CSE301 Bio-Computation

Assessment 3

Vehicle Logo Classification with MLP and RBF

Name: Wilbert Osmond
Student ID: 1926308

Major: Exchange (non-UoL)
University: Xi'an Jiaotong-Liverpool University (XJTLU)
Region: Suzhou, Jiangsu, China Mainland

Period: Fall 2019
Date: 20/12/2019

1. Introduction

In this project, two different classification algorithms, multilayer perceptron (MLP) and radial-basis function network (RBFN), are implemented using MATLAB to classify vehicle logo. Several parameters are experimented for MLP (i.e. the number of hidden units, learning rate, and momentum) and RBFN (the number of RBF centers). Based on several experiments on the different parameters in both models, the best MLP model and the best RBFN model are then compared using confusion matrix and accuracy. Furthermore, both MLP and RBFN models' convergence performance using MSE for each epoch in the training are also compared.

This project unites our lecture work with that of our laboratory tutorials, showcasing our skills in both supervised algorithms. Through MATLAB coding of MLP and RBFN, this project allows us for a full understanding of the step-by-step of each concept. This project displays the importance of the relevant concepts and proper MATLAB coding, as well as improves critical analysis skills of the students completing it.

2. Methodology

I used the dataset (logo.mat) provided along with the assessment, which contains 117 samples of 5 different types of vehicle logo images, along with corresponding class labels (1-5) and 80 features in each sample extracted by appropriate feature extraction algorithm. For both MLP and RBFN models, I divided the dataset 80% into a training set and 20% into a testing set.

An MLP is a feedforward artificial network that is composed of more than one perceptron. It is particularly characterized by its multiple layers, consisting of at least three layers of nodes: an input layer, a hidden layer, and an output layer. Each node is a neuron that utilizes a nonlinear activation function (in this project, we used the sigmoid activation function), excluding the input nodes. MLP uses a supervised learning technique called backpropagation for training. With its multiple layers and nonlinear activation, MLP can distinguish complex data that is not linearly separable, which is very useful to classify our logo dataset.

To run the MLP design, modelMLP.m can be run. Creating the MLP can be achieved easily by using the feedforwardnet() function with training function 'traingdm'. As explained in MathWorks Documentation (2019), 'traingdm' updates weight and bias values according to gradient descent with momentum backpropagation. This training function allows us to manually change the parameters of the learning rate and momentum, so we can manually experiment their different adjustments to compare their different effects on the network's accuracy later. Training the MLP can be done easily by feeding in the training data and targets into the train() function which returns the trained network. The trained network can then be simulated with the testing data to generate the output for cross-validation and accuracy measurement.

An RBF network (RBFN) is an artificial neural network that uses radial basis function as activation functions. For a classification task as in this project, RBFN performs classification by measuring the input's similarity to examples from the training set (McCormick, 2013). RBFN are usually known to consist of three layers: an input layer, a hidden layer containing RBF neurons with a non-linear RBF activation function and a linear output layer. In this project, the RBF neurons are the centers, and the output layer consists of five nodes which represents each of the five category targets. The RBF learning

is two-phase, which is a very common learning scheme (Schwenker, 2000), such that two layers of an RBFN are learned separately. The first training phase is the adaptation of centers and scaling parameters (i.e. the variances in this project). These may be trained along with the output weights, but this takes a long time (Bishop, 1995), hence they are determined in advance. In this project, RBF centers are trained by k-means clustering, as it places the RBF centers only in regions of the input space where there are significant amounts of examples. After the centers c_i have been fixed, the output layer is trained by supervised learning, wherein the optimal weights that minimize the error at the output can be computed with a linear pseudoinverse solution (Wikipedia, 2019). An important input to the linear pseudoinverse solution is the kernel matrix, which is obtained from Gaussian basis functions. Computing a linear multiplication of the weight matrix and the kernel matrix gives the outputs, which are then tasked to classify our vehicle logo dataset.

