

Carl Ridnert (940325-0112)

Óttar Guðmundsson (910302-0450)

**Lab assignment 1 -Radial basis functions, competitive learning and self-organisation**

**Part 1**

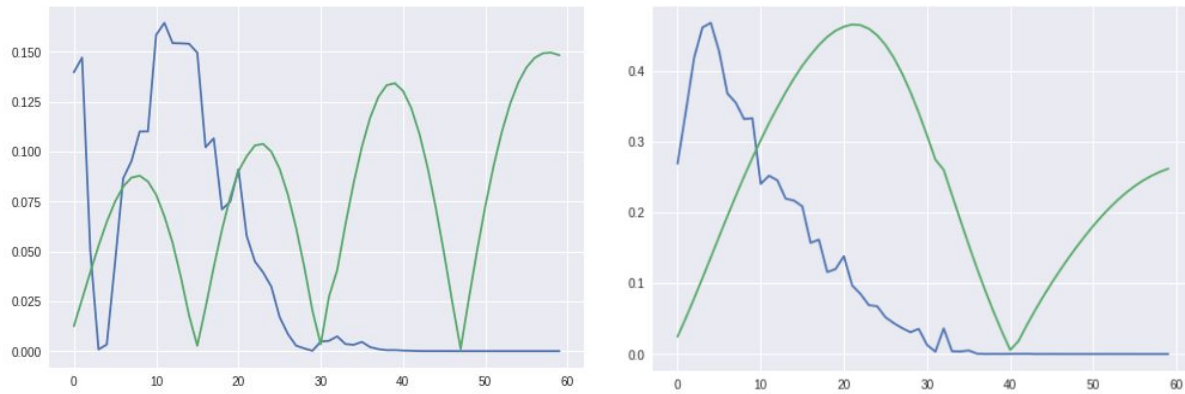
**3.1**

By trying the batch method with sigma 0.9 on the sin data we managed to get the absolute residual error under 0.1, 0.01 and 0.001 with 3 or 4 nodes using the batch method. Adding more nodes from this point increased the error until about 10 nodes after which the error started to decrease. Using the incremental methods with same sigma and eta 0.01 required around 15 nodes before rising again, but reaching the same minimal error by adding another 15 nodes and so on. The green line depicts this behaviour and each error estimate is after the sequential training over all data points/epochs.

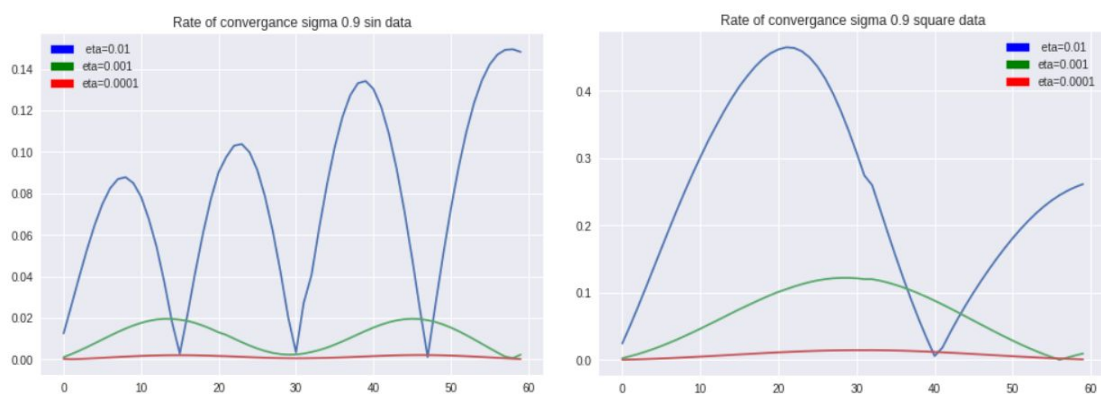
By testing the same methods on the square data, we managed to get the error lower than 0.001 using the batch method by using a lot more nodes, around 37 but not with the incremental function. .

