Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 1

Perceptron rule and robust backpropagation in multi-layer perceptrons

Filip Husnjak, Ivan Klabucar and Corentin Royer

September 14, 2021

1 Main objectives and scope of the assignment

Our major goals in the assignment were

- to implement the perceptron and the MLP to perform classification, function approximation and time-series prediction tasks.
- to study the results and the limits of perceptron and MLP.
- to implement backpropagation and to make the learning process resistant to overfitting

We kept here a limited scope in the data we used. For classification and function estimation, we used 2 dimentionnal data.

2 Methods

We used python for all our experimentations. For the first part we added the numpy library for matrices computation, for the second part, we used pyTorch in addition to numpy. We used either VS Code or PyCharm as IDE. We used github for versioning and a simpler sharing of the tasks. We used a few different metrics (some of them computed by scikit learn) to assess classification performances, namely accuracy and confusion matrices. Accuracy is the fraction of correctly classified samples when the confusion matrix gives a deeper understanding of the classification performance for each class.

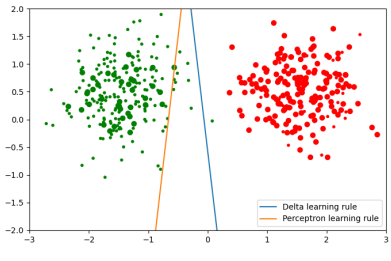
3 Results and discussion - Part I

3.1 Classification with a single-layer perceptron

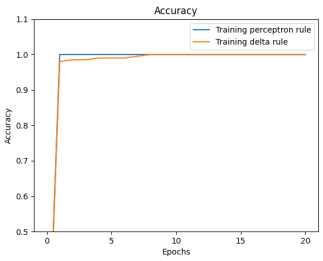
We implemented a single layer perceptron as a class with the option of using perceptron learning or the delta rule for training. The perceptron has 3 weights (one for each dimension and a bias).
To test the algorithm we create a cloud of points on the 2 dimensional plane. The points are part of 2 classes and are drawn from 2 normal distributions. We can choose to make the classes overlap to create non linearly separable data or have the classes clearly distinct.

3.1.1 Classification of linearly separable data

When we apply the perceptron with the perceptron learning rule, we can completely classify the data by learning an hyperplane between the 2 classes. Moreover, the algorithm converges in less than 5 steps most of the time (see 1b).
However the boundary than is computed is adjacent to one of the point, thus there is no margin between the boundary and the points. This is an issue because the data we used during training are not necessarily at the border of the distribution (see 1a).



(a) Learning with perceptron rule and with delta rule. The small dots are the sample not seen during training.



(b) Evolution of the accuracy for the delta rule and the perceptron learning rule ($\eta = 0.001$).

Figure 1: Comparison of the perceptron learning rule and the delta learning rule.

We then applied the delta learning rule, the algorithm solves the previous problem by finding the boundary with the biggest margin (see 1a). Here again, the algorithm converges in less than 10 steps.
Both of these algorithms converges irrespective of the initial value of there weights.

3.1.2 Classification of non linearly separable data

When we use non linearly separable data with the perceptron learning rule, it doesn't converges but switches back and forth between close states whereas the delta rule works the same as before and find the optimal hyperplane.
We made further testing by altering the cloud of points, we subsampled it with different degrees on each class (we only kept p1 percent of class A and p2 percent of class B). For each scenario, we measured the accuracy and the unbalance between the false positive and the false negative as computed with the formula in fig 2.

