Skynet: Parallelized Simulation Of Neural Network Architectures

Chirag Sangani
chirags@iitk.ac.in

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Abstract - **This paper proposes a framework architecture for the efficient simulation of large feed-forward neural networks by exploiting the inherent parallelism present in the nature of the problem. A brief discussion on the nature of the neural network architecture supported is followed by a detailed analysis of the framework's design. Simulation data from an implementation of the reference framework design shows promising results with marked improvements in performance, often limited only by the system on which the simulation is run.**

KEYWORDS

Skynet, neural network, parallel computing, simulation.

I. INTRODUCTION

Artificial neural networks are a mathematical model of computation inspired by biological neural networks. A neural network consists of a group of neurons that are interconnected and process information by communication and computation. Mathematically, neural networks are nonlinear statistical data modeling tools - they are usually employed to model complicated, often nonlinear, relationships between inputs and outputs, or to find patters in data by clustering. These usages are employed by a method called as learning - a phase where the neural network behaves as an adaptive system that changes its structure in response to the input provided and the output generated.

The utility of neural networks emerges from the fact that it can "learn" a relationship function from observed data. This is particularly useful in applications where the complexity of the data or its size makes the inference of such a function by hand impractical.

Broadly, the application of neural networks falls in the categories of function approximation, classification, data processing and robotics. Other applications may include system control, decision making, pattern and sequence recognition, financial applications, filtering and data mining, etc. Artificial neural networks may also be used to understand the nature of biological neural networks, since their functioning and structure is closely related.

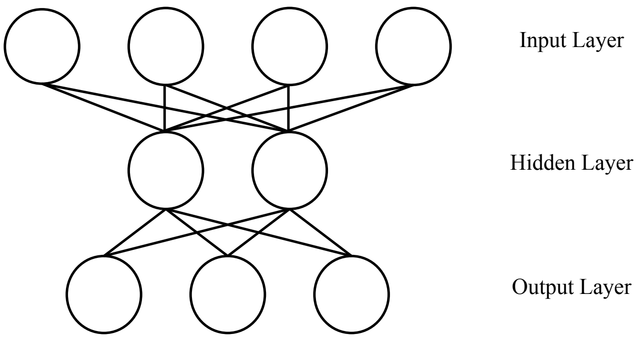

Fig. 1: A typical multi-layered perceptron neural network architecture.

II. NEURAL NETWORKS

This section provides a brief description of the neural network architecture known as a multilayered feed-forward network, which is the architecture supported by the Skynet framework. This architecture is powerful in the fact that it is proven to be a universal function approximator, given the appropriate parameters of capacity and learning algorithm [2]. However, in general, determining the values of these parameters is hard.

A feed-forward neural network consists of layers of input and output nodes, with optional layers in between known as hidden layers. Connections exist only from one layer to another in an creasing order starting from the input. This means that data can flow in only one direction - from the input nodes to the output nodes.

The perspective of a single node in the network comprises of multiple input connections from nodes of a lower layer and a single output to nodes of a higher layer. All connections have a weight associated with them. Consider a mathematical model of a node *k*, with input values $i_1, i_2, ..., i_n$ associated with weights $w_1, w_2, ..., w_n$ (Fig. 2). Each node has an associated function known as the activation function, denoted by *f*. This activation function may be different for different nodes in the network, or it may be the same within the same layer or it may be the same

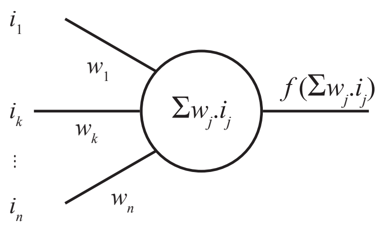

Fig. 2: The mathematical model of a single node in a multi-layered feed-forward neural network.

throughout the network. The output of the node O_k for the input vector $\underline{i}$ is defined as:

$$O_k(\underline{i}) = f(\Sigma w_j.i_j)$$

In literature, it is common to find an extra additive term in the above equation, called as the bias. This additive constant can be interpreted as an extra input node with a constant output of 1 and a connection weight of the desired bias *b*. Thus, the equation above suffices for the description of a multilayered feed-forward network.

