Draft version November 14, 2020
Preprint typeset using LaTeX style emulateapj v. 12/16/11

CLASSIFICATION AND REGRESSION, FROM LINEAR AND LOGISTIC REGRESSION TO NEURAL NETWORKS

Ø. Strand, I. Rivå

Draft version November 14, 2020

ABSTRACT

We consider a fully connected feed forward Neural network, stochastic gradient descent and multinomial logistic regression. We study how our neural network performs with linear regression on the Franke function and classification on the MINST data set of hand-written numbers. We extensively analyse the hyperparameters for the SGD on the Franke data to optimize our models. All methods are compared with sci-kit Learn's corresponding methods. We found that larger batch sizes with small learning rates achieve the same results as the opposite case, with a significantly smaller computational time, and momentum speeds this up even further. Best achieved R^2 score for linear regression using SGD was 0.83, with the OLS achieving 0.89. On 400 datapoints the OLS performed better than the neural network on Frankedata achieving an R^2 score of 0.862 compared to 0.739 for our neural network, showing that OLS works well with limited data. The multinomial regression method achieved an accuracy score of 0.969, with sklearn's LogisticRegression achieving 0.95. The neural network achieved an accuracy score ~ 0.95.

1. INTRODUCTION

Taking inspiration from nature and traits that have developed and adapted over thousands of years can often lead to good, efficient results. Emerging from the idea of creating a learning process based on Neuroplasticity, the first multi-layered neural network was published by Ivakhnenko and Lapa in 1965. (1) Since then the field of machine learning has grown substantially and neural networks are being used more and more. In the future we predict that the human race will be heavily dependant on this technology, which makes it an extremely relevant topic. The ways in which one can use neural networks are abundant. Image recognition, speech recognition and computer vision. These are just a couple of the many applications there are out there. An understanding of neural networks is a gateway to be able to work intuitively and effectively in all or any of these areas of application.To understand how neural networks work we studied how they are built and used on both linear regression and classification problems. We also saw how Neural networks can be implemented on real data by examining the MNIST data set of hand-written numbers and compare our method with sci-kit Learn's implementations (7). This paper presents feed forward neural networks covering both linear regression and classification, as well as delving into stochastic gradient decent, the method used to optimize the neural network, and logistic regression. It also discusses the choice of activation functions, cost function and the initialization of weights and biases of a neural networks. The paper will first go through the theory and algorithms associated with SGD, neural networks and logistic regression and then how we chose to implement said methods. The results will be presented followed by the discussion section where we will evaluate and compare the different methods and analyse the data. Then follows the conclusion where we will take a look at how our methods can be improved upon and summarize the discussion.

orjanstr@student.matnat.uio.no ,
ingunriv@student.matnat.uio.no

2. THEORY

In this section we will introduce the theory behind stochastic gradient descent(SGD), fully connected feed forward neural network and logistic and multinomial regression.

2.1. *Stochastic gradient descent*

2.1.1. *Gradient descent*

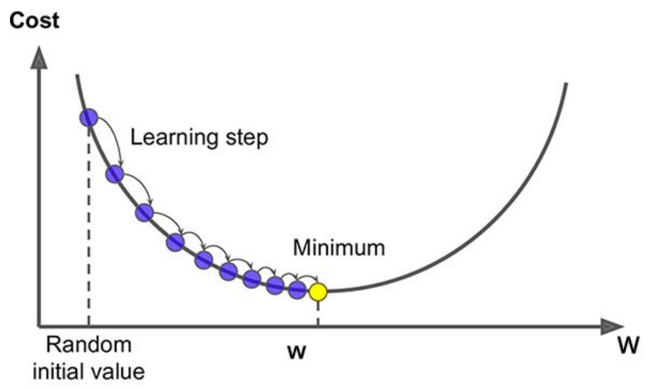


Fig. 1.— Gradient descent. Source: (sng, 2020) (9)

In the previous project we found the set of coefficients that minimizes the cost function analytically. This is, however, not always a feasible solution. The cost function may not always be strictly convex, having several local minima, meaning we cannot simply find the global minimum by setting the gradient $\nabla C(\theta) = 0$. It is therefore better implement a numerical solution for the coefficients, or weights.

One of these solutions is called gradient descent. Figure 1 shows the simple example of gradient descent performed on a convex cost function. The goal of gradient descent is to take steps in the opposite direction of the slope to reach the minimum. The slope is represented by the gradient of the cost function with respect to the coefficients $\nabla C(\theta)$. If the slope is positive it points away from the minimum, and if the slope is negative it points towards the minimum. If we then take a step in the opposite direction of the slope, we would approach the minimum more and more. We wish to decay the step length as we approach the minimum, so we define the step length as $\eta \nabla C(\theta)$, where η represents the learning rate. This rate determines how quickly we decay the step length. The full implementation then becomes

$$\theta \leftarrow \theta - \eta \nabla_\theta C(\theta). \tag{1}$$

This method does, however, come with some problems. First, there's the problem of local minima. Gradient descent is not able to distinguish local minima and global minima, meaning we can get stuck in local minima, resulting in an inaccurate model. A way to overcome this is to increase the learning rate such that the descent method jumps over the local minima, but it could also jump over the global minimum to end up in a local one on the other side. We must thus find a the optimal learning rate such that we end up in the global minumum.

Second, it is computationally demanding to calculate this method. Say we wish to optimize 50 coefficients for 1000 datapoints. Normally we need to iterate 1000 times to reach the minimum. This means that we in total need to make $50 \times 1000 \times 1000 = 5 \times 10^7$ calculations, which would take a long time to do.

We introduce the stochastic gradient descent method to solve these problems, which is the descent method we will be studying in this article.

2.1.2. *Stochastic gradient descent*

In stochastic gradient descent (SGD for short) we, instead of iterating on the entire dataset, select a single random datapoint to calculate the step from. We then move the coefficients, select new random datapoint and calculate the step again. We iterate through this process until we reach a minimum. If we use the same example as before, we would end up with 5×10^7 iterations with 1000 datapoints for regular gradient descent. For SGD, however, we would only iterate over 1 datapoint per step, resulting in only 5×10^4 calculations. If we had 10^6 datapoints, SGD would reduce the calculations by a factor of one million. So this is clearly a powerful tool for reducing computational time.

Another way to do SGD is to select a handful of datapoints called mini-batches instead of only one [2]. The benefit of this method is that we can perform SGD in a more vectorized way, reducing the computational time even more.

