Classification and Regression? Insights in Linear Regression, Logistic Regression and Neural Networks

Project 2, FYS-STK4155, UiO

Varvara Bazilova, Sergio Andrés Díaz Meza

October 2020

Abstract

Various regression and classification methods are widely used Machine Learning algorithms. Classification approaches often utilize Neural Networks to address different problems with binary or other type of outcome (e.g. pixel based image classification or getting some insights about the dataset itself). In this report we present the results of application of Neural Network (with the backpropagation) algorithms to different problems (regression and classification). We compare the behaviour of the different activation functions in the Neural Network of the same architecture. We also explore Gradient Descent methods to optimize our algorithms. We conclude, that writing our own code provides more insides on the implemented algorithm, but using the existing libraries, such as Tensorflow allows to achieve a decent accuracy scores faster. Also scaling and "pre-processing" of any dataset is extremely important and has a huge impact on the results, unlike the penalty.

1 Introduction

The analysis of the various datasets require different solutions and approaches, depending of the goal of the analysis. To address some problems simple regression models are sufficient, others require more sophisticated approaches to analyze the data and/or make predictions. In this project we asses not only the various regression models, bus also the classification problems (logistic regression) and neural networks.

Therefore aim of the project 2 is to explore more about various Machine Learning algorithms for regression and classification: by developing our own Stochastic Gradient Descent, writing our code for the feed-forward Neural Network and comparing the results with the existing algorithms (such as Tensorflow).

We consider 3 types of data (Franke's function (Bazilova and Díaz, 2020), MNIST dataset of handwritten digits and Wisconsin cancer diagnosis dataset ("Data" section) for regression and classification using Ordinary Least Square approach with or without penalty influence and Neural Network of a certain architecture for regression and/or classification.

The report consists of: 8 major sections, including Introduction, Background of the study, description of Data and Methods, Results, Discussion and Conclusions, Code availability and the list of References. The report includes 16 figures.

The code is available as a supplementary material (see "Code availability" section).

2 Background

2.1 Gradient Descent Methods

Gradient Descent (GD) methods are the family of the optimization methods, used for finding a minimum of a given cost function. This is an iterative scheme, that takes into a count the learning rate (gradient step - hyperparameter) and initial parameters.

The basic idea of gradient descent is that that a function $F(x)$ decreases fastest if the one goes from x in the direction of the negative gradient. The iterations continue, until the minimum is reached (gradient equals or is close to zero). The minimum can be given by the certain accuracy (or interval): the bigger the interval is, the faster the gradient converges. In the ideal case the cost function reaches the global minimum.

The GD methods are sensitive to the learning rate changes, since the minimum is only reaches, if $F(x_{k+1}) \leq F(x_k)$ and the learning rate is sufficiently small.

Good initial guess of the parameters and the optimal learning rate values are crucial to the GD applications. If the initial guess is not good enough, then it will take a lot of time to converge to the minimum. If the learning rate is too small it will also take a long time to converge and if it is too large we can experience erratic behavior and stability problems. The learning rate depends on the given function and the chosen gradient descent method.

There are multiple gradient methods, such as Standard GD, Momentum GD, Schastic (mini batch) GD, Adaptive GD and many more. In this report we only focus on some of them.

2.1.1 Standard Gradient Descent

When computing Standard GD the entire training dataset is used.

The limitations of this methods are: Standard GD is sensitive to initial guess of the parameters (initial conditions), treating local minimum as a global (and therefore making a mistake), sensitive to learning rate parameter and it becomes computationally expensive, if the chosen learning rate is too small.

2.1.2 Stochastic Gradient Descent

Stochastic GD methods imply, that not the whole dataset is used for computing the gradient, but some randomness is introduced to the iterative scheme in one way or another: a single random sample is introduced on each iteration. The gradient is computed independently using so-called "batches" and then the expected value is used. Therefore the batches are sampled from a given dataset in a stochastic way.

Another way of computing the Stochastic GD is using so-called "mini-batches". The points are sampled without replacement and computed sequentially foe every "mini-batch" independently:

$$\beta_{(1)}^{(1)} = \beta_0 - \gamma_0 * \sum \bigtriangledown \beta(c(\beta^{(0)})) \tag{1}$$

Where γ_0 is a learning rate parameter at the first iteration, $c(\beta^{(0)})$ is a cost function on the first iteration. Learning rate can be fixed or can vary with iterations. The iterations continue until the entire dataset is sampled. The procedure can be repeated many times (until the last "mini-batch". This defines a so-called "epoch".

The stochastic component allows to avoid the local minimum problem, since the gradient might not get "stuck" there. However the computational cost might be a bottleneck for this method, however the calculations can be run in parallel. Using mini-batches is one of many different ways of computing Stochastic GD.

2.2 Neural Network

A neural network (NN) is a two-stage regression or classification model (Friedman et al., 2001), typically represented by a network diagram:

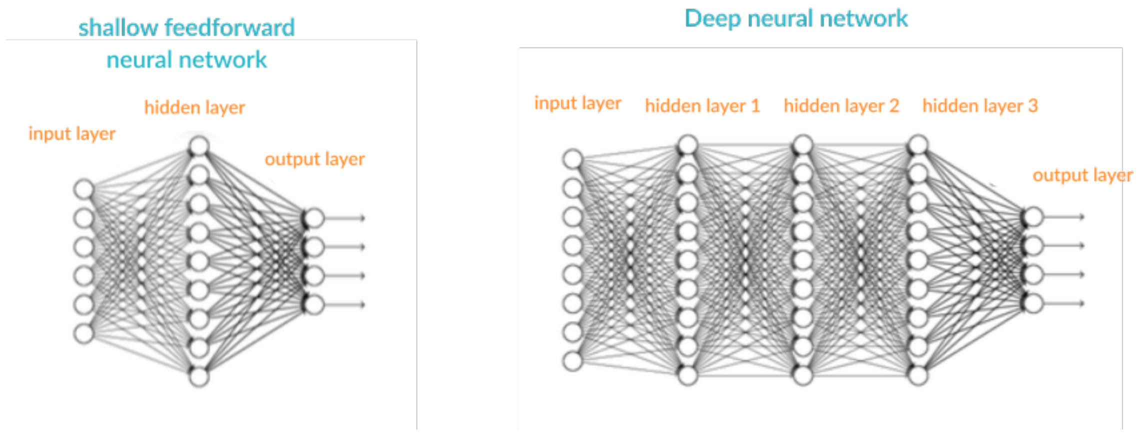


Figure 1: Neural Network diagram (Goodfellow et al. (2016))

NN consists of multiple layers: input layer, set of hidden layers, and output layer. Each layer consists of number of so called ”neurons” or ”nodes”, vector of biases $[b]$ and vector of weights $[w]$ (the length of a vector correspond to the number of nodes in the layer). Layers are sequentially connected between each other with activation functions, applied to neurons. Activation function can be defined differently for every layer. Most commonly used activation functions are: Sigmoid function, Exponential Linear Unit (ELU), Rectified Linear Unit (ReLU), Leaky Rectified Linear Unit (leaky ReLU), Tanh, Softmax function etc.

2.2.1 Feed Forward Neural Network and Activation Function

Feedforward Neural Network (FFNN) is the type of NN, where the layers are connected only in one direction: forward (from the input towards the output through the sequence of hidden layers). The connections do not go backwards or form loops (unlike some other NN types, e.g. reccuring NN). The output from one layer is used as input to the next layer (Goodfellow et al., 2016). FFNN is the simplest NN algorithm. To connect the last hidden layer and the output layer (the last activation function) the softmax function is used in this report.

2.2.2 Weights & Biases

When training a NN, weight and biases are introduced on each layer of NN. Weights (w) and biases (b) are initialized as vectors. These vectors are usually initialized as normally distributed random numbers from 0 to 1 (normally distributed small numbers).

2.2.3 Backpropagation

Backpropagation algorithm is an algorithm, which is often used to train the FFNN. The underlying math behind the backpropagation algorithm is the derivation of a cost function in respect to weights and biases: $\frac{\partial c}{\partial w}$ and $\frac{\partial c}{\partial b}$ (by the chain rule). After comparing the output layer with the targets the error is computed (Goodfellow et al., 2016). Then the backpropagation algorithm computes the gradient using the derivatives for each layer, iterating backward from the last layer. The gradient allows us to evaluate how quickly the cost changes when we change the weights and biases. This allows the NN to update weighs and biases values, since at the beginning they are initialized randomly.

