University of Surrey
Department of Electronic Engineering
MSc Computer Vision, Robotics and Machine Learning

EEEM005 AI and AI Programming Coursework
NN Matlab simulation for 'Understanding how to solve pattern recognition problems using backpropagation'

By
Kongkea Ouch
6664607

Submitted to
Dr. Terry Windeatt
8th May 2021

Executive Summary

Many sophisticated problems in our world are made simple by employing the neural power of Multilayer Perceptron (MLP) [1]. This work explores multiple experiments of MLP using backpropagation algorithm to solve pattern recognition of breast cancer in MATLAB program. In the first experiment, multiple base classifiers of a wide range of node and epoch combinations yield the optimal combination of 8 nodes and 8 epochs. In the second experiment, it is found that the use of an ensemble of classifiers always outperforms the use of optimal base classifiers from the first experiment and that the ideal node and epoch combination of ensemble classifier is not relevant to that of the first experiment due to variation in majority pooled results from multiple classifiers of an ensemble. It is also of interest that the ideal size of an ensemble is proven to follow a known rule of thumb to an extent. In the third experiment, it is discovered that the use of the 'Resilient Backpropagation' optimiser results in the greatest prediction accuracy and most efficient computational usage but not necessarily the fastest. The 'Levenberg-Marquardt' optimiser makes the training quickest and still gives decent test performance, while the 'Scaled Conjugate Gradient' gives a well-rounded overall result in terms of speed, efficiency and accuracy. The remaining future works include exploring other optimisers and datasets as well as differentiating between two equiprobable classes of 'overlapping' two-dimensional Gaussians [2].

Introduction

As stated by Raj et al. [1], Multilayer Perceptron (MLP) is a good starting point into the world of deep learning as it has important real-world use cases such as pattern classification, prediction and approximation [1]. An MLP is a form of feedforward artificial neural network that is consisted of one or more layers of artificial perceptron or neurons mimicking human's brain neurons. They are made up of three kinds of layers: an input layer that takes in the input information for other layers process, a number of hidden layers that compute and derive meaning from the input, and at last an output layer makes the final call to predict or classify based on the processed information [1].

The backpropagation algorithm is used to train the MLP's neurons by updating and correcting the weights and biases of the neurons based on the backwards propagated errors through the network, which are calculated using many performance functions depending on the training context, such as mean squared error and cross-entropy [4]. To optimize the backpropagation algorithm, many network optimisers can be used depending on the problem such as Levenberg-Marquardt, Scaled Conjugate Gradient and Resilient Backpropagation [4]. On top of this, a pooling of multiple classifiers based on a majority vote is also a renowned approach in implementing MLP to get the best performance [1].

Three experiments will be performed in MATLAB with their built-in breast cancer dataset to explore the usage of MLP in binary classification of predicting severity of breast cancer. First, individual classifier will be investigated by training over multiple combination of nodes and epochs and observing how each combination affects the error rate of training and testing. After this observation, an optimal node and epoch combination will be reported and selected for the use in second experiment. For the second, the concept of ensemble of classifiers will be implemented by majority vote and random starting weight. The best node and epochs from experiment will be used to train multiple ensembles of different sizes and the results here will be compared to the performance of the first experiment. Other node and epoch combinations will also be explored for various ensembles in order to observe if there is any improvement in performance. Last but not least, in the third experiment, it will be a replication of the second experiment but with different training optimisers to identify the most suitable one for this study's problem and report how each optimiser works. Figures and tables included in the content are mostly summarized, so please refer to the appendix for detailed results.

Experiment 1

In this experiment, we aim to find the ideal test error rate and its related node/epoch combination based on training of multiple pattern recognition feedforward neural networks that solve breast cancer detection problem that aims to classify inputs of the cancer to targets classes of serious (1) or not serious (0).

In order to achieve this, we train the networks using MATLAB cancer dataset, 'trainscg' optimiser and 'crossentropy' loss function over 21 node/epoch combinations of nodes = [2 8 32] and epochs = [1 2 4 8 16 32 64] based on the assignment's requirement [2]. The dataset is randomly divided into 50% samples for training and 50% samples for testing [2]. Each of the 21 node/epoch combinations is run 30 times with different 50/50 split to minimize outputs' variation and obtain a robust overall result by averaging over the 30 iterations [2]. The minimum gradient of the networks is set to zero so that each training is allowed to take place until each desired number of epochs is reached. Then, the outcome of train/test error rates and standard deviations of all combinations are plotted on their respective graphs which will allow us to spot the optimal node/epoch values as well as the relevant test error rate.

On top of this, having run one iteration of experiment 1, we continue to repeat two more iterations of experiment 1 to investigate and capture the variations of best node/epoch combinations and associated test errors. The best combination that occurs the majority of times of the 3 iterations of experiment 1 shall be decided as the optimal one that shall be used in the next experiment.

Below are results from all 3 iterations of experiment 1. Iteration is represented by letter ‘i’ and standard deviation is denoted by ‘std’ in each figure’s title. From left to right, they are listed in order of iterations 1, 2 and 3. From top to bottom, they are listed in order of test error rate, train error rate, test error rate’s standard deviation and train error rate’s standard deviation.

