KU LEUVEN

Master of Statistics

Final project

Data Mining and Neural Networks

Nozomi Takemura
M.Sc. statistics
(nozomi.takemura@student.kuleuven.be)

Supervisor: Prof. dr. ir. Johan Suykens

June 27, 2018

Contents

Exercise Session 1

1 Function Approximation (noiseless case)

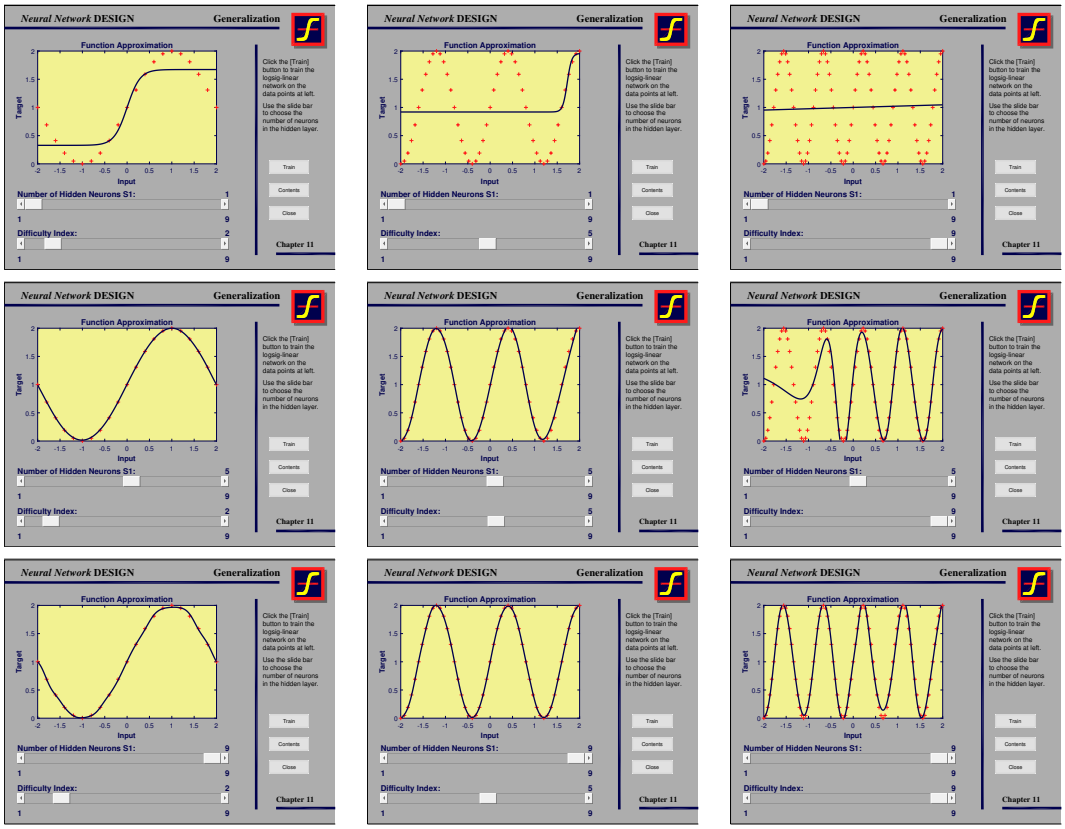


Figure 1: The relationship between the number of hidden neurons and complexity of the function

From the figure, the following points are found regarding the relationship between the number of hidden neurons N and the complexity of the function to approximate (C). Firstly, as the number of neurons increases, the corresponding neural network (NN) tends to approximate more accurately an unobserved function from the given data. Secondly, the higher the degree of complexity of a underlying function is, the more difficult a neural network correctly estimates such function.

Indeed, observing the fig.1, we first notice that a network with only 1 neuron (top figures) poorly fits the data and cannot learn the underlying function properly. For example, when the complexity of the function is medium or higher such as *Difficulty Index*: 5 or 9, the network ends up with estimating a function which is almost linear in the most of the input space. Even when a model complexity is small with Difficulty Index 2, the original function is not well approximated.

On the other hand, when the number of neurons is relatively large, 5 (middle figures) or 9 (bottom figures) for example, the hidden true function is almost perfectly estimated at least up to the difficulty index 5. However, with the number of neurons 5 and function complexity index 9, the neural network suffers from learning the function correctly, though it is better than the one with one hidden neuron. For the case where there are 9 hidden neurons, the neural network looks perfectly being trained.

Underfitting can be described as a situation such that the function is parameterized with a model consisting fewer parameters than the true model's, while overfitting can be explained as a case where a model with more parameters than the true model's is selected to approximate the function [6]. In the current setting,

we can say that underfitting occurs when a neural network with a small number of neurons is used to learn a function with high complexity. For instance, it can be said that three cases of the first row and the rightest case in the middle row in the fig.1 are underfitting. On the contrary, we can say that overfitting happens when a neural network whose number of neurons is large is employed to fit a function with little complexity. For example, the overfitting cases are shown in the fig.2, where the actual function is smoothly estimated by the neural network with one neuron whereas outputs from neural networks with 8 and 9 neurons respectively end up yielding wiggly lines.

One of the possible reasons why the number of neurons has an effect on the overfitting/underfitting is that it determines the number of parameters: weight w.

Thus, although the input data are noise free, the choice of number of neurons corresponding to the unseen function would be quite important.

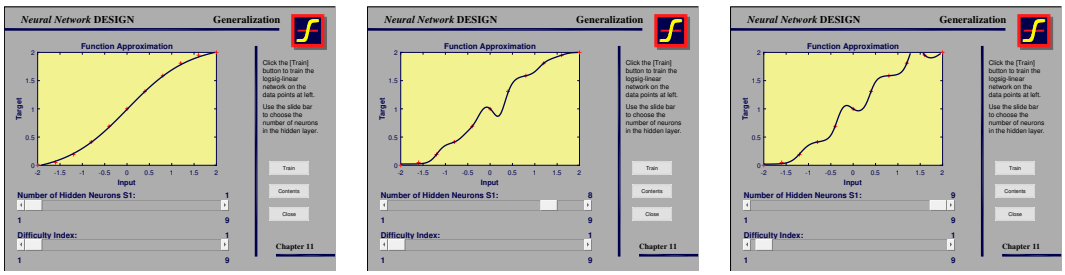


Figure 2: Overfitting cases: From left to right, $(N, C) = (1, 1), (8, 1), (9, 1)$

2 The role of the hidden layer and output layer

Let a neural network with P input patterns, n features, and one output layer composed of one neuron be considered. This is a perceptron, the output for y of pth observation can be modeled with a output layer activation function f as following:

$$y_p = f((\sum_{i=1}^{n} w_{i,p}x_{i,p}) + b), \quad f : \text{activation function}, \; w_{i,p} : \text{interconnection weights}, \; b : \text{bias}.$$

If we take a pure linear activation function for f, namely $f : X \to X$, the above equation can be written as

$$y_p = \sum_{i=1}^{n}(w_{i,p}x_{i,p}) + b.$$

This is exactly as same as the linear regression form. Thus, one layer is required to perform linear regression using neural network. To be detailed, in this layer, one neuron is needed. As mentioned above, identity function needs to be employed as the transfer function in this layer.

2.1 $-cos(0.8\pi x_p)$

In this problem, a function $-cos(0.8\pi x_p)$ with $1 \leq x \leq 1$ is considered. The fig.3 shows the plot of responses of the function $f(x) = -cos(0.8x_p)$ with 21 inputs of x equally spaced in the range $[0, 1]$. The line appears to be almost linear in the interval $[0, 1]$ except for both edges, implying that a linear model wound mostly capture the relationships between x and y. However, it is important to note that it is impossible to describe the behavior of this function roughly from 0 to o.2 and from 0.9 to 1 with a linear model.

2.2 Neural network with one hidden layer with two neurons

Following to a single neurons model, a neural network with one hidden layer with two neurons is thought. $\boldsymbol{x}'$ and $\boldsymbol{y}'$ are used as 1×21 input and output row vectors for training. The training is done with the default setting of matlab. After the training, using the function **hidden layer weights** and **hidden layer transfer function**, weights (w), biases b, and activation function f in the hidden unit are extracted. Based on these, activations from the hidden layer, $x_p^1 = f(x_p w_{1,p} + b_1)$ and $x_p^2 = f(x_p w_{2,p} + b_2)$, are gained. Note that the transfer function of the hidden neuron is tangent hypobolic function and that for the output neuron is identity function as default in matlab.

The fig.4 compares activations and y along with input pattern p. To model the relationship between y_p and (x_p^1), it would be enough to take the sum of negatively weighted x_p^1 and x_p^2. In fact, it is revealed that weights associated to the output neuron are indeed negative with $-0.2124, -0.9096$.

In the way similar to the above, the $output_p$ is calculated with the weights, bias, and transfer function of the output neuron in the following way: $output_p = f^{'}(\sum_{i=1}^{2}(w_{i,p}^{'} x_p^i) + b^{'})$. As seen in the fig.5, the trained network succeeds in estimating responses. Since the input data are not contaminated by noise in this example, the output from the network would be well capturing the original function.

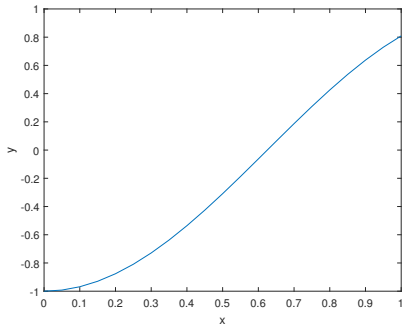


Figure 3: $-cos(0.8x_p$

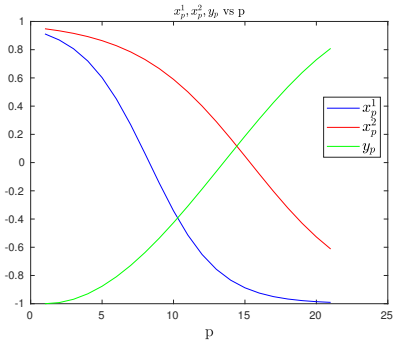


Figure 4: pth input pattern vs $x_p^1, x_p^2, and y_p$

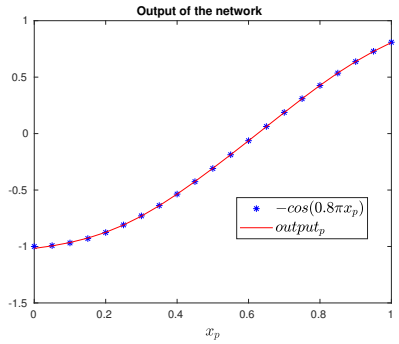


Figure 5: $y_p = -cos(0.8\pi x_p)$ vs output$_p$

3 Function Approximation (noisy case)

In this problem, a set of training data of $f(x) = cos(x)$ contaminated by the Gaussian noise with the input x range $-1 \leq x \leq 1$ is considered. Applying several one hidden layer neural networks, the following factors are studied.

- Effect of size of the training data set: 30, 150, or 1500 (N)
- Effect of the amount of noise: 0.2, 0.6, or 1.2 (σ)
- Effect of the number of neurons: 1, 3, 5, or 7 (N^h)
- Effect of magnitude of the regularization term: 0, 0.3, 0,7 or 1.0 (μ)
- Effect of different training algorithm: Resilient Backpropagation, Scaled Conjugate Gradient, BFGS Quasi-Newton, or Levenberg-Marquardt
- Effect of stopping rule criteria: 'Minimum gradient reached' or 'Validation stop'

3.1 Effect of size of the training data set

The fig.6, 10, 14, and 18 show a shift of approximated function from 30, 150, or 1500 training data set (from left to right in each figure) for different training algorithm. It can be seen that an approximated function is more likely to capture the true cosine curve as size of inputs pattern increases. In fact, when 1500 data points are used as a training data, the approximated function appears to well capture the hidden function.

3.2 Effect of the amount of noise: 0.2, 0.6, or 1.2

The influence of different magnitude of noise is nextly investigated. In particular, the cases of Gaussian noises with amplified standard deviations $0.2Std, 0.6Std, 1.2Std$ are compared. With the reason to easily observe the effect of noise, the setting with $N = 150, r = 0, N^h = 7$ is employed. As can be seen in fig.7, 11, 15, and 19, the true function is almost well described by the approximated functions from different training algorithm when the noise is small ($\sigma = 0.2$). However, as the amount of noise is raised, a network starts learning not only the true signal but noise, resulting in a wiggly moving curve. Especially, when the noise is largest at $\sigma = 1.2$ (the rightest plot in each figure), the difference between true function and the approximated function is remarkable, which would be an example of overfitting.

3.3 the number of neurons: 1, 3, 5, or 7 (N^h)

The number of neurons is closely related to the performance of the neural netwrok to be applied. This would be mainly because the number of parameter (interconnecting weight w) increases as more neurons exist in the hidden layer, leading to explaining the training data more. In fact, when only one neuron is used, the network fails to capture the curvature the data show. However, with seven neurons in the hidden layer, the network succeeds in learning such a curvature and seems to well approximate the true function.

3.4 Effect of magnitude of the regularization term: 0, 0.3, 0,7 or 1.0 (μ)

Regularization is a technique in which the cost function in the optimization problem is slightly modified such that an additional term suppressing the norm of solution vector [2]. To be more concrete, the cost function can be written with regularization term μ as following:

$$E(\omega) = \frac{1}{N}\sum_{n=1}^{N}\{t_n - y(x_n;\omega)\}^2 + \frac{1}{2}\mu\sum_i \omega_i^2.$$

When regularization term is large, the first term is less likely to be concerned compared to the second term, which often results in a simple model such that the norm of parameter vector of that model is small. On the other hand, when μ is small, the first term has more impact on the optimization problem than the 2nd term. Hence, errors tend to be minimized, ending up a more complicated model. Indeed, fig.9, 13, 17, and 21 reveal that the approximated functions look very similar to the underlying true function in the case $\mu = 0$ (leftest plot in each figure), while those in the case $\mu \geq 0.3$ are almost linear and disagree with the hidden function. This is exactly corresponding to the effect of suppression on norm of parameter. We can observe that a seemingly small change from 0 to 0.3 of regularization term has a huge impact on the result. It is possible to automatically find the optimal regularization parameter μ by the bayesian framework [4].

BACKPROPAGATION

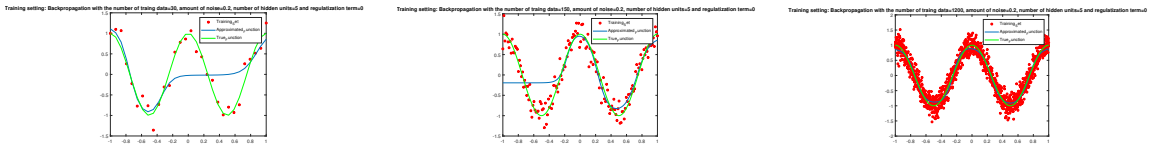


Figure 6: Effects of different number of training data set

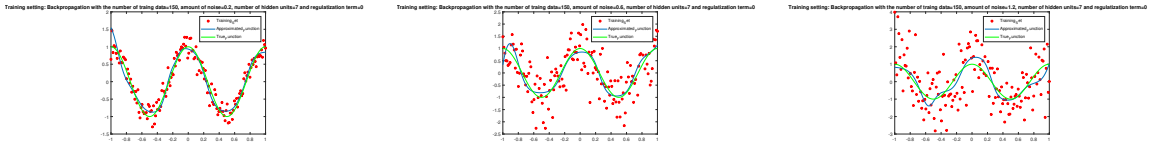


Figure 7: Effects of different size of noise in the training data

Figure 8: Effects of different number of units

Figure 9: Effect of different regularization

conjugate gradient

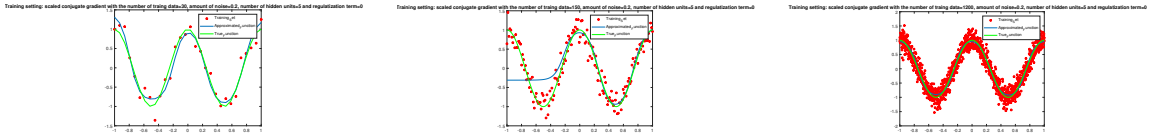


Figure 10: Effects of different number of training data set

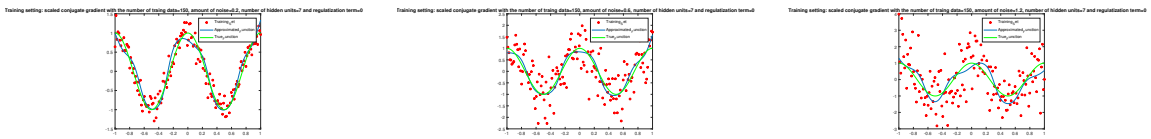


Figure 11: Effects of different size of noise in the training data

Figure 12: Effects of different number of units

Figure 13: Effect of different regularization

quasi-Newton

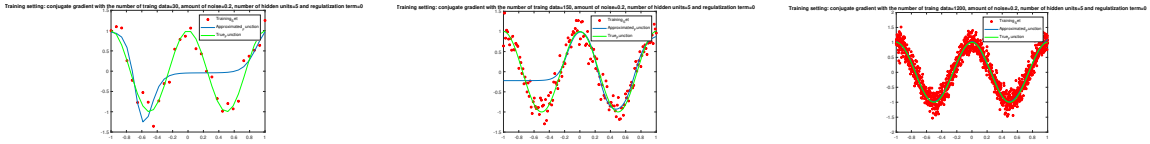


Figure 14: Effects of different number of training data set

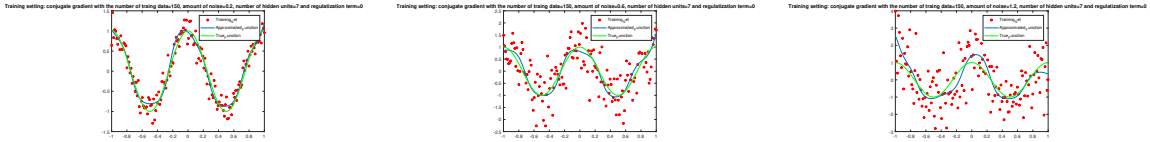


Figure 15: Effects of different size of noise in the training data

Figure 16: Effects of different number of units

Figure 17: Effect of different regularization

Levenberg-Marquardt

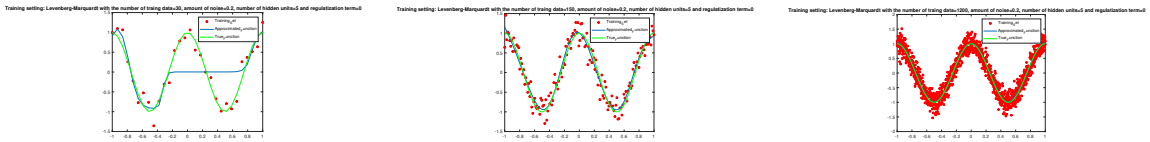


Figure 18: Effects of different number of training data set

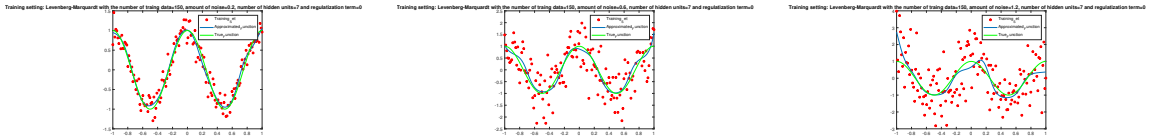


Figure 19: Effects of different size of noise in the training data

Figure 20: Effects of different number of units

Figure 21: Effect of different regularization

3.5 Effect of different training algorithm

The study on influence of different learning algorithms is done here. The table.1, 2, 3, and 4 show occurrences of stopping rules during $3 \times 3 \times 4 \times 4 = 144$ experiments with different size of training data, the amount of noise, the number of neurons, and the magnitude of regularization for each algorithm. Among cases ending in each stopping rule, the averages of best MSEs for training dataset, best MSEs for validation dataset, number of epochs (iterations) to yield best MSEs, and required training times are also reported. To investigate the effect of different learning algorithms, the statistics for "Minimum gradient reached" are only discussed in this part.

