Faculty of Science
Master of Statistics and Data Science
Academic year: 2020 - 2021

Data Mining and Neural Networks

Report

by

Khachatur Papikyan r0825613

Assignment 1: Training Algorithms and Generalization **3**
1.1 The perceptron and beyond *3*
1.1.1 How to perform a linear regression with a perceptron? Description of the link between those two models *3*
1.1.2 Plot the function. Will a linear model adequately capture the relationship between the input and output data? Is this related to over- or underfitting? *3*
1.2 Backpropagation in feedforward multi-layer networks *4*
1.2.1 Backpropagation *4*
1.2.2 Function approximation: comparison of various algorithms *5*
1.3 Personal regression example *6*
1.3.1 Defining own datasets *6*
1.3.2 Building and training own feedforward neural network *6*
1.3.3 Performance assessment *7*
1.4 Bayesian inference *8*
1.4.1 Comparison of algorithms *8*
1.4.2 Discussion of regularization effects *8*
Assignment 2: Time-series Prediction and Classification **9**
2.1 Time-series prediction *9*
2.1.a.1 Investigating the model performance with different lags and number of neurons *9*
2.1.a.2 Would it be sensible to use the performance of this recurrent prediction on the validation set to optimize hyperparameters? *10*
2.1.b.1 Building networks for city temperature time-series predictions *10*
2.1.b.2 Performance assessment *11*
2.1.b.3 Conclusion *11*
2.2 Classification *12*
2.2.1 Breast cancer data visualization *12*
2.2.2 Binary classification for breast cancer *12*
2.3 Automatic relevance determination *13*
Assignment 3: Unsupervised Learning and Data Visualization **15**
3.1 Self-organizing map *15*
3.1.1 SOM on banana dataset *15*
3.1.2 SOM on covertype dataset *16*
3.2 Principal component analysis *17*
3.2.1 PCA on uncorrelated data *17*
3.2.2 PCA on correlated data *17*
3.2.3 PCA experiment with functions mapstd(x)and processpca(x,maxfrac) *17*
3.3 Autoencoder *18*
3.4 Stacked autoencoder *19*
Assignment 4: Variational Auto-Encoders and Convolutional Neural Networks **21**
4.1 Weight initialization and batch normalization *21*

4.1.1 Scale of weight initialization 21
4.1.2 Relationship between batch normalization and batch size 22
4.2 Variational autoencoders 23
4.2.1 Similarities and differences between stacked autoencoders and variational autoencoders 23
4.2.2 Optimizers: pros and cons 23
4.3 Convolutional neural networks 24
4.3.1 Toy example 24
4.3.2 CNNex.m 24
4.3.3 CNNDigits.m 25
References **27**

Assignment 1: Training Algorithms and Generalization

1.1 The perceptron and beyond

1.1.1 How to perform a linear regression with a perceptron? Description of the link between those two models

The link between linear regression and perceptron is apparent in their corresponding equations' structures. In essence, these can be regarded as the same kind of equations, where the target (or output) variable is being represented as a sum of a certain constant and a weighted sum of other variables. In other words, both sum the product of input and parameterized variables, as well as a bias term, to best estimate an outcome variable.

1.1.2 Plot the function. Will a linear model adequately capture the relationship between the input and output data? Is this related to over- or underfitting?

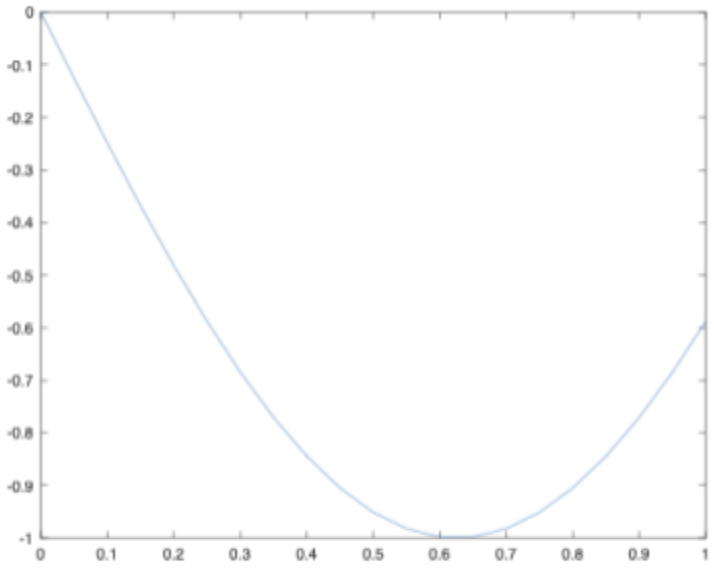


Figure 1: Plotted function

A linear model does not seem to be the most optimal choice in this particular case, because from the plot in Figure 1 it becomes evident that the relationship between the input and output data is clearly non-linear. Consequently, a linear model would probably suffer from underfitting, which is the case when the model is having trouble in generalizing well on the new, unseen data because of not being able to capture the relationships between the input and output data of the training set sufficiently well. Overfitting would be the opposite case, when the model fits the training data "too" perfectly (i.e. memorizes the training data) and because of that fails to generalize well on the new, unseen data.

1.2 Backpropagation in feedforward multi-layer networks

1.2.1 Backpropagation

	$weight_{old}$	$weight_{new}$
w_1	3	3.02156
w_2	4	4.00762
w_3	5	5
w_4	6	6
b_1	1	1.01078
b_2	2	2.00762
w_5	8	8.010244
w_6	9	9.0106022
b_3	7	7.02507

Table 1: Initial and updated network weights

The process of updating the network consists of two steps: forward pass and backward pass. In the forward pass the input data is passed through the network with predefined weights, and based on this, predictions are made. Then the error of predictions with regards to the target values (in this case mean squared error) is computed, which is the loss function that needs to be optimized in order to improve the predictive power of the network. This is done by computing the partial derivatives of the error function with respect to the corresponding network weights during the backward pass. Once these derivatives are computed, the network weights can be updated with the following formula: $weight_{new} = weight_{old} - \eta * \frac{\partial Error}{\partial weight_{old}}$, where $\eta = 0.01$ is the user-defined learning rate.

Table 1 demonstrates the updated network weights after one step of simple gradient descent, as well as the initial network weights.

The complete manual process of updating the network after one step of simple gradient descent can be consulted in the scratch paper photos from Table 2. The higher resolution pictures can be found here.

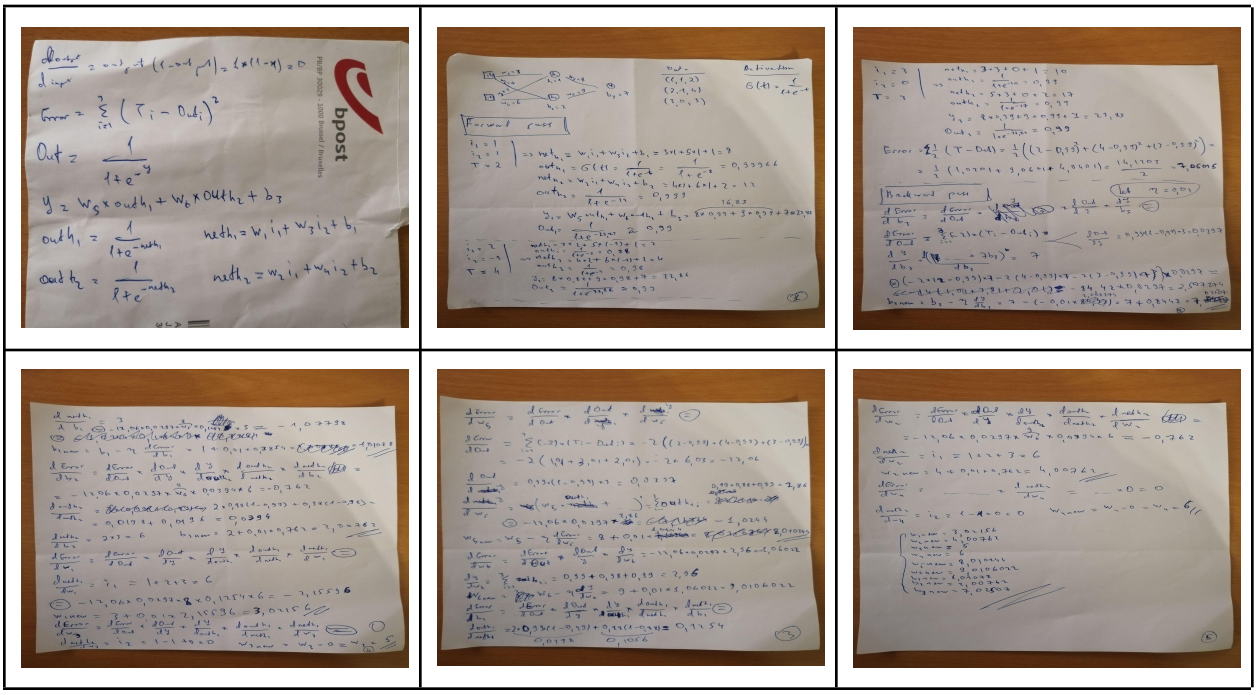


Table 2: The complete analytical process of updating the network weights

1.2.2 Function approximation: comparison of various algorithms

Various configurations of gradient descent (i.e. *traingd*), gradient descent with adaptive learning rate (i.e. *traingda*) and Levenberg-Marquardt algorithm (i.e. *trainlm*) are chosen for comparison, the results of which are displayed in Table 3. In order to get comparable results in terms of dataset size, epochs, noise and timing, the same neural network architecture consisting of a single hidden layer with 50 neurons is used for every algorithm.

