Training an Agent to Beat Donkey Kong using Q-Learning

Intro Artificial Intelligence | University of Denver | Catherine Tifft

Throughout the history of developing AIs, playing games as well as or better than humans has been a benchmark for success. The most famous example of this is computer chess. Inspired by the Pacman examples from class, my goal was to develop and train an AI agent that can beat a Python version of the first level of classic Donkey Kong. Starting from random actions (staying still, moving left or right, jumping, or attempting to climb), the agent should first learn to reach the top of the course in as few steps as possible. Then this knowledge should be applied to a course populated with barrels and flames for Mario to avoid. Q-Learning was chosen to train the agent as it is an often used strategy for arcade games, with Deep Q-Networks being the first algorithm that achieved human-level performance on many classic Atari games (Badia). In this report, I will explain my implementation of the Q-Learning algorithm, the results of training the agent, and a discussion of what I learned from the project and considerations for future iterations of the project.

To convert the python code for use with Q-Learning, I first built a QAgent class to keep track of values returned from state-action pairs and to store them in a q-table. I also built a class to hold the game environment which connects an Agent (the brains of the operation) to the Player class that, in the regular program, is controlled by user inputs. This required moving the game logic into a step function so that individual rewards could be calculated for each action chosen by the agent and applied. The reward that is used in the update rule, which uses temporal difference learning. These actions update the game physics just as user input would. The agent also needs the hyperparameters alpha (learning rate, controls value of new information learned), gamma (discount factor, controls value placed on future rewards vs immediate rewards), and epsilon (exploration rate, controls how much action choice depends on random chance). I set these to 0.05, 0.95, and 1.0 respectively. These parameters mean the agent will need a lot of experience and values future rewards (reaching Pauline) more than immediate ones. Epsilon decays at a rate of 0.995 per episode so that as the q-table values become more accurate they are considered more by the agent. Rewards were set to +100 for a victory, -100 for a failure, and -.1 as a step penalty. To develop a baseline policy for the agent, I first trained it on a simplified map that had no obstacles/enemies (barrels and fireballs). A failure was defined as Mario falling out of the window. Mario's goal in this run is to learn to value moving toward ladders and climbing up them. For this version, I ran 1000 episodes and then saved the final policy for use in the complex, more dangerous map. After implementing the barrels and flames, I added new features to the state for Mario's distance from the nearest barrel and flame. I also made the rewards system more complex: -.1 step penalty, 500 for a win, -250 for death, and 2*distance moved vertically (this rewards moving upwards).

On the simplified version of the map, Mario successfully reached the top of the course on 86.5% of the runs. The exploration rate, which is decayed after each episode, stabilizes to the

minimum I set (0.05) at 600/1000 episodes. The highest rewards received were small positives (less than 100), indicating that the agent did not learn to avoid unnecessary steps well enough, reaching the victory state only after a lot of fumbling, noisy movement. This is easily visible when running with the learned policy with render settings turned on. That being said, there was some learning of reducing total steps: the absolute maxima of Fig. 2 was from one of the first episodes and extremely high step counts lessened as episodes went on, with a noticeable interval from around 650 to 1000 (around when the exploration rate comes to its minimum) where both step counts and total rewards seemed to somewhat stabilize with less major spikes from random actions. Overall, the trends on Fig. 1 and Fig. 2 are not exactly impressive, but they do exist. It seems that the agent learned to make it up the course in around 100 episodes, then stagnated trying to increase the reward through reducing steps.

