Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 1

Learning and generalisation in feed-forward networks — from perceptron learning to backprop

Adrian Campoy Rodriguez, Gustavo Teodoro D. Beck and Nimara Doumitrou-Daniil

January 27, 2020

1 Main objectives and scope of the assignment

Our major goals in the assignment were: (1) learning and implementing feed-forward neural networks in a supervised setting, for both single and multi layer perceptrons; (2) successfully applying these methods in classification, function approximation, encoding and generalisation tasks; (3) analyzing their performances, identifying their limitations and trying to minimise them in order to create a more robust algorithm.

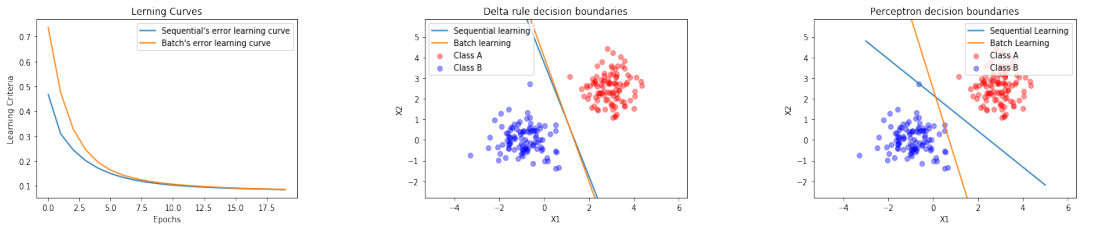
2 Methods

During this project the programming language Python 3 was used in three different IDEs such as Jupyter Notebook, VS Code and Spyder. Some particular toolboxes were used for specific tasks. For instance, SciKit Learn was used as an additional support to develop different methods applied along the project; TensorFlow and Keras were used in the last part of this assignment to generate different neural networks, as specified in the assignment description. Finally, the GitHub repository was used to store and manage different versions and pieces of code.

3 Results and discussion - Part I

3.1 Classification with a single-layer perceptron

The binary (**A** and **B**) classification of linearly seperable data was generated with different means and variances for each class ($meanA = [3.0, 2.5], meanB = [-1.0, 0.0], varianceA = 0.5, varianceB = 0.5$). In order to classify them, two learning methods with different weight update approaches were tested as seen in the following charts. In delta rule, batch and sequential learning converged in 20 epochs.



(a) Delta Rule learning curves. **(b)** Delta Rule Decision Boundaries. **(c)** Perceptron Decision boundaries.

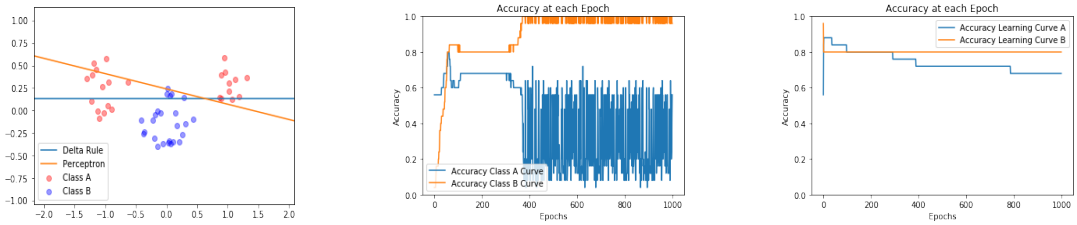
Figure 1: Decision Boundaries for Delta rule and Perceptron Learning (Batch and Sequential).

However, the perceptron sequential converged with 30 epochs, while batch learning converged with 50 epochs. We also observed that the initialization of the weights have a relatively important role when it comes to how fast the algorithms converge. Lastly, Delta rule converged to a line that

lied evenly in between the classes (increased chance of good generalization), while Perceptron terminates as soon as it finds one of the infinite solutions.

When it comes to analysing scenarios without adding bias, the only cases where perceptron will classify the classes correctly are when the classes are linearly separable by a line that passes through the origin.

