Neural Networks

S. H. Magnússon, G. S. Cabrera, and B. S. Dalen
(Dated: Saturday 9th November, 2019)

A research project on the efficiency and performance of neural networks is presented. Network systems are compared to both stochastic gradient descent for a *logistic case*, as well as the Ridge regression scheme for a *linear regression* case. Taiwanese banking data is used for logistic regression, while Franke's function is used for the linear regression. A central focus of the project is the optimization of hyperparameters λ and η (*l2* regularization and *learning rate*, respectively) in order to produce the best predictions for a feed-forward neural network. It was found that...

I. INTRODUCTION

In the field of statistics, regression methods have been in development for many decades; more recently, computational developments have managed to implement these methods in new ways, allowing us to perform exceptional feats unseen in the past. An *Artificial Neural Network* (ANN) is one such exceptional statistical tool – such a network is an extension of simple linear regression, expanding to have greater dynamic and customized prediction power. ANNs have recently been applied to a phenomenal amount of research in practically *all* fields; examples of this are ANNs with the ability to identify and distinguish between faces (see the work by *H.A. Rowley*, *S. Baluja*, and *T. Kanade* on face detection using neural networks [**?**]), or networks with the ability to predict localized solar radiation to optimize solar panel design (see the work of *A. Mellit*, *M. Menghanem*, and *M. Bendekhis* on sunshine duration and air temperature predictions using neural networks [**?**]).

The project presented aims to implement such statistical methods (ANNs) in an attempt to predict the behaviour of *credit card holders*. More specifically, we are interested in the probability that a given customer will default on his/her credit card debt. For this, a large credit card holder dataset from a Taiwanese bank is used for model training and testing. This dataset has been previously analyzed by *I-Cheng Yeh et al.* [**?**], offering us a good performance benchmark for the methods derived and implemented in the study presented.

Various methods are also applied to this data for comparison: firstly, a method of *stochastic gradient descent* (with mini-batches) is applied to the data. This model provides a good basis of accuracy as it is relatively simple to implement in relation to a neural network. A *multilayer perceptron* model is subsequently built for comparison. This model is the principal type of ANN studied in this paper, as it is a reasonable tradeoff between complexity and performance in the context ANNs. This network is first built around predicting the credit card data, or a so-called *binary classification* case, though the model is later applied to a more general *regression* case. Data is generated for the regression case using *Franke's function*, and the ANN will be compared to the so-called *Ridge* regression scheme. The ridge regression scheme is quite simple and effective, once again providing a baseline which the ANN is expected to outperform.

The report is split into sections designed to raise subjects for discussion and lead the reader to similar conclusions as those presented. A section on *Theory and Algorithms* provides the mathematical background needed for the implementation of the regression schemes presented (and implemented) in the *Method* section. Following this are the *Results*, where the performances of the implemented algorithms are demonstrated and subsequently discussed in the *Discussion* section. The paper is then summarized in a *Conclusion* of the research.

This project is a collaboration between *Steinn H. Magnússon*, *Gabriel S. Cabrera*, and *Bendik S. Dalen*. The code used to produce the results of the study can be found on the following **Project 2 Github Repository**.

II. THEORY AND ALGORITHMS

The theory building up to the ANN expressions involves some simple linear regression method introduction. A similar introduction of the linear regression theory was conducted in a previous project of the semester (G. S. Cabrera and B. S. Dalen's project 1 [**?**], and S. H. Magnússon and S. Håpnes' project 1 [**?**]), though a short reminder of the expressions is appropriate.

A. Linear Regression

Linear regression is a standard method found in statistics used to (approximately) fit some function to a set of data. This is a system of regression which involves the following system for predicting an outcome $\hat{y}$ for some given input/set of features $X = [x_0, x_1, \ldots, x_{p-1}]$:

$$\hat{y}_i = \sum_{j=0}^{p-1} x_{i,j}\beta_j, \tag{1}$$

where p is the number of *features* of the input data X. In the case of the credit card data, these features vary from the age of the credit card holder to his/her marital status. The features p are simply the *inputs* into our system. Rewriting

this in terms of a linear system of equations yields:

$$\begin{aligned}\hat{y}_0 &= \beta_0 x_{0,0} + \beta_1 x_{0,1} + \ldots + \beta_{p-1} x_{0,p-1} \\ \hat{y}_1 &= \beta_0 x_{1,0} + \beta_1 x_{1,1} + \ldots + \beta_{p-1} x_{1,p-1} \\ &\vdots \\ \hat{y}_{n-1} &= \beta_0 x_{n-1,0} + \beta_1 x_{n-1,1} + \ldots + \beta_{p-1} x_{n-1,p-1},\end{aligned}$$

where n is the *sample size* of the data set. In the case of the credit card data, this is the number of people which we have data regarding. This system of equations can be rewritten in matrix-vector multiplication form as:

$$\hat{y} = X\hat{\beta} \tag{2}$$

There are several schemes which aim to find the optimal predictor $\hat{\beta}$; two such schemes will be introduced in this report, though they have been explained in greater detail in a previous report. See to the previous project collaboration of S. H. Magnússon and S. Håpnes [**?**] for more detail on the subject.

1. Ordinary Least Squares

The Ordinary Least Squares regression method (OLS) is the simplest and most intuitive to implement. This scheme involves minimizing a *Cost Function*, something which is generally the goal of any linear or logistic regression method. The cost function is in general a measure of how well a calculated prediction $\hat{y}$ performs in relation to the 'true' data y. In the case of OLS, the cost function is the *Mean-Squared Error* (MSE):

$$MSE(y, \hat{y}) = \frac{1}{2n} \sum_{i=0}^{n-1} (y - \hat{y})^2 , \tag{3}$$

where $\hat{y}$ is the prediction $\hat{y} = X\hat{\beta}$ presented previously. Deriving the cost function w.r.t the predictor $\hat{\beta}$ yields the minimum of the cost function, assuming there's no maximum. This results in the optimal prediction of $\hat{\beta}$ to be:

$$\frac{\partial}{\partial \hat{\beta}} MSE = 0 \Rightarrow \hat{\beta}^{OLS} = \left(X^T X\right)^{-1} X^T y \tag{4}$$

See to appendix A for the proof of this. It's worth noting that the appendix introduces the cost function as the *Residual Sum of Squares* (RSS). This function is identical to the MSE function except for a factor of $1/2n$, meaning that the minimum of both functions is still the same $\hat{\beta}$ expression.

This is the baseline simplest model for a prediction, and is the foundation for several other variants of regression. One of which being the method which will be compared to the neural network, namely the *Ridge Regression* scheme.

2. Ridge Regression

Originally, the ridge regression scheme was designed to solve cases of the matrix product $X^T X$ being *singular*:

$$\det\left(X^T X\right) = 0 \Rightarrow \left(X^T X\right)^{-1} \text{ does not exist.} \tag{5}$$

To solve this, the ridge scheme added a small *hyperparameter* λ to each of the diagonal elements of the matrix product, resulting in the sum being invertable once more:

$$\det\left(X^T X + \lambda I\right) \neq 0 \Rightarrow \left(X^T X + \lambda I\right)^{-1} \text{ exists,} \tag{6}$$

where I is the identity matrix and λ is some adjustable hyperparameter. Minimizing the MSE by setting the partial derivative w.r.t $\hat{\beta}$ equal to zero yields the following optimal Ridge scheme solution:

$$\frac{\partial C}{\partial \hat{\beta}} = 0 \Rightarrow \hat{\beta}^{Ridge} = \left(X^T X + \lambda I\right)^{-1} X^T y \tag{7}$$

This is the equation for the Ridge scheme, where λ is adjustable and typically quite small, X is the input/set of features, and y is the outcomes corresponding to those features, the "goal" of the prediction. Some research will be conducted to find the optimal hyperparameter λ of the Ridge scheme, though this has already been done in our previous project (linked to earlier).

