Reinforcement Learning for Atari 2600 (JStella Learning Environment)

Ian Cichy
Computer Science Dept.
UW-La Crosse
cichy.ian@uwlax.edu

Abstract

JStella, a Java based open-source Atari 2600 emulator, allows users to play some of the most iconic video games of all time. Building upon this system, substantial modifications were made to develop the JStella Learning Environment (JSLE). The JSLE provides the ability to develop and run many different types of machine learning algorithms on a wide variety of Atari 2600 games. Focus then shifted to testing multiple algorithms that would be robust enough to work on different games and improve performance compared to human players. Q-learning has been used as a simple but elegant solution for the game *Breakout*.

1 Introduction

While there are numerous tools available for learning machine learning algorithms, many are data-driven with little to no graphical interface. The JStella Learning Environment (JSLE) is a system designed to be more accessible and visually interesting to students. The JSLE can be used in a classroom setting to teach students how to write different types of learning algorithms. To make this possible, the original JStella system was overhauled to allow other code to interface with it. Game information had to be made available for agents to access, and the game had to be controllable by an agent sending it actions.

2 Creating the JStella Learning Environment

The Stella Atari 2600 emulator, software that allows one system to behave like another, allows users to play Atari 2600 games on their own computer [1]. JStella is an open-source, community-made Java implementation of the Stella system [2]. Its open-source nature made it possible to add additional code to the JStella system to allow users to write their own player agents. The emulation core of JStella has been largely left untouched to allow a completely authentic experience to that of the original Atari 2600.

2.1 Emulation Speed

Speed was a major concern, as many learning algorithms need to play thousands of rounds on a given game to learn how to play effectively. Each frame in the original JStella system was rendered for 17 milliseconds, or 60 frames a second overall. This means a game could take well over a minute to play and even the simplest learning algorithms may require days of runtime. Without compromising the emulator, it was possible to set the render time as low as one millisecond. This allows algorithms to take hours instead of days for an equal number of trials. However, this hard lower bound on run-speed still ultimately limits the potential of the JSLE for large amounts of trials or for very complex algorithms.

2.2 Agent and Game Information

To make any learning algorithm work, a user must create a Java class for their agent. The agent will observe the game's state and then send some control action(s) back to the system to perform. This agent will be paired with an existing game class in the system, or a new Java class if the game is not supported already. Agents are automatically called by the system after every frame rendered by the emulator, which allows the agent to get the most up-to-date state information. After an agent observes this information, it sends some action(s) back to the system. The emulator then processes this information and generates the next frame of the game. Each game class contains a list of valid inputs or actions. Since the Atari 2600 only had a joystick with a single button, there is a maximum of 18 inputs.

Game information is held in a class specifically tailored for that game. The JStella system has an emulated 128 bytes of RAM each holding an integer from 0 to 255. This holds all necessary information during runtime, such as player position, game score, etc. For each game the RAM is laid out differently, and key values need to be found by hand. Luckily

these values are always in the same memory locations across multiple runs of the same game, so the process only needs to be done once. This game class then serves as a tool to group useful information from the game's RAM, while ignoring memory regions that are empty or useless. Each game class allows for full customization by the user to decide what values they would like to track and send to their agent.

3 Q-Learning with Breakout

Q-learning is a reinforcement learning technique that uses positive and negative reward associated with some actions [3]. An agent will hope to maximize its long-term reward by using localized feedback based on the state of the game. Q-learning also employs a randomness percentage that starts at 100%, and is reduced every few episodes until it is zero. Randomness usually helps the agent find states that give high reward that it may not have seen by acting rationally. This algorithm was then paired with *Breakout*, a game where the player controls a paddle and tries to bounce a ball into rows of bricks to earn points, shown in Figure 1.

Figure 1: The Atari 2600 game Breakout

Breakout has 6 rows of 18 bricks each. The bottom two rows are worth 1 point each, the middle two 4 points each, and the top two 7 points each, for a maximum of 432 points.

The following two tests were each given 1,600 30-second episodes to play *Breakout*. Each episode consisted of 30,000 actions sent after every frame of the game. Each test started by taking random actions and slowly reducing randomness until episode 640, when the agent was no longer random. Both tests used the same set of three actions: left, right, and no move (which causes the paddle to stay in one place).