Next, we examined the performance of the algorithms on linearly non-separable data. The single-layer perceptron struggled to find the non existent separating hyperplane (notice how it oscilates 2b), while the delta rule based algorithm finds one that optimized the trade-off (square error). The following analysis is based on splitting the data set into training and testing, being the latter a sub-sample of 25% the data set, sampled in four different ways: (1) random 25% from each class; (2)random 50% from class **A**; (3) random 50% from class **B**; (4) 20% from a subset of class **A** for which class $\mathbf{A}(1,:)<0$ and 80% from a subset of class **A** for which class $\mathbf{A}(1,:)>0$. The following example corresponds to the single-layer perceptron and the delta rule for the first case:



(a) Boundaries of Perceptron and Delta rule. **(b)** Perceptron performance curve of class A and B. **(c)** Delta rule performance curve of class A and B.

Figure 2: Delta rule and Perceptron upon non-linearly seperable data set.

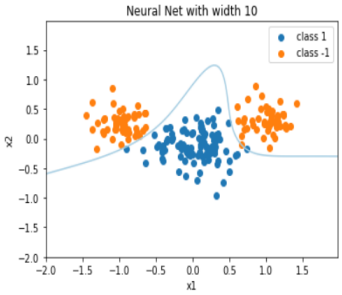
Analyzing the different sub-samples was rather illuminating and showcased different aspects of this algorithm. Firstly, the decision boundaries varied depending on our training data. However, this effect was very evident when the subsampling was biased (not evenly sampled from each class), as the performance of our classifiers was lacking for the subrepresented class. This was especially evident in the fourth case, where it struggled in characterizing inputs from class A with positive features (as we excluded 80% of them from training).

3.2 Classification and regression with a two-layer perceptron

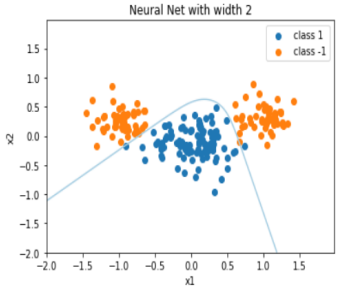
3.2.1 Classification of linearly non-separable data

Multi layer perceptrons, utilizing non linear activation functions, are able to capture more complex decision boundaries. To test this claim, we generated linearly non-separable data produced as described in the previous section, generating the decision boundary seen in figures 3a and 3b. We examined the effect of the width of the hidden layers on the train error, as well as plotted the training and validation learning curves. We ran these experiments for multiple configurations-splits (4b refers to the second type of split, as described in the assignment) of the train-validation and we witnessed similar results. Only in the fourth case, where our validation consisted of only one class, we obtained a different validation curve (it followed a similar trend, however it lay much higher than the training error). Increasing the width of the hidden layer lowered the average error, however the curve remained the same (qualitatively).
As seen in 4b, initially, learning and validation curves follow a similar downward trend. However, as we continue training on the data, our model overfits, as it begins memorizing the training inputs at the expense of generalization (notice how the validation error increases). We also compared sequential and batch learning in this environment, and noticed that sequential learning showcased increased validation error (0.002 increase), since the gradient on a single sample $\nabla(f(x_i))_{x_i \in \text{train}}$ is a poorer estimator for $\nabla(f(x))_{x \in \text{test}}$ than $\frac{1}{N}\sum_n \nabla(f(x_n)_{x_n \in \text{train}})$. Note that

the magnitude of the effect of the batch size is problem specific, and it is possible that in different scenarios the validation error difference could be more significant.

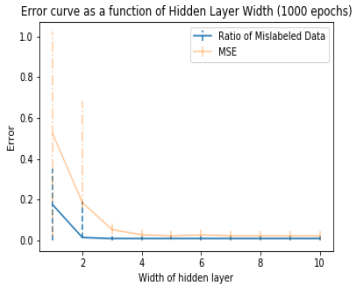


(a) Decision boundary for width 10.

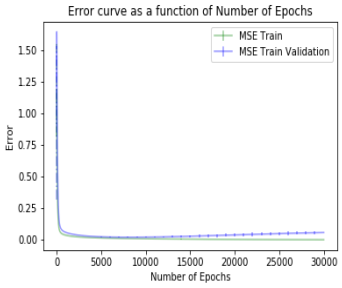


(b) Decision boundary for width 2.

