Name: Hong Hai Tran
ID: 20174583

Deep Learning with Alpha Go – Project #2

Task 1. Linear environment

- Source code and plot

```
import numpy as np
import matplotlib.pyplot as plt
from qlearning import *

class linear_environment:
    def __init__(self):
        self.n_states = 21       # number of states
        self.n_actions = 2      # number of actions
        self.next_state = np.zeros([self.n_states, self.n_actions], dtype = np.int)
        for i in range(self.n_states):
            if (i == 0) or (i == 20):
                self.next_state[i] = i
                continue
            self.next_state[i,0] = i - 1
            self.next_state[i,1] = i + 1
        self.reward = np.zeros([self.n_states, self.n_actions])
        self.reward[1,0] = self.reward[19,1] = 1                    # set reward
        self.terminal = np.zeros(self.n_states, dtype = np.int)
        self.terminal[0] = self.terminal[self.n_states-1] = 1       # set terminal
        self.init_state = 10     # initial state

env = linear_environment() # Environment

class epsilon_profile: pass

# Epsilon = 1 (random walk), n_episodes = 1
print 'Epsilon = 1 (random walk), n_episodes = 1:'
n_episodes = 1      # number of episodes to run
max_steps = 1000    # max number of steps to run in each episode
alpha = 0.2         # learning rate
gamma = 0.9         # discount factor
epsilon = epsilon_profile()
epsilon.init = 1.    # initial epsilon in e-greedy
epsilon.final = 1.   # final epsilon in e-greedy
epsilon.dec_episode = 0.    # amount of decrement in each episode
epsilon.dec_step = 0.       # amount of decrement in each step
Q, n_steps, sum_rewards = Q_learning_train(env, n_episodes, max_steps, alpha, gamma, ep
silon)
print 'Q(s,a):'
print Q
test_n_episodes = 1     # number of episodes to run
test_max_steps = 1000   # max number of steps to run in each episode
test_epsilon = 0.       # test epsilon
test_n_steps, test_sum_rewards, s, a, sn, r = Q_test(Q, env, test_n_episodes, test_max_
steps, test_epsilon)
print 'test_sum_rewards is', test_sum_rewards[0]
print 'test_n_steps is', test_n_steps[0]

# Epsilon = 1 (random walk), n_episodes = 5
print '\nEpsilon = 1 (random walk), n_episodes = 5:'
```

```
49. n_episodes = 5  # number of episodes to run
50. Q, n_steps, sum_rewards = Q_learning_train(env, n_episodes, max_steps, alpha, gamma, ep
    silon)
51. print 'Q(s,a):'
52. print Q
53. test_n_steps, test_sum_rewards, s, a, sn, r = Q_test(Q, env, test_n_episodes, test_max_
    steps, test_epsilon)
54. print 'test_sum_rewards is', test_sum_rewards[0]
55. print 'test_n_steps is', test_n_steps[0]
56.
57. # Epsilon = 1 (random walk), n_episodes = 1000
58. print '\nEpsilon = 1 (random walk), n_episodes = 1000:'
59. n_episodes = 1000  # number of episodes to run
60. Q, n_steps, sum_rewards = Q_learning_train(env, n_episodes, max_steps, alpha, gamma, ep
    silon)
61. print 'Q(s,a):'
62. print Q
63. print 'Mean of n_steps is', np.mean(n_steps)
64. test_n_steps, test_sum_rewards, s, a, sn, r = Q_test(Q, env, test_n_episodes, test_max_
    steps, test_epsilon)
65. print 'test_sum_rewards is', test_sum_rewards[0]
66. print 'test_n_steps is', test_n_steps[0]
67.
68. # Epsilon is gradually decreased to 0, n_episodes = 100
69. print '\nEpsilon is gradually decreased to 0, n_episodes = 100:'
70. n_episodes = 100    # number of episodes to run
71. max_steps = 1000    # max number of steps to run in each episode
72. alpha = 0.2         # learning rate
73. gamma = 0.9         # discount factor
74. epsilon.init = 1.    # initial epsilon in e-greedy
75. epsilon.final = 0.   # final epsilon in e-greedy
76. epsilon.dec_episode = 1./n_episodes     # amount of decrement in each episode
77. epsilon.dec_step = 0.                   # amount of decrement in each step
78. Q, n_steps, sum_rewards = Q_learning_train(env, n_episodes, max_steps, alpha, gamma, ep
    silon)
79. print 'Q(s,a):'
80. print Q
81. test_n_episodes = 1     # number of episodes to run
82. test_max_steps = 1000   # max number of steps to run in each episode
83. test_epsilon = 0.       # test epsilon
84. test_n_steps, test_sum_rewards, s, a, sn, r = Q_test(Q, env, test_n_episodes, test_max_
    steps, test_epsilon)
85. print 'test_sum_rewards is', test_sum_rewards[0]
86. print 'test_n_steps is', test_n_steps[0]
87.
88. plt.figure(1)
89. x_plot = np.arange(1, n_episodes+1)
90. plt.plot(x_plot, n_steps)
91. plt.grid(True)
92. plt.xlim(0, n_episodes+1)
93. plt.ylim(ymin = 0)
94. plt.xlabel('n_episodes')
95. plt.ylabel('n_steps')
96. plt.show()
```

