FYS-STK3155 Project 2

Lasse Pladsen, Parham Qanbari, & Sander V. Vattøy

November 16, 2023

Abstract

In this project we explored neural networks and their applications. First, we explored various optimization methods for linear regression by using the gradient descent approach. Here we obtained an mean squared error (MSE) of 1.36. Writing our own neural network for regression we obtain $MSE = 21.7$ and coefficient of determination (R^2) of 0.98 for stochastic minibatch gradient descent, using the optimizer method ADAM with the activation function Sigmoid. Replacing Sigmoid activation with ReLU and leaky ReLU resulted in better predictions of $MSE = 1.5$ and $R^2 = 1$. We also train our network for a classification problem on the 'Wisconsin Breast Cancer' dataset. Our best prediction accuracy was 0.98 which was obtained with leaky ReLU using 2 layers and 20 nodes as the network architecture. In general for classification, the leaky ReLU performed best across various configurations with an average score of 0.95, while ReLU performed on average 0.93 and Sigmoid at 0.77. Although, leaky ReLU was 2% faster than ReLU it was also five times slower in terms of computation time. With this in consideration our best model for neural network classification analysis was ReLU. Comparing this with plain logistic regression method actually provided better predictions than our network with the accuracy = 0.96.

I. INTRODUCTION

Neural networks have become a big deal in the world of machine learning and data predictions. They are a computer's way of mimicking how our brains work. In this project, we're digging into feed-forward neural networks, a popular type, and specifically, how we can make them work better using the gradient descent method and back-propagation algorithm.

Gradient descent, also called steepest descent, is a popular method for tweaking the inner parameters of neural networks. Back-propagation is a way to fine-tune weights in the network by minimizing errors between what the model predicts and what actually happens. Using these methods we allow the network to essentially learn to fit itself onto a data set.

Our main goal is to see how changing input parameters like learning rates, network structure, activation functions, affects how well feed-forward neural networks perform. By playing around with these, we hope to conclude the best methods and parameters for certain different problems.

This report is split into six major sections. The first section (A. Gradient descent regression) focuses on basic regression and comparing methods for gradient descent.

In the second section (B. Neural network regression) we write our neural network and perform linear regression.

The third section (C. Comparing activation functions) we focus on tweaking some of the parameters of our network's regression from the previous section, mostly on whats called the activation function, but this we will come back to in the theory part of the report.

The fourth section (D. Neural network classification) focuses on using our network for binary classification purposes like predicting breast cancer. Binary in the sense that we have two output categories, zero being a benign cancer and one being a malignant cancer.

In the fifth section (E. Logistic regression) we compare the performance from the fourth section (classification) to plain logistic regression.

The final section (F. Evaluation of the various algorithms) is about summarizing all techniques used in the report and performing an evaluation on them.

II. THEORY

A. Design matrix

B. Design matrix for linear regression

We create a so-called design matrix $\mathbf{X}$ for the regression methods from the input data vector $\mathbf{x} \in \mathbb{R}^{\mathbf{n}}$. Each row in $\mathbf{X}$ represents polynomial variables from one data sample. We choose a max polynomial degree p with n data samples such that $\mathbf{X} \in \mathbb{R}^{n\times(p+1)}$. The design matrix is given as the following (Hjorth-Jensen, 2023)

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^p \\ 1 & x_2 & x_2^2 & \dots & x_2^p \\ \ddots & \ddots & \ddots & \ddots & \ddots \\ 1 & x_n & x_n^2 & \dots & x_n^p \end{bmatrix} \tag{1}$$

C. Measuring prediction performance

To calculate the performance of our predictions we use different methods depending on what type of problem we are dealing with.

For classification problems we use the accuracy score defined as

$$\begin{aligned} Accuracy &= \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \\ &= \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \end{aligned} \tag{2}$$

where I is the indicator function that simply checks if they are equal:

$$I(t = y) = \begin{cases} 1, & t = y \\ 0, & t \neq y \end{cases} \tag{3}$$

For regression problems we use a so-called cost function and the coefficient of determination. A cost function is a function that defines how we measure error and it is what one wishes to minimize, because it directly relates to a more precise model.

The coefficient of determination, denoted R^2, is a measure that tells us how well our models can predict new data and is defined as

$$R^2(\mathbf{z}, \tilde{\mathbf{z}}_i) = 1 - \frac{\sum_{i=0}^{n-1}(z_i - \tilde{z}_i)^2}{\sum_{i=0}^{n-1}(z_i - \bar{z})^2} \tag{4}$$

C1. Ordinary least squares

Ordinary least squares (OLS) is a linear regression method that uses the mean squared error/difference (MSE) between a true value and a prediction as the cost function C. The MSE cost function is expressed as

$$MSE = C(\boldsymbol{\theta}) = \frac{1}{n}\sum_i \left(t_i - y_i(\boldsymbol{\theta})\right)^2 \tag{5}$$

where n is the number of data points, $\boldsymbol{t}$ is the target vector and $\mathbf{y}(\boldsymbol{\theta}) = \boldsymbol{X\theta}$ is the prediction vector. Here $\boldsymbol{X}$ is the design matrix and the parameter $\boldsymbol{\theta} \in \mathbb{R}^{p+1}$ represents the vector of coefficients and the intercept term that the model learns through training, defining the relationship between the input features and the predicted output.

C2. Ridge

Ridge is another linear regression method that adds a shrinkage term called an L2 regularization to the MSE from OLS' cost function:

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_i \left(t_i - y_i(\boldsymbol{\theta})\right)^2 + \lambda \sum_i \theta_i^2 \tag{6}$$

where λ is the regularization/shrinkage parameter.

C3. Cross entropy/logistic regression

Cross entropy is a logistic regression method also used for classification problems. This method uses the Sigmoid function $\sigma(x)$ (23) to make a prediction by classifying some input as a probability (between 0 and 1). The cross entropy for binary classification is defined as

$$C(\boldsymbol{\theta}) = -\sum_i \left[t_i \log(y_i) + (1 - t_i)\log(1 - y_i)\right] \tag{7}$$

D. Gradient descent

Gradient descent is a way to minimize a function. By using the gradient one can move towards and potentially locate a minima of the function. All these coming expressions are also described in Goodfellow et al. (2016).

For a cost function $C(\boldsymbol{\theta})$ with $\boldsymbol{\theta} = \begin{pmatrix} \theta_1 & \theta_2 & \dots & \theta_n \end{pmatrix}$ we are interested in finding the optimal $\boldsymbol{\theta}$:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, C(\boldsymbol{\theta}) \tag{8}$$

The cost function's gradient is defined as

$$\boldsymbol{\nabla} C(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial C(\boldsymbol{\theta})}{\partial \theta_1} & \frac{\partial C(\boldsymbol{\theta})}{\partial \theta_2} & \dots & \frac{\partial C(\boldsymbol{\theta})}{\partial \theta_n} \end{pmatrix} \tag{9}$$

To minimize the cost function the gradient descent technique begins an initial guess $\boldsymbol{\theta}_0$, then in steps we reduce this guess using the gradient $\boldsymbol{\nabla} C(\boldsymbol{\theta})$ by the following algorithm for a general iteration step i:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta_i \boldsymbol{\nabla} C(\boldsymbol{\theta_i}) \tag{10}$$

where the term η_i is called the *learning rate* which can either be some chosen constant or scaled over time (iterations) using a chosen scaling method. If the learning rate is sufficiently small the new guess will always be closer to the optimal $\hat{\boldsymbol{\theta}}$ such that $C(\boldsymbol{\theta}_{i+1}) \leq C(\boldsymbol{\theta}_i)$. With enough iterations we should approach the cost functions minima. The minima is not guaranteed to be a global minima as this method could potentially get stuck in a local minima.

