KU Leuven

Data Mining and Neural Networks

Akshat Dwivedi (Student Number: )

Instructor: Prof. Johann Suykens
Course: G9X29A

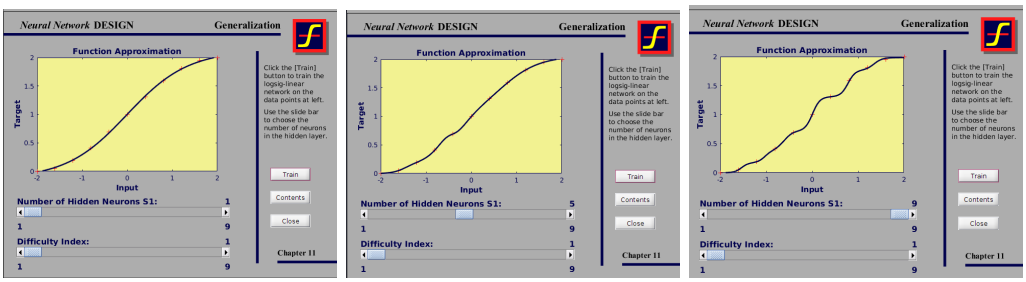
January 11, 2015

Contents

1 Session 1

1.1 Function approximation (noiseless case)

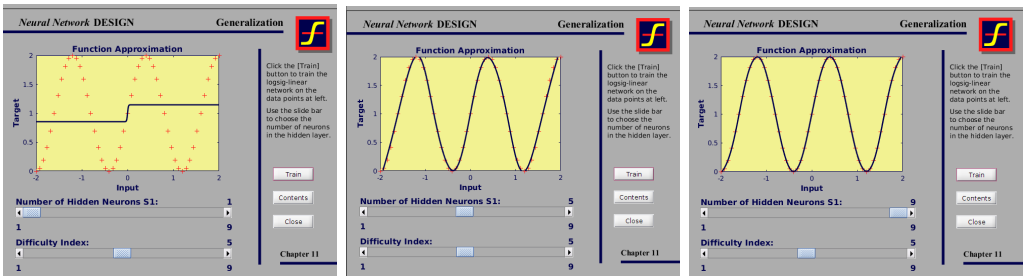
This section involves playing with the neural network demo nndd11gn that comes with the MATLAB neural network toolbox. There are two options: selecting the number of hidden units in the network and the difficulty of the function to be approximated (scale 1-9). Index 1 corresponds to what appears to be a single sigmoid curve. The network trained on the data consists of log-sigmoid as the activation function on the units in the hidden layer. The output layer consists of one node and has the linear (identity) function as the activation function. This means that the output layer outputs without transforming the input it receives from the hidden layer. The first case (figure 1) compares the use of 1, 5 and 9 neurons to approximate a simple sigmoid function with difficulty index 1.



(a) Index = 1, neurons = 1 (b) Index = 1, neurons = 5 (c) Index = 1, neurons = 9

Figure 1: Neural network demo nnd11gn

In figure 1, we see that 1 neuron is sufficient to approximate the original function and the curve is rather smooth. In case of 5 or 9 neurons, the approximated curve is overfitting the data which is indicated by the bumps in the approximated function.

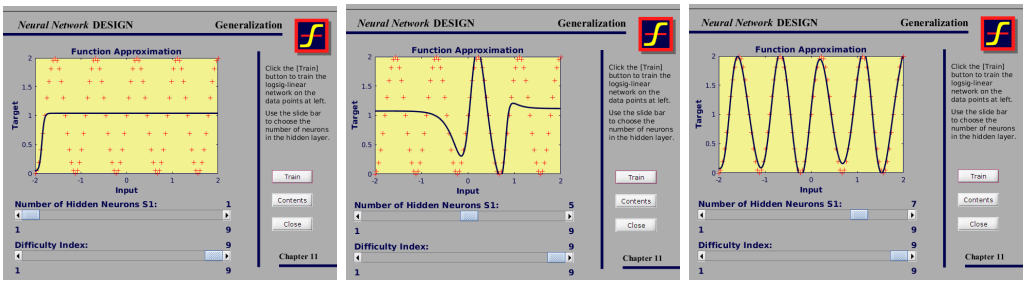


(a) Index = 5, neurons = 1 (b) Index = 5, neurons = 5 (c) Index = 5, neurons = 9

Figure 2: Neural network demo nnd11gn

In figure 2, we see that using 1 neuron is insufficient to approximate the underlying function (after multiple epochs). Using 5 neurons is sufficient to approximate the underlying function (although there were a few cases where the optimization algorithm

reached a local minima and the network failed to approximate the function.) This is a well-known drawback of using neural networks. The network was able to capture the underlying relationship in the data with 9 neurons across multiple epochs. This would usually lead to overfitting, unless regularization was used to keep the network weights small or some form of pruning were applied after training the network.



(a) Index = 9, neurons = 1 (b) Index = 9, neurons = 5 (c) Index = 9, neurons = 7

Figure 3: Neural network demo nnd11gn

Finally, we consider the case with the maximum difficulty index: 9. Based on figure 3, we see that 1 neuron is completely insufficient for approximating the true function. 5 neurons is insufficient to approximate the function as well, however, it is able to approximate a part of the true function. We do not show the 9 neuron case, but instead, show that 7 neurons was sufficient to approximate the underlying function in about 80% of the cases. Adding 8 or 9 neurons would only improve this number. Lastly, one should be careful at deriving rules of thumbs based on this demo, as the underlying function did not have any noise in the data, and the number of observations available in the training set would affect the training and the architecture of the network as well.

The under/over fitting is due to the number of hidden units and the activation function employed. In the 5 or 9 index case, the underlying function appears sinusoidal. Thus, using a sinusoid as an activation function can reduce the number of hidden units required to approximate the true function, compared to the sigmoid function that was used. However, the sinusoidal function is not monotonic, and hence the universal approximation theorem does not guarantee that one can learn the function by using this activation function (monotonicity of the activation function is a requirement of the theorem).

1.2 The role of the hidden and output layers

Linear (ordinary least-squares) regression can be performed by using the simplest neural network architecture, which is composed of a single hidden layer comprising a single node with a linear activation function and training using squared error loss. The input layer has p nodes (the number of features or columns in the dataset) and the network has one output neuron in the output layer. Sometimes, an additional bias term can be added to the input layer as well, in which case the input layer has $p + 1$ nodes. The input to the network is a collection of n p-dimensional vectors. Thus, n is the number

of records or observations, and p is the columns of the data matrix. Symbolically, it can be written as

$$y = \sigma(w^T x + \beta) \tag{1.1}$$

where $\sigma(w^T x + \beta) = w^T x + \beta$ is the linear (identity) activation function,

$$w^T x + \beta = w_1 x_1 + \ldots + w_p x_p + \beta \tag{1.2}$$

and β is the bias (threshold) term.

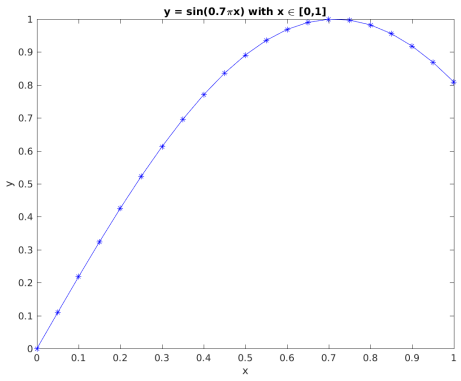
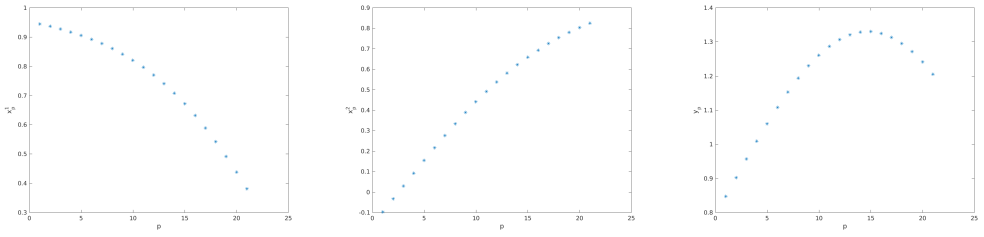


Figure 4: Plot of y = sin(0.7πx) vs x

Figure 4 contains a plot of the function $y_p = \sin(0.7\pi x_p)$ where $x_p \in [0, 1]$, i.e., 21 equally spaced points between 0 and 1. Since this function is nonlinear, linear regression as described in the starting paragraph will not be able to capture the true underlying relation between the inputs and the outputs.

We first train a network with two neurons in the hidden layer using the above values of x and y. The biases for the two hidden neurons b_1 = -1.7735 and b_2 = 0.0982; and the weights w_1 = 1.3714 and w_2 = -1.2680. The inputs to the network were not scaled before training. The activation function on the neurons in the hidden layer is the hyperbolic tangent function [2/1+exp(-2n)]-1. The activation function for the output layer is the linear/identity function which mirrors the information from the hidden layer to the single node in the output layer.

Figure 5 shows a plot of x_p^1, x_p^2 and y_p against $p = 1, \ldots, 21$. Here we see parts of the true function that are approximated by the two neurons x^1 and x^2. Individually, either neuron cannot model the true function, however part (c) shows that a linear combination of the two neurons (with unit weights) does capture the true relationship. Notice that the y-axis shows the function outside the range of the original y. This is because we have not yet taken a weighted combination of the neurons yet. The final output from the network is plotted in figure 6.



(a) x_p^1 (b) x_p^2 (c) $y_p = x_p^1 + x_p^2$

Figure 5: Plot of x_p^1 x_p^2 and y_p against $p = 1, \ldots, 21$

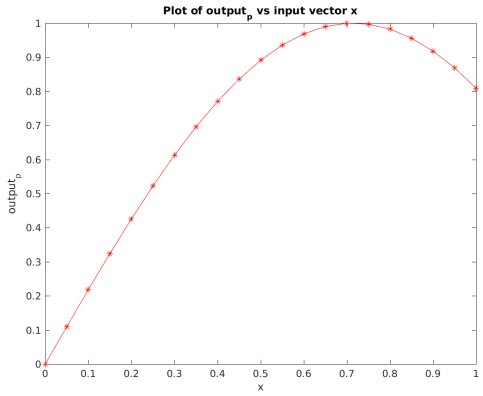
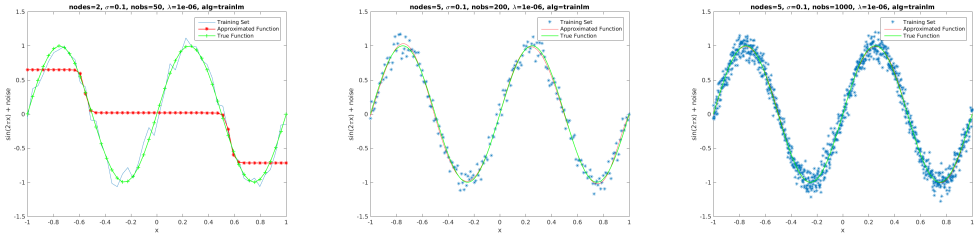


Figure 6: Plot of output$_p$ vs. x$_p$. This is indistinguishable from the original target y.

