FYS-STK4155 H22: Project 2
Developing Neural Networks for Regression and Classification problems

Lasse Totland, Domantas Sakalys, Synne Mo Sandnes

(Dated: November 18, 2022)

This report covers the development of Feed Forward Neural Networks (FFNN) implementing back-propagation to solve a regression problem and a classification problem. For the regression problem we used a three-dimensional dataset $z = f(x, y) + \epsilon$ generated from the Franke function with additional normal-distributed noise. The optimal learning rate and regularization parameter with the RELU activation function were determined to be $\eta = 3$ and $\lambda = 0$ by comparing the mean square error (MSE) curves for different configurations. The final network outperformed our predictions in Project 1 after 5000 epochs at $MSE \approx 0.011$ ($MSE \approx 0.012$ using ordinary least squares linear regression) and is predicted to improve further with more training. We have demonstrated that our network converges at around the same MSE as the scikit learn MLPRegressor method. For the classification problem we applied an FFNN and logistic regression to the Wisconsin breast cancer dataset. Despite technical issues with the classification network it appears to have achieved an accuracy of 0.892 with $\eta = 0.0008$ and $\lambda = 11$ using the Tanh activation function. Using a grid search with logistic regression applied to the cancer dataset, an accuracy of 0.89 was achieved with two separate hyperparameter configurations for a prediction on test data.

I. INTRODUCTION

Artificial Neural Networks are a form of deep machine learning that mimics the complex information processing mechanics of the brain; the neurons and their connections. The appeal of deep learning is that an algorithm, with the right tuning, can be trained to perform certain tasks without supervising and may perform them faster or even find useful but complicated patterns we never would have ourselves.

The aim of this project is to further our understanding of these networks by developing our own neural network code for regression and classification problems from scratch, analyse their performance and compare with established methods. Specifically, we'll construct Feed Forward Neural Networks (FFNN) and implement Backpropagation to train the network for a set amount of epochs. This part of the network attempts to minimize a chosen cost-function every epoch by applying Gradient Descent.

For the regression problem we'll use the Franke function data that we used for the analysis in Project 1 [1] so we can compare it with the network's performance. We'll consider the Sigmoid, RELU and Leaky RELU activation functions for this network, as well as learning rate η, regularization parameter λ and amount of epochs, and determine the ideal configuration for minimizing the error. Mean Square Error (MSE) will be used as cost function here.

As a binary classification problem, we'll study the Wisconsin breast cancer dataset [2] and write a network that determines from a set of tumor characteristics whether a patient does or does not have breast cancer. We'll consider the Sigmoid and hyperbolic tangent activation functions for the network. Hyperparameters η and λ will once again be analysed to fine-tune the network. Logistic regression will be applied to compare with our network.

The python code used in the project can be found in our GitHub repository: https://github.com/lassetotl/fys-stk/tree/main/project%202.

II. THEORY

In this section we will be presenting relevant theory for this project. When writing this theory section, we primarily used source [3].

A. Gradient Descent

Gradient descent is an optimization algorithm with goal of finding a minimum of a cost-function by iteration. The basic principle is that a function $F(\mathbf{x})$, where $\mathbf{x}(x_1, x_2, x_3, ..., x_n)$ decreases fastest by moving in the direction of negative gradient of that function $-\Delta f(\mathbf{x})$.

In machine learning the gradient descent is used in order to find the optimal β value. In previous project we have derived cost function as function of β.

$$C(\beta) = \frac{1}{n}||\mathbf{X}\beta - \mathbf{y}||_2^2 \tag{1}$$

where n is the number of datapoints we are working with. By taking the gradient of this cost function with respect of β and expressing it in a vector and matrix form we get

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

As for the Ridge-approximation method, the cost function is defined by equation 3

$$C(\beta) = \frac{1}{n}||\mathbf{X}\beta - \mathbf{y}||_2^2 + \lambda||\beta||^2 \tag{3}$$

And its derivative with respect to β is given in equation 4

$$\nabla_\beta C(\beta) = 2(\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) + \lambda\beta) \tag{4}$$

In order to find the optimal β, we pick random β in the cost function 1 and iterating by taking a small step towards the negative gradient of it. This will lead to the lowest value of Cost function if we do this enough of times. The iteration would take form as

$$\beta_{k+1} = \beta_k - \gamma\nabla_\beta C(\beta) \tag{5}$$

where $k = 0, 1, 2, 3, ...$ and γ is the learning rate. By deciding the value of learning rate we control of how large these steps are. Figure 1 illustrates gradient descent.

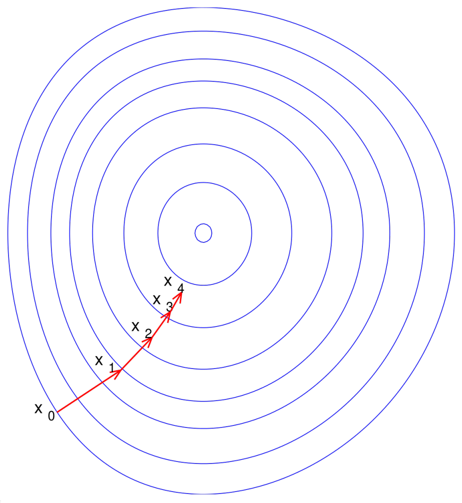

Figure 1. This is an example of a gradient plot. By starting at random point x_0 we find the gradient of the function and walk in the negative direction of this gradient. This will take us to the minima of the function. Taken from [4].

Algorithm for plain gradient descent is provided in algorithm 2 in appendix VI.

B. Momentum in Gradient Descent

We can extend regular gradient descent with momentum. Momentum helps with the search of minima by building inertia in a direction of the gradient. By having an inertia in our algorithm, we overcome oscillations of noisy gradients.

$$\beta_{k+1} = \beta_k - \gamma\nabla_\beta C(\beta) + M \tag{6}$$

Equation 6 is an extension of equation 5. Here, M is the momentum term defined as $M = m\mathbf{v_{t-1}}$ where m is globally defined momentum constant, and $\mathbf{v}$ is the change of the momentum that varies with every loop of calculation. The algorithm for this extension is provided in algorithm 3.

C. Stochastic Gradient Descent

This gradient descent methods splits the data into M sized bathces, called minibatches. In stochastic gradient descent (SGD in short) we pick random minibatches from the data. Then we approximate the gradient by taking the sum over the datpoints in the one of the minibatches that we picked at random in each gradient descent step.

We define the gradient in one minibatch in equation 7

$$\nabla_\beta C(\beta) = \sum_{i=B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i\beta) \tag{7}$$

Here in this equation we sum up all the gradients for all the points $\mathbf{x}_i$ in minibatch B_k.

This randomness can help to avoid getting stuck in a local minima. If it happens so that we end up in a local minima, there is probability that the SGD-method can jump out of this local minima. Additionally, SGD allows us to use larger learning rate. This boosts up the computational speed. As for the difference between regular gradient descent and SGD, Grant Sanderson has a great analogy, which goes like this;

Regular gradient descent is like a well behaved man walking down the hill that takes small and careful steps. While stochastic gradient descent is like a drunk man stumbling down the hill with large steps. Both of these men will end up at the bottom, but only one of them will come there quicker - the drunk man. This is why we use large learn-rate in SGD.

Algorithm of this method is provided in algorithm 4 in the appendix section.

