

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF ARTIFICIAL INTELLIGENCE

|  |
| --- |
|  |
|  |
| Artificial neural networksPaper Florentijn Degroote  R0575914  [florentijn.degroote@student.kuleuven.be](mailto:florentijn.degroote@student.kuleuven.be)  2018-2019 |



# Assignment 1

## Perceptron

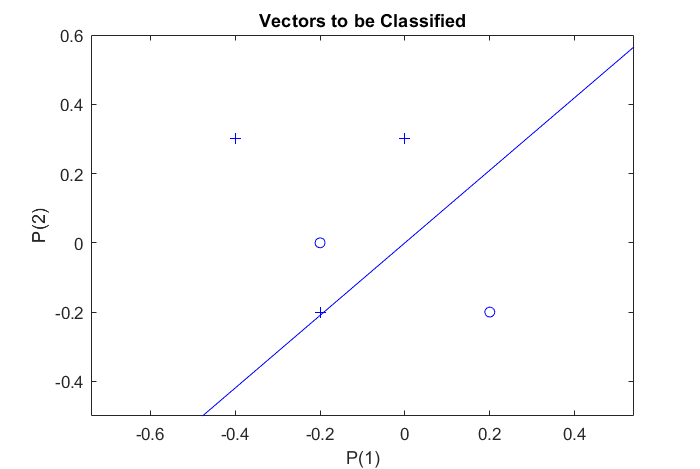
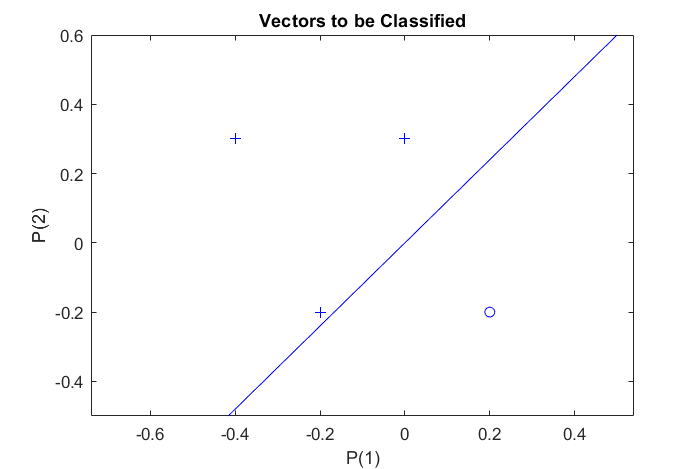


Figure 2

Figure 1



A perceptron is able to classify linearly separable data, as shown in figure 1. Its limits are reached when trying to classify the points as shown in figure 2. The most basic case where the perceptron fails as a perfect classifier is in the XOR problem ([Minsky, 2017](#Minsky2017)).

## Backpropagation in feedforward multi-layer networks

#### Function approximation

In order to approximate the function in the range of , 189 equally spaced points of distance 0.05 were generated. 75 percent of these points were randomly chosen to be part of the training set, thus assigning the remaining 47 points to the test set. For the first bag of tests, no noise was added. The data is fed 50 times (50 epochs) to a feedforward two-layer network, varying the number of neurons (1,5,20,50, and 200) and training algorithms (gradient descent (“traingd”), gradient descent with adaptive learning rate (‘traingda”), Fletcher-Reeves conjugate gradient descent (“traincgf”), Polak-Ribiere conjugate gradient algorithm (“traincgp”), BFGS quasi Newton algorithm (“trainbfg”), Levenberg-Marquardt algorithm (“trainlm”) and Bayesian regularization backpropagation (“trainbr”)) and using the default values for the other parameters. The results for computation time for the training phase and RMSE (root mean squared error) of both the training and test set were measured[[1]](#footnote-1). The results can be found in figures 3,4, and 5, respectively.

Figure 5

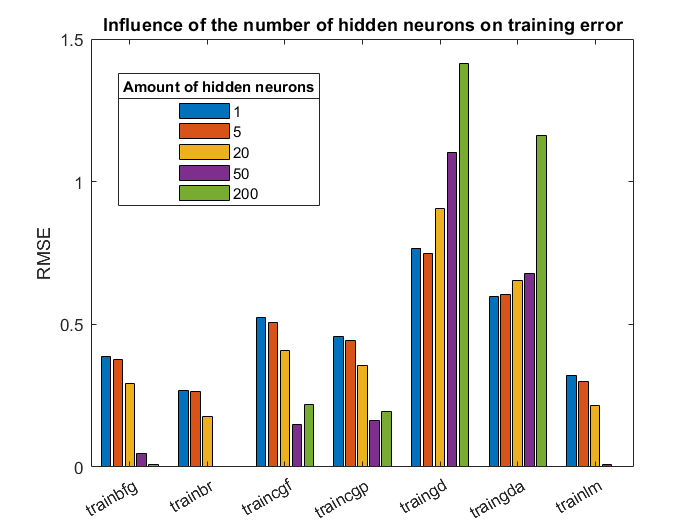
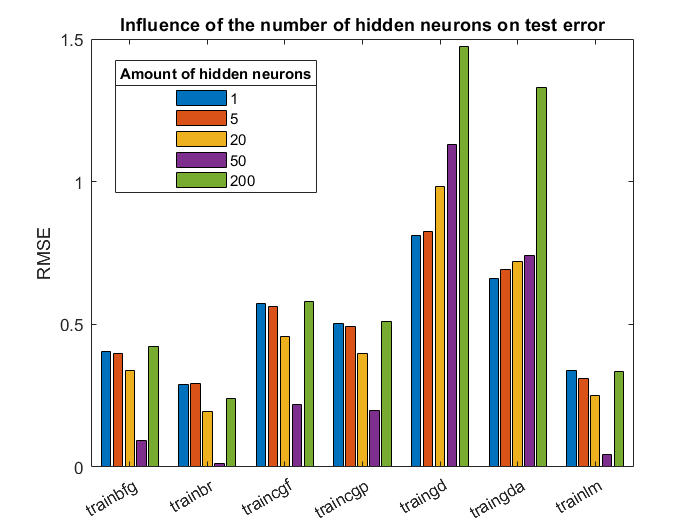


Figure 4

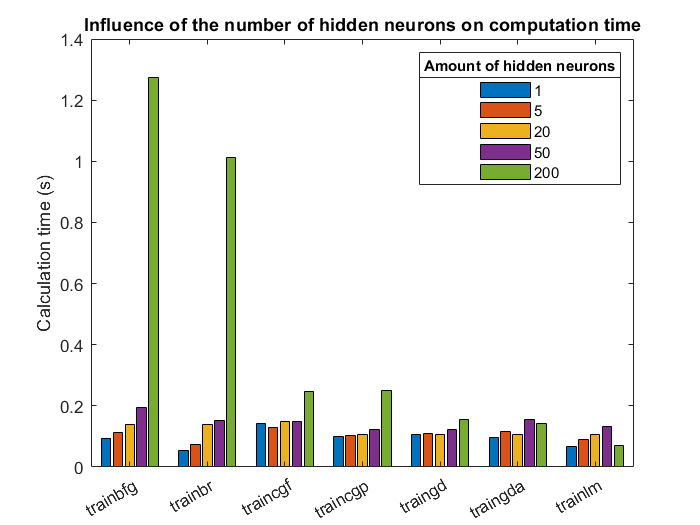


Figure 3

Because a bigger amount of hidden neurons involves more computations, there is a general rising trend in computation time for bigger networks. Computation times for 200 hidden neurons are remarkably big for trainbfg and trainbr in comparison to the others, including traingd. For trainbr this can be explained by the expensive posterior calculations. The decrease in computation time for trainlm is due to only completing 3 epochs before the algorithm stops automatically. This happens because the measured performance is falling below the default threshold. In figure 4 it is shown that for most algorithms, the training set’s RMSE is decreasing for an increasing amount of hidden neurons. Here, the added extra neurons introduce more flexibility and allow the model to fit the training data better. Though, this is not the case in the two gradient descent algorithms. The extra neurons introduce an extra complexity that traingd and traingda can’t cope with. This is a form of overparametrization. Remark that we only have 142 elements in the training set, which is less than the amount of hidden neurons in the most complex case! It is also noted that the same is happening when looking at the graph for the test error. In this graph, the RMSE per input is higher than it is for the training set, overall. This is logical, because the neural net is trained on the training data, not on the test data. The test data is “new” for the neural network. Notice that trainbr, trainbfg and trainlm have quite nice results for the combination of training set and test set for 50 hidden neurons. On the other hand, all except trainlm of these settings do impose a bigger computation time compared to the alternative learning algorithms. For 200 hidden neurons, the differences in RMSE are big, indicating overfitting/overparametrization. In the case of trainbr, the inherent regularizing characteristic limits the amount of overfitting (values of the weights are kept low).

The second bag of experiments were similar to the ones before. The difference here is that there is a gaussian noise added to every target with a mean of 0 and standard deviation of 0,5. The results can be found in figure 6,7, and 8. Again, overfitting is easily detectable for 200 neurons. As expected, the results of computation times and RMSE per input in the training set are also similar to the previous results, although the RMSE of the training set is slightly higher, because the added noise is not modelled perfectly (and shouldn’t be, otherwise the model is overfitted). The RMSE per input in the test set is higher than before, however. Especially for the aforementioned cases of 50 neurons because the noisy data introduces differences in what the model predicts and what the target is.

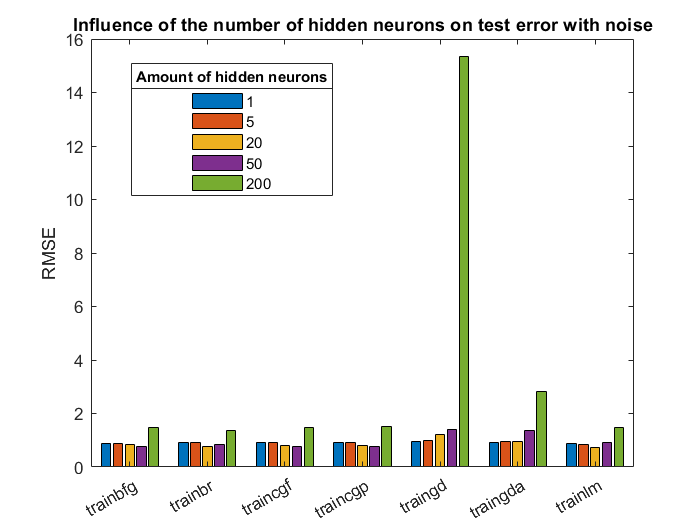


Figure 8

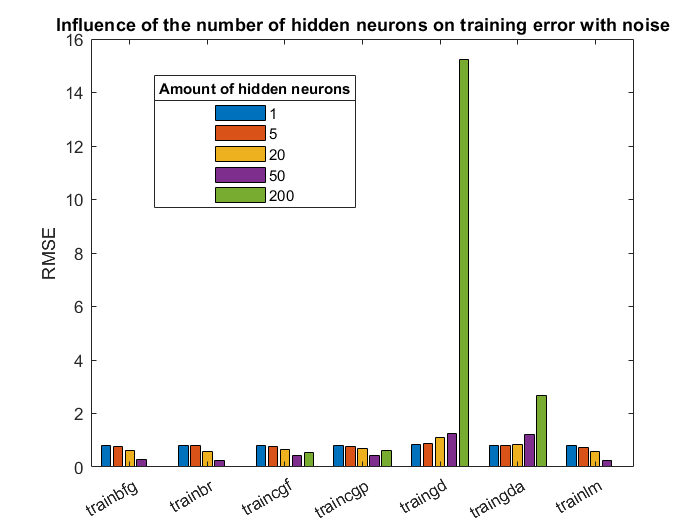


Figure 7

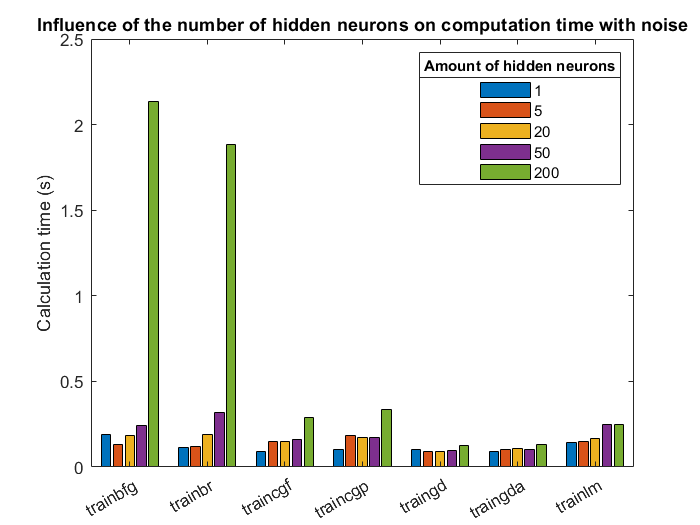


Figure 6

#### Personal regression

Based on my student number, is equal to . Three independent samples were generated (using random generators constructed in bulk), containing 1000 samples each. The first sample will be used as training set and is thus used to train the neural nets with. The next 1000 samples total the validation set, used to tune the architectural/hyperparameters of the model and see how well the model is trained. The last 1000 samples are used as test set. The test set is only used as a last test, to check whether overfitting (to what extent the model was trained to fit the validation set only) occurred. The results of the test set should not be used to alter parameters upon.

As before, during the experiment phase, we will use 50 epochs. Because trainlm, trainbr, and tranbfg gave the best results in the previous tests, because the setting is quite similar (though training set is larger) and because the goal here is to minimize errors, we will check the results of these two. The results in figure 9, 10, and 11 were obtained for a varying amount of neurons and using tansig as activation function in the one and only hidden layer.

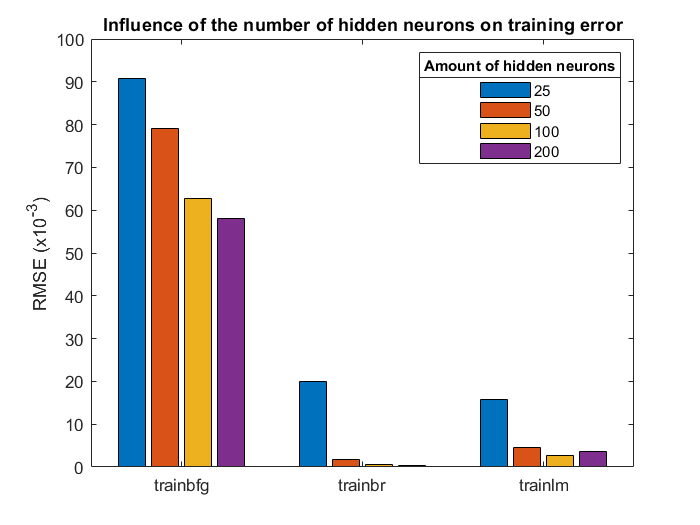


Figure 10

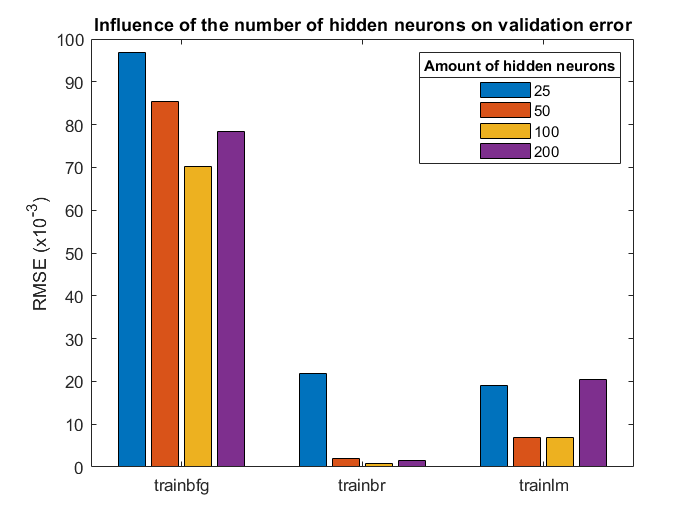


Figure 11

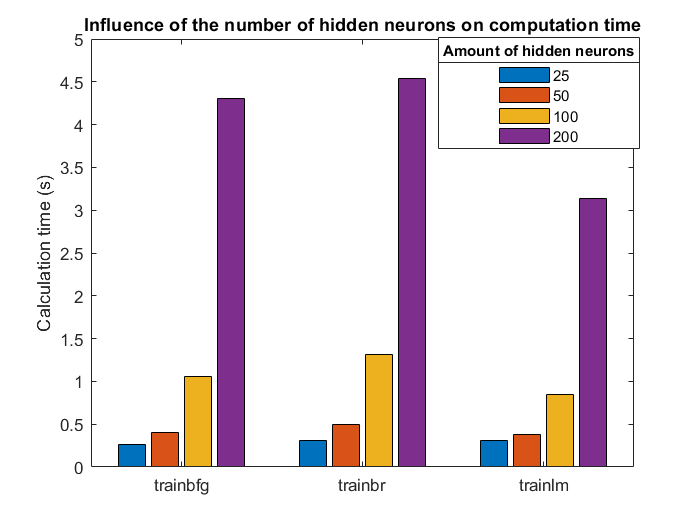


Figure 9

Because the results of Bayesian regularization seem to outperform the Levenberg-Marquardt algorithm, we will continue with this learning algorithm (note that the Levenberg-Marquardt is used in Bayesian regularization to update the weights and bias). Here, 100 hidden neurons offer a good trade-off between computation time and RMSE’s.

In figures 12 and 13, results are shown for 2 and 3 hidden layers of the net, respectively. These shallow networks are computationally expensive and more complex. The examined combinations of number of hidden layers and hidden neurons are not outperforming the one-hidden-layer network, so we will stick with the latter.

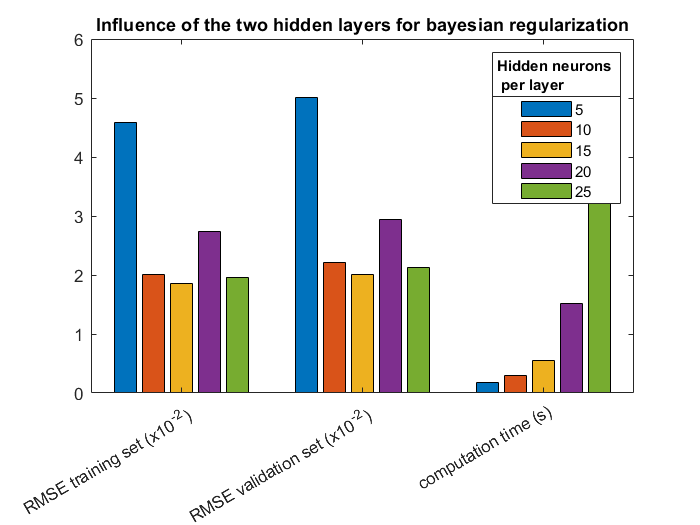


Figure 12

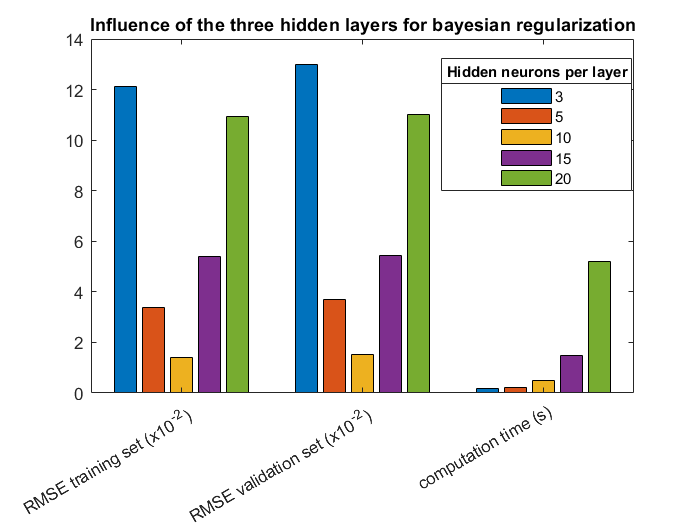


Figure 13

Now that the number of hidden layers and neurons are determined, one can wonder whether the default (tansig : tangent hyperbolic) is really the best performing activation function in this case. On figure 14, the results are shown for four different activation functions. Here, the results of 10 different runs (rng(1) up until rng(10)) were averaged. The tansig seems to be superior to others.

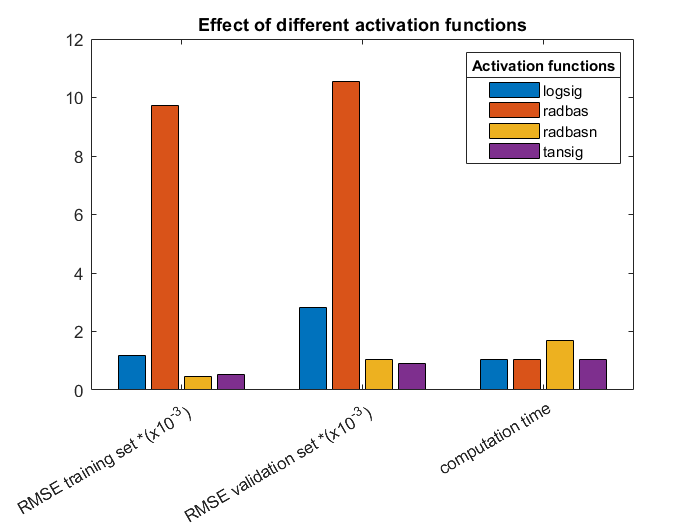


Figure 14

The resulting network is thus composed from a feedforward neural net with one hidden layer containing 100 hidden neurons. The activation function used in this hidden layer is the tangent hyperbolic. For the output layer, the standard hard-limit transfer function is used. Let’s now carry out a performance assessment on the test set. Figure 15 shows a 3D scatterplot of the targets of the training set. Figure 16 shows the targets of the test set (dots changing colours in the X1 direction) and the output of the network on the test inputs (black circles). We can conclude that the test set resembles the same distribution as the training set and that the net is capable of mimicking this distribution just fine. Table 1 shows the squared mean error values of training, validation and test set. These are all extremely low. The difference between test and training set is quite big, in proportion. On figure 17 we can indeed see that there is a visible difference between the two. More epochs for this network could lead to overfitting. On the regression plots generated by the nntraintool, Matlab gives the R squared a rounded value of 1, indicating a near perfect fit.