2.3 Logistic Regression

3 Data and Methods

3.1 Data

The data, used in this report considered in the report consists of 3 parts:

1) Franke Function is a two-dimensional function, that has two Gaussian peaks of different heights, and a smaller dip, with normal distributed random values to introduce stochastic nosie to the data (described in detail in the Data and Methods section of the Report 1 "Exploring Linear Regression Methods: Validation and Terrain Model Applications") (Bazilova and Díaz, 2020). Data, generated from Franke Function is used for Stochastic Gradient Descent for Ordinary Least Squares (SGD-OLS) and Neural Network (NN). Inputs used will the the X and Y values represented ina Design Matrix (see Bazilova and Díaz (2020) for explanation) to predict the Z value (surface prediction).

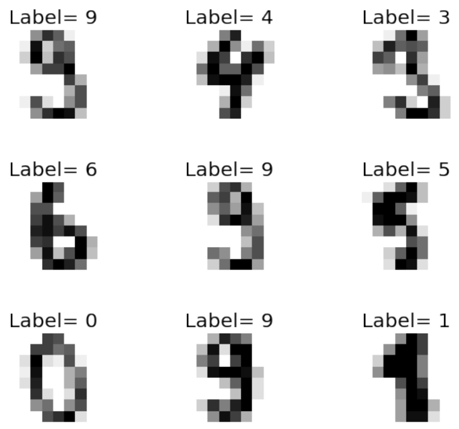


Figure 2: 9 randoms images from the MINST dataset (LeCun and Cortes, 2010) with their respective labels.

2) MNIST (National Institute of Standards and Technology) dataset - tha databese (see Figure 2), that consists a set of hand-written digits. It is often used as an example dataset for Neural Networks or classification. It consists of 60,000 training images and 10,000 testing images (LeCun and Cortes, 2010). In this report MNIST data is used for the classification and Neural Network algorithm assessment. Inputs will be images reshaped into 1D array with the length equal to the pixels in the each image, containing values of the pixels, normally ranging between 0 and 255.

3) The Wisconsin Breast Cancer Diagnostic dataset (Dua and Graff, 2017) (see Figure 2). This dataset consists of various tumour parameters (such as radius, area, texture etc.) used to determine whereas the patient has benign cancer or malignant cancer. The dataset is often used as an example for various machine learning applications with binary outcome (e.g. classification).

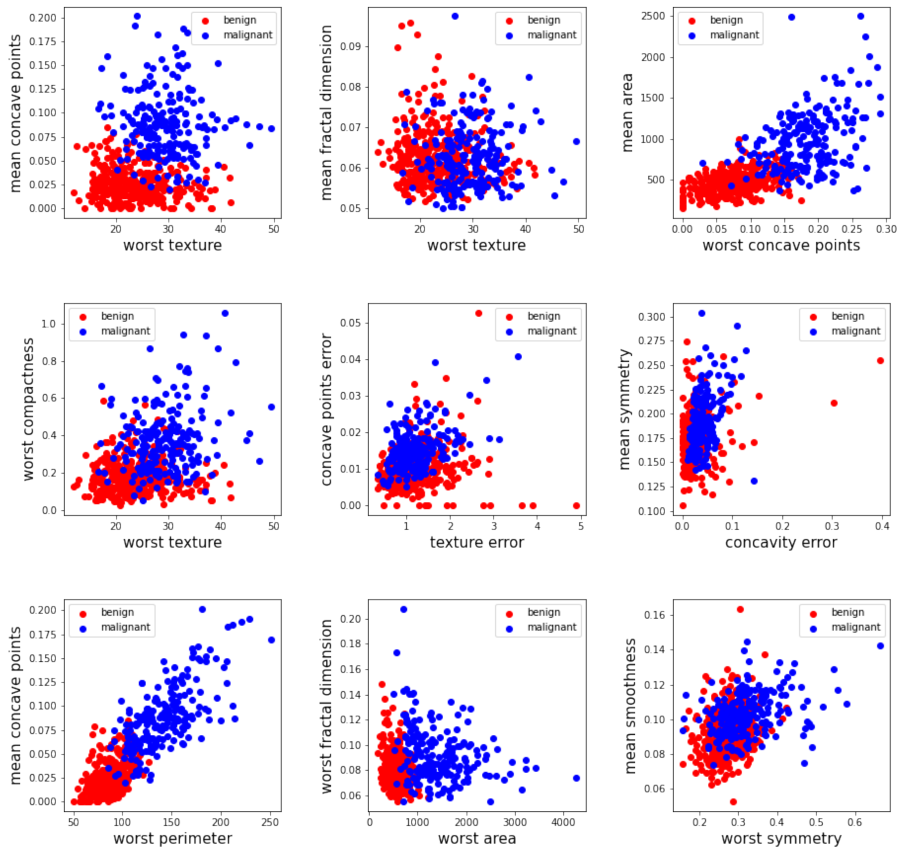


Figure 3: 9 plots between random features from the Wisconsin Breast Cancer Diagnostic dataset (Dua and Graff, 2017). Red and blue dots are the data that has been classified as benign cancer and malignant cancer, respectively.

3.2 Methods

3.2.1 Stochastic Gradient Descent for Ordinary Least Square (SGD-OLS)

The Ordinary Least Square regression (OLS) is combined with the Stochastic Gradient Descent (SGD) in our own implemented code, so that the coefficients for the Franke's function regression are updated by a gradient descent, governed by an established learning rate. The SGD-OLS was applied to study the Franke's function fit. This time, compared to previous study in the Report 1 (Bazilova and Díaz, 2020), the data from Franke's function was shuffled before the regression implementation. This was done to assure that when the data is sampled randomly in the mini-batches, it has more probability of describing more general the Franke's function surface values.

The code was made as a function (function *SGD* inside *methods.py* file) to control the entry of the input values and the output values, iteration numbers, mini-batch size, learning rate and penalty. Some of the parameters are empty values, unless the user wants to make use of them, so a value different to *None* is introduced. Thus the algorithm obeys the new values and including them in the operation. In case, if the learning rate is not entered, then the code runs with a Scaled Learning Rate method. Introducing a penalty value would change the algorithm to make a SGD with a Ridge Regression form instead of a OLS, meaning adding a penalty parameter to the coefficients. Equations 2 and 3 shows the difference in the gradient calculations when it is OLS VS a Ridge regression.

$$\bigtriangledown \beta = X^T(X\beta - Y) \tag{2}$$

$$\bigtriangledown \beta = X^T(X\beta - Y) + \lambda * \beta \tag{3}$$

We implemented the code varying the penalty values, learning rates, mini-batches, and epochs against the polynomial degree since this is a function fit.

3.2.2 Feed Forward Neural Network

We wrote our own algorithm for Feed Forward Neural Network (FFNN), which will be refer from now on as Beta Neural Network (Beta NN). The algorithm was implemented as a class object, so all the functions are defined inside the class and everything operates inside it, including the parameter updates in the initialized class. File *nn.py* contains the class NeuralNetwork, which receives several inputs for being initialized: inputs, targets, a list of the number of nodes per each layer (architecture), number of nodes in output, number of iterations, learning rate, activation function to use, penalty parameter and an option for using the Softmax function at the output layer. The activation functions, included to the class are: Sigmoid, Hyperbolic Tangent (Tanh), Rectifier Linear Unit (ReLU) and Leaky ReLU. The algorithm also has the functions for checking shapes of Weights and Biases initialized in all the layers, a *train* function to make the iterations and update the Weights and Biases matrices, and *predict* function to evaluate for other datasets or the Test set once the NN is trained.

The Beta NN was used to study the Franke function values prediction and the MINST dataset classification. For this purposes the number of outputs had to be changed so it matches the purposes. One output node was used for determining the Franke function values, without using the Softmax function.

For the MINST dataset, the output layer has to match the number of neurons with the same quantity of all possible classes of the MINST dataset and using the Softmax function at the end. Each neuron will predict the probability of the input to belong to the class the node was assigned to. The maximum probability between them recall the final class to which the input belongs to. In order to enter the targets for Cost Function calculations and output classification form probabilities to a defined class, an encoder and a decoder method was programmed inside the *methods.py* file to convert class labels into, what is called, logits and vice versa.

The implementation of the Beta NN algorithm is made in order to explore variations of performance (Accuracy for classification and MSE for regression) with epochs, penalty values, learning rates, and activation functions and the architecture itself (number of hidden layers and number of nodes per layer). In order to validate the Beta NN implementation, another algorithm for FFNN was used from Keras, a Deep Learning API from Tensorflow environment package. the implementation of this NN algorithm was made in such a way so that same variations and changes in architecture can be done as with the Beta NN. The rest of the document will refer to this algorithm as Keras NN, initialized in the *project_2.ipynb* jupyter notebook.

3.2.3 Logistic Regression

We wrote also our own algorithm for Logistic Regression in the *nn.py* called *Logistic_Regression* files as another class with similar internal functions as the Beta NN (*train* and *predict*). Since this method was used to predict a classification between two classes (binary classification), then a probability in the algorithm is computed using the Sigmoid function and a threshold is established (normally 0.5) to determine whether the input is from one class (probability higher than 0.5) or from the other one (probability less or equal than 0.5). By determining the probability of an input in one of the classes, the remaining is the probability of the other class.