B. Logistic Regression and Gradient Descent

Logistic regression is similar to linear regression, though it aims to produce predictions for classification purposes. The study of predicting the credit card data presented in this report is one of logistic regression, as it regards classifying people into two categories, namely (yes = 1) or (no = 0). Other classification examples range from hand-written image detection (case of 10 classes, see to Michael Nielsen's book [**?**]), to declaring whether a breast cancer tumor is malicious or not (see to work by Fombellida J. et al. [**?**]). Logistic regression works in approximately the same way as linear regression, though the output of the network should always be within the range of $\hat{y}_i \in [0, 1]$ for $i \in [0, 1, \ldots, n-1]$. Values between this are typically disregarded when testing the prediction, such that:

$$\hat{y}_i = \begin{cases} 1 & \text{if } \hat{y}_i \geq 1 \\ 0 & \text{if } \hat{y}_i < 1 \end{cases} . \tag{8}$$

If the number of *classes* is larger than two, then this is not quite the case. However, this report will not go into logistic regression cases which are not binary, so this definition of how we determine the outcomes will hold for the present study.

1. Stochastic Gradient Descent method

The primary comparison to the logistic neural network system is one of *stochastic gradient descent* (SGD). This

system is comprised of a prediction $\hat{y}$ which is fed through the so-called *sigmoid* function, defined as:

$$\sigma(\hat{y}) = \frac{1}{1+\exp(-\hat{y})}. \tag{9}$$

The sigmoid function is used to place the prediction into the domain of $\sigma(\hat{y}) \in [0,1]$, where the prediction accuracy is then assessed using a cost function and the training data y. The typical function to use in the logistic regression case is the *categorical cross entropy* function. The MSE function is most useful in the regression case. Both cost functions will be presented in further detail later on.

The stochastic gradient descent method is *iterative*, where the goal of the method is to update the predictor $\hat{\beta}$ in a way that decreases the cost function with each iteration. Ultimately this will produce a solution $\hat{\beta}$ which is in a local minimum of the cost function. Hopefully this minimum is also global, if the cost function is non-convex. Once the corrections/updates made to the predictor are sufficiently small, or the improvements are negligible, the method declares that it has found an approximate solution to the task of minimizing the cost function. As mentioned, this method can get caught in a local minimum of the cost function, resulting in a not-so-optimal predictor $\hat{\beta}$. Figure 1 illustrates a conceptualization of such a local minimum case.

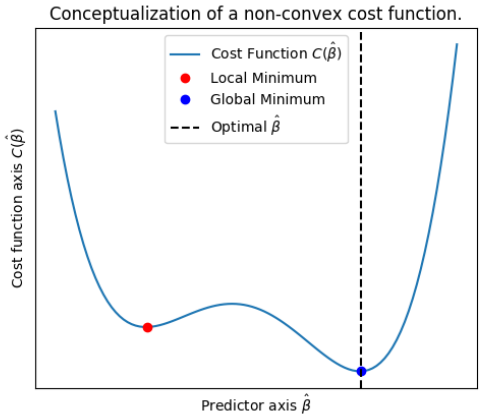

FIG. 1. Local minimum vs. Global minimum of non-convex cost function conceptualization figure.

There are a few practical solutions to solving the problem of local cost function minimum (such as using an adaptive learning rate to "anneal" closer to the global minimum [**?**]), but the solution introduced in this research is using *data batches*. This will be introduced briefly for the SGD solver, and described in further detail later on in the **Neural Network Dynamics** section.

The SGD method is initialized with a guess $\hat{\beta}_0 \neq 0$, and the output $\sigma(\hat{y}_0 = X\hat{\beta}_0)$ is calculated using the sigmoid function from equation 9. This will typically be a poor guess, resulting in the error between the training data y and first prediction $\hat{y}_0$ being relatively large. To improve this, the following gradient is applied to the predictor $\hat{\beta}$:

$$\hat{\beta}_1 = \hat{\beta}_0 - \eta \nabla_{\hat{\beta}_0} C, \tag{10}$$

where $\nabla_{\hat{\beta}_0} C$ is the gradient of the cost function with respect to the first guess $\hat{\beta}_0$, and η is the so-called *learning rate*. This is referred to as one iteration of the method, where $C(\hat{\beta}_1) < C(\hat{\beta}_0)$ is guaranteed given that the learning rate is well adjusted. This learning rate η is a subject which dictates how large the step size of the descent is, if the step is too large, then the cost function is not guaranteed to decrease after the iteration. There are many ways of modeling the learning rate, making this a central parameter of interest of the study presented. The learning rate will be discussed in greater detail later, though keep in mind that it is an adjustable parameter that is typically quite small, similar to the $l2$ regularization hyperparameter λ.

As previously mentioned, the MSE cost function is used for linear regression, and the *categorical cross-entropy* cost function is used for logistic regression cases. The categorical cross-entropy is found by taking the negative log of the so-called *Maximum Likelihood Estimation*:

$$P(\mathbb{D}|\hat{\beta}) = \prod_{i=0}^{n-1} \Big[p(y_i = 1|x_i,\hat{\beta})\Big]^{y_i} \Big[1 - p(y_i = 1|x_i,\hat{\beta})\Big]^{1-y_i}, \tag{11}$$

which is defined as the total likelihood for all possible outcomes of a data set $\mathbb{D} = \{(x_i, y_i)\}$ which has binary labels $y_i \in \{0,1\}$. Taking the negative logarithm of these probabilities produces the generalized categorical cross-entropy, or log-likelihood, cost function expression:

$$C(\hat{\beta}) = \sum_{i=0}^{n-1} \Big(y_i \log p(y_i = 1|x_i,\hat{\beta}) + (1-y_i)\log\Big[1 - p(y_i = 1|x_i,\hat{\beta})\Big]\Big), \tag{12}$$

or rewritten as [**?**]:

$$C(\hat{\beta}) = -\sum_{i=0}^{n-1} \big(y_i \hat{y}_i - \log\left(1 + \exp(\hat{y})\right)\big). \tag{13}$$

Applying the gradient of $\hat{\beta}$ to the cross-entropy function

$$\nabla_{\hat{\beta}} C = \begin{pmatrix} \frac{\partial}{\partial \hat{\beta}_0} \\ \frac{\partial}{\partial \hat{\beta}_1} \\ \vdots \\ \frac{\partial}{\partial \hat{\beta}_{p-1}} \end{pmatrix} C, \tag{14}$$

where $\hat{\beta}_i$ is the i-th component of the vector (not to be confused with the iterations of the SGD algorithm), yields:

$$\nabla_{\hat{\beta}} C = -X^T (y - p), \tag{15}$$

where $p = \sigma(\hat{y})$ is the prediction produced by $\hat{\beta}$. This is the gradient step introduced previously, yielding our expression for the stochastic gradient descent algorithm to be:

$$\hat{\beta}_{k+1} = \hat{\beta}_k - \eta \left(-X^T \left(y - \sigma(X\hat{\beta}_k)\right)\right). \tag{16}$$

This is iterated until the gradient is so small that $\hat{\beta}_{k+1} \approx \hat{\beta}_k$, which is declared to be a sufficient predictor. This is the iterative gradient descent method used to minimize the categorical cross-entropy cost function, specifically for the sigmoid function $\sigma(\hat{y})$. See to M.H. Jensen's FYS-STK4155 Lecture notes for more detail on deriving these expressions [**?**] (Logistic Regression section of the citation).