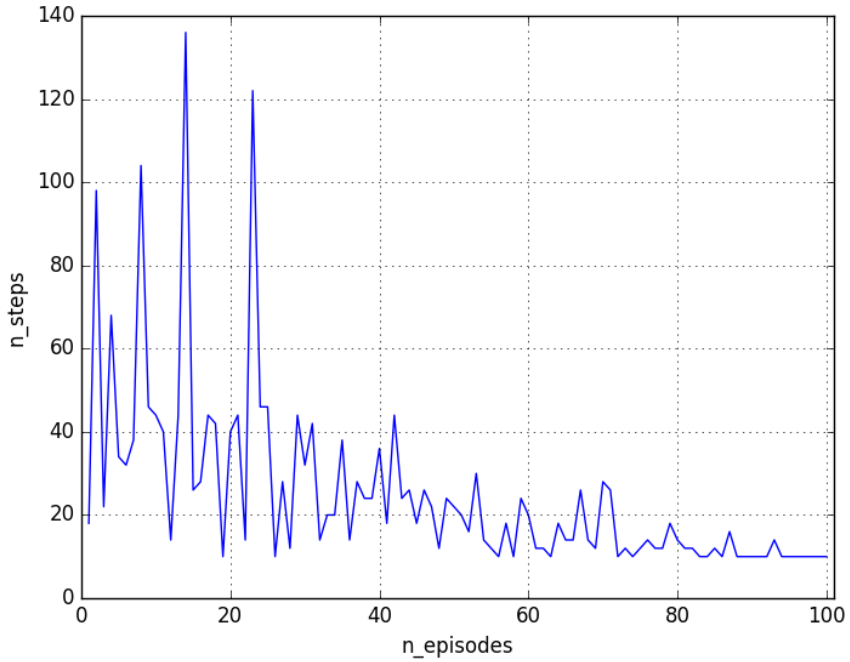

Figure 1. Task 1 - Linear environment

- Explanation on code design

  As the task has many small assignments, I divide my code into some sections, each section corresponds to one assignment.

  First thing is the linear_environment class to define environment for the reinforcement learning problem. In the class, number of states, number of valid actions in each state, next state when taking an action at a state, terminal state, reward, and initial state are all defined following the requirements.

  A class of epsilon profile is also defined to specify value of epsilon used during training.

  I first fix the value of epsilon during training, and run problem with n_episodes equal to 1, 5 and 1000. At each case, I print the Q values after training, as well as the total reward and number of taken steps when testing. In the case n_episodes equal to 1000, I also print the average steps taken during the whole training.

  Then I change the epsilon value from fixing to gradually changing to 0. At this time, I print Q values, total reward and number of taken steps when testing, and then plot a figure of number of taken steps as y axis and number of episode as x axis.

- Answers to given questions
  - For the first case, when n_episodes is 1, the agent never exhibits an optimal policy. The updating function of Q value is:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[R + \gamma \times max_{a'} Q(s',a')]$$
$$= 0.8Q(s,a) + 0.2[R + 0.9 \times max_{a'} Q(s',a')]$$

    At first, all the Q values are 0. When training happens, the first updating of Q value will be at the end of the first episode, when the agent gets the reward from the environment. The updated Q value is $0.8 \times 0 + 0.2 \times [1 + 0.9 \times 0] = 0.2$. This can only be Q(1,0) or Q(19,1) (but not both). All the other Q values are still 0. Therefore, when

the agent uses these Q values to decide which action to take at each state, it cannot get the optimal action because all Q values around the initial state (state 10) are equal to 0. The optimal policy is then unachievable.

- When n_episodes is 5, the agent sometimes exhibits an optimal policy but sometimes it does not. The achievability of optimal policy depends on how far the reward is propagated. At first episode, the agent goes to the final state, which results in the updating of Q(1,0) value or Q(19,1) value to 0.2 (explained above). After that, sometimes the agent can go to state 1 and state 19, but instead of going to terminal state, it goes back to the direction of initial state (state 10). This case helps in propagating rewards from the terminal state back through all other states, and helps the agent to choose optimal action in each state, therefore achieve optimal policy. However, sometimes the agent just goes to the terminal state, and no reward is propagated back. Therefore, all Q values for other states are still 0, which is the reason why sometimes the agent does not exhibit optimal policy.
- When n_episodes is 1000, the agent exhibits an optimal policy. The average number of steps per episode during training is about 100 steps (averaged value measured in several runs). This is because the value of epsilon is 1 during the whole training, which makes the agent takes action randomly, not optimally.
  - The Q values obtained by running the code:

```
[[ 0.          0.        ]
 [ 1.          0.81      ]
 [ 0.9         0.729     ]
 [ 0.81        0.6561    ]
 [ 0.729       0.59049   ]
 [ 0.6561      0.531441  ]
 [ 0.59049     0.4782969 ]
 [ 0.531441    0.43046721]
 [ 0.4782969   0.38742049]
 [ 0.43046721  0.34867844]
 [ 0.38742049  0.38742049]
 [ 0.34867844  0.43046721]
 [ 0.38742049  0.4782969 ]
 [ 0.43046721  0.531441  ]
 [ 0.4782969   0.59049   ]
 [ 0.531441    0.6561    ]
 [ 0.59049     0.729     ]
 [ 0.6561      0.81      ]
 [ 0.729       0.9       ]
 [ 0.81        1.        ]
 [ 0.          0.        ]]
```

  - Derive Q analytically

    We have the updating formula for Q value:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[R + \gamma \times max_{a'} Q(s',a')]$$
$$= 0.8 \times Q(s,a) + 0.2[R + 0.9 \times max_{a'} Q(s',a')]$$

    We know that Q converges to q_*. Therefore, we have:

$$q_*(s,a) = 0.8 \times q_*(s,a) + 0.2[R + 0.9 \times max_{a'} q_*(s',a')]$$
$$\Leftrightarrow q_*(s,a) = R + 0.9 \times max_{a'} q_*(s',a')$$

Consider $s = 19$ and $a = 1$:

$$q_*(19,1) = 1 + 0.9\times 0 = 1$$

For $s = 18$ and $a = 1$:

$$q_*(18,1) = 0 + 0.9\times max_a q_*(19,a) = 0.9\times q_*(19,1) = 0.9\times 1 = 0.9$$

because $q_*(19,1)$ leads to terminal state with reward +1, so it is always bigger than $q_*(19,0)$ which leads the agent to the reverse direction.

Similarly, for $s = 17$ and $a = 1$:

$$q_*(17,1) = 0 + 0.9\times max_a q_*(18,a) = 0.9\times q_*(18,1)$$
$$= 0.9\times 0.9 = 0.9^2 = 0.81$$

Then, for all s having values from 10 to 19, we have:

$$q_*(s,1) = 0.9^{19-s} \quad \text{where } s \in \{10,11,\dots,19\} \qquad (1)$$

Similarly, for all s having values from 1 to 10, we can obtain:

$$q_*(s,0) = 0.9^{s-1} \quad \text{where } s \in \{1,2,\dots,10\} \qquad (2)$$

We can observe that $q_*(10,0) = q_*(10,1)$, which implies that from the initial state, moving to either left or right gives the same reward.

Now, consider $s = 19$ and $a = 0$:

$$q_*(19,0) = 0 + 0.9\times max_a q_*(18,a) = 0.9\times q_*(18,1)$$
$$= 0.9\times 0.9 = 0.9^2 = 0.81$$

For $s = 18$ and $a = 0$:

$$q_*(18,0) = 0 + 0.9\times max_a q_*(17,a) = 0.9\times q_*(17,1)$$
$$= 0.9\times 0.9^2 = 0.9^3 = 0.729$$

Then, for all s having values from 11 to 19, we have:

$$q_*(s,0) = 0.9^{21-s} \quad \text{where } s \in \{11,12,\dots,19\} \qquad (3)$$

Similarly, for all s having values from 1 to 9, we have:

$$q_*(s,1) = 0.9^{s+1} \text{ where } s \in \{1,2,\dots,9\} \qquad (4)$$

For $s = 0$ or $s = 20$, which are two terminal states:

$$q_*(s,a) = 0 \quad \text{where } s \in \{0,20\}, a \in \{0,1\} \qquad (5)$$

Using the inferred formulas from (1) to (5), we can see that the Q values obtained by running the code are close to theoretical values.

- When choosing epsilon = 1 initially and linearly reduce it to 0 at the end of training, the agent is both exploring and exploiting depending on the value of epsilon. Figure 1 shows number of episodes as x axis and number of taken steps as y axis. We can see that the number of steps converges to the optimal value at the end of training.

Task 2. 4-legged spider

- Source code
  - spider.py

```
import matplotlib.pyplot as plt
import numpy as np
from qlearning import *
from spider_ani import *
from spider_env import *
from wait import *

env = spider_environment()

n_episodes = 1          # number of episodes to run, 1 for continuing task
max_steps = 10000000    # max. # of steps to run in each episode
alpha = 0.1             # learning rate
gamma = 0.9             # discount factor

class epsilon_profile: pass
epsilon = epsilon_profile()
epsilon.init = 1.    # initial epsilon in e-greedy
epsilon.final = 0.   # final epsilon in e-greedy
epsilon.dec_episode = 0.  # amount of decrement in each episode
epsilon.dec_step = 1. / max_steps   # amount of decrement in each step

Q, n_steps, sum_rewards = Q_learning_train(env, n_episodes, max_steps, alpha, gamma, epsilon)
print(sum_rewards[0])

test_n_episodes = 1  # number of episodes to run, 1 for continuing task
test_max_steps = 20  # max. # of steps to run in each episode
test_epsilon = 0.    # test epsilon
test_n_steps, test_sum_rewards, s, a, sn, r = Q_test(Q, env, test_n_episodes, test_max_steps, test_epsilon)
for j in range(test_max_steps):
    print('Step %02d: %10s %10s %10s %5.2f' % (j+1, bin(s[0,j]), bin(a[0,j]), bin(sn[0,j]), r[0,j]))
print(test_sum_rewards[0])

ani = spider_animation(Q, env, test_max_steps, test_epsilon)
ani.save('spider.mp4', dpi=200)
# plt.show(block=False)
# wait('Press enter to quit')
```

  - spider_env.py

```
import numpy as np

# Spider Environment
class spider_environment:
    def __init__(self):
        self.n_states = 256
        self.n_actions = 256
        self.reward = np.zeros([self.n_states, self.n_actions])
        self.terminal = np.zeros(self.n_states, dtype=np.int)   # 1 if terminal state
        self.next_state = np.zeros([self.n_states, self.n_actions], dtype=np.int)
        transition = [[1,0,2,0],[1,0,3,1],[3,2,2,0],[3,2,3,1]]
        for s in range(self.n_states):
```

```
            for a in range(self.n_actions):
                total_down  = 0.
                total_force = 0.
                legStillDown = [0, 0, 0, 0]
                for i in range(4):  # 4 is number of legs
                    # Set next_state
                    s_eachLeg  = (s >> (2*i)) & 3
                    a_eachLeg  = (a >> (2*i)) & 3
                    sn_eachLeg = transition[s_eachLeg][a_eachLeg]
                    self.next_state[s,a] += (sn_eachLeg << (2*i))
                    # Set total_force
                    s_legUp = s_eachLeg & 1
                    s_legFw = (s_eachLeg >> 1) & 1
                    a_legFw = (a_eachLeg & 3) == 2
                    a_legBw = (a_eachLeg & 3) == 3
                    total_force += ((s_legUp == 0 and s_legFw == 1 and a_legBw == 1) -
(s_legUp == 0 and s_legFw == 0 and a_legFw == 1))
                    # Set total_down
                    sn_legUp = sn_eachLeg & 1
                    legStillDown[i] = (s_legUp == 0 and sn_legUp == 0)
                    total_down += legStillDown[i]
                # Set reward
                if total_down == 0.:
                    self.reward[s,a] = 0
                elif total_down >= 3.:
                    self.reward[s,a] = total_force / total_down
                elif (total_down == 2) and ((legStillDown[0] == 1 and legStillDown[3] =
= 1)
                    or (legStillDown[1] == 1 and legStillDown[2] == 1)):
                    self.reward[s,a] = total_force / total_down
                else:
                    self.reward[s,a] = 0.25 * total_force / total_down
        self.init_state = 0b00001010    # initial state
```

- Explanation on code design
  - spider.py

    First, the environment is obtained. Because this is a continuing task, I use n_episode of 1 and max_steps of 10,000,000 (10 million steps). Learning rate is set to 0.1 and discount rate is set to 0.9. The epsilon is set so that initial value is 1.0 and end value is 0. The decreasing amount after each step is set to 1/max_steps. There is only one episode, so the decreasing amount for each episode is set to 0. After that, I train the agent using the given *Q_learning_train* function and test the agent using the given *Q_test* function. In the end, I save the video of the agent taking actions following trained Q values.
  - spider_env.py

    I declare a class which first declares number of states and number of actions. After that, I initialized matrices of rewards and next states as well as terminal states, which are all 0 (no terminal state). Next, I define a transition matrix where each value transition[state][action] corresponds to next state of one leg, because all the legs have the same behavior.

    The first for loop wraps around the available states, and the second one wraps around the available actions. The for loops are used to calculate value of next states and rewards. For each iterated pair of state and action, I first define total_force and

total_down to be 0. The legStillDown array is to determine the legs which are down both before and after taking an action. Then I have another for loop which wraps around the number of legs, which is 4. In this for loop, I get state and action of each leg using right bit-shifting (>>) and & operator. After that, I use transition matrix to infer the next state of the corresponding leg, then use left bit-shifting and addition to calculate the next state of all the legs (8 bits) after looping over the 4 legs. I also calculate value of total_down and total_force in each loop. After that, I use the value of total_down and legStillDown array to calculate the reward corresponding to current pair of state and action. Finally, I set the value of initial state.

- Values of state, action, next state and reward during 20 test steps are shown below using the mentioned order:

```
Step 01:     0b1010 0b11001111    0b10000  0.67
Step 02:    0b10000  0b1101000   0b111001 -0.12
Step 03:   0b111001    0b10111  0b1101001  0.00
Step 04:  0b1101001 0b10111110 0b11000011  1.00
Step 05: 0b11000011  0b1000001 0b10010110  0.00
Step 06: 0b10010110 0b11101011   0b111100  1.00
Step 07:   0b111100    0b10100  0b1101001  0.00
Step 08:  0b1101001 0b10111110 0b11000011  1.00
Step 09: 0b11000011  0b1000001 0b10010110  0.00
Step 10: 0b10010110 0b11101011   0b111100  1.00
Step 11:   0b111100    0b10100  0b1101001  0.00
Step 12:  0b1101001 0b10111110 0b11000011  1.00
Step 13: 0b11000011  0b1000001 0b10010110  0.00
Step 14: 0b10010110 0b11101011   0b111100  1.00
Step 15:   0b111100    0b10100  0b1101001  0.00
Step 16:  0b1101001 0b10111110 0b11000011  1.00
Step 17: 0b11000011  0b1000001 0b10010110  0.00
Step 18: 0b10010110 0b11101011   0b111100  1.00
Step 19:   0b111100    0b10100  0b1101001  0.00
Step 20:  0b1101001 0b10111110 0b11000011  1.00
```

Task 3. Breakout and DQN

- Source code
  - breakout.py

```
import numpy as np
import tensorflow as tf
import random
from breakout_env import *

# An instance of environment
env = breakout_environment(5, 8, 3, 1, 2)

xavier_init = tf.contrib.layers.xavier_initializer(uniform = False)
xavier_conv_init = tf.contrib.layers.xavier_initializer_conv2d(uniform = False)
def q_network(X_input, env, name):
    with tf.variable_scope(name) as scope:
        conv1 = tf.layers.conv2d(X_input, filters = 20, kernel_size = [3,3],\
            strides = 1, padding = 'VALID', activation = tf.nn.relu,\
        kernel_initializer = xavier_conv_init)
        conv2 = tf.layers.conv2d(conv1, filters = 40, kernel_size = [2,2],\
            strides = 1, padding = 'VALID', activation = tf.nn.relu,\
        kernel_initializer = xavier_conv_init)
        conv2_flat = tf.reshape(conv2, shape = [-1, 5*2*40])
        fc1 = tf.layers.dense(conv2_flat, units = 100, activation = tf.nn.relu,\
        kernel_initializer = xavier_init)
        outputs = tf.layers.dense(fc1, env.na, kernel_initializer = xavier_init)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,\
                                       scope = scope.name)
    trainable_vars_name = {var.name[len(scope.name):]: var\
                           for var in trainable_vars}
    return outputs, trainable_vars_name

# Define networks
X_input = tf.placeholder(tf.float32, shape = [None, env.ny, env.nx, env.nf])
train_q_vals , train_vars  = q_network(X_input, env, name = "q_network/train")
target_q_vals, target_vars = q_network(X_input, env, name = "q_network/target")

# Copy train DQN to target DQN
copy_ops = [target_var.assign(train_vars[var_name])
            for var_name, target_var in target_vars.items()]
update_target_dqn = tf.group(*copy_ops)

# Define training configs
learning_rate = 1e-4
with tf.variable_scope("training"):
    X_action = tf.placeholder(tf.int32, shape = [None])
    target_q = tf.placeholder(tf.float32, shape = [None, 1])
    X_act_onehot = tf.one_hot(X_action, env.na, dtype = tf.float32)
    train_q = tf.reduce_sum(tf.multiply(train_q_vals, X_act_onehot),\
                            axis = 1, keep_dims = True)
    cost = tf.reduce_sum(tf.square(target_q - train_q))
    optimizer = tf.train.AdamOptimizer(learning_rate)
    training = optimizer.minimize(cost)

# Class ExperienceMemory to store experience
class ExperienceMemory():
    def __init__(self, memorySize = 1000):
        self.memory = []
        self.memorySize = memorySize
    def add(self, experience):
```

```
        if len(self.memory) + len(experience) >= self.memorySize:
            self.memory[0:(len(experience)+len(self.memory))-self.memorySize] = []
        self.memory.extend(experience)
    def sample(self, size):
        return np.reshape(np.array(random.sample(self.memory, size)),[size, 5])

# Start tensorflow session
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
saver = tf.train.Saver()

# Training parameters
gamma = 0.99            # Discount rate
n_episodes = 2000       # Number of episodes
max_steps = 200         # Max steps in one episode
memorySize = 5000       # Size of experience memory
preTrainSteps = 5000    # Step of random actions before training begins
miniBatchSize = 32      # Experience taken from memory in each training iteration
totalSteps = 0          # Used to track when to start training
targetUpdateStep = 500  # Number of training steps to wait to update target network
trainingStep = 0        # Used to track the time to update target network
initEpsilon = 1.        # Initial value of epsilon
endEpsilon = 0.         # End value of epsilon
epsilonStep = 1./1000   # Epsilon changing rate
epsilon = initEpsilon   # Epsilon value used for e-greedy

# Init experience memory
expMem = ExperienceMemory(memorySize)

# Copy initial value of weights from trainQN to target QN
update_target_dqn.run()

# DQN with experience replay
for episode in range(n_episodes):
    s = env.reset()
    reward = 0
    for step in range(max_steps):
        if np.random.rand() < epsilon or totalSteps < preTrainSteps:
            a = np.random.randint(env.na)   # random action
        else:   # Optimal action
            y_hat = train_q_vals.eval(feed_dict = {X_input: np.reshape(s, [1, env.ny, env.
nx, env.nf])})
            a = np.random.choice(np.where(y_hat[0]==np.max(y_hat))[0])
        # Next step in environment
        sn, r, terminal, _, _, _, _, _, _, _, _ = env.run(a - 1)
        totalSteps += 1
        reward += r
        # Add experience to memory
        expMem.add(np.array([s, a, r, sn, terminal]).reshape(1,5))
        # Check if training can be started
        if totalSteps > preTrainSteps:
            # Sample random minibatch
            miniBatch = expMem.sample(miniBatchSize)
            sBatch   = np.vstack(miniBatch[:,0]).reshape(miniBatchSize, env.ny, env.nx, en
v.nf)
            aBatch   = np.vstack(miniBatch[:,1]).reshape(miniBatchSize)
            # Calculate target Q value
            q_target = []
            for i in range(miniBatchSize):
                if miniBatch[i,4] == 1:     # If terminal state
                    q_target.append(miniBatch[i,2])
```

```
                else:
                    y_target = target_q_vals.eval(\
                        feed_dict = {X_input: np.reshape(miniBatch[i,3], [1, env.ny, env.n
x, env.nf])})
                    q_target.append(miniBatch[i,2] + gamma * np.max(y_target))
            q_target = np.array(q_target).reshape(miniBatchSize,1)
            # Start training
            training.run(feed_dict = {X_input: sBatch, X_action: aBatch, target_q: q_targe
t})
            trainingStep += 1
            # Update target network
            if trainingStep % targetUpdateStep == 0:
                update_target_dqn.run()
        if terminal:
            if totalSteps > preTrainSteps and epsilon > endEpsilon:
                epsilon -= epsilonStep
            if totalSteps > preTrainSteps:
                currCost = cost.eval(feed_dict = {X_input: sBatch, X_action: aBatch, targe
t_q: q_target})
                print 'Episode: %d\tStep: %3d\tReward: %2d\tCost: %f\tEpsilon: %f'\
                    %(episode+1, step+1, reward, currCost, epsilon)
            break
        # Update two most recent frames
        s = sn

# Save trained parameters
save_path = saver.save(sess, "./breakout.ckpt")
```

- breakout_ani.py

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib.patches as patches
from breakout_env import *
from wait import *

env = breakout_environment(5, 8, 3, 1, 2)

def q_network(X_input, env, name):
    with tf.variable_scope(name) as scope:
        conv1 = tf.layers.conv2d(X_input, filters = 20, kernel_size = [3,3],\
            strides = 1, padding = 'VALID', activation = tf.nn.relu)
        conv2 = tf.layers.conv2d(conv1, filters = 40, kernel_size = [2,2],\
            strides = 1, padding = 'VALID', activation = tf.nn.relu)
        conv2_flat = tf.reshape(conv2, shape = [-1, 5*2*40])
        fc1 = tf.layers.dense(conv2_flat, units = 100, activation = tf.nn.relu)
        outputs = tf.layers.dense(fc1, env.na)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,\
                                       scope = scope.name)
    trainable_vars_name = {var.name[len(scope.name):]: var\
                           for var in trainable_vars}
    return outputs, trainable_vars_name

X_input = tf.placeholder(tf.float32, shape = [None, env.ny, env.nx, env.nf])
train_q_vals , train_vars  = q_network(X_input, env, name = "q_network/train")

# Load trained parameters of network
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
saver = tf.train.Saver()
```

```
try:
    saver.restore(sess, "./breakout.ckpt")
    print 'Loading trained model'
except:
    print 'A new model is being trained'

class breakout_animation(animation.TimedAnimation):
    def __init__(self, env, max_steps, frames_per_step = 5):
        self.env = env
        self.max_steps = max_steps
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_aspect('equal')
        self.objs = []
        # boundary
        w = 0.1
        ax.plot([-w,-w,env.nx+w,env.nx+w],[0,env.ny+w,env.ny+w,0],'k-',linewidth=5)
        # bricks
        wb = 0.05
        self.bricks = []
        self.brick_colors = [['red'], ['blue','red'], ['blue','green','red'], ['blue','gre
en','yellow','red'], ['blue','green','yellow','orange','red'], \
            ['purple','blue','green','yellow','brown','orange','red'], ['purple','blue','g
reen','yellow','brown','orange','red']]    # add more colors if needed
        for y in range(self.env.nb):
            b = []
            yp = y + (self.env.ny - self.env.nt - self.env.nb)
            for x in range(self.env.nx):
                b.append(patches.Rectangle((x + wb, yp + wb), 1-2*wb, 1-
2*wb, edgecolor='none', facecolor=self.brick_colors[self.env.nb-1][y]))
                ax.add_patch(b[x])
                self.objs.append(b[x])
            self.bricks.append(b)
        # ball
        self.ball = patches.Circle(env.get_ball_pos(0.), radius = 0.15, color = 'red')
        ax.add_patch(self.ball)
        self.objs.append(self.ball)
        # score text
        self.text = ax.text(0.5 * env.nx, 0, '', ha='center')
        self.objs.append(self.text)
        # game over text
        self.gameover_text = ax.text(0.5 * env.nx, 0.5 * env.ny, '', ha='center')
        self.objs.append(self.gameover_text)
        self.frames_per_step = frames_per_step
        self.total_frames = self.frames_per_step * self.max_steps
        # paddle
        self.paddle = patches.Rectangle((env.p, 0.5), 1, 0.5, edgecolor='none', facecolor=
'red')
        ax.add_patch(self.paddle)
        # for early termination of animation
        self.iter_objs = []
        self.iter_obj_cnt = 0
        # interval = 50msec
        animation.TimedAnimation.__init__(self, fig, interval=50, repeat=False, blit=False
)

    def _draw_frame(self, k):
        if self.terminal:
            return
        if k == 0:
```

```
            self.iter_obj_cnt -= 1
        if k % self.frames_per_step == 0:
            q_values = train_q_vals.eval(feed_dict = {X_input: np.reshape(self.env.s, (1,
self.env.ny, self.env.nx, self.env.nf))})
            self.a = np.argmax(q_values) - 1
            self.p = self.env.p
            self.pn = min(max(self.p + self.a, 0), self.env.nx - 1)
        t = (k % self.frames_per_step) * 1. / self.frames_per_step
        self.ball.center = self.env.get_ball_pos(t)
        self.paddle.set_x(t * self.pn + (1-t) * self.p)
        if k % self.frames_per_step == self.frames_per_step - 1:
            sn, reward, terminal, p0, p, bx0, by0, vx0, vy0, rx, ry = self.env.run(self.a)