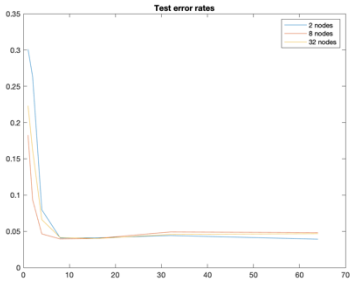


Figure 1: Test error, i = 1

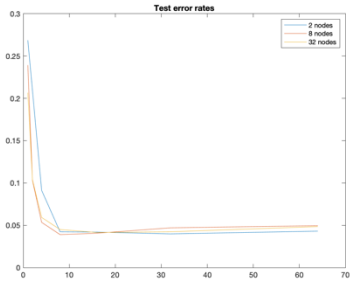


Figure 2: Test error, i = 2

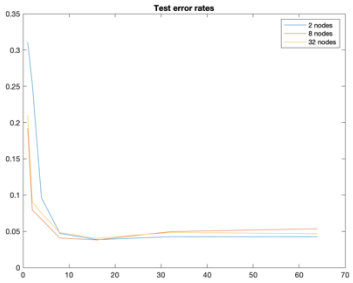


Figure 3: Test error, i = 3

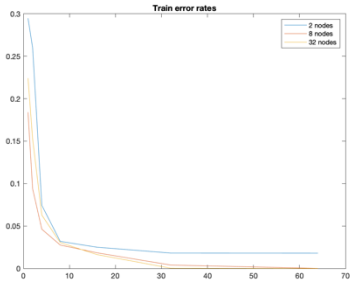


Figure 4: Train error, i = 1

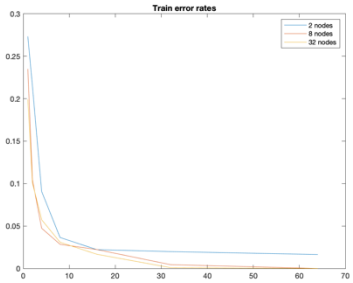


Figure 5: Train error, i = 2

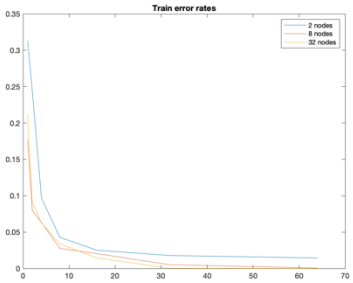


Figure 6: Train error, i = 3

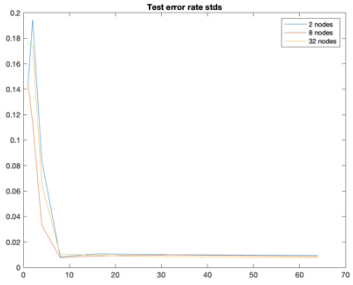


Figure 7: Test error std, i = 1

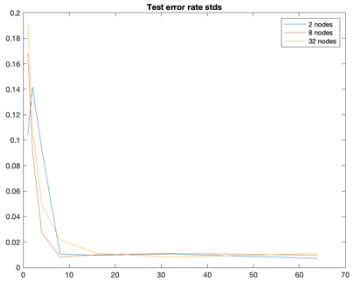


Figure 8: Test error std, i = 2

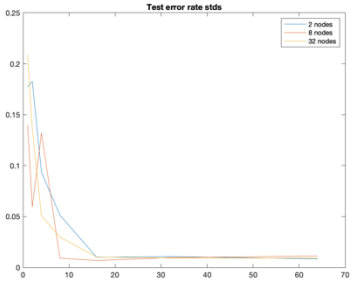
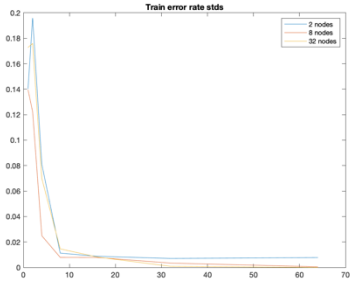
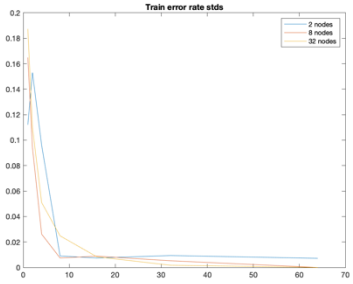


Figure 9: Test error std, i = 3





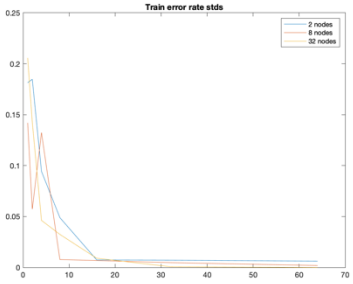


Figure 10: Train error std, i = 1 *Figure 11: Train error std, i = 2* *Figure 12: Train error std, i = 3*

Test error rate							
Node/Epoch	**1**	**2**	**4**	**8**	**16**	**32**	**64**
2	0.301	0.2633	0.0796	0.0408	0.041	0.0442	0.0391
8	0.1828	0.0933	0.0464	**0.0394**	0.0402	0.0491	0.0482
32	0.2231	0.1618	0.0661	0.0413	0.0401	0.0457	0.0468

Table 1: Test error, i = 1

Test error rate							
Node/Epoch	**1**	**2**	**4**	**8**	**16**	**32**	**64**
2	0.2687	0.2094	0.0912	0.0423	0.0419	0.0398	0.043
8	0.2392	0.103	0.0534	**0.039**	0.0406	0.047	0.0495
32	0.2067	0.1038	0.0592	0.045	0.0414	0.0423	0.0485

Table 2: Test error, i = 2

Test error rate							
Node/Epoch	**1**	**2**	**4**	**8**	**16**	**32**	**64**
2	0.3107	0.2533	0.0962	0.0471	0.0385	0.0426	0.0425
8	0.1922	0.0791	0.0666	0.0408	**0.0381**	0.0495	0.0535
32	0.21	0.0908	0.0749	0.0482	0.0403	0.0485	0.0467

Table 3: Test error, i = 3

In term of test error rate, combination of 8 nodes and 8 epochs obviously outperforms other possible combinations most of the time as observed throughout the three iterations from Figure 1 to 3. Even combination of 8 nodes and 16 epochs in iteration 3 is lower than the aforementioned combination according to Figure 3 and Table 3, the 8 nodes/16 epochs combination suffers from higher standard deviation of test error rates compared to the 8 node/8 epoch combination during the 30 iterations run signifying that the 8 node/8 epoch combination is more prone to variation and hence still the champion of ideal test error rate here.

As for training error rate, the networks converge very well with both 8 and 32 nodes except with the later giving better loss a bit sooner but not much of significance considering test error rate best measures generalization of a model against unseen data and that the use of 8 nodes continues to perform excellently against variation in iteration 3's train error standard deviation of Figure 12.

Hence, across all 3 iterations of experiment 1, the best node/epoch combination is 8/8 as it generally results in a point where test error rate bottoms out and starts going up from there indicating that this is the suitable place to stop training before overfitting ruins the model. Moreover, what is noticeable is the result of 8 nodes yielding best performance satisfies the rule of thumb by Heaton [3] that '(1) the number of nodes should be between the input layer's size and the output layer's size (9 features < 8 nodes < 2 class labels), (2) the number of nodes should be 2/3 of the input layer's size added with the output layer's size (6 features + 2 class labels = 8 nodes) and (3) the number of nodes should be less than double the input layer's size (8 nodes < 18 features)'. The number of epochs on the other hand is not known to be reliably dependent on a priori assumption like the number of nodes. It requires trials and errors to see at what epoch the test error rate would become minimal and stop improving so that the model can generalize well to test data and avoid learning unnecessary noises that might degrade network's performance. Overall, this 8 node/8 epoch combination provides the optimal number of nodes and epochs which we will use later to conduct experiment 2.

Experiment 2

We have obtained the best value of nodes and epochs from experiment 1 which was executed over various individual classifiers of different node/epoch combinations. Now, we are going to test this optimal value of nodes and epochs on ensemble of individual classifiers initialized with random weights and with their classification results combined together by Majority Vote method to compare with the performance of experiment 1's best individual classifier of 8 nodes and 8 epochs [2].

