A study of Stochastic Gradient Decent in Neural Networks and its applicability to categorization and low degree polynomial regression

Elin Finstad & Anders Bråte
Institue for Physics
University of Oslo
Norway
November 13, 2020

Machine learning has been shown to be able to produce results that are better than many statistical analytical tools in a large array of applications. Thus we see machine learning affecting areas in most parts of science and technology as a whole. The effectiveness in many machine learning algorithms comes down to a cost function minimization problem, which has for a longer time been solved by the iterative method of Stochastic Gradient Decent (SGD). This paper will introduce SGD and tie it into a neural network, applying both to the known Franke function, then ultimately to the well known MNIST dataset to recognize hand written integers. Stochastic gradient decent was shown to be able to produce results close to, but not exactly, the accuracy of OLS as they reached an MSE of around 0.18 and 0.04 respectively. In applying our Neural Network to the MNIST dataset we got an accuracy of 0.98 after tweaking the parameters.

I. INTRODUCTION

Machine learning has been shown in recent times to be next to none in applicability and in effectiveness in producing results, especially in a modern age where data collection and technology melts into culture itself. Though machine learning is known by the majority from uses such as image recognition, in which convolutional neural networks (CNN) work best, machine learning has a longer history in the applied sciences for categorizing. One example where use of machine learning goes way back is identifying particles in the LHC, or in general in particle physics where data is plentiful.

The algorithms that let machines understand handwritten letters, or even recognize faces are complicated and hard to understand. In this paper we start by looking at a Stochastic Gradient Decent (SGD), and then using many of the same ideas and concepts to make a feed forward neural network (FFNN) with back propagation, before examining a special case of Neural networks known as logistical regression. Building on this, we explore different strengths and weaknesses of SGD and the neural networks, as well as different variations and improvements.

The methods employed in this paper were examined by applying them to the Franke function [1]. Neural networks with back propagation is not something usually applied to function regression in the way we approximate the Franke function, however it is a natural progression from more intuitive regression methods that are easier to grasp when approaching machine learning. This paper will follow this progression as well. Initially looking at SGD applied to the Franke function, whilst comparing the results to that of the paper 'An introductory study of regression methods with machine learning and its applications' [1]. Then we develop our neural network and apply this to the Franke function. Subsequently we regard a more natural application of our neural network, and approach digit identification using the MNIST database of handwritten digits [2]. Lastly we approach the special case of logistical regression, again applied to the MNIST dataset.

II. THEORY

Gradient descent

The gradient of a function f with respect to the variable x tells how much the function changes when the variable x changes. Generally the function is often dependent on several variables, hence it is found by finding the partial derivatives.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots \right)$$

The gradient descent is an iterative method where, from a starting point (random starting point), the

method follows the gradient to reach the lowest point in the function space. In our case we are looking at the cost function, which tells us how far our model is from the actual training data. This cost function space often has a very complex shape, which is why we must use this iterative method, and not simply an analytical solution like when using Ordinary Least Squares.

Our position in this cost function space, which is analog to the weights in the weight vector is moved in the opposite direction of the gradient. The idea is to reach the minimum of a function by moving in the direction with the steepest descent. This 'direction' is found by calculating the gradient of the function with respect to the current point and move one step length in this direction. The method is repeated until the gradient is zero, at the minimum of the function. Given a starting point, the next iteration is given by

$$w_{i+1} = w_i - \eta_k \nabla_w C(w)$$

where η_k is a positive step length, also known as 'learning rate'. [3]

Hopefully the algorithm converges to a global minimum, but there is always a chance of ending up in a local minimum (depending on the cost function). The solution will depend on the cost function C, the starting point w_0 and the step length η_k. Whether we end up in a local or global minimum depends on the form of the cost function and the choice of starting point.

The convergence rate depends on the learning rate, or the step increment. Too large increment can cause divergence, while too small increment can result in slow convergence. A popular choice for learning rate is to choose a decreasing learning rate such that after a few steps, when we hopefully start to approach the global minimum, the learning rate decreases so to not overstep the global minimum. Eventually when fairly close to the global minimum (hopefully) we take smaller and smaller steps to hone in on the exact lowest point in the cost function space.

Stochastic gradient descent

Calculating the gradient of the cost function ∇C can be time consuming, causing the gradient descent algorithm to run slow. One way to speed up the algorithm is to estimate the gradient by computing it for only a small sample of randomly chosen training inputs, called a *mini-batch*. If the sample size is large enough, then the average of the gradients for this sample will be approximately equal to the average of gradients for all data points. This method is called a *stochastic gradient descent*.

The stochastic gradient descent thus works by picking out a randomly chosen mini-batch of training inputs and train with those. Then a new mini-batch is chosen and trained with until all mini-batches have been used. This completes one *epoch*, and the training starts over with a new epoch.
One advantage of this stochasticity is just that, It's a bit random. The random selection of datapoints in our mini-batch means the gradient has a bit of leeway to vary randomly, thus having the effect of helping us out of local minima, and towards the global minimum.

To speed up the algorithm even more, the stochastic gradient descent is often used with a momentum variable. The momentum increases the acceleration of the gradient in the right direction, hence the convergence rate is increased. With a momentum parameter $0 \leq \gamma \leq 1$ the update of the weights can be written as

$$v_{i+1} = \gamma v_i + \eta \nabla C \tag{1}$$
$$w_{i+1} = w_i - v_{i+1} \tag{2}$$

Larger value of γ results in higher acceleration towards the minimum of the gradient, while $\gamma = 0$ results in the ordinary SGD.

The weights are usually initialized with small values around zero, drawn from either a uniform or normal distribution. If all the weights are set to zero, then all neurons will have the same output and the network will be useless. This choice can have a lot to say in whether the method reaches the global minimum, so this is of importance. In developing our neural network additional care has to be taken in the choice of starting weights, which we will see later.

The architecture of neural networks

The first, and left most layer is called the input layer and the neurons in this layer are called the input neurons. The right most layer is called the output layer containing the output neurons. The number of output neurons does not have to be the same as the number of input neurons. The layers in between are referred to as the hidden layers, since they are neither input nor output layers. The number of hidden layers may vary from one network to another. The input and output layers are often straight forward to implement, while the hidden layers are more complicated.

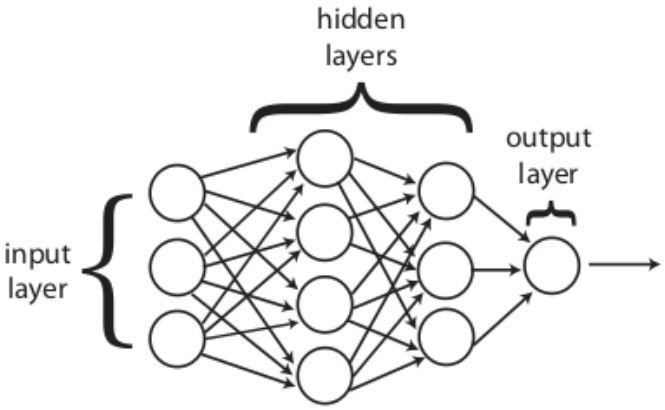

Figure 1. Structure of a neural network [4]. The input layer consists of all the features of the data, for example x and y for the franke function. The number of hidden layers and the neurons in these can be varied. The output layer consists of the approximated values, or a category.

