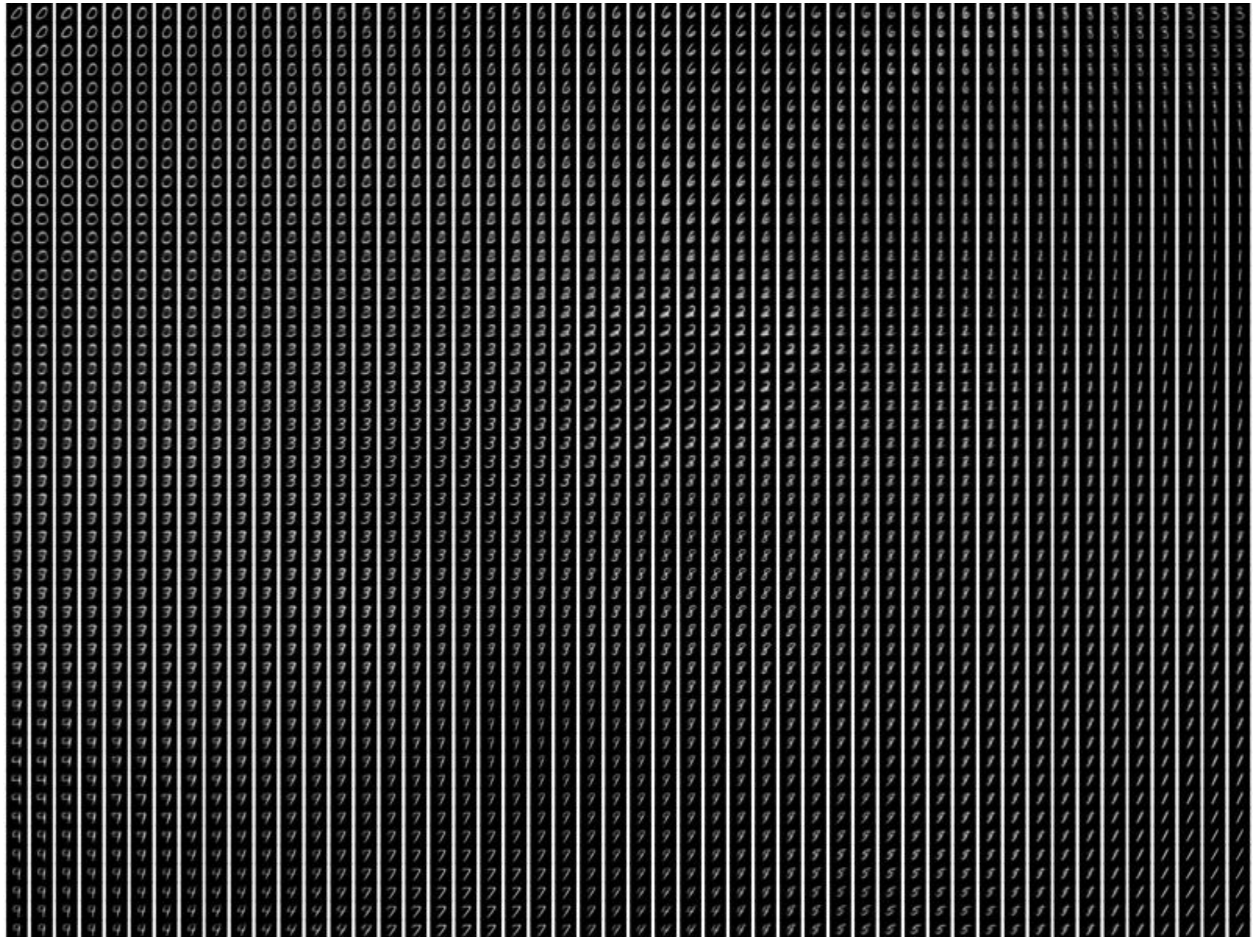


Assignment - Learning in Artificial Intelligence



Report created by:

- Nikolaj Schaldemose Reibke
- Almir Mehanovic

nirei12@student.sdu.dk
almeh12@student.sdu.dk

Date Created: 25 - 5 - 2016

Time Created: 1464211220130 milliseconds since 1st of January 1970

The Report was split into the two first assignments of the entire assignment.

The first Section "Supervised learning: Regression with neural networks" was 100% written by Almir Mehanovic.

The second Section "Unsupervised learning: Self-organising map" was written 100% by Nikolaj Reibke

Supervised learning: Regression with neural networks

Introduction

Neural networks (ANNs) can be made to fit any function given the right topology and the right parameters for learning. In this assignment two different functions will be approximated using ANNs of different topologies and varying parameters for learning. The two functions are:

$$y(x) = \sin(2\pi x) + \sin(5\pi x), \text{ where } x = -1:0.002:1$$

$$z(x,y) = \exp(-(x^2 + y^2) / 0.1), \text{ where } x = -1:0.05:1, \text{ and } y = -1:0.05:1$$

Following parameters will be investigated as to how they affect training time, test time, and MSE of the neural network:

- Number of hidden layers
- Number of neurons in each layer
- Epochs performed during training
- Learning rate and learning rate decay

The next section will briefly describe the implementation used. Next the results of various experiments will be presented and discussed.

Implementation

The task was solved using the *R* programming language and the *deepnet*¹ library which provides a ANN implementation. The *deepnet* library provides functions to train a fully connected ANN with any number of layers and neurons in each layer.

The algorithm used to train the network is standard backpropagation in conjunction with SGD as the optimization function. The function is implemented such that a batch size can be specified for computing the gradient. Instead of computing the true gradient ("offline mode") or the gradient at a single example ("online mode"), the gradient is computed against a number of training examples specified by the batch size. The batch size used in all of the experiments is 100.

Experiments

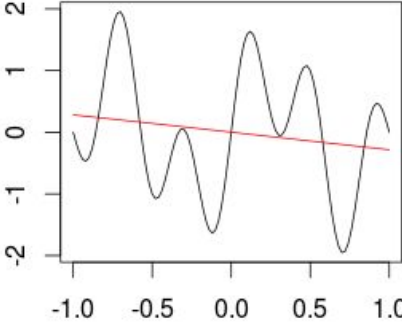
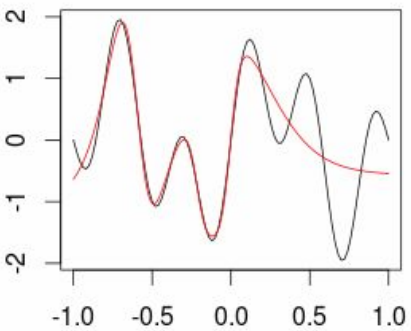
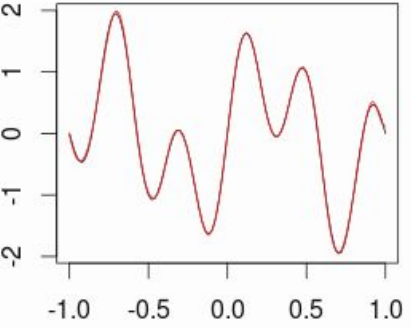
Experiment 1: Good fits and bad fits