The process of evaluating the output for a particular input vector $\underline{i}$ is an iterative process over all layers in the network. The input is fed to appropriate nodes in the hidden or output layer through weighted connections, and the output of the layer is determined. This output is then fed to the next layer as input, or is returned as the final outcome in case of the last layer. As well shall see, this iterative process lends itself very well to parallelization and pipelining, both of which are characteristics of the Skynet framework.

Training a neural network involved determining the number of nodes in the hidden layer according to the complexity of the problem, the connectedness of the network as per the number of connections in the network, the activation functions to be used and the weights of the connections between nodes. Regression algorithms such as back propagation do exist for determining the weights of the connections so that the error for a particular training data set is minimized [1]. However, this paper is solely concerned with the efficient simulation of an already trained network. While certain modifications to the Skynet framework may allow for efficient and parallelized implementation of training algorithms, they are beyond the scope of this paper and, henceforth, will not be touched upon.

III. The Skynet Framework

A. Overview

The Skynet framework exploits the property that the output of a node depends only on the outputs of nodes of a previous layer to extract massive parallelism from the problem of finding the solution of the input vector. The simulation works as follows: multiple threads are set up for simulating the network. The task of simulating nodes is spread evenly over all the threads. On a simulation "tick", every thread simulates one iteration of all the nodes assigned to it, and passes on the output to the next layer of nodes. The output at the output layer is also collected, and the input for the next iteration is set. After a certain number of ticks have elapsed, the simulation is deemed complete and the collated outcomes are returned to the user.

B. Node Descriptor

Within the framework, each node is associated with certain properties. These include a unique identifier for that node, its activation function, a map of input connections with the input nodes' identifiers as the keys and the weights of the connections as the values, flags to determine whether the node is part of the input or output layers, the output of the node for the most recent "tick" and a temporary variable for interim calculations. All these properties are encapsulated within a single object representing that node. This object is unique, though multiple references to it may exist. Inside the system memory, this object is allocated such that its associated thread experiences the least latency while accessing it, since this access lies in the critical path of the execution of each iteration. This means that in a distributed shared memory multiprocessor system, this object is most likely to be allocated to the local memory of the processor executing the object's associated thread.

```
Object Node,
     Integer : Identifier
     Real : ActivationFunction(Input)
     Map(Integer, Real) : Inputs
     Flag : IsInputNode
     Flag : IsOutputNode
     Real : MostRecentOutput
     Real : TemporaryVariable
EndObject
```

Fig. 3: The unique descriptor of a single node within the Skynet framework.

C. Simulation Thread Descriptor

A simulation thread is the workhorse of the framework, performing the actual task of simulation. Along with simulation, a thread is also required to synchronize with other simulation threads, and the control thread, in order to ensure correctness of execution. Generally, a thread runs in its own hardware context in parallel with other threads, however, it may and does access data globally available as well as local to other threads. Such accesses are necessary in cases of input data arriving from a node that is assigned to a remote thread.

The local data of a thread includes a map of identifiers to the objects of the nodes associated with that thread and a flag to indicate termination of the program. Each simulation thread also has access to global synchronization constructs to implement the "tick" model.

```
Object SimulationThread,
      Flag : Terminate
      Map(Integer, Node) : Nodes
EndObject
```

Fig. 4: The unique descriptor of a simulation thread within the Skynet framework.