D1. Gradient descent with momentum

This extended method of gradient decent uses a momentum/memory term from the previous iteration throughout the computation. This helps the convergence of $\boldsymbol{\theta}$ by remembering the direction that the previous iterations moved in. The algorithm can be described as

$$v_i = \gamma v_{i-1} + \eta_i \boldsymbol{\nabla} C(\boldsymbol{\theta_i}) \tag{11}$$

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - v_i \tag{12}$$

where $0 \leq \gamma \leq 1$ is a chosen momentum rate.

D2. Stochastic minibatch gradient descent

This method of gradient descent samples a batch with size M of the total dataset and instead calculates the gradient of the sampled batch. If the dataset has N datapoints then the total number of minibatches will be $m = N/M$. For an iteration over a sampled batch we use the phrase iteration, and the total iteration over all minibatches is called an epoch. This method has the potential to speed up the computational time, the convergence time of $\boldsymbol{\theta}$, and potentially provide better results by introducing randomness/stochasticity which lowers the chance of getting stuck at a local minima. In this report

we will refer to this minibatch sampling as Stochastic gradient descent (SGD), and can be described as

In the same way as described for simple gradient decent, it is also possible to add a momentum term to SGD (SGDM), and will make the $\boldsymbol{\theta}$ convergence better.

E. Methods for scaling the learning rate

The choice of learning rate is very important to the results and because of this there are many different ways to adapt the learning rate over time/iterations to improve the results.

E1. Adagrad

The Adagrad algorithm scales the learning rate η_i with a moment term $\mathbf{r}$ with the following expressions

$$\text{initialize } \mathbf{r}_0 = 0$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \odot \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \tag{13}$$

$$\eta_{i+1} = \frac{\eta_i}{\delta + \sqrt{\mathbf{r}_{i+1}}} \odot \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \tag{14}$$

where $\odot$ is the *Hadamard product* which means element-wise multiplication, and δ is a small constant to avoid dividing by zero.

E2. RMSProp

The RMSProp method is similar to Adagrad, but it adds a decay rate ρ to scale the learning rate as such:

$$\text{initialize } \mathbf{r}_0 = 0$$

$$\mathbf{r}_{i+1} = \rho \mathbf{r}_i + (1-\rho) \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \odot \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \tag{15}$$

$$\eta_{i+1} = \frac{\eta_i}{\delta + \sqrt{\mathbf{r}_{i+1}}} \odot \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \tag{16}$$

E3. ADAM

The ADAM method is very similar to RMSProp but it uses an addition moment term $\mathbf{r}, s$ and decay rate ρ_1, ρ_2 and it also uses momentum. It is generally one of the more reliable methods for scaling and can be describes by the following:

$$\text{initialize } \mathbf{s}_0 = 0$$

$$\text{initialize } \mathbf{r}_0 = 0$$

$$\mathbf{s}_{i+1} = \rho_1 \mathbf{s}_i + (1-\rho_1) \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \tag{17}$$

$$\mathbf{r}_{i+1} = \rho_2 \mathbf{r}_i + (1-\rho_2) \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \odot \boldsymbol{\nabla} C(\boldsymbol{\theta}_i) \tag{18}$$

$$\hat{\mathbf{s}}_{i+1} = \frac{\mathbf{s}_{i+1}}{1-\rho_1^i} \tag{19}$$

$$\hat{\mathbf{r}}_{i+1} = \frac{\mathbf{r}_{i+1}}{1-\rho_2^i} \tag{20}$$

$$\eta_{i+1} = \eta_i \frac{\hat{\mathbf{s}}_{i+1}}{\delta + \sqrt{\hat{\mathbf{r}}_{i+1}}} \tag{21}$$

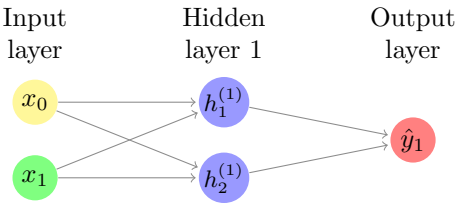

Figure 1: Neural network schematic.

F. Feed-Forward Neural Network

We design our Feed-Forward Neural Network (FFNN) algorithm using an input layer, at least one hidden layer and an output layer. In order to train the model we use backpropagation to calculate the parameters giving the most accurate prediction.

The basic design of out NN is demonstrated by the schematic (1). The input layer takes in $\mathbf{X}$ design matrix which is then multiplied by weights $\mathbf{W}_l$ in the general hidden layer l and the bias $\mathbf{b}_l$ in the hidden layer. This result in in the function $\mathbf{z}_l$ (22):

$$\mathbf{z}_l = \mathbf{X}\mathbf{W}_l + \mathbf{b}_l \tag{22}$$

Before moving to the next layer in the network, $\mathbf{z}_l$ needs to be fed into an activation function. It will consider whether nodes should be activated, with the resulting value $a = f(\mathbf{z}_l)$, where it is possible to test a NN for different activation functions.

For the output layer $l = L$, depending on the problem at hand, $\mathbf{z}_L$ could also be fed into an activation layer. For binary classifications problems it's useful to feed $\mathbf{z}_L$ into the an activation function (see F1). However, for regression problems we wish to use the actual calculated value for a prediction, not its activation value. Thus, for regression problems $\mathbf{z}_L$ is not fed into the Sigmoid function.

F1. Activation functions

An activation function is a function we use in a neural network under calculation of the weighted sum of output-weights and biases from the nodes in our layers, to be able to do back-propagation through the feed forward process. Inspired from biological networks (Hjorth-Jensen, 2023) the resulting value from the activation $a = f(\mathbf{z}_l)$ represents the activation level of a neuron, or in our case a node. It is a way to quantify the activity in the network in a normalized way. In this project we will consider three activation functions; Sigmoid-, ReLU- and Leaky ReLU function.

The Sigmoid function is defined as

$$f(x) = \frac{1}{1+e^x}, \tag{23}$$

and will take the inputs $\mathbf{z}_l$ in the feed forward process and give a value between 0 and 1:

$$a_l = f(\mathbf{z}_l) \to [0, 1] \tag{24}$$

This normalization between 0 and 1 makes the Sigmoid activation function useful for classification problems (Dorsaf, 2021). However, it has some limitations, one of which are specially relevant for this project. When a neuron activation becomes 0 or 1 in the training it becomes saturated and its derivative becomes zero, thus the update of weight and biases stop (Dorsaf, 2021), or as its said in the field, they die. This could disturb the training process of the neuron and essentially hinder learning and finding the optimal weights and biases (Dorsaf, 2021). However, the Rectified linear unit function does not have a saturation problem.

The ReLU (Rectified linear unit) function is defined as

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{25}$$

Its values range from 0 to infinity,

$$a_l = f(\mathbf{z}_l) \to [0, \infty] \tag{26}$$

The ReLU function also has a limitation that is relevant for our project, which is a dying problem for negative values (Dorsaf, 2021). Since values that are negative are set to zero, it could cause a saturation problem if too many values are negative. With many values being set to zero in the activation function the learning for the neuron essentially stops. A third activation functions exists that could counteract this negative value saturation problem, namely the leaky RELY function.

The Leaky ReLU fuction is defined as,

$$f(x) = \begin{cases} x, & x > 0 \\ x \cdot \delta, & x \leq 0 \end{cases} \tag{27}$$

and ranges from negative infinity to positive infinity,

$$a_l = f(\mathbf{z}_l) \to [-\infty, +\infty] \tag{28}$$

where δ is a small constant - we use the constant 10^{-2}. The constant δ aims to multiply the negative values by a small values so that they are not set to zero, thus hindering saturation for negative values (Dorsaf, 2021).