Resilient Backpropagation (BP) In this given setting, the goal of the backpropagation approach is to find a solution for an optimization problem $min_{w^l_{1,j}} E = \frac{1}{P}\sum_{p=1}^{P}(\frac{1}{2}\sum_{i=1}^{1}(y_p - x_p^1)^2)$. This method is a gradient descent approach and thus it is generally slow because a very small step length is required to properly search parameter space. In fact, the average training time 0.30 for the case 'Minimum gradient reached' (table.1) is longer than the other methods. Furthermore, the order of average number of epochs is different from those of the others', which would be result of the requirement of small learning rate.

Table 1: Effect of stopping rule (Backpropagation)

BP	frequency	tr.best_perf	tr.best_vperf	tr.best_epoch	tr.time(length(tr.epoch)
'Minimum gradient reached'	35	1.526e-10	1.526e-10	24.4857	0.3040
'Validation stop'	109	0.7087	0.6512	35.2844	0.4236

Conjugate gradient (CG) Conjugate gradient is a type of conjugate gradient descent algorithms. In the conjugate gradient approach, the search direction in each iteration is determined by the conjugation of residuals [10]. It is proven that this method reaches convergence at latest n steps [2]. The average training time 0.23 is shorter than the BP approach but still longer QN and LM procedures.

Table 2: Effect of stopping rule (conjugate gradient)

SCG	frequency	tr.best_perf	tr.best_vperf	tr.best_epoch	tr.time(length(tr.epoch)
'Minimum gradient reached'	48	0.1038	0.09045	2.2292	0.2256
'Validation stop'	96	0.7589	0.6967	20.5729	1.2588

Levenberg-Marquardt algorithm (LM) In the LM algorithm, the following cost function gained by a Taylor expansion $min_{\Delta x} f(x) = f(x_0) + \nabla f(x_0)\Delta x + \frac{1}{2}\Delta x^T H \Delta x$ is optimized with a constrain $||\Delta x||^2$. This yields the Lagrangian function and its solution provides the optimal step as $\Delta = -[H + \lambda I]^{-1}\nabla f(x_0)$ [2]. This enables us to obtain Δx more easily in comparison with the Newton methods in which it is often difficult to calculate the step $\Delta x = -H^{-1}\nabla f(x_0)$ because Hessian is often not able to be inverted. By doing this, LM algorithm finds both the locally steepest direction and optimal step size for each iteration. Furthermore, Hessian H can be approximated using Jacobian matrix and hence computationally expensive calculation of H is avoided. With these reasons, the LM algorithm converges faster than backpropagation approach relying on the steepest gradient descent. Indeed, the average training time 0.09 is much smaller than those for BP and CG. However, its MSE value for validation dataset 0.23 seems to be much larger than those for BP and CG. This would be because the LM method tends to find local minima not global minima. This could be tackled by trying a lot of initial starting points.

Table 3: Effect of stopping rule (Levenberg-Marquardt)

LM	frequency	tr.best_perf	tr.best_vperf	tr.best_epoch	tr.time(length(tr.epoch)
'Minimum gradient reached'	10	0.2711	0.2325	5.400	0.0949
'Validation stop'	36	0.8234	0.7712	29.9167	0.1786

Quasi-Newton algorithm (QN) In QN algorithm, the Hessian H is tried to be approximated with the information during the learning scheme [2]. This leads to faster speed of convergence. For example, its average required training time 0.09 is comparable to LM method and much better than BP and CG. Notablely, its average MSE on validation dataset 0.03 is much better than that for LM approach.

Table 4: Effect of stopping rule (quasi-Newton)

QN	frequency	tr.best_perf	tr.best_vperf	tr.best_epoch	tr.time(length(tr.epoch)
'Minimum gradient reached'	43	0.03208	0.03067	1.2790	0.08802
'Validation stop'	101	0.7045	0.6489	14.33	0.1605

3.6 Effect of stopping rule criteria: 'Minimum gradient reached' or 'Validation stop'

In this part, the difference of statistics is focused between the cases 'Minimum gradient reached' and 'Validation stop' on the table1, 2, 3, and 4. Regarding the threshold value for 'Minimum gradient reached', the default setting of matlab is used: 1×10^{-5} for BP, 1×10^{-6} for (S)CG, 1×10^{-7} for LM, and 1×10^{-6} for QN. For the criterion of 'Validation stop', 6 is used for the all approaches, meaning a training process ends if the MSE for validation set fails to decrease 6 iterations continuously.

Firstly, it is obvious that MSEs of training dataset and validation sets for the cases 'Validation stop' are much larger than those for the other ending case regardless the learning algorithms except for LM approach. This may reflect that minimization of the cost function ends up at a local minima or saddle point, and therefore, the parameter vector which is very different from the true one is found as the optimal solution. Moreover, frequencies of the 'Validation stop' are at least twice as large as the other stop, implying the difficulty of discovering a possible global minima as a solution. Furthermore, for the CG, LM, and QN algorithms, differences in the epochs for two stopping cases appear to be large. This may indicate the importance of an initial point being close to the global minima such that a solution is less likely to be trapped around local minimas or saddle points.

3.7 Effect of regularizations on the cost function

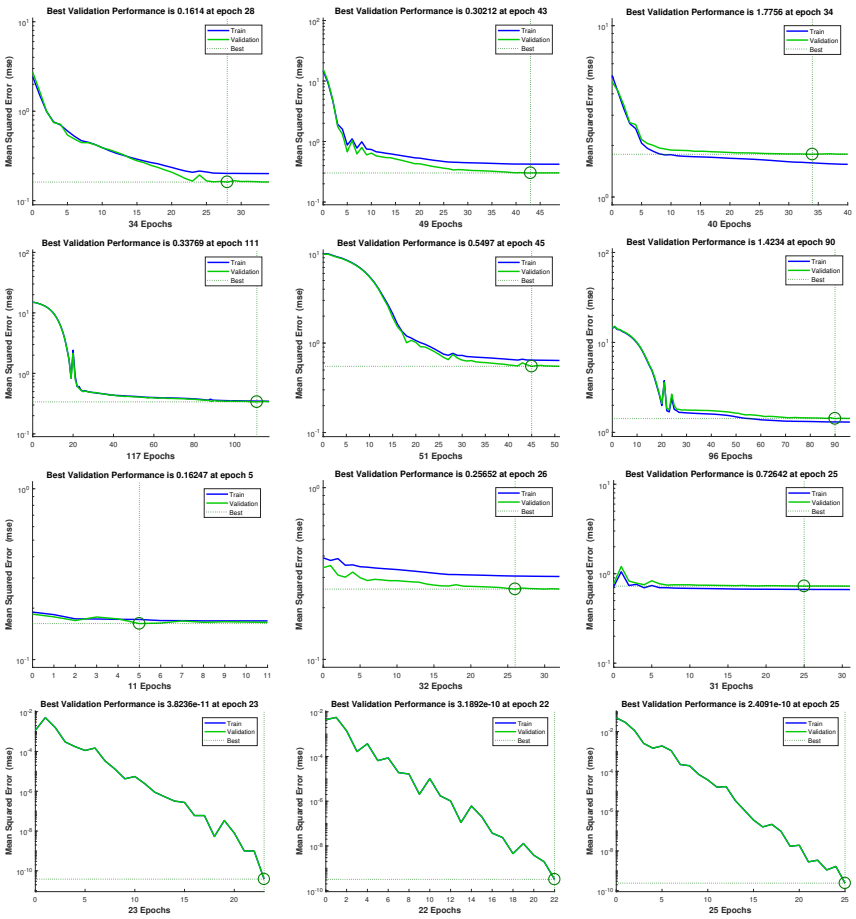


Figure 22: Effect of different regularization $(0, 0.3, 0.7, 1.0)$ (top to bottom) on cost function with different amount of noise $(0.2, 0.6, 1.2)$ (left to right)

4 Curse of dimensionality

In this problem, an approximation of the nonlinear function $f(x) = sinc(\sqrt{\sum_{i=1}^{m} x_i^2})$ with $m = 1, 2$, or 5 with one layer neural network is discussed.

As a set of input data, $10, 60, 150, 210$ are used for comparison. It is important to divide the available data into training, validation, and test data. The training data are used to learn the data, namely obtain the parameter of the neural network. However, there is a risk that the parameter can be learned only to well explain the given training data and not to describe the true population. Hence, it is necessary to check if a trained network can be generalized to another unseen data from the same population. This can be done by studying performance of the trained network on the validation dataset. The final parameter set for the network is often found such that its performance measure, for example mean squared error (MSE), for validation dataset is minimized or become at least under certain threshold value. After the final model is chosen in this way, its performance is reported with test dataset.

If the size of training data is too small, the network can fail to learn much enough from the data. As a rule of thumb, the ratio of training, validation, test sets are $0.7, 0.15, 0.15$, which are used as default setting of dividing ratio for 'dividerand' of matlab. From hereafter, the inputs are automatically separated into three parts with the ratio 0.7, 0.15, and 0.15.

The function approximation is done with different number of inputs $10, 60, 150, 210$ and number of neurons $5, 20, 60, 100$. Because of the long training time especially for high dimensional input, the experiment with much larger size of inputs is not conducted.

3. For each of the cases with different number of inputs m, visually check the neural net approximation by plotting the approxmiated function against sinc(r). For each training algorithm (BP, SCG, QN, LM), approximated functions vs true functions ($sinc(r)$) with 5 dimensional inputs are compared, as shown in the fig.23. It is observable that the true sinc function is not perfectly well approximated even with more than 70000 input points and 100 hidden neurons. it can be also seen that the magnitude of error tends to be higher around $sinc(r) = 0$. Furthermore, the order of absolute error for the approximated function trained with 5 hidden neurons is likely to be larger than that produced from the network with 100 hidden neurons. However, the difference in MSE for both setting under the use of BP, SCG, and QN algorithm does seem to be small. (table.5) When the LM-algorithm is employed, the setting of 100 hidden neurons leads to roughly 20% smaller MSE than the 5 neurons' setting.

Furthermore, a significant effect of number of neurons on the required training time is observed. The order of training time for all training method is 1 when 5 hidden neurons are used, while the order becomes 3 when 100 hidden neurons' setting is adopted (table.5.) Such a difference in order appears to be larger as the dimension of inputs increases. In fact, with 2 dimensional 7140 training inputs, the training time needed for SCG with 100 hidden neurons is 33.3472, order 2. With 1 dimensional 7140 training inputs, this figure even reduces to 22.3687.

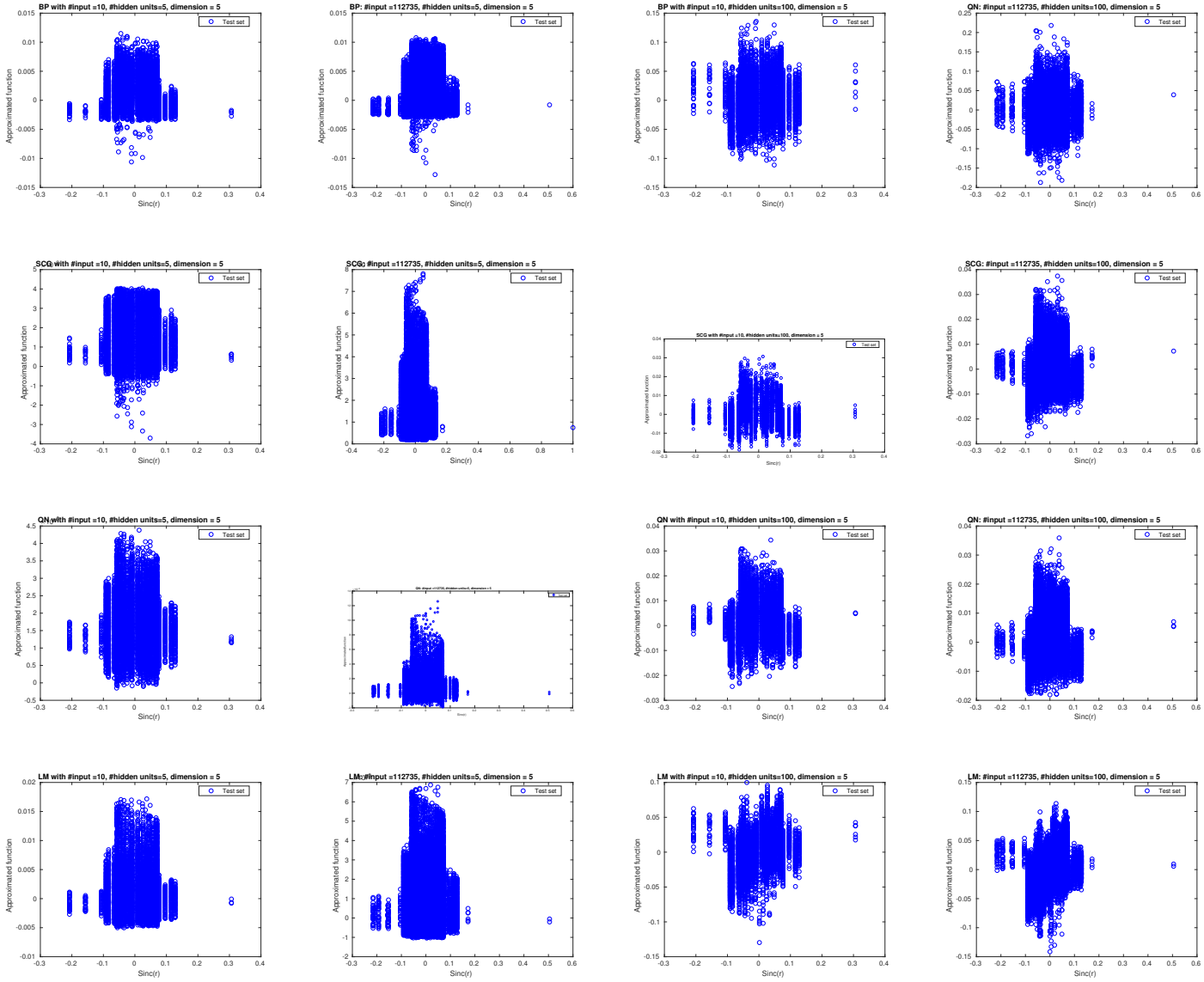


Figure 23: Five dimensional approximated function vs $sinc(r)$. From left to right, (inputs (training), neurons) = (70000, 5), (112735, 5), (70000,100), (112735, 100). From top to bottom, employed training algorith is (BP, SCG, QN, LM).

4. Which training methods are most suitable in each of the cases? For the 5-dimensional input setting, the conjugate gradient method would be suitable. This conclusion is reached to consider possible drawbacks of the other three methods in higher dimension setting. First and foremost, in high dimensional inputs, the number of interconnection weights increases, which implies that an appropriate choice of learning algorithm is of importance. Especially, rise in the number of interconnection weights would be problematic for the other three procedures.

In terms of backpropagation method, from the table.5, it can be seen that the required training time for backpropagation algorithm is the longest among all. This would be due to a property of backpropagation algorithm that is a kinds of gradient method to gradually find a 'best' set of parameter yielding a possible local minima. This can leads to a relatively slow convergence, and hence backpropagation would not be the best option in this case.

Quasi-newton method and Levenberg-Marquardt method can be seen as Newton types' algorithm, both of which try to deal with the problem associated to hessian matrix often appearing during the application of Newton type method. Since both methods utilize not only gradient but approximation of hessian, its speed of convergence is faster than the methods solely based on gradient. However, approximation of hessian in the Levenberg-Marquardt method is often dependent on the calculation of Jocobian matrix, which can be computationally expensive in high dimensional input setting with relatively large number of hidden neurons although it does not require calculation of second derivatives. Thus, this method would not suitable. Regarding to quasi-Newton method, it needs a large amount of memory to hold the matrix of interconnection weights; thus this method may not be the best option neither. On the contrary, conjugate gradient method does not require calculating Jacobian matrix nor storing a huge matrix [2].

With these reasons, conjugate gradient method would be recommended in the 5 dimensional input setting. In the 1 and 2 dimensional input setting, the influence of each algorithm would become smaller than the 5 dimensional case. Thus, all of four methods might become a suitable option.

5. How does the increase in dimensionality effect the number of neurons? To investigate the effect of rise in dimensionality on the number of neurons, the MSE for each dimensional case is compared. From the table.5 to 11, it can be found that the orders of MSEs for the network with 100 hidden neurons are all 10^{-2} except for a small number of training inputs such as 70. Especially, if the 1 and 2 dimensional cases are focused, it may be said that quite a few of orders of MSEs seem to be the same for the networks consisting the same size of hidden neurons. These observations could be an example situation where neural networks are not likely to be suffered from curse of dimensionality. In other words, under the usage of neural netwroks, the degree of dimension of the input space will not be responsible for the approximation error [2]. In fact, it is known that the order of magnitude of approximation error for MLPs with one hidden layer is associated to $\frac{1}{n_h}$ with n_h =the number of hidden neurons, and such a downword trend in MSE values may be shown in the table.6 (for $n_h = 10, 100, 1000$).

Table 6: Comparison in approximation errors between neural networks (SCG algorithm) and polynomial regressions for 2 dimensional 7140 training inputs data.

#neurons	MSE(test) set	$\|error\|$ (test set)	traing time	stopping rule
1	0.011	103.9545	0.2768	Minimum gradient reached.
10	0.0136	112.576	0.7212	Validation stop
100	0.0018	51.8089	33.3472	Validation stop.
1000	7.08E-04	31.4158	817.0692	Maximum epoch reached.
#polynimial degree	MSE(training set)	$\|error\|$ (training set)	#parameters	
2	0.0132	510.8501	6	
3	0.0132	510.8531	10	
4	0.0129	507.9505	15	
5	0.0129	507.9227	21	

Table 5: Comparison in approximation errors of neural networks (SCG algorithm) with 100 and 5 hidden neurons for 5 dimensional 7000/112735 training inputs data.

#algorithm	#training inputs	#neurons	MSE	traing time	stopping rule
SCG	7000	5	0.0035	2.7719	'Validation stop.'
		100	0.0036	167.3942	'Validation stop.'
	112735	5	0.0037	2.0001	'Validation stop.'
		100	0.0036	394.9256	'Validation stop.'
LM	7000	5	0.0035	0.8016	Maximum MU reached.
		100	0.0029	117.1173	'Maximum MU reached.'
	112735	5	0.0036	1.1424	Maximum MU reached.
		100	0.0028	228.3522	'Maximum MU reached.'
BP	7000	5	0.0035	2.7421	'Validation stop.'
		100	0.0034	434.0947	Maximum epoch reached.
		100	0.0034	465.1422	Maximum epoch reached.
	112735	5	0.0036	8.7924	'Validation stop.'
		100	0.0045	801.3017	Maximum epoch reached.
QN	7000	5	0.0035	7.0889	Validation stop.
		100	0.0034	302.7339	'Validation stop.'
	112735	5	0.0036	8.7993	Validation stop.
		100	0.0036	313.0824	'Validation stop.'

