UiO : Universitetet i Oslo

Project 2 - Classification and Regression

FYS/STK4155

Andreas Dyve
Jørn-Marcus Høylo-Rosenberg

University of Oslo
November 20, 2021

Abstract

In this project we aim to investigate how a neural network compares to linear and logistic regression of Franke's function and the Wisconsin breast-cancer data respectfully. Our findings affirm that analytical least squares provide the best solution for regression, while the added complexity and flexibility of a neural network allow us to slightly improve upon the results of logistic regression. Moreover we study the effect of different activation functions in a neural network, and discover that the RELU class of functions are superior for regression, but does not improve over the logistic function in our binary classification example.

Contents

1 Introduction

Neural networks have received an enormous amount of attention in the world of machine learning in recent years. Improvements in the field has made it possible to solve a wide range of problems that computers has not been able to do earlier. We all know about the different AIs beating world-class players in various games, but neural networks are becoming increasingly important in many more areas because of its high flexibility and problem solving ability, such as autonomous driving, stock market trading and medical diagnosis. The other side of the coin when it comes to the flexibility is that it gives rise to many hyper-parameters, which all have to be tuned for each specific problem. This can often be difficult, and require a thorough understanding of how the network works.

In this project, we will build a feed-forward neural network (FFNN) and explore its applications in both linear regression and classification problems. The regression problem we aim at solving is the Franke's function, a commonly used example for polynomial fitting. Building our way up to the neural network, we will start by solving the ordinary least squares and ridge regression analytically to find the optimal parameters. Then, the analytical solution is replaced by a stochastic gradient descent algorithm, which is the core minimization algorithm in a neural network, used to update the weights and biases that in turn determine the outputs. With the insight hopefully gained by tuning the parameters in stochastic gradient descent, our neural network is built and used to predict the Franke's function.

Next, we want to use our neural network for classification. The problem at hand is the Wisconsin breast cancer data, where features from images are used to predict if tumors are benign or malignant. Once again, the parameters of the neural network are tuned to and the predictions are made.

Finally, we provide a comprehensive discussion of our results, looking into the different algorithms, network architecture and hyper-parameters used before some concluding remarks at the end.

2 Theory

2.1 Stochastic Gradient Descent

When solving a problem with machine learning, the goal is to minimise the cost function. A common way to find the minimum numerically is gradient descent. The basic idea is that the cost function decreases most rapidly in the direction of the negative gradient.

For a function $F(x)$, it can be shown that if

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

, with $\gamma_k > 0$, then $\mathbf{F}(\mathbf{x}_{k+1}) < \mathbf{F}(\mathbf{x}_k)$, then we always move towards smaller function values. It is normal to start with an initial guess x_0 and iterate towards a minimum. The parameter γ is called the step length or learning rate.

Since cost functions usually are complicated and have a lot of curvature, regular GD has a tendency to get stuck in local minima or saddle points. Stochastic gradient descent and variants of this address some of the shortcomings of normal GD. The basic idea is that the cost function can be written as a sum of n data points,

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x_i}, \beta) \tag{2}$$

and thus, the gradient is the sum over i data points

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x_i}, \beta) \tag{3}$$

We introduce randomness/stochasticity by dividing the data into intervals, called mini-batches. With the size of each mini-batch being M and number of data points n, the number of mini batches is n/M, denoted B_k, $k = 1, ..., \frac{n}{M}$. The gradient can now be approximated by replacing the sum over n data points with the sum of data points in one mini-batch picked at random for each iteration of gradient descent.

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x_i}, \beta) \rightarrow \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x_i}, \beta) \tag{4}$$

A gradient step is now calculated by

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

Where k is picked at random between 1 and n/M. One iteration over the number of mini-batches is commonly referred to as an epoch.
Taking the sum over randomly chosen mini-batches has two main benefits. One, the randomness decreases the chance of getting stuck in local minima. Second, if the size of each mini-batch is chosen to be much less than the number of data points, $M << n$, then the computation of the gradient becomes much cheaper. Too small mini-batch size will, however, lead to inaccurate gradient calculations, so this is an important tuning parameter in the model.

The learning rate, γ, decides how long each step in the direction of the gradient is taken. This is a complicated parameter to optimize, and that is why we have different algorithms to change and scale it. The simplest one is to have it as a constant. If the right value is chosen, this can in simple problems be good enough to reach a minimum after a certain amount of epochs. However, a constant learning rate can

often result in too slow convergence or diverging results. If the learning rate is set too low, the steps towards the minimum will be too small to reach the minimum for a set number of epochs. On the contrary, if it set too high, the minimum might be skipped, resulting in no convergence at all. Ideally, we want to take larger steps in the beginning, and then decrease the steps as we move closer to the minimum. One way to achieve this is to introduce a learning schedule. This can also be implemented in a variety of ways, but in this project we have chosen to use the time based decay method. It's purpose is to decrease as the number of epochs increases, and the amount that it decreases is determined by a hyperparameter. The formulation is as follows

$$\gamma_t = \frac{\gamma_{t-1}}{1 + k \cdot \text{epoch}_t} \tag{6}$$

Where k is the decay hyperparameter. From this, we can see that if k=0, then the learning rate remains constant. Increasing k rapidly increases the learning rate when the number of epochs becomes large.

2.1.1 Momentum based SGD

SGD can be extended to include a momentum term which serves as a memory of the direction the gradient is moving. It is implemented as follows

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta E(\theta_t) \tag{7}$$

Notice that the terminology is now slightly changed. η_t is now the learning rate at time t, $E(\theta_t)$ is the cost function and θ are the parameters we want to optimize. v_t is from this equation the running average of the previous encountered gradients. γ is the momentum hyperparameter. It has a value between 0 and 1 and sets the weight of previously encountered running averages, giving the most recent gradient the highest contribution while earlier gradient's contribution exponentially decays. The component of the gradients which is not in the direction of the minimum tends to cancel each other out and average towards zero, while the component towards the minimum is reinforced, resulting in a greater velocity towards the minimum. The parameters θ are then updated by

$$\theta_t = \theta_{t-1} - v_t \tag{8}$$

In SGD with momentum, the learning rate is still limited by the steepest direction of the cost function, which changes depending on where we are in the landscape. Ideally, we could keep track of the curvature and take large steps where it is flat and smaller steps where it is steeper. One approach is to calculate the hessian for each iteration and adjust the learning rate, but this is computationally expensive. Ideally, we would like a method that can adaptively change the learning rate according to the landscape, without having to calculate or approximate hessians. Some good methods that does this include AdaGrad, AdaDelta, RMS-Prop, and ADAM. In this project, we have used the RMS-Prop method, so this will be explained in the next section.

2.1.2 RMS-prop

Root mean squared propagation, or RMS-Prop, keeps track of the running average over the first moment of the gradient, but it also keeps track of the second moment, denoted by $s_t = \mathbf{E}[g_t^2]$. The RMS-prop is implemented using the following equations

$$g_t = \nabla_\theta E(\theta) \tag{9}$$

$$s_t = \beta s_{t-1} + (1-\beta) g_t^2 \tag{10}$$

$$\Delta\theta = \frac{\eta_t}{\sqrt{s_t + \epsilon}} g_t \tag{11}$$

$$\theta_{t+1} = \theta_t - \Delta\theta \tag{12}$$

, where beta controls the averaging time of the second moment and is typically chosen to be 0.9 and $\epsilon \sim 10^{-8}$ is a small constant to prevent divergences. From the formula, we can see that the learning rate is reduced where the norm of the gradient becomes large. This allows a larger learning rate for flat directions, greatly speeding up the convergence, and slower learning rate for steep directions, preventing exploding gradients.

2.2 Neural-Networks

The theoretical aspects covered in this section aims to introduce the central concepts in a neural network, and the numerically implementation. The theory is mostly gathered from the lecture material provided in fys-stk4155 [13] [14], Goodfellow's book on deep learning methods [7], ch.6-7, and Nielsen's book on deep learning, particularly ch 4 and 5 [15].

2.2.1 Background

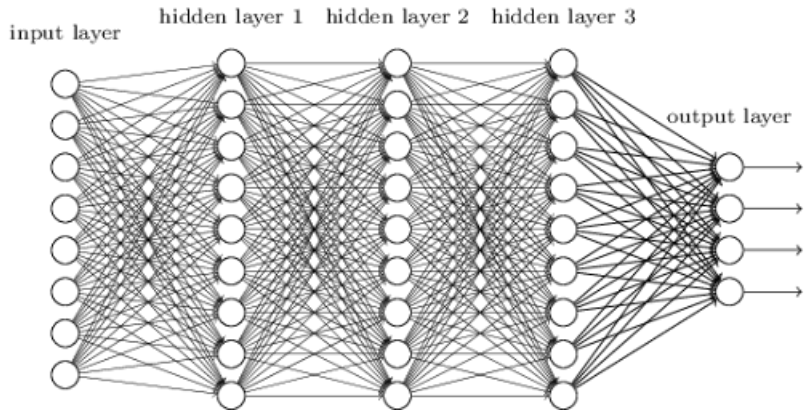


Figure 1: Basic architecture of a neural network. Source: Nielsen, "Neural Networks and Deep Learning." Ch 1: `http://neuralnetworksanddeeplearning.com/images/tikz13.png`

The basic architecture of a neural network (NN) is illustrated in figure 1, the system is comprised of an input layer, a given number of hidden layers, and finally an output layer. Each layer consist of an arbitrary number of neurons or nodes, represented as white circles in the figure. A neural network work similar to biological neurons in the sense that a node/neuron has an activation, and that activation is related to the activation's of other neurons in the network. In an Artificial neural network (ANN), the most common model is the feed-forward network that restrict the flow of data exclusively in the forward direction, meaning that the activation's in layer l is a function of the activation's in the previous layer l-1. Extending this concept further we get the fully connected feed-forward neural network that works by having every node connected by a set of weights, represented as the black lines in figure 1, and biases. In a fully connected feed-forward network, a given node's activation is calculated by the weighted sum of the activations in the preceding layer, as in **??**.

$$w_1 a_1 + w_2 a_2 + w_3 a_3 + ... + w_n a_n \tag{13}$$

The full expression is

$$a = f\left(\sum_i^n w_i x_i + b_i\right) \tag{14}$$

, where a is the activation of a node, x_i will then be the output of the neurons in the preceding layer. Its common to abbreviate the contents of f as a quantity z, resulting in the equation $a = f(z)$ w The other terms, b and f, refer to the bias and activation function respectfully. The activation function is used to transform the value of the weighted sum. A very typical choice of activation function is the logistic function, known as the Sigmoid. More specifics on activation functions will be discussed towards the end of this section.

2.2.2 Training the network

One of the biggest benefits of neural networks is the flexibility. By varying the architecture, activation functions, parameters, weights and biases, a network of simply one hidden layer can approximate any multi-dimensional function to a desired accuracy, according to the universal approximation theorem. The question then becomes how to train a neural network to approximate any function?

