REGRESSION AND CLASSIFICATION: FROM LINEAR METHODS TO NEURAL NETWORKS

SHAFAQ NAZ SHEIKH
HTTPS://GITHUB.UIO.NO/SHAFAQNS/FYS-STK4155/TREE/MASTER/PROJECT2

Abstract. In this article we study machine learning algorithms for regression and classification problems. For Regression we study ordinary least squares(OLS) and ridge regression. Instead of using the normal equations, we develop our own code for stochastic gradient descent(SGD), to train our regression models. The data we will be using for regression analysis is generated using the Franke function. For classification we will study feed-forward neural networks(FFNN) and logistic regression. In addition we will develop our own code for implementing the FFNN, as well as logisitic regression. The data we will be using for analysing the classification methods, is the so-called MNIST data set. This is a set of images representing hand written numbers from zero to nine. We are also going to look at the effects of the various hyper parameters involved in these methods. We find that the gradient descent method is an optimal choice, provided we are able to find the right parameters. On the other hand, we find that the FFNN is a better method, as compared to logistic regression for classification problems.

Date: 7th November 2020.

1. Introduction

Machine learning represents the science of giving computers the ability to learn without being explicitly programmed. The goal is to be able to make predictions from data, as well as extract important information and the underlying patterns in large data sets. In pretty much every machine learning problem, we start out with some kind of a data set $\mathbf{X}$, a model $g(\beta)$, which is a function of the parameters β and a cost function $C(\mathbf{X}, g(\beta))$, that allows to judge how well the model $g(\beta)$ explains our observations $\mathbf{X}$. Then the model is fit by finding the values of β that minimize the cost function. Ideally we would like to be able to solve for $\hat{\beta}$ analytically, however this is not always possible. Therefore we have to apply numerical methods to compute the minimum. This becomes an optimization problem.

In this article we start by presenting a theoretical background of the machine learning algorithms we will use, which are the FFNN, and logistic regression. We relate these problems to the optimization problem, and present the methods for regression and classification. In *regression*, the output is a quantitative measurement, and we want to find a functional relationship between an input data set and a reference data set. In *classification*, the outputs are divided into two or more classes, and we want to find a model that assigns inputs into one of these classes. Finally we present our results, followed by a discussion of the various methods before we giving a conclusion.

2. Theory

2.1. **The optimization problem.** Optimization problems lie at the heart of most machine learning approaches. This is because when doing regression or classification with different methods, we often end up with a cost function that we wish to minimize. If an analytical expression cannot be found for the minimum of this cost function, we would have to use numerical methods to find it. This is where stochastic gradient descent(SGD) comes in. First let us explain gradient descent(GD), starting from an initial value, gradient descent is run iteratively to find the optimal values of the parameters to find the minimum value of the given cost function. The iterative step is given by,

$$\beta_{k+1} = \beta_k - \eta \nabla_\beta C(\beta_k), k = 0, 1, \cdots \tag{1}$$

where η is called the learning rate, which is chosen before starting the algorithm, $\nabla_\beta C(\beta_k)$ is the gradient of the cost function. Before starting the algorithm we have to initialze β_0, this can be done by filling β_0 with random values. GD is guaranteed to approach arbitrarily close to the global minimum if the cost function we want to minimize is convex, provided the learning rate is not too high.

SGD is a variant of gradient descent, stochasticity is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$. We can approximate the gradient by replacing the sum over all data points with a sum over the data points in the minibatches picked at random in

each gradient descent step, the cost function becomes:

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

The descent step in SGD looks like,

$$\beta_{j+1} = \beta_j - \eta_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{2}$$

We pick k randomly, with equal probability form $[1, n/M]$. An iteration over the number of mini batches (n/M) is commonly known as an *epoch*. I have implemented an adaptive learning rate, which is given by,

$$\eta_j(t; t_0, t_1) = \frac{t_0}{t + t_1} \tag{3}$$

Here $t = e \times m + i$, where e denotes the current epoch, $t_0, t_1 > 0$ are two fixed numbers, m is the number of minibatches, and $i = 0, ..., m - 1$. This function will go to zero as the number of epochs increases. SGD with minibatches is computationally effective, as we are computing the gradient on small random sets of the training data,

Another version of SGD is SGD with momentum. I have also implemented this version of the algorithm. The SGD is almost always used with a momentum or inertia term that seres as a memory of the direction we are moving in parameter space. The descent step in SGD with momentum is given by,

$$\mathbf{v}_j = \alpha \mathbf{v}_{j-1} + \eta_{j-1} \nabla_\beta C(\beta_{j-1}) \tag{4}$$

$$\beta_{j+1} = \beta_j - \mathbf{v}_j \tag{5}$$

We have introduced a momentum parameter α, with $0 \leq \alpha \leq 1$. Momentum optimization cares about the previous gradients, at each iteration, it adds the local gradient to the momentum vector $\mathbf{v}_j$, and it updates β by simply subtracting this momentum vector. A typical momentum value is $\alpha = 0.9$. This method is typically faster than the SGD method.

2.2. **Feed Forward neural networks.** A feed forward neural network, also known as a *multi layer perceptron(MLP)*, is a neural network where the signal only flows in one direction, from the inputs to the outputs. It is an example of an *artificial neural network(ANN)*, an ANN is an information processing model that is inspired by the way our biological nervous system functions, that is, the way in which the brain processes information. ANN were first introduced back in 1943 by the nerophysiologist Warren McCulloch and the mathematician Walter Pitts. The idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed some activation threshold in order to yield an output. If the threshold is not overcome, the neuron remains inactive, that is, the output is zero.