N RBF	Sin data - batch	Sin data - incremental	N RBF	Square data - batch	Square data - incremental
1	0.1397032606	0.0125112267	28	0.0366788563	0.3961163218
2	0.1470566023	0.0259787621	29	0.0308133707	0.3700429583
3	0.0503036838	0.0395858699	30	0.0355967544	0.3406201426
4	<u>0.0008467219</u>	0.0527286321	31	0.0128750743	0.3085046007
5	0.0033053366	0.0646999936	32	0.0033441220	0.2744575525
6	0.0440876680	0.0748361428	33	0.0362157012	0.2600029684
7	0.0865833980	0.0824480561	34	0.0039134495	0.2245646161
8	0.0951614080	0.0869271533	35	0.0035216619	0.1891828571
9	0.1100223871	0.0878463688	36	0.0050452610	0.1545338803
10	0.1100808988	0.0849871365	37	<u>0.0006131422</u>	0.1211579811
11	0.0783861459	0.0783861459	38	0.0003732431	0.0894383664
12	0.1584128218	0.0676052275	39	0.0001990864	0.0595992953
13	0.1644671430	0.0542543389	40	0.0006101872	0.0317208271
14	0.1543326488	0.0371660702	41	0.0002222119	<u>0.0057653019</u>
15	0.1542138354	0.0178797027	42	0.0003066766	0.0183900166
	0.1539843932	<u>0.0027168763</u>			0.0409191921
		0.0217148889			

Carl Ridnert (940325-0112)  
 Óttar Guðmundsson (910302-0450)



The rate of convergence spikes when the learning rate is high and adds to the error, giving rise to oscillations. It seems like on certain intervals it reaches optimum solution but rises again, depending on the value of the learning rate. So by changing the rate, the spikes are higher and the interval of change is more frequent. Note that the error is always positive in contrast to the residual because we took the root mean of the residual error. The same characteristics would hold for the residual error plots with the only difference that every other peak would be inverted.



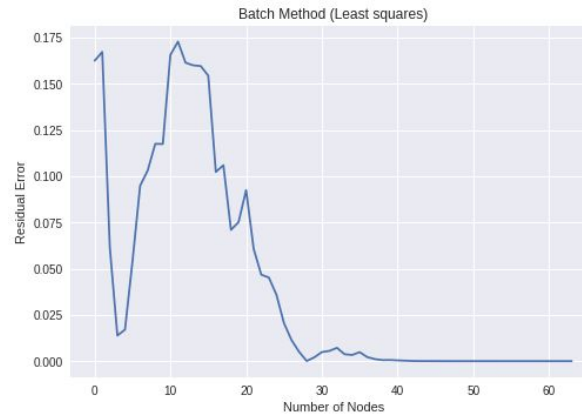
A modification to the output can be made so that the output will be rounded to -1 or +1, then it can be shown that the residual error can be zero for the incremental version. It is not quite zero for the least squares method, this can be seen in the following plots. This method could perhaps be useful in data transfers over noisy signals since then one is concerned with the prediction of binary signals.



Carl Ridnert (940325-0112)  
Óttar Guðmundsson (910302-0450)

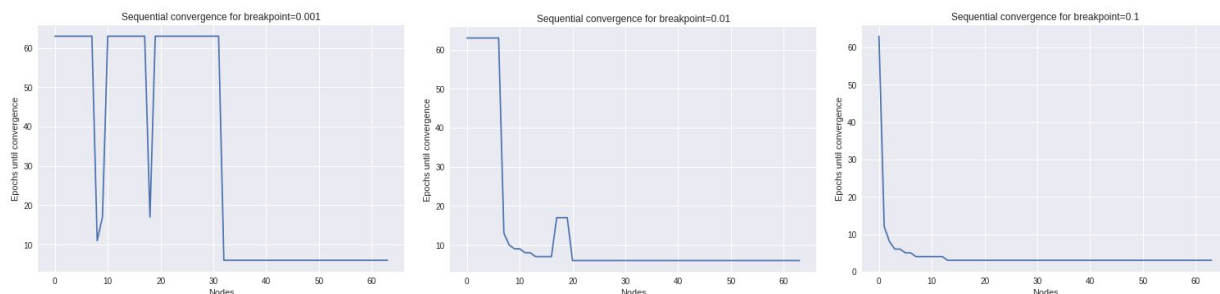
### 3.2

With gaussian noise added to the sinusoidal function, we look at the batch learning method and the incremental method. The number of nodes are varied and for the batch version we obtain the following graph depicting the residual error.



One can notice that the threshold of 0.1 is achieved early for 2 nodes and then later at 16 nodes. The threshold of 0.01 is achieved with 4 nodes and also after about 26 nodes.

