Practical 4: Reinforcement Learning

William Burke, Buse Aktas, Matthew Holman
wburke@g.harvard.edu, buseaktas@g.harvard.edu, mholman@cfa.harvard.edu
wburke, BuseAktas

1 May 2018

1 Technical Approach

In this practical developed a reinforcement learning agent to play Swingy Monkey, a game in which a player controls a monkey, trying to avoid tree trunks while swinging from vines. There are two possible actions at each state: jump to a new vine, or keep swinging on the current vine. A player scores 1 point when the monkey passes a tree trunk without hitting it. Hitting a tree trunk results in a penalty of -5. Falling off the bottom of the screen or jumping off the top of the screen results in a penalty of -10.

There are four sources of randomness in the game; the value of gravity, the distances between the trees, the heights of the gaps in trees, and the strengths of jumps are random. This makes it hard to write down a set of rules which will work in all cases.

We implemented a learner which decides to jump or not to jump at each time increment, based on its current state and the corresponding data in a Q table for each available state-action pair. The state returned by Swingy Monkey includes the variables in the below table.

As we observed through our examination of Swingy Monkey and repeated trials, some of the state variables are repetitive, and not necessarily directly useful for a learner agent. Since the tree trunk always has a constant gap height, the top and the bottom of the gap are linked to each other. A similar relationship exists between the top and bottom of the monkey, as the monkey's height never changes. The score of the current state is irrelevant, since we are already accounting for the +1 reward for each time the monkey passes a tree trunk without hitting it. Inspecting the Swingy Monkey source code also showed us that there is another important parameter which affects the dynamics of motion: the gravity. It can only take the value of either 1 or 4, and it is randomly determined for each game played. Since this is an important part of the environment/state to account for, and since it is easy to calculate using the two initial time steps, we always extract the gravity before taking any actions.

State Values Directly Pulled from Swingy Monkey

State Element	Integer Pixel Values
Score	integer value between 0 and inf
Distance to tree trunk	integer pixel value between 0 and 600
Top of Tree Trunk	integer pixel value between 0 and 400
Bottom of Tree Trunk	integer pixel value between 0 and 400
Velocity of Monkey	integer pixel value between -75 and 20
Top of the Monkey	integer pixel value between 0 and 400
Bottom of the Monkey	integer pixel value between 0 and 400

Processed and Binned State Values

State Element	Possible Values
Fall off Screen	0 or 1
Tree Danger Zone - Vertical Plane	-1 or 0 or 1

We decided to implement a Q-learning algorithm for our agent. (For this we followed notes from both CS181 and CS182.) This approach calculates the quality of each state-action pair as it explores. These Q values are initialized to 0. When the agent is at state s and takes action a, it observes a reward $r(s, a)$ and it reaches state s'. The observation at each state is the reward plus the maximum possible Q value for the next state reached. Then we take a weighted average of the previously collected observations and the new ones and keep adding. This value iteration update, derived from the Bellman Equations can be seen below:

$$O(s, a) = r(s, a) + \underset{max_{a'}}{Q}(s', a')$$

$$Q^*(s, a) = (1 - \alpha) * Q(s, a) + \alpha * Q(s, a),$$

where α is the learning rate.

In order to make efficient use of this algorithm, we have to make sure that the learner can reach many states, returning to states with sufficient frequency to learn a policy. We tested using the states as defined in the first table at full resolution. However, the resolution of the Q dictionary was extremely large, and the low probability of the agent reaching any given state was very low. Thus we needed to find a way to further bin the above-mentioned states, with the goal of only passing enough information for the agent to be able to decide on a high-reward and low-penalty action at newly reached states. We initially just got rid of the repetitive state variables, and discretized them using different bin sizes. Despite this, the learning agent was not arriving at an effective policy.

As a first step in all cases, we determined the effective gravity from the change in velocity in the first time step. Given the position, velocity, and gravity, we can extrapolate what will happen in the future if the monkey does not jump. On the other hand, we can not deterministically extrapolate what will happen if the monkey jumps, because the jump velocity is assigned randomly at each new jump. We developed a simpler set of features, with a much smaller number of states, that allows the Q-learner to quickly develop an optimal policy. Essentially, this is whether the monkey will fall below the bottom of the screen in the next time step and whether the monkey will fall below the bottom of the tree in the next time step. The second table shows the states which we defined using the previous states. There are only six distinct states.

Once we arrived at this simple, effective approach, we tuned both the learning rate α and the ϵ-greedy exploration rate ϵ, and the jump probability during exploration so that we would explore the state space during the initial games, calculate an optimal policy, and then exploit that policy in the subsequent games. We used $\alpha = 0.9$. We used an exponentially decaying function for ϵ:

$$\epsilon = \exp(-f \cdot E),$$

where E is the epoch and f is a factor to adjust the decay time scale ($f \sim 0.1 - 0.5$). This allows more exploration in the earlier epochs, tapering to essentially no exploration at the end.

