FYS-STK4155 - Project 2

Classification and Regression, from linear and logistic regression to neural networks

Eirill Strand Hauge – Anders Julton

(Dated: November 15, 2019)

The aim of this project was to study and compare logistic and linear regression algorithms to a multilayer perceptron neural network. The logistic comparison was done by studying a binary classification problem. The classification problem was default payments of customers in Taiwan, with 23 explanatory variables. For logistic regression the methods standard gradient descent, stochastic gradient descent, with and without mini-batches and with momentum, and ADAM optimizer were implemented. The cross-entropy was used as the cost function. The methods were compared by calculating the area ratio of the cumulative gains plot. All the methods yielded approximately the same area ratio score, ranging from 0.532 for standard Gradient Descent to 0.545 for Stochastic Gradient Descent with momentum and mini-batches. The multilayer perceptron neural network was also compared to the logistic regression methods using the sigmoid, ReLU, PReLU and the hyperbolic tangent as activation functions. Of the four functions, sigmoid yielded the overall best results. The artificial neural network performed better than standard logistic regression on the classification problem, when using the sigmoid as activation function, with an area ratio of 0.555.

For the linear regression comparison we continued our study [*Linear Methods for Regression*, 2018] of the Franke's function. MSE was used as cost function in the MLP neural network. For the neural network using ReLU as activation function yielded the best result, with a R^2 score of 0.931, where our previous best R^2 score in the study of linear regression was 0.956, using the Ridge algorithm.

I. INTRODUCTION

The proverb "The right tool for the right job" is as relevant for a machine-learning programmer as it is for a carpenter. Knowledge of which algorithm and method that is better suited for the problem at hand, and why, can have significant effect on both results and computational speed. In this project we will compare two such methods: Regression algorithms and Neural Networks. We will compare the neural network to both logistic and linear regression algorithms.

The logistic regression comparison will be based on the work done by I-Cheng Yeghand Che-hui Lien [15] where they studied the case of customers' default payments in Taiwan. We will limit ourselves to compare different gradient methods in logistic regression and a MLP (multilayer perceptron) neural network by their ability to predict binary classification of default payment based on 23 explanatory variables. As the data is heavily biased towards non-risky customers ($\approx 88\%$), we will use the area ratio in the cumulative gains chart, instead counting number of correct predictions, as a measurement of predictive strength. We will also examine how down and up sampling of the original data effects our model's predictive capabilities. For the multilayer perceptron neural network the different activation functions in the hidden layers will be compared.

In our previous paper [5], we studied linear regression and its application to machine learning. Specifically we examined different algorithms to reproduce the Franke's function. In this project we will extend the neural network to handle linear regression algorithms and compare results with those attained previously. Again, different activation functions in the hidden layers will be compared.

In the classification case we expect the artificial neural network to yield better results than the regression algorithms. This based on the fact that a artificial neural network aims to emulate a human brain, where one can argue that the human brain has evolved to become a highly functional classification machine.

How the network fairs when it comes to the linear regression comparison is less certain. While the network might perform close to or as good as the regression algorithms, we believe the negative aspect of the added complexity with the network to outweigh potential differences.

We will start by introducing the concepts of binary logistic regression and several variations of the gradient decent, followed by an introduction to multilayer perceptron neural networks. A description of the implementation will then given, before presenting and then discussing the results of the project.

II. METHOD

Before introducing the methods used in this project, a small note on the choices in notation should be clarified. Matrices will be denoted with hats and vectors will be written in bold. This means that $\hat{\mathbf{A}}$ is understood to a matrix, while $\mathbf{A}_i$ and $\mathbf{b}$ are vectors. Lastly, A_{ij}, b_i and c are scalars.

A. Logistic Regression

The following section featuring logistic regression is based on the lecture notes of Morten Hjort-Jensen [6], the book *The Elements of Statistical Learning* [4, p. 119

- 121] and Aurélien Géron's descriptions of logistic regression from hos book [3, p. 137 - 142].

Logistic regression is a linear method for classification. The discussion of logistic regression assumes that the reader is familiar with linear regression.

Much like linear regression, logistic regression computes a weighted sum of the input features, $\boldsymbol{\theta}^{\boldsymbol{T}}\hat{\mathbf{X}}$, where $\hat{\mathbf{X}}$ is a $N \times M$ matrix consisting of the input data and $\boldsymbol{\theta}$ is a vector of length M containing the calculated weights. Each instance of input data $\mathbf{X}_i$ will contain M input features, with typically $X_{i0} = 1$, such that θ_0 is the intercept value. Instead of predicting the returned value of a function, logistic regression computes the probability of belonging to a certain category. There can be any number of categories, but we will limit our discussion to binary classification.

Since there are only two categories, it suffices to predict whether the input belongs to *one* of the categories. The probability is calculated by sending the weighted sum through a perceptron function. In the binary case, the sigmoid function is used for this purpose, which is given by

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

which has an output range of $\sigma(x) \in (0, 1)$.

Logistic regression is so-called supervised learning, meaning that the training set must also provide a *solution*-vector, $\mathbf{y}$ of length N. In the binary case there are only two possible values, $y_i \in \{0, 1\}$, corresponding to data input $\mathbf{X}_i$ belonging ($y_i = 1$) or not belonging ($y_i = 0$) to the class in question.

The probability of the i'th data instance, $\mathbf{X}_i$, belonging to the class in question, i.e $y_i = 1$, is given by

$$p(y_i = 1|\mathbf{X}_i, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^{\boldsymbol{T}}\mathbf{X}_i), \tag{2}$$

and the corresponding probability of belonging to the *other* class is

$$p(y_i = 0|\mathbf{X}_i, \boldsymbol{\theta}) = 1 - p(y_i = 1|\mathbf{X}_i, \boldsymbol{\theta}). \tag{3}$$

The same equations in vector notation are then

$$p(y = 1|\hat{\mathbf{X}}, \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^{\boldsymbol{T}}\hat{\mathbf{X}}) \tag{4}$$

$$p(y = 0|\hat{\mathbf{X}}, \boldsymbol{\theta}) = 1 - p(y = 1|\hat{\mathbf{X}}, \boldsymbol{\theta}). \tag{5}$$

In binary regression, only one set of θ values are needed. In the case of K classes, K sets of θ values are computed. The perception function is then no longer the sigmoid function, but the so-called *softmax* function. For each given set of input features, $\mathbf{X}_i$, a probability will be calculated for each class, and $\mathbf{X}_i$ will be predicted to belong in the class which yielded the highest probability.