2.2.1. *Mathematical Model.* An MLP is composed of three or more layers, the first layer being the *input layer*, then one or more *hidden layers*, and the final layer is an *output layer*. Each layer can have an arbitrary number of *nodes*, or *neurons*. We can represent the behaviour of a neuron as a mathematical model. One neuron can take in multiple values and return one value. Let us suppose that we have n input values x_i where $i \in [o, n]$, then the output a is calculated as

$$a = f\Big(\sum_{i=1}^{n} w_i x_i + b\Big) = f(z) \tag{6}$$

where z is defined as $\sum_{i=1}^{n} w_i x_i + b$, w_i are the weights of each input, and b is the bias term. The bias is a measure of how easy it is to get the neuron to fire.f is what we call an activation function. The wights, bias and activation functions are chosen by us, usually depending on the problem we wish to solve. An example of an activation function is logistic *Sigmoid* function, given by

$$Sigmoid = \sigma(z) = f(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

If the neural network has L layers, we can write this expression in matrix vector form. For a layer l we can calculate the vector of z values $\hat{z}^l$ as

$$\hat{z}^l = (\mathbf{W}^l)^T \hat{a}^{l-1} + b^l,\ l \in [1, L] \tag{8}$$

Where $\mathbf{W}$ is the matrix of the weights for all the neurons in layer l, this matrix has dimension given by the number of neurons in the previous layer $l-1$ times the number of neurons in the current layer l. a and b are written in vector form. Here we define $\hat{a}^0$ as the input to the neural network, then we can calculate $\hat{a}^l$ as $f(\hat{z}^l)$ for starting from $l = 1$ until $l = L$, where L is the output layer.

2.2.2. *Feed forward.* Now we are ready to train the model. For each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer, this is called the *feed forward algorithm*. We use this to make predictions with the neural network. Starting with the input $\hat{a}^0$, we calculate $\hat{z}^1$, and from there $\hat{a}^1$, which we can use to find $\hat{z}^2$ and so on. We continue in this way until we have found $\hat{a}^L$, this is then the output.

2.2.3. *Activation functions.* The **output layer** is an important part of the neural network. In the output layer, the activation function we use depends on the type of problem we want to solve. If we are doing a classification problem, then we would apply the softmax activation function to the output, the softmax function is given by,

$$softmax = f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{9}$$

If we are doing binary classification, then we would typically use the sigmoid activation function given by (6) on the output. Although we could still use the softmax function.

If we are doing a regression problem we would simply use no activation function at all on the output layer. However it is possible to use the identity activation function on the output layer to make the predictions, the identity function is given by $f(z) = z$. These activation function functions are specific for the output layer only.

When it comes to the **hidden layers**, we can apply any activation function we want. The choice of activation function in the hidden layers is something that we have to choose. Some examples of activation functions are,

- The hyperbolic tangent function, $f(z) = tanh(z)$
- The RELU function(rectified linear unit), $f(z) = max(0, z)$
- The Leaky RELU function(leaky rectified linear unit)

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

- The arctan function, $f(z) = arctan(z)$
- The sigmoid, shown in (6) can also be used
- The sine function, $f(z) = sin(z)$

2.2.4. *Back propagation.* In order for the neural network to give good results, we have to find a good set of weights and biases. To find the best values for all the weights and biases, we usually use the back propagation algorithm. Back propagation is the workhorse of learning in neural networks. The goal is to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the cost function C with respect to any weight w or bias b in the network. This tells us how quickly the cost changes when we change the weights and biases.

The algorithm consists if four equations, we start by defining the quantity $\hat{\delta}^l$, which we call the error in the l^{th} layer. First we calculate the error in the output layer $\hat{\delta}^L$, this is done by calculating

$$\hat{\delta}^L = \frac{\partial C}{\partial(\hat{a}^L)} \odot f'(\hat{z}^L) \tag{10}$$

Here $\odot$ denotes the Hadamard product. Now we can compute the remaining δs with the equation,

$$\hat{\delta}^l = \left(\mathbf{W}^{l+1}\right)^T \hat{\delta}^{l+1} \odot f'(\hat{z}^l) \tag{11}$$

The quantity δ_j^l has the useful property that it can be related to the partial derivatives of the cost functions, that is we can calculate,

$$\frac{\partial C}{\partial b_j^L} = \delta_j^l \tag{12}$$

and

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{l-1}\delta_j^l \tag{13}$$

Now we are ready for the back propagation, with equations (11) and (12), we know the derivative of the cost function with respect to the biases and

weights. We use a SGD optimization algorithm to update the weights and biases using the equations,

$$w_{jk}^1 = w_{jk}^1 - \eta \delta_{jk}^l a_k^{l-1} \tag{14}$$

$$b_j^l = b_j^l - \eta \delta_j^l \tag{15}$$

The algorithm is called back propagation because we compute the error vectors δ^l backwards, starting from the output layer L.

2.2.5. *Cost functions.* The cost function that we wish to minimise will depend on whether we are solving a regression problem, or a classification problem. For a regression problem, the cost function is usually given by the mean squared error(MSE),

$$MSE = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2 \tag{16}$$

here y_i is the actual data, and $\hat{y}_i$ is the date points predicted by the model. For a classification problem the cost function is given by the cross entropy,

$$CE(\mathcal{D}) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \; log(p_k(x_i)) \tag{17}$$

It is given for a data set $\mathcal{D}$, with N data points (consisting of x and y), where y has K number of classes and $p_k(x)$ gives the probability of class k given the input x.

2.3. **Logistic Regression.** Logistic regression, also called *logit regression* is commonly used to estimate the probability that an instance belongs to particular class k. It is a linear method for classification. Since our predictor takes values in a discrete set, we can always divide the input space into a collection of region labeled according to the classification. Logistic regression can be generalized to support multiple classes. This is called *softmax regression* or *multinomial logistic regression.*

2.3.1. *Softmax regression.* We will focus on softmax regression since the data we want to classify in this article is the data set representing handwritten digits from zero to nine. The method can be described as,

- Given an input $\mathbf{x}$, the softmax regression model first computes a **score** $s_k(\mathbf{x})$ for each class k using the equation,

$$s_k(\mathbf{x}) = (\beta_k)^T \mathbf{x} \tag{18}$$

  where β_k represents a specific class k.
- Once we have computed the score of every class for the instance $\mathbf{x}$, we can estimate the probability $\hat{p}_k$ that the instance belongs to class k by running the scores through the softmax equation, which is given by (9), however we repeat it here for notational purposes.

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K} exp(s_j(\mathbf{x}))} \tag{19}$$

  Where $\sigma(\cdot)$ is the sigmoid function given by (7). K is the total number of classes.