To avoid encountering any local minimum of the cost function, there is a random, or stochastic, factor included. This is known as the application of *mini-batches*, where the gradient is split up into multiple 'sub-gradients':

$$\nabla_{\hat{\beta}} C = \sum_{b=1}^{B} \nabla_{\hat{\beta}_b} c, \tag{17}$$

where B is the number of batches and b is a mini-batch.

C. Neural Network Dynamics

An ANN is generally a large system of many components which can be customized to conduct all kinds of studies. Before we can go into detial on the design of the network we will introduce the two most important algorithms to a neural network. These are the algorithm responsible for prediction and the algorithm responsible for learning, the so-called *Feed-Forward*, and *Backwards-Propagation* algorithms, respectively.

1. *Feed-Forward*

The general theory of the neural network is to predict using the following linear equation:

$$z = XW + b, \tag{18}$$

where X is the input data set, W are the associated weights, and b is the bias associated with the output. Figure 2 illustrates the output of a single neuron:

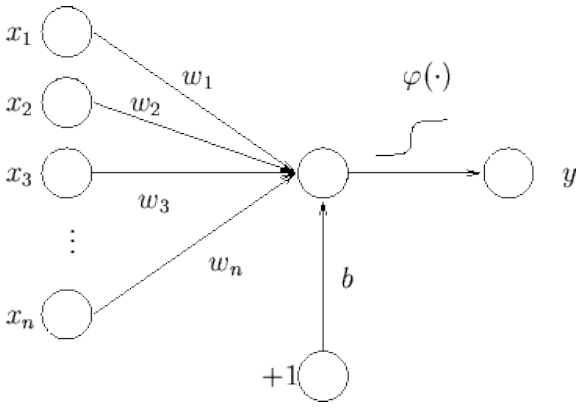

FIG. 2. An illustration of how the inputs X and weights W and bias b feed into a single neuron. Image taken from Antti Honkela.

An *activation function* is applied to z to break the linearity of the equation, such that the output is transformed to:

$$a = f\left(XW + b\right), \tag{19}$$

where f is the activation function. In figure 2, the symbol φ represents the activation function. In this report, the function will be signified by f.

It is worth noting that this is identical to the SGD system introduced in the previous section, where the predictor $\hat{\beta}$ is now replaced by the weights W, and the activation for the *SGD* method was the sigmoid function $f = \sigma$. The difference is that the neural network is designed to have multiple *layers* of such equations. Increasing the number of layers within the neural network is simple: The outputs of one layer become the inputs to the next layer, and so on. The extra/new layers are referred to as 'hidden'. Figure 3 illustrates what this looks like for three such layers:

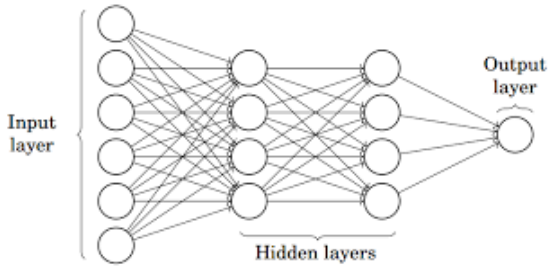

FIG. 3. A fully connected ANN illustration of two hidden layers and one output neuron. Image taken from: Vojtech Pavlovsky

Generalizing to a multilayer perceptron (MLP, simply means more than one layer) network yields the equations:

$$a^1 = f^1\left(XW^1 + b^1\right) \tag{20}$$
$$a^2 = f^2\left(a^1W^2 + b^2\right) \tag{21}$$
$$a^3 = f^3\left(a^2W^3 + b^3\right), \tag{22}$$

and so on, where f^1, f^2, and f^3 are activation functions (e.g. sigmoid), and W^i and b^i are the weights and biases corresponding to the output i. The general equations for the activation a for a given layer l are then given by:

$$z^l = a^{l-1}W^l + b^l \tag{23}$$

$$a^l = f^l(z^l). \tag{24}$$

Here, f^l is the activation function of layer l. It is good to keep these activation functions vague/general, as there are many such functions to choose between and the layers do not necessarily have the same ones.

The *feed forward* algorithm combines the weights and biases associated with the neural network to produce an output $\hat{y}$ in the following fashion:

1. Initialize by calculating z^1 and a^1 using equations 18 and 19
2. Save these and feed forward the activation from layer 1 to layer 2, calculating z^2 and a^2 using equations 23 and 24 respectively.
3. Continue this process until you get to the final output layer $l = L$, where $a^L = f^L(z^L) = \hat{y}$ is the prediction.

Since we are operating on a set of data which has a binary output 1 (default) or 0 (not default), then the final activation function f^L will always be the sigmoid function σ (for the same reasons as presented in the SGD section). Further explanation of activation functions and their variants are presented later on.

After the output $a^L = \hat{y}$ of the network is produced and all parameters z^l and a^l (for $l \in 1, 2, 3 \dots L$) are calculated, then the backwards propagation algorithm can be initiated.

2. Backwards-Propagation

The backwards propagation algorithm is an essential step to the network *learning* from its mistakes. The algorithm assesses the performance of the output $\hat{y}$, searches for the weights and biases which were most responsible for the inaccurate results, and tries to correct them to perform better. In essence this is a gradient descent algorithm, as the goal is once again to find a global cost function minimum:

$$\frac{\partial C}{\partial \theta} = 0, \tag{25}$$

where θ represents all the adjustable parameters of the regression. In the case of simple OLS, θ is the predictor $\hat{\beta}^{OLS}$, and in the case of the neural network, θ is the set of weights and biases $\theta \in (W, b)$. The backwards propagation algorithm aims to calculate the two parameters

$$\frac{\partial C}{\partial W} \text{ and } \frac{\partial C}{\partial b} \tag{26}$$

in order to adjust the current weights W and b to decrease the cost function:

$$W = W - \frac{\partial C}{\partial W} \text{ and } b = b - \frac{\partial C}{\partial b}. \tag{27}$$

This is done iteratively in the same fashion as the SGD algorithm.

To derive expressions for these partial derivatives, the following chain rule trick is used:

$$\frac{\partial C}{\partial W^l} = \frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial W^l}, \tag{28}$$

$$\frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial b^l}, \tag{29}$$

where W^l and b^l represent the weights and biases of layer l. The first component of layer $l = L$ needs to be calculated separately for the algorithm, before propagating backwards through layers $l \in [L-1, L-2, \dots, 1]$. These following four partial derivatives are therefore all that is needed to perform back propagation:

$$\frac{\partial C}{\partial z^L}, \frac{\partial C}{\partial z^l}, \frac{\partial z^l}{\partial W^l}, \text{ and } \frac{\partial z^l}{\partial b^l}. \tag{30}$$

In this paper, the partial derivative of the cost function with respect to the weighted output z^l will often times be referred to as:

$$\delta^l = \frac{\partial C}{\partial z^l}. \tag{31}$$

To derive expressions for the four essential partial derivatives presented in equation 30, the final layer $l = L$ is first examined. Another chain rule trick yields:

