A Numerical Study of Regression and Classification with Neural Networks

Felicia Jacobsen
(Dated: November 13, 2020)

ABSTRACT

This report includes a study of Neural networks for both regression and a 10-class classification problem. This approach is compared with linear- and logistic regression using stochastic gradient descent with mini-batches. The data used for regression was a 3D surface computed with the Franke function with added noise, and for classification we used 8x8 images of hand-written digits from the MNIST database. With regression, the optimal MSE of 0.042 was obtained with the MLPRegressor from Scikit-learn's package, closely followed by our own implemented network with an MSE of 0.043, while linear regression gave an MSE of 0.054. For classifying the digits, our own network, Scikit-learn's MLPClassifier and logistic regression obtained equally high accuracies of 98%, but the two former methods needed 10 times as much training compared to logistic regression. This makes logistic regression with SGD the optimal approach for this data set.

I. INTRODUCTION

A highly versatile algorithm in Machine Learning was initially inspired of the architecture of the human brain. This algorithm is naturally called Neural Networks (NN) and can be used to perform many different tasks. Our main task is to implement this algorithm and study the performance of NN in both classification and regression problems.

What's fascinating about the NN, is that it's a fairly simple algorithm when you consider how many parameters it handles at once, and the fact that it can tackle highly complex problems in only a few simple lines of code. It can recognize underlying relationships in a data set, making them ideal for handling problems in Machine Learning. These problems can be recommending new videos based on user history in products like TikTok or YouTube, face recognition for smart phones, or handling data from sensors used for self-driving cars.

In the research presented, Neural Networks will be used to model both a regression problem and a classification problem. For the study of the regression problem, we will consider the Franke function which takes input values x and y and produces a output value which is a sum of four Gaussian terms. While for the classification problem, we will study the MNIST data set which is a set of pictures of hand-written numbers from 0 to 9 collected from American Census Bureau employees [6].

This report will be presented in six main sections. The first of which is this introduction, and the second one being a comprehensive theory section containing information about the applied concepts and algorithms when performing the numerical study. The third section contains the methods used when implementing the concepts mentioned in the theory for this specific purpose. In the fourth section, we present the result. These will be discussed in the fifth section. Finally, the last section will contain the conclusion with a summary of the findings from this research.

For further investigation and engagement, all of the results, report, figures and codes will be available at **this GitHub repository.**

II. THEORY

Gradient Descent

The Gradient Descent (GD) method is a versatile method for solving an optimization problem. One of the main properties of this method is that it changes parameters iteratively in order to converge towards a solution. In order to apply this method, we need the gradient of the cost function we want to use. The cost function is a measure of the predictive performance of the ML model. A typical example of a cost function is the mean squared error (MSE) which measures the squared deviance of the predicted output from the true output. When we want to obtain a predictive model with good performance, the cost function is essentially what we want to minimize.

The benefit with computing a gradient of a function is that we know the direction and how fast a function increases. Since we're dealing with a minimization problem, we want to go in the direction opposite of the gradient of the cost function. When we have reached the bottom and the gradient is zero, we have converged to a minimum. The method is therefore called Gradient Descent, since we're descending towards the bottom of a parameter space.

A possible parameter initialization is by guessing with random parameters in the model which we want to optimize, which is called random initialization. For each iteration, we improve these parameters gradually.

The main task is for the algorithm to improve the cost for every iteration. The improvement of the model parameters can be decided by the size of the step when moving towards the minimum. We call this the learning rate, and this is one of the most central hyper parameter when dealing with GD. Having a too small learning rate will be time expensive. Having a too large learning rate will lead to instability, for example by jumping across the minimum and ending up even further away.

One common challenge in GD is finding the global minimum. High dimensional parameter spaces can contain multiple local minimum or saddle points, and a naïve implementation of GD could indicate that we've found the global minimum when we in fact only have obtained a local minimum. One solution to this problem is to use **Batch Gradient Descent**. We split up the training set into many batches such that we now calculate the average gradient for the entire mini-batch. Each gradient step involves calculating the gradients for all data points in that specific mini-batch, and then average these gradients to perform a more sophisticated step. Thus, the model will perform a step based on a greater picture of the parameter space. This will reduce the risk of ending up in a local minimum.

The weights, $\hat{\beta}_i$ in our model is iteratively updated by using GD with mini-batches.

$$\hat{\beta}_i = \hat{\beta}_{i-1} - \frac{\eta}{N_b} \sum_{j \in b_k} \frac{\partial c_j(x_j, \beta)}{\partial \beta}, \tag{1}$$

where η denotes the learning rate, N_b denotes the number of data points in a mini-batch, b_k denotes the specific data points the mini-batch, and c_i being a general cost function.

The number of times we iterate through the entire training data set is referred to as epochs. If we have too few epochs, we might stop the process before we reach the minimum. The number of epochs is another central hyper parameter which we need to tune along with the learning rate in order to improve our model.

Since batch GD computes the gradient for an entire batch, it is not time efficient when dealing with a large training set [2]. A more efficient way of using GD is **Stochastic Gradient Descent** (SGD), which involves considering a batch of randomly chosen data points. This will make the algorithm iterate faster at the cost of slower convergence, and will make it manageable to apply to a large training set. One downside to SGD compared to Batch GD, is that due to its stochastic nature, the cost function will fluctuate over the training instances and will not necessarily decrease, only decrease on average. This in the long run, the cost will decrease with fluctuations.

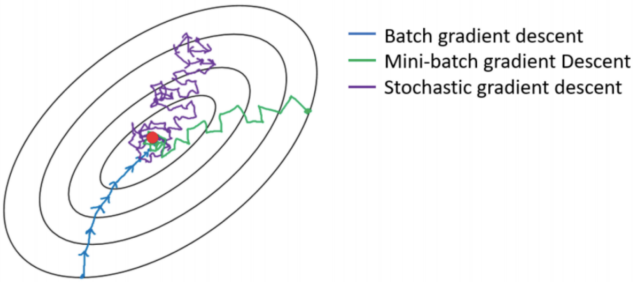

FIG. 1. The characteristic nature of various GD methods. The elliptical circles represent the surface of the gradient of the cost function, while red dot represents the minimum. Source: [3]

In figure 1, we're observing the characteristics of mini-batch gradient descent (green lines), stochastic GD (purple) and the GD with whole training set as the batch (blue).

With a cost function which contains many bumps and valleys, SGD is more sufficient at preventing getting stuck in a local minimum due to its stochastic nature, when compared to Batch GD.

One huge benefit with GD's, is that can be used for both regression and classification problems. It is also faster when performing Linear Regression compared to solving the singular value decomposition approach [2].

A. Logistic Regression with SGD

The Logistic Regression is a linear classification algorithm which models discrete outputs. When feeding an input, the algorithm will predict the probability of this input belonging to a specific class or event.

This algorithm is similar to Linear regression in the way that it combines the input values, X, linearly using weights or coefficients to predict an output value y. The main difference from Linear Regression, is that the predicted output is computed by the following

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2}$$

where $\sigma(x)$ denotes the function which we used to predict output $\hat{y}$, and x being an array of input values. This is how we predict the output when we have a 2-class classification problem. The function 2 is called the Sigmoid function, and will always provide outputs between 0 (first class) or 1 (second class). When dealing with GD algorithms and NN's, the functions for making the output predictions are called **activation functions**. For classification, the output is often denoted as a_i instead of $\hat{y}_i$ where the "a" is used to denote activated output, i.e. passed through the activation function.

With a random initialization of the weights β, we have now predicted our output probability by using the Sigmoid function. We now need to go back and tweak our weights by applying the gradient of the cost function 1. The cost function here, will in a regression problem typically be the MSE. When we're dealing with a classification problem, we deal with a cost function called the **Cross-Entropy**.

B. Multinomial Logistic Regression with SGD

Often, we have a multi-class classification problem, meaning that we want to predict outputs belonging to more than two classes. For example, we want to classify different dog breeds based off on their features like weight, height, fur etc. We can thus have hundreds of different breeds as classes. Using Logistic Regression for this classification problem is called Multinomial Logistic regression. Combining this method with SGD is analogous to having a single-layer Neural Network, which we will elaborate further in the later subsection about NN's.

When dealing with a multi-classification problem, we can no longer make the predictions based on the Sigmoid activation function 2. We also need to ensure that the required activation function will predict probabilities for every class, and that the sum of these probabilities is one. An activation function with this sum to one constraint is the Softmax activation function, and it is the multi-class version of the two-class Sigmoid function, computed with the following equation

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{3}$$

where x_i is an input element. The β denotes the weights and X being the input values. The denominator in the function scales the probabilities such that the predicted probabilities of each class sums up to one.

