FYS-STK4155 - Project2 TITLE SHOULD BE DESCRIPTIVE

Gard and Are*
Department of Geoscience, University of Oslo

David Andreas Bordvik
Department of Informatics, University of Oslo
(Dated: November 15, 2021)

Background: The context needed to understand our paper

Purpose: What is the purpose of our paper?

Method: The methods used

Results: Summarize our results

Conclusions: Describe our conclusions

INTRODUCTION

Feed-Forward Neural Networks takes the idea of connecting artificial neurons in layers to create a fully connected network. Information flows in a single direction through the network, from the input layer to the evaluation of the input in the output layer. A Feed-Forward network can be used to approximate a function. The network does this by learning which parameters, given some input, results in the closest representation of the correct output [**?**].

The **Universal approximation theorem** states that a Neural Network can approximate any function with as few as a single hidden layer to any precision [**?** **?**]. Hence, any feedforward network with a single hidden layer is in theory ample to represent any function, due to their underlying universality [**?**]. For this project, we will use the **Universal approximation theorem** as motivation to develop our own Feed-Forward Neural Network code to study both classification and regression.

When studying regression using our developed Feed-Forward Neural Network code, we will return to the terrain data used in Project 1 when studying polynomial fitting with different variants of Least Squares Regression. Moreover, we will compare our findings in this project with the result of our previous project to further study the differences polynomial fitting and gradient based approximations.

Furthermore, we will study classification using the Wisconsin Breast Cancer Data provided by SciKit-learn [**?**]. When studying the accuracy of the Neural Net in the context of classification, we will also develop code for Logistic Regression for comparison.

The following sections will include background theory and proposals to algorithms which will be needed to construct our Feed-Forward Neural Network. All algorithms developed will be discussed in detail, however for the concrete implementation we refer to the GitHub repository linked in the Appendix under the section Source Code.

For clarity, all source code developed has been written using the Python programming language. Results obtained from our own developed code will also be compared to functions from the Machine Learning libraries SciKit-Learn [**?**], PyTorch [**?**] and TensorFlow [**?**].

In the following section, we will derive the theory and develop an algorithm for Stochastic Gradient Descent. Using that algorithm as a basis, we will further derive theory and develop code for our own Neural Network and Logistic Regression. With the developed algorithms, we will study their behavior and skill when used with both the Terrain data from Project 1 as well as the Breast Cancer data provided by SciKit-learn [**?**]. The results obtained from our own developed code, as well as the benchmarks with the previously stated Machine Learning software will be presented in the **Results** section. A discussion of our results will be confined to the **Discussion** section. Finally, the project will be concluded with our final thoughts for the results as well as this project as a whole in the **Conclusions** section.

THEORY AND METHODS

Moving along the gradient

Gradient Descent is an iterative algorithm that minimizes a given function by following its gradient down towards the global minimum. For a given cost-function $C(\beta)$ with predictors β and a hyperparameter η (which will be described in the following paragraph), the process of Gradient Descent can be expressed mathematically as follows

$$\beta_{k+1} = \beta_k - \eta \nabla_\beta \left(C(\beta_k) \right) \tag{1}$$

By inspecting Equation (**??**), it can be seen that given some random initialization of β_0, each step β_k will be closer to the optimal $\hat{\beta}$ than the previous step. This is done by repeatedly calculating the cost function and

* afkvanum@mail.uio.no

taking a step in the direction of its gradient. Moreover, the length of the step taken is decided by the newly introduced hyperparameter η. Where consecutively large values of η results in a large step along the gradient, and a small value of η a small step along the gradient. The learning rate is an essential parameter to ensure that the gradient descent converges, as small values might end up slow convergence and large values result in divergence [**?**].

Another pitfall for gradient based convergence methods is the existence of local minimums. As not all functions are created equal, some functions might feature some unevenness that the gradient descent will mistake as a global minimum. As such, gradient descent on the form as in Equation (**??**) will get stuck in any local minima that it encounters. However, given the case of regression, the MSE cost function is a convex function. As such, there is only one minimum of the function which is the global minimum [**?**]. Thus, when considering regression, there is no risk of the algorithm getting stuck in a local minima.

Introducing stochasticity to gradient descent

By utilizing the entire gradient of the cost function to compute the next step along the gradient, a lot of computational resources are utilized. Especially if one are to consider a large data-set consisting of several datapoints and features. A method to alleviate some of the computational demand of the algorithm is to only compute the gradient for a smaller subset of the data. With Stochastic Gradient Descent, this idea is implemented with a stochastic element, that is, which subset of the data is used is selected at random. The resulting gradient algorithm converges towards the same global minimum as regular gradient descent over the entire data-set, though at a higher pace following an uneven gradient path [**?**].

Another benefit of Stochastic Gradient Descent (SGD) compared to its non-stochastic variant is its ability to exit local minima. As the SGD never reaches a true minimum, a local minima might not be able to contain the algorithm when recomputing the gradient based on a different subset of data. On the other hand, as a consequence of never achieving true minimum, SGD will never return optimal values when compared to non-stochastic gradient descent.

Scheduling the learning rate

As pointed out, the Stochastic Gradient Descent algorithm never achieves a minima, thus in effect the algorithm will always move until a set number of epochs has been reached. To simulate a stopping behavior of the algorithm, a learning rate scheduler which dynamically changes the learning rate can be implemented. In the context of this project, a scheduler which reduces the learning rate over time will be implemented. The effect of reducing the learning rate over time is that the step length taken along the gradient after several epochs is smaller. Furthermore, Stochastic Gradient Descent might benefit from such a dynamic learning rate, as it can be assumed that the algorithm have already closed in on convergence before reaching maximum number of epochs. The scheduler in turn will then be able to dampen all future steps taken, to minimize the effect of the random walk behaviour of the algorithm around the minima. For our SGD algorithm, the learning rate scheduler we have implemented is an inverse scaling similarly to the *invscaling* parameter passed on to SciKit-learn's **SGDRegressor** class. That is, the scheduler updates the learning rate following the equation $\eta = \eta_0/t^{t_0}$ [**?**]. However, in an attempt to reduce the immediate impact of the learning rate scheduler, the first 10 epochs will be completed without scheduling.

Implementing Stochastic Gradient Descent with minibatches

By introducing the concept of mini-batches to Gradient Descent, we have to split our data randomly into mini-batches such that the resulting design matrix is

$$X_{\text{perm}} = \{X_0, X_1, \ldots, X_M\}^T$$

where M represents the number of mini-batches. Moreover, X_i contains randomly drawn rows from X. The random draw can be both with and without replacement. In the case of the prior, some minibatches can be reused while others are skipped in their entirety. Whereas in the latter case, all minibatches have to be used to constitute one epoch. The difference between these implementations of Stochastic Gradient Descent is that the prior version with replacement tends to converge somewhat faster than the version without replacement [**?**]. Furthermore, the target vector t is shuffled in such a way that the rows in X_{perm} still adheres to the same target t_i.

With the construction of X_{perm}, the algorithm moves forward by computing the gradient and moving along its direction one mini-batch at a time. Traversing through all mini-batches or a set number of them constitutes to one epoch. The algorithm just described is then rerun for a specified number of epochs.

