E-181 : Practical 4

Due on Friday, April 18, 2014

Praphruetpong Athiwaratkun

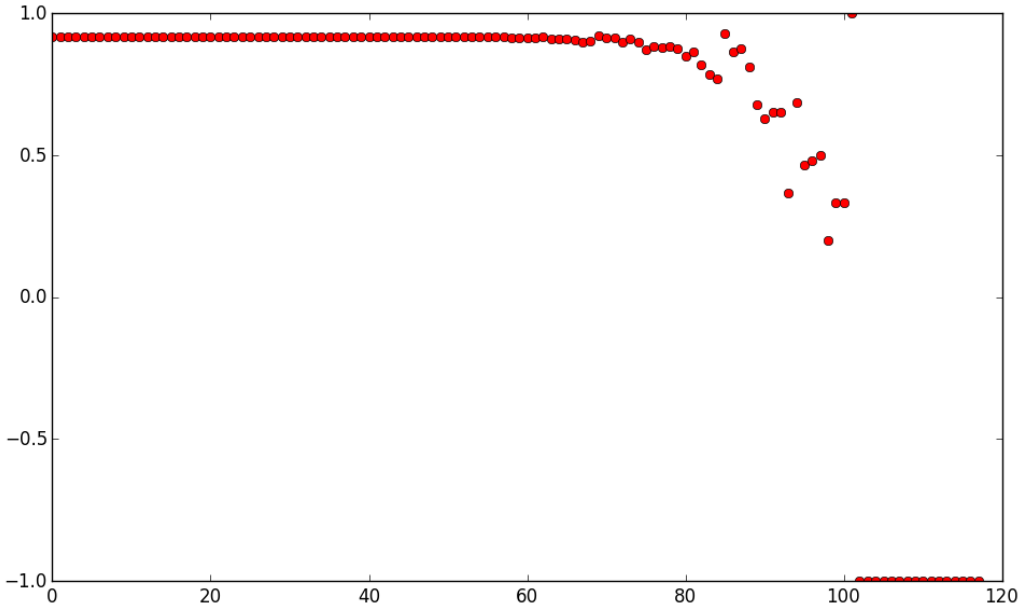
Warm Up

In this warm up, we aim to find the best policy for the lawn dart game. I choose to implement value iteration because of its simplicity for not needing to solve a linear system. Since the state space is small enough, the value iteration should converge quickly enough, which is the case as observed.

Results

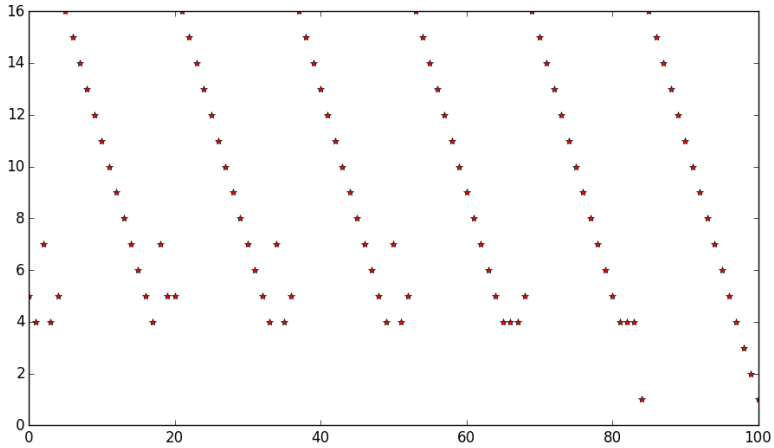
Below is the plot of the expected reward starting at each state $s \in \{0, \ldots, 117\}$. Note that for $s = 0, \ldots, \approx 80$, the expected reward is around 0.90. The expected rewards are about the same for each state in this range because they are relatively far enough from the target score of 101 such that there is no clear difference whether being at $s = 0$ or $s = 60$, for instance, can cause the optimal strategy to perform distinctly better or worse.
For a non terminal state $s \in \{0, 1, \ldots, 101\}$, the expected reward is the highest for $s = 85$. This is because $s = 85$ is the earliest state in which we can aim for the target score 101 directly by throwing the dart at 16. By being the earliest one, it has more room to adjust if the dart does not get to 16 and causes the expectation value to be high.

