Practical 4: Reinforcement Learning

Team members:
Michael Kosowsky, kosowsky@g.harvard.edu
Lucas Orona, orona@g.harvard.edu
James Potter, jpotter@g.harvard.edu

May 6, 2019

(Note: our code has been submitted in Canvas, as well as GitHub. Our GitHub link is: https://github.com/jimmy-potter/Practical_4)

1 Technical Approach

In this practical, we created a reinforcement learning model to try to solve a game. Our goal was to teach a monkey to jump between trees without hitting them, the ceiling or the floor.

Our first decision was how to model this reinforcement learning problem. We decided to approach it by modeling the Q-function, since we are only given states but not the dynamics in a given epoch, meaning that we would have to model those hidden features somehow in our algorithm.

The other major initial decision was how to reduce the state space. The state space is large enough at any given point in the game that for the purpose of Q-learning, it is effectively continuous, making it almost impossible for the monkey to learn the entire raw state space. We decided to start with a very simple quantization/binning process in order to debug the Q-function itself and to determine the flexibility of the model. For our initial bins, we chose three variables from the state space dictionary: 1) the monkey's height from bottom, 2) the closest tree's bottom height, and 3) the horizontal distance to the nearest tree. We noticed that monkey did not change heights during the game, and the tree gaps were always the same size, so we were able to ignore two redundant variables from the state space. We noticed that the tree distance variable would sometimes become negative if no new tree had appeared to the right of the monkey yet, so we set a specific horizontal distance bin for when the tree distance is negative. For our initial testing, we ignored the velocity component, since we expected that we might want to model gravity and velocity in more complex ways. We started by dividing the three state space variables of interest into 100-pixel bins, which reduced the state space considerably.

We then added in velocity values to our state space, making sure to bin them as well to not increase the state space too drastically. Adding in the velocity values was important to determine what could happen in the future - predicting the future trajectory of the monkey is important to make sure the monkey jumps before it is too late.

The other necessary component to predicting trajectory is the monkey's acceleration. In the game, the monkey's acceleration at a point is due to the gravity in the system. Each initialization of the game chooses a value of gravity. Because gravity is not time-dependent, we were able to code our learning algorithm to figure out the system's gravity early-on in the game, by comparing successive velocity values (that occured without jumps in between them), and using the simple formula $a = \frac{dv}{dt} \approx \frac{\Delta v}{\Delta t}$.

Our original code made the decision to jump or not based just on the Q at the current state. We realized it would be better to average this with the Q of the state we will jump or fall to, as the decision to jump does not occur in a vaccum. We took a running average of the jump velocities and used this to predict which state we would jump to. This helped improved the consistency of our score.

In addition, we decided to retroactively reward and penalize whenever points were scored, instead of just rewarding or penalizing the immediate state before scoring the points. After scoring points, we rewarded the most recent jump command. We also penalized the last jump for any penalty incurred with the penalty being scaled by γ^d, where d is the number of steps in the past the jump was. This is because a point scoring jump is automatically good and should be fully rewarded, while a penalty incurring jump can either itself be bad (like jumping when directly underneath the tree) or merely a victim of a later inaction (we actually needed to jump twice, but never made the second jump). The γ on this helps to differentiate the two. Not surprisingly, we found that setting γ too low led to the monkey being too jumpy and γ too high led to the monkey not being jumpy enough.

We also decided to retroactively reward inaction. The monkey dynamics are deterministic and only evolve under the influence of gravity. Using this fact, we chose to reward any state that would deterministically evolve into a point scoring state (and so it wouldn't pay to jump).. Whenever a point is scored, we rewarded every state that would free fall into that point scoring state. We also penalized states that lay on the path to penalties. Again, we scaled the penalty by γ^d because these states may also be the victims of later inaction. We used this strategy because it is similar in spirit to Alpha Go's strategy of trying to achieve advantageous board configurations rather than trying to outright win the game from each spot. Our retroactive rewarding helps us to identify these safe, point scoring states in which we should do absolutely nothing and just collect the points.

We then included an auto jump protection for the monkey. In the low gravity state this occurs if he enters the bottom 30 pixels with negative velocity. For the high gravity case, this occurs if he enters the bottom 100 pixels with negative velocity. We added this feature because the penalty for falling off the bottom is known, and we do not need to explore this space. Not having the monkey have to think about these decisions improved the efficiency of our learning algorithm.

We also decided to improve our model by removing some of the binning and replacing it with what we called "QSmoothing." Because our state space is a continuous distribution, we chose to model every point scoring space with a Gaussian distribution, as we assumed that if in one state it was advantageous to jump, it would likely be advantageous to jump in some nearby state as well. For example, if state x scored 5 points, we would add $5 * f(\epsilon)$ points to some state $x + \epsilon$ close by, where $f(\epsilon)$ was a function that took values between 0 and 1, depending on the distance to the point-scoring state. The closer the state was to the known state, the more likely it would be for the state to want to take a similar action. Implementing QSmoothing allowed us to use a large number of states, binning velocity with binsize 5 and position with binsize 4, instead of the larger bins of 100 from before.

We found that while our QSmoothing allowed us to quickly solve the state space, it frequently led to cases where we would fall into a bad pattern (for example, after 20 epochs, the monkey would continuously jump until it hit the ceiling). We attempted to fix this problem by tuning the strength of our Gaussian (making it fall off faster with distance), as well as limiting how many points could be affected by a nearby point.

We then changed our QSmoothing function to apply only on the retroactive rewards. A point scoring state can be very close to a penalty incurring state, and so we found that smoothing over the current state tended to decrease performance by making the near term planning blurry. However, the long term planning is already blurry, and smoothing over this helps to ensure that we see the full gradient of states that can lead to the current state.

We also found that using randomness to explore the state space was unnecessary and decreased performance. The auto jump feature helped us to explore jumping and our averaging the decision based on present and future states, retroactive rewarding/penalizing and smoothing filled out the state space sufficiently without the need for randomness.

2 Results

The plot below shows the performance of our initial model ("basic model," in blue, as described in the TA section above) versus our final model (" final model," in orange, also described in TA). Both models were tested ten times, over 500 epochs, for this plot. The x-xaxis shows the epoch, while the y-axis shows the mean score for that epoch over the ten runs. We see that the final model consistently outperformed the initial one, in both maximum total score and for almost every epoch.

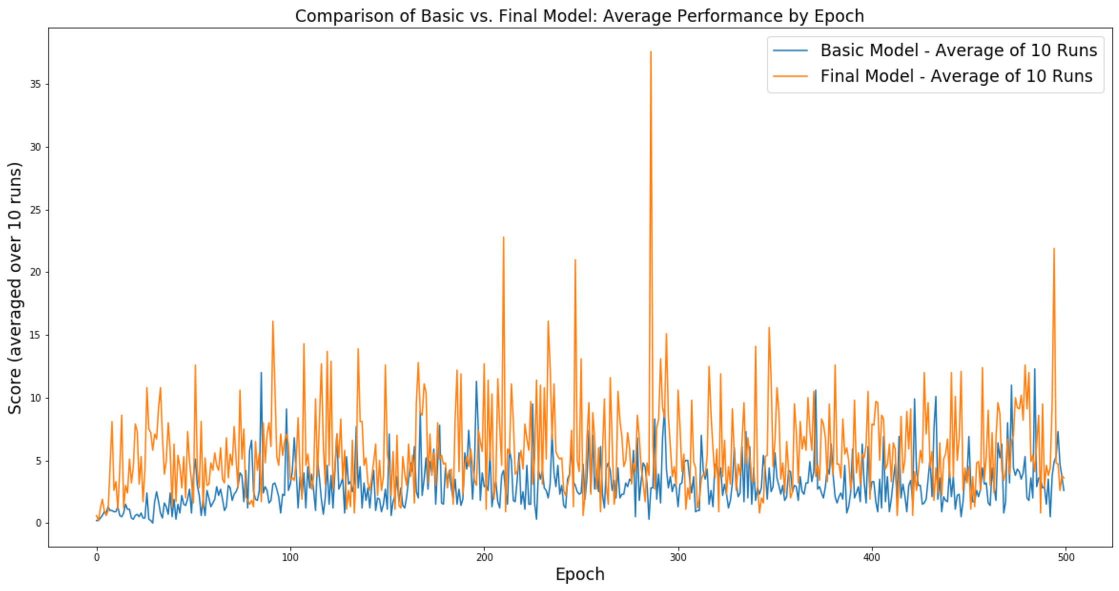

Figure 1: Performance of initial vs final model

The mean max score for the basic model after 500 epochs was 48.2, while the mean max score for the final model after 500 epochs was almost double that, 75.9.

The below table shows an example of how we tested our hyperparameters for each model. the table is from one of our earlier models. We varied our three RL hyperparameters. We ran each set 5 times, for 100 epochs, and recorded the mean max score and the standard deviation of that max. When choosing the hyperparameters to go with, we would want a combination of the model that had the best success (highest max score), while also being the most consistent (lower standard deviation).

Hyperparameter set	**Mean max score**	**StDev of max**
$\gamma = 0.8, \epsilon = 0.02, \alpha = 0.6$	15.4	8.2
$\gamma = 0.8, \epsilon = 0.03, \alpha = 0.5$	14.2	4.5
$\gamma = 08, \epsilon = 0.03, \alpha = 0.6$	23.0	18.4
$\gamma = 0.8, \epsilon = 0.04, \alpha = 0.5$	29.6	10.2
$\gamma = 0.8, \epsilon = 0.04, \alpha = 0.6$	13.0	5.6
$\gamma = 0.8, \epsilon = 0.05, \alpha = 0.5$	20.2	22.0
$\gamma = 0.85, \epsilon = 0.03, \alpha = 0.6$	11.4	7.0
$\gamma = 0.9, \epsilon = 0.03, \alpha = 0.5$	19.2	11.5
$\gamma = 0.9, \epsilon = 0.03, \alpha = 0.6$	24.8	9.7
$\gamma = 0.95, \epsilon = 0.03, \alpha = 0.6$	19.2	5.9

Table 1: Testing hyperparameters

3 Discussion

As is the case with most machine learning problems, there were two main things we had to consider for this practical - how to choose our features, and how to model them.

Our feature selection was simple at first. We initially started with just position values, for the distance above the ground, distance to nearest tree, and height of the nearest tree. We spent some time tuning the hyperparameters for this feature set. We decided that our feature set was not enough, and so we slowly increased our feature set to include both velocities and accelerations, as those would allow us to predict a future trajectory, and be able to make decisions based on the trajectory as well. In this problem, it is just as important to know when not to act as it is to know when to act, and knowing these trajectories allowed us to implement our inaction rewards.

When we had our features implemented, we focused on the modeling aspect of the problem. We had to both improve our learning algorithm as well as tune our model hyperparameters, such as greedyness and our γ value. Some of this adjusting was done by trial and error, but we also made sure to watch the gameplay to determine what were the frequent causes of monkey collisions. Observing patterns in the failure modes often gave us intuition for what was wrong and how to fix it.

This strategy led us to implementing retroactive rewards, as well as the jump protection for when the monkey was close to the bottom. Once we felt confident with our strategy, we added our QSmoothing function to allow us to remove some of our binning and in doing so model a larger state space. Doing this increased our accuracy, as we would be able to make more precise decisions.

If we had more time, we would improve the QSmoothing to include a term for how many time steps back we are smoothing. We would like to make the smoothing tighter at closer time steps and wider for further back time steps. This would enable us to even more finely produce the gradient of Q that helps to make better decisions.