Project 2: Feed Forward Neural Network

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

Adele Zaini*

(Dated: November 20, 2021)

I. ABSTRACT

II. INTRODUCTION

The neural network, also known as artificial neural network, is a subset of machine learning techniques and is at the heart of deep learning algorithms. Its name and structure is inspired by the human brain, mimicking the way that biological neurons interact to one another. The very powerful point of this technique is that it can explore new ways of analysing and interpreting data, that are limited by the classical methods, such as regression ones.

In this report, we will implement a Feed Forward Neural Network with backpropagation and Stochastic Gradient Descent (SGD) in Python. We will firstly concentrate on a preliminary step for the implementation that is the optimization technique of SGD. Afterwards the focus is on the implementation of the Neural Network class both in the Regression and Classification cases, while exploring the optimization of different hyperparameters and network architectures. Comparisons with Linear and Logistic Regression will be performed to check the results. The implementation of the Logistic Regression is part of this work.

The report is organized as follows. After this introduction, the methods are explored considering the background theory and the algorithms implemented. The following section will present the results of the work, while commenting and discussing them. The final section is to summarize the analysis into the conclusions and to have an overview on future perspective.

III. METHODS

A. Theory and algorithms

The basic idea of Neural Network lies on building a system of interconnected units, called *neurons*, divided in multiple layers, that broadcast signals throughout the neural system. The structure is composed of an input layer, one or more hidden layers, with different number of neurons in them, and the output layer. The broadcasting of the signal happens thanks to *activation functions* that elaborate the input signals for each neuron into the ouput ones. They are coupled with weights and biases associated to each neuron so that the output is more or less relevant when broadcasted to the next layer. This process starts from the input layer and it runs thoughtout each neuron of each layer, until resulting into an output, that is the model prediction (i.e.*Feed Forward Neural Network*). The next step is to *train* the network. Given already the expected *true* output, it is compared to the model prediction thanks to a *cost function* that evaluates the performance of the model. The aim is to minimize this value and using algorithms, such as the *backpropagation* algorithm, is possibile to optimize the parameters (i.e. weights and biases) to have the closest result to the expected one. This last step is extremely important because then the network is able to *predict* the results given any other input data. The overall performance of the network is evaluated thanks to the cost function on a test dataset.

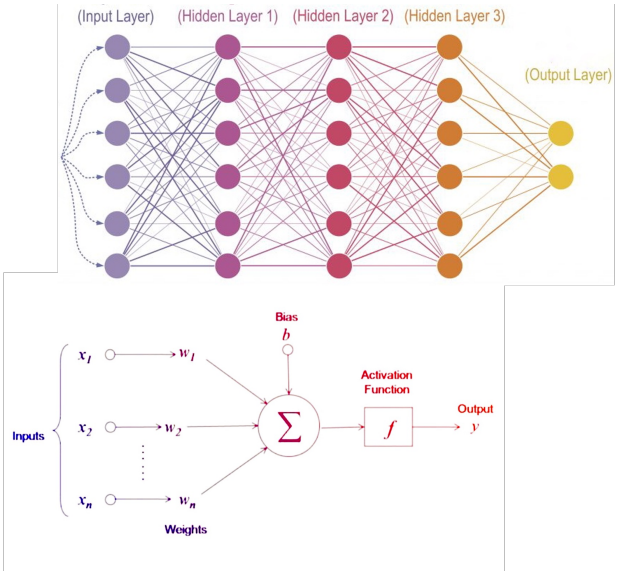

FIG. 1. Examples of Neural Network architecture, with both several neurons divided into the hidden layers and with a focus on one single neuron and its schematic operation.

Feed Forward Neural Network – The key idea of this type of Neural Network is that the information moves in only one direction: forward through the lay-

* GitHub profile: https://github.com/adelezaini/

ers. As depicted in Figure 1, each single neuron has $\{x_i\}$ as input data, which is linearly combined with the respective weights $\{w_i\}$ and biases $\{b_i\}$ (it can be a single value for all the neurons of the layer or different values): $z = \sum_i (w_i x_i + b_i)$.

The output y is produced via the activation function f as follows:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z), \tag{1}$$

In a dense FFNN with multiple neurons and hidden layers, the inputs $\{x_i\}$ are the outputs of the neurons in the preceding layer, and the output y is one the input for the next layer. At the end, the equation results to be:

$$y_i^{L+1} = f^{L+1}\left[\sum_{j=1}^{N_L} w_{ij}^3 f^L\left(\sum_{k=1}^{N_{L-1}} w_{jk}^{L-1}(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots + b_k^2 + b_1^3\right.\right. \tag{14}$$

where L is the number of hidden layers.

Back propagation – The back propagation is the core algorithm behind how neural networks learn. After a first forward run thoughout the network, to have the first attempt of a prediction, the network needs to be trained. In other words, it means that the parameters (i.e. weights and biases) need to be optimized to minimize the cost function, which quantifies the difference between the model output and the expected output values.

