Things to include in the report

1. Create SGD regression
   - Test for different learning rates, mini batches, epochs, adaptive learning rate?
   - Compare with OLS and Ridge on Franke function
   - For ridge study results as function of the hyper-parameters λ and learning rate (heat map)
2. Create NN
   - Discuss intializing weights and biases, cost functions, activation functions and all that stuff. What to do on the output layer regarding activation function.
   - Compare NN regression on Franke function vs. OLS and Ridge (λ and η heat map)
   - Compare to SKLearn/Tensorflow
3. Try out different activation functions (Sigmoid, Relu and Leaky relu
4. Classification (costfunction = logistic)
   - Test logic gates classification
   - Breast cancer data: Important to scale here. Discuss also methods/whether to reduce dimensions. Measure performance with accuracy. Again: optimize for hyper-parameters and various activation functions perhaps.
5. Write logistic regression code using SGD
   - Compare NN to logistic regression on MNIST data
6. Summarize everything

Classification and Regression, from linear and logistic regression to neural networks

Sakarias Frette, Mikkel Metzsch Jensen, William Hirst
(Dated: November 13, 2021)

'Twas a beautiful morning, the sun was shining and the birds were humming. Nothing short of a great start on the day. With work starting in only 2 hours, I ran to the kitchen to assure a sufficient and delicious breakfast before the 8 hour shift of hard labor. As my mother used to say, a great breakfast makes a great man, and I lived for that sentence every day.

CONTENTS

I. INTRODUCTION

The use of Neural Networks have exploded in academia the last 10 years, and continue to impress with its capabilities in complex problem solving. DeepMind made its AlphaGo, which beat the reigning champion 4/5 games, facial recognition in China is so good now that it works as ID on the subway, and cars can now drive them selves on the road with higher than 99% success rate [3] .

Neural Networks and regression have somewhat overlapping area of use, especially in with regards to classification and regression. In fact, logistic regression and a Neural Network with no hidden layers are practically the same thing. Neural Networks can however scale much better in dimensionality, such that it can solve complex problems such as quantum many body problems, image recognition and complex differential equations.

II. THEORY

A. Gradient descent

The essence of Machine learning is in a sense, just a cost minimization problem with respect to some parameters. For machine learning these parameters is the weights and the biases, but for now we just consider a general parameter $\boldsymbol{\theta} = \{\theta_1, \theta_2, \dots \theta_n\}$ for n dimensions. The goal is to choose $\boldsymbol{\theta}$ such that we minimize a given cost function $C(\boldsymbol{\theta})$ from a set of data points in the design matrix $\mathbf{X}$ and corresponding target values $\mathbf{t}$. One common approach to such a problem is the Ordinary Least Squares method (OLS) for which we find the optimial $\boldsymbol{\theta}_{opt}$ that minimize the cost function as

$$\boldsymbol{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$$

This is the main workhorse behind many regression problems, but as the number of data points increases the matrix inversion becomes computational expensive. Furthermore, the matrix product $\mathbf{X}^T\mathbf{X}$ might be singular and thus the above approach needs additional computations to work out. The concept of gradient descent provides a computational efficient alternative to this problem.

1. *Regular Gradient descent*

For a given cost function $C(\boldsymbol{\theta})$ we can approach the minimum by calculating the gradient $\nabla_\theta C$ with respect to the unknown parameters $\boldsymbol{\theta}$. When evaluating the gradient in a specific point $\boldsymbol{\theta}_i$ in the parameter space, the negative of the gradient will correspond the direction for which a small change $d\boldsymbol{\theta}$ in the parameter space will result in the biggest decrease in the cost function. By choosing a step size η, better known as the learning rate, we can iterate forward in the parameter space as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta\nabla_\theta C(\boldsymbol{\theta}_i). \tag{1}$$

In order to get convergence towards a minimum we must choose the learning rate η to be sufficiently small such that it does not "step over" the minimum point. The usual approach to choosing η is simply to define it as a hyperparameter and run the gradient descent scheme for different η's and choose the one that yields the best result.

For some simpler cases, where we are able to calculate the so-called Hessian matrix, we can compute an approximation for the optimal learning rate. For a convex problem the gradient is only equal to zero at the global minimum for which the optimal parameter must satisfy.

$$g(\boldsymbol{\theta}) = \nabla_\theta C(\boldsymbol{\theta}) = 0.$$

By using Newton-Raphson's method we can iterate towards the root for the gradient by following the tangent line (1. order approximation) as

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \frac{g(\boldsymbol{\theta}_i)}{g'(\boldsymbol{\theta}_i)},$$

where

$$g'(\boldsymbol{\theta}_i) = \frac{\partial g(\boldsymbol{\theta}_i)}{\partial \boldsymbol{\theta}_i} = \nabla^2_\theta C(\boldsymbol{\theta}_i) = \mathbf{H}$$

The final iteration scheme reads

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \mathbf{H}^{-1}\nabla_\theta C(\boldsymbol{\theta}_i)$$

which resemble equation 1 with $\eta = \mathbf{H}^-1$. Thus the best learning parameter for a first order approximation is found as the inverse of the hessian. Notice that this is only applicable in cases where the cost function is twice differentiable and where we can compute the inverse of the Hessian. However, for a convex problem the hessian is symmetric and positive definite for convex problems for which the matrix is always invertible.

source for the other stuff: (http://www.cs.cornell.edu/courses/cs4780/2015fa/web/lecturenotes/lecturenote07.html.

2. *Stochastic gradient descent*

A variation of the regular gradient descent is the Stochastic gradient Descent (SGD) for which we introduce the concept of mini batches. For each step we divide the data into randomly into mini batches of size m. For each batch j we compute the gradient and the contribution for the updated parameter $\boldsymbol{\theta}^j_{i+1}$ using to equation

1. Conventionally one would then sum up all such contributions and update the final parameter as the mean value

$$\boldsymbol{\theta}_{i+1} = \sum_j \boldsymbol{\theta}^j_{i+1},$$

but in our implementation we are going to simply update $\boldsymbol{\theta}$ for each batch as it is yields practically identical results for a slight memory storage benefit.

The main benifit of using SGD is that the computation of the gradient can become computational expensive when dealing with increasing data. Since the gradient calculation involves a number of matrix operations it becomes computational beneficial to perform more gradient calculations of the reduced mini bathces rather than one for the whole data set at once.

3. *Gradient descent with momentum*

As a last alternative to optimize the gradient descent, we introduce momentum also known as inertia. The key idea is that we keep track of the previous gradient step, such that we keep a memory of the direction we are moving in parameter space. When increasing the momentum parameter γ the movement through parameter space becomes more steady and less effected by small fluctuations in the cost function. We can think of this as a heavy ball rolling down a rugged landscape. the heavier the ball the less prone to sudden changes in direction it will be. The main idea is that the momentum concept can reduce the change of getting stuck in local minimums when travelling through parameters space.

Formally we define this as:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \tag{2}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{3}$$

where $\mathbf{v}_t$ is the velocity term for timestep (epoch) t. Notice that whebn choosing $\gamma = 0$ we have zero momentum and equation 3 becomes equivilant with the regular gradient descent method described in equation 1.
Notice that there exist a whole variety of different momentum-based methods, for which some of well known might be the "RMS prop" or "ADAM optimizer".

B. Neural Networks

We now introduce the basic concepts of the Neural Network (NN). Specifically we are going to address a dense feedforward neural network, meaning the information is only moving forward trough the network and each node in a given layer is connected to each and every node following layer. The general structure of the network is sketched in figure 1. In the beginning we have an input layer with one node for each feature of the data. Following we have a series of so-called hidden layers which can have a custom chosen number of nodes for each layer. Finally we have the output layers with one node per target category. For a simple True and False type of problem a single output node which takes values between 0 and 1 is sufficient for the prediction. For classification between multiple classes we would require an output node for each class.

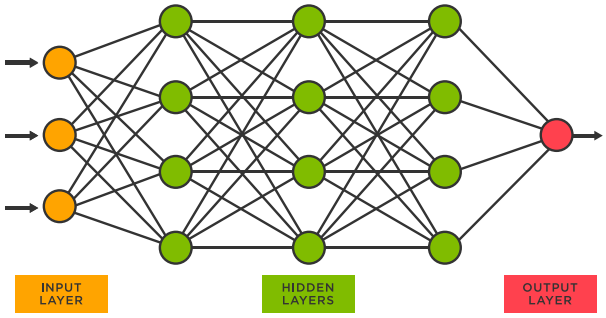


Figure 1: Conceptual illustration of Neural Network. https://www.tibco.com/reference-center/what-is-a-neural-network

The connection between the nodes is controlled by the introduction of the weights w and biases b. The weights and biases serves as the unknown parameter which we need to choose such that the network predictions minimizes a given cost function. For an already trained network we can collect the predictions by performing the so-called feed forward mechanism which is explained in details in the following section.
Howebver, before diving into the details behind the neural network and the key algorithms for performing training and prediction, we want to put down some notation. We aim to use some of the most common notation, but is easy to get lost in the following sections, and thus we provide a table (I) for which you can always go back to clarify the meaning of variables and indexes.

