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

Jørn Eirik Betten, Derya Aricigil, Behnoosh Ashrafi
(Dated: November 20, 2021)

We perform regression using SGD and MLP Regressor model on Franke function, and investigate the obtained result with linear regression. Then we proceed to implement MLP Classifier model as well as logistic regression on Wisconsin Breast Cancer data set. The results show that the MLP model and SGD does not appear to outperform linear and logisitic regression as expected.

I. INTRODUCTION

While trying to analyse data, we face many challenges: Data availability, time and space complexity of the model, noisy data set, etc. To have a good accuracy and results alongside a model with acceptable time and space complexity is the goal. Here, we try to use different machine learning methods for regression and classification on two data sets. For regression we will use the Franke function as a data set and apply different regression methods such as self implemented OLS, Ridge and SGD with different hyper-parameters. We then compare the results and with Scikit-learn's version. For the classification problem we use the Wisconsin Breast Cancer as our data set in logistic regression. We also build a multilayer perceptron (MLP) for both regression and classification by adding (for classification) an activation function in the last layer (output layer). We will describe each method and model, and discuss the results. Our codes can be found in our GitHub repository [1].

II. FORMALISM

Gradient descent

Gradient descent (GD)

Unless explicitly stated otherwise on mathematical formalism, all variables, such as θ and X, are vectors or matrices.
Given a dataset $\mathcal{D} = (X, z)$ of length N, where X is the dataset and z is the response variable, we define a model function $f(X, \theta)$ and a cost function $E(\theta)$ (or alternatively written $C(\theta)$). By calculating the gradient of the cost function with respect to θ and updating all variables in the path towards the steepest descent, we are able to optimize θ, and depending on the function f the minimum discovered must ideally be global since it is the minimum of the function f that will yield the best fit. The direction of the steepest descent is given by

$$g = -\nabla_\theta E(\theta), \tag{1}$$

where $\nabla_\theta E(\theta)$ is the gradient of the cost function with respect to θ. We are also defining a learning rate, η, typically chosen as a constant for all parameters θ, but can be chosen to be "time-dependent" or dependent on the number of iterations already run. We denote the gradient in iteration t as $\nabla_{\theta_t} E(\theta_t)$ and the iteration-dependent learning rate as η_t. In gradient descent we will use the magnitude of the learning rate as a hyper-parameter to calculate the optimal parameters $\hat{\theta}$ in an iteratively scheme written mathematically as

$$\begin{aligned} g_t &= -\eta_t \nabla_{\theta_t} E(\theta_t), \\ \theta_{t+1} &= \theta_t + g_t, \end{aligned} \tag{2}$$

which may converge to $\hat{\theta}$ as $t \to \infty$. This is called relaxed fixed point iteration within mathematics, and will be a successful updating scheme for any uniformly convex cost function in the parameter space, if η is small enough, and given infinite iterations. However, the cost function need not be uniformly convex, and there is the chance of ending up in a local minima. In the case of a non-uniformly convex cost function, the starting point of the iterations would be critical to whether we end up in a local or global minima. When choosing the learning rate, η, we need to consider the following: If the learning rate is too small, it will be computationally expensive, i.e. the number of iterations will be large. If on the other hand the learning rate is too large, we may end up in a situation where we move past the minima and almost as far away from the minima, on the other side of the "valley of the minima in parameter space". This will also end up being computationally expensive, as we do not get closer to the minima for every iteration. If the learning rate is too large, we might even increase the error for every iteration. In that case

$$\theta_{t+1} = \theta_t + g_t \tag{3}$$

no longer is a contraction in the parameter space, which means it will not converge. The choice of learning rate is therefore of dramatic importance, and must not be taken lightly. It is often decided after running a so-called grid search, where we try and run multiple different learning rates and compare their efficiency. Another disadvantage of GD is the computation of every data point, especially if the data size $\mathcal{D}$ is large, due to how computationally expensive it becomes.

Stochastic gradient descent (SGD)

Stochastic gradient descent (SGD) avois the aforementioned limitations of GD through introduction of stochasticity and by dividing the data into subsets, m mini-batches of size M, $m = N/M$. This is not only computationally more efficient as we only need to compute derivatives for every mini-batch (rather than every single data point) but it also randomizes the data to ensure the algorithm does not become trapped in a local minima. Otherwise, we update the θ in exactly the same way as in GD. To improve further on SGD, it is ideal to repeat the calculations over a set of randomly selected mini-batches. Calculation over a set of m mini-batches correspond to a single epoch, which can be performed many times and contributes to improved estimation of θ as the algorithm is allowed to pick different combinations of mini-batches at every epoch.

We use the same data points N and polynomial degree for the Franke function (with slight noise) that were used in estimating optimal values in OLS and Ridge Regression for comparison purposes [2] and this time the gradient of the OLS and Ridge cost functions will be applied. Additionally, parameters β will be denoted by θ. For every mini-batch m there is a corresponding gradient given in eq. 4-5 over M data points. The gradients are used to estimate θ as shown in eq. 2.

$$\nabla_\theta C(\theta) = \frac{2}{M} X^T (X\theta - \mathbf{y}), \tag{4}$$

$$\nabla_\theta C_{\text{ridge}}(\theta) = \frac{2}{M} X^T (X\theta - \mathbf{y}) + 2\lambda\theta \tag{5}$$

Stochastic gradient descent (SGD): Learning Rate

By letting the learning rate γ be a function of time t, $\gamma_j(t; t_0, t_1) = \frac{t_0}{t+t_1}$, the rate can be reduced with increasing number of epochs since $t = epoch \cdot m + i$, where i denotes the ith mini-batch m and $i = 0, ..., m$. The result is a smaller change in estimation of θ_{t+1} at higher t allowing better convergence since it is multiplied with the gradient and then subtracted from the previous θ_t. This ensures that for an adequately small γ that $C(\theta_{t+1}) \leq C(\theta_t)$, i.e. the gradient becomes smaller with every iteration.

Stochastic gradient descent (SGD) with momentum

SGD combined with a momentum term is one of the most widely used algorithms [3]. The momentum term indicates the direction in which the gradient moves in the parameter space, storing information of the gradient's direction along the function's landscape that is used to estimate the next θ; and thus, it enables more efficient convergence [3] by helping the algorithm converge faster along flatter curves and slower along steeper curves. The equations for the momentum section of the algorithm is shown in eq. 6. The extent of the momentum is controlled utilizing a momentum term α, with $0 \leq \alpha \leq 1$, and the method reduces to SGD if α is 0.

$$\begin{aligned} \theta_{t+1} &= \theta_t - (\alpha \cdot p_t + \gamma \cdot \nabla_\theta C(\theta)) \\ p_{t+1} &= \alpha \cdot \nabla_\theta C(\theta) + \gamma \cdot \nabla_\theta C(\theta) \end{aligned} \tag{6}$$

Regression and classification

In certain cases it is more useful to distribute a set of data into a group of K classes (hard classifier) or assign a probability value for the data to be in a particular class K, known as a *soft classifier*.

