Comparing Neural Networks with Regression methods for Classification and Regression problems

Gard Pavels Høivang* and Are Frode Kvanum†
Department of Geoscience, University of Oslo

David Andreas Bordvik‡
Department of Informatics, University of Oslo
(Dated: November 18, 2021)

In this project, we studied both the terrain data used in Project 1 as well as the Wisconsin Breast Cancer Data with our own implementation of a Feed Forward Neural Network. We have implemented Stochastic Gradient Descent as a framework for the other algorithms. For the terrain data, we compare our own Neural Network with the OLS and Ridge regression algorithms which we developed in Project 1. When studying the categorical Wisconsin Breast Cancer Data, we have also developed code for Logistic Regression. Our developed algorithms have also been benchmarked with the popular machine learning libraries PyTorch and Tensorflow. By tuning the hyperparameters, we have shown that our own Neural Network outperforms our models from Project 1, when fitting the terrain data with an MSE = 0.0971 compared to our best result of MSE = 0.1602 from OLS. We have also shown that our Neural Net is able to achieve comparable MSE values with that of Tensorflow given the Leaky ReLU as activation function. With regards to classification, our Neural Network performs comparatively to both Logistic Regression as well as PyTorch. We conclude with our Neural Network being the method of choice when fitting the terrain data, **AND SOME MORE WITH CLASSIFICATION!**

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 [5].

The **Universal approximation theorem** states that a Neural Network can approximate any function with as few as a single hidden layer to any precision [3, 8]. Hence, any feedforward network with a single hidden layer is in theory ample to represent any function, due to their underlying universality [9]. 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 [11]. 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 [11], PyTorch [10] and TensorFlow [1].

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 [11]. 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.

* gardph@mail.uio.no
† afkvanum@mail.uio.no
‡ davidabo@mail.uio.no

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 (1), 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 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 [6].

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 (1) 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 [6]. 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 [6].

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 $\eta = \eta_0 / t^{t_0}$ [11]. 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 minibatches 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 [6]. 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 (2)

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

Stochastic Gradient Descent is implemented as pseudocode in Algorithm (1)

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

For clarity, Algorithm (1) can easily be extended with a regularization term, such as the l2 regularizer making it comparable to Ridge Regression. In the case of regularization, 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 (2) 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 (2).

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

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 [6]. Our implementation of a Feed Forward Neural Network is inspired by the popular Machine Learning libraries TensorFlow [1] and PyTorch [10], 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 [12].

Feed Forward pass

With out Feed Forward pass, information is sent in one direction through the layers of the model. More specifi-

cally, 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 (3) below

Data: Input matrix $\boldsymbol{x}$
Result: Estimated value of the true output $\boldsymbol{y}$
$\boldsymbol{a}^0 = x$;
for *layer* $l = 1, \ldots, L$ **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 [5]. 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

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

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 (3) 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 (4) 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 ($l-1 \to l \to l+1$) and that the Equation (5) is not an algebraically correct equation. As a quick note, the error is not propagated to the input. As such, to compute the final weight update between the input and first hidden layer, we perform a similar calculation as in Equation (5) but with the value in the input nodes instead of the activated node value.

The equations derived and explained in this section forms the basis for the backpropagation algorithm which we will implement for our Neural Network. The algorithm is stated in Algorithm (4). Note for correctness that though backpropagation is just the backpropagation of the error, not the weight and bias update, we include the latter in this algorithm for simplicity.

Data: Input matrix $\boldsymbol{x}$, targets $\boldsymbol{t}$
Result: Updated weights $\boldsymbol{W}$ which constitutes a fully trained network
Compute $\frac{\partial C}{\partial \boldsymbol{a}^L}$;
$\boldsymbol{\delta}^L \leftarrow \frac{\partial C}{\partial \boldsymbol{a}^L} \circ f'(\boldsymbol{z}^L)$;
// Backpropagation;
for *Layer* l *in* $L-1, L-2, \ldots, 2$ **do**
  $\boldsymbol{\delta}^l \leftarrow \boldsymbol{\delta}^{l+1} \boldsymbol{w}^{l+1} \circ f'(\boldsymbol{z}^l)$;
end
// Update weights and biases;
for *Layer* l *in* $L, L-1 \ldots, 3$ **do**
  $\boldsymbol{w}^l \leftarrow \boldsymbol{w}^l - \eta * \boldsymbol{a}^{l-1^T} \boldsymbol{\delta}^l$;
  $\boldsymbol{b}^l \leftarrow \boldsymbol{b}^l - \eta * \sum \boldsymbol{\delta}^l$;
end
$\boldsymbol{w}^2 \leftarrow \boldsymbol{w}^2 - \eta * \boldsymbol{x}^T \boldsymbol{\delta}^2$;
$\boldsymbol{b}^2 \leftarrow \boldsymbol{b}^2 - \eta * \sum \boldsymbol{\delta}^2$;
return *A trained model*

Algorithm 4: Backpropagation of the error computed at the output layer of a Neural Network, with subsequent weight and bias update computed using Stochastic Gradient Descent. Note that the superscript indicate layer position, and the usage of 1-indexing.

Defining our activation functions

When defining the mathematical framework for Algorithm (3) and (4), we stated the ambiguous activation function f. The activation function of a node takes as input the prior layer's output multiplied by the weights connecting both layers as well as adding a bias term, and returns as output an activated value which is then sent forward to the next layer of the model. As such, the activated value of a node is by large defined by what activation function is associated with its layer. Moreover, for an activation function to be compliant with the universal approximation theorem stated in the Introduction, the function have to obey to the following restrictions:

1. Non-constant
2. Bounded
3. Monotonically-increasing
4. Continuous

Moreover, as the output of our Neural Network will be a linear function of the inputs, the above restrictions allow for the Network to fit non-linear functions by themselves introducing non-linearity.

For this project, we will implemented three different activation functions, these are:

- Sigmoid
- ReLU
- Leaky ReLU

The **Sigmoid** function can be seen in Equation (6).

$$\sigma(x) = \frac{1}{1+\exp(-x)} \tag{6}$$

The function produces values in the range of $(0, 1)$, and is in line with the four restrictions posed above. Unfortunately, a major problem with the sigmoid as activation function is the vanishing gradient [4]. The vanishing gradient problem arises in Deep Neural Networks when the sigmoid function gets an extremely large value as input, which it then outputs close to saturation moreover leaving the derivative close to 0. Furthermore, the gradients already tend to weaken backwards trough the layers. Thus, combining the weakening gradient with with the derivative of the sigmoid potentially close to 0 could hinder the backpropagation algorithm to reach the first layers leaving them unchanged.

As a consequence of the vanishing gradient problem, other activation functions have been proposed. In this project, we will inspect the ReLU function family, as it does not saturate for large positive values as the case was for sigmoid.

The **ReLU** function is defined mathematically in Equation (7)

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{7}$$

Though Equation (7) does not simulate nature to the same extent that the sigmoid does, it is expected that our Neural Network code will perform better with ReLU activation than sigmoid activation. However, a problem with the ReLU function is how it has the ability to output 0, as also seen in Equation (7). Moreover, what happens if a node is activated to 0 is that the node effectively is rendered useless. Moreover the node is prone to keep outputting 0 if the weighted sum of the previous layers stay negative. This is known as the dying ReLU, as the nodes essentially dies out.

A fix to the dying ReLU problem is to implement the **Leaky ReLU** as activation, which slightly changes the ReLU function to the form described in Equation (8)

$$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{8}$$

Weight initialization

EMPTY SECTION

Model architecture

For this project, we have developed two different model architectures which we will study in conjunction with the analysis of the datasets. As these different architectures pose different complexities in regards to model parameters and design, our goal is to study model dependency concerning model complexity. Our two proposed model architectures are presented visually in Figure (1) and (2). For a more technical implementation with regards to Source Code, we refer to the GitHub repository in the appendix where both models can be found under *code/models.py*.

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 [11]. 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.