The difference between majority vote approach with experiment 1 is that we use only a single base classifier in previous experiment and over here an odd number of multiple base classifiers are utilized and the dominating majority of predicted outputs across all base classifiers will be computed and pooled into the outputs of an ensemble classifier instead. After getting the majority vote ensemble, we run it 30 times to get an average with unique 50/50 split of test/set samples like we did with experiment 1 except this time we have to make sure the ensemble's consisting classifiers must share the same test/set split and training settings while differing in starting weights. We shall test on many diverse sizes of classifier ensemble which range from 3 to 25 with an increment of 2 and observe how each ensemble performs against the experiment 1's individual classifier.

Once we have understood the outcome of using 8 nodes and 8 epochs for different ensembles, we can proceed to leave the optimal combination behind and play around with other nodes and epochs combinations in case they offer better results similarly to how we did in experiment 1. With this trial, it is possible to expect a new champion of node/epoch combination as a result of the majority vote impact.

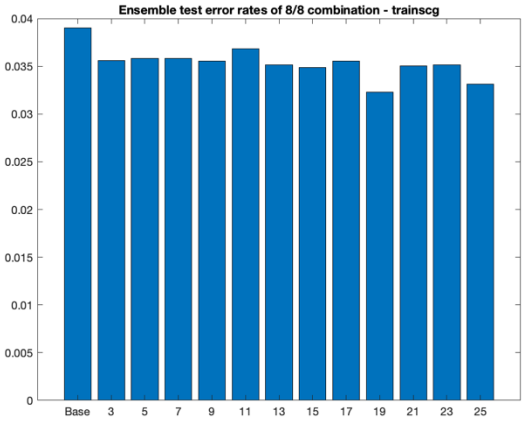


Figure 13: Ensemble test error rates of 8/8 combination – trainscg

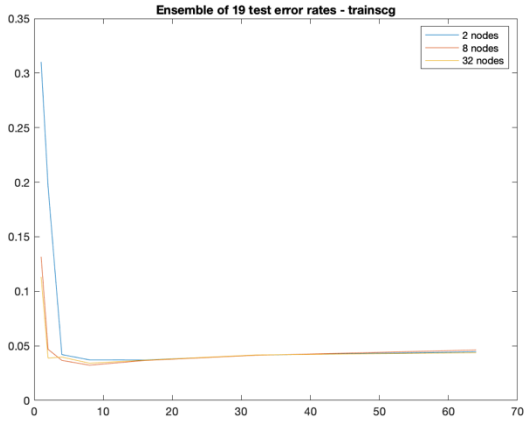


Figure 14: Ensemble of 19 test error rates

Based on result from Figure 13, all the ensembles using the optimal 8/8 combination perform better than the best combination itself from experiment 1 with test error rate of 3.8% taken from Table 2 and incorporating into above figure as base classifier. We can also see from Figure 14 that the ensemble of 19 classifiers which has the lowest test error rate of 3.23% based on Figure 13 trains and generalizes best to unseen test data at the optimal 8/8 combination of base classifier.

Further combinations of node/epoch are tested over 3 to 25 ensembles to get a bigger picture of what is happening with the best test error rate and optimal node/epoch combination for each ensemble in Table 4. For detailed individual graphs of these combinations plotted similar style to Figure 14, please refer to the appendix.

Ensemble	3	5	7	9	11	13	15	17	19	21	23	25
Test error rate	0.0356	0.0348	0.0333	0.0352	0.0339	0.0338	0.0349	0.0329	**0.0323**	0.0341	0.033	0.0331
Node	**8**	2	2	2	8	8	**8**	32	**8**	32	32	**8**
Epoch	**8**	16	16	8	4	16	**8**	8	**8**	8	8	**8**

Table 4: Ensemble's optimal test error rate and associated node/epoch combination - trainscg

According to Table 4, the optimal node/epoch combination keeps changing between ensembles with only four ensembles (3, 15, 19 and 25) sharing the same ideal nodes and epochs as individual classifier while the other ensembles vary in node/epoch combinations. Unlike the base classifier which does not rely on majority vote, the ensemble makes use of this to reduce the variation in the prediction performance of its member classifiers hence improve the overall accuracy. However, in some cases, it is possible that its performance can suffer from having a member classifier dragging down the average performance of the ensemble system or having a strongly performing base classifier unable to demonstrate its ability because majority vote spread overs its performance with other classifiers to provide an overall robust performance. Combined with the fact that the assembling classifiers use random starting weights, this might be impacting the choice of optimal nodes and epochs comparing to individual classifier as there is more classifiers to take into account and hence no guarantee that the best performance across all ensembles always depend on the same nodes and epochs. It all depends on the problem set we are looking to tackle.

Experiment 3

We further repeat experiment 2 with two other optimisers namely 'trainlm' and 'trainrp' to find and compare ensemble's optimal test error rate and related node/epoch combination [2]. These two optimisers seem to perform differently to the 'trainscg' we dominantly used in experiment 1 and 2.

Ensemble	3	5	7	9	11	13	15	17	19	21	23	25
Test error rate	0.0373	0.0364	0.0345	0.0367	0.0337	0.035	0.034	0.0349	0.0338	0.0337	0.0335	**0.0333**
Node	8	8	8	32	8	32	32	8	32	8	32	**8**
Epoch	2	2	2	1	2	1	1	2	1	2	1	**2**

Table 5: Ensemble's optimal test error rate and associated node/epoch combination – trainlm

Ensemble	3	5	7	9	11	13	15	17	19	21	23	25
Test error rate	0.0352	0.0356	0.0346	0.0338	0.0348	0.0333	0.033	0.0351	0.0345	0.0322	0.0332	**0.032**
Node	8	2	32	2	2	2	2	2	2	2	8	**2**
Epoch	16	16	8	32	16	16	16	64	16	16	8	**16**

Table 6: Ensemble's optimal test error rate and associated node/epoch combination – trainrp

Unlike 'trainscg' optimiser which used only 19 classifiers (Table 4), both 'tranilm' and 'trainrp' took up to 25 classifiers to reach the least test error rate. In term of speed, trainlm performs the fastest mostly between 1 and 2 epochs while trainrp takes the most time varying between 8 to 64 epochs. Hence, the best choice for this problem of binarily classifying the severity of breast cancer is depended on the available resource at hand and the extent to which the researcher can compromise between prediction's accuracy and the training speed. If both optimal test error rate and computational saving are prioritised, which is most ideal for this cancer classification problem, then 'trainrp' optimiser is the obvious solution delivering an astounding 3.2% test error rate (Table 6) at the cost of using only 2 nodes and 16 epochs. If researcher wants the fastest training time and a compromise in performance, then 'trainlm' optimiser is recommended as it generally takes only between 1 to 2 epochs while still offering a decent test error rate of approximate 3.3% (Table 5). On the other hand, 'trainscg' optimiser is probably the best middle ground between speed, resource and accuracy giving optimal test error rate of 3.23% which is still close to 3.2% of 'trainrp'. Next, we will explore how these optimisers work and the extent to which their aforementioned experimental performance fares with the literature.