1.3 Function approximation (noisy case)

For this part, a training set of a sinusoidal signal sin(2πx) on [-1,1] with additional Gaussian noise is generated with 50, 200 and 1000 observations. A validation set is chosen to be of the same size as the training set with the sinusoidal signal on [-0.9,0.9] with additional Gaussian noise as well. Three different levels of noise σ are chosen, namely 0.1, 1.0 and 2.0 for simulating the Gaussian error $\varepsilon \sim N(0, \sigma)$. Regularization $(0.8, 0.3$ and $10^{-4})$ or early stopping were used while training the network and a multi-layer perceptron with a single hidden layer with 5, 10 and 20 neurons was used.

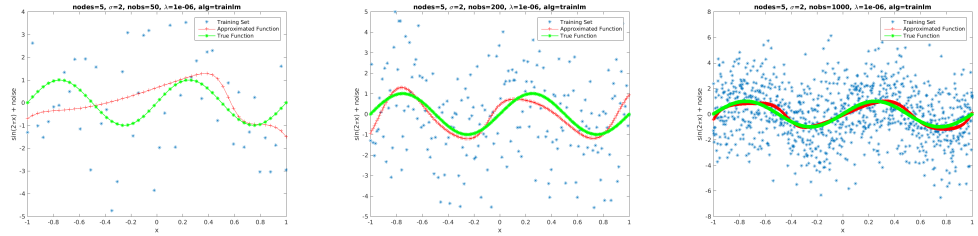
The primary training algorithm used was the Levenberg-Marquadt algorithm (LM) which was chosen due to its fast convergence. The division of the training and the validation sets were done using fixed indices, which eliminates the variability from random division of the dataset into training and validation sets before training the network. The initial seed and the random number generator were fixed at the start. This allows us to make all the comparisons without worrying about additional sources of variability as all sources of variability have been eliminated for these comparisons and the results can be reproduced using the code given in the appendix.



(a) $n_h = 2$, n = 50 (b) $n_h = 5$, n = 200 (c) $n_h = 5$, n = 1000

Figure 7: LM algorithm, noise $\sigma = 0.1$ and using early stopping.

In figure 7, we compare the cases with 50, 200 and 1000 observations. The first image was fit with 2 neurons, however, this resulted in a poor fit and the number of neurons after this were taken to be 5 (except for the sections on regularization and comparing different number of hidden units). The noise is fixed at $\sigma = 0.1$ and we see that the noisy function does not deviate too much from the true function. In the second part, we see that with 200 observation, the network is able to approximate the true function and this accuracy is improved with 1000 observations (part (c)). This would indicate that the presence of noise can be overcome with more data.



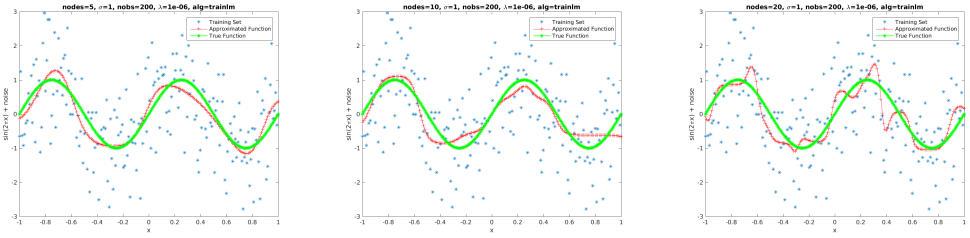
(a) n = 50 (b) n = 200 (c) n = 1000

Figure 8: noise $\sigma = 2.0$, $n_h = 5$, early stopping.

In figure 8, we see that with a large amount of noise ($\sigma = 2.0$) and a small number of observations, the network is unable to approximate the underlying process with 5 neurons, as it was able to before. However, with increasing cases (200 and 1000), the network improves at approximating the underlying function.

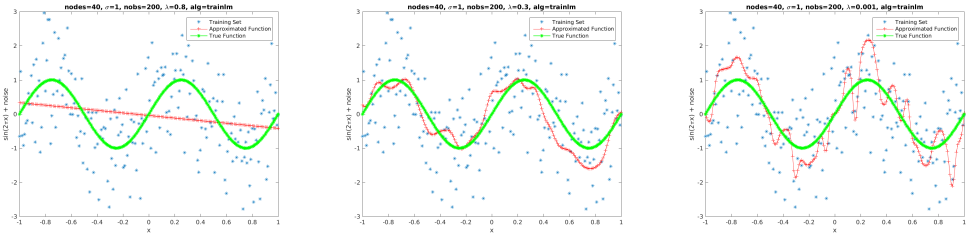
This third figure 9 compares the accuracy of the network with different hidden units. The figures with 5 and 10 neurons are still relatively close to the true function, however, we see that adding more neurons leads to the network overfitting to the given training set, although it still models the general shape of the true function.

The fourth figure 10 assess the impact of choosing an appropriate regularization parameter. The figure on the left shows too much regularization ($\lambda = 0.8$) which results



(a) $n_h = 5$ (b) $n_h = 10$ (c) $n_h = 20$

Figure 9: Comparing the number of hidden units. Noise $\sigma = 1.0$ and n = 200 are fixed.



(a) $\lambda = 0.8$ (b) $\lambda = 0.3$ (c) $\lambda = 10^{-4}$

Figure 10: $\sigma = 1.0$, $n_h = 40$, **regularization** λ, n = 200

in a large bias in the approximated function and fails to approximate the true function. The approximation function in the middle ($\lambda = 0.3$) appears to give a decent approximation to the true function, however, it is still learning patterns in the data that are a result of noise (random fluctuations) and not due to the underlying process. The third figure results in too little regularization ($\lambda = 10^{-4}$) which results in a large variance in the approximated function but little bias. The approximated function is overfitting the training data and is failing to obtain a good generalization. Thus, the middle figure appears to be a decent example of the bias-variance trade-off. Selecting a λ value between 0.3 and 0.6 should most likely result in a good fit to the data.

Algorithms	Levenberg-Marquadt	Scaled Conjugate Gradient	BFGS quasi-Newton	resilient backpropagation
$n_h = 5$ epochs = 200	25s	21s	39s	11s
$n_h = 40$ epochs = 50	39s	25s	59s	13s

Table 1: Comparing the training times (in seconds) of the different algorithms. Early stopping and regularization were not used; only runtimes were of interest. n = 10^6, noise $\sigma = 1.0$ and number of hidden units (5 and 40) were held fixed.

This final comparison uses the LM, SCG, BFGS and resilient backpropagation algorithms for training the network. The number of cases were increased to 10^6 instead of 1000. Although there was only one feature, adding more features to the dataset would enable us to distinguish between the algorithms to a greater degree. The noise level was fixed at 1.0, and n_h was 5 (case 1) and 40 (case 2). All the networks were trained without regularization or early stopping and for a fixed number of epochs (case 1: 200) and (case 2: 50). From table 1, we see that based on runtimes, resilient backpropagation is the fastest, with scaled conjugate gradient coming in second place. Thus, for much larger datasets, it would make sense to use the resilient backpropagation alogorithm.

1.4 Curse of dimensionality

Our aim in this part is to model the popular $y = sinc(x)$ function for $x \in \mathbb{R}^m$, m = 1, 2 and 5 dimensions using a neural network.

Case m = 1

First, we create a training set and a test set. The training set comprises of 100 equally spaced points on [-5,5] starting at -5. For selecting the test set from the same interval, we try to ensure that there is no overlap between the training and the test sets. For this, we can choose n_{test} from [-5,5] by using a number that is coprime to 100. We take 53 equally spaced points on the interval [-4.9,4.9] starting at the end point -4.9. This results in the training set A ($\subset$ [-5,5]) and test set B ($\subset$ [-4.9,4.9]) being mutually disjoint. This is shown in figure 11.

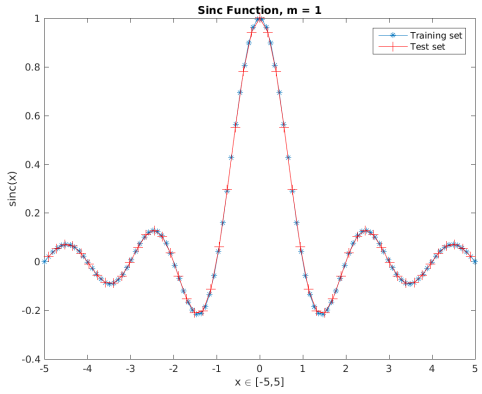
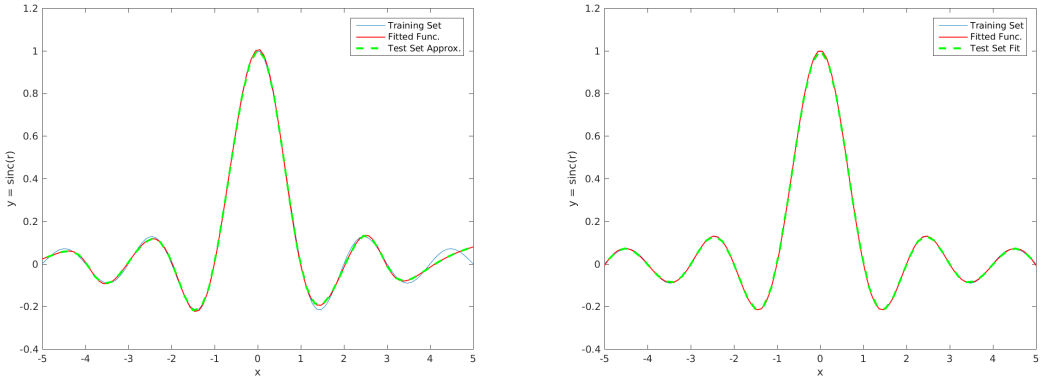


Figure 11: Plot of the 1-dimensional sinc(x) function for x. Training set is from [-5,5] and test set is from [-4.9,4.9]. Both sets are mutually disjoint.

Figure 11 contains a plot of the training and the test sets. We train two MLPs on the training set with one hidden layer and 5 neurons. The difference between the two networks is that the first one uses the hyperbolic tan function as the activation function and the second one uses the radial basis function. Figure 12 shows the differences in the

approximated functions from these two networks. The network with the tanh activation provides a poorer fit for $x > 3.8$. We prefer to go with the radial basis network as it provides a better fit with a smaller number of neurons.