To run the MLP design, modelRBFN.m can be run, which will call a separate function from kernelmat.m. Training and simulating the RBFN cannot be as straightforward as using the usual functions for RBFN newrb() and sim() in this project. This is because with newrb() we cannot manually input the number of centers generated from k-means clustering, as we want to experiment different number of centers and compare their effects in this task. Instead, the centers have to be first generated using the kmeans() function and then I manually coded compute kernel matrix. Other things that had to be manually coded were the pseudo inverse solution to obtain the optimal output weights as well as the linear multiplication of the kernel matrix and weight matrix to generate the output values.

3. Experimental Results and Analysis

3.1. Experiment Procedures

For both MLP and RBFN designs, the procedure contains three main steps. The first step is data pre-processing, in which the process is the same for both MLP and RBFN designs. The second step is creating and training the network, which process differs for MLP and RBFN designs. The final step is to evaluate the neural network performance, by testing each network with their different testing processes and then use the same process to create each of their confusion matrix and subsequently also their accuracy.

3.1.1. Step 1: Data pre-processing

The first step of our experiment is pre-processing the data in which we prepare the inputs and outputs matrices. We start by loading the dataset logo.mat, which contains eohsamples (features) and eohlabels (targets). We then define eohsamples from logo.mat as the input matrix X, but we extract only features 1-64 and remove 65-80. This is because all inputs in features 65-80 are null for all samples, so removing it can increase computational efficiency without losing any information, as input = 0 will contribute nothing to the computation. Subsequent to defining the data features for all samples as input, we normalize the data so all features can be uniformly evaluated. After that, we define the eohlabels from logo.mat as the target vector T. After getting the input and target matrices, we split both input and target data into 80% training and 20% testing by using cross-validation.

3.1.2. Step 2: Training the neural network

3.1.2.1. Training the MLP Model

The second step of the MLP design starts with defining the network architecture as a feedforwardnet with one layer of *n* hidden units and a training function of 'traingdm'. As explained in MathWorks Documentation (2019), 'Traingdm' is a network training function that updates weight and bias values according to gradient descent with momentum backpropagation. This training function allows us to manually change the parameters of the learning rate and momentum, which we can manually adjust to compare their different effects on the network's accuracy later. Finally, we train the neural network given the inputs and targets, and returns the network, training record, and the output vector Ytrain.

3.1.2.2. Training the RBFN Model

There are two training phases in the second step of the RBFN design. The first training phase is the adaptation of centers and scaling parameters (i.e. the variances in this project). The centers are obtained by k-means clustering and the number of centers can be adjusted. The variances are obtained from the squared summed distances from each point to every centroid divided by the number of training points. The kernel matrix for training PHItrain is then computed using kernelmat() function that I created, given the input of the training data, centers, and variances. The kernelmat() function essentially computes for the phi matrix in which each element φ_{ij} is a Gaussian basis function $\exp\left(\frac{||x_i - c_j||}{2\sigma_j^2}\right)$ for i = index of sample, j = index of center, x = sample, c = center, σ = variance; with an added column of 1s for the bias term.

The second training phase is the training of output weights by using the pseudo inverse solution for each target category, where we make output values binary – 1 for samples that belong to the same category as the output node, and '0' for all samples. For instance, we are learning the weights for output node 1, then all target category 1 samples are labeled as '1' and all category 2-5 samples are labeled as 0.

3.1.3. Step 3: Evaluate the neural network performance

The final step of the MLP design starts by simulating the neural network with the 20% test data, for cross-validation and ensuring generalizability. We then round the test output elements to {1,2,3,4,5} as the target values, to enable the computation of the confusion matrix later. We round output values lower than 1 to 1, higher than 5 to 5, and everything else in between to the nearest integer. On the other hand, the final step of the RBFN design starts by computing the kernel matrix for testing given the 20% test data, centers and variance. We then simulate the network process from hidden layer to output layer by multiplying the weight matrix obtained from training and the kernel matrix for testing. The output value of each sample is the maximum score from all the five target categories.

