KU LEUVEN

Data Mining and Neural Network (G9X29A)
Final Assignment Report, January 2015

Student: Dafydd James
Student Number: R0479799

January 11, 2016

Contents

Exercise 1 3
1.1 Function Approximation (noiseless case) 3
1.2 The role of the hidden layer and output layer 4
1.2.1 4
1.2.2 and 1.2.3 4
1.2.4 and 1.2.5 4
1.2.6 5
1.2.7 6
1.3 Function Approximation (noisy case) 6
1.4 Curse of dimensionality 11
1.4.1 One dimensional case 11
1.4.2 Two Dimensional case 12
1.4.3 Five dimensional case 13
Exercise 2 13
2.1 Santa Fe laser data – time-series prediction 13
2.2 Alphabet recognition 17
2.3 Pima Indians Diabetes – classification problem 19
Exercise 3 21
3.1 Dimensionality reduction by PCA analysis 21
3.2 Input selection by Automatic Relevance Determination (ARD) 22
4 Appendix- Matlab Code 24

Exercise 1

1.1 Function Approximation (noiseless case)

For simple monotonic functions there is no need for a hidden unit (note that one hidden unit is the same as no hidden units) as any network is capable of approximating any continuous function given a particular activation function. The addition of a hidden unit wouldn't change anything, while adding more than 1 would lead to unnecessary oscillations due to overfitting, increasing the variance and in turn the error. A more complex function (more inflection points in this case) will need a larger number of hidden units to increase the degrees of freedom (flexibility) in the model. Trying a number of difficult combinations, it becomes clear that when having a number of hidden units equal to the number of inflection points, a perfect fit can be obtained, although this is not found every time due to the many local minima, and its accuracy depends on the initial weights and biases applied. This comes as no surprise due to the fact the adding a hidden unit leads to the same result as adding 1 to the order of polynomial, which adds another inflection point. However in reality the optimal number of neurons is not always the best, as the network generally struggles to train (easier if no noise).

The inclusion of more hidden units doesn't visually appear to make any difference (as long as not an extreme addition) due to the function being noiseless, however in general a model should be penalised for its complexity (Occam's Razor for nonlinear models), which would lead to choosing the minimum number of hidden units, therefore reducing the number of effective parameters. If overfitting is present when trying to minimise the objective function the interconnection weights will be optimised towards a local minimum that represents the training data (will model random noise if present), the more hidden units added the more of a problem these false local minima become. Running an independent test set against this will allow one to see where this the model stops estimating the underlying function and starts estimating random noise.

Having less hidden units than inflection points, will lead to a model that is never be capable of accurately predicting the function as there is not enough effective parameters to allow the model to fluctuate sufficiently. In this case the variance will be low, but at the expense of large bias. In Figure 1.1 and 1.2 two of the extreme samples are shown. In Figure 1.1 there is clear overfitting as we have 9 hidden units and no inflection points. This gives a perfect fit to the training set, but at the expense of large oscillations. This therefore is a poor representation of the actual function. In Figure 1.2 there is clear under-fitting as the model no flexibility to estimate the complex nature of the function. So, to recap, in the case of a noiseless model, a perfect fit can be obtained by allowing for a number of hidden units equal to the number of inflection points, and this is the optimal option in every case, although it can be difficult to due to the many local minima.

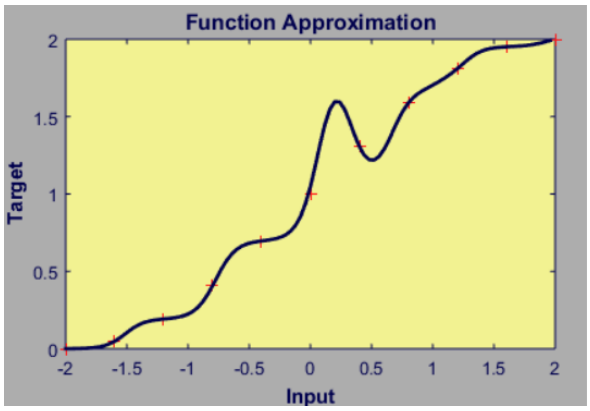

Figure 1.1: 9 hidden units

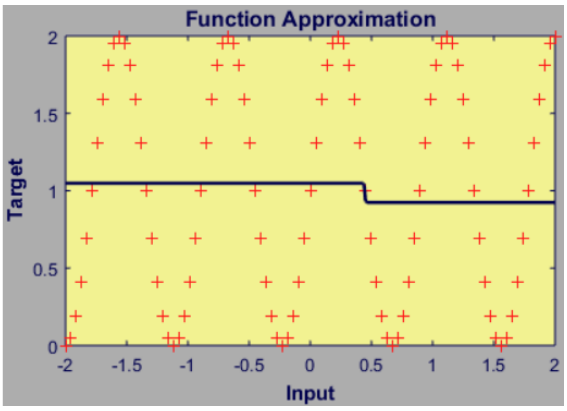

Figure 1.2: 1(0) hidden units

1.2 The role of the hidden layer and output layer

1.2.1

Given the linear equation represented by

$$y_p = w_1 x_{1,p} + w_2 x_{2,p} + \ldots + w_n x_{n,p} + \beta$$

to train a network to perform linear regression, it is not necessary to include any hidden layers. As stated in part 1.1, any continuous function can be measured without any hidden units, and hence the network works sufficiently with just the outputs and inputs. For defining layers, it is common practice not to consider the inputs as a layer, so we will not consider it one when referring to number of layers for the remainder of this report. Given the above comments the number of layers required to learn this function is 1 (an output layer). The number of neurons in this layer is simply the 1 output node. And the transfer function (activation function) used for this node is the linear activation function as it has an infinite range and models the linear relation accurately.

1.2.2 and 1.2.3

Creating a vector of 21 input patterns that are equally spaced on the closed interval [0, 1] we create output patterns of the form:

$$y_p = \sin(0.7\pi x_p)$$

In Figure 1.3 a plot of the relationship of the input and output pattern is shown. This graph show a non-linear function, and hence modelling it using a linear model would not adequately capture the relationship between the input and output patterns. Using the architecture of the network chosen for part 1.2.1 would also give a model that is not capable of capturing the relationship, as the network chosen can only be used to approximate a monotonic function, which is not present here. To estimate a non-linear function a hidden layer with two neurons will need to be added to the architecture of the network.

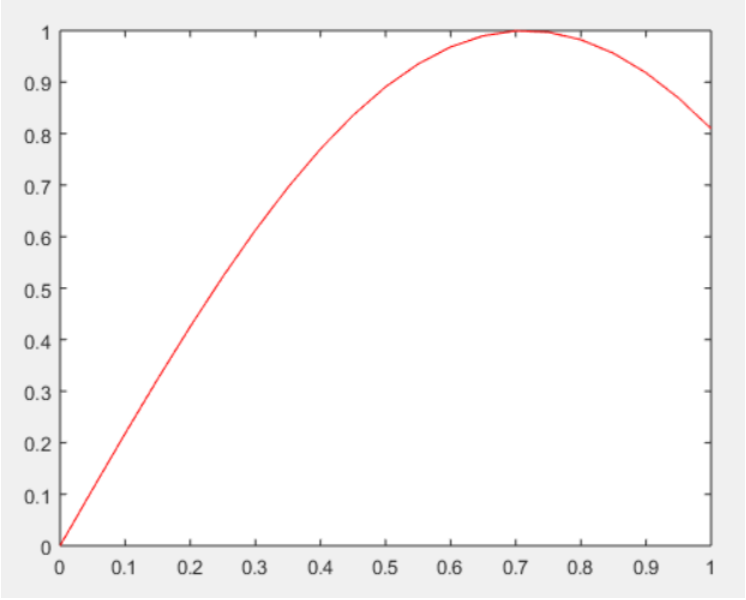

Figure 1.3: Plot of function $y_p = \sin(0.7\pi x_p)$ vs x

1.2.4 and 1.2.5

Using the code given, which has a network with one hidden layer containing two hidden nodes (as suggested in part 1.2.3, we obtain a model for approximating the function. This model approximates

the function perfectly, as the errors are negligible and the r-squared value is 1 for the training, validation and test set. When looking into the make-up of this model we obtain values for the biases (1.8538, 0.0888) and interconnection weights (-1.4607, -1.2946), along with their transfer function (found to be tansig) for each the hidden units. Using these values the activations of the hidden units are in the form

$$x_p^1 = \text{tansig}(-1.4607(x) + 1.8538)$$

$$x_p^2 = \text{tansig}(-1.2946(x) + 0.0888)$$

For the first and second hidden neuron respectively. In Table 1.1 the activation for each input is given for both hidden units.

X values	Activation 1	Activation 2
0.00	0.9521	0.0886
0.05	0.9448	0.0240
0.10	0.9364	-0.0407
0.15	0.9267	-0.1050
0.20	0.9157	-0.1685
0.25	0.9031	-0.2306
0.30	0.8887	-0.2910
0.35	0.8723	-0.3490
0.40	0.8537	-0.4045
0.45	0.8326	-0.4572
0.50	0.8088	-0.5069
0.55	0.7820	-0.5534
0.60	0.7519	-0.5967
0.65	0.7184	-0.6368
0.70	0.6812	-0.6737
0.75	0.6401	-0.7075
0.80	0.5949	-0.7384
0.85	0.5457	-0.7664
0.90	0.4924	-0.7919
0.95	0.4351	-0.8148
1.00	0.3740	-0.8354

Table 1: Activations

1.2.6

In Figure 1.4 a graph shows the activation functions for each of the hidden units (x_p^1 and x_p^2) along with the output (y_p). To model the relationship between y_p and (x_p^1, x_p^2), we need to obtain the interconnection weights between the hidden units and the output, the bias at the output and the activation function for the output. The value of y_p can then be obtained using the formula

$$y_p = \text{activation funcion}\left(weight1\left(x_p^1\right) + weight2(x_p^2) + bias\right)$$

where $weight1$ is the weight of the first hidden neuron to the output and $weight2$ is the weight of the second hidden neuron to the output.

1.2.7

Using Matlab functions the interconnection weights of the hidden neurons to the output are given as 1.6097 and -1.8834 for neuron 1 and 2 respectively, the bias for the output node is found to be -1.3658 and the activation function is found to be linear. The relationship between y_p and (x_p^1, x_p^2) can now be modelled by

$$y_{out} = \text{purelin}\left(1.6097\left(x_p^1\right) - 1.8834\left(x_p^1\right) - 1.3658\right)$$

In Figure 1.4 it can be seen that y_{out} replicates the actual function yp perfectly just as in stated in part 1.2.4 where the errors were miniscule and the r-squared value was 1.

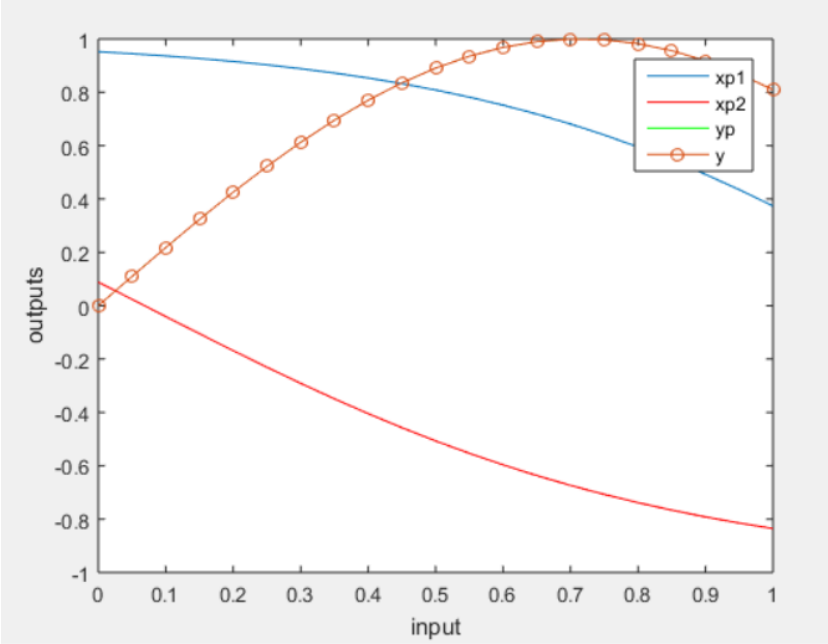

Figure 1.4: Graph of activations, output, and neural networks calculated output

1.3 Function Approximation (noisy case)

Knowing that a sinusoidal function changes direction at intervals of π radians it is possible to estimate the amount of neurons needed to estimate the function sufficiently. Generating data from -1 to 1 using a sinusoidal function will give inflection points at intervals of every 0.5, beginning at -.75, as sin (-2 π) is 0 which is a midpoint. Given said statements it is clear that there will be 4 inflection points between -1 and 1. As stated in part 1.1, the number of hidden units necessary to accurately measure the true underlying function is equal to the number of inflection points, therefore before running any models, it is already possible to say that the ideal number of hidden neurons is equal to 4. However finding the global minima to ensure this is found every time the analysis is run, may be quite difficult.