Figure 3: Decision boundary for different widths of the Hidden Layer. Increasing the width leads to more complex decision boundaries. Note, however, that similar boundaries with (left) were achievable utilizing fewer hidden nodes.



(a) Error as a function of the hidden Layers.



(b) Validation and Training curve for Neural Network of width 10.

Figure 4: As expected, (left) increasing the width of the hidden layer decreases the average error, as the model's capacity increases. The right graph illustrates the notion of overfitting. Similar curves were obtained when examining the ratio of mislabeled data. To account for the random initializations, all experiments were conduced multiple times (mean and std presented in 4b, while median and quartile in 4a).

3.2.2 The encoder problem

One characteristic application of Artificial Neural Networks is for encoding data to a lower dimensional space. Such an encoding is often opted for when compressing data, or if we wish to find the underlying lower dimensional manifold of our data (e.g. images of digits or faces).

In our experiments, we tried to encode the finite, discrete space

$\mathcal{D} = \{e^i \in \mathbf{R^8} | e^i_i = 1, e^i_j = -1, i \in \{1, 2, ..., 8\}\}$ into our two (8-2-8) and three (8-3-8) dimensional space induced by our hidden layer. Since the hidden nodes have continuous values ($\mathbf{R}^2$, $\mathbf{R}^3$), such a mapping is always obtainable (we simply map 8 elements to a continuous space). This, however, does not ensure (by itself) convergence, as it may still get stuck in local minima. Nevertheless, we did not witness a diverging initialization. Furthermore, lowering their dimensions (8-2-8 vs 8-3-8) did impact the speed for which the convergence was obtained. For convergence, we check whether the rounded outputs match the corresponding inputs.

Interestingly, for the case of (8-3-8), we obtained (focusing on their sign):

$$h_1 = (+,-,-), h_2 = (-,-,+). h_3 = (+,-,+), h_4 = (-,-,-)$$

$$h_5 = (+,-,-). h_6 = (-,+,-). h_7 = (+,+,+), h_8 = (-,+,+)$$

It, therefore, encodes our eight samples in their 3-binary representation.

3.2.3 Function approximation

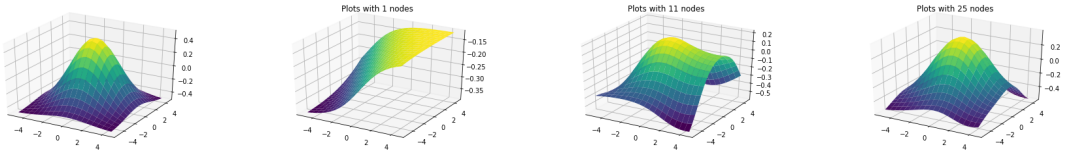
In this section, the aim was to approximate a Gaussian function: $f(x, y) = e^{-\frac{\left(x^2+y^2\right)}{10}} - 0.5$ The data set was composed of 400 data points with two features which are the coordinates X and Y. The bell shaped Gaussian function was the target to be approximated. The training data set was composed of 100 random data points and the remaining 300 were used as test.The number of nodes in the hidden layer was varied from 1 node to 25. This process was repeated 20 times with different initialization values of the weights in order to obtain the MSE plot for each width of the hidden layer. The MSE was calculated over all the data points. The criteria to select the best model was the smallest MSE (see figure 5a).

(a) MSE of function approximation with increasing number of nodes in the hidden layer

(b) MSE for Neural Network with one hidden layer and 24 nodes width with increasing training data.

Figure 5: MSE for increasing nodes in hidden layer with fixed training set and increasing training set with fixed nodes in hidden layer.

