Neural Networks, Logistic and Linear Regression for Data Classification and Regression Problems

Fredrik Hoftun & Lidia Luque & Ada Weinert Ravn
Faculty of Mathematics and Natural Sciences, University of Oslo

(Dated: November 20, 2021)

In this project we study the relative prediction accuracy of linear and logistic regression and artificial neural networks (ANNs). We implement the feedforward neural network (FFNN) that can be used for both regression and classification problems and compare it to respectively logistic and linear regression methods including ordinary least squares (OLS) and ridge. To test the linear regression we apply both methods on Franke function data, and after careful hyper-parameter tuning, get a R2 score of 0.86 while using the FFNN. We find that this performance is significantly better than the best score otherwise achieved using the linear regression models, with the best score belonging to OLS, which attained R2 of 0.79. We further test the linear regression models and compare OLS to linear regression with stochastic gradient descent (SGD), SGD with momentum, and Adam as optimizers, and find that OLS remains as the most accurate among those implementations. For prediction of classification the neural network and logistic regression, models have been tested on the Wisconsin Breast Cancer Dataset, used to classify tumor malignancy based on set of features. We find that our fine-tuned FFNN reaches an accuracy of 98.8%, while the best model using logistic regression achieves only 97.0% accuracy.

I. INTRODUCTION

Neural networks have in the recent years become a key component of many technologies we interact with on a daily basis. They are applied in everything from Netflix's "watch next" recommendations to flagging of suspicious transactions by banks. In recent years, neural networks have made their way into medicine, where trained models can be used to help physicians make better diagnostic and treatment decisions. Their extended use is due the fact that they offer a relatively simple way of making sense of large amounts of data when an exact model is not available. Inspired by biological models of neurons, this algorithm can be adapted to suit both simple and complex problems, often within regression and classification applications. The goal of this project is to explore the performance of a feedforward neural network (FFNN) in regression and classification applications, and compare its scores to results achieved with linear and logistic regression models. We will implement ordinary least squares (OLS) and ridge regression models with and without the stochastic gradient descent including a set of optimizers. Along with the FFNN, the linear models will then be tested on the Franke's function, a function widely used for testing various fitting and interpolation algorithms, with the goal of comparing the methods in terms of solving a regression problem. Afterwards FFNN and the logistic models will be applied on the Wisconsin Beast Cancer Dataset, to evaluate their relative performances in terms of classification problems. Proper assessment of our models will be performed using comparison of the relative R2 scores. Comparing the application of different methods on the same datasets will give us an opportunity to evaluate the relative performance and precision of each of them.

In this project we will use the simplest kind of neural network, a feedforward neural network (FFNN) to study both regression and classification problems. We will first describe the necessary background theory before discussing our implementation. The results, including their discussion, will be presented afterwards, followed by a short conclusion based on our findings. All the code used in this project can be found in our GitHub [1].

II. THEORY

The necessary background regarding linear regression has already been covered in the previous project and we refer to it for the in-depth explanation of the relevant theory [1]. The course pages [2] and the book on neural networks [3] have also been used as background in this section.

A. Linear and Logistic Regression

As a summary we can say that regression is a method aiming to fit a set of measured data $\mathbf{y} = [y_0, y_1, y_2, \cdots, y_{n-1}]$ to a statistical model described by some unknown function f with $\mathbf{x} = [x_0, x_1, x_2, \cdots, x_{n-1}]$ as inputs, i.e. construct a function $y(x)$ where $y_i = y(x_i)$.

[1]https://github.com/lidialuq/fys-stk4155/tree/main/Project2/code

The *linear* in the name comes from the assumed linear (continuous) relation between the dependent variable $\mathbf{y}$ and the regressor vector $\mathbf{x}$, resulting in a simple recipe for fitting data.

The most popular linear regression model is the *ordinary least squares* (OLS), where the *mean squared error* (MSE) is used to define the cost function

$$C(\theta)_{\mathrm{OLS}} = \frac{1}{n}\{(\mathbf{y}-\mathbf{X}\theta)^T(\mathbf{y}-\mathbf{X}\theta)\}$$

Here θ is the vector of parameters resulting in the cost function being minimized, n is the number of inputs in the input data and $\mathbf{X}$ is our design matrix.

We also included the *ridge regression* algorithm in our project, with the following cost function, corresponding to weighted MSE

$$C(\theta)_{\mathrm{RIDGE}} = \frac{1}{n}\{(\mathbf{y}-\mathbf{X}\theta)^T(\mathbf{y}-\mathbf{X}\theta)\} + \lambda\theta^T\theta$$

where λ is the *regularization parameter.*

However for classification problems there is often a need for a discrete outcome corresponding to the number of potential variables (classes). What we then need is a method aiming to fit the same set of inputs x to an outcome, or class, y_i, where $i = 1, 2, \cdots, K$, and K is the number of possible outcomes.

This can be done either using the *perceptron* model, sometimes called the "hard classifier", where each datapoint x_i is deterministically assigned to a category y_i, or a *soft classifier* which outputs the probability of given category rather than a single value. The most common example of a soft classifier is the *logistic regression* model, where the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the so-called logistic (or *sigmoid*) function

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

Assuming binary output with $y_i \in [0, 1]$, we expect two parameters θ to fit our logistic function, with the probabilities of each defined as follows

$$p(y_i = 1|x_i, \theta) = \frac{\exp(\theta_0 + \theta_1 x_i)}{1+\exp(\theta_0+\theta_1 x_i)}$$

$$p(y_i = 0|x_i, \theta) = 1 - p(y_i = 1|x_i, \theta)$$

where x is a set of input variables and θ are the coefficient to estimate our data with.

This can be approximated as the product of the individual probabilities of a specific outcome $y_i \in 0, 1$ from a total likelihood of all possible outcomes from a dataset $\mathcal{D} = (y_i, x_i)$.

$$P(\mathcal{D}|\theta) = \prod_{i=1}^{n}[p(y_i = 1|x_i, \theta)]^{y_i}[1 - p(y_i = 1|x_i, \theta))]^{1-y_i}$$

With the goal of maximizing the probability of the observed data and assuming independent samples we apply *Maximum Likelihood Estimation* (MLE) principle and obtain the log-likelihood

$$\mathcal{C}(\theta) = \sum_{i=1}^{n}(y_i \log p(y_i = 1|x_i, \theta)$$

$$+(1-y_i)\log[1 - p(y_i = 1|x_i, \theta))])$$

We reorganize the algorithm with θ_0 and θ_1 as our weights. We know that the *cost function* is equal to the negative log-likelihood thus resulting in the following equation

$$\mathcal{C}(\theta) = -\sum_{i=1}^{n}(y_i(\theta_0+\theta_1 x_i) - \log(1+\exp(\theta_0+\theta_1 x_i)))$$

This is known as *cross-entropy* which will serve as our cost function for logistic regression. This is often supplemented with additional regularization terms, which will be discussed in further sections.

The first derivative of out cross-entropy cost function can be given in terms of its components

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

where $\mathbf{y}$ is a vector of elements y_i with length n, $\mathbf{p}$ is a vector of fitted probabilities $p(y_i|x_i, \theta)$ and $\mathbf{X} \in \mathbb{R}^{n\times p}$ is the design matrix containing the values of x_i.

If we then define a diagonal matrix $\mathbf{W}$ with elements $p(y_i|x_i, \theta)(1-p(y_i|x_i, \theta))$ we can also define the expression of the second derivative

$$\frac{\partial^2\mathcal{C}(\theta)}{\partial\theta\partial\theta^T} = -\mathbf{X}^T\mathbf{W}\mathbf{X}$$

Oftentimes solving for the optimal value of θ using the inverse is analytically impossible. In that case it is common to use an approach employing the numerical optimization, such as *gradient descent* to find values of θ, a method which will be discussed in the following section.