Figure 1: Converged Value vs. State



Warm Up continued on next page...

Figure 2: Optimal Policy (1 to 16) for Each Non-Terminal State



Swingy Monkey

In this practical, I implement a $Q-$ learning model. Most of the work entails adjusting the right model configurations such as the scheduling of how parameters might evolve or the state space representation. This writeup will explain the most successful approach observed.

Algorithm Details

Representation of Q Function

I implement Q-learning with several choices of the representation of Q function such as

1. discretizing the state space into blocks and represent Q as a matrix.
2. using neural network as a Q function.

The option 1 seems to work best as I notice that the change in Q function from adjacent states can be quite dramatic. The neural network tends to smooth out the function which makes it imprecise; that is, the stochastic gradient descent for state s directly affects states near s where in fact the optimal Q values of two nearby states can be drastically different. To get to the optimal Q with neural network might require a lot more iterations than the other approach.

State Space

The available information that can be used to as a state are:

1. The position of the top part of the tree.
2. The position of the bottom part of the tree. Since the gap between the top and the bottom of a tree is always fixed (200), this variable can be treated as redundant.
3. The position of the top part of the monkey.
4. The position of the bottom part of the monkey. Similarly, since the height of a monkey is fixed, this variable is redundant with 3.
5. The monkey's velocity.
6. The distance between the monkey and the tree. Note that if the monkey hits the tree at the side, the distance is 10.

Optimal State Space

I define a state to be an $n-$tuple, with each dimension involving the variables above. I scale each dimension of the tuple with additive and multiplicative factors. The addition factor is to make sure the variables are non zero as I choose to store the variables in a matrix. The multiplicative scale is to map a large number of blocks to a small number of blocks. Note that there is a tradeoff between the speed of back propagation of Q values and the precision. If we choose small blocks, the space is large and it is slower to propagate Q values for credit assignment as the algorithm only propagates back 1 state at each iteration. If we choose large blocks, we lose the precision and the ability to distinguish the edges/corners that are necessary for the monkey's survival. The Q values also tend to smooth out which could be undesirable. The optimal state space found is constructed by

- **scaled horizontal distance = (Distance from the tree + 150)/50**

  I notice that the behavior around the tree edges are particularly important for the monkey's performance. I shift it with +150 to make it non negative as the original distance can be as low as −115. In addition, since each horizontal distance changes by 25 per frame, the block dimension is chosen to be a multiple of 25. I tried $25, 50, 75$ and found that 50 seems optimal.

- **scaled vertical gap = (top tree position - top monkey position + 200)/25** The block dimension chosen is 25 because I notice that for the monkey to cross the vertical gap, it needs a finer details of the where it is vertically. Adjusting the block size of the bigger or smaller does not seem to improve the performance.

I define a state to be **s = (scaled distance, scaled vertical gap)**. I also omit **the monkey's velocity** because it makes the space larger and makes it harder for the Q values to propagate. I also omit **the explicit monkey's position** because if the monkey can navigate through the gaps of the trees which range in vertical position from $(0, 200)$ to $(200, 400)$, it should not be too close to the bottom or the top of the screen in any case.

Learning Rate Scheduling

I try several scheduling approaches for the learning rate α. The methods that seem to work are

- Constant $\alpha = 0.2$. This seems to result in fast learning. However, good results in early iteration are sometimes overruled, possibly due to the non-convergence of Q values since α does not decrease.
- $\alpha = e^{-1-(max\ scores/15)}$. I choose this rather than α directly depending on the number of iterations because if the monkey is still not performing well, we still want to update the Q values. The measure of how well the monkey has done is chosen to be the past maximum scores. The constant scaling is to ensure reasonable initial value and decaying rate.

Exploration vs. Exploitation

I schedule $\epsilon = 0.5 \cdot e^{-n/500}$ where n is the number of iterations. If a random number $m \in [0, 1)$ is less than ϵ, then the monkey will choose to explore by randomizing another $m \in [0, 1)$ and jump if m is less than a threshold, which is chosen to be 0.08.
In addition, if either of $N[s, 0]$ or $N[s, 1]$ are non zero but the Q values are not propagated there yet (both $Q[s, 0]$ and $Q[s, 1]$ are zero), then explore as well. If either of $N[s, 0]$ or $N[s, 1]$ are non zero and some $Q[s, a]$ are non zero, then exploit. If both $N[s, 0]$ and $N[s, 1]$ are zero, then explore. Note that the ϵ scheduling will overrule this decision (in favor of exploring), which causes the monkey to explore frequently during early iterations.

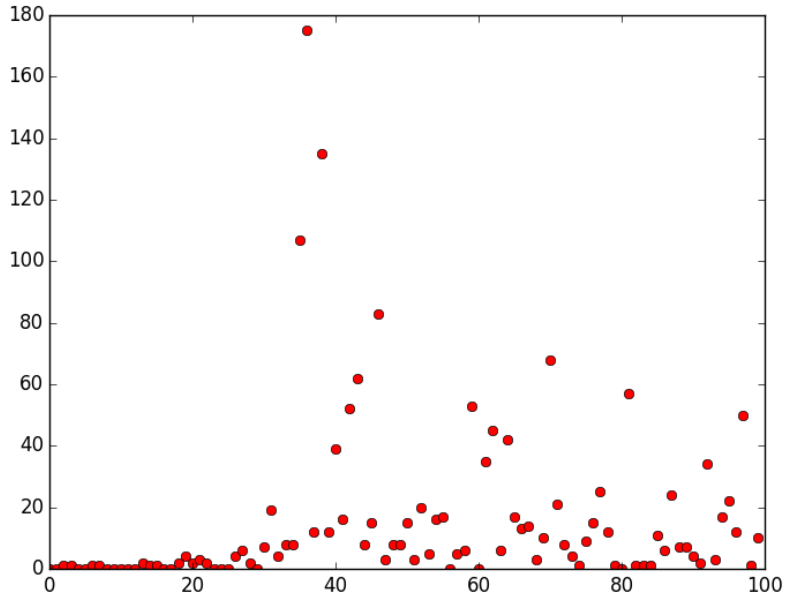
Other Factors

Discounting factor used is $\gamma = 0.8$.

Results

This section provides the plots of the scores for each episode. When α is constant,

Figure 3: Monkey's Scores Versus Episode for Constant α



If α decays, the Q matrix converges and the performance is, on average, consistently better over time. The results are still quite probabilistic in that different games can have quite different average results. This might be because the Q matrix is trapped in some local optimal. The following three games have exactly the same configuration parameters.

