CS181, Practical № 4

Team randomJungles: Nicole Lee, Harry McNamara, Luke Mueller 04/29/2016

Objective

Use reinforcement learning to achieve reasonable scores for the game "SwingyMonkey" while minimizing the time required produce such scores.

Model and Geometry

Strictly speaking, our system is a partially observable Markov decision process (POMDP). We have access to most state observables, given as a dictionary containing the monkey velocity, the monkey position (top and bottom; the monkey x position is fixed at $x_{monkey} = 300$) and the position of the nearest tree gap (top, bottom, and x distance). It is a POMDP in that the state also involves a random gravity g (between 1 and 4); however, we can infer the gravity value directly by waiting on the first state, and reading the velocity on the second state (and we can thus treat the problem as a true MDP). Transitions are deterministic, but are not given explicitly; thus we perform reinforcement learning to infer $Q(s, a)$ directly.

The complete state space is formally discrete and finite, but is effectively so large as to be continuous. Including 4 g values, the total number of entirely enumerated states is:

$$\begin{aligned} N(\text{states}) &= N(\text{g}) \times N(\text{monkey top}) \times N(\text{monkey bottom})(\text{velocity}) \\ &\quad \times N(\text{tree dist}) \times N(\text{tree top}) \times N(\text{tree bot}) \\ &\approx 4 \times 400 \times 400 \times 20 \times 200 \times 200 \times 300 \\ &\approx 10^{14} \end{aligned}$$

One thing we noticed immediately is that we can reduce the dimensionality of the state space by ignoring redundant information. For example, the monkey height doesn't change: we don't need to separately track the top and bottom of the monkey. Similarly, the distance between trees is fixed, so the top and bottom height of the next tree gap are redundant

We decided to reduce our state space to 4 variables: gravity g, velocity v, and the distance vector between the bottom of the monkey and the bottom of the tree gap (r_x, r_y) (as the tree gap height is conserved). Still, this leaves us a state space of:

$$N(\text{states}) \approx 4 \times 20 \times 300 \times 200 \approx 4.8 \times 10^6$$

Which is still an intimidating number to train. For our implementations we experiment with coarse-graining this space to make it more efficiently trainable. Eventually, we implement a neural network to approximate $\hat{Q}(s, a)$.

Implementation of Q-Learner with Q Table

Given the discrete nature of "SwingyMonkey", conventional Q-Learning seemed a reasonable first approach to this problem. By combining our intuition about the game's mechanics and Q-learning algorithms taken from class, we iteratively constructed a multilevel nested dictionary to inform the monkey's future actions (jump or fall).

In particular, at each *current* state s' (input by the distribution code as an argument the 'action callback()' function) we observed certain feature values (i.e. Monkey's horizontal and

vertical distance to next tree, velocity, etc.), and extracted Q values for each action associated with those feature values - calling this vector $Q^{old}_{s',a'}$. Using this vector, we selected the best action as new action = $\arg\max_{\alpha'}(Q^{old}_{s',a'})$, and observed the reward r. Our learner, having remembered the prior state s (i.e. the state that transitioned to s'), then updates the Q value corresponding to the last state visited and last action a using the standard Q-Learning algorithm:

$$Q^{new}_{s,a} \leftarrow (1-\alpha)Q^{old}_{s,a} + \alpha\left[r + \gamma \max_{\alpha'} Q^{old}_{s',a'}\right] \quad (1)$$

where $0 \leq \alpha \leq 1, 0 \leq \gamma \leq 1$. As our individual game epochs were short, we took $\gamma = 1$. Our implementation then updates the last state, last action variables using the new state s', and the new action (which it also returns). Iterating this algorithm eventually fills in the Q-dictionary. Note that in order to balance exploration vs. exploitation, we also introduce an 'epsilon-greedy' randomization step of our action choice (we can reduce epsilon with each epoch as we get more confident in our Q-table). The python file 'MonkeyLearner.py' contains this implementation.

Discretizing Variables

Technically speaking, we could have used the the full suite of variables at our disposal, including the pixel grid for both monkey and tree position, to construct the state space. However, doing so was beyond the computational abilities of our machines. Instead, we selected a subset of these variables to construct the state space based on our intuitions about the game's mechanics. For example, rather than using separate dimensions for trees and the monkey, we calculated the difference between the bottom of the monkey and the top of the lower tree. As a result, we were able to capture information about the monkey and $both$ the lower and upper trees (in terms of vertical direction) under a single dimension. This is because the gap between lower and upper trees is constant throughout the game. Thus, rewards based on this distance metric corresponded to hitting either tree.

In addition, we included variables for the monkey's horizontal distance to the next tree, the monkey's velocity, and the game gravity. The game gravity was estimated during the second state as the change in velocity from the first state.

At this point the state space was still quite large. We were able to fill the Q table without any further variable tuning, but required thousands of epochs before the monkey started showing "decent" scores. Following our objective, we explored options to further descretize our state space. Notably, we played with different bin sizes for the location-based variables and the velocity.

