Data mining and Artificial Neural Networks

Assignments report

Philippe MOUSSALLI
Master of Science in Biomedical Engineering
January 2020

Table of Contents

Assignment 1: Training Algorithms and Generalization

1.1) The perceptron and beyond

The perceptron is a linear classifier that consists of one input and one output layer with an activation function for binary classification. As such, it can be used to approximate linear functions. In the context of linear regression, it can be performed with a perceptron with no activation function that tries to minimize the sum of squared error by updating the weights accordingly using backpropagation. The number of input neurons is equals to the number of variables in the linear regression model.

Equation of activation function of perceptron: $a = w1\ x1 + w2\ x2 + \cdots . + wn\ xn + b \ \ (1)$
Equation for linear regression approximation: $y = a1\ x1 + a2\ x2 + \cdots + an\ xn + b \quad (2)$

Although linear regression and the perceptron are similar in that they both try to provide a linear approximation to estimate the value of a function, they do possess some fundamental differences:

	Linear regression	**Perceptron**
Output type	Continuous (number)	Discrete (class label)
Estimation	Best fit line	Decision boundary
Evaluation	Sum of squared errors	Accuracy (Confusion matrix)

Table 1: *Differences between linear regression and perceptron*

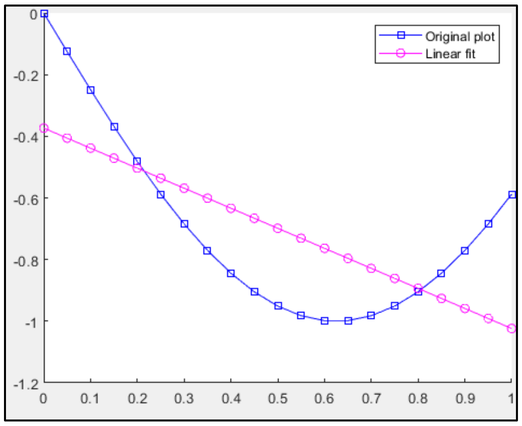


Figure 1: *Plot of a sine function and its linear approximation*

A sine function was plotted in **Figure 1** along with its linear approximation. It can be observed that a linear model is not capable of properly representing a sine function due to its relative complexity. This is a classic representation of underfitting in which a model does not possess enough degrees of freedom to accurately represent the given function. This can be remedied by increasing the order of the model (2nd degree).

In order to better approximate the sine function. A neural network was initialized with one hidden layer consisting of two neurons. The output of those neurons is modelled by a non linear hyperbolic tangent function *'tansig'*. In contrast, the activation function of the output layer is a linear transfer function *'Purlin'*.

The output of the neural network can be modelled using the following equation:

$$y = purlin(W_o.tansig(W_i.x + b_i) + b_o) \ (3)$$

Where:
W_i, W_o are the values of the interconnection weights between of the input layer and the output layer respectively
b_i, b_o are the bias values associated with the input and output layer respectively

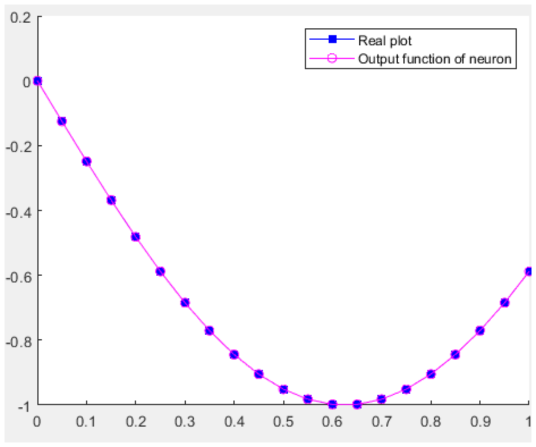


Figure 2: *Sine function and its approximation using a neural network with one hidden neuron*

The approximation of the sine function by a neural network with a hidden layer consisting of two hidden neurons can approximate the sine function with good accuracy **(Figure 2)**. Such a network provides better approximations of non linear functions as opposed to a perceptron with one hidden layer.

Theoretically, any non-linear function can be approximated as a linear combination of non-linear basis functions. As the complexity of the functions to be approximated increases, more neurons and layers need to be added to the network to model the function properly. However, by increasing the complexity of the network, it will start to memorize the training set and lose its ability to generalize on unseen new datasets.

1.2) Backpropagation in feedforward Multi-Layer network

2.1) Different training algorithms for backpropagation were tested, A few are described:

1) Trainbfg (Quasi-Newtonian algorithm): This method was developed to attend the limitation of the computations of Newton's method (Computation of second derivative) which requires the estimation of the Hessian matrix and its inverse. It attends to this drawback by calculating an approximation of the Hessian at each iteration which required only information from the derivative and the loss function. Although it converges after a few iterations, it still requires quite a lot of memory making it more ideal to use on small networks.

2) Traincgf (Conjugate gradient with Fletcher-Reeves method): This method converges faster than normal gradient descent algorithms as it searches for directions along the conjugate direction as opposed to the direction in which the function decreases more rapidly. It can be considered a good algorithm for large networks as it does not require a lot of memory.

3) Traingd (Gradient descent): This method estimates the gradient of the steepest descent of the loss function that is estimated by the sum of squared error of the output and adjusts the weights accordingly. It is susceptible to converge at a local minimum.

4) Trainlm (Levenberg-Marquardt algorithms): Like the Quasi-Newtonian method, this method estimates an approximation of the Hessian for improve efficiency and speed. It performs like a gradient descent when the parameters are far from the optimal values. As it reaches the optimal value, it performs similarly to the Newtonian method. This allows it to converge quickly to a local minimum.

Original data set							
	Training algorithm						
Epochs	trainbfg	traincgf	traincgp	traingd	traingda	trainlm	trainbr
1	0.093	0.167	0.130	0.131	0.098	0.772	0.953
14	0.879	0.824	0.674	0.032	0.069	1.000	1.000
1000	1.000	0.999	0.999	0.618	0.874	1.000	1.000

Table 2: *R values for training a network using different algorithms for different epoch between output and testing set*

The performance of the algorithms was examined using a feedforward neural network with 1 hidden layer consisting of 50 neurons. The initialization of the interconnection weights was set to be the same in order to provide a standardized condition for performance comparison.

The results in **Table 2** show that 'traingd' performed worse out of all the methods for all epochs, however it was the fastest to converge. 'traingda' which is a gradient descent algorithm that uses adaptive learning performed considerably better than 'traingd' in terms of accuracy at a similar convergence time. This makes it a better choice than its precedent counterpart for approximating extremely large network or classification problems as it is relatively fast and computationally inexpensive but lack in terms of accuracy.

'trainlm' was the fastest to converge reaching an optimal correlation at only 14 iterations. It is relatively more computationally expensive than 'traingd' and 'traingda'.

Noisy data set							
	Training algorithm						
Epochs	trainbfg	traincgf	traincgp	traingd	traingda	trainlm	trainbr
1	-0.140	-0.082	0.039	-0.064	-0.147	0.546	0.803
14	0.839	0.486	0.253	0.029	-0.016	0.873	0.912
1000	0.860	0.892	0.917	0.767	0.731	0.806	0.894

Table 3: *R values for training a network using different algorithms for noisy data set*

A random noise at about 40% of the range of the function was added to It. In terms of accuracy 'trainlm' performed the best at about 14 iterations only to be surpassed by 'traincgp' at 1000 iterations **(Table 3)**. We can notice a drop in the correlation of 'trainlm' at 1000 iterations probably due to overfitting: Since the epochs are preset, we are not allowing the validation set to regulate the training iterations required before the error starts to increase. Thus, the results above provide only a slight estimation about the performance of each of the algorithms.

Personal regression example: The data was drawn randomly from the underlying distribution in order to approximate over all the data set and avoid potential bias.
Different settings were tested for the networks until a final configuration with the following parameters was chosen: 2 input layers, 1 hidden layer with 10 layers, 'tansig' activation function for the hidden layer and 'purelin' activation function for the output layer. The learning algorithm was chosen as 'trainlm' for its ability to obtain a low MSE and good generalization properties. The choice of 10 layers provided a low MSE on the test data (4.28x10^{-4}) as opposed to (1.15x10^{-2}) for 6 hidden neurons. The influence of the choice of hidden neurons can be further visualized from the plot level error curve **(Figure 3)**. Although adding more hidden layers or neurons could reduce the MSE even further. It requires a longer time for convergence and may lead to overfitting. Different non-linear activation functions were tested and 'tansig' provided the best approximation for the given non linear function. Since we are dealing with a regression problem, a linear activation function was chosen on the output layer.