Table 7: Comparison in approximation errors between neural networks (SCG algorithm) and polynomial regressions for 2 dimensional 1750 training inputs data.

#neurons	MSE(test) set	$\|error\|$ (test set)	traing time	stopping rule
1	0.0142	26.9883	0.2774	Validation stop
10	0.0161	28.8071	0.2713	Validation stop
100	0.0073	23.2204	3.0899	Validation stop.
1000	0.0043	18.7786	185.9925	Maximum epoch reached.
#polynimial degree	MSE(training set)	$\|error\|$ (training set)	#parameters	
2	0.0128	122.7688	6	
3	0.0128	122.7276	10	
4	0.0125	122.5027	15	
5	0.0125	122.5029	21	

Table 8: Comparison in approximation errors between neural networks (SCG algorithm) and polynomial regressions for 2 dimensional 70 training inputs data.

#neurons	MSE(test) set	$\|error\|$ (test set)	traing time	stopping rule
1	0.0046	0.8576	0.1734	Validation stop
10	0.0028	0.6137	0.1555	Validation stop
100	0.0521	0.2124	0.2124	Validation stop.
1000	1.63	16.2993	0.2854	Validation stop.
#polynimial degree	MSE(training set)	$\|error\|$ (training set)	#parameters	
2	0.0065	4.109	6	
3	0.0065	4.0474	10	
4	0.006	3.8837	15	
5	0.006	3.9125	21	

Table 9: Comparison in approximation errors between neural networks (SCG algorithm) and polynomial regressions for 1 dimensional 7000 training inputs data.

#neurons	MSE(test set) 'perform'	MSE(test set) by hand	$\|error\|$ (test set)	traing time	stopping rule
1	0.0733	0.0732	295.53	1.3004	Validation stop
10	1.15E-04	3.33E-05	6.4632	9.9251	Validation stop
100	0.0087	1.23E-05	3.2173	22.3687	Validation stop
1000	0.8712	6.00E-06	2.1639	7.86E+02	Maximum epoch reached.
#polynimial degree	MSE(training set)	$\|error\|$(training set)	#parameters		
2	0.0751	1.38E+03	3		
3	0.0751	1.38E+03	4		
5	6.03E-02	1.23E+03	6		
9	0.0397	1.17E+03	10		
15	6.26E-04	155.2617	16		

Table 10: Comparison in approximation errors between neural networks (SCG algorithm) and polynomial regressions for 1 dimensional 700 training inputs data.

#neurons	MSE(test set) 'perform'	MSE(test set) by hand	$\|error\|$ (test set)	traing time	stopping rule
1	0.0821	0.0821	31.3603	0.3137	Validation stop
10	4.12E-04	3.23E-04	2.0238	0.4226	Validation stop
100	0.0088	2.94E-05	0.5235	2.0594	Validation stop
1000	1.1521	2.81E-01	60.4646	4.70E+00	Maximum epoch reached.
#polynimial degree	MSE(training set)	$\|error\|$ (training set)	#parameters		
2	0.075	137.1435	3		
3	0.075	137.1348	4		
5	0.0608	124.2172	6		
9	0.0397	118.0247	10		
15	6.13E-04	15.2754	16		

Table 11: Comparison in approximation errors between neural networks (SCG algorithm) and polynomial regressions for 1 dimensional 70 training inputs data.

#neurons	MSE(test) set	$\|error\|$ (test set)	traing time	stopping rule
1	0.0444	2.4812	0.2247	Validation stop
10	0.0073	0.9016	0.2372	Validation stop
100	0.0827	3.3312	0.3127	Validation stop.
1000	5.5064	24.3413	0.501	Validation stop.
#polynimial degree	MSE(training set)	$\|error\|$ (training set)	#parameters	
2	0.0568	11.5097	3	
3	0.0568	11.5099	4	
5	0.0474	10.6863	6	
9	0.0317	10.2436	10	
15	6.27E-04	1.5479	16	

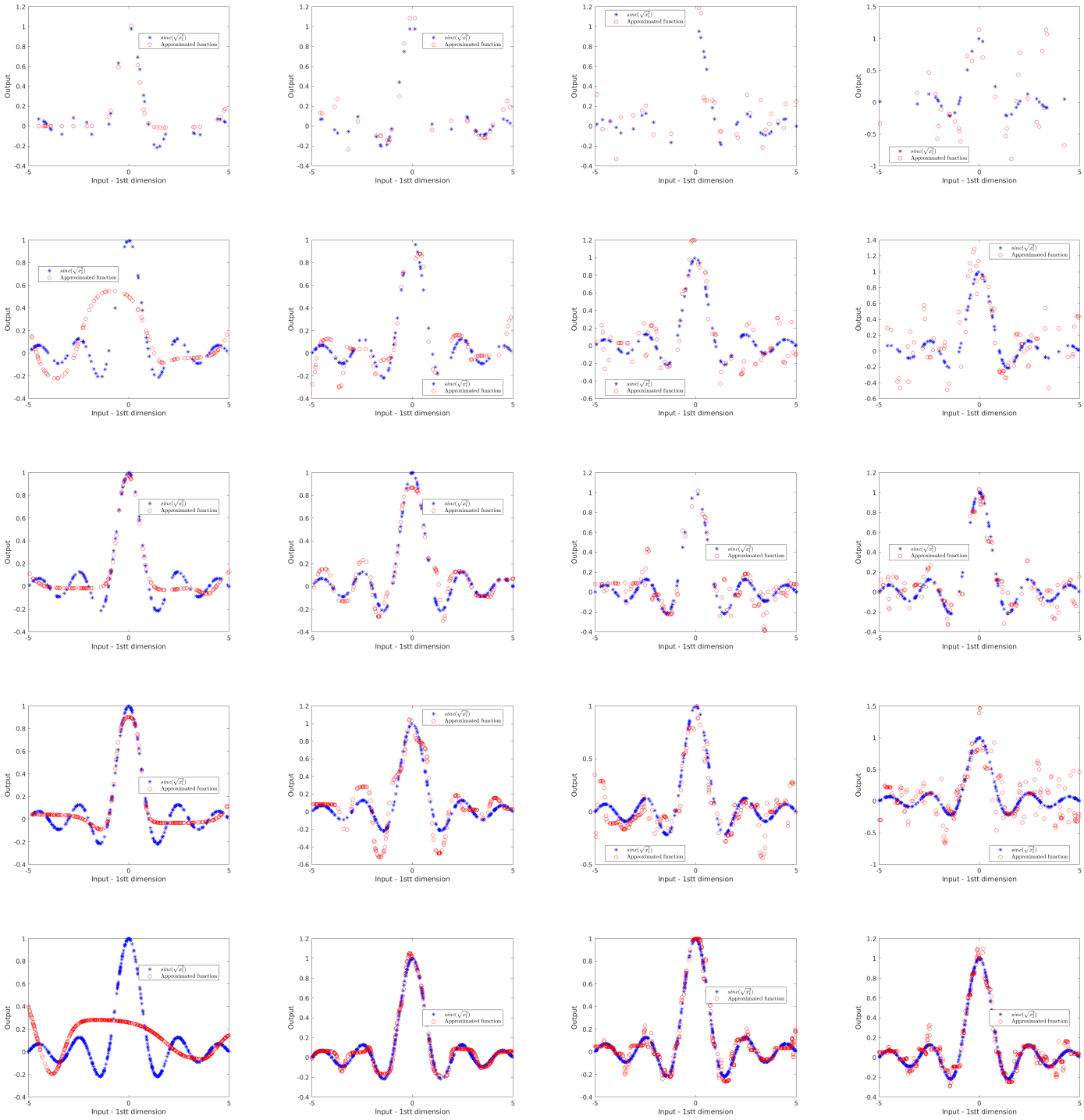


Figure 24: One dimensional approximated function vs true function with different setting: sizes of inputs (210,600,900,1500,3000) from top to bottom and the number of neurons (5,20,60,100) from left to right.

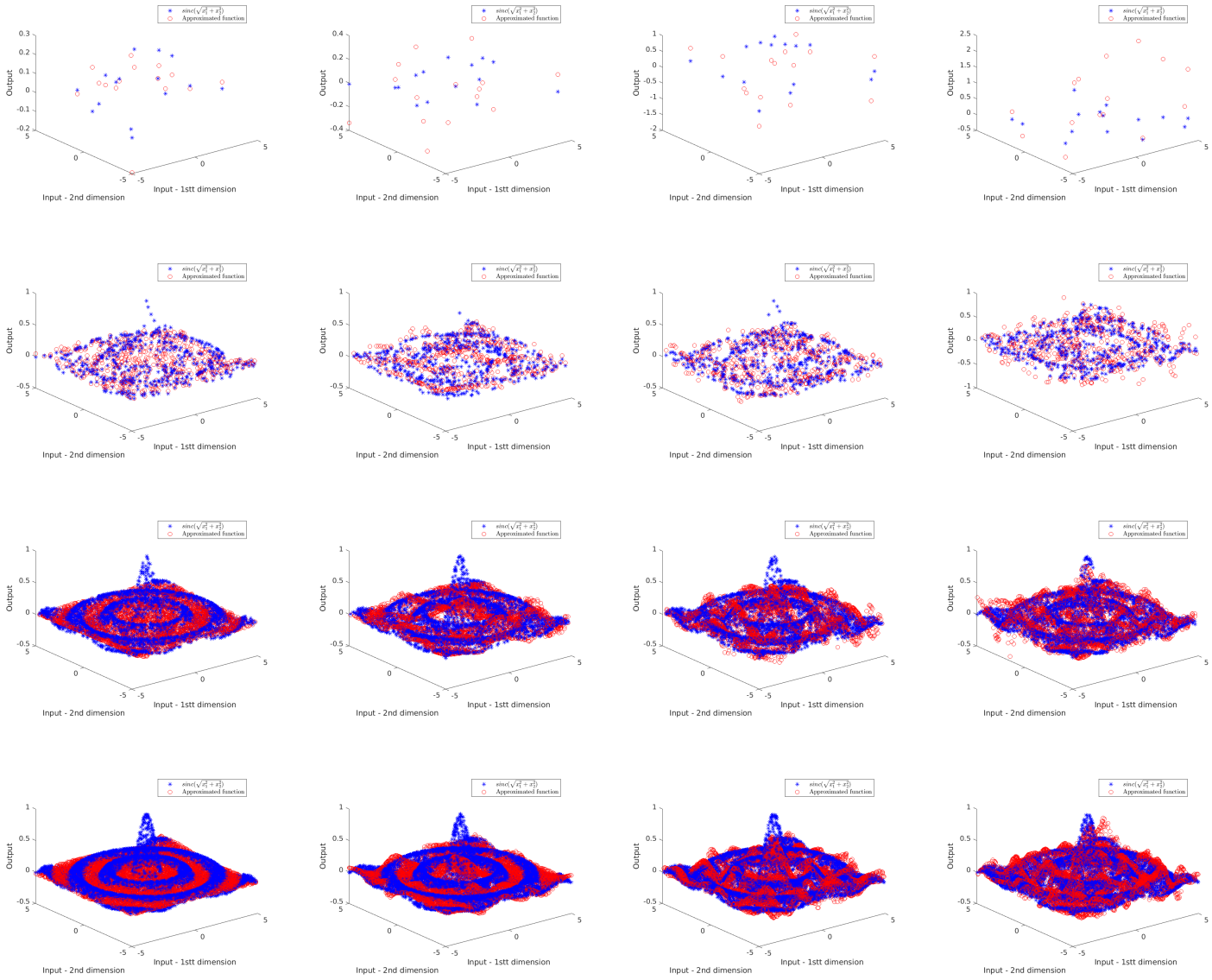


Figure 25: Two dimensional inputs with BP- approximated function vs true function with different setting: sizes of inputs (10,60,150,210) from top to bottom and the number of neurons (5,20,60,100) from left to right.

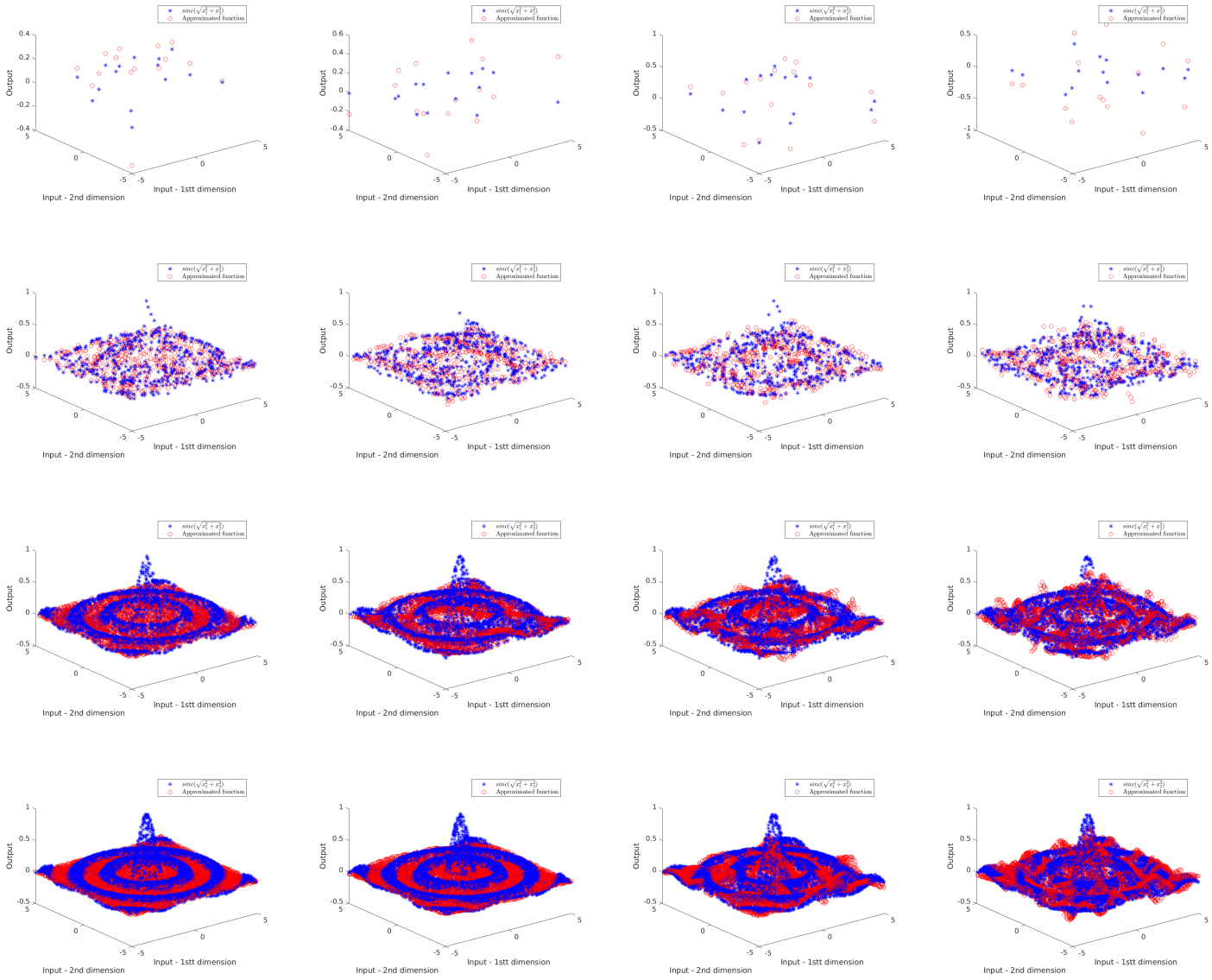


Figure 26: Two dimensional inputs CG - approximated function vs true function with different setting: sizes of inputs (10,60,150,210) from top to bottom and the number of neurons (5,20,60,100) from left to right.

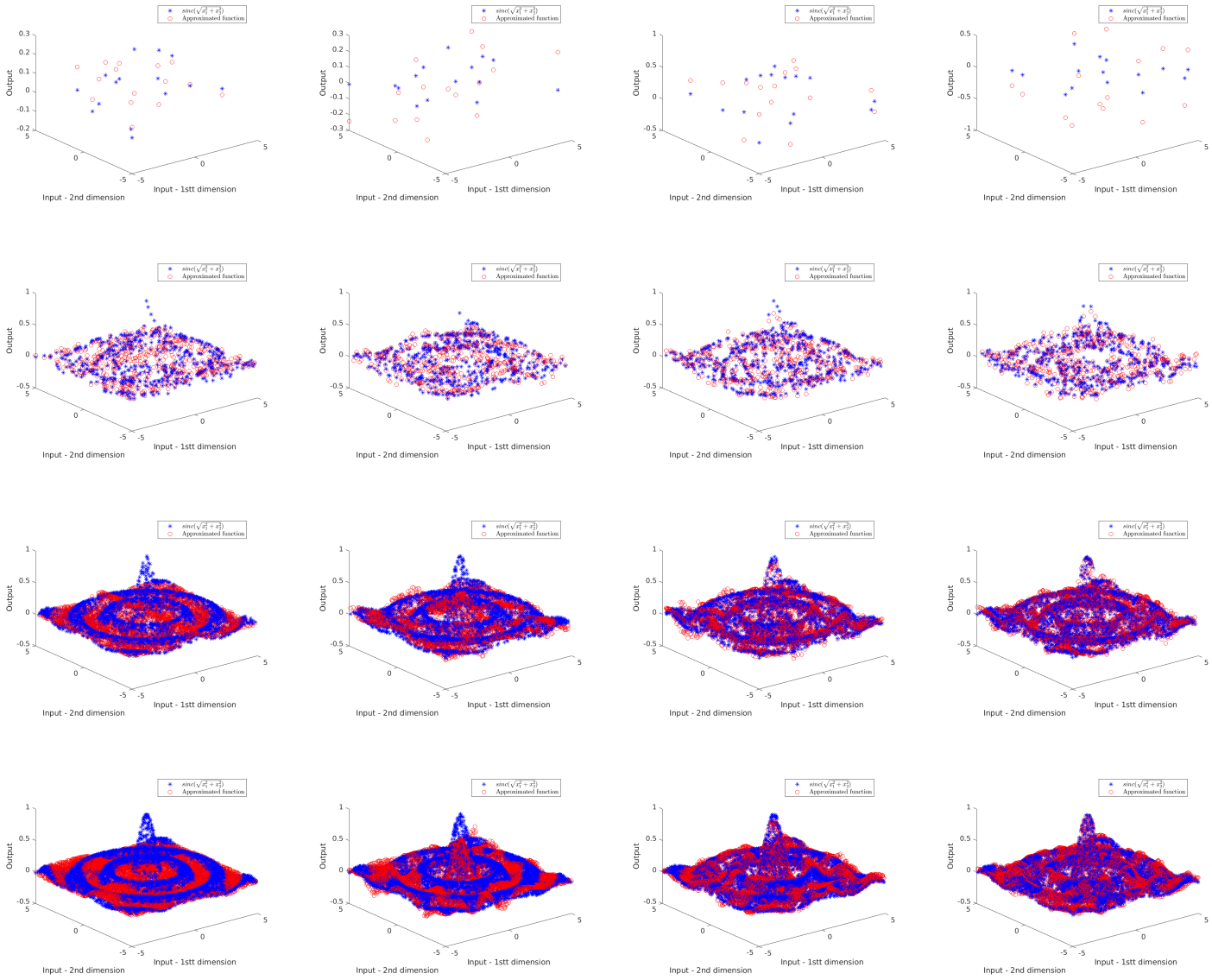


Figure 27: Two dimensional inputs LM - approximated function vs true function with different setting: sizes of inputs (10,60,150,210) from top to bottom and the number of neurons (5,20,60,100) from left to right.

