Design of an Intelligent Traffic Light System

Andrew Bradley

March 30, 2004

Abstract

Currently, traffic lights follow hard-coded sequences for directing traffic. Although this pattern may change during the day, the lights do not have the ability to adjust their own sequence according to varying traffic conditions. The purpose of this project is to create a system by which traffic lights respond to changing traffic conditions. This project makes use of neural networks and genetic algorithms, allowing the lights to learn the proper way to direct traffic as the simulation runs.

1 Introduction

This project has two components–a simple traffic simulator and a decision making system for the traffic lights. Cars enter the system randomly and work their way through the simulated set of intersections, stopping when appropriate at traffic lights. The traffic lights adjust their signaling patterns according to the number of cars waiting in each direction, how long it has been since the light last changed, and the status of its neighbors. Neural networks

provide the traffic lights with brains, allowing them to make decisions. Genetic algorithms are used to choose the best performing traffic lights and create a new generation of traffic lights based on the top performers. In this way, inefficient traffic lights are eliminated from the simulation, leaving only the more efficient signals behind. As a result, the population of lights will, over time, become better at directing traffic and minimizing congestion.

2 Neural Networks Introduction

The human brain is constructed of cells called *neurons*. Each cell accepts some number of inputs and then, based on the total value of the inputs, the neuron decides whether or not to fire an output. In the brain, vast numbers of neurons are wired together, sending their outputs to other neurons, and ultimately allowing humans to make complex decisions about things. Neural networks attempt to replicate this process electronically.

2.1 Neural Network Uses

Neural networks have a variety of uses, especially in optimization problems. For example, programs have been written that handle the management of nurse shifts. Other programs solve classification problems, learning how to identify different situations and classify them. Some neural networks are applied in more light-hearted tasks, such as in the creation of a landmine collecting creature that evolves over time to become more adept at its task.

2.2 Neural Network Details

Neural networks follow the same architecture as the brain, except the neurons are represented electronically. In a biological neuron, messages are passed from cell to cell over gaps called *synapses*. Input messages then travel along *dendrites*. After the cell generates its output, an output messages is sent out along an *axon*. Just as with biological neurons, a neuron in a neural net has a set of inputs that it accepts that it then uses to calculate its output.

2.2.1 Weights

The inputs fed into a neuron are weighted; they are multiplied by numbers that determine each input's relative importance. This is similar to the way grades in classes are often determined—a 100 percent score on a test is given more weight than a 100 percent score on a quiz. It is through modifying the weights in the neurons that a neural network learns.

2.2.2 Activation Function

A neuron's decision-making process is handled by its activation function. One popular type of activation function, and the type used by this project, is a step function. It attempts to replicate the way neurons in humans and animals behave. Neurons in biological brains have two possible outputs when given input—they either fire or they don't fire. There is no "half-fire" output for a biological neuron. The step function first creates a sum, S, of all the weighted inputs, according to the following equation, where a represents the inputs and w represents the weights.

$$S = \sum_{i=1}^{n} a_i w_i$$

After calculating the sum, the step function checks to see if S exceeds a particular *threshold* value. If it does, the function outputs a one, else the function returns zero as its output. Below is an equation for this function, where "T" represents the threshold value.

$$O = \begin{cases} 0 & : \quad a \leq T \\ 1 & : \quad a > T \end{cases}$$

2.3 Perceptron Learning

The whole purpose of neural networks is to allow programs to learn. There are various methods by which networks can be trained. One of the earliest methods was the *perceptron learning algorithm*, described next. With this algorithm, the network is given test data consisting of inputs and a known output. The program then calcluates what it thinks the output should be and compares it to what it should have calculated. The weights are modified based on how far off the program was, and the process repeats until all the outputs for the test data are properly calculated. Thus, the network is trained by learning from example. Below is a psuedo-code algorithm for accomplishing this (Adapted from *Artificial Intelligence* by Elaine Rich and Kevin Knight.

1. Create a perceptron with n inputs and n weights.

2. Initialize the weights to random values.

3. Iterate through the training set, collecting all test inputs misclassified by the current set of weights.

4. If all examples are correctly classified, the weights are correct. Exit the program.

5. Otherwise, compute a vector sum S of misclassified inputs. If an input X incorrectly failed to fire the neuron, add X to S. If X incorrectly fired the neuron, add -X to S. Multiply S by some arbitrary value known as the scale factor.