Now, we predict our probability vector for every observation using the Softmax activation function. How do we improve our predictions using SGD? The only thing we miss is the gradient for our multi-classification problem. We therefore introduce the Cross-Entropy which we briefly mentioned in the Logistic Regression using SGD for a two-class classification problem.

Cross-Entropy

The cross-entropy defines our cost function when we have a classification problem with NN's. This function measures the difference between K different distribution functions, where K is the number of classes in our classification problem. This cost function stems from Information theory where it's useful to measure the surprise of an event. The cross-entropy is a measure of surprise, and when we for example calculate that parameter y belongs to class 1, but in reality it belongs to class 2, then the cross entropy measures how surprised we are when we learn the true class of y. We get a low surprise if the output is what we'd expected, and a high surprise if we predicted wrongly. Let's look into more examples where we measure the entropy from distributions.

A probability distribution where most of the events have a small probability, whereas just a few number of events have a high probability is a probability distribution which is considered less surprising, and thus have a low entropy. Probability distributions where each event contains evenly distributed probabilities is more surprising and thus is considered to have larger entropy.

Let us observe the expression of the cross entropy for the multi-class case

$$E = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \ln(a_i), \tag{4}$$

where we sum over the number of classes n , and a_i denotes the predicted output by the activation function, which is the Softmax function from eq. 3 for the multi-class case, and Sigmoid 2 for two-class case. Both y_i and a_i are discrete probability distributions.

Since we're dealing with a SGD method, we need the gradient of the Cross-Entropy given that we have Softmax (Sigmoid multi-class case) as the activation function,

$$\nabla_a E = a - y, \tag{5}$$

where we have computed the gradient of the Cross-Entropy cost function 4 with respect to the activated output vector a from Softmax function.

Note that eq. 5 holds for both MSE and Softmax (or Sigmoid). Because the gradient of the cross entropy is equivalent in both regression (given MSE as cost function) and classification (given Softmax as activation function) when computing the partial derivative for E with respect to a in eq. 4. By setting $a = f(x) = x$ as the identity activation function, and with MSE being $\text{C(y,f(z))} = \frac{1}{2}(y - f(z))$, we find the derivative of the cost function with respect to f(z):

$$\nabla_a E = \frac{\partial C(y, f(z))}{z} = \frac{\partial C(y, f(z))}{\partial f(z)} \cdot \frac{\partial f(z)}{\partial z}, \tag{6}$$

$$= \frac{\partial}{\partial f(z)}(\frac{1}{2}(y - f(z))) \cdot \frac{\partial z}{\partial z}, \tag{7}$$

$$= -(y - f(z)) \cdot 1 = f(z) - y, \tag{8}$$

$$= a - y, \tag{9}$$

and obtain the equivalent expression when combining $\nabla_a C$ with cross-entropy loss with Sigmoid or Softmax activation functions. Note that the derivative of MSE is in reality $2(a-y)$, but it's generally accepted to use eq. 9 since the factor 2 can be baked into the learning rate.

In some cases, it can be useful to add a regularization parameter to the Cross-Entropy. The cost function will be

$$E = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \ln(a_i) + \frac{\lambda}{2n}\sum_{j=1} w_j^2, \tag{10}$$

which is the Cross-Entropy with an L2 regularization term. Here, λ denotes the regularization or penalty parameter, and w_i denotes each weight. When including L2 regularization, we add a parabola with a minimum at the origin to the cost function surface. When the L2 parameter increases, the more steeper the parabola will become. When L2 penalty is too large, this will overwhelm the cost function because it will distort the surface such that we can hardly use this to modify our weights. More specifically, if the increase in penalty is larger than the decrease of the cost, then the net effect of regularization is that the total cost gradient will increase. This effect can make us go further from the optimal solution.

A benefit of regularization however, is to minimize the magnitude of the weights in order to reduce the possibility of overfitting the training data. In many cases, L2 regularization may show accuracy improvement for both regularization and classification.

Neural Networks

A fully connected neural network consists of an input layer, hidden layers and an output layer, where each layer consists of one or several nodes. Choosing the architecture of the network like the number of hidden layers and nodes as well as other hyper parameters is a matter of trial and error. There exists some general guidelines for the architecture of a NN in terms of layers and nodes.

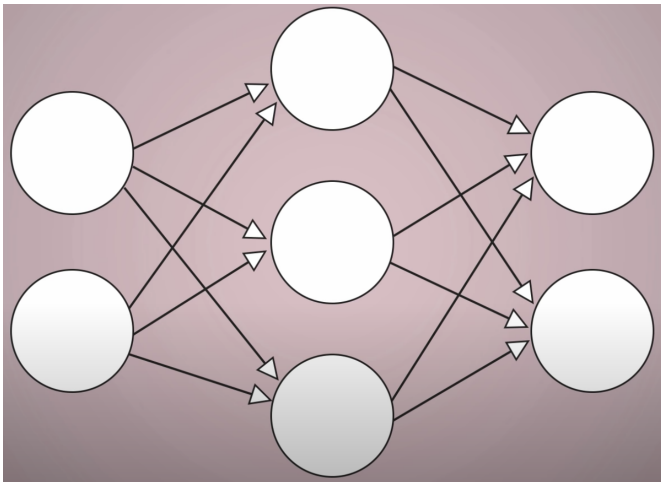

FIG. 2. Figure represents an artificial Neural Network with one hidden layer. This serves as a simple illustration of the architecture of a NN. Figure from [1]

The illustration 2 illustrates the NN where the circles are nodes. In this figure, a layer consists of the set of nodes that are on the same vertical axis.

The first two nodes on the left illustrates input layer. If we have two inputs, suppose a point in the plane (x, y), the one node from the input layer would take the x-coordinate, while the other would be the y-coordinate. Generally speaking, the number of nodes in the input layer is the same as the number of features from the input data.

There are three nodes in the middle, which they together serve as the hidden layer. Neural networks often consists of several hidden layers, and a NN with many hidden layers is usually referred to as a Deep Neural Network. The last layer is the output layer, where each node signifies an output.

For regression, these output nodes may be coordinates, but for classification, these are probabilities of a specific output belonging to a specific class. For multi-classification problem of K classes, we thus have K different nodes containing probabilities for each of the K classes.

In a Feed Forward Neural Network the information moves forwards from one layer to the next. The information consists of inputs, weights and biases. When we feed our input from the input layer to the first hidden layer, we matrix-multiply the input with the weights and thereafter sum with the bias vector. For the NN in figure 2, the first weight matrix has the shape of 2×3 since we have two input nodes and three nodes in the hidden layer. The first bias vector has the the shape of 3 to match the number of nodes in the hidden layer (which is also the shape after the matrix multiplication

of the weights).

In the hidden layer we predict an output which we can pass forward in another hidden layer, or directly onto the output layer. After an input has passed through weights and biases, the output is also run through an activation function. Remember that we want to find underlying relationships in the data set, often non-linear. When the data follows a linear relationship, we in fact don't need a NN to solve these kinds of problems. Having NN's to solve regression or classification problems allows us to detect non-linear relationship in the data.

Input and output data very often follow a complex relationship, and the behaviour is often non-linear. This is where activation functions come in, which introduce the possibility of modelling non-linear relationships in the data set. There are several different activation functions, like the Sigmoid as mentioned earlier. There is also the hyperbolic tangent function, the Rectified Linear Units (Relu), the Leaku Relu, Relu6, etc. The mathematical expressions of each activation function and their derivatives are listed in Appendix B.

We now have an predicted output by the activation function. We can choose to have more hidden layers with different activation functions, or choose to pass it directly to the output layer.

The algorithm of passing forward an input through to the next layer is as follows

$$a^l = f(z^l) = f(y^{l-1}W^l + b^l), \tag{11}$$

where f is our activation function, y^{l-1} is one observation of input data and is a column vector as well as z^l, the W^l denotes the weight matrix and the b^l is our bias column vector. The shapes are $z^l, b^l \in \mathbb{R}^n$, $y^{l-1} \in \mathbb{R}^p$, and the weights matrix has the shape of $W \in \mathbb{R}^{p\times n}$. Note that the letter l tells us which layer we're currently in. This algorithm is called the *forward pass*, but we also need to train the network in order to make better predictions.