2.1.3. *Adding momentum*

In an ideal model, we would take steps directly towards the minimum. In such a case, we could reach convergence faster by increasing the learning rate. Unfortunately we often see that the steps oscillate on their way towards the minimum. This means that an increase in learning rate would only increase these oscillations, and not necessarily help us reach the minimum any faster. A way to combat this effect is to remember the general direction of the steps by taking the weighted average and adding it to the step, pushing it more towards the minimum and allowing larger learning rates:

$$v \leftarrow \gamma v + \eta \nabla_\theta C(\theta), \quad 0 < \gamma < 1 \tag{2}$$

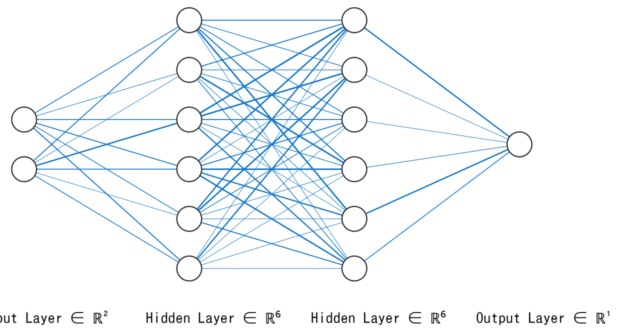
$$\theta \leftarrow \theta - v \tag{3}$$

For further detail on implementation, see section 3.1

[2] Strictly speaking, SGD only uses one datapoint. The method of dividing into mini-batches is called mini-batch gradient descent, but we will refer to it as SGD in this article.

2.2. *Feed Forward Neural Network*

Artificial neural networks are algorithms that have taken inspiration from nature and are constructed in a similar way to neural networks in the brain. The neural network consists of neuron-like "nodes" organized into layers. All of these nodes have individual "weights", "biases" and "activation functions that process the signal the node receives and produces an output. Every node in a layer is connected to the next layer and the previous one as we can see in figure 2



Input Layer $\in \mathbb{R}^2$
Hidden Layer $\in \mathbb{R}^6$
Hidden Layer $\in \mathbb{R}^6$
Output Layer $\in \mathbb{R}^1$

FIG. 2.— A neural network with 3 layers. Source:(2)

The input is fed through the layers and adjusted by the different nodes to compute an output in a step called "feed forward". This output is then compared with our target value and adjustments are made to the weights and biases to improve the model in a process called "back propagation". This is how the neural network learns. We will now examine this process more closely.

2.2.1. *Feed forward*

In feed forward the input is fed to all the nodes in the first layer, where it is multiplied by the corresponding weights and the nodes' individual biases is added. This value, which we call z, was then put through an activation function now serving as the input for the next layer. We have that:

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

$$a_j^l = f(z_j^l),$$

where w is weights, x is input b is bias, $f(z)$ is the activation function and a is the activation. This process is repeated for all the nodes in all the layers all through the network, ending up with the output. Having completed the first pass through our network we have calculated a very inaccurate initial model. We then had to improve this model by the means of back propagation.

2.2.2. *Back propagation*

In back propagation we go back thorough our network to update the weights and biases to improve the prediction power of our network. To do this we first calculate the output error for the last layer L,

$$\delta_j^L = f'(z_j^l)\frac{\partial \mathcal{C}}{\partial (a_j^L)}.$$

When we have that, we can proceed to propagate the errors backwards through the layers using (in matrix notation):

$$\Delta_l = \Delta_{l+1} W_{l+1}^T \circ f'(Z_l).$$

Where Z and Δ are arrays containing all the z and δ values, respectively, for the nodes in the layer. From this we are able to calculate the gradients for the weights and the biases,

$$\nabla W_l = a_l^T \Delta_l$$

$$\nabla b_l = \sum_{i=1}^{n} \Delta_l,$$

where n is the number of inputs. We then update the weights and biases using the gradients:

$$W_l \leftarrow W_l - \eta \nabla W_l$$

$$b_l \leftarrow b_l - \eta \nabla b_l,$$

where η is learning rate. Using stochastic gradient decent we will go through feed forward and back propagation repeatedly to train our network. One problem that can arise during back propagation is exploding or vanishing gradients. To understand how this works we first have to take a deeper look at activation functions.

2.2.3. *Activation function*

First let us look at which activation function we want to employ for the final layer of our neural network and how it affects the model. The activation function decides what is to be sent to the nodes in the next layer, in this case the final output, and it restricts the output from the nodes to a certain range. The activation function we want to choose for the output layer depends on what kind of task our neural network is doing. If we are working with regression we would want to use a linear activation function for the final layer because we are trying to predict a continuous numerical value that is not constrained.

If we are doing classification we would want to use either Sigmoid or Softmax. Both Sigmoid and Softmax (see equation4) outputs a probability distribution, the sum of which is 1 for the sum of the output neurons, but Sigmoid can only be used in binary classification. If you have more than 2 classes you would want to use the Softmax.

Lets now talk about the activation functions in the hidden layers. The activation function has the important role of introducing non-linearity to our network, and there are many functions to choose from depending on your needs. Here we will look at the Sigmoid, the reLU, and the Leaky-reLU.

The Sigmoid function is a well known function because of its early popularity, but it is often discarded in favour of more robust activation functions. The most prominent of its problems is the vanishing gradient problem which can slow down learning in our network considerably. The vanishing gradient problem arises in deep neural networks when you multiply multiple gradients together. An effect of the back propagation algorithm is that to calculate the gradient for the first layer, you will need to multiply as many gradients together as you have hidden layers. This poses a problem if those gradients lie between 0 and 1. This is a known problem for the Sigmoid, since the values are bound between 0 and 1 and the value of its derivative does not exceed 0.25. If this is the case the weights are going to be smaller and smaller as we propagate through our network resulting in gradually impairing the learning the further we are from the output end of our network. One solution is to use a different activation function.

The reLU activation function solves the problem the Sigmoid has with vanishing gradients, since the gradient is either 0 or 1, and is therefore a popular choice, but that does not mean it does not come with some problems of its own. One of the most prominent is the dying reLU problem. If z is a negative value, then the reLU function will return 0. This will cause the loss function to be zero and the weights will remain unchanged, essentially killing the concerning node, and partially stopping the network from learning. one can solve this problem by implementing a "leaky reLU" where , for negative numbers, it returns a small value for the gradient instead of zero.

Another problem akin to the vanishing gradient problem is the exploding gradient problem, which can occur when the weights are not initialized properly. This is also a problem that occurs with deep neural networks. If the weights are initialized very big, even though the output of the activation function may be less than one, the value of the weights will make the gradient explode when sufficiently many of them are multiplied together. This creates a very unstable network, going back and forth between values, never converging. This can be avoided by applying weight regularization but it is also the reason why it is very important to take great care when initializing the neural network.

A problem with these activation functions is that they are not zero centered so the output of the node will always be either positive or negative which will make the gradient of the weights either positive or negative.This means that the path chosen will not be optimal, as some gradients are out of the question. This causes the path to the minimum to oscillate around the optimal straight path which is less than perfect.

