Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 2

Radial basis functions, competitive learning and self-organisation

Tobias Bezner and Nik Joel Dorndorf

February 11, 2020

1 Main objectives and scope of the assignment

Our major goals in the assignment were

- to understand the structure and functionality of RBF-Networks by implementing the learning functions on our own
- to learn about and investigate the competitive learning approach in connection to self-organizing-maps
- to get to know the differences and similarities compared to MLP-networks

In this assignment we tried to implement all functions on our own, in order to understand the functionality of the ANN. Due to the scope of the tasks, we stepped through them and expanded our knowledge.

2 Methods

We implemented everything in Python. We used PyCharm as scripting environment and Github to share our code. In this assignment we implemented all functions concerning the ANN ourselves and did not use libraries. For this reason, we only used standard libraries like numpy and matplotlib.

3 Results and discussion - Part I: RBF networks and Competitive Learning

3.1 Function approximation with RBF networks

In the first part of the assignment we investigated RBF-simulations for the functions sin(x) and the square function. For the first tests, we used batch-learning calculated with the least square approach and data without noise. As you can see in Table 1 the absolute residual error for hte sin function is smaller than for the square function. With our setup it was not possible to reduce the error lower than 0.1, we calculated the RBF's with a width of 0.1. The error for the square function can be reduced to zero if we apply the sign function on the output and train the network with 4 nodes positioned on the maxima and minima.

For the next tests, we added Gaussian noise to the training and test datasets and implemented the delta rule for online-learning (no batch). As you can see in Figure 2, the batch learning provides slightly smaller errors than the online-learning with the delta rule. This can be explained by the definition of the least square error approach which is used for the batch learning and provides the optimal solution for the given data samples. The simulation were calculated for the delta-rule with a learning-rate of 0.01 and 100 epochs. In general, the error decreases with the number of nodes but converges to a minimum. With an even higher number of nodes the error starts to increase again and we hit the limit caused by the number of training samples. Overall the sin(x) is much easier to approximate than the square function. The jumps in the function cause big errors in the region around. For low numbers of nodes higher widths of the RBF's provide clearly better results. This makes sense because a small number of nodes with small width can not reach every point of the space. For very small widths, the batch learning becomes unstable what creates large residual squared errors, whereas the delta learning still reaches good results.

Hidden nodes	15	22	31	50
abs. res. error (sin)	0.2783	0.1	0.01	0.001
abs. res. error (square)	0.448	0.188	0.1	0.1

Table 1: Abs. res. error for different numbers of nodes without noise in data

Compared to the simulations without noise, the approximations with the noisy data were not able to reduce the residual error clearly under 0.01, while the approximation without noise in learning and test data reached errors below 0.001. For all the testing, we distributed the nodes equally over the investigated interval, compared to random positioning this worked much better. For random positioning, we had to increase the width to reduce the squared residual error. The lowest mean error over 15 runs we reached was 0.015 compared to 0.008 with the uniform distribution.

We also compared the single-layer-perceptron trained with backpropagation of

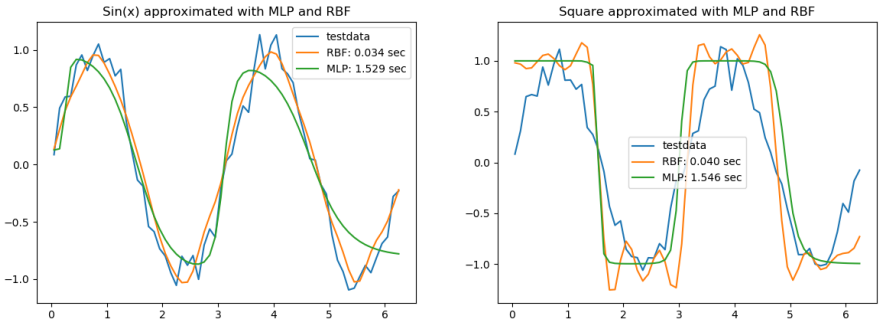

Figure 1: Comparison of RBF and MLP approximations for sin(x) (left) and square-function (right)