1. Initialization

An untrained NN must be able to pass a signal all the way to the output layer in order to be trained, and ideally we don't want this signal to be neither too small nor too large. When we first create the weights and biases, we need to initialize these in order to produce an output. We thus make a stupid neural network.

It is common to initialize the biases to be zero, but not the weights. Initializing all the weights with zeros gives the consequence of the input signal vanishing. If the weights are initialized with too large values, the signal grows largely through each layer and diverges, but too small values might cause the signal to shrink when it passes through each layer until it's too small to be used.

There are two popular methods for initializing the weights, the first being Xavier/Glorot Initialization and the second being He Normal Initialization [5].

Both of these initializations assign the weights from a Gaussian distribution, where we have zero mean and a finite variance. With each passing layer, we want the variance of the weights to remains the same in order to keep the cost gradient from exploding or vanishing. We thus need to initialize the weights such that the variance remains the same for every input parameter. Xavier et al. [5] arrived with the variance being $\frac{1}{N_{l-1}}$, where N_{l-1} is the number of neurons in the previous hidden layer. He initialization is almost identical, only that we have a factor 2 in the numerator.

2. Forward Pass

The algorithm in eq. 11 sums up what happens in the forward pass. Between every layer, the output from the previous layer is matrix multiplied with a weight and added with a bias being column vector, passed through an activation function. When we have a regression problem, we use the identity activation function ($f(z) = z$) on the output layer, since we want unbounded values. This is the same as having no activation function between last hidden layer and output layer.

When we're dealing with a classification problem however, we want to predict bounded probabilities in the nodes of the output layers. If we have two nodes in the output layer, we need the Sigmoid since the two predicted values in the two nodes represent the probability of of the input being in class 1 and class 2. If we have more that two nodes in the output layer, we have a multi-classification problem, so we need the Softmax to predict bounded probabilities for each node in the output layer and these must sum up to 1. We summarize this into two points

- Regression $\rightarrow$ no activation function between last hidden layer and output layer (unbounded values).
- Classification $\rightarrow$ Softmax or Sigmoid between last hidden layer and output layer (bounded probabilities which sum up to 1).

3. Backpropagation

We have now initialized the weights, and have obtained predictions by the forward pass algorithm. The last step is to train our network to make better predictions. The training part of the NN is when send a signal backwards and modify our weights and biases

in order to improve our model. This is based on the Gradient Descent method.

Let us first look at the equations which summarize the backpropagation algorithm

$$\delta^L = \nabla_a E \odot f'(z^L), \tag{12}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l), \tag{13}$$

$$\frac{\partial E}{\partial b_j^l} = \delta_j^l, \tag{14}$$

$$\frac{\partial E}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l, \tag{15}$$

where δ is often called local gradient, error, or simply delta. This local gradient is used to update the weights and biases. The $\nabla_a E$ denotes the gradient cost function with respect to the activated signal a. $f'(z^l)$ denotes the gradient of the activation function in layer l. w^{l+1} denotes the weights matrix in layer $l+1$. While $\frac{\partial E}{\partial b_j^l}$ and $\frac{\partial E}{\partial w_{jk}^l}$ denotes the partial derivative of the cost function with respect to biases and weights in layer l, respectively.

The weights and biases are updated using 14 and 15 as well as the learning rate

$$b_{j,new}^l = b_{j,old}^l - \eta \frac{\partial E}{\partial b_j^l}, \tag{16}$$

$$w_{jk,new}^l = w_{jk,old}^l - \eta \frac{\partial E}{\partial w_{jk}^l}, \tag{17}$$

where E is the cost function and η is the learning rate. What we observe in eq. 16 and 17, is that the gradient of the cost function with respect to each parameter in the network uses the gradients to update each parameter with a GD step. The model will be trained by applying backpropagation.

One epoch is when all the observations in our training set has been used for training (forward pass + backpropagation) once.

The learning rate and the number of epochs are hyperparameters we need to tune in order to improve our NN, as we saw previously for GD. But now we also need to include the activation functions, the number of layers and the number of nodes in each layer which we need to tune in order to improve the NN. When tuning these parameters, we have a chance of observing the case of exploding or vanishing gradients.

Exploding and Vanishing Gradients

What we observed in the backpropagation is that we used the gradient of the cost function with respect to each parameter to update the weights and biases. The parameters is updated using a GD step.

These gradients can often get smaller and smaller as we pass each hidden layer. When these gradients become too small, the weights and biases in the lower layers will not get updated and are left unchanged. Thus, training will not result in a improved prediction. This problem is referred to as the case of vanishing gradients.

In other cases, we get the opposite effect of the vanishing gradient problem. The gradients gets larger and larger when passing through each hidden layer and will evidently explode such that the algorithm diverges, and we can't update the parameters with divergent values. Both exploding and vanishing gradients are common problems for DNN's.

MNIST Data Set

The MNIST database consists of collected handwritten digits released in 1999. They are often reviewed as the "Hello World" example when training NN's for image classification. The database consists of 70.000 images of size 28x28 pixels where each image is labeled from 0 to 9. These images serve as a basis for benchmarking classification algorithms. The benefit of using this specific database is that it's already labeled and is easy to interpret.

III. METHOD

Linear Regression Using Stochastic Gradient Descent

For the study of linear regression using SGD, we used generated two arrays of normally distributed values between 0 and 1. These arrays serve as the x- and y-coordinates and are inputs for the Franke function which serves as our height, or z-coordinate. We generate a total of 1000 observations. We add an extra noise on our output, which is normally distributed and has a standard deviation of 0.2.

A design matrix of a total of 21 features with the highest polynomial degree of 5 has considered in the SGD method. The design matrix has been created using the *PolynomialFeatures* function, borrowed from the sklearn.preprocessing module. This is the same function which we analyzed with linear regression in Project 1 which is located at **this** GitHub repository.

Instead of obtaining a solution of the OLS regression problem based on the singular value decomposition, we now apply an iterative approach with the SGD method. We start by splitting the input (X) and output data (y)

into training and tests set, where test set contains 20% of the total data. The algorithm starts with a loop over the number of epochs, followed by a second loop, which loops over every instance for every mini-batch. The inner loop contains the gradient of the cost function. If the L2 regularization is larger that 0, the cost-gradient is the derivative of the Ridge cost function with respect to the weights β_i is given by

$$\nabla_\beta C^{\text{Ridge}} = -\frac{2}{N_{\text{batch}}} \sum_{\text{batch}_i} X^T(y - X\beta)^T + 2\lambda\beta, \quad (18)$$

where β is a column vector of model weights, initialized with with a uniform distribution function with $\beta_i \in [-0.5, 0.5]$ using **numpy.random.random(X.shape[1]) - 0.5** from the numpy framework, where X.shape[1] denotes the number of features in the design matrix. N_{batch} denotes the total number of data points in that specific mini-batch, y denotes the true output (Franke function with noise), and λ is the penalization parameter.

If we have no regularization however, we're dealing with the cost gradient of the OLS, namely the MSE cost function

$$\nabla_\beta C^{\text{OLS}} = -\frac{2}{N_{\text{batches}}} \sum_{\text{batch}_i} X^T(y - X\beta)^T, \quad (19)$$

which is the same as the cost gradient for Ridge regression when regularization $\lambda = 0$.

In the inner loop, the batch consists of random indices in the data set. These indices apply to the design matrix X (in axis=0) and for the true values y. We generate a list of random indices using **numpy.random.randint(0, N, size=batch_size)**, where N is the number of observations in our training set, and batch_size indicates. This list of random indices can take values from zero to the no. of observations in our training set, and has a total number of instances which is equal to the batch size. Some observations can be picked several times. This list of random indices is assigned to y and X, which evidently define the mini-batches for every loop.

Depending on the value of penalization, we compute the cost gradient with respect to every parameter and based on randomly picked observations with eq. 18 or 19. We thus have a cost gradient for each mini-batch, which is the mean of the cost gradients of all the observations in the batch. This will be used to update our weights using eq. 1, as explained in the theory section.

We present our results by calculating the MSE of the trained model on the test set with the number of epochs. Thus we can observe the time development of our SGD model. For every plot, we vary the number of batches and the learning rate.

When including regularization in the model, the MSE is shown in a heatmap where the learning rate and L2 penalization parameter have been varied.