2.2.4. *Initialization*

We have to find some way of initializing our weights because if the weights are initialized to zero our network will not learn anything.

One commonly used method is to initialize the weights with random normally distributed numbers. This is an easy approach, but if the weights are initialized too large the neurons can get saturated slowing down learning. This problem is especially prominent when we have a large amount of neurons pr layer, because of the standard deviation. The z from feed forward will have a standard deviation corresponding to the square root of the sum of the individual standard deviations. Considering a network with n input nodes, we have that for a neuron in the next layer, the standard deviation is $\sigma = \sqrt{n}$. For a deep neural network with many layers this will be a large number. To prevent this we chose a different way of initializing our weights. The problem is that the standard deviation increases as we add more nodes, so if we find a way to stabilize the standard deviation we might get a better result. To do this we scale the initialization of the weights, in layer l, with a value $\frac{1}{\sqrt{n_{l-1}}}$, referred to in Xavier et al as standard weight initialization. (3)

In the same paper Xavier et al. introduces a form of initialization that improves upon this standard approach in some settings, it is often called Xavier initialization. It seeks to maintain the activation variance and the gradient's variance in back propagation. The weights are chosen from a random uniform distribution between $\pm\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$. This initialization is mostly used for the Tanh function, and is not the best for the other activation functions, especially the reLU-based ones, as seen in He et al. (4) The paper, shows that He initialization is better when used with reLU-like activation functions. It scales the weight by $\frac{2}{\sqrt{n_{l-1}}}$.

The bias is often initialized to 0, but to be sure that all the nodes are activated and our network does not come to an early stop we initialize it with small number.

2.2.5. *Regularization*

We want to implement regularization on our network to reduce overfitting. A very common form of regularization is weight decay, or L2 regularization, constraining the size of the weight to prevent them from growing too large. With smaller weights, our network will not change too erratically based on noisy inputs, and therefore will be less likely to overfit the training data. Thus creating a more generally applicable model. We might also notice that the L2 penalty is the same one that is used in ridge regression (10) . To implement weight decay the cost function is changed to include regularization by adding a sum of the squared weights. $\mathcal{C} = \mathcal{C}_{\prime} + \lambda\sum_{ij} w_{ij}^2$, making our network prioritize solutions with small weights.

2.2.6. *Cost function*

The cost function represents the value we want to minimize during our training of the network so the choice of cost function depends on what you want to achieve and how you define the performance of your network. When we are solving a regression problem it is very intuitive to use the mean squared error as cost function since minimizing the distance between our model and the target is a good measure of our model.

for classification we want our model to predict a probability distribution, giving us the probability for all of our possible outcomes so we can pick the most likely one, and therefore we would want to use cross-entropy as our cost function as it compares the difference between two probability distributions.

For further detail on implementation of neural network, see section 3.3

2.3. *Logistic regression*

So far we have studied how linear regression works for fitting a line to a dataset. We will now study another regression problem, namely classification, where we instead of fitting a line, place our data into classes. This is called logistic regression.

2.3.1. *Multinomial/Softmax classification*

In this article we will study the case where we have more than two classes to sort our data into. This type of classification is called Softmax, or multinomial, regression. The goal with Softmax regression is to take the input, convert it to a form comparable with the target, and then adjusting a set of weights such that the method will place any input of the required form in the correct class.

The shape of the weight set for Softmax regression is $[n_{features} \times n_{classes}]$, meaning our model $\mathbf{Z} = \mathbf{XW}$ will have the shape $[n_{data} \times n_{classes}]$. This provides us with a class score for each input. The higher the class score for an input, the more evidence we have that the input belongs to said class. Now we wish to normalize this score to rather represent the probability that the input belongs in a certain class. This normalization is provided by the Softmax function, which takes the score and divides by the sum of all scores for the input:

$$f(Z_i) = \frac{e^{Z_i}}{\sum_{j=0}^{k-1} e^{Z_j}} \tag{4}$$

With this normalization, the sum of all probabilities will equal 1. We can then compare this with our target represented in one-hot form. If a target is in one hot form, the element representing the class is set to 1, while the rest is set to 0. We can evaluate our model on the target, $\mathbf{T}$, using the average of all cross entropies as the cost function:

$$\frac{1}{n}\sum_{i=0}^{n-1} T_i \log(Z_i) \tag{5}$$

We can further calculate the derivatives with respect to $\mathbf{W}$, giving us the gradient, in matrix notation:

$$\nabla C(W) = -\frac{1}{n}(\mathbf{X}^T(\mathbf{T} - \mathbf{Z}(\mathbf{W})) - \lambda\mathbf{W}) \tag{6}$$

Note that we added the L2-norm to the gradient for regularization. If $\lambda = 0$ we will have no regularization. We can then perform SGD to find the set of weights that optimize the classification method.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta\nabla C(\mathbf{W}) \tag{7}$$

For further detail on implementation, see sections 3.1 and 3.4

3. METHODOLOGY

3.1. *Setting up our SGD method*

We wanted our SGD method to work similarly to SciKit's method. So we created two separate SGD methods; one for linear regression, and one for classification, taking inspiration from the code provided by M. H. Jensen's lecture notes of week 39 (5). Our goal with the linear regression method was to use SGD to find the optimal set of coefficients, β, for a polynomial fit of the Frankedata, but in a way that can be generalized for other linear regression problems as well. This method will initialize the weights with shape $[n_{features}]$ representing the coefficients of a polynomial fit. It then shuffles the dataset and splits it into b minibatches. In the iteration process it will loop through b randomly sampled minibatches, and perform gradient descent (as discussed in section 2.1) on said batches, with the option of added momentum and penalty. This method uses the mean squared error (MSE) as the cost function, with its gradient in matrix form.

$$C(\beta) = \frac{1}{n}\sum_{i=0}^{n-1}(Z_i - T_i)^2 + \lambda\sum_{j=0}^{p-1}\beta_j^2 \tag{8}$$

$$\nabla C(\beta) = \frac{2}{n}(\mathbf{X^T}(\mathbf{Z} - \mathbf{T})) + 2\lambda\beta \tag{9}$$

Here $\mathbf{T}$ and $\mathbf{Z}$ represent the target and the predicted model $\mathbf{X}\beta$, respectively. We have also added penalty to the cost function.

The classification method initializes the weights as discussed in section 2.3. It then converts the input to the probability form provided by the Softmax function (eq. 4). It then performs SGD in the same fashion as the linear regression case, using cross-entropy as the loss function (equations 5 and 6), providing us with a set of optimized weights for the given classification problem. This method is general, and works both for the binary and the multinomial case.

