Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 1b

Learning with backpropagation and generalisation in multi-layer perceptrons

Alex Norlin, Athanasios Charisoudis and Giovanni Castellano

February 8, 2022

1 Main objectives and scope of the assignment

In this lab, we studied multi-layer perceptron networks. In particuar, we implemented the backpropagation algorithm in order to perform classification tasks on different types of data. Our major goals in the assignment were :

- to implement the Backprop algorithm (batch and sequential)
- to study generalisation capabilities of the model
- to understand how autoencoders work
- to understand how different number of hidden nodes affect the performance of the model
- to approximate continuous functions using neural networks
- to learn how to choose between different models
- to predict time series with MLPs and study how noise affects learning

During the lab, we only studied two-layers and three-layers MLP. Moreover, most of the experiments were performed using batch Backpropagation.

2 Methods

All the experiments were implemented using Python. The most important libraries we used are: Numpy, Matplotlib, and Pytorch.

3 Results and discussion

3.1 Classification and regression with a two-layer perceptron

3.1.1 Classification of linearly non-separable data

To begin, we generated some non-linearly separable data from some Gaussin distributions. Afterwards, we tried to classify them using a two-layers neural network with 3 hidden nodes. As we can observe in figure 1(a), we got a good decision boundary, although not all the points are correctly classified. We also tried to perform the same task using different numbers of hidden nodes. Figure 1(b) illustrates the MSE for each case; when we use just one hidden neuron the performance of the network is very bad, but two hidden nodes are enough to get a low MSE. Even increasing the hidden nodes, the MSE is never zero for this dataset. This could be because the algorithm is likely to get stuck in local minimums.
To test the generalisation capability of the model, we removed a subset of the training set and used it as validation set according to the following 3 cases:

1) random 25 per cent from each class
2) random 50 per cent from class A
3) 20 per cent from a subset of classA for which $classA(1,:) < 0$ and 80 per cent from a subset of classA for which $classA(1,:) > 0$

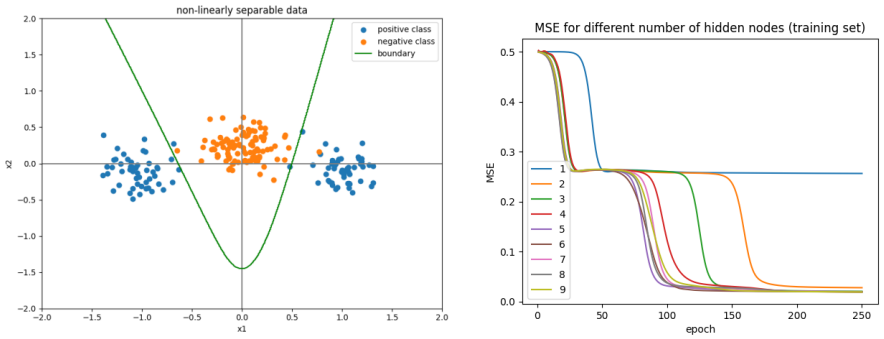

Figure 1: (a) Decision boundary generated using a MLP with 3 hidden nodes. (b) MSE for different number of hidden nodes. (alpha = 0.9; learning rate = 0.05;)

For the first two cases, we got good results, and the boundary was very similar to figure 1(a). However for the third case, we got a high validation error; the result improved using sequential backprop, although we know that this depends on the initial conditions and the order of the data.

3.1.2 The encoder problem

In this part of the lab, we trained a symmetric network to output the same sample given as input; such kind of networks are called autoencoders and they can be used to perform different tasks such as: dimensionality reduction, anomaly detection, and images denoising.