In brief, the training of a neural networks works by applying gradient descent methods to update the weights and biases of the network. For the network in figure 1, this means tuning over 250 parameters just for the weights alone. This method of tuning the network weights and biases to improve performance, is implemented by a feed-forward pass and a back-propagation algorithm. In conjunction with stochastic gradient descent, the feed-forward and back propagation is conducted on a given number of mini-batches, we say that one iteration over all mini-batches is one epoch.

The feed-forward pass takes fruition from equation **??**, by extending this expression into a general number of hidden layers we get the feed forward algorithm in equation **??**.

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(...f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)...\right) + b_k^2\right) + b_1^3\right] \tag{15}$$

Which in terms summarize the concepts of a feed-forward multi-layer perceptron model, the only independent variable in the system is the input x_n, used to calculate the outputs in the next layer.

To begin the back propagation algorithm, the first step is to define a cost function to evaluate the outputs of a feed-forward pass (activations in the output layer). Look to section (4.2) for a discussion on choice of cost functions. From here, we relate the change in weights and biases to changes in the cost function in terms of a quantity δ^L.

$$\frac{dC(W^L)}{dw_{jk}^L} = \delta_j^L a_k^{L-1} \tag{16}$$

$$\frac{dC}{db_j^L} = \delta_j^L \tag{17}$$

The quantity δ_j^L is defined as

$$\delta_j^L = f'(z^L) \odot \frac{dC}{d(a^L)} \tag{18}$$

, for every node j in the output layer, and can essentially be related to the speed of which a layer learn. The complete derivation of δ^L and back propagation is neglected in this article, please refer to other resources for this, for example [13], or Nielsen's intuitive look on the meaning of δ [15] ch.4. We then back-propagate to the preceding layers to update the weights and biases throughout the network in the following manner described by 3 essential equations.

$$\delta^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \cdot f'(z_j^l) \tag{19}$$

$$w_{jk}^l = w_{jk}^l - \eta \delta_j^l a_k^{l-1} \tag{20}$$

$$b_j^l = b_j^l - \eta \delta_j^l \tag{21}$$

, where η is the learning rate as in the stochastic gradient descent algorithm.

2.2.3 Activation functions

The purpose of an activation function in neural networks is to add non-linearity to the model, thus extending the applicability beyond the scope of regression problems. The historically most significant activation function in deep learning is the sigmoid function, illustrated in figure 2 and expressed as

$$f(x) = \frac{1}{1 + e^{-x}} \tag{22}$$

The Sigmoid work by transforming the input to values in the range [0,1], and found its place in neural nets mostly because of the parallel and inspiration from biological neurons. However, there are several drawbacks with using the Sigmoid for deep learning, most notably the problem of vanishing gradients. For large inputs (positive and negative), the output will saturate at values 1 and 0, with corresponding derivatives close to zero, as seen in figure 2. This in terms result int the gradients $\frac{dC}{dw}$ and $\frac{dC}{db}$ to become very small, from **??**. Secondly, this causes the numerous layers to learn at different speeds, meaning that the layers optimize at different rates, favouring the later layers. [4].

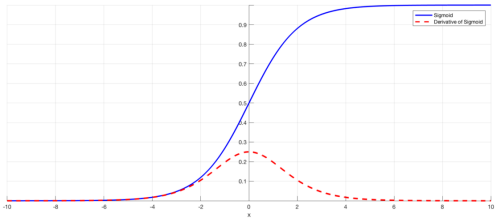


Figure 2: The Sigmoid and the derivative of the Sigmoid. From `https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e`

One of the methods to overcome the drawbacks of the Sigmoid, is simply to substitute the activation function in favor of the RELU class of activation functions. Bellow is a summary of the RELU and leaky-RELU functions.

$$RELU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{23}$$

$$LRELU(x) = \begin{cases} x & x \geq 0 \\ cx & x < 0 \end{cases} \tag{24}$$

The RELU function improves on the Sigmoid by not saturating large values, and fixing the derivative of negative inputs to 0. The downside of this is the so-called dead-node problem where nodes with large negative inputs become inactive due to zero-gradients. This is fixed by functions such as the leaky-RELU or ELU, however in practice the dead-node problem can often be neglected beside in very large networks [4].

(a) RELU (b) leaky-RELU

Figure 3: The RELU and leaky-RELU functions, and their derivatives. Images from: `https://towardsdatascience.com/comparison-of-activation-functions-for-deep-neural-networks-706ac4284c8a`

2.2.4 Weights and biases

The difficulties of the logistic function discussed above also applies to the RELU function, but rather than the gradients vanishing, we might encounter exploding gradients. The reason for this is related to how we initialize the weights of the network. The most straightforward approach is to initialize all weights according to a standard normal distribution, ie a Gaussian distribution with mean 0 and standard deviation 1. In a function such as the Sigmoid, this will result in the variance increasing from layer to layer when input is passed through the network, causing the inputs $(Wa+b)$ to a given node to increase exponentially (negative or positive). To counteract this problem, the method of choice, found originally by Xavier (2010) [6], is to initialize the weights so that the activations have s fixed variance across the network, ie $var(a^{l-1}) = var(a^l)$. This is found true exclusively for

$$var(a^l) = n^{l-1} var(W^l) var(a^{l-1}) \tag{25}$$

Thus the variance of the weights must be set to $1/(n^{l-1})$ in order to refrain from problems such as vanishing or exploding gradients. In conclusion we have the following way of initializing the weights.

$$\mathcal{N}\left(\mu = 0, \sigma^2 = \frac{1}{n^{l-1}}\right) \tag{26}$$

, please refer to [11] for a full derivation for these expressions and statements. One key concept of the Xavier initialization is that it only applies for linear activations such as the Sigmoid or tanh [8]. For RELU activation, the most common way of initializing the weights follows He initialization [8].

$$\mathcal{N}\left(\mu = 0, \sigma^2 = \frac{2}{n^{l-1}}\right) \tag{27}$$

Along with these expressions, its implied that the biases are initialized as zero. Otherwise its also possible to initialize the bias as a small positive number.

We conclude this section on gradient problems by mentioning the term "Unstable gradients" as Nielsen refers to it [15] ch.5. The main takeaway from this chapter is a general difficulty of deep neural networks consisting of several hidden layers, where the fully connected nature between the layers will always lead to different learning speeds throughout the network. Particularly the first hidden layer will always end up lesser optimized than the latter layers. Thus when choosing the architecture of a neural network, this is an important aspect to be aware of.

Finally, we wish to conclude the section by looking at overfitting in neural networks. The two most popular methods to counteract overfitting is either to increase the training data, or to include a weight decay called the regularization term, the most common being L2 regularization. The implementation of a regularization term is rather straight forward, bellow is an example of L2 regularization of the cross-entropy cost function.

$$C = -\frac{1}{n}\sum \left[t_j ln a_j^L + (1 - t_j) ln(1 - a_j^L)\right] + \frac{\lambda}{2n}\sum w^2 \tag{28}$$

The purpose of this term is primarily to reduce the size of the weights, while minimizing the cost function simultaneously. Together with the learning rate η, make up the two most determining parameters to make or break the performance of a neural net.

2.3 Logistic regression

We will in this article skip the theory of logistic regression as this is simply as subclass of an artificial neural network with one hidden node and layer, with a logistic function at a single output node. See [10] for more information on this. The implementation and choice of cost function in the code stems from the lecture material of fys-stk4155 [9].

2.4 Evaluation

To evaluate the performance of the neural network we employ the mean squared error MSE and R2-score for regression problems, and the accuracy score for logistic problems, defined bellow.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 \tag{29}$$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{30}$$

$$Accuracy = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{31}$$

The mean squared error and R2 score are common metrics for evaluating a regression model. Generally the R2 score is considered to be a superior measure to MSE. The accuracy score is quite simplistic, in that the score is calculated by counting the number of correct and incorrect predictions.

3 Data-sets

3.1 Franke's function

The data set used for regression in this project is the Franke's function. It is a two dimensional function, widely used as a test function for interpolation problems. The function is a weighted sum of four exponentials and is implemented as

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \end{aligned}$$

In addition, stochastic noise is added to the function according to the distribution $N(0,1)$. A 3D plot of the function without noise is shown in figure 4.

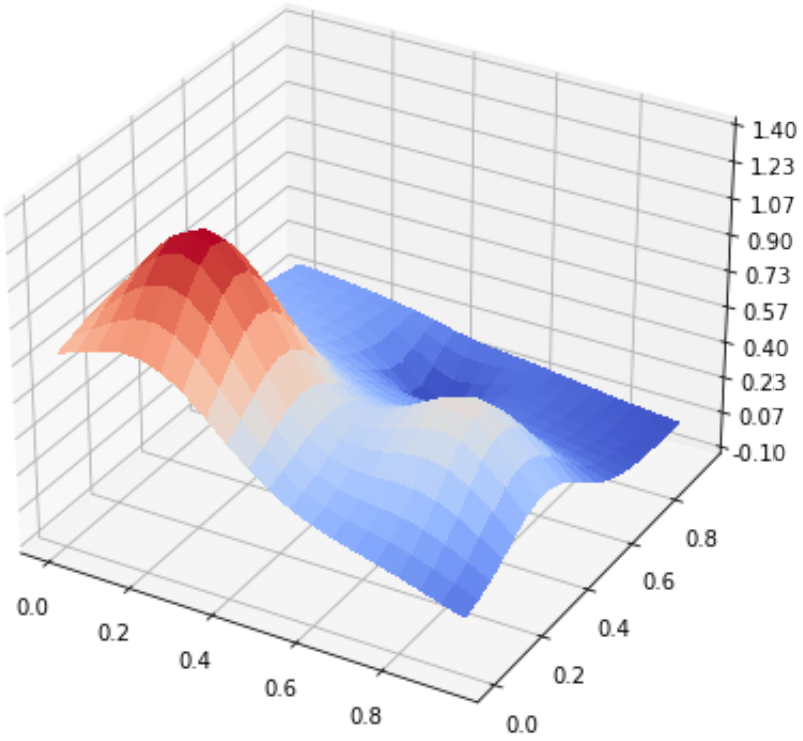


Figure 4: A 3D plot of Franke's function.

3.2 Wisconsin Breast cancer data

The data used for classification is the Wisconsin breast cancer data-set from the UCI repository[1]. The classification in the data-set is to distinguish benign breast tumors from malignant. 31 Features are generated from digitized images of a fine needle aspirate (FNA) of a breast mass, and include among others radius, texture, smoothness and concavity. The data-set has 569 samples, where 357 are benign and 212 are malignant.

Bellow is a summary of different pre-processing methods on the breast-cancer data we conducted with scikit-learn's neural network functionality. In this test, we compare the original data provided by scikit-learn, then we standardize the data with scikit-leans standard scaler that subtracts the mean and divides by the standard deviation, and lastly we attempted to reduce the dimensionality of the data to exclusively 4 dimensions by an algorithm found in fys-stk4155 lecture notes (See code). For the analysis, we specified that scikit should use a stochastic gradient descent solver with 100 epochs, while allowing scikit to decide/optimize all other paramaters. From Figure 5, clearly the data should be scaled beforehand to achive the optimal accuracy.