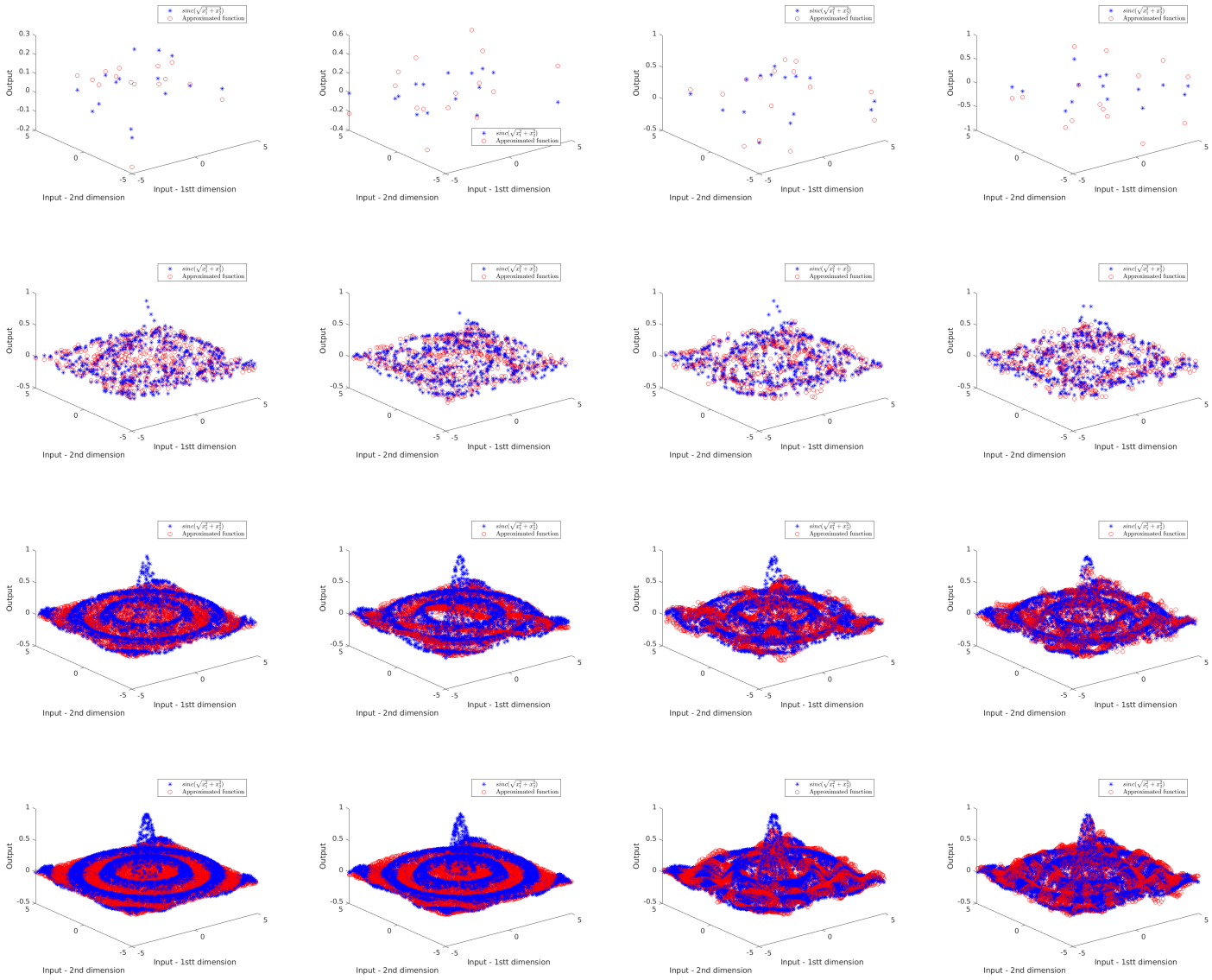


Figure 28: Two dimensional inputs QN - approximated function vs true function with different setting: sizes of inputs (10,60,150,210) from top to bottom and the number of neurons (5,20,60,100) from left to right.

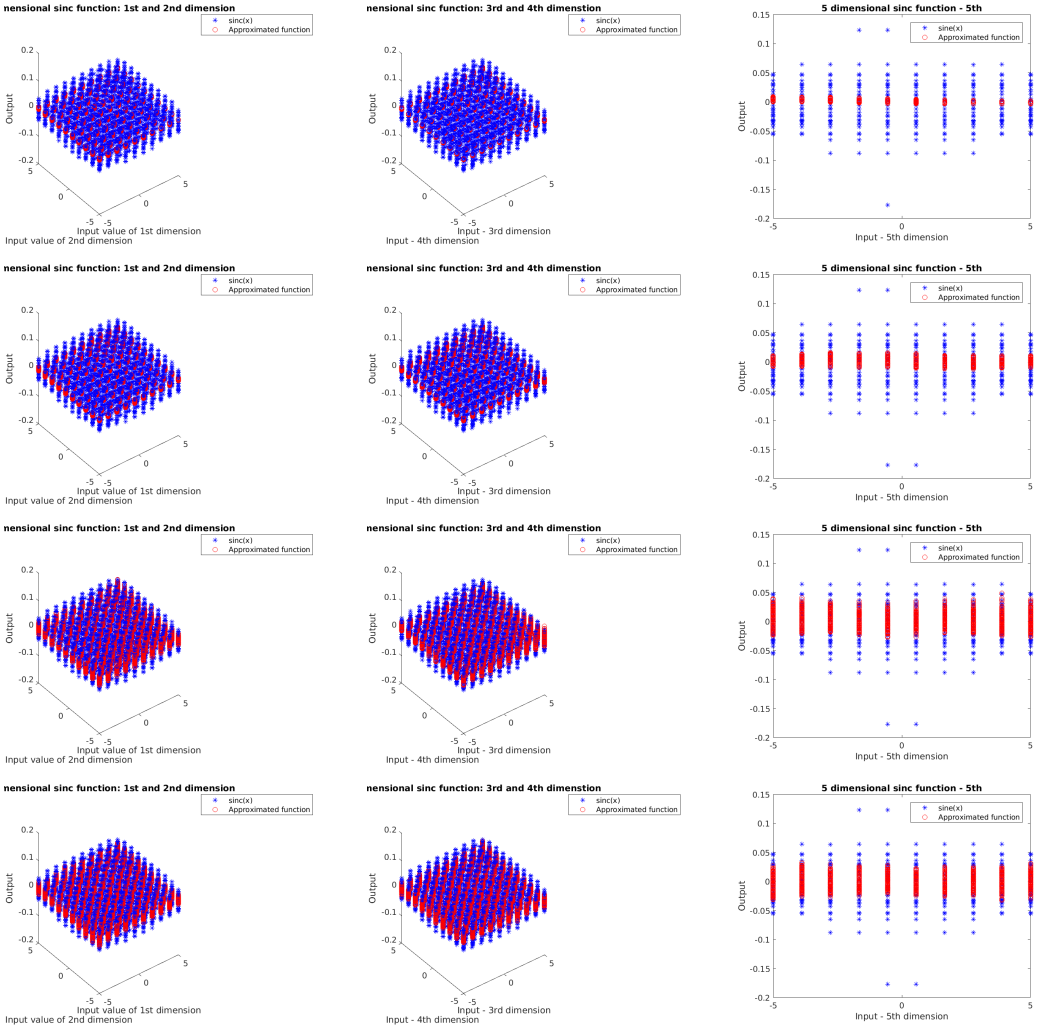


Figure 29: 10 inputs - five dimensional approximated function vs true function with different number of neurons (5,20,60,100) from top to bottom and with different dimensional plot ($1^{st}dimvs2^{nd}dim$,$3^{rd}dimvs4^{th}dim$,$5^{th}dim$) from left to right.

Exercise Session 2

5 Santa Fe laser data - time-series prediction

In this problem, Santa Fe laser data are used, which are time series laser intensity measurements. The first 1000 observations (lasertrain.dat) are employed as training and validation dataset. The next 100 observations (laserpred.dat) are utilized as the test dataset. The plot is shown in the fig.30.

85% of 1000 observations are assigned to the training set while the other 15% are used for the validation set. To enhance the generalization of the model and give a better fit for the test set, a regularization term 10^{-5} and early stopping rule are applied.

At first, models with different number of lag and neurons are investigated with the Levenberg-Marquardt algorithm. The setting $(lag, \#neurons) = (70, 30)$ is found to give the lowest MAE 17.39 and MSE 938 among several slightly different settings (table.12). The fitting result for the test set is shown in the bottom-right in the figure.31. It appears that the trained network somewhat captures the change in the pattern after $t = 60$ although the fitting result for that part is not good.

With $(lag, \#neurons) = (70, 30)$, the study on performances for test sets with different algorithms is followed. The results are again found in the table12, where it is revealed that Levenberg-Marquardt algorithm leads to the minimum MAE and MSE while backpropagation approach yields the worst MAE and MSE. What is more, the fig31 shows that none of the algorithms can perfectly learn the signal pattern both before and after $t = 60$.

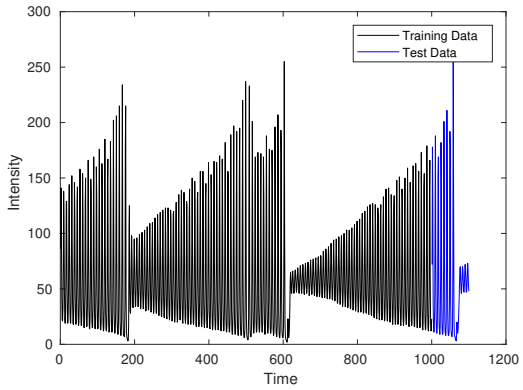


Figure 30: Training set and test set: time vs intensity

Table 12: Performance for test datasets with different setting

algorithm	Lag	#neurons	MAE	MSE
Levenberg-Marquardt	70	10	38.8407	2467
Levenberg-Marquardt	70	30	17.3912	937.6541
Levenberg-Marquardt	70	40	43.2618	3335
Levenberg-Marquardt	80	30	46.2326	4867
Levenberg-Marquardt	100	30	21.8682	1006
Levenberg-Marquardt	200	30	20.7413	1452
Scaled Conjugate Gradient	70	30	110.6055	2584
BFGS Quasi-Newton	70	30	38.3663	2556
Resilient Backpropagation	70	30	53.3695	5149

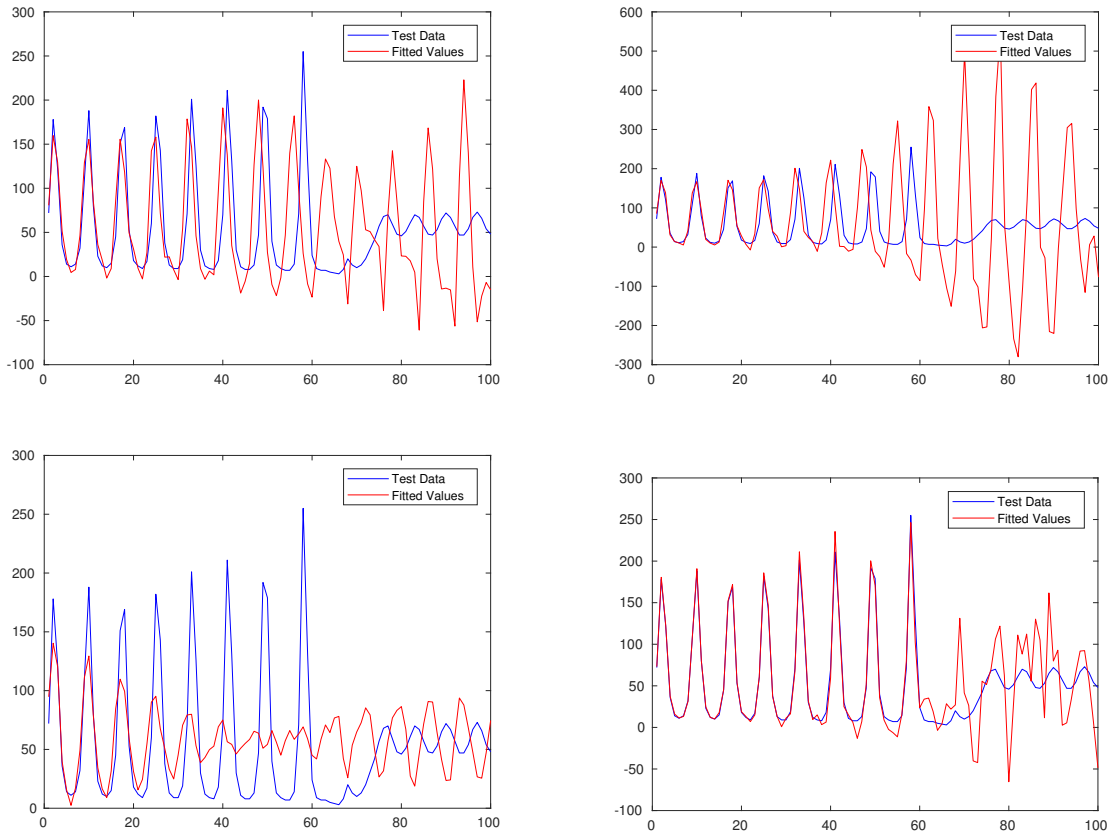


Figure 31: Test results with lag = 70 and number of neurons 30 for different learning algorithms: back-propagation (upper left), conjugate gradient (upper right), quasi-newton (bottom left), and Levenberg-Marquardt (bottom right).

6 Alfabet recognition

In this problem, an matlab demo for character recognition **appcr1** is studied. An example input set is given by the script **prprob** in which 26 alphabet characters are given as 35 (#features) $\times$ 26 (#inputs) matrix X and corresponding labels are given by a 26 $\times$ 26 matrix T. Each column of X is corresponding to a 5×7 bitmap for an alphabet. Each column of T is a standard basis vector in $\mathbb{R}^{26}$; namely, when jth (column) input is the ith letter of the alphabet, only ith row of jth column is 1 and the other 25 elements are 0. An example of the input vector for A is shown in the left image of fig.32.

With these examples of patterns, a feedforward neural network with 25 hidden neurons and 26 neurons in its output layer is trained for this supervised learning problem. As a transfer function in the hidden layer and output layer, the symmetric sigmoid transfer function **tansig** and the linear transfer function **purelin** are respectively employed. To achieve a good generalization of a trained network, early stopping rule with a validation dataset is also used. Furthermore, the training is done with the Levenberg-Marquardt algorithm.

Based on this basic network structure, two neural networks are trained. One is obtained from the non-noise 26 training inputs data and the other is gained from the noisy 780 training inputs data (e.g. right figure of fig.32). Then, the performance of these networks are compared using various test data sets. To be concrete, error rates are calculated using an test dataset that is 30 copies of each letter of the alphabet contaminated by noise (35 $\times$ 780 matrix). Furthermore, the shift of error rates over different magnitudes

of noise is plotted (33). Note that an output matrix whose elements are between -1 and 1 is transformed into an appropriate standard basis vector based matrix of class labels by **compet** function.

As can be seen in the fig.33, unseen noisy data of alphabet images are more correctly classified when a neural network trained with noisy input data is employed than when the network trained with pure data is used.

Figure 32: Left: a bit map of A without noise. Right: a bit map of A with noise

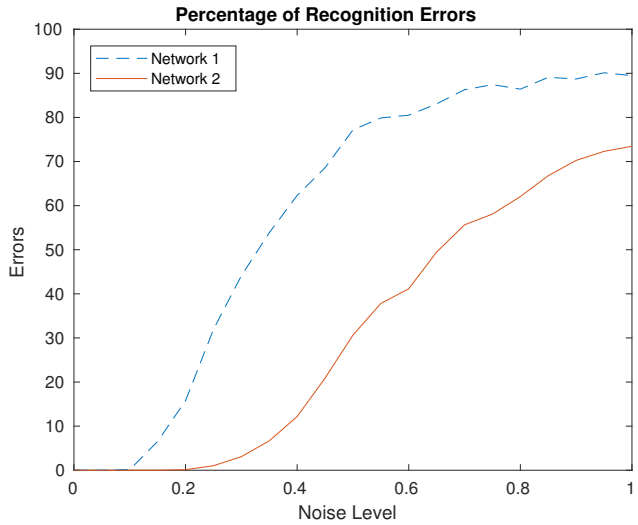


Figure 33: Error rates over different amount of noise. Network1 is trained with pure data while network2 is done with noisy data

7 Breast Cancer Wisconsin - classification problem

In this problem, Breast Cancer Wisconsin (Diagnostic) Data Set **bcw.mat** is studied. The data sets are composed from 569 inputs with their dimension 30 and corresponding 569 binary labels. To perform a classification task on an unseen future sample from the same population, a MLP with one hidden and one output layer is trained with the build-in matlab routine **patternnet**. The default transfer functions in hidden and output layers are utilized: hyperbolic tangent sigmoid transfer function in the hidden layer and log-sigmoid transfer function mapping into a value between 0 and 1 in the output layer.

The data are again divided into training, validation, and test sets with their proportion 0.7, 0.15, 0.15 respectively with the same reason mentioned in previous exercise. A variety of statistics related to a

network's performance are compared, which are attained via different training algorithms, magnitudes of regularization term, stopping rules, number of neurons in hidden layer, initial weights.

To begin with, impact of the different number of neurons in the hidden layer on the area under the Receiver Operating Characteristic (ROC) curve (AUC) is studied for various regularization terms and learning algorithms. One of reasons to select AUC as a performance measure is that simply using misclassification rates results in ignoring the fact that a false negative can be more severe than a false positive. The ROC curve describes the sensitivity (true positive rate) with respect to the (1− specificity) (false positive rate), and AUC reflects the capability fo the classifier. In this case, since output from the network ranges between 0 and 1, the curve is drawn per different threshold value. Moreover, the larger the AUC, the better the classier is, and when there is no curve but a linear line for a classifier, for example fig.36, it means that it has no classification capability [2].

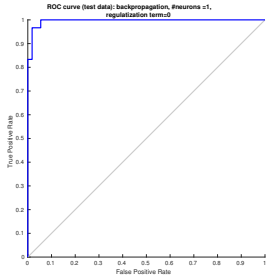
The fig.34, 35, 36, and 37 show the ROC curves for test dataset. It is revealed that quas-newton method does not work well with any given settings of hidden neurons and regularization terms (36). This is probably because ....

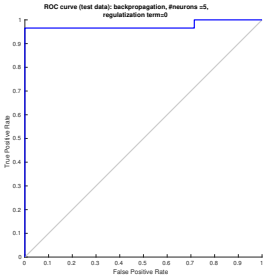
Regarding the backpropagation learning algorithm, although there are some differences in the shape of curves over the number of hidden neurons, it seems to be difficult to find a specific relationship between the number of neurons and shape of ROC curves except for the case that regularization terms = 1. From the bottom fives ROC curves in the fig.34, it can be seen that AUC tends to be large when the number of hidden neurons is small if the #neurons=5 case is excluded, where the training of the network would be done at a local minima or saddle point. This may be due to the fact that the given regularization term 1 is so large that a minimization for norm of the parameter vector is more emphasized than a minimization of error especially when there are more neurons, which results in smaller AUC.