- The softmax regression classifier predicts the class with the highest estimated probability. We make the prediction, $\hat{y}$ by

$$\hat{y} = argmax_k \ \hat{p}_k = argmax_k \ \sigma(s(\mathbf{x}))_k \tag{20}$$

Now we look at the training of the model. We wish to have a model that estimates a high probability for the target class (and a low probability for the other classes). This can be achieved by minimizing our cost function given by the cross entropy (17). With this notation the cross entropy looks like,

$$C(\beta) = -\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \ log(\hat{p}_k^{(i)}) \tag{21}$$

Here $y_k^{(i)}$ is one, if the target class for the i^{th} class is k, otherwise it is zero. The gradient of this cost function with respect to β_k is given by,

$$\nabla C(\beta_k) = \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)} \tag{22}$$

Now that we have know the cost function, and its gradient for every class, we can use SGD with minibatches or one of its variants to find the parameter β that minimizes the cost function.

2.3.2. *Binary case.* In the special case where $K = 2$, we can show that softmax regression reduces to logistic regression. Define the probabilities by,

$$P(y = i|\mathbf{x}) = \phi_i(\beta_i^T\mathbf{x}) = \frac{exp(\beta_i^T\mathbf{x})}{\sum_{j=1}^{2} exp(\beta_j^T\mathbf{x})} \tag{23}$$

With $K = 2$ we get,

$$\begin{aligned}\phi_1(\beta_1^T\mathbf{x}) &= \frac{exp(\beta_1^T\mathbf{x})}{exp(\beta_1^T\mathbf{x}) + exp(\beta_2^T\mathbf{x})} \\ &= \frac{1}{1 + exp(-(\beta_1 - \beta_2)^T\mathbf{x})} \\ &= \sigma(-\beta^T\mathbf{x})\end{aligned}$$

$$\begin{aligned}\phi_2(\beta_2^T\mathbf{x}) &= \frac{exp(\beta_2^T\mathbf{x})}{exp(\beta_1^T\mathbf{x}) + exp(\beta_2^T\mathbf{x})} \\ &= \frac{1}{1 + exp((\beta_1 - \beta_2)^T\mathbf{x})} \\ &= \sigma(-\beta^T\mathbf{x})\end{aligned}$$

Where we have used the fact that $\sum_{i=1}^{K} \phi_i = 1$ and $\beta = \beta_1 - \beta_2$. This shows that softmax regression is just a generalization of logistic regression.

3. Method

3.1. **Regression.** For the regression task, the data set that we will be studying is generated by the Franke function, which is a real multivariate function, $f : \mathbb{R}^2 \to \mathbb{R}$. This is the same data that we studied in project one. In order to make the data more realistic we add noise, that is normally distributed. We divide the data set into *training data* and *test data*, we use 20% of the data for testing. We perform the training on our training data and evaluate the performance of our predictor on the test data. To measure how well our model performs, we use the mean squared error(MSE) and the R^2 score. These are the same risk metrics we introduced in project on. Another important step is scaling the data, just as we have done in project one. I have used python's built in function *standardscaler* o scale the data.

To find the optimal parameter $\hat{\beta}$ we have to minimize the cost function, in this case the cost function is given by the MSE, shown in equation (15).

- **SGD** The first method to find β is by using SGD, we use the iterative method to find the value of β that minimizes the MSE. We perform OLS regression and Ridge regression. For OLS, the gradient of the cost function is given by,

$$\nabla_\beta C(\beta) = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) \tag{24}$$

  And in this case we even have an analytical expression for $\hat{\beta}$, given by,

$$\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y} \tag{25}$$

  For ridge regression, the gradient of the cost function is similar to the one for OLS, except we now add a regularization term, λ, the cost function is given by

$$\nabla_\beta C(\beta) = \frac{2}{n}(\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y})) + 2\lambda\beta \tag{26}$$

  And the analytical expression for finding $\hat{\beta}$ is,

$$\hat{\beta} = (\mathbf{X^T X} + \lambda\mathbf{I})^{-1}\mathbf{X^T y} \tag{27}$$

  Here $\mathbf{X}$ is the design matrix, n is the sample size, and $\mathbf{I}$ is the identity matrix. Once we calculate the gradients, we can find β using the iterations shown in equation (1).
- **Neural Network** The second method we will use is the feed forward neural network. To perform regression we will use the identity function as the activation function in the output layer. We will be looking at different activation functions in the hidden layer. The output layer will have one neuron.

We have many parameters that we have to choose, like the number of epochs, the batch size, the learning rate and lambda. We will find the optimal values for them by doing a grid search.

3.2. **Classification.** For the classification task, the data set we will be studying is the MNIST dataset. This is a set of 70,000 small images of digits handwritten by high school students and employees of the US Central Bureau. Each image is labeled with the digit it represents. The data represents digits from zero to nine, so there are ten categories of classes. Given the grayscale values for the pixels of the digitized image of the handwritten digit, we want to predict its class label. To measure how well our model performs, we use the *accuracy score*, the accuracy score is given by,

$$\text{Accuaracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}$$

The accuracy score tells how many labels our model was able to guess correctly. Here n is the total number of targets.

In this article we will be using a data set consisting of 1797 images, each image has $8 \times 8 = 64$ pixels. The size of the design matrix is $\mathbf{X} \in \mathbb{R}^{1797 \times 64}$. We divide the data set into training data and test data, we will use 20% of the data for testing. This means that we will use 1437 images for training, and 360 images for testing. We convert the *labels* or *target* vector y into a one-hot vector. This is a binary vector, where the vector y will have a 1 on the index which represents the label of the image. For example if the label is five, the the vector y will have a 1 on the fourth index(indexing starts from zero), and zeros everywhere else. This is called label encoding.

To find the optimal parameter $\hat{\beta}$ we have to minimize the cost function, in this case the cost function is given by the cross entropy, given by (17).