The idea behind this algorithm is to go back thoughout the network, layer by layer, using the *chain rule* to find the optimal parameters. This algorithm can be coupled with different optimization techniques that evaluate the partial derivatives for the chain rule.

The starting point is at the end of the network: evaluating the total error $\hat{\delta}^L$ of the output layer by computing all δ_j^L for each outputs:

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

Then the back propagate error is computed for each $l = L-1, L-2, \dots, 2$ layer as

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

Finally, the weights and the biases are updated using an optimization tecnique for each $l = L-1, L-2, \dots, 2$, such as the gradie descent algorithm:

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

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

where η is the learning rate.

The backpropagation is an algorithm for determining how a single training example would like to nudge the weights and biases, in terms of what relative proportions to those changes cause the most rapid decrease to the cost function. But this algorithm needs to be run over all the examples (i.e. datapoints for the Regression case and digits/images for the Classification case), this is why it is convenient to use *Stochastic Gradient Descent* with mini-batches with an outer loop that steps through multiple epochs of training.

Activation functions – The choice of the activation functions is a key element in the performance of the network. The classical choice is the sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}},$$

but in the past it has been shown its limits for large input values (i.e. *vanishing gradient problem* in applying the back propagation algorithm). Other choices can be: - *hyperbolic tangen*t: $f(x) = \tanh(x)$ - *ReLU (Rectified exponential Linear Unit)*: - *eLU (...)*: - ...

For the output layer the most common activation function is the *softmax*: ... While in the Regression case, no activation function is needed for the output layer.

Cost function and metrics – The **cost function** (or loss function) is a method of evaluating how well the neural network fits the dataset or, in other words, it quantifies the error between the predicted output values and the expected ones. It is used to in the training process to optimize the paramaters in order the get to the minimum of this function. Depending on the problem, the cost function can have different expressions. The most common choice for the regression and the classification case are the Mean Squared Error (MSE) and the Cross-Entropy loss (CE) respectively.

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

where $\hat{y}_i$ is the predicted value of the $i-th$ sample, y_i is the corresponding true value and n is the total number of datapoints.

$$CE = -\sum_{c=1}^{M}(y_c\ log(p_c)$$

where y_c is the binary indicator (0 or 1) if the class label c is the correct classification label, $p_c =$ is the predict

probability of the class c (i.e. output value before filtered by the output activation function).

The **metrics** is a function that evaluates the overall performance of the neural network. It's a similar concept to the cost function, but while the cost function is used on the training dataset to optmize the neural network parameters (i.e. it quantifies the *in-sample error*), the metrics is used on test datasets at the end of the training process (i.e. it quantifies the *out-of-sample error*). There are different expressions for the metrics too. For the regression and the classification cases, the MSE and the Accurancy score are the most in use.

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

Here t_i represents the target, y_i the outputs of the FFNN code, n is the number of targets t_i and I is the indicator function: 1 if $t_i = y_i$ and 0 otherwise.

Gradient Descent and Stochastic Gradient Descent – The Gradient Descent (GD) is an optimization technique and the most common way to train neural networks. It is an algorithm to minimize an objective function (i.e. cost function $C(\beta)$) parameterized by model's parameters β by updating the parameters in the opposite direction of the gradient of the objective function $\boldsymbol{\nabla}_\beta C(\beta)$ to the parameters. It is based on the *Newton-Raphson's method*, an approximation to the first order of the Taylor expansion, where the learning rate η (i.e. inverse of the second derivative) determines the size of the steps we take to reach a (local) minimum. It is possible to demonstrate that in the matrix notation the optimal value of the learning rate is $\hat{\eta} = \frac{1}{\lambda_{max}}$, where λ_{max} is the maximum eigenvalue of the Hessian matrix, so that it grants convergence and efficiency in the iterations. In other words, iterating this "parameters update", we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley (Figure 2). Nevertheless, the Gradient Descent algorithm

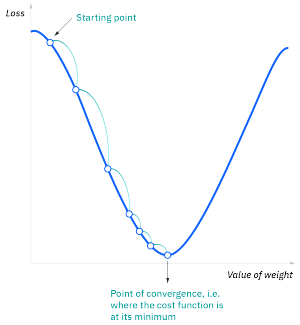

FIG. 2. Intuitive idea of the Gradient Descent optimization when updating the parameters.

represents limitations connected to its characteristic of being very precise. These concerns its low efficiency and its high dependence on initial conditions that leads to be trapped in local minima. In order to improve this algorithm, "randomness" needs to be included and the way is to implement the Stochastic Gradient Descent (SGD). While in the GD the gradient is evaluated from the entire dataset, in the SGD it is calculated from a randomly selected subset of the data, called *mini-batch*. The *mini-batches* are embedded in an iteration loop over the number of *epochs*.