There was more similarity between this method and the SGD-OLS than to the Beta NN. However certain features inside the code had to be modified. The Cost Function is now determined as the difference between the logits (0 or 1, representing each class) and the probability calculated for that input, and this is used to update the coefficients. Each coefficient represents a linear function that separates the classes depending on the different features (making clusters), which is something similar as what is illustrated in Figure 3, between the red and blue dots for each relation between different features (variables).

Since the order of the algorithm is inspired in the SGD-OLS methods and the Beta NN class structure, the Logistic Regression is made in such a way that we can study variations in parameters such as penalty parameters and epochs, which will be seen in results.

4 Results

4.1 Stochastic Gradient Descent

SGD with OLS shows certain amount of noise in all of their plots (Figure 4 and 5). The first line of the first mentioned Figure illustrates a possible low error trend from the upper-left corner of the plots towards the down-right corner. That means that for the better performance, the learning rate must be decreased, if the polynomial degree becomes higher. Thw second line shows the relation between epoch number and polynomial degrees: the error reduces when we advance from low to high epoch number. Nevertheless, higher errors in polynomial degrees around 19 can be noticed and for epochs around 5. At the first line of Figure 5, it is mainly show random values by varying the mini-Batch size against the polynomial degree. However, a tendency to lower errors is evidenced when passing from 5 polynomial degree down to 0. The second line shows relative low errors for very low values of penalty, but then the error increases when reaching 0.1.

4.2 Neural Network

4.2.1 Franke Function Prediction

Figure 6 shows performance on Train and Test sets using Beta NN and Keras NN. Both of them doesn't show a decaying trend while increasing complexity degree like the one shown in Figure 3 by Bazilova and Díaz (2020), but divergent results in a closed range. Also, Keras NN shows higher errors than the Beta NN and both Train and Test sets behave nearly the same.

The Figure 7 shows the influence of certain parameters in a Neural Network (Beta NN used). Penalty parameter gives some stochastic noise to the performance, however, this noise gets considerably scaled (the variation gets smaller) after certain amount of iterations as shown in the first line of the Figure, but then spikes up rapidly while approaching penalty values of 10^{-1}. For the Learning rate, Figure 7, second line shows a constant performance until we reach values of learning rates near 10^{-2}. From this point the performance increase and error decreases substantially, creating a sort of "error valley" in a range of learning rates, until it spikes up faster at near 1.9. Figure 8a and 7b shows for 100 epochs a sort of a "cross-section" of the Figure 7, first and second line.

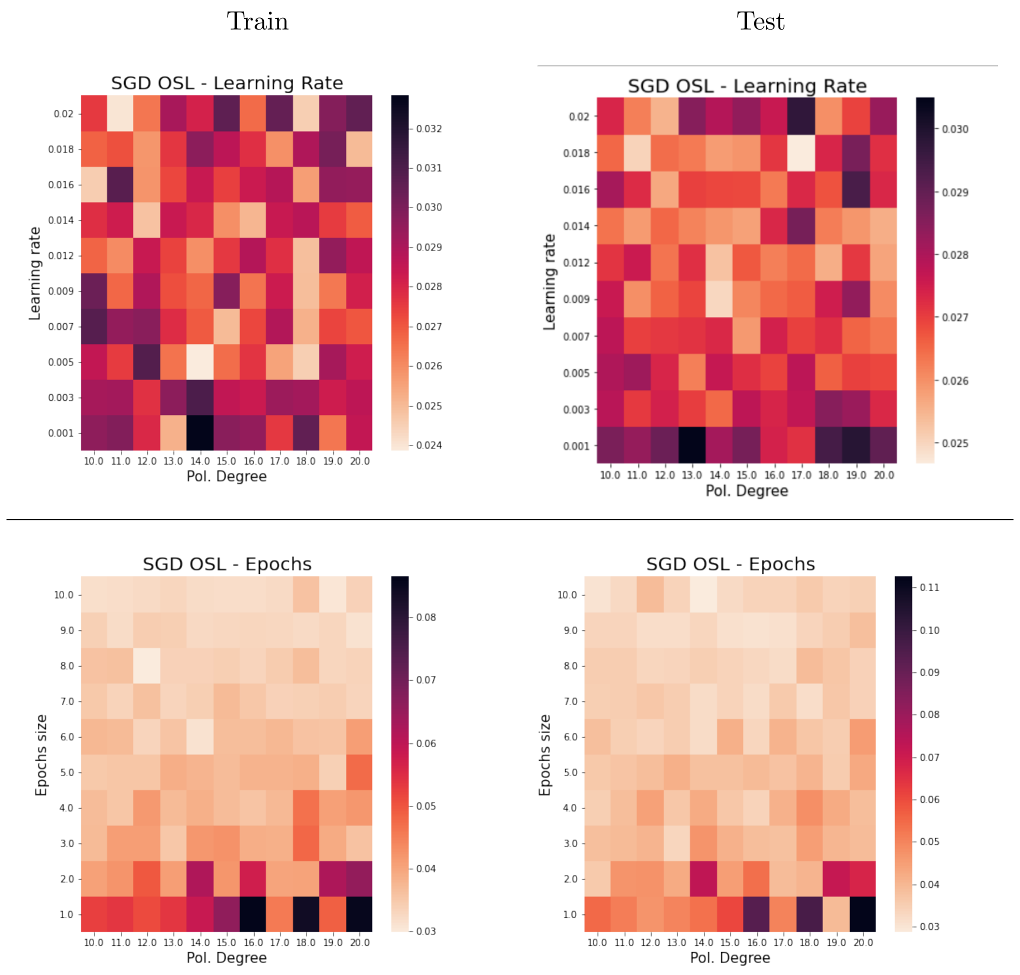


Figure 4: Stochastic Gradient Descent performance (MSE) for Train and Test sets (left and right columns) varying complexity degree (from 10 to 20 polynomial degree) and several parameters. *First line:* Varying Learning Rate, *Second line:* Varying Epoch of number of iterations, *Third line:* Varying mini-Batch size, *Fourth line:* Varying the Penalty parameter.

As explained in the Methods section, the NN are computed with sigmoid function as the standard activation function. However, Figure 9 shows the variability on performance when these activation functions are changed to ReLU and Leaky ReLU. Both Sigmoid (also supported by other figures) and ReLU (Figure 9a and 9b) proves to converge to low error values, nevertheless, Leaky ReLU (Figure 9c) gives high errors since the beginning.

4.2.2 MINST Classification

For a first insight into NN classification, Figure 10 shows the performance of the Beta NN in contrast with the Keras NN for an image classification problem over the MINST dataset. In this case the Keras NN seems to converge to an Accuracy score near to 1 sooner than the Beta NN by a high difference of iteration numbers (epochs), however, both of them are able to reach near 1 of accuracy after certain

