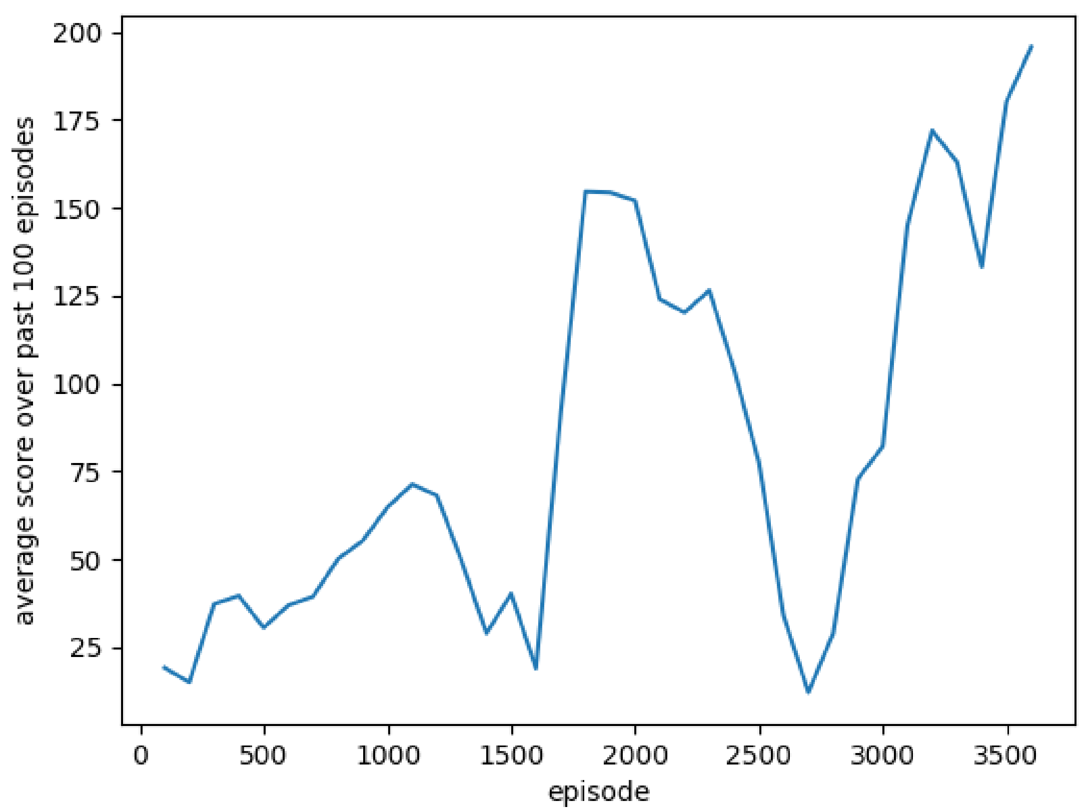
EECS 498: Reinforcement Learning
Homework 5 Responses

Tejas Jha
tjha

December 12, 2018

This document includes my responses to Homework 5 questions. Responses that involved the use of coding will provide references to specific lines of code to provide a better overview of how the problem was approached. The code can either be referenced in the Appendix or in the accompanied python script submitted with this assignment.

Question 1



(a)

(b) To try and get the model working on MountainCar-v0,

Suppose the reward function for an MDP is a linear function of d features for a state s

$$R(s) = \alpha_1\phi_1(s) + \alpha_2\phi_2(s) + ... + \alpha_d\phi_d(s)$$

where the $\phi_1...\phi_d$ are fixed, known and bounded basis functions mapping from state space S to the reals.

As presented in the Algorithms for Inverse Reinforcement Learning paper, we can define a value function for a policy π that maps $S \mapsto A$ for any state s_1 as the following:

$$V^\pi(s_1) = E[R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + ...|\pi]$$

where the expectation is over the distribution of the state sequence $(s_1, s_2, ...)$.

Additionally from the same paper, we can use the notation V_i^π to denote the value function of the policy π in the MDP when the reward function is $R = \phi_i$. We can prove that for any policy π, the value function can be defined as $V^\pi(s_1) = \alpha_1 V_1^\pi + \alpha_2 V_2^\pi + ... + \alpha_d V_d^\pi$

We can substitute the reward function into the expectation in the value function equation and use the linearity of expectation to expand it to a sum of expectations of fixed and known values. There expectations would simplify to just the terms. This is shown below:

$$V^\pi(s_1) = E[R(s_1) + \gamma R(s_2) + \gamma^2 R(s_3) + ...|\pi]$$
$$V^\pi(s_1) = E[\alpha_1\phi_1(s_1) + \alpha_2\phi_2(s_1) + ... + \alpha_d\phi_d(s_1)$$
$$+\gamma * (\alpha_2\phi_1(s_2) + \alpha_2\phi_2(s_2) + ... + \alpha_d\phi_d(s_2)) +$$
$$\gamma^2 * (\alpha_1\phi_1(s_3) + \alpha_2\phi_2(s_3) + ... + \alpha_d\phi_d(s_3)) + ...|\pi]$$

$$V^\pi(s_1) = \alpha_1\phi_1(s_1) + \alpha_2\phi_2(s_1) + ... + \alpha_d\phi_d(s_1)$$
$$+\gamma * (\alpha_2\phi_1(s_2) + \alpha_2\phi_2(s_2) + ... + \alpha_d\phi_d(s_2)) +$$
$$\gamma^2 * (\alpha_1\phi_1(s_3) + \alpha_2\phi_2(s_3) + ... + \alpha_d\phi_d(s_3)) + ...$$

The terms can now be regrouped using the alpha terms.