The performance of this model is then measured with the accuracy score equal to the number of correctly guessed targets t_i divided by the total number of targets n, as following

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

where I is the indicator function, 1 if $t_i = y_i$ and 0 otherwise for a binary classification problem.

B. Gradient Descent

Gradient descent is an optimization algorithm used to find minima of a function $\mathbf{f}(\mathbf{x})$ which is expected to decrease the fastest while going towards the direction of the negative gradient $-\nabla\mathbf{f}(\mathbf{x})$. When used in training of a machine learning model its applied to the convex cost function to iteratively minimize it to its local minimum. Each iterative step is taken towards the direction of the minimum until a point is reached where the cost function is as small as possible.

$\mathbf{f}(\mathbf{x_{k+1}}) \leq \mathbf{f}(\mathbf{x_k})$ for a small enough learning rate $\eta_k > 0$ when

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \eta_k \nabla\mathbf{f}(\mathbf{x_k})$$

This gives a basis for the method of gradient descent, also called *Steepest Descent*. Ideally its value will converge towards the global minimum of the function $\mathbf{f}(\mathbf{x})$, which is guaranteed in the case that $\mathbf{f}(\mathbf{x})$ is a convex function.

While simple to implement this method has multiple severe limitations. It is highly sensitive to the choice of learning rate η_k due to the relation between output at each step described above. With small values the function will take a long time to converge, and in the opposite case we can expect to see erratic behaviors on the output. It is also expensive with regards to computation time. Probably the biggest downfall though is the risk of finding the local over the global minima and becoming stuck. Due to its deterministic nature without careful selection of initial parameters finding the local minimum instead might be unavoidable.

1. *Stochastic Gradient Descent*

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above. We can rewrite the cost function which we want to minimize as a sum over n datapoints. This allows for the gradient to be estimated as a sum of i partial gradients

$$\nabla_\theta C(\theta) = \sum_{i}^{n} \nabla_\theta c_i(x_i, \theta)$$

We introduce stochasticity by taking the gradient of a subset of data, so-called minibatches. Batches are defined with B_k, where $k = 1, \cdots, n/M$, for minibatch size of M as a subset of n data points. This results in the approximation of total gradient using a sum or gradients of selected minibatches

$$\nabla_\theta \mathcal{C}(\theta) = \sum_{i=1}^{n} \nabla_\theta \mathcal{C}_i(x_i, \theta) \rightarrow \sum_{i\in B_k}^{n} \nabla_\theta \mathcal{C}_i(x_i, \theta)$$

where k is chosen randomly with equal probability $[1, n/M]$. This gives us a gradient descent step as following

$$\theta_{j+1} = \theta_j - \eta_j \sum_{i\in B_k}^{n} \nabla_\theta \mathcal{C}_i(x_i, \theta)$$

where η_j is the learning rate at step j.

Minimization can be done by iterating over the gradients and weighting them with the *learning rate* η.

$$\theta \leftarrow \theta - \eta \nabla \mathcal{C}(\theta)$$

An algorithm will then sweep through the training set and update the beta values until in converges. This convergence is calculated using the cost function of the regression method. It is possible to divide the gradients into several mini-batches with a given size. If the number of batches $M = 1$ we have the usual SGD, but we have SGD with mini-batches if $M > 1$. Here we have defined the size of the batch as $S = n/M$, which gives a larger size if M is small and a smaller size if M is large.

2. *SGD with Momentum*

The regular SGD method comes with modifications. Momentum is the simplest modification. It adds an exponential decay factor $\gamma \in (0, 1)$ that acts as inertia on the gradient, preventing oscillations. This is analogous to momentum in physics. The minimization iterative scheme looks like this:

$$\alpha \leftarrow \gamma\alpha + \eta\nabla\mathcal{C}(\theta)$$
$$\theta \leftarrow \theta - \alpha$$

In addition to the regular SGD we have also implemented two additional methods, namely SGD with momentum, aka *momentum*, and *Adam* (Adaptive Moment Estimation).

3. *Adam*

The most advanced SGD modification comes with Adam. Adam adds two hyperparameters β_1, the forgetting factor of the gradient, and β_2, the forgetting factor for the second moment of the gradient. The forgetting factors help average out the gradients and adapt η, so it does not require a learning schedule. The minimization iterative scheme looks like this:

$$\begin{aligned} m &\leftarrow \beta_1 m + (1-\beta_1)\nabla C(\theta) \\ v &\leftarrow \beta_2 v + (1-\beta_2)(\nabla C(\theta))^2 \\ \hat{m} &= \frac{m}{1-\beta_1^{t+1}} \\ \hat{v} &= \frac{v}{1-\beta_2^{t+1}} \\ \theta &\leftarrow \theta - \eta\frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon} \end{aligned}$$

Where t in β^{t+1} means the timestep and ϵ is a small number to prevent division by zero.

C. Neural Networks

Artificial neural networks (ANN) are biologically inspired computational systems, which can learn to perform various tasks based on provided input examples. Their functionality aims to mimic the biological brain, with interconnected neurons sending signals among each other in the form of mathematical functions. In this project we will focus exclusively on the simplest implementation of this system called the *feedforward neural network* (FFNN), where the information only follows one direction: forward. The specific type we have developed is often called *multilayer perceptron* (MLP), which is characterized by the existence of *hidden layers* in between the input and output layers, each consisting of multiple nodes. An MLP is a *fully- connected neural network*, which means that each neuron is connected to all neurons in the previous layer. A general representation of this system can be seen in figure 1.

1. *FFNN Architecture*

The system is represented by a set of neurons, also called nodes, arranged in layers and interconnected with each other as shown on figure 1. In each layer the neurons are connected to the neurons of the previous layer with strength of connection represented by weights w. We will see that each neuron in our network can be defined by a model function, and as specific for feedforward network, can only pass information to the neurons ahead of it. This causes the input information to flow through the network from the input layer, through the hidden layers and towards the output, without backtracking or loops. We will now take a look at the structure of each neuron.

On each node all inputs p_i are multiplied by their corresponding weights w_i and then passed into the node to

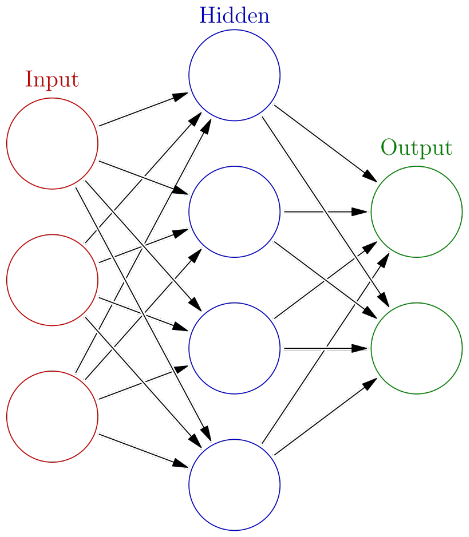

Figure 1. The figure shows a simple schematic of a feedforward neural network. It consists of three layers, an input layer, a hidden layer and an output layer. Each of the layers have a varying number of nodes, e.g. the input layer has three nodes. The nodes are connected to every node in the layers next to its own. Note that the number of hidden layers can vary and this is an example with one hidden layer. Image taken from Wikipedia Commons [4].

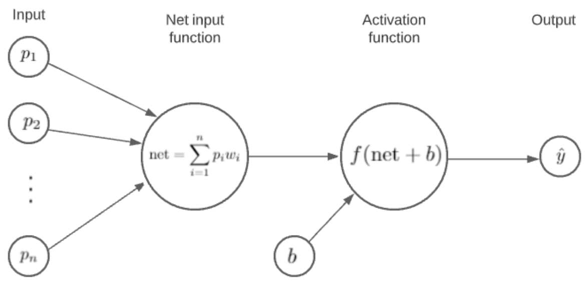

Figure 2. The figure shows a model of a single neuron in our artificial neural network. We calculate the net sum of the inputs p from the previous layers multiplied by weights w. This, together with bias b, then serves as an input to the activation function f which creates the predicted output $\hat{y}$. Same process then repeats on the following node, this time with $\hat{y}$ passed on to the input.

be evaluated by the activation function. The evaluated output, denoted by $\hat{y}$ can then be describes as the following

$$\hat{y} = f(\sum_{i=1}^{n} w_i x_i + b_i) = f(z)$$

where b_i is a small bias added in case of zero activation weights on inputs. The model of this equation is shown in Figure 2.