In the calculation of the hidden layer errors we also need the derivatives of the activation functions - which are the following equations (29)(30)(31):

The derivative of the Sigmoid function can be calculated to

$$f'(x) = f(x)(1 - f(x)) \tag{29}$$

The derivative of the ReLU:

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{30}$$

and the derivative of Leaky ReLU as

$$f'(x) = \begin{cases} 1, & x > 0 \\ \delta, & x \leq 0 \end{cases} \tag{31}$$

F2. Backpropagation

The parameters we want to optimize in the neural network is the weights $\mathbf{W}$ and biases $\mathbf{b}$ in the hidden layers and the output layer. To start the network we use random parameters, then we use a backpropagation algorithm to optimize the parameters. This means we start at the output layer L and propagate backwards through the hidden layers l while updating each of the parameters by using the proceeding layer $l + 1$.

For optimization of the parameters we update the weights and biases with a learning rate multiplied with the proceeding layer's error. The error of the output layer L can be calculated by differentiating the cost function with respect to the output z_L (32) and for the general hidden layer l we use the proceeding layer's error and weights as mentioned (33) (Hjorth-Jensen, 2023).

$$\begin{aligned} \boldsymbol{\delta}^L = \frac{\partial C}{\partial \mathbf{z}} &= \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \frac{\partial \mathbf{C}}{\partial \mathbf{a}} \\ &= f'(\mathbf{z^L}) \frac{\partial \mathbf{C}}{\partial \mathbf{a}} \end{aligned} \tag{32}$$

$$\begin{aligned} \boldsymbol{\delta}^l &= \boldsymbol{\delta}^{l+1} (\mathbf{W^{l+1}})^T \frac{\partial \mathbf{a^l}}{\partial \mathbf{z^l}} \\ &= \boldsymbol{\delta}^{l+1} (\mathbf{W^{l+1}})^T f'(\mathbf{z^l}) \end{aligned} \tag{33}$$

In order to find the optimal weights and biases we use the gradient descent method (see section D) to update them iteratively.

The expressions for the gradients for the gradient descent method are as following (Hjorth-Jensen, 2023)

$$\nabla \mathbf{W}^l = (\mathbf{a}^{l-1})^T \boldsymbol{\delta}^l \tag{34}$$

And for the first layer we will just have the design matrix as $\mathbf{a}$:

$$\nabla \mathbf{W}^{l=1} = (\mathbf{X})^T \boldsymbol{\delta}^{l=1} \tag{35}$$

The gradient for updating the bias for a the general layer l is

$$\nabla \mathbf{b}^l = \sum_{i=1}^{n_{inputs}} \boldsymbol{\delta}_i^l \tag{36}$$

We perform this algorithm iteratively until we have optimal parameters based on what we will consider a acceptable convergence threshold.

III. METHODS

A. Gradient descent regression

We begin with an analysis of gradient descent methods. In this analysis we will use both analytical gradients and automatic gradients (using the Python library `Autograd`) for four different gradient descent methods; gradient decent (GD), as well as with momentum (GDM), and stochastic gradient decent (SGD), as well as with momentum (SGDM). We wish to compare these and find out which method gives the best results.

We do this analysis for both OLS and Ridge regression for the chosen second order polynomial

$$f(x) = 4x^2 + 3x + 2 + \epsilon, \tag{37}$$

where ϵ represent the noise we simulate in our model. We choose to fit this function within the interval $x \in [-5, 5]$ which is shown in figure 2.

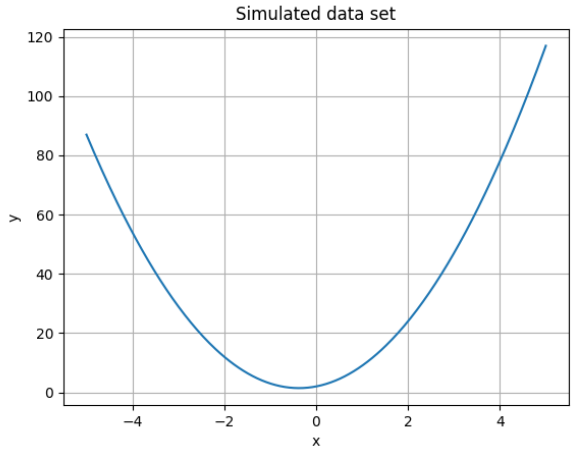

Figure 2: Plot of the polynomial we use to simulate our data set (here without noise).

In the whole project we will use a 80-20 training-testing split for all the data we use. For this we use the Python library `Scikit-learn`'s implementation `train_test_split`.

We also try out the three methods for scaling learning rate; Adagrad, RMSProp and ADAM. Different parameters will need to be tuned to give the best optimal results, for example the learning rate rate η and the shrinkage term λ for Ridge regression. Using the Python library `Seaborn` we can do what is called a grid search where we plot the performance as function of multiple parameters.

B. Neural network regression

We will now write our Neural Network implementation using the feed-forward and back propagation algorithms in Python. It should be flexible with regards to the network structure, activation and cost functions, and other parameters. We choose to focus on using SGD algorithm with the ADAM learning method because these should in general give the best results.

Afterwards we will use our network's regression capabilities on our second order polynomial (37). Here we use the OLS cost function/MSE (5), and the sigmoid function (23) as the hidden layers' activation function. For the output we wish to have continuous predictions instead of probabilities so we do not use an activation function for the output layer. We then use our FFNN and train our data, and thereafter test our model against `Scikit-learn`. Before proceeding we make an analysis of the regularization parameters and the learning rate used to find the optimal MSE and R^2 scores in the calculations.

C. Comparing activation functions

We test and compare three acitvation functions, the Sigmoid (23), ReLU (25) and leaky ReLU (27). For the classification problem we only change the activation functions in the hidden layers, and keep the Sigmoid function in output layer.

D. Neural network classification

Next, we will perform a classification analysis using our neural network on the 'Wisconsin Breast Cancer' data set (Wolberg, 1992). The data set contains 30 features, each with $n = 569$ data points. The features represent various characteristics of a cancer sample and our goal is to predict whether the cancer sample based on the data is 'malignant' or 'benign'. The targets are binary classified where 1 is a malignant cancer sample, while 0 is benign.

We try to fit a network model with one hidden layer and compare it with using two hidden layers. For these two version we vary the number of nodes in each hidden layer, and we also compare the three different activation functions - namely, Sigmoid (23), ReLU (25) and leaky ReLU (27). Furthermore, we implement SGD and Adam optimizer to the model and experiment with the various configurations. We store the prediction accuracies of the various configurations and discuss the observations.

We experiment to find the parameter combinations with the best accuracy score. We also wish to compare our network's performance to the performance of the library `Scikit-learn`. This library implements classification in the `MLPClassifier` class.

E. Logistic regression

For this section we compare the network's classification performance with a pure logistic regression (7) using the SGD algorithm with ADAM, and the same data set (Wisconsin breast cancer) Here we also study the results as function of the learning rate η and regularization parameter λ. We also compare the results with `Scikit-learn`'s implementation `LogisticRegression`.

F. Evaluation of the various algorithms

Finally we wish to do a summary and evaluation of the different methods and algorithms, comparing them to each other and concluding the best methods for each problem.

IV. RESULTS

A. Gradient descent regression

We first did a grid search with SGD and ADAM to find the optimal parameters for our gradient descent comparisons using Ridge regression. The result is shown in figure 3 and we found the optimal parameters to be

Learning rate	$\eta = 0.3$
Regularization rate	$\lambda = 0.001$
Number of epochs	$n_{\text{epoch}} = 50$
Number of minibatches	$m = 20$

Using these optimal parameters we do the regression with the different gradient descent methods, we plot the MSE drop-off with regards to number of iterations calculating the gradients with analytical expressions in figure 4 and with automatic differentiation (`Autograd`) in appendix figure 15. These shown results are discussed in more detail in the discussion part of the report, but in general we can see in figure 3 that we obtain $MSE = 1.36$.