Percentage kept from each class	Accuracy	$\lvert\frac{FP-FN}{FP+FN}\rvert$
100%/ 100%	0.91%	0.29
50%/ 50%	0.93%	0.27
80%/ 20%	0.85%	0.87

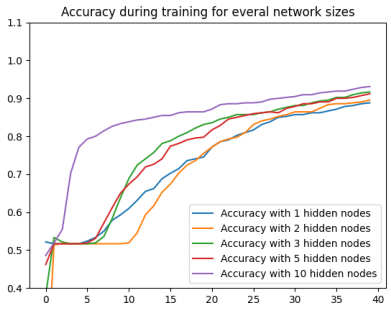
Figure 2: Performance at different level of subsampling

A bigger unbalance leads to a worse accuracy and generates unbalance between the rate of false positives and false negatives. This is due to the fact that the learning rule gives all points the same weight in the error computation, thus an unbalanced dataset pushes the boundary toward the smaller class, the boundary is not in the middle of the 2 distributions anymore.

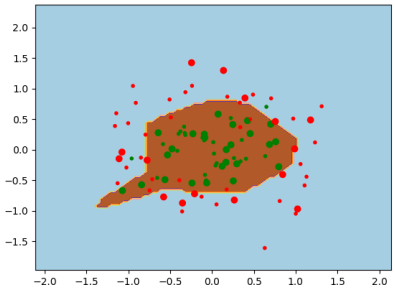
3.2 Classification and regression with a two-layer perceptron

3.2.1 Classification of linearly non-separable data

We used the multi layer perceptron (MLP) to classify the non linearly separable data. We can choose the number of hidden neurons in the MLP. We ran the network with different number of neurons and plotted the accuracy (see 3a).
We see that the number of neurons does not impact the performance of the network at the end of the training. This is probably due to the simple dataset (two gaussian distributions) that does not require a complicated boundary. Then we applied the MLP on another dataset, two concentric distribution of points. This time we increased the number of neurons to 1000 and we can see overfitting in the boundary representation (see 3b).



(a) Evolution of the accuracy depending on the number of epochs for different network sizes.

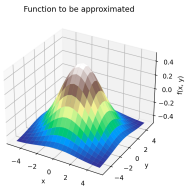


(b) Decision boundary of the MLP.

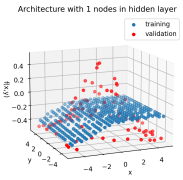
Figure 3: Training of the MLP for different number of neurons and representation of overfitting.

3.2.2 Function approximation

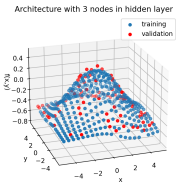
In this section we tested the multi-layer perceptron's ability to approximate an arbitrary continuous function shown in figure 4a.



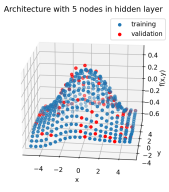
(a) Function to be learned



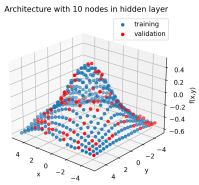
(b) 2x1x1 Architecture



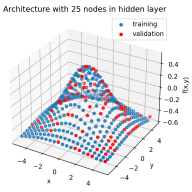
(c) 2x3x1 Architecture



(d) 2x5x1 Architecture



(e) 2x10x1 Architecture



(f) 2x25x1 Architecture

Figure 4

We tested multiple two-layer perceptron architectures (2xHx1) with varying number of nodes H in the hidden layer. The training was done on a subset containing 80% of samples, while the other 20% was used for validation. For each choice of hyper-parameters training was done multiple times with different weight initialization in order to determine average performance of every architecture. Some of the trained models' predictions can be seen on Figure 4.

We have trained n=100 models of each architecture on the same 80/20 data split in order to assess the average performance of these architectures. Measures of performance we have computed are the mean and standard deviation of the mean squared error on the validation set and can be seen on Figure 10.