The predicted values in the binary case are then given by

$$\tilde{y}_i = \begin{cases} 1, & p(y_i = 0|\mathbf{X}_i, \boldsymbol{\theta}) \geq 0.5 \\ 0, & p(y_i = 0|\mathbf{X}_i, \boldsymbol{\theta}) < 0.5 \end{cases}. \tag{6}$$

In order to learn how to classify correctly given the input data, logistic regression minimizes the cross-entropy with respect to $\boldsymbol{\theta}$. The cross-entropy for the binary case is given by

$$J(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \left[y_i \log p_i + (1 - y_i) \log (1 - p_i) \right], \tag{7}$$

where the prediction value is given a short notation $p_i \equiv p(y_i = 1|\mathbf{X}_i, \boldsymbol{\theta})$. The partial derivative with respect to the weight of the j-th input feature is

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = -\sum_{i=1}^{N} X_{i,j} \left(y_i - p_i \right). \tag{8}$$

A more compact form, using vector notation, is

$$\frac{\partial}{\partial \boldsymbol{\theta}} J(\boldsymbol{\theta}) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = -\hat{\mathbf{X}}^T \Big(\mathbf{y} - p(y = 1|\hat{\mathbf{X}}, \boldsymbol{\theta}) \Big). \tag{9}$$

To find a minima for the cross-entropy in eq. (7), several different approaches can be used. In this project, we implemented several versions of Gradient Descent, as well as the ADAM optimizer.

B. Optimization and Gradient Methods

The descriptions below of the variations of the gradient decent, momentum and mini-batches are all based on lecture notes by Morten Hjort-Jensen [8] and Aurélien Géron's book [3, p. 113 - 123].

1. *Gradient Descent*

The method of Gradient Descent uses the negative gradient to find the fastest way to a minima of a function. This gives a recursive formula to optimize the θ values,

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{10}$$

where $\eta > 0$ is the step length, also called learning rate. The learning rate should be chosen to be a small value to ensure that the gradient decent moves closer to a minima for each iteration.

The initial guess determines how fast the gradient descent will converge. Should the cross-entropy contain local minimas in addition to the global minima, the Gradient Descent method could converge to a local minima instead of the desired global minima. Which minima the method will converge towards, depends on the initial guess. This means that if the landscape of the cross-entropy is a very uneven, the Gradient Descent may converge toward a local minima lying far above the global minima, should the initial guess be "unlucky". A variation of the Gradient Descent, called Stochastic Gradient Descent, addresses a few of the limitations of the classical Gradient Descent.

2. Stochastic Gradient Descent with Momentum

The method of the Gradient Descent uses the whole training set in each iteration. To avoid this, the Stochastic Gradient Descent picks a random instance of the training set in each iteration. This greatly reduces the memory usage and computations per iteration. The iterative scheme then becomes

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \mathbf{X}_r^T \left(p_r - y_r\right), \tag{11}$$

where r is a random index.

Unlike the Gradient Descent, which approaches the minima quite monotonously, the Stochastic Gradient Descent will be far less regular. The cost function will decrease on average, but the value will have a tendency to bounce. This means that the Stochastic Gradient Descent will not stabilize at the minima, but rather "jump around" close the minima.

Should the minima that the Gradient Descent methods converge to, indeed be the global minima, the Gradient Descent will achieve a more accurate answer than the Stochastic Descent. However, if the cost function happens to be very uneven, containing several local minimas, the Stochastic Gradient Descent will be able to escape a smaller local minima, and potentially find a "better" local minima.

Momentum can be added to the iterative scheme. A term proportional with the change of the previous iteration is added, $\Delta\boldsymbol{\theta}_n = \boldsymbol{\theta}_n - \boldsymbol{\theta}_{n-1}$. The formula of the Stochastic Gradient Descent with Momentum is then

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \gamma\Delta\boldsymbol{\theta}_n - \eta\mathbf{X}_r^T(p_r - y_r), \tag{12}$$

where γ is the momentum parameter, $0 \leq \gamma \leq 1$. The momentum keeps track of the direction of movement in parameter space and helps smooth out the oscillations towards convergence one normally has with stochastic gradient descent methods.

3. ADAM optimizer

The ADAM optimizer uses an average of the first and second moment of the gradient to optimize the learning rate of different parameters. [10]

We therefore start keeping track of the average first moment, $\mathbf{m}_n$ and average second moment $\mathbf{s}_n$. These are given by the recursive formulas

$$\mathbf{m}_n = \beta_1\mathbf{m}_{n-1} + (1-\beta_1)\boldsymbol{\nabla}_{\boldsymbol{\theta}}J(\boldsymbol{\theta}) \tag{13}$$

$$\mathbf{s}_n = \beta_2\mathbf{s}_{n-1} \ + (1-\beta_2)[\boldsymbol{\nabla}_{\boldsymbol{\theta}}J(\boldsymbol{\theta})]^2, \tag{14}$$

where the β values determines the averaging time. The modified terms for the average first and second moment of the gradient are

$$\tilde{\mathbf{m}}_n = \frac{\mathbf{m}_n}{1-\beta_1^n} \tag{15}$$

$$\tilde{\mathbf{s}}_n = \frac{\mathbf{s}_n}{1-\beta_2^n}. \tag{16}$$

The final scheme for the ADAM optimizer is given by

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta\frac{\tilde{\mathbf{m}}_n}{\sqrt{\tilde{\mathbf{s}}_n} + \epsilon}, \tag{17}$$

where η is again the learning rate and ϵ regulates to prevent divergences.

4. Mini-Batches

The methods of Gradient Descent and the ADAM optimizer are computationally expensive. To speed up the calculations, mini-batches are introduced. In each iteration, a subset of the training data is used to improve the θ-values. The size of the subset, or mini-batch as it is called, should be large enough to gain a speed up from the Stochastic Gradient Descent, but small enough to avoid loss of speed due to high demand of memory.

When using mini-batches, the train data set should frequently be shuffled to avoid the model overfitting due to possible relationships the used sorting of the data might have.