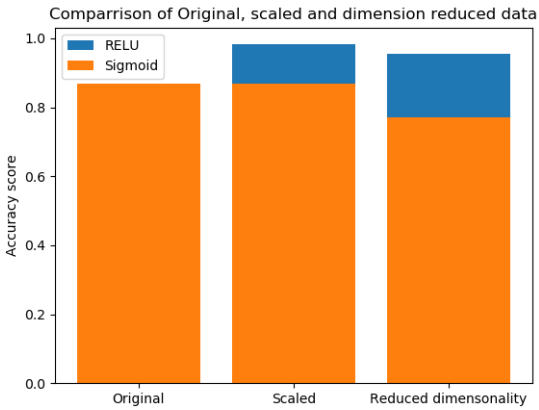


Figure 5: Accuracy score from scikit-learn's neural network when pre-processing the Wisconsin breast-cancer data by "original", "standardized" and "Reduced dimensionality" for both Sigmoid and RELU activation in the hidden layers.

4 Methodology

The implementation, and material, data relevant for this project can be found at `https://github.com/jornmarh/fys-stk4155.git/project2/`.

4.1 implementation of Stochastic gradient descent

In this section, we will show how the different SGD variants has been implemented. The data to be modelled is the Franke function discussed in the datasets section. For fitting the polynomial, OLS and ridge regression were used, replacing the analytical solution with the various SGD algorithms.

There are many different parameters to take into consideration when implementing SGD. The most important parameters, which is common for all of the different SGD algorithms, are number of epochs, the size of each mini-batch and the learning rate. Then, there are the specific parameters for the different methods. In SGD with momentum, γ has to be determined for weighting the running averages of previously encountered averages. Similarly, in RMS-prop, β has to be determined. When implementing ridge regression, the regularisation parameter, λ, also have to be tuned.

We found that good starting point was to compute the error as function of the number of epochs. In this project, we used the mean squared error and the r^2 score as the error estimates. This was first done with a qualified guess of a suitable learning rate and mini-batch size in order to get a feel for how the error evolved. We then tested with different learning rates and compared how fast and how well the error converged. A

grid-search was also conducted between the mini-batch size vs learning rate[2]. For the hyperparameters γ and β in SGD with momentum and RMS-prop, grid-searches were conducted between the learning rate and the hyperparameter to find the best combination.

4.2 Neural network

To update the paramaters with back-propagation as described in equation **??**, **??** and **??**, we need a cost function to evaluate the output of a feed-forward pass. In a regression example, the cost function may be the L2 norm, mainly due to simplifying the mathematics. In a classification problem, a typical choice is the cross-entropy. In our binary classification problem, we take use of a variant of the cross-entropy, defined bellow.

$$C = -\sum_{i=1}^{n}[t_i log(a_i^L) + (1 - t_i) log(1 - a_i^L)] \tag{32}$$

with derivative with respect to the output activations

$$\frac{dC(W)}{da_i^L} = \frac{a_i^L - t_i}{a_i^L(1 - a_i^L)} \tag{33}$$

Additionally, in our simple case we can apply the Sigmoid in the output layer. Along with some algebraic operations, see [3], the derivative of the Sigmoid function can be written as

$$\frac{df(z)}{dz} = f(z)(1 - f(z)) \tag{34}$$

Inserting this into equation **??** leads to the following simple expression for δ^L in the output layer

$$\delta^L = f'(z^L) \odot \frac{dC(W^L)}{da^L} = a^L - t \tag{35}$$

where t is the target values. The same expression can also be derived for the regression problem. Here, we do not need an activation function for the output layer, ie $f(z^L) = z^L$ and the derivative $f'(z^L) = 1$. With a quadratic cost function in the output layer we get the final expression for δ^L from:

$$C = \sum \frac{1}{2}(a - t)^2 \tag{36}$$

$$\frac{dC}{da} = (a - t) \tag{37}$$

$$\delta^L = \frac{dC}{da} * f'(z) = (a - t) * 1 = (a - t) \tag{38}$$

In regard to the various parameters that may be adjusted, such as the number of epochs, mini-batch size, number of hidden neurons, hidden layers and hyper-parameters. Our strategy was to first decide on a well functioning architecture of the neural network and balance the parameters of gradient descent between

computation time and convergence. As proposed by [12], the optimal batch-size for deep learning often lie in the range of 2-32. Venturing beyond these values quickly evolved to exploding gradients in our examples, in the end we settled for 10 as the mini-batch size. With all other parameters set, we looked to the learning rate and regularization parameter. This procedure was employed in order to put a greater emphasis on optimizing η and λ, since the hyper-parameters are the most crucial.

Another point of our methodology is the use of cross-validation. In this project we only employ cross-validation to study the classification case as a measure to save resources and time, meaning that the results regarding regression are more likely to be subject to a random-factor and may be error-prone.

5 Results

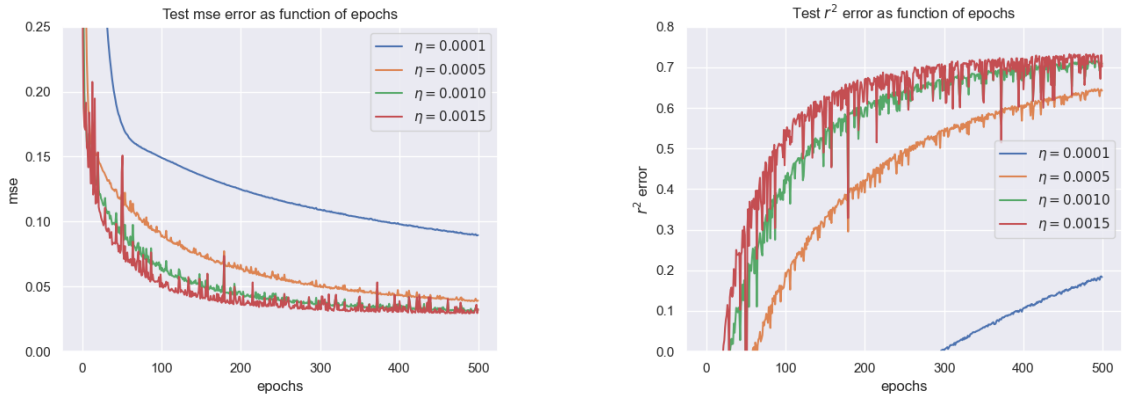
5.1 SGD algorithms

In this section, the results from the various variants of SGD are presented. Since the Franke function has an analytical solution for the minimum of the cost function, this was first calculated. Using 400 (20x20) datapoints and some added normally distributed noise, our degree five polynomial fit achieved a mean squared error of 0.0233 and r^2 score of 0.7870 using ordinary least squares, which serves as a reference to the performance of our SGD algorithms.

SGD without learning schedule

In figure 5, we can see the mean squared error as a function of the number of epochs, tested with different learning rates. The chosen size of each mini-batch was 10. The grid-search between η and mini-batch size showed quite similar results with mini-batch sizes of 5 to 20 datapoints, which can be viewed on our github[2]. Above 20, we encountered overflow in the matrix multiplication when computing the gradient.

The results aligned quite well with our expectations. With $\eta = 0.0015$, the error becomes very noisy, especially for the r^2 error. On the other hand, $\eta = 0.0001$ converges too slowly to give a satisfactory result. $\eta = 0.001$ seems to be the optimal learning rate, giving a mean squared error of 0.0319 and and r^2 error of 0.7089 using 500 epochs. For comparison, using scikit's SGDRegressor with the same parameters and a constant learning rate, the mean squared error was 0.0379 and the r^2 error was 0.6542.



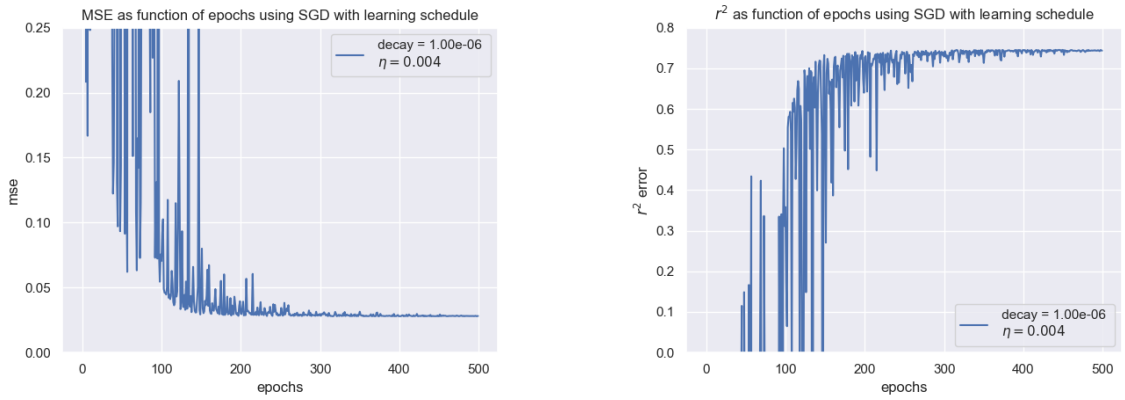
(a) mean squared error

(b) r^2 error

Figure 6: Errors as function of number of epochs, showing the effect of different learning rates.

SGD with learning schedule

Implementing the learning schedule explained in the methodology section and hand-tuning the decay, we could begin with an initial higher learning rate. Setting the initial learning rate to 0.004 with a decay of 10^{-6}, we managed to get somewhat faster convergence and an improvement to an mse of 0.0281 and r^2 of 0.7433, clearly coming closer to the analytical result. A plot of the errors as function of epochs can be shown in figure 6. It is interesting to observe the the highly fluctuating errors in the beginning when the learning rate is high, before the model manages to converge nicely as the learning rate decays. Altering the scikit regressor to use a "invscaling" as learning rate, we got an mse of 0.0390 and an r^2 error of 0.6436.



