30

Using Neural Networks to Control Agent Threat Response

Michael Robbins

30.1 Introduction

Neural networks are one of the oldest and most widely used machine learning techniques, with a lineage dating back to at least the 1950s. Although there has been some concern within the game AI community that they might not be the right fit for games, our experience with using them in *Supreme Commander 2* has been tremendously positive. Used properly, they can deliver compelling behaviors with significantly less effort than it would take to hand-code them. In *Supreme Commander 2*, neural networks were used to control the fight or flight response of AI controlled platoons to great effect. Far from being useless, neural networks added a lot of value to the AI without an exorbitant amount of effort.

There are numerous resources both in print and on the web that describe the basics of neural networks, and even provide sample code. The books *Artificial Intelligence for Games* [Millington 09] and *AI Techniques for Game Programing* [Buckland 02] are great resources for getting started, while *Game Programming Gems 2* [Manslow 01] provides sample code and a wide range of practical hints and tips. This article will focus on the specifics of how neural networks were used in *Supreme Commander 2*.

30.2 What Is a Neural Network

There are many different types of neural networks but this article will focus on multilayer perceptrons (MLPs), which were chosen for *Supreme Commander 2* because they're relatively easy to implement and simple to use.

MLPs typically consist of three layers of neurons or "nodes," as they are often called in the neural network literature. These layers are the input layer, the hidden layer, and the output layer. Each node has a value associated with it that lies in the range zero to one and indicates its level of excitation. Nodes are connected to other nodes by unidirectional "weights," which are the analog of biological synapses and allow the level of excitation of one node to affect the excitation of another. In an MLP, each node receives stimulation only from nodes in the preceding layer and provides stimulation only to nodes in the next layer.

Data is fed into an MLP by setting the levels of excitation of the nodes in the input layer. Each node in the hidden layer then receives an amount of stimulation that is equal to an internal bias plus the sum of the products of the levels of excitation of each node in the input layer and the weight by which it is connected to it. The excitation of each node in the hidden layer is then calculated by applying a nonlinear activation function to the value that represents its level of stimulation. The logistic function is the standard choice of activation function for MLPs and produces a level of excitation in the range zero to one.

This process is repeated with each layer in the network receiving stimulation from the preceding layer until the levels of excitation of the network's output nodes have been updated; these levels constitute the network's output and hence its response to the earlier input. The behavior of an MLP is determined entirely by the values of its weights and biases, and the process of training it consists of finding the values of the weights and biases that minimizes some measure of the difference between the network's outputs and some ideal target values.

30.3 Setting Up a Neural Network

For *Supreme Commander 2*, it was decided to use an MLP to control a platoon's reaction to encountering enemy units. We decided to use a total of four MLPs, one for each platoon type: land, naval, bomber, and fighter. We split the MLPs this way so that each platoon type could learn what it needed to without interfering with the other platoon types.

The bulk of the AI's platoon logic would exist inside of a finite-state machine that would use the MLP to decide what to do when the platoon encountered enemy resistance and would continue to use the MLP to reevaluate the constantly changing situation. MLPs provide a great way to accomplish this because they can quickly size up a situation based on their training. In any situation, an MLP can give an AI the ability to determine which enemy targets it should attack first or to retreat if it found itself outmatched. To accomplish this, the first thing that needs to be done is to decide what information the MLP needs to make these decisions and how it should be represented.

30.3.1 Choosing Inputs

Inputs are supplied to an MLP by setting the values that represent the levels of excitation of its input nodes. These values are typically bounded to lie in the range zero to one, though the range minus one to plus one also works well with MLPs. For *Supreme Commander 2*,

inputs were created by taking the ratio between the friendly and enemy values of certain statistics which included number of units, unit health, overall damage per second (DPS), movement speed, resource value, shield health, short-range DPS, medium-range DPS, long-range DPS, and repair rate. All input values were clamped to lie in the range zero to one, so the reciprocals of the ratios were also included to provide the network with useful information about the relative sizes of the statistics even when the friendly statistic exceeded the enemy statistic. These statistics were gathered from friendly and enemy units in a radius around the AI's platoon. Altogether, 17 ratios were calculated, and hence the network had 34 inputs.

This relatively large number of inputs worked well in *Supreme Commander 2* but could be problematic in other applications, particularly if there were only a few thousand examples that could be used in training. This can lead to what is called "overfitting," which is where a network effectively learns certain specifics of the training data rather than the general patterns that lie within it. Overfitting is apparent when a network performs significantly better during training than it does when tested. Overfitting is most easily prevented by retraining with a simpler network (or by providing a larger set of training data, of course). Thus, in general, it's a good idea when choosing inputs to find as small a set as possible. At the same time, the MLP will only be able to account for information that you provide to it, so the desire to have a small input set needs to be balanced against a desire to include as much of the relevant information as possible. At the end of the day, you'll need to experiment to find what works for your project.