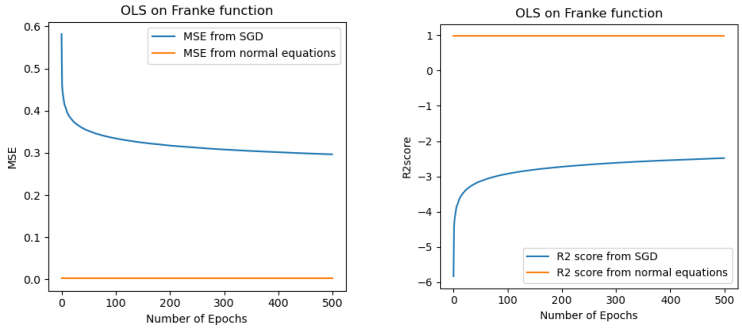
- **Neural Network** The first method we will use to find $\hat{\beta}$ is the feed forward neural network. To perform classification we will use the softmax activation function, given by (9), in the output layer. For binary classification, we can use the sigmoid function (7) in the output layer. We will have 10 neurons in the output layer, this is because we have 10 categories, i.e, the digits from $0 - 9$.
- **Logistic regression** The second method we will use of the logistic regression, or in this case it would be multinomial logistic regression since we have 10 classes. We will use the SGD method to find the minimum of the cost function

Of course there are many parameters that we have to choose, just like in the regression case. We will be looking at the effect of the various parameters.

4. Results

4.1. **Regression case.** The results given here are produced from the Franke function with a sample size of $n = 100$, a polynomial degree, $d = 5$ and Gaussian noise equal to 0.01.

4.1.1. *SGD results.* For OLS with SGD with momentum, I have used 500 epochs, and a mini-batch size of 20. I have calculated $\hat{\beta}$ using SGD and the normal equations to see how close the iterative method is the analytic $\hat{\beta}$, then I plot the MSE and the R^2 score to see how well the fit is. I make this calculation at every epoch. In figure 1a, we ca see that the MSE for the normal equations is zero throughout, this shows that my $\hat{\beta}$ is great fit to the data points. The SGD does not give an MSE of zero, however we



(A) Plot of MSE (B) Plot of R^2 score

FIGURE 1. Results of OLS regression on the franke function, produced with 500 epochs, and a batch size of 20 and ofcourse $\lambda = 0$.

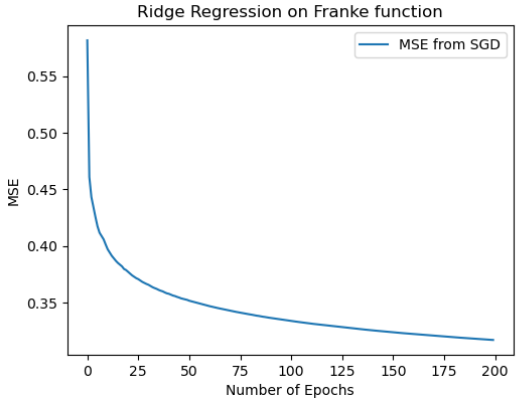


FIGURE 2. Results of ridge regression on the franke function, produced with 200 epochs, and a batch size of 20, with $\lambda = 0.01$

can see that as the number of epochs increases, the MSE is going towards zero. Since SGD is an iterative scheme, we might have to iterate a bit more to reach an MSE of zero. The learning rate used is an adaptive one, it is given by (3), so I have not changed the learning rate manually in this case. Perhaps using more epochs, the MSE would get closer to zero, or maybe I can change the number of mini batches. Similarly figure 1b shows the R^2 score of $\hat{\beta}$ from the SGD and the normal equations. As we can see the normal equations give an R^2 of one, so my model fits well to the data points. As for the SGD method, we can see that as the number of epochs increases, the R^2 score gets closer to one. If I iterated a bit more I could get closer to the analytical $\hat{\beta}$.

For the ridge regression, I used $\lambda = 0.01$, 200 epochs, and a batch size of 20. Figure 2 shows the results. We can see that as the number of epochs increases, the MSE decreases. Starting with an MSE of 0.55, it goes down

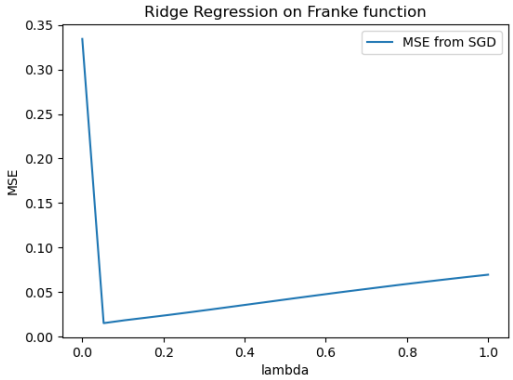


FIGURE 3. Results of the ridge regression on the franke function, produced with 100 epochs, and a batch size of 20. Here we have plotted 20 values of $\lambda \in [0, 1]$, and the corresponding MSE, in order to find the optimal λ.

to 0.22 with just 200 epochs, it goes towards zero. This shows that as the number of epochs increases, we get a better fit, which is what we would expect, as more epochs means that $\nabla C(\beta)$ is calculated several times, giving a better approximation to the actual value.

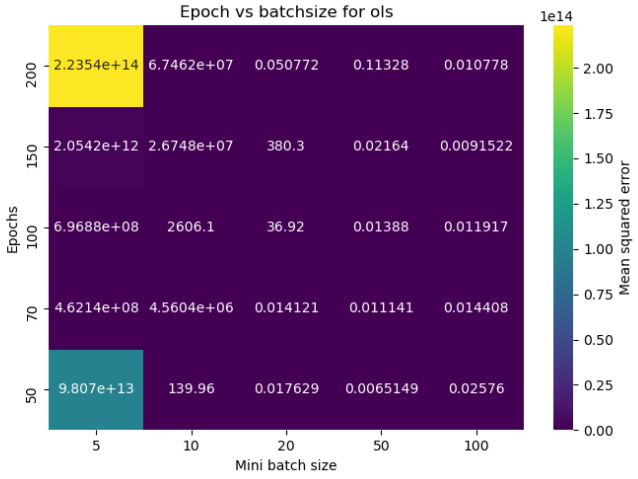
The value of λ is significant for ridge regression. I have tried to find the optimal λ, that is, a value of λ such that the MSE is as close to zero as possible. Figure 3 shows this, I have plotted different λ's and their MSE. We can see from the figure that the MSE is lowest for $\lambda \approx 0.05$. Using this λ we would get the best fit with 100 epochs and a batch size of 20.

