Evolution of Neural Networks Using Genetic Algorithms

Wesley Kerr, Suranga Hettarachchi
University of Wyoming
Laramie, WY 82071
wkerr@cs.uwyo.edu

1 The Task

RoboCup has come a long way since it's creation in '97 [1] and is a respected place for machine learning researchers to try out new algorithms in a competitive fashion. RoboCup is now an international competition that draws many teams and respected researchers looking for a chance to create the best team. Originally we set out to create a team to compete in RoboCup. This was an ambitious project, and we had hopes to finish within the next year. For this semester, we chose to scale down the RoboCup team towards a smaller research area to try our learning algorithm on. The scaled down version of the RoboCup soccer environment is known as the "Keepaway Testbed" and was started by Peter Stone, University of Texas [2]. Here the task is simple, you have two teams on the field each with the same number of players. Instead of trying to score a goal on the opponent the teams are given tasks, and one team is labeled the *keepers* and the other is labeled the *takers*. It is the task of the *keepers* to maintain possesion of the ball and it is the task of the *takers* to take the ball. The longer the keepers are able to maintain possesion of the ball the better the team.

There are several advantages to this environment. First, it provides some of the essential characteristics of a real soccer game. Typically it is believed that if a team is able to maintain possesion of the ball for long periods of time they will win the match. Secondly, it provides realistic behavior much the same as the original RoboCup server. This is accomplished by introducing noise into the system similar to the original RoboCup, and similar to what would be received by real robots. Finally, when you want to go through the learning process this environment is capable of stopping play once the *takers* have touched the ball, and the environment is capable of starting a new trial based on that occurrence.

Although the RoboCup Keepaway Machine Learning testbed provided an excellent environment to train our agents, we still needed to scale down the problem in order to do a feasibility study. Based on the Keepaway testbed, we created a simulation world with one simple task. One agent is placed into the world and has to locate the position of the goal. This can be thought of as an agent in a soccer environment needing to locate either the ball or another teammate. It was in this environment where we tested our methods for learning autonomous agents.

2 Description of Simulated World

We created a simulated world for our agent to learn the task of location of a goal. A visualization of the simulated world can be seen in figure 1. The world is defined by:

1. Walls defined by two points p_1 and p_2
    - w_{x1} - p_1 x-coordinate

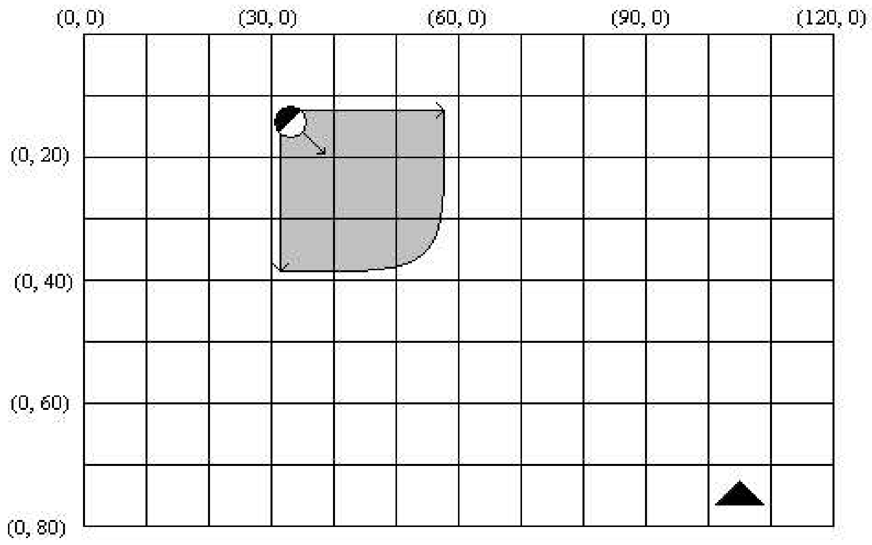

Figure 1. An idealized view of the simulator.

