EEEM005 AI and AI Programming Neural Network Simulation

Yi Zhou 6567008

Executive Summary

In this assignment, an MLP with one hidden layer is trained to make a binary classification on the Wisconsin Breast Cancer Dataset. Firstly, I try different combinations of the number of nodes and the number of epochs. According to the experimental results, if using too few hidden nodes, the performance of the network will be poorer due to the overfitting problem. Using too many hidden nodes can also cause some problems: The classifier may not generalize well on test data due to the overfitting problem. Too many hidden nodes may also make the optimization difficult and thus consumes a lot of training time. Moreover, the number of training epochs can also affect the performance. At the first few epochs, the underfitting occurs due to insufficient training. If we continue training after the loss function converges, the classifier starts to overfit the training set. Finally, considering both the performance and training time, an MLP with 8 nodes and 16 training epochs is used to achieve a test error rate of 0.03869.

To further improve the performance, a hard-voting mechanism is applied. Voting is a simple way to ensemble different classifiers. The ensembled classifier will have a better generalization and better performance than the individuals. According to the experiments, ensemble too many classifiers does not make sense since the performance will reach a plateau and the extra computational cost will be wasted. We can ensemble some complex but under-trained models to achieve comparable performance with a well-trained model. In this assignment, I ensemble 15 MLPs with 8 nodes and 16 training epochs to achieve a test error rate of 0.02657.

Moreover, three different optimizers are studied. Resilient backpropagation (Rprop) is a first-order method which uses the sign of gradients for updating. It makes an adaptive update step for each weight. Levenberg-Marquardt Backpropagation (LM) combines the strength of both gradient descent and Newton’s method. SCG uses a set of conjugate bases to approximate the updating step in Newton’s method. A regularized term is used to overcome the indefiniteness of the Hessian matrix. According to the experimental results, LM is fast to converge but it consumes lots of time per epoch, especially for a complex model with many parameters. SCG seems a moderate choice with a balanced performance in accuracy, stability and speed. Rprop can be slightly faster than SCG, but as the first-order method, it may suffer from the oscillatory optimization path.

Finally, a classifier is designed for a two-class Gaussian mixture distribution. From the experiments, I find for any two classifiers, even the classification errors are similar, the decision boundary could look very different. As training goes, the neural network tends to have a decision boundary with more complex shape. According to the experiments, a hard-voting classifier, which consists of 15 MLPs with 8 nodes and 32 training epochs, fits better with the optimal Bayes decision boundary.

1. Introduction

A multilayer perceptron (MLP) is a class of feedforward artificial neural network model. It is widely used in both the regression and the classification problem. The aim of this assignment is to study the characteristics of MLP with one hidden layer. The experiments are based on the Matlab implementation of a two-class classification problem using the Neural Network Toolbox. Several topics are discussed in this report, including the selection of the hidden nodes and epochs, the voting methods, the performance of different optimizers and the effect of different settings on decision boundary.

The structure of this report is: In chapter 2, some theoretical backgrounds, including neural network architecture, optimization methods and ensemble methods are introduced. Chapter 3 introduces the datasets used in this assignment, one is the Wisconsin Breast Cancer Dataset, the other is generated from two Gaussian distributions. In chapter 4, the experimental results are shown and discussed.

2. Model Description

2.1 Neural Network Architecture

Neuron is the basic component of a neural network. Each neuron takes a feature vector as input. Then, the input dot-products with a weight vector and add a bias term. Finally, the output scalar is sent to an activation function. The single-layer perceptron is the simplest neural network which has only one layer with multiple neurons. Whereas the single-layer perceptron is only able to classify linearly separable patterns, the multilayer perceptron can be used for arbitrary classification problems. It can be proved that an MLP with one hidden layer and sigmoid activation can approximate any functions, provided the sufficient neurons in the hidden layer are available. An MLP contains at least three layers of nodes: an input layer, one or more hidden layers and an output layer. As figure 2.1 shows, an MLP with one hidden layer is used in this assignment.

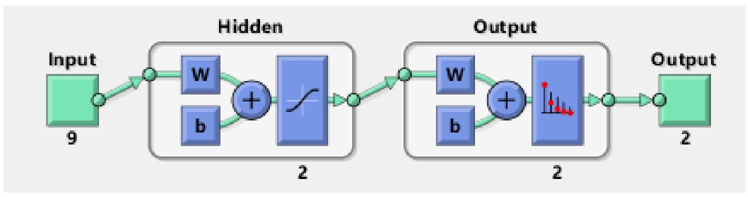


Figure 2.1 MLP structure

For classification problem, we want the network make a prediction of the probability of the input belongs to some specific class. For two class problem, we can use the sigmoid function as the activation in the final layer to output a probability. If we have more than two classes, softmax activation function can be used instead. The task of a neural network is to maximize the likelihood of observing the training data. For two class case, let p_{out} represents the output probability of a particular class, and t represents the binary target. Then, the likelihood of observing the whole dataset is

$$\prod_{N} p_{out}^{t}\,(1 - p_{out})^{1-t}$$

Maximizing the likelihood is equivalent to minimizing the negative log likelihood. Thus, we define the loss function as

$$L = -\sum_{N} t \cdot log(p_{out}) + (1-t) \cdot log(1-p_{out})$$

which is called cross entropy loss function. The backpropagation is used to update the weight parameters to minimize this loss function. The frequently used optimizers will be introduced in the next chapter.

2.2 Learning Methods

2.2.1 Steepest Descent Backpropagation

Training a multi-layer perceptron can be considered as a numerical optimization problem.

Let us take a first order Taylor expansion of the loss function

$$L(w_{k+1}) = L(w_k + \Delta w) \approx L(w_k) + \nabla L(w)^T|_{w=w_k} \Delta w$$

We want $L(w_k) < L(w_{k+1})$, thus the second term, which is the inner product of the gradient and the update step, should be negative. The steepest descent occurs when the update step is along the direction of negative gradient. Thus, we can choose Δw as

$$\Delta w = -\eta \nabla L(w)|_{w=w_k}$$

where η is the learning rate. The gradient $\nabla L(w)$ is computed using the chain rule.

During the training process, there are generally two ways to update the weights. The first is to accumulate all the gradient across the entire epochs and update the weights once per epoch. This is called batch gradient descent.

$$w_{k+1} = w_k - \eta \textstyle\sum_{i=1}^{N} \nabla L(w)|_{w=w_k}$$

If the dataset is large, it will take long time for one update. In such case, stochastic gradient descent (SGD) is favoured. The only difference is to select the batch size N equal to 1. Then, we can incrementally update the weights per single training data.

$$w_{k+1} = w_k - \eta \nabla L(w)|_{w=w_k}$$

SGD will suffer from a slow convergence due to the inherent variance. In practice, we always sum the gradients over a mini-batch instead of using a single data point. Therefore, in deep learning papers, SGD always refers to mini-batch gradient descent.

2.2.2 Resilient Backpropagation

Resilient backpropagation (Rprop) is very similar to the idea of steepest gradient descent. The difference is it only use the sign of the gradient instead of the magnitude. Rather than updating all parameters with a fixed value, Rprop adaptively adjust each parameter with different values. The idea is if the successive gradients are in the same direction, then we accelerate the update. If the gradient changes sign, we decrease the update.