In addition to λ, there are other parameters in the iterative scheme that we also have to fine tune to find the best model. I have tried to find the optimum number of epochs and batch sizes for OLS and ridge. Figure 4 shows this.The MSE is lowest for 50 epochs and a mini batch size of 20 for OLS. For Ridge regression the MSE is relatively lower than for OLS, we can use a batch size of 50, and 70 epochs for best results.

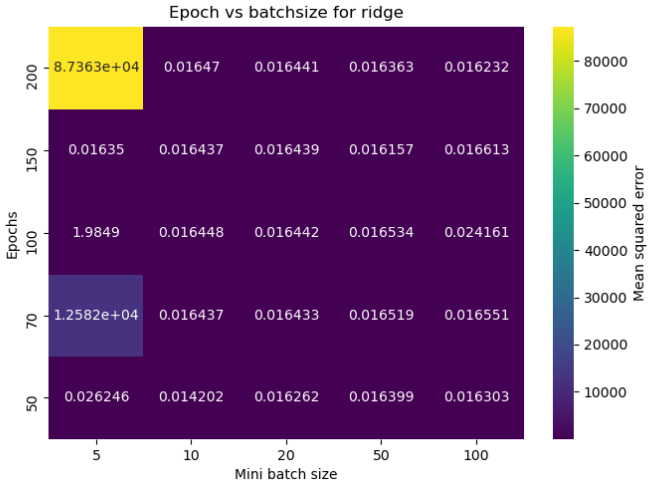
4.1.2. *FFNN results.* All results using FFNN for regression is produced with one input layer, two hidden layers, and one output layer.

Performing regression on the franke function with a FFNN, I get the results shown in figure 5. I have made a plot of the MSE and the R^2 score. Figure 5 shows that the MSE goes towards zero, while the R^2 score goes towards one as the number of epochs increases. This shows that the model is a good fit. I have used 100 epochs, with more epochs, I would get better results. Since there are many parameters in a FFNN, I have done a grid search to find the optimal value of λ and η.

Figure 6 shows the results of my grid search to find the best λ and η. I have used the RELU activation in the hidden layers. In my own implementation I was unable to implement the RELU function properly, I get an error of overflow when I use my own RELU function. I have used *sklearn's MLPRegressor* with RELU activation to get these results. I found that the best $\lambda = 0.316$ and the best $\eta = 0.01$.



(A) Epoch vs batch size for OLS on the franke function.

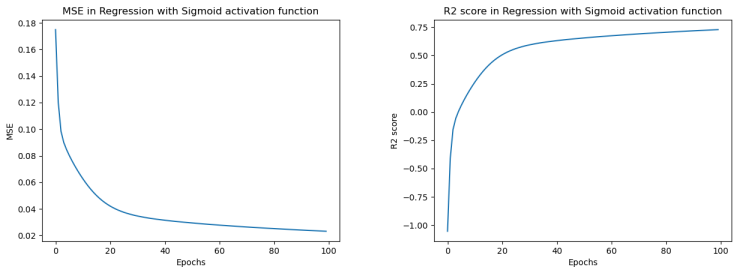


(B) Epoch vs batch size for Ridge on the franke function, with $\lambda = 0.07$

FIGURE 4. Grid search to find the optimum number of epochs and batch-size. This is done for OLS and Ridge regression.

4.1.3. *Study of Activation functions.* In the FFNN, the kind of activation function we use in the hidden layers has a significant effect on the results. I have studied different activation function in the hidden layers. In my own implementation, I get an overflow error when I use RELU or LeakyRelu, therefore I have studied the activation functions $tanh(x)$, $sin(x)$ and $arctan(x)$.

Figures 7,8 and 9 show the results of the various activation functions. We can see that the parameters λ and η play a large role in the results. Figure 7 shows the MSE with $tanh(x)$ as activation. For the optimal parameters, the MSE is much lower than for other parameters. IN figure 7a, the MSE



(A) Plot of MSE (B) Plot of R^2 score

FIGURE 5. Results of regression on the franke function using a FFNN, with sigmoid activation in the hidden layers, produced with 100 epochs, $\eta = 0.5$ and $\lambda = 0$.

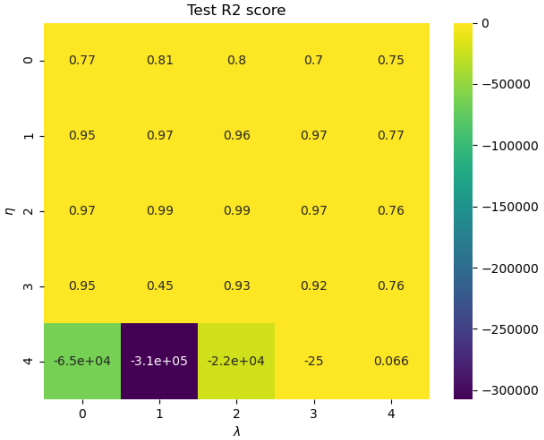
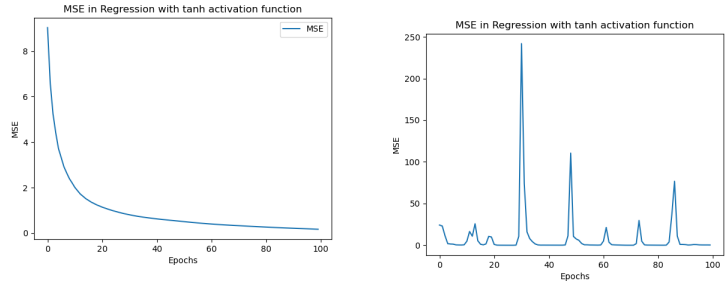


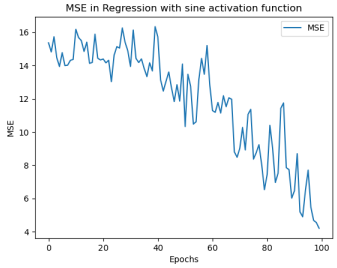
FIGURE 6. R^2 score with different values of λ and η for regression on franke function using FFNN with 100 epochs and ReLU activation in hidden layers.