Feedforward neural network

A neural network where the output from one layer is used as the input to the next layer is called a feedforward neural network. This means the network does not contain loops such that the input does not depend on the output from the activation function. These are the simplest forms of Neural Networks, and is often combined with a learning algorithm like back propagation.

Activation functions

In neural networks, the activation function of a node defines the output of the node given some input. In this report, the activation function will be denoted by σ, a function of the weighted sum z. The activation functions can be divided into linear activation function and non-linear activation functions. The linear activation functions are, as expected, a line or linear. Hence, the output will not be limited between any range. The identity function $\sigma(z) = z$ is such a function. The non-linear activation functions are more widely used, since they make it easier for the model to generalize or adapt to variety in the data.

Initialization of the weights is a topic of many research paper, and as we shall see later an important topic. Traditionally a starting weight of uniformly or normally distributed points about zero has been used, however methods such as the 'Xavier Glorot' method [5] have been shown to be able to reduce the efffects saturation of the activation function present when using the Softmax function or ReLU and Leaky-ReLU. A non saturated activtation function can be defined as $lim_{z\to\pm\infty}\sigma(z) = \pm\infty$, since the activation function spans all of $\mathbb{R}$. The Sigmoid function is a clear example of a saturated activation function, as for larger values of z it will at most return 1, which is perfect for categorizations.

Sigmoid

The sigmoid function looks like an S-shape between 0 and 1, given by

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{e^z+1}, \tag{3}$$

with the derivative

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(1-\sigma) \tag{4}$$

The main reason to use this is exactly that it exist between 0 and 1, such that the output will be constrained to values in this interval. This is useful when the model is used to predict probabilities. On the other hand, the sigmoid function can cause the neural network to get stuck as the derivative is, for the most part, very close to zero. When the derivative is zero, the weights and biases will not be updated.

Historically, the sigmoid used to be a common choice of activation function for neural network, but is nowadays often replaced by rectified linear units (ReLUs) and leaky rectified linear units (leaky ReLUs). [4, p. 47]

Rectified linear unit

A variant of the sigmoid function is the rectified linear unit (ReLU), which is given by

$$\sigma(z) = \begin{cases} 0 & z \leq 0 \\ z & z > 0 \end{cases} = \max\{0, z\} \tag{5}$$

with the derivative

$$\sigma'(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{6}$$

As seen, the sigmoid function saturates when the output gets close to 0 or 1, and the network stops learning. This is not an issue for ReLU activation function when the weights increase. On the other hand, ReLU does saturate when the weights are negative, such that the gradient vanishes and the neurons stop learning. Hence, the output from ReLU lies in the interval $(0, \infty)$.

ReLU is a bit faster to compute than other activation functions and can, in most cases, be used for the hidden layers. [3]

Leaky ReLU

Leaky ReLU is an improved variant of ReLU in the sense that it does not saturate, such that the gradient does not vanish. Instead of replacing the input by 0 if it is negative, the leaky ReLU function multiply the input by 0.01. Thus the output from leaky ReLU lies in the interval $(-\infty, \infty)$. The activation function and its derivative is given by

$$\sigma(z) = \begin{cases} 0.01z & z \leq 0 \\ z & z > 0 \end{cases} = \max\{0, z\}, \tag{7}$$

$$\sigma'(z) = \begin{cases} 0.01 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{8}$$

Since the gradient of the leaky ReLU function does not vanish, the neurons continue to learn no matter what the weights are. This makes leaky ReLU better than ReLU, which in turns is better than sigmoid. [3]

Softmax

In many cases it is convenient to use an activation function which outputs numbers from a probability distribution. Softmax is such an activation function. It is defined in a way where all the outputs are guaranteed to sum up to 1. [6, p. 71] As a result of this, when one of the outputs increases, then all the other ones decreases. Softmax is defined as

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{J} e^{z_j}} \tag{9}$$

with the derivative

$$\frac{\partial \sigma(z_i)}{\partial \sigma(z_j)} = \sigma(z_i)(\delta_{ij} - \sigma(z_j)). \tag{10}$$

[3] The output is also always positive, since the exponential function is positive. Hence, the output from the softmax function is a set of positive numbers which sums up to 1. It is favorable over sigmoid, as it is not guaranteed a sigmoid layer form a probability distribution even though it only outputs numbers between 0 and 1.

Cost functions

A cost function measures the error between the predicted values and the expected values and presents it in a form of a real number. In the neural network it used to measure how well the training works and to update the weights and biases based on how the cost function changes with respect them. For linear regression, we use the mean squared error (MSE) cost function, while for logistic regression we use the cross entropy cost function.

Mean squared error

Our cost function initially is a Mean Square Error (MSE) with a regularization parameter defined as

$$C(w) = (Xw - z)^T(Xw - z) + \lambda w^T w. \tag{11}$$

Here X is the design matrix, z is the expected value of the datapoints we are approximating, are the weights and λ is a regularization parameter (see [1]). The MSE is a statistical measurement which takes the average of the squares of the error, and is good choice of cost function when fitting a line. Since we in the gradient descent algorithm wish to follow the derivative of the cost function we naturally need the derivative as well. This is done in appendix B, and we arrive at the equation

$$\nabla C(w) = 2X^T(X^T w - z) + 2\lambda w. \tag{12}$$

Categorical cross entropy

When categorizing like we do when applying ML to MNIST for example, we are looking at probabilities for multiple categories. Naturally we therefore wish for all probabilities to lay in the interval $[0, 1]$ and for the sum of probabilities to equal 1. In addition we are no longer looking at points in space, but rather probabilities, meaning the simple MSE cost function will not give us what we want. We therefore introduce a cost function that gives a measure of how well our model is doing based on probabilities. The cross entropy cost function is given as

$$C(\hat{w}) = -logP(D|\hat{w})$$

where

$$P(D|\hat{w}) = \prod_{i=0}^{n} \prod_{c=0}^{C-1} [P(y_{ic} = 1)]^{y_{ic}}$$

and

$$P(y_{ic}|\hat{x_i}\hat{w}) = \frac{exp((\hat{a}_i^{hidden})^T \hat{w}_c)}{\sum_{c'=0}^{C-1} exp((\hat{a}_i^{hidden})^T \hat{w}'_c)}$$

In these equations D is the dataset and w the weights we optimize. What is done in the equations above is extending the more common binary cross entropy function to allow for multiple categories. see [3] for derivation.

