Project 2 FYS-STK4155

Sigurd Holmsen, Øystein Høistad Bruce

November 2021

Contents

1 Abstract

A feed forward neural network is inspired by how a biological brain processes information, and is one of the most popular machine learning models. Our aim in this project is to write and present a neural network code, test it, and compare the results to other popular machine learning algorithms. In the previous project we looked at linear regression, where we assumed a specific model (polynomials) to apply to a data set. However, neural networks have the advantage that they do not need any assumptions on the complexity of the data, but we can feed data directly into the model and usually end up with a good fit. We will show some of the advantages/disadvantages of neural networks compared to other machine learning methods.

2 Introduction

After we have explained and written a neural network code, we want to test both how well it is able to predict data and how quickly it can be trained. We will adapt the model to both a linear regression case and a binary classification case. We will do this in several steps.

First, we need to program a gradient descent algorithm, and we will be using a stochastic gradient descent to save computing time. A gradient descent uses gradients of a given cost function to optimize the model parameters. We will also chose a momentum based gradient descent which uses the steepness of the last gradient to adjust the length of the next step. The stochastic gradient descent chooses a random subset of the data to compute gradients, so that we do not need to use the entire data set.

Next up, the neural network class. We implemented a code that was able to initialize and train a feed forward neural network. We are going to integrate the stochastic gradient descent algorithm as the method for tuning the weights and biases inside the model. We will analyse both a regression- and classification problem with the neural network and compare the results against the linear- and logistic regression methods respectively.

In this report, we are going to look over the methods we have been implementing in this project. Then, an introduction to the result section with some general information about the upcoming analysis, followed by the actual analysis and discussion. In the end, we will conclude the project with some learning outcomes and further analysis we would like to do.

We will introduce the data we have used in the sections 4.1.2 and 4.1.3 for the regression- and classification case, respectively. The code is stored in the Github repository attached at the last page of the report (section 6).

3 Method

Here, we explain the functions we have programmed and some concepts behind them. The functions take in input data X and output data y, so data must be generated or extracted from somewhere before the models can be used.

3.1 Measure the quality of a model

3.1.1 R2-score

We will look at the R2 score when adapting a model to data where the output values are a subset of $\mathbb{R}$.

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2} \tag{1}$$

3.1.2 Accuracy score

We will look at the accuracy score to measure how well the adapted model is, when the target values are discrete. The accuracy is the same as the amount of correct prediction you will get with the model.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i)}{n} \tag{2}$$

where

$$I(t_i) = \begin{cases} 1 & t_i = y_i \\ 0 & t_i \neq y_i \end{cases}$$

t_i is the predicted target value of the model, y_i is the actual value and n is the number of data points.

3.2 Gradient Descent

The goal of gradient descent is to find parameters that minimizes a cost function. We have written a gradient descent function that takes in initial values of the parameters that will be tuned, a cost function, the learning rate η, a hyper-parameter λ and the number of iterations, as well as input and output data X, y. The learning rate η decides the length of each step in the search direction. In this method, we have used the python package autograd for finding the gradients of the cost function. We then use the gradients and the learning rate to converge towards the minimum of the cost function, after which the function returns the updated/ tuned parameters that define the model. The algorithm we used is explained in the next subsections.

3.2.1 Momentum

We have used a momentum based gradient descent, which means the length of each step in the gradient descent depends on the size of the previous gradient. The point is to do longer steps when gradients are steeper, and smaller steps when the gradients are flatter. In this way, the gradient descent adjusts to the "terrain" of the cost function. The algorithm can be expressed as follows (inspired by [2]):

$$v_t = \gamma v_{t-1} + \eta_t \frac{dC}{d\theta} \tag{3}$$

$$\theta_{t+1} = \theta_t - v_t \tag{4}$$

where the parameter $\gamma \in [0, 1]$ is the amount of momentum we want to use ($\gamma = 0$ means no momentum). v_t is the amount of tuning of the parameter at time t. θ is the parameter we want to tune, and $\frac{dC}{d\theta}$ is the change in cost with respect to θ.

3.2.2 Stochastic

We have made our gradient descent method stochastic, meaning we do not necessarily compute the gradients for the entire data set X, y, but for a small, randomly chosen batch X_k, y_k. If we let the size of the mini-batches be M, we will have m mini-batches, where $m = \frac{\text{number of data}}{M}$ (rounded down to nearest integer). We then select a random integer $k \in [0, 1, ...(m-1)]$, and select the k-th mini-batch:

$$X_k = X[k \cdot M : (k+1) \cdot M]$$
$$y_k = y[k \cdot M : (k+1) \cdot M]$$

In this way, we have produced a randomly chosen mini-batch from X, y, and computed the gradient $\frac{\text{number of data}}{M}$ times. We also decide the number of epochs, that is how many times we will repeat this process. The goal of the stochastic gradient descent is to save computational power and time, as we compute lower dimensional gradients.

3.3 Neural Network

3.3.1 Architecture

We have programmed a neural network class. Regarding the network architecture, the class takes the following parameters: number of input nodes, number of output nodes and a list of hidden layers with corresponding number of nodes. Each element of the list of hidden layers is an integer representing the number of nodes of the related layer. This allows us to set up the network architecture however we would like, where number of nodes per layer can vary. For more

information on how to initiate a class, please look at the README file.

Then we initialize the weights using the standard normal distribution and the biases as small float numbers, more precisely a vector where every element is 0.01. The class also has parameters for the activation function for the hidden layers and for the output layer, where the Sigmoid function is the default activation function for the hidden layers. Then, the user can choose the values, in a method, for the stochastic gradient descent. That includes the learning rate η, the regularization parameter λ, batch size and number of epochs in the SGD, and the hyper-parameter γ.

The class does not scale data, so this must be done beforehand if needed. For more information on how the code works, look at the comments and doc-strings in the code, as well as the README file.

3.3.2 Regularization parameters

Our network takes a regularization parameter λ which adds a term to the cost function: $\lambda \|w\|_2^2$. This term is added in attempt to avoid an overfit of the model. This is implemented as a parameter in the Neural Network class as explained above.

3.3.3 Cost Function

When a neural network is initialized, one has to decide which cost function to use, as this will influence the gradient descent, or more specifically the back propagation algorithm. The cost function must be given in a method inside the class. It is important that the cost function can be interchanged, as the linear regression and classification cases use different cost functions.

3.3.4 Feed Forward

The feed-forward function takes a set of data and predicts the output given the model weights and biases. The data should be a matrix of dimension $(\#data, \#features)$. This means we are approximating several output values simultaneously. Each node is connected to all the nodes in the next layer, and every node has its own bias. After each node has received values from the previous nodes, multiplied with the weight matrix and added with the bias, it it sent through the hidden layer activation function. This value is sent to the nodes in the next layer, and this process is repeated until the output node(s) are reached. The output node(s) may have a different activation function than the hidden layers, or it may not have an activation function at all. The feed forward algorithm is inspired by [2].

3.3.5 Activation functions

The activation functions are used in the feed forward part underneath, the different activation function we are looking at in this project are:

Sigmoid

$$f(z) = \frac{1}{1 + \exp -z} \tag{5}$$

Rectified Linear Unit (RELU)

$$f(z) = \begin{cases} z, & z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Leaky Rectified Linear Unit (Leaky RELU)

$$f(z) = \begin{cases} z, & z > 0 \\ -\alpha z & \text{otherwise} \end{cases} \tag{7}$$

for some (small) $\alpha > 0$

3.3.6 Back Propagation

To tune our weight and bias parameters, we use a momentum stochastic gradient descent (SGD). To find gradients for both the weights and biases, we perform the back propagation algorithm. First, we choose a cost-function, for instance a variant of the mean square error, as a measurement of the quality of our model:

$$MSE = \frac{1}{2n} \sum_i (y_i - \tilde{y}_i)^2 \tag{8}$$

Then we are interested in finding the gradients for a given layer l:

$$\frac{\delta C}{\delta w_{ij}^l}, \frac{\delta C}{\delta b_j^l} \tag{9}$$

According to the back propagation algorithm, this can be written as:

$$\frac{\delta C}{\delta w_{ij}^l} = \delta_j^l a_k^{l-1}, \frac{\delta C}{\delta b_j^l} = \delta_j^l \tag{10}$$

where

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{11}$$

Here, f is the activation function for the given layer. We can compute this if we know the value of δ^{l+1}. We can compute, the last layer delta in such way:

$$\delta^L = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{12}$$

So, we are able to find all the δ values starting backwards. After we have found all the gradients for both the weights and biases, we use this for the momentum based SGD in the same way as described in the section above. When the SGD has iterated through all the given epochs, the training of the model is complete. We have been inspired by [2]

3.3.7 Regression case

For the regression case, we will not need an activation function for the output layer. This is because we may want to fit any number from $-\infty$ to ∞, and then we do not want to put a boundary on the value of the output. We also use the MSE as the cost function, which is accounted for in the back propagation algorithm.