D. Adagrad method

Adaptive gradient descent method (adagrad for short) is a regular gradient descent with an adaptive learning rate. This means, that the learning rate constant γ in equation 5 adapts with every calculation loop with

$$\gamma_{\text{new}} = \frac{\gamma}{\sqrt{\epsilon + \text{diag}(\mathbf{G})}} \tag{8}$$

Where G is a matrix constructed by the gradients

$$\sum_{\tau=1}^{t} g_\tau g_\tau^T \tag{9}$$

While ϵ is a very small number that prevents division by zero. Algorithm of this method is provided in appendix section as algorithm 5

E. Logistic regression

Logistic regression calculates the probability of an event occurring. When using logistic regression, we most often encounter two alternative outcomes, which is generally expressed as a binary outcome (true or false). Because the outcome is a probability, the dependent variable ranges from 0 to 1. A logit transformation is applied on the odds in logistic regression, which is the probability of success divided by the probability of failure. [5] Logistic regression will also act as a springboard for neural network algorithms and supervised deep learning. The minimization of the cost function in logistic learning results in a non-linear equation in the parameters ($\hat{\beta}$). Minimization algorithms are thus required for optimization of the problem.

F. Neural Networks

Artificial neural networks (ANN) are computational systems that, mainly without any task-specific rules encoded into them, may learn to perform tasks by considering examples. It is meant to resemble a biological system in which neurons communicate with one another by delivering mathematical functions as signals between layers. Each connection between neurons in a layer is represented by a weight variable, and any number of neurons may be present in a layer.

With a long history of development, artificial neural networks are directly related to the progress of computer science and computers in general. To understand how the brain processes signals, McCulloch and Pitts created an artificial neuron model in 1943. Since then, several researchers have enhanced the model.

The main goal is to emulate neural networks found in the human brain, which is made up of billions of neurons that exchange electrical signals with one another. To produce an output, a neuron's incoming signals must rise above its activation threshold. If the threshold is not exceeded, the neuron remains dormant and produces nothing.

This behaviour has served as the basis for a manageable mathematical model of an artificial neuron

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{10}$$

where the neuron's output y is the result of its activation function f, which takes a weighted sum of signals as input.

Conceptually, it is advantageous to categorize neural networks into the following four groups:

1. general purpose neural networks for supervised learning
2. neural networks designed specifically for image processing
3. neural networks for sequential data
4. neural networks for unsupervised learning

G. Multilayer perceptrons

Multilayer perceptron (MLP) networks are fully-connected feed-forward neural networks made up of a minimum of three layers. They consist of an input and an output layer, as well as one or more hidden layers. The layers are composed of neurons with non-linear activation functions. See Figure 2 for a sketch of what such a network may look like.

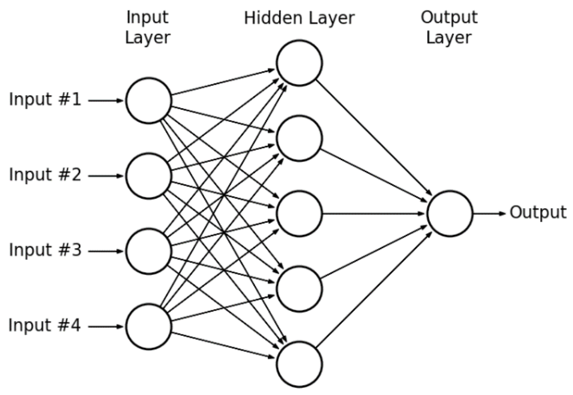

Figure 2. A hypothetical example of Multilayer Perceptron Network with one output. All of the arrows represent weights between nodes, signifying their correlation. A network with several outputs would represent a classification network with several categories. Taken from [6].

A continuous multidimensional function can be approximated to arbitrary accuracy for feed-forward neural networks with only one hidden layer. We have here assumed that the hidden layer's activation function is a non-constant, bounded and monotonically-increasing continuous function, and that the hidden layer has a finite number of neurons. This is a result of the Universal Approximation Theorem.

The Universal Approximation Theorem states that continuous functions on compact subsets of real functions can be approximated by an FFNN with only one hidden layer with a finite number of neurons. With appropriate parameters, simple neural networks can represent a variety of useful functions. The multilayer feedforward architecture enables the neural networks to be universal approximators. This means that there is a network containing a specific number of neurons in the hidden layer that can estimate almost any known function. In other words, the outcome can be approximated for any number of inputs and outputs, regardless of what f(x) is [7].

It should be noted that the activation function restrictions only apply to the hidden layer; the output nodes are always presumed to be linear in order to not limit the range of output values.

The activation function generates the output y

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z), \tag{11}$$

where the output of the neurons from the previous layer in a FFNN is the x_i that is inserted in the function. The weighted sum of the outputs of all the neurons in a layer is given to the neuron in the following layer as a result of MLP's beeing fully connected.

Each layer in an MLP with only linear activation functions performs a linear transformation of its inputs. The NN's output will be a linear function of the inputs no matter how many layers there are. To accommodate non-linear functions, we must include some type of nonlinearity into the NN. The logistic Sigmoid function and the hyperbolic tangent function are examples of nonlinearity introduced to the NN in order to fit the nonlinearity. The logistic Sigmoid function is given by

$$f(x) = \frac{1}{1+e^{-x}}, \tag{12}$$

and the hyperbolic tangent function is given by

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{13}$$

The rectified linear unit (RELU) activation function can be defined as

$$f(x) = \begin{cases} \alpha(e^x - 1), & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}, \tag{14}$$

where a constant $\alpha = 0$ represents a standard RELU activation function and $\alpha = 0.01$ represents a 'leaky' RELU. The leaky variant is useful to prevent cases where the RELU is inactive for most inputs and giving us vanishing gradients.

The Softmax activation function takes a vector x of K real numbers and converts them to a probability distribution summing up to 1, and is defined like so for a given index $i \in 1, ..., K$:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}. \tag{15}$$

The choice of activation functions characterizes a neural network. MLP's are flexible, shown by the structure of their equations. They are built up as a sum of scaled activation functions, with weights and biases c_i that can be adjusted. This allows a change of slope or rescaling of the activation function.

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \tag{16}$$

The basis functions for neural networks, which correspond to the weights, are learnt from the data. This is not the case for other methods like linear and logistic regression. The name "hidden layers" refers to the fact that they are not related to the observables.

Having the input $\hat{x}$ we can define the activation $\hat{z}$ of a neuron j as the basis expansion of the input. The activation of the neuron on the l-th layer is defined as a function of the bias b_k^l, the outputs $\hat{a}^{l-1}$ and the weights added from the previous layer

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l, \tag{17}$$

where M_{l-1} represents the total number of neurons for the previous layer.

H. Back-propagation algorithm

In order to create a reliable model, our neural network needs to be trained properly. This can be done using the back-propagation algorithm. Back-propagation is a method of propagating the total loss back into the neural network to determine how much of the loss is attributable to each node, and then altering the weights to minimize the loss [8]. In order to use the algorithm, the expressions for the derivative of the cost function C and the activation function σ must be known. Depending on the problem, the cost-function will differ.

Algorithm 1 shows pseudo-code representation of all the steps in the back-propagation training procedure (for one hidden layer) we'll be using in all of our networks.