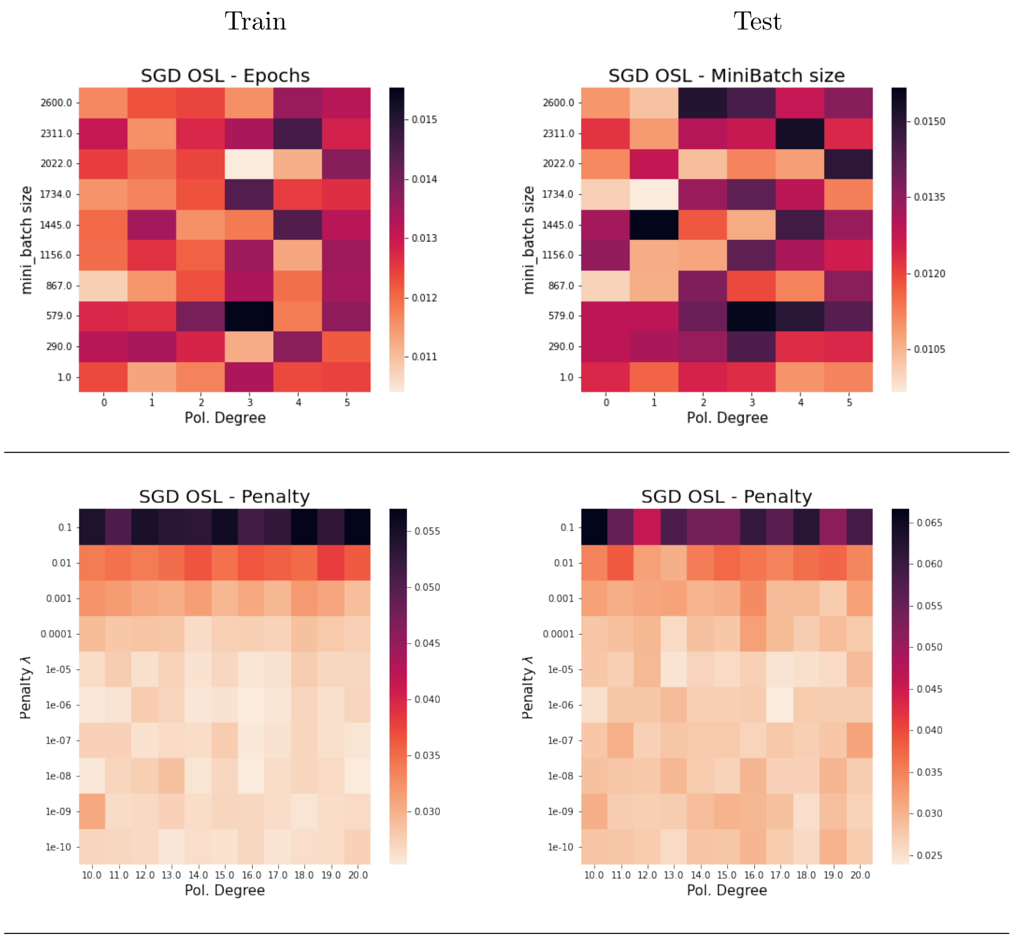
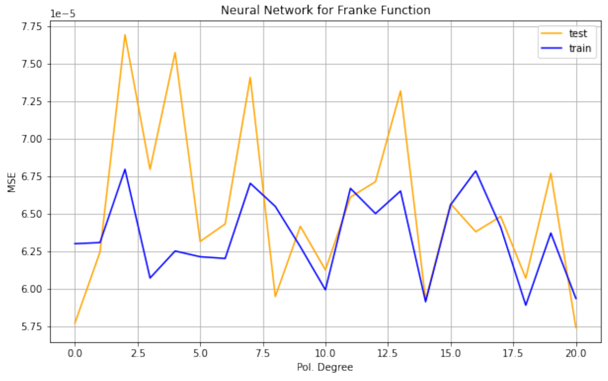


Figure 5: Stochastic Gradient Descent performance (MSE) for Train and Test sets (left and right columns) varying complexity degree (from 10 to 20 polynomial degree) and several parameters. *First line:* Varying mini-Batch size, *Second line:* Varying the Penalty parameter.

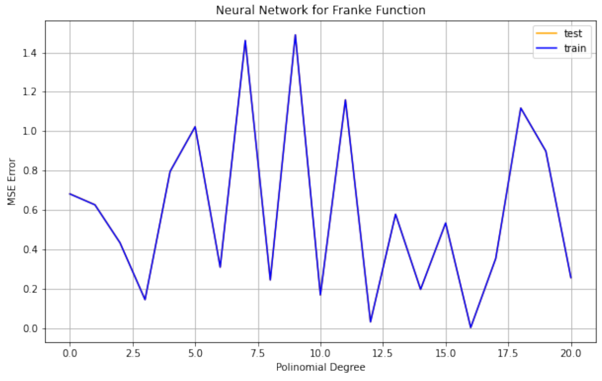
iteration numbers.

It might that Sigmoid is not the proper activation function for this classification problem in particular. Therefore, Figure 11 illustrates the behavior of four types of activation functions on the MINST dataset. Tanh seems to give low variations in contrast to the rest of the activation functions and also converges faster to the stabilization than the others. On the other hand the Sigmoid function seems to perform way worse than the other activation functions, both with local variations and the need of high iterations for a stable convergence.

Besides the activation functions, other factors that might affect is the simple architecture of the Neural Network. Figure 12 shows differences in performance when both the number of hidden layers and the number of nodes in each hidden layer are affected. It is noticeable an accepted performance (Accuracy $>$ 0.8) from 1 to 3 hidden layers. However, when more hidden layers are added, the number of nodes per layer must increase in an exponential way if an accepted performance wants to be achieved.



(a) Beta NN.



(b) Keras NN.

Figure 6: NN performance for the Franke function by dependence with polynomial degree. Train and Test sets are indicated by blue and orange lines, respectively. Epochs used: 100, Learning rate: 0.014. Architecture: Three hidden layers (10 nodes each). Activation function: Sigmoid. No use of Softmax function at the output layer.

4.3 Breast Cancer Classification

Breast Cancer dataset (Dua and Graff, 2017) was used for classification. However, as it was mentioned before, this classification in binary (only two possible classes). Figure 13 shows the difference in performance between the Beta NN and the Logistic Regression for this problem in particular (a binary classification). Results shows that the Beta NN converges faster than the Logistic Regression, but Beta NN achieves stability in less accuracy values than the Logistic Regression (near 0.85 against 0.9). Also, it is evidenced how Logistic Regression starts with an high accuracy value in the first iteration (near 0.8), decreases its performance to near 0.47 and then rises it up again in order to stabilize it (fully trained) at near 0.9.

When applying a variation of penalty to the iterations of the Logistic Regression, the behavior is similar as for the NN while studying the Franke function (Figure 7). Figure 14 now shows the variations in performance when penalty is applied against the iterations. At the first epochs, the penalty parameter changes several performance values. However as the iterations as higher, there are no variations in performance while varying the penalty parameter.

5 Discussion

5.1 SGD

The noise on the plot (Figure 4 and 5) can be easily related to the nature of the stochastic behavior of the algorithm. It is useful to remember that in Standard Gradient Descent the gradient is computed using all the elements in the training set. However, in the Stochastic Gradient Descent a certain amount of mini-batches; each one made by random sampling, computes the gradient to be used and that over the iterations. This gives a sort of a ”random walk” through the cost function that could converge to the ideal optimal solution later than the Standard method, but it could overpass local minimums until a global one is reached. This random walk can be seen in the mentioned noise.

In the learning rate variability, the mentioned low error trend that goes from upper-left to lower-right corner is dependent to the complexity degree. To the more polynomial degree to fit, a lower learning

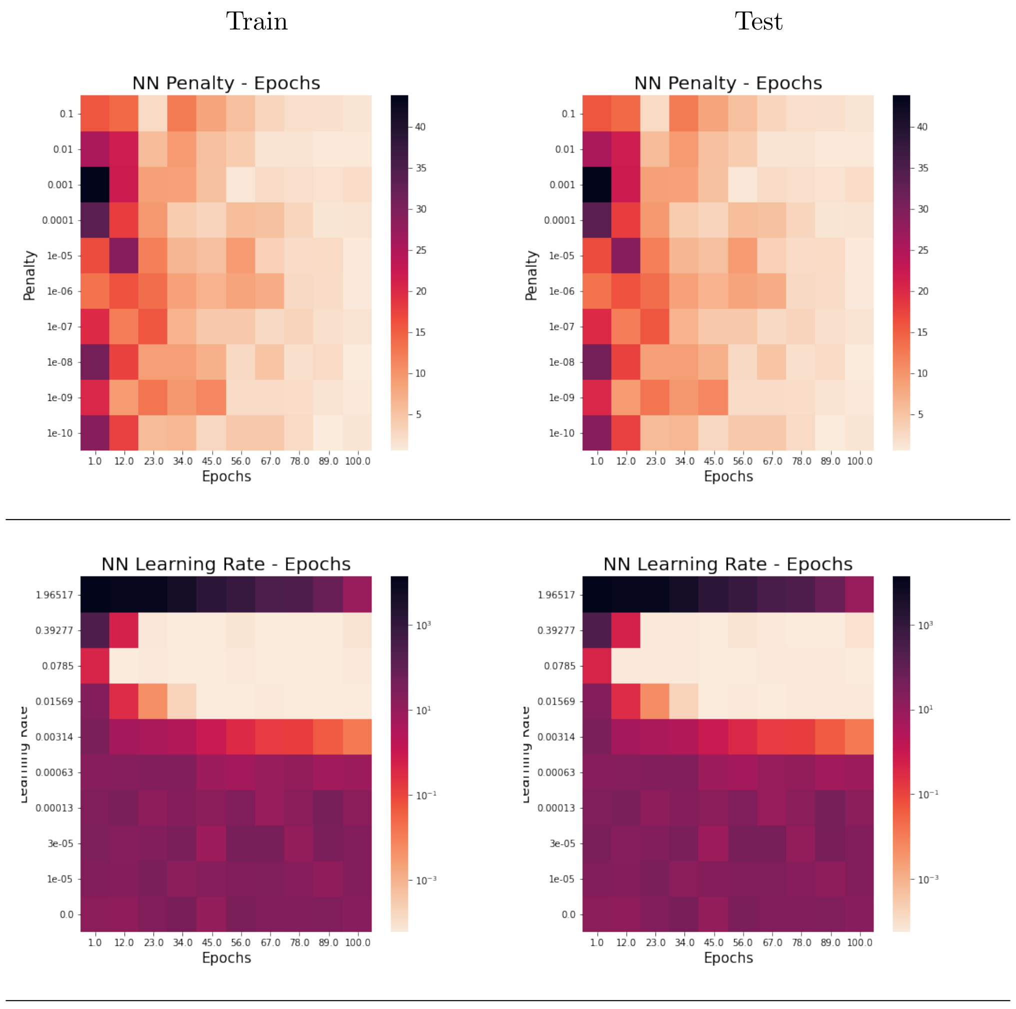
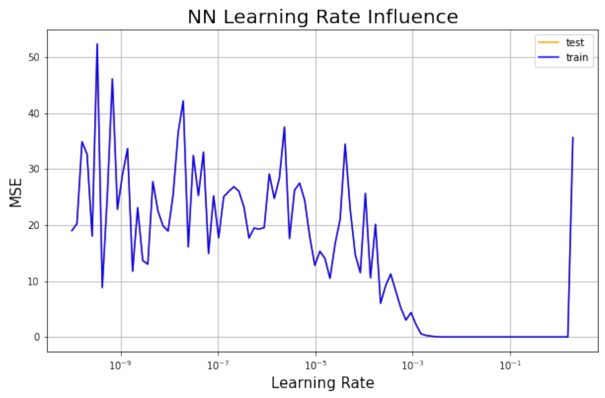


