Classification and Regression, from linear and logistic to feed forward neural networks

Fuad Dadvar — Jonny Aarstad Igeh
Department of Physics, University of Oslo, Norway
`https://github.uio.no/Jonnyai/FYS-STK4155/tree/main/project%202`

November 20, 2022

Abstract

In this report, we've studied Feed-Forward Neural Networks for both regression analysis and binary classification. We build intuition of the optimization of the parameters in the neural networks by studying gradient descent methods of a linear regression of a simple 1D polynomial evaluated on data points in the range [0,1] using either OLS or Ridge to define our cost function. We find the SGD method the most efficient, and implement this in our FFNN.

We build our FFNN for regression and test this for various adjustments of the hyper-parameters. As expected, we find the network to be highly sensitive to our choices of both initializations of the weights, and any and all of the hyper-parameters. Testing our network on a simple 1D polynomial using 1000 datapoints in [-3,3], and applying min-max scaling on the data, we find that using 5 hidden layers with 20 neurons using Leaky ReLu gives an MSE of 0.0004 and R2 score of 0.9900, and we find that ReLu outperforms the sigmoid function greatly.

Lastly, we use our FFNN to classify breast cancer cases using the Wisconsin Breast Cancer ML dataset and compare this with our own Logistic Regressor. We find that our neural network, with optimal parameters $\lambda = \eta = 0.01$, using 10 neurons in 2 hidden layers with sigmoid activation, gives an accuracy score of 0.96, and outperforms the 0.94 of the logistic regressor.

Contents

1 Introduction **4**

2 Method and theory **5**

2.1 Preprocessing and scaling of data . . . 5
2.2 Gradient Descent . . . 6
2.2.1 Plain Gradient Descent . . . 7
2.2.2 Gradient Descent with momentum . . . 8
2.2.3 Stochastic Gradient Descent . . . 8
2.2.4 Stochastic Gradient Descent with momentum . . . 9
2.2.5 Adaptive Learning rates . . . 9
2.3 Neural Network . . . 11
2.3.1 Feed forward . . . 12
2.3.2 Activation functions . . . 13
2.3.3 Cost function . . . 14
2.3.4 Back propagation . . . 15
2.3.5 Running the network . . . 15
2.3.6 Optimizing the network: Hyper-parameters . . . 16
2.4 Logistic regression . . . 17

3 Results **19**

3.1 Gradient Descent . . . 19
3.1.1 Optimal hyper parameters: comparison OLS vs RIDGE . 19
3.1.2 Rate of convergence: Gradient Descent methods . . . 20
3.1.3 Rate of convergence: Adaptive learning methods using SGD 20
3.2 Feed-Forward Neural Network: Regression . . . 21
3.2.1 Comparison between activation functions . . . 22
3.2.2 Adjusting hyper-parameters: Sensitivity of FFNN . . . 22
3.3 Feed-Forward Neural Network: Classification . . . 24
3.4 Logistic regression: Classification . . . 26

4 Discussion **27**

4.1 Pre-processing and scaling of data . . . 27
4.2 Gradient Descent methods . . . 27
4.2.1 Optimal hyperparameters: comparison OLS vs RIDGE . 27
4.2.2 Convergence: Gradient descent methods . . . 28
4.2.3 Convergence: Adaptive learning rate methods . . . 28
4.3 Neural Network . . . 29
4.3.1 Regression Analysis . . . 29
4.3.2 Classifier . . . 29
4.4 Logistic Regression . . . 30

5 Conclusion **31**

A Appendix **32**
A.1 MSE change for given λ and η in a heatmap plot. 32
A.2 Code: Gradient descent 33
A.2.1 Plain gradient descent 33
A.2.2 Gradient descent with momentum 33
A.2.3 Stochastic gradient descent 34
A.2.4 Stochastic gradient descent with momentum 34
A.3 Code: Adaptive learning method 35
A.4 Code: Forward propagation 36
A.5 Code: Sigmoid and cost -function, and gradient cost function . . 36

1 Introduction

Neural networks are an invaluable tool in the real world, it is used in every-day technologies, such as medical diagnosis, image and speech recognition, and more. Thus the need arises for bright minds that can develop, and maintain these technologies.

In this report we are looking to build intuition, and understanding, of how these neural networks function - and how the networks behave when experiencing changes in it's hyper-parameters. We will work with a multi-layer-perceptron network (MLP) for both regression analysis and binary classification. Firstly, we will look at various gradient descent methods applied to linear regression (which is the simplest form of machine learning) before moving on to building our first Feed-Forward Neural Network (FFNN), and extending this to function well for both regression and classification problems, before making our own logistic regression classifier.

The gradient descent methods we will study are standard and stochastic, with and without momentum. We will use these when optimizing the neural network built for regression and classification. Lastly, we will build a logistic regression algorithm to compare with our FFNN classifier.

Necessary theory and introduction of the various methods used to build are models, as well as how they've been implemented numerically, are presented neatly in the theory and method section. The results from testing our programs are shown in the results section, and discussed thoroughly in the discussion section. Lastly, we present a critical evaluation of the various algorithms we've delved into.

2 Method and theory

As stated in the introduction, we are going to study both regression problems and classification problems. When we study regression in gradient descent, we will use the following 1D polynomial to produce data:

$$f(x) = 2 + 4x - x^2,$$

on a data set $x \in [0, 1]$ for a varying amount of data points, with added noise (from a normal distribution with r$\sigma^2 = 0.5$), unless otherwise specified.

When working with the FFNN regression we are using the polynomial:

$$f(x) = 2x + 4x^2 - 2x^3,$$

evaluated on $x \in [-3, 3]$ for 1000 datapoints, with added noise $\sigma^2 = 0.02$. The data sets are readily available to be checked in our code on the GitHub repository[1].

When we study the classification problems we are going to use the Wisconsin Breast Cancer ML dataset (WBC), which is a hugely popular dataset to build, test and evaluate classification models. This is supplied through sklearn.datasets[1]. The seed used for all the randomizations is np.random.seed(0).

2.1 Preprocessing and scaling of data

In most of these tasks, unless deemed needless, we will scale our data using ScikitLearn - and train test split our datasets when training the various models. When we work with the WBC dataset we will use Standard scaling using sklearn.StandardScaler[2], and when we work with the polynomial we use min max scaling using sklearn.MixMaxScaler[3], and we use sklearn.train_test_split[4] for the splitting of our data set. In the discussion section we will present a critical evaluation of our choices.

For more in-depth theory behind scaling and splitting of data, and the usage of this, is based on the theory presented in our first report, found in [2].

[1]`https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`

[2]`https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`

[3]`https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`

[4]`https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`

2.2 Gradient Descent

The method of gradient descent (GD) origins from the fact that a function $F(\boldsymbol{x})$, where $\boldsymbol{x} \equiv \boldsymbol{(x_1, \ldots, x_n)}$, decreases the fastest when one goes from $\boldsymbol{x}$ in the direction of the negative gradient of said function, $-\nabla F(\boldsymbol{x})$. This gives us an opportunity to localize a minima, $\boldsymbol{x}_{min}$, for the function by the iterative scheme as follows:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma_k \nabla F(\boldsymbol{x}_k), \tag{1}$$

with $\gamma_k > 0$. This parameter is called the *learning rate*, and can be thought of as the step-size in each iterative "step". If we size this sufficiently small (such that we don't "jump" across the minima), we can guarantee that $F(\boldsymbol{x}_{k+1}) \leq F(\boldsymbol{x}_k)$ and naturally we are moving towards a local minima. (We use local, since this method do not guarantee that the minima found is truly the global minima).

We can use this method to optimize *any* function $F(\boldsymbol{x})$, and if we choose the function to be the cost-function that evaluates the performance of some mathematical model - we can use this to optimize said model. This is what we are going to do with our regression model, and our neural network. We will study four different methods for gradient descent, namely

- Plain Gradient Descent
- Momentum based Gradient Descent
- Stochastic Gradient Descent
- Momentum based Stochastic Gradient Descent

By initiating a given cost function $C(\boldsymbol{\beta})$ (depends on the functionality of our model) we can approach the minimum of this cost function, and thus improve our model, by calculating the gradient $\nabla_{\boldsymbol{\beta_i}} C$ with the respect to the unknown parameters $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_n)$. With this, we can now optimize the parameters by finding the minima of the cost function, using the gradient descent method as follows:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta \nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_k), \tag{2}$$

where η denotes the learning rate. Knowing when to stop the iterative process is up to choice, and the "optimal" choice varies from method to method. Some choices are to count the number of iterations, and stop at some pre-set value (to prevent the process going on forever), or find some suitable stopping critera that stops the process (usually that the gradient becomes sufficiently small, meaning we are near a local minima).