B. Neural network regression

After writing our neural network we again need to find the optimal parameters to minimize our cost function fitting the network to our chosen polynomial (37). Using the Sigmoid function as hidden activation we plot the η and λ grid searches in figure 5. We found that the optimal amount of batches was around 15-20 by creating different grid plot, and we ran all training with $n_{epochs} = 1000$ number of epochs. With this we found the optimal combination to be

Learning rate	$\eta = 0.01$
Regularization rate	$\lambda = 0.1$
Number of minibatches	$m = 15$

Using these optimal parameters our prediction fit is plotted in 6, while prediction using the network from

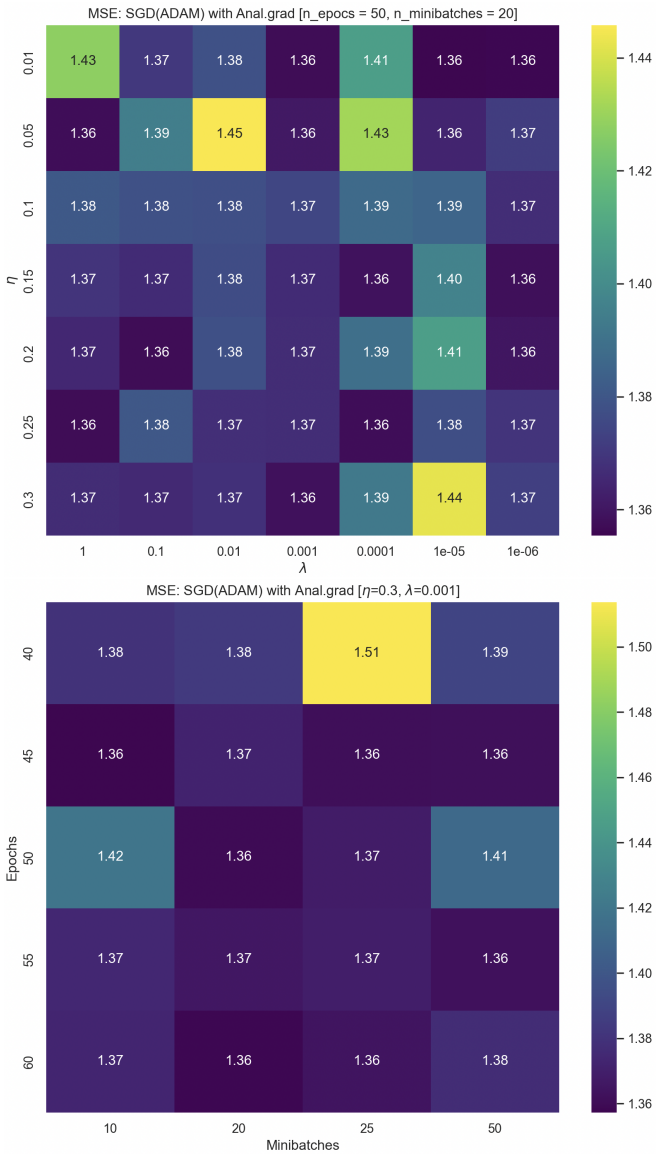

Figure 3: Grid search plot of different parameters using the SGD algorithm with the ADAM scaling method. By first finding optimal η and λ, we thereafter found optimal amount of epochs and minibatches.

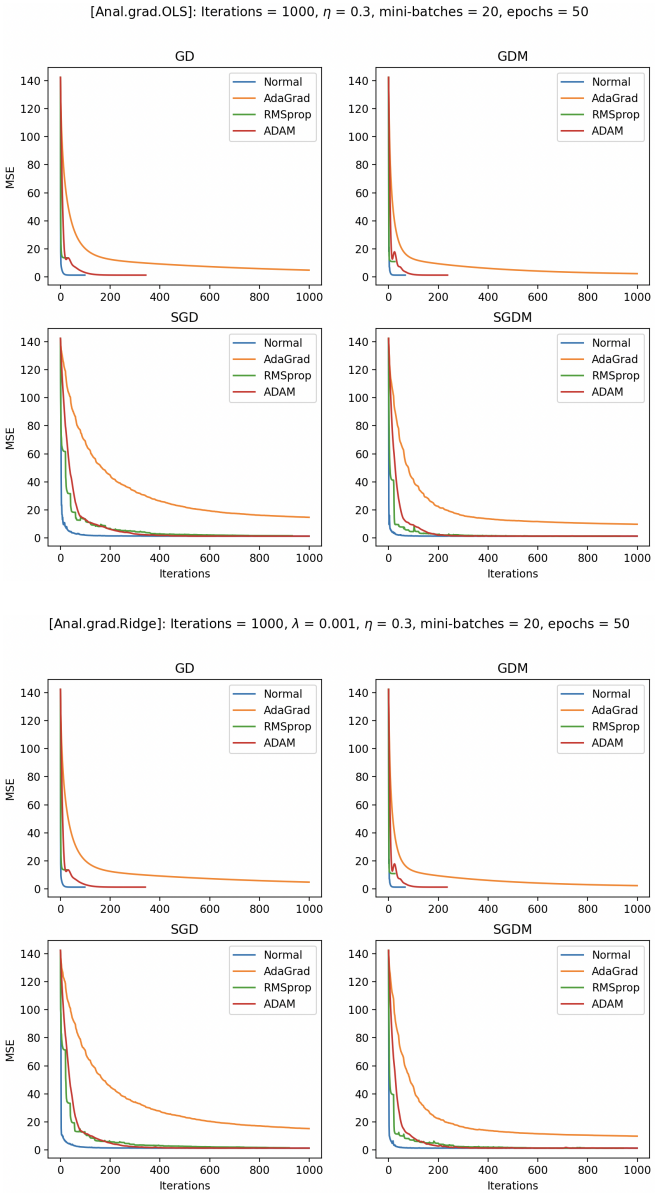

Figure 4: Plots of gradient descent and scaling methods using analytical gradient with optimal hyper parameter λ, learning rate η, and number of epochs and mini-batches. The four first plots are with OLS and the last four are with Ridge.

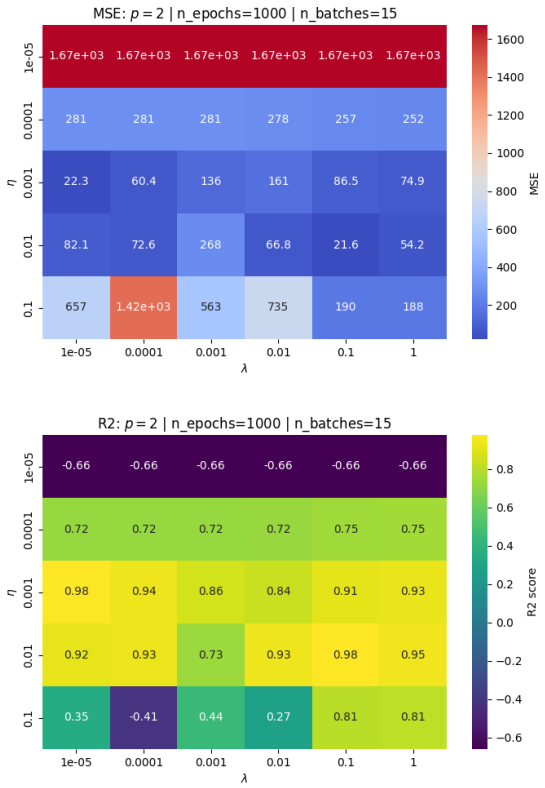

Figure 5: MSE (top) and R^2 (bottom) parameter grid plot for SGD with ADAM using 1000 epochs and 15 batches, with the Sigmoid function as hidden activation. From this we can find the optimal parameters relating to the minimum MSE.

