# Pavlov's Arcade

## Reinforcement Learning for a Breakout AI

Angela Jiang
Northwestern University
2317 Ridge Avenue
Evanston, IL
(616)635-9604
angelajiang2014
@u.northwestern.edu

Motoki Mizoguchi
Northwestern University
1116 Garnett Place
Evanston, IL
(859)230-0662
motokimizoguchi2013
@u.northwestern.edu

Max New
Northwestern University
1930 Ridge Avenue
Evanston, IL
(985)397-1770
maxnew2013
@u.northwestern.edu

## ABSTRACT

In this paper we describe the design and implementation of a bot to play the arcade game Breakout. Instead of manually writing the bot, we use reinforcement learning techniques to learn a strategy by repeated sessions of Breakout where the learner is rewarded for advancing through the game and punished for losing. We show that our bot's game performance increases with repeated plays, though it does not converge to a strategy that never loses.

## Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

## General Terms

Experimentation

## Keywords

Q-Learning, Reinforcement Learning

## 1. INTRODUCTION

Video games are designed to challenge human players. A new player starts with no knowledge of the game mechanics and learns by reward and punishment which strategies are good or bad. A key aspect of game design is creating an experience that helps the player to learn the game quickly through this conditioning. In our project, we directly apply this philosophy of gaming to the construction of an AI player. Just as a human would, the bot plays the arcade game repeadetly, being rewarded for advancing and punished for losing. Starting from a random strategy, the learner uses a Q-Learning reinforcement technique to update its policy.

In section 2 we describe Q-Learning, the Breakout arcade game and previous work on machine-learning based game AIs. In sections 3 and 4 discuss how we adapt the Q-Learning technique to the game of Breakout. In section 5 we evaluate the efficacy of our learning technique, and we describe future work in section 6.

## 2. BACKGROUND

### 2.1 Q-Learning

In reinforcement learning, the AI receives positive and negative rewards based on the outcome of its actions. For each state, the optimal move may not be clear based on immediate rewards. Reinforcement learning algorithms allow the bot to learn based on delayed rewards so that it can choose moves that maximize future rewards. In the case of breakout, at every instant, the bot must choose a move so that in the future, the ball will hit the paddle and avoid losing the game.

Q-Learning is an algorithm for reinforcement learning that does not provide a model for taking actions. Instead, it learns what the optimal selection for each possible state is. The final policy is the set of each possible state and the optimal action to take. The algorithm is practical for a Breakout bot because we know what the possible states are in advance and the outcome of each move is deterministic. For any finite markov decision process, q-learning is able to converge to find an optimal policy. Unlike on-policy reinforcement learning algorithms such as TD-learing or SARSA, Q-learning if an off-policy method. The final policy does not depend on the way the state space is explored. This will avoid the possiblity of getting trapped at local minima where the optimal move is not explored because the reward is underestimated at first.

The algorithm uses dynamic programming to explore the state space by iteratively approximating the state-action value, Q. The estimates of the Q values of all (state,action) pairs are used to determine the next move. In the next iteration, we are able to estimate the outcome of the move from the reward we receive. The previous Q value is updated accordingly. There is also a possible discount factor applied to observed rewards that motivates the bot to choose the shortest number of moves to a future reward.

### 2.2 Breakout

Breakout is a popular arcade game first released in 1976 by Atari, Inc. In the game, the player has control of a single paddle that sits at the bottom of the screen and can be moved left or right. Above the paddle are a number of bricks the player attempts to eliminate. A ball is released into the game and bounces around the screen. The player has to prevent the ball from passing through the bottom of the screen or they lose a life. In addition, the player advances the game by bouncing the ball into the bricks, which disappear when hit. When all of the balls on the screen are eliminated, a new set of bricks appear on screen and the game continues. The game eventually ends when the player loses all of their lives or a time limit runs out.

While simple, the optimal strategy for the game is not immediately clear. We measure the performance of a strategy by the number of bricks hit until the player runs out of lives or a time limit expires. To demonstrate the trickiness of implementing a Breakout bot consider the most obvious strategy which is just to keep the paddle underneath the ball at all times. This is easy to program, but is it an optimal strategy? The player never loses, but there is no guarantee that the game is advanced and the player may go into arbitrarily long periods in which a ball is never hit. A more advanced strategy would take into account where the bricks are and aim the ball to hit them quickly. We take such considerations into account when evaluating our bot's performance.

### 2.3 Related Work

There have been many works on implementing machine learning in video games. Some research for machine learning video game has been done through implementation of a Neural Networks [4, 5]. Neural Nets has been implemented for arcade game such as Ms. Pac-Man [3]. Reinforcement learning has been implemented in more complex video games such as Super Mario [2]. For implementing reinforcement learning into arcade games, work has been done to provide easier access to the states in many Atari arcade games through the arcade learning environment project [1].