Neural Network - Regression

The FFNN which is implemented as a class, taking a flexible number of layers and a flexible number of nodes in each layer, where we also can specify the activation function between each layer. When choosing the number of nodes in each layer we decrease by 2/3 of the number of nodes when moving from one hidden layer to the next hidden layer. The selection of the network's architecture will also come down to being a matter of trial and error.

We start by splitting the input (X) and output data (y) into training and tests set, where test set contains 20% of the total data.

1. Initialization

Once we have specified the layers and activation functions, we loop over number of layers and initialize the matrix of weights and bias vector. The weights and biases are shaped according to the number of nodes in the surrounding layers. The weight matrices are initialized using He initialization as referred to in the theory section. The biases are initialized with zeros.

1. weights = numpy.random.randn(layers[i], layers[i + 1]) * numpy.sqrt(2/ layers[i]),
2. biases = numpy.zeros((1, layers[i + 1])).

We also need to make a series of if-statements, because a list of strings are specifying the activation functions for each layer. The network class contains list attributes which specifies the activation functions and their gradients for all layers.

2. Forward Pass

The NN class contains a forward pass method which is an implementation of the algorithm in eq. 11. This method takes an input signal z, and loop over the number of layers and for each layer we activate the signal with that layer's activation function which we have specified. The forward pass thus returns the predicted output from the last layer.

3. Backpropagation

The NN class contains another method for training. We need to train or model by backpropagating the signal through our NN.

We start by looping over the no. of epochs, which we refer to as the outermost loop. A second level loop, iterates over instances in every batch, and we will keep referring to this loop as the main loop. In the beginning of the main loop, instances are randomly chosen from input and output-data in our mini-batch as described in the Linear regression using SGD implementation. We call the input- and output-data as **x_batch** and **y_batch**. We specify our signal **z** and activated signal **a** as empty lists, except the fist column in list **a** is the first input **x_batch**. Using a third loop, which runs over every layer, we apply the forward pass algorithm again, but now we store lists of both the output signals and activated output signals for every layer.

Inside the main loops, we define the gradient cost function with the cross entropy 5, such that we can use the same NN for both linear regression and classification given that we have the Softmax or Sigmoid activation function between the last hidden and output layer. This is because if MSE or Softmax is implemented into eq. 4, the expression of the gradient is the same for each of the cases.

The local gradient, delta is initialized using eq. 12, using our list of activation derivatives of the signal **z** and multiplying with the gradient of the Cross-Entropy. We use the local gradient starting from the parameters obtained from the forward pass in last layer L to proceed.

In the main loops, the list of the cross entropy with respect to the weights, called **dW**, is initialized with the mean of the local gradient as in eq. 15 from layer L. The list of partial derivatives of cross entropy with respect to the biases, **dB**, in the current layer is updated with the activated signal from the second last layer and and the local gradient delta in layer L, as in eq. 14.

We add another loop inside the main loop, the delta loop, which iterate backwards through the hidden layers. Since we already have local gradient as well as dW, dB for the last layer, i.e. delta[-1], dW[-1] and dB[-1], we range the loop from 2 to L and will use the index of the layers [-i], such that we can update the lists from the second last layer to the first (the last layer gradients were computed right before the delta loop due to δ^L having a slightly different expression).

We have thus obtained dW and dB between all the layers, and loop over all layers inside the main loop to update the weights and biases as described in the theory, eq. 16 and eq. 17.

Let's summarize this NN class:

1. init(self, layers, activation functions): initialize weights and biases and adds lists of activation functions and activation derivatives to be used later based on layers and activation functions which must be list of integers and strings respectively.

2. forward_pass(self,x): method for passing data through NN and predicting outputs. Returns predicted outputs.

3. back_prop(self, x, y, η, epochs, batch_size): method which will train the network a number of times specified by epochs parameter. Returns nothing, but updates the weights and biases iteratively based on derivatives of activation functions, and thus trains the network.

To present our results, we calculate the MSE with respect to the test set, and predict the Franke function surface with the trained model in a 3D plot. We plotted the MSE for different values for learning rates with a fixed number of batches for every number of epochs up to 1000. Similarly, we did this but by finding the MSE for models of varying number of mini-batches as a function of epochs.

We listed the MSE in a table where the number of nodes, the number of hidden layers, the number of epochs, different combinations of activation functions, learning rates and the number of mini-batches in a table. We can thus observe this and compare what we observed when tweaking with these hyper parameters. To find the optimal solution, I could have implemented a GRID search method, but we're not necessarily interested in finding the optimal parameters, but make an attempt at learning the effects of tweaking each of these.

Lastly, we also collected MSE scores obtained from MLPRegressor in a table, which is the NN from the Scikit-learn package. We computed the resulting MSE's for different activation functions, learning rates, epochs, different layers and nodes as well as different mini-batch sizes.

The choice of parameters, especially choosing the depth of the NN would lead to either exploding or vanishing gradients. Some learning rates also made our algorithm explode, and also trying for higher epochs of 100.000 resulted in a very long run time and with no further improvements of the MSE. Thus, we have only presented the results of networks when using 100 to 10.000 epochs.

Neural Network - Classification

4. Preprocessing the Data

For classification, the MNIST data set, consisting of 1797 images is used. Every image has 8x8 pixels, and pixels of the images are flattened such that the input data

has a shape of (1797, 64), thus leaving us with 64 features, and 10 classes. The data is imported from Scikit-learn, with **from sklearn import datasets** where images is **datasets.load_digits().images** and labels/targets are **datasets.load_digits().targets**. In order for the NN to handle the target data, we need make it one-hot encoded. Such that an image which has a target 4 will thus become [0,0,0,0,1,0,0,0,0,0] in the one-hot encoded form. Thus, the total target 1D vector is transformed into a matrix of shape (1797,10). The input (X) and output data (y) is split into training and tests set, where test set contains 20% of the total data. The input data, X will range from 0 (black pixel) to 255 (white pixel) is scaled element-wise with the maximum value in the matrix, thus having all values range from 0 to 1. Scaling the data such that we have small variances in each observation evidently makes our data easier to interpret.

5. Performing Classification

In order to perform a classification, the last activation between the last hidden layer and the output layer is switched from being the Identity to the Softmax, since we now want bounded values (probabilities). The NN will thus predict probabilities for each image having belonging to a specific class.

The class containing the highest probability will be rounded off to 1 and the others will round down to zero, such that we now have a one-hot encoded array for every image which we can compare with the actual target.

6. Scoring the Classification Performance

We implement an accuracy function which measures the classification performance of the NN. This function counts the fraction of accurately predicted classes for the images with respect to the size of the test data. In this score function, the true target and the predicted target from the network are taken as arguments. The function iterates over every row in the matrices and counts the number of times where the label 1 is located at the same column (label) for the true target and the predicted target. The function returns corr/N, where corr is the number of correctly classified labels and N is the size of the test set.

We tested for different activation functions, different layers and nodes, batch sizes in order to obtain the highest accuracy score as possible. We gathered the highest accuracy scores of the NN together with the highest obtained from the MLPClassifier as well as the Logistic regression classifier in a table for comparison. MLPClassifier is a NN classifier from the Scikit-learn package.

7. Confusion Matrix

The result of the NN's prediction for the multi-class classification problem is presented using a so-called confusion matrix. The confusion matrix is a table that is often used to describe the performance of a classification model on a set of test data for where we predicted and true values. Each matrix element C_{ij} counts the number of inputs of label i which was predicted as label j. The values on the diagonal show the "true positives", i.e. the number of correctly predicted images for each class. If we for example have values outside this diagonal, the matrix provides information about what the wrongly classified images were labeled as (e.g. digit 8 was wrongly classified as a digit 9).

Logistic regression using Stochastic Gradient Descent

The NN's ability to classify the images from the MNIST data set will be compared to a simple Logistic regression function using SGD. Note that this is equivalent to the FFNN, but with having only an input and output layer, e.g. no hidden layers as mentioned in the theory section. Since we have a multi-classification problem with 10 classes, we need the Softmax as the activation function, and the Cross-Entropy as the gradient to update the weights. We will therefore not update the weights using dW like we observed for eq. 17 and thus never need the local gradient.

The input (X) and output data (y) is split into training and tests set, where test set contains 20% of the total data.