2.2.1 Plain Gradient Descent

The way of plain gradient descent(PlainGD) is shown in eq. (1). In this report, we start by doing linear regression - and then it can be shown that the cost function (using Ridge) becomes:

$$\nabla_\beta C(\beta) = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) + 2\lambda\beta, \tag{3}$$

where λ is the ridge parameter ($\lambda = 0$ yields standard OLS) and we simply insert this expression for the gradient of the cost function into eq. (2).
This choice of stopping criteria (that the gradient will be smaller than some threshold value) assumes that as we near the local minima, the curvature flattens (as we know, the gradient is zero *at* the minima) and thus we can assume to be close to a local minima. The initial values for β are plucked randomly from a normal distribution, and this is just our choice of how we initialize - if you have previous knowledge of your data and you can make a better guess, this is ideal and will most likely shorten the time your model takes to converge to a local minima. (It may infact also increase the likelihood that you hit the global minima if you are sufficently smart in your choice of β_0).

2.2.2 Gradient Descent with momentum

We may improve upon our gradient descent scheme by introducing *momentum*. With this we mean that our scheme takes into account the size of the previous gradient, and thus the scheme often times will converge at a quicker rate. The scheme then becomes:

$$\begin{aligned}\mathbf{v_t} &= \gamma \mathbf{v_{t-1}} + \eta \nabla_\beta C(\beta_t),\\ \beta_{t+1} &= \beta_t - \mathbf{v_t},\end{aligned}$$

where $\gamma \in [0,1]$ is our momentum parameter, and this decides how much the previous gradient affects the ensuing gradient. With the addition of momentum, we now improve the gradient descent method such that it will move faster when the gradients are small (flat-like curvature) and slow down when the gradient are large (steep curvature). In theory, this should increase the rate at which the descent method converges. One can implement this in Python using the code in appendix section((2))

2.2.3 Stochastic Gradient Descent

As seen in regular gradient descent, we need to perform a derivative of the cost function with respect to every parameter β_i, as well as do matrix multiplication with the entire design matrix. When we are looking at problems with many parameters and many data points, this will be an incredible numerical load to calculate. This is where stochastic gradient descent makes it's entrance.

We divide the data set in m mini-batches, each containing M datapoints. If we now rewrite the cost function as a sum over all the n datapoints $\{\boldsymbol{x_i}\}_{i=1}^n$:

$$C(\beta) = \sum_{i=1}^{n} c_i(\boldsymbol{x_i}, \beta). \tag{4}$$

We see that this implies that the gradient can be computed as a sum over n-gradients:

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta).$$

Now we introduce stochasticity, and reduce the numerical load, by instead computing an approximation to the gradient using only the datapoints in a given (random) minibatch (denoted as B_k, where $k = 1, \dots, \frac{n}{m}$):

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\boldsymbol{x}_i \beta) \longrightarrow \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta).$$

Therefore a Stochastic gradient descent step will have the form:

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i \beta), \tag{5}$$

where k is chosen randomly and with equal probability from $[1, n/M]$. We call an iteration over the number of minibatches n/M an epoch. Code implementation is shown in appendix section((3)).

2.2.4 Stochastic Gradient Descent with momentum

Similar to the plain gradient descent case, we can improve our stochastic gradient descents rate of convergence by introducing momentum. This is done in Python which is shown in appendix section((4)).

2.2.5 Adaptive Learning rates

Up till this point, we've assumed the use of constant learning rate η - however, this is not necessarily a good choice. Like when we added momentum, changing the learning rate as we move in parameter space may also increase the rate of convergence for our gradient descent method. In this report we will study these ways of adaptive learning rates

- Decaying learning rate

  This is the simplest one, and this makes the stepsize smaller with each iteration - eventually stropping the model. This is similar to having a set amount of iterations as a stopping criteria. The learning rate is then given by

$$\eta(t) = \frac{t_0}{t_1 - t}. \tag{6}$$

- RMSprop
  - With RMSprop we build on the momentum based GD and keep track of the second moment (denoted $\mathbf{s}_t$), and use this to scale the learning rate and the iterative scheme becomes

$$\begin{aligned} g_t &= \nabla_\beta C(\beta), \\ s_t &= \theta s_{t-1} + (1-\theta) g_t^2, \\ \beta_{t+1} &= \beta_t - \eta \frac{g_t}{\sqrt{s_t + \epsilon}}, \end{aligned}$$

    where θ (usually $= 0.9$) controls the effect that the second moment has on the GD step. ϵ is a small constant used to prevent division by zero.

- ADAM
  - The ADAM optimizer builds on momentum like RMSprop, by keeping track of the running average of both first- and second moment of the gradient. It is similar to RMSprop, and the scheme becomes

$$
\begin{aligned}
g_t &= \nabla_\beta C(\beta), \\
m_t &= \theta_1 m_{t-1} + (1-\theta_1) g_t, \\
s_t &= \theta_2 s_{t-1} + (1-\theta_2) g_t^2, \\
\hat{m}_t &= \frac{m_t}{1-\theta_1^t}, \\
\hat{s}_t &= \frac{s_t}{1-\beta_2^t}, \\
\beta_{t+1} &= \beta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{s}_t + \epsilon}},
\end{aligned}
$$

    where again, θ_1 and θ_2 control the effect that the first and second moment have, and are usually set to 0.9 and 0.99 respectively.η and ϵ are the same as in RMSprop.

- ADAGRAD
  - The ADAGRAD optimizer takes into account all the previous gradients when computing the ensuing gradients - and the scheme becomes

$$
\begin{aligned}
g_t &= \nabla_\beta C(\beta), \\
G_t &= \sum_{\tau}^{t} g_\tau g_\tau^T, \\
\beta_{t+1} &= \beta_t - \eta \frac{g_t}{\sqrt{G_t}}.
\end{aligned}
$$

    To preserve some numerical load, we usually only take the diagonal elements of G_t when implementing ADAGRAD numerically.

2.3 Neural Network

[3] Neural network is a machine learning techniques vaguely inspired by how network of neurons function in the brain. There are several types of neural networks, and in this project we will a study multi-layer perceptron (MLP). The MLP is characterized by one or more *hidden layers* between the input and the output layers. We have each hidden layer consist of *neurons*, also called *nodes*. The neurons in a layer l are not connected to one another, but each neuron in a layer l is connected to every neuron in the previous layer, $l-1$, which is characteristic for a MLP (called fully-connected). In figure 1 we can see this schematically, where every neuron in one layer is connected to every neuron in the previous layer by lines corresponding to some weight and bias.

This means that the input to a specific neuron i, is a weighted sum of the inputs from *all* the outputs, denoted x_j, from neurons j in the previous layer, i.e

$$z_i^l = \sum_j^k w_{ij}^l + b_i^l.$$

As seen in the figure, the input if fed through the network and processed by all the weights and biases, before it is ultimately passed through the output layer giving us the actual output from our neural network.

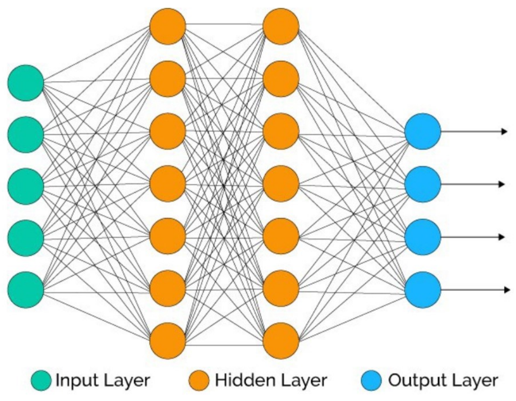


Figure 1: [5]Schematic diagram of a neural network with five inputs, two hidden layers of seven neurons and a output layer giving us four outputs

2.3.1 Feed forward

The input provided to our network is fed through every hidden layer before it is passed to the output layer. This process is often times called the ”Feed-forward stage” of the neural network. If we look at the output of a single neuron in our network, this can be expressed as follows

$$y = f\big(\sum_{i=1}^{n} w_i x_i\big).$$

Meaning, the output of a neuron in layer l is a weighted sum of the outputs from all the n neurons in the previous layer. This output is usually passed through an activation function (which we will discuss more in detail further into the report), and thus we can denote this as

$$y = a_j^l = f(z_j^l),$$

where a_j^l denotes output from node j in layer l after you pass the weighted sum z_j^l through the activation function f.