C. Multilayer Perceptron Neural Network

The following introduction given on Artificial Neural Networks is based on Morten-Hjort Jensen's lecture notes [7], MIT professor Patrick H. Winston's lectures on artificial neural networks [14] and Michael Nielsen's explanation of the back propagation [12].

The animal brain consists of a vast number of interconnected neurons, a neural network. A neuron in the network can send chemical or electrical current signals to connected (target) neurons. If the magnitude of the signal is above a certain limit, the neuron is activated, or "fires". Artificial neural networks mimic this behavior, in order to *learn* from a set of training data, as humans (and other animals) learn from experiences. As slightly hinted towards, the artificial neural network is a type of machine learning.

The artificial network is divided into layers, where each layer contains an arbitrary number of artificial neurons, also often called nodes. Each node is connected to, and sends signals to, all the nodes in the succeeding layer. Each connection has an associated weight variable, indicating the strength of the connection.

A multilayer perceptron will consist of an input layer, at least one hidden layer and an output layer. The total amount of layers counts the hidden layers and the output layer, but not the input layer. This gives a total of $L \geq 2$ layers in a multilayer perceptron neural network. The input layer consists simply of the features of the input data. The output layer will depend on what the model will try to predict.

The activation of each artificial neuron depends greatly on the activity in the artificial neurons in the previous

layers. In the multilayer perceptron neural network, the artificial neurons in the same layer do not communicate. The activation is calculated in the *feed-forward propagation*, while the weight variables are modified in the *back propagation* in order to improve predictions. This is repeated until the desired prediction score is achieved, or the maximal number of iterations is reached. We will start by explaining the feed-forward propagation before tackling the back propagation.

1. The Feed-Forward Propagation

The hidden layers contain activation functions. The activation functions f must be non-linear, bounded and monotonically increasing continuous functions. All the nodes in a given layer uses the same activation function, but the different layers may utilize different activation functions. We will therefore denote the activation functions with an super script $l = 1, 2, .., L$, such that f^l is the activation function of the l'th layer. The Sigmoid function (1) is commonly used, but a collection of activation functions (35 - 38) can be found in the appendix. Before going into what the output of the hidden layers look like, we first have to circle back to the previously mentioned weight variable.

We will introduce a matrix representation of the weights between the nodes. Each layer will have its own weight matrix, $\hat{\mathbf{W}}^l$, where W^l_{qk} corresponding to the weight variable associated with the connection between the k'th node in the l'th layer and the q'th node in the $(l-1)$'th layer. The output, or activation, of the first hidden layer is then

$$\hat{\mathbf{a}}^1 = f^1\left(\left(\hat{\mathbf{W}}^1\right)^T \hat{\mathbf{X}} + \mathbf{b}^1\right), \tag{18}$$

where $\mathbf{b}^l$ is a vector containing the biases, such that b^1_k is the bias of the of the k'th node in the first layer. Each layer uses the output of the previous layer as part of the input, giving a recursive formula for the activation of the l'th layer

$$\hat{\mathbf{a}}^l = f^l\left(\left(\hat{\mathbf{W}}^l\right)^T \hat{\mathbf{a}}^{l-1} + \mathbf{b}^l\right). \tag{19}$$

For simpler notation, we will now give a name for the input of each layer

$$\hat{\mathbf{z}}^l \equiv \left(\hat{\mathbf{W}}^l\right)^T \hat{\mathbf{a}}^{l-1} + \mathbf{b}^l, \tag{20}$$

such that $\hat{\mathbf{a}}^l = f^l(\hat{\mathbf{z}}^l)$. The activation of the k'th node in the l'th layer is

$$\mathbf{a}^l_k = f^l(\mathbf{z}^l_k) = f^l((\mathbf{W}^l)^T_k \hat{\mathbf{a}}^{l-1} + b^l_k). \tag{21}$$

Note that the sum over the previous layers nodes is $\left(\hat{\mathbf{W}}^l\right)^T_k \hat{\mathbf{a}}^{l-1} = \sum_{q=1}^{Q} W_{qk}\mathbf{a}^{l-1}_q$, where Q is the number of nodes in the previous layer $(l-1)$. The activation value of the k'th node in the l'th layer that the i'th instance of data set induces, is then

$$\left[a^l_k\right]_i = f^l(\left[z^l_k\right]_i) = f^l((\mathbf{W}^l)^T_k \left[\mathbf{a}^{l-1}\right]_i + b^l_k). \tag{22}$$

The weight matrix $\hat{\mathbf{W}}$ is independent of the input instance, while each instance will yield a separate activation value. The sum is in this case $(\mathbf{W}^l)^T_k \left[\mathbf{a}^{l-1}\right]_i = \sum_{q=1}^{Q} W_{qk}[a^{l-1}_q]_i$.

The process of calculating all the activation values a^l_k is known as the *feed-forward propagation*. After the feed-forward propagation, the weights will need to be adjusted in order to fit, or *learn*, from the data in what is called the *back propagation*.

2. Back Propagation

Like in both the linear and logistic regression, the artificial neural network aims to minimize a cost function, which will depend on the type of problem at hand. The cost functions will shortly be presented, but first we need to consider how many nodes the output layer will contain.

In the case of an artificial neural network solving a classification problem consisting of K classes/categories, the number of nodes in the output layer should be K. Each node in the output layer corresponds to a class, meaning that the activation value $\left[a^L_k\right]_i$ is the probability of the i'th input instance to belong in the k'th category.

The binary case is the exception confirming the rule, where there is only need for *one* node in the output layer. The probability value of the i'th instance to belong to the class in question is $p(y=1|\mathbf{X}_i, \hat{\mathbf{W}}) = \left[a^L_1\right]_i$, and the prediction value will be the same as for binary logistic regression, eq. (6). The probabilities in vector notation will therefore be

$$p(y=1|\hat{\mathbf{X}}, \hat{\mathbf{W}}) = \mathbf{a}^L_1 \tag{23}$$

$$p(y=0|\hat{\mathbf{X}}, \hat{\mathbf{W}}) = 1 - \mathbf{a}^L_1. \tag{24}$$

In the case of a the binary logistic neural network, we again implemented the cross-entropy function, but with an added regularization parameter λ to constraint the size of the weights

$$J(\hat{\mathbf{W}}^L) = -\sum_{i=1}^{N}\left[y_i \log a^L_i + (1-y_i)\log\left(1-a^L_i\right)\right] + \lambda \sum_{ij} W^2_{ij}. \tag{25}$$

An artificial neural network to solve linear problems was also implemented in this project. The neural network should then mimic a function $g(x) = y$. The input data will then be a design matrix, as explained in previous work. [5] Like in the binary case, only one node should

be used in the output layer. The predicted value of $g(x_i)$ is then $\left[a_1^L\right]_i$.

