Classification and Regression, from linear and logistic regression to neural networks
Project 2 FYS-STK 3155

Liang Jia and Emil B. Hågan
20.11.2021

Contents

Abstract

As technology developed and our perception of reality and the universe changed our methods of describing the world changed with it. With the new age and the development of the computer we changed how we do calculations to a numerical perspective. With this tool we could reach new heights. One of these tools are *Neural Networks* (NN). With this method we can analyze big data sets, transcribe speech to text, filter noise from a recording. The aim of this article was to study regression and classification problems where we develop our own Feed-Forward Neural Network (FFNN). Where we first looked at Stochastic Gradient Descent with added momentum. Then creating our own FFNN, where we looked at different activation functions. Further we implement a classification analysis where we change our cost function to calculate an accuracy score for. For the final part we looked at logistic regression code and compare this method with our FFNN. The aim is to see how well our FFNN preforms and the capabilities of a NN. The data set we studied was the Wisconsin cancer data from pythons package *sci-kit learn*. The advantages of the Sigmoid function is that it preforms well in its bounds and overall not terrible outside this area, the same can be said for the ReLU and the linear functions. Although the linear function might lose some of the data and miss some of the information. The worse functions where the Leaky ReLU and tanh, which preformed so bad for this NN that we should not use them for this application. By testing different activation function in the output layer we also saw that the best preforming function was the softmax, where it gave consistent results. The softmax is normally used for where we have an output in multi-class classification problems.The implementation of logistic regression on the same data got quite similar performance, with right hyper parameters. They both can reach about 97% accuracy on test data and more than 99% on train data. The best activation functions for classification cases will be the softmax and sigmoid and for regression the linear would work best.

1 Introduction

We have always wanted answers to the big questions, as technology developed and our perception of reality and the universe changed our methods of describing the world changed with it. With the new age and the development of the computer we changed how we do calculations to a numerical perspective. With this tool we could reach new heights and one of these tools are *Neural Networks* (NN). With this method we can analyze big data sets, transcribe speech to text, filter noise from a recording as Nvidia is doing with "NVIDIA RTX Voice" and something you might not think of at first, but they way NETFLIX recommends entertainment to a specific viewer.

The aim of this article is to study regression and classification problems where we develop our own Feed-Forward Neural Network (FFNN). This might seem like a big task to start with so we will first look at Stochastic Gradient Descent (SGD) with added momentum. Then we will look at creating our own FFNN where we first create an implementation for a simple case and then generalize it further. With our FFNN we will also look at different activation functions such as Sigmoid, RELU and leaky RELU. Further we will implement a classification analysis where we change our cost function to calculate an accuracy score for *TRUE* or *FALSE*. For the final part we will look at logistic regression code and compare this method with our FFNN. The aim is to see how well our FFNN preforms and the capabilities of a Neural Network. The data set we studied was the Wisconsin cancer data from pythons package *sci-kit learn*, which is already well structured and ready for the implementation.

The code for the different parts can be found at our GitHub

2 Theory

2.1 Stochastic Gradient Descent

In this first section we will look at the SGD which derives from Gradient Descent (GD), but addresses some of the issues with GD. Some of these issues with the GD is that with this method we locate the local minima of our function, now the problem here is that we might not know if this minima is the one we want or another local minima within the function we study. When we for example look at terrain data we might get some issues with this. The next issue we will face with the GD is that the initial conditions matter heavily on the result. This problem is because with different initial conditions we might end up at another local minima and GD is also very sensitive to the learning rate, if this learning rate is small the training will take longer and will have need of more computational power. Therefore this heavily influence the computational time of the method and GD is computationally expensive to calculate for data sets which are large.

However SGD addresses these issues and is therefore much better. The SGD comes from studying the cost function and minimize it, the function can be expressed as a sum over n data points of $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta). \tag{1}$$

To compute the SGD we can define it as a sum over i gradients as such:

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta). \tag{2}$$

We also want to further develop this method and add momentum to the SGD. With the SGD we don't calculate the exact derivative of our loss function, but we estimate it in small batches. Therefore we are not always moving in the right direction of our minima, which comes from "noise" in our derivatives. The added momentum will help negate this issue. Another reason for adding momentum is in the ravines. This ravine is where the surface curve very sharply in one demotion, compared to any other. These commonly appeared close to these minimas and the SGD has issues with locating them. This is illustrated for normal SGD with figure 1 and with momentum in figure 1. We can look at momentum as a way to keeping memory of which direction we are moving in.

Figure 1: Shows SGD illustration with ravines from Bushaev, V. (2017, December 5) [5].

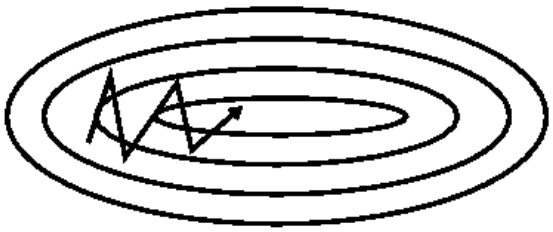

Figure 2: Shows SGD with momentum illustration with ravines from Bushaev, V. (2017, December 5) [5].

Now lets express SGD with momentum and a typical implementation of the method:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta \mathbf{E}(\boldsymbol{\theta}_t + \gamma \mathbf{v}_{t-1}) \tag{3}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t, \tag{4}$$

here we introduce a momentum parameter γ which is defined within $0 \leq \gamma \leq 1$. The gradient is here also over all the different mini batches we just don't denote it for simplicity. The equation is updated with a Nestrov Acceleration Gradient or NAG rule and allows for a larger learning rate than that of a standard gradient decent with momentum. Lastly for this section we define our learning rate (LR) or schedule as:

$$lr = \frac{t_0}{t + t_1}, \tag{5}$$

where t_0 and t_1 is the number of mini-batches and epochs respectively. The only variable in the LR is t and it is this parameter that is sent to the equation to update the LR. Note that we in the code don't calculate the gradients or derivatives, but we use *autograd* which is a python library that does this for us.

2.2 Logistic Regression

In addition to SGD and NN, we will also implement the logistic regression algorithm, which is used to model probability of a certain category. Logistic regression is mostly applied in binary problems, true or false, positive or negative etc. Furthermore, logistic regression will guide us to the next part of neural network.

Since we aimed to use logistic regression for the binary classification, it is very important to encode the binary categories into 0 and 1. We chose the probability higher than or equal to 0.5 as 1 and otherwise 0.

$$y_i = \begin{bmatrix} 0 & no \\ 1 & yes \end{bmatrix} = \begin{bmatrix} 0 & p < 0.5 \\ 1 & p >= 0.5 \end{bmatrix} \tag{6}$$

As known, the linear regression equation can be written as

$$y = \beta_0 + X_1\beta_1 + X_2\beta_2 + ... + X_n\beta_n \tag{7}$$

Given that the probability of a data X_i belonging to a certain category can be calculated by logit/Sigmoid function as follow:

$$p(t) = \frac{1}{1 + exp^{-t}} = \frac{exp^t}{1 + exp^t}, \tag{8}$$

where 1 - p(t) = p(t)

After the combination of this two equation we will take two parameters as illustration, we get the probabilities for each category by:

$$p(y_i = 1|X_i, \beta) = \frac{exp^{\beta_0 + X_1\beta_1}}{1 + exp^{\beta_0 + X_1\beta_1}} \tag{9}$$

$$p(y_i = 0|X_i, \beta) = 1 - p(y_i = 1|X_i, \beta) \tag{10}$$

The maximum likelihood estimation (MLE) was deployed to estimate the parameters of an assumed probability distribution with given a data set. Then our target is to maximize the probability of the given data set. The approximate likelihood of every individual probabilities with respect to a certain outcome y_i which can be obtained from:

$$p(D|\beta) = \prod_{i=1}^{n} [p(y_i = 1|X_i, \beta)]^{y_i} [1 - p(y_i = 1|X_i, \beta)]^{1-y_i} \tag{11}$$

The cost/loss function was obtained by negatively log-likelihood, which is known as the cross entropy in the statistics.

$$C(\beta) = -\sum_{i=1}^{n} (y_i(\beta_0 + X_1\beta_1) - log(1 + exp^{\beta_0 + X_1\beta_1}) \tag{12}$$

In addition the loss from the data set, the l2 regularization loss was also taken into consideration to avoid overfitting of the model on training data. Besides, the l2 regularization was applied in the code of Neural Network as well.

$$C_{reg} = \lambda \sum_{i=ij} w_{ij}^2 \tag{13}$$

In the training of logistic regression, we used simply SGD algorithm to get the optimized parameters and the minimized cost at the same time.

Finally, we will evaluate the performance of our model based on the following equation:

$$Accuracy = \frac{\sum_{i=1}^{n} I(\tilde{y}_i = y_i)}{n} \tag{14}$$

We will also need the R2 score which is defined as:

$$R^2(\hat{y}\hat{\tilde{y}}) = 1 - \frac{\sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1} (y_i - \overline{y}_i)^2}, \tag{15}$$

and the mean value of $\tilde{y}$ is given by:

$$\overline{y} = \frac{1}{n} \sum_{i=0}^{n-1} y_i. \tag{16}$$

2.3 Neural Network

We have now implemented and understood some linear models from Ordinary Least Square (OLS), Ridge, Lasso to the SGD. Now we want to look at Artificial Neural Network (ANN). There are a few different variants of NN. Here we will only focus on the first and simplest one, Feed Forward Neural Network (FFNN), where we have a visual illustration in figure 3 describing how a NN works.

We can define the weights and biases as $P = \{P_{hidden}, P_{outout}\}$ for each neuron and we have N_{hidden} amount of neurons of features in the hidden layer. This P_{hidden} will be a matrix with $N_{features} \times N_{hidden}$ dimensions, where $N_{features}$ are number of features of the input layer. We define our P_{hidden} such that we have the bias in the first column and weights in the other for each hidden neurons. For the output layer the P_{output} will be a $N_{hidden} \times N_{output}$ matrix, with the same shape as P_{hidden} for each neuron. We should define the N_{output} layer based on our targets to be predicted. We then let the function we want to study be $F(N(x, P)) = xN(x, N)$ and $A(x)$ be where our function is $i = 0$. Then we want to find the cost function our NN must solve:

$$\begin{aligned} &\min \left\{ (F'(N(x, P))) - (-\gamma F(N(x, P)))^2 \right\}, \\ &\min\{(F'(N(x, \{P_{hidden}, P_{outout}\}))) \\ &- (-\gamma F(N(x, \{P_{hidden}, P_{outout}\})))^2\}. \end{aligned} \tag{17}$$

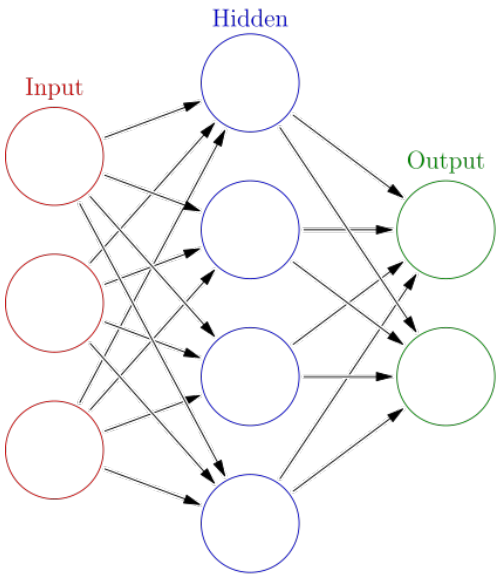

Figure 3: Shows a simple FFNN structure from Wikipedia [2]

We don't just minimize for one value of x, but also for every N iteration as well and then the total error becomes derived form equation 18:

$$\min \left\{ \frac{1}{N} \sum_{i=1}^{N} (F'(N(x,P))) - (-\gamma F(N(x,P)))^2 \right\} \quad (18)$$

then we let $\boldsymbol{x}$ be a vector with x_i elements and also described in terms of P_{hidden} and P_{output}:

$$\min_{P_{hidden}, P_{output}} = C(\boldsymbol{x}, \{P_{hidden}, P_{output}\}). \quad (19)$$

The cost function we chose is the Mean Square Error (MSE) for the regression problem, which is defined as:

$$MSE(\hat{y}\hat{\tilde{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \quad (20)$$

For the loss function we use cross entropy to solve the classification problem, which is defined in equation 12.