Logistic Regression

A soft classifier in a binary system is achieved with the help of a logistic function known as the *Sigmoid function* (eq. 7), yielding probability values between 0 and 1 and thereby generating a non-linear relationship between the inputs and outputs.

$$p(z) = \frac{1}{1 + \exp -z} = \frac{\exp z}{1 + \exp z} \tag{7}$$

Instead of using Mean Squared Error (MSE) as the cost function as in linear regression, Maximum Likelihood Estimation (MLE) is used for a logistic model and obtained by maximizing the product of all probabilities over the entire dataset $\mathcal{D}$ given parameters β over all N data points

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

The minimization of the cost function above can be written shortly in the following way, where y is given by the data and p contains the predicted probabilities, $p(y_i|x_i, \boldsymbol{\beta})$, and is referred to as the *cross entropy*.

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p}) . \tag{8}$$

SGD method will be implemented using the gradient of the cost function given above in eq. 8, where p is the sigmoid of randomly generated θ values in the first iteration and y_i are the data values. The regularization parameter of L_2 norm, λ, will also be added to the gradient as a differentiated form with respect to θ

$$\frac{\partial}{\partial \theta} \lambda \parallel \theta \parallel_2^2 = 2\lambda\theta \tag{9}$$

We will then proceed to calculate the predicted probabilities p using the Sigmoid function and divide them into binary outcomes.

Neural networks

A neural network is a model inspired by and imitates the way neurons send signals among themselves in the brain. Due to the potential complexity arising from its structure, it is regarded as a more powerful computational model in comparison to OLS, Ridge and logistic regression. Several types of neural networks exist and among them which we build in this project is a type of artificial neural network (ANN); feed forward neural network (FFNN), where all neurons are connected to one another making the network *fully connected*. Information passes only in one direction through three types of layers: input, hidden and output. FFNN is also known as a *multilayer perceptron* (MLP) due to its multiple layers, each layer able to contain different number of neurons. This is in contrast to a single perceptron model, which consists of only one layer.

Mathematical formalism

Let i denote the ith neuron, with $i = 1, ..., N_l$ for layer l, where w_{ij}^l represents the extent of interaction between a neuron i in layer l and every neuron j in the subsequent layer $l-1$ (eq. 10). f is a non-linear activation function; b is a weight parameter set to 0.01 to prevent a zero argument z, as this will be relevant for the back propagation algorithm. If $l = 1$, then y_j^0 is the input of data points from the design matrix X, x_j. Eq. 10 is easily implemented as a matrix-vector multiplication.

$$y_i^l = f(z_i^l) = f\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right), \tag{10}$$

where we denote the signal of activation of neuron i and layer l by y_i^l, the bias of that neuron as b_i^l, and the "weight matrix", w, here weighting the signals from the jth neuron of the previous layer with w_{ij}. For all the hidden layers the signal will be propagated forward until in this manner until we hit the output layer. Here there will be a special type of activation function that is suited to the kind of data we are trying to fit with our neural network. More on this in section II. This is the basic idea of the feed-forward algorithm.

Back propagation

The back propagation algorithm can be thought of as the core behind any neural network. The basic idea behind it is to update the weights of layers of neural network in a way such that error becomes smaller for the next iteration of the feed-forward algorithm. However, this can be quite difficult as the neural network can have multiple layers. What the back propagation algorithm takes advantage of, is a relationship between the errors of the layers l and $l-1$. To break it down, let us introduce some mathematical formalism. Let's denote the error at neuron i and layer l as ϵ_j^l. If we define some cost function C, we can define the error in layer L, the outer layer, as

$$\epsilon_i^L = \frac{\partial C}{\partial z_i^L}. \tag{11}$$

It is here that we take advantage of the relation between the errors in layers next to each other. We have that the error in layer $L-1$ is

$$\epsilon_i^{L-1} = f'(z_i^{L-1}) \sum_j^{N_L} \epsilon_j^{L-1} w_{ji}^L, \tag{12}$$

and so we have that the error in layer l is related to the matrix-multiplication of the error in layer $l+1$ and the transpose of the weights of that layer times the derivative of the activation function of layer l, generalized as

$$\epsilon_i^l = f'(z_i^l) \sum_j^{N_{l+1}} \epsilon_j^{l+1} w_{ji}^l. \tag{13}$$

Next, we have that the partial derivative of the cost function with respect to the bias can be written as

$$\frac{\partial C}{\partial b_i^l} = \epsilon_j^l, \tag{14}$$

and the partial derivative of the of the cost function with respect to the weights is written as

$$\frac{\partial C}{\partial w_i j^l} = \epsilon_i^l y_j^{l-1}. \tag{15}$$

All these mathematical facts are used in the back propagation algorithm, sketched in 1.

Activation functions

The activation function in the hidden layer warps the output of the layer of neurons in a specified way. The sigmoid function

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{16}$$

was a common choice for a long time, but the derivative becomes zero for large values of input, and we'll end up with a neural network that does not update the weights of the layers in the back propagation algorithm. This is famously called the vanishing gradient problem. Some other activation functions for the hidden layers, where

Algorithm 1 Back propagation algorithm

$a_L \leftarrow$ feedForward
procedure BACK PROPAGATE
$\epsilon_L \leftarrow a_L - \text{target}$ ▷ Error in output
$\nabla_{b_L} C \leftarrow \sum \epsilon^L$ ▷ Computing gradient of bias of output layer
$\nabla_{w_L} C = matmul(a_{L-1}^T \epsilon_L)$ ▷ Computing gradient of weights of output layer
$\epsilon_{L-1} \leftarrow matmul(\epsilon_L w_L) a'_{L-1}$ ▷ Computing error in layer $L-1$.
for $l = N-1, N-2, \ldots, 2$ **do**
$\nabla_{b_l} C \leftarrow \sum \epsilon_l$ ▷ Computing gradient of bias of layer l.
$\nabla_{w_l} C \leftarrow a_{l-1}^T \epsilon_l$ ▷ Computing gradient of weights at layer l.
$\epsilon_{l-1} \leftarrow matmul(\epsilon_l w_l) a'_{l-1}$ ▷ Computing the error in layer $l-1$.
$\nabla_{b_1} C \leftarrow \sum \epsilon_1$ ▷ Computing gradient of bias of layer l.
$\nabla_{w_1} C \leftarrow X^T \epsilon_1$ ▷ Computing gradient of weights at layer 1.
for $l = 1, 2, \ldots, N$ **do**
▷ Update weights and biases using gradients. Here we can have L_2 regularization parameters, learning rate is essential, momentum could be added.