Algorithm 1 Back-propagation

```
$y \leftarrow$ target dataset
$X \leftarrow$ input
$C \leftarrow$ cost function
$N \leftarrow$ number of epochs
$\eta \leftarrow$ learning-rate
$\lambda \leftarrow$ regularization parameter
$w, b \leftarrow$ initial weights, bias
procedure TRAINING NETWORK(update w and b)
    for N do
        a $\leftarrow$ Feed Forward output
        $\delta_o = dC(y, a)/da \leftarrow$ output error
        $\delta_h = \delta_o * w_o^T \leftarrow$ hidden error
        $g_{ow},\ g_{hw},\ g_{ob},\ g_{hb} \leftarrow$ get cost gradients
        if $\lambda \neq 0$ then
            $g_o \mathrel{+}= \lambda * w_o$
            $g_h \mathrel{+}= \lambda * w_h$
        end if
        $w_o \mathrel{-}= \eta * g_{ow}$
        $w_h \mathrel{-}= \eta * g_{hw}$
        $b_o \mathrel{-}= \eta * g_{ob}$
        $b_h \mathrel{-}= \eta * g_{hb}$
    end for
end procedure
```

I. Cost functions

In machine learning, loss helps us understand the difference between the predicted value and the actual value. The cost/loss function will calculate the absolute difference between each prediction we make and the actual value, it does not matter if our predictions were too high or too low. [9]. Further explanation of cost functions can be found in [1].

When implementing back-propagation into our neural networks, we need an appropriate cost function to evaluate the performance of the model compared to the target data to be able to update the gradients.

1. *Mean Square Error*

The Mean Square Error (MSE) of a predicted model $\tilde{y}$, given the observed data y is defined as

$$MSE(y,\ \tilde{y}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2, \tag{18}$$

where n is the number of data points. The point of finding the mean *squared* error is that squaring reduces the complexity given by negative values. This is a conventional choice as cost function for regression networks.

What MSE does is that it takes each point of our approximated data, and subtracts it from corresponding real data points. The purpose of MSE is then to evaluate the difference between these two data points. If the difference is large, we get a high value of MSE, which means that our approximated data is a poor approximation, because it differs from the real data.

2. *Binary Cross-Entropy loss*

For a binary classification case where the results are either of two outputs we'll use the Binary Cross-Entropy loss function, also called the 'logarithmic loss function', to evaluate our classification networks. The function for a predicted model $\tilde{y}$, given the observed data y is defined as

$$L(y,\ \tilde{y}) = -\frac{1}{n}\sum_{i=1}^{n} y_i log(\tilde{y}_i) + (1 - y_i)log(1 - \tilde{y}_i), \tag{19}$$

where n is the number of data points.

III. METHOD

A. Comparison of gradient descent methods

We wish to study different methods of the gradient descents. In theory section we have introduced plain gradient descent, stochastic gradient descent (II A, II C) and two extensions. We wish to look deeper into these two methods and compare their extensions to find the most reliable one. In order to do this, a polynomial 20 will be used. We will first use ordinary least squares method to find the gradients to this polynomial.

$$P(x) = -0.5x^3 + 1.5x^2 + 1.5x + 3N \tag{20}$$

Where N is added noise to our data.

1. *Plain and Stochastic gradient descent*

The goal of this method is to find optimal θ-value which gives the global minima in cost function defined in equation 1. Here the learning rate can be chosen freely as a hyperparameter. However, we start off by choosing it to be 0.001, and we choose to analyse this with up to 70 iterations. We then calculate the mean square error (MSE) for different numbers of iterations, i.e. we create a function of MSE with the number of iterations as the variable. We do this for the following methods,

- Plain gradient descent (GD)
- GD with momentum
- GD with adagrad
- GD with adagrad and momentum

The same analysis is performed on the following algorithms

- Regular stochastic gradient descent (SGD)
- SGD with momentum
- SGD with adagrad
- SGD with adagrad and momentum

However, the learning rate is chosen to be 0.01 this time. The values of MSE is chosen to be a function of number of epochs, we choose to analyse with up to 70 epochs. As for the size of minibatches, it is chosen to be 10.

2. *Gradient descent and Ridge approximation*

Depending on the results of precious subsection, we will pick one method which is favorable for our situation, and preform ridge approximation instead of ordinary least square. Until now, we have chosen specific values of learning rate. We now want to investigate various learning rates and examine how the polynomial's (20) approximation affects them.

The learning rate values we are going to test are 10 values ranging from 10^{-3} to 1 on a logarithmic scale. Since we are performing a ridge approximation, we also test different values of λ. For these values, we test with the same values as the for learning rate. After that, we scan through all of these values and calculate the MSE. We save this MSE, and generate a heatmap that compares how well these various values of λ and learning rate do.

B. Developing a regression neural network

This section covers the development of a Feed Forward Neural Network (FFNN), with implemented backpropagation, applied to a regression problem. The final code for the network in the form of a class called '*FFNN_Regression*' is based off of the python script presented in the lecture notes of the FYS-STK 4155 course (section 14.2.9) [3].

The data we are going to use was generated from the Franke function, giving a three-dimensional dataset, in Project 1. Specifically, the data was generated by inserting uniformly generated points $x, y \in [0, 1]$ into the Franke function $f(x, y)$ and then adding stochastic, normal-distributed noise $\epsilon \sim N(0,\ \sigma^2)$. We define the dataset as:

$$z = f(x, y) + \epsilon, \tag{21}$$

assuming a variance of $\sigma^2 = 0.1$. The 'z' has the form of a two-dimensional 40 x 40 matrix corresponding to the (x, y) points. A corresponding design matrix X from project 1 is loaded as well ('.npy' files for z and X are saved in the 'misc code' folder in the github repository) and 80% of z and X are assigned as training data, the 20% remaining as test data.

After splitting, the training set of z ('z_train') is flattened to an array of 1280 points of altitude values, which will serve as the nodes in the input layer, the first step in the 'Feed Forward' part of the network architecture. Similarly, 'X_train' is defined with 21 columns/features (because the initial design matrix is produced for a fifth polynomial degree) and 1280 rows corresponding to each input.

As visualized in Figure 2, each input node connects to every node in the hidden layer. Each connecting arrow in the figure represent a weight influencing the correlation of the values in the hidden and the input nodes, where a connection of small weight has little correlation, vice versa. The importance of a node is also determined by an activation function that decides quantitatively how 'active' a node is. We'll use and compare the Sigmoid (12), RELU and Leaky RELU (14) activation functions for the hidden layers. No activation function is required for the output layer in a regression network, as we don't want to limit the output to values between 0 and 1. For the regression case the feed forward pass produces one output with the same dimensions as the z_train array.

The initial weights in the hidden layer are generated randomly (normally distributed $N(0,\ \sigma^2)$ with variance $\sigma^2 = 0.01$) in the form of a matrix with 20 rows for each hidden node and 21 columns for each feature. The output layer has one node and thus one column of weights generated the same way. Each node in the hidden and output layer is affected by an added bias of 0.01. The method '*create_biases_and_weights*()' is used in the script when creating an instance of the FFNN class to generate these matrices with initial weights and biases for the separate networks so that each network starts training from scratch and isn't influenced by previous training of others, which could happen with global variables.

The first feed-forward pass will likely give a poor output that we can evaluate with a cost function, and we need to implement back propagation to optimize this cost function with gradient descent.

We choose the MSE (18) as cost function for the regression network. The sum in the MSE function corresponds to the nodes of the previous layer summed over to calculate their collective output. When calculating the output error δ_o as defined in Algorithm 1 in the back-propagation algorithm we need the cost function derivated over the output, simply giving us the difference between the observed and predicted data divided by the amount of inputs n (ignoring a factor 2 because we're using random, constantly changing weights either way).

With the network set up and running as intended, and saving the MSE at every epoch, we can determine ideal learning rate η and regularization parameter λ by plotting several MSE curves and comparing their learning. We'll compare effectiveness, stability and their final MSE to determine what parameters we want to use in our regression network.