In the case of the first hidden layer the inputs p_i our inputs are the initial inputs containing the features of our dataset and in the following layers l this is replaced by the predicted outputs of the nodes in layer l_{i-1}. We initialize the input weights with random non-equal values, and expect them to result in a wildly inaccurate predicted output on the first forward pass. Finally the bias b_i is set to be a small, non-zero value.

This process continues until we reach the output layer, where we adjust the weights and biases before making another, more accurate forward pass. This continues for a given number of epochs (passes) until our accuracy score is satisfied.

2. *Activation Functions*

The purpose of the *Activation Function* is to add non-linearity to the neural network. Also referred to as the *Transfer Function*, it is a function applied to the output of a node to determine its activation. Without it the output would not hold for any FFNN, no matter its depth, and would result in a linear solution. This is in accordance with the *Universal Approximation Theorem*, which states that a neural network with at least one hidden layer can approximate any continuous function.

The *(logistic) sigmoid* activation function was one of the first activation functions used in neural networks. It is still used today, mainly as the activation function in the output layer, in models where a measure of probability is needed as an output. While it can also be applied to the hidden layers, it often leads to vanishing gradients, especially for deeper neural networks with several hidden layers. It follows the definition below

$$f_{\text{SIGMOID}}(z) = \frac{1}{1+e^{-z}}$$

with output in range $[0, 1]$. The above-mentioned vanishing gradient problem occurs due to very large and small inputs to the sigmoid function leading to null gradients.

A more commonly used activation function, in terms of its application to the hidden layers is the basic *Rectified Linear Unit* (ReLU) function

$$f_{\text{ReLU}}(z) = \max(0, z)$$

with output in range $[0, \infty]$. However even this function has a significant downside, which lays in its failure to correctly map negative values on input, immediately turning them to zeros. This decreases the ability of the network to learn and is known as the dying ReLU problem.

An attempt to solve the above issue results in one the many variants of the ReLU, called the *Leaky ReLU*

$$f_{\text{LEAKYReLU}}(z; \alpha) = \begin{cases} z, & \text{if } z \geq 0 \\ z\alpha, & \text{if } z < 0 \end{cases} \tag{1}$$

Here the leak α increases the range of the function bringing it to $[-\infty, \infty]$.

While there are many other activation functions, the scope of this project only covers the three mentioned in this paragraph, namely: logistic sigmoid, ReLU and leaky ReLU.

3. *Back propagation*

The data is trained based on the value of the selected *cost function* $\mathcal{C}$ comparing our network output with the training data. The more accurate the model the lower the cost output from $\mathcal{C}$. To minimize said value we need a method of adjusting the weights and biases by estimating the gradient of the cost function, a process called *back propagation.*

In the equations below f is our activation function, $f'(z_j)$ refers to its derivative for activation z, j refers to node, k to class or entry, L denotes layer, and w and b are our weights and biases respectively.

In general terms the output error $\hat{\delta^L}$ is computed using the following

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

and the back propagate error for each layer $l = L-1, L-2, \cdots, 2$ is given by

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

The weights and biases are then updated for each layer l according to

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

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l$$

where η is the learning rate, defining the rate of adjustments of the two parameters for each epoch.

4. Training Neural Network and the Cost Functions

While back propagation is the process of adjusting weights and biases, an optimization process is needed to train the model. In our case we will apply the *Stochastic gradient descent* process (discussed in section B), where the gradient refers to the error gradient of the predicted outcome. The algorithm seeks to change the weights and biases applied on the input nodes, in a way that reduces the error, lowering the value of the gradient. *cross-entropy* and *mean squared error* (MSE) are the two main types of the loss functions, used for this purpose, and the only ones covered in the scope of this project. These functions, also called *cost functions* aim to reduce the aspects of a possibly highly complex system down to a single scalar value, a number according to which the solutions are ranked and compared. The cost functions used are listed below

$$\mathcal{C}_{\text{CROSS-ENTROPY}} = -\sum_{i=1}^{n}(y\log(p) + (1-y)\log(1-p))$$

$$\mathcal{C}_{\text{MSE}} = \frac{1}{2}\sum_{i=1}^{n}(\hat{y_i} - y)^2$$

Here y and $\hat{y}$ are the true and predicted targets respectively and p is the predicted likelihood of the observation.

Another term is commonly added to the cost function, proportional to the size of its weights. This allows a better control of the weights and reduces overfitting of our model prediction. We applied the measure of the weight size called *L2-norm* resulting in the following addition to our cost functions:

$$\mathcal{C}_{\text{CROSS-ENTROPY}} \leftarrow \mathcal{C}_{\text{CROSS-ENTROPY}} + \lambda\sum_{ij} w_{ij}^2$$

$$\mathcal{C}_{\text{MSE}} \leftarrow \mathcal{C}_{\text{MSE}} + \lambda\sum_{ij} w_{ij}^2$$

where λ is known as the regularization parameter.

III. DATASETS

We will test the methods introduced in the theory section on two different datasets. Our regression algorithms will be tested on data sampled from a two-dimensional function called Franke function. To test our classification algorithms we will used the Wisconsin Breast Cancer Dataset, where tumor malignancy is predicted from tumor features.

A. Franke Function

The Franke function is a test function commonly used to evaluate different surface interpolation techniques. It consists of a surface with "twin Gaussian peaks and a sharper Gaussian dip superimposed on a surface sloping towards the first quadrant"[5]. The function, given by a weighted sum of exponentials, is evaluated for $x, y \in [0,1]$. We have sampled this function at 400 uniformly distributed points and added stochastic noise ϵ generated from a normal distribution with 0 mean and 0.1 variance. We fit the data to a polynomial of degree 3 the design matrix will thus consist of all 10 combination of features of degree ≤ 3. In this work we use a train-test split of 70%-30%, and normalize both the train and the test sets by subtracting the mean of the train sample. All the steps used to make the dataset can be found in `code/data/franke_function.py`.

B. Wisconsin Breast Cancer Dataset

The Wisconsin breast cancer dataset is a classic dataset used to test machine learning algorithms. It was created in 1995 and consists of 30 features extracted from images of breast tissue biopsies. There are 569 datapoints, of which 212 are classified as malignant -labeled as 0-, and the remaining 357 as benign -labeled as 1 [6]. The data used in this project was taken from the University of California Irvine (UCI) Machine Learning Repository. Before using this dataset, we divide the data into a train and a test set using a 70%-30% split. We also normalize the data by, for each feature in the train and test dataset, removing the mean and dividing by the standard deviation of the feature in the train dataset.

IV. IMPLEMENTATION

A. OLS and Ridge Regression with SGD

Our SGD algorithm is very flexible, it calculates the gradient, its cost and, using one of the three gradient descent methods, it estimates the parameter θ. The algorithm is implemented in a class called "SGD" and takes the following inputs:

- The activation function, here linear, which returns itself
- The gradient method, either "ordinary" or "ridge" for ridge regularization

- The design matrix train set
- The true value train set
- The learning rate η
- The ridge hyperparameter λ
- The maximum number of epochs
- The number of mini-batches M, where the size is N/M, where N is the number of parameters θ to be estimated
- Optionally the tolerance of the loop, i.e. if $|C_1 - C_0| < tol$ terminate the loop. Here C_1 is the cost in step 1 and C_0 is the cost in step 0. The predefined value is 10^{-7}

In addition to these parameters, momentum needs the exponential-decay factor γ to be specified, and Adam needs the forgetting factor for the gradients β_1 and the forgetting factor for the second moments of gradients β_2.

