A Comparison of Neural Networks to Linear & Logistic Regression for Regression and Classification Problems

Martin Moen Carstensen, George Stanley Cowie, Jan Egil Ødegård
(Dated: November 13th, 2020)

In this project we have taken a closer look at applications of neural networks in both linear regression and classification problems. For the linear regression method, we've attempted to fit our model to the Franke function, which is a weighted sum of exponentials. For the classification case, we've trained our models to recognize hand written digits in the MNIST data set. We also made an attempt at comparing neural networks to other common linear and logistic regression methods which tend to use gradient methods. Our results tend to show that SGD methods are inferior to matrix inversion methods, both in terms of efficiency and accuracy, being several orders of magnitude worse in both categories. This could however be prone to optimization through gradient boosting or the use of more complex SGD models. Compared to the neural networks, the SGD method comes short, while they are comparable in accuracy to the OLS method. Potentially, the neural networks could prove to be more optimal than OLS, but the amount of free parameters to choose among are computationally overwhelming. Likewise, for the classification cases, we get similar accuracy results using the neural network approach to the logistic regression approach, with the neural networks potentially being better for an optimal choice of parameters. We also suffer from the curse of many free parameters in this case however, being a minor inconvenience to this approach. One important take-home from our findings is the fact that an optimal learning rate is always the most important parameter to fit, regardless of approach and the problem at hand.

I. INTRODUCTION

For complex problems in both linear regression and classification, one can not simply expect that a true value can be approximated by a simple fit to the data. Rather, the opposite often tends to be the case. Complex and high-dimensional approximations to data is often times prone to overfitting, such that we are threading a fine line between accuracy and model choice. This is where neural networks come into play, being a non-linear way to approximate data, it could prove useful in accurately modelling complex data sets.

In this report, we present a comparison of neural networks to linear and logistic regression methods for function evaluation and classification. First, we will compare a matrix inversion approach to our gradient methods using both the Ordinary Least Squares and Ridge regression methods. This is followed by a comparison between the gradient methods to a neural network. Some interesting metrics will be both computational time spent training the models, as well as the accuracy of the models. Ideally, We would also like to test different activation functions within the neural networks to see which one work the best. For the linear regression, we will attempt to fit the Franke function, which is a weighted sum of exponentials. We then move over to classification problems, where we will compare neural networks to a logistic regression method. We are also interested in the comparison both in terms of accuracy and computational time here. For the classification case, we will use the MNIST data set, which is a data set of hand written digits, in which we will attempt to classify which pictures correspond to which digits by model training.

CONTENTS

II. THEORY

The theory section is split into the cases for linear regression, where we attempt to fit our data to a continuous function, and for classification, where we attempt to identify which 'box' we would classify our data.

A. Linear Regression analysis

Here we will look closely at the Stochastic Gradient Descent method, and Neural Networks. For other linear regression methods also mentioned, for instance Ordinary Least Squares or Ridge, see our previous project report[1].

1. *Stochastic Gradient Descent*

When performing linear regression, we are often led to problems in which we are required to invert a matrix. Examples for Ordinary Least Squares (OLS) and Ridge respectively are[2][3]:

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

$$\hat{\beta}_{Ridge} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbb{1})^{-1}\mathbf{X}^T\mathbf{y}. \tag{2}$$

Here, $\mathbf{X}$ is our design matrix, $\mathbf{y}$ is the true value for our training data, λ is a free hyperparameter, and $\hat{\beta}$ are the optimal parameter values to fit our model using said method. As is evident in equations 1 & 2, we need to invert a matrix, and this matrix is typically close to being singular. As such, when inverting a close-to-singular matrix numerically, we could stumble into numerically unstable results [4].

One of the methods to avoid doing the matrix inversion all-together is by jumping back a step when deriving the above relationships. Looking at the cost functions for OLS and Ridge respectively, we have:

$$C(\beta)_{OLS} = \frac{1}{n}\Big[(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)\Big] \tag{3}$$

$$C(\beta)_{Ridge} = \frac{1}{n}\Big[(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)\Big] + \lambda\beta^T\beta \tag{4}$$

When deriving the optimal parameters $\hat{\beta}$ for these methods, we take the gradient of equations 3 & 4, and set this to 0 to find the optimal parameters. (This essentially yields the minimal value of the cost function when fitting the data points). Doing this gives us (Derivations done by Hjorth-Jensen [5]-[6]):

$$\nabla_\beta C(\beta)_{OLS} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) \tag{5}$$

$$\nabla_\beta C(\beta)_{Ridge} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) + 2\lambda\beta \tag{6}$$

We want to find the gradient in equations 5 & 6 numerically in an attempt to find the global minimum of the respective cost functions. One of the standard ways of doing this revolves around calculating the gradient of the function in the following manner:

$$\nabla_\beta C(\beta)_{GD} = \sum_{i=0}^{n-1} \nabla_\beta c_i(\mathbf{X}_i, \beta). \tag{7}$$

Here, we sum a gradient calculation over all data points to numerically approximate the correct gradient. This will yield the steepest descent towards the minimum value, however, this will also become computationally expensive with large data sets. The need to calculate this for many parameters β will also add to the computational load.

To remedy this we will introduce some stochasticity. By randomly selecting a mini-batch, which is a subset of the data, and calculating the gradient on said subset, the equation will now read:

$$\nabla_\beta C(\beta)_{SGD} = \sum_{i \in B_k} c_i(\mathbf{X}_i, \beta). \tag{8}$$

Here, B_k is the mini-batch we calculate the gradients over, replacing the full data set. Finding the updated parameters β is done in the following manner:

$$\beta^{(i+1)} = \beta^{(i)} - \gamma \cdot \nabla_\beta C(\beta). \tag{9}$$

Were the parameter γ is defined as the learning rate. This method of adding stochasticity greatly increases the computational speed when finding the gradient. However, due to the stochastic nature of the method, the gradient in parameter space will not always be the ideal one, and as such, we will need this method to run through more steps to find the optimal parameters.

A lingo side-note: If our mini-batch consists of 2 elements, this is known as a stochastic gradient descent, while if the mini-batch size is larger than 2, it is known as a mini-batch gradient descent. As such, a stochastic gradient descent is only a special case of the mini-batch gradient descent.

Using the methods stated in equation 9, together with the gradient calculations in either equations 7 or 8, we get a method that should converge, given that our learning rate γ is small enough. A large γ will induce a divergence in our parameters β, and is as such not ideal (Chapter 4. [7]). A constant learning rate will also make our solution converge up to a point, in which stochasticity takes over and the solution will 'bounce around the optimal parameter values in parameter space' (see figure 1).

