LEARNING PONG

Matthew McBrien
School of Electrical and Computer Engineering
Georgia Institute of Technology
mmcbrien3@gatech.edu

Alexandra Melehan
School of Mechanical Engineering
Georgia Institute of Technology
amelehan3@gatech.edu

Matthew Munns
School of Industrial and System Engineering
Georgia Institute of Technology
mattmunns@gatech.edu

April 21, 2020

ABSTRACT

Reinforcement learning is a machine learning methodology that enables an agent to interact with and learn from its surroundings, dynamically adapting to successes and failures it encounters. In this project, we train a reinforcement learning agent to play a modified version of the game PONG. We use the genetic algorithms NeuroEvolution and NeuroEvolution of Augmenting Topologies to guide the agent's improvement through experience playing the game. In addition, we leverage serverless computing technology to efficiently train the agent in parallel. We find that NEAT is a far superior learning algorithm to NeuroEvolution in terms of efficiency.

1 Introduction

Our project is to use Reinforcement Learning to train a model to play a competitive, one-one-one video game. The game we have chosen to implement is a custom game that is very similar to Pong. To train our model, we implement two different algorithms: 1) NeuroEvolution (NE) and 2) NeuroEvolution with Augmenting Topolgies (NEAT). NE uses a predefined neural net structure and iteratively tunes the weights of the connections using a genetic algorithm. NEAT does the same, but it starts with a minimal structure and builds upon this by adding nodes and connections randomly. All of our code is in a github repository.

2 Methodology

We seek to develop a neural network that will control a paddle to play our modified PONG game effectively. The neural network will make a decision for the paddle for every frame of the PONG game. The network will take in eight inputs: the paddle's X and Y position, the opponent's X and Y position, the ball's X and Y velocity, and the ball's X and Y position. The output of the neural network has four output nodes, corresponding to movement up, down, left and right. Whichever output node has the largest value is the movement that is executed at that frame. So at every single frame the neural net will select a key to press that corresponds to movement in a certain direction.

2.1 Reinforcement Learning

We use two reinforcement learning algorithms to train the neural networks. These algorithms are NeuroEvolution (NE) and NeuroEvolution of Augmenting Topologies (NEAT). The training process consists of neural network controlled players competing against each other within groups called Generations. Generations have a predefined population size, where each member of the population is controlled by a unique neural network. In context of the population, each unique neural net is frequently referred to as a genome. After each member of the population competes with each other member, each genome is scored according to an objective function, and a new generation is developed according to the reinforcement algorithm controlling training.

The success of a particular neural network affects its likelihood to appear in subsequent generations. Success is defined by the experimenter and in our case is separated

into three phases. Each phase has an objective function that awards points or applies penalties for events that may occur during a game. The three phases' objective functions are defined below:

With:

h = the number of times the paddle hit the ball
p = the number of times the ball is passed to the opposite side
g_a = the number of goals against
g_f = the number of goals for
f_s = the number of frames in which the paddle is static

The objective functions for each phase are:

1. $f_{obj} = 0.5h - 0.2f_s$
2. $f_{obj} = 0.5h + 2p - 0.2f_s$
3. $f_{obj} = 0.5h + 2p + 5g_f - 10g_a - 0.2f_s$

The strategies needed to succeed at our version of PONG are complex, and a three stage learning strategy, with more complex objective functions in subsequent stages, allows our reinforcement learning agent to develop fundamental skills like hitting the ball before tackling more involved skills such as scoring.

2.1.1 NeuroEvolution

NeuroEvolution is a set of techniques within reinforcement learning that uses an evolutionary algorithm. The evolutionary algorithm generates, mutates, and scores generations of neural networks. Using this technique, the NeuroEvolution algorithm can begin without any data and be able to learn how to operate successfully in its environment. The algorithm creates a generation consisting of different neural networks. Once each neural network has experienced the environment, they are assigned a score using the objective function. The best neural networks have the highest chance of persisting onto the next generation (similar to natural selection), and mutations can randomly occur that affect the next generation as well.

The NeuroEvolution algorithm alters the generations by only changing the weights of the connections in the network [1]. Weights may mutate randomly up or down. Our NeuroEvolution algorithm was manually implemented in Python.

2.1.2 NeuroEvolution of Augmenting Topologies

The second algorithm, NeuroEvolution of Augmenting Topologies (NEAT), expands upon the original NeuroEvolution algorithm by implementing the ability to change the topology of the neural nets. Layers, connections, and nodes may be added to the network through random mutation. Note that these topological features may only be added, not taken away. This generally tends to lead to faster training as the changing topology results in more sweeping changes to the functioning of the neural network [2]. The NEAT algorithm is implemented in Python using the NEAT package [3].