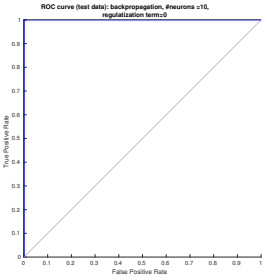
Under the usage of conjugate gradient algorithm, ROC curves with regularization term 1 (5 figures displayed in the bottom of 35) again disclose that all trained networks with different size of hidden neurons excepting for #neurons = 100 have no discriminant power. However, given #neurons = 100, the trained networks's AUC is very high.

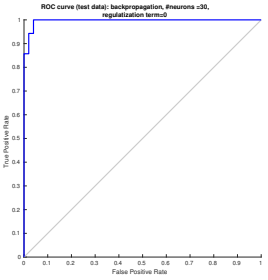
When the Levenberg-Marquardt is employed in a learning procedure, two notable aspects are found. One is seemingly larger AUCs for classifiers trained with the regularization term 1 in comparison with those gained via the other algorithms (five figures in the bottom of fig.37). The other remarkable point is that quite a few ROC curve do look like having AUC=1.

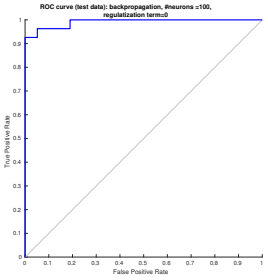
Furthermore, a possible association between AUCs or shape of ROC curves and size of hidden neurons is investigated, but any clear association between them is found. Rather, the effect of the number of neurons seems to be associated to the magnitude of given regularization terms.

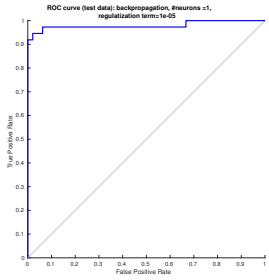

ROC curve (test data): backpropagation, #neurons =1, regularization term=0
True Positive Rate
False Positive Rate

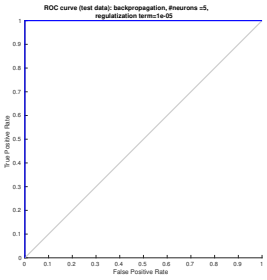

ROC curve (test data): backpropagation, #neurons =5, regularization term=0
True Positive Rate
False Positive Rate

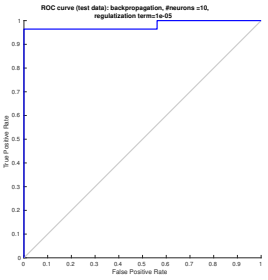

ROC curve (test data): backpropagation, #neurons =10, regularization term=0
True Positive Rate
False Positive Rate

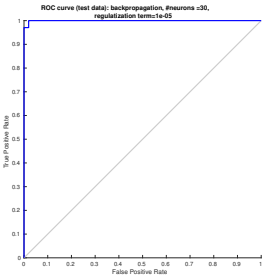

ROC curve (test data): backpropagation, #neurons =30, regularization term=0
True Positive Rate
False Positive Rate

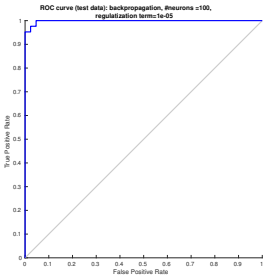

ROC curve (test data): backpropagation, #neurons =100, regularization term=0
True Positive Rate
False Positive Rate

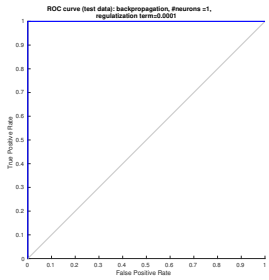

ROC curve (test data): backpropagation, #neurons =1, regularization term=1e-05
True Positive Rate
False Positive Rate

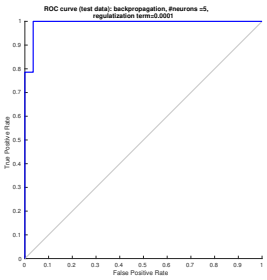

ROC curve (test data): backpropagation, #neurons =5, regularization term=1e-05
True Positive Rate
False Positive Rate

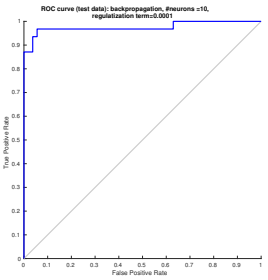

ROC curve (test data): backpropagation, #neurons =10, regularization term=1e-05
True Positive Rate
False Positive Rate

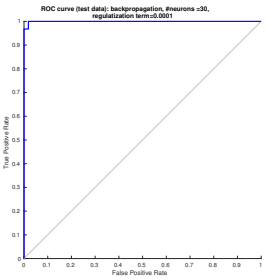

ROC curve (test data): backpropagation, #neurons =30, regularization term=1e-05
True Positive Rate
False Positive Rate

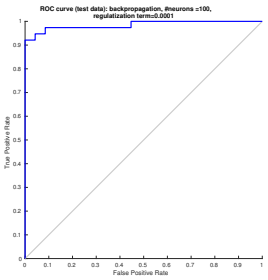

ROC curve (test data): backpropagation, #neurons =100, regularization term=1e-05
True Positive Rate
False Positive Rate

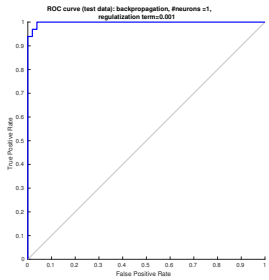

ROC curve (test data): backpropagation, #neurons =1, regularization term=0.0001
True Positive Rate
False Positive Rate

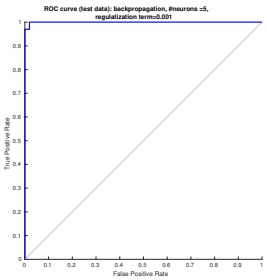

ROC curve (test data): backpropagation, #neurons =5, regularization term=0.0001
True Positive Rate
False Positive Rate

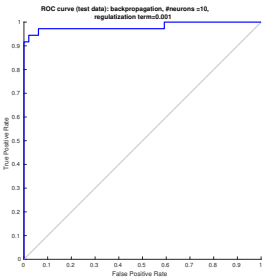

ROC curve (test data): backpropagation, #neurons =10, regularization term=0.0001
True Positive Rate
False Positive Rate

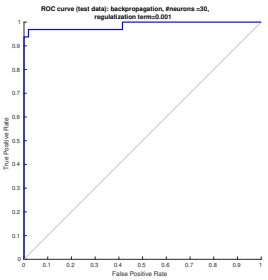

ROC curve (test data): backpropagation, #neurons =30, regularization term=0.0001
True Positive Rate
False Positive Rate

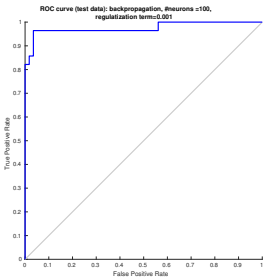

ROC curve (test data): backpropagation, #neurons =100, regularization term=0.0001
True Positive Rate
False Positive Rate

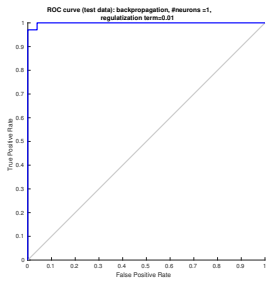

ROC curve (test data): backpropagation, #neurons =1, regularization term=0.001
True Positive Rate
False Positive Rate

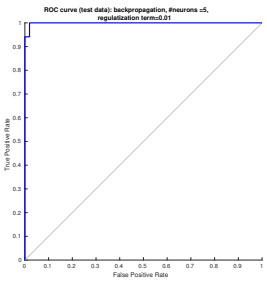

ROC curve (test data): backpropagation, #neurons =5, regularization term=0.001
True Positive Rate
False Positive Rate

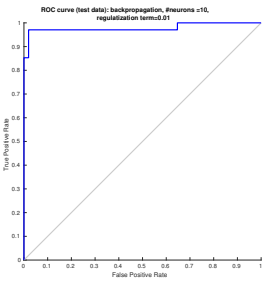

ROC curve (test data): backpropagation, #neurons =10, regularization term=0.001
True Positive Rate
False Positive Rate

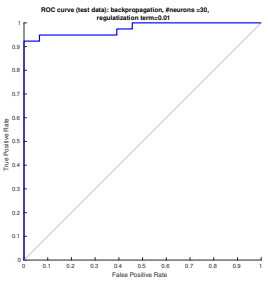

ROC curve (test data): backpropagation, #neurons =30, regularization term=0.001
True Positive Rate
False Positive Rate

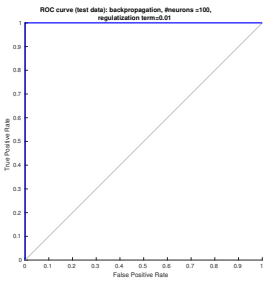

ROC curve (test data): backpropagation, #neurons =100, regularization term=0.001
True Positive Rate
False Positive Rate

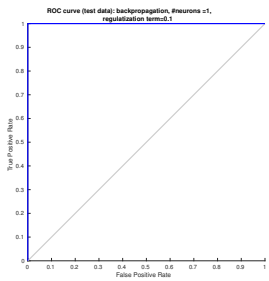

ROC curve (test data): backpropagation, #neurons =1, regularization term=0.01
True Positive Rate
False Positive Rate

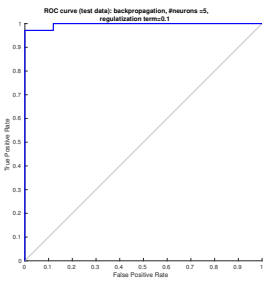

ROC curve (test data): backpropagation, #neurons =5, regularization term=0.01
True Positive Rate
False Positive Rate

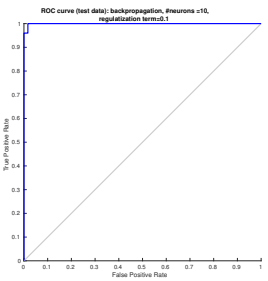

ROC curve (test data): backpropagation, #neurons =10, regularization term=0.01
True Positive Rate
False Positive Rate

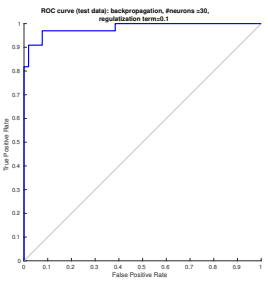

ROC curve (test data): backpropagation, #neurons =30, regularization term=0.01
True Positive Rate
False Positive Rate

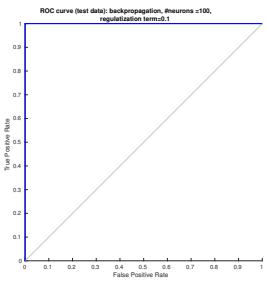

ROC curve (test data): backpropagation, #neurons =100, regularization term=0.01
True Positive Rate
False Positive Rate


ROC curve (test data): backpropagation, #neurons =1, regularization term=0.1
True Positive Rate
False Positive Rate


ROC curve (test data): backpropagation, #neurons =5, regularization term=0.1
True Positive Rate
False Positive Rate


ROC curve (test data): backpropagation, #neurons =10, regularization term=0.1
True Positive Rate
False Positive Rate


ROC curve (test data): backpropagation, #neurons =30, regularization term=0.1
True Positive Rate
False Positive Rate


ROC curve (test data): backpropagation, #neurons =100, regularization term=0.1
True Positive Rate
False Positive Rate

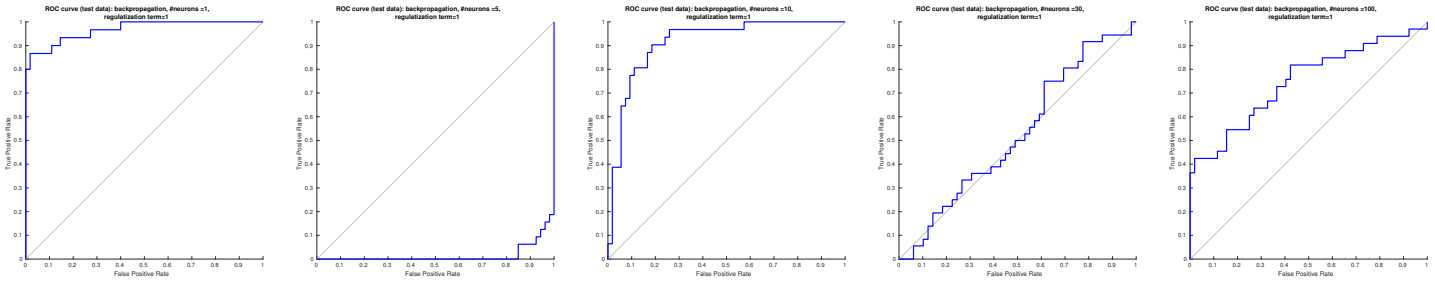
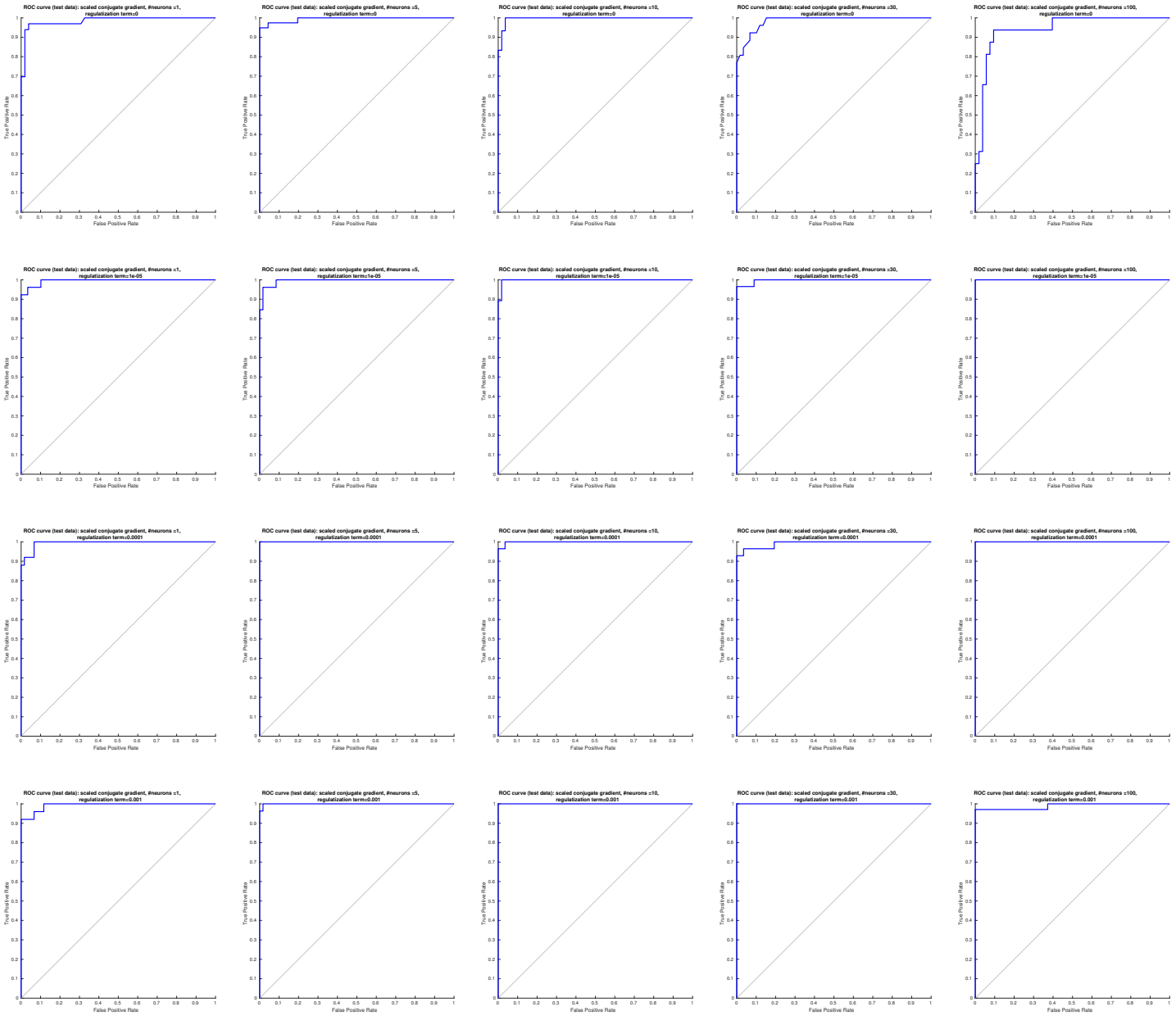


Figure 34: Backpropagation algorithm: the effects of different number of neurons and size of regularization terms on ROC curve. #neuros are (1,5,10,30,100) from left to right and regularization terms are $(0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$from top to bottom.



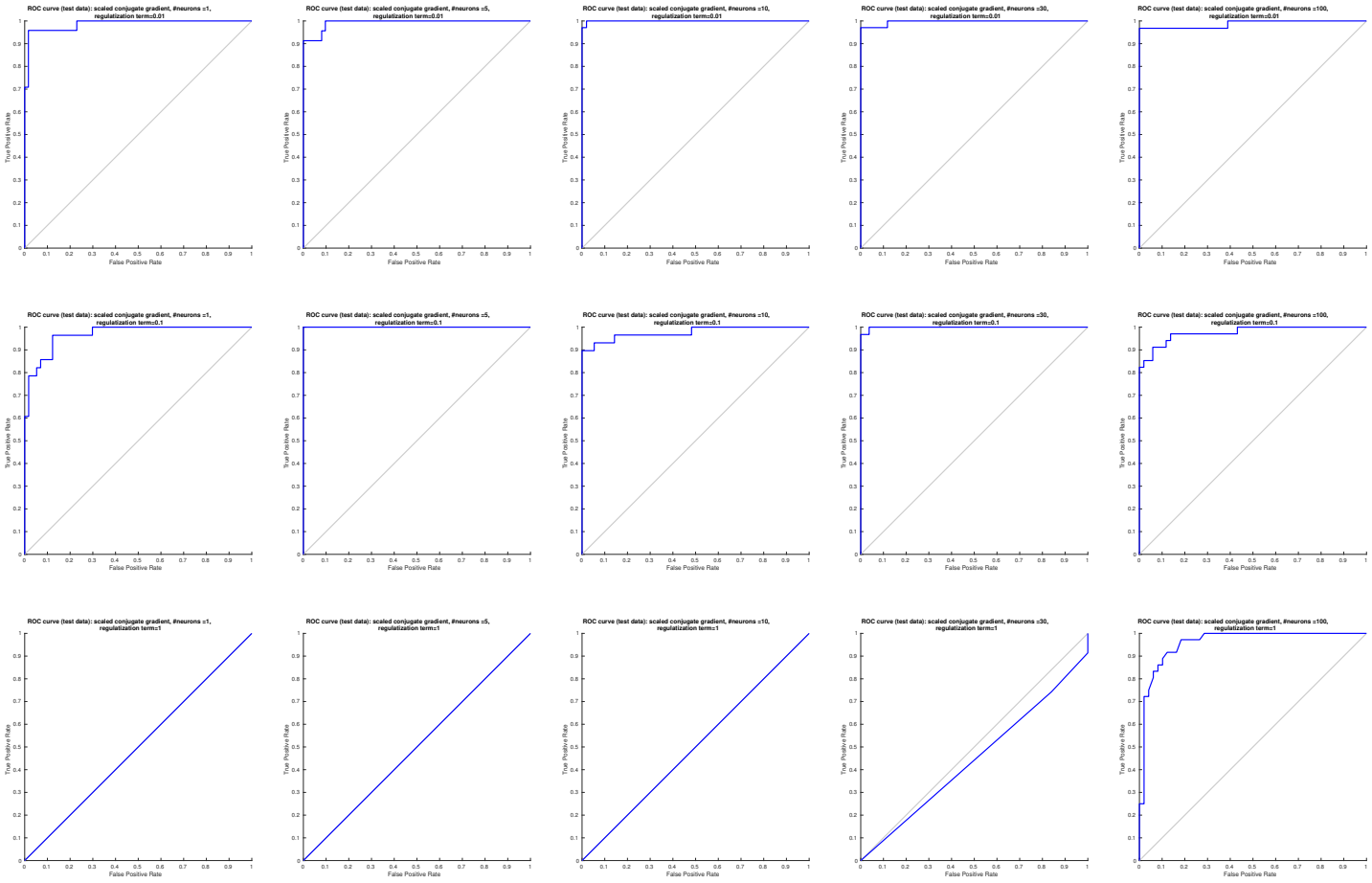


Figure 35: Conjugate gradient algorithm: the effects of different number of neurons and size of regularization terms on ROC curve. #neuros are (1,5,10,30,100) from left to right and regularization terms are $(0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$from top to bottom.