The mean squared error was used as the cost function for the linear neural network

$$MSE(\hat{\mathbf{W}}) = \frac{1}{N}\sum_{i=1}^{N}(\left[a_1^L\right]_i - y_i)^2. \tag{26}$$

Now that we have both cost functions, we will start to generalize how to compute the gradients of the cost functions. First, we need to see that the gradient of both cost functions with respect to $\mathbf{a}_1^L$ will only differ by a scalar. Since we are interested in the direction of the gradient, we will make a common expression for the gradient of the cost function $C(\mathbf{W})$

$$\boldsymbol{\nabla}_{\mathbf{a}^L} C = \mathbf{a}_1^L - \mathbf{y}. \tag{27}$$

To find the gradients, we need an expression for the error of the k'th node in the l'th layer. These are denoted δ_k^l, and are set to be

$$\delta_k^l \equiv \frac{\partial C}{\partial \mathbf{z}_k^l} = \frac{\partial C}{\partial \mathbf{a}_k^l}\frac{\partial f^l(\hat{\mathbf{z}}^l)}{\partial \hat{\mathbf{z}}^l}. \tag{28}$$

For the output layer, the error of the only node will be

$$\delta_1^L = \frac{\partial C}{\partial \mathbf{a}_1^L}\frac{\partial f^L(\mathbf{z}_1^L)}{\partial \mathbf{z}_1^L}. \tag{29}$$

Since the output layer only has one node, the error in the layer, δ^L will be

$$\delta^L = \delta_1^L = \boldsymbol{\nabla}_{\mathbf{a}^L} C \odot \frac{\partial f^L(\mathbf{z}_1^L)}{\partial \mathbf{z}_1^L}, \tag{30}$$

where $\odot$ is the Hadamard product.

For the error in the hidden layers we get a recursive scheme

$$\delta^l = ((\hat{\mathbf{W}}^{l+1})^T \delta^{l+1}) \odot \frac{\partial f^l(\hat{\mathbf{z}}^l)}{\partial \hat{\mathbf{z}}^l}, \tag{31}$$

going *backwards* in the propagation.

Finally, we get the expression for the gradients of the cost function. The gradient of the cost function with respect to the weights of the nodes in the l'th layer is

$$\boldsymbol{\nabla}_{\mathbf{W}^l} C = \hat{\mathbf{a}}^{l-1}\delta^l, \tag{32}$$

and the gradient of the cost function with respect to the bias of the nodes in the l'th layer is

$$\boldsymbol{\nabla}_{\mathbf{b}^l} C = \delta^l. \tag{33}$$

Now that the gradients are known, the weights and the biases can be updated such that the cost function will get closer to a minima. In the implementation, we used the gradient descent for this purpose.

D. Asserting the Methods

For the logistic regression we used the cumulative gain chart to examine the classification accuracy of both the logistic regression algorithms and the neural network [1]. The area ratio, defined as

$$\text{Area ratio} = \frac{\text{area between model and baseline curve}}{\text{area between best and baseline curve}}, \tag{34}$$

where the definitions can be seen in figure (3), were used to determine the predictability strength of the gradient descent methods and the neural network.

For the Franke's function we again used MSE and R^2 scores, as described in our previous project [5].

III. IMPLEMENTATION

The full implementation can be found at `https://github.com/andersjulton/FYS-STK4155/tree/master/Project_2`

The implementation was done in the programming language Python. All figures found under results were produced using the *matplotlib* library [9].

A. Credit Card Data

The credit card data was read and sorted using the *pandas* package. Using the package *sklearn*[13], categorical features (sex, marriage, education) was one-hot encoded, while features containing continuous features (age, bill statement, payment history, amount of credit given) was standardized. Uncategorized outliers were removed. The columns containing repayment status was divided into two columns each. One containing values 1-7 and the other -2, -1 and 0. -2 and 0 were unspecified categories, but made up roughly 87% of the data, so they were kept as a class and one-hot encoded along with -1.

B. Resampling

Down and up sampling was implemented with the *imblearn* package [11].

For down sampling, *RandomUnderSampler* was used, which randomly removes rows belonging to the majority class of the original data set. The number of rows removed is given by a user-defined ratio N_{min}/N_{maj}, the desired ratio between the minority class and the majority class. In our binary classification problem class 1 was the minorty and class 0 was the majority, with original ratio $N_1/N_0 \approx 0.12$.

For up sampling, SMOTE (Synthetic Minority Oversampling Technique) was implemented, as presented in Berry and Linoff's book on mastering data mining [2].

C. Logistic Regression

The methods for regression were implemented in a class hierarchy. An abstract parent class was implemented to contain most of the supporting methods, such as cost function, plotting and accuracy measurements. The various gradient descent methods were implemented as child classes of the parent class.

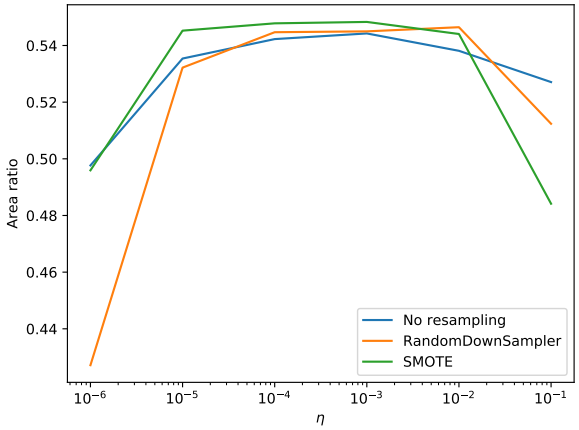

FIG. 1: Comparison of random down sampling, SMOTE and no resampling, using ADAM optimizer.

D. Multilayer Perceptron Neural Network

The Multilayer Perceptron (MLP) neural network was implemented in a class hierarchy. A parent class was implemented containing functions for initializing the hidden layers, weights and biases and the feed-forward and back propagation functions. The regression methods, with their respective cost functions, were implemented as child classes of the parent class mentioned above.