(a) tanh activation (b) radial basis function (RBF)

Figure 12: Using an MLP with 5 hidden units to approximate sinc(x) on [-5,5].

The advantage of using a smaller number of neurons will become clearer for the cases with higher dimensionality.

Case m = 2

Similar to the case for m=1, the aim is to use a neural network to train the `sinc` function for 2 dimensions. This function is also known as the mexican hat function. A plot of this function over the 2-dimensional interval $[-5, 5]^2$ is given in figure 13.

It is interesting to note, that in the 1-dimensional case, 100 points were used (training set) and there was a lesser amount of "empty space" in the plot. Comparing this to the 2-dimensional case, we see that there is a lot more "empty space" in 3-dimensions compared to 2-dimensions. If we take 100 equally spaced points for x_1 and $x_2 \in [-5, 5]$, we get a grid of $100^2 = 10{,}000$ points in the training set. Thus, we observe that the number of cases n has now increased from 100 to 10,000. Generally speaking, for an m-dimensional input space, the training set would contain 10^m cases. This is known as the *curse of dimensionality*.

We decide to go with the full training set with 10,000 points and use early stopping with a validation set consisting of 53 equally spaced points on [-4.9,4.9]. Thus, the validation set consists of $53^2 = 2809$ points. Finally, a separate test set of 40^2=1600 points on $[-4.8,4.8]^2$ is created to evaluate the network performance and is not used at the time of training the network. The advantage of selecting these three sets is that they are mutually disjoint and hence, the network is evaluated on sets which are similar, but not the same as the training set. The activation function function used is the radial basis function which is selected due to it's advantage in using a smaller number of neurons.

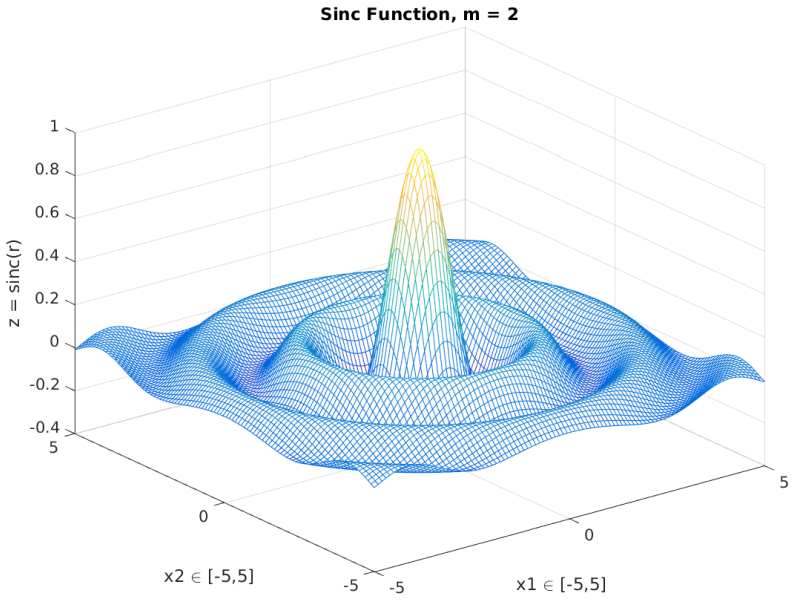


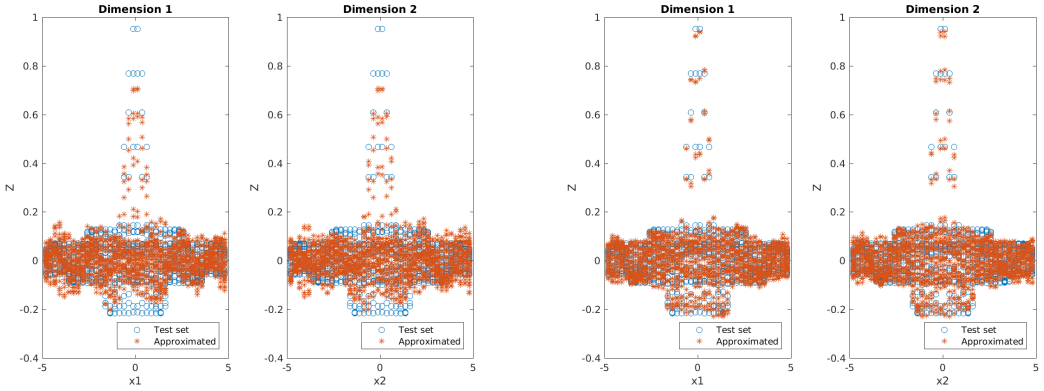
Figure 13: Mexican hat function on the 2-dimensional interval $[-5, 5]^2$

Levenberg-Marquadt algorithm is used to train the network due to it's fast convergence rate, however resilient backpropagation was tried and resulted in a poorer fit compared to the LM algorithm. The network was trained with 25 and 35 neurons.

In figure 14(a), we see that the network with 25 neurons is not able to adequately approximate the underlying function. The MSE on the test set is 0.0039. In 14(b) we see that the performance of the network with 35 neurons results in a rather good fit. The MSE was found to be 0.000494.

Case m = 5

Now we arrive at the interesting part of the exercise: evaluating the `sinc` function in 5 dimensions. The first problem we run into is that the training and test sets cannot be chosen in a manner that was employed for the 1- and 2-dimensional cases. Selecting 100 equally points on a 5-dimensional grid, for example, will lead to 100^5=10,000,000,000 observations which is simply too impractical. Furthermore, it would be challenging to visualize the approximated function due to the dimension being greater than 3. Thus, we compare the fitted models based on MSE on the training and test sets. 5 networks are trained with 100, 200, 300, 400 and 500 nodes in the hidden layer. Furthermore, we used the scaled conjugate gradient algorithm to train the network as it is suitable for large scale problems as it does not store large matrices in memory. Training set was chosen to be 15 equally spaced points on the $[-5, 5]^5$ interval which results in $15^5 = 759375$ cases used to train the network. A validation set of 10 equally spaced points was



(a) 25 neurons (b) 35 neurons

Figure 14: Mexican hat function in 2 dimensions. Plot of the test set with the approximated function on the test set. 25 and 35 neurons were tried.

Hidden units	Training Set	Test sets
100	0.0014	0.0014
200		
300		
400		
500		

Table 2: MSE of the network on the test set for the sinc function in 5 dimensions.

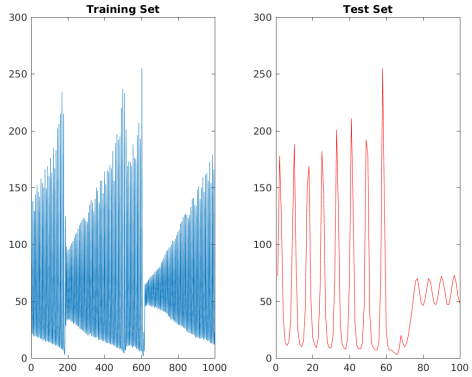
chosen on the $[-4.9, 4.9]^5$ which resulted in 100000 cases in the validation set. Similarly, a test set of 10 equally spaced points was chosen on the $[-4.8, 4.8]^5$ which resulted in 100000 cases in the test set. Early stopping was used to train the network, however, due to time constraints, we trained the network for 100 epochs and training was stopped whichever of these two occurred first.

2 Session 2

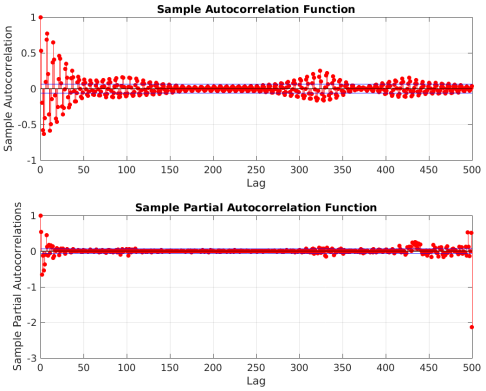
2.1 Santa Fe laser prediction

This exercise involves time-series prediction. The data are generated by a chaotic laser (which is a nonlinear dynamical system) and 1000 points are available for training the model. 100 observations are available for assessing the prediction of the model, which are not used at the time of training the model. A plot of the training set and the test set are given in figure 15.

We train an MLP with one hidden layer in feedforward mode using the training set.



(a) Training and Holdout data



(b) Autocorrelation plots for the Santa Fe series.

Figure 15: Santa Fe data: series plotted on the left; with autocorrelation plots on the right.

Symbolically, it can be expressed as

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

where p is the lag of the series. The trained network was then used as a recurrent neural network to predict the next 100 iterations

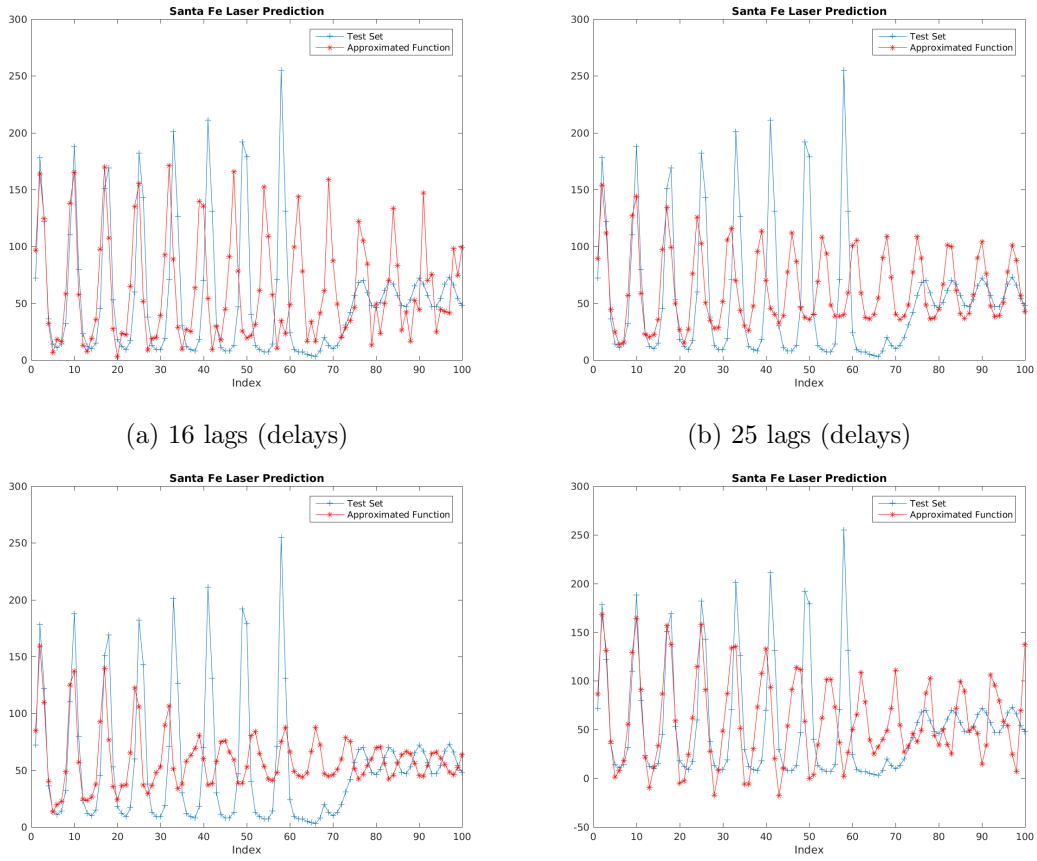
$$\hat{y}_{k+1} = w^T \tanh(V[\hat{y}_k; \hat{y}_{k-1}; \ldots; \hat{y}_{k-p}] + \beta)$$

where the current predicted observation is fed back into the network to predict the next observation.