3.3.8 Classification case

For the classification case, we want the output to be interpreted as probabilities. Hence we want to limit the output between 0 and 1. Since we are studying a binary classification problem, we use the Sigmoid function as the output layer activation function. We have used the log-loss cost function when deriving the back propagation algorithm:

$$C(W) = -\sum_{i=0}^{n} y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i) \tag{13}$$

3.4 Logistic Regression

When performing the logistic regression, we have used the following cost function and model (inspired by [1]):

$$\prod_{i=0}^{n} p(x_i = 1|\beta)^{y_i}\, p(x_i = 0|\beta)^{1-y_i} \tag{14}$$

Where

$$p(x = 1|\beta) = \frac{e^{\beta_0 + \beta_1 x_1 + ... + \beta_n x_n}}{1 + e^{\beta_0 + \beta_1 x_1 + ... + \beta_n x_n}} \tag{15}$$

In this equation, the $\boldsymbol{\beta}$ is the parameter we want to train and $\boldsymbol{x}$ is the input features and n is the number of input features. We will tune the $\boldsymbol{\beta}$ parameters using a SGD, and when the model is trained, the p function will predict a probability given $\boldsymbol{x}$. In the final model, a probability greater than 0.5 will refer to a prediction of 1 and otherwise the model will be predicting 0.

To train our model, we will perform the SGD as before, where we will use the autograd package to find the gradients of the cost function.

4 Results and discussion

4.1 Introduction to the results

In this section, we will refer to some tests we have made in the **test_project_2.py** file which will be found in the **project2** folder inside the github repository, attached at the end of the report (section 6). To have a great experience with the testing file, we recommend you to read the **README.md** file inside the **project2** folder. We will provide the necessary parameters for reproducing the tests while we present the results.

4.1.1 General comments on the results

Here is a list of things to be aware of according to the results we are presenting:

1. When we are comparing two parameters, we will look at the R2-score and accuracy score for the regression- and classification problem (respectively). This will very often be divided into to groups, training and testing data.

2. Among some great results there will be some *wierd* results also (even with a small change of a parameter value). This is often a case where the model is predicting the same target value for all input value. Such cases can maybe be some *local* minimum of the cost-function. We have not *fixed* those occurrences, but we know about the problem and after some testing we found out that re-initializing the weights and biases would help (more detailed in section 4.1.4).

3. Exploding and vanishing gradients are a central problem in neural networks, as when this occurs, we do not get a well adapted model. To avoid this in our testing, we have often restricted our grid search values.

4. In our analysis, we had to initialize some parameters to be able to start tuning the first parameters. Whenever we achieved some results from a previous testing, we added the best parameters to the new testing.

5. The number of epochs we have chosen in the analysis isn't optimized for the best model. More iterations will, very often, achieve a better model. As a result we have chosen to go with some *friendly* number of epochs to save computing expenses. We will show in section 4.2.1 that this assumption is reasonable.

6. We need to make some initial guess of the parameters we want to tune with the SGD-algorithm. We have split the initial guess into:

   (a) linear regression case: $[0.0, 0.1, 0.1, ..., 0.1]$

   (b) logistic regression case: $[0.1, ..., 0.1]$

4.1.2 Introducing the data - regression case

For the regression case, we have generated data from the Franke function similarly as in project 1. That means we generate x, y -pairs, both randomly selected between 0 and 1, and send them into the Franke function. A noise term is then added, which is a normally distributed value with a noise multiplier parameter *noise*.

We have scaled the data when we compare the results from project 1 with the results from the *SGD*-algorithm. We are doing that because the code from project 1 does so. We do not find it necessary to scale the data in further/ other analysis with the Franke function data, since the data set is restricted, each feature is distributed identically and we are taking a pseudo random sampling.

In our analysis, we have looked at a data set consists of 100 datapoints, and a noise of 0.01.

4.1.3 Introducing the data - classification case

For the classification case, we have used the Wisconsin cancer data found in the Sci-kit learn package *sklearn.datasets*. The data consist of 569 observations and 30 features. If you have a lot of data, then you use all of it to create a best possible model. Therefore we will use all the observations in our analysis.

We will only consider two features from the data set, and that is because we are able to predict very well with just those two. We have utilized principal component analysis to rule out the two most important features from the data set. Principal component analysis (PCA) looks at the unit vectors (representing the features) with the greatest eigenvalues. The two most important features are having a total *explained variance ratio* above 0.99 which is considerably high. We scale the data by the maximum of each input feature, so that the largest input value will be 1 for each feature. This is to adjust for different units for each feature.

4.1.4 Initializing the weights and biases

We are initializing the weights according to a standard normal distribution and all the biases are valued 0.01 before it starts to tune these parameters. We are initializing the biases at 0.01 since it ensures that all neurons have some output which can be backpropagated in the first training cycle.

We have looked at the possibility of refreshing the weights and biases when the predictions are horrible, often when the model is predicting the same targets for all kinds of input, but were told that this wasn't a great way to fix the problem. Those occasions would refer to an *unstable* model, and we should change parameters and rerun the program instead.

4.2 Regression problem

4.2.1 OLS - results of Stochastic Gradient Descent

For the OLS regression, we have compared the R2-score for different learning rates and number of minibatches. We want to check how far we are from the analytical solution from project number 1. As mentioned in section 4.1.2, we will use Franke function data for this problem. The following values are being set:

```
number_of_epochs = 200
degree = 5
gamma = 0.7
run_main_OLS = True
list_number_of_minibatches = [1, 5, 8, 10, 16, 20, 40, 80]
```

In this case, we want to make the number of iterations constant in the SGD-algorithm. For this to be the case, we need to change the number of epochs variable to be the same for different number of minibatches. With these values, we can turn into the **test_project_2.py** file, and scroll to **test 1**. When we run the code, we will be provided with these figures (fig. 1 and 2).

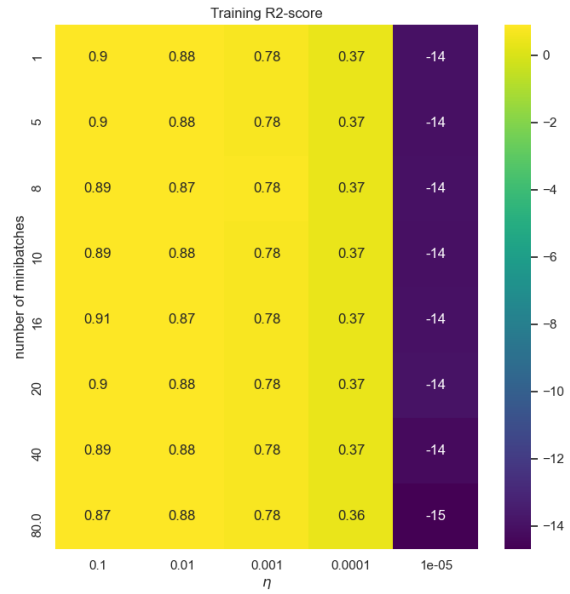

Figure 1: Data: training, R2-score of combination of number of minibatches and the learning rate

We observe that the optimal learning rate is around 0.1 or 0.01 for this case, as this gives the highest R2-score. As a result of that, it will be the closest one to the analytical solution. When we run **test 1**, we will also be informed, with a print in the terminal, that the R2-score for training and testing is around 0.982 and 0.964 (respectively). Which is not that far away from the results we are getting in these numerical approaches, but with a greater number of iterations we would probably get even closer (this will be analyzed further).

