Classification and Regression, from linear and logistic regression to neural networks

Daniel Pinjusic and Gudmund Gunnarsen

November 16, 2020

Abstract

In this project we look at the performance of Feed Forward Neural Networks and Linear and Logistic Regression using Stochastic Gradient Descent. We find that Stochastic Gradient Descent methods are inferior in both accuracy and time expenditure than analytic solutions of OLS and Ridge regression. Logistic Regression and FFNN appear to be roughly equal when applied to the MNIST data set of handwritten digits. More testing is necessary to draw further conclusions on neural network performance relative to regression methods, but we have learned much of the fundamentals of FFNNs and classification problems in general.

1 Introduction

Many of the problems faced in the modern world require at the base a level of human interaction. For instance, recognition of faces, text or speech are all problems that human beings solve quite trivially, while a computer utilizing a program to automatically recognize these elements would be difficult to create. This is why Neural Networks are of interest, a type of program that allows the program itself to modify values and fine-tune its own algorithm. Neural Networks are in a sense based on the human brain - they mimic neurons in some sense, and thus are as expected quite competent at tasks that human beings can solve, but on a larger scale.

In this project, we write a neural network to classify handwritten digits from the MNIST [4] data set. We compare the neural network and its capabilities with that of a logistic regression algorithm that we also write, as well as the effectiveness of the stochastic gradient descent method for finding minima of cost functions, using linear regression. The purpose is to study and understand the differences and advantages of the different methods, and to develop our own code so as to be able to produce more complicated applied solutions to future problems using these methods.

First we will present some basic theory around Neural Networks, and logistic regression. Then we explain the methods and algorithms behind stochastic gradient descent and neural networks. Finally we present and discuss various comparisons between the neural network, logistic and linear regression models.

2 Theory

2.1 Neural Networks

The main idea of a neural network is to simulate a biological system, where neurons interact by sending signals between each other. In the human brain, there are billions of such interconnected neurons communicating by sending electrical signals to each other. Each neuron stores these signals as it gets them, and if the voltage over a neuron changes enough over a short interval of time, it will send out a signal of its own. If the accumulated voltage is not enough, the neuron will remain inactive and have zero output.

In a neural network, the electrical signals are made up of mathematical functions that are passed between layers of neurons where each layer can contain an arbitrary number of neurons. Every neuron i has a connection transferring its output to the input of some other neuron j in the network, and this connection has an assigned weight variable w_{ij}. If we define $x_i, \dots, x_n$ as the signals received by a neuron, and n as the number of other neurons, then we can create a model for the output of a single artificial neuron as

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u), \tag{1}$$

where $f(u)$ is the activation function.

There are many types of neural networks, but we will focus on a so called Feed-forward neural network(FFNN) that is fully connected, meaning that our network will have an input layer, one or more hidden layers and an output layer, where all the neurons have a connection to every other neuron in the network, but there is no direct connection between any neuron in the same layer. Such a network is also often called a multilayer perceptron(MPL).

2.2 Feed-forward

A feed forward neural network is a neural network where there is no cycle or loop between the connections of the neurons. All information moves in one direction, forward, starting with in the input layer, moving through any the hidden layers and ending at the output layer.

We define the output y for each neuron in the network as

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z), \tag{2}$$

where b_i is an added bias to each neuron and the inputs x_i are the outputs from the neurons in the preceding layer. The bias is added such that one can control the chance of a neuron being activated.

The model works by taking in a number of inputs x_i in the first layer. Then, it calculates the activation $z_i^1 = \sum_{j=1}^{M} \left(w_{ij}^1 x_j + b_i^1\right)$ for the first hidden layer, where M is all possible inputs to a given neuron i in the first layer. When z_i^1 is know, it is possible to calculate the output for the first hidden layer as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right). \tag{3}$$

Note that we are here assuming that each layer has the same activation function f, but it is not unusual for each hidden layer to have a different activation function. If we take this into account, and generalise Eq (3) for all layers, we get

$$y_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right), \tag{4}$$

where N_l is the number of neurons in each layer l and $y_j^0 = x_j$.

2.3 Universal approximation theorem

The universal approximation theorem states that standard multi-layer feed-forward networks, with as few as one hidden layer, are capable of approximating any continuous function with arbitrary precision. [6]. This is important, as it tells us that our FFNN will be able to approximate any continuous function.

2.4 Activation functions

An activation function defines the output of a neuron given an input or a set of inputs. Choosing an activation function that is non-linear for the hidden layers of a FFNN will make that FFNN conform to the universal approximation theorem. Some typical examples of non-linear activation functions include the logistic Sigmoid function

$$f_S(x) = \frac{1}{1+e^{-x}} \tag{5}$$

$$\frac{\partial f_S(x)}{\partial x} = f_S(x)(1-f_S(x)), \tag{6}$$

the hyperbolic tangent function

$$f_H(x) = \tanh x \tag{7}$$

$$\frac{\partial f_H(x)}{\partial x} = \frac{f_H(x)^2}{\sinh^2 x}, \tag{8}$$

The Rectified linear unit(ReLU) function

$$f_{ReLU}(x) = max\{0, x\} \tag{9}$$

$$\frac{\partial f_{ReLU}(x)}{\partial x} = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \\ \text{undefined} & x = 0 \end{cases} \tag{10}$$

and the leaky ReLU function

$$f_{leakyReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0.01x, & x < 0 \end{cases} \tag{11}$$

$$\frac{\partial f_{leakyReLU}(x)}{\partial x} = \begin{cases} 1, & x \geq 0 \\ 0.01, & x < 0. \end{cases} \tag{12}$$

In the case of a multi layer classification problem, on can use a function such as the softmax function in the final layer, which returns the probability of a guess being correct.

2.5 Cost functions

Cost functions help measured the performance of a model for a given set of data. It quantifies the error between the models predicted values and expected values in the form of some real number. Cost functions are usually minimized, such that one can find the parameters that give the most accurate prediction and thereby decreasing the return of the cost function as much as possible. There exist several different cost functions, and which one works best is dependent on what problem one is trying to solve. In our case, we will be doing both regression and classification, and will therefore choose cost functions that work for either regression or classification. In the case of regression, we use the quadratic cost function

$$\mathcal{C}(W^L) = \frac{1}{2}\sum_{i=1}^{n}(a_i^L - t_i)^2, \tag{13}$$

where t_i are the targets we want to reproduce and a_i^L are the outputs from the network after propagating through all inputs $\hat{x}$. This function has the derivative

$$\frac{\partial \mathcal{C}(W^L)}{\partial a_i^L} = (a_i^L - t_i). \tag{14}$$

For classification problems, we use the negative log likelihood function

$$\mathcal{C}(W^L) = -\sum_{i=1}^{n}(t_i \log a_i^L). \tag{15}$$

2.6 Regularization

It is common to add a term to the cost function, to constrain the size of the weights so that they do not grow out of control. This means that the weights can not grow arbitrarily large to fit the training data, and this will reduce overfitting. We will be adding the L2 norm $\lambda \sum_{ij} w_{ij}^2$ to our cost functions, where λ is the regularization parameter.

2.7 Backpropagation

The goal of backpropagation is to optimize the values for the weights and biases so that we can minimize the cost function, and thereby obtain the best prediciton. Since the cost function is just an average over all the losses, we can do a partial derivative of a single sample from the output layer, and then work our way backwards through the network.

First, let us define the activation z_j^l of neuron j in the l'th layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l, \tag{16}$$

where we define $a_i^{l-1} = f^{l-1}(z_j^{l-1})$ as the outputs from the previous layer, b_j^l as the biases from layer l and M_{l-1} as the total number of neurons in layer $l-1$.

We then want to find the partial derivative w.r.t the weights and biases of the cost function in the output layer L,

$$\frac{\partial \mathcal{C}(W)}{\partial w_{jk}^L} \quad \text{and} \quad \frac{\partial \mathcal{C}(\hat{W})}{\partial b_k^L}. \tag{17}$$

The partial derivative of the cost function in output layer L with with respect to the weights w_{jk} can be found as

$$\frac{\partial \mathcal{C}(W^L)}{\partial w_{jk}^L} = \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} \frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} \left(\frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} \right) \tag{18}$$

The term $\frac{\partial z_j^L}{\partial w_{jk}^L}$ is not dependent on the activation function, so can be determined for a general case as

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1} \tag{19}$$

The other two terms are dependent of our choice of activation function and cost function. So far, we have determined that the partial derivative of the cost function with respect to the weights is

$$\frac{\partial \mathcal{C}(W^L)}{\partial w_{jk}^L} = \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} \left(a_k^{(L-1)} \frac{\partial a_j^L}{\partial z_j^L} \right), \tag{20}$$

were we now define the output error δ_j^L as

$$\delta_j^L = \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \tag{21}$$

In the expression for the output error, the first term measures how fast the cost function is changing as a function of the j'th output activation. When the cost function has a small dependency to a neuron output, δ_j^L will be small. The second term measures how fast the activation function f changes for a given z_j^L.

Now that the output error δ_j^L has been defined, we can rewrite the (20) in a more compact form as

$$\frac{\partial \mathcal{C}(W^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{(L-1)} \tag{22}$$

This is the first equation of back propagation, starting at the output layer.

If we apply the chain rule on (23), we find

$$\delta_j^L = \frac{\partial \mathcal{C}(W^L)}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}(W^L)}{\partial z_j^L} = \frac{\partial \mathcal{C}(W^L)}{\partial b_j^L} \frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}(W^L)}{\partial b_j^L}. \tag{23}$$

The term $\frac{\partial b_j^L}{\partial z_j^L}$ simplifies to 1 and we have thus found expressions for the partial derivative w.r.t the weights and biases of the cost function. This also tells us that δ_j^L is equal to the exact rate of change of the cost function with respect to the bias.

Next, we want to find the derivatives of the cost function for a general layer l. As seen in (23), it is possible to define the error as

$$\delta_j^l = \frac{\partial \mathcal{C}(W^l)}{\partial z_j^l}. \tag{24}$$

This equation holds for all layers in the network. Since we are working from the output layer to the input layer, we wish to relate the error of layer l to the already known error in layer $l+1$. As such, starting from (24), we find, using the chain rule that

$$\delta_j^l = \sum_k \frac{\partial \mathcal{C}(W^l)}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}. \tag{25}$$

The second factor simplifies to

$$\begin{aligned} \frac{\partial z_k^{l+1}}{\partial z_j^l} &= \frac{\partial}{\partial z_j^l}\left(\sum_{i=1}^{M_l} w_{ik}^{l+1} a_k^l + b_k^{l+1}\right) \\ &= w_{ik}^{l+1} \frac{\partial a_k^l}{\partial z_j^l}, \end{aligned} \tag{26}$$

which when inserted into (25) gives

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \frac{\partial a_k^l}{\partial z_j^l}. \tag{27}$$

We now have the equations needed to set up the back-propagation algorithm for any cost and activation function. The final step then involves updating the weights and biases as

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1} \tag{28}$$
$$b_j^l \leftarrow= b_j^l - \eta \delta_j^l \tag{29}$$

using some method of gradient descent, where η is the learning parameter.

2.8 Logistic Regression

An alternative to using a Neural Network for problems of classification is to use logistic regression. In logistic regression, we model the likelihood of a system to be in a given class using the Softmax function, defined as

$$\begin{aligned} P(C=k|N=i|\boldsymbol{\beta}) &= \frac{\exp(\mathbf{X}\boldsymbol{\beta})_{i,k}}{1+\exp(\sum_{k'}(\mathbf{X}\boldsymbol{\beta})_{i,k'})} \\ P(C=K|N=i|\boldsymbol{\beta}) &= \frac{1}{1+\exp(\sum_{k'}(\mathbf{X}\boldsymbol{\beta})_{i,k'})} \end{aligned} \tag{30}$$

where $P(C=k|N=i|\boldsymbol{\beta})$ for logistic regression refers to the probability that the data sample $N=i$ belongs to the class $C=k$ according to the model. $\boldsymbol{\beta}$ are the parameters of the model and $\mathbf{X}$ is the matrix of predictors for each point. Note that for multiple classes, $\boldsymbol{\beta}$ is a $p \times (c-1)$ matrix, for p total predictors and c classes. Since this function represents a probability, the parameters for the final class $C=K$ are set by the first $c-1$ parameters.

In logistic regression, the cost function is defined using the Maximum Likelihood Estimation principle, or MLE. In this principle, the product of the probabilities of the model producing the correct results are all multiplied together. Taking the negative logarithm of this expression yields the cost function itself,

$$\begin{aligned} C(\boldsymbol{\beta}) &= -\log \prod_{i,k} P(C=k|N=i|\boldsymbol{\beta})^{y_{i,k}} \\ &= -\sum_{i,k} y_{i,k} \log P(C=k|N=i|\boldsymbol{\beta}), \end{aligned} \tag{31}$$

where $y_{i,k}$ is an element of the classification matrix of samples $\boldsymbol{y}$, equal to 1 for exactly one class for each sample. Thus the product in equation (31) picks out only the probabilities that the model classification matches the actual classification in the training data, and minimizing the negative logarithm will maximize the likelihood that the model guesses correctly in all of the samples.

Using the Softmax function in equation (30) for the probability as we do in a classification problem with multiple classes yields the specific cost function

$$C(\boldsymbol{\beta}) = -\sum_{i,k} y_{i,k} \left((\mathbf{X}\boldsymbol{\beta})_{i,k}(1-\delta_{k,K}) - \log\left(1 + \sum_{k'} \exp(\mathbf{X}\boldsymbol{\beta})_{i,k'}\right)\right). \tag{32}$$

We omit the dependencies on $\mathbf{X}$ and $\boldsymbol{y}$ in the arguments of C, as the parameters $\boldsymbol{\beta}$ are the important values for the solution of this problem. Since this function is too difficult to minimize analytically, we need a method for numerically minimizing functions. The Stochastic Gradient Descent method that we use requires the gradient of the cost function, which in the logistic regression case is

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\mathbf{X}^T(\boldsymbol{y} - \boldsymbol{p}), \tag{33}$$

where $\boldsymbol{p}$ is the $n \times c$ matrix of probabilities defined by the softmax function in equation (30), with n being the total number of data samples.

2.9 Linear Regression

We shall also need the gradient of the cost function used in linear regression. We omit deriving them but refer to the lecture notes [3]. The cost function is the Mean Squared Error, which is

$$C(\boldsymbol{\beta}) = \frac{(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})}{n} + \lambda \boldsymbol{\beta}^T\boldsymbol{\beta} \tag{34}$$

for Ridge regression, returning to Ordinary Least Squares for the hyperparameter $\lambda = 0$. The gradient of the cost function for Ridge regression is

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}) + 2\lambda|\boldsymbol{\beta}|. \tag{35}$$

With these gradients we are ready to use Stochastic Gradient Descent to solve the regression problems.

3 Method

3.1 Stochastic Gradient Descent

Neural networks as well as linear and logistic regression all require the minimization of some form of Cost/Loss function. While linear regression has analytic expressions for the minima of some cost functions, such as for OLS and Ridge regression, there is a need for a reliable, general method for finding minima of arbitrary functions. Gradient Descent methods were first studied in 1847 by Cauchy, but a more detailed look at Steepest Descent methods for nonlinear minimization problems was done by Haskell Curry in 1944 [1]. One of the most common methods in modern times, especially with the large scale parameter spaces required by Neural Networks is Stochastic Gradient Descent (SGD)[5]. This numeric method is essentially a stochastic approach to the steepest descent method. It involves starting at a randomized point in parameter space, finding the gradient of the Cost function $C(X, Y, \boldsymbol{\beta})$ of the parameters $\boldsymbol{\beta}$ to be minimized, or other function to be minimized at that point. The variables X, Y are generally predictors and data points, respectively, in regression methods, but can in principle be any variables associated with a data set. Once the gradient is found, the algorithm moves a small step in the opposite direction in parameter space. This will eventually lead to some form of minimum of C.

Since C might have local minima that are not of interest, SGD separates the search for a minimum into multiple steps: By splitting the dataset (X, Y) into minibatches and then calculating in each step the gradient of C relating only to that particular minibatch, local minima can be smoothed out, and the time required to calculate C and gradient for each minibatch can be greatly reduced for complicated models. When all the minibatches have had their gradients calculated, the algorithm has completed one epoch, and the dataset is randomly distributed once more into minibatches until a given number of epochs have passed. Finally, the learning rate η of the algorithm determines how far in the direction opposite the gradient in parameter space the algorithm moves per step. Ideally it is modified to be small when the gradient is steep, and large when the gradient is shallow, to maximize efficiency and avoid missteps. After a number of epochs, C is assessed at the located minimum, and the whole process repeats with a new randomized starting location, comparing the minimum of each iteration to increase the probability of finding the desired global minimum.

Algorithm 1: Stochastic Gradient Descent

```
Result: Coordinates $\boldsymbol{\beta}$ in parameter space of the minimum of the cost function
initialize $error$;
for $iterations$ do
    for $epochs$ do
        Shuffle $X, Y$;
        Create $M$ minibatches $m_i$ of $X, Y$;
        for $m_i$ $in$ $M$ do
            $\eta = \frac{t_0}{1+\frac{epoch*M+i}{t_1}}$;
            $\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} + \eta \nabla C(\boldsymbol{\beta}_{old})_m$;
        end
    end
    if $C(\boldsymbol{\beta}_{new}) < error$ then
        $\boldsymbol{\beta}_{best} = \boldsymbol{\beta}_{new}$;
        $error = C(X, Y, \boldsymbol{\beta}_{best})$;
    end
end
```

In the SGD algorithm used in this project, we use a decaying learning rate defined as

$$\eta = \frac{t_0}{1 + \frac{t}{t_1}},$$

where t_0, t_1 are parameters to be chosen, while t corresponds to the total number of steps the algorithm has taken in parameter space. Furthermore, since the gradient descent algorithm performs a number of steps proportional to the number of minibatches M, η is scaled by a factor $\frac{1}{M}$. This means that, setting aside the effects of learning rate decay, the total distance traversed in parameter space should remain similar for large and small minibatches.

3.2 Network model

For the classification case, we choose a network with one hidden layer, consisting of 50 neurons and use the sigmoid function as the activation. The input will consist of 64 neurons, as the images in the MNSIT data set have $8x8 = 64$ pixels or features. The output layer will consist of 10 neurons, as in this case we are doing multi-class classification of a set of numbers from $1 \ldots 9$. As such, it is natural to split the outputs into 10, as this will tell us the probability of the number in any image being a number in the set $1 \ldots 9$.

For the regression case, we choose a polynomial degree of $p = 5$ giving us 21 input neurons. We choose one hidden layers with 5 neurons. For the output layer, we choose a single neuron and no activation function.

For both regression and classification, the weights are randomly sampled from a uniform distribution, and the biases are just the singular value 0.01 for every neuron. We choose this value, or any non zero value for the bias because a bias with value 0 could cause the gradients to vanish during back propagation and training of the network. A low value for the bias helps ensure that back propagation and forward propagation do not come to a sudden stop because of the vanishing gradient. We also do the adjustments of our hyperparameters using an epochs = 10 and a batch-size of 100.

Before we start training our model, we want to make it as efficient as we can. When working with computers, it is often beneficial to vectorize equations for optimal run time. In this case, that means we rewrite the feed-forward equation (4) as

$$\hat{a}^l = f^l(\hat{z}^l) = f^l(\hat{W}^l \hat{a}^{l-1} + \hat{b}^l), \tag{36}$$

and the back propagation equations (23), (22) and (27) as

$$\delta_j^L = \frac{\partial \mathcal{C}(\hat{W}^L)}{\partial \hat{a}^L} \frac{\partial \hat{a}^L}{\partial \hat{z}^L}, \tag{37}$$

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial \hat{W}^L} = \delta_j^L \hat{a}^{(L-1)}, \tag{38}$$

$$\delta_j^l = \delta^{l+1} \hat{W}_{l+1}^T \circ \frac{\partial \hat{a}^l}{\partial \hat{z}^l}, \tag{39}$$

respectively, where $A \circ B$ signifies the element-wise multiplication of A and B.

Algorithm 2: Feed forward

```
$\hat{z}^1 = \hat{W}^1 \hat{X} + \hat{b}^1$;
$\hat{a}^1 = f^1(\hat{z}^1)$;
for hidden layers do
    $\hat{z}^l = \hat{W}^l \hat{a}^{l-1} + \hat{b}^l$;
    $\hat{a}^l = f^l(\hat{z}^l)$ ;
end
$\hat{z}^L = \hat{W}^L \hat{a}^{L-1} + \hat{b}^L$;
$\hat{a}^L = f^L(\hat{z}^L)$;
```

After the feed forward, run the back propagation, and update the weights and biases.

Algorithm 3: Back-propagation

```
Calculate the last layer error $\delta_j^L$;
Find $\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial \hat{W}^L} = \delta^L \hat{a}^{(L-1)}$;
$\delta^l = \delta^{l+1} \hat{W}_{l+1}^T \circ \frac{\partial \hat{a}^l}{\partial \hat{z}^l}$;
for layers do
    $\delta^l = \delta^{l+1} \hat{W}_{l+1}^T \circ \frac{\partial \hat{a}^l}{\partial \hat{z}^l}$;
    Update weights and biases: ;
    $\hat{W}_l -= \eta \delta_j^l \hat{a}^{l-1}$ ;
    $\hat{b}^l -= \eta \delta_j^l$ ;
end
```

4 Results

4.1 Function Fitting

We apply Ordinary Least Squares and Ridge regression on the Franke function [2] and compare the performance of the analytic minima of the cost function against the ones found through SGD. In figure figure 1 the MSE and R2 score for OLS on the Franke function are plotted against different numbers of minibatches, and compared with the analytic values. The minibatch sizes are chosen to be $1, 2, 4, \dots, 2^n$ such that $2^i \leq N$, for N total points. We also train our neural network on the Franke function, comparing several activation functions with several hyperparameters.

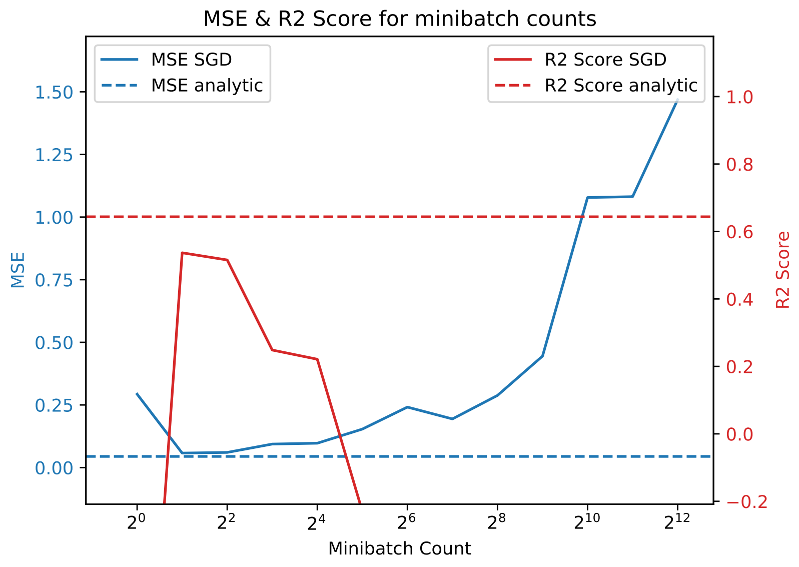


Figure 1: SGD applied to OLS on the Franke function with various numbers of minibatches, compared with the analytic values of OLS. We see that the optimal number of minibatches appears to be 2. The total number of data points used was $2^{12} = 4096$, such that the rightmost data point corresponds to the case where each minibatch contains only a single data point.

In figure 2 we plot the MSE for a decaying learning rate, with the decay controlled as $\eta(t) = \frac{t_0}{1+\frac{t}{t_1}}$, where t is the number of minibatch iterations in the gradient descent algorithm.

We tested also SGD on Ridge regression using the Franke function, shown in figure 3. The SGD determined minima show the same behaviour as the analytic ones, but shifted to a higher degree of inaccuracy.

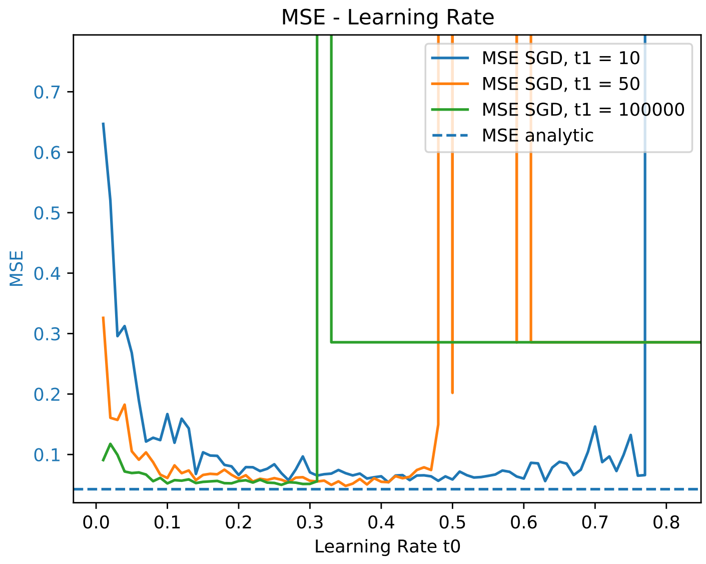


Figure 2: SGD applied to OLS on the Franke function, compared with the analytical values of OLS. The learning rate decay is controlled by $t1$, while the learning rate is proportional to $t0$. We see that the MSE suddenly diverges and grows unstable above certain learning rates, dependant on the decay. With higher decay, the gradient descent algorithm has a larger range of functional learning rate values before the error diverges.

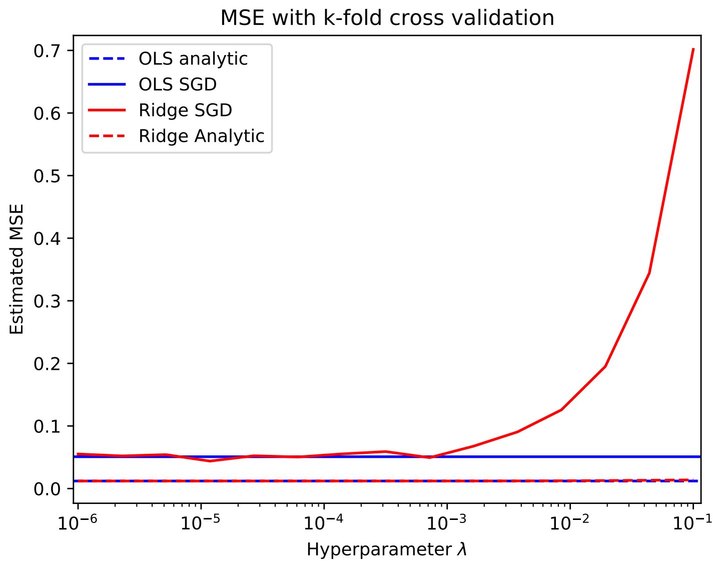


Figure 3: SGD applied to OLS and Ridge regression on the Franke function, compared with the analytical values of OLS and Ridge regression. Kfold cross-validation is used to estimate the dependence on the hyper-parameter λ.

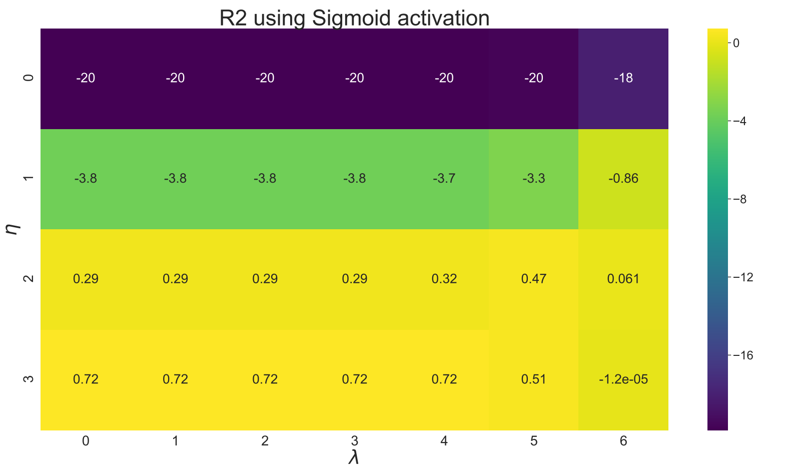


Figure 4: R2 score of Neural network with 1 hidden layer and 1 neuron trained on the Franke function with sigmoid activation in the hidden layer.

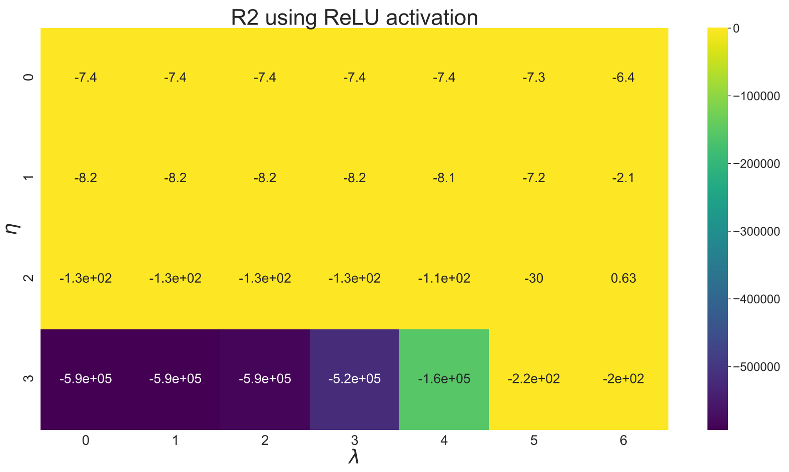


Figure 5: R2 score of Neural network with 1 hidden layer and 1 neuron trained on the Franke function with ReLU activation in the hidden layer.

4.2 Classification

We created our own code for logistic regression and applied it to the MNIST data set [4] of hand-written digits. In figure figure 8 we show the confusion matrix of the logistic model and the neural network model, both with $n = 360$ digits in the test set, trained on 1437 other digits.

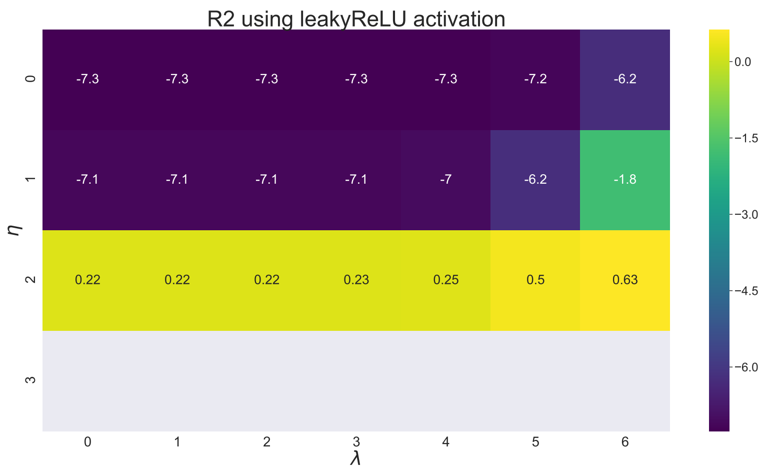


Figure 6: R2 score of Neural network with 1 hidden layer and 1 neuron trained on the Franke function with leaky ReLU activation in the hidden layer.

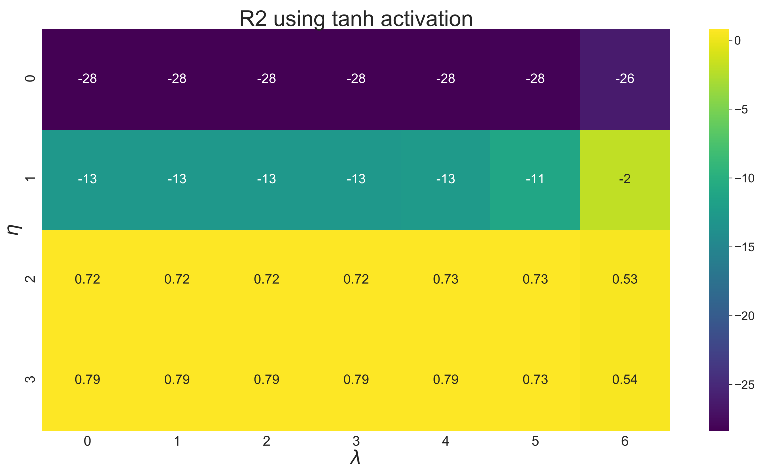
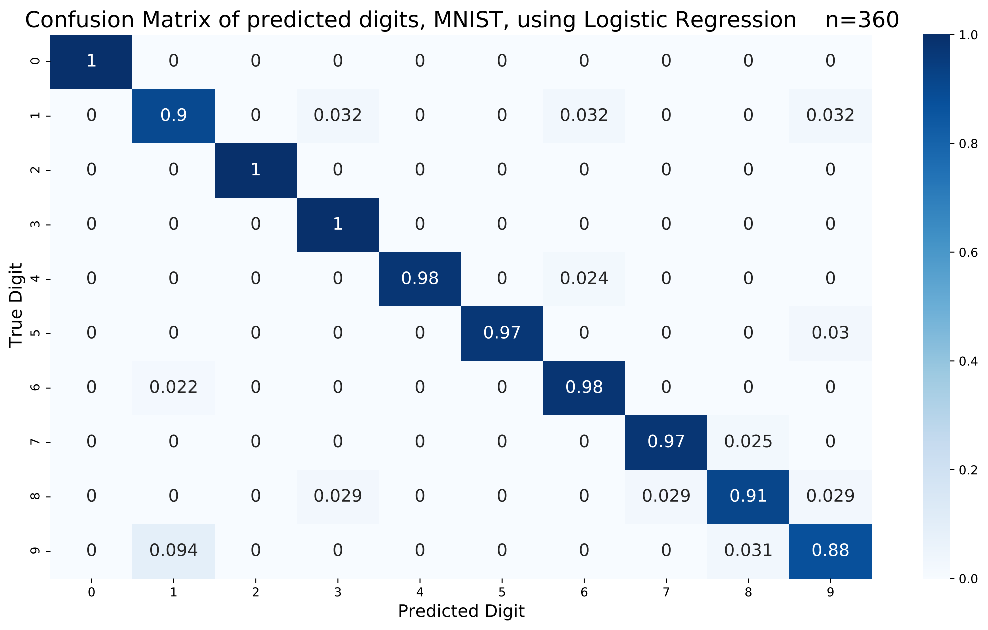


Figure 7: R2 score of Neural network with 1 hidden layer and 1 neuron trained on the Franke function with hyperbolic tangent activation in the hidden layer.

To find optimal learning rates for logistic regression we tested for various learning rates, shown in figure 9.



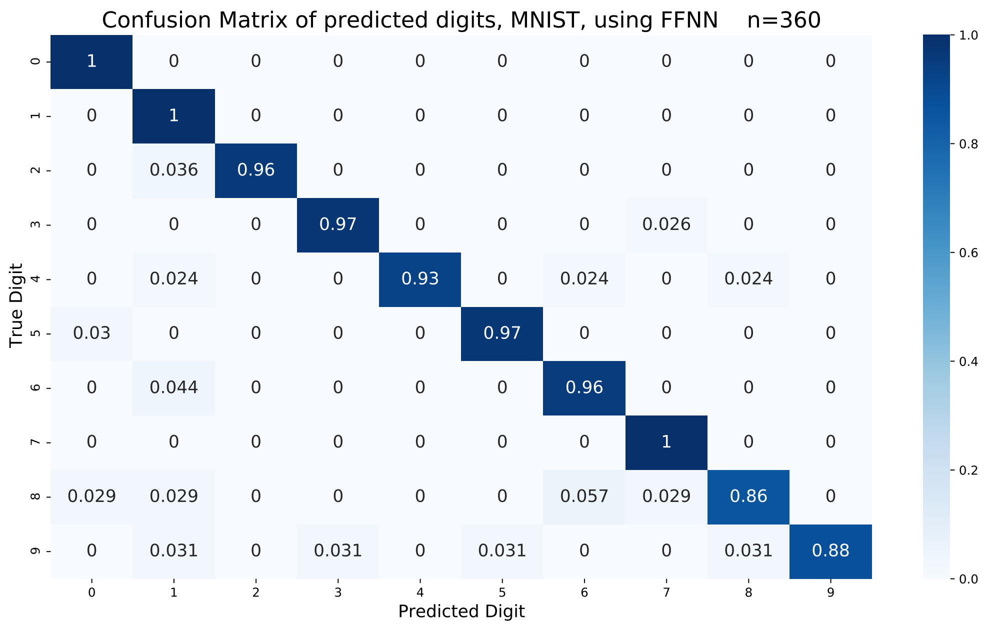


Figure 8: Confusion matrices of predicted digits in the MNIST data set, using logistic regression (above) and FFNN (below). There are 360 digits in the test set used for this matrix.

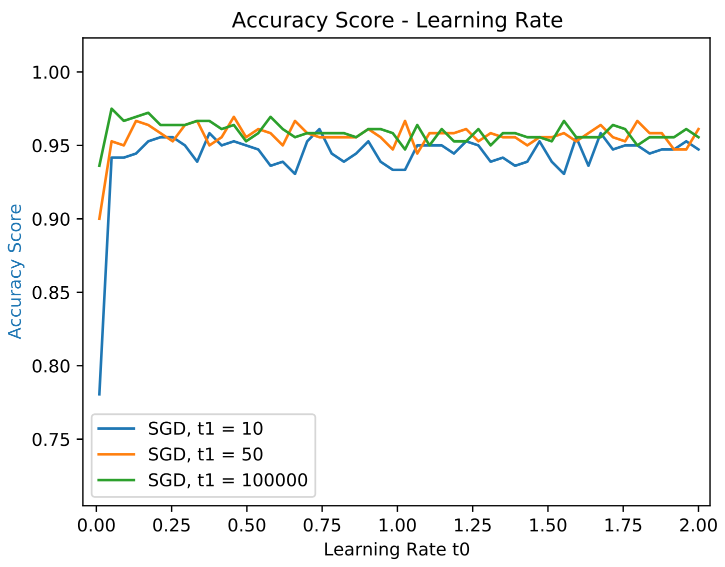


Figure 9: Accuracy score plotted for logistic regression on the MNIST data set, as a function of learning rate, for various learning rate decay scaling. The minibatch size is 359, which is one quarter of the training set size rounding down.

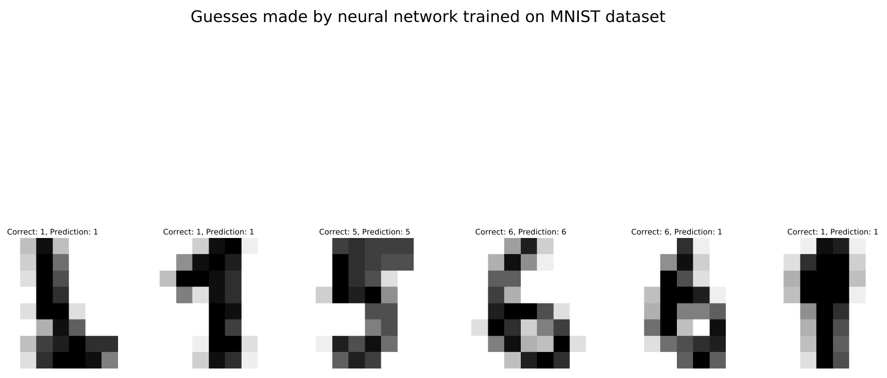

Guesses made by neural network trained on MNIST dataset
Correct: 1, Prediction: 1
Correct: 1, Prediction: 1
Correct: 5, Prediction: 5
Correct: 6, Prediction: 6
Correct: 6, Prediction: 1
Correct: 1, Prediction: 1

Figure 10: A sample of the MNIST data set, with the prediction made by our neural network and the corresponding correct label.

5 Discussion

5.1 Linear Regression

When applying SGD to OLS and Ridge regression, we expect that SGD will perform worse and produce answers with higher MSE and lower R2 scores, as OLS and Ridge regression have analytic answers for the global minimum of the MSE. The results shown in figure 1 and figure 2 show this for OLS. While the gradient descent algorithm might possibly produce a lower error through random chance on the test data set, the analytically determined values from OLS will always be the best possible values for the training set.

In figure 1 we see that the SGD algorithm seems to produce best results for relatively low numbers of minibatches, between 2 and 4 the algorithm manages to almost reproduce the OLS analytic minimum. In the extremes of having only one single batch for the entire training set, there is a sharp dropoff in accuracy, with the R2 score indicating that the error is worse than predicting the average of the Franke function. Even a single split into two batches produces a large accuracy improvement, seemingly corroborating the purpose of splitting the training data into minibatches to avoid getting bogged down in local minima.

At the other end of the spectrum, having a single minibatch for every single data point seems to produce much bigger errors, as one might expect. With singular data points, it is possible that the global minimum across all data points the algorithm is searching for is simply not present for each minibatch. It is also worth noting that the time spent on the algorithm is proportional to the minibatch count.

