FYS-STK4155 Project 2: Classification and Regression - from linear and logistic regression to neural networks

Sushma Sharma Adhikari, Gert Kluge and H. Alida F. Hardersen
Fysisk Institutt, Universitetet i Oslo
(Dated: November 13, 2020)

Using Neural Networks for classification and regression is an interesting area of study. Here we compare the implementation of a Feed-Forward Neural Network (FFNN) for linear regression and classification, to using basic OLS and Ridge linear regression models, as well as a Logistic regression method for classification. We find that the learning rate used in the SGD optimisation has a large impact on the regression results, and using a cyclic learning schedule yields an explained R^2-score of 0.76 compared to a non-cyclic schedule which yields an R^2 of -0.09 for pure SGD regression. In the case of pure SGD, we found that the optimal value when using a static learning rate is $\alpha = 0.1$, which yields a MSE value of 0.02 and an R^2-score of 0.81. When using the FFNN with a Leaky ReLU activation function the performance is close to that obtained from classic OLS, with a MSE below 0.015 for both. The FFNN preforms well for classification of the MNIST dataset, with the best prefomance when we introduce a L1/L2 regularization parameter. When comparing the preformance of the FFNN to those of Logistic regression we found that the Neural network preformed approximately 3% better.

I. INTRODUCTION

The history of neural networks starts in 1943 when Warren Mculloch and Walter Pitts created a computational model based on how neurons in the brain are connected. In the years later, Artificial Neural Networks (ANN) have become a staple in the field of Machine learning and are used in many applications such as Gmail's smart sorting and suggestions on Amazon. Neural Networks comes in different types for different purposes such as the Convolutional neural network which is specifically designed for image processing and the Recurrent Neural Networks which is used for sequential data. Common for most ANN's is that they consist of one or more hidden layers sandwiched between an input layer and an output layer, and all layers contain a number of nodes which are connected. The first and simplest type of network is the Feed-Forward Neural Network (FFNN) where information only moves forward in the layers. ANN's are the foundation of artificial intelligence, and can be used to find solutions which tend to be difficult to attain with a simple logistic model. They model non linear processes and are a useful tool for solving problem such as classification, regression, decision making, pattern recognition etc. In this project we will be using a FFNN to study both classification and regression problems. The Network is trained using the Stochastic Gradient Descent(SGD) method and will be tested using various activations functions for the hidden layers, such as Sigmoid , Softmax, RELU and Leaky RELU. For regression we will again use data created using the Franke Function, as we did in project 1 [1]. For classification we will be using the MNIST dataset, which contains images representing hand-written numbers from zero to nine. We will begin by introducing the methods we have used and parts of the theory behind them. In the case of linear regression we will simply summarize the theory developed in project 1 [1], so we refer to that paper for more details regarding OLS and Ridge regression. We will then present and discuss our results, before reaching a conclusion.

II. LINEAR REGRESSION

Assume the response y is a linear combination of the features $x_1, x_2, ...$, then

$$y = \hat{y} + \text{noise}, \qquad \hat{y} = X\beta \tag{1}$$

β are the regression parameters and X is the design matrix. The objective is to find the values of the regression parameters β that models the data the best. These values will be indicated with a hat: $\hat{\beta}$. To give an indication of how well the model fits the data we define a cost function $C(\beta)$ that describes the difference between the actual response variable y and the prediction $\hat{y}$. This difference is called the residuals, ϵ, and the most common choice of cost function is the mean square error (L2-norm, residual sum of squares)[1]:

$$C = ||(y - \hat{y})||_2^2 = \frac{1}{n}\epsilon^T\epsilon, \quad \text{where,} \quad \epsilon = y - \hat{y} \tag{2}$$

We can find the optimal parameter values in two ways:
1. Analytic: Where we find the exact analytic solution to $\nabla C = 0$. This gives the OLS regression parameters:

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{3}$$

2. Iteratively: Where we find a numerical approximation of this solution by using an iterative method such as the gradient descent method where we use the gradient of the cost function,

$$\nabla C(\beta) = \frac{2}{n} X^T (X\beta - y) \tag{4}$$

This method is described in more detail below. We can also use the Ridge regression method where we introduce the hyperparameter λ to avoid issues with singular matrices[1]. The cost function for the ridge regression method is,

$$C(\beta(\lambda) = \frac{1}{n}(y - X\beta)(y - X\beta)^T + \lambda\beta^T\beta \tag{5}$$

and it's gradient is

$$\nabla C(\beta, \lambda) = \frac{2}{n} X^T (X\beta - y) + 2\lambda\beta \tag{6}$$

III. THE GRADIENT DESCENT METHOD

Gradient descent is an algorithm used to minimise some function by iteratively moving in the direction of the steepest descent. With a given cost function, this can be used to optimise regression parameters or weights and biases when it is not possible to solve for where the cost function is zero analytically. If we plot the cost as a function of parameters, we can take the derivative of the cost function for each parameter value to find the slope of the curve for that value, this is also called the gradient ∇C. The optimal parameter values are the ones for which the cost is at it's minimum, which is the bottom of the curve in the figure. Here, the gradient of the cost function would also be close to zero. We can call the parameters β, but it can be any function. The gradient descent algorithm finds this minimum by first making an initial guess for the parameters β, and calculating the gradient,

$$\nabla C = \left(\frac{\partial C}{\partial \beta_1}, \frac{\partial C}{\partial \beta_2}, ... \right)^T \tag{7}$$

We could try out millions of different parameter values to see which gives the smallest value, but it would take a lot of time and computing power. Instead, what the Gradient descent algorithm does is to only do a few calculations when the gradient is large and increase the number of calculations as it becomes closer and closer to zero. Or in other words, the step size $\Delta\beta$ decreases as the gradient decreases. It is however important to choose the proper step size because as the gradient increases, the step-size also increases and if we take too large a step we could end up skipping over the global minimum of the curve and suddenly have a larger cost than we started with. Therefore, $\Delta\beta$ should be related to the gradient value which is done by multiplying the gradient with a small positive parameter called the learning rate, α. The step size becomes

$$\Delta\beta = -\alpha\nabla C \tag{8}$$

The new values of the parameters can then be calculated,

$$\beta_{i+1} = \beta_i - \alpha\nabla C(\beta_i) \tag{9}$$

The parameters will be updated multiple times, each time decreasing the cost function C, until we either reach a maximum number of iterations where we hope we have reached a global minimum, or when we reach a stopping criteria. There are multiple stopping criteria to choose from, but in this project we will stick to the criteria that the cost diverges. That is, we calculate the cost with the beta values at the end of each iteration(sometimes referred to as an epoch in the case of stochastic gradient descent or batch gradient descent) and check if the cost has changed from the previous x iterations. To recap, the algorithm for the basic gradient descent method is:

1. Make an initial guess for the regression parameters
2. Calculate the gradient of the cost function $\nabla C(\beta_i)$
3. Calculate the step size, $\Delta\beta_i = \alpha\nabla C(\beta_i)$
4. Update the parameters, $\beta_{i+1} = \beta_i - \alpha\nabla C(\beta_i)$
5. Repeat steps 2-4 until a maximum number of iterations or a stopping criteria is reached.

