KU LEUVEN

FACULTY OF SCIENCE

Exam Report

Data Mining and Neural Networks

Daniel Gerardo GIL SANCHEZ
daniel.gilsanchez@student.kuleuven.be
MSc Statistics

Supervisor: Prof. Johan Suykens

Academic year 2018-2019

Contents

1 Exercise Session 1

1.1 Function Approximation (noiseless case)

To talk about overfitting and underfitting, it is important to define both situations first. Overfitting is when the training process is totally driven by the training error, so the network memorizes the training examples. When new data are presented, the error is going to be large because the network has not learned how to generalize to new situations. On the other hand, underfitting is when the network does not fit the training data well nor generalize to new situations.

Figure 1 shows some cases of underfitting and overfitting. In the upper left, there is a simple nonlinear function (displayed as red crosses) and a simple network with just one hidden neuron (black line). In this case, the fit is not good enough when the target function reaches either -1 or 1 in the vertical axis. The upper right plot shows a typical underfitting situation, when the fit of the function is only good in a small range in the domain of the input variable but behaves poorly outside this range. On the other hand, both plots in the bottom show overfitting in a simple nonlinear function as well as in a complex one (left and right respectively). In both situations the data are fitted (almost) perfectly, leaving no room to generalize new situations. It is important to mention that this case does not have any noise, so the fit is not wiggled as it usually is in overfitting situations.

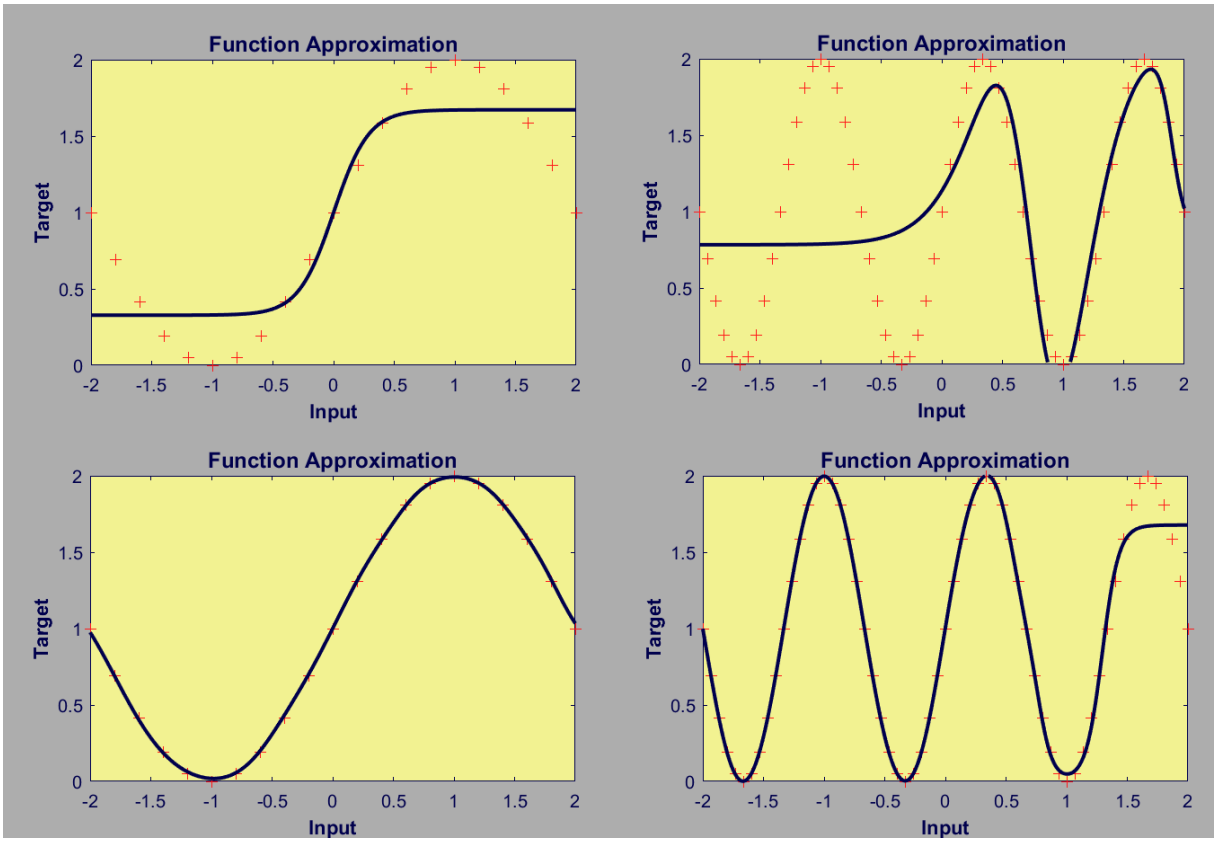


Figure 1: Underfitting (top) and Overfitting (bottom) scenarios

The reason why any of these situations appear in any model building process (not only neural networks) is because of the complexity of the model in relation with the data at hand. If the number of parameters in the model is close to the total number of observations, then it is very likely to have overfitting. Conversely, if the number of parameters in the model is much smaller that the total number of observations, then there is no chance of overfitting but

the likelihood of getting a underfitted model increases. Of course, it is difficult to know how complex a model should be to avoid both scenarios, so other methods are used to deal with these situations, such as early stopping and regularization.

1.2 The role of the hidden layer and output layer

1. *Given a set of input and output patterns $\{(x_p, y_p)\}_1^P$, how would you solve the linear regression problem using a neural network?*

   This can be seen as the McCulloch-Pitts model with the only difference that instead of using a nonlinear activation function, such as $tanh(\cdot)$, a linear activation is used (the **identity function**). This model is a weighted sum of the inputs, denoted as x in a n-dimensional space, plus a bias term that represents a threshold to activate a neuron. Following McCulloch-Pitts model for illustration, only **one hidden layer** is needed to learn a linear regression function with only **one neuron**. A representation of such neural network is:

   $$y = f(\sum_i^n w_i x_i + b) = f(a) \tag{1}$$

   Where w_i are the interconnection weights in the context of neural networks or the parameter of each variable in the context of regression. b is the bias term in neural network context or the intercept in regression jargon. x are the input patterns or covariates and y is the target or dependent variable. As it is mentioned above, f is the **identity function**, so equation 1 results in:

   $$y = \sum_i^n w_i x_i + b \tag{2}$$

   Which is similar to the equation used in regression. Figure 2 shows the architecture of a neural network that represents a linear regression[1].

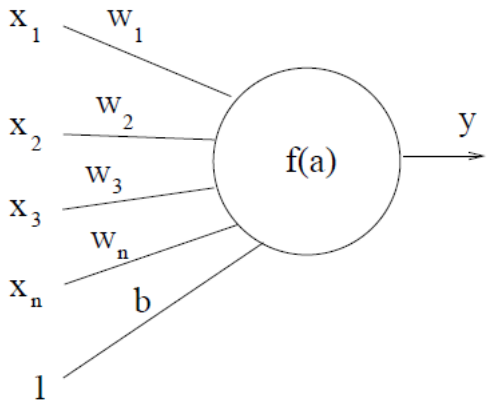


Figure 2: Neural network architecture, f is the identity function.

2. and 3. A vector of 21 observations equally spaced in the interval [0,1] is created, and the response variable is calculated as $y_p = -cos(0.8\pi x_p)$ with $p = 1, ..., 21$. In Figure 3 this relationship is plotted. There, it can be seen that a linear regression may fit the data well in the range [0.3,0.7] but not further. In fact, if the domain of the input variable was extended, it would be possible to see that the function reaches to 1 and decreases again until -1, so a linear model would not be a good idea. In this sense, if the data at hand is only in the range [0,1], a model with the architecture chosen in the previous numeral could give a decent representation. However, if data outside the range [0,1] were to be used, the model would not be useful anymore.

[1]This image is taken from the lectures of this course

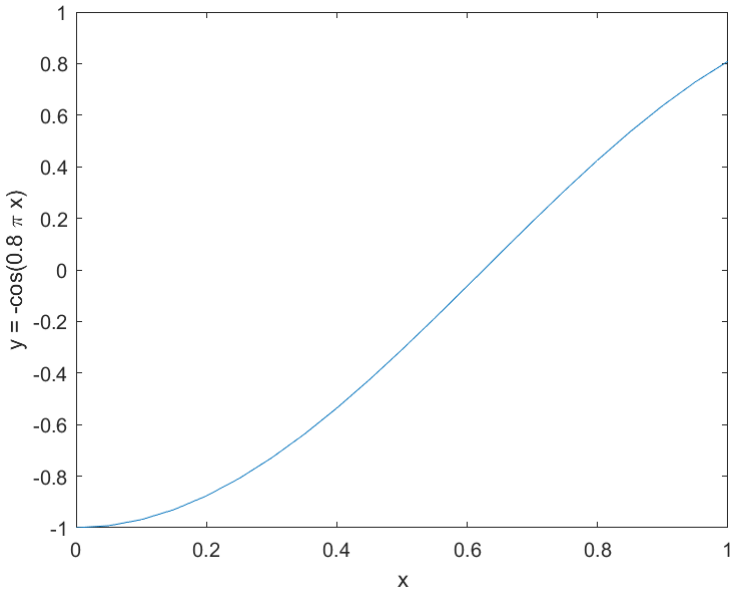


Figure 3: Relationship between input and output patterns

4. The next step is to train a neural net with one hidden layer and two neurons. The configuration of the network is changed to prevent it from rescaling the inputs and outputs. It is important to mention that the training process starts with different initial values for weights and biases and different division of the data into training, validation and test sets. For this reason, the network is trained several times to increase the likelihood of finding a good generalization and avoid local optima.

   To do so, the neural net is trained 100 times using different starting values for weights and biases and different divisions of the dataset for training, validation and testing. Each time that the neural net reaches convergence, usually by early stopping, the Mean Squared Error (MSE) is saved and then compared to keep the solution with the lowest performance value. Notice, that in order to get reproducible results the seed in the random number generator is set to 0689432, which is my student number.

5. The activation of each neuron can be computed by using the estimated weights and bias in the calculation of the linear combination of inputs and plugging this result in the activation function, as follows:

$$a = \sigma(Vx + \beta) \tag{3}$$

   Where a is a matrix with the activation value for each input pattern in each neuron. σ is the activation function. V and β are the weights and biases of the hidden neurons and x are the input patterns. In this case, σ is the *hyperbolic tangent sigmoid function* ($tansig$), V and β are:

$$V = \begin{bmatrix} 1.8896 \\ -5.2995 \end{bmatrix} \ , \ \beta = \begin{bmatrix} -1.1702 \\ -0.2583 \end{bmatrix} \tag{4}$$

   Table 1 shows these activation values.

Input	Pattern (x)	Neuron 1	Neuron 2
1	0	-0.824	-0.253
2	0.05	-0.792	-0.48
3	0.1	-0.754	-0.657
4	0.15	-0.71	-0.783
5	0.2	-0.66	-0.866
6	0.25	-0.603	-0.919
7	0.3	-0.539	-0.952
8	0.35	-0.469	-0.971
9	0.4	-0.392	-0.983
10	0.45	-0.309	-0.99
11	0.5	-0.222	-0.994
12	0.55	-0.13	-0.996
13	0.6	-0.036	-0.998
14	0.65	0.058	-0.999
15	0.7	0.151	-0.999
16	0.75	0.242	-1
17	0.8	0.329	-1
18	0.85	0.41	-1
19	0.9	0.486	-1
20	0.95	0.555	-1
21	1	0.617	-1

Table 1: Activation value in each hidden neuron

6. Figure 4 plots the values of the output, and each activation value against the input patterns. It can be seen that the first neuron follows the pattern of the output, whereas the second neuron is inversely related with it.

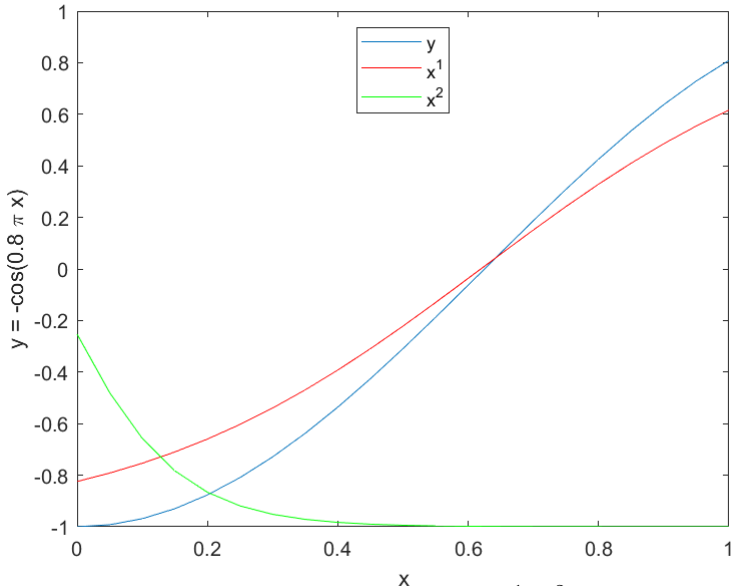


Figure 4: Output values (y) and activations (x^1, x^2) for each input pattern

7. Once the network is estimated, it is possible to calculate an estimation for the output

pattern. This is conducted by applying the following formula:

$$output = W\sigma(Vx + \beta) + \beta^* \tag{5}$$

Where σ, V, β and x are defined as in equation 3. W and β^* are the weights and bias of the output neuron, which in this case are:

$$W = \begin{bmatrix} 1.3383 \\ 0.1571 \end{bmatrix} \ , \ \beta^* = 0.1429 \tag{6}$$

Notice that the activation function of the output layer is the identity function, thus it is not expressed in equation 5. These results are shown in Figure 5 and compared with the true function. It is clear that the neural net perfectly fitted the true function.

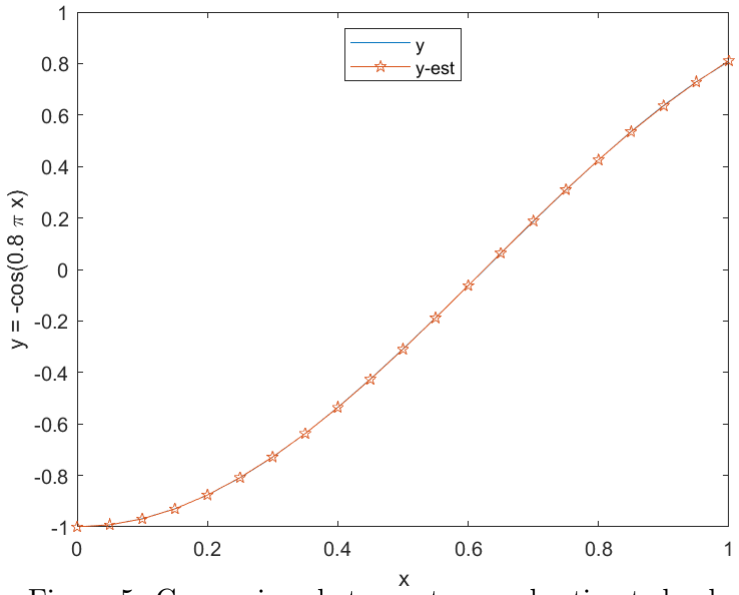


Figure 5: Comparison between true and estimated values

1.3 Function Approximation (noisy case)

A lot of different neural networks can be fitted by changing the options given in the exercise. To investigate how a neural net changes when different specifications are set, each of the options is analyzed by holding the remain options constant. For instance, to measure whether the fit of a neural net is better when more data points are used, the remain characteristics of the neural net such as the number of hidden neurons and training algorithm are constant.

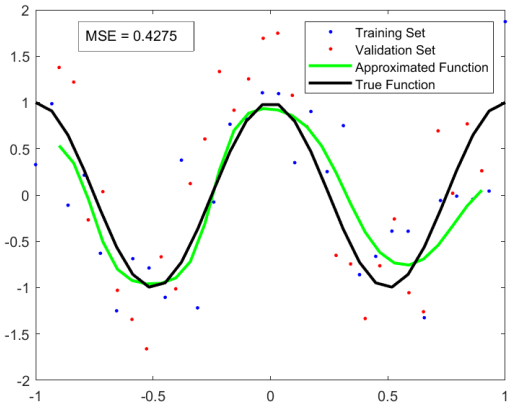
The comparison of the fit between different neural networks is conducted visually and through the Mean Squared Error (MSE) calculated on a test set. This test set is completely different and independent of the train set. In this way one can be sure that the test set in one neural net is not going to be used as a training set in other neural net.

In regard to local optima, it is well known that each time a neural net is trained, different initial weights and biases, and different divisions of the data into training, validation and test sets are considered. Thus, each neural network is trained 100 times, and the net with the lowest MSE is kept for comparison. As before, in order to get reproducible results the seed in the random number generator is set to 0689432.

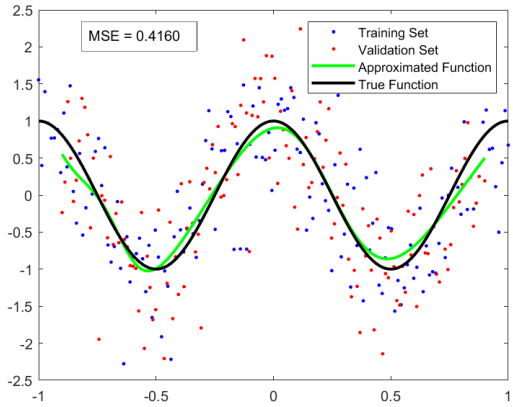
1. *Size of the training data set*

Three different sizes are tested in this comparison: 30, 150 and 1200 data points. In this case, the standard deviation is set to 0.6 to control the amount of noise, the number of hidden neurons is 5, the training algorithm is the Levenberg-Marquardt, which is the default in MATLAB, to improve generalization early stopping in used, thus no regularization is used and the selection of initial weights is at random (although a seed is set for reproducibility).