$$V^\pi(s_1) = \alpha_1 * (\phi_1(s_1) + \gamma\phi_1(s_2) + \gamma^2\phi_1(s_3) + ...)$$
$$+\alpha_2 * (\phi_2(s_1) + \gamma\phi_2(s_2) + \gamma^2\phi_2(s_3) + ...)$$
$$+...$$
$$+\alpha_d * (\phi_d(s_1) + \gamma\phi_d(s_2) + \gamma\phi_d(s_3) + ...)$$

$$V^\pi(s_1) = \alpha_1 * E[\phi_1(s_1) + \gamma\phi_1(s_2) + \gamma^2\phi_1(s_3) + ...|\pi]$$
$$+\alpha_2 * E[\phi_2(s_1) + \gamma\phi_2(s_2) + \gamma^2\phi_2(s_3) + ...|\pi]$$
$$+...$$

$$+\alpha_d * E[\phi_d(s_1) + \gamma\phi_d(s_2) + \gamma^2\phi_d(s_3) + ...|\pi]$$

Using the definition of V_i^π and the equation for the value function, we can simplify the above expression to:

$$V^\pi(s_1) = \alpha_1 V_1^\pi + \alpha_2 V_2^\pi + ... + \alpha_d V_d^\pi$$

Since s_1 is a variable to represent any arbitrary state, we can also just write the above as:

$$V^\pi(s) = \alpha_1 V_1^\pi + \alpha_2 V_2^\pi + ... + \alpha_d V_d^\pi$$

This concludes the proof.

Question 2

Appendix: Relevant Code - tjha_hw5.py

```
# Tejas Jha
# EECS 498 Reinforcement Learning HW 5

import random
import gym
import math
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

class Memory():
    def __init__(self, max_size=1000):
        self.buffer = deque(maxlen=max_size)

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        idx = np.random.choice(np.arange(len(self.buffer)),
                               size=batch_size,
                               replace=True)
        return [self.buffer[ii] for ii in idx]

class DQNCartPoleSolver():
    def __init__(self, n_episodes=2000, C=1, n_win_ticks=195,
        max_env_steps=None, gamma=1.0, epsilon=0.5, epsilon_max=0.5,
        epsilon_min=0.01, epsilon_decay=0.0001, alpha=0.0001, batch_size
        =32, monitor=False, quiet=False):
```

```python
        self.memory = Memory(10000)
        self.env = gym.make('CartPole-v0')
        if monitor: self.env = gym.wrappers.Monitor(self.env, '../data/cartpole-1', force=True)
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_max = epsilon_max
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.alpha = alpha
        self.n_episodes = n_episodes
        self.n_win_ticks = n_win_ticks
        self.batch_size = batch_size
        self.quiet = quiet
        self.C = C
        if max_env_steps is not None: self.env._max_episode_steps = max_env_steps

        # Init model
        self.model = Sequential()
        self.model.add(Dense(64, input_dim=4, activation='relu', use_bias=True))
        self.model.add(Dense(2, activation='linear'))
        self.model.compile(loss='mse', optimizer=Adam(lr=self.alpha))

        self.target_model = Sequential()
        self.target_model.add(Dense(64, input_dim=4, activation='relu', use_bias=True))
        self.target_model.add(Dense(2, activation='relu'))
        self.target_model.compile(loss='mse', optimizer=Adam(lr=self.alpha))
        self.target_model.set_weights(self.model.get_weights())

    def remember(self, state, action, reward, next_state, done):
        self.memory.add((state, action, reward, next_state, done))

    def choose_action(self, state, epsilon):
        return self.env.action_space.sample() if (np.random.random() <= epsilon) else np.argmax(self.model.predict(state))

    def get_epsilon(self, t):
        return self.epsilon_max - min(self.epsilon_decay * t, self.epsilon_max - 0.01)

    def preprocess_state(self, state):
        return np.reshape(state, [1, 4])

    def replay(self, batch_size):
```

```
        x_batch, y_batch = [], []
        minibatch = self.memory.sample(self.batch_size)
        for state, action, reward, next_state, done in minibatch:
            y_target = self.model.predict(state)
            y_target[0][action] = reward if done else reward + self.gamma * np.max(self.target_model.predict(next_state)[0])
            x_batch.append(state[0])
            y_batch.append(y_target[0])

        self.model.fit(np.array(x_batch), np.array(y_batch), batch_size=len(x_batch), verbose=0)

    def run(self):

        #preTrain(self.batch_size)

        scores = deque(maxlen=100)

        store_avg = []

        for e in range(self.n_episodes):
            state = self.preprocess_state(self.env.reset())
            done = False
            i = 0
            total_reward = 0
            while not done:
                action = self.choose_action(state, self.get_epsilon(e))
                next_state, reward, done, _ = self.env.step(action)
                next_state = self.preprocess_state(next_state)
                self.remember(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward
                i += 1

            scores.append(total_reward)
            mean_score = np.mean(scores)

            if mean_score >= self.n_win_ticks and e >= 100:
                if not self.quiet: print('Ran {} episodes. Solved after {} trials    '.format(e, e - 100))
                return e - 100
            if e % 100 == 0 and not self.quiet:
                print('[Episode {}] - Mean survival time over last 100 episodes was {} ticks.'.format(e, mean_score))
                store_avg.append(mean_score)

            self.replay(self.batch_size)

```

```
            if e % self.C == 0:
                self.target_model.set_weights(self.model.get_weights())

        if not self.quiet: print('Did not solve after {} episodes '.format(e))
        return e



if __name__ == '__main__':
    agent = DQNCartPoleSolver()
    agent.run()
```