SwingyMonkey Reinforcement Learning
Ryan Kerr, Tony Li, George Zhang
CS181 Practical 4 Writeup
https://github.com/gzhang01/cs181practicals/tree/master/practical4

1 Abstract

Our goal was to teach the monkey agent in the Swingy Monkey game to learn to play the game and avoid losing for as long as possible. Because we were working on an infinite horizon (the game will keep playing until the monkey loses), we ended up using Q-learning, having the monkey learn from the previous action and reward at each step and maximizing the Q-value. After playing around with the state space, the representation of our Q-values, and the parameters, we were able to successfully have our monkey play the game and not lose for a reasonable amount of time after around a few dozen epochs of learning.

2 Technical Approach

We started by implementing a Q-learning based approach. Figuring out the state space was difficult initially however. Our first idea was to create a state space that took into account the monkey's bottom position, the tree's top and bottom positions, and the distance from the monkey to the tree. However, we knew we couldn't use the pixel value directly, so we split up the spectrum into sections, or boxes. We initially used boxes of size 40 pixels. However, we quickly realized that such a large state space would be impossible to learn on quickly. The screen is 400 pixels high by 600 pixels wide. By using a 40 pixel box, we would have 10 boxes each for monkey's bottom and tree's top and bottom, and 15 boxes for the distance from monkey to tree. That alone is a state space with $15 \cdot 10^3$ options, meaning we would need on the order of that many updates before we produced any meaningful table. This doesn't even take into account the fact that each of these states have two possible actions, so the Q-table would be twice as big as this. Thus we learned early on that in order for our algorithm to learn quickly, we would have to keep the state space as small as possible.

To shrink the state space, we started by replacing tree top and tree bottom parameters with a single parameter that is the signed difference between the monkey bottom and the tree bottom. We decided that this would be a more useful metric, since we would likely want to be slightly above zero on this. We kept monkey bottom in the mix, as it would be a metric that told us whether we are in danger of falling off the top or bottom of the screen. We note that while the full range of monkey bottom to monkey top is 800 pixels, and at our box size of 40, we would have 20 boxes. This is much better than the 100 boxes we had before when saving both tree top and tree bottom. However, it is worse than the 10 boxes we'd have if we only tracked tree bottom. We justified this change by reasoning that it is very unlikely for us to utilize the full 20 boxes. In fact, most of our values for this parameter will fall near 0, and so we would only use a few boxes around the middle of the range. Thus we felt okay about making this move, and indeed, we got marginally better results.

The last parameters we played with for the state space were gravity and velocity. Gravity we ended up adding, since the monkey's rate of fall is significant for decision-making. This doubled our state space, since gravity is either -1 or -4, which we hashed to 0 and 1 respectively for our Q-matrix. Velocity was an interesting parameter. We thought that the velocity of the monkey should be relevant in making decisions, since if we're falling rapidly, we would probably want to jump (ideally we do not change y position very rapidly). However, we found through testing that adding in a velocity parameter resulted in very poor learning. We're not quite sure whether it is a poor parameter or whether it merely expanded the state space too much, but we attempted to incorporate it several times throughout the process but every time we did the results dropped dramatically. As a result, we ended up ignoring it. Thus the final parameters in our state space were monkey bottom, distance to tree, monkey bottom - tree bottom (which we'll call margin from now on), and gravity.

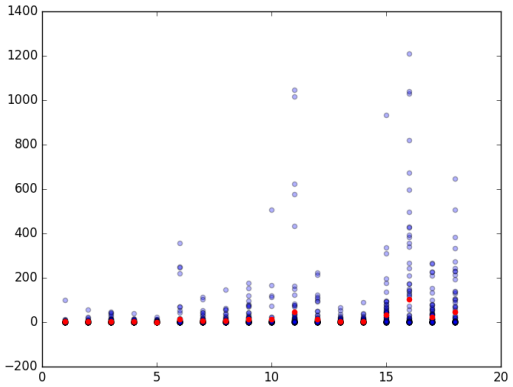
Given this state space, we did a quick calculation of how many boxes we had. With a box size of 40 pixels, we have 10 boxes for monkey bottom, 15 for tree distance, a worst case of 20 for the margin, and 2 for gravity. Thus there are about 6000 possible combinations. We decided we had to shrink this space more. We experimented with box sizes of 60 and 75, and we saw slight improvements. However, we reasoned that making the box size too big would make the margin boxes slightly useless. Since they are very clustered around zero, having big boxes makes it difficult to have an accurate jump through the tree. Thus we decided to split the boxes into a box for the first two parameters, and a marginBox for the margin. We tested marginBox values of 20, 25, and 30 and found that 25 worked best. Even though this gives us a worst case of 32 boxes, we found that the results were much better, again likely because the accessible states are clustered around zero. With this value, we continued to experiment with the box value and settled on a value of 150. This gives us 3 boxes for monkey bottom, 4 boxes for tree distance, a worst case 32 for margin, and 2 for gravity. This is a worst case 768 possible states, but we suspect that the useful states are much fewer in number.

At this point, we started playing with parameters. The α and γ parameters gave us some different results, but ultimately we settled on $\gamma = 0.5$. We found that a low and steadily decreasing α performed best, so we started it out at 0.2 and dropped it proportionately with 1 / $\lfloor$tick / 25$\rfloor$. Here, we defined tick as any time when we receive non-zero rewards. Given these parameters, we found that our algorithm learned relatively quickly and produced good results with higher probability.