Different combinations of lags and neurons in the single hidden layer were tried and the scaled conjugate gradient algorithm was used for training the network with a regularization parameter of 10^{-6}. The cost function employed for training was the MAE (mean absolute error). The evaluation of the network on the test set was done using the MAE (mean absolute error). The (partial) autocorrelation plot of the original series is given in figure 15. We can see from the PACF that the first 16 spikes are significant, thus, we can start with a NAR model with 16 lags. For brevity, we restrict ourselves to trying 16, 25, 40 and 60 lags and use (number of lags)/2 as the number of hidden layer neurons. Using regularization should keep the weights of the network from getting too big. Alas, there appear to only be rules of thumb for choosing the number of hidden units and no concrete method.

The data were divided first in blocks (80% for training and 20% for validation), however, random division of the dataset into training and validation resulted in better performance. Random division does not seem particularly sensible a method with time series data as the observations are not IID samples but are correlated with each other.

Destroying this correlation structure is akin to then throwing the baby out with the bathwater. After selecting the number of lags, we then retrained the network by manually selecting the training and the test set indices and assessing the effectiveness of the network at approximating the target series with the holdout data. The approximated function from the fit are shown in figure 16.



(a) 16 lags (delays)

(b) 25 lags (delays)

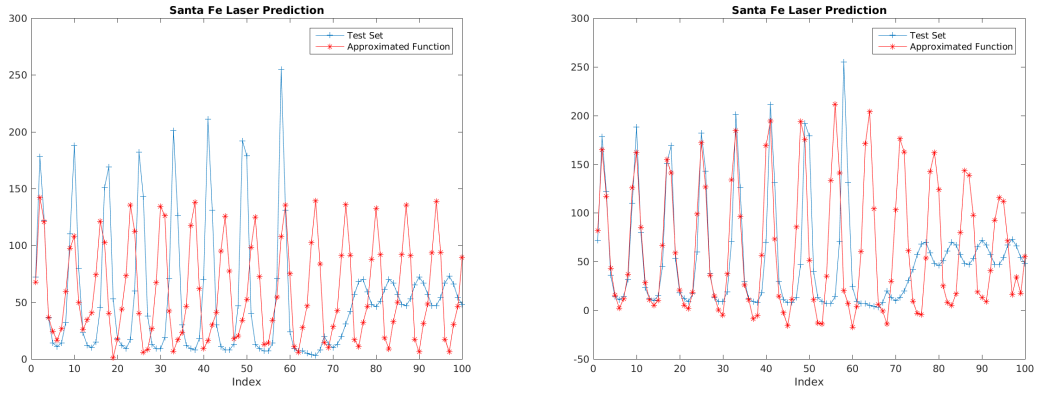
(c) 40 lags (delays)

(d) 60 lags (delays)

Figure 16: Santa Fe: approximated function fit to the holdout set. 4 different lag values were considered (16, 25, 40, 60) and hidden units = (lags/2). Regularization parameter is taken to be 10^{-6}.

Based on these graphs, we pick the 16 lags case and the 60 lags case for the manual training/validation set division and compare the accuracy of the approximated function. We took the first 80 observation (1:80, 101:180, ..., 901:980) as the training set and the last 20 as validation set (81:100, 181:200, ..., 981:1000). Other combinations could be tried as well, however manually specifying training and validation sets can lead to loss of important information from the training series. Splitting time series into training and validation sets is a non-trivial task, and one usually learns in a time series course to fit a

model by training the model based on minimizing the one-step ahead prediction errors. A plot of the approximated function for 16 and 60 lags is given in figure 17.



(a) 16 lags (b) 60 lags

Figure 17: Santa Fe dataset: Manual division of the data into training and validation sets and using 16 and 60 lags.

2.2 Alphabet recognition demo: appcr1

This demo uses a neural network (MLP) for a classification task, namely alphabet recognition. A feedforward network with a 25 hidden units in a single hidden layer and 26 nodes in the output layer (for the 26 different characters in the alphabet) was used for training the network. Two different sets of inputs (shown in figure 18 were used to train two different networks (noiseless: net1; noisy: net2), both possessing the same architecture (figure 19).

(a) Noiseless Case (b) Noisy Case

Figure 18: Alphabet recognition: Letter D without noise and with noise

The input data presented to net1 contains no noise. In the second case, the input

data presented to net2 contains a noisy version (added noise from a $N(0, 0.2^2)$) of the letter D. Thus, it seems reasonable to expect net2 to perform well on both the noiseless and noisy cases compared to net1, but net1 to perform worse on noisy data than net2's performance on noiseless data.

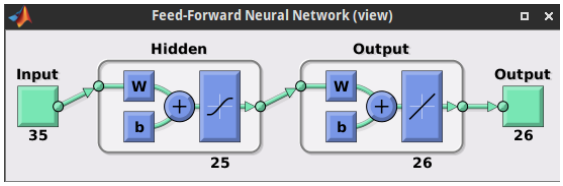


Figure 19: Architecture of the network used for alphabet recognition demo *appcr1*

The dataset is created as follows: each image is broken into a 5x7 bitmap of a letter. These 35 pixel values are added to a matrix with 26 columns. Thus, it results in a 35x26 matrix, with each column representing a letter of the alphabet, and each row corresponds to a pixel of the image. In the noiseless data, the pixels are either 0 or 1. However, in the noisy case, some of the pixels are now not exactly 0 or 1 but have some added noise. The target matrix is a 26x26 identity matrix which maps each alphabet to a class. Thus, the fifth diagonal entry will indicate the letter E, etc. Finally, the networks are tested with 20 different levels of noise $(0.05, 0.1, 0.15, \ldots, 1.0)$. A plot comparing the performances of the two networks is presented in figure 20.

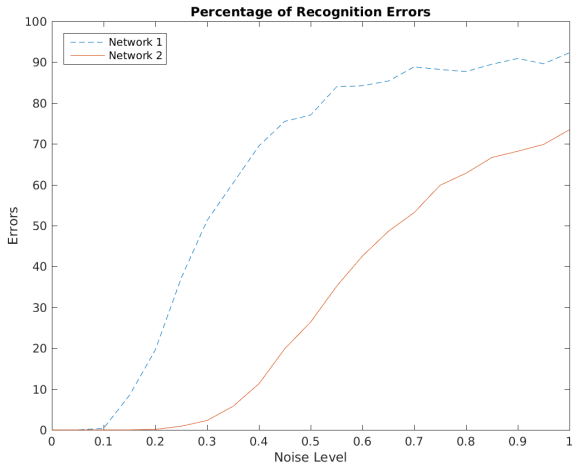


Figure 20: Character recognition: plot comparing the classification performance of the two networks on inputs with 20 different levels of noise $(0.05, 0.1, 0.15, \ldots, 1.0)$. Network 2 naturally performs better than network 1 due to being trained on noisy data.

Thus, we see from figure 20 that net2 has overall much better performance than net1. Net1 is able to handle noise upto 0.2 with an error rate less than 20%. Net2, on the other hand, has less than 20% error for noise upto σ=0.5 and has negligible errors for $\sigma \leq 0.25$ (as the network was trained on data with noise parameter 0.2). Net1 has negligible errors for $\sigma \leq 0.1$. It is a little surprising to see that despite being trained on noiseless data, network 1 is robust to a small amount of noise.

2.3 Pima Indians diabetes (classification problem)

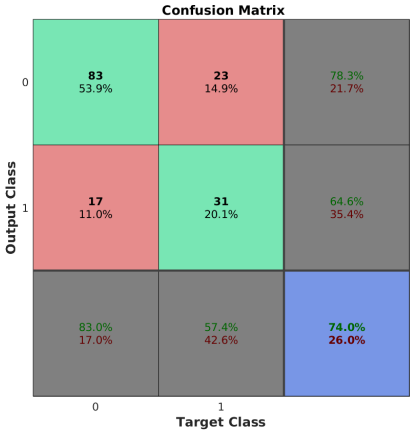
This problem is a classification problem, where a neural network will be trained as a classifier. We consider the UCI Pima Indians Diabetes dataset, which contains 768 records of female Pima Indians on 8 features. Further, the target variable has two levels: -1 (no diabetes) and 1 (diabetes). Next, we use a multilayer perceptron with one hidden layer for the classification problem at hand by considering it as a regression problem where the network is trained in the same way as regression. However, the values predicted after training the network $\notin \{0, 1\}$, but $\in [0, 1]$. Thus, we use the $Y = 0.5$ as the decision rule, where $Y > 0.5$ is classified as 1, and $Y < 0.5$ is classified as 0.

The features were linearly scaled before training, and the target variable was scaled to {0,1} from {-1,1}. We used the scaled conjugate gradient algorithm with cross-entropy as the loss function. Early stopping was used instead of regularization. Four separate networks were trained with 2, 5, 8 and 10 neurons by using 60% of the observations in the training set, and 20% for the validation and test sets respectively. The cross-entropy errors and the corresponding confusion matrices are presented below.

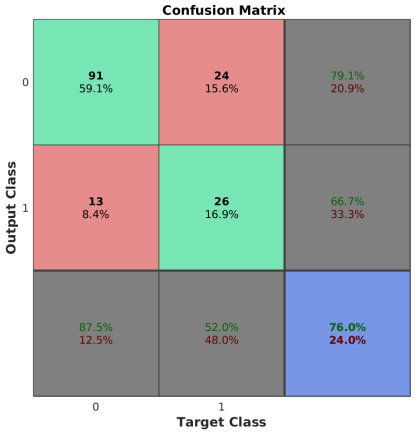
From figure 21, we see that the performance for all the networks is very similar. Each of the networks were trained multiple times (not included here) to assess the sensitivity of the classifier to random initial weights. Across all the different runs, the classifier was able to correctly classify somewhere between 70-80% of the observations in the test set. This indicates that the training algorithms keep getting stuck in local minima and is rather sensitive to random data division as well as the initial weights for the training algorithms. Since all the networks with different neurons give similar results, we can select the network with 2 or 5 neurons in the hidden layers for parsimony. To reduce the sensitivity of the model to initial weights, one can apply 10-fold cross-validation and average the weights from these models for prediction. Furthermore, since the division of the data into the training/validation/test sets were random, this could be a factor in the (relatively) large differences between different models (with the same number of neurons). The cross-validation method should lead to a robust network in this case as well.