this problem does not occur, are the rectified linear unit (ReLU), and the leaky rectified linear unit functions. These do not have the same issue with the derivative, and the vanishing gradient problem is not an issue for them. The rectified linear unit and leaky rectified linear unit functions,

$$\sigma(z) = \text{ReLU}(z) = \begin{cases} z & \text{if } z > 0, \\ 0 & \text{if } z \leq 0, \end{cases}$$

$$\sigma(z) = \text{LeakyReLU}(z) = \begin{cases} z & \text{if } z > 0, \\ \alpha & \text{if } z \leq 0, \end{cases}$$

returns input if the input is larger than 0, and return either 0 (ReLU) or a hyper-parameter α (LeakyReLU) if the input is negative or 0. The hyper-parameter α is typically set to a low positive number, like 0.01.
In the output layer we use an activation function aswell, depending on what kind of dataset we are trying to fit with our neural network. If we try to perform regression analysis on a continuous function, it is natural to just use a linear function that returns the input from the output layer. For classification problems we need to find the probability for the different categories we have as possible targets. There are two main types of these activation functions, either a function that returns one category with a certainty, called a hard classifier, and an activation function that returns the probability for that category, called a soft classifier. For a binary case, we have the hard classifier, which we call "binary classifier", which yields the vector $[0, 1]$ or $[1, 0]$ depending on whether the value for the second index or first index are the largest. There is also the soft classifier, the softmax function, which will yield the probability for a given category when there are multiple categories, and is defined as

$$p(z_k) = \text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{i=1}^{n_{categories}} e^{z_i}} \tag{17}$$

MLP Regressor vs MLP Classifier

For an MLP Regressor model the output will equal the number of inputs, i.e. N data points, and no activation function is implemented to transform the output from the last hidden layer such that the results are handed from the neural network directly. This is in contrast to an MLP Classifier model, which uses an a non-linear activation function for the output to calculate probabilities using Eq. 17 or classify these calculated probabilities into binary outcomes if number of classes $K = 2$.

Because probabilities are calculated for each class, we utilize a *one hot vector* for every data point. An example of such a vector as an input to the model is Eq. 18 for $K = 5$ classes and classification of y into class $K = 3$. The output, if we are only using the softmax function, will therefore consist of output vectors filled with the probability distribution for the classes; each class will have its own probability value.

$$y = 3 \quad \rightarrow \quad \boldsymbol{y} = (0, 0, 0, 1, 0) \tag{18}$$

Performance

To measure the performance of our models we will utilize MSE, R_2 and accuracy score. MSE (Eq. 19) measures the average square of the deviation between N data points and predicted data points. A small MSE score is therefore indicative of a good fit.

Similarly, R^2(Eq. 20) has a fraction that takes the square of the deviation between data and predicted data and is divided by the square of the difference between data and mean of all predicted data points. The smaller the deviations in the squares, the less subtraction from 1 resulting in a higher R^2 score. Accuracy score (Eq. 21) is used for classification models, for which the indicator function I is 0 or 1 for $\tilde{y}_i \neq y_i$ and $\tilde{y}_i = y_i$, respectively, and represents the fraction of the predicted data that matches the data set.

$$MSE(z,\tilde{f}) = \frac{1}{N}\sum_{i=0}^{N-1}(z_i - \tilde{f}_i)^2 \tag{19}$$

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1}(z_i - \tilde{f}_i)^2}{\sum_{i=0}^{N-1}(z_i - \bar{z})^2} \tag{20}$$

$$Accuracy = \frac{\sum_{i=1}^{n} I(\tilde{y}_i = y_i)}{n} \tag{21}$$

DATASETS

Franke function

The data set we perform our regression analysis on is generated by the Franke function with normally distributed randomly added noise with variance 0.2. We genereate a 40 by 40 meshgrid of the Franke function in the area of $x \in [0,1]$ and $y \in [0,1]$, and thus we have 1600 datapoints, which we split into testing and training sets. In [2] we found that the optimal polynomial degree for reproducing the dataset using linear regression was a degree of 20. The design matrix will therefore have 231 features. We set the size of the training and testing set to respectively, 80 percent and 20 percent of total data size.

Wisconsin breast cancer data

The data set we perform classification analysis on is the Wisconsin breast cancer data set. There are 455 data points with 30 features in this set, and the target of our analysis is whether the tumor is malignent or benign. The target is therefore split into two categories, (benign and malignent), and we use the one-hot approach to split it.

RESULTS AND DISCUSSION

Stochastic Gradient Descent

For OLS the optimal R_2 score is found for $M = 15$ and 150 epochs at $\eta = 0.2$, as shown in Fig. 1, though the MSE does not completely agree due its lowest value at $M = 10$ and the same number epochs. We also check the optimal batch size and epochs for Ridge (Fig. 2), finding that $M = 20$ with 150 epochs are optimal yielding R_2 and MSE is the same as for OLS, at $M = 10$.

At this point it becomes clear that larger epochs corresponds to better values, which is confirmed by Fig. 3 with $R_2 = 0.62$ at the same number of epochs at $\lambda = 0$ and $\eta = 0.5$. However, as we will see, a η value of 0.2 will yield better results, and this is due to our choise of learning rate interval fed to the grid search as the estimated scores are highly sensitive to η. A similiar event occurs with mini-batch size, $M = 15$ yields better values than $M = 20$.

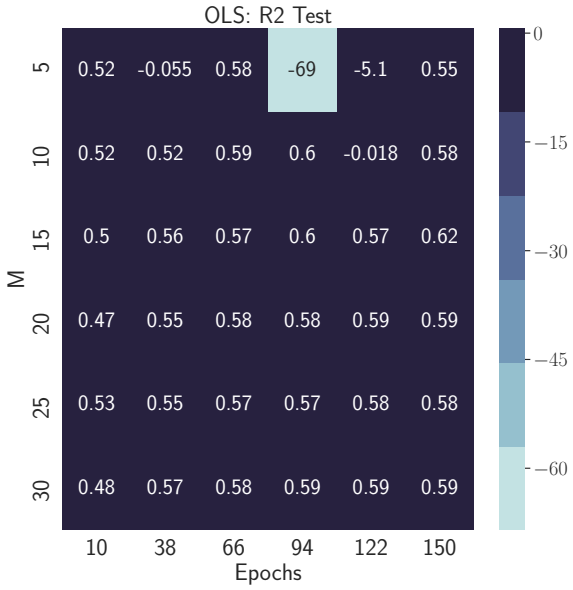

FIG. 1. SGD with no momentum for OLS shown using R_2 score for varying batch size M and number of epochs. The best R_2 score is 0.62.

SGD vs Linear Regression

An optimal λ value of 0 indicates that OLS fits the data set better than Ridge, though there is still a slight difference in the results, Ridge is equivalent to OLS at $\lambda = 0$.