The gradient step can be described mathematically as follows in Equation (**??**)

$$\beta_{j+1} = \beta_j - \eta_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta_j) \tag{2}$$

Stochastic Gradient Descent is implemented as pseudocode in Algorithm (**??**)

For clarity, Algorithm (**??**) can easily be extended with a regularization term, such as the l2 regularizer making it comparable to Ridge Regression. In the case of regulariza-

Data: Design Matrix (X), target array (t) and initial guess at predictors θ
Result: Estimated value of the predictors θ
for *epoch in number of epochs* **do**
  for *batch in number of batches* **do**
    $x_i \leftarrow$ X[batch];
    $t_i \leftarrow$ t[batch];
    Compute $\nabla_\theta C(\theta)$ with respect to x_i and t_i;
    $\eta \leftarrow learning_schedule(\eta, \ldots)$;
    $\theta \leftarrow \theta - \eta * \nabla_\theta C(\theta)$;
  end
end
return θ

Algorithm 1: Stochastic Gradient Descent with minibatches and learning rate scheduler

Data: Design Matrix (X), target array (t) and initial guess at predictors θ
Result: Estimated value of the predictors θ
for *epoch in number of epochs* **do**
  for *batch in number of batches* **do**
    $x_i \leftarrow$ X[batch];
    $t_i \leftarrow$ t[batch];
    Compute $\nabla_\theta C(\theta)$ with respect to x_i and t_i;
    $\eta \leftarrow learning\_schedule(\eta, \ldots)$;
    $v \leftarrow \gamma v + \eta * \nabla_\theta C(\theta)$;
    $\theta = \theta - v$;
  end
end
return θ

Algorithm 2: Stochastic Gradient Descent with momentum, minibatches and learning rate scheduler

tion, the computation of the Cost-function gradient have to be updated with the regularization term accordingly. For our Stochastic Gradient Descent implementation, we have used the automatic differentiation library Autograd, with regularization included in the MSE cost function.

Adding momentum to SGD

Equation (**??**) can be generalized with a momentum term as follows

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + \eta_y \nabla_\theta E(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_t$$

Where the momentum parameter $\gamma \in [0, 1]$. The intuition behind the introduction of the momentum parameter is the drag coefficient used in mechanics to describe friction. It's usage is similar when introduced in Stochastic Gradient Descent, as the momentum parameter enables the possibility for the gradient to attain momentum when traversing along a slope. Eventually if the gradient is ascending a gradient, due to the momentum, it might stop and return back down towards the minima. Thus Stochastic Gradient Descent with momentum enables the gradient to traverse through some local minima, while eventually stopping at the global minima. The algorithmic implementation of Momentum Stochastic Gradient Descent can be seen in Algorithm (**??**).

Artificial Neural Network

Writing our own Feed Forward Neural Network

As alluded to to in the introduction, we will develop code for our own Feed Forward Neural Network utilizing minibatches extending our implementation of Stochastic Gradient Descent. Our Feed Forward Neural Network will be implemented as an Multilayer Perceptron with fully connected layers. The Multilayer Perceptron is defined such that it consists of one input layer, an unspecified amount of one or more hidden layers, and finally a output layer [**?**]. Our implementation of a Feed Forward Neural Network is inspired by the popular Machine Learning libraries TensorFlow [**?**] and PyTorch [**?**], with its syntax and usage closely mimicking the aforementioned libraries.

The training of a single minibatch can be summarized as follows. Firstly, one minibatch will be passed forward through the network, computing an output value at each neuron layer by layer. This constitutes the feed forward pass algorithm. Then, as a minibatch has reached the final output layer, an attempt to predict the error with a chosen cost-function is made. Finally, we traverse backwards through the network, calculating the contribution each node made to the output error. This process is known as the Backpropagation algorithm [**?**].

Feed Forward pass

With out Feed Forward pass, information is sent in one direction through the layers of the model. More specifically, in each layer the output from the previous layer is given as input, Moreover, the weights associated with the connection strength between each neuron and a bias term is inputted for each node in the layer. Then, for every node in the hidden layer, an activation function associated with the given layer computes the specified layer's output value. As we are implementing a fully connected layer, this computation can be performed simultaneously for all nodes in a single layer, mathematically written as

$$\boldsymbol{a}^l = f(\boldsymbol{X}\boldsymbol{W}^l + \boldsymbol{b}^l)$$

where the matrices $\boldsymbol{X}$ is the input data, $\boldsymbol{W}$ is the strength of connection weights between neurons and $\boldsymbol{b}$ is a bias vector. The function f is an activation function, which will be described in greater detail in a coming section.

This process is then repeated for each layer constituting the model, until finally the output node is reached. At the output node, depending on whether the Neural Network is to be used for regression or classification, an activation of the input values might be performed. Moreover, if the model is fully trained and ready to make predictions, the Feed Forward pass is what would map an unseen observation x_i to some output value y_i. In other words, this algorithm is both used during training, and for assigning a given input correct output values when in operation.

Our implementation of the Feed Forward pass, is given in Algorithm (**??**) below

Data: Input matrix $\boldsymbol{x}$
Result: Estimated value of the true output $\boldsymbol{y}$
$\boldsymbol{a}^0 = x$;
for *layer* $l = 1, \dots, L$ **do**
  $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a} + \boldsymbol{b}^l$;
  $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$;
end
return $\boldsymbol{a}^L$

Algorithm 3: Feed Forward pass for a Neural Network consisting of Fully Connected Layers

Backpropagation

After the Feed Forward pass has been performed, the output which is returned by the attempt to map some input using with the network is used with the cost function to compute an error [**?**]. This error is then propagated backwards throughout the model to calculate the error gradient. In terms of a regression problem, the Cost function in terms of the weights can be written as

$$C\left(\hat{W}\right) = \frac{1}{2}\sum_{u=1}^{n}\left(y_i - t_i\right)^2$$

where t_i is the target value and y_i is the output of the model. Furthermore, we want to understand how sensitive our cost function is to changes to the weights. As such, we want to define the derivative of the cost function with respect to the weights. More specifically, with note that the superscript determines that we are in the output layer and the subscript which two nodes are connected, we want to compute

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} \tag{3}$$

Equation (**??**) uses the chain rule to compute how a change in the weights first influences the un-activated node value, which again influences the activated node value which finally influences the Cost error itself.

Moreover, the two last terms of the chain rule can be written as

$$\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = f'(z_j^L)a_k^{L-1}$$

Where f' is the derivative of the activation function defined previously. Moreover, by combining these two equation, we can define the error of the layer as

$$\boldsymbol{\delta}^L = f'(\boldsymbol{z}) \circ \frac{\partial C}{\partial \boldsymbol{a}^L}$$

which leads us to the final expression of the derivative of the cost function with respect to the weights as

$$\frac{\partial C}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}$$

With the equations needed to start the algorithm, we want to use the computed error in the current layer to express the error in the previous layer. For a general layer l, the error is defined as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Which can be expressed in terms of the chain rule as the sum

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Where we point out that

$$\delta_k^{l+1} = \frac{\partial C}{\partial z_k^{l+1}}$$

and

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1}a_i^l + b_j^{l+1}$$

with M_l being the number of nodes in the current layer l. This leads us to the equation for the error in the current layer l

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_k^l) \tag{4}$$

Equation (**??**) explains the concept of the error propagating backwards through the layers of the model. The final step of the algorithm is to update the weights and biases of the model, which we have implemented using Stochastic Gradient Descent as follows

$$\begin{aligned} w_{jk}^l &= w_{jk}^l - \eta\delta_j^l a_k^{l+1} \\ b_j^l &= b_j^l - \eta\delta_j^l \end{aligned} \tag{5}$$

where we specify that the layers are sequenced increasingly $(l-1 \rightarrow l \rightarrow l+1)$ and that this is not an algebraically correct equation.

The Wisconsin Breast Cancer Dataset

As briefly mentioned in the introduction, our implemented neural network will also be used in a classification context on the Wisconsin breast cancer data set that comes included with SciKit-Learn [**?**]. The dataset consists of 30 features computed from a sample of a breast mass taken from 569 individuals. The dataset also consists of a diagnosis attribute describing whether the current individual has a benign or malignant tumor; in other words if the tumor is cancerous or not. The spread of the current dataset with regards to the diagnosis attribute is a distribution where 357 tumors are benign, and 212 are malignant.