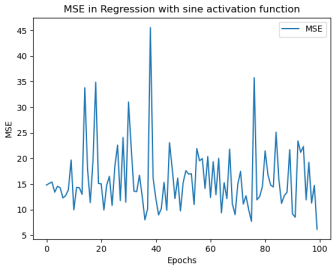
(A) Results with the optimal parameters, $\lambda = 0.316$ and $\eta = 0.01$

(B) Results with $\lambda = 0.4$ and $\eta = 0.1$

FIGURE 7. Regression on franke function with FFNN, using $tanh(x)$ activation in the hidden layers, produced with 100 epochs

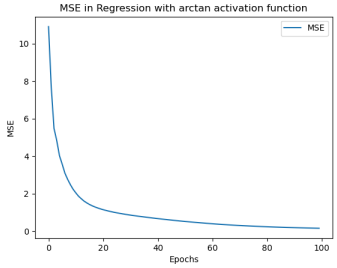


(A) Results with the optimal parameters, $\lambda = 0.316$ and $\eta = 0.01$

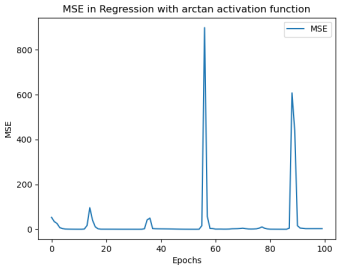


(B) Results with $\lambda = 0.4$ and $\eta = 0.1$

FIGURE 8. Regression on franke function with FFNN, using $sin(x)$ activation in the hidden layers, produced with 100 epochs



(A) Results with the optimal parameters, $\lambda = 0.316$ and $\eta = 0.01$



(B) Results with $\lambda = 0.4$ and $\eta = 0.1$

FIGURE 9. Regression on franke function with FFNN, using $arctan(x)$ activation in the hidden layers, produced with 100 epochs

goes towards zero, as the number of epochs increase. This means the model is improving. In figure 7b, the MSE is unstable when we do not use the optimal parameters. Even if we increase the number of epochs, the MSE does not decrease.

Figure 8 shows the behaviour with $sin(x)$ activation. Figure 8b shows unstable behaviour, this kind of behaviour is expected as sine is a periodic function, $tanh(x)$ and $arctan(x)$ are also periodic functions. However we can see that with the optimal paramters, the MSE is getting closer to zero.

Figure 9 shows similar behaviour with the arctan acitivation. The MSE decreases as the epochs increase for the optimal paramters. This shows similar behaviour to the $tanh(x)$ function, as it is shown in figure 7. We can also see that $tanh(x)$ and $arctan(x)$ are preferable to use as activation in the hidden layer, since they show stable behaviour, compared to the $sin(x)$ function.

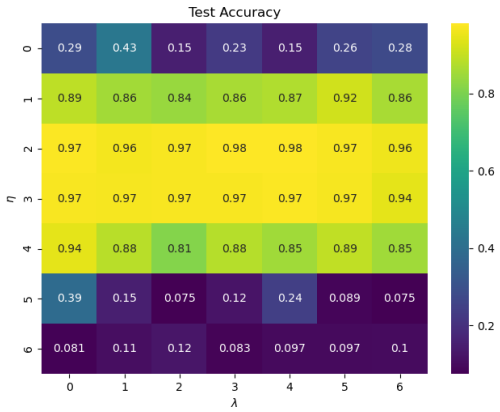


FIGURE 10. Accuracies with different values of λ and η for classification with MNIST digits, produced using 100 epochs and bacth size of 20.

4.2. **Classification case.** Now we present the results from classification of the MNIST handwritten digits.

4.2.1. *FFNN results.* All results using the FFNN for classification are produced with one input layer, one hidden layer, and one output layer. There are 50 neurons in the hidden layer. Since the FFNN has many parameters, first we perform a grid search to find the optimal values for λ and η.

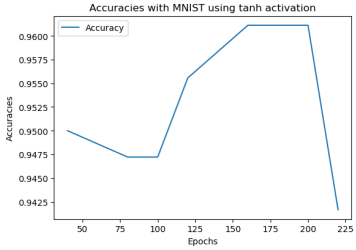
Figure 10 shows the grid search to find the optimum values for λ and η. These results were obtained by using *sklearn's MLPClassifier*, in python. From scikit-learn's grid search I find that the best $\lambda = 0.01$ and the best $\eta = 0.01$, they get an accuracy of 0.972, but when I used these value in my own code, I got an accuracy of 0.9611, which is still good. In my own grid search, I found that the best $\lambda = 0$ and the best $\eta = 0.01$, with an accuracy of 0.94. When I did my own grid search, I used the sigmoid as activation in the hidden layer. Some values of λ and η give me an overflow. I also tried to find the optimum epochs and mini batch size.

From figure 11, I find that the best number of epochs is 100, and the best batch size is 120, using this and the optimum value of $\lambda = 0.01$ and $\eta = 0.01$, I get an accuracy of 0.95.

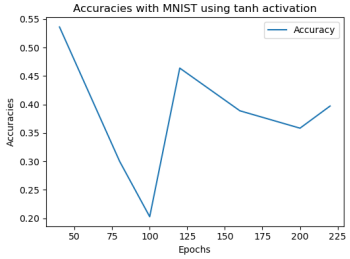
4.2.2. *Study of activation functions.* Just like for the regression case, I also looked at activation functions in the hidden layer, and how it affects the accuracy of the model. I did not manage to get RELU and LeakyRELU to work in my own implementation, I have studied the same activations as in the regression case, namely, $tanh(x), sin(x)$ and $arctan(x)$.