Table I: Notation

Matrices		
Notation	Description	Type
X	Design Matrix (input data).	$\mathbb{R}^{N\times\text{features}}$
t	Target values.	$\mathbb{R}^{N\times\text{categories}}$
y	Model output, the prediction from our network.	$\mathbb{R}^{N\times\text{features}}$
W^l	The weight matrix associated with layer l which handles the connections between layer $l-1$ and l .	$\mathbb{R}^{n_{l-1}\times n_l}$
w^l_{ik}	The weight connecting node i in layer $l-1$ to node k in layer l.	$\mathbb{R}$
B^l	The bias vector associated with layer l which handles the biases for all nodes in layer l.	$\mathbb{R}^{n_l\times 1}$
b^l_j	Bias acting on node j in layer l.	R
z	Node output before activation.	
a	Activated node output.	R
Functions		
C	Cost function	
σ^l	Activation function associated with layer l.	
More...		
n_l	The number of nodes in layer l.	
L	Number of layers in total with $L-2$ hidden layers.	
N	Total number of datapoints.	

1. *Feeding Forward*

The forward mechanism is the main mechanism behind the usage of the neural network. For a single data point the data flow becomes as outlined in the following.

1. The input nodes receive the input data for each feature.

2. Each input node then send the data value into each node of the following hidden layer with a scaling according to the associated weight of every single connection.

3. Each node in the hidden layer sums up the weighted contributions from the input layer an add a bias value associated to that given node. We denote this raw node output by z.

4. The unactivated value z is immediately send through an activation function σ associated with the layer to produce the activated value $a = \sigma(z)$.

5. Each node in the hidden layer then forward the activated value into the next layer using the same procedure. Notice that the number of nodes in the hidden layers no longer corresponds to any given features and one can only speculate on how the complex network creates its own sub-features and interprets the underlying correlations in the data.

6. Finally the stream of forwarded data enters the output layer where the activation values on each node corresponds to the network predictions for each category.

The feed forward mechanism is for a simple four layer network is shown in figure. 2.

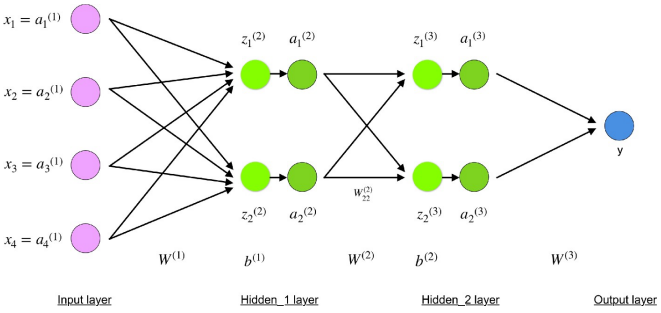


Figure 2: Illustration of simple 4-layer neural network. Source: towardsdatascience.com.

The feed forward step to obtain the activation value on a given layer $l \neq 1$ (not the input layer) is given as

$$z^l_j = \sum_{i=1}^{n_{l-1}} w^l_{ij} a^{l-1}_i + b^l_j, \quad a^l_j = \sigma^l(z^l_j) \tag{4}$$

where z^l_j is the input value to node j in layer l, w^l_{ij} is the weight connecting node i in layer $l-1$ to node j in layer l, a^{l-1}_i is the activation value from node i in layer $l-1$ and b^l_j is the bias associated with node j in layer l. We

can obtain a matrix-variant of equation 4 by defining

$$W^l = \begin{bmatrix} w^l_{11} & w^l_{21} & \dots & w^l_{n_{l-1}1} \\ w^l_{12} & w^l_{22} & \dots & w^l_{n_{l-1}2} \\ \vdots & \vdots & & \vdots \\ w^l_{1n_l} & w^l_{2n_l} & \dots & w^l_{n_l n_{l-1}} \end{bmatrix},$$

$$(W^l)^T = \begin{bmatrix} w^l_{11} & w^l_{12} & \dots & w^l_{1n_l} \\ w^l_{21} & w^l_{22} & \dots & w^l_{2n_l} \\ \vdots & \vdots & & \vdots \\ w^l_{n_{l-1}1} & w^l_{n_{l-1}2} & \dots & w^l_{n_{l-1}n_l} \end{bmatrix},$$

$$A^l = \begin{bmatrix} a^l_1 \\ a^l_2 \\ \vdots \\ a^l_{n_l} \end{bmatrix}, \qquad B^l = \begin{bmatrix} b^l_1 \\ b^l_2 \\ \vdots \\ b^l_{n_l} \end{bmatrix}, \qquad Z^l = \begin{bmatrix} z^l_1 \\ z^l_2 \\ \vdots \\ z^l_{n_l} \end{bmatrix},$$

such that we get

$$Z^l = (W^l)^T A^{l-1} + B^l, \qquad A^l = \sigma(Z^l). \tag{5}$$

The complete feed forward algorithm is simply done by using equation 5 for layer $l = 2, \dots, L$, for which produce the neural network prediction $\mathbf{y}$.

2. *Back Propagation*

The back propagation algorithm is the main workhorse behind the training of the neural network. This serves the purpose of tuning the weights and biases such that the prediction from our neural network (after running the feed forward algorithm) becomes better and better. That is we want to minimize a cost function C, which will evaluate how close the prediction $\mathbf{y}$ comes to the target values $\mathbf{t}$. A common choice for regression type problems is the mean squared error as

$$C(\mathbf{y}) = ||\mathbf{y} - \mathbf{t}||_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2$$

but one can choose from a wide selection of cost function in general.

By calculating the gradient gradient $\nabla_{w,b}C$ with respect the weights and biases, we can use gradient descent (see section II A) to minimize the cost function. Thus need to evaluate $\partial C/\partial w^l_{ij}$ and $\partial C/\partial b^l_j$. We begin by looking at the last layer $l = L$. By using the chain rule we get

$$\frac{\partial C}{\partial w^L_{ij}} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j}\frac{\partial z^L_j}{\partial w^L_{ij}}.$$

We remember that

$$a^L_j = \sigma(z^L_j), \qquad z^L_j = \sum_{i=1}^{n_{L-1}} w^L_{ij} a^{L-1}_i + b^L_j,$$

such that we get

$$\frac{\partial C}{\partial w^L_{ij}} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j)a^{L-1}_i,$$

Notice that the remaining derivatives can easily be calculated when deciding on a specific cost and activation function. For the bias we simply get

$$\frac{\partial C}{\partial b^L_j} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j}\frac{\partial z^L_j}{\partial b^L_j} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j)\cdot 1$$

We use this to motivate the definition of the local gradient

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j},$$

such that

$$\delta^L_j = \frac{\partial C}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j).$$

This yields the more compact expressions

$$\frac{\partial C}{\partial w^L_{ij}} = \delta^L_j a^{L-1}_i \qquad \frac{\partial C}{\partial b^L_j} = \delta^L_j$$

The local gradient δ^l_j is also commonly called the "error" since it says something about how big an influence the j^{th} node in layer l have on the change of the cost function. We let $\boldsymbol{\delta}^l$ denote the vector containing all the local gradients associated with layer l and we can write it out as a matrix equation for the last layer

$$\boldsymbol{\delta}^L = \nabla_a C \odot \frac{\partial \sigma}{\partial z^L}, \quad \nabla_a C = \left[\frac{\partial C}{\partial a^L_1}, \frac{\partial C}{\partial a^L_2}, \dots, \frac{\partial C}{\partial a^L_{n_L}}\right]^T$$

where $\odot$ is We can define the local gradient δ^l_j for the j^{th} node on a general layer l in terms of δ^{l+1}_k for the k^{th} node on the next layer $l+1$ by using the chain rule

$$\begin{aligned} \delta^l_j &\equiv \frac{\partial C}{\partial z^l_j} \\ &= \sum_k \frac{\partial C}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_j} \\ &= \sum_k \frac{\partial z^{l+1}_k}{\partial z^l_j}\delta^{l+1}_k \end{aligned}$$

We remember

$$z_k^{l+1} = \sum_{j=1}^{n_l} w_{jk}^{l+1} a_j^l + b_k^{l+1} = \sum_{j=1}^{n_l} w_{jk}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

Differentiating, we obtain

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{jk}^{l+1} \sigma'(z_j^l),$$

And by substituting back we find

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{ij}^{l+1} \sigma'(z_j^l)$$

The back propagation algorithm then becomes

- Compute

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

- for $l = L-1, L-2, \ldots, 1$ compute:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{jk}^{l+1} \sigma'(z_j^l)$$

  Finally we update weights and biases:

$$w_{jk}^l \leftarrow w_{jk}^l - \eta \delta_j^l a_i^{l-1}$$
$$b_j^l \leftarrow b_j^l - \eta \delta_j^l$$

  , where η is still the learning rate which decides the size of the step.

There is a risk associated with updating the weights as shown above, as there is no control for exploding weights from exploding gradients. One way to avoid this, is to introduce a regularization parameter, λ, which scales the weights. The weights are thus updated in the following way:

$$w_{jk}^l \leftarrow w_{jk}^l - \eta(\delta_j^l a_i^{l-1} + \lambda w_{jk}^l)$$
$$b_j^l \leftarrow b_j^l - \eta \delta_j^l,$$

which is appropriate when using Regular stochastic gradient descent. From II A 2 we split the data into batches, thus when using Stochastic gradient descent, we have to divide the weights correction with the batch size:

$$w_{jk}^l \leftarrow w_{jk}^l - \frac{\eta}{b_{size}}(\delta_j^l a_i^{l-1} + \lambda w_{jk}^l)$$
$$b_j^l \leftarrow b_j^l - \eta \delta_j^l.$$

Work in progress: I will distribute the text into appropriate sections above after writing down my string of thoughts in the following.