Results

The results from our Q-Table approximation show that algorithm is simply a trade-off between bin size and time. If we allowed the algorithm time to fill in the entire state space (without any binning/discretization) the scores in each epoch would increase without bound. However, if we confine ourselves to the problem of optimizing bin sizes, we see some interesting patterns. For example, dividing the horizontal and vertical distances between the monkey and the next tree by 50 gave us the best scores over 5000 epochs. This division value also provided fast learning - nearly on par with a division value of 100 in the first 1000 epochs. By contrast, setting the division value to 10 did not produce meaningful scores until nearly 4000 epochs, and setting the division value to 1 (no binning/discretization) did not produce any meaningful scores over 5000 epochs.

We also see that some of the discretizations reach their highest scores in early games rather than their latest ones. For example, figures 1 and 2 show a clear and substantial drop for the division = 50 line around 3000 epochs. It could be that we need additional hyper-parameter tuning (i.e. $\alpha, \gamma, epsilon_{start}$) to make sure the algorithm discourages/encourages exploration

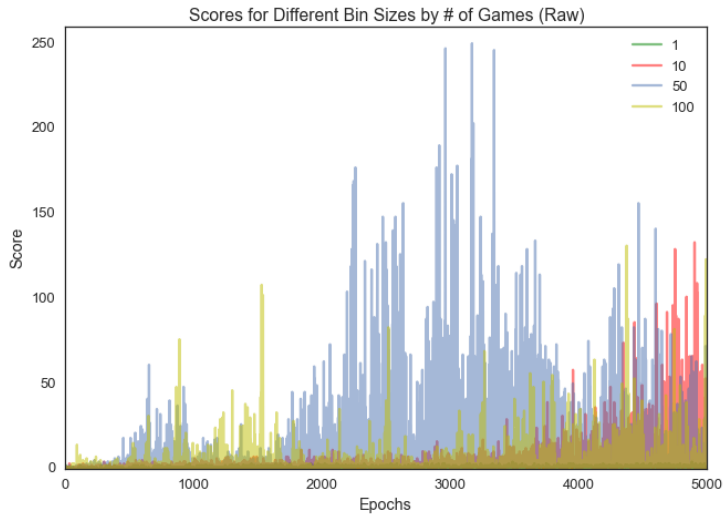


Figure 1: Examples scores for runs of 5000 epochs of the table Q-approximator. Lines represent different divisions of the state space with regard to horizontal and vertical distance between monkey and tree. Note: $\alpha = 0.7$, $\gamma = 1$, $\epsilon_{start} = 0.2$.

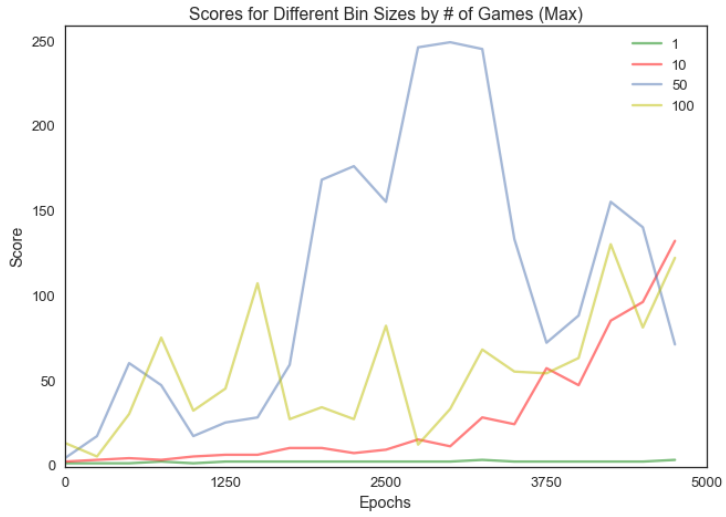


Figure 2: Examples scores for runs of 5000 epochs of the table Q-approximator, taking the max of every 250 epochs.

after a certain point. Though we do not see this pattern appear yet for the division = 1 or division = 10 lines, presumably, the pattern would be the same.

Q-Learner with Feature Approximation

Another approach to handling the effectively continuous state space of the game is introduce a model (linear or otherwise) to approximate Q values, and to use gradient descent to update the model parameters. This means we don't have to rely on our agent landing adjacent to the few states in which rewards were directly incurred to update values. Formally, we introduce a model:

$$\hat{Q}_{\{\theta\}}(s, a)$$

Which we train online as our RL agent explores the state space.