One important thing to note, for this project and the previous one [2], is the dramatic effect a noise of 0.2 has on the estimation of MSE and R_2. Once this noise is minimized to, ex. 0.01, the parameters significantly improve. A noise of value 0.2 shows that no momentum yields the best results, which is contradicting the expectations of introducing the momentum, and yet, when the noise is decreased, the effect is reversed: the momentum term starts to have a positive influence on the process of optimizing the parameters.

One way to assess the effects of the momentum term is plotting it against MSE test values. An optimal parameter with noise 0.2 with the help of the minimum is obtained, and upon implementing this new momentum term at 0.6204, the estimated MSE and R_2 yield nearly the same values as without momentum, and these are all

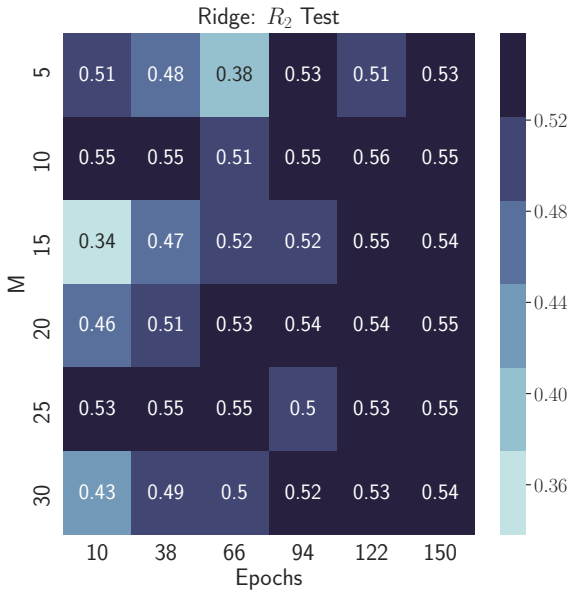

FIG. 2. SGD with no momentum for Ridge shown using R_2 score for varying batch size M and number of epochs. The best R_2 score is 0.56.

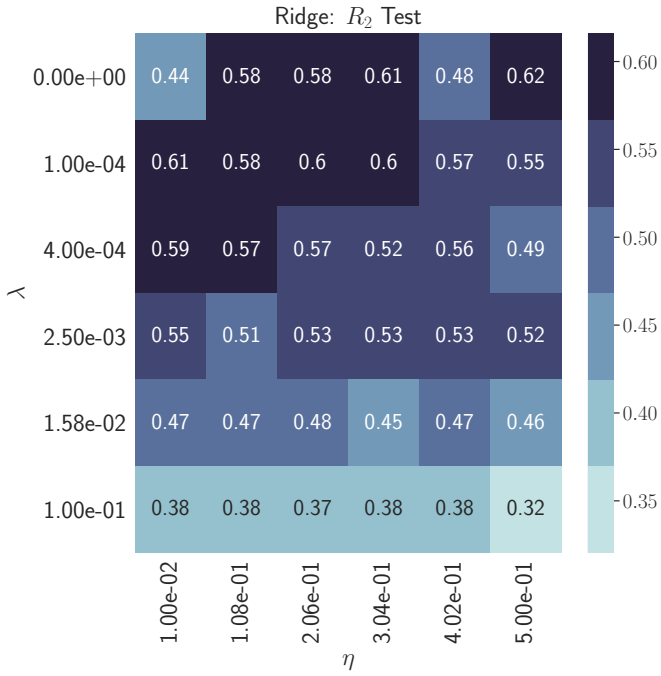

FIG. 3. SGD with no momentum for Ridge shown using R_2 score for varying hyper-parameters λ and η. The best R_2 score is 0.62.

slightly above the ones estimated using linear regression for OLS. All results are shown in Table I. It is also evident from the table that linear regression obtains better values or R^2 and MSE than SGD, but this is due to the λ set to 0 in SGD since, as mentioned previously, it yields the optimal results with grid search. For this reason, we will focus on OLS only for further discussion.

Our results from Table I indicates that linear regression is quite close to SGD. Normally we would expect linear regression to be outperformed by SGD due to its tendency of lacking adequate generalization of the data; while SGD randomizes different parts of the data set to avoid this generalization issue. In our case however, this similarity may be due to the particular data presented and its size. Perhaps, alternatively, implementing a significantly larger epoch than 150, if overflow can be avoided, may yield better results; but this is not computationally efficient.

Logistic Regression

A momentum term of zero and above yield similiar MSE and accuracy scores, and as previously discussed for SGD we assume the cause is due to the large noise. According to calculations done for Fig. 4, the optimal momentum at 0 is therefore expected. We also with these calculations the optimal η at 0.8702.

Using the optimal values for momentum and η we proceed to perform grid search. First, we find the optimal mini-batch size and epoch using an arbitrary $\lambda = 0.0531$,

	OLS	Ridge	Scikit OLS	Scikit Ridge
SGD (momentum)	0.57	0.57	0.51	0.54
	0.0549	0.0551	0.0631	0.0593
SGD	0.58	0.58	0.51	0.54
	0.0540	0.0540	0.0631	0.0593
Linear Reg.	0.56	0.62	-0.71	0.62
	0.0558	0.0491	0.2186	0.0491

TABLE I. R^2 (above) and MSE test (below) scores from SGD without momentum and momentum of 0.62 calculated with optimal parameters obtained after grid search: $\lambda = 0, \gamma = 0.2$, $M = 15$ and epochs=150, and results from Scikit's SDGRegressor and linear regression.

producing the values given in Table. II.

Accuracy	MSE	R^2	M	Epochs
0.94	0.061	0.74	30	80
0.94	0.061	0.74	6	20
0.94	0.061	0.74	12	100

TABLE II. Optimal mini-batch size M and epoch obtained through grid search.