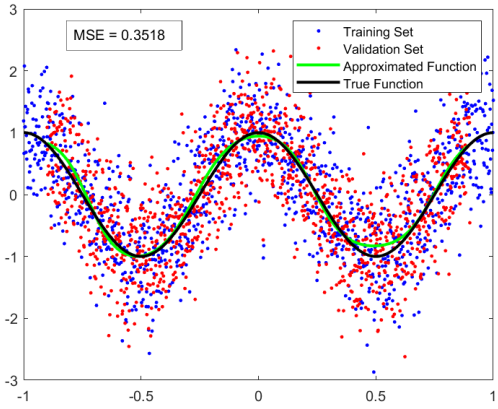
Figure 6 shows the fit and the MSE for each training dataset size. The lowest MSE is presented in the neural net with 1200 observations, where the approximated function is almost the same as the true function over the whole range of the input variable. On the other hand, in both neural nets with fewer observations, it is clear that the approximated function is very similar to the true function but, they are not as powerful in predicting new data (see MSE). This is a great example to show that more data does not necessarily lead to better estimations, since it all depends on several characteristics of the neural net and in this case the quality of the prediction was approximately the same, with way less data points (see plot 6a).



(a) N = 30



(b) N = 150



(c) N = 1200

Figure 6: Comparison of neural nets with different training sample sizes

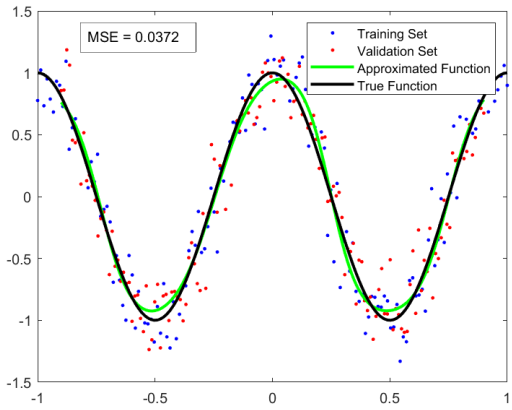
2. *Amount of noise on the training data*

   In this case three different standard deviations are considered to add noise to the cosine wave: 0.2, 0.6 and 1.2. As before, the remaining characteristics are held constant and

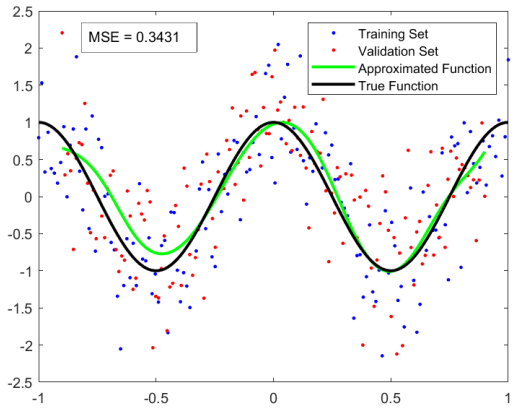
the size of the dataset is fixed at 150. Note that the choice of the sample size could be anything because the purpose of this section is to analyze the differences in the fit when different noises are added to the raw data.

Figure 7 shows the fit and the MSE for each training dataset with different noise. The lowest MSE is presented in the neural net with the smallest standard deviation. This result is expected due to the fact that there is not much variability, so it is easier to predict. On the other hand, the training data set that had a standard deviation of 1.2, presents the highest MSE and therefore is less reliable to make predictions. In regard to the visual representation, it is clear that in plot 7c the lack of fit is usually located from 0.25 onward.

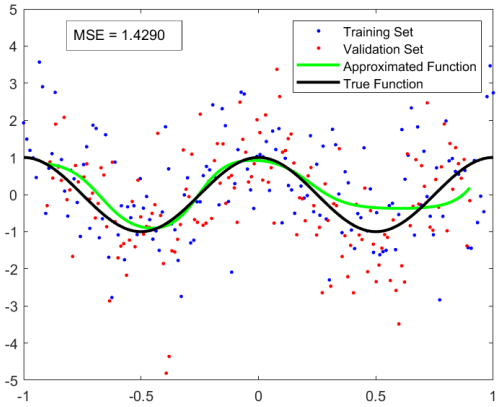
This example complements the results showed in the previous section where the fit was better when the number of input patterns were large. It actually tells that it does not really matter the number of observations but how much variance there is in them. The larger the variance, the harder it is to get a good fit.



(a) sd $= 0.2$



(b) sd $= 0.6$



(c) sd $= 1.2$

Figure 7: Comparison of neural nets with different noise in the training sample

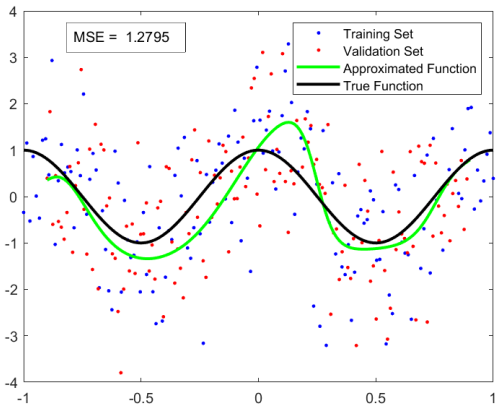
3. *Number of hidden neurons*

In this case three different scenarios are considered in regard to the number of hidden neurons: 6, 10 and 20. As before, the remaining characteristics are held constant, the size of the training dataset is 150 and the amount of noise considered is a standard

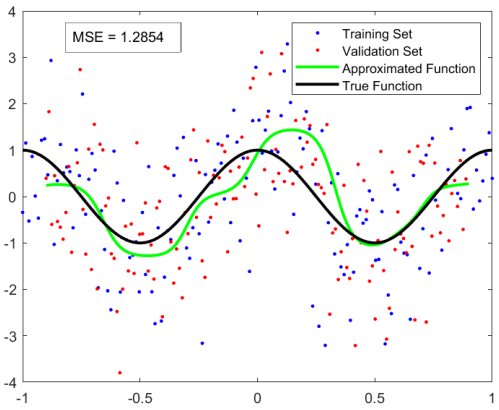
deviation of 1.2. There are two reasons to choose this amount of noise: first, because it showed the worst performance in comparison to the other two scenarios and second, because it is a more realistic example.

In regard to the number of hidden neurons, it was already shown in the previous comparison that 5 hidden neurons were not enough to get a decent representation of the true function. For this reason, the number of hidden neurons considered are larger than 5.

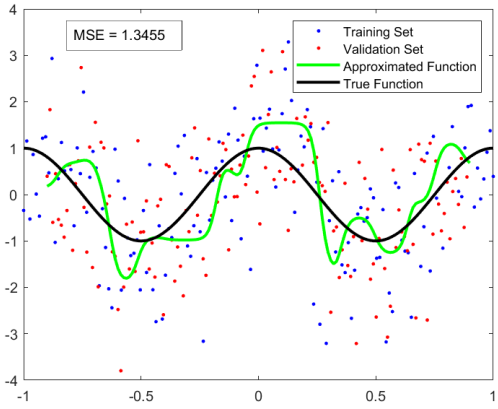
In Figure 8 the comparison is presented. The neural network that has the lowest MSE is the one that has 6 hidden neurons. Graphically, it can be seen that it is closer to the true function and it is very similar to the approximated function of the net with 10 hidden neurons. In contrast, the net with 20 hidden neurons is a clear example of overfitting for two reasons: first, the MSE is higher than in any other case and second, the approximation function is not as smooth as the true function, meaning that the net memorized the training patterns instead of having learned the general pattern.



(a) 6 hidden neurons



(b) 10 hidden neurons



(c) 20 hidden neurons

Figure 8: Comparison of neural nets with different number of hidden neurons

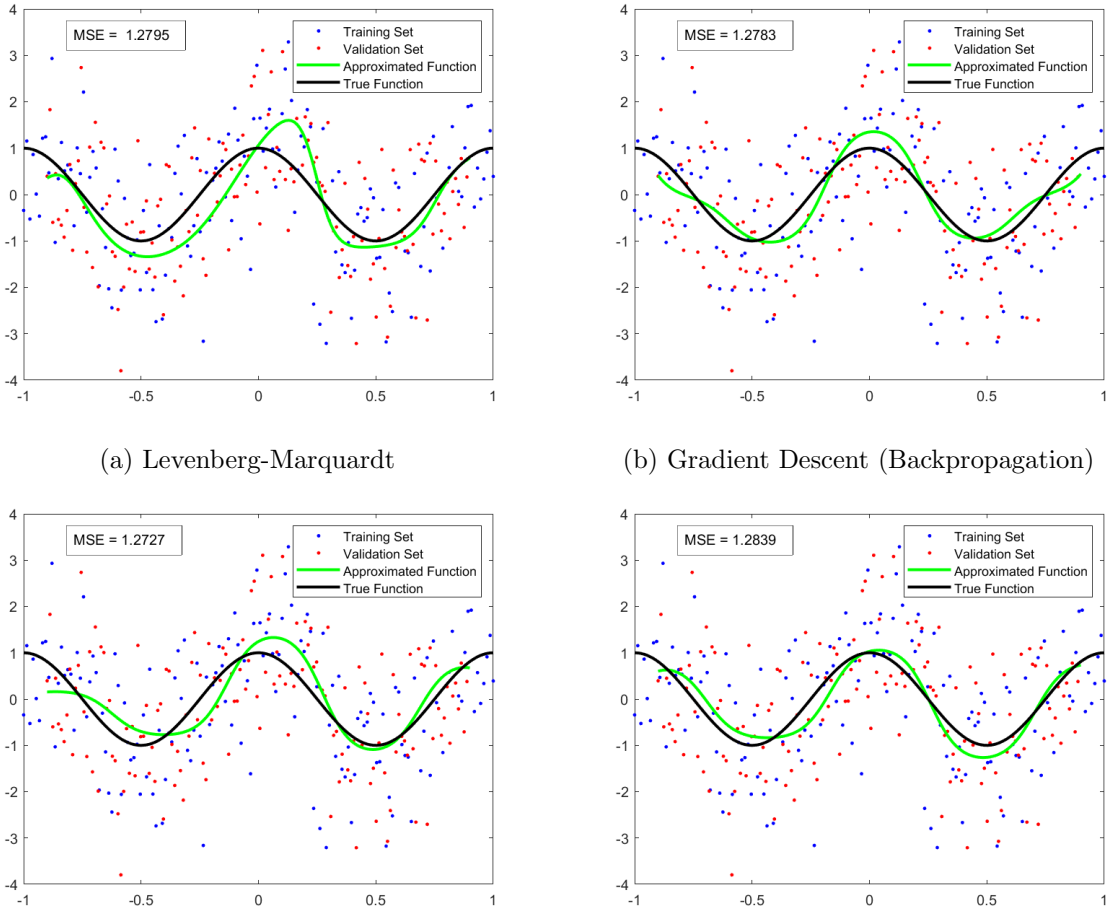
4. *Training algorithms*

In this case four different training algorithms are compared: backpropagation, conjugate gradient, quasi-Newton and Levenberg-Marquardt, which is the default in MATLAB. As before, the remaining characteristics are held constant, the size of the training dataset

is 150, the amount of noise considered is a standard deviation of 1.2 and the number of hidden units is 6.

It is important to mention that backpropagation is a term that sometimes is used to refer to gradient descent algorithm, so in MATLAB the gradient descent algorithm was conducted.

Figure 9 shows the difference in fit between each algorithm. Visually, it can be seen that all of the algorithms produce similar results. In regard to the MSE, the algorithm that presents the lowest value is the conjugate gradient and the highest value is the Quasi-Newton. It is important to mention that even though the difference in MSE is not large between algorithms, some of them were slower in comparison to Levenberg-Marquardt, which is well known to perform well in function fitting problems and it is considerably faster than the other ones in such situations (see Matlab, 2018). The message of this example is that not always the same algorithm performs better than other ones, it always depends on the problem and the data at hand. The best way to keep the best solution is to try different algorithms and use the one that has better performance.



(a) Levenberg-Marquardt

(b) Gradient Descent (Backpropagation)

(c) Scaled Conjugate Gradient

(d) BFGS Quasi-Newton

Figure 9: Comparison of neural nets using different training algorithms

5. *Local minimum vs Early stopping*

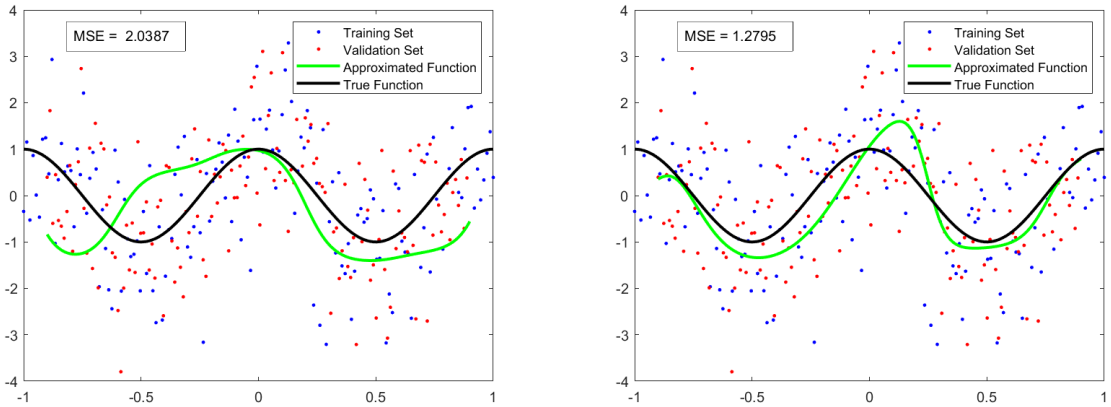
In this case, two different stopping rules are considered: local minimum and Early stopping. Local minimum is performed by checking the gradient in each iteration and stopping

the algorithm when this value is smaller than a threshold, which by default is $1e-5$. On the other hand, in early stopping rule, the error on the validation set is monitored during the training process and the algorithm stops when the error on the validation set start to rise.

To control which stopping rule to use in MATLAB, the number of validation checks is set to 0 in the local minimum case and 6 in early stopping (default value). This number of validations is the number of iterations that the error is increasing after reaching a local minimum. If no iteration is considered, the algorithm stops at the first local minimum and if six iterations are considered, the algorithm stops at a deeper point.

As before, the remaining characteristics are held constant, the size of the training dataset is 150, the amount of noise considered is a standard deviation of 1.2, the number of hidden units is 6 and the training algorithm is Levenberg-Marquardt, which is the faster one.

The difference between both stopping rules is presented in Figure 10. So far this criterion is the one that impacts the most the performance of a neural network. In the left plot, the approximated function is far from the true function and as a consequence the error in the test set increased a lot. This example is important in the sense that shows that the optimization process of a neural net is far from trivial and one should be aware of the default configurations that each software usually has, because it significantly impacts the final result.



(a) Local minimum (b) Early Stopping

Figure 10: Comparison of neural nets using different stopping rules

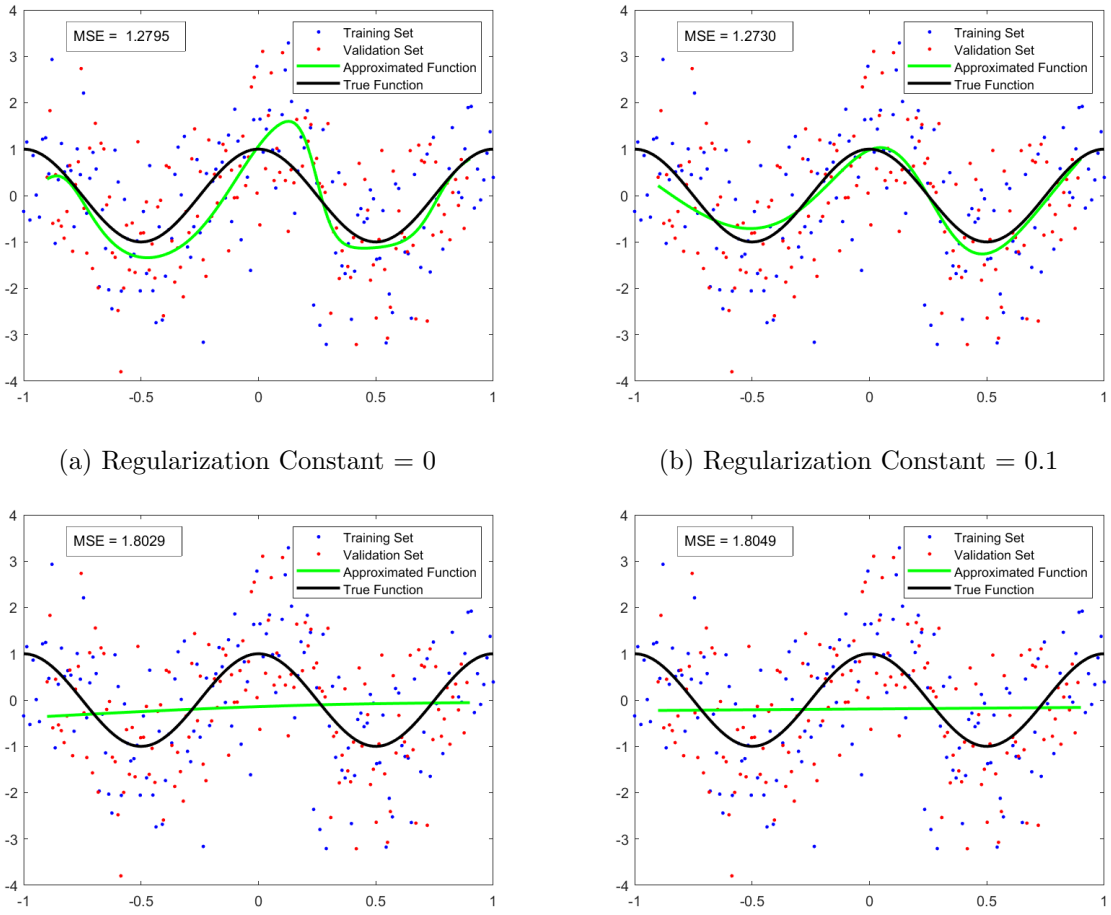
6. *Regularization constant*

In this case different regularization constants are considered: from 0, where no regularization is conducted, to 0.5 where in the optimization process the original cost function and the weight decay term have the same importance. Four different constants are considered, namely 0, 0.1, 0.25 and 0.5.

As before, the remain characteristics are held constant, the size of the training dataset is 150, the amount of noise considered is a standard deviation of 1.2, the number of hidden neurons is 6, the training algorithm is Levenberg-Marquardt.