A separate class was made for the activation functions and their derivatives.

IV. RESULTS

A. Study of Credit Card Data

1. *Logistic Regression*

To test the effect of down and up sampling, a run using ADAM optimizer was done for the RandomUnderSampler method, SMOTE and no resampling, for various values of η. The results are shown in figure (1). The three methods are seen to perform approximately equally well in the region $\eta \in (10^{-5}, 10^{-2})$. Figure (2) shows area ratio as a function of down sampling ratio for the RandomUnderSampler method, using ADAM optimizer. Based on this, and that down sampling led to faster computations, RandonUnderSampler was used in all runs with logistic regression, with $N_1/N_0 = 1$.

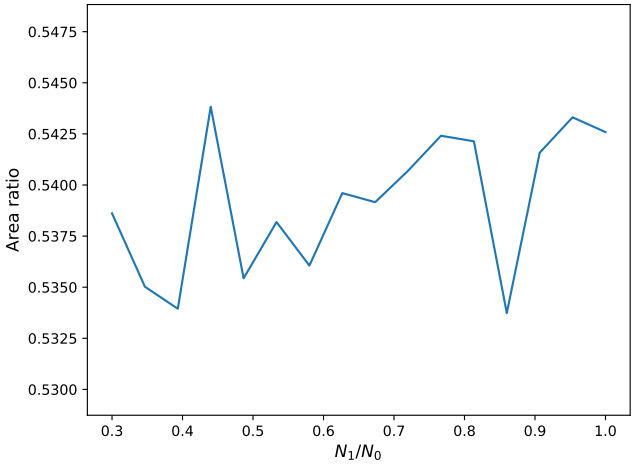

FIG. 2: Plot of area ratio as a function of down sampling ratio, using ADAM optimizer.

A grid search of the learning rate η and hyperparameter λ was done with the various implementation of gradient descent for the logistic regression. The resulting scores from the best combination of η and λ are shown in table I.

TABLE I: Table of area ratio scores for logistic regression on credit card data using Gradient Descent, Stochastic Gradient Descent with momentum (GDM), Stochastic Gradient Descent with momentum and mini-batches (GDM MB) and ADAM optimizer.

Method	Area ratio
Gradient Descent	0.532
GDM	0.536
GDM MB	0.545
ADAM	0.544

Figures (3)-(6) shows the cumulative gain plots used to calculate the area ratio of the different methods.

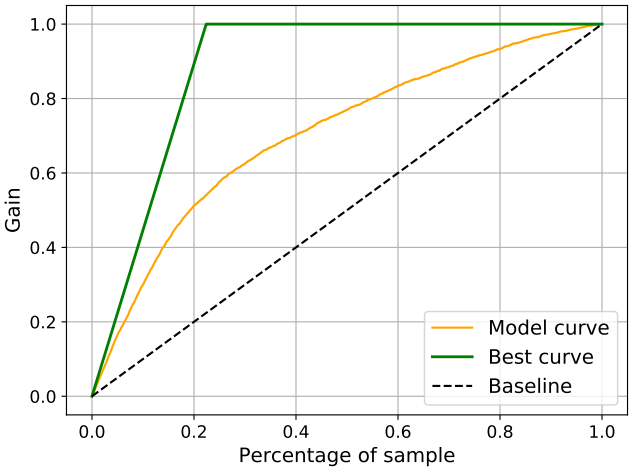

FIG. 3: Cumulative gain plot with gradient descent.

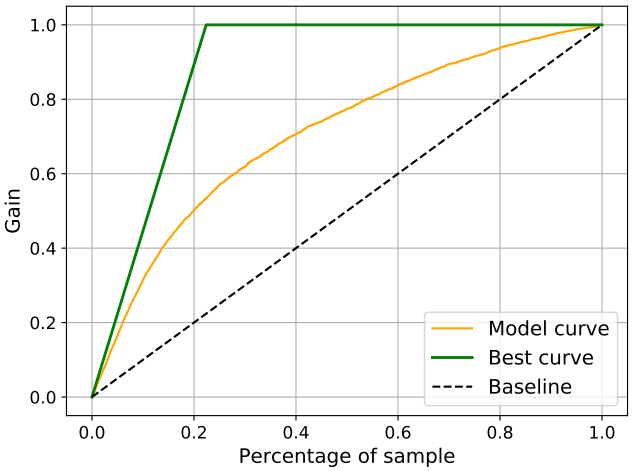

FIG. 4: Cumulative gain plot with stochastic gradient descent with momentum and without mini-batches.

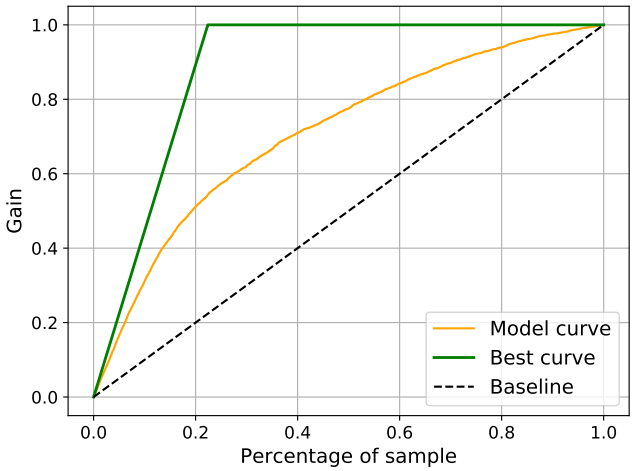

FIG. 5: Cumulative gain plot with stochastic gradient descent with momentum and mini-batches.

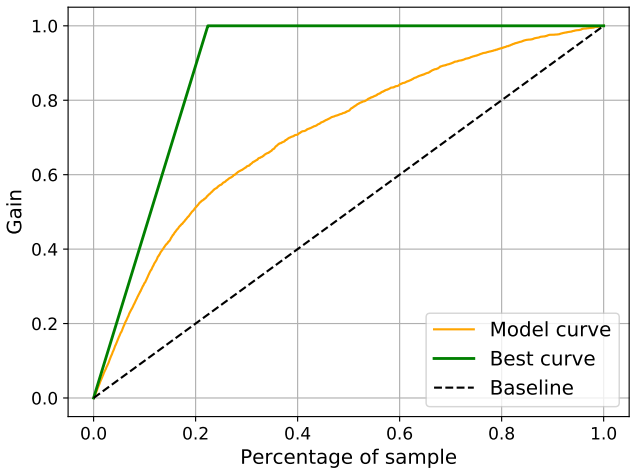

FIG. 6: Cumulative gain plot with ADAM optimizer.