Now we want to move on and describe the way we obtain $\boldsymbol{z}$ from the hidden biases and weights. We can define $z_{i,j}^{hidden}$, where j is the neuron of the input layer, as:

$$\begin{aligned} z_{i,j}^{hidden} &= b_i^{hidden} + w_i^{hidden} x_j \\ &= \left(b_i^{hidden} + w_i^{hidden} \right) \begin{pmatrix} 1 \\ x_j \end{pmatrix}, \end{aligned} \quad (21)$$

furthermore we can define for weighting at i-th iteration of the hidden neurons as:

$$\begin{aligned} z_i^{hidden} &= (b_i^{hidden} + w_i^{hidden} x_1 + b_i^{hidden} \\ &+ w_i^{hidden} x_2 + ... + b_i^{hidden} + w_i^{hidden} x_N) \\ &= \begin{pmatrix} b_i^{hidden} & w_i^{hidden} \end{pmatrix} \begin{pmatrix} 1 & 1 & ... & 1 \\ x_1 & x_2 & ... & x_N \end{pmatrix} \\ &= \boldsymbol{p}_{i,hidden}^T \boldsymbol{X}, \end{aligned} \quad (22)$$

We have that $\boldsymbol{p}_{i,hidden}^T$ is each row in P_{hidden}. Now that we have found z_i^{hidden} for every iteration of i we can send this to the activation function of our choice $a_i(\boldsymbol{z})$ and we will have the output of this function $\boldsymbol{x}_i^{hidden}$ equal to:

$$\boldsymbol{x}_i^{hidden} = f(\boldsymbol{z}_i^{hidden}) \quad (23)$$

The output is then sent to the output layer which is comprised of N neurons and combines for each of the neurons in our hidden layer. Lastly the output layer combines the weights and biases of the output layer and we now have the output of $\boldsymbol{z}_i^{output}$:

$$\boldsymbol{z}_i^{output} = \begin{pmatrix} b_i^{output} & w_i^{output} \end{pmatrix} \begin{pmatrix} 1 & 1 & ... & 1 \\ x_1 & x_2 & ... & x_N \end{pmatrix} \quad (24)$$

In this NN we also wanted to add back propagation and feed forward which we will define now and we will start with back propagation. We have to choose a optimization method to minimize the cost function and we will use SGD with momentum, which is described above, for this purpose.

The activation functions we want to study are the Sigmoid, ReLU, Leaky ReLU, the tanh function and

a linear function which is just the same as its inputs. They are defined as such:

$$
\begin{aligned}
Sigmoid &= \frac{1}{1+e^{-x}} \\
ReLU &= \max(0,x) \\
Leaky\ ReLU &= \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \\
tanh &= \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}
\end{aligned} \tag{25}
$$

One might also want to add an accuracy part to see how well we predict the values. We want to study the accuracy of the FFNN and it is described as the sum of correctly guessed targets divided by the total number of the set. We then have the function:

$$
Accuaracy = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{26}
$$

Where t_i is the wanted target an d y_i is the one from our FFNN and n is the total number of the set of targets.

3 Method and data

3.1 Stochastic Gradient Descent

We started by looking at some example code of GD, then started to code with the equations above for SGD and momentum 1 and 4 respectively. Bellow is an outtake of our implementation of just the SGD with momentum calculation. Then we use this code with conjunction with our code for project one with some small adjustments. We have changed the inversion calculation the Ordineary Least Squars (OLS) and Ridge regression method with our SGD. We do this simply by calling our SGD in the class and changing our x and y input to that of the OLS and Ridge regression. We now get the theta by calling our SGD *"StochasticGradientDecent(x, y).SGD()"*.

```
    def SGD(self):
        X = np.c_[np.ones((len(self.x_full),1)), self.x_full]
        #size_matrix = x.shape[0]
        self.theta = np.random.randn(X.shape[1],1) #Initilize theta for matrix shape.

        #xy = np.c_[x.reshape(size_matrix, -1), y.reshape(size_matrix, 1)]

        #Main SGD loop for epochs of minibatches
        for epoc in range(self.n_epoc):
            #Second SGD loop with random choice of k
            for k in range(self.m):
                random_index = self.M*np.random.randint(self.m)
                xi = X[random_index:random_index+self.M]
                yi = self.y_full[random_index:random_index+self.M]

                eta = self.ls(epoc*self.m+k) #Calling function to cal. eta


                #self.v_ = gamma*self.v_ + eta*gradient(x_iter, y_iter, self.theta - gamma*self.v_) #Cal. v where gradient is from autograd
                place_hold = self.theta + self.gamma*self.v_
                x_grad = egrad(self.gradient, 2) #Gradient with respect to theta
                self.v_ = self.gamma*self.v_ + eta * x_grad(xi, yi, place_hold) #Cal. v where gradient is from autograd,  self.gradient(xi, yi, self.theta)
                self.theta = self.theta - self.v_ #Theta +1 from this itteration of theta and v

        return self.theta
```

To collect the data we do two different things, firstly we write the R2 score and the MSE (Mean Square Error) to a file and plot the MSE and R2 score. We also plot the noise on the MSE as well.

We also do some initial testing to make sure we input the correct parameters and variables. Where we check for the length of x and y, that the epochs are the right size and that γ is in range. This makes it easier to know if something is wrong with the initialization of the code and so we don't have to check for every new line of code each time we do something new.

```
    def __call__(self):

        #Checks matrix size of rows
        size_matrix = self.x_full.shape[0]
        if size_matrix != self.y_full.shape[0]:
            raise ValueError("'x' and 'y' must have same rows")

        #Check to see if batches are right size
        self.n_epoc = int(self.n_epoc)
        if not 0 < self.n_epoc <= size_matrix:
            raise ValueError("Must have a batch size less or equal to observations and greater than zero.")

        #Checks gamma is in range
        if not 0 <= self.gamma <= 1:
            raise ValueError("Gamma must be equal or greater than zero and equal or less than 1.")

```

```
        #Checks gamma is in range
        if not 0 <= self.gamma <= 1:
            raise ValueError("Gamma must be equal or greater than zero and equal or less than 1.")
```

3.2 Logistic Regression

Since logistic regression is a linear model, we start by initializing the parameters (weights and biases) for the linear equation. For simplicity, we just initialize the weights as a zero matrix with number of features of input data with 1 dimension and bias as 0. This way we could easily test if the training session is changing the parameters.

```
    def wb_init(self, dim):
        weights = np.zeros((dim,1))
        bias = 0
        return weights, bias
```

The cost function was obtained from combining the Sigmoid function and linear transform of the input data.

```
    def sigmoid(self, z):
        s = 1/(1 + np.exp(-z))
        return s

    def probability(self, weights, X, bias):
        prob = self.sigmoid(np.dot(weights.T, X) + bias)
        return prob

    def cost(self, prob, y, num_sample, weights):
        data_cost = np.mean(-np.sum(y * np.log(prob) + (1-y) * np.log(1-prob)))
        self.data_cost.append(data_cost)

        reg_cost = 0
        if self.regularizer_l2 > 0:
            reg_cost = self.regularizer_l2 * np.sum(weights**2)
            self.reg_cost.append(reg_cost)

        #print("data_cost", data_cost)
        #print("reg_cost", reg_cost)


        cost = data_cost + reg_cost
        #print("cost", cost)

        return cost
```

For the SGD algorithm, we just used a simple version of SGD, where only learning rate was taken into consideration. The useful information like costs, parameters ect. was recorded in the dictionary for further calculation.

```
    def sgd(self, weights, bias, X, y, print_cost = False):
        costs = []

        for i in range(self.n_iter):

            grads, cost = self.model_status(weights, bias, X, y)
            grad_weights = grads["grad_weights"]
            grad_bias = grads["grad_bias"]

            #updates
            weights = weights - self.lr * grad_weights
            bias = bias - self.lr * grad_bias

            #record cost
            costs.append(cost)

            #print cost every 100 training epoch
            if print_cost:
                print ("Cost after iteration %i: %f" %(i, cost))

        parameters = {
            "weights": weights,
            "bias": bias
        }

        gradients = {
            "grad_weights": grad_weights,
            "grad_bias": grad_bias
        }

        return parameters, gradients, costs
```

The training function is calling the initialization function to get weights and bias, the SGD function is to get optimized parameters using train data, and test the model on our test data.

```
    def train(self, X_train, y_train, X_test, y_test, print_cost = False, print_score = False):
        #initilize parameters with normal distribution
        weights, bias = self.wb_init(X_train.shape[0])

        #update parameters with sgd algorithm
        parameters, gradients, costs = self.sgd(weights, bias, X_train, y_train, print_cost)
        self.weights = parameters["weights"]
        self.bias = parameters["bias"]

        #make predictions
        y_pred_test = self.predict(X_test)
        y_pred_train = self.predict(X_train)
        #print(y_pred_train, y_train)

        train_acc = self.accuracy(y_pred_train, y_train)
        test_acc = self.accuracy(y_pred_test, y_test)
        if print_score:
            print ("train accuracy: %f" %(train_acc))
            print ("test accuracy: %f" %(test_acc))

        output = {
            "costs": costs,
            "data_cost": self.data_cost,
            "reg_cost": self.reg_cost,
            "y_pred_train": y_pred_train,
            "y_pred_test": y_pred_test,
            "weights": self.weights,
            "bias": self.bias,
```

```
            "learning rate": self.lr,
            "train accuracy": train_acc,
            "test accuracy": test_acc
        }

        return output
```

The evaluation of logistic model is based the theory part of logistic regression, and calculated as percent and programmed as follow:

```
    def accuracy(self, y_pred, y):
        accuracy = 100 - np.mean(np.abs(y_pred - y)) * 100
        return accuracy
```

3.3 Neural Network

The aim of this part is to write a FFNN and we start by writing some code for the simplest case where we have one hidden layer to get a better understanding of how NN works. We start by setting up our biases and weights, where the weights are normally distributed and we add a small number 0.01, so if they are zero they get a value which is nonzero (we might get complications if we get a divide by zero). Bellow is a snip-it of the code for the defining of the weights an biases.

```
    def crt_b_w(self):
        # weights and bias in our hidden
        #Note addind +0.01 so that if we have zero its changed to a low value
        self.h_weights = 0.01 + np.random.normal(self.features, self.hidden_neurons) #with normal distribution
        self.h_bias = np.zeros(self.hidden_neurons)

        # weights and bias in our output
        self.out_weights = 0.01 + np.random.normal(self.hidden_neurons, self.categories)
        self.out_bias = np.zeros(self.categories)
```

For the back propagation for this the simple case we update the weights and biases in the hidden and output layer. Where the output layer is updated with the activation function and error in the hidden layer for the weights and the error in the output for the biases. The hidden layer is updated for the weights with the X^T and the error in the hidden layer and the bias is the error in the hidden layer. We also update both the weights depending on if $\lambda > 0$, where we multiply λ with the weights and sum them. The last step is to update the weights and biases in the hidden and output layer with η which is our learning rate 5 and then subtract that from the original weights and biases in both layers. We then have the code:

```
    def backprop(self):
        self.error_in_out = self.probability - self.Y_full

        self.error_in_hidden = np.matmul(self.error_in_out, self.out_weights.T) * self.activation_hidden*(1-self.activation_hidden)

        self.grad_weight_out = np.matmul(self.activation_hidden.T, self.error_in_hidden)
        self.grad_bias_out = np.sum(self.error_in_out, axis=0)


        self.grad_weight_hidden = np.matmul(self.X_full.T, self.error_in_hidden)
        self.grad_bias_hidden = np.sum(self.error_in_hidden, axis=0)

        if self.lmbda > 0:
            self.grad_weight_out += self.lmbda * self.out_weights
            self.grad_weight_hidden += self.lmbda * self.h_weights

        self.out_weights -= self.eta*self.grad_weight_out
        self.out_bias -= self.eta*self.grad_bias_out

        self.h_weights -= self.eta*self.grad_weight_hidden
        self.h_bias -= self.eta*self.grad_bias_hidden
```

Then we have the feed forward method which is quiet simply taken from then theory in equation 23 and added to code:

```
    def ff(self):
        #Feed forward for network saved globaly in class
        self.z_hidden = np.matmul(self.X_full, self.h_weights) + self.h_bias

        activation_hidden = self.activation_func_hidden(self.z_hidden)

        self.z_out = np.matmul(activation_hidden, self.out_weights) + self.out_bias
        self.a_expect = self.activation_func_out(self.z_out)
        self.probability = self.a_expect/np.sum(self.a_expect, axis=1, keepdim=True)
```

Finally we add a step to train the function which will take random data points and run both the feed forward and back propagation. We only need to do this one to get the parameters and then we can simply run just for the outputs to get the error, probability and so on.

```
    def train_function(self):
        indec = np.arange(self.inputs)

        for i in range(self.epochs):
            for l in range(self.iter):
                data_points=np.random.choice(indec, size=self.batch_sz, replace=False)

```

```
                self.X_full = self.X_full[
    data_points]
                self.Y_full = self.Y_full[
    data_points]

                self.ff()
                self.backprop()
```

After looking at this simple case of a FFNN we needed to make it generalized so we could change the number of wanted hidden layers. We didn't know where to start, but referenced to the book *"Neural Networks from Scratch in Python"* (Kinsley, H. & Kukieła, D. 2020. [4]) and the lecture notes [1], which are excellent at describing how to set up a NN and for it to have a customizable number of hidden layers. For this we changed the method quite a bit. We now need to add a forward, backward and a predict to all the activation functions so that for each hidden layer we adjust for the activation function. For the layer where we define the weights and biases we also need a forward and backward function to update them accordingly. We start by doing the same as for the simple case, we initialize the weights and biases the same way, but add a regularization to use in the backward function later. The implementation we ended up using is as follows:

```
class Layer():
    def __init__(self, n_features, neurons,
    lmbd = 0):
        # Initialize weights and biases
        self.n_features = n_features
        self.neurons = neurons
        self.weights = 0.01 * np.random.randn(
    self.n_features, self.neurons) #with
    normal distribution
        self.bias = np.zeros((1,self.neurons))

        # Set strength of regularization, the
    regularizer should be greater than or
    equal to 0
        self.weight_regularizer_l2 = lmbd

    def forward(self, inputs, train):
        self.input = inputs

        #Calculate output value from previous
    layer's inputs, weights and biases
        self.out = np.dot(inputs, self.weights
    ) + self.bias

    def back(self, d_val):
        # gradients on weights and biases
        self.grad_weights = np.dot(self.input.
    T, d_val)
        self.grad_bias = np.sum(d_val, axis=0,
     keepdims=True)

        #L2 regularization on weights
        if self.weight_regularizer_l2 > 0:
            self.grad_weights += 2 * self.
    weight_regularizer_l2 * self.weights

        #Gradient on inputs
        self.grad_input = np.dot(d_val, self.
    weights.T)
```