The overall goal for the classification will be to predict a diagnosis with accuracy above a certain threshold on unseen data. For this, we will in the coming sections first rewrite our Neural Network code such that it can be used for classification, as well as regression. Second, we will develop our own Logistic Regression model based on the previously developed Stochastic Gradient Descent algorithm. With both our updated Neural Network and Logistic Regression model, we will test the models against each other to see which model gives the most accurate diagnosis.

Introducing Logistic Regression

To further validate the fitness of our implemented neural network, we also implemented a logistic regression model. These models are exceptionally well suited for cases where the goal is to assign observations to discrete classes. At the core of this regression model is the output of the probability of the observed data belonging to the class in question.

For a binary classification problem, as the Wisconsin breast cancer data, and 2 arbitrary parameters, the probabilities can be formulated as the sigmoid function, mathematically defined as

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)}$$
$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta})$$

where y_i is defined as the binary target data. For this specific case, we use a dataset with 30 features. The computation of probabilities can therefore be formulated as

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i + \beta_2 x_i + \ldots + \beta_{29} x_i)}{1 + \exp(\beta_0 + \beta_1 x_i + \beta_2 x_i + \ldots + \beta_{29} x_i)}$$
$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta})$$

Furthermore, we want to establish a cost function which will produce a convex plot. This is crucial, as a non-convex plot will create problems when trying to optimize the parameters using stochastic gradient descent. We need to ensure that any local minimizer is also a global minimizer[ref week38.ipynb]. To achieve this, we will opt for using cross-entropy, defined as

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i) - \log\left(1 + \exp(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i)\right)\right) \tag{6}$$

Our aim is to minimize this cost functions with respect to all parameters β for all n observations

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i)}{1 + \exp(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i)} \right)$$
$$\vdots$$
$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_{29}} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i)}{1 + \exp(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i)} \right)$$

For the Wisconsin breast cancer data set, we will define a target vector y, consisting of the binary diagnostic data, a design matrix $\boldsymbol{X}$, and a vector p consisting of the probabilities of each observations, produced by the above mentioned sigmoid function.

The first derivative of the cost function can then be formulated as

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

We have chosen to implement Logistic Regression with both stochastic gradient descent and Newton Raphson´s method. Algorithm (**??**) describes how SGD was implemented. Here we have introduced a learning rate η and

a regularization parameter λ. In this algorithm, all β parameters will be updated after each iteration through a mini-batch.

Data: Design Matrix (X), target array (t) and initial guess at predictors θ
Result: Estimated value of the true predictors β
for *epoch in number of epochs* **do**
  for *batch in number of batches* **do**
    $x_i \leftarrow$ X[batch];
    $t_i \leftarrow$ t[batch];
    $p \leftarrow$ probabilites(x_i, θ);
    Compute $\frac{\partial \mathcal{C}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$;
    $\theta \leftarrow \theta - \eta * (\frac{\partial \mathcal{C}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} + 2 * \lambda * \theta)$;
  end
end
return θ

Algorithm 4: Logistic Regression with Stochastic Gradient Descent and l2 regularization

Another approach for logistic regression is to solve using Newton Raphson´s method. This approach forces us to introduce a term with second derivatives

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}$$

Here, matrix $\boldsymbol{W} = p(y_i|x_i, \boldsymbol{\beta})(1 - p(y_i|x_i, \boldsymbol{\beta})$, is computed with the p values computed in same manner as in the algorithm for stochastic gradient descent.

Data: Design Matrix (X), target array (t) and initial guess at predictors θ
Result: Estimated value of the true predictors β
for *epoch in number of epochs* **do**
  $p \leftarrow$probabilites(x_i, θ);
  $W \leftarrow p(y_i|x_i, \boldsymbol{\theta})(1 - p(y_i|x_i, \boldsymbol{\theta})$;
  $hessian \leftarrow \boldsymbol{X^T W X}$;
  $\theta \leftarrow \theta - (X^T W X)^{-1} \times \frac{\partial \mathcal{C}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$;
end
return θ

Algorithm 5: Logistic Regression with Newton Raphson´s method

Computing the accuracy of a logistic model

Common for both algorithms is the approach in how we compute the accuracy. With all the epochs done, β has reached its final estimation, based on the training data. We compute the probabilities of the observed data in the test set X_{test} with

$$p(y_{test}|\boldsymbol{X_{test}}, \boldsymbol{\beta}) = \frac{1}{1 + \exp -(\beta \boldsymbol{X_{test}})}$$

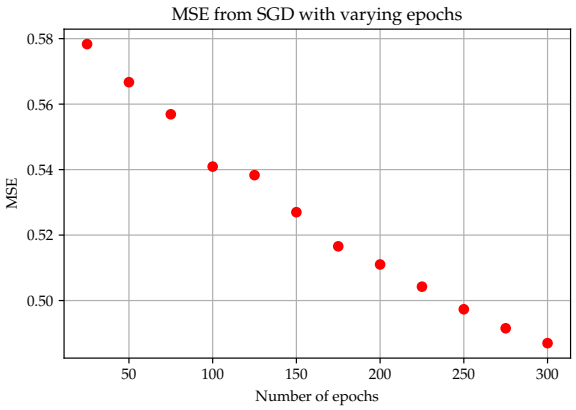


FIG. 1: MSE calculated from Stochastic Gradient Descent optimization of the predictors θ as function of the number of epochs with all other hyperparameters kept fixed

The values are then tresholded, assigning all probabilities > 0.5 to class 1 and all < 0.5 to 0. The accuracy for n samples is then simply calculated by

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{7}$$

Where I is the indicator function, which is 1 if $t_i = y_i$ and 0 otherwise.

RESULTS

When creating the results, we note that all are generated from the Source Code in the appendix. For a further explanation on how to reproduce these results, we refer to the GitHub repository which contains a README explaining how to run the supplied source code.

Stochastic Gradient Descent

Our Stochastic Gradient Descent is implemented akin to Algorithm (**??**). For all results concerning Stochastic Gradient Descent, a complexity of degree 6 is used in the setup of the Design Matrix. This results in 27 individual predictors after scaling and removal of the Intercept, which is in line with the preprocessing performed in Project 1. Moreover for clarity, unless specified the results are generated without a learning rate scheduler. Figure (**??**) plots the MSE computed from our SGD implementation against the number of epochs used.

The following Figure (**??**) plots the MSE as a function of batch_size using the SGD algorithm in **??**.

TABLE I: Runtime in seconds computed for SGD over an increasing number of epochs

#epochs	Runtime [s]
25	8
50	17
75	27
100	38
125	45
150	57
175	65
200	80
225	78
250	79
275	88
300	95

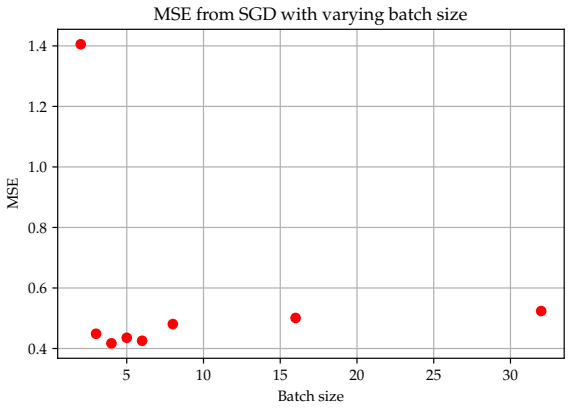


FIG. 2: MSE calculated from Stochastic Gradient Descent as function of the batch size with all other hyperparameters kept fixed

Both Figures (**??**, **??**) are created using a grid search algorithm over different values of η and λ, presented as a heatmap.

Table (**??**) displays MSE results computed from different configurations of the developed SGD algorithm as well as the MSE computed from the Momentum Stochastic Gradient Descent algorithm explained in Algorithm (**??**).