Now, in the action callback function, we determine which action to take (assuming we don't take a randomized epsilon greedy step) by finding $\arg\max_{a'}(\hat{Q}_{\{\theta\}}(s', a'))$

We can construct a target for training out model as

$$target = r_t + \max_{a'}(\hat{Q}_{\{\theta\}}(s', a'))$$

And we can update our parameters as:

$$\theta_i \leftarrow \theta_i + \alpha \left(target - \hat{Q}_{\{\theta\}}(s, a) \right) \frac{\partial \hat{Q}_{\{\theta\}}(s, a))}{\partial \theta_i}$$

Neural Network implementation

We choose to implement the function approximator as a neural network (contained in the python file 'NetLearner3.py'), as it is intuitively apparent that there are nonlinearities in our chosen features (i.e. a given velocity has different predictive value depending on the height of the monkey, and so on). Computing gradients for neural networks analytically is not feasible - we tried implementing the autograd package to take derivatives of neural networks with respect to parameters, but the neural network methods threw errors with the autograd 'node' classes which we could not resolve expediently. Instead, we sought neural network packages which could be trained online (e.g. with SGD) as data come in.

We used the keras package (which runs on top of Theano; documentation and installation instructions can be found at http://keras.io/) to implement a network with one layer of 100 relu nodes, and a linear output layer. We coarsely normalize the input features of the model to [0,1] by dividing by the maximum value for each. We found that, while our neural network did in fact learn (achieving a maximum score of >30), its performance was highly stochastic and unstable, and it underperformed the coarse-grained Q-table learners. We include a simple run of scores of 1000 epochs for reference.

Hyper-parameter tuning

We tried implementing different versions of neural networks, and found in general that performance was highly sensitive to network design and hyper-parameters. For example, we tried configuring a network with an output layer with 2 nodes, that would output Q values for each action taking only the state as inputs, but found it to be ineffective. Increasing the training rate caused Q values to diverge, but decreasing the training rate obviously caused training to slow. We lacked deep intuition about how to design layers of the network, but presumably this also influences performance.

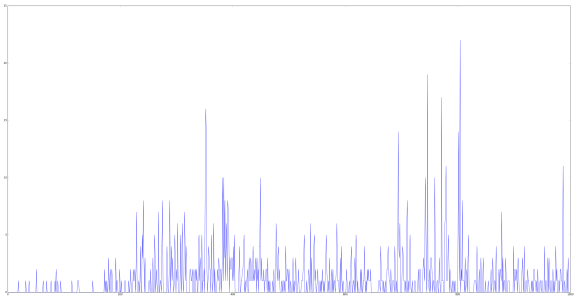


Figure 3: Example scores for a run of 1000 epochs of the network Q-approximator. Performance generally improves over time, but is stochastic and underperforms the coarse-grained Q-learner. In most 1000 epoch runs, the maximum score is about 30.