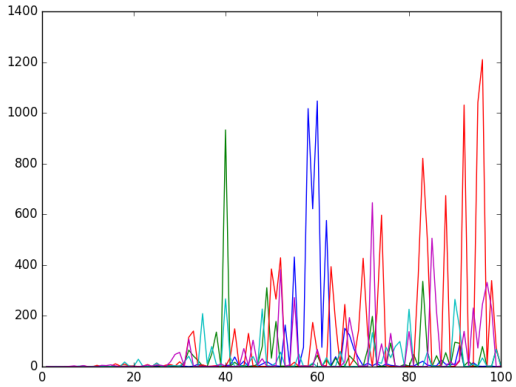
One last point we noticed as we were testing was that our algorithm would start performing well after a few dozen epochs, but after another few dozen, it would start performing worse, with our mistakes generally being jumping too high. We reasoned that since the default behavior we coded was for the monkey to fall, it learned that bottom of the screen was bad and middle was good. However, as it continued learning and occasionally jumped higher, the monkey would return to the middle due to gravity, and thus the Q-learner assigned high values to the upper boxes as well, leading the learning to jump more often than it should. We hoped that by implementing an ϵ-greedy approach, with high exploration initially and high exploitation later, we would explore the upper boxes early and assign them negative values and avoid this issue. Ultimately, we found that implementing ϵ-greedy, in addition with the decreased number of boxes due to our shrinking state space, we no longer had this problem anymore.

3 Results

We adjusted the parameters and our implementation of the state space and re-ran the program for 100 epochs each time until we achieved decent results. All of our data is located in the file data.txt. Unfortunately, since we have 100 numbers for each trial, we cannot provide all our data in this report. However, we have produced the following visualizations of our results:



(a) Scores on Each Epoch for 18 Different Runs



(b) Scores over Epoch for 5 Runs

In figure (a), we have plotted 18 different runs we made. We can see that compared to some of our later results, our earlier attempts performed relatively poorly. We plotted each score for a single run in a single column, and plotted the average score as a red dot. To save space, we will not explain all our runs. However, we can see that run 16 is relatively noteworthy. This is the first run when we implemented a decreasing α with our small state space. Run 18 was a rerun of these same parameters, and while it performed worse than run 16, it still performed better than all our other trials.

Overall information is good, but we also care about how long it took our algorithm to learn and do well. In figure (b), we plot scores over the 100 epochs for 5 different runs. We selected runs 11 (first time we separate marginBox and box and decrease state space for box; blue), 15 (first time decreasing α over time; green), 16 (starting α at 0.2 instead of 0.5; red), 17 (trying $\gamma = 0.2$; cyan), and 18 (repeat of run 16; magenta). We can see that each of these started producing noticable results on this scale around epoch 30. However, we started getting double digit results for some of the later runs as early as epochs 10-15. We can see that for the blue run especially, we have a few peaks and then it largely disappears, which indeed was the issue we had with the monkey going high later on in the run. Later trials where we decrease α, such as the red and magenta lines do not show this issue; in fact, the red line seems to perform better and better as the epochs go on. Ultimately, from this data, we decided to settle with a Q-learner that uses the parameters that the red trial used, namely a box value of 150, a marginBox value of 25, a α value starting at 0.2 and decreasing with passing ticks, a γ value of 0.5, and an ϵ-greedy approach that decreases with passing ticks.

4 Conclusion

Our task was to implement reinforcement learning to train our machines to play Swingy Monkey. We took a Q-learning approach, with the goal of using new state / reward and previous state / action combinations to assign Q-values to each state-action pair. We found early on that a large state space would take very long to learn, and so we needed to shrink the size of our state space as much as possible while maintaining good results. We accomplished this by using a large box size to discretize the position of the monkey and the distance to the tree into as few boxes as possible. However, we maintained a small marginBox size, as we believed precision in getting through the tree was important and few of the extreme boxes would be used, since the margin would be largely clustered around zero.

In terms of other parameters, we found that a decreasing α was useful. This makes sense, since we would prefer to update our Q-values quickly early on to promote faster learning, but after we've experienced a lot, we probably want to be less prone to updating based on fluke values. A low α allows for this, since we weight our original estimate more. We did not experiment as much as γ; however, we believe that since the game could theoretically go on forever (or at least a very long time, as evidenced by the scores of 1000+ we occasionally got), we should select a low γ value. We did not find significant differences between $\gamma = 0.2$ and $\gamma = 0.5$, and so went with a value of 0.5, which produced marginally better results.

Finally, we want to discuss the use of ϵ-greedy in our situation. We did find that prior to implementing ϵ-greedy, we tended to do worse later in the run. We've provided a hypothesis as to why this is true above. However, the solution to this is a little unclear, as the combination of implementing ϵ-greedy and a decreased state space seemed to have the biggest effect. Obviously exploring new states is a good thing. However, it is equally likely that the elimination of some states means that we no longer had some of those unexplored boxes. Indeed, since we spend most of our time in the middle to low regions of the screen anyway, the default behavior of dropping that we had before ϵ-greedy may help us learn what to do in that space more quickly, which might actually be more beneficial overall. Thus while ϵ-greedy is good for balancing exploration and exploitation, the net effect of it is unclear in our specific situation.