D. Control Thread

The control thread is responsible for initialization of the simulation threads, fair distribution of workload among the threads, initialization of the synchronization constructs part of the "tick" model, initiation of the simulation and its eventual termination, collection of output data and feeding of input data every cycle, and output generation. The local data of the control thread contains various information to assist it in its work, and is also visible to all simulation threads. While the control thread runs in its own hardware context, it does not consume any processor resources during simulation: it lays dormant throughout the duration of the "tick", awaking only to check the state of the simulation and to feed/collect data to/from input/output nodes.

```
Object ControlThread,
      Map(Integer, Real)[] : Input
      Map(Integer, Real)[] : Output
      Map(Integer, Node) : Nodes
      Map(Integer, Node) : InputNodes
      Map(Integer, Node) : OutputNodes
      Map(Integer, Node)[] : HiddenNodes
      SimulationThread[] : Threads
      TickModel : Timer
EndObject
```

Fig. 5: The unique descriptor of the control thread within the Skynet framework.

The `Input` and `Output` objects are arrays of integer to real maps. Each entry in the array is a map, and corresponds to a single input/output vector. The map associates with each input/output node identifier a real value denoting the input/output for that node. The output map for a particular index value *i* corresponds to the input map for that index value.

The object `Nodes` is a map from an integer identifier to its associated unique `Node` object. Note that this mapping only provides a reference to the unique value, which is ideally located in the local memory of the associated simulation thread. The objects `InputNodes` and `OutputNodes` are subsets of this map. `HiddenNodes` is an array of such subsets, each entry in the array representing a map of hidden nodes present in a particular layer.

E. The "Tick" Model

The Skynet framework exploits the parallelism present in the simulation of nodes within the same layer. However, the task of simulating nodes across is not parallelizable, since an inherent dependence is present: the output of a node in a lower layer is presented as an input to a node in a higher layer. Hence, these simulation tasks need to be necessarily sequential. To introduce this sequential nature, a library of synchronization constructs is requires that enables the implementation of the "tick" model.

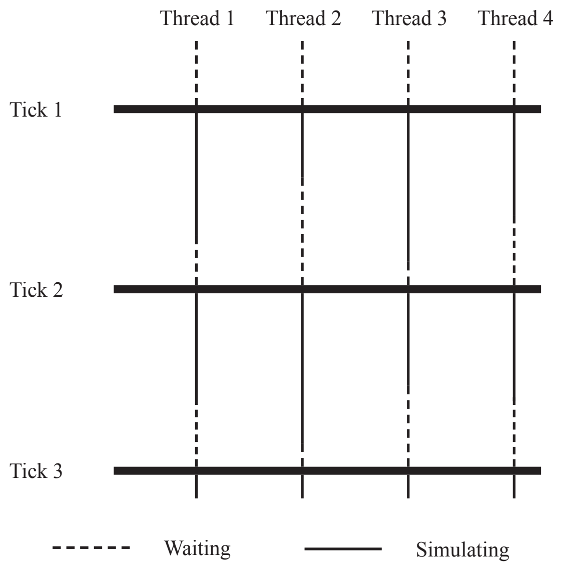

Fig. 6: The thread trace for a four-thread system utilizing the "tick" model.

A "tick" is a single iteration in the simulation of a particular layer. Before the tick arrives, all simulation threads are in a waiting state. The tick wakes all waiting threads, which now carry out the task of simulation of their ward nodes. On completion of one iteration of simulation, a simulation thread waits until another tick arrives before proceeding with the next iteration. A tick is generated by the system only when all threads have completed their current simulation iteration. Fig. 6 provides a visual representation of this process.

The tick model allows for sequential simulation of layers, with the simulation of individual layers internally parallelized. On the first tick, the input nodes can be fed with the values of the input vector. On the subsequent tick, these values can be fed to the lowermost hidden layer node, and their outputs can be computed in a parallelized fashion. This process can be continued until the output nodes provide the output values corresponding to the original input vector. The process can now be repeated in its entirety, leading to a naive simulation model as shown in Table I. The values in the table represent the input vector processed. The position of the value in the table denotes the phase of the processing - the rows denote on which tick the operation was performed, while the columns denote the layer. For example, i_1 was in the input layer stage at tick 1, hidden layer 1 stage at tick 2, and so on. Once the processing of the input vector is finished and the data is collected from the output layer, the processing of the next input vector begins.

