EECS 498: Reinforcement Learning
Homework 3 Responses

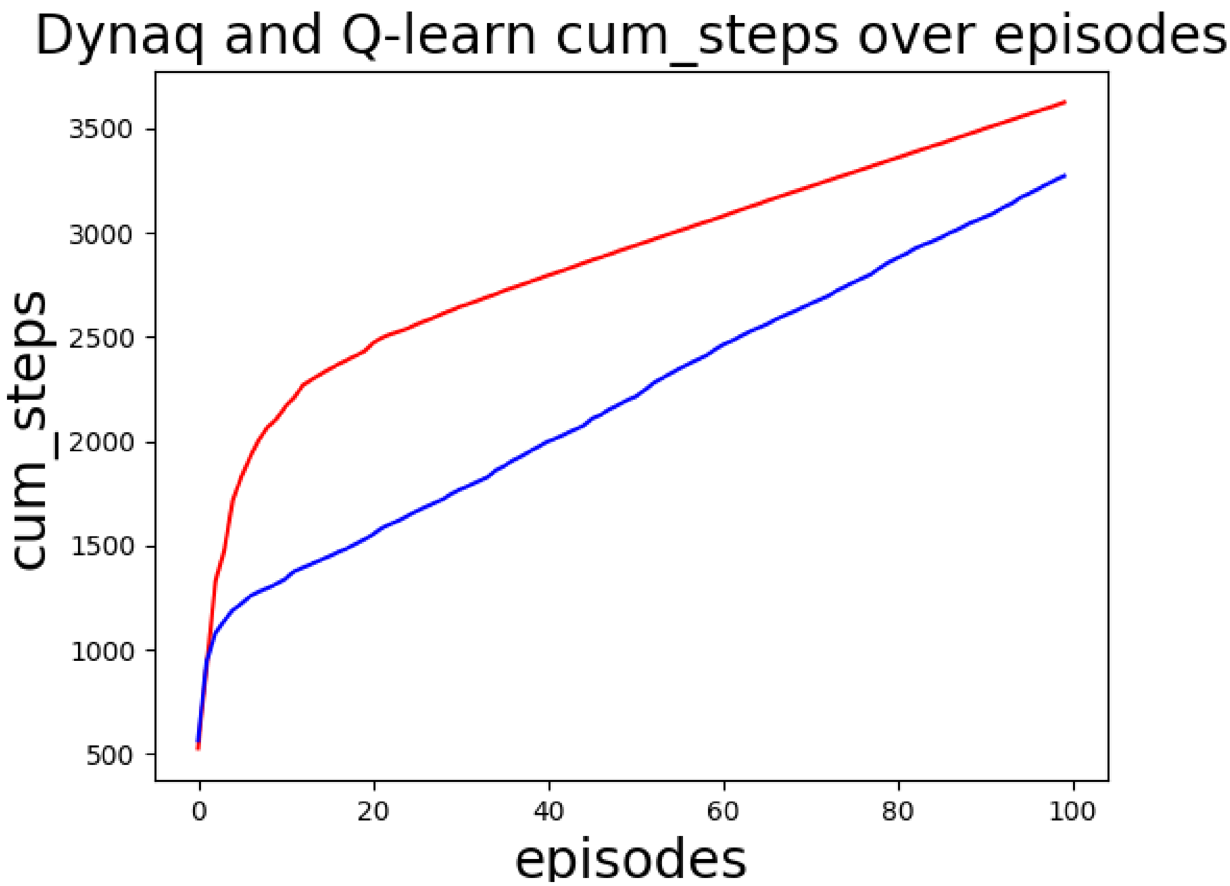
Tejas Jha
tjha

November 5, 2018

This document includes my responses to Homework 3 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.