2. *Artificial Neural Network*

A grid search of the learning rate η and hyperparameter λ was done for the various activation functions. The resulting scores from the best combination of η and λ are shown in table II, where combination that gave the highest area ratio score was deemed the best combination.

TABLE II: Table of area ratio scores for the neural network on credit card data using the sigmoid, ReLU, PReLU and tanh as activation functions in the hidden layer. The runs were done with 50 epochs and 500 as batch size. 2 hidden layers were used with 5 and 10 neurons in the hidden layers, respectively. In all runs the sigmoid was used as the output activation function.

Method	Area ratio
Sigmoid	0.550
ReLU	0.519
PReLU	0.520
tanh	0.512

As an example, figure (7) shows a plot of the grid search results for the sigmoid function as the acitvation function in the hidden layers, where the values in the squares are the area ratio for the corresponding combination of η and λ. Corresponding grid search results for the ReLu, PReLU and the hyperbolic tangent as the activation function in the hidden layers can be found in figures (15 - 17) in the appendix.

FIG. 7: Grid search of η and λ with sigmoid as hidden layer activation function. Area scores are shown in the squares.

Figure (8) shows a plot of area ratio score as a function of number of neurons in the hidden layer.

Figure (9) shows a histogram of area ratio score for 100 runs with equal initial values.

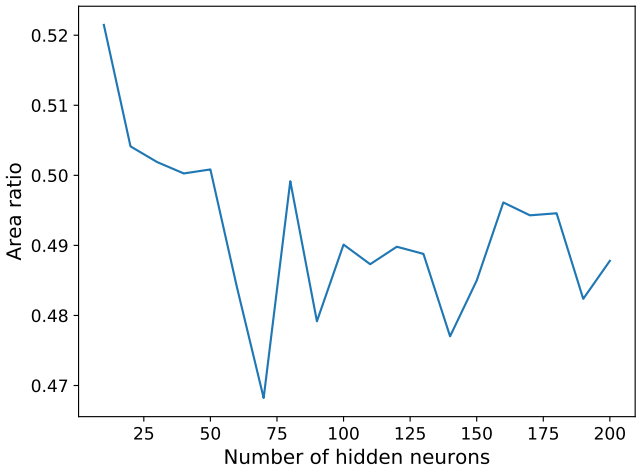

FIG. 8: Plot of area ratio as a function of neurons in hidden layer. One hidden layer was used, with sigmoid as activation function in both hidden and output layer.

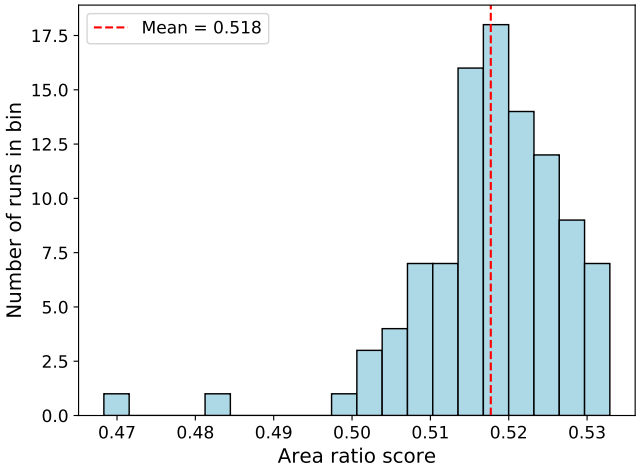

FIG. 9: Histogram of area ratio scores of 100 runs with equal initial values. The runs were done with 50 epochs, batch size of 500, $\eta = 0.01$, $\lambda = 0.1$, 20 hidden neurons with one hidden layer. Sigmoid was used as activation function in both hidden and output layers.

B. Franke's Revised

A grid search of the learning rate η and hyperparameter λ was done for the Franke's function using the different activation functions. Table III shows the results from the combination of η and λ that gave the highest R^2 score.

A study of potential overfitting is shown in figures (10) and (11), where the MSE for the test and training data are plotted as a function of polynomial degree and number of epochs respectively. 5 runs per polynomial degree and number of epochs were done with identical initial

value, where the MSE scores are the mean of the 5 runs.

Figure (12) shows a 3D-plot of the model fit using ReLU as activation function. Figure (13) and (14) shows plot of the test data and training data respectively.

TABLE III: MSE and R^2 scores for the activation functions used to fit the Franke's function. The same activation function was used in all hidden layers and the output layer. 4 hidden layers were used with 100 neurons in each layer, with 100×100 number of data points for training.

Activation function	MSE	R^2
Sigmoid	0.074	-0.016
ReLU	0.005	0.931
PReLU	0.005	0.929
tanh	0.024	0.671

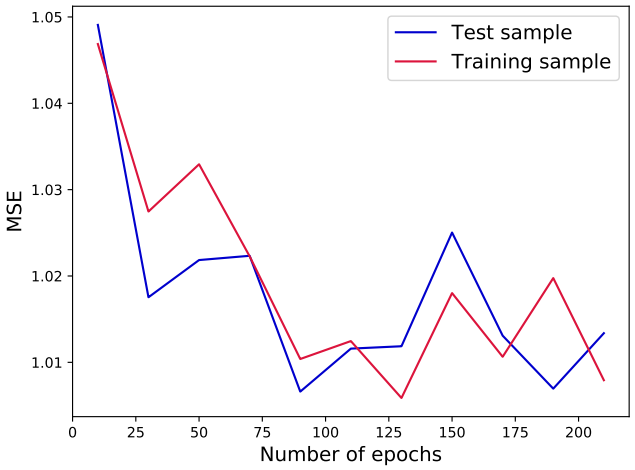

FIG. 11: Plot of training and test error as a function of number of epochs for neural network, using ReLU as activation function.