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}, \tag{32}$$

and utilizing the relation between a and z presented in equation 24 yields the following:

$$\frac{\partial C}{\partial z^L} = \nabla_a C \odot f'^L(z^L). \tag{33}$$

This is one of our equations, and is kept general for the sake of cost- and activation function flexibility. $\nabla_a C$ represents the gradient of C with respect to the output a^L, f'^L is the derivative of the activation function of the output layer, and the $\odot$ operator represents element-wise multiplication. This is also known as the Hadamard product. Moving on to the next equations, we use another chain rule trick:

$$\frac{\partial C}{\partial z^l} = \frac{\partial C}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial a^l}\frac{\partial a^l}{\partial z^l}. \tag{34}$$

$$\delta^l = \delta^{l+1}\frac{\partial z^{l+1}}{\partial a^l}\frac{\partial a^l}{\partial z^l}. \tag{35}$$

This product consists of elements which we can express differently, namely as:

$$\frac{\partial z^{l+1}}{\partial a^l} = \frac{\partial}{\partial a^l}\left(a^l W^{l+1} + b^{l+1}\right) = W^{l+1}, \text{ and} \tag{36}$$

$$\frac{\partial a^l}{\partial z^l} = f'^l(z^l). \tag{37}$$

This leaves us with the following expression (inserting equations 36 and 37 into equation 34):

$$\frac{\partial C}{\partial z^l} = \left((W^{l+1})^T \frac{\partial C}{\partial z^{l+1}}\right) \odot f'^l(z^l) \tag{38}$$

$$\delta^l = \left((W^{l+1})^T \delta^{l+1}\right) \odot f'^l(z^l) \tag{39}$$

This is another one of the equations required for the backwards propagation algorithm. Now, examination of the final terms yields:

$$\frac{\partial z^l}{\partial W^l} = \frac{\partial}{\partial W^l}\left(a^{l-1}W^l + b^l\right) = a^{l-1} \tag{40}$$

$$\frac{\partial z^l}{\partial b^l} = \frac{\partial}{\partial b^l}\left(a^{l-1}W^l + b^l\right) = 1. \tag{41}$$

Inserting all these relations back into equations 28 and 29 yields:

$$\frac{\partial C}{\partial b^l} = \delta^l \text{ , and} \tag{42}$$

$$\frac{\partial C}{\partial W^l} = (a^{l-1})^T \delta^l. \tag{43}$$

The four equations for the backwards propagation algorithm can then be summarized as (rewriting the vectors with index notation):

$$\delta^L = \nabla_a C \odot f'^L(z^L), \tag{44}$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot f'^l(z^l), \tag{45}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l, \tag{46}$$

$$\frac{\partial C}{\partial W_{j,k}^l} = a_k^{l-1}\delta_j^l, \tag{47}$$

where $\nabla_a C$ is the gradient of the cost function w.r.t the output a, and f'^l is the derivative of the activation function of layer l w.r.t z^l.

This is all that is needed to ensure that the weights and biases can be corrected in a manner that reduces the cost function C in relation to the adjustable parameters $\theta \in (W, b)$. Once again, there is the possibility of being caught in a local minimum in the case of a non-convex cost function C. The learning rate η and batch concepts are therefore presented in the ANN context, though they are very similar to what was presented for the SGD section.

3. Learning rate, batches and l2 regularization

To research the impact of these gradient factors to the biases and weights, a learning rate is included to control the size of the steps. This is identical to the SGD learning rate η, which is simply a value which the gradients are multiplied with. The backwards propagation algorithm now updates the biases and weights by the factors:

$$b_j^l = b_j^l - \eta \frac{\partial C}{\partial b_j^l}, \tag{48}$$

$$W_{j,k}^l = W_{j,k}^l - \eta \frac{\partial C}{\partial W_{j,k}^l}. \tag{49}$$

There are several ways of modeling the learning rate, some of which involves the *adaptive* learning rate or the *time-based* learning rate. However, these are not implemented into the current research. The learning rate will be set to a (typically small) constant value throughout the project. The best learning rate value is a topic which will be of interest when a search of the ANN which produces the most accurate predictions is researched.

The neural network gradient is segmented into batches in a similar way to what was done in the SGD section. This is done to avoid the problem of local minimum of the cost function; a problem which has been introduced already. This is once again done by averaging the gradients of multiple inputs at a time, making it so that the input matrix X is now of size $(batchsize \times p)$, and the weights and biases are now updated using the sum of these average gradients. The data set must be shuffled after each batch is iterated over using the feed-forward and backwards propagation algorithms presented. This is done to further increase the likelihood of the algorithm to reach a global minimum, as the averages will differ slightly depending on the batchsize. If the batchsize is large, then the average of the batch gradient will typically not change very drastically; the batchsize is therefore recommended to be around $B = 100$. This is already quite a large number of cases, so should not be increased much more. After each of the batches have updated the weights and bias gradients and the data set is shuffled, a new *epoch* begins. This signifies all the data has been cycled through and the gradients have been updated accordingly. This equates to one 'iteration' $k \to k + 1$ of the SGD method presented previously. The number of epochs is not quite as important as the change in the gradients *per* epoch. As, if an epoch does not cause the weights and biases to change very much, then the current parameters are declared as sufficient to produce a cost function minimum.

Once more subject is introduced before moving onto an introduction of cost- and activation functions, namely l2 regularization. The learning rate is one of the parameters which is adjustable and of interest, the other one being the l2 regularization hyperparameter λ. The l2 regularization involves adding an additional factor/penalty to the original cost function C_0, such that:

$$C = C_0 + \frac{\lambda}{2n}\sum_j |W_j|^2 \tag{50}$$

is the new cost function, where p is the number of features. This ultimately penalizes the weights if they are too large, making it so that the weights W are encouraged to be smaller. The net impact of this factor is that the derivative

of the cost function w.r.t the weights presented earlier now has an additional factor which subtracts a fraction of the current weights:

$$W_{j,k}^l = W_{j,k}^l - \eta \left[\frac{\partial C_0}{\partial W_{j,k}^l} + \frac{\lambda}{n} W_{j,k}^l \right] \tag{51}$$

$$W_{j,k}^l = \left(1 - \frac{\eta\lambda}{n} \right) W_{j,k}^l - \eta \frac{\partial C_0}{\partial W_{j,k}^l}, \tag{52}$$

where η is the learning rate, λ is the $l2$ regularization hyperparameter, C_0 is the cost function without $l2$ regularization, and p is the number of features/inputs into the network. This hyperparameter λ is something which will be studied in detail in an attempt to find the value which produces the most accurate $\hat{y}$ prediction (best weights and biases).

D. Cost- and Activation functions

A quintessential component to the neural network system is the choice of cost- and activation functions. The feed-forward system is dictated and fed through multiple activation functions, vastly influencing the outcome. This will be a large point of debate in the methods, results and the discussion section. A few cost- and activation functions will be presented here for this study, though far more variants exist in their respectful families. Table I lists a few cost functions which are explored, and their respectful derivatives:

	$C(y, a^L)$	$\nabla_a C(y, a^L)$
Mean-Squared Error	$\mathbb{E}\left[\frac{1}{2}(y - a^L)^2\right]$	$a^L - y$
Cross-Entropy	$\begin{cases} -\log(a^L) & \text{if } y = 1 \\ -\log(1 - a^L) & \text{if } y = 0 \end{cases}$	$a^L - y$

TABLE I. Several cost function examples. Mainly, the cross-entropy and mean-squared error (MSE) functions will be explored in this study. As explained previously, the activations of the final layer a^L are equivalent with the prediction of y, namely $\hat{y}$.