In categorizing, like explained above, we are working with values between 0 and 1, meaning we wish to use the softmax activation function on the output layer to ensure this. This has the benefit that we can simplify the differentiation of this cost function to be

$$\frac{\partial C}{\partial z_k^L} = \hat{y_k} - \tilde{y_k}$$

[7] where $\hat{y_k}$ is the approximation, and $\tilde{y_k}$ is the actual datapoint.

Logistic regression

A special case of Neural networks that is often used for binary categorization, but can also be utilized on multiple categories is Logistic regression. This method uses the softmax activation function and categorical cross entropy, but no hidden layers. This has the benefit that many of the pitfalls that deep networks bring, like exploding and vanishing gradients and general overfitting is of no concern. This in turn however does also limit it from some of the benefits that deep learning provides, such as the ability to learn the intricacies of the datasets. Especially for datasets with many features this will not be sufficient to find the specific features of the data that will require deeper learning. This is also the reason Logistic regression often is applied to binary systems.

Backpropagation

Backpropagation is an algorithm mainly used to train the feedforward neural network. It is a fast way of computing the gradient of the cost function, where the gradient is computed with respect to any weight or bias. Hence, the expression tells how the gradient of the cost function changes as the weights or biases changes.

The goal of the backpropagation algorithm is to calculate the partial derivative $\partial C/\partial w$ and $\partial C/\partial b$. This is done by using the chain rule. The cost function depends on the activation function, the activation function depends on the weighted sum z, and z again depends on the weights and biases. Hence, the chain rule results in

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b}.$$

Doing these calculations separately for each weights and biases is inefficient. Instead, the backpropagation calculates the gradient for each layer individually in reversed order to avoid duplicate calculations and computing unnecessary intermediate values. By starting at the last layer and work backward, it is easier to see how the change of weights and biases affects the output compared to when starting at the first hidden layer. If the calculations had started at the first hidden layer, then we would most likely end up with duplicate and unnecessary calculations as you have to go through all layers to see how the change in the first one affects the output. The gradient of the weighted input of each layer is denoted δ^l from back to front and is often referred to as the error. We will use k to refer to k-th neuron in the $(l-1)$-th layer and j to refer to the j-th neuron in the l-th layer.

The backpropagation algorithm depends on four equations[6], where the first one is the equation for δ^l for the output layer is given by

$$\delta_k^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{13}$$

Then the error for the next layers $L-1, L-2, \dots 2$ are calculated recursively

$$\delta_j^l = \sum_k \delta_j^{l+1} w_j^{l+1} \sigma'(z_j^l). \tag{14}$$

The error is then used to calculate the partial derivatives of the cost function with respect to the weights and biases

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

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{16}$$

The partial derivatives are used to update the weights and biases for each layer $l = L-1, L-2, \dots 2$ by using the gradient descent with the learning rate η

$$w_{jk}^l \leftarrow w_{jk}^l - \eta\frac{\partial C}{\partial w_{jk}^l} = w_{jk}^l - \eta\delta_j^l a_k^{l-1}, \tag{17}$$

$$b_j^l \leftarrow b_j^l - \eta\frac{\partial C}{\partial b_j^l} = b_j^l - \eta\delta_j^l. \tag{18}$$

To avoid overfitting, a regularization parameter λ may be introduced in the backpropagation algorithm. In deep learning the regularization parameter penalizes the weight matrices of the nodes. One of the most common regularization parameters is L2. The penalty is added to the cost function, such that the gradient of the cost function is changed in the backpropagation algorithm. In L2 the cost function is given by

$$C(\theta) = \mathcal{L}(\theta) + \lambda\|w\|_2^2, \tag{19}$$

with the gradient, with respect to the weights, given by

$$\nabla C(\theta) = \nabla\mathcal{L}(\theta) + 2\lambda w. \tag{20}$$

The regularization parameter forces the weights to decay towards zero (but not exactly zero). By reducing the weights, a simpler, and thus more generalizable, solution with the same low error will most likely be selected. Selecting a more generalized solution will reduce overfitting.

III. METHOD

Setting up the neural network

To set up our neural network, we use classes. Each layer is an instance of a DenseLayer class, such that it is easy to change the number of neurons in and the activation function used for each layer. To put together the entire network from input layer to output layer, the instances are combined in a list and passed to feedforward and backpropagation. To test our network, we compare our results with the ones which can be obtained using keras [8].

Stochastic gradient descent

A pseudocode for the stochastic gradient descent is included below. To randomly split our data in a given number of minibatches, we shuffle an array containing all the indexes of the training set and index with this to randomly shuffle the data. Thereafter the shuffled data is split in a given number of equally sized minibatches.

Algorithm 1 Stochastic gradient descent

Initialize the weights w, the learning rate η and the momentum parameter γ
$v = 0$
for epoch $= 1 : n_{epochs}$ **do**
    Randomly split data in a given number of minibatches
    for minibatch $= 1 : n_{batches}$ **do**
        $v \leftarrow \gamma v + \eta \nabla C$
        $w \leftarrow w - v$ ▷ Update the weights
    end for
end for

Backpropagation

When implementing the backpropagation algorithm it is easier to calculate the four equations Equations (13) to (16) using matrix operations. Using the *Hadamard product* (denotede by $\circ$), an element-wise multiplication, the four equation can be rewritten as follows [6]

$$\delta^L = \nabla_a C \circ \sigma'(z^L), \tag{21}$$

$$\delta^l = \delta^{l+1}(w^{l+1})^T \circ \sigma'(z^L), \tag{22}$$

$$\frac{\partial C}{\partial w^l_{jk}} = (a^{l-1}_k)^T \delta^l_j, \tag{23}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j, \tag{24}$$

where the shapes are

$$\begin{aligned} \delta^L &= (n_{inputs}, n_{output}), \\ \delta^l &= (n_{inputs}, n_{hidden}), \\ \frac{\partial C}{\partial w^l_{jk}} &= (n_{features}, n_{hidden}), \\ \frac{\partial C}{\partial b^l_j} &= (n_{hidden}). \end{aligned}$$

Now that the four equations are on a matrix-based form, they are easy to implement in an algorithm. A pseudocode is included below

Algorithm 2 Backpropagation

Set up the input data, the activation z of the input layer and compute the activation function and the output a
Perform feedforward on the network. Compute all z^l and the output from the activation functions a^l for $l = 1, 2, \dots L$.
Compute the error for the output layer
$\delta^L = \nabla_a C \circ \sigma'(z^L)$
for l = L - 1 : 1 **do**
    $\delta^l = \delta^{l+1}(w^{l+1})^T \circ \sigma'(z^l)$
    Update weights and biases using gradient descent
    $w^l \leftarrow w^l - \eta(a^{l-1})^T \delta^l - 2\eta\lambda w^l$
    $b^l \leftarrow b^l - \eta\delta^l$
end for
$\delta^0 = \delta^2(w^1)^T \circ \sigma(z^0)$
$w^0 \leftarrow w^0 - \eta(X^T\delta^L) - 2\eta\lambda w^0$
$b^0 \leftarrow b^0 - \eta\delta^0$

The MNIST dataset

The MNIST dataset is very popular and widely used in machine learning, hence many programming software packages include this. We use the sklearn.datasets package which has the MNIST dataset in the call 'loadDigits'. Each datapoint is an 8 by 8 image of a handwritten integer, of which there are 1797 in total.