The 'trainlm' [5] is an optimiser used in MATLAB that adjusts weight and bias during training based on Levenberg-Marquardt (LM) algorithm. Even it is not fitted for pattern recognition problems, the LM

algorithm is known for its speed in backpropagation as proven by its minimal use of epochs in Table 5, hence it is often the go-to for a supervised algorithm especially for function fitting problems [4]. According to Marquardt [6], it aims to solve a nonlinear least square minimization problem but suffers from being stuck with a local minimum which is not ideal for training performance. To offset this, the LM algorithm [6] combines second order information from neural network and gradient information from dynamically switching between the Gauss-Newton (GN) algorithm and the Vanilla Gradient Descent (VGD) algorithm. Therefore, the error function will always be decreased at each step of the algorithm hence improving the training performance.

The 'trainscg' [7] is an optimiser used in MATLAB that adjusts weight and bias during training based on the scaled conjugate gradient (SCG) algorithm. Like the LM algorithm, the SCG algorithm, created by Møller [8], uses second order information from neural network but it is built in mind to alleviate the problem of exhaustive line search. It is capable of solving complex problems effectively because it uses different conjugate gradient methods that do not need a line search at each iteration [8]. The strength in performing in complex problem is realized in Table 4 as it delivers a better optimal test error rate than that of 'trainlm' optimiser even it consumes more time.

The 'trainrp' [9] is an optimiser used in MATLAB that adjusts weight and bias during training based on the resilient backpropagation algorithm (RPROP). The RPROP algorithm was developed out of necessity by Riedmiller and Braun [10] to deal with back propagation converging slowly with network's weights stuck around local optima due to the use of classical gradient-descent. It improves over the traditional back propagation by locally modifying the weight updates based on how the error function behaves and converges generally faster than other algorithms [10]. Hence, it is an ideal choice for pattern recognition problem which requires high computational power to train and the lowest error rate due to sensitivity of the problem. This is proven by its outstanding performance of optimal error rate at a fraction of computational workload as observed in Table 6.

Conclusion

The real-world benefits of MLP are undeniable [1]. This study explores the implementation of backpropagation to train MLP in various training contexts to address the problem of pattern recognition of breast cancer in MATLAB program. It is observed from the experiments that ensemble classifier tends to provide better performance than individual classifier in term of accuracy but at the cost of using more time and computational power than the former. The optimal node and epoch combination of individual classifier does not always translate into that of ensemble classifier as a result of the majority vote's mechanism

increasing the diversity of the ensemble's results. An unexpected but welcoming finding is that this study's optimal ensemble's size adheres to a known rule of thumb to some extent. As for training optimization, the 'Resilient Backpropagation' function outperforms other candidates by giving the best test performance and consuming the least memory. The second-best optimiser is the 'Scaled Conjugate Gradient' optimiser serves a jack of all trades (speed, efficiency and accuracy). The last one is 'Levenberg-Marquardt' optimiser aces the training speed while still giving an acceptable test accuracy, but its strong performance is when applied on function fitting problems not pattern recognition problems [4]. In the real world, the choice of using an ensemble and specific training optimiser depends on a researcher's available resource, need and situation. More works can be explored, including implementing this study using other optimisers and datasets and differentiating between two equiprobable classes of 'overlapping' two-dimensional Gaussians [2].

References

[1] P. Raj and P. Evangeline, *The digital twin paradigm for smarter systems and environments: the industry use cases*. Cambridge, MA ; San Diego, CA ; Kidlington, Oxford ; London: Elsevier, AP, Academic Press, an imprint of Elsevier, 2020.

[2] T. Windeatt, " Assignment 1) NN matlab simulation for 'Understanding how to solve pattern recognition problems using backpropagation.'" 2021.

[3] J. Heaton, *Introduction to Neural Networks for Java*, 2nd ed. Heaton Research, Inc, 2008.

[4] *Train and Apply Multilayer Shallow Neural Networks - MATLAB & Simulink*. [Online]. Available: https://www.mathworks.com/help/deeplearning/ug/train-and-apply-multilayer-neural-networks.html.

[5] *Levenberg-Marquardt backpropagation - MATLAB*. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/trainlm.html.

[6] Marquardt, D., "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *SIAM Journal on Applied Mathematics*, Vol. 11, No. 2, June 1963, pp. 431–441.

[7] *Scaled conjugate gradient backpropagation - MATLAB*. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/trainscg.html.

[8] M. F. Møller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, no. 4, pp. 525–533, 1993.

[9] *Resilient backpropagation - MATLAB*. [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/trainrp.html.

[10] Riedmiller, M., and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*,1993, pp. 586–591.

Appendix 1: Exp 1 Code

```
close all
clear
clc

load cancer_dataset

[x,t] = cancer_dataset;

tind = vec2ind(t);

epochs = [1 2 4 8 16 32 64];
nodes = [2 8 32];

combo_itrs = 30;
exp_itrs = 3;

trainFcn = 'trainscg';

l_nodes = length(nodes);
l_epochs = length(epochs);

s_test_errs = zeros(l_nodes,l_epochs,exp_itrs);
s_train_errs = zeros(l_nodes,l_epochs,exp_itrs);
s_test_err_stds = zeros(l_nodes,l_epochs,exp_itrs);
s_train_err_stds = zeros(l_nodes,l_epochs,exp_itrs);

for s = 1:exp_itrs
    train_errs = zeros(l_nodes, l_epochs);
    train_err_stds = zeros(l_nodes, l_epochs);

    test_errs = zeros(l_nodes, l_epochs);
    test_err_stds = zeros(l_nodes, l_epochs);

    for eInd = 1:l_epochs

        for nInd = 1:l_nodes

            itr_train_errs = zeros(1,combo_itrs);
            itr_test_errs = zeros(1,combo_itrs);

            for i = 1:combo_itrs
                net = patternnet(nodes(nInd), trainFcn);
                net.trainParam.epochs = epochs(eInd);
                net.trainParam.min_grad = 0;
                net.input.processFcns = {'removeconstantrows','mapminmax'};
                net.divideFcn = 'dividerand';
                net.divideMode = 'sample';
                net.divideParam.trainRatio = 50/100;
                net.divideParam.valRatio = 0/100;
                net.divideParam.testRatio = 50/100;
                net.performFcn = 'crossentropy';
```

```
                net.plotFcns = {'plotperform','plottrainstate','ploterrhist',
...
                    'plotconfusion', 'plotroc'};

                [net,tr] = train(net,x,t);
                % a = tr
                y = net(x);
                yind = vec2ind(y);

                percenterrs = sum(tind ~= yind)/numel(tind);
                testPercenterrs = sum(tind(tr.testInd) ~=
yind(tr.testInd))/numel(tind(tr.testInd));
                trainPercenterrs = sum(tind(tr.trainInd) ~=
yind(tr.trainInd))/numel(tind(tr.trainInd));

                itr_train_errs(1,i) = trainPercenterrs;
                itr_test_errs(1,i) = testPercenterrs;


            end

            test_errs(nInd,eInd) = mean(itr_test_errs);
            test_err_stds(nInd,eInd) = std(itr_test_errs);

            train_errs(nInd,eInd) = mean(itr_train_errs);
            train_err_stds(nInd,eInd) = std(itr_train_errs);

        end

    end

    s_test_errs(:,:,s) = test_errs;
    s_train_errs(:,:,s) = train_errs;
    s_test_err_stds(:,:,s) = test_err_stds;
    s_train_err_stds(:,:,s) = train_err_stds;

    figure(1)

    plot(epochs, test_errs)
    title('Test err rates')
    legend('2 nodes','8 nodes','32 nodes')
    saveas(gcf,['test_' num2str(s) '.png'])


    figure(2)

    plot(epochs, train_errs)
    title('Train err rates')
    legend('2 nodes','8 nodes','32 nodes')
    saveas(gcf,['train_' num2str(s) '.png'])

    figure(3)

    plot(epochs, test_err_stds)
    title('Test err rate stds')
```

```
    legend('2 nodes','8 nodes','32 nodes')
    saveas(gcf,['test_std_' num2str(s) '.png'])


    figure(4)

    plot(epochs, train_err_stds)
    title('Train err rate stds')
    legend('2 nodes','8 nodes','32 nodes')
    saveas(gcf,['train_std_' num2str(s) '.png'])


end
```