The first step is to compare the gradient of the current iteration to the previous iteration. We can use the dot product of two gradients to determine if the sign of gradient changes. Let us use c_{ij} to represent the indicator for node the $j\ th$ parameter in the $i\ th$ node.

$$c_{ij} = \left.\frac{\partial L(w)}{\partial \mathrm{w}_{ij}}\right|_{w=w_{k+1}} \cdot \left.\frac{\partial L(w)}{\partial \mathrm{w}_{ij}}\right|_{w=w_k}$$

If c_{ij} is negative, indicating the gradient changes its sign, we increase the weight by an updating step. Similarly, if c_{ij} is positive, we decrease the weight by an updating step. If c_{ij} is zero, there will be no weight change.

$$\Delta \mathrm{w}_{k+1}^{(i,j)} = \begin{cases} -\Delta_{k+1}^{(i,j)}, & if\ c_{ij} > 0 \\ +\Delta_{k+1}^{(i,j)}, & if\ c_{ij} < 0 \\ 0, & otherwise \end{cases}$$

As the training process, the step Δ_{k+1} is adaptively updated for each different weight. Let us use $\Delta_{k+1}^{(i,j)}$ to represent the $j\ th$ parameter in the $i\ th$ node. If c_{ij} is positive, the updating step is increased by a growth factor η^+. If c_{ij} is negative, the updating step is decreased by a decay factor η^-. If c_{ij} is zero, the updating step is unchanged.

$$\Delta_{k+1}^{(i,j)} = \begin{cases} \eta^+ \Delta_k^{(i,j)}, & if\ c_{ij} > 0 \\ \eta^- \Delta_k^{(i,j)}, & if\ c_{ij} < 0 \\ \Delta_k^{(i,j)}, & otherwise \end{cases}$$

In practice, both η^+ and η^- can be considered as default parameters for the algorithm, which are usually set to 1.2 and 0.5. The initial update step is set to 0.07. Since the parameters can be adaptively updated, there is no need to tune them for every training.

Compared to the steepest gradient descent, resilient backpropagation enables a fast update in the flat area, therefore a faster convergence.

2.2.3 Levenberg-Marquardt Backpropagation

Levenberg-Marquardt (LM) can be considered as a hybrid of both the gradient descent and the Newton's method. Firstly, we make a quick review of the Newton's method. To derive it, the second order term is retained when doing Taylor expansion of the loss.

$$L(w_{k+1}) = L(w_k) + g_k^T \Delta w_k + \frac{1}{2} \Delta w^T H_k \Delta w_k$$

where $g_k = \nabla L(w)|_{w=w_k}$ is the gradient and $H_k = \nabla^2 L(w)|_{w=w_k}$ is the Hessian matrix.

From an optimization point of view, learning is equivalent to minimizing this approximated quadratic function with respect to the update step Δw_k. Thus, the update rule for Newton's method is given by

$$\frac{\partial L}{\partial \Delta w_k} = H_k \Delta w_k + g_k = 0$$

$$\Delta w_k = -H_k^{-1} g_k$$

Since the computation of the Hessian matrix is time-consuming, it can be approximated by $H_k = J_k^T J_k$, where J_k denotes the Jacobian matrix.

Newton's method takes more aggressive update to fast converge to the extreme point of a quadratic surface. However, it cannot distinguish between minima, maxima and saddle points. Moreover, it can oscillate if the surface is complex.

In general, newton's method is fast but is possible to oscillate or diverge, while gradient descent is slow but guaranteed to converge. To combine the strength of these two methods, we can define a trust region. Inside the trust region, the Hessian is not nonnegligible. Thus, the second-order Newton's method is suitable for update. Beyond the trust region, the first order approximation is enough, then we can use gradient descent. LM is one of such trust region methods. It takes the form of

$$w_{k+1} = w_k - (H_k + \lambda I)^{-1} g_k$$

where I is identity matrix and λ is the damping ratio. When the Hessian matrix is large, indicating the curvature is large, LM acts like Newton's method. In smooth region where curvature is small, LM acts like gradient descent. As a result, the LM method can combine the strengths of both.

2.2.4 Scaled Conjugate Gradient Descent

The second order methods need computation the inverse of Hessian matrix which is time consuming. Conjugate gradient descent can be considered as an approximation to the second order methods. The key is to use a set of conjugate bases to approximate the step in Newton's method, and search the path iteratively along these bases. Its computational complexity is lower than that of the second-order methods, but it can alleviate the zig-zag problems in the first-order methods.

For the second-order Taylor expansion of the loss function, the critical point $\Delta \mathrm{w}_k^*$ is determined by

$$\frac{\partial L}{\partial \Delta w_k} = H_k \Delta \mathrm{w}_k^* + g_k = 0$$

Let $p_1, \dots, p_N$ be a conjugate system determined by the Hessian. Suppose we takes m step to reach the critical point $\Delta \mathrm{w}_k^* = \Delta \mathrm{w}_k^1 + \cdots + \Delta \mathrm{w}_k^m$. Then, the step to $\Delta \mathrm{w}_k^*$ from the initial point $\Delta \mathrm{w}_k^1$ can be expressed as a linear combination of the conjugate bases as

$$\Delta \mathrm{w}_k^* - \Delta \mathrm{w}_k^1 = \sum_{i=1}^{N} \alpha_i p_i$$

The weight α_1 can be determined by

$$p_1^T H_k (\Delta \mathrm{w}_k^* - \Delta \mathrm{w}_k^1) = p_1^T \left(-g_k - H_k \Delta \mathrm{w}_k^1\right) = \alpha_1 p_1^T H_k p_1$$

$$\alpha_1 = -\frac{p_1^T \left(H_k \Delta \mathrm{w}_k^1 + g_k\right)}{p_1^T H_k p_1} = -\frac{p_1^T \left.\frac{\partial L}{\partial \Delta w_k}\right|_{\Delta w_k = \Delta \mathrm{w}_k^1}}{p_1^T H_k p_1}$$

In practice, we initialize the first step p_1 as the steepest gradient at that point, i.e.

$$p_1 = -\left.\frac{\partial L}{\partial \Delta w_k}\right|_{\Delta w_k = \Delta \mathrm{w}_k^1}$$

and construct the other conjugate basis recursively. An approximation of the Hessian matrix can be used to reduce the computational cost. The conjugated gradient descent algorithm requires the Hessian matrix to be definite. However, in neural network, the Hessian matrix for the loss function can be indefinite in some areas, leading to a poor performance. To solve this, scaled conjugated gradient descent (SCG) adopts a similar method as LM. It uses a scalar λ_k to regulate the indefiniteness, i.e. $H_k + \lambda_k I$. If the Hessian matrix is indefinite, λ_k will be raised to make the matrix definite. Due to the page limitation, the details will not be fully introduced in this report, as they can be found in the original paper[1].

2.3 Voting Classifier

Different weight initializations and the randomness in the training data will lead to different classifiers. Ensemble method makes full use of multiple such classifiers in order to improve the generalization. It trains multiple classifiers and make a prediction based on all the different classifiers. Voting is a simple way to ensemble different classifiers. It can be divided into two classes: hard voting and soft voting.

In hard voting, each classifier predicts the class independently and then makes a voting. The class with majority voting wins. In soft voting, the predicted probabilities of each classes by different classifiers are weighted average. The final classification is made based on the highest average probability.

This voting mechanism is especially helpful for a small dataset. In such case, the training data could be not representative and the classifier could not generalize well on the whole dataset. In other words, the classifier is likely to approximate the training data locally. Assemble these local classifiers could make a performance improvement over the whole data.

