Becoming the Jump King with Reinforcement Learning

Al Ferreras and Demetri Cassidy
University of Massachusetts Lowell
Lowell, Massachusetts
{al_ferreras, demetri_cassidy}@student.uml.edu

***Abstract*—We present an artificial intelligence agent that's able to learn policies for navigating a video game's environment through the use of an actor-critic method. Our agent was created using an artificial neural network that takes in the position of the game's player character on screen as coordinates and returns probabilities for each action the player character can take based on potential future rewards. From here, the agent samples an action from these probabilities, executes it, calculates rewards for that action, and adjusts its parameters based on its performance. Once the agent was implemented, it was trained to play the video game Jump King without any major changes to the agent's structure or algorithm. Having trained the agent for several hours, it has been able to progress through two of the game's screens.**

I. Introduction

Jump King is a video game that was developed and published by the Swedish company Nexile in 2019. In it, the player controls the game's eponymous main character as he climbs various platforms to reach the top of a tower. Although the game has a small amount of mechanics that are relatively easy to understand, it requires a great amount of precision to progress through it. It is because of these factors that we believe Jump King provides a unique opportunity to test the efficacy of deep reinforcement learning algorithms. For this project, we chose to implement the actor-critic method, which combines the benefits of actor-only methods and parameterized policies with those of critic-only methods and value function approximation [1]. With this method of deep reinforcement learning, our goal is to create an artificial intelligence agent that is able to learn from the environment of Jump King and progress through two in-game screens.

II. Literature Review

Deep reinforcement learning is a subset of machine learning that combines aspects of reinforcement learning, such as learning from an environment through trial and error, with other aspects of deep learning, like using an artificial neural network (ANN) to learn from high-level inputs. Additionally, this method allows for the agent to retain learned information as it progresses through thousands of timesteps. This area of artificial intelligence has shown to be successful in accomplishing similar goals with other video games. For instance, Deep Q-Learning (DQL), which uses ANNs to accomplish Q-Learning instead of Q-tables, has been used in the past to have agents to learn how to play Atari 2600 games [2]. In this case, the agent that Minh et. al created observes an image from an Atari 2600 emulator, determines the current state of the game's environment, chooses an action for exploring or exploiting the environment, and updates the ANN based on the reward for that action [2].

Although this implementation accomplished its intended goal, it may not be able to produce similar goals with other video games. In Jump King, navigating the game's world requires precise movements, which makes learning an effective policy a long and difficult process for a DQL agent. Although exploration can be utilized to speed up this process, this would still be inefficient due to the nature of the Q-learning algorithm. One possible solution is to implement a modified version of DQL known as Deep Recurrent Q-learning (DRQL). As detailed by Justesen et. al, what separates a DRQL agent from a DQL agent is the addition of a recurrent layer in the structure of its ANN just before the output [3]. This layer, which usually has either a long short-term memory or gated recurrent unit architecture, allows for an agent's network to feed the output of a previous state back into itself, along with new input [3]. As a result, the agent is able to better select optimal actions in a partially-observable environment by remembering information from previous states [4].

In order to evaluate the effectiveness of DRQL, Lample and Chaplot designed an agent that utilizes this method of deep reinforcement learning to figure out how to play the game Doom. Once it was implemented, their agent was able to extract information about the game's states from each frame of the game and feed it into a long short-term memory ANN, along with an additional layer of neurons for handling the game's features. From here, the ANN would train the agent's Q-learning, while the hidden layer would handle training the game features. As a result of this, the agent became able to navigate single player levels, detect and kill on-screen enemies, and pick up power-ups without many deaths or suicides [4]. While DRQL has proved to be successful in teaching an AI agent to play a complex game like Doom, it is much harder to implement compared to a traditional DQL agent. As described by Lample and Chaplot, while it is relatively straightforward to implement a DRQL model by itself, implementing it on top of a DQL model, along with getting it to work with the game features required much additional work [4].