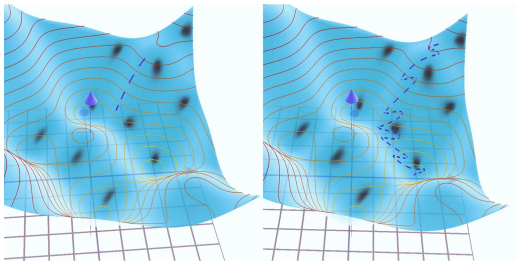

FIG. 3. Difference of optimization paths over a two-dimensional surface: a) precise and inefficient Gradient Descent (left) and b) random and efficiency Stochastic Gradient Descent (right).

Optimizers – There are variants of the optimization algorithms, such as SGD with momentum, AdaGrad, RMSprop, ADAM.

1. *Gradient Descent*:

$$\beta \leftarrow \beta - \eta\, \boldsymbol{\nabla} C(\beta)$$

2. *Gradient Descent with momentum*:

$$v \leftarrow \gamma v + \eta\, \boldsymbol{\nabla} C(\beta)$$
$$\beta \leftarrow \beta - v$$

where γ is the *drag* parameter, commonly set to $\gamma = 0.9$.

3. *AdaGrad*:

$$r \leftarrow r + g \odot g$$
$$\beta \leftarrow \beta - \frac{\eta}{\sqrt{r+\delta}} \odot g$$

where $g = \boldsymbol{\nabla} C(\beta)$, $\odot$ is the *Hadamard product* or element-wise product and $\delta \sim 10^{-8}$ is a parameter to avoid division by 0.

4. *RMSprop*:

$$r \leftarrow \rho r + (1-\rho) g \odot g$$
$$\beta \leftarrow \beta - \frac{\eta}{\sqrt{r+\delta}} \odot g$$

where ρ is usually set to 0.9.

5. *ADAM*:

$$\hat{m} \leftarrow [\beta_1 m + (1-\beta_1) g](1-\beta_1)$$
$$\hat{v} \leftarrow [\beta_2 v + (1-\beta_2) g^2](1-\beta_2)$$
$$\beta \leftarrow \beta - \frac{\eta}{\sqrt{\hat{v}+\delta}} \odot \hat{m}$$

where $\beta_1 \sim 0.9$ and $\beta_2 \sim 0.999$.

[]**lerning rate**]

B. Code implementation

The algorithms previously explained have been implemented in a class called `NeuralNetwork`. When initializing the class, the architecture (i.e. list of neurons and activation functions for each hidden layer) and "static" parameters (e.g. learning rate, optimizer...) are set. The default architecture sees one hidden layer with two neurons and the *sigmoid* as activation function. If the number of hidden layers (i.e. lenght of the list of neurons) is higher than one and the activation list is composed by a single element, the class automatically set that function as the activation for each hidden layer. The activation function of the output layer is set to `None` in the regression case and `softmax` in the classification case. Exceptions are thrown if the input arguments do not meet the requirements (e.g. the activation function is not in the list of the implemented ones).

The core of the class is `fit()` method, which receives `X` and `Y` as arguments, initializes the weights and the biases with the normal distribution $N(0, n)$ (where n is the number of inputs to the neuron, number of neurons of the previous layer) and trains the network. It returns the network trained and ready to predict outputs from any input dataset (`predict()` method).

The following picture shows the class structure and sequential and embedded operations in a schematic way:

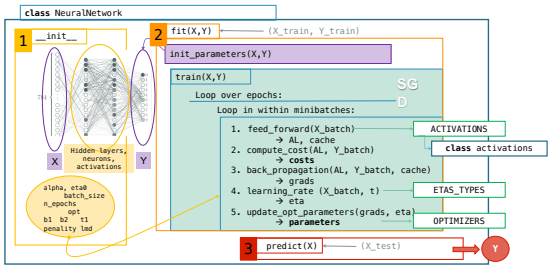

FIG. 4. Class `NeuralNetwork`: structure and methods.

The duties of the methods are:

- `__init__()`: initialize the NN in terms of architecture (layers, neurons, activations functions) of the network and all the parameters needed for the backpropagation and the optimization algorithms.
- `fit()`:fit the network with the given input X and output Y and initialize layers, parameters and train the network.
- `init_parameters()`: initialize the weights and the biases in the given architecture (i.e. [2,4,3,2] as list of number of neurons), using a normal distribution.
- `feed_forward()`: implements the Feed Forward algorithm
- `compute_cost()`: implement the cost function with optional regulation techniques 'l1' and 'l2'
- `cost_function()`: return the cost function specific to Regression or Classification case.
- `backpropagation()`: implement the backward propagation
- `learning_rate()`: update the learning rate when it is different than a given constant value. Different types of learning rate can be chosen.
- `update_opt_parameters()`: Update parameters using gradient descent algorithm with the given optimizer.
- `train()`: the core algorithm of the NN object: 1) Feed Forward to arrive to the output layer; 2) Backpropagation to evaluate all the gradients for each example/feature; 3) Update parameters (weights and biases) according to the optimizer (throughout the examples/features); 4) All wrapped up in the stochastic part (i.e. epochs and mini batches) of the SGD algorithm.
- `predict()`: predicting values with the Feed Forward algorithm, to be used after training. The output depends on the Regression or Classification case.
- `model_performance()`: implements the metrics specific for the Regression or Classification case.