B. Neural Network

We have implemented the easiest, most well known neural network: a feed forward neural network. Our implementation is quite flexible, with the following variables being user-defined upon initialization of our FFNN class:

- The number of input and output nodes
- The number of hidden layers, and the number of nodes in each of these layers
- The activation functions for the hidden layers and, independently, for the output layer. We have implemented the Sigmoid, ReLu and LeakyReLu functions.
- The cost function. We have implemented mean square error and binary cross entropy.
- The initialization of the weights. Our implementation has two choices: Normal distribution, with mean 0 and variance 1, and Xavier normalized distribution. The later is a uniform distribution with limits $\pm\sqrt{6/(n+m)}$ where n is the number of neurons in the previous layer, and m is the number of neurons in the current layer.
- The initialization of the biases. All biases are initialized to the same user defined value (we only use 0 and 0.01 as initializations).

Four more variables are defined upon training of the network:

- The learning rate
- The value of the for L2 regularization coefficient
- The maximum number of epochs to train for
- The minibatch size

The feed forward and the back propagation methods of the network class work as explained in the theory section for the general case, but only stochastic gradient descent (with L2 regularization, if so chosen) is implemented.

The train method behaves as expected by calling the feed forward and the back propagation methods to update the weights and biases for each iteration for every epoch, but also stops the training once one of these conditions have been met:

- The chosen maximum number of epochs to train for is reached
- Exploding gradients cause an overflow event
- The difference in the train loss between consecutive epochs has been less than 1e−6 for the last 10 epochs

C. Logistic Regression with SGD

Akin to linear regression, logistic regression works in a similar way by using the "SGD" class. However a big difference this time is that the activation function is not linear, but is the sigmoid function. The tolerance value in the class is smaller for logistic regression, which increases precision but decreases computation speed.The test-set is scored by measuring the accuracy.

V. RESULTS AND DISCUSSION

A. Linear Regression with Stochastic Gradient Descent

We will start by doing hyperparameter tuning on our linear regression algorithms with an iterative optimizer ("vanilla" SGD, momentum SGD or Adam) using the Franke function dataset. For comparison, we will use the standard analytic OLS and ridge regression, with which we get R2 scores of 0.792 and 0.790 (for ridge we used $\lambda = 10^-6$ which in our experience from Project 1 gives good results).

We start by tuning the decay factor γ used in SGD with momentum. After trying six linearly spaced values between 0.001 and 0.9999 we find that $\gamma = 0.80$ gives the highest R2 score. The Adam optimizer uses the forgetting factors β_1 and β_2 (first and second moment of gradients). To tune these we do a grid search with five linearly spaced values for each of $\beta_1 \in [0.900, 0.999]$ and $\beta_2 \in [0.99000, 0.9999]$. We get the best preforming model with $\beta_1 = 0.9495$ and $\beta_2 = 0.9999$.

With these parameters tuned, it is now time to tune the hyperparameters that are used in all three optimizers: the learning rate η, the number of mini-batches M and the regularization parameter λ. We do a 3D grid search over these parameters for ridge regression with each of the three optimizers. The parameters resulting in the lowest MSE are given in table I. For completion, the grid search results -with M already tuned- for "vanilla" SGD, momentum STG and the Adam optimizer are included in the appendix in figures 8, 9 and 10 respectively.

Method	η [10^{η}]	λ [10^{λ}]	M
Vanilla test-set	0.444	-4.25	20
Momentum test-set	0.444	-4.25	20
Adam test-set	-0.667	-4.25	20

Table I. Best hyperparameters η, λ and M for different optimizers. Values taken from the best preforming parameters in figures 8, 9 and 10.

To get out final results, we randomly sample the Franke function, in the method explained in section III A, 300 times. Table II shows the averaged R2 scores for the different linear regression methods. The first thing to note is that setting λ to 0 (that is using OLS instead of ridge), gives the best results for all gradient descents methods. ridge does preform slightly better than OLS when not using gradient descent, but the differences in scores are very small. It seems that regularization is not needed when using linear regression on this dataset, and indeed often decreases performance. The likeliest reason for this is that since we use a relatively large train dataset -270 datapoints- and only 10 features, the model doesn't overfit, and thus regularization only increases the bias without decreasing the variance. Let's now move on to the differences between the optimization methods. Clearly the analytic methods without gradient descent do the best job. This isn't surprising: an analytic solution is always going to perfectly minimize the cost function, while gradient descent can only approach the minima. However, an analytic solution is often not available -that's what gradient descent is for. In that case, our best bet would be using either SGD with momentum or the Adam optimizer -momentum preforms slightly better- which both preform significantly better than the "vanilla" SGD. Momentum SGD decent might be preferred due to it's lower computational footprint compared to Adam.

	Ordinary	Ridge
Method	*Mean R2 score*	*Mean R2 score*
OLS regression test-set	0.792	
Ridge regresion test-set		0.790
Vanilla test-set	0.729	0.726
Momentum test-set	0.783	0.767
Adam test-set	0.781	0.760

Table II. Test mean R2 score from linear regression on the Franke function data. First two rows shows the OLS regression and ridge regression ($\lambda = 10^{-4}$) results. The later rows show the various SGD methods.

B. Regression with Neural Network

After having seen the gradient descent method at work, we will now explore making use of it in our neural network. As we've just seen, the momentum SGD preforms better -when used in linear regression- than the standard SGD. However, for simplicity, we have only implemented SGD in out neural network. Our job here will be to tune the network to get the best possible performance given this constrain.

As stated in the methods section, we use the mean square error as the cost function when doing regression. The MSE error is the standard choice for the cost function when doing regression. There are good reasons for this: it is a simple function, it is easy to see how minimizing it leads to a good fit, and it has a simple derivative which is proportional to the difference between the predicted and the true output. While other functions would also work -for instance other exponentials of the mean error- the MSE is simple and works, so there is no reason not to use it. Choosing or tuning hyperparameters is a harder task. The first obvious one to look at are the activation functions. Since this network preforms regression, the output should be a number $a \in \mathbb{R}$. the easiest way to accomplish this is simply not having an activation layer in the output layer, which is also the standard when doing regression. In the code, for reusability proposes, the output layer activation function is not removed but instead set to *linear*, which will merely return the input unchanged. As for the activation functions of the hidden layers, there are many choices, but we will use the *sigmoid* function as a starting point. In keeping with the "keep it simple" theme, we will do our first experiment on a network with a single hidden layer with 50 nodes and train it for a maximum of 1000 epochs with mini-batches of 10 samples. The reason? We have got to start somewhere, and this strikes a good balance between run time and outcomes.

We now have two more hyperparameters to tune: the learning rate η and the regularization coefficient λ. Figure 3 shows the R2 test scores resulting from a gridsearch of η and λ. It shows that, for the aforementioned hyperparameters, the fit has a peak R2 score with a learning rate of 0.1 and a regression coefficient of 1e−5 and/or 1e−6. This is a quite high learning rate, and using a more complex neural network (increasing the number of nodes or hidden layers) quickly leads to exploding gradients in the back propagation, which makes it impossible to train the network. For instance, using the same network but doubling the amount of nodes in the hidden layer to 100 leads to exploding gradients for all regularization coefficients when the learning rate is 1e−1. Luckily for us, the R2 is very stable in this region, and all combinations of η = 1e−2 and 1e−1 with λ = 1e−3 or lower give R2 scores of between 0.70 and 0.72.