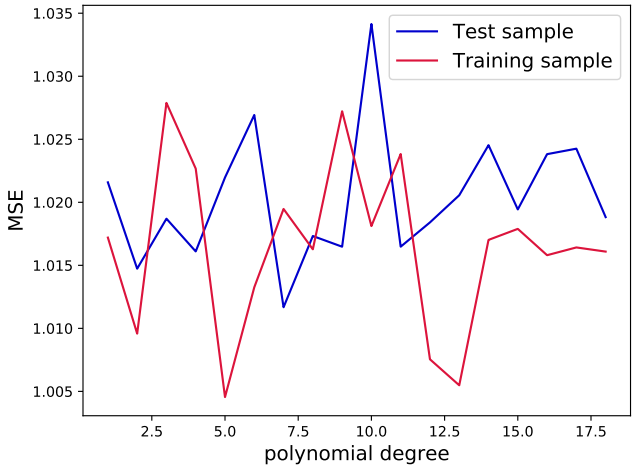

FIG. 10: Plot of training and test error as a function of polynomial degree for neural network, using ReLU as activation function.

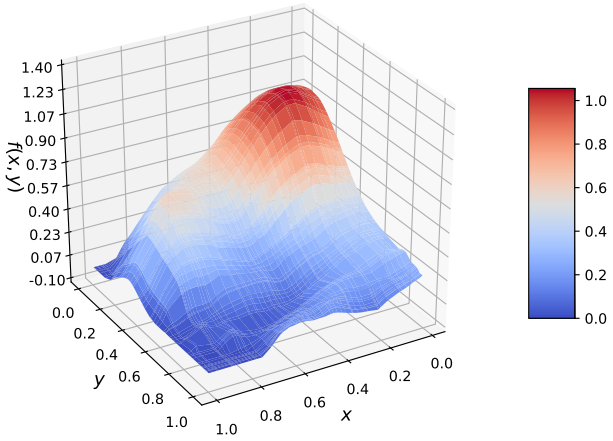

FIG. 12: Plot of model fit of Franke's function using ReLU.

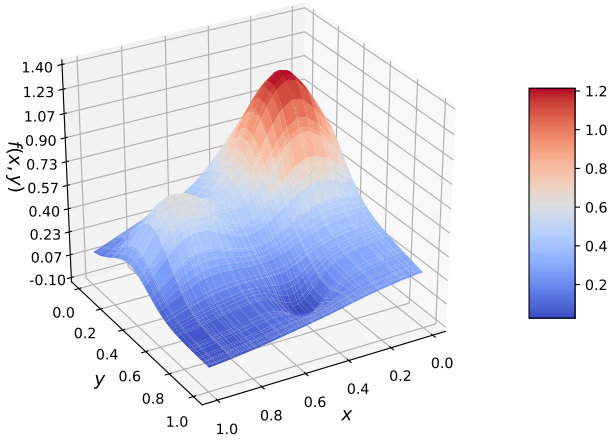

FIG. 13: Plot of test data of the Franke's function

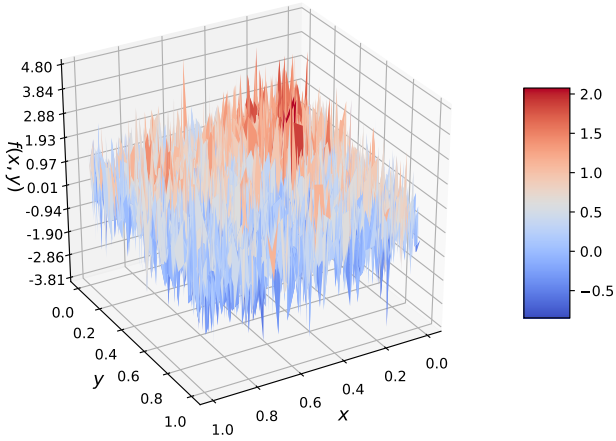

FIG. 14: Plot of noisy training data of the Franke's function

V. DISCUSSION

A. The Classification Problem

From table I we see that all the gradient methods performed equally well in classifying default payments. In table II we see that sigmoid performed better than the three other activation functions that was implemented. The neural network with sigmoid also outperformed the gradient descent methods used in the logistic regression case.

The hyperparameter λ seemed to have little to no discernible effect on the performance of the neural network. From figure 7, $\lambda = 0.1$ is seemingly the optimal choice, but multiple runs showed that whichever λ gave the highest score varied from run to run. To test this further, 100 runs were done with identical initial values. From figure (9) we see the stochastic effect the random initialization of the weights, and possibly the random selection of batches, has on the scores, where the runs shows a clear spread in scores, insinuating many local minima in our cost function. Some outliers were several standard deviations away from the mean. The stochastic effect can possibly again be seen in figure (8). The figure shows a plot of area score as a function of number of neurons in the hidden layer, with the score being a mean of 5 runs per number of neurons. The score seemed to go down as the number of neurons increased, but due to the stochastic effect discussed above we can not conclude this correlation firmly.

To mitigate the stochastic effect when searching for the optimal combination of parameters, the mean of many more runs per combination would have to be calculated, which was deemed out of the time scope for this project.

The learning rate η had a significant effect on the performance of the model. For ReLU and PReLU as activation functions in the hidden layer, a higher value of η can cause the network to classify all the targets in one class. In figure (15) and (16) we see this happen for $\eta = 0.01$. For lower values of η, we see that the network is learning to slowly to converge to a minima.

B. Neural Network and Linear Regression

Revisiting the Franke's function, we see from table III that the ReLU functions clearly outperforms both sigmoid and tanh. Since the output values of the Franke's function can be larger than 1, it is not surprising that the sigmoid and tanh were less optimal for this task, with max output values of 1.

From figure (10) and (11) we see no sign of overfitting as was seen for the linear regression algorithms in our previous project. More thorough search with multiple runs to mitigate the stochastic effect might have revealed underlying trends not seen in the figures.

Our network did not manage to outperform our previous best score when using the Ridge regression method, which had a best R^2 score of 0.956. Further fine-tuning and a wider grid-search of the various parameters of the network might have given better results, but the benefit of this is questionable when considering that the implementation of the linear regression algorithms are relatively simpler than the implementation of the neural network.

C. Future Improvements

Future improvements of the implementation could include implementing different functions for the learning rate η instead of holding η as a constant.

For the artificial neural network, further exploring by varying number of layers, nodes per layer, and different

combinations of activation functions in the different layers would be an interesting study.