This is class is an parent class of `class NN_Regression` and `class NN_Classifier`, which inherit all the methods, but the output activation function initialization, the cost function and the metrics implementations.

This class is coupled to another class: `class activations`. This is an abstract parent class, that defines two methods: `eval(X)` and `gradient(X)`. These methods are then implemented in each derivative class to evaluate the function result and the function gradient at a given `X` respectively. The following activations functions are implemented: `sigmoid`, `tanh`, `elu`, `relu`, `leaky_relu`, `softmax`. In order to use this class into the NeuralNetwork one, a dictionary was created to collect the derivative classes and the usage is the following:

```
ACTIVATIONS = {'sigmoid': sigmoid, 'tanh': tanh,
    'relu': relu, 'leaky_relu': leaky_relu,
    'elu': elu, 'softmax': softmax, None: None}
activations_list = ['sigmoid','relu']
    ...
act_func = ACTIVATIONS[activations_list[l]]
Al = act_func(Zl, alpha).eval()
```

For any further information, the GitHub repository (https://github.com/adelezaini/MachineLearning)

contains all the source code with a detailed documentation.

Neural Network Routine After describing the class itself, we will focus briefly on the steps needed when using neural networks to solve supervised learning problems. The routine is the following:

1. *Collect and pre-process data*: elaborate on the given input and output datasets in order to make them suitable for the Neural Network operations;
2. *Define model architecture, choose cost function and optimizer*: initialize the Neural Network overall structure.
3. *Initialize the model parameters*: weights and biases are initialized with a normal distribution;
4. *Train the model with the training dataset*: in a loop of a certain number of epochs and mini-batches:
   - Calculate current loss (forward propagation)
   - Calculate current gradient (backpropagation)
   - Update paramaters (gradient descent or another optimization algorithm)
5. *Make prediction with the test data*: after the paramters are optimized;
6. *Evaluate model performance and adjust hyperparameters*: analysis over the several paramters in order to find the minimum of the metrics of the model.

Linear Regression class In order to solve the first task and perform comparison with the Neural Network code, a class called `LinearRegression`, and the derivative `class OLSRegression, RidgeRegression` and `LassoRegression`, has been implemented from the code of project 1. Methods to evaluate the optimal β has been implemented for both the matrix-invertion evaluation and the Stocastic Gradient Descent algorithm, with the different optimizers previous exposed. Here is the list of the methods and their duties:

- `__init__()`: create the object given the X and y.
- `split()`: split the data into training and test subsets.
- `rescale()`: rescale the data using the `StardardScaler` of Scikit-Learn.
- `solver()`: regression equation, to be implemented in each derivative class.
- `gradient()`: gradient of the `solver`, to be implemented in each derivative class.
- `fit()`: fit the model and return beta-values, according to the `solver`.
- `fit_SK()`: fit the model and return beta-values, using Scikit-Learn.
- `fitGD()`: fit the model and return beta-values, using the Gradient Descent and calling the `gradient` method.
- `fitSGD()`: fit the model and return beta-values, using the Stochastic Gradient Descent and calling the `gradient` method.
- `predictSGD_BS()`: fit the model and return beta-values, using the Stochastic Gradient Descent and the bootstrap algorithm.
- `predict()`: predict y values given an external X. The methods of `predict_train()` and `predict_test()` are analogous but they work on the internal members.
- `rescaled_predict`: rescale y prediction to the original data range.
- `MSE_train()`, `MSE_test()`, `R2_train()`, `R2_test()`: evaluate the MSE or the R2 score on internal members.
- `Confidence_Interval()`: return the confidence interval of the beta-values.

A simple example of usage is:

```
model = OLSRegression(X,y)
model.split().rescale().fit()
mse = model.MSE_test()
```

All the source code can be found in the GitHub repository: https://github.com/adelezaini/MachineLearning in the directory `Projects/Project2`.

IV. RESULTS AND DISCUSSION

The work for this project is divided in different tasks. Task *a* focuses on the Stochastic Gradient Descent performance, while task *b, c, d* focus on exploring neural networks. Task *e* regards the implementation of the Logistic Regression code.

The datasets are the Franke Function and the *Boston Housing* for the regression case, and the *Wisconsin Breast Cancer* and the *MNIST digits* for the classification case.

A. Stochastic Gradient Descent

Perform an analysis of the results for OLS and Ridge regression as function of the chosen learning rates, the

number of mini-batches and epochs as well as algorithm for scaling the learning rate.