We define our general NN to have L layers in total for which the first layer is the input layer and the last layer is the output layer. This means that we have $L-2$ so-called hidden layers. The input layer must have the same numbers of nodes as there are features in the training data and the last layer must have the same number of nodes as there are categories associated with the target values. For instance in the a case of binary classification, meaning true or false, 1 or 0, etc. we could simple have one output node which can take value in the range $[0,1]$ and thus be rounded of to either 0 or 1 for the prediction. In another case of the MNIST data set where we want to classify handwritten number between 0 and 9, we would instead chose to have 10 output nodes, one for each possible number $\{0,1,2,3,4,5,6,7,8,9\}$.

$$Z^L = W^L a^{L-1} + B^L$$
$$a^L = \sigma(Z^L)$$

Lets assume that we simple have on node in each layer. We choose the cost function to be the mean squared error

$$C = (y-t)^2 \tag{6}$$

We should choose the weights and bias such that C is minimized. How do we do this? The idea is the find the gradient, for which changing the weights and biases in a small step in the direction of the negative gradient, we will lower C. The gradient with respect to its weights is ...

C. Logistic regression (not sure where to put it

Logistic function definition

$$p(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t}$$

We look at the binary case where $y_i = 0$ or $y_i = 1$. We define polynomial model on order n as

$$\hat{y}_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2, \ \dots \ , \beta_n x_i^n$$

We define the probabilities of getting $y_i = 0 \vee 1$ given a input point x_i and beta values $\boldsymbol{\beta} = (\beta_0, \beta_1, \ \dots \ , \beta_n)$

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{e^{\hat{y}_i}}{1+e^{\hat{y}_i}}$$
$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta})$$

We want to choose the parameters $\boldsymbol{\beta}$ such that we maximum the likelihood of saying the observed data $\mathcal{D} \in \{x_i, y_i\}$, that is the Maximum Likelihood Principle (MLE) principle. The likelihood of having the observed data $\mathcal{D}$ given $\boldsymbol{\beta}$

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} \Big(p(y_i = 1|x_i\boldsymbol{\beta})\Big)^{y_i} \Big(1 - p(y_i = 1|x_i, \boldsymbol{\beta})\Big)^{1-y_i}$$

In order to minimze computations we can define the log-likelihood as

$$\log\left(P(\mathcal{D}|\boldsymbol{\beta})\right) = \sum_{i=1}^{n} y_i \log\Big(p(y_i = 1|x_i\boldsymbol{\beta})\Big) + (1-y_i)\log\Big(1 - p(y_i = 1|x_i, \boldsymbol{\beta})\Big),$$

and in order to get the cost function on the more conventional form as minimization problem we define the cost function as the negative log-likelihood:

$$C(\boldsymbol{\beta}) = -\log P(\mathcal{D}|\boldsymbol{\beta}). \tag{7}$$

We find the gradients

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\sum_{i=1}^{n} \begin{pmatrix} y_i - p(\hat{y}_i) \\ x_i y_i - x_i p(\hat{y}_i) \\ \vdots \\ x_i^n y_i - x_i^n p(\hat{y}_i) \end{pmatrix} = -X^T\Big(\mathbf{y} - P(X\boldsymbol{\beta})\Big)$$

For the L2 regularization we add the L2-norm to the costfunction as

$$C(\boldsymbol{\beta}) = -\log P(\mathcal{D}|\boldsymbol{\beta}) + \lambda||\boldsymbol{\beta}||_2^2, \qquad \lambda > 0$$

where

$$\lambda||\boldsymbol{\beta}||_2^2 = \lambda\sum_{i=1}^{n}\beta_i^2 \quad \Longrightarrow \quad \frac{\partial \lambda||\boldsymbol{\beta}||_2^2}{\partial \boldsymbol{\beta}} = 2\lambda\boldsymbol{\beta}$$

This gives the full gradient expression with L2 regularization as

$$\frac{\partial C(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -X^T\Big(\mathbf{y} - P(X\boldsymbol{\beta})\Big) + 2\lambda\boldsymbol{\beta}$$

III. IMPLEMENTATION

The code can be found on this Github address.

A. SGD algorithm

Result: Stochastic gradient descent algorithm
Initialize layers, weights, biases;
while *epoch* < *epochs* **do**
  Create batches;
  while *batch* < *batches* **do**
    Fetch X, y data in batch;
    Find gradient;
    Update velocity;
    Update Θ;
  end
end

B. Neural network

As done fore SGD, we created a class to run all Neural network calculations. Each instance of the class must specify the structure of the network such as number of hidden layers and number of nodes per hidden layer as well as defining all functions such as the cost function and activation function. Based on the st

Result: Neural Net
Initialize layers, weights, biases;
while *epoch* < *epochs* **do**
  Create batches;
  while *batch* < *batches* **do**
    Set first layer to batch;
    Feed Forward;
    Back propagate;
    Update Weights and Biases;
  end
end

C. Layers, weights and biases

TensorFlow allows for manually adding a layer, defining how many nodes, and the dimensions of the input data, as well as the activation function for that layer. We chose however to set up our hidden layers as exact copies, meaning that you define a number of layers and nodes, and then all the hidden layers have the same number of nodes, and in fact the same activation. Lastly one can choose a different activation for the last layer, if needed. The weights and biases follow the same structure, to ensure that the dimensions of the tensor elements, i.e the weights for a given layer or bias for a given layer, are equal to said layer.

INSERT HERE ABOUT SCALING BIAS AND WEIGHTS, RELU/LEAKY$_RELUneedsthis$

D. Activation functions

We defined 4 different cost functions to use in hour network, namely the Sigmoid, RELU, leaky RELU and SOFTMAX. Sigmoid is defined as

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

RELU is defined as

$$f(x) = x^+ = max(0, x)$$

Leaky RELU is defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{if otherwise} \end{cases}$$

SOFTMAX is defined as

$$f(x) = \frac{e^x}{\sum_j e^{x_j}}$$

E. Practical implementation

1. *Learning rate*

Using a constant learning rate η can sometimes lead to either too long computation, or overstepping along a slope. A way to correct this is to adjust the learning rate η by how many epochs have been run. We created our own η decay function and it is given as :

$$\eta = \eta_0 A \cdot \sigma_s(k \cdot (t_D - \xi))$$

Here we have an initial η_0 value, an amplitude A, a drop time t_D dictating the half time of decay, epoch ξ and a steepness parameter k. σ_s is the Sigmoid activation function, as defined in the section above. The amplitude is a correction parameter to ensure that the learning rate at epoch 0 is the initial learning rate η_0. Thus we define a piece wise function A given as

$$A = \begin{cases} 2 & \text{if } \xi = 0 \\ 1 + e^{-kt_D} & \text{if } \xi > 0 \end{cases}$$

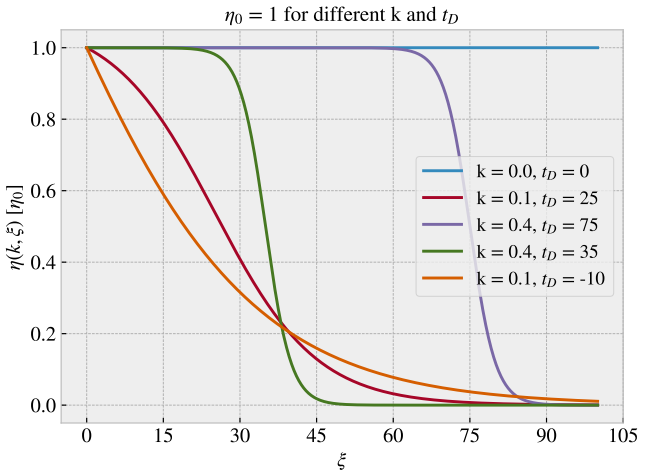


Figure 3: η decay as function of steepness k, drop time t_D, amplitude A and epoch ξ. Drop time increase by three for each iteration of k value.

By fine tuning the steepness and the drop time, these logistic functions can take shape of exponential decay or constant functions.

The learning rate can also be extended by means

of being scaled by the batch size, i.e $\eta \rightarrow \eta/n$. This makes the learning rate much smaller, but it allows us to avoid derivations with very small numbers, and thus helps avoiding numerical errors better.

2. *Handling vanishing and exploding gradients*

Another way to speed up calculations is to ensure that the gradient does not disappear. Vanishing gradients can in extreme cases lead to no training, i.e no updating of weights and biases, and so we added two criteria to mitigate this. First we require the gradient to always be larger than a given tolerance, default to 10^{-8}. Second, we require the gradient to be finite. Sometimes the gradient explodes, leading to extreme changes in the weights and biases, and so we implemented a check to ensure that the gradient is always finite. That way we do not get NaN or inf in our calculations. It is worth mentioning that we just as well could have implemented the so called gradient clipping method, or other methods to control exploding gradients.