A way to solve this is by introducing a process named simulated annealing (Chapter 4. [7]). This

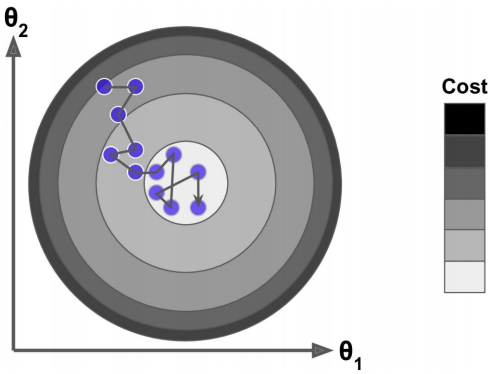

Figure 1. Illustration of the path when attempting to find the optimal parameters in parameter space. (Fig. 4-9 in Géron [7])

method introduces a 'time-dependency' on the learning rate, such that for each subsequent run, the learning rate decreases. The form of the learning rate γ_j is then given by a function known as the learning schedule [8]:

$$\gamma_j(t;t_0,t_1) = \frac{t_0}{t+t_1}. \tag{10}$$

Here, $t_0, t_1 > 0$ are free parameters that we pick, and $t = e \cdot m + j$ is our 'time dependency', with m denoting the number of mini-batches and e denoting which epoch we're currently running through ($e = 0, 1, 2, \cdots$). j is then the iteration number when iterating over the mini-batches ($j = 0, 1, \cdots, m-1$).

2. *Neural Networks & Linear Regression*

Functions also be parametrized by using neural networks. A neural network is a system which is heavily inspired from biology and neuroscience, and consists of a pre-determined design of nodes and links connecting said nodes. It is usually split into 3 regions, namely the input layer, the output layer and a series of hidden layers. A general illustration of such a network is seen in Figure 2.

To understand these networks better, we are better off looking at an individual node, and how they function. In a mathematical sense, each individual node performs the following calculation:

$$\text{Output} = f\left(\sum_{i=1}^{N} x_i w_i + b_j\right) = f(u) \tag{11}$$

Here, x_i are the inputs received from the i'th node in the previous layer, and w_i is the corresponding weight attributed to the given input, and b_j is the bias corresponding to our node. In the case that the previous layer consists of N different nodes, the sum then stretches over the entirety of the previous layer. In this case. The function $f(\cdot)$ is what's known as an activation function, of wich there are several to chose from:

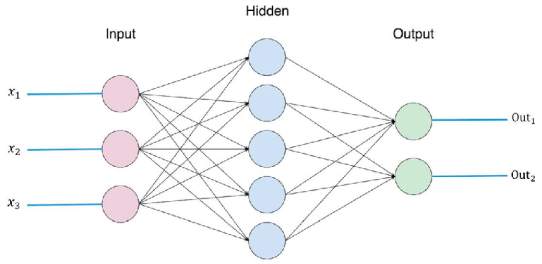

Figure 2. An illustration of the general architecture of a neural network with 3 inputs and 2 outputs. What this figure doesn't show is the fact that we can have an arbitrary number of hidden layers, as well as an arbitrary number of nodes within each hidden layer. These specifications are subject to design for any given problem. (Figure 2 in Raissi *et al.* [9])

- The Sigmoid: $f(u) = \sigma(u)$ (stated in equation 16)
- The Tanh-function: $f(u) = \tanh(u)$
- The RELU-function: $f(u) = \text{Max}(0, u)$
- The Leaky-RELU-function: $f(u) = \text{Max}(0.01 \cdot u, u)$

The output from this node is then distributed to the nodes in the next layer which they are connected to. For an arbitrary number of hidden layers and an arbitrary number of nodes per hidden layer, this creates a non-linear link between a set of input values and a set of output values of the network as a whole.

We are free to determine the weights w_i for each individual connection, as well as the bias b_j for each individual node, and these parameters are the ones we will be able to train and adapt for our given model. For this, we will use the back-propagation algorithm, in which we will adjust the biases and weights in accordance to the accuracy score for the neural network. For this, we need to first define a cost function $\mathcal{C}$. This will then be used to calculate the output error:

$$\delta_j^N = f'(z_j^N)\frac{\partial \mathcal{C}}{\partial (a_j^N)} \tag{12}$$

Here, $f'(z_j^L)$ is the function value of the derivative of the activation function and the expression containing $\mathcal{C}$ is the gradient of the cost function. The superscript N indicates that this is the N'th layer (the output layer), and the subscript j indicate that this is the j'th node in said layer. Furthermore, we want to calculate the back-propagation error corresponding to the individual hidden layers as:

$$\delta_j^n = \sum_k \delta_k^{n+1} w_{kj}^{n+1} f'(z_j^n) \tag{13}$$

Here, n runs over $n \in \{N-1, N-2, \cdots, 2\}$ with these being all of our hidden layers (not including the input layer). For each individual node with subscript j, we sum

over the errors from the nodes in the $l+1$'th layer, multiplied with the corresponding weights and the derivative of the activation function. This creates an iterative scheme where we can calculate all the δ_j^n's using only out output layer. We then use these new values to update our weights and biases in the following manner:

$$w_{jk}^n \leftarrow w_{jk}^n - \eta \delta_j^n a_k^{n-1} \tag{14}$$

$$b_j^n \leftarrow b_j^n - \eta \delta_j^n \tag{15}$$

Here, the variable η will be our learning rate, taken to be constant. As for the cost function, this is chosen to fit the problem we are trying to optimize. We will typically pick the mean squared error for this.

For a linear regression, our output layer will only consist of a single node, namely the function value. The input layer will consist of the function values $\{x_1, x_2, \cdots, x_m\}$ to our function $f(x_1, x_2, \cdots, x_m)$.

B. Classification Problems

1. *Logistic Regression*

In logistic regression, instead of optimizing for a given polynomial (as with the linear regression models), we are looking to adapt our function to a sigmoid function stated as follows[10]:

$$\sigma(t) = \frac{e^t}{1+e^t} = \frac{1}{1+e^{-t}}. \tag{16}$$

Here σ now gives us outputs in the range $\sigma \in [0,1]$. Furthermore, we can define the variable t to be represented as a linear combination of independent variables in the following way:

$$t = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n. \tag{17}$$

Here, x_i are freely defined from the problem we're looking at (for example the color value of a given pixel for image analysis), and θ_i are free parameters which we adjust using our training data.