To both methods, we have added the option of an adaptive learning rate that will shrink as we iterate through epochs. We make the learning rate, η, depend on the number of epochs and batch iterations.

$$\eta = \frac{t_0}{t_1 + t}, t = e \cdot n_{batch} + i \tag{10}$$

Where the index i represents the batch iteration, e represents the current epoch and n_{batch} is the total number of batches we iterate through. The goal of this is to kill the step length after a reasonable amount of time such that we don't move back and forth in the optimal minimum once we have been there a while. With this adaptive learning rate, we can also take larger steps in the beginning, as it will shrink with time, helping us over some local minima on our way.

3.2. *Tuning the hyper parameters for SGD on the Franke dataset*

With our SGD method in place, we started studying the hyper-parameters. The goal with this is to study the effects of each hyper-parameter individually, as well as seeing how they affect each other. We chose to study how they work for the linear regression case, where we used our SGD method to optimize the coefficients, minimizing the loss function (eq. 8) on the Franke dataset. We split the Franke data into training and test sets with a 4:1 ratio. We used the training set to train our model, optimizing the coefficients, before applying the test data to study the error. We compare the results of our method to that of the OLS and Ridge methods discussed in the previous project. (10)

We started off by studying the learning rate for varying batch sizes in order to find a good range of learning rates to study further. We studied both a constant and an adaptive learning schedule, following equation 10. We then selected a learning rate within that range, and its corresponding batch size and learning schedule, and gradually added momentum to study how it helps with convergence. We then, without momentum, gradually added a penalty as well to see if we get a better fit to the test data. We used the R^2-score to measure the error of our model, and compared it to the R^2-scores from the OLS and Ridge methods. Once we felt we had an intuitive understanding of the hyper-parameters, we tweaked all of them together to find an optimal model. We used random search for this, which selects random values from provided sets of hyper-parameters, from ranges based on our analysis, and stores the best result.

3.3. *Neural network*

We started by creating the architecture for our network. Setting up the correct dimensions for the weights and the biases for every layer. We created our neural network so that you can choose between a quick setup with the same number of nodes in all the hidden layers, or by choosing the nodes for each layer separately. Looping through the layers we create a list of weights $= (n_{l-1}, n_l)$. and biases $= (n_l.)$, where $n_l$represents the number of nodes in layer l.

We initialize our weights according to which activation function is being used. If we used the Sigmoid function the weights were initialized with standard initialization, see 2.2.4, but if the activation function was in the reLU-family we used He initialization. We initialized the biases with a small value 0.01.

We then proceeded to define the feed forward method. Both the feed forward method and the back propagation method is done according to M.H. Jensen's lecture notes of week 41 (6) . We start by defining variables z ,a and δ to be of shape $(n_l,\ n_{datapoints})$. We loop through the layers using the equations in section 2.2.1 to calculate the values for the first layer, from then onward we calculate z and a using:

$$z_{l+1} = z_l w_l + b_{l+1}$$

$$a_{l+1} = f(z_{l+1}).$$

For the last z we apply the activation function for the output layer to obtain our output. For the output layer we chose a linear activation function for the linear regression problem and the Softmax activation function for the classification problem. As for the hidden activation functions we tried varying them to see the effect on our model, but we mostly used the Leaky-reLU. We implemented back propagation by first calculating the error for the last layer. $\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial(a_j^L)}$,

then we looped though the layers and computed the rest of the errors. $\Delta_l = \Delta_l W_l^T \circ f'(Z_h)$

The gradient for the weights and the biases are computed according to the formulas described in section 2.2.2. We also multiply both the gradients by a factor of $\frac{1}{n_{datapoints}}$, because when we calculate the gradient of the weights, using $\nabla W_l = a_l^T \Delta_l$, we sum over the inputs to get rid of the input dimension, which for a large amount of inputs would cause overflow. We also added a constraint, $\lambda\sum_{ij} w_{ij}^2$, proportional, to the size of the weights to reduce overfitting, λ is the regularization parameter.This is the L2 regularization(weight decay) we discussed earlier.

The weights and biases are then updated in the as described in section 2.2.2.

For the training of the neural network we used the stochastic gradient descent code example seen in M.H. Jensen's lecture notes of week 41 (6) as it it easier to create a SGD just for our neural network rather than trying to use the code we made previously because of the way we implemented it in our first code.

For linear regression we looked at Frankedata from project 1 (10), that was generated using Franke's function for random uniformly distributed $\boldsymbol{x}, \boldsymbol{y} \in [0,1]$ plus some noise $\boldsymbol{\epsilon} \in N(1, \sigma^2)$, with a total of 400 datapoints, the same amount that we used in project 1, and 20 features, corresponding to degree 5. The data is also scaled and standardized, and we reuse our method from project 1 (10) to split the data into training and testing sets. First we did a random grid search through different hyper-parameters to find a relatively good model so we had something to start from. We then compared our result with sci-kit Learn's implementation of linear regression for neural networks and OLS from project 1, and made a table to organise our results.

We plotted the MSE for increasing epochs for different learning rates to see if the cost function was strictly convex.

We then looked at how changing the penalty λ and the learning rate η had an effect on our model by plotting the result in a heatmap and compared with ridge regression from project 1 (10). We also plotted the MSE Sigmoid, reLU and Leaky-reLU activation functions for a deep neural network with 4 hidden layers to compare them and see if our results fit with existing theory.

For classification we looked at a subset of the MNIST dataset of handwritten numbers with a total of 1797 datapoints from sci-kit Learn. The dataset consist of 10 classes corresponding to the digits 0-9. Each datapoint is an image of a digit with 8×8 pixels all in the range 0-16. To use the dataset in our method we had to transform the y values of target digits to a "one hot vector" to use as our probability distribution for comparing with the output in our neural network. We used random grid search, like for regression, to find some usable values to start with, we then tried varying the values to improve upon this "base" model. When we were sufficiently happy with our accuracy score we compared our method with sci-kit Learn.

3.4. *Multinomial regression on the MNIST dataset*

The goal here is to find an optimal fit for the MNIST dataset, based on the knowledge gained from the analysis of section 3.2. We started off by loading the MNIST dataset from sklearn's dataset library. We then split the dataset into training and test sets like before, training our model on the training set, and testing it on the test set. We converted the target labels to one-hot form, initialized a random set of weights (see section 2.3), and used the classification method from our SGD model to optimize the weights (see section 3.1). The output from our model is in probability form, and so to get it to the correct labels, we simply selected the class with the highest probability for each input. We then calculated the accuracy score by dividing the amount of correct labels with the total amount of labels. We tweaked the hyperparameters using random search in order to get the best result, and compared it with the result from the neural network.

4. RESULTS

We will now present the results. See section 5 for further discussion of these results.