It can be observed from figure 5 that the higher the number of nodes in the hidden layer, the less the MSE. Also, this can be appreciated in the approximations that the network was making. With very few nodes, the approximation was rather basic and the network was not able to grasp enough information as to correctly reproduce the function. However, when increasing the number of nodes, the complexity that the network was able to reproduce increased and, therefore, the output was proportionally similar to the original function. The minimum is found to be around approximately 24 nodes, therefore the model with 24 nodes was selected as the best model. Moreover, the size of the training data set was changed from 80% to 20% of all the data set in pieces of 20 data points. As before, the algorithm was tried 20 times for each size with different weight initializations. The results clearly show that, as expected, the larger the training data size, the better the results (see figure 5a). On the other hand, in order to speed up convergence without compromising the generalisation performance, the learning rate η can be increased in such a way that the convergence occurs faster although this has the risk that, while training, the model may fail to find the optimum minimum. Using momentum was a good approach, since the weights are not modified only with the current update but also with the previous update, in such a way that if the model is on its way to a minimum, momentum speeds up the convergence towards that minimum.



(c) Original bell shaped Gaussian function

(d) Approximation with 1 node

(e) Approximation with 11 nodes

(f) Approximation with 25 nodes

Figure 6: Approximations to Gaussian function with increasing number of nodes in the hidden layer

4 Results and discussion - Part II

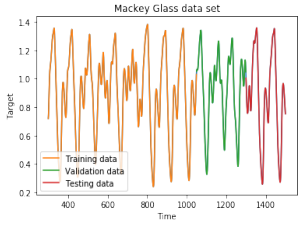
4.1 Two-layer perceptron for time series prediction - model selection, regularisation and validation

This section involved working with Mackey-Glass chaotic time series. In figure 7a the data and the shapes of the training, validation and test data sets are shown.
A 2 layer neural network - modeled with TensorFlow and Keras framework - was used on the aforementioned data in order to accomplish the specified tasks. The best neural network with batch learning (batch size 50 samples) was obtained via grid-search with two hyperparameters: the width the hidden layers (1,4 and 8) and L_2 Regularization factor ($10^{-3}, 10^{-1}$ and 1). The results are summarized in table 1:

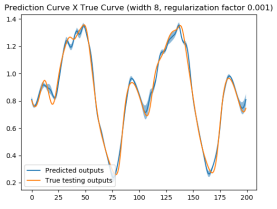
		Number of nodes (2 layers)			Number of nodes (3 layers)		
		1	4	8	1	4	8
Reg. factor	0.001	0.466	0.008	0.004	0.535	0.358	0.095
	0.1	0.385	0.017	0.106	0.629	0.017	0.190
	1	0.399	0.037	0.035	0.548	0.290	0.199

Table 1: MSE error for each neural network configuration in the validation set after 300 epochs (indicated by early stopping) of training for two-layer and three-layer perceptrons (three-layer perceptron results were obtained with noise $\sigma = 0.03$). Note: The width of the first hidden layer of the 3 layer neural network is equal to the width which yielded the best results for the 2 layer neural network.

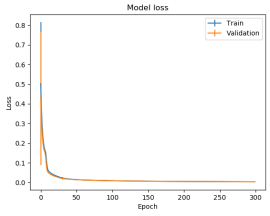
Based on the MSE error, the 2 layer model with a hidden layer of width 8 and 0.001 regularization factor was selected as the best model, while the second layer of the 3 layer model is configured with 4 nodes and 0.1 regularization factor. The resulting Mackey-Glass approximation can be observed in figure 7b as well as the resulting learning curves 7c.
It is also observed in table 1 that, in general, the larger the regularisation factor, the larger the error. We consider that this is due to the fact that, since early stop is done, the neural network does not reach an overfitting stage. Nevertheless, because we are adding the regularization term, we are reducing the complexity of the model. This is helpful when the model has overfitted but, since the model has not yet overfitted, we are reducing complexity on a rather still simple model. This gives as a result a neural network which is not capable of correctly reproducing the expected output. It is however expected that, if the neural network is left to run for a large number of epochs, it will overfit and then the regularization factor will play a crucial role by minimising the weights, making the model less complex and being able to generalize better. Thus, higher regularization factors will produce better results than small regularization factors, unlike what is observed now.



(a) Mackey-Glass without noise.



(b) Neural network predicted values.