Size of training data – It is already common knowledge that the larger the dataset is, the more accurate the model will be. This is due to the random error being averaged out more for large datasets than small datasets, along with the large number of observations being capable of accurately piecing together the actual function (a complex function with many turns and curves, cannot be measured accurately by 4 observations for instance). As well as this a large number of observations have the

benefit of allowing for sufficiently sized datasets for training, validation and test reasons, which allows for better generalisation (avoids modelling noise) in the model. As an example (sd =0.5 and number of hidden neurons =4), 3 choices of training data size will we chosen, 50, 200 and 1000. Here the r-squared value of each was found to be 0.80847, 0.81861 and 0.8309 for 50, 200 and 100 respectively, and the validation sets performance (mean squared error (MSE)) to be 0.26914, 0.263, and 0.24497 similarly. This shows that a higher number of observations does in fact generate a more accurate model up to a point where there is no longer any bias in the curve and the only error is due the random variation. This can also be seen graphically in Figure 1.5, as networks produce a better replication of the function with a higher number of observations. It might be worth noting however that larger datasets do take longer to calculate, and in certain cases this could become overly time consuming.

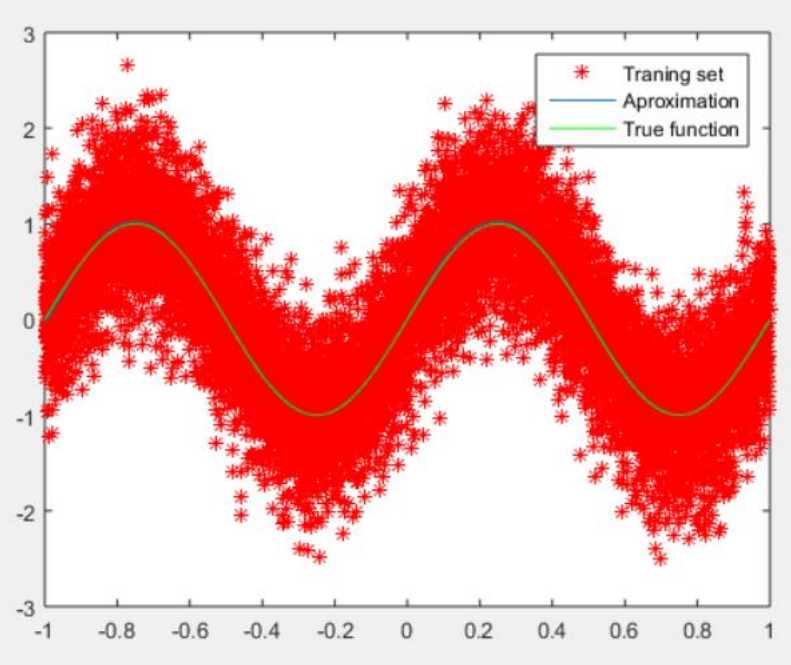

Figure 1.5: 1000 observations

Choice of standard deviation – Standard deviation measures how varied the observations are. Higher values of this will lead to larger deviations from the true underlying, while a standard deviation of 0 will model with all observations following the same function (fit the underlying function perfectly in this case). As 50 training observations seemed quite sufficient for estimating the function, this will be used, along with hidden neurons =4, for testing different values of standard deviation. Using sd equal to 0, 0.5 and 1 we obtain r-squared values equal to 1, 0.80847 and 0.37478, and a MSE equal to 2.7 x 10^{-8}, 0.263 and 1.387 for 0, 0.5 and 1 respectively. This shows that models with lower standard deviations are easier to estimate accurately. In Figure 1.5, 1.6 and 1.7 the differences can be seen graphically.

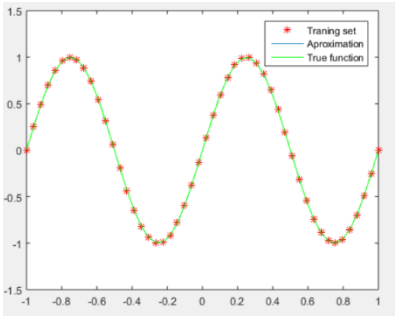

Figure 1.5: sd =0

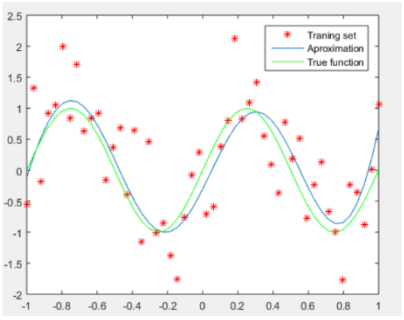

Figure 1.6: sd =0.5

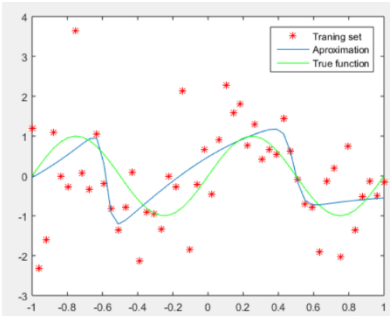

Figure 1.7: sd =1

<u>Number of hidden units</u> – As has already been stated, 4 is a sufficient number of hidden units to accurately estimate the model in question, as has been seen when using a standard deviation of 0, but for argumentative reasons, different numbers of neurons will be considered. Using a sd of 0.5 and number of training observations equal to 50, choices of 1,4, 5 and 9 neurons will be analysed. Here the values of r-squared and MSE are 0.44, 0.856, 0.866 and 0.857, and 0.57, 0.249, 0.242 and 0.29 respectively for each of the choices. It can be seen in Figure 1.8 and 1.9 that allowing for one neuron greatly lacks the flexibility to accurately estimate the true function, while 9 neurons oscillates unnecessarily and begins measuring random errors (which is obviously not ideal). In Figure 1.10 and 1.11 then model more accurately models the underlying function. It is interesting to note that the model with 5 hidden units actually measures the model better (higher r-squared and lower MSE). As stated in part 1.1 the network with the optimal number of hidden units struggled to train the data with noise, and hence ended up with worse results. To recap, 4 neurons is the appropriate measure the underlying function (only because we know the underlying function, in most real life cases this will not be the case), but in cases of high standard deviation, the network can struggle to train this and it may be more appropriate to use 5 neurons.

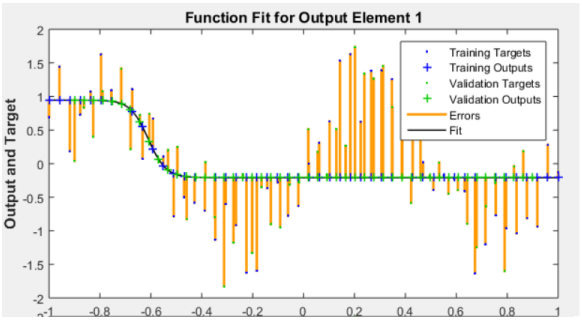

Figure 1.8: 1 hidden unit

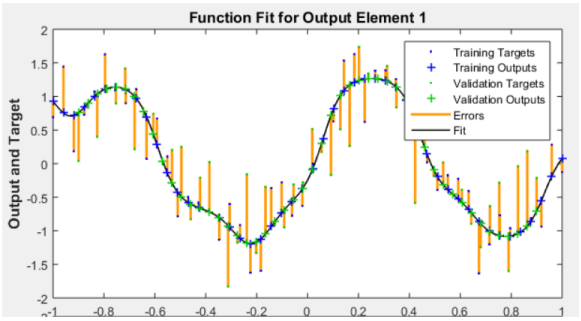

Figure 1.9: 9 hidden units

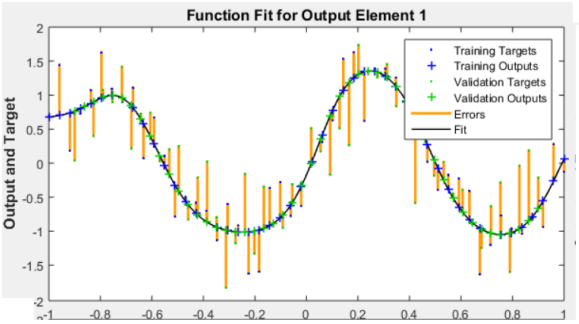

Figure 1.10: 5 hidden unit

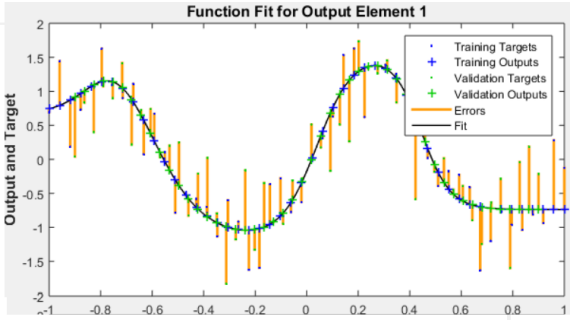

Figure 1.11: 4 hidden units

Training algorithms – There are many different training algorithms, each with their own pros and cons. The simplest optimisation algorithm is the steepest descent algorithm, however it is quite basic and there are more advanced and quicker algorithms. Levenberg-Marquardt is the most common method, with quick and reliable results for simple models, but can have a very slow convergence rate for more complex models. This has been used for all the calculations above. The Quasi-Newton method offers a more sophisticated method for exploiting the gradient information, but is quite computationally expensive leading to longer programming execution times. The above methods are known as backpropagation methods. The backpropagation methods offer a simple and efficient way of modelling networks. Unfortunately when the networks contain many interconnection weights, these methods struggle to store the matrix information and hence cannot handle large scale networks. For large scale networks one can use conjugate gradient methods. This method is also very fast. Bayesian regularisation methods are very robust and do not require a validation set to estimate the model, however they are very complex and often very time consuming.

Using 50 observations, a standard error of 0.5 and 4 neurons, as we have tended to in report thus far, the training algorithms are compared. The values given in Table 1.2 show that the Levenberg-Marquardt method is the best in every aspect (converges quicker and is more accurate). This was as expected due to the simple nature of the problem. If the number of observations, neural networks and standard deviation was increased, this could lead to different algorithms being more useful.