4.1. *SGD hyperparameters on Franke dataset*

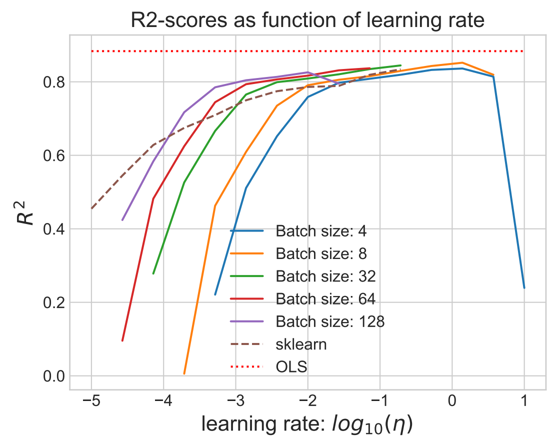


FIG. 3.— Regression on Franke dataset with $n_{datapoints} = 10000$, $\sigma_\epsilon = 0.1$, polynomial degree = 6 using our SGD method of minimizing the loss. Figure shows R^2 score on test data as function of learning rate for varying batch sizes. The model uses a constant learning schedule, $n_{epochs} = 500$, cost function: MSE

In figure 3 we see that lower batch sizes allow higher learning rates, and higher batch sizes allow lower learning rates. All batch sizes have similar maximum R^2 scores, but batch size 8 with a learning rate around $\eta \sim 10^0$ yields the highest R^2 score. None of the batch sizes yield reasonable R^2 scores for learning rates lower than $\eta \sim 10^{-5}$, whereas sklearn's method yields low but reasonable R^2 scores for $\eta \sim 10^{-5}$. In this figure, we have excluded all values that cause extreme underfitting or overflow.

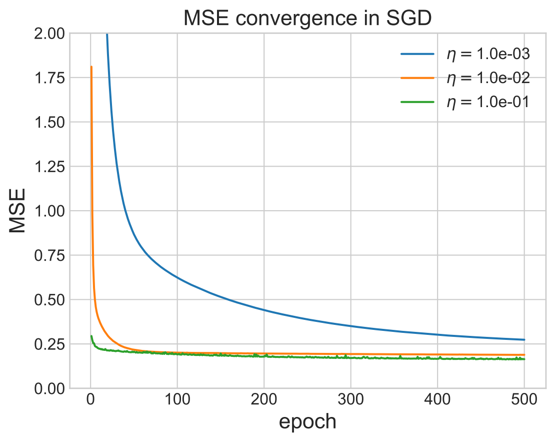


FIG. 4.— Regression on Franke dataset with $n_{datapoints} = 10000$, $\sigma_\epsilon = 0.1$, polynomial degree = 6 using our SGD method of minimizing the loss. Figure shows MSE as function of epochs. The model uses a constant learning schedule. Batch size = 32, $\gamma = 0$, $\lambda = 0$ cost function: MSE

We see in figure 4 that, for this particular model, lower learning rates converge slower than the higher ones. With a learning rate of $\eta \sim 10^{-3}$ the model does not converge with 500 epochs.

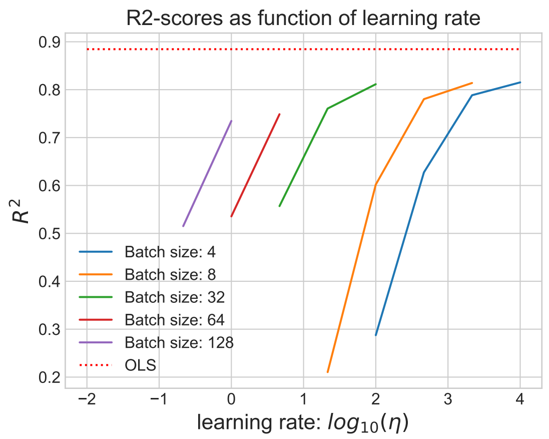


FIG. 5.— Regression on Franke dataset with $n_{datapoints} = 10000$, $\sigma_\epsilon = 0.1$, polynomial degree $= 6$ using our SGD method of minimizing the loss. Figure shows R^2 score on test data as function of learning rate for varying penalties. The model uses an adaptive learning schedule as described in section 2.1.2, with t_1 set to 1. $n_{epochs} = 500$, cost function: MSE

Figure 5 shows the effect of the adaptive learning rate (equation 10). The first thing to notice is that with the batch sizes 32, 64 and 128 we get overflow very quickly so we see no convergence for these values. For lower batch sizes (4 and 8) we start to see some level of convergence for very large learning rates $\eta > 10^3$. We see that the regions that yield reasonable R^2 scores for each of the batches are pushed more towards higher learning rates.

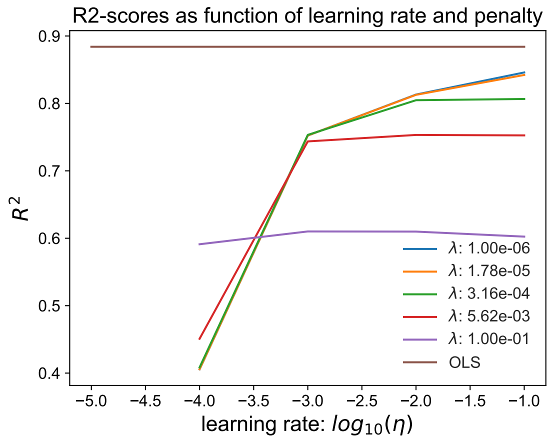


FIG. 6.— Regression on Franke dataset with $n_{datapoints} = 10000$, $\sigma_\epsilon = 0.1$, polynomial degree $= 6$ using our SGD method of minimizing the loss. Figure shows R^2 score on test data as function of learning rate for varying penalties. The model uses a constant learning schedule, $n_{epochs} = 500$, batch size: 32, cost function: MSE

Figure 6 we see that for learning rates higher than $\eta \sim 10^{-3}$, the penalty converges towards lower R^2 scores faster than the non-penalized model. For learning rates lower than $\eta \sim 10^{-3}$, we see that the penalties $\lambda \sim 10^{-1}$ and $\lambda \sim 10^{-3}$ increase the R^2 score of the model.