We tried to interpret the values assumed by the hidden nodes of an autoencoder for different input samples. In particular, we built a neural network with 8 input-output nodes, and 3 hidden nodes; subsequently, we trained it with a dataset generated by the function *np.eye(8,8)*. The first thing we noticed is that the output of the network is not exactly equals to the input but it is a good approximation; In other words, the MSE after training was very low but not zero. Looking at the hidden nodes, we found a very interesting pattern in the signs of the values; for each input sample the hidden nodes assumed a different configuration of signs (e.g., sample 1: -+-, sample 2: +-+, sample 3: -++, ...). We also observed a similar pattern in the weights matrix. Such encoding is very similar to binary encoding; we also notice that to encode 8 different values we need 3 bits ($2^3 = 8$), and that is why we get such good results using 3 hidden nodes.

Decreasing the number of hidden nodes to 2 it becomes harder for the network to encode the dataset; we trained the network many times and in most cases we got a much higher MSE than using 3 hidden nodes. Moreover, the signs of the values in the hidden nodes did not assume a particular meaning (in fact using two bits we can just encode 4 numbers).

3.1.3 Function approximation

There was a huge difference in performance when going to a very small number of hidden nodes while there wasn't a big change when going from 20 up to 25 hidden nodes. With 1 or 2 hidden nodes you can clearly see the underlying sigmoid activation functions. For the models with a higher count of hidden nodes, the results looked almost identical to that in figure 3 (a). In order to choose a *best* model we ran multiple test with each model (learning rate = 0.05, alpha = 0.9, 50% of all data as training data) and averaged their final MSE of the total data and looked at their approximation (and the absolute error of it, see figure 3 (b)). The result was that, on average, the 25 node network performed the best with a MSE of 0.00020 ± 0.00019 over 100 runs. The difference in performance were however not that significant when the number of nodes were higher than, around, 15. Meaning that if one would want to prioritize computation time one might opt for the smaller network.

Then we ran tests on the network with 25 hidden nodes this time with a varying amount of training data, from 80% of all data down to 20%. The result was not that surprising, the more training data the lower the MSE on the total data. This should be the case as the total data is the training + validation data.

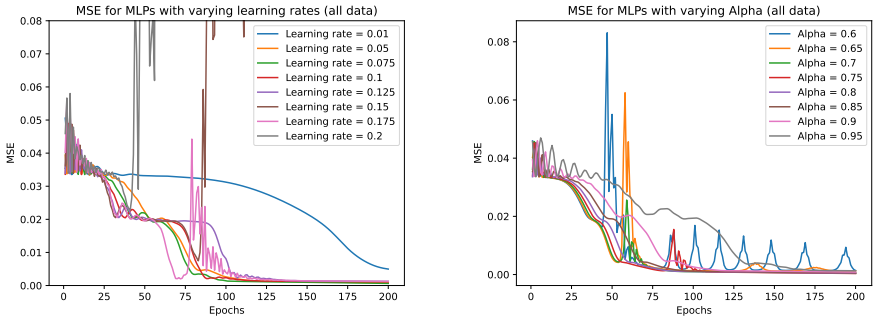

Figure 2: (a) MSE on all data for the previous best network with varying learning rates. (b) MSE on all data for the previous best network with varying alpha values.

Lastly we used fine tuned the hyperparameters of the previously best model to speed up convergence. As can be seen in figure 2 there wasn't much improvement to be done by increasing or decreasing the learning rate alone, but by lowering the alpha, which controls how much impact the old weights have on the new update, to 0.8 we can speed up convergence a bit. To reduce it to a even lower value would run the risk of making the convergence worse as the MSE could explode as can be seen in the case alpha 0.7 for example. The best network tested achieved an average MSE of 0.00011 ± 0.00006 with 25 hidden nodes, a learning rate of 0.05, an alpha of 0.8 and trained on 80% of the available data.