`Scikit-learn` is shown in figure 7. Our own fit is very good within our chosen x-interval with

$$MSE = 21.7$$
$$R^2 = 0.98$$

while the `Scikit-learn` network provided

$$MSE = 122.6$$
$$R^2 = 0.88$$

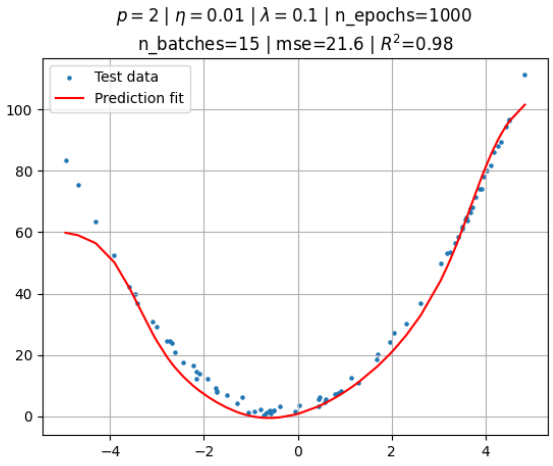

Figure 6: Test data prediction of our polynomial using our own neural network with optimal parameters.

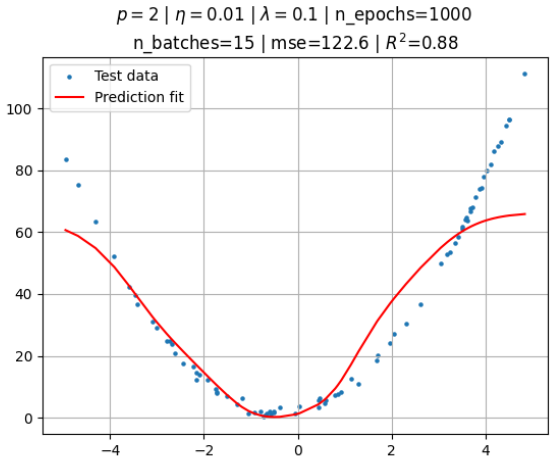

Figure 7: Test data prediction of our polynomial using `Scikit-learn` neural network. For these particular parameters the prediction is not that good compared to our network.

C. Comparing activation functions

Replacing the hidden activation with the ReLU and leaky ReLU activation functions we find new optimal parameters looking at the MSE and R^2 scores. As mentioned above, we again focus on 15 batches for SGD and ADAM. The results for the ReLU function and leaky ReLU is respectively shown in figure 8 and 9. We found that they both have the same optimal parameters which are

Learning rate	$\eta = 0.0001$
Regularization rate	$\lambda = 0.01$
Number of epochs	$n_{\text{epoch}} = 1000$
Number of minibatches	$m = 15$

and they had almost the same performance

$$MSE \simeq 1.5$$
$$R^2 = 1$$

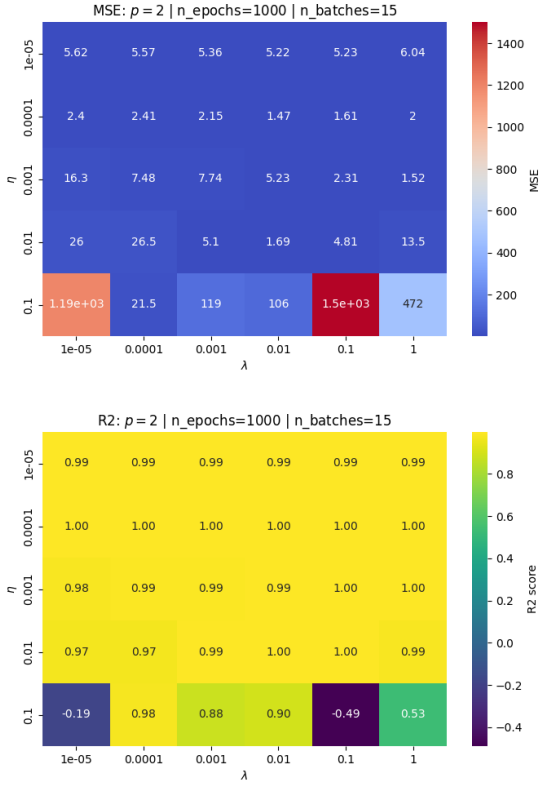

Figure 8: MSE parameter grid plot for SGD with ADAM using 1000 epochs and 15 batches, with the ReLU function as the hidden activation. From this we can find the optimal parameters relating to the minimum MSE.

D. Neural network classification

We tested the neural network using no optimizer, implementing SGD and Adam, and at last compared with scikit `MLPClassifier`. We experimented the model with plain gradient descent using tree different activation functions, Sigmoid (23), ReLU (25) and leaky ReLU (27). Furthermore, we varied the number of nodes in the network where we tested with 5, 10 and 20 nodes. In the

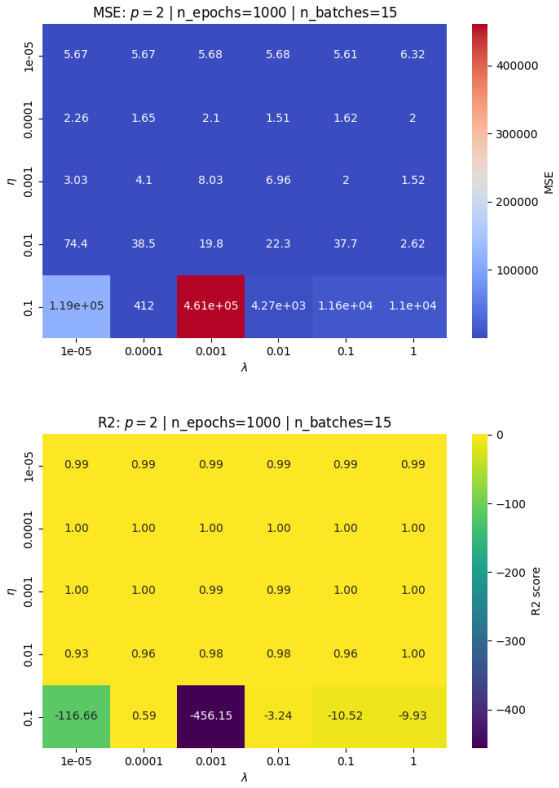

Figure 9: MSE parameter grid plot for SGD with ADAM using 1000 epochs and 15 batches, with the leaky ReLU function as the hidden activation. From this we can find the optimal parameters relating to the minimum MSE.

model with two hidden layers, both layers had identical number of nodes. We tested the following learning rates η, hyperparameters λ, number of epochs and number of minibatches:

Learning rates $\eta = [10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1]$
Regularization rates $\lambda = [10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1]$
Number of epochs $n_{\text{epoch}} = 1000$
Number of minibatches $m = 5$

The prediction accuracies obtained from the plain model are summarized in table (1), the optimized model is summarized in table (2), and the results from Scikit `MLPClassifier` is summarized in table (3).

The raw grid plots showing accuracy as functions of the learning rate η and the regularization λ are shown in figure (10) for Sigmoid, figure (11) for ReLU, and figure (12) for leaky ReLU.

Results using plain gradient descent

Model Configuration		Sigmoid	ReLU	lRELU
1 Hidden Layer	5 Nodes	0.91	0.91	0.91
	10 Nodes	0.87	0.92	0.90
	20 Nodes	0.88	0.93	0.90
2 Hidden Layers	5 Nodes	0.80	0.71	0.87
	10 Nodes	0.89	0.85	0.92
	20 Nodes	0.85	0.81	0.88

Table 1: The table values show the prediction accuracies we found for our different model configurations using plain gradient descent. The second layer configuration has equal amounts of nodes in each layer.