For the activation functions we can look at one example, lets look at the Sigmoid to accommodate for different hidden layers we have to calculate for the forward, backward and the predicted value. Almost every part has this method since we need to update the for each of the hidden layers back and fourth. The forward will just be the sigmoid function of the input, but the backward is $evaluated\ \ value * (1 - output) * output$. Lastly the predicted part is $(output\ > 0.5) * 1$ and the function as code looks like this (we do this for each of the different activation functions):

```
class Activ_Sigmoid():
    def forward(self, inputs, train):
        self.input = inputs

        self.out =  1/(1 + np.exp(-inputs))
    def back(self, d_val):
        self.grad_input = d_val * (1 - self.
    out) * self.out

    def predict(self, out):
        return (out > 0.5) * 1
```

One of the biggest changes is that we split the different parts into their own Classes so that we can utilize them freely as we want and input them with the different layers. Another change is that we now use attributes to classify if we have different parameters or weights for example, by classifying the different inputs of the classes. After changing the code to the more advanced version we also add a method part where we run through the different classes and do the train and finalization. The training is done similarly to the part above we run through the epochs and train our model depending on the activation function chosen and the weights and biases, our implementation of the train is as follows (just a small part of it):

```
    def train(self, X, y, *, n_epoc = 1,
    validation_data = None, print_epoch =
    False):

        self.accuracy.init(y)

        for epochs in range(1, n_epoc+1):

            out = self.forward(X, train=True)

            loss_dat, reg_loss = self.loss.cal
    (out, y, regularization = True)
            loss = loss_dat + reg_loss

            predict = self.activ_out.predict(
    out) #??
            accuracy = self.accuracy.cal(
    predict, y)

            self.back(out, y)

            self.optimiz.pre_up_par()
            for layer in self.tlayer:
                self.optimiz.up_par(layer)
            self.optimiz.post_up_par()
```

We then have the finalization part which runs through the layers by first taking the first layer where $i == 0$ and set the previous and the next object, then we run through for $i < layercount - 1$ for the previous and the next object and lastly save the last objects. Then we check if layer has attribute "weights" and if it does we add it to the trainable layers (we don't need to check for biases) and update the loss object for the trainable layers. We then have the code:

```
        for i in range(layer_iter):

            if i == 0:
                self.layer[i].prev = self.
    layer_inp
                self.layer[i].next = self.
    layer[i+1]

            elif i < layer_iter - 1:
                self.layer[i].prev = self.
    layer[i-1]
                self.layer[i].next = self.
    layer[i+1]

            else:
                self.layer[i].prev = self.
    layer[i-1]
                self.layer[i].next = self.loss
                self.activ_out = self.layer[i]

            if hasattr(self.layer[i], "weights
    "):
                self.tlayer.append(self.layer[
    i])

        # Update loss object with trainable
    layers
        self.loss.remember_training_layer(self
    .tlayer)
```

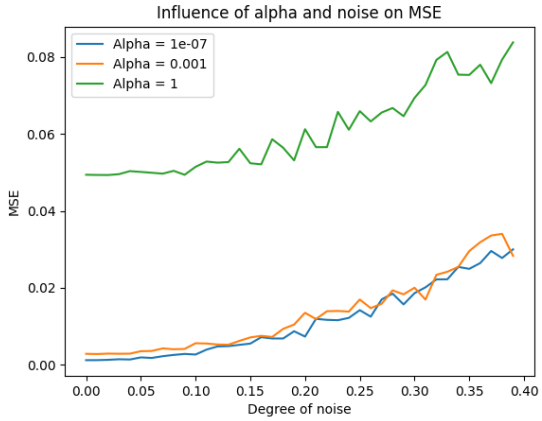

Figure 4: Show the influence of alpha and noise on our MSE with SGD.

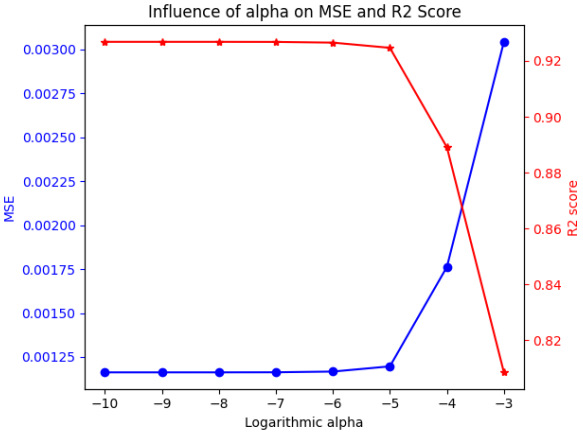

Figure 5: Show the influence of α on MSE Vs. R2 score with SGD.

4 Results

4.1 Stochastic Gradient Descent

With our SGD we tested the implementation by changing the inversion calculation from project of the OLS and Ridge regression. We have the resulting images for the SGD in figure 4 and 5 and for the results from project 1 with pythons inversion we have the figures 6 and 7.

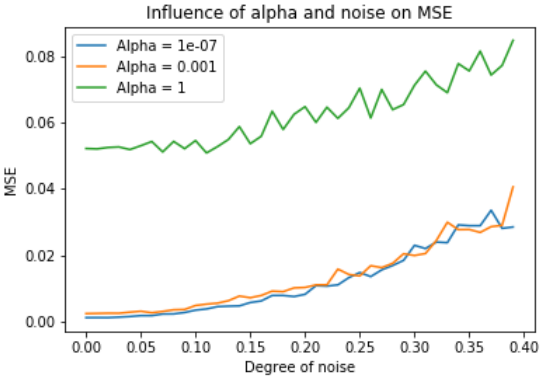

Figure 6: Show the influence of alpha and noise on our MSE with python inversion.

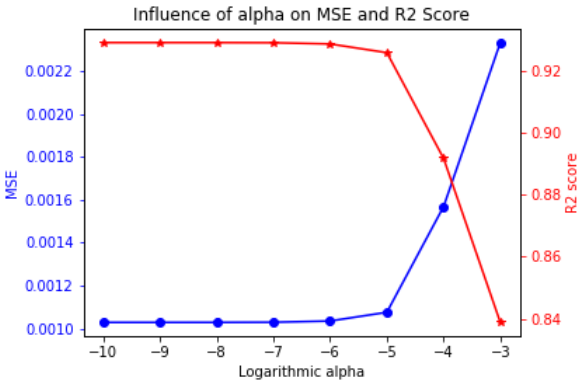

Figure 7: Show the influence of α on MSE Vs. R2 score with python inversion.

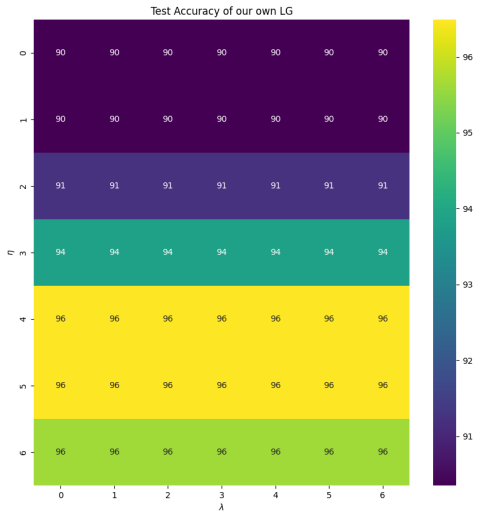

Figure 9: Show the test accuracy of our own LG.

4.2 Logistic Regression

Our logistic regression code was tested with Wisconsin breast cancer data set. The results of training cost was plotted against the increasing epochs (figure 8). In addition, the grid search method was used to find the best combination of the learning rate η and regularization parameter λ with respect to accuracy on training data (figure 9) and accuracy on test data (figure 10).

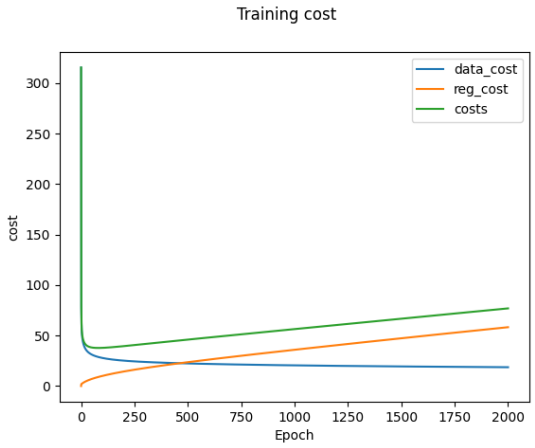

Figure 8: Show the training cost.

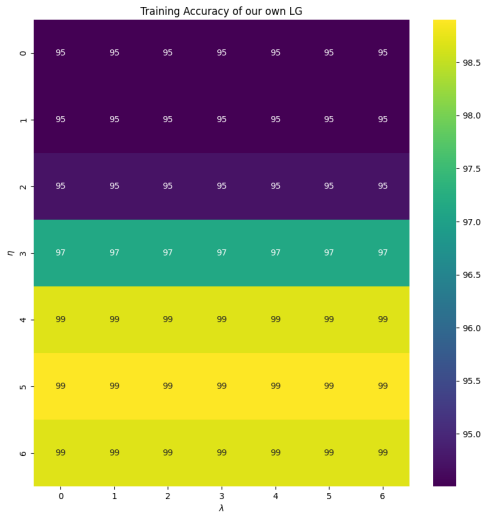

Figure 10: Show the train accuracy of our own LG.

The SGD classifier from sci-kit learn package was used to compare the performance of our own SGD code. The SGD optimizer is the only part that has been replaced with. The similar grid search method is employed as well shown in figure 11 and figure 12.

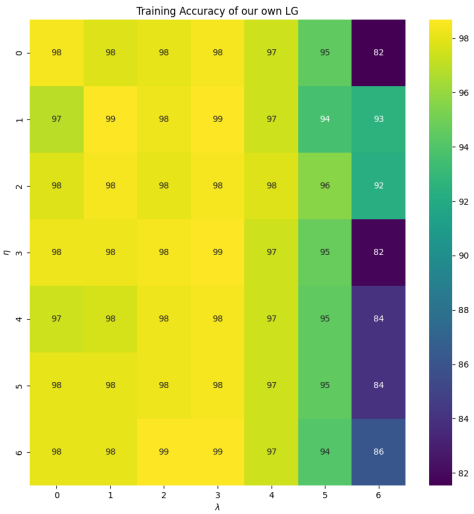

Figure 11: Show the train accuracy of our own LG with SGD classifier from sklearn package.

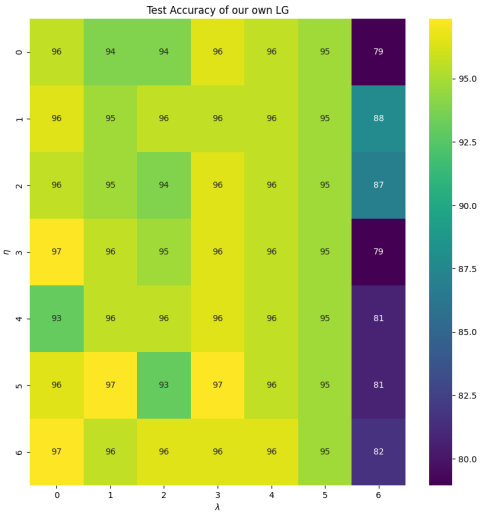

Figure 12: Show the train accuracy of our own LG with SGD classifier from sklearn package.

4.3 FFNN with different activation functions

We firstly looked at the influence of number of neurons in the hidden layer on train and test accuracy and result is shown in figure 13.

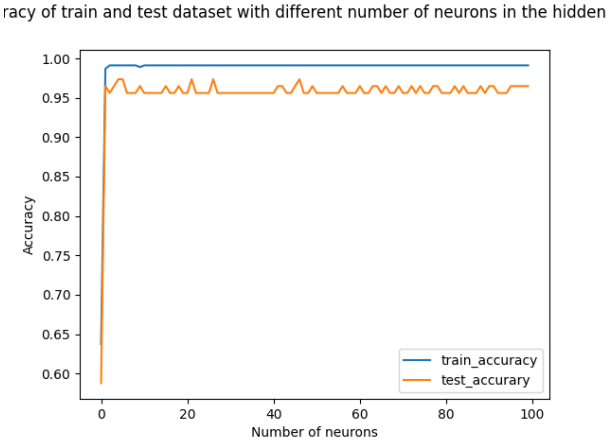

Figure 13: Show the train and test accuracy with different number of hidden neurons.

We tested our FFNN for different the activation functions and made a heat map for the accuracy of the different functions where we plotted both the test accuracy and the train accuracy to see the difference. Firstly we tested the Sigmoid function which is shown in figure 14 and 15 for the test and train. With the heat map it is easy to see the difference before and after and we can see that the accuracy in train is much higher than that of the test.