Considering that the sigmoid function σ spans the values between 0 and 1, it is ideal to classify a binary problem. (Is the given picture a cat or a dog? Is the cancer tumor a benign tumor or not? etc..). Is the value close to 1, we have strong belief that the statement we're testing is true, and if it's close to 0, the opposite is the case. However, it is not ideal when considering problems in which you want to classify a set of more than 2 classes.

Instead, one can use multinomial logistic regression, or softmax regression [11] to classify a problem where we have more than 2 possible outcomes. This replaces the logistic function σ in equation 16 with the softmax function Σ as:

$$\Sigma(t_i) = \frac{e^{t_i}}{\sum_{j=1}^{k} e^{t_j}} \tag{18}$$

Where the sum running over j is the sum over all possible classes K, and $\Sigma(t_i)$ indicate the probability of class i to be true as compared with the other classes. We have that t_i is written as follows:

$$t_i = \theta_{i,0} + \theta_{i,1} x_1 + \theta_{i,2} x_2 + \cdots + \theta_{i,n} x_n \tag{19}$$

In the special case for softmax where $K = 2$, it is reduced to the binary case discussed above [11], and is as such a generalization for K number of classes. We then want to optimize the parameters θ_i for each of the classes individually by using our training data. The cost function we want to optimize is on the following form:

$$J(\theta) = -\Big[\sum_{i=0}^{m-1} \sum_{k=0}^{K-1} \mathrm{I}\{y_i = k\} \ln \Big\{ \frac{\exp\{\theta_{k,0} + \theta_k^T x_i\}}{\sum_{j=0}^{K-1} \exp\{\theta_{j,0} + \theta_k^T x_i\}} \Big\} \Big] \tag{20}$$

where m is the number of data points and K are the number of classes. To train this, we want to use the SGD algorithm already discussed, and for this, we need to find an expression for the gradient, which is given as:

$$\nabla_{\theta^{(k)}} J(\theta) = -\sum_{i=1}^{m} \Big[x^{(i)} \Big(I\{y^{(i)} = k\} - P(y^{(i)} = k | x^{(i)}; \theta) \Big) \Big] \tag{21}$$

This replaces our gradient in the standard linear regression case. We also need to rewrite the corresponding design matrix. For the MNIST data set, we get a single-channel picture, where each pixel correspond to a value between 0 and 1. Each pixel value can as such be attributed to an array value, directly placed into the column of our design matrix. Our design matrix will be of size $M \in \mathbb{R}^{p \times N}$, where N denotes the number of training data and p denotes the number of pixels for each picture. The individual values corresponding to the 'intensity of light' of any given pixel.

2. *Neural Networks & Classification*

For a classification problem, much of the theoretical structure regarding the networks design (as discussed in section II A 2) remain the same, but with some key differences. These differences come in the form of a changed activation function, specified for a classification problem, as well as some changes to the design of our neural network.

We will primarily look into image recognition of single-channeled images (not three channels as in RGB). Every pixel will thus correspond to a single value

between 0 and 1. For an $n \times m$ picture, we will have $n \cdot m$ number of nodes in our input layer, corresponding to the color value of each individual pixel. For the output, we have k different classes which we will attempt to distinguish our test results into. As such, our output layer will consist of k different nodes, each directly corresponding to 1 of our class values.

The choice in activation functions for the hidden nodes will be the sigmoid function σ (from equation 16),which can output values between 0 and 1. Further, for our output function, we will be using the softmax function, which is stated in equation 18. By using this as our activation function for the output layer, the sum over all classes k will normalize to 1, so that we can justify a probability interpretation directly from the output of the nodes.

3. *Accuracy Score*

When looking at classification problems, we need to define an accuracy score which is suited for a discrete spectrum of outputs. We define this accuracy score as:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=0}^{n-1} I(t_i = y_i). \tag{22}$$

Here, t_i is our predicted outcome and y_i is the true value. the indicator function I gives output 1 if $t_i = y_i$, and 0 otherwise.

III. ALGORITHM

Here we will look at the implementation for the linear regression and classification cases separately.

A. Linear Regression

We will here look at how to build the algorithms for the Stochastic Gradient Descent method and neural network separately.

1. *Stochastic Gradient Descent*

For the Stochastic Gradient Descent case, the general algorithm will be as stated:

Minibatches = # Datapoints / Minibatch size
θ = Randomly_initiated_float_array
for Epochs **do**
  for Minibatches **do**
    k = Randomly_initiated_int $\in [0, n - M]$
    $X_k = X[k : k + M]$
    $y_k = X[k : k + M]$
    if OLS **then**
      ∇ = OLS_Gradient(X_k, y_k, θ, n)
    else if Ridge **then**
      ∇ = Ridge_Gradient$(X_k, y_k, \theta, n, \lambda)$
    end if
    t = epoch * # minibatches + minibatch
    γ =Learning_Schedule(t)
    $\theta = \theta - \gamma * \nabla$
  end for
end for
Return θ

In this algorithm, X is the design matrix and y the corresponding true value. For a more in depth discussion of these methods see our previous project [1]. The Learning Schedule-function is stated in equation 10, while the gradients are stated in equations 5 and 6. This will then calculate the ideal θ by use of stochasticity given the size of the minibatches and the number of epochs.

2. Neural Network

Since neural networks consists of a system of independent nodes and neurons connecting the nodes, this system lends itself to object-orientation, which we will take advantage of.

The general algorithm for training the neural network, given a set number of epochs, number of inputs and the batch sizes, can be simply written as follows:

Iterations = # inputs // Batch size
for Epochs **do**
  for Iterations **do**
    rndindx = Random_Indexes(Batch size; no replace)
    X_data = Full_X_data$_{rndindx}$
    Y_data = Full_Y_data$_{rndindx}$
    Run Function: Feed_forward()
    Run Function: Back_Propagation()
  end for
end for
Run Function: Predict(X)

In the algorithm above, the random_index function will return a set of random indexes with any given batch size. Further, we look closer at the algorithms of the feed-forward and back-propagation algorithms. First the feed-forward:

z_0 = inputs * weight$_1$ + bias$_0$
node_output$_j$ = activation_function(z_0)
for Hidden Layers **do**
  z_j = node_output$_{j-1}$ * weight$_i$ + bias$_j$
  node_output$_j$ = activation_function(z_j)
end for
z_n = node_output$_{n-1}$ * weight$_n$ + bias$_n$
output = activation_function(z_n)

This basically runs through every single layer, calculating the output of every single node within each layer before moving onto the next one. The outputs of each node depend on the activation function given to said node, as well as the results stemming from the previous layer and the intrinsic bias and weights.