(c) Training and validation curves.

Figure 7: Best two-layer perceptron.

L2 weight regularization forces a penalization (cost) on the L_2 norm of the weights, by adding the sum of their squared values to the error term. This in turn restricts the models' complexity, which aids on avoiding overfitting. Figures 8 and 10 show the effect of the regularization term, where a higher factor makes the weights approximate to zero and, thus, the model becomes less complex. Moreover, notice how in figure 9 the output layer shows a higher variance (compared to figures 8 and 10), since the weights in this layer weren't regularized. If they were, the weights

would yield to zero and then predicting a straight line instead of representing the chaotic time series.

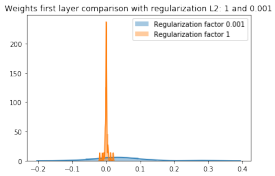


Figure 8: Histogram of weights in first (input) layer of two-layer perceptron.

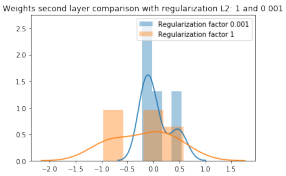


Figure 9: Histogram of weights in third (output) layer of two-layer perceptron.

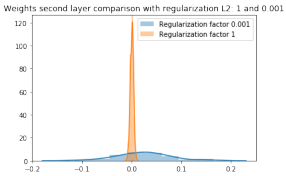


Figure 10: Histogram of weights in second layer of three-layer perceptron.

4.2 Comparison of two- and three-layer perceptron for noisy time series prediction

In the case of the three-layer perceptron for time series prediction, the same procedure as before was followed. A grid search was performed with the same hyperparameters as in the previous section. The results of the approximation with different noise and regularization factors can be observed in figures 11, 12 and 13.

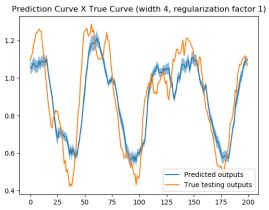


Figure 11: Best 3 layers model prediction with noise 0.03.

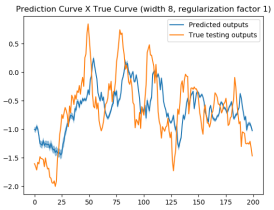


Figure 12: Best 3 layers model prediction with noise 0.18.

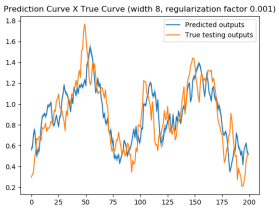


Figure 13: Best 3 layers model prediction with noise 0.09 and regularization factor 0.001

The same analysis with noise was made with the 2 layers neural network and the performance were similar. However, the standard deviation of the 2 layers model was higher, indicating that the extra complexity (layer) increases the variability of the model and describes the data set more accurately in these noisy situations. In addition, it can be observed in table 1 that, in general, the number of nodes and the MSE have an inverse proportional change in both type of neural perceptrons (two-layer and three-layer perceptron). Also, it can be seen that the lower the regularization factor, the lower the error. This may indicate that it is necessary to have a complex neural network (and, thus, a complex set of weights) in order to correctly approximate the data set. However, by increasing the regularization factor we expect that model complexity will be reduced and with more epochs this model with have a lower MSE error then the others.

Finally, we examined the computational complexity of these models, and witnessed that increasing the number of layers (weight matrices) from 2 to 3 (50% increase) resulted in a similar 50% increase in computation cost for the aformenetioned grid search (18 minutes and 55 seconds for the two-layer perceptron and 27 minutes and 7 seconds for the three-layer perceptron)

5 Final remarks

This lab's completeness (numerous experiments) helped us to grasp the theoretical concepts explained during the lectures. It also served as a good exercise to become familiar with the scientific and concise format of reports that is usually requested in the workplace. In addition, apart from implementing delta rule and perceptron learning (single and multi-layer), we had the opportunity to test one of the most used framework (Keras) for neural networks formulation. On the other hand, we believe that more practical and real life data sets could have been tested, instead of the aforementioned toy examples.