Figure 7: Beta NN performance (MSE) for Train and Test sets (left and right columns) varying epochs and several parameters. *First line:* Varying λ penalty parameter, *Second line:* Varying the Learning rate.

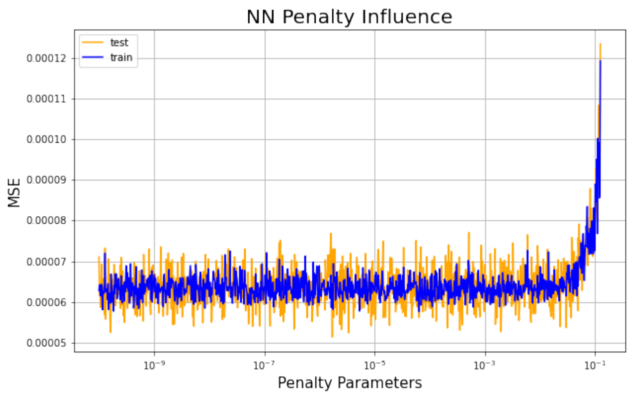
rate is needed. This could indicate that the Cost Function has more variations at higher orders, which is something evidenced in the Report 1 Bazilova and Díaz (2020) when high polynomial degrees could create an overfitting problem.

The variability of Epochs just shows what was expected. The more iterations there are, the faster it takes to the model to converge to an optimal value and just do local variations around the minimum. However, this particular case is very interesting. It stands out and shows unexpected behaviour: its' "fast" convergence to such lower error values. Even with an epoch of 1, the error can reach down between 0.05 and 0.06. Without looking to the epochs, all the heatmaps from Figures 4 and 5 describe also good performances (low errors). This is because another parameters that hasn't been shown affects the convergence of the results, and in this case is the initial guess of the coefficients.

To see more clearly the latter topic, another extra plot was added to the one of Epoch variability in Figure 4 (second line). Now the Figure 15 shows the difference in convergence errors when the initial guess of coefficients for Franke Function is made with the *numpy.random.random()* function (random values from 0 to 1) and when is made with *numpy.random.randint()* function (random integer values from 0 to



(a) Learning rate variability.



(b) Penalty variability.

Figure 8: Beta NN performance for the Franke function by varying penalty parameter λ and learning rate. Train and Test sets are indicated by blue and orange lines, respectively. Epochs used: 100, Learning rate: 0.014. Architecture: Three hidden layers (10 nodes each). Activation function: Sigmoid. No use of Softmax function at the output layer.

10). Here the higher initial values of the coefficients of the polynomial regressions starts with higher errors than the initialization with random values from 0 to 1 and converges more slowly to similar error values than the one with initial values from 0 to 1. Therefore there is a "lucky strike" at the beginning, which tells that the ideal coefficients are between 0 and 1 and that the one just was lucky on initializing near the optimal. This is also confirmed by the stochastic methodology that could assure trespassing a local minimum in the Cost Function.

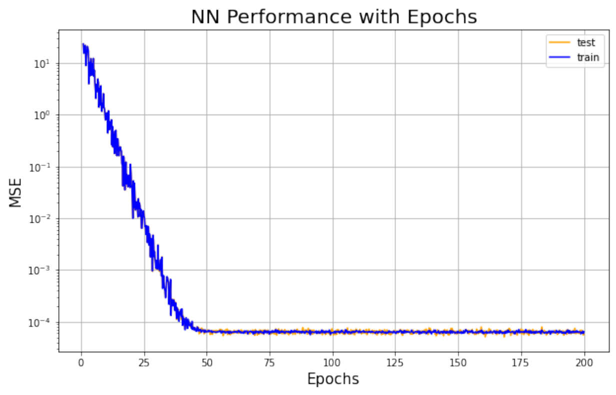
The mini-batch size variability was expected to behave the following way: the increase in the mini-batch size would create more error, since the stochastic re-picking of the samples would create new Cost Functions each time. And new Cost Functions might not see local minimum created by other group of selected samples. However, the results here, which seem to be very random can't be explained, since it is suspected that the code has something strange concerning the variations with mini-batch sizes. Also big polynomial degrees create complexity in calculations and errors. The ranges of polynomial degrees used for this case were not the desired ones. all the other plots has ranges from 10 to 20 since Bazilova and Díaz (2020) showed good performance between these values.

Regarding the penalty parameters, a logarithmic distribution was used in the same way as the ones suggested for the last report (Bazilova and Díaz, 2020). Ranges from 10^{-10} to 10^{-4} seems to give good results but not generating too much improvements in the performance. Above those ranges, the penalty begins to influence the performance in a bad way. The main suggestion of this is simply that the model does not need the apply of a penalty to improve the performance, at least for the Franke Function.

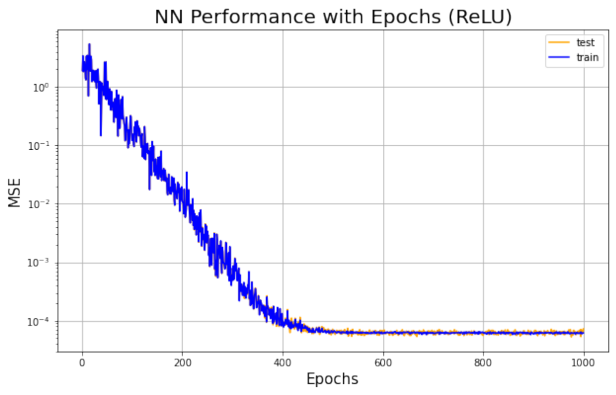
5.2 Neural Network

5.2.1 Franke Function Prediction

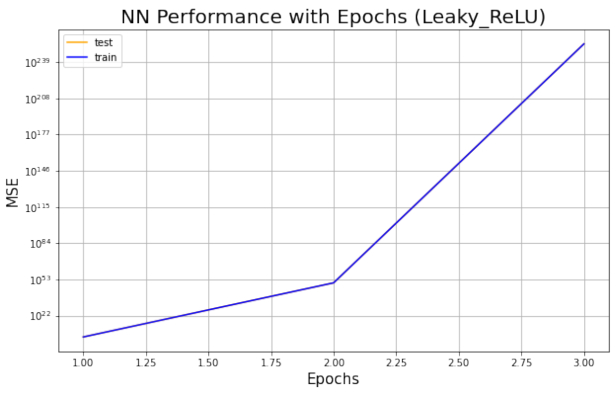
The best Figure to describe the behavior of the NN applied to the Franke function is the Figure 6. According to the Beta NN performance against the polynomials degree, the MSE error shows even better performance than the SGD-OLS. This is something expected, but the nature of the fitting is different, depending on the objective. If we just want to predict the data in the way it comes, then it is better to use a NN, since it performance is more accurate than the OLS. However, if a functional fit is needed,



(a) Sigmoid.



(b) ReLU.

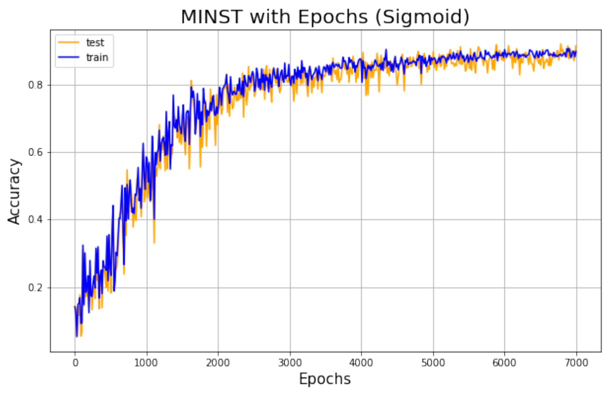


(c) Leaky ReLU.