VI. CONCLUSION

In this project we implemented 4 logistic regression algorithms and a MLP neural network with 4 different activation functions. This to study and compare their ability to predict default payments of credit card users using data from one of the major banks in Taiwan. We also fitted polynomials to the Franke's function using the neural network, and compared the results to our previously attained scores using linear regression methods.

Confirming our hypothesis, the neural network outperformed the logistic regression algorithms in the classification study, with a area ratio score of 0.555, using sigmoid as activation function in both hidden and output layers. The best area ratio score for the logistic regression algorithms was 0.545 using Stochastic Gradient Descent with momentum and mini-batches.

Fitting the Franke's function, we were not able to produce better results using the neural network than those attained previously. The best R^2 score for the network was 0.931 using ReLU as activation function in both hidden and output layers. 0.956 was the previous best R^2 score using the Ridge algorithm.

[1] M.J.A. Berry and G.S. Linoff. *MASTERING DATA MINING: THE ART AND SCIENCE OF CUSTOMER RELATIONSHIP MANAGEMENT*. Wiley India Pvt. Limited, 2008.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[3] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. O'REILLY, 1 edition, 2017.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[5] Eirill Hauge and Anders Julton. Linear methods for regression. `https://github.com/andersjulton/FYS-STK4155/tree/master/Project_1`, 2019.

[6] Morten Hjorth-Jensen. Applied Data Analysis and Machine Learning Lecture Notes — Logistic Regression. `https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html`, 2019.

[7] Morten Hjorth-Jensen. Applied Data Analysis and Machine Learning Lecture Notes — Neural Networks, From the Simple Perceptron to Deep Learning. `https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html`, 2019.

[8] Morten Hjorth-Jensen. Applied Data Analysis and Machine Learning Lecture Notes — Optimization and Gradient Methods. `https://compphysics.github.io/MachineLearning/doc/pub/Splines/html/Splines.html`, 2019.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[11] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

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

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] Patrick H. Winston. Artificial Intelligence — Lecture 12 & 13: Neural Nets. `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/`, 2015.

[15] 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, Part 1):2473 – 2480, 2009.

VII. APPENDIX

A. Collection of Activation Functions

In this section you will find a collection of activation functions used in the layers of the artificial neural networks, explained in the method section. The activation functions are used in the feed-forward propagation, while the derivatives are used in the back propagation. The derivatives of the activation functions are therefore also included.

The Sigmoid function was introduced in the section describing logistic regression as the perceptron function, eq. (1). This function is also commonly used in the layers of an artificial neural network. We will repeat the Sigmoid function, given as

$$\begin{aligned} f(x) = \sigma(x) &= \frac{1}{1+e^{-x}}, \\ f'(x) &= \sigma(x)(1-\sigma(x)). \end{aligned} \tag{35}$$

The range of this function is $f(x) \in (0, 1)$.

The ReLU function is also a commonly used a activation function, which is given as

$$\begin{aligned} f(x) &= \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}, \\ f'(x) &= \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}. \end{aligned} \tag{36}$$

The range of this function is $f(x) \in [0, \infty)$.

A variation of the ReLU, called PReLU, is another activation function. The PReLU function is given by

$$\begin{aligned} f(x) &= \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}, \\ f'(x) &= \begin{cases} \alpha, & x < 0 \\ 1, & x \geq 0. \end{cases} \end{aligned} \tag{37}$$

The range of this function is $f(x) \in (-\infty, \infty)$.

The last activation function in this collection is the hyperbolic tangent, which we know to be

$$\begin{aligned} f(x) &= \tanh(x), \\ f'(x) &= 1 - \tanh^2(x). \end{aligned} \tag{38}$$

The range of this function is $f(x) \in (-1, 1)$.

B. Additional Results: Binary Classification Using Artificial Neural Network

In this section you will find three plots with grid search results for three different acitvation function in the hidden layers, where the values in the squares are the area ratio for the corresponding combination of η and λ. Figure (15) shows the results when PReLU was used as the activation function, in figure (16) ReLU was used and in figure (17) the hyperbolic tangent was used as the activation function.

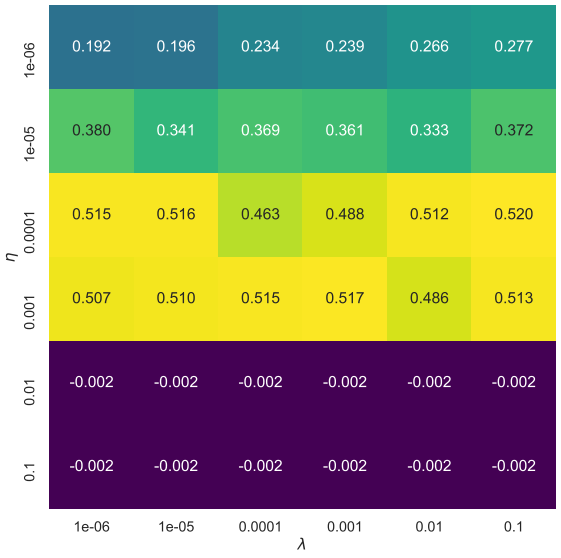

FIG. 15: Grid search of η and λ with PReLU as hidden layer activation function.

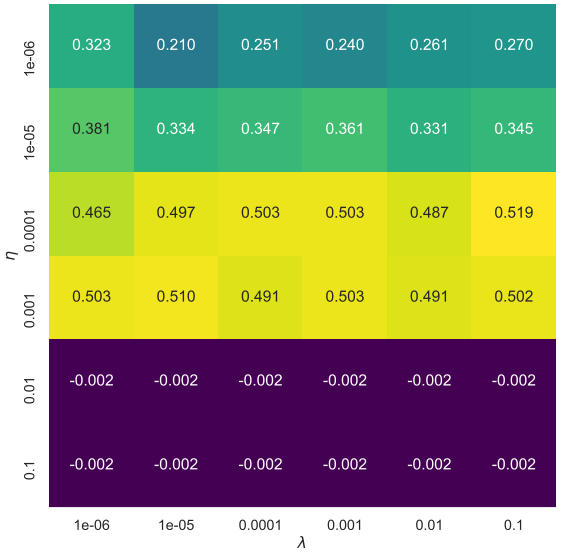

FIG. 16: Grid search of η and λ with ReLU as hidden layer activation function.

FIG. 17: Grid search of η and λ with tanh as hidden layer activation function.