In Figure 11 the difference in fit between each regularization constant is presented. Graphically, there is a huge difference between no regularization and the constant equals to 0.5, because in the latter one (plot 11d) the approximated function is just a straight line

with a small slope. On the other hand, there is an improvement in the performance when the regularization is equal to 0.1 (plot 11b), the approximated function is almost the same as the true function. In this comparison is clear that the selection of the regularization constant is really important because it impacts the quality of the results. In fact, given that this choice is not straightforward, a Bayesian approach can be taken by considering all weights and biases as random variables.



(a) Regularization Constant = 0

(b) Regularization Constant = 0.1

(c) Regularization Constant = 0.25

(d) Regularization Constant = 0.5

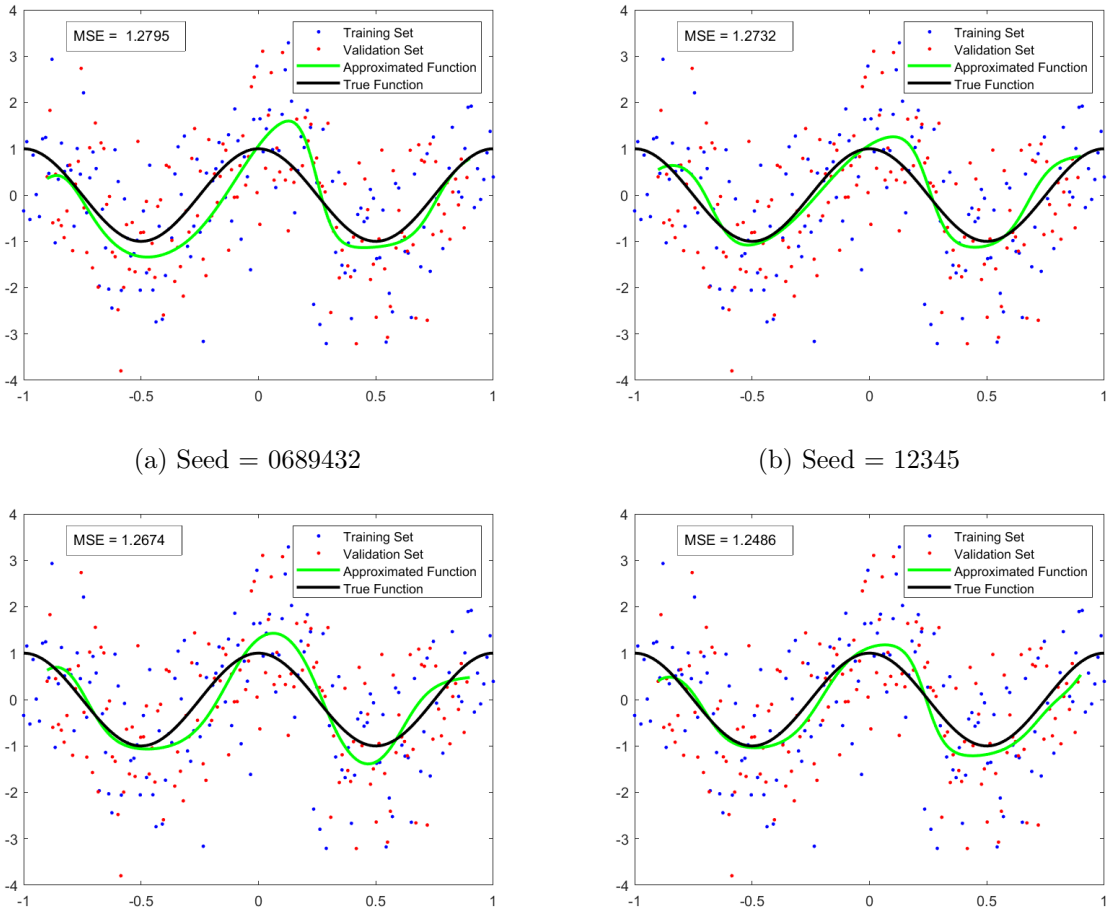
Figure 11: Comparison of neural nets using different regularization constant

7. *Initial weights*

As it was mentioned above, each time a neural net is trained, different initial values are given to weights and biases. Usually, these values are random, but they can be set if prior information is available. In this case different starting values are considered by controlling the seed in the random number generator. Four seeds are compared: 0689432, 12345, 67890 and 111111. Note that this comparison can be done by just training different times the same neural net, but to get reproducible results it is conducted in this way. As before, the remaining characteristics of the neural nets are held constant.

Figure 12 shows the fit of each neural net using different starting weights and biases. Visually, all approximated functions are quite similar, and in some cases they are closer to the true function (see Figures 12b and 12d). In regard to the MSE, the differences are

not large but still they give the message of the importance of considering different starting values, since it increases the likelihood of getting a better solution.



(a) Seed = 0689432

(b) Seed = 12345

(c) Seed = 67890

(d) Seed = 111111

Figure 12: Comparison of neural nets using different initial weights and biases

All in all, different characteristics of a neural network were investigated and compared. Some interesting results were found in regard to the performance of a neural net in a test dataset. First, it does not really matter the number of observations available, but the variance there is them. The more variability there is, the harder it is to get a good solution. Second, the number of hidden units impacts directly the quality and performance of a neural net. Too complex neural nets tend to overfit the data which lead to bad predictions. Third, regularization is a useful tool to avoid overfitting and forget about the number of hidden units. Fourth, the choice of the training algorithm has to be influenced by the problem that is being analyzed, some of them are faster or lead to better results than others in determined situations. Finally, always train a neural network several times to avoid bad local optima solutions.

1.4 Curse of dimensionality

In this section, different neural networks are conducted to get a good approximation of the $sinc(t)$ function, which is defined as follows:

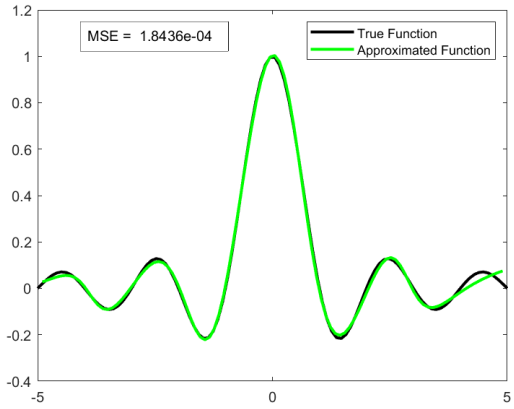
$$f(x) = sinc\left(\sqrt{\sum_{i=1}^{m} x_i^2}\right), \quad sinc(t) = \begin{cases} \frac{sin(\pi t)}{\pi t} & t = 0 \\ 1 & t \neq 0 \end{cases} \tag{7}$$

Three different cases are considered: when m is equal to 1, 2 and 5. In each case, different training algorithms are used, and the Mean Squared Error is calculated to make a comparison between them and decide which leads to better results. This performance measure is computed on a test set, which is completely different and independent to the training set. In this way one can be sure that the test set in one neural net is not going to be used as a training set in other neural net. In addition, to avoid bad local optima solutions, each neural network is trained several times and the net with the lowest MSE is kept for comparison. As before, in order to get reproducible results the seed in the random number generator is set to 0689432.

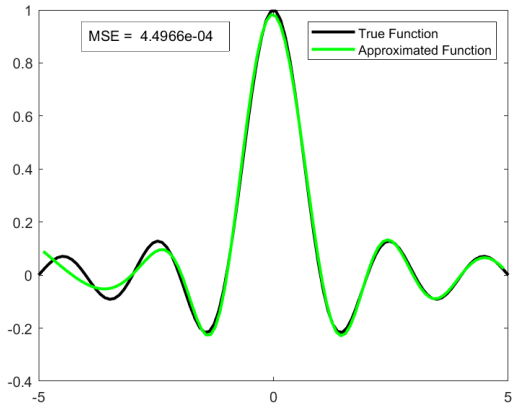
1. *m = 1 (one-dimensional sinc function)*

   In this case, only one input is considered and it is created as linearly spaced vector between -5 and 5 of size 100. The test set is similarly created but the range considered is between -4.9 and 4.9. The network used for this problem is a 1-5-1 network with $tansig$ activation functions in the hidden layer and linear activation function in the output layer. Different number of hidden neurons are tried, from 2 to 10, and 5 is the final choice because it leads to good results in a rather simple network. Different training algorithms are considered, namely Levenberg-Marquardt, Bayesian Regularization, Quasi-Newton and Scaled conjugate gradient. Each neural net is trained 100 times, where different initial random weights and biases are used.

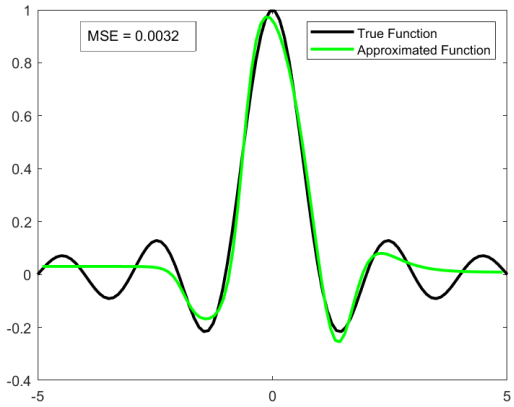
   Figure 13 shows the difference in fit between each algorithm and the corresponding MSE. Visually, it can be seen that there are some differences in the fit, especially in the Quasi-Newton and the scaled conjugate gradient (plots 13c and 13d). In both, the approximated function is close to the true function in the range [-2,2], but outside this range the quality of the prediction is not as good. In regard to the MSE, the algorithm that presents the lowest value is the Levenberg-Marquardt and the highest value is the scaled conjugate gradient. It is important to mention that even though the difference in MSE is not large between the LM algorithm and the Bayesian regularization, the latter took almost 10 times more time to get the solution. Therefore, the LM algorithm is preferred in this case.



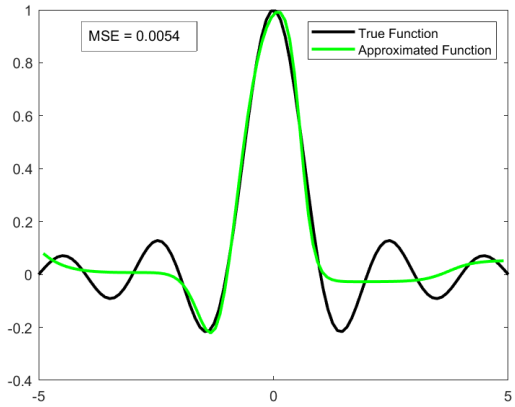
(a) Levenberg-Marquardt



(b) Bayesian regularization



(c) BFGS Quasi-Newton



(d) Scaled Conjugate Gradient

Figure 13: Comparison of neural nets using different training algorithms, when m = 1

2. *m = 2 (mexican hat function)*

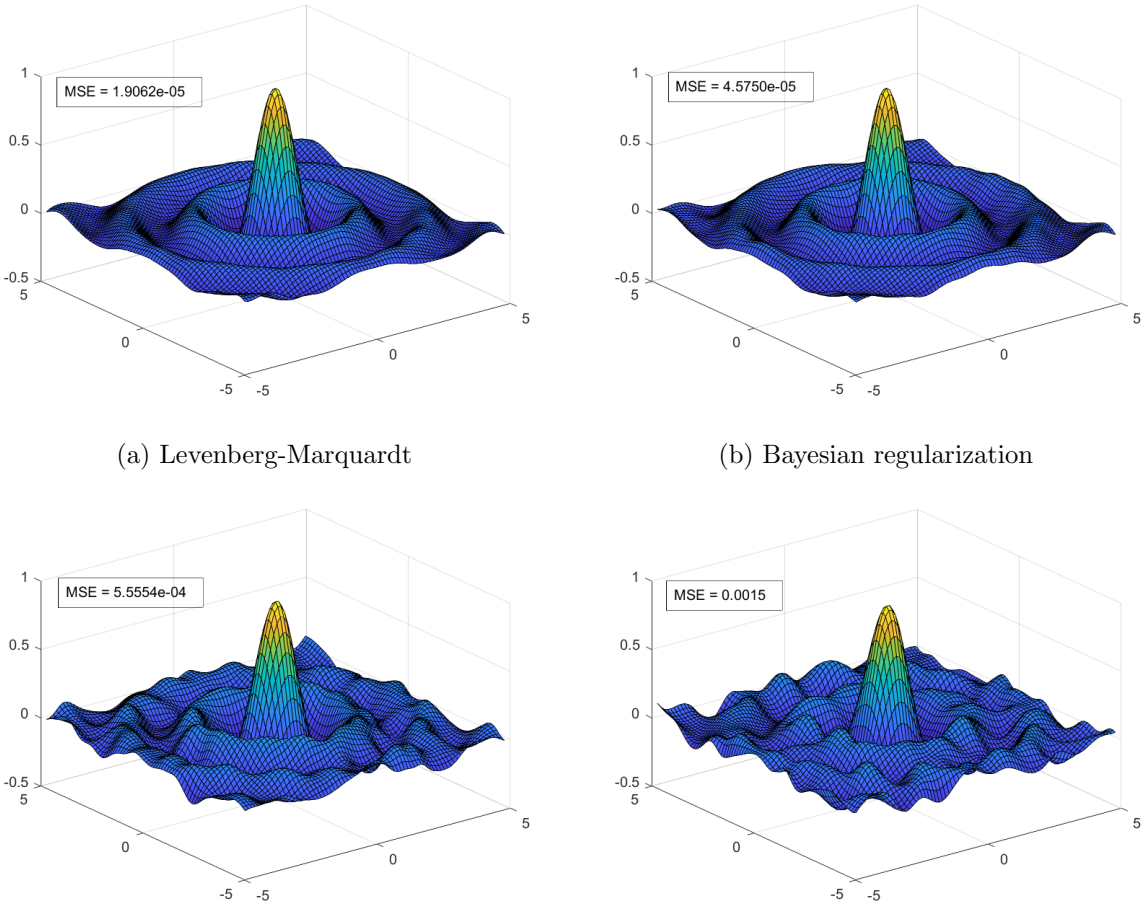
   In this case, the input dataset is harder to get because it is necessary to get all possible pair combinations between the first and the second variable to build a grid. To do so, for $x1$ a sequence of 101 numbers between -5 and 5 is created in a vector and then each element is repeated 101 times, resulting in a vector of 10201 elements. Then for $x2$ another sequence of 101 numbers between -5 and 5 is created and the whole vector is repeated 101 times, resulting also in a vector of 10201 elements. This is a small example of how both input patterns look like:

$$x = (x_1, x_2) = \begin{bmatrix} -5 & -5 & -5 & ... & 5 & 5 & 5 \\ -5 & -4.9 & -4.8 & ... & 4.8 & 4.9 & 5 \end{bmatrix}$$

   In a similar fashion, a grid of 79×79 observations is created for the test set with a sequence of numbers between -4.9 and 4.9, resulting in two vectors with 6241 elements each. The network used for this problem is way more complicated that in the previous case because of the size of each input. A 2-50-1 network with tansig activation functions in the hidden layer and linear activation function in the output layer is trained. Different number of hidden neurons are tried, from 5 to 100, and 50 is the final choice because it leads to good results. As before, different training algorithms are considered and each

neural net is trained only 20 times because the execution time for each training process is larger.

Figure 14 shows the difference in fit between each algorithm and the corresponding MSE. Visually, it can be seen that there are a lot of differences in the fit, especially in the Quasi-Newton and the scaled conjugate gradient (plots 14c and 14d). In both, the approximated function presents a lot of noise around the highest peak. In regard to the MSE, the algorithm that presents the lowest value is the Levenberg-Marquardt and the highest value is the scaled conjugate gradient. As in the case with just 1 input, the LM algorithm is preferred.



(a) Levenberg-Marquardt

(b) Bayesian regularization

(c) BFGS Quasi-Newton

(d) Scaled Conjugate Gradient

Figure 14: Comparison of neural nets using different training algorithms, when m = 2

3. *m = 5*

In this case, the input dataset is more difficult to get because it is necessary to get all possible combinations between the five input variables to build a grid in a sixth dimensional space. This process is conducted in a similar way as in the previous case, for each variable a sequence of just 5 values between -5 and 5 is created and then each vector is repeated the corresponding number of times to get all combinations, resulting in vectors of 15625 elements.

In regard to the test set, the process is repeated by creating a sequence of four numbers between -4.75 and 4.75, resulting in vectors with 4096 elements each. The network used

for this problem is way more complicated that in the previous case because of the size of each input. A 5-100-1 network with tansig activation functions in the hidden layer and linear activation function in the output layer is trained. Different number of hidden neurons are tried, from 50 to 150, and 100 is the final choice because it leads to a MSE smaller than 0.01 without being "too" complicated. As before, different training algorithms are considered and each neural net is trained only 10 times because the execution time for each training process is larger.

Since it is not possible to show graphically the true function nor the approximated function, Table 2 shows the difference in MSE for each neural net. Surprisingly, the training of all networks started with the same initial weights and biases and the optimal solution was the same, or at least they got the same minimum, for the last three networks. However, the Quasi-Newton algorithm was the faster one and therefore it is chosen.

	MSE
Levenberg-Marquardt	0.0027
Bayesian Regularization	0.0018
BFGS Quasi-Newton	0.0018
Scaled Conjugate Gradient	0.0018

Table 2: Comparison of MSE between neural networks trained with different algorithms

To conclude, several neural networks were fitted to obtain a good approximation of the $sinc$ function in 2, 3 and 6 dimensions. In the first case, 5 hidden neurons and the Levenberg-Marquardt algorithm were needed to obtain a good approximation of the true function in the interval $[-4.9, 4.9]$ of the input variable. In the second case, 50 hidden neurons and the LM algorithm were needed to get a decent representation of the true function. And in the third case, 100 hidden neurons and the Quasi-Newton were needed to get a good MSE. It is important to mention that in the last scenario, three different algorithms resulted in the same MSE but the faster one was preferred. In regard to the curse of dimensionality, it is clear that the number of hidden neuron increases as the dimension of the input increases, however the final number of parameters needed to obtain a good approximation is not as large as it would be needed if a polynomial was considered.