30.3.2 Choosing Outputs

When inputs are applied to an MLP, it computes outputs in the form of values between zero and one that represent the levels of excitation of its output nodes. For *Supreme Commander 2*, it was decided that each output node would represent the expected utility of one of the actions that the platoon could take. These actions included attack the weakest enemy, attack the closest enemy, attack the highest value enemy, attack a resource generator, attack a shield generator, attack a defensive structure, attack a mobile unit, attack an engineering unit, and attack from range. Although the platoon could run away, the act of running away was not associated with any individual output. Instead, it was decided that the platoon would run away if none of the network's outputs were above 0.5, because that indicated that no individual action was expected to have particularly high utility.

30.3.3 Choosing the Number of Hidden Nodes

It is the hidden nodes in an MLP that are responsible for its ability to learn complex nonlinear relationships, and the more hidden nodes a network has, the more complex are the relationships that it can learn. Unfortunately, increasing the number of hidden nodes also comes at the cost of increased training time and, as with increasing numbers of inputs, an increased risk of overfitting. Unfortunately, the optimum number of hidden nodes is problem dependent and must be determined by trial and error. One approach is to initially test your network with only two or three hidden nodes, and then add more until acceptable performance is achieved. For more complex decisions, it's reasonable to start with a larger network, but you will want to ensure that the trained network is thoroughly tested to make sure that its performance under test is consistent with its performance during training.

For *Supreme Commander 2*, we found that a network with 98 hidden nodes achieved good and consistent performance during both training and testing. Such a network would be too large for many other applications, particularly when the amount of training data is limited, but given our ability to generate arbitrarily large amounts of training data and the complexity of the decision being made, this worked well for us.

30.4 Training a Neural Network

Training an MLP usually involves repeatedly iterating through a set of training examples that each consist of a pairing of inputs and target outputs. For each pair, the input is presented to the network, the network computes its output, and then the network's weights and biases are modified to make its output slightly closer to the target output. This process is repeated for each example in the training set, with each example typically being presented hundreds or thousands of times during the course of training.

In *Supreme Commander 2*, we decided not to create a fixed set of training examples but to generate examples dynamically by making the AI play against itself. This was achieved by putting two AI platoons on a map and having them battle against each other as they would in a regular game, except we would run the game as fast as possible to speed up iteration time. During the battle, the AI's platoons would act the same as they would in a regular game. The AI's neural networks would make a decision as to which action should be performed whenever opposing platoons met on the battlefield by gathering data about the friendly and enemy units in a radius around the platoon and feeding that data into the MLP. Instead of actually taking the action suggested by the network, however, each platoon was made to perform a random action and a measure of how good those actions were—a measure of their utility—was derived using a fitness function. The utility measure then formed the target output for the output node corresponding to the random action, and the target outputs for all other output nodes were set to each node's current level of excitation; in this way, the network updated its weights and biases to improve its estimate of the utility of the random action but didn't attempt to change any other outputs. Random actions were used instead of the actions suggested by the networks to ensure that a good mix of actions were tried in a wide range of circumstances. An untrained network will typically repeatedly perform the same action in a wide range of circumstances and hence will learn extremely slowly—if it learns at all.

This training process produced an MLP that responded to an input by estimating the utility of each of the different actions. Choosing the best action was then a simple matter of choosing the action associated with the output that had the highest level of excitation. The key to ensuring that these actions were appropriate was to make sure that the fitness function—which assessed the utility of actions during training—assigned the highest utility to the action that was most appropriate in each situation.

30.4.1 Creating the Fitness Function

The fitness function's job is to evaluate the results of the selected action to determine how much better or worse the situation became as a result of its execution. For *Supreme Commander 2*, this was achieved by gathering the same set of data (number of units, DPS values, health, etc.) that were used to make the initial decision, and then examining how those data values changed when the action was taken.