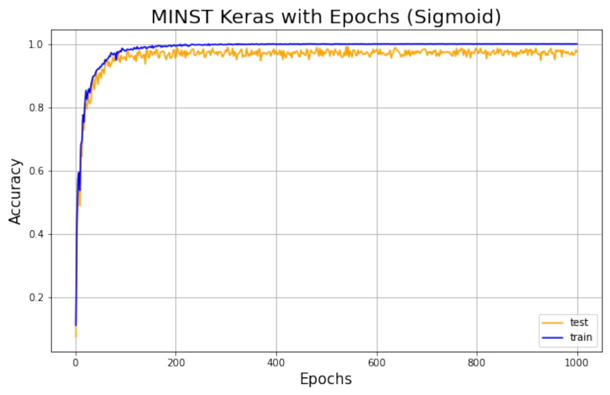
Figure 9: Beta NN performance for the Franke function by varying the activation function in the hidden layers varying epochs. Train and Test sets are indicated by blue and orange lines, respectively. Learning rate: 0.014. Architecture: Three hidden layers (10 nodes each). No use of Softmax function at the output layer.

then OLS would give a more realistic approximation. The NN can predict without fitting a polynomial function, so if there would be a plot an X vs Z for a Y fixed value in the Franke function (fit between NN and SGD-OLS). The one can notice, how SGD-OLS made a continuous smooth function trying to pass through all possible points, whereas the NN will be shown as straight lines with several jumps (not smooth). This means that at each point of the SGD-OLS method, all points close between them will have a very similar value, if the derivative is evaluated on those points. However, since NN would not create smooth, then the derivatives would be different between them. This can also say that SGD-OLS is less perceptive to introduced stochastic noise to the original dataset, while NN tries to take it into account in such a way to create a non continuous function to fit the noise also.

Another thing to notice is that for Beta NN and Keras NN there is a local variation around an error value, but the convergence from high to low error values is not happening in this case. This just tells that the NN is not perceptible to polynomial degrees, but actually it will try to fit as well as possible the inputs regarding how many features it comes (features in this case are operations between X and Y). By



(a) Beta NN.



(b) Keras NN.

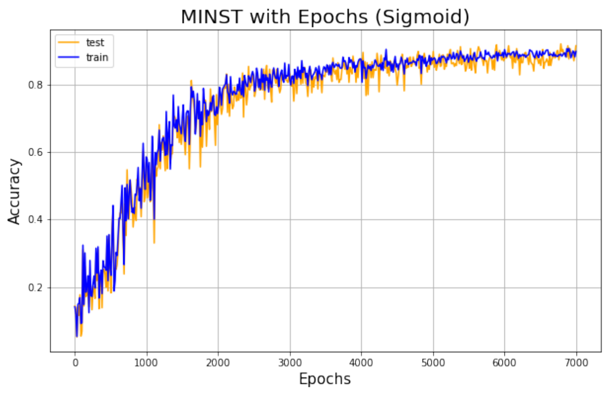
Figure 10: Beta NN and Keras NN performance (Accuracy) for the MINST dataset classification while varying epochs. Train and Test sets are indicated by blue and orange lines, respectively. Learning rate: 0.014. Architecture: One hidden layers (50 nodes), Activation function: Sigmoid.

just passing the X and Y values, the NN will perform the regression. This is why from this step, only polynomial degree of 1 was used (X and Y values). When comparing the Beta and the Keras NN, it seems that there are more variations and more errors on the Keras than in Beta NN, ignoring that the behavior is the same. We believe that the difference is mostly marked by the initialization of the inputs. Since Beta NN was written specifically for this report, there is more control on the initialization of the data and the operations inside hidden layers. However, since Keras NN is an imported library, there is not so much control on the things that happen inside, specially in the pre-processing of the data, since at some point it must be normalized or standarized somehow so the algorithm might not explode when it is operated with Weights and Biases. Hence, this is a bit of a "black box" for the user. And there is not much knowledge on the pre-processing steps (if there are such) of the algorithms used in Keras/Tensorflow do to the data before passing them through the established architecture of the Keras NN.

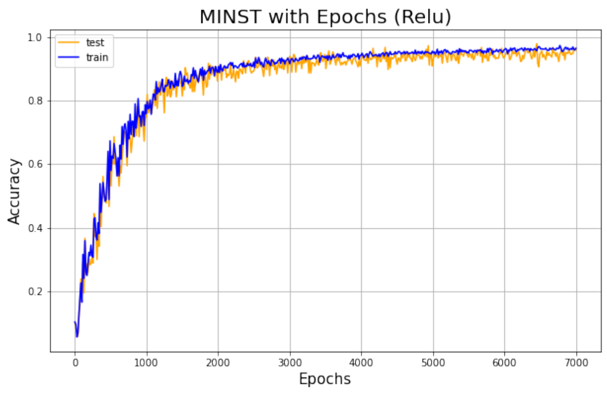
Now, seeing the hyper-parameter changes for the Beta NN (Figure 7) algorithm, the behaviors observed were the expected. Penalties in low number of iterations (Figure 7, first line) influences in the performance error, but as the iterations are more, the NN trains better so that the penalties are no longer needed. It is possible that actually what the penalty do is to smooth the "fit function" in the same example illustrated in the beginning of this section, on this discussion. The random variations at the Franke function for NN can also be seen in the Figure 8b.

Concerning the learning rates in the last mentioned Figure (second line), very small learning rates performs bad, until we encounter learning rate values near of 0.00314. From here it is noticed a range of optimal learning rates from near 0.01569 to near 0.39277. Also Figure 8a shows what it is a cross-section, and here it evidenced even more a numerical problem than cannot be evident on the heatmap. The flat part of the plot only shows a numerical stability problem when computing the Softmax function due to very low reached values in one of the inputs. After this range, it "explodes". This happens due to the standard learning rate for the rest of the performance tests was chosen as 0.014.

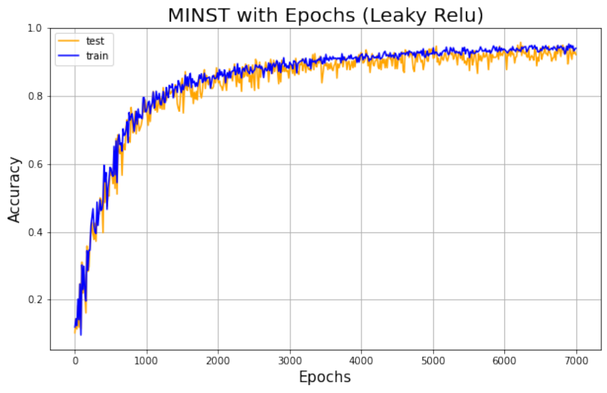
Regarding the variations in activation functions: the Sigmoid (standard in this exercise) was compared with ReLU and Leaky ReLU. The Sigmoid is the one who performs the best for the Franke function, converging after 50 iterations, while ReLU was converging 10 times slower (near 500 iterations). In addition, Leaky ReLU made the performance to "explode" to extremely high error values. Even so, just the first three epochs were plotted since it exploded way fast. This might be due to the main difference



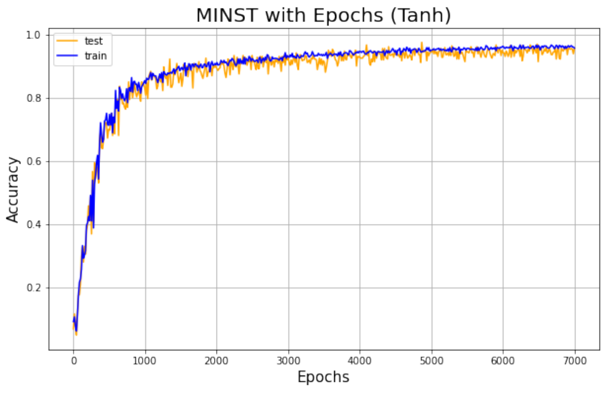
(a) Beta NN with Sigmoid.



(b) Beta NN with ReLU.



(c) Beta NN with Leaky ReLU.



(d) Beta NN with Tanh.

Figure 11: Beta NN performance (Accuracy) for the MINST dataset classification while varying epochs, and with different activation functions: Sigmoid, ReLU, Leaky ReLU and Hyperbolic Tangent. Train and Test sets are indicated by blue and orange lines, respectively. Learning rate: 0.014. Architecture: One hidden layers (50 nodes).

between the Leaky ReLU and the ReLU itself, and is the low leaks when evaluating values less than 0, but for the positive values, its a simpler linear activation function. On the other hand, the Sigmoid function maintains the values between 0 and 1. This can also be due to the way of the input values, since pre-processing for NN is important, and so there can be a strong relation between the activation function performances and the way the inputs are introduced or the pre-processing of the original data before it merges with the hidden layer operations.