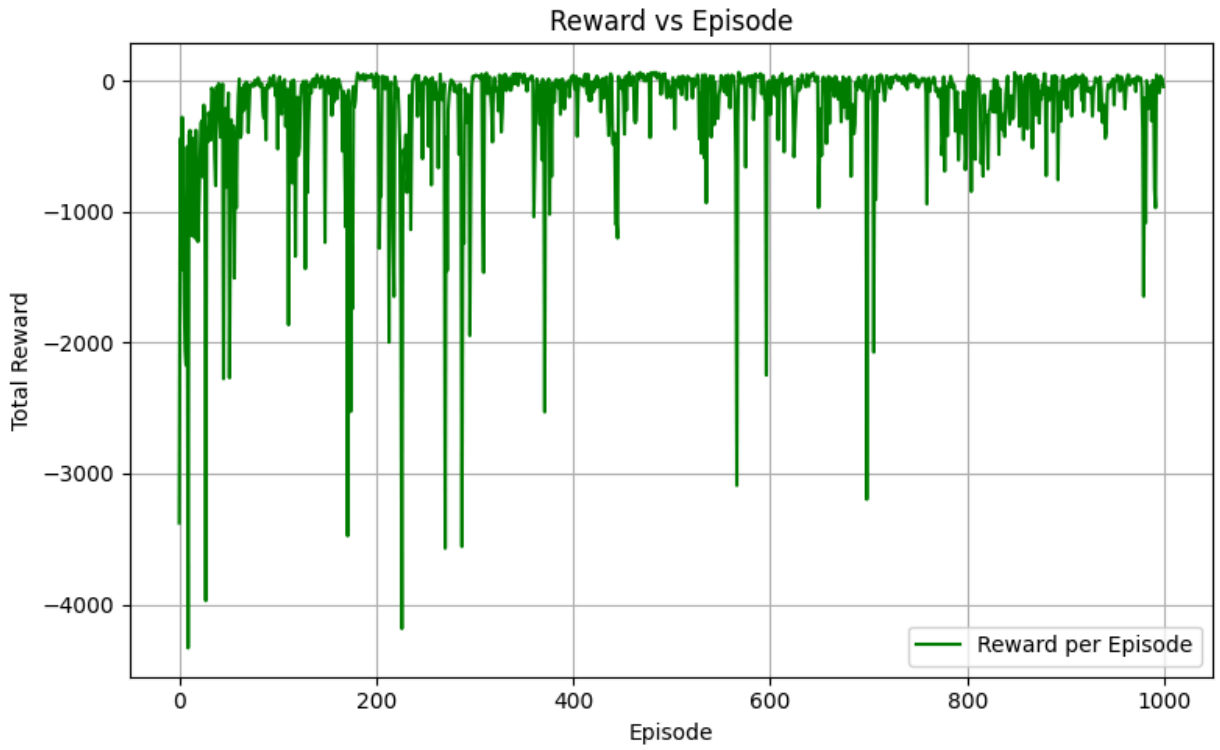

Figure 1

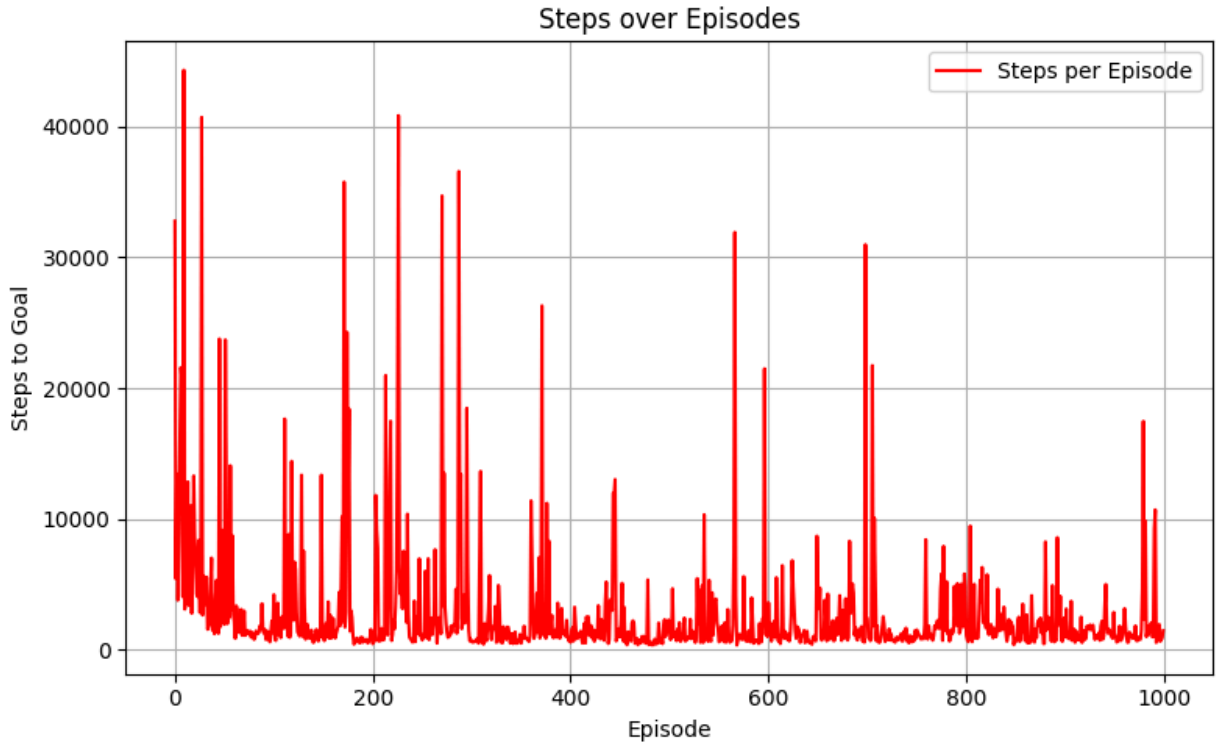

Figure 2

Despite the additions to the state representation and the reward system, the agent was much less successful when barrels and flames were added. I experimented with the hyperparameters, reducing the learning rate to 0.01 and increasing the decay on the exploration rate to 0.9995. After 3000 episodes, there was a victory rate of only 0.79% percent. A 500 episode run with fully randomized action choices obtained a victory rate of 0.61%, making the learned policy for this version only slightly better than a completely random chance policy. I worried that the previously saved policy from the simple version I loaded into the agent might have been unexpectedly decreasing the performance, so I ran 1000 episodes starting with a blank q-table, but the agent performed even more abysmally, reaching the victory state only once. This at least confirmed that starting learning with a simplified map was a bit helpful for the agent.