We update the weights, as analogous to eq. 1, but we add an additional term for regularization,

$$w_{jk,new} = w_{jk,old} - \frac{\eta}{N_b} \sum_{\text{batch}_k} \frac{\partial E}{\partial w_{jk}} + \frac{\eta \cdot \lambda}{N} w_{jk,old}, \quad (20)$$

where the Cross-Entropy error given Softmax as activation function, is computed by

$$\frac{\partial E}{\partial w_{jk}} = (y_{\text{pred}} - y_{\text{batch}})^T X_{\text{batch}}, \quad (21)$$

where y_{pred} is the predicted target, y_{batch} and X_{batch} denotes randomly selected true targets and inputs in a mini-batch respectively. N_b is the total number of observations in each mini-batch.

The results will be presented with in two figures. The will represent the accuracy score on the y-axis over time (epochs) on the x-axis, for different values of learning rates. The accuracy in plot label represent the final score after the last epoch.
In the second plot, we now include a L2 regularization

term and vary this like we did for the learning rates. We thus observe the accuracy score as a function of time (epochs) on the x-axis.

IV. RESULTS

The following results will be sectioned into two parts. The first contains results from regression problems of Franke function data with a simple SGD model and the NN. The last section contains results from the multi-classification problem of digits from the MNIST dataset. These results are produced by logistic regression using SGD, followed by the NN.

When specifying activation functions, we use T, R, LR, I, Sig and Sof as abbreviations which denote tanh, Relu, Leaky Relu, Identity, Sigmoid and Softmax respectively.

Regression

1. *Linear regression using SGD*

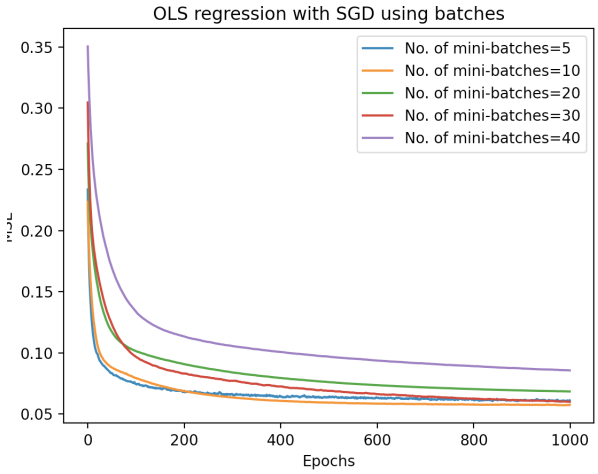

FIG. 3. The SGD method applied for the linear regression problem with MSE as cost function on Franke function data but for various number of mini-batches, while the other parameters are held constant. The learning rate was 0.001.

Figure 3 shows Linear regression using the iterative SGD method with MSE as the cost function. The number of mini-batches has been regulated, whereas other parameters like learning rate was held constant at 0.001. The maximum number of epochs was chosen to be 1000. The figure shows the different models' MSE vs. Epochs for the test set, based on the models being trained with the training data. The data is based off on Franke function output data and modeled using design matrix consisting of polynomials with highest degree being degree 5.

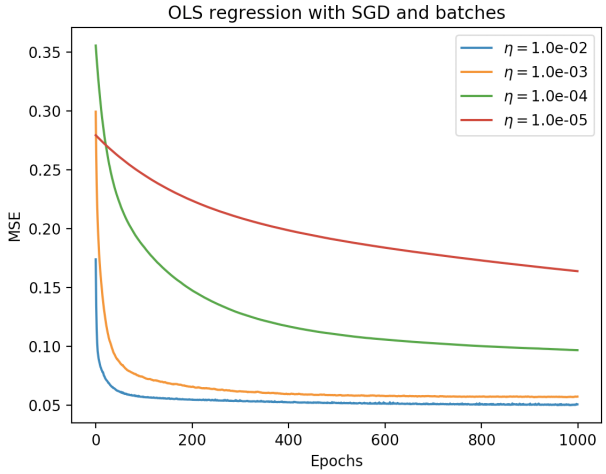

FIG. 4. Linear regression using SGD with a total of 15 mini-batches on Franke function data. The model is plotted with MSE of test data vs. the number of epochs, whereas the learning rate η is varied with 4 different values.

Figure 4 shows the Linear regression problem solved iteratively using SGD with a fixed number of mini-batches of 15 and a cost function being the MSE to update the model weights. This MSE is calculated using the predicted output and the true output from the test set, and the model is trained using the training data.

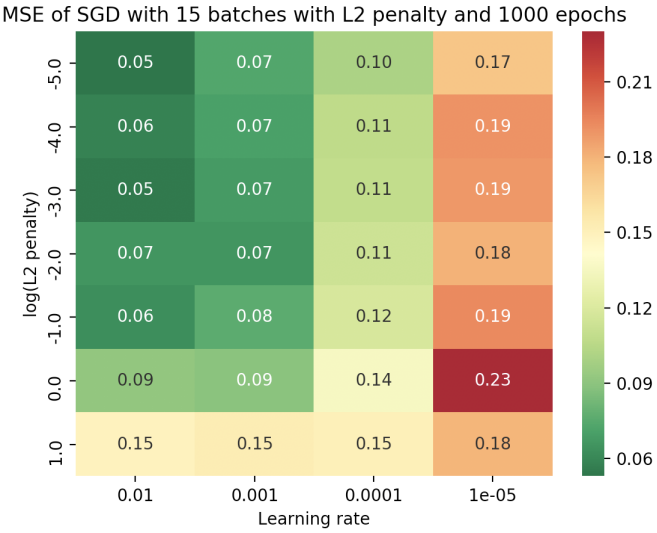

FIG. 5. A heatmap of the various MSE's computed with a combination of L2 penalties and learning rates with the Linear regression using SGD with 15 mini-batches. The data used is the Franke function data.

In figure 5 we observe a heatmap of the MSE's computed by the test set of Franke function data. The MSE's are computed based on the trained model using training data with hyper parameters L2-penalty and

learning rate has been varied over. The L2 penalty took values of $[10, 0, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}]$, while the learning rate took values of $[10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}]$. The red color in the heatmap illustrates higher MSE values, whereas the darker green highlights the lower MSE values.

TABLE I. Collected MSE for Linear regression models using SGD with and without L2 penalty on the Franke function data. The various parameters such as learning rate η and no. of mini-batches n_b was held fixed.

Epochs	**L2 penalty**	η	n_b	**MSE**
1000	-	0.01	15	0.054
1000	10^{-5}	0.01	15	0.056

Table I contains the resulting MSE with Linear regression using SGD. The two rows show the same model, sharing the equal number of epochs, the same learning rate η and the number of mini-batches n_b, but the two models are with and without L2 penalty.

2. *Neural Network*

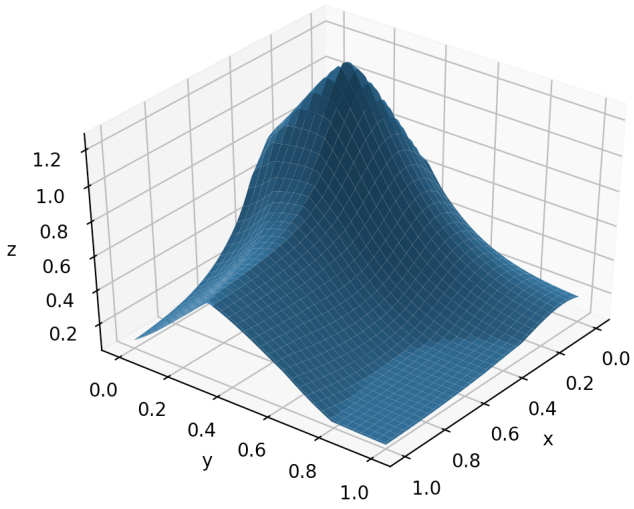

FIG. 6. Predicted surface on the total Franke function dataset based on trained NN using hidden 4 hidden layers with nodes [80,40,20,10] containing [T, T, T, LR, I] as activation functions for the hidden layers respectively as well as the output layer. The model used a learning rate of 0.1 and a total of 1000 epochs and 50 mini-batches.

Figure 6 shows the predicted 3D surface of the Franke function of the trained NN. The hidden layers have nodes [80,40,20,10], where each of these numbers specify the number of nodes in the hidden layers. For all of the hidden layers as well as the output layer, the corresponding activation functions is [tanh, tanh, tanh, leaky relu, identity] in that order. This NN gives an MSE of 0.043 of the Franke function test data.

TABLE II. Various number of runs of the implemented FFNN on Franke function data where parameters like number of layers and nodes, activation functions, learning rate η, no. of epochs **E**, and no. of mini-batches has been tweaked.

