FYS-STK4155 - Project 2

Tor-Andreas Bjone, Fredrik Junker Pedersen, Jens Junker Pedersen

November 18, 2022

Contents

Abstract

In this project we did implement our own Feed Forward Neural Network (FFNN) with numerous optimization methods.

We started by fitting a second order polynomial with the gradient decent methods of plain GD and stochastic with the AdaGrad and ADAM and RMSProp optimizers. We found that the stochastic methods in general converged faster (less iterations to reach low MSE) with larger minibatches.

Adding momentum was beneficial for all the gradient descent methods, except when utilizing the ADAM optimizer. We discovered that the ADAM optimizer did a lot worse than the other optimzers and than without any optimizer. AdaGrad and RMSprop performed similarly and these optimizer excelled ecspecially when reducing the minibatch sizes.

AdaGrad was the most suitable optimizer for the polynomial fitting problem and produced a perfect fit to the target polynomial on eyesight.

When moving over to polynomial fitting with a Neural Network (NN) we found that a dense neural network with one hidden layer of 5 neurons produced the best results. We analyzed the results using activation function Sigmoid, ReLU and leaky ReLU. Each activation function had its advantages and disadvantages with Sigmoid obtaining the best MSE score, but being most probable to get stuck on poor MSE scores, while the opposite was true for leaky ReLU, and was in the midle with regards to both properties.

The resulting fit after 2000 iterations using Sigmoid was pretty similar to the target, allthough we could clearly see some deviations.

In the last part we did a classification analysis on the Wisconsin Breast Cancer data. We experimented with different hyperparameter combinations. In our last run we tried to find the overall best parameter combination. This resulted in an accuracy score of 0.9825 on our test data. According to the literature the result is better than what is to be expected for the Wisconsin Breast Cancer data. The reason is that we did not include any re sampling methods when we trained our model.

We did find an error in the calculation of our L2 regularization parameter, λ. λ was multiplied with the gradient of the weights instead of the weights from the previous iteration, effectively acting as an increased learning rate. Our analysis of the L2-regularization parameter is therefore invalid and should not be cited.

1 Introduction

In this report we will look at Gradient decent (GD) methods and Neural Networks (NN) and it will be heavily based on the lecture notes by Morten Hjorth-Jensen [2].

GD methods are an effective way to find the local (and hopefully global) minimum of a function. And it can not be understated how important GD methods are in machine learning, statistics and optimization because of their simple implementation and computational effectiveness.

Neural networks are a subset of machine learning based on artificial neurons, weights and biases. These are important in many field and the use of NNs are growing rapidly. They are especially useful in medicine as they can be trained to recognize patterns in images much better than a human can. But we will likely see neural networks implemented for a huge number of applications in almost all industries in a couple of years. We will look at the benefits and disadvantages of neural networks in different use cases such as polynomial fitting and look at the optimal parameters for the network in those cases.

First we will introduce the OLS and Ridge method of regression, which will be tested against the GD methods. Then we look at some of the most common GD methods. Namely plain GD, Stochastic GD, GD with momentum, and RMSProp, AdaGrad and Adam.

Then we move over to the neural network part of this report. We look at how to initialize the network, the Feed Forward algorithm, different activation functions and training using back propagation. We will then test the network on polynomial fitting against regression methods.

After this we will test the network on a classification problem, namely the Wisconsin breast cancer dataset provided by sklearn. And at this point we will also implement logistic regression as this is suited for this particular dataset.

2 Method

2.1 Regression methods

Say we have a response $\mathbf{y} \in \mathbb{R}^n$ to p-number of features $\mathbf{x}_i = [x_{i0}, x_{i1}, ..., x_{ip-1}]$, where the set of these $\mathbf{X} = [\mathbf{x}_0\ \mathbf{x}_1\ ...\ \mathbf{x}_{n-1}]$ is called the design matrix of the model. The point of regression analysis is then to find a assume linear relationship between $\mathbf{X}$ and $\mathbf{y}$. This assumption gives rise to the linear regression model where $\boldsymbol{\beta} = [\beta_0, \beta_1, ..., \beta_{p-1}]^T$ are the regression parameters and the error variable $\boldsymbol{\epsilon}$ is an unobserved random variable that adds "noise" to the linear relationship between the dependent variable and regressors. This gives the model

$$\tilde{y}(x_i) = \sum_{j=0}^{p-1} \beta_j x_{ij} = \mathbf{X}_{i*}\boldsymbol{\beta}.$$

Or on vector form

$$\tilde{\boldsymbol{y}} = \mathbf{X}\boldsymbol{\beta}.$$

We can then re-write the response in terms of the model and noise as

$$\begin{aligned} \mathbf{y} &= \tilde{\boldsymbol{y}} + \boldsymbol{\epsilon} \\ &= \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}. \end{aligned}$$

2.1.1 Ordinary least squares (OLS)

The method of ordinary least squares computes the unique line that minimises the sum of squared differences between the true data and that line. In other words we have an optimisation problem

on the form

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2 = \frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}||_2^2,$$

where we have used the definition of a norm-2 vector, that is

$$||\boldsymbol{x}||_2 = \sqrt{\sum_i x_i^2}.$$

We use this to define the cost function of OLS as

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

The $\boldsymbol{\beta}$ that optimizes this we call $\hat{\boldsymbol{\beta}}_{OLS}$, and looking at the cost function we see that this will be when the gradient is zero.

2.1.2 Ridge

The ordinary least squares method can be inaccurate when the model have highly correlated independent variables. Ridge regression tries to solve this by adding a regularization parameter λ, called a hyperparameter, to the optimization problem. We start by re-writing the optimization problem as

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda||\boldsymbol{\beta}||_2^2,$$

where we require that $||\boldsymbol{\beta}||_2^2 \leq t$, where t is a finite number larger than zero. Our cost function the becomes

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

As with the OLS this is minimized when the gradient is zero, and we call the $\boldsymbol{\beta}$ that optimizes this for $\hat{\boldsymbol{\beta}}_{Ridge}$.

2.2 Cost-function

For both of the regression methods presented here they have what is called a cost-function $C(\boldsymbol{X},\boldsymbol{\beta})$. This is a measure of how close the predicted solution is to the true solution (often called the target). You can define many different cost functions, but in the above we used the mean squared error. The reason the OLS and Ridge regression methods are useful when evaluating a model is twofold. They have nice derivatives, which makes them easy to implement in a gradient method. And they have closed-form analytical expressions that can be found by deriving the cost-function with respect to $\boldsymbol{\beta}$, which we can then use to compare to our predicted solutions.

2.3 Gradient methods

The algorithms presented in this section are taken from chapter 8 of the book Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville [1].

The goal of all gradient methods is to minimize the cost function. And we do this by iteratively calculating the gradient of the cost function and stepping in the direction where it gets smaller, updating the parameters and doing the calculations again. The easiest of the gradient methods is the plain gradient decent, which we will look at first. All of these gradient decent methods have a stopping criterion that is usually given by a maximum number of iterations or when the step size is smaller than a given value.

2.3.1 Plain gradient decent

Algorithm 1 The plain gradient decent algorithm

Require: Learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met **do**
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{N}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\boldsymbol{y}^{(i)})$. ($N$ is size of training set)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$
  end while

In the algorithm for the plain gradient decent (1) we see that there is an added term ϵ (or often η) which is called the learning rate. This is implemented to regulate the step-size so that we don't easily overstep the minimum of the cost-function. This learning rate is not always constant and we then call it the learning schedule. For example it is usual to use a learning schedule that decays with the number of iterations as can be seen in the function below.

```
FUNCTION epsilon(t, t0, t1):
    return t0/(t+t1)
ENDFUNCTION
```

2.3.2 Stochastic gradient decent (SGD)

The problem with plain gradient decent is that when the size of the dataset gets big, so does the computation time. Stochastic gradient decent (2) fixes this by splitting the dataset into mini-batches and calculating the gradient only on this mini-batch of the dataset. When this mini-batch is chosen at random it should approximate the gradient, given that the dataset is smooth and well behaved.

Algorithm 2 The SGD algorithm

Require: Learning rate schedule $\epsilon_1, \epsilon_2, ...$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\boldsymbol{y}^{(i)})$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \boldsymbol{g}$
    $k \leftarrow k + 1$
  end while

2.3.3 SGD with momentum

One simple way of improving the convergence rate of the SGD method, is to include a momentum term:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta E(\theta) + \gamma(\theta_t - \theta_{t-1}),$$

where γ is a momentum parameter, with $0 \leq \gamma \leq 1$. That last term serves as a memory term. Our goal is to find the global minima where the gradient is zero. Our ordinary gradient descent method is sensitive to noise, and local variations in our data. This often lead to a behavior where our values for θ oscillates towards the global minima. By introducing a momentum term our gradient method will better be able to move directly towards the global minima. This is due to information about the previous gradients in our update scheme. This can be implemented by the algorithm (3)

Algorithm 3 The SGD with momentum algorithm

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \boldsymbol{y}^{(i)})$.
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

2.3.4 RMSProp, AdaGrad and Adam

In stochastic gradient descent, with and without momentum, we still have to specify a schedule for tuning the learning rates as a function of time. As discussed in the context of Newton's method, this presents a number of dilemmas. The learning rate is limited by the steepest direction which can change depending on the current position in the landscape. To circumvent this problem, ideally our algorithm would keep track of curvature and take large steps in shallow, flat directions and small steps in steep, narrow directions. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature. However, this is very computationally expensive for extremely large models. Ideally, we would like to be able to adaptively change the step size to match the landscape without paying the steep computational price of calculating or approximating Hessians. Recently, a number of methods have been introduced that accomplish this by tracking not only the gradient, but also the second moment of the gradient. These methods include AdaGrad, AdaDelta, Root Mean Squared Propagation (RMS-Prop), and ADAM. In the following algorithms $\odot$ is the Hadamat product, that is an element-wise array operation.

In RMS prop (4), in addition to keeping a running average of the first moment of the gradient, we also keep track of the second moment.

It is clear from this algorithm that the learning rate is reduced in directions where the norm of the gradient is consistently large. This greatly speeds up the convergence by allowing us to use a larger learning rate for flat directions.

A related algorithm is the ADAM optimizer (5). In ADAM, we keep a running average of both the first and second moment of the gradient and use this information to adaptively change

Algorithm 4 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho\boldsymbol{r}+(1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$.
Compute parameter update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta+\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{1+\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

the learning rate for different parameters. The method isefficient when working with large problems involving lots data and/or parameters. It is a combination of the gradient descent with momentum algorithm and the RMSprop algorithm discussed above.

Algorithm 5 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for momentum estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = 0$ and $\boldsymbol{r} = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\boldsymbol{y}^{(i)})$.
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1\boldsymbol{s} + (1-\rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2\boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{1}{1-\rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
Compute parameter update: $\Delta\boldsymbol{\theta} = -\epsilon\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

The AdaGrad algorithm (6) individually adapts the learning rate for each of the parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient. This makes the parameters with larger derivatives of the loss function get a bigger decrease in the learning rate than the parameters with smaller derivatives of the loss function.

