Fall 2017 ME759 Final Project Report
University of Wisconsin-Madison

Parallelizing Neural Network Process

Vinay V. Raikar

December 21, 2017

Abstract

Latest generation automotive engine need fast predictive models for engine calibrations. Critical engine parameters are interdependent for engine performance and emissions. These calibration models are developed using limited test data sets during engine development. The interdependency between variables is determined using Neural Network response surface models which are then used to optimize engine calibrations. Generating the Neural Network models takes significant time as it involves many iterations to find models with level of fidelity required for this work.

In most cases, the code to generate these models is written to execute sequentially. At the same time, most of the latest desktops and laptops are multicore machines whose computing power is underutilized. As Neural Network model development and analysis is an iterative process, multi-core computer architecture can be used for parallel processing this iterative process and reduce computation time. This project focuses on parallel processing the iterative process of building Neural Network using Matlab and its parallel processing tool box to reduce the computation time.

Contents

1. Neural Network (NN) Application Background

1.1 Description

In engine calibration development there is lot of interdependency involved with various engine parameters. For example, variables engine loads, speed, spark timing effects emissions and these are highly nonlinear. Neural Network models are used to analyze the nonlinearity associated with all inputs that affect the output.

Neural Network is a seeding based approach, so hundreds of models need to be developed using known inputs for a particular output(s) both being measured from test. Based on the nonlinearity of the problem there are nodes (hidden layers) which need to be increased to have a good correlation between Neural Network model and the test data.

2. Process Code

2.1.1 Baseline Code

Inputs to the code:

a. X (Input(s)) and Y (Output) both measured data. X data is a 2D array which can range from two to any number of channels. Number of column represents channels and row represents data points for those channels. Y data is a 1D array which is function of all the input channels. Both X and Y data is divided into a training data and validation data
b. Neural Network Node Numbers: This is the number of hidden layers for each set of NN family and it is a 1-D array. Typically this can range from 5 to 15. In this baseline implementation NN models were developed for 5, 6, 7, 8, 9, 10 NN nodes (total 6 nodes).
c. Number of Neural Network models developed per Node. Typically 100 or more.

Main code functions:

a. Training data is used to train the Neural Network models for each node to develop hundreds of model.
b. For each model developed
    - Again training X data is used to simulate Y using the model generated.
    - Validation X data is used to simulate Y using the model generated.
c. For each of the preceding two bullet points RMS Error is calculated and summed.
d. All the hundreds models are compared using RMS Error and select few (fixed 5) are selected.
e. Steps a to d are repeated for each node family
f. An end result is model set for each node. Total say (6 x 5)
g. Plots are generated to compare

All the models are manually analyzed to select the node and its respective family (Not part of the code)

