizes and learning rates.

*Table 1: Table of parameters that gave the highest **AUC** in the classification problem. Learning rate γ, minibatch size m, accuracy acc and AUC is shown for Logistic Regression and Neural Network.*

Method	γ	m	acc	AUC
Logistic regression	0.15	870	0.817	0.765
Neural network	0.035	1	0.821	0.782

*Table 2: Table of parameters that gave the highest **accuracy** in the classification problem. Learning rate γ, minibatch size m, accuracy acc and AUC is shown for Logistic Regression and Neural Network.*

Method	γ	m	acc	AUC
Logistic regression	0.0069	1645	0.820	0.758
Neural network	0.52	4	0.823	0.777

Table 3: Fraction of true and false negatives and positives for Logistic regression and Neural Network applied on the classification problem, when optimal parameters are used.

Method	Positive		Negative	
	True	False	True	False
Logistic regression	432	210	3964	776
Neural network	456	215	3959	752

3.2 Verification by Comparison to Scikit-Learn

3.3 Neural Networks for Regression on Franke's Function

Looking at Figure 6, we show a heatmap for MSE as a function of minibatch size and learning rate. We observe a zone of low MSE for m=1 and $\gamma \approx 10^{-2}$. Additionally, for approximate ranges $m \in [10^2, 10^3]$ and $\gamma \in [10^{-4}, 10^{-2}]$, in the lower right corner, we observe a zone where the MSE is an order of magnitude higher or more.

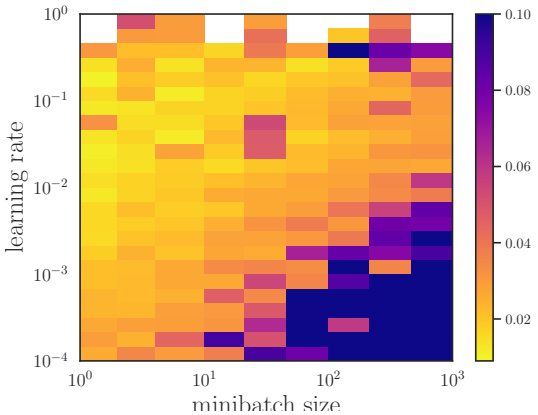

Figure 6: Heatmap showing the MSE of the Neural Network applied on Franke's function for minibatch sizes $m \in [1, 1000]$ and learning rates $\gamma \in [10^{-4}, 1]$. MSE decreases from blue to yellow.

4 Discussion

5 Conclusion

We studied classification and regression using a neural network and logistic regression algorithm. For classification we used the 2005 Taiwan credit card data, and for regression we fitted the Franke function. In both cases, the

Table 4: The minimum MSE achieved when applying different regression models and Neural Network on Franke's function.

Method	MSE
Ordinary Least Square	0.017
Ridge regression	0.015
Lasso regression	0.021
Neural Network	0.011

neural network was compared to other methods: logistic and linear regression, respectively. To compare methods we used the accuracy and ROC AUC score functions for classification, and R2 and MSE for regression. For classification, we found that the neural network gave the highest accuracy of HER, slightly higher than logistic regression, which had a accuracy of HER. As for the ROC AUC score, the neural network also performed better with a score of HER. In general, the LR algorithm seemed to converge faster compared to the neural network, although no formal analysis was made.

As for regression, we found that the MSE

References

Morten Hjort-Jensen. Data Analysis and Machine Learning Lectures: Optimization and Gradient Methods. 2019a.

Morten Hjort-Jensen. Data Analysis and Machine Learning: Logistic Regression. 2019b.

Eina B. Jørgensen, Anna Lina P. Sjur, and Jan-Adrian H. Kallmyr. Using linear regression for fitting terrain data. 2019.

Michael A. Nielsen. *Newral Networks and Deep Learning*. Determination Press, 2015.

Shui-Hua Wang, Preetha Phillips, Yuxiu Sui, Bin Liu, Ming Yang, and Hong Cheng. Classification of alzheimer's disease based on eight-layer convolutional neural network with leaky rectified linear unit and max pooling. *Journal of medical systems*, 42(5): 85, 2018.

I-Cheng Yeh and Che-hui Lien. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2):2473–2480, 2009.