Algorithm 6 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
Compute parameter update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

2.4 Gradient descent polynomial fitting

We will use gradient descent implemented in Python to fit a second order polynomial

$$f(x) = x^2 + 1 \tag{3}$$

with 100 linearly spaced points $x\epsilon[-1, 1]$. The data are then split in to training and test sets with 80% and 20% randomly selected data respectively.

Our prediction is the matrix product of the second order polynomial design matrix $\mathbf{X}$ and corresponding coefficient column vector θ

$$\mathbf{y_{pred}} = \mathbf{X}\theta. \tag{4}$$

θ were initialized with values sampled from a normal distibution with mean 0 and variance 1. The gradients were calculated with respect to θ by `autograd's grad` function on the Ordinary Least Squares or Ridge (if L2 regularization parameter $\lambda \neq 0$) cost function.

For each epoch in stochastic gradient descent, a minibatch of size m were randomly sampled n/m times without replacement from the data of size n, such that every data point were utilized in each epoch.

To find the best prediction, we analyzed the mean squared error resulting from different parameters described in table 1-3. To reproduce our results, run the file `results_gradientdescent.py` in the following Github repository `https://github.com/fredrikjp/FYS-STK4155/tree/master/Project2`

Table 1: Parameters utilized in polynomial fitting using gradient descent for run 1 and 2

Parameter	Run 1	Run 2
η	`linspace(0.6, 1, 5)`	`linspace(0.1, 1, 10)`
λ	$(0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1})$	0
γ	0.1	(0, 0.2, 0.4, 0.6, 0.8)
gd epochs	100	N/A
sgd epochs	25	25
mini batch size	20	20
RMSprop rho	N/A	0.99
ADAM (beta1, beta2)	N/A	N/A

Table 2: Parameters utilized in polynomial fitting using gradient descent for run 3 and 4. The parameter values in square brackets "[]" for the tuning tuning methods in the column title in corresponding order

Parameter	Run 3	Run 4: [Plain, AdaGrad, RMSprop, ADAM]
η	`logspace(-3, 0, 10)`	[0.9, 0.9, 0.9, 0.2]
λ	0	0
γ	(0, 0.2, 0.4, 0.6, 0.8)	[0.4, 0.4, 0.4, 0]
gd epochs	N/A	N/A
(mini batch size, sgd epochs)	(20, 25)	((2, 5), (5,12), (10,25), (80,200))
RMSprop rho	N/A	0.99
ADAM (beta1, beta2)	(0.9, 0.99)	(0.9, 0.99)

Table 3: Parameters utilized in polynomial fitting using gradient descent for run 5 and 6.

Parameter	Run 5	Run 6
η	0.9	
λ	0	
γ	0.4	
sgd epochs	20	
mini batch size	10	

The parameters in table 1-3 are learning rate η, L2 regularization parameter λ, momentum parameter γ, number of gradient descent epochs "gd epochs", number of stochastic gradient descent epochs "sgd epochs", stochastic gradient descent minibatch size "mini batch size", RMSprop parameter "RMSprop rho" and ADAM parameters "ADAM (beta1, beta2)".

2.5 Neural network

A neural network is a computer system that works kind of like the brain. It is composed of a set of interconnected processing nodes, or neurons, that activates using weight and biases. This network can be dense, meaning that all neurons in a layer is connected to all neurons in the previous and next layer. The reason they are called neuron is that they mimics the behavior of a biological neuron. We will look specifically at a feed-forward multi-layer perception (MPL) neural network, often called a dense neural network.

2.5.1 Initialization

One of the most important aspects of training a neural network is the initialization of the weights and biases. This process sets the starting point for the network and can have a significant impact on the performance of the model. There are a number of different methods that can be used to initialize the weights and biases, and the choice of method can be critical for the success of the training process. Some of the more common methods include random initialization, Xavier initialization, and He initialization.
We will use random initialization, meaning we will set the weights using a random distribution. And the biases we will set to 0.001. It is common to initialize the biases as a zero, or a small number to assure that the network don't die. We also need to be careful to not set the weights to high as this might cause the network to explode.

2.5.2 Feed-forward

Feed-forward means that the processing of information in the network only flows trough the nodes in one direction, from the input layer to the output layer.
To implement this we let the first layer of the network get the input. Then we use an activation function on that layer, send that to the next layer, activation, ..., and so on until we reach the output layer. Mathematically

$$a_L = \sigma((z_L)) = \sigma(a_{L-1}) = \sigma(\sigma(z_{L-1})) = ...$$

until we reach the first layer, and the input. Next we will talk about these activation functions.

2.5.3 Activation functions

A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). The following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem
- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions. This is typically done with the Sigmoid or hyperbolic tangent function.

Sigmoid and Hyperbolic tangent: The sigmoid activation function

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

is an ideal activation function for a hidden layer given it's easy derivative

$$\sigma'(z) = (1 - \sigma(z))z.$$

This is a widely used activation function, but one must be careful of overflow due to the exponential term. The hyperbolic tangent activation function

$$\sigma(z) = tanh(z),$$

with the derivative

$$\sigma'(z) = sech^2(z)$$

behaves much like the sigmoid except is has an output from negative one to plus one, compared to the sigmoid which has an output from zero to one. Both of these functions have a problem in many-layer networks, and that is the vanishing gradients problem. Therefore in many-layer networks other functions are ideal, such as ReLU.

ReLU: The rectified linear unit (ReLU) is piecewise linear and will output the input directly if it is positive and will output zero if not. It is as follows

$$\sigma(z) = argmax(0, z),$$

and has a split derivative, where $\sigma'(z) = 1$ if $z \geq 0$ or else it is zero. This is also an ideal activation function for a hidden layer and does much better than sigmoid or hyperbolic tangent at many-layer networks. It is the default activation function for MLPs and convolutional neural networks (CNN). And by solving the vanishing gradients problem it makes the network learn faster and preform better.

Leaky ReLU: The leaky ReLU activation function has a small slope for negative values instead of zero. So that $\sigma(x) = ax$ when $x < 0$ and this slope is determined before training. In other words it is not a hyperparameter. Other than that it is exactly as ReLU.

Softmax: The Softmax activation function is a little different than the other functions in that it produces a normalized probability distribution over the predicted output classes and it is therefore often used as the activation for the output layer of the network.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}}.$$

This inputs a vector $\vec{z}$ and outputs a normalized probability distribution. The probability distribution is handled in the numerator and the normalization in the denominator. It is also often useful to re-define

$$z_i \leftarrow z_i - argmax(\vec{z_i})$$

as to prevent overflow in the exponential. This will not affect the probabilities.

2.5.4 Training and evaluation

To train the neural network we do what is called back-propagation. This is a way to adjust the weights and biases to minimize the error in the predictions made by the network. The steps of the back-propagation algorithm is as follows. First we calculate the error in the L'th layer, the output layer

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial(a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \ldots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 2$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

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

We keep doing this until we reach some max number of training steps or the gradients gets sufficiently small.

To evaluate the performance of the network we will look at the square of the difference between the wanted outcome, or target, t and the output of the network. We call this the accuracy of the network.

2.6 Neural Network polynomial fitting

Our Neural Network implemented in Python was used to fit the same polynomial as the one fitted with gradient descent in equation 3, but now with 1000 linearly spaced input values $x \epsilon [-1, 1]$. As this is a regression problem, we did not use any activation functions on the output layer, and the mean squared error was employed as the cost function in our neural network. We used sigmoid, relu or leaky relu as activation function for the hidden layers. The wheights were initialized using a normal distribution with variance 1 and mean 0, and the biases were initialized with value 0.01.

To optimize the polynomial fit, we studied the mean squared error of our trained network with different parameters described in table 4-6. To reproduce the results, run the file `results_NN_polyfit.py` in the following GitHub repository: `https://github.com/fredrikjp/FYS-STK4155/tree/master/Project2`.

Table 4: Neural network polynomial fit parameters for run 1 and 2

Parameter	Run 1	Run 2
eta	`linspace(0.05, 0.5, 5)`	`logspace(-7, -1, 7)`
depth	1	1
width	5	5
activation hidden	sigmoid	relu
gamma	0	0
lambd	$(0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$	$(0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$
n minibatches	10	10
epochs	2000	2000

For benchmarking we used `sklearn MLPRegressor` on "etaa" array with different learning rates and "lambd" array with different L2 regularization parameters as shown in the code below:

```
from sklearn.linear_model import LogisticRegression
```

Table 5: Neural network polynomial fit parameters for run 3 and 4

Parameter	Run 3	Run 4
eta	`logspace(-7, -1, 7)`	`linspace(0.1, 2, 5)`
depth	1	1
width	5	5
activation hidden	leaky relu	sigmoid
gamma	0	0
lambd	$(0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$	$(0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$
n minibatches	10	10
epochs	2000	2000

Table 6: Neural network polynomial fit parameters for run 5 and 6

Parameter	Run 5	Run 6
eta	`logspace(-6, 0, 7)`	0.3875
depth	1	1, 2, 3
width	5	5, 10, 20
activation hidden	relu	sigmoid
gamma	0	0
lambd	$(0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$	0
n minibatches	10	10
epochs	2000	2000

```
j=0
for eta in etaa:
    i = 0
    for lmb in lambd:
        NN_sigmoid = MLPRegressor(hidden_layer_sizes=(5), activation="logistic",
    solver="sgd", alpha=lmb, batch_size = batch_size, learning_rate_init = eta,
    momentum = 0, max_iter=n_epochs ,n_iter_no_change=2000)
        NN_sigmoid.fit(x_train, y_train.ravel())  # x_train column vector with
    input values, y_train column vector with target values
        MSE[j, i] = NN_sigmoid.loss_
        i+=1
    j+=1
```

2.7 Classification problems

Classes in neural networks are defined by the output of the network. For example, if the output of the network is a single class, then the network is said to be a single-class network. If the output of the network is two classes, then the network is said to be a two-class network. And so on.

A common classification problem is image classification. In this problem, the input to the network is an image, and the output is the class of the image. For example, the output might be "cat", "dog", "bird", etc.

We have chosen to train our network using the Wisconsin breast cancer dataset by sklearn.

There are 569 samples in the Wisconsin breast cancer data set and it has two classes, malignant and benign. Each of the samples has 30 features such as the patients' age, the stage of their

cancer, size of the tumor, etc. This dataset is very useful in neural network classification problems because it is a well-known data set that has been used extensively in research. Additionally, the data set is small enough that it can be used to train a neural network without requiring much computational power.

2.7.1 Cross Entropy

Cross entropy is a measure of how well a set of predicted probabilities match the actual classification. If the predicted probabilities are exactly correct, then the cross entropy will be zero. If the predicted probabilities are far from the actual classification, then the cross entropy will be large.