2 Exercise Session 2

2.1 Santa Fe laser data - time-series prediction

In time series problems, the use of the neural networks is different. The training process is the same as in every other kind of problem, feedforward mode, but the prediction process requires some changes. Once the training is finished, the neural network is used in an iterative way as a recurrent network, this is also called close loop, and it consists in using the prediction at some point t, to predict the value in $t + 1$.

The goal of this section is to predict the next 100 points from a chaotic laser dataset. To do this, different Non-linear Autoregressive (NAR) neural networks with different specifications are trained and then compared to get the best possible prediction. This comparison is conducted visually and by measuring the performance on a test set via the Mean Square Error (MSE).

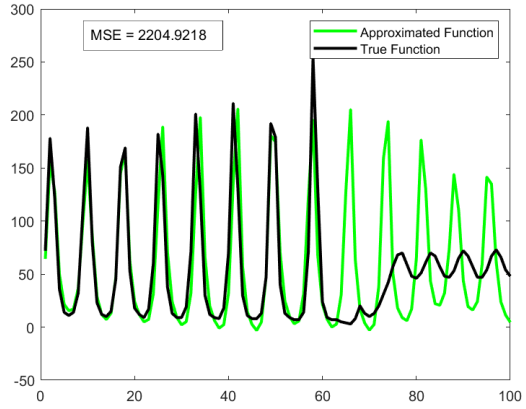
In regard to local optima, it is well known that each time a neural net is trained, different initial weights and biases, and different divisions of the data into training, validation and test

sets are considered. Thus, each neural network is trained 100 times, and the net with the lowest MSE is kept for comparison. As before, in order to get reproducible results the seed in the random number generator is set to 0689432.

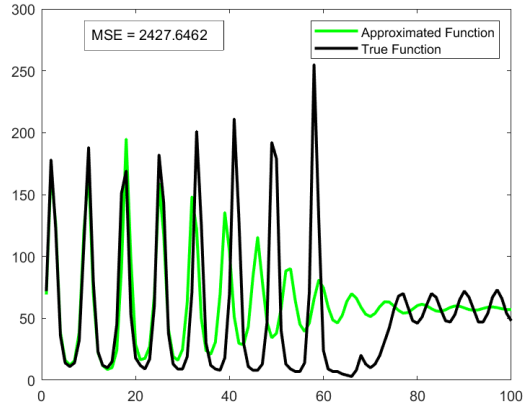
1. *Lag order*

   Four different lags are tried in this comparison: 2, 7, 14 and 21. The lag 2 is chosen because it is usual in time series problems, is like analyzing the series in differences. Lag 7, 14 and 21 are chosen because every 7 time periods there is a peak in the series, this is similar to analyze the data in seasonal differences. To compare the performance of these neural networks, all characteristics are constant: 10 hidden units, the training algorithm is the Levenberg-Marquardt, the number of validation checks is 6 (default), no regularization is considered, and the selection of initial weights, which is controlled by the seed in the random number generator.

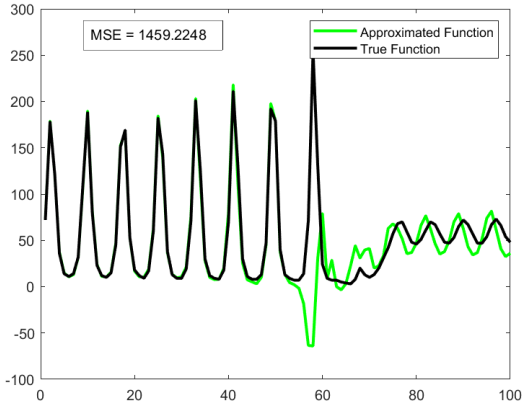
   In Figure 15 the quality of the prediction of each neural net is presented. Visually, the predictions of all nets are close to the true value up to time 60 approximately. The main difference is noticeable after time 60, where some them are far from the true value (Figure 15a) and some are close (Figure 15c). In regard to the MSE, the best prediction is given by the neural network that has a lag of 14, which means that the time series is persistent because the present value depends on past values up to 14 time periods.



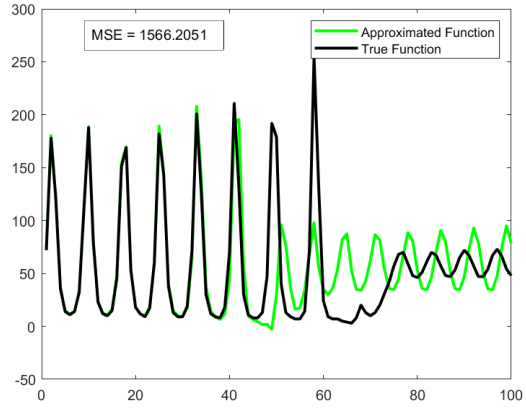
(a) Lag = 2



(b) Lag = 7



(c) Lag = 14



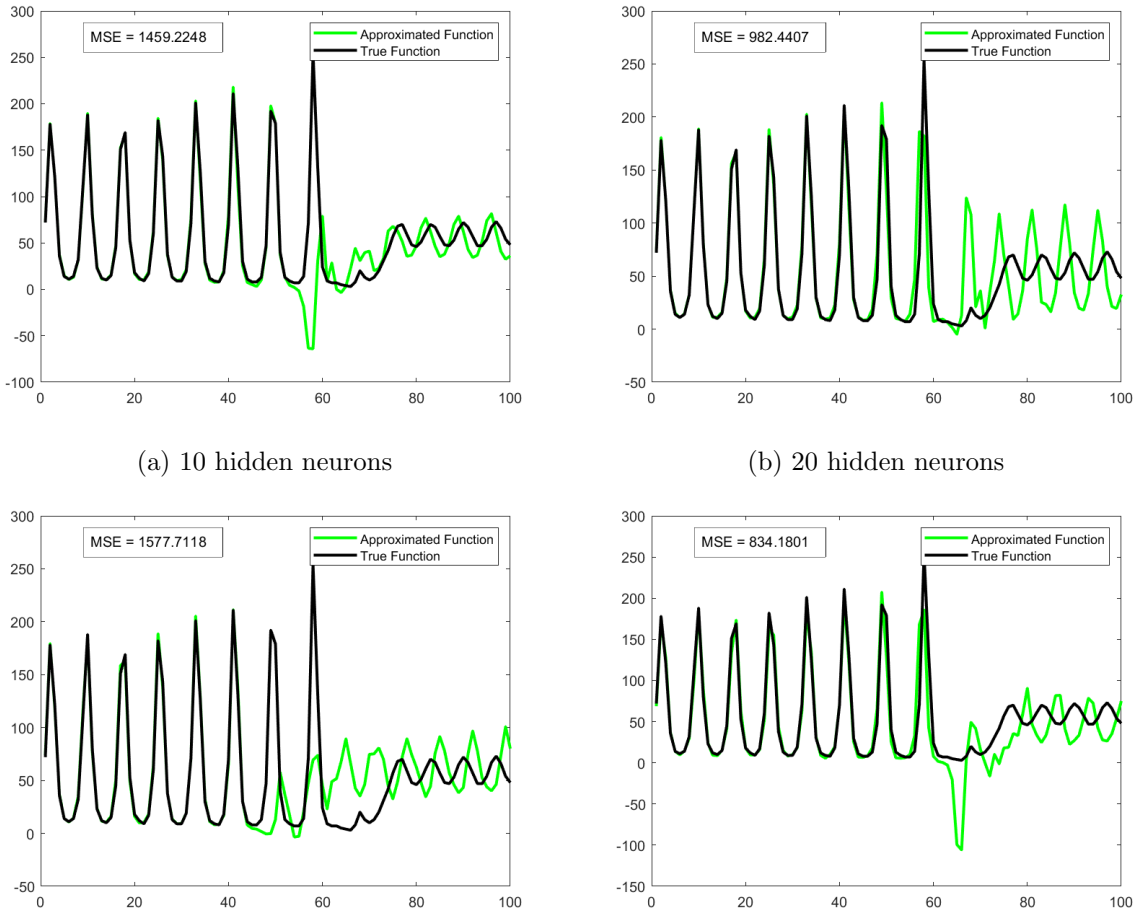
(d) Lag = 21

Figure 15: Comparison of neural nets using different lags

2. *Number of hidden neurons*

   In this case four different scenarios are considered: 10, 20, 30 and 50 neurons. As before, the remaining characteristics are held constant, and the lag considered is 14. It is important to mention that more than 4 scenarios were considered in the training process (8, 9, 11, 12, 15 neurons), however only these 4 cases are shown because they produce an overview of the overall prediction trend.

   Figure 16 shows the prediction of each neural net and its corresponding performance measure. In terms of MSE, the best prediction is given by the net that has 50 hidden units followed by the one that has 20 units. Visually, the net that has only 10 hidden neurons is closer to the true value from the 70th period onward, but its prediction in the last peak, in the interval 50 to 62, is not very accurate. Something similar happens to the net with 30 neurons, the prediction in the last peak is poor. So, even though the prediction of the net with 50 units is not as close as, say, the one that has 10 units in the last part of the time window, it is preferred because it is capable of predicting correctly the last peak.



(a) 10 hidden neurons

(b) 20 hidden neurons

(c) 30 hidden neurons

(d) 50 hidden neurons

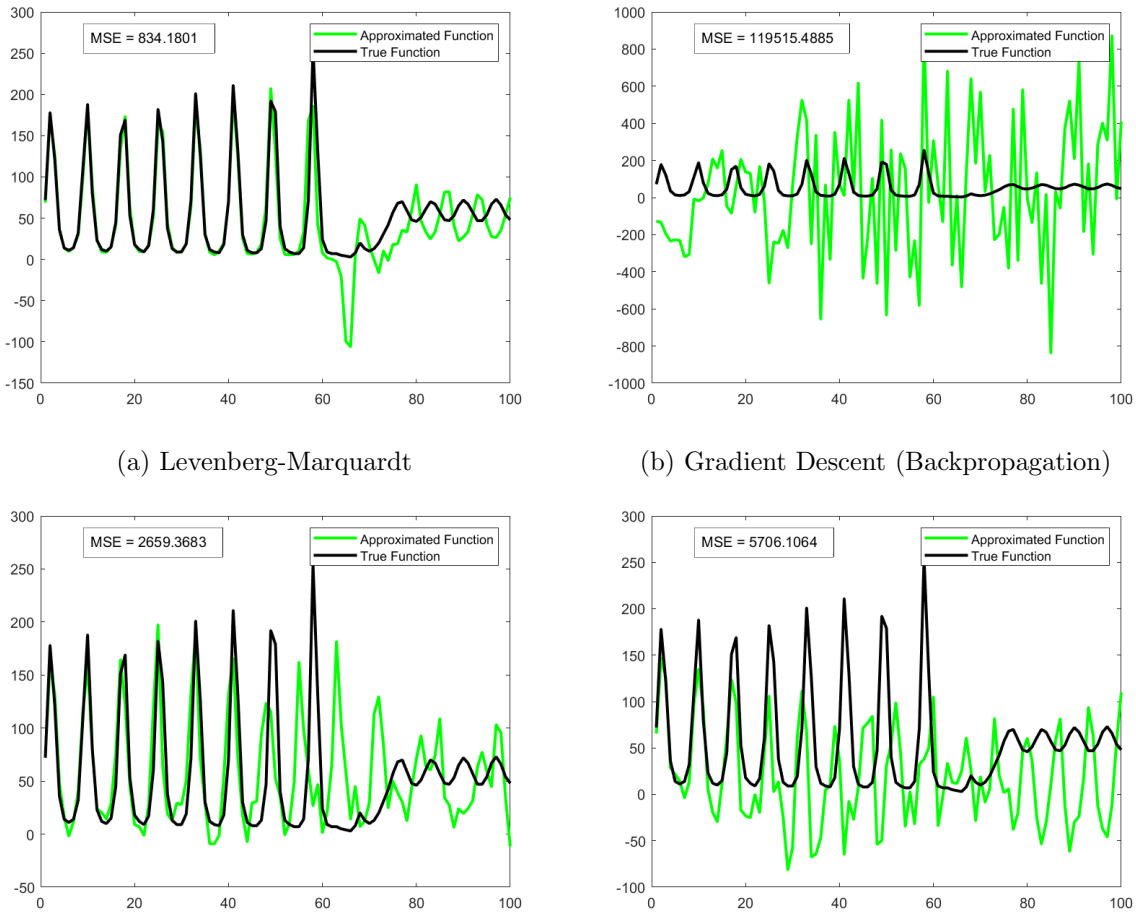
Figure 16: Comparison of neural nets using different number of hidden units

3. *Training Algorithms*

   In this case four different algorithms are compared: Levenberg-Marquardt (default), Backpropagation, Conjugate Gradient and Quasi-Newton. As before, the remaining characteristics are held constant, the lag is 14 and the number of hidden units is 50. It is important

to mention that backpropagation is a term that sometimes is used to refer to gradient descent algorithm, so in MATLAB the gradient descent algorithm was conducted.

Figure 17 shows the difference in prediction between each algorithm. Visually, the results are surprising, at least for the gradient descent algorithm (plot 17b). In this case, the training process was different than in the other algorithms, because the MSE increased at each iteration instead of decrease, as it is supposed to. As a result, the prediction could not be far from the true value. In regard to the scaled conjugated gradient and the Quasi-Newton algorithms, the quality of the prediction is not better than in LM, and both took more time to get the solution.



(a) Levenberg-Marquardt

(b) Gradient Descent (Backpropagation)

(c) Scaled Conjugate Gradient

(d) BFGS Quasi-Newton

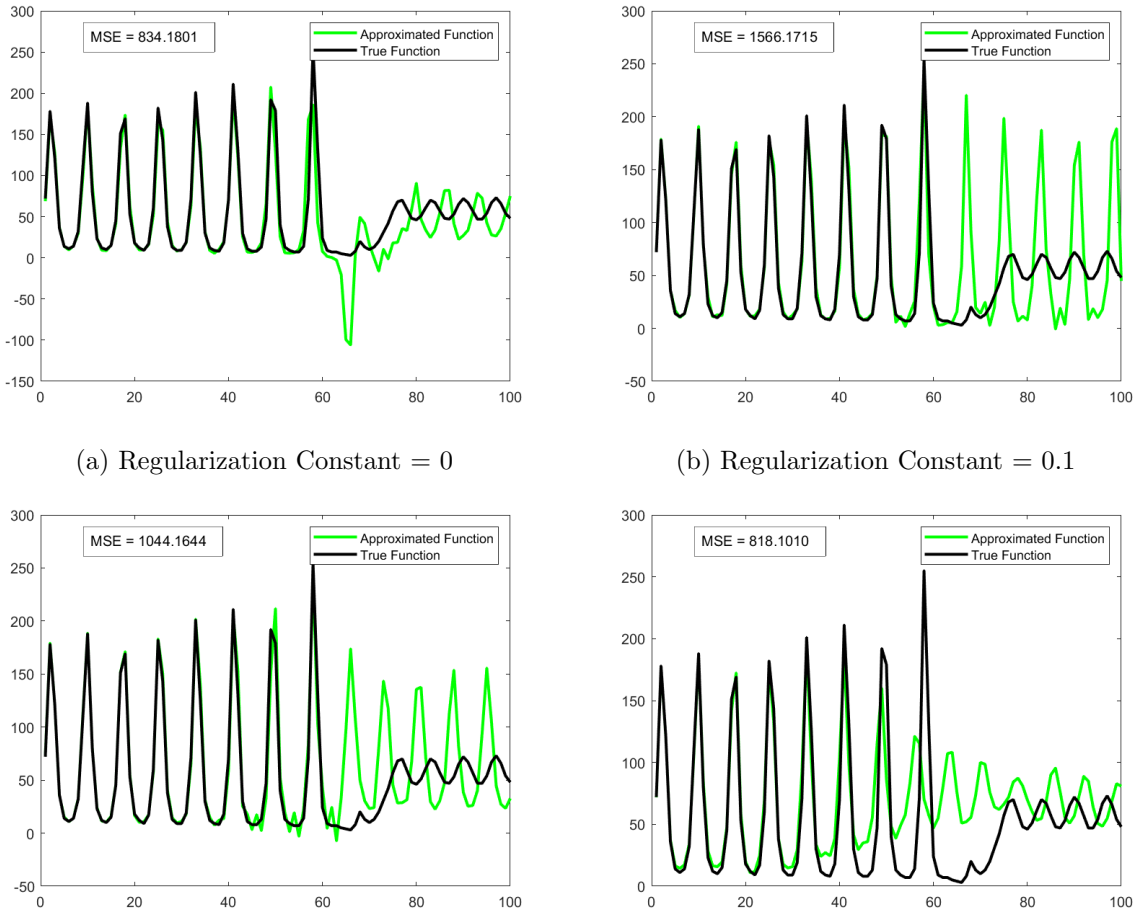
Figure 17: Comparison of neural nets using different training algorithms

4. *Regularization Constant*

In this case different regularization constants are considered: from 0, where no regularization is conducted, to 0.5 where in the optimization process the original cost function and the weight decay term have the same importance. Four different constants are considered, namely 0, 0.1, 0.25 and 0.5. As before, the remaining characteristics are held constant, the lag is 14, the number of hidden units is 50 and the training algorithm is the Levenberg-Marquardt.

In Figure 18 the difference in prediction between each regularization constant is shown. Visually, the prediction is practically the same for all nets up to the 40th time period. After

this point the difference in quality is mostly explained in the last 30 time periods. The network that presents the lowest MSE is the one that has the regularization constant equal to 0.5 (Plot 18d) and the largest is the one that has a regularization constant equal to 0.1 (Plot 18b). Notice, that even though the last net present the lowest MSE, the prediction from time 50 to 70 is not very good. In regard to the first neural net, with no regularization, it is possible to see that the prediction stick to the true value in more time points that in the last network. The reason why its corresponding MSE is larger is because of the prediction in the interval 60 to 70. For this reason, the best neural net to make predictions the first one with no regularization.