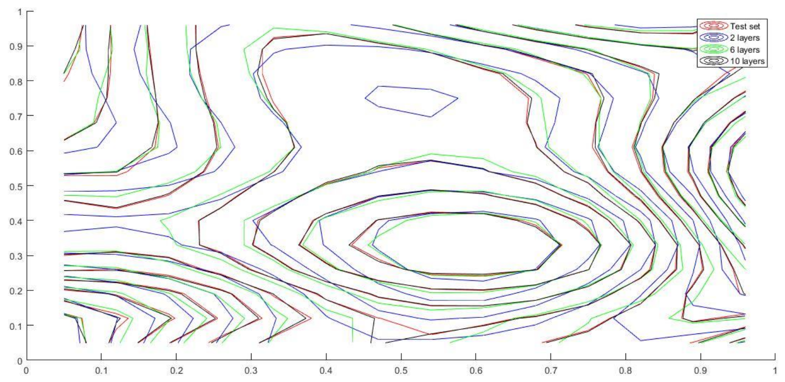


Figure 3: *Error surface plot for different neurons in the hidden layer*

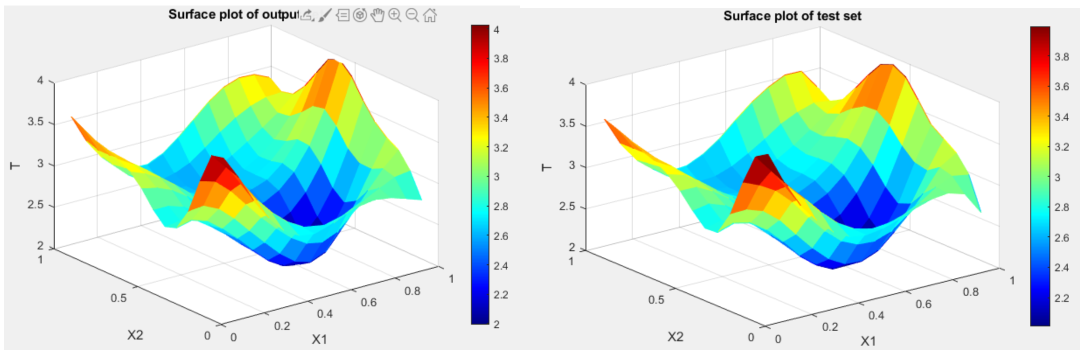


Figure 4: *Surface plot of output of neural network and of test set*

1.3) Bayesian inference

Bayesian regularization provides a method for dealing with overfitting in ANN. The algorithm tries to provide a best fit for the data by maximizing the posterior probability of the weights whilst providing good regularization conditions to avoid overfitting. Since small weights generally perform better on unseen data. The algorithm works by introducing a penalization term for large weights. The model focuses on optimizing two parameters: β for model fit optimization and α also known as weight decay to penalize large weight value.

$$M(w) = \beta E_D(w) + \alpha E_w(w) \quad (4)$$

If α is too large, then the posterior density of the weights is skewed to zero, this will lead to a high bias but relatively low variance. On the other hand, if β is too large, then the model will fit the data but fail to generalize over new ones. Thus, the optimization aims to tune the two parameters with respect to each other in to obtain an adequate bias/variance tradeoff. It does so by defining the number of effective parameters in the neural network (setting additional weights to zero). In that regard, it is optimal for estimation of complex functions whose number of hidden neurons and layers that need to be initialized is hard to approximate.

It can be seen from **(Table 2)** and **(Table 3)** that the algorithm performs well with clean and noisy data and that it converges to an optimal value at low values of iterations. We now consider the example used in the second part of the exercise for estimating sin(x^2). This time we overparametrized the network to contain 3 hidden layers with the following number of neurons: 50, 20 and 10 and tested for clean and noisy data.

The correlation result for clean and noisy data were as follows: 0.92 and 0.89 for 14 iterations which was slightly better than most of the available learning algorithms for the overparametrized network. This effect is attributed to the ability of the algorithm to well estimate the number of effective parameters. Since the algorithm relies on LM for learning, it requires computation of the estimation the hessian of the performance index thus making it computationally expensive.

1.4) Curse of Dimensionality

The goal of this exercise session was to obtain different estimations of the sinus cardinal function using two different models: a neural network and a polynomial. Two different dimensions were tested for the data (2 and 7) and the degree of polynomial was varied between 1 to 8. A fixed radius of 7 was chosen for all examples for simplification of comparison.

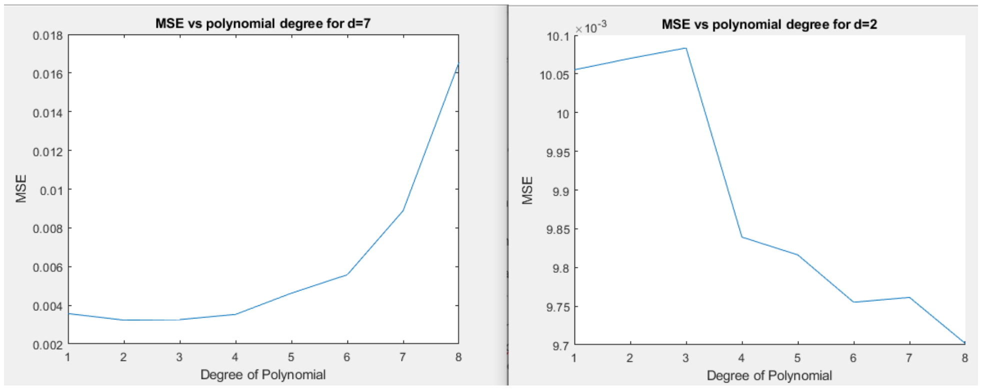


Figure 5: *MSE (test data) vs degree of polynomial for cardinal sinus estimation for different dimensions*

It can be seen in **Figure 5** that the MSE of the polynomial decreases with the increase of order for data with lower dimension as the polynomial increases. The opposite can be seen when comparing the data in higher dimensions, that is because the number of parameters increases exponentially (6435 for 8th order compared to 45 parameters in d=2 for the same order). A higher degree of freedom means that the model with tend to overfit the training data and fail to generalize well when applied to new unseen data (training set in our case). This effect is not as prominent when estimating using a NN as seen in **Table 4** where the MSE keeps a relatively stable value despite the change of number of dimensions. The number of dimensions in NN can be considered independent of the dimensions of the input space as the number of interconnection weights vary almost linearly with an increase of dimensions. Hence, increasing the number of dimensions does not severely affect the estimation of a NN. It was also noticed that the run time for polynomial estimation was considerably larger for neural networks as the number of dimension and order increase.

Although neural networks are better able to handle the curse of dimensionality then other approximators. It is still advisable to reduce the number of dimensions in the pre-processing phase for better approximation of useful data (exclude unnecessary data) and faster convergence.

Table 4: MSE and number of parameters for NN

Dim	**MSE (NN)**	**#parameters (NN)**
2	4.5E-03	24
7	3.5E-03	44

Assignment 2: Time Series Predictions and Classifications

2.1) Time series prediction

So far, we have seen how neural networks are used to approximate different linear and non-linear functions. In this exercise session, we are going to apply a network for time series prediction for the Santa Fe laser data set (non linear time series). We are given 1000 points and the aim is to predict the next 100 points.

This is an example of a neural network autoregression model where there is a correlation between past values and future values that succeed them. In that case our model not only depends on the current input but on past outputs as well. Those outputs can be either embedded in the training set or part of the ongoing predictions of the network that are fed back to it as it continues to iterate. It is worth to note that the AR model is characterized by a stochastic process which means that uncertainties are bound to appear while forecasting. The order of the model is characterized by the lag which indicates the number of past values that are to be included in the prediction. The optimal amount of lag is dependent on the type of data set to be analyzed and must be optimized accordingly.

We have chosen a neural network with one hidden layer with 'Levenberg Marquardt' training algorithm. In order to give well rounded prediction of the next 100 samples, we needed to test for a range of hyperparameters (Number of hidden neurons in the hidden layer, amount of lag). The values of lag range from 20 to 125 samples and the number of hidden neurons to be tested are 20,30 and 50 neurons. The training simulation was conducted 10 times and the results were averaged to account for the different variations that arise when the network in initialized.

For the first part of the exercise session, we have trained our neural network for the different range of chosen hyperparameters and estimated the error on the test set. No validation set was used in the training or for early stopping. For the second part, we partitioned the training data into sub-training set and validation set **(Figure 6)**.