The naive simulation model detailed above, while correct,

TABLE I

NAIVE SIMULATION MODEL FOR A FOUR-LAYERED NETWORK

Tick	Layer			
	Input	Hidden 1	Hidden 2	Output
1	i_1			
2		i_1		
3			i_1	
4				i_1
5	i_2			
6		i_2		
7			i_2	
8				i_2
9	i_3			
10		i_3		

leads to a potential waste of resources. Consider the simulation model depicted in Table I. The resources allocated to the input layer are utilized only for one out of the four stages of processing for any input vector. The same holds for the resources allocated to the other layers. Intuitively, it is possible to avoid this wastage by beginning the processing of the next input vector at the input layer stage immediately from the next tick, thus "compressing" out the simulation model by removing the empty gaps. The resulting model is called as a "pipelined" simulation model since it derives the technique from pipelined execution stages of a microprocessor [6]. Table II shows the pipelined simulation model for the neural network considered in Table I.

The performance improvement afforded by the pipeline model depends upon the number of layers present in the network. Since the next tick cannot occur until the vectors currently in the pipeline are not completed, the optimum performance case for the minimum time per tick occurs when all layers require approximately equal amounts of time to simulate [4]. In the asymptotic case, the speedup factor S which denotes the decrease in time is given by:

TABLE II

PIPELINED SIMULATION MODEL FOR A FOUR-LAYERED NETWORK

Tick	Layer			
	Input	Hidden 1	Hidden 2	Output
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	i_4	i_3	i_2	i_1
5	i_5	i_4	i_3	i_2
6	i_6	i_5	i_4	i_3
7	i_7	i_6	i_5	i_4
8	i_8	i_7	i_6	i_5
9	i_9	i_8	i_7	i_6
10	i_{10}	i_9	i_8	i_7

TABLE III

PHASE-PIPELINED SIMULATION MODEL FOR A FOUR-LAYERED NETWORK

Tick	Layer			
	Input	Hidden 1	Hidden 2	Output
1+	$RE(i_1)$			
1-	$WO(i_1)$			
2+	$RE(i_2)$	$RE(i_1)$		
2-	$WO(i_2)$	$WO(i_1)$		
3+	$RE(i_3)$	$RE(i_2)$	$RE(i_1)$	
3-	$WO(i_3)$	$WO(i_2)$	$WO(i_1)$	
4+	$RE(i_4)$	$RE(i_3)$	$RE(i_2)$	$RE(i_1)$
4-	$WO(i_4)$	$WO(i_3)$	$WO(i_2)$	$WO(i_1)$
5+	$RE(i_5)$	$RE(i_4)$	$RE(i_3)$	$RE(i_2)$
5-	$WO(i_5)$	$WO(i_4)$	$WO(i_3)$	$WO(i_2)$

RE denotes "Read and Execute". WO denotes "Write Output"

$$S = 1 / n$$

Where n is the number of layers (including input and output layers) present in the network.

The introduction of a pipelined model of simulation gives rise to a correctness issue called as a "read after write" race condition [3]. This condition occurs when the read for a particular node's input conflicts with the write to that input node's subsequent iteration's output. Consider the simulation model depicted in Table II. In tick 1, the input node writes the output for i_1 to its output variable. In the subsequent tick, this output is read by a node in hidden layer 1, but this read may conflicts with the write to the input node's output variable for i_2. Since the two processes occur in parallel, it is possible that the former is preceded by the latter. In such a case, the node in the hidden layer 1 would read the output for i_2, and the data would be lost.

To resolve this issue, it is necessary to serialize the write to an output variable and subsequent reads to that variable. To do so, the Skynet framework employs the technique of dividing the "tick" into two phases - a positive "read and execute" phase, followed by a "write" phase. In the "read and execute" phase, no node is allowed to modify any output variable. The output values from the input nodes are read, the new output is generated, and is stored in a local, temporary variable. In the "write" phase, no node is allowed to read another node's output variable. The value stored in the temporary variable is now transferred to the node's output variable. This process completes a single tick. The advantage of this method is that race conditions are completely eliminated. The disadvantage, however, is that the extra synchronization point inserted between the two phases increases the time for a single tick, thereby decreasing performance. This decrease in performance, however, will be shown to be amortized by the increase in performance due to pipelining.