Using different network topologies and learning parameters can have great effects on the outcome of the network, yielding good or bad fits. Below images illustrate some examples of

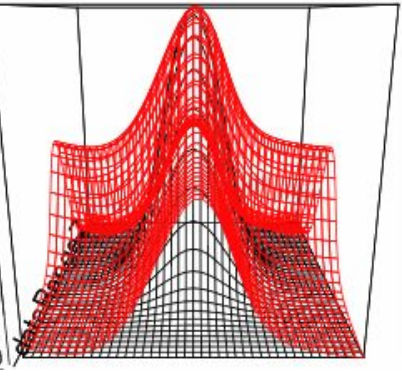
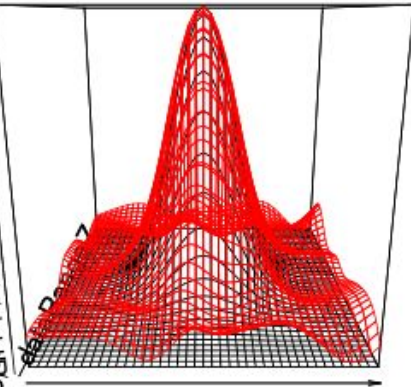
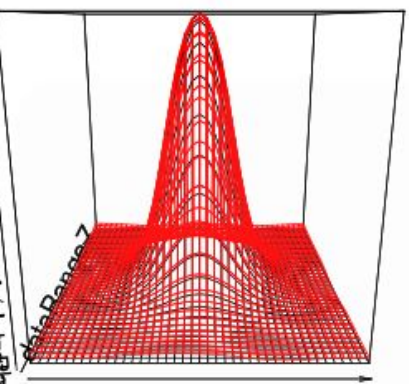
¹ <https://cran.r-project.org/web/packages/deepnet/>

good and bad fits. Black lines are the target values, and red lines indicate the network output.

$y(x)$

		
<p style="text-align: center;">Bad fit</p> <p>This fit is the result of trying to fit 15 neurons in a single layer to the function, but ramming into a local minimum which the algorithm could not escape.</p>	<p style="text-align: center;">Better fit</p> <p>Here 5 neurons in a single layer are fitted. The algorithm has converged to a global minimum, but the small number of neurons are not enough to properly fit the input.</p>	<p style="text-align: center;">Good fit</p> <p>Here two layers with 20 neurons each are fitted, yielding an almost perfect fit.</p>

$z(x)$

		
<p style="text-align: center;">Bad fit</p> <p>This is as good as the fit can get when using only 5 neurons in one hidden layer.</p>	<p style="text-align: center;">Better fit</p> <p>This fit is produced with 20 neurons in one hidden layer. Adding more neurons the fit will hardly get any better with a single hidden layer to fit the z function.</p>	<p style="text-align: center;">Good fit</p> <p>Using two hidden layers with 5 neurons in each an almost perfect fit can be produced. It beats almost any amount of neurons in a single hidden layer network.</p>

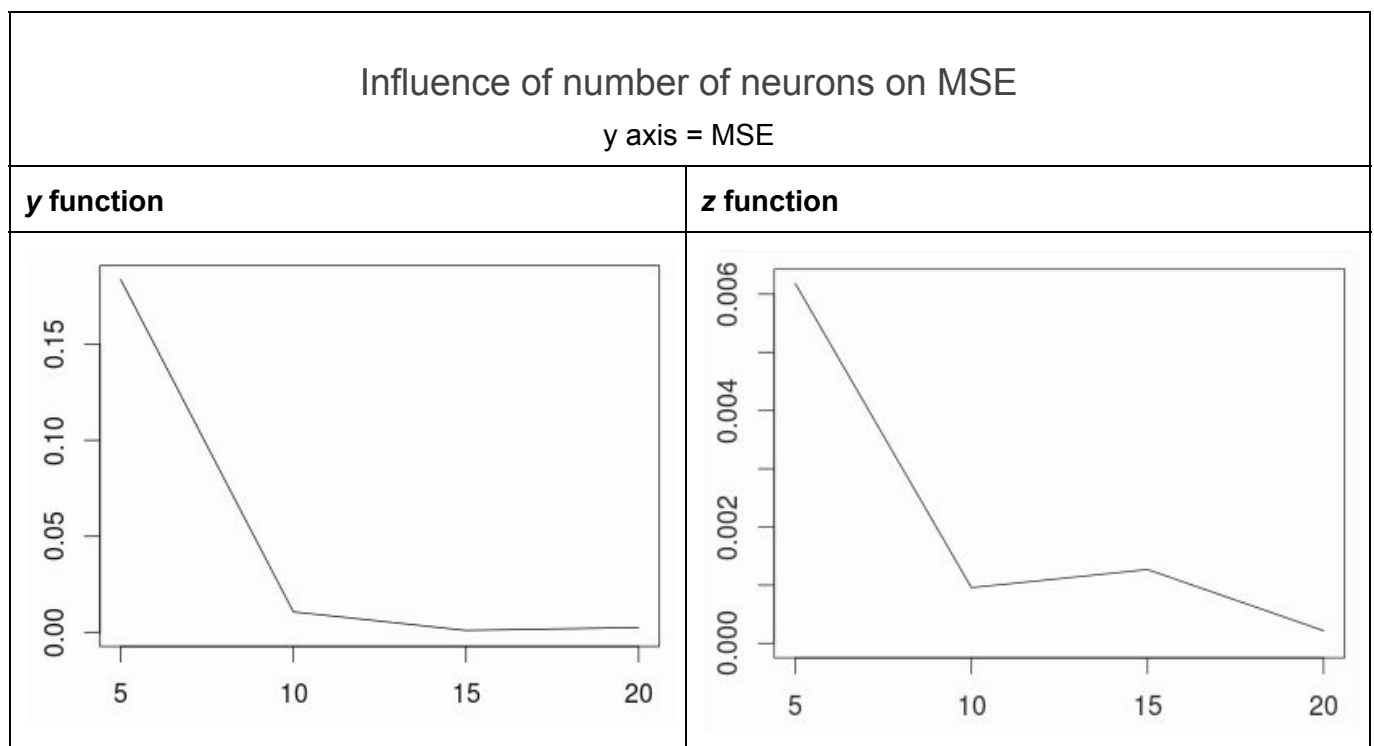
The above examples show that both the number of neurons and the number of hidden layers are important in order to get a good fit. For the y function a decent fit can be achieved using

only a single layer with enough neurons. However the z function can not be fitted properly with only a single layer. Two layers with fewer neurons are much more successful.