We will also attempt to construct the same Feed Forward regression network with backpropagation using the scikit learn method 'MLPRegressor' to compare with our original script.

With the regression network functional, we are going to compare the MSE with our results from Project 1, where OLS regression performed the best on data defined exactly as we have z in Equation 21. From Figure 11 in Appendix B we see that the OLS regression settled at $MSE \approx 0.012$ for higher polynomial degree fits to the training data. We will use this value to make the comparison with the performance of the regression network.

C. Developing a classification neural network

The dataset we'll use for the classification problem is the Wisconsin cancer dataset, accessed in the script with the sklearn 'datasets' method. This is a binary classification case where every 569 instances, in this case with 32 features each with values corresponding to various tumor characteristics, have a corresponding value 0 ($False$) or 1 ($True$) whether or not the patient is diagnosed with breast cancer.

For this analysis we extract just four of the features from the dataset: texture, perimeter, compactness and symmetry. This means we construct a design matrix X with dimensions (569, 4) and a target column matrix y of length 569.

For the network we're still going to use an FFNN with back-propagation like we did for the regression network, but there are a few key differences in the network architecture.

Firstly the output layer now features two nodes, each containing the probability of the result being either 0 or 1. The Softmax activation function as defined in Equation (15) is applied on the output layer to make sure that the two probabilities add up to 1. We'll also use the Binary Cross-Entropy Loss function as defined in Equation (19) as cost function, and the output error for the back-propagation algorithm we find to be:

$$\delta_o = \frac{dC}{da} = \frac{\tilde{y} - y}{\tilde{y}(1-\tilde{y})}.$$

The network architecture is unchanged from the regression network otherwise.

To evaluate the performance we'll save the loss and accuracy score (simply correlation between model and observed data, divided by number of inputs) for every epoch of training. We'll compare the training for the Sigmoid and the hyperbolic tangent (Tanh, eq. 13) activation functions for the hidden layer. As with the regression network, we'll perform a hyperparameter analysis by comparing accuracy development for various configurations. We'll also compare the results with a similar network constructed with the scikit learn method 'MLP-Classifier', and later with a logistic regression code written by us.

D. Logistic regression with SGD

We will now use the same breast cancer data set provided by the sklearn dataset module. We have 30 features from which we wish to perform a binary classification. Meaning that by taking in these 30 features, we wish to classify the breast mass to either be breast cancer or not. For this section, we are going to perform logistic regression analysis.

We will be using Stochastic gradient descent in order to find the most optimal values for weights and biases. We will also include Ridge-approximation. This means that we will have λ values as our hyper-parameter. In this case, we will also have learning rate as another hyper-parameter. Using heatmap visualization, we will examine how these two hyper-parameters affect our logistic regression. The values of learning rate that we are going to test, are 10 values ranging from 10^3 to 1 on a logarithmic scale. Since we are performing a ridge approximation, we also test different values of λ. For these values, we test with the same values as the learning rate.

We start off by splitting the data into test data and train data. Where 80% of all data is training data, and the rest is test data. We then create a design matrix X where the columns are these 30 features, and the rows are just data points from different patients. Then for each different value of these two hyper-parameters in test data, we will perform SGD to find the optimal weight and bias. Here, we use 70 epochs, and choose the size of a minibatch to be 10.

After finding the optimal weights and biases with SGD method, we define a variable given as

$$X * (w + b)$$

where X is the design matrix which include the test data, and w and b is the found optimal weights and biases. We put this variable into the Sigmoid function which is defined in equation 12 and get the probability of patient having a breast cancer as an output.

Since we are doing a logistic regression, the output we get is a number between 0 and 1. We wish to have an output that can only be either 0 or 1. Therefore, we process the output to be 0 if the Sigmoid output is less the 0.5, and 1 if the Sigmoid output is larger or equal to 0.5.

This will be done for every different value of learning rate and λ. From the sklearn dataset, we have also gotten the target data. We will be calculating the accuracy of our predictions by comparing it with the target data. Target data holds all the true values. The sklearn metrics module will be used to calculate the accuracy score

For comparison of the results, we use Scikit-Learn's logistic regression module.

IV. RESULTS & DISCUSSION

A. Gradient descent

1. *Plain and Stochastic gradient descent*

By choosing learning rate to be 0.001 and plotting plain gradient descent with its extensions as comparison, we get the plot which is shown in figure 3

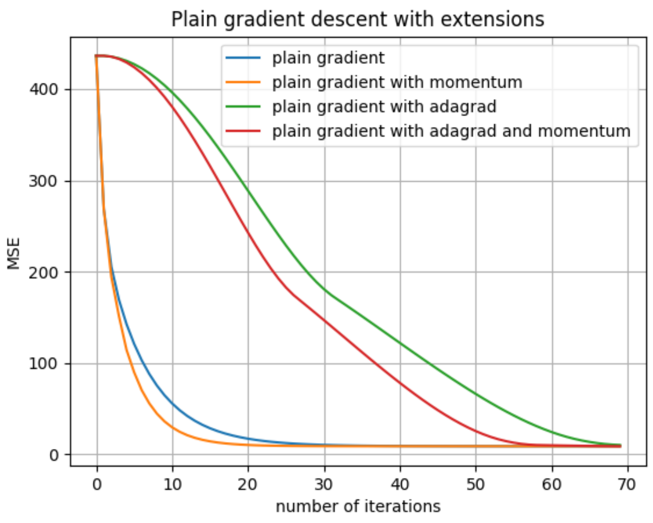

Figure 3. Comparison of 4 plain gradient descent methods

In figure 3 it is clear to see that plain gradient with momentum builds momentum quickly, and runs towards low MSE quicker than any other methods. From the figure, it also seems that Adagrad method takes larger number iterations to fall to the minima in the cost-function. As for the lowest values of MSE for each method, table I is provided.

From table I we can see that all the methods ends up with approximetely the same fit with a given time. However, plain gradient with adagrad ends up with the highest MSE.

As for the stochastic gradient descent, we plot the

Table I. MSE in plain gradient descent methods

Method	Lowest MSE
Plain GD	8.85
Plain GD with moment.	8.83
Plain GD with adagrad	10.23
Plain GD with adagrad and moment.	8.86

MSE with number of epochs. Here we have chosen the learning rate to be 0.01 and size of a minibatch to be 10. Figure 4 shows the comparison of SGD and its expansions.

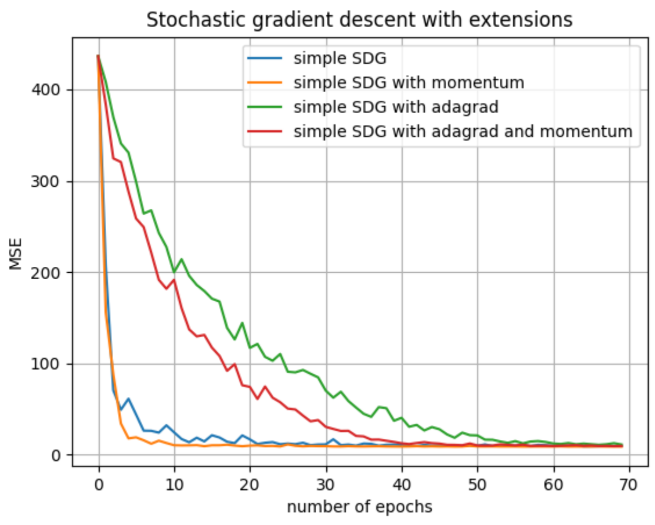

Figure 4. Comparison of 4 stochastic gradient descents.