2.1.2 Flow Chart

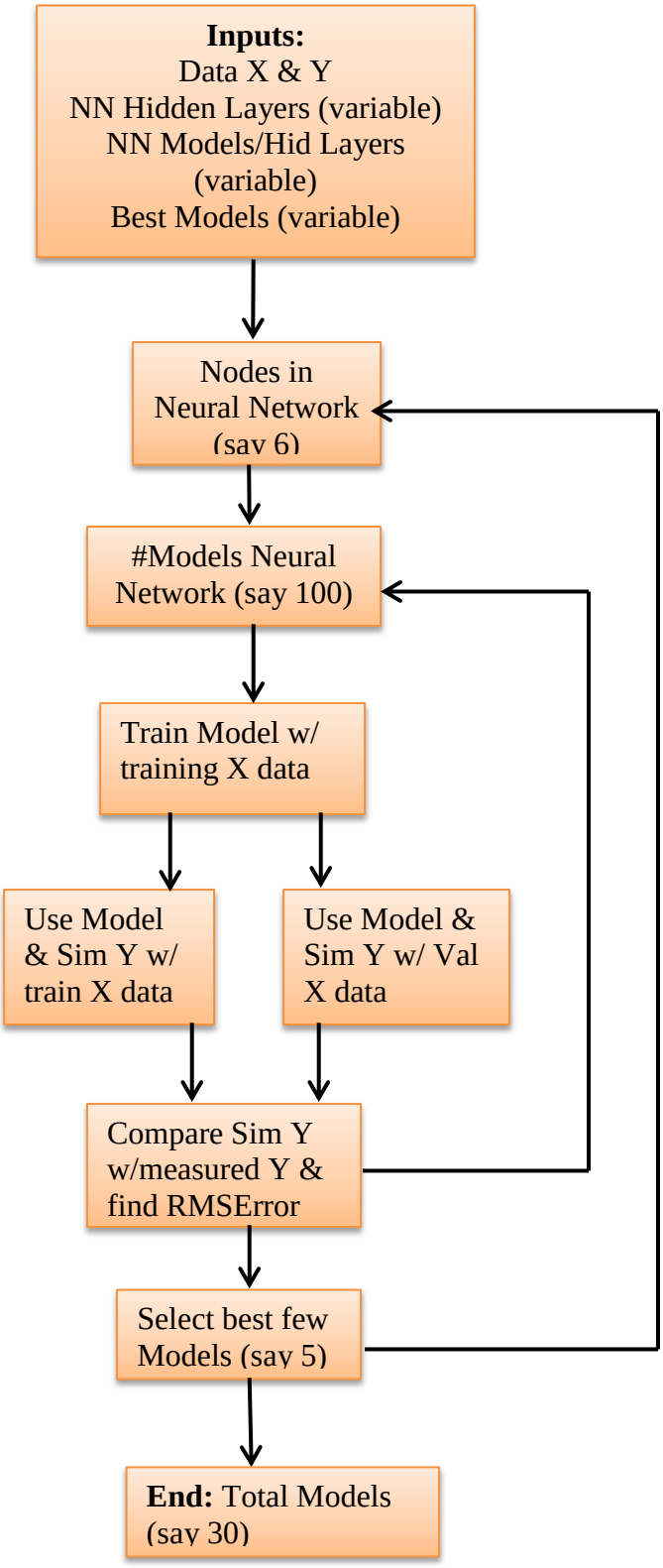
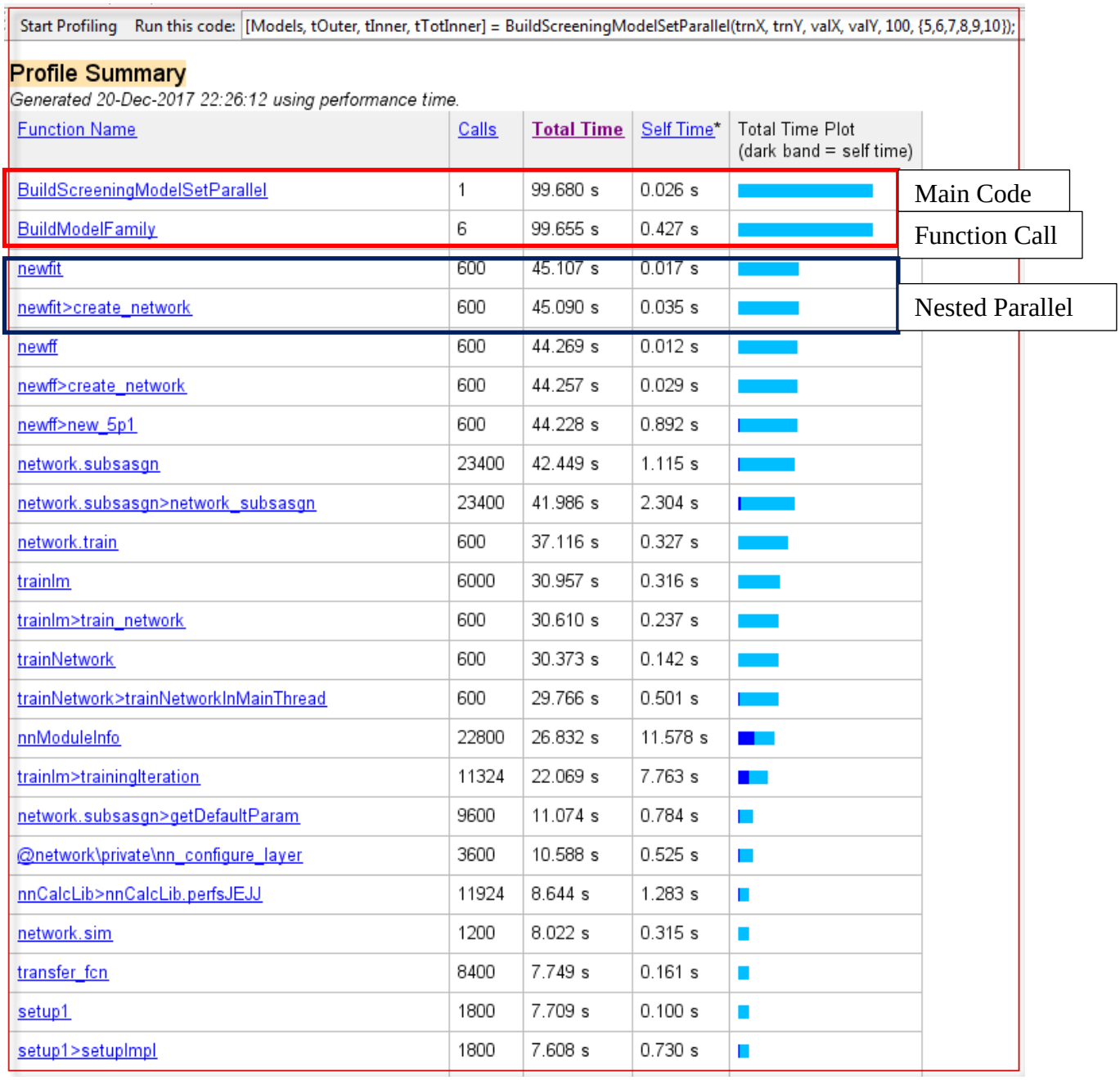