(a) mean squared error

(b) r^2 error

Figure 7: Plots showing how the error converges for SGD with a learning schedule

In the next section, the results from SGD with momentum and RMS-prop are presented. The values for the different hyperparameters were found by conducting grid-searches for the learning rate and the belonging

hyperparameter, which can be viewed in our github repository[2]

SGD with momentum

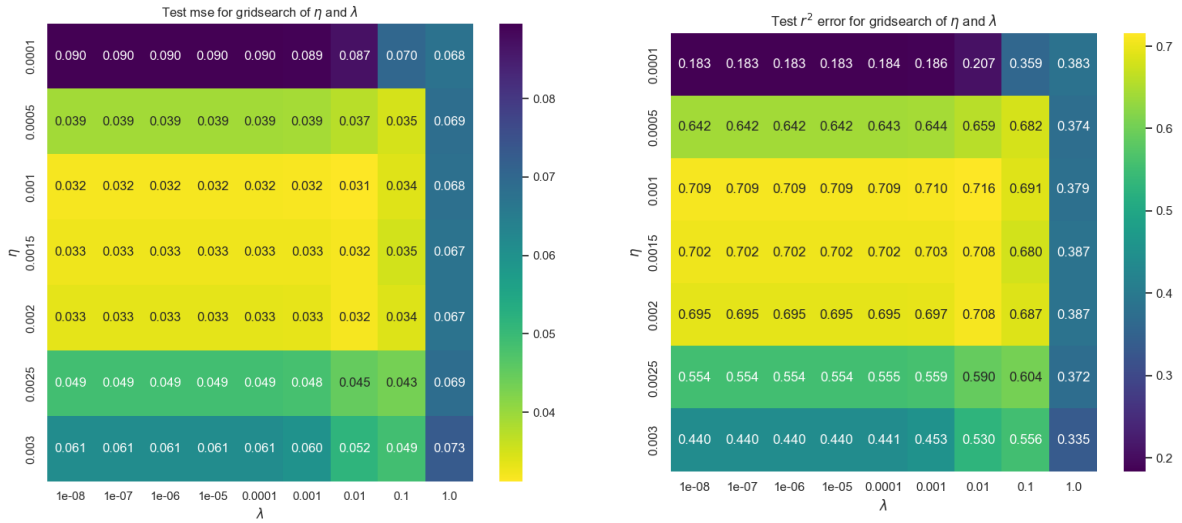
SGD with momentum was first implemented with a constant learning rate. γ was found to give the best result with a value of 0, equivalent to not using momentum, and learning rate of 0.0015. Adding the learning schedule and finding the optimal value for γ to be 0.6 with an initial learning rate of 0.003, we managed to improve the mean squared error to 0.0273 and the r^2 error to 0.7506.

RMS-prop

For the RMS-prop algorithm, the best value was found to be a learning rate of 0.02 and β as low as possible. The result was slightly poorer performance compared SGD momentum, giving an mse of 0.028 and r^2 error of 0.742.

Ridge regression

For ridge regression, regularization hyperparameter λ was introduced. A grid-search was conducted in order to find the best value of lambda and the learning rate. Figure 7 shows a heatmap of mse and r^2 scores with different values for η and λ. We observe a clear interval of learning rates that achieve good results, between 0.001 and 0.002. For the hyperparameter λ, we see that 0.01 gives the best scores, although it does not seem to affect the result to a high degree.



(a) mean squared error

(b) r^2 error

Figure 8: Gridsearch to find optimal values for η and λ

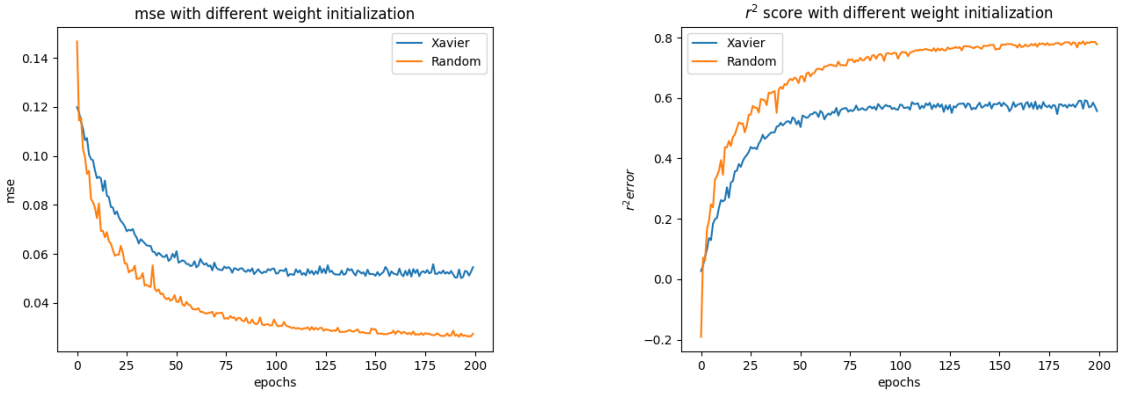
Other algorithms with ridge regression

The learning schedule, SGD with momentum and RMS-prop was also implemented for ridge regression the same way as was done for OLS. The parameters were optimized by grid-search and the mean squared error and r^2 score were evaluated. To avoid repetition, we skip straight to the result and reveal that none of the algorithms managed to achieve better scores with the ridge regression than with OLS, where an MSE of 0.027 and r^2 score of 0.716 were the best results, achieved with SGD momentum with a learning schedule. The "L2" penalty was also included in the scikit SGDRegressor, where a mean squared error of 0.0354 and r^2 error of 0.6765 was achieved, using the same parameters as our own model.

5.2 Neural network

5.2.1 Regression of the Franke function

For evaluating the neural network, we took basis in evaluating the mean squared error and the r2-score as function of the number of epochs, as we did in the SGD analysis. First, the initial network architecture was set, as described in the methodology section. We found that 2 hidden layers containing 40 neurons each was a good combination. We begin the analysis of our own neural network by looking at the performance when using the Sigmoid function in the hidden layers. The results can seen in figure 9 for both a normal distribution and Xavier initialization of the weights and zero-initialization of the biases.



(a) mean squared error

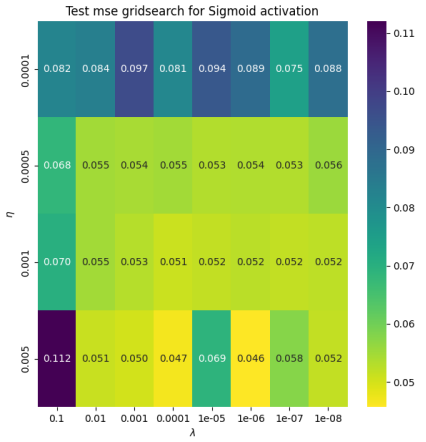
(b) r^2 error

Figure 9: The convergence of MSE and R2-score for the Sigmoid activation function when initialized with random weights and Xavier weights on test data

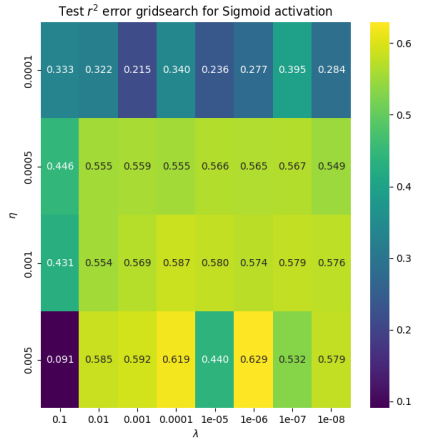
Comparatively, both methods were conducted with learning rate $\eta = 1e-3$ and regularization parameter $\lambda = 1e-7$. After 200 epochs we see that the Xavier weights begin better and converges quicker, however the standard distributed weights reach an overall better score. At 200 epochs, the results from both initialization schemes, along with the results of scikit-learns DNN for comparative parameters, can be found bellow.

- Standard normal: MSE 0.024, R2: 0.803
- Xavier: MSE 0.046, R2: 0.629
- Scikit-learn: MSE 0.053, R2: 0.572

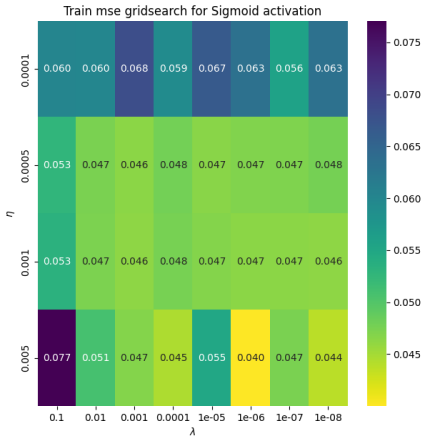
Next, we conduct a grid-search of the hyper-parameters η and λ to optimize the mean squared error and r2 score. Given the general consensuses in the literature on weight initialization in deep-learning [11], [6], [15], and accordance with scikit-learns DNN, we favour the results from Xavier initialization.



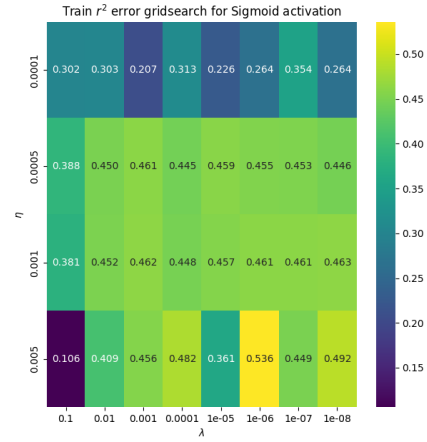
(a) mean-squared test error



(b) r^2 test error



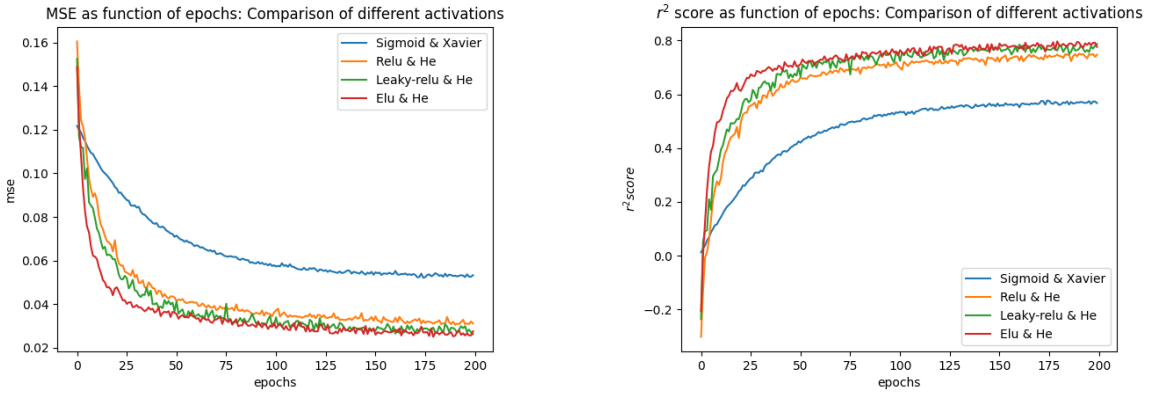
(c) mean-squared training error



(d) r^2 training error

Figure 10: A grid-search over various values of η and λ to achieve optimal model performance in terms of the MSE (left) and R2-score (right) for the Sigmoid function

The trend displayed in figure 10 show that the optimal model apply large learning rates and small regularization terms. We then replace the Sigmoid function with improved activation functions such as the RELU, followed by the leaky-RELU and finally the exponential RELU (ELU). These results can be seen in figure 11m where we compare both the final score, and the convergence-rate. In this figure, we employed Xavier initialization for the Sigmoid, and He initialization for the others as commonly recommended. Additionally, all tests were conducted with equal learning rate and regularization parameter: $\eta, \lambda = 1e-3, 1e-6$.



(a) mean squared error

(b) r^2 error

Figure 11: Comparison of the Sigmoid, RELU, leaky-RELU and ELU on regression of the Franke-function.