Lets look at the results from using $tanh(x)$ in the hidden layer. From figure 12a we can see the general pattern that as the number of epochs increases, the accuracy does increase. We almost get an accuracy of 1.0, for 170 - 200 epochs, we get an accuracy of 0.96, which is great. In figure 12b, we can see that with different parameters, no matter how many epochs we

FIGURE 11. Accuracies with different epochs and batch sizes, produced with the optimum $\lambda = 0.01$ and $\eta = 0.01$

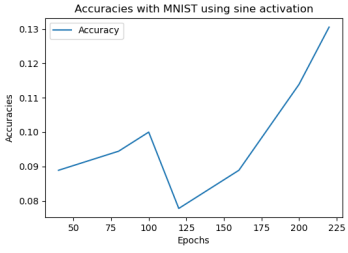


(A) Results with the optimal parameters, $\lambda = 0.01$ and $\eta = 0.01$

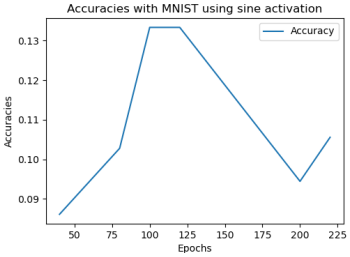


(B) Results with $\lambda = 0.1$ and $\eta = 0.3$

FIGURE 12. Classification on MNIST with FFNN, using $tanh(x)$ activation in the hidden layers, produced with 100 epochs

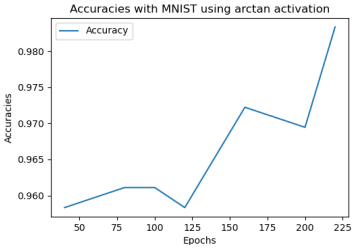


(A) Results with the optimal parameters, $\lambda = 0.01$ and $\eta = 0.01$

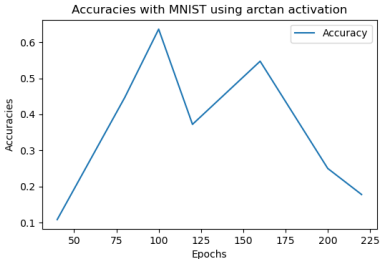


(B) Results with $\lambda = 0.1$ and $\eta = 0.3$

FIGURE 13. Classification on MNIST with FFNN, using $sin(x)$ activation in the hidden layers, produced with 100 epochs



(A) Results with the optimal parameters, $\lambda = 0.01$ and $\eta = 0.01$



(B) Results with $\lambda = 0.1$ and $\eta = 0.3$

FIGURE 14. Classification on MNIST with FFNN, using $arctan(x)$ activation in the hidden layers, produced with 100 epochs

have, the maximum accuracy we can get with $\lambda = 0.1$ and $\eta = 0.3$ is 0.50, with more epochs, the accuracy goes down.

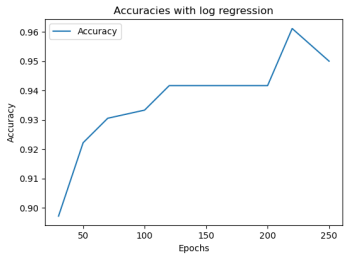
In the case of $sin(x)$, we get poor results. Even by using the optimal parameters, we can only get an accuracy of around 0.13. This is shown in figure 13a. Although we can see that the accuracy curve is begining to increase after 125 epochs. Maybe if we increased the number of epochs, we would get an accuracy closer to one. Figure 13b shows an accuracy of 0.13 with 125 epochs, at 225 epochs, we can see that the curve is beginning to go up again, maybe we could have more epochs.

With $arctan(x)$ used as the activation, the results are shown in figure 14. From figure 14a, the accuracy is pretty close to 1.0 when the number of epochs is 220, and we use the optimal values for λ and η. This is a great result. Without the optimal parameters, the results are inaccurate. We managed to get an accuracy of 0.7, even though we increase the epochs, the accuracy does not seem to improve. In order to get the best results, it would pe preferrable to use $arctan(x)$ or $tan(h)$ as activations in the hidden layer, $sin(x)$ does not give a high accuracy.

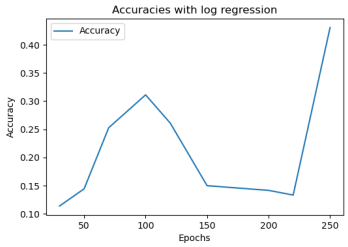
4.2.3. *Logistic regression results.* We now present the results gotten form logistic regression, or multinomial logistic regression, to be more specific.

With 100 epochs, a batch size of 20, $\lambda = 0.01$, and $\eta = 0.01$, I get an accuracy of 0.93611, when I use the logistic function that I have implemented. In figure 15, I have plotted epochs vs accuracies. In figure 15a, we can see that we get an accuracy of almost 0.96 with 220 epochs. However in figure 15b, we only manage to get an accuracy of 0.40. This shows that the choice of parameter is important in getting good results.

I have also looked at the batch size vs accuracies. This result was produced with 100 epochs. In that context, we would expect to get better results when the minibatch size is small, this means there are more minibatches to train the model, and so we would get better results. We get best results with a batch size of 50. To get better results with a bacth size of 200, we would have to increase the numbe of epochs from 100, to maybe 300 or 400. As we



(A) Results with $\lambda = 0.01$ and $\eta = 0.01$



(B) Results with $\lambda = 0.3$ and $\eta = 0.8$

FIGURE 15. Logistic regression on MNIST, plotting epochs vs accuracies, produced with a batch size of 20.

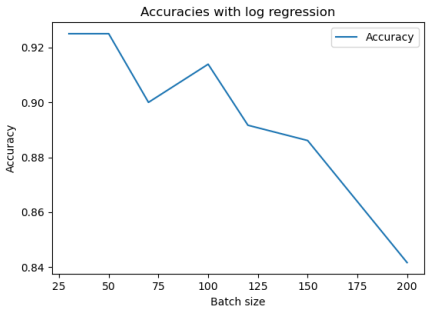


FIGURE 16. Logistic regression on MNIST, plotting accuracies with different batch sizes, produced with $\lambda = 0.01$ and $\eta = 0.01$ and with 100 epochs.

can see, the number of epochs is also important in choosing the bach size for best results.