The symbol $\mathbb{E}$ indicates the *expectation value* of some list of values, formally defined as:

$$\mathbb{E}[a] = \sum_{i=0}^{n-1} a_i \cdot P(a_i), \tag{53}$$

where $P(a_i)$ is the probability of the outcome a_i. When the results are uniformly distributed (have equal outcomes), then the expectation value is given by

$$\mathbb{E}[a] = \frac{1}{n} \sum_{i=0}^{n-1} a_i. \tag{54}$$

1. Mean-Squared Error

Mean-squared error, or the *L2 loss function* is the most typical cost function to use in non-logistic regression cases. This is a very consistent and simple function to measure the performance of a prediction $\hat{y}$ in relation to the training data y.

2. Cross-Entropy

The most common cost function used in logistic regression is the cross entropy, or *log-loss* function. The function listed in the table measures the performance of a classification model, where all outputs are set to be between 0 and 1. This function stems from the *Maximum Likelihood Estimation* principle presented previously. This cross entropy expression is specific to the case of binary classification (two outcomes). This is the case for the credit card data logistic regression (default or not default), so the cross-entropy is the cost function of choice for neural network logistic regression study.

These are the two choice cost functions to perform the classification and regression research. In addition to choosing a cost function for predicting, one must also make a choice of activation functions. Table II lists a few activation functions which are explored, and their respectful derivatives:

	$f(z_i)$	$f'(z_i)$
Sigmoid	$(1 + e^{-z_i})^{-1}$	$f(z_i)\,(1 - f(z_i))$
tanh	$\tanh(z_i)$	$1 - f^2(z_i)$
ReLU	$\begin{cases} z_i & \text{for } z_i > 0 \\ 0 & \text{else} \end{cases}$	$\begin{cases} 1 & \text{for } z_i > 0 \\ 0 & \text{else} \end{cases}$
ReLU6	$\begin{cases} z_i & \text{for } z_i \in [0, 6] \\ 0 & \text{for } z_i < 0 \\ 6 & \text{for } z_i > 6 \end{cases}$	$\begin{cases} 1 & \text{for } z_i \in [0, 6] \\ 0 & \text{else} \end{cases}$
Softmax	$e^{z_i} \cdot (\sum_k e^{z_k})^{-1}$	$\begin{cases} s_i(1 - s_j) & \text{for } i = j \\ -s_i \cdot s_j & \text{for } i \neq j \end{cases}$
Softsign	$z_i \cdot (1 + \lvert z_i \rvert)^{-1}$	$(1 + \lvert z_i \rvert)^{-2}$

TABLE II. Table listing a number of common activation functions and their derivatives. Several of these are of interest and are implemented into the study.

These are some key activation functions which may be studied. One thing to note is that the derivative of the softmax function is a matrix since the activation of node a_j can depend on all the other activations from the denominator $\sum_k e^{z_k}$ of the function. Technically, the derivative stated

in the table is really the derivative $\frac{\partial}{\partial a_j} f(z_i)$. Following is a short discussion of the strengths and some weaknesses of some of the activation functions introduced in table II:

3. Sigmoid function

The Sigmoid function is typical for predictive applications as it is bounded to $\sigma \in [0, 1]$. This is useful for logistic regression purposes as discussed previously, though it does not come without its flaws. The sigmoid function has the problem of *vanishing gradients* for values outside of the range (approximately) $x \in [-6, 6]$. The derivative of the sigmoid is consistently vanishing in these areas $\sigma'(z) \approx 0$ for $z \in [\leftarrow, -6] \vee [6, \rightarrow]$, resulting in the weights and biases not being updated properly in the backwards propagation algorithm. This problem can however be circumvented through proper data preprocessing and weights initialization. If the node values z values are within this range, the network should be able to tell the difference between them in the gradient. This requires the inputs X to be scaled correctly as well as the weights and biases to not be too large.

4. Tanh function

Although the hyperbolic tangent function has different bounds than the sigmoid function (ranging from -1 to 1), it can just as easily be applied to binary classification cases. The only difference needed is to edit the input (training and testing) data to have the same ranges. In the case of the binary credit card classification, all the zeros (not default) would need to be changed to -1's. This is not done in this study, though it is a possibility. This function also has vanishing gradients for large values, though this can be avoided by proper data processing once again.

5. ReLU6

ReLU is shown to be specifically good for image recognition applications [**?**]. Keeping to the same family, though introducing an upper bound of 6 is useful in this case. ReLU6 is therefore an excellent variant of the ReLU function, keeping some of it's characteristics and making sure that it is bounded from both below and above.

6. Weight- and Bias Initialization

Xavier and He [**?**] conducted researched regarding what the best bias and weight initialization expressions were, and found that the following are the optimal mean- and standard deviation values for the weights (specifically for the ReLU activation function family):

$$\mu = 0 \text{ and } \sigma = \sqrt{\frac{2}{n_i + n_{i+1}}}, \tag{55}$$

where n_i is the number of nodes in the current layer, and n_{i+1} is the number of nodes in the next layer. The weights are initialized randomly with these parameters, and the biases are initialized as zero in this scheme (henceforth referred to as the *Xavier initialization*).

Other initialization schemes can come up, though it is very important that these are not large in absolute value. This may cause vanishing gradients in some of the activation functions, as mentioned above.

E. Classification

1. Neural Network Application

The neural network applied to the classification data is what is known as a *multilayer perceptron*. This means that the network consists of non-linear activation functions and a scheme of fully-connected weights and biases. This means that each of the neurons from layer i are connected by a weight and a bias to the neurons in layer $i + 1$ and so on.

2. Accuracy Assertion of Classification

There are several ways to assert the accuracy of the logistic model. One which is the simplest and most intuitive is the *accuracy score*, which measures the fraction of correctly guessed targets:

$$Acc = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}(\hat{y}_i = y_i), \tag{56}$$

where

$$\mathbb{1}(\hat{y}_i = y_i) = \begin{cases} 1 & \text{if } \hat{y}_i = y_i \\ 0 & \text{else} \end{cases}. \tag{57}$$

This is the simplest accuracy metric possible, though there are some weaknesses to designing a model around maximizing this. A few more accuracy metrics are therefore introduced in this section for a more in-depth analysis of the performance of the classification methods. One of these is the so-called *confusion matrix*:

		True	
		0	1
Pred	0	$a_{0,0}$	$a_{1,0}$
	1	$a_{0,1}$	$a_{1,1}$

This matrix conveys the model performance in greater detail than the accuracy score, as the errors $a_{0,1}$ and $a_{1,0}$ are *not equivalent*. They are called *false positives* and *false negatives*, respectively. If the data being predicted is something like patient data rather than credit card data and the logistic regression involves predicting whether or not a patient has a malicious disease, then it is easier to imagine how these two differ. It is far worse to mis-classify

a sick patient than a healthy one. In our case, if 1 = sick, then we would like for $a_{0,1}$ to be as small as possible, but it is far more important that $a_{1,0} \to 0$. Ultimately though, in any case, the perfect model diagonalizes this matrix, such that $a_{1,0} = a_{0,1} = 0$.