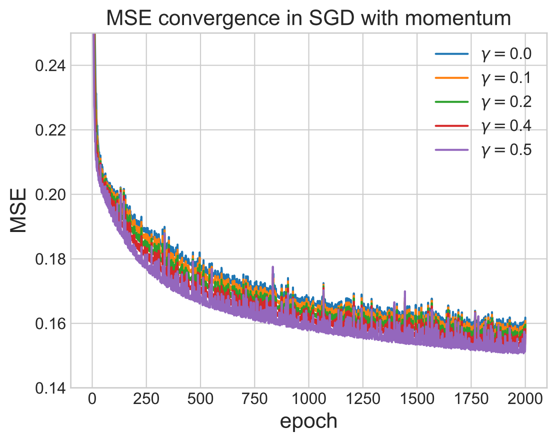


FIG. 7.— Regression on Franke dataset with $n_{datapoints} = 10000$, $\sigma_\epsilon = 0.1$, polynomial degree $= 6$ using our SGD method of minimizing the loss. Figure shows MSE on test data as function of learning rate for varying rates of momentum γ. The model uses a constant learning schedule, $\eta = 0.05$, batch size: 32, cost function: MSE

In figure 7 we see how momentum affects the convergence of a model for the selected learning rate and batch size. We can see that increasing the momentum parameter γ lowers the MSE faster than for the original model. We also see noisier convergence for higher γ. In this particular figure, we chose to exclude $1 > \gamma > 0.5$ as these values caused extreme noise that made the figure unreadable.

TABLE 1
RANDOM SEARCH R^2 SCORES OF SGD ON FRANKE

–	R^2 score	η	γ	λ	Batch Size
OLS	~ 0.89	—	—	—	—
Run 1	~ 0.83	10^{-2}	0.55	$6.0 \cdot 10^{-4}$	64
Run 2	~ 0.82	10^{0}	0.11	$1.0 \cdot 10^{-5}$	4
Run 3	~ 0.79	10^{-1}	0.11	$2.2 \cdot 10^{-4}$	16
Run 4	~ 0.81	$2.2 \cdot 10^{-2}$	0.00	$1.1 \cdot 10^{-5}$	32

Table 4.1 shows the resulting R^2 scores for 4 runs of random search using our SGD regression method for a polynomial fit of degree 6 on the Franke dataset with $n = 1000$ datapoints and $\sigma_\epsilon = 0.1$. Each random search ran through 100 iterations with randomly selected hyperparameter values using a constant learning schedule and 500 epochs. The search selected values from the following ranges:

$$[H]\text{Batch sizes} \in [4,\ 16,\ 32,\ 64]\ \eta \in \left[10^{-2}, 10^{-1}\right]\ \gamma \in [0, 1\rangle\ \lambda \in \left[10^{-5}, 10^{-1}\right]$$

We see that none of the runs managed to approach the OLS solution very well (within a margin of 0.06). The best run used a small learning rate and a high batch size, with momentum and low penalty. The second best run used a relatively large learning rate with a small batch size, again with low penalty and some momentum.

- 4.2. *Neural network*

Score		
Method	R^2 score	Accuracy
NN	0.739± 0.019	0.952 ± 0.002
Sklearn	0.836± 0.011	0.956± 0.012
OLS	0.862 ± 0.00	-

TABLE 2
DONE WITH 1 HIDDEN LAYER, 100 HIDDEN NODES, BATCH SIZE = 32, 300 EPOCHS AND LEAKY-RELU FOR 5 RUNS FOR REGRESSION WE HAVE: 400 DATAPOINTS $\lambda = 0.005$, $\eta = 0.008$, AND FOR CLASSIFICATION: 1797 DATAPOINTS $\lambda = 0.036$, $\eta = 0.013$. THE VALUES USED ARE THE ONES WE FOUND TO BE GOOD FOR OUR NETWORK, NOT NECESSARILY THE BEST VALUES FOR SCI-KIT LEARN'S METHOD

In 2 we can see that sci-kit learn is generally better than our own method, which comes in last. We can see that OLS is the best option for linear regression.

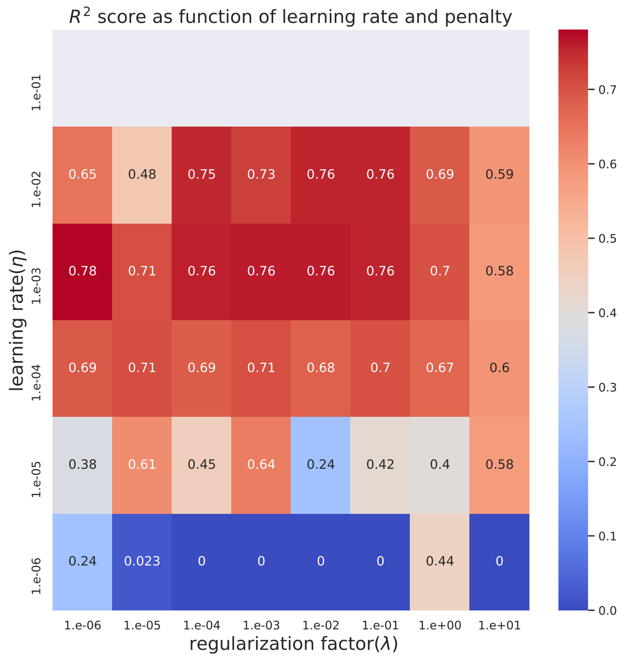


FIG. 8.— dataset: frankedata, 1500 datapoints, with 1 hidden layer , 100 hidden nodes, leaky-reLU, batch size = 32, epoch = 300. all R2 values below 0 has been set to 0. the empty squares are overflow

We see from figure 8 that the R2 value steadily increases with learning rate until it results in overflow. We can see that we get underfitting when the regularization parameter increases. We can also see that the values are not smooth, but fluctuates.

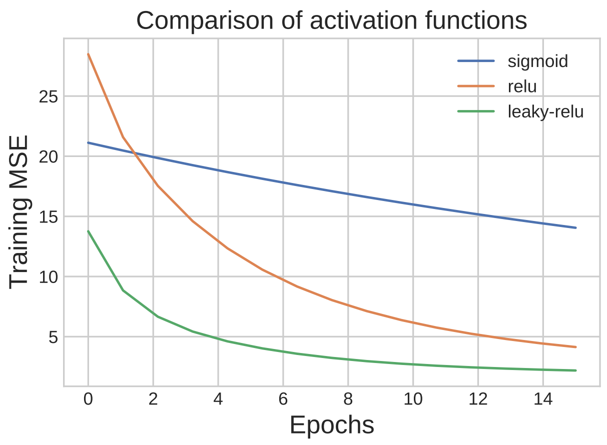


FIG. 9.— dataset: frankedata, with 4 hidden layer , 10 hidden nodes, learning rate = 0.0008, penalty = 0.005, batch size = 32

looking at figure 9 we can see that that the Sigmoid activation function converges noticeably slower than the reLU and the leaky reLU functions. Both the reLU functions seem to be converging at about the same pace, we know that the initial values of the weights are based on a random normally distributed numbers so the intercept value is random.