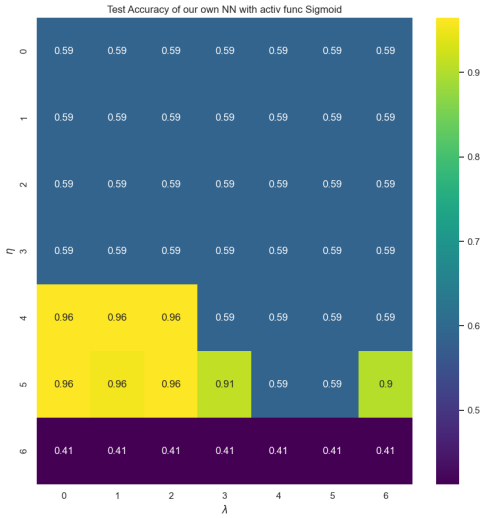

Figure 14: Show the test accuracy of the Sigmoid activation function in then hidden layers.

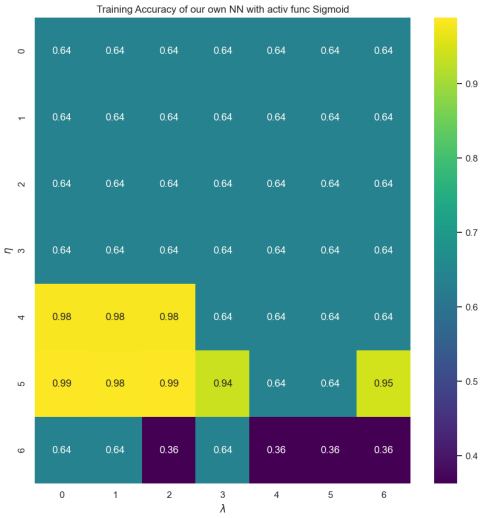

Figure 15: Show the train accuracy of the Sigmoid activation function in then hidden layers.

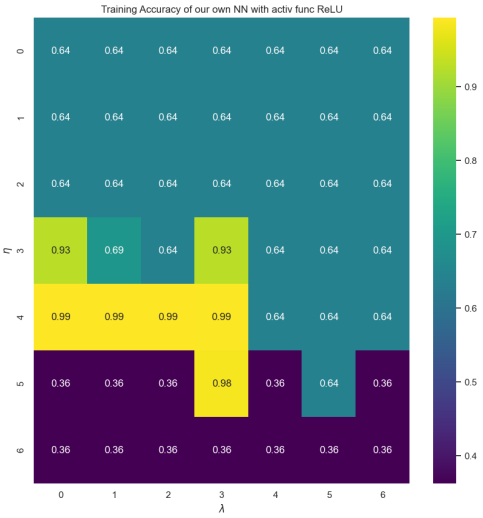

Figure 17: Show the train accuracy of the ReLU activation function in then hidden layers.

Then we tested for the ReLU activation function, which is presented in figure 16 and 17. Where we have an increase in accuracy from the test to the train. We can see that for both the function tested we have a pattern for the accuracy.

We then tested the activation function Leaky ReLU and we have the resulting figures 18 and 19. Where we have quite low accuracy in both cases and it does not increase more than 5% at best and it actually decreases for bigger η.

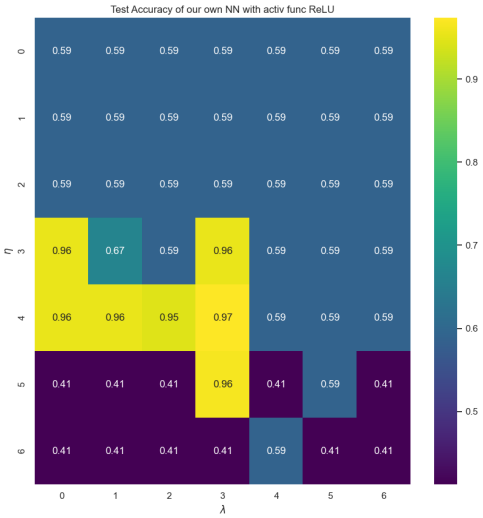

Figure 16: Show the test accuracy of the ReLU activation function in then hidden layers.

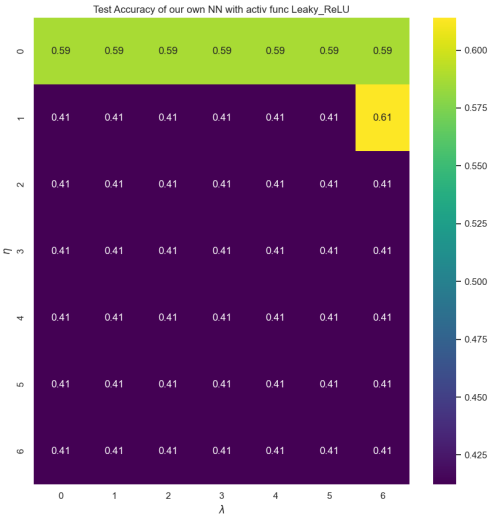

Figure 18: Show the test accuracy of the Leaky ReLU activation function in then hidden layers.

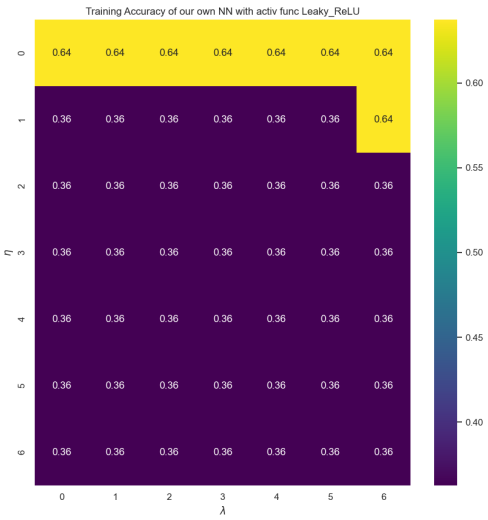

Figure 19: Show the train accuracy of the Leaky ReLU activation function in then hidden layers.

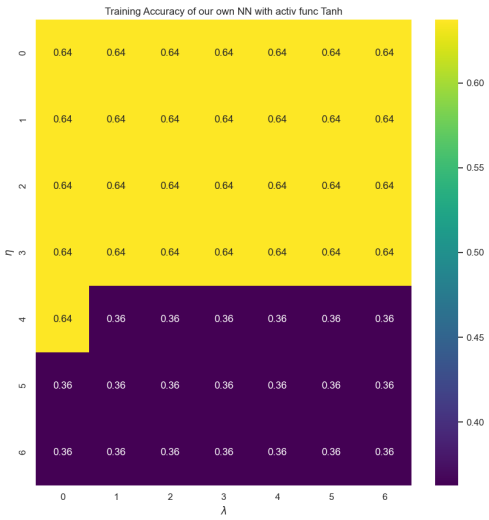

Figure 21: Show the train accuracy of the tanh activation function in then hidden layers.

We ended up testing for two more activation functions the tanh and a linear function. For the tanh we have the resulting figures 20 and 21. This activation function gave low accuracy's for all values of η and λ, it is better for η from 10^{-5} to 10^{-1}. Then for the linear function we have the resulting images 22 and 23. With this activation function we have a wider verity of parameters to choose from and have optimal accuracy's.

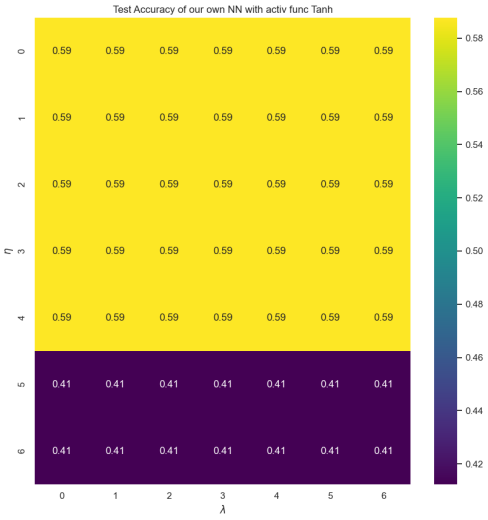

Figure 20: Show the test accuracy of the tanh activation function in then hidden layers.

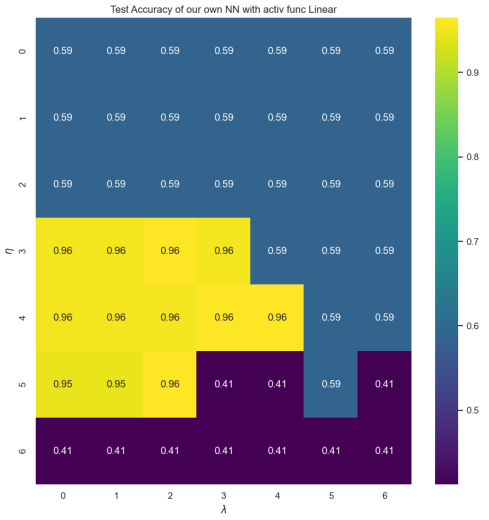

Figure 22: Show the test accuracy with a linear activation function in then hidden layers.

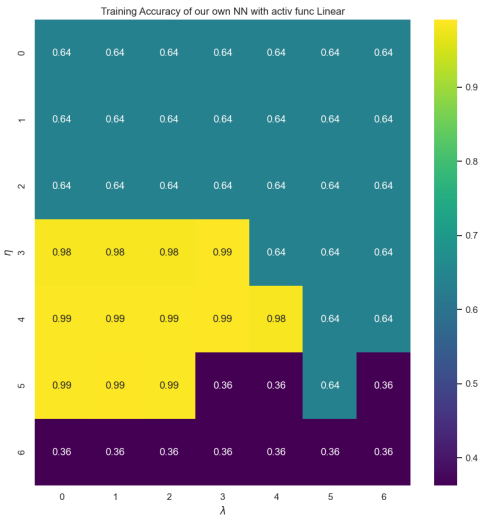

Figure 23: Show the train accuracy with a linear activation function in then hidden layers.

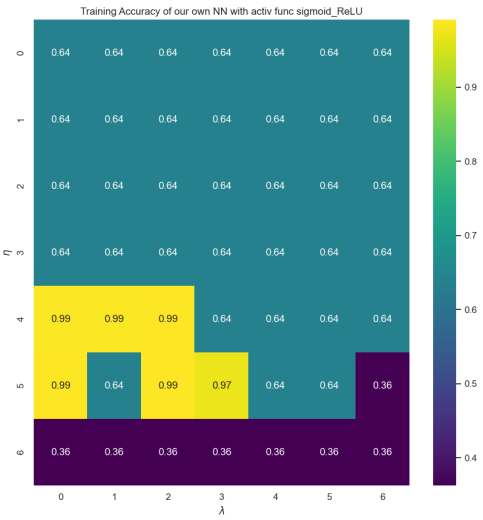

Figure 24: Show the train accuracy with a Sigmoid and ReLU activation functions

The accuracy for all of the activation functions except the tanh function had a final accuracy from 96% to 98% which is very high and acceptable result for the output of the NN.

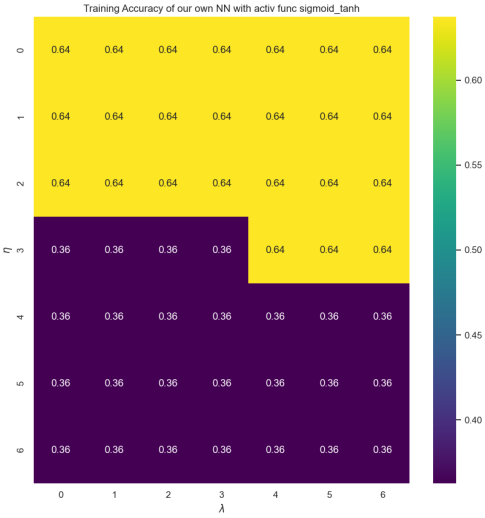

Figure 25: Show the train accuracy with a Sigmoid and tanh activation functions.

We used softmax for all of the results above, but we wanted to see what happen if we changed this up and used for example the ReLU or the tanh function instead. Simply changing the functions gave us the resulting figures for Sigmoid and ReLU 24 and for the Sigmoid and tanh functions we have figure 25.

4.4 NN with Keras

We also wanted to test our code with that of one of the already existing library's NN and chose Keras. We can simply just send our data to the library to get the same results as our own code. Where we tested for one activation function and chose to test with the Sigmoid function. From this we have the figures 26 and 27. Where we can see that we have a much larger area

for choosing η and λ, but the accuracy is still as good as what we got from our FFNN.

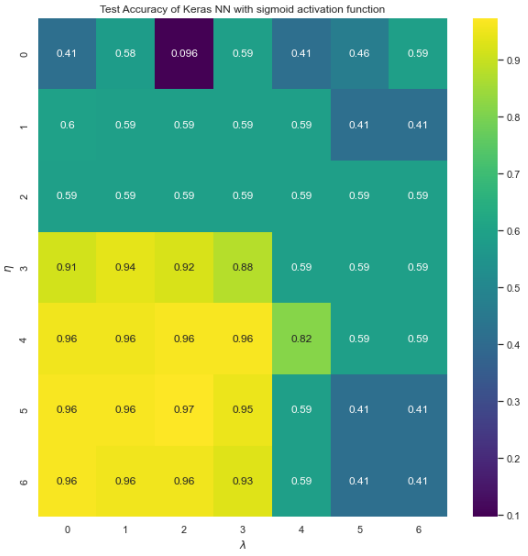

Figure 26: Show the test accuracy with Keras with Sigmoid as the activation function.