In general, as the neural network approximator seemed to underperform relative to the coarse-grained Q-table learner, we eventually abandoned the approach. However, it did train faster than the relatively less coarse-grained (e.g. 1 and 5 bin size) Q-table learners. It is possible that with more time and patience with hyper-parameter tuning, this approach could eventually outperform the coarse-grained learner.

```
# Imports.
import numpy as np
import numpy.random as npr

from SwingyMonkey import SwingyMonkey


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

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

                self.isFirstState = True # to catch weirdness for
first state of game
                self.isSecondState = False

                self.Q = {} # initialize Q table. use a dictionary
for now
                self.gamma = 0.7 # temporal discount value, not too
high since don't expect many duplicate states
                self.eps = 0.2 # do random action 5% of the time, for
exploration?
                self.g = npr.choice([1,4]) # make a random guess at
the gravity
                self.alpha = 0.7

                #self.lowTop = 0

        def reset(self):
                self.last_state  = None
                self.last_action = None
                self.last_reward = None
                self.g = npr.choice([1,4])
                self.eps = max([0.01, self.eps-0.001])
                self.isFirstState = True
                self.isSecondState = False

        def hash_state(self, state):
                # hashes a state dictionary to a Q-table index using
relevant features
                # monkey velocity, and r_x, r_y between monkey and
tree

                v = state['monkey']['vel']
                HorizDist = state['tree']['dist']
```

```
                TreebotDist = state['monkey']['bot']-state['tree']
['bot']
                Treetop = state['tree']['top']

                # coarse-grain position here, if necessary?

                dv = 1
                dx = 1
                dy = 1
                rv = np.round(float(v)/float(dv))
                rHor = np.round(float(HorizDist)/float(dx))
                rBot = np.round(float(TreebotDist)/float(dy))

                return [rv, rHor, rBot, Treetop]


        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.
                '''

                alpha = self.alpha

                # first, just sit tight if it's the first state
                if self.isFirstState:
                        self.last_state = state
                        self.last_action = 0
                        # move to the second state
                        self.isFirstState = False
                        self.isSecondState = True
                        # don't jump on the first state
                        return 0

                ## find hash values for different states
                s_old = self.hash_state(self.last_state)
                s_new = self.hash_state(state)

                a_old = self.last_action

                # calculate the gravity
                if self.isSecondState:
                        self.g = -state['monkey']['vel']
                        self.isSecondState = False

                '''
                don't jump if the top tree is really low
                if s_old[3] < 225:
                        self.lowTop +=1
```

```
                        print self.lowTop
                        return 0
                '''

                # get value to update from old state
                try:
                        Q0 = self.Q[self.g][s_old[0]][s_old[1]]
[s_old[2]][a_old]
                except KeyError:
                        Q0 = 0 # if no value there yet

                # find max Q value at new / current state, as well as
next action to return
                try:
                        QnewStay = self.Q[self.g][s_new[0]][s_new[1]]
[s_new[2]][0] # Q value for not jumping
                except KeyError:
                        QnewStay = 0
                try:
                        QnewJump = self.Q[self.g][s_new[0]][s_new[1]]
[s_new[2]][1] # Q value for jumping
                except KeyError:
                        QnewJump = 0

                if QnewJump > QnewStay: # if Q value higher for
jumping
                        new_action = 1
                        Qmax = QnewJump
                else: # otherwise, don't jump
                        new_action = 0
                        Qmax = QnewStay

                # epsilon greedy: with probability epsiolon,
overwrite new action w random one
                if npr.rand() < self.eps: # then choose randomly
                        new_action = npr.rand() < 0.5

                # and update Q value
                # will need layered try-catch blocks to handle
exceptions when the dictorary hashes don't exist yet
                # start trying to hash into actual state, then go
backwards from there

                try:
                        self.Q[self.g][s_old[0]][s_old[1]][s_old[2]]
[a_old] = (1-alpha)*Q0 + alpha*(self.last_reward + self.gamma*Qmax)

                except KeyError: # if we cannot hash in here?

                        try: # try one level up
```

```
                                        self.Q[self.g][s_old[0]][s_old[1]]
[s_old[2]] = {}
                                        self.Q[self.g][s_old[0]][s_old[1]]
[s_old[2]][a_old] = (1-alpha)*Q0 + alpha*(self.last_reward +
self.gamma*Qmax)

                                except KeyError: # if can't hash here either?

                                        try:
                                                self.Q[self.g][s_old[0]]
[s_old[1]] = {}
                                                self.Q[self.g][s_old[0]]
[s_old[1]][s_old[2]] = {}
                                                self.Q[self.g][s_old[0]]
[s_old[1]][s_old[2]][a_old] = (1-alpha)*Q0 + alpha*(self.last_reward +
self.gamma*Qmax)

                                        except KeyError: # etc....
                                                try:
                                                        self.Q[self.g]
[s_old[0]] = {}
                                                        self.Q[self.g]
[s_old[0]][s_old[1]] = {}
                                                        self.Q[self.g]
[s_old[0]][s_old[1]][s_old[2]] = {}
                                                        self.Q[self.g]
[s_old[0]][s_old[1]][s_old[2]][a_old] = (1-alpha)*Q0 +
alpha*(self.last_reward + self.gamma*Qmax)
                                                except KeyError:
                                                        # if no data for
this gravity yet; make the whole dictionary!
                                                        self.Q[self.g] = {}
                                                        self.Q[self.g]
[s_old[0]] = {}
                                                        self.Q[self.g]
[s_old[0]][s_old[1]] = {}
                                                        self.Q[self.g]
[s_old[0]][s_old[1]][s_old[2]] = {}
                                                        self.Q[self.g]
[s_old[0]][s_old[1]][s_old[2]][a_old] = (1-alpha)*Q0 +
alpha*(self.last_reward + self.gamma*Qmax)

                        #print self.Q[self.g][s_old[0]][s_old[1]][s_old[2]]
[a_old]
                        # update last action and state
                        self.last_action = new_action
                        self.last_state  = state

                        # and return action
                        return self.last_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 = 100, t_len = 1):
        '''
        Driver function to simulate learning by having the agent play
a sequence of games.
        '''

        for ii in range(iters):
                # Make a new monkey object.
                swing = SwingyMonkey(sound=False,                  #
Don't play sounds.

text="Epoch %d" % (ii),       # 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)
                #print swing.score, learner.eps

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

                #print ii

        return


if __name__ == '__main__':

        # Select agent.
        agent = Learner()

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

```
# Run games.
run_games(agent, hist, 5000, 1)

print max(hist)

# Save history.
np.save('hist2',np.array(hist))
```

```
# Imports.
import numpy as np
import numpy.random as npr

from SwingyMonkey import SwingyMonkey

# also: keras for neural network
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop, SGD


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

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


        self.isFirstState = True # to catch weirdness for first state
of game
        self.isSecondState = False # update gravity on *second* state

        #self.Q = {} # initialize Q table. use a dictionary for now
        self.gamma = 1 # temporal discount value? finite horizon so
maybe we don't need this?
        #self.eps0 = 1 # do random action 5% of the time, for
exploration?
        self.eps = 0.2 # start more random
        self.g = 0
        self.alpha = 0.5