The next step for both MLP and RBFN is to create a confusion matrix for each model. From the confusion matrix, we can compute the accuracy by summing the diagonal elements of the confusion matrix (i.e. the true positive for every target label) and divide it with the length of the target (i.e. 23).

3.2. Results discussion and analysis

3.2.1. MLP Results

For MLP, we experiment with different number of hidden units, learning rate, and momentum constant. As shown in Figure 3.1, increasing the number of hidden units (especially above 20 (MathWorks, 2019)) can improve results. When we increase the number of hidden units from 5 to 30, we can see the significant increase in accuracy but also an increase in the number of epochs needed to converge to minimum. More neurons in the hidden layer provide the network with more flexibility, as there are more parameters that the network can optimize. The flexibility allows for more accurate classifications as well as being able to run for more epochs before reaching (lower) minimum convergence, but more parameters to compute also lead to longer time elapsed. However, if we make the hidden layer too large, for example to 60 hidden units, we might cause the problem to be under-characterized and the network must optimize more parameters than there are data vectors to constrain these parameters.

Hidden Units = 5			Hidden Units = 30			Hidden Units = 60		
#	Accuracy	Epochs	#	accuracy	Epochs	#	Accuracy	Epochs
1	0.9231	685	1	0.8462	248	1	0.8718	250
2	0.9487	1000	2	0.9573	1000	2	0.8803	170
3	0.5983	63	3	0.9487	1000	3	0.906	568
4	0.9402	1000	4	0.9658	1000	4	0.8974	822
5	0.7863	204	5	0.9231	1000	5	0.9145	696
6	0.7265	99	6	0.6239	19	6	0.4274	24
7	0.4786	18	7	0.8889	581	7	0.9316	348
8	0.9145	1000	8	0.8632	1000	8	0.9316	1000
9	0.1709	12	9	0.8974	491	9	0.9402	555
10	0.3077	13	10	0.812	122	10	0.9487	1000
Average	0.67948	409.4	Average	0.87265	646.1	Average	0.86495	543.3

Figure 3.1: Effects of different numbers of hidden units, with learning rate 0.01 and momentum constant 0.6.

The selection of a learning rate is of critical importance in finding the true global minimum of the error distance. As shown in Figure 3.2, when we start with learning rate 0.01, we observe a huge average of 409.4 epochs. Indeed, backpropagation training with too small learning will make slow progress. When we increase the learning rate to 0.1, the average epochs decrease drastically to 40.7, as the network learns faster, although it slightly reduces its accuracy. When we increase the learning rate further to 0.5, the average epochs decrease again to 6.1, but this time the accuracy drastically drop to 0.18. Too large a learning rate will proceed faster but may simply produce oscillations between relatively poor solutions. Therefore, higher learning rate increases minimum epochs taken to reach minimum but decreases accuracy.

Learning Rate = 0.01			Learning Rate = 0.1			Learning Rate = 0.5		
#	Accuracy	Epochs	#	Accuracy	Epochs	#	Accuracy	Epochs
1	0.9231	685	1	0.2393	26	1	0.1966	6
2	0.9487	1000	2	0.1966	17	2	0.1453	6
3	0.5983	63	3	0.8547	38	3	0.1111	6
4	0.9402	1000	4	0.9145	98	4	0.2308	6
5	0.7863	204	5	0.8889	52	5	0.2393	6
6	0.7265	99	6	0.5983	30	6	0.1709	6
7	0.4786	18	7	0.7692	90	7	0.1368	6

8	0.9145	1000	8	0.641	22	8	0.1282	6
9	0.1709	12	9	0.1966	21	9	0.2137	7
10	0.3077	13	10	0.2906	13	10	0.2564	6
Average	0.67948	409.4	Average	0.55897	40.7	Average	0.18291	6.1

Figure 3.2: Effects of different learning rates, with 5 hidden units and momentum constant 0.6.

