Practical 4:
Learning to play Swingy Monkey

Samuel Daulton, Taylor Killian, and Andrew Petschek (ML Marauders)

April 27, 2016

1 Technical Approach

In this practical we were tasked with creating a learning agent to achieve a high score in the game Swingy Monkey. The goal of Swingy Monkey is to pass (by dodging) as many tree trunks as possible without flying off the screen upwards or downwards, which results in a game over. The monkey has two actions: jump to a new vine (at a random height above) or swing on the current vine. The location of the gap in each tree trunk is random, as is the gravity of each game. to We were provided with code for the Swingy Monkey game and additional code for retrieving information from the game at each timestep. Specifically, we were provided with a Python dictionary containing the following:

```
{'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> }}
```

The reward for the previous timestep is given to be:

$$\text{Reward} = \begin{cases} 1, & \text{for passing a tree trunk} \\ -10, & \text{for falling off the bottom of the screen} \\ -10, & \text{for jumping off the top of the screen} \\ 0, & \text{otherwise} \end{cases}$$

In this section, we outline our methodology for setting up this problem to be approached using reinforcement learning. In particular, we specify the Markov decision process framework for the problem. In subsequent sections, we outline two distinct reinforcement learning methodologies used to learn policy capable of achieving high scores.

1.1 Markov Decision Process (MDP) Specification

To model Swingy Monkey as a MDP, we used the reward function and actions as defined in Section 1. The state space for our model is all possible tuples of the form:

```
(tree dist, tree top, tree bot, monkey vel,
      monkey top, monkey bot, gravity)
```

Transition Probabilities?

1.2 Learning Gravity

In the after the first timestep of each game, the agent learns what the value of gravity by calculating the vertical distance that the monkey moves (when the monkey swings on its current vine). Explicitly,

$$\text{gravity} = -|\text{monkeybot}_0 - \text{monkeybot}_1|$$

Note that, for purposes of the game, gravity is a constant velocity.

2 Approach 1: Batch Reinforcement Learning

Our first approach was to use batch reinforcement learning to allow an agent to learn from the actions and rewards of a batch of epochs. The methodology is as follows:

1. Run 1 batch of 100 epochs were the agent takes a random action at each timestep. The purpose of this is to create an initial dataset where each observation in the dataset is of the form: (state, action) $\longrightarrow$ reward.
2. Train a Random Forest Regressor using state and action as predictor variables and reward as the response variable. This trained Random Forest is our Q-function. For a given state and action, the Random Forest will return the expected value.

3. Using this Q-function, we obtain the best policy given the historical data as, for each state s:

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} Q(s, a) \tag{1}$$

4. Run 10 optimization (i.e. learning) batches each consisting of 50 epochs, in which the agent takes a random action instead of following the policy with of .1. The idea with the random actions is to explore 10% of the time. After each batch, append the dataset from the most recent batch to the aggregated dataset of all previous batches (including the initial random batch) and retrain a Random Forest Regressor using the entire dataset. Hence, we update our Q-function to reflect what the agent has learned. Using this Q-function, we again obtain the best policy given the historical data using the Equation (1).

5. Run 1 test batch of 100 epochs using the optimal policy, in which the agent takes no random action. Various metrics including mean, maximum, and minimum score over the 100 epochs are used to determine the effectiveness of the optimal learned policy.

Note that we chose to use 50 epochs per learning batch to obtain a satisfactory balance between performance and runtime.

3 Approach 2: Online Q-Learning

The second approach we took was using Online Q-learning, again using a Random Forest Regressor to infer the Q-function. The methodology is as follows:

1. Run 1 epoch where the agent takes random actions.

2. Train a Random Forest Regressor using the state-action-reward dataset from the first epoch to infer the Q-function.

3. Obtain the optimal policy using equation (1).

4. Run 100 learning epochs. In the first learning epoch, the agent takes a random action with a probability of .5 for each timestep. After every 10 epochs, the probability of an agent taking a random action is reduced by 10%. This is to ensure that the agent explores a lot at first and slowly transitions to exploiting more as time goes on, until the agent solely exploits. After each epoch, append the dataset from the most recent epoch to the aggregated dataset of all previous epochs (including the initial random epoch) and retrain a Random Forest Regressor using the entire dataset. Using this Q-function, we again obtain the best policy given the historical data using the Equation (1).

Note that the last 10 epochs are used as to test the policy, since the agent takes no random actions in those epochs. Various metrics including mean, maximum, and minimum score over the 100 epochs are used to determine the effectiveness of the optimal learned policy.

4 Analysis of Methods