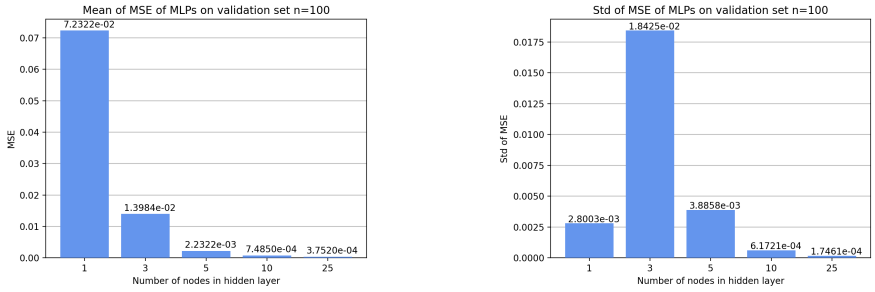


Figure 5

Average MSE falls as the complexity of the model and its ability to learn more complex shapes grows. We can see that the most complex network, the one with 25 hidden nodes, performed best displaying lowest average MSE and narrowest spread. The enormous variance of the architecture with 3 hidden nodes is somewhat peculiar, but can be explained by plotting predictions of some of these models. From Figure 6 we can deduce that the 2x3x1 architecture is just on the verge of being complex enough to model a bell shape and thus for some combinations of initial weights it struggles to learn the bell shape, while for other combinations it performs fine.

Figure 6

To examine the effect of the ratio of the split into training and validation sets, we trained additional 100 models of our 'best' architecture 2x25x1 on a 20/80 split. This is in contrast to our previous experiment when the training/validation split was 80/20 percent. We can see the comparison of these two architectures in Figure 8. Predictions of two models of this architecture trained on a 20/80 split can be seen in Figure 7.

Without sufficient data samples to be trained on, a relatively complex 2x25x1

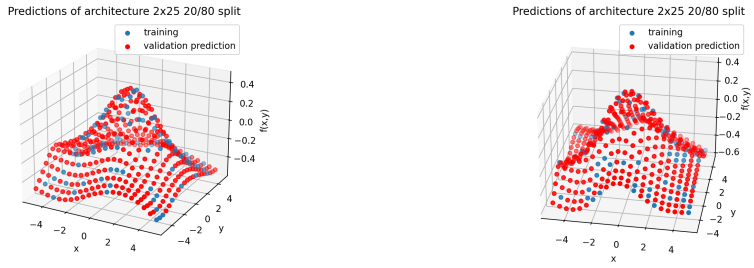


Figure 7

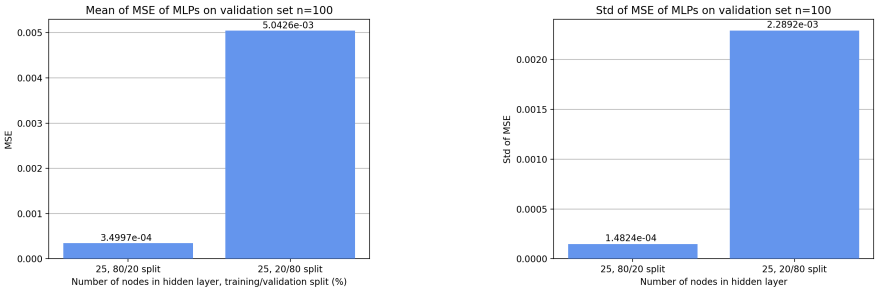


Figure 8

architecture settles into an overly complex shape that fits available data points well, but poorly predicts unseen data points. In other words it's overfitting. Even though this architecture performs well in circumstances where training data is abundant, if it has to 'make a guess' about an area of space it hasn't received any information on, it makes bad, overly complex predictions. Not only that, but we also get radically different models on the same training set from iteration to iteration as evident from Figure 8b in which we can see that the spread of the average MSE on the validation set is much larger for models trained on the 20/80 split. This shows just how much sensitive complex models are to the actual subsets randomly sampled for training. If the training data is biased, complex models will usually make bad, counter intuitive guesses for parts of the input space that weren't well represented in the training sample.