One more accuracy metric is the area under a *cumulative gain chart*. The cumulative gain chart is designed to evaluate binary classifiers by calculating the following two parameters:

$$\text{TPR} = \frac{a_{1,1}}{a_{1,1} + a_{1,0}} \tag{58}$$

$$\text{SUP} = \frac{a_{1,1}}{a_{1,1} + a_{0,1}} \tag{59}$$

TPR is the *True Positive Rate* (often times referred to as the sensitivity), and SUP is the *Support*, or *Predictive Positive Rate* of the prediction. To construct the cumulative gain chart, the TPR and SUP must be calculated for every data point. Once this is accomplished, the values are ordered such that the predicted outputs furthest from 0.5 are prioritized. Next, the values are cumulatively summed up and the resulting series are normalized to 1. This normalization is important, as we are interested in the *rates* of the prediction. Following is a figure of a cumulative gain chart of a model with an accuracy of 68%, produced to conceptualize the accuracy metric:

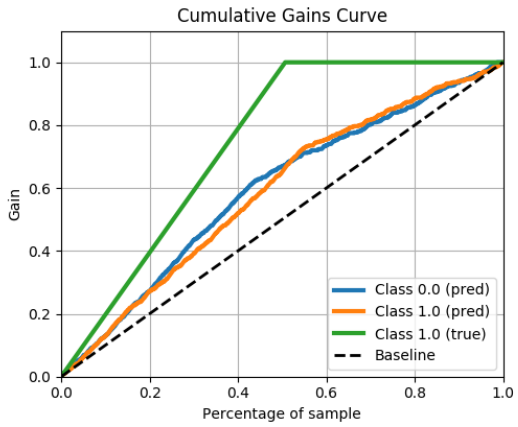

FIG. 4. Conceptualized figure of a cumulative gain chart. The figure includes all cases of 0 and 1 predictions, and a perfect 1 model. The model also includes a b aseline illustrating what a random 50/50 pick of 0's and 1's looks like.

Hopefully, a prediction model will be somewhere in between the perfect model and the 50% random pick baseline. The better the model, the larger the area between the cumulative model curve and the baseline will be. This is therefore the measure of the model accuracy.

One more metric is finally introduced for binary classification assessment: The $F1$ Score. This is formally defined as

$$F_1 = 2\left(\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}\right), \tag{60}$$

where the Precision, or the Positive Predictive Value (PPV), is related to the confusion matrix elements presented above in the following fashion:

$$\text{Precision} = \frac{a_{1,1}}{a_{1,1} + a_{0,1}}, \tag{61}$$

and the Recall, or the True Positive Rate, is related by:

$$\text{Recall} = \frac{a_{1,1}}{a_{1,1} + a_{1,0}}. \tag{62}$$

The Precision assesses the *true positives* $a_{1,1}$ in relation to the *false positives* $a_{0,1}$, while the Recall assesses the true positives in relation to the *false negatives* $a_{1,0}$[**?**].

F. Regression

The goal with the regression study portion of the project is to use neural networks and linear regression methods to fit *Franke's Function*. This function is typical to use when testing regression methods, and is given by:

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+ \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &- \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right), \end{aligned} \tag{63}$$

This function exhibits a very interesting surface plot which is very recognizable in prediction data. The function surface is illustrated in figure 6

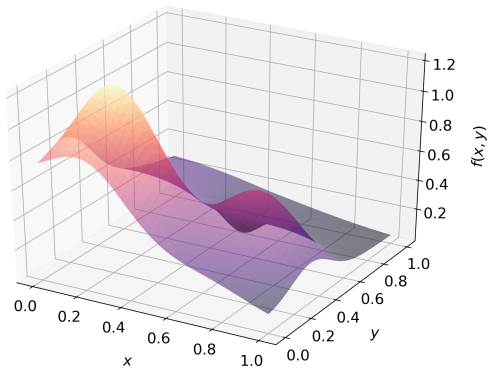

FIG. 5. The *Franke function* surface plot illustration for x and y values ranging from zero to one.

Attempting to recreate this function using ridge regression and a neural network would be improper without some functional noise. Adding some noise (as a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$) produces the following data set:

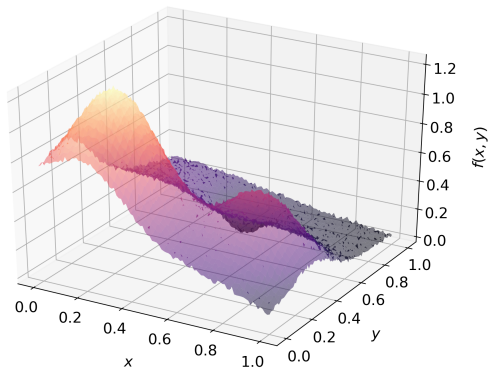

FIG. 6. The *Franke function* surface plot illustration for x and y values ranging from zero to one. Random Gaussian noise with mean value $\mu = 0$ and standard deviation $\sigma = 0.01$ is added to each data point $f(x_i, y_i)$.

This is more similar to the data the methods will attempt to recreate. Two methods are implemented to attempt to reproduce Franke's function: *Ridge Regression* (presented previously) and a *Neural Network* (this time applied to data which is not classification-oriented).

1. Neural Network Application

The neural network applied to the regression data is very similar to the classification network described previously (multilayer perceptron). Now however, instead of a constant binary output $a^L \in [0, 1]$, the network is trained to output the results of Franke's function. This means that the output activation function cannot be a function which bounded asymptotically; some common activation functions such as the *Leaky ReLU* come to mind, though the most intuitive one is simply using the *identity* $f(x) = x$. The research conducted on Franke's function will include the learning rate and hyperparameter as previously, so a search for an optimal value for both of these will once again be performed.

2. Accuracy Assertion of Regression

The accuracy metrics used for the prediction of Franke's function are the MSE (introduced earlier) and the so-called R2-Score. These two accuracy metrics are given by the following expressions:

$$MSE(y, \hat{y}) = \frac{1}{2n} \sum_{n-1}^{i=0} (y - \hat{y}) \tag{64}$$

$$R2(y, \hat{y}) = 1 - \frac{RSS}{TSS}, \tag{65}$$

where RSS is the *residual sum of squares*, and TSS is the *total sum of squares*. These are respectively defined as:

$$RSS(y, \hat{y}) = \sum_{i=0}^{n-1} (y - \hat{y})^2 \tag{66}$$

$$RSS(y, \hat{y}) = \sum_{i=0}^{n-1} (y - \bar{y})^2 \tag{67}$$

$$\bar{y} = \frac{1}{n} \sum_{i=0}^{n-1} y_i. \tag{68}$$

These expressions are all that is needed to calculate the R2 score and MSE. As discussed earlier, we would like the MSE (cost function in the case of regression) to be as small as possible. The R2 score is different, it is a score which is always in the domain of $R2 \in [0, 1]$, where $R2 \to 1$ for a 'perfect model', meaning that the R2 score is a parameter we would like to maximize.

III. METHOD

The algorithms presented are implemented and simulated in Python3.6 using several libraries. Some of these have functions which are implemented explicitly; the PCA is for example implemented using a package from the scikit-learn library. This will be gone into further detail later on. Before building the neural network, the data must be prepared. This is called preprocessing.

A. Preprocessing of the Data Set

The most important section of any data analysis is getting to know and performing a preprocessing of the data set. This involves formating the data in a manner which is easily interpreted and understood by the neural network nodes. Firstly, an outlier filtration is performed on the data.

1. Outlier Filtration

Outlier filtration involves removing data which does not belong in the dataset. Data which is out of the defined boundaries can often be found in large datasets which are input by people, as is likely to be case for the credit card data presented. There are a few defined boundaries of the data which are easy to filter out. This is for example the gender definition made by the bank, where 1=male and 2=female. This automatically excludes all values outside of this definition.

The definitions presented are:

- X1: Amount of given credit. Must be positive.
- X2: Gender (1=male, 2=female)
- X3: Education (1=graduate school, 2=university, 3=highschool, 4=other).

- X4: Marital Status (1=married, 2=single, 3=others)
- X5: Age. Must be positive.
- X6 - X11: History of past payment. Can be any integer from -2 to 9.
- X12-X17: Amount of bill statement. Can be negative.
- X18-X23: Amount of previous payment. Must be positive.

Before any scaling is applied, these categorical outliers are removed.

2. Column Scaling

Some of the columns have values which require scaling before they are input into the network. If the data were not scaled in the preprocessing, then the sigmoid activation function (as mentioned in the disappearing gradient discussion) of the columns with large values would produce redundant outputs (impossible to tell from each other). The network would not be able to distinguish between these values as well as it would if the ages were centered around 0 with a standard deviation of one (the typical way to scale). In this case, the 'resolution' of the sigmoid (or similar functions such as the hyperbolic tangent) would be far better, resulting in a network which can tell the difference between the behaviour of a 20 and a 40 year old.

3. One-Hot Encoding classifiers

Columns X2, X3, and X4 classify the person in a different way than, say, column X1. The difference between the two is that one is a spectrum while the other is binary. For example, the numbers in column X1 express the *degree* of something (e.g. 2 is twice that of 1). However, column X2 places the person in either category 1 (male) or 2 (female). In this case the numbers are not related in the same way as previously. These are two categories and must therefore be expressed in out neural network using *One-Hot Encoding*. This would be done in the following manner for five people:

$$\begin{bmatrix} \text{Male} \\ \text{Female} \\ \text{Female} \\ \text{Male} \\ \text{Female} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{69}$$

This is what is known as one-hot encoding, and is widely used to encode categorical variables. Columns X3 and X4 must also be converted to this format, as they also have categorical information:

$$\begin{bmatrix} \text{2 (University)} \\ \text{4 (Other)} \\ \text{1 (Grad School)} \\ \text{3 (Highschool)} \\ \text{4 (Other)} \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{70}$$

Hopefully, it is clear now why the encoding method is called *one-hot*, as it creates an array reserving one element for each categorical possibility, the array element which is non-zero is then considered the "hot" one.

4. History of past payment columns

The history of past payment columns (columns X6-X11) is an interesting case, as the dataset lists it as a combination of categorical and continuous variables. The research paper by I-Cheng Yeh and Che-hui Lien [**?**] lists these variables as quote:

> X6–X11: History of past payment. We tracked the pastmonthly payment records (from April to September,2005) as follows: X6 = the repayment status in Septem-ber, 2005; X7 = the repayment status in August,2005;...;X11 = the repayment status in April, 2005.The measurement scale for the repayment status is:-1 = pay duly; 1 = payment delay for one month;2 = payment delay for two months;...; 8 = paymentdelay for eight months; 9 = payment delay for ninemonths and above.

This is descriptive enough, though upon examination of the data listed in these columns, there are several cases of -2 and 0 (though only -1 and 1-9 are defined). This indicates to outliers, though a very significant amount of the dataset has values -2 and 0 in columns X6-X11. Upon further research into what the values represent, an article (link to it **here**) theorizes that[**?**]:

- -2 is 'Balance paid in full and no transactions in this period' (inactive card).
- -1 is 'Balance paid in full, but account has a positive balance at the end of period' (as previously).
- 0 is 'customer paid minimum due amount but not entire balance'.

This is based on how banks operate and is what this research will use as a basis.

B. Neural Network Design

An object oriented code was built in Python 3.6.8 with the aim to make a generalized reusable code which has little to no specifications to the problem at hand. A large number of libraries are used for various different functionalities, the most important (excluding trivial unnecessary libraries) of them being:

- NumPy version 1.17.2
  - NumPy is an excellent scientific programming library for handling of large array structures (e.g.

matrix-matrix multiplication) and has a large number of useful built in functions (e.g. exponential and logarithm functions)

- Pandas version 0.25.1
  - Pandas is primarily used for large data set manipulation and data processing. Several pandas functionalities are used when extracting and preprocessing the credit card data set.
- CuPy version 6.5.0
  - CuPy is a package which allows python to make full use of GPU architecture, accelerating matrix library operations using NVIDIA CUDA.
- Matplotlib version 3.1.1
  - Matplotlib's PyPlot package is a Python library which is incredibly useful for data visualization and plotting. Several of the plots presented in this paper are generated by Matplotlib.

1. Overview of Neural Network Class

The neural network is implemented into an object oriented code in python, where several of the network attributes can be stored. There are also a large amount of functions implemented into the class which have various objectives, ranging from network training to network performance analysis.

The class is designed to accept a set of input training matrices $\mathbf{X}$ and $\mathbf{Y}$; a preprocessing method is called that automatically implements *one-hot encoding* on the categorical columns of the input datasets, then automatically scales X based on the activation function selected for the hidden layers. Y is then scaled depending on the range of values of the output layer's activation function (for the *sigmoid* function, this would be in the range 0 and 1, while tanh requires values in the range -1 and 1. These scaling factors are then stored as instance attributes, and can be saved and reloaded when performing a prediction.

Once all the preprocessing has been completed, there is an upsampling method that can be used in cases where the ratio of 0 to 1 outputs is not close to 1; this randomly selects inputs linked to the underrepresented outputs, and appends them to the end of the input array.

Next, a hidden layer configuration is selected, as well as an activation function for the hidden layers; a separate activation function can be selected for the output layer. At this point, each datapoint is then randomly shuffled and subsequently divided into a set of batches, the number of which depends on the selected *batch-size*. The shuffling is important, since the upsampled points were previously appended to the dataset rather than inserted into random locations. At this point, it is simply a matter of implementing the *feed-forward* and *backpropagation* algorithms for each batch, then each epoch. For the purpose of efficacy, there is a GPU processing option that will implement CUDA vectorization.

Finally, the trained sets of weights and biases are stored in `.npy` files, where they can be loaded and used in a later prediction. This is necessary, since training requires a lot of time and processing power.

2. Logistic Regression/Stochastic Gradient Descent

The neural network class can be trained with an empty hidden layer configuration; if the sigmoid activation function is selected as the output activation function, this will perform a feed-forward and backpropagation over the input and output layers, and perform logistic regression without the need for a separate class.

C. Hyperparameter and Learning Rate analysis

The simplest way of performing a search for the optimal parameters is initializing a grid search over multiple hyperparameter values $\lambda_i \in [\lambda_1, \lambda_2, \ldots, \lambda_{r1}]$ and learning rates $\eta_i \in [\eta_1, \eta_2, \ldots, \eta_{r2}]$, producing an array of $(r1 \times r2)$ trained neural networks. These networks must all have the same design to perform the analysis of λ and η properly, so they are initialized with two hidden layers, each with one-hundred nodes.

D. Classification

The goal of the classification research presented is to use the accuracy metric to compare the two methods of stochastic gradient descent and artificial neural network. A supplementary research study is conducted on the neural network, where an extensive search of which network structure produces the best results. The additional accuracy metrics of the area under the cumulative gain curve and $F1$ score are included for this study.

Using the neural network class, an analysis is performed on the credit card dataset – using *cross-entropy* as a cost function, a grid search is performed for ten logarithmically-spaced values $\lambda \in [10^{-8}, 10^{-6}]$ and ten linearly-spaced values $\eta \in [0.06, 0.2]$. This is performed for an empty layer configuration (logistic regression) and for two hidden layers with one-hundred nodes each, using the tanh activation function in the hidden layers (for the second case), and the sigmoid function on the output layer.

E. Regression

The goal of the regression research presented is use the mean-squared error MSE cost function to compare the Ridge regression and neural network schemes.

The linear regression study conducted on Franke's function in this report is nearly identical to the study from our previous project. The code from that project is therefore

utilized to produce results which are compared to the neural network. To see the details of the Ridge scheme code and the results of the project, visit the project 1 github page presented previously.

Since we are no longer working with binary outputs, the parameters that are being tested are no longer the same (accuracy, area under curve, and $F1$ scores do not apply to regression cases). The way that the regression results are assessed is by the MSE cost function presented previously.