From figure 4 we can see the same pattern as in figure 3. SGD with momentum comes quickest to the lowest MSE. The corresponsing lowest MSE values for each method i provided in table II

Table II. MSE in stochastic gradient descent methods

Method	Lowest MSE
Regular SGD	9.17
SGD with moment.	8.88
SGD with adagrad	10.857
SGD with adagrad and moment.	9.36

From table II we can see that the SGD follows the same tendency as in GD. One major difference however, is the that SGD fluctuates rapidly as it descends towards the minima in the cost function. We can also see that it descends more quickly. This is something expected, since we have used larger learning rate for SGD. This shows that by using SGD, we save a large amount of computational time.

2. *Gradient descent and Ridge approximation*

We chose the stochastic gradient descent with momentum as a favorable method. This is because it is quick in computation, and has relatively low MSE. This method has been analysed with ridge regression method as for comparing the different values of learning rate, and λ. The number of epochs has been chosen to be 70. And the size of minibatch to be 10. Figure 5 shows the results of this comparison of these hyper-parameters.

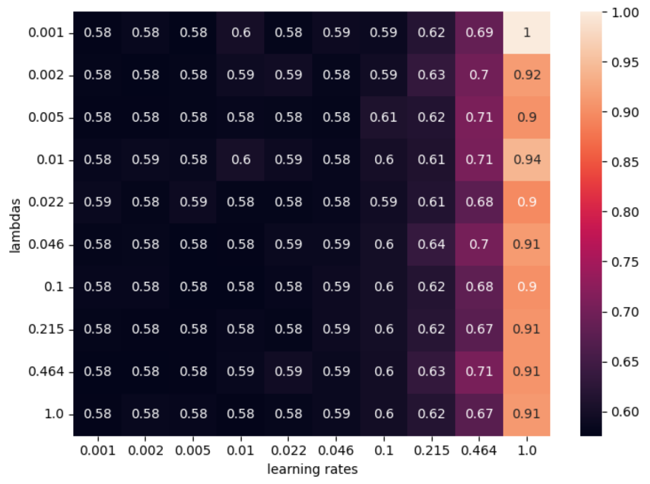

Figure 5. Comparison of λ and learning rate values on Stochastic gradient descent with momentum.

From figure 5 we can see that as long as the learning rate is below value of 0.1, we get a relatively good approximation. As for the λ values, we can see that they do not have a large affect on the approximation.

B. Regression neural network results

To tune the RELU regression network we have performed a parameter analysis, Figure 12 in Appendix B shows that the choice of regularization parameter is seemingly inconsequential at 100 or more epochs as long as it isn't too large ($\lambda = 0.01$ performs significantly worse), so we just chose $\lambda = 0$. Increasing the learning rate leads to the error converging earlier, but they seem to eventually converge toward around the same value. We chose $\eta = 3$ as higher values seem to give less stable descents based on the $\eta = 4$ curve. The network using the Sigmoid converges much quicker and uses a learning rate of $\eta = 0.01$.

To get a more comprehensive impression of the influence of the relation of the parameters on the final network we could could have used a grid-search with a heatmap to present the performance of each combination of λ and η, as in the lecture notes. However, the impression we have from our own analysis is that being more precise than we already have been won't influence the network that much.

Using these parameters along with one hidden layer with 20 nodes, we compared the mean square error for Sigmoid, RELU and Leaky RELU activation functions as functions of the epochs used. These MSE curves are plotted in Figure 6. The figure shows us that the Sigmoid network converges at around $MSE = 0.073$ (rounded to three decimals) after 100 epochs, and the RELU networks stop at $MSE = 0.011$ after 5000 epochs without fully converging yet. The X-axis is log-scaled because of how slow the RELU networks converge.

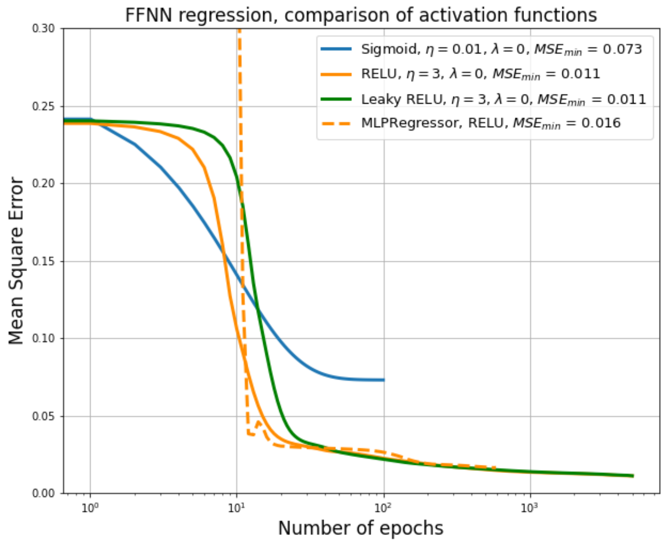

Figure 6. Development of mean square error for regression networks using different activation functions. Solid lines represent our own FFNN script. The network using the sigmoid function converges quickly with a low learning rate and its training is stopped after only 100 epochs. The RELU networks trained for 5000 epochs. A network constructed with the MLPRegressor method, also using RELU and corresponding parameters λ and η, is included for comparison.

Firstly, it is obvious that the Sigmoid network is performing the worst. Even though it is convenient that it converges so quickly, the final MSE score of 0.073 is significantly outperformed even by a first degree polynomial OLS fit ($MSE \approx 0.035$) as seen in Figure 11. We determine this network to be insufficient for our purposes.

The performance of the RELU and Leaky RELU networks are much more impressive, even though they take longer to train. For the range of epochs used by us the two of them give very similar results so further discussion about the 'RELU network' regards them both. At 5000 epochs they reduce the MSE down to 0.011 and the network hasn't even entirely converged yet at that point. In other words, the RELU regression network slightly outperforms our OLS regression from project 1 ($MSE \approx 0.012$) at 5000 epochs, and we assume it still has some room for improvement by training for more epochs until it actually converges.

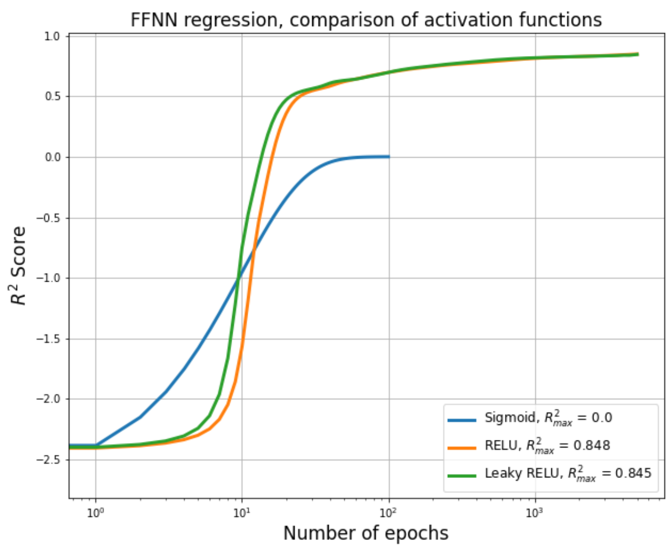

Figure 7. Development of R^2 score for regression networks corresponding to Figure 6.

