CS 181: Machine Learning — Practical 4

Testing, Testing, 1, 2, 4

Neil Chainani, Avery Faller, Ed Weng

Friday, April 29, 2016

GitHub URL: https://github.com/averyfaller/CS181Practical/tree/master/4/practical4-code

Given a fully-observable simple game, can we use Q-learning to have an agent train itself to play using a model-free approach? In this paper, we examine the difficult problem of implementing Q-learning on a "continuous" state space.

1 Technical Approach

1.1 Introduction

For this assignment, we were given code that runs a simple game loop where a monkey has to navigate horizontally through a forest by either jumping or swinging on a vine at each state. While the space is in fact discrete, it is essentially continuous, for all intents and purposes, given its sheer size. We can see this by multiplying the number of values for just a few of the dimensions in our space: (monkey top) x (tree top) x (tree distance) x (gravity) x (velocity) x (action) = 400 * 400 * 600 * 2 * 100 * 2 = 38,400,000,000. Running Q-Learning for such a large state space would take an enormous amount of memory and would require significant training time. We quickly deduced that we needed a way to reduce the number of states.

We attempted a number of techniques to reduce the number of states that would be necessary for the agent to play the game effectively. Some of the methods that we experimented with were binning states together and using regression techniques, including Random Forests and Neural Networks, to estimate Q-values.

We discovered that Random Forests work relatively well, while Neural Networks are very finicky and run slower due to their longer training times. However, in the end, the simplest approach using a dictionary, binning states, and applying Q-learning worked the best.

1.2 Methods

To begin, we built a simple Q-learner that would store the q-scores in a dictionary by transforming states into underscore-separated strings. It quickly became obvious when running this that the agent wasn't getting better in any reasonable amount of iterations. As mentioned earlier, the state space was simply too large for Q-learning to be effective.

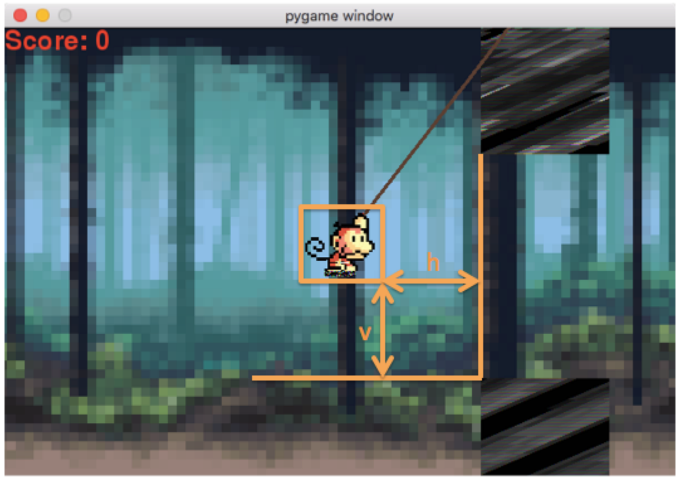

Figure 1: Reduced state with two position variables.

We began to optimize this implementation. First we decided that monkey top and bottom contained duplicate information and both were not needed. Additionally, the tree top and bottom similarly contained duplicate information. Since the height of the monkey and the gap in the tree was not changing over time, only the tops or bottoms for the monkey and tree were needed but not both. While removing these variables did not reduce the state space, it did reduce the size of our state strings. Once we eliminated the redundant information, we were left with seven variables: tree bottom, monkey bottom, tree distance, gravity, velocity, and action.

Next, we learned that a better metric for tracking the position of the monkey might be to examine the relative distance between the bottom of the monkey and the bottom of the tree. After all, what is important is not where the tree is, but where the monkey is relative to that tree. We reduced the monkey position to just two dimensions, v and h **(Figure 1)**.

We then attempted binning as a method of reducing the number of states, using integer division to bin values that were close together. We used different binning values for the distances and the velocities as the two tended to operate on different scales and it was important to keep velocity granular.

This binning greatly improved performance, and with some tuning of parameters, we were able to build a functioning game player.

Finally, we realized that we could remove velocity from the state string, further reducing the number of states that the agent would have to learn. While this removal would, in turn, theoretically worsen performance, given that the agent would have less information to work with, it would also reduce the number of states that the agent would have to learn, so the trade-off, for us, was worth it.

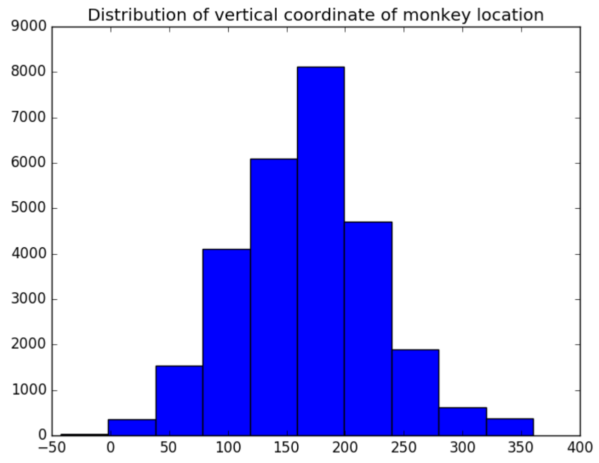

Figure 2: Reduced state with two position variables.