Figure 1. Flow chart for baseline code

2.1.3 Baseline Code Profiling



Profile Summary

Generated 20-Dec-2017 22:26:12 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
BuildScreeningModelSetParallel	1	99.680 s	0.026 s	
BuildModelFamily	6	99.655 s	0.427 s	
newfit	600	45.107 s	0.017 s	
newfit>create_network	600	45.090 s	0.035 s	
newff	600	44.269 s	0.012 s	
newff>create_network	600	44.257 s	0.029 s	
newff>new_5p1	600	44.228 s	0.892 s	
network.subsasgn	23400	42.449 s	1.115 s	
network.subsasgn>network_subsasgn	23400	41.986 s	2.304 s	
network.train	600	37.116 s	0.327 s	
trainlm	6000	30.957 s	0.316 s	
trainlm>train_network	600	30.610 s	0.237 s	
trainNetwork	600	30.373 s	0.142 s	
trainNetwork>trainNetworkInMainThread	600	29.766 s	0.501 s	
nnModuleInfo	22800	26.832 s	11.578 s	
trainlm>trainingIteration	11324	22.069 s	7.763 s	
network.subsasgn>getDefaultParam	9600	11.074 s	0.784 s	
@network\private\nn_configure_layer	3600	10.588 s	0.525 s	
nnCalcLib>nnCalcLib.perfsJEJJ	11924	8.644 s	1.283 s	
network.sim	1200	8.022 s	0.315 s	
transfer_fcn	8400	7.749 s	0.161 s	
setup1	1800	7.709 s	0.100 s	
setup1>setupImpl	1800	7.608 s	0.730 s	

Figure 2. Profile Summary with sequential processing

Above highlighted selection in red is used to parallelize the code to reduce total time in the following sections. Highlight in blue can be parallelized using nested parallel.

2.2.1 Parallel Code

Using Matlab parallel processing tool kit, outer most loop was parallelized, Refer Figure 1. Core count (workers in Matlab environment) was varied to study the performance gains and see the point of diminishing returns. Below snap shot shows the portion of the code.

```
34      %Create a model family for each of the aParms
35 -    tInner = zeros(nFams, 1);
36 -    tStart = tic;
37 -    parpool(4);
38 -    parfor i = 1:nFams
39
40 -        [sMdl, tinner] = BuildModelFamily(trnX, trnY, tstX, tstY, nMdlKeep, TrainAlg, ...
41              'NN', aParms{i}, MaxAttempts);
42 -        tInner(i) = tinner;
43
44 -        ModelParms(:,i) = {sMdl.ModelParms};
45 -        Mdls(:,i) = sMdl.Models(1:nMdlKeep);
46
47 -        trnRMSE(:,i) = sMdl.trnRMSE;
48 -        trnnRMSE(:,i) = sMdl.trnnRMSE;
49 -        valRMSE(:,i) = sMdl.valRMSE;
50 -        tstRMSE(:,i) = sMdl.tstRMSE;
51 -        oaRMSE(:,i) = sMdl.oaRMSE;
52
53 -    end
54
55 -    tOuter = toc(tStart);
56 -    tTotInner = sum(tInner);
57
58 -    TotMdls = nFams * nMdlKeep;
59 -    Models.ModelParms = cell(TotMdls, 1);
60 -    Models.Models = cell(TotMdls, 1);
61 -    Models.trnRMSE = zeros(TotMdls,1);
62 -    Models.trnnRMSE = zeros(TotMdls,1);
63 -    Models.valRMSE = zeros(TotMdls,1);
64 -    Models.tstRMSE = zeros(TotMdls,1);
65 -    Models.oaRMSE = zeros(TotMdls,1);
66
67 -    for i = 1:nFams
68 -        i1 = (i-1) * nMdlKeep + 1;
69 -        i2 = i1 + nMdlKeep - 1;
70 -        Models.Models(i1:i2) = Mdls(:, i);
```