Levenberg-Marquardt	BFG Quasi Newton	Gradient Descent	Scaled Conjugate Gradient	Bayesian Regularisation	Resilient Back-Propagation
00:12	00:28	00:29	00:12	00:56	00:15
0.265	0.37	0.54	0.46	0.58	0.45
0.81	0.68	0.37	0.57	0.36	0.58

Table 1.2: Training Algorithms performance

Training until a local minimum is reached vs early stopping on a validation set – So far the method of early stopping on a validation set has been used, where the network stops training after 6 non-decreasing iterations. The stopping criterion can be changed to stop after any amount of non-increasing iterations. By stopping after a certain amount of iterations it is possible to halt the training before it starts measuring noise and stops estimating the underlying function. This leads to the model having a good generalisation. This method makes use of the validation set, however if we want to ignore the validation set it is possible to stop the training when a local minimum is reached. To do this we set a stopping criteria at a certain gradient, so when that gradient has been reached, the training will stop. This method does not consider the validation set in any way. To compare the two methods 50 observations, sd =1, hidden neurons = 4 and the Levenberg-Marquardt training algorithm will be used. The early stopping on a validation set will remain set at 6 non-decreasing iterations, while the gradient stopping criteria will be set at 0.001. The MSE is 0.251 and 0.256, while the r-squared is 0.82 and 0.82 for early stopping on validation and gradient descent without the validation set respectively. These both show very similar values although the gradient method did take a lot longer to complete. In general it is wise to consider the training set, to ensure what is being measured is not random error and is generalised, however using the gradient stopping criteria does allow for a greater amount of data to be used in evaluation.

With or without regularisation – Regularisation can help remove the flexibility of a model. For example if there is large overfitting in a model, a regularisation term can reduce this. This method

reduces the numbers of effective parameters by removing any that have an Eigen value less than a given constant (regularisation constant). This allows for the model to select an appropriate number of inflection points (just as the hidden neurons). This can be very useful if the structure of the function is not known. One should try many values of regularisation constants and opt for the value that give the best results. For an example here, regularisation constants of 0 (no regularisation), 0.1, 0.01 and 0.001 will be considered on a network of 9 neurons (with other conditions the same as examples above). The MSE and r-squared are 0.273 and 0.83, 0.437 and 0.69, 0.285 and 0.81, 0.262 and 0.85 for regularisation constants of 0, 0.1, 0.01 and 0.001 respectively. This shows that using a regularisation constant of 0.001 is the best predictor as it generalises the model well. No regularisation allows for too much variance in the model, while too high a regularisation term allows for too much bias. Graphically in Figure 1.12, 1.13, 1.14 and 1.15 it can observed that a regularisation constant of 0.001 most accurately resembles a sine function, while others are either over or under fitting. It should be noted that a regularisation term is linked to the number of neurons chosen, and the term can never increase flexibility.

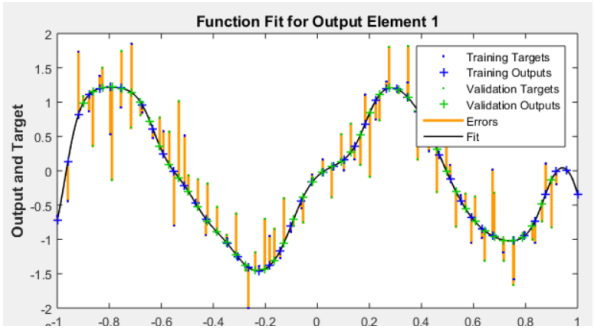

Figure 1.12: Regularisation constant = 0

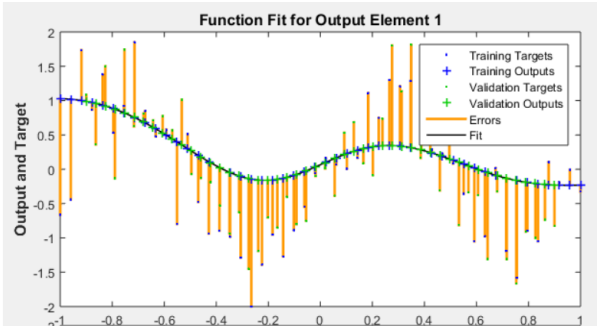

Figure 1.13: Regularisation constant = 0.1

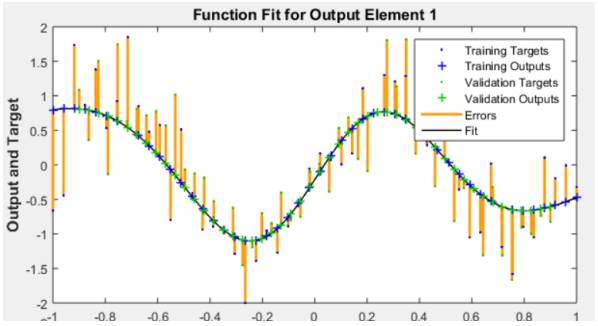

Figure 1.14: Regularisation constant = 0.01

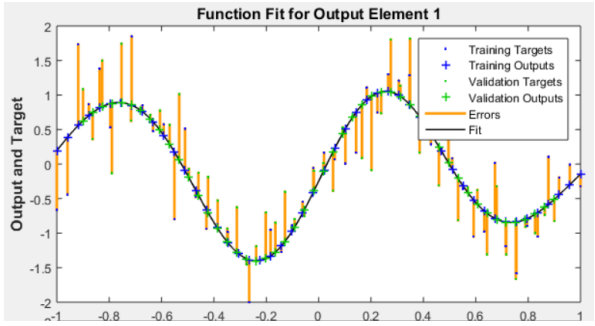

Figure 1.15: Regularisation constant = 0.001

<u>Choice of initial weights</u> – Choosing initial weights can ensure that the global minima is reached and not the local minima. There are many local minima solutions when training neural networks. One will want to avoid poor local minima solutions that give poor estimates. To do this one should apply small initial interconnection weights. Testing the data with 50 observation in the training set, 4 hidden neurons and an SD of 0.5, like above, we choose different initial weights to test. The choices are zero initial weights, random weights between -1 and 1, the initwb method (Nyguyen-Widrow initialization algorithm) and the initwb method In Mathlab. The resulting r-squared, MSE and time values are given in table 1.3. Here we can see that the best performance was given by the using the initwb function which is default method in Matlab for the Levenberg-Marquardt in Matlab.

Initial zero	Initial wb	Initial nw	Initial random
00:30	00:15	00:17	00:20

0.263	0.36	0.24	0.38
0.83	0.67	0.84	0.64

Table 1.3: Initial weights

Note: If not stated the data has been processed using 50 training observations, 50 validation observations, 4 hidden neurons, standard error of 0.5, the Levenberg-Marquardt algorithm, early stopping on a validation set and initial weights found using the 'initnw' method. No test set was used on this data, which reduces the generalisation of the MSE and r-squared that was obtained. Although the r-squared and MSE basically represent how well the data is modelled, we went with both as MSE is the comparison technique, but r-squared can tell us how good the model is, and not what is just the best. One could have used a cross validation technique here, but the whole thing was time consuming, so was neglected.

1.4 Curse of dimensionality

The curse is dimensionality is related to the problems that occur due high dimensional spaces. These high dimensional spaces cause the data to become very sparse, and due to this sparsity it can become difficult to evaluate statistical significance or classify variables. The data necessary to do so grows exponentially which leads to the model requiring large processing power. By using neural networks instead of polynomial expansion it becomes possible to approximate the error independently of the dimension. That is the approximation for MLPs with one hidden layer is of the order $\mathcal{O}(1/n_h)$, while for polynomial expansion it is $\mathcal{O}(1/\mathrm{n}_\mathrm{p}^{2/n})$, where n_h is the number of hidden neurons, n is the number of dimensions and n_p is the number of terms in the expansion. In the case of neural networks the number of interconnection weights grows less dramatically than that of the parameters involved in polynomial expansions.

Throughout this question, an analysis will be run where we hope to find a good approximation of the sinc function using differing numbers of dimensions. The sinc function will be analysed in 1 dimension, 2 dimensions and 5 dimensions. As the number of dimensions increase, the data points will also. If we are to analyse every dimension equally the amount of data points will be equal to n^d, where n is the number of observations in one dimension and d is the number of dimensions. It is obvious for this reason that a large number of observations cannot be considered as the number of data points would become very large.

1.4.1 One dimensional case

To start it is of interest to see how many inflection points the function will have, which allows for the optimal choice of hidden neurons. In Figure 1.16 we can observe 9 inflection points between -5 and 5, meaning the optimal number on units (especially considering the noiseless nature of the approximation) will be 9 for the 1 dimensional case.

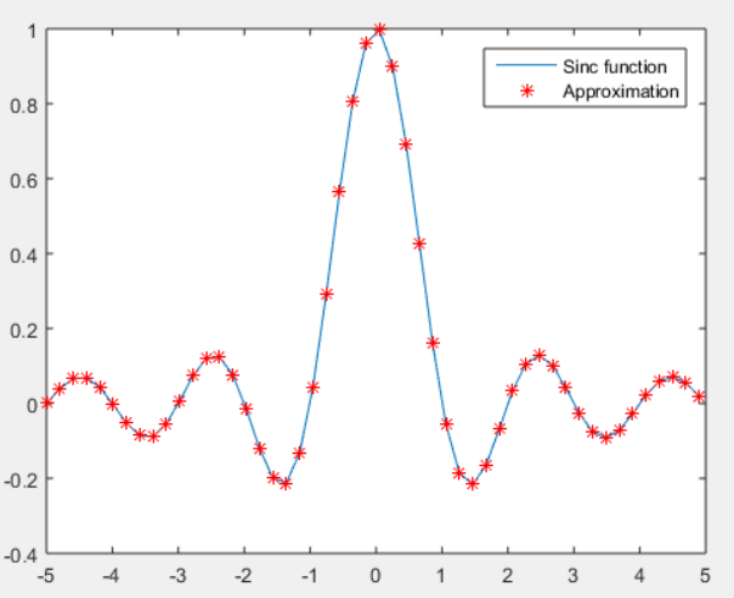

Figure 1.16: Sinc function and approximation in 1 dimension

100 observations (50 applied to a training set, and 25 to both validation and test set) are applied for estimation purposes. Using these observations with the 9 neurons it is possible to find a perfect estimation of the underlying function (as expected due to absence of noise). The test set, validation set and training set all have r-squared values of 1, showing perfect estimation. In Figure 1.16 it is clear that the approximation perfectly measures the sinc function.

1.4.2 Two Dimensional case

For the 2-dimensional case, 50 observations in each dimension will be applied (as I didn't have the patience to wait for the calculations), leading to a 50^2 data points. These were split into training, validation and test sets in the same ratio as the one dimensional part of the question. As there is no noise here, using slightly too many hidden neurons will not overestimate the function too greatly (as long as it's not too far over). However to keep these observations at bay, a regularisation constant could be considered. Testing the model with a without a regularisation constant, it was found that a better approximation could be found by neglecting the regularisation constant. The resulting approximation once again gave a perfect fit of the data with all r-squares equalling 1. Graphically the approximation vs actual function can be seen in Figure 1.17, which confirms the perfect fit.