Accuracy

In order to understand whether the model has learned properly or not, we need to define a metric. The metric can for instance be the accuracy score, which gives the percentage of successful classifications. This metric is not differentiable, hence it can not be used to update the weights and biases in the backpropagation algorithm. The formula for the accuracy score is given by

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{25}$$

where t_i is the guessed targets and I is the indicator function for binary classification, 1 if $t_i = y_i$ and 0 otherwise.

IV. RESULT

First, we start studying the stochastic gradient descent method and compare it with the results from Project 1 [1].

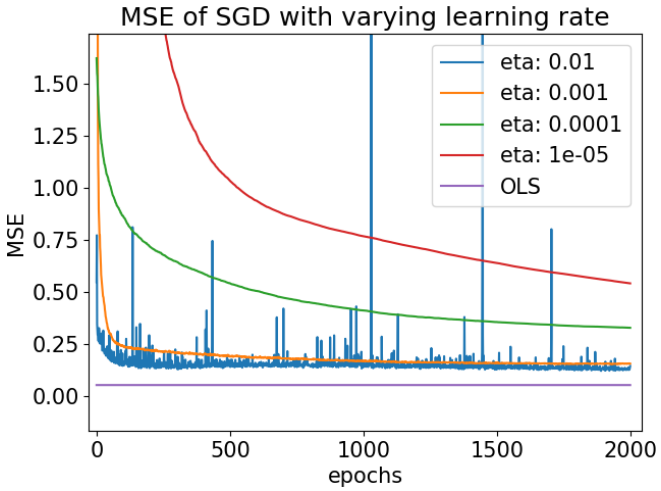

Figure 2. Progress of the SGD method with $\lambda = 0$ and $\gamma = 0.5$ for various learning rates, compared to the OLS method [1]. As can be seen lower learning rates converge slower, yet a high learning rate is highly unstable.

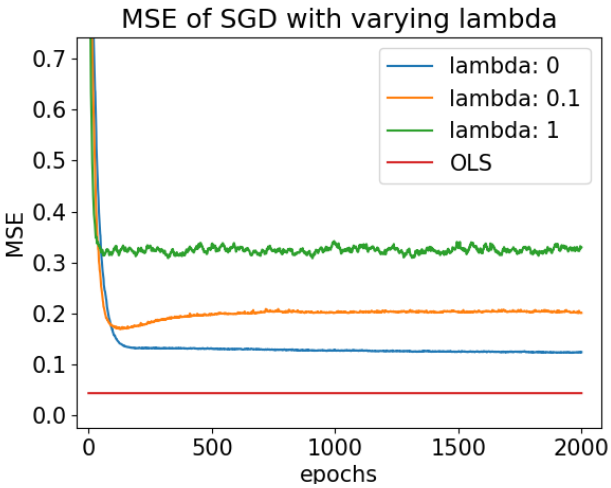

Figure 4. Progress of the SGD methods with varying λ, compared to the OLS method [1]. As can be seen, lambda is of no benefit for this method.

Moving on from the stochastic gradient descent method, we start testing our network on the Franke function with different activation functions for the hidden layers. In the following three heatmaps Figures 5 to 6, we look at the mean squared error for different combinations of epochs and minibatches. The MSE is given logarithmically to better separate the scores, and all the heatmaps for Franke function are created using 100×100 datapoints.

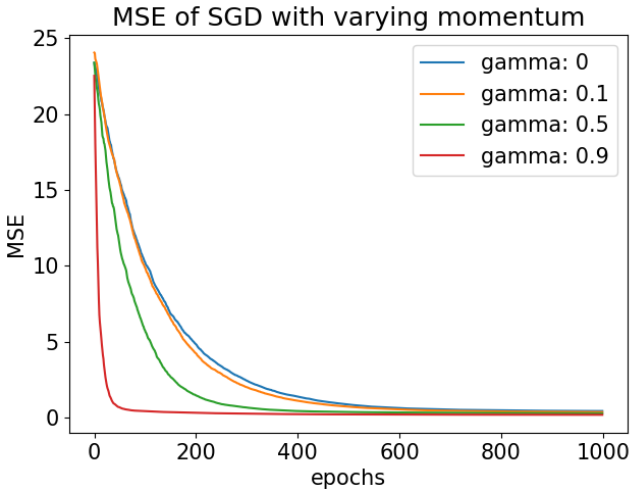

Figure 3. Progress of the SGD method with $\lambda = 0$ and $\eta = 0.00001$ for various momentum values. The gamma, or momentum variable does indeed help the method converge towards a minimum quicker.

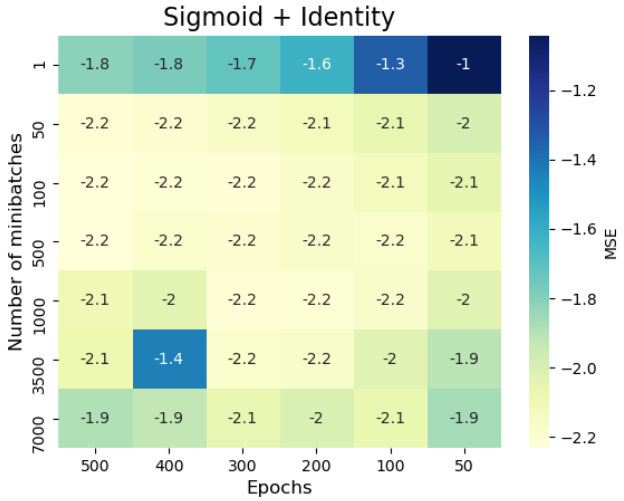

Figure 5. The mean squared error, on a logarithmic scale, for different combinations of epochs and minibatches for the Franke function. The penalty is set to $\lambda = 0$ and the learning rate $\eta = 0.5$. The network uses two hidden layers with 10 neurons and sigmoid as activation function, and no activation function on the output layer.

