Learning Breakout using NEAT

James T. Murphy III[*] Skyler Thomas[†]

December 7, 2017

Abstract

We present a replication of the learning of a version of Breakout via machine learning. The method used is NeuroEvolution of Augmenting Topologies (NEAT). We show that given the current and past few game states as inputs, NEAT can evolve a neural network that can completely clear a game of Breakout.

1 Introduction

Explain Breakout, explain NEAT algorithm overview.

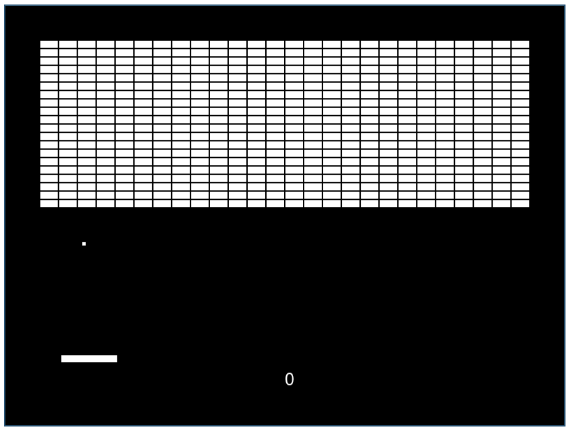

Figure 1: The initial setup of the game Breakout.

[*]jamesmurphy@math.utexas.edu
[†]skylerthomas0@gmail.com

2 Results

We find there are qualitatively three eras of evolution of the network:

1. *Pre-ball-tracking era*: the network typically does nothing or makes a few small adjustments at the very beginning of the game, scoring a minimal number of points usually 0 to 30.

2. *Ball-tracking era*: the network definitively makes an effort to keep the paddle horizontally aligned with the ball. It clears a sizeable portion of the blocks, and only fails in edge cases or by infinitely looping without clearing some remaining blocks. It typically scores 100 to 400 points.

3. *Aiming era*: the network tracks the ball as before, but also makes sporadic adjustments to aim at remaining blocks. It clears all or nearly all of the blocks, achieving a score from 515 to the maximum 520.

A large percentage of the time, around 90% of attempts, the NEAT algorithm stagnates at a local maximum fitness somewhere in the first two eras. Simply repeating the experiment a sufficient number of times eventually yields a trained network that can achieve the maximum score. The resultant network is typically very small, having 5 or less non-input nodes and under 15 total connections.

It turned out to be critical to the success of the NEAT algorithm that several previous game states (3 is sufficient) were included as inputs to the neural networks. Giving only the current state, including the direction of the ball, as input was not sufficient in any of our experiments to produce a network that could even reliably track the ball.

Another pitfall that was avoided was giving too many inputs to the networks. If the game state fed to the net-

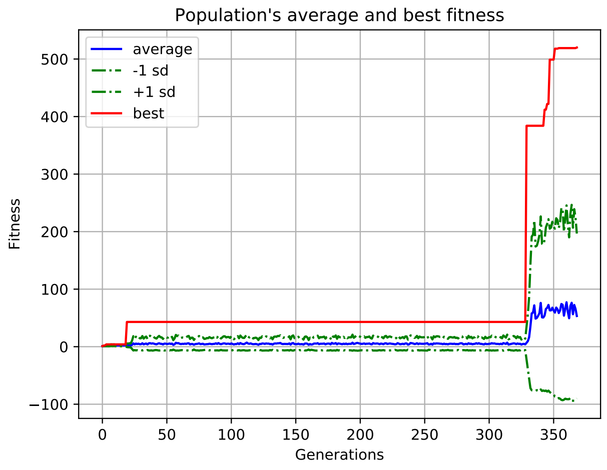

Figure 2: A successful training experiment. In this case, the pre-ball-tracking era lasts until around generation 325, where the network abruptly learns how to track the ball. The ball-tracking era then lasts until around generation 350, where the network learns to avoid missing stray blocks. The aiming era is eached in the last few generations. Generations, each of 200 individuals, took an average time of 2.835 seconds to evaluate running on an Intel Core i7-4790K @ 4.00GHz in parallel on 6 cores. The total runtime was around 17.5 minutes.

work included information about all 520 blocks, then regardless of whether the game history was given, none of our experiments produced a network that could reliably track the ball. The vast majority of networks trained in this manner did precisely nothing, making no button presses at all. On the other hand, training was able to produce an aiming era individual when the density of blocks remaining in a few (under 10) predetermined regions were passed as inputs. Training also succeeded when the game state passed to the networks included no information about the remaining blocks whatsoever, i.e. a network that saw only the paddle and ball could still achieve a maximum score.

3 Methods

We modified a Pygame [6] implementation of Breakout (cf. [4]) to allow human mouse input and human keyboard input (for testing purposes), or automated keyboard input. Then we used NEAT-Python [1] to run the NEAT algorithm.

To allow the neural network to play Breakout, each frame a representation of the game state is computed and passed as the inputs to a neural network. The neural network has two outputs "L" and "R", which are binarized and taken to indicate whether the left or right arrows are being pressed by the neural network.

The representation of the game state includes the following information:

- the x and y coordinates of the ball, normalized as a percentage of the width of the game
- the x-direction ±1 and y-direction ±1 of the ball,
- the angle of the ball, normalized as a percentage of 2π
- the x coordinate of the center of the paddle, normalized as a percentage of the width of the game
- the history of the previous four up to a fixed number of time steps in the past
- (potentially) the density of remaining blocks in e.g. each of the four block quadrants or similar fixed configuration of predetermined regions of blocks.

After carefully choosing the parameters of the algorithm, a neural network it trained to play Breakout using NEAT with the final score of playing a game being the fitness function.

Since the success of the algorithm is so sensitive to the configuration, we reproduce it here:

```
[NEAT]
fitness_criterion     = max
fitness_threshold     = 520
pop_size              = 200
reset_on_extinction   = True

[DefaultGenome]
# node activation options
```

```
activation_default      = sigmoid
activation_mutate_rate  = 0
activation_options      = sigmoid

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = .5
bias_max_value          = 5
bias_min_value          = -5
bias_mutate_power       = .10
bias_mutate_rate        = 0.05
bias_replace_rate       = 0.05

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.9
conn_delete_prob        = 0.05

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.05

feed_forward            = True
initial_connection      = unconnected

# node add/remove rates
node_add_prob           = 0.1
node_delete_prob        = 0.05

# network parameters
num_hidden              = 0
num_inputs              = 18
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = .5
weight_max_value        = 5
weight_min_value        = -5
weight_mutate_power     = 0.125
weight_mutate_rate      = 0.160
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 200
species_elitism      = 1

[DefaultReproduction]
elitism            = 5
survival_threshold = 0.05
```

4 Discussions

Talk about Deep Q learning and other approaches that are better than ours.

5 References

[1] CodeReclaimers. Neat-python. https://neat-python.readthedocs.io, August 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-

level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] F. Primerano. Breakout. `https://github.com/Max00355/Breakout`, December 2015.

[5] SethBling. Mari/o machine learning for video games. `https://www.youtube.com/watch?v=qv6UVOQ0F44`, June 2015.

[6] P. Shinners, R. Dudfield, M. von Appen, and B. Pendleton. Pygame. `https://www.pygame.org/`, January 2017.

[7] K. O. Stanley and R. Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 569–577. Morgan Kaufmann Publishers Inc., 2002.

[8] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.