3. *Cost functions*

We used two cost functions for the purpose of the datasets we wanted to look at, Mean squared error and Cross entropy. Note from equation 6 that the actual definition of the mean squared error has a factor $1/m$ where m is the number of elements in the dataset y. As mentioned in the section about learning rate, this factor is implemented in the learning rate, which amongst other things allows AutoGrad, a numerical derivation package for Python, to handle the derivation easier.

4. *Scaling*

The need to scale the data and the type of scaling varies from data set to data set. For the data set created to predict the Franke's function, we used uniformly distributed values between 0 and 1, and so here there is no need to scale. However, the breast cancer data set has entries spanning from 0 to 4000, so standard scaling is necessary here. Standard scaling subtracts the mean from the data set and divides on the standard deviation.

F. Data sets

1. *Franke's function*

In the rapport we will be doing both regression and classification analysis. For the regression we will be using Franke's function. The Franke's function, f(x,y), is a analytical function often used in machine learning. The function is a weighted sum of four exponentials which we will study in the domain $x, y \in [0,1]$. The function reads as follows

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+ \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &- \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \end{aligned} \tag{8}$$

For more on the Franke's function, see [2].

2. *Breast cancer*

For classification we will first be using scikit learns breast cancer data. The data is structured such that each point of the input variables are different attributes (30 in total) of a tumor and the output data is a binary, malignant or benign. Given the number of attributes and the binary nature of the data, the network must contain 30 features and 1 category. The data set includes, in total 569 datapoints.

3. *MNIST*

The MNIST data set contains 1797 images of handwritten number between 0-9. Each image has a resolution of 8×8 pixels. AN example of the data is shown in figure 4.

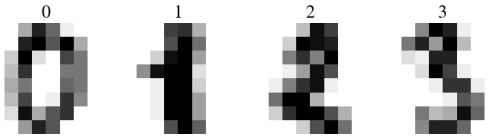


Figure 4: A set of four datapoints from sckit learns breast canser data. Each point with the corresponding value written above.

The MNIST output data, or target is original structured as a list with 1797 elements, where each element is a number between 0-9. This means that the Network still would have 9 categories. As we want the output to be a binary classification case, we restructure the data such that each number is replaced with a 10 dim array. Each array is filled with binary values, where the index

equal to the original number is set to 1. For example:

$$3 \rightarrow \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \quad 7 \rightarrow \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix} \quad or \quad 5 \rightarrow \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix} \tag{9}$$

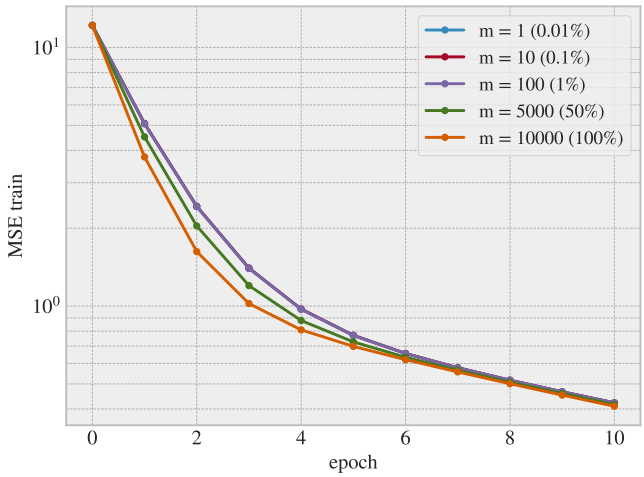


Figure 5: MSE on train data for SGD regression on Franke's function (default settings) with fixed learning rate $\eta = 0.1$ and varying batch size m. The percentage denotes the batch size relative to the number total number of data points.

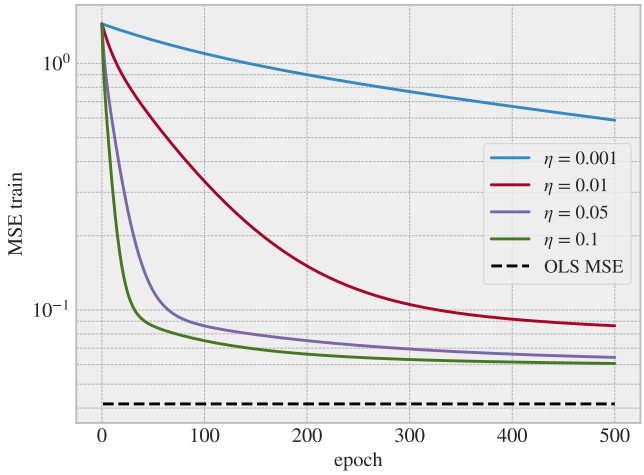


Figure 6: MSE on train data for SGD regression on Franke's function (default settings) with full gradient descent ($m = 10^4$) and varying learning rate η. The black-dotted line indicates the MSE for OLS regression.

IV. RESULTS AND DISCUSSION

A. Validation

1. SGD

Note: Say that we choose same seed for repeated experiments for a fair comparison...

We begin by validating and testing the implementation of our SGD method. For this we use Franke's function, which we analyzed in project 1 (REFERENCE). We use a default 100×100 meshgrid (10^4 data points), a complexity of $n = 5$, referring to the highest polynomial order, and add normal distributed noise to the output with standard deviaion $\sigma = 0.2$. When comparing train and test results we use a default split of 80% training data and 20% test data.

First, we take on analyzing non-momentum OLS-gradient SGD. We investigate the convergence of the model as a function of epochs regarding the hyperparameters: Batch size m and learning rate η. We do this by keeping one parameter constant and changing the other respectively. For this we use the whole data set, and thus the MSE is for the triaining data. The result for varying batch size and fixed learning rate is shown in figure 5 and the results for varying learning rate and constant batch size is shown in figure 6

From figure 5 we see that the convergence rate is only slightly affected by the batch size, with a small benifit of having the biggest batch size (this was even less clear with other seeds). But the difference becomes negligible already at the 10^{th} epoch onwards. However, from figure 6 we see that the learning rate has a much more distinct impact on the convergence rate. To get a better understanding of the influence from the learning rate we compute the final MSE after 10^3 epochs for both training and test data (default split: 0.2) as a function of the learning rate with full gradient descent (see figure 7).

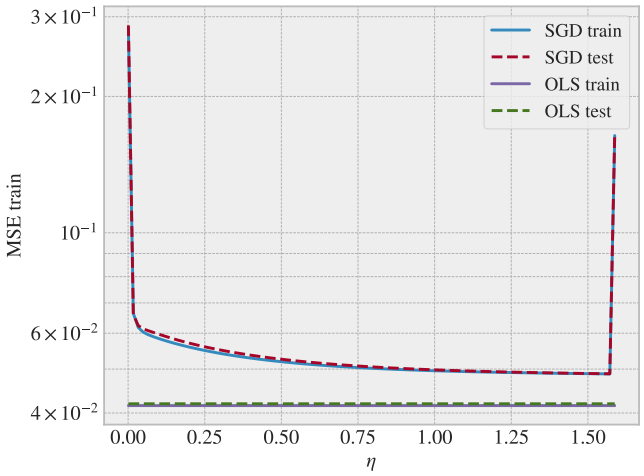


Figure 7: MSE on train and test data for SGD regression on Franke's function (default settings) with full gradient descent ($m = 10^4$) for 10^3 epochs as a function of learning rate η. This is compared to OLS regression for train and test data seen in the bottom of the plot.

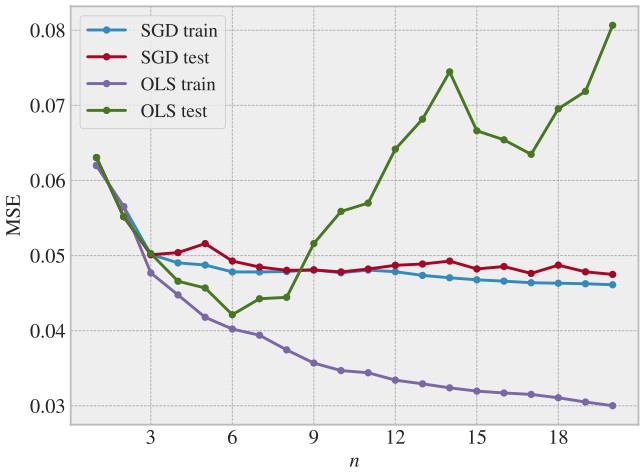


Figure 8: Comparison of SGD and OLS on Franke's function for varying model complexity n. SGD uses full gradient descent with 10^4 epochs and learning rate $\eta = 0.1$. The data consists of 400 data points.

From figure 7 we see that the best performance from the SGD is found with a quite high learning rate. We compute the minimum to be

$$\arg\min_{\eta}(\text{MSE test}) = 1.57 \pm 0.02$$

With a resolution of 0.02 in the leaning rates, the minimum point occurs on the second last index of the tested learning rates, while the last index correspond to the peak on the right-hand side of the plot. Thus we see an apparently non-continuously behaviour (considering our resolution) where the best learning rate comes just before a critical limit where the learning rate will yield poor convergence. However, when running similar test on other data sets we see that this limit changes quite a lot, and thus one should be careful of using a to high learning rate. For the following results we use $\eta = 0.1$ as a safe bet.