3. Datasets

The dataset used in this project is the Wisconsin Breast Cancer Database. It consists of 699 data. Each data has 9 features and one binary label. The meanings of the features are shown in the below table. The ratio of the positive samples to negative samples is 65.5% to 34.5%. The data set is divided into training set and test set according to the ratio of 50% to 50%.

Table 3 Wisconsin Breast Cancer Database

Index	Features	Values
1	Clump thickness	[0,1]
2	Uniformity of cell size	[0,1]
3	Uniformity of cell shape	[0,1]
4	Marginal Adhesion	[0,1]
5	Single epithelial cell size	[0,1]
6	Bare nuclei	[0,1]
7	Bland chromatin	[0,1]
8	Normal nucleoli	[0,1]
9	Mitoses	[0,1]
	Binary Target	Values
	Benign	0
	Malignant	1

The auxiliary dataset is generated from two Gaussian distributions shown as below. Class 1 has mean $\mu_1 = [0,0]$ and covariance $diag(1)$. Class 2 has mean $\mu_2 = [2,0]$ and covariance $diag(4)$. In the optimal case, each data should be assigned according to the Bayes decision rule which maximize the probability of $p(w_i|x)$. This probability can be interpreted as the probability of assign the data point x to the class w_i. The Bayes error is the optimal error the classifier can achieve. In other words, it is the lower boundary of the classification error. The

Bayes boundary is where two probability is equal, i.e. $p(w_1|x) = p(w_2|x)$. In the figure below, the Bayes optimal boundary is shown as the yellow circle.

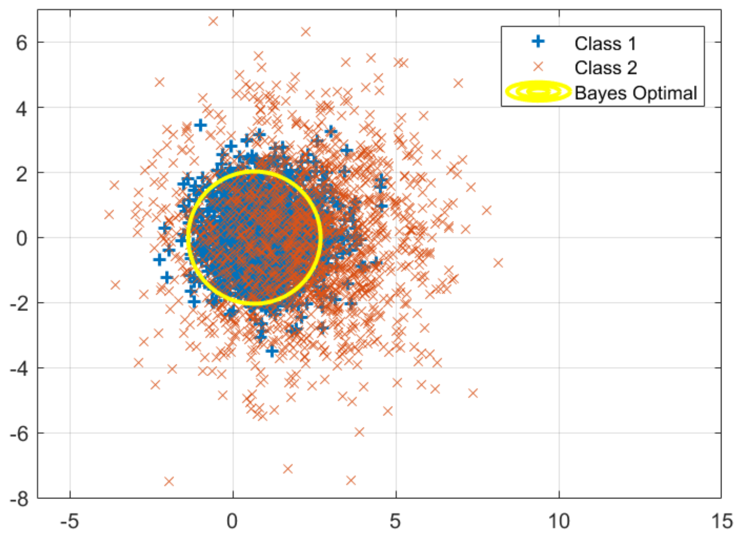


Figure 3 Two-Class Gaussian Distributions

4. Experimental Results

4.1 Hidden Nodes and Epochs

It is very important to select a suitable number of hidden nodes and training epochs for an MLP classifier. In this experiment, a series of MLP are trained on the Wisconsin Breast Cancer Database. The number of nodes is chosen from [2,4,8,32] and the number of epochs is chosen from [2,8,16,32,64]. For every case, we train the classifier 30 times with different split of dataset. The means and standard deviations of both training error rate and test error rate are recorded. The experimental results are shown in figure 4.1.1.

Firstly, let us investigate each plot separately. At the first few epochs, all of these four metrics are high. Since there are not enough training data for the optimizer, the neural networks suffer from the underfitting problem. Their prediction is inaccurate and has a high variance due to the randomness. As the training process goes from the 3rd epoch to the 8th epoch, the performance improves a lot. From the view of optimization, the optimizer is taking greedy steps to quickly converge to the minimum of the loss surface. From the 8th to the 16th epoch, the test error drops slightly as the training error keeps dropping fast, indicating the overfitting occurs. If we continue training, after 32nd epoch, the training error is still decreasing while the test error starts to increase. The overfitting phenomenon is becoming severe. Moreover, the variance of test error also increases. Since the classifier tends to overfit the training data near boundary. As we split different training samples, the local distribution of these data is also random. Thus, the overfitted classifier could have an increasing variance of test error.

Next, let us investigate the effect of the network structure, i.e. the hidden nodes, on the classification performance. From figure 4.1, we can find as we use more hidden nodes, the training error drops faster and the training error can converge to a smaller number. In other words, if we use a more complex model, the model has better fitting ability and is easier to overfit the training data. In addition, too many nodes also make the optimization difficult and thus consumes a lot of training time. On the other hand, if we use too few hidden nodes, for example 2 nodes, the best performance of the model is poorer than that of complex model.

Therefore, we need to find a suitable stop point between underfitting and overfitting. In practice, an early stop can be adopted as a regularization method.

To determine the best combination of nodes and epochs, we need jointly consider several aspects: Firstly, the overfitting cannot be too severe. Secondly, the variance of test error should be small. Thirdly, the mean of test error should be as small as possible. Finally, the training speed should be as fast as possible. From figure 4.1, 4 nodes with 16 epochs and 8 nodes with 16 epochs has comparable performance. From figure 4.2, it is a little bit strange, but the experiments show 8 hidden nodes structure is much faster than 4 hidden nodes structure. The reason could be the internal optimization mechanism used by Matlab. Anyway, since the 8 hidden nodes case is faster, we finally pick it as the classifier structure. The minimal test error is 0.03869 which occurs at the 16th epoch.