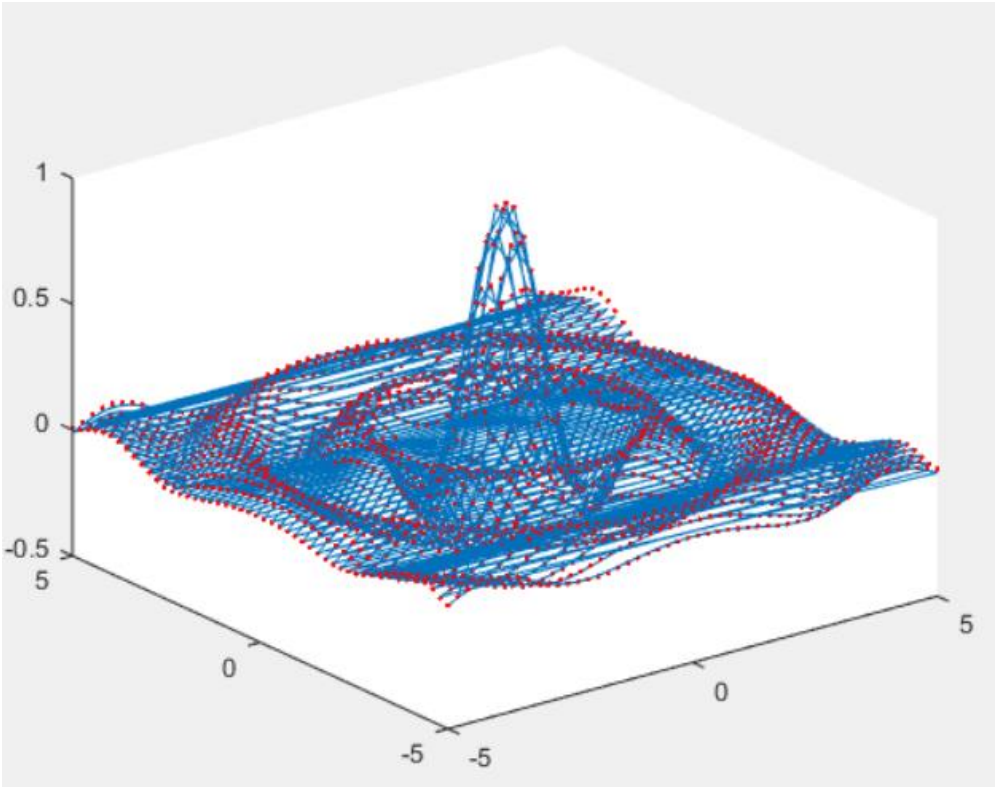

Figure 1.17: Plot of approximation (red) vs true function (blue) for the 2D case

1.4.3 Five dimensional case

For the 5 dimensional case, the intuition is the same as the 2 dimensional case, however there is a large problem with the exponential growth of data by adding dimension. If 50 observations were to be used here, a total of 50^5 data points would be needed, which is in the billions. This amount of data is obviously too much for my mere laptop to analyse. Due to this, the number of observations was dropped 10 in each dimension (I tried more, but my computer didn't appreciate it). However with 10 observations (throughout training, validation and test sets) and 9 infection points, there wasn't much prediction power. For this reason it was decided to drop both the validation and test set and rely solely on the training set, using minimum gradient criteria to reach the optimal solution. As well as this the data was only evaluated from -3 to 3 to allow for a more accurate read of the model. It was thought necessary to incorporate a large amount of neurons due to the 5 dimensions, so 150 (not nearly as many as I would have liked) was used. Even with all these restrictions the data still took a very long time to analyse (36 minutes), and eventually the model was found to not be too accurate with an r-squared of 0.68 and an MSE of 0.0019. It was thought that a scaled conjugate gradient might be able to model this data better, due to the large amount of connection weights, however after running this program with this method, the results were found to be worse.

Exercise 2

2.1 Santa Fe laser data – time-series prediction

In the following question a time series prediction analysis on the Santa Fe chaotic laser will be ran. The aim of this test will be to predict the next 100 observations given the previous thousand. Observing the dataset as a whole (training and test alike) in Figure 2.1, it is clear that there is a very abnormal makeup to the time series that will not be easily analysed. In this graph there is clear

volatility, where the variance constantly increases over time to a point where it suddenly drops sharply before it starts rising again. Due to this only happening every 400 odd observations, this will not be an easy aspect to replicate. Along with this we observe a highly fluctuation structure that reverts from high to low values constantly.

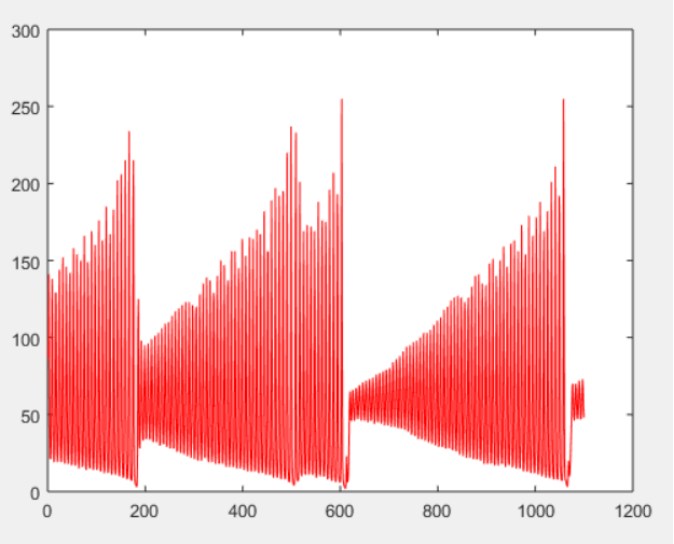

Figure 2.1: Santa Fe Data

To model this time series a non-linear autoregressive with exogenous outputs (NARX) will be employed. This model is evaluated by the data's previous outputs (lags) along with an external input. However, in the current case there is no external input and hence the model is created solely on the data's previous outputs. The model can be shown by

$$\hat{y}_{k+1} = \mathrm{f}(y_k, y_{k-1}, \dots, y_{k-p})$$

where k is the current time point and p is the order of the system(number of lags). This model is parameterised by

$$\hat{y}_{k+1} = w^t tanh(V[y_k, y_{k-1}, \dots, y_{k-p}] + \beta)$$

Where $w^t \in \mathrm{R}^n$ is the transposed matrix of interconnection weights for the output, $V \in \mathrm{R}^{n*p}$ is the matrix of interconnection weights from the inputs to the hidden layers and β is the matrix of biases going to each hidden layer. Note, n is the number of hidden units in the hidden layer. The data was randomly split into groups of 75% for the training set and 25% for the validation set. Looking at the plot without lags it is possible to see the structure of the correlation between lags. In Figure 2.2, it is simple to see that a pattern occurs with a period of 8 observations. It is believed that taking 8 lags should be sufficient. This will allows us to estimate the prediction using all the data contained from the one period back. This means that anywhere over one period back should not affect the predicted value. Note that in the correlation plot although there are significant correlations at greater than 8 time points back, these are assumed to be linked with the 8 lags we have included and hence should be nulled by these 8 lags taken. In figure 2.3 a new autocorrelation plot with the inclusion of the 8 lags is shown. As can be seen this has pretty much removed all the correlation in the data, meaning this should suffice for the amount of lags needed to estimate future predictions. Note, this does not account in any way for the abrupt change every 400 odd observations.

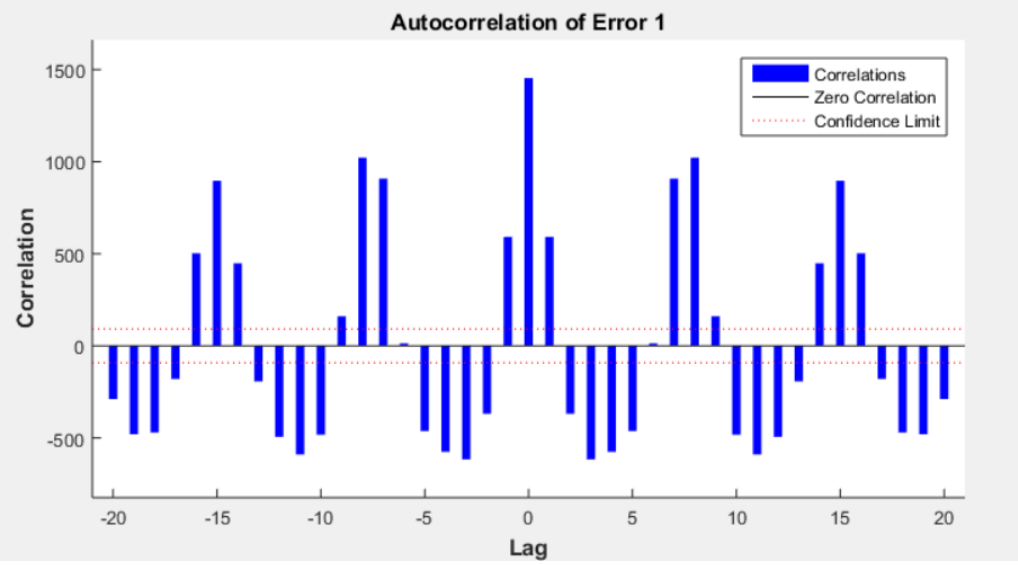

Figure 2.2: Autocorrelation plot with no lags

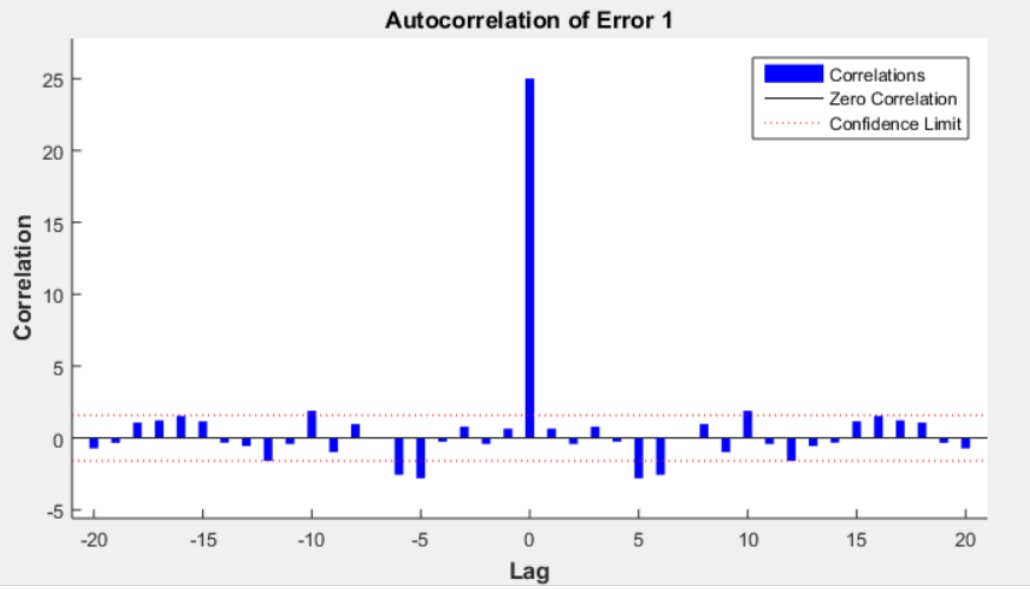

Figure 2.3: Autocorrelation plot with 8 lags

Running a NARX model (with no exogenous input) on the data using 8 lags and 15 hidden neurons, an outcome is achieved that can be observed in Figure 2.4. In order to make predictions on the training we use a recurrent network. Unlike the other networks that have been considered thus far, this network feeds the obtained outputs back into the inputs at each iteration.

As we can see here the data can predict the values quite well up to the point of the sharp decrease in values, however as soon as the values drop, the prediction becomes very poor. The MSE and MAE achieved for this output are 4.84×10^3 and 46.6 respectively.

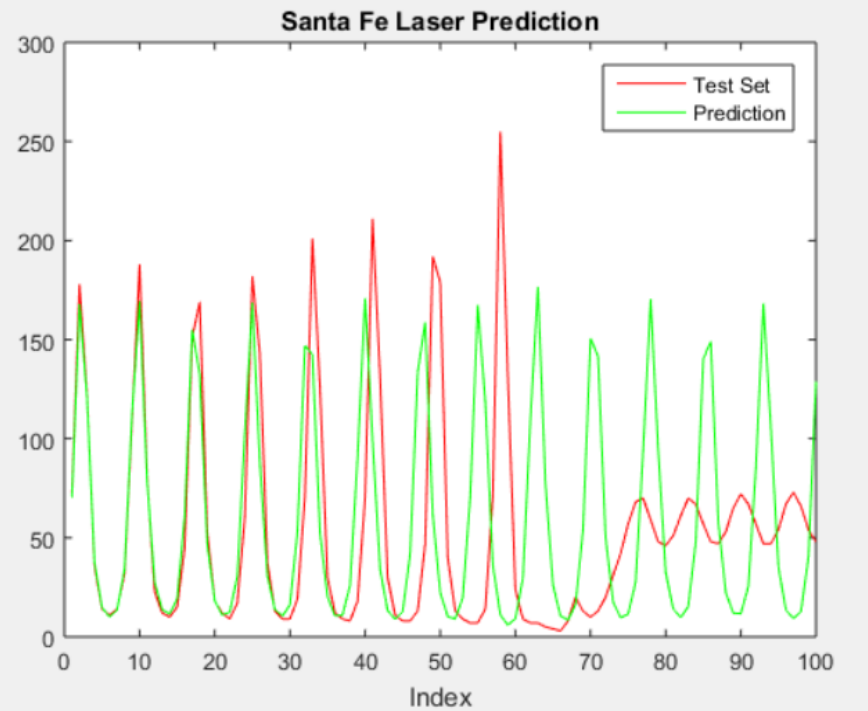

Figure 2.4: Prediction vs Actual

As a possible alternative one could model the data up to the point of the sharp drop, and use this for the training instead, then estimate how often the drop occurs and try to predict the next one. Looking at Figure 2.1 the first drop seems to occur at about the 170th observation, while the second at about 610th. To ensure the drop isn't modelled, 20 observations on either side will be dropped. Therefore the dataset that shall be observed will be from the 190th to the 590th observation. Doing this we obtain an autocorrelation plot with as good as no correlation as seen in Figure 2.5.