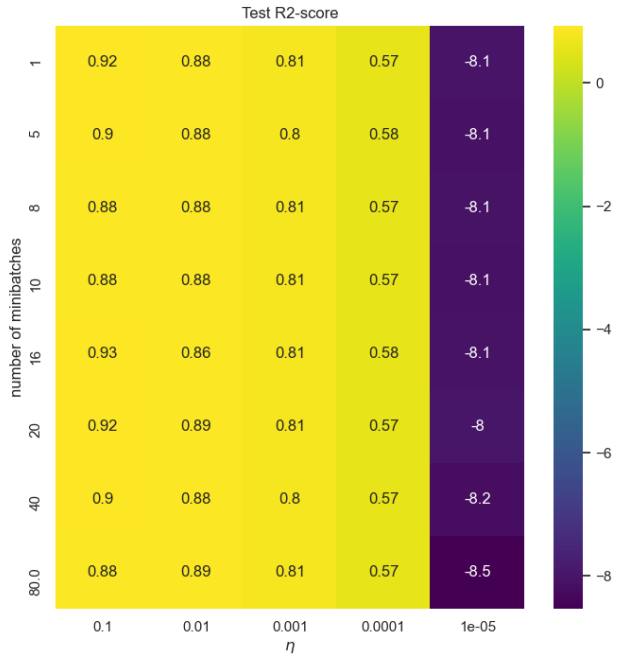

Figure 2: Data: test, R2-score of combination of number of minibatches and the learning rate

If the mini-batch size is too small (equivalently, that the number of minibatches is too high), there may be too much randomness in the computed gradients to converge quickly, and if the size is too large, the problem of computing time arises, which defeats the purpose of the stochastic gradient descent. In our case, we do not see that much difference in the R2-score for different batch sizes. Hence, since the mini batch size does not affect the R2-score we will choose the smallest batch size for faster computing time.

Now, we want to show that higher $number\_of\_epochs$-value leads to better results (if the solution seems to converge). The following two figures (fig. 3 and

4), showing only the data set for testing, are small evidence that the statement is true. In these figures, we are looking at the same problem as above - but with these values (respectively):

```
number_of_epochs = 10
number_of_epochs = 1
```

The other parameters are the same as in the previous testing, so we are only interested in looking at the results with respect to the *number_of_epochs*. Again, we are looking at **test 1** in the **test_project_2.py** file.

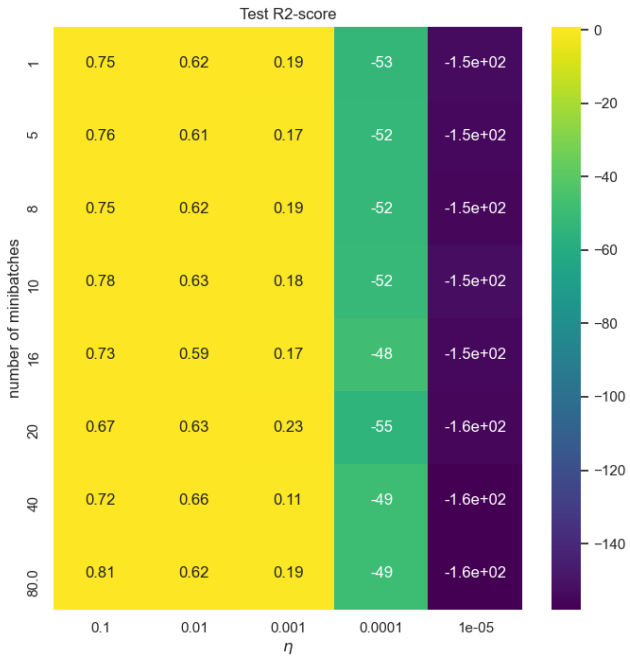

Figure 3: Same as figure 2, but *number_of_epochs* = 10

By looking at figure 2, 3 and 4 we can confirm that higher number of iterations in the SGD-algorithm will lead to a better result. Then a question arises, why do we not increase the number of epochs to infinity? The answer is simple, the computing time will be increased as the number of epochs increases. Therefore, we need to restrict the number of epochs for us to be able to move forward with the analysis. So, in later analysis we have not optimized the number of epochs since it is not optimizable. Therefore, we have often picked the parameter so

the runtime is not too large, but also enough iterations for a model with good results.

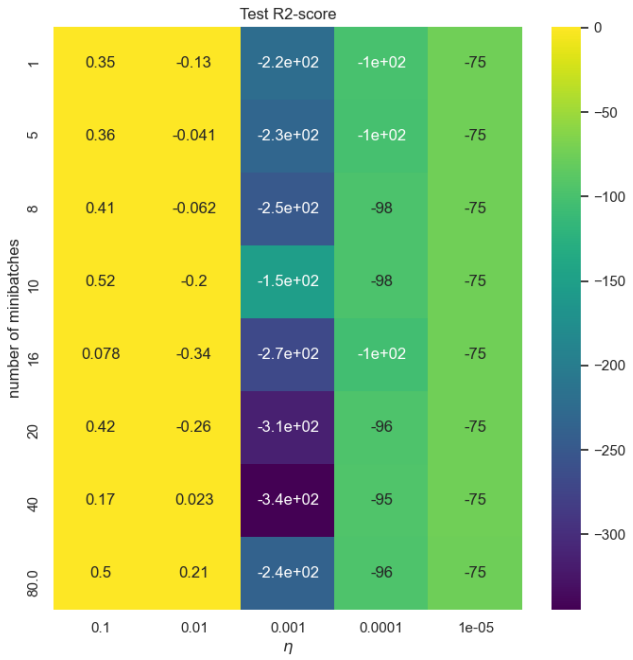

Figure 4: Same as figure 2, but $number\_of\_epochs = 1$

4.2.1.1 Scaling the learning rate We want to study an algorithm for scaling the learning rate. We have tried the following algorithm:

$$f(t) = \frac{5}{t + 50} \tag{16}$$

This algorithm scales the learning rate to be smaller over time. We compare the results with and without the algorithm, where we run five tests both with and without the scaling algorithm. You can find this test by navigate in the **test_project_2.py** file to **test 13**. We use these values for testing:

```
n_epochs = 200
gamma = 0
eta = 0.1
batch_size = 10
```

This would give out a result in this form:

```
With Scaling learning rate algorithm
Test 1: MSE: 0.189056
Test 2: MSE: 0.189427
Test 3: MSE: 0.189085
Test 4: MSE: 0.188920
Test 5: MSE: 0.188831
Without Scaling learning rate algorithm
Test 1: MSE: 0.184023
Test 2: MSE: 0.183986
Test 3: MSE: 0.184069
Test 4: MSE: 0.183949
Test 5: MSE: 0.184023
```

Since we do not observed an improved MSE, we do not use the scaling algorithm for the rest of the project as this saves time and may also lead to better results. We already have a momentum based SGD, so perhaps the scaling algorithm is somewhat redundant.

4.2.2 RIDGE - results of Stochastic Gradient Descent

We make a graph comparing the R2-score of our model as a function of learning rate and hyper-parameter λ for Ridge regression.

We observe that the results are sensitive to the learning rate η, based on figure 5 (showing the result of the testing data). A too large learning rate may cause the gradient descent to miss the global minimum we are trying to find, while a too small learning rate may not converge quickly enough to the minimum. When comparing different learning rates to each other, we have used static learning rates to better see the difference in results. We also see that a too high hyper-parameter λ will unnecessarily punish good fits while too low λ will fail to punish overfitting. The optimal choice seems to be $\lambda = 10^{-5}, \eta = 1$. Note that when attempting to use even higher learning rates, the program crashed as our parameters diverged. We can clearly see that the accuracy of the model is more sensitive to the learning rate η than the hyper-parameter λ. This indicates that we should be careful when choosing the learning rate, as it may greatly change the resulting model.