Results using optimization

Model Configuration		Sigmoid	ReLU	lReLU
1 Hidden Layer	5 Nodes	0.86	0.89	0.92
	10 Nodes	0.63	0.89	0.94
	20 Nodes	0.61	0.92	0.95
2 Hidden Layers	5 Nodes	0.84	0.97	0.98
	10 Nodes	0.74	0.96	0.98
	20 Nodes	0.94	0.97	0.98

Table 2: The table values show the prediction accuracies we found for our different model configurations using optimizers Adam and SGD. The second layer configuration has equal amounts of nodes in each layer.

The average prediction accuracies for each activation function in the optimized models is:

Sigmoid: 0.77
ReLU: 0.93
leaky ReLU: 0.95

Results using Scikit-learn

Model Configuration		Sigmoid	ReLU	lReLU
1 Hidden Layer	5 Nodes	0.95	0.95	N/A
	10 Nodes	0.94	0.94	N/A
	20 Nodes	0.95	0.95	N/A
2 Hidden Layers	5 Nodes	0.97	0.96	N/A
	10 Nodes	0.96	0.96	N/A
	20 Nodes	0.97	0.94	N/A

Table 3: The table values show the prediction accuracies we found for different model configurations. These results are from the Scikit `MLPClassifier` using SDG and Adam optimizer. The function does not have leaky ReLU as activation, hence N/A marking.

For Sigmoid a learning rate of 10^{-1} and regularization term of 10^{-5} showed the best accuracy 0.94 for model configuration of 2 layers and 20 nodes.

Grid plot using Sigmoid activation

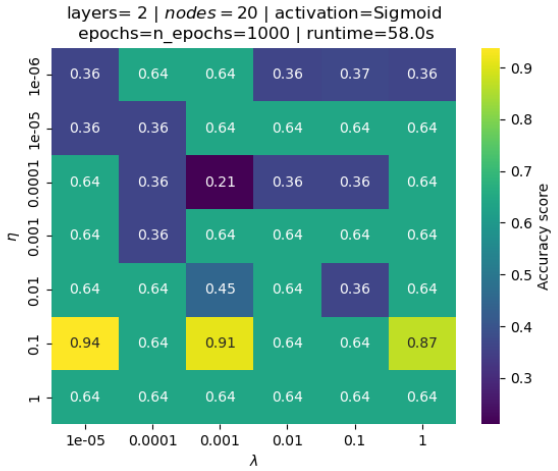

Figure 10: Grid plot of prediction accuracies using **Sigmoid** activation function in the optimized model as a function of learning rates η and regularization term λ. The model configuration was 2 layer with 20 nodes.

For ReLU a learning rate of 10^{-1} and regularization term of 10^{-3} showed the best accuracy 0.97 for model configuration of 2 layer and 20 nodes. Identical accuracy was obtained with 2 layers and 5 nodes.

For leaky ReLU a learning rate of 1 and regularization term of 10^{-5} to 10^{-3} all resulted in accuracy of 0.98. The same accuracy was obtained for model configuration of 2 layer and 5, 10, and 20 nodes.

E. Logistic regression

We now instead use plain logistic regression on the Wisconsin breast cancer data. The parameter grid search is shown in figure 13, we found multiple equally good

Grid plot using ReLU activation

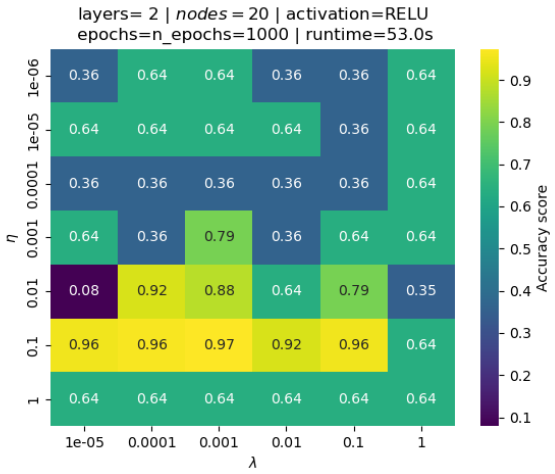

Figure 11: Grid plot of prediction accuracies using **ReLU** activation function in the optimized model as a function of learning rates η and regularization term λ. The model configuration was 2 layers with 20 nodes.

Grid plot using leaky ReLU activation

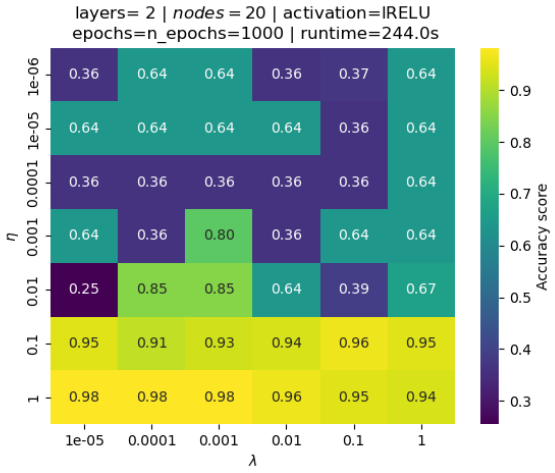

Figure 12: Grid plot of prediction accuracies using **leaky ReLU** activation function in the optimized model as a function of learning rates η and regularization term λ. The model configuration was 2 layer with 20 nodes.

parameter combinations, one such combination is

$$\begin{aligned} \text{Learning rate} &\quad \eta = 0.001 \\ \text{Regularization rate} &\quad \lambda = 0.01 \\ \text{Number of minibatches} &\quad m = 20 \end{aligned}$$

which resulted in

$$\text{Accuracy} = 0.96$$

Using `Scikit-learn`'s logistic regression implementation had accuracies independent on the same set of parameters, the grid plot is shown in figure 14 with

$$\text{Accuracy} = 0.96$$

Logistic regression with own code

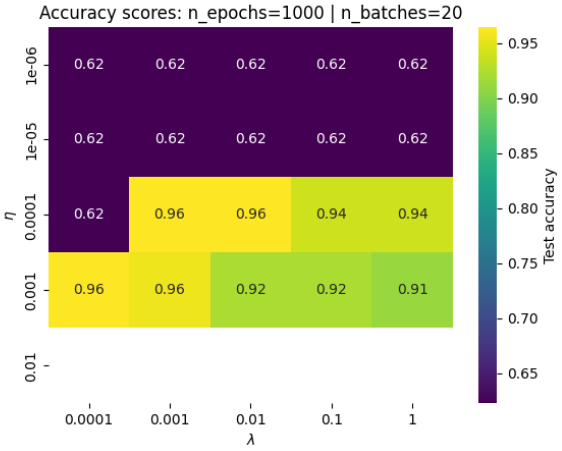

Figure 13: Accuracy scores of model prediction to breast cancer data using logistic regression. Note that for $\eta = 0.01$ our fit reached overflow leading to no usable prediction. There are four equally good parameter combinations.

V. DISCUSSION

A. Gradient descent regression

Comparing the different gradient descent methods we see that the plain GD seems to have a faster MSE drop-off, but ADAM is generally the second best, and end up with a very small MSE. In general we expected ADAM to perform the best in most (if not all) cases, as this is the scaling method most widely used. We are not exactly sure if our approach is to fault in this behaviour, but its possible that there is something in our implementation that might be wrong. In any case, we chose to mostly focus on using SGD with ADAM in the rest of the project – and we actually saw that this pretty much always gave better, and more consistently better, results than all other methods.