The dataset is generated by the Franke Function taken $n = 25$ for each dimensions (x, y) (total datapoints: $N = 25^2 = 625$) with an added stochastic noise normally distributed $N(0, 0.1)$. The design matrix X of a polynomial of `degree=5` together with z is the input to initialize the `OLSRegression` and `RidgeRegression` objects. Splitting into train and test datasets and rescaling with `StardardScaler(with_std=False)` from Sklearn have been performed before fitting the model with `n_epochs = 50, n_minibatches = 10` and default hyperparameters $\lambda = 1e-12$ and $\eta_0 = 1\lambda_{max}$ (λ_{max} is the maximum eigenvalue of the Hessian matrix of X). This first choice is reflected is quite poor MSE, that is equal to 2.1815 for both OLS and Ridge Regression, in the face of $MSE_{sk} = 0.0267$ given by the Sklearn model. These result is surely improvable with the optimization of the parameters.

The performance analysis starts with the optimization of the learning rate, evaluated with a *learning schedule* dependent on two parameters t_0 and t_1. Afterwards the numbers of minibatches and epochs are explored. At the end a cross-investigation of learning rate and λ gives the best combination of the Ridge case. In each step the optimal parameter found in the previous step is taken as input parameter for the next fitting.

From the results of this MSE analysis (Figures 5, 6, 7, 8) we can then conclude that this implementation of the SGD, choosen a *learning schedule* for the learning rate, shows the best performance for $\eta_0 = 10^{-5}$ $(t_0, t_1 = 5\ 10^{-3}, 500)$, $n_{minibatches} = 12$, $n_{epochs} = 150$ for OLS, while $\eta_0 = 5\ 10^{-2}$ $(t_0, t_1 = 25, 500)$, $n_{minibatches} = 50$, $n_{epochs} = 150$ for Ridge. These optimized parameters give a MSE in the order of magnitude minor of 0.1.

Figures 9 shows that the best combination of λ and η is rispectively 1.87 10^-4 and 4.33 10^4, giving an excellent $MSE = 0.049$.

Further comments can be done on the stochastic behaviour in the trends (Figure 7, Figure 8) (leveled by the rolling mean) when re running the code. A version of the fitting has been implemented with the bootstrap algorithm (`fitSGD_BS()`), but the results do not change that much and it's computational more expensive. The reason relies on the intrinsic stochastic character of the SGD algorithm.

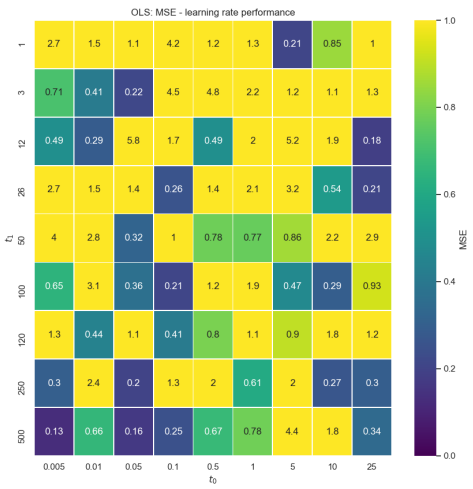

FIG. 5. **Task a – SGD**: MSE performance of the learning rate, choosing a *learning schedule* with parameters t_0 and t_1. Maximum MSE value set to 1. The optimal values are $t_0 = 5\ 10^{-3}$ and $t_1 = 500$, given an initial learning rate of $\eta_0 = 10^{-5}$.

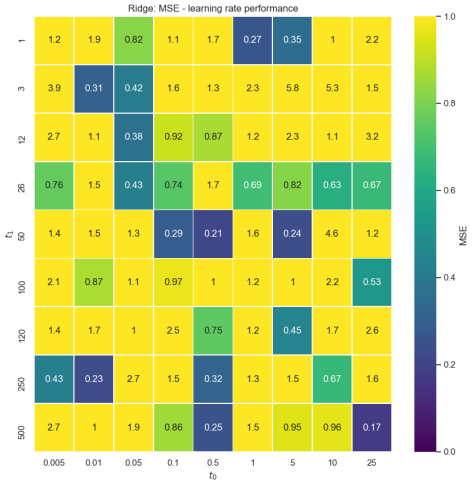

FIG. 6. **Task a – SGD**: MSE performance of the learning rate, choosing a *learning schedule* with parameters t_0 and t_1. Maximum MSE value set to 1. The optimal values are $t_0 = 25$ and $t_1 = 500$, given an initial learning rate of $\eta_0 = 5\ 10^{-2}$.

B. Neural Network code

Write an FFNN code for regression with a flexible number of hidden layers and nodes using the Sigmoid function as activation function for the hidden layers. Train your network and make an analysis of the regularization parameters and the learning rates employed to find the optimal MSE and R2 scores. Compare the results obtained with the Linear Regression code and your own Neural Network code.