We will use the binary cross entropy as our cost function. We will interpret the Sigmoid function as the probability that our sample is benign, $P(y = 1)$ (see section 2.7.3) The expression for the binary cross entropy is; $-log(P(y = 1))$ if the sample belong to calss 1 (benign), and $-log(1 - P(y = 1))$ if the sample belongs to class 0 (malignant).

If we sum over all the n data points we get our cost function [2]:

$$\mathcal{C}(\boldsymbol{\theta}) = -\ln P(\mathcal{D} \mid \boldsymbol{\theta}) = -\sum_{i=1}^{n} y_i \ln[P(y_i = 0)] + (1 - y_i) \ln[1 - P(y_i = 0)]$$

, where $\mathcal{D}$ is our datset, $\boldsymbol{\theta}$ is our hyper parameters and y_i is the target value.

2.7.2 Accuracy

We used the accuracy score to measure the performance of our model. The accuracy score is defined as:

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

where n is the number of samples. I is equal to 1 if we correctly predicted the target values and 0 else.

2.7.3 Binary classification with Neural Network

All code was implemented and analyzed in Python. Mentioned function names refers to specific python packages and modules. To reproduce the figures, run the file with name `test_analysis.py` in the following GitHub repository, `https://github.com/jensjpedersen/Projects_FYS-STK4155/tree/main/Project2`

Our cancer data was split into training and test data, with a test size of 20%. Scikit-learn's `train_test_split` method was used for this purpose. In order to keep the dataset splits consistent between runs, the `random_state` parameter was set to zero. The flowing code can be used to reproduce our data.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
np.random.seed(5)
cancer = load_breast_cancer()
targets = cancer.target[:,np.newaxis]
```

```
test_size = 0.2
features = cancer.feature_names
X_train, X_test, y_train, y_test = train_test_split(cancer.data,targets,
                                    random_state=0, test_size=test_size)
```

Before attempting to train the network the data was scaled, since each feature lives on different scales. All 30 features was scaled by subtracting the mean and dividing by the standard deviation, with scikit-learns `StandardScaler` object.

In order to keep the results consistent between runs, a seed value of 5 was set with the numpy's random module.

Since our data only consist if two classes, malignant and benign we will use the binary Cross Entropy (equation 2.7.1) as our cost function. Where our targets is a column vector with 0's and 1's (malignant and benign).

The Sigmoid function (see (2.5.3)) was used as our activation function in the output layer. We can interpret the sigmoid function as probability function. We then define the output as the probability that our prediction belongs to class 0:

$$P(y = 0|z^L, \theta) = \sigma(z^L),$$

where θ is the weights and the bias, and z^L is the input to the activation function. σ is the sigmoid function. For our binary classification problem it follows that:

$$P(y = 1|z^L, \theta) = 1 - \sigma(z^L).$$

In order to find the optimal values for hyper-parameteres, multiple runs with different combination of hyper-parameter was used. All runs with it's specific hyper-parameters is listed in table 7.

Our predicted accuracy score in Run 6 was benchmarked with Keras (Python API for Tensor Flow), but with some changes to the hyper-parameter values. The following python code was used to predict the accuracy with the Keras api:

```
import tensorflow as tf
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(5, activation=tf.nn.sigmoid))
model.add(tf.keras.layers.Dense(5, activation=tf.nn.sigmoid))
model.add(tf.keras.layers.Dense(1, activation=tf.nn.sigmoid))
opt= tf.keras.optimizers.Adagrad(learning_rate=1)
model.compile(optimizer=opt, loss='BinaryCrossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=300, batch_size=len(y_train))
val_loss, val_acc = model.evaluate(X_test, y_test)
print(val_acc)
```

Table 7: Different runs used to find the optimal parameters of our NN in classification of breast cancer data

Parameter	Run 1	Run 2	Run 3
eta	0.001	0.00001, 0.0001, 0.001, 0.01, 0.1	0.1
depth	1	1	1, 2, 3
width	10	10	5, 10, 15, 20
activation hidden	sigmoid	sigmoid	sigmoid
gamma	0.9	0.9	0.9
lambd	0.0	0, 0.0001, 0.001, 0.01, 0.1	0.1
tuning method	none	none	none
n mini batches	1, 5, 10, 15, 20	20	20
epochs	400	200	200

Table 8: Different runs used to find the optimal parameters of our NN in classification of breast cancer data

Parameter	Run 4	Run 5	Run 6
eta	0.001	0.001	0.001
depth	2	2	2
width	5	5	5
activation hidden	sigmoid	sigmoid, relu, leaky relu	sigmoid
gamma	0.9, 0	0	0.9
lambd	0	0	0.01
tuning method	none, adam, rms prop, adagrad	none	adagrad
n mini batches	20	20	20
epochs	200	200	300

2.7.4 Logistic regression

Logistic regression is a statistical method for classifying data into two groups. It is often used in binary classification problems.

Logistic regression works by using a logistic function to map the data points onto a curve. The logistic function is a sigmoid function (equation 2.5.3), which takes any real valued number and maps it onto a value between 0 and 1. Values above 0.5 is assigned to class 1, and below 0.5 to class 0.

The logistic function can be used for any number of predictors. For multiple predictors the function takes the form:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2+x_2+\ldots+\beta_p x_p)}},$$

where p is the number of predictors. Our problem is then to fit the coefficients β_i for each feature x_i such that our cost function is minimized.

Our developed FNN code can easy be reconfigured to a logistic regression solver as discussed below.

We will again use the Wisconsin Breast Cancer data for our logistic regression problem. The full dataset was used with all 30 features. That data was split and standardized with the same random seed value as in section 2.7.

All hidden layers was the removed from the neural network. Thus, our neural network consisted of an input layer outputting 30 feature vectors, and an output layer of one node and the sigmoid function as the activation function. Our feed forward pass in the network, therefore takes the form as in equation 2.7.4, where β_i is the weights into the output layer. Again, we used cross entropy as our cost function. In order to speed up the convergence rate of the SGD method, we included momentum, $\gamma = 0.9$ in all our calculations. We did experiment with different number of mini-batches, different values for the learning rate γ and L2 regularization parameters λ, to see if we could reproduce the results from our FFNN.

We did use the sklearn package to benchmark our logistic regression result, with the following python code:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print(logreg.score(X_test,y_test))
```

3 Results and Discussion

3.1 Gradient decent methods

Figure 1: Plain gradient descent: test MSE as a function of η and λ. The parameters utilized are shown in table 1 under run 1.

Figure 2: Gradient descent with momentum: test MSE as a function of η and λ. The parameters utilized are shown in table 1 under run 1.

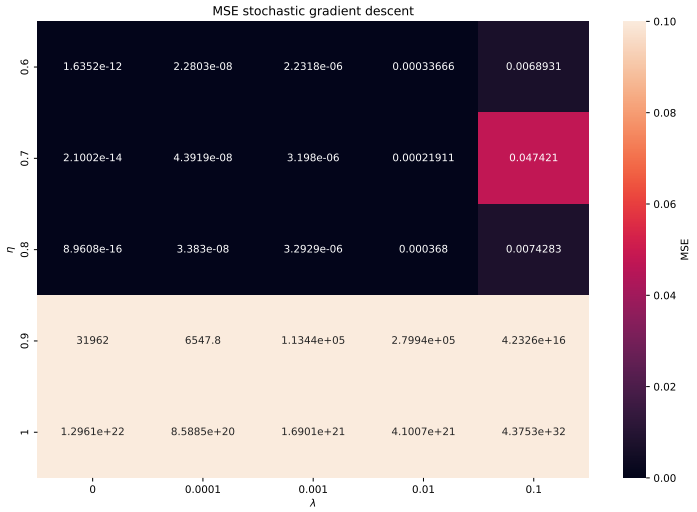


Figure 3: Plain stochastic gradient descent: test MSE as a function of η and λ. The parameters utilized are shown in table 1 under run 1.

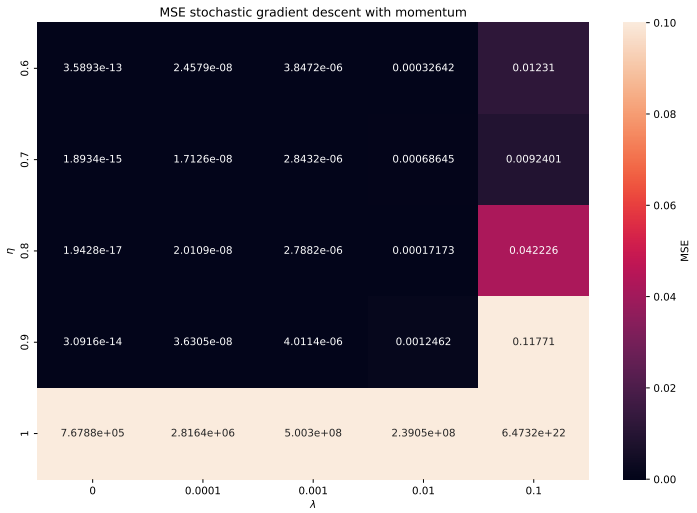


Figure 4: Stochastic gradient descent with momentum: test MSE as a function of η and λ. The parameters utilized are shown in table 1 under run 1.

Figure 1-4 shows MSE scores of the test data for plain and stochastic gradient descent with and without momentum as a function of different learning rates η and L2-regularzation parameters λ. Both plain and stochastic gradient descent has the same amount of total iterations as there are 25 sgd epochs and $n\_data/mini\_batch\_size = 80/20 = 4$ minibatches giving a total amount of $25 \cdot 4 = 100$ iterations for stochastic gradient descent. We observe that we get a worse MSE score with larger learning rates up to a point where the score blows up shown for $\eta = 1$. This makes sense as larger learning rates results in larger update to the parameters θ. Thus, too large learning reate will overshoot the minima of the cost function, while too small will cause slow convergence. We also observe the MSE to increase with increasing L2-regularzation parameter λ. This also makes sense as λ counters overfitting at the cost of poorer training fit, but since we are optimizing the coefficients of a same order polynomial as the target, overfitting will not be a problem. We are therefore better off without the L2-regularzation parameter.

The momentum term is proportional to the previous variable change, such that the components of this vector term that overshoots will be in the opposite direction and thus causing less of a step wich is why we observe the MSE blowing up without momentum, while not blowing up with momentum for $\eta = 0.9$. Momentum also gives better scores as it allows for bigger steps before the minima of the cost function is traversed.

The performance of plain and stochastic gradient descent is similar, with plain being slightly better. As the cost functions are convex, there is no local minimas for plain gradient descent to get stuck on, such that the performance difference will come down to the randomly sampled minibatches in stochastic gradient descent.