Logistic regression with Scikit-learn

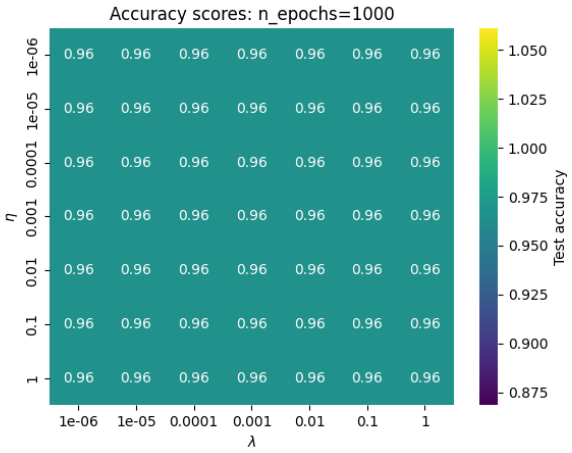

Figure 14: Accuracy score of model prediction to breast cancer data using `Scikit-learn` logistic regression. Here every plotted parameter combination is equally good.

Focusing on the ADAM results of this part, the MSE only converges for GD and GDM, but for all instances we generally end up with a very small MSE ($MSE = 1.36$ for SGD as shown in figure 3).

In general it seems OLS and Ridge performed almost the same, it is hard to distinguish them, at least with the chosen $\lambda = 0.001$.

B. Neural network regression

With the neural network constructed using the Sigmoid activation function, the correlation to our second order polynomial using SGD with ADAM is found to have $MSE = 21.7$ and $R^2 = 0.98$ with the related optimal parameters.

Using `Scikit-learn` was much worse with the same parameters with the values; $MSE = 122.6$ and $R^2 = 0.88$. Something important to note is how we used the parameters which were optimal for the neural network on the `Scikit-learn` as well, which actually make this model representation less accurate (there are most likely parameters which would give a better representation while using `Scikit-learn`).

C. Comparing activation functions

Replacing the Sigmoid function with ReLU and leaky ReLU as the hidden activations improved performance greatly. Almost every combination of parameters tested in the grid plot with these provided a determination R^2 over 0.96, and most MSE's were below 10. So, not only was the best performances better using Sigmoid (the highest R^2 was actually 1 which means perfect test prediction), but also the performances were better for many more parameter combinations.

D. Neural network classification

Our experimentation with different model configuration and parameters showed that on average and in terms of absolute accuracy the leaky ReLU function had the best prediction accuracy of 0.98. leaky ReLU had on average a accuracy of 0.95, ReLU 0.93 and Sigmoid 0.77. Comparing results from plain gradient descent and optimized (see tables 1, 2) show that the 2 hidden layer configuration performed roughly 5-15% worse compared to 1 hidden layer using plain gradient descent. In contrast, using SGD and Adam for optimization showed the 2 layer configuration to increase accuracy by roughly 5-15%, for all three activation functions. Changing the number of nodes in the hidden layers does not seem to have an consistent effect on the performance, thus we may rule out the intuitive thought that more nodes is better. It seems that there is not a clear cut model configuration that gives good performance. This is an interesting observation as it shows the various ways in which model performance is affected using different configurations. Nevertheless, the best prediction accuracy of 0.98 was obtained using optimized model with 2 hidden layers.

Taking a closer look at the results from the optimized method (2) we can see that ReLU and leaky ReLU consistently perform well across all model configurations with an accuracy of ≥ 0.89, whereas the Sigmoid show accuracies spanning 0.61 to 0.94. This could be caused by the Sigmoid function's inherent limitation of saturation where the gradients become zero and learning stops (Dorsaf, 2021). leaky ReLU does not have this limitation, whereas ReLU's limitation of negative values being set to zero and causing saturation does not seem to have occurred for our model.

It is reassuring that the ReLU function performed almost equally well in terms of accuracy, because when we take a closer look at computational efficiency the ReLU was far superior to the leaky ReLU. More specifically, the ReLU computed the results in 53 seconds, whereas the leaky ReLU used 244 seconds (see 11, 12) i.e leaky ReLU used almost 5 times longer to perform the same grid search with the same number of layers, nodes and parameters. This finding corroborate with the observation from (Dubey, Singh, & Chaudhuri, 2021) noting that for simple classification models leaky ReLU performs better, but ReLU is more efficient. In our project this was not an issue, as our goal was to explore our model, not a analysis of the data. However, for larger projects where computational time matters, it becomes important to think about this dilemma. In some cases it might be better to trade a few accuracy score points for 5 times faster computation.

The `MLPClassifier` does not have leaky ReLU as an activation function, thus we are not able to compare our results with `MLPClassifier`. For the Sigmoid function the `MLPClassifier` outperformed our model with an accuracy score of 0.97, whereas our best model resulted in 0.94. However, for the ReLU function our best model resulted in 0.97, whereas the `MLPClassifier` 0.96. Thus for the ReLU function our model outperformed the `MLPClassifier` by a insignificant numerical amount, but had an significant effect on our morale.

For the learning rate and regularization parameters both ReLU and leaky ReLU had a similar grid plot pattern (11, 12). For the ReLU a learning rate of 10^{-1} and regularization terms ranging from 10^{-5} to 10^{-1} resulted in accuracies ≥ 0.92. For the leaky ReLU learning rates of 10^{-1} and 1 and regularization terms ranging from 10^{-5} to 1 resulted in accuracies ≥ 0.91. The Sigmoid show an accuracy of ≥ 0.87 for a learning rate of 10^{-1} and regularization terms 10^{-5}, 10^{-3} and 1 (10).

This variation in optimal parameters demonstrates the usefulness exploring various parameters and visualizing results, because it is clear that there is not one clear parameters combination that give good results, but there can be several combination of parameters and also model configurations that could result in good accuracy.

E. Logistic regression

It seems that classification by plain logistic regression outperformed our NN in general for all activations and parameter combinations. We also used SGD with ADAM for this part which improved the performance, however not when using the NN as previously discussed. This is not what we expected, we expect that a neural network would perform marginally better. There may be some influence regarding our implementations, for example for the logistic regression we found it easier to just use the NN code with no hidden layers at all, as this would be equal to plain regression with no net. However for the previous section we used our previous non-class-based NN code which we believe may not be implemented as well, especially as Morten's implementation from the lecture notes that we had to resort to using (more on this is the final section of our discussion). Especially unexpected was that, Scikit's `LogisticRegression` also outperformed `MLPClassifier` which was surprising. This leads us to believe that in this case, logistic regression is a better fit for binary classification than a classification neural network, or that there may be some other systematic bias that we have overlooked.

F. Evaluation of the various algorithms

Calculating the gradients using the analytical expressions and automatic differentiation performed the same as expected, however using analytical provides a faster running code, which is optimal (at least when we have the expressions already calculated, or they are easy to calculate). We expected SGD with ADAM to be the best overall gradient descent implementation, however our results did not show that, as previously mentioned.

Moving on to the neural network implementation and

using it for regression, our code actually outperformed `Scikit-learn` for the chosen parameters with a MSE about ten times lower, and 10% higher R^2. However, testing with different parameters `Scikit-learn` performs just as good and sometimes slightly better, this is something to keep in mind if you were to choose which implementation to use. `Scikit-learn` is most likely the better implementation overall, however the library `Tensorflow` might be even better suited to neural networks so also look into that. We opted to use automatic differentiation when implementing the NN to save time spent on calculating the analytical gradients. The exception is for NN classification, where we used analytical expressions as we had already calculated these during our development.

It is important to note here that our NN implementation did not work with regression, there is some bug somewhere that we spent too long trying to debug, therefore we had to convert to using the lecture code's implementation of the same class-based FFNN by Morten. This is unfortunate, but you could still find all our implementations on the Github repository (appendix A).