Figure 3. Parallel code snippet

Highlighted blue section shows the use of parallel processing with four cores. In Matlab, "parpool" call opens parallel computing cores, similar to "omp_set_num_threads ()" in OpenMP and "parfor" parallel for loop. Highlighted red section shows the function call to develop NN family for each node and select best few models.

2.2.2 Problem encountered in Parallelization

a. Though the baseline code was tailored such that 'for' loop would work sequentially and in parallel, parallel execution had errors. Last part of the code after the 'for' loop, stores all the models in 1D array for further manual processing (preferred method by separate code where they are analyzed). Matlab did not like the indexing where all the cores stored model in 1-D array, race condition. So a 2D array was created such that each loop stored model in different columns of the array. This resolved the issue. However, 2-D array was converted back to 1-D array outside the outer loop for further processing in a different program.

b. Different tasks of "BuildModelFamily" function call could be further parallelized but implementing nested parallel is not allowed in Matlab. See Figure 6. in the Appendix, this shows profiling for the "BuildModelFamily". There are two tasks that can be parallelized.

2.2.3 Profiling with Parallel (6 cores)

Start Profiling Run this code: [Models, tOuter, tInner, tTotInner] = BuildScreeningModelSetParallel(trnX, trnY, valX, valY, 100, {5,6,7,8,9,10});

Profile Summary

Generated 20-Dec-2017 22:18:42 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
BuildScreeningModelSetParallel	1	31.785 s	0.010 s	
parallel_function	1	18.922 s	0.021 s	
parallel_function>distributed_execution	1	18.874 s	0.016 s	
...>remoteparfor.getCompleteIntervals	5	18.830 s	0.023 s	
java.util.concurrent.LinkedBlockingQueue (Java method)	28	18.752 s	18.752 s	
parpool	1	12.853 s	0.002 s	
doParpool	1	12.850 s	0.002 s	
Pool.Pool>Pool.hBuildPool	1	12.534 s	0.004 s	
Pool.Pool>iStartClient	1	12.501 s	0.007 s	
...tiveClient>InteractiveClient.start	1	12.481 s	0.023 s	
...eractiveClient.waitForSessionCreation	1	9.800 s	0.066 s	
getJavaFutureResult	11	9.734 s	0.007 s	
java.util.concurrent.FutureTask (Java method)	13	9.727 s	9.719 s	
Job.Job>Job.submit	1	1.218 s	0.003 s	
...er>FileSerializer.saveFieldsToFile	21	0.796 s	0.029 s	
...tiveClient>iCreateCommunicatingJob	1	0.768 s	0.005 s	
FileSerializer>iSaveMat	27	0.671 s	0.664 s	
Job.Job>Job.attachRequiredFiles	1	0.626 s	0.011 s	
...b>CJSCommunicatingJob.submitOneJob	1	0.580 s	0.002 s	
Local.hSubmitCommunicatingJob	1	0.557 s	0.014 s	

Figure 4. Profile Summary with parallel processing

3.0 Results and Observations.

3.1 Results Discussion

Comparing the results between serial and parallel, compute time was reduced by 61% from sequential to parallel using six cores.