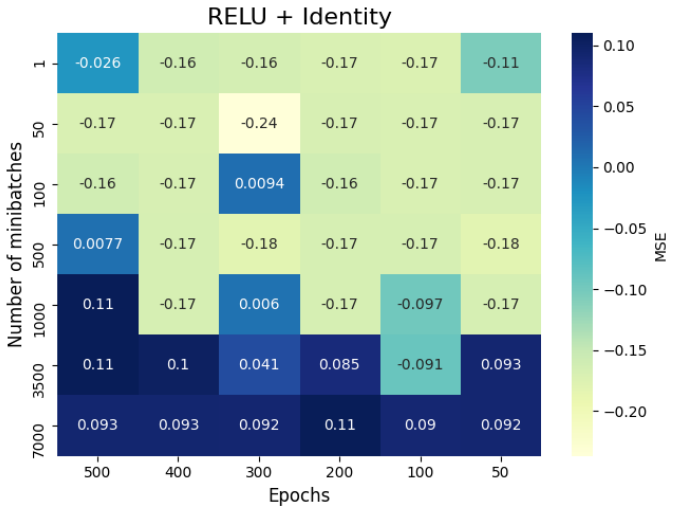

Figure 6. The mean squared error, on a logarithmic scale, for different combinations of epochs and minibatches for the Franke function. The penalty is set to $\lambda = 0$ and the learning rate $\eta = 0.5$. The network uses two hidden layers with 10 neurons and ReLU as activation function, and no activation function on the output layer. The best combination is clearly 300 epochs and 50 minibatches.

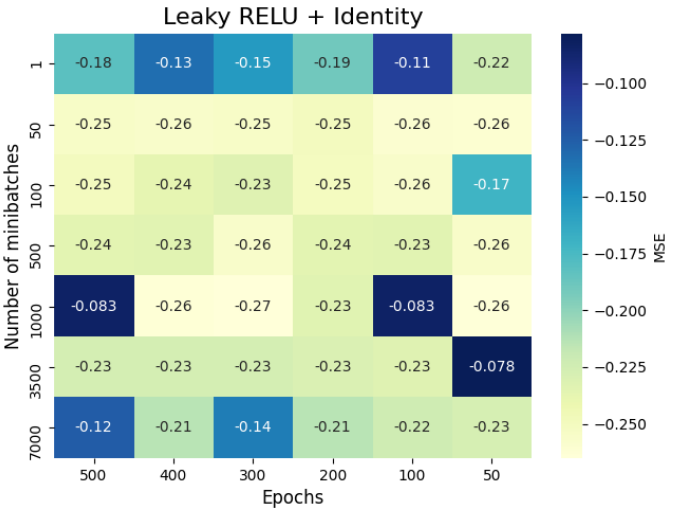

Figure 7. The mean squared error, on a logarithmic scale, for different combinations of epochs and minibatches for the Franke function. The penalty is set to $\lambda = 0$ and the learning rate $\eta = 0.5$. The network uses two hidden layers with 10 neurons and leaky ReLU as activation function, and no activation function on the output layer.

Next, we use the best combination of minibatches and epochs, based on the previous heatmaps, to find the best combination of the learning rate η and the regularization parameter λ. For the network with sigmoid as the activation function on the hidden layers, we continue the analysis with 300 epochs and 100 minibatches, as this results in a low MSE, but still not too computational expensive. With the same arguments, we run the network with ReLU as the activation function on the hidden layers with 300 epochs and 50 minibatches, and the one with leaky ReLU with 100 epochs and 100 minibatches.

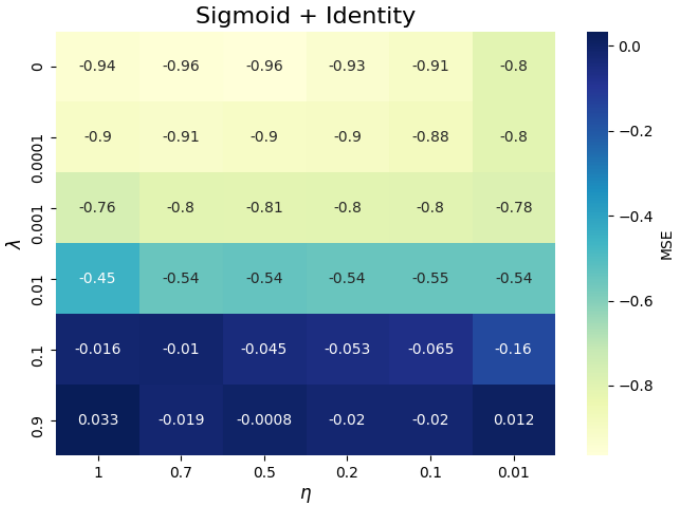

Figure 8. The mean squared error, on a logarithmic scale, for different combinations of η and λ for the Franke function. The network is run with 300 epochs and 100 minibatches. Low or non penalty gives low MSE.

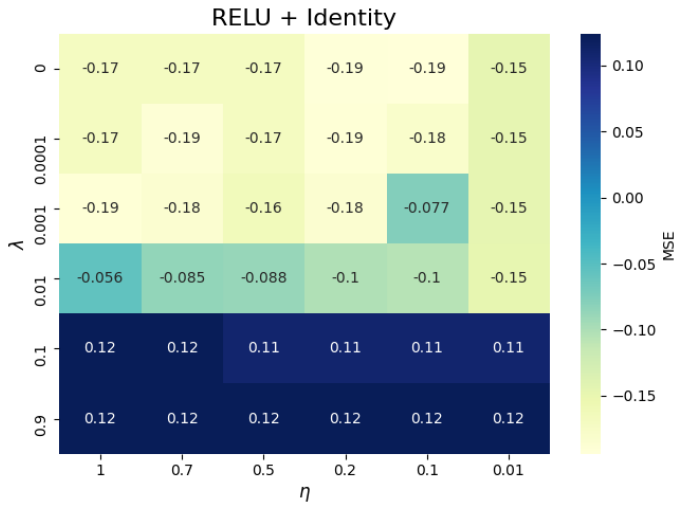

Figure 9. The mean squared error, on a logarithmic scale, for different combinations of η and λ for the Franke function. The network is run with 300 epochs and 50 minibatches. Low or non penalty gives low MSE.

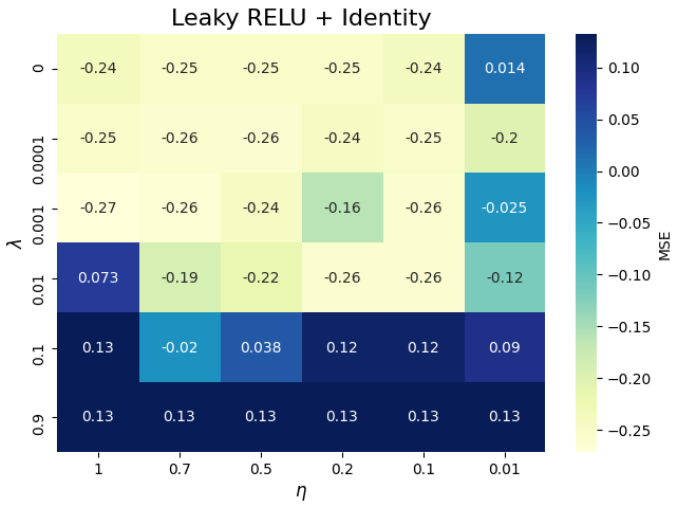

Figure 10. The mean squared error, on a logarithmic scale, for different combinations of η and λ for the Franke function. The network is run with 100 epochs and 100 minibatches. Low or non penalty gives low MSE.