A. Stochastic Gradient Descent

For complex models with a large dataset and many features, the gradient descent method can be slow and inefficient. To speed up the process the stochastic gradient descent (SGD) method can be used, where a randomly selected subset of the data is used to calculate the gradient in each step. With the observation that the cost function, and its gradient, can be written as a sum over n datapoints,

$$C = \sum_{i=1}^{n} C_i, \quad \text{and} \quad \nabla C = \sum_{i=1}^{n} \nabla C_i \tag{10}$$

A minibatch B_k of M randomly selected datapoints is chosen to use for training, and with n datapoints the number of such minibatches is $m = n/M$. One iteration over all m minibatches is called an epoch, and the gradient is then calculated as an average over a minibatch,

$$\nabla C(\beta) = \frac{1}{M} \sum_{i \in B_k} \nabla C_i(\beta) \tag{11}$$

This should speed up the calculation because we do not use the entire dataset to calculate the gradient. The SGD method is sensitive to the value chosen as the learning rate, it is therefore common to instead introduce a *learning schedule* so that the learning rate decreases with each step. If we let $e = 0, 1, 2, 3, ...$ be the current epoch, $t_0, t_1 > 0$ be two fixed numbers, and $t = em + i$ where m is the number of minibatches and $i = 0, 1, ..., m-1$ is the current minibatch, then the learning rate is given by the function[2],

$$\alpha(t) = \frac{t_0}{t + t_1} \tag{12}$$

so the learning rate will approach zero as the number of epochs become large. For complex functions, we may get stuck in a local minimum. To account for this, SGD is

normally used with a momentum term that accounts for the direction in which we are moving.

$$\beta_{t+1} = \beta_t - \alpha m_t \tag{13}$$
$$m_t = \gamma m_{t-1} + (1-\gamma)\nabla C \tag{14}$$

This works because in SGD we are estimating the gradient on a small batch so the direction we are moving in may not always be the optimal one. So this momentum term can give a better estimate that is closer to the true gradient. If we are stuck in a saddle point, momentum can accelerate the gradients in the right direction.

IV. LOGISTIC REGRESSION

For simple classification problems, as linear regression is unbounded, logistic regression which ranges from 0 to 1 is appropriate. We will first consider a simple logistic regression model and then we will discuss a more complex feed forward neural network. Let us consider a dataset $\{x_i, y_i\}$ consisting of n samples with p features or predictors. The dependent variables y_i are discrete and i dimensional vector, $i = 0, 1, 2, ...,$ (K classes). Primarily the problem is to predict the output class given $\hat{X} \in R^n \times p$. In the case of a binary class with outcomes $y_i = 0$ and $y_i = 1$, in contrast to linear regression where the dataset $y_i = f(x_i) + \epsilon_i$, noise ϵ_i follows a Gaussian distribution, in logistic regression $y_i = p(x_i) + \tilde{\epsilon_i}$ where $\tilde{\epsilon_i}$ follows binomial distribution and $p(x_i)$ is the likelihood function. The likelihood function can be given by the Sigmoid30 (logistic activation function) function for binary problem with two features and with $y \in 0, 1$, or by Softmax for multi-class. The output from logistic regression is the estimated probability, through which one can infer given an input X how probable can be the actual value. For a binary class,

$$p(y_i = 1|x_i; \boldsymbol{\beta}) = \frac{\exp\{(\beta_0 + \beta_1 x_i)\}}{1 + \exp\{(\beta_0 + \beta_1 x_i)\}} \tag{15}$$
$$p(y_i = 0|x_i; \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}) \tag{16}$$

Where the β's are parameters of the model chosen to maximize the likelihood, or in other words weights to be extracted from data. Logistic regression models are usually fit by using the maximum likelihood estimator[3]. The total probability of observing an event from a dataset $\{(x_i, y_i)\}$,

$$P(y|X\beta) = \prod_{i=1}^{n-1} P(y_i|x_i, \hat{\beta}) \tag{17}$$
$$= P(y_i|x_i, \hat{\beta})^{y_i}(1 - P(y_i|x_i, \hat{\beta}))^{1-y_i} \tag{18}$$

where

$$P(y_i|x_i, \hat{\beta}) = \frac{\exp\ (\beta_0 + \beta_1 x_i)}{1 + \exp\ (\beta_0 + \beta_1 x_i)} \tag{19}$$

The Likelihood function is given by ,

$$L(\hat{\beta}) = -p(y_i = 1|x_i; \boldsymbol{\beta}) p(y_i = 0|x_i; \boldsymbol{\beta}) \tag{20}$$

the logarithm of this is a monotonically increasing function i.e., for

$$p(x) > p(y) \implies log(x) > log(y) \tag{21}$$

this implies the log-likelihood and the likelihood have same relation. Taking logarithm makes it easier as it turns the exponential into a summation. Also, if the likelihood becomes very small, it will not be able to distinguish between two very small values of y. We need a cost function such that the effect of small changes are also reflected for the purpose of optimization. So, the cost function can be defined as the negative logarithm of the likelihood function,

$$C(\hat{\beta}) = -\Sigma_{i=1}^{n} \left[y_i\ logP_i + (1 - y_i)\ log\ (1 - P_i) \right] \tag{22}$$

which can be further rewritten as

$$C(\hat{\beta}) = -\Sigma_{i=1}^{n} \left[y_i\ (\beta_0 + \beta_1 x_i) - \log(1 + \exp\ (\beta_0 + \beta_1 x_i)) \right. \tag{23}$$

This equation is also known as cross entropy. The logistic function takes y-values to be between 0 and 1, while for cross-entropy we extract the difference between probabilities that are close 0 or 1. Minimizing the cost function w.r.t two parameters β_0 and β_1,

$$\frac{\partial C}{\partial \beta_0} = -\sum_{i=0}^{n-1}(y_i - P_i) = \sum_i (P_i - y_i) \tag{24}$$

For a vector $\hat{y_i}$ with elements n and an $n \times p$ matrix $\hat{X_i}$, we have the derivatives

$$\frac{\partial C}{\partial \beta} = X^T(P - Y) = g(\beta) \tag{25}$$

$$\frac{\partial^2 C}{\partial \beta\ \partial \beta^T} = H = \sum_i P_i\ (1 - Pi)\ X_i)\ X_i\ X_i^T = \hat{X}^T \hat{W} \hat{X} \tag{26}$$

Where H is the Hessian matrix. For the present problem discussed in the project we have to focus on multi-class classification ,i.e. for $\{x^{(1)}, x^{(2)}, ...x^{(n)}\}$ we have,

$$y = \text{Softmax}\{z_1, z_2, z_3, ...z_K\} = \frac{e^{z_k}}{\sum_j e^{z_j}} \tag{27}$$

We would have $K = 10$ different classes for MNIST digit recognition. The predicted probability for the K-th class given a sample vector $\hat{x}$ and a weighting vector $\hat{\beta}$ is (with multiple predictors, we have Softmax function, it is generalization of sigmoid function, considered when we need

to handle multiple class problem.), The cost function defines as a sum over the cross-entropy loss for each point x_i in the dataset.

$$C(\beta) = -\left[\sum_{i=1}^{n}\sum_{l=1}^{K}\log\frac{\exp(\beta_{li}x_i)}{\sum_{j=1}^{K}\exp(\beta_{ij}x_i)}\right] \tag{28}$$

By minimizing $C(\beta)$ we can find β. With norm 2 regularization (L_2), our cost function is

$$C(\hat{\beta}) = -\left[\sum_{i=1}^{n}\sum_{l=1}^{K}\log\frac{\exp(\beta_{li}x_i)}{\sum_{j=1}^{K}\exp(\beta_{ij}x_i)}\right] + \lambda\|\beta\|^2$$

where $\lambda > 0$ is the regularisation strength. An extra term to the cost function, proportional to the size of the weights. Constraining the size of the weights, so that they do not grow out of control. Constraining the size of the weights means that the weights cannot grow arbitrarily large to fit the training data, and in this way reduces over-fitting. It is not possible to find analytical solution as can be seen in Eqn.25 derivative is non linear in β hence to find optimal parameters β's we used gradient descent method. Also from Eqn.26 where we have hessian to be positive , which implies our cross entropy is a convex function for positive semi definite matrix X.