        ## initialize neural network to random weights
        # this should be optimized still
        self.model = Sequential()
        self.model.add(Dense(output_dim = 100,
batch_input_shape=(1,5), init = 'lecun_uniform' ))
        self.model.add(Activation("relu"))
        # two outputs for 2 actions!
        self.model.add(Dense(1, init='lecun_uniform'))
        self.model.add(Activation("linear")) #linear output so we can
have range of real-valued outputs

        ## initialize model
        rms = RMSprop()
        self.model.compile(loss='mse', optimizer=rms)
```

```
        #sgd = SGD(lr=0.001)
        #self.model.compile(loss='mean_squared_error', optimizer=sgd)

    def reset(self, ii):
        self.last_state  = None
        self.last_action = None
        self.last_reward = None
        self.g = 0
        # decrement epsilon value?
        #self.eps = float(self.eps0)/float(ii+1)
        self.eps = max([0.01, self.eps-0.001]) # always do at least
10% random actions
        self.isFirstState = True
        self.isSecondState = False

    def getFeats(self, state, action):
        # takes state dictionary + action and converts to features
        # to feed into NN
        v = state['monkey']['vel']
        rx = state['tree']['dist']
        ry = state['monkey']['bot']-state['tree']['bot']
        h = state['monkey']['bot']

        # coarse-grain position here, if necessary?

        #dx = 1
        #dy = 1
        #rx = np.round(float(rx)/float(dx))
        #ry = np.round(float(ry)/float(dy))
        #h = np.round(float(h)/float(dy))

        #instead: normalize to max values?
        rx = float(rx)/float(300) # max dist of 300
        ry = float(ry)/float(200) # max diff here is +/- 200?
        v = float(v) / float(10) # guessing it's about 10ish max/min?
        g = float(self.g)/float(4) # 4 values?

        # can also coarse-grain velocity?
        #dv = 1
        #v = np.round(float(v)/float(dv))

        #tmp = [g, v, rx, ry, h] # 5-dimensional feature vector
        # now: try excluding height
        # also for now: ignore g?
        tmp = [g, v, rx, ry, action]
        # convert to 1x6 numpy array that NN expects
        feat = np.ndarray([1,5]) # 5 or however many features there
are
        feat[:] = tmp
```

```
        return feat

    def QNet(self, feat):
        # uses a neural network to approximate Q using a state-action
pair (s,a)
        return self.model.predict(feat)


    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.
        '''
        #print state
        # first, just sit tight if it's the first state
        if self.isFirstState:
            self.last_state = state
            self.last_action = 0
            #self.g = -state['monkey']['vel']
            #self.g = 1 # to train more quickly?
            # don't jump on the first state
            self.isFirstState = False
            self.isSecondState = True
            #self.model.train_on_batch(self.getFeats(self.last_state),
0*np.random.rand(1,2))
            return 0


        # if second state, then update gravity
        if self.isSecondState:
            self.g = -state['monkey']['vel']
            self.isSecondState = False

        # You might do some learning here based on the current state
and the last state.

        # You'll need to select and action and return it.
        # Return 0 to swing and 1 to jump.

        ## find Q values for old state
        Q0 = self.QNet(self.getFeats(self.last_state,
self.last_action))
        # and for new state!
        Qstay = self.QNet(self.getFeats(state,0))
        Qjump = self.QNet(self.getFeats(state,1))
        # take max
        if Qjump > Qstay: # if Q value higher for jumping
            new_action = 1
            Qmax = Qjump
        else: # otherwise, don't jump
```

```
            new_action = 0
            Qmax = Qstay
        #print Qstay, Qjump, new_action

        # update target vector?
        Qtarg =  self.last_reward + Qmax

        # gradient descent to update weights
        #self.model.fit(self.getFeats(self.last_state), Qtarg,
batch_size = 1, nb_epoch = 1, verbose = 0 )
        self.model.train_on_batch(self.getFeats(self.last_state,
self.last_action), Qtarg)

        # epsilon greedy: with probability epsiolon, overwrite new
action w random one
        if npr.rand() < self.eps: # then choose randomly
            new_action = npr.rand() < 0.5

        # update last action and state
        self.last_action = new_action
        self.last_state  = state

        # and return action
        return self.last_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 = 100, t_len = 100):
    '''
    Driver function to simulate learning by having the agent play a
sequence of games.
    '''

    for ii in range(iters):
        # Make a new monkey object.
        swing = SwingyMonkey(sound=False,                  # Don't
play sounds.
                             text="Epoch %d" % (ii),       # 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)
        print swing.score, learner.eps

        # Reset the state of the learner.
        learner.reset(ii)
        #print learner.Q


    return

if __name__ == '__main__':