Datapoints Epochs Noise	50 100 0	50 100 0.5	50 250 0	50 250 0.5	1000 100 0	1000 100 0.5	1000 250 0	1000 250 0.5
traingd	1.4967s	1.3755s	1.6541s	1.4781s	1.4306s	1.4982s	1.5616s	1.6876s
traingda	0.34349s	0.32261s	0.41043s	0.46163s	0.37089s	0.38686s	0.51401s	0.52961s
trainlm	0.32978s	0.67766s	0.29726s	1.1491s	1.2007s	1.1807s	2.2354s	2.2159s

Table 3: Comparison of various algorithms and configurations by timing

From the results of Table 3, *traingda* seems to have the best performance in terms of speed. For smaller datasets and smaller number of epochs *trainlm* consistently shows the second best speed results, but with the increase of the number of epochs and dataset size the time needed for convergence or reaching maximum number of epochs significantly increases, thus leaving *trainlm* behind *traingd*. The results from Table 3 also suggest that adding noise to the dataset tends to add additional difficulties for the algorithms, which slows down their executions and may lead to a worse fit.

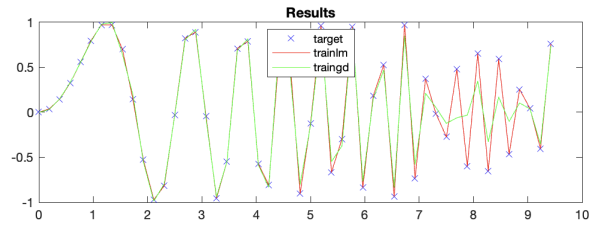


Figure 2: Comparison of trainlm and traingd

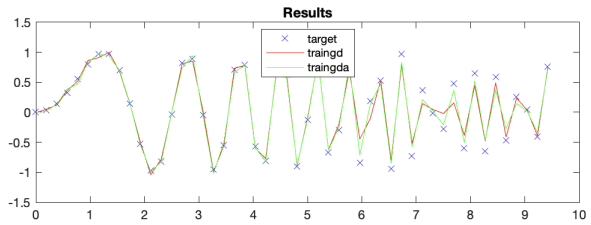


Figure 3: Comparison of traingd and traingda

	trainlm	traingd	traingda
Datapoints	50	50	50
Epochs	250	10000	2000
Noise	0	0	0
Time	0.93228s	6.2448s	1.398s
R	1	0.96799	0.99304

Table 4: Comparison of traingd with trainlm and traingda

Table 4 further strengthens the findings in Table 3. According to the results from Table 4, for the same dataset and with the same neural network architecture *trainlm* leads to a better fit (i.e. the highest R value, which is comparing Target vs Output and the closer it is to 1, the better) and takes much less time to finish the execution. To get somewhat closer to *trainlm*'s fit, *traingd* required more than 10000 epochs and *traingda* more than 2000, making these algorithms more expensive in terms of timing as well. So despite the fact that *trainlm* may initially seem less time efficient, it takes a leading position in the combination of the delivered accuracy and time.

1.3 Personal regression example

1.3.1 Defining own datasets

As my student number is r0825613 and the list of top 5 largest digits in descending order consists of 8, 6, 5, 3 and 2, each new point of T_{new} is constructed as $T_{new} = \frac{8*T_1+6*T_2+5*T_3+3*T_4+2*T_5}{8+6+5+3+2}$. Then $X1$, $X2$ and T_{new} are concatenated into a single $13600\ x\ 3$matrix of doubles. From this matrix 3000 samples are randomly selected without replacement, of which 1000 samples for training set, 1000 samples for validation set and another 1000 samples for test set. In particular, 1000 samples are randomly selected without replacement from the 13600 x 3 matrix as a training set, then the matrix is updated and the 1000 samples that are already selected are removed. Another 1000 samples are randomly sampled without replacement from the newly updated 12600 x 3 matrix as a validation set and the matrix is updated once more by removing the samples that are already selected for the validation set. Finally, the third random sampling without replacement is done from the lately updated 11600 x 3 matrix in order to get 1000 random samples for the test set. This process is done to ensure that all the three sets are independent from each other and do not overlap, to make sure that no datapoint has 0 probability of being selected and also to account for maximum variability possible.

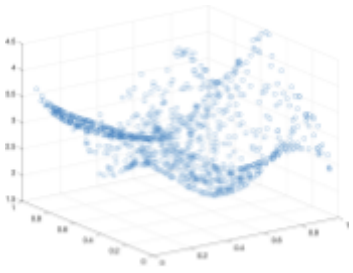


Figure 4: Training Surface

1.3.2 Building and training own feedforward neural network

The neural network chosen for solving this problem consists of a single hidden layer of 20 neurons. Levenberg-Marquardt algorithm is chosen as the learning algorithm, and as transfer functions the defaults are taken, namely $tansig$ for the hidden layer and $purelin$for the output layer. Evidently, these are the final settings, but before getting here different combinations of the network configurations were tried in order to figure out the best one among them. In particular, varying numbers of neurons were tried for the hidden layer, 3 different learning algorithms were tried (i.e. gradient descent, gradient descent with adaptive learning rate, Levenberg-Marquardt algorithm) and different transfer functions apart from the defaults (e.g. $logsig$for the hidden layer, etc.). The models were validated based on their performances on the validation set, and after that the final model was selected. The $Output\ vs\ Target$ regression plots for the validation set for the neural networks with a single layer of 20 neurons, where the above-mentioned 3 algorithms are the learning algorithms and where the default transfer functions are the transfer functions are demonstrated in Table 5. As it can be seen, $traingd$ and $traingda$ seem to be overfitting because for both of them the performance on the validation set has decreased by 5 percentage points compared to the one on the training set. $trainlm$does not seem to have this problem, which is an important criterion for generalization.

	$traingd$	$traingda$	$trainlm$

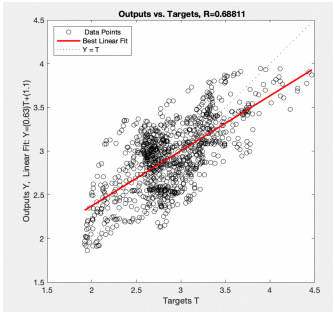
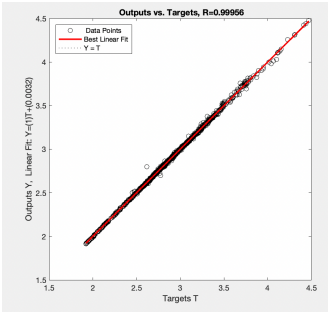
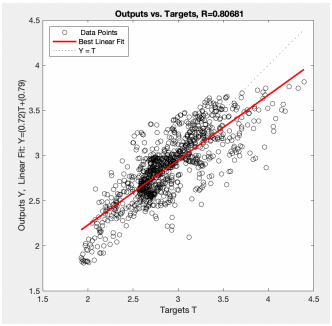
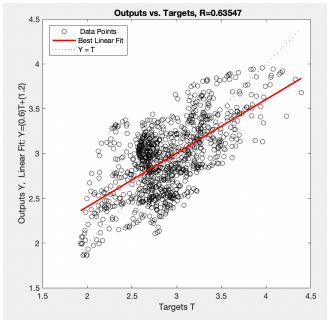
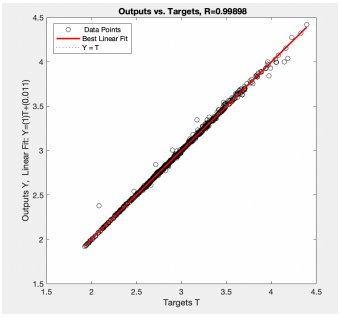
Training set			
Validation set			

Table 5: Comparison of the algorithms in a neural network of a single hidden layer with 20 neurons and default transfer functions for the training and validation sets

1.3.3 Performance assessment

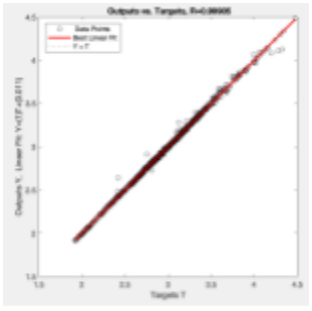


Figure 5: Regression plot

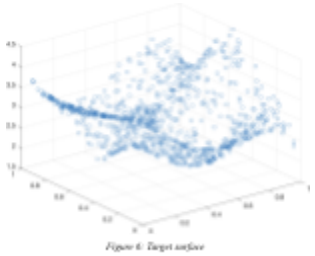


Figure 6: Target surface

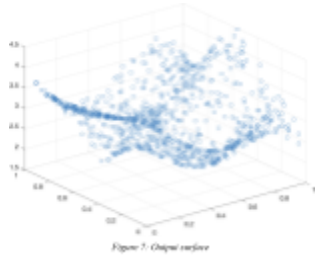


Figure 7: Output surface

The final assessment of the chosen model's performance is done on the test set. As it can be seen from Figure 5 it gives almost identical results as on the training and validation sets. Furthermore, Figures 6 and 7 confirm the adequacy between the target and model's output surfaces. The loss curves in Figure 8 shows that the best validation performance was achieved after 147 epochs, although 1000 epochs were allocated by default for training the network. After the 147-th epoch the validation set performance was starting to decrease, hence why the training was stopped. Obviously, the performance on the training set would continue to improve even after the 147-th epoch as seen in Figure 8, but that is exactly when the model would start overfitting, and as the model's generalization performance is of greater importance, the training process of the neural network was stopped there. The same thing can be observed for the test set as well. The final *RMSE* value on the test set is 0.0032, which also on its turn signifies a good model fit. Overall, the chosen network is performing highly accurately and without overfitting. The network could potentially be further optimized by playing with even more learning algorithms, changing the learning rates, researching for more transfer functions, changing the stopping criteria for training or taking another loss function, etc.