Classification using our own Neural Network

In this section, we will highlight the important differences that separates a our Neural Network for regression and classification problems. First, a separate cost function than the MSE has to be supplied and solved after the feed forward pass (3). In the case of classification, the cost function to minimize is the Cross Entropy function for a binary case. The equation will be derived in the coming section, but can be seen in Equation (9). Secondly, the output-layer is activated using the logistic sigmoid function, to ensure that the predicted values stay in the range of $(0, 1)$ during training. This is in comparison to the output layer for a regression Neural Network, which is not required to contain an activation function, though ReLU could be used if the target is purely positive. Finally, when predicting new values using the Neural Network, they have to be tresholded with regards to some threshold to ensure that the predicted output is categorized. In this project, we are only dealing with a binary classification problem with the Wisconsin Breast Cancer Dataset [11], thus when predicting new values using our Neural Network they have to be categorized into 0 or 1 (True / False).

Though

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

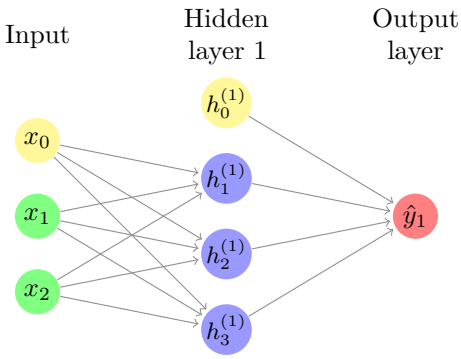


FIG. 1. Example of a small model architecture having two input features, three neurons in the first hidden layer, and one output

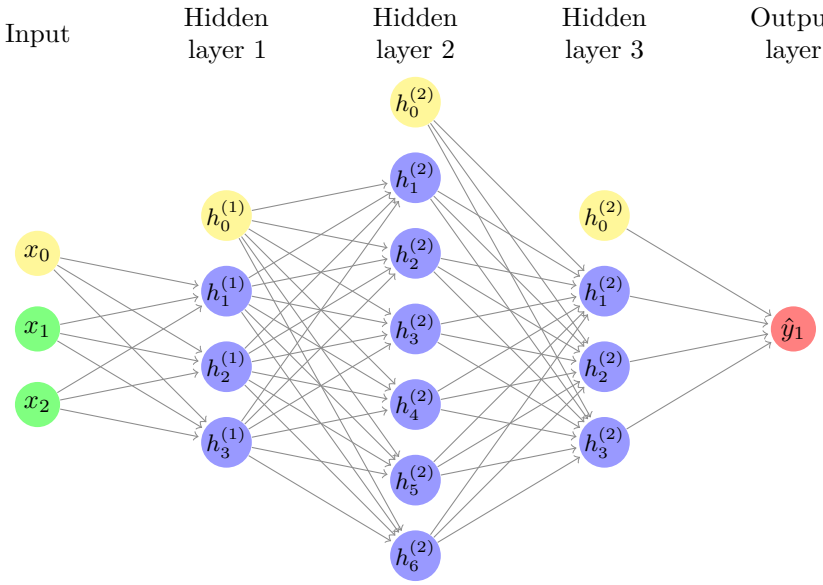


FIG. 2. Example of a large model architecture having two input features, three neurons in the first hidden layer, five neurons in the second hidden layer, three neurons in the third hidden layer and one output

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

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

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i) - \log\left(1 + \exp(\beta_0 + \beta_1 x_i + \ldots + \beta_{29} x_i)\right)\right) \tag{9}$$

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

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

$$\vdots$$

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

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

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

We have chosen to implement Logistic Regression with both stochastic gradient descent and Newton Raphson´s method. Algorithm (5) 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 5: Logistic Regression with Stochastic Gradient Descent and l2 regularization

Newton Rapshon's method

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);
  $W \leftarrow p(y_i|x_i,\boldsymbol{\theta})(1-p(y_i|x_i,\boldsymbol{\theta})$;
  $hessian \leftarrow \boldsymbol{X^T W X}$;
  $\theta \leftarrow \theta - \left(X^T W X\right)^{-1} \times \frac{\partial \mathcal{C}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$;
end
return θ

Algorithm 6: 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

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

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

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{10}$$

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. The color maps have been developed by Fabio Crameri, which fairly represents the data as well as being readable for those with color-vision deficiency [2].

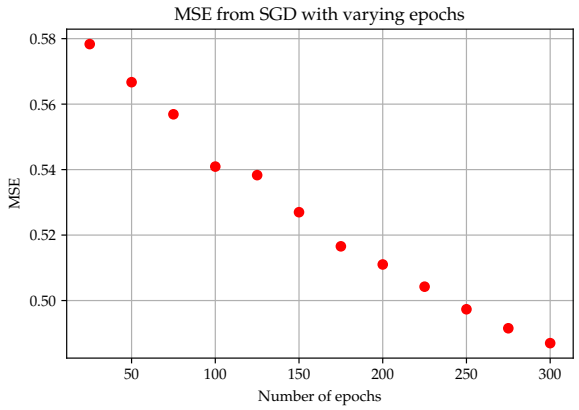


FIG. 3. MSE calculated from Stochastic Gradient Descent optimization of the predictors θ as function of the number of epochs with all other hyperparameters kept fixed

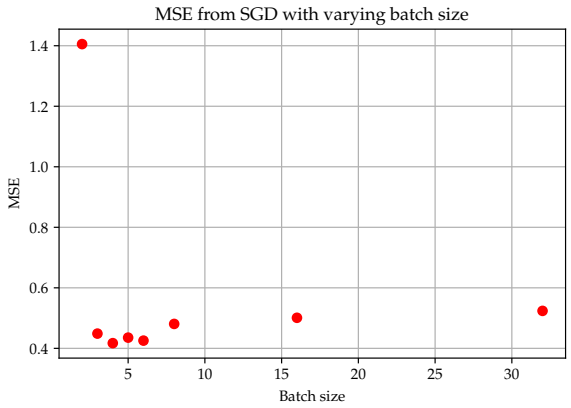


FIG. 4. MSE calculated from Stochastic Gradient Descent as function of the batch size with all other hyperparameters kept fixed

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

Stochastic Gradient Descent

Our Stochastic Gradient Descent is implemented akin to Algorithm (1). 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 (3) plots the MSE computed from our SGD implementation against the number of epochs used.

The following Figure (5) plots the MSE as a function of batch_size using the SGD algorithm in 1.

Both Figures (5, 6) are created using a grid search algorithm over different values of η and λ, presented as a heatmap.

Table (II) 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 (2).

Arificial Neural Net

Optimal parameters

Parameters and model complexity

Performance of activation functions

Own Neural Net vs Tensorflow

Neural Net vs Ordinary Least Squares on terrain data

Robustness of our Neural Net

Insert CV plot using Leaky RELU and Large architecture

Logistic Regression

Insert results of classification

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 (3) 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