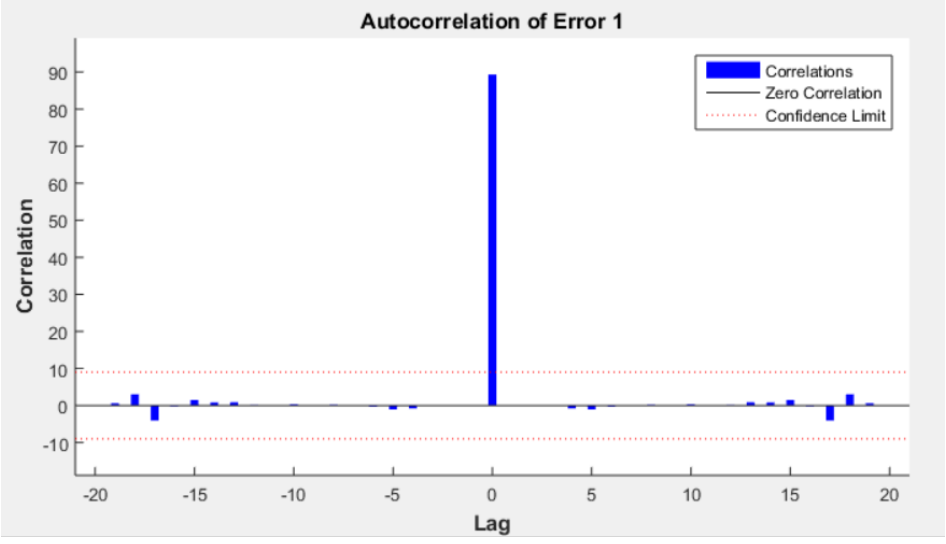

Figure 2.5: Autocorrelation of reduced data with 8 lags

Now an estimated drop will have to be included. Given that the previous drop occurred 440 time points after the preceding drop, the next estimated drop will be at 1050 (610+440). Therefore the first 50 observations will be estimated from the final observation of the training set, while the 2nd 50 will be given by observations just after the previous jump. Doing this we get an MAE and MSE of 20.83 and 1.127 x10^{3} respectively. This is an improvement to the case where the sharp drop wasn't accounted

for. From figure 2.6 below, the modelling accuracy of the estimate can be seen. Although the estimate was dropped too early, the majority of the data is quite well estimated. It was found that by allowing for a great deal of lags that the MSE and MAE could be improved, however I found this to be a very unreliable method, as it modelled nicely up to the point of a drop and then just fluctuated randomly, not really explaining any reality in the model.

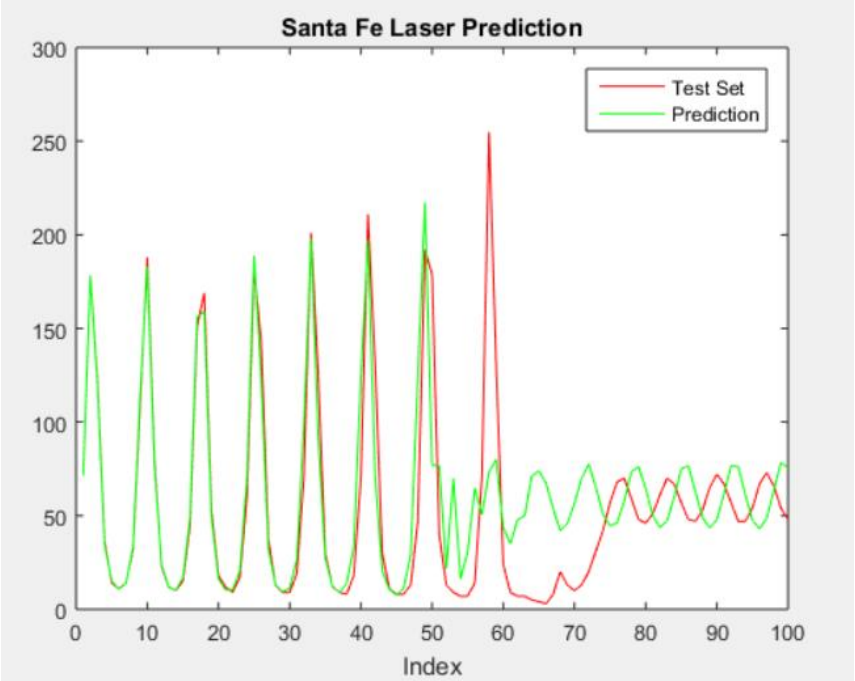

Figure 2.6: Prediction vs Actual accounting for the drop.

2.2 Alphabet recognition

In the following question the Alphabet recognition demo appcr1 in Matlab will be explored. This demo demonstrates how a neural network can perform character recognition. In this demo the letter is evaluated in a 2D manner, where the picture containing the letter is broken in 35 pixels (input neurons) which can be given a value of either 1 or 0, depending on whether the pixel contains data or not. The hidden layer in this network contains 25 hidden neurons. The output layer in this network contains 26 neurons, where each one of these neurons indicates a different letter, which creates a 26 x 26 identity matrix. If one of these neurons fires, this indicates that the network thinks the letter is the one corresponding to this neuron. It does seem that these output neurons could be limited to 5 binary responses as they can give 2^5=32 possible values for the character recognition however, using these 26 output neurons allows the network to further analyse each input pixel in a way that is individual for each character. This allows training to be done in the same way as for regression, where the letters are the target values for the outputs.

For example the letter “T” is expected to have a straight vertical line in the bottom centre pixel. The inclusion of the 26 outputs allows “T” to be distinguished amongst other letters that sometimes also have a data in the same input pixel. This is done by heavily weighting the hidden neurons interconnection weight to output when the images are overlapping, and lightly weighting others. Therefore when all the heavily weighted neurons of “T” are firing, the output will indicate the letter “T”. The structure of this the network as explained can be observed in Figure 2.7. Which confirms what

has already been stated, and also show the associated transfer and activation functions, which are a sigmoid tangent and linear respectively.

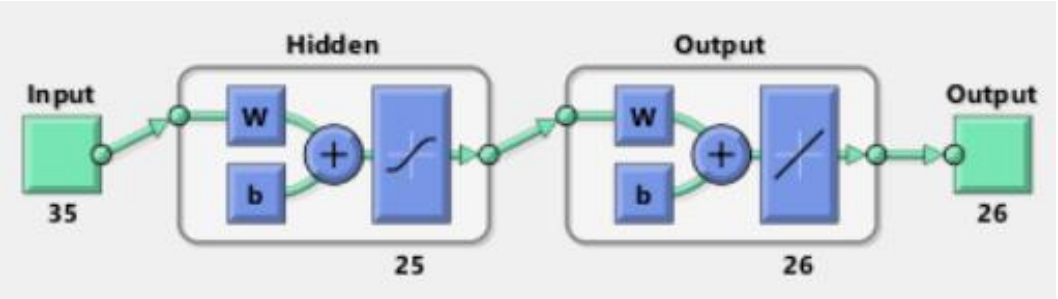

Figure 2.7: Network of character recognition demo

This demo trains two neural networks, one without noise (perfectly and identically written letters) and the other with noise (different images for the same letter each time). Both were modelled using early stopping on a validation set. The networks were trained using a Levenberg- Marquardt training algorithm, as the model was relatively simple. The second network was created using noise, where 30 noisy (different) copies of each letter were analysed. This allows us to view how well the network is able to recognise letters when they are not always written the same. As has been explained above, repeating patterns in input pixels will apply higher weights to the characters that include them. By not including noise any change from the norm will be met with a poor weight and the associated character may not be easily recognisable to the network (may not fire the characters neuron). The two different networks show the plotted values of the letter "A" in Figure 2.8 and 2.9. Here it can be seen that the noiseless version, Figure 2.8, heavily expects the input pixels to be filled in a shape that resembles "A", while Figure 2.9 has a lot more variation, where different weights of importance are applied to different input pixels (big boxes have larger weight and small boxes have less weight).

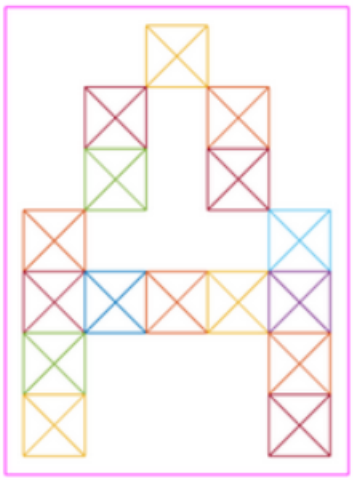

Figure 2.8: Noiseless training of "A"

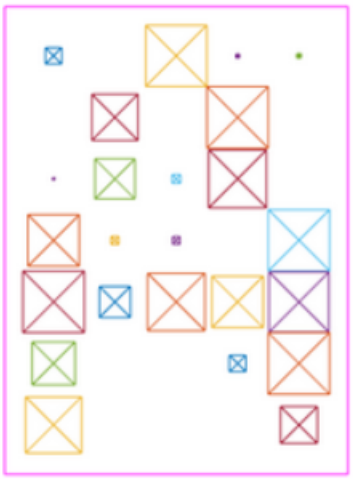

Figure 2.9: Noisy training of "A"

In Figure 2.10 we can see that Network 1 (without noise) has larger errors due to noise than Network 2 which was trained without, as was expected due to points given above.

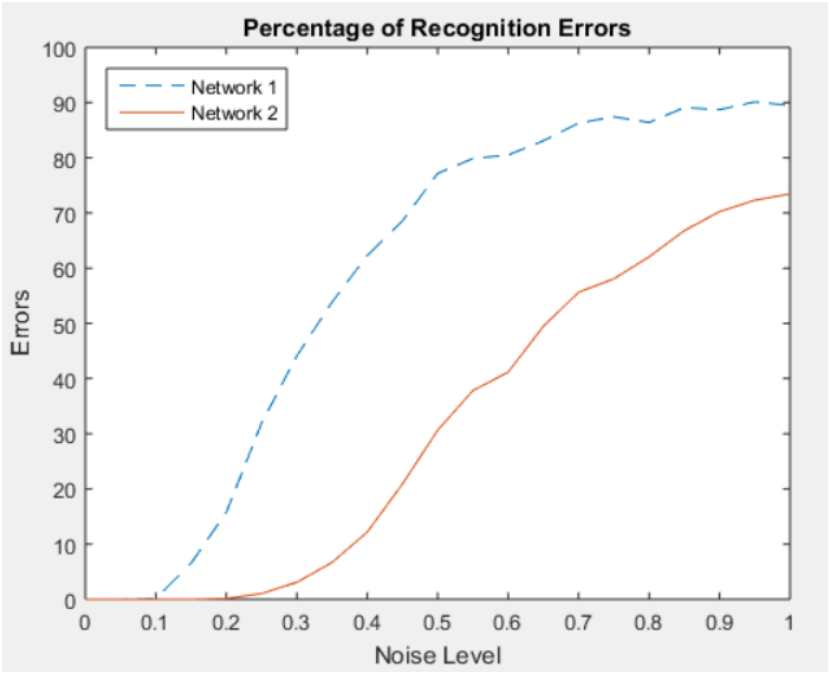

Figure 2.10: Percentage of recognition errors for noiseless and noisy cases

2.3 Pima Indians Diabetes – classification problem

Using data on 768 female Pima Indians and their diabetes diagnoses, the problem of classification will be explored. In this dataset there are 8 predictor variables relating to the patients' medical history, which will be standardised to ensure no one variable applies a larger weight on the analysis than others. The response here is either given as a 1 if the patient does actually have diabetes and a -1 if not. This response if transformed to the sign (hardlimiter) function as it is the more common transfer function and the one that Matlab requires. In the current situation it is known that even with the best classification techniques, about a quarter of the population are misclassified, meaning that the aim will be to get an classification rate of about 75%, and not much higher.