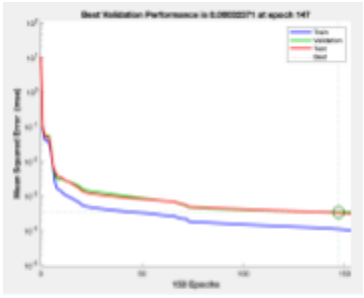


Figure 8: Loss curves

1.4 Bayesian inference

1.4.1 Comparison of algorithms

Noise = 0	Noise = 0.5

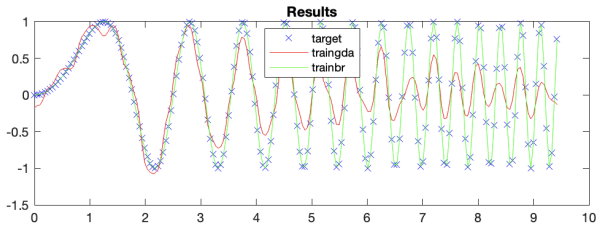
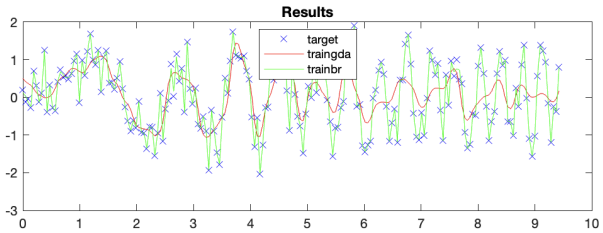

	

Figure 9: target, traingda and trainbr	*Figure 10: target, traingda and trainbr*

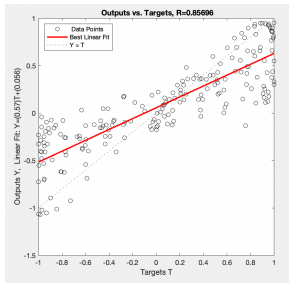
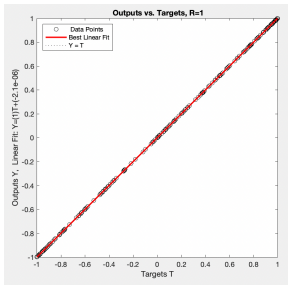

	

Figure 11: Regression plot for traingda	*Figure 12: Regression plot for trainbr*

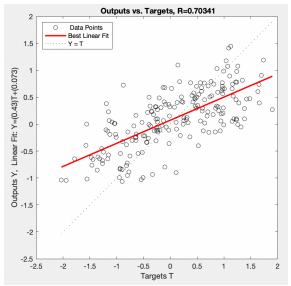
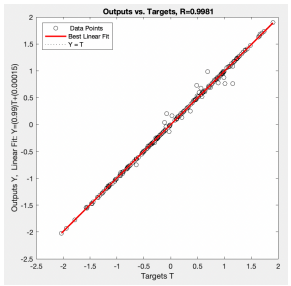

	

Figure 13: Regression plot for traingda	*Figure 14: Regression plot for trainbr*

Table 6: Comparison of traingda and trainbr with and without noise

For this part, an overparameterized neural network was used. In particular, it consisted of 2 hidden layers with 50 and 20 neurons respectively on its layers. 200 datapoints were considered with and without noise added, and for the training of each of the algorithms 250 epochs were allocated. The results from Table 6 show that in both of the cases (i.e. with and without noise) $trainbr$has been highly accurate and has also significantly outperformed $traingda$. Of a particular interest are Figures 11 and 13, where it is shown that the performance of $traingda$ falls dramatically with added noise (15 percentage points). Obviously, the performance of $traingda$ would improve with more epochs being added, because the execution of algorithm was stopped because of reaching maximum epochs, but considering the current trend, most probably it would simply lead to memorization of data and would not improve the network's generalization performance.

1.4.2 Discussion of regularization effects

In the context of the above-discussed 2 algorithms, the importance of regularization becomes more evident. In case of $traingda$regularization is not envisaged by default, whereas in $trainbr$ there is indeed a regularization term, which punishes the network for memorizing the data and thus makes it more generalizable, which is demonstrated in comparisons from Table 5. Whether there is noise added or not makes almost no difference for $trainbr$, whereas significantly diminishes the performance of $traingda$

Assignment 2: Time-series Prediction and Classification

2.1 Time-series prediction

2.1.a.1 Investigating the model performance with different lags and number of neurons

Before starting building models and investigating their performances, the network inputs and the test data were preprocessed through standardization because neural networks may be very sensitive in this regard. In essence, the datasets were rescaled to have $mean = 0$and $standard\ deviation = 1$.

Once the datasets were appropriately preprocessed, different combinations of lags and number of neurons were tried in order to come up with an optimal configuration of those hyperparameters. Lags in the range of 30:150 were tried with 10 point incremental steps. For the number of neurons in the hidden layer 30, 50 and 100 neurons were tested. The winning combination is the one with the lowest MSE value and the selected final network consists of a single hidden layer with 50 neurons and 100 point lags. The given learning algorithm is the Levenberg-Marquardt algorithm. Apart from these, the final model was also 10-fold cross validated in order to ensure that the network's performance does not deviate significantly from fold to fold. The average MSE value over those 10 folds is 0.9249 but this result has to be treated carefully, because it might be a result of arbitrary values of MSE over the 10 folds. One of the prediction results of these 10 folds is demonstrated in Figure 15.

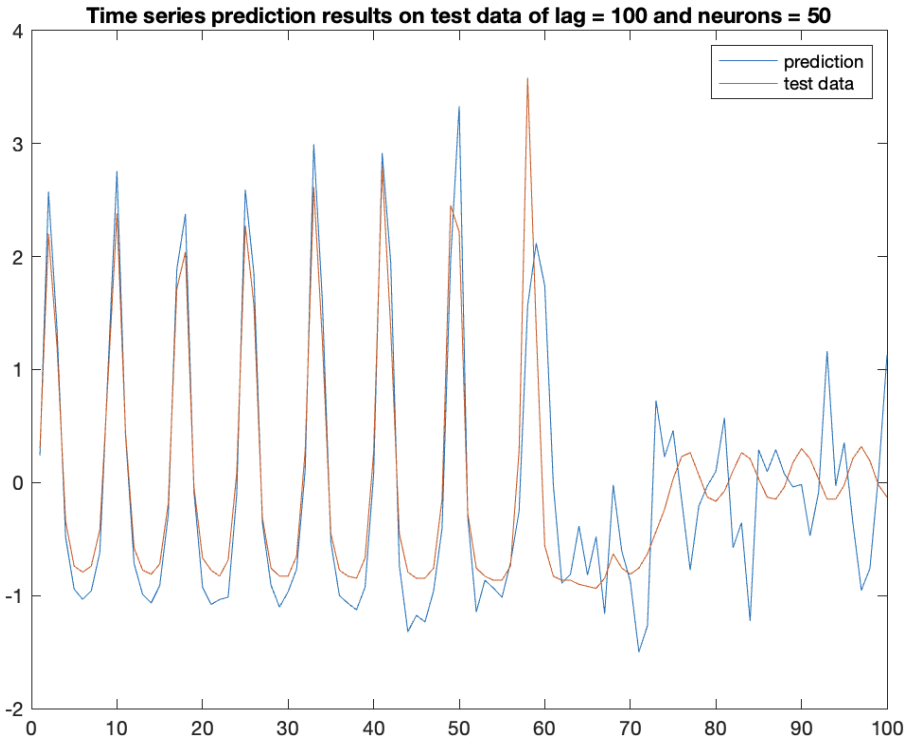


Figure 15: Time series prediction with lag=100 and neurons=50

2.1.a.2 Would it be sensible to use the performance of this recurrent prediction on the validation set to optimize hyperparameters?

Given the current set of instructions, it might be less sensible using the performance of this recurrent prediction on the validation set to optimize hyperparameters. In particular, the problem might lay in the further subdivision of training set into training and validation sets meaning that the first 80% of the initial training data was allocated for training and the last 20% for validation. In this regard, it could lead to the situation demonstrated in Figure 15 where after a certain point (i.e. 60-th value) the actual behavior of the time series changes dramatically and the network faces difficulties with generalizing well in this range of the series, because in the historical data that it was trained on it did not encounter this kind of pattern. As a result, if there were another subdivision of training, validation and test sets, the above-cited approach could make more sense. Also, it is important to mention that by the given instructions, only 50 epochs were allocated for training the neural network, which might not necessarily be the most optimal choice because quite often the training used to stop because of reaching the maximum epochs, whereas the gradient was still continuing to decrease. Obviously, the overall performance of the model is not very satisfying and more simulations along with modifications in the given set of instructions from the problem statement could potentially leave less room for improvement.

2.1.b.1 Building networks for city temperature time-series predictions

Training	Test

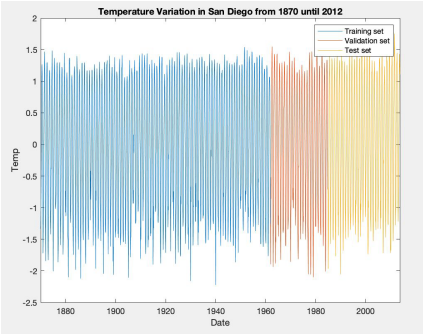
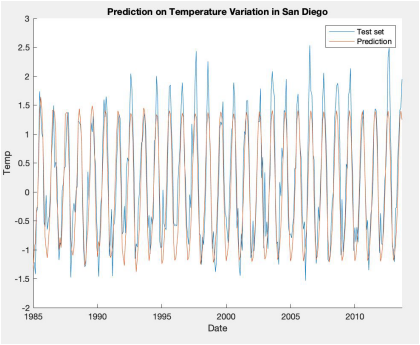

	

Figure 16: Temperature variation in San Diego	*Figure 17: Prediction on temperature variation in San Diego*

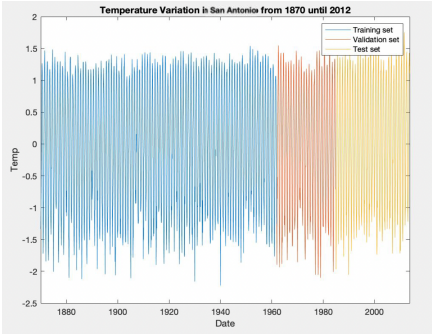
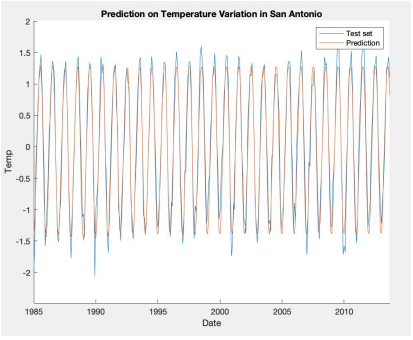

	

Figure 18: Temperature variation in San Antonio	*Figure 19: Prediction on temperature variation in San Antonio*

Table 7: Temperature variations in San Diego and San Antonio

Two US cities, namely San Diego and San Antonio, were chosen as the cities whose historical meteorological data would be used to build 2 neural networks for time series predictions. Analogously to section 2.1.a, here as well both the training and test sets were first standardized before passing any input into the neural networks. In the view of huge dataset size and heavy computational requirements, within reasonable time limits different configurations were tried in terms of learning functions, lags, number of hidden layers, number of neurons in those hidden layers, number of epochs, etc. In particular, gradient descent with adaptive learning rate was chosen as the learning function for both of the networks considering its relatively fast execution and satisfying generalization performance.

For San Diego, the selected final model was configured to have a single hidden layer with 10 neurons and a lag of 140. 400 epochs were allocated for the training of this network. The training and test sets, as well as the predictions on the test set are demonstrated in Figures 16 and 17.

In the case of San Antonio, the selected final model also consisted of a single hidden layer with 10 neurons, but here 100 lags were taken instead of 140 of the previous model. In order to train this network, 1000 epochs were allocated. The training and test sets, as well as the predictions on the test set are demonstrated in Figures 16 and 17.

The performances of both of these networks were 10-fold cross validated.

2.1.b.2 Performance assessment

The prediction results of both of these models can be consulted in Figures 17 and 19. The average MSE for San Diego's model over 10 iterations on the test set is 0.22 and for San Antonio's model over the same number of iterations on the test set is 0.037.

The performances of both of the models could potentially further be improved through inclusion of more hidden layers and/or playing with the number of neurons in one layer more subtly, testing more learning algorithms, playing more with the maximum number of epochs, modifying the cross-validation folds for trying to come up with a more robust network architecture, etc. Apart from that, San Diego's temperature variations seem to be more variable than San Antonio's ones, so a model that utilizes a different approach to weather forecasting might potentially be more appropriate. For example, a model providing better results for San Diego might have less lags in order to sample more recent data.

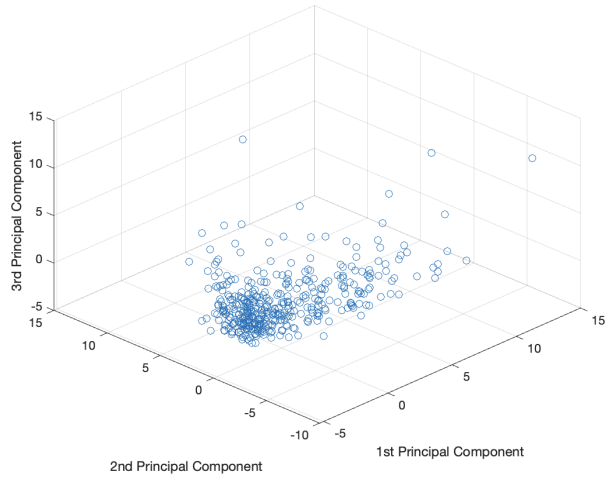
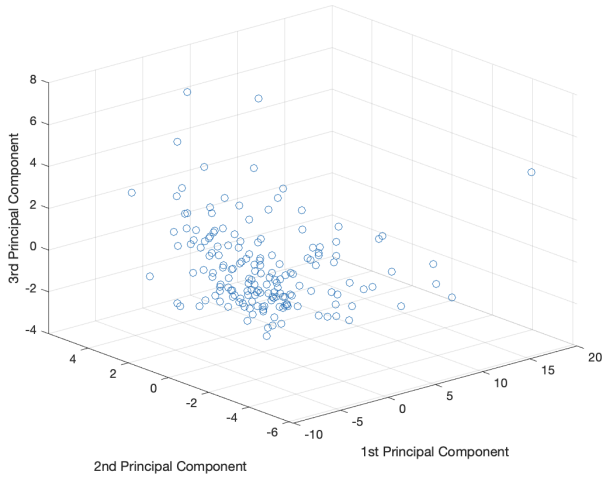
2.1.b.3 Conclusion

The general conclusion is that the recent temperature variation between the two cities is quite different. Generally speaking, the temperature in San Antonio seems to be more variable than in San Diego on a yearly basis, and the expected changes in weather seem to be closer to the recent variations. The mild climate in San Diego, in its turn, shows how much more extreme these recent changes are compared to the past (e.g. reaching changes in temperature as high as almost 3 standard deviations away from the mean).

2.2 Classification

2.2.1 Breast cancer data visualization

As the breast cancer dataset is a high-dimensional one, a natural need of dimensionality reduction arises. One such technique is PCA and in order to run PCA on these datasets (i.e. training and test sets), first both of the sets were standardized as required by the PCA technique. Figures 20 and 21 visualize these datasets in 3 dimensions according to their respective component loadings. In addition to this, Table 8 also contains information about these component loadings as well as the total variability explained by these 3 principal components for both of the datasets.

Training set	Test set
 *Figure 20: Top 3 principal components*	 *Figure 21: Top 3 principal components*

Variability explained by the principal components

First principal component	44.73%
Second principal component	19.21%
Third principal component	9.49%
Total	73.43%

First principal component	43.77%
Second principal component	18.34%
Third principal component	9.99%
Total	72.1%

Table 8: Breast cancer data

2.2.2 Binary classification for breast cancer

For this binary classification task the chosen final network consists of 4 hidden layers with 50, 20, 40 and 10 neurons respectively on each hidden layer. As this was a classification task, instead of taking MSE as an error function, binary cross entropy was used as a performance metric. The transfer function on hidden layers was chosen to be $tansig$and on the final layer it was $logsig$,

because before starting the model building process, the labels of both training and test sets were transformed to be 1/0 instead of being -1/1 as in the initial dataset. This transformation was done in order to make it easier to interpret the output values in the range of 0:1 as probabilities. In particular, 0 would mean absence of breast cancer and 1 would signify its presence. Both the training and test sets were standardized. Scaled conjugate gradient (i.e. *trainscg*) was chosen to be the learning algorithm of the current network. By default, 1000 epochs were allocated for the training of this network, but the execution was stopped much earlier (i.e. after the 17-th iteration) as the validation performance had reached its minimum. Apart from this, the network was 10-fold cross validated in order to ensure a stable generalization performance before finally testing it on the holdout test set of 169 observations. There as well, the model performed highly accurately by demonstrating 98.81% hit rate (i.e. accuracy) and misclassifying only 2 out of 169 observations of the test set. Figure 22 shows the ROC curve on the test set.