        # Select agent.
        agent = Learner()

        # Empty list to save history.
        hist = []

        # Run games.
        run_games(agent, hist, 1000, 10)
       #print agent.Q

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

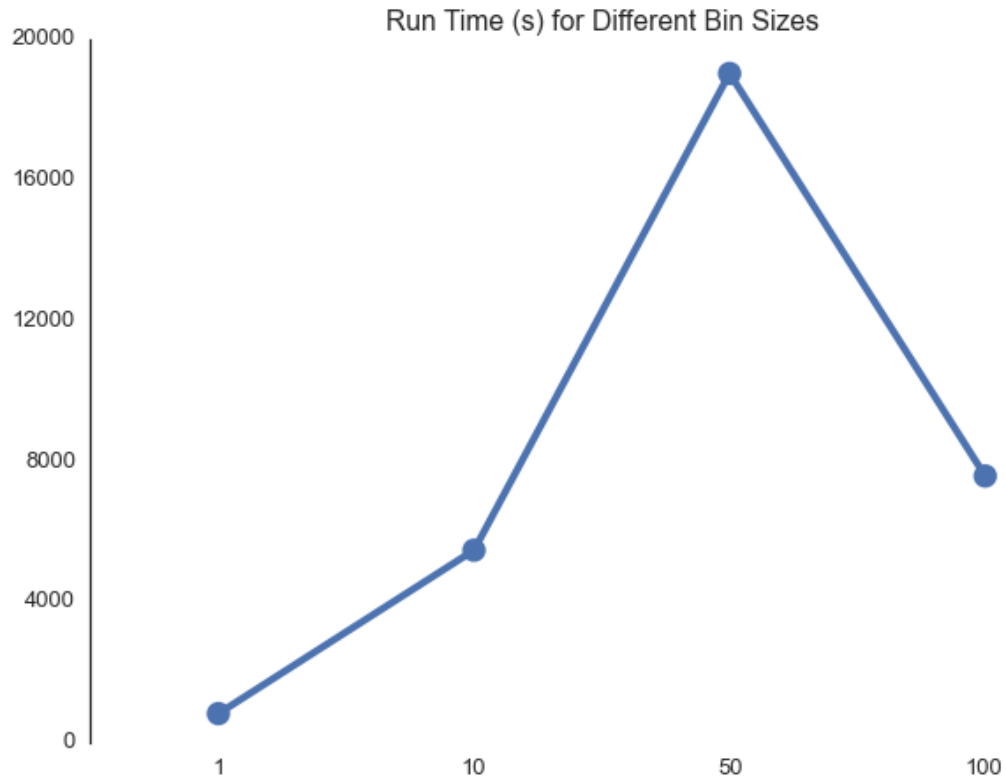
score_vis

April 29, 2016

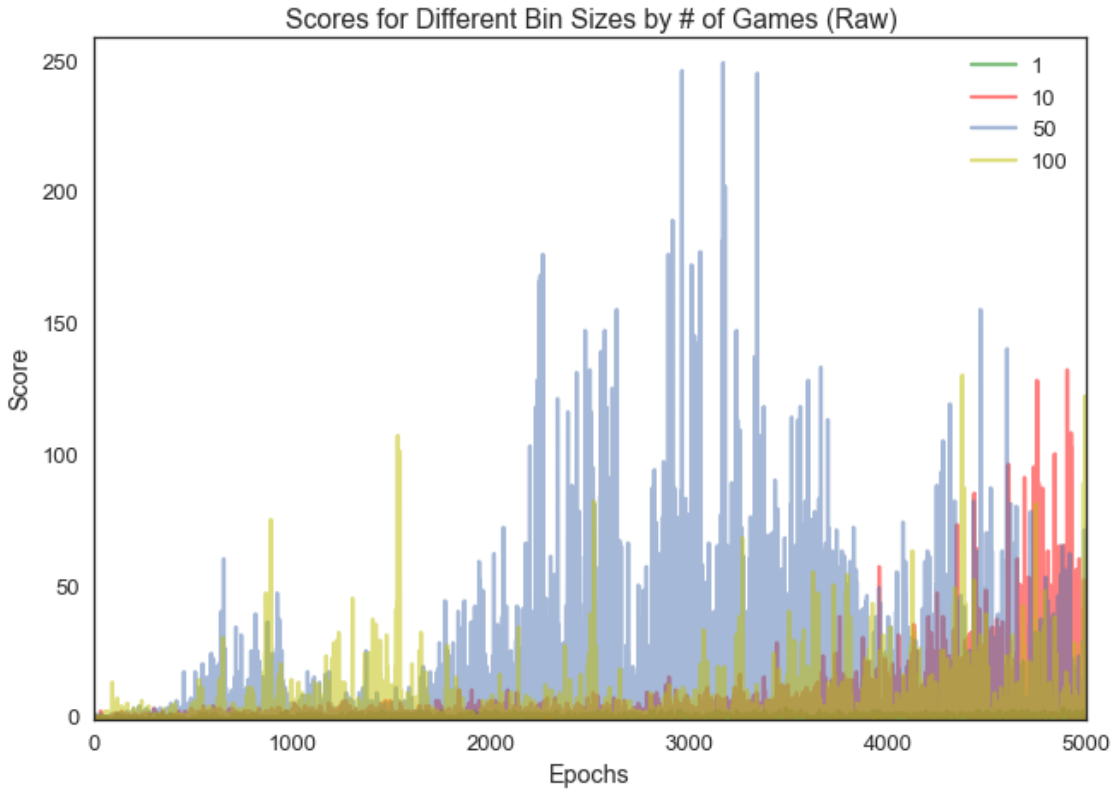
In [1]:
```
%matplotlib inline
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import pandas as pd
pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 100)
pd.set_option('display.notebook_repr_html', True)
import seaborn as sns
sns.set_style("whitegrid")
sns.set_context("poster")
```