For our implementation of Logistic regression we simply run our neural network with no hidden layers, using the softmax activation function and the cross-entropy cost function in equation 23.

V. NEURAL NETWORKS

Artificial neural networks (ANN) perform tasks without programming it with any specific rules related to the tasks to be performed. As the name suggest, inspired by the general idea of biological networks of neurons in the brain, these computational models have the ability to learn from series of repeatedly shown patterns and predict the outcomes based on the given input data. As mentioned, the ANN's consists of an input layer, one or more hidden layers and an output layer all of which consists of a certain number of nodes which are connected to the nodes of the other layers. The connections between the nodes have a corresponding weight which relate to the importance between the inputs.

A. Feed-Forward Neural Network

The feed-forward neural network is as mentioned the simplest type of ANN. In these types of networks the information only flows in the forward direction, with no feedback connections where the outputs are fed back into itself, hence the name. The output y of each node is created using the activation function a,

$$y = a\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z) \tag{29}$$

where w_i are the weights, b_i is the bias and x_i are the inputs. The output of one layer of neurons is the input for the next layer. If the network is fully connected, so that all neurons in one layer are connected to all neurons in the other layer, each neuron receives the weighted sum of the outputs from all neurons in the previous layer.

B. Activation functions

To produce an output for the nodes in a layer we need to use an activation function. The output is then used as input for the next layer and so on until we reach the final output. There are many different activation functions to choose from, and because machine learning is not based on theoretical predictions[4], the optimal choice of activation function is based largely on trial and error. Here we will present some of the activation functions used in our FFNN.

1. Sigmoid

The Logistic activation function is given by equation 30, with z_i as the output value from the ith node

$$\sigma(z_i) = \frac{1}{1+e^{(-z_i)}} \tag{30}$$

This returns a value between 0 and 1, which makes this an especially good choice when a prediction of the probability of the output is needed. However, the Sigmoid function may lead to long training time as the calculation of the exponential can be computationally heavy. In addition, the Sigmoid function has a small gradient in most of it's domain, which can lead to the SGD algorithm running slowly (see for example [5]). This may be avoided however, for example by employing the cross entropy as cost function.

2. ReLU (Rectified Linear Unit)

The ReLU function is one of the most popular activation functions, it has an output of zero for any input less than zero and if the input is larger it returns the value as is,

$$a(z_i) = \begin{cases} 0 & \text{if } z_i \leq 0 \\ z_i & \text{if } z_i > 0 \end{cases} \qquad a(z_i) = \max(0, z_i) \tag{31}$$

with this ReLU function [6] it is easier to back-propagate without any error. But it has some drawbacks, one of

which is the dying ReLU problem. For input values below 0, neurons become inactive and only output 0 for any input and when this happens to a large number of neurons it can affect the performance of the neural network. This problem can be solved by rather using a variant called leaky ReLU.

3. Leaky ReLU

Leaky ReLU is an extension of ReLU achieved by making some slight changes in the slope in the left of $x = 0$ (for negative values)[7]

$$a(z_i) = \begin{cases} az_i & \text{if } z_i < 0 \\ z_i & \text{if } z_i \geq 0 \end{cases} \tag{32}$$

where a is the leakage parameter. This prevents neurons from dying off by avoiding zero-gradients.

4. Softmax

The softmax activation function normalizes an input value into a vector of values that follows a probability distribution whose total sums up to 1, and is given by 33

$$\text{softmax}(z) = \frac{e^z}{\sum e^z} \tag{33}$$

It is often used in the final layer of a network. In addition to these activation functions, we will also be implementing a linear activation function that simply returns the input value and a binary step function.

C. Our implementation

In this project, we have limited ourselves to neural networks that have a single hidden layer. This is because neural networks with more layers (deep neural network) are potentially a lot more complicated, and introduce a large number of extra variables that would explode the scope of this project. We have only treated the number of nodes in the hidden layer as a free variable, that is, the input and output layers have had a fixed number of nodes, corresponding to the number of input and output parameters. To be specific, the networks that we used for the regression of Franke function data had two input nodes (corresponding to the independent variables x and y) and one output node (corresponding to the estimated function value). For the classification of MNIST data we used 784 input nodes (corresponding to the number of pixels in each picture), and 10 output nodes (corresponding to the 10 classes).

We have also fixed the activation functions for the output layer to be the linear function $f(z) = z$ for the regression task, and the softmax function for the classification case. For regression, the linear function is suitable as it is unbounded (from both above and below), which are essential properties for the estimated function value. For the classification case, softmax is the most suitable choice (that we are familiar with), since it emulates the behaviour of probabilities (that is, the activations for each class sum up to one), while the exponential function exaggerates differences in the function argument, thus forcing the network to make clearer choices between the classes.

As for the cost function, we have mostly limited us to the quadratic cost, that is equation (2), except summed over the output values. There are good arguments for using different cost functions, in particular the cross-entropy (23), which has the advantage of speeding up training when activations are close to saturation [5]. This choice of cost function would also have made it possible to compare the performance of our neural networks to that of logistic regression more directly. However, we have made this choice to simplify the scope of our project.

In general, we have initialised the weights of our neural networks using a Gaussian random variable with variance 1 and mean 0. We might have gotten an advantage from using a lower variance, to reduce the number of activations that are initially saturated, but we never experienced this to be necessary. The biases were generally initialized to 0. A different choice would have given the activations an initial bias different from 0, which could slow down learning (since the sigmoid gradient is largest around 0).

TABLE I: Specifications of the data generated using the Franke Function with n datapoints, the noise contribution has size μ. The fraction of the data that is saved for testing is given by the test size.

Size (n)	Noise strength (μ)	Test size
1024	0.1	0.2

VI. RESULTS AND DISCUSSION

In the first part we are using data generated using the Franke Function with normally distributed noise of strength μ, specifications can be found in table I. The data is separated into testing and training sets and both are scaled by subtracting the mean of the test set and dividing by the variance. The design matrix is generated using monomials up to and including degree d as described in [1].

