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

Leonidas Valavanis, Polydefkis Gkagkos and Kai Ma

February 1, 2019

1 Main objectives and scope of the assignment

Our major goals in the assignment were

- to design single-layer networks using Perceptron Learning and Delta rule
- to design multi-layer networks using the generalized Delta rule and back-drop
- to apply the networks in classification, function approximation, regression (time series) tasks
- to find limitations and risks of the networks
- to experiment with the parameters and understand the behavior of the networks

The lab

The lab focuses on understanding in depth how the covered topics work and at the same time implement basic algorithms.

2 Methods

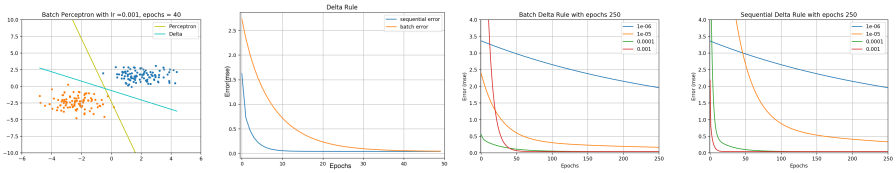
We have used python to implement our code. For part **2**, we used Keras to implement the MultiLayer Perceptron with backend Tensorflow. We used many different parameters and displayed the results in Section 4.

3 Results and discussion - Part I

3.1 Classification with a single-layer perceptron *(ca.1 page)*

In Figure 1 (a) the final decision boundaries between the Perceptron Learning and the delta rule is visualized for our linearly separable dataset. As shown, the Delta rule algorithm has a more generalized boundary than the perceptron learning. The sequential mode of the delta rule algorithm converges faster than batch mode as the learning rate increases, as shown in Figure 1 (c) and (d). The Batch mode makes significant steps and is prone to get stuck in a local minimum, nevertheless, the sequential mode takes smaller steps. Moreover, Delta rule is susceptible to weights initialization, and depending on the weights it can converge from the first few iterations or after iteration number 30.

Removing the bias from the weights results in a slower convergence of the algorithm or no convergence at all even if the data are linearly separable. Without the bias, if the data are not separated in origin, then the algorithm will not find a solution, regardless the learning rate or the number of iterations.



(a) (b) (c) (d)

Figure 1: Single Layer Perceptron Learning Curve with $\eta = 0.0001$ Batch Approach and Delta Learning Rule

We tested with non-linear separable datasets with delta learning. In four cases of reduced data sets, perceptron was not able to separate two data classes. Based on our observations, the generalization is not affected by the resulting sampling bias.

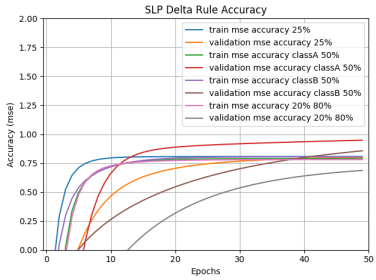
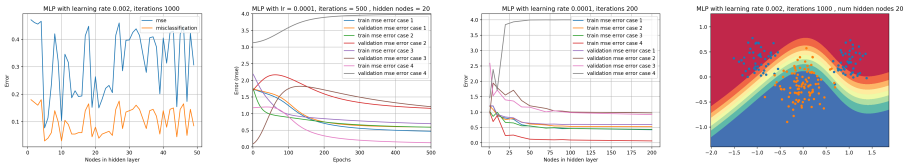


Figure 2: Accuracy Curve $\eta = 0.001$ with Batch mode and Delta Learning Rule

3.2 Classification and regression with a two-layer perceptron *(ca.2 pages)*

3.2.1 Classification of linearly non-separable data

As the number of nodes in the hidden layer increase, the mse should drop to zero since the model overfits. Because our data cannot be perfectly separated, the mse and misclassification cannot drop to zero, but it is close to zero, as shown in Figure 3 (a). The fluctuations may be explained because the algorithm is stuck in a local minimum. The learning curves for the several datasets is similar, but in case 4 there is great dissimilarity, as shown in Figure 3 (b), where the train is small, and the validation is increasing which means that the generalization is poor and the model overfits. Moreover, as the number of nodes increases in Figure 3 (c) the error increases except dataset case 4 again in which the model does not generalize. The decision boundary is shown in Figure 3 (d).



