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)

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
# Homework 5
# 12 December 2018


import gym
```

```
from gym import wrappers
import random
import math

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import Adam
import matplotlib.pyplot as plt

import numpy as np

# hyper parameters
EPISODES = 100  # number of episodes
EPS_START = 0.5  # e-greedy threshold start value
EPS_END = 0.01  # e-greedy threshold end value
EPS_DECAY = 0.0001  # e-greedy threshold decay
GAMMA = 1.0  # Q-learning discount factor
LR = 0.0001  # NN optimizer learning rate
HIDDEN_LAYER = 64  # NN hidden layer size
BATCH_SIZE = 32  # Q-learning batch size


class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []

    def push(self, transition):
        self.memory.append(transition)
        if len(self.memory) > self.capacity:
            del self.memory[0]

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)


class Network(nn.Module):
    def __init__(self):
        nn.Module.__init__(self)
        self.l1 = nn.Linear(4, HIDDEN_LAYER)
        self.l2 = nn.Linear(HIDDEN_LAYER, 2)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = self.l2(x)
```

```python
            return x


env = gym.make('CartPole-v0')
env = wrappers.Monitor(env, './tmp/cartpole-v0-1')

model = Network()
if use_cuda:
    model.cuda()
memory = ReplayMemory(10000)
optimizer = optim.Adam(model.parameters(), LR)
steps_done = 0
episode_durations = []


def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. *
        steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        return model(Variable(state, volatile=True).type(FloatTensor)).
            data.max(1)[1].view(1, 1)
    else:
        return LongTensor([[random.randrange(2)]])


def run_episode(e, environment):
    state = environment.reset()
    steps = 0
    while True:
        environment.render()
        action = select_action(FloatTensor([state]))
        next_state, reward, done, _ = environment.step(action[0, 0])

        # negative reward when attempt ends
        if done:
            reward = -1

        memory.push((FloatTensor([state]),
                     action,  # action is already a tensor
                     FloatTensor([next_state]),
                     FloatTensor([reward])))

        learn()

        state = next_state
```

```
        steps += 1

        if done:
            print("{2} Episode {0} finished after {1} steps"
                  .format(e, steps, '\033[92m' if steps >= 195 else '\033[99m'))
            episode_durations.append(steps)
            plot_durations()
            break


def learn():
    if len(memory) < BATCH_SIZE:
        return

    # random transition batch is taken from experience replay memory
    transitions = memory.sample(BATCH_SIZE)
    batch_state, batch_action, batch_next_state, batch_reward = zip(*transitions)

    batch_state = Variable(torch.cat(batch_state))
    batch_action = Variable(torch.cat(batch_action))
    batch_reward = Variable(torch.cat(batch_reward))
    batch_next_state = Variable(torch.cat(batch_next_state))

    # current Q values are estimated by NN for all actions
    current_q_values = model(batch_state).gather(1, batch_action)
    # expected Q values are estimated from actions which gives maximum Q value
    max_next_q_values = model(batch_next_state).detach().max(1)[0]
    expected_q_values = batch_reward + (GAMMA * max_next_q_values)

    # loss is measured from error between current and newly expected Q values
    loss = F.smooth_l1_loss(current_q_values, expected_q_values)

    # backpropagation of loss to NN
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()


def plot_durations():
    plt.figure(2)
    plt.clf()
    durations_t = torch.FloatTensor(episode_durations)
    plt.title('Training...')
    plt.xlabel('Episode')
```

```python
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated


for e in range(EPISODES):
    run_episode(e, env)

print('Complete')
env.render(close=True)
env.close()
plt.ioff()
plt.show()
```