As shown in Figure 3.3, when the momentum constant is very small such as 0.05, we observe a high average accuracy of 0.93, but it takes very long to converge with average epochs of 800.6 (or more, since we use early stopping with maximum epochs of 1000). Such very small momentum constant would make the training similar to the regular backpropagation, where the weight update is determined purely by the gradient descent. When we increase the momentum constant to a moderate 0.6, we observe a significant decrease in average epochs of 646.1, while the accuracy slightly decreases to 0.87. Increase the momentum smoothen the weight changes and suppresses cross-stitching. When all weight changes are all in the same direction, the momentum amplifies the learning rate so that it converges faster, as we notice the decrease in average epochs when we increase the momentum constant. However, when we adjust the momentum constant to be too large such as 0.95, we notice that even though the average epochs also drastically decrease to 8.2, the accuracy also drastically drops to 0.20, rendering our classification algorithm ineffective. In the case where the momentum constant is too large, the gradient descent is completely ignored, and the update is mostly based on only the momentum.

Momentum Constant = 0.05			Momentum Constant = 0.6			Momentum Constant = 0.95		
	Accuracy	Epochs		Accuracy	Epochs		Accuracy	Epochs
1	0.9402	1000	1	0.8462	248	1	0.1026	11
2	0.906	1000	2	0.9573	1000	2	0.3248	6
3	0.9658	1000	3	0.9487	1000	3	0.1624	11
4	0.9316	1000	4	0.9658	1000	4	0.094	6
5	0.9487	1000	5	0.9231	1000	5	0.2222	6
6	0.9744	1000	6	0.6239	19	6	0.1282	6
7	0.8547	164	7	0.8889	581	7	0.1966	6
8	0.9573	1000	8	0.8632	1000	8	0.1026	6
9	0.9145	524	9	0.8974	491	9	0.2906	12
10	0.8889	318	10	0.812	122	10	0.359	12
Average	0.92821	800.6	Average	0.87265	646.1	Average	0.1983	8.2

Figure 3.3: Effects of different momentum constants, with 30 hidden units and learning rate 0.01.

3.2.2. RBFN Results

For RBFN, we experimented with different number of centers (i.e. RBF neurons in the hidden layer). As shown in Figure 3.4, even when we start with 5 centers, the network already gives a high average accuracy of 0.93. As the centers are equivalent to hidden units, then increasing the number of centers also provides the network with more flexibility, as there are more parameters that the network can optimize and thus reach a higher accuracy. This is evident when we increase the number of centers to 10, the network gives an average accuracy of 0.97 that is even higher. Increasing the number of centers further to 15, however, does not significantly change the accuracies. An important thing to note is that for all three chosen numbers of centers, we notice quite frequent of a perfect accuracy of 1, even for centers = 5. It might be plausible to say that the possibility of getting such perfect accuracy is high when there is small testing data (23 samples). Nevertheless, depending on the task, choosing 5 centers might be sufficient for classification with high accuracy.

Centers = 5		Centers = 10		Centers = 15	
	Accuracy		Accuracy		Accuracy
1	0.8261	1	1	1	0.913
2	0.9565	2	1	2	0.9565
3	1	3	0.913	3	1
4	0.8696	4	1	4	0.9565
5	1	5	1	5	1
6	1	6	0.9565	6	0.9565
7	0.913	7	0.9565	7	0.9565
8	0.8261	8	1	8	0.913
9	1	9	1	9	1
10	0.913	10	0.8696	10	1
Average	0.93043	Average	0.96956	Average	0.9652

Figure 3.4: Effects of different momentum constants, with 30 hidden units and learning rate 0.01.

3.3. Performance Comparison: MLP vs RBFN

As mentioned earlier, the RBF centers are equivalent to hidden units. In MLP, 5 hidden units gives only 0.68 average accuracy; whereas in RBFN, 5 RBF centers already give 0.93 average accuracy. With the seemingly best number of hidden units (=30), the best learning rate (=0.01), and the best momentum constant (=0.05), the MLP model can reach 0.93 average accuracy but with quite large epochs of more than 800 (see Figure 3.3). On the other hand, RBFN with a significantly smaller number of hidden units (i.e. centers) of 5 also reach 0.93 average accuracy, and with a drastically smaller epoch of 1 (due to the usage of the direct computation of optimal weights through the pseudo inverse solution). A disclaimer, however, is that each epoch in RBFN takes longer than in MLP, given that everything else is constant. This is because MLP only use the same activation function of sigmoid in the hidden nodes, whereas RBFN has different activation functions (Gaussian parameterized by different centers and variances) to compute at each hidden node.

Note that through the different test runs, the accuracy for RBFN is consistently high (above 0.8), unlike MLP which can have fluctuating accuracies. An explanation for this might be that RBFN uses the pseudoinverse solution (in this project) which can find optimal weights. The existence of such solution provides RBF networks an explicit minimizer (when the centers are fixed). This is unlike MLP that uses gradient descent which needs to rely on different initialization of weights, some of which might be bad seeds.

Through different experiments as discussed in Sections 3.2.1 and 3.2.2, I decided that the best model (that achieves the highest accuracy) for MLP is hidden units = 30, learning rate = 0.01, momentum constant = 0.05; whereas for RBFN is centers = 10. Figure 3.5 shows the best MLP model's (0.9565 accuracy) convergence performance using MSE for each epoch in the training process. The best validation performance is MSE = 0.26215 at epoch 217. On the other hand, for RBFN, the mean squared error (MSE) can be coded manually by comparing the test targets and the output targets, such as `MSEtrain = immse(Ttrain',Ytrain).` For a direct comparison, I computed the MSE of an RBFN that also yields 0.9565 accuracy (same as the best MLP model) is 0.0106, which is much lower than the best MLP model's minimum MSE.

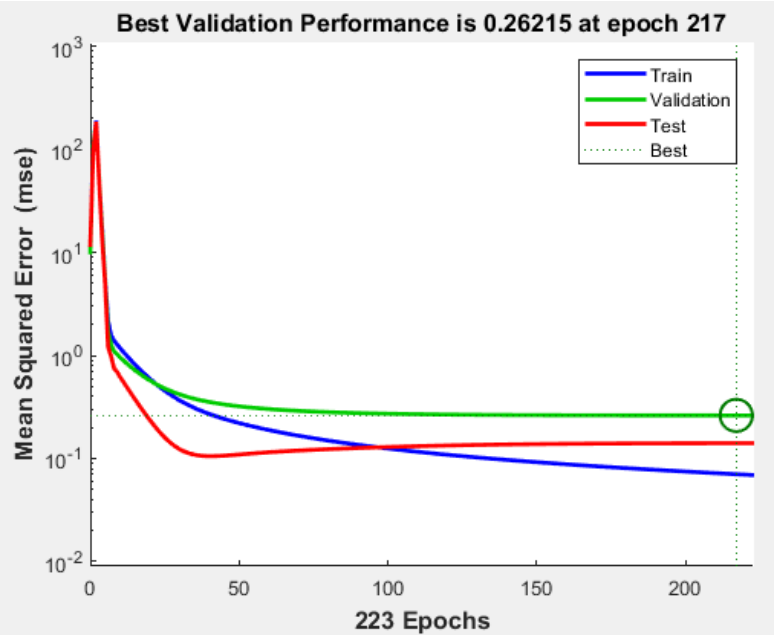

Figure 3.5: Best MLP model's convergence performance using MSE for each epoch in the training process.