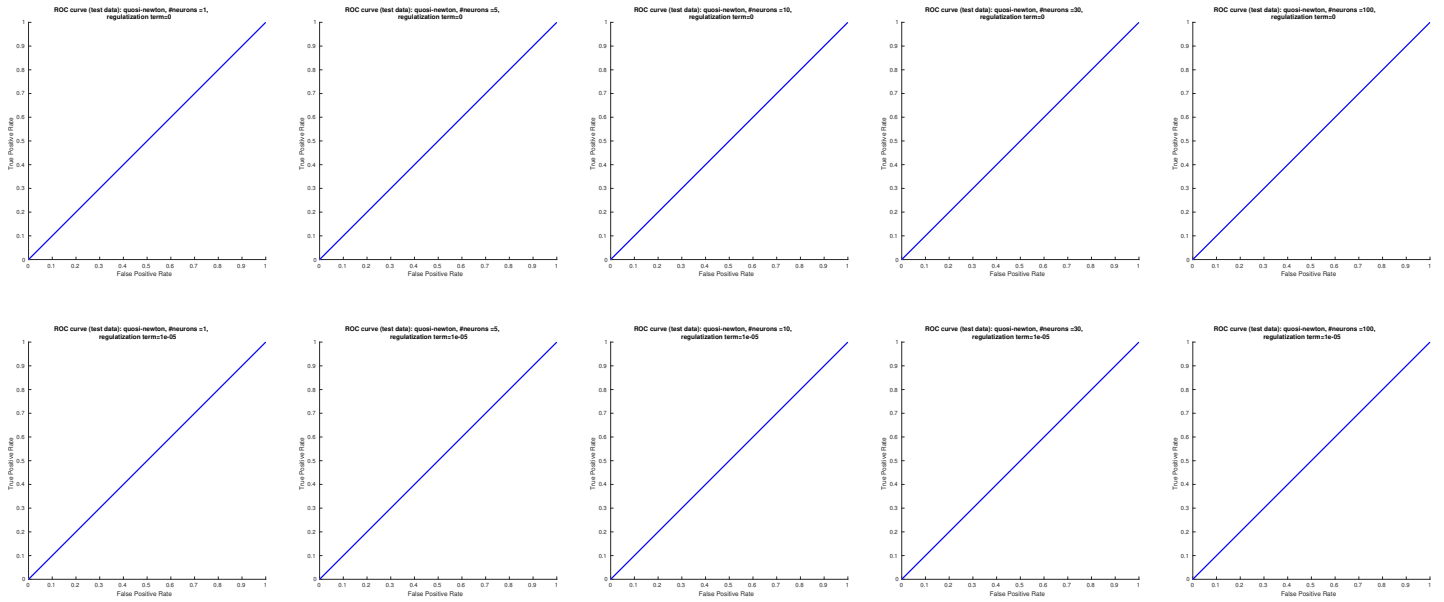
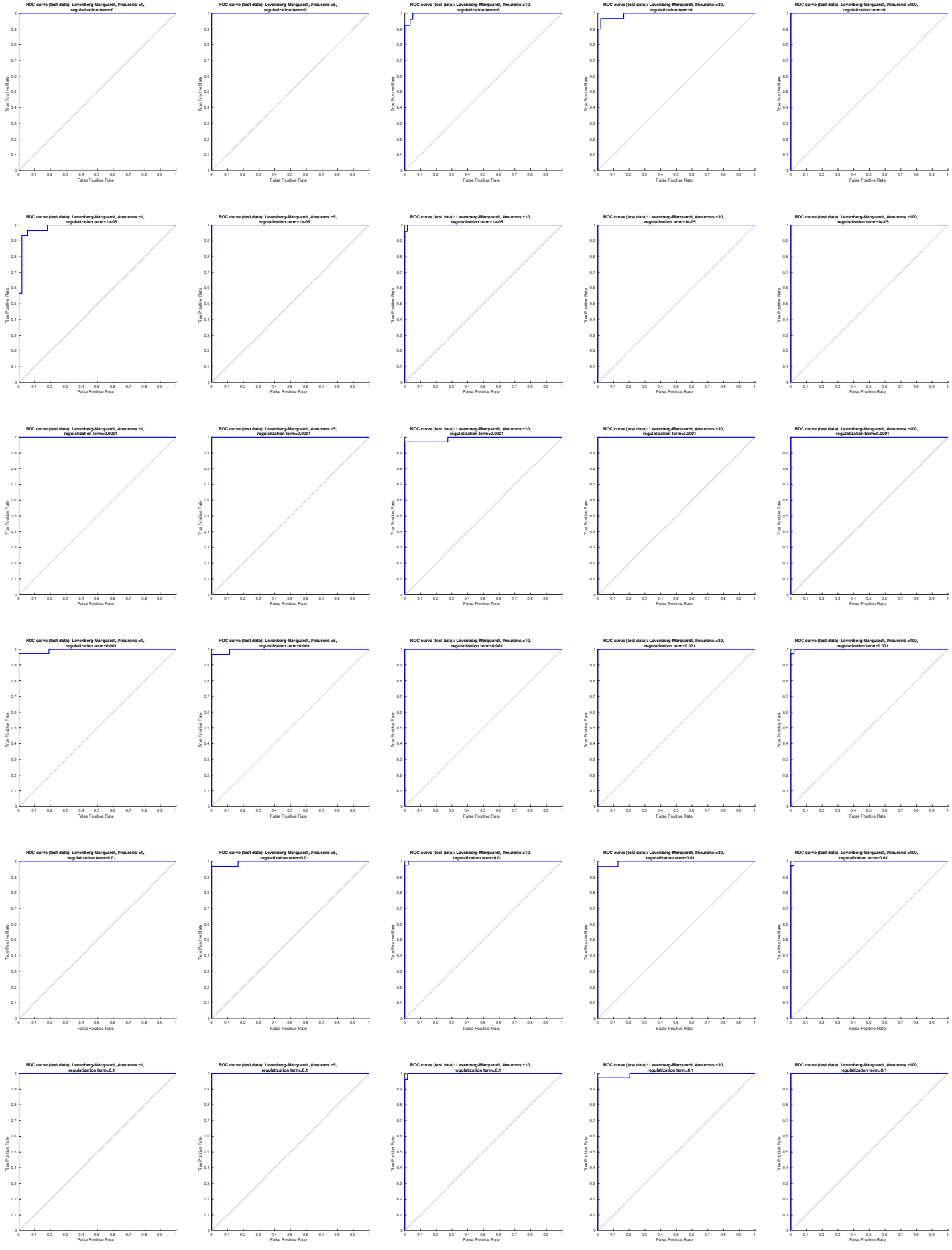


Figure 36: Quasi-newton algorithm: the effects of different number of neurons and size of regularization terms on ROC curve. #neuros are (1,5,10,30,100) from left to right and regularization terms are $(0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$from top to bottom.


ROC curve (test data): Levenberg-Marquardt, #neurons =1, regularization term=0
ROC curve (test data): Levenberg-Marquardt, #neurons =5, regularization term=0
ROC curve (test data): Levenberg-Marquardt, #neurons =10, regularization term=0
ROC curve (test data): Levenberg-Marquardt, #neurons =30, regularization term=0
ROC curve (test data): Levenberg-Marquardt, #neurons =100, regularization term=0
ROC curve (test data): Levenberg-Marquardt, #neurons =1, regularization term=1e-05
ROC curve (test data): Levenberg-Marquardt, #neurons =5, regularization term=1e-05
ROC curve (test data): Levenberg-Marquardt, #neurons =10, regularization term=1e-05
ROC curve (test data): Levenberg-Marquardt, #neurons =30, regularization term=1e-05
ROC curve (test data): Levenberg-Marquardt, #neurons =100, regularization term=1e-05
ROC curve (test data): Levenberg-Marquardt, #neurons =1, regularization term=0.0001
ROC curve (test data): Levenberg-Marquardt, #neurons =5, regularization term=0.0001
ROC curve (test data): Levenberg-Marquardt, #neurons =10, regularization term=0.0001
ROC curve (test data): Levenberg-Marquardt, #neurons =30, regularization term=0.0001
ROC curve (test data): Levenberg-Marquardt, #neurons =100, regularization term=0.0001
ROC curve (test data): Levenberg-Marquardt, #neurons =1, regularization term=0.001
ROC curve (test data): Levenberg-Marquardt, #neurons =5, regularization term=0.001
ROC curve (test data): Levenberg-Marquardt, #neurons =10, regularization term=0.001
ROC curve (test data): Levenberg-Marquardt, #neurons =30, regularization term=0.001
ROC curve (test data): Levenberg-Marquardt, #neurons =100, regularization term=0.001
ROC curve (test data): Levenberg-Marquardt, #neurons =1, regularization term=0.01
ROC curve (test data): Levenberg-Marquardt, #neurons =5, regularization term=0.01
ROC curve (test data): Levenberg-Marquardt, #neurons =10, regularization term=0.01
ROC curve (test data): Levenberg-Marquardt, #neurons =30, regularization term=0.01
ROC curve (test data): Levenberg-Marquardt, #neurons =100, regularization term=0.01
ROC curve (test data): Levenberg-Marquardt, #neurons =1, regularization term=0.1
ROC curve (test data): Levenberg-Marquardt, #neurons =5, regularization term=0.1
ROC curve (test data): Levenberg-Marquardt, #neurons =10, regularization term=0.1
ROC curve (test data): Levenberg-Marquardt, #neurons =30, regularization term=0.1
ROC curve (test data): Levenberg-Marquardt, #neurons =100, regularization term=0.1
True Positive Rate
False Positive Rate

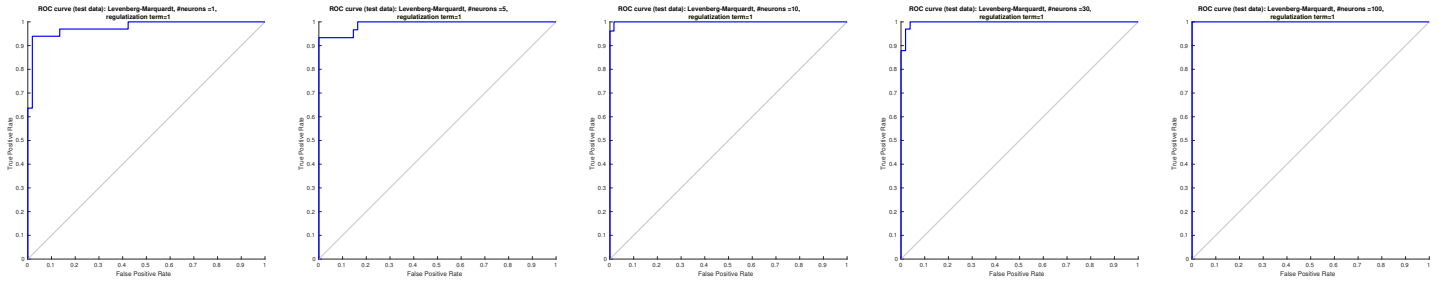


Figure 37: Levenberg-Marquardt algorithm: the effects of different number of neurons and size of regularization terms on ROC curve . #neuros are (1,5,10,30,100) from left to right and regularization terms are $(0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$from top to bottom.

Following the study on ROC plot, averages and medians of various measurements through in total 35 experiments in which combinations of 5 types of #neurons and 7 types of regularization terms are used. Although, these statistical values themselves are not meaningful, they could be still useful to investigate differences among different setting. Here, effects of different stopping criteria, training algorithms, and initial weights are mainly focused.

The sample means and medians for the test set are summarized on the table.14, 13, 15, and 16. Note that each network is trained with four training algorithms and default initial weights: a random assignment, and the tables are organized by how to stop the training scheme.

Within each table, frequency indicates how many times a specific stopping rule works, test error rate gives the percentage of misclassification.

From the table.14 and 13, we can observe that misclassification rates are much smaller for the cases where 'Validation stop' rule are applied on average, regardless the usage of conjugate gradient or backpropagation algorithms. In both cases, the medians of performance measure "tr.best.tperf" are much smaller for the 'Minimum gradient reached' cases. Nevertheless, networks obtained from that stopping rule show the poor generalization for new testset. For example, In SCG cases, misclassification rates 0.5261 on average which is much larger than 0.0987 for the average value in the other stopping rule. Similarly, AUC 0.6982 is also much worse than 0.9910. The same comment is also applicable for the backpropagation case. A possible reason for this phenomena is that both training schemes have trapped by a local minima or saddle point.

Table 13: SCG: stopping rules

SCG	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	7	mean	0.1799	2.1429	0.2002	0.5261	0.6982	0.1799
		median	2.336e-17	2	0.1884	0.5647	0.5	0
'Validation stop'	28	mean	0.2140	8.4643	0.2724	0.0987	0.9910	0.2128
		median	0.0930	1.5	0.2591	0.0353	0.9948	0.0928

Table 14: BP: stopping rules

BP	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
' 'Minimum gradient reached'	4	mean	1.355e-08	25	mg_time	0.5618	0.5505	0
		median	5.952e-09	25.5	0.2323	0.4824	0.6366	0
'Validation stop'	31	mean	0.09683	10.1	0.2382	0.03567	0.9906	0.09176
		median	0.08361	3	0.2357	0.02353	0.996	0.07852

As shown on the table.15, it can be found that early stopping rule does not occur. Moreover, average training time 4.68 is much longer than those for backpropagation and conjugate gradient algorithms, their median values are more or less same. Hence, it can be said that quasi-newton approach can result in a training process with long time, though many training cases would not take such a long time. On the other hand, average misclassification rate 0.4114 and AUC 0.5 are roughly as same degree as those for back-propagation and conjugate gradient algorithm whose training process are stopped by "minimum gradient reached" rule, suggesting that quasi-newton method should not be adopted in this binary classification task.

Table 15: QN: stopping rule

QN	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	35	mean	0.5542	0.6286	4.68	0.4114	0.5	0.5541
		median	0.6453	0	0.2681	0.3765	0.5	0.6453
'Validation stop'	0							

With the Levenberg-Marquardt algorithm, two additional stopping rules 'Performance goal met' and 'Maximum μ reached' are introduced. It is worthwhile to point out that almost all average and median values of misclassification rate and AUC for any of four different stopping cases are better than any other those for classifiers trained by the other three algorithms. Hence, this approach can be regarded as a stable method. However, it is also important to note that the average and median training time in the case that 'Validation stop' rule is applied, $5.089, 1.233$ (16), are much longer than those for the other algorithms. Considering that this stopping rule frequently occurs (16 out of 35), the time required to complete training in LM approach may become the longest among four learning algoritms.

Table 16: LM: stopping rule

LM	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	3	mean	0.02122	12.67	0.2226	0.02353	0.989	0.02122
		median	0.02353	12	0.2168	0.02353	0.9897	0.02353
'Validation stop'	16	mean	0.01352	2.125	5.089	0.01691	0.9986	0.01357
		median	0.01299	1	1.233	0.01765	1	0.01326
Performance goal met	3	mean	-0.06979	0	0.09103	0.007843	0.9998	0.545
		median	0.545	0	0.07555	0	1	0.1449
'Maximum MU reached	13	mean	0.03439	0.4615	2.161	0.01176	0.9972	0.1719
		median	0.01268	0	0.2708	0.01176	0.9994	0.01991

The table17, 18, 19, and 20 provides the summary statistics for different training algorithms , given that all initial interconnection weights are set to 1. Although there are some minor changes with respect to frequency of each stopping rule or others, there does not seem to be any remarkable changes in the results compared with table.14, 13, 15, and 16. It is important, however, that the amount of simulation is not enough to make a solid statement on the effect of initial weight on classifiers because only two different initial weight systems, random assignment and fixed assignment at 1, are studied.

Table 17: BP (all initial weights = 1): stopping rule

BP (all initial weights = 1)	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	5	mean	1.11e-08	26.2	0.2412	0.3788	0.5377	0
		median	4.651e-09	26	0.2537	0.3412	0.6429	0
'Validation stop'	30	mean	0.1175	6.5	0.2542	0.04627	0.9908	0.1105
		median	0.1104	0	0.253	0.04706	0.9954	0.1016

Table 18: SCG (all initial weights = 1): stopping rule

SCG (all initial weights = 1)	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	10	mean	0.3217	1.9	0.2305	0.4941	0.6976	0.3217
		median	0.3007	2	0.2126	0.4235	0.7298	0.3007
'Validation stop'	25	mean	0.1558	9.84	0.2903	0.06682	0.9915	0.1487
		median	0.08214	1	0.2806	0.02353	0.9962	0.06909

Table 19: QN (all initial weights = 1): stopping rule

QN (all initial weights = 1)	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	35	mean	0.5558	0.8571	3.957	0.4097	0.5	0.5558
		median	0.6465	1	0.3052	0.3765	0.5	0.6465
'Validation stop'	0							

Table 20: LM (all initial weights = 1): stopping rule

LM (all initial weights = 1)	frequency	statistics	tr.best.tperf	tr.best_epoch	tr.time	test error rate	test AUC	testper
'Minimum gradient reached'	0							
'Validation stop'	21	mean	0.01462	2.857	5.146	0.01737	0.9951	0.01466
		median	0.0131	1	0.4178	0.01176	0.9989	0.01311
Performance goal met	2	mean	-0.04995	0	0.1483	0.01176	0.9989	0.2035
		median	-0.04995	0	0.1483	0.01176	0.9989	0.2035
'Maximum MU reached	12	mean	0.01836	0.5	1.967	0.02059	0.987	0.0278
		median	0.01892	0	0.2902	0.02353	0.9924	0.02599

Excercise Session 3

8 Dimensionality reduction by PCA analysis

In this problem, a relatively high dimensional input matrix, 21×264 (#features×#inputs), accompanied by corresponding 3×264 output matrix is considered. In such a high dimensional case, it is common to conduct a dimensionality reduction. Principal component analysis would be one of the most popular methods for it. Hereafter, the linear PCA is briefly explained mainly referring to [1, 2].