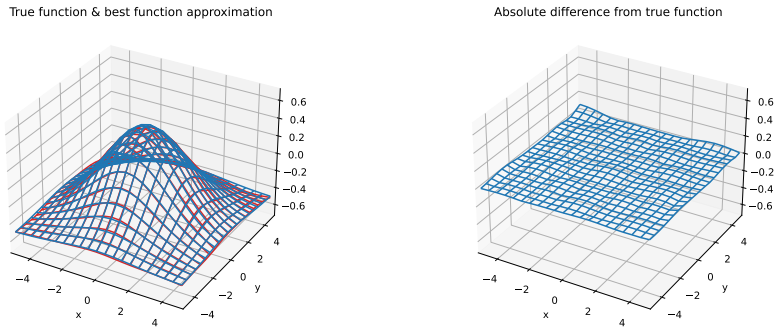

Figure 3: (a) True function (blue) and the network approximation (red). (b) The absolute difference between the plots in (a).

3.2 Multi-layer perceptron for time series prediction

For the purposes of this part, PyTorch framework was used to define and train a 3-layer perceptron network. The entire dataset comprised 1200 samples of the Mackey-Glass time series (after it has reached a pseudo-stationary behaviour; in our case we ignored the first 280 samples) and was split to 850-150-200 samples to form the training, validation and test sets respectively. All models were trained for 2000 epochs wherein each the entire training set was used (batch training). If not otherwise mentioned, early stopping was used, to ensure overfitting prevention, with number of rest steps equal to 30. The weights (i.e. weights and biases) were initialized from a zero-mean Gaussian with an externally set and (usually) varying std. All models were trained using the Adam optimizer with $lr = 0.01$ and $\beta_1 = 0.9, \beta_2 = 0.999$.

3.2.1 Three-layer perceptron for time series prediction - model selection, validation

In this sub-section, the validation set was used to tune model hyperparameters. In particular, grid search was performed on the number of hidden units ($nh_1 = \{3, 4, 5\} \times nh_2 = \{2, 4, 6\}$; 9 combinations) where the validation set provided an estimate of the goodness of each model. We also accounted for the effect of random weight initialization as explained in what follows.

Single-Run Model Selection Initially, we trained each model configuration (i.e. for a given number of units in each hidden layer) only once. After each training for 2000 epochs, each model was evaluated using the validation set and based on these evaluations the best and worst performing architectures were selected. In this first experiment, initialization σ was fixed to 0.01. The best performing model architecture was $(nh_1, nh_2) = (4, 2)$, validation MSE 0.00132 and test MSE 0.00089. On the other side, the worst performing one was $(nh_1, nh_2) = (4, 6)$, validation MSE 0.00228 and test MSE 0.00216 (almost doubles). What was pretty clear from multiple runs of this experiment (varying std), was that *the selected architectures were not robust and highly dependent on the weights initialization* (i.e. we got different best model combinations, such as $(5, 2)$ or $(4, 4)$ and correspondingly for the worst models).

Multiple-Run Model Selection To alleviate the aforementioned issue and derive a more robust model selection scheme, for each combination of the number of hidden units we trained the corresponding models for 4 initialization-stds ($std = \{0.01, 0.05, 0.1, 0.2\}$ and 10 runs for each std; leading in 40 runs per configuration. The average validation MSE over these 40 runs was used as each model's test performance estimate, which in turn was used to select the best configuration. This procedure results in the best performing model architecture once again being $(nh_1, nh_2) = (4, 2)$, and exhibiting validation MSE 0.00066 ± 0.00144 and test MSE 0.00051 ± 0.00267. Furthermore, the worst performing one is now $(nh_1, nh_2) = (5, 6)$, exhibiting validation MSE 0.00809 ± 0.00214 and test MSE 0.01052 ± 0.003. In figure 4, real vs predicted test set values are given.