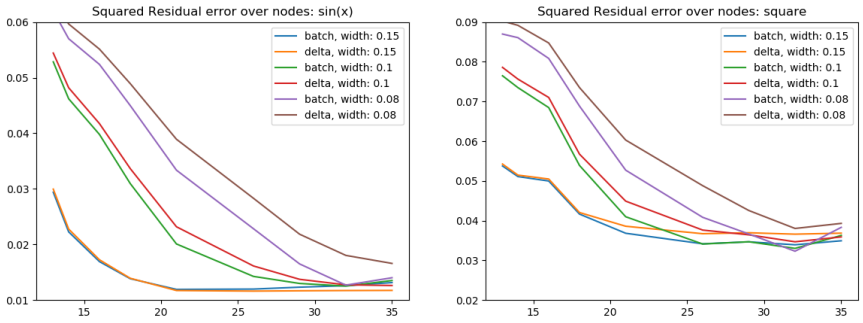

Figure 2: Squared Residual Error for different widths and number of nodes for sin(x) (left) and square-function (right)

the first assignment with the RBF-network. Both networks were trained in batch mode with the noisy data and 26 nodes in the hidden layer. In order to reach acceptable results for the MLP network we had to increase the number of epochs to 10000 (learning rate: 0.04; alpha: 0.8). Therefore, the calculation time increased and was much higher than for the RBF-network. The results of the simulations are shown in Figure 1. The squared residual error is in both cases smaller with the RBF-network, although the MLP-network delivers a very smooth output function.

3.2 Competitive learning for RBF unit initialisation

In the next part of the assignment, we added competitive learning for the initialisation of positions of the RBF's. Therefore, we set up a CL approach which choose one winner node and moves only this node and a second CL approach which also moves neighbors of the winner. These simulations were calculated with the same number (16) and width of the nodes (0.3), which generated good

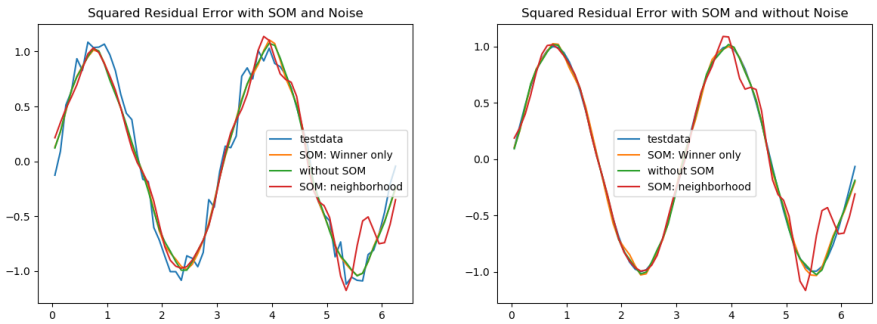

Figure 3: Comparison of Squared Residual Error for sin(x) with different RBF-initialisation. With noisy data (left) and without (right).

results in the previous tasks in 3.1. In Figure 3 the two CL simulations are compared with the previous uniform distribution of the nodes without noise and with added noise. As you can see, in this case the 'CL - winner only' and the approach without CL reach both good results. The 'CL - neighbor' was not able to reach these good results. We calculated the CL approaches with 10 Epochs and a learning rate of 0.2. For the neighbourhood, we chose 2 in each direction, so in the case of a total number of 16 nodes 5 are moved in each iteration, which seemed to be a reasonable number. Compared to the previous results, we did not reach an improvement for approximation of the sin-function, so the uniform distribution works already very good.

In the second part, we expand the approach to approximate two-dimensional-functions. For the given data examples, we were able to calculate reasonable results with a network of 20 nodes. We set the learning rate to 0.2 and only calculated one epoch to initialize the positions of the nodes. With these settings, we reached an absolute residual error of 0.013.

4 Results and discussion - Part II: Self-organising maps

4.1 Topological ordering of animal species

For the topological ordering of animal species we used 20 epochs for all of the experiments since early tests showed that more epochs were not making a huge difference. The starting number of neighbours was set at the beginning and then linearly decreased to zero neighbours over the epochs. The random initialisation of the means plays a big role for the grouping and over multiple runs, we got a lot of different results. What is interesting to notice, is how the number and the range of the groups changed depending on the starting number of neighbours. We call a 'group' a set of animals that has the same winning node at the end. In Figure 4 one can see the number of dif-

ferent groups, meaning how many nodes were at least one time the winning node in the end, and the range of the groups, describing the range from the winning node with the smallest index to the one with the highest. We can see that both values are increasing with the number of starting neighbours. An explanation could be that some nodes have a 'better' starting weight than others. Because they are winning a lot in the first epochs with a small number of neighbours only nodes withing a certain index range are being trained and the other ones are just left out. This changes when we train a bigger neighbourhood in the beginning.
An example of an ordering starting with 5 neighbours:

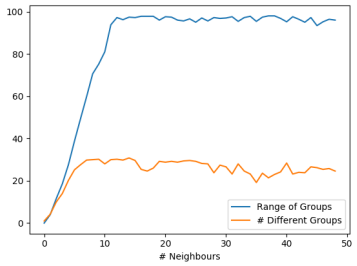

Figure 4: Number and Range of different groups

12 : 'elephant' 'rabbit' 'rat'
13 : 'bear' 'dog' 'hyena' 'walrus'
14 : 'ape' 'bat' 'cat' 'lion' 'skunk'
15 : 'antelop'
16 : 'camel' 'giraffe' 'horse' 'kangaroo' 'pig'
18 : 'crocodile' 'frog' 'seaturtle'
19 : 'duck' 'ostrich' 'pelican' 'penguin'
20 : 'beetle' 'dragonfly' 'grasshopper'
21 : 'butterfly' 'housefly' 'moskito'
22 : 'spider'

4.2 Cyclic tour

Again, we keep the number of epochs to 20 for this task and only play around with the starting size of the neighbourhood. With at least a starting neighbourhood size of 1, the SOM outputs reasonable results sometimes. This stabilizes with from a starting neighbourhood size of 2. For this starting size and higher ones, the model outputs nearly the same order every time. The only thing changing are two pairs of cities that are very close to each other. In most of the cases these the points of a pair have the same winning node in the end what leads to an ambiguous order. Two of the possible routes can be seen in Figure 5. This result arises probably from the fact that the difference between the distance between the close nodes and the distance to the other nodes is large.

4.3 Clustering with SOM

For the two-dimensional clustering we trained our network with a two-dimensional neighbourhood. The chosen number of epoch was again 20 and we started with a neighbourhood size of three linearly decreasing to zero. The results for the different attributes can be seen in Figure 6. What one can directly notice is that especially the party membership has an huge impact on the voting behaviour. The traditional

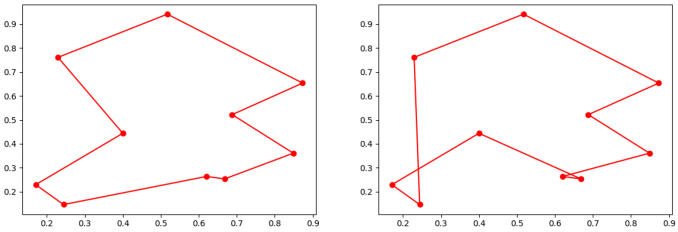

Figure 5: Two possible routes for the order by the SOM

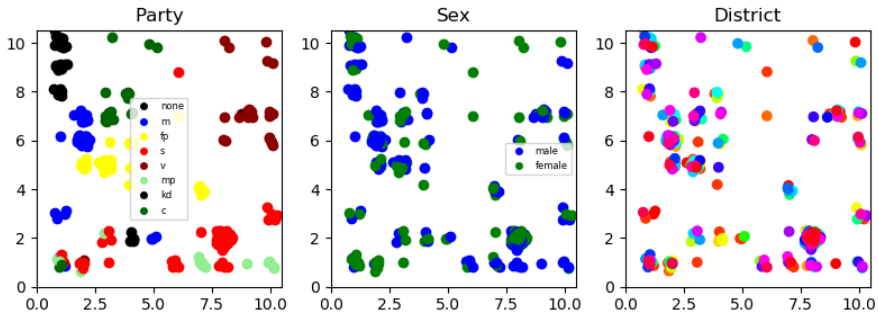

Figure 6: SOM Mapping of Votes (points are randomly moved by a small factor to see clouds)

left-right scale is depicted nicely with left and right being switched in this mapping. Interesting is also, that the members without a party tend to vote similarly. Looking at the other two attributes, it seems like there is no real pattern. Taking a closer look, you can see in the bottom left corner some female members. Looking at the parties, one can see that they do belong to a lot of different parties, so maybe the sex has an influence there. The district a member is coming from seems to have nearly no influence but there is again a group of members (in red) in the lower left part of the plot which are all belonging to the same district. They seem to belong all to the same party but are quite far away from the other members of their party in the 'party'-plot. To put it in a nutshell, the party membership seems to have by far the highest influence on the voting behaviour which does make a lot of sense.

5 Final remarks

Overall, a lot of the results were probably as you would expect them to be. The function approximation was working pretty nicely and very fast. The fact that letting the nodes take their initial value with competitive learning did not improve the performance stems probably from the fact that the function is very easy because it is periodic and symmetric. The SOM Mapping reminded us a lot of k-means clustering which would probably lead to very similar results.