We proceed to choose $M = 6$ and $epoch = 20$ over other combinations in Table II for the following reasons: 1. Smaller batch size is more ideal than large ones, especially considering our data set has $N = 40$ points, such that $M = 6 > M = 30$; 2. Less epochs translates to less

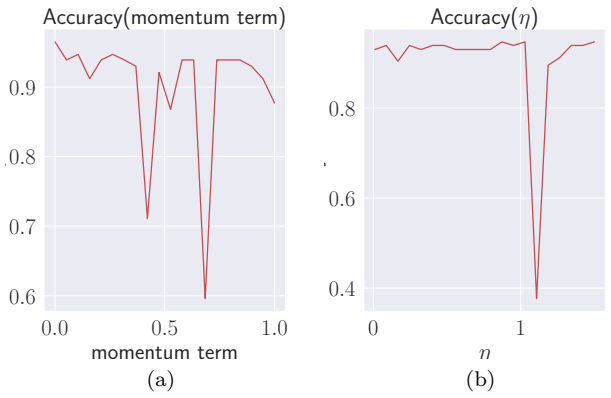

FIG. 4. Accuracy score as a function of (a) momentum term (b) learning rate η

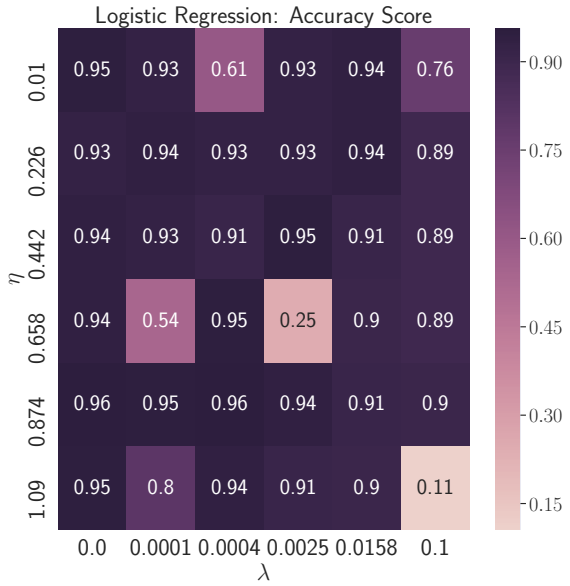

FIG. 5. Grid search for optimal λ and η using one of the possible optimal mini-batch sizes and epochs: $M = 6$ and $Epochs = 20$.

computational expenses, $Epochs = 20 > Epochs = 100$, for the same accuracy score.

A grid search for optimal λ and η is performed with $M = 6$ and $epoch = 20$, yielding $MSE = 0.044$ and $R^2 = 0.81$ with an accuracy score of 0.96, as shown in Fig. 5. The optimal parameters are then determined to be $\lambda = 0.004$ and $\eta = 0.874$.

Table III shows the results obtained with optimal parameters ($M = 6$, $epoch = 20$, $\lambda = 0.004$ and $\eta = 0.874$) using logistic regression without momentum. SGD method surpasses SciKit very slightly with a score of 0.96 and a lower MSE of 0.0439.

	MSE	Accuracy Score
SGD	0.0439	0.96
LogReg SicKit	0.0526	0.95

TABLE III. MSE and accuracy score obtained for logistic regression method with SGD using optimal parameters and momentum set to zero as well as SciKit results.

Feed forward Neural Network

Deciding on activation function in neural network for regression analysis

We ran a grid search for different activation functions against a range of learning rates, η, and with other parameters: number of epochs: 50, mini-batch size, $M = 10$, regularization parameter $\lambda = 0.0001$, momentum, $\gamma = 0.0$, and with one layer of 100 neurons. We observe in figures 7 and 6 that the mean squared errors (MSE) generally lower for the instance where we use the sigmoid function as the activation function, and that the R^2-scores is generally a bit higher for the case of rectified linear unit activation function. We acknowledge that the MSE is lower for the sigmoid, and we use that for our regression analysis on the Franke function.

Regression analysis on the Franke function

We put the "model" parameter in Neural Network to "Regressor" and remove the activation function in the last layer by assigning the "last-act-function" to "return input". Our output layer only has one neuron which gives the predicted value. To find the best architecture for our Neural Network we used grid search for different values of each of our hyper parameters and compared them. To achieve this we used constant values for every hyper parameter except the ones we were studying at any given point. After finding the optimum value for them we applied them for the rest of the grid searches. First we used grid search for η and λ.

As is apparent in the figures 8 and 9 the best MSE (0.044) and R^2 (0.5) scores happen in the same grid, for $\lambda = 10^{-3}$ and $\eta = 10^{-1}$. Another observation in these plots is that for both MSE and R^2 for $\eta \geq 10^0$ the results suddenly get really bad. This shows the overflowing, meaning that the η value is too large now and is leading us out and away from the optimal value instead of towards it. We tried many different parameters and we observed that in almost all of them these values for λ and η are the best. Even in those that they were not the optimum value they yield very good results. Based on this we decided to use these values for η and λ for the rest of the analysis.

In the next step we tried to find the optimum values for number of layers and number of neurons in the layers. We used the best values for η and λ from previous grid

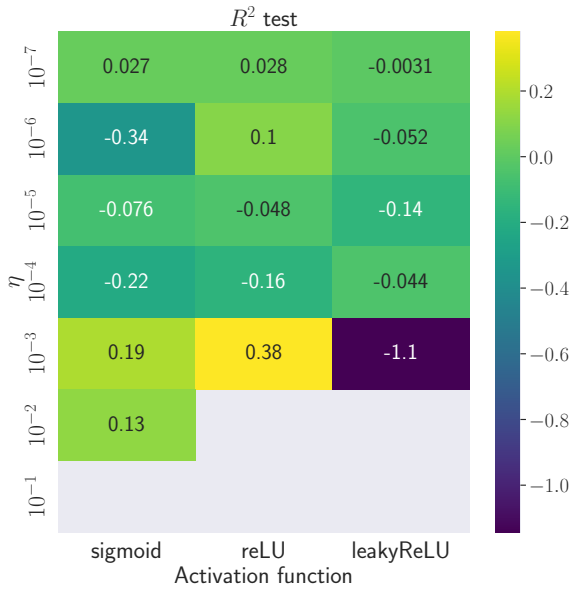

FIG. 6. R^2-scores in a heatmap where activation function used in the hidden layer is along the x-axis, and learning rate, η, is along the y-axis. Other hyperparameters: regularization, $\lambda = 0.0001$, momentum, $\gamma = 0.0$, number of epochs is 50, mini-batch size, $M = 10$, and we had one hidden layer containing 100 neurons. Where there are no values in the heatmap, we experienced overflow, and the regression yielded a NaN-value. The dataset the fit is performed on is 1600 data points of the Franke function with normally distributed random noise of variance 0.2 added.

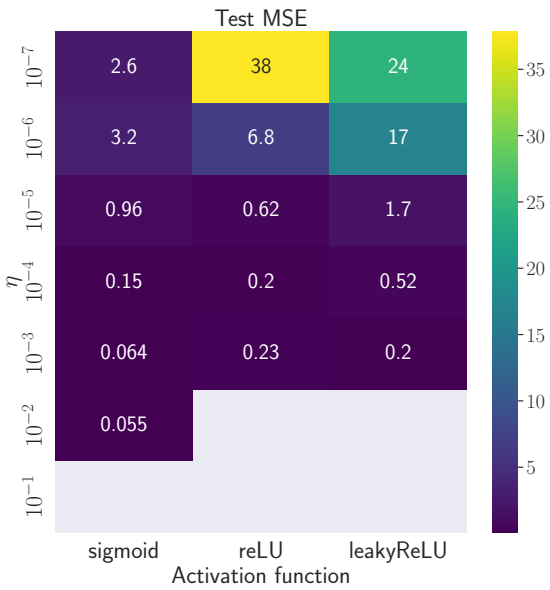

FIG. 7. Mean squared errors in a heatmap where activation function used in the hidden layer is along the x-axis, and learning rate, η, is along the y-axis. Other hyperparameters: regularization, $\lambda = 0.0001$, momentum, $\gamma = 0.0$, number of epochs is 50, mini-batch size, $M = 10$, and we had one hidden layer containing 100 neurons. Where there are no values in the heatmap, we experienced overflow, and the regression yielded a NaN-value. Along with figure 6

search.