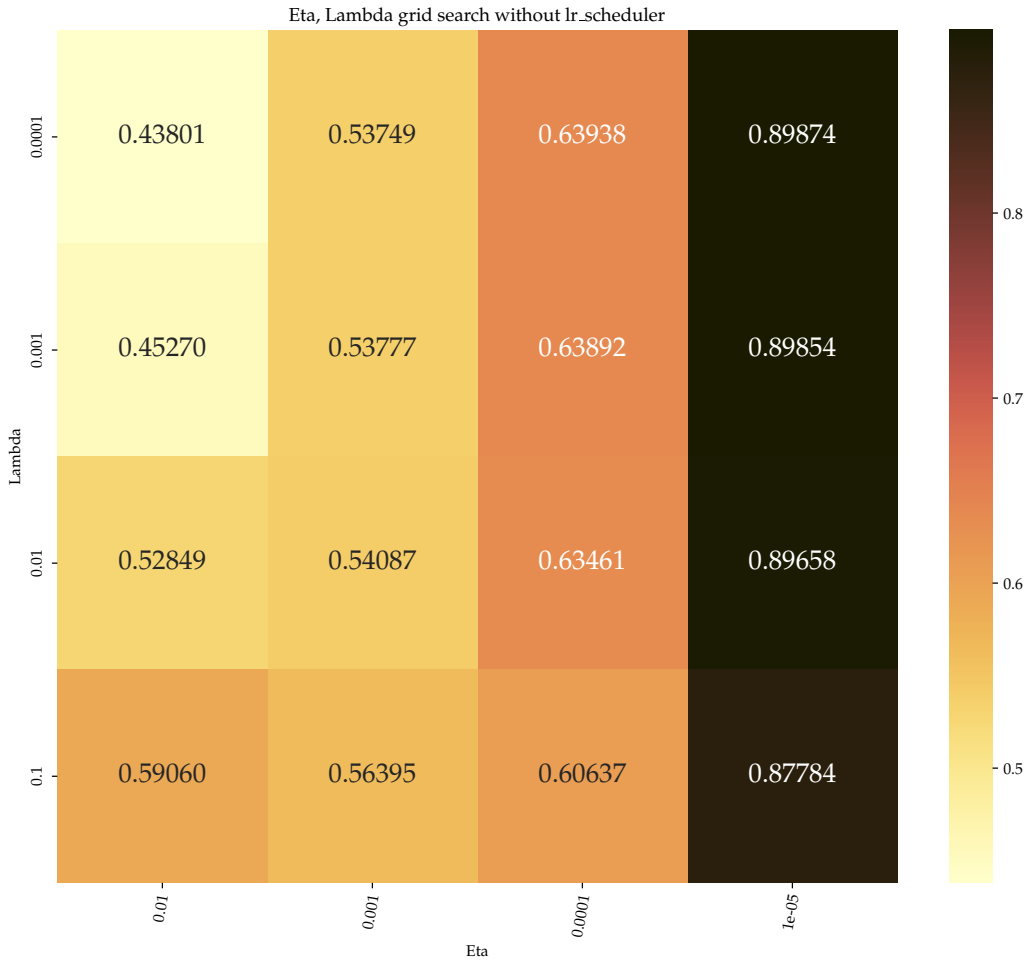


FIG. 5. MSE calculated for varying η and λ values in a grid search with learning rate scheduler turned off

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

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 (I). 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 (4), 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 the results of Figure (3) and (4), we can conclude that Stochastic Gradient Descent on the Terrain data

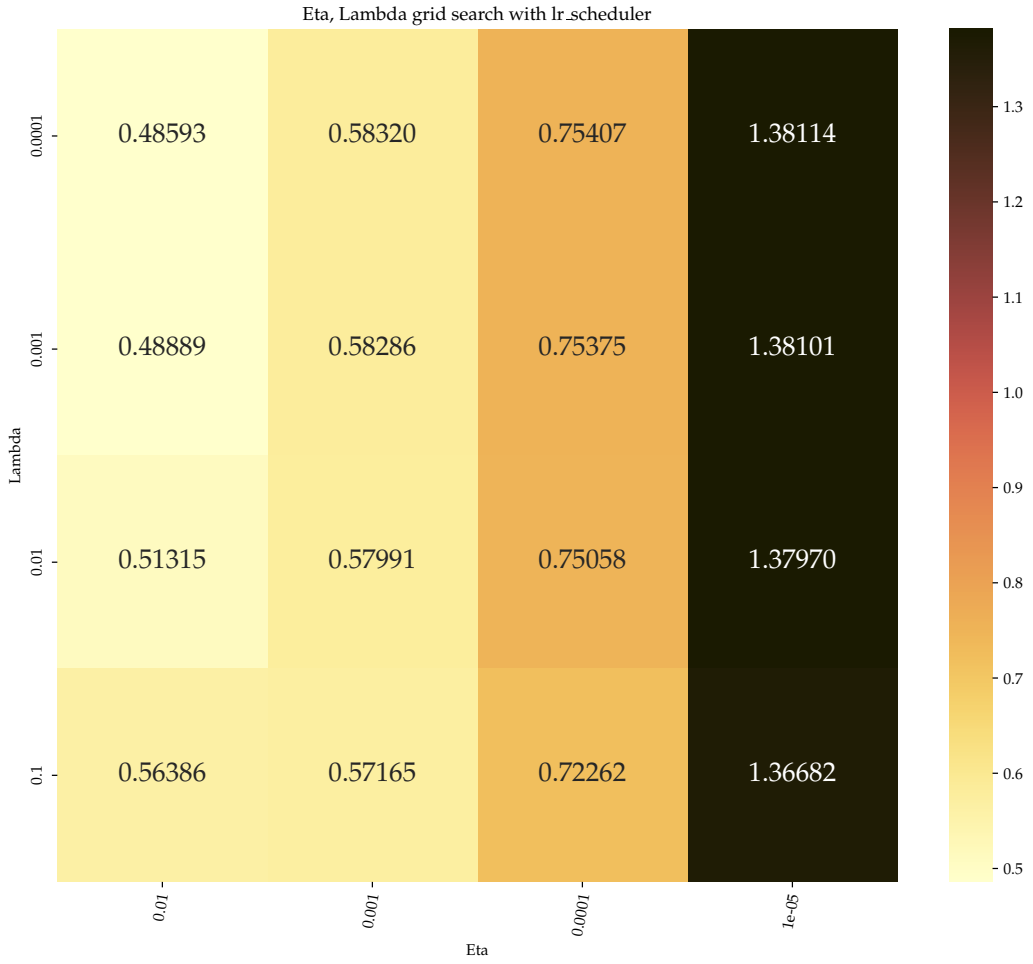


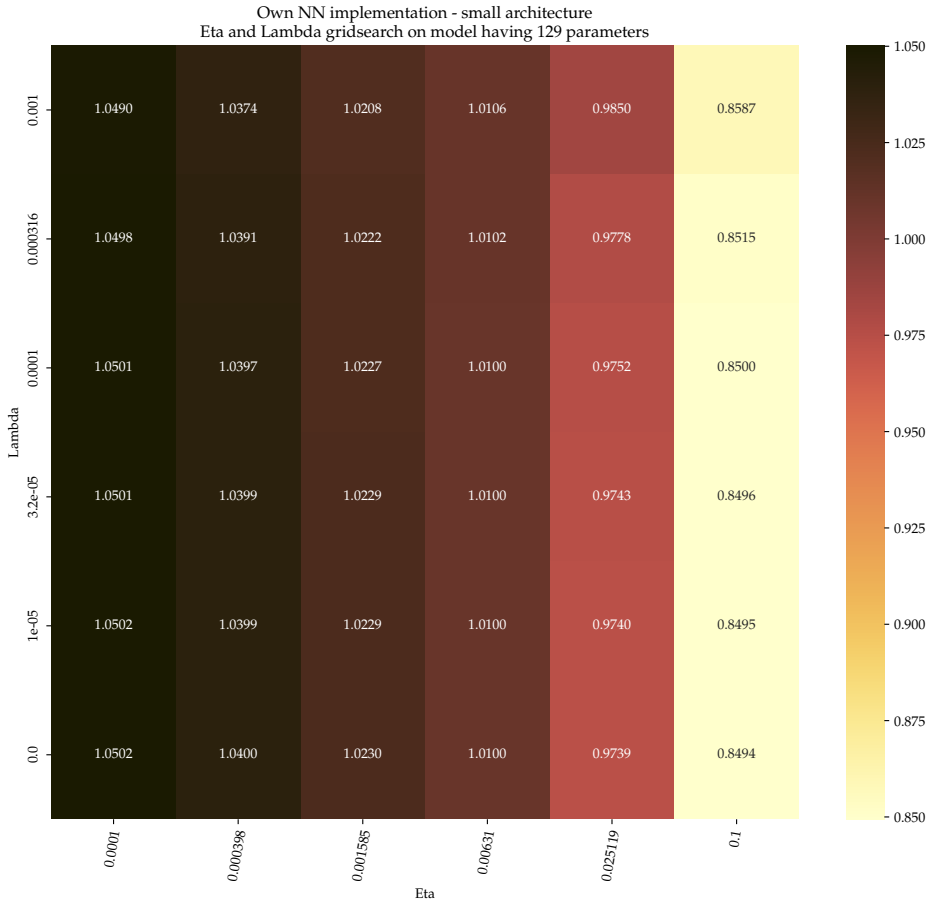
FIG. 6. MSE calculated for varying η and λ values in a grid search with learning rate scheduler turned on