Neurons	2	5	8	10
Error	0.4618	0.4796	0.4776	0.4639

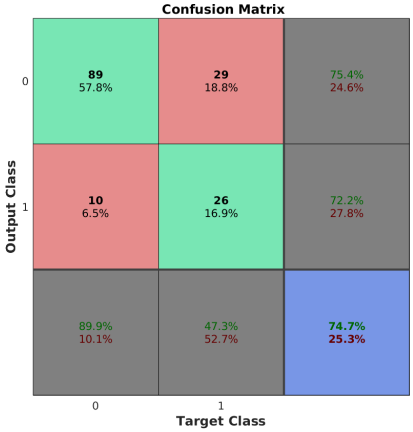
Table 3: Pima diabetes: Cross-entropy error from training the network with different number of neurons for the Pima Indians diabetes classification problem.



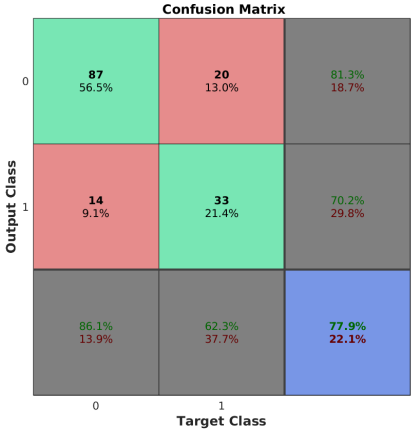
(a) 2 Neurons



(b) 5 Neurons



(c) 8 Neurons



(d) 10 Neurons

Figure 21: PIMA Classification: confusion matrix with 2, 5, 8 and 10 neurons in the hidden layer

3 Session 3

3.1 Dimensionality reduction: Principal Component Analysis

This dataset contains 21 measurements (inputs) on 264 samples. There are three target variables (namely LDL, VLDL and HDL) with multiple levels. First, we use all the 21 input variables for training the network and compare the performance of the Levenberg-Marquadt and the Bayesian regularization algorithms on the test set. Then, we transform the original input space using PCA and reduce the dimension of the input space from 21-dimensions to a 4-dimensional space (i.e., keeping the first 4 principal components).

Both the inputs and outputs were normalized before training the network. This is done to ensure that all the variables are on the same scale, and hence importance is not given to any particular variable(s) by the principal components simply due to differences in scale. The network was trained using 5 hidden units (with tanh activation function) in a single hidden layer with 3 nodes (with linear activation function) in the output layer. Early stopping rule was applied by using an independent validation set where the performance of the neural network was assessed on this validation set and the training was not stopped as long as the performance of the network on this validation set kept decreasing.

	Training set	Test set
Early Stopping	0.3036	0.2626
Bayesian Regularization	0.3103	0.2589

(a) 21 inputs

	Training set	Test set
Early Stopping	0.3908	0.2605
Bayesian Regularization	0.3436	0.2704

(b) 4 inputs

Table 4: PCA: Performance of the network with 21 inputs compared to 4 inputs and using Levenberg-Marquadt algorithm with early stopping and with Bayesian regularization

From table 4, we see that the performances of the two networks on the test set are very similar. The effective number of parameters γ with 21 inputs is 46 (out of 128), and with 4 inputs is 36 (out of 43). This indicates that a lot of the parameters are largely redundant. Additionally, since the MSE on the test set is very similar to the MSE on the training set in all cases, this means that the network has good generalization ability as it performs similarly on the test data that was not used to train the network. When the early stopping rule was used, the performance of the network was assessed on a separate validation set. However, in case of Bayesian regularization, the training and validation sets were used to train the model and the performance was assessed on the test set.

Since the network with reduced input has a much smaller number of less relevant

parameters, we can select the model with the PCA transformed inputs. As an aside, if interpretation of the model were the aim, then the model with initial inputs would most likely be selected unless the 4 principal components were easily interpretable. However, since neural networks can be "blackboxes", so to speak, we decide to go with the model with reduced inputs. Lastly, we select the model with Bayesian regularization based on the principle of parsimony in that the regularization aims at keeping the interconnection weight values small.

3.2 Input selection: Automatic Relevance Detection

Demo: demard

This demo shows how a MLP can be used for automatic relevance detection on the input variables to determine which variables are relevant for the target variable. The input variables are $X_1 \sim \mathbb{U}(0, 1)$ with small amount of Gaussian noise, X_2 a copy of X_1 with higher amount of Gaussian noise and X_3 is an IID sample from a Gaussian distribution. The target variable $\mathrm{t} = \sin(2\pi x_1) + \varepsilon$, where ε is Gaussian noise. Thus, X_1 should be really relevant for determining the target variable, X_2 less relevant and X_3 irrelevant out of these three. The function is displayed in figure 22.

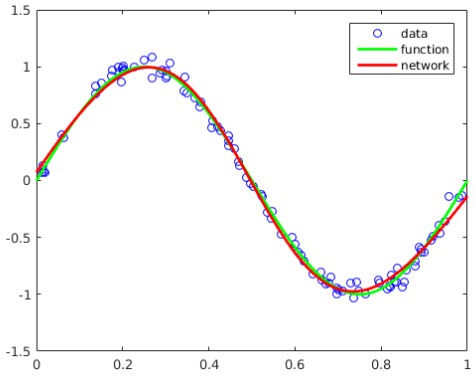


Figure 22: ARD Demo: data generating function (green) and approximated function (red)

The prior distribution over the weights is given by the ARD Gaussian prior with a separate hyper-parameter for the group of weights associated with each input. The network is trained by error minimization using scaled conjugate gradient function SCG. There are two cycles of training, and at the end of each cycle the hyper-parameters are re-estimated using the evidence.

This results in the following values for the α_i's, i.e. the hyperparameters for the corresponding input to hidden unit weights: $\alpha_1 = 0.21013$, $\alpha_2 = 1.70906$, and $\alpha_3 = 3154.38379$ after the first training cycle. After the second training cycle, the hyperparameter values

corresponding to the three inputs x_1, x_2 and x_3 are $\alpha_1 = 0.17555$, $\alpha_2 = 21.07742$ and $\alpha_3 = 153182.97363$.

Since each alpha corresponds to an inverse variance, we see that the posterior variance for weights associated with input x1 is large, that of x2 has an intermediate value and the variance of weights associated with x3 is small. The corresponding weight values are given below:

$$\begin{aligned} x_1: \quad & -3.18149 \quad 1.10024 \\ x_2: \quad & -0.24288 \quad 0.05027 \\ x_3: \quad & 0.00119 \quad -0.00048 \end{aligned}$$

We see that the network is giving greatest emphasis to x1 and least emphasis to x3, with intermediate emphasis on x2. Since the target t is statistically independent of x3 we might expect the weights associated with this input would go to zero. However, for any finite data set there may be some chance correlation between x3 and t, and so the corresponding alpha remains finite.

Demo: demev1

This demonstration illustrates the application of Bayesian re-estimation to determine the hyperparameters in a simple regression problem. It is based on a local quadratic approximation to a mode of the posterior distribution and the evidence maximization framework of MacKay. Similar to the previous demo, we generate a synthetic data set consisting of a single input variable x sampled from a Gaussian distribution, and a target variable t = sin(2πx) with additional Gaussian noise. A plot of this figure is given in 23.

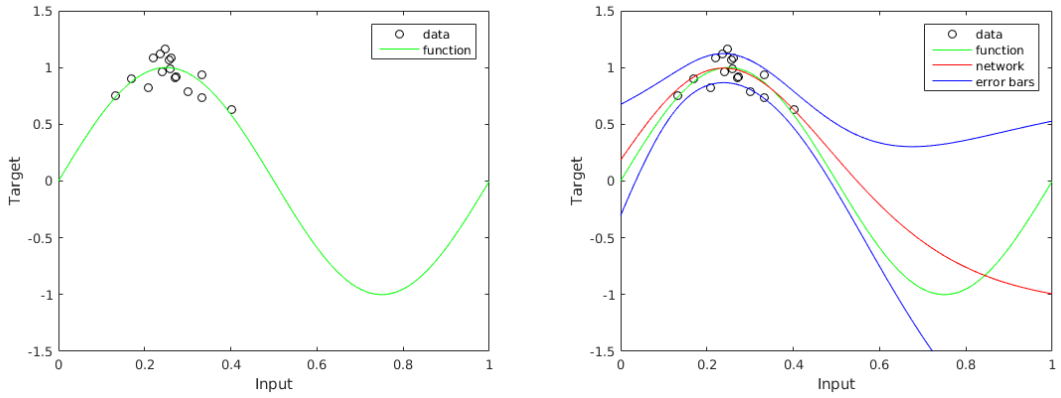


Figure 23: demev demo: available data and the true function (green) with error bars (blue) and approximated function (red)

Next we create a two-layer MLP network with 3 hidden units and one node in the output layer with a linear activation function. The model assumes Gaussian target noise governed by an inverse variance hyperparmeter β, and uses a simple Gaussian prior distribution governed by an inverse variance hyperparameter α.

The network weights and the hyperparameters are initialised (to 0.01) and then the weights are optimized with the scaled conjugate gradient algorithm, with the hyperparameters kept fixed. After a maximum of 500 iterations, the hyperparameters are re-estimated using the evidence. The process of optimizing the weights with fixed hyperparameters and then re-estimating the hyperparameters is repeated for a total of 3 cycles. The estimated α value is 0.18028 and β value is 66.96329 (compared to the true β value of 100.00).

A plot of the function represented by the trained network is given in figure 23. This corresponds to the mean of the predictive distribution. We can also plot 'error bars' representing one standard deviation of the predictive distribution around the mean. Notice how the confidence interval spanned by the 'error bars' is smaller in the region of input space where the data density is high, and becomes larger in regions away from the data.

Ionosphere data: classification task

This dataset contains 351 observations on 33 variables (inputs) and 1 binary target variable {-1,1} which is scaled to {0,1} for the binary classification task. We use Bayesian learning coupled with a sophisticated method of automatic relevance detection (ARD) for determining the inputs most relevant to the (classification) task at hand. This method is a modification of the method proposed by Foresee and Hagan (1997) for ARD, in which hyperparameters α_i are taken for each set of weights associated with the inputs to the network (33 in our case) instead of a single hyperparameter α. After training (using the scaled conjugate gradient algorithm), separate values for the α_i's are obtained for each of the corresponding x_i's in the network.