(a) Regularization Constant = 0

(b) Regularization Constant = 0.1

(c) Regularization Constant = 0.25

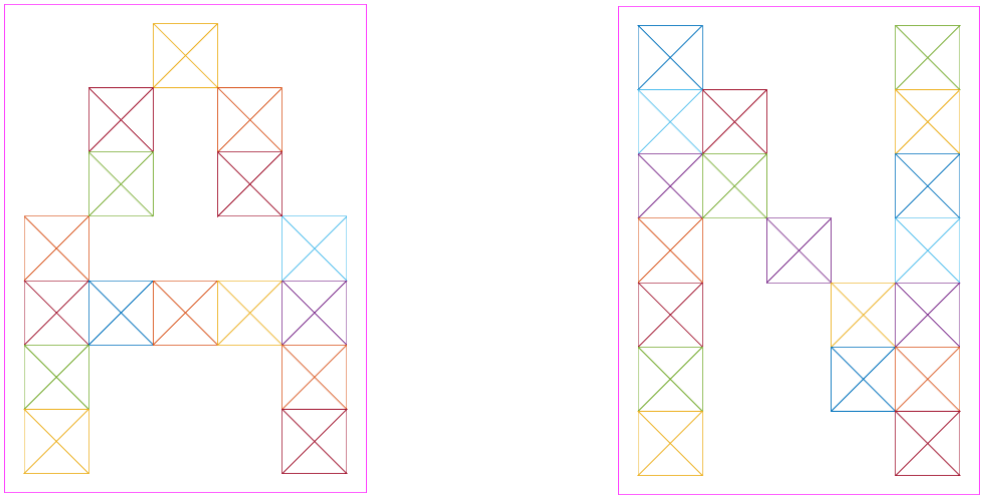
(d) Regularization Constant = 0.5

Figure 18: Comparison of neural nets using different regularization constant

In conclusion, different characteristics were compared and some interesting results were found in regard to the quality of the prediction. First, the lag needs to be chosen carefully because the quality of the prediction depends on this value. Second, it looks like the gradient descent algorithm is not applicable to this kind of problem because in the training process, the MSE increased at each iteration instead of decrease, as it is expected. Third, the best prediction is given by a neural net with 50 hidden neurons, using the LM algorithm and no regularization on a dataset with lag 14. It is important to mention that in all of these cases the correlogram of the residuals were checked but, none of them were white noise, so the results can be improved.

2.2 Alphabet recognition

This an example of how to train a neural network to recognize alphabet characters. The input is a matrix of 26 columns (one for each letter) and 35 rows that correspond to a 5x7 bitmap. The values of this matrix are either 0 or 1, indicating what cell should be colored, so to speak. The output is an identity matrix of 26x26 that indicates which is the final letter. Figure 19 shows how the input data look like.



(a) Letter A (b) Letter N

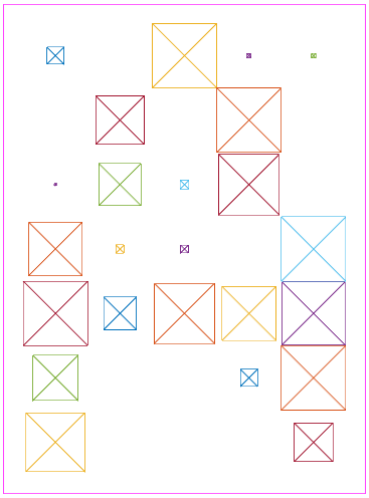
Figure 19: Example of how the input looks like

A neural network with one hidden layer and 25 hidden neurons is used to tackle this problem. The characteristics of the net are the default in the software, meaning that it is trained with the Levenberg-Marquardt algorithm, the dataset is randomly divided in training, validation and test sets, and the algorithm stops when the network is no longer likely to improve on the training or validation sets.

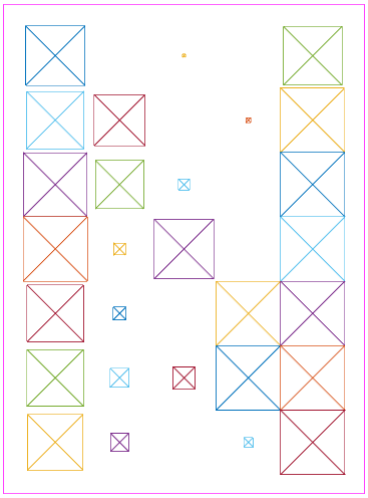
Since the neural net is trained only once in the example and given that a seed is set in the beginning of the code, it is certain to say that the training of the neural net stopped because the gradient of the cost function (MSE in this case) was smaller that $1e-7$, which is the default value.

In addition, another neural net with the same characteristics is trained with noisy data. To do so, 30 copies of each letter are created and random values are added to each of them. Figure 20 shows similar examples with random noise.

The training of this second neural network stopped because the validation error started to increase. By default, when the validation error increases for 6 continuous iterations the training stops to prevent overfitting.



(a) Letter A with noise



(b) Letter N with noise

Figure 20: Example of how the input with noise looks like

Once both neural nets are trained, their performance is tested by predicting different test sets. To do so, 21 different noisy test datasets are used and in each of them the amount of noise varies by changing the standard deviation of the noise (from 0 to 1). This performance is measured as a percentage of recognition errors, and it is calculated by first transforming the maximum predicted value in each column of the output pattern to 1 and the remaining values to zero, so each column has only a single 1 which corresponds to the classified letter[2]. Then the absolute difference between the predicted and the true values is computed, resulting in a matrix of zeros and ones. In this matrix when the classification is correct, the whole column is filled with zeros, whereas each time there is a misclassification, there are two 1s in the respective column. Thus, to compute the final percentage of misclassifications, all of the values in this matrix are summed and then divided by two times the number of columns to classify (2*780).

As a result, the second neural net presents the lowest percentage of recognition errors, meaning that it is the best to predict new data (see Figure 21). This is an expected result because the second neural net was trained with noisy data whereas the first one was not.

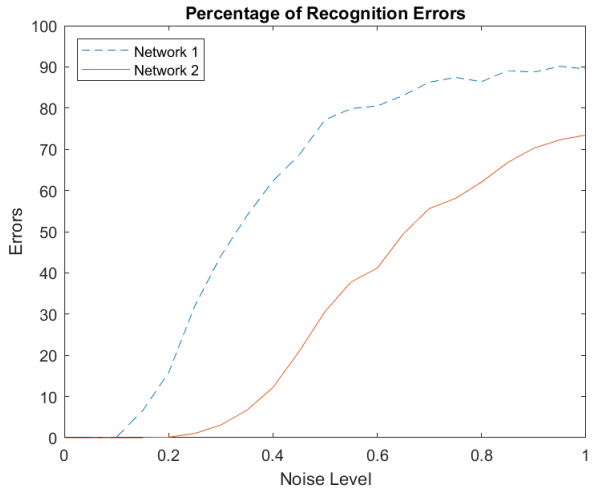


Figure 21: Comparison of both neural nets is terms of percentage of recognition errors.

[2]In this example, there are 780 (30*26) columns in the output pattern, that corresponds to 30 noisy copies of each letter and 26 possible letters

2.3 Breast Cancer Wisconsin - classification problem

The main difference between a function approximation and a classification problem is the architecture. Usually, the activation function used in the output layer is the softmax transfer function, which is a function that returns values in the range [0,1] inclusive. This value can be interpreted as the probability of an observation to belong to a specific class.

The purpose of this section is to create a nonlinear classification rule to decide whether a patient has breast cancer or not. To do so, several neural networks with different specifications are trained and then compared to get the best solution possible. This comparison is conducted measuring the performance on a test set via cross-entropy (CE), the percentage of misclassifications and the ROC curve.

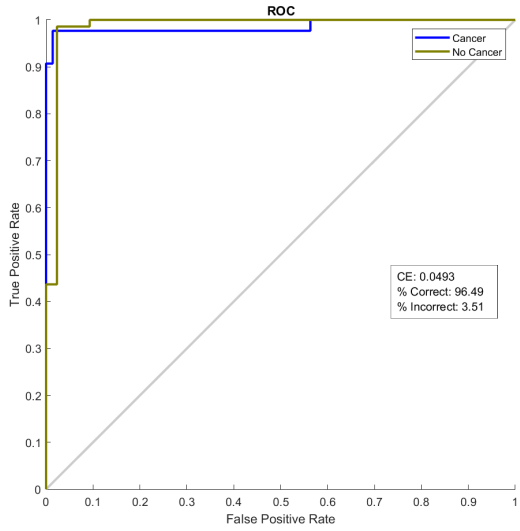
The original dataset, which has information of 569 patients, is divided in two subsets: a training set with 80% of the patients (455 observations) and a test set with the remaining 20%, which corresponds to 114 patients. The purpose of this division is to be sure that a test set in one neural network is not going to be used as a training set in other neural net.

In regard to local optima, it is well known that each time a neural net is trained, different initial weights and biases, and different divisions of the data into training, validation and test sets are considered. Thus, each neural network is trained 100 times and the net with the lowest cross-entropy is kept for comparison. As before, in order to get reproducible results the seed in the random number generator is set to 0689432.

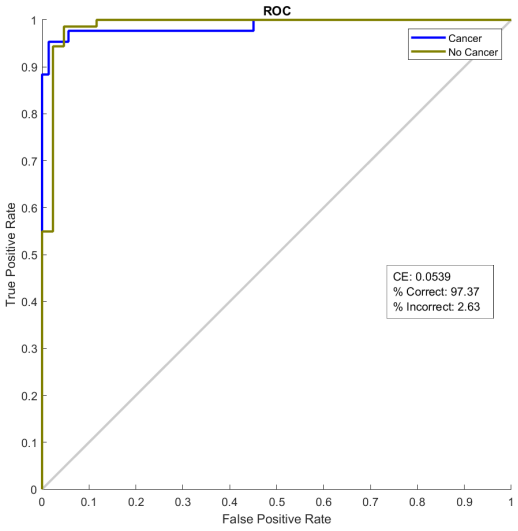
1. *Number of hidden neurons*

   Four different architectures are considered: 2, 5, 10 and 15 hidden neurons. To compare the performance of these nets, the remaining characteristics are held constant, such as the training algorithm, which is the conjugate gradient, the number of validation checks, the regularization constant, which is equal to zero and the selection of initial weights (controlled by the seed in the random number generator).

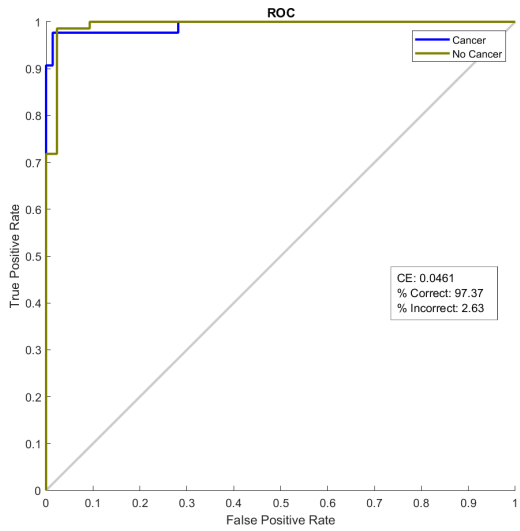
   Figure 22 shows the ROC curve with the percentage of correct classifications and misclassification and the respective cross-entropy, for all neural networks on the test set. The first conclusion that can be drawn is that all nets give good results, because the area under the ROC curve is almost 1, which is the perfect scenario. The percentage of correct and incorrect misclassifications is the same for the nets with 5, 10 and 15 hidden neurons. The net with the smallest cross-entropy is the one with 10 units. These results lead to conclude that the neural network with 5 hidden neurons performs best that the others because it produces solid results with less number of parameters (is the most parsimonious).



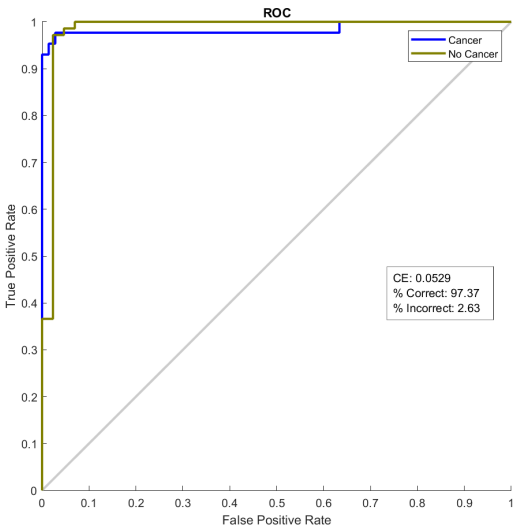
(a) 2 hidden neurons



(b) 5 hidden neurons



(c) 10 hidden neurons



(d) 15 hidden neurons

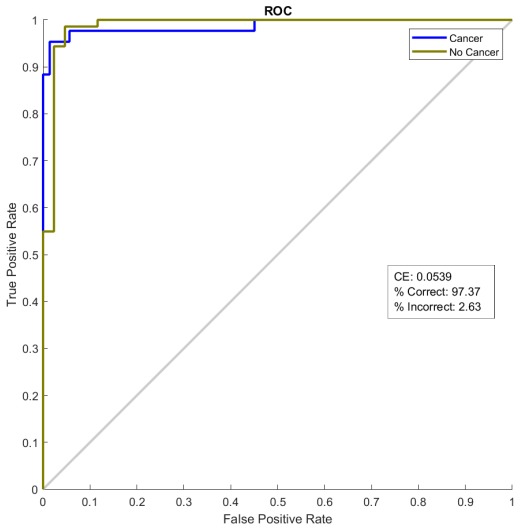
Figure 22: Comparison of neural nets using different number of hidden neurons

2. *Training algorithms*

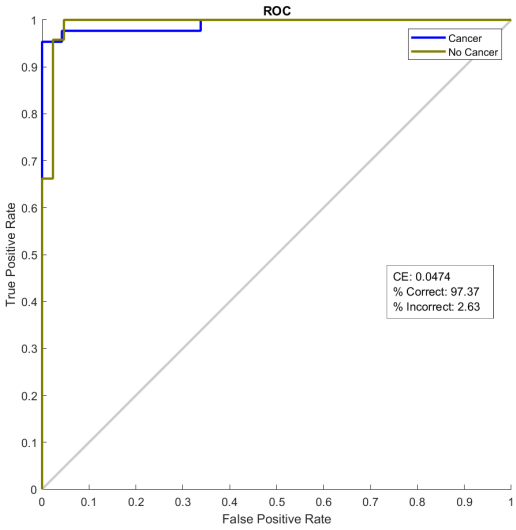
In this case four different training algorithms are compared: backpropagation, Levenberg-Marquardt, quasi-Newton and conjugate gradient, which is the default in MATLAB for pattern recognition. As before, the remaining characteristics are held constant and the number of hidden units considered is 5. It is important to mention that backpropagation is a term that sometimes is used to refer to gradient descent algorithm, so in MATLAB the gradient descent algorithm was conducted.

Figure 23 shows the difference in classification between each algorithm. Visually, all nets seem to provide similar results because the area under the ROC curve is almost 1, which is the desirable output. In regard to the percentage of classifications, the conjugate gradient and the Levenberg-Marquardt present the highest value (97.37%). On the other

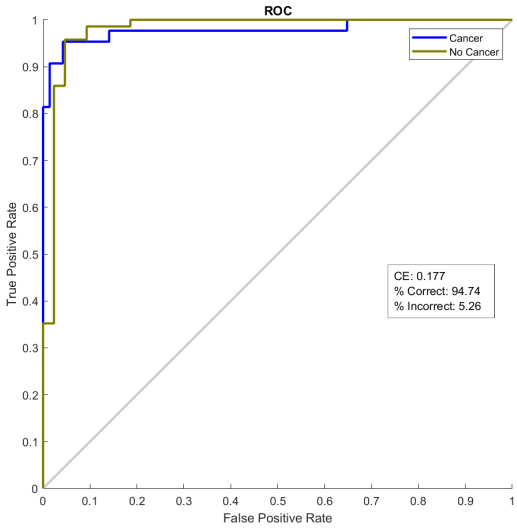
hand, the gradient descent algorithm has the highest percentage of misclassifications (5.26%), although this value is acceptable in a problem like this one. These results lead to conclude that the algorithm with the best performance and speed is the conjugate gradient, an expected result given the simulations presented in Matlab (2018).



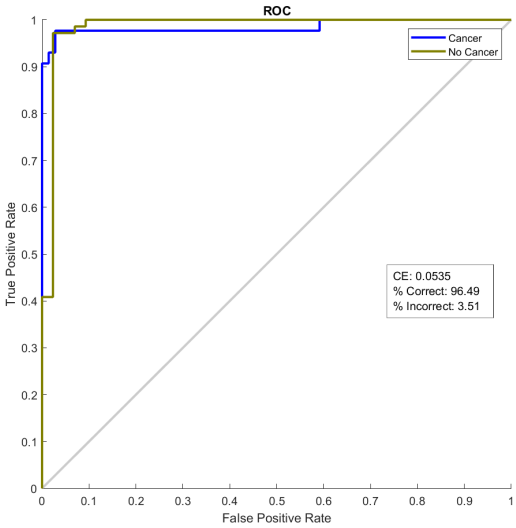
(a) Scaled Conjugate Gradient



(b) Levenberg-Marquardt



(c) Gradient Descent (Backpropagation)



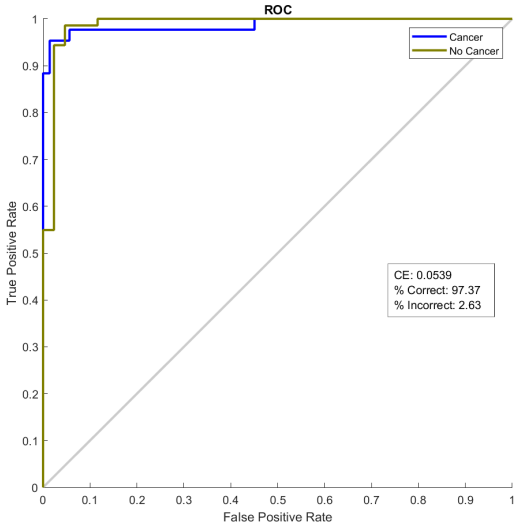
(d) BFGS Quasi-Newton

Figure 23: Comparison of neural nets using different training algorithms

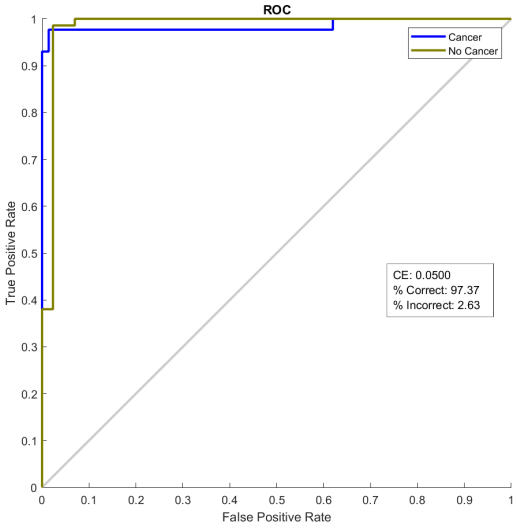
3. *Regularization constant*

The percentage of good classifications in the previous sections are really good for a classification problem like this one. This may suggest that those neural networks can be overfitted. To avoid this problem, different regularization constants are considered: 0, where no regularization is conducted, 0.01, 0.1 and 0.2. As before, the remain characteristics are held constant, the number of hidden neurons is 5 and the training algorithm is the conjugated gradient.