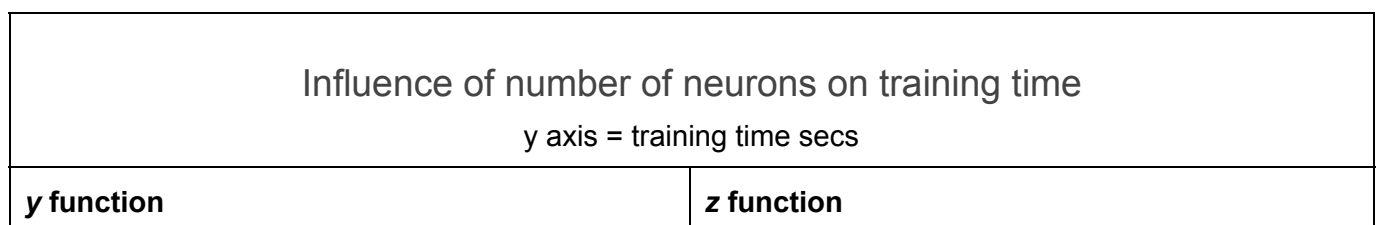
Experiment 2: Number of neurons in a single hidden layer

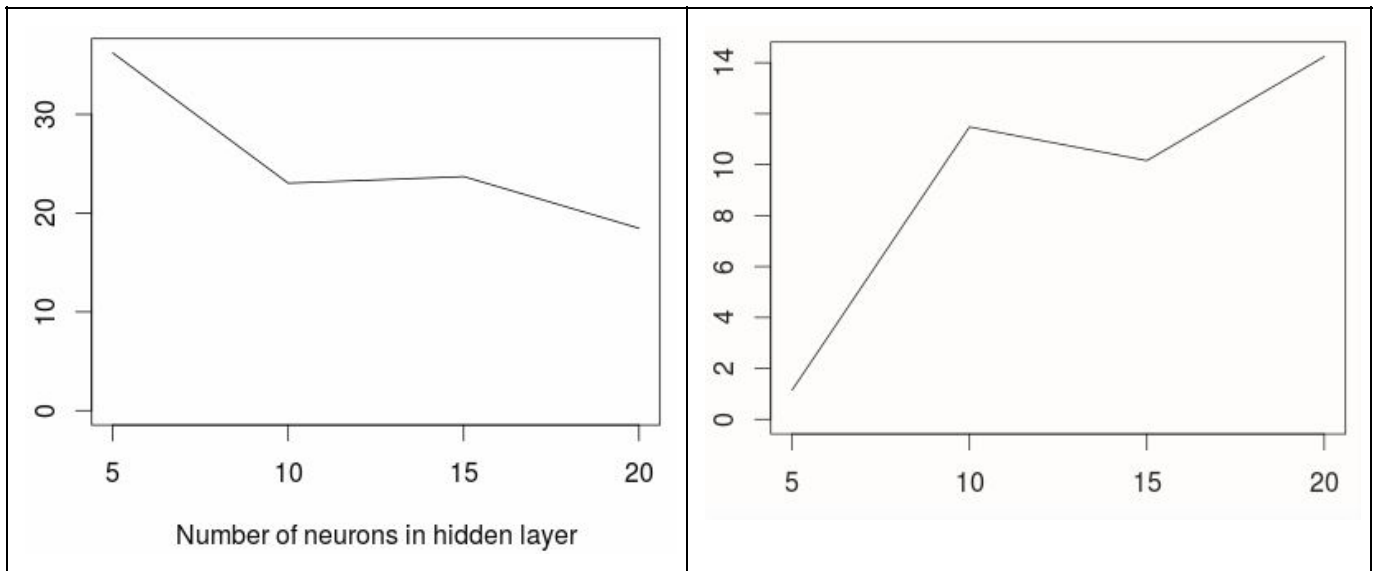
This experiment will try to investigate how the number of neurons in a single hidden layer influence training time, MSE and the number of epochs needed in order for the network to converge. The experiment was conducted by repeatedly training a neural network on the two functions respectively using 5, 10, 15 and 20 neurons in a single hidden layer ANN. In this experiment the stopping criteria is not a predetermined number of epochs, rather training is stopped when the partial derivatives of the error reach below 0.1.

Below diagrams show the results obtained from the experiment.



The results above show that the MSE of the networks decline as the number of neurons increase which is what can be expected since more neurons are able to fit more complex data. It seems that going beyond 10 neurons has no significant effect on the MSE.

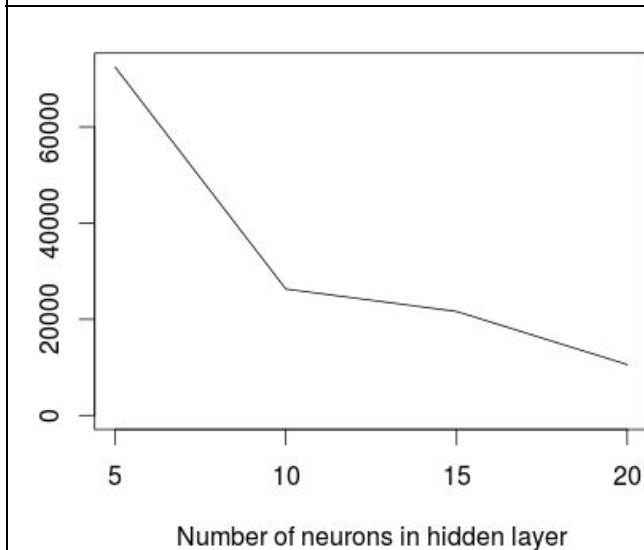




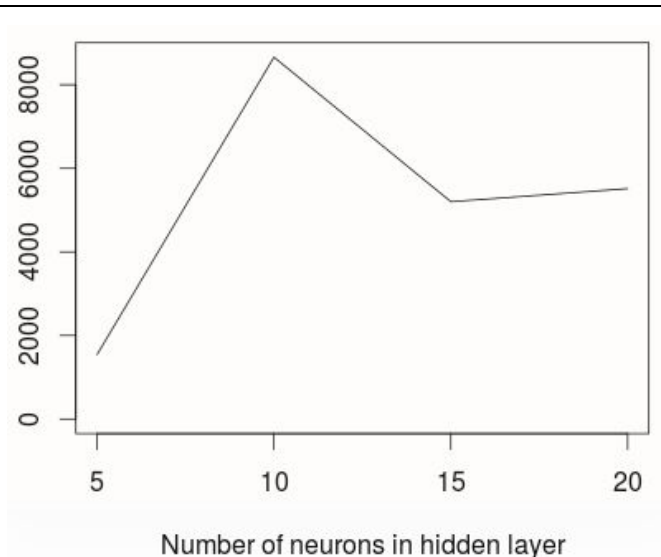
Influence of number of neurons on number of epochs