Table III shows the modified pipelined simulation model with read and write phases.

```
Object TickModel,
      Integer : CurrentTick
      Integer : MaxTicks
      Barrier : PositivePhase
      Barrier : NegativePhase
EndObject
```

Fig. 7: The unique descriptor of a TickModel within the Skynet framework.

F. Simulation Algorithm

A simulation run in the context of the Skynet framework begins with the generation of the network architecture and the input vector map arrays. This task is performed by a front end interface to the control framework, and may provide multiple features. However, this front end interface is strictly not a part of the framework - Skynet expects that the network architecture and input vector map arrays are readily available at the start of the simulation.

The framework is initialized by the creation of a single control thread. Before the thread is run, it is provided with parameter values for the variables `Input`, `Output`, `Nodes`, `InputNodes`, `OutputNodes` and `HiddenNodes` (see Fig. 5).

The control thread begins by initializing the required number of threads, which can be passed as a parameter. It is recommended that this number be equal to the number of hardware contexts available. The major bulk of computation here involves the fair distribution of the node tasks to the individual threads. Once this is complete, the control thread sets the initial input vector and initiates the positive phase for the first tick. Thereafter, it waits until the requisite tick have elapsed, on which it terminates the simulation threads and generates the output. Fig. 8 shows a succinct algorithm of the execution of the control thread.

The working of a simulation thread is relatively simpler. The thread runs within an infinite loop, expecting an infinite number of ticks. At the start of every iteration, it checks if the simulation is still running, terminating otherwise. On passing this check, it simulates the “read and execute” operation of every node associated with it, storing the outcome in a temporary variable. The thread then awaits the negative phase. In the “write output” phase, the thread simply transfers the value in the temporary variable for every node into the node’s output, and awaits the next tick. Fig. 9 gives the algorithm for the execution of a simulation thread.

The control thread synchronizes with the simulation threads on every tick for the beginning of the positive phase. This is accomplished by the `Await` function. The control thread need not await for the negative phase. Though not shown, the input vectors and output vectors are set / collected as part of the positive phase `Await` function - this happens before threads start executing so as to ensure correctness.

```
Function ControlThread.Start(),
   Initialize(Threads)
   Initialize(Timer)
   For Node in Nodes,
      Threads[Hash(Node)].Nodes.Add(Node)
   EndFor
   For Node in InputNodes,
      Node.output = Input[0].Get(Node)
   EndFor
   For SimulationThread in Threads,
      SimulationThread.Start()
   EndFor
   For Timer.CurrentTick = 1 : Timer.MaxTicks,
      Timer.PositivePhase.Await()
   EndFor
   For SimulationThread in Threads,
      SimulationThread.Terminate = true
   EndFor
   GenerateOutput()
EndFunction
```

Fig. 8: The execution algorithm for the control thread.

```
Function SimulationThread.Start(),
   While(true),
      If Terminate,
         Die()
      EndIf
      For Node in Nodes,
         Real Net = 0.0
         For InputNode in Node.InputNodes,
            Net = Net + Node.InputNodes.
                  Get(InputNode) * InputNode.output
         EndFor
         Node.TemporaryVariable = Node.
             ActivationFunction(Net)
      EndFor
      ControlThread.Timer.NegativePhase.Await()
      For Node in Nodes,
         Node.Output = Node.TemporaryVariable
      EndFor
      ControlThread.Timer.PositivePhase.Await()
   EndWhile
EndFunction
```

Fig. 9: The execution algorithm for a simulation thread.

IV. Performance analysis

Analysis of the execution algorithms of the control and simulation threads reveals that the performance of the framework depends upon the following factors:

1. The average number of nodes associated with each thread.
2. The implementation of the "Hash" function and the "Map" interface.
3. The implementation of the "Barrier" interface.
4. The number of input vectors.