2 Results

Vertical bins stepsize	Learning rate	Discount factor	Max score
Dynamic bins	0.9	0.9	65
1	0.9	0.6	10
25	0.9	0.6	484
10	0.9	0.6	100
50	0.9	0.6	266
25	0.75	0.6	36
50	0.9	0.9	1347
25	0.9	0.9	2866

2.1 Factoring in Gravity

Early on, we attempted to factor gravity into our state representation. In order to do so, we forced the monkey to swing for the first two ticks and then looked at the change in velocity between the first two states. By doing so, we were able to detect that there were only two different types of gravity, one that caused the monkey's velocity to change by 1 and another that caused the monkey's velocity to change by 4.

Recognizing that the different gravities would dramatically affect the monkey's action set, we knew that we would need to have two sets of Q-matrices to accommodate the differences. In effect, this doubled our state space, but quickly proved to be extremely valuable in improving results.

2.2 Epsilon Greedy

One thought we had was that we should incorporate epsilon-greedy into our algorithm such that the monkey would continue to explore actions. However, we eventually realized that given there are only two states, both with a reward value of 0 if the q-value had not been computed previously for that state, then it would favor the previously unseen state over a state that had generated a negative reward. In this way, the dictionary-binning technique was already exploring a large amount of the state space and Epsilon greedy was therefore unnecessary.

2.3 Bin Sizing

We experimented heavily with different bin sizes. First we tried binning with a uniform step size and experimented with both small bin sizes and large bin sizes.

Then we tried binning with a non-uniform step size. The idea is that perhaps the algorithm needs to be able to make decisions with finer granularity at certain positions. We ended up plotting a histogram of where vertically the monkey was located at each time step, over the course of 100 epochs **(Figure 2)**. We saw that there was a normal distribution, with the highest density between 100px and 250px, so we decided to increase the granularity in this range, and decrease it along the fringes of the window. However, these dynamic bins actually resulted in a poorer score, so we decided to continue with equally spaced bins. Our sweet spot was a bin size of 25.

2.4 Neural Networks and Random Forest

We were able to achieve pretty decent results with Random Forests. Using the bins we had set up for our simple dictionary model, our agent consistently achieved cumulative scores across 100 epochs in the 70s. Removing the bins caused Random Forests' performance to drop precipitously to the low 20s.

However, this wasn't even close to the scores we were achieving with simple dictionary-binning. Additionally, we had to modify several parameters in order to get the agent to work with Random Forests. Firstly, the Epsilon-greedy technique that we had not found necessary for dictionary-binning was needed to encourage the agent to explore different actions. Whereas in dictionary-binning if the state-action combination was unexplored, it would favor that state, with any regressor, like Random Forests, the regressor would attempt to predict what the q-score would be for the unexplored state, and would likely return a q-score that was very close to the q-score of the same state but with the other action, making exploring unlikely.

We found Neural Networks to be very finicky. Attempting to tune them with the correct number of hidden layers and nodes was very difficult and it was hard to tell if the agent was improving or if improvements were simply due to natural variation in rounds of game-play. Eventually we settled on two hidden layers of 10 nodes, both using Sigmoid regression. Without bins the Neural Networks did terribly, achieving a score below 10, rarely passing more than 1 tree in an epoch. With bins, was a different story with the agent getting cumulative scores in the 20s.

Overall, the agent did better with the simple dictionary-binning than with Neural Networks or Random Forests. Given more time to tune the parameters of these methods, we could probably achieve higher scores, although whether they would be able to rival those of the simple dictionary-binning remains to be seen. However, Neural Networks considerably slowed down the speed of game-play and the advantage they provide is not obvious at this point in time. Random Forests, don't noticeably slow down the speed of the game, but they do add complication without increasing the cumulative score.

3 Discussion

Once we simplified our model to just contain the binned vertical and horizontal positions of the monkey relative to the tree, all we really needed to do was to tune our parameters. Our final state space, with 32 vertical bins, 8 horizontal bins, 2 gravity states, and 2 actions, was much smaller than what we started with. This reduction in state space (from 38 billion to one thousand) allowed our agent to find an effective policy in less than 50 epochs. We noticed that all of our attempts to run positions through random forests and neural nets were for naught, and what worked well was a simple dictionary.

One issue we encountered was the inconsistency of our results. We could run the same model twice in a row, and get a best score of more than 1,000 on 100 epochs, and get below 30 on the next. On poor runs like these, we noticed that for the first 50 epochs, the monkey would get stuck in the same loop every time: it would jump too high, miss the first tree, and repeat from the start. It seemed that it wasn't able to learn much from those runs, and consequently wouldn't do well. But one successful run would have an exponentially compounding effect in future epochs. The first few epochs were crucial to how well the agent would perform later on. To mitigate this variance, we could determine the optimal policy early on in the process, and keep it static into the rest of the epochs.

In terms of improving Random Forests, it might be interesting to have a combination approach that uses dictionary binning for states that have already been seen and a Random Forest to predict the q-score of states that it has yet to visit. This might fix some of the issues that the Random Forest Agent was having.