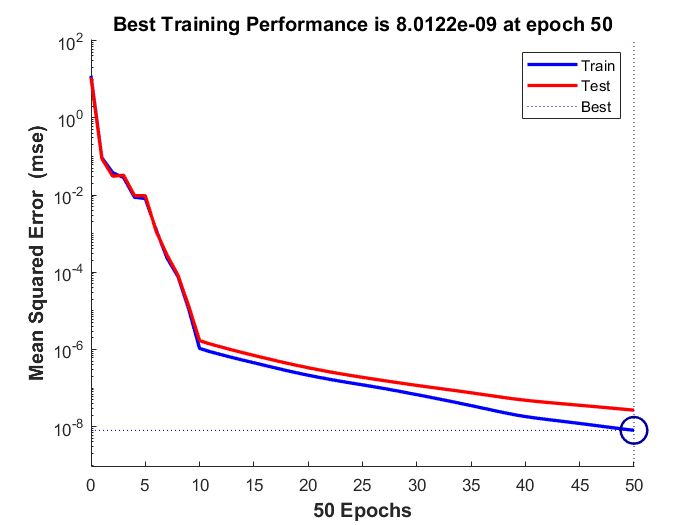


Figure 17



Figure 15

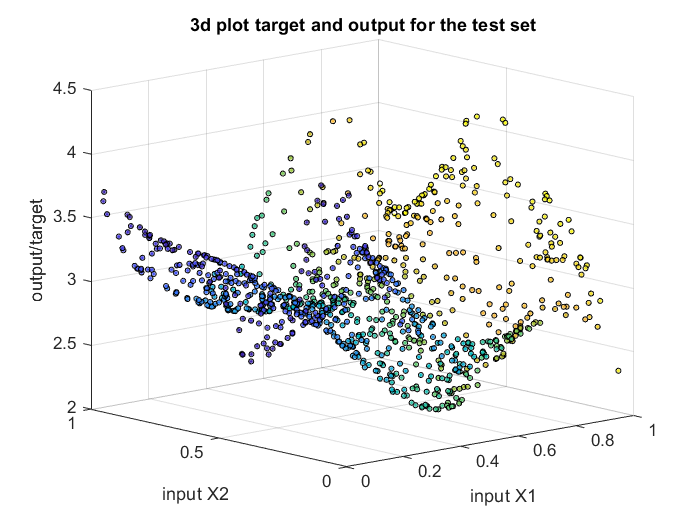


Figure 16

Of course, this method of setting up hyperparameters in order to find a good/optimal neural net is not guaranteed to generate the best possible neural network. The focus could lie elsewhere; when for example time was the focal point, a network with more hidden neurons would be chosen instead. Also, the order in which the hyperparameters are determined can be altered in order to find another (and maybe better) neural network. Hyperparameters can be tuned repetitively, or more values of hyperparameters could be taken into account in the tests. For illustrative purposes, a test was ran using 950 epochs for both 50 and 100 hidden neurons. We can clearly see that overfitting is

|  |  |
| --- | --- |
| Data set | SME value |
| Training set |  |
| Validation set |  |
| Test set |  |

Table 1

# Assignment 2

## Hopfield Network

When input is given to a Hopfield network, the network converges to the closest attractor corresponding with that input. However, the possibility exists that the input converges to a stable state x ∉ P that is not part of the attractor set P the network is initialized with (stored patterns). In other words, they correspond to local minima in the energy function that shouldn't be there (false patterns).

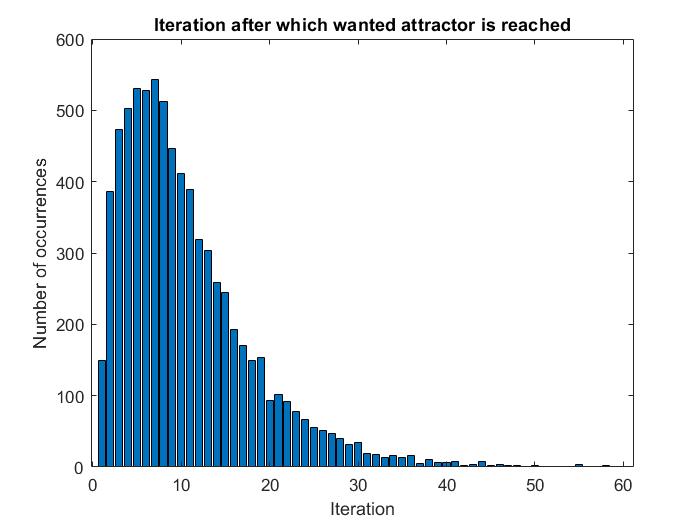


Figure 18

In a first stadium, a Hopfield network with three 2-dimensional attractors was constructed: (1;1), (-1;-1), and (1;-1). 10000 new 2-dimensional points were randomly generated and fed to the network. Figure 18 shows how many iterations were necessary before the network evolved to one of the three desired states. As you can see, this data seems poisson distributed and has its mean around 10 iterations. In this test, the network converged 7518 (roughly 75 percent) times to one of the three states. In the remaining cases, the network converged to the state (-1:1). This is thus a spurred state/false pattern for this network. But there are other states the network can get stuck in. Initial states of high symmetry (for example (0,0)) will not change or get stuck in other local minima, because the attraction of some (in case of (0,0) all) attractors is equally big. Similar results were also registered for a 3D hopfield network with states (1;1;1), (-1;-1;1), and (1;-1;-1). The local minima of high symmetry in this case are for example (0.366..;-0.366..;0.366..) when started with (0;0;0) or (-0.0605;-1;-0.0605) when started with (0;y;0) where y < 0. Changing the algorithm so that little changes are randomly made in the values of the features of the state could solve this problem.

A hopfield network can also be used to recognize digits. Using the provided hopdigit() function, tests were conducted with different gaussian noise levels on the pixels of the digits that are fed to the network and with a varying number of iterations of the network. The 3D graph in figure 19 shows the number of correctly recovered digits for different combinations of noise. Figure 20 depicts the number of spurious states for these same combinations. Here, the spurious states for low numbers of iterations are more often than not unstable states (the hopfield network did not yet converge). It is clear that when the noise level is increasing, the hopfield is having a harder time at correctly labelling the images. When the number of iterations is increasing, the hopfield has more time to converge and is able to recover the correct digits better. Also, when the noise level is small, the image is not distorted too much and the hopfield network needs less iterations to get to the stored patterns. When the noise level is becoming big (>10), the accuracy is decreasing sharply and is not perfermong much better than random guessing. Investigating the results for these high noise levels in combination with enough iterations (>50), a spurred state was discovered. This false pattern is graphically reconstructed and depicted in figure 21. The reasons why the network is not able to reconstruct the noisy digits are that -for high noise levels- the images are simply too distorted and that sometimes, the network converges to false patterns, which are not part of the stored pattern set.



Figure 21

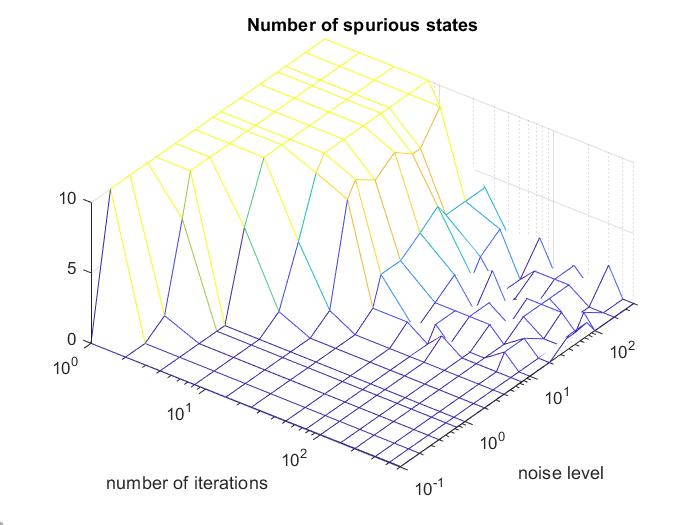


Figure 20

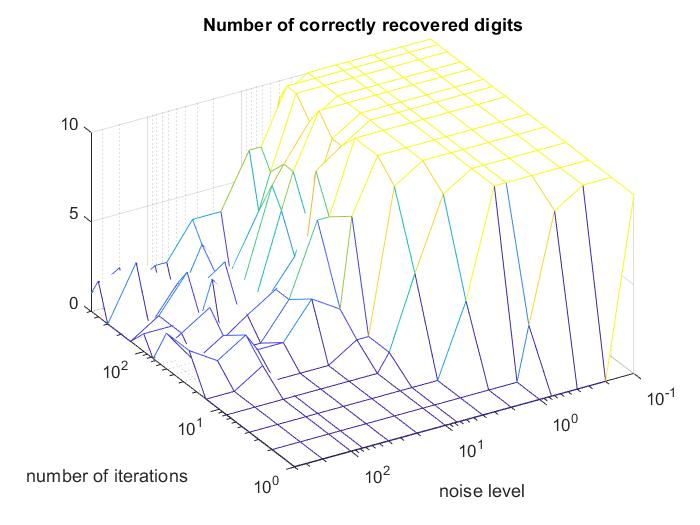


Figure 19

## Long Short-Term Memory Networks

#### Neural Network

Using a neural network, we can try to predict the test set of size 100 of the Santa Fe data set. In a first step, this data is normalized. This is done to obtain a better fit and to prevent the training from diverging. As training set we use the training set which consists out of 1000 consecutive data points. This training set is partitioned, based on the size of the lag. When the lag size is 5, the data set is split in such a way that the data is split into a number of consecutive parts of size 6. In this setting, the first 5 datapoints are fed into the network and the 6th datapoint is used as target. That means that de input of the neural network has exactly a lag amount of neurons. The second parameter that can be altered is the size of the number of hidden neurons in the first and only hidden layer. When the training phase is over (50 epochs), the test phase can commence by feeding the last lag amount of datapoints of the test set in the MLP. The input is the MLP is then shifted so that the remotest time step is dropped and the output of the MLP is added as the most recent input. As a measure of quality, RMSE of the standardized data is wielded. In a preliminary analysis, combinations of number of hidden neurons (1,2,5,10,25,50) and lag size (1,5,10,25,50,100,200,500,999) were tried out to see which setting gives the lowest overall RMSE on the first *50* datapoints of the test set. In the 3D plot of figure 22 the inverted value of the RMSE is depicted with respect to the two parameters. Because the inverted RMSE is used, good configurations are easily detected. The plot shows that 2 configurations are worth looking at, namely {lag: 50, hidden neurons: 5}, which has a RMSE of 0.0870 and {lag: 100, hidden neurons: 25}, which has a RMSE of 0.1108. The third peak in the plot is from the configuration {lag: 50, hidden neurons: 50}, with a RMSE of 0.1046, is disregarded because it is simply a more parametrized model than the first one mentioned with higher RMSE.

Now that the two settings are chosen to further be looked into, we can try to predict the full 100 datapoints instead of 50. Figure 23 shows the results for the first configuration, figure 24 depicts the second.

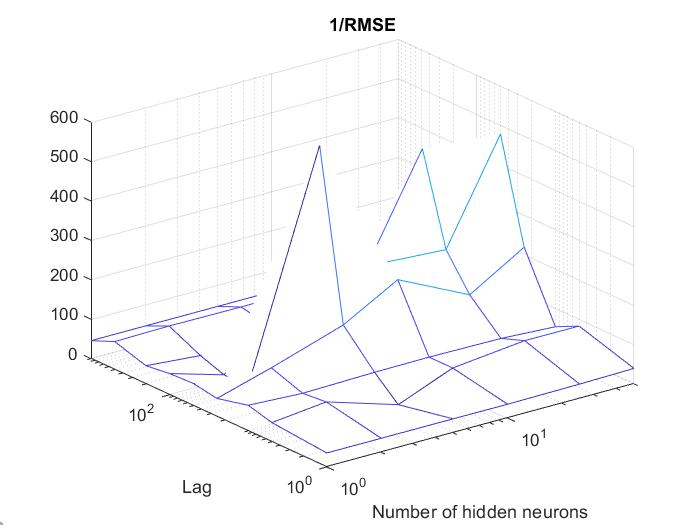


Figure 22

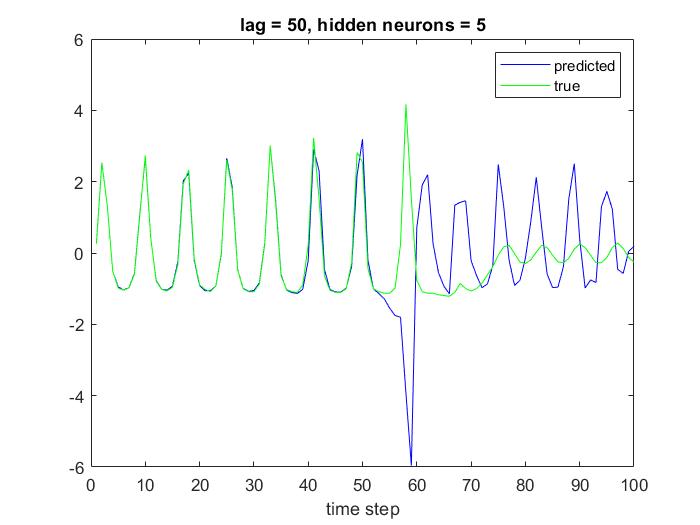


Figure 23

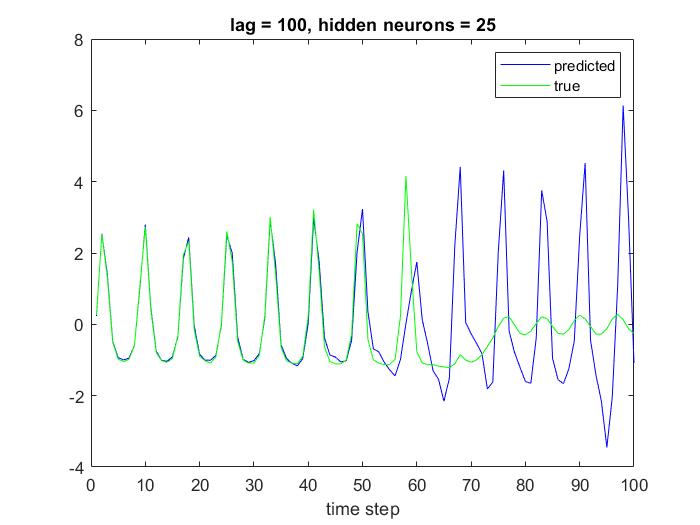


Figure 24

It appears that the first setting goes awry around datapoint 55. The second setting is maybe more correct but still far from good. We can conclude that both settings are not able to model the long-term dependencies, such as the sudden collapse of laser emission the pattern of the test set around datapoint 65, which is probably initiated already around time step 55. As an extra it should also be noted that the second setting takes 54 seconds of computation time because of the higher amount of hidden neurons, whereas the first setting only takes 1.3 seconds. Making the epoch higher (up to 500) is to no avail as for both settings the model is not able to predict the collapse of emission.

#### Long Short-Term Memory Network

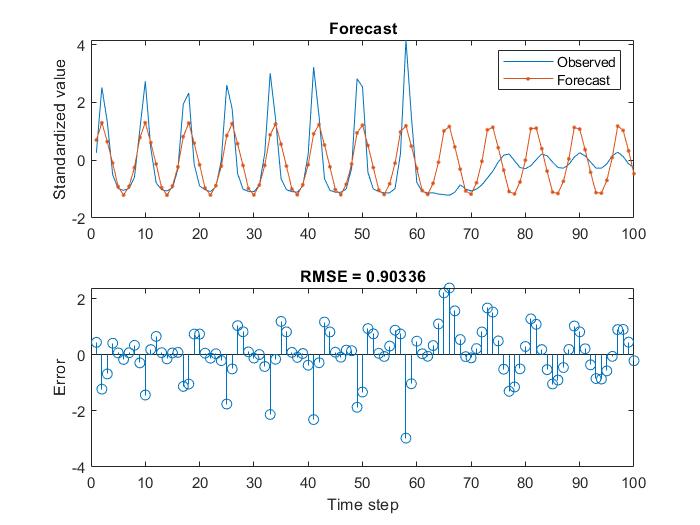
A long short-term memory network is a network consisting of a sequence layer and an LSTM layer. The purpose of the former is just to input the data as a sequence to the network and the purpose of the latter is to learn long-term dependencies. Simplified, the LSTM layer has memory (“gated cells”) and learns when to allow access of the contents of gated cells, reads, writes and erasures so that patterns that are spread out of a multitude of timesteps in the past can be learnt. The LSTM layer exists of an input gate, a forget gate, a cell candidate and an output gate. These four elements are a neural network on their own. Inside the LSTM layer, there is a flow of information apart from the input and output of the LSTM layer.

In order to learn these dependencies in the Santa Fe data set, we make use of the regression variant of the LSTMN. Here, the LSTM layer is followed by a fully connected layer and a regression output layer. The regression layer computes the half-mean-squared-error loss for regression problems, used to update the weights of the LSTM layer. For the Santa Fe data set, the number of features and number of responses needed to initialize the sequence input layer and the fully connected layer with, respectively, are both equal to one. Arbitrarily, the learning rate is set to 0.01 and chosen to drop to 40 percent of its value every 100 epochs. For the optimization function of the LSTM layer, the suggested function by Matlab, called ‘Adam’ is wielded ([Kingma, 2014](#Kingma2014)). The Adam optimization algorithm is an extension to stochastic gradient descent in which there are two important improvements;

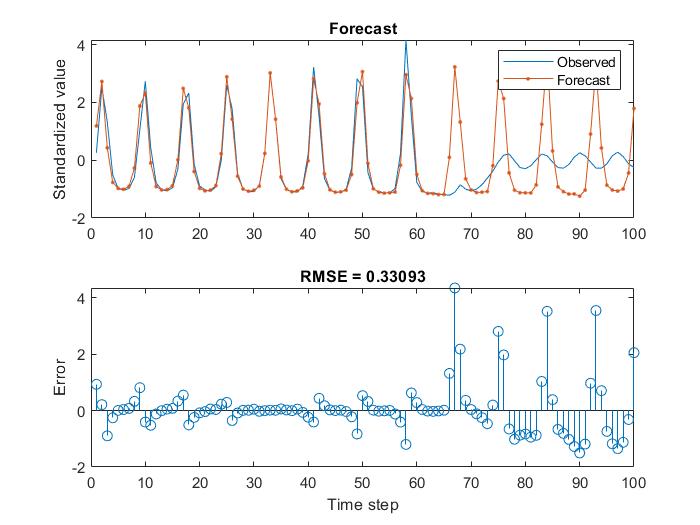
1. an Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse and
2. Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems.

The effect of two parameters are looked into, namely the number of epochs, and the number of hidden units in the neural nets in the LSTM layer. The number of hidden units corresponds to the amount of information remembered between time steps (the hidden state). The hidden state can contain information from all previous time steps, regardless of the sequence length. If the number of hidden units is too large, then the layer might overfit to the training data.

Similar as in section 2.i, and figure 22, a mesh graph was constructed with the 1/RMSE value of the standardized data. This was done for 1,2,5,10,25,50,75,100,125,150,175,200,250,300,350, and 400 hidden neurons, for different epochs: 50,100,150,200,250, and 300. The lowest RMSE measured for the first 50 datapoints, which accords with a RMSE of 0.3309. This was reached for the configuration of {epoch: 250, hidden neurons: 150}. As shown in figures 25 & 26, the model is able to follow the frequency quite nicely, though it seems that the frequency of the forecast is slightly higher, as there is a negative phaseshift in the first couple of iterations and around iteration 50, the phaseshift has become positive. The model is also doing its job qua amplitude, but after the 50th iteration, it seems to go awry as well. Just as like with the MLP, this configuration is not able to predict the collapse.



Figures 27 & 28



Figures 25 & 26

It must be noted that this is the best configuration for the first half of the test set, and therefor a test was conducted using the RMSE on the total test set. The single best configuration was {number of epochs: 300, hidden neurons: 2} with a RMSE value of 0.9034 of which the results are shown in figures 27 & 28. Because it’s clear that this configuration does not predict quite well enough, we conclude that an LSTM network in this framework of chosen (fixed) parameters is not able to predict the collapse.

When comparing LSTM and MLP in this setting, keeping in mind that we might have missed a better model by limiting the amount of hyper parameters to tune, they both seem to have two big resemblances. This being that they are able to predict both amplitude and frequency in an appropriate manner. Also, both approaches weren’t able to predict the collapse from time step 60 on. In the depicted figures, the MLP hinted that something would be off at that moment in time, while the best model generated from the tests with respect to LSTM did not. Also because the best configuration of the tests on MLP is ran in 1.3 seconds, and the best LSTMN is ran in 16.4 seconds, I would say that the MLP has resulted in the most competitive predictor of the two.

# Assignment 3

## Principal Component Analysis

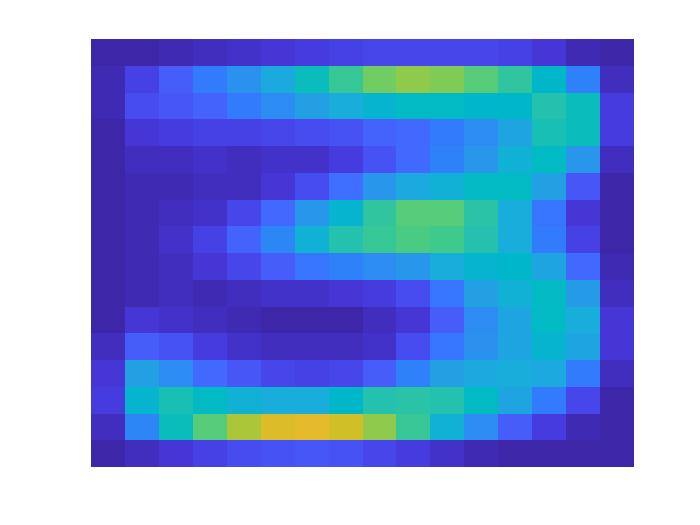


Figure 29

The data set of 26\*26 pixels, all representing threes, has a total of 256 dimensions. PCA can be used in this case to reduce dimensionality. Advantages are that the threes can be stored more compactly, and operations on the reduced data can be sped up. The mean (that is, taking the mean over all the data, per dimension/pixel) of the data set is shown in figure 29. After having created the covariance matrix, the eigenvectors and eigenvalues can be calculated using the eigenvalue decomposition algorithm. The eigenvectors represent the direction (based on the original 256 dimension) in which the data has the highest variances. These eigenvectors are all orthogonal to each other. Each eigenvector has an associated eigenvalue that represents how much of the total variance is explained by that eigenvector. Listing the eigenvalues in descending order lists indicates how much principal components can be used. The plot hereof is called a scree plot and is shown as a bar plot in figure 30. A rule of thumb to determine a fair amount of PCA’s is to find the elbow point in that point. Here, the elbow is more like a flexion, but five could be a good candidate. Let’s now have a look at the results when trying to reconstruct the digits by using up to the first four principal components, shown in figures 31 – 34. Reconstructing the original matrix by its approximation is done by multiplying the original matrix A by the matrix of eigenvectors and its transpose. The variance that accords with each of the figures is calculated as the accumulated eigenvalues for that amount of PCA’s divided by the total sum of eigenvalues (=total variance).