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

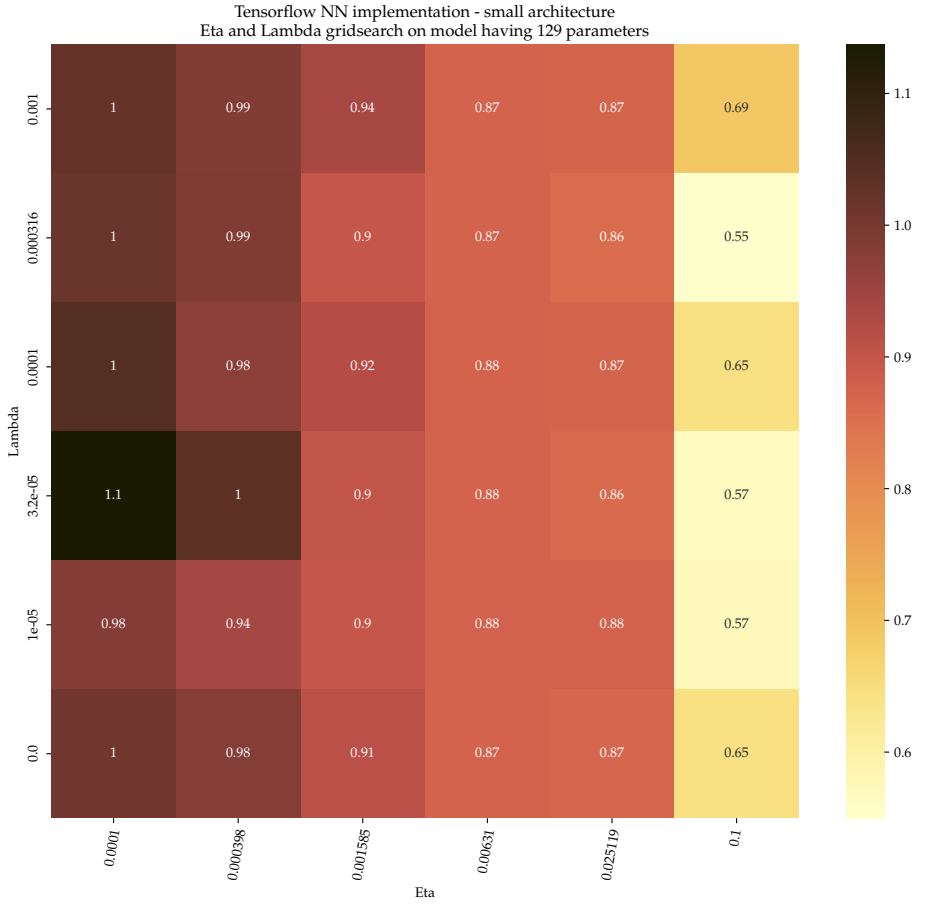
SGD with lr scheduler	Momentum SGD with lr scheduler	SGD w.o. lr scheduler	SciKit-learn	OLS
0.4778	0.4354	0.4171	1.0869	0.3216

TABLE IV. Optimal grid search parameters for the sigmoid activation function

Sigmoid - Best grid search	Results
MSE	0.8494
R2	0.168
parameters	129
neurons at first hidden layer	32
learning rate	0.1
lambda	0
epochs	100
batch size	300