4 Results and discussion - Part II

For part 2 we implemented a class *NeuralNetwork* with parameterized number of hidden layers and nodes in each hidden layer. To introduce non-linearity into our system we used sigmoidal transfer functions inside hidden layers while in output layer we didn't use any activation function so it remains linear. For

training we used batch version of the backpropagation algorithm using a hold out (validation) set for early stopping. We performed multiple tests with different configurations for two hidden layers using a method called grid search.

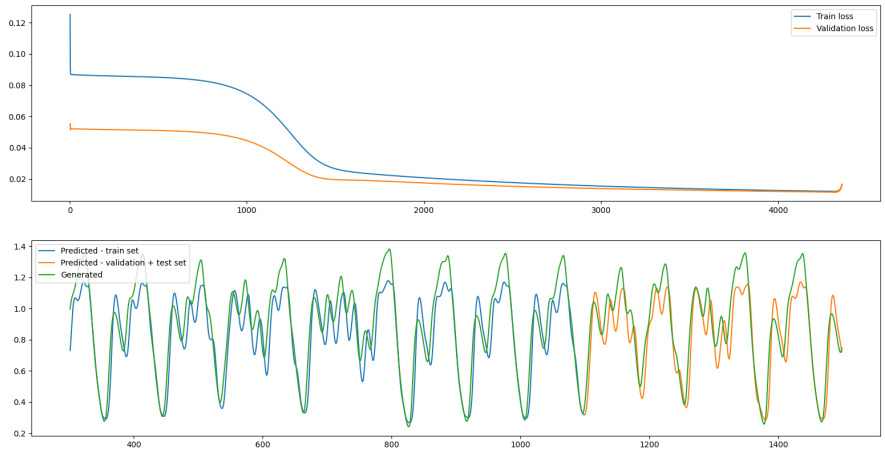
4.1 Three-layer perceptron for time series prediction - model selection, validation

First step was to train different configurations of neural network on data set without any noise and perform an evaluation of each configuration. Since we had a lot of data for training and the function behaves almost like a periodic function we expected good results on both train set and validation set as well as on a test set. The results are represented in table 9.

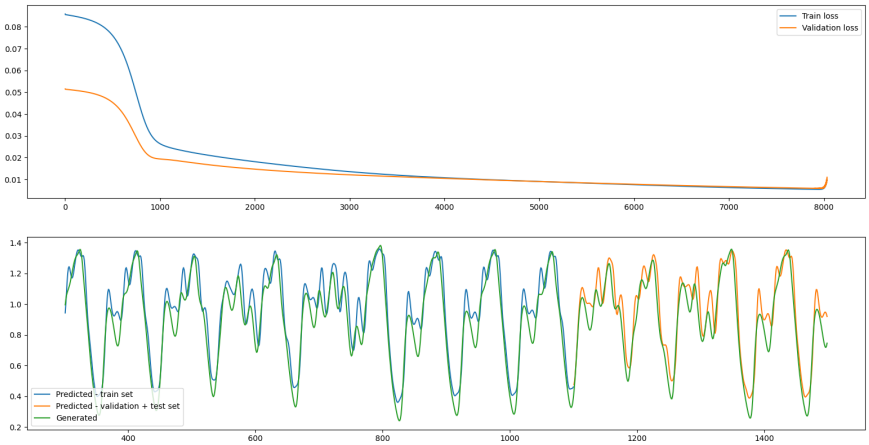
Hidden layer configuration	Average validation error after the training
2x2	0.00444
2x5	0.00918
2x15	0.01279
4x2	0.00447
4x5	0.00338
4x15	0.00459

Figure 9: Grid search with different number of nodes in hidden layers

The results clearly show that configurations 2x5 and 2x15 are the worst while configuration 4x5 is optimal. Predictions and errors for the best and worst configuration can be seen on the graphs below. First graph represents train and validation errors at certain number of epochs while second graph represents final result after the training. As shown in the graphs optimal configuration results in a better model with better generalisation performance.
Network with optimal configuration is much more robust with respect to random weight initialisation compared to the network with worst configuration. Each run with different random weights it manages to find good solution similar to other runs, while the worst configuration differs a lot for different initial weights, suggesting it has a much greater variance.