5. Discussion

We now discuss our results from both of the methods. The SGD has two main benefits; firstly, it introduces randomness which decreases the chances of the scheme getting stuck in local minima. Second, if the minibatch size is small compared to the size of the data set, the computation is much cheaper, as we are only computing the data over the data points in the minibatches. This method is better suited for cases where there are a large number of features, or too many training instances to fit in memory. In this case, MSE and Cross entropy are convex functions, this means there are no local minima, just one global minimum. In addition if the learning rate is too small, then we would need many iterations to converge, which would take too long. If the learning rate is too high, we might end up farther away from the minimum, towards larger and larger values, leading towards divergence.

To find the optimal $\hat{\beta}$ by the normal equations, we have to compute the inverse of $\mathbf{X^T X}$, which is an $n \times n$ matrix. Inverting a matrix is a computationally expensive operation, typically requiring $O(n^3)$ computations. The normal equations get very slow when the number of features increases.

On the other hand, this equation is linear with regards to the number of instances in the training set. Once we have trained the set, we can make the predictions in linear time. Making predictions on twice as many instances will just take roughly twice as much time.

Neural networks can be seen as natural, more powerful extensions of supervised learning methods, such as linear and logistic regression or the softmax methods. A neural network is able to approximate any complex function, this is as a result of the universal approximation theorem. However, neural network are overparametrized, the flexibility of neural networks is also one of their main drawbacks. There are just too many hyperparamteres to tweak. This is something that we have experienced in this article. We have to find the optimal value for the number of epochs, the minibatch size, the learning rate, and regularizatin paramter. Even in a simple FFNN we would have to choose the number of hidden layers, the number of neurons in each layer, the type of activation function in each layer, the weights and bias initialization and much more. As we have seen in this article, the choice of these parameters is significant on the results obtained from training the data. For example in the case of classification on MNIST, with the parameter choice of $\lambda = 0.01$ and $\eta = 0.01$, we get an accuracy of 0.96, while with $\lambda = 0.1$ and $\eta = 0.3$, we get an accuracy of 0.55. This is quite a dramatic change, therefore this choice of parameters cannot be ignored. In addition neural networks are prone to overfitting the training set, however this can be solved by introducing l_1 or l_2 regularization, just like I have done in my implementation. We can also face the vanishing gradients problem, or the related exploding gradients problem, which makes lower layers difficult to train.

When it comes to activation function in the hidden layers, we can say that $tanhh(x)$ and $arctan(x)$ should be used in the hidden layers. This is also what we saw in the results section. This is because $tanh(x)$ is S-shaped, continuous, differentible, and has output in the range from -1 to 1, this makes each layer's output more or less normalized(i.e centered around zero). With the $arctan(x)$, we converge our derivatives quadratically against zero for large input values, $arctan(x)$ is considered faster than $tanh(x)$. On the other hand $sin(x)$ can be ignored as an activation, the periodic nature of sinusoidal activation functions can give rise to a 'rippling' cost function with bad local minima, this makes training difficult. This is what we saw in the results section. In general, more popular choices are RELU, which is not differentiable at zero. It works well in practice, and is fast to compute.

Logistic regression is simple to implement, it makes no assumptions about distributions of classes in feature space. It has good accuracy when the data set is linearly separable. On the other hand non linear problems cannot be solved with logistic regression, because it has a linear decision surface.

6. Conclusion

In conclusion, we can say that SGD is a better method for regression, compared to the normal equations. The SGD method with minibatches is much faster for large data sets, since we compute the gradients on small

subsets of the data. While normal equations can only perform linear regression, SGD method can be used to train other methods, like a FFNN. The training part o a FFNN is done by updating the weights and biases using the SGD scheme. The FFNN is a better method for classification, than the logistic regression. In my results, for 100 epochs, a batch size of 20, $\lambda = 0.01$ and $\eta = 0.01$, I got an accuracy of 0.9611 with the FFNN, and an accuracy of 0.9361 using softmax regression. This can be explained by the universal approximation theorem, which says that if the activation function used in the FFNN is like a sigmoid function, and the function being approximated is continuous, a FFNN can approximate it pretty well. Therefore a FFNN performs better at modeling the relationship between the handwritten digits and the corresponding labels. Using a FFNN for regression seems like overkill. If we are able to fit a regression model to our data, then we should not bother with using a FFNN, as a FFNN is computationally intensive compared to OLS or ridge regression. To conclude my study of activation functions in the hidden layer, I can say that $tanh(x)$ is the best choice, as it gives a MSE close to zero, and an accuracy close to one, according to my results. In general the activation functions are ranked as LeakyRELU $>$ RELU $>$ $tanh$ $>$ sigmoid. Regardless of our problem, the main factor is the choice of hyperparamters in these methods, like λ, η, number of epochs, number of hidden layers etc. With the optimum choice of these parameters, we can get suitable results with any activation functions, as my results have shown.

7. References

- All codes can be found in the github repository `https://github.uio.no/shafaqns/FYS-STK4155/tree/master/Project2`

- Hjorth-Jensen, M., 2020. Week 40: From Stochastic Gradient Descent To Neural Networks. [online] Compphysics.github.io. Available at: `https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40-reveal.html` [Accessed 8 November 2020].

- Hjorth-Jensen, M., 2020. Week 41 Tensor Flow And Deep Learning, Convolutional Neural Networks. [online] Compphysics.github.io. Available at: `https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41-reveal.html` [Accessed 8 November 2020].

- Nielsen, M., 2020. Neural Networks And Deep Learning. [online] Available at: `http://neuralnetworksanddeeplearning.com/chap2.html` [Accessed 8 November 2020].

- Geron, A., 2017. Hands-On Machine Learning With Scikit-Learn And Tensorflow. O'Reilly Media, pp.chapter 4, 10, 11.

- Wang, R., 2018. Multi-Class Classification And Softmax Regression. [online] Fourier.eng.hmc.edu. Available at: `http://fourier.eng.hmc.edu/e176/lectures/logisticRegression/node2.html` [Accessed

9 November 2020].

- Saha, A., 2020. Comparison Between Logistic Regression And Neural Networks In Classifying Digits. [online] Medium. Available at: https://medium.com/ai-in-plain-english/comparison-between-logistic-regressio [Accessed 9 November 2020].