To start the analysis, a linear classification will be considered. Fisher's basic linear discriminant analysis will be initially considered. This is not an iterative procedure, so we will not expect the result to be as accurate as a neural networks technique which uses optimisation techniques. This resulting method misclassifies 29% of the observations, which isn't too much worse than the 25% expected to be misclassified initially.

Using the perceptron algorithm another form of linear classification will be analysed, but this time using neural networks techniques. Unlike linear discriminant analysis this technique does optimise the network. The formula used for this optimisation is given by:

$$y = \text{sign}(v^T x + b)$$

With input $x \in R^8$, output $y \in R^1$, bias b, interconnection weights $v^T \in R^8$ and the activation function of the sign form. This cycle updates the weights at each iteration (Off-Line training, as with the whole project) until the error is not able to be reduced any further. The resulting outputs from the perceptron algorithm show that only 24.5% of the test sets results are misclassified, which is very good, considering about a quarter of patients are generally misclassified even with the best classification techniques. It might be of interest to note that the perceptron algorithm does not make use of a training or test set, so the results are not very generalised.

As that dataset is sizeable, it was decided to randomly break the dataset into 3 sets, training (60%), validation (20%) and test (20%). As there are no neurons here the use of regularisation would be pointless, so was neglected here. The test was run using the Levenberg-Marquardt training algorithm.

Although the linear classification gave a nice result, a non-linear method should be analysed to see if it gives better classification than the linear. This is done using 25 neurons, which may seem a bit excessive, but the inclusion of a regularisation term should protect against unnecessary fluctuation. As the dataset was sufficiently large the most robust method of estimation seemed to be early stopping with a validation set, so although the minimum gradient was considered (and had a minimal difference), stopping with a validation set was opted for. Running the test using different training algorithms it was found that Leverberg-Marquardt algorithm outperformed the others. In Figure 2.11, 2.12, 2.13 and 2.14 the Confusion matrices and ROC curves show the Levenberg-Marquardt and Bayesian Regularisation algorithms. Here we can see that the Levenberg-Marquardt algorithm only misclassifies the data 20.8% of the time, while the Bayesian Regulator misclassifies a slightly higher 25.3% of the time.

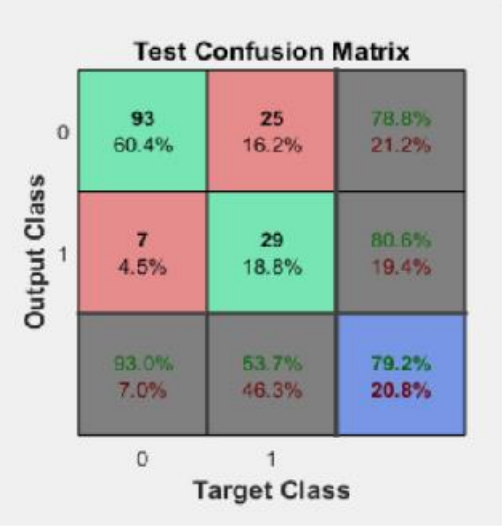

Figure 2.12: Confusion Matrix L-M

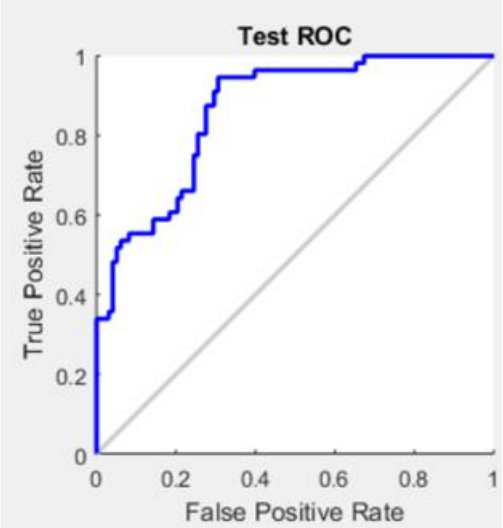

Figure 2.13: ROC Curve L-M

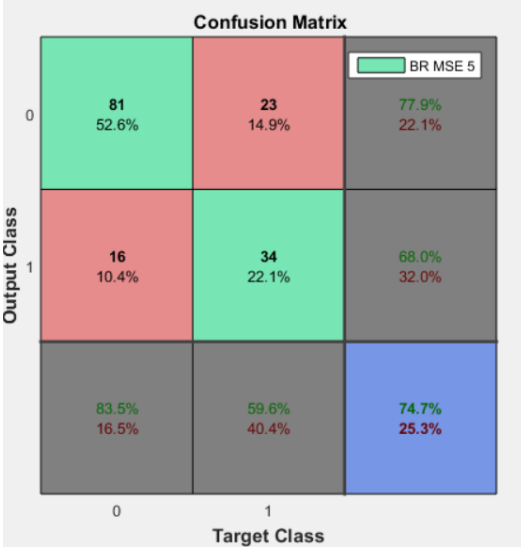

Figure 2.14: Confusion Matrix BR

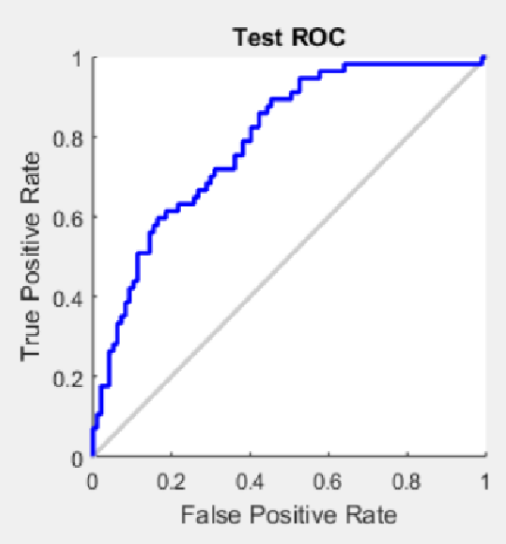

Figure 2.15: ROC Curve BR

Another important aspect of the misclassification may be to look at how the subjects have been misclassified. It may be that giving falsely positive result is far worse than giving a falsely negative result. For example, if a test was done for detecting cancer, falsely giving people the all clear is far worse than bringing people in for further check-ups (false positive). In this case one will want to protect against false negatives (not finding diabetes even though it is present). The Sensitivities, Specificities and false negative are given below.

Sensitivity of Levenberg-Marquardt Method = $TP/(TP + FN) = 29/(29 + 25)$ = 54%

Specificity of Levenberg-Marquardt Method = $TN/(TN + FP) = 93/(93 + 7)$ = 93%

False Negatives rate of Levenberg-Marquardt = 1 Specificity = 46%

Sensitivity of Bayesian Regulation Method = $TP/(TP + FN) = 34/(34 + 23)$ = 60%

Specificity of Bayesian Regulation Method = $TN/(TN + FP) = 81/(81 + 16)$ = 84%

False Positive rate of Bayesian Regulation = 1 Specificity = 40%

Unfortunately both algorithms specify the false negatives poorly, with the Levenberg-Marquardt actually classifying the Sensitivity worse. In this case it might be better to apply heavier importance to the diabetes target class to protect against false negatives.

To recap, although the non-linear methods allow more flexibility, they don't improve upon the non-linear method a huge amount. A classification which manages to correctly classify the data over 75% of the time has been achieved, as was the goal at the start of the question.

Exercise 3

3.1 Dimensionality reduction by PCA analysis

By using Principal Component Analysis (PCA) it is possible to reduce large dimensional input spaces to more defined and understandable dimensions. This reduction allows for less parameters to be estimated, which is desirable when considering complexity criteria. It can also allow us to visually interpret the data, by allowing a humanly dimensional scale of reading (2D or 3D) if the inputs are reduced sufficiently. These new inputs are hoped to have similar underlying attributes that can be interpreted together (eg. Apples, oranges and pears could be classified as fruits). By using PCA it is possible to remove redundancies, which are basically inputs that effect the output(s) in the same way, and hence are highly correlated. PCA does this by orthogonalising the input vectors, so they are uncorrelated with each other. These orthogonalised input vectors are linear combinations of the inputs and are known as Principal components (PCs), which each explain a certain amount of variance in the model. PCA then removes any PCs that don't explain enough variance (decided by the user).

A linear PCA is run on a dataset containing 264 data points, 21 inputs and 3 outputs in a biomedical application. This network was trained using 50% of the observations, while 25% were applied to both a validation and test set alike. The inputs were standardised to ensure no inputs dominated the PCA by applying a proportionally (proportional to other inputs) large amount of variance to the analysis. This gives each input equal importance in analysing the PCA. (The number of PCs used was determined by removing any PCs that contributed less than a specific percentage (user chosen) to the total variation in the model. The resulting eigenvectors from othogonalising are 20.15, 0.75, 0.05, 0.02, 0.01 and other that are too small to mention, which total a value of 21 (as the inputs were standardised and each have variance =1). From this I personally would be quite content with the amount of variance

explained by the 1st factor alone, but will add the 2nd as it contains almost 4% of the variance, which may be deemed important. Using 5 hidden layers the outputs in Table 3.1 were obtained using the Levenberg-Marquardt and Bayesian Regulation training algorithms, in which the test set was used to analyse the results.

		Levenberg-Marquardt	Bayesian Regularization
All Inputs	Seconds	00:13	02:50
	MSE	0.28	0.26
	R-squared	0.79	0.83
Principal Components	Seconds	00:10	00:34
	MSE	0.38	0.31
	R-squared	0.76	0.77

Table 3.1: PCA vs Inputs, and Levenberg-Marquardt vs Bayesian Regularisation

In Table 3.1 it is obvious that reducing the number of inputs reduces the amount of time until convergence, but reduces the accuracy of the prediction. In the case of the Bayesian Regularisation algorithm this difference is far more prominent. The Bayesian Regularisation performs better than Levenberg-Marquardt in this case, proving to be a more powerful predictor, however it also does takes a lot longer to converge. It is clear that the most powerful model comprises of all inputs using the Bayesian regularisation, however one has to make a decision on what is more important, speed or accuracy. Dimensional reduction also has the advantage of reducing the inputs to a dimension that can be used to understand the data, both graphically and interpretively. This kind of info could be very informative for making sense of 1000s of inputs or to people who are not well in tuned with statistics. Realistically there is no real clear answer on what is the preferable choice here. Sometimes accuracy will be seen as more important, for example, if the analysis was conducted on a fatal disease and reasons for it. The causes of this disease are going to want to be understood as well as possible, and no timeframe should be seen as too long. In this case, the most complex model should be analysed. However, perhaps in an econometrical environment calculations will have to be completed on a great numbers of inputs, but the conclusions will have to be evaluated quickly, due to time constraints, and in a manner that can be understood by the boss men in the company. Using PCA would be the optimal choice in this case.

3.2 Input selection by Automatic Relevance Determination (ARD)

Exploring the two demos “demard” and “demev1” we delve into the method of Automatic Relevance Determination (ARD). This method is used to evaluate the most important inputs in the model. This can be achieved by looking at the hyperparameters of the weights which connect each unit to all the hidden units in the next layer. These hyperparameters are the inverse of the variance within each input. The variance gives an idea about the size of the weights of each input, where a small variance indicates that the weights are quite close to zero and hence the input has a low relevance, while a high variance indicates a hyperparameter of weights that are connected to a highly relevant input. As such, hyperparameter values that have large values are seen as irrelevant, while small hyperparameter values indicate relevant inputs.