5.2.2 MINST Classification

In the previous sections, the Beta and Keras NN were used for studying a regression problem. In this case, they were applied to study a classification problem. It is based on computing probabilities of for which class does an specific input fits more. The Figure 10 shows the difference between the Beta and the Keras NN for the MINST dataset classification, and this time the performance of the Keras NN is evidently better than the Beta NN. Keras needs between 200 and 400 iterations to be fully and optimally trained,

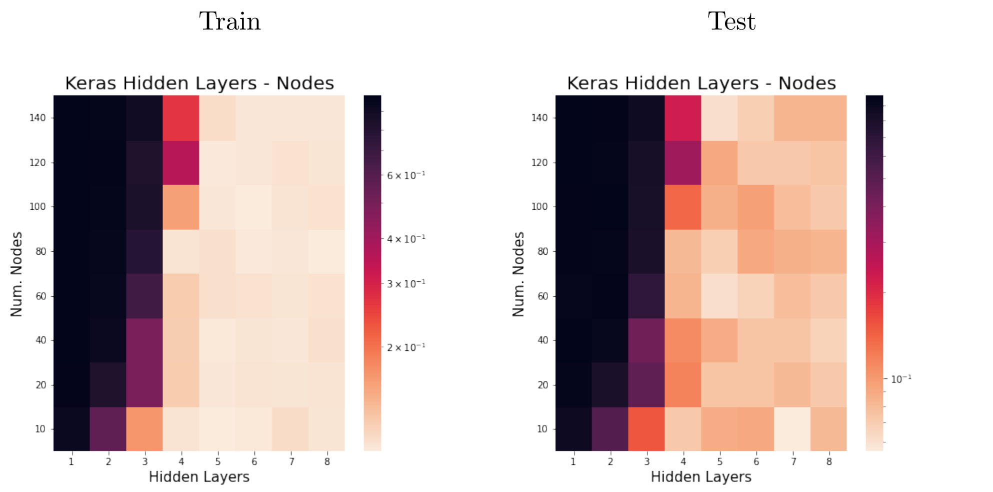
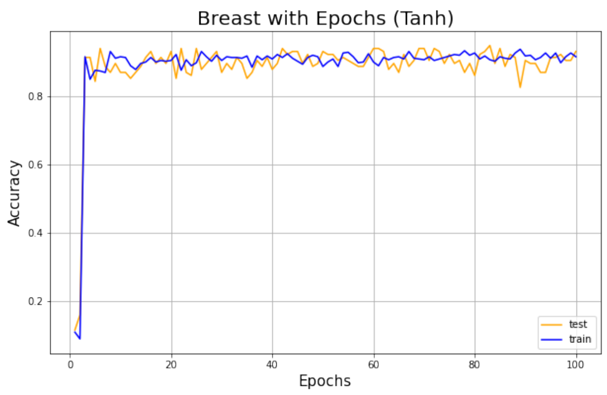
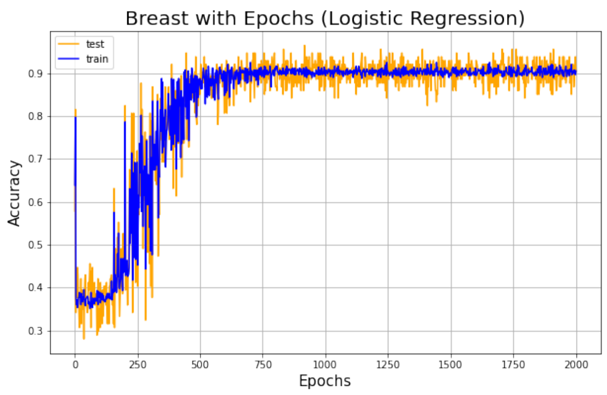


Figure 12: Keras NN performance (Accuracy score) for Train and Test sets of the MINST dataset (left and right columns) varying hidden layer numbers and node numbers.



(a) Beta NN with Tanh.



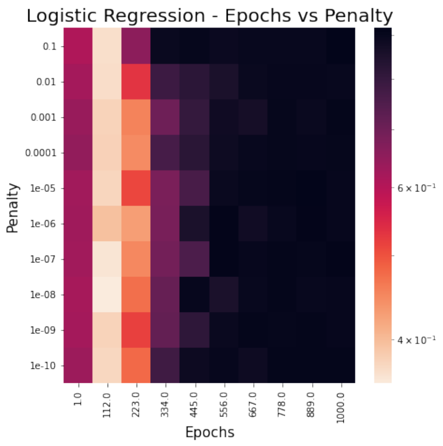
(b) Logistic Regression

Figure 13: Beta NN and Logistic Regression performance (Accuracy) for the Breast Cancer dataset classification while varying epochs. Beta NN made use of the Tanh activation function. Train and Test sets are indicated by blue and orange lines, respectively. Learning rate: 0.014. Architecture of the Beta NN: 1 hidden layer, 50 nodes, 2 outputs.

whereas Beta needs at least 7000 iterations.

The concern here is that the MINST data is not initially normalized. Since the MINST are just pixels from images, their values might range from 0 to 255 (corresponding to the standard 8 bit color-coding). A normalization brings the values down to ranges between 0 and 1, or an standarization brings them between -1 and 1. So there is something in the pre-processing that Keras NN does in a way, that makes the convergence and performance to be so close to 1. On the other hand, since data normalization for Beta NN was not done in the same way, the performance is worse and it needed more iterations to reach an optimal performance.

Train Test



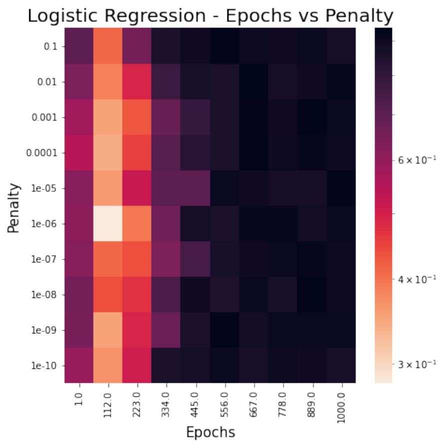
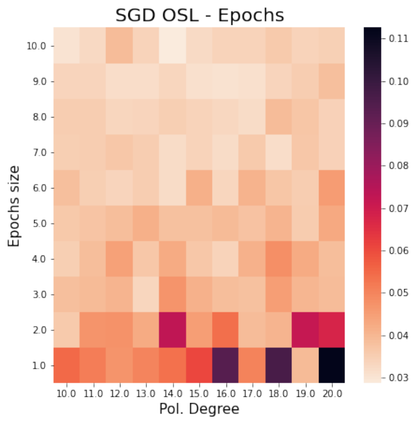


Figure 14: Logistic Regression performance (Accuracy) for Train and Test sets of the Breast Cancer dataset (left and right columns) varying epochs and penalty (regularization) values.

numpy.random.random() *numpy.random.randint()*



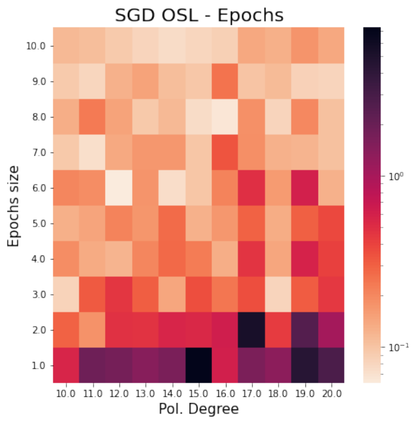
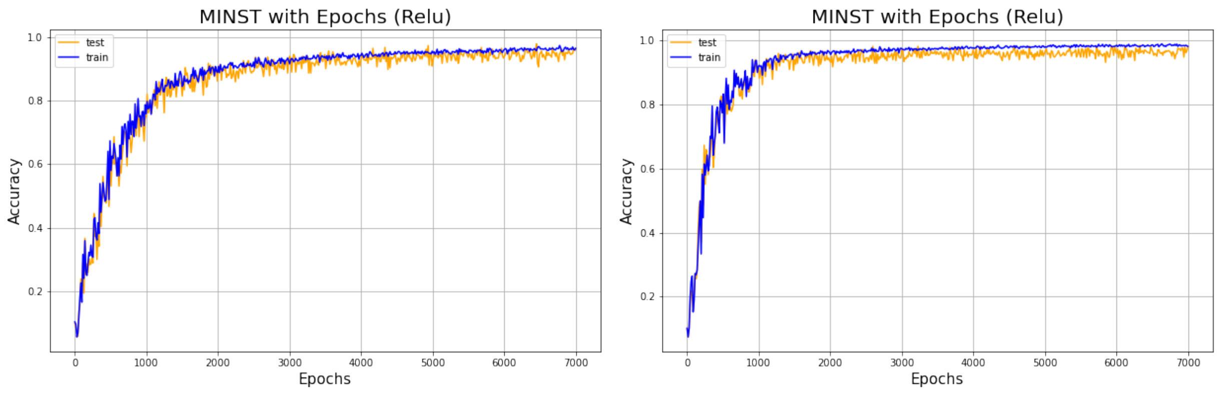


Figure 15: Stochastic Gradient Descent performance (MSE) for Train sets varying complexity degree (from 10 to 20 polynomial degrees) and the number of Epochs. Left column shows the initialization by *numpy.random.random()* function and the right is initialization with *numpy.random.randint()*.