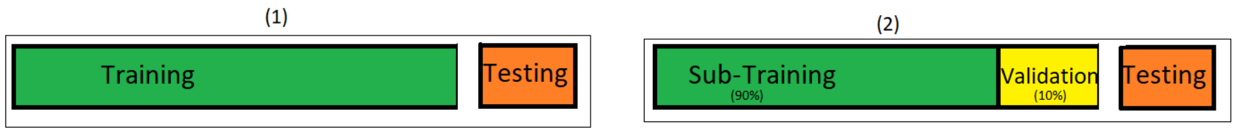


Figure 6: *Different types of data partitioning used for hyperparameter tuning*

If we only use the test set to evaluate the performance of our model. We might obtain good results on test set, but those results do not generally mean that the model will do well when it is applied on new data. The estimation we get is biased and might lead to overfitting. For this reason, we need a validation set to tune the hyperparameters of our model. A good performance of the model on the validation set does not guarantee that it will perform well on the testing set. Nevertheless, we will obtain a more unbiased approximation of its actual performance that gives a better overall estimate on how well it behaves.

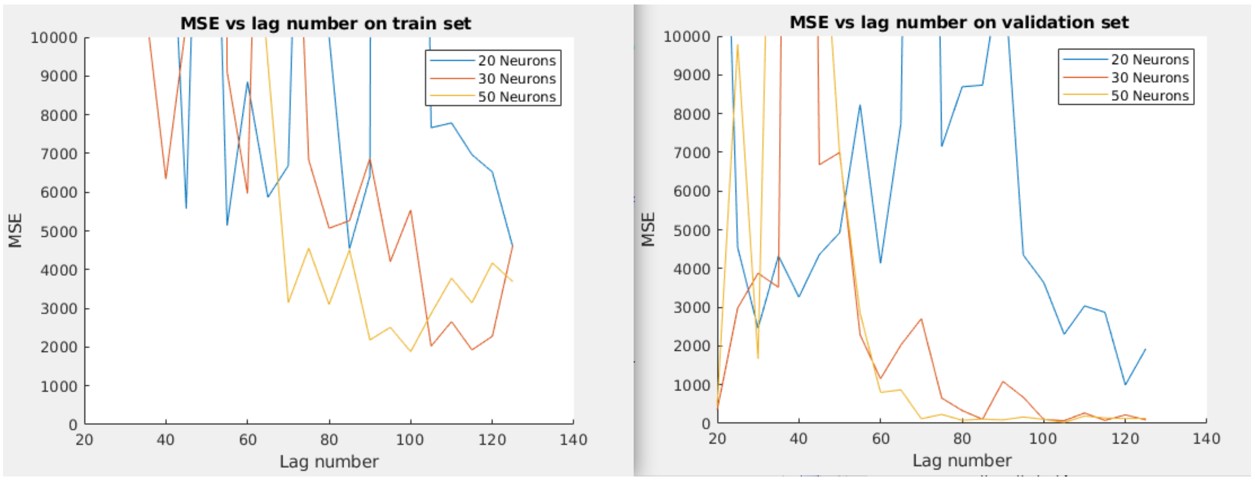


Figure 7 *MSE for training based on the using the test set (left) and the validation set (right)*

In **Figure 7**, we are comparing different MSE values used during the training process (y-axis was restricted for better visualization of optimal hyperparameters to choose). The performance of the network for 20 neurons is poor compared to the other configuration for all lag values. This is due the fact that the network required more hidden neurons to approximate the non-linearity of the given time series.

The MSE of the validation set provides a better approach for hyperparameter selection as mentioned earlier. For that reason, a lag of 80 was chosen with 50 neurons in the hidden layer as this is the threshold for which the MSE stabilizes. **Figure 8** demonstrates the result of the prediction for the chosen hyperparameters.

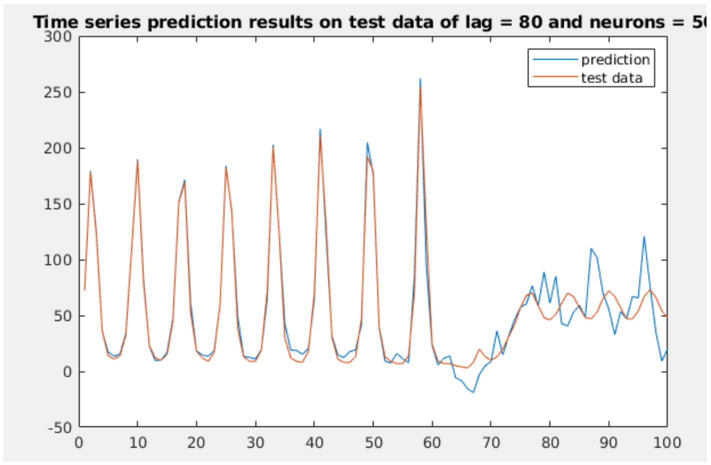


Figure 8: *Santa Fe times series prediction for chosen hyperparameters*

2.2) Concentration analysis

We now consider another time series data prediction that is concerned with predicting the concentration of PM 2.5 particles for the city for Shanghai. We are now given 1000 data points: 700 for training and 300 for testing. Like the previous approach, we divided out training dataset into sub-training (400 samples) and validation (300 samples) for hyperparameter configuration. The network was trained using 'trainlm'.

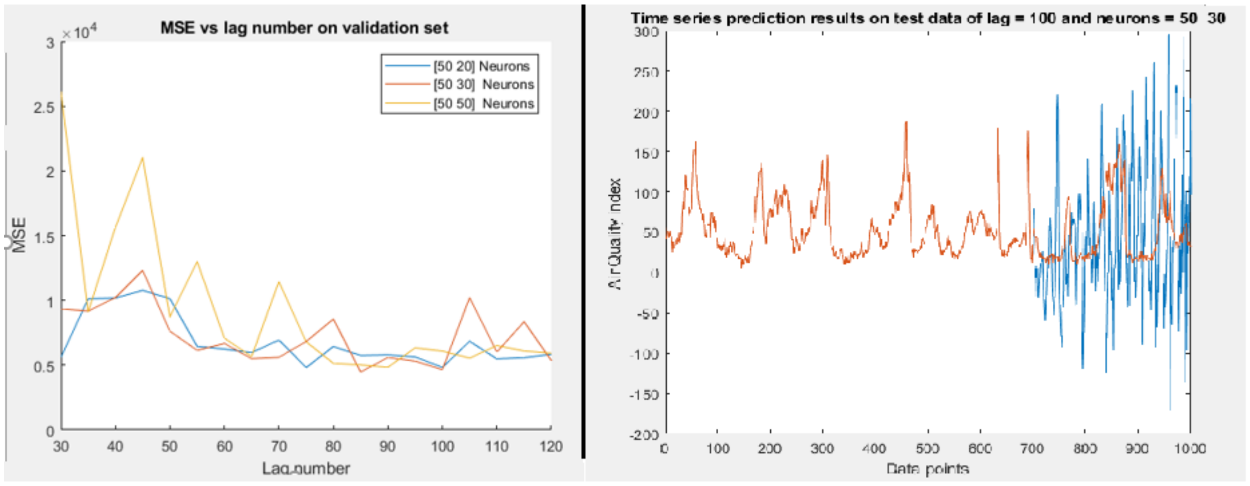


Figure 9: *MSE for training based on the using the validation set (left). Prediction results of last 300 samples overlapped with actual dataset (right)*

We decided to include two hidden neurons since the function to be estimated is more complex and there are more datapoints to be considered. Contrary to the previous time series dataset, we do not obtain any convergence to an optimal minimum when estimating the MSE error on the validation dataset regardless of the choice of hidden units **(Figure 9).** There is a general trend of decrease of MSE as the lag increases with some fluctuations probably related to overfitting. Even when the error tends to converge to a local minimum, it is a few orders of magnitudes higher than the span of our original dataset.

In the end, the choice of optimal hyperparameter -albeit not optimal- was chosen with a lag of 100 and for 50 and 30 neurons for the 1st and 2nd hidden neurons respectively. Afterwards, the training was implemented on the training set and the predicted results were plotted overlapping with the original test set **(Figure 9).**

As expected, the prediction was quite poor and did not actually reflect the actual trend or order of magnitude of the actual data. This can be explained by the fact that the data exhibits a more complicated trend than the SantaFe dataset (which resembles a sinusoid). In addition, 300 data points were to be predicted as opposed to 100 in SantaFe. The results do not actually give any informatory value. A more complex model is needed in order to predict the underlying trend and provide adequate estimates that could be of practical use.