In Figure 7 we see a plot of the R^2 score function corresponding to the same networks. The first thing to note, when comparing with the score with the OLS method seen in Figure 11 is that the initial score is negative, demonstrating the poor first 'guess' from the network. The Sigmoid network converges at $R^2 = 0.0$ which signifies a really poor prediction. The RELU network performs the best at 5000 epochs with $R^2 = 0.848$, still not having converged yet. The Leaky RELU network is right behind at $R^2 = 0.845$.

An interesting thing with these results is that the RELU network results seem to have both a lower MSE and a lower R^2 score than our OLS results ($R^2 \approx 0.89$ at its best). This may seem counter-intuitive, but these are not necessarily corresponding evaluation functions. The MSE evaluates deviation from the target data (squared), and the score determines correlation. Either way we observe that both OLS and the RELU regression network are viable methods for approximating noisy Franke data, and that the network has improving potential with further training.

Despite the fact that our results seem good, the way we trained the network around the fifth degree polynomial fit design matrix X could be detrimental to its performance when applied to other datasets. Simply put, we give the network a bias towards suggesting a fifth degree polynomial for the approximation. In this case it isn't a poor fit but the network could have found a good fit on its own, per the Universal Approximation Theorem. For a case of a different terrain dataset that we're not as familiar with, the prediction could be really poor as a result. A more flexible choice would be to just chose the x and y data corresponding to the z values and let the network do all of the work.

Figure 6 also shows a comparison between the RELU network MSE curve and a scikit learn MLPRegressor method implementation of a RELU network with the same parameters. Notice how its initialized weights (we could not define these ourselves) gives such a large initial MSE that the network needs 10 epochs to reach around the error range of the other curves.

The 'adam' solver was used because of a technical issue with the stochastic gradient descent ('sgd') solver where the error diverged and increased every epoch. This method stops training on its own earlier when the the loss hasn't decreased sufficiently for 10 epochs (default) within a defined tolerance.

The MLPRegressor method's performance was generally more unstable than our own code, and the error would suddenly diverge after enough epochs. The dashed curve seen in Figure 6 is from a hand-picked random seed ('np.random.seed(0)' in general and 'random_state = 0' as a MLPRegressor argument) where its performance was more stable.

Even though the MLPRegressor network has a much higher initial MSE than our own RELU network, it does start converging more smoothly at about the same time (at around 10 epochs). From that point onwards it seems to agree fairly well with our own RELU network, but stops automatically at around 600 epochs. Forcing it to continue for further epochs in this random seed leads to it diverging dramatically, so we just let it stop automatically. We can still conclude that our own network code seems to agree fairly well with established methods despite the instability of the MLPRegressor network.

We did not succeed in making our regression class able to construct networks with arbitrary numbers of hidden layers within the deadline, though one layer seems to be enough to fit the Franke data.

C. Classification neural network results

The classification network class 'FFNN_Classification' does not appear to be as refined as the regression network class we've made. It does produce some results, and though we're not as confident that they're correct we'll still analyse them.

In Figure 8 we see the accuracy development of the classification networks using the Sigmoid (20 hidden neurons) and Tanh (30 hidden neurons) activation functions. We observe that the accuracy does gradually improve as the networks train across the epochs, but that they don't converge as we expected. In this case the best accuracies for both networks are about 0.89, and it would be possible to save the outputs from these epochs but it is not especially sophisticated compared to having the accuracy converge (which would let us determine if more training is useful or not, like in our regression analysis).

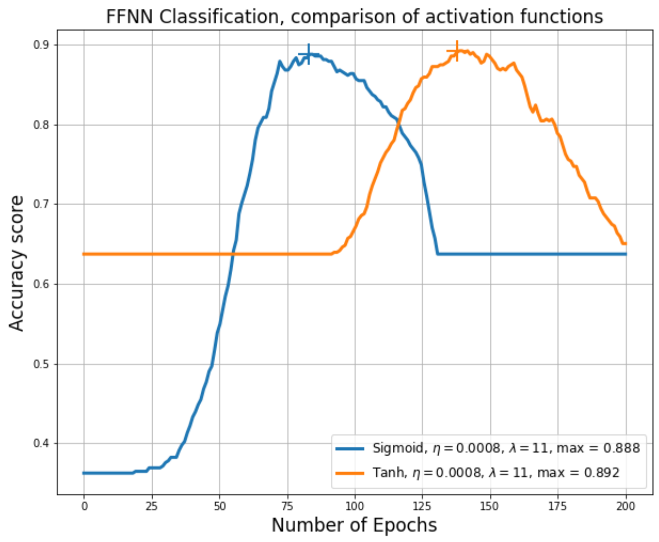

Figure 8. Development of accuracy for classification networks approximating to the Wisconsin breast cancer dataset using four features. The peaks of each curve is marked, the maximum accuracies presented in the labels.

Another bug that is visible here is how different networks (entirely different objects in the python script) seem to be affecting eachother. When running the script, the Sigmoid network trains first, and then we see that the next network seems to start training with the weights that the previous network finished with. Reversing the order (letting the Tanh network train first, this is demonstrated in Figure 13 in Appendix B) gives an entirely different result, which is odd as it was never an issue with the regression class that is structured very similarly. This issue prevents us from doing meaningful hyperparameter analyisis, though our attempt can be seen in the notebook for the classification analysis. The configurations used by the network were therefore determined by just experimenting and attempting different values.

When plotting the loss as function of epochs for the same networks training (Figure 9) we see that they both seem to converge within 100 epochs, though the difference between the initial and final values are minuscule. When the cost function of a network converges it is usually a sign that further training won't change anything, but in the accuracy plot we clearly see this not to be the case.

We also wrote short corresponding networks with tensorflow keras ('sequential()'), and scikit learn ('MLPClassifier()') which both gave validation scores of about 0.588. All in all, the maximum accuracies from Figure 8 are a little suspect, but we'll proceed the analysis with these values either way.

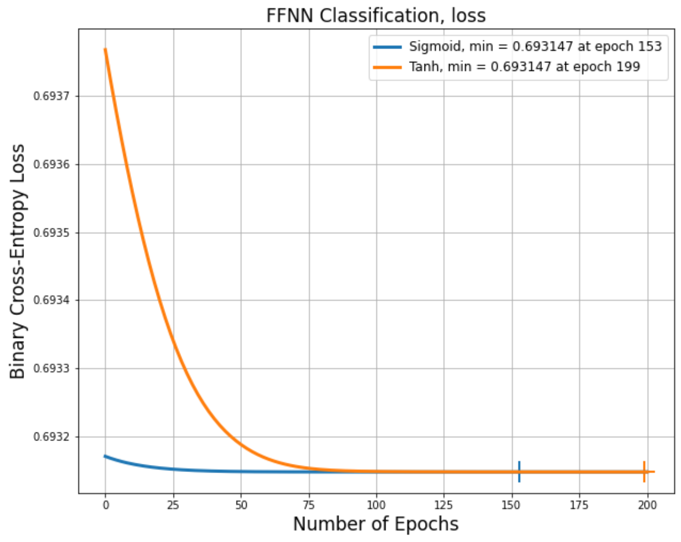

Figure 9. Loss as function of epochs for the classification networks.