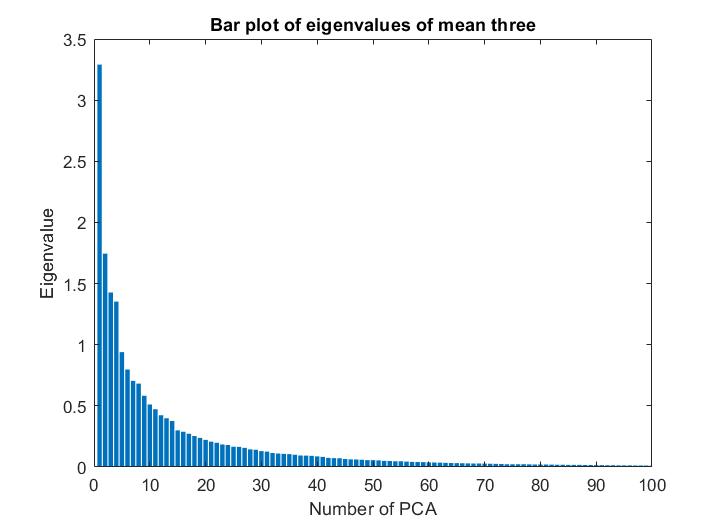
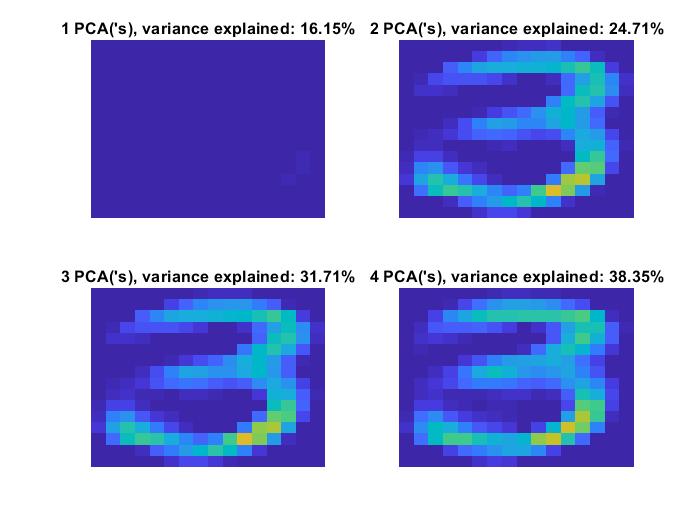


Figure 30



Figures 31, 32, 33 & 34

The first PCA alone does not graphically show the mean of threes, instead it shows a almost entirely blue screen, indicating that the pixel values are very low. For two, three and four PCA’s, the three is clearly visible and we can conclude that two PCA’s are enough to reconstruct the shape of the mean three graphically. The part of the variance that is not included (75.29%) will to great extent account for the variance between threes, that is not decisive for a three to be a three.

Each time a compression of the original matrix is executed, information can be/is lost. One way of measuring the differences in the original matrix and the reconstructed matrix is by calculating the mean RMSE (“MRMSE”) of every of 256 dimensions. The general trend is that the MRSME is lowering for increasing PCA’s, as shown in figure 35, which is logical because more variance of the original matrix is explained and the original matrix is thereby better approximated. Of course, we would expect the MRMSE to be zero entirely when reconstructing the original matrix using all variance (256 PCA’s). Instead we see that there is a small error still (in order of 10-16), because of rounding errors.

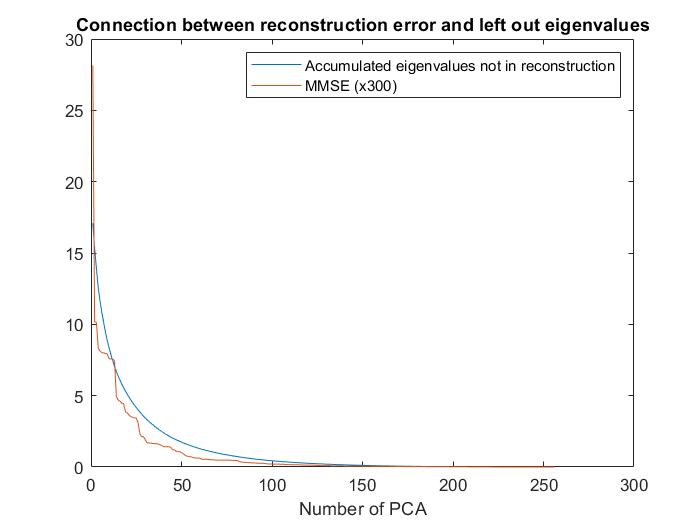


Figure 36

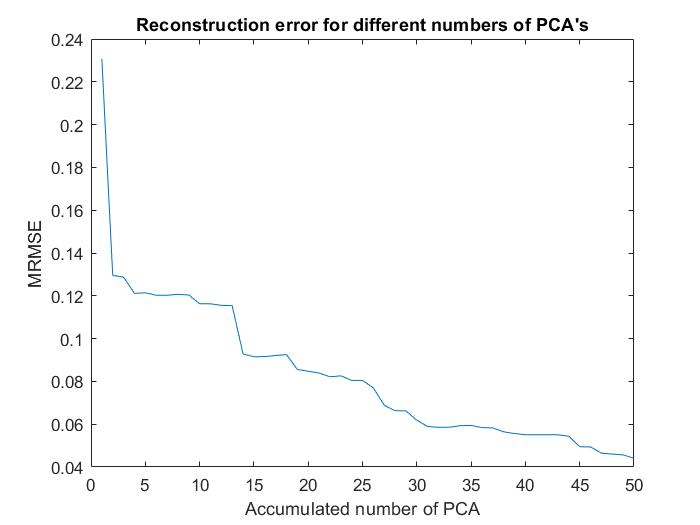


Figure 35

Lastly, when plotting the accumulated eigenvalues for a certain amount PCA that are not used for reconstruction we see that there is a proportional trend with the mean MSE of the reconstructed error. This is why if the eigenvalues decline sharply after projecting onto the first few, the errors will drop quickly as well.