<u>2.3) Classification</u>

In this part of this exercise, we will deal with a binary classification problem for predicting whether a tumor is malignant or not. We start off with a large dataset consisting of 30 different features that characterize the tumors. A correlation matrix was plotted to visualize the dependencies between different features.

We tested our neural network for different training methods and neuron numbers. The number of epochs was fixed to 50 for each configuration. K-fold cross validation was used to measure performance of the different classifiers (K=10) where 10% of the data was chosen randomly for testing to avoid any statistical dependencies while the rest was chosen for training. Different metrics were estimated to characterize the performance:

1) <u>Accuracy:</u> the ratio of the samples that were correctly classified (positive or negative)

2) <u>Sensitivity:</u> The ratio of correctly classified positive cases. In the medical contests, it is generally favorable to obtain a higher percentage of sensitivity in order to avoid a missed diagnosis

3) <u>False positive rate:</u> The ratio of values that were falsely classified as positive label

The summary of the results can be visualized in **Table 5.**

Training method	Accuracy			Sensitivity			False positive rate		
	Number of neurons								
	20	30	50	20	30	50	20	30	50
traingd	78%	76%	83%	79%	81%	87%	23%	26%	20%
trainlm	98%	95%	96%	98%	94%	97%	2%	3%	3%
trainbr	95%	95%	96%	95%	94%	95%	4%	4%	3%

Table 5: *Performance evaluation of different neural networks (K-fold cross validation)*

We can see that the results were relatively poor for gradient descent method since this training method may tend to converge at a local minimum. Although the algorithms perform slightly better as the number of the neurons increase.

Both Levenberg-Marquardt and Bayesian regularization methods showed consistently good results for different number of neurons configurations. For measuring the performance on the testing data. ‘trainlm’ was chosen with 20 neurons as it converges faster than ‘trainbr’ and performs well on all estimated metrics.

The model with the optimal parameters was later trained on the whole training data and the predictions were made based on the hyperparameters of the test data set. The model gave very good performances with an accuracy of 95% a sensitivity of 98% and a false positive rate of 5%.

2.4) Automatic Relevance detection (ARD)

Automatic relevance detection is a method used to determine the degree to which the input variables are relevant for a given target vector. It does so by initializing separate hyperparameters α_i for each set of weights associated with a given input in the model network (prior on the weights). The prior is set to have a Gaussian distribution. After the network is trained, a Bayesian re-estimation is applied and the hyperparameter values are updates to get a posterior approximation associated with the inputs and outputs. The α values are inversely proportional to the variance so that the highly relevant inputs are associated with small α values and large posterior interconnection weights.

We consider the previous example with the breast cancer dataset that consists of 30 different inputs. After applying the provided ARD demo for our dataset, we obtained different posterior weights (α_i) associated with each feature. We trained our original MLP network with the optimal hyperparameters chosen before (20 neurons in the hidden layer,'trainlm') and reduced the training and testing data to 4,7 and 14 features respectively such that the first hyperparameters were the ones associated with the highest relevance. We trained the network on the training set and estimated the performance by plotting the confusion matrix associated with the test set **(Table 6).**

n=169	Number of inputs					
	4		7		14	
	P	N	P	N	P	N
P	99	8	100	7	98	9
N	3	59	1	61	1	61
Accuracy	93%		95%		94%	

Table 6: *Confusion matrices for the breast cancer dataset trained on different numbers of features (based on ARD estimation)*

The network performed slightly worse when only the first 4 features were selected compared to 7 or 14 features. It seems that 7 features are a good choice of selection for this dataset since the performance is on par to when all the features were selected (accuracy of 95%, slightly different results with false-positives and false-negatives).

By removing the irrelevant features, we managed to keep only the ones with relevance and reduce the complexity of our network. A simpler network with a reduced input space is desirable when dealing with a high dimensional dataset where accuracy and computational power are of vital importance. In some cases, it might even improve the overall accuracy of the results if some of the features are only considered to be 'noise' that do not contribute to the prediction of the outcome.

Assignment 3: Unsupervised Learning and Data Visualization

3.1) Principal component analysis

3.1.1) Redundancy and random data

Principal component analysis (PCA) is a widely used method in the field of machine learning mainly in the preprocessing stages. It is best employed with high dimensional data to reduce the number of overall dimensions by summarizing the correlation in the dataset with smaller set of linear combination.

We were tasked to use PCA to reduce the number of dimensions for two different datasets: The first one consists of a 50-dimensional Gaussian dataset and the second one aims at detecting serum cholesterol levels from 21 measurements of the spectral content of a blood sample. For both cases, we applied the PCA method to project the dataset onto different number of eigenvectors. MSE of all the observations was estimated by reconstructing the reduced dataset and comparing it to the original one (**Figure 10).**

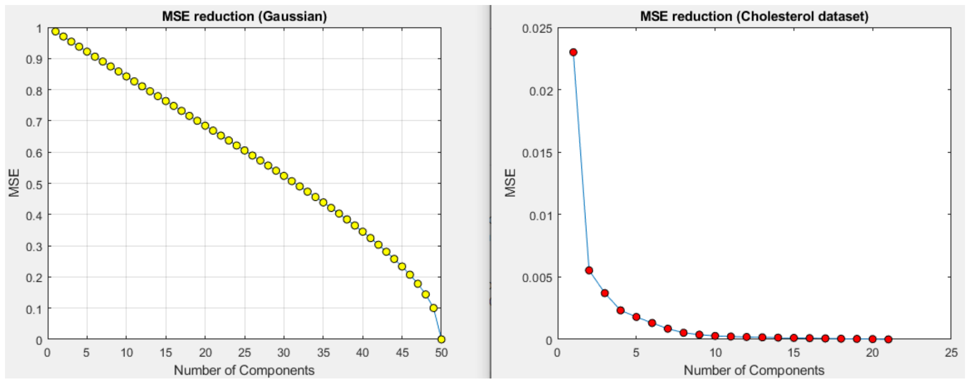


Figure 10: *MSE reduction as a function of number of components the dataset is project onto for the Gaussian (left) and the cholesterol dataset (right)*

From the figure, we can notice that the reduction of the random data exhibits a linear trend. In other words, the MSE decreases linearly as we increase the number of components to project onto. This is explained by the fact that the data is highly uncorrelated due to its noisy nature and thus the contribution of variability onto each component are similar. This is further confirmed by the fact that the eigenvalue of the covariance matrix is similarly spaced one to the other such that there isn't a component which is highly discriminative. In order to obtain 95% reconstruction of the original dataset, the dataset need to be projected onto the first 46 components.

This is not the case with the second case, the MSE rapidly reduces for the first components with the highest eigenvalues and then converges at around 10 components indicating that we could reduce the dimensions of our dataset by half with minimal reconstruction error. This implies that some of the features of the dataset are highly correlated with each other and the structure can be expressed when projection onto the principal components. Here, projection on one component only gives information about 95% of the dataset variability.

3.1.2) PCA on handwritten digits

In this example, we are dealing with high dimensional images (256 dimensions). We want to reduce the number of dimesnions that are not considered to be indicative objects (in our case identifying the number 3) by choosing an optimal basis function of lower dimension that will lead to a suitable reconstruction of our bulk element. **Figure 11** demonstrates the reconstruction of the letter 3 using 1 component and 4 components. Comparing with the original image, reducing the dimension of the from 265 to 4 dimensions allows us to recover the salient features of the original input image.

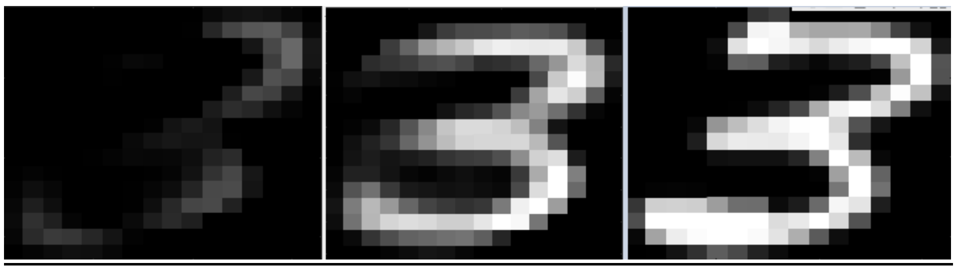


Figure 11: *Original image (right). Reconstruction with first 4 component (middle). Reconstruction with first component (left)*

We then plotted the least square reconstruction error for the first 50 components as well as the corresponding cumulative eigenvalues for those same components **(Figure 12)**. The reconstruction error and cumulative sum are inversely proportional. As the cumulative sum of eigenvalues converges, projecting the data onto additional components does not lead to a better discrimination of the image as the additional components are not representative of the overall variance in the dataset. Hence, the reconstruction MSE will reach a minimum and converge as well.