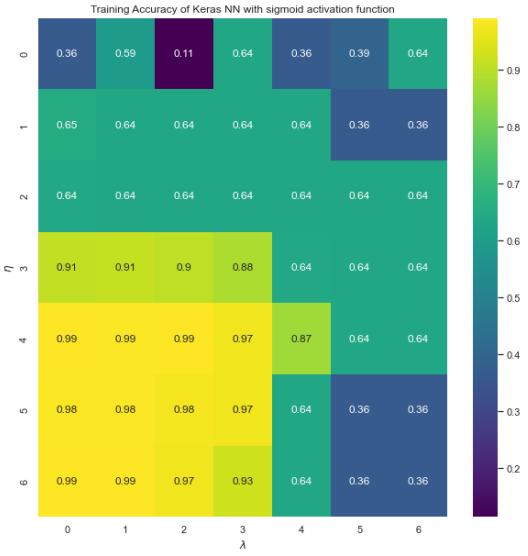

Figure 27: Show the train accuracy with Keras with Sigmoid as the activation function.

5 Discussion

5.1 Stochastic Gradient Descent

From the resulting images we can clearly see that changing the inversion calculation with our own SGD there is not a huge difference. We have that the MSE and R2 score staid almost the same and the difference is negligible, the same can be said for the influence of noise and α, but the most interesting part is the MSE and R2 score.

5.2 Logistic Regression

The training cost from figure 8 shows how the cost from our data set and regularization cost change with the increasing epochs. However, the regularization cost becomes higher and higher with the increasing epoch, which we expected to be plateau from a certain epoch. In addition, the train and test accuracy upon different learning rate η and regularization parameter λ indicates that the model is sensitive to the learning rate, but not responsive to the regularization parameter. Therefore, we speculate that the SGD algorithm may have some defect, resulting in the continuous changing of parameters for every epoch.

In order to validate if the SGD algorithm has problems, we chose to use the SGD classifier from sci-kit learn package in replacement of our own SGD algorithm used in logistic regression. In the resulting SGD classifier it seems that the regularization parameter λ is able to influence the accuracy on both train and test data.

Ignoring of the regularization parameter, we could get the best performance with learning rate of 0.1 and 1.

5.3 FFNN with different activation functions

The number of neurons does not seem to influence both train and test accuracy that much as long as the number of neurons are not too small. This means that we could improve the runtime hugely without sacrificing the performance by selecting the proper number of neurons in the hidden layer.

The different activation functions differed from each other and we can see that for the three first, we have some difference in where they are optimal for the different values of the hyper-parameters η and λ. We got the biggest area for these parameters by using the linear function as the activation function. Surprisingly the ReLU function preformed much better than the Leaky ReLU. We would expect the leaky version to preform much better, since we don't have to face the "dying ReLU" problem and could catch some characteristics when we have values for ReLU which are always under zero. We would expect that the backward would preform on the learning rate, but it preforms worse. From the Sigmoid function we can see that it works well for our data where we have either "True" or "False", but it is not a zero-centric function which might give us some problems. This function is also quite computationally expensive because of its exponential in its nature, this is the

same for tanh. The ReLU on the other hand is not as computationally expensive.

We also changed the softmax function with both ReLU and tanh to see if they would preform better. The resulting images shows that they did change the outcome, but not as one would expect. We might expect that the change would impact each other positively where they have low accuracy's, whereas making it preform inadequately instead.

5.4 NN with Keras

When we tested the Keras package the resulting heat maps where we have a large area for the choosing the hyper-parameters η and λ. There are a lot of values that give an accuracy of over 90%. The difference in Keras and our own code is not huge if we look at the Sigmoid results we have a lot of similarities, but just a smaller area for the η and λ values. Then our code preforms well, but we have some improvements if we want to make it even better. In the Keras training results there is an anomaly for $\eta = 10^{-5}$ an $\lambda = 10^{-3}$, where we have en exceptionally small accuracy. This might come form the model breaking down with the Sigmoid function and resulting in a abnormally small accuracy which is not good.

6 Conclusion

The resulting accuracy's from our model and that of already well made implementations we can see that we get quite good accuracy's as long as we are careful about choosing the right hyper parameters. We get an accuracy well above 90%. The advantages of the Sigmoid function is that it preforms well in its bounds and overall not terrible outside this area, the same can be said for the ReLU and the linear functions. Althoug the linear function might lose some of the data and miss some of the information. The worse functions where the Leaky ReLU and tanh, which preformed so bad for this NN that we should not use them for this application. By testing different activation function in the output layer we also saw that the best preforming function was the softmax, where it gave consistent results. The softmax is normally used for where we have an output in multi-class classification problems.

The implementation of logistic regression on the same data got quite similar performance, with right hyper parameters. They both can reach about 97% accuracy on test data and more than 99% on train data. The runtime of both models is highly dependent on the number of iteration. It would be interesting to test the runtime for both models with their best combination of hyper parameters for the further steps of implementing these two models. The best activation functions for classification cases will be the softmax and sigmoid and for regression the linear would work best.

A Appendix A CODE:

The code and results can be found following this link to our GitHub

A.1 Stochastic Gradient Decent

```

import numpy as npn
import autograd.numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from autograd import grad
from autograd import elementwise_grad as egrad  # for functions that vectorize over inputs
from autograd import holomorphic_grad as hgrad
from sklearn import linear_model
#from AnalysisFunctions import *
from sklearn.preprocessing import PolynomialFeatures
import matplotlib.pyplot as plt
from matplotlib import cm
from imageio import imread

"""
Analasys functions from project 1
"""

def FrankeFunction(x,y, noise = 0):
    term1 = 0.75*np.exp(-(0.25*(9*x-2)**2) - 0.25*((9*y-2)**2))
    term2 = 0.75*np.exp(-((9*x+1)**2)/49.0 - 0.1*(9*y+1))
    term3 = 0.5*np.exp(-(9*x-7)**2/4.0 - 0.25*((9*y-3)**2))
    term4 = -0.2*np.exp(-(9*x-4)**2 - (9*y-7)**2)
    return (term1 + term2 + term3 + term4 + noise*np.random.randn(len(x)))


def R2(zReal, zPredicted):
    """
    :param zReal: actual z-values, size (n, 1)
    :param zPredicted: predicted z-values, size (n, 1)
    :return: R2-score
    """
    R2 = 1 - (np.sum((zReal - zPredicted)**2)/np.sum((zReal - np.mean(zReal))**2))
    return R2

def MeanSquaredError(zReal, zPredicted):
    """
    :param zReal: actual z-values, size (n, 1)
    :param zPredicted: predicted z-values, size (n, 1)
    :return: Mean squared error
    """
    MSE = np.sum((zReal - zPredicted)**2)/len(z)
    return MSE

def betaCI_OLS(zReal, beta_mean, X):
    """
    :param zReal: actual z-values, size (n, 1)
    :param beta_mean: mean of beta
    :param X: dataset
    Compute a 90% confidence interval for the beta coefficients
    """

    # Calculate variance squared in the error
    z_hat = X.dot(beta)
    N, P = np.shape(X)
    sigma2 = (np.sum(np.power((zReal-z_hat), 2)))/N

    # Calculate the variance squared of the beta coefficients
    var_beta = np.diag(sigma2*np.linalg.inv((X.T.dot(X))))

    # The square root of var_beta is the standard error. Confidence intervals are calculated as mean
     +/- Z*SE
    ci_minus = beta_mean - 1.645*var_beta
    ci_plus = beta_mean + 1.645*var_beta
```

```
    return ci_minus, ci_plus


def betaCI_Ridge(zReal, beta_mean, X, l):
    """
    :param zReal: actual z-values, size (n, 1)
    :param beta_mean: mean of beta
    :param X: dataset
    Compute a 90% confidence interval for the beta coefficients - Ridge
    """

    # Calculate variance squared in the error
    z_hat = X.dot(beta)
    N, P = np.shape(X)
    sigma_2 = (np.sum(np.power((zReal-z_hat), 2)))/N

    # Calculate the variance squared of the beta coefficients
    XTX= X.T.dot(X)
    R, R = np.shape(XTX)
    var_beta = np.diag(sigma_2*np.linalg.inv((XTX + l*np.identity(R))))

    # The square root of var_beta is the standard error. Confidence intervals are calculated as mean
     +/- Z*SE
    ci_minus = beta_mean - 1.645*var_beta
    ci_plus = beta_mean + 1.645*var_beta

    return ci_minus, ci_plus

def plotFrankes(x_, y_, z_):
    """
    Plot Franke's function
    """

    fig = plt.figure()
    ax = fig.gca(projection='3d')

    surf = ax.plot_surface(x_, y_, z_, cmap=cm.coolwarm,
                           linewidth=0, antialiased=False)

    # Customize the z axis.
    ax.set_zlim(-0.10, 1.40)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))


    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z - Franke')

    # Add a color bar which maps values to colors.
    clb = fig.colorbar(surf, shrink=0.5, aspect=5)
    clb.ax.set_title('Level')

    plt.show()

#Ordinary Least Squared function
def ols(x, y, z, degree = 5):
    #x: vector of size(n, 1)
    #y: vector of size(n,1)
    #z: vector of size(n,1)
    xyb_ = np.c_[x, y]
    poly = PolynomialFeatures(degree)
    xyb = poly.fit_transform(xyb_)

    #Change from inverse to SGD
    #print("SGD=",np.shape(StochasticGradientDecent(x = (xyb.T.dot(xyb)), y=z).SGD()))
    #beta = (StochasticGradientDecent(x = (xyb.T.dot(xyb)), y=z).SGD()).dot(xyb.T).dot(z)
    #beta = StochasticGradientDecent(x = xyb, y=z).SGD().dot(xyb.T).dot(z)
    beta = StochasticGradientDecent(x, y).SGD().dot(xyb.T).dot(z)
    #beta = np.linalg.inv(xyb.T.dot(xyb)).dot(xyb.T).dot(z)

    return beta
```

```
137
138 def RidgeRegression(x, y, z, degree=5, l=0.0001):
139     """
140     :param x: numpy vector of size (n, 1)
141     :param y: numpy vector of size (n, 1)
142     :param degree: degree of polynomial fit
143     :param l: Ridge penalty coefficient
144     :return: numpy array with the beta coefficients
145     """
146     # Calculate matrix with x, y - polynomials
147     M_ = np.c_[x, y]
148     poly = PolynomialFeatures(degree)
149     M = poly.fit_transform(M_)
150
151     # Calculate beta
152     A = np.arange(1, degree + 2)
153     rows = np.sum(A)
154     #Change from inverse to SGD
155     #beta = (StochasticGradientDecent(x = (M.T.dot(M) + l * np.identity(rows)), y=y).SGD()).dot(M.T)
        .dot(z)
156     beta = (StochasticGradientDecent(x = (M.T.dot(M) + l * np.identity(rows)), y=y).SGD()).dot(M.T).
        dot(z)
157
158     return beta
159
160 def Lasso(x, y, z, degree=5, a=1e-06):
161
162     X = np.c_[x, y]
163     poly = PolynomialFeatures(degree=degree)
164     X_ = poly.fit_transform(X)
165
166     clf = linear_model.Lasso(alpha=a, max_iter=5000, fit_intercept=False)
167     clf.fit(X_, z)
168     beta = clf.coef_
169
170     return beta
171
172 def bootstrap(x, y, z, p_degree, method, n_bootstrap=100):
173     # Randomly shuffle data
174     data_set = np.c_[x, y, z]
175     np.random.shuffle(data_set)
176     set_size = round(len(x)/5)
177
178     # Extract test-set, never used in training. About 1/5 of total data
179     x_test = data_set[0:set_size, 0]
180     y_test = data_set[0:set_size, 1]
181     z_test = data_set[0:set_size, 2]
182     test_indices = np.linspace(0, set_size-1, set_size)
183
184     # And define the training set as the rest of the data
185     x_train = np.delete(data_set[:, 0], test_indices)
186     y_train = np.delete(data_set[:, 1], test_indices)
187     z_train = np.delete(data_set[:, 2], test_indices)
188
189     Z_predict = []
190
191     MSE = []
192     R2s = []
193     for i in range(n_bootstrap):
194         x_, y_, z_ = resample(x_train, y_train, z_train)
195
196         if method == 'Ridge':
197             # Ridge regression, save beta values
198             beta = RidgeRegression(x_, y_, z_, degree=p_degree)
199         elif method == 'Lasso':
200             beta = Lasso(x_, y_, z_, degree=p_degree)
201         elif method == 'OLS':
202             beta = ols(x_, y_, z_, degree=p_degree)
203         else:
204             print('ERROR: Cannot recognize method')
205             return 0
206
207         M_ = np.c_[x_test, y_test]
```

```
        poly = PolynomialFeatures(p_degree)
        M = poly.fit_transform(M_)
        z_hat = M.dot(beta)

        Z_predict.append(z_hat)

        # Calculate MSE
        MSE.append(np.mean((z_test - z_hat)**2))
        R2s.append(R2(z_test, z_hat))

    # Calculate MSE, Bias and Variance
    MSE_M = np.mean(MSE)
    R2_M = np.mean(R2s)
    bias = np.mean((z_test - np.mean(Z_predict, axis=0, keepdims=True))**2)
    variance = np.mean(np.var(Z_predict, axis=0, keepdims=True))
    return MSE_M, R2_M, bias, variance



class StochasticGradientDecent(object):
    """docstring for StochasticGradientDecent."""

    def __init__(self, x, y, n_epoc = 50, M = 10, n=1000, gamma=0.9, dtype = "float64"):

        self.x_full = x
        self.y_full = y
        self.n_epoc = n_epoc
        #size of each minibatch
        self.M = M
        self.n = n
        self.gamma = gamma
        #Some initial conditions
        #nunber of minibatch
        self.m = int(self.n/self.M)
        self.t0 = self.M
        self.t1 = self.n_epoc
        #theta dimension is based on the number of columns in design matrix
        self.v_ = 0

    def __call__(self):

        #Checks matrix size of rows
        size_matrix = self.x_full.shape[0]
        if size_matrix != self.y_full.shape[0]:
            raise ValueError("'x' and 'y' must have same rows")

        #Check to see if batches are right size
        self.n_epoc = int(self.n_epoc)
        if not 0 < self.n_epoc <= size_matrix:
            raise ValueError("Must have a batch size less or equal to observations and greater than
    zero.")

        #Checks gamma is in range
        if not 0 <= self.gamma <= 1:
            raise ValueError("Gamma must be equal or greater than zero and equal or less than 1.")

        #Checks gamma is in range
        if not 0 <= self.gamma <= 1:
            raise ValueError("Gamma must be equal or greater than zero and equal or less than 1.")

    #gradient
    def gradient(self, x, y, theta):
        return (2.0/self.M)*x.T @ ((x @ theta) - y).T

    #This the learning scheduel for eta
    def ls(self, t):
        return self.t0/(t+self.t1)

    #The eta values function
    #def eta(self, t):
        #return self.t0**2/(t+self.t1)