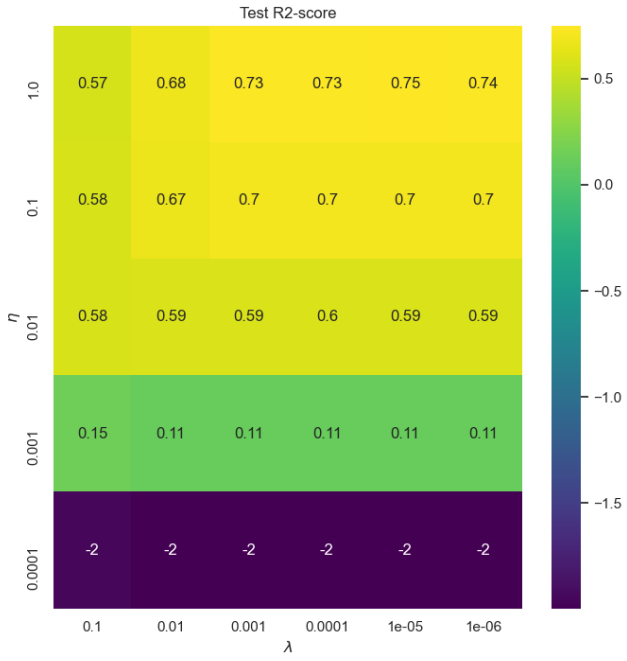

Figure 5: Plot of the R2-score of the model as function of learning rate η and hyper-parameter λ. We have used a degree of 4 for the polynomials, $batch\_size = 10$, $number\_of\_epochs = 200$

To be able to reproduce figure 5 and the analysis of the batch size below, you need to insert the following values to **test 1** in the test file.

```
n = 500
number_of_epochs = 200
no_of_minibatches = 10
degree = 4
gamma = 0.7
run_main_RIDGE = True
```

Then, we find the best mini-batch size. We run a test comparing the MSE for different sizes:

```
MSE: 0.232627 (NUMERICAL (batch size = 1))
MSE: 0.232035 (NUMERICAL (batch size = 5))
MSE: 0.231755 (NUMERICAL (batch size = 10))
MSE: 0.231771 (NUMERICAL (batch size = 30))
MSE: 0.231771 (NUMERICAL (batch size = 60))
MSE: 0.231758 (NUMERICAL (batch size = 100))
```

We notice that a batch size of 10 gives the lowest MSE, hence this is the optimal choice for the ridge regression. Since there are 80 data points for training (100 in total), a batch size of 10 gives out 8 mini batches.

4.2.3 Neural network

We will use our neural network to predict the Franke function as in project 1. How we extracted our data is described in section 4.1.2. To make the model work as correct as possible, we have to choose the correct parameters. With the Franke function data set, we will not set any activation function for the output layer. That is because our neural network is able to handle all target values, not just values between 0 and 1. We will start the analysis by studying the Sigmoid as an activation function for the hidden layers.

First, we need to find the optimal architecture of the neural network

We are finding the optimal architecture by looking at a heatmap with the number of layers and nodes at the x- and y-axis (respectively). To be able to create such a plot, we need to initialize some parameters. We started by setting the parameters in the following way:

```
n_epochs = 300
batch_size = 10
gamma = 0
lmbda = 0.0001
eta = 0.001
```

With these values, we received the following plots as shown in figure 6 and 7, by using **test 9** in **test_project_2.py**. The plot tells us how we should build up our neural network by picking some architecture, a combination of the amount of nodes and layers, that gives us the highest R2-score.

By looking at the figures 6 and 7, we can observe that the amount of nodes needs to be (much) greater than the number of hidden layers for the neural network

to be good as a model. We can also see that among the cells with the highest R2-score, there will be some "bad" ones also. Those cells (with R2-score around 0), have (probably) gone to a local minimum - where the model is predicting the same target values for all kind of input data (as mention in section4.1.4). We tried to find a region where the architecture was kind of stable and worked good as a model, those values were 40 hidden nodes and 3 hidden layers. So, from now on we are going to evaluate the remaining parameters given this architecture.

A too high complexity of the model will be more computationally expensive, and may also result in an overfit, while a too simple model seems to give an underfit.

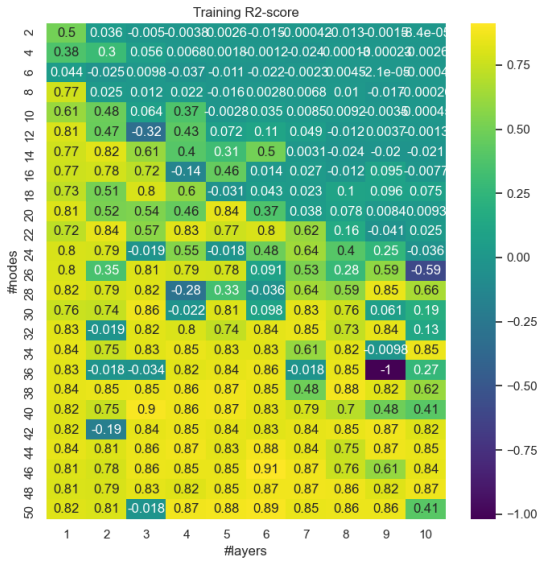

Figure 6: Data: training, R2-score of different architecture of the neural network

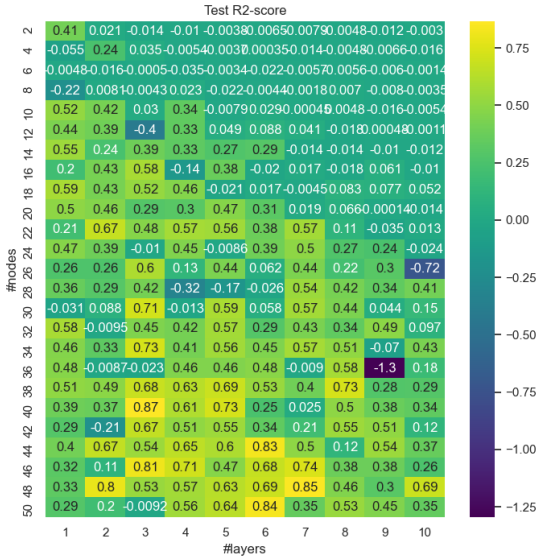

Figure 7: Data: testing, R2-score of different architecture of the neural network

Secondly, we need to find the optimal batch size and momentum parameter

Now we want to find the optimal batch size and momentum parameter by looking at a heatmap with the momentum parameter and batch size at the x- and y-axis (respectively). To be able to create such a plot, we need to initialize some parameters (again). Now, we have optimized the architecture of the neural network (by last interpretations), and will be using those as initial values. So, the parameters we now are producing plots with are:

```
node_list = [40]*3
n_epochs = 300
lmbda = 0.0001
eta = 0.001
```

If we now insert those parameter values inside **test 10** in **test_project_2.py**, then we will get two plots (figure 8 and 9). By looking at the plots, it seems like the most "stable" values (the models that are not predicting the same output) lies in the middle of the plot. We can see the same trend in the R2-score for both training and testing data. So, we have chosen a batch size of 16, and a gamma value of 0.4.

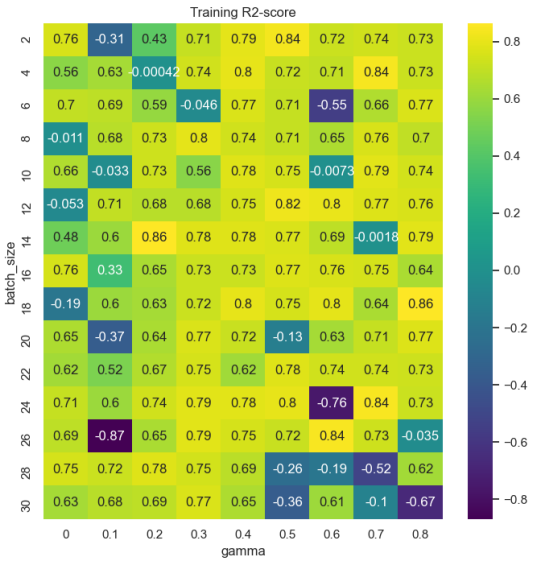

Figure 8: Data: training, R2-score of different batch size and gamma values

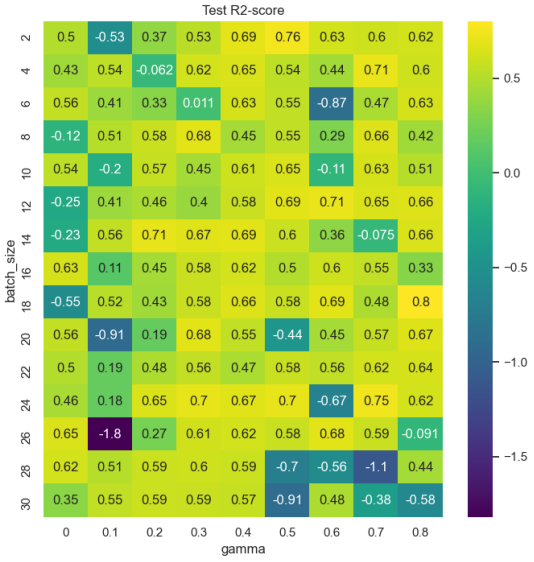

Figure 9: Data: testing, R2-score of different batch size and gamma values

Last, we need to find the optimal hyperparameter lambda and learning rate

Now we want to find the optimal hyperparameter lambda and learning rate by making a grid search. Now, by the parameters we found for the batch size, momentum (in the SGD) and the architecture of the neural network, we are able to use those values for finding the optimal lambda and eta values. We will try to find the parameters, η and λ, with the greatest R2-score. We are setting the parameters in the following way:

```
node_list = [40]*3
n_epochs = 2000
batch_size = 16
gamma = 0.4
```

If we now insert those parameter values to **test 4** in **test_project_2.py**, then we will get two plots (figure 10 and 11). The figures tells us that the best learning rate, η, is around 10^{-3} and 10^{-4} and the most stable lambdas are between 10^{-3} and 10^{-6}.