Arificial Neural Net

ay

Optimal parameters

Parameters and model complexity

Performance of activation functions

ay

Effect of weight initialization

ay

Neural Net vs Ordinary Least Squares on terrain data

ay

Robustness of our Neural Net

ay

Logistic Regression

DISCUSSION

Analyzing our Stochastic Gradient Descent implementation

As described in the introduction, all results above are generated using the Terrain data from Project 1 with the same data preprocessing and a fixed degree of 6. The degree was chosen as a compromise between complexity to the data and complexity to the computations. Figure (**??**) shows the epoch dependance of the SGD algorithm measured in MSE. As can be seen in the figure, as the number of epochs are increased, the MSE is reduced. In this specific case, the reduction of MSE tend to closely follow a parabola, though for 100 epochs the MSE is reduced somewhat more than average. As such, the MSE seem to converge somewhere around 0.48, which is the approximate value at 300 epochs. However, though it can clearly be seen that a higher amount of epochs results in a more precise estimation of the predictors, the computational time for each run increases with epochs as seen in Table (**??**). Hence, as a consequence of running some of the computations on an Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz, the number of epochs are constrained to 100 for further computations. Though we acknowledge that 100 epochs are not necessarily the optimal number of epochs if just concerning the MSE.

Inspecting Figure (**??**), it can be clearly seen that the lowest MSE values are attained for a batch size around 3 to 6 samples per batch. For batch sizes 8 and larger, the MSE tend to continuously increase following a weak slope. On the other hand, reducing the batch size to 2 samples per batch severely increases the MSE. Furthermore, for a batch size of 1 sample per batch, the MSE is not returned to due computational overflow (not shown). By combining

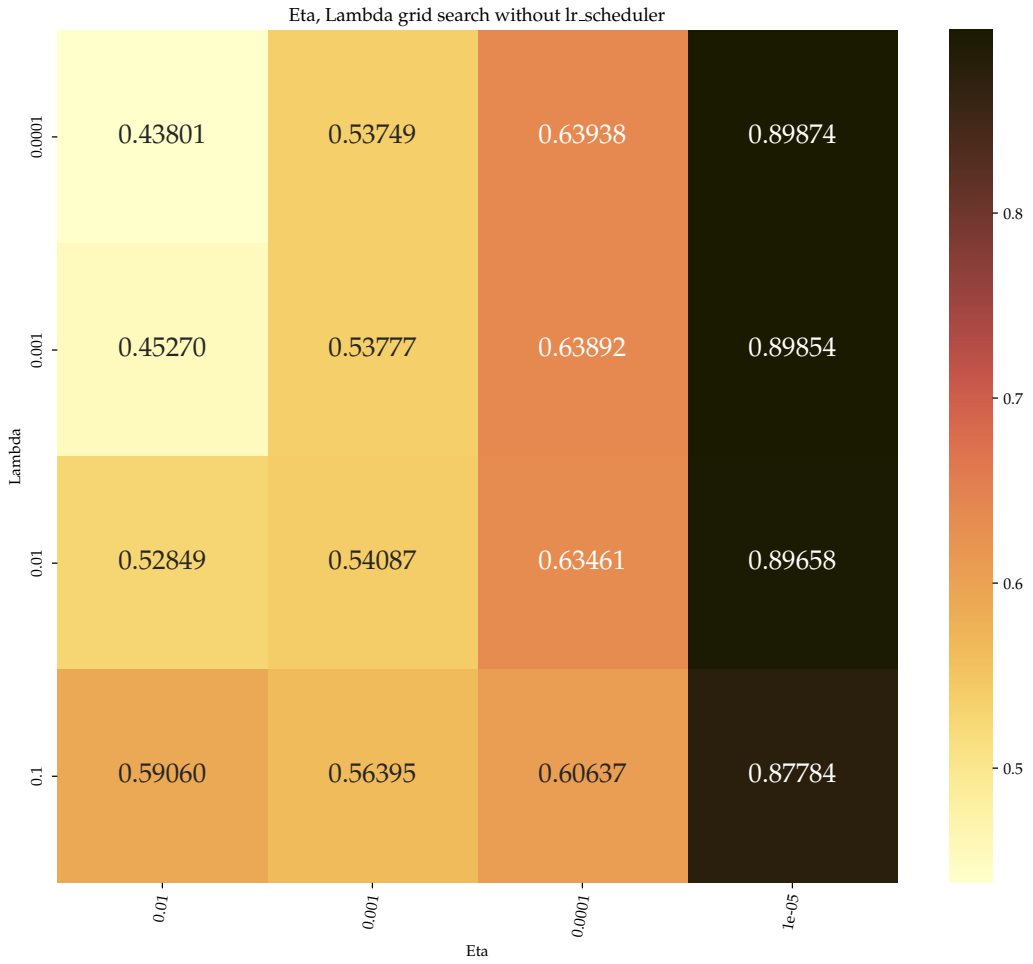


FIG. 3: MSE calculated for varying η and λ values in a grid search with learning rate scheduler turned off

TABLE II: MSE values from different SGD configurations based on the optimal hyperparameters as found in the grid search

SGD with lr scheduler	Momentum SGD with lr scheduler	SGD w.o. lr scheduler	SciKit-learn	Ridge
0.4783	0.4377	0.4203	1.0764	0.3256

the results of Figure (**??**) and (**??**), we can conclude that Stochastic Gradient Descent on the Terrain data performs optimal if given a descent number of epochs and a batch size of approximately 4. However, as previously discussed concerning computational resources, we have considered 100 epochs with a batch size of 4 the optimal when generating the coming results.

The two heatmaps shown in Figures (**??**) and (**??**) show the effect of the learning rate η as well as the l2 regularization term λ without and with a learning rate scheduler respectively. By comparing the two plots, it can be seen that the learning rate scheduler causes all but one model to attain higher MSE. Which might be a result of too aggressive scheduling.

Moreover, a secondary observation of the inclusion of the learning rate can be pointed out. By comparing the two rightmost columns of Figure (**??**) and (**??**), it can be seen that SGD differ greatly in MSE value for $\eta = 0.00001$ regardless of regularization parameter. This might be an effect of scheduling the learning rate, in combination with the learning rate being initialized to such a low value that the model is unable converge in

