Report 1: Artificial Neural Networks

Eric Scott, supervised by Roy Villafañe, Andrews University

February 02, 2009

For CPTR 495, Independent Study in Computational Intelligence

1 Complex Adaptive Systems

I am coming to the study of Computational Intelligence (CI) technology with the understanding that it is integrally related to the more general field of Complex Adaptive Systems (CAS).

The new and interdisciplinary field of complex systems holds that "the 'take home message' of the lessons from the history of science is that methodological reductionism, the analytical decomposition of structures to parts, should be completed by searching for organizational principles, too."[1] These "organizational principles" of nature include a wide variety of phenomena, drawing heavily from the study of nonlinear dynamics, chaos, emergent properties, game theory, and agent-based systems, and finds applications from physics to biology to the social sciences.

Complex systems takes a top-town, generalist approach to try and shed light on the macroscopic dynamics that result from the interaction of microscopic parts. "At the most basic level, the field of complex systems challenges the notion that by perfectly understanding the behavior of each component part of a system we will then understand the system as a whole."[2]

John Holland describes a complex *adaptive* system as one that displays "coherence under change." The details change – the people in the city change, the antibodies in your body change – but, he asserts, "your immune system

[1] Péter Érdi, *Complexity Explained* (Berlin: Springer, 2008), 24

[2] John H. Miller and Scott E. Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life* (Princeton University Press, 2007), 3.

is coherent enough to provide a satisfactory scientific definition of your *identity*".[3]

The two behaviors of a CAS, adaptation (*Learning*, if you will) and coherence (*Robustness* in the face of change) make them a very attractive candidate for imitation in artificial systems. Indeed, while computational models have proven crucial to the exploration of complex systems in nature, those selfsame natural systems have served as inspiration for the development of more sophisticated computational paradigms (As presented by Nancy Forbes in *Imitation of Life: How Biology is Inspiring Computing* (MIT Press, 2004)).

As the lines between natural phenomena and machine blurr, some advocate the viewing of computation as a property of the natural world.[4] The new nature-inspired tools are again being used in computational models to explore complex systems in nature, and the cycle continues in a two-way relationship.

2 Computational Intelligence

These new tools form the field of Computational Intelligence, which is the focus of this study. An extension (some say successor) to AI, CI can be taken to include the subareas of artificial neural networks, evolutionary computation, swarm intelligence, artificial immune systems, fuzzy systems, and an extensive array of hybrid systems that combine two or more of these solutions, playing on the strengths and weaknesses of the various approaches. These are the topics covered by my texts, and I was pleased to see that my secondary text – *Computational Intelligence: An Introduction* by Engelbrecht – is used in the pioneering CI course at the Missouri University of Science and Technology.[5]

[3] John Holland, *Hidden Order: How Adaptations Build Complexity* (Cambridge: Perseus Books, 1995), 1-4.

[4] Colin Johnson, "Teaching Natural Computation", *Computational Intelligence Magazine*, vol. 4 no. 1, February 2009, 24-30.

[5] Ganesh K. Kumar Venayagamoorthy, "A Successful Interdisciplinary Course on Computational Intelligence", *IEEE Computational Intelligence Magazine*, vol. 4 no. 1, February 2009, 14-23.

3 Artificial Neural Networks

Artificial Neural Networks (ANNs) find their inspiration in a simplified model of neurons in the brain. Donal Hebb established in 1943 that a network of these mathematical neurons can "learn" when exposed to data, and Hava Siegalmann proved that an ANN has the full computational capacity of a Universal Turing Machine. The development of further learning algorithms – most notably the concept of backpropagation independently invented by Werbos and Parker in 1974 and 1982, respectively – have made ANNs a powerful tool and highly active area of research.

3.1 ANN Basics

An ANN is represented as a digraph. The information, the "code" or "knowledge", is stored in the edges of the graph which connect each neuron. This is similar to the width of synapses in the brain, which determines how strong the output signal of a neural is when it arrives at its target. When a network is trained (Which entails exposing it to massive amounts of example input until it learns how to produce the desired output), it is the weights on these synapses that are modified until satisfactory performance is acheived.

A basic ANN design is the Feed-Forward Neural Network (FFNN), in which *no directed cycles are allowed*. In a simple version of this template, each neuron of the first layer feeds into every neuron of the second layer, and so on. I assume this layout below in the section on back-propagation, but be aware that FFNNs can have somewhat more intricate designs. For example, connections can exist directly between the input layer and the output layer. An alternative design strategy is the *recurrent* network, in which cycles (feedback loops) *are* allowed, which offers more learning capacity in exchange for being more difficult to train.

3.2 Knowledge-Based Design

It should be noted that a trained ANN is an extremely complex system. Their usefulness comes from our ability to exploit self-organizing principles via the training process. Reverse engineering the weights to extract this knowledge in human-understandable terms once they are set, however – especially in recurrent networks – is an intractable problem for more than a handful of neurons.

An alternative to raw *knowledge extraction* is *knowledge-based design*, in which we design the network with comprehensibility in mind by considering "crude domain knowledge to generate the initial network architecture, which is later refined in the presence of training data."[6] The implementations I have seen so far tend to use elements of fuzzy systems to combine the learning power of ANNs with the lucidity of *fuzzy rules* (Which I will study later this semester).

3.3 Activation Functions

Neural networks are an implentation of a "connectionist system," that is, a massively parallel processing system.[7] They are generally simulated on a conventional computer, but we mustn't forget that they represent an entirely different architecture.

The heart of a neuron's processing power is its activation function. Neurons can be defined as either summation units (SU), where the sum of incoming signals is passed to the activation function, or as product units (PU), in which the product of the signals is passed. i.e.,

$$net_{SU} = (\sum w_i f_i) - \theta \quad (1)$$

$$net_{PU} = (\prod w_i f_i) - \theta \quad (2)$$

where w_i is the weight of the incoming edge (synapse), f_i is the output signal of the neuron associated with that edge, and θ is a constant.

An even wider variety of activation functions exists, which define the output signal of the neuron based on its net input. Activation functions generally follow one of the following forms: linear, step (Binary/Threshold), ramp (Linear with a max and min), sigmoid (Logistic), hyberbolic tangent, or gaussian. Which function is preferred depends on the application. The vibe I'm getting from my reading so far is that application of CI tools is as much an art as it is a science.

[6] Sushmita Mitra and Yoichi Hayashi, "Neuro-Fuzzy Rule Generation: Survey in Soft Computing Framework", *IEEE Transactions on Neural Networks*, vol. 11 no. 3, May 2000, 748-768.

[7] Linda Null and Julia Lobur, *The Essentials of Computer Organization and Architecture* (Boston: Jones and Bartlett, 2003), 439.

3.4 Gradient Descent and Backpropagation

Backpropagation is one of the most common ways to train an ANN. It is an example of a *supervised* training model, in which example answers are provided for the network to conform to, as opposed to *unsupervised* learning, in which a network learns to classify inputs into different categories without being trained explicitly. Simple, two-layer feed-forward networks can be trained fairly effectively with such methods as Hebbian learning (Which is unsupervised), but adding hidden layers improves learning capacity while decreasing learning efficiency. Backpropagation made the problem of training a network's hidden layers tractable by providing a "method for calculating derivatives exactly and efficiently." [8]

The basic idea is that we define an error function that tells us how much each of the output neurons differs from their intended value, then send this error signal backwards through the network so each weight has an idea of just how wrong they are. Then we use calculus to compute how best to individually adjust each weight to improve the output. This one of the primary motivations for using continuous activation functions instead of discrete "step" functions: calculus provides an easy way to implement the gradient descent paradigm for greedy function optimization. By setting the initial weights randomly and then running this backpropagation algorithm several times, local minima can be discovered in the network's error.

My texts showed enough of the mathematics involved to facilitate implementation, but not enough to show why propagating the error signals back actually works. The following represents the work I had to do to make the matter clear to myself, and to show that the entire process is a straightforward dynamic programming implementation of gradient descent equations.

The overall error of the ANN's output can be respresented with the sum squared error of the output neurons:

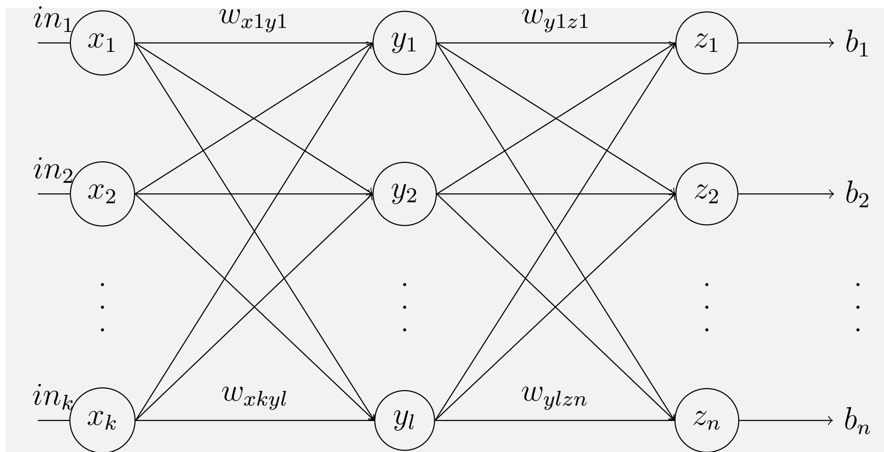
$$\varepsilon = \frac{1}{2} \sum_{k=1}^{m} \sum_{j=1}^{n} (b_{kj} - z_{kj})^2 \tag{3}$$

where n is the number of neurons in the output layer, b is the training answer, and z is the output of the neuron's activation function. It's often advantageous to run multiple trials before applying weight corrections, so

[8] Paul Werbose, "Backpropagation Through Time: What It Is and How To Do It," *Proceedings of the IEEE*, vol. 78, no. 10, Oct. 1990.

that a higher variety of samples are tested before we start pushing the weights in a specific direction. m, then, is the number of separate training examples we've selected for this training run. Alternatively, weight changes can be applied after each sample if the learning rate (step size) η is sufficiently small. In the following we will consider only the case of $m = 1$, but it is straightforward to extend the equations to $m > 1$.

Figure 1: A basic Feed-Forward Neural Network (FFNN)



With the error defined, we can now apply gradient descent to find the best change in the weights:

$$w'_{ij} = w_{ij} - \eta \frac{\partial \varepsilon}{\partial w_{ij}} \tag{4}$$

The tricky part is computing this partial for an arbitrary synapse. As a warm-up problem to begin exploring what we're trying to do, take the simple case of altering a synapse w_{qr} connecting a neuron y_q in a hidden layer to z_r in the output layer. Since z_r is the only output neuron that is changing, ε only varies as a function of z_r, i.e. $\frac{\partial \varepsilon}{\partial w_{qr}} = \frac{\partial \varepsilon_r}{\partial w_{qr}}$, where $\varepsilon_r = \frac{1}{2}(b_r - z_r)^2$. Now the

chain rule of partial differentiation gives us:

$$\frac{\partial \varepsilon_r}{\partial w_{qr}} = \frac{\partial \varepsilon_r}{\partial z_r}\frac{\partial z_r}{\partial w_{qr}} \tag{5}$$

$$= \frac{\partial}{\partial z_r}(\frac{1}{2}(b_r - z_r)^2)\frac{\partial z_r}{\partial w_{qr}} \tag{6}$$

$$= (b_r - z_r)\frac{\partial z_r}{\partial w_{qr}} \tag{7}$$

Note that if we had simply used the difference $\sum_j (b_j - z_j)$ as our ε instead of the SSE, we would lose some useful information (Namely the size of the error margin) since $\frac{\partial \varepsilon_r}{\partial z_r}$ would reduce simply to -1. I find this an interesting illustration of how squared errors are important.

Now, since z_r is the activation function $z_r = f_{zr}(net)$, we have:

$$\frac{\partial \varepsilon_r}{\partial w_{qr}} = (b_r - z_r)\frac{\partial f_{zr}(net)}{\partial w_{qr}} \tag{8}$$

$$= (b_r - z_r)\frac{\partial f_{zr}(net)}{\partial net}\frac{\partial net}{\partial w_{qr}} \tag{9}$$

Assuming summation units, we can use equation (1) to simplify this as follows. Note how much more complex this would be if we used product units.

$$\frac{\partial \varepsilon_r}{\partial w_{qr}} = (b_r - z_r)\frac{\partial f_{zr}(net_{SU})}{\partial net_{SU}}\frac{\partial net}{\partial w_{qr}} \tag{10}$$

$$= (b_r - z_r)\frac{\partial f_{zr}(net_{SU})}{\partial net_{SU}}\frac{\partial}{\partial w_{qr}}((\sum_{i=1}^{l} w_{ir}y_i) - \theta) \tag{11}$$

$$= (b_r - z_r)\frac{\partial f_{zr}(net_{SU})}{\partial net_{SU}}y_q \tag{12}$$

Finally, we define $\delta_{zr} = (b_r - z_r)\frac{\partial f_{zr}(net_{SU})}{\partial net_{SU}}$ as the "error signal," the part of $\frac{\partial \varepsilon_r}{\partial w_{qr}}$ that can be passed backwards through the network to help define deeper weight adjustments. Our general equation for backpropagation across summation units is then

$$\frac{\partial \varepsilon_r}{\partial w_{qr}} = \delta_{zr}y_q \tag{13}$$

The error signal δ is propagated backwards through the network across the synapses via a summation function, allowing us to use the results of subproblems (higher level weight changes) to solve the equations for deeply nested

neurons. For example,

$$\delta_{yq} = \frac{\partial f_{yq}(net_{yq})}{\partial net_{yq}} \sum_{j=1}^{n} \delta_{zj} w_{qj} \tag{14}$$

To see why this is done we need to extend our mathematical inquiry deeper into the network.

In the derivation of equation (13) we made the simplifying assumption that the synapse in question is directed into the output layer. Now take the case of a weight w_{pq} connecting a neuron x_p in the input layer to y_q. By affecting y_q, all neurons the output of y_q is connected to are effected, making our "chain" of differentiation substantially larger. We will take y_q to be connected to every neuron in the output layer, $z_j, 0 \leq j \leq n$. Proceeding similar to the above, we now have:

$$\frac{\partial \varepsilon}{\partial w_{pq}} = \frac{\partial}{\partial w_{pq}} \sum_{j=0}^{n} \frac{1}{2}(b_j - z_j)^2 \tag{15}$$

$$= \sum_{j=0}^{n} \frac{\partial \varepsilon_j}{\partial w_{pq}} \tag{16}$$

$$= \sum_{j=0}^{n} \frac{\partial \varepsilon_j}{\partial z_j} \frac{\partial z_j}{\partial y_q} \frac{\partial y_q}{\partial w_{pq}} \tag{17}$$

$$= \frac{\partial y_q}{\partial w_{pq}} \sum_{j=0}^{n} (b_j - z_j) \frac{\partial f_{zj}(net_{zj})}{\partial y_q} \tag{18}$$

$$= \frac{\partial f_{yq}(net_{yq})}{\partial net_{yq}} \frac{\partial net_{yq}}{\partial w_{pq}} \sum_{j=0}^{n} (b_j - z_j) \frac{\partial f_{zj}(net_{zj})}{net_{zj}} \frac{\partial net_{zj}}{\partial y_q} \tag{19}$$

Assuming summation units:

$$\frac{\partial \varepsilon}{\partial w_{pq}} = \frac{\partial f_{yq}(net_{yq})}{\partial net_{yq}} \frac{\partial net_{yq}}{\partial w_{pq}} \sum_{j=0}^{n} (b_j - z_j) \frac{\partial f_{zj}(net_{zj})}{net_{zj}} \frac{\partial net_{zj}}{\partial y_q} \tag{20}$$

$$= \frac{\partial f_{yq}(net_{yq})}{\partial net_{yq}} \frac{\partial (\sum_h x_h w_{hq})}{\partial w_{pq}} \sum_{j=0}^{n} (b_j - z_j) \frac{\partial f_{zj}(net_{zj})}{net_{zj}} \frac{\partial (\sum_i y_i w_{ij})}{\partial y_q} \tag{21}$$

$$= \frac{\partial f_{yq}(net_{yq})}{\partial net_{yq}} x_p \sum_{j=0}^{n} (b_j - z_j) \frac{\partial f_{zj}(net_{zj})}{net_{zj}} w_{qj} \tag{22}$$

$$= \frac{\partial f_{yq}(net_{yq})}{\partial net_{yq}} x_p \sum_{j=0}^{n} \delta_{zj} w_{qj} \tag{23}$$

$$= \delta_{yq} x_p \tag{24}$$

Once the δ values have been recursively propagated through the network by the algorithm implied by equation (14), the relationship demonstrated by (13) and (24) can be used in (4) to calculate optimal weight changes. Note that this solution would not be upset by weights that jump layers, for example connecting neurons in x to the output layer.

As far as solutions to specific activation functions, my texts seem to prefer the logistic function, which has a simple derivative, but I've been using the closely-related hyperbolic tangent to test my code. We derive the δ for $z_r = tanh(net_{SU}) - \theta$ as follows.

$$\frac{\partial \varepsilon_r}{\partial w_{qr}} = \delta y_q \tag{25}$$

$$\delta = \frac{1}{y_q} \frac{\partial \varepsilon_r}{\partial z_r} \frac{\partial f_{zr}(net)}{\partial net} \tag{26}$$

$$= (b_r - z_r) \frac{\partial}{\partial net} (tanh(net) \tag{27}$$

$$= (b_r - z_r)(1 - tanh^2(net)) \tag{28}$$

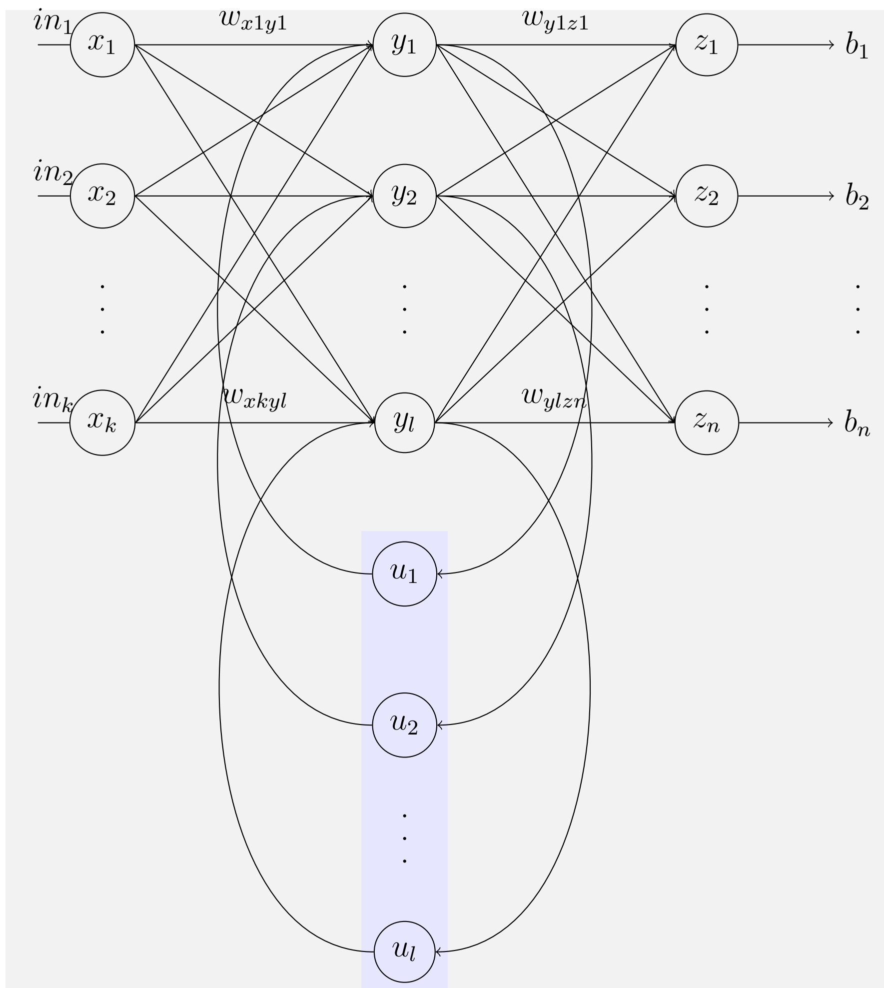
$$= (b_r - z_r)(1 - z_r^2) \tag{29}$$

3.5 Recurrent Networks

As aforementioned, a Recurrent Neural Network (RNN) allows synapses to loop back onto neurons in previous layers. This creates a temporal nature, as

past signals effect present ones. My texts barely scratch the surface of RNNs, containing a grand total of three pages on them, most of which are taken up by large figures. Engelbrecht presents two example Simple Recurrent Neural Networks (SRNNs). One of these, the Elman SRNN, uses an extra hidden layer ("context units") to form a feedback loop (See figure 2). The weights connected to the context layer are set at 1, so they provide a straightforward temporal memory for the network.

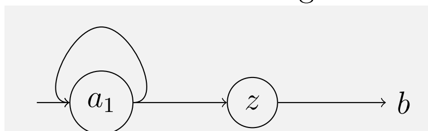
Figure 2: The Elman SRNN



Now that I understand that the backpropagation algorithm for FFNNs is a straight-forward application of gradient descent, I can apply similar princi-

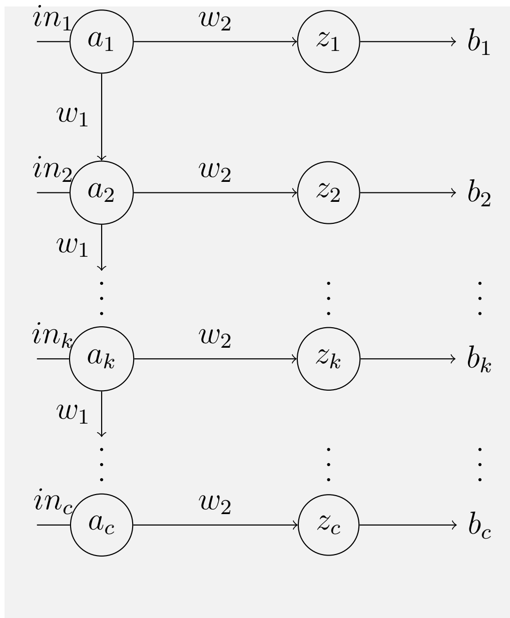
ples to explore RNNs and demonstrate their difficulties for myself. Lacking introductory literature on the topic, I scratched out the following basic RNN on a napkin over Chinese food before attempting to take on the Elman SRNN (See figure 3).

Figure 3: The RNN from my Napkin



In trying to apply gradient descent to this kind of network, I constructed the graph in figure 4, which is equivalent to figure 3 if c equals the number of discrete time steps entered into the recurrent network. Even though figure 4 is somewhat complicated, it has no cycles, and thus is an FFNN and can be trained with backpropagation. Note that some awkwardness arises since there are actually only two weights involved despite the large number of edges. When weight changes are applied, an entire group of edges are modified.

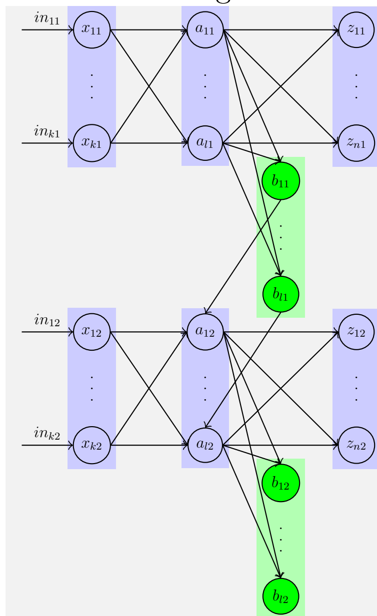
Figure 4: Unfolding of figure 3



A natural question to ask at this juncture is whether an arbitrary RNN – such as the Elman RNN – can be converted into a FFNN and vice versa. This is reminiscent of the relationship between deterministic and nondeterministic finite automata, where the former can be thought of as an expanded version of the latter. As already mentioned, my texts (And Wikipedia, for that matter), only provide a very light treatment of RNNs, and the answer to this question has eluded me so far. It seems intuitively plausible to me that the answer is yes, and that any RNN can be converted to an FFNN, but I've found no statements or proofs of this.

To explore this concept of converting between ANN types further, we examine the Elman RNN in figure 2. It is straightforward to show that figure 5 is an equivalent FFNN for a finite number of time steps (In this case $c = 2$). Note how the context layer (in green) carries information between quantums.

Figure 5: Unfolding of figure 2 for two time steps



Note also that the structure of this FFNN is a function of the number of time steps in the input. We call the process of unfolding the RNN into an FFNN for a specific number of time steps and then training it Backpropagation Through Time (BPTT). Clearly this can be a very slow process, as the number of relationships to be considered in the FFNN version is immense. As

a result, Genetic Algorithms and Particle Swarm Optimization are popular alternative training methods to BPTT for RNNs (We will discuss alternative optimizatin paradigms to gradient descent in report 2).

3.6 Code Status

I've put together an ANN framework in C#, trying to keep it dynamic enough to tinker with different kinds of networks. It uses linked structures to represent the graph (As opposed to the adjacency matrices I used in a previous C framework, which allowed for some clever matrix multiplication but overall made for confusing arithmetic). Some modifications will be required to provide support for RNNs. I have not yet implemented backpropagation, having been anxious to get on to genetic algorithms (Which will be the topic of my next report), but the reasoning I went through while working through the above will make it a straight-forward process to write the code at my leisure.

4 Time Spent

Date	Time Spent	Task
12/04/08	10:00	Wrote a basic neural network simulator in C (For CPTR 487 – time spent not counted towards independent study)
12/19/08	01:30	Explored parallel design strategies, reading about parallel algorithms in my CPTR 276 book, tinkering with Erlang, and testing multiple threads via C#/Mono on edmund.cs.andrews.edu. I intend to make a multithreaded CI framework.
12/13/08	03:00	Skimmed textbooks and recent literature getting a feel for strategic approaches to evolutionary computation, such as co-evolution and methods for dealing with multi-objective functions, dynamic environments, etc.
12/21/08	04:00	Began designing a framework to implement neural networks and genetic algorithms together, writing a functional neural network simulator in C# (Still lacking training algorithms).
12/24/08	01:00	Read the first chapter of Holland's Hidden Order, in which he presents Complex Adaptive Systems.
12/25/08	01:00	Read the genetic algorithm section of Eberhart, ch. 3.
12/26/08	04:00	Began GA framework (C#), deriving and implementing an inverse Gray code function.
12/27/08	03:00	Fleshed out the GA with rough draft crossover, roulette selection, and population initiation. About 40% done.
12/28/08	02:00	Read about backpropagation, and turned my attention to preprocessing, skimming the chapter on attribute (feature) extraction in Witten, and reading the section on the Fast Fourier Transform in my book from CPTR 276.

Date	**Time Spent**	**Task**
12/29/08	03:00	Continued trying to fully understand backpropagation, deriving error signal differentials for tanh(x) (The texts only have derivations for the logistic function). Created Course Plan PDF.
12/30/08	00:30	Created outline for report 1
01/14/09	05:00	Implemented mutation and began tinkering with different parameters to solve a toy problem. Results are poor pending implementation of crossover (?).
01/22/09	00:30	Read Ganesh Venayagamoorthy, A Successful Interdisciplinary Course on Computational Intelligence, *IEEE Computational Intelligence Magazine*, vol. 4 no. 1, February 2009, 14-23.
01/22/09	02:15	Began drafting report 1
01/27/09	05:00	Worked on report 1 / backpropagation derivation
01/30/09	04:00	Worked on report 1 / General derivation of backpropagation
01/30/09	00:30	Read an article on Complex Systems and Neural Nets
01/30/09	05:00	Learned TeX graphics with Tikz / Explored backpropagation on non-feed-forward networks
01/31/09	04:00	Explored RNNs further / Updated report 1
For this Report:	49:15	
To Date:	49:15	