D. Logistic regression with SGD

By comparing the different values of learning rate and λ which are stated in the method section, we get the following heatmap plot which is shown in figure 10

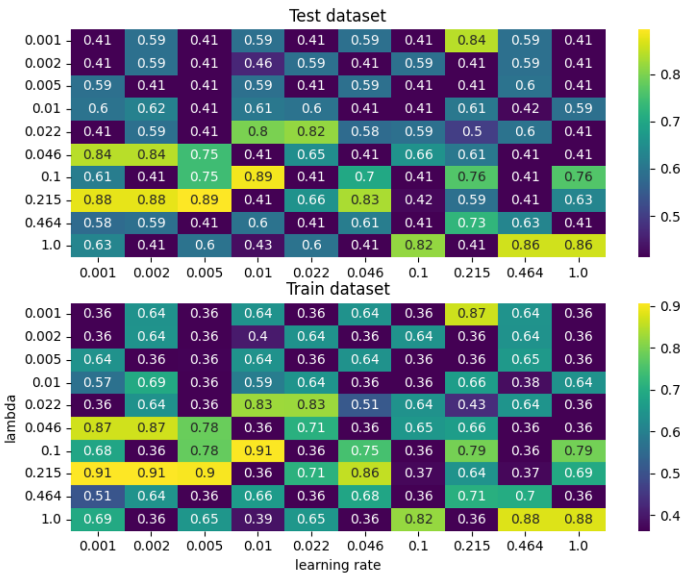

Figure 10. Comparison between different values of λ and learning rate on test set, and training set. Here on the x-axis are the learning rate values, and on the y-axis are the lambda values.

from figure 10 we can see that there are couple of value combinations of λ and learning rate that gives high accuracy score. For example if we have learning rate set as 0.01, and $\lambda = 0.1$, we get fairly high accuracy score. We have been calculating with 455 number of patient data in the training dataset. If we however would increase the number of data, we could train our logistic regression to become more precise. Resulting the accuracy score to become higher.

As for the comparison with Scikit-Learn module, it gives an accuracy rate of 0.93. As for our build logistic regression, the highest accuracy we get is at 0.89 at test data set, and 0.91 at train data set. This means that our build logistic regression approximation is fairly good in comparison with Scikit-learn, but not as good. One reason for this could be that we have used regular stochastic gradient descent method. We have seen from previous result that Stochastic gradient descent with momentum gives better results. If we would have used this method, we might have gotten results as good as Scikit-learn's.

As for the comparisson with FFNN-method, the accuracy score was achieved to be 0.89, which is the same as we have achieved with the test data set by using logistic regression with SGD.

V. CONCLUSION

After comparing our results, we found that for both the plain gradient descent and the regular stochastic gradient descent, the method with momentum descended quickest to the lowest MSE. The mean square error for the SGD with momentum was not significantly lower than for the GD with momentum, but we did find SGD more favorable since it requires less epochs.

The RELU and Leaky RELU regression networks performed marginally better than our OLS regression from project 1 at 5000 epochs, and we believe there is still some possibility for improvement by training for more epochs until it converges. Due of time constraints, we were unable to provide our regression class with the ability to construct networks with any number of hidden layers.

For the classification case, the network appears to have reached an accuracy of 0.892 with $\eta = 0.0008$ and $\lambda = 11$ using the hyperbolic tangent activation function. However, our networks did not behave as we expected them to, and we fear that our results might be inaccurate due to flaws.

In comparison to Scikit-learn, our logistic regression approximation is decent but not as good. We might have achieved a higher accuracy using the stochastic gradient descent with momentum instead, as our previous findings demonstrate that this method produces more favorable outcomes. By gathering more data we could train our logistic regression and improve its accuracy.

VI. APPENDIX A: GRADIENT DESCENT ALGORITHM COLLECTION

Algorithm 2 Plain gradient descent

```
$N \leftarrow$ number of iterations
$\theta \leftarrow$ Choose random point in cost-function
$\gamma \leftarrow$ learning-rate
for N do
    $g \leftarrow$ gradient
    $\theta = \theta - \gamma * g$
end for
```

Algorithm 3 Plain gradient descent with momentum

```
$N \leftarrow$ number of iterations
$\theta \leftarrow$ Choose random point in cost-function
change $\leftarrow 0$
mom $\leftarrow$ momentum
$\gamma \leftarrow$ learning-rate
for N do
    $g \leftarrow$ gradient
    changeNew $= \gamma * g(\text{mom}) * \text{change}$
    $\theta = \theta -$ changeNew
    change = changeNew
end for
```

Algorithm 4 Stochastic Gradient Descent

```
$X \leftarrow$ Design matrix
$y \leftarrow$ target data
$M \leftarrow$ define size of each minibatch
$m \leftarrow$ define number of minibatches
n_epochs $\leftarrow$ number of epochs
$\gamma \leftarrow$ learning rate
for n_epochs do
    for m do
        $X_k \leftarrow$ random minibatch from X
        $y_k \leftarrow$ random minibatch from y
        $g \leftarrow$ find gradient for these minibatches
        $\theta = \theta - \gamma g$
    end for
end for
```

VII. APPENDIX B: ADDITIONAL FIGURES

[1] S. M. S. Lasse Totland, Domantas Sakalys, "Modelling 3d terrain using linear regression," (2022).

Algorithm 5 Adagrad method

```
$\epsilon \leftarrow$ small number to prevent division by zero
$G \leftarrow$ nxn matrix where n is the number of features
$\gamma \leftarrow$ Learning rate
**for i** in number of iterations **do**
    **for i do**
        $g \leftarrow$ find the gradient
        $G = g * g^T$
        $\gamma = \frac{\gamma}{\epsilon + \sqrt{\text{diagonal(G)}}}$
        $\theta = \theta - \gamma g$
    **end for**
**end for**
```

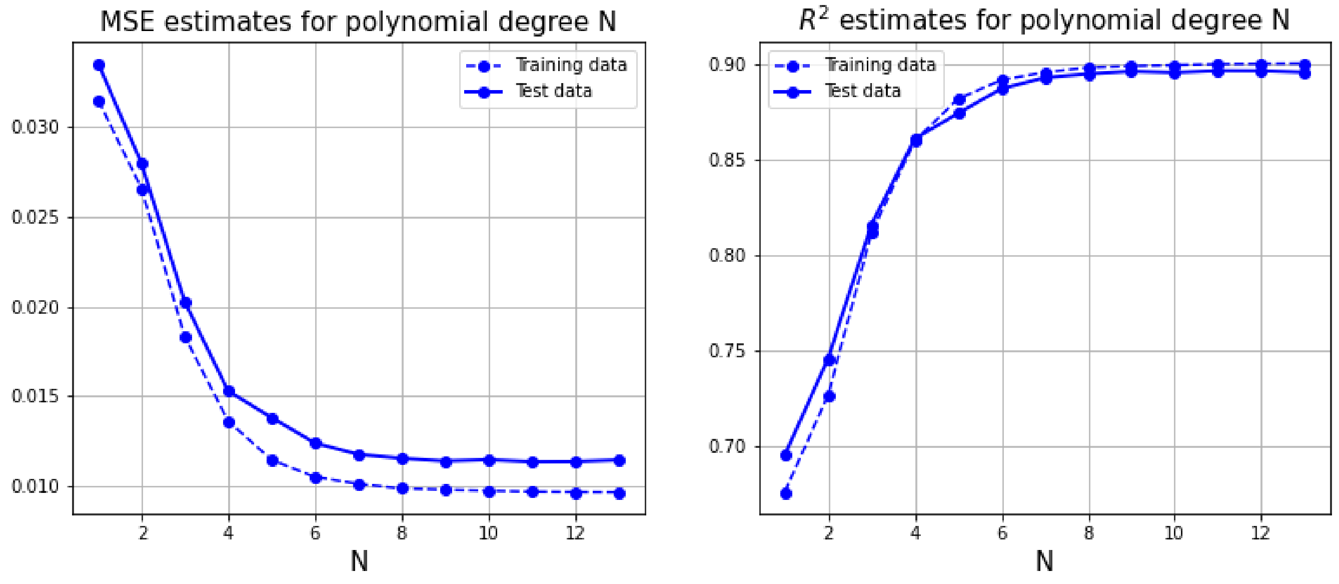

Figure 11. From Project 1: MSE and R^2 from OLS regression of a Franke function dataset defined like in Equation 21 for a larger range of polynomial degrees. The MSE for the fit to test data settles at around 0.012.