## 3. DESIGN

In order to run Q-Learning, we need to consider the state of the Breakout game. In a game of Breakout, there is a court containing bricks, ball, and paddle. The ball, the paddle, and each brick will have their x-y coordinates and size of their object. The ball and the paddle also has their velocity. The x-y coordinates are given by pixel values, which depends on the window size of the browser that one runs the game. This may range from 300 to 800 pixels in each dimension. With just the paddle and the ball, the order of the state space can be $10^9$ to $10^{11}$ in size. In order to keep the state space small and consistent with different screen size, we decided to normalize the x-y state space to the width of a smallest brick unit called chunk. The court was normalized to a 15 chunks by 15 chunks field, and the x and y values of the ball and the paddle were represented using the mapping to this field. This mapping reduced the state space for the two states by a factor of $10^5$.

Furthermore, some states can be ignored or further simplified. Because the paddle does not move up or down in the game, the y coordinate of the paddle can be ignored. Also, the paddle has a width that stays constant, and the paddle width reduces the position that the paddle can take on the space. For the velocity of the ball, the speed component was kept constant, so we considered the direction of the velocity. We simplified the velocity to take on 6 different direction:

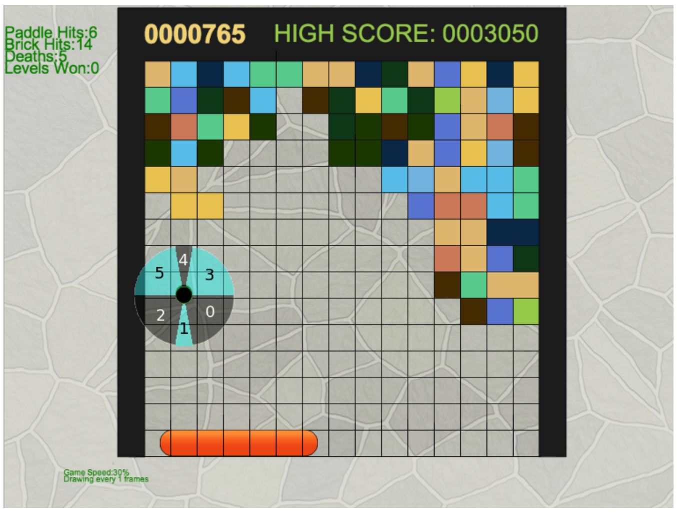

Figure 1: State Space representation: The court is broken into 15 by 15 chunks, and the velocity of ball is represented in 6 states. Without bricks, the state space representation is reduced to $\approx 10^4$ states.

up left, up straight, up right, down left, down straight, down right.

For the bricks, one way of representing is to see whether or not a brick exists in each position in the 15 by 15 field. That representation has $2^{15*15} \approx 10^{67}$ states, which is too big. We first separated the court into 5 vertical regions with 3 chunks in width, and recorded the number of bricks located inside that region. We represent the bricks representing regions with the highest brick number, which takes 5 states.

At any given point during the game, players either move the paddle to the left, to the right, or keep it still. Thus, given a state, there are 3 actions to choose from. A reward is given to the bot for every time it hits a paddle or a brick, and a bigger reward when it finishes a game. Heavy punishment is given to an action that results in a loss.

## 4. IMPLEMENTATION

Our game is adapted from an open source Breakout implementation written in Javascript. Instead of receiving commands from the keyboard, it queries our server for each move. It is therefore synchronous with the server and will only update the game after each response from the server. The Python server provides the Javascript bot with its next move.

The game provides the server with a reward and the current state of the game and receives a command to stay, move left or move right a single chunk. It will also notify the server if the game is won or lost. In each case, the server will update the q-table values for the previous state with the observed reward. It will then determine the bot's next move.

Before our server can provide the optimal moves to our bot, it must learn the strategy. Our server therefore has two modes, learning and testing. In learning mode, our server chooses moves based on an $\epsilon$-greedy algorithm to allow for exploration. In testing mode, it provides the bot with the move that is expected to maximize its reward.