Moving on, we test our network on the MNIST data set of hand-written numbers for both a simple and a more complex neural network. We change the cost function to categorical cross entropy and use softmax as the activation function on the output layer, to ensure we get a probability distribution as output.

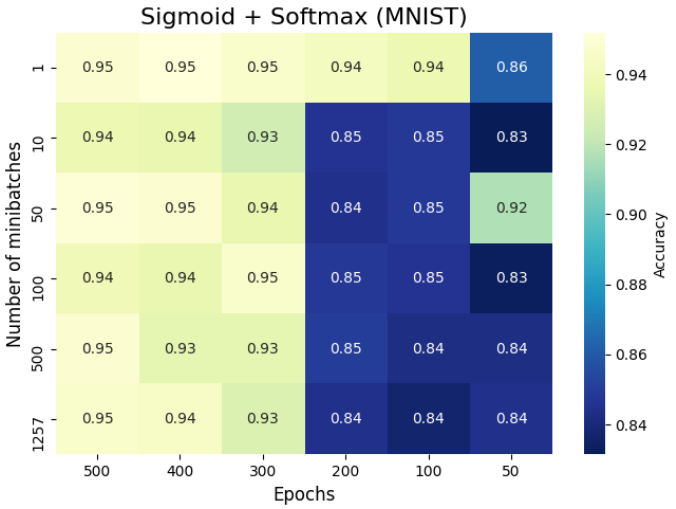

Figure 11. The accuracy score for different combinations of epochs and minibatches for a network with two hidden layers containing 50 neurons each, when $\lambda = 0$ and $\eta = 0.5$. The best accuracy is reached with a high number of epochs.

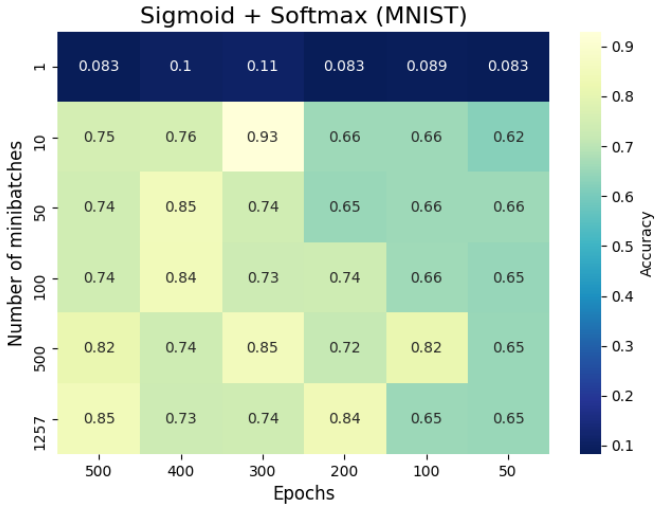

Figure 12. The accuracy score for different combinations of epochs and minibatches for a more complex neural network, when $\lambda = 0$ and $\eta = 0.5$. The network contains five hidden layers with [50, 50, 60, 40, 40] neurons respectively.

By comparing Figure 11 and Figure 12 it is clear that the accuracy is lower in general for a more complex neural network when categorizing the MNIST data. Hence, we continue the analysis with the network with two hidden layers of 50 neurons each and now run with 300 epochs and 100 minibatches.

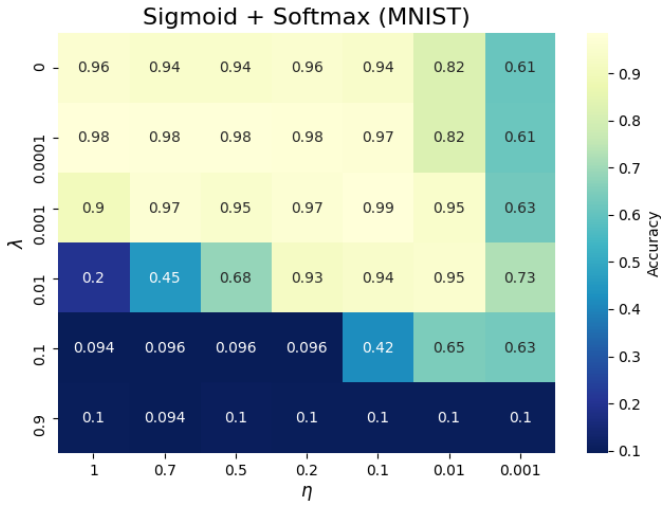

Figure 13. The accuracy score for varying η and λ when using a neural network with two hidden layers, containing 50 neurons each with sigmoid as activation function, and softmax on the output layer. The network is run with 300 epochs and 100 minibatches. The accuracy score is best for low values of λ.

From Figure 13, we observe that for 300 epochs and 100 minibatches, the accuracy is when $\eta = 0.1$ and $\lambda = 0.001$. Based on this result, we create a confusion matrix to see which digits are misclassified.

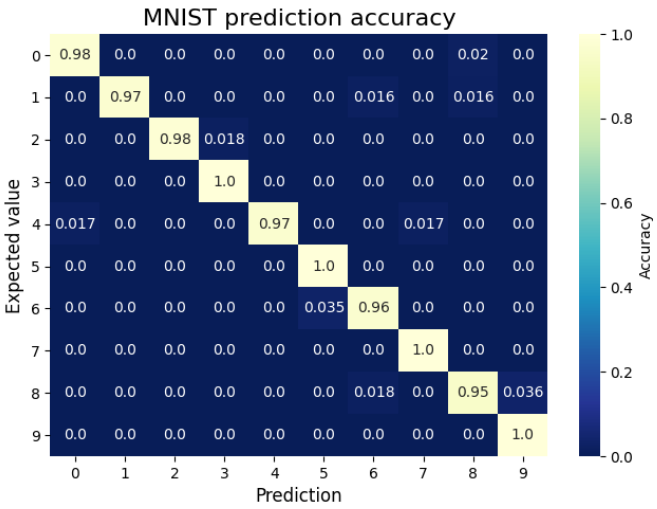

Figure 14. Confusion matrix for the best combination of parameters (300 epochs, 100 minibatches, $\eta = 0.1$, $\lambda = 0.001$) on the MNIST data set using a neural network with two hidden layers of 50 neurons each. In most cases the classification is correct, but e.g. the network misclassifies 6 as 5 and 8 as 9 in 3.5% of the cases.

Last, but not least, we perform a logistic regression on the MNIST data set.