The above factors do not take into consideration factors such as operating system intervention, machine architecture, uneven load balancing, network latency and traffic, etc. These factors are generally difficult to quantify, but important nevertheless, making a formal analysis of the performance of the framework hard. However, it is possible to obtain an idea of the performance of the framework through simulation results and profiling data. The following section presents results obtained from a reference implementation of the Skynet framework.

V. Simulation results

A. Overview

For the purposes of simulation and testing, a reference implementation of the Skynet framework was implemented using the Oracle® Java™ platform [5]. The object-oriented nature of this platform, coupled with an extensive library for data structures and concurrency lends itself very well to support the development of Skynet. In addition, the portable nature of the platform means that Skynet can run on any architecture supported by the JVM™.

The `Map` interface used within the Skynet framework was implemented using the `HashMap` class. For the implementation of synchronization constructs, the `CyclicBarrier` class was chosen.

The source code for the reference implementation is available at http://www.chiragsangani.com/assets/ZIP/Skynet.zip.

B. Profiling Results

To analyze the performance of the control and simulation thread algorithms, the framework was tasked with the simulation of a neural network architecture with random input values. This network consisted of 3 layers and 300 nodes, spread evenly over all the layers. Two consequent layers were fully connected. The weights of the connections were randomized, since they do not affect the computational complexity. The simulation was run twice - once with 1 thread, and the second time with 4 threads, and the execution was profiled. A total of 100 input vectors were presented to the system.

The machine on which the profiling was performed comprised of a quad-core Intel® Core™ i7 740QM processor, paired with dual-channel 8 GB DDR3 RAM at 1333 MHz. The processor was clocked at a maximum frequency of 2.93 GHz. The platform was JRE® 6 running on Microsoft® Windows® 7 x64.

TABLE IV
Profiling results of a single thread execution

Function	Time %	Invocations
Map.Get()	41.5%	4040000
Self Time	57.2%	1
Barrier.Await()	1.3%	202

TABLE V
Profiling results of a multiple thread execution

Function	Simulation Thread 1		Simulation Thread 2	
	Time %	Invocations	Time %	Invocations
Map.Get()	11.4%	1010000	6.7%	1010000
Self Time	3.2%	1	9.9%	1
Barrier.Await()	85.4%	202	83.4%	202

Function	Simulation Thread 3		Simulation Thread 4	
	Time %	Invocations	Time %	Invocations
Map.Get()	22.5%	1010000	18.1%	1010000
Self Time	19.6%	1	24%	1
Barrier.Await()	57.9%	202	57.9%	202

The results of the profiling tests are presented in tables IV and V. Table V shows the profiling result for the execution of a single simulation thread. Table VI shows the profiling result for the execution of 4 simulation threads.

While small variations in numbers can be safely ignored due to error margins, the most striking change in transitioning from a single-threaded program to a multithreaded program is that synchronization occupies a majority of the lifetime of a simulation thread. This time is extra overhead time spent in synchronizing the multiple threads, and is required to be as little as possible relative to the simulation time.

The time to synchronize threads using a barrier is a function of the number of participating threads, and is independent of the size of the neural network architecture. Therefore, the number of threads to be initialized should be determined according to the size of the network - the larger the network, the more number of simulation threads. This thumb rule will be exemplified by simulation timing results presented in the next subsection.