y axis = number of epochs

y function



z function

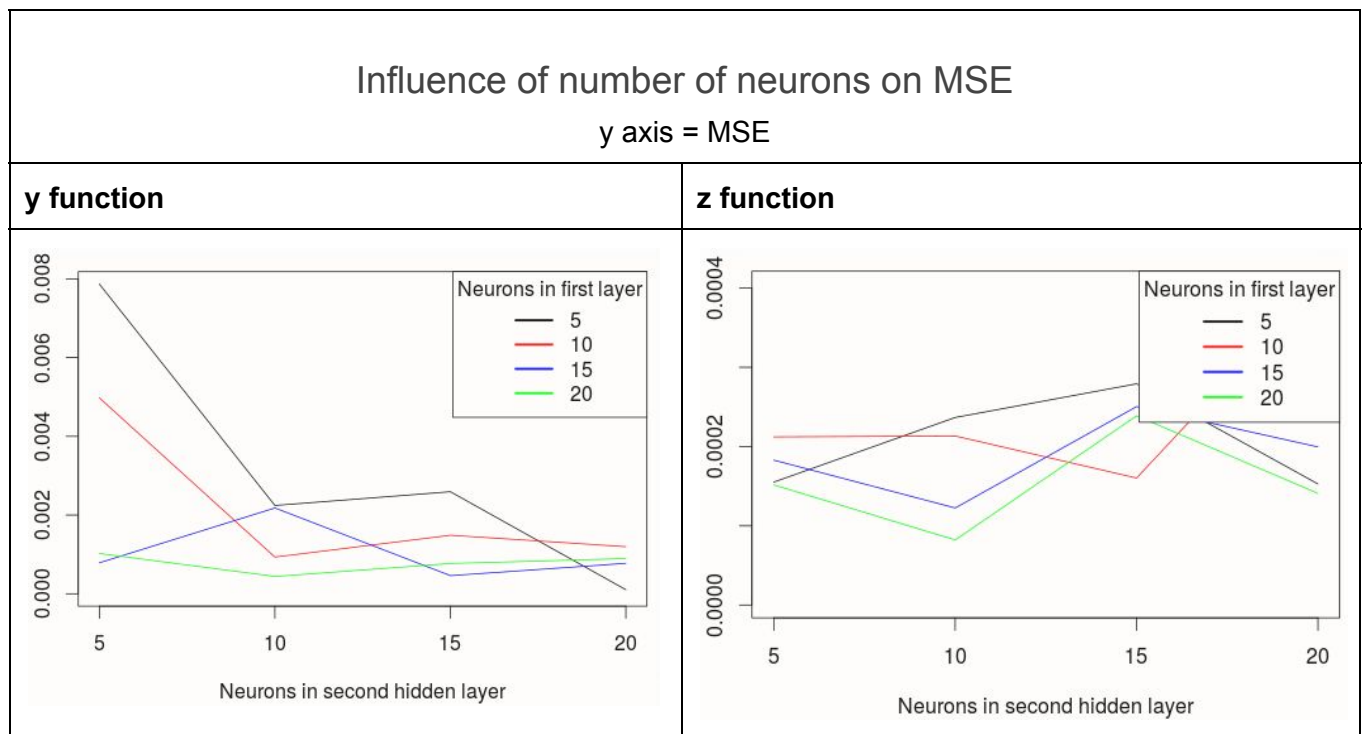


The results for training time and number of epochs show that there is a great correlation between the two, which is obvious since every iteration over the training examples takes time.

An interesting thing here is that the number of epochs are negatively correlated with the number of neurons for the y function, while they are positively correlated for the z function. The Reason that the y function is converging faster the more neurons it is given is that at 5 neurons it cannot model the function to even nearby a good fit, and the error function will respond with a high error, while once more neurons are added it fits the function much better and thereby the error function responds with a much lesser error.

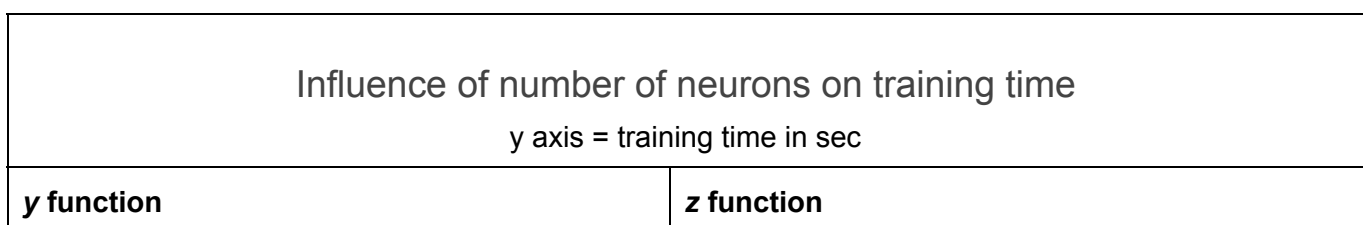
Experiment 3: Number of neurons in two hidden layers

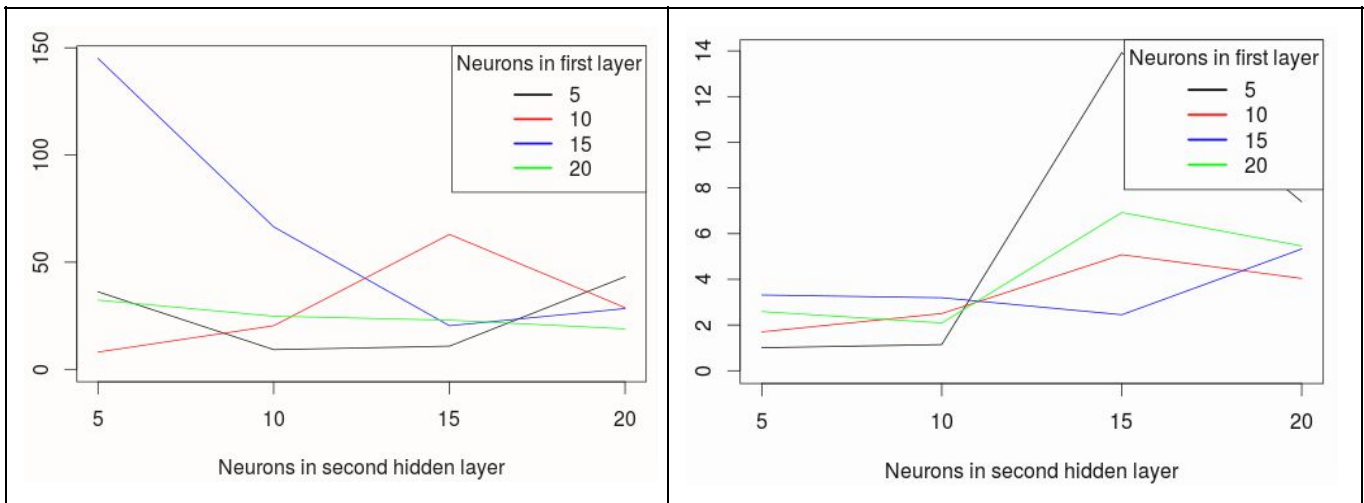
This experiment is similar to the previous, however varying number of neurons for two hidden layers will be tried out. The results obtained from this experiment will be compared to the results obtained in experiment two where only a single hidden layer was used. Below diagrams show the results for of this experiment.