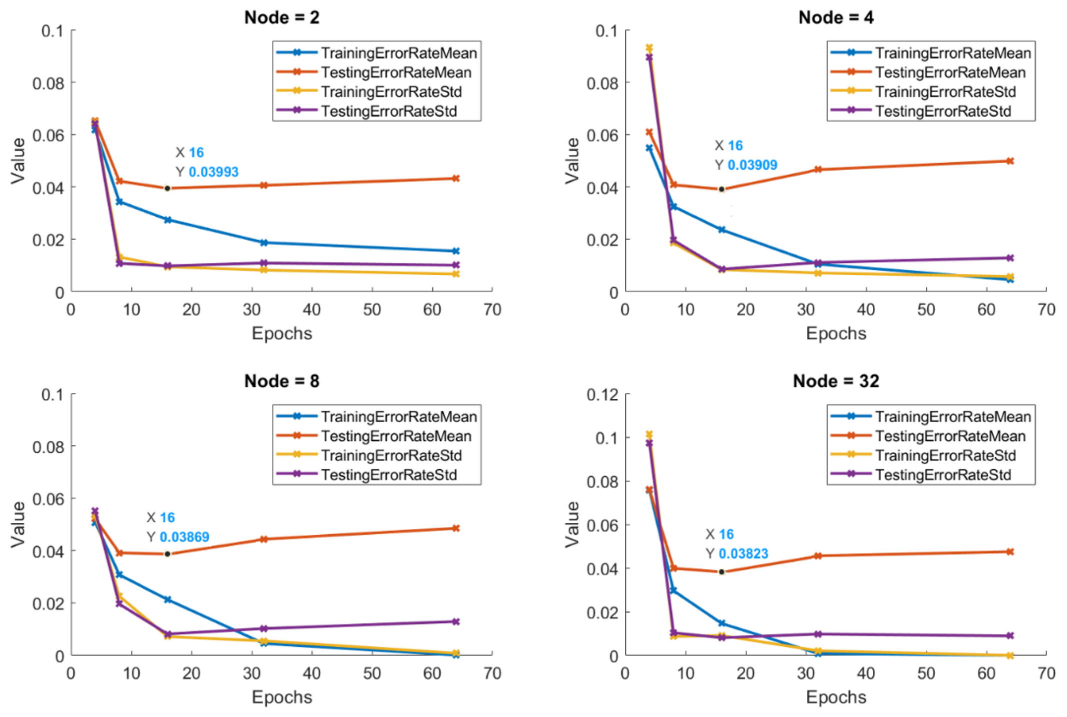


Figure 4.1 The effect of the number of nodes and the number of epochs

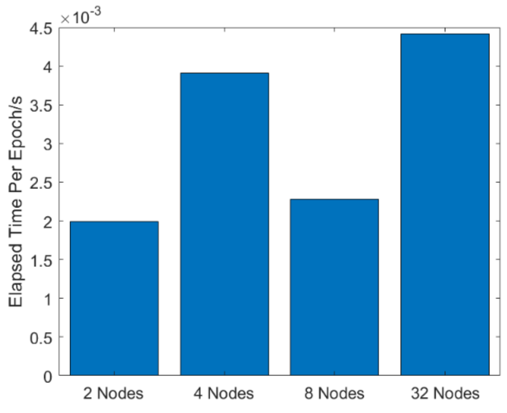


Figure 4.2 Consumed time per epoch

4.2 Model Ensemble

In this section, a hard voting is used to ensemble several classifiers to make a robust prediction. The base classifier is selected as an MLP with 8 hidden nodes. The model is trained for 16 epochs and reaching a 0.03869 mean error rate. Figure 4.2.1 shows the effect of number of ensembled classifiers on the performance. The initial point is the case when no voting applied. We can find as we use more classifiers, both the mean and the standard deviation of the test error rate decreases. Because of the noise and random model initializations, the optimized models of every training are not deterministic. For some hard test samples, the prediction could be of low confidence. By adopt voting mechanism, such ambiguity can be reduced, yielding to a stable prediction. Moreover, the training error rate increases and converges with the test error. An individual classifier will overfit the training data but generalize poorly on test data, leading to a high variance. By voting of different classifiers, the ensembled classifier will have a better generalization. Thus, the overfitting problem in the training data is greatly alleviated. From the figure, if we ensemble 15 classifiers, the mean test error rate reaches its lowest as 0.02448. If we use more classifiers, the performance does not improve significantly but it takes many times of inference time. Therefore, ensemble of 15 classifiers could be a proper choice for this task.

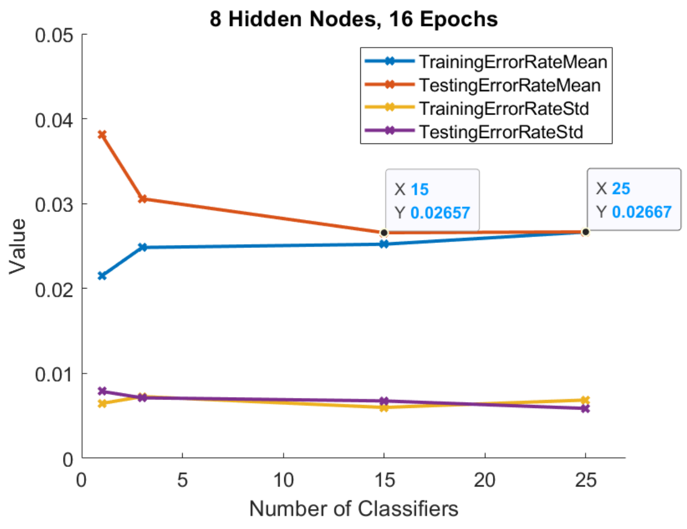


Figure 4.2.1 Ensemble of different numbers of classifiers

Furthermore, I investigate how the number of hidden nodes and training epochs affect the ensemble performance. Here I test the individual classifier with 2, 8, 32 nodes and 8, 16, 32 epochs. In each case, I make a hard voting using 15 individually trained MLPs and record the mean and standard deviation of the test error rate. As the figure 4.2.2 shows, the lighter blue indicates a smaller value, which is desirable. The left column corresponds to the performance of base classifiers. We can find the performance can be improved by either using complex structure or more training epochs.

Firstly, let us investigate each column where the network structure is fixed. As we train more epochs, the performance improves, but the improvement is not significant if we train more than 16 epochs. After 16 epochs, the individual classifiers are well trained. There is little margin for model ensemble to improve.

Next, let us investigate each row where the epoch is fixed. As we use a more complex structure with more hidden nodes, the performance improves. Especially when we train few epochs, the individual classifier will underfits the training set. However, for complex models with strong fitting capability, they tend to only approximate the local structure of the feature space. Then, if we ensemble lots of such complex but under-trained models, the overall performance could improve a lot. From the figure, we can find for the ensemble of 15 MLPs with 32 nodes after 8 epochs training, the mean test error is 0.0261, which is comparable with the ensemble of 15 well-trained MLPs with 8 nodes. Therefore, considering both the time and accuracy, ensemble of 15 MLPs with 32 hidden nodes and 8 epochs training could be the best choice in this example.

In summary, we can draw the following conclusions: The ensembled classifier will have better generalization and better performance than the individuals. The ensemble of some complex but under-trained models can have a comparable performance with ensemble of some less complex but well-trained models. The margin of improvement reduces as the individual models are trained for more epochs.

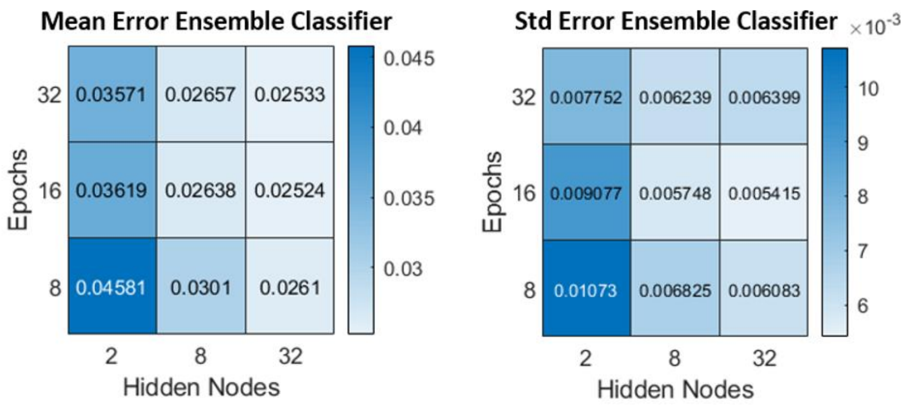


Figure 4.2.2 The effect of individual classifiers' hidden nodes and training epochs

4.3 Optimizer Selection

In this section, I compare the performance of different optimizers, including resilient backpropagation (Rprop), Levenberg-Marquardt (LM) backpropagation and scaled conjugate gradient descent (SCG). The evaluation is based on 5 aspects: consumed time, mean of training error, standard deviation of training error, mean of test error and standard deviation of test error. Let us firstly give a fast review of the characteristic of each optimizer. According to section 2.2, Rprop only uses the sign of gradients for updating, thus enabling a fast convergence. The other advantage is there are fewer hyperparameters to tune. However, since it only uses the first-order information, it could be oscillatory and slower than the other two. LM uses both the first-order gradient and the second-order Hessian matrix: Where the curvature is large, LM acts like Newton's method and finds a fast path for the descent. In a smooth region where the curvature is small, LM acts like gradient descent. It will slow down but is guaranteed to converge. However, since it needs to approximate the inverse of the Hessian matrix, the computation cost is intensive for high-dimensional cases. SCG can be considered as an approximation to the second order methods. It uses a set of conjugate bases to approximate the step in Newton's method, and search the path iteratively along these bases. Moreover, a regularized term is used to overcome the indefiniteness of the Hessian matrix. Its computational complexity is lower than that of the second-order methods.