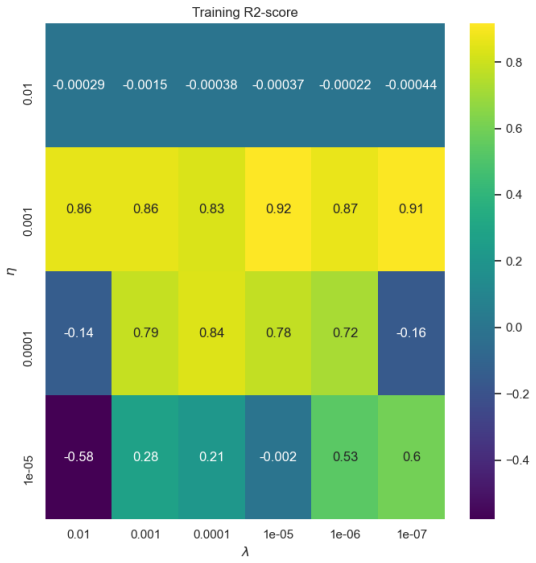

Figure 10: data: training, R2-score of different lambda and eta values

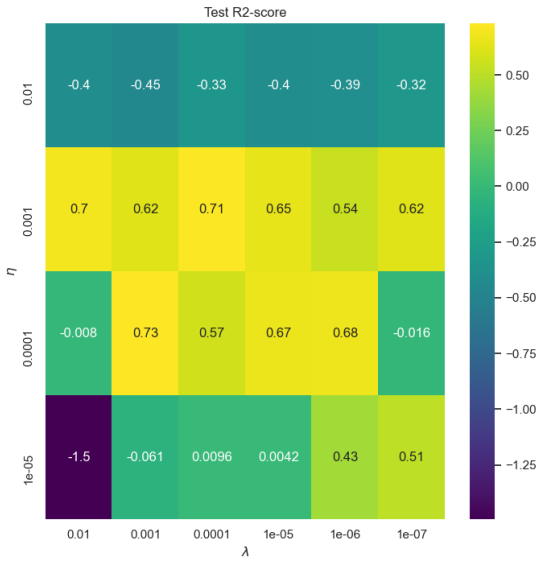

Figure 11: data: training, R2-score of different lambda and eta values

We can see at fig. 10 and 11 that for a greater λ value, the model will not be as overfitted as it would be without such a parameter. Also if the learning rate exceeds 10^{-3}, we observe occurrences of exploding- or vanishing gradients, which leads to a poor model. With a too low learning rate, we observe that in some cases the training have gone too slow to reach some well tuned parameters. This could be fixed by setting up the number of epochs, but this would require more computing power which is something we want to avoid if possible. We conclude from this that the optimal parameters are: $\eta = 10^{-3}$, $\lambda = 10^{-4}$, $\gamma = 0.4$, batch size $= 14$, hidden layers $= 4$, nodes in hidden layers $= 40$.

4.2.3.1 Different activation functions So, the fully analysis we have done so far was with the Sigmoid as the activation function for the hidden layer. Now, we are ready for looking at some other activation functions. We have gone into details in the Sigmoid case, and will just briefly comment the optimization of the coming activation functions.

4.2.3.1.1 the RELU To study the RELU function, we find all other parameters again. We find the optimal layers, nodes, γ, batch size, η and λ again:

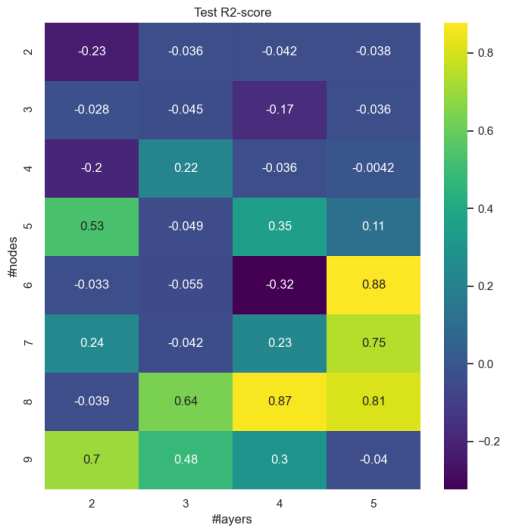

Figure 12: Studying the R2-score (testing data) with the RELU activation function for different layers and number of nodes

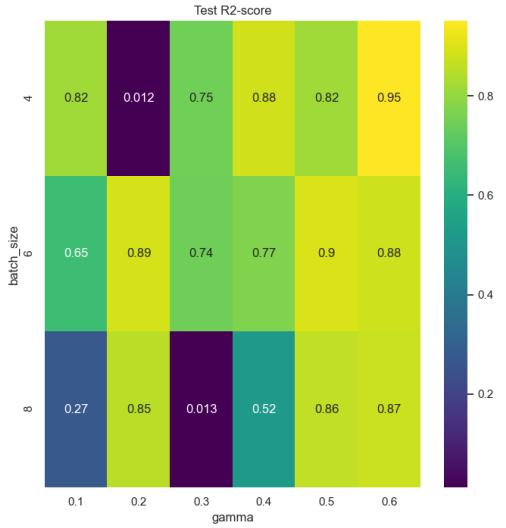

Figure 13: Studying the R2-score (testing data) with the RELU activation function for different batch sizes and γ values

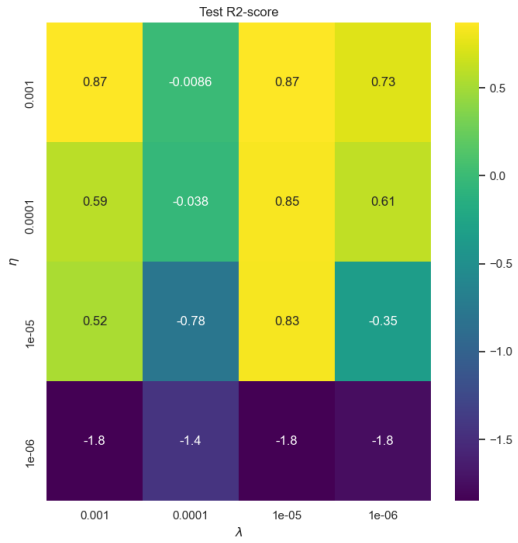

Figure 14: Studying the R2-score (testing data) with the RELU activation function for different learning rates and λ

We note that when the learning rate was higher than 10^{-3}, the code did not run, presumptuously due to exploding gradients. It appears that the optimal parameters for the RELU activation function are $\eta = 10^{-3}, \lambda = 10^{-5}, \gamma = 0.5$, batch size $= 6$, number of layers $= 4$, nodes per layer $= 8$.

According to fig. 17, we achieve an R2-score of 0.87 for the testing data. Compared to the same tests for the Sigmoid function, the RELU activation function does not seem to tolerate many hidden layers and many nodes, as this may cause gradients to vanish or explode. We sometimes get *Runtime Warnings* for underflow from python which indicated that gradients are vanishing, which can be expected of the RELU function as the gradient is zero for certain input values. We get a slightly higher R2-score with the RELU than the Sigmoid, meaning RELU may be the better choice for the data we are working with.