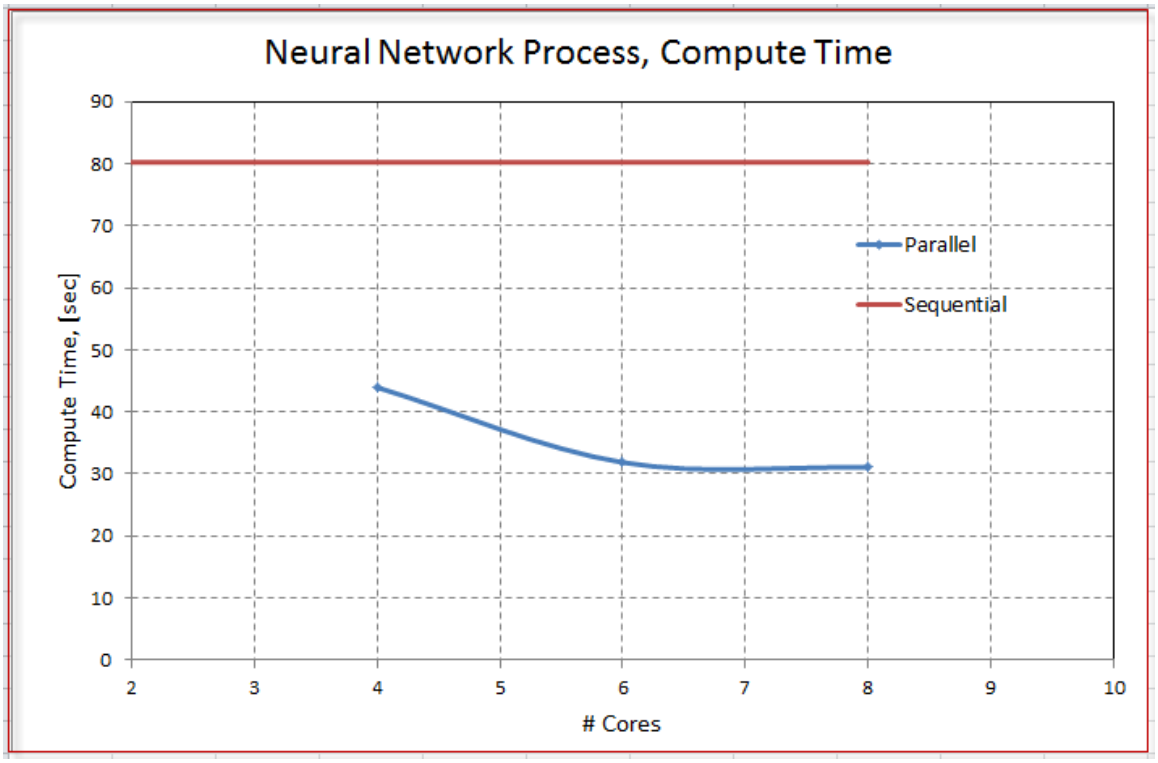


Figure 5. Performance difference using Sequential & Parallel Computing

Above plot shows the effect of diminishing returns as the number of cores used to parallelize. In this particular exercise, outer most loop looped through six times and hence there is no gains using more than six cores.

