CS181 / CSCI E-181 Spring 2014 Practical 4

David Wihl
davidwihl@gmail.com

Zachary Hendlin
zgh@mit.edu

April 18, 2014

Warm-Up

The warm-up consisted of the lawn darts game using a grid of values. We used a finite horizon value iteration as the game would be certain of eventually ending even if the number of throws was indeterminate.

We chose to use Value Iteration over Policy Iteration because of the number of steps to execute the game were reasonable and would provide a contrast to the main problem which used Policy Iteration. The utility was defined by the Total Reward as in this case, there was only one way possible to have a positive reward ($score = 101$). There was no reason to use discounted rewards as there was no reward for intermediate scores or using less dart throws. We chose a γ of 0.9.

We set up the problem so the game could be played three ways: manually by a user, automatically via predetermined coded logic, and automatically via Reinforcement Learning.

If the $max(Q_k(s, a))$ yielded zero as in the early states, we explicitly chose to attempt a throw at the square with the largest score (in the premise of minimizing lawn throws even though was that not an explicit goal of the exercise).

The peak values occurred when it was possible to win the game with a low probability of losing (as shown in the red horizontal line in the plot - See Figure 1). This started occurring when the score = 85, as this was the lowest state where winning the game was possible (positive reward) and losing (negative reward) was not yet possible.

Reinforcement Learning was more complicated to implement than via pre-determined logic, but provided significantly more flexibility should the parameters of the game change.

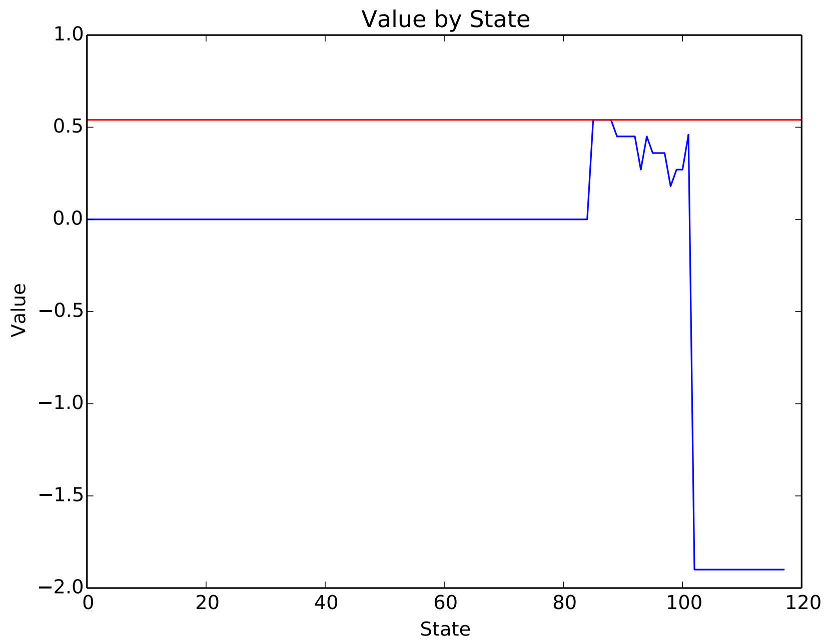


Figure 1: Finite Horizon Values vs States

Swingy Monkey

In building a reinforcement learning system to play *swingy money*, our aim was to write an *action callback* function that would, for given a state as an input, provide an action (either jump or don't jump) for that period. The actions should improve as we learn more about state space and resultant rewards, making it a perfect candidate for the application of reinforcement learning.

Reinforcement Learning was more complicated to implement than pre-determined logic, but was significantly more flexible should the parameters of the game change or evolve.

To learn the dynamics of the system, model-free reinforcement learning was selected as the method of choice. While we considered the possibilities of trying to explicitly model the system (e.g. consider velocity, change in positions to determine angle, etc.), that approach seemed too 'engineered' in comparison to having the reinforcement learning system learn the dynamics of the system.

Our state space is given by:

- Pixel of tree's bottom coordinate

- Pixel of tree's top coordinate
- Pixel distance from tree
- Velocity of the monkey
- Pixel of the bottom of the monkey
- Pixel of the top of the monkey ...