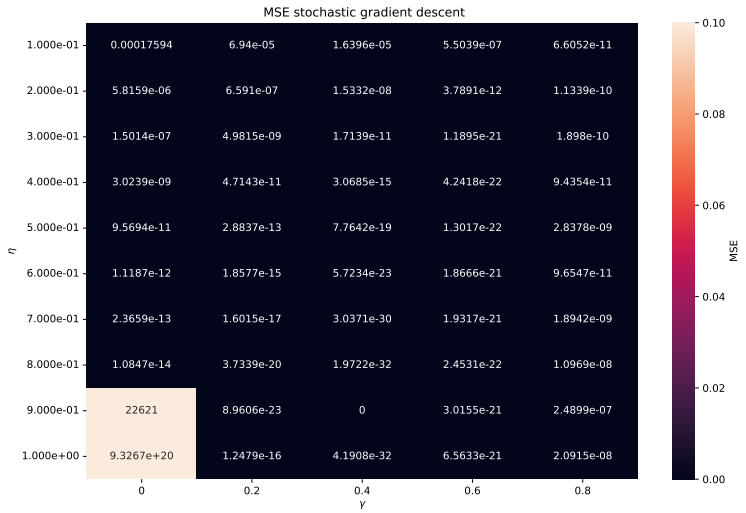


Figure 5: Stochastic gradient descent: MSE as a function of momentum parameter γ and learning rate η. The parameters utilized are shown in table 1 under run 2

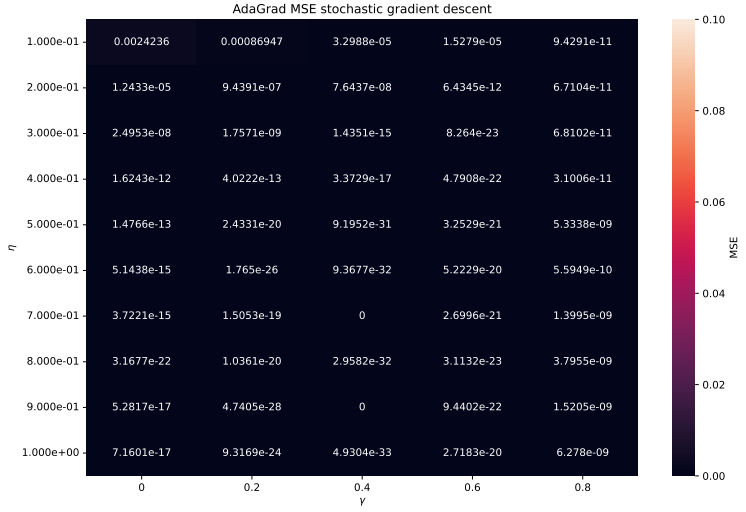


Figure 6: Stochastic gradient descent, with momentum and tuning method AdaGrad: MSE as a function of momentum parameter γ and learning rate η. The parameters utilized are shown in table 1 under run 2

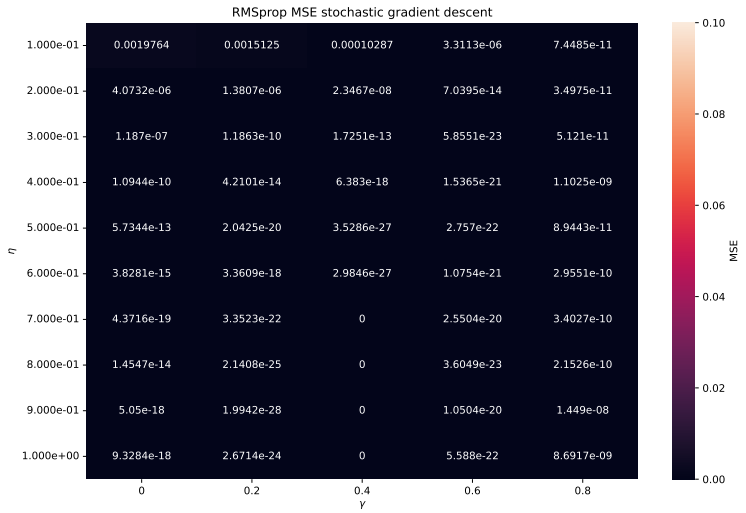


Figure 7: Stochastic gradient descent, with momentum and tuning method RMSprop: MSE as a function of momentum parameter γ and learning rate η. The parameters utilized are shown in table 1 under run 2

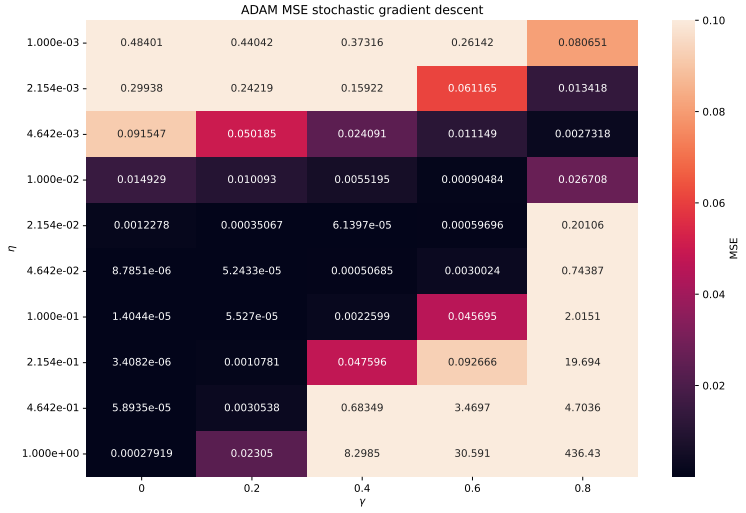


Figure 8: Stochastic gradient descent, with momentum and tuning method ADAM: MSE as a function of momentum parameter γ and learning rate η. The parameters utilized are shown in table 2 under run 3

Figure 5-8, shows the MSE from stochastic gradient descent for different learning rates η and

momentum parameters γ for no tuning method, AdaGrad, RMSprop and ADAM. We observe that ADAM does not benefit from momentum, while AdaGrad, RMSprop, and plain stochastic gradient descent shows best results for momentum parameter $\gamma = 0.4$ with appropriate learning rates η. Generally, the optimal learning rates decreases with grater
gamma which makes sense since more momentum will increase the stepsize.

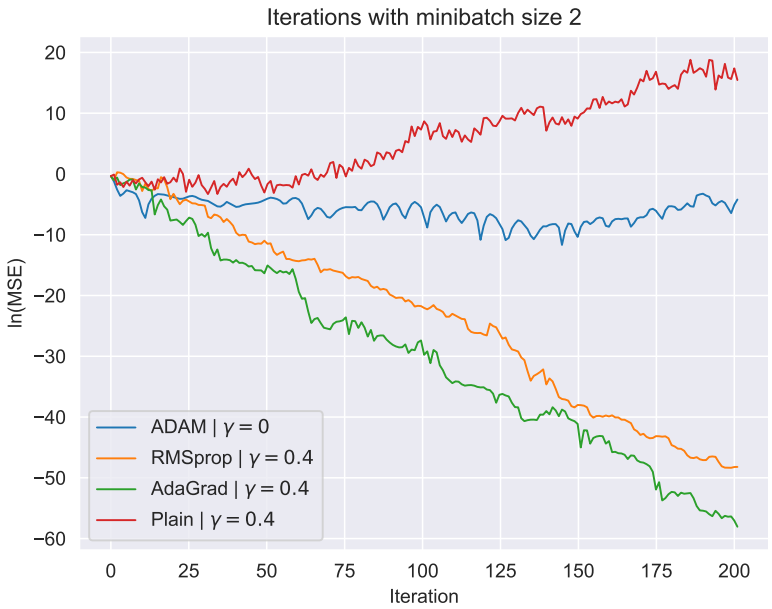


Figure 9: Stochastic gradient descent using plain, AdaGrad, RMSprop and ADAM as tuning methods: ln(MSE) as a function of iterations with minibatch size 2. The parameters utilized are shown in table 2 under run 4.

Figure 10: Stochastic gradient descent using plain, AdaGrad, RMSprop and ADAM as tuning methods: ln(MSE) as a function of iterations with minibatch size 5. The parameters utilized are shown in table 2 under run 4.

Figure 11: Stochastic gradient descent using plain, AdaGrad, RMSprop and ADAM as tuning methods: ln(MSE) as a function of iterations with minibatch size 10. The parameters utilized are shown in table 2 under run 4.

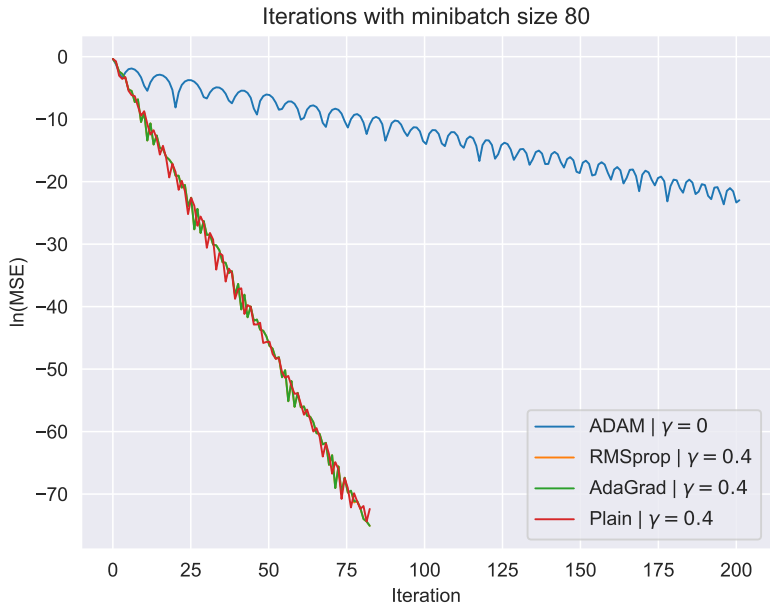


Figure 12: Stochastic gradient descent using plain, AdaGrad, RMSprop and ADAM as tuning methods: ln(MSE) as a function of iterations with minibatch size 80. The parameters utilized are shown in table 2 under run 4.

Figure 9-12 shows the natural logarithm of MSE as a function of iterations for minibatch size 2, 5, 10 and 80 respectively. Each figure shows stochastic gradient descent with momentum parameter γ for no tuning method (plain), tuning method AdaGrad, RMSprop and ADAM utilized to fit the polynomial in equation 3. As the training data size is 80, minibatch size 80 corresponds to non-stochastic gradient descent. The reason the plots dissapear in the vicinity of $y = -70$ is that the MSE traverses the lowest positive number possible in python such that they are set to zero wich is undefined in the logarithm function. We observe faster convergence with larger minibatch sizes for all the methods. No tuning method, AdaGrad and RMSprop have similar performance for minibatch sizes 80 and 10. For smaller sizes, we observe that AdaGrad performs best closely followed by RMSprop, while Plain starts deviating and is its MSE is exploding for minibatch size 2. ADAM performs worst (except compared to the exploding Plain) and is converging quite a lot slower, allthough the difference seems to lessen with smaller minibatch size. Smaller minibatch size introduces the probability of more variance in the gradient which suggests ADAM might be more applicable to more complex cost functions.

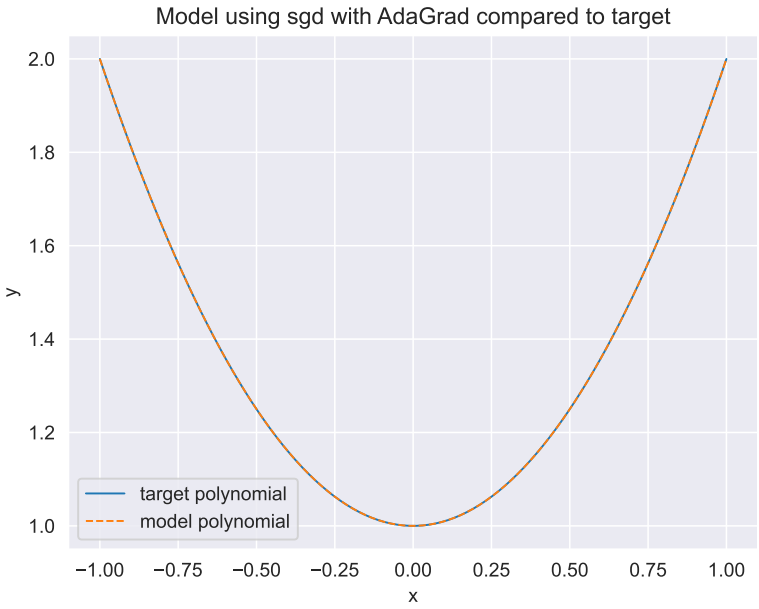


Figure 13: Resulting polynomial after stochastic gradient descent with optimizer AdaGrad compared to the target function. The parameters utilized are shown in table 3

Figure 13 shows the resulting polynomial from stochastic gradient descent with tuning method AdaGrad compared to the target polynomial. As expected from the MSE analysis, the prediction is perfect at eye level. A perfect fit could also easily be achived with for example ordinary least squares, allthough this is computationally heavy for huge datasets where the option of using minibatches in stochastic gradient descent would be beneficial.

3.2 Neural Network Regression

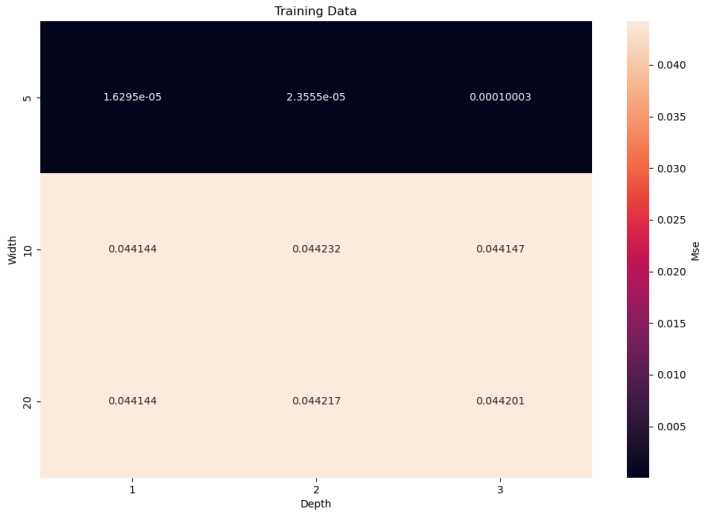


Figure 14: MSE score predicted on the polynomial training data with respect different number of hidden nodes (Width) and layers (Depth). Parameters used is listed in table 6 under Run 6.

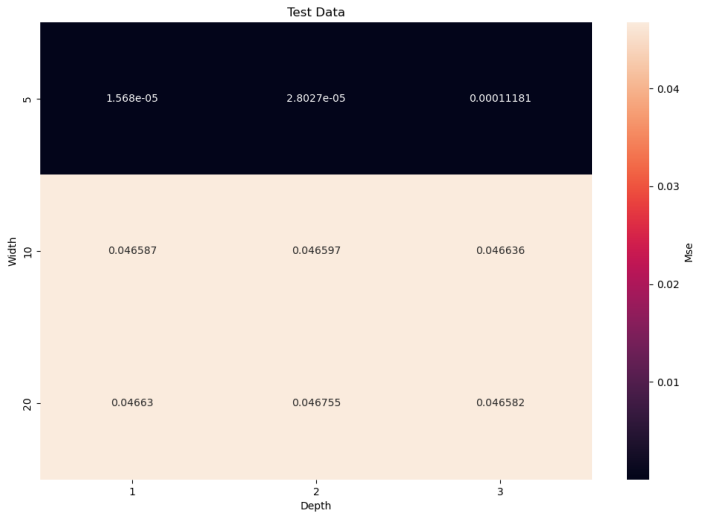
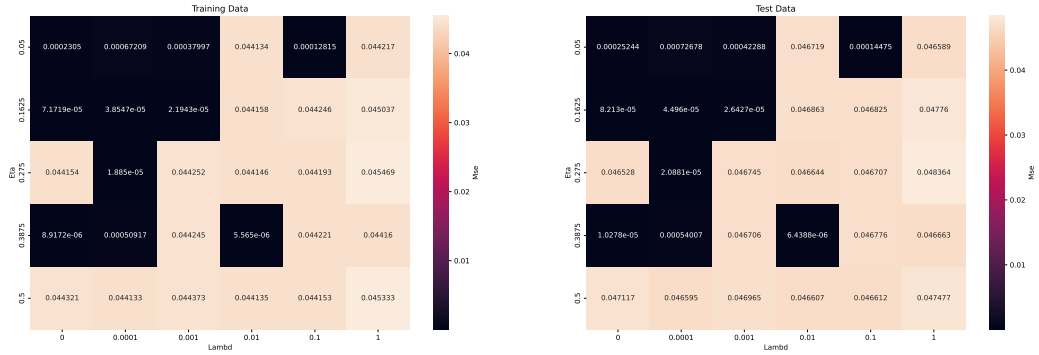


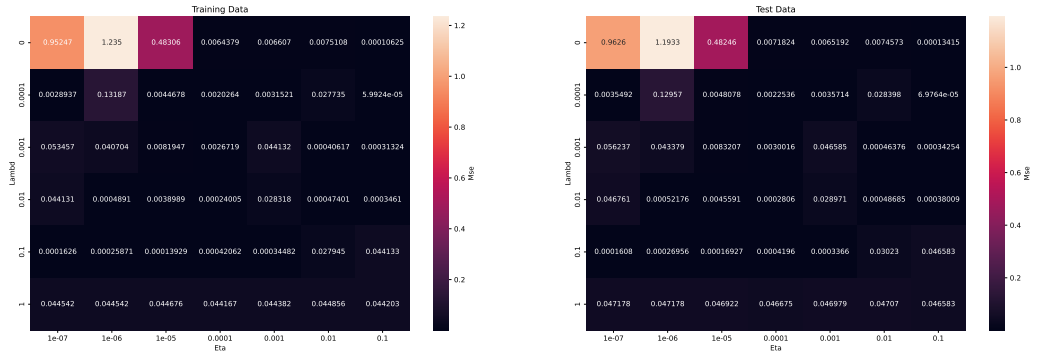
Figure 15: Neural network with activation function Sigmoid: MSE score from prediction on the polynomial data with respect different number of hidden nodes (Width) and layers (Depth). Parameters used are listed in table 6 under Run 6.

Figure 14 and figure 15 shows respectively training and test MSE for different numbers of hidden layers, and different numbers of nodes in the hidden layer(s). The best MSE was obtained with 1 hidden layer consisting of 5 nodes. With width 10 and 20, the network gets stuck on training MSE around 0.044, such that less width should be explored when searching for optimal structure.



(a) Train MSE (b) Test MSE

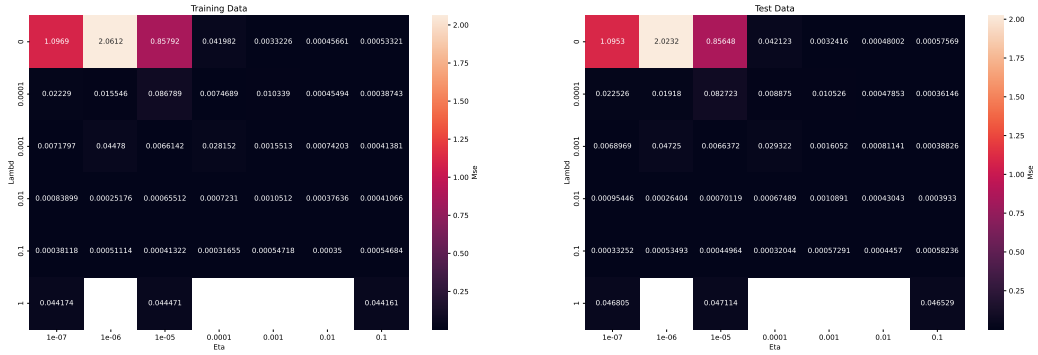
Figure 16: Neural network with activation function Sigmoid: MSE as a function of η and λ. The parameters utilized are shown in table 4 under run 1.



(a) Train MSE (b) Test MSE

Figure 17: Neural network with activation function relu: MSE as a function of η and λ. The parameters utilized are shown in table 4 under run 2.

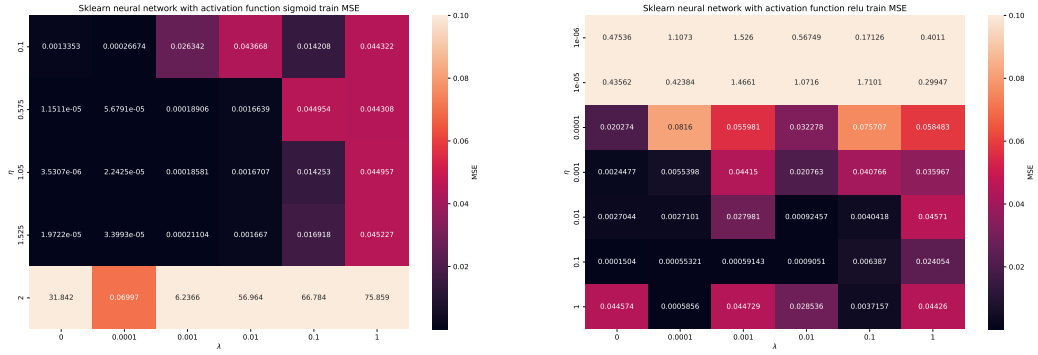
Figure 16-18 shows training (a) and test (b) MSE for different L2-regularzation parameter λ (lambd) and differnt learning rates η (eta) using activation function Sigmoid, ReLU and leaky ReLU on the hidden layer. The same scheme are shown in figure 19 with activation functions Sigmoid (a) and ReLU (b) using `sklearn MLPRegressor`. For sigmoid, we observe that the MSE are more likely to get stuck for the larger λ and η values. It is difficult to determine the optimal parameters as no trend is observable other than η should be larger than 0.0500 and less than 0.5000, while λ should be less than 0.0100. The best score was achived with $\eta = 0.0100$ and $\lambda = 0.3875$, but every neighboring values are poor suggesting this might just be a lucky run



(a) Train MSE

(b) Test MSE

Figure 18: Neural network with activation function leaky relu: MSE as a function of η and λ. The parameters utilized are shown in table 5 under run 3.



(a) Sigmoid MSE

(b) Relu MSE

Figure 19: SkLearn neural network with activation function sigmoid and relu: Train MSE as a function of η and λ. The parameters utilized are shown in table 5 under run 4 and in table 6 under run 5 for Sigmoid and ReLU respectively.

from the model's randomness. More runs with different random seeds should be analyzed to give more certain answers.

With ReLU as activation function,the best score was obtained with $\eta = 0.1000$ and $\lambda = 0.0001$. There seems to be a trend where for smaller η', larger λ is beneficial. The same trend is visible for leaky ReLU where the best score was for $\eta = 10^{-6}$ and $\lambda = 0.0100$.

Using activation function sigmoid for the hidden layer gave the best score with $MSE = 6.4388 \cdot 10^{-6}$ for the polynomial test data. Then the next best test score was obtained using ReLU with $MSE = 6.9764 \cdot 10^{-5}$, but with less of chance that the network gets stuck at training MSE around 0.044. Leaky ReLU performed worst with a score of $MSE = 2.6404 \cdot 10^{-4}$, but it also had the least probability of getting stuck. Thus every activation function had its advantages and disadvantages. Comparing training and test MSE, we observe that the trained network does not seem to suffer from overfitting.

Our results seems reasonable by comparing with the results form `sklearn's` neural network, allthough for different learning rates and `sklearn's` model does not seem to benefit from L2-regularzation. This is because our implementation of the L2-regularization parameter was wrong as discussed in section 3.5.

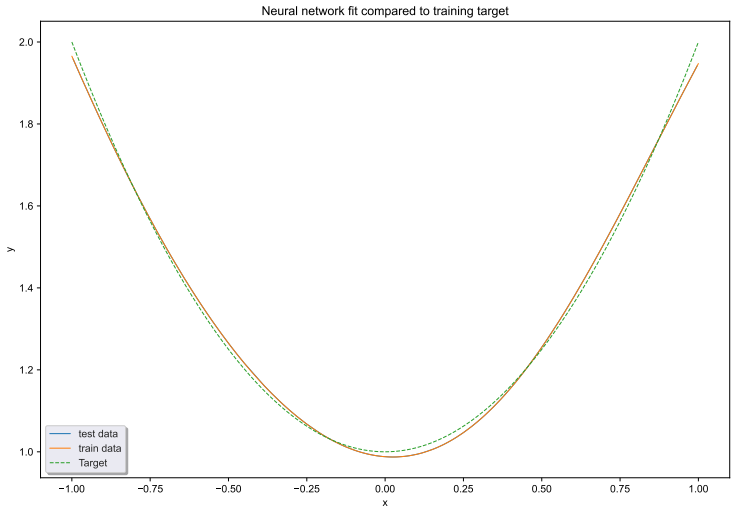


Figure 20: Resulting output after training the neural network with activation function sigmoid compared to the training target. The parameters utilized are shown in table 6 under run 6.

Figure 20 shows the resulting polynomial fit after training the neural network with activation function Sigmoid. Comparing our model to the target, we observe pretty similar shapes although we can clearly see some deviation. The polynomial fit obtain from gradient descent were much better (which is also obvious comparing MSE), but we have to keep in mind that gradient descent used the second order polynomial design matrix such that it was given the information of which order polynomial to optimize (which were the same order as the target polynomial) , while the neural network did not have this information.

3.3 Classification

In figure 21 the accuracy score is plotted for different number of mini-batches. We observe that the convergence rate increases with increasing number of mini-batches. This trend is expected. The total amount of iterations/training is higher for a larger number for mini batches. Another reason is that we are less likely to get stuck in global minima, since the multiple different samples is used to calculate gradient of the cost function. We also observe that the accuracy score tend to increase both for the training and test data with increasing number of mini-batches. We chose 20 number of mini-batches as our best value in the next. However, 15 number of mini-batches seems to give similar performance.

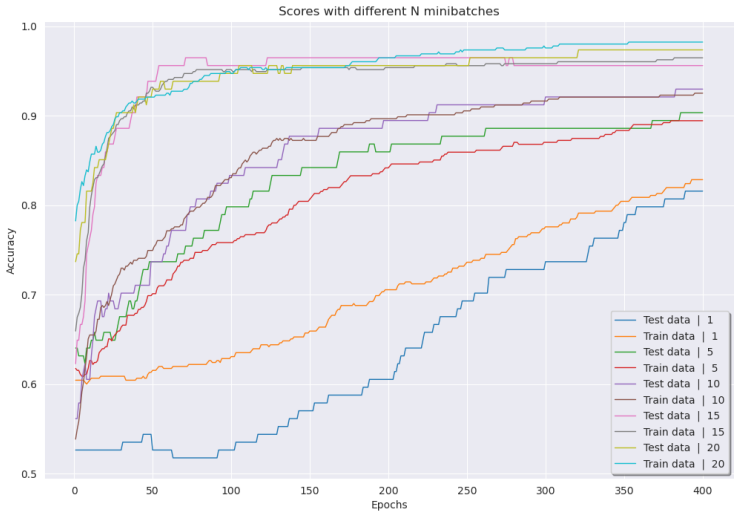


Figure 21: Accuracy score for train and test data plottet with respect to epochs for Run 1 (see table 7). The number on the labels correspond to number of mini-batches.

In figure 22 and 23 is accuracy score for run 2 plotted for different values of regularization parameter and learning rate for the training and test data respectively. For our test data the best Accuracy of 0.9825 was gotten with $\lambda = 10^{-4}$, $\eta = 0.1$ and $\lambda = 0.01$, $\eta = 0.01$. Generally we observe that the accuracy tend to increase with increasing learning rate. This make sense since the model is learning faster. High learning rates often leads exploding loss scores. This is not observed in this case. By choosing the largest learning rate, we can maximize the accuracy and minimize CPU cycles.

We observe an increasing trend in accuracy with respect to increasing λ values (Lambda in figure). This is more significant for the smaller values of learning rate. To further study this trend we plotted the accuracy score for different λ values with respect to epochs in figure 24. We clearly see in increased convergence rate for higher values of lambda. Moreover, the accuracy both for the train and test data is higher for larger values of λ. For further fine-tuning of parameters we will set $\lambda = 0$, to better observe how the different parameters affect the score with respect to epochs.

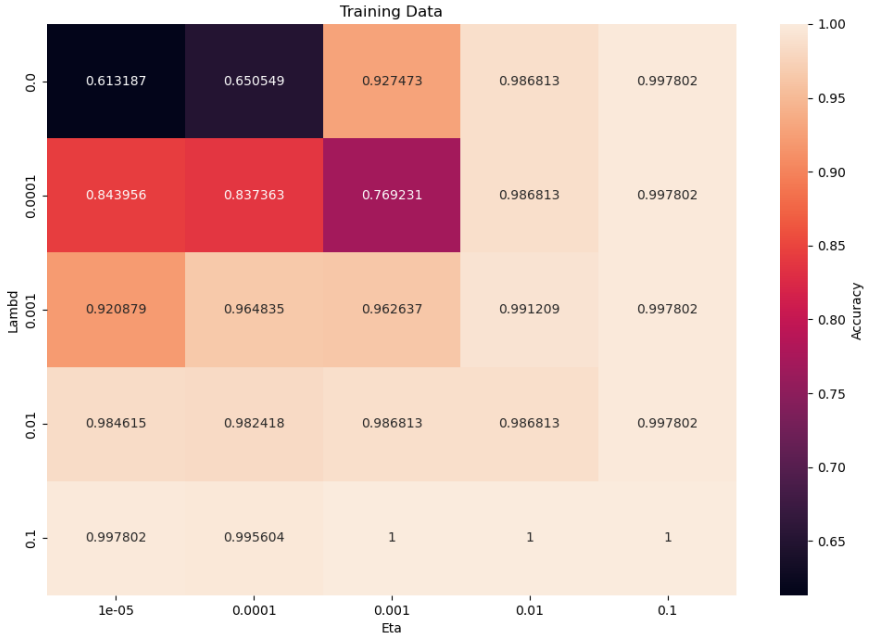


Figure 22: Heatmap of accuracy score for training dta after 200 epochs with respect to different learning rates (Eta) and different L2-regularzation parameters (Lambd). Spesific parameters used in run 2 is listed in table 7

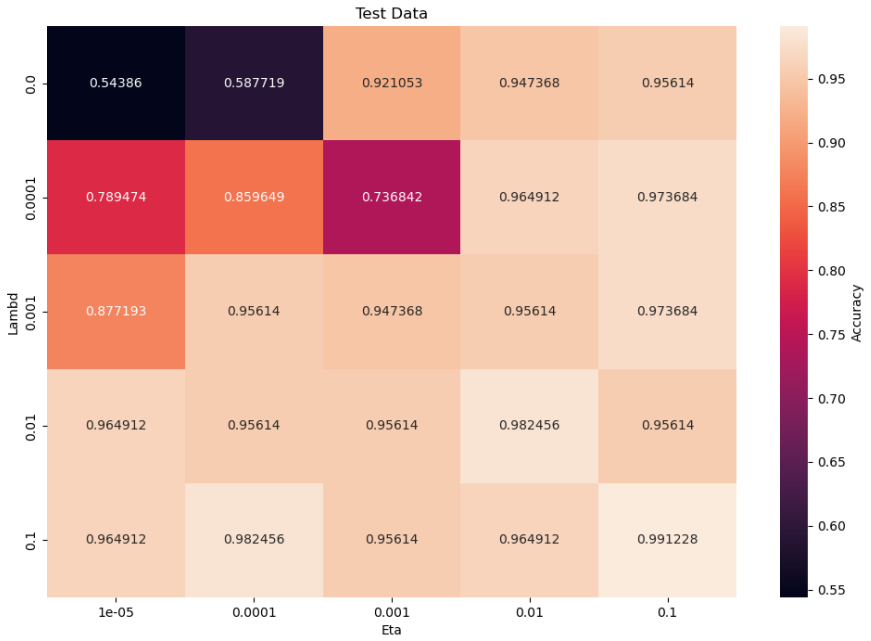


Figure 23: Heatmap of accuracy score for test dta after 200 epochs with respect to different learning rates (Eta) and differn L2-regularization parameters (Lambd). Specific parameters used in run 2 is listed in table 7

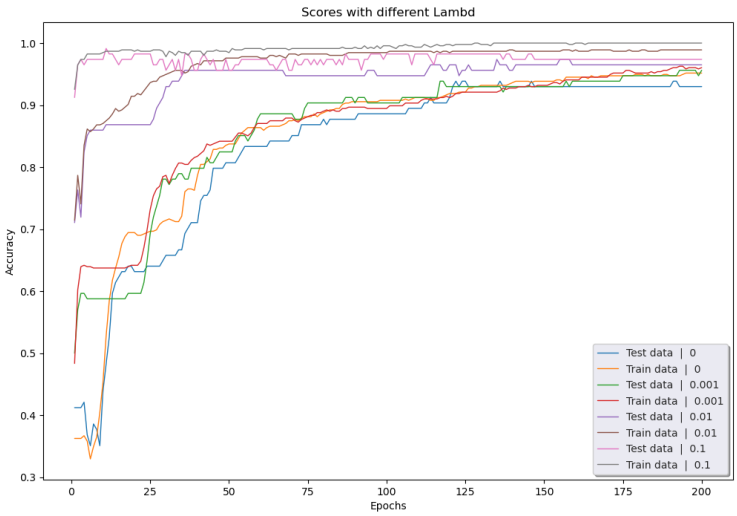


Figure 24: Plot of accuracy score for training and test data with respect to epochs. Each line correspond to different values of λ as indicated by the number in the legend. A learning rate of 0.001 was used.

In Run 3 different network architecture was tested with respect number of nodes (width) and number of hidden layers (depth). The heat map's for the train and test data is shown in figure 25 and 26. For our test data the best accuracy score was gotten with 2 hidden layers with 5 nodes in each, with an accuracy of 0.9912. There is no clear trend with respect to architecture. We will chose 2 hidden layers and 5 hidden nodes as our best parameters. However the variations with respect to depth and width, may bet attributed to other random phenomena such as mini-batch selection. There is no reason to chose a more complex model since this will increase the computational cost of the network.

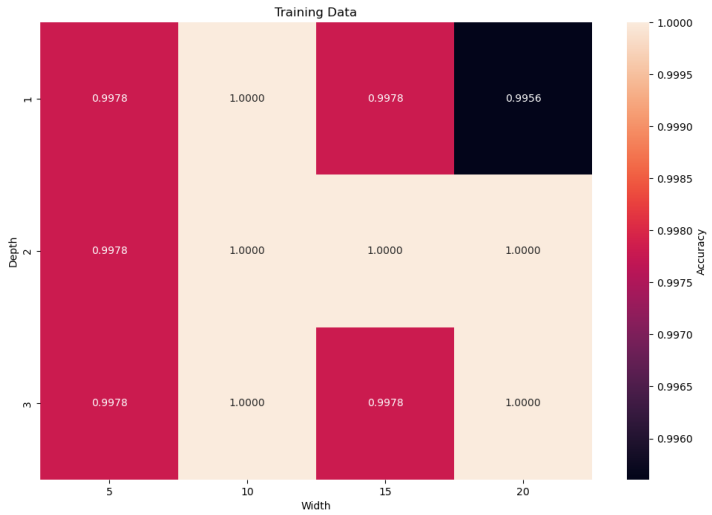


Figure 25: Accuracy score for train data with respect to different N number of hidden layers (depth) and number of hidden nodes in each hidden layer (width)

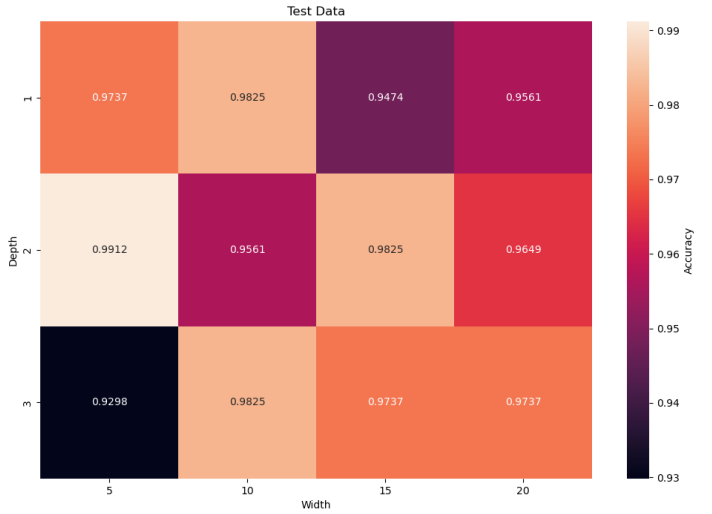


Figure 26: Accuracy score for test data with respect to different N number of hidden layers (depth) and number of hidden nodes in each hidden layer (width)

In Run 4 we tested different learning rate tuning methods. In figure 27 the accuracy score is plotted with respect to epochs for manual (none), adagrad, adman and the rms prop tuning method. The learning rate parameter was set to 0.001. We observe that RMS Prop and ADAM

converges much faster than Adagrad and without any tuning. The poor performance of Adgrad is probably due the low value initial value for η. Both ADAM and RMS-prop performance well on both the training and test data with a high accuracy and convergence after around 50 epochs.

In figure 28 the same lines are plotted but with momentum, $\gamma = 0.9$. This causes a significant improvement both the accuracy and convergence rate with Adagrad and plain SGD. Adagrad gives better prediction on the test data than both RMS prop and ADAM. The added momentum to ADAM causes more fluctuations with respect to epochs than without momentum.

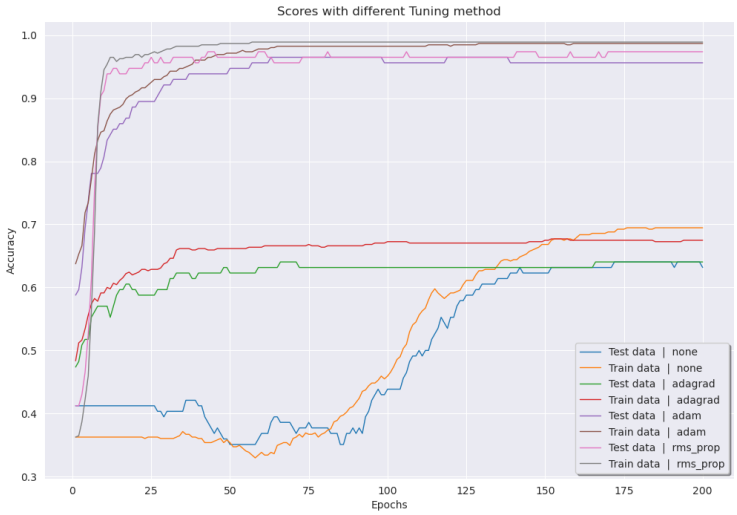


Figure 27: Accuracy score for test and data with different tuning methods plotted as function of epochs. Momentum (γ) is set to 0.

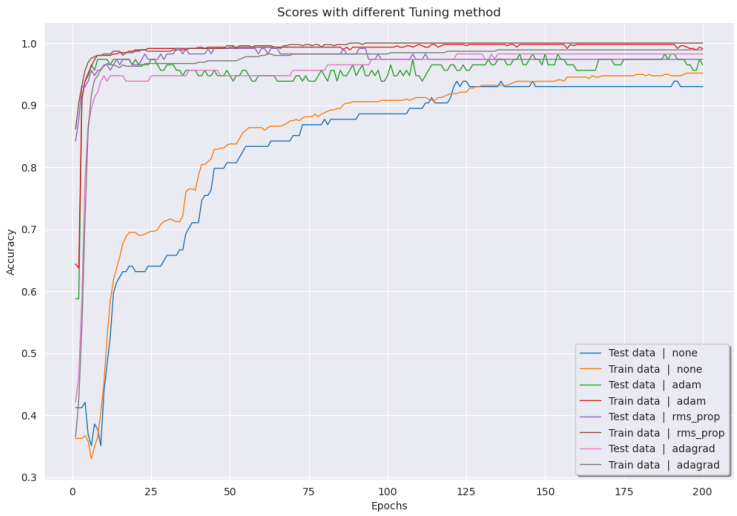


Figure 28: Accuracy score for test and data with different tuning methods plotted as function of epochs. Momentum (γ) is set to 0.9.

In run 5 we tested different activation functions on the hidden. From run 4 we observed that both tuning method and added momentum improved the results significantly. To understand better the different activation functions affect the results we used a momentum parameter $\gamma = 0$ with constant learning rate, $\eta = 0.001$. In figure 29 the accuracy is plotted for different activation functions in the hidden layers. Both RELU and leaky RELU outperforms the sigmoid activation function. RELU and Leaky RELU performs similar on the training data, but RELU gives the best overall accuracy on the test data.

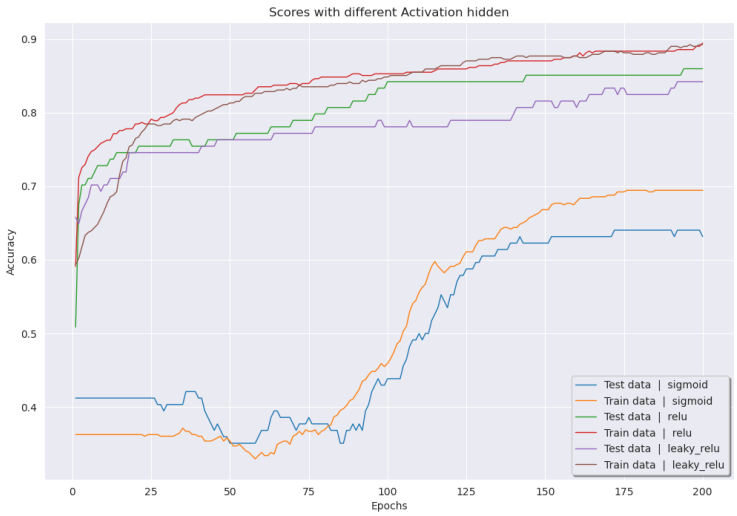


Figure 29: Accuracy score for train and test data with respect to epochs. Different activation functions on the hidden layers was used as indicated in the label

In run 6 our model reached an accuracy of 0.9825 after 165 epochs (see figure 30). The result was stable with increasing number of iterations. We tried to select the optimal values for hyper-parameters on the basis of the previous experimentation. However, we did use the sigmoid function on the hidden layers and not REL or Leaky REL, as they performed worse. The parameters used in run 6 is listed in table 8.

The overall best accuracy score on the test data seen in this experiment was observed in Run 3, where we experimented with different network architecture. Then we got a score of 0.9912. We suspected that this was due to luck and random events in our mini-batch selection method. Overall our model seemed to perform really well on a Classification problem with the Wisconsin Breast Cancer data. We compared our results from run 6 with Kares (python API for Tensor Flow), With the exact same parameters, except $\eta = 1$ and $\gamma = 0$. We then got a test accuracy prediction of 0.9825, that matches our finding.

We originally tried to find the optimal parameters of the Network, by improving a few parameters at the time. However, our best parameter choices created a model with excellent predations after only a few epochs. Therefore, we decided to remove some of the parameters to better be able to study how each individual parameter affected the model. However, there is not certain that all the parameters will play nicely together. Thus, a more methodical approach should have been applied in order to see how different parameters affect each other.

From the literature we found that models created with an artificial neural network and random forest produces test accuracy's of 92.44 and 95.64 respectively [3]. Our model does a better job on the test data. We don't expect our simple FINN code to be superior to others. However, we do not suspect any error in implementation since our benchmarking gave the same result.

There is one serious drawback with our approach. We did not use any re-sampling technique when we trained our model. Our high accuracy score is probably due to our specific train-data subset selection. Bootstrap re-sampling would have been a good choice to better control the randomness

in our model. Then we would have got better insight model variation due to parameter selection, and not random events. However, this approach would have been too expensive for our laptops. Therefore, a re-sampling technique such as k-fold cross validation should have been applied.

To better control the randomness in our model we should have predefined a mini batch sequence, and also used the same initial weights when this was possible. Then we could have more easily seen if the variations in our predictions was due to random events or not. We did not study how the weights and bias initialization affected the results. Insight in how random events affected our model could have been obtained by applying Bootstrap re-sampling.

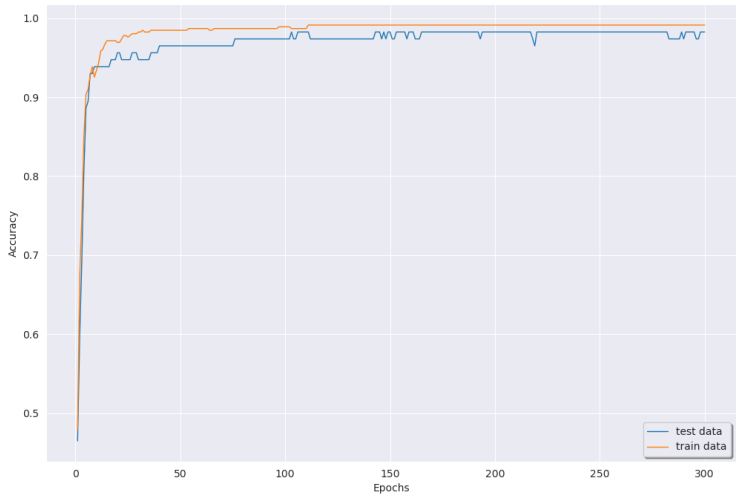


Figure 30: Run 6: Plot of accuracy score for train and test data, where we tried to find the overall best hyperparameter combination

After all these calculations there is only a few obvious choices for hyperparameters. A larger number of mini-batches did increase the accuracy, convergence rate and reduces required CPU cycles. We did not observe any over-fitting in our model, but a larger value of lambda did increase the convergence rate. There was no clear trend in accuracy score with respect to network architecture, but a depth of 2 and width of 5 gave the best prediction on the test data. Tuning methods such as Adam and RMS prop, significantly improved the convergence rate. Adagrad and manual tuning was inferior to rms prop and adam, but significantly improved when momentum was added. Adagrad then performed the best on the test data.

3.4 Logistic regression

In figure the accuracy score obtained with logistic regression is plotted for different number of mini-batches. Again a small mini-batch size seems to be the obvious choice as discussed for our neural network. In figure 32 and 33 the accuracy on the Cancer Data is plotted with different learning rates and L2-regularization parameters for the train and test data respectively. The best score for both the training and test data was obtained with a learning rate $\eta = 0.001$

without any regularization. Our best prediction on the test data got an accuracy of 0.9035. Our neural network code does contain 11 times as many weights. Thus, we expect it to be able to produce more complex patterns. With did benchmark our result against scikit-learn's Logistic regression method. We then got a test accuracy of 0.9649. We was not able to improve our logistic regression algorithm to produce this result. We do not suspect that our implementation is wrong. Scikit-learn's may contain further optimization. Hence, the better accuracy score. However, our Neural Network did produce a better accuracy score than Scikit-learns logistic regression method.

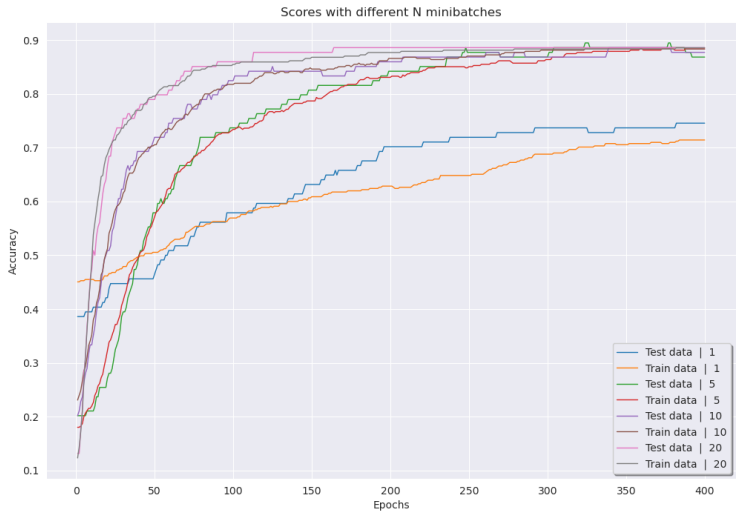


Figure 31: fig:Accuracy score obtained with logistic regression on the Wisconsin Breast Cacner data. Predication on training and test data is plotted for different number of mini-batches as indicated in the label.

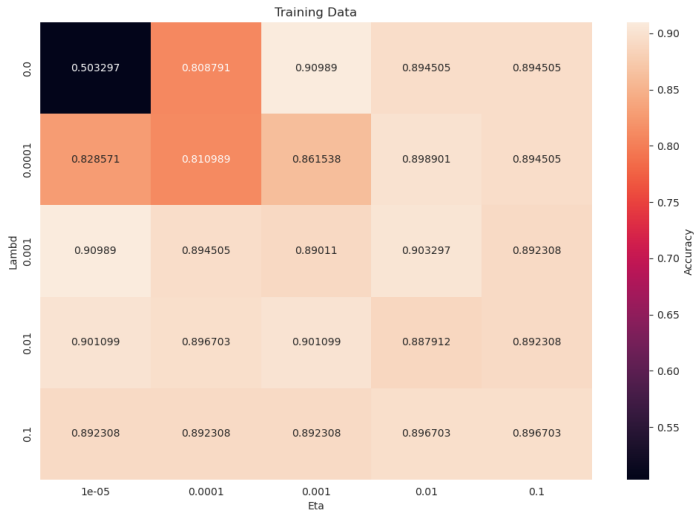


Figure 32: Accuracy score on Wisconsin Breast Cancer training data, with respect to different learning rates and L2 regularization parameters.

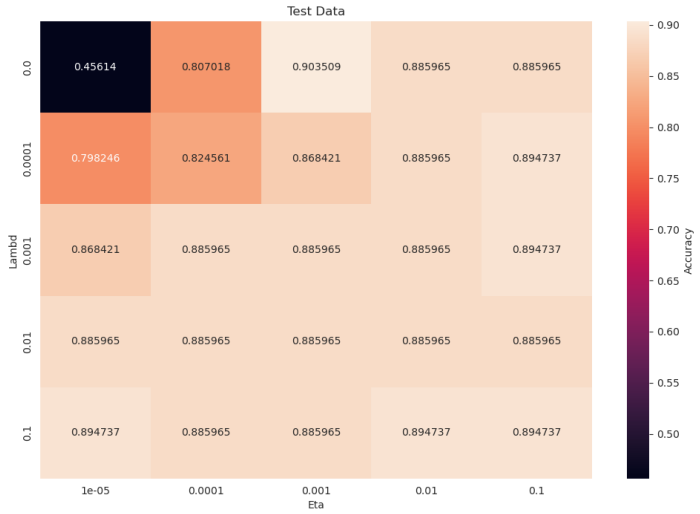


Figure 33: Accuracy score on Wisconsin Breast Cancer test data, with respect to different learning rates and L2 regularization parameters.

3.5 Error in L2-regularzation

After our report was completed and ready for delivery we found an error in how the L2 regularization parameter was calculated. Instead of multiplying the regularization parameter with our weights from our previous iteration, the regularization parameters was multiplied with the gradient of the weights, effectively increasing the learning rate. Our analysis of the L2 regularization parameter is therefore not valid.

4 Conclusion

We fit a second order polynomial using a variety of GD methods. Specifically we looked at regular GD, momentum GD, plain SGD, momentum SGD and also using AdaGrad, Adam and RMSProp for tuning the learning rate. We found that for a momentum of 0.8 decreasing the learning rate always led to a decrease in the MSE, but for momentum values of around 0.4 a higher learning rate of about 0.6-0.8 would produce better results. In general the lower the momentum was, the higher we could have our learning rates. But this could be due to the simplicity of the dataset and should be explored further in future work. The Adam method did not benefit from adding a momentum, which may be explained by it already having a first and second moment.

We also found that adding a L2-regularization gave worse MSE.

For increasing batch sizes of the stochastic methods we found that all methods converged to low MSE values faster. ADAM preformed quite a lot worse than the other methods. This could be because Adam is better at noisy datasets, and we tried to fit a second order polynomial without any noise. This should be explored in future work, using different datasets.

Then we tried to fit a neural network to the same polynomial. We used a dense neural network with one hidden layer and 5 neurons, with sigmoid, ReLU and leaky ReLU as activation functions for the hidden layer. Each activation function had its advantages and disadvantages with Sigmoid obtaining the best MSE score, but being most probable to get stuck on poor MSE scores, while the opposite was true for leaky ReLU, and was in the midle with regards to both properties. The L2-parameter was discovered to be implemented in a wrong way, such that it effectively changed the learning rate. For future work we would try implementing a sparse neural network to decrease the number of computations required for a forward and backward pass of the network.

The resulting polynomial fit after 2000 iterations from our neural network were pretty similar to the target, allthough deviations were clearly visible. The neural network did a lot worse than gradient descent, but was able to give a good fit without using a design matrix which were used by gradient descent.

In our classification problem with our FFNN on the Wisconsin Breast Cancer data, we did experiment with different hyperparameters. We found that the most influential parameters was mini batch size, momentum, and tuning method. 10-20 mini batches was superior compared with fewer mini batches. Added momentum significantly improved our SGD (constant learning rate) and Adagrad method. Other tuning method such RMS prop and Adam did not benefit from added momentum. All tree tuning methods did preform similar and should be tested on a NN classification problem. Adagrad benefits from a higher learning rate than RMS prop and Adam.

After experimentation width different parameters, we manged to obtain an accuracy score of 0.9825 on the test data. According to the literature this is better than to be expected. One significant weakness with our approach is that we did not utilize any re-sampling technique.

Thus, our high accuracy is probably a result of a lucky selection of training- and test-data.

Random events such a mini batch selection and weights selection should have been better controlled. Then, we could have more accurately determined if the variations in our scores was due to hyperparameter selection or random events. Re-sampling techniques could have been utilized to obtain this insight.

The same classification analysis obtained with logistic regression preformed worse, with an accuracy score of 0.9035. However, we manged to obtain an accuracy of 0.9649 with sci-kit learn (python package). Why the two results differ is not clear. Overall classification with a FFNN outperforms logistic regression.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[2] Morten H. J. *Week 41 Constructing a Neural Network code, Tensor flow and start Convolutional Neural Networks*. "`https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html`". 2022.

[3] Leena Vig. "Comparative Analysis of Different Classifiers for the Wisconsin Breast Cancer Dataset". In: *Open Access Library Journal* 1 (2014), pp. 1–7.