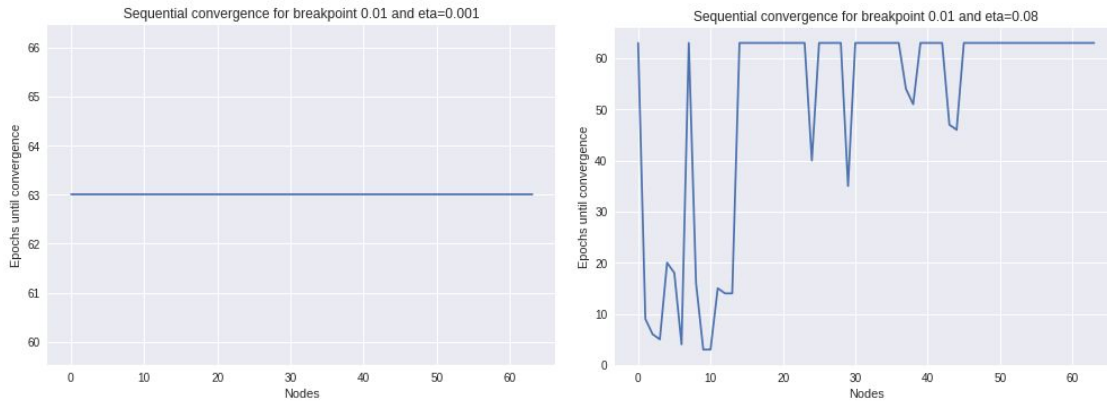
Next the same kind of analysis is done for the sequential case, here we run the code in epochs where one epoch is the weight update after one new datapoint is included. The following graphs depict the number of epochs needed to obtain the desired error thresholds as a function of the number of nodes. Since the length of data was 64, this is the maximal number of epochs and thus flat parts of the graph at 64 indicate that the error threshold was not achieved.



From these plots it is evident that a threshold of 0.1 is achieved for all number of nodes in 0 steps. The threshold of 0.01 is achieved in the lowest number of epochs after 17 nodes after which it remains constant. Finally, the threshold of 0.001 is achieved fastest with 19-24 nodes.

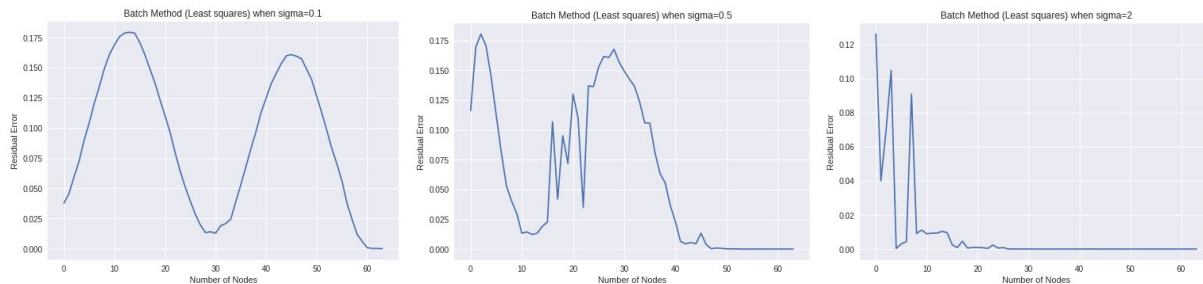
How the learning rate affects the convergence is illustrated in the following two graphs

Carl Ridnert (940325-0112)  
Óttar Guðmundsson (910302-0450)



First it is too slow resulting in non-convergence, secondly it is too fast resulting in an unstable behaviour. The eta value used in the previous three plots seems to be the better choice.

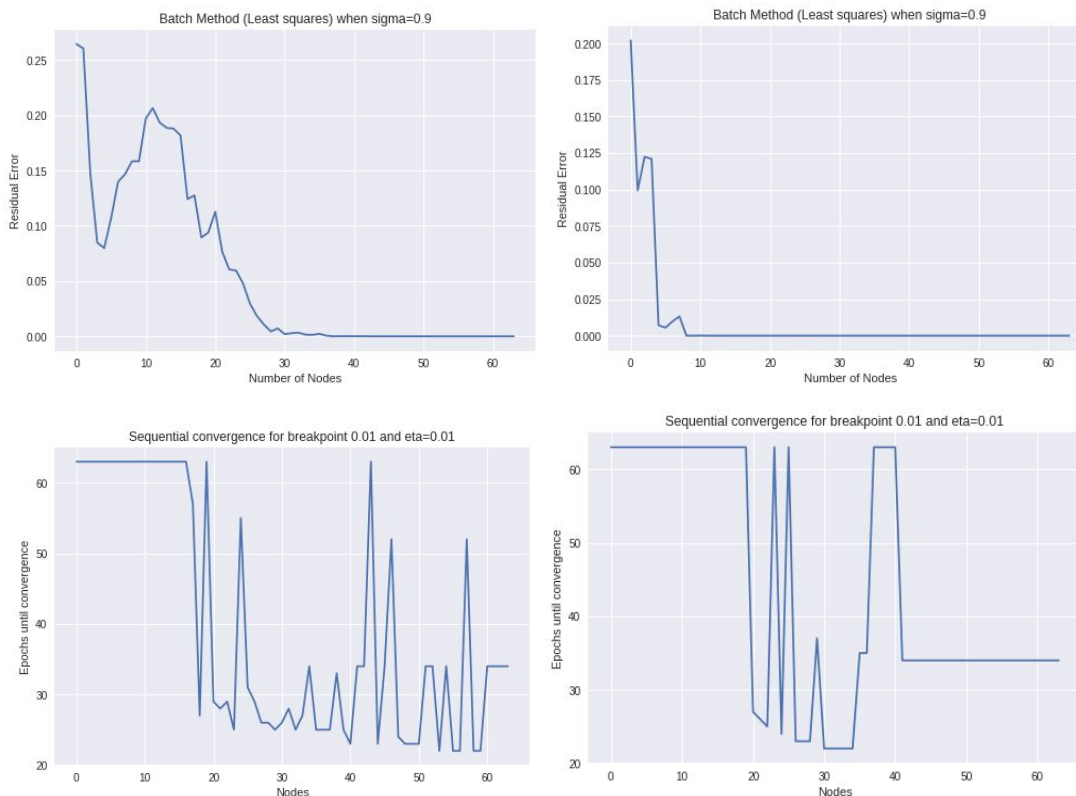
Changing sigma value or the width of the RBF's affects the plots in the following way