Further, we have for the back-propagation algorithm:

∇_a = (output$_i$ - true_value$_i$)
$\delta_j^N = f'(z_j^N) \cdot \nabla_a \mathcal{C}$
$\delta_j^{(N-1)} = f'(z_j^{(N-1)}) \cdot \delta_j^N$
if $\lambda > 0$ **then**
  $\delta_j^{(N,N-1)}$ += $\lambda \cdot w^{(N,N-1)}$
end if
$w^N -= \eta \delta^N \nabla_a^{N-1}$
$b^N -= \eta \delta^{N-1}$
for Hidden layers (backwards loop) **do**
  $\nabla_a = \delta_j^{(n+1)} \cdot w_j^n \cdot f'(z_a^{(n+1)}$
  $\delta_j^n = f'(z_j^{(n)} \cdot \delta_j^{(n+1)}$
  if $\lambda > 0$ **then**
    $\delta_j^n += \lambda \cdot w^{(n+1)}$
  end if
  $w^{(n)} -= \gamma \cdot \delta^n \nabla_a^{(n+1)}$
  $b^{(n)} -= \gamma \cdot \delta^{(n+1)}$
end for
$\nabla_a = \delta_j^2 \cdot w_j^1 \cdot f'(z_a^2)$
$\delta_j^1 = f'(z_j^1) \cdot \delta_j^2$
if $\lambda > 0$ **then**
  $\delta_j^1 += \lambda \cdot w^{(2)}$
end if
$w^0 -= \eta \cdot \delta^0 \cdot \nabla_a^1$
$b^0 -= \eta \cdot \delta^1$

Note that both subscripts and superscripts refer to indexes in this notation. Subscripts refer to the node, and the superscript refer to the layer. Else, what we are in practice doing is changing our weights and biases according to the rules established in equations 12-15, treating the 'edge cases' close to the output and input nodes explicitly, and doing a general sweep over the rest of the hidden layers. This in total should then optimize our neural network, such that we can perform the predict-function on the design matrix corresponding to our test-values X.

3. Fitting function (Franke Function)

The function we will attempt to fit is the same one attempted in the previous report[1], namely the 2-dimensional Franke function, which is stated as:

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left\{-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right\} \\ &+ \frac{3}{4}\exp\left\{-\frac{(9x+1)^2}{49} - \frac{9y+1}{10}\right\} \\ &+ \frac{1}{2}\exp\left\{-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right\} \\ &- \frac{1}{5}\exp\left\{-(9x-4)^2 - (9y-7)^2\right\}. \end{aligned} \tag{23}$$

This equation is defined for $x, y \in [0,1]$, and takes the form seen in figure 3.

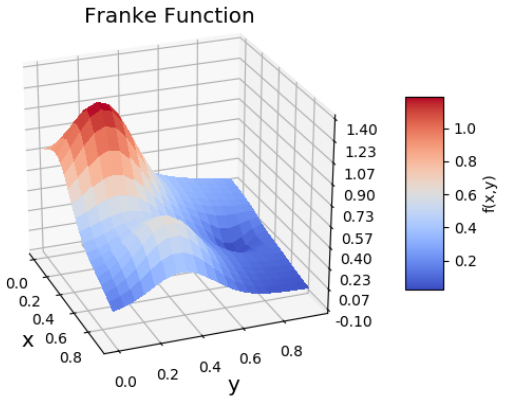

Figure 3. The Franke function visualized in a 3-dimensional plot. Somewhat resembling a mountainous topological map.

B. Classification

Here we look at the algorithmic implementation of the logistic regression using an SGD approach and the neural network cases separately.

1. Logistic Regression

For the logistic regression case, we will perform the SGD approach, in a similar manner to the one discussed for the linear regression case. The key differences lie in the fact that we're working with parameters $\theta \in \mathbb{R}^{m\times k}$ where m is the number of parameters per class and k are the number of classes. (As opposed to $\theta \in \mathbb{R}^m$ for the regular SGD-case).

Minibatches = # Datapoints / Minibatch size
θ = Randomly_initiated_float_array
for Epochs **do**
  for Minibatches **do**
    k = Randomly_initiated_int $\in [0, n-M]$
    $X_k = X[k : k+M]$
    $y_k = X[k : k+M]$
    if OLS **then**
      ∇ = OLS_Gradient(X_k, y_k, θ, n)
    else if Ridge **then**
      ∇ = Ridge_Gradient$(X_k, y_k, \theta, n, \lambda)$
    end if
    t = epoch * # minibatches + minibatch
    γ =Learning_Schedule(t)
    $\theta = \theta - \gamma * \nabla$
  end for
end for
Return θ

2. Neural Network

The general algorithm for this neural network is more or less identical to the algorithm already discussed for the linear regression case (in section III A 2). The differences when applying this to a classification problem however are as follows:

- We will restrict ourselves to only using the sigmoid function as activation function for the hidden layers
- We will use the softmax function as activation function for the output layer
- The input layer will consist of n nodes, where n denoted the number of pixels in the data set we will look at. (For the MNIST data set, this is $8\times 8 = 64$ nodes)
- The output layer will consist of k nodes, where k denotes the number of classes we can expect to classify (For the MNIST data set, this is 10 output nodes)

The output nodes should then individually return the probability for their respective digit to be the one visible on the figure.

3. Classification Data Set (MNIST data set)

For these classification problems, we will attempt to classify the MNIST data set, which is a data set consisting of hand written digits accompanied with labels corresponding to which digit they are. In figure 4, we see some of these hand written numbers[10].

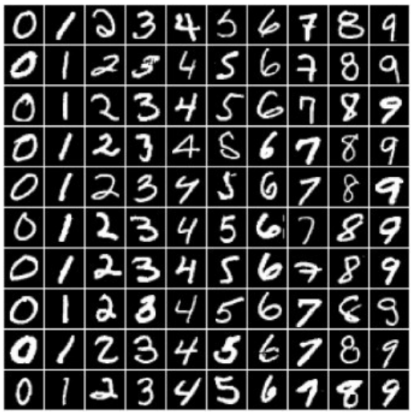

Figure 4. Example of handwritten digits from the MNIST dataset. These numbers come together with labels corresponding to the numbers that they are supposed to look like. (Saha [10])

IV. RESULTS

We will split up the results for the linear regression and classifications, and look at these separately.

A. Linear Regression

1. *SGD methods*

In figure 5, we have OLS for SGD and matrix inversion compared in terms of the complexity of the model and the corresponding MSE score. This was the case for $N = 2000$ data points sampled from the Franke function, with a noise factor of 0.2 added. For the SGD case, we iterated over 500 epochs, with a minibatch size of 2. From this plot, SGD seems to be inferior to the matrix inversion approach in regards to accuracy.