```

```
    def SGD(self):
        X = np.c_[np.ones((len(self.x_full),1)), self.x_full]
        #size_matrix = x.shape[0]
        self.theta = np.random.randn(X.shape[1],1) #Initilize theta for matrix shape.

        #xy = np.c_[x.reshape(size_matrix, -1), y.reshape(size_matrix, 1)]

        #Main SGD loop for epochs of minibatches
        for epoc in range(self.n_epoc):
            #Second SGD loop with random choice of k
            for k in range(self.m):
                random_index = self.M*np.random.randint(self.m)
                xi = X[random_index:random_index+self.M]
                yi = self.y_full[random_index:random_index+self.M]

                eta = self.ls(epoc*self.m+k) #Calling function to cal. eta


                #self.v_ = gamma*self.v_ + eta*gradient(x_iter, y_iter, self.theta - gamma*self.v_)
    #Cal. v where gradient is from autograd
                place_hold = self.theta + self.gamma*self.v_
                x_grad = egrad(self.gradient, 2) #Gradient with respect to theta
                self.v_ = self.gamma*self.v_ + eta * x_grad(xi, yi, place_hold) #Cal. v where
    gradient is from autograd,  self.gradient(xi, yi, self.theta)
                self.theta = self.theta - self.v_ #Theta +1 from this itteration of theta and v

        return self.theta


"""
Analysis of a Lasso Regression model of Franke's function from project 1
"""

"""

# Load data
X = np.load('data.npy')
x = X[:, 0]
y = X[:, 1]
z = FrankeFunction(x, y)

alphas = [10**-10, 10**-9, 10**-8, 10**-7, 10**-6, 10**-5, 10**-4, 10**-3]
alpha_logs = [-10, -9, -8, -7, -6, -5, -4, -3]

Bs = []
for al in alphas:
    Bs.append(Lasso(x, y, z, degree=5, a=al))

# Generate new test data
x_test = np.random.rand(200)
y_test = np.random.rand(200)
z_test = FrankeFunction(x_test, y_test)

M_ = np.c_[x_test, y_test]
poly = PolynomialFeatures(5)
M = poly.fit_transform(M_)
MSEs = []
R2s = []
text_file = open("../Results/SGD/Bootstrap_lasso_SGD.txt", "w")
for i in range(len(alphas)):
    z_predict = M.dot(Bs[i])
    MSE = MeanSquaredError(z_test, z_predict)
    MSEs.append(MSE)
    R2_score = R2(z_test, z_predict)
    R2s.append(R2_score)
    text_file.write('--- Alpha value: {0} ---\n Mean Squared error: {1:.7f} \n R2 Score: {2:.7f}\n'.
    format(alphas[i], MSE, R2_score))

# make plot
fig, ax1 = plt.subplots()
ax1.plot(alpha_logs, MSEs, 'bo-')
ax1.set_xlabel('Logarithmic alpha')
# Make the y-axis label, ticks and tick labels match the line color.
```

```
ax1.set_ylabel('MSE', color='b')
ax1.tick_params('y', colors='b')

ax2 = ax1.twinx()
ax2.plot(alpha_logs, R2s, 'r*-')
ax2.set_ylabel('R2 score', color='r')
ax2.tick_params('y', colors='r')

plt.title('Influence of alpha on MSE and R2 Score')
fig.tight_layout()
plt.savefig('../Results/SGD/MSE_R2_alpha_SGD.png')

# Investigate how the alpha values are influenced by noise
noise = np.arange(0, 0.4, 0.01)
alphas = [10**-7, 10**-3, 1]
Bs = []

# Generate more data to test
x_test = np.random.rand(200)
y_test = np.random.rand(200)
M_ = np.c_[x_test, y_test]
poly5 = PolynomialFeatures(5)
M = poly5.fit_transform(M_)

for al in alphas:
    B = []
    #print(al)
    for n in noise:
        z = FrankeFunction(x, y, noise=n)
        B.append(Lasso(x, y, z, degree=5, a=al))
    Bs.append(B)

lines = []
plt.figure()
for i in range(len(alphas)):
    text_file.write('--- alpha value: {} --- \n'.format(alphas[i]))
    line = []
    for j in range(len(noise)):
        z_test = FrankeFunction(x_test, y_test, noise=noise[j])
        z_predict = M.dot(Bs[i][j])
        MSE = MeanSquaredError(z_test, z_predict)
        line.append(MSE)
        R2_score = R2(z_test, z_predict)
        text_file.write(' Noise: {0} \n Mean Squared error: {1:.7f} \n R2 Score: {2:.7f}\n'.format(
    noise[j], MSE, R2_score))
    plt.plot(noise, line, label='Alpha = {0}'.format(alphas[i]))

plt.legend()
plt.xlabel('Degree of noise')
plt.ylabel('MSE')
plt.title('Influence of alpha and noise on MSE')
plt.savefig('../Results/SGD/alpha_noise_MSE_SGD.png')

MSE_l, R2_l, bias_l, variance_l = bootstrap(x, y, z, method='Lasso', p_degree=5)
text_file.write('--- BOOTSTRAP --- \n')
text_file.write('MSE: {} \n'.format(MSE_l))
text_file.write('R2: {} \n'.format(R2_l))
text_file.write('Bias: {} \n'.format(bias_l))
text_file.write('Variance: {} \n'.format(variance_l))

text_file.close()
"""



"""
Part 6 from project 1
"""

# Load the terrain
terrain1 = imread('SRTM_data_Norway_2.tif')
# Show the terrain
plt.figure()
```

```
plt.title('Terrain area')
plt.imshow(terrain1, cmap='gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

# Choose a smaller part of the data set
terrain = terrain1[500:750, 0:250]
# Show the terrain
plt.figure()
plt.imshow(terrain, cmap='gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('../Results/SGD/terrain_original.png')

# Make zero matrix to later fit data
num_rows, num_cols = np.shape(terrain)
num_observations = num_rows * num_cols
X = np.zeros((num_observations, 3))

# make a matrix with all the values from the data on the form [x y z]
index = 0
#X = X-np.mean(X)
for i in range(0, num_rows):
    for j in range(0, num_cols):
        X[index, 0] = i  # x
        X[index, 1] = j  # y
        X[index, 2] = terrain[i, j]  # z
        index += 1

# OLS example
# extract x, y, z
xt = X[:,0, np.newaxis]
yt = X[:,1, np.newaxis]
zt = X[:,2, np.newaxis]




degree = [2, 4, 6, 8]
text_file = open("../Results/SGD/terrain_CI_ols.txt", "w")
for d in degree:
    beta = ols(xt, yt, zt, degree=d)

    M_ = np.c_[xt, yt]
    poly = PolynomialFeatures(d)
    M = poly.fit_transform(M_)
    z_predict = M.dot(beta)


    T = np.zeros([num_rows, num_cols])
    index = 0
    # create matrix for imshow
    for i in range(0, num_rows):
        for j in range(0, num_cols):
            T[i, j] = (z_predict[index])
            index += 1
    plt.figure()
    plt.imshow(T, cmap='gray')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.savefig('../Results/SGD/terrain_ols_d{}.png'.format(d))


    z_test = np.zeros(zt.shape[0])
    for i in range(zt.shape[0]):
        z_test[i] = zt[i][0]

    beta_test = np.zeros(beta.shape[0])
    for i in range(beta.shape[0]):
        beta_test[i] = zt[i][0]


```

```
    conf1, conf2 = betaCI_OLS(z_test, beta_test, M)
    #print(conf2.shape)
    for i in range(len(conf1)):
        text_file.write('Beta {0}: {1:5f} & [{2:5f}, {3:5f}] \n'.format(i, beta_test[i], conf1[i], conf2[i]))

text_file.close()

# Evaluate model with bootstrap algorithm
text_file = open("../Results/SGD/terrain_mse_ols.txt", "w")
MSE, R2, bias, variance = bootstrap(xt, yt, zt, p_degree=8, method='OLS', n_bootstrap=100)
text_file.write('MSE: {0:5f} & R2: {1:5f} & bias: {2:5f} & var: {3:5f}'.format(MSE, R2, bias, variance))
text_file.close()

##############################################################################################
##Ridge Regression
##############################################################################################
text_file = open("../Results/SGD/terrain_CI_ridge.txt", "w")
for d in degree:
    beta = RidgeRegression(xt, yt, zt, degree=d)

    M_ = np.c_[xt, yt]
    poly = PolynomialFeatures(d)
    M = poly.fit_transform(M_)
    z_predict = M.dot(beta)


    T = np.zeros([num_rows, num_cols])
    index = 0
    # create matrix for imshow
    for i in range(0, num_rows):
        for j in range(0, num_cols):
            T[i, j] = (z_predict[index])
            index += 1
    plt.figure()
    plt.imshow(T, cmap='gray')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.savefig('../Results/SGD/terrain_ridge_d{}.png'.format(d))


    z_test = np.zeros(zt.shape[0])
    for i in range(zt.shape[0]):
        z_test[i] = zt[i][0]

    beta_test = np.zeros(beta.shape[0])
    for i in range(beta.shape[0]):
        beta_test[i] = zt[i][0]


    conf1, conf2 = betaCI_OLS(z_test, beta_test, M)
    #print(conf2.shape)
    for i in range(len(conf1)):
        text_file.write('Beta {0}: {1:5f} & [{2:5f}, {3:5f}] \n'.format(i, beta_test[i], conf1[i], conf2[i]))

text_file.close()

# Evaluate model with bootstrap algorithm
text_file = open("../Results/SGD/terrain_mse_ridge.txt", "w")
MSE, R2, bias, variance = bootstrap(xt, yt, zt, p_degree=8, method='Ridge', n_bootstrap=100)
text_file.write('MSE: {0:5f} & R2: {1:5f} & bias: {2:5f} & var: {3:5f}'.format(MSE, R2, bias, variance))
text_file.close()

##############################################################################################
##Lasso Regression
##############################################################################################
text_file = open("../Results/SGD/terrain_CI_lasso.txt", "w")
for d in degree:
    beta = Lasso(xt, yt, zt, degree=d)

```

```
    M_ = np.c_[xt, yt]
    poly = PolynomialFeatures(d)
    M = poly.fit_transform(M_)
    z_predict = M.dot(beta)


    T = np.zeros([num_rows, num_cols])
    index = 0
    # create matrix for imshow
    for i in range(0, num_rows):
        for j in range(0, num_cols):
            T[i, j] = (z_predict[index])
            index += 1
    plt.figure()
    plt.imshow(T, cmap='gray')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.savefig('../Results/SGD/terrain_lasso_d{}.png'.format(d))


    z_test = np.zeros(zt.shape[0])
    for i in range(zt.shape[0]):
        z_test[i] = zt[i][0]

    beta_test = np.zeros(beta.shape[0])
    for i in range(beta.shape[0]):
        beta_test[i] = zt[i][0]


    conf1, conf2 = betaCI_OLS(z_test, beta_test, M)
    #print(conf2.shape)
    for i in range(len(conf1)):
        text_file.write('Beta {0}: {1:5f} & [{2:5f}, {3:5f}] \n'.format(i, beta_test[i], conf1[i], conf2[i]))
text_file.close

# Evaluate model with bootstrap algorithm
text_file = open("../Results/SGD/terrain_mse_lasso.txt", "w")
MSE_l, R2_l, bias_l, variance_l = bootstrap(xt, yt, zt, method='Lasso', p_degree=8)
text_file.write('MSE: {0:5f} & R2: {1:5f} & bias: {2:5f} & var: {3:5f}'.format(MSE_l, R2_l, bias_l, variance_l))
text_file.close()


"""
n = 1000
x = 2*np.random.rand(n,1)
y = 4+3*x+np.random.randn(n,1)
X = np.c_[np.ones((n,1)), x]

#print(X)


if __name__ == "__main__":
    theta = StochasticGradientDecent(x, y).SGD()
    print(theta)
"""
```