Having a large sigma seems to provide a better result in terms of error for a fewer number of nodes. This might be because the data is evenly spaced without clusters so that each node covering or responding to more data points is beneficial.

Carl Ridnert (940325-0112)  
 Óttar Guðmundsson (910302-0450)

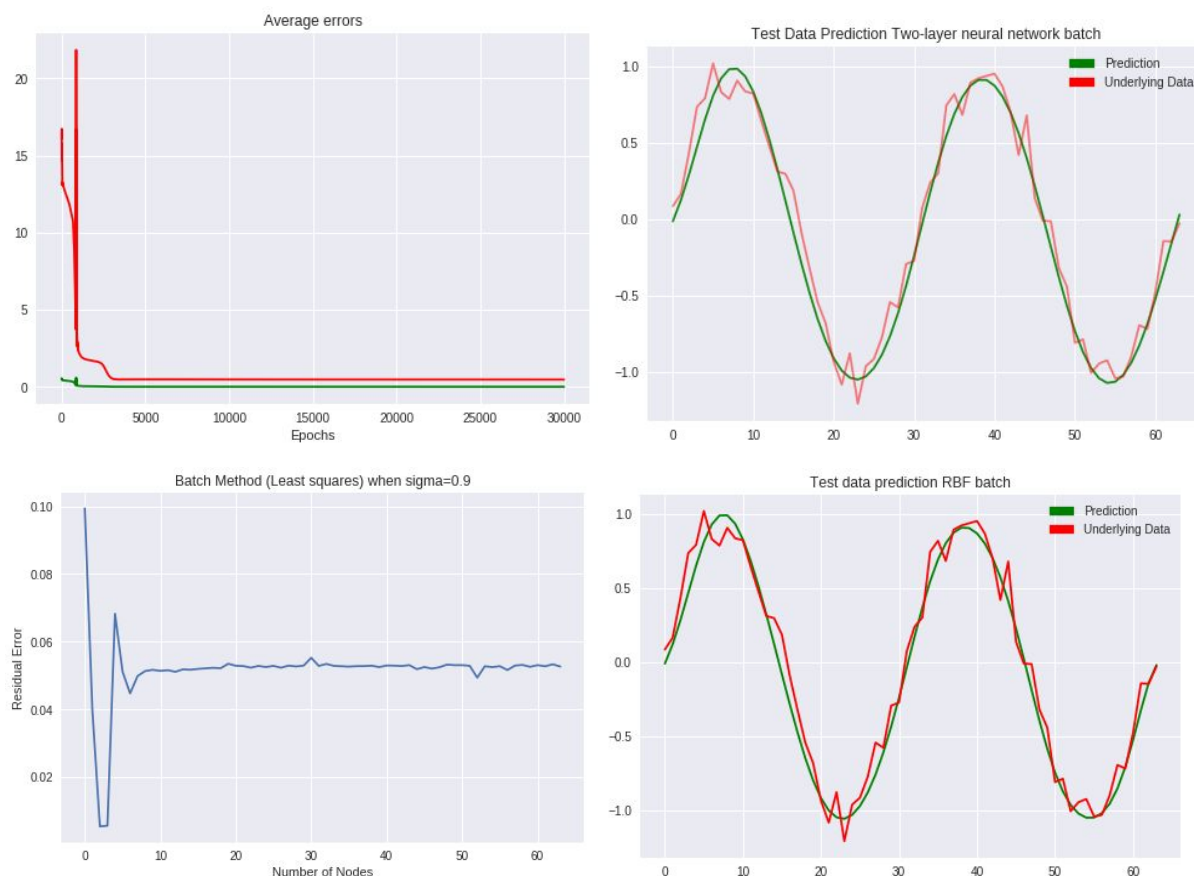
The following plots illustrate how the choice of placement for the RBF nodes affect the error for batch and sequential learning respectively. Our method of choice was to place the nodes equally spaced on the axis from 0 to  $2\pi$ . To the left hand side we have the plots when nodes are placed according to this strategy and to the right when they are placed randomly.