In the “demard” demo a dataset is explored where three are 3 inputs, one (X1) directly related to the output with a small amount noise, another (X2) directly related to X1 with a larger amount of noise and a third that is randomly sampled. As one would imagine X1 will have the highest relevance, followed by X2 and finally X3 (which has no link to the output). This demo shows how ARD can be utilised to create a network to analyse the most relevant inputs. When running Automatic relevance

determination on this data, the expected relevance of the inputs was indeed the highest for X1 and lowest for X3.

In the "demev1" demo a dataset with a single output is analysed in relation to a sin function. However the input data is only given for a small portion of the sin function period. This example shows how a network can be trained using ARD and shows how gradually pruning the network can lead to more accurate estimates.

Here we will run ARD on the UCI dataset ionsphere with 351 data points and 33 inputs. Like question 2.3 this is again a classification problem, and here we will be hoping to maximise the accuracy of the classification. A training and test set were separated out of the data, where the training set incorporated a random 75% of the original data, and the test set was comprised of the remaining 25%. The training dataset set was then fed using 2 hidden neurons with the scaled conjugate gradient (SCG) algorithm (as is the Matlab default for ARG). In Figure 3.1 and 3.2 a confusion matrix and receiver operating characteristic (ROC) curve of the test set when ran though the network the training set outputted is shown. This shows that considering all inputs only 15.8% of the data has been misclassified.

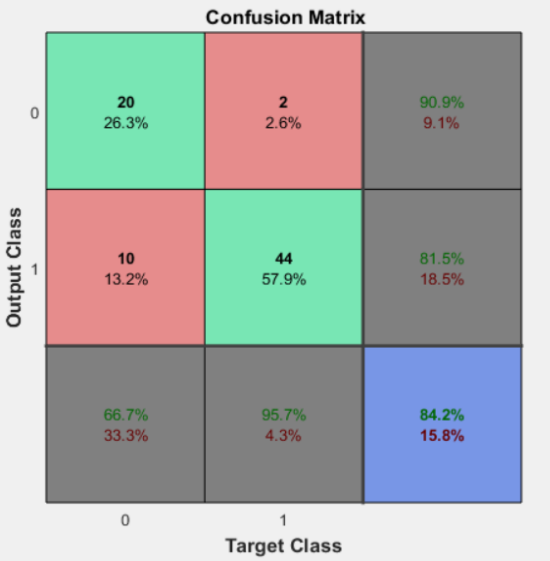

Figure 3.1: Confusion matrix (33 inputs)

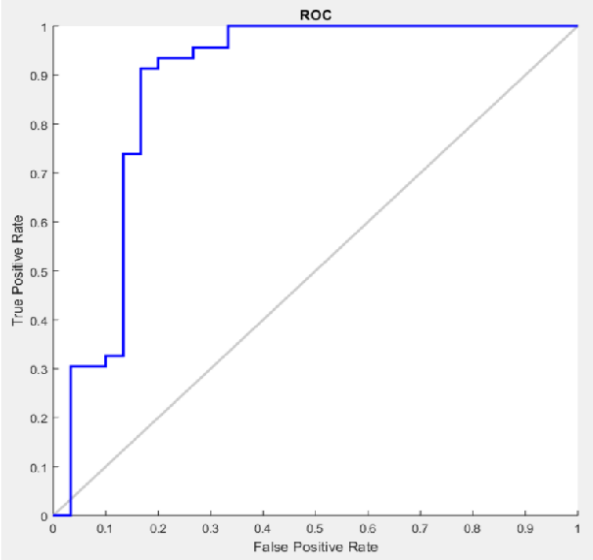

Figure 3.2: ROC curve (33 Inputs)

By running the ARD, it is possible to group together the most relevant inputs in the data. It was decided that any inputs that had hyperparameters that contributed less than 0.5% of the total amount of inverse variance in the model would be the removed from the model. Using this cut-off, 4 inputs remained, variable no.26, 12, 1 and 2. Removing all other inputs and re-running the program as we did with all 33 outputs, a comparison can be made to see how much info was lost with the removal of inputs. In Figure 3.3 and 3.4 the reduced input confusion and matrix and ROC are shown. Comparing the two networks there isn't a huge amount of discriminatory power lost due to the reduction of dimensions, where the network with 33 inputs correctly classifies the data 84% of the time, while the 4 can inputs are capable of correctly classifying the data 76% of the time.

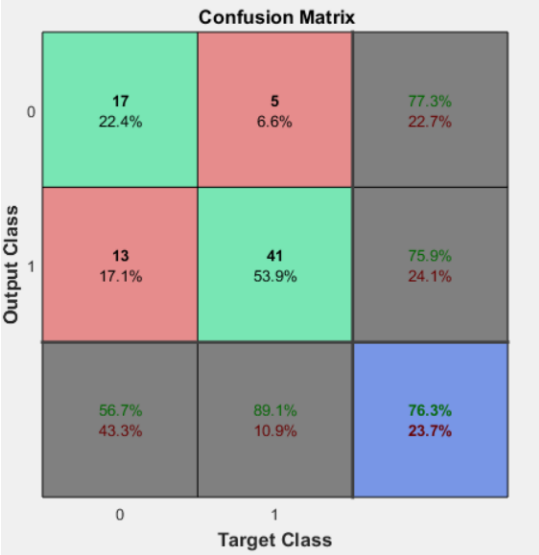

Figure 3.3: Confusion matrix (4 inputs)

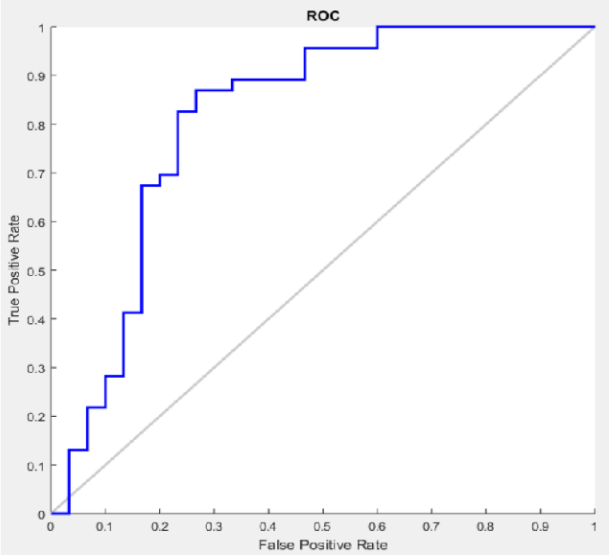

Figure 3.4: ROC curve (4 Inputs)

4 Appendix- Matlab Code

```
%% Excercise 1.1

clc;
clear all;

nnd11gn;

% Excercise 1.2.2
x=linspace(0,1,21);

% Excercise 1.2.3
y = sin(0.7*3.14*x);
plot(x,y,'r-');

% Excercise 1.2.4
net = fitnet ( 2 ) ;
net = configure ( net , x , y ) ;
net . inputs {1}. processFcns = { } ;
net . outputs {2}. processFcns = { } ;
[ net , t r ] = train ( net , x , y ) ;

% Excercise 1.2.5
function [biases, weights] = hidden_layer_weights(net)
    if length(net.layers) ~= 2
        error('This function is meant for networks with one hidden layer');
    end
    biases = net.b{1};
    biases2 = net.b{2};
    weights = net.IW{1,1};
    weights2 = net.LW{2,1};
end
```

```
[ biases , weights ] = hidden_layer_weights ( net ) ;

function [fun] = hidden_layer_transfer_function(net)
    if length(net.layers) ~= 2
        error('This function is meant for networks with one hidden layer');
    end
    Transfer_fun1 = str2func(net.layers{1}.transferFcn);
    Activation_function = str2func(net.layers{2}.transferFcn);

xp1 = tansig(biases(1) + weights(1)*x);
xp2 = tansig(biases(2) + weights(2)*x);

activations = [xp1; xp2];
activations';
% Excercise 1.2.6


% Excercise 1.2.7
yp = purelin(weights2(1)*xp1 + weights2(2)*xp2 + biases2);

plot(x, xp1)
xlabel('input')
ylabel('outputs')
hold on;
plot(x,xp2,'-r')
plot(x,y,'-g')
plot(x,yp,'-o')
legend('xp1','xp2','y','output')
hold off;



%% Excercise 1.3

n= 50
e=0.5
train_x =linspace(-1 , 1 , n) ;
train_y =sin( 2*pi*train_x ) + e*randn(size( train_x )) ;
val_x =linspace(-0.9 , 0.9 , n ) ;
val_y =sin( 2*pi*val_x ) + e*randn(size( val_x )) ;
x = [ train_x val_x] ;
y = [ train_y val_y] ;

net.trainParam.max_fail = 6;
net.trainParam.min_grad = 0.0001;
net.performParam.regularization = 0.000;

net.initFcn='initlay';
net.layers{1}.initFcn = 'rands';
net.layers{2}.initFcn = 'rands';
net.biases{1}.initFcn = 'rands';
net.biases{2}.initFcn = 'rands';
net.inputWeights{1,1}.initFcn = 'rands';
net.inputWeights{2,1}.initFcn = 'rands';
net.layerWeights{1,1}.initFcn = 'rands';
net.layerWeights{2,1}.initFcn = 'rands';

net = fitnet (4 , 'trainlm');
net.divideFcn = 'divideind' ;
net.divideParam = struct('trainInd', 1:n,'valInd',n+1 :2*n,'testInd',[]);
```

```
[ net , tr ] = train (net,x,y) ;

%Get approximated function on training set
train_yhat = net(train_x);

plot(train_x,train_y,'r*');
hold on;
plot(train_x, train_yhat,'-');
plot(train_x,sin(2*pi*train_x),'g-')
hold off;
legend('Traning set','Aproximation','True function')

%% Exercise 1.4

clc;
clear all;

n=100;
x1 = linspace(-5, 5, n);
train_x = x1(1:2:n);
val_x = x1(2:4:n);
test_x = x1(4:4:n);
x = [train_x val_x test_x];
y = sin(pi*x)./(pi*x);
plot(train_x, sin(pi*(train_x))./(pi*(train_x)), '-')
hold on;
plot(val_x, sin(pi*(val_x))./(pi*(val_x)), 'r*');

net = fitnet (9 , 'trainlm');
net.divideFcn = 'divideind' ;
net.divideParam = struct('trainInd', 1:n./2,'valInd',n./2+1
:n./2+25,'testInd',n./2+26 :n);
[ net , tr ] = train (net,x,y) ;
train_yhat = net(train_x);

plot(train_x,sin(pi*(train_x))./(pi*(train_x)),'-');
hold on;
plot(train_x,train_yhat,'r*');
hold off;
legend('Sinc function', 'Approximation');

%% part 2 (2 dimensional)
clc;
clear all;

n=50
x1 = linspace(-5, 5, n)'*ones(1, n);
a = reshape(x1, size(x1,1)*size(x1, 2),1);
b = reshape(x1', size(x1,1)*size(x1, 2),1);
x = [a([1:2:n*n 2:2:n*n]) b([1:2:n*n 2:2:n*n])];
r = sqrt(x(:,1).^2 + x(:,2).^2);
y = sin(pi*(r))./(pi*(r));

plot(r,y)
plot3(x(1:n*n,1), x(1:n*n,2), sin(pi*(r(1:n*n)))./(pi*(r(1:n*n))));

train_x = x(1:(n*n)./2,:);
val_x = x((n*n)./2+1 : (n*n)./2+(n*n./4),:);
test_x = x((n*n)./2+(n*n./4)+1 : (n*n),:);
net = fitnet(81, 'trainlm');
net.performParam.regularization = 0.000000;
```

```
net.divideFcn = 'divideind';
net.divideParam = struct('trainInd', 1:(n*n)./2, 'valInd', (n*n)./2+1 :
(n*n)./2+(n*n./4), 'testInd', (n*n)./2+(n*n./4) : (n*n));
[net, tr] = train(net, x', y');
yhat = net(x');

plot(r(1:(n*n)./2), yhat(1,1:(n*n)./2), 'b.');
hold on;
plot(r((n*n)./2+1 : (n*n)./2+(n*n./4)), yhat(1,(n*n)./2+1 :
(n*n)./2+(n*n./4)), 'r.');
hold on;
plot(r, y, 'g-');
legend('Training', 'Validation', 'Target')

plot3(x(1:n*n,1), x(1:n*n,2), sin(pi*(r(1:n*n)))./(pi*(r(1:n*n))));
hold on;
plot3(x(1:n*n,1), x(1:n*n,2), yhat,'r.');

%% Part 3 (5 dimensions)

clc;
clear all;

n=10
% got to be an easier way to do this, but fuck it
x1 = linspace(-3, 3, n)'*ones(1, n);
x2 = linspace(-3, 3, n)'*ones(1, n*n);
x3 = linspace(-3, 3, n)'*ones(1, n*n*n);
x4 = linspace(-3, 3, n)'*ones(1, n*n*n*n);
a1 = (reshape(x1, size(x1,1)*size(x1, 2),1));
a2 = [a1;a1;a1;a1;a1;a1;a1;a1;a1;a1];
a3 = [a2; a2;a2; a2;a2; a2;a2; a2;a2;a2];
a = [a3; a3;a3; a3;a3; a3;a3; a3;a3; a3];

b1 = (reshape(x1', size(x1,1)*size(x1, 2),1));
b2 = [b1;b1;b1;b1;b1;b1;b1;b1;b1;b1];
b3 = [b2; b2;b2; b2;b2; b2;b2; b2;b2; b2];
b = [b3; b3;b3; b3;b3; b3;b3; b3;b3; b3];

c1 = (reshape(x2', size(x2,1)*size(x2, 2),1));
c2 = [c1;c1;c1;c1;c1;c1;c1;c1;c1;c1];
c = [c2; c2;c2; c2;c2; c2;c2; c2;c2; c2];

d1 = (reshape(x3', size(x3,1)*size(x3, 2),1));
d = [d1;d1;d1;d1;d1;d1;d1;d1;d1;d1];

e = reshape(x4', size(x4,1)*size(x4, 2),1);
x = [a([1:2:n^5 2:2:n^5]) b([1:2:n^5 2:2:n^5]) c([1:2:n^5 2:2:n^5])
d([1:2:n^5 2:2:n^5]) e([1:2:n^5 2:2:n^5])];
r = sqrt(x(:,1).^2 + x(:,2).^2 + x(:,3).^2 + x(:,4).^2+ x(:,5).^2);
y = sin(pi*(r))./(pi*(r));

train_x = x(1:(n^5),:);
net = fitnet(150, 'trainscg');
net.trainParam.min_grad = 0.0001;
net.divideFcn = 'divideind';
net.divideParam = struct('trainInd', 1:(n^5), 'valInd', [] , 'testInd',
[]);
[net, tr] = train(net, x', y');
yhat = net(x');
```

```
plot(r(1:(n^5)), yhat(1,1:(n^5)), 'b.');
hold on;
plot(r, y, 'g-');
legend('Training', 'Target')

plot3(x(1:n^5,1), x(1:n^5,2), sin(pi*(r(1:n^5)))./(pi*(r(1:n^5))));
hold on;
plot3(x(1:n^5,1), x(1:n^5,2), yhat,'r.');
```