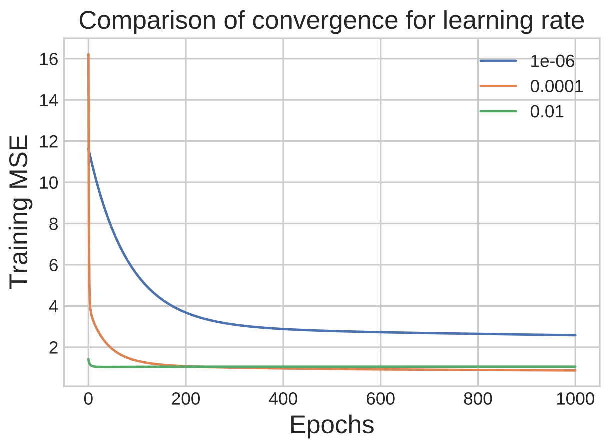


FIG. 10.— Convergence to different MSE values for different learning rates. dataset: frankedata, 1 hidden layer, 100 hidden nodes, batch size = 32, $\lambda = 0.05$ with leaky-reLU

Looking at figure 10 we can see that the the model converges for different values for learning rate = 0.01 and 0.0001 compared to 1e-06.

4.3. *Logistic regression*

TABLE 3
Random search accuracy scores for multinomial regression on the mnist dataset

–	Accuracy	η	γ	λ	Batch Size
SKlearn	~ 0.950	–	–	–	–
Run 1	~ 0.969	$4.6 \cdot 10^{-1}$	0.44	$7.7 \cdot 10^{-5}$	16
Run 2	~ 0.967	$2.2 \cdot 10^{-1}$	0.44	$1.3 \cdot 10^{-1}$	32
Run 3	~ 0.969	4.6	0.66	$2.8 \cdot 10^{-5}$	4
Run 4	~ 0.963	4.6	0.00	$2.2 \cdot 10^{-4}$	4

Table 4.3 shows the resulting accuracy scores for 4 runs of random search using our multinomial regression method on the MNIST dataset. Each random search ran through 100 iterations with randomly selected hyperparameter values using a constant learning schedule and 500 epochs. The search selected values from the following ranges:

- Batch sizes $\in$ [4, 16, 32, 64, 128]
- $\eta \in \left[10^{-2}, 10^{-1}\right]$
- $\gamma \in [0, 1\rangle$
- $\lambda \in \left[10^{-5}, 10^{-1}\right]$

We can see once again that the model favors low learning rates with large batch sizes, or high learning rates with small batch sizes. The top 3 runs used a momentum of around 0.55 ± 0.11. We see that all our model predictions exceed the accuracy score of sci-kit Learn's model.

5. DISCUSSION

5.1. *Hyper-parameter tuning for SGD on Franke data*

We start off by looking at the shape of our cost function. Since all learning rates seemingly cause convergence towards the same score (fig 4), we can expect the cost function to be convex, meaning we have one clear minimum. If the cost function was not convex, we would see a more sporadic distribution of R^2 scores in figure 3, which is clearly not the case. So judging from these two results, we can be pretty sure to say that our cost function is indeed convex. This means that the problem of getting stuck in local minima is not present in this particular case where we fit a line to the Franke data. So any underfitted model, caused by a low learning rate, simply needs more iterations to reach the minimum. The problem with this is that it is slow. So let's look at how the various hyper-parameters could help us avoid this problem.

If we look at figure 3 and table 4.1 it becomes evident that we can achieve a high R^2 score by using larger batch sizes with small learning rates or small batch sizes with high learning rates. This effectively means that having a high learning rate with a small batch size, and a low learning rate with a large batch size is equivalent, as confirmed by Smith et. al (2018) (8). What we can tell from this is that a large batch size will speed up the convergence of our model. With the way we set up our algorithm (section 3.1), we will significantly reduce the number of iterations by increasing the batch size. This means that we can use a large batch size with a small learning rate, and effectively achieve the same result as for a small batch size and large learning rate in a fraction of the time.

With the addition of momentum (figure 7) we could speed up this convergence even more. However, add too much momentum and our model will quickly diverge. Judging from table 4.1, the sweet spot seems to be around $\gamma \sim 0.5$. The penalty seems to only cause a negative effect on our model. We believe this is due to our model not being very overfit, and we would get a similar result as for Ridge on the terrain data in the previous project (10). The Franke dataset in the previous project started having trouble with overfitting around polynomial degree 6, which is why we chose to study this degree in this article. But in the previous one, we used 400 datapoints, which quickly causes overflow for our SGD method. We have thus used 10000 datapoints in this one, which means our results will more closely resemble those found for the terrain data.

The chosen adaptive learning rate (eq. 10) did not perform very well for our model. What happened was that the learning rates that came close to converging suddenly diverged, which is not what we would expect, as the learning rate decreases with time, keeping us in the minimum. We can see that this is mainly the case for larger batch sizes. With a larger batch size, we reduce the total amount of batches we work on, meaning the learning rate will not decay as fast. So that would explain why we see larger batch sizes diverge faster than smaller ones. However, we would not expect to see them diverge faster than the constant case. A reason for this could be that since we decay the learning rate, we need higher initial learning rates, and since the larger batch sizes generally perform better with lower learning rates, we could see divergence more quickly. This is, however, just a speculation, and we will not draw any conclusions from this result.

5.2. *Feed Forward Neural Network*

5.2.1. *Hyper-parameter tuning*

When trying different configurations of hyper-parameters for the simplest form of random initialization we could see that we quickly got overflow with reLU, and Leaky-reLU activation functions and it was almost impossible to create working configurations with more than 1 hidden layer. This got better when we used He or standard initialization method. This is just as we would expect from the theory discussed in 2.2.4

The number of nodes in the hidden layers and the number of hidden layers both add to the complexity of the neural network meaning that we can fit more complex models. This is one of the reasons neural networks are so useful, it can fit very complex models, but the more complex the model is the more data is required to train the model. Since we have a quite simple model increasing the number of layers results in overfitting. We can see that tuning up the penalty reduces this overfitting and improving our model again. Choosing the wrong learning rate can make it so that we end up in a local minimum with the chance of not getting a very good score.

Looking at figure 9 we can clearly see that the Sigmoid activation function is a lot slower than the reLU and the leaky-reLU. This is because of the vanishing gradient problem. The deeper the neural network is the more of a problem this becomes. As discussed in the theory 2.2.3, in deep neural networks the learning is slowed down by the low value for the gradients for the weights. For our models 1 hidden layer turned out to give the best vales, and since we only have 1 hidden layer, vanishing gradients would not be much of a problem. In this example the reLU and the leaky reLU seem to fare quite similarly. We would expect the leaky-reLU to be better in cases the dying reLU is more of a problem

Fitting a model with a sufficient number of epochs is very computationally demanding and it takes a lot of time to go through a random search to find the best parameters.

