Problem 3 (Reinforcement Learning, 20 pts)

In 2013, the mobile game Flappy Bird took the world by storm. You'll be developing a Q-learning agent to play a similar game, Swingy Monkey (See Figure 1a). In this game, you control a monkey that is trying to swing on vines and avoid tree trunks. You can either make him jump to a new vine, or have him swing down on the vine he's currently holding. You get points for successfully passing tree trunks without hitting them, falling off the bottom of the screen, or jumping off the top. There are some sources of randomness: the monkey's jumps are sometimes higher than others, the gaps in the trees vary vertically, the gravity varies from game to game, and the distances between the trees are different. You can play the game directly by pushing a key on the keyboard to make the monkey jump. However, your objective is to build an agent that learns to play on its own.

You will need to install the `pygame` module (http://www.pygame.org/wiki/GettingStarted).

Task

Your task is to use Q-learning to find a policy for the monkey that can navigate the trees. The implementation of the game itself is in file `SwingyMonkey.py`, along with a few files in the `res/` directory. A file called `stub.py` is the starter code for setting up your learner that interacts with the game. This is the only file you need to modify (but to speed up testing, you can comment out the animation rendering code in `SwingyMonkey.py`). You can watch a YouTube video of the staff Q-Learner playing the game at http://youtu.be/l4QjPr1uCac. It figures out a reasonable policy in a few dozen iterations.

You'll be responsible for implementing the Python function `action_callback`. The action callback will take in a dictionary that describes the current state of the game and return an action for the next time step. This will be a binary action, where 0 means to swing downward and 1 means to jump up. The dictionary you get for the state looks like this:

```
{ '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> }}
```

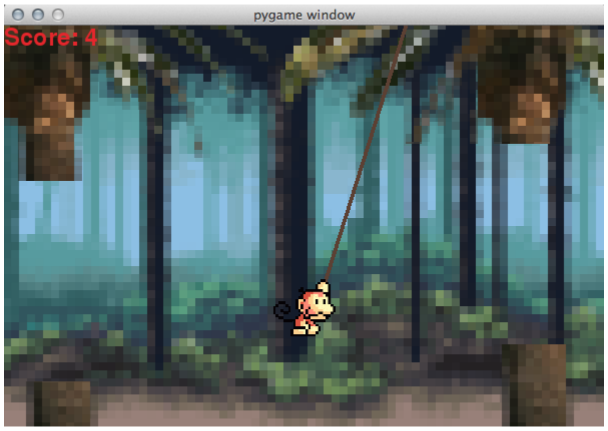
All of the units here (except score) will be in screen pixels. Figure 1b shows these graphically.

Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

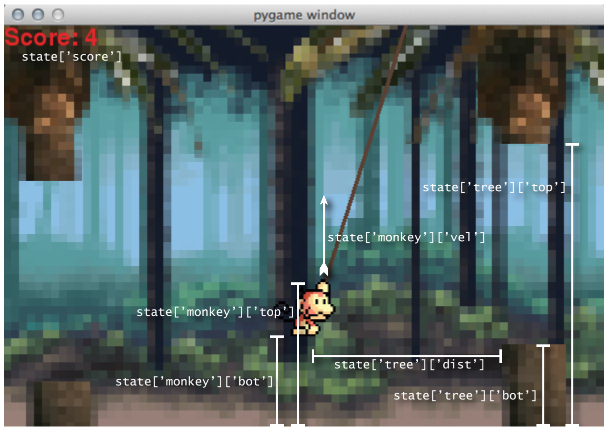
Requirements

Code: First, you should implement Q-learning with an ϵ-greedy policy yourself. You can increase the performance by trying out different parameters for the learning rate α, discount rate γ, and exploraton rate ϵ. Do not use outside RL code for this assignment. Second, you should use a method of your choice to further improve the performance. This could be inferring gravity at each epoch (the gravity varies from game to game), updating the reward function, trying decaying epsilon greedy functions, changing the features in the state space, and more. One of our staff solutions got scores over 800 before the 100th epoch, but you are only expected to reach scores over 50 before the 100th epoch. **Make sure to turn in your code!**

Evaluation: In 1-2 paragraphs, explain how your agent performed and what decisions you made and why. Make sure to provide evidence where necessary to explain your decisions. You must include in your write up at least one plot or table that details the performances of parameters tried (i.e. plots of score vs. epoch number for different parameters).