Note: **The class `NeuralNetwork` has been implemented for both the Regression and Classification cases but because of a number of unexpected problems I had to face is currently not running**

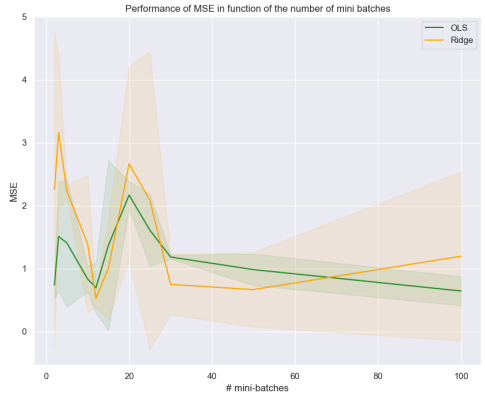

FIG. 7. **Task** a **– SGD**: MSE performance over the number of minibatches. A rolling mean has been implemented to better appreciate the trend.

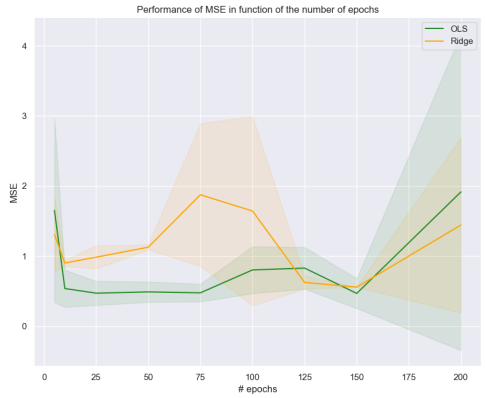

FIG. 8. **Task** a **– SGD**: MSE performance over the number of epochs. A rolling mean has been implemented to better appreciate the trend.

yet. The reason relies on the fact that I wished to implement a very versatile class, but I have been trapped myself into the complexity of my own code. Unfortunately this led me to spend several days (weeks) on elaborating this class, lacking on the rest of the project. In order not to be blocked by an progress iun the analysis, I'll present here the work I would have done if the code was running. The class credit is to Simone Mirabella, another student of the course, with whom I started structuring the code at the beginning. The class is basically the same as mine, but it runs. The implementation of my class is in the following link: https://github.com/adelezaini/MachineLearning/blob/nn/Projects/Project2/neuralnetwork.py. Any feedback is more than welcomed! I also apologise for lacking in the depth of the further

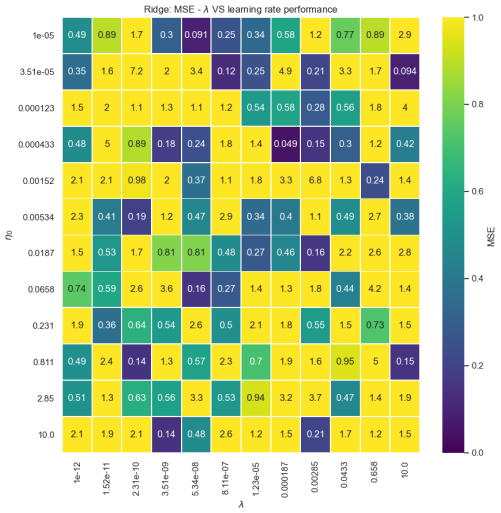

FIG. 9. **Task** a **– SGD**: MSE performance in a cross-investigation over λ and the learning rate η.

analysis but the time left was too little to explore more.

The dataset choosen for this task is the Boston Housing because it makes the Neural Network perform on a real case of Regression in Supervised Learning. Firstly the two most correlated features has been selected, the dataset has been splitted into train and test and they have been rescaled by the mean and the standard deviation.

	OLS	Ridge	NN_no_reg	NN_l2_reg
MSE_train	29.30	31.96	0.33	0.42
MSE_test	35.35	23.05	0.61	0.65
R2_test	0.57	0.63	0.39	0.35

TABLE I. **Task** b **– NN Regression**: Table summarizing the comparison between Linear and NeuralNetwork methods on the error analysis. The table refers to NN training of Figure 10 and Figure 11.

Figures 10 and 11, we can appreciated the optimization process of the neural network [**regulation comment**] The comparison with the the Linear Regression (Table I) suggests the excellent performance of the Neural Network, both with or without the regularization option, over the classic Linear Regression methods.

While training the network, the user can learn how to better set the number of hidden layers and number of neurons in each layer. Increasing the number of hidden layers does not increase the performance consistently, while the pattern of the number of neurons throughout the hidden layers seems more important. Starting from a number close to the input dimension and go progres-

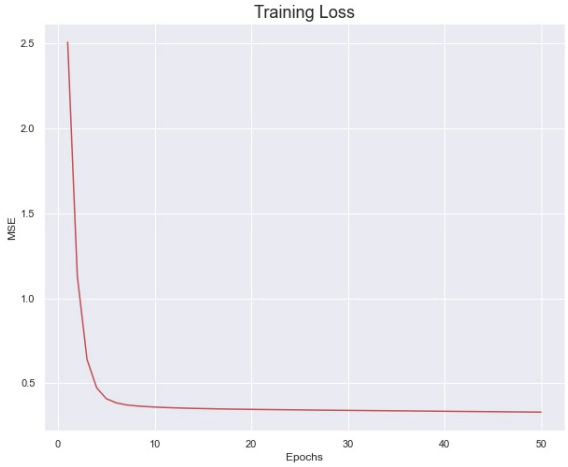

FIG. 10. **Task** b **– NN Regression**: Training of the NN with two layers of dimensions $[200, 50]$, $n_{epochs} = 50$, $n_{minibatches} = 32$ and $\eta = 0.001$. No regularisation has been considered.