In [12]:
```
sns.set(style="white", context="talk")
runtimes = [810.7, 5457.3, 19015.3, 7567.9]
ax = sns.pointplot(x=('1', '10', '50', '100'), y=runtimes)
ax.set(title="Run Time (s) for Different Bin Sizes",ylim=(0,20000),yticks=range(0,20001,4000))
for p in ax.patches:
    height = p.get_height()
    ax.text(p.get_x()+0.25, height+200, '%d'%height, fontsize=14)
sns.despine(bottom=True)
```

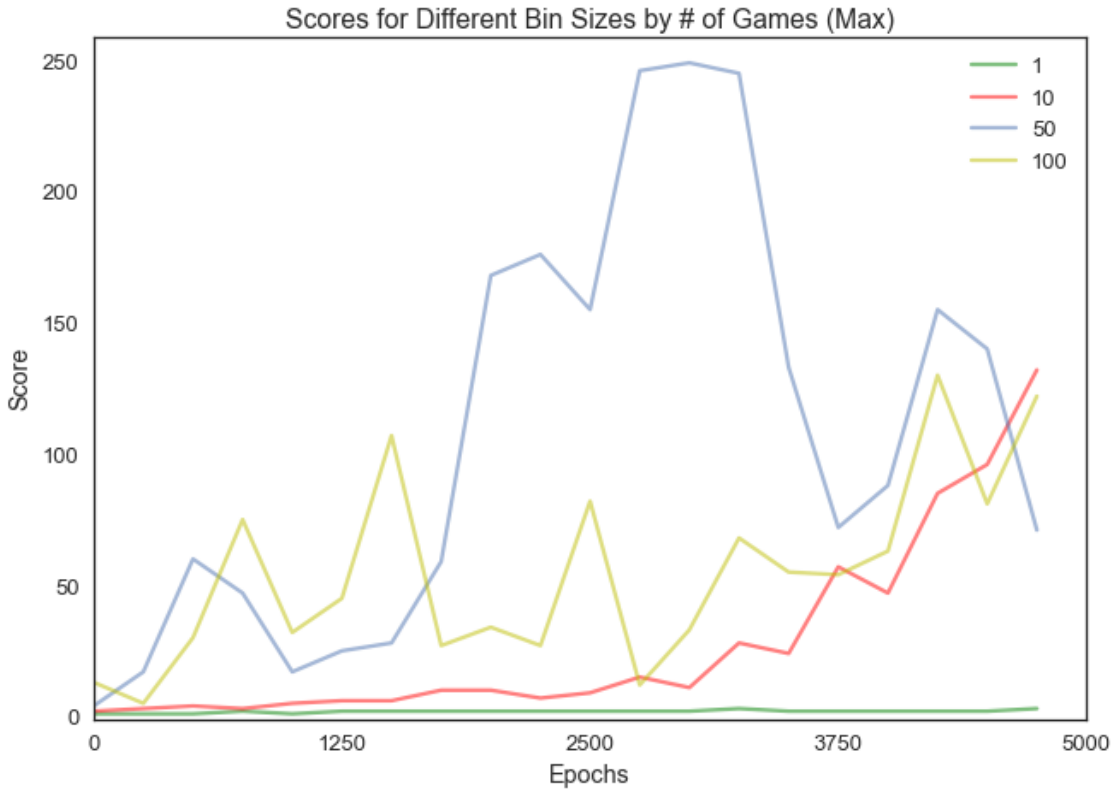
```
[]
```



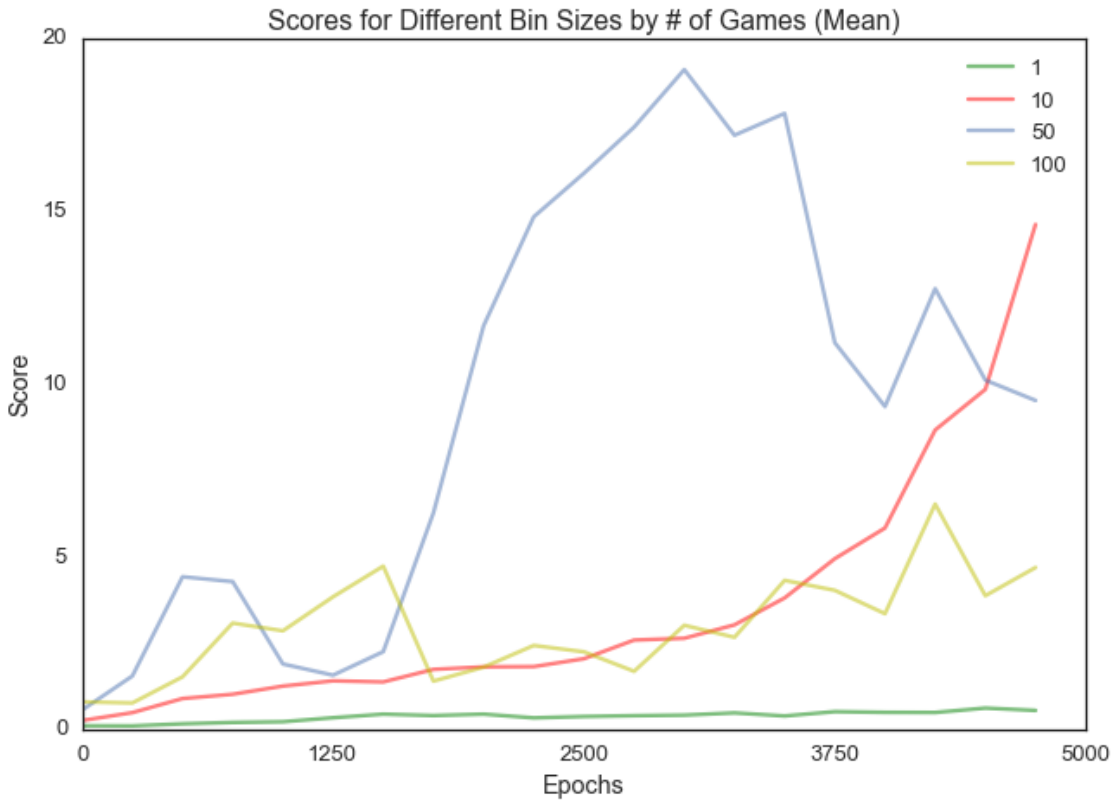

```
In [16]: d1 = np.load("1_5000.npy")
         d10 = np.load("10_5000.npy")
         d50 = np.load("50_5000.npy")
         d100 = np.load("100_5000")

In [29]: ds = [d1, d10, d50, d100]

In [55]: plt.plot(d1, alpha = 0.5, color='g', label = "1")
         plt.plot(d10, alpha = 0.5, color='r', label = "10")
         plt.plot(d50, alpha = 0.5, label = "50")
         plt.plot(d100, alpha = 0.5, color='y', label = "100")
         plt.legend()
         plt.title("Scores for Different Bin Sizes by # of Games (Raw)")
         plt.xlabel("Epochs")
         plt.ylabel("Score")
         plt.ylim(0,260)
         plt.show()
```




```
In [41]: interval = 500
         max1  = np.max(d1.reshape(-1, interval), axis=1)
         max10 = np.max(d10.reshape(-1, interval), axis=1)
         max50 = np.max(d50.reshape(-1, interval), axis=1)
         max100 = np.max(d100.reshape(-1, interval), axis=1)

In [57]: interval = 250
         max1  = np.max(d1.reshape(-1, interval), axis=1)