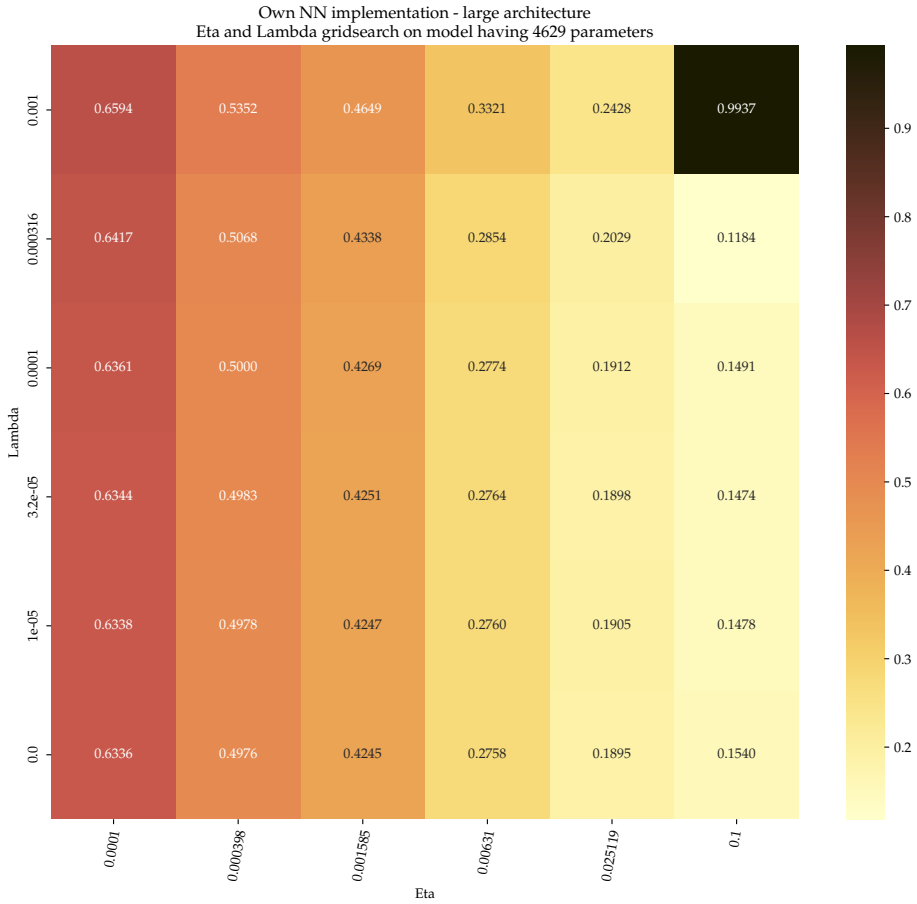


(a) Own NN model - Showing number of parameters, η and λ

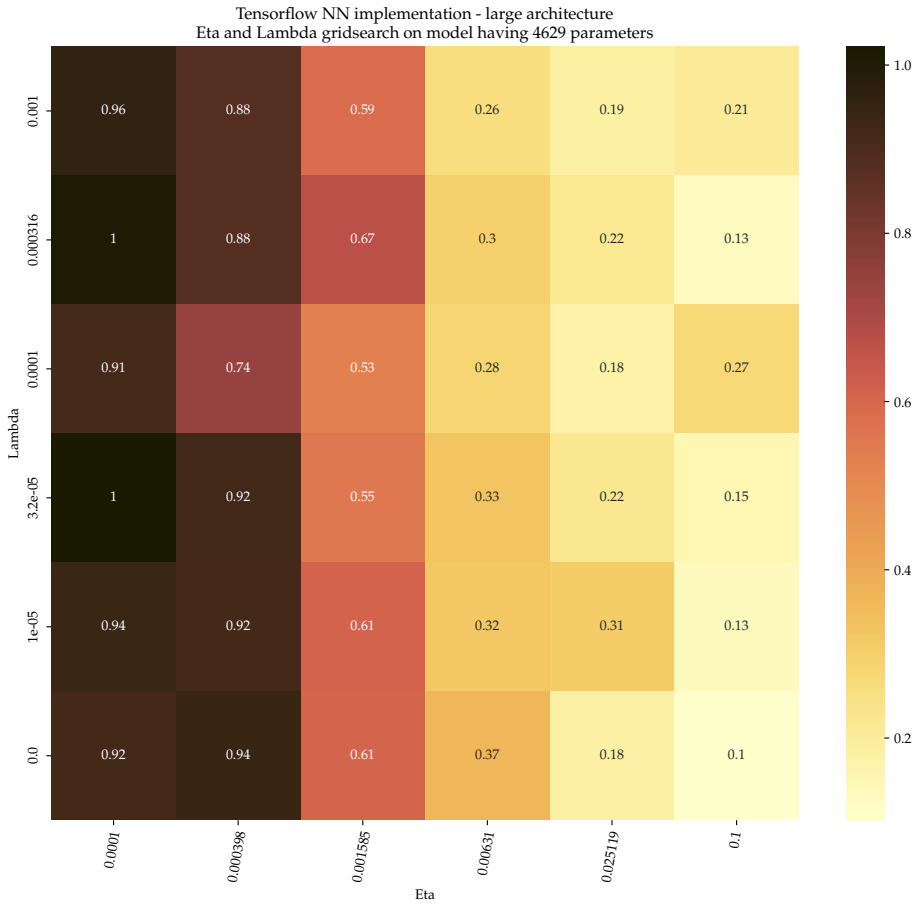


(b) Tensorflow model - Showing number of parameters, η and λ

FIG. 7. Grid search visualized for the best parameters using sigmoid activation function

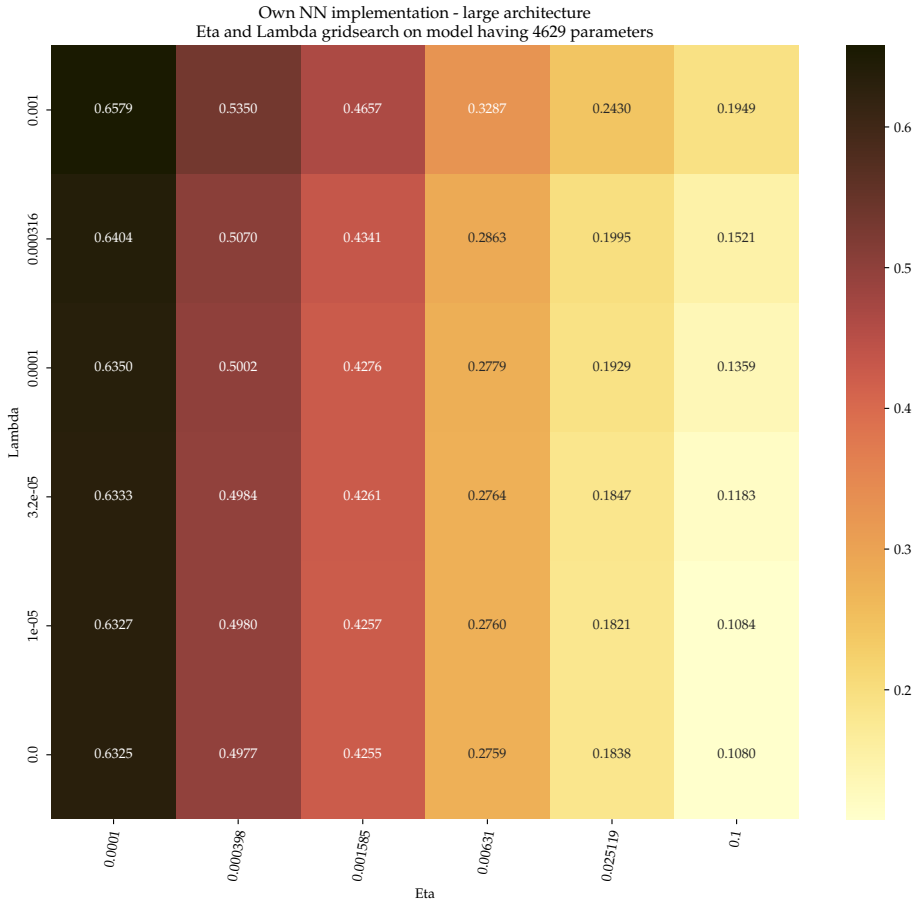


(a) Own NN model - Showing number of parameters, η and λ

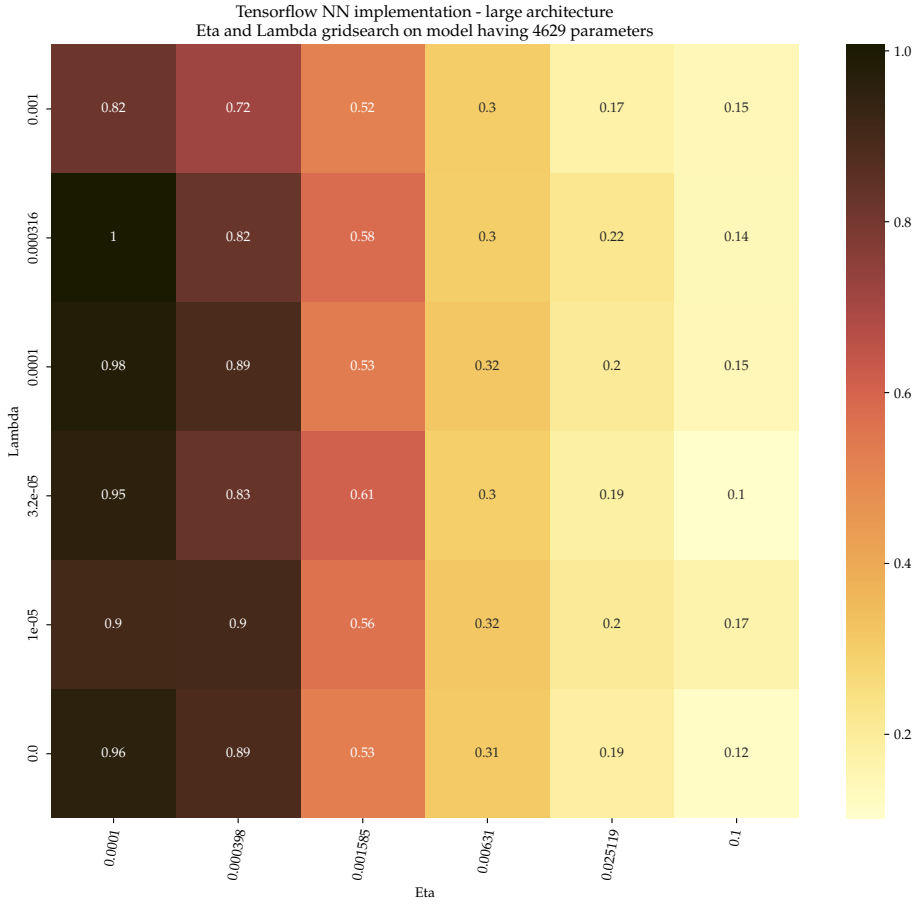


(b) Tensorflow model - Showing number of parameters, η and λ

FIG. 8. 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. 9. Grid search visualized for the best parameters using Leaky RELU activation function