4 Code

```python
# Imports
import numpy as np
import numpy.random as npr
import matplotlib.pyplot as plt
import operator

from SwingyMonkey import SwingyMonkey
from scipy import stats

# get histograms - NC
# try different bins - EW
# talk about epsilon greedy approach - EW
# talk about neural nets / random forest - NC / AF
# talk about dictionary - EW

class Learner(object):
    '''
    This agent jumps randomly.
    '''
```

```
def __init__(self):
    self.last_state = None
    self.last_action = None
    self.last_reward = 0
    self.gravity = 0

    # For histogram plots so we can create better buckets
    self.vertical_hist = []
    self.horizontal_hist = []

    # This is the learning rate applied to determine to what extent
    # the new information will override the previous information
    self.learning_rate = .9

    # This is the discount factor which we will use to determine the importance of
    # future rewards
    self.discount_factor = .6
    self.q_scores = {}

    # self.vertical_bins = [-400, -200, -150, -100, -50, -25, 0, 25, 35, 45, 55, 65,
        75, 85, 95, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 225, 250,
        300, 400]
    self.vertical_bins = np.arange(-400, 400, 25)
    self.horizontal_bins = np.arange(-200, 600, 100)

def reset(self):
    self.last_state = None
    self.last_action = None
    self.last_reward = 0
    self.gravity = 0

def set_gravity(self, prev_state, next_state):
    self.gravity = prev_state['monkey']['vel'] - next_state['monkey']['vel']

def should_set_gravity(self):
    return self.gravity == 0

def state_action_to_array(self, state, action):
    vertical_dist = state['monkey']['bot'] - state['tree']['bot']
    horizontal_dist = state['tree']['dist']

    self.vertical_hist.append(vertical_dist)
    self.horizontal_hist.append(horizontal_dist)

    vertical_bin = np.digitize(vertical_dist, self.vertical_bins)
    horizontal_bin = np.digitize(horizontal_dist, self.horizontal_bins)

    arr = np.array([
          vertical_bin,
          horizontal_bin,
          self.gravity,
          action])

    return arr
```

```python
def get_key_from_state_action(self, state, action):
    arr = self.state_action_to_array(state, action)
    return "_".join(map(str, arr))


def get_Q_score(self, state, action):
    key = self.get_key_from_state_action(state, action)

    if self.q_scores.has_key(key):
        return self.q_scores[key]
    else:
        return 0

def set_Q_score(self, state, action, q):
    key = self.get_key_from_state_action(state, action)
    self.q_scores[key] = q

def action_callback(self, state):
    '''
    Implement this function to learn things and take actions.
    Return 0 if you don't want to jump and 1 if you do.
    '''

    if state is None or self.last_action is None:
        self.last_state = state
        new_action = 0
    else:
        if self.should_set_gravity():
            new_action = 0
            if self.last_action == 0 and new_action == 0:
                self.set_gravity(self.last_state, state)

        prev_Q = self.get_Q_score(self.last_state, self.last_action)
        swing_Q = prev_Q + self.learning_rate * (self.last_reward +
            self.discount_factor * self.get_Q_score(state, 0) - prev_Q)
        jump_Q = prev_Q + self.learning_rate * (self.last_reward +
            self.discount_factor * self.get_Q_score(state, 1) - prev_Q)

        # set new_action to 0 if swing_Q > jump_Q, save best_Q
        new_action, best_Q = max(enumerate([swing_Q, jump_Q]),
            key=operator.itemgetter(1))

        # Update the Q score for the last state and action
        self.set_Q_score(self.last_state, self.last_action, best_Q)

    print state

    # Update last state and last action for next iteration
    self.last_state = state
    self.last_action = new_action

    return new_action
    def reward_callback(self, reward):
        '''This gets called so you can see what reward you get.'''
        self.last_reward = reward

def run_games(learner, hist, iters = 1000, t_len = 100, r_iters = 10):
    '''
    Driver function to simulate learning by having the agent play a sequence of games.
    '''
    print "Running %d games" % iters

    for ii in range(iters):
        # Make a new monkey object.
        if ii < r_iters:
            learner.train = True
            print "Running a training epoch"

        swing = SwingyMonkey(sound=False,             # Don't play sounds.
                             text="Epoch %d, Max Score %d" % (ii, max(hist)) , # Display
                                 the epoch on screen.
                             tick_length = t_len,       # Make game ticks super fast.
                             action_callback=learner.action_callback,
                             reward_callback=learner.reward_callback)

        # Loop until you hit something.
        while swing.game_loop():
            pass

        # Save score history.
        hist.append(swing.score)

        # Reset the state of the learner.
        learner.reset()

    return


if __name__ == '__main__':

    # Select agent.
    agent = Learner()

    # Empty list to save history.
    hist = [0]

    # Run games.
    run_games(agent, hist, 100, 1, 1)

    #print "Num states: %d" % len(agent.Q)

    # Save history.
    np.save('hist',np.array(hist))

    # Print max score
    print "Max score was: %f" % max(hist)
```

```python
    print "Total Score was %d" % np.sum(hist)
```