(a) (b) (c) (d)

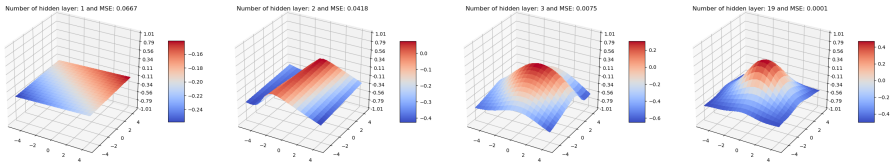
Figure 3: (a) Effect of the size of the hidden layer on the performance.(b) MSE error curves for the training and the validation sets depending on the epochs. (c) MSE error curves for the training and the validation sets depending on the size of the hidden layers. (d) Decision boundary produced by MLP.

3.2.2 The encoder problem

The autoencoder tries to generate from the reduced encoding a representation as close as possible to its original input, hence its name. So the purpose of autoencoders is dimensionality reduction, amongst many others. An autoencoder with three nodes in the hidden layer will always converge and will optimize the weights to replicate the initial data given. This convergence strongly depends on the learning rate and on the number of epochs the Multilayer Perceptron uses. After checking the internal code for the activations of the hidden layer, we discovered that in a few iterations the patterns start to resemble a binary format. The same thing happens with the weights matrix for the first layer. If we use a strictly binary activation function, then the patterns tend to give us all the possible combinations of 2^3, but eventually, the same thing happens for the sigmoid activation after a high number of iteration. Regarding the number of nodes in the hidden layer, if we use two nodes with bias, then we can represent the actual input data, but if we skip bias, then the model cannot represent more than 2^2 binary numbers, so we get a misclassification error of 35%.

3.2.3 Function approximation

Varying the number of nodes in the hidden layer from 1 to 25 gave a good insight into the learning process. More specifically, a model with 1 node in the hidden layer was able to represent only the mean of the data in a flat dimension, with 2 nodes variance is added and the approximation becomes a parabola, with 3 layers we add the concept of skewness and the output tends to become Gaussian (which reminds us of the original input) [**?**]. By using 4 or more nodes in the hidden layer, the model steadily decreases the mean squared error, and the approximation becomes more accurate. We conclude that using more than 20 hidden nodes makes the model overfit the input data because when we added noise, it was able to approximate even the noise of the input function.



(A) (B) (C) (D)

Figure 4: Model approximations with different nodes in the hidden layer.

Using 25 models with the same hyperparameters except for the number of hidden nodes gave us the results on the figure (5) where we can compare the mean squared error (mse) of the models. As we explained above, increasing the number of nodes improves the mse of the model. We have an expected enormous improvement on the first five models and as the number of nodes increases the error steadily decreases. The "best" model regarding computational cost/complexity can be considered the 5th because of the trade-off between the mse and the number of hidden nodes. The mse, in this case, is relatively low and the number of hidden nodes is also small. Experimenting with various input training sets yield the results on figure (6). The outcome is to be expected as our model has fewer data to learn from on every split hence the increase on the mse. The performance can be increased by tuning some hyperparameters like the learning rate and the momentum of the model without affecting the generalization of the model.

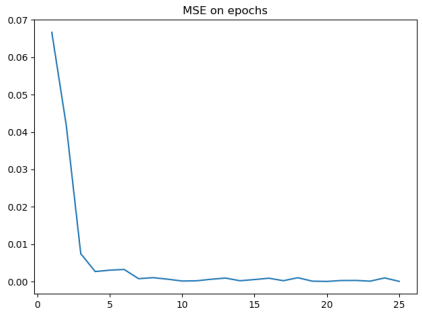


Figure 5: Models comparison with error estimation.

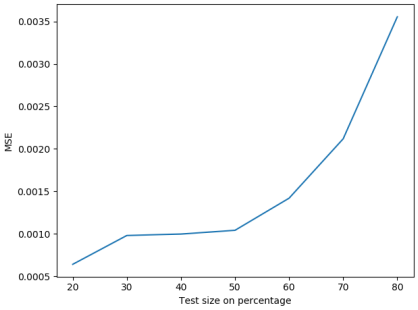
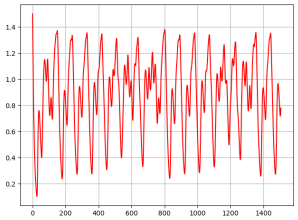


Figure 6: Model comparison with different train sizes

4 Results and discussion - Part II *(ca.2 pages)*

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

In the right figure, we illustrate the original input of the time series data.