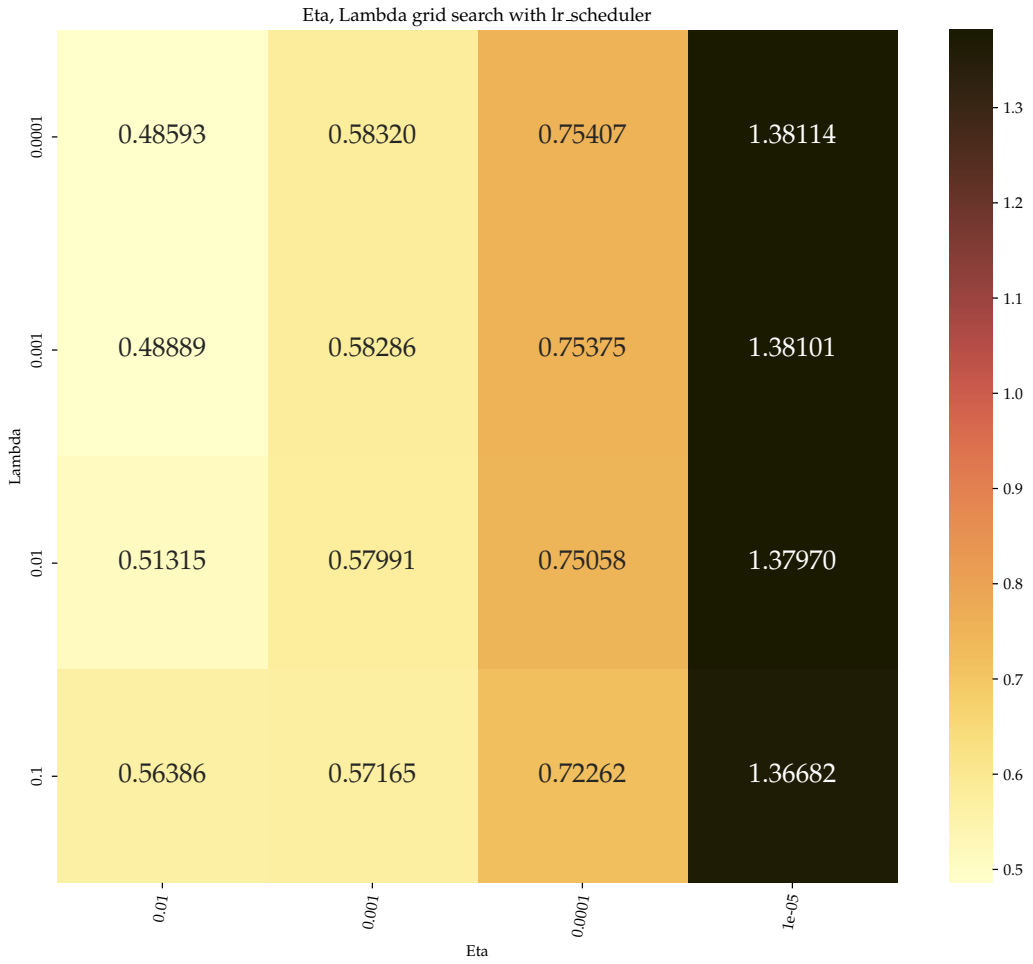


FIG. 4: MSE calculated for varying η and λ values in a grid search with learning rate scheduler turned on

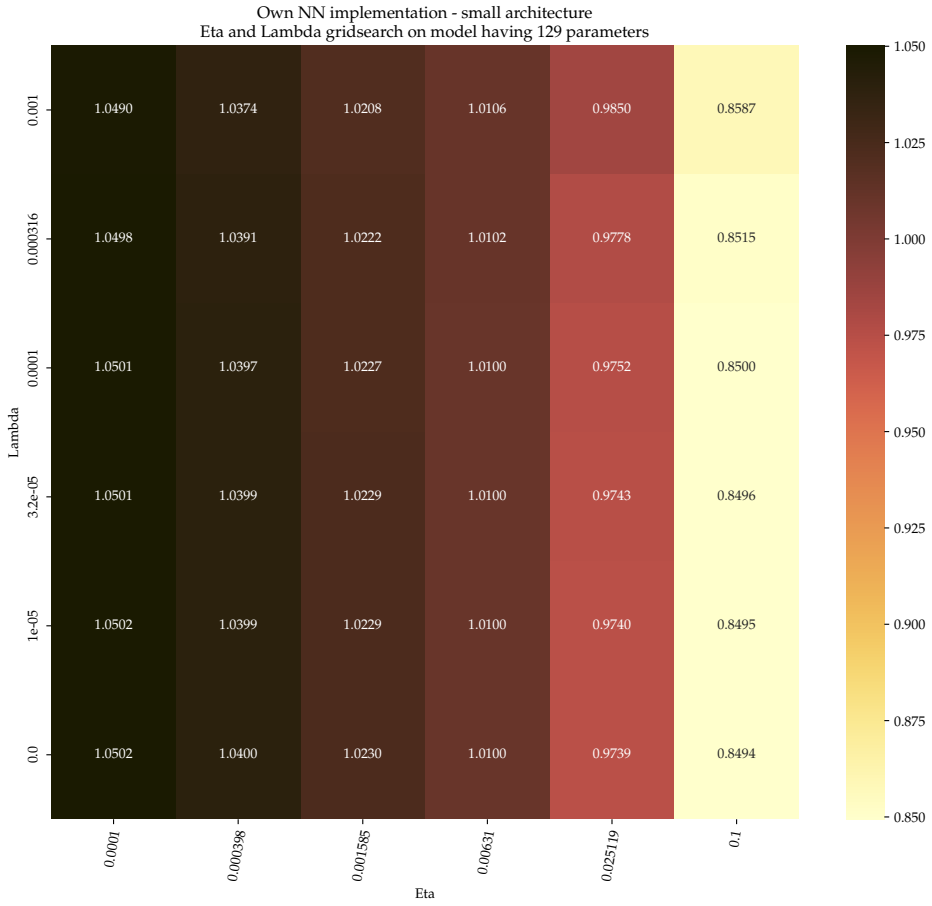
TABLE III: MSE values from different SGD configurations based on the optimal hyperparameters as found in the grid search, without regularization

SGD with lr scheduler	Momentum SGD with lr scheduler	SGD w.o. lr scheduler	SciKit-learn	OLS
0.4778	0.4354	0.4171	1.0869	0.3216

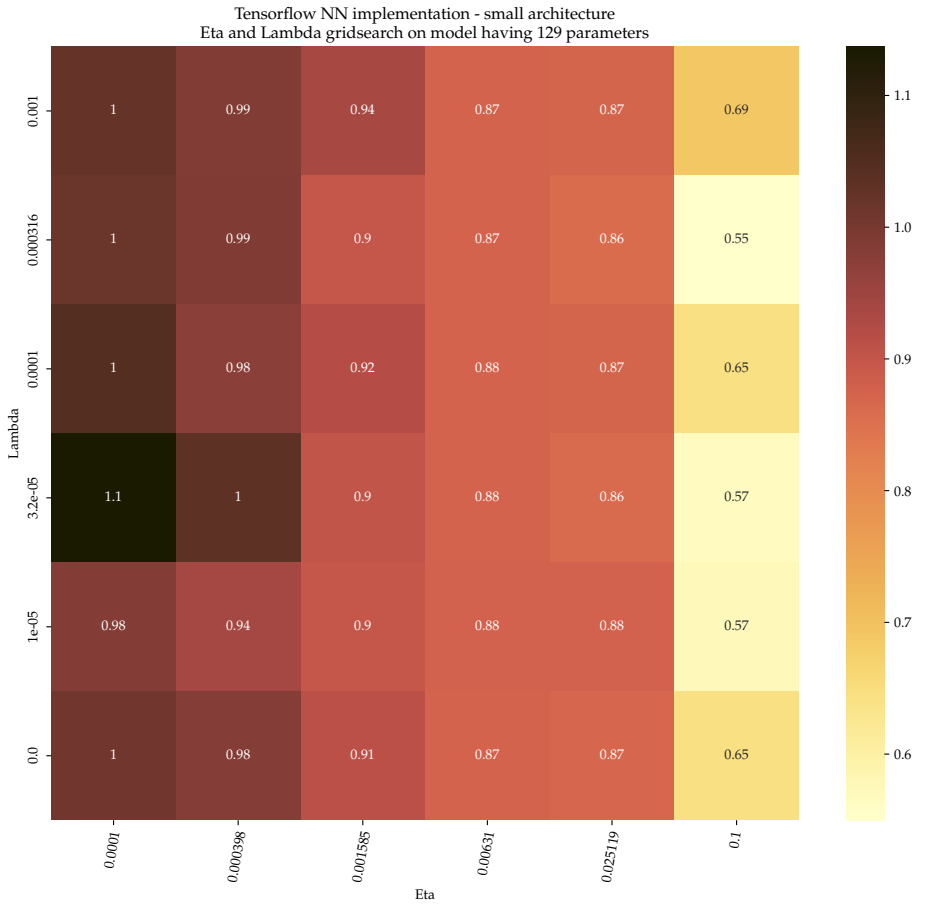
time of the scheduling reducing all movement along the gradient. Furthermore, the difference in MSE between the model for higher learning rates is not as great as for $\eta = 0.00001$, which then might be a result of the model reaching close to a minima before scheduling.