(a) SwingyMonkey Screenshot



(b) SwingyMonkey State

(a) Screenshot of the Swingy Monkey game. (b) Interpretations of various pieces of the state dictionary.

Solution

Let α be the learning rate, γ be the discount factor, and ϵ is the probability that the agent takes a random action. Note that the following results are of one particular iteration, since the probabilistic nature of Q-Learning means that scores may vary slightly across iterations.

Parameters	Top Score over 100 Epochs	% of 100 Epochs with Scores over 50
$\alpha = 0.5$, $\gamma = 0.5$, $\epsilon = 0.1$	6	0%
$\alpha = 0.2$, $\gamma = 0.5$, $\epsilon = 0.1$	6	0%
$\alpha = 0.5$, $\gamma = 0.8$, $\epsilon = 0.1$	15	0%
$\alpha = 0.2$, $\gamma = 0.8$, $\epsilon = 0.1$	21	0%
$\alpha = 0.2$, $\gamma = 0.8$, $\epsilon = 0.2$	7	0%

I initially started with a baseline of $\alpha = 0.5$, $\gamma = 0.5$, and $\epsilon = 0.1$. As shown in the table, the results of the baseline model were poor. I suspected that reducing the learning rate to $\alpha = 0.2$ would improve the performance, as too high of a learning rate means that the Q-values are updated too quickly. However, reducing the learning rate by itself did not improve the model's performance significantly. Then, I increased the discount factor to $\gamma = 0.8$ so that the agent cares more about achieving a high score (since a perfect agent could play the game infinitely long). Given the infinite time frame of the game (infinite time frame in one epoch since the game could be played for forever if the agent never makes a mistake) the increase of the discount factor was the correct modeling choice as it significantly improved the scores of the agent. With this improved model, I reduced the learning rate to $\alpha = 0.2$ again and found that this time, with $\gamma = 0.8$, reducing the learning rate also significantly improved my agent's performance. Finally, I raised the random action probability to $\epsilon = 0.2$ which significantly dropped the agent's score. Therefore, I chose the parameters of $\alpha = 0.2$, $\gamma = 0.8$, and $\epsilon = 0.1$ as the optimal model.

However, my agent still has not achieved a score of 50 in any epochs, so I implemented epsilon-decay by raising ϵ to a power c (i.e. ϵ^c) where c is dependent on the number of epochs so that way ϵ decays over higher epochs. Let λ be some constant and e be the number of epochs for a current iteration, such that the probability of a random action on some epoch e is given by $\epsilon^{1+\lambda e}$. The below table details the results of one particular iteration for different values of λ, where $\alpha = 0.2$, $\gamma = 0.8$, and $\epsilon = 0.1$.

Value of λ	Top Score over 100 Epochs	% of 100 Epochs with Scores over 50
$\lambda = 0.1$	297	2%
$\lambda = 0.2$	1303	4%
$\lambda = 0.25$	982	16%
$\lambda = 0.3$	565	4%
$\lambda = 0.4$	113	1%

We found that the optimal value of λ was $\lambda = 0.25$ which optimally trades off between allowing the agent to learn by taking random actions, but also ensuring that in later epochs the agent is not taking random actions when it has already learned enough to score well on the game. This makes intuitive sense, since setting a higher value λ causes ϵ to decay faster (approach near zero is fewer epochs). For values of λ less than $\lambda = 0.25$, ϵ does not decay quickly enough, so even at later epochs the agent is still taking random actions even though it has updated (learned) the Q-values in many of the earlier epochs, which causes it to lose the game earlier than it should (before the agent can achieve a score of 50). This causes the agent to be inconsistent over the epochs as demonstrated by the low percentage of epochs with scores over 50 for $\lambda = 0.1$ and $\lambda = 0.2$. Similarly, for values of λ greater than $\lambda = 0.25$, ϵ decays too quickly, so in the early epochs the agent is taking too few random actions and does not learn enough to score well on the game, as evidence by the lower scores and lower percentage of epochs with scores over 50 for $\lambda = 0.3$ and $\lambda = 0.4$. $\lambda = 0.25$ appears to be the optimal rate for ϵ to decay at as evidenced by the good results in the table, where the agent achieved a high score of 982 and scored over 50 on 16% of the epochs.