- w_{y1} - p_1 y-coordinate
- w_{x2} - p_2 x-coordinate
- w_{y2} - p_2 y-coordiante

2. Agent
   - r_x - x-coordinate of the agent
   - r_y - y-coordinate of the agent
   - θ - angle the agent is facing relative to x-axis
3. Goal
   - g_x - x-coordinate of the goal
   - g_y - y-coordinate of the goal
4. View Frustum
   - $\theta + / - 45°$

The agent's effectors are:

1. turn(θ) - turns the agent θ degrees
2. move(v) - moves the agent along current angle with velocity v

The agent's sensors are:

1. position - returns the agents current position r
2. angle - returns the agents current angle θ

3. goal location - returns the goal location if within the view frustum, otherwise returns $g_x = -1$ and $g_y = -1$

4. wall locations - returns the locations of all of the walls

The realized version of the simulator can be seen in figure 2. This is what is viewable by the user to monitor the performance of the agent. The agent uses its sensors and effectors to navigate the simulated world until it locates the goal location. The simulated world is a real valued world, therefore the agents positions will always be real valued. Now that we have created a simulated world to experiment in we need to determine a method of learning how to navigate the world successfully.

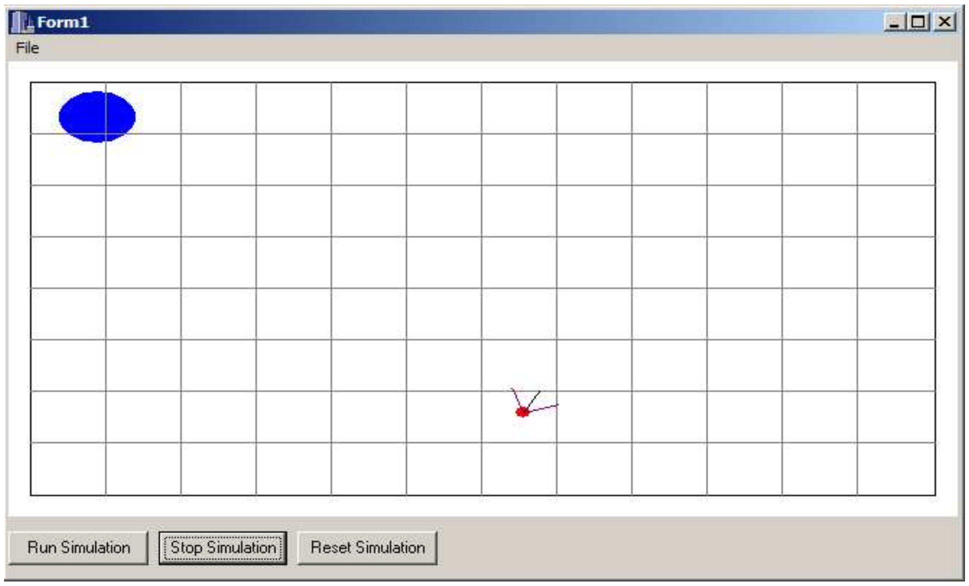

Figure 2. A view of the actual simulator.

3 Methods

Peter Stone has used neural networks successfully in the RoboCup environment to learn low level behaviors, and then used a decision tree algorithm to determine what behavior to execute given the current situation of the game[3]. We sought to extend this research and have a fully encompassing neural network capable of making the decisions on what actions to take as well as deciding the parameters for that action.

We created a feed-forward neural network of several layers in order to accompmlish the given task. The traditional technique for learning the weights in a neural network has been back-propogation. We decided that since genetic algorithms have been used successfully to solve optimization problems, and one could view the weights connecting neural networks as something that needs to be optimized, we would use genetic algorithms in order to solve the task of learning the weights for the neural network. We originally planned to use the genetic algorithm for determining the optimal topology of the neural network as well, but that was never implemented. Learning the topology was the primary reason for choosing genetic algorithms on both levels. Also, we hoped that since the genetic algorithm *feels* its way around the search space it might be capable of finding a solution faster than back-propogation.

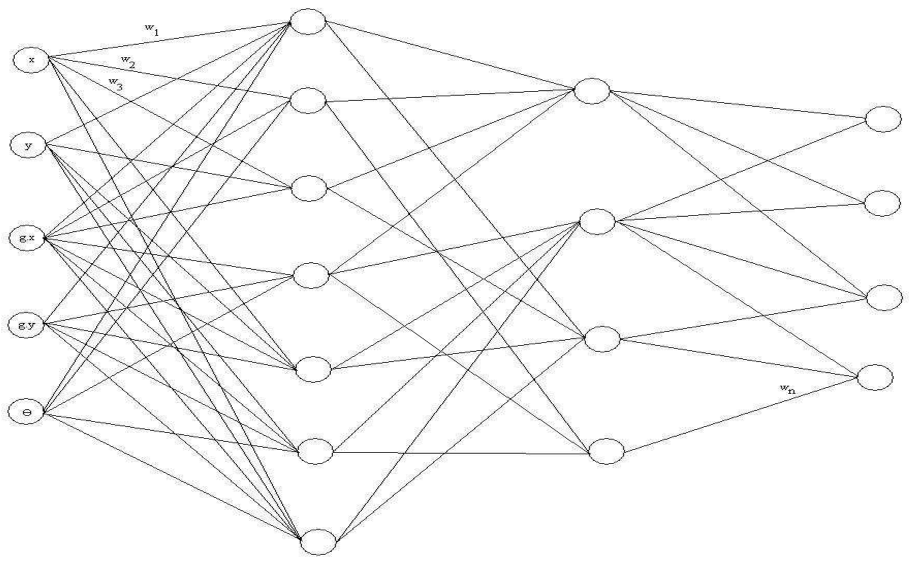

Figure 3. A view neural network design.

4 Neural Network Design

The original intention of the project was to use a genetic algorithm to learn the topology of the neural network, and therefore remove the creation of the topology for the neural network from the designers. In order to test the internal genetic algorithm used to learn the weights of the neural network, an arbitrary neural network was designed. Figure 3 shows the topology chosen for the original neural network. We designed a neural network with one input layer, two hidden layers, and one output layer. The inputs to the neural network consist of most of the information attainable from the simulated world.

Inputs:

1. r_x - x location in simulator
2. r_y - y location in simulator
3. g_x - x location of goal in simulator
4. g_y - y location of goal in simulator
5. θ - current heading

There are four outputs from the neural network. Let o_i represent the output value for $1 \geq i \leq 4$, and let V be the maximum velocity that the agent can move with. We chose to cap the maximum velocity because in both real robots and the RoboCup simulation environment agents have a maximum speed. The first two outputs are used to determine what action to perform, turn or move. If $o_1 > o_2$ the agent would move forward with velocity $v = o_3 \times V$. Since we are using the sigmoind function for determining activation levels, v will always be a value between 0 and V. Otherwise, the agent would turn with angle $\theta = o_4 \times 360°$. Again since we are using the sigmoid function θ will always range between 0 and $360°$. Now all that is left is how we plan to teach this neural network to navigate the simulated world by altering the weights between connections in the neural network.

5 Genetic Algorithm Design

In order to design a genetic algorithm to learn the weights for the given neural network, we have to answer several questions. How are we going to represent the data? How are we going to measure the fitness of the individual in the population? How are we going to select individuals for the next population? How are we going to change the population?

We decided to represent each individual in the population as a string of real numbers. The length of the string is equal to the number of connections in the neural network. Figure 3 shows the network weights and their indexes which correspond to their index into the individual's string. So, our individual is a string $w_0, w_1, \ldots, w_n$ where n is the number of connections in the neural network. The fitness of a given individual was determined by running it through the simulation a given number of times. This is a number that can be varied since we randomly place the agent and goal each time in the simulation, but typically we ran through the simulator 50 times before determining the fitness of the individual. The agent was allowed to move until it found the goal or a maximum number of iterations was surpassed, typically 1000 moves. In order to make the solution a maximization problem, we kept track of how many iterations it took to find the goal, μ. We also could calculate the minimum number of steps it would take to find the goal, σ. Let N be the number of trial runs through the simulator. We found the average fitness over the runs to be the fitness for the individual as shown below:

$$\frac{\sum^{N} \frac{\sigma}{\mu}}{N}$$

Because of our definition of the fitness of the individual, the better the agent got at find the goal the closer the fitness was to one, and less fit individuals were closer to zero.

Individuals were selected for the next population by stochastic universal sampling. The idea behind this being the most fit individuals were probabilistically more likely to have children than those less fit individuals. We also used an elitist scheme which maintains the best individual found so far as the first individual of the population. We chose to use the elitist scheme in order to make sure that the best individual was always capable of influencing the other individuals when it came time for the new population. Finally, we modify the next generation by combining both recombination and mutation in order to introduce variety into the population. Individuals are selected randomly for recombination and we perform 1-point crossover by selecting a random point between 1 and n. Everything past the crossover point is swapped between the two selected individuals.

The neural network is read in and the genetic algorithm is initialized to a population of 200 individuals. For the given neural network each individual has a length $n = 76$. Each index in the individual is initialized to a random number between -6.0 and 6.0. The recombination rate for the genetic algorithm is set at 0.6 and the mutation rate is 0.01. Now that we have a complete design we can design the experiment and see how well it performs.

6 Experimental Results

The gentic algorithm was allowed to run for 1500 iterations and the learning curve can be found in figure 4. The running time of the algorithms for 1500 iterations was roughly three days. Therefore we only ran one experiment in order to generate the average fitness curve. Since the genetic algorithm is a stochastic algorithm, more runs would be needed to determine an average fitness over an average of the runs, but lack of time and since the project has been halted, this was not explored. The best individual in the population was able to find the goal in 6.25 times the minimum number of steps and had a fitness of 0.16. One thing to note about this learning curve is that the genetic algorithm was capable of learning the weights for the neural netowrk, and the average fitness typically peaked fairly early in the life-cycle. This was not the only run of the genetic algorithm though.

The original run of the algorithm found a fitness of 0.32. Unfortunately, the parameter settings have since been lost and we have been unable to duplicate those results. Other combinations besides those listed above

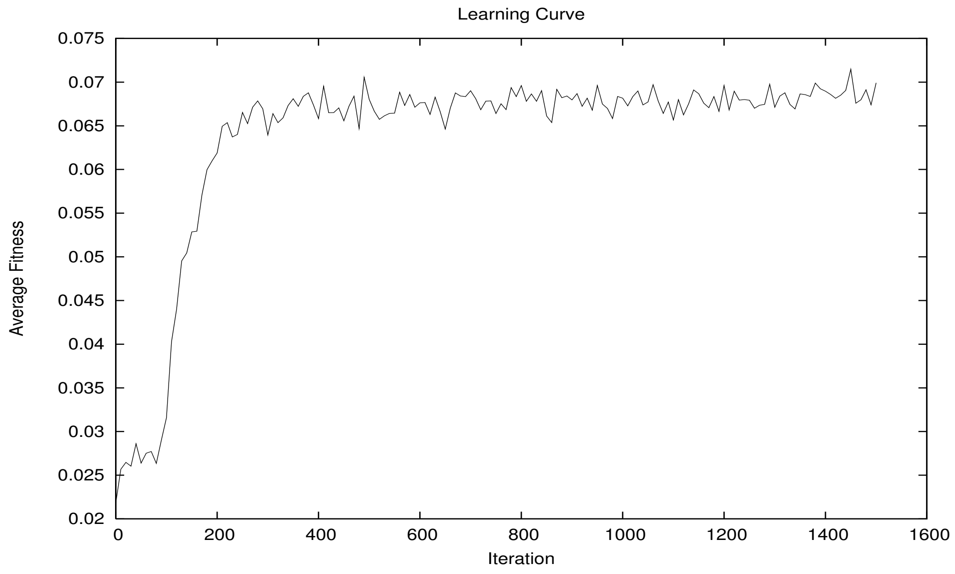

Figure 4. Learning curve for the genetic algorithm.

were tried in order to achieve a higher fitness. Instead of 1-point recombination, another form of recombination known as uniform recombination was used with limited success. Uniform recombination evaluates every index in the individual and essentially evaluates a coin flip to determine if it will swap that index between individuals. Modifications were made to the mutation rate were made. Above we have listed that the mutation rate was 0.01, but in other runs the mutation rate was reduced and increased. All of this was trying to determine what settings yield the best results for our problem, but all of these changes were unable to duplicate the initial results. The most suprising fact of all is that even though 0.16 and 0.32 seem such small fitnesses, actual intelligence is visible when watching the agent in the system. In fact, both solutions appear to have the same solution strategy even though they have differing weights. It's quite facinating to watch. The agent has a plan for solving the problem and follows that plan until completion unless it runs into a wall. This led to the thought that we could use wall positions as inputs into the neural network, although this was never implemented. Overall, the agent can successfully navigate to the goal every time without the presence of walls and appears to be intelligent about how it gets there.

7 Conclusions

When we started the project we set out to create a team to compete in RoboCup. We believed our approach to be a novel one in this arena, and very capable of competing with the current champions. Unfortunately, the discovery of a paper by Peter Stone[3], showed that this approach had already been documented, with greater enhancements than those we would be providing. At this point the research was halted and other topics were chosen to explore. Eventually we would like to revisit this problem, possibly reperforming the layered-learning technique described by Stone.

References

[1] I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multi-agent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.

[2] P. Stone and R. Sutton. Keepaway soccer: a machine learning testbed. In *RoboCup-2001: Robot Soccer World Cup V*. Springer-Verlag, 2001.

[3] S. Whitestone and P. Stone. Concurrent layered learning. In *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, July 2003.