         max10 = np.max(d10.reshape(-1, interval), axis=1)
         max50 = np.max(d50.reshape(-1, interval), axis=1)
         max100 = np.max(d100.reshape(-1, interval), axis=1)
         plt.plot(max1, alpha = 0.5, color='g', label = "1")
         plt.plot(max10, alpha = 0.5, color='r', label = "10")
         plt.plot(max50, alpha = 0.5, label = "50")
         plt.plot(max100, alpha = 0.5, color='y', label = "100")
         plt.legend()
         plt.title("Scores for Different Bin Sizes by # of Games (Max)")
         plt.xlabel("Epochs")
         plt.ylabel("Score")
         plt.ylim(0,260)
         plt.xticks([0,5,10,15,20], [0,1250,2500,3750,5000])
         plt.show()
```



In [56]:
```
interval = 250
max1  = np.mean(d1.reshape(-1, interval), axis=1)
max10 = np.mean(d10.reshape(-1, interval), axis=1)
max50 = np.mean(d50.reshape(-1, interval), axis=1)
max100 = np.mean(d100.reshape(-1, interval), axis=1)
plt.plot(max1, alpha = 0.5, color='g', label = "1")
plt.plot(max10, alpha = 0.5, color='r', label = "10")
plt.plot(max50, alpha = 0.5, label = "50")
plt.plot(max100, alpha = 0.5, color='y', label = "100")
plt.legend()
plt.title("Scores for Different Bin Sizes by # of Games (Mean)")
plt.xlabel("Epochs")
plt.ylabel("Score")
plt.xticks([0,5,10,15,20], [0,1250,2500,3750,5000])
plt.show()
```


Scores for Different Bin Sizes by # of Games (Mean)
Score
Epochs
0
5
10
15
20
0
1250
2500
3750
5000
1
10
50
100

In [ ]: