```
# 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))
```