First of all The MSE results for the two layer network structures show much greater accuracy as compared to the results for single layer networks from experiment 2. For the y function two layers with 5 neurons in each yield an MSE of 0.008 while the same amount of neurons in a single layer have MSE around 0.2. For the z function those numbers are 0.00025 compared to 0.001 for 10 neurons in a single layer.

Since both functions are non-linear the introduction of more hidden layer(s) appropriate since it makes it possible for the network to model non-linear behaviour. Normally one should consider the risk of overfitting the network to the training data when adding additional neurons and hidden layers, however since we are trying to fit a neural network to a function this consideration does not apply.

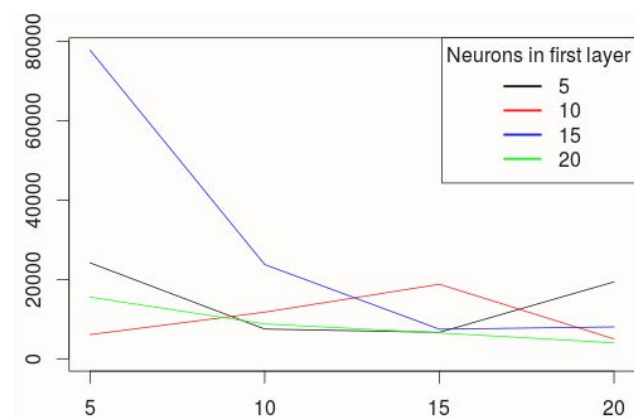




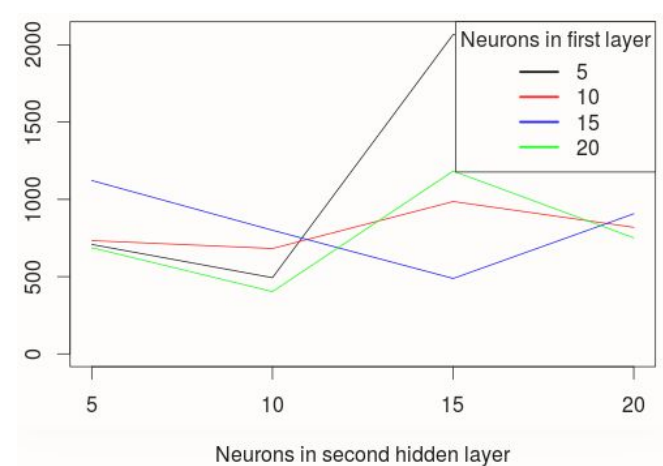
Influence of number of neurons on number of epochs

y axis = number of epochs

y function



z function



The same correlations found with the previous experiment with single layer networks seems to be present for two hidden layers. The y function converges in fewer epochs and thereby faster with more neurons while the z function does not.

The introduction of hidden an additional hidden layer seems to only mildly affect the training time for the neural networks.

Unsupervised learning: Self-organising map

This Section has been 100% written and created by Nikolaj Schaldemose Reibke
Github Repo: <https://github.com/Wisienkas/ai-collection>

Introduction

The second assignment chosen for the assignment is Self Organized Maps with Option 1: To cluster Images of digits. A slight change from the description has been that the description mention to use the mnist dataset with digits of size 15x15 pixels, but it has been chosen to do it with 28x28 pixels instead, which is the original size of the images. The images contain handwritten digits from 0 to 9, the digits is converted to have values between 0 and 1 rather than between 0 and 255.

There has been performed a lot more experiments than listed below, but they have been discarded after inspecting the resulting map, also the underlying algorithm had slight changed and any experiments previous to changes had to be discarded to have a similar algorithm basis for the experiments.

The implementation

The Self Organized Map is developed by Nikolaj Reibke in python, and the implementation of the procedure will shortly be explained in this section. The Exact implementational code is attached in the zip file and additionally available on Github at the link listed in the section.

The Python implementation has been created using mostly Matrixes rather than using an object oriented approach where the only object oriented aspect of the implementation is the entire class "SOM" which is the structure.

The Implementation has resulted in a 2d layout of neurons, using inverse time as decay function for both the kernel width and the learning rate. The Inverse time function is as follow: $f(t) = a / (t + b)$ Where t is the current iteration, a and b is factors chosen at initialization of the map.

Initialization of the Self Organized Map

The initialization of an SOMs instance is done by defining:

- Amount of input dimensions, should match the input data used to train the map
- width of map, the amount of neurons in width
- height of map, the amount of neurons in height
- Learning Rate, overall learning rate(not actual learning rate)
- Sigma, width used for gaussian kernel to define the width of a standard deviation.
- a, parameter for inverse time decay function
- b, parameter for inverse time decay function

Initially weights are randomly initialized from 0 to 1.

Training the Map

Training the self organized map is done by feeding it the input vectors in the same dimension as stated at the initialization of the map.

The step for inputting a sample is defined in the order listed below:

1. Find Learning Rate based on the current iteration.
2. Find kernel based on current iteration
3. Find best matching unit using correlation distance
4. Create a distance map from the best matching unit
5. Match the distance map with the kernel to apply the influence for each cell
6. Apply difference from input vector on each cell given the influence in step 5 and learning rate from step 1
7. Increase Iteration by 1

The results

This chapter contains all the results from the maps. The table shows the parameters for the experiments. All the results from the results is attached with the report in a zip.

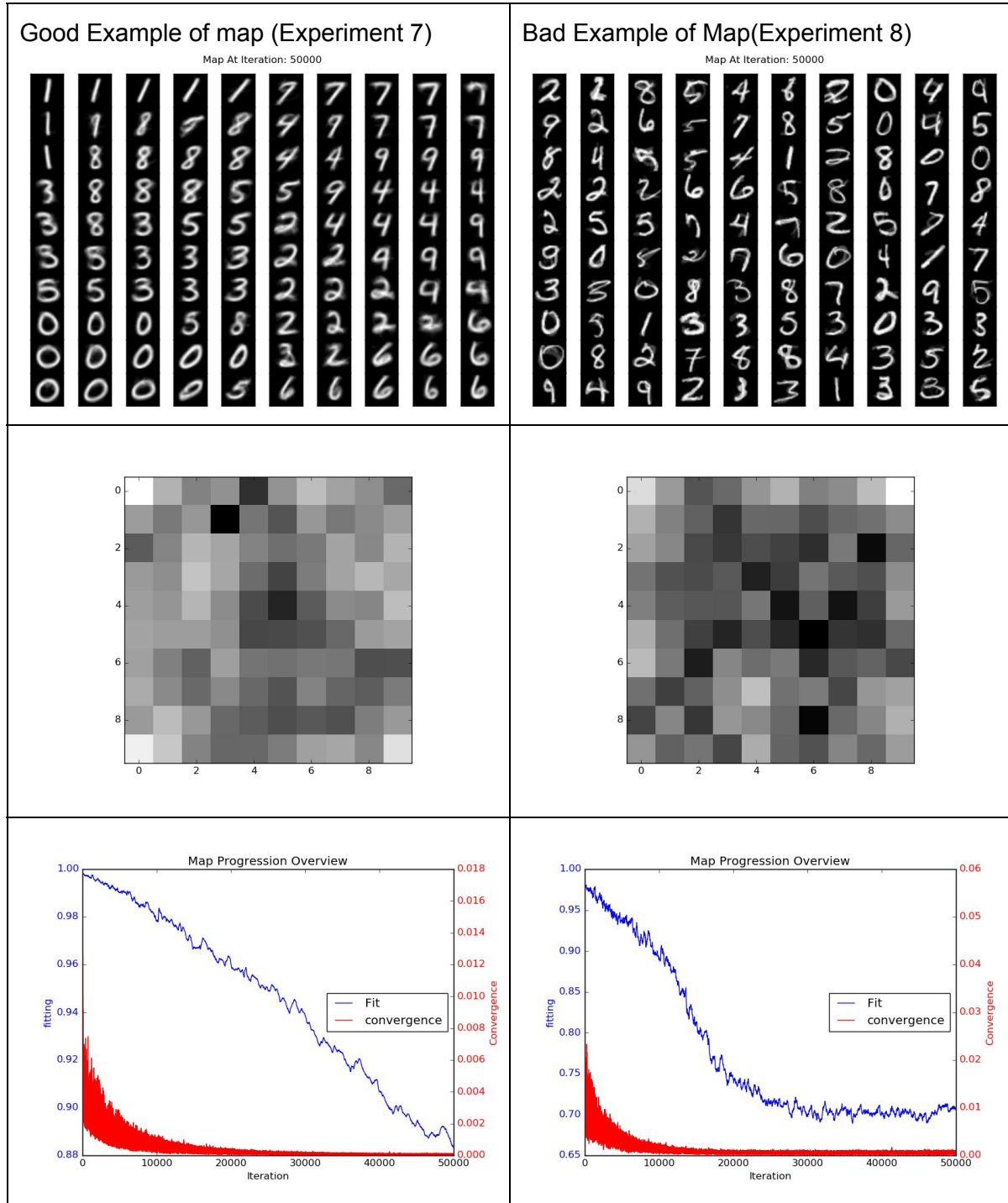
The results is contained in a graph showing the overall change in the map and the fit of the map given the iterations. Additionally both fitting gray maps has been made and the actual map is also available.

Ex. no.	Learning	Sigma	A	B	Iterations run	Went well?
1	0.15	1	1000	200	20.000	Yes
2	0.15	3	1000	200	20.000	No
3	0.01	3	1000	200	20.000	No
4	0.15	1	10000	2000	20.000	No
5	0.05	2	2000	2000	50.000	Yes
6	0.05	0.5	5000	1000	50.000	Yes
7	0.05	0.25	5000	5000	50.000	Very Much
8	0.2	0.12	5000	5000	50.000	To Specific
9	0.05	0.25	10000	5000	100.000	Okay
10	0.05	0.2	500	50	50.000	Okay
11	0.05	0.2	100	10	5.000	Best So Far

12	0.6	0.7	30	30	1000	Best at 350
----	-----	-----	----	----	------	-------------

Example of good and bad maps

For different parameters given to the Self Organized map it can converge in different ways, and in some cases it might take very long to converge. For all the experiments a max amount of iterations was chosen and some “snapshots” of the map was taken during the run.