4.2.3.1.2 Leaky RELU We will again find all the optimal parameters, now by using the leaky RELU activation function for the hidden layers:

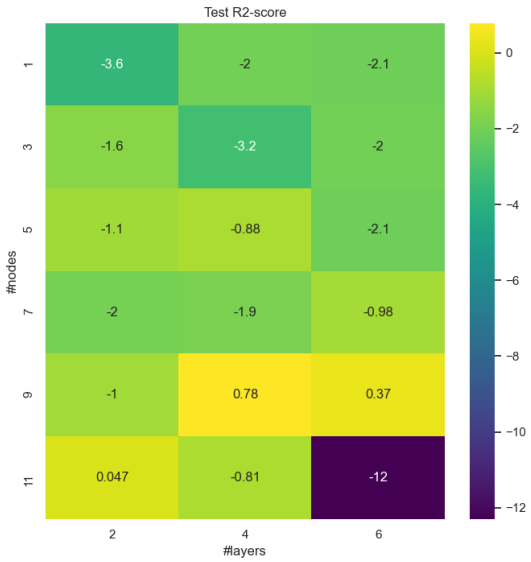

Figure 15: Studying the R2-score (testing data) with the Leaky RELU activation function for different amount of layers and nodes

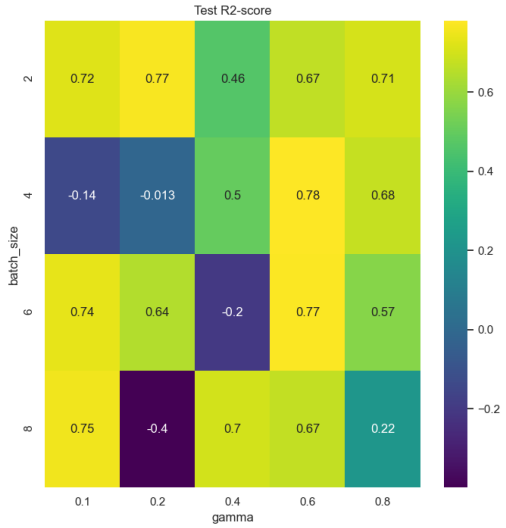

Figure 16: Studying the R2-score (testing data) with the Leaky RELU activation function for different γ values and batch sizes

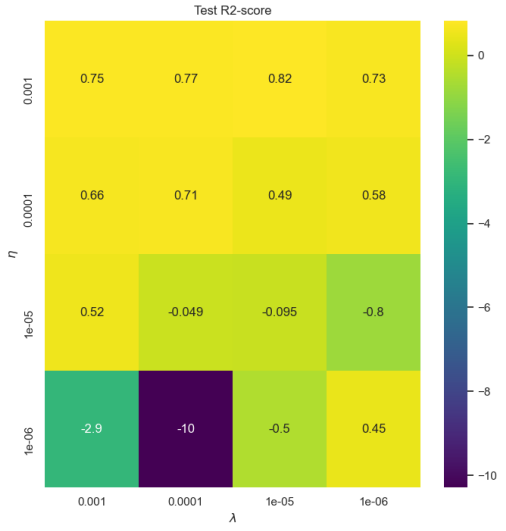

Figure 17: Studying the R2-score (testing data) with the Leaky RELU activation function for different learning rates η and λ values

We conclude that the following parameters seem optimal for the Leaky RELU activation function: $\eta = 10^{-3}, \lambda = 10^{-5}, \gamma = 0.6$, batch size = 5, number of hidden layers = 4, nodes per layer = 9. For these parameters we achieve an R2-score of 0.82 for the test-data, slightly less than for the RELU activation function.

The results look similar to the ones for the RELU function, but the leaky RELU seems more stable, as the program seems more stable and we do not get *Runtime Warnings* from python because of disappearing gradients.

4.2.3.1.3 Evaluation We observe that all the activation functions achieve roughly the same R2 scores. The computing time is also similar. Meanwhile, one has to be more careful when choosing parameters for the RELU function(s), as the gradients can more easily vanish which causes a terrible model.

4.2.4 Comparison: Neural network and linear regression

We compare out neural network to our code from project 1, i.e. the normal linear regression with an analytical solution. We are still using the data from the Franke function. We compare both R2-scores and the running time of each method. We run the test with the optimal parameters for the case with activation function Sigmoid, so the parameters from section 4.2.3. We can reproduce the results by running **test 3** in our test file:

```
> Neural Network (time spent: 12.7519s):
** (R2) TRAINING DATA: 0.822
** (R2)  TESTING DATA: 0.7969

> OLS (time spent: 0.0005s):
** (R2) TRAINING DATA: 0.9724
** (R2)  TESTING DATA: 0.9636

> RIDGE (time spent: 0.0003s):
** (R2) TRAINING DATA: 0.9201
** (R2)  TESTING DATA: 0.8661
```

We observe that in this case, our neural network does not only achieve a lower R2-score than both the OLS and Ridge regression methods, it is also much slower. The reason for this is probably that we are only using 2 input features (as in project 1), and the matrix inversion performed in OLS and Ridge regression does not require heavy computation. If we instead had looked at a data set with hundreds of input features, the neural network would most likely be a better model as it does not suffer as badly from the curse of dimensionality as matrix inversion.

4.3 Classification problem

The data set we are using in the classification problem is described in section 4.1.3.

4.3.1 Neural network

We will use our neural network to predict 0 and 1's of the *Wisconsin Breast Cancer* data set. Since we are looking at targets with 0 or 1, it will be a much better idea to choose some kind of an activation function for the output layer too (as well for the hidden layers). In the coming analysis, we have chosen to start looking at the activation function Sigmoid for both hidden- and output layer(s). To make a great model, we need to find the greatest hyperparameters.

First, we need to find the optimal architecture of the neural network

We are finding the optimal architecture by making a grid search with the number of layers and nodes at the x- and y-axis (respectively). To make a good model, we need to find the greatest hyperparameters. We will start out by searching for a great architecture with these parameters:

```
n_epochs = 300
batch_size = 10
gamma = 0
lmbda = 0.0001
eta = 0.001
```

With these values, we received the following plots as shown in figure 18 and 19, by using **test 11** in **test_project_2.py**. The plots tells us how we should build up our neural network by picking some architecture, a combination of the amount of nodes and layers, that gives us the highest accuracy score.

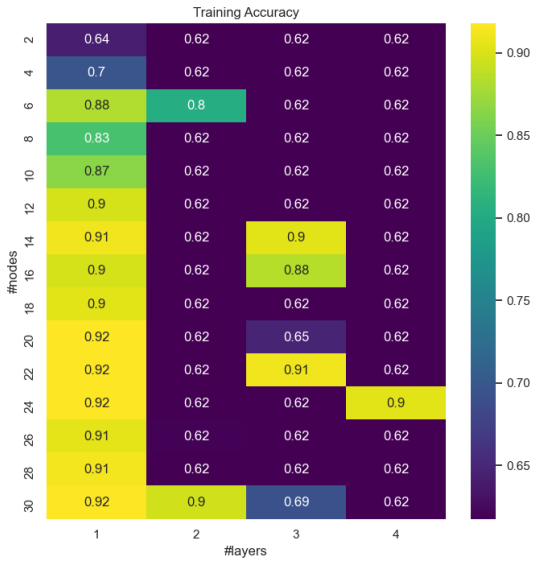

Figure 18: Data: training, accuracy of different architecture of the neural network

By looking at the figures 18 and 19, we do observe some similarities with the regression case, that the number of nodes need to be much better than the number of hidden layers for the model to predict well. We will again find some architecture that is kind of stable, in the sense of slightly different parameters will not predict much worse, but also try to find the combination of nodes and layers that will achieve the highest accuracy score.

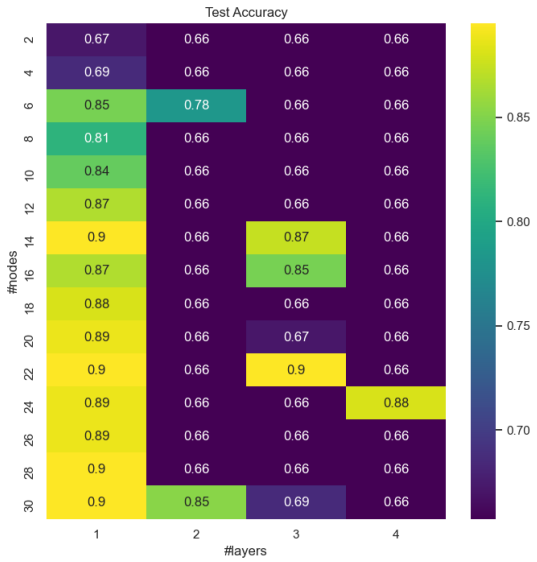

Figure 19: Data: testing, accuracy of different architecture of the neural network

Fig. 18 and 19 are telling us that the number of nodes and layers should be about 20 and 1 (respectively). So, we are going to use those values in the further analysis of the other parameters we want to tune. We pick 20 as the number of hidden nodes because the lower amount of nodes, the more *computing friendly* it gets to train the model.