From figure **??** its clear that there is significant gain to be had by replacing the activation function from the Sigmoid. In this example we see that RELU drastically improves performance compared to the logistic function, both in terms of convergence and final score. And further improvement from the leaky-RELU and ELU. Conducting a grid-search similar to that for the Sigmoid, we was able to achive a best 0.026 MSE with RELU activation with $\eta, \lambda = 1e-3, 1e-5$. The graph showed an analogues trend to figure 10, in which higher learning rate and lower regularization provided the best score, these results were also in good agreement with what we got out of scikit-learn's neural net with RELU activation, with MSE 0.024. A summary of our findings, in addition to results from project number 1, ie analytical OLS and Ridge regression, can be found in table **??**. The heat map's of RELU activation and results from scikit-learn's dnn can be found at/or our github and appendix.

	MSE	r2 score	η	λ
OLS Analytic	0.0233	0.7870	-	-
Ridge Analytic	0.0239	0.7703	-	10^{-7}
SGD OLS	0.0319	0.7089	0.001	-
SGD OLS best (momentum)	0.0273	0.7506	0.001 (adaptive)	-
SGD OLS RMS-prop	0.0280	0.7420	0.015 (adaptive)	-
SGD Ridge	0.0310	0.7160	0.001	0.01
SGD Ridge best (momentum)	0.0273	0.716	0.001 (adaptive)	0.01
SGD Ridge RMS-prop	0.0275	0.7425	0.03 (adaptive)	0.01
NN Sigmoid (Xavier)	0.046	0.629	0.005	10^{-6}
NN Relu	0.026	0.7900	0.001	10^{-5}
NN Leaku Relu	0.026	0.791	0.001	10^{-7}
NN Elu	0.021	0.828	0.0009	10^{-5}
Scikit Sigmoid	0.053	0.572	0.05	10^{-5}
Scikit Relu	0.024	0.0804	0.05	10^{-5}

Table 1: Summary of the MSE and R2 score, including parameters used, as a result of different methods ranging from analytical expressions to neural nets.

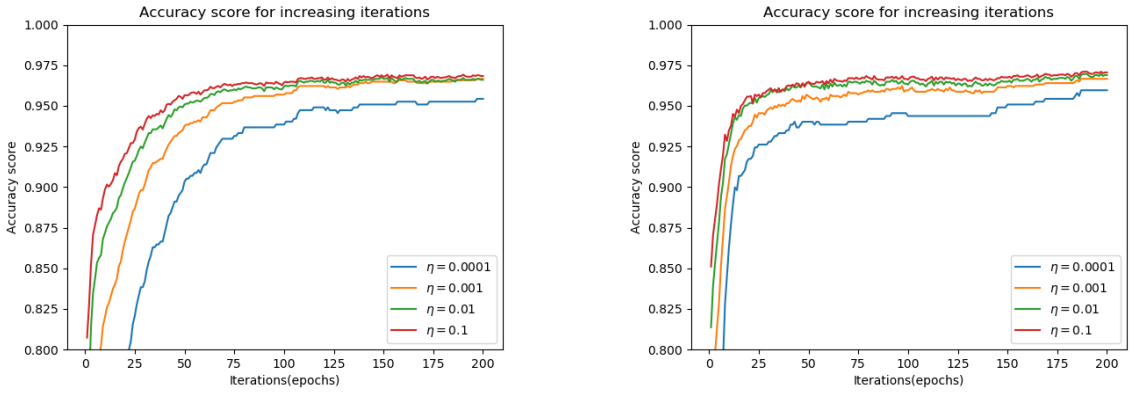
5.2.2 Classification of the Wisconsin breast-cancer data

In table 2) we have displayed the results from both our own neural network, and scikit-learn how the accuracy varies with network architecture. From this data we see that generally there is not much benefit to be had from extending the complexity of the network beyond 50 hidden nodes and 1 hidden layer, both in our own implemented network, and the neural network of scikit-learn.

	Hidden nodes			
Hidden layers	50	100	150	200
1	0.977 / 0.977	0.977 / 0.975	0.977 / 0.977	0.977, 0.975
2	0.975 / 0.977	0.975 / 0.974	0.974 / 0.975	0.973 / 0.975
3	0.974 / 0.974	0.975 / 0.975	0.975 / 0.975	0.973 / 0.975
4	0.929 / 0.974	0.968 / 0.974	0.974/ 0.974	0.972 / 0.975

Table 2: Accuracy score (own DNN/scikit DNN) for numerous network-sizes. 200 total iterations(epoch=200), $\eta, \lambda = 10^{-3}, 10^{-7}$

In table 2 we chose an arbitrary number of epochs = 200, but is this number truly justified? Bellow we show the accuracy-convergence on the test data as a function of different learning rates ranging from $1e^{-1} - 1e^{-4}$ for both a 50x1 architecture and 200x1 (hidden nodes x hidden layers).



(a) 50 hidden nodes

(b) 200 hidden nodes

Figure 12: Convergence for a 50x1 network (a) and 200x1 network (b) up to 200 epochs

As figure 12 insinuate, the best performances are found at high learning rates. Bellow in figures 13 and 14 we perform a grid search to validate this hypothesis and simultaneously study the dependence on the

regularization parameter.

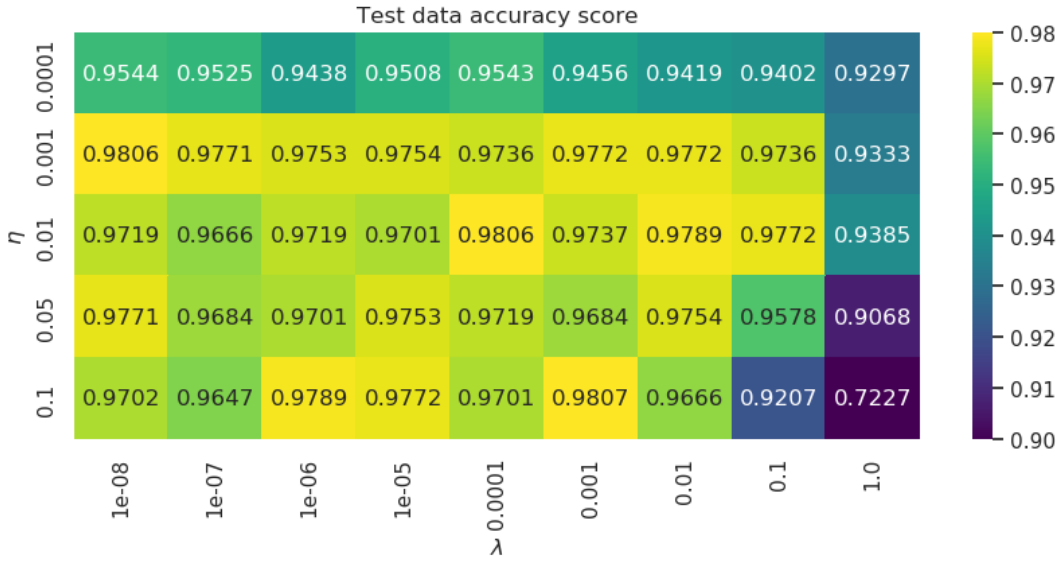


Figure 13: Grid search for optimal learning rate and regularization term of the test data for Sigmoid activation

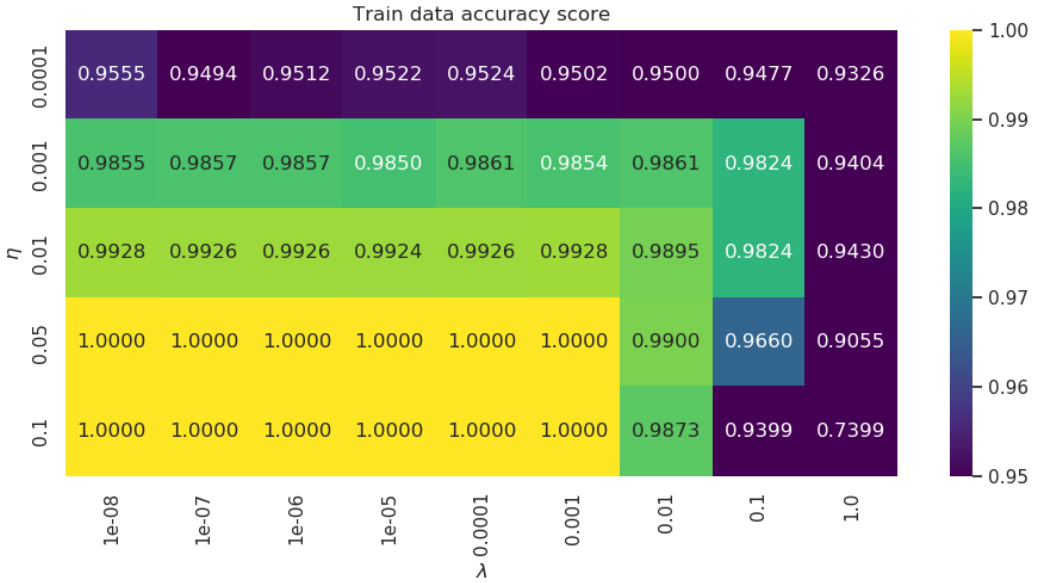


Figure 14: Grid search for optimal learning rate and regularization term of the train data for Sigmoid activation

Not surprisingly, we find strong performances for learning rates equal to $1e^{-3}$ or higher, with a best 0.9807 and 1.0 for the test and training data respectfully. These values also correspond excellent with what was found for Sciktit-learn's neural net, figures can be found in the appendix.

With RELU activation we found equally high scores on the test and training data, but at a smaller range of parameters. Particularly on the test data with a single score above 0.98 for $\eta, \lambda = 1e^{-3}, 1e^{1}$, as can be seen in the figures bellow.