One usually adds a bias to prevent our system from stopping, which would happen if the weights w_i are zero, and with this we can make an expression for a difference equation that explains the mathematical processes in the deep layers of the network

$$a_j^l = f^l(z_j) = f^l\Big(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\Big). \tag{7}$$

We convert this to matrix notation and we will have:

$$\boldsymbol{a}^l = f(\boldsymbol{w}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l). \tag{8}$$

This is easily implemented in Python which is shown in appendix section((6)).

The way one initializes the networks various weights and biases is not written in stone, and there are many ways to do so. What we've done is to sample the weights from standard normal distribution using NumPys random.randn function, and scale the initialized weights according to the **He-et-al initialization**[3], and the biases are all initialized to a small value 0.01.

```
heuristic = np.sqrt(2 / self.dimension_out)                            # He-et-al heuristic
self.weights = np.random.randn(self.dimension_in, self.dimension_out) * heuristic
self.bias = np.zeros((1, self.dimension_out)) + 0.01
```

This is not the only way to do this initialization, and varying the method of initialization will change the performance and/or stability of the network. Our choice is backed on the paper by He et al, and our choice of activation function (namely, ReLu and Sigmoid) - since this scaled initialization will reduce the likelihood of exploding gradients, which we will study later.

2.3.2 Activation functions

The choice of activation function for the hidden layers is also something that must be judged from the desired field of application for the neural network. Some of the most common activation functions are the following:

- Sigmoid function
  - The sigmoid function will output values on the interval $[0, 1]$ and is expressed as
$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{9}$$
    and has the derivative
$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 + \sigma(x)). \tag{10}$$
- ReLu (Rectified linear unit)
  - The ReLu function outputs values on the interval $[0, \infty]$, and is expressed as
$$f(x) = \max(0, x) \tag{11}$$
    with a derivative
$$\frac{d}{dx}f(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases} \tag{12}$$
- Leaky ReLu
  - The leaky ReLu functions is similar to the regular ReLu, and can be expressed as
$$f(x) = \begin{cases} x, & x > 0 \\ 0.01x, & x < 0 \end{cases} \tag{13}$$
    and this fixes the issue with disappearing negative derivatives in the normal ReLu, by manipulating the derivative as follows
$$\frac{d}{dx}f(x) = \begin{cases} 1, & x > 0 \\ 0.01, & x < 0 \end{cases} \tag{14}$$

The sigmoid function is most commonly used when building neural networks for classification problems, while the ReLu functions are popular in regression neural networks.

2.3.3 Cost function

Now, with an output from the network we can evaluate the performance of our network by comparing with some target data. This comparison, and the subsequent evaluation of our neural network, is done by the cost function.
Which cost-function to choose depends on the desired output of the network, and there are many to choose from. We denote the cost function as a function of the parameters, θ_i, in our network as

$$C(\theta_i).$$

In many cases, especially regression analysis, a good choice for a cost function is the Mean Squared Error (MSE), which we can express simply as

$$C(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y(\theta)_i - t_i)^2, \tag{15}$$

where y_i is the output from our network, and t_i the targets that we'd like our network to find. If the parameters in our model is mostly the weights in each layer, we can see the cost-function as a function of the weights in a specific layer as

$$C(\mathbf{W}^l) = \frac{1}{n}\sum_{i=1}^{n}(a_i^l - t_i)^2.$$

What is now interesting, is we can try to optimize our network by looking at how this costfunction *change* as we vary the parameters (weights), and thus the output, of our network, and by that, improve the performance. Finding the derivative

$$\frac{\partial C(w_{ij}^l)}{\partial W^l} = \frac{\partial C(w_{ij}^l)}{\partial a_i^l}\frac{\partial a_i^l}{\partial w_{ij}^l} = \frac{2}{n}(a_i^l - t_i)\frac{\partial a_i^l}{\partial w_{ij}^l}.$$

Now the final derivative depends on our choice of activation function in layer l. This can be further derived to find a final expression for the cost function as(full derivation is found in the lecture notes [4] chapter 13):

$$\nabla_W C(\mathbf{W^l}) = \frac{\partial C(\mathbf{W}^l)}{\partial w_{ij}^l} = (a_i^l - t_i)a_i^l(1 - a_i^l)a_j^{l-1} = \delta_i^l a_j^{l-1}, \tag{16}$$

where we've defined the error, δ^l in layer l:

$$\delta_i^l = f'(z_i^l)\frac{\partial C}{\partial a_i^l} = a_i^l(a_i^l - t_i)(1 - a_i^l).$$

This quantity will depend on how much the output from layer l affects the cost-function, and also how much the activation function f at layer l will vary at a given activation value z_i.

2.3.4 Back propagation

With an expression for our cost-function, we can now optimize the weights and parameters in our network (naturally, the cost function is a function of all the weights since they directly influence the output of the network). Meaning, we have the following:

$$\theta_{i+1} = \theta_i - \eta \nabla_W C(\theta_i). \tag{17}$$

This shows, that by optimizing our cost function by changing the various weights in the layer of our network, we directly improve the performance of our neural network. Furthermore, as seen from eq. (16), each error in the hidden layer will depend on the output of the previous layer - and by calculation (again, see full derivation in lecture notes[4]) we can make a difference equation for the error (going backwards):

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

This equation is the backbone of the entire backpropagation algorithm! With this, we can go backwards from the outputlayer - through all the hidden layers - to the input layer and then find the gradients to the weights in each layer. Then we can use eq. (17) to update the weights - by gradient descent (see theory for various gradient descent methods in chapter 7 in the lecture notes[4]) - and improve our network gradually. In our implementation we've chosen SGD to optimize our network after performing the backpropagation algorithm.

2.3.5 Running the network

Now we are ready to set the structure for how our network will train, and improve it's performance, with these steps

- Initialization
  - We must first initialize our network structure - by this we need to define the number of hidden layers, the number of nodes in each layer and choose initial values for the weights and biases. As previously mentioned, this is done by sampling from a normal distribution for the weights, while the biases are all initialized at 0.01.
- Feed forward
  - Giving an input to the network will now produce an output, as the data is gone through all the layers. With this output, one can calculate the costfunction and evaluate the performance of the network.
- Backpropagation
  - With the cost function calculated, we can begin optimizing the parameters in the network by backpropagating, starting with the output layer and working back through all the layers.

- Prediction
  - With out networks weights trained and optimized, we may use the network to make predictions, and if we've optimized it successfully, we should get some pretty results.

The implementation of all these steps numerically can be seen in the Neural Network script FFNN.py which is presented on github[6]). Follow the README.md instructions (and the comments in the scripts) to run the various algorithms.

2.3.6 Optimizing the network: Hyper-parameters

The MLP neural network has many hyper-parameters that may change both it's performance *and* stability. These deep learning neural network are very sensitive to adjustments and choices of these hyper-parameters, which may include the following

- Learning rate
- Regularization
- Initialization
- Activation function
- Scaling of input data
- Number of hidden layers
- Number of neurons in hidden layers
- Choice of optimization algorithm
  - Parameters belonging to said optimization algorithm

There are no fixed answer to what the optimal values for these parameters are, and is something to be found by trial-and-error. Poor choices of parameters may cause the network to diverge greatly, and perhaps not learn at all, and good choices may make the network incredibly efficient and accurate. In our implementation we've chosen a constant learning rate, and the way of SGD to optimize the weights and biases.

[6]code at `https://github.uio.no/Jonnyai/FYS-STK4155/tree/main/project%202`

2.4 Logistic regression

Logistic regression is a simple method used for classifying a set of input variables $\boldsymbol{x}$ to an output or a class called y_i, $i = 1, 2, ..., K$ Where K represents number of classes. The prediction output classes which the input variables belongs to is given by the design matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$, where this matrix contains n samples that carry p features. We separate between hard and soft classification, where this determines the input variable to a class deterministically or the probability that a given variable belongs in a certain class.