C. Timing Results

To analyze the real world performance of the Skynet framework, a number of simulation runs were performed for different neural network sizes. The machine on which these timing experiments were run was a distributed shared-memory multiprocessor machine comprising of 4 Intel® Xeon™ server-grade processors, each supporting 4 threads of execution, for a

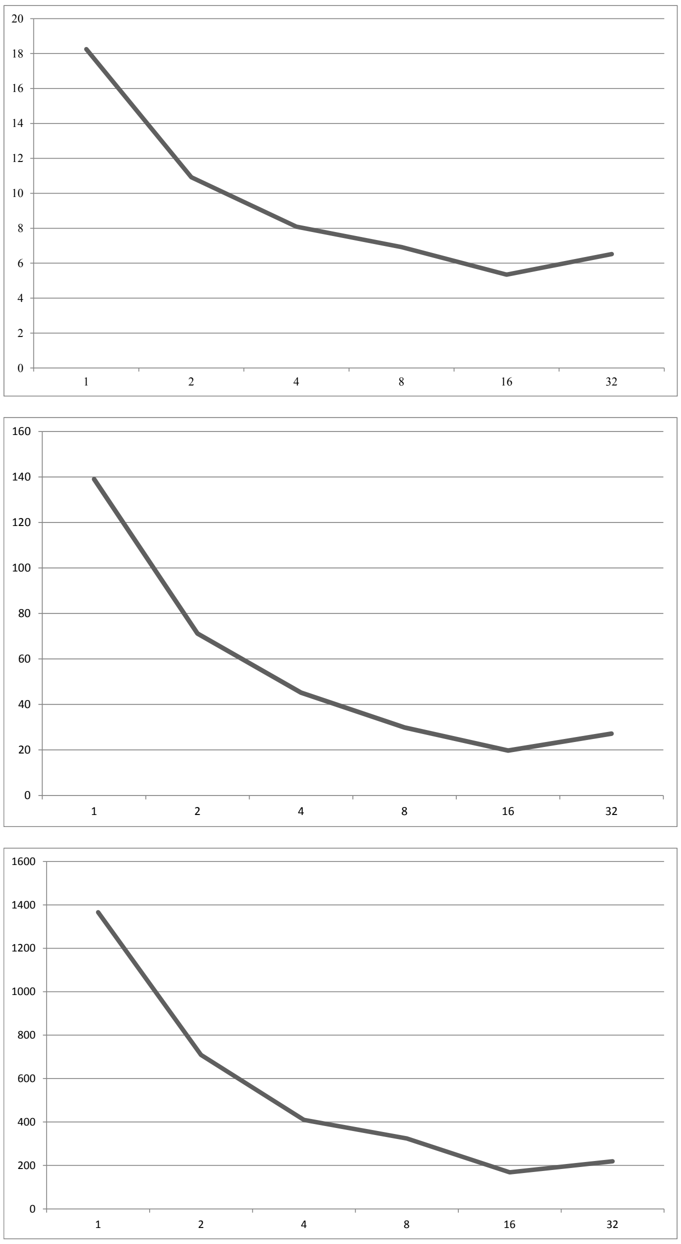

Fig. 10 (a), (b), (c) (from top to bottom): Simulation time (in seconds) for different number of threads for (a) 10, (b) 100 and (c) 1000 input vectors respectively.

total of 16 hardware contexts. A total of 8 GB physical memory was available to the system. The platform was JRE® 6 running on a 64-bit Linux distribution.

Figures 10(a), 10(b) and 10(c) show the execution time required for simulation runs with varying number of threads for 10, 100 and 1000 input vectors respectively. The neural network architecture in these runs consisted of 1000 input nodes, 100 output nodes and 1 hidden layer comprising of 10000 hidden nodes. The layers were fully connected, giving rise to 11 million connections. The simulation times for different number of vectors and threads were collected and plotted to show trends in the data.

From the data, it is obvious that the application of multiple

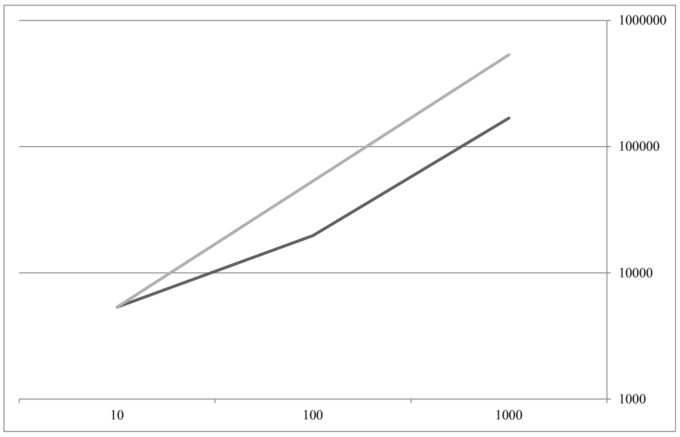

Fig. 11: Simulation time (in seconds) on a log scale for different number of input vectors.(dark line) vs. a linear increase in simulation time (light line).