One reason we do not get the best fits with our neural network is because it is difficult to find the optimal hyper-parameters and there are a lot of them. this is good in one regard, as it makes it possible to fit almost anything but finding that set of hyper-parameters proved difficult.

5.2.2. *Comparison with OLS*

As a stark contrast to linear regression, neural networks require a lot of data to perform optimally. This may be a reason why we do not get very good results with neural network compared OLS for 400 datapoints. Even though, with Frankedata, we were trying to fit a simple model, the best model we got was with 100 layers, and that is a considerably larger number of weights that need to be fit, compared with β values in OLS. So for simple models where you do not have a lot of data, OLS would be the better option. Looking at sci-kit learns neural network implementation we can see that it gets considerably higher R^2 scores than our own network, even with many of the same hyper-parameters and finding values for the hyper-parameters that made the sci-kit Learn model perform better than OLS was not very time consuming, if we bumped the datapoints up to 1000.

5.2.3. *Comparison with the SGD*

We can see from tables 4.1 and 2 that our neural network fails to improve upon the results obtained from the OLS and the SGD methods. Note that for 0 hidden layers, the NN should behave equally to the SGD method, as we are in both cases optimizing the same set of weights (but initialized differently) using the same method, namely stochastic gradient descent. So for 0 hidden layers, we would expect similar values. With the increase of hidden layers, we add complexity to our model, making it more flexible and in return allowing us to further optimize it. The problem is that we have more parameters in our NN, so finding the optimal parameters is more challenging. We have also not added momentum to our NN, but we see in table 4.1 that, even for no momentum, we still get a better score for the SGD method.

We can see from figure 10 that the cost function is not strictly convex for our neural network, meaning that it does not have one single global minimum. This means that tweaking the learning rate is not as trivial as it was for the SGD method. Here, finding the optimal learning rate means finding a learning rate which can jump over local minima and make its way to the global minimum. So there is a set of learning rates that will cause lower R^2 values no matter how many iterations we have. We believe this may be the reason we fail to see improved R^2 scores for the neural network in relation to the SGD method, as we have not found the set of learning rates that make it over all the local minima.

5.2.4. *Comparison with Ridge*

Compare learning rates and penalty From figure 8 we see that the penalty will help our model for larger learning rates. We can already tell that this is very similar to the Ridge plot seen in figure 5 of the previous project (10), where, for increased complexity, the learning rate reduced overfitting. For lower learning rates, our model will get stuck in local minima, which can serve as a penalty in itself. Whereas for higher learning rates, our model will fit the training data better, causing overfitting. So we can see that the global minimum is not necessarily the optimal minimum. Therefore, the penalty can help if the learning rate gets too large. The reason we see overfitting here, and not for the SGD case, is that here we use 1500 datapoints for the Franke data, and we used 10000 for the SGD case. It is easier to overfit to fewer datapoints for lower complexities (as we saw for terrain and Franke in the previous project).

5.2.5. *Comparison with multinomial regression*

Looking at table 4.3 we see once again that our neural network fails to improve upon the results obtained from the multinomial regression method. However, we do get similar values. We especially get close to SciKit's method. As with the SGD, we expect to see the models behave equally if the network has no hidden layers. But, again, we cannot test this, only speculate. As before we believe this failure of improvement comes from the fact that it is a lot harder to tweak the neural network, as it is more sensitive to the hyperparameters. We will also note that the network only iterated over 300 epochs, which could be a reason for the lower score. As for the parameters, the network seems to favor similar values as the multinomial regression method.

5.2.6. *Further discussions*

In our analysis of the learning rate and penalty, we chose to study only the R^2 scores. We could have studied the MSE here as well, but since R^2 is proportional to 1 - MSE, we don't feel an analysis of the MSE would add anything. We only used R2 score, but it works like 1 - MSE for Frankedata from prokect 1 (10).

6. CONCLUSION

Larger batch sizes with smaller learning rates achieve the same results as small batch sizes with high learning rates, but much faster. Momentum will speed up convergence, as expected. For the SGD method, the penalty had no positive effect, due to lack of overfitting. The chosen adaptive learning rate allows larger learning rates, but quickly causes divergence for large batch sizes. For the SGD case, the cost function is convex, meaning we need only increase the iterations to reach the minimum for low learning rates.

The neural network did not perform as well as the SGD and the multinomial regression methods. This is due to the limited data as well as the increased complexity of the neural network, making it harder to optimize. The cost function for our neural network is not convex, meaning it is more difficult to find optimal learning rates. We believe the neural network should behave like the SGD and the multinomial regression methods without hidden layers, but we were not able to test this. The penalty had a similar effect with the FFNN as it did with the Ridge.

The Sigmoid function converges slower than reLU and leaky reLU. We got overflow when initializing the weights randomly, but this problem was solved with standard for sigmoid, for reLU leaky reLU we use He initializing. The effect is that the standard deviation between the nodes sticks to 1.

Neural networks have the ability to fit very complex models because of the large amount of hyper-parameters, but this also makes it more difficult to find the correct model for your data. Fitting complex models also require a lot of data, which is not always trivial to obtain.

If we could make any improvements we would like to implement early stopping, greatly reducing the computation time. Another thing that could be interesting to take a look at is sparse neural networks, and see if we could reduce the computation time and still get the same score. Since neural networks are so data hungry, and in the real world we cannot, on a whim, create more data like we can with Franke data we also think it would be beneficial to look at ways to artificially inflate our dataset, for instance, by rotating the pictures a little, as a form of re-sampling. It would also be nice to see if different activation functions could help improve our network by making it more stable, but a lot of them are more computationally demanding than reLU. We would also be interested in looking at different SGD variants such as RMSprop and ADAgrad to increase the time it takes to converge. Another thing to note is that Since we are not splitting into train test and validation sets, but just settling for training and test, we might be tweaking parameters to give good values on test data by not using validation data to compare with.

Github with source code:
https://github.com/RivaIngunn/classification_and_regression

REFERENCES

[1]Artificial neural network. https://en.wikipedia.org/wiki/Artificial$_neural_network$, 2020.

[2]EunBeenKim alexlenail, sof. Nn-svg. https://alexlenail.me/NN-SVG/index.html.

[3]Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[4]Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[5]Morten Hjorth-Jensen. Week 39 optimization and gradient methods. https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html, 2020.

[6]Morten Hjorth-Jensen. Week 41 tensor flow and deep learning, convolutional neural networks. https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html, 2020.

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

[8]Sam Smith, Pieter jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size. 2018.

[9]Charis sng. An introduction to gradient descent. https://medium.com/ai-in-plain-english/an-introduction-to-gradient-descent-4c04b4c063bd.

[10]Ingunn Rivå Ørjan Strand. A review of various linear regression methods and resampling techniques. 2020.