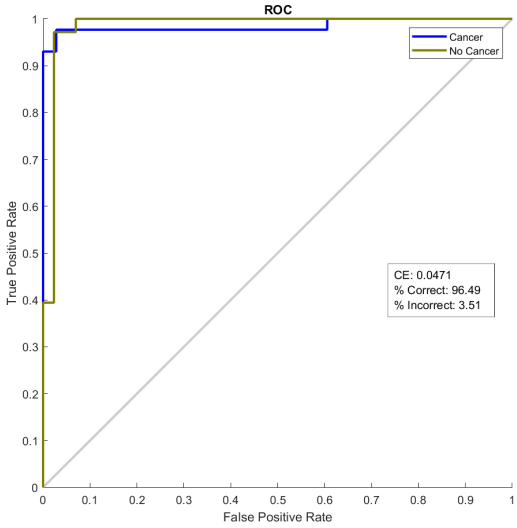
In Figure 24 the difference in performance for each regularization constant is presented. The first conclusion that can be drawn is that all nets give good results, because the area under the ROC curve is almost 1, which is the perfect scenario. The percentage of correct and incorrect misclassifications is the same for the nets with regularization constant equal to 0, 0.01 and 0.2, suggesting that the neural nets in the previous sections were not overfitted. These results lead to conclude that any regularization constant will provide solid results in terms of classification of new patients.



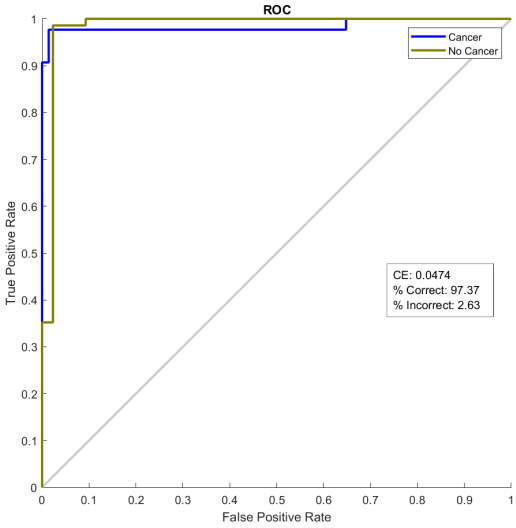
(a) Regularization constant = 0



(b) Regularization constant = 0.01



(c) Regularization constant = 0.1



(d) Regularization constant = 0.2

Figure 24: Comparison of neural nets using different regularization constants

To conclude, different characteristics of a neural network were investigated and compared. In this case of pattern recognition, in contrast to the previous case of function approximation, different parametrizations of the neural net give the same number of misclassification, which by the way is really small (just 3 cases over 114). The number of hidden neurons considered

was the smallest possible because a model with fewer parameters is easier to estimate and maintain in the long run. With respect to the training algorithm, the performance was equally good but the conjugate gradient was chosen because is faster. No regularization constant should be considered in this case because the performance of the neural net was not affected by it. Therefore, the best model to classify new patients is a net with 5 hidden neurons, using the conjugate gradient with no regularization.

3 Exercise Session 3

3.1 Dimensionality reduction by PCA analysis

In this section a bio-medical dataset is used to compare results in performance between a neural net trained with original observations and a neural net trained with the solution of a prior Principal Component Analysis (PCA). The input dataset consists in 21 spectral measurements of 264 blood samples and the idea is to predict three kinds of cholesterol for each blood sample: LDL, VLDL and HDL.

The first step is to conduct a PCA with the goal of having less (uncorrelated) input variables that explain a reasonable amount of the variance of the original dataset. To do so, each spectral measurement is standardized by subtracting its mean and dividing it by its standard deviation. In this way, all measurements can be compared between each other, and the solution of the PCA is easier to get.

Then the PCA analysis is performed on the standardized variables. The criteria to select the number of principal components is to retain all of those whose contribution to the total variation is more than 0.001. As a result, 4 principal components are retained, implying that this solution at least explains 98.30% of the total variability obtained in the original input dataset[3].

Having done this, the next step is to train different neural networks. The first neural network considered is trained with the original dataset, which has 264 blood samples with 21 spectral measurements. This dataset is then divided between training, validation and test set, with 132, 66 and 66 blood samples respectively. The architecture of this net is 21-5-3, which means that there are 21 input measurements, just only hidden layer with 5 hidden neurons, and the output layer with 3 measurements (LDL, VLDL and HDL). The training algorithm is Levenberg-Marquardt and no regularization is conducted. To avoid bad local optima solutions, the network is trained 100 times to ensure that different initial weights and biases are considered. However, in order to get reproducible results the seed in the random number generator is set to 0689432.

In a similar fashion, a second neural network is also trained with the original dataset. All of the considerations mentioned above remain the same in this case, but the training algorithm. In this neural network a Bayesian regularization algorithm is used with the purpose of getting a solution that generalizes well. This algorithm updates the weights and biases according to the Levenberg-Marquardt algorithm, but the cost function (also called error or energy function) is modified by having now a linear combination of squared errors and weights. This algorithm modifies this linear combination so that at the end a network with good generalization results.

Additionally, a third neural network is trained with the solution of the PCA analysis. In this case, the input dataset does not have 21 spectral measurements, but 4 principal components that explain approximately 98% of the total variation. This dataset is also divided in training, validation and test sets as before and the remain characteristics and the architecture is the same as in the second neural network, namely it has 5 hidden neurons and it uses a Bayesian regularization algorithm.

[3] Assuming that all of the remain principal components contribute 0.001 to the total variation, the solution with just 4 components explains at least $1 - (17 * 0.001)$, with 17 principal components left out of the solution.

Table 3 shows the performance of each neural network in terms of Mean Squared Error (MSE) calculated on the training and the test set. Surprisingly, the first neural network (LM algorithm) is the one that has the best performance in both training and test datasets. It seems that the Bayesian regularization algorithm does not improve the quality of prediction in comparison with the LM algorithm, so no regularization is needed in this case. It is important to mention though, the difference in test MSE between the first and the second neural networks is not large, and therefore it could happen that if other dataset were used to train both networks, the second net could perform better than the first one. As it was mentioned before, bayesian regularization uses the LM algorithm to update weights and biases, so the performance of the second net can be influenced by the data at hand.

Network	Training	Test
LM algorithm	0.2684	0.2446
Bayesian Regularization	0.3055	0.2487
PCA	0.3442	0.2659

Table 3: Comparison of MSE between neural networks on training and test datasets

In regard to the third neural network, trained with 4 principal components instead of 21 original measurements, the performance in both training and test MSE is worse than Bayesian regularization on the original data. In this sense and given that there are not many inputs, the **best solution is the second neural network**. Although there are other motivations to choose the neural net with original data: first, it is clear that PCA is trained way more fast that the original dataset (because of the number of inputs), however in this case is not enough to be faster because the dataset is not large and the computing time difference is not huge. Second, when a PCA solution is considered to train a network, it is important to have in mind that each time a new blood sample comes into the dataset, the solution may change in the number of components retained or in different projections resulting of an update of the analysis. So, if the neural network is trained in online mode (weights and biases are updated each time new samples appear), the PCA solution will require an extra step.

3.2 Input selection by Automatic Relevance Determination (ARD)

In this section, the UCI ionosphere dataset is considered to conduct input selection by Automatic Relevance Determination. The dataset consists of 33 input variables and 351 observations in each of them. The output is a 2-class variable with values -1 or 1.

In this methodology, the weights and biases are considered as random variables and therefore they have a corresponding distribution. A different Gaussian prior distribution with a separate hyperparamater is considered, corresponding to each group of weights for each input variable. These hyperparameters are all set to 0.01, but will be re-estimated after the net is trained. The architecture of the neural network is 33-2-1 with tansig activation function in the hidden layer and logistic activation function in the output layer. The net is trained 5 times using the scaled conjugate gradient algorithm and at the end of each cycle the hyperparameters are re-estimated. As before, in order to get reproducible results the seed in the random number generator is set to 0689432.

The first column of Table 4 shows the hyperparameters for the corresponding input to hidden units weights. Each of these values represents an inverse variance, so the larger the value, the smaller the posterior variance will be. Variables 8, 15, 17, 19 and 28, present the largest values and therefore they are considered as less relevant. This is confirmed by looking how

the corresponding weights are close to zero (columns 2 and 3). On the other hand, variables 1, 9, 22 and 32 are considered as the most relevant because they present the largest inverse variance values. In Table 4, the less relevant are in red and the most relevant are in blue.

Inputs	First Training			Second training		
	Hyperpar.	Neuron 1	Neuron 2	Hyperpar.	Neuron 1	Neuron 2
1	0.008	1.721	−13.17	0.008	2.298	15.892
2	0.047	1.345	−6.379	0.038	−1.489	7.056
3	0.122	−3.005	−2.693	7.505	−0.213	−0.267
4	0.139	3.713	0.714	0.075	4.268	2.873
5	0.261	2.578	−0.933	3.241	0.551	0.506
6	0.09	4.332	−1.753	0.262	2.169	1.664
7	0.081	4.565	1.941	0.102	4.353	−0.726
8	167.914	0.005	0.018			
9	0.027	−1.937	−8.442	0.079	1.494	4.731
10	0.066	−5.3	1.383	0.02	0.062	−9.864
11	7.491	0.37	0.104	365.834	−0.005	−0.001
12	1.916	0.125	−0.925	0.34	−1.634	1.755
13	0.081	−3.154	−3.794	467.502	0.008	−0.005
14	0.537	0.807	−1.641	0.042	0.618	6.835
15	1769.108	0.000	0.003			
16	0.262	−0.068	−2.715	552.955	−0.003	0.009
17	811.9	0.005	−0.001			
18	8.795	−0.328	0.019	0.102	−1.202	4.207
19	292.539	0.014	−0.009			
20	8.552	−0.341	0.052	0.015	5.453	−10.176
21	0.301	−0.723	2.427	0.349	−0.835	−2.17
22	0.033	4.62	6.224	0.019	3.14	−9.606
23	3.559	−0.242	−0.677	0.171	−2.189	2.536
24	0.247	1.437	−2.427	0.053	−0.722	6.065
25	3.016	−0.208	−0.734	0.355	−0.047	2.317
26	0.203	−2.974	−0.101	0.01	−8.555	11.59
27	8.866	0.144	−0.224	0.028	1.07	−8.21
28	624.25	−0.006	0.005			
29	0.066	3.116	−4.527	0.017	−0.273	10.73
30	0.102	−2.381	−3.708	0.162	−1.132	3.252
31	4.301	0.464	0.375	17.741	0.035	−0.103
32	0.036	3.748	6.474	0.032	2.071	−7.549
33	0.061	−5.616	0.958	0.268	−0.989	−2.475

Table 4: Hyperparamaters and corresponding weights to neuron 1 and 2 in each neural network

The neural net is trained a second time under the same considerations mentioned before with the only difference that variables 8, 15, 17, 19 and 28 are removed from the analysis. Columns 4, 5, 6 in Table 4 shows the resulting hyperparameters and their corresponding weights. In this case, variables 11, 13, 16 and 31 are now considered as less relevant and variables 1, 20, 26 and 29 are the most relevant. The fact that there are other variables that are considered as less (more) relevant, can be explained by a possible correlation that exists between these variables and the ones removed. Having removed them, reveal new correlations between these variables and the ones that are now present in the dataset.

References

Matlab (2018). Choose a multilayer neural network training function. https://nl.mathworks.com/help/deeplearning/ug/choose-a-multilayer-neural-network-training-function.html. Last accessed on 4-1-2019.