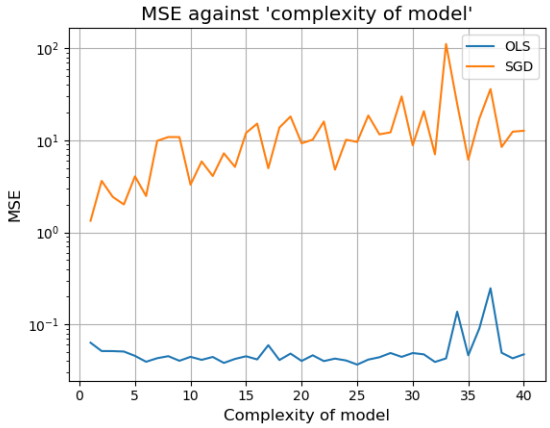

Figure 5. Mean Squared Error plotted against complexity of model for both matrix inversion (OLS) and SGD methods. Here, we've used $N = 2000$ datapoints with a noise factor of 0.2. For the SGD, we've used 500 epochs and a minibatch size of 2.

In figure 6, we have OLS for SGD and matrix inversion compared in terms of time spent computing the algorithms and the corresponding MSE score. The complexity was run over polynomial degrees $n \in [20, 40]$, and the number of epochs was given as $10 \cdot \#terms$. We also iterated over $N = 100$ data points. From this scatter plot, it seems like SGD also comes short in regards to computational time.

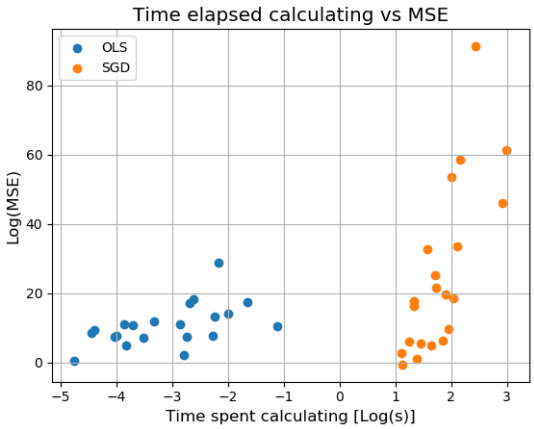

Figure 6. Mean Squared Error scatterplot against CPU time for matrix inversion (OLS) and SGD methods. $N = 100$ data points with a noise factor of 0.2, We've used $10 \cdot \#$ terms number of epochs, with a minibatch size of 2

In figure 7, we have looked at variation of the hyperparameter λ and learning rate γ when performing Ridge regression for SGD methods. From this, it is clear that picking a correct learning rate is much more vital for convergence than a correct hyperparameter, even though varying both come into play. This figure uses $N = 2000$ data points sampled over the Franke function with a factor 0.2 of noise. 200 epochs with a minibatch size of 2 has been used. For the entire plot, we've modelled the function as a degree-15 polynomial, and the optimal learning rate and hyperparameter might change for different choices of parametrization.

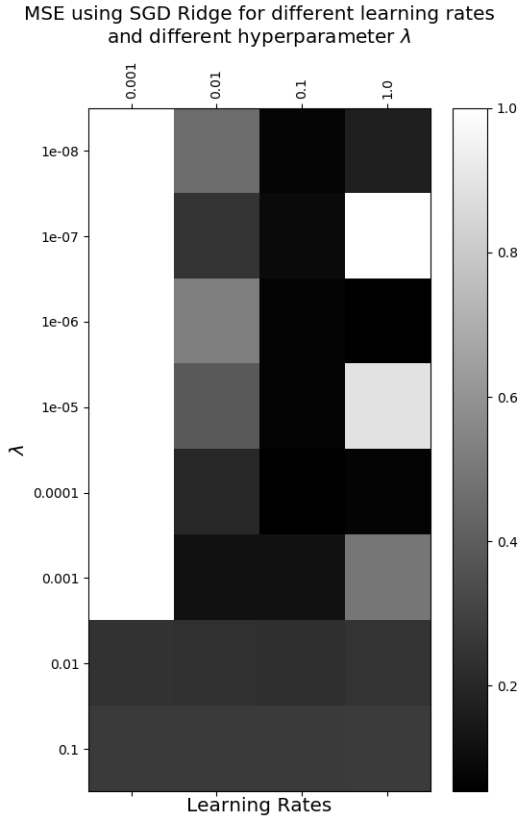

Figure 7. Mean Squared Error for different learning rates γ and ridge hyperparameters λ. This is for $N = 2000$ data points with a noise of 0.2. This is fitted to a polynomial of degree 15, using 200 epochs and a minibatch size of 2.

2. *Neural Networks*

In figure 8, we have plotted the mean squared error of our output as a function of varying learning rate of the back-propagation algorithm. For this figure, we have 125 hidden neurons in each hidden layer, and we have 2 hidden layers. We have iterated through 1000 epochs with a batch size of 50. This has been applied on a data set of 1000 data points corresponding to the Franke function with a noise term of 0.1. For the hyperparameter, we've set this to $\lambda = 0.1$. As the activation function for both the hidden neurons and the output neurons we've used the sigmoid function. From this plot, it is apparent that a correct choice of a learning parameter γ is important to minimize the accuracy of our model.

In figure 9, we have plotted the mean squared error of our output as a function of number of hidden neurons for the hidden layers individually. We've used 2 hidden layers in our model. The learning rate has been set to $\gamma = 0.0001$. We've used 1000 epochs and a batch size of 50. This has been applied on a data set of 1000 data points corresponding to the Franke function, with a noise term of 0.1. For the hyperparameter, this has been set to $\lambda = 0.1$. As activation function for both the hidden neurons and the output neurons, we've used the sigmoid function. From this plot, we see that the dependence on the number of hidden neurons is pretty minor, and somewhat random.

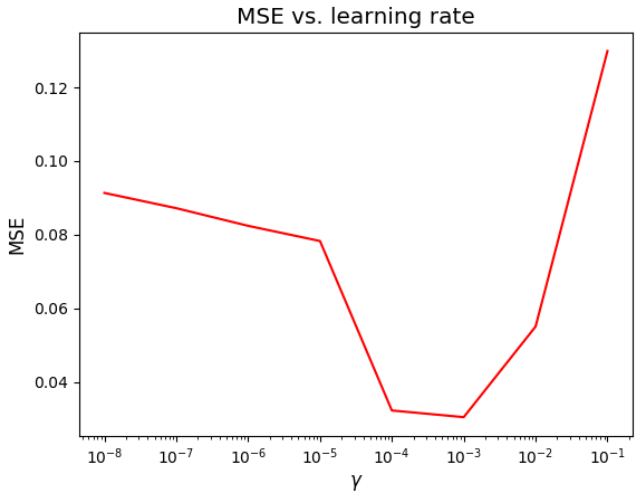

Figure 8. Mean Squared Error plotted against learning rate γ for 125 hidden neurons per layer and 2 hidden layers, 1000 epochs with a batch size of 50, using $\lambda = 0.1$. We've modelled this using 1000 data points with a noise of 0.1. Sigmoid has been used as activation function for all nodes.