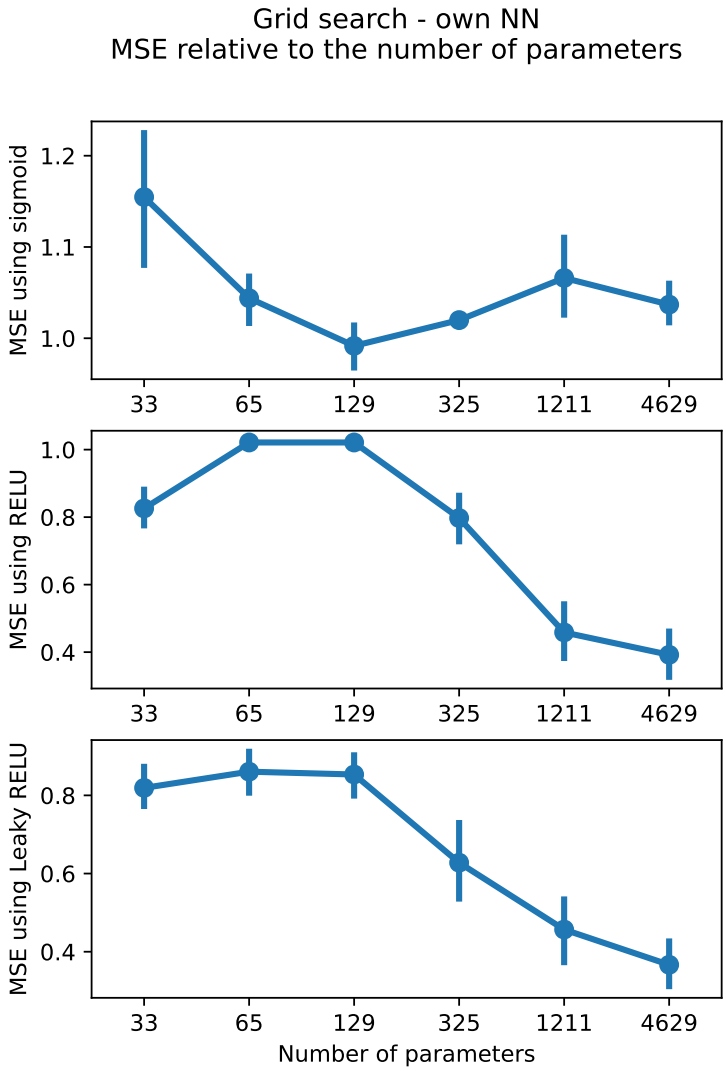


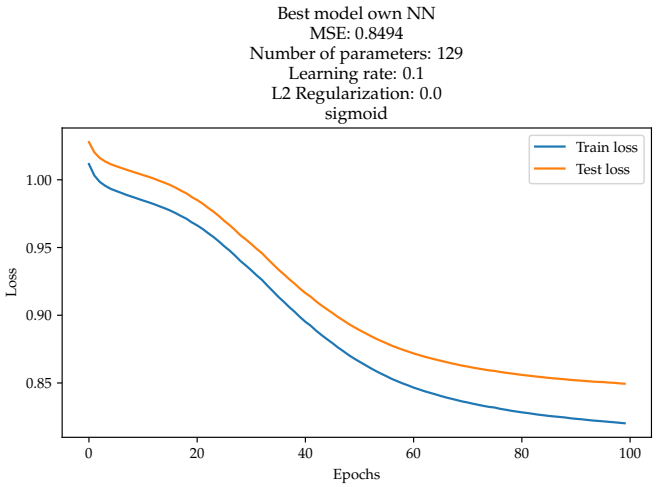
FIG. 10. MSE score vs number of paramteters in model architecture. Vertical lines visualize the minimum and maximum MSE value obtained using gridsearch over η and λ with respect to the current model architecture

TABLE V. Optimal grid search parameters for the RELU activation function

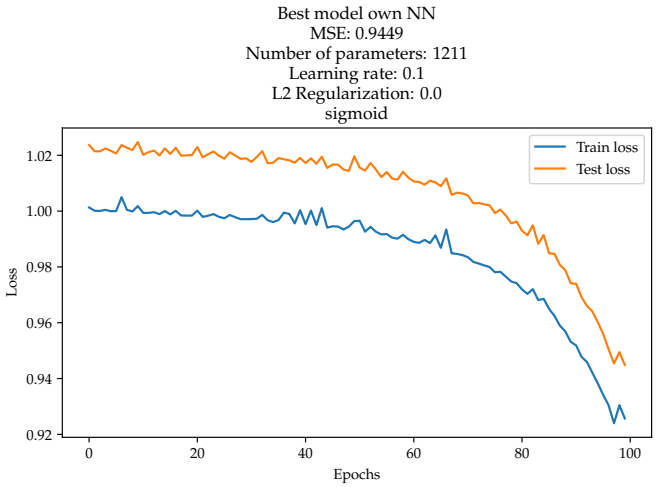
RELU - Best grid search	Results
MSE	0.1184
R2	0.8841
parameters	4629
neurons at first hidden layer	32
learning rate	0.1
lambda	0.000316228
epochs	100
batch size	300

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.

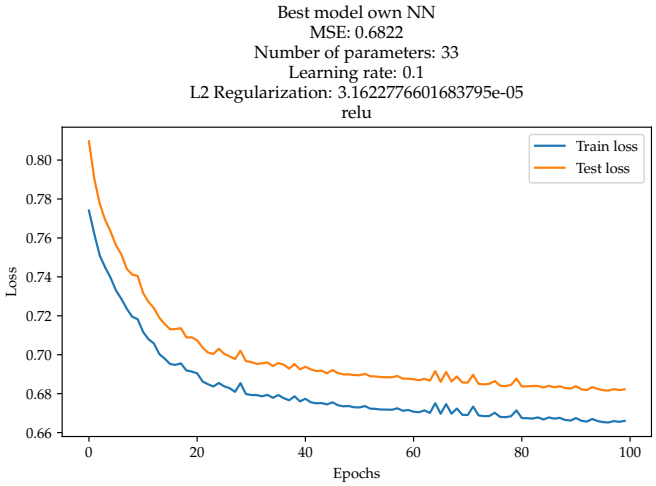
The two heatmaps shown in Figures (5) and (6) show the effect of the learning rate η as well as the l2 regularization term λ without and with a learning rate scheduler respectively. By comparing the two plots, it can be seen that the learning rate scheduler causes all but one model



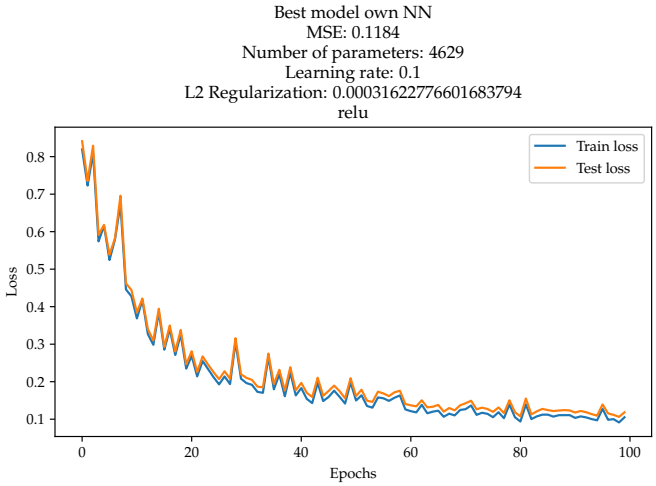
(a) Best fitted NN model using sigmoid and small arhitecture



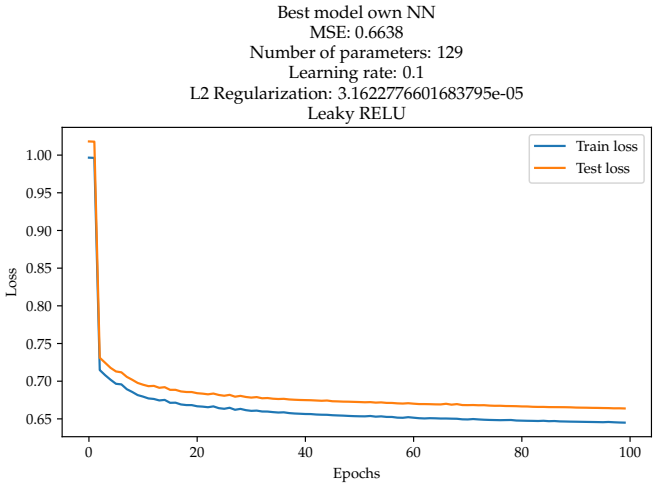
(b) Best fitted NN model using sigmoid and large arhitecture