The quick fall of the MSE at the first few components is illustrated by the fact that the cumulative sum has a high slope in the beginning and then slowly converges to approximately zero.

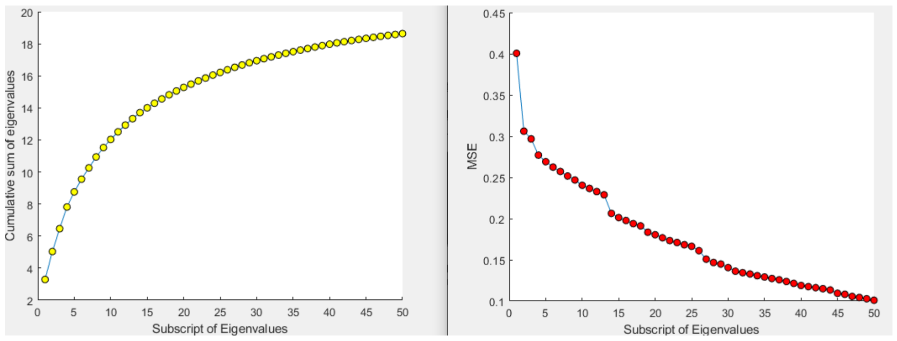


Figure 12: *Cumulative sum of eigenvalues (left). MSE reduction as a function of number of components the dataset is project onto (right)*

3.2) K-means Clustering

K means is a popular unsupervised classification that aims to partition the data into K cluster in which each datapoint belongs to the cluster with the nearest centroid. The algorithm begins by initializing random clusters and the data points are assigned to the nearest cluster. The clusters are then reassigned to the mean of the data points in an iterative way until the data points are no longer reassigned or convergence limit is reached.

The given example consisted of 3 rings (classes) of different diameters contained one within the other with a few outliers from each class overlapping the other class **(Figure 13)**.

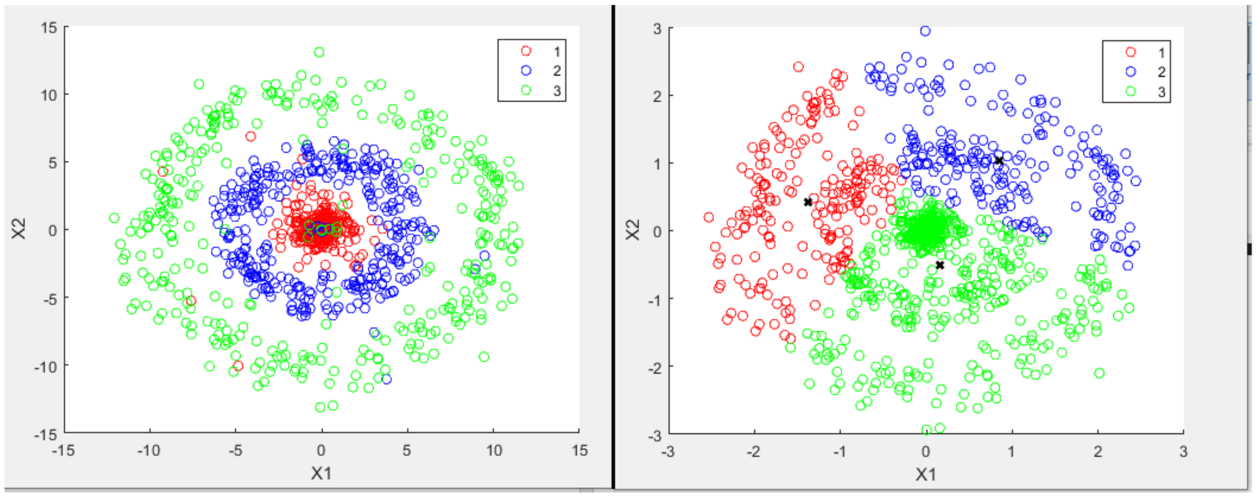


Figure 13: *Original dataset partitioning (left). Data set partitioning after applying k-means (right).*

The data was normalized prior to classification so that they are centered around the origin and have a standard deviation of one. Since we have prior knowledge about the classes of our data, we set K=3 in order to cluster the dataset into 3 classes.

We can see that the obtained results do not reflect on the actual trend of the original dataset as a lot of data is misclassified. The reason for that is that K-means is dependent on the geometrical shape of the data. Ideally, K-means would be used when the data exhibits a spherical like shape and is distinguishably separate one from the other. In our case, most of the data is clustered around the center which means that that the centroids will be pulled towards it and misclassify some datapoints around it. If the rings were not contained within each other but in distinct areas within the plane, the algorithm would have probably performed better.

It is worth to note that since the clusters are randomly initialized, the output of the classifier will be different at each execution. To check the validity of K means, it is recommended to use cross validation in order to generalize on the performance of the algorithm.

3.3) Density based methods

3.3.1) Gaussian mixture models (GMM)

Gaussian mixture model is classification method that aims to estimate the source of the datapoints, the first step of the algorithm is to initialize random priors (Gaussian distribution) depending on the number of defined clusters (Similar to K-means where K cluster seeds are initialized randomly). The points are assigned a posterior probability (Bayesian posterior) for each class that varies between 0 and 1 (soft clustering). In K-means, the points are assigned to either zero or one (hard clustering). This step is called the Expectation step. The second step is to update the mean and the standard deviation of the Gaussian parameters to fit the points assigned to them. In other words, the algorithm adjusts the mean and variance of each gaussian models so that it maximizes the likelihood that each sample came from the distribution. This is called the Maximization step. The algorithm iterates between the two steps until a convergence is reached.

In our examples, we tested again for the ring dataset and tried to perform an unsupervised classification with GMM for a combination of hyperparameters **(Figure 14)**.

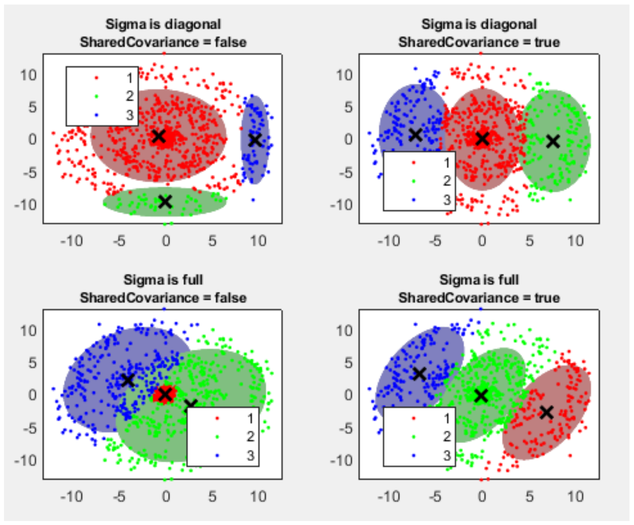


Figure 14: *GMM algorithm applied on the ring algorithm with k=3 and different combination of hyperparameters*

Since the 'rings' dataset consists of three different rings of different variances and are more circular shape. Initializing the gaussians to have similar covariance and diagonal shape will lead to a poor estimation because they do not reflect upon the actual properties of the distribution of each class. Hence, the best hyperparameters is when 'Sigma' is set to 'full' (the gaussian can adapt any shape and position) the covariances of the different distributions are not equal. We can see that for this configuration, the inner most ring was classified properly as it can be represented by Gaussian distribution. However, for the other two classes, the classification is elusive since the clusters do not belong to any standard parametric class of densities.

3.3.2) Kernel density estimation

All the clustering methods that we have tried so far rely on the assumption that the clusters of the datapoints follow a parametric distribution. In real life data application, the data distribution may follow a non-parametric pattern which renders the use of parametric clustering algorithms infeasible. Kernel density estimation (KDE) provides a probabilistic model that describes the distribution of the data for both parametric and non-parametric functions. That way we can model the overall distribution of our data. Here, we are not required to specify the number of clusters in the data like in K-means or GMM. Other applications for KDE include simulating new random data that have the same underlying distribution as our input.

We are going to apply the Density based spatial clustering of applications with noise (DBSCAN) algorithm to cluster our data. DBSCAN's main assumption is that clusters are dense group of points such that a point belongs to cluster if there are a lot of points around it that belong to that cluster. This is done by assignment of two parameters: "epsilon" and "minPoints". We pick a random data point of our dataset and estimate whether there are more that minPoints in the epsilon distance around it. When the condition is met, then the chosen point and all the points in the epsilon region belong to a cluster. Otherwise it counts as a noise. The algorithm converges when all the points have been labeled to belong to a certain cluster or to be noise.