Although DQL and DRQL could've fulfilled our need for a reinforcement learning method for our agent, we instead went with using an Actor-Critic method. Much like DQL and DRQL, an actor-critic method utilizes aspects of Q-value-based learning in order to learn from its agent's environment. However, what makes it different from other methods is that those aspects are combined with other aspects of policy based-learning, such as using value approximation functions to evaluate actions already taken[1]. By doing this, an agent is able to learn from an environment more efficiently when compared to one utilizing traditional DQL [1]. While it might not able to handle more complex environments like DRQL can, it is also less difficult to implement and maintain.

III. Methodology

At the heart of our agent is an ANN created with Keras and Tensorflow that carries out the learning process for the Actor-Critic Method. The network's architecture consists of a critic layer containing a single neuron, an actor layer containing a neuron for each possible action that the agent can take, and two neuron layers with default dimensions of 1024 and 512 neurons. These last two layers are shared between the actor and critic layers. The actor layer is responsible for returning tensors containing probabilities for each action, while the critic layer uses a value approximation function to critique and update the performance of both the actor layer and itself. Unlike the shared layers, which utilize the standard ReLU activation, the actor layer utilizes Sigmoid activation since unlike the ReLU and Softmax activation methods, it doesn't cause the probabilities it returns to zero out and break. The ANN's learning rate, α, is at 0.0001 by default, since if it were much higher, it would also cause the actor's probabilities to zero out. This architecture also contains weights for all shared neuron units, the action units in the actor layer, and for the critic neuron. This differs from DQL which doesn't make use of weights for the actor neurons.

When our agent begins to run through a set of episodes, it immediately resets Jump King, creates a new save file, and finds the current position of the player character within the game's environment. This is achieved through the use of OpenCV, which is a commonly-used computer vision library, the Python Imaging Library, and a picture of the character's head. Basically, the agent takes a screenshot of the window the game is running in and finds the (x, y) coordinates of the player character within that image using OpenCV's object detection method. Once this is done, the agent identifies which screen the character is on by cropping the screenshot into a 50-pixel wide image and using the same object detection method to find the image within a composite of cropped images from the first five screens of the game. This is especially important since it allows accurate rewards to be given when the agent progresses past the game's first screen. The (x, y) coordinates and the screen number are then returned as a tuple.

After this, the agent passes the position and screen number to its ANN, causing the actor layer to return a tensor with action probabilities. Depending on the agent's current exploration rate, which starts at 0.5 for new models or 0.01 for models that are well into training, it can either obtain an action by sampling a probability from the tensor or select a randomly picked action in order to perform exploration. From here, the agent executes the action, locates the player character's new position, and calculates a reward based on the difference in height and which screen the character is present on. The agent then stores the new reward and position in memory and trains the ANN using that same information.

Whenever learning needs to occur, the first thing that happens is that the agent converts the old and new states into tensor objects that are then passed into the ANN. Afterwards, the critic layer returns the approximate values for both the new and previous states. Then, the agent obtains the action probabilities for the previous state from the actor layer, calculates the categorical distribution of those probabilities, and obtains the log_prob for the agent's l last action from that distribution. After all this, the agent computes the temporal difference (TD) error of the last state transition with the equation:

$$\delta(s,a,s`) = R(s,a,s`) + \gamma V(s`) - V(s)$$

where $R(s,a,s`)$ is the reward for transitioning between states and γ is the discount factor (we used $\gamma = 1$).

Once the TD error has been calculated, the agent finds the total loss of the last step by adding together the actor loss ($-log\_prob*\delta$) and the critic loss (2δ). From here, the gradient of the total loss, with respect to the trainable variables, is computed and applied to the ANN's optimizers.

This entire process is what makes up a step for the agent. Each episode that the agent performs is comprised of 50 steps. After each episode, the agent saves the average score, along with the average score's history over all 50 steps, into a .csv file. This file will be used for graphing the agent's performance over time. Additionally, the agent reduces its learning rate after each episode by 20%. Whenever our agent is ran, it terminates after executing 10 episodes.