We know compare OLS and SGD for different model complexities. Notice hat we used $n = 5$ up to this point but know we take on a smaller data set of 20×20 (400 data points) and $n = [1, 20]$. For SGD we use full gradient descent and 10^4 epochs. We continue to use a split of 0.2 and noise $\sigma = 0.2$. The results are shown in figure 8

As expected from earlier results, we see that OLS outperformes SGD on the low model complexity which we have already seen for $n = 5$ in figure 7. However, as the model complexity increases the OLS regression results in overfitting and thus the test MSE increases considerable. SGD does not really prone to overfitting in this particular case, and we see that the MSE is steady, actually slightly decreasing, for increasing model complexity.

Until now we have mostly been using full gradient descent as we saw (from figure 5) that this gives a slightly better convergence on a short scale of epochs. When introducing the Neural Network we expect to deal with far bigger gradients (for weights and biases) which will result in heavier matrix operations. For smaller batch sizes we reduce the size of these matrix operations with a cost of having to run through a for-loop for more iterations. In order to get an idea of this computational balance of for-loop iteration and matrix operations we time the algorithm for increasing sizes of the design matrix. We fix $n = 10$, resulting in 66 features and increase the size of the meshgrid from 100×100 (10^4 data points) to 1000×1000 (10^6 data points) while measuring the time pr. epoch. averaged over 10 epochs. We use $\eta = 0.1$ and initialize from same random seed each time. The results are shown in figure 9

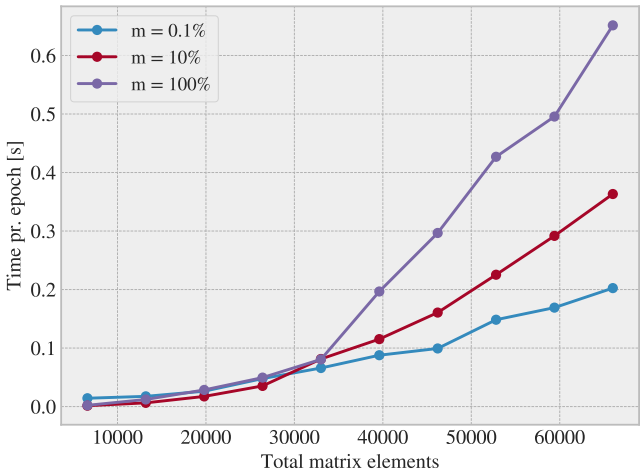


Figure 9: Timing of SGD as a function of the total number of elements in the design matrix for different batch size (given as a percentage of the total number of data points). While keeping $n = 10$, corresponding to 66 features, we varied number of data points in the interval $N \in N = [10^4, 10^6]$ and measured the computation time pr. epoch. We ran each SGD-regression for 10 epochs staring from the same random seed. We see that small mini batches gets computational beneficial for big matrices.

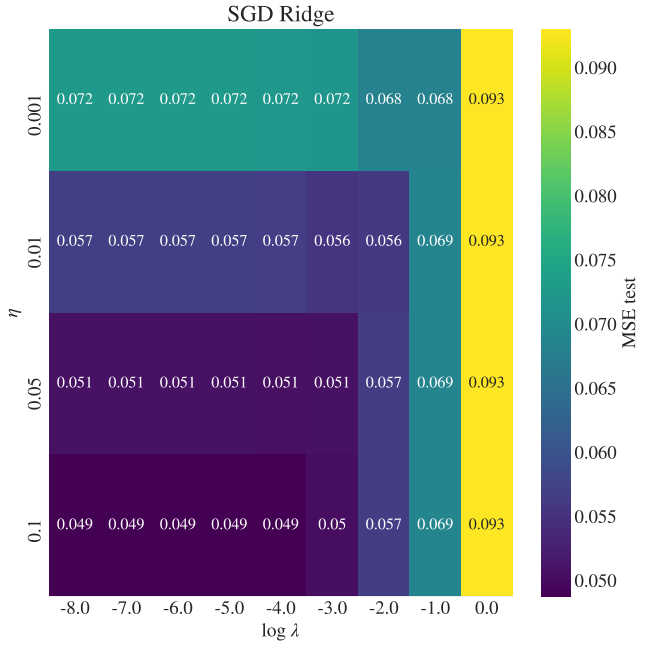


Figure 10: MSE test for SGD with Ridge gradient for different combinations of learing rates η and Ridge regulazation parameter λ on Franke's function. We use the default data set with full gradient descent.

From figure 9 we see that for big matrix sizes, the smallest batch size yields the lowest computation time. For smaller matrices, the small batches actually gives a slightly longer computation time, and the intersection between these domains is visually determined to be around 30,000 matrix elements.

We also want to compare Ridge regression and SGD with Ridge gradient computation. In order to do this we begin by studying the optimal hyperparameters for SGD with with Ridge gradien. When fixing the batch size to full gradient descent we are left with learning rate η and the regularization parameter λ as the main hyperparameters. We go back to using the default data set, as stated in the beginning of this section, and compute the MSE on the test data for different combinations of η and λ. The result is shown in figure 10

From figure 10 we see that the best MSE test is found at the highest tested learning rate $\eta = 0.1$ and the lowest regularization values $\lambda \leq 10^{-4}$. Looking back to figure 7 one could theorize that we have a similar dependence on the learning rate as we had with the OLS gradient. This we fix the learning rate to $\eta = 0.1$ and compute the MSE on test date (default data) for different λ-values using 10^4 epochs for the SGD. The result is shown in figure 11.

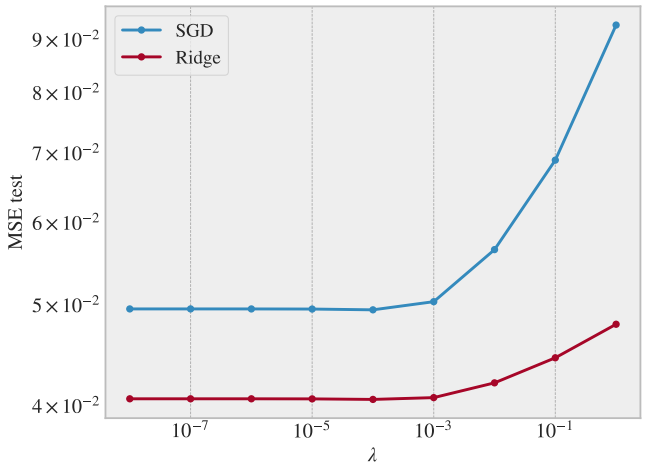


Figure 11: MSE test for SGD (Ridge gradient) and Ridge regression as a function of λ on Franke's function. We used 10^4 epochs and $\eta = 0.1$ with full gradient descent on the default data set.

From figure 11 we see that the lowest MSE is achieved for decreasingly values of λ this reducing the comparison to the one of OLS.

Finally we introduce the SGD with momentum. We use full gradient descent on the default data and investigate the convergence rate for different learning rates and momentum paramter γ. The result is shown in figure 12.

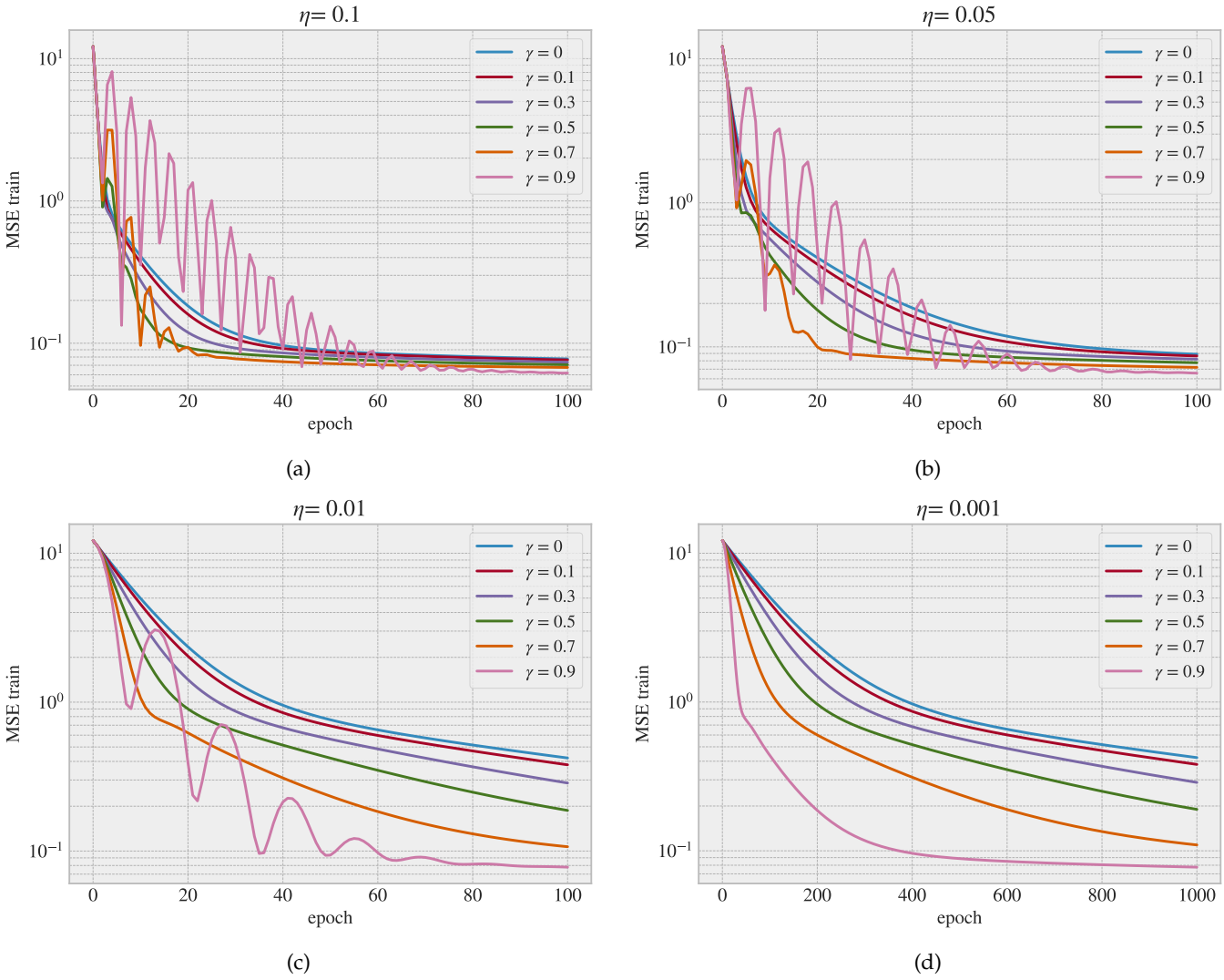


Figure 12: MSE train for different momentum parameters γ and learning rates η as a function of epochs. We used full gradient descent on on the default Franke's function data set.