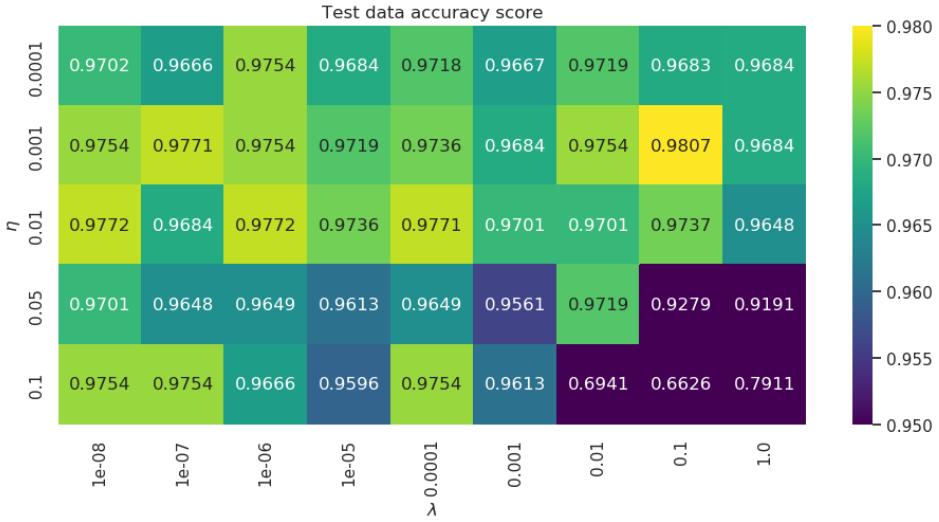


Figure 15: Grid search for optimal learning rate and regularization term of the test data for RELU activation

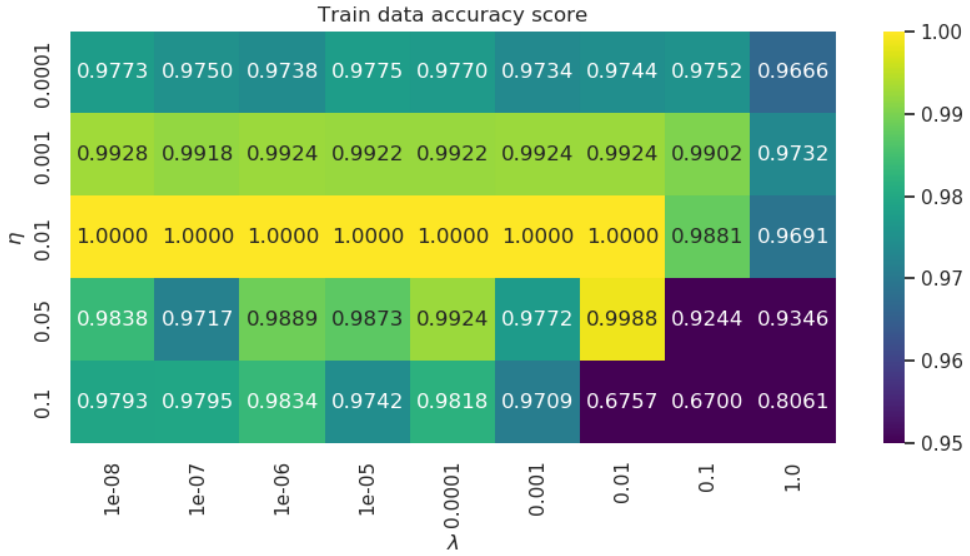


Figure 16: Grid search for optimal learning rate and regularization term of the train data for RELU activation

From figure 17 we do not observe any noteworthy improvements for our neural network between Sigmoid, RELU or leaky-RELU activation in the hidden layers for the first 100 epochs.

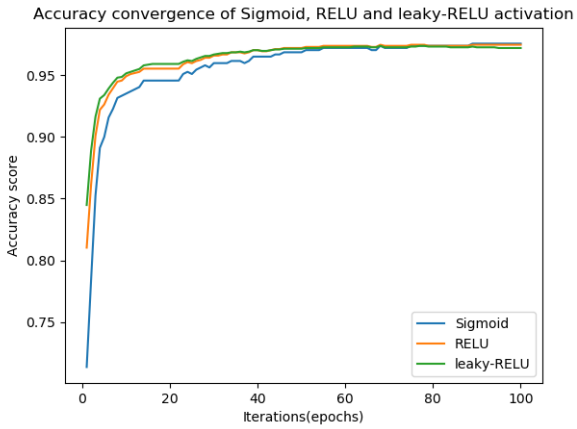


Figure 17: Sigmoid, RELU and leaky-RELU performance for 100 epochs

To follow is the results from our own implementation of logistic regression for various learning rates and regularization parameters for both the test and train data. We find very similar accuracy score, but slightly lower for the training data compared to the neural network.

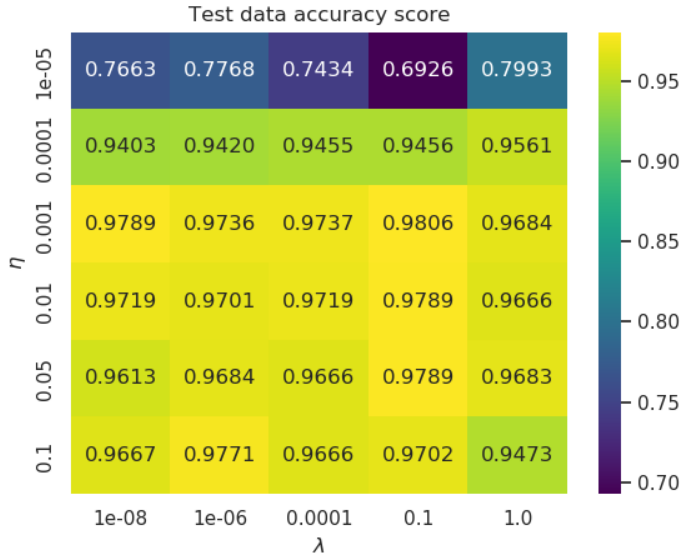


Figure 18: Grid search for optimal learning rate and regularization term of the test data with logistic regression

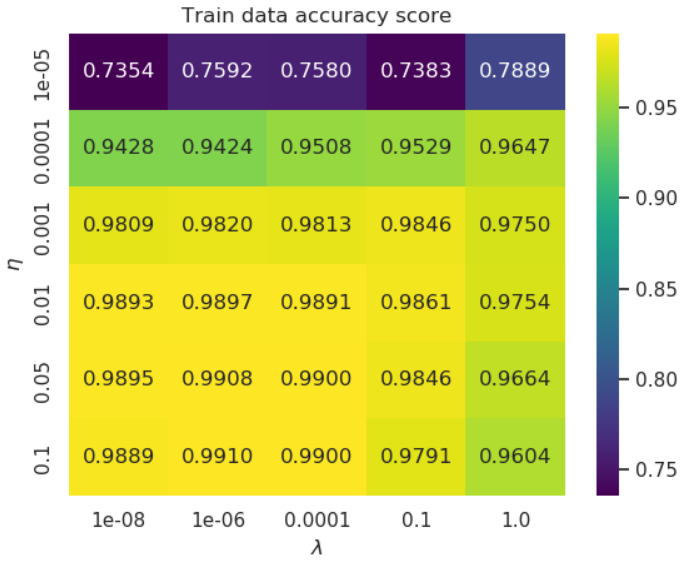


Figure 19: Grid search for optimal learning rate and regularization term of the train data with logistic regression

To conclude this section, we summarize our findings regarding the classification of the breast-cancer data from neural network and logistic regression, also including what we got with scikit-learn.

	Sigmoid	RELU	Logistic regression
Own	98.07 %	98.07 %	98.06 %
Scikit	98.24 %	98.07 %	97.37 %

Table 3: summary of classification of the Wisconsin breast-cancer data. All calculations allowed 200 epochs.

6 Discussion

6.1 Regression of the Franke-Function

In the application of solving linear regression with a neural network, we was able to get away with using a small network size of only 40 hidden nodes an 2 hidden layers. One of the benefits we observed from this network size, as opposed to larger sizes, was the amount of gradient problems. One more point on the architecture, without applying cross-validation which is both costly and meticulous to implement, it was also difficult to separate different architectures from choices that optimize our model, or that simply are better suited for one distinct seed.

From figure 9 its clear that the results are well converged at 200 epochs with Xavier weights. When setting up the weights with standard distribution, we allow the variance of the inputs to either increase or decrease when passing through the network. This may be the reason why the "Random" line does not converge. It would have been beneficial to iterate beyond 200 epochs, but we wanted a nice comparative baseline. All parameters regarding both the layout of our network and gradient descent methods, was thus chosen to both yield close to optimal scores, but also to limit computational time and errors in the network.

In figure (9) it may seem like random initialization is superior to Xavier weights. But this is most likely due to a random factor. It would have been interesting to cross-validate this method to either justify or disprove the results. On the other hand we can dishonor the results from looking at Scikit's neural network that outputs values much closer to our network with Xavier weights. Along with the general perception of weight initialization in deep learning and our discussion in section (2.2.4), this makes for a good enough justification to favour the results of the Xavier weights. Further, the data plotted in figure (9) was done with zero-initialized biases. All though it's not uncommon to initialize the biases with a small positive number to ensure that all neurons initially are active in the network. Xavier and He initialization infer that the infant biases be set to zero. Thus we applied zero biases for Random as well for the sake of comparison. Later on we tested for some nonzero biases but did not observe magnificent gain or penalty on the results.

In the heatmap, depicted in figures 10, a-d, where we searched for the optimal values of the learning rate and penalty parameter, we discover two two important features in our network. First, our own developed network seem to outperform the implementation in Scikit-Learn, which does not necessarily transfer to our network being the better developed. Rather, this dissimilarity most likely stems from the choice of seeds and random-state in Scikit, which is bound to be the case without testing with either bootstrapping or cross-validation. More other, scikit employs various advanced algorithms to train its neural network, such as adagrad stochastic gradient descent. With an even larger learning rate, we was able to lower the mean squared error in scikit-learn closer to that of our network.

Additional, despite our efforts to manually tune/optimize the arguments of scikit's learn neural network, we are not able to control and set every parameter and architecture of sciktit-learns network as we can our own. This may lead to the scikit dnn to be poorly optimized. But, of greater importance is the trend and dependencies of the learning rate and penalty parameter depicted in the heatmap for both Scikit's neural net and our own exertion. Namely that the learning rate is far more crucial to the system compered to lambda. Which is not surprising when comparing with our findings in project 1 where we did linear regression on the Franke function with ordinary least squares, ridge and lasso regression. The conclusion from that project was that least squares provided the best fit to the problem, hence a penalty parameter did not serve any purpose on that data set. Given that we are dealing with the same problem, but with a different model. A reasonable answer is that the Franke-function does not require regularization for good results. One more point on this, the regularization term's main purpose is to restrict the size of the weights, hence reducing the complexity of the network to help limit overfitting. The relationship between lambda in the test data in figure (10a+b) may indicate that with the small network size, the model is less prone to overfitting. Given more time, it would have been beneficial to attempt a broader study of hyper-parameters with different network sizes to validate this theory. On that note, we see that compared to the heat-map of of the training data, that we are able to obtain quite similar values, thus substantiate our claim regarding overfitting. Generally, we see that the training data does not depend on the regularization term unless for very large values, which is to be expected, as there is no overfitting on training data. The bottom row with 0.005 learning rate does seem to indicate a lambda dependence, but as mentioned in the SGD analysis, these high learning rates are very volatile, so without cross-validating the results its challenging to make any qualitative statement.

Activation functions and final comments on regression By substituting the activation function in the hidden layer we was able to significantly improve our network's performance on regression of the Franke function. As depicted in figures 11a and b, as well in table 1, the RELU, leaky-RELU and ELU all correlate to better performance. The results from the different activation functions on the Franke function is very in-line with what one might except from the literature and the theoretical aspects covered section (2.2.3). From this section we saw that the Sigmoid has several drawbacks that the RELU aims to resolve, and following its descendant's the leaky-RELU and ELU to further improve. We do note another bump in performance with these variants of the RELU as seen in table 1, however because of its easy implementation and adequate results, the original RELU is most often the preferred activation function for the hidden layers in neural networks.