IV. Results

During testing, we ran our agent against Jump King 30 times, allowing it to learn about the game's environment over 300 episodes. Due to how time-consuming each episode is, this learning process lasted for well over 24 hours. Fortunately, since the agent was able to reach the top of the second screen, we were able to achieve our initial goal. However, as we graphed and analyzed the data that we've collected, we found that the agent's performance was more volatile than we expected.

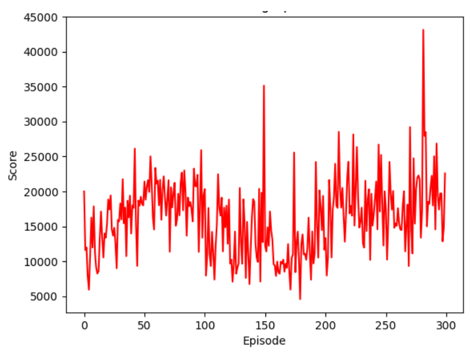

Fig. 1. The scores that our agent received over 300 episodes

As shown in the graph above, there is a slight upward trend in the average scores for about the first 75 episodes. At that point, despite some spikes, the averages begin to trend downward until around the 150 episode mark. From there, the average scores begin to increase again until the end of testing. Despite this, as the graph below shows, there is a consistent upward trend in rewards as the agent performs more steps. This shows that it is successful in learning from its environment over time.

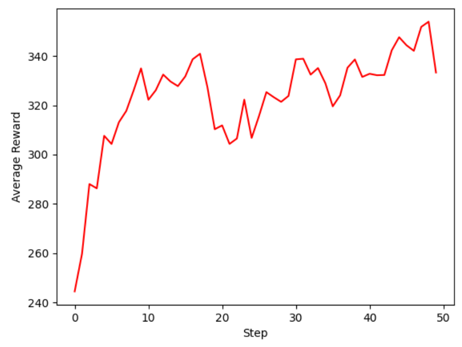

Fig. 2. The average reward that our agent received at each step over 300 episodes

When compared with the results from the studies referenced earlier, the performance of our agent performed somewhat worse than the ones utilizing DQL and DRQL. Despite having volatile performance that's similar to our agent, the average scores for the agent Minh et. al utilized consistently trended upward throughout 100 episodes [2]. Similarly, Lample and Chaplot's agent experienced no downward trends during its 70-hour training period, but had less volatility compared to the agents created by Minh et. al and us [4].

V. Discussions

Based on our results, it is clear that an artificial intelligence agent that is making use of an actor-critic method for learning is capable of learning how to play Jump King. What this implementation excels at especially is its ability to learn how to navigate a reasonable amount of the game's map with not much training. Had there been more time to work on this, we would've focused on training the agent for even longer to see if it would help it make progress in the game's third screen. Along with this, we would also focus on optimizing our implementation of the actor-critic method in order to get its performance to be more consistent. In addition, we only allowed our agent to execute a limited amount of actions within the environment for the sake of brevity. If we were to do additional testing, we would like to extend the agent's list of actions to include the rest of the player character's set of possible moves.

In addition, there were a few other issues with our agent that we were unable to fix due to time constraints. For example, running games for training would take an unreasonable amount of time due to lengthy animations of the player character jumping towards and falling down platforms. Additionally, there were instances during training where the character would land in an area that's obscured by objects in the game's foreground, which caused our agent to stall without manual interventions. Ideally, we would like to figure out how to speed up the duration of each episode and to have the agent avoid these foreground objects automatically.

VI. Conclusion

Despite less than desirable performance, we are satisfied with the agent's ability to reach our goal. For us, it serves as a great starting point for continued work in this sub-field of artificial intelligence. In the future, we would like to expand upon this research by implementing more agents that instead utilize DQL and DRQL for learning the environment. Even though it would be a difficult and time-consuming task, we are interested in the idea of how their performance in learning how to play Jump King would compare with our Actor-Critic agent.

References

[1] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Society for Industrial and Applied Mathematics*, vol. 42, 04 2001.

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

[3] N. Justesen, P. Bontrager, J. Togelius, and S. Risi, "Deep learning for video game playing," 2019.

[4] G. Lample and D. S. Chaplot, "Playing fps games with deep reinforcement learning," 2018.