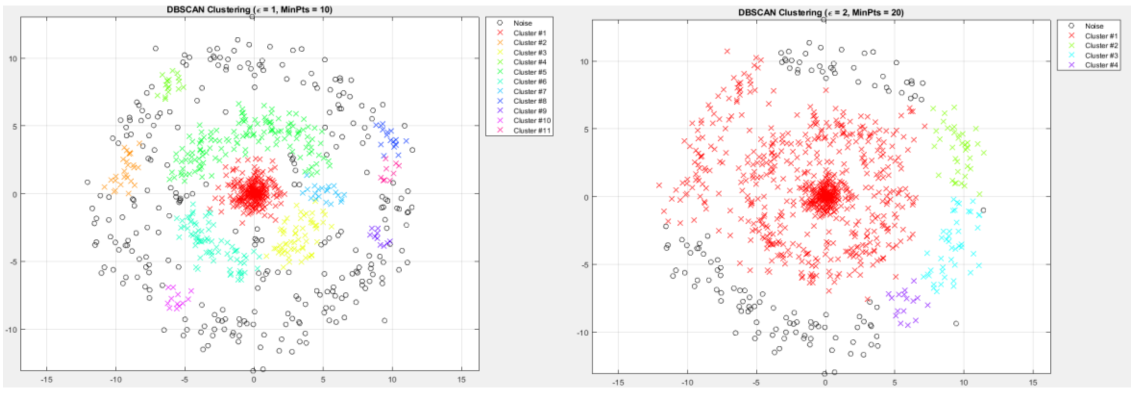


Figure 15: *DBSCAN clustering applied to the ring dataset for two different configurations: 1) ε=1, MinPnts=10 (left). 2) ε=2, MinPnts=20 (right)*

Figure 15 demonstrates the application of DBSCAN for our 'rings' dataset. For the first configuration where we chose a small epsilon is equal to 1 and MinPnts is 10. The inner ring was classified relatively well since the cluster points are densely packed together. However, datapoints from the outer rings are going to be classified as separate clusters or noise since their datapoints are more dispersed one to the other. A possible remedy for this issue would be to increase the epsilon area and points threshold to better classify the outer rings. But the obtained effect is that they will get clustered in with the inner ring since there isn’t a distinct separation between the three rings: a clustering chain will form from the inner ring and spread to the outer ring.

3.4) Prototype Vectors and Data Visualizations

Self-organizing maps is an unsupervised clustering algorithm that aims to simplify a high dimensional data into a representation that is more easily interpretable. It reduces the high dimensional features to a 2 dimensional hexagonal or rectangular space where the inputs of similar features are assigned to neighboring clusters. This dimensionality reduction is achieved through vector quantization where the number of formed clusters are the **codewords** (nodes in the training network)**.** In SOM, codewords are related to the grid size of the SOM. The training occurs in an iterative step as follows:

1) Node weights are initialized randomly.
2) A vector is chosen at random and from the training set and presented to the list of **codewords** (nodes)
3) Every node is examined and the one which one’s weight most resembles the input vector is known as the best matching unit **(BMU).** This is calculated through the Euclidean distance.
4) Depending on the learning rate (which is estimated as a radius around the BMU). The weights of the neighboring neurons are adjusted to make them more like the input vector. The closer the node, the more its weights get altered.
5) The steps repeat for all the different vectors in the training set

In order to visualize how the neurons of the lattice are adjusted are adjusted with SOM. We implemented the algorithm on a given dataset *(‘banana.mat’)*. The weights associated with the neurons were plotted in the input space of the dataset before and after applying SOM algorithm.

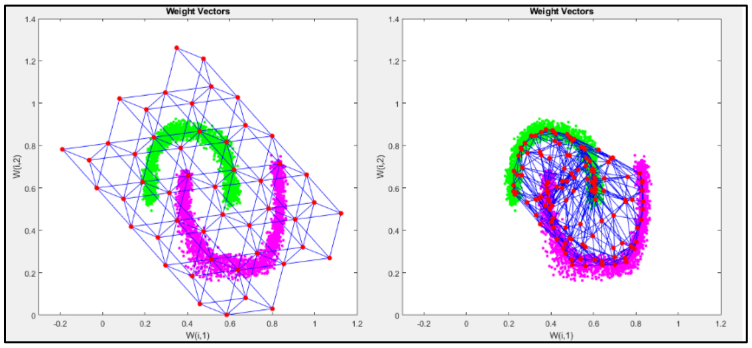


Figure 16: *SOM weights before and after SOM training for ‘banana’ dataset*

Since the neurons of the lattice do not have an activation function, the nodes are represented exclusively by their weights. We can visualize in **Figure 16** the change of position of the datapoints after training. Most of the datapoints are clustered on the dataset. The output units are crowded in the center of each U-shaped class where the density of input patterns is concentrated, this is due to the competitive rule where the winning neurons tend to pull the neighboring neurons close to it. The adjusted lattice space provides a good overall representation of the distribution of the dataset.

We will now try to visualize the UCI dataset of Covertype Data set using SOM. This dataset contains 581012 records of 54 input variables related to cartographic variables. The predictions are for 5 different forest cover type and are labelled from '1' to '5'. The data was centred and scaled by subtracting the mean and dividing by the standard deviation of each features so that no feature is dominant over the other during the clustering process (SOM relies on Euclidean distance between the input variables and the neuron weights).

Different configurations were tested out for SOM. The Adjusted Random index (ARI) was chosen as a criterion to measure the performance of each configuration. ARI gives us a good estimate of the measure of the similarity of the datapoints presents in the clusters and the actual clustered data (true labels). The chosen configuration for SOM training is as follows: 100 neurons (10x10 grid) with a hexagonal lattice and training with 100 epochs. The ARI for this configuration was 0.64.

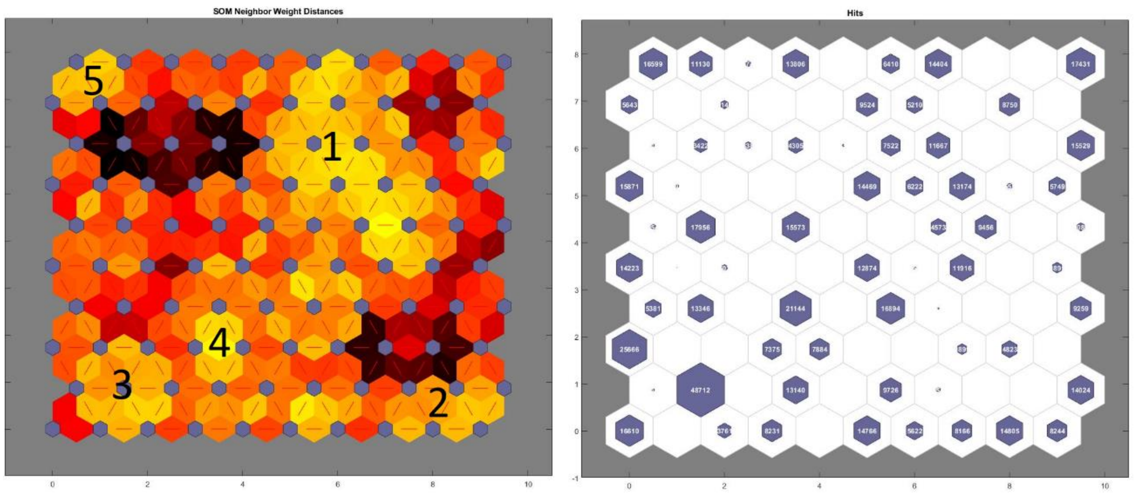


Figure 17: *SOM neighbor weight distance (left) and number of hits (right) for "covertype" dataset*

We can visualize the SOM neighbor distances (U-matrix) in **Figure 17**. This shows us the distances between neighboring map units where the dark cells represent a large distance between adjacent cell units, so they indicate a cluster border. Whilst elements closely packed are represented by light colored and those are generally the defined clusters. We expect to visualize 5 distinct clusters since we know that our datapoints are classified in 5 classes. We can visualize those clusters in our U-matrix (numbered in **Figure 17**). It is worth to note that our dataset's classes are not represented equally: label 1 and label 2 account for about 85% of the dataset which might have led to some classes being more over-represented than others on the SOM mapping.