Figure (**??**) shows that the optimal hyperparameters for our SGD algorithm given the current dataset, is $\eta = 0.01$ and $\lambda = 0.0001$. As such, Figure (**??**) suggests that less regularization leads to lower attained MSE. This complies with the results of Table (**??**), were an even lower MSE value is attained when noe regularization is present. Furthermore, by comparing the Ridge regression value in Table (**??**) with the OLS value in Table (**??**) it can be seen that OLS scores better than Ridge regression for the optimal regularization parameter as found in Figure (**??**). These results are in line with what we discovered in Project 1 when analyzing our Ridge Regression algorithm and OLS algorithm on the same Terrain data, namely that OLS scores better in regards to MSE than Ridge.

By comparing the values in Table (**??**), it can be seen that the MSE values computed with our SGD algorithm gets close to the Ridge regression value, with an approx-

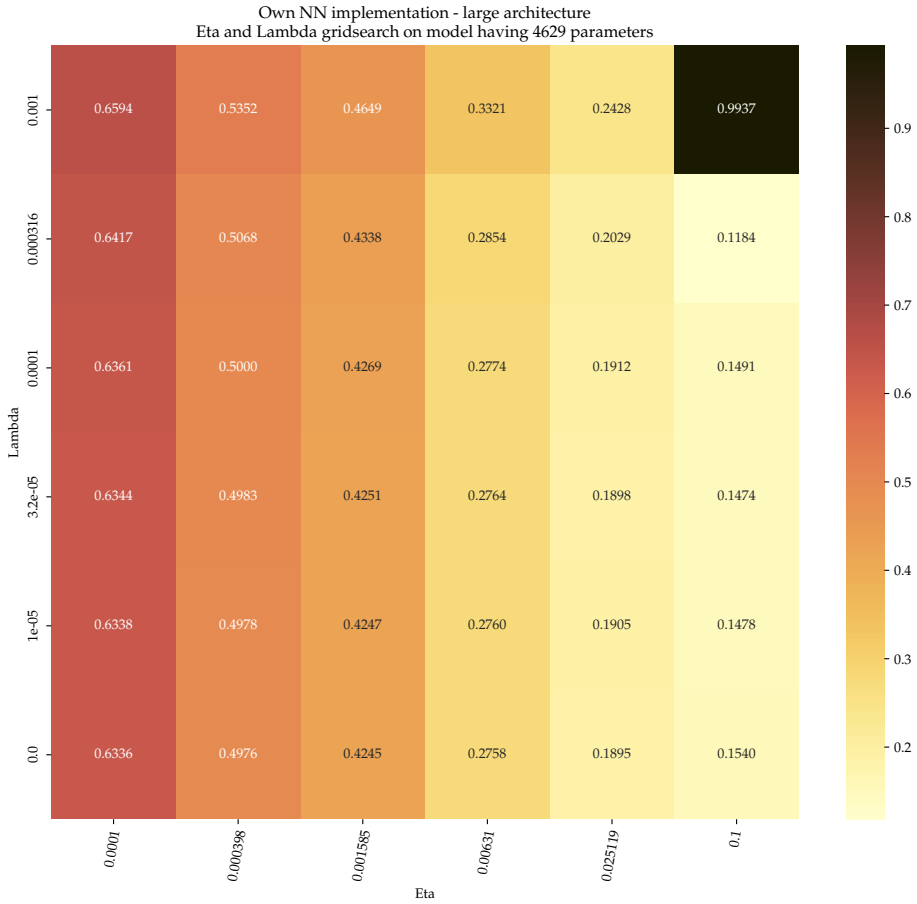


(a) Own NN model - Showing number of parameters, η and λ

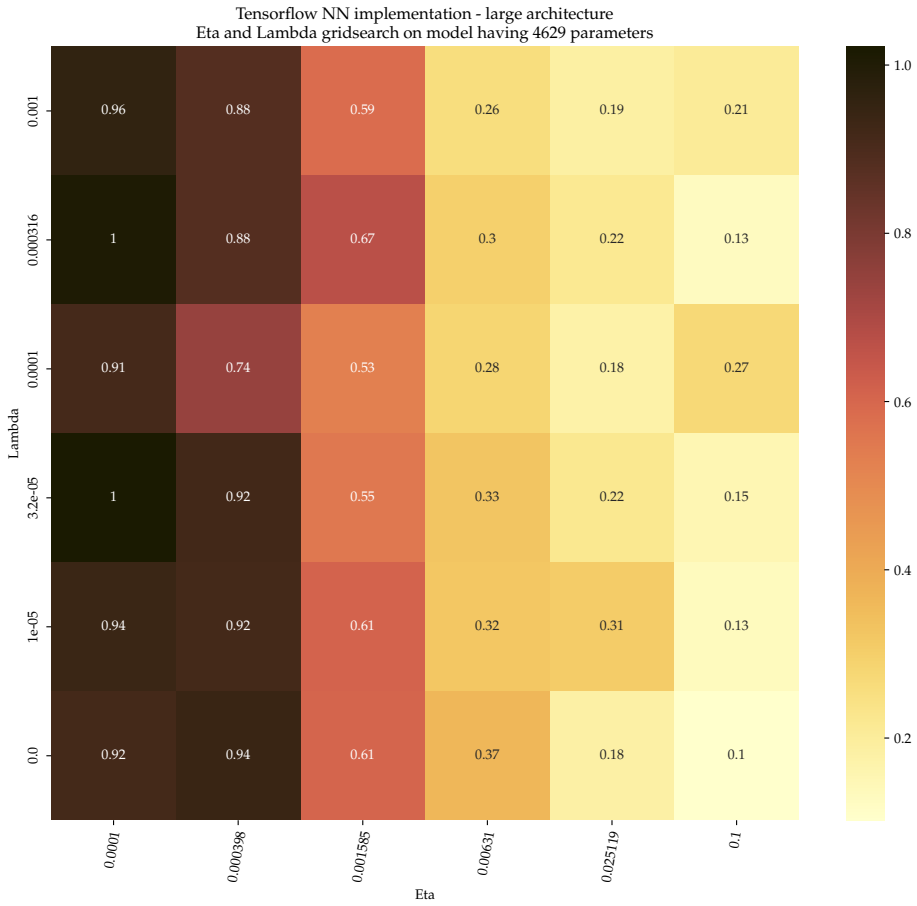


(b) Tensorflow model - Showing number of parameters, η and λ

FIG. 5: Grid search visualized for the best parameters using sigmoid activation function