The objection of the PCA is to find an optimal projection into certain low dimensional subspace of the input data points such that the error is minimized. Given a set of centered or normalized data points $\boldsymbol{x_i} \in \mathbb{R}^d$ and projection P with certain fixed rank k (dimension of the projection), this can be regard as an optimization problem as following:

$$\min_{P \in \mathbb{R}^{d\times d},\ \mathrm{rank}(P)=k} \frac{1}{n} \sum_{i \in [n]} ||\boldsymbol{x}_i - P\boldsymbol{x}_i||^2. \tag{8.1}$$

Although this problem is a nonconvex problem with the constrain of the rank of P, there is an efficient way of solving this problem by using the eigenvector decomposition approach. Concretely, the above problem can be reformulated in the following way with the Pythagorean theorem: $\sum_i ||x_i - Px_i||^2 = \sum_i ||x_i||^2 - \sum_i ||Px_i||^2$. Therefore, solution for eq.8.1 is equivalent the one for maximization problem

$$\begin{aligned} &\max_{P \in \mathbb{R}^{d\times d},\ \mathrm{rank}(P)=k} \frac{1}{n} \sum_i ||Px_i||^2 = \max_{P \in \mathbb{R}^{d\times d},\ \mathrm{rank}(P)=k} \frac{1}{n} \sum_i Tr[Px_i x_i^T P^T] \\ &= \max_{P \in \mathbb{R}^{d\times d},\ \mathrm{rank}(P)=k} Tr[P(\frac{1}{n} \sum_i x_i x_i^T) P^T] = \max_{P \in \mathbb{R}^{d\times d},\ \mathrm{rank}(P)=k} Tr[PSP^T], \end{aligned} \tag{8.2}$$

given a $S = cov(X)$ where X is whole data points matrix. Here, S is eigenvalue-decomposable because it is a square matrix. Since $S \in \mathbb{R}^{d\times d}$ is a symmetric matrix, it has d eigen values, and its decomposition is given as $S = U\Lambda U^T = \sum_{i\in[d]} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T$ with an orthogonal matrix U of a set of eigenvectors $(\boldsymbol{u}_i)$ and a diagonal matrix Λ with eigenvalues (λ_i). Assuming that S has eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$ and corresponding eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_d$, the Rayleigh Quotient shows that the maximum/minimum quadratic forms of S and corresponding solutions are given as

$$\max_{||z||=1} \boldsymbol{z}_{max}^T S \boldsymbol{z}_{max} = \lambda_1, \quad \min_{||z||=1} \boldsymbol{z}_{min}^T S \boldsymbol{z}_{min} = \lambda_d, \quad , \tag{8.3}$$

with the optimal vectors $\boldsymbol{z}_{max} = \boldsymbol{u_1}$ and $\boldsymbol{z}_{min} = \boldsymbol{u_d}$. In other words, the eigenvector corresponding to the largest eigenvalue of S yields the maximum quadratic form. While the Rayleigh Quotient result is obtained by looking for just one dimensional subspace because $\boldsymbol{z}_{max/min}$ is just a vector, the solution matrix P for $\max_{P\in\mathbb{R}^{d\times d}} Tr[PSP^T]$ is attained by looking at a k dimensional subspace because of the a fixed rank k constrain on P. Extending the Rayleigh Quotient result, a optimal cost and projection P_{opt} are found as

$$\max_{\mathrm{rank}(P)=k,\ P:P^2=I} Tr[PSP^T] = \lambda_1 + \lambda_2 + \cdots + \lambda_k, \quad P_{opt} = U_{(k)} U_{(k)}^T, \tag{8.4}$$

where $U_{(k)}$ means the $d \times k$ matrix of top-k eigenvectors. Based on the fact that $\boldsymbol{u}_i$ is orthogonal to one another, P spans the subspace $[\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_k]$, suggesting that the optimal projection results are limited

in the k dimensional subspace spanned by top k eigen vectors, which means that the dimension decreases from d to k. The choice of k can be done with cross-validation. This is the basic flow of PCA. Although the linear PCA works well in a set of data in which there is a linear association among input variables, it does not perform well in a situation where there is a nonlinear relationship among them. In such case, nonlinear PCA can be suitable. This method is different from the linear PCA in the sense that mapping inputs variables into certain lower dimensional space is carried out with not linear function but nonlinear one. This nonlinear mapping can be parametrized multilayer perceptrons [2].

In the given problem, $d = 21, n = 264$, and its covariance matrix is shown in the table.21, from which a strong linear association among inputs variables is found as almost all elements are larger than 0.9. This allows us to expect that linear PCA shoud work well to achieve dimensionality reduction.

Table 21: Correlation matrix

variable	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	1.0000	0.9996	0.9987	0.9971	0.9905	0.9850	0.9812	0.9790	0.9805	0.9766	0.9664	0.9490	0.9270	0.9067	0.8961	0.8910	0.8853	0.8813	0.8787	0.8772	0.8646
2	0.9996	1.0000	0.9997	0.9984	0.9913	0.9857	0.9817	0.9796	0.9821	0.9792	0.9695	0.9518	0.9291	0.9081	0.8971	0.8918	0.8862	0.8822	0.8796	0.8783	0.8665
3	0.9987	0.9997	1.0000	0.9994	0.9918	0.9858	0.9813	0.9792	0.9824	0.9803	0.9707	0.9521	0.9281	0.9058	0.8940	0.8884	0.8826	0.8786	0.8760	0.8748	0.8637
4	0.9971	0.9984	0.9994	1.0000	0.9932	0.9870	0.9819	0.9794	0.9832	0.9820	0.9728	0.9538	0.9286	0.9049	0.8922	0.8860	0.8800	0.8758	0.8731	0.8723	0.8623
5	0.9905	0.9913	0.9918	0.9932	1.0000	0.9986	0.9957	0.9938	0.9944	0.9911	0.9841	0.9726	0.9561	0.9386	0.9284	0.9231	0.9178	0.9141	0.9118	0.9111	0.9016
6	0.9850	0.9857	0.9858	0.9870	0.9986	1.0000	0.9991	0.9980	0.9974	0.9930	0.9872	0.9794	0.9671	0.9529	0.9442	0.9396	0.9347	0.9313	0.9292	0.9284	0.9184
7	0.9812	0.9817	0.9813	0.9819	0.9957	0.9991	1.0000	0.9997	0.9978	0.9921	0.9866	0.9808	0.9713	0.9596	0.9522	0.9482	0.9436	0.9403	0.9384	0.9374	0.9267
8	0.9790	0.9796	0.9792	0.9794	0.9938	0.9980	0.9997	1.0000	0.9983	0.9926	0.9875	0.9828	0.9745	0.9638	0.9570	0.9532	0.9490	0.9458	0.9440	0.9431	0.9328
9	0.9805	0.9821	0.9824	0.9832	0.9944	0.9974	0.9978	0.9983	1.0000	0.9979	0.9942	0.9883	0.9773	0.9636	0.9551	0.9506	0.9463	0.9432	0.9413	0.9413	0.9337
10	0.9766	0.9792	0.9803	0.9820	0.9911	0.9930	0.9921	0.9926	0.9979	1.0000	0.9986	0.9923	0.9786	0.9616	0.9510	0.9456	0.9413	0.9382	0.9364	0.9372	0.9332
11	0.9664	0.9695	0.9707	0.9728	0.9841	0.9872	0.9866	0.9875	0.9942	0.9986	1.0000	0.9966	0.9846	0.9681	0.9570	0.9514	0.9474	0.9446	0.9428	0.9443	0.9426
12	0.9490	0.9518	0.9521	0.9538	0.9726	0.9794	0.9808	0.9828	0.9883	0.9923	0.9966	1.0000	0.9953	0.9841	0.9752	0.9705	0.9673	0.9651	0.9638	0.9654	0.9644
13	0.9270	0.9291	0.9281	0.9286	0.9561	0.9671	0.9713	0.9745	0.9773	0.9786	0.9846	0.9953	1.0000	0.9965	0.9916	0.9885	0.9865	0.9850	0.9840	0.9852	0.9825
14	0.9067	0.9081	0.9058	0.9049	0.9386	0.9529	0.9596	0.9638	0.9636	0.9616	0.9681	0.9841	0.9965	1.0000	0.9988	0.9973	0.9962	0.9952	0.9947	0.9951	0.9901
15	0.8961	0.8971	0.8940	0.8922	0.9284	0.9442	0.9522	0.9570	0.9551	0.9510	0.9570	0.9752	0.9916	0.9988	1.0000	0.9996	0.9992	0.9985	0.9981	0.9980	0.9907
16	0.8910	0.8918	0.8884	0.8860	0.9231	0.9396	0.9482	0.9532	0.9506	0.9456	0.9514	0.9705	0.9885	0.9973	0.9996	1.0000	0.9998	0.9994	0.9991	0.9985	0.9897
17	0.8853	0.8862	0.8826	0.8800	0.9178	0.9347	0.9436	0.9490	0.9463	0.9413	0.9474	0.9673	0.9865	0.9962	0.9992	0.9998	1.0000	0.9998	0.9997	0.9991	0.9902
18	0.8813	0.8822	0.8786	0.8758	0.9141	0.9313	0.9403	0.9458	0.9432	0.9382	0.9446	0.9651	0.9850	0.9952	0.9985	0.9994	0.9998	1.0000	0.9999	0.9993	0.9905
19	0.8787	0.8796	0.8760	0.8731	0.9118	0.9292	0.9384	0.9440	0.9413	0.9364	0.9428	0.9638	0.9840	0.9947	0.9981	0.9991	0.9997	0.9999	1.0000	0.9996	0.9913
20	0.8772	0.8783	0.8748	0.8723	0.9111	0.9284	0.9374	0.9431	0.9413	0.9372	0.9443	0.9654	0.9852	0.9951	0.9980	0.9985	0.9991	0.9993	0.9996	1.0000	0.9947
21	0.8646	0.8665	0.8637	0.8623	0.9016	0.9184	0.9267	0.9328	0.9337	0.9332	0.9426	0.9644	0.9825	0.9901	0.9907	0.9897	0.9902	0.9905	0.9913	0.9947	1.0000

To perform PCA, the input vectors are firstly normalized by **mapstd** function, which prevent a variable with huge variance from dominating a large part of whole variance of the input data. As a next step, the linear PCA is performed by the command **[pp, pca˙p] = processpca(pn, 'maxfrac', 0.001)** in matlab with maximum fraction of variance for removed rows 0.001; namely less than 0.1%.

The outputs from **processpca** are mainly two: **pp** stores $k \times n$ input matrix mapped into a lower dimensional subspace and **pca˙p.transform** stores the $k \times d$ transformation matrix (a set of top-k eigenvectors $\boldsymbol{U}_k^T$). Hence, applying this matrix on the normalized input data produces the same matrix as **pp**.

Table 22: Diagonal matrix Λ_4 with top-4 eigenvalues

Λ	PC1	PC2	PC3	PC4
PC1	20.1537	-0.0000	-0.0000	0.0000
PC2	-0.0000	0.7544	-0.0000	0.0000
PC3	-0.0000	-0.0000	0.0512	0.0000
PC4	0.0000	0.0000	0.0000	0.0227

Table 23: Correlation matrix after PCA

Cor	PC1	PC2	PC3	PC4
PC1	1.0000	-0.0000	-0.0000	0.0000
PC2	-0.0000	1.0000	-0.0000	0.0000
PC3	-0.0000	-0.0000	1.0000	0.0000
PC4	0.0000	0.0000	0.0000	1.0000

In this set of dataset, dimensionality is decreased from 21 to 4, meaning $k = 4$. In fact, the diagonal matrix with top-4 eigenvalues Λ_4 obtained by the command **pca˙p.transform*cormat*pca˙p.transform'** is shown in the table22. The trace of this matrix is 20.9821, indicating the ratio of sum of top-4 eigenvalues on sum of full eigenvalues $\frac{20.9821}{21} = 0.9991$ due to the fact that the input data is normalized. In other word, 99.91 % of variance of original inputs can be described with only 4 dimension. Furthermore, the

command **pca˙p.transform *pca˙p.transform'** results in I_4, assuring that the transformation matrix is a semi-orthogonal matrix. Moreover, the optimal projection matrix P_{opt} given in eq.8.4 can be obtained by the command **pca˙p.transform'*pca˙p.transform**. Indeed, calculation of $Tr[PSP_T]$ also results in 20.9821. In addition, the correlation matrix after PCA is shown in the tab.23, from which no correlations among new axis of transformed data are observed. This matches the orthogonality of eigenvectors.

8.1 Effect of dimensionality reduction on Levenberg-Marquardt and Bayesian Reguralization algorithm

To investigate how the dimensionality reduction brings an influence on the neural network training and performance, four networks are trained using Levenberg-Marquardt and Bayesian Reguralization algorithms, with the original inputs in 21 dimensional space and input projected on 4 dimensional subspace by PCA respectively. The dataset is divided into training, validation, test sets with their ratios 50%, 25%, 25%. Each network consists of one hidden layer with Tan-Sigmoid transfer function and one output layer with pure-linear transfer function. A few measures on performance of each network are shown in the tab.24

Table 24: Comparison in performance on test set. Levenberg-Marquardt vs Bayesian Reguralization and PCA vs non-PCA

		Levenberg-Marquardt	Bayesian Reguralization
Inputs (original)	MSE (test)	0.2791	0.2589
	Training time	0.3600	2.2062
Inputs (after PCA)	MSE (test)	0.3718	0.2704
	Training time	0.2516	0.5382

As expected, MSEs on test set for original data for the networks trained with the original inputs are smaller than those trained with the transformed inputs by PCA for both training algorithms. In particular, the difference in MSEs for Levenberg-Marquardt method, roughly 25% decrease by using original inputs, is larger than that for Bayesian Reguralization with around 5% decrease.

On the other hand, the amount of time required to complete training is shorter in the case the inputs mapped into 4 dimensional subspace by PCA than in the other case for both algorithms use. This does make sense since the size of input dimension is responsible for the number of parameter, e.g interconnection weights, and thus the higher dimension a set of inputs have, the longer time training of a neural network takes. Especially, difference in the training time in the usage of Bayesian Reguralization algorithm, whose training time is generally much longer than the other popular methods, is significant; when mapped inputs by PCA is used, it only takes more or less 25% of training time needed to finish the training with the original inputs. Even in the usage of Levenberg-Marquardt algorithm, which is known to be a faster method, using the inputs projected on to the lower-dimension by PCA in the training leads to approximately 30% drop in training time.

Considering all above, it can be concluded that there is a trade-off relationship between network performance (MSE) and computational efficiency (training time). When very powerful computational environment is available, a network is not so complexed, or a computationally efficient training algorithm is used, it might be reasonable to use the original inputs with high dimension to acquire high performance. On the contrary, when such a good computational environment is not available, a network is very complicated, or a computationally heavy training algorithm is selected, it would be more reasonable to firstly perform dimensionality reduction. Furthermore, in a situation where the inputs are thought to be definitely contaminated by noise, usage of PCA would be recommendable for noise reduction.

9 Input selection by Automatic Relevance Determination (ARD)

Automatic Relevance Determination (ARD) [3] is one of Bayesian models, which identifies the input variables related to prediction of outputs [2]. The usage of this method is verified with the reason that inclusion of input variables irrelevant to the outputs deteriorates a conventional neural network performance especially in a situation where the size of inputs is small whereas their dimension is high. This is mainly because such kinds of neural networks with even regularization find themselves to unable to make parameter associated to irrelevant inputs variables be zero since correlations among those variables and outputs randomly appear because of the finite data set [5].

The basic concept of ARD is to build a model in bayesian framework where a prior belief on parameter in the model, which implicitly capture the measure of *relevance*, is introduced to allow the model to make inference on variables with high relevancy [5].

9.1 Demo: demard.m

This demonstration is done with the following three inputs variables and one output variable of synthetic data.

- x_1: a sample from uniform distribution between $[0, 1]$ contaminated by a small amount of Gaussian noise (std=0.002)
- x_2: a copy of x_1 with relatively large amount of Gaussian noise (std=0.02) being additionally added
- x_3: a sample from the Gaussian distribution with (mean, std)=$(0.5, 0.2)$
- t: $sin(2\pi x_1)$ with Gaussian noise (std=0.05)

This implies that ARD is expected to find that $x_1 1$ is the most relevant variable to the output with x_2 having some relevancy while x_3 has no relevancy to predict outputs.

The prior for weights is given by the ARD Gaussian prior whose hyper-parameter is differently set for a set of weights connected to each input variable. A multi-layer perceptron with the number of hidden units 3 is employed. The training is done with the scaled conjugate gradient algorithm to minimize error. The number of re-estimation of hyper-parameter is set at 2. Hereafter, the basic flow of ARD is briefly described mainly referring to the output of [7].

Since hyper-parameter is given as an inverse of variance, the larger its magnitude is, the smaller the variance is. This means the ARD Gaussian prior $w_i \sim \mathcal{N}(0,\, \alpha_i^{-1})$ [11] has very large PDFs around 0. As a result, posterior distribution is almost dominated by the prior unless the given evidence is comparable to such an informative prior. In other words, a maximum posterior probability (MAP) for w_i is expected to roughly become a value around 0. Thus, when the re-estimated hyper-parameter of an input variable is large, the posterior distribution of a weight associates to that variable has smaller variance and its MAP would be estimated as a small value. This implicitly means that the input variable corresponding to a re-estimated large hyper-parameter has little relevance on outputs. For the cases in which the re-estimated hyper-parameter for the set of weights on an input variable is small, the reasoning discussed above is applied in the opposite way. To be details, the prior has a large variance and thus it is expected that the prior and likelihood function derived from the given data are more likely to be compromised in this case than in a large hyper-parameter case. This suggests that posterior of $w_i$is accompanied by a larger variance and hence its MAP is less likely to be closely 0. In other words, the input variable which is associated to a batch of weights whose re-estimated hyper-parameter is small has a strong relevance on outputs. In this way, ARD based on Bayesian framework is conducted.

Table 25: Re-estimated hyper-parameter

hyper-parameter for	hidden unit (x_1) weights	hidden unit x_2 weights	hidden unit x_3 weights	hidden unit biases	second-layer weights	output unit biases	Coefficient of data error	γ
initial	0.01	0.01	0.01	0.01	0.01	0.01	50.0	
cycle1	0.2042	1.9797	842.0867	0.5373	0.0317	3.4426	176.5643	9.8988
cycle2	0.1778	43.7254	83746.3750	0.6862	0.0174	4.3372	276.5948	8.4334

The table.25 shows the initial hyper-parameter and re-estimated ones. From the bottom line, re-estimated hyper-parameter values for three input variables are respectively $\alpha_{x_1} = 0.1778$, $\alpha_{x_2} = 43.7254$, $\alpha_{x_3} = 83746.3750$. Based on the discussion above, it is concluded that x_1 is the most relevant to the output variable, and x_2 would have moderate relevance to the output. The x_3 can be regarded to have no relevancy on the output. This result matches our previous expectation based on the fact that a synthesis data are used. In fact, as observed in the table.26, the magnitude of weights related to x_1 is the largest among all and that associated to x_2 is 2nd largest with much smaller extent. Regarding weights on x_3, their magnitude is considerably smaller than the others. Again, note that the estimated weights does not become 0 because of the occurrence of random correlation between x_3 and outputs t originated from the usage of finite dataset despite of the fact that x_3 and output t are generated such that they are independent from one another.