The input variables are normalized before training the network *for comparing the relative importance of the input variables on the target variable.* An added benefit of normalizing the inputs may result in faster convergence of the training algorithm. Further, scaling the target variable to the unit interval [0,1] and using a logistic activation function on the output layer does not require estimation of the hyperparameter β and the reduced cost function to be optimized is $M(w) = -G(w) + \alpha E_W(w)$, where $G(w)$ is the usual log-likelihood function and the loss function employed is the cross-entropy loss.

The dataset was split into training (67%) and test sets (33%). Initially, an MLP with 33 inputs and a single hidden layer with 2 hidden units was trained on the given data. Before training, the hyperparameters α_i's for the weights of the first hidden unit are initialized to 0.01 and the same is done for the weights in the second hidden unit as well

as the bias terms for both hidden units. Then, the α_i's are estimated iteratively from the data.

Based on the α_i's (using cut-off as < 100 which corresponds to a variance 0.01 as alpha corresponds to inverse variance), we select inputs 8,19,21,22,26 and 33 and retrain the network. In other words, the larger the posterior variance of the inputs, the more relevant that input is at predicting the target variable.

The ROC curves for the full and reduced models are given in figures 24(a) and 24(b) respectively, and the confusion matrices for the models are given in figures 25(a) and 25(b) respectively.

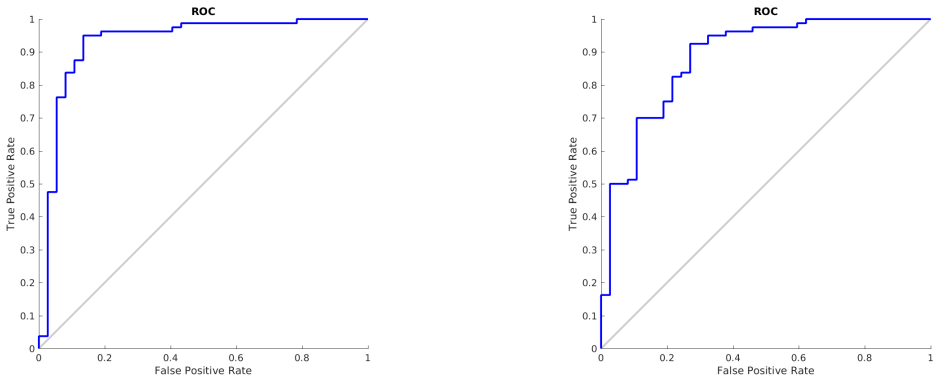


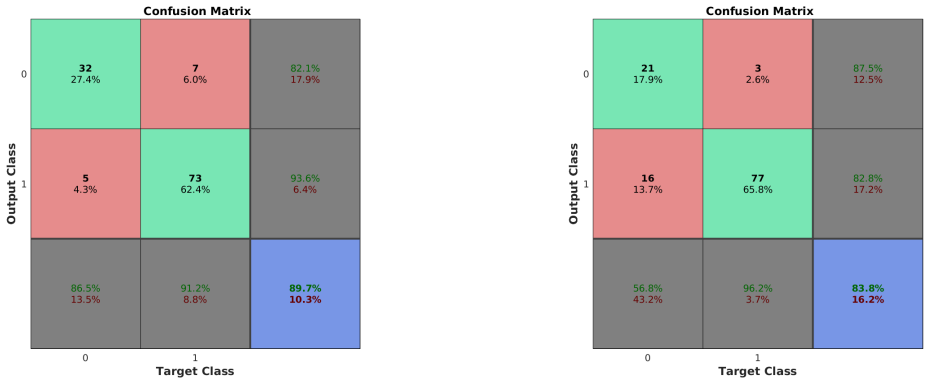
Figure 24: Ionosphere: ROC curves for the full model (Left) and reduced model (Right)

Based on the ROC curves, the reduced model seems to be performing well, but clearly not as well as the full model. As for the confusion matrices, the full model was able to correctly classify 90% of the observations in the test set, compared to 84% classification rate for the reduced model on the test set. Thus, we see that reducing the model from 33 inputs to 6 inputs still leads to quite a good classifier.

Finally, the performance of the reduced model could be improved further by the stepwise removal of less relevant predictors from the full model and retraining the model each time instead of removing a large number of predictors in one step, as was done for this model. k-fold cross-validation could be employed for improving the performance of both the full and reduced models instead of using a fixed training and test sets as was done above.

References

[1] `http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-10.html`



(a) 89.8% correctly classified (b) 83.8% correctly classified

Figure 25: Ionosphere: Confusion matrices for the full model (Left) and reduced model (Right)

4 Appendix (MATLAB Code)

Session 1

```
%% (1.1) Function Approximation (noiseless case)
clear;
clc;
nnd11gn

%% (1.2) Role of the hidden layer and the output layer
clear;
clc;

x = linspace(0,1,21);
y1 = sin(0.7 * pi * x);
plot(x, y1, 'b-*');
title('y = sin(0.7\pix) with x \in [0,1]');
xlabel('x');
ylabel('y');
%print('\home\ad\Desktop\images\sinpix', '-dpng');

%% training network with 1 hidden layer and 2 neurons
net = fitnet(2);
net = configure(net, x, y1);
net.inputs{1}.processFcns = {};
net.outputs{2}.processFcns = {};
[net, ~] = train(net, x, y1);
```

```
24
25 %% layer weights and biases; activation functions
26 % hidden layer with two neurons
27 [biases, weights] = hidden_layer_weights(net);
28 transfer_function = hidden_layer_transfer_function(net);
29 str2func(net.layers{1}.transferFcn) % hidden layer
30
31 % output layer with single node
32 [biases_out, weights_out] = output_layer_weights(net);
33 transfer_out = output_layer_transfer_function(net);
34 str2func(net.layers{end}.transferFcn) % output layer
35
36 %% Calculations using the previous section
37 p = 1:21;
38 x1_train = x * weights(1) + biases(1);
39 x2 = x * weights(2) + biases(2);
40
41 x1p = transfer_function(x1_train);
42 plot(p, x1p, '*');
43 xlabel('p');
44 ylabel('x_p^1');
45 %print('\home\ad\Desktop\images\x1p', '-dpng');
46
47 x2p = transfer_function(x2);
48 plot(p, x2p, '*');
49 xlabel('p');
50 ylabel('x_p^2');
51 %print('\home\ad\Desktop\images\x2p', '-dpng');
52
53 plot(p, x1p + x2p, '*');
54 xlabel('p');
55 ylabel('y_p');
56 %print('\home\ad\Desktop\images\yp', '-dpng');
57
58 output_p = transfer_out(weights_out(1) * x1p + weights_out(2) *
       x2p + biases_out);
59 plot(x, output_p, 'r*-');
60 title('Plot of output_p vs input vector x');
61 ylabel('output_p');
62 xlabel('x');
63 %print('\home\ad\Desktop\images\outputp', '-dpng');
64
65 %% (1.3) Function Approximation (noisy case)
66 clear;
```

```
67  clc;
68
69  %% Fixing the initial seed for the rng
70  rng(1,'twister');
71  s = rng;
72
73  %% Set parameter options; create a training and validation set
74  data_size = 100000; % 50, 200, 1000, 100000 (final part for
        comparing algorithms)
75  std_dev = 1.0; % 0.1, 1.0, 2.0
76  train_algorithm = 'trainbfg'; % trainscg, trainlm, trainbfg
        trainrp
77  regularization_parameter = 1e-3; % in [0,1], 0.8, 0.1, 1e-4
78  neurons = 40; % 5, 10, 20, 40
79
80  %% Create dataset
81  rng(s); % cuts down on the randn variability
82  train_x = linspace(-1, 1, data_size);
83  train_y = sin(2 * pi * train_x) + (std_dev * randn(size(train_x
        )));
84  val_x = linspace(-0.9, 0.9, data_size);
85  val_y = sin(2 * pi * val_x) + (std_dev * randn(size(val_x)));
86  noise_x = [train_x val_x];
87  noise_y = [train_y val_y];
88
89  %% fit the model
90  net = fitnet(neurons, train_algorithm);
91
92  % early stopping or regularization
93  net.divideFcn = 'dividetrain'; % 'dividetrain' for comparing
        the algorithm runtimes, 'divideind' for others
94  net.divideParam = struct('trainInd', 1:data_size, ...
95      'valInd', (data_size + 1):(data_size * 2), ...
96      'testInd', []); % no test set
97  %net.performParam.regularization = regularization_parameter;
98  net.trainParam.epochs = 50; % 50, 200 for assessing training
        times
99
100 %training the network
101 [net, tr] = train(net, noise_x, noise_y);
102
103 %% Get approximated function on training set
104 title_string = strcat('nodes=', num2str(neurons), {', '}, '\
        sigma=', num2str(std_dev), ...
```

```
105     {', '}, 'nobs=', num2str(data_size), {', '}, '\lambda=',
            num2str(regularization_parameter), ...
106     {', '}, 'alg=', train_algorithm);
107 train_y_hat = net(train_x);
108 plot(train_x, train_y, '*');
109 hold on;
110 plot(train_x, train_y_hat, 'r+-', 'LineWidth', 0.5);
111 plot(train_x, sin(2 * pi * train_x), 'g*-', 'LineWidth', 1);
112 hold off;
113 title(title_string);
114 legend('Training Set', 'Approximated Function', 'True Function'
        );
115 xlabel('x');
116 ylabel('sin(2\pix) + noise');
117 %print('\home\ad\Desktop\images\o', '-dpng');
118
119 %% (1.4) Curse of dimensionality
120 clear;
121 clc;
122
123 %% m = 1
124 x1_train = linspace(-5,5,100);
125 x1_test = linspace(-4.9,4.9,53);
126
127 intersect(x1_train, x1_test); % test if there is overlap
        between the training and test sets
128
129 y1_train = sinc(x1_train);
130 y1_test = sinc(x1_test);
131
132 x1 = [x1_train, x1_test];
133 y1 = [y1_train, y1_test];
134
135 plot(x1_train,y1_train, '*-', 'MarkerSize', 6);
136 hold on;
137 plot(x1_test,y1_test, 'r+-', 'MarkerSize', 10);
138 hold off;
139 xlabel('x \in [-5,5]');
140 ylabel('sinc(x)');
141 title('Sinc Function, m = 1')
142 legend('Training set', 'Test set');
143 print('\home\ad\Desktop\images\sinc1', '-dpng');
144
145 %% fit net 1
```

```
146 net = fitnet(5); % 5
147 net.layers{1}.transferFcn ='radbas';
148 net.divideFcn = 'dividetrain';
149 % net.divideParam = struct('trainInd', 1:100, ...
150 %     'valInd', [], ...
151 %     'testInd', []); % no validation set
152 %net.performParam.regularization = 1e-6;
153
154 [net, tr] = train(net, x1_train, y1_train);
155 %perf = perform(net, x1, y1);
156 %perf_test = perform(net, x1_test, y1_test);
157
158 % approximated function
159 plot(x1_train,y1_train, '-'); % training
160 hold on;
161 plot(x1_train, net(x1_train), '-r', 'LineWidth', 1);
162 plot(x1_test, net(x1_test), '--g', 'LineWidth', 2);
163 hold off;
164 legend('Training Set', 'Fitted Func.', 'Test Set Fit');
165 xlabel('x');
166 ylabel('y = sinc(r)');
167 %print('\home\ad\Desktop\images\sinc1log', '-dpng');
168 %print('\home\ad\Desktop\images\sinc1radbas', '-dpng');
169
170 %% Curse of Dimensionality (contd.)
171 %% m = 2
172 [X1,X2] = meshgrid(linspace(-5,5,100));
173 R = sqrt(X1.^2 + X2.^2);
174 Z = sinc(R);
175
176 figure
177 mesh(X1, X2, Z)
178 %surf(X1, X2, Z)
179 xlabel('x1 \in [-5,5]');
180 ylabel('x2 \in [-5,5]');
181 zlabel('z = sinc(r)');
182 title('Sinc Function, m = 2');
183 %print('\home\ad\Desktop\images\sinc2', '-dpng');
184
185 %% Training, validation and test sets
186 train_x = [X1(:), X2(:)].';
187 temp1 = train_x.*train_x;
188 train_z = sinc(sqrt((temp1(1,:) + temp1(2,:))));
189
```

```
[val1,val2]=meshgrid(linspace(-4.9,4.9,53));
val_x = [val1(:), val2(:)].';
temp2 = val_x.*val_x;
val_z = sinc(sqrt((temp2(1,:) + temp2(2,:))));

% for training the network
inputs = [train_x val_x];
target = [train_z val_z];

[test1, test2] = meshgrid(linspace(-4.8,4.8,40));
test_x = [test1(:), test2(:)].';
temp3 = test_x.*test_x;
test_z = sinc(sqrt((temp3(1,:)+temp3(2,:))));

%% fit net 2
net = fitnet(25, 'trainlm'); % 25, 30, 35
net.layers{1}.transferFcn ='radbas';
net.divideFcn = 'divideind';
net.divideParam = struct('trainInd', 1:10000, ...
    'valInd', 10001:12809, ...
    'testInd', []); % no test set at the time of training
%net.performParam.regularization = 1e-2;
net.trainParam.epochs = 4000;

[net, tr] = train(net, inputs, target, 'useParallel', 'yes');

%% Mesh plot
Y_hat = net(test_x);

plot3(test_x(1,:),test_x(2,:),test_z,'b*', 'MarkerSize',1);
title('Mexican Hat Function')
xlabel('x1 \in [-4.8,4.8]')
ylabel('x2 \in [-4.8,4.8]')
zlabel('z = sinc(r)')
hold on;
plot3(test_x(1,:),test_x(2,:),Y_hat,'r--');
legend({'Test Set','Approx. Function'});
hold off;
%print('\home\ad\Desktop\images\sinc2fit35', '-dpng');

%% Plot by dimension
Y_hat = net(test_x);

figure;
```

```
234 subplot(1,2,1)
235 plot(test_x(1,:), test_z, 'o', test_x(1,:), Y_hat, '*');
236 xlabel('x1');
237 ylabel('Z');
238 title('Dimension 1');
239 legend('Test set', 'Approximated', 'Location', 'southeast');
240 subplot(1,2,2)
241 plot(test_x(2,:), test_z, 'o', test_x(2,:), Y_hat, '*');
242 xlabel('x2');
243 ylabel('Z');
244 title('Dimension 2');
245 legend('Test set', 'Approximated', 'Location', 'southeast');
246 %print('\home\ad\Desktop\images\sinc2point25', '-dpng');
247 display(tr.best_perf)
248 perf_test = perform(net, test_z, Y_hat)
249
250 %% m = 5
251 [X1,X2,X3,X4,X5] = ndgrid(linspace(-5,5,15));
252 train_x = [X1(:), X2(:), X3(:), X4(:), X5(:)].';
253 temp1 = train_x.*train_x;
254 train_z = sinc(sqrt((temp1(1,:) + temp1(2,:) + temp1(3,:) +
       temp1(4,:) + temp1(5,:))));
255
256 [val1,val2, val3, val4, val5] = ndgrid(linspace(-4.9,4.9,10));
257 val_x = [val1(:), val2(:), val3(:), val4(:), val5(:)].';
258 temp2 = val_x.*val_x;
259 val_z = sinc(sqrt((temp2(1,:) + temp2(2,:) + temp2(3,:) + temp2
       (4,:) + temp2(5,:))));
260
261 % for training the network
262 inputs = [train_x val_x];
263 target = [train_z val_z];
264
265 [test1, test2, test3, test4, test5] = ndgrid(linspace
       (-4.8,4.8,10));
266 test_x = [test1(:), test2(:), test3(:), test4(:), test5(:)].';
267 temp3 = test_x.*test_x;
268 test_z = sinc(sqrt((temp3(1,:) + temp3(2,:) + temp3(3,:) +
       temp3(4,:) + temp3(5,:))));
269
270 %% fit net 2
271 net = fitnet(300, 'trainscg'); % 100, 200, 300, 400, 500
272 net.layers{1}.transferFcn ='radbas';
273 net.divideFcn = 'divideind';
```

```
net.divideParam = struct('trainInd', 1:(15^5), ...
    'valInd', (15^5)+1:length(inputs), ...
    'testInd', []); % no test set at the time of training
%net.performParam.regularization = 1e-2;
net.trainParam.epochs = 500;

[net, tr] = train(net, inputs, target, 'useParallel', 'yes');

Y_hat = net(test_x);
format long
display(tr.best_perf)
perf_test = perform(net, test_z, Y_hat)
```