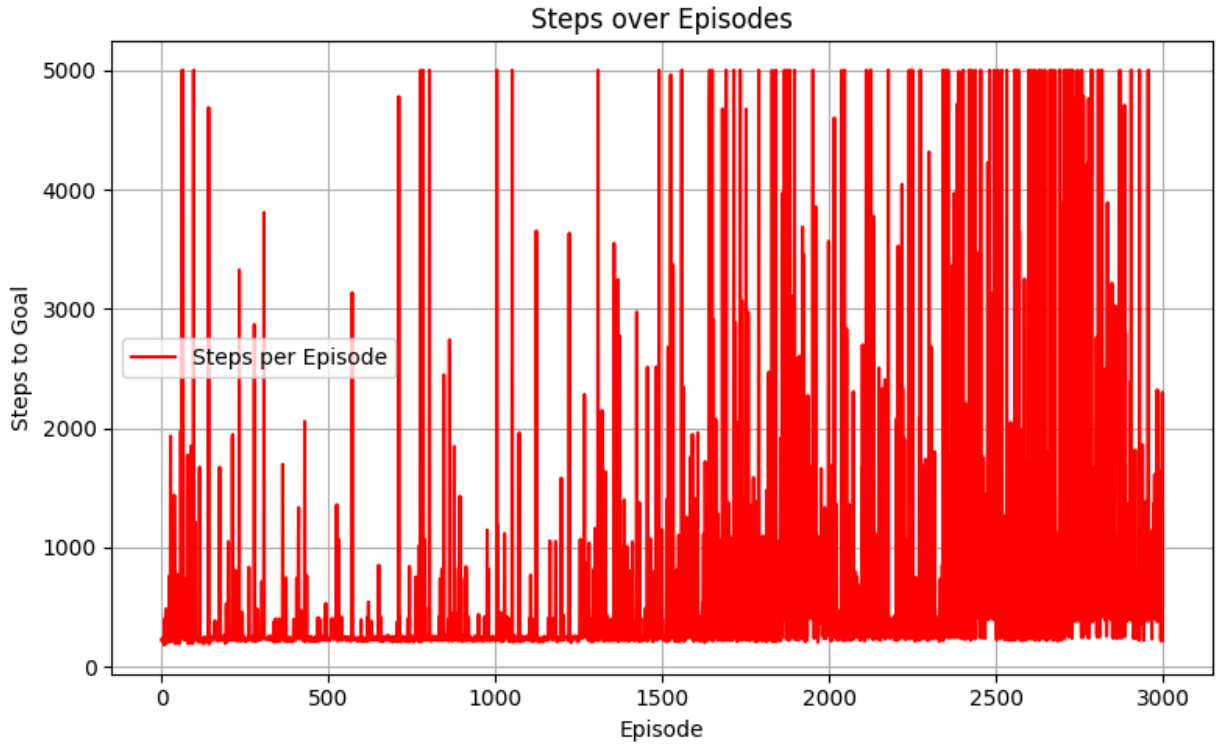

Figure 3

I noticed the agent getting stuck quite frequently, so I set a step limit of 5000. In Fig. 3, it is clear that the agent was matching this limit often. Interestingly, performance seems to get worse over time with low step counts rarer as episodes progress. One theory I have for why this is happening is the agent may be over-prioritizing avoiding enemies, leading to it jumping around avoiding enemies without much effort in trying to progress up the course. To see if I could confirm this, I watched Mario with the complex-map learned strategy loaded then visually compared this to Mario on the simple map with the simple-map learned strategy loaded. It appears that simple-map Mario is more inclined to move toward the ladders and to attempt climbing them whereas enemy-map Mario flails left and right with more jumping and less climbing. This is not scientific evidence, but I think it supports my theory for enemy-avoidance being over rewarded.

Figure 4

Unfortunately, something very odd happened with the reward graph (Fig. 4). It seems rewards were scaled by around 1000 considering total rewards were usually between -200 and -400. I am honestly not sure how this happened or why each reward is mirrored on the positive quadrant. A sample of every 5 episodes (I only output every 5 episodes, trusting in pyplot) from episodes 2000-2100 gives an average total reward of -320.48 and an average step count of 750.1.

In conclusion, the Q-learning agent was fairly successful at learning to beat a simplified map with only platforms and ladders with no enemies. It is important to keep in mind that I trained the model on one unvarying level, and it is unclear how well the policy learned from this training would generalize to other levels of Donkey Kong. However, I believe that adding state features such as proximity to ladders and distance to the top platform would allow for the agent to learn a more general strategy. When complex features such as barrels and flames are added to the environment, my implementation of Q-learning is not sufficient. Possible strategies for improving the implementation would be finer tuning of the hyperparameters, an expansion of state features, and a more carefully designed reward system. I prioritized future rewards, but it may have made more sense to reduce the gamma for the enemy-map version as Mario has an additional goal of avoiding enemies instead of just racing to the top. Additionally, the state representation was admittedly limited and would possibly be improved by tracking distance to the nearest ladder and distance from the goal platform. As for the rewards system, I would adjust the rewards related to enemies, add rewards for climbing up, add penalties for climbing down,

and add penalties for staying around the same area for too long. The physics of Donkey Kong such as jumping/falling (for both Mario and the barrels and flames) make Q-learning difficult as there are a large number of possible states for Mario compared to a game like Pacman where movement is restricted to a grid. In a future version of this project, I would experiment with balancing state representation features and managing the size of the q-table. While I did not achieve a very impressive policy, I am happy that in both versions of the environment Mario was able to beat the level as this was my original goal for the project.

References

Badia, Adrià Puigdomènech. (2020). Agent57: Outperforming the Atari Human Benchmark. Cornell University, 1-9. https://doi.org/10.48550/arXiv.2003.13350

Fan, Ian. (2018). Reinforcement Q-Learning for Atari Games. Wolfram Community. https://community.wolfram.com/groups/-/m/t/1380007

Ozkohen, Paul. (2018). Learning to Play Donkey Kong Using Neural Networks and Reinforcement Learning. *Communications in Computer and Information Science,* 1-16. https://doi.org/10.1007/978-3-319-76892-2_11

Q-Learning in Reinforcement Learning. (2025). Geeks for Geeks. https://www.geeksforgeeks.org/machine-learning/q-learning-in-python/

Zhou, Xiaolin. (2022). Optimal Value Selection of Q-learning Parameters in Stochastic Mazes. *Journal of Physics: Conference Series,* 1-9. https://iopscience.iop.org/article/10.1088/1742-6596/2386/1/012037/pdf

Base code for Donkey Kong from https://github.com/plemaster01/PythonDonkeyKong