From this analysis, it appears that a small number of minibatches will produce optimal results with respect to accuracy and time.

Figure 2 shows that the learning rate η is of vital importance in obtaining a meaningful minimum when using stochastic gradient descent. Beyond a certain learning rate, the algorithm ceases to find a meaningful minimum at all. If the learning rate is too high, the algorithm will skip over minima when traversing the parameter space, so this does fit our perception of learning rate.

Moreover, this maximal learning rate depends on learning rate decay. Without decay or with slow decay, the learning rate must be controlled in a strict interval to avoid divergence. On the other hand, the possible minima seem to be lower for smaller decay, implying that lower or no decay might produce better results if optimized, but are more prone to failure.

The "landscape" or gradient of the cost function in parameter space is very dependent on the given data set, shape of the cost function and model used, so the learning rate must be tuned for each individual problem set to achieve optimal results. In this sense, a high rate of decay can be valuable to increase the interval of acceptable learning rates.

For the case of Ridge regression, figure 3 shows quite clearly that the hyperparameter acts much the same on the SGD-found minima as it does on the analytic ones. However, from equation (35), we see that λ appears directly in the expression for the gradient, and thus a large λ will lead to a large gradient. We can then see why the accuracy decrease for larger λ seems to be so much more prevalent for the SGD case than the analytic one. Larger hyperparameters produces divergences, similarly to figure 2 for high learning rates.

5.2 Logistic Regression

When comparing figure 9 with figure 2, we see that the interval of acceptable learning rates for SGD seems to be larger for logistic regression than linear regression. Furthermore, the learning rate decay has a smaller effect on the accuracy score. The accuracy score seems to stabilize around 0.95 for all three decay rates, and whereas the linear regression case had divergences appearing already around $t0 = 0.3$, the logistic case kept stable until roughly $t0 = 3$.

In figure 8 we see that logistic regression seems to produce a result on par with the FFNN algorithm.

5.3 Neural Network performance

As can be seen in figures 4, 5, 6, 7, the two logistic functions, sigmoid and the hyperbolic tangent function both perform much better than the ReLU and leaky ReLU functions for this regression case. This is quite unexpected, as in general, the RelU function and its relatives should perform better than the hyperbolic tangent which again should be and is better than the sigmoid function. Seeing as the leakyReLU outperforms the ReLU function, and the hyperbolic tangent outperforms the Sigmoid function, it might be the case that the ReLU family of functions is worse due the the complexity, or lack thereof in the problem at hand. It might also be that ReLU function and its relative the leakyReLU perform worse due to the our choice of hyperparameters, or in the case of the ReLU function, due to the fact that ReLU has a chance of "killing" neurons in the network during training, meaning that the neurons stop outputting values other than zero, however, this would not explain why the leakyReLU performs so poorly compared to the logistic functions. Most likely, it is a case of hyperparamters values being chosen poorly or not optimally, ensuring that the weights and biases never reach the best values given the training epochs. The R2 scores might also suggest that the gradients vanish/explode seeing as our code for the neural network had cases of NaN values appearing during training. Finally, it might be that ReLU functions perform better when used over several hidden layers, however, we were not able to test this.

5.4 Comparing Neural Networks with Regression

Overall, it seems that both for the linear regression and logistic regression cases that the FFNN is similar in terms of accuracy, provided a prudent choice of minibatch size and learning rates. While this is somewhat surprising, it is possible that differences in the SGD-algorithm used in the FFNN and logistic and linear regression is responsible. The back propagation algorithm in the FFNN code does not repeat to test multiple starting points in the weights and biases, which could potentially lead to an unfortunate starting point that finds only a local minimum in the output error.

Further modification and improvement is necessary, as is more testing, to conclude on any general terms the qualities of FFNN and logistic regression. However, from the above results, ordinary linear and logistic regression methods appear to provide solutions on par with the feed forward neural network method. Perhaps the problems studied here are too simple for a neural network to have a large impact.

6 Conclusion

Overall, it seems that for both the classification problem and function fitting problem explored in the project, FFNN is roughly equivalent with the SGD-based Logistic and

Linear regression. Given that linear regression for Ridge and OLS have analytic solutions requiring almost no computing power or runtime to compute, the FFNN and SGD-based OLS and Ridge regression studied in this project seem unnecessary for function fitting to data. However, more complicated models and data structures might prove otherwise. On the other hand, Logistic regression does not have any analytic solution, and for classification problems SGD appears to do quite well, in both FFNN and Logistic regression achieving an accuracy score of above 95% when recognizing hand drawn digits from the MNIST data set.

Future work would involve fine-tuning the FFNN and SGD algorithms, and more testing to see if perhaps on more complicated data sets, both for function fitting and classification problems, the neural network might perform better than the more ordinary regression methods.

7 Appendix

Github repository with codes and figures can be found at `https://github.com/Gudmunhg/ML-projects`.

References

[1] HASKELL B. CURRY. "THE METHOD OF STEEPEST DESCENT FOR NON-LINEAR MINIMIZATION PROBLEMS". In: *Quarterly of Applied Mathematics* 2.3 (1944), pp. 258–261. ISSN: 0033569X, 15524485. URL: `http://www.jstor.org/stable/43633461`.

[2] Richard Franke. *A Critical Comparison of Some Methods for Interpolation of Scattered Data.* 1979. URL: `https://calhoun.nps.edu/handle/10945/35052`.

[3] Morten Hjorth-Jensen. *FYS-STK3155/4155 Applied Data Analysis and Machine Learning.* URL: `https://github.com/CompPhysics/MachineLearning`. (accessed: 10.10.2020).

[4] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[5] S. Sra, S. Nowozin, and S.J. Wright. *Optimization for Machine Learning.* Neural information processing series. MIT Press, 2012. ISBN: 9780262016469. URL: `https://books.google.no/books?id=JPQx7s2L1A8C`.

[6] Zi Wang, Aws Albarghouthi, and Somesh Jha. *Abstract Universal Approximation for Neural Networks.* 2020. arXiv: `2007.06093 [cs.LG]`.