Session 2

```
%% 2.1 Santa Fe time series prediction
clear;
clc;

% reading in the training set
fid = fopen('/home/ad/Desktop/KUL Course Material/Data Mining And Neural Networks/MATLAB/Exercises/lasertrain.dat','r');
datacell = textscan(fid, '%f', 'Delimiter','\n', 'CollectOutput', 1);
fclose(fid);
A.data = datacell{1};
santafe = A.data;

% reading in the prediction set
fid = fopen('/home/ad/Desktop/KUL Course Material/Data Mining And Neural Networks/MATLAB/Exercises/laserpred.dat','r');
datacell = textscan(fid, '%f', 'Delimiter','\n', 'CollectOutput', 1);
fclose(fid);
B.data = datacell{1};
santafe_pred = B.data;

% santafe = tonndata('/home/ad/Desktop/KUL Course Material/Data Mining And Neural Networks/MATLAB/Exercises/lasertrain.dat', false, false)
% santafe_pred = tonndata('/home/ad/Desktop/KUL Course Material/Data Mining And Neural Networks/MATLAB/Exercises/laserpred.dat', false, false)
```

```
21
22 %% plotting both side by side
23 figure
24
25 subplot(1,2,1)
26 plot(santafe, '-');
27 title('Training Set');
28
29 subplot(1,2,2)
30 plot(santafe_pred, 'r-');
31 title('Test Set');
32
33 print('\home\ad\Desktop\images\santafe', '-dpng');
34
35 figure;
36 subplot(2,1,1, 'align')
37 autocorr(mapminmax(santafe), 500) %acf
38 subplot(2,1,2, 'align')
39 parcorr(mapminmax(santafe), 500) %pacf
40 print('\home\ad\Desktop\images\santafe_acf', '-dpng');
41
42 %% Setting up the network
43 lags = 20; % 16, 25, 40, 60
44 %neurons = round(lags/2); % rule of thumb
45 neurons = 2*lags;
46 train_alg = 'trainscg'; % trainscg, trainbfg, trainlm, trainbr
47
48 % converting into a form which can be used by the narnet
     function
49 train_data = con2seq(santafe');
50 test_data = con2seq(santafe_pred');
51
52 % fitting the model, model to be trained in feedforward mode
53 net = narnet(1:lags, neurons, 'open', train_alg); %'open' (
     default)- feedforward 'closed'- recurrent
54
55 net.performParam.regularization = 1e-7;
56 net.trainParam.epochs = 1000; % 2000
57 net.performFcn = 'mse';  % 'mse', 'mae'; info: use help
     nnperformance
58
59 %net.divideFcn = 'divideblock'; % not splitting the data into
     train/validation because time series
60 % but using narnet has net.divideMode = 'time' so it's cool?
```

```
% setting up the training and validation sets

%[trainInd,valInd] = divideint(1000, 0.8, 0.2);
% splitting the set into 70% for training and 30% for
    validation
% trainInd = [1:70, 101:170, 201:270, 301:370, 401:470,
    501:570, 601:670, 701:770, 801:870, 901:970];
% valInd = [71:100, 171:200, 271:300, 371:400, 471:500,
    571:600, 671:700, 771:800, 871:900, 971:1000];
% net.divideFcn = 'divideind';
% net.divideParam = struct('trainInd', trainInd, 'valInd',
    valInd, 'testInd', []);
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 30/100;
net.divideParam.testRatio = 0/100;

%% Training the network
[Xs,Xi,Ai,Ts] = preparets(net,{},{},train_data);
[net, tr] = train(net,Xs,Ts,Xi,Ai);

Y = net(Xs,Xi,Ai);
perf = perform(net,Ts,Y)

train_errors = cell2mat(gsubtract(Ts,Y));

% closing the loop and doing 100 multi-step ahead predictions
netc = closeloop(net);
netc.name = [net.name ' - Closed Loop'];
Y_hat = [train_data((1000-lags+1):1000) num2cell(nan(100,1))];

[xc,xic,aic,tc] = preparets(netc,{},{},Y_hat);
Y_hat = netc(xc,xic,aic);

% calculating the residuals and the MAE
test_residuals = gsubtract(test_data,Y_hat);
mae = 0.01 * sum(abs(cell2mat(test_residuals))) % 0.01 = 1/100;
    formula uses 1/N

% Saving the predicted function
figure;

% subplot(1,3,1, 'align')
% autocorr(train_errors, 100) %acf
%
```

```
100 % subplot(1,3,2, 'align')
101 % parcorr(train_errors, 100) %pacf
102 %
103 % subplot(1,3,3, 'align')
104 plot(santafe_pred, '-+');
105 hold on;
106 plot(cell2mat(Y_hat), 'r-*');
107 hold off;
108 xlabel('Index');
109 title('Santa Fe Laser Prediction');
110 legend('Test Set', 'Approximated Function');
111
112 %print('\home\ad\Desktop\images\santafe_pred60', '-dpng');
113 %print('\home\ad\Desktop\images\manual_lag60', '-dpng');
114
115 %% 2.2 alphabet recognition (playing with the demo file
    inlcuded)
116 % appcr1 % using neural network for alphabet recognition
117
118 %% 2.3 Classification Problem (Pima Indian Diabetes)
119 clear;
120 clc;
121
122 load('/home/ad/Desktop/KUL Course Material/Data Mining And
    Neural Networks/Final exam/pidstart.mat', '-mat')
123
124 %% Training the network
125 [inputs, std_input] = mapstd(Xnorm'); % normalizing the input
    variables
126 target = hardlim(Y)'; % converting from [-1,1] to [0,1]
127
128 neurons = 10; % 2, 5, 8, 10 (check ROC print statement below as
     well!)
129
130 net = patternnet(neurons);
131
132 % divide the data into training, validation and test sets
133 net.divideParam.trainRatio = 60/100;
134 net.divideParam.valRatio = 20/100;
135 net.divideParam.testRatio = 20/100;
136
137 [net, tr] = train(net, inputs, target);
138 perf = tr.best_vperf
139 %view(net)
```

```
140
141 %% evaluating the network on the test set
142 testX = inputs(:, tr.testInd);
143 testT = target(:, tr.testInd);
144
145 testY = net(testX);
146 testClass = testY > 0.5;
147
148 plotroc(testT, testY) % ROC for the test set
149 print('\home\ad\Desktop\images\pima_roc_10', '-dpng');
150
151 plotconfusion(testT, testY) % confusion matrix for the test set
152 print('\home\ad\Desktop\images\pima_confusion_10', '-dpng');
153
154 % overall percentage of (in)correct classification on the test
    set
155 [c,cm] = confusion(testT,testY)
156
157 fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-
    c));
158 fprintf('Percentage Incorrect Classification : %f%%\n', 100*c);
```