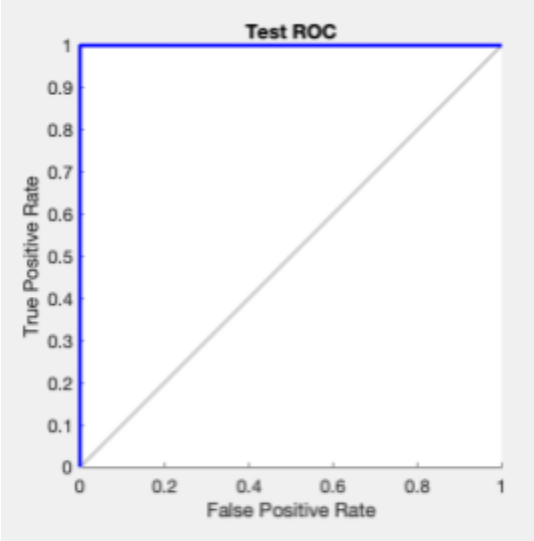


Figure 22: Test ROC

2.3 Automatic relevance determination

In order to determine the most relevant variables in terms of correctly classifying the breast cancer from section 2.2.2, automatic relevance determination technique using a multi-layer perceptron was used from *demard.m* netlab script. For a particular variable to be selected, the weights associated to that input after 2 cycles of re-estimations in absolute terms should be greater than 0.001 (i.e. $abs(weight) > 0.001$). After applying this threshold, 10 variables out of 30 meeting the selection criterion were selected, namely the variables number 2, 3, 4, 14, 17, 20, 22, 23, 24 and 28. This process applied both to training and test sets. Once the variable selection was finalized, both the training and test sets went through the same preprocessing steps as described in section 2.2.2. The chosen network architecture, learning algorithm and transfer functions, cross validation folds and everything else was implemented analogously. The only difference was that this time the network should take 10 inputs instead of 30 as in the previous section. As shown in the confusion matrix for the test set in Figure 23, the overall hit rate of the current model is 97%, meaning that only 5 out of 169 observations are misclassified by the model. The test set ROC curve from Figure 24 also states that the network performs highly accurately. Overall, compared to the previous model from section 2.2.2 where all the 30 variables were being given as inputs to the network, the current model with 10 inputs and everything else being equal has performed only slightly worse (hit rates 98.81% vs 97%), which means that the 10 most relevant variables detected by ARD have indeed been very informative.

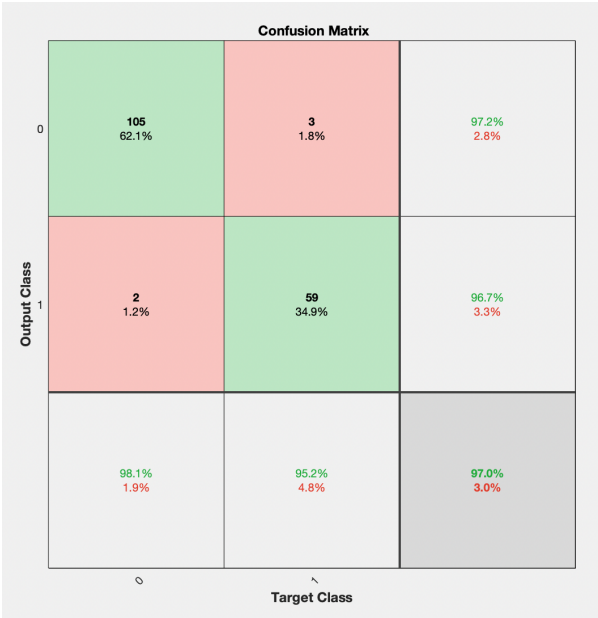


Figure 23: Test confusion matrix

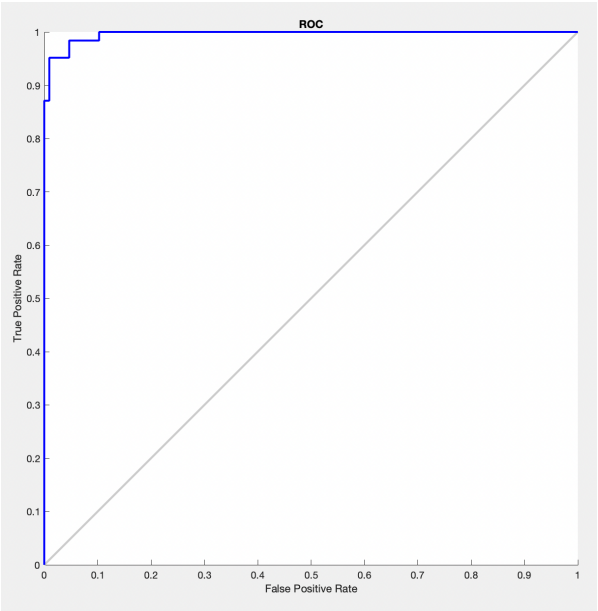
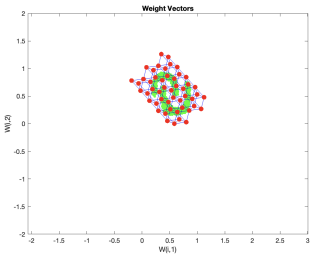
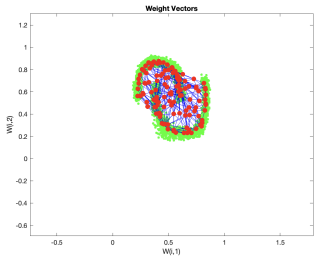
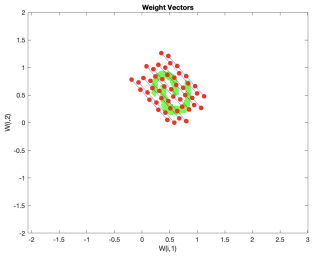
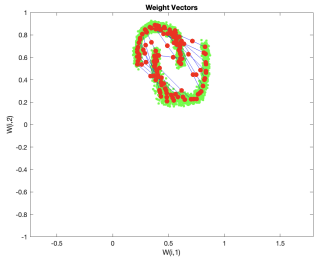
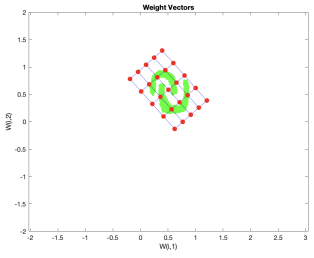
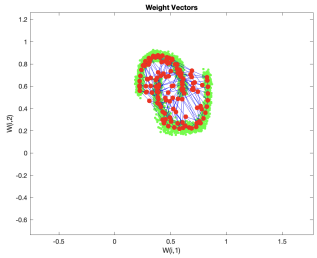
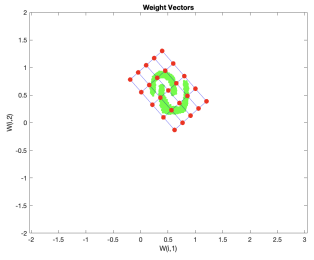
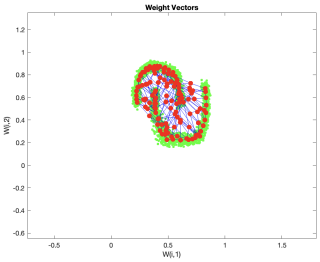


Figure 24: Test ROC

Assignment 3: Unsupervised Learning and Data Visualization

3.1 Self-organizing map

3.1.1 SOM on banana dataset

	Before training	After training
Hextop, Linkdist		
Hextop, Mandist		
Gridtop, Linkdist		
Gridtop, Mandist		

A 5-by-5 grid of neurons was used to generate SOMs for the banana dataset. Two topology functions were explored, namely *hextop* and *gridtop*, in combination with link and Manhattan distances. Table 9 demonstrates these combinations before and after training within 100 epochs.