6. Modify the weights by adding the elemnts of vector S to them. In other words, Add S_i to W_i.

7. Return to step three.

2.3.1 Scale Factor

The scale factor described above is a value used to determine how much how much to change the value of the weights after each iteration. The weights are modified by the vector S, therefore, multiplying S by a smaller scale factor will cause more gradual changes in the weights, and probably slower learning. Using a scale factor that is too large, however, can cause the weights to change too dramatically to reach an optimum value.

Notes One problem arose while testing out this algorithm. Sometimes, the weights would come out such that all of the calculated outputs were too low. After calculating the vector S, multiplying this by the scale factor, and adding S to the old weight vector W, the calculated

outputs would be too large. The new vector S would be the opposite of the previously calculated vector S; multiplying it by the scale factor and adding it to the weight vector would result in the previous set of weights. Thus, the algorithm would simply oscillate between two sets of weights and never find a solution. To counteract this problem, the vector S is no longer multiplied by only the scale factor. It is also multiplied by a random number between 0 and 1. This prevents the oscillation problem.

3 Unsupervised Learning and Genetic Algorithms

Some networks are trained by by learning from example, as in the Perceptron Learning Algorithm which was described earlier. Sometimes, an unsupervised learning algorithm, in which a neural network learns on its own, is better. Using supervised learning in this project would have required the creation of an algorithm defining the best way to direct traffic. Unsupervised learning allows the program to discover for itself the most efficient method to direct traffic and also allows it to adjust to changing conditions.

3.1 Darwinian Evolution

Darwinian evolution is one of the cornerstones of modern biology. According to Darwin's Theory of Evolution by Natural Selection, life is a struggle and those individuals most suited to their environment are the ones most likely to survive long enough to reproduce. This concept is summed up quite nicely as *survival of the fittest*. Individuals who are less fit tend to die without reproducing. The result of natural selection is that the forms of genes, or

alleles, used to create successful individuals endure, while the alleles responsible for creating less successful individuals are eliminated from the gene pool.

3.2 Genetic Algorithms Details

Originally, neural networks and genetic algorithms were viewed as competing forms of artificial intelligence, until it was realized that they could be used together. Genetic algorithms attempt to mimic natural selection and Darwinian evolution. A group of individual objects is given a task; the ones that perform the task the most efficiently are considered the fittest. Just as in biological evolution, where the fittest individuals are the ones most likely to reproduce, genetic algorithms select the most fit objects and use them as templates to create a new generation of objects.

3.3 Genetic Algorithms and Neural Networks

When genetic algorithms and neural networks are used together, the weights of the neural networks are treated as genetic code. Every individual object has its own set of weights that determine how it does its job. After each generation, the best performing individuals are selected and, in a sense, mated. A new generation of individuals, utilizing the weights of the past top performers is created. Often, slight mutations are added to the weights to help constantly tweak the brains of the individuals in the hopes of finding better sets of weights.

4 Simulation and Classes

Although the simulation aspect of this project is secondary to the AI, it still is quite important. The simulation consists of a matrix of intersections with cars waiting to move between intersections. Cars move when the light is green in their direction; the rest of the time, they wait. The traffic lights monitor the situation around themselves, making decisions about whether or not to change.

4.1 Java Classes

Because of the relatively large numbers of intersections that had to be dealt with, object-oriented programming (OOP) was an absolute necessity. OOP makes it easy to deal with large numbers of independent objects of the same type. Furthermore, this project uses a graphical display; the built-in graphical user interface (GUI) features of the Java programming language made it a natural choice for coding the program. The classes used by the simulation are described below.

4.1.1 Class World

This class, defined in World.java, is the driver for the simulation. It contains a timer loop that calls an update method, located in the Intersection class. Class World is also responsible for creating the layout used by the simulations graphical display. It also contains some static public variables used by the rest of the simulation, including numbers for the dimensions of the simulated world and the number of times the iteration will run.