2.2 Game Development

We have developed a PONG-like game that builds on PONG by adding movement capabilities and a more specific method of scoring. We created the game using the python package Pygame. The main differences are that the goals are smaller and the players have the ability to move in both the x and y dimensions.

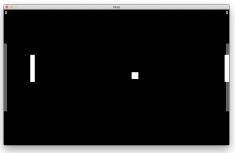

Figure 1: A screenshot of the developed game

The goal of the game is to knock the ball into the opponent's net by controlling your paddle. The faster the paddle is moving when it collides with the ball, the more speed is imparted to the ball. This allows for a more rich set of moves for a player (or computer) to master when compared to the traditional Pong game.

2.3 Parallelization

There is an obvious problem from having the neural nets compete in a round-robin style. If each neural net must compete against every other neural net one time and the number of neural nets in a generation is N_{nn}, then the number of games that must be played in a generation, G_{gen}:

$$G_{gen} = \binom{N_{nn}}{2}$$

If there are a meager 30 neural nets per generation, this means that $G_{gen} = 435$. If each game takes only 0.1 seconds, which we have found to be possible, each generation will take almost 1.5 minutes. This number will explode with more neural nets. We have looked at a number of solutions, such as limiting the amount of games, but the most exciting has been to use the AWS cloud for parallelization.

To parallelize the round-robin like tournament in the cloud, we are using AWS Lambda which is a serverless computing platform and AWS Kinesis which is a data stream platform. We have written a lambda function that is capable of executing a Pong game with two neural nets. When the game is completed, the result is submitted to the kinesis data stream. The lambda function can be executed in a highly parallel fashion. This means we can drastically cut down on the time per generation. Preliminary results

have shown that we can complete a generation of 45 neural nets ($G_{gen} = 990$) in about 8 seconds, compared to about 110 seconds when run sequentially and locally. One of the bottlenecks that had to be avoided was the time to actually submit this many requests to Lambda. Submitting 990 post requests from a single thread can take tens of seconds, which is unacceptable. To ameliorate this, we implemented fanout within the Lambda functions. This means that, for example, 20 different games are submitted to each lambda function. The Lambda function then submits 16 of those games to new Lambda functions and locally executes the remaining four. When the second level Lambda function receives the 16 requests, it submits 12 of them to another Lambda function and locally executes the remaining four. This continues until all of the games have been executed. This drastically cuts down on the time per generation.

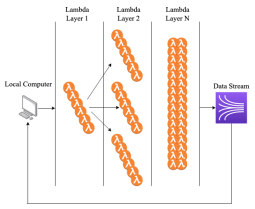

Figure 2: A depiction of how the computer responsible for training submits the games to Lambda functions using a fanout technique and then retrieves results from a Kinesis Data Stream.

3 Results

Most of the results discussed here will be qualitative. Discussion of what steps were performed and how well the algorithm learned the game of pong is here.

3.1 NeuroEvolution

NeuroEvolution proved to be an ineffective means of learning complex tasks. Since the researcher must hand-select the topology of the neural net there is a tradeoff that is introduced. Generally, small neural nets will learn something simple quickly while large neural nets are capable of learning something more complex in a longer amount of time.
In our case we tested both a simple neural net and a complex neural net. When we passed the algorithm a neural net with no hidden layers the algorithm was capable of learning something basic but was unable to learn more complicated tasks. After 300 generations under the phase 1 objective function, the NeuroEvolution algorithm developed some semblance of skill. The paddle moved up and down, occasionally following the ball. However, after completing more generations and increasing the complexity of the objective function, the NeuroEvolution did not progress in any way.
When we gave the NeuroEvolution a more complicated starting neural net, with 10 hidden layers that were fully connected, it was unable to learn any strategy whatsoever after 600 generations.

3.2 NeuroEvolution of Augmenting Topologies

NEAT proved to be a much more effective learner than NeuroEvolution. Starting with the most basic neural net, which only contained the 8 input nodes and the 4 output nodes, it had learned an effective defensive game of pong after only 825 generations of learning, each generation having a size of 45 genomes. This strategy consisted of moving the paddle to the back wall and then following the ball's movement up and down to prevent the ball from entering the goal. Since the main most significant value of the objective function was goals scored against the neural net, it is logical that the algorithm developed a defensive strategy.
Additionally, in direct comparison to NeuroEvolution, after 300 generations of using the phase 1 objective function, both of the algorithms had developed similar strategies – moving up and down to hit the ball. However, NEAT was clearly farther along in terms of development as it was significantly better at tracking the ball.

4 Conclusions

While both NeuroEvolution and NEAT are capable of learning some task, it is clear that NEAT is a much more efficient learner. This follows from the fact that is a more generic learner and has one less parameter than NeuroEvolution, namely the topology of the neural net. The topology is a complicated yet critical component of the algorithm's ability to optimize the objective function. Removing it from the researcher's hands generally speeds up learning time.
It is noteworthy that while the final product produced by NEAT is a competent player, it fails on two fronts. First, it is nearly incapable of mounting an offensive threat in the game. This is due to the final objective function and how that objective function was built over the phases. The final objective function emphasised defense by having a large negative constant associated with the goals against variable. This meant it was almost always more beneficial to emphasise defense in the final phase of the objective function. Additionally, the two phases of the objective function prior to the final one did not help to learn the necessary skills to attack in the game. From the previous phases of the objective function, the algorithm only learned to track the ball's position using it's y velocity. This skill does not translate to anything in terms of offense, but it translates well to defending the goal. Ultimately, this is the only real strategy that the algorithm could perform – defend the goal.
An additional failing of the algorithm has to do with the size of the population. The small population size severely limited the learning of the algorithm. There are two ways this is limiting. Firstly, more genomes per generation sim-

ply means that the algorithm is exploring more of the solution space per generation. More random permutations will generally lead to a better result. However, since this is a competitive game, it also means that the environment is being explored less due to the small population size. Exploring the environment of the game is impossible without varied opponents to explore against. Having more genomes per population would lead to discovering potential weaknesses in other genomes. This problem manifest itself in the form of glaring error in the final neural net. While it was capable of tracking the ball when it is moving up and down, which it is most of the time, if the ball comes in a direct line, i.e. with 0 for the y velocity, the neural net does not understand how to track the ball. This means its defense is completely permeable. This failing most likely occurred because it never had this strategy used against it. There were no neural nets in the population that were hitting it straight at the goal, so a defense against this strategy was not valuable to the algorithm. This means that in a competitive game having a large population size and having that population represent a wide range of strategies is critical to creating a robust final neural net.

Finally, it should be noted how important the objective functions are. It is critical that the objective function represent the true intent of the researcher. There should be only one method of optimizing the objective function, and it should be an acceptable strategy within the game. In an earlier version of our work, the algorithm learned a trick to optimize the objective function that was outside of what the researchers expected. The original objective function at this stage was only dependent upon the number of times the paddle hit the ball; the number of frames in which the paddle isn't moving wasn't penalized. In an earlier version of the game, there was a way for the ball to glitch into the paddle. The algorithm quickly learned this skill. It sat still and used the strategy to rack up an endless amount of "hits". This led to us patching this glitch and adding a penalty for a lack of movement.

5 Future Work

There are several ways that this work could be furthered to improve learning performance. Firstly, finding a way to increase population size significantly, to hundreds of genomes per generation, would greatly increase the learning rate of the system. This could be done by using a different cloud computing service to distribute the games across a wide range of servers. Additionally, instead of having every genome compete against every other genome a single time, resulting in $\binom{n}{2}$ games per generation, a random selection of games could be selected. This would decrease the necessary computational effort per generation while resulting in most likely a negligible drop in the scoring of the genomes.

The objective function is also an area for improvement. To learn more advanced skills, such as hitting the ball in order to score a goal, the objective functions must be more finely tuned and probably layered more finely in complexity. How many generations should study a single objective function is also critical but was not deeply explored in this study. An additional set of inputs could have been used (opponent paddle's X and Y velocity), but this would have made the project much more complex. In the future, this could be added in order to predict the ball's velocity through basic momentum theory.

Finally, which inputs are given to the neural net is a parameter of reinforcement learning that our version of NEAT retains. This could be removed entirely, making NEAT a nearly parameter-free learning algorithm, besides the setting of the objective function. The removal of specific inputs would take significant work as it would require the entirety of the game to be an input in the form of an image. Using convolution neural nets to play games has been explored in [4].

References

[1] Edmund Ronald and Marc Schoenauer. "Genetic Lander: An Experiment in Accurate Neuro-Genetic Control". In: *Proc. 3rd Conf. Parallel Problem Solving from Nature*. Springer-Verlag, 1994, pp. 452–461.

[2] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[3] Alan McIntyre et al. *neat-python*. https://github.com/CodeReclaimers/neat-python.

[4] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).