(a) Worst configuration: 2x15



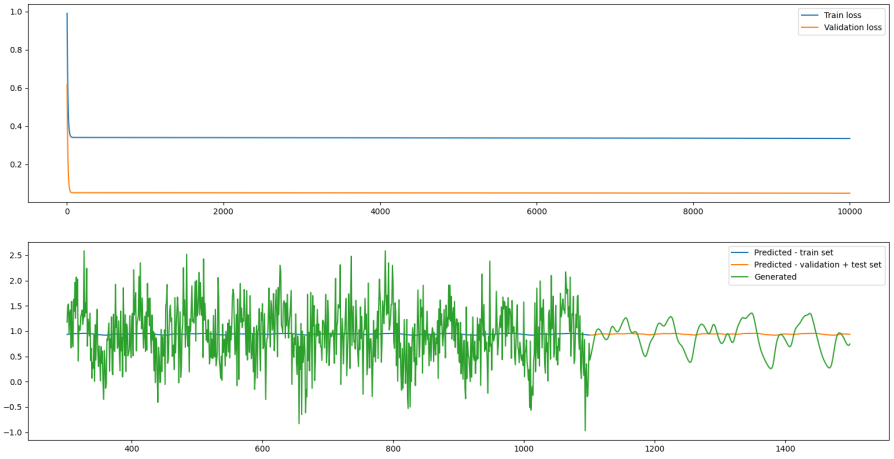
(b) Best configuration: 4x5

Figure 10: Performance comparison of the models with worst and best configuration

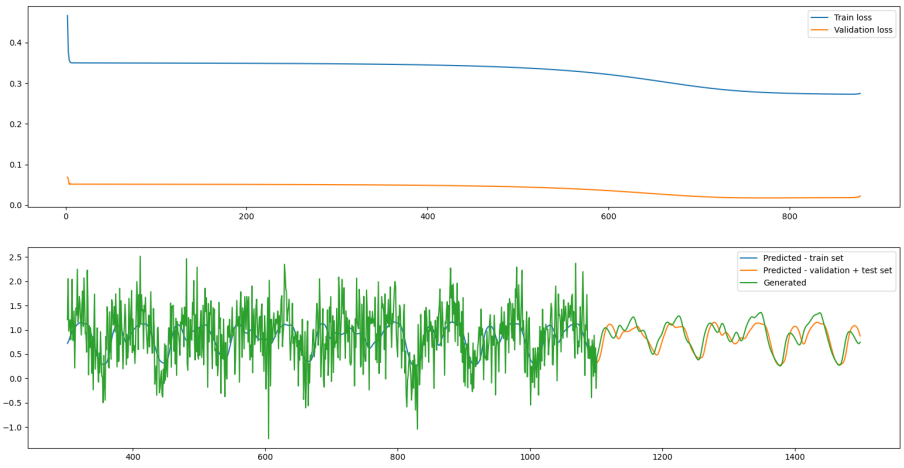
4.2 Three-layer perceptron for noisy time series prediction with penalty regularisation

By adding noise to the data we expect the increase of generalisation error and overall worse performance of the network. We introduced a new parameter λ which controls the importance of L^2 regularisation factor which helps in preventing the network from overfitting to the noise in the data and improves generalisation capabilities. The smaller the λ is the network will fit the training data better which means it will fit more noise, while larger lambdas prevent network from fitting too much noise, however if the parameter is too large it can

cause underfitting. On the graphs below this effect is illustrated. First graph represents network trained with large lambda which resulted in small variance but big bias. Second graph shows the prediction of the network that was trained with optimal value for regularisation parameter. It can be seen that network did not fit the noise so it generalises really well on unseen data.