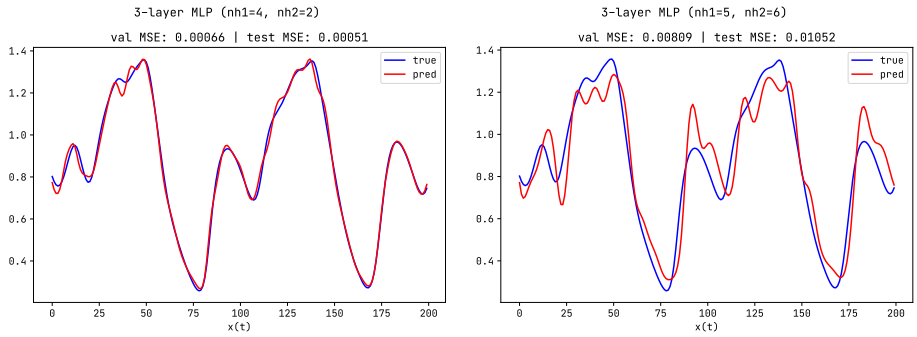

Figure 4: Test set performance of the best (left) and worst (right) models.

	$\sigma = 0.05$		$\sigma = 0.09$		$\sigma = 0.15$	
	best	worst	best	worst	best	worst
h_1	**4**	4	**5**	3	**4**	4
h_2	**3**	9	**2**	2	**9**	3
λ	**0.10**	1.00	**0.10**	1.00	**0.50**	1.00
val MSE	**0.00906**	0.01726	**0.02081**	0.02215	**0.04470**	0.04731
test MSE	**0.00630**	0.01934	**0.02565**	0.04107	**0.04875**	0.05917

Table 1: Best and worst model architectures and their performance for a fixed number of first hidden layer units (4), for varying additive noise σ and weight decay λ.

3.2.2 Three-layer perceptron for noisy time series prediction with penalty regularisation

Mandatory Task Subsequently, we add zero-mean Gaussian noise to the time series samples with $\sigma = 0.05$. Then, in the first part of this sub-section, we fix the number of units in the first hidden layer to $nh_1 = 4$ and perform grid search on nh_2 in the values $3, 6, 9$. We then repeat this grid search when noise σ increases from 0.05 to 0.15 and compare the best found model to the previous one. Finally, we use Weight Decay (L2) regularization to prevent overfitting to noise components and skip early stopping. Weight decay mixing coefficient, λ is inserted directly in the Adam optimizer (instead of as an extra loss term), forming the so-called $AdamW$ optimizer presented in [1]. We searched for $\lambda \in \{0.01, 0.1, 0.5, 1.0\}$. In all of our experiments we once again used the hold-out validation set as a measure of model performance and for model selection.

What can be seen from Table 1, is firstly that in the presence of low-variance (0.05) additive WGN, still the best architecture with $nh_1 = 4$ is one with less hidden units in the second hidden layer (i.e. 3), but when σ increases more units are needed to account for the extra capacity needed. In addition, when noise σ

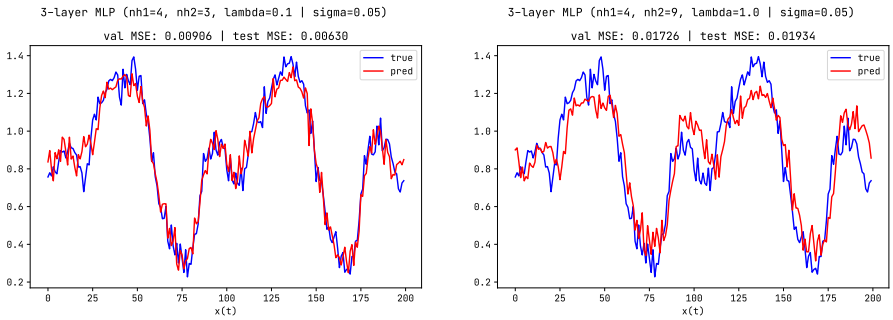

Figure 5: Test set performance of the best (left) and worst (right) models for additive WGN with $\sigma = 0.05$.