Hidden layers	**Activation**	η	**E**	n_b	**MSE**
[40, 25, 10]	[LR, LR, R, I]	10^{-3}	10^3	50	0.054
[40, 25, 10]	[LR, LR, Sig, I]	10^{-3}	10^3	50	0.13
[40, 25, 10]	[LR, LR, Sig, I]	10^{-3}	10^4	50	0.058
[40, 25, 10]	[LR, T, R, I]	10^{-3}	10^3	50	0.054
[80, 40, 20, 10]	[R, LR, T, R, I]	10^{-3}	10^3	50	0.060
[40, 25, 10]	[LR, Sig, R, I]	10^{-2}	10^3	50	0.058
[40, 25, 10]	[LR, T, R, I]	10^{-2}	10^3	50	0.053
[80, 40, 25, 10]	[T, LR, T, LR, I]	10^{-2}	10^3	50	0.048
[40, 25, 10]	[LR, Sig, R, I]	0.1	10^3	50	0.050
[80, 40, 25, 10]	[T, T, T, LR, I]	0.1	10^3	50	0.043
[100, 80, 40, 25, 10]	[T, T, R, T, LR, I]	0.1	10^3	50	0.043
[100, 80, 40, 25, 10]	[LR, T, R, T, LR, I]	0.1	10^4	50	0.047
[80, 40, 25, 10]	[T, R, T, LR, I]	0.1	10^3	80	0.044
[80, 40, 25, 10]	[T, R, T, LR, I]	0.01	10^3	60	0.047

Table II shows the computed values for different parameters of the NN. The parameters are the nodes in the hidden layers, the number of hidden layers, the activation functions, the learning rate, the total number of epochs, the number of mini-batches and the corresponding MSE for each network evaluated on the test data.

TABLE III. Various number of runs of the MLPRegressor, which is the NN from Sklearn package. The NN takes Franke functions data as inputs where parameters such as number of layers and nodes, activation functions, learning rate η, no. of epochs, and no. of mini-batches has been tweaked.

Hidden layers	**Activation**	η	**Epochs**	n_b	**MSE**
[100, 70, 40, 10]	[R,R,R,R,I]	0.01	100	30	0.042
[100, 70, 40, 10]	[R,R,R,R,I]	0.1	1000	50	0.070
[80, 40, 25, 10]	[R,R,R,R,I]	0.001	100	30	0.045
[80, 40, 25, 10]	[T,T,T,T,I]	0.1	1000	30	0.139
[80, 40, 25, 10]	[R,R,R,R,I]	0.1	1000	30	0.082

Table III shows the computed MSEs as results of different hyper parameters in the MLP regressor which is the NN from sklearn package. The hyper parameters were the number of hidden layers and number of nodes in each layer, learning rate η, the total number of epochs and the number of mini-batches.

TABLE IV. The accuracy scores by the predicted output of the test set of Franke function data for various regression methods: Logistic regression with SGD, NN and scikit-learn's MLPRegressor.

Model	**Hidden layers**	η	**Epochs**	n_b	**MSE**
Linear reg. with sgd	-	0.01	1000	15	0.054
MLPRegressor	[100, 70, 40, 10]	0.01	100	30	0.042
NN	[80,40,25,10]	0.1	1000	15	0.043

Table IV is the collected minimum MSE of each of the

regression models applied: Linear regression with SGD, MLPRegressor from Scikit-learn, and the implemented NN, each model with their respective parameters.

TABLE V. An overview of minimum MSE's of test set of Franke data with linear regression methods, with tuned hyperparameters by CV and bootstrap from Project 1 [4].

Resampling	**OLS**	**Ridge**	**Lasso**
CV	0.0542	0.0462	0.0478
Bootstrap	0.0568	0.0474	0.0483

Table V shows the MSE of Franke function test data of Linear regression models with OLS, Ridge and Lasso regression trained with CV and Bootstrap. This data was produced in Project 1, and is presented here for comparison.

Classification

3. Logistic regression with SGD - Single layer NN for multi-classification

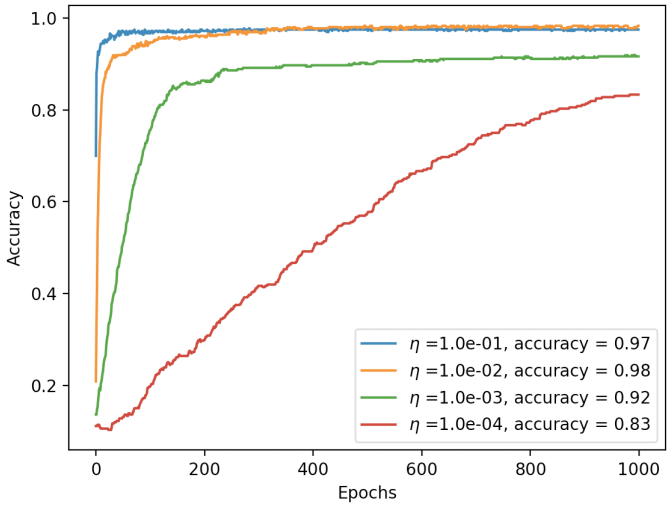

FIG. 7. The accuracy (fraction of correctly classified observations) vs. the number of epochs for the Logistic regression method using SGD and Softmax as activation for the multi-classification problem on MNIST dataset of handwritten letters. We have varied the learning rate, while the no. of mini-batches is held fixed at 15.

Figure 7 shows the accuracy score vs. the number of epochs for the Logistic regression model using SGD and the Softmax as activation function. There was no regularization included in this model. The MNIST dataset was used in the model in order to classify images of digits between 0 and 9. The figure shows models with different learning rates η, and their final accuracy scores after a total number of 1000 epochs. The number of mini-batches is held fixed at 15.

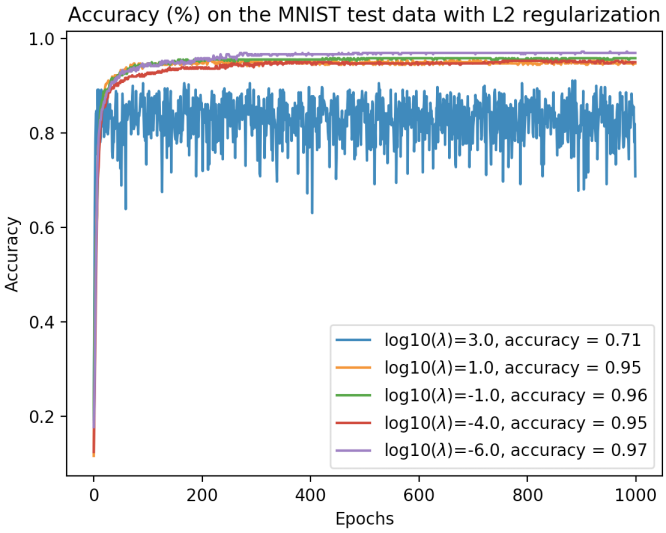

FIG. 8. The accuracy (fraction of correctly classified observations) vs. the number of epochs for the Logistic regression method using SGD and Softmax as activation for the multi-classification problem on MNIST dataset of handwritten letters. These models applied different values of the L2 regularization parameter

.

Figure 8 shows the Logistic regression model using the SGD of 15 mini-batches and a learning rate of 0.001. The L2 regularization parameter was varied in this model where we applied four different values of this parameter.

4. Neural Network

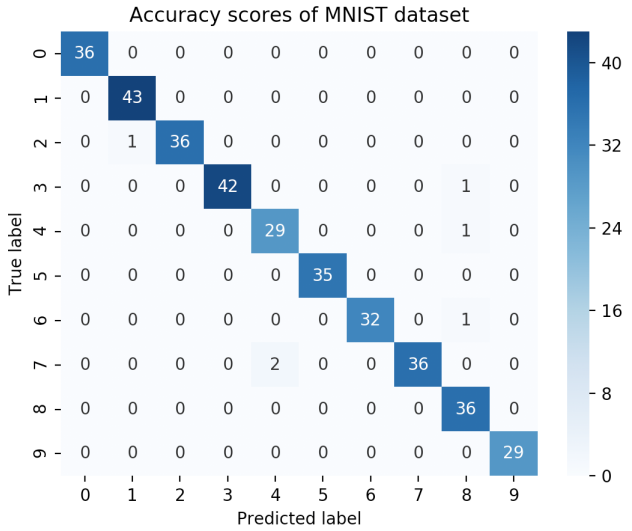

FIG. 9. The confusion matrix shows an overall accuracy score of 0.96. The layers consisted of layers with no. of nodes [64, 300, 200, 150, 100, 70, 10] (including the input and output layer) where we used activation functions [T, T, T, T, T, Sof] respectively. The number of epochs was 1000, with a learning rate of 0.1 and 15 mini-batches.

Figure 9 illustrates the confusion matrix of the NN's predictive performance on the test data from the MNIST data set, where the last activation function is the Softmax function. We had five hidden layers with nodes [300, 200, 150, 100, 70] and the activation functions [T, T, T, T, T, Softmax] in that specific order. The learning rate was 0.1 and total mini-batches set to 15.

5. Results from both Logistic regression and NN classifier

TABLE VI. An overview the accuracy scores by the predicted classes of the test set of MNIST data for various classifiers: Logistic regression with SGD, NN and scikit-learn's MLPClassifier.

Model	η	**Epochs**	n_b	**L2-penalty**	**Accuracy**
Logistic reg.	0.1	100	15	-	98%
MLPClassifier	0.01	1000	15	0.001	98%
NN	0.1	1000	15	-	98%

The results in table VI shows the highest accuracy scores achieved by the three methods. For logistic regression using SGD, we used Softmax between input and output layer. For the implemented NN, we had hyperbolic tangent as activation function between all layers except Softmax before the output layer, and a total of 5 hidden layers with 300, 200, 150, 100, 70 no. of nodes from the first hidden layer to the last. The MLPClassifier had Relu as the activation function and was initialized with no hidden layers.

V. DISCUSSION

Regression

When observing the linear regression with SGD and varying batch sizes in figure 3, the resulting MSE after 1000 epochs is varying with the different batch sizes. For smaller no. of mini-batches, the final MSE is lower compared to the MSE's for the larger no. of mini-batches. In fact, increasing the no. of mini-batches increases the MSE. The behaviour which we observe in the figure may be due to the fact that models with high no. of mini-batches require many more epochs before the gradient converges towards the minimal MSE. A large amount of mini-batches, will essentially lead us to update and correct the weights more often. Geometrically, the more updates, the merrier in this case. Because what we do here is when we're drawing several segments, each in the direction of the approximated (mean) gradient at the start at each iteration. When considering a more extreme case where one batch corresponds to the entire training data, this would be analogous to moving in the direction of the exact (global) gradient. Having a high no. of mini-batches would require more epochs for gradient convergence, and even though many no. of mini-batches require more epochs (thus, take longer time), the prediction would be better in terms of MSE for this specific case. Thus, using the model with 40 mini-batches would need more epochs than 1000 to arrive at the same MSE as for 5 batches, but have a downside of being computationally expensive and have a longer execution time.

When observing the effect of the choice of learning rate in 4, what immediately catches the eye is that larger learning rates converges faster to the minimum MSE when compared to larger learning rates for the same number of epochs. As referred to in the theory, a too small learning rate is computationally expensive in the sense that it requires more epochs to converge towards the minimum. Because of the stochastic nature of SGD, having a smaller learning rate may result in less fluctuations. Having a learning rate of 0.01 resulted in finding the minimum MSE for a relatively small number of epochs for this data set. The same learning rate also resulted in having small fluctuations in the curve compared to the other curves which are more smooth and thus indication a confirmation of what we explained in the theory regarding fluctuations with larger learning rates. We did not try larger learning rates than 0.01, but according to the theory, this could lead to larger instabilities.

A potential improvement to obtain a SGD model

converges fast at an optimal prediction but without being computationally expensive, is to introduce some adaptive learning rate. This concept is based on shrinking the learning rate every time the model improves. In this way, we can increase the chance at arriving at the optimal solution when we get closer to the minimum of the gradient without the risk of jumping further up the gradient surface. This approach can potentially make the model arrive faster for this optimal solution without needing a huge amount of epochs.

We have observed the effect of the following hyper-parameters: the number of mini-batches and the choice of learning rate η. We now move on to observing the effect of L2 regularization together with the learning rate. When first observing the heatmap in figure 5, values with the largest MSE value is when having high values of penalization, for example values of 10 and 1. Another area where MSE is relatively large, is where the learning rate is relatively small, with values of 10^{-4} and 10^{-5}. This is because smaller learning rates take longer time to converge as discussed in the theory and as we observed in figure 4. The minimal MSE is obtained with the combination of the smallest value of L2 penalty of 10^{-5} and the largest learning rate of 0.01, for the specific case of having a total of 1000 epochs and 15 mini-batches. When incorporating L2 regularization in the model we obtained the minimum MSE of 0.056, while having no penalty gave in fact a lower MSE of 0.054 as observed in table I. This indicates that having no regularization is more optimal for this specific regression problem.

When observing the performance of the NN from table II on the Franke data, the results give the minimum MSE of 0.043 with two types of networks. Thus performing better than for linear regression with SGD. This MSE indicates that this error may be a contribution from the noise (variance) of the Franke data which has a value of 0.040, and may also indicate that the network succeeds in not overfitting the data, ant thus does model the noise in the data. The one network which gave the minimum MSE consists of a total of 5 hidden layers, having 100, 80, 40, 25, and 10 no. of nodes respectively. With activation functions, Tanh, Tanh, Relu, Tanh, Leaky Relu and Identity. The second network had a total of 4 hidden layers, having 80, 40, 25 and 10 number of nodes in each layer, with activation functions Tanh, Tanh, Leanky Relu and Identity as last. For these two networks we had a learning rate of 0.1, and 1000 epochs and 50 mini-batches. For the first network, when increasing the number of mini-batches from 50 to 80 (second last row in table II) the result is an increasing MSE from 0.43 to 0.44. This may indicate that when increasing the number of mini-batches, it also requires that we need to increase the number of epochs to arrive at a low MSE as indicated earlier for linear regression, but this also may be due to the stochastic nature of the SGD.
When using the Sigmoid activation function, the results indicate that it requires more training to obtain sufficient predictions with a learning rate of 0.001. Having 1000 epochs gives an MSE of 0.13, while increasing epochs to 10000 for the same network reduces the MSE to 0.058. This is a decrease of more than a factor of 2, and is an indication that the Sigmoid function may require more training in order to converge to the optimal predictions compared to the other activation functions. Instead of increasing epochs, we observed that when we increased the learning rate to 0.01, the Sigmoid converged faster. Another thing to note about activation functions, is that our network does not seem to prefer a specific activation function over the other for this specific data set.

When comparing the implemented NN to the MLPRegressor from Scikit-learn in table III, this NN obtains a high MSE of 0.070 when having 0.1 as the learning rate. Also, we obtain a relatively high MSE of 0.139 when including the hyperbolic tangent as an activation function. This network seem to have a different behaviour compared to our own implemented version, even though we tried to give the same parameters as possible to make it act as our own network. For example, it obtains high MSE's for large learning rates and it also prefers one activation over the other, especially the Relu function. Our network gave a minimum MSE of 0.043 with a learning rate of 0.1 and 1000 epochs, while Scikit-learn's network give a MSE of 0.042 with a learning rate of 0.01 and only a total of 100 epochs. Scikit-learn's MLPRegressor uses "adam" as its solver, which is stochastic gradient-based method but still not similar to our own network's architecture. Based on the results in table IV, the solver "adam" converges faster for smaller learning rates a relatively low number of epochs when compared to our own NN, and thus being a better model for this data set in terms of regression.

As found in project 1 [4], we obtained MSE's from solving linear regression using the singular value decomposition approach. From these models, the minimum MSE is 0.0462 using L2 regularization and tuned parameters with 5-fold cross-validation. We obtained a range of smaller MSE's using our implemented NN which was even smaller than those obtained using linear regression with resampling techniques. Even though the SVD approach gives a sufficient MSE, the NN approach is faster and gives smaller MSE's as indicated by the results in table V from an earlier project [4].

Overall, the MLPRegressor provided by the Scikit-learn framework gave the best MSE for the case of regression on Franke data.

Classification

This section discusses the results of studying the 10-class classification problem of hand-written digits in the MNIST data set.

The logistic regression with SGD approach is equivalent to having a single-layer network (zero hidden layers). The achieved result was an accuracy score of 98% with a learning rate of 0.1 with a number of 100 epochs according to table VI.

When varying the learning rates of the one-layer network in figure 7, having a smaller learning rate resulted in decreasing the accuracy which was also observed for linear regression using SGD and in the NN as seen in figure 4 and in table II.