The 'numbers of hit' is a histogram representation of the dataset on the map. It shows the number of vectors classified for each neuron. We can see that the clustered regions are represented by a high number of vector assignment. The borders lack any neuron assignment to them, but they are useful for forming cluster borders. We also plotted the weights per input vector (not shown here) which gives us information about the influence of each feature for the classification. Overall, we could see that a lot of inputs could have been disregarded in favor of a faster computation time since they offer no contribution in the clustering process.

Assignment 4: Autoencoders and Convolutional Neural Networks

4.1) Autoencoders

An Autoencoder (AE) is a NN in which the network output is a close approximation of the input. The aim is to uncover interesting structure about the data: highly correlated features can be expressed in terms of lower dimensions. This is of importance when training an image classifier consisting of high dimensional data to significantly decrease processing time.

The architecture of an AE consists three parts: an encoder (data compression), a decoder (data reconstruction), and a bottleneck region (hidden layer) in the network that contains far less neurons than the input dimension. That way the network is able represent the data in far less dimensions that than the ones expressed in the input. This also ensures that the network does not simply learn to 'memorize the data'. What is obtained in this layer is a latent representation of the input that contains the most discriminative features of the data. In general, an AE network is built from many autoencoders stacked with each other ('Stacked Autoencoder') in which the output of each layer acts as an input to another layer.

AE bear a lot of similarities with PCA. In PCA, high dimensional correlated data are expressed in terms of orthogonal independent components of lower dimensions. PCA generally works when there is a linear relationship between the data since it is only capable of projection the data unto a line or a hyperplane. With Autoencoders, non-linear relationships can be learned: the lower dimension representation of the data in the latent space are projected onto a manifold thanks to the non-linear activation functions of the hidden layers. In fact, an AE with linear activation functions will behave similarly to PCA where the number of hidden layers in the bottleneck region is equivalent to the number of PC's chosen to project the data unto.

The classical methods for training a stacked autoencoders is through a greedy layer-wise training in which the first AE is trained on the raw input and then the weights and biases are adjusted accordingly. The output from the first layer is used as an input for the second layer to adjust the weights and biases of that layer. The process is repeated for all the layers. In the end, all weights and biases are updated by a process called 'finetuning' by backpropagating the gradient of the loss function.

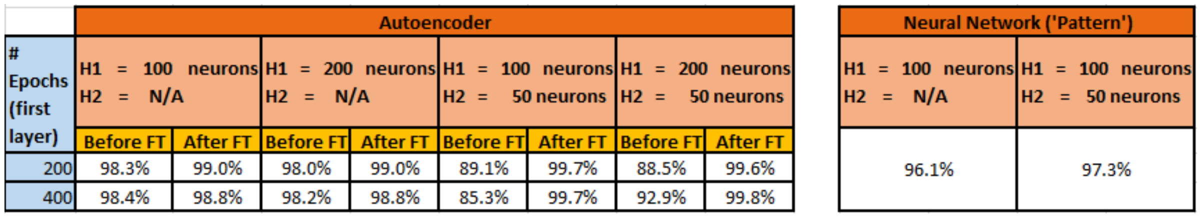
An ideal AE must be able to accurately reconstruct the trained without memorizing it (overfitting) in order to generalize well on new unseen data. Hence, we need to apply regularization that forces the network to only activate neurons that correspond to relevant feature input. In that way, we obtain what we call a 'sparse autoencoder'. The regularization term in the loss function will penalize the activations of the neurons instead of the weights themselves. In general, there are two regularization terms that are to be included when training an Autoencoder:

1) L1 norm regularization: Also known as lasso regression, it penalized the absolute value of the sum of activation in a given hidden layer. This way, neurons with small activations will have zero

activation functions. The aim is to reduce the complexity of the network and obtain a sparse representation.

2) KL divergence: KL is employed to measure the deviation of distribution of two given samples. In our case, we will choose a sparsity parameter ρ which is the average activation value for a collection of samples. Neurons that have an activation that largely deviates from our chosen ρ will be penalized. Thus, we constrain the activation of a neuron for a collection of inputs obtain a sparse representation of our Autoencoder.

In our example, we will use autoencoders to try to classify a set of number images (MNIST dataset). We tried training the for different number of hidden layers and different hyperparameter configuration. The output of the last layer of the AE is inputted to a "softmax" layer to classify the obtained latent representation in a supervised fashion. After training each layer separately, the layers were stacked up and finetuning was performed on the network to optimize the network and obtain a more accurate reconstruction. In addition, the results were compared with that of a normal NN (accuracy). For each different configuration, the results were averaged out over 5 different runs in order to obtain a better generalization of the results since the weights are randomly initialized at every run. The results of the training can be visualized in **Table 7.**



	Autoencoder							
# Epochs (first layer)	H1 = 100 neurons H2 = N/A		H1 = 200 neurons H2 = N/A		H1 = 100 neurons H2 = 50 neurons		H1 = 200 neurons H2 = 50 neurons	
	Before FT	After FT	Before FT	After FT	Before FT	After FT	Before FT	After FT
200	98.3%	99.0%	98.0%	99.0%	89.1%	99.7%	88.5%	99.6%
400	98.4%	98.8%	98.2%	98.8%	85.3%	99.7%	92.9%	99.8%

Neural Network ('Pattern')	
H1 = 100 neurons H2 = N/A	H1 = 100 neurons H2 = 50 neurons
96.1%	97.3%

Table 7: *Accuracy of classification of MNIST dataset for an AE (left) and a Neural network (right) with different configurations*

The default options provided in the example consisted of two hidden layers: 100 and 50 neurons in the first and second layers respectively with an accuracy of 99.7% after fine-tuning. We managed to obtain a slightly -albeit small- improvement when doubling the number of neurons in the first hidden layer (an improvement of 0.1% for a total accuracy of 99.8%).

Fine-tuning increased the overall accuracy for all different configuration of Autoencoders. When comparing the accuracy of Autoencoders with one hidden layer, we can see that the Autoencoders with two hidden layers had a better overall accuracy after fine tuning but a lower one before tuning: an AE with two hidden layers has a more complex architecture thus is more prone to overfitting. Fine-tuning managed to effectively regularize the network by enforcing the network to reach a global optimum solution. This is less evident in an AE with one hidden layer since it is less succeptible to overfitting (fine tuning sill manages to improve the accuracy, but to a lesser extent to that of AE with two layers).

In all cases after fine tuning, the AE outperformed the conventional NN in terms of accuracy. A NN tries to predict an output based on a set of features, in this case we were comparing the set of inputted images to the labels. On the other hand, an AE network compares the input function with itself. This allows us to extract more latent features of the images that are more discriminative of the actual data **(Figure 18)**.

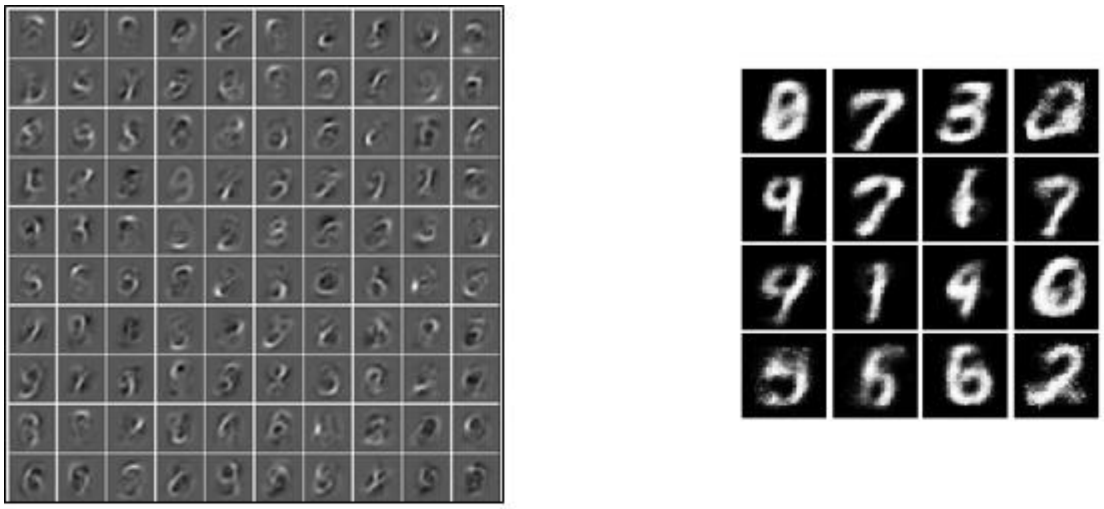


Figure 18: *Latent representation of the Autoencoder features for the MNIST dataset (left). Generated numbers using variational autoencoders (right)*

4.2) Variational Autoencoders

Let's assume that we want to use an AE for content generation, when randomly sampling from each latent attribute. There is a high probability that we will obtain a meaningless representation once the sample is decoded since we have a punctual, explicit and discontinued distribution of attributes in the latent space. The goal with VAEs is to regularize the latent space in order to obtain a continuous and complete distribution of attributes in the latent space. This regularization encourages each latent space attribute to have a standard normal distribution so that the distributions overlap, and no attributes dominate other during sampling. This way, we can obtain a latent variable model. Randomly sampling from this distribution will lead to an exploitable and interpretable content that can be used for content generation.