From figure 12 we see that momentum parameter has a quite significant influence on the convergence rate. For the small learning rate $\eta = 0.001$ (see 12d the highest tested $\gamma = 0.9$ showcased a clear advantage. However, for bigger learning rates, $eta = 0.1$ (figure 12a the higher γ's gave an oscillation MSE and $\gamma = 0.9$ converged the slowest, meaning that it decreases more slowly. But as the oscillations settles down we actually get the lowest MSE after 100 epochs with $\gamma = 0.9$ for all learning rates tested. When increasing γ to 0.99 we see the same effect but on a much bigger scale of epochs.

2. *Neural Network*

In this part we study our implementation of the Neural network. As a starting point we choose three logic gates: AND, OR, XOR (exclusive or) which is represented as the data points in table II

Table II: Logic gates: AND, OR and XOR

Input		Output		
x_1	x_2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

When performing standard OLS we get the predic-

tions shown in table III, where we get 100% accuracy on the AND and OR gate but only 75% on the XOR gate. When displaying the data points graphically it is clear to see that the data points of the XOR-gate cannot be separated by a single straight line (see figure 13. Hence it is clear that OLS will never give a satisfactory prediction on this nonlinear problem.

Table III: OLS predicitons on the logic gates: AND, OR and XOR

Gates	Predictions	Accuracy
AND	$y = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$	100%
OR	$y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$	100%
XOR	$y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$	75%

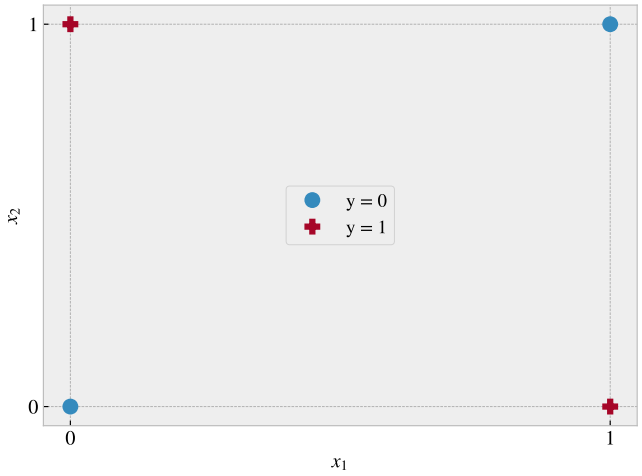


Figure 13: Data points for the XOR-gate. Showcases grpahical how we cannot seperate the 0's (blue) and 1's (red) by a single straight line

Hence we now use the XOR gate as benchmark for our neural network. We choose a simple architecture with 1 hidden layer containing 4 hidden nodes. Notice that it is possible to solve the XOR-problem with only 2 hidden nodes, but we found that having 4 nodes made the training simpler. We use full gradient descent, $\eta = 1$ and $\lambda = \gamma = 0$ with sigmoid as activation function and cross entropy as the cost function. We manage to get a perfect fit, 100 % accuracy, on the training data after roughly 90 epochs. The learning accuracy development is shown in figure **??**.

Next we study regression on the Franke's Function from section III F 1 data using our Neural Network. We used the same initial conditions for the Neural Network as with the linear regression models[1], and got the following results:

Table IV: Table with R2 and MSE score for our Neural Net, OLS regression and Ridge regression with $N = 30$, $\sigma = 0.2$, $\lambda = 10^{-4}$, $\gamma = 0$, $\eta = 10^{-1}$, 3 hidden layers, 42 nodes, batch size $= 25$ and at the most 400 epochs.

Score	**NN**	**OLS**	**Ridge**	**TensorFlow**
MSE	0.032	0.034	0.034	0.062
R2	0.844	0.838	0.841	0.489

Here we see that our Neural Network performs as well as OLS and Ridge regression on the Franke's function data, and even Tensorflow. This is however the optimized version, and needed two hyperparameter grid searches, as shown in the images below.

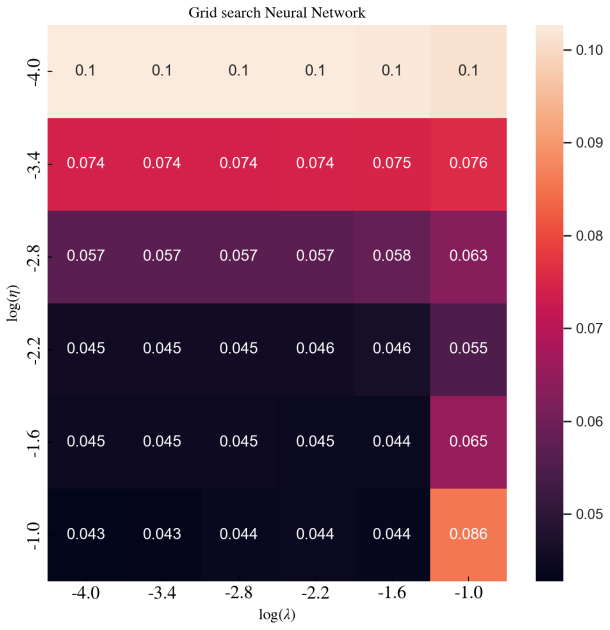


Figure 14: Heatmap showing MSE score for different hyperparameters $\lambda \in [10^{-4}, 10^{1}]$ and $\eta \in [10^{-4}, 10^{1}]$. Max number of epochs is 400, batch size 25, 2 hidden layers and 70 hidden nodes

Here we find the optimal hyper parameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. The regularization constant λ acts in the same way as it does for Ridge regression, which further validates our findings of the MSE, given that λ is so small. The next hyper parameter grid search is for number of hidden layers and number of hidden nodes per hidden layer.

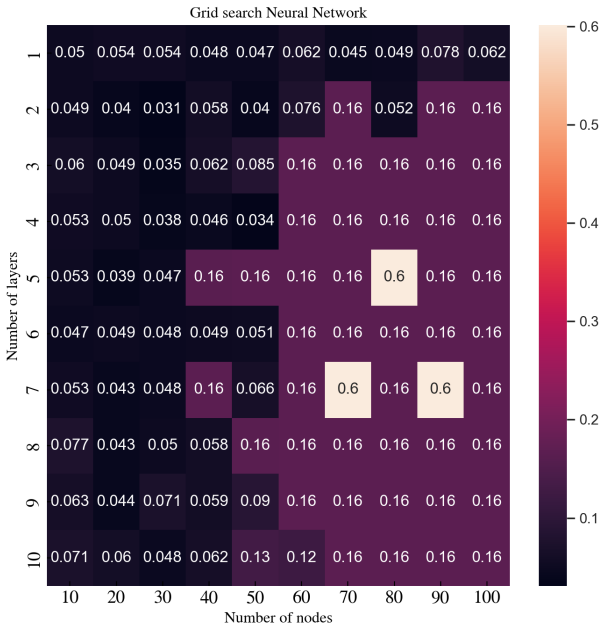


Figure 15: Heatmap showing MSE score for different hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Max number of epochs is 400, batch size 25, hidden layers $\in [1, 10]$ and hidden nodes $\in [10, 100]$

From this image we see that 4 hidden layers and 50 nodes per layer is the optimal choice for the Franke's function data. The grid is spaced with a 10 node and 10 layer increase for each iteration, and this could mean that the actual lowest combination is somewhere in between say 40 and 50, or 50 and 60 nodes. The following image show that there are combinations with lower MSE:

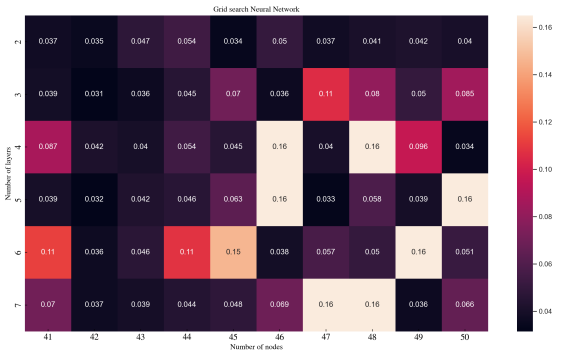
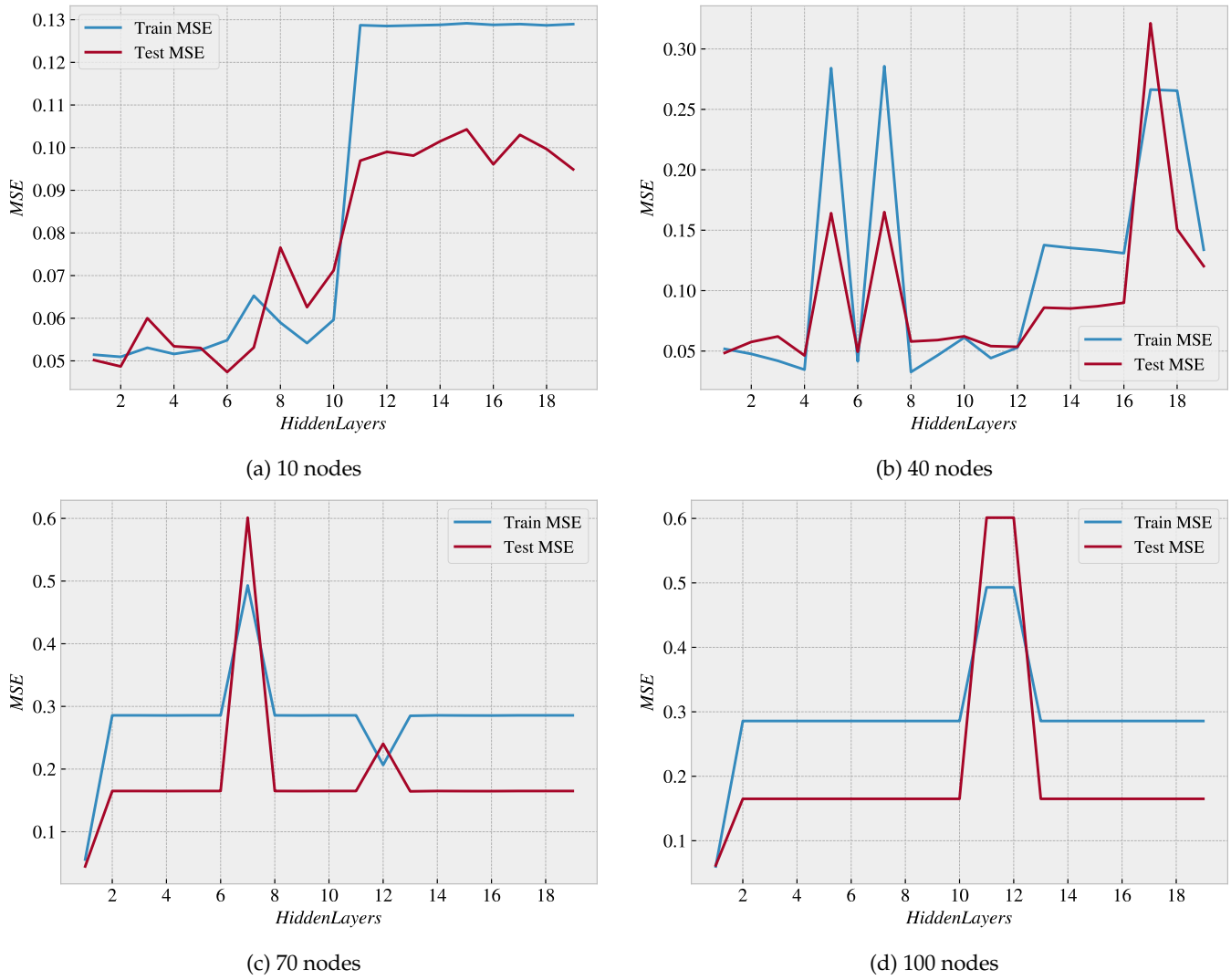


Figure 16: Heatmap showing MSE score for different hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Number of epochs 400, batch size 25, hidden layers $\in [2, 7]$ and hidden nodes $\in [41, 50]$

Here we see that upon further inspection with a much smaller and specified set of nodes, we find the ideal combination of nodes and layers is in fact 3 layers and 42 nodes.

There are certain combinations of the two hyper parameter grid searches that gives a "much" larger MSE score than the optimal parameter combination, and so the images below thus checks if the network is overfitting. If there was any overfitting we would expect the test MSE to diverge from the train MSE for an increase in layers. But as the images show, we see little to no signs of overfitting. Given these results we feel confident that our network is good enough to solve more complex problems, such as breast cancer classification and number recognition.



(a) 10 nodes

(b) 40 nodes

(c) 70 nodes

(d) 100 nodes

Figure 17: MSE as function of hidden layers for both training and test data. Top left corner has 10 nodes, top right has 40 nodes, bottom left has 70 nodes and bottom right has 100 nodes. We also have $\eta = 0.1$, $\lambda = 10^{-4}$, batch size $= 25$ and at most 400 epochs.

With these hyperparameters fixed, we plotted our prediction on a new dataset, with the same seed, and got the following results

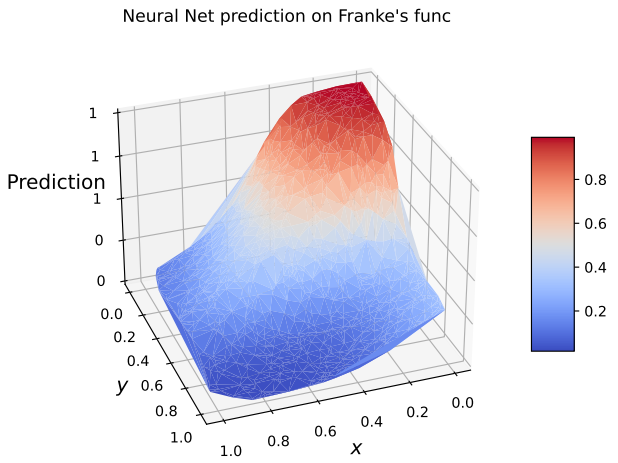


Figure 18: Heatmap for hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Max number of epochs 200, batch size 25, 2 hidden layers and 30 hidden nodes.

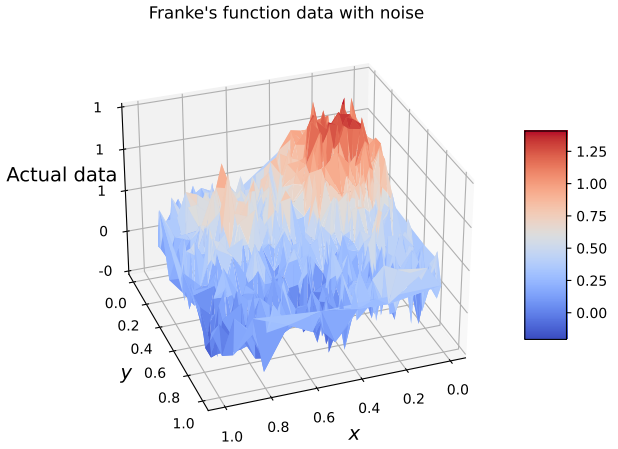


Figure 19: Heatmap for hyperparameters $\lambda = 10^{-4}$ and $\eta = 10^{-1}$. Max number of epochs 200, batch size 25, 2 hidden layers and 30 hidden nodes.

Here we see that our network makes a very good fit to the noisy data. This is further evidence of good validation of our network, and expected, given the MSE and R2 score from table IV A 2.

B. Franke function

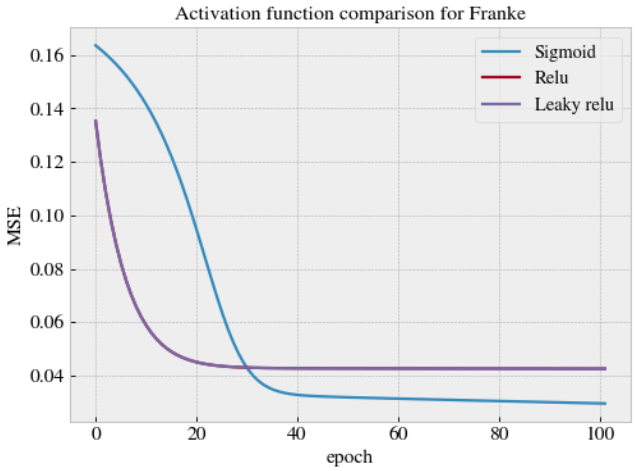


Figure 20: Comparison between sigmoid(blue), relu(red) and leaky relu(purple) as activation functions for calculations on the Franke data. The calculations were done with 2 hidden layers with 30 nodes each. γ and λ values were both set to 0 and η was set to 0.01.