threads using the Skynet framework provides a huge boost to the performance of simulation of a large network. Closer observation reveals that for a large number of input vectors (Fig. 10(c)), the speedup factor for an increase from one to two threads is almost 0.5, i.e., the performance is almost doubled. This performance improvement continues for an increasing number of threads, albeit slower, indicating that the cost of synchronization is slowly catching up with the boost in performance due to multiple threads. The performance boost maximizes at 16 threads, with an increase in execution time for 32 threads. This increase in execution time can be explained by the fact that there are only 16 hardware contexts available at any moment. Thus, even if 32 threads are present, only 16 can run concurrently, the rest waiting in line for the active threads to finish their work or to get descheduled. This overhead of waiting and scheduling causes a decrease in performance as compared to 16 threads.

Fig. 11 shows the advantage multiprogramming has for large input vectors. The dark line represents the time of execution for exponentially increasing number of input vectors on an exponential vertical scale of time in seconds. The light line represents a linear increase in time. The number of threads in consideration is 16. The super-linear increase in performance can be explained by the decrease in the fraction of overhead cost as compared to the actual simulation time.

To summarize, multithreaded simulation helps immensely in the application of large neural networks. The performance gains for highly parallelized systems increase proportionately to the size and complexity of the network.

VI. Future Work

The basic framework of Skynet allows for parallelized simulations, and is highly flexible and modular. It can be easily modified to support other network architectures, such as recurrent networks.

Currently, Skynet supports only the simulation of a neural network. Training methods such as back-propagation, etc. are not supported. While it is possible to implement a batch-algorithm for training, it would run considerably slower as compared to inbuilt support for training. Since training requires a process similar to simulation, i.e., modification of weights based upon feedback for output, it is relatively simple to implement a training algorithm that is also parallelized and enjoys the same advantages as the current simulation framework.

VI. Conclusion

This paper proposes a framework for the efficient and parallelized simulation of large neural networks. A detailed description of the proposed framework follows, with emphasis on object-model details. Algorithms for a reference implementation of the framework are provided, with detailed descriptions. A reference implementation of the framework is tested with multiple simulations to verify the claims of improved performance. Within error margins, the framework is shown to indeed benefit the simulation of large-scale networks, with anomalies, if any, explained to be artefacts external to the framework. A brief summary and potential for future work concludes this paper.

Acknowledgement

The author gratefully acknowledges the assistance of Dr. Krithika Venkataramani at the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur for her excellent teachings, helpful comments and suggestions.

References

[1] Arthur Earl Bryson, Yu-Chi Ho (1969). Applied optimal control: optimization, estimation, and control. Blaisdell Publishing Company or Xerox College Publishing. pp. 481.

[2] Cybenko., G. (1989) “Approximations by superpositions of sigmoidal functions”, Mathematics of Control, Signals, and Systems, 2 (4), 303-314.

[3] Hennessey J. and Patterson D., *Computer Architecture: A Quantitative Approach*, 4th ed., Elsevier, ISBN 978-81-312-0726-0. p. A-11.

[4] Kunkel, S. R. and Smith, J. E. (1986). Optimal Pipelining in Supercomputers. In ISCA-13, pages 404-414, Tokyo, Japan.

[5] Oracle Technology Network for Java Developers: http://www.oracle.com/technetwork/java/index.html

[6] Raul Rojas (1997). “Konrad Zuse’s Legacy: The Architecture of the Z1 and Z3”. IEEE Annals of the History of Computing.