One can notice that for the sequential learning this change seems to matter less than for batch learning where actually the random positioning of nodes on the x-axis seems to lead to better performance.

Carl Ridnert (940325-0112)  
Óttar Guðmundsson (910302-0450)

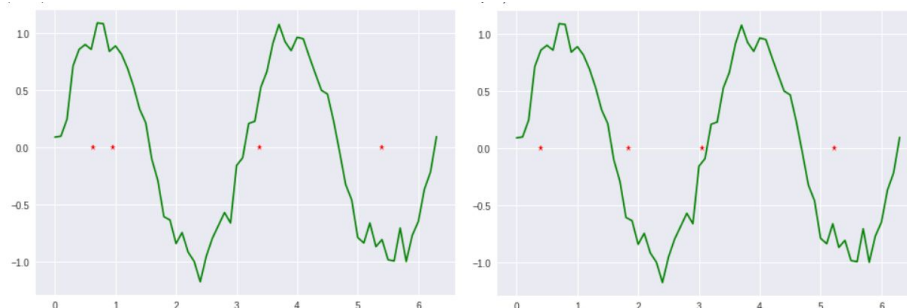
Finally we make a comparison with a two layer perceptron.



We see that the RBF does a better job at fitting the function since we obtain a mse of below 0.01 for 4 nodes whilst the neural network managed a 3-run average of 0.12 which is higher. Also the neural network took a lot longer to train so one would prefer the RBF network in this simple case.

### 3.3

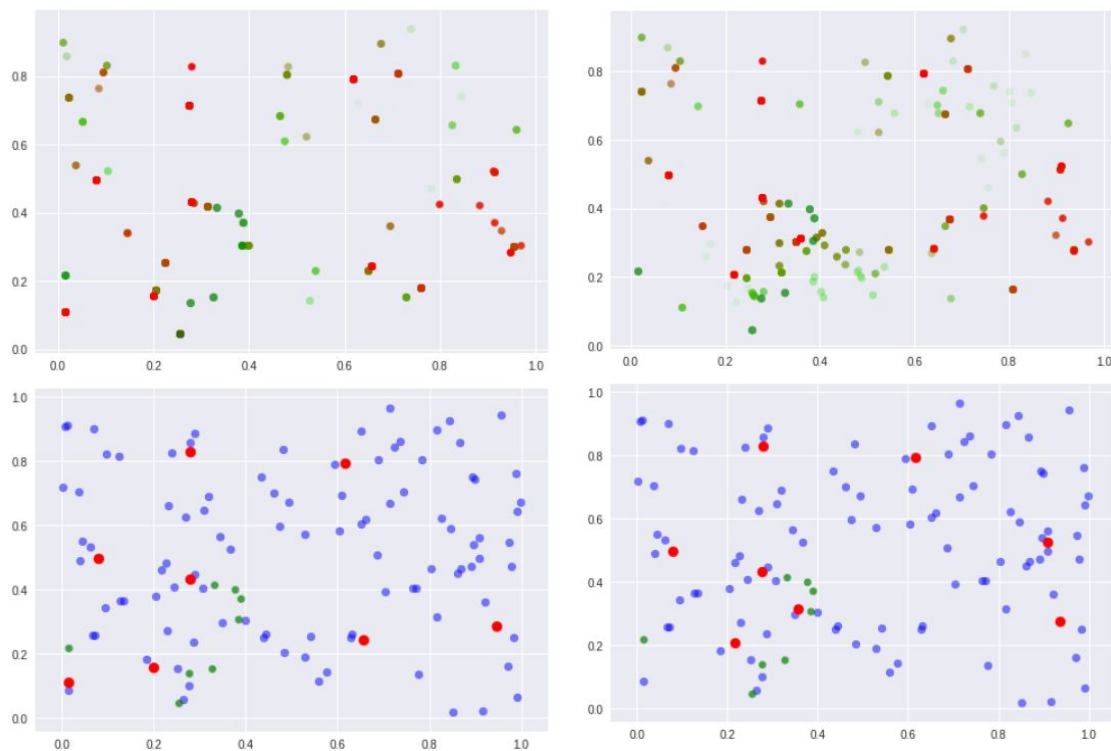
For the competitive learning on the sin data we managed to get the error below 0.01 by setting the variables the same as in 3.1 (4 nodes and 0.9 sigma). It didn't matter if we used a neighbourhood updating method, the errors were nearly the same. The average error was a little bit higher in the non noisy data, it was 0.004 with CL and 0.003 with neighbour updating. For the noisy data we got around 0.003 and 0.002.



Carl Ridnert (940325-0112)  
 Óttar Guðmundsson (910302-0450)

For the ballist.dat data we adjusted the cluster by picking about half the points, or 55 to be exact. We picked 8 nodes for the data since it was quite spread out. We used different learning rates for updating the winning node and its neighbours. We implemented a custom method for updating the neighbours, but all clusters that were within winning range (the distance from the winning cluster to the target vector) of the winning node were drawn closer to it by a percentage of their distance to the winning node. The range interval was then lowered in every epoch.

By looking at the graphs here below, we can see that only when updating the winning node (to the left) we get at least 1 dead node in the far left corner. When updating the neighbours (right plots) is being used, they seem to cover more area. An interesting graph is the heatmap in the top layer, but it shows the movement of the nodes in every epoch where the nodes start of from the green points and progress towards the red ones. We note that there seems to be more movement when using neighbouring updates which is in line with theory, we have less dead units here.



Without using the CL method and picking random positions our average error was 0.002076. Using the CL method we reduced the error by nearly 80% or 0.000422. Finally by using our updating neighbour method we got the error down by 50% 0.000235. These results varied between random starting points and weights but the results were mostly reduced by this much.

## Part 2

### 4.1

Our final result is shown in the table below, it shows the sorted positions of the 32 animal species where each are associated to one output node, so the list was sorted after the output nodes. The first column is the results when only the winning node was updated, the second one is when updating neighbours in a range of "range=0.6" around the winning node by nEta, range and nEta was decreased each epoch to the final values displayed below. This was the optimal clustering in our opinion. The right column shows that the clustering becomes bad when the learning rate and range is too high.

Old method Range 0.6 eta = 0.2 nEta = 0.15	New method eta = 0.2 final_eta = 0.03 nEta = 0.1 final_nEta = 0.01 range = .6 final_range = 0.05	New method eta = 0.3 final_eta = 0.05 nEta = 0.05 final_nEta = 0.03 range = .7 final_range = 0.1
[[ '13.0', ["beetle"] ]] [[ '13.0', ["butterfly"] ]] [[ '13.0', ["dragonfly"] ]] [[ '13.0', ["grasshopper"] ]] [[ '13.0', ["housefly"] ]] [[ '13.0', ["moskito"] ]] [[ '13.0', ["spider"] ]] [[ '22.0', ["ape"] ]] [[ '22.0', ["bear"] ]] [[ '25.0', ["skunk"] ]] [[ '29.0', ["walrus"] ]] [[ '35.0', ["kangaroo"] ]] [[ '42.0', ["cat"] ]] [[ '42.0', ["lion"] ]] [[ '44.0', ["dog"] ]] [[ '47.0', ["hyena"] ]] [[ '5.0', ["antelop"] ]] [[ '5.0', ["camel"] ]] [[ '5.0', ["giraffe"] ]] [[ '5.0', ["horse"] ]] [[ '5.0', ["pig"] ]] [[ '63.0', ["rabbit"] ]] [[ '63.0', ["rat"] ]] [[ '71.0', ["duck"] ]] [[ '71.0', ["ostrich"] ]] [[ '71.0', ["pelican"] ]] [[ '71.0', ["penguin"] ]] [[ '77.0', ["bat"] ]] [[ '77.0', ["elephant"] ]] [[ '81.0', ["crocodile"] ]] [[ '81.0', ["frog"] ]] [[ '81.0', ["seaturtle"] ]] 	[[ '10.0', ["ape"] ]] [[ '10.0', ["cat"] ]] [[ '10.0', ["lion"] ]] [[ '10.0', ["skunk"] ]] [[ '15.0', ["crocodile"] ]] [[ '15.0', ["frog"] ]] [[ '15.0', ["seaturtle"] ]] [[ '32.0', ["bat"] ]] [[ '32.0', ["elephant"] ]] [[ '32.0', ["kangaroo"] ]] [[ '32.0', ["rabbit"] ]] [[ '32.0', ["rat"] ]] [[ '41.0', ["antelop"] ]] [[ '41.0', ["camel"] ]] [[ '41.0', ["giraffe"] ]] [[ '41.0', ["horse"] ]] [[ '41.0', ["pig"] ]] [[ '66.0', ["bear"] ]] [[ '66.0', ["dog"] ]] [[ '66.0', ["hyena"] ]] [[ '66.0', ["walrus"] ]] [[ '72.0', ["beetle"] ]] [[ '72.0', ["butterfly"] ]] [[ '72.0', ["dragonfly"] ]] [[ '72.0', ["grasshopper"] ]] [[ '72.0', ["housefly"] ]] [[ '72.0', ["moskito"] ]] [[ '72.0', ["spider"] ]] [[ '80.0', ["duck"] ]] [[ '80.0', ["ostrich"] ]] [[ '80.0', ["pelican"] ]] [[ '80.0', ["penguin"] ]] 	[[ '62.0', ["crocodile"] ]] [[ '62.0', ["duck"] ]] [[ '62.0', ["frog"] ]] [[ '62.0', ["ostrich"] ]] [[ '62.0', ["pelican"] ]] [[ '62.0', ["penguin"] ]] [[ '62.0', ["seaturtle"] ]] [[ '64.0', ["antelop"] ]] [[ '64.0', ["ape"] ]] [[ '64.0', ["bat"] ]] [[ '64.0', ["bear"] ]] [[ '64.0', ["camel"] ]] [[ '64.0', ["cat"] ]] [[ '64.0', ["dog"] ]] [[ '64.0', ["elephant"] ]] [[ '64.0', ["giraffe"] ]] [[ '64.0', ["horse"] ]] [[ '64.0', ["hyena"] ]] [[ '64.0', ["kangaroo"] ]] [[ '64.0', ["lion"] ]] [[ '64.0', ["pig"] ]] [[ '64.0', ["rabbit"] ]] [[ '64.0', ["rat"] ]] [[ '64.0', ["skunk"] ]] [[ '64.0', ["walrus"] ]] [[ '91.0', ["beetle"] ]] [[ '91.0', ["butterfly"] ]] [[ '91.0', ["dragonfly"] ]] [[ '91.0', ["grasshopper"] ]] [[ '91.0', ["housefly"] ]] [[ '91.0', ["moskito"] ]] [[ '91.0', ["spider"] ]] 

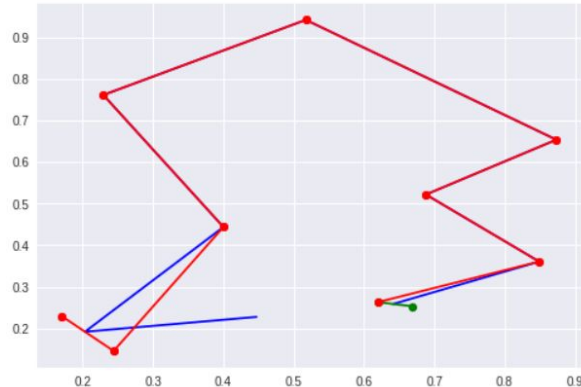
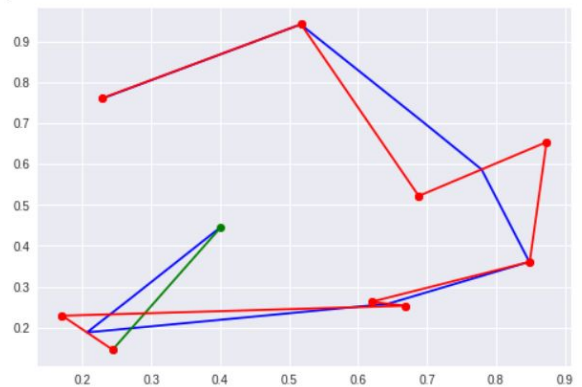


Carl Ridnert (940325-0112)

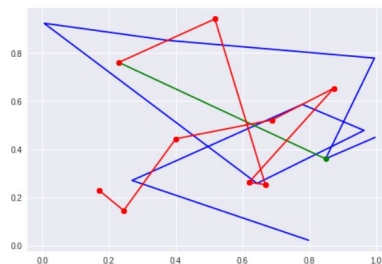
Óttar Guðmundsson (910302-0450)

#### 4.2 Cyclic

Here we used the same architecture as before, the only difference being changing the neighbouring method to cyclic instead of using range. The algorithm started with the winning node and its 2 closest neighbours in both directions. The number of neighbours was set to 1 at epoch 126 and then to 0 at epoch 376, but these epochs are selected as a percentage of finished epochs. We trained for 500 epochs, with step 0.2 and neighbour step 0.1 and got these results

Non adjusting learning rates	Adjusting learning rates
<pre>[0.  0.6683 0.2536] [0.  0.6195 0.2634] [1.  0.8488 0.3609] [2.  0.6878 0.5219] [3.  0.8732 0.6536] [4.  0.5171 0.9414] [5.  0.2293 0.7610] [7.  0.4000 0.4439] [8.  0.2439 0.1463] [8.  0.1707 0.2293] Distance: 2.43768081</pre>	<pre>[0.  0.6683 0.2536] [0.  0.6195 0.2634] [1.  0.8488 0.3609] [2.  0.8732 0.6536] [2.  0.6878 0.5219] [4.  0.5171 0.9414] [5.  0.2293 0.7610] [6.  0.4000 0.4439] [8.  0.2439 0.1463] [8.  0.1707 0.2293] Distance 2.14692352</pre>
	

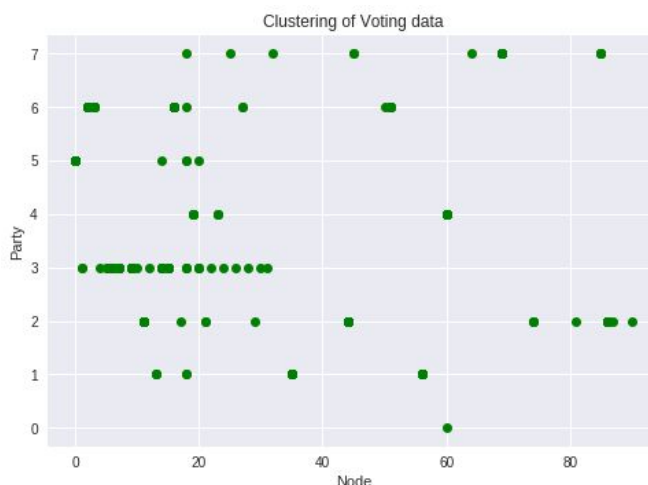
By lowering the neighbours we seemed to get a decent fit with most cities having their own clusters. If we used the same degrading learning rate as in 4.1 the cities seemed to share clusters more frequently. Without using the neighbouring method, we were quite off with distance 4.54348196.



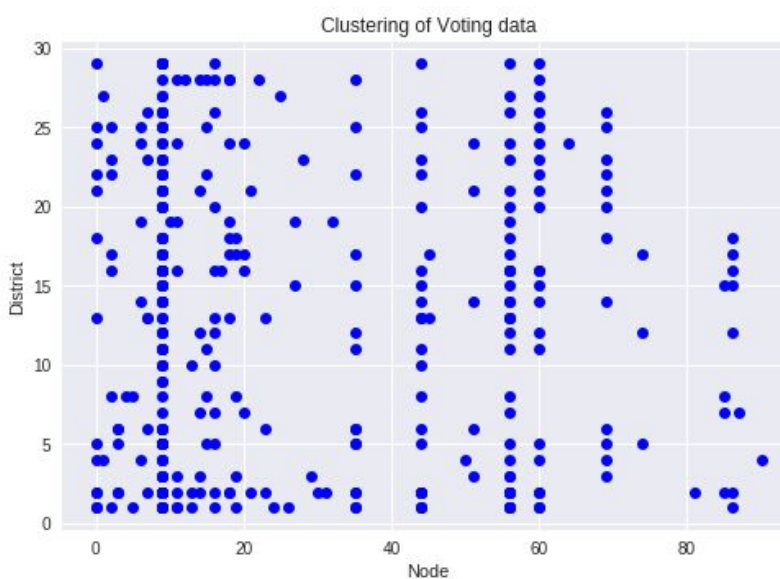
Carl Ridnert (940325-0112)  
Óttar Guðmundsson (910302-0450)

### 4.3 Votes

Using the same architecture as the same assignments but with 349 inputs and 100 output nodes we obtained following clustering of data which is illustrated below for the different factors District, Gender and Party where in party 0=no party, 1='m', 2='fp', 3='s', 4='v', 5='mp', 6='kd', 7='c'.



From the district data it is hard to spot any clear patterns. However, we can see that for districts 20 up to 30 there is no representation in nodes higher than 70 which distinguish these districts from the rest. In the party data we see that for S there is a quite strong consensus among the voters activating nodes to the left, this is the case for mp as well. The other parties are more spread out and the low number of data makes conclusions prone to error.



Carl Ridnert (940325-0112)  
Óttar Guðmundsson (910302-0450)

In the Gender data we observe a large spread over the nodes for both genders, however the majority seems to be located at nodes to the left which could be a result from the fact that "S" is by far the biggest party.