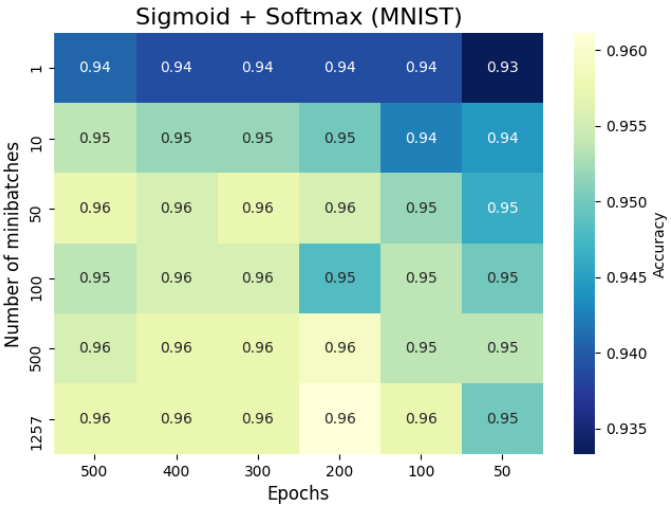

Figure 15. The test data accuracy score for different combinations of epochs and minibatches when using logistic regression on the MNIST data. Here, $\lambda = 0$ and $\eta = 0.1$.

Based on the results in Figure 15, we continue with 200 epochs and 500 minibatches, as this gives a high accuracy score of 96%, but is not as computational expensive as 200 epochs and 1257 minibatches, which corresponds to only one training sample in each minibatch.

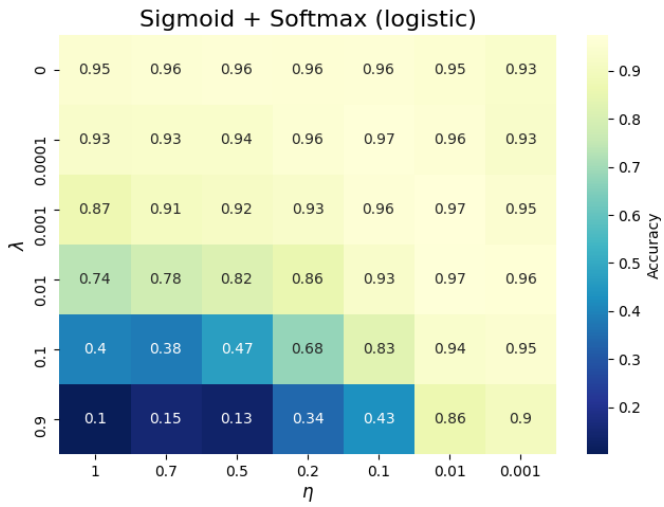

Figure 16. The accuracy score for logistic regression for different combinations of η and λ. The accuracy score is overall good, except when both the learning rate and the penalty is high.

Figure 16 shows that when using $eta = 0.01$ and $\lambda = 0.001$, the accuracy score is 97%. Continuing with the same amount of epochs and minibatches and this choice of learning rate and penalty, we produce a confusion matrix too see which digits get misclassified.

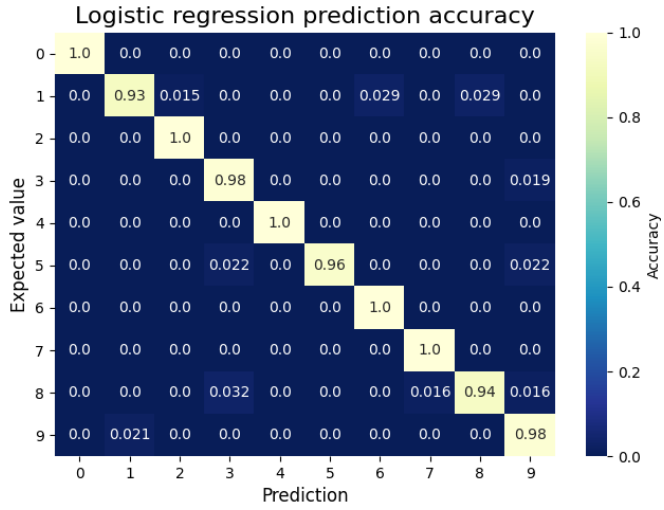

Figure 17. Confusion matrix when using logistic regression on the MNIST dataset of hand-written digits. Here, $n_{epochs} = 200$, $n_{batches} = 100$, $\eta = 0.01$ and $\lambda = 0.001$.

V. DISCUSSION

Stochastic Gradient Decent

By applying stochastic gradient descent on the Franke function we see how much better the analytic solution used in the OLS is compared to the iterative Gradient

descent method 2. Plot 2 also illustrates the effect of learning rates. As can be seen, the learning rate of 0.01 is much too big, as the large spikes in MSE indicates. The method converges towards a local minimum very quickly, but the spikes makes the method much too unstable. On the other hand we also observe that the rate at which the other plots converge is much too slow for learning rates lower than 0.001. A natural conclusion is the use of changing learning rate, which gets lower as the epochs increase. This will also help the method get into very local minimum, where other methods might overstep a deep small well in the cost function space. The methods slowness in convergence can be mitigated with the use of a momentum parameter γ, as we will discuss later.

Figure 4 shows how the method does not benefit from having a large λ factor. As explained in the theory section under MSE II, the regularization parameter λ is useful for preventing overfitting. The reason why this is not a problem, leading to a higher λ giving worse results is the fact that the stochastic element introduced by the mini batches has a similar effect. The mini batch stochasticity prevents from overfitting as it adds a bit of random noise, which leads to a more general model.

The use of a momentum variable γ as stated in the theory, should help the method converge faster by moving faster in directions with a larger gradient. Figure 3 does indeed show this, as the higher γ values quickly converge towards a low MSE. This can in large part mitigate the slowness of the methods shown in fig 2, as the method with $\eta = 0.000001$ had a hard time converging by 2000 epochs seems to be converged by less than 100 in figure 3.

The OLS method applied in Figure 2 and figure 4 is the method from "An introductory study of regression methods with machine learning and its applications" [1] that gave the best results, and as can be seen, it is indeed the better method when compared to the iterative SGD method. It does however have some drawbacks, as it depends on making a design matrix, the size of which is dependent on the number of datapoints and polynomial degree which our neural network does not have to. SGD which is an iterative method is on the other hand completely general, and does not rely on an analytical solution, like OLS and ridge regression. This is the main advantage which makes SGD the clear favorite for complex datasets such as MNIST.

When studying the Franke function, we use the mean squared error as cost function. This cost function tells how close the prediction is to the expected value when approximating a line, an intuitive way to tell how good the results are. MSE is also easy to differentiate, thus it is easy to implement in the network.

Neural Networks on the Franke function

The MSE for the Franke function changes a lot when changing the activation function used on the hidden layers. As seen in Figures 5 to 7, the MSE is clearly best when sigmoid is applied to the hidden layers compared to ReLU and leaky ReLU. Based on the fact that sigmoid is more prone to saturation, and thus vanishing gradient, we expected this to give worse results than both ReLU and leaky ReLU. When the gradient vanishes, the network stops learning, hence leaky ReLU should give the best results as it does not saturate. Even though sigmoid gives the best results, the leaky ReLU performs better than ReLU on the Franke function.