2 Results

The performance of our different decision algorithms can be seen in Figure 2. The learning algorithm clearly learns, as the average score gets an improvement over time. We have compared our learning

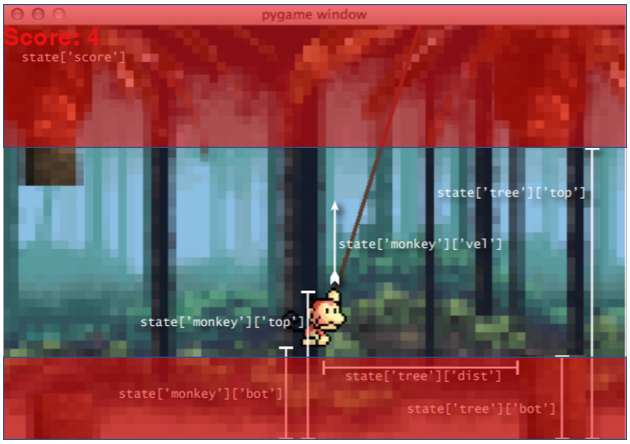

Figure 1: Tree Danger Zone, shown in red

Decision Algorithm	Average Score	Maximum Score
Baseline: Random Jump	0.115	1
Learner 1	177	1632
Manual	713	5395

Table 1: The average and maximum score for each the three approaches we developed and tested.

algorithm to two other algorithms, one is the very basic baseline where at each time iteration the agent performs a random jump with a low probability. The second we compare it to is one that calculates the possibility of a collision at each time step given the state and performs an action accordingly. This algorithm which we called "Manual" does no learning, it already knows how the system will behave. The learner we have implemented on the other-hand uses a Q-learning approach, collects data throughout epochs and tries to learn the dynamics the "Manual" algorithm already knows. Table 1 shows a numerical comparison of the average and maximum score for these three decision algorithms.

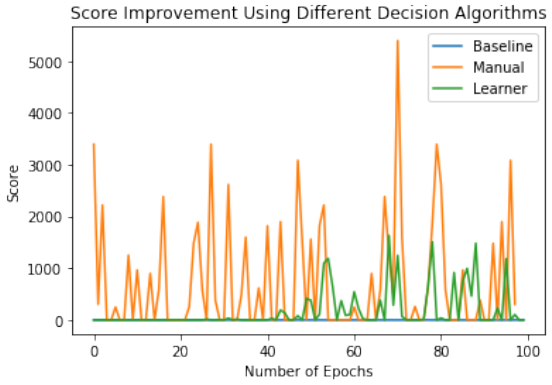

Figure 2: Scores with Different Decision Algorithms

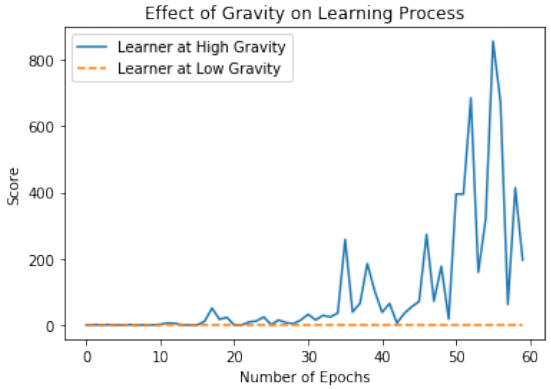

Figure 3: Gravity's effect on learning. We see the performance of our approach in the high gravity case is excellent, while that for the low gravity case is significantly poorer.

3 Discussion

Our initial, repeated attempts at Q-learning were not successful because most of the states in our Q-learning space were not being visited.

Changing strategy, as we attempted to write down the system dynamics for the "Manual" algorithm, we had a major break-though. Instead of keeping track of the absolute velocity, position of the monkey, and position of the tree, we realized that we could use combined variables. Instead of storing the position, the velocity, and the gravity, we could the predicted position at the next step if we did not jump, i.e. pos+vel-grav. Instead of storing the position of the monkey and the tree, we could store their relative position. As explained above, we simplified these variables as much as possible, with allowed values of just 1, 0, or -1.

Although we found that the two variables listed in the table above led to the best performance, we experimented with many more variables than these. For example, given the position, velocity, and gravity, we know the trajectory of the monkey and can predict whether the monkey will collide with the top or bottom of the tree or with the top or bottom of the screen. We tried adding dynamical information, like predicting collision locations and time steps. We also tried adding variables that would predict potential collisions that could occur as a result of jumping. Encoding these as boolean variables, we expected that incorporating this information would improve our performance, but this was not the case. Adding system dynamics did not improve performance.

We have two hypotheses regarding this. First, the smaller state space allows a more complete exploration and thus a more refined policy. Second, the state space is directly related to the expected outcome. Even though our learner does not use the system dynamics, we have introduced some knowledge of the physics into the state space. It is worth noting that we have not included detailed or long-term physics, just summaries of the current state variables, thus this learner should continue to be effective under other circumstances.

Figure 3 shows our results for different values of gravity. We found that our Q-learner achieves near-perfect performance for the high gravity ($g = 4$) case. After rapidly learning the policy, that only failures are due to random exploration from the ϵ-greedy algorithm. Our approach does not perform nearly as well for the low gravity ($g = 1$) case. The monkey frequently achieves a high enough velocity that it cannot recover. We explored a number of approaches to improve the performance, with limited success. We ultimately adopted the same approach for both gravity regimes. The average performance is excellent; to do better overall we could explore function approximation or deep learning approaches.