We study the binary, two-class case with $y_i \in [0, 1]$. The probability that the input variable x_i belongs in class y_i is given by the logistic function, also called the sigmoid function:

$$p(x) = \frac{1}{1 + e^{-x}}. \tag{18}$$

A vectorset of predictors $\boldsymbol{\beta}$, which is prior to estimate with our data gives the following probabilities:

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp\Big(\boldsymbol{\beta}^T x_i\Big)}{1 + \exp\Big(\boldsymbol{\beta}^T x_i\Big)}, \tag{19}$$

$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}). \tag{20}$$

We define a set of all possible outputs in our dataset $\boldsymbol{\mathcal{D}} = \{(y_i, x_i)\}$, with binary labels $y_i \in [0, 1]$, where these data points are drown independently and are identically distributed. Therefore we introduce the method used in primarily statistics called Maximum Likelihood Estimations(MLE) principle. We approximate the total likelihood for all possible outputs of $\boldsymbol{\mathcal{D}}$ by the product of the individual probabilities of a spesific output y_i, which follows:

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|x_i, \boldsymbol{\beta}]^{y_i} [1 - p(y_i = 1|x_i, \boldsymbol{\beta})]^{1-y_i}. \tag{21}$$

We maximize our probability by using MLE, by defining doing logarithmic of eq(21), hence we obtain log-likelihood in $\boldsymbol{\beta}$:

$$\log(P(\mathcal{D}|\boldsymbol{\beta})) = \sum_{i=1}^{n} (y_i \log p(y_i = 1|x_i, \boldsymbol{\beta}) + (1 - y_i) \log[1 - p(y_i = 1|x_i, \boldsymbol{\beta})]). \tag{22}$$

By reordering the logarithms we will also have our cost function from eq(22):

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \Big[y_i \boldsymbol{\beta}^T x_i - log\Big(1 + exp(\boldsymbol{\beta}^T x_i)\Big)\Big]. \tag{23}$$

We minimize our cost function which is also the same as maximizing the log-likelihood, and this give us:

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\sum_{i=1}^{n} x_i(y_i - p(y_i = 1|x_i, \boldsymbol{\beta}) = 0. \tag{24}$$

This expression can be rewritten to a more compact form by defining the daigonal matrix $\boldsymbol{W}$ wth elements $p(y_i = 1|x_i, \boldsymbol{\beta})(1 - p(y_i = 1|x_i, \boldsymbol{\beta})$, $\boldsymbol{y}$ vector with y_i values and $\boldsymbol{p}$ as the vector of fitted probabilities. We can then express eq(24)in matrix form as such:

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{p}), \tag{25}$$

which is easily implemented into Python and are shown in appendix section((7)). Where we've implemented the sigmoid function, the cost function and also the gradient of the cost function. We will use SGD to optimize this function when we develop our logistic regression algorithm.

When using this Logistic Regression method it will give us an array of probability. We chose a single output, meaning each point in the array will be a number $y$pred$\in [0, 1]$. We use it as a hard classifier meaning the following prediction is made

$$y_{\text{pred}} = \begin{cases} 1, & y \geq 1 \\ 0, & y < 0.5 \end{cases}$$

3 Results

3.1 Gradient Descent

3.1.1 Optimal hyper parameters: comparison OLS vs RIDGE

Implementing the methods of gradient descent as presented in section (2.2) to optimize the cost functions for both OLS and Ridge to solve linear regression (as seen in the subsequent subsections), and applying this to the 1d polynomial and the simple dataset introduced in the method/theory section, we can produce the following table when studying how the MSE vary when changing the learning rate (with OLS defining our cost function). The results are presented in table (1)

	$\boldsymbol{\eta} = \mathbf{0.00001}$	$\boldsymbol{\eta} = \mathbf{0.0001}$	$\boldsymbol{\eta} = \mathbf{0.001}$	$\boldsymbol{\eta} = \mathbf{0.01}$	$\boldsymbol{\eta} = \mathbf{0.1}$
MSE	16.261	10.242	1.663	1.600	1.598

Table 1: MSE as a function η, the learning rate. We ran the gradient descent method for 1000 number of iterations.

We perform the same analysis, but now we add the l_2 - regularization term that characterize the cost function using Ridge, and produce the following table

	$\boldsymbol{\eta} = \mathbf{0.00001}$	$\boldsymbol{\eta} = \mathbf{0.0001}$	$\boldsymbol{\eta} = \mathbf{0.001}$	$\boldsymbol{\eta} = \mathbf{0.01}$	$\boldsymbol{\eta} = \mathbf{0.1}$
MSE	16.255	10.250	1.690	1.515	1.513

Table 2: MSE as a function of learning rate η, with constant $\lambda = 0.1$ for the same number of iterations, we've chosen $\lambda = 0.1$ due to optimal MSE, this is to be shown in a heatmap plot in appendices7.

3.1.2 Rate of convergence: Gradient Descent methods

By implementing the various steps introduced in the method and theory section for PlainGD(**??**), PlainGD with momentum(**??**), SGD(3) and SGD with momentum(**??**) we produce a rate of convergence for MSE as a function of number of iterations which is set to equal 100.

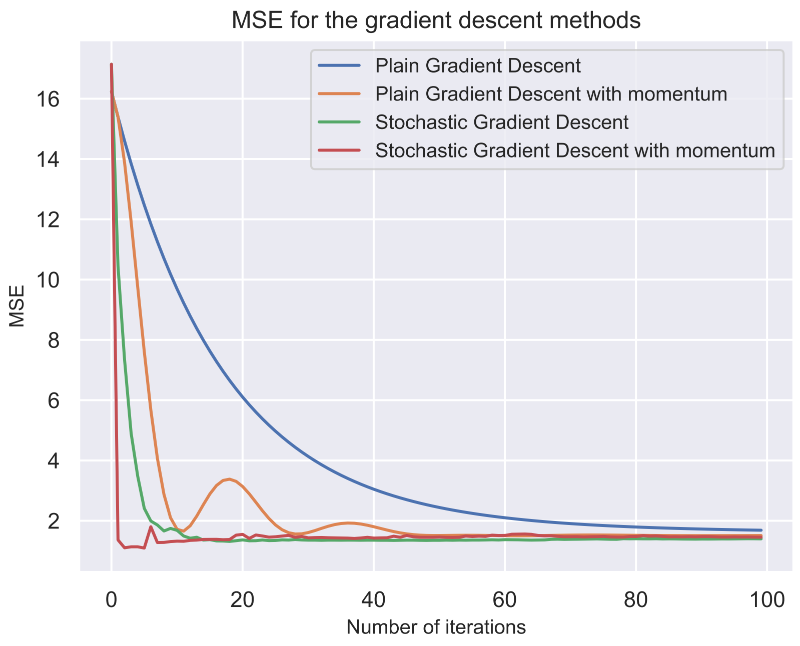


Figure 2: In this figure we see MSE as a function of gradient descent steps for PlainGD, PlainGD with momentum, SGD and SGD with momentum, where we have set hyperparameters $\eta = 0.01$, $\lambda = 0.1$, $\gamma = 0.9$, $M = 10$ and n iterations of epochs to be 100.