One element to conscider, which in recent years with deep learning has become very important, is that ReLU and leaky ReLU might be better when the weights and biases are initialized differently, like with the Xavier Gloror method. There are reasons to believe both ReLU and leaky ReLU should have the same initialization, hence when we initialize with normal distribution the leaky ReLU still performs better than ReLU, as we can see in Figure 9 and Figure 10. Almost no matter the combination of penalty λ and learning rate η, the network performs better with leaky ReLU than ReLU as the activation function on the hidden layers.

Figure 8 to 10 shows the various activation functions as the parameters λ and η are varied. As can be seen a large regularization parameter λ is rarely beneficial, mostly due to reasons stated previously regarding mini batches introducing stochasticity that prevents overfitting. A small λ is shown to be of slight benefit, as in figure 10 a band of better MSE values are found around $\lambda = 0.001$. It is important to remember that these values do suffer a bit from the same randomness that gives it the benefit of better bias, and these values that vary by so little might have been struck by a bit of luck. The general threnden however is clear, low to no λ, and η greater than 0.01 and you're likely to get a decent result.

Neural Networks on the MNIST dataset

For the MNIST data set of hand-written digits we change the cost function to categorical cross entropy, since we are no longer approximating a function. When predicting the digits, we transform the labels to one-hot vectors and output a probability distribution from the network by using softmax as the activation function on the output layer. This means we are comparing probability distributions, thus the cross entropy is a better choice as cost function.

As shown in figure 11 and fig 12 theres is no benefit

to added complexity. In fact, the resemblance of over-fitting, a constant danger in all machine learning, seems to be what is happening. The best values for a two hidden layer network is used for exploring η and λ in figure 13.
In figure 13 we see how a small lambda of $\lambda = 0.0001$ produces the best results, however the learning rate has little to say when it comes to the final accuracy score. When looking at $\eta = 0.001$ it is safe to assume this method simply has not had time to reach the minimum, or is unable to get out of a small local minima.

The choice of number of Neurons in each layer is something we found to not be of a major importance. When using reasonable values for the other parameters, a difference in 50 and 100 neurons in a two layer network was not enough to differentiate between these effects and random effects.

Figure 14 shows lastly the confusion matrix of the different prediction and expected values for the best parameters from figure 13. It is not surprising that integers such as 6, 8 and 5 get confused, as these often have a close resemblance when written by hand, which is exactly what we see in this model. A total accuracy of 0.98 is achieved.

Logistic regression

Logistic regression is, as we have discussed, a very shallow neural network with no hidden layers. Despite the simplicity of the model we suspect this method will work well none the less, which is also confirmed in figure 15 to 17.
A clear preference to a large amount of batches is shown in figure 15, where the best values are found to be for as many batches as datapoints. This does negate some of the positive effects of mini batches, but clearly if one has the computing power, and time, it seems to be beneficial in this special case. With that said the varaince of values for any combination of batches and epochs is very small when compared to the previous methods.
Figure 16 shows a quite clear and nice tendency towards low η and low λ. The best values however are found in a diagonal band towards the lower end of λ and η. Again, the best parameters are used in 17 where we find a total accuracy of 0.97.

The reason for why such a simple model performs as well as our two hidden layer Neural network has to do with the complexity of data. Here we had the relatively simple case of black integers written on a white background, and no rotation or similar differences in the images, things that would right away make a more complex dataset and the results with the logistic regression worse.

VI. CONCLUSION

For stochastic gradient descent we found OLS to be far better than SGD. This is due to the fact that OLS uses an analytical solution to minimize the cost function, meaning it is not a general case solution like SGD, meaning it is not applicable to the MNIST dataset. Despite this we found convergence towards a fair MSE for SGD, which was sped up greatly by the momentum parameter. Due to the specifics of our data we had little benefit from λ, apart from some cases using the Neural network. We also found that paramater tweaking and adjusting was paramount for a good solution.
SGD is as mentioned rarely used for function regression meaning our result is not surprising, yet it did bridge the gap from regression to Neural networks.

For classification we found that a complex netowrk was not necessary, as we found when comparing two and five layer networks. This is also found to be the case as we were able to produce very good results with the shallow logistical regression method.

The use of ReLU and leaky-ReLU was surprisingly worse than the sigmoid function. This can be attributed to the fact that we only initialized the weights with random values about zero. This has been shown in papers to be an antiquated method, that especially for deep networks with ReLU is not sufficient, and often leads to exploding or vanishing gradients [5]. We did however find that leaky-ReLU was better than the basic ReLU, which fits with what we expected.

In conclusion we found that a shallow Neural network with two hidden layers with 50 neurson each, $\eta = 0.1$, $\lambda = 0.001$, 300 epochs and 100 mini batches worked best, and produced an accuracy of 0.98.
The logistical regression did also perform very well, achieving an accuracy of 0.97, however it is unknown how well this method does with more complex datasets. Our Neural network is very easy to scale to more complex models by adding layers and tweaking parameters. But for a dataset like MNIST it is clear that logistical regression is sufficient.

Further work to consider in the future is the initialization of the weight and bias vectors. In this paper we stuck to initializing these with small values of uniform or normal distribution however this has been shown to be an antiquated way of choosing the very important starting point [5]. A too large or too small initialization of weights can lead to vanishing or exploding gradients, especially when using activation functions such as Sigmoid or ReLU.
We did also not include any data on the choice of number of neurons per layer. This is also something to look into, as the effects of this is relatively unknown. A future paper building on our results should indeed look into this.

The use of a varying learning rate in this paper is also fairly limited. This should in theory produce a better MSE, as it would allow for the method to reach lower levels of MSE, but the time and effort it takes to get this just right meant that we did not take the time to do so.

Appendix A: Code for reproduction

For code used in this project see https://github.com/elinfi/FYS-STK4155/tree/master/Project2.

Appendix B: deriving the derivative of the MSE ridge cost funciton

$$\begin{aligned} C(w) =&(Xw-z)^T(Xw-z)+\lambda w^T w \\ =&w^T X^T X w - 2w^T X^T z + z^T z + \lambda w^T w \end{aligned}$$

$$\begin{aligned} \nabla C(w) =&2X^T X w - 2X^T z + 2\lambda w \\ =&2X^T(X^T w - z) + 2\lambda w \end{aligned}$$

[1] E. Finstad and A. Bråte, "An introductory study of regression methods with machine learning and its applications," 2020.

[2] http://yann.lecun.com/exdb/mnist/.

[3] M. Hjorth-Jensen, "Week 41 tensor flow and deep learning, convolutional neural networks." https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html, 2020. Accessed: 07-11-2020.

[4] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, p. 1–124, May 2019.

[5] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[6] M. A. Nielsen, "Neural networks and deep learning," 2018.

[7] A. Solberg, "denseneuralnetworkclassifiers." https://www.uio.no/studier/emner/matnat/ifi/IN5400/v20/material/lectureslides/in5400_lecture3_nnet_290120.pdf, 2020. Accessed: 10-11-2020.

[8] F. Chollet *et al.*, "Keras," 2015.