4 Appendix

- **Script Exercise Session 1**

```
% Function Approximation (noiseless case)
nnd11gn

%% The role of the hidden layer and output layer
clear
% 2
x = linspace(0, 1, 21);
% 3
y = -cos(0.8 * pi * x);
figure; plot(x, y, '-'); xlabel('x'); ylabel('y = -cos(0.8 \pi x)');

% 4
% To avoid local optima initialize the NN several times
% Set seed value to obtain reproducible results
setdemorandstream(0689432)  % This number is my student number
net = fitnet(2);
net = configure(net, x, y);
net.inputs{1}.processFcns = {};
net.outputs{2}.processFcns = {};
n_start = 100;
NN = cell(1, n_start);
y_est = zeros(n_start, length(x));
perfs = zeros(1, n_start);
for i = 1:n_start
   fprintf('Training %d/%d\n', i, n_start);
   NN{i} = train(net, x, y);
   y_est(i, :) = NN{i}(x);
   perfs(i) = mse(net, y_est(i,:), y);
end
[m,i] = min(perfs)
% 5.
[biases, weights] = hidden_layer_weights(NN{i});
net_act_fun = hidden_layer_transfer_function(NN{i});
input_act = net_act_fun(weights * x + biases);
% 6.
figure; plot(x, y, '-'); xlabel('x'); ylabel('y = -cos(0.8 \pi x)');
hold on;
plot(x, input_act(1,:), 'r-'); plot(x, input_act(2,:), 'g-');
hold off;
legend({'y','x^1','x^2'},'Location','north')
% 7.
[biases_out, weights_out] = output_layer_weights(NN{i});
act_fun_out = output_layer_transfer_function(NN{i});

% Calculation of output
output= act_fun_out( weights_out * net_act_fun( weights * x + biases)) + biases_out;
figure; plot(x, y, '-'); xlabel('x'); ylabel('y = -cos(0.8 \pi x)');
hold on;
plot(x, output, 'p-');
hold off;
legend({'y','y-est'},'Location','north')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Function Approximation (noisy case)
clear
%%% Comparison of dataset size
% Size of the training data set (e.g. 30, 150, 1200 data points).
clear
setdemorandstream(0689432);

tr_x1 = linspace(-1, 1, 30);
tr_x2 = linspace(-1, 1, 150);
tr_x3 = linspace(-1, 1, 1200);
val_x1 = linspace(-0.9, 0.9, 30);
val_x2 = linspace(-0.9, 0.9, 150);
val_x3 = linspace(-0.9, 0.9, 1200);

% The remain characteristics are held constant.
% Amount of noise = standard deviation of 0.6
tr_y1 = cos(2 * pi * tr_x1) + 0.6 * randn(size(tr_x1));
tr_y2 = cos(2 * pi * tr_x2) + 0.6 * randn(size(tr_x2));
tr_y3 = cos(2 * pi * tr_x3) + 0.6 * randn(size(tr_x3));
val_y1 = cos(2 * pi * val_x1) + 0.6 * randn(size(val_x1));
val_y2 = cos(2 * pi * val_x2) + 0.6 * randn(size(val_x2));
val_y3 = cos(2 * pi * val_x3) + 0.6 * randn(size(val_x3));
[sum(tr_y1);sum(tr_y2);sum(tr_y3);sum(val_y1);sum(val_y2);sum(val_y3)]

% Training algorithm= Levenberg-Marquardt
% Hidden neurons = 5
% Early stopping, no regularization
net1 = fitnet(5, 'trainlm');
net2 = fitnet(5, 'trainlm');
net3 = fitnet(5, 'trainlm');

% Training net
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x1));
for i = 1:n_start
```

```
    fprintf('Training %d/%d\n', i, n_start);
    NN1{i} = train(net1, tr_x1, tr_y1);
    val_t1(i, :) = NN1{i}(val_x1);
    perfs1(i) = mse(net1, val_t1(i, :), val_y1);
end
[m1,i1] = min(perfs1)

% plot results (1_3_1)
figure;
plot(tr_x1, tr_y1, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x1, val_y1, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x1, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x1, cos(2 * pi * tr_x1),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE = 0.4275','FitBoxToText','on');

% Training net2
setdemorandstream(0689432);
n_start = 100;
NN2 = cell(1, n_start);
perfs2 = zeros(1, n_start);
val_t2 = zeros(n_start, length(val_x2));
for i = 1:n_start
    fprintf('Training %d/%d\n', i, n_start);
    NN2{i} = train(net2, tr_x2, tr_y2);
    val_t2(i, :) = NN2{i}(val_x2);
    perfs2(i) = mse(net2, val_t2(i, :), val_y2);
end
[m2,i2] = min(perfs2)

% plot results (1_3_2)
figure;
plot(tr_x2, tr_y2, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x2, val_y2, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x2, val_t2(i2,:),'-','LineWidth',2,'Color','green');
plot(tr_x2, cos(2 * pi * tr_x2),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE = 0.4160','FitBoxToText','on');

% Training net3
setdemorandstream(0689432);
n_start = 100;
NN3 = cell(1, n_start);
perfs3 = zeros(1, n_start);
val_t3 = zeros(n_start, length(val_x3));
for i = 1:n_start
    fprintf('Training %d/%d\n', i, n_start);
    NN3{i} = train(net3, tr_x3, tr_y3);
    val_t3(i, :) = NN3{i}(val_x3);
    perfs3(i) = mse(net3, val_t3(i, :), val_y3);
end
[m3,i3] = min(perfs3)

% plot results (1_3_3)
figure;
plot(tr_x3, tr_y3, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x3, val_y3, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x3, val_t3(i3,:),'-','LineWidth',2,'Color','green');
plot(tr_x3, cos(2 * pi * tr_x3),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE = 0.3518','FitBoxToText','on');

%%% Comparison of noise
clear
setdemorandstream(0689432);

% Size of the training data set is set to 150 observations.
tr_x = linspace(-1, 1, 150);
val_x = linspace(-0.9, 0.9, 150);

% Amount of noise (0.2, 0.6, 1.2)
tr_y1 = cos(2 * pi * tr_x) + 0.2 * randn(size(tr_x));
tr_y2 = cos(2 * pi * tr_x) + 0.6 * randn(size(tr_x));
tr_y3 = cos(2 * pi * tr_x) + 1.2 * randn(size(tr_x));
val_y1 = cos(2 * pi * val_x) + 0.2 * randn(size(val_x));
val_y2 = cos(2 * pi * val_x) + 0.6 * randn(size(val_x));
val_y3 = cos(2 * pi * val_x) + 1.2 * randn(size(val_x));
[sum(tr_y1);sum(tr_y2);sum(tr_y3);sum(val_y1);sum(val_y2);sum(val_y3)]
% The remain characteristics are held constant.
% Training algorithm= Levenberg-Marquardt
% Hidden neurons = 5
```

```
% Early stopping, no regularization
net1 = fitnet(5, 'trainlm');
net2 = fitnet(5, 'trainlm');
net3 = fitnet(5, 'trainlm');

% Training net1
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net1, tr_x, tr_y1);
  val_t1(i, :) = NN1{i}(val_x);
  perfs1(i) = mse(net1, val_t1(i, :), val_y1);
end
[m1,i1] = min(perfs1)

% plot results (1_3_4)
figure;
plot(tr_x, tr_y1, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y1, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE = 0.0372','FitBoxToText','on');

% Training net2
setdemorandstream(0689432);
n_start = 100;
NN2 = cell(1, n_start);
perfs2 = zeros(1, n_start);
val_t2 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN2{i} = train(net2, tr_x, tr_y2);
  val_t2(i, :) = NN2{i}(val_x);
  perfs2(i) = mse(net2, val_t2(i, :), val_y2);
end
[m2,i2] = min(perfs2)

% plot results (1_3_5)
figure;
plot(tr_x, tr_y2, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y2, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t2(i2,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE = 0.3431','FitBoxToText','on');

% Training net3
setdemorandstream(0689432);
n_start = 100;
NN3 = cell(1, n_start);
perfs3 = zeros(1, n_start);
val_t3 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN3{i} = train(net3, tr_x, tr_y3);
  val_t3(i, :) = NN3{i}(val_x);
  perfs3(i) = mse(net3, val_t3(i, :), val_y3);
end
[m3,i3] = min(perfs3)

% plot results (1_3_6)
figure;
plot(tr_x, tr_y3, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y3, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t3(i3,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE = 1.4290','FitBoxToText','on');


%%% Comparison of number of hidden neurons
clear
setdemorandstream(0689432);

% Size of the training data set is set to 150 observations.
tr_x = linspace(-1, 1, 150);
```

```
val_x = linspace(-0.9, 0.9, 150);

% Amount of noise (sd=1.2)
tr_y = cos(2 * pi * tr_x) + 1.2 * randn(size(tr_x));
val_y = cos(2 * pi * val_x) + 1.2 * randn(size(val_x));
[sum(tr_y);sum(val_y)]
% The remain characteristics are held constant.
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net = fitnet(6, 'trainlm');
%net = fitnet(10, 'trainlm');
%net = fitnet(20, 'trainlm');

% Training net
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net, tr_x, tr_y);
  val_t1(i, :) = NN1{i}(val_x);
  perfs1(i) = mse(net, val_t1(i, :), val_y);
end
[m1,i1] = min(perfs1)

% plot results (1_3_7)
figure;
plot(tr_x, tr_y, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m1),'FitBoxToText','on');

%%% Comparison of training algorithms
clear
setdemorandstream(0689432);
% Size of the training data set is set to 150 observations.
tr_x = linspace(-1, 1, 150);
val_x = linspace(-0.9, 0.9, 150);
% Amount of noise (sd=1.2)
tr_y = cos(2 * pi * tr_x) + 1.2 * randn(size(tr_x));
val_y = cos(2 * pi * val_x) + 1.2 * randn(size(val_x));
[sum(tr_y);sum(val_y)]
% The remain characteristics are held constant.
% Early stopping, no regularization
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net = fitnet(6, 'trainlm');
%net = fitnet(6, 'traingd'); %Backpropagation = Gradient descent
%net = fitnet(6, 'trainscg'); % Conjugate gradient
%net = fitnet(6, 'trainbfg'); % Quasi-Newton
% Training net
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net, tr_x, tr_y);
  val_t1(i, :) = NN1{i}(val_x);
  perfs1(i) = mse(net, val_t1(i, :), val_y);
end
[m1,i1] = min(perfs1)

% plot results (1_3_7)
figure;
plot(tr_x, tr_y, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m1),'FitBoxToText','on');

%%% Comparison of stopping rule
clear
setdemorandstream(0689432);
% Size of the training data set is set to 150 observations.
tr_x = linspace(-1, 1, 150);
val_x = linspace(-0.9, 0.9, 150);
% Amount of noise (sd=1.2)
tr_y = cos(2 * pi * tr_x) + 1.2 * randn(size(tr_x));
val_y = cos(2 * pi * val_x) + 1.2 * randn(size(val_x));
[sum(tr_y);sum(val_y)]
% The remain characteristics are held constant.
```

```
% Early stopping, no regularization
net = fitnet(6, 'trainlm');
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net.trainParam.max_fail = 0
%net.trainParam.max_fail = 6
% Training net
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net, tr_x, tr_y);
  val_t1(i, :) = NN1{i}(val_x);
  perfs1(i) = mse(net, val_t1(i, :), val_y);
end
[m1,i1] = min(perfs1)

% plot results (1_3_7)
figure;
plot(tr_x, tr_y, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m1),'FitBoxToText','on');

%%% Comparison of regularization constant
clear
setdemorandstream(0689432);
% Size of the training data set is set to 150 observations.
tr_x = linspace(-1, 1, 150);
val_x = linspace(-0.9, 0.9, 150);
% Amount of noise (sd=1.2)
tr_y = cos(2 * pi * tr_x) + 1.2 * randn(size(tr_x));
val_y = cos(2 * pi * val_x) + 1.2 * randn(size(val_x));
[sum(tr_y);sum(val_y)]
% The remain characteristics are held constant.
% Early stopping, no regularization
net = fitnet(6, 'trainlm');

% Modifying regularization constant
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net.performParam.regularization = 0
%net.performParam.regularization = 0.1
%net.performParam.regularization = 0.25
%net.performParam.regularization = 0.5

% Training net
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net, tr_x, tr_y);
  val_t1(i, :) = NN1{i}(val_x);
  perfs1(i) = mse(net, val_t1(i, :), val_y);
end
[m1,i1] = min(perfs1)

% plot results (1_3_7)
figure;
plot(tr_x, tr_y, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m1),'FitBoxToText','on');

%%% Comparison of initial weights
clear
setdemorandstream(0689432);
% Size of the training data set is set to 150 observations.
tr_x = linspace(-1, 1, 150);
val_x = linspace(-0.9, 0.9, 150);
% Amount of noise (sd=1.2)
tr_y = cos(2 * pi * tr_x) + 1.2 * randn(size(tr_x));
val_y = cos(2 * pi * val_x) + 1.2 * randn(size(val_x));
[sum(tr_y);sum(val_y)]
% The remain characteristics are held constant.
% Early stopping, no regularization
net = fitnet(6, 'trainlm');

```

```
% Training net
setdemorandstream(0689432);
% UNCOMMENT THE SEED THAT YOU WANT TO USE
%setdemorandstream(13245);
%setdemorandstream(67890);
%setdemorandstream(111111);
n_start = 100;
NN1 = cell(1, n_start);
perfs1 = zeros(1, n_start);
val_t1 = zeros(n_start, length(val_x));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net, tr_x, tr_y);
  val_t1(i, :) = NN1{i}(val_x);
  perfs1(i) = mse(net, val_t1(i, :), val_y);
end
[m1,i1] = min(perfs1)

% plot results (1_3_7)
figure;
plot(tr_x, tr_y, 'o','MarkerSize',2,'MarkerFaceColor','blue',...
    'MarkerEdgeColor','blue');
hold on;
plot(val_x, val_y, 'o','MarkerSize',2,'MarkerFaceColor','red',...
    'MarkerEdgeColor','red');
plot(val_x, val_t1(i1,:),'-','LineWidth',2,'Color','green');
plot(tr_x, cos(2 * pi * tr_x),'g-','LineWidth',2,'Color','black');
hold off;
legend('Training Set','Validation Set','Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m1),'FitBoxToText','on');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Curse of dimensionality
clear
setdemorandstream(0689432);
% m = 1
% training set
tr_x = linspace(-5,5,100)
tr_y = (sin(pi*tr_x))./(pi*tr_x)
% test set
test_x = linspace(-4.9, 4.9,100)
test_y = (sin(pi*test_x))./(pi*test_x)

% Net1_1: algorithms
setdemorandstream(0689432);
net1_1 = fitnet(5);
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
% net1_1 = fitnet(5,'trainbr'); % Bayesian Regularization
% net1_1 = fitnet(5,'trainbfg'); % Quasi-Newton
% net1_1 = fitnet(5,'trainscg'); % Scaled Conjugate Gradient

net1_1 = configure(net1_1, tr_x, tr_y);
net1_1.inputs{1}.processFcns = {};
net1_1.outputs{2}.processFcns = {};
n_start = 100;
NN1_1 = cell(1, n_start);
y_est1_1 = zeros(n_start, length(test_x));
perfs1_1 = zeros(1, n_start);
for i = 1:n_start
   fprintf('Training %d/%d\n', i, n_start);
   NN1_1{i} = train(net1_1, tr_x, tr_y);
   y_est1_1(i, :) = NN1_1{i}(test_x);
   perfs1_1(i) = mse(net1_1, y_est1_1(i,:), test_y);
end
[m1_1,i1_1] = min(perfs1_1)

% plot results
figure;
plot(tr_x, tr_y,'g-','LineWidth',2,'Color','black');
hold on;
plot(test_x, y_est1_1(i1_1,:),'-','LineWidth',2,'Color','green');
hold off;
legend('True Function','Approximated Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String','MSE =  1.8436e-04','FitBoxToText','on');

%%%%%%%%%%%%%%%%%%%
% m = 2
%clear
tr_x = repelem(-5:0.1:5,length(-5:0.1:5))
tr_y = repmat(-5:0.1:5,1,length(-5:0.1:5))
tr_aux = sqrt(tr_x.^2 + tr_y.^2)
tr_z = (sin(pi*tr_aux))./(pi*tr_aux)
% all to matrix
tr_xmat = vec2mat(tr_x,101)
tr_ymat = vec2mat(tr_y,101)
tr_zmat = vec2mat(tr_z,101)
figure;surf(tr_xmat,tr_ymat,tr_zmat)
tr_data = [tr_x;tr_y]

% Test set
test_x= repelem(-4.9:0.125:4.9,length(-4.9:0.125:4.9))
test_y = repmat(-4.9:0.125:4.9,1,length(-4.9:0.125:4.9))
test_aux = sqrt(test_x.^2 + test_y.^2)
```

```
test_z = (sin(pi*test_aux))./(pi*test_aux)
% all to matrix
test_xmat= vec2mat(test_x,79)
test_ymat= vec2mat(test_y,79)
test_zmat = vec2mat(test_z,79)
figure;surf(test_xmat,test_ymat,test_zmat)
test_data = [test_x;test_y]

% Train neural network
% net2_1_1: LM algorithm
setdemorandstream(0689432);
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net2_1 = fitnet(50);
% net2_1 = fitnet(50,'trainbr');
% net2_1 = fitnet(50,'trainbfg');
% net2_1 = fitnet(50,'trainscg');

net2_1 = configure(net2_1, tr_data, tr_z);
net2_1.inputs{1}.processFcns = {};
net2_1.outputs{2}.processFcns = {};
%view(net2_1)
n_start = 20;
NN2_1 = cell(1, n_start);
z_est2_1 = zeros(n_start, length(test_z));
perfs2_1 = zeros(1, n_start);
for i = 1:n_start
    fprintf('Training %d/%d\n', i, n_start);
    NN2_1{i} = train(net2_1, tr_data, tr_z);
    z_est2_1(i, :) = NN2_1{i}(test_data);
    perfs2_1(i) = mse(net2_1, z_est2_1(i,:), test_z);
end
[m2_1,i2_1] = min(perfs2_1)

% Transform data to matrix
z_estmat2_1 = vec2mat(z_est2_1(i2_1,:),79)
figure;surf(tr_xmat,tr_ymat,tr_zmat);
figure;surf(test_xmat,test_ymat,z_estmat2_1);


%%%%%%%
% m = 5
%clear
tr_x1 = repmat(repelem(-5:2.5:5,length(-5:2.5:5)^4),1,length(-5:2.5:5))
tr_x2 = repmat(repelem(-5:2.5:5,length(-5:2.5:5)^3),1,length(-5:2.5:5)^2)
tr_x3 = repmat(repelem(-5:2.5:5,length(-5:2.5:5)^2),1,length(-5:2.5:5)^3)
tr_x4 = repmat(repelem(-5:2.5:5,length(-5:2.5:5)),1,length(-5:2.5:5)^4)
tr_x5 = repmat(-5:2.5:5,1,length(-5:2.5:5).^5)
%[tr_x1;tr_x2;tr_x3; tr_x4 ;tr_x5]
tr_aux = sqrt(tr_x1.^2 + tr_x2.^2 + tr_x3.^2 + tr_x4.^2 + tr_x5.^2)
tr_z = (sin(pi*tr_aux))./(pi*tr_aux)
tr_data = [tr_x1;tr_x2;tr_x3; tr_x4 ;tr_x5]

% Test set
test_x1 = repmat(repelem(-4.75:2.5:4.75,length(-4.75:2.5:4.75)^4),1,length(-4.75:2.5:4.75))
test_x2 = repmat(repelem(-4.75:2.5:4.75,length(-4.75:2.5:4.75)^3),1,length(-4.75:2.5:4.75)^2)
test_x3 = repmat(repelem(-4.75:2.5:4.75,length(-4.75:2.5:4.75)^2),1,length(-4.75:2.5:4.75)^3)
test_x4 = repmat(repelem(-4.75:2.5:4.75,length(-4.75:2.5:4.75)),1,length(-4.75:2.5:4.75)^4)
test_x5 = repmat(-4.75:2.5:4.75,1,length(-4.75:2.5:4.75).^5)
test_data = [test_x1;test_x2;test_x3;test_x4;test_x5]
test_aux = sqrt(test_x1.^2 + test_x2.^2 + test_x3.^2 + test_x4.^2 + test_x5.^2)
test_z = (sin(pi*test_aux))./(pi*test_aux)

% Train neural network
% net3_1: LM algorithm
setdemorandstream(0689432);
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net3_1 = fitnet(100);
%net3_2 = fitnet(100,'trainbr');
%net3_3 = fitnet(100,'trainbfg');
%net3_4 = fitnet(100,'trainscg');
net3_1 = configure(net3_1, tr_data, tr_z);
net3_1.inputs{1}.processFcns = {};
net3_1.outputs{2}.processFcns = {};
n_start = 10;
NN3_1 = cell(1, n_start);
z_est3_1 = zeros(n_start, length(test_z));
perfs3_1 = zeros(1, n_start);
for i = 1:n_start
    fprintf('Training %d/%d\n', i, n_start);
    NN3_1{i} = train(net3_1, tr_data, tr_z);
    z_est3_1(i, :) = NN3_1{i}(test_data);
    perfs3_1(i) = mse(net3_1, z_est3_1(i,:), test_z);
end
[m3_1,i3_1] = min(perfs3_1)
```

- **Script Exercise Session 2**

```
    % Setting working directory
cd '...'
%% Santa Fe laser data - time-series prediction
clear
% Importing data
lasertrain = importdata('lasertrain.dat');
laserpred = importdata('laserpred.dat');