Finally, the outputs from the trained network and true underlying function are compared with by plotting on x_1 (fig.38). We can see that the trained network appears to succeed in capturing the unseen true function.

	w_{i1}	w_{i2}
x_1	-3.15993	1.09993
x_2	-0.16043	0.02840
x_3	0.00222	-0.00100

Table 26: Weights from x_i to j^{th} hidden unit w_{ij}

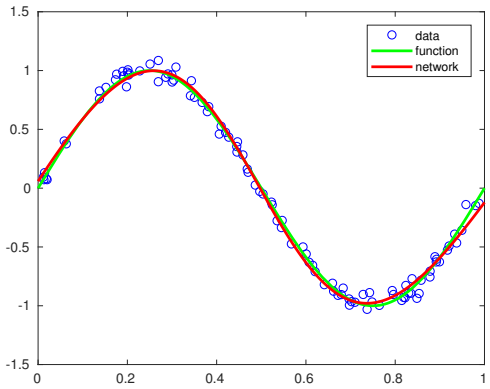


Figure 38: The function corresponding to the trained network vs $f(x) = sin(2\pi x)$ with x_1

9.2 demev1

In this demonstration, a Bayesian regression technique for MLP is illustrated with a synthetic data. The MLP network has one hidden layer with 3 hidden neurons and one output layer with one output neuron. The activation function in output layer is linear map. As input, 16 input patterns (x, t) are used where x is sampled from Gaussian distribution with (mean, std) $= (0.25, 0.07)$ and t is $sin(2\pi x)$ contaminated by Gaussian noise with std $= 0.1$. The relationship between noisy inputs patterns and the true function $sin(2\pi x)$ is depicted in the plots 'o' and green curve in the fig.39.

As prior, the Gaussian noise for targets parameterized by inverse variance hyper-parameter β is assumed under the model as well as a simple Gaussian prior for weights and biases specified with the hyper-parameter α, which is defined as an inverse of variance.

As initialization, $\alpha = 0.01$ is assigned to hyper-parameter parameterizing initial prior, $\beta = 50$ is given to the hyper-parameter specifying initial noise. Subsequently, using **mlp** function, the wights in the network are initialized as following:

- $[w_{h11} \;\; w_{h12} \;\; w_{h13}] = [0.5767 \;\; 0.5034 \;\; 0.9123]$
- $[b_{h1} \;\; b_{h2} \;\; b_{h3}] = [0.4728 \;\; 0.8420 \;\; -0.8503]$
- $[w_{o11} \;\; w_{o21} \;\; w_{o31}]^T = [-0.0099 \;\; -0.0784 \;\; -0.8020]^T$
- $b_{o1} = 0.1287$

After initialization has been done for weights w_i and hyper-parameter, the network is trained with scaled conjugate gradient algorithm with its maximum number of iterations set at 500. In this way, updating the weights are done for certain set of hyper-parameter. Once such an optimization on weights is completed, hyper-parameter is re-estimated using the **evidence** function. Within possible outputs from **evidence** function, *NET, GAMMA* are stored, where GAMMA shows the number of parameters which are well determined [8]. This cycle is repeated three times in total.

Re-estimated hyper-parameter in each cycle is shown in the table.27. The hyper-parameter *beta* is finally re-estimated at $\beta = 67.0382$ in comparison to the true value that is inverse of variance of noise distribution $= \frac{1}{0.1^2} = 100$.

In the fig.39, error bars beside the approximated function by the network are also shown. The area between these two bars is called confidence interval, within which 95% of the conditional means (expected) output $E(y|x)$ are located. We can see that the width of the confidence interval is tighter around the input region roughly $[0.1, 0.4]$ whereas the width increases as a newly given x becomes far from the input region. This is intuitively correct since we have poor confidence on the true response around the region without any data.

	α	β	γ
cycle1	0.10541	70.74711	6.73668
cycle2	0.16521	69.54089	5.88161
cycle3	0.17988	67.0382	5.85715

Table 27: Re-estimated hyper-parameter

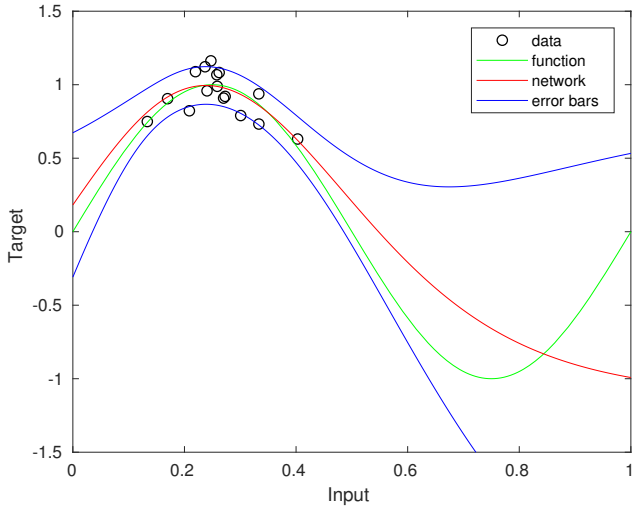


Figure 39: The function corresponding to the trained network vs $f(x) = sin(2\pi x)$ with inputs

A briefly theoretical explanation for this phenomena is studied here mainly referring [9]. Firstly, let a true function value with nonlinear function $f()$ parameterized by θ, given an input x_0 $h\langle x_0; \theta\rangle$ be considered.

Assuming that such a function value is estimated by $\eta_0\langle\hat{\theta}\rangle = h\langle x_0;\hat{\theta}\rangle$, which is approximated as

$$\eta_0\langle\hat{\theta}\rangle \approx \eta_0\langle\theta\rangle + a_0^T(\hat{\theta}-\theta) \quad \text{with } a_0 = \frac{\partial h\langle x_0,\theta\rangle}{\partial\theta} \tag{9.1}$$

with Taylor expansion around θ. Using 9.1, the confidence interval is calculated in the following:

$$\eta_0\langle\hat{\theta}\rangle \pm q_{1-\alpha/2}^{t_{n-p}} \cdot se\langle\eta_0\langle\hat{\theta}\rangle\rangle, \quad \text{with } se\langle\eta_0\langle\hat{\theta}\rangle\rangle = \hat{\sigma}\sqrt{\hat{a_0^T}(A\langle\hat{\theta}\rangle^T A\langle\hat{\theta}\rangle)^{-1}\hat{a_0}}, \tag{9.2}$$

where A is a $n(\#\text{inputs}) \times p(\#\text{parameter})$ matrix whose (i,j) element is defined as following:

$$A_{i,j} := \frac{\partial\eta_i\langle\theta\rangle}{\partial\theta_j}. \tag{9.3}$$

Based on 9.2, we can see the width of the confidence interval is dependent on the vector a^0. From the definition of a^0, we can expect that the more outside from the range of inputs x a newly given input x_0 is, the larger the norm of a_0 will become. This is because $\hat{\theta}$ is obtained with optimization solely based on the given input data, and thus the magnitude of each element of a_0 would be smaller when a new input is within the range of input data than when that is out of such a range. This leads $se\langle\eta_0\langle\hat{\theta}\rangle\rangle$ (eq.9.2) to be much larger for the x laying outside of input range, which results in wider confidence intervals for those x.

9.3 ionosphere data

In this problem, **ionosphere data** with 351 observations having dimension 33 are investigated using a binary classification technique from a nonlinear regression perspective. As a pre-process step for this supervised learning task, ARD is conducted to figure out which variables are most relevant to the output variables.

After important variables are identified, a 2-layer feed-forward network , one of which is fed with full (33) inputs and the other is fed with 4 inputs, are trained. For binary classification based on nonlinear regression, logistic transfer function is selected as output activation function. Furthermore, 5 hidden units are used in the hidden layer.

To perform ADR, a zero-mean isotropic Gaussian prior with inverse variance is taken as prior and its hyper-parameter is introduced. Since there are 33 original inputs variables, the following initial-hyper parameter is set:

- Hyper parameter for hidden unit weights associated to x_i, $i = 1, 2, \cdots, 33 : [0.01,\ 0.01,\ \cdots, 0.01]$
- Hyper parameter for hidden unit biases: 0.01
- Hyper parameter for second-layer weights: 0.01
- Hyper parameter for output unit biases: 0.01.

Based on these initial hyper-parameter, the network weights, hidden interconnection weights 33×5 matrix W_1, 1×5 hidden units biases matrix b_1, 5×1 2nd layer weights matrix W_2, and an output unit biase b_o are initialized using **mlp**. Using the same setting of options for the optimizer, the networks is trained by scaled conjugate gradient algorithm, followed by updating the hyper-prior α, β.

Among all updated parameter, the magnitude of first 33 hyper-parameters are compared on the plot 40 with respect to variables' index. As discussed before, since the hyper parameter is defined as inverse of variance of prior, the smaller the value is, the more relevant the corresponding input variable is to the output variable. It can be seen that re-estimated hyper-parameters for variables $x_1, x_{10}, x_{14}, x_1, x_{27}$ look

smaller than others, though it seems to be difficult to find a clear threshold. Thus, the log(hyper-parameter) is plotted (fig.41), where a jump from 4^{th} smallest log(hyper-parameter) to 5^{th} smallest one is observed. With this reason, the 4 variables shown in the tab.28 are considered to have high relevancy to the output and finally selected as a result of ARD.

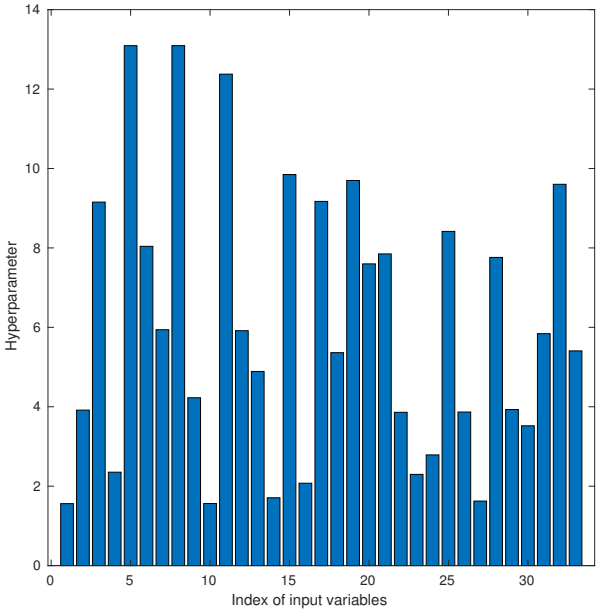


Figure 40: Index of input variables vs re-estimated hyper-parameter

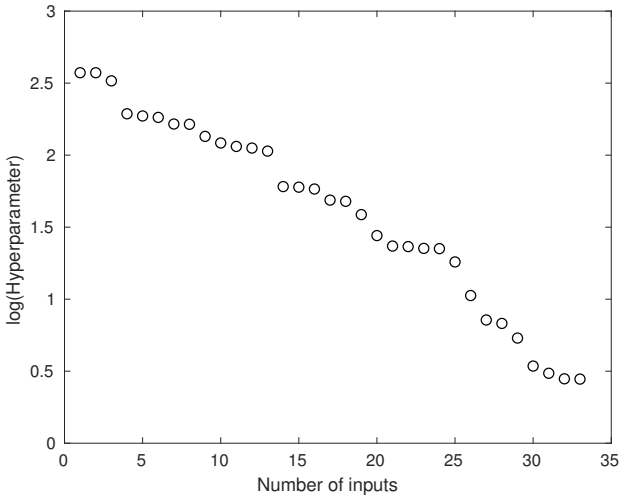


Figure 41: The log(hyper-parameter)

Table 28: Indexies of variables selected by ADR

Selected variables	X_{14}	X˙27	X˙10	X˙1
Hyper-parameter	1.7084	1.6251	1.5637	1.5606

Table 29: Comparison in AUC and misclassification rate between original inputs variables and 4 input variables selected by ARD

	33 inputs variables (Original)	4 Inputs variables (ARD)
AUC	0.9066	0.8939
Misclassification rate	0.1600	0.1500

Using those four variables chosen in ARD process, another network is trained with the other setting held the same as the previous training process for the network with full input variables. To compare the results, confusion matrices (fig.42, 43), ROC curves (fig.44, 45), AUCs, and misclassification rates (29) for the test set are shown for each trained network.

From confusion matrices, we can get the ratio among true positive, false positive, false negative, and true negative for each network. We can notice that misclassification rate of the trained network with 4 variables 0.15 is 1% smalle than the counter part of the trained network with original full inputs varialbes 0.16%. Although this may induce us to easily conclude that the former one would be better classifer than the other, the caution is needed. We have to investigate if the penalty for missclassification, false posive and false negative, is the same or not. If the cost for false positive is not as same as that for fales negative, we should look at ROC curve and AUC.

The difference in ROC curve between two networks does not appear to be remarkable; in fact, AUC for network trained with original inputs variables 0.9066 and that for the network trained with only 4 variables

0.8939 are relatively high and different from each other by roughly 1.3%. This difference may sound so tiny that we could ignore it; however, if the penalty of misclassification is large, it could be justified to use the original inputs variables, although training of such a network is more computationally expensive.

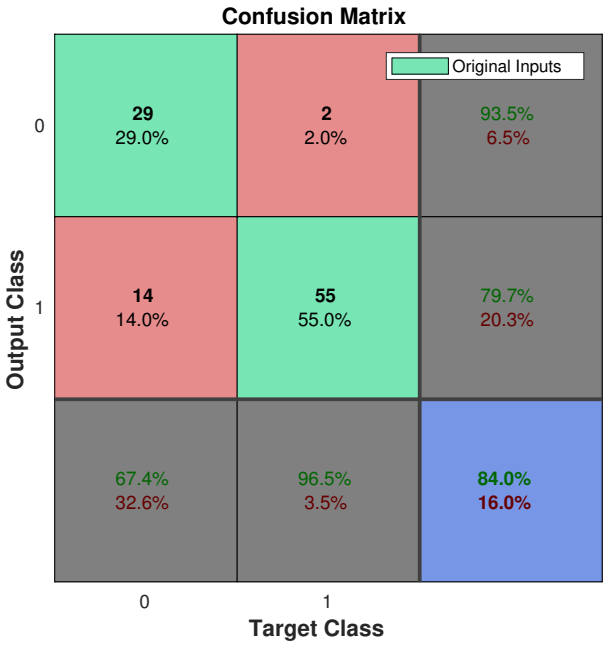


Figure 42: Confusion matrix for test results with trained network from original input variables

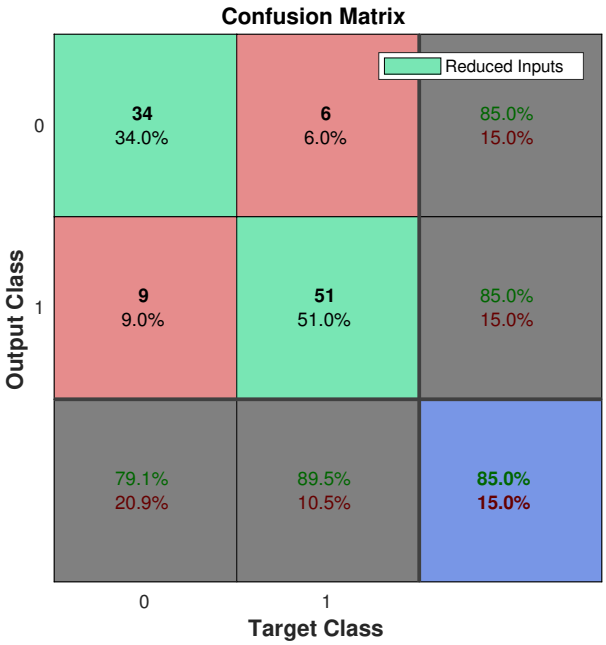


Figure 43: Confusion matrix for test results with trained network from 4 variables selected by ARD

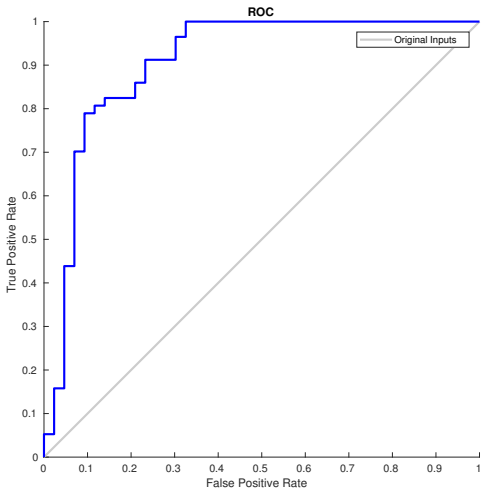


Figure 44: ROC curve for test results with trained network from original input variables

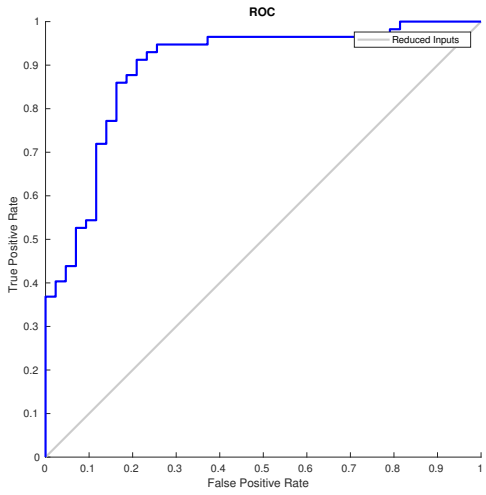


Figure 45: ROC curve for test results with trained network from 4 variables selected by ARD

References

[1] Anima Anandkumar. Anima anandkumar lecture 1+2, 2014. url=https://www.youtube.com/watch?v=NB5NHWzTRSY&t=3212s(Accessed on 3 January 2018).

[2] Suykens J. Data mining and neural networks. 2015.

[3] D. MacKay and R. Neal. Automatic relevance determination for neural networks. *Technical report in presentation*, 1994.

[4] David J. C. Mackay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992.

[5] David J. C. Mackay. Bayesian non-linear modeling for the prediction competition. *Maximum Entropy and Bayesian Methods*, page 221–234, 1996.

[6] Allan D. R. MacQuarrie and Chih-Ling Tsai. *Regression and time series model selection*. World Scientific, 2007. url=https://books.google.be/books?id=5hPtCgAAQBAJ&pg=PA8&lpg=PA8&dq="underfittingisdefined"&source=bl&ots=qK5Qofp2bv&sig=A-uF3pxSfg2L6Tcmv57SeyZR-4o&hl=nl&sa=X&ved=0ahUKEwjev9_copHYAhVPnRQKHeg_ASoQ6AEINzAC#v=onepage&q=%22underfitting%20is%20defined%22&f=false (Accessed on 17 December 2017).

[7] Ian T Nabney. deemev1.m (netlab): Demard automatic relevance determination using the mlp. http://www.ncrg.aston.ac.uk/netlab/, 1996-2001.

[8] Ian T Nabney. evidence.m (netlab): Re-estimate hyperparameters using evidence approximation. http://www.ncrg.aston.ac.uk/netlab/, 1996-2001.

[9] Andreas Ruckstuhl. Introduction to Nonlinear Regression. *ZHAW Zürcher Hochschule für Angewandte Wissenschaften*, 1(October), 2010.

[10] J. R. Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. 1994. url=https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

[11] Michael E. Tipping. The relevance vector machine, 2000.