Figure 4.3 shows the experimental results. The blue, orange and yellow represent SCG, LM and Rprop respectively. Firstly, let us investigate each column, where the training epoch is same but the number of nodes vary. As we use more complex model, the training and test error rate decrease for all three optimizers. Among them, the error of LM drops more obviously. Therefore, we can infer LM finds a faster optimization path than the other two when model is complex. Then, let us consider the consumed time by each optimizer. When the number of nodes is small, the consumed time is roughly equal for all these three optimizers. Rprop is a little bit faster but its performance is also worse than others. When the number of nodes is large, for example the 32 nodes case, LM consumes more than three times as much time as the other two, but have a significant improvement on the performance. Therefore, we can say that LM finds a faster converging path at the expense of more consumed time.

Next, let us investigate each row, where the number of nodes is fixed but training epochs vary from 8 to 32. For LM, the performance changes a little as epoch increases from 8 to 16, indicating the loss function has already converged at the $8th$ epoch. If we continue training after convergence, the time per epoch for LM increases. Therefore, an early stop should be applied if the loss converges. For SCG and Rprop, their error rates continue to drop from the $8\ th$ epoch to the $16\ th$ epoch, indicating a slower convergence than LM. At the $4th$ epoch, the performance of Rprop is sometimes better than SCG but also sometimes much worse. As a first-order method, Rprop could have a zig-zag optimization path, leading to an oscillatory learning curve. SCG follows a smoother path and thus has a steadily drop of error rate during the training process.

In summary, LM needs fewer epochs to converge but consumes more time per epoch. Since the overall time equals time per epoch times required epochs for convergence, we need to carefully consider this trade-off. Generally, it is not suggested to use LM for a complex model with many parameters. Rprop is slightly faster than SCG, but its learning curve is not as smooth as the other two. SCG seems a moderate choice with a balanced performance in accuracy, stability and speed.

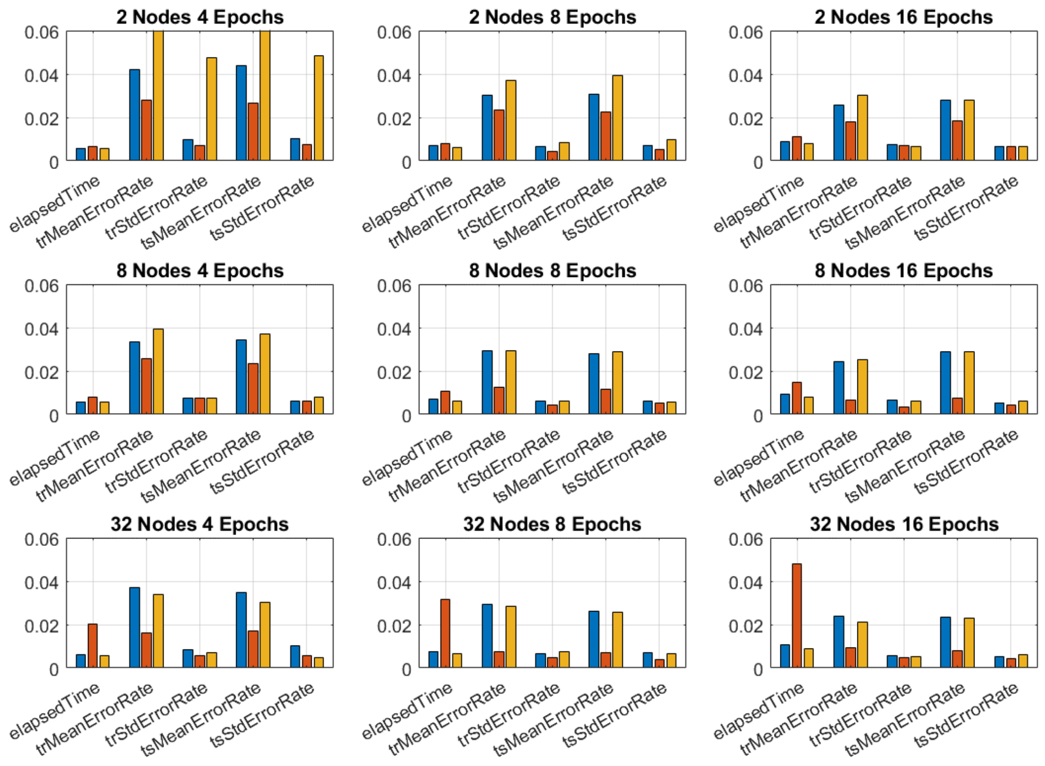


Figure 4.3 Performances using different optimizers
(blue: SCG; orange: LM; yellow: Rprop.
x-axis labels from left to right: time per epoch; mean of training error rate; standard deviation of training error rate; mean of test error rate; standard deviation of test error rate
the time is rescaled by divided by 2000)

4.4 Decision Boundary for a Mixture of Gaussians

In order to find the optimal nodes and epochs for this Gaussian dataset, I try different combinations of parameter settings. The results are shown in figure 4.4.1. From the figure, we can find the two nodes and 4 nodes structures give relatively large errors. 8 nodes and 16 nodes structure give a similar minimum test error, which are 0.2513 at 16 epochs and 0.2495 at 16 epochs. Considering the risk of overfitting, I select the 8 nodes structure as the base classifier. Then, in order to further improve the accuracy, I using a soft voting with 15 base classifiers.

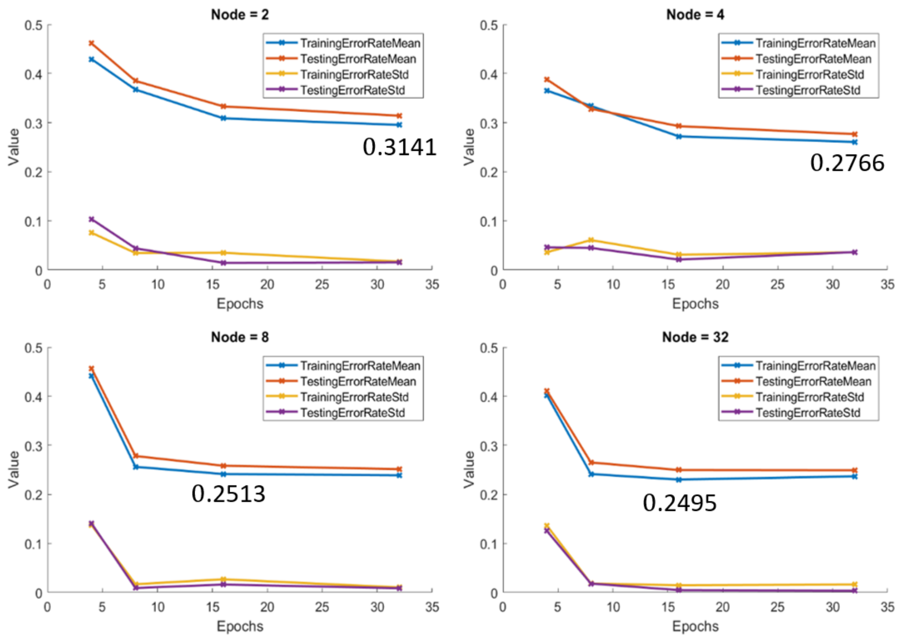
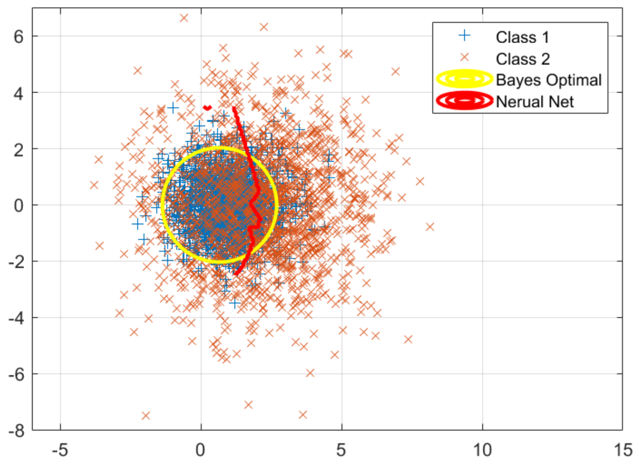
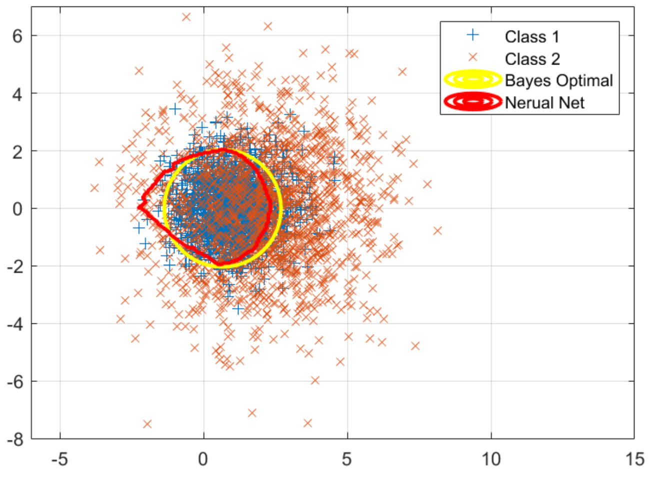


Figure 4.4.1 Evaluations of classifiers with different nodes and training epochs

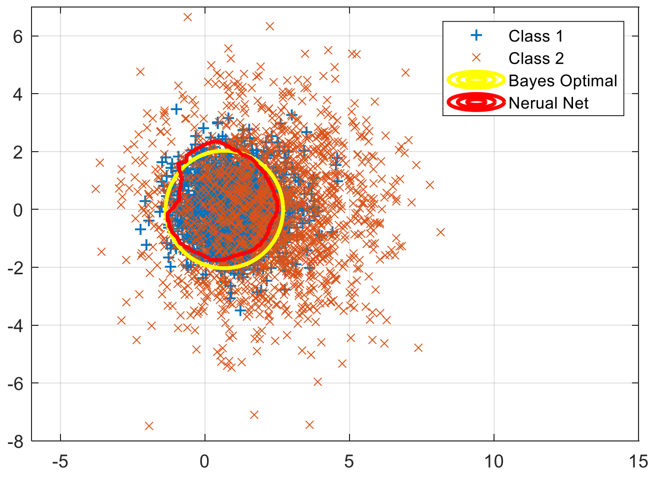
Unlike the decision boundary corresponding to the Bayes error, it is not possible to get the theoretical solution of the neural networks. Therefore, I make a gird with a size of 90x120 and resolution of 0.05 over the data space. At each grid point, I use the ensembled classifier to make a prediction. Finally, I cluster the data with the same predicted classes and extract the contour as the decision boundary. Then, I adjust the number of epochs to better fit the Bayes decision boundary. The decision boundaries are plotted as the red curve in figure 4.4.2 (a)(b)(c). We can find that although the classification errors are similar, the decision boundary looks very different. In the first image, the number of training epochs is 4, and its decision boundary is an open line. Clearly, the underfitting occurs, which is consistent with the curve in figure 4.4.1. In the second image, the number of training epochs is 16, and its decision boundary is a closed smooth contour. In the third image, the number of training epochs is 32, and its decision boundary is a closed jittering contour. We can infer, as training goes, the neural network pays more attention to the data near the boundary, yielding to a decision boundary with more complex shape. It should be noticed that a full recovery of the circle boundary is almost impossible. Because of the nonlinearity in the activation function, there always exists some information loss in the backpropagation process, thus preventing a full recovery. An alterative way to fully recover the decision boundary in this multiple Gaussian case it to use RBF networks. To get the best fitted decision boundary, we need to find a compromised choice between underfitting and overfitting. In this problem, the decision boundary in the third image, i.e. the 32 epochs case, is more suitable since it fits better with the Bayes decision boundary.



(a) training for 4 epochs



(b) training for 16 epochs



(c) training for 32 epochs

Figure 4.4.2 The decision boundary

5 **Conclusion**

In this assignment, an MLP with one hidden layer is trained to make a binary classification on the Wisconsin Breast Cancer Dataset. The best performance is achieved using a voting classifier with 15 base MLP classifier. Each MLP classifier has 8 hidden nodes and is trained for 16 epochs. Moreover, a similar voting classifier is trained to recover the decision boundary of two-class Gaussian distributions. In addition, several experiments are the parameter settings are made and some findings are summarized as follows:

It is very important to select a suitable number of hidden nodes for an MLP classifier. On the one hand, using too few hidden nodes will lead to underfitting. On the other hand, using too many hidden nodes can also cause some problems: Firstly, the trained classifier may not generalize well on test data due to the overfitting problem. Too many nodes also make the optimization difficult and thus consumes a lot of training time. The number of training epochs can also affect the performance of the classifier. At the first few epochs, the loss function not yet converges, thus an underfitting occurs. If we continue training after the loss function converges, the classifier tends to overfit the training set. Therefore, it is suggested to adopt the early stop as a regularization method.

Voting is a simple way to ensemble different classifiers. The ensembled classifier will have a better generalization and better performance than the individuals. It is not suggested to ensemble too many classifiers since the performance will reach a plateau and the extra computational cost is wasted. The ensemble of some complex but under-trained models can have a comparable performance with an ensemble of some less complex but well-trained models.

Different optimizers have their own advantages and disadvantages. In practice, we need to consider the trade-off between the converging speed and the computational complexity. LM is fast to converge but it consumes lots of time per epoch, especially for a complex model with many parameters. SCG seems a moderate choice with a balanced performance in accuracy, stability and speed. Rprop can be slightly faster than SCG, but as the first-order method, it may suffer from the oscillatory optimization path.

For two classifiers, even the classification errors are similar, the decision boundary could look very different. As training goes, the neural network pays more attention to the data near the boundary, yielding to a decision boundary with more complex shape. In practice, we need to find a compromised choice between underfitting and overfitting. In order to fully recover the decision boundary for multiple Gaussians case, RBF networks can be used instead.

Reference

[1] Møller, Martin F. *A scaled conjugate gradient algorithm for fast supervised learning*. Aarhus University, Computer Science Department, 1990.
[2] Beale, Hagan Demuth, Howard B. Demuth, and M. T. Hagan. "Neural network design." *Pws, Boston* (1996).
[3] Heaton, Jeff. *AIFH, Volume 3: Deep Learning and Neural Networks*. 2015.

Appendix

A1. cancer_nn.m

```
clear
clc
rng(3)  % For reproducibility

load cancer_dataset.mat
inputs = cancerInputs;
targets = cancerTargets;
% shallow network
hiddenLayerSize = [2 4 8 32];
trainFcn = 'trainscg';
performFcn = 'crossentropy';
nEpochs = [4 8 16 32 64];
loop = 50;
elapsed = zeros(1,length(hiddenLayerSize));

for i = 1:length(hiddenLayerSize)
trMeanErrorRate = zeros(1,length(nEpochs));
tsMeanErrorRate = zeros(1,length(nEpochs));
trStdErrorRate = zeros(1,length(nEpochs));
tsStdErrorRate = zeros(1,length(nEpochs));

for j = 1:length(nEpochs)
if nEpochs(j) == 16
    tic;
end
trErrorRate = zeros(1,loop);
tsErrorRate = zeros(1,loop);



net = patternnet(hiddenLayerSize(i), trainFcn,performFcn);
net.trainParam.epochs = nEpochs(j);

net.divideParam.trainRatio = 0.5;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 0.5;
for l=1:loop
net = init(net);
[net,tr] = train(net,inputs,targets);

target = targets(1,:);
trTarg = [target(tr.trainInd);1-target(tr.trainInd)];
tsTarg = [target(tr.testInd);1-target(tr.testInd)];

outputs = net(inputs);
output = outputs(1,:);
trOut = [output(tr.trainInd); 1-output(tr.trainInd)];
tsOut = [output(tr.testInd); 1-output(tr.testInd)];

[trErrorRate(l),~,~,~] = confusion(trTarg,trOut);
[tsErrorRate(l),~,~,~] = confusion(tsTarg,tsOut);
```

```
performance = perform(net,targets,outputs);
end
trMeanErrorRate(j) = mean(trErrorRate);
trStdErrorRate(j) = std(trErrorRate);
tsMeanErrorRate(j) = mean(tsErrorRate);
tsStdErrorRate(j) = std(tsErrorRate);

if(nEpochs(j) == 16)
    elapsed(i) = toc;
end

end

figure(1)
subplot(2,2,i)
hold on
plot(nEpochs,trMeanErrorRate,'x-')
plot(nEpochs,tsMeanErrorRate,'x-')
plot(nEpochs,trStdErrorRate,'x-')
plot(nEpochs,tsStdErrorRate,'x-')
legend('TrainingErrorRateMean','TestingErrorRateMean','Trainin
gErrorRateStd','TestingErrorRateStd')
xlabel('Epochs')
ylabel('Value')
t = 'Node = '+ string(hiddenLayerSize(i));
title(t)
ax = gca;
ax.FontSize = 12;
set(findall(gcf,'type','line'),'linewidth',2);
axis([0 70 0 0.1])

figure(2)
X = categorical({'2 Nodes','4 Nodes','8 Nodes','32 Nodes'});
X = reordercats(X,{'2 Nodes','4 Nodes','8 Nodes','32 Nodes'});
Y = elapsed/50/16;
bar(X,Y)
ylabel('Elapsed Time Per Epoch/s')
ax = gca;
ax.FontSize = 12;
set(findall(gcf,'type','line'),'linewidth',2);

end
```

A2. cancer_nn_voting.m

```
clear
clc
rng(3)  % For reproducibility

load cancer_dataset.mat
inputs = cancerInputs;
targets = cancerTargets;
% shallow network
hiddenLayerSize = [2 8 32];
```

```
nEpochs = [8 16 32];
trainFcn = ["trainscg", "trainlm", "trainrp"];
performFcn = 'crossentropy';
nClassifier = [1 3 15 25];

loop = 30;

tic

% voting nodes = 8, epoch = 16
trMeanErrorRate = zeros(1,length(nClassifier));
tsMeanErrorRate = zeros(1,length(nClassifier));
trStdErrorRate = zeros(1,length(nClassifier));
tsStdErrorRate = zeros(1,length(nClassifier));
for i = 1:length(nClassifier)
trErrorRate = zeros(1,loop);
tsErrorRate = zeros(1,loop);

net = patternnet(8, "trainscg",performFcn);
net.trainParam.epochs =16;

for l=1:loop
nets = {};
net.divideParam.trainRatio = 0.5;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 0.5;
for j = 1:nClassifier(i)
net = init(net);
[net,~] = train(net,inputs,targets);
nets{j} = net;
end

[~,tr] = train(net,inputs,targets);

target = targets(1,:);
trTarg = [target(tr.trainInd);1-target(tr.trainInd)];
tsTarg = [target(tr.testInd);1-target(tr.testInd)];

output = zeros(1,length(target));
for j = 1:nClassifier(i)
net = nets{j};
outputs = net(inputs);
output = output + round(outputs(1,:));
end

output = sign(output/nClassifier(i) - 0.5)/2 + 0.5;
% hard voting
trOut = [output(tr.trainInd); 1-output(tr.trainInd)];
tsOut = [output(tr.testInd); 1-output(tr.testInd)];

[trErrorRate(l),~,~,~] = confusion(trTarg,trOut);
[tsErrorRate(l),~,~,~] = confusion(tsTarg,tsOut);
end
trMeanErrorRate(i) = mean(trErrorRate);
trStdErrorRate(i) = std(trErrorRate);
tsMeanErrorRate(i) = mean(tsErrorRate);
```

```
tsStdErrorRate(i) = std(tsErrorRate);
end

figure(1)
hold on
plot(nClassifier,trMeanErrorRate,'x-')
plot(nClassifier,tsMeanErrorRate,'x-')
plot(nClassifier,trStdErrorRate,'x-')
plot(nClassifier,tsStdErrorRate,'x-')
legend('TrainingErrorRateMean','TestingErrorRateMean','Trainin
gErrorRateStd','TestingErrorRateStd')
xlabel('Number of Classifiers')
ylabel('Value')
%t = 'Number of Classifiers = '+ string(nClassifier(i));
title('8 Hidden Nodes, 16 Epochs')
axis([0 27 0 0.05])
ax = gca;
ax.FontSize = 12;
set(findall(gcf,'type','line'),'linewidth',2);

toc

% nClassifier = 15
tic
trMeanErrorRate = zeros(length(hiddenLayerSize),
length(nEpochs));
trStdErrorRate = zeros(length(hiddenLayerSize),
length(nEpochs));
tsMeanErrorRate = zeros(length(hiddenLayerSize),
length(nEpochs));
tsStdErrorRate = zeros(length(hiddenLayerSize),
length(nEpochs));

for i = 1:length(hiddenLayerSize)
for j = 1:length(nEpochs)
trErrorRate = zeros(1,loop);
tsErrorRate = zeros(1,loop);

net = patternnet(hiddenLayerSize(i), "trainscg", performFcn);
net.trainParam.epochs =nEpochs(j);

for l=1:loop
nets = {};
net.divideParam.trainRatio = 0.5;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 0.5;
for k = 1:15
net = init(net);
[net,~] = train(net,inputs,targets);
nets{k} = net;
end

[~,tr] = train(net,inputs,targets);

target = targets(1,:);
trTarg = [target(tr.trainInd);1-target(tr.trainInd)];
```

```
tsTarg = [target(tr.testInd);1-target(tr.testInd)];

output = zeros(1,length(target));
for k = 1:15
net = nets{k};
outputs = net(inputs);
output = output + round(outputs(1,:));
end

output = sign(output/15 - 0.5)/2 + 0.5;
% hard voting
trOut = [output(tr.trainInd); 1-output(tr.trainInd)];
tsOut = [output(tr.testInd); 1-output(tr.testInd)];

[trErrorRate(l),~,~,~] = confusion(trTarg,trOut);
[tsErrorRate(l),~,~,~] = confusion(tsTarg,tsOut);
end
trMeanErrorRate(i,j) = mean(trErrorRate);
trStdErrorRate(i,j) = std(trErrorRate);
tsMeanErrorRate(i,j) = mean(tsErrorRate);
tsStdErrorRate(i,j) = std(tsErrorRate);
end
end
toc
xvalues = {'2', '8', '32'};
yvalues = {'32','16','8'};
figure(2)
ax = gca;
ax.FontSize = 12;
set(findall(gcf,'type','line'),'linewidth',2);
subplot(2,2,1)
h = heatmap(xvalues,yvalues,flipud(trMeanErrorRate));
h.XLabel = 'Hidden Nodes';
h.YLabel = 'Epochs';
h.Title = 'TrainingErrorRateMean';
subplot(2,2,2)
h = heatmap(xvalues,yvalues,flipud(tsMeanErrorRate));
h.XLabel = 'Hidden Nodes';
h.YLabel = 'Epochs';
h.Title = 'TestingErrorRateMean';
subplot(2,2,3)
h = heatmap(xvalues,yvalues,flipud(trStdErrorRate));
h.XLabel = 'Hidden Nodes';
h.YLabel = 'Epochs';
h.Title = 'TrainingErrorRateStd';
subplot(2,2,4)
h = heatmap(xvalues,yvalues,flipud(tsStdErrorRate));
h.XLabel = 'Hidden Nodes';
h.YLabel = 'Epochs';
h.Title = 'TestingErrorRateStd';



trMeanErrorRate = zeros(length(trainFcn),
length(hiddenLayerSize)*length(nEpochs));
```

```
trStdErrorRate = zeros(length(trainFcn),
length(hiddenLayerSize)*length(nEpochs));
tsMeanErrorRate = zeros(length(trainFcn),
length(hiddenLayerSize)*length(nEpochs));
tsStdErrorRate = zeros(length(trainFcn),
length(hiddenLayerSize)*length(nEpochs));
elapsedTime = zeros(length(trainFcn),
length(hiddenLayerSize)*length(nEpochs));

for s = 1:length(trainFcn)
for i = 1:length(hiddenLayerSize)
for j = 1:length(nEpochs)
tic
trErrorRate = zeros(1,loop);
tsErrorRate = zeros(1,loop);

net = patternnet(hiddenLayerSize(i), trainFcn(s), performFcn);
net.trainParam.epochs =nEpochs(j);

for l=1:loop
nets = {};
net.divideParam.trainRatio = 0.5;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 0.5;
for k = 1:15
net = init(net);
[net,~] = train(net,inputs,targets);
nets{k} = net;
end

[~,tr] = train(net,inputs,targets);

target = targets(1,:);
trTarg = [target(tr.trainInd);1-target(tr.trainInd)];
tsTarg = [target(tr.testInd);1-target(tr.testInd)];

output = zeros(1,length(target));
for k = 1:15
net = nets{k};
outputs = net(inputs);
output = output + round(outputs(1,:));
end

output = sign(output/15 - 0.5)/2 + 0.5;
% hard voting
trOut = [output(tr.trainInd); 1-output(tr.trainInd)];
tsOut = [output(tr.testInd); 1-output(tr.testInd)];

[trErrorRate(l),~,~,~] = confusion(trTarg,trOut);
[tsErrorRate(l),~,~,~] = confusion(tsTarg,tsOut);
end
trMeanErrorRate(s, 3*i-3+j) = mean(trErrorRate);
trStdErrorRate(s, 3*i-3+j) = std(trErrorRate);
tsMeanErrorRate(s, 3*i-3+j) = mean(tsErrorRate);
tsStdErrorRate(s, 3*i-3+j) = std(tsErrorRate);
elapsedTime(s, 3*i-3+j) = toc;
```

```
end
end
end
figure(3)
for i = 1:9
subplot(3,3,i)
X =
categorical({'trMeanErrorRate','tsMeanErrorRate','trStdErrorRa
te','tsStdErrorRate', 'elapsedTime'});
Y =
[trMeanErrorRate(:,i),tsMeanErrorRate(:,i),trStdErrorRate(:,i)
, tsStdErrorRate(:,i), elapsedTime(:,i)/15*30/10]';
bar(X,Y)
ylim([0, 0.06])
t = string(hiddenLayerSize(ceil(i/3))) + ' Nodes '+
string(nEpochs(i-ceil(i/3)*3+3)) + ' Epochs';
title(t)
end
```

A3. Gaussian_nn.m

```
clear
rng('default')  % For reproducibility
R1 = mvnrnd([1 0],[1 0; 0 1],3300/2);
R2 = mvnrnd([2 0],[4 0; 0 4],3300/2);

inputs = [R1;R2]';
targets = [ones(size(R1,1),1), zeros(size(R1,1),1);
zeros(size(R2,1),1), ones(size(R2,1),1)]';

trainFcn = 'trainscg';
performFcn = 'crossentropy';

% distribution
figure(1)
plot(R1(:,1),R1(:,2),'+')
hold on
plot(R2(:,1),R2(:,2),'x')
axis equal
grid on

% bayes optimal
u = linspace(-1.5, 3, 450);
v = linspace(-2.5, 3.5, 600);
z_optimal = zeros(length(v), length(u));
for i = 1:length(u)
    for j = 1:length(v)
          if  ln_gaussian_2d([1;0], [1 0; 0 1], [u(i);
v(j)])>ln_gaussian_2d([2;0], [4 0; 0 4], [u(i); v(j)])
            z_optimal(j,i) = 1;
          end
    end
end
```

```matlab
figure(1)
hold on
contour(u,v,z_optimal,[0,1],'y', 'LineWidth', 2)

u = linspace(-1.5, 3, 90);
v = linspace(-2.5, 3.5, 120);
z = zeros(length(v), length(u));

% nn
% net = patternnet(8, "trainscg",performFcn);
% net.trainParam.epochs =32;
% net.divideParam.trainRatio = 300/3300;
% net.divideParam.valRatio = 0;
% net.divideParam.testRatio = 3000/3300;
% net = init(net);
% [net,~] = train(net,inputs,targets);
% for i = 1:length(u)
%     for j = 1:length(v)
%           output = round(net([u(i); v(j)]));
%           z(j,i) = output(1);
%     end
% end

% nn_voting
net = patternnet(8, "trainscg",performFcn);
net.trainParam.epochs =32;
nets = {};
net.divideParam.trainRatio = 300/3300;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 3000/3300;
for j = 1:8
net = init(net);
[net,~] = train(net,inputs,targets);
nets{j} = net;
end

for i = 1:length(u)
    for j = 1:length(v)
        output = 0;
        for k = 1:8
            net = nets{k};
            outputs = net([u(i); v(j)]);
            output = output + round(outputs(1,:));
        end
        z(j,i) = sign(output/8 - 0.5)/2 + 0.5;
    end
end

figure(1)
hold on
contour(u,v,z,[0,1],'r', 'LineWidth', 2)
legend("Class 1", "Class 2", "Bayes Optimal","Nerual Net")
axis([-6,15,-8,7])
```