Listing 30.1. Fitness function snippet from *Supreme Commander 2*.

```
float friendRatio = 0.0f;
int numData = 0;
for (int i = 0; i < mFriendData.size(); ++i)
{
    if (mFriendData[i] > 0.0f)
    {
        ++numData;
        friendRatio += (newFriendData[i]/mFriendData[i]);
    }
}
if (numData > 0)
    friendRatio /= numData;
float enemyRatio = 0.0f;
numData = 0;
for (int i = 0; i < mEnemyData.size(); ++i)
{
    if (mEnemyData[i] > 0.0f)
    {
        ++numData;
        enemyRatio += (newEnemyData[i]/mEnemyData[i]);
    }
}
if (numData > 0)
    enemyRatio /= numData;
DetermineNewOutputs(friendRatio, enemyRatio, mOutputs, mActionIndex);
network->FeedAndBackPropagate(mInputs, mOutputs);
```

Listing 30.1 gives a snippet of the fitness function we used on *Supreme Commander 2*. It first takes the ratio between the new and old values for each type of data. Note that since all of these values are likely to have stayed the same or gone down, all of these ratios should be between 0 and 1, which constrains the magnitude of the later calculations to something reasonable. Next, we take the average of the ratios for the friendly units and for the enemy units. This gives a sense of how much the overall tactical situation has changed for each side not only in terms of damage taken, but also in terms of every significant capability—shields, damage output, number of units remaining, and so forth. The resulting averages are passed into `DetermineNewOutputs` which determines what the correct output—called the *desired output*—value should have been using Equation 30.1.

$$desiredOutput = output \times \left(1 + \left(friendRatio - enemyRatio\right)\right) \tag{30.1}$$

This desired output value is then plugged into the corresponding output node of the MLP, and the MLP goes through a process of adjusting weights and biases, starting at the output layer and working its way back to the input layer in a process called *back propagation*. This is how an MLP learns.

30.4.2 Adjusting Learning Parameters

The training of an MLP is typically controlled by a learning rate parameter that controls the sizes of the changes the network makes when adjusting its weights and biases. A higher

learning rate allows for larger changes, which can lead to faster learning but increases the risk of numerical instability and oscillations as the network attempts to zero in on optimum values; a lower rate can make training impractically slow. One common trick is therefore to start training with a higher learning rate and decrease it over time—so you initially get fast learning but, as the weights and biases approach their optimum values, the adjustments become more and more conservative. For *Supreme Commander 2*, we initially started with a learning rate of 0.8 and gradually lowered it down to 0.2.

MLP training algorithms usually also have a parameter called *momentum*, which can be used to accelerate the learning process. Momentum does this by reapplying a proportion of the last change in the value of a weight or bias during a subsequent adjustment, thereby accelerating consistent changes and helping to prevent rapid oscillations. As with the learning rate, a higher value for the momentum parameter is good initially because it accelerates the early stages of learning. For *Supreme Commander 2* we started with a momentum value of 0.9 and eventually turned momentum off entirely by setting it to zero.

30.4.3 Debugging Neural Networks

A neural network is essentially a black box, and that makes debugging them difficult. You can't just go in, set a breakpoint, and figure out why it made the decision it did. You also can't just go in and start adjusting weights. This is a large part of the reason why neural networks are not more popular. In general, if an MLP is not performing as desired, then it's usually a problem with the data its receiving as input, the way its outputs are interpreted, the fitness function that was used during training, or the environment it was exposed to during training.

For example, if an MLP performs well during training but performs less well during testing, it could be because the environment the network was exposed to during training wasn't representative of the environment it experienced during testing. Maybe the mix of units was different, or something changed in the design? It could also be due to overfitting, in which case a network with fewer inputs or fewer hidden nodes might perform better. If an MLP performed well during training but its behavior isn't always sensible, then it might be that the fitness function that was used during training was flawed—perhaps it sometimes assigned high utility to actions that were inappropriate or low utility to actions that were appropriate—more on this point later. If an MLP fails to perform well even during training, then it's usually because either its inputs provide too little relevant information or it has too few hidden nodes to learn the desired relationships.

If you are using neural networks in a game, these points need to be stressed. When debugging neural networks, the solution is usually not to find the point of failure by setting a breakpoint. You have to think about the network's inputs, its outputs, and how your fitness function is training your neural network.

30.4.4 Case Study: Repairing a Bug in the Fitness Function

In *Supreme Commander 2,* each player starts with a unit called an ACU and whichever player destroys their opponent's ACU first wins the game. However, when an ACU is destroyed, it blows up in a large nuclear explosion, taking out most of the smaller units and buildings in a wide area. For the neural network this posed a problem: since the network was trained on tactical engagements, it didn't know about winning or losing. All it saw was that when it sent units up against an ACU, most of them were destroyed.

This introduced a bug that made the AI unwilling to commit troops to attack an ACU. It would overwhelm players with massive groups of units but, as soon as it saw an ACU, it would turn tail and run. The problem wasn't in the behavior code, and it wasn't something that could be tracked down by setting a breakpoint; the problem was in the fitness function.