Session 3

```
1 %% 3.1 Dimensionality Reduction By PCA
2 clear;
3 clc;
4
5 % the same code is run multiple times with 'trainlm' and '
    trainbr', and
6 % with pn (all inputs) and pp (4 inputs)
7
8 %% Load the cholesterol data (comes with matlab)
9 % This will create a 21x264 choInputs matrix of 264 input
    patterns
10 % and a 3x264 matrix choTargets of output patterns
11 %doc cho_dataset % dataset details
12 load cho_dataset
13
14 %% Standardize the variables
15 % doc mapstd;
16 [pn, std_p] = mapstd(choInputs);
17 [tn, std_t] = mapstd(choTargets);
```

```
18
19 %% PCA
20 %doc processpca;
21 [pp, pca_p] = processpca(pn, 'maxfrac', 0.001); %reduces data
       from 21 dimensions to 4 dimensions
22 [m, n] = size(pp)
23
24 %% Set indices for test, validation and training sets
25 Test_ix = 2:4:n;
26 Val_ix = 4:4:n;
27 Train_ix = [1:4:n 3:4:n];
28
29 %% Configure a network
30 % compare between LM and bayesian regularization (trainbr) (
       adjust the
31 % nodes in hidden layer appropriately)
32 net = fitnet(5, 'trainbr'); % trainlm and trainbr
33
34 net.divideFcn = 'divideind';
35 net.divideParam = struct('trainInd', Train_ix, ...
36 'valInd', Val_ix, ...
37 'testInd', Test_ix);
38 [net, tr] = train(net, pp, tn); % pn - original data; pp -
       reduced data
39
40 %% Get predictions on training and test
41 % pn - original data; pp - reduced data
42 Yhat_train = net(pp(:, Train_ix));
43 Yhat_test = net(pp(:, Test_ix));
44 perf_train = perform(net, tn(:, Train_ix), Yhat_train);
45 perf_test = perform(net, tn(:, Test_ix), Yhat_test);
46
47 %% 3.2 Automatic Relevance Detection
48 % In order to apply ARD, download the Netlab software from
49 % http://www.ncrg.aston.ac.uk/netlab/ or use the following link
       to
50 % get to the download's page directly: http://cl.ly/Rw0P
51
52 clear; % clear workspace
53 clc; % clear console
54
55 %% (a) run demo demard
56 demard
57
```

```
58 %% (b) run demo demev1
59 demev1
60
61 %% (c) UCI ionosphere data classification using MLP and ARD
62 % code in this section is taken from demard.m with appropriate
63 % modifications where necessary
64 load('/home/ad/Desktop/KUL Course Material/Data Mining And
       Neural Networks/Final exam/ionstart.mat', '-mat')
65 [inputs, std_input] = mapstd(Xnorm); % normalizing the input
       variables
66 targets = hardlim(Y); % converting from [-1,1] to [0,1] thus no
        need to find hyperparameter beta
67
68 train_ind = [1:6:351 2:6:351 4:6:351 5:6:351]; % 67% of the
       data used as training
69 test_ind = [3:6:351 6:6:351]; % 33% of the data used as a test
       set
70
71 %% Set up network parameters.
72 nin = 33;                          % Number of inputs.
73 nhidden = 2;                       % Number of hidden units.
74 nout = 1;                          % Number of outputs.
75 aw1 = 0.01*ones(1, nin);           % First-layer ARD
       hyperparameters.
76 ab1 = 0.01;                        % Hyperparameter for hidden
       unit biases.
77 aw2 = 0.01;                        % Hyperparameter for second-
       layer weights.
78 ab2 = 0.01;                        % Hyperparameter for output
       unit biases.
79 %beta = 50.0;                      % Coefficient of data error.
80 % do not need to estimate beta since the likelihood function is
        taken to be
81 % the cross-entropy function
82
83 %% Create and initialize network.
84 prior = mlpprior(nin, nhidden, nout, aw1, ab1, aw2, ab2);
85 net = mlp(nin, nhidden, nout, 'logistic', prior);%, beta);
86 % logistic because binary classification problem
87
88 %% Set up vector of options for the optimiser.
89 nouter = 10;                       % Number of outer loops
90 ninner = 10;                       % Number of inner loops
91 options = zeros(1,18);             % Default options vector.
```

```
92  options(1) = 1;                    % This provides display of
       error values.
93  options(2) = 1.0e-7;    % This ensures that convergence must
       occur
94  options(3) = 1.0e-7;
95  options(14) = 1000;                % Number of training cycles in
       inner loop.
96
97  %% Train using scaled conjugate gradients, re-estimating alpha
       and beta.
98  for k = 1:nouter
99    net = netopt(net, options, inputs(train_ind,:), targets(
         train_ind), 'scg');
100   [net, gamma, logev] = evidence(net, inputs(train_ind,:),
         targets(train_ind), ninner);
101   clc;
102 end
103
104 %% Selecting Parameters
105 i = 1:nin;
106 val = net.alpha(1:nin,:);
107 fprintf(1, '  alpha%i = %8.5f\n', [i; val']);
108 %fprintf(1, '  beta  =  %8.5f\n', net.beta);
109 fprintf(1, '  gamma = %8.5f\n\n', gamma);
110 fprintf(1, '  log-likelihood = %8.5f\n\n', logev);
111
112 %% Display the weights for each of the inputs
113 %weight_table = table(index, net.w1(:,1)', net.w1(:,2)');
114 disp('This is confirmed by looking at the corresponding weight
       values:')
115 disp(' ');
116 fprintf(1, '     x%i:     %8.5f     %8.5f\n', [i; net.w1']);
117 disp(' ');
118
119 %% Predictions from the model and comparing with the test set
120 [pred, z] = mlpfwd(net, inputs(test_ind, :));
121
122 %% Plot confusion matrix and ROC curves for the network with
       all inputs
123 plotconfusion(targets(test_ind)', pred');
124 print('\home\ad\Desktop\images\ion_pred_full_confusion', '-dpng
       ');
125
126 plotroc(targets(test_ind)', pred');
```

```
127 print('\home\ad\Desktop\images\ion_pred_full_roc', '-dpng');
128
129 %log-likelihood =  -9646.29802
130
131 %% Training the network using a smaller number of inputs
132 clear;
133 clc;
134
135 load('/home/ad/Desktop/KUL_Course_Material/Data_Mining_And_
    Neural_Networks/Final_exam/ionstart.mat', '-mat')
136 [inputs, std_input] = mapstd(Xnorm); % normalizing the input
    variables
137 targets = hardlim(Y); % converting from [-1,1] to [0,1] thus no
     need to find hyperparameter beta
138
139 train_ind = [1:6:351 2:6:351 4:6:351 5:6:351]; % 67% of the
    data used as training
140 test_ind = [3:6:351 6:6:351]; % 33% of the data used as a test
    set
141
142 reduced_inputs = [8 19 21 22 26 33];
143
144 % same as for the case with all inputs
145 nin = length(reduced_inputs);                    % Number of
    inputs.
146 nhidden = 2;                    % Number of hidden units.
147 nout = 1;                       % Number of outputs.
148 aw1 = 0.01*ones(1, nin);        % First-layer ARD
    hyperparameters.
149 ab1 = 0.01;                     % Hyperparameter for hidden
    unit biases.
150 aw2 = 0.01;                     % Hyperparameter for second-
    layer weights.
151 ab2 = 0.01;                     % Hyperparameter for output
    unit biases.
152
153 prior = mlpprior(nin, nhidden, nout, aw1, ab1, aw2, ab2);
154 net = mlp(nin, nhidden, nout, 'logistic', prior);%, beta);
155
156 nouter = 10;                    % Number of outer loops
157 ninner = 10;                    % Number of inner loops
158 options = zeros(1,18);          % Default options vector.
159 options(1) = 1;                 % This provides display of
    error values.
```

```
160 options(2) = 1.0e-7;    % This ensures that convergence must
    occur
161 options(3) = 1.0e-7;
162 options(14) = 1000;              % Number of training cycles in
    inner loop.
163
164 % include selected inputs instead of all in the next two lines
165 for k = 1:nouter
166   net = netopt(net, options, inputs(train_ind, reduced_inputs),
        targets(train_ind), 'scg');
167   [net, gamma, logev] = evidence(net, inputs(train_ind,
      reduced_inputs), targets(train_ind), ninner);
168   clc;
169 end
170
171 i = 1:nin;
172 val = net.alpha(1:nin,:);
173 fprintf(1, '  alpha%i =  %8.5f\n', [i; val']);
174 %fprintf(1, '  beta  =  %8.5f\n', net.beta);
175 fprintf(1, '  gamma =  %8.5f\n\n', gamma);
176 fprintf(1, '  log-likelihood =  %8.5f\n\n', logev);
177
178 % Predictions from the model and comparing with the test set (
    using only the relevant inputs)
179 [pred, z] = mlpfwd(net, inputs(test_ind, reduced_inputs));
180
181 % Plot confusion matrix and ROC curves
182 plotconfusion(targets(test_ind)', pred');
183 print('\home\ad\Desktop\images\ion_pred_reduced_confusion', '-
    dpng');
184
185 plotroc(targets(test_ind)', pred');
186 print('\home\ad\Desktop\images\ion_pred_reduced_roc', '-dpng');
187
188 %log-likelihood =  -643.50711
```