3.1.3 Rate of convergence: Adaptive learning methods using SGD

Implementing the methods for adaptive scaling of the learning rate η, as presented in sections (2.2.5), using stochastic gradient descent (see (2.2.3)). We perform the gradient descent method using decaying learning rate, with the same function $f(x)$ on the same dataset $x \in [0, 1]$ for 100 data points, and compare rates of convergence by studying the MSE score(we base this in the theory section of report[2]), as we did for the various GD methods (2.2). With this, we produce the following plot

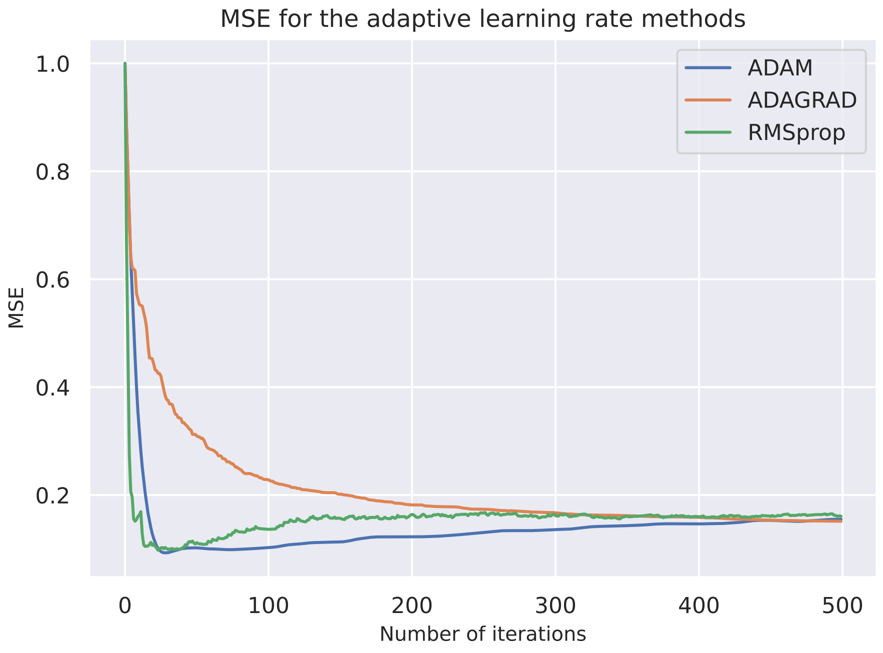


Figure 3: Rates of convergence for the various adaptive learning methods to scale the learning rate $\eta(t) = \frac{5}{50+t}$, i.e $t_0 = 5, t_1 = 50$, using $M = 10$, $n = 100$, $\lambda = 0.001$

3.2 Feed-Forward Neural Network: Regression

By implementing the various steps introduced in the method section for the neural network (2.3), and choosing the hidden layers' activation function to be the Sigmoid, ReLu or Leaky Relu function and a linear activation function (does nothing) in output layer we've built a FFNN for regression. The code can be found in FFNN.py on GitHub[7]. We test our network for various hyper-parameters (see section (2.3.6)), and we find that the best error scores (MSE and R2) is achieved (at 10 000 epochs using ReLu), with 1 hidden layer containing 4 neurons to be $\eta = 0.1$, $\lambda = 10^{-6}$

[7]`https://github.uio.no/Jonnyai/FYS-STK4155/tree/main/project%202`

3.2.1 Comparison between activation functions

By choosing a different activation function for the hidden layers, namely Sigmoid, ReLu or Leaky ReLu as presented in section (2.3.2), we may improve the performance of our network. The results (MSE and R2score) are presented in table (3), with Scikit-Learns Linear Regression model used on the same dataset included for comparison

	Sigmoid	**ReLu**	**Leaky ReLu**	**SKLearn linreg**
MSE	0.018	0.001	0.001	NaN
R2 score	0.591	0.974	0.974	0.537

Table 3: Error estimates for the three different activation function in the hidden layers, with Scikit-learn linear regression for comparison. Network trained with 10000 epochs, $\eta = 0.1$, $\lambda = 10^{-6}$, 1 hidden layer with 4 neurons

3.2.2 Adjusting hyper-parameters: Sensitivity of FFNN

By increasing the complexity of our FFNN, we can produce the following fit to the data

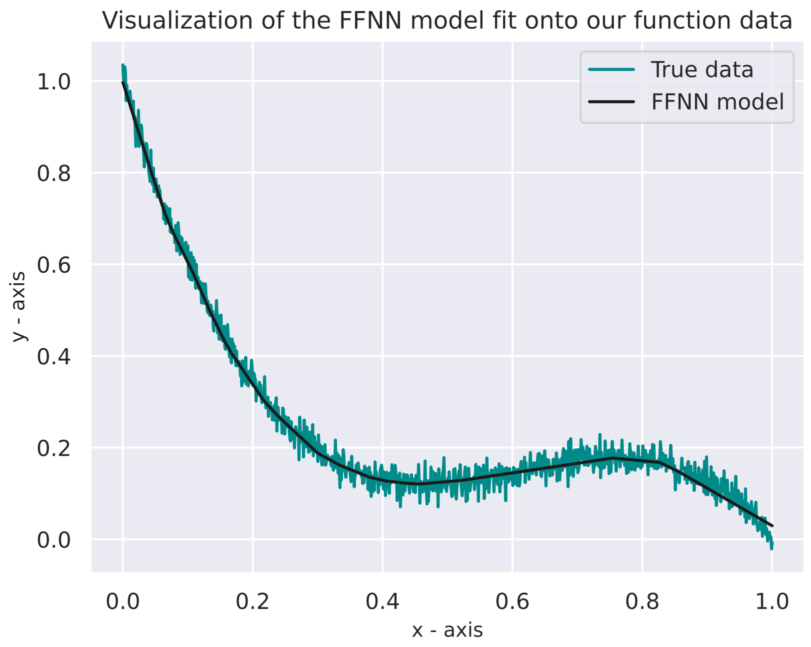


Figure 4: Linear approximation from our FFNN, predicted at entire dataset x,y, with 20 neurons, $M = 10$ and 5 hidden layers (500 epochs)

and this fit to the data has an R2 score of: 0.990 and MSE score: 0.00044.

We fiddle with the complexity and produce the following tables of error scores

	Hidden.L = 1	**Hidden.L = 5**	**Hidden.L = 10**
MSE	0.0016	0.0004	NaN
R2 score	0.973	0.990	NaN

Table 4: Hidden.L = Hidden layers. 500 epochs, 20 neurons, eta = 0.1, lmbd = 10^{-6}

	Neurons = 1	**Neurons = 20**	**Neurons = 26**
MSE	0.0438	0.0004	NaN
R2 score	-0.0001	0.990	NaN

Table 5: The error scores as function of number of neurons in the hidden layers. $M = 10$, 500 epochs, $l = 5$ using leakyReLu

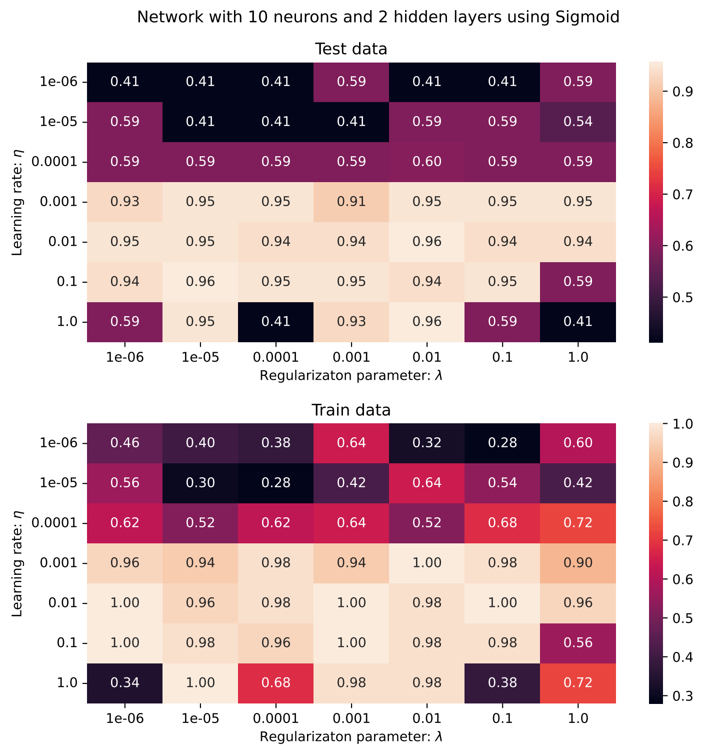
Now, we remove the He et al - heuristic that scales the weight initialization, as we mentioned in section (2.3.1), and perform the same analysis for different number of hidden neurons

	Neurons = 1	**Neurons = 5**	**Neurons = 6**
MSE	0.0438	0.0438	NaN
R2 score	-0.0001	-0.0004	NaN

Table 6: Same hyper-parameters table (5), but without He et al weights scaling.