A. Testing the Stochastic Gradient Descent Algorithm

We will begin by studying results found when using SGD for OLS regression. The SGD method is very sensitive to both the value of the learning rate, and the number of epochs, as can be seen in figure 1. It shows the

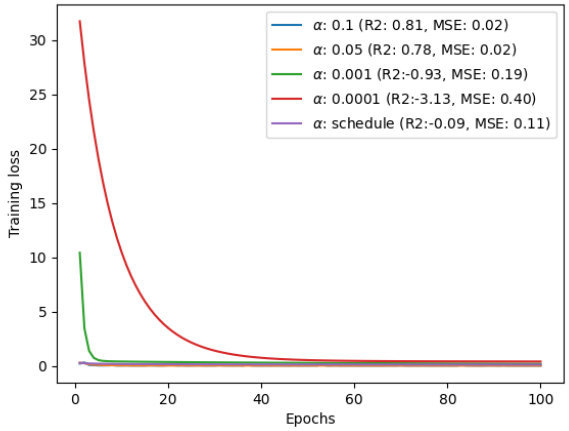

FIG. 1: The training loss as a function of epochs using SGD regression for OLS with different learning rates α. The batch size used is $M = 5$ and the polynomial degree of the design matrix is set to $d = 7$

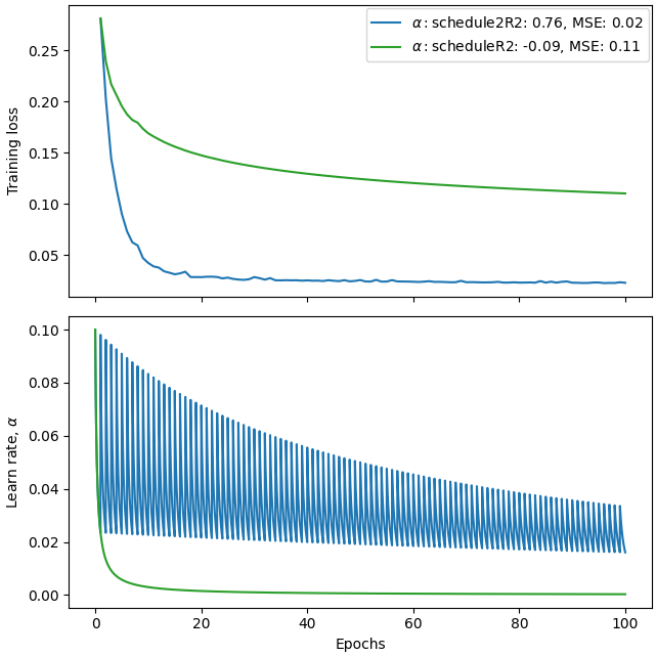

FIG. 2: Training loss as a function of epochs for the two schedule types. Here shedule2 corresponds to the updated learning schedule we tested. The batch size used is $M = 5$ and the polynomial degree is set to $d = 7$

training loss calculated using equation 2 as a function of the number of epochs for five different values of the learning rate $\alpha = 0.1, 0.05, 0.001, 0.0001, 10^{-5}$, as well as using the learning schedule described in 12 with $t_0 = 5$ and $t_1 = 50$. With a very small learning rate of $\alpha = 10^{-5}$, the step size is so small that we would need more epochs before converging to a minimum value of the loss, and as the learning rate increases the number of epochs needed decreases, as we would have expected. The optimal values of α turn out to be 0.1 and 0.05, where the testing loss (MSE) is 0.02. A very interesting result is that when using the learning schedule as described in (12), the step size becomes small very quickly so it would seem that we would need a larger number of epochs before we reach a minimum in the training loss. As an experiment we try to adjust the learning schedule by replacing $t = ep*m+i$ with $t = ep + i$ so that the learning rate will decay less rapidly and so it will vary cyclically between each epoch. In a way we restart the learning rate at each epoch, while also decreasing the starting value a small amount each new epoch. A comparison between the two schedules is shown in figure 2. The top figure shows that the training loss decreases more rapidly for the cyclic method than it does for the standard one. In [8], Dauphin et al describes how one source of difficulty could be the saddle points, in addition to a local minimum with higher error than the global minimum has. So a reason for why the first learning schedule had difficulty may be that we keep getting stuck in these saddle points, while for the cyclic learning schedule we then increase the learning rate which allows for a more rapid traversal of these saddle point plateaus, thus the improvement.

When looking at the performance of SGD when varying the size of the minibatches and the learning rate, we find that for small batch size and high learning rate the calculation of the loss fails because of an overflow error. However, when using the cyclic learning schedule with batch size 2 and 100 epochs the performance is remarkably better with a R^2-score of 0.73 and a test error of 0.026. From table II we can see that the best combination for this dataset using a design matrix of polynomial degree 7 is batch size 5 with a learning rate of 0.1, closely followed by the combination batch size 5 and learning rate 0.05. We find that the training loss for these two models are not perfect continuous lines as it may seem in 1, as shown in figure 3 there are spikes which may be due to "bad batches" which are minibatches with high loss. When adding a momentum parameter $\gamma = 0.9$ these spikes are dampened. These "bad batches" can become problematic for training with small batch sizes and high learning rates, and is possibly one of the reasons that the algorithm fails for smaller batch sizes. We have also used the same two models to study the performance as a function of model complexity as shown in figure 4, in comparison to using the ordinary OLS regression method. We find that both the R2 and MSE values compare well to the OLS method for low complexity. The SGD method performs badly for polynomial degrees between 4-6, while we see some improvement for degree 7-10.

Now moving on to Ridge regression, figure 5 shows the training loss as a function of epoch number using different values of hyperparameter λ and learning rate α. The training loss for $\lambda = 0.001$ and $\lambda = 10^{-5}$ are very similar for all learning rates, with the spikes of $\lambda = 0.001$ being somewhat smaller. The spikes in training loss are more predominant when using a learning rate of $\alpha = 0.1$,

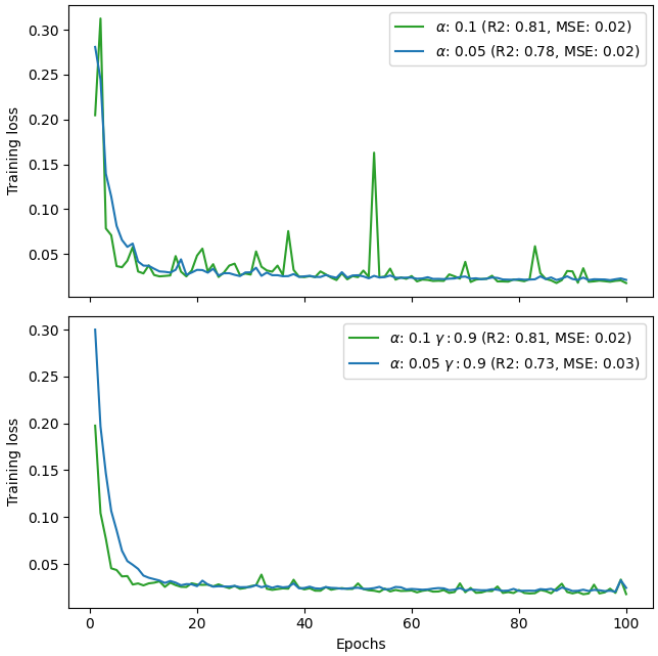

FIG. 3: The top figure shows the two best results from figure 1. The bottom figure shows the training loss when introducing a momentum element with $\gamma = 0.9$. The batch size used is $M = 5$ and the polynomial degree is set to $d = 7$.