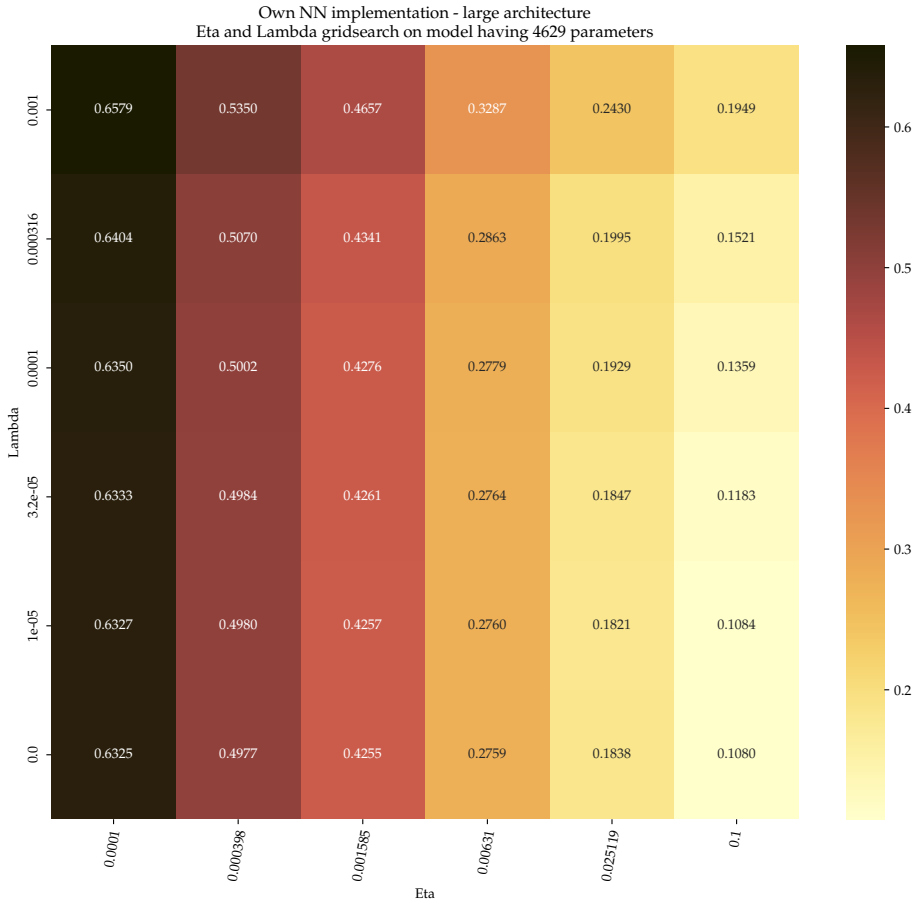


(a) Own NN model - Showing number of parameters, η and λ

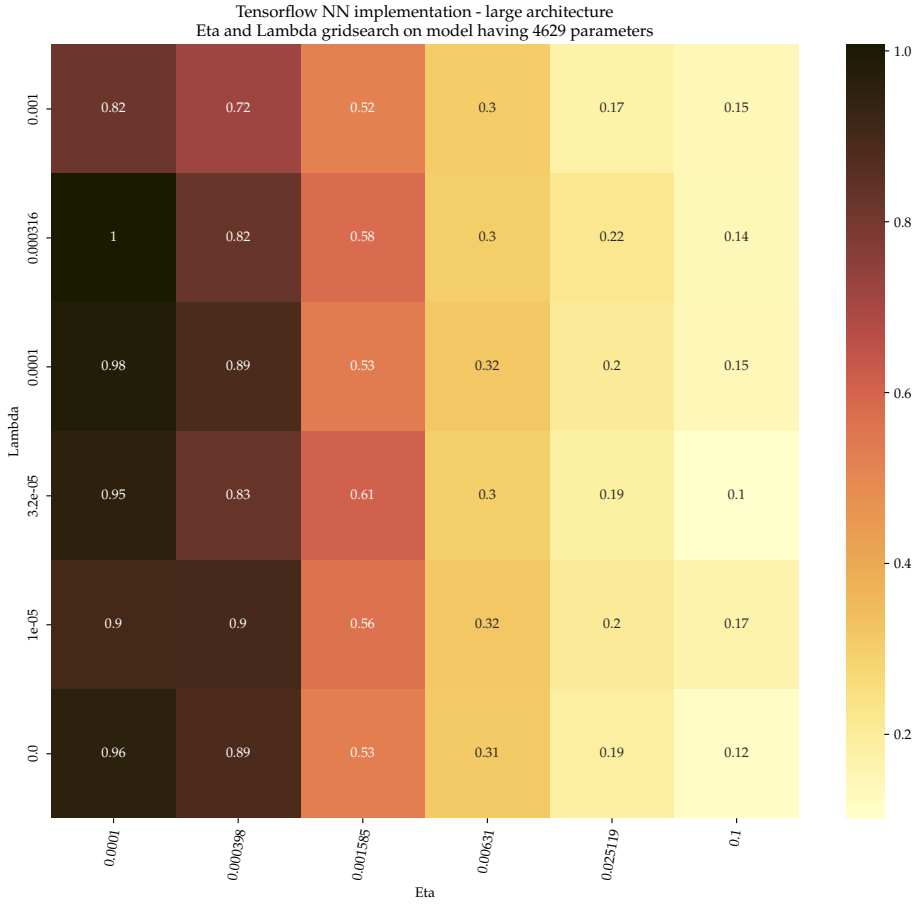


(b) Tensorflow model - Showing number of parameters, η and λ

FIG. 6: Grid search visualized for the best parameters using RELU activation function

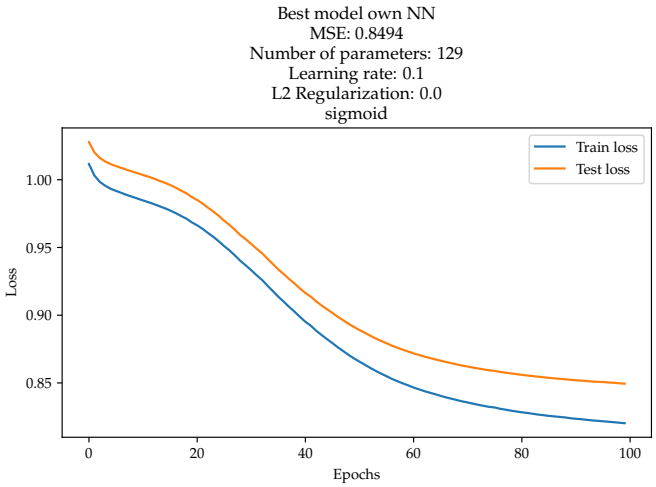


(a) Own NN model - Showing number of parameters, η and λ

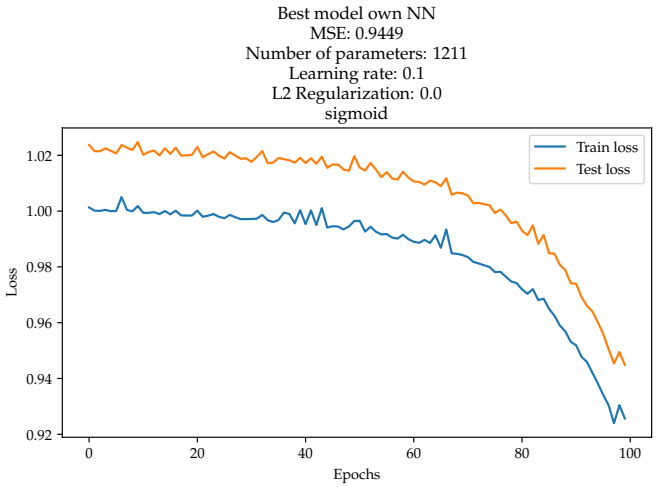


(b) Tensorflow model - Showing number of parameters, η and λ

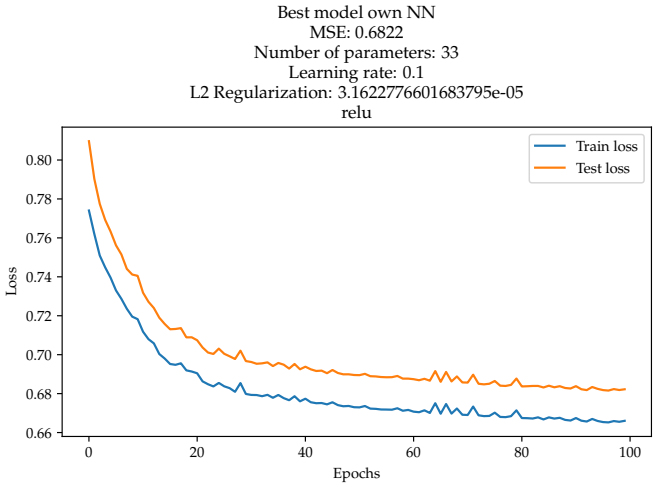
FIG. 7: Grid search visualized for the best parameters using Leaky RELU activation function