Once we realized what the problem was, the solution was simple: we needed to modify the fitness function to take into account the destruction of an enemy ACU. Basically, we needed to teach the neural network that it was worth taking out an ACU whatever the cost. This was done by modifying the fitness function to provide a very positive measure of utility whenever an enemy ACU was destroyed. Instead of relying on the results of Equation 30.1, the fitness function would return a desired output of double whatever the original MLP output was, clamped to a maximum of 1.0. After retraining the network with the new fitness function, we saw a huge improvement. The AI would run from the ACU if it only had a small number of units but, if it had a large enough group to take it down, it would engage, winning the game as the enemy's ACU blew up in spectacular fashion.

30.5 Adjusting Behavior

Even though the behavior of an MLP is fixed once it's been trained, it's still possible to use it to generate AI that exhibits a variety of different behaviors. In *Supreme Commander 2*, for example, we added an aggression value to the AI personality. This value was used to modify the ratios that were input to the MLP to mimic the effect of the AI's units being stronger than they actually were. This made the MLP overestimate the utility of more aggressive actions, producing an overall more aggressive AI.

Rather than always having the AI perform the action for which the MLP estimated highest utility, different action selection schemes could be considered. For example, the AI could select one of the N highest utility actions at random or select an action with probability proportional to its utility. Both of these schemes would produce behavior with greater variety though they both involve selecting actions that are probably suboptimal and hence would probably produce AI that is easier to beat.

30.6 Neural Network Performance

The run-time performance of an MLP is determined by how many nodes it has. In *Supreme Commander 2*, each MLP has 34 input nodes, 15 output nodes, and 98 hidden nodes and we never saw a network take longer than 0.03 ms to compute its output (during an eight-player AI match). Since feeding a MLP forward is basically just a bunch of floating-point math, this is not surprising. Performance will, of course, vary depending on hardware and the details of your implementation, but it is unlikely that the time taken to query an MLP will be a problem.

30.7 Benefits of Using a Neural Network

Probably the most notable benefit of using a neural network over something like a utility based approach is that you don't have to come up with the weights yourself. You don't have

to figure out whether health is more important than shields in any particular decision or how they compare to speed. This is all worked out for you during training. Each of *Supreme Commander 2*'s neural networks took about an hour of training to reach a shippable level of performance. We did, however, have to complete the training process several times before we ended up with a set of neural networks that worked well, mostly due to snags such as the ACU problem that was mentioned earlier.

A major benefit of the input representation that was used in *Supreme Commander 2* was that it provided an abstract representation of the composition of a platoon that remained valid even when the statistics of individual units changed; the neural network is not looking at specific units, only their statistics. As long as there weren't any radical changes in the game's mechanics, the networks were able to continue to make good decisions as the statistics of individual units were modified to produce a well-balanced game.

30.8 Drawbacks of Using Neural Networks

Like most things in life, using a neural network solution doesn't come free. There are certainly some drawbacks to using them over more traditional methods, the foremost of those being their black box nature. With most solutions you can come up with a tool that designers can use to adjust the behavior of the AI; at the very least you can make small adjustments to alter its behavior to suit their needs. With neural networks this is difficult, if not altogether impossible. On *Supreme Commander 2*, we got lucky because we had a separate AI system for the campaign mode than we did for skirmish mode. The designers could make any changes they wanted for the campaign but they did not want to have control over skirmish mode. Unfortunately, most projects are not that lucky.

The other issue is the training time. Unlike with other techniques, where you can easily make small changes, if you change anything to do with a neural network—its inputs, the interpretation of its outputs, the fitness function, and the number of hidden nodes—you have to start training from scratch. Even though training is hands-off, the time it takes makes it difficult to quickly try things out.

30.9 Conclusion

Whenever the subject of neural networks in *Supreme Commander 2* comes up, two questions are frequently asked: Was it worth using them, and would you use them again? The answer to both is yes. We firmly believe that the AI in *Supreme Commander 2* would not have had the same impact without the use of neural networks. Moreover, if someone proposed doing *Supreme Commander 3*, you can bet neural networks would play a part.

That being said, neural networks are not for every project, and they are certainly not the be-all and end-all of AI. Neural networks are a tool like any other in that they have specific strengths and weaknesses. They are very handy if you have a well-defined set of actions or responses and designers don't require a lot of control. If your designers are going to want to fine-tune things or you have to work with multiple sets of responses to accommodate things like different AI personalities, however, you may want to look at other options.

References

[Buckland 02] M. Buckland. *AI Techniques for Game Programming*. Cincinnati, OH: Premier Press, 2002, pp. 233–274.

[Manslow 01] J. Manslow. *Game Programming Gems 2: Using a Neural Network in a Game: A Concrete Example*. Hingham, MA: Charles River Media, 2001, pp. 351–357.

[Millington 09] I. Millington and J. Funge. *Artificial Intelligence for Games*. Burlington, MA: Morgan Kaufmann, 2009, pp. 646–665.