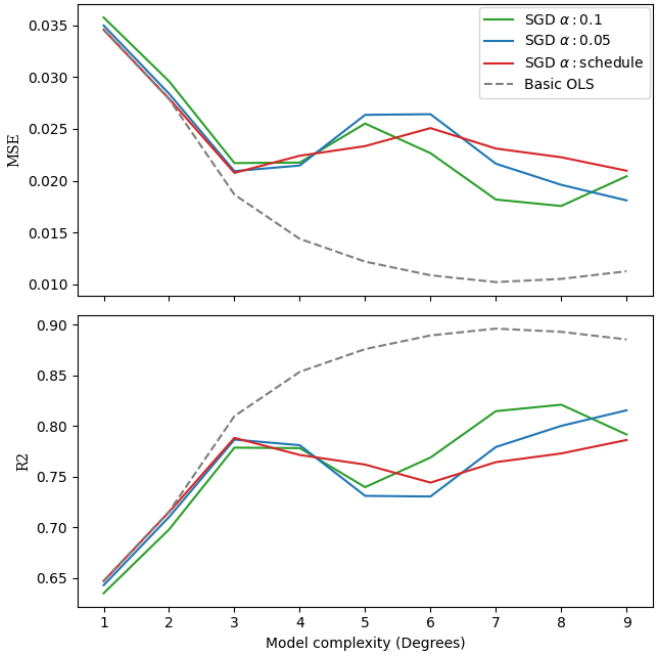

FIG. 4: The mean squared error and R^2-score as a function of model complexity (polynomial degree of the design matrix) when using the SGD algorithm for OLS. The green and blue line are results gained with learning rates of 0.1 and 0.05 respectively and the red is when using the cyclic learning schedule. The batch size used is $M = 5$ and the number of epochs is 100. The dotted gray line shows the corresponding result when using the basic OLS regression method.

TABLE II: Performance of the stochastic gradient descent algorithm for OLS with 100 epochs for different learning rates (top header) and batch sizes (left header). The data is described in table I, polynomial degree used is 7.

		0.1	0.05	0.001	0.0001
2	R^2	nan*	-10^{23}	0.69	-1.59
	MSE	nan*	10^{22}	0.03	0.25
5	R^2	0.81	0.78	-0.93	-3.13
	MSE	0.02	0.02	0.19	0.40
10	R^2	0.76	0.74	-2.55	-19.00
	MSE	0.02	0.03	0.35	1.96
100	R^2	-2.06	-2.65	-202.36	-319.99
	MSE	0.30	0.36	19.93	31.46

*Overflow encountered when calculating the gradient

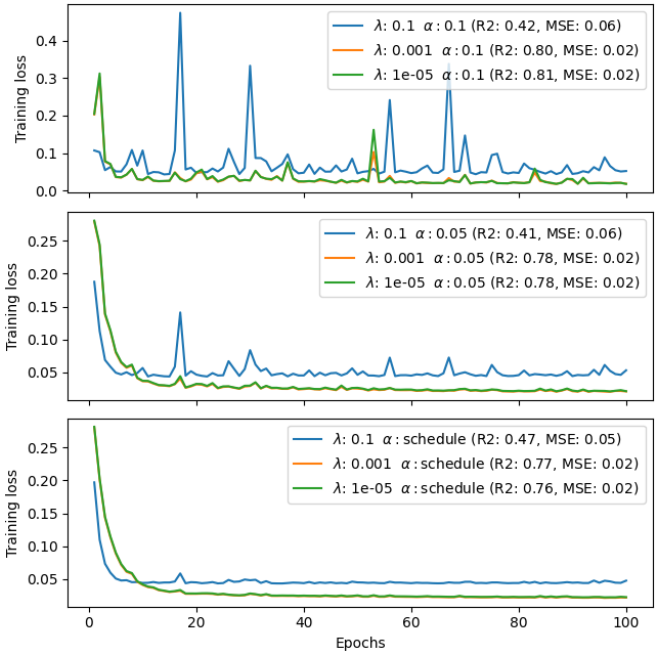

FIG. 5: Training loss as a function of epoch number using various values of hyperparameter λ and learning rate α

and when using the cyclic learning rates the spikes are significantly dampened. The model with the best R2-score is the one we gain by using a learning rate of 0.1 with a hyperparametervalue of 10^{-5}.

B. Selection of hyperparameters

Before we start presenting results on the performance of our neural networks, we discuss briefely how we went about determining the "optimal" hyperparameters for each regression (or classification) task. We include this as part of the results, rather than the theory, because we are basing this discussion more on our experience — re-

sulting from many cumulative hours of experimentation with the code — as opposed to theoretical deliberations (although the latter certainly play a role in interpreting the procedures that we will discuss). As such, this discussion also serves as a description of how the various parameters affect the performance of a given network.

Determining the optimal hyperperameters for a neural network is very laborious; there is no analytical recipe for determining them, and to find them by looping over the range of their possible values would be too computationally expensive (at least at our level). In addition, it turns out that when we use activation functions that are unbounded (such as RELU or a linear activation function, but also Sofmax), the networks become vulnerable to overflow-related errors, presumably because some activations will perpetually increase. Determining the best hyperparameters was therefore a matter of patient trial and error, and the final choices can at most be considered very rough estimates to the optimal set — if such a thing exists. Nevertheless, we did find that we could narrow down the range of suitable hyperparameters by following a rough set of guidelines when performing these trial and errors.

Here, we will not provide quantitative evidence for our claims, as this would simply require us to generate and print too many figures. Thus we keep this discussion experience-based. We refer loosly to the "predictive accuracy" of a network to refer to its ability to predict a value of the Franke function (or to correctly classify a handwritten digit). In general, we have measured this during our experimentation, using the value of the mean square error (MSE) between the predicted values and a set of testing data. In the case of classification, we have used the accuracy score, that is the proportion of testcases that the network was able to classify correctly.

We found that the size of the mini-batches which were employed in the SGD algorithm had no significant effect on a network's ability to predict the values of the Franke function. We would expect that making the mini-batches bigger has the effect of improving the accuracy of the gradients that are estimated during each epoch — however, larger mini-batches also mean fewer of them per epoch, which means that the network's weights and biases are updated fewer times per epoch and in total. It is conceivable that these two effects tend to cancel each other, which offers an explanation to why the size of the mini-batches doesn't seem to have a strong effect on the network's performance. Nonetheless, it was sometimes useful to increase the size of the batches, and thus the accuracy of the estimated gradient. This helped reduce fluctuations in the cost (which ideally should decrease gradually with each epoch), and preventing related problems in training. The variable which had by far the clearest effect on fluctuations however was the learning rate. Determining the learning rate was generally done by finding a value low enough so that there were no large fluctuations in the cost (as a function of epochs) that could interfere with the training process, but no lower, to prevent the SGD algorithm from slowing down unnecessarily. This trade-off is intuitive, as large learning rates have the potential to make the networks weights and biases "overshoot" some local extrema. This problem could be addressed by the momentum or learning-schedule approach, both of which our code in principle has the capacity for, but we haven't been able to test and explore these because of time constraints.

TABLE III: The hyperparameters chosen for most of our neural networks, unless otherwise specified.