4.1.2 Class Intersection

This project is largely a simulation of intersections, so this class is extremely important. The class definition contains variables indicating the number of cars waiting in each direction, as well as references to neighboring intersections. There is also a traffic light attached to each intersection, whose status during each iteration is determined by the method *chooseChange()*. This is the method, along with *changeWeights()*, also in Class Intersection, and *chooseBestFitness()*, in Class World, responsible for handling the learning process in this project.

4.1.3 Class TrafficLight

To make things easier, traffic lights are represented in their own class. Each light has data indicating which direction it is currently allowing traffic to move in as well as how long it has been since the light last changed. When compared to the other classes, Class TrafficLight is the simplest of them all.

4.1.4 Class IntersectionPanel

This class makes use of Java's built-in graphics features. It provides a simple way to view the status of a traffic light visually. Each instance of this class is an extension of the JPanel class, which nine squares organized in a GridLayout. The corners are place holders, the middles of each side correspond to north, south, east, and west queues of cars, and the center shows the status of the traffic light with an icon.

4.2 Simulation Process

Overall, the simulation is rather simple. The world consists of a matrix of intersections, each of which is connected to its neighbors. During each iteration, the simulation cycles through each intersection, checking to see if the light should be changed. Cars are moved accordingly; if a light is green for more than one iteration, the number of cars allowed to more through the light increases. This is an effort to simulate a real-world fact about intersections—namely that cars do not start moving instantly after a light changes, but rather slowly increase their speed. Border intersections are handled by having cars randomly come in from the outside world. More details about the overall program design are next.

5 Program Design

This project follows a design somewhat similar to a tutorial found at http://www.ai-junkie.com/nnt1.html. Neural networks and genetic algorithms are used in conjunction with each other to create a population of traffic lights that gets smarter over time.

5.1 Activation Function

The traffic lights, after each iteration, really only have two choices to pick from regarding what to do next—they can switch directions or they can stay the same. Yellow lights have been ignored for now in this project. With that in mind, the activation function uses a step activation function, as described previously.

5.2 Perceptron Learning

The perceptron algorithm is not particularly useful for this project. Using it would have required the creation of test data demonstrating the best way to direct traffic under varying conditions. This would have been very time consuming to do. The goal of this project was to create a system of traffic lights that would learn by themselves how to direct traffic. For this reason, genetic algorithms (described next) were used in the simulation. Perceptrons were used for preliminary work in this project to create a program that could be trained to add two numbers together. The program is given two inputs and the expected output. It then calculates what it thinks the output should be and compares its calculated output to the expected output. The weights are continuously modified until the calculated output consistently falls within +/- 1.5 of the expected output. Once this is completed, the program is tested-it is given sets of inputs and it displays the calculated outputs, allowing the user to see if the training was successful.

5.3 Genetic Algorithms

If the perceptron algorithm isn't useful for this program, then another form of learning is needed. In this project, genetic algorithms are used to train the traffic lights. Fitness of each traffic light in the simulation is calculated after every five cycles by taking into consideration the number of cars that the light has allowed through in the past iteration, the number of cars still waiting to move through the light, the number of cycles it has been since the light last changed, and the number of times the light has changed in the last five cycles. The first

value, the number of cars that have moved through the light, increases the light's fitness. Cars waiting to pass through the light reduce the light's fitness. The number of cycles that have passed since the light changed is relatively neutral. Finally the number of times the light has changed reduces the light's fitness if there are several cars waiting in each direction. The reason for this is that sometimes the traffic lights will constantly switch direction; if this is done to excess, it becomes incredibly inefficient, so the lights are heavily penalized for doing it. The exact equation used is shown below, where F is fitness, N, S, E, and W are the cars waiting in line in the north, south, east, and west directions respectively, M is the number of cars that have moved through the light, C is the number of excessive light changes, and I is the number of cycles that have passed since the light last changed. Excessive light changes means that the light changed when there were more than six cars waiting in the directions the light was green in.

$$F = \frac{M}{5} + \frac{N + S + E + W}{4} - C - I$$

6 Resources

Currently, I have three major sources that I have relied on. The first is a website:

http://www.ai-junkie.com/nnt1.html

And two books:

Artificial Intelligence by Elaine Rich and Kevin Knight

Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig.