```
""" Trains an agent with (stochastic) Policy Gradients on Pong. Uses OpenAI Gym. """
import numpy as np
import pickle
import gym

# hyperparameters
H = 200 # number of hidden layer neurons
batch_size = 10 # every how many episodes to do a param update?
learning_rate = 1e-4
gamma = 0.99 # discount factor for reward
decay_rate = 0.99 # decay factor tempfor RMSProp leaky sum of grad^2
resume = False # resume from previous checkpoint?
render = False

# model initialization
D = 80 * 80 # input dimensionality: 80x80 grid
if resume:
  model = pickle.load(open('save.p', 'rb'))
else:
  model = {}
  model['W1'] = np.random.randn(H,D) / np.sqrt(D) # "Xavier" initialization
  model['W2'] = np.random.randn(H) / np.sqrt(H)

grad_buffer = { k : np.zeros_like(v) for k,v in model.items() } # update buffers that add up gradients over a b
rmsprop_cache = { k : np.zeros_like(v) for k,v in model.items() } # rmsprop memory

def sigmoid(x):
  return 1.0 / (1.0 + np.exp(-x)) # sigmoid "squashing" function to interval [0,1]

def prepro(I):
  """ prepro 210x160x3 uint8 frame into 6400 (80x80) 1D float vector """
  I = I[35:195] # crop
  I = I[::2,::2,0] # downsample by factor of 2
  I[I == 144] = 0 # erase background (background type 1)
  I[I == 109] = 0 # erase background (background type 2)
  I[I != 0] = 1 # everything else (paddles, ball) just set to 1
  return I.astype(np.float).ravel()

def discount_rewards(r):
  """ take 1D float array of rewards and compute discounted reward """
  discounted_r = np.zeros_like(r)
  running_add = 0
  for t in reversed(range(0, r.size)):
    if r[t] != 0: running_add = 0 # reset the sum, since this was a game boundary (pong specific!)
    running_add = running_add * gamma + r[t]
    discounted_r[t] = running_add
  return discounted_r

```

```
49: def policy_forward(x):
50:   h = np.dot(model['W1'], x)
51:   h[h<0] = 0 # ReLU nonlinearity
52:   logp = np.dot(model['W2'], h)
53:   p = sigmoid(logp)
54:   return p, h # return probability of taking action 2, and hidden state
55:
56: def policy_backward(eph, epdlogp):
57:   """ backward pass. (eph is array of intermediate hidden states) """
58:   dW2 = np.dot(eph.T, epdlogp).ravel()
59:   dh = np.outer(epdlogp, model['W2'])
60:   dh[eph <= 0] = 0 # backpro prelu
61:   dW1 = np.dot(dh.T, epx)
62:   return {'W1':dW1, 'W2':dW2}
63:
64: env = gym.make("Pong-v0")
65: observation = env.reset()
66: prev_x = None # used in computing the difference frame
67: xs,hs,dlogps,drs = [],[],[],[]
68: running_reward = None
69: reward_sum = 0
70: episode_number = 0
71: while True:
72:   if render: env.render()
73:
74:   # preprocess the observation, set input to network to be difference image
75:   cur_x = prepro(observation)
76:   x = cur_x - prev_x if prev_x is not None else np.zeros(D)
77:   prev_x = cur_x
78:
79:   # forward the policy network and sample an action from the returned probability
80:   aprob, h = policy_forward(x)
81:   action = 2 if np.random.uniform() < aprob else 3 # roll the dice!
82:
83:   # record various intermediates (needed later for backprop)
84:   xs.append(x) # observation
85:   hs.append(h) # hidden state
86:   y = 1 if action == 2 else 0 # a "fake label"
87:   dlogps.append(y - aprob) # grad that encourages the action that was taken to be taken (see http://cs231n.gith
88:
89:   # step the environment and get new measurements
90:   observation, reward, done, info = env.step(action)
91:   reward_sum += reward
92:
93:   drs.append(reward) # record reward (has to be done after we call step() to get reward for previous action)
94:
95:   if done: # an episode finished
96:     episode_number += 1
```

```
 97:
 98:     # stack together all inputs, hidden states, action gradients, and rewards for this episode
 99:     epx = np.vstack(xs)
100:     eph = np.vstack(hs)
101:     epdlogp = np.vstack(dlogps)
102:     epr = np.vstack(drs)
103:     xs,hs,dlogps,drs = [],[],[],[] # reset array memory
104:
105:     # compute the discounted reward backwards through time
106:     discounted_epr = discount_rewards(epr)
107:     # standardize the rewards to be unit normal (helps control the gradient estimator variance)
108:     discounted_epr -= np.mean(discounted_epr)
109:     discounted_epr /= np.std(discounted_epr)
110:
111:     epdlogp *= discounted_epr # modulate the gradient with advantage (PG magic happens right here.)
112:     grad = policy_backward(eph, epdlogp)
113:     for k in model: grad_buffer[k] += grad[k] # accumulate grad over batch
114:
115:     # perform rmsprop parameter update every batch_size episodes
116:     if episode_number % batch_size == 0:
117:       for k,v in model.items():
118:         g = grad_buffer[k] # gradient
119:         rmsprop_cache[k] = decay_rate * rmsprop_cache[k] + (1 - decay_rate) * g**2
120:         model[k] += learning_rate * g / (np.sqrt(rmsprop_cache[k]) + 1e-5)
121:         grad_buffer[k] = np.zeros_like(v) # reset batch gradient buffer
122:
123:     # boring book-keeping
124:     running_reward = reward_sum if running_reward is None else running_reward * 0.99 + reward_sum * 0.01
125:     print ('resetting env. episode reward total was %f. running mean: %f' % (reward_sum, running_reward))
126:     if episode_number % 100 == 0: pickle.dump(model, open('save.p', 'wb'))
127:     reward_sum = 0
128:     observation = env.reset() # reset env
129:     prev_x = None
130:
131:   if reward != 0: # Pong has either +1 or -1 reward exactly when game ends.
132:     print ('ep %d: game finished, reward: %f' % (episode_number, reward) + ('' if reward == -1 else ' !!!!!!!!')
```