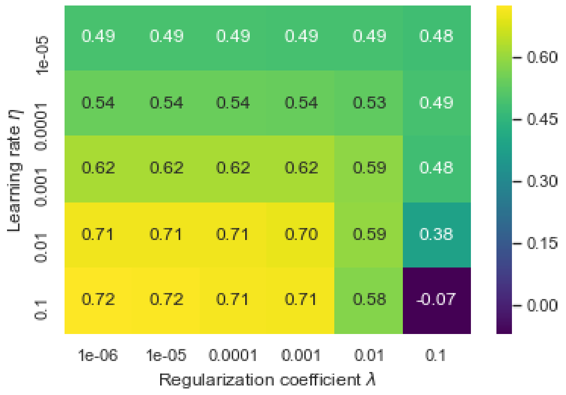

Figure 3. Test R2 scores for different combinations of learning rates and regression coefficients. The networks are trained for a maximum of 1000 epochs.

Is it possible to increase these scores? Out first attempt at it will be to simply increase the maximum number of epochs we train for 10-fold, which results in figure 4. Unfortunately, the higher R2 scores do not increase. However, the scores do generally increase for lower learning rates. This makes sense: the training for higher learning rates requires less epochs, and in our case the training stops at about 1000 epochs for the higher training rates, so increasing it further has on effect, while increasing it for lower training rates where the network was still improving, does. All in all the increases, where present, are small, and since we do not see improvements for the better preforming learning rates, we will from now on use a maximum number of epochs of 1000.

There is one more important pattern to discuss in figures 3 and 4: Regularization with large coefficients hurts the fit, and once λ gets to about 1e−3 and below, the coefficients are too small to have an effect. From this it seems fair to say that regularization does not help in this problem. To try to piece out why this is the case, we have done four experiments: Increase the noise of the input data, increase the complexity of the FFNN by increasing the neurons in the hidden layer and the number of hidden layers (separately), and increasing the maximum epochs for training. The goal was to achieve overfitting, in which case it it likely regularization would help. However, the training and testing R2 scores remained very close to each other in all these tests. It appears that this network does not manage to minimize the cost function enough for overfitting to be a problem. Thus, from here on we will set the regression coefficient λ to 0.

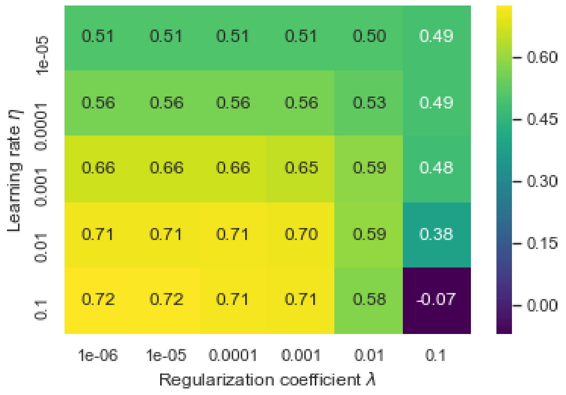

Figure 4. Test R2 scores for different combinations of learning rates and regression coefficients. The networks are trained for a maximum of 10000 epochs.

Next, we will look at different activation functions for the hidden layers. Figure 5 shows the test results of our FFNN with the hyperparameters discussed above (including $\lambda = 0$ and setting training to a maximum of 1000 epochs) and three different hidden layer activations: Sigmoid, ReLu and LeakyReLu (with $\alpha = 0.01$, which is the standard). ReLu and LeakyReLu preform almost exactly the same and far above the sigmoid activation. We are now getting R2 scores of 0.83 for learning rate 0.01, which is over 0.1 over the best score using the sigmoid function. We do have to choose one of them in order to continue tuning the network. With such small -and probably not significant- differences between ReLu and LeakyReLu, it likely does not matter. We will go with ReLu, in keeping with our keep-it-simple motto.

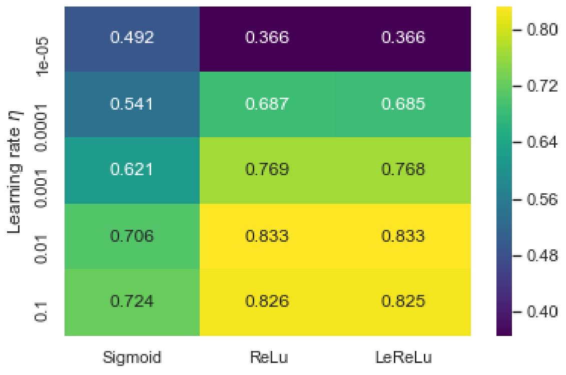

Figure 5. Test R2 scores for different combinations of learning rates and activation functions.

It is now time to look at the actual structure of our

FFNN. Up until now we have used a very simple network with only one hidden layer consisting of 50 neurons. Figure 6 shows the results of grid search with different hidden layer structures and learning rates (note that we have excluded the smallest η-values, which had poor performance in all our experiments). The more complex network consisting of four hidden layers gives the highest score, with a test R2 of 0.871 when $\eta = 0.01$. Note, however, that the same hidden layer structure also leads to the *lowest* R2 score when $\eta = 0.1$. An explanation for this might be that the cost function of a network with many hidden layers is more complex and in a sense has more "fine structure", meaning that higher learning rates run the danger of missing the global minimum and thus are unable to optimize the function reliably. We have tried rerunning this analysis five times, and in two of the experiments the network run into exploding gradient issues for this hyperparameter combination ($\eta = 0.1$ on the most complex network). The general trend we have seen is that training networks with complexity much beyond our 50-100-100-50 structure often fails due to exploding gradients. Setting a low enough learning rate fixes this issue, but the network takes too long to converge and preforms poorly.

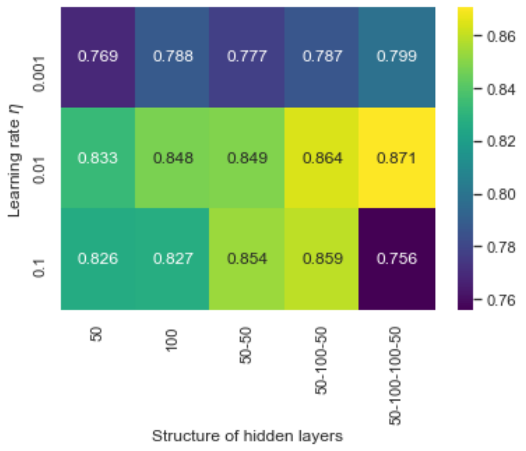

Figure 6. Test R2 scores for different combinations of learning rates and network structures. The network structures are, from left to right: One hidden layer with 50 neurons, one hidden layer with 100 neurons, two hidden layers with 50 neurons each, three hidden layers with respectively 50, 100 and 50 neurons and four hidden layers with respectively 50, 100, 100 and 50 neurons.

Up until now we have used the Xavier normal initializer to initialize the weights. Why not a simple normal distribution? While experimenting on the classifier network, we found that this initialization usually outperforms initializing with the normal distribution. More importantly, it leads to a very significant reduction in overflow events produced by exploding gradients. This is specially the case for complex networks such as our best preforming one, and in fact, training this network with weights initialized using the normal distribution leads to overflow for all 20 initialization samples we have tried.

The last experiment we will do is comparing two kinds of bias initialization: setting all biases to 0, which is the most common initialization, or setting them to a small value which we will set to 0.01. The reasoning behind the later is that a small non-zero bias ensures that no neuron becomes zero in the first feed forward pass, and thus all have a gradient to back propagate. In practice, however, this does not make a difference for this network. We trained the network twenty times with randomized normally distributed starting weights and initialized the bias to 0, then did the same but changes the bias to 0.01. The mean R2 test score for the experiment with initial bias 0 was 0.8610, while setting the bias to 0.01 gave a mean of 0.8609. In other words, there is no significant difference between these initializations. It is also worth noting that the variance in the scores was small, in the order of 1e−5. This is why we can get away with doing most of our experiments only once, as long as the starting weights are chosen from the same distribution, their exact values do not significantly change the outcome.