In table (1) we have listed the results from the many methods one can apply machine learning to analyze the Franke function. From this summary the key findings is first of all that our own implementation of a neural network compare satisfactory well with what we get from scikit-learn, with modest differences because

of reasons discussed above. This is a nice validation of our results. Secondly, we observe that gradient descent and neural networks come extremely close to approximate analytical ordinary least squares. We do note that with the exponential RELU function the neural network supersede the results obtained with analytical least squares regression. This is odd, as it should not be possible to "beat" the analytical answer. On the other hand neural networks can to greater degree spot complex non-linear relationships, and thus does a better job at seeing through the noise and distortion from a small data-sample. According to (Leroy), the main benefits of neural networks compared to linear regression was found for very noisy and modest data-sets. In our regression analysis, we studied the Frankefunction of a 20x20 grid with added stochastic noise with mean 0 and variance 0.15. It would have been interesting to study this relationship between ordinary least squares and neural networks for a greater range of noise and data-samples to test this idea.

6.2 Classification of the Wisconsin breast-cancer data

In the classification of the Wisconsin breast-cancer data we found that a network consisting of only 50 hidden neurons and 1 hidden layer does an excellent job, with an initial accuracy of 97 percent. From table 2, its obvious that increasing the amounts of hidden layers would only lead to overfitting, seen from the low accuracy on the test data. Related to the immense amount of calculations to be performed, and limited time, we was only able to pursue optimization of the smaller network, which is a shame as it would have been instructive to relate the theory covered in sections 2.2.3-2.2.4 to a more complex network. Moving on to the convergence with respect to the accuracy score, figure 12 a and b confirm two central topics of our neural network, and deep learning in general. Firstly, a larger network requires fewer epochs to converge as opposed to a smaller one. And secondly, when training a network with stochastic gradient descent, smaller learning rates demand a greater number of iterations to converge as seen in figure (12b) for eta = 1e-5. For further investigation, we opted for the lesser network, and allowing the full 200 epochs, rather than employing the fast convergence of the larger architecture. This is mainly due to as in for the regression analysis, also here we noted an increase in unstable gradients for larger networks, in accordance with the input from Nielsen's book on deep networks.

In contrast to the regression analysis, we now employ cross-validation to perform the grid-search for ideal hyper-parameters. The result of this is that the predictions on the training data is largely independent of lambda excluding very high values, while as expected we observe a greater dependency on the test data. Specifically, we see from figure 13 a discrepancy bet wen 96 percent and 98 percent for the same learning rate.

Conversely to our findings for the Franke function, replacing the activation function does not advance the accuracy score on the Wisconsin breast-cancer data. In figure 15 and 16 we display the A1 accuracy scores for the test and training data respectfully. Again, we learn that the choice of learning rate outweigh the penalty parameter, as in the training data we obtain a perfect accuracy of 100 percent with eta = 1e-2

for a wide range of lambdas up tp 1e-2. On the other hand, for the test data in figure 15, the regularization parameter is crucial to the accuracy score, and we must employ a large lambda of 0.1 to reach the maximum accuracy. Such a large value of lambda may be a sign of overfitting the training data.

We generally find that that the results from RELU activation is in poor agreement with the RELU implementation of scikit-learn's DNN (see appendix). We relate these discrepancies to primarily two factors. The first is the convergence of scikit-learn, for the sake of comparison and computation time (large with CVD and grid-search) we restricted scikit-learn to 200 epochs, that constantly resulted in convergence-Warning flags. We later attempted singular analysis of scikit-learn with 1000 epochs, and did not note too big of a change, however this remains as a possible source of errors. Secondly, we can attribute the performance of scikit-learn to the number of features and methods accessible compared to our own code, most notably methods such as gradient clipping or batch normalization to avoid exploding gradients. This was most prominent when testing large learning rates in our own network. As seen from the heatmaps of the training data between our own implementing (figure 16) and scikit-learn. Going along with this comparison, we observe much greater agreement between scikit learn and our own network at learning rates 1e-2 and lesser for both the training and test data. Additionally, to further justify this statement, we find overall good agreement for train and test data between our own developed network and scikit-learn when using the logistic function. Regardless of the differences between our own network and scikit-learn we find that both are in agreement that the Sigmoid function work excellently to classify the breast-cancer data. In contrast to regression of the Franke function, where we discovered that the Sigmoid function is a terrible choice compared to the RELU.

While the predictions of our neural network could not match those of the analytical methods for linear regression, the neural network enhance the results from stochastic gradient descent with logistic regression. Where we found test scores around 97 percent from both our own implementation and scikit's implementation with 10-fold cross validation. These results are backed by the theory of neural networks, and essentially comes down to the architecture of an artificial neural network, compared to logistic regression. The universal approximation theorem formulated by (G. Cybenkot) [5] states that the complex architecture of an artificial neural network with appropriate weights and biases, given an activation function (originally Sigmoid) can approximate any continues function with just one hidden layer. From this theorem we can conclude that an ann should always outperform logistic regression in theory. In practise however this is not always the case considering the challenges in training neural networks. As seen in the very comparable accuracy score with logistic regression and the neural net, we may argue that logistic regression is a superior method for this data-set and classification simply as a result of its easy implementation and fast computation speed. Additionally neural nets are also more prone to overfitting as seen by studying the accuracy on the train data for both methods. On the other hand, neural nets allow for much greater flexibility and with proper optimization will always produce better results.

7 Conclusion

In this project we have studied the performance of a neural network for regression on Franke's function and binary classification of the Wisconsin breast-cancer data. In the regression case we was able to come extremely close to the analytical answer produced by ordinary least squares with OLS yielding MSE of 0.023 with our network producing an MSE of 0.026, compared to ordinary least squares with stochastic gradient descent equal to 0.0273. We found similar results also for the classification task, where we obtained an accuracy score of around 98% from the neural network, slightly higher and more consistent than we found with logistic regression. As is expected from the added complexity a neural network provide over logistic regression.

Furthermore we discovered that the Sigmoid function is not well suited for a neural network to perform regression. On the other we found the Sigmoid to yield excellent accuracy in classification. Our take on this result is that while the Sigmoid remap all data between 0 and 1, the RELU function only transform negative values. In our implementation of a neural network for regression we did not use any activation function to the output layer, hence for positive inputs, the RELU network is essentially a linear relationship. And the Sigmoid will do a non-linear transformation of the data regardless of inputs. We might therefore suspect that the RELU for this reason is better equipped to model linear dependencies like the Franke-function. However we have not found any literature to back this statement, so this will serve as primarily our own take on whats happening. Another key discovery of this project was that although neural networks provide great flexibility and generality, which allow them to approximate any function, this makes them a hassle to tune and optimize. Particularly, we found the use of cross-validation extremely handy in this case, to neglect a certain level of uncertainty from our results.

In terms of further research, we would thoroughly enjoy a broader study of neural networks on different data-sets. In that way we could get a real feel for the use of neural networks, as in this project we obtained very similar results using easier to implement and quicker methods like logistic and linear regression.

8 Appendix

Here we include some of the results from scikit-learn that are relevant for the discussion, additional plots such as grid searches with higher learning rates can be located at our github address.