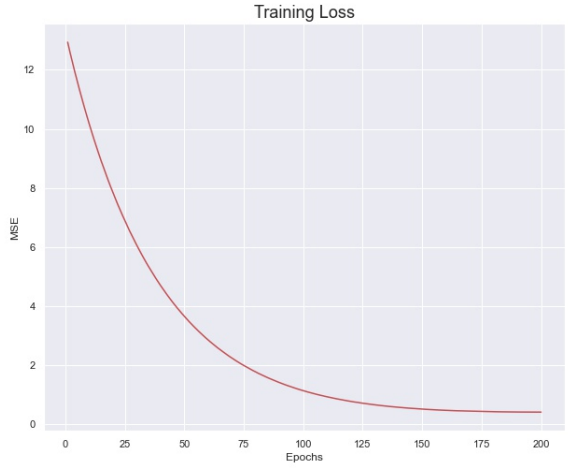

FIG. 11. **Task** b **– NN Regression**: Training of the NN with two layers of dimensions $[200, 50]$, $n_{epochs} = 200$, $n_{minibatches} = 32$ and $\eta = 0.01$. 'L2' regularisation has been considered with a $\lambda = 0.1$.

sively to the number of the outputs gives the best results in terms of performance.

Regarding the optimization of the hyperparameters, we can appreciate from Figures 12 and 13 that the neural network best performs with $\eta = 0.001$ and $\lambda \in [10^{-12}, 0.001]$.

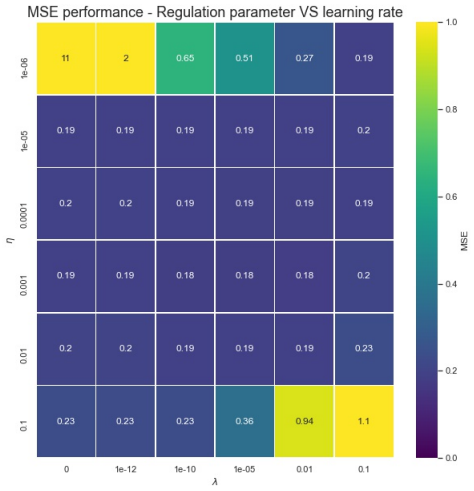

FIG. 12. **Task** b **– NN Regression**: Cross-investigation on MSE performance of the regularisation parameter λ and the learning rate η.

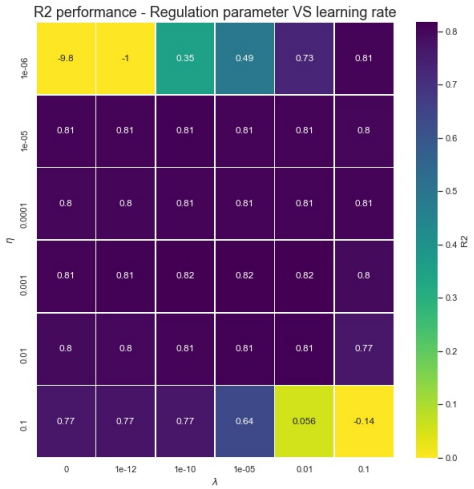

FIG. 13. **Task** b **– NN Regression**: Cross-investigation on R2 performance of the regularisation parameter λ and the learning rate η.

C. Testing different activation functions

Test different activation functions for the hidden layers. Try out the Sigmoid, the RELU and the Leaky RELU functions and discuss your results.

Figure 14 shows that the choice of the activation function can influence the efficiency at the beginning of the training but that the performance converges to be the same independently by the choosen activation function.

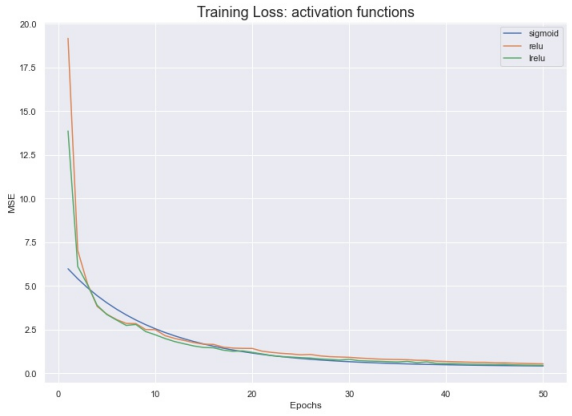

FIG. 14. **Task c – NN Regression**: Different activation functions of the hidden layers are explored in terms of MSE performance.

