A SourceForge.net Project: tMans, an Agent-based Neural Network Simulator, Repast, and SourceForge CVS

John Korecki
Computer Science & Engineering REU
University of Notre Dame
Fall ‘04 - Spring ‘05

September 2005

1 Introduction

The Agent Based Neural Network Simulator is an object oriented framework designed using agent based simulation concepts. The framework itself was "initially developed to explore self organizing structures in biologically inspired neural networks", but can be extended to explore the properties of both biological and artificial networks [1]. To work with the Agent Based Neural Network Simulator, one must first understand artificial neural networks, as well as the "self organizing structures", particularly structures exhibiting small-world properties, present in biological neural networks. In addition, the simulator is implemented using Repast, an agent based simulation toolkit, allowing very flexible control over model parameters both in code and at run time. The simulator is also available on Sourceforge as a tool for others to use as a framework for neural network modeling. Sourceforge acts as a point of distribution, so understanding the features and tools available to support developers aids in the success of the simulator.

2 Small-World Networks

The Agent Based Neural Network Simulator combines together research from many different fields. One must complete some background reading before understanding the underlying concepts. One of the first areas to examine is small-world networks in order to understand the performance measures used to evaluate the simulator. Small-world networks became an important topic of research when it was shown that real networks have been found to demonstrate these properties. A few examples include the neural network of the nematode worm *C. elegans*, the collaboration graph of actors in

feature films (an indicator that social networks are small-world in nature), some transportation networks, power grids, and even the internet. Small-world networks can be large networks that have a small feel to them, and help describe disease outbreak or the robustness of the internet against a terrorist attack. A simple definition of a small-world network is one in which, despite the overall size of the network, for any two nodes chosen there is consistently a short path between them. From work with the neural network of the nematode worm *C. elegans*, it has been shown that a biological neural network exhibits the characteristics of a small-world network. A neural network simulator could potentially use the same measures for performance and data propagation.

Performance Measures

Two performance measures were initially suggested by Strogatz and Watts [2] for identifying networks with small-world tendencies. The first is the characteristic path length. For a given graph *G*, the characteristic path length is defined as:

$$L(G) = \frac{1}{N(N-1)} \sum_{i \neq j \in G} d_{ij}$$

where d_{ij} is the shortest length from node *i* to node *j*. The characteristic path length is basically the average of the minimum distance between any two nodes in a graph, and gives a general feel for the length of any edge in the graph. The second performance measure suggested to identify networks with small-world properties is the clustering coefficient, or the measure of the average cliquishness, defined for a graph *G* as:

$$C(G) = \frac{1}{N} \sum_{i \in G} C_i \qquad C_i = \frac{e_{G_i}}{k_{G_i}(k_{G_i} - 1)/2}$$

where N is the number of nodes in G, G_i is the subgraph containing all of the nodes to which node i connects, e_i is the number of edges in G_i and k_i is the number of nodes in G_i. For a given node i in graph G, there is a set of nodes to which node i connects. These form the set of nodes called G_i If G_i were fully connected, i.e. each node in G_i had an edge to each other node in G_i, then the number of edges in G_i is $k_i(k_{i-1})/2$. Dividing the actual number of edges in G_i , e_i , by the total possible number of edges, we get a ratio describing how fully connected the neighbors are of node i in G. By summing the ratios for each node and dividing by the number of nodes, we find the average measure of how fully connected nodes are in G.

Small-world networks are then defined as networks having a large amount of clustering and short characteristic path lengths. These networks are important because they shows that given two nodes in G, there exists a short path between them, even when the total number of nodes in G is rather large. A classic example is the graph of actor collaboration. Given an actor or actress, one can find relatively small number of collaborations leading to any other actor or actress. Regular graphs are usually highly clustered, but have long path lengths, while random graphs show no clustering and have short characteristic path lengths.

Another measure of performance used for small-world networks utilizes the harmonic mean of all distances in the graph. One major distinction made using this measure is the difference between $l_{i,j}$ and $d_{i,j}$. The former is the distance between two nodes i and j, either in terms of a weight or a physical distance, while the latter is the sum of the distances between each two nodes along the shortest path between nodes i and j. The connectivity length can be defined as:

$$D(G) = H(\{d_{i,j}\}_{i,j \in G}) = \frac{N(N-1)}{\sum_{i,j \in G} 1/d_{i,j}}$$

This is the distance at which all nodes would need to be relative to each other to maintain performance of the graph, where performance is defined by

$$P = \sum_{i,j \in G} v / d_{i,j} \ ,$$

the sum of all velocities of data flow between two nodes divided by the shortest length path between those two nodes. The connectivity length of the graph is basically the mean rate of information propagation in a network. This measure is most visibly applicable to situations where traffic maps exhibit small-world characteristics, or in disease outbreak simulations [3].

3 Neural Networks

As the name implies, another large field the Agent Based Neural Network Simulator (ABNNSim) taps into is the field of neural networks [4]. Traditional neural networks use nodes that compute some function of the sum of the weights of its inputs. Hopefully this function can be adjusted to produce a desired result given a set of data. Modifying the weights of the inputs based on error mimics synaptic learning and helps train a neural network to compute a proper function of the input and produce similar results for further input. Commonly, two classes of functions are used in the net: sigmoidal and radial basis functions. Sigmodial basis functions are functions that produce a curve having an 'S' shape, known as sigmoid curves [5]. One example of a sigmoid

curve is the logistic function defined by the formula:

$$P(t) = \frac{1}{1 + e^{-t}}$$

These classes of functions are used known as universal basis functions; any function can be expressed as a linear combination of universal basis functions. Using a linear combination of universal basis functions then allows the neural net to adjust the parameters of these functions to find an equation which should accurately interpolate new data.

Training Neural Networks

In order to train the neural net, it is fed sets of input data whose result is known. Some function is used to determine the error value and a method of gradient decent is applied to adjust the weights of inputs to each node. The error between the value produced by the net and expected value determines the amount of adjustment to the parameters. On each node, the weights assigned to inputs are updated to be some portion of the change in error with respect to the change in weight. This altering of the weights to approach some minimum error point trains the current node. Then, the amount of error that a current node experienced is sent back to the nodes (if any) that contributed values forward to the node. This method of moving error from the nodes responsible for output back through the network is called error back-propagation. Conceptually, for any given node, if it experienced a high level of error, there are two possible reasons; either the node is not adjusted correctly or the nodes that act as input are not adjusted correctly. Back-propagation allows for the training of nodes that do not receive direct feedback about the validity of their results.

Important to the discussion of training a neural network is what portion in the change in error to use when modifying the weights, called the learning rate. If the learning rate is too high, the move toward the minimum will be too large, skipping around and no converging to any point. If the rate is too low, then the net will suffer from slow convergence. Another issue when training neural nets is overfitting. When a neural net is trained on a particular set too frequently, the net becomes too finely adjusted to the sample data. In other words, the neural net memorizes the data instead of attempting to generalize solutions. Common approaches to combat overfitting include using validations sets and training with nodes. Validation sets are set not used to train the net, but to monitor the nets progress towards a general solution. For some number of training sets used, the net is checked against a validation set to ensure the neural net is not overfitting the data. Using noise means to corrupt the data in such a way that the neural net does not have the opportunity to fit itself exactly to input data sets. The noise has to be such that the data corruption does not alter the general solution greatly.

Experimental Neural Networks

After examining the theory of small-world networks and the basic concepts of traditional neural networks, the next area to understand is some of the related works with experimental neural networks. Of particular interest and relevance to ABNNSim is HebbNets [6][7]. HebbNets attempts to simulate a nervous system by using a concept called Hebbian Learning. Hebb's postulate states that "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased." Basically, when neuron A fires, if a

neuron B helped neuron A reach its firing threshold, then the link between neurons A and B strengthens. It is possible to draw parallels between using back-propagation to adjust input weights. Hebbian Learning forms the basis for the adaptability of the nervous system. Working off the assumption that Hebbian Learning can be sustained by random noise input, HebbNets attempts to examine the effect of random noise on network structure. HebbNets showed a lower local connectivity length compared to corresponding random networks and though the connectivity structure was sparse, information flow was tolerant and efficient. By increasing the level of noise, the average connection length spiked, but the network efficiency stayed relatively the same. These results demonstrate that random noise input on each neuron can sustain and shape the neural network structurally. HebbNets can be viewed as the synthesis of preferential attachment and the model of Watts and Strogatz, but can produce results without the explicit requirement on growing and without a direct mechanism of weight rewiring. HebbNets shows that local learning rules and random noise input can form and sustain small-world networks.

4. Repast

Previous research on small world and neural networks helps to understand the intent of ABNNSim. It's also important to understand the tools used to implement ABNNSim, such as Repast, the agent-based simulation toolkit used in the implementation of the neural network. Repast has many features that make it a good choice for agent based simulation. It supports dynamic access to agent properties, agent behavioral equations, and model properties at run time, allowing greater control over

simulation execution. The built-in scheduling framework supports sequential and parallel discrete event operations. Events that execute in parallel have a simulated concurrency, while events that are se to execute sequentially can be placed into 'ActionGroups,' where event execution can be ordered accordingly.

Repast 3.0 surpasses Repast 2 by far, introducing new features that provide increased flexibility and functionality, while maintaining reverse compatibility with previous Repast releases; Simulations written in Java using Repast 2 are compatible with Repast3.0, and in the case of ABNNSim required little more than a change in the Java CLASSPATH variable when compiled. Repast 3.0 is fully implemented in both Java and in C#. Repast was originally written in Java and the implementation in C# is a port to the .Net Framework. In addition, RepastJ simulations are cross-platform, allowing simulations to move easily from system to system. In addition, RepastPy, the Repast implementation in Python, takes created simulations and exports them to Java to actually run.

RepastPy and Repast.Net

RepastPy uses a front end GUI to allow access to common simulation elements and can be used to create simple models. One can create custom agents and environments and set simple parameters, and then start the simulation from within the RepastPy GUI. The RepastPy schedule editor allows flexibility in scheduling events. Actions are scheduled on time ticks and each object in the model has its own schedule to follow. A master schedule contained in the model keeps track of the schedules for each object. In addition, projects can be exported to Java then modified. In this way, RepastPy allows one to create simple projects easily, and then build more complex simulations in Java. An

added feature of the relationship with Java is the ability to import Java functions into RepastPy. When a simulation is initiated in RepastPy, the familiar Repast GUI opens to allow dynamic access to simulation and agent parameters.

Repast.Net, implemented in C#, allows one to create simulations in many different languages through the .Net Framework. The .Net Framework is " a development and execution environment that allows different programming languages & libraries to work together seamlessly to create Windows-based applications that are easier to build, manage, deploy, and integrate with other networked systems" [8]. Basically, since Repast.Net is implemented in C#, one can implement Repast simulations in any of the .Net languages (VB.Net, C++, J#, C#, etc.) for a Windows environment.

Of the new Repast language offerings, RepastPy is by far the easiest to use. It allows users to quickly develop simple simulations using provided templates. In addition, RepastPy will export to Java, making it an excellent choice for rapid simulation development and for learning about Repast. RepastJ has the advantage over other Repast implementations because of Java's cross platform nature and because Java is the original language for the Repast implementation. Repast.Net is superior to the other Repast implementations in terms of language choice flexibility since it opens the development of Repast simulations to a wide range of languages and through that, developer who do not know Java yet still wish to explore Repast's modeling and simulation abilities.

New Features

Repast 3.0 also comes with a few features and libraries worth noting. One such feature is the Automated Monte-Carlo Point and Click Framework. This feature allows one to automate running simulation with a range of input values. It supports incremented

data ranges, lists of values, or constant values for the input to a simulation, as well as nested parameter groupings. One of the best features is that the Framework will create an XML file saving the settings for a particular test in order to re-run the test without having to reset all of the desired parameters. Using the Framework also limits the need for developing test scripts to run simulations and generate output.

Some of the additional libraries worked into Repast include support for a Geographical Information System using either ArcGIS or OpenMap. This allows simulations using actual geographical data. There is also support for genetic algorithms, traditional neural networks, regression analysis, and systems dynamics, which allow one to wrap equations in pseudo agents and use the scheduler to ensure the correct timing of computation.

5. ABNNSim

These concepts and libraries support the functionality of ABNNSim. The simulator itself is organized into three primary classes. The *ABNNSim* class controls the simulation execution and set parameters needed by Repast. It acts as the starting point for the simulation by creating each of the individual objects used, setting up graphs or file output, and scheduling events to occur at certain time ticks in the simulation. The two active classes in the simulation are the *Axon* class and the *Neuron* class. The *Neuron* class represents the individual nodes in the network, and holds information about the neuron such as position in the world, firing threshold, and timing information. It also contains methods for firing the *Neuron* object and sending feedback to those *Neuron* objects that helped push other *Neuron* objects to fire. The *Axon* class acts as the link between *Neuron*

class objects, or the edges between the nodes in the network. It controls information flow between two *Neuron* objects, both with respect to firing and with respect to feedback. Various support classes exist as well. The *NetworkConstructor* class sets up the network based on an input value set in the Repast GUI. For instance the network can be created initially in a random manner. The *NetworkAnalyzer* class calculates network wide performance measures, such as clustering coefficients, average connectivity length, and average harmonic mean distances. The *ABNNSim* class will call the *NetworkConstructor* class to create the network, then schedule neuron activity and axon pruning, and acts as a framework for expanding the simulation to explore new concepts.

Extensions to the ABNNSim

One expansion to the ABNNSim explored was adding distance based network creation limitations. To implement the functionality, the *Neuron* class was modified to compute its Cartesian distance to any other *Neuron* object in the network. After every node has been placed and initialized, the *ABNNSim* class checks a flag set in the repast GUI to determine network type. If the network type chosen is the distance biased formation, then the *NetworkConstructer* class calls a different routine customized for the formation of the chosen network. In this case, the function chosen randomly chooses two nodes, then checks the distance between the two. If the distance is higher than some cut-off, the connection is not formed. In this way, the network is randomly created but will only contain edges between nodes below some set length.

The next modification to the simulator was to compute three dimensional Cartesian distances and to randomly create the network based on some cutoff of Cartesian distance in three dimensions. This modification also needed some modification

of the Neuron class to store not only the x and y components, but also the z component of the Neuron's location in the simulation space. The method of network creation basically functions the same, with the exception of using the three dimensional Cartesian distance between Neurons instead of the two dimensional distance.

These two modifications show the flexibility of ABNNSim to meet new requirements and research goals. Distance based network creation could be an interesting point of research since in biological neural networks, the probability of a neuron connecting a near-by neuron is more likely than connection to a neuron a long distance away. Network creation based on distance-based limitations and the results of running that simulation may be closer to true neural networks than considering the network as a graph with each edge having a certain weight. The *Axon* class and the firing mechanism can be modified to account not only for normal network weights, but also for a distance fall-off. As a signal propagates from a neuron to another as a result of a Neuron firing, or on feedback being returned, the distance a signal must travel can determine the strength of the signal at the receiving neuron.

6. Sourceforge CVS

After working with ABNNSim and its underlying concepts and libraries, one must consider the organization of the development code and accessibility of the code for use. To serve both of these purposes, Sourceforge.net provides a CVS repository. Sourceforge.net is "the world's largest Open Source software development web site, hosting more that 100,000 projects and over 1,000,000 registered users with a centralized resource for managing projects, issues, communications, and code," [9] and makes an

ideal location for hosting ABNNSim. Using Sourceforge's tools the code becomes both centralized and managed while also being open to the public for use as a tool for neural network simulation.

Every project created on Sourceforge is given a CVS repository for code control and versioning. One of the best sources of information on CVS in general is Open Source Development with CVS 3rd Edition, an open-source book written about using CVS [10]. Read-only anonymous public access is granted to repositories as well as web-based access. For anonymous access, pserver is used with a name of anonymous@cvs.sf.net and a blank password. Since the pserver method of authentication sends a password in plain text, this type of access is not used for developer access. Web-based access is proved using ViewCVS, an open source project hosted on Sourceforge. Developers on a project are as a default granted permission to commit changes to that project's repository. SSH is used for authentication to the CVS repository. All commands to the CVS repository must be prefaced with login. Any combination of CVS client and SSH client can be used to connect to Sourceforge's CVS; there are tutorials available on Sourceforge for using Tortoise CVS and PuTTY in windows.

Sourceforge's CVS does not provide interactive login. All CVS commands must be sent individually to the CVS server, and any file removal requires the aid of Sourceforge staff. Also available, the daily changes to the repository are compressed and put into a tar archive each night. Under the project admin page, there is a link to download the nightly tar file of the CVS tree.

7. Conclusions

Working with the Agent Based Neural Network Simulator has been both challenging and enjoyable. There is a broad base of knowledge on which the simulator draws to be a successful tool. The current implementation draws on concepts from fields such as artificial and biological neural networks, psychology, and network theory. The ABNNSim also acts as a very flexible framework for simulation development and through object-oriented programming techniques is easily modified to adapt to new ideas and concepts. In this way, the ABNNSim is very much a tool where the knowledge one brings to the simulation defines the results one will draw from the data. This framework is easily enhanced and augmented to create fuller simulations, as shown in two examples of adding distanced based wiring preference to network creation. The power of the ABNNSim as a simulator also comes from the functionality of the Repast toolkit. The new features and libraries available in Repast3.0 will help to develop more complete simulations using agent-based techniques. Understanding the concepts and theories behind the simulator and also the technologies used in the simulator's implementation gives one the most effective command over ABNNSim as a framework for simulation development.

8. Sources

1. Schoenharl, "An Agent Based Modeling Approach For The Exploration Of Self-Organizing Neural Networks"

2. Strogatz and Watts, "Collective Dynamics of 'Small-world' Networks"

3. Massimo Marchioria and Vito Latora; "Harmony in the small-world"

4. http://www.idsia.ch/NNcourse/linear2.html

5. http://en.wikipedia.org/wiki/Sigmoid_function

6. Szirtes, Palotai, and Lorincz; "HebbNets: Dynamic network with Hebbian Learning rule"

7. Szirtes, Palotai, and Lorincz; "Emergence of scale-free properties in Hebbian networks"

8. MSDN .Net Developer Center; http://msdn.microsoft.com/netframework/gettingstarted/default.aspx

9. http://sourceforge.net/docman/display_doc.php?docid=6025&group_id=1

10. http://cvsbook.red-bean.com/OSDevWithCVS_3E.pdf