Practical 4: Reinforcement Learning

James Zatsiorsky, zatsiorsky@college.harvard.edu
Rajko Radovanovic,rradovanovic@college.harvard.edu

Github REPO: https://github.com/zatsiorsky/p4

04/29/2017

1 Approach

1.1 State Space

In order to build a Q-learning agent to play *Swingy Monkey* effectively, we needed to define a state space S over which $Q_{s,a}$ is defined, where $s \in S$. Our actions a take on the value $a =$ jump or $a =$ don't jump. We encode these actions as 1 and 0, respectively. Let us examine the state information that we are provided:

```
{ 'score': <current score>,
  'tree': { 'dist': <pixels to next tree trunk>,
            'top': <height of top of tree trunk gap>,
            'bot': <height of bottom of tree trunk gap> },
  'monkey': { 'vel': <current monkey y-axis speed>,
              'top': <height of top of monkey>,
              'bot': <height of bottom of monkey> }}
```

We immediately run into issues if we try to directly encode the information provided to us into states to use in Q-learning. First, we notice that the distance measurements are provided in units of pixels. On inspection of the `SwingyMonkey.py` file, we see that the screen is configured to be 600 pixels horizontally and 400 pixels vertically. Therefore, in trying to encode only the horizontal distance to the next tree and the height of the top of the tree trunk gap, we would need approximately $400 \times 600 = 240,000$ states. This only encodes a portion of the information provided to us, so we can see that we clearly need to reduce the size of the state space.

Our first order of business was coming up with a state representation that would result in a smaller state space S while still preserving the important information relevant to playing *Swingy Monkey* well. Intuitively, we understood that knowing the horizontal distance to the next tree is important in playing the game successfully. Therefore, we wanted to preserve the `dist` attribute of the provided state. However, in order to keep the size of the state space from blowing up, we decided to discretize the horizontal pixels into buckets. For instance, if we decided to discretize the horizontal pixels into five buckets, then our modified `dist` measurement would be one of $\{0, 1, 2, 3, 4\}$ depending on which bucket the monkey fell into. We let `md` be a tunable parameter that determines how coarse the discretization should be. A larger `md` value corresponds to a coarser discretization. We now define our modified `dist` measurement as:

$$\texttt{dx} = \lfloor \frac{\texttt{state['tree']['dist']}}{\texttt{md}} \rfloor$$

We took a similar approach to encode the vertical distance between the top of the monkey and the top of the tree trunk gap. We thought that this information was extremely critical to the success of our learner. For instance, if the monkey is far below the top of the tree gap, it is probably an indication that it is time to jump. We define the attribute `dy` as:

$$\texttt{dy} = \lfloor \frac{\texttt{state['monkey']['top']} - \texttt{state['tree']['top']}}{\texttt{my}} \rfloor$$

where `my` is another tunable parameter that indicates how coarse the discretization should be. We make this parameter unique from `md`, since intuitively we thought that a finer discretization would be needed in the vertical direction than in the horizontal direction.

We should note that we do not need the 'bot' attribute of the tree, since the tree gap size is a constant 200 pixels for each tree. Therefore, encoding the 'bot' attribute would be redundant and

increase the state space unnecessarily. If we know where the monkey is relative to the top of the tree gap, we know where it is relative to the bottom of the tree gap as well. We decided not to encode the vertical position of the monkey independently, as we hoped that the relative vertical distance to the tree gap would be sufficient to train a successful Q-learning agent.

After playing the game repeatedly by hand, we noticed that the gravity setting had a *significant* impact on the game's performance. It was difficult to even get past one or two tree gaps on the low gravity setting. We thought that it was necessary for our Q-learning agent to take into account the gravity setting. Therefore, our third attribute of our new state space is whether or not the game is in a high gravity setting or a low gravity setting, indicated as 0 or 1, respectively. We can easily compute whether or not the game is in a low gravity setting by comparing the velocity of the monkey across two time periods. In encoding this binary variable, we are essentially training two Q-learning agents: one for the low-gravity mode and one for the high-gravity mode.

The last feature in our state space S is the velocity of the monkey. At first, we encoded simply whether or not the sign of the velocity was positive with a 0 or 1. Given the other state features, we figured that a descending monkey might need to make different decisions than an ascending one. We also tried discretizing the velocity into buckets as we did with the distance measurements. Let's call this modified velocity measurement `v`.