Table III shows the hyperparameters of the FFNN we have experimentally found to preform best for this regression problem. With a mean R2 test score 0.86, its predictions are far from perfect. How does this compare to Scikit-Learn's implementation? When choosing the hyperparameters so as to have a network as close as possible to ours (*scikit-learn*'s implementation, *MLPRegressor*, does not let the user choose weight and bias initializations), the resulting R2 test score was 0.87, very similar to our network's.

Lastly, we compare our FFNN R2 scores to the results obtained using the linear regression method that preformed best, as tested in section V A, namely OLS. The most basic implementation of OLS, not including the gradient descent, received the overall highest score among the linear regression models with a R2 score of 0.79. This is significantly lower than our network's score 0.86. This is somewhat of an expected result, as the ideal target does not align with the linear function output, even with removal of the added random noise. Additionally while linear regression can only predict assuming linear dependencies, neural networks should be able to predict the non-linearities in the input data. Following that assumption increasing the degree of the polynomial with which we model the data should reduce the difference in performance between the OLS and the neural network implementation, as the relationship becomes more linear. Testing this is, however, outside the scope of this project.

Hidden layers	4
Neurons per hidden layer	50, 100, 100, 50
Learning rate	0.01
Activation in hidden layers	ReLu
Activation in output layer	None
Weight initialization	Xavier Normalized
Regularization coefficient	0
Maximum training epochs	1000
Minibatch size	10

Table III. Specifications for our regression FFNN after hyperparameter tuning.

C. Breast Cancer Classification with Neural Network

At its core, a neural network attempts to find the set of weights and biases that minimize a given cost function. Since neural networks use a very general way of optimizing this function -gradient descent-, as long as said function suits the problem at hand, the network itself is highly reusable. We can thus adapt out neural network to do a classification problem. We will use the Wisconsin breast cancer dataset to test this network.

Out starting point is a neural network that optimizes the binary cross entropy cost function with a sigmoid function as the activation function in the output layer. Using a bound function such as the sigmoid is important in classification problems, as we ultimately want to convert the output value into a binary output by applying a threshold. As for the activation functions in the hidden layers, we will start off by using ReLu, since this worked well for the regression problem. We choose a simple model structure -one hidden layer with 50 neurons- and train for a maximum of 100 epochs. Figure 7 shows the heatmap with the test accuracies for different learning rates and regularization coefficients. The first thing to note is how remarkably high the accuracies are for many of the combinations. Getting accuracies upwards of 90% for such a large range of parameters, using such a simple network, indicates that this is an easy classification problem. Indeed, simple linear regression gets 96% accuracy in literature [7]. We get our best accuracy of 98.8% with $\eta = 0.01$ and $\lambda = 0.1$. Table IV shows the hyperparameters of this neural network.

While it has been easy to get to a accuracy of almost 99%, getting the last percent is a lot more challenging. We have tried using different network structures and activation functions for the hidden layers (sigmoid and leakyReLu) without any luck. Indeed, we have not been able to get the accuracy any higher than 98.8%. Using *Scikit Learn*'s neural network implementation for classification, *MLPClassifier* with similar parameters gives a comparable result of 98.2% accuracy (some hyperparameters, like weight initialization, cannot be chosen in this

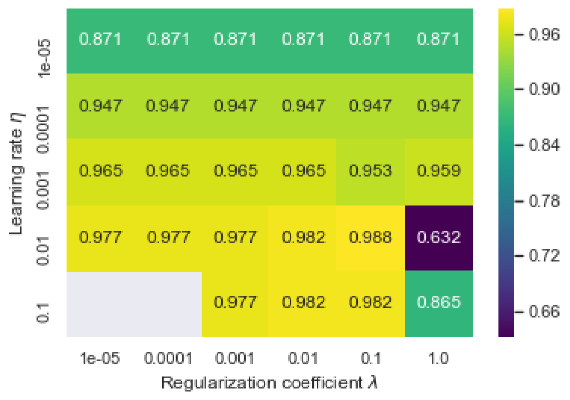

Figure 7. Test accuracies for different combinations of learning rates and regularization coefficients. Grey indicates that the network could not be trained for that combination of parameters due to exploding gradients.

Hidden layers	1
Neurons per hidden layer	50
Learning rate	0.01
Activation in hidden layers	ReLu
Activation in output layer	Sigmoid
Weight initialization	Xavier Normalized
Regularization coefficient	0.1
Maximum training epochs	100
Minibatch size	10

Table IV. Specifications for our classification FFNN after tuning of the hyperparameter.

implementation). Higher accuracies have been reported in literature with other machine learning methods. For example, Nguyen et al. show that a random forest classifier and feature selection techniques can get an average accuracy of 98.8% on the same dataset [8]. However, implementing such approaches is outside the scope of this work, and we settle for our neural network as shown in table IV.

Up until now we have only looked at the accuracy as a measure of performance. Accuracy is a simple measurement, but has two major flaws: it does not take into account unbalanced labels and it treats all true and false predictions the same. Let us start with the former: this dataset consists of 63% benign cases. This means that getting 63% accuracy is trivial -classifying all datapoints as benign will do. Given that the accuracies we get are much higher than this threshold, this is not a big concern, but it it worth keeping in mind. The model with $\eta = 0.01$ and $\lambda = 1$, for instance, has an accuracy of 63%, which in this case means it is no better than chance. The later is of more importance to us: Since we are trying to diagnose malignant tumors, we might have a greater tolerance towards false positive than false negatives. Table V shows the confusion matrix for the model with 98.8% accuracy. There are two misclassified datapoints, both false negatives. In other words: Based on this model we would wrongly tell two women that they do not have cancer, when they actually do. Presumably it would be better for the model to be cautious and classify tumors as malicious when in doubt, so that they can be checked with other methods. The easiest way of accomplishing this is by changing the threshold in the model output. Table VI shows the confusion matrix for the same model, but with the threshold for classification as benign set to 0.8. Here there are no false negatives, but this comes at the cost of 9 false positives which reduce the accuracy to 94.7%.

	Pred. malignant	Pred. benign
True malignant	61	2
True benign	0	108

Table V. Confusion matrix for our FFNN implementation. A threshold of 0.5 is used on the output.

	Pred. malignant	Pred. benign
True malignant	62	0
True benign	9	99

Table VI. Confusion matrix for our FFNN implementation. A threshold of 0.8 is used on the output.

D. Breast Cancer Classification with Logistic Regression

We will now compare our network to our implementation of a simpler classification algorithm, logistic regression.

Hyper parameter tuning is done in the same way as in linear regression. We start by tuning the optimizer-dependent hyperparameters, and find that for momentum SGD, $\gamma = 0.80$ gives the best test accuracy. For Adam, $\beta_1 = 0.92475$ and $\beta_2 = 0.99495$ gives the best performance.

With these parameters tuned, we now turn to the remaining hyperparameters: the learning rate η, the number of mini-batches M and the regularization parameter λ. As in section V A, we do a 3D grid search over these parameters for ridge regression with each of the three optimizers. The parameters resulting in the highest accuracy are given in table VII. For completion, the grid search results -with M already tuned- for "vanilla" SGD, momentum STG and the Adam optimizer are included in the appendix in figures 11, 12 and 13 respectively.

We can see from figure 11 and 12 that the accuracy is high for a wide range of ηs and λs, indicating that the this dataset is easy to classify and thus not overly sensitive to changes in hyperparameters. This is consistent with out findings in section V C.

Method	η [10^{η}]	λ [10^{λ}]	M
Vanilla test-set	-1.222	-6.0	15
Momentum test-set	-1.778	-6.0	5
Adam test-set	-1.778	-2.5	1

Table VII. Table with the best values from figures 11, 12 and 13

Using the tuned hyper-parameters we did 100 runs of logistic regression on the breast cancer dataset and calculated the mean accuracy for each optimizer method. The maximum number of training epochs was set to 1000. The results are presented in table VIII.

	Ordinary	Ridge
SGD method	*Mean accuracy* [%]	*Mean accuracy* [%]
Vanilla test-set	96.6	96.4
Momentum test-set	95.6	95.5
Adam test-set	95.5	97.0
Scikit Learn test-set	94.7	97.7

Table VIII. Test accuracies from logistic regression on the breast cancer data.

Using the Adam optimizer with regularization we get an accuracy of 97%, which is the best accuracy amongst the methods we have implemented. However, "vanilla" SGD without regularization also does well with an accuracy of 96.6%. While for linear regression, as seen in section V A, the "vanilla" SGD did significantly worse than the other optimizers, the same cannot be said in this case. Scikit-Learn's logistic regression with regularization does achieve a better result with 97.7% accuracy. Scikit-Learn's implementation uses a different optimizer, Limited-memory BFGS, which we have not tested and probably accounts for the discrepancy.

As expected we achieve poorer accuracies with logistic regression than with our neural network, as the later can learn more complex relations.

VI. CONCLUSION

We have implemented a feed forward neural network and tuned it to work on a regression dataset -a sampling of the Franke function-, and a classification dataset -the Wisconsin Breast Cancer data. In the classification problem we achieved an accuracy of 98.8%, while the performance was considerably worse in the regression problem, with the highest R2 score achieved being 0.86. Both of these results are better than the ones we achieved using more conventional methods. Using logistic classification on the breast cancer dataset gave an accuracy of 97.0%, while for our regression problem, OLS gave a considerably lower R2 score of 0.79.

While we, in our neural network, use SGD as an optimizer, we show that when using linear regression on our dataset, more advanced optimizers such as SGD with momentum or Adam preform better. This makes it likely that our network's results could be improved further by implementing one of these optimizers.

[1] L. Luque and K. Beshkov, "Fys-stk4155 project 1: Linear regression," pp. 1–2, 10 2021.

[2] M. Hjorth-Jensen, "Applied data analysis and machine learning, fys-stk3155/4155 at the university of oslo, norway," 2021.

[3] M. A. Nielsen, "Neural networks and deep learning," 2015.

[4] W. Commons, "Artificial neural network with layer coloring," 2013.

[5] R. Franke, "A critical comparison of some methods for interpolation of scattered data," tech. rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1979.

[6] W. H. Wolberg, W. N. Street, and O. L. Mangasarian, "Breast cancer wisconsin (diagnostic) data set," *UCI Machine Learning Repository [http://archive. ics. uci. edu/ml/]*, 1992.

[7] A. F. M. Agarap, "On breast cancer detection: An application of machine learning algorithms on the wisconsin diagnostic dataset," in *Proceedings of the 2nd International Conference on Machine Learning and Soft Computing*, ICMLSC '18, (New York, NY, USA), p. 5–9, Association for Computing Machinery, 2018.

[8] C. Nguyen, Y. Wang, and H.-N. Nguyen, "Random forest classifier combined with feature selection for breast cancer diagnosis and prognostic," *Journal of Biomedical Science and Engineering*, vol. 06, pp. 551–560, 01 2013.

APPENDIX

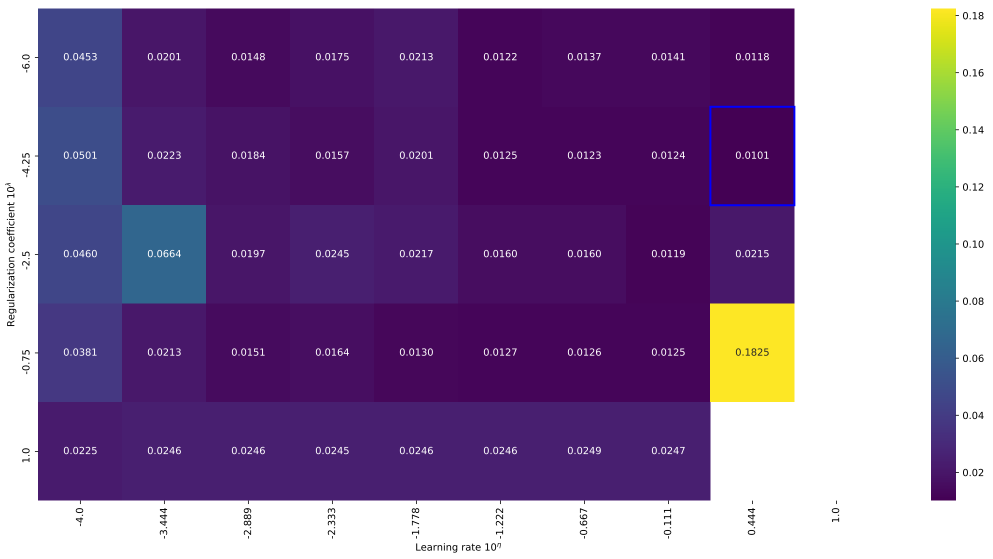

Figure 8. Vanilla heat map of η and λ values where $M = 20$. The figure shows how the MSE decreases as η increases and lowers as λ increases, where the best combination is framed in a blue rectangle. We see there are some white spots on the map, situated around $\eta = 10$. The white spots are due to invalid values (NaN or infinities) and are left out.

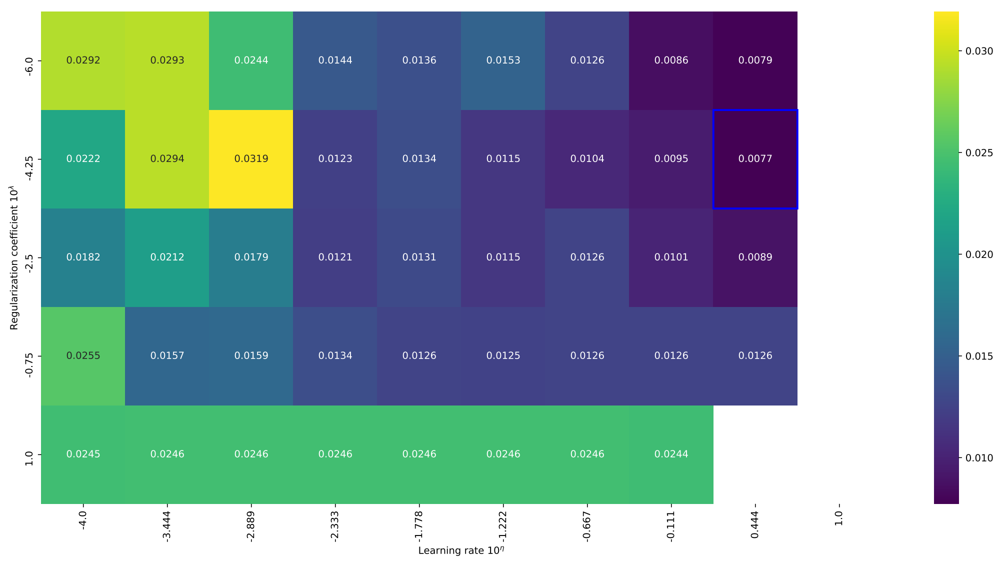

Figure 9. Momentum heat map of η and λ values where $M = 20$ and $\gamma = 0.80$. The figure shows how the MSE decreases as η increases and lowers as λ increases, where the best combination is framed in a blue rectangle. We see there are some white spots on the map, situated around $\eta = 10$. The white spots are due to invalid values (NaN or infinities) and are left out.

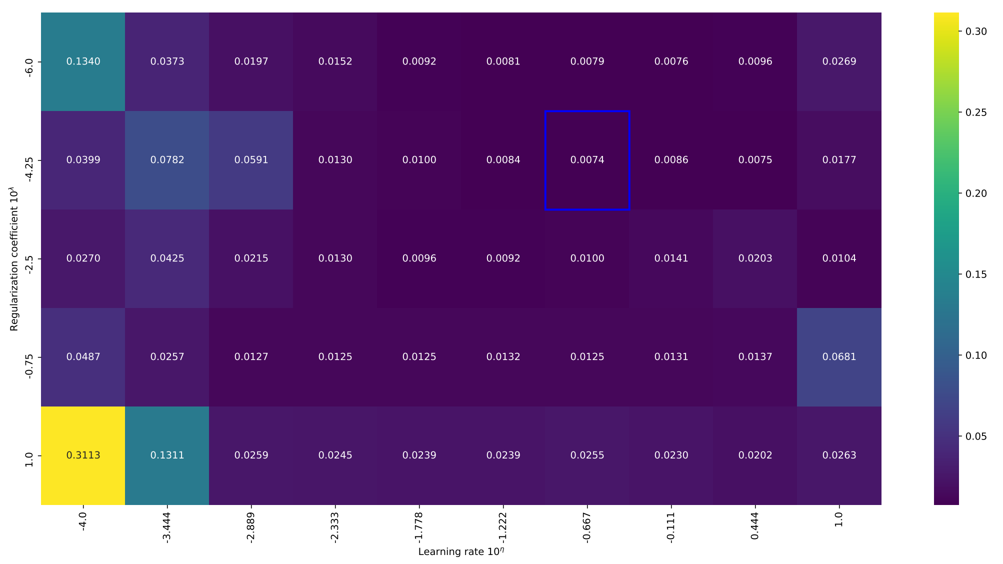

Figure 10. Adam heat map of η and λ values where $M = 20$, $\beta_1 = 0.9495$ and $\beta_2 = 0.9999$. The figure shows how the MSE decreases as η increases and lowers as λ increases, where the best combination is framed in a blue rectangle.

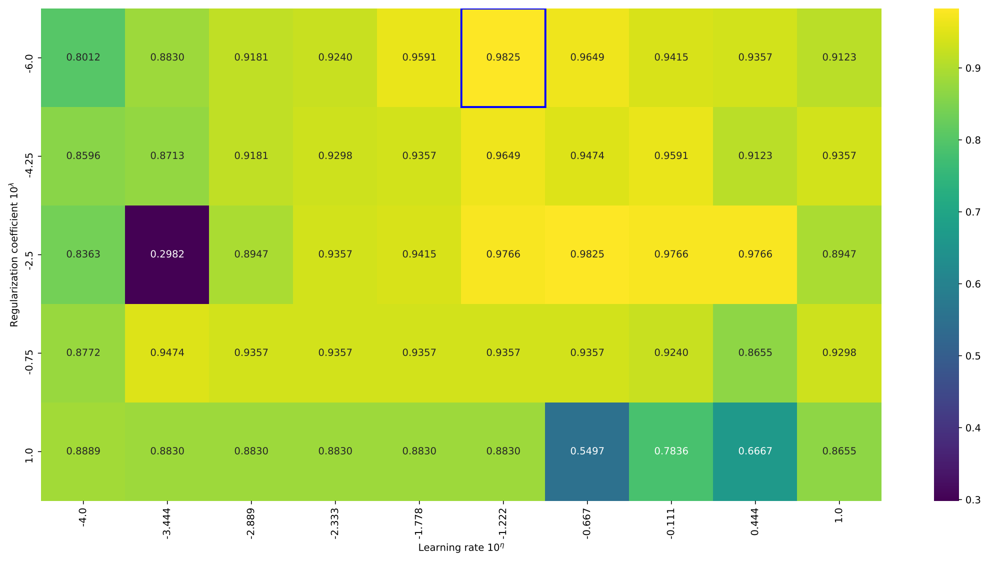

Figure 11. Heat map showing accuracies for different combinations of η and λ values with $M = 15$ and a vanilla SGD optimizer. The best combination is framed in a blue rectangle.

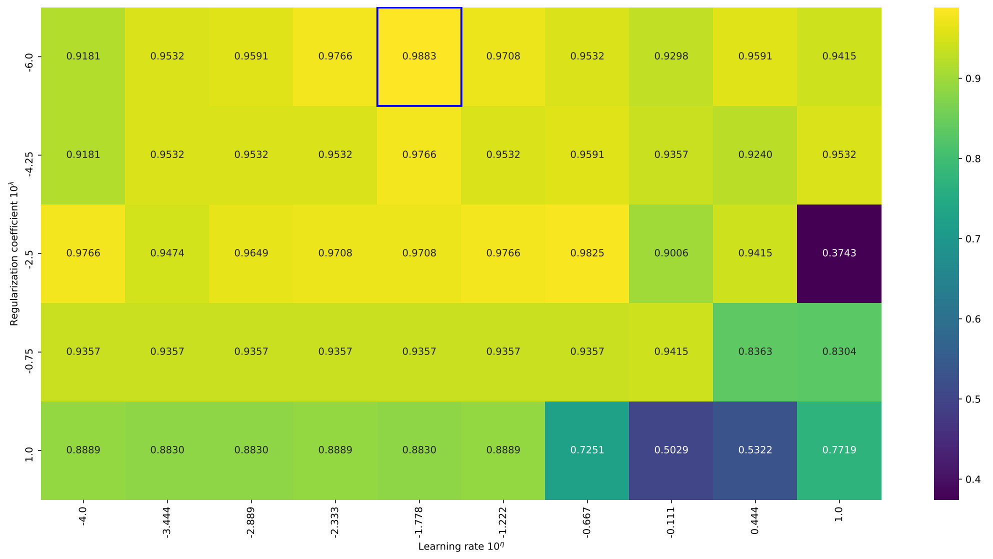

Figure 12. Heat map showing accuracies for different combinations of η and λ values with $M = 5$, $\gamma = 0.80$ and a SGD optimizer with momentum. The best combination is framed in a blue rectangle.

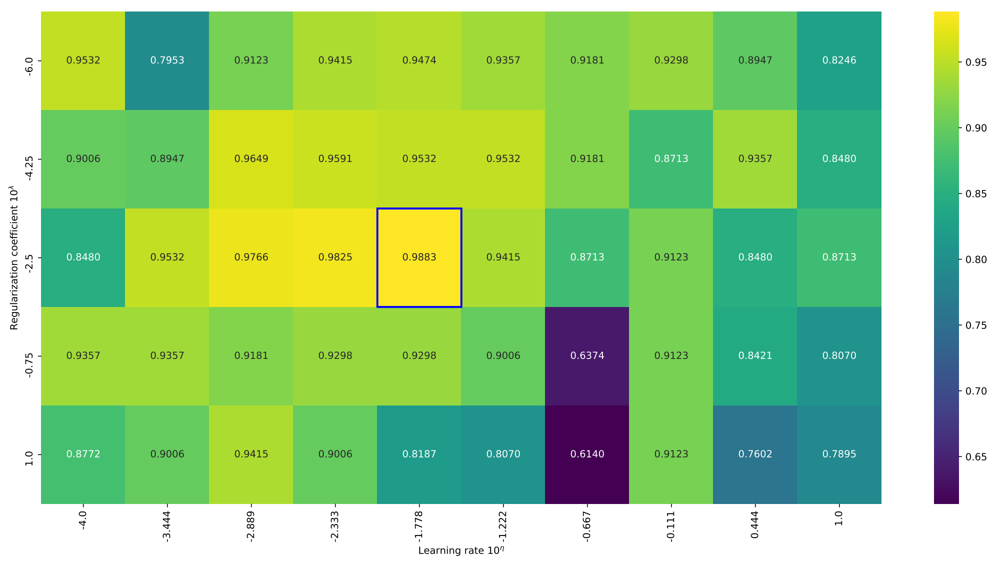

Figure 13. Heat map showing accuracies for different combinations of η and λ values with $M = 1$, $\beta_1 = 0.92475$ and $\beta_2 = 0.99495$ and the Adam optimizer. The best combination is framed in a blue rectangle.