(a) $\lambda = 0.1$

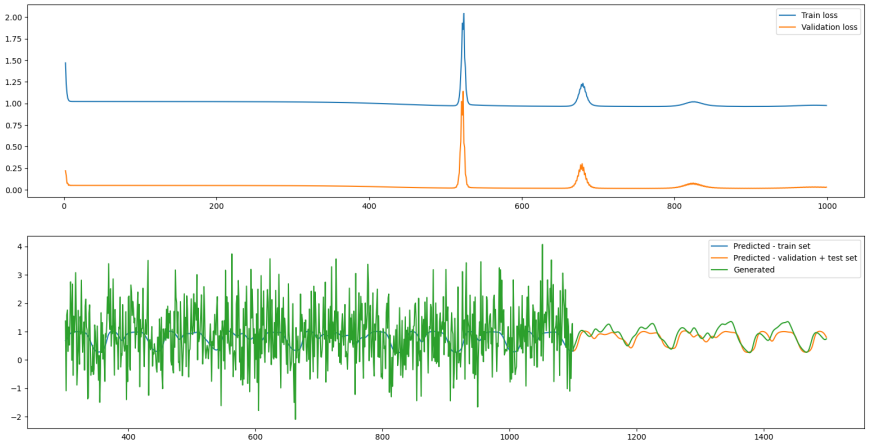


(b) $\lambda = 0.001$

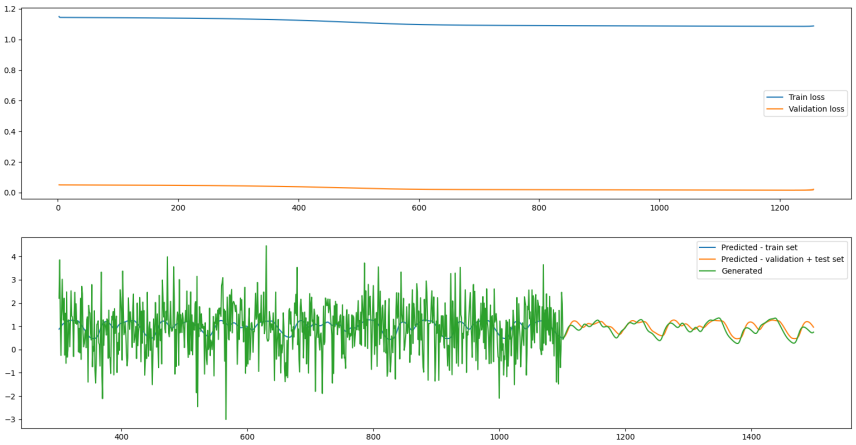
Figure 11: Performance comparison of the models trained with large and smaller regularisation parameters

We performed multiple tests with different amount of noise and concluded that the more noise there is the bigger the regularisation factor needs to be to counter overfitting. That can be seen in the figures below where we introduced more noise. First figure represents lambda that was optimal for smaller amount of noise and the second figure represents increased regularisation factor so it coun-

ters the additional noise that we introduced which results in better performance.



(a) $\lambda = 0.001$



(b) $\lambda = 0.0015$

Figure 12: Performance comparison of the models trained with different regularization parameters on data with more noise

5 Final remarks

It was helpful to get some hands on experience with many of the concepts we learned in class. For example, it was very informative to see how adding noise to input data was a double edged sword when it came to a complex model. It started to fit some of the noise and actually produced worse predictions. To counteract this we had to increase the regularization parameter lambda (weight decay) to compensate for the mentioned overfitting. Moreover, in 3.2.2. it's difficult to see without rotating the models in Figure 4, but it is noticable that the 2x3x1 architecture is actually made up of three 2x1x1 architectures rotated around the center point. We can see this by observing a slight triangle shape in the sides of the 2x3x1 architecture. We feel that observations like these gave us better intuition about the topic at hand.