Test 1 used a state space made up of an abstracted version of the ball's *x* and *y* position, the direction it was moving (also split into x and y components), and its speed. The ball's *x* position was split into 6 sections: left of paddle, right of paddle, and four regions based on locations on the paddle that yield different bounce directions. The *y* position of the ball was split into ten 20-pixel regions that comprise the height of the playfield. This gave 3,174 states for a total of 9,522 state-action pairs. The agent in this scenario received 1000 units of reward every time the paddle successfully hit the ball and lost 1000 units every time the ball fell off the screen. It also incurred a penalty of 1 unit for each movement to the left or right, which incentivizes the agent against unnecessary motion.

Test 2 used a state space consisting only of the same abstracted version of the ball's *x* position as used in Test 1. This gave a much smaller set of 6 states, for a total of 18 state-action pairs. The agent in this test-case was rewarded with 8 points when the ball was above the center of the paddle (the middle two of the four over-paddle segments) and 3 points when above the edges (the outer two segments). This test also incurred a penalty of 1 unit for any of the three actions selected. Figure 2 compares the high scores for each test averaged over every 5 runs.

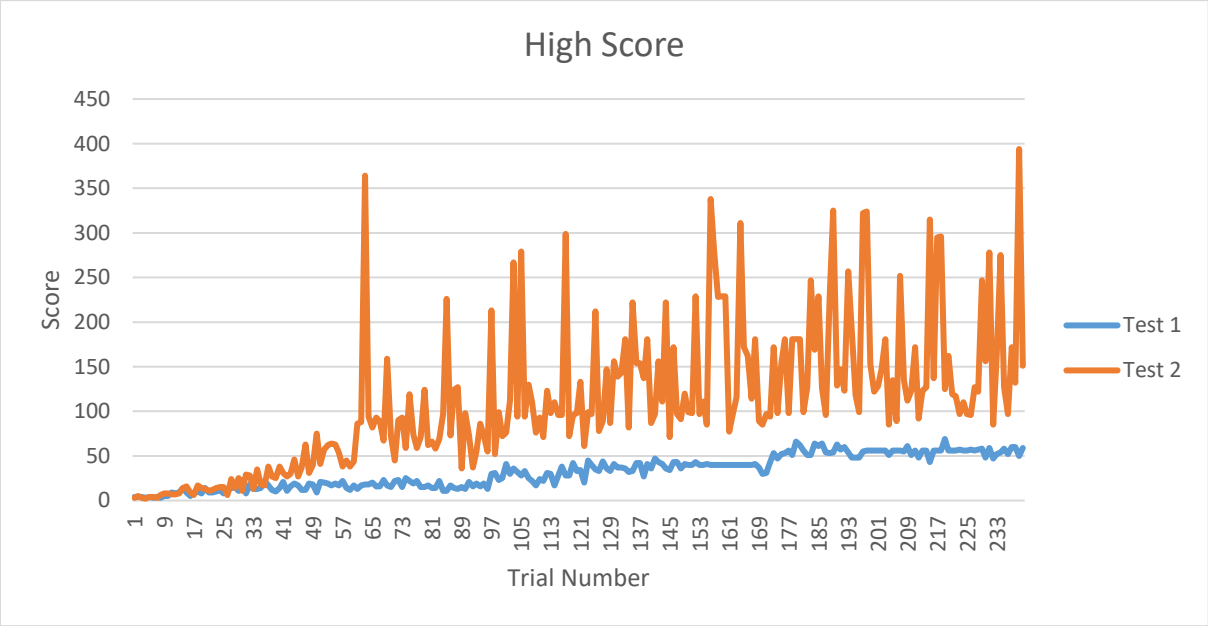

Figure 2: High scores achieved during runs of Test 1 and Test 2

It can clearly be seen that while Test 1 was more consistent, Test 2 was able to vastly outperform Test 1 in its ability to earn points. This came as a surprise due to the simplicity of the state space and reward function for Test 2 compared to Test 1. Interestingly, the more successful agent quickly develops a simple heuristic policy of staying directly under the ball as much as possible; in turn, this simple policy is almost able to maximize performance, coming close to clearing the board on many games. More work continues to be done, seeking to improve performance on *Breakout* and other games to the level of (or above) human player performance.

References

[1] The Stella Team, "Stella: A multi-platform Atari 2600 VCS emulator", stella-emu.github.io, 2017

[2] The JStella Team, "JStella – Atari 2600 Emulator", jstella.sourceforge.net, 2008

[3] C. J. Watkins and P. Dayan, "Technical note: Q-learning," Machine Learning, vol. 8, pp. 279–292, 1992.