Table 9: SOMs for banana dataset

3.1.2 SOM on covertype dataset

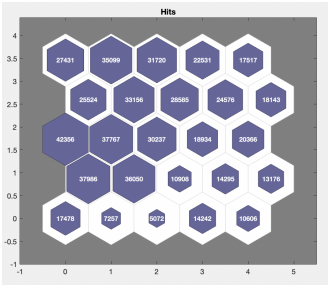
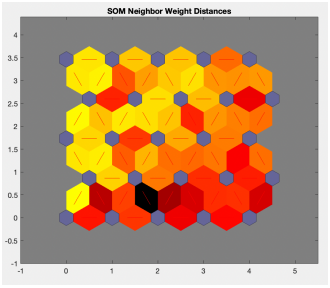
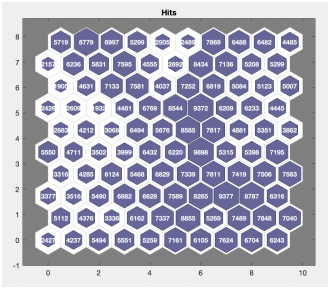
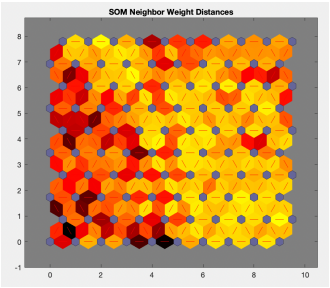
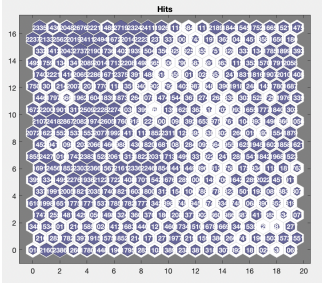
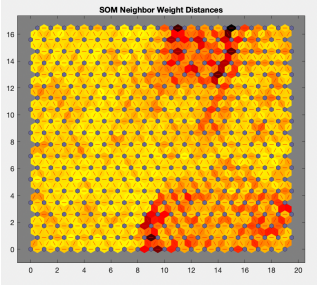
	SOM Sample Hits	SOM Neighbor Distances
Grid: 5x5 Epochs: 200 Topology: hextop Distance: linkdist ARI: 0.0082 RI: 0.6145		
Grid: 10x10 Epochs: 200 Topology: hextop Distance: linkdist ARI: 0.0059 RI: 0.6214		
Grid: 20x20 Epochs: 200 Topology: hextop Distance: linkdist ARI: 0.0023 RI: 0.6233		

Table 10: SOM on covertype dataset

Several combinations of grid-size and topologies were explored. The Rand Index (RI) and Adjusted Rand Index (ARI) were checked for each combination to assess the performance of the SOM produced such that the closer the value is to 1, the more similar the clustering results are to the clustering of the true class labels. However, since the dataset itself is unbalanced, the ARI is expected to be driven close to zero, thereby rendering it unsuitable in providing a conclusive assessment of the performance of SOM for this dataset. Consequently, this fact has to be carefully considered in the assessment of the SOM's performance.

Results from Table 10 suggest that increasing the grid-size further may potentially improve the Rand Index, meaning that the SOM-generated clustering results might improve in agreement with the true class labels as the grid size increases. Concerning the SOM neighbor distances, clustering can be done by considering the partitions created by the dark-colored areas, since darker colors mean larger distances between the neurons and lighter colors indicate closer distances.

3.2 Principal component analysis

3.2.1 PCA on uncorrelated data

Number of principal components taken	RMSE
11	0.8165
22	0.6418
44	0.2365
49	2.2410e-15

Table 11: RMSE values between reconstructed and original data for different dimensions

3.2.2 PCA on correlated data

Number of principal components taken	RMSE
1	0.0201
2	0.0045
3	0.0031
4	0.0022

Table 12: RMSE values between reconstructed and original data for different dimensions

As demonstrated in Tables 11 and 12, compared to randomly generated uncorrelated data, PCA is much more efficient in dimensionality reduction for highly correlated data like $choles\_all$. This is because in highly correlated data, the correlated variables explain the same variability in data, which gives a hint that instead of including every single variable, the overall variability might be explained by a very few principal components. In this particular case of the component p (a 21×264 matrix) from $choles_all$dataset, even the first principal component alone already explains 96.52% of the variability in data, meaning that even if dimensionality reduction is implemented, there will be a very insignificant information loss. On the contrary, reducing the dimensionality for randomly generated uncorrelated data has resulted in substantial information loss.

3.2.3 PCA experiment with functions $mapstd(x)$and $processpca(x, maxfrac)$

Similar trends and results as in sections 3.2.1 and 3.2.2 are explored here. By playing with several values (e.g. 0.001, 0.005, 0.01, 0.03, 0.04, 0.05) of $maxfrac$parameter, which is the maximum fraction of variance for removed rows (default is 0), and computing the $RMSE$values, it proves the findings from the previous 2 sections stating that PCA performs very well on highly correlated data and poorly on an uncorrelated one. Moreover, the trend of improving the $RMSE$ values by taking more principal components (i.e. getting closer to 0 in terms of $maxfrac$ parameter) is followed here as well.

3.3 Autoencoder

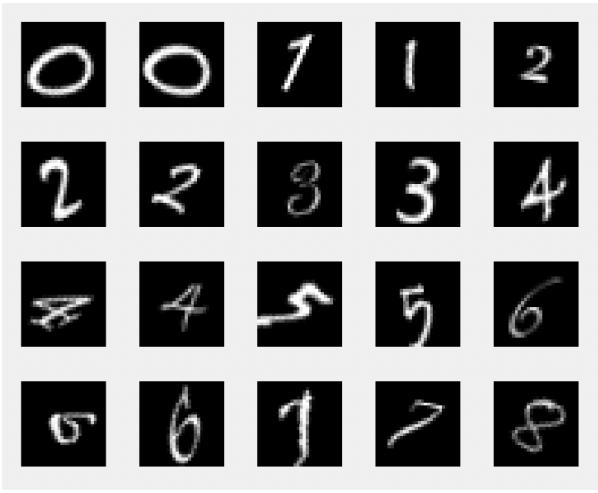


Figure 25: Test data

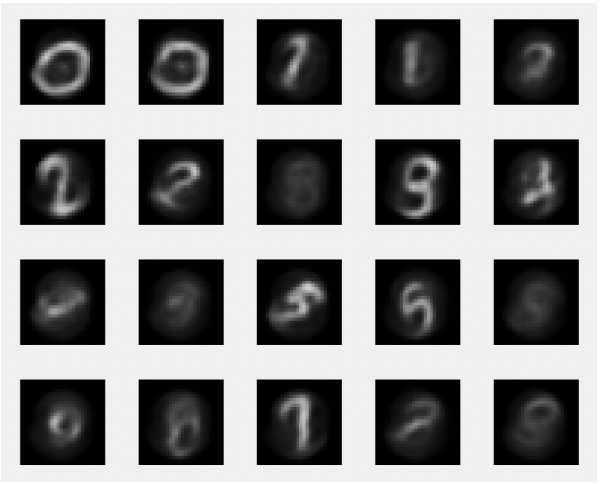


Figure 26: 50 neurons, 1000 epochs network results

Figure 27: 200 neurons, 2559 epochs network results

Figure 28: 400 neurons, 1000 epochs network results

Figures 26, 27 and 28 demonstrate the results of playing with different numbers of neurons in the hidden layer of the autoencoder as well as varying numbers of epochs allocated for training the network. It can be observed that the results have improved both by increasing the epochs threshold and by increasing the number of neurons in the hidden layer. Obviously, in all the three networks they had learned compressed representations of the input images because of having less neurons than inputs (i.e. 50 vs 784, 200 vs 784 and 400 vs 784), and the ultimate goal here is representing the input data as close as possible to itself by using a network of optimal sparsity (i.e. not to use 784 neurons if similar results can be attained by just 392 of them) and with optimal number of epochs allocated for training of network, because both of these components are important in terms of time efficiency as well as network's performance accuracy. In this context, in order to further improve the current reconstruction results, it could be interesting to further fine tune the numbers of neurons and epochs and, apart from that, to play with other

parameters' values used when creating the autoencoder, such as *L2WeightRegularization*, *SparsityRegularization* etc.

3.4 Stacked autoencoder

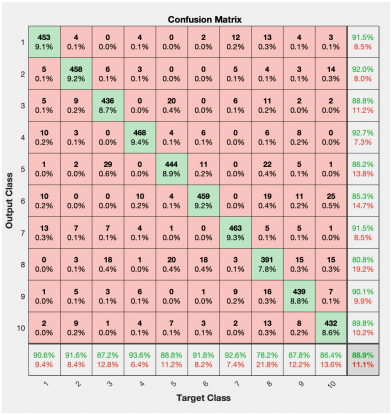


Figure 29: Deep net ($n_{h_1} = 100, n_{h_2} = 50$) before fine-tuning

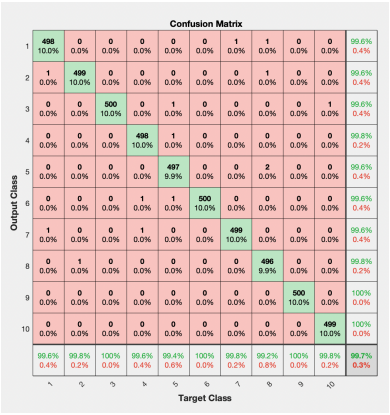


Figure 30: Deep net ($n_{h_1} = 100, n_{h_2} = 50$) *after fine-tuning*

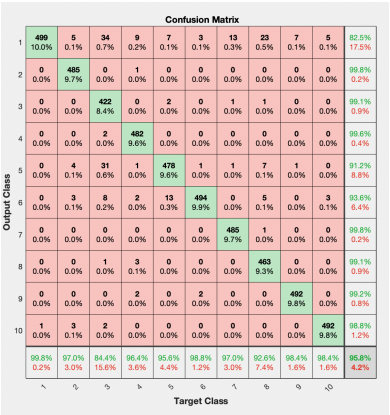


Figure 31: Normal neural net ($n_h = 100$)