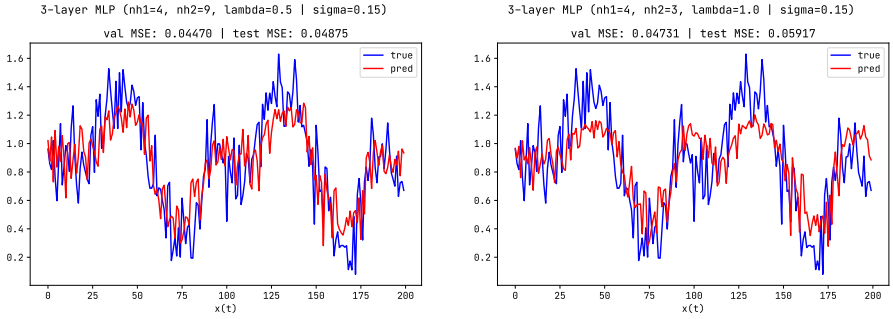

Figure 6: Test set performance of the best (left) and worst (right) models for additive WGN with $\sigma = 0.15$.

is small, increasing the weight decay leads to poorer performance (probably due to limiting model expressivity), but when σ increases, weight decay λ should increase as well to limit overfitting to noise. Actual predictions of the best and worst model architectures can be seen in figures 7 for $\sigma = 0.05$ and 7 for $\sigma = 0.15$.

Non-mandatory Task In this last task, we performed grid search on $nh_1 \times nh_2 \times \lambda$ (as in 3.2.1), once again using 40 runs/combination. The noise σ was set to 0.09 and the corresponding best model and its performance can be seen in Table 1. Next, we re-trained the best and worst performing models of 3.2.1 and tested them in the noisy data, without weight decay. The best model of 3.2.1 ($nh_1 = 4, nh_2 = 2$) exhibited 0.02099 validation MSE (worst than of the model with weight decay), but the test MSE was better, 0.01785. In addition, the worst model architecture of 3.2.1 ($nh_1 = 5, nh_2 = 6$) also exhibited better test MSE (0.01750) than the model with decay. The obvious conclusion to be drawn, is that *increasing the model's capacity (either by using more nodes, or*

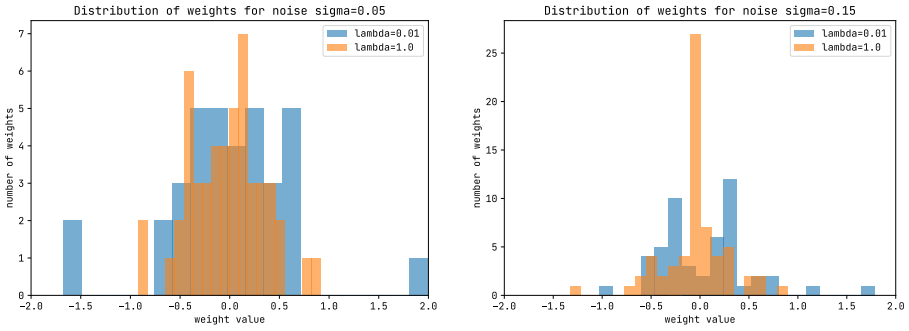

Figure 7: Weights distribution for $\sigma = 0.05$ (left) and $\sigma = 0.15$ (right) of the best models, and for varying weight decay λ.

by not using L2 regularizer) makes it perform better in the presence of noisy data. However, the test MSE is increased in the presence of noise, as expected.

4 Final remarks

It was a fun learning experience working with new tools for some of us, like PyTorch. Moreover, it was interesting to study the behaviour of the network with different types of data. We understood how important is to choose good hyperparameters and learned how to evaluate different models. Studying the impact of noise on the dataset made us realize how complicated it can be to work with real world data. More on final remarks is given in the presentation.

References

[1] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: *arXiv e-prints*, arXiv:1711.05101 (Nov. 2017), arXiv:1711.05101. arXiv: `1711.05101 [cs.LG]`.