## Stacked Autoencoders

A stacked autoencoder is, just like PCA, a way of reducing dimensionality. In a unsupervised setting, high dimensional data is fed to a encode layer which has less neurons than the number of dimensions of the input layer. This encode layer can be seen as the only hidden layer in this subnetwork. The output layer again has the number of dimensions equal to the input layer. By training this subnetwork in such a way that the output layer should be equal to the input layer, “encoding” happens from the input layer to the encoding layer. That is, the subnetwork learns to make the data more compact by representing the important information with less nodes than the input layer. We call the procedure of the data flow from the encoding layer to the output layer “decoding”. Once the encode layer of this subnetwork is trained, additional encode subnetworks can be constructed with the former encode layer as input, to reduce the dimensionality even further. Once all the encode layers are trained, a SoftMax layer is added to the end of the structure and in a supervised way trained with the corresponding labels of the training data. Once this procedure is finished, the deep network is finalized. Though, a final fine-tuning of the deep network as a whole can improve the accuracy ever further, as shown by the experiments carried out on the digittrain\_dataset in table 2. Here, each encoding layer is given by a number depicting the amount of neurons followed by a number between brackets, which represents the number of epochs. All settings for all layers are kept as they were given to us in DigitClassification.m. In the 3 encode layer setting, the second layer is a replica of the first.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 40(400) | 50(400) | 75(400) | 80(400) | 100(400)  50(100) | 150(400)  75(100) | 150(400)  80(200)  40(100) | 150(400)  100(200)  50(100) |
| Without  fine tuning | 90.46 % | 94.40 % | 96.68 % | 97.06 % | 83.70 % | 96.26 % | 23.38 % | 45.48 % |
| With  fine tuning | 99.38% | 99.16% | 97.08% | 97.76 % | 99.68 % | 99.42 % | 99.18 % | 99.28 % |

Table 2

From the results we can presume that often, the accuracies are higher for lower dimensional data that gets fed to the SoftMax layer. Also, fine tuning seems imperative in order to get to a higher accuracy, and the differences between finetuning and not finetuning are bigger for smaller (small amount of neurons in last layer) networks. Lastly, we can conclude that bigger networks do not necessarily imply better result. The best configuration of the ones mentioned above is the default one from the DigitClassification file, consisting of 100 neurons in the first and 50 neurons in the second encoding layer.

## Convolutional Neural Networks

#### Answers on questions

The script CNNex.m was ran to use AlexNet, a pre-trained CNN, that is able to classify images between three classes: laptop, ferry or airplane. Below an example image is given for each. The images have three dimensions in total with size 227, 227, and 3. The three indicates the three different colour dimensions: red, green and blue.



Figure 37

1. The second convolutional layer includes 96 convolution masks (also called receptive fields) that make it possible to extract basic image features such as blobs and edges. The dimensions of these masks are 11 by 11 by 3. Again, the three represents the RBG colours. The width and height of the masks that are covered and learnt are 11 by 11.
2. After the second layer, which is the first convolutional layer, a ReLu layer and Cross Channel Normalization layer are mounted. The former enhances non-linearity by only keeping track of the positive outputs of the convolutional layer. The latter was mounted there for normalization purposes but the role of such a layer has been outplayed by other regularization techniques (such as dropout and batch normalization) ([Stanfordcs321](#Stanfordcs321)). The layer responsible for downsampling, and thus reducing dimensionality, is the pooling layer. Because the stride (step by which the transparent mask is moved over the convolutional mask) is 2 by 2 and the padding is zero for every dimension, the 9 by 9 receptive fields are shrunk to 5x5 masks. The poolsize is 3 by 3 which means that a 3 by 3 transparent mask is hoovered over each convolution mask and the max of this projection is the replacement of that particular part of the convolutional mask of the new mask.
3. Because the CNN was pre-trained to divide images into 1000 different classes, the dimension of the final/classification layer consists of 1000 neurons. Since the original pixel dimensions were 227 by 227 by 3, which totals to 154587 features, using a total number of 1000 neurons in the end is still a significant dimension reduction. In this case of classification, only three neurons/classes would actually be used to classify images as either a laptop, ferry or airplane.

#### Digit classification

Just like the analysis of auto encoders, some tests were performed using different architectures for CNN’s to determine which architectures and meta parameters are favourable for this problem. First meta parameters were tuned for one convolutional-layered CNN’s. The tune parameters in question are filter size and number of filters. The computation times for every combination were around 30 seconds and quasi equal. Table 3 shows the results of accuracy.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | **Size** | | **Amount** |  | | 2 | 3 | 4 | 5 | 7 | 9 | 11 |
| 1 | 63.52% | 63.44% | 67.44% | 70.24% | 72.32% | 73.80% | 66.36% |
| 2 | 81.16% | 83.44% | 84.16% | 90.60% | 90.40% | 88.80% | 90.76% |
| 5 | 88.08% | 94.44% | 97.20% | 96.72% | 98.08% | 98.08% | 97.80% |
| 10 | 93.40% | 97.12% | 97.48% | 98.24% | 98.84% | 98.48% | 98.16% |
| 20 | 92.96% | 97.32% | 97.96% | 98.36% | 99.04% | 99.00% | 98.96% |
| 35 | 95.00% | 97.88% | 98.12% | 98.76% | 98.84% | 99.16% | 99.12% |
| 50 | 95.40% | 97.56% | 98.64% | 98.68% | 99.04% | 99.08% | 99.08% |
| 75 | 95.88% | 97.68% | 98.00% | 98.64% | 98.64% | 98.76% | 98.84% |
| 100 | 95.88% | 98.08% | 98.40% | 98.92% | 98.68% | 99.00% | 98.72% |

Table 3

The best accuracy is 99.16 % and obtained for 35 filters of size 9 by 9. For a two convolutional-layered CNN the same parameters were altered, and now having 2 times the amount of filters in the second layer. Again, computation times were around 30 seconds. The accuracies are shown in table 4.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | **Size** | | **Amount** |  | | 2 | 3 | 4 | 5 | 7 | 9 |
| 1 | 11.56% | 36.80% | 48.16% | 63.88% | 52.56% | 26.12% |
| 2 | 58.88% | 59.56% | 76.12% | 80.80% | 68.88% | 54.72% |
| 5 | 60.96% | 80.04% | 91.80% | 95.28% | 95.88% | 91.48% |
| 10 | 68.68% | 92.16% | 96.80% | 97.56% | 98.40% | 96.04% |
| 20 | 74.24% | 95.04% | 98.68% | 99.04% | 99.00% | 99.00% |
| 35 | 80.32% | 97.40% | 98.68% | 99.24% | 99.44% | 98.92% |
| 50 | 87.44% | 98.24% | 99.32% | 99.24% | 99.72% | 99.36% |
| 75 | 90.88% | 98.76% | 99.24% | 99.28% | 99.68% | 99.32% |
| 100 | 94.20% | 98.68% | 99.12% | 99.44% | 99.60% | 99.20% |

Table 4

For this architecture, choosing 50 filters of size 7 by 7 gives the most outstanding result of 99.72%, beating the best results of stacked autoencoders by 0.04%. As the accuracies of the proposed neural nets with one hidden layer of 100 hidden units (96.66 %) and two hidden layers of 100 and 50 hidden units (97.46 %) are inferior to the ones listed above from CNN’s and stacked autoencoders, we can conclude that both CNN and stacked autoencoders are matched to each other and are superior to traditional neural networks when it comes to high dimensional data, such as images. Again, a lot of parameters can be tweaked in these approaches, giving both flexibility but also complexity as it is not trivial to find the best combination of those.

# Assignment 4

## Principal Component Analysis

# Bibliography

Minsky, Marvin, and Seymour A. Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

KINGMA, Diederik P.; BA, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

<http://cs231n.github.io/convolutional-networks/>

1. By means of reproducibility, the same random generators are made use of, unless stated otherwise.

   rng(1); % random generator for initializing nets

   s = RandStream('mt19937ar','Seed',1); % random single generator

   Also, for every stated result of an experiment, the average of 10 rounds’ results is taken [↑](#footnote-ref-1)