Hyperparameter	Regression	Classification
Number of nodes in hidden layer	100	30
Mini batch size	30	30
Learning Rate	0.1	3
Regularization paramter λ	0	0
Number of data points λ	10000	10000

As for the number of nodes in the hidden layer, we also found that this had a minimal effect on the network's performance. In other words, the prediction accuracy didn't change significantly whether the number of hidden nodes was 10, 100, or 1000. This is perhaps not very surprising, given that we modelled the Franke function quite well in project 1 using at most a few tens of parameters. It would have been interesting to seen if we would see strong signs of over fitting for numbers of hidden nodes closer to the number of data points (10 000), but this was simply too computationally costly compared to the reward. We did try to reduce the number of data points to 1000, and of course got significant decreases in the predictive accuracy.

Table III summarizes which hyperparmaters we used for our neural networks. These values apply in every case, except where otherwise specified. We leave the discussion on regularization parameters for the next subsection.

C. Comparison to other models

We now compare the prediction accuracy of the neural-network to that of the OLS method. In figures 6 and 7 we can see how the MSE of the network's predictions improve with the number of training epochs. It is apparent that the end result — and MSE of about 0.04 and an R^2-score of about 0.7 — is comparable, but somewhat worse than for SGD applied to OLS (see again figure 4). Our network seems even less intelligent if we hold it up to the results of project 1: here the OLS method achieved even lower MSE's, down to 0.001. This makes some sense, since the SGD algorithm in practice makes a rough approximation of the optimal parameters, while the algorithms applied in project one find them analytically. However, one is led to ponder why we went through the effort of coding an entire new (and complicated) algorithm. This question will be answered later on when we show that the same FFNN can be used for classification by simply changing the activation function and tweaking

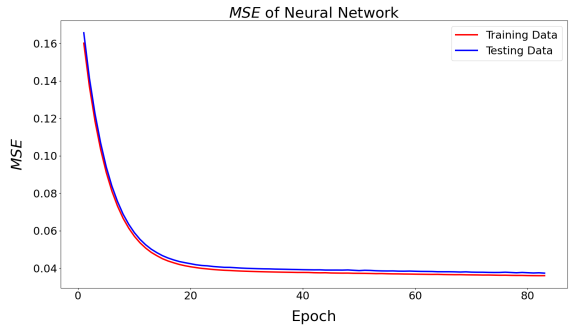

FIG. 6: Evolution of the MSE during the process of training the neural network to Franke function data. The smoothness of the curves, and the rather large number of epochs is a result of us having used a rather small learning rate, 0.001.

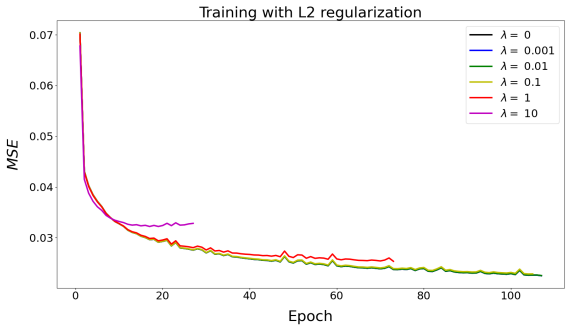

FIG. 8: Evolution of the MSE during the process of training the neural network to Franke function data, given for several values of the L2 regularization parameter λ. The learning rate used here was 0.1. The reason why some curves are cut short is because of the stopping condition included in the SGD algorithm.

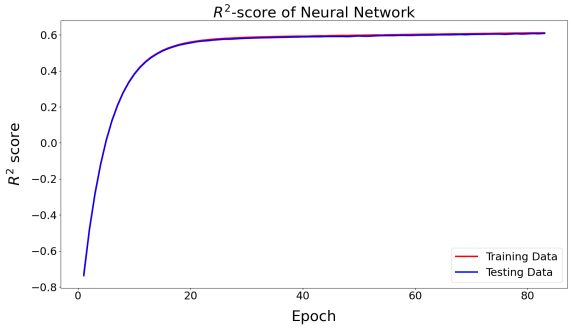

FIG. 7: Evolution of the R^2-score during the process of training the neural network to Franke function data. The smoothness of the curves, and the rather large number of epochs is a result of us having used a rather small learning rate, 0.001.

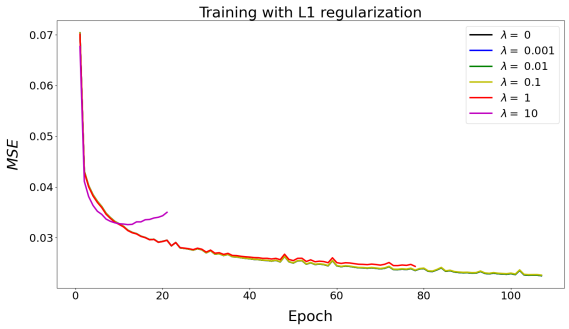

FIG. 9: Evolution of the MSE during the process of training the neural network to Franke function data, given for several values of the L1 regularization parameter λ. The learning rate used here was 0.1. The reason why some curves are cut short is because of the stopping condition included in the SGD algorithm.

the hyperparameters. In this way a neural network is far more versatile compared to the regular OLS method where we are limited to a certain type of input data.

Of course, our neural network doesn't represent the best possible performance that a neural network may be able to achieve for this task, given the extremely large number of variables that could be tweaked to improve it. One of those variables is of course the collection of hyperparameters, which we discussed in the previous subsection, and which we already admitted to not having determined completely optimally. It is worth mentioning though, that our experience has shown that when we change some of these hyperparameters (for example the number of hidden nodes, or the learning rate), the results do not change much. In fact, the new curves would almost overlap perfectly with the ones in figures 6 and 7 (we omit showing this graphically, to save some work and space). We thus need to make more sophisticated tweaks in order to improve the performance significantly.

D. Introducing regularization parameters

Figures 8 and 9 illustrate how the training process is affected by introducing respectively an L2 and L1 regularization parameter. Disappointingly, the effect is almost negligible, as is apparent by the fact that most of the MSE curves in these figures (except for the ones that correspond to extreme values of the regularization parameter λ) overlap almost completely.

The vanishing influence of regularization on the performance of the neural network should perhaps not be a great surprise. Regularization is after all a technique that is meant to reduce the effects of overfitting — however, there are no clear signs that overfitting has been a problem in the cases that we have presented so far. This is particularly apparent from the observation that the MSE calculated from the test-data traces that of the training data almost perfectly. This is similarly the case for the R^2-score. For an overfitted algorithm, we would

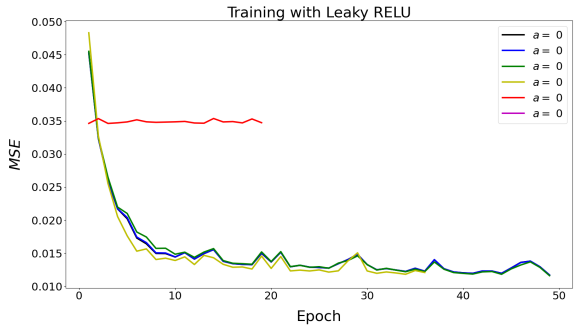

FIG. 10: Evolution of the MSE during the process of training the neural network to Franke function data. In this case we have used a leaky RELU function with leakage parameter a for the activation function of the hidden layer. **NOTE:** the values for λ should be, from black to magenta: 0, 0.001, 0.01, 0.1, 1 and 10 (we have avoided compiling the figure again to fix this, due to computational cost). For $a = 10$ many of the predicted values turned out to be nan. The learning rate used here was 0.01. The reason why some curves are cut short is because of the stopping condition included in the SGD algorithm.