It can be seen from **Figure 18** that the generated numbers are somehow blurry, this is generally associated with the tradeoff between the reconstruction error and the KL-divergence between the prior and learned distribution.

The main similarities between both AE and VAE is that they both try to reduce the number of dimensions of the input data. The bottleneck region encourages the data to be expressed in terms of the latent features that are the most salient and discriminative of the given data. Both networks have a similar architecture in that they both contain an encoder and a decoder. However, they differ slightly in the bottleneck region. The main differences between the two networks as well as the optimizers employed in each one of them are outlined in the tables below:

	Autoencoder	Variational Autoencoder
Purpose	Learns to reconstruct original data (classification, noise reduction and anomally detection)	Learn to generate new data
Bottlneck region	1-D layer that contain neurons equivalent to the number of latent representations	2-D layer where one layer represent the mean and the other represent the variance of the point that is sampled from the latent distribution
Latent space	Latent representation	Latent distrubtion (normally distributed)
Sampling	Random sampling is unlikely to lead to an interpretable content when decoded	Randomly sampling from each latent attribute leads to a content with mixed attributes when decoded (requires re-parametrization trick)
Reconstruction error	Minimize reconstruction loss between estimated input and original input	Maximize the reconstruction likelihood (binary cross entropy)
Regularization	Enforces a sparsity constraint by deactivating redundant neurons (avoid memorizing the training data and overfitting)	Encourages the learned distribution P(z/x) to have a distribution similar to a standard normal distribution (ensure that sampling from latent space enables a generative process)
Output (MNIST)	Reconstructed, noisless image	Blurry generated image

Table 8: *Differences between an Autoencoder (AE) and a Variational Autoencoder (VAE)*

	Scaled conjugate gradient (AE)	ADAM optimizer (VAE)
Description	- Searches for directions along the conjugate direction as opposed to the direction in which the function decreases more rapidly.	-Extended version of stochastic gradient descent -it updates the weights iteratively based on first and second momentum. -Contains decayng learning rates
Advantages	- It can be considered a good algorithm for large networks as it does not require a lot of memory	-Little memory requirement -Well suited for problems that are large in terms of data or parameters -Appropiate for non-stationary objectives -Appropiate for problems with noisy or sparse gradients -Little tuning required for hyperparameters -Seperate learning rate for each parameter
Disadvantages	- Learning rate is not adapted per parameter (single learning rate across all parameters) -Requires careful hyperparameter tuning -May take long time to converge	- May not work for some very non-stationary distributions

Table 9: *Differences between the optimizers employed in AE and VAE*

<u>**4.3) Convolutional Neural Networks (CNN)**</u>

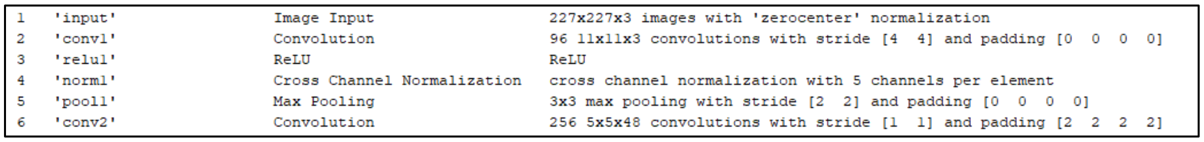
The traditional MLP architecture doesn't scale well for image classification problems. Let's imaging we have a 32x32x3 images. We will have 3072 weights in total that will connect one neuron in the first layer. Scaling to the full size of the network, this amount of weights will increase exponentially which translates to more computational processing time and a potential for overfitting.

Convolutional Neural Networks (CNN) were designed to answer for the limitations of MLP for image classification. For this exercise session we will train "CoveNet" CNN which was trained to classify 1000 images into different categories. The network consists of multiple repeated layers of which the functional units are as follows (not in order):

1) **<u>The input layer:</u>** The input images that are to be processed by the network, images are scaled to a 227x227x3 size before any processing occurs.
2) **<u>Convolutional layer:</u>** Output of neurons connected to a local region in the input ("Receptive field"). The output volume of the layer depends on the number of filters (kernels) used. The first convolutional layers generally identify low level features. The identified features increase in complexity as we go deeper in the network.
3) **<u>RELU:</u>** elementwise activation function which outputs only positive values. It is employed in CNN since it achieves a fast convergence for gradient descent (accelerated learning).
4) **<u>Cross channel normalization:</u>** Diminishes responses that are relatively large for a given set of channels in order to make local responses more pronounced.
5) **<u>Pool layer:</u>** Down samples the output of the convolutional layer along the spatial direction. This dimensionality reduction helps reduce the number of parameters to be computed. In addition, it the mapped feature after down sampling makes the network more translationally invariant.
6) **<u>Fully connected layer (FCL):</u>** contains number of neurons equivalent to the number of classes. Class score probabilities are extracted from the highest-level features using a 'Softmax' activation function.

Below are the answers to the questions asked in the report:

i. The weights between the input and the convolutional layer are of the dimension 11x11x3x96. This dimension is related to the receptive field size of the used filter (F=11), the volume of the input (3 channels) and the number of filters used (K=96) which defines the volume of the first convolutional layer. Unlike an MLP, the weight is independent of the input and governed only by the number and size of the filters. The weights here are filter values of each filter which are randomized initially and the get updates during the training process.



```
1   'input'    Image Input                    227x227x3 images with 'zerocenter' normalization
2   'conv1'    Convolution                    96 11x11x3 convolutions with stride [4  4] and padding [0  0  0  0]
3   'relu1'    ReLU                           ReLU
4   'norm1'    Cross Channel Normalization    cross channel normalization with 5 channels per element
5   'pool1'    Max Pooling                    3x3 max pooling with stride [2  2] and padding [0  0  0  0]
6   'conv2'    Convolution                    256 5x5x48 convolutions with stride [1  1] and padding [2  2  2  2]
```

Table 10: *First 6 layers of the "CoveNet" architecture*

ii. We will use the following formula to estimate the size of the output layer:

$$Spatial\ size\ of\ output\ volune = \frac{W - F + 2P}{S} + 1 \quad (5)$$

<u>Where:</u> W is the input size, F is the receptive field size, P is the number of zero padding and S is the stride

Using the above formula, we estimated that the spatial size of the first convolutional layer is 55x55x96. The Pooling layer acts as a dimensionality reduction with a stride of [2 2] and a kernel of 3x3 **(Table 10).** Inputting the previous values and the obtained spatial size into (1) we obtain the dimensions of the input of layer 6 (27x27x96). The pooling layer reduced the spatial dimension by more than a half but does not affect the volume.

iii. The final number of neurons that are used for the classification is 1000 which is equivalent to the number of classes that the neurons can predict. The final output layer before the FCL is of 4096, those are the final high dimensional features that are each connected to the classification neurons and will determine the final output of the classification. There is a significant reduction of dimensions compared to the initial image input from 154,587 (227x227x3) to 4096.

In this example, we are going to train different architectures of CNN for classification of hand-written digits, we were given 1000 images. The data was partitioned as follows: 750 data points for training set and 250 for testing set. The results of the training can be visualized in **Table 11**.

CNN architecture: Input → Conv layer 1 (5x5 kernel) → Relu Layer → max pooling (2x2 with stride of 2) → Conv layer 2 (5x5 kernel) → Relu layer → FCL (10 neurons) → softmax layer

# filters (conv layer 1)	**12**	**12**	**24**	**24**	**48**	**48**	**48**
# filters (conv layer 2)	**12**	**24**	**12**	**24**	**12**	**24**	**48**
Time (s)	259	247	364	352	520	553	570
Accuracy	15%	82%	26%	89%	10%	89%	89%

Table 11: *Training of CNN for hand-digit classifications for different number of filters in the convolutional layers*

Reducing the number of filters in the second layer dramatically decreases the overall accuracy: more high-level features need to be derived in order to obtain discriminatory predictive features.