C. Breast cancer classification

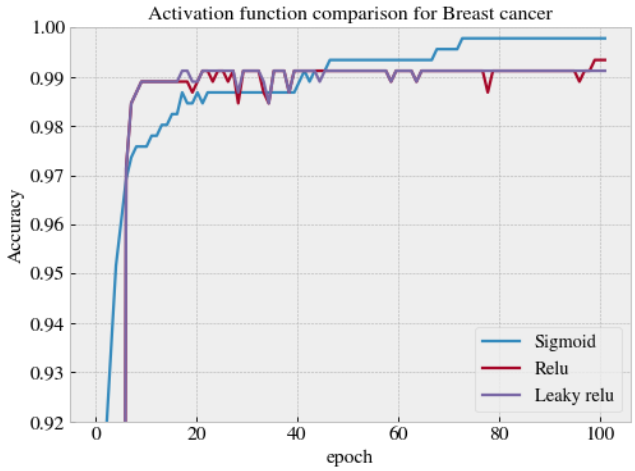


Figure 21: Comparison between sigmoid(blue), relu(red) and leaky relu(purple) as activation functions. The calculations were done with 2 hidden layers with 30 nodes each. γ and λ values were both set to 0 and η was set to 0.01.

From the figure 21, we observe that sigmoid produced the best values, although only slightly. In addition we see that relu and leaky relu produce rather similar accuracy. As expted, we see that the general trend of the accuracy produced from relu is sligthly more stable than

that produced from leaky relu. One interesting observation is that relu and leaky relu quicker reaches their maximum accuracy. They both reach maximum accuracy after 8 epochs, where as sigmoid required almost 70 epochs.

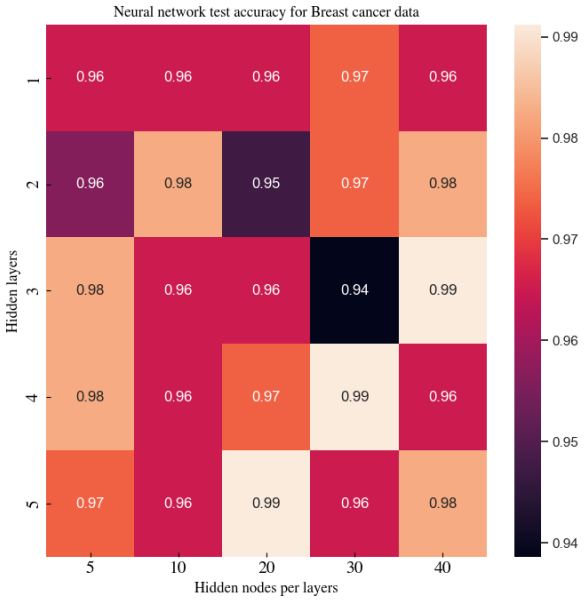


Figure 22: Heatmap for different values of hidden layers and nodes per layer. The calcluation were done with $\eta = 0.01$, $\gamma = 0$, batch size 25 and run for a maximum of 200 epochs.

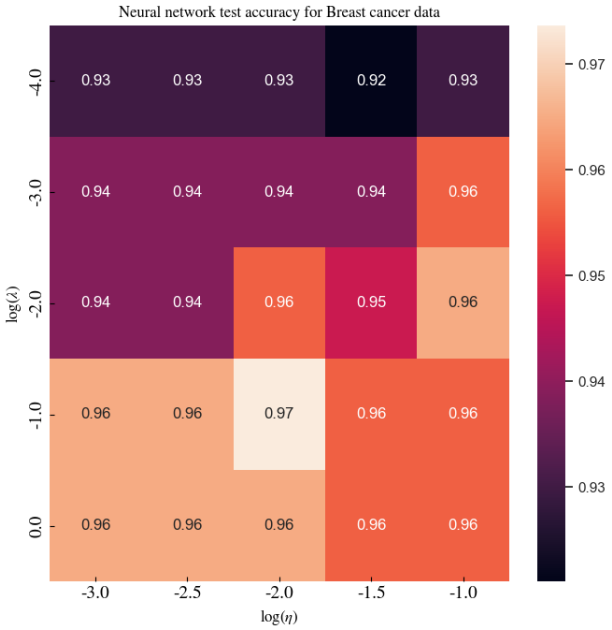


Figure 23: Heatmap for hyperparameters $\lambda \in [10^{-4}, 1]$ and $\eta \in [10^{-4}, 10^{-1}]$. Number of epochs 200, batch size 25, 2 hidden layers and 30 hidden nodes.

	Neural network	**Scikit learn**
Accuracy	97 %	96 %

Table V: Comparison between the Neural network and Scikit learn. Both with 2 hidden layers with 30 nodes each, batch size equal to 50 and with a maximum of 100 epochs.

D. MNIST classification

As stated in section (INSERT SECTION) we will be comparing our Neural Network to our logistic regression solver. All calculations and comparisons will be done using the MNIST data set.
By running a grid search for the hyper-parameters η and λ we find the grid shown in figure (25). The training of the network was done on training data and the accuracy test was done on test data.

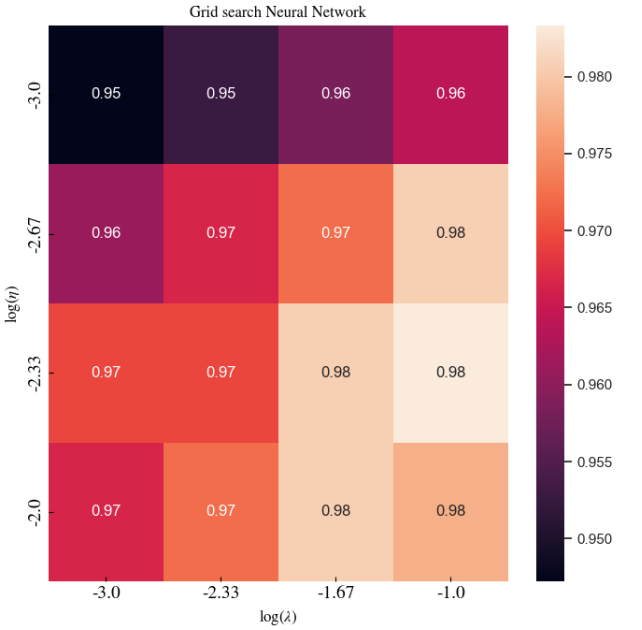


Figure 24: Neural Network heatmap for hyperparameters $\lambda \in [10^{-3}, 10^{-1}]$ and $\eta \in [10^{-4}, 10^{-2}]$. Maximum number of epochs 200, batch size 35, 2 hidden layers and 20 hidden nodes.

From (25) we see that the optimal values for our Neural Network are $\lambda = 10^{-2.33}$ and $\eta = 10^{-2}$.
By doing the same grid search for our logistic regression solver, we find

	Nerual network	Logistic regression	Scikit
Accuracy	98%	93 %	95%

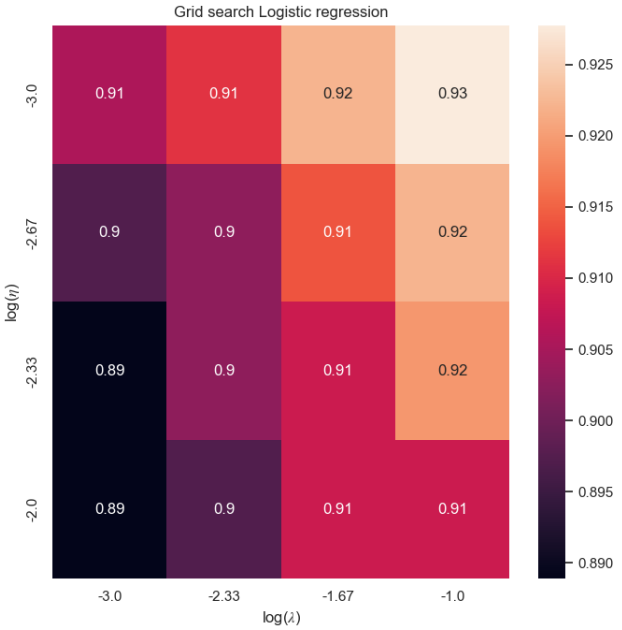


Figure 25: Logistic regression heatmap for hyperparameters $\lambda \in [10^{-3}, 10^{-1}]$ and $\eta \in [10^{-4}, 10^{-3}]$. Number of epochs 200, batch size 35, 2 hidden layers and 20 hidden nodes.

Comparing our Neural Network with our logistic regression solver and Scikit learns logistic regression we find the following results

Figure 26: Digits in test data with the Neural Networks predicted values. Green digits indicate correct prediction, red indicate incorrect. The Neural Network was trained with 2 hidden layers with 35 nodes each, batch size 50, $\eta =$, $\lambda =$ and after 50 epochs.

Figure 27: 64 digits from the test data with the Neural Networks predicted values. Green digits indicate correct prediction, red indicate incorrect. The Neural Network was trained with 2 hidden layers with 35 nodes each, batch size 50 and after a maximum 200 epochs. The hyper-parameters were chosen from a grid search.

V. CONCLUSION

We are best. □

REFERENCES

[1] Frette et al. *Project 1 on Machine Learning*. URL: https://github.com/Gadangadang/Fys-Stk4155/blob/main/Project%201/article/Project_1_current.pdf. (accessed: 10.11.2021).

[2] Morten Hjort-Jensen. *Project 1 on Machine Learning*. URL: https://compphysics.github.io/MachineLearning/doc/Projects/2021/Project1/pdf/Project1.pdf. (accessed: 10.11.2021).

[3] Tesla Inc. *Tesla Vehicle Safety Report*. URL: https://www.tesla.com/VehicleSafetyReport. (accessed: 25.10.2021).