[2] D. W. H. Wolberg, "Breast cancer wisconsin (diagnostic) data set," https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic), Accessed 14.11.22. (1995).
[3] M. Hjorth-Jensen, "Applied data analysis and machine learning," https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html, accessed 03.11.22.
[4] "Gradient descent," https://en.wikipedia.org/wiki/Gradient_descent#/media/File:Gradient_descent.svg Accessed 17.11.22.
[5] What is logostic regression?, IBM https://www.ibm.com/topics/logistic-regression. Accessed 15.11.22.
[6] H. Hassan, A. Negm, M. Zahran, and O. Saavedra, International Water Technology Journal **5** (2015).
[7] Milind Sahay (2020), *Neural Networks and the Universal Approximation Theorem*. https://towardsdatascience.com/neural-networks-and-the-universal-approximation-theorem-8a389a33d30a. Accessed 15.11.22.
[8] Anas Al-Masri (2019), *How Does Back-Propagation in Artificial Neural Networks Work?*. https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7 Accessed 16.11.22.
[9] "Introduction to loss functions," https://www.datarobot.com/blog/introduction-to-loss-functions/ Accessed 18.11.22.

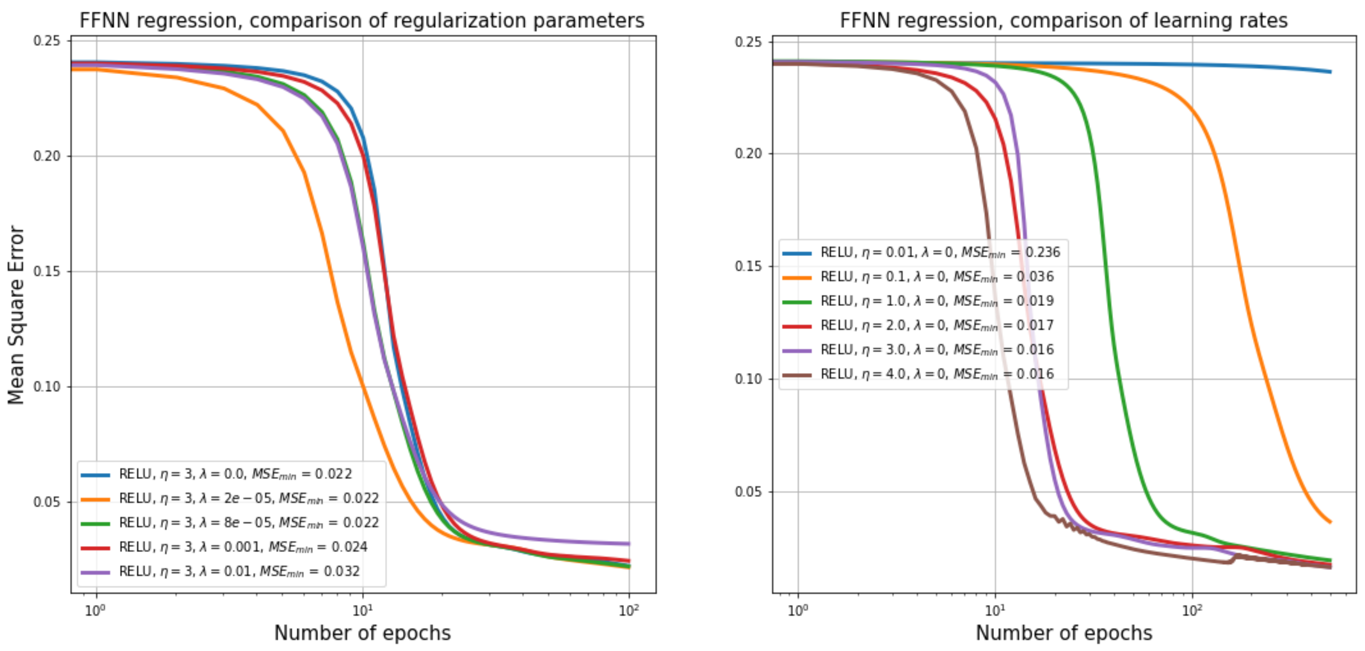

Figure 12. Plots comparing MSE from FFNN regression networks after 100 epochs with varying regularization parameter λ and learning rate η (for 500 epochs) respectively, all with RELU activation functions. All networks consist of one hidden layer containing 20 nodes.

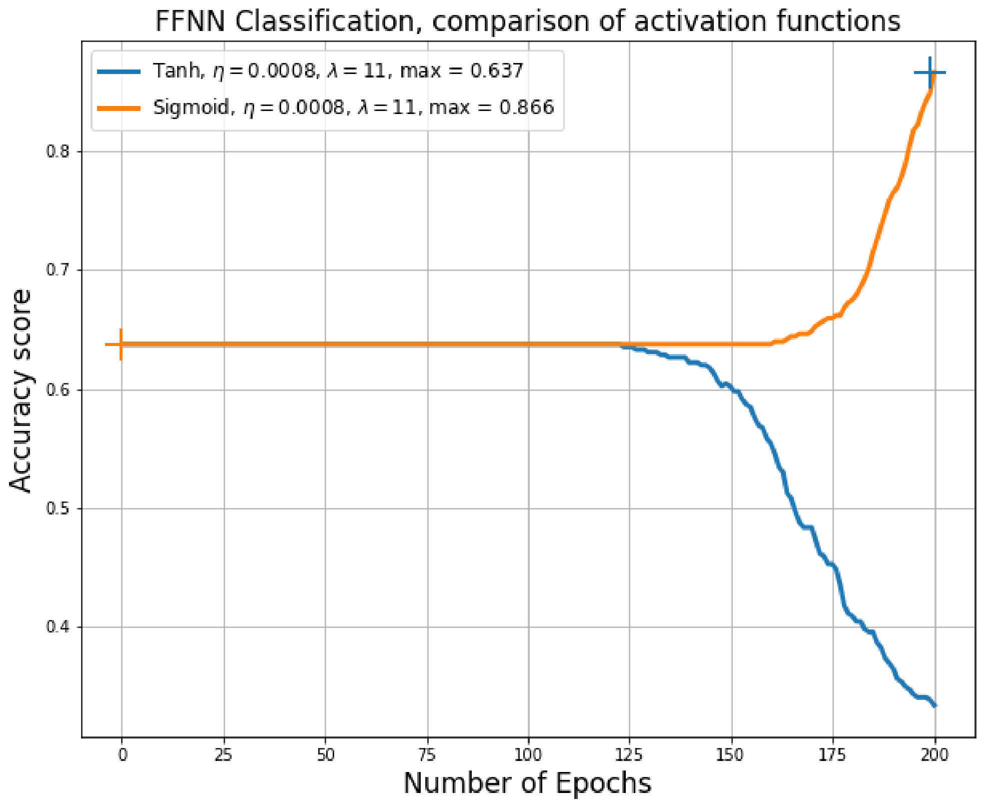

Figure 13. Development of accuracy for classification networks approximating to the Wisconsin breast cancer dataset using four features. This run used the exact same setup as the one resulting in Figure 8 except that the Tanh network performed its training first.