The figures 7 and 8 shows more clearly the characteristic stochastic nature of SGD, with the lines having significant fluctuations, when compared to linear regression with SGD in figures 3 and 4. We generated only 1000 observations for the regression problem, but had a total of 1797 images (observations) for the classification problems and thus almost doubling the number of data points. Another reason can be due to the number of mini-batches which was only 15.

We now incorporate L2 regularization into our model. The figure 8 shows various regularization parameters with a total number of epochs af 1000, a fixed learning rate of 0.01 and a total of 15 mini-batches. Note that we don't observe any effect of regularization before $\lambda = 10^3$. Even for a large penalty of 10, we obtain the same accuracy of 97% as for decreasing the penalty down of 10^{-6}. We need to go as high as having a penalization of 1000 to see an effect on the accuracy. Having a penalization of 100 or larger results in worsening the accuracy. High penalization would evidently lead to large errors, which in turn lead to low accuracy scores due to shrinking weights when modifying them for every iteration. Increasing the penalty would shrink the weights further, thus allowing less freedom in our model (lower model complexity). In cases of too high penalization this leads to underfitting. The effect of regularization from an L2 parameter of 1000 gives us an accuracy of 71% and may indicate that regularization does not improve our model since we observe increase of accuracy for very high or for very low values of penalization.

When observing the performance of our own NN, we observe a high level of accuracy when observing the confusion matrix in figure 9 as a result of training with a total of 1000 epochs. The NN seem to perform well for this classification problem. The information from our confusion matrix tells us that our model misclassified 7's as 4's on two images. It also misclassified a 2 as a 1 of one image, the 3, 4 and a 6 as an 8 in one occasion each. We have a total of 6 wrongly classified images out of a total of 360 images. The performance of the NN gives us an accuracy of 98% for 1000 epochs. This NN had an architecture of 5 hidden layers, with nodes of 300, 200, 150, 100, 70 from first to last hidden layer. We had a Tanh as activation function between each layer, except Softmax as the last.

The table VI shows a summary of all the classifiers used on the MNIST dataset with their corresponding hyper parameters and accuracy scores. We have gathered the set of hyper parameters which gave the best accuracy score of all the runs. The models are the Logistic regression using SGD with batches, the Neural Network and the MLPClassifier from Scikit-learn. Given that we needed a learning rate of 0.1 and 100 epochs for the one-layer network (Logistic regression), this indicates that this method converges faster at the optimal parameters compared to the NN and the MLP classifier. When trying to lower the number of epochs or the learning rate for these last two models, this only resulted in lower accuracy scores.

In terms of easy implementation and fast producible results, the results in table VI indicate that the one-layer NN (Logistic reg.) approach is sufficient when compared to the the DNN and the MLPClassifier for this specific case. All three methods arrive with an equally high accuracy of 98%, but the deep NN's needed 10 times as more epochs in order to arrive at the same accuracy score as the simple-layer network. The one-layer network has an benefit of predicting at high accuracy scores, having a short execution time, and being relatively easy to implement. Thus giving an indication of being the most sufficient classifier when compared to DNN's for this specific case.

These astoundingly high accuracy scores obtained from NN's might be due to the fact that the image resolution we're handling is only 8×8 pixels and thus being very small. The network will probably not be very good in predicting digits that are spatially shifted (moved toward the edges or rotated in the input). When dealing with larger images, the NN would not be able to perform equally as well. In practice, using Convolutional Neural networks is usually preferred for classifying high resolution images, because they can apply methods for handling spatial invariance, background noise, and other "distractions" from the actual object in the image.

VI. CONCLUSION

For the Linear regression problem with SGD, larger number of batches (over 30) resulted in needing a higher number of epochs to converge than for the models with mini-batches of 5 and 10. Similarly, lower learning

rates indicated slower training progressions and required more epochs to converge compared to models with higher learning rates. When combining different L2 regularization parameters we obtained worse MSE'S for increasing the penalties and decreasing the learning rates. The optimal MSE was at a value of 0.056 for 10^{-5} as penalty and 0.01 learning rate for 1000 epochs, while having no penalty gave a lower MSE of 0.054. These results indicated that the model performed better when not including any regularization for this specific data set.

We observed that our implemented NN obtained the lowest MSE of 0.043, while using the MLPRegressor from Scikit-learn resulted in the minimum MSE of 0.042, with 100 epochs and 0.01 in learning rate. Linear regression using SGD resulted in an MSE of 0.054. Using a neural network approach resulted in contributing to better results in terms of smaller MSE. The MLPRegressor gave the lowest MSE of all the three methods and was also faster in terms of having only 100 epochs, closely followed by our own implemented NN with 1000 epochs.

For classification on MNIST data set, a one-layer NN converged fastest to the best achieved accuracy of 98%. This classifying method gave relatively high accuracy scores and was computationally fast. Including L2 regularization gave low scores for very high penalization possibly due to underfitting, and little to no improvements for values in the range from 1 to 10^{-6}. For classification, Logistic regression using SGD as we recognized as a one-layer network was faster at computing the lowest accuracy in terms of number of epochs. While all three classification methods, including MLPClassifier and the implemented NN gave equally good accuracy scores of 98%. The latter needed 10 times more epochs to arrive at the accuracy as the one-layer network did. Thus, indicating that one-layer network was sufficient in terms of accuracy and fast in terms of execution time for this specific data set.

Future Work

Neural networks gave impressing results in both regression and classification when compared to Linear- and Logistic regression with SGD. We came to terms with a one-layer network being the best method for classification in this case, but a NN gave better results in terms of regression. It can be believed that the network can give us further improvements in terms of both regression when having more computational power, as the smaller learning rates may give more accurate results but need more CPU power to do so. Thus, NN can potentially perform even more impressively than it did for this study, when including more power. For classifying images, a different type of NN is generally used for this study, namely Convolutional NN's. It would be interesting to observe it's performance on this data set in terms of accuracy and execution time.

REFERENCES

[1] (2018). Neural networks explained - machine learning tutorial for beginners. Accessed: 02-11-2020.
[2] Aurelie-Geron (2007). *Hands On Machine Learning with Scikit Learn Keras and Tensorflow*. Oreilly.
[3] Hsieh, C.-J. (2019). Machine learning algorithms. Accessed: 10-11-2020.
[4] Jacobsen, F. (2020). A numerical study of linear regression and resampling methods. https://github.com/feliciajacobsen/FYS-STK4155/tree/main/Prosjekt%201. Accessed: 02-11-2020.
[5] Kakaraparthi, V. (2018). Xavier and he normal (he-et-al) initialization. Accessed: 15-10-2020.
[6] LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database. Accessed: 10-11-2020.

VII. APPENDIX A: GLOSSARY

A simple glossary list is presented. The intention of table VII is to serve as a encyclopedia for abbreviations for whenever needed.

Abbreviation	Full word
CV	Cross validation
ML	Machine learning
OLS	Ordinary least squares
MSE	Mean squared error
SD	Standard deviation
FFNN	Feed Forward Neural Network
NN	Neural Network
DNN	Deep Neural Network
GD	Gradient Descent
SGD	Stochastic Gradient Descent

TABLE VII. A glossary list containing often used abbreviations.

VIII. APPENDIX B: ACTIVATION FUNCTIONS

Sigmoid

$$f(z) = \frac{1}{1+e^{-z}}, \tag{22}$$

$$f'((f(z))) = f(z)(1-f(z)). \tag{23}$$

Softmax

$$f(z) = \frac{e^z}{\sum e^z}, \tag{24}$$

$$f'((f(z))) = f(z)(1-f(z)). \tag{25}$$

Relu

$$f(z) = \max(z, 0), \tag{26}$$

$$f'((f(z) > 0)) = 1, \text{else } f'(f(z)) = 0. \tag{27}$$

Leaky Relu

$$f(z) = \max(0.01 \cdot z, z), \tag{28}$$

$$f'((f(z) > 0)) = 1,\ f'(f(z) < 0) = 0.01 \tag{29}$$

Relu6

$$f(z) = \min(\max(z, 0), 6) \tag{30}$$

$$f'(0 < f(z) < 6) = 1, \text{else } f'(f(z)) = 0. \tag{31}$$

Tanh

$$f(z) = \frac{2}{1+e^{2z}} - 1, \tag{32}$$

$$f'((f(z)) = 1 - f(z)^2. \tag{33}$$

Identity

$$f(z) = z, \tag{34}$$

$$f'((f(z))) = 1. \tag{35}$$