To summarize, the states s in our new state space S are 4-tuples that take on the following form:

$$s = (\texttt{dx}, \texttt{dy}, \mathbb{I}_{\text{high-gravity}}, \texttt{v})$$

1.2 Choosing Actions

Given that we are in state s, we need to decide whether to jump (1) or to swing (0). We take an ϵ-greedy approach to selecting actions. With probability ϵ we *explore*, and with probability $1 - \epsilon$ we *exploit*. However, when we explore we do not want to choose each action with equal probability. Let us examine why this is the case. The screen is 600 pixels wide, and after each frame the monkey moves 25 pixels horizontally. Therefore, in the time it takes the monkey to move across the screen once, 24 states have been processed. If we jumped or swung with equal probability in the exploration phase, this would correspond to ≈ 12 jumps. This is clearly unreasonable, as the monkey should only jump a few times in the time it takes to traverse the screen once. Therefore, we let a new parameter `thresh` determine the probability that we take the jump (1) action in the exploration phase. We tune this parameter for optimal performance below.

Over the course of training, we decrease the value of ϵ, which corresponds to decreasing the amount of exploration we do over time. Once we hone into a good policy after many iterations, we want to focus on exploitation rather than exploration. We tuned the value of ϵ to find the amount of exploration that leads to a successful agent.

When we exploit with probability $1 - \epsilon$, we compare the Q values at state s for actions 1 and 0. We take the action that maximizes the Q value. If the Q values are the same for the two actions, we use the same `thresh` parameter to choose between jumping and swinging.

1.3 Taking a Shortcut

After playing the game multiple times by hand, we noticed that it is suboptimal to jump when the monkey is above the top of the tree gap. Therefore, in choosing an action a at state s, we also check to see whether the monkey is above the top of the tree gap. If the monkey is above the top of the tree gap, we always swing. This shortcut diverges from traditional Q-learning, but we found that it led to a successful learning agent.

1.4 Data Structure

We used a Python `Counter` data structure from the `collections` module to represent our Q matrix. We found that this was particularly convenient because the `Counter` value defaults to zero if the key does not exist. There is no need to instantiate $Q_{s,a} = 0$ for all $s \in S$, $a \in A$ on initialization. We simply key in on (s, a) in order to extract the $Q_{s,a}$ value.

2 Tuning

We tuned a number of parameters in order to find a combination that maximized the mean score of the learner. We did not tune the parameter γ since in the context of this problem, a value of

$\gamma = 1$ made sense. There is no reason to think that points in the game now are worth more than points in the game later. The parameters that we tuned are:

1. ϵ: the exploration rate
2. `thresh`: the jump threshold for exploration (1.2)
3. η: the learning rate
4. `md`: multiplicative factor to discretize horizontal distance `dx` (1.1)
5. `my`: multiplicative factor to discretize vertical distance `dy` (1.1)
6. `mv`: multiplicative factor to discretize velocity `v` (1.1)

Figure 1: Tuning the 'thresh' Parameter

Threshhold			
Row Labels	**Average of mean**	**Average of 50_perc**	**Count of death_mode**
0.001	79.32230769	1.384615385	13
0.005	74.96625	1.65625	16
0.008	75.1225	5.90625	16
0.01	62.125625	1.4375	16
0.025	44.48588235	1.235294118	17
0.05	22.23833333	1.388888889	18
0.1	10.44736842	0.657894737	19
Grand Total	**50.2753913**	**1.917391304**	**115**

What is important to understand about tuning parameters for qlearning is that each learning process is highly uncertain. With the same set of parameters we could achieve drastically different results. And so, every time a parameter was tuned, we had to make sure to have at least 30-40 repetitions of 100 games EACH, so we could get rid of noise in the performance of the algorithms. This is because the environment, or game, could be more or less hepful/nice to our learner. Sometime, with the same parameters, we would achieve a max score of 50 within the first 5 game iterations, and sometimes we wouldnt get that high even in 100 games, with the same set of parameters. Hence, all no conclusions about parameter effectiveness could be made without many repetitions, restarting the learning each time.

It also key to note that we were optimizing performance of 100 games, which could actually decrease performance in the long run, since there is a trade-off between the speed of the learner and the precision with which it can play later on.

In figure 1, we are in the process of tunning simulations and we begin to see a trend indicating that the smaller thresholds are better, meaning we want our monkey to be jumping less of the time on average, when performing randomly. The rightmost column, counts, indicates how many 100 game simulations we have run. The second and third columns give the average of the mean and median scores of each each 100 game iteration. With another 15-20 simulations, the trend became even clearer as we eliminated noise.

An additional note on tunning is that we first performed it over a large grid with 5-6 values for each paramater and then chose optimal sets of parameters from there. We then moved on to fine tune the parameters one by one towards the end. This mattered because many of these parameters were highly interconnected in relation to performance. For example, the exact bin size for speed and for vertical distance between the monkey and the trees definitely mattered, since whether we would have a collision or not was a function of both.

3 Results

After performing the tuning specified in section 2, we ran our Q-learning agent with the optimal parameters. We noticed that the performance on the high-gravity setting far-exceeded the performance on the low-gravity setting. It was nearly impossible to play the low-gravity setting by hand since the heights of the jumps are so variable. Therefore, it is possible that even an optimal policy would be unable to play the low-gravity setting effectively. We are quite happy with our results on the high-gravity setting. We display our optimal parameters in the table below:

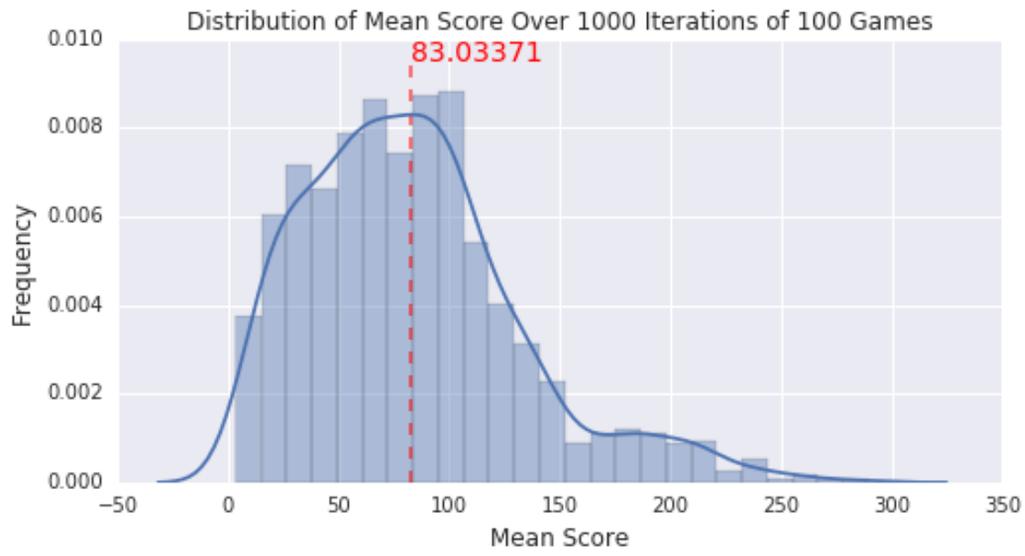
To test how good our learner performed, we played 1,000 iterations of 100 games and recorded the mean and max value over every game. On average, we achieve a **mean score of 83** over the

Table 1: 100-Game Optimal Parameters

Parameter	Optimal Value
eta	0.1
epsilon	0.225
thresh	0.001
md	119
my	36
mv	56

first 100 games played. On average, we achieve a **max score of 1,130** over the first 100 games played. The maximum max score we achieved over all 100-game iterations was 4,915. We show the distributions of these values below. The mean score seems interestingly bimodal.

Figure 2: Distribution of 100-Game Mean Scores



Lastly, we note that even our best parameter set was not able to achieve great scores on the low gravity version of the game. The max score ever achieved (across 1000 iterations of 100 games, with a new learner starting each time), was 10, very low compared to the high gravity version. You can see an example of how different the performance was for our algorithm depending on gravity in the plot of a 100 game iteration below. Note, interestingly, there is no clear linear learning trend over time in this example, which we found somewhat peculiar.

Figure 3: Distribution of 100-Game Max Scores

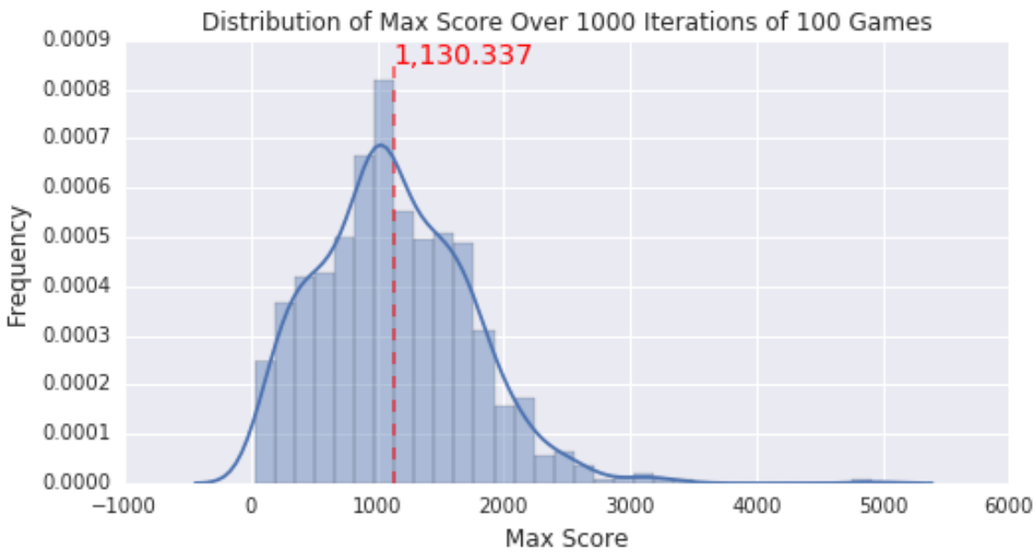


Figure 4: 100-Game Example