Appendix 2: Exp 2 and 3 Code

```
clear
close all
clc

load cancer_dataset

[x,t] = cancer_dataset;

epochs = [1 2 4 8 16 32 64];
nodes = [2 8 32];
classifiers = 3:2:25;
itrs = 30;
trainFcns = {'trainscg','trainlm','trainrp'};
performFcns = {'crossentropy','mse','crossentropy'};
c = 0;
tfInd = 3;

tind = vec2ind(t);
len_tind = length(t);
len_epochs = length(epochs);
len_nodes = length(nodes);
len_classifiers = length(classifiers);
trainFcn = trainFcns{tfInd};
performFcn = performFcns{tfInd};
total_itrs = sum(classifiers)*itrs*len_nodes*len_epochs;

ind_train_errs = zeros(len_nodes,len_epochs,len_classifiers);
ind_test_errs = zeros(len_nodes,len_epochs,len_classifiers);

ens_train_errs = zeros(len_nodes,len_epochs,len_classifiers);
ens_test_errs = zeros(len_nodes,len_epochs,len_classifiers);

for nInd = 1:len_nodes

    net = patternnet(nodes(nInd), trainFcn);
    net.performFcn = performFcn;
    net.trainParam.min_grad = 0;
    net.input.processFcns = {'removeconstantrows','mapminmax'};
    net.plotFcns = {'plotperform','plottrainstate','ploterrhist', ...
        'plotconfusion', 'plotroc'};

    for eInd = 1:len_epochs

        net.trainParam.epochs = epochs(eInd);

        for cInd = 1:len_classifiers
            itr_ind_train_errs = zeros(classifiers(cInd),itrs);
            itr_ind_test_errs = zeros(classifiers(cInd),itrs);

            itr_ens_train_errs = zeros(1,itrs);
            itr_ens_test_errs = zeros(1,itrs);

            for iInd = 1:itrs
```

```
                [trainInd,valInd,testInd] = dividerand(len_tind,0.5,0,0.5);
                net.divideFcn = 'divideind';
                net.divideMode = 'sample';
                net.divideParam.trainInd = trainInd;
                net.divideParam.valInd = valInd;
                net.divideParam.testInd = testInd;

                yinds = zeros(1,len_tind,classifiers(cInd));

                for i = 1:classifiers(cInd)
                    % Train the Network
                    [net,tr] = train(net,x,t);
                    % Test the Network
                    y = net(x);

                    yind = vec2ind(y);
                    yinds(:,:,i) = yind;

                    testPercenterrs = sum(tind(tr.testInd) ~=
yind(tr.testInd))/numel(tind(tr.testInd));
                    trainPercenterrs = sum(tind(tr.trainInd) ~=
yind(tr.trainInd))/numel(tind(tr.trainInd));

                    itr_ind_train_errs(i,iInd) = trainPercenterrs;
                    itr_ind_test_errs(i,iInd) = testPercenterrs;

                    net = init(net);

                    c = c + 1;
                    progress = c/total_itrs*100;
                    disp([trainFcns{tfInd} ' e' num2str(epochs(eInd)) ' n'
num2str(nodes(nInd)) ' i' num2str(iInd) ' ens' num2str(classifiers(cInd)) '_'
num2str(i)])
                    disp([num2str(progress) ' ' num2str(c)])

                end

                ens_yind = zeros(1,len_tind);
                limit = floor(classifiers(cInd)/2);

                for i = 1:len_tind
                    isTrue = sum(yinds(:,i,:) == 1) > limit;

                    if (isTrue)
                        ens_yind(i) = 1;
                    else
                        ens_yind(i) = 2;
                    end
                end

                itr_ens_train_errs(iInd) = sum(tind(tr.trainInd) ~=
ens_yind(tr.trainInd))/numel(tind(tr.trainInd));
                itr_ens_test_errs(iInd) = sum(tind(tr.testInd) ~=
ens_yind(tr.testInd))/numel(tind(tr.testInd));
```

```
            end

            ens_train_errs(nInd,eInd,cInd) = mean(itr_ens_train_errs);
            ens_test_errs(nInd,eInd,cInd) = mean(itr_ens_test_errs);

            ind_train_errs(nInd,eInd,cInd) = mean(itr_ind_train_errs,[1 2]);
            ind_test_errs(nInd,eInd,cInd) = mean(itr_ind_test_errs,[1 2]);


        end

    end

end



%
% legend_labels = cellstr(append(string(classifiers), ' classifiers'));
%
% for tfInd = 1:len_trainFcns
%     for cInd = 1:len_classifiers
%
%         plot(epochs,ens_test_errs(:,:,cInd))
%         title(['Test err rates - ' num2str(epochs(eInd)) ' classifiers - '
trainFcns(tfInd)])
%         legend(legend_labels)
%         saveas(gcf,['Test_err_rates_' num2str(classifiers(cInd))
'_classifiers_' trainFcns(tfInd) '.png'])
%
%         plot(epochs,ens_train_errs(:,:,cInd))
%         title(['Train err rates - ' num2str(classifiers(cInd)) '
classifiers - ' trainFcns(tfInd)])
%         legend(legend_labels)
%         saveas(gcf,['Train_err_rates_' num2str(classifiers(cInd))
'_classifiers_' trainFcns(tfInd) '.png'])
%
%     end
% end
```