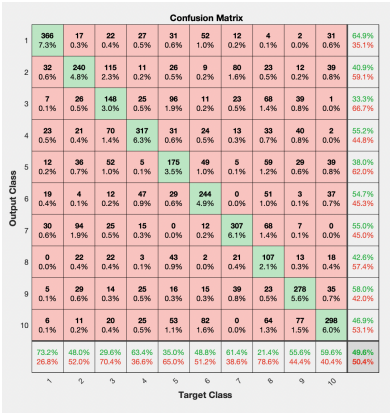


Figure 32: Deep net ($n_{h_1} = 50, n_{h_2} = 25$) *before fine-tuning*

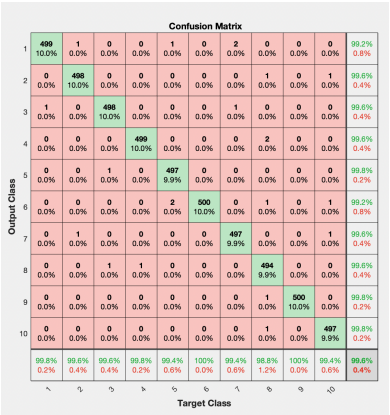


Figure 33: Deep net ($n_{h_1} = 50, n_{h_2} = 25$) *after fine-tuning*

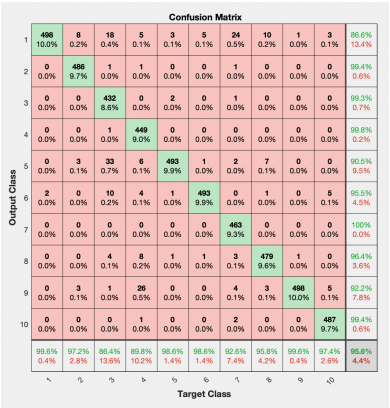


Figure 34: Normal neural net ($n_h = 100$)

	Before fine-tuning	After fine-tuning	Normal neural net	
$n_{h_1} = 100$ $n_{h_2} = 50$	88.86%	99.72%	$n_h = 100$	95.84%
	85.5%	99.84%		96.32%
	84.82%	99.86%		96.72%

Average	**86.39%**	**99.81%**		**96.29%**
$n_{h_1} = 50$ $n_{h_2} = 25$	49.6%	99.58%	$n_h = 100$	95.56%
	53.8%	99.36%		94.74%
	47.28%	99.58%		97.96%
Average	**50.23%**	**99.51%**		**96.09%**

Table 13: Comparison of deep and normal neural nets

To accomplish this task, first the default deep neural network was tested with 2 hidden layers of autoencoders consisting of 100 and 50 neurons respectively and a final softmax layer in order to do classification.

As shown in Figures 29, 30 and the corresponding statistics from Table 13, the average test set accuracy of this network over 3-fold cross-validation was 86.39% before fine-tuning and 99.81% after fine-tuning, the latter already signifying highly accurate results. It is worth mentioning that a normal neural network consisting of a single hidden layer with 100 neurons also turned out to perform very well (i.e. only slightly worse than the deep net) averaging in 96.18% test set accuracy over 6-fold cross-validation. Consequently, considering the time requirements for training neural networks, it would be interesting to see if similar results could be obtained with fewer neurons in the hidden layers of the stacked autoencoder (all the other parameters remaining intact).

The new network consisted of 2 hidden layers of autoencoders with 50 and 25 neurons respectively (which was not only twice as less as the initial deep net, but also 25 neurons less than the normal single layer neural network) and a final softmax layer.

Figures 32, 33 and Table 13 suggest that after fine-tuning the new deep net has shown very similar results (99.51% average accuracy on the test set over 3-fold cross-validation) to the initial deep net, though before fine-tuning it had performed very poorly being no better than a random model (50.23% average accuracy on the test set over 3-fold cross-validation). The latter confirms the boosting effect of fine-tuning as well, stating that after fine-tuning the smaller network has managed to perform almost as successfully as the larger one. In addition to this, the new deep net has also outperformed the normal neural network consisting of a single hidden layer of 100 neurons having even less neurons but an additional layer.

Assignment 4: Variational Auto-Encoders and Convolutional Neural Networks

4.1 Weight initialization and batch normalization

4.1.1 Scale of weight initialization

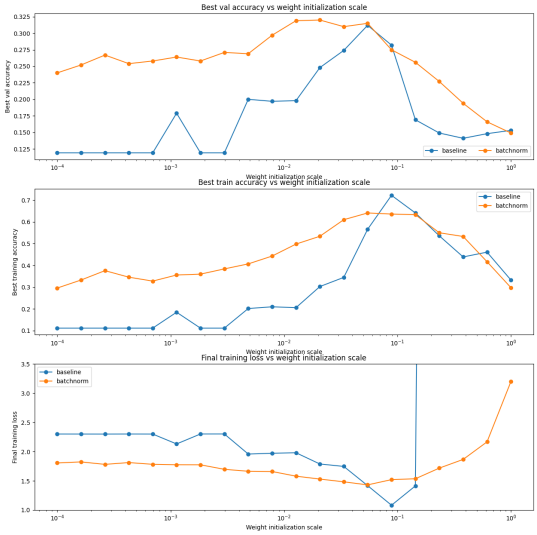


Figure 35

As demonstrated in Figure 9 from running the script $BatchNormalization.py$, the accuracy (both training and validation) of a non-batch-normalized model is affected much more negatively by various scales of weight initialization. In particular, in a non-batch-normalized model its accuracy quite strongly depends on the scale of weight initialization and very small or too large scales reduce the accuracy. This is because of the vanishing or explosion of the layer activation outputs meaning that with too small or very large scales of weight initialization the convergence of the network gets affected in a way that it might require much more epochs to improve its accuracy and hence be more time-consuming and less resource-efficient. On the other hand, with batch-normalization the network becomes less sensitive to the scale of the weight initialization, which, in its turn, results in a more stable model performance.
Figure 10 also strengthens the findings showing that the batch-normalized networks tend to outperform the non-batch-normalized ones.

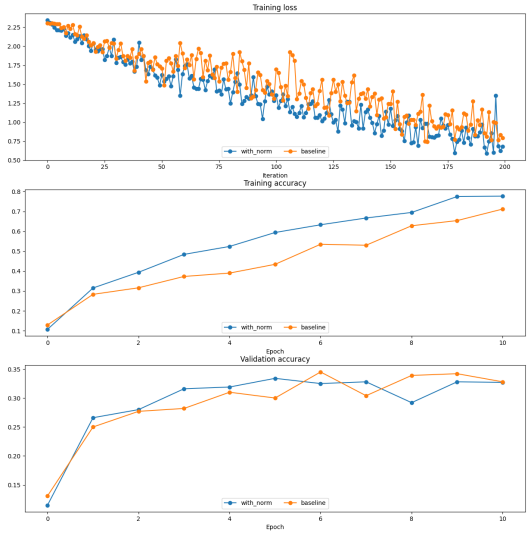


Figure 36

4.1.2 Relationship between batch normalization and batch size

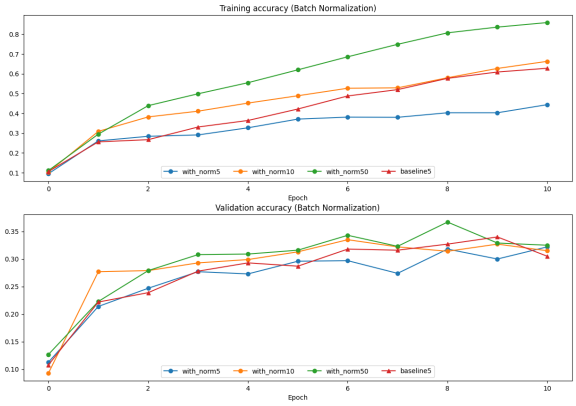


Figure 37

As it can be observed in Figure 11, the batch-normalized networks seem to perform much better with larger batch sizes. This happens because batch-normalization estimates the mean and standard deviation from the mini batches and uses them to scale the inputs to the layer and scale and shift the outputs, and following this logic the larger batch sizes give better estimates due to the so called Law of large numbers, where the estimates of population parameters get more reliable with increasing sample sizes. The consequences of this become the more accurate results that the network provides.

4.2 Variational autoencoders

4.2.1 Similarities and differences between stacked autoencoders and variational autoencoders

Both the stacked autoencoders and variational autoencoders are capable of extracting features from high-dimensional data and then reconstructing the input in an unsupervised manner. Just as the standard autoencoders (including the stacked ones), the variational autoencoders are architectures composed of both an encoder and a decoder and that are trained to minimize the reconstruction error between the encoded-decoded data and the initial data. But the stacked autoencoders have limitations in terms of content generation meaning that because of lack of some regularity of the latent space for stacked autoencoders, in order to encode and decode with minimum information loss, they may end up severely overfitting implying that some points of the latent space will give meaningless content once decoded. This problem arises naturally because during the training, the network takes advantage of any overfitting possibilities to achieve its task (i.e. encoding and decoding with as few loss as possible, no matter how the latent space is organised) as well as it can, unless it is explicitly regularised. So, in order to be able to use the decoder of the autoencoder for generative purposes, it has to be ensured that the latent space is regular enough. And the variational autoencoders are exactly the ones that can be defined as autoencoders whose training is regularised to avoid overfitting and to ensure that the latent space has good properties that enable generative process. In particular, in order to introduce some regularisation of the latent space, a slight modification of the encoding-decoding process is done: instead of encoding an input as a single point as with the stacked autoencoders, with the variational autoencoders the input is encoded as a distribution over the latent space, and a regularisation term over that returned distribution is added in the loss function in order to ensure a better organisation of the latent space. The model is then trained as follows:

- first, the input is encoded as distribution over the latent space,
- second, a point from the latent space is sampled from that distribution,
- third, the sampled point is decoded and the reconstruction error can be computed,
- and finally, the reconstruction error is backpropagated through the network and the weights are updated.