A.2 FFNN simple case

```
import autograd.numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from autograd import grad
from autograd import elementwise_grad as egrad  # for functions that vectorize over inputs
from sklearn import linear_model
from SGD import StochasticGradientDecent


"""Feed Forward Neural Network"""
class NeuralNetwork(object):
    def __init__(self, x, y, hidden_neurons=50, categories= 10, n_epochs=10 ,batch_sz = 100, lmbda =
```

```
 0.001, activation_func_hidden = "sigmoid", activation_func_out = "tanh"):

        #Setting up initial conditions for class
        self.X_full = x
        self.Y_full = y

        self.input = x.shape[0]
        self.feauters = x.shape[1]
        self.hidden_neurons = hidden_neurons
        self.categories = categories
        self.n_epochs = n_epochs
        self.lmbda = lmbda
        self.batch_sz=  batch_sz
        self.iter = self.input // self.batch_sz
        self.eta = (self.n_epochs/2)/(self.n_epochs/2+self.n_epochs) #Learning scheduel

        #Initilize theta from SGD
        self.theta = StochasticGradientDecent(x, y, n_epoc = self.n_epochs, M = self.categories, n=
    np.size(self.input), gamma=0.3).SGD()

        #Allows other activation function for hidden layer
        if activation_func_hidden == "sigmoid":
            self.activation_func_hidden = self.sigmoid
        #Allows other activation function for output layer
        elif activation_func_out == "tanh":
            self.activation_func_out = self.tanh
        elif activation_func_out == "relu":
            self.activation_func_out = self.re

        #Creating bias and weight by running function
        self.crt_b_w()

    def __call__(self):

        #Checks matrix size of rows
        size_matrix = self.x_full.shape[0]
        if size_matrix != self.y_full.shape[0]:
            raise ValueError("'x' and 'y' must have same rows")

        #Small tests to check input
        size_matrix = X.shape[0]
        self.n_epochs = int(self.n_epoch)
        if not 0 < self.n_epoch <= size_matrix:
            raise ValueError("Must have a 'epochs' size less or equal to observations and greater
    than zero")

        self.batch_sz = int(self.batch_sz)
        if  self.batch_sz <= 0:
            raise ValueError("'Batch size' must be greater than 0.")



    def crt_b_w(self):
        # weights and bias in our hidden
        #Note addind +0.01 so that if we have zero its changed to a low value
        self.h_weights = 0.01 + np.random.normal(self.features, self.hidden_neurons) #with normal
    distribution
        self.h_bias = np.zeros(self.hidden_neurons)

        # weights and bias in our output
        self.out_weights = 0.01 + np.random.normal(self.hidden_neurons, self.categories)
        self.out_bias = np.zeros(self.categories)

    #Sigmoid activation function
    def sigmoid(self, x):
        return 1/(1 + np.exp(-x))

    #Tanh activation function
    def tanh(self, x):
      return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))

    #ReLU activation function
    def relu(self, x):
```

```
        return np.maximum(0,x)

    def cost_MSE(self y_h):
        n = np.size(y)
        C = 0
        for i in range(n):
            C += (self.Y_full[i] - y_h[i])**2
        return 1/n * C

    def ff(self):
        #Feed forward for network saved globaly in class
        self.z_hidden = np.matmul(self.X_full, self.h_weights) + self.h_bias

        activation_hidden = self.activation_func_hidden(self.z_hidden)

        self.z_out = np.matmul(activation_hidden, self.out_weights) + self.out_bias
        self.a_expect = self.activation_func_out(self.z_out)
        self.probability = self.a_expect/np.sum(self.a_expect, axis=1, keepdim=True)


    def ff_out(self):
        #feed forward output saved localy in function
        z_hidden = np.matmul(X, self.h_weights) + self.h_bias

        activation_hidden = self.activation_func_hidden(z_hidden)

        z_out = np.matmul(activation_hidden, self.out_weights) + self.out_bias
        a_expect = self.activation_func_out(z_out)
        probability = a_expect/np.sum(a_expect, axis=1, keepdim=True)
        return probability

    def backprop(self):
        self.error_in_out = self.probability - self.Y_full

        self.error_in_hidden = np.matmul(self.error_in_out, self.out_weights.T) * self.
    activation_hidden*(1-self.activation_hidden)

        self.grad_weight_out = np.matmul(self.activation_hidden.T, self.error_in_hidden)
        self.grad_bias_out = np.sum(self.error_in_out, axis=0)


        self.grad_weight_hidden = np.matmul(self.X_full.T, self.error_in_hidden)
        self.grad_bias_hidden = np.sum(self.error_in_hidden, axis=0)

        if self.lmbda > 0:
            self.grad_weight_out += self.lmbda * self.out_weights
            self.grad_weight_hidden += self.lmbda * self.h_weights

        self.out_weights -= self.eta*self.grad_weight_out
        self.out_bias -= self.eta*self.grad_bias_out

        self.h_weights -= self.eta*self.grad_weight_hidden
        self.h_bias -= self.eta*self.grad_bias_hidden

    def predict(self, X):
        return np.argmax(self.ff_out(X), axis=1)

    def pred_prob(self, X):
        return self.ff_out(X)

    def train_function(self):
        indec = np.arange(self.inputs)

        for i in range(self.epochs):
            for l in range(self.iter):
                data_points=np.random.choice(indec, size=self.batch_sz, replace=False)

                self.X_full = self.X_full[data_points]
                self.Y_full = self.Y_full[data_points]

                self.ff()
                self.backprop()
```

A.3 FFNN final code

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


#define Layers
"""Initializing for weights an biases for flexible number of layers with number of imputs and amount
     of neurons or hidden layers."""
class Layer():
    def __init__(self, n_features, neurons, lmbd = 0):
        # Initialize weights and biases
        self.n_features = n_features
        self.neurons = neurons
        self.weights = 0.01 * np.random.randn(self.n_features, self.neurons) #with normal
    distribution
        self.bias = np.zeros((1,self.neurons))

        # Set strength of regularization, the regularizer should be greater than or equal to 0
        self.weight_regularizer_l2 = lmbd

    def forward(self, inputs, train):
        self.input = inputs

        #Calculate output value from previous layer's inputs, weights and biases
        self.out = np.dot(inputs, self.weights) + self.bias

    def back(self, d_val):
        # gradients on weights and biases
        self.grad_weights = np.dot(self.input.T, d_val)
        self.grad_bias = np.sum(d_val, axis=0, keepdims=True)

        #L2 regularization on weights
        if self.weight_regularizer_l2 > 0:
            self.grad_weights += 2 * self.weight_regularizer_l2 * self.weights

        #Gradient on inputs
        self.grad_input = np.dot(d_val, self.weights.T)

class Layer_Input:
    def forward(self, inputs, train):
        self.out = inputs


'''Define activation function'''
class Activ_Sigmoid():
    def forward(self, inputs, train):
        self.input = inputs

        self.out =  1/(1 + np.exp(-inputs))
    def back(self, d_val):
        self.grad_input = d_val * (1 - self.out) * self.out

    def predict(self, out):
        return (out > 0.5) * 1

class Activ_Leaky_ReLU():
    def forward(self, inputs, train):
        self.input = inputs

        self.out = np.maximum(0.01 * inputs, inputs)

    def back(self, d_val):

        self.grad_input = d_val.copy()

        #gradient will be 0.01 when inputs are negative
        self.grad_input[self.input <= 0] = 0.01

```

```
    def predict(self, out):
        return out

class Activ_ReLU():
    def forward(self, inputs, train):
        self.input = inputs

        self.out = np.maximum(0, inputs)

    def back(self, d_val):

        self.grad_input = d_val.copy()

        #gradient will be 0 when inputs are negative
        self.grad_input[self.input <= 0] = 0

    def predict(self, out):
        return out

class Activ_Linear():
    def forward(self, inputs, train):
        self.input = inputs
        self.out = inputs

    def back(self, d_val):
        self.grad_input = d_val.copy()

    def predict(self, out):
        return out

class Activ_tanh():
    def forward(self, inputs, train):
        self.input = inputs

        self.out  = (np.exp(inputs) - np.exp(-inputs)) / (np.exp(inputs) + np.exp(-inputs))

    def back(self, d_val):
        self.grad_input = d_val.copy()

        self.grad_input = 1-self.out**2

    def predict(self, out):
        return out

class Activ_Softmax():

    def forward(self, inputs, train):
        self.input = inputs

        # Get unnormalized probabilities
        exp_val = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))

        # Normalize them for each sample
        probabilities = exp_val / np.sum(exp_val, axis=1, keepdims=True)

        self.out = probabilities

    def back(self, d_val):

        # Create uninitialized array
        self.grad_input = np.empty_like(d_val)

        # Enumerate outputs and gradients
        for index, (single_out, single_d_val) in enumerate(zip(self.out, d_val)):
            # Flatten output array
            single_out = single_out.reshape(-1, 1)
            # Calculate Jacobian matrix of the output
            jacobian_matrix = np.diagflat(single_out) - np.dot(single_out, single_out.T)
            # Calculate sample-wise gradient
            # and add it to the array of sample gradients
            self.grad_input[index] = np.dot(jacobian_matrix, single_d_val)

    def predict(self, out):
```

```
        return np.argmax(out, axis=1)

'''Define Optimizer'''
class Optim_SGD():

    def __init__(self, momentum = 0, decay = 0, lr = 1):
        self.momentum = momentum
        self.lr = lr
        self.decay = decay

        self.cur_lr = lr
        self.iter = 0

    def pre_up_par(self):
        if self.decay:
            self.cur_lr = self.lr * (1 / (1 + self.decay * self.iter))

    def up_par(self, layer):
        if self.momentum:
            #If layer dont consist of momentum we create them
            if not hasattr(layer, "weight_momentum"):
                layer.weight_momentum = np.zeros_like(layer.weights)
                layer.bias_momentum = np.zeros_like(layer.bias)

            #Creating weight updates
            weight_up = self.momentum * layer.weight_momentum - self.cur_lr * layer.grad_weights
            layer.weight_momentum = weight_up

            #Creating bias updates
            bias_up = self.momentum * layer.bias_momentum - self.cur_lr * layer.grad_bias
            layer.bias_momentum = bias_up

        else:
            weight_up = -self.cur_lr * layer.grad_weights

            bias_up = -self.cur_lr * layer.grad_bias


        layer.weights += weight_up
        layer.bias += bias_up

    def post_up_par(self):
        self.iter += 1

'''Define Loss'''
class Loss:

    def reg_loss(self):
        #initialize regularization loss
        reg_loss = 0

        #calculate regularization loss for each the training layer
        for layer in self.training_layer:
            # L2 regularization - weights
            if layer.weight_regularizer_l2 > 0:
                reg_loss += layer.weight_regularizer_l2 * np.sum(layer.weights * layer.weights)

        return reg_loss

    def remember_training_layer(self, training_layer):
        self.training_layer = training_layer

    #calculate losses from data and regularization with model output and true values
    def cal(self, out, y, *, regularization=False):
        #loss from sample
        samp_loss = self.forward(out, y)
        #mean loss
        loss_dat = np.mean(samp_loss)

        if not regularization:
            return loss_dat

        return loss_dat, self.reg_loss()
```

```
215
216
217 # Categorical Cross-entropy loss
218 class Loss_CC(Loss):
219
220     def forward(self, predict, y_real):
221         #size of each batch
222         sample = len(predict)
223
224         # Clip data to avoid denominator of 0
225         predict_clip = np.clip(predict, 1e-8, 1 - 1e-8)
226
227         #Probabilities for target values of categorical labels
228         if len(y_real.shape) == 1:
229             cc = predict_clip[range(sample), y_real]
230
231         elif len(y_real.shape) == 2:
232             cc = np.sum(predict_clip * y_real, axis = 1)
233
234         nll = -np.log(cc)
235         return nll
236
237     def back(self, d_val, y_real):
238
239         sample = len(d_val)
240         lab = len(d_val[0])
241
242         if len(y_real.shape) == 1:
243             y_real = np.eye(lab)[y_real]
244
245         #Claculate and normalize
246         self.grad_input = (-y_real/d_val)/sample
247
248 class Loss_MSE(Loss):
249
250     def forwar(self, predict, y_real):
251         samp_loss = np.mean((y_real - predict)**2, axis=-1)
252
253         return samp_loss
254
255     def backward(self, d_val, y_real):
256         sample = len(d_val)
257
258         out = len(d_val[0])
259
260         #Claculate and normalize
261         self.grad_input = (-2 * (y_real - d_val) / out) / sample
262
263 '''Define Accuracy'''
264 class Accuracy:
265
266     def cal(self, predict, y):
267         # Get comparison results
268         comparisons = self.compare(predict, y)
269         acc = np.mean(comparisons)
270
271         return acc
272
273     def cal_accum(self):
274
275         acc = self.accum_sum / self.accum_count
276
277         return acc
278
279     # Reset variables for accumulated accuracy
280     def reset_var(self):
281
282         self.accum_sum = 0
283         self.accum_count = 0
284
285
286 class Accuracy_Classification(Accuracy):
287
```