Appendix 3: MATLAB Code of Plotting Playground

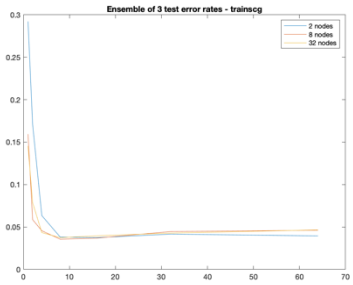
```
epoch = [1 2 4 8 16 32 64];
node = [2 8 32];

ensemble = 3:2:25;

trainFcn = 'trainrp';



% figure(1)
%
% y = reshape(ens_test_errors(2,4,:),[1 12]);
%
% bar([0.039 y])
% b.FaceColor = 'flat';
% b.CData(1,:) = 222;
% set(gca,
'XTickLabel',{'Base','3','5','7','9','11','13','15','17','19','21','23','25'}
)
%
% title('Ensemble test error rates of 8/8 combination - trainscg')
% saveas(gcf,'ensemble_test_error_8_8.png')

best_loss = zeros(3,12);


for eInd = 1:length(ensemble)
    [M,I] = min(ens_test_errors(:,:,eInd),[],'all','linear');

    best_loss(1,eInd) = M;

    [a,b] = find(ens_test_errors(:,:,eInd) == M)
    best_loss(2,eInd) = node(a);
    best_loss(3,eInd) = epoch(b);


%     figure(1)
%
%     plot(epochs, ens_test_errors(:,:,eInd))
%     title(['Ensemble of ' num2str(ensemble(eInd)) ' test error rates - '
trainFcn])
%     legend('2 nodes','8 nodes','32 nodes')
%     saveas(gcf,[trainFcn '_ensemble_test_' num2str(ensemble(eInd)) '.png'])
%
%     figure(2)
%
%     plot(epochs, ens_train_errors(:,:,eInd))
%     title(['Ensemble of ' num2str(ensemble(eInd)) ' train error rates - '
trainFcn])
%     legend('2 nodes','8 nodes','32 nodes')
%     saveas(gcf,[trainFcn '_ensemble_train_' num2str(ensemble(eInd))
'.png'])
End
```

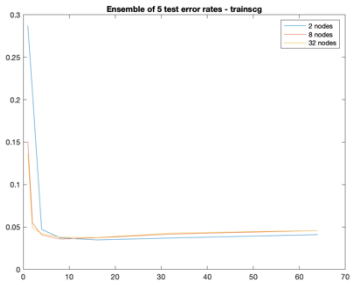
Appendix 4: MATLAB Test Error Raw Output of Experiment 2

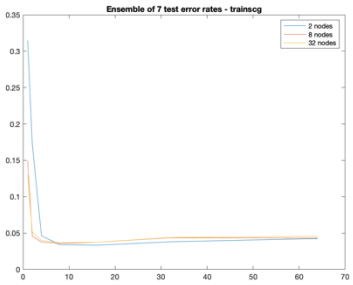
Ensemble 3							
Node/Epoch	1	2	4	8	16	32	64
2	0.2278	0.0804	0.0384	0.0419	0.0408	0.0432	0.0405
8	0.1353	0.0373	0.0403	0.0443	0.0498	0.0474	0.0505
32	0.0681	0.0431	0.0442	0.0449	0.0481	0.0469	0.0467
Ensemble 5							
Node/Epoch	1	2	4	8	16	32	64
2	0.1448	0.081	0.0397	0.0439	0.0437	0.0442	0.0431
8	0.0534	0.0364	0.037	0.043	0.0454	0.0449	0.0442
32	0.0414	0.0417	0.0423	0.0446	0.047	0.0397	0.0471
Ensemble 7							
Node/Epoch	1	2	4	8	16	32	64
2	0.1302	0.0447	0.0382	0.0401	0.0413	0.0429	0.0432
8	0.0499	0.0345	0.0373	0.0435	0.0451	0.0441	0.0408
32	0.0487	0.037	0.0417	0.0463	0.0442	0.0443	0.0413
Ensemble 9							
Node/Epoch	1	2	4	8	16	32	64
2	0.1085	0.049	0.0391	0.0423	0.0394	0.0388	0.0412
8	0.0386	0.038	0.0407	0.0443	0.043	0.0439	0.0425
32	0.0367	0.0391	0.0413	0.0456	0.0454	0.0476	0.0428
Ensemble 11							
Node/Epoch	1	2	4	8	16	32	64
2	0.1201	0.042	0.0363	0.04	0.044	0.0445	0.0415
8	0.0397	0.0337	0.037	0.0451	0.0479	0.0447	0.0437
32	0.0393	0.037	0.0402	0.0474	0.0437	0.0453	0.0447
Ensemble 13							
Node/Epoch	1	2	4	8	16	32	64
2	0.0942	0.053	0.0363	0.0388	0.0402	0.0426	0.0428
8	0.0389	0.0354	0.0378	0.0425	0.0436	0.0442	0.0434
32	0.035	0.0371	0.0421	0.0455	0.0429	0.0427	0.045
Ensemble 15							
Node/Epoch	1	2	4	8	16	32	64
2	0.0888	0.0443	0.0349	0.0411	0.0415	0.0428	0.0383
8	0.0348	0.0355	0.0384	0.0408	0.0437	0.0438	0.0422
32	0.034	0.0379	0.0387	0.0467	0.0457	0.0445	0.0479
Ensemble 17							
Node/Epoch	1	2	4	8	16	32	64
2	0.0964	0.0446	0.0383	0.0411	0.0409	0.0391	0.042

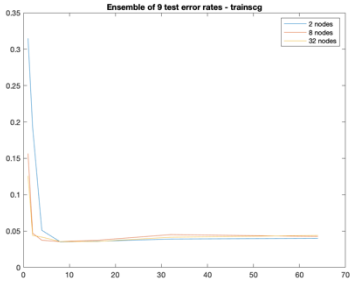
8	0.037	0.0349	0.0392	0.0415	0.0434	0.0414	0.0426
32	0.0355	0.0366	0.0398	0.0456	0.0459	0.0428	0.0439
Ensemble 19							
Node/Epoch	1	2	4	8	16	32	64
2	0.0976	0.0471	0.0381	0.039	0.0385	0.0393	0.0444
8	0.0431	0.0339	0.0383	0.0428	0.0444	0.0435	0.0415
32	0.0338	0.0374	0.042	0.0457	0.0457	0.0445	0.0434
Ensemble 21							
Node/Epoch	1	2	4	8	16	32	64
2	0.091	0.0449	0.0364	0.0395	0.0383	0.0423	0.0399
8	0.0366	0.0337	0.0366	0.0411	0.0451	0.0388	0.0429
32	0.0363	0.0357	0.0412	0.0435	0.0423	0.0452	0.0437
Ensemble 23							
Node/Epoch	1	2	4	8	16	32	64
2	0.0977	0.0433	0.0364	0.0423	0.0412	0.0395	0.0402
8	0.0371	0.0343	0.0388	0.039	0.0392	0.0429	0.0431
32	0.0335	0.035	0.0391	0.0436	0.0437	0.0463	0.0459
Ensemble 25							
Node/Epoch	1	2	4	8	16	32	64
2	0.0874	0.043	0.0347	0.0421	0.0414	0.0414	0.0382
8	0.0353	0.0333	0.0368	0.0392	0.0456	0.0408	0.0447
32	0.0367	0.0337	0.0391	0.043	0.045	0.0459	0.0431