In this particular case no problem is shown, but from literature the choice of the activation function can influence the network's performance. When the gradients get smaller and smaller as the backpropagation algorithm progresses down to the first hidden layers. As a result, the GD update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is known in the literature as the vanishing gradients problem. In other cases, the opposite can happen, namely the the gradients can grow bigger and bigger. The result is that many of the layers get large updates of the weights the algorithm diverges. This is the exploding gradients problem. The ranking taken from the literature shows LeakyReLU performing better than ReLU, which performs better than the Sigmoid.

D. Classification analysis using neural networks

Change the code to perform also in the classification casr. Discuss your results and give a critical analysis of the various parameters, including hyper-parameters like the learning rates and the regularization parameter λ (as you did in Ridge Regression), various activation functions, number of hidden layers and nodes and activation functions.

E. Logistic Regression code

Compare the FFNN code with Logistic regression.

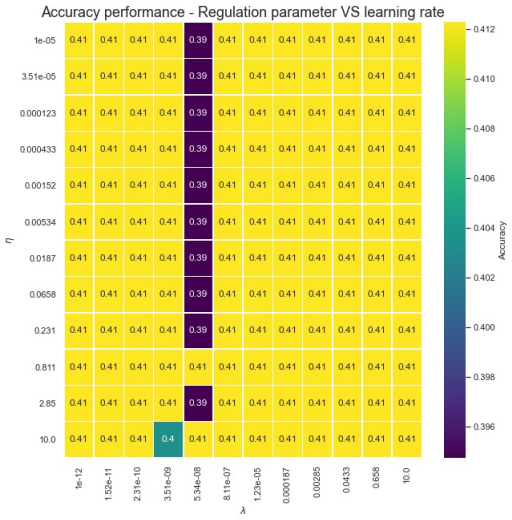

FIG. 15. **Task d – NN Classification**: Cross-investigation on Accurancy performance of the regularisation parameter λ and the learning rate η.

FIG. 16. **Task d – NN Classification**: Different activation functions of the hidden layers are explored in terms of training performance performance. At the end the Sigmoid and eLU function has an accurancy of 0.412, while ReLU of 0.096.

V. CONCLUSION

This last section covers task f of the project: Critical evaluation of the various algorithms.

In this report we explored the performance of the neural networks and related algorithms, compared to standard techniques as Linear and Logistic Regression. We can state that overall the neural network can lead to excellent results, when optimizing properly the parameters and the network architecture.

Most of the work behind this report has been focused

on the implementation of the project, compared to the analysis and optimization of the parameters for which little time was left. Yet some interesting results can be highlighted.

Starting from the SGD algorithm, the analysis shows the best performance when considering optimal values for the different parameters: $\eta_0 = 10^{-5}$, $n_{minibatches} = 12$, $n_{epochs} = 150$ for the OLS Regression, while $\eta_0 = 5\ 10^{-2}$, $n_{minibatches} = 50$, $n_{epochs} = 150$ for the Ridge Regression. Figures 9 shows that the best combination for the Ridge of λ and η is respectively $1.87\ 10^{-4}$ and $4.33\ 10^{4}$, giving an excellent $MSE = 0.049$. All the figures manifest the intrisic stochastic behaviour of the algorithm.

Regarding the neural network, we can appreciate its better performance over the classical method of Linear Regression (Table I), in particular considering as hyperparameters $\eta = 0.001$ and $\lambda \in [10^{-12}, 0.001]$ (Figures 12 and 13). The choice of the activation function does not seem to influence consistently the performance when training the network over several iterations (Figures 14 and 16).

[**logistic**]

Surely we can consider this work still open to improvements. Starting from fixing the code and filling the gaps, but also on the analysis of the performance. It would be very interesting to explore better the optimization algorithms, several combinations of number of hidden layer and respective neurons, various kind of learning rate, varying the input datasets, but also implementing new methods, such as the Batch Normalization, the Dropout or the Gradient Clipping.

REFERENCES

- FYS-STK3155 courses' notes
- Michael Nielsen, Neural Networks and Deep Learning
- UdiBhaskar/Deep-Learning GitHub repository
- Fisseha Berhane, Logistic Regression with a Neural Network mindset
- ML Glossary
- 3Blue1Brown, What is backpropagation really doing? — Chapter 3, Deep learning
- Sebastian Ruder, An overview of gradient descent optimization algorithms
- Medium, Aishwarya V Srinivasan, Stochastic Gradient Descent — Clearly Explained
- Medium, Mohammed Zeeshan Mulla, Cost, Activation, Loss Function—— Neural Network—— Deep Learning.
- Michael Nielsen, How the backpropagation algorithm works
- Author Rose Wambui,Cross-Entropy Loss and Its Applications in Deep Learning.
- Sophia Yang, Multiclass logistic regression from scratch.
- Image Classification using Logistic Regression in PyTorch