We note, however that since these pixel and velocity states are integers taking on a wide range of values:
treemin='bot': 11, 'top': 211, 'dist': -115
treemax='bot': 140, 'top': 340, 'dist': 310
monkeymin='vel': -47, 'bot': -44, 'top': 12
monkeymax='vel': 18, 'bot': 364, 'top': 420

To deal with this 'continuous' set of values, we decided to bucket values (we explored various bucket sizes, from 2 to 100), in order to be able to visit enough of the states and learn the dynamics of the system. We found that having too many buckets would cause the system to perform poorly for many more iterations, as it takes much longer to create a non-sparse set of Q-values. So for our six variables, we have:

$$2 * k^6$$

where k is the number of bins per variable. So for k=5, we have to visit 31,250 states to visit each state,action pair. For k=10, we have visit 2,000,000 states to get one estimate of each q value.

Q Learning

In Q learning, the agent aims to learn the set of rewards associated with each (state, action) pair, both for the immediate next step, as well as the discounted value of future rewards to taking a particular action.

The advantage of this approach is the flexibility it provides in being able to learn sets of optimal actions, however –as we found in this practical – it can take many iterations to explore enough states to perform well.

We define our Q function:

$$Q(s,a) = R(s,a) + \gamma(\sum_{s\prime} P(s\prime|s,a) \max_{a\prime \epsilon A} Q(s\prime,a\prime) \tag{1}$$

The update of the Q fuction at each time set for a (state, action) pair is:

$$Q(s,a)_new = Q(s,a) + \alpha(r + \gamma \max_{a\prime \epsilon A} Q(s\prime, a\prime) - Q(s,a)) \tag{2}$$

where:
γ = discount factor, between 0 and 1 (inclusive)
α = learning rate, between 0 and 1 (inclusive)

Approach

For our first time step, we don't know anything about the Q function value and so take a random action.

For all subsequent time steps, we compute the Q value as given above for the previous state's action and reward. We then determine, for the state the agent is, which action (jump or no jump) has a higher Q-value in the dictionary of Q-values that we have updated at each step.

Since the Q-values are not populated for state-action pairs we have not visited, we set all unvisited state-action Q-value pairs to zero at initialization.

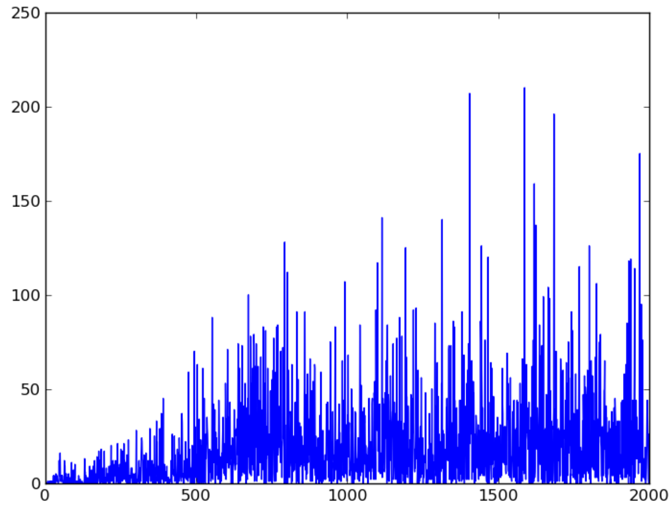
In order to caputure the fact that there is some stochastic element to the outcome (as we know that the game engine, for instance, randomizes state-reward mappings), we choose the opposite of the optimal action a very small fraction or the time (.01 percent of the time).

Results

With only minor parameter tuning, $\alpha = 0.05$,$\gamma = 0.9$, 5 bins, P(try random)=.001, and training on 2000 game epochs, we were able to achieve a maximum score of 193 and an average score of 16.986.

For runs 2000 to 4000, we were able to achieve a max score of 210, and an average score of 18.36

Runs 2000 to 4000 are shown below on the x-axis, with score on the y-axis.



As expected, in the intial epochs, the RL-based gameplay doesn't do very well as all state action pairs have a Q-value of zero. With more iterations, we see that the scores improve substantially.

Having a sufficiently large number of visits to each state and action is key to the Q-learning approach performing well.

References

[1] *Reinforcement Learning*, Sutton & Barto, 1998, ISBN-10: 0-262-19398-1