```
    def __init__(self, *, binary=False):
        # Binary mode?
        self.binary = binary

    def init(self, y):
        pass

    # Compares predictions to the ground truth values
    def compare(self, predict , y):

        if not self.binary and len(y.shape) == 2:
            y = np.argmax(y, axis = 1)

        return predict == y


class Accuracy_Regression(Accuracy):

    def __init__(self, y):
        self.precision = None

    def init(self, y, reinit=False):
        if self.precision is None or reinit:
            self.precision = np.std(y) / 250

    # Compares predictions to the ground truth values
    def compare(self, predict, y):
        return np.absolute(predict - y) < self.precision

'''Define Model'''
class Method():

    def __init__(self):
        self.layer = []
        self.train_acc = None
        self.test_acc = None

    def add_to_list(self, layer):
        self.layer.append(layer)

    def set_param(self, *, loss, accuracy, optimiz):
        self.optimiz = optimiz
        self.accuracy = accuracy
        self.loss = loss

    def finall(self):

        self.layer_inp = Layer_Input()
        layer_iter = len(self.layer)
        self.tlayer = []

        for i in range(layer_iter):

            if i == 0:
                self.layer[i].prev = self.layer_inp
                self.layer[i].next = self.layer[i+1]

            elif i < layer_iter - 1:
                self.layer[i].prev = self.layer[i-1]
                self.layer[i].next = self.layer[i+1]

            else:
                self.layer[i].prev = self.layer[i-1]
                self.layer[i].next = self.loss
                self.activ_out = self.layer[i]

            if hasattr(self.layer[i], "weights"):
                self.tlayer.append(self.layer[i])

        # Update loss object with trainable layers
        self.loss.remember_training_layer(self.tlayer)


```

```
    def train(self, X, y, *, n_epoc = 1, validation_data = None, print_epoch = False):

        self.accuracy.init(y)

        for epochs in range(1, n_epoc+1):

            out = self.forward(X, train=True)

            loss_dat, reg_loss = self.loss.cal(out, y, regularization = True)
            loss = loss_dat + reg_loss

            predict = self.activ_out.predict(out) #??
            accuracy = self.accuracy.cal(predict, y)

            self.back(out, y)

            self.optimiz.pre_up_par()
            for layer in self.tlayer:
                self.optimiz.up_par(layer)
            self.optimiz.post_up_par()

            """
            if print_epoch:
                print(f'n_epoch: {epochs}, ' +
                      f'accuracy: {accuracy:.3f}, ' +
                      f'loss: {loss:.3f} (' +
                      f'loss in data: {loss_dat:.3f}, ' +
                      f'loss in reglarization: {reg_loss:.3f}), ' +
                      f'learningrate: {self.optimiz.cur_lr}')
            """

            if epochs == n_epoc:
                self.train_acc = accuracy


        if validation_data is not None:

            X_val, y_val = validation_data

            out = self.forward(X_val, train=False)

            loss = self.loss.cal(out, y_val)

            predict = self.activ_out.predict(out)

            accuracy = self.accuracy.cal(predict, y_val)

            self.test_acc = accuracy

            if print_epoch:
                print(f'validation, ' +
                      f'acc: {accuracy:.3f}, ' +
                      f'loss: {loss:.3f}')

        return self.train_acc, self.test_acc


    def forward(self, X, train):

        self.layer_inp.forward(X, train)

        for layer in self.layer:
            layer.forward(layer.prev.out, train)

        return layer.out

    def back(self, out, y):

        self.loss.back(out, y)

        for layer in reversed(self.layer):
            layer.back(layer.next.grad_input)

```

```
'''Test Wisconsin Cancer Data'''
data = load_breast_cancer()
X = data['data']
y = data['target']

#Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

#Feature Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

n_neurons = 100
train_accuracy = []
test_accurary = []
for n in range(n_neurons):

    # Instantiate the model
    model = Method()
    # Add layers
    model.add_to_list(Layer(30, n, lmbd=5e-4))
    model.add_to_list(Activ_ReLU())
    model.add_to_list(Layer(n, 2))
    model.add_to_list(Activ_Softmax())

    # Set loss, optimizer and accuracy objects
    model.set_param(
        loss=Loss_CC(),
        optimiz=Optim_SGD(lr=0.05, decay=5e-5, momentum=0.9),
        accuracy=Accuracy_Classification()
    )

    # Finalize the model
    model.finall()

    # Train the model
    train_acc, test_acc = model.train(X_train, y_train, validation_data=(X_test, y_test), n_epoc
    =200, print_epoch = False)
    train_accuracy.append(train_acc)
    test_accurary.append(test_acc)

x = range(n_neurons)
fig = plt.figure()
plt.plot(x, train_accuracy, label = "train_accuracy")
plt.plot(x, test_accurary, label = "test_accurary")
fig.suptitle('Accuracy of train and test dataset with different number of neurons in the hidden
    layer')
plt.xlabel('Number of neurons')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('../Results/acc_n_nruons_NN.png')
```

A.4 Breast cancer data implementation code

```
from NN import *
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import numba as nb
import timeit

# ensure the same random numbers appear every time
np.random.seed(100)

# load breast cancer data
data = load_breast_cancer()
X = data['data']
y = data['target']
```

```
18
19 #@nb.jit(nopython=True)
20 def run_NN(X, y, active = "ReLU"):
21
22     sns.set()
23     #train test split
24     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
25
26     #Feature Scaling
27     sc = StandardScaler()
28     X_train = sc.fit_transform(X_train)
29     X_test = sc.transform(X_test)
30
31     #set up learning rate eta and regularization lambda for grid search
32     lr = np.logspace(-5, 1, 7)
33     lmbd = np.logspace(-5, 1, 7)
34
35     train_accuracy = np.zeros((len(lr), len(lmbd)))
36     test_accuracy = np.zeros((len(lr), len(lmbd)))
37
38     # grid search
39     for i, eta in enumerate(lr):
40         for j, lm in enumerate(lmbd):
41             # Instantiate the model
42             model = Method()
43
44             if active == "ReLU":
45                 # Add layers ReLU
46                 model.add_to_list(Layer(30, 64, lmbd=lm))
47                 model.add_to_list(Activ_ReLU())
48                 model.add_to_list(Layer(64, 64, lmbd=lm))
49                 model.add_to_list(Activ_ReLU())
50                 model.add_to_list(Layer(64, 2))
51                 model.add_to_list(Activ_Softmax())
52
53             elif active == "Sigmoid":
54                 # Add layers Sigmoid
55                 model.add_to_list(Layer(30, 64, lmbd=lm))
56                 model.add_to_list(Activ_Sigmoid())
57                 model.add_to_list(Layer(64, 64, lmbd=lm))
58                 model.add_to_list(Activ_Sigmoid())
59                 model.add_to_list(Layer(64, 2))
60                 model.add_to_list(Activ_Softmax())
61
62             elif active == "Leaky_ReLU":
63                 # Add layers Leaky_ReLU
64                 model.add_to_list(Layer(30, 64, lmbd=lm))
65                 model.add_to_list(Activ_Leaky_ReLU())
66                 model.add_to_list(Layer(64, 64, lmbd=lm))
67                 model.add_to_list(Activ_Leaky_ReLU())
68                 model.add_to_list(Layer(64, 2))
69                 model.add_to_list(Activ_Softmax())
70
71             elif active == "Linear":
72                 # Add layers Linear
73                 model.add_to_list(Layer(30, 64, lmbd=lm))
74                 model.add_to_list(Activ_Linear())
75                 model.add_to_list(Layer(64, 64, lmbd=lm))
76                 model.add_to_list(Activ_Linear())
77                 model.add_to_list(Layer(64, 2))
78                 model.add_to_list(Activ_Softmax())
79
80             elif active == "Tanh":
81                 # Add layers tanh
82                 model.add_to_list(Layer(30, 64, lmbd=lm))
83                 model.add_to_list(Activ_tanh())
84                 model.add_to_list(Layer(64, 64, lmbd=lm))
85                 model.add_to_list(Activ_tanh())
86                 model.add_to_list(Layer(64, 2))
87                 model.add_to_list(Activ_Softmax())
88
89             elif active == "sigmoid_tanh":
90                 # Add layers sigmoid and tanh
```

```
                model.add_to_list(Layer(30, 64, lmbd=lm))
                model.add_to_list(Activ_Sigmoid())
                model.add_to_list(Layer(64, 64, lmbd=lm))
                model.add_to_list(Activ_Sigmoid())
                model.add_to_list(Layer(64, 2))
                model.add_to_list(Activ_tanh())

            elif active == "sigmoid_ReLU":
                # Add layers sigmoid and ReLU
                model.add_to_list(Layer(30, 64, lmbd=lm))
                model.add_to_list(Activ_Sigmoid())
                model.add_to_list(Layer(64, 64, lmbd=lm))
                model.add_to_list(Activ_Sigmoid())
                model.add_to_list(Layer(64, 2))
                model.add_to_list(Activ_ReLU())


            # Set loss, optimizer and accuracy objects
            model.set_param(
                loss=Loss_CC(),
                optimiz=Optim_SGD(lr=eta, decay=5e-5, momentum=0.9),
                accuracy=Accuracy_Classification()
            )

            # Finalize the model
            model.finall()

            # Train the model
            train_acc, test_acc = model.train(X_train, y_train, validation_data=(X_test, y_test),
    n_epoc=200,print_epoch = False)

            train_accuracy[i][j] = train_acc
            test_accuracy[i][j] = test_acc


    fig, ax = plt.subplots(figsize = (10, 10))
    sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
    ax.set_title("Training Accuracy of our own NN with activ func " + active )
    ax.set_ylabel("$\eta$")
    ax.set_xlabel("$\lambda$")
    plt.savefig('../Results/eta_lmd_train_acc_' + active + ".png")

    fig, ax = plt.subplots(figsize = (10, 10))
    sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
    ax.set_title("Test Accuracy of our own NN with activ func " + active)
    ax.set_ylabel("$\eta$")
    ax.set_xlabel("$\lambda$")
    plt.savefig('../Results/eta_lmd_test_acc_' + active + ".png")

    return train_accuracy, test_accuracy


start = timeit.default_timer()
run_NN(X, y, active = "Tanh")
stop = timeit.default_timer()
print('Time: ', stop - start)




"""
fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy of our own NN")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.savefig('../Results/eta_lmd_train_acc.png')
fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(test_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Test Accuracy of our own NN")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.savefig('../Results/eta_lmd_test_acc.png')
```

```
"""
```

A.5 Keras NN Cancer data code

```
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Sequential      #This allows appending layers to existing models
from tensorflow.keras.layers import Dense           #This allows defining the characteristics of a
    particular layer
from tensorflow.keras import optimizers             #This allows using whichever optimiser we want (
    sgd,adam,RMSprop)
from tensorflow.keras import regularizers           #This allows using whichever regularizer we want
     (l1,l2,l1_l2)
from tensorflow.keras.utils import to_categorical   #This allows using categorical cross entropy as
    the cost function

from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
import numpy as np
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

#Kera neural network was runned on conda environment
#The figure was downloaded and put in our github

# one-hot in numpy
def to_categorical_numpy(integer_vector):
    n_inputs = len(integer_vector)
    n_categories = np.max(integer_vector) + 1
    onehot_vector = np.zeros((n_inputs, n_categories))
    onehot_vector[range(n_inputs), integer_vector] = 1

    return onehot_vector

def create_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_categories, eta, lmbd):
    model = Sequential()
    model.add(Dense(n_neurons_layer1, activation='sigmoid', kernel_regularizer=regularizers.l2(lmbd)
    ))
    model.add(Dense(n_neurons_layer2, activation='sigmoid', kernel_regularizer=regularizers.l2(lmbd)
    ))
    model.add(Dense(n_categories, activation='softmax'))

    sgd = optimizers.SGD(lr=eta)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model


data = load_breast_cancer()
X = data['data']
y = data['target']
y = to_categorical(y)

X_train, X_test, Y_train, Y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
#Feature Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

DNN_keras = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        DNN = create_neural_network_keras(n_neurons_layer1, n_neurons_layer2, n_categories,
                                          eta=eta, lmbd=lmbd)
        DNN.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)
        scores = DNN.evaluate(X_test, Y_test)

        DNN_keras[i][j] = DNN

        print("Learning rate = ", eta)
```

```
        print("Lambda = ", lmbd)
        print("Test accuracy: %.3f" % scores[1])
        print()
```

B Appendix B REFERENCES:

References

[1] Lectures and notes from lectures, https://compphysics.github.io/MachineLearning/doc/web/course.html

[2] Wikipedia. (2021, November 11). Artificial Neural Network. Wikipedia. Retrieved November 19, 2021, from https://en.wikipedia.org/wiki/Artificial$_n eural_n etwork$

[3] Kakaraparthi, V. (2019, October 29). Activation functions in neural networks. Medium. Retrieved November 20, 2021, from https://prateekvishnu.medium.com/activation-functions-in-neural-networks-bf5c542d5fec.

[4] Kinsley, H., amp; Kukieła, D. (2020). Neural Networks From Scratch (Vol. 1). Harrison Kinsley.

[5] Bushaev, V. (2017, December 5). Stochastic gradient descent with momentum. Medium. Retrieved November 20, 2021, from https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d.