Figure 4: Monkey's Scores Versus Episode for Decreasing α : Game 1

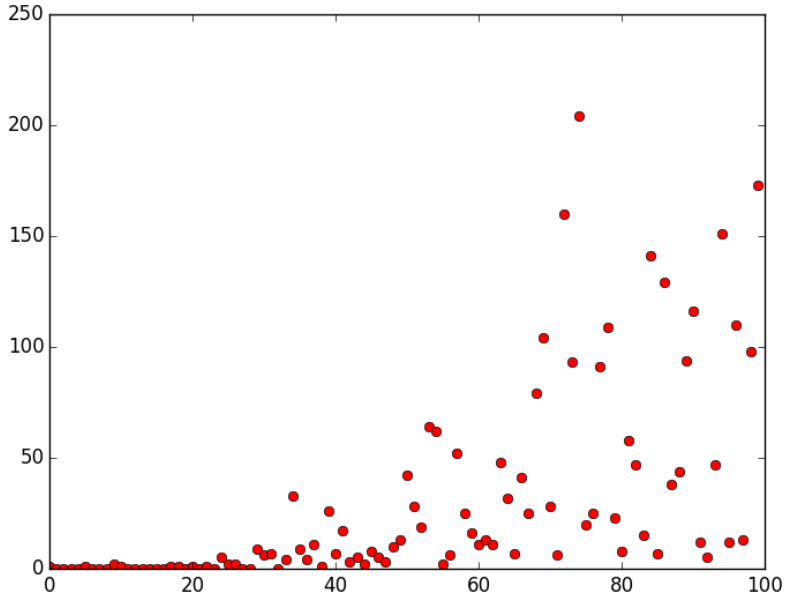


Figure 5: Monkey's Scores Versus Episode for Decreasing α : Game 2

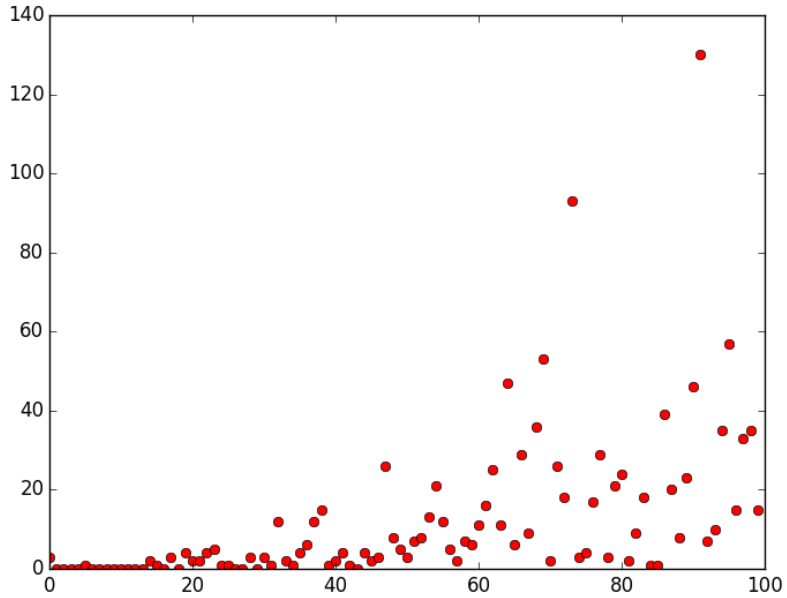
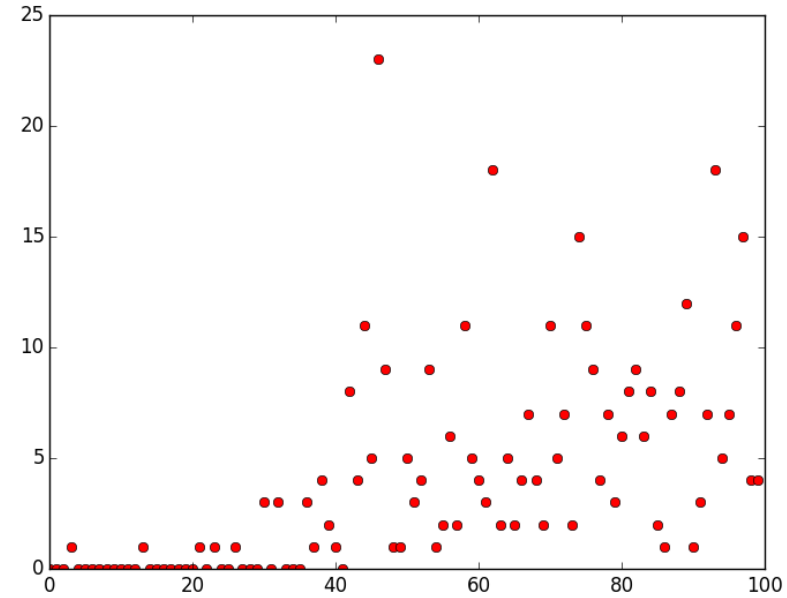


Figure 6: Monkey's Scores Versus Episode for Decreasing α : Game 3



I also observe that being in the region above the top of the tree should never allow the monkey to jump for optimal performance. This is not the case since the the propagation of Q values can be slow. By explicitly disallowing the monkey to jump if the top position of the monkey is greater than the top position of the tree indeed results in better performance. The best performance observed can be as high as more than 400 consecutive successes in an episode. This result happens once from running about 10 times.

Figure 7: Monkey's Scores Versus Episode for Decreasing α : Game 3