As it is learning, the server will continually update values in the q-table. We therefore need the q-table to be persistent

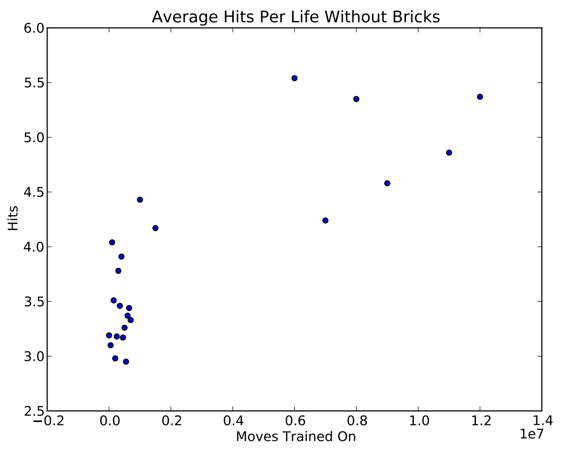

Figure 2: Average Hits per Life

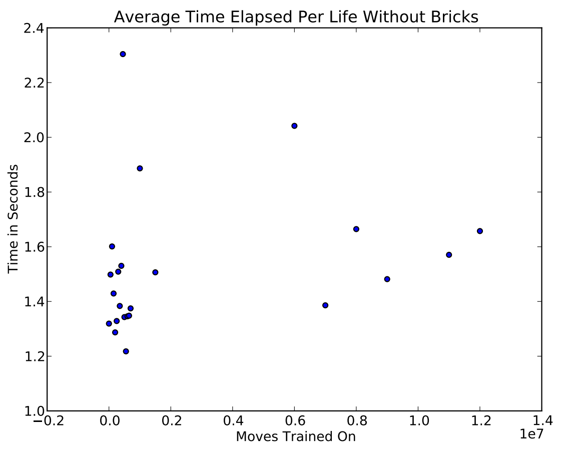

Figure 3: Average Time elapsed per Life

each time we run our server. We serialize the current table each time the server is stopped with ctrl-c. Also to acquire continual snapshots of the state of our server, we serialize the table every 1,000,000 moves in learning mode and 50,000 moves in testing mode.

## 5. RESULTS

### 5.1 Experiment

While our learning is conducted, we take snapshots of the state of the table at intervals of 50,000 and 1,000,000 moves trained on. We then were able to test on these intermediary tables and compare performace. The performance metrics that we used were

- Number of hits of the ball
- Levels beaten
- Time elapsed before losing

Each bot was allowed 100 lives and results are averages. In Figure2, we look at average number of paddle hits as our bot trains. We see that the number of moves the bot trains on correlates to a larger number of paddle hits. This shows that the bot is indeed learning based on our reward for paddle hits. In a level without bricks, we would expect an optimal learner to prolong the game indefinitely. We look that average time elapsed in Figure 3. The short time duration per game is due to the game being sped up for testing purposes. There is less of a noticeable correlation here. This is evidence that our original assumption of what the optimal strategy is may not be correct.

We also see less correlation in bots tested in levels that include bricks. This is expected since we do not include bricks in the state space. Because of this, the outcome of each of our actions is no longer deterministic. For example, while learning, the bot may discover that a paddle position causes the ball to bounce diretly onto the paddle. However, if bricks are introduced into the state, this is no longer the case. Another potentially optimal strategy is to always move towards the ball. The performance using this strategy shouldn't change if bricks are introduced. Our results show that this is probably not the optimal strategy learned either.

## 6. FUTURE WORK

For this project, we were able to perform Q-Learning in a small field and no information about the bricks was used in our state representation. The initial goal was to have an implementation where some sort of information about the bricks on the court was included in the state after we achieve a successful bot that can keep itself alive. Future work can be done to extend our simple reinforcement learner to include the state of the bricks on the court so that it can aim the ball against the bricks for faster clearing of a level.

## 7. CONCLUSION

In our project, we applied reinforcement learning techniques to the construction of a Breakout bot. In our testing, we can observe that our bot's performance increases with training. However, because we see little correlation to other metrics of game strategy, we cannot determine if our proposed strategy is optimal.

## 8. REFERENCES

[1] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, (2012). "The arcade learning environment: An evaluation platform for general agents." *arXiv preprint arXiv:1207.4708.*

[2] J. Togelius, S. Karakovskiy, J. Koutnik, J. Schmidhuber, (2009, September). "Super mario evolution." *In Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* (pp. 156-161). IEEE.

[3] T. G. Tan, J. Teo, P. Anthony, J. H. Ong. "Neural network ensembles for video game AI using evolutionary multi-objective optimization." HIS, page 605-610. IEEE, (2011)

[4] R. Graham, H. McCabe, S. Sheridan, "Neural networks for real-time path-finding in computer games" `http://www.gamesitb.com/nnpathgraham.pdf`

[5] R. Graham, H. McCabe, S. Sheridan, "Neural Pathways for real-time dynamic computer games." `http://gamesitb.com/Graham_EGIreland.pdf`