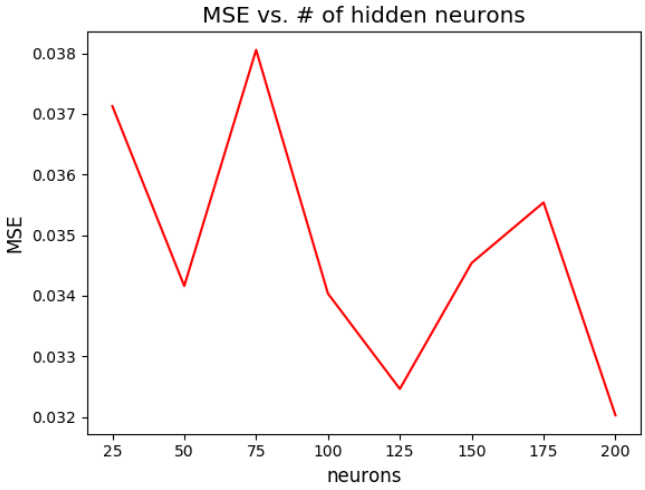

Figure 9. Mean Squared Error plotted against number of hidden nodes, for 2 hidden layers, learning rate $\gamma = 0.0001$, 1000 epochs with a batch size of 50, using $\lambda = 0.1$. We've modelled this using 1000 data points with a noise of 0.1. Sigmoid has been used as activation function for all nodes.

In figure 10, we have plotted the mean squared error of our output as a function of the number of epochs when training the model. We've used 2 hidden layers and 125 hidden neurons per layer. The batch size that has been used is of size 50. The learning rate $\gamma = 0.0001$. This has been applied on a data set of 1000 data points corresponding to the Franke function, with a noise term of 0.1. For the hyperparameter, this has been set to $\lambda = 0.1$. As activation function for all nodes, we've used the sigmoid function. From this plot, it is apparent that more epochs of training will result in a better model. This will however come at the cost of computational time training the model.

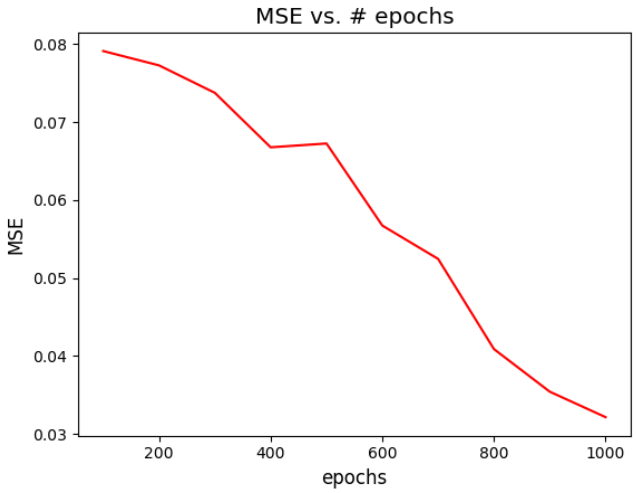

Figure 10. Mean Squared Error plotted against the number of epochs, using 125 hidden neurons per layer on 2 hidden layers. The batch size is 50, and $\lambda = 0.1$. We've modelled this using 1000 data points with a noise of 0.1. Sigmoid has been used as activation function for all nodes.

B. Classification

1. *Neural Networks*

In figure 11, we have plotted the accuracy score vs. choices of learning rate γ and hyperparameter λ for the neural network case. The higher the score, the more accurate the model. For training, we've used 100 epochs with a batch size of 100. 50 hidden neurons per hidden layer, which we have 3 of. The learning rates go logarithmically from $\gamma = 10^{-6}$ at the top to $\gamma = 10^{-1}$ at the bottom, while the parameter λ go logarithmically from $\lambda = 10^{-6}$ on the left to 10^{0} on the right. It is here apparent that the correct choice of learning rate γ is extremely important in determining the accuracy of the model.

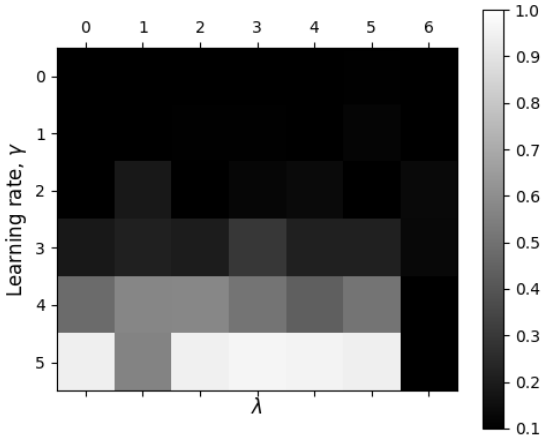

Figure 11. Classification accuracy for differing choices of λ and learning rate γ. We've used 100 Epochs, with a batch size of 100. We got 50 hidden neurons in 3 hidden layers. The learning rate is logarithmically separated in the range $\gamma \in [10^{-6}, 10^{-1}]$, and $\lambda \in [10^{-6}, 10^{0}]$.

In figure 12, we have plotted the confusion matrix of the neural network model, using the same parameters as in figure 11, but with $\gamma = 0.1$ and $\lambda = 0.001$. We see that this model correctly identifies the digits more often than not.

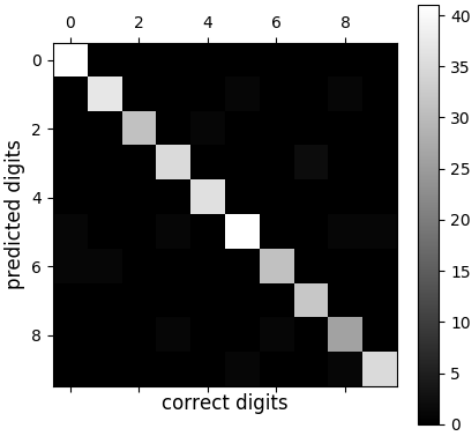

Figure 12. Confusion matrix for the same choice of parameters as in figure 11, but locking in $\gamma = 0.1$ and $\lambda = 0.001$. The color bar indicates the number of 'counts' of each individual case

2. *Logistic Regression*

In figure 13, we have plotted the accuracy score for predictions as a function of learning rate γ, using 1000 epochs to train our model. It is here apparent that choosing the optimal learning rate is of upmost importance when training the model.

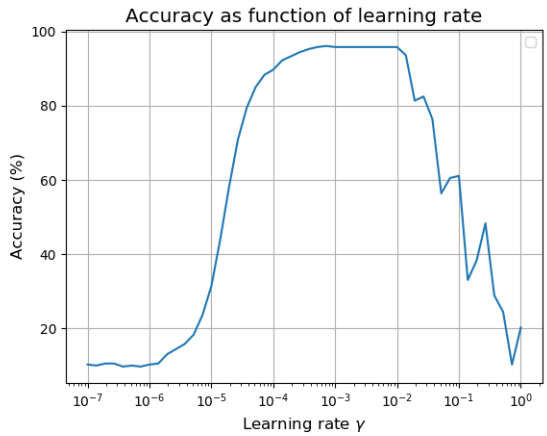

Figure 13. Classification accuracy plotted against learning rate γ for the logistic regression case. Here, we've trained running over 1000 epochs.

In figure 14, we have plotted the confusion matrix of the logistic regression model, using the learning rate $\gamma = 10^{-2}$, for 1000 epochs training the model. We see that our model clearly correctly identifies most of the digits, which is a good result. Notice the fact that the number 8 more often than random chance gets prescribed to be predicted as 1, which is interesting, as these numbers look nothing alike.

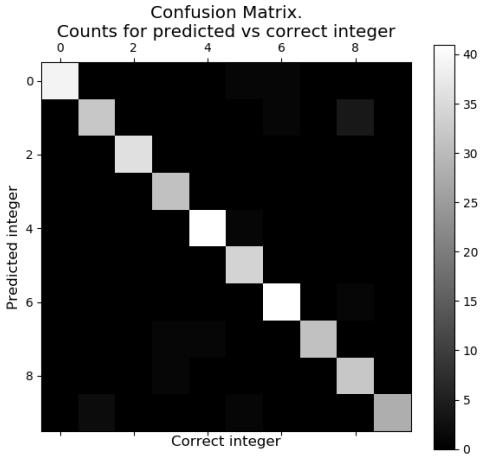

Figure 14. Confusion matrix of logistic regression model using learning rate $\gamma = 10^{-2}$ running through 1000 epochs. The color bar indicates the number of 'counts' for each individual case.

V. DISCUSSION

This discussion will mainly focus on 3 different areas. First, we will compare our results from running SGD with previous methods to see how they stack up. Secondly, we will discuss the case of how our neural network code dealt with approximating a function. Finally, we will look at how both logistic regression and neural networks dealt with the classification problems.

A. Stochastic Gradient Descent Comparisons

In figure 5 & 6, we have plotted the differences in the MSE score for the SGD approach and the matrix inversion approach when doing an Ordinary Least Squares-fit to our data. In both of these plots, we can see that both in terms of complexity of our model, as well as in regards to computational time spent, the SGD method is worse off than the matrix inversion counterpart (for our selected number of data points N, epochs, noise and minibatch size).

It is apparent from figure 6 that the SGD approach to fitting is both a lot slower, and more prone to large errors, as opposed to the matrix-inversion counterpart. The fact that the matrix-inversion counterpart is more accurate is to be expected, as we are using a pseudo-inverse functionality in Numpy[12], which circumvents some of the singular-matrix problematics we might usually encounter. However, considering that the matrix inversion scales as $\mathcal{O}(n^3)$, with n being the number of matrix elements, the fact that the gradient method scaled this terrible in comparison throws some shade on the validity of SGD as an effective method.

It is worth mentioning that for figure 6, we only scouted the range between polynomial degrees 20 and 40. The primary reason for this is the fact that any internal timer within python didn't have the resolution to properly time the matrix inversion consistently for lower polynomial degrees. This also means that we have moved well into 'overfitting territory' for both methods, and the absolute numbers therefore needs to be taken with a pinch of salt.

One thing we however can deduct from this plot is how the errors tend to scale. As the computational time increases for the SGD methods, the error flies to the roof, indicating large overfitting, while the tendency for a slope for the matrix inversion method isn't that steep. A first approximation therefore shows a tendency for a better scaling in terms of computational time vs error for the matrix inversion as opposed to our gradient approach.

Regardless of gradient methods being more intuitive to implement, they bring both more errors, and overall longer computations for comparable results. This is however without a look into gradient boosting methods, or other optimization methods like the momentum-based SGD or ADAM optimizer, which could both better performance and computational efficiency.

For the Ridge-case, we have in figure 7 plotted the mean squared error, together with our constant choice of learning rate γ and the Ridge fitting hyperparameter λ. In this figure, it is apparent a correct choice of learning rate is much more influential to our results than the choice of hyperparameter λ. This leads us to conclude that the correct choice of learning parameter γ should be a priority when sweeping over the parameter space in the lookout for optimal parameters, while the hyperparameter λ don't need to be regarded in the cases where one is far away from an optimal learning rate.

B. Neural Networks and OLS, a Linear Regression Comparison

Comparing our neural network approach in figures 8-10 to that of the SGD methods in figures 5-6, we can immediately see that the neural networks are by far more accurate than the SGD approach. Comparing the neural networks to the matrix-inversion method of OLS also indicates that the accuracy scores of the neural networks are comparable in orders of magnitude.

Disclaimer: it is important to note that our results for SGD / OLS runs in figures 5-6 are not directly comparable to that of the neural networks in figures 8-10. We have a different size of our data set, as well as a different noise term. However, we see tendencies such that we can make qualitative comparisons between the methods as already discussed.

It is from figure 8 apparent that finding an optimal learning rate is of upmost importance when training a neural network, and there is a fine line between an optimal choice and a bad choice. In figure 9, we see that the choice of the number of hidden neurons in each hidden layer tends to be better for a higher number of hidden neurons, but fluctuates a lot, so this parameter is not necessarily one we want to prioritize optimizing for. In figure 10, we see that the accuracy score is better for higher number of epochs. This is as expected, but one must also take into consideration that a larger number of epochs drives the CPU time up, and the accuracy will stagnate when the number of epochs grow. It is therefore important to here strike a balance between CPU time and the accuracy we want to achieve. In addition to these three parameters to optimize for, we could also optimize for the number of hidden layers, and other designs of the neural network like different activation functions and different number of neurons in different hidden layers.

One of the downsides of the Neural Network approach is the many parameters we can choose to optimize this for. We got the learning rate γ, as well as the hyperparameter λ, the number of hidden nodes, number of hidden layers, and this can further be made more complex by having a varying number of nodes in each of the hidden layers, as well as different activation functions for the individual nodes. This creates a high-dimensional parameter space that is a possibility to explore and optimize for, but at the cost of computational cycles.

One thing to note is that we've only used the sigmoid activation function when applying this to our problem. The primary reason for this is the fact that the other activation functions (Such as RELU, Leaky RELU, etc..) had a tendency to return diverging results with no physical interpretation. For this reason, results for the other activation functions have not been included here, as they were not sensible. However, given that the Franke function spits out values roughly in the range $f(x, y) \in [0\ ,\ 1.2]$, and the sigmoid function spitting out values in the range $\sigma(x) \in [0\ ,\ 1]$, this could attribute to some of our error, as the output of the neural network couldn't return values in the upper range of the Franke function. By use of other activation functions, or by normalising $\sigma(x)$ with a factor 1.2, we could potentially reduce the error even further. This has not been attempted by us.

Due to the high-dimensional parameter space that the neural networks create, one can assume that an optimally trained neural network with the correct design should prove to be a lot greater than the standard linear regression approaches. This is however a computationally heavy burden to perform, making an optimal model of neural networks theoretically within reach, but computationally difficult to achieve. It could however be made the case that, given a higher dimensional function $f(x_1, \cdots, x_n)$ to fit, the neural network could prove better than the least squares-approach. This is a possible extension of our problem.

C. Classification and Digit Recognition

Looking at our results for the classification problem of the MNIST data set using logistic regression in figures 13-14, we see that our program is quite accurate, as long as the learning rate is optimized. We observe that, for an optimal choice of learning rate, our logistic regression model reaches roughly 95% correctly identified digits, which is fascinating if we would say so ourselves. It doesn't perform perfectly however, and funnily enough it wrongly assumes the number 8 to be a 1 more often than chance would imply, but overall the results are quite good. It is however vital that the correct learning rate γ is chosen, as is apparent in figure 13, as a poorly chosen learning rate, both too high and too low, could result in less-than-optimal results.

Compared to the neural networks in figures 11-12, we see that our results are comparable. The neural network scored up to 96% accuracy for the best cases, which is of the same accuracy as for the logistic regression case. In figure 11, it is apparent that the correct choice of learning rate γ is vital in establishing an accurate model, whereas λ also plays a minor role. This is the case for our specific choice of design of the network, and the sensitivity and optimal parameters might change given a different design of neural network.

To affirm that our results make sense, we can in figures 14 & 12 see the confusion matrix plotted for logistic regression and neural networks respectively. The fact that most counts of predictions vs. actual target values landing on the diagonal further affirms that our results are sensible.

For the neural network case, we have freedom in the design of the model, as well as several parameters to optimize for. As such, it could very well be that the neural network approach could be optimized for so well that it is indistinguishable in performance to a human. However, as with the linear regression case above, it also suffers from this curse of parameter freedom, as it is a heavy computational load to optimize for all of these degrees of freedom. From this, it is somewhat apparent that neural networks can potentially outperform the logistic methods, but at what cost?

VI. CONCLUSION

In this report, we've looked closely at linear regression methods, both in terms of gradient approaches, and in terms of neural networks. We've also looked at classification problems, comparing neural networks to that of logistic regression.

In regards to the SGD case, it is apparent that it is both orders of magnitude slower, and less accurate than its matrix-inversion counterpart. Even though a similar thorough comparison has not been done for Ridge, it is apparent that both scouting through for optimal learning rates and hyperparameters, in addition to running through several epochs and iterations, will bring the efficiency of these methods down. Despite gradient methods being a more intuitive approach to approximating a function, in regards to computational speed and accuracy, it is inferior to the previously discussed approaches of matrix inversion.

From our findings, it appears that neural networks are comparable in accuracy to that of the matrix inversion methods of OLS. However, given a larger set of parameters that we need to optimize for, the winnings might go up in the spinning, as we say in Norwegian. Whether or not the neural networks could outperform OLS methods for an optimal set of parameters, or in the case of a many-dimensional function to fit, remains an unanswered question. However, being comparable methods in this special case, it is also an interesting question to look into.

For the classification problems regarding the MNIST hand-written data set, we see that the logistic regression methods are comparable in terms of accuracy to the neural network approach. The neural networks are however slower to train, and with many free parameters and design choices to make for our model, is struck with the curse of the freedom of parameter choice. One must therefore, as with the linear regression case, strike a balance between accuracy and computational time spent training our models.

If anything, we are led to conclude with one vital thing: the choice of optimal learning rate using the above methods is the single most important parameter to fit. A wrong choice of learning parameter is a curse that could ruin any gradient methods' hunt for success.

[1] M. M. Carstensen, G. S. Cowie, and J. E. Ødegård, "An attempt of applying linear regression on parametrizing topography map data," (Dated: 02/11/2020).

[2] M. Hjorth-Jensen, "Lecture notes on machine learning, fys-stk3155/4155 - week 35: Linear regression and review of statistics and probability theory, p.17," (Dated: 07/11/2020).

[3] M. Hjorth-Jensen, "Lecture notes on machine learning, fys-stk3155/4155 - week 37: Ridge and lasso regression p.15," (Dated: 07/11/2020).

[4] D. C. Lay, "Linear algebra and its applications, p. 132," (2016).

[5] M. Hjorth-Jensen, "Lecture notes on machine learning, fys-stk3155/4155 - week 39: Optimization and gradient methods. p.38," (Dated: 02/11/2020).

[6] M. Hjorth-Jensen, "Lecture notes on machine learning, fys-stk3155/4155 - week 39: Optimization and gradient methods. p.43," (Dated: 02/11/2020).

[7] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, 1st ed. (OReilly Media, Inc., 2017).

[8] M. Hjorth-Jensen, "Lecture notes on machine learning, fys-stk3155/4155 - week 40: From stochastic gradient descent to neural networks. p.10," (Dated: 02/11/2020).

[9] M. Raissi, N. Ramezani, and P. Seshaiyer, Letters in biomathematics , 1 (2019).

[10] A. Saha, "Comparison between logistic regression and neural networks in classifying digits," (Dated: 04/11/2020).

[11] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, C. Suen, A. Coates, A. Maas, A. Hannun, B. Huval, T. Wang, and S. Tandon, "Softmax regression," (Dated: 04/11/2020).

[12] NumPy_Developers, "Numpy pseudo inverse documentaton," (Dated: 12/11/2020).