To further show RBFN's superiority over MLP, there are three things to note. First, an RBFN with 0.9565 accuracy is not even the best RBFN model. The RBFN model can reach a perfect accuracy of 1 at its best (see Figure 3.6), in which case the MSE may be even lower. Second, such low MSE for RBFN is obtained only through one training iteration, since the optimal weights are directly computed one time using the pseudo inverse solution. Last, as explained by Schwenker et al. (2000), the performance of RBF classifiers trained with two-phase learning can optionally be improved through a third backpropagation-like training phase, adapting the whole set of parameters (i.e. RBF centers, scaling parameters, and output layer weights) simultaneously. Such three-phase learning RBF networks further gives a practical advantage by enabling the possibility to use unlabelled training data for the first training phase.

Best MLP Model	Best RBFN Model
C = 2 1 0 0 0 0 5 0 0 0 0 0 4 0 0 0 0 0 3 0 0 0 0 0 8 Accuracy = 0.9565	C = 3 0 0 0 0 0 4 0 0 0 0 0 4 0 0 0 0 0 4 0 0 0 0 0 8 Accuracy = 1 MSEtrain = 0

Figure 6: Confusion matrix and accuracy of the best MLP model and the best RBFN model. The diagonal elements in the confusion matrix are the true positives for every target category. Accuracy is computed by summing the diagonal elements divided by the total elements (i.e. number of samples).

4. Conclusion

The MLP and the RBFN are both layered feedforward networks that produce nonlinear function mappings. In this project, both the MLP and RBFN designs have one hidden layer, although MLP uses the same sigmoid activation function whereas RBFN uses different Gaussian activation functions parameterized by different centers and variances. Due to their activations, information is represented locally in RBFN, whereas information is represented globally in MLP. MLP is trained by backpropagation training algorithm, whereas RBF is trained by the pseudo inverse solution using the kernel matrix. Several parameters of MLP (i.e. number of hidden units, learning rate, momentum) and RBFN (i.e. number of RBF centers) are experimented. For MLP, more hidden units, smaller learning rate, and smaller momentum contribute to higher accuracy, but longer time taken to converge. For RBFN, more RBF centers, which are initialized from k-means clustering, contribute to higher accuracy.

Between MLP and RBFN, RBFN's performance is significantly superior than MLP's. With the pseudo inverse solution, RBFN can directly compute the optimum weights and converge to minimum MSE in one training 'epoch'. The best model (that achieves the highest accuracy) for MLP with a maximum accuracy of 96% and minimum MSE of 0.26215 in epochs 800+ is hidden units = 30, learning rate = 0.01, momentum constant = 0.05; whereas for RBFN with a maximum accuracy of 100% and minimum MSE of 1 in epoch 1 is centers = 10. RBFN can achieve a higher accuracy faster (smaller epochs) – due to training using the pseudo inverse solution – with smaller number of hidden units (i.e. the RBF centers). In fact, it is also possible to further improve RBFN's performance by adding an optional backpropagation training phase in RBFN. In this project, however, that is not necessary, because even with a small number of neurons, the RBFN model can already reach a high accuracy.

5. References

Bishop,C. (1995). *Neural Networks for Pattern Recognition, 164-190*. Oxford, UK: Oxford University Press.

MathWorks. (2019). MathWorks Documentation.

McCormick, C. (2013). Radial Basis Function Network (RBFN) Tutorial. Retrieved from https://mccormickml.com/2013/08/15/radial-basis-function-network-rbfn-tutorial/

McCormick, C. (2014). RBF Network MATLAB Code. Retrieved https://chrisjmccormick.wordpress.com/2013/08/16/rbf-network-matlab-code/

Schwenker, F., Kestler, H.A., Palm, G. (2001). Three learning phases for radial-basis-function networks.
Neural Networks 14(4-5), 439-458.

Wikipedia. (2019). Radial basis function network. Retrieved from https://en.wikipedia.org/wiki/Radial_basis_function_network/