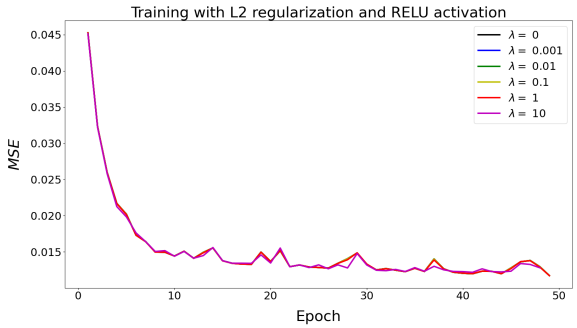

FIG. 11: Evolution of the MSE during the process of training the neural network to Franke function data, given for several values of the L2 regularization parameter λ. For the hidden layer activation function, we have used a leaky RELU function with leakage parameter $a = 0$. The learning rate used here was 0.01. The reason why some curves are cut short is because of the stopping condition included in the SGD algorithm.

also expect that the R^2 score approaches 1 (for the training data), at least if we include more fitting parameters (or nodes in the hidden layer). However, the R^2-score remains stable at just over 0.7, even when using 1000 nodes in the hidden layer. Perhaps, for the purpose of investigation, it would have been advantageous to choose the number of data points closer to the number of hidden nodes; that way over fitting would have been more likely to be a problem. This would anyways have been a more interesting investigation, given that it is more often the case that data is scarce rather than plentiful.

E. RELU as activation function

Until now, we have implicitly taken the sigmoid function as the activation function for the hidden layer. We now discuss how the network's performance changes if we replace the sigmoid function with the RELU function (the activation function for the output layer still remains linear). In practice, we will be using a leaky RELU function, with a variable "leakage" parameter a; we view the ordinary RELU function as a special case of the leaky RELU, with $a = 0$. Figure 10 illustrates how training is affected by introducing the leaky RELU function for different values of a. The clearest and most significant take-away is that the MSE of the trained network falls to below 0.015, and is thereby competitive with even the basic OLS (figure 4). We should note that we are using a 10000 data points to train the neural network, while the models illustrated by figure 4 were trained on only 1000 - however, the comparison should be viable — at least roughly — as long as neither of the models are severely overfitted.

The other observation we can make from figure 10 is that the MSEs corresponding to various values of the parameter a overlap all but perfectly for each of the trainig epochs. The leakage thus seems to have no real effect on the training of the network. This is of course except for the "extreme" values $a = 1$ and $a = 10$. We can regard these values as extreme, as leakage part of the RELU function starts to dominate the function for these values. This means that negative activations become common place in the network. It makes intuitive sense that this would disrupt det predictive accuracy of the network, as the function values of the Franke function are generally positive.For completeness, we check whether regularization has any effect on the neural network when we use RELU as activation function. Figure 11 suggests that this is not the case, as the training curves still overlap completely for every value of λ. This is not particularly surprising; there is no apparent reason why the neural network should be more vulnerable to overfitting with the RELU as activation function.

F. Neural Network for Classification

We will now make some of the same considerations for a neural network that is designed and trained to classify handwritten digits of the MNIST data set. Figure 12 illustrates how the accuracy (i.e. the proportion of a number of test cases that the algorithm was able to classify correctly) changes with the number of training epochs, given for the standard set of hyperparameters (table III). One difference to the regression cases stands out immediately: the accuracy for the training data is significantly higher than that for the testing data, at least once the accuracy becomes saturated. We can interpret this as a sign that the neural network is experiencing some overfitting. This is possibly because the amount of

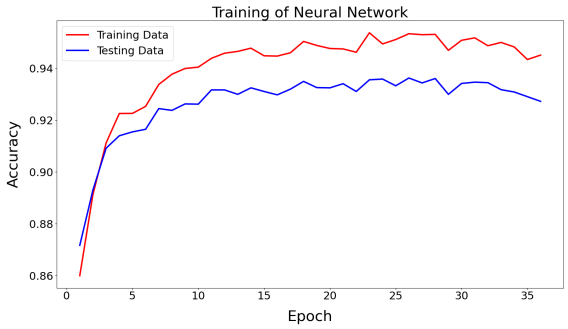

FIG. 12: Evolution of the test accuracy during the process of training the neural network to the MNIST data set.

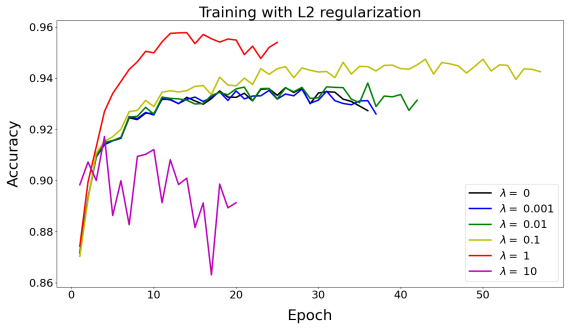

FIG. 13: Evolution of the test accuracy during the process of training the neural network to the MNIST data set, given for various values of the L2 regularization parameter λ. The fluctuations in the case $\lambda = 1$ caused the training to be cut off, perhaps too early, but we haven't rerun it, due to the high computing time.

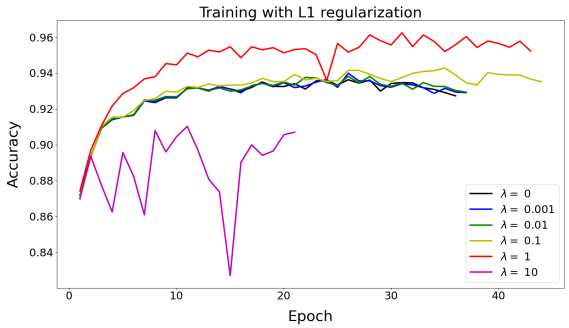

FIG. 14: Evolution of the test accuracy during the process of training the neural network to the MNIST data set, given for various values of the L1 regularization parameter λ.

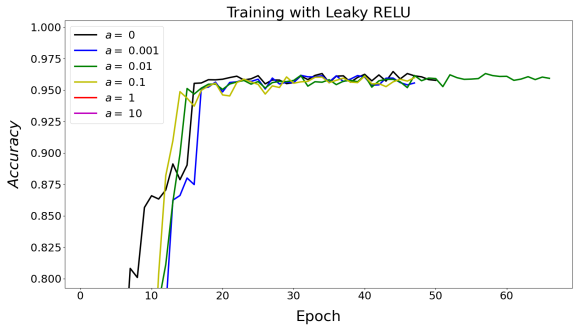

FIG. 15: Evolution of the test accuracy during the process of training the neural network to the MNIST data set. Here, we have used a leaky RELU function with leakage parameter a as activation for the hidden layer. The plots for $a = 1$ and $a = 10$ do not reach high enough accuracies to show up on the graph. We included an L2 regularization parameter $\lambda = 1$

input variables is now 784, rather than 2, which is more comparable to the number of data points that we are training on.

If the algorithm is overfitting the data, then we expect the prediction accuracy to improve when we introduce regularisation parameters. We train the network on the same data for different values of the regularisation parameter λ and display the results in figures 13 and 14. Indeed, for $\lambda = 1$, the accuracy almost increases to 96% for both L2 and L1, an approximate 1.5% improvement from $\lambda = 0$. For $\lambda = 10$ the accuracy fluctuates wildly, indicating that the training process is unstable. It is possible that this could be fixed with a lower learning rate, and it is not out of the question that a higher regularisation parameter could result in still higher accuracies. We haven't been able to try this though, due to the high computation time this requires.

As with the regression case, we test whether we can improve the results by using a (leaky) RELU activation function for the hidden layer, rather than a sigmoid. We illustrate how the accuracies change during the training process in this case in figure 15. It seems as though the results in this case are equally good as for the corresponding case using the sigmoid function (figure 13. This is somewhat surprising to us: it makes sense to think that the RELU is a better choice for regression, since it has the ability to produce an unbounded range of activations. For classification however, the final activations are not supposed to exceed 1, so we would intuitively expect an appropriately bounded activation function to be more suitable in the hidden layer as well. On the other hand, it could be that unboundedness is an inherent advantage in itself, seeing that this allows for a higher range of possible values of all the weights and biases.

Figure 15 also illustrates that the magnitude of the leakage parameter a is inconsequential for the final prediction accuracy of the neural network — except of course for the "extreme" values. This was to be expected, following our discussion on the regression case.

Finally, we also make a quick comparison of the neural network's classification ability with that of a logistic regression algorithm. The accuracy turns out somewhat worse — about 92% versus 95% — than for our neural network, even though we have included the same L2 reg-

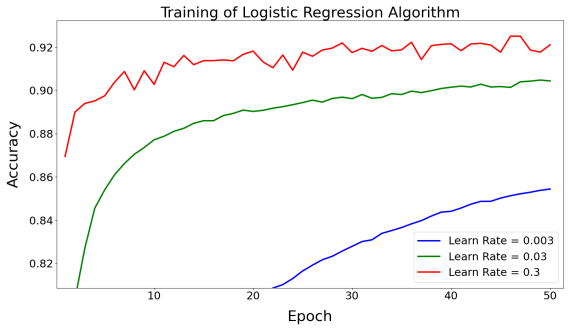

FIG. 16: Evolution of the test accuracy during the process of training the logistic regression algorithm to the MNIST data set. We included an L2 regularization parameter $\lambda = 1$. The number of epochs was limited to 50.

ularization parameter $\lambda = 1$. It is interesting to note that our experimentation with the code indicate that there is little or no change to the results when we increase or decrease λ. This is quite unexpected, seing that the logistic regression algorithm is equivalent to training a neural network, except with no hidden layers. The only feature of logistic regression that we haven't investigated in depth already is that we use the cross entropy cost function, rather than the quadratic cost which we have used throughout for the neural networks. However, we don't see any obvious reasons why the cross entropy would be less sensitive to regularization, or more vulnerable to overfitting.

Another unexpected feature of figure 16 is that the final accuracy attained by the algorithm seems to be dependent on the learning rate, contrary to what we have observed for neural networks so far. Since we have cut off the training processes at 50 epochs for the logistic regression, we don't know for certain what the maximum attainable accuracy is in each case; however it seems clear that the latter is quite different for the different learning rates, given that the curves seem to flatten out at vastly different points. It is not clear to us why the exclusion of a hidden layer or the use of the cross entropy would have this effect.

To round off, we also check how well our algorithms have done compared to more professional implementations. We set up and train a neural network using *Scikit-Learn's* `MLPClassifier` class. To make the comparison as valid as possible, we use the same number of hidden nodes, 30, regularization parameter $\lambda = 1$, choose SGD as optmization algorithm, and the sigmoid function as activation function for the hidden layer. Surprisingly, the accuracy score from `MLPClassifier` is then 0.8072. With $\lambda = 0$ on the other hand, it becomes 0.9264. If we leave the number of hidden nodes at it's default value, 100, the accuracy becomes 0.8056 for $\lambda = 1$ and 0.947 for $\lambda = 0$. It is almost a bit shocking that the "professional" algorithm would behave so differently from ours. Not only does it behave significantly worse with the regularization parameter, rather than better. It also improves significantly by increasing the number of nodes, while the performance of our network seems to be relatively independent of the latter. What could be so different between our algorithm and the one from *Scikit-Learn*? The activation function of the output layer springs to mind, but we have not been able to find what this is for `MLPClassifier`. The initialization of biases and weights may also be different, but this should not have much of an effect on the final result. There is probably a huge range of techniques and variations that we don't yet know that may be applied to `MLPClassifier`. We'll take solace in the fact that the best performance by `MLPClassifier` that we have quoted here is still outperformed by our best neural network, which got accuracies up to 0.96!

VII. CONCLUSION

For basic OLS and Ridge regression, we can see that using a cyclic learning rate improves the results significantly when compared to the non-cyclic one. To determine the cause of this further study outside the time limit for this project would need to take place. For model complexities between three and six the SGD method preforms significantly worse, which could indicate that the algorithm gets stuck in a local minima. This is also something that should be further investigated in future work. Using the FFNN for regression using the Leaky ReLU activation function yiels a lower mean squared error compared to the pure SGD regression, and close to that of the OLS regression. However, using the neural network for this type of regression seems to be a bit of an overkill when the results obtained using classic OLS are as good and less demanding in regards to computing power. For classification we find that introducing a regularization parameter in our FFNN was essential to prevent overfitting. As we have only used the quadratic cost function, it would be interesting to investigate how the network preforms with a different cost function such as the cross-entropy used in the case of logistic regression. When comparing our FFNN's classification abilities to those of Logistic regression we found that the neural network preformed better.

[1] S. S. Adhikari, G. Kluge, and A. Hardersen, "Regression analysis and resampling methods," (2020, acsessed: 2. November 2020), https://github.com/Alidafh/FYS-STK4155/blob/master/project1/report/FYS_STK4155_project1.pdf.
[2] M. Hjorth-Jensen, "Week 40: From stochastic gradient descent to neural networks," (2020, accessed: 2. November 2020), https://github.com/CompPhysics/MachineLearning/blob/master/doc/pub/week40/ipynb/week40.ipynb.
[3] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction; 2nd ed.*, Springer Series in Statistics (Springer, Dordrecht, 2009).
[4] M. Anderssen, *Preformance of Deep Learning in Searches for New Physics Phenomena in Events with Leptons and Missing Transverse Energy with the ATLAS Detector at the LHC*, Master's thesis, UiO (2020), http://www.duo.uio.no/.
[5] M. A. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015).
[6] V. Nair and G. E. Hinton, in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10 (Omnipress, Madison, WI, USA, 2010) p. 807–814.
[7] A. L. Maas, "Rectifier nonlinearities improve neural network acoustic models," (2013, accessed: 13. November 2020), https://www.semanticscholar.org/paper/Rectifier-Nonlinearities-Improve-Neural-Network-Maas/367f2c63a6f6a10b3b64b8729d601e69337ee3cc.
[8] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," (2014), arXiv:1406.2572 [cs.LG].