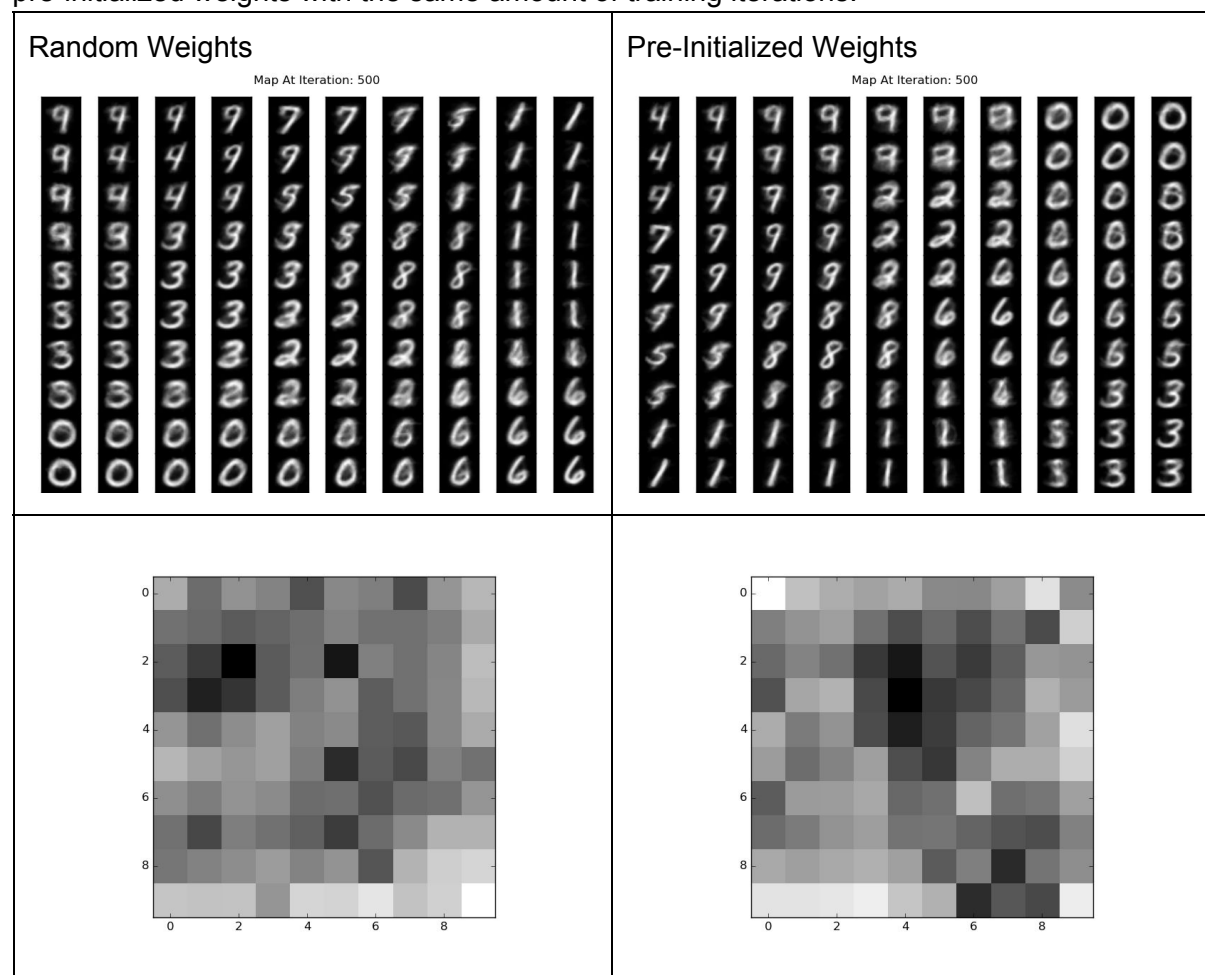


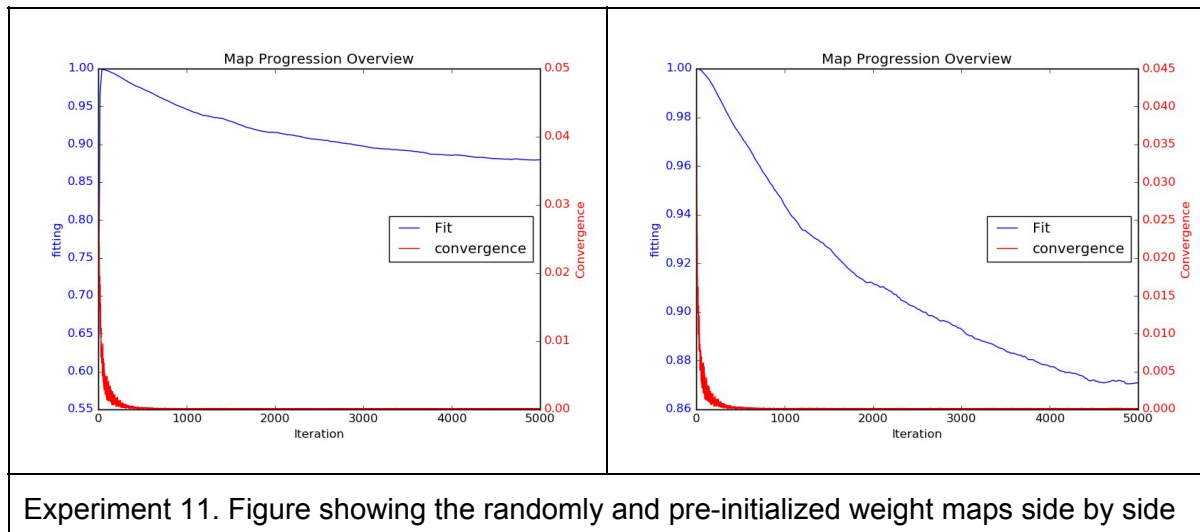
In the two examples given above, the good example has clusters of digits which is correlated to each other next to each other. As the grayscale fitting map shows it is seen that a few points on the map is not super correlated, but compared to the fitting map of the bad example a lot of the map has no or very bad correlation to the neighbours.

The graphs also given for both the good and the bad example shows a clear difference in fitting as well. Where at the 50000th iteration the good example have a mean u-matrix score of 0.88 and the bad example has stomped down to 0.70. The U-matrix correlation has been calculated where each cell look at the 8 cells around it.

Randomly versus Pre-Initialized Weights and Performance Tweaking

Will explain the differences found between the randomly initialized weights and the pre-initialized weights with the same amount of training iterations.





Experiment 11. Figure showing the randomly and pre-initialized weight maps side by side

From the looks of it, not just from the example shown here, but in general all the experiments, it was found that there wasn't that much of a difference between the two in terms of how well the clusters was collected and how well the maps was fitted compared to each other, they have around the same fit for the same iteration all the way through. The graphs can be a little deceiving as the fitting line for the randomly initialized self organized map starts at a very bad fitting and thereby the graph include that number into the range interval. But after very few iterations the fitting goes straight up to 1 as both the learning and the kernel in the start is very broad, causing all the cells to be very similar.

A slight difference between the two cases is seen by looking at the convergence rate which is the red line shown in the graphs, where the pre-initialized map seems to converge slightly faster than the randomly initialized map.

Performance wise another trait was found to greatly impact at which speed the map would converge. Where many of the first experiments only obtained somewhat mediocre results even after running 20k to 50k iterations, it was later figured that the decay function was greatly impacting at what rate the map could converge. It seemed that a combination the learning rate given, the width of the kernel and the decay functions parameters could greatly change the convergence rate, which went down to having a good map already at 500 iterations. The sigma had to have the correct width at first and then decay at the correct speed, where the cells would still be allowed to being influenced enough to contain actual partial matching units to the dataset.

Having a decay which went to slow and the map would be influenced too much by all the number which would both cause the same digits to be spread into multiple clusters around the map, but would also cause the map to take much longer to converge.

Having a decay which went to fast and you'd risk to have empty fields which never became a best matching unit for any entries in the dataset.

The overall best sigma found at a 10 x 10 map was 0.2.

The overall best learning rate found was 0.05

The overall best parameters for the inverse time function was 100 and 10.

Discussion

Will go through some of the experiments which has been conducted in order to discuss what went right and what went wrong. The parameters for all the experiments can be found in the top of the section Self Organized Maps.

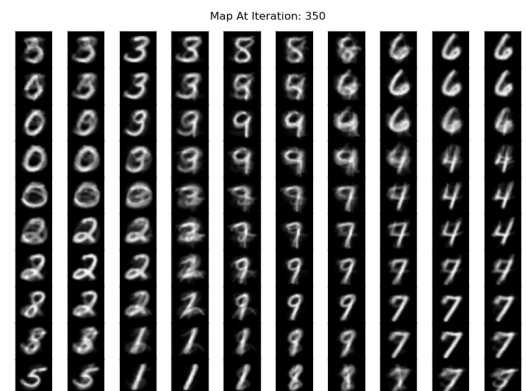
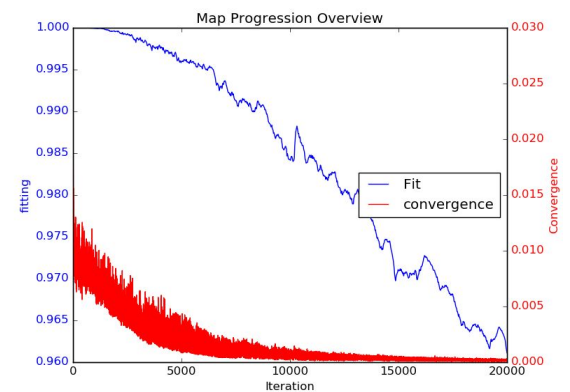
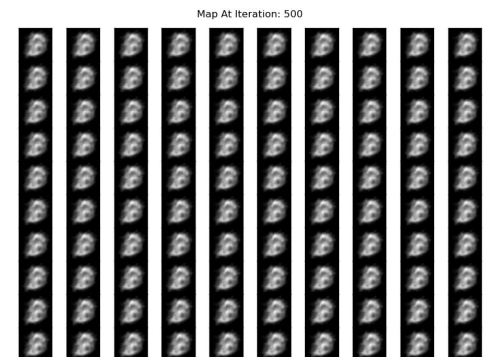
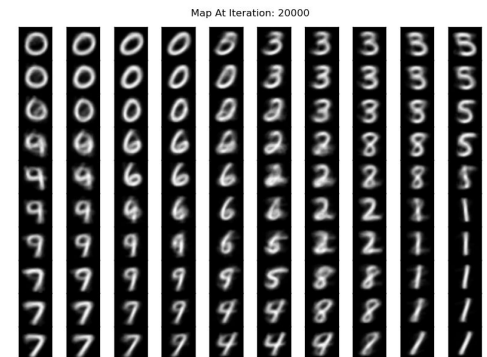
Experiments