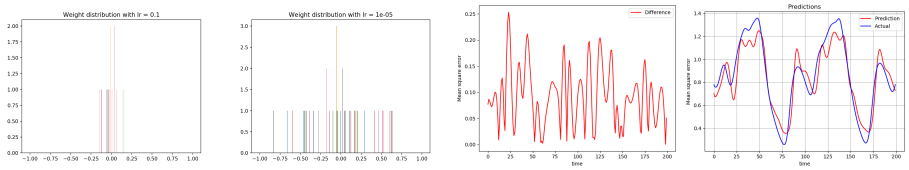


For the experiments, many different configurations were used. After experimenting, we conclude that the L2 norm is a better regularizer than L1 as it achieved substantially better results. Moreover, the best number of nodes in the first hidden layer is 4 with mse error 0.11 on the test data as shown in the below table. We also used early stopping with the patience of 10-15 epochs to stop the training.

Epochs	Val split	n Nodes	Batch Size	Regularizer	Optimizer	Metric	Pred loss
5	0.1	5	300	L2(lr=0.01)	SGD	mse	0.1284
5	0.1	5	300	L2(lr=0.0001)	SGD	mse	0.0821
5	0.2	5	50	L2(lr=0.0001)	SGD	mse	0.0615
5	0.2	8	50	L2(lr=0.0001)	SGD	mse	0.0354
40	0.35	8	32	L2(lr=0.0001)	SGD	mse	0.0259
500	0.35	8	32	L2 (lr=0.0001)	SGD	mse	0.0125
500	0.35	7	32	L2 (lr=0.0001)	SGD	mse	0.0142
500	0.35	6	32	L2 (lr=0.0001)	SGD	mse	0.0139
500	0.35	5	32	L2 (lr=0.0001)	SGD	mse	0.0154
500	0.35	4	32	L2 (lr=0.0001)	SGD	mse	0.0111
500	0.35	3	32	L2 (lr=0.0001)	SGD	mse	0.0151

The weight distribution with different regularization values is shown below in Figure 7 (a), (b). As expected, when the regularization value increases, the weights drop to zero. Thus, regularization suppresses the weights to zero and

decreases the variance of the model. When predicting the test dataset the values are similar to the targets as shown in figure Figure 7 (d) and the difference is shown in Figure 7 (c).



(a) (b) (c) (d)

Figure 7: Weight distributions and predictions on test dataset

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

We added noise only on the input dataset. As shown in Figure 8 the training error is larger than the validation for different values of the regularization. That means that the noise generalizes the model better. From our results as the noise increases the error increases although it should be the opposite since the model would be more robust. More noise, augments the training data making the model more robust, and the difference between the validation and train error increases as the noise increases. Moreover, increasing the regularization may decrease the validation error since it penalizes the weights and prevents overfitting. When adding noise with small regularization, the network learns the noise and the weights have significant variance. Thus, with noise, the regularization should be larger, so the weights are distributed around zero and overfitting is avoided.

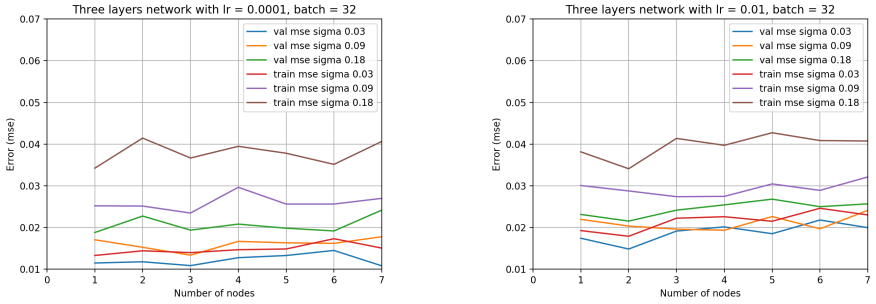


Figure 8: Validation and train error with different regularization

The best results for noise $\sigma = 0.09$ is **4** nodes in the first hidden layer and **3** in the second. The results were slightly better when we added regularization in both layers. A noise term was added only on the train data. Concerning generalization, when adding noise, the validation and test error is smaller than the training error on both networks. The two-layer network is slightly better.

Both models generalize pretty well. The complexity cost (time) of the network is depending on the epochs, the number of training data, the number of nodes and the number of layers. Of course, adding an extra layer to the model would increase the time complexity.

5 Final remarks *(max 0.5 page)*

The lab overall was very useful and gave us excellent insight into the learning process behind the single and multilayer perceptron. However, some of the questions could be more clear (such as complexity cost/time, where to add noise on the dataset) and save time to do the actual work and understand concepts in depth.