Figures 10 and 11 shows us the best number of layers are 7 with 100 neurons in each layer. We added this information to the next grid search for number of batches and batch sizes.

As we can see in figures 12 and 13, the best results for R^2 (0.56) yield when batch size is 1 and the number of epochs is 20. This means that the best result happen when we have only one data for each round which is not sensible. This is why we looked at the other close to maximum values we got for R^2 (0.52, 0.53, 0.54 for batch sizes 2, 5 and 10 respectively). We tried different batch sizes corresponding to these scores to examine different gamma values in [0,1). We take 10 gamma values to check. gamma values = 0, 0.1, 0.2, ..., 0.9. The best values are $R^2 = 0.44$ and MSE = 0.044 for batch size = 10, number of epochs = 20 and gamma = 0.6. All these hyper parameters are listed in table IV.

The best R^2 result we get is 0.56 which is equal to what we get in project number 1 [2] with linear regression. It is expected for neural network to yield better results as it removes the linearity and gives the model more flexibility to fit the function. This could be because of our data set. It could be because our data set is better fit with a

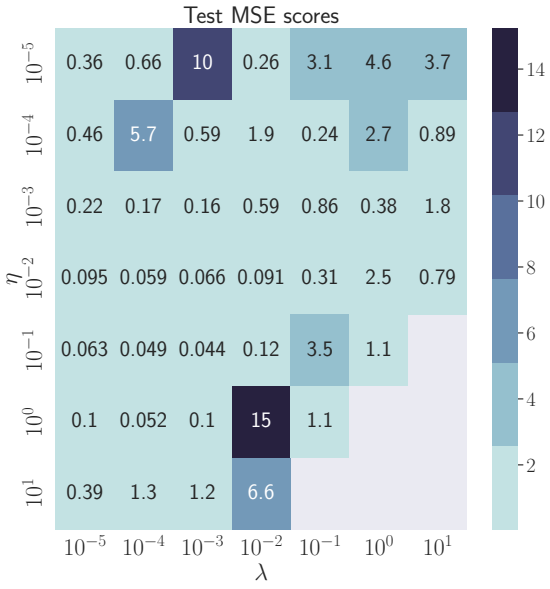

FIG. 8. MSE grid search for η and λ

FIG. 9. R^2 grid search for η and λ

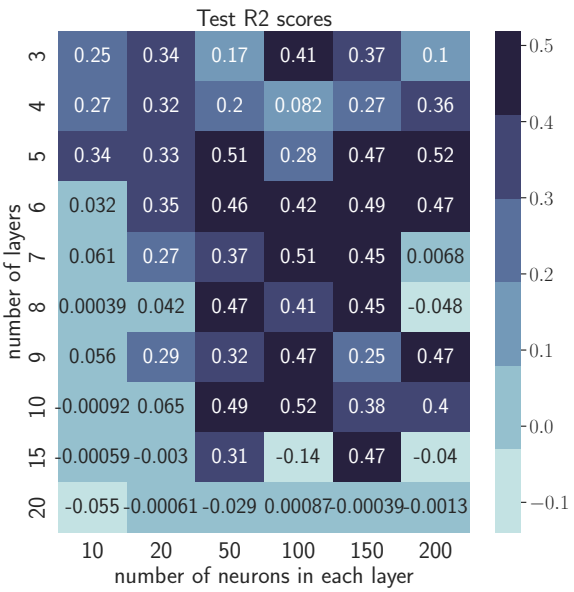

FIG. 11. R^2 grid search for number of layers and number of neurons

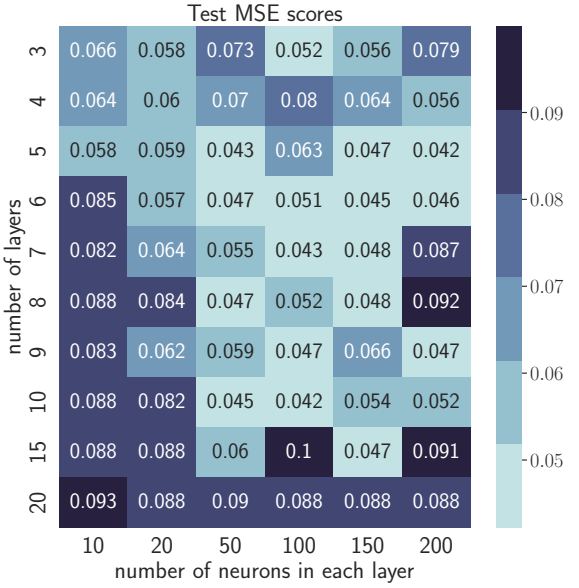

FIG. 10. MSE grid search for number of layers and number of neurons

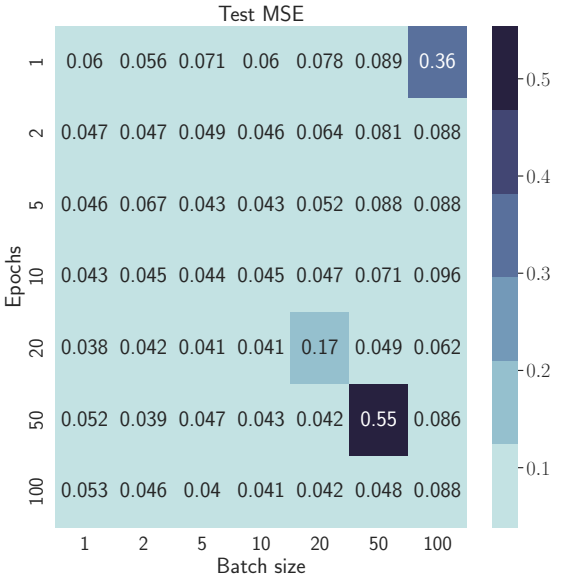

FIG. 12. MSE grid search for batch sizes and number of epochs

linear function. This shows us how much depended data analysis and machine learning is to the data set.