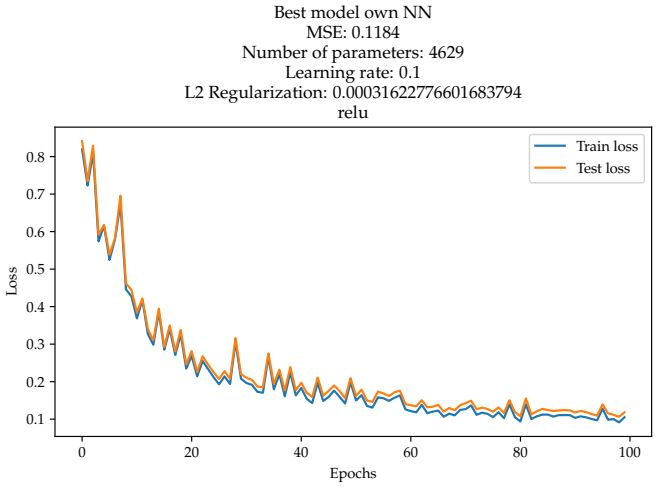
(a) Best fitted NN model using sigmoid and small arhitecture



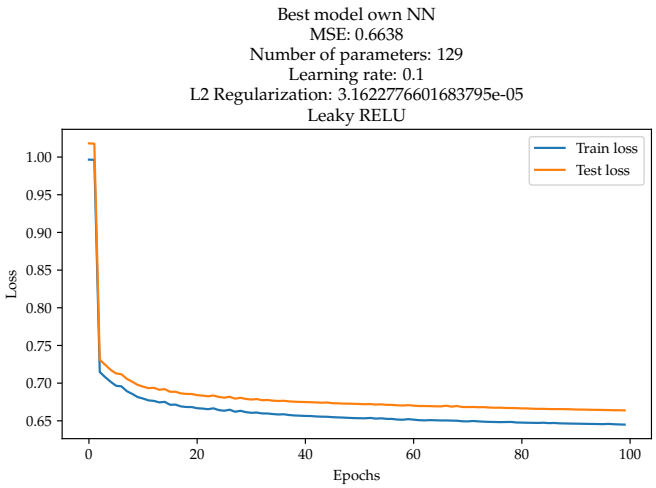
(b) Best fitted NN model using sigmoid and large arhitecture



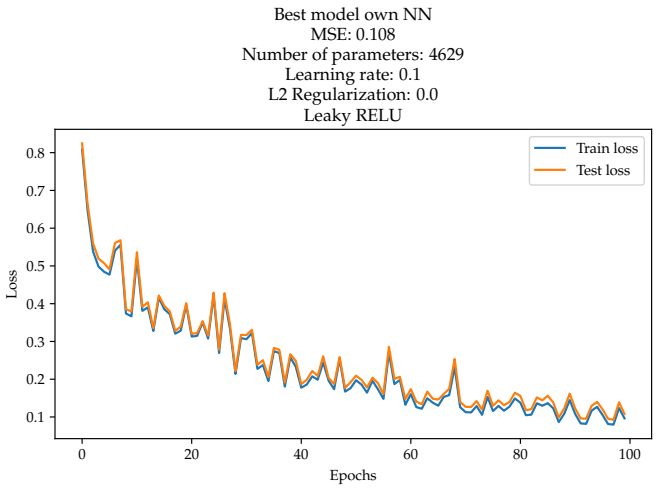
(c) Best fitted NN model using RELU and small arhitecture



(d) Best fitted NN model using RELU and large arhitecture



(e) Best fitted NN model using Leaky RELU and small arhitecture



(f) Best fitted NN model using Leaky RELU large arhitecture

FIG. 8: Small vs large architecture

imate difference of 0.1 for SGD without learning rate scheduler. Moreover, our SGD outperforms that of SciKit-learn, which is more than doubled of our computed MSE values. The same trends can be seen for the OLS case in Table (**??**). Our SGD algorithm outperforms that of SciKit-learn, though lower MSE values are achieved as discussed in the previous paragraph.

Another note when discussing Table (**??**) and (**??**) is the inclusion of Momentum SGD as described in Algorithm (**??**). With the added momentum parameter, lower MSE values are attained as all the hyperparameters are kept the same. Though the values are not as good as normal SGD without the learning rate scheduler.

Comparing Stochastic Gradient Descent and Newton Raphson for Logistic Regression

As demonstrated by Algorithm (**??**) and (**??**), there are several distinct differences between the two implementations. While Newton Raphson´s method only iterates through the epochs, the stochastic gradient descent will have additional iterations through the batches. This might lead to the impression that the latter is computationally more complex than the first. This is however not the case, as the inversion of a $k \times k$ matrix has a complexity of $O(k^3)$, which has to be carried out for each iteration as the parameters change with every update [**?**]. For cases with a large number of parameters, the exponentially increasing computational burden imposed often makes SGD the preferred solver.

Arificial Neural Net - Regression problem

To quantify the power of our neural network model, we studied how well the network performed relative to results obtained using Ordinary Least Squares (OLS) in project 1. Our best fit for the OLS model was MSE equal to 0.169, which served as our initial reference score for the NN model. For the models to compete on similar terms, we reused the same image patch from project 1 and fitted our NN on the same data using the same scaling process. We scale both the x and y input coordinates and the targets by subtracting their mean values. For the OLS fit, we feed the model using degree 10 for both the x and y coordinates for the OLS fit, which seeded the model with XX features to work on. One significant difference for our Neural Net is that we only feed the model with the raw data, meaning we input the first-order inputs of x and y. By doing this, we left it to the NN model to find the optimal polynomial and its degree to be incorporated within the different layers and neurons of its Neural Net. For a well-implemented Neural Net, it is often beneficial to leave it to the model to find the optimal relationships rather than seeding it to look for a specific degree. To be fair, the true optimal degree is unknown even though we approximated the best fit to be of degree 10 for the OLS model specifically. Seeding the NN to match a specific degree can go both ways; it can help the model converge faster with a better fit, but it can also restrain the model with too little or too much unnecessary input data. Feeding the model with too many features can also result in the model reaching the curse of dimensionality issue at some point, leaving the model with a worse fit, since having too many features may result in an increased level of noise. A common way to tackle too many features is to use the Principal Component Analysis (PCA) algorithm. This will extract the most essential features. However, increasing the number of degrees for our design matrix (order of magnitude of our raw input) and applying PCA on it is not a good choice since such an approach yields dependent input data when the higher level feature are constructed from the order of magnitude of the raw data; thus, the condition for using PCA is not met. This left us with the most optimal approach for input feature for the Neural Net to just being the coordinates x and y from the input data. One of the major challenges of training a network full of neurons is the choice of hyper parameter. A common way to find good hyperparameters is to traverse parts of the parameter space by using a grid search.

Optimal parameters

For finding optimal hyperparameters we utilized a grid search for different hyperparameters as shown in table xx. We decided to keep batch size and number of epochs constant to limit our options for computational reasons. We ran grid search with the chosen hyperparameters for all three activation functions; Sigmoid, RELU, and Leaky RELU, using both the small and the large network architecture shown in figure xx. The most optimal MSE value for all the three activation types relative is shown in figure x1,x2,x3. Here we see that.... Relative to number of parameters.

Parameters and model complexity

In terms of model complexity and the number of parameters we see that our model behaves best for ... xxxx Plots relative to model complexity for all activation functions.

Performance of activation functions

Comparing the different activation functions, the Sigmoid function has the worst performing given the parameters tested. RELU is performing significantly better than Sigmoid, and the best performing activation function is the Leaky RELU.

Effect of weight initialization

We first tried to initialize the weights using gaussian distribution with N(0,1). This resulted in significantly worse performance than our final initialization strategy. Our initial weight initialization strategy also resulted in the model being sensitive to exploding gradients from the training process for RELU and Leaky RELU. We struggled with predicted values exceeding float64 relatively often.

Neural Net vs Ordinary Least Squares on terrain data

Comparing our best fitted NN with the best OLS fit in project 1 is shown in figure1. It is clearly that our NN outperforms the OLS model. The

Robustness of our Neural Net

CONCLUSIONS

Source Code

Link to github repository containing all developed code for this project: https://github.com/AndreasBordvik/FYS-STK4155-Prj2_report