Secondly, we need to find the optimal batch size and momentum parameter

Now we want to find the optimal batch size and momentum parameter by making a grid search with the momentum parameter and batch size at the x- and y-axis (respectively). To be able to create such a plot, we need to initialize some parameters (again). Now, we have optimized the architecture of the neural network (by last interpretations), and will be updating those in the choice of parameters. So, the parameters we now have are:

```
node_list = [20]*1
n_epochs = 300
lmbda = 0.0001
eta = 0.001
```

If we now insert those parameter values inside **test 12** in **test_project_2.py**, then we will get two plots (figure 20 and 21). By looking at the plots, it seems like the best parameters is to set a gamma value of 0.9, and it seems like it doesn't depend that much on the batch size, so we will in this case choose 14. It can often be a great risk of choosing such a high momentum parameter, gamma, since it can easily fail to converge. Since the model predicts best with such values, we will go for it anyways.

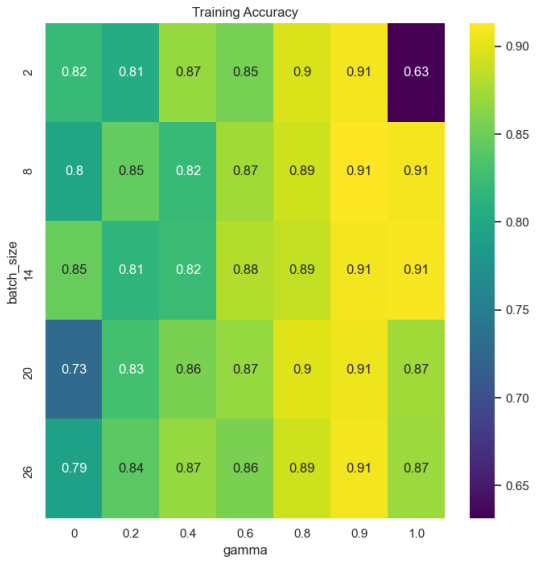

Figure 20: Data: training, accuracy of different batch size and gamma values

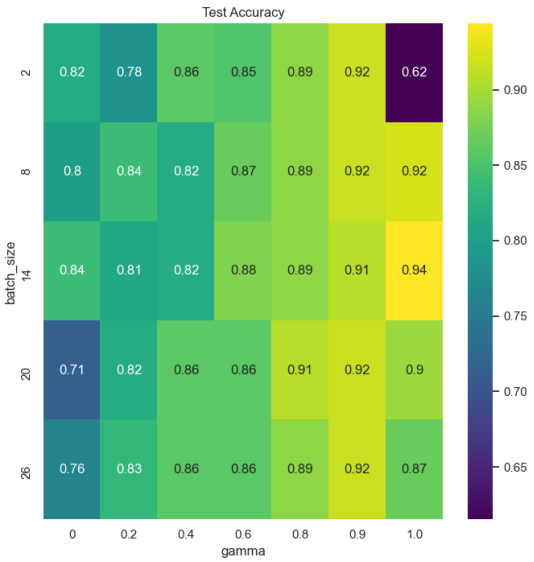

Figure 21: Data: testing, accuracy of different batch size and gamma values

Last, we need to find the optimal hyperparameter lambda and learning rate

Now, we want to find the optimal hyperparameter lambda and learning rate by making a grid search. Now, with the parameters we found for the batch size, momentum (for the SGD) and the architecture of the neural network, we are able to find the optimal lambda and eta values. We will try to find those parameters that will get the highest accuracy score. We are setting the parameters in the following way:

```
node_list = [20]*1
n_epochs =  200
batch_size = 14
gamma = 0.9
```

If we now insert those parameter values into **test 6** in **test_project_2.py**, we will get two plots (figure 22 and 23).

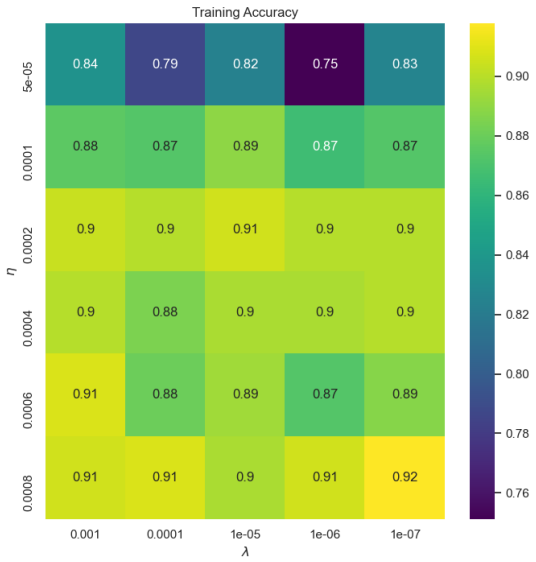

Figure 22: data: training, accuracy of different lambda and eta values

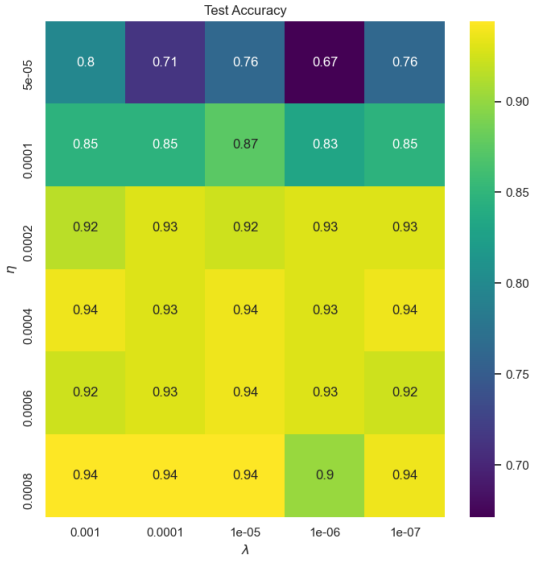

Figure 23: data: testing, accuracy of different lambda and eta values

Figure 22 and 23 tells us that a learning rate up towards 10^{-3} is a good value, but the risk of getting exploding gradients will be higher with a higher learning rate. Therefore, we found it necessary to restrict the eta values that we are looping over (more restrict than before). The learning rate is so sensitive because the momentum parameter (gamma) is at 0.9, which means that is will use 90% of the last gradient value into the new iteration. The figures shows us that it doesn't depend that much on the parameter lambda, most likely because we have chosen a simple architecture which is not likely to lead to an overfit. Although, we can still see some trends for lambdas work as an *anti-overfitting* parameter.

4.3.1.1 Different activation functions Since we are dealing with a binary classification case, the Sigmoid function is obviously the best choice for the output layer. For the hidden layers, we study how the Sigmoid function compares to the RELU and leaky RELU. First, we plot the accuracy score for the training data against iterations, with the Sigmoid activation function for the hidden layers. The following plots (fig. 24, 25, 26) are achieved by running **Test 5** in the **test_project_2.py** - file, with the same parameters that we have been chosen as described in the previous subsection, that is:

```
node_list = [20]*1
number_of_epohcs = 200
batch_size = 14
gamma = 0.9
eta=8e-4
lmbda=1e-5
```

and changing the hidden activation function between (respectively):

```
FFNN.set_activation_function_hidden_layers('sigmoid')
FFNN.set_activation_function_hidden_layers('RELU')
FFNN.set_activation_function_hidden_layers('Leaky_RELU')
```

Figure 24: Accuracy of training data over iterations with the Sigmoid function for the hidden layers

We create the same plot for the RELU function for the hidden layers. We have changed the γ parameter from 0.9 to 0.5 as this gives a better result:

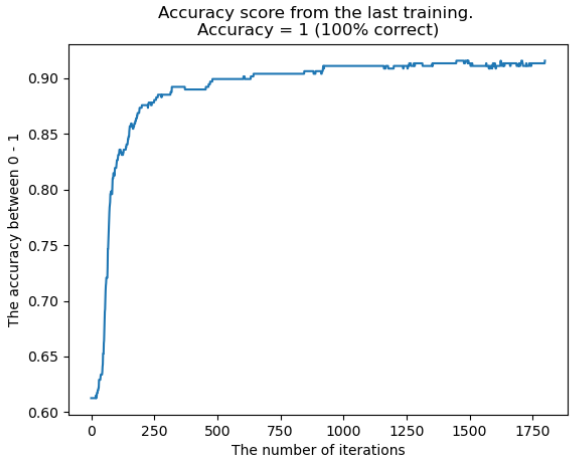

Figure 25: Accuracy of training data over iterations with the RELU function for the hidden layers