To illustrate more the pre-processing importance, and since it will presented also in further discussions, the Figure 16 presents the difference between using two types of normalization. One pre-procesing (the left image) was made by subtracting to the dataset the mean value of the pixels and divide all by the variance, while the other one (the right image) was done by subtracting to the dataset the minimum value of the pixels and then dividing it by the difference between the maximum and the minimum values. It shows that the second normalization makes the training more efficient since the Beta NN can achieve scores superior to 0.8 at epochs below 1000 while the first normalization can only achieve them right after 1000 epochs.

For the performances of different activation functions with the MINST classification, more were taken



(a) Beta NN. (b) Keras NN.

Figure 16: Beta NN performance (Accuracy) for the MINST dataset classification while varying epochs. Left and right images presents different types of input normalization. Train and Test sets are indicated by blue and orange lines, respectively. Learning rate: 0.014. Architecture: One hidden layers (50 nodes), Activation function: ReLU.

into account. Now it is illustrated good performances between all activation functions (Figure 11). The best ones for converging near one where the Tanh and the ReLU functions and the fastest ones where also those. Tanh converges faster than all the other ones because it can approximate to a linear function near the value of 0, and this according to Sussillo and Abbott (2014) makes the algorithm to learn more efficiently, when the weights and the biases are initialized with small random values, which is the case in this exercise, both for Beta and Keras NN. We can also see, how the Tanh function can assimilate more to the Keras NN performance from Figure 10b. In such a way can also approximate more to the 1 accuracy score than the other activation functions.

Continuing with the analysis, the hidden layers and nodes also plays important roles in the performance of the NN. The Figure 12 shows the performance of the Keras NN varying the architecture, meaning the hidden layer numbers and the nodes in each layer. Another possibility would have been to made different nodes in each layer, but that computation would have been too expensive. The plots seems to show a relationship between number of hidden layers and an exponential increase in the number of nodes in response to the first parameter. For this classification purposes, simple architectures of 1 or 2 hidden layers work well. Adding more layers and nodes than the required would "overtrain" the model and performance will be worse for new data, as it is shown in the region of low accuracy scores.

The investigation of hidden layers and nodes were made with Keras NN because Beta NN didn't perform well for hidden layers superior to one. For more layers, than 1, Beta NN doesn't converge through epochs and shows low accuracy scores. We suspect that the part of this is due to the pre-processing of the data, which again, is something that was not really inder control, when working with Keras.

5.2.3 Breast Cancer dataset Classification

Similary, scaling and doing pre-processing improved the binary classification. Also, choosing the Tanh as the activation function provided the best performance for the Beta NN as it reached a high accuracy score in very few iterations. More of this discussion will be continued in the Logistic Regression section of this discussion.

5.3 Logistic Regression for Breast Cancer dataset

Here also the pre-processing of the dataset was crucial. For the logistic function to be properly applied to the dataset, the data must be scaled to be between 0 and 1, so that the code might adjust the probabilities in a proper way. There are some variations at the beginning of the iterations, but once it reaches a convergence value, the variation range decreases. The Logistic Regression performed better at the end than the Beta NN for a binary classification problem. It was expected that the performance of the Beta NN would have been way better than the Logistic Regression, based on what it was discussed before. Logistic Regression tries to create relationships and divisions of the target groups with their features by using linear regressions, which finally is forced into a Logistic function (Sigmoid) to map a probability. So by doing regressions, certain divisions might be not clear between certain features, however the NN can overcome this with some overfitting by not doing regressions with smooth functions, but with "jumps". We propose, that this result, on the contrary to the expected, might be due to the correct pre-processing of the information before feeding it to the NN.

On the penalty variations for the Logistic Regression: penalties doesn't have influence at higher number of iterations since the uncertainties are even lower and the coefficients of the regressions were improved until there is no need for a regularization. It is visible, that for the Test set the noise in accuracy scores are higher with the use of penalties than with the Train set (Figure 14) which is expected. Besides that, Figure 13b shows a "cross-section" of what is reflected in the heatmap (figure 14) for a fixed penalty.

This is more pronounced in the lost of influence of penalty values at higher number of iterations. Based the several discussions of penalty low influence at high iteration numbers, it can be claimed that the penalty influence has a relationship or happens due to the noise in performance during training. for high ranges of noise, the penalty values regulates more the performance, while at low noise, the penalty is not required anymore and does minor changes to the computation.

6 Conclusions

Keras function is a bridge to the clear understanding of the difference between the SGD-OLS fit and the NN. The SGD-OLS tries to find the optimal betas, then its trying to find a function that could fit the values. Because of this, the result function is smooth and continuously differentiable, so it will not allow too much the stochastic noise added. On the other hand, the NN will also take into account this noise and will try to perform the best to fit everything, included as much as possible, the stochastic noise of the data. Therefore, the "fit function" product would not be smooth and continuously differentiable, but with jumps and not realistic, and this would explain why NN performed better when trying to predict. Due to this, the decision for using one or another might depend if the objective is to find a mathematical smooth function that can describe it, or if the objective is just to predict, without caring of the way it is done. This last thing also explains why polynomial degree has no effect on the behavior of the NN training and prediction, since with a polynomial degree of 1 it could perform well.

Due to the stochastic procedure, SGD has the ability to compute several cost functions, since each time the mini-batch size is be different, and thus, could make the search to overpass a local minimum made by other mini-batch groups. Besides that, introducing the continuous iterations reduces the influence of the polynomial degrees more than what we expected. Concerning learning rates, the polynomial degree changes affects influences the chose of a good learnig rate, possibly due to the overfittings and local variations that could create easily local minimums for the mini-batches to try to detect them and trespass them to reach the global one. In addition, guessing the initial coefficients also makes the algorithm more efficient since it could be proximal to the optimal values, and therefore, giving good results in just the iteration number 1 as evidenced in this report.

Regarding NN: all models and tests showed that pre-processing is extremely important and determines the efficiency of the code/model, when training is done, both for regression or classification problems. In addition, the activation functions alongside with the way weights and biases are initialized, influence on the performance efficiency: less number of iterations required to achieve convergence on the training part. Tanh showed the best performance for the MINST data classification, because of the way the weights and biases are initialized (random small values). The function is monotonic and assimilates to a linear function with values closed to 0.

The penalty parameter in the NN works for only low epochs number, but becomes useless for high number of iterations since the NN is well trained and all the biases and weights were updated until convergence to their optimal values. And as an addition, regarding the network architecture, the number of hidden layers and nodes in order to perform a good classification were actually not as complex as expected, and so only one hidden layer of 50 nodes performed well.

In this assessment, Logistic Regression performed better for binary classification than the FFNN, which contradicts with what we expected, and we believe that this might have happened due to pre-processing of the data. The penalty parameters behaved in the same way as with the SGD-OLS. This was expected since the procedure is of the same nature as a gradient descent after evaluating a defined cost function. The study shows that the penalty influence could be due to the variations of noise ranges in the early iteration stages. When iterations are larger, penalty is not needed and noise variations in performance are reduced.

Further studies requires to play more with weights initialization, which will respond then with better activation functions rather than the ones that performed the best here in classification problems, and extension of using Logistic Regression not only for binary classification but multiple classes as well, in the same way as a NN can handle. Also an important topic to study better is the pre-processing of the data and how to address this depending on the type of data (image pixels, count populations, distributions in areas, etc.) in case of a own developed NN algorithm like the Beta NN. In addition, a good thing this report takes away from the previous one (Bazilova and Díaz, 2020) is that in this study, the data for Franke was shuffled so there were no tendencies of certain points at the beginning of groups to describe certain Z values rather than the general view of the population, and thus, it could also have influenced on the convergence efficiency for the SGD-OLS.

7 Code Availability

The code for this report can be found: `https://github.com/Doctus5/FYS-STK4155`, project2 folder. The folder includes the test runs and the 'readme' file.

References

Bazilova, V. and Díaz, S. (2020). Exploring linear regression methods: Validation and terrain model applications.

Dua, D. and Graff, C. (2017). UCI machine learning repository.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

Sussillo, D. and Abbott, L. (2014). Random walk initialization for training very deep feedforward networks. *arXiv preprint arXiv:1412.6558*.