```

```
% Plotting data
figure;plot(lasertrain);
figure;plot(laserpred);

% Transforming the format
lasertrain = tonndata(lasertrain,false,false)
laserpred = tonndata(laserpred,false,false)

%%% Comparing lags
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net1 = narnet(1:2,10);
%net1 = narnet(1:7,10);
%net1 = narnet(1:14,10);
%net1 = narnet(1:21,10);

% Training net1
setdemorandstream(0689432);
n_start = 100;
NN1_1 = cell(1, n_start);
perfs1_1 = zeros(1, n_start);
test_t1_1(n_start).output = zeros(2, length(laserpred));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  [Xs,Xi,Ai,Ts] = preparets(net1,{},{},lasertrain);
  NN1_1{i} = closeloop(train(net1,Xs,Ts,Xi,Ai));
  yini = lasertrain(end-NN1_1{i}.numLayerDelays+1:end); % Last values from training data
  [Xs,Xi,Ai] = preparets(NN1_1{i},{},{},[yini laserpred]);
% predict on test data
  test_t1_1(i).output = NN1_1{i}(Xs,Xi,Ai)
  perfs1_1(i) = perform(NN1_1{i}, laserpred, test_t1_1(i).output)
end
[m1_1,i1_1] = min(perfs1_1)

figure;
plot(fromnndata(test_t1_1(i1_1).output, true, false, false),'-','LineWidth',2,'Color','green');
hold on
plot(fromnndata(laserpred, true, false, false),'g-','LineWidth',2,'Color','black');
hold off;
legend('Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m1_1),'FitBoxToText','on');
% White noise
E = gsubtract(laserpred,test_t1_1(i1_1).output);
ploterrcorr(E)

%%% Comparing hidden units
% UNCOMMENT THE NET THAT YOU WANT TO TRAIN
net1 = narnet(1:14,10);
%net1 = narnet(1:14,20);
%net1 = narnet(1:14,30);
%net1 = narnet(1:14,50);

% Training net1
setdemorandstream(0689432);
n_start = 100;
NN2_1 = cell(1, n_start);
perfs2_1 = zeros(1, n_start);
test_t2_1(n_start).output = zeros(2, length(laserpred));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  [Xs,Xi,Ai,Ts] = preparets(net1,{},{},lasertrain);
  NN2_1{i} = closeloop(train(net1,Xs,Ts,Xi,Ai));
  yini = lasertrain(end-NN2_1{i}.numLayerDelays+1:end); % Last values from training data
  [Xs,Xi,Ai] = preparets(NN2_1{i},{},{},[yini laserpred]);
% predict on test data
  test_t2_1(i).output = NN2_1{i}(Xs,Xi,Ai)
  perfs2_1(i) = perform(NN2_1{i}, laserpred, test_t2_1(i).output)
end
[m2_1,i2_1] = min(perfs2_1)

figure;
plot(fromnndata(test_t2_1(i2_1).output, true, false, false),'-','LineWidth',2,'Color','green');
hold on
plot(fromnndata(laserpred, true, false, false),'g-','LineWidth',2,'Color','black');
hold off;
legend('Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m2_1),'FitBoxToText','on');
% White noise
E = gsubtract(laserpred,test_t2_1(i2_1).output);
ploterrcorr(E)

%%% Comparing training algorithms
net1 = narnet(1:14,50);

% UNCOMMENT THE ALGORITHM YOU WANT TO USE
net1.trainFcn = 'trainlm';
%net1.trainFcn = 'traingd';
%net1.trainFcn = 'trainscg';
%net1.trainFcn = 'trainbfg';

% Training net1
setdemorandstream(0689432);
n_start = 100;
NN3_1 = cell(1, n_start);
perfs3_1 = zeros(1, n_start);
```

```
test_t3_1(n_start).output = zeros(2, length(laserpred));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  [Xs,Xi,Ai,Ts] = preparets(net1,{},{},lasertrain);
  NN3_1{i} = closeloop(train(net1,Xs,Ts,Xi,Ai));
  yini = lasertrain(end-NN3_1{i}.numLayerDelays+1:end); % Last values from training data
  [Xs,Xi,Ai] = preparets(NN3_1{i},{},{},[yini laserpred]);
% predict on test data
  test_t3_1(i).output = NN3_1{i}(Xs,Xi,Ai)
  perfs3_1(i) = perform(NN3_1{i}, laserpred, test_t3_1(i).output)
end
[m3_1,i3_1] = min(perfs3_1)

figure;
plot(fromnndata(test_t3_1(i3_1).output, true, false, false),'-','LineWidth',2,'Color','green');
hold on
plot(fromnndata(laserpred, true, false, false),'g-','LineWidth',2,'Color','black');
hold off;
legend('Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m3_1),'FitBoxToText','on');
% White noise
E = gsubtract(laserpred,test_t3_1(i3_1).output);
ploterrcorr(E)

%%% Comparing training algorithms
net1 = narnet(1:14,50);
% UNCOMMENT THE REGULARIZATION YOU WANT TO USE
net1.performParam.regularization = 0
%net1.performParam.regularization = 0.1
%net1.performParam.regularization = 0.25
%net1.performParam.regularization = 0.5

% Training net1
setdemorandstream(0689432);
n_start = 100;
NN4_1 = cell(1, n_start);
perfs4_1 = zeros(1, n_start);
test_t4_1(n_start).output = zeros(2, length(laserpred));
for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  [Xs,Xi,Ai,Ts] = preparets(net1,{},{},lasertrain);
  NN4_1{i} = closeloop(train(net1,Xs,Ts,Xi,Ai));
  yini = lasertrain(end-NN4_1{i}.numLayerDelays+1:end); % Last values from training data
  [Xs,Xi,Ai] = preparets(NN4_1{i},{},{},[yini laserpred]);
% predict on test data
  test_t4_1(i).output = NN4_1{i}(Xs,Xi,Ai)
  perfs4_1(i) = perform(NN4_1{i}, laserpred, test_t4_1(i).output)
end
[m4_1,i4_1] = min(perfs4_1)

figure;
plot(fromnndata(test_t4_1(i4_1).output, true, false, false),'-','LineWidth',2,'Color','green');
hold on
plot(fromnndata(laserpred, true, false, false),'g-','LineWidth',2,'Color','black');
hold off;
legend('Approximated Function','True Function');
dim = [0.2 0.6 0.3 0.3];
annotation('textbox',dim,'String',sprintf('MSE = %.4f',m4_1),'FitBoxToText','on');
% White noise
E = gsubtract(laserpred,test_t4_1(i4_1).output);
ploterrcorr(E)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Alfabet recognition
clear
openExample('nnet/appcr1')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Breast Cancer Wisconsin - classification problem
clear
filebreast = 'C:\Users\danie\Documents\Daniel Gil\KULeuven\Stage 2\Term 1\Data Mining and Neural Networks\Exercise
      Sessions\Datasets\bcw.mat'
load(filebreast)

aux(Y==0) = 1
aux(Y==1) = 0;
YY= [Y; aux]
%vec2ind(YY) 1 is cancer and 2 is no cancer
clearvars aux

% division of data
% 90% for designing the neural net
% 10% for testing
Q = size(X,2) %569 columns
Q1 = floor(Q * 0.80); % 455 columns
Q2 = Q - Q1; % 114 columns
setdemorandstream(0689432)
ind = randperm(Q);
ind1 = ind(1:Q1);
ind2 = ind(Q1 + (1:Q2));
tr_x = X(:, ind1);
tr_y = YY(:, ind1);
test_x = X(:, ind2);
test_y = YY(:, ind2);
[sum(tr_x(1,:));sum(tr_y(1,:));sum(test_x(1,:));sum(test_y(1,:))]
```

```
200
201 %%% Comparing hidden units
202 % UNCOMMENT THE NET THAT YOU WANT TO TRAIN
203 net1 = patternnet(2);
204 %net1 = patternnet(5);
205 %net1 = patternnet(10);
206 %net1 = patternnet(15);
207
208 % Training net1
209 setdemorandstream(0689432);
210 n_start = 100;
211 NN1 = cell(1, n_start);
212 perfs1 = zeros(1, n_start);
213 test_t1(n_start).output = zeros(2, Q2);
214 for i = 1:n_start
215   fprintf('Training %d/%d\n', i, n_start);
216   NN1{i} = train(net1, tr_x, tr_y);
217   test_t1(i).output = NN1{i}(test_x);
218   perfs1(i) = perform(net1, test_y, test_t1(i).output);
219 end
220 [m1,i1] = min(perfs1)
221
222 [c1,cm1] = confusion(test_y,test_t1(i1).output)
223 fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c1));
224 fprintf('Percentage Incorrect Classification : %f%%\n', 100*c1);
225 figure;plotconfusion(test_y,test_t1(i1).output)
226 figure;plotroc(test_y,test_t1(i1).output)
227 dim = [0.7 0.2 0.3 0.3];
228 corr = 100*(1-c1);
229 incorr = 100*c1;
230 str = {sprintf('CE = %.4f',m1),sprintf('% Correct = %.2f',corr),'sprintf('% Correct = %.2f',incorr)};
231 annotation('textbox',dim,'String',str,'FitBoxToText','on');
232
233 %%% Comparing algorithms
234 % UNCOMMENT THE NET THAT YOU WANT TO TRAIN
235 net1 = patternnet(5,'trainscg');
236 %net1 = patternnet(5,'trainlm'); % Levenberg
237 %net1 = patternnet(5,'traingd'); % Backpropagation = Gradient descent
238 %net1 = patternnet(5,'trainbfg'); % Quasi-Newton
239
240 % Training net1
241 setdemorandstream(0689432);
242 n_start = 100;
243 NN1 = cell(1, n_start);
244 perfs1 = zeros(1, n_start);
245 test_t1(n_start).output = zeros(2, Q2);
246 for i = 1:n_start
247   fprintf('Training %d/%d\n', i, n_start);
248   NN1{i} = train(net1, tr_x, tr_y);
249   test_t1(i).output = NN1{i}(test_x);
250   perfs1(i) = perform(net1, test_y, test_t1(i).output);
251 end
252 [m1,i1] = min(perfs1)
253
254 [c1,cm1] = confusion(test_y,test_t1(i1).output)
255 fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c1));
256 fprintf('Percentage Incorrect Classification : %f%%\n', 100*c1);
257 figure;plotconfusion(test_y,test_t1(i1).output)
258 figure;plotroc(test_y,test_t1(i1).output)
259 dim = [0.7 0.2 0.3 0.3];
260 corr = 100*(1-c1);
261 incorr = 100*c1;
262 str = {sprintf('CE = %.4f',m1),sprintf('% Correct = %.2f',corr),'sprintf('% Correct = %.2f',incorr)};
263 annotation('textbox',dim,'String',str,'FitBoxToText','on');
264
265
266 %%% Comparing regularization
267 net1 = patternnet(5,'trainscg');
268
269 % Modifying regularization constant
270 % UNCOMMENT THE REGULARIZATION YOU WANT TO USE
271 net1.performParam.regularization = 0
272 %net1.performParam.regularization = 0.01
273 %net1.performParam.regularization = 0.1
274 %net1.performParam.regularization = 0.2
275
276 % Training net1
277 setdemorandstream(0689432);
278 n_start = 100;
279 NN1 = cell(1, n_start);
280 perfs1 = zeros(1, n_start);
281 test_t1(n_start).output = zeros(2, Q2);
282 for i = 1:n_start
283   fprintf('Training %d/%d\n', i, n_start);
284   NN1{i} = train(net1, tr_x, tr_y);
285   test_t1(i).output = NN1{i}(test_x);
286   perfs1(i) = perform(net1, test_y, test_t1(i).output);
287 end
288 [m1,i1] = min(perfs1)
289
290 [c1,cm1] = confusion(test_y,test_t1(i1).output)
291 fprintf('Percentage Correct Classification   : %f%%\n', 100*(1-c1));
292 fprintf('Percentage Incorrect Classification : %f%%\n', 100*c1);
293 figure;plotconfusion(test_y,test_t1(i1).output)
294 figure;plotroc(test_y,test_t1(i1).output)
295 dim = [0.7 0.2 0.3 0.3];
```

```
corr = 100*(1-c1);
incorr = 100*c1;
str = {sprintf('CE = %.4f',m1),sprintf('% Correct = %.2f',corr),'sprintf('% Correct = %.2f',incorr)};
annotation('textbox',dim,'String',str,'FitBoxToText','on');
```

- **Script Exercise Session 3**

```
% Setting working directory
cd '...'

%% Dimensionality reduction by PCA analysis
clear
a) Investigate PCA analysis in order to achieve a dimensionality reduction
load cho_dataset

% Standardize the variables
[pn, std_p] = mapstd(choInputs);
[tn, std_t] = mapstd(choTargets);
% PCA
[pp, pca_p] = processpca(pn, 'maxfrac', 0.001);
[m, n] = size(pp)
% To get scores after pca (pp object) just do pca_p.transform*pn

% (b) For the case of 21 inputs, define a training, validation and test set and apply the Levenberg-Marquardt
      algorithm:
% Set indices for test, validation and training sets
Test_ix = 2:4:n;
Val_ix = 4:4:n;
Train_ix = [1:4:n 3:4:n];
% Configure a network
% LM algorithm
net1 = fitnet(5);
net1.divideFcn = 'divideind';
net1.divideParam = struct('trainInd', Train_ix, ...
    'valInd', Val_ix, ...
    'testInd', Test_ix);

% Training net1
setdemorandstream(0689432);
n_start = 100;
NN1 = cell(1, n_start);
mse_train1 = zeros(1, n_start);
mse_test1 = zeros(1, n_start);
yhat_train1(n_start).output = zeros(n_start, length(tn(:,Train_ix)));
yhat_test1(n_start).output = zeros(n_start, length(tn(:,Test_ix)));

for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN1{i} = train(net1, pn, tn);
  yhat_train1(i).output = NN1{i}(pn(:, Train_ix));
  yhat_test1(i).output = NN1{i}(pn(:, Test_ix));
  mse_train1(i) = mse(net1, tn(:,Train_ix), yhat_train1(i).output);
  mse_test1(i) = mse(net1, tn(:,Test_ix), yhat_test1(i).output);
end
[m1,i1] = min(mse_test1)

% Investigate whether the performance can be improved by means of Bayesian regularization (trainbr). Compare the
      results on test data.
net2 = fitnet(5,'trainbr');
net2.divideFcn = 'divideind';
net2.divideParam = struct('trainInd', Train_ix, ...
    'valInd', Val_ix, ...
    'testInd', Test_ix);

% Training net2
setdemorandstream(0689432);
n_start = 100;
NN2 = cell(1, n_start);
mse_train2 = zeros(1, n_start);
mse_test2 = zeros(1, n_start);
yhat_train2(n_start).output = zeros(n_start, length(tn(:,Train_ix)));
yhat_test2(n_start).output = zeros(n_start, length(tn(:,Test_ix)));

for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN2{i} = train(net2, pn, tn);
  yhat_train2(i).output = NN2{i}(pn(:, Train_ix));
  yhat_test2(i).output = NN2{i}(pn(:, Test_ix));
  mse_train2(i) = mse(net2, tn(:,Train_ix), yhat_train2(i).output);
  mse_test2(i) = mse(net2, tn(:,Test_ix), yhat_test2(i).output);
end
[m2,i2] = min(mse_test2)

mse_test1(i1)
mse_test2(i2)
mse_train1(i1)
mse_train2(i2)

% (c) Compare the training results of the case of 21 inputs (original inputs) and 4 inputs (after dimensionality
      reduction by PCA) by applying trainbr. Which choice would you make between the two options? Motivate your choice.
net3 = fitnet(5,'trainbr');
net3.divideFcn = 'divideind';
net3.divideParam = struct('trainInd', Train_ix, ...
    'valInd', Val_ix, ...
    'testInd', Test_ix);
```

```
% Training net2
setdemorandstream(0689432);
n_start = 100;
NN3 = cell(1, n_start);
mse_train3 = zeros(1, n_start);
mse_test3 = zeros(1, n_start);
yhat_train3(n_start).output = zeros(n_start, length(tn(:,Train_ix)));
yhat_test3(n_start).output = zeros(n_start, length(tn(:,Test_ix)));

for i = 1:n_start
  fprintf('Training %d/%d\n', i, n_start);
  NN3{i} = train(net3, pp, tn);
  yhat_train3(i).output = NN3{i}(pp(:, Train_ix));
  yhat_test3(i).output = NN3{i}(pp(:, Test_ix));
  mse_train3(i) = mse(net3, tn(:,Train_ix), yhat_train3(i).output);
  mse_test3(i) = mse(net3, tn(:,Test_ix), yhat_test3(i).output);
end
[m3,i3] = min(mse_test3)

[mse_train1(i1) mse_test1(i1);mse_train2(i2) mse_test2(i2);mse_train3(i3) mse_test3(i2)]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% (2) Input selection by Automatic Relevance Determination (ARD)
demard
demev1

clear
fileion = 'ionstart.mat'
load(fileion)
YY=[Y>=0];

randn('state', 0689432);
rand('state', 0689432);

% Set up network parameters.
nin = 33;                       % Number of inputs.
nhidden = 2;                    % Number of hidden units.
nout = 1;                       % Number of outputs.
aw1 = 0.01*ones(1, nin);        % First-layer ARD hyperparameters.
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);

% Set up vector of options for the optimiser.
nouter = 5;                     % 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) = 500;              % 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, YY, 'scg');
  [net, gamma] = evidence(net, X, YY, ninner);
  fprintf(1, '\n\nRe-estimation cycle %d:\n', k);
  disp('The first 33 alphas are the hyperparameters for the corresponding');
  disp('input to hidden unit weights.  The remainder are the hyperparameters');
  disp('for the hidden unit biases, second layer weights and output unit')
  disp('biases, respectively.')
  fprintf(1, '  alpha =  %8.5f\n', net.alpha);
  fprintf(1, '  beta  =  %8.5f\n', net.beta);
  fprintf(1, '  gamma =  %8.5f\n\n', gamma);
  %disp(' ')
  %disp('Press any key to continue.')
  %pause
end

% Hyperparameters
 net.alpha
% Weights
net2.w1

%% Second neural net without some variables
X2 = X
X2(:,[8 15 17 19 28])=[];
YY=[Y>=0];

randn('state', 0689432);
rand('state', 0689432);

% Set up network parameters.
nin = 28;                       % Number of inputs.
nhidden = 2;                    % Number of hidden units.
nout = 1;                       % Number of outputs.
aw1 = 0.01*ones(1, nin);        % First-layer ARD hyperparameters.
ab1 = 0.01;                     % Hyperparameter for hidden unit biases.
aw2 = 0.01;                     % Hyperparameter for second-layer weights.
ab2 = 0.01;                     % Hyperparameter for output unit biases.
```

```
182  beta = 50.0;                    % Coefficient of data error.
183
184  % Create and initialize network.
185  prior2 = mlpprior(nin, nhidden, nout, aw1, ab1, aw2, ab2);
186  net2 = mlp(nin, nhidden, nout, 'logistic', prior2, beta);
187
188  % Set up vector of options for the optimiser.
189  nouter = 5;                     % Number of outer loops
190  ninner = 10;                    % Number of inner loops
191  options = zeros(1,18);          % Default options vector.
192  options(1) = 1;                 % This provides display of error values.
193  options(2) = 1.0e-7;    % This ensures that convergence must occur
194  options(3) = 1.0e-7;
195  options(14) = 500;              % Number of training cycles in inner loop.
196
197  % Train using scaled conjugate gradients, re-estimating alpha and beta.
198  for k = 1:nouter
199    net2 = netopt(net2, options, X2, YY, 'scg');
200    [net2, gamma] = evidence(net2, X2, YY, ninner);
201    fprintf(1, '\n\nRe-estimation cycle %d:\n', k);
202    disp('The first 28 alphas are the hyperparameters for the corresponding');
203    disp('input to hidden unit weights.  The remainder are the hyperparameters');
204    disp('for the hidden unit biases, second layer weights and output unit')
205    disp('biases, respectively.')
206    fprintf(1, '  alpha =  %8.5f\n', net2.alpha);
207    fprintf(1, '  beta  =  %8.5f\n', net2.beta);
208    fprintf(1, '  gamma =  %8.5f\n\n', gamma);
209    %disp(' ')
210    %disp('Press any key to continue.')
211    %pause
212  end
213  % Hyperparameters
214  net2.alpha
215  % Weights
216  net2.w1
```