With the neural network class, a grid search is performed such that there are two hidden layers with one hundred nodes each, using the tanh activation function for the hidden layers and sigmoid for the output layer. This time, ten logarithmically-spaced values $\lambda \in [10^{-6}, 10^{-5}]$ are selected, as well as ten linearly-spaced values $\eta \in [0.08, 0.2]$.

IV. RESULTS

A. Classification

Following are some results from the neural network classification project. Figure 7 illustrates the accuracy scores for network $N3$:

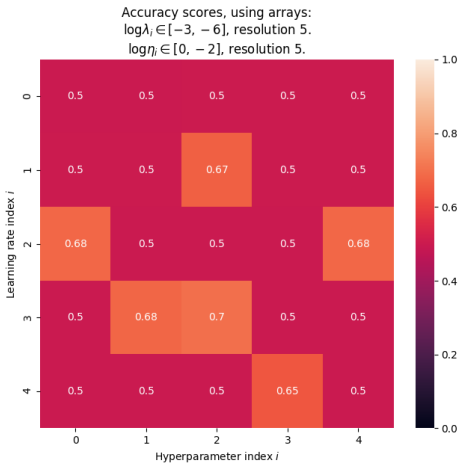

FIG. 7. Grid search with label $N3$ results. This figure illustrates the accuracy scores for several neural networks with various hyperparameters λ and learning rates η. The range and resolutions of the parameters are in the title, while the indices of the axes range from largest to smallest.

Figure 8 illustrates the area under the cumulative gain curve for network $N3$:

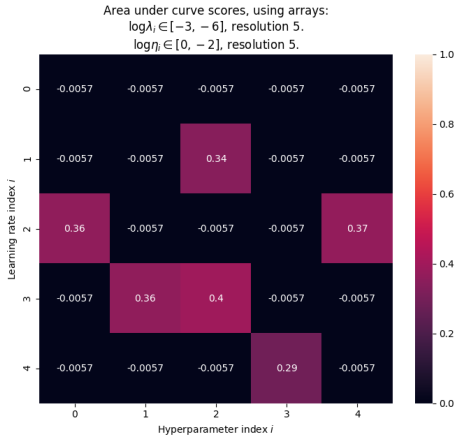

FIG. 8. Grid search with label $N3$ results. This figure illustrates the area under the cumulative gain curve for several neural networks with various hyperparameters λ and learning rates η. The range and resolutions of the parameters are in the title, while the indices of the axes range from largest to smallest.

Figure 9 illustrates the $F1$ scores for network $N3$:

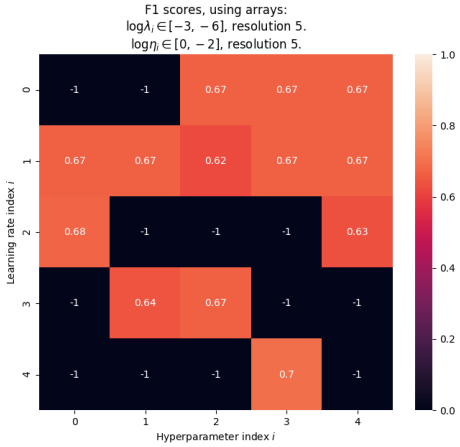

FIG. 9. Grid search with label $N3$ results. This figure illustrates the $F1$ for several neural networks with various hyperparameters λ and learning rates η. The range and resolutions of the parameters are in the title, while the indices of the axes range from largest to smallest.

B. Regression

Following is a table from project 1, listing the precision of three regression schemes applied to Franke's function:

TABLE III. Table listing the final results and comparisons of the regressional methods applied to Franke's function with $n = 10,000$ data points.

Scheme	MSE minimum	p_{deg}	$\log(\lambda)$
OLS	0.2493	8	–
Ridge	0.2489	11	-9
Lasso	0.2524	8	-12

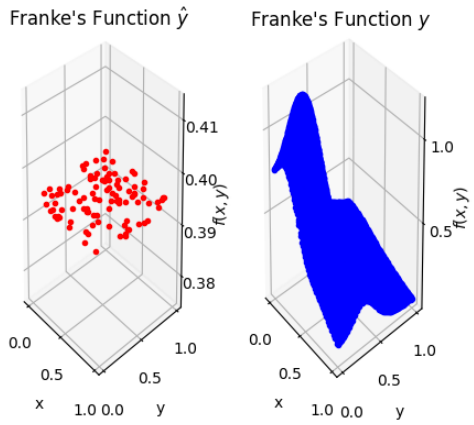

FIG. 10. The ANN regression on Franke's function. With $MSE = 4.08$.

V. DISCUSSION

A. Classification

B. Regression

Ridge regression accomplished $MSE_{min} = 0.2489$ at polynomial degree $p_{deg} = 11$ and hyperparameter $\log \lambda = -9$.

C. Comparison of the Methods

VI. CONCLUSION

REFERENCES

APPENDIX A: OLS PREDICTOR DERIVATION

Setting the Residual Sum of Squares (RSS) partial derivative with respect to the predictor β equal zero yields:

$$\frac{\partial}{\partial \beta}\ RSS = 0 \tag{71}$$

Inserting the expression for RSS (given by $2n \cdot MSE$, where MSE is defined in equation 3) yields:

$$\frac{\partial}{\partial \beta}\ (y - X\beta)^2 = 0, \tag{72}$$

rewriting as:

$$\frac{\partial}{\partial \beta}\left((y - X\beta)^T (y - X\beta)\right) = 0, \tag{73}$$

expanding:

$$\frac{\partial}{\partial \beta}\left(y^T y - y^T X\beta - (X\beta)^T y + (X\beta)^T (X\beta)\right) = 0, \tag{74}$$

removing the factors which are independent of β and utilizing the rule

$$\left(A^T B\right)^T = B^T A \tag{75}$$

yields:

$$\frac{\partial}{\partial \beta}\left(-y^T X\beta - \beta^T X^T y + \beta^T X^T X\beta\right) = 0, \tag{76}$$

Also, seeing as the product $y^T X\beta$ is a scalar, we have that:

$$y^T X\beta = \left(y^T X\beta\right)^T = \beta^T X^T y \tag{77}$$

Since the transposed of a scalar equals the scalar. This allows us to rewrite equation 76 to

$$\frac{\partial}{\partial \beta}\left(-2\beta^T X^T y + \beta^T X^T X\beta\right) = 0, \tag{78}$$

Differentiating with respect to β yields:

$$-2X^T y + \frac{\partial}{\partial \beta}\left(\beta^T X^T X\beta\right) = 0 \tag{79}$$

One more matrix-vector differentiation formula is needed for the final product:

$$\frac{\partial(a^T A a)}{\partial a} = 2Aa = 2a^T A \tag{80}$$

for vectors a and a symmetric matrix A. We can now derive the final formula for β:

$$-2X^T y + \frac{\partial}{\partial \beta}\left(\beta^T X^T X\beta\right) = 0 \tag{81}$$

$$-2X^T y + \frac{\partial}{\partial \beta}\left(\beta^T A\beta\right) = 0 \tag{82}$$

$$-2X^T y + 2A\beta = 0 \tag{83}$$

$$-2X^T y + 2X^T X\beta = 0 \tag{84}$$

$$-X^T y + X^T X\beta = 0 \tag{85}$$

by the declaration of $X^T X = A$ being symmetric. This finally yields:

$$X^T X\beta = X^T y \tag{86}$$

$$\beta = \left(X^T X\right)^{-1} X^T y. \tag{87}$$