Question 2.1

```
clc;
clear all;

trainset = importdata('lasertrain.dat');
trainset2 = trainset(190:590);
testset = importdata('laserpred.dat');


totalset = [trainset; testset];
plot(totalset, 'r-');

xtrain2 = tonndata(trainset, false, false);
%xtrain2 = tonndata(trainset2, false, false);

[trainInd,validInd] = dividerand(1000,0.75,0.25);
%[trainInd,validInd] = dividerand(401,0.75,0.25);

net=narnet(1:500, 40);
net.performParam.regularization = 0.001;
net.trainParam.max_fail = 25;
net.divideFcn = 'divideind';
net.divideParam = struct('trainInd', trainInd, 'valInd', validInd,
'testInd', []);

[Xs, Xi, Ai, Ts] = preparets(net,{},{},xtrain2);
net1 = train(net, Xs, Ts, Xi, Ai);
net = closeloop(net1);
view(net)

xtest2 = tonndata(testset, false, false);

ypred = nan(100+500, 1);
ypred = tonndata(ypred, false, false);
xtrain2 = tonndata(trainset, false, false);
ypred(1:500) = xtrain2(end-(88-1):end);
[xc, xic, aic, tc] = preparets(net, {}, {}, ypred);

ypred = fromnndata(net(xc, xic, aic), true, false, false);
output = num2cell(ypred)';

MSE = perform(net, xtest2, output);
MAE = mae(net, xtest2, tonndata(ypred, false, false));

figure;
plot(testset, 'r-');
hold on;
plot(ypred, 'g-');
```

```
hold off;
xlabel('Index');
title('Santa Fe Laser Prediction');
legend('Test Set', 'Prediction');

%
ypred2 = nan(50+8, 1);
ypred2 = tonndata(ypred2, false, false);
xtrain2 = tonndata(trainset, false, false);
ypred2(1:8) = xtrain2(end-(8-1):end);
[xc, xic, aic, tc] = preparets(net, {}, {}, ypred2);

ypred2 = fromnndata(net(xc, xic, aic), true, false, false);
output2 = num2cell(ypred2)';

ypred3 = nan(50+8, 1);
ypred3 = tonndata(ypred3, false, false);
xtrain2 = tonndata(trainset, false, false);
ypred3(1:8) = xtrain2(618-(8-1):618);
[xc, xic, aic, tc] = preparets(net, {}, {}, ypred3);

ypred3 = fromnndata(net(xc, xic, aic), true, false, false);
output3 = num2cell(ypred3)';

output4 = [output2 output3];
ypred4 = [ypred2; ypred3];

MSE = perform(net, xtest2, output4);
MAE = mae(net, xtest2, tonndata(ypred4, false, false));

figure;
plot(testset, 'r-');
hold on;
plot(ypred4, 'g-');
hold off;
xlabel('Index');
title('Santa Fe Laser Prediction');
legend('Test Set', 'Prediction');


%% Exercise 2.2

clc;
clear all;

appcr1;

%% Exercise 2.3

clc;
clear all;

PID = importdata('pidstart.mat');
[x, stdx] = mapstd(PID.Xnorm');
t = hardlim(PID.Y');
[x1, stdx] = mapstd(PID.Xnorm);
t1 = hardlim(PID.Y);
```

```
[trainInd,valInd,testInd] = dividerand(768,0.6,0.2,0.2);

cls = fitcdiscr(x1,t1);
resuberror = resubLoss(cls)
R = confusionmat(cls.Y,resubPredict(cls))

n=25;
net = patternnet(n);
net = perceptron;
net.divideParam.trainRatio = 60/100;
net.divideParam.valRatio = 20/100;
net.divideParam.testRatio = 20/100;
net.trainFcn = 'trainlm';
net.performParam.regularization = 0.1;
net.trainParam.max_fail = 6;
net.trainParam.min_grad = 0.00000000000001;
[net, tr] = train(net, x, t);

testX = x(:,tr.testInd);
testT = t(:, tr.testInd);
testY = net(testX);
testClass = testY > 0.5;
figure, plotconfusion(testT, testClass), legend('BR MSE 5');

[c,cm] = confusion(testT,testY)


  %%Exercise 3.1

clc;
clear all;

doc cho_dataset
load cho_dataset
% Standardize the variables
doc mapstd;
[pn, std_p] = mapstd(choInputs);
[tn, std_t] = mapstd(choTargets);

doc processpca;
[pp, pca_p] = processpca(pn, 'maxfrac', 0.01);
[pc,score,latent,tsquare] = princomp(pn');
[m, n] = size(pp)

%part b
Test_ix = 2:4:n;
Val_ix = 4:4:n;
Train_ix = [1:4:n 3:4:n];

net = fitnet(5);
net.divideFcn = 'divideind';
net.trainFcn = 'trainbr';
net.divideParam = struct('trainInd', Train_ix, ...
'valInd', Val_ix, ...
'testInd', Test_ix);
[net1, tr] = train(net, pn, tn);
[net2, tr] = train(net, pp, tn);

% Get predictions on training and test
```

```
Yhat_train = net1(pn(:, Train_ix));
Yhat_test = net1(pn(:, Test_ix));
MSE_train = perform(net1,tn(:,Train_ix),Yhat_train)
MSE_test = perform(net1,tn(:,Test_ix),Yhat_test)

Yhat_train1 = net2(pp(:, Train_ix));
Yhat_test1 = net2(pp(:, Test_ix));
MSE_train = perform(net2,tn(:,Train_ix),Yhat_train1)
MSE_test = perform(net2,tn(:,Test_ix),Yhat_test1)



%% Exercise 3.2

clc;
clear all;

demard;

demev1;

% excercise part
uic = importdata('ionstart.mat');

[x, std_input] = mapstd(uic.Xnorm); % normalizing input
t = hardlim(uic.Y); % converting from [-1,1] to [0,1] thus no need to find
beta

[trainInd,testInd] = dividerand(351,0.75,0.25)

nin = 33;            % Number of inputs.
nhidden = 5;            % Number of hidden units.
nout = 1;            % Number of outputs.
aw1 = 0.01*ones(1, nin);
ab1 = 0.01;          % Hyperparameter for hidden unit biases.
aw2 = 0.01;          % Hyperparameter for second-layer weights.
ab2 = 0.01;          % Hyperparameter for output unit biases.
beta = 50.0;            % Coefficient of data error.

% Create and initialize network.
prior = mlpprior(nin, nhidden, nout, aw1, ab1, aw2, ab2);
net = mlp(nin, nhidden, nout, 'logistic', prior, beta); %logistic because
binary classification problem

% Set up vector of options for the optimiser.
nouter = 8;          % Number of outer loops
ninner = 10;                % Number of inner loops
options = zeros(1,18);      % Default options vector.
options(1) = 1;          % This provides display of error values.
options(2) = 1.0e-7;     % This ensures that convergence must occur
options(3) = 1.0e-7;
options(14) = 300;       % Number of training cycles in inner loop.

% Train using scaled conjugate gradients, re-estimating alpha and beta.
for k = 1:nouter
  net = netopt(net, options, x(trainInd,:), t(trainInd,:), 'scg');
  [net, gamma] = evidence(net, x(trainInd,:), t(trainInd,:), ninner);
end
```

```
% hyperparameter values
clc;
i = 1:nin;
p = net.alpha(1:nin,:);
fprintf(1, '  alpha%i =  %8.5f\n', [i; p']);
fprintf(1, '  beta  =  %8.5f\n', net.beta);
fprintf(1, '  gamma =  %8.5f\n\n', gamma);

% Corresponding weight values
fprintf(1, '    %8.5f    %8.5f\n', net.w1');

[pred, z] = mlpfwd(net, x(testInd, :));

plotconfusion(t(testInd,:)', pred');
plotroc(t(testInd,:)', pred');

x5 = x(:,1);
x2 = x(:,2);
x3 = x(:,12);
x4 = x(:,26);
x1 = [x5, x2, x3, x4];

nin = 4;
aw1 = 0.01*ones(1, nin);

prior = mlpprior(nin, nhidden, nout, aw1, ab1, aw2, ab2);
net = mlp(nin, nhidden, nout, 'logistic', prior, beta);


for k = 1:nouter
  net = netopt(net, options, x1(trainInd,:), t(trainInd,:), 'scg');
  [net, gamma] = evidence(net, x1(trainInd,:), t(trainInd,:), ninner);
end

[pred, z] = mlpfwd(net, x1(testInd, :));

plotconfusion(t(testInd,:)', pred');
plotroc(t(testInd,:)', pred');
```