We then compared the two other activation functions (ReLU and leaky ReLU) to the Sigmoid function used in the NN above and they both provided much better performance for every parameter combination. We got up to 400 times lower MSE and a 166% better determination R^2 with both the ReLU functions (comparing the non-overflow-like MSE values which were above 1000). As mentioned both the ReLU's had almost identical performance, the difference in optimal MSE's were only $\approx 3\%$ difference and they both reached perfect determination scores of $R^2 = 1$. This leads us to weigh both the ReLU functions as the optimal activations for our polynomial simulated data.

In part D we studied NN classification of 'Wisconsin Breast Cancer' data set (Wolberg, 1992). We implemented various configuration of learning rates, hyperparameters and number of nodes and layers in the network. In line with our results from the regression problem, ReLU and leaky ReLU also performed better on the classification problem compared two the Sigmoid function. This is not suprising as we know the Sigmoid function limitation, namely saturation of neurons where gradients become zero and learning stops. leaky ReLU does not have this limitation, but ReLU has for zero values - however, this was not a problem in our model. In terms of absolute numerical value, leaky ReLU performed better on average by 2% which could be considered an insignificant difference. Nevertheless, the leaky ReLU was 5 times slower in terms of computation time, thus we could argue that the ReLU is a more suitable activation function in situations where computation time matters. For larger data sets and projects where computation time matters, it becomes important to consider the trade off between 2% better accuracy versus 5 times faster computation time.

Thus we could argue based on our results and observations that different models and different problems can be optimized in different ways and with different configurations. As many other things in life there does not seem to be a 'one size fits all' for machine learning models. A model needs to be adjusted for the problem at hand and experimentation with various parameter and model configurations is required for developing a good model.

VI. CONCLUSION

Throughout this project we choose to focus on the usage of ADAM and SGD. Using Ridge regression we achieved a test prediction with the $MSE = 1.36$ with the optimal parameters $\eta = 0.3$, $\lambda = 0.001$, $n_{epoch} = 50$ and $m = 20$ minibatches.

Moving on to neural network regression we found that using ReLU or leaky ReLU activation provided the best predictions $MSE = 1.5$ and $R^2 = 1$ with the optimal parameters $\eta = 0.01$, $\lambda = 0.1$, and $m \geq 15$. These were significantly better than using Sigmoid activation.

For the classification analysis our best prediction accuracy was 0.98 which was obtained with leaky ReLU using 2 layers and 20 nodes. In general, the leaky ReLU performed best across various model configurations with an average score of 0.95, while ReLU performed on average 0.93 and Sigmoid at 0.77. Although, leaky ReLU was 2% faster than ReLU it was also five times slower in terms of computation time. With this in consideration our best model for classification analysis was ReLU.

In the end we applied a logistic regression method to the breast cancer data, which resulted in a better accuracy at 0.96 compared to the classification problem using a neural network.

Overall results seem to indicate that there is not a 'one size fits all' method for our machine learning problems. We learned different problems require different methods, and that it is important to experiment with various configurations and explore how the models learn from the data. This correlates to the total picture of our results.

Appendix A. Github repository

https://github.com/LassePladsen/FYS-STK3155-projects/tree/main/project2

Appendix B. List of source code

Here is a list of the code we have developed in this project which can be found in the above Github repository:

- `part_a_GD.py` - gradient descent methods and analysis
- `part_b_creating_neural_network_code.ipynb` - creating our neural network
- `part_b_ffnn.py` - object-oriented implementation of our neural network

- `part_b_activation.py` - collection of activation functions
- `part_b_cost.py` - collection of cost functions
- `part_b&c_regression.py` - neural network regression analysis
- `part_d_1_layer.ipynb` - neural network classification using 1 hidden
- `part_d_1_layer_plain.ipynb` - neural network classification using 1 hidden plain gradient descent.
- `part_d_2_layers.ipynb` - neural network binary classification using 2 hidden layers
- `part_d_2_layer_plain.ipynb` - neural network classification using 2 hidden layers plain gradient descent.
- `part_d_testing.ipynb` - plain gradient descent implementation
- `part_e_logreg.py` - logistic regression classification analysis

Appendix C. Automatic gradient descent

In figure 15 we can see the MSE result we get in gradient descent when using automatic gradient. If you compare it to the analytical gradient results it is possible to see similarities, but the analytical instance more often converge faster with smaller MSE.

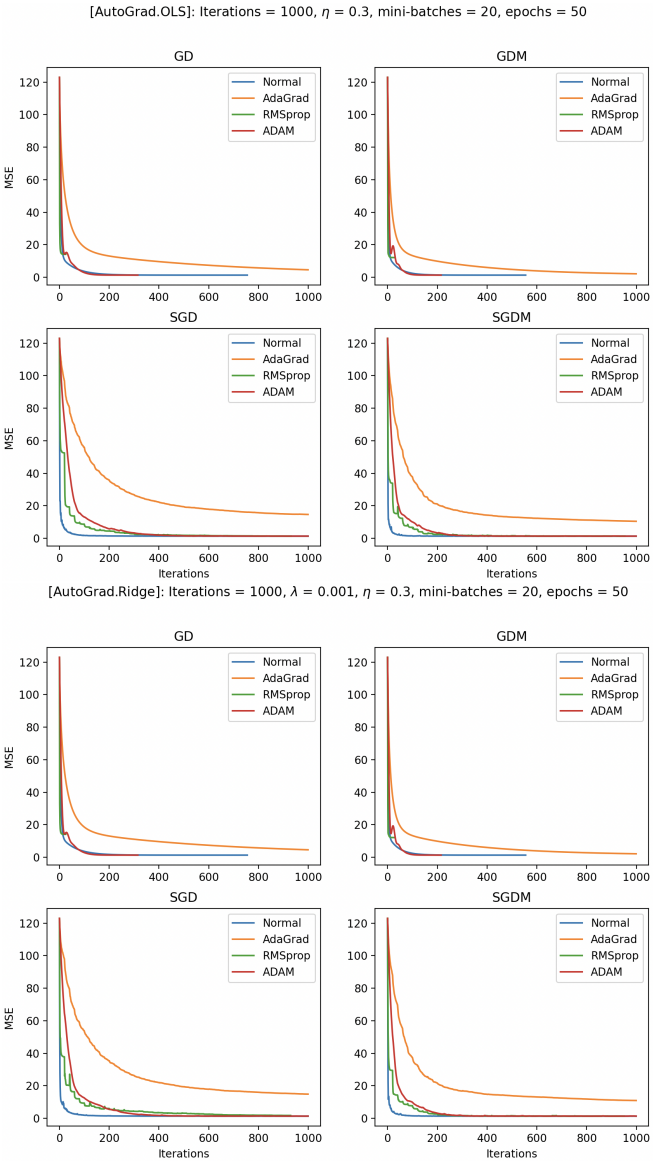

Figure 15: Plot of gradient descent and scaling methods using automatic gradient with optimal regularization hyper parameter λ, learning rate η, and number of epochs and minibatches. The four first plots are with OLS and the last four are with Ridge. We do this mostly to confirm our results for the analytical gradient.

References

Dorsaf, S. (2021). *Comprehensive synthesis of the main activation functions pros and cons.* Retrieved 2023-11-29, from https://medium.com/analytics-vidhya/comprehensive-synthesis-of-the-main-activation-functions-pros-and-cons-dab105fe4b3b

Dubey, S. R., Singh, S. K., & Chaudhuri, B. B. (2021). A comprehensive survey and performance analysis of activation functions in deep learning. *CoRR, abs/2109.14545*. Retrieved from https://arxiv.org/abs/2109.14545

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* The MIT Press, Cambridge, Massachusetts. (http://www.deeplearningbook.org)

Hjorth-Jensen, M. (2023). *Applied data analysis and machine learning.* https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html. ([Online; accessed 30-September-2023])

Wolberg, W. (1992). *Breast cancer wisconsin.* https://archive.ics.uci.edu/dataset/15/breast+cancer+wisconsin+original. ([Online; accessed 07-November-2023])