            self.sum_reward += reward
            if reward > 0.:
                self.bricks[ry][rx].set_facecolor('none')
                self.text.set_text('Score: %d' % self.sum_reward)
            if terminal:
                self.terminal = terminal
                self.gameover_text.set_text('Game Over')
                for _ in range(self.total_frames - k - 1):
                    self.iter_objs[self.iter_obj_cnt].next()     # for early termination o
f animation (latest iterator is used first)
        self._drawn_artists = self.objs

    def new_frame_seq(self):
        iter_obj = iter(range(self.total_frames))
        self.iter_objs.append(iter_obj)
        self.iter_obj_cnt += 1
        return iter_obj

    def _init_draw(self):
        _ = self.env.reset()
        self.sum_reward = 0.
        self.p = self.env.p    # current paddle position
        self.pn = self.p       # next paddle position
        self.a = 0             # action
        self.terminal = 0
        for y in range(self.env.nb):
            for x in range(self.env.nx):
                self.bricks[y][x].set_facecolor(self.brick_colors[self.env.nb-1][y])
        self.ball.center = self.env.get_ball_pos(0.)
        self.paddle.set_x(self.p)
        self.text.set_text('Score: 0')
        self.gameover_text.set_text('')

ani = breakout_animation(env, 200)
ani.save('breakout.mp4', dpi=200)
# plt.show(block=False)
# wait('Press enter to quit')
```

- Explanation on code design
  - breakout.py

    First I get an instance of the breakout game environment. The game environment will have 8 pixels in height, 5 pixels in width and provides 2 frames for training. The number of valid actions is 3, which are moving left, moving right and not moving. Next I define Xavier initializer to use to initialize weight values. Then I define a function *q_network* which defines a deep convolutional neural network. Input of the network

is the two frames that the environment provides. The network consists of two convolutional layers, one fully-connected hidden layer and one output layer. The first convolutional layer has 20 filters, each filter has 3 pixels in each dimension height and width, with stride set to 1 for both dimensions. The padding is set to 'VALID' which means no padding. The second convolutional layer has 40 filters, each has 2 pixels in each dimension height and width, with stride set to 1 for both dimensions. I also do not use padding in this layer. Both convolutional layers use xavier_initializer_conv2d to initialize the weight values. The next hidden layer is a fully-connected layer with 100 units. The three mentioned layers all use rectifier linear unit (ReLU) as activation function. The output layer is a fully-connected layer with 3 output, each corresponding to a valid action. The function returns output values and trainable parameters of the network. After that, I create two networks using the defined function, one used for training (training network) and one used to generate labels for training process (target network). Then I define copy operator to copy weight values from training network to target network.

Next, I define training configurations. I use mean squared error (MSE) as the cost function and use Adam Optimizer (instead of RMSProp, which is used in the DQN paper by V. Mnih et. al) with learning rate of 1e-4. The inputs to the training process include two frames at state S_t, two frames at state S_{t+1} and chosen action of agent. Next I design an experience memory which can add experience and sample experience randomly. The experience here includes state s, action a, reward R(s,a), next state s' and a value to indicate whether s' is a terminal state or not. The discount rate γ is set to 0.99. Number of episodes for training is 2000, and maximum number of steps in each episode is set to 200. The memory size of experience is set to 5000.

Before starting training, I initialize the experience memory and copy initial weight values from training network to target network. At each episode, the first thing I do is to reset the environment (reset game). For training, first I use e-greedy to choose action for agent. The epsilon value is initialized to 1 and gradually decrease to 0 during training. The decrease happens at the end of each episode. Then I run next step in the environment. The corresponding experience is stored into the experience memory. Because the experience memory is empty at first, so in the first 5000 training steps, I choose action randomly to fill up the memory. No training happens. After pre-train, in each step, I uniformly sample a batch of experience from memory (in this case, size of the batch is 32) and use these to train the network. The target network is updated every 500 training steps. At the end of each training step, I also update the current state by setting current state *s* equal to the next state *sn*. After training, I save the trained parameters into breakout.ckpt files.

- breakout_ani.py

  First I define the same network as in breakout.py file, then I load the trained parameters from *breakout.ckpt** files. In the function *_draw_frame*, instead of taking random action, I change the code to use the output of the neural network to choose optimal action. The rest of the code is left unchanged.

- Instruction on running the code
  - First run breakout.py, which trains the network and saves trained parameters

- Then run breakout_ani.py, which saves the video of agent playing game. The code can be change to play animation instead of saving video using the last two lines.

- Performance of agent
  The trained agent achieves the maximum score (15) with 86 steps.
- 5 screen shots of the video