For the stacked autoencoders, the metric for the reconstruction error is the mean squared error between the original input and the reconstructed input from the decoder, and for the variational autoencoders, it is the expected value of the log-likelihood of input given the latent variable described by the approximate posterior distribution.

4.2.2 Optimizers: pros and cons

Scaled conjugate gradient is the default training algorithm that was used for the stacked autoencoders, while Adam optimizer was used for the variational autoencoders. The former is one of the conjugate gradient methods, which uses second-order derivatives in minimizing the error function, but unlike the other conjugate gradient methods, it does not require a line search at each iteration, thus reducing the computational costs. On the other hand, Adam is an

optimization algorithm whose parameters have a learning rate that gets individually updated from estimates of the first and second moments of the gradients. The authors describe Adam as combining the advantages of two other extensions of stochastic gradient descent. Specifically:

- Adaptive Gradient Algorithm (AdaGrad), that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
- 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 (e.g. noisy).

Adam realizes the benefits of both AdaGrad and RMSProp. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance).

Scaled conjugate gradient is easy to implement and is computationally less expensive in the sense that the number of computations per iteration is reduced because there is no need to perform a line search. However, it may require numerous iterations before convergence.

Adam is one of the most famous optimizers used in deep learning and has several advantages over the scaled conjugate gradient. It is computationally efficient, has small memory requirements, and works well on large datasets. However, it may have difficulties with converging to a global optimum and the problem of weight decay.

4.3 Convolutional neural networks

4.3.1 Toy example

i.

$$y = Ax$$

$$x = \begin{pmatrix} 2 \\ 5 \\ 4 \\ 1 \\ 3 \\ 7 \end{pmatrix} \quad y = \begin{pmatrix} 6 \\ 6 \\ 7 \\ 8 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \quad x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 6 \\ 6 \\ 7 \\ 8 \end{pmatrix} = \begin{pmatrix} 6 \\ 6 \\ 13 \\ 14 \\ 7 \\ 8 \end{pmatrix}$$

ii.

Simply transposing the matrix A and multiplying with y does not exactly recover x. Nevertheless, it does recover the dimensions of x.

4.3.2 $CNNex.m$

i.

The weights from the dimensions $11\times11\times3\times96$ of the weights of the first convolutional layer (layer 2) demonstrated by the function call $size(convnet.Layers(2).Weights)$ represent the following:

- $11\times11$is the filter size (i.e. both the height and width of the filter),
- 3is the number of channels in the input (i.e. red, green, blue as the inputs are colored images),
- 96is the number of filters applied to the input data in the current convolutional layer.

ii.

In the context of current network, the output height and width of the first convolutional layer can be computed as follows:

$$\frac{Input\ Size - Filter\ Size + 2 * Padding}{Stride} + 1 \quad (1)$$

where $Input\ Size = 227,\ Filter\ Size = 11,\ Padding = 0,\ Stride = 4.$ The above-mentioned formula would be slightly modified if the $Dilation\ Factor$was not equal to 1. Consequently, the output height and width for this layer will be $\frac{227-11+0}{4} + 1 = 55$. The output height and width of the max pooling layer will be computed analogously to formula (1) only by substituting the $Filter\ Size$by the $Pool\ Size$, namely

$$\frac{Input\ Size - Pool\ Size + 2 * Padding}{Stride} + 1 \quad (2)$$

where $Input\ Size = 55,\ Pool\ Size = 3,\ Padding = 0,\ Stride = 2$. Thus, the height and width of the resulting output will be$\frac{55-3+0}{2} + 1 = 27$, which, at the same time, will signify a 27 x 27 input for the 6-th layer of the network.

iii.

The final dimension of the problem (i.e. the number of neurons used for the final classification task) is 1000, because the loaded pre-trained convolutional neural network was trained to solve a multiclass classification problem consisting of 1000 classes. Compared to the initial dimension, which was 227 x 227 x 3 , the output dimension is 154 times less.

4.3.3 $CNNDigits.m$

To accomplish this task, several models with different configurations were tried. The given default model, consisting of 9 layers, namely 28 x 28 x 1 input, convolutional with 12 filter of size 5, ReLU, 2 x 2 max pooling with a stride of 2, convolutional with 24 filters of size 5, ReLU, fully connected dense with 10 neurons, softmax and classification layers respectively showed quite poor performance for neural networks by delivering only about 82% accuracy on the test set.

The chosen final model, training performance of which is demonstrated in Figure 38, consists of 7 layers. The first layer is the 28 x 28 x 1 input followed by a convolutional layer with 30 filters of size 5, a ReLU layer, a 2 x 2 max pooling layer with a stride of 1, a fully connected dense layer of 10 neurons representing the output size of 10 possible classes, a softmax layer and a final classification layer. Apart from the demonstrated training results, this network has also proved its high level of performance on the test set showing 97.56% accuracy.

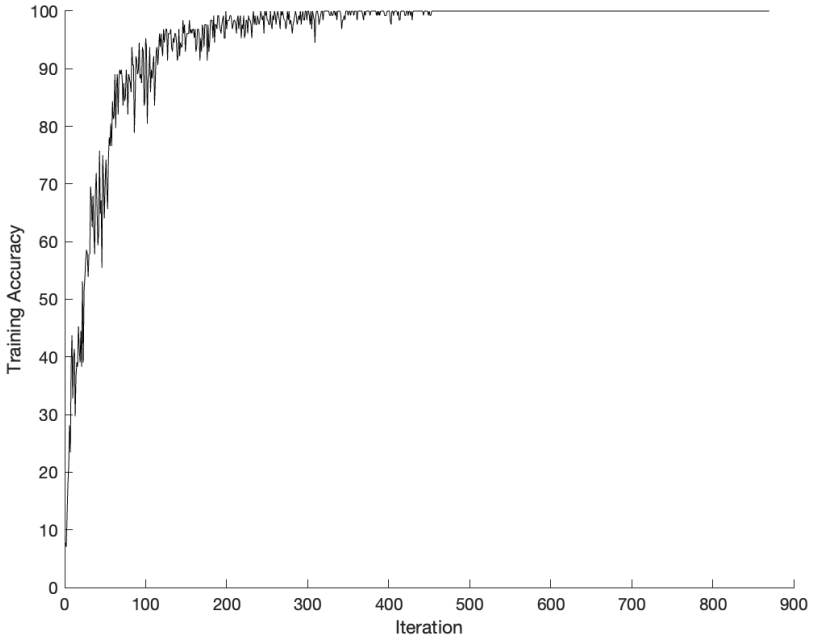


Figure 38: Training performance of the final network

References

[1] Johan Suykens, Data Mining and Neural Networks, 2020
[2] Climate Change: Earth Surface Temperature Data
(https://www.kaggle.com/berkeleyearth/climate-change-earth-surface-temperature-data)
[3] Breast Cancer Wisconsin (Diagnostic) Data Set
(https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29)
[4] UCI Machine Learning Repository
(https://archive.ics.uci.edu/ml/index.php)
[5] SOM
(https://en.wikipedia.org/wiki/Self-organizing_map)
[6] UFLDL Tutorial: Autoencoders
(http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/)
[7] MathWorks documentation: Train Stacked Autoencoders for Image Classification
(https://ww2.mathworks.cn/help/deeplearning/ug/train-stacked-autoencoders-forimage-classification.html)
[8] Diederik P. Kingma, Max Welling. An Introduction to Variational Autoencoders
(https://arxiv.org/ pdf/1906.02691.pdf)
[9] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015
(https://arxiv.org/pdf/1502.03167.pdf)
[10] UFLDL Tutorial: Convolutional Neural Network
(http://ufldl.stanford.edu/tutorial/supervised/ ConvolutionalNeuralNetwork)
[11] MathWorks documentation: Image Category Classification Using Deep Learning
(https://nl. mathworks.com/help/vision/examples/image-category-classification-using-deep-learning.html)
[12] MathWorks documentation: Create Simple Deep Learning Network for Classification
(https://nl.mathworks.com/help/deeplearning/examples/create-simple-deep-learning-network-for-classification.html)
[13] MathWorks documentation: Specify Layers of Convolutional Neural Network
(https://nl.mathworks.com/help/deeplearning/ug/layers-of-a-convolutional-neural-network.html)
[14] Joseph Rocca, Understanding Variational Autoencoders (VAEs), 2019
(https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73)
[15] Jason Brownlee, Introduction to the Adam Optimization Algorithm for Deep Learning, 2017
(https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/)
[16] StatQuest with Josh Starmer: Neural Networks
(https://statquest.org/video-index/)