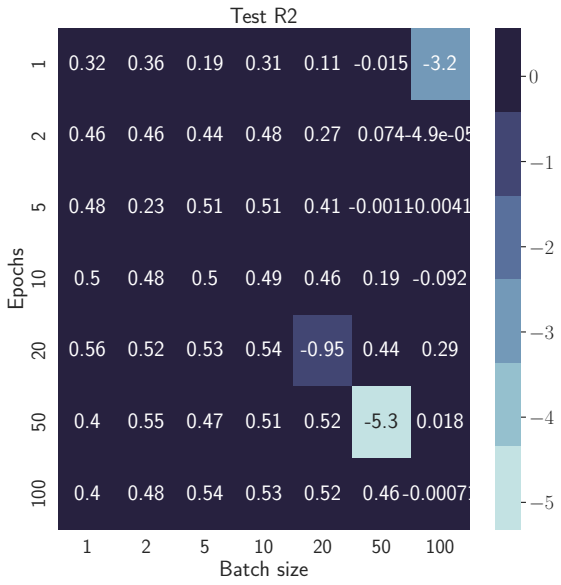

FIG. 13. R^2 grid search for batch sizes and number of epochs

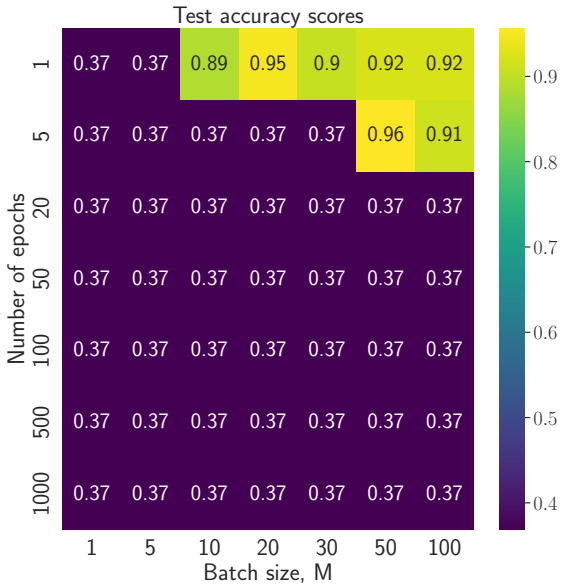

FIG. 14. The last grid search for optimal parameters

Optimal hyperparameters regression	Value
Optimal learning rate, $\hat{\eta}$	0.01
Optimal regularization parameter, $\hat{\lambda}$	0.0001
Optimal number of hidden layers	7
Optimal number of neurons	100
Optimal number of epochs	20
Optimal size of mini-batch, $\hat{M}$	10
Optimal gamma rate	0.6

TABLE IV. Optimal hyperparameters found via grid searches on the regression on the Franky function data. The best result we get is R^2 score of 0.56.

Optimal hyperparameters classifier	Value
Optimal learning rate, $\hat{\eta}$	0.1
Optimal regularization parameter, $\hat{\lambda}$	1
Optimal number of hidden layers	1
Optimal number of neurons	20
Optimal number of epochs	5
Optimal size of mini-batch, $\hat{M}$	50

TABLE V. Optimal hyperparameters found via grid searches on the classification on the Wisconsin breast cancer data. When these are applied we get an accuracy score of 0.96, which means that 96 per cent of the predicted values matches the testing part of the dataset.

Classification analysis on the Wisconsin Breast Cancer data using a neural network.

In figure 14 we see our final grid search for the optimal parameters of the feed forward neural network. After adjusting all parameters we get an accuracy test score of 0.96, which means 96 per cent "hit"-rate. The optimal parameters for the feed-forward neural network are shown in table V. The heat maps covering the grid searches can be found in III. Notice also from figure 14 that the classification fail when the number of epochs gets too large. An accuracy score of 0.37 can be compared with yielded values predict the same category for all input, and thus fails. Why it happens, we am unsure.

We also used the scikit learn classifier to compare our results to. We used the same optimal parameters in table V to train the model. The result can be found in table VI.

Scikit learn vs own classifier	scores
Scikit learn score train	0.91
Scikit learn score test	0.92
Own classifier train	0.92
Own classifier test	0.94
Own classifier MSE	0.052

TABLE VI. Comparing scikit learn MLP classifier with self implemented classifier using the breast cancer data.

Comparison of classification analysis.

SGD and our implementation of a neural network performed both very well with the Wisconsin breast cancer data, with high accuracy of 96 percent, and in conclusion they are both good candidates for performing classification analysis.

Comparison of regression analysis.

We used three main methods for regression analysis; linear regression, SGD and an implementation of a neural network. It is expected that the neural network implementation should perform better than the other methods, but our implementation did not. They all performed on the same level, with the implementation of Ridge linear regression having the best R^2 score of 0.62. In conclusion, none of regression analysis methods succeeding in giving a good approximation for the data set with added noise of a substantial size.

Conclusion

We performed regression analysis on a data set made by using the Franke function with normally distributed random noise of variance 0.2. In our previous project [2] we performed linear regression on the same data set (and with the same noise) and obtained $R^2_{OLS} = 0.56$ and $R^2_{Ridge} = 0.62$. After performing SGD, we find that the maximum R^2 score is 0.58 for OLS and Ridge (with $\lambda = 0$ as the optimal hyper-parameter) without momentum, and with momentum implemented the score drops to 0.57.

When implementing a neural network for regression on the same data set, we get an optimal R^2 score of 0.56. These methods yield very similar scores, with Ridge linear regression outperforming the others by a small margin. It is expected that the neural network implementation should perform better, though that is not our result.

For our classification analysis we used the Wisconsin breast cancer data set. The maximum achieveable accuracy score is 0.96, slightly higher than using Scikit. When using a neural network for the same task, we also get a maximum accuracy score of 0.96. The two implementations are yielding very similar results.

III. APPENDIX

Data backing up results on classification of the Wisconsin Breast Cancer dataset

Grid searching

Figures 15 and 16 shows our initial grid search for optimal η and λ. As you can observe, the optimal hyper-parameters: optimal learning rate, $\hat{\eta} = 0.01$ and optimal regularization parameter, $\hat{\lambda} = 10^{-5}$. We used these parameters for our next grid search, in number of neurons and number of layers-space, shown in figures 17 and 18 where the accuracy scores for training and testing sets are outlined. Figure 19 shows the comparative plot for the training set to figure 14.

H

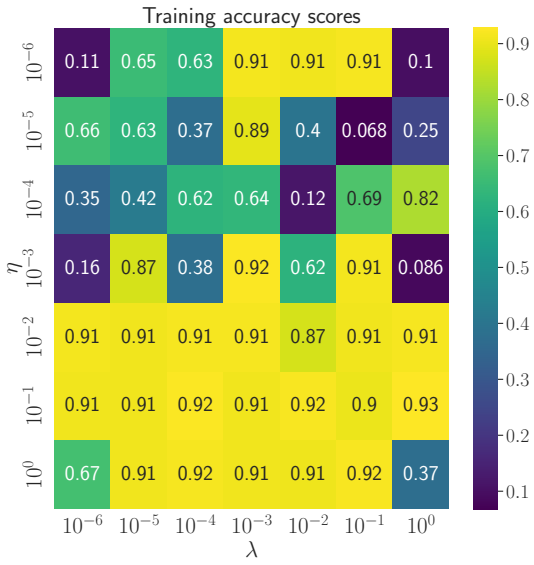

FIG. 16. The figure shows the accuracy scores on the training set on the grid of values of η and λ of the same ranges as in figure 15. The training values are high in same areas as in the comparative test case, (figure 15). The data set the fit is performed on is the Wisconsin breast cancer data set.