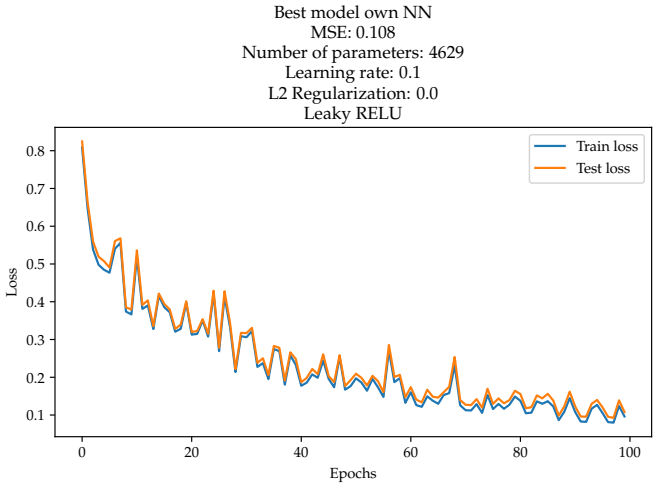
(c) Best fitted NN model using RELU and small arhitecture



(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. 11. Small vs large architecture

TABLE VI. Optimal grid search parameters for the Leaky RELU activation function

Leaky RELU - Best grid search	Results
MSE	0.108
R2	0.8942
parameters	4629
neurons at first hidden layer	32
learning rate	0.1
lambda	0
epochs	100
batch size	300

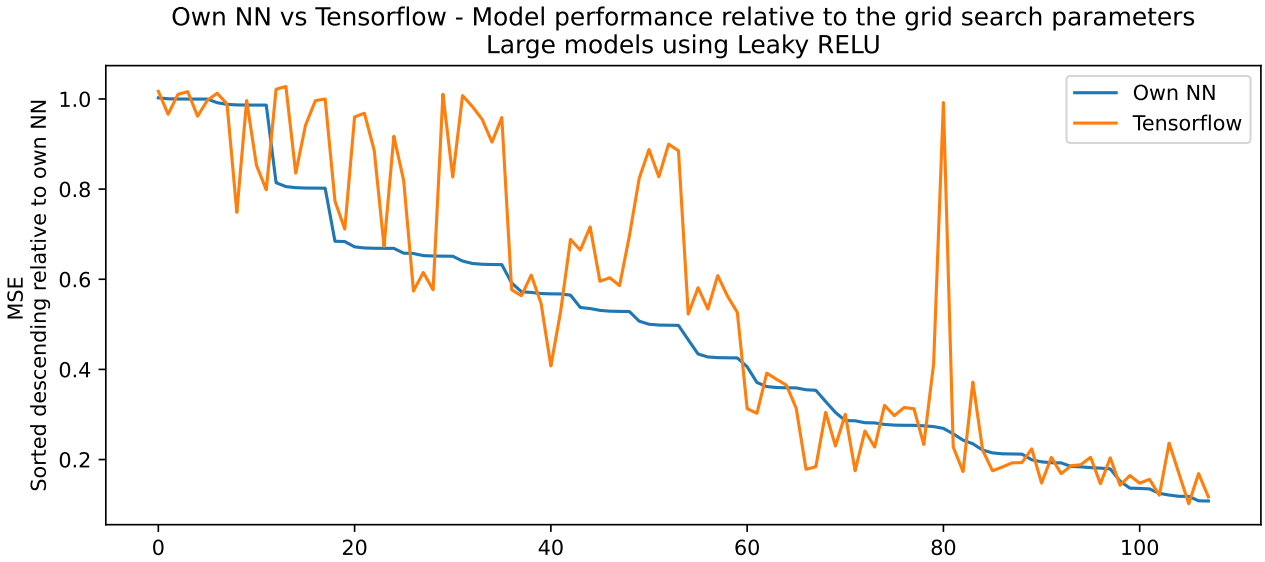


FIG. 12. MSE comparison between own NN and Tensorflow - MSE from grid search using Leaky RELU

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 (5) and (6), 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 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 (5) shows that the optimal hyperparameters for our SGD algorithm given the current dataset, is $\eta = 0.01$ and $\lambda = 0.0001$. As such, Figure (5) suggests that less regularization leads to lower attained MSE. This complies with the results of Table (III), were an even lower MSE value is attained when noe regularization is present. Furthermore, by comparing the Ridge regression value in Table (II) with the OLS value in Table (III) it can be seen that OLS scores better than Ridge regression for the optimal regularization parameter as found in Figure (5). 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 (II), it can be seen that the MSE values computed with our SGD algorithm gets close to the Ridge regression value, with an approximate 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 (III). 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 (II) and (III) is the inclusion of Momentum SGD as described in Algorithm (2). With the added momentum parameter, lower MSE values are attained as all the hyperparameters are kept

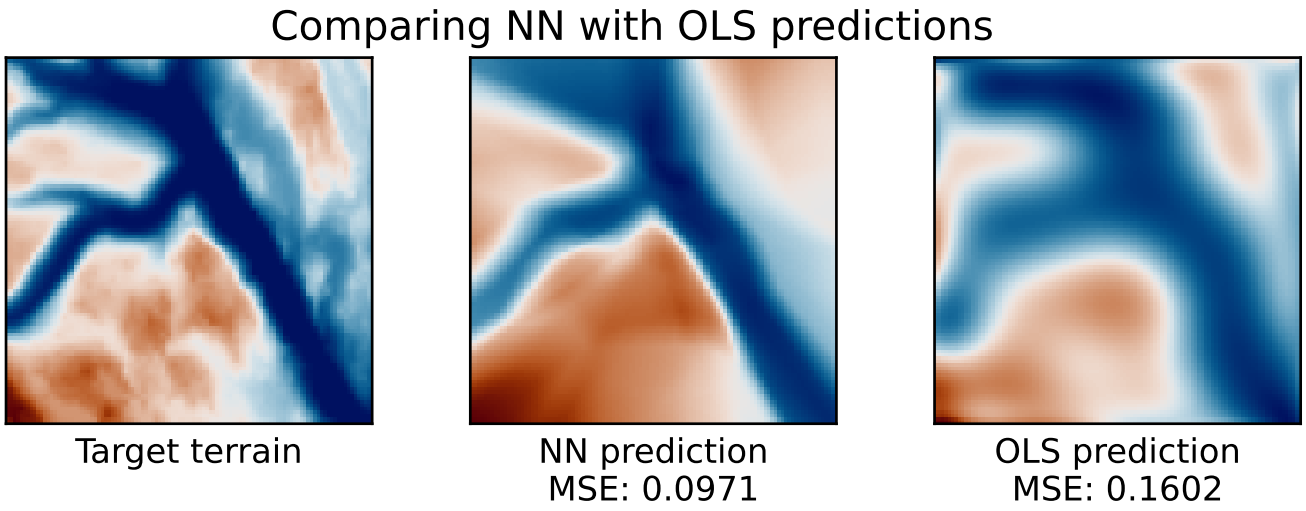


FIG. 13. 2D prediciton comparison between own NN and own OLS

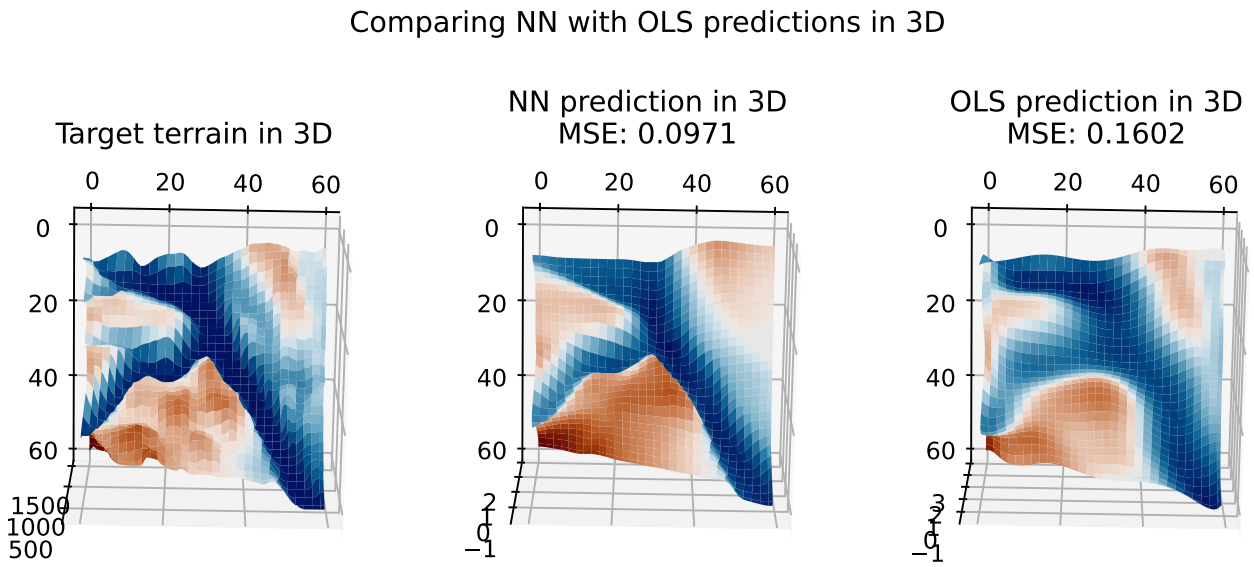


FIG. 14. 3D prediciton comparison between own NN and own OLS

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 (5) and (6), 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 [5]. 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 0.1601 as shown in figure **??**, 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 respective 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 65 features to work on. One significant difference for our Neural Net is that we only feed the NN model with the raw data, meaning we input the first-order inputs of x and y, two features in total. By doing this, the NN model receives the task of finding the optimal polynomial and its degree, and this will 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 as done for the OLS can go both ways; it can help the NN 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 NN 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 in our design matrix (order of magnitude of our raw input) and applying PCA on it is not a good choice in this case. 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 **??**. 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 **??**. The most optimal MSE values from the grid search can be found in the heatmaps for all the three activation types; Sigmoid in figure **??**, RELU in figure **??**, Leaky RELU in figure **??**. Figure **??**, **??**, and **??** shows how the similar Tensorflow model performs on the same grid search parameters. The most optimal hyperparameters for each activation function is summaries in table **??**, **??**, and **??**.

Parameters and model complexity

Figure 10 is showing how the model performance is connected to complexity. The plot is showing MSE performance for all the six model complexities having $33, 65, 129, 325, 1211, 4629$ parameters relative to the hyperparameters used during grid search. The small model architecture consisting of only one hidden layer is relating to parameter sizes $33, 65, 129$. The large model architecture consisting of three hidden layers where the second hidden has twice the debth (neurons) as the first and last layer, and the large architecture is relating to parameter sizes $325, 1211, 4629$. The vertical line in the plot for each parametercount is not a standard deviation, but it shows how the MSE varies relative to the different hyperparameters used during grid search. A complete grid search was conducted for all the six parametercounts. The variation in MSE performance for each parametercount indicates how sensitive the model is to the hyperparameters considering the activation function and the model complxity.

The sigmoid activation function

The sigmoid function seems to be the activation function more sensitive to hyperparameters out of all the activation functions tested, and the vertical lines in figure 10 for the sigmoid function looks to superseed the others in variation. Moreover, the sigmoid function seems to fit slightly better for small architectures having smaller complexity and fewer parameters. It looks like the sigmoid function is not able to harness the potential of increased model complexity that well. The sigmoid function is know to saturate at the upper and lower bounds, and this may lead to the gradients being washed out during training. This can in turn lead to the known vanishing gradient problem. Looking at the plot, we believe that the known problems with the sigmoid activation is surfacing within the figure, and that it is clear that the sigmoid is not being as potent as RELU and Leaky RELU in terms of performance and robustnes to hyperparameters.

The RELU activation function

The RELU activation function seems to tackle higher model complexity comparing the sigmoid activation function as shown in figure 10. The MSE values using

RELU are decreasing as the model complexity is increasing considering MSE relative to all hyperparameters indicated in the vertical line. The vertical lines for RELU spans a relatively short range of MSE values especially for lower model complexity, and this indicates that the RELU is maximizing the potential in smaller model complexities better than sigmoid regardless of the choice of hyperparameters. Comparing sigmoid with the RELU, it seems that RELU is able to harness the potential residing within increased model complexity better. The lowest MSE values are achieved at the highest tested model complexity using the large architecture having 4629 parameters. The reason for this improvement looks to be that the RELU do not saturate the same way as the sigmoid function. We also discovered that the RELU greatly benefits from the He initializations of weights [7] in terms of model performance. The positive effect initializing the weights using the HE approach versus plain normal distribution is significant. Before we introduced the HE initialization the RELU model performed similar to the model using sigmoid activation. Using the RELU activation function our model often encountered float64 overflow, cause of gradient explosion, and our model was very sensitive high learning rate and high lambda values for regularization. Applying the HE weight initialization drastically improved overflow issue and made our model more robust to hyperparameters. We were able to push learning rate as high as 0.1 with the HE initialization, and than discovered our best fit using a learning rate of 0.1. Furthermore, given the trend that MSE gets better at higher learning rates and higher model complexity as seen in figure 10 we believe that there is even greater potentials than what we had time to discover for our model when using RELU and HE initialization.

The Leaky RELU activation function

Building on the results and experiences using the RELU activation function, we ventured into the domain of the Leaky RELU activation function. As shown in 10 the results from the Leaky RELU activation function is further improving model performance over RELU for some hyperparameter combinations. It looks that Leaky RELU is more sensitive to the usage of hyperparameters given the longer vertical lines for Leaky RELU over its RELU sibling. However, it looks that the Leaky RELU have slightly more potential in terms of pure MSE minimization for some combination of hyperparameters over RELU.

Performance of activation functions

Comparing the best results for the different activation functions in table IV, V,VI, the Sigmoid function has the highest MSE value. 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 only using gaussian distribution with $\sim N(0, 1)$ for all activation functions. When implementing the HE [7] method for RELU and Leaky RELU, and Xavier [4] for Sigmoid we immediately got significantly better performance than with our initial weight initialization. When using the more optimal weight initialization strategies we also discovered less problems of exploding gradients during in the model training especially for RELU and Leaky RELU.

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 in terms of MSE.

In figure **??** and **??** we see that the NN model is more greedy than the OLS model when it comes to minimizing errors. It looks like the NN model prioretizing parts in the terran where it is more beneficial to minimize the error. E.g., minimizing along the deep valey and along the cliff. These are parts of the terrain that potentials yields the large error due to the steep variation in the terrain. Albeit Figure (13, c) fits the finer details in the original terrain to a larger extent than (13, b), this is circumvented by its inability to precisely fit these details. Thus yielding a positive gain to the MSE. As such, it might seem more productive to precisely fit some structures while disregarding other structures when fitting the terrain data, with regards to lowering the MSE.

This might be a derived result from the Bias-Variance tradeoff which we thourgly discussed in Project 1, were we studied MSE as result of model generalizability. Moreover, we concluded that a overfitted model in terms of its own training data would have difficulties correctly fitting new data. As the Neural Net predicts a new value using a significantly smaller amount of predictors in comparison to the OLS model ($2 < 65$), the model is not forced by the composition of predictors to fit a certain way. Which is reflected in the lack of structures in the fitted terrain compared to OLS as seen in Figures (13) and (14). However, the smaller amount of predictors does not imply that the Neural Network would not be able to overfit. Had the Neural Network been trained on fewer datapoints, the threshold for overfitting the model could have been reached at an earlier epoch.

CONCLUSIONS

Source Code

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

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Fabio Crameri. Scientific colour maps, 2021.

[3] G. Cybenko. Approximation by superpositions of a sigmoidal function. 2(4):303–314, dec 1989.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

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

[6] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly UK Ltd., October 2019.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. February 2015.

[8] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. 2(5):359–366, jan 1989.

[9] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. Last update: Thu Dec 26 15:26:33 2019.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. December 2019.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. 323(6088):533–536, oct 1986.