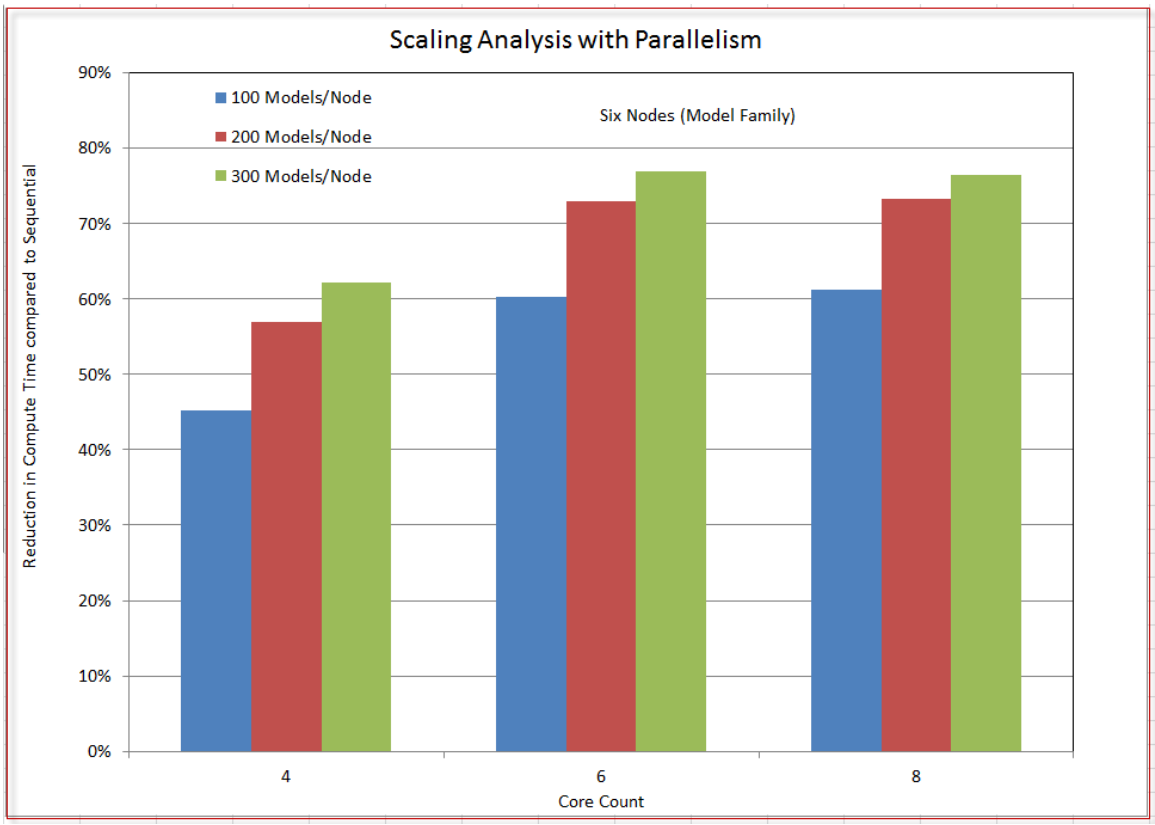


Figure 6. Scaling Analysis with Parallel Processing

Fig 6. Shows percent reduction in compute time compared to sequential, with increased number of models developed per node. Usually depending on the complexity of the inputs models can range from 100 to 500. Similarly, node is the critical variable to analyze complexity in the input data. This typically can range from 5 to 15 nodes in a iterative process.

Overall more than 50% reduction in time was achieved. For example, highly non-linear model with 1000 or more data points with iterative process takes ten minutes. This can now be done in less than five minutes. This is significant, because if the output i.e. select few models doesn't satisfy analyst correlation criteria he might choose to run again with different set of parameters.

3.2 Other Observations

a. When executed in sequential, each loop on an average took 20% less time compared to when executed in parallel. This is due to overhead associated with parallel computing.

b. Matlab has its own rule in the number of core count allowed for specified loops. For example in this case, less than four cores could not be used for six loops (i.e. NN nodes). It only allowed core count of four to eight, eight being machine core count. Similarly, for eight loops it did not allow cores less than six. This behavior could not be understood.

4.0 Conclusion & Future Scope

4.1 Conclusion: Most modern desktop have multi-core capability. Utilizing available compute capability this Neural Network model family development project reduced compute time by at least 50% in most cases.

4.2 Future Scope:

a. Matlab implementation of cores depending on loops used was not understood. With increasing number of nodes for NN models, current parallel code is an iterative process in the selection of cores. Matlab allocation of cores need to be looked into and recoded to make the code robust for any number of loops (NN nodes).
b. Current project focus was limited X data set (374 x 6). What if data set is large array with million or more data points? In which case train function used in Matlab to train NN model would may take more time than the current total time. Matlab allows train function to use GPU resource to train the model faster. This can be implemented for large data set.

Appendix

BuildModelFamily Function call profiling summary. In the figure below, highlighted section in red shows opportunities for more parallelism. This nested parallelism cannot be implemented due to Matlab limitations.

Start Profiling Run this code: [Models, ModelTime] = BuildModelFamily(trnX, trnY, valX, valY, 6, 'trainlm', 'NN', 10, 100)

Profile Summary

Generated 21-Dec-2017 15:53:33 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
BuildModelFamily	1	17.372 s	0.096 s	
newfit	100	8.161 s	0.004 s	
newfit>create_network	100	8.157 s	0.007 s	
newff	100	8.015 s	0.003 s	
newff>create_network	100	8.012 s	0.008 s	
newff>new_5p1	100	8.004 s	0.166 s	
network.subsasgn	3900	7.634 s	0.193 s	
network.subsasgn>network_subsasgn	3900	7.563 s	0.488 s	
network.train	100	5.968 s	0.069 s	
trainlm	1000	4.631 s	0.058 s	
nnModuleInfo	3800	4.596 s	1.962 s	
trainlm>train_network	100	4.568 s	0.036 s	
trainNetwork	100	4.532 s	0.025 s	
trainNetwork>trainNetworkInMainThread	100	4.425 s	0.075 s	
trainlm>trainingIteration	1504	3.262 s	1.281 s	
network.subsasgn>getDefaultParam	1600	1.926 s	0.135 s	
@network\private\nn_configure_layer	600	1.828 s	0.104 s	
setup1	300	1.494 s	0.018 s	

Figure 7. Profile Summary with sequential on BuildModelFamily function.