3.3 Feed-Forward Neural Network: Classification

By implementing the various steps introduced in the method section for the neural network, and choosing the sigmoid-function for both our hidden layers *and* the output layer - we have constructed a network that can do binary classification. We've implemented a hard classifier, and the code for the network is found on github (see README.md for instructions). Running the classifier on the UCI ML Wisconsin Breast cancer dataset included in sklearn[8] for various regularization parameters λ and learning rate η, yields the following accuracy scores, in figure (5) [9]



Test data: η \ λ	1e-06	1e-05	0.0001	0.001	0.01	0.1	1.0
1e-06	0.41	0.41	0.41	0.59	0.41	0.41	0.59
1e-05	0.59	0.41	0.41	0.41	0.59	0.59	0.54
0.0001	0.59	0.59	0.59	0.59	0.60	0.59	0.59
0.001	0.93	0.95	0.95	0.91	0.95	0.95	0.95
0.01	0.95	0.95	0.94	0.94	0.96	0.94	0.94
0.1	0.94	0.96	0.95	0.95	0.94	0.95	0.59
1.0	0.59	0.95	0.41	0.93	0.96	0.59	0.41

Train data: η \ λ	1e-06	1e-05	0.0001	0.001	0.01	0.1	1.0
1e-06	0.46	0.40	0.38	0.64	0.32	0.28	0.60
1e-05	0.56	0.30	0.28	0.42	0.64	0.54	0.42
0.0001	0.62	0.52	0.62	0.64	0.52	0.68	0.72
0.001	0.96	0.94	0.98	0.94	1.00	0.98	0.90
0.01	1.00	0.96	0.98	1.00	0.98	1.00	0.96
0.1	1.00	0.98	0.96	1.00	0.98	0.98	0.56
1.0	0.34	1.00	0.68	0.98	0.98	0.38	0.72

Figure 5: Seaborn heatmap of the accuracy score from our FFNN classifier with 50 epochs, batchsize of 50 using 2 hidden layers with 10 neurons. The activation function in each layer is the Sigmoid function

Choosing the parameters $\lambda = \eta = 0.001$, and varying other parameters such as activation function, number of neurons n, number of epochs, batch size M,

[8]see`https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`

[9]see `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html`

number of hidden layers l as explained in section 2.3.6 we produce the following tables of accuracy scores

	Epoch = 100	**Epoch = 300**	**Epoch = 1000**
test score	0.91	0.94	0.95
train score	1.0	1.0	1.0

Table 7: Different epochs, $n = 5$, $l = 1$, $M = 50$, Sigmoid function

	Sigmoid function	**ReLU**	**Leaky ReLU**
test score	0.96	0.95	0.94
train score	1.0	1.0	1.0

Table 8: Different activation functions for the hidden layers, 500 epochs, $n = 5$, $M = 20$, $l = 3$

	Hidden.L = 1	**Hidden.L = 3**	**Hidden.L = 10**
test score	0.96	0.96	0.59
train score	1.0	1.0	0.8

Table 9: Hidden.L = Hidden layers. Different number of hidden layers, 500 epochs, $n = 5$, $M = 20$, Sigmoid function

	Batch size = 20	**Batch size = 35**	**Batch size = 50**
test score	0.95	0.94	0.93
train score	1.0	1.0	0.96

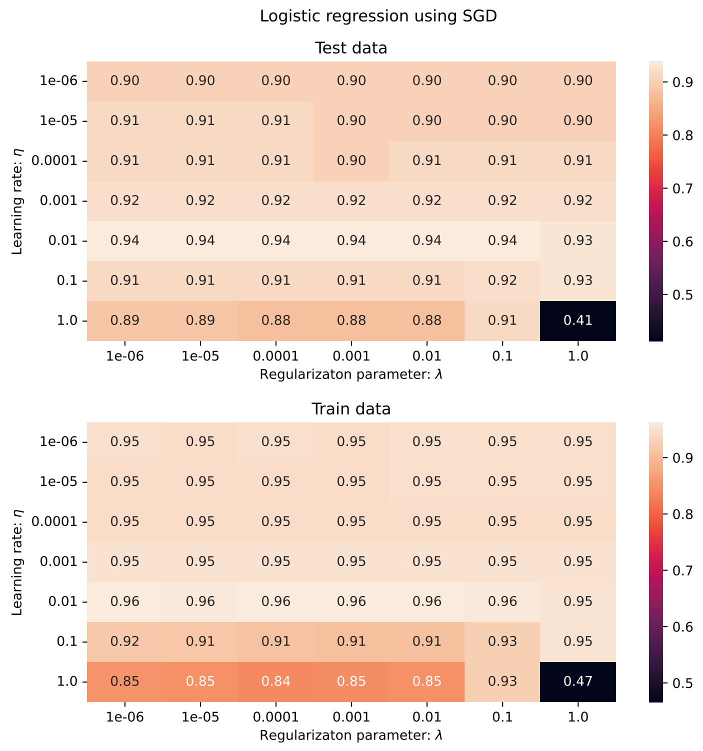
Table 10: Different batch sizes, 500 epochs, $n = 2$, $l = 1$, Sigmoid function

	Neurons = 2	**Neurons = 5**	**Neurons = 10**
test score	0.59	0.96	0.95
train score	0.5	1.0	0.95

Table 11: Different number of neurons, 500 epochs, $l = 3$, $M = 20$, Sigmoid function

3.4 Logistic regression: Classification

Now, we produce a similar heatmap of accuracy score as shown in figure (5) by choosing the same parameters for SGD (50 epochs and batch size = 50), which produce the following



Test data: η \ λ	1e-06	1e-05	0.0001	0.001	0.01	0.1	1.0
1e-06	0.90	0.90	0.90	0.90	0.90	0.90	0.90
1e-05	0.91	0.91	0.91	0.90	0.90	0.90	0.90
0.0001	0.91	0.91	0.91	0.90	0.91	0.91	0.91
0.001	0.92	0.92	0.92	0.92	0.92	0.92	0.92
0.01	0.94	0.94	0.94	0.94	0.94	0.94	0.93
0.1	0.91	0.91	0.91	0.91	0.91	0.92	0.93
1.0	0.89	0.89	0.88	0.88	0.88	0.91	0.41

Train data: η \ λ	1e-06	1e-05	0.0001	0.001	0.01	0.1	1.0
1e-06	0.95	0.95	0.95	0.95	0.95	0.95	0.95
1e-05	0.95	0.95	0.95	0.95	0.95	0.95	0.95
0.0001	0.95	0.95	0.95	0.95	0.95	0.95	0.95
0.001	0.95	0.95	0.95	0.95	0.95	0.95	0.95
0.01	0.96	0.96	0.96	0.96	0.96	0.96	0.95
0.1	0.92	0.91	0.91	0.91	0.91	0.93	0.95
1.0	0.85	0.85	0.84	0.85	0.85	0.93	0.47

Figure 6: Accuracy score as function of hyper-parameters λ and η using SGD with 50 epochs and $M = 50$

4 Discussion

4.1 Pre-processing and scaling of data

In both the regression analysis and the classification problem we used train test splitting of our data to keep our network from overfitting the data, and produced a very good fit to a very specific dataset. This is an issue because we do not know if the "good" fit is truly good because our network is a good approximator, or we've overfitted the dataset in question. Therefore, we solve this issue by splitting the data set and evaluating the trained network/model on a separate test set, to verify the performance of our model properly.

We needed to MinMaxScale (or StandardScale for the classifier, and the logistic regressor) in order for our network to function properly. By min-max scaling the datapoints for the regression problem we force the inputs into the range [0,1], but still keeping all the properties unique to the data. This may prevent our network from diverging and getting exploding gradients, since the floating point numbers representing the datapoints used are much, much smaller. It will also greatly improve the speed of the network, since the "path" from the initial guess is smaller (given you initialize the network using a standard distribution around 0).

Our choice to use StandardScaler when working with the WBC dataset is due to the great deviance in the values of the various features. Thus we standard scale each column vector in the design matrix, which in turn will improve the performance of our network (since we are working with a fully-connected network, remember that each node is a weighted sum of *all* the previous). This may also prevent the network from diverging since the data is uniform, and we can compare it to the same data used for the regression analysis.