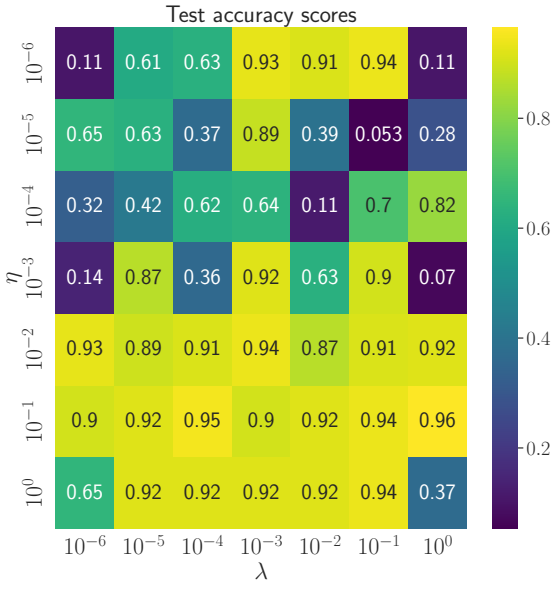

FIG. 15. The figure shows a grid search of six values of η and λ in the range 10^{-5} to 10, where we have set all the other parameters to some hopefully reasonable numbers. We observe that the best value is when $\eta = 0.1$ and $\lambda = 1$. The data set the fit is performed on is the Wisconsin breast cancer data set.

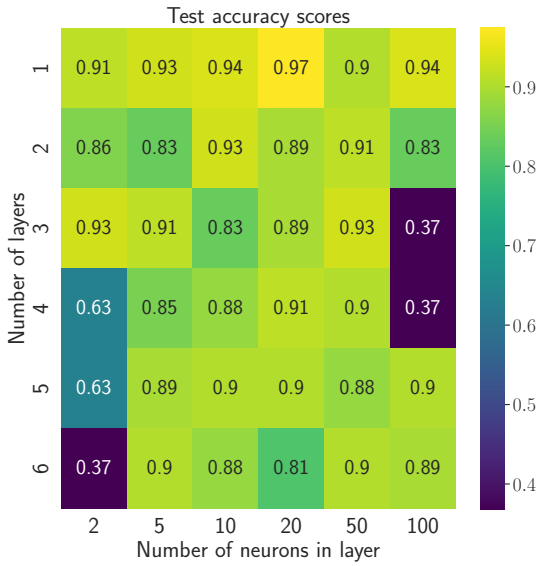

FIG. 17. The figure shows a grid search for optimal number of layers and number of neurons with optimal parameters $\hat{\eta} = 0.01$ and $\hat{\lambda} = 10^{-5}$, and the results are for the accuracy score on the test set. We observe that the optimal parameters for the number of layers and number of neurons in these layers are five layers with 100 neurons in each of them. The data set the fit is performed on is the Wisconsin breast cancer data set.

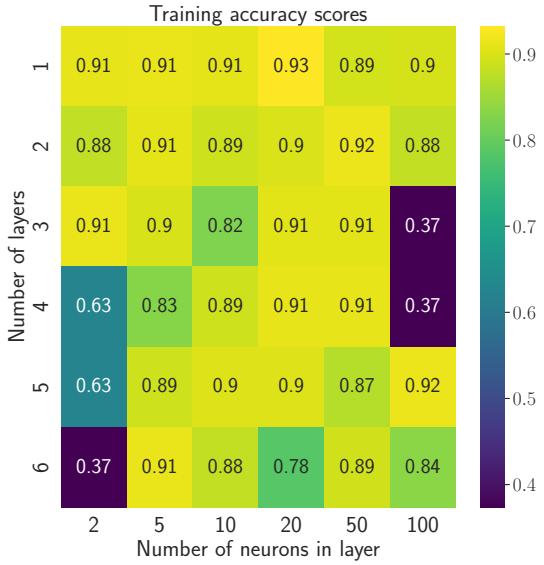

FIG. 18. The figure shows the comparative figure to figure **??** where we have now done the accuracy score on the training set. The values for the accuracy score on the training set are high in the same areas as for the test set in figure 17. The data set the fit is performed on is the Wisconsin breast cancer data set.

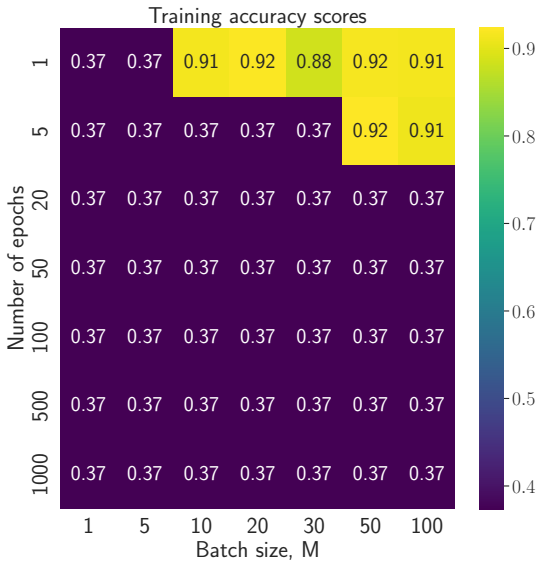

FIG. 19. The figure shows the values of the grid search for optimal parameters of numbers of epochs and batch size, M, and we measure it by using the accuracy score on the training set. We see that the we fail to predict the dataset when we the number of epochs are too large. We find that the optimal parameters in this space is $M = 50$ and the number of epochs is equal to 5. The data set the fit is performed on is the Wisconsin breast cancer data set.

[1] Jørn Eirik Betten, Derya Aricigil, and Behnoosh Ashrafi. Project 2 github repository, 2021. URL https://github.com/deryaaricigil/FYS-STK4155-Project-2.

[2] Jørn Eirik Betten, Derya Aricigil, and Behnoosh Ashrafi. Project 1, 2021. URL https://github.com/deryaaricigil/FYS-STK4155-Project-1/blob/main/PROJECT1FYS-STK4155-2236.ipynb.

[3] Vitaly Bushaev. Stochastic gradient descent with momentum. 2017. URL https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d.