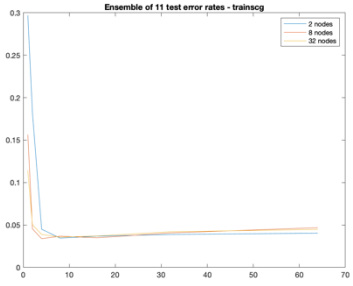
Appendix 5: Extra Figures of Experiment 2

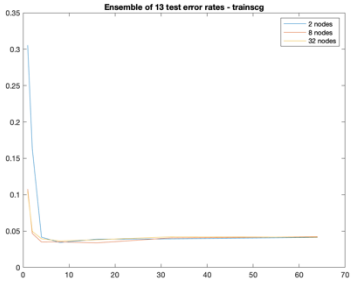


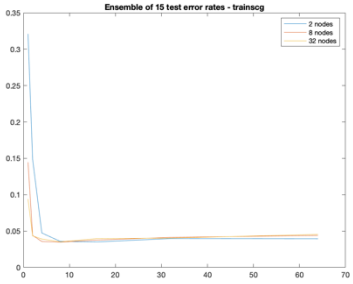


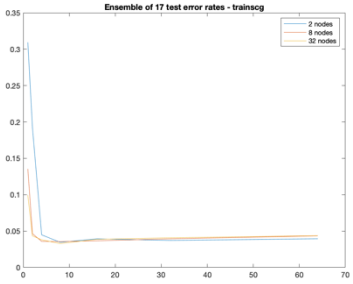


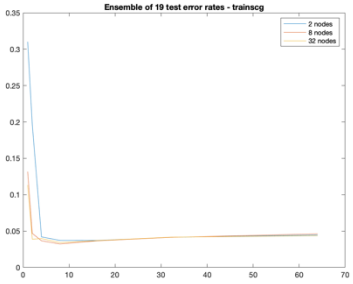


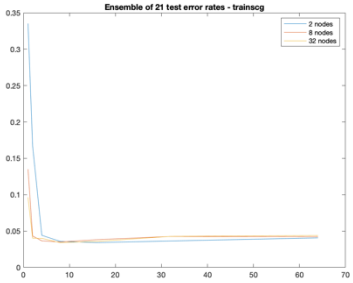


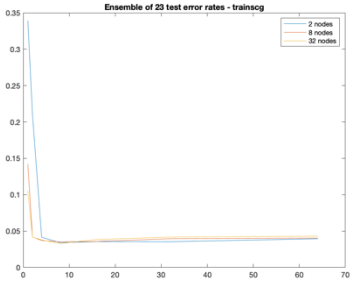


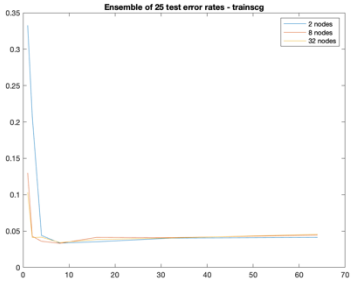


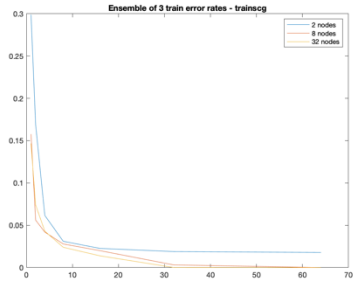


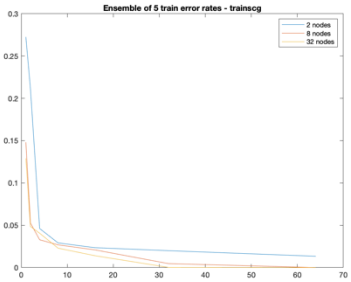


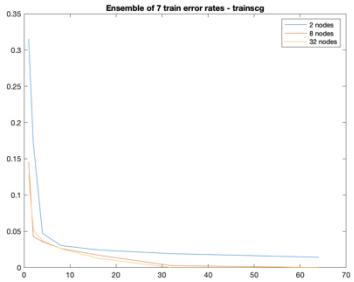


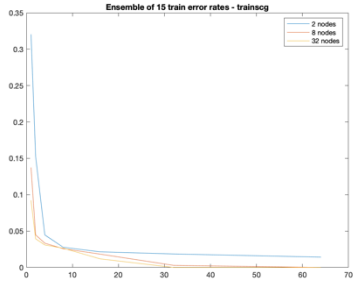


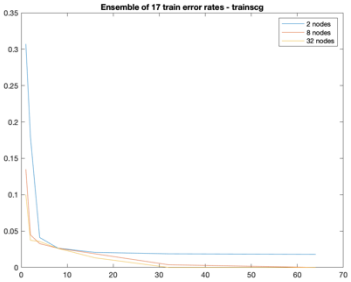


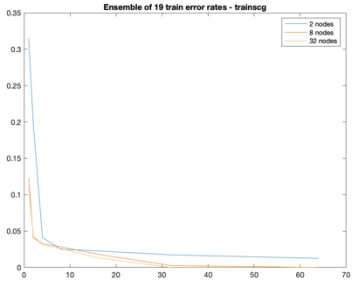

Ensemble of 3 train error rates - trainscg
2 nodes
8 nodes
32 nodes

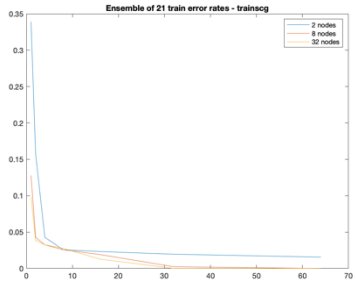

Ensemble of 5 train error rates - trainscg
2 nodes
8 nodes
32 nodes

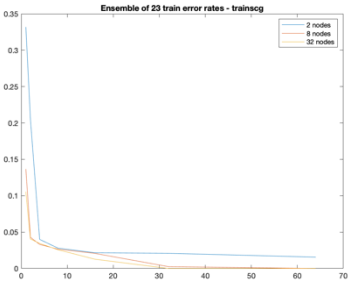

Ensemble of 7 train error rates - trainscg
2 nodes
8 nodes
32 nodes

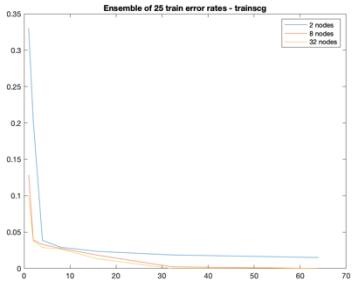

Ensemble of 15 train error rates - trainscg
2 nodes
8 nodes
32 nodes

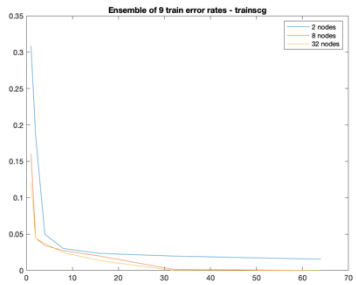

Ensemble of 17 train error rates - trainscg
2 nodes
8 nodes
32 nodes

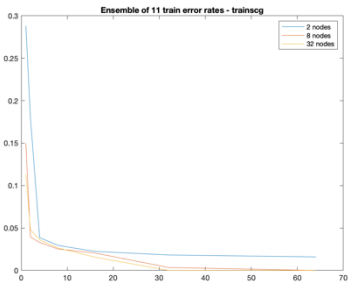

Ensemble of 19 train error rates - trainscg
2 nodes
8 nodes
32 nodes

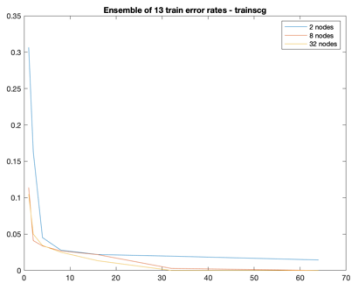

Ensemble of 21 train error rates - trainscg
2 nodes
8 nodes
32 nodes


Ensemble of 23 train error rates - trainscg
2 nodes
8 nodes
32 nodes


Ensemble of 25 train error rates - trainscg
2 nodes
8 nodes
32 nodes


Ensemble of 9 train error rates - trainscg
2 nodes
8 nodes
32 nodes


Ensemble of 11 train error rates - trainscg
2 nodes
8 nodes
32 nodes


Ensemble of 13 train error rates - trainscg
2 nodes
8 nodes
32 nodes

Appendix 6: MATLAB Test Error Raw Output of Experiment 3

Ensemble 3							
Node/Epoch	1	2	4	8	16	32	64
2	0.3754	0.2361	0.126	0.0492	0.0387	0.0382	0.0386
8	0.1771	0.1062	0.0432	0.0377	0.0352	0.0425	0.039
32	0.1209	0.0482	0.0376	0.0386	0.04	0.0391	0.0412
Ensemble 5							
Node/Epoch	1	2	4	8	16	32	64
2	0.3125	0.1967	0.1244	0.04	0.0356	0.0367	0.0403
8	0.175	0.0926	0.0433	0.037	0.038	0.0409	0.0412
32	0.1154	0.0505	0.0379	0.0381	0.0392	0.0412	0.044
Ensemble 7							
Node/Epoch	1	2	4	8	16	32	64
2	0.3139	0.2547	0.1229	0.0411	0.037	0.037	0.0382
8	0.1428	0.0904	0.0405	0.0348	0.0385	0.041	0.0422
32	0.1125	0.0447	0.0352	0.0346	0.0403	0.0403	0.0458
Ensemble 9							
Node/Epoch	1	2	4	8	16	32	64
2	0.2835	0.2156	0.1286	0.0454	0.0371	0.0338	0.0352
8	0.1506	0.0964	0.0391	0.0359	0.0369	0.0398	0.0411
32	0.1134	0.0445	0.0341	0.0347	0.0407	0.0384	0.043
Ensemble 11							
Node/Epoch	1	2	4	8	16	32	64
2	0.3051	0.2144	0.1458	0.0407	0.0348	0.0349	0.0366
8	0.1398	0.0846	0.0387	0.0352	0.037	0.0387	0.0424
32	0.1065	0.0469	0.0367	0.0412	0.039	0.0435	0.0423
Ensemble 13							
Node/Epoch	1	2	4	8	16	32	64
2	0.2694	0.221	0.1331	0.0393	0.0333	0.035	0.0391
8	0.1612	0.0853	0.0356	0.0339	0.0377	0.0412	0.0434
32	0.102	0.0422	0.0348	0.0382	0.0388	0.0385	0.0418
Ensemble 15							
Node/Epoch	1	2	4	8	16	32	64
2	0.2463	0.222	0.1354	0.039	0.033	0.0358	0.0376
8	0.12	0.0797	0.0402	0.0349	0.0372	0.0418	0.041
32	0.1065	0.0438	0.0385	0.0375	0.0393	0.0393	0.0383
Ensemble 17							
Node/Epoch	1	2	4	8	16	32	64

2	0.2516	0.2323	0.1271	0.0414	0.0385	0.0362	0.0351
8	0.1352	0.0921	0.0417	0.0356	0.0376	0.0407	0.0416
32	0.0991	0.0388	0.0355	0.0379	0.0373	0.0432	0.042
Ensemble 19							
Node/Epoch	1	2	4	8	16	32	64
2	0.2468	0.2061	0.1359	0.0393	0.0345	0.0378	0.0381
8	0.1355	0.0779	0.0405	0.0349	0.0349	0.0378	0.0398
32	0.1085	0.0421	0.0366	0.0346	0.041	0.0394	0.0438
Ensemble 21							
Node/Epoch	1	2	4	8	16	32	64
2	0.2341	0.2455	0.1202	0.0414	0.0322	0.0402	0.039
8	0.135	0.0828	0.0386	0.0327	0.0394	0.0387	0.0411
32	0.0903	0.0461	0.0363	0.0369	0.039	0.0425	0.0433
Ensemble 23							
Node/Epoch	1	2	4	8	16	32	64
2	0.275	0.2518	0.11	0.041	0.0373	0.0345	0.0357
8	0.157	0.0737	0.0419	0.0332	0.0368	0.0386	0.0436
32	0.1112	0.045	0.0366	0.0369	0.0378	0.0423	0.0422
Ensemble 25							
Node/Epoch	1	2	4	8	16	32	64
2	0.2637	0.2479	0.1345	0.0407	0.032	0.0358	0.0398
8	0.1222	0.08	0.0375	0.0359	0.0346	0.0385	0.0381
32	0.0939	0.042	0.0355	0.0367	0.0362	0.0429	0.0437