4.2 Gradient Descent methods

4.2.1 Optimal hyperparameters: comparison OLS vs RIDGE

In this result, we see in figure(1) and figure(2) that when we increase our size of learning rate, our MSE decreases as well. The MSE is even better in this table(2), that is due to the hyper parameter λ, and its effect on the optimization. We tried to calculate the best learning rate, η by evaluating different λ values, ranging from 0.000001[1E-5 - 1E-1]. The best parameter for our model is $\lambda = 0.01$. By using this parameter we evaluate different η values to find the best estimate for MSE. By implementing the λ parameter, and thus moving to Ridge regression, we can see that the MSE is decreasing. This is to be expected since when we add another tunable parameter to our cost function, that will penalize outlying β values (when the method is trying to optimize, as is characteristic of Ridge regression, see chap. 4 in lecture notes[4]), we assume to better localize a local minima for our cost function.
If we look at the comparison between the two tables we expect that table(2) shows a more significant decrease in MSE than table(1), which is backs our

choice of using Ridge Regression in our subsequent analysis - more hyper-parameters give us more control of the outcome in the various methods, and this is especially true when developing the code for our FFNN.

4.2.2 Convergence: Gradient descent methods

In figure (2) we observe the rate of convergence of all the gradient descent methods we wanted to study. We evaluate the methods and see that each method differs in how many numbers of iterations they iterate through before they reach a low MSE. We can see a clear pattern difference in all four methods, where the momentum of each corresponding method is more effective then the origin method. We see also that the SGD method reaches a potential local minimum faster then the other methods.

We observe that the rate of convergence increases when we introduce momentum. When we include momentum, as mentioned in section (2.2.2), we can expect that the method will move faster from areas with a high curvature (steep, large gradient), and slow down more when the area is flatter (small gradient). Furthermore, we see that the rate of convergence is more rapid for the stochastic methods - and based on this result we use SGD explicitly in the subsequent analysis (for the remainder of testing, and also implemention in the FFNN). This choice is further backed upon SGD being more cost effective numerically, since we need only evaluate the gradients on a smaller subsection of the entire data set, which can be seen in section((2.2.3))

Another observation to be made, is the fluctuating MSE score for the methods including momentum. This stems exactly from the momenta term, if the method finds the minima, but the previous gradient was rather large, it may result in "stepping" out of the local minima forcing the method to go backwards. This fluctuation seen in SGD with momentum, and while not outperforming the regular SGD method, further backs our choice in SGD for our NN code.

4.2.3 Convergence: Adaptive learning rate methods

We can clearly see the various rates of convergence from the different learning rate scaling methods in figure (3). From this, we can deduce that RMSprop and ADAM optimization greatly outperform the method of ADAGRAD. What we experienced finding this result was that ADAGRAD was more computationally demanding (at least for our implementation), and this makes ADAGRAD even less appealing to use for optimizing. There is, however, very little to differentiate the methods of RMSprop and ADAM - but we can see the following: While RMSprop convergences at a faster rate towards what seems to be a minimum, it diverges and ADAM will reach a "better" minima for the MSE after few iterations. After 400 iterations all three methods seem to converge to the same MSE value (finds the same local minima for the cost function).

4.3 Neural Network

When running our neural network code, we experience firsthand how sensitive both the performance and stability of our network are to adjustments in the hyperparameters. This is what we've illustrated in section (3.2.2). We can see that by increasing the complexity, we may either produce better results or make the network diverge and cause overflow (denoted by NaN). An interesting point, that strengthens our choice to make the He et al initialization scaling, is when we remove it - our system can't seem to handle more than 5 hidden layers before diverging, compared to the 25 it could handle beforehand! This shows what great effect this scaling had to reduce the likelihood for exploding gradients (like we assumed in the method section). We experienced the same happening when we excluded the MinMaxscaling of our data, but this is left to the reader to verify as we did not have the time to produce pleasing results to illustrate. Note that these results were produced when using the network for regression.

4.3.1 Regression Analysis

When working our network for regression, we made an attempt to illustrate what activation functions was deemed best for this purpose. We expected that the ReLu variants would outperform the Sigmoid when doing regression (since the sigmoid forces all data to [0,1], putting alot of dependency on the output weights to produce the actual datapoints) - and this is what we found. There was a great increase in accuracy when switching from Sigmoid to ReLu (or leaky ReLu) and we even outperformed the scikitlearns linear regression (note that we did not try our best to make scikit learn linreg model work very well either)!

We also illustrate how nicely the curve produced from predicting with our network on the entire dataset in figure (4) would look. Something interesting to note is the shape on the fitted line, it seems to have many straight lines, and they are shaped almost like the ReLu function (flipped sideways etc.). This may come from the fact that with multiple-hidden layers (the figure was produced with a deep network, 5 hidden layers), we evaluate the data inside the network on the ReLu function multiple times, so it is natural to expect that this will leave its mark on the output! In fact, by choosing proper hyper-parameters, we were able to see clearly how the system starts by "guessing" a simple ReLu shape, and by increasing the number of epochs (or complexity) we could see how the network "shaped" the ReLu function to fit the curve better. As this was not explicitly related to the goal of this project we excluded this from the result section, but it was an interesting observation nonetheless.

4.3.2 Classifier

Working with the classifier, we found that it needed less epochs to produce satisfying results (satisfying means an accuracy score 95%), which makes sense if you take into account our discovery of the network attempting to "fit" the activation functions to the data. When doing classification the network wants to produce a binary output, either 0 or 1 - and not a ”continuous” line. We found good results using only 50 epochs (compared to the 10 000 epochs for the regression analysis), and with only needed two hidden layers. We could most likely produce similar, or better, performance levels using fewer layers since a classifier is not as dependent on the weights producing the continuous output as we mentioned - we only want 0 or 1.

The heatmap shown in figure (5) illustrates how the networks accuracy depends heavily on the learning rate parameter η and the regularization λ. In the ensuing tables we see how the classifier seem to be more robust than the regression. This is a direct result from using Sigmoid as an activation layer one can assume, since this makes both the output error smaller (resulting in smaller gradients) and the output-input between layers are scaled in the range [0,1]. We also find little deviance in the accuracy when moving from Sigmoid to the ReLu functions, but we still find Sigmoid to be the best. With less optimal parameters we may find a greater deviance, but we still produce the expected result - Sigmoid outperforms the other activation functions.

4.4 Logistic Regression

After building our logistic regression algorithm to perform classification, we evaluate it on the same, scaled WBC dataset - and produce a similar heatmap that we made for the FFNN classifier. We can see that the LR algorithm is more stable in the performance of the analysis when adjusting the hyper-parameters, and overral performs better (if we do not optimize the regressor) - however, the FFNN performs better for optimized hyper-parameters which is in tune with our previous findings that the neural network is greatly sensitive to the choice of hyper-parameters.

5 Conclusion

When building our neural network we found time and time again how incredibly sensitive the networks performance and stability is to adjustments in it's hyper-parameters. By picking poor parameters our network diverges, or doesn't learn what so ever, and with good parameters it outperferms more stable and robust models such as regular linear regression using SciKit learn, or our own logistic regression model for classification.

The conclusion we make from studying, and comparing, neural networks to other well-implemented algorithms that solve similar problems - is that the neural network is a much more effective method to solve these problems, given that we use it correctly!

In tune with the results we presented, we found that Ridge outperforms OLS, and gives more flexibility in our algorithms when looking to define a cost function to optimize. Moving onwards, we found that ADAM and RMSprop yield the best convergence rate when comparing between adaptive learning rate methods, but the extra computational cost (and increased complexity when implementing this numerically) made us choose a constant learning rate when building the neural network. We saw that this choice still made our network outperform more ingrained methods (with optimal parameters).

References

[1] F. Dadvar and J. A. Igeh, *Modelling 2d terrain data by linear regression: An introduction to simple machine learning*, 2022. [Online]. Available: `https://github.uio.no/Jonnyai/FYS-STK4155/tree/main/project%202`.

[2] F. Dadvar and J. A. Igeh, *Modelling 2d terrain data by linear regression: An introduction to simple machine learning*, 2022. [Online]. Available: `https://github.uio.no/Jonnyai/FYS-STK4155/tree/main/project_1`.

[3] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, 2015. DOI: `10.48550/ARXIV.1502.01852`. [Online]. Available: `https://arxiv.org/abs/1502.01852`.

[4] M. H. Jensen, *Applied data analysis and machine learning*, Copyright 2021, 2021. [Online]. Available: `https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html#`.

A Appendix

A.1 MSE change for given λ and η in a heatmap plot.

We evaluated the MSE for various and η values and found a good estimate for which λ value to use when observing the optimal change in MSE for set of learning rate values, which is to be shown in the section of results(2)

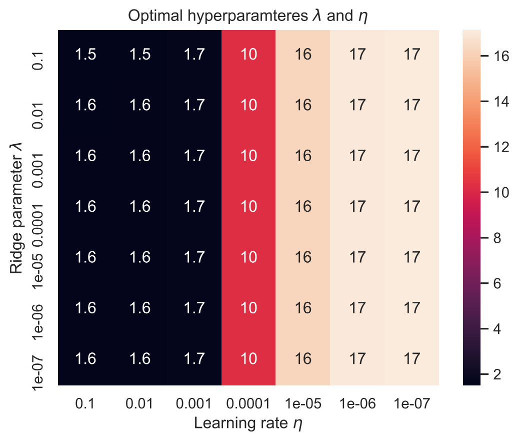


Figure 7: This figure shows the change in MSE for given: hyperparameter and η visualized in a heatmap plot.

A.2 Code: Gradient descent

A.2.1 Plain gradient descent

```python
def PlainGD(X, y, eta, eps, lmbda):
        """Performs linear regression using Plain Gradient Descent
        args:
            X               (np.array): Design matrix
            y               (np.array): Target data
            eta                (float): Learning rate
            eps                (float): Stopping criteria (when gradient is smaller
                                          than this value we stop)
            lmbda              (float): Ridge parameter lambda

        returns:
            beta            (np.array):   Array of beta parameters in linear model
        """
        beta0 = np.random.randn(X.shape[1], 1)
        gradient = 2 / n * X.T @ (X @ beta0 - y) + 2 * lmbda * beta0
        beta = beta0 - eta * gradient
        while np.linalg.norm(gradient) > eps:
            gradient = 2 / n * X.T @ (X @ beta - y) + 2 * lmbda * beta
            beta -= eta * gradient
        return beta
```

Code Listing 1: The following code shows PlainGD implementation in python.

A.2.2 Gradient descent with momentum

```python
def PlainGD(X, y, eta, eps, lmbda):
        """Performs linear regression using momentum based Gradient Descent

        args:
            X               (np.array): Design matrix
            y               (np.array): Target data
            eta                (float): Learning rate
            eps                (float): Stopping criteria (when gradient is smaller
                                          than this value we stop)
            lmbda              (float): Ridge parameter lambda
            gamma              (float): Momentum parameter

        returns:
            beta            (np.array):   Array of beta parameters in linear model
        """
        beta0 = np.random.randn(X.shape[1], 1)
        gradient = 2 / n * X.T @ (X @ beta0 - y) + 2 * lmbda * beta0
        beta = beta0 - eta * gradient
        v_prev = 0
        while np.linalg.norm(gradient) > eps:
            gradient = 2 / n * X.T @ (X @ beta - y) + 2 * lmbda * beta
            v = gamma * v_prev + eta * gradient
            beta -= v
            v_prev = v

        return beta
```

Code Listing 2: The following code shows gradient descent with momentum implementation in python.

A.2.3 Stochastic gradient descent

```python
def SGD(X, y, eta, M, n_epochs, lmbda):
        """Performs linear regression using Stochastic Gradient Descent
        args:
            X               (np.array): Design matrix
            y               (np.array): Target data
            eta                (float): Learning rate
            M                  (int)  : Size of minibatches
            n_epochs           (int)  : Number of epochs (iterations of all minibatches)
            lmbda              (float): Ridge parameter lambda
        returns:
            beta            (np.array):   Array of beta parameters in linear model
        """
        beta = np.random.randn(X.shape[1], 1)
        n = X.shape[0]
        m = n / M
        for epoch in range(n_epochs):
            for i in range(m):
                k = np.random.randint(m-1)
                Xi = X[k*M:k*M+M]
                yi = y[k*M:k*M+M]
                gradient = 2 / M * (Xi.T @ (Xi @ beta - yi)) + 2 * lmbda * beta
                beta -= eta * gradient
        return beta
```

Code Listing 3: The following code shows stochastic gradient descent implementation in python.

A.2.4 Stochastic gradient descent with momentum

```python
def SMGD(X, y, eta, gmama, M, n_epochs, lmbda):
    """Performs linear regression using momentum based Stochastic Gradient Descent
        args:
            X               (np.array): Design matrix
            y               (np.array): Target data
            eta                (float): Learning rate
            gamma              (float): Momentum parameter
            M                  (int)  : Size of minibatches
            n_epochs           (int)  : Number of epochs (iterations of all minibatches)
            lmbda              (float): Ridge parameter lambda
        returns:
            beta            (np.array):   Array of beta parameters in linear model
        """
        beta = np.random.randn(X.shape[1], 1)
        n = X.shape[0]
        m = n / M
        v_prev = 0
        for epoch in range(n_epochs):
            for i in range(m):
                k = np.random.randint(m-1)
                Xi = X[k*M:k*M+M]
                yi = y[k*M:k*M+M]
                gradient = 2 / M * (Xi.T @ (Xi @ beta - yi)) + 2 * lmbda * beta
                v = gamma * v_prev + eta * gradient
                beta -= v
                v_prev = v
        return beta
```

Code Listing 4: This following code shows stochastic gradient descent with momentum implementation in python.

A.3 Code: Adaptive learning method

This Python code shows the implementation of these adaptive learning rate methods:

- Decaying learning rate
- RMSprop
- ADAM
- ADAGRAD

```python
def ADAM(beta):
    delta = 1e-8
    avg_time1 = 0.9
    avg_time2 = 0.99
    m_prev = 0
    s_prev = 0
    t = 0
    g = ... # Gradient of C wrt. parameters
    while np.linalg.norm(g) > eps:
        g = ... # Gradient of C wrt. parameters
        m = avg_time1 * m_prev + (1 - avg_time1) * g
        s = avg_time2 * s_prev + (1 - avg_time2) * g **2
        mhat = m / (1 - avg_time1 ** t)
        shat = s / (1 - avg_time2 ** t)

        beta -= eta * mhat / (np.sqrt(shat) + delta)
        m_prev = m
        s_prev = s
        t += 1

    return beta

def RMSprop(beta):
    delta = 1e-8
    avg_time = 0.9
    s_prev = 0
    g = ... # Gradient of C wrt. parameters
    while np.linalg.norm(g) > eps:
        g = ... # Gradient of C wrt. parameters
        s = avg_time * s_prev + (1 - avg_time) * g ** 2

        beta -= eta * g / (np.sqrt(s) + delta)
        s_prev = s

    return beta

def ADAGRAD(beta):
    g = []
    delta = 1e-8
    while np.linalg.norm(g[-1]) > eps:
        g.append(... # Gradient of C wrt. parameters)
        k = len(g)
        G = sum(g[i] @ g[i].T for i in range(k))
        dim1, dim2 = G.shape
        I = np.eye(dim1, dim2)
        beta -= eta * 1 / np.sqrt(np.diag(G + delta * I)) * g

    return beta
```

Code Listing 5: In the following Python code, the latter 3 (advanced) methods of adaptive learning are presented numerically.

A.4 Code: Forward propagation

```
def forward_prop(self, input_value):
    self.input = input_value
    self.val = self.input @ self.weights + self.bias
    self.output = self.activation_function(self.val)

    return self.output
```

Code Listing 6: This Python code shows the implementation of Forward propagation algorithm.

A.5 Code: Sigmoid and cost -function, and gradient cost function

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def cost_func(self, ytilde):
    loss = -np.mean(self.y * (np.log(ytilde)) - (1 - self.y) * np.log(1 - ytilde))

    return loss

def grad_cf(self, X, y, ytilde):
    m = X.shape[0]
    gradW = - 1 / m * X.T @ (y.reshape(ytilde.shape) - ytilde)

    gradb = 1 / m * np.sum((y.reshape(ytilde.shape) - ytilde))

    return gradW, gradb
```

Code Listing 7: This Python code shows the implementation of Sigmoid function, cost function and the gradient of the cost function.