The first experiment seen on the right, is the pre-initialized map, converged well for the iterations given in the first experiment. Unfortunately it had a few clusters like the 8's which can be found in 2 separate clusters on the map. It seems that the reason that there are 2 separate clusters of 8's is because the 2 clusters of 8 differ in their rotation, where the 8's in the upper right side of the map are straight up. the 8's to the left of the 1's in the right bottom corner are leaning to the right. the issue with digits being slightly rotated from each other has caused bisection of said digits in multiple experiments.

Additionally a few digits seem to be very similar especially if written in very specific formats. To list a few of these similarities between digits:

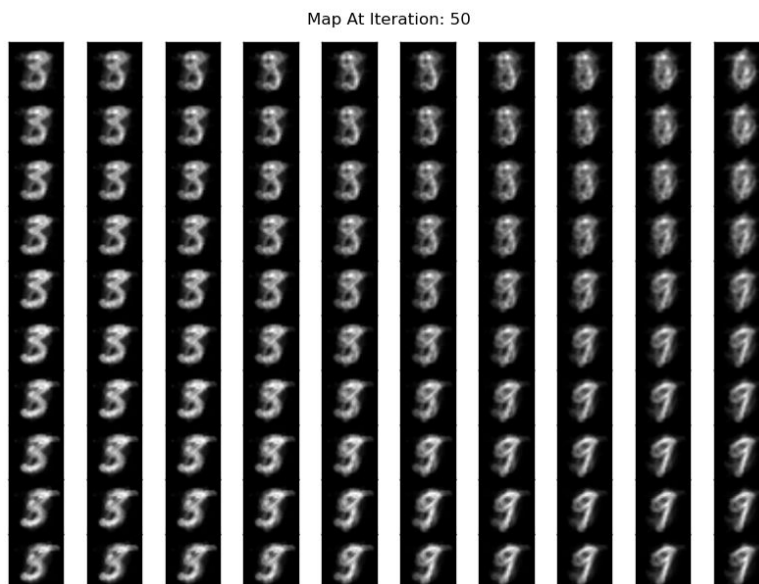
- 4 has seen to be distinguished very closely to 9.
- 7 has also been seen to blend into 9's
- 1 and very narrow 8's often gets connected
- 5 and 8's generally is connected
- 0 connects to multiple sources like 6, 8, 9 and 5
- 3 and 2 are very similar

Returning to the actual experiment another unfortunate variable which was very important was the time that it took to actually have the experiment converge compared to experiment 11, showcased in the previous section. The map on the right side of the list above is experiment 1 at 500 iterations, and as seen it has not converged at all, but is rather just a blurring combination of all the numbers. But the graph says it contains a fitting score of 1 or 0.999 at 500 iterations. But what the fitting tells us is just that for each cell, its neighbouring cell looks more or less the same, which is true to the case as all the cells is just nearly the same combination of all the numbers. It also gives another important information that a fitting of 1 is not good. But rather a fitting around 0.9 to 0.95 is better. Additionally

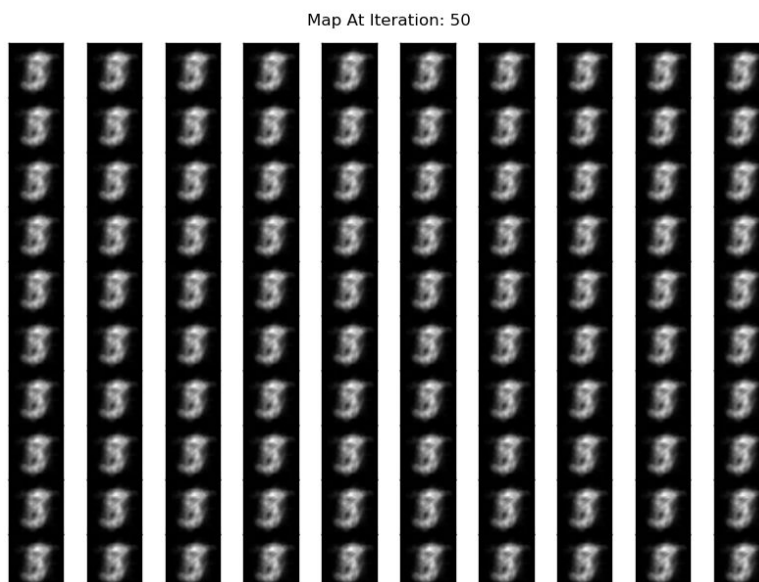


it gives away the fact that the width of the neighbouring function is way too high, since all the cells are looking the same. Given a smaller gaussian kernel the map would be less like the above example shown of Experiment 1 at 500 iterations.

Experiment 12 tries to achieve not filling the entire map with the same value for every cell in the start by having a very fast decay function and a moderate sigma kernel, along with a high learning rate to affect the individual cell more. Also seen in the below figure from experiment 12 at 50 iterations it has cells which are not all the same, which is unlike all the other experiments.



Another figure which does not achieve this is Experiment 10 where it is seen that all the cells are looking alike.



All the additional images can be found in the folder “experiments” in the zip folder attached. The naming schema is that each experiment starts with “experimentX-” where X is the experiment number, followed by either “non-” for randomly initialized or “pre-” for pre-initialized. After that the maps is contained as “mapAt” followed by the Iteration the

snapshot was taken. the same goes for the U-matrix which instead of "mapAt" has "goodnessAt" and the graph for the entire experiment is found as "experiment{number}-{pre/non}-graph".

A overview of all the experiment graphs

Conclusion

The self organized map is a structure which can cluster inputs to be represented in less space and will be clustered where the input vectors are looking a like.

When training a self organized map it is important to choose the right the parameters for the self organized map in order to optimize the speed of the training and the convergence of the map. The combination of the correct choice in decay function, neighbourhood function and learning rate can be crucial in the amount of time it takes for the map to be converges. As shown in the experiment, the right parameters can change the amount of iterations required from 350 iterations to 100.000 iterations.

For reducing the havoc among the map, it can be pre-initialized in order to make the input vectors which are alike, more likely to be matching the cells which are also nearby, if the self organized map is pre-initialized corrected. But it might not give that much if many data is very like each other and especially if a group can have sub groups which have differences which are bigger than the differences to another type.