8.1 Regression

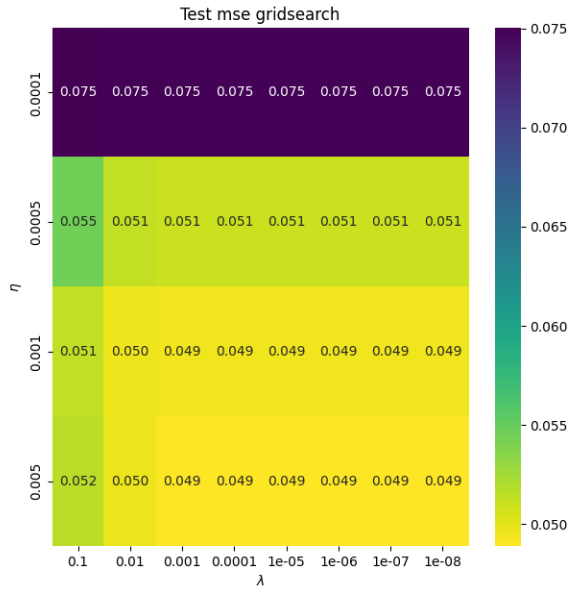


Figure 20: Sigmoid MSE train

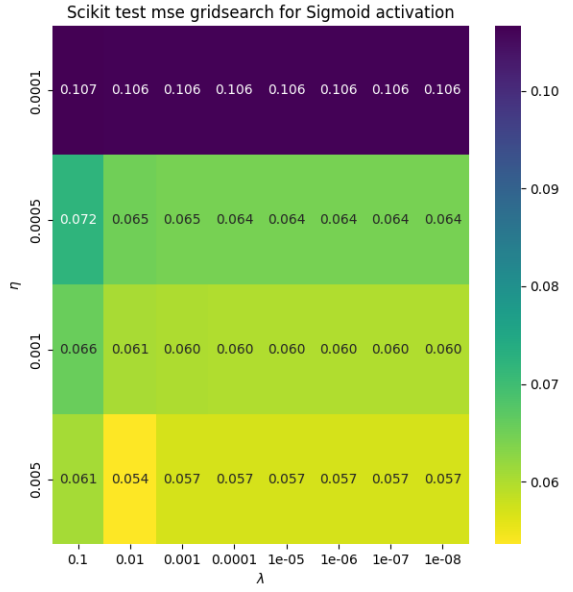


Figure 21: Sigmoid MSE test

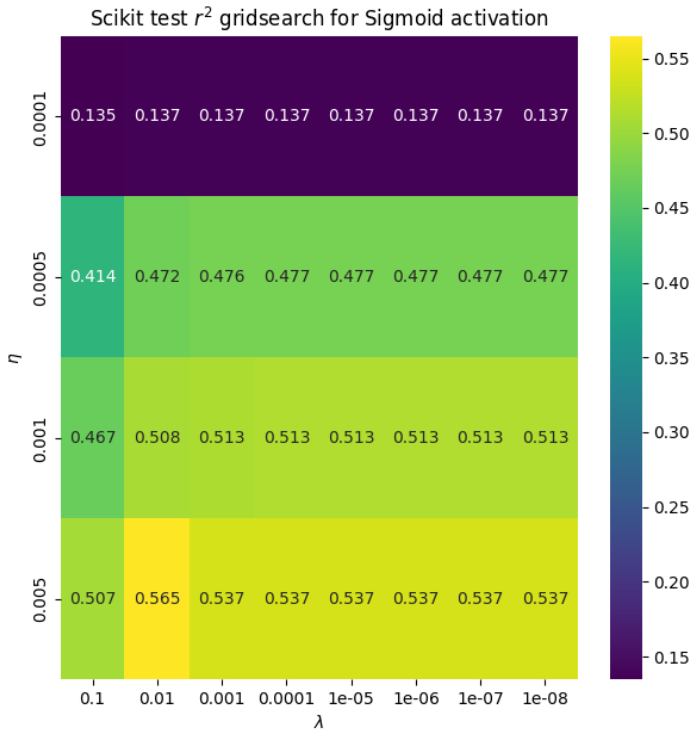


Figure 22: Sigmoid R2 test

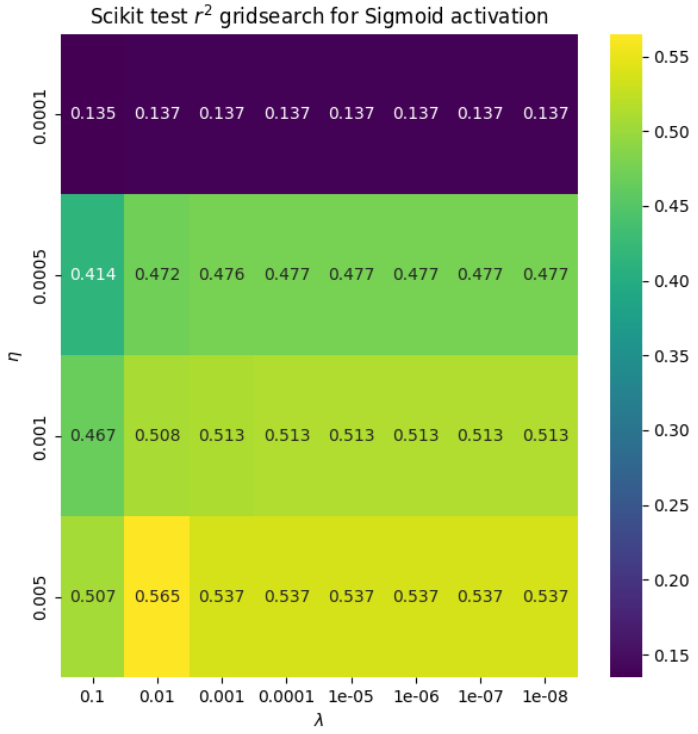


Figure 23: Sigmoid r2 train

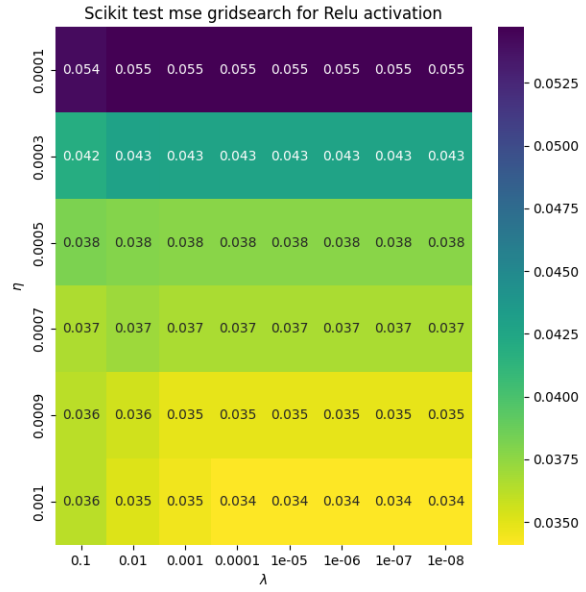


Figure 24: RELU MSE test

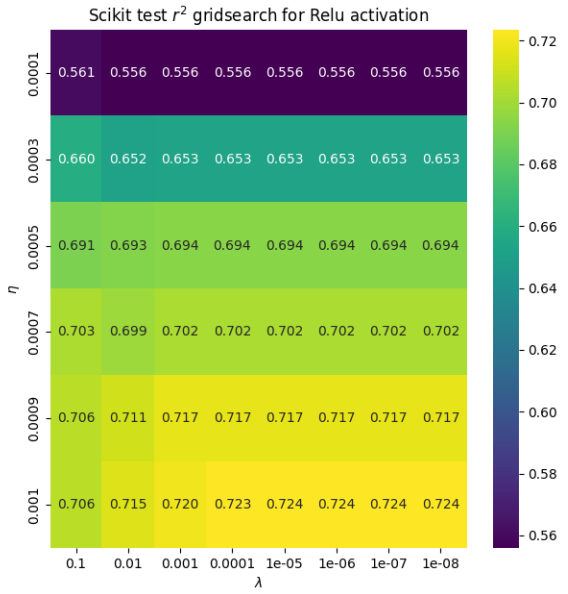


Figure 25: RELU R2 test

8.2 Classification

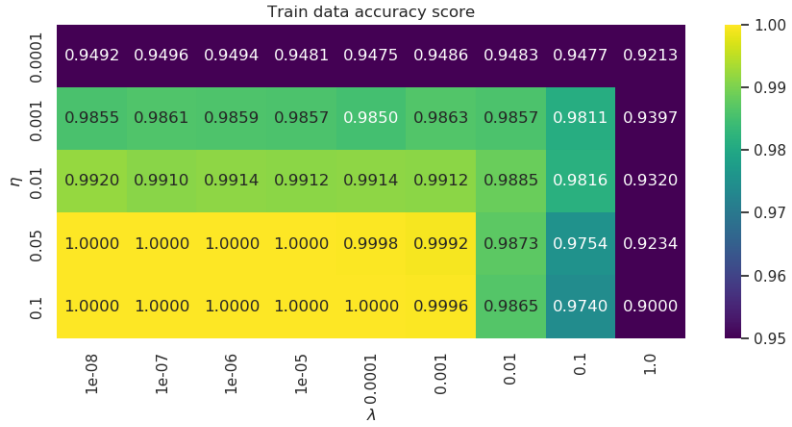


Figure 26: Sigmoid train

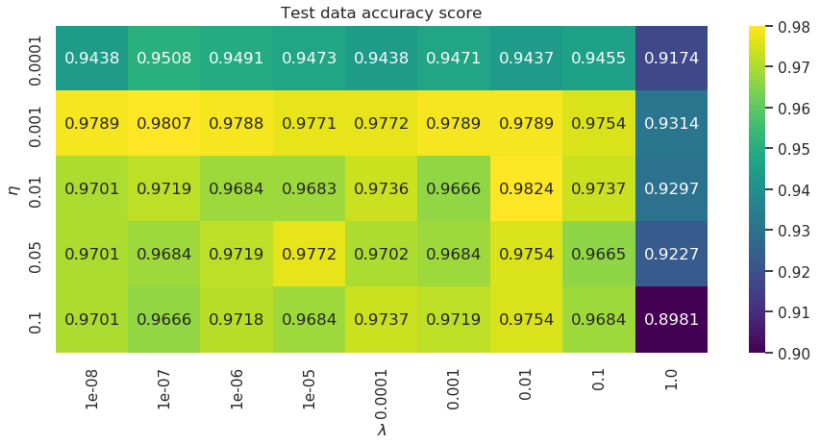


Figure 27: Sigmoid test

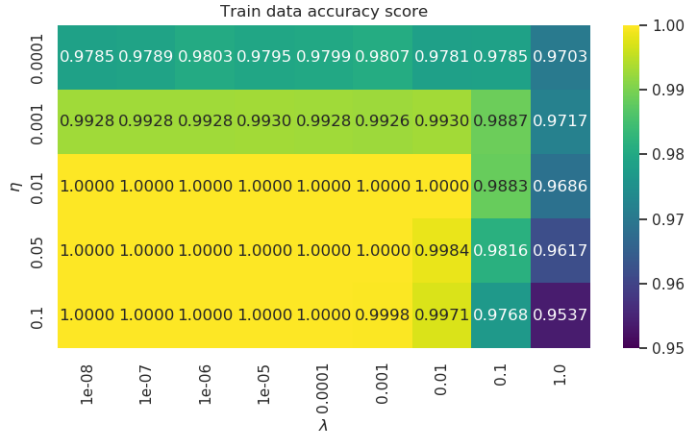


Figure 28: RELU train

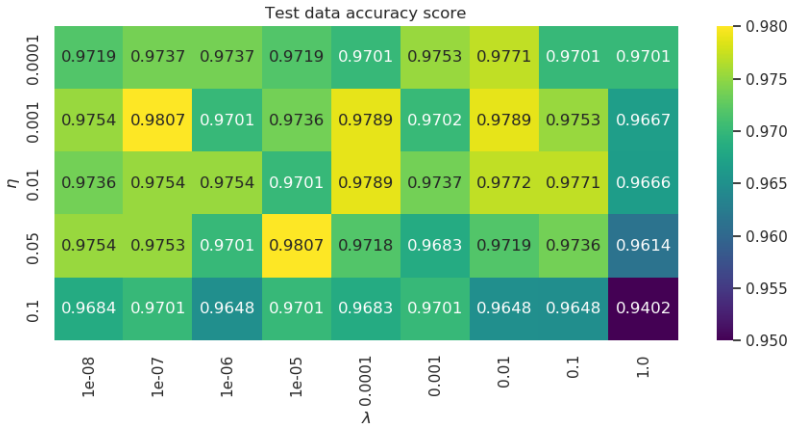


Figure 29: RELU test

References

[1] URL: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

[2] URL: https://github.com/jornmarh/fys-stk4155/tree/main/project2/figures.

[3] Arc. *Derivative of the Sigmoid function*. en. July 2018. URL: https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e (visited on 10/28/2021).

[4] Casper Hansen. *Activation Functions Explained - GELU, SELU, ELU, ReLU and more*. en. Aug. 2019. URL: https://mlfromscratch.com/activation-functions-explained/ (visited on 11/10/2021).

[5] G. Cybenko. "Approximation by superpositions of a sigmoidal function". en. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274 (visited on 11/20/2021).

[6] Xavier Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Journal of Machine Learning Research - Proceedings Track* 9 (Jan. 2010), pp. 249–256.

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

[8] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". en. In: *arXiv:1502.01852 [cs]* (Feb. 2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852 (visited on 11/17/2021).

[9] Morten Hjorth-Jensen. *6. Logistic Regression — Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter4.html (visited on 11/20/2021).

[10] jamesdmccaffrey. *Why a Neural Network is Always Better than Logistic Regression*. en. July 2018. URL: https://jamesmccaffrey.wordpress.com/2018/07/07/why-a-neural-network-is-always-better-than-logistic-regression/ (visited on 11/20/2021).

[11] Katanforoosh and Kunin. *AI Notes: Initializing neural networks*. 2019. URL: https://www.deeplearning.ai/ai-notes/initialization/ (visited on 11/12/2021).

[12] Dominic Masters and Carlo Luschi. "Revisiting Small Batch Training for Deep Neural Networks". en. In: *arXiv:1804.07612 [cs, stat]* (Apr. 2018). arXiv: 1804.07612. URL: http://arxiv.org/abs/1804.07612 (visited on 11/11/2021).

[13] Morten Hjorth-Jensen. *13. Neural networks — Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html#deriving-the-back-propagation-code-for-a-multilayer-perceptron-model (visited on 11/17/2021).

[14] Morten Hjorth-Jensen. *14. Building a Feed Forward Neural Network — Applied Data Analysis and Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter10.html#developing-a-code-for-doing-neural-networks-with-back-propagation (visited on 10/28/2021).

[15] Michael A. Nielsen. "Neural Networks and Deep Learning". en. In: (2015). Publisher: Determination Press. URL: http://neuralnetworksanddeeplearning.com (visited on 11/10/2021).