Finally we create the same plot for the leaky RELU with the same parameters as for the RELU:

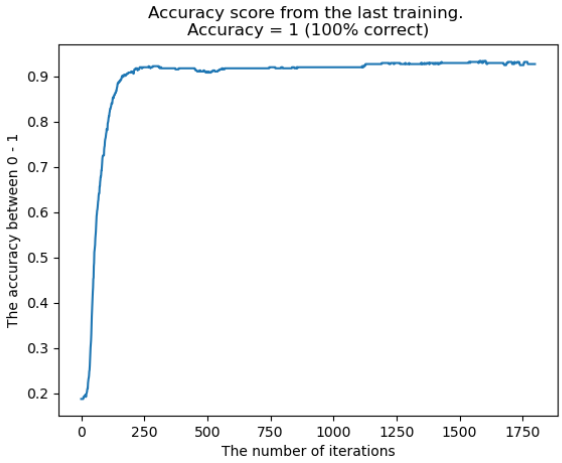

Figure 26: Accuracy of training data over iterations with the leaky RELU function for the hidden layers

We observe that all the activation functions for the hidden layers give an accuracy score of above 90%, yet the leaky RELU function seems to coverge to an accurate model in fewer iterations than both the Sigmoid and the normal RELU, hence it is the best choice for this model.

4.3.2 Logistic regression

Now, we want to look at how well the logistic regression works for predicting the same data as we have studied in the last section (the *Wisconsin Breast Cancer* data set 4.1.3). We want to find the optimal values for the learning rate and the hyperparameter λ. We will do that by compare the different accuracy score the model will give us with the different parameters. To do so, we need to start with some specified parameters, which are:

```
n_epochs = 40
batch_size = 14
gamma = 0.8
```

With these values, we make a heat map for different η and λ's to find the optimal choice. This can be reproduced by **test 7** in the testing file.

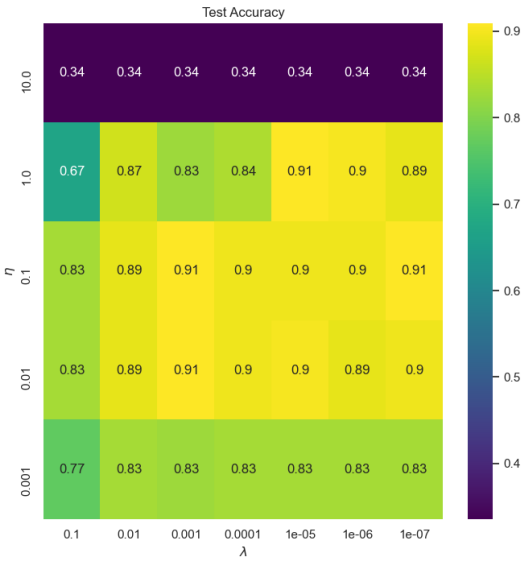

Figure 27: Accuracy of different η and λ values for the testing data

In figure 27, we observe that the optimal parameters are around $\eta = 0.1, \lambda = 10^{-3}$. These are the parameters we will be using when comparing with the neural network.

4.3.3 Comparison: Neural network and logistic regression

The values we are gonna use for the comparison between the different methods for modelling the classification case are the optimal ones from the last two sections, which are:

```
# Values for neural network
n_epochs = 4
batch_size = 14
gamma = 0.5
eta = 0.001
lmbda = 1e-5
FFNN.set_cost_function(logistic_cost_NN)
FFNN.set_activation_function_hidden_layers('RELU')
FFNN.set_activation_function_output_layer('sigmoid')

# Values for logistic regression
n_epochs = 3
cost_function = cost_logistic_regression
batch_size = 14
gamma = 0.8
eta = 0.1
lmbda = 1e-3
```

The results we get in the terminal by running **test 8** in the **test_project_2.py** file are as follows:

```
>> RUNNING TEST 8:
>>> Results of classification problem with Neural Network <<<
ACCURACY_train =  0.9108
ACCURACY_test =  0.9161
TIME SPENT:  0.069

>>> Results of logistic regression, with SGD <<<
ACCURACY_train =>  0.9178
ACCURACY_test =>  0.9161
TIME SPENT:  0.134
```

We have done some testing to find two great models with respect to the number of epochs, because we wanted to find models that predict well and train quickly. We can see that for similar accuracy, in both the training- and test data, the time spent training the neural network is half of the logistic regression case.

5 Conclusion

We started by looking at the Stochastic Gradient Descent algorithm (SGD). Regarding linear regression, gradient descent is an alternative to a matrix inversion, since we often use matrix inversion as a calculation for getting the analytical solution to the problem. The matrix inversion is time consuming, and therefore we introduced the gradient descent algorithm to train some model. The stochastic in the gradient descent is also a further improvement to the algorithm, where we consider a smaller amount of data to train the model with, which would make the process even faster. Now, we have a method for tuning some of the parameters of the model, but the hyperparameters sent in to the training must be optimized in some way for the model to be a good one.

One of the first thing we found out was that the learning rate and gamma had a huge impact on the results we were going to achieve. A too high learning rate would often result in a bad prediction or divergence with respect to the tuned parameters, and a too small learning rate would converge slowly (if it even converged to the *right* solution). The combination between the learning rate and the momentum parameter was critical to success, where the momentum helped smaller learning rates to converge faster. We sometimes found that out model was unstable, meaning it did not get consistently good results. In such a case, we needed to choose different parameters. Finally, we figured out that in most cases the number of iterations was highly correlated with the prediction we got. The higher number of epochs, iterations, the better model.

The first thing we noticed when testing our neural network code was that for it to work properly, there are a lot of parameters to be tuned, everything from architecture, learning rate, hyper-parameters to activation functions. We have learned that different data requires different parameters. We have also learned that it is not always given a priori how a change in a parameter affects the result, this information must often be obtained through tests. We have also realized how small changes in parameters may cause the gradient descent to diverge, and one must be especially careful when choosing the learning rate, as mentioned. We have also realized that the quality of the neural network may also be limited by the accessible computing power. Training of large architectures requires large computations, and this may be limited if one is using for instance a laptop. Different activation functions also give different results, and some can more easily cause vanishing or exploding gradients than others, especially the RELU function.

When comparing our neural network to the linear regression code from project 1, we noticed that the neural network was far slower in achieving about the same results as the linear regression. From this we conclude that when there exist machine learning methods that are less complex and time consuming than the neural network which, it might be the better option. Let it be noted that for the linear regression case, we only used two input features and restricted

our data set to 100 data points. Had we used more features or more data, the linear regression code may not have been as fast anymore. We also note that we could have increased the complexity of the neural network by changing it's architecture, but our code already required heavy computation, and we did not find it necessary to run tests that would have taken even longer time.

When comparing the neural network to the logistic regression in the classification case, both methods achieved a high accuracy score, but the neural network did it faster. In this case, both methods required gradient descent, and the neural network trained faster than the logistic regression model. We only studied one data set, so it is hard to tell if the neural network would have been faster for another set of data.

Our time with this project was limited, so it was necessary to restrict the project in some way. Further analysis we would probably have done would be to look at more types of data set. In this case, we have only looked at a binary classification problem (the target value in the classification case is just 0 or 1). The input data, in both the regression- and classification case, had just two input features, since we restricted the data set as explained in section 4.1.3. With other data sets, we would maybe see some new insights and get other results from our analysis than we got in this project. The last thing we would have liked to look into is various number of hidden nodes in the different hidden layers and see if there is some more insights there. The neural network class is ready for handling such analysis, but we did not find it necessary in this project.

6 Github repository

Github REPO - project 2

7 Bibliography

[1] Morten Hjort-Jensen. *Data Analysis and Machine Learning: Logistic Regression*. URL: https://www.uio.no/studier/emner/matnat/fys/FYS-STK4155/h21/forelesningsvideoer/LectureSeptember23.mp4?vrtx=view-as-webpage.

[2] Morten Hjort-Jensen. *Week 40: From Stochastic Gradient Descent to Neural networks*. URL: https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html.