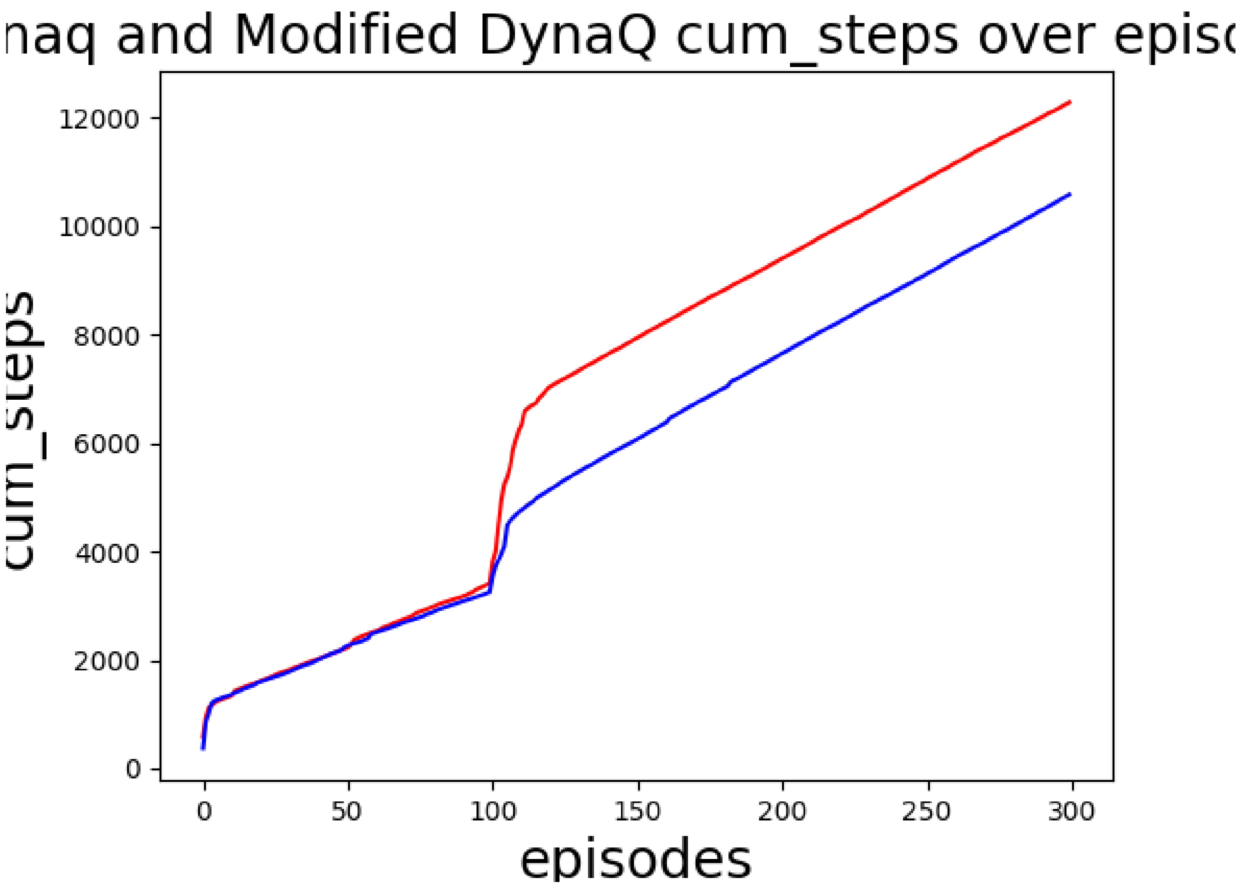
Question 1

(a) The dynaq algorithm was implemented following the pseudocode shown on page 164 of the textbook. In the below plot, the red curve corresponds to the use the Q-learning algorithm and the blue curve represents the use of dynaq algorithm. As can be seen by the images, there curve for dynaq appears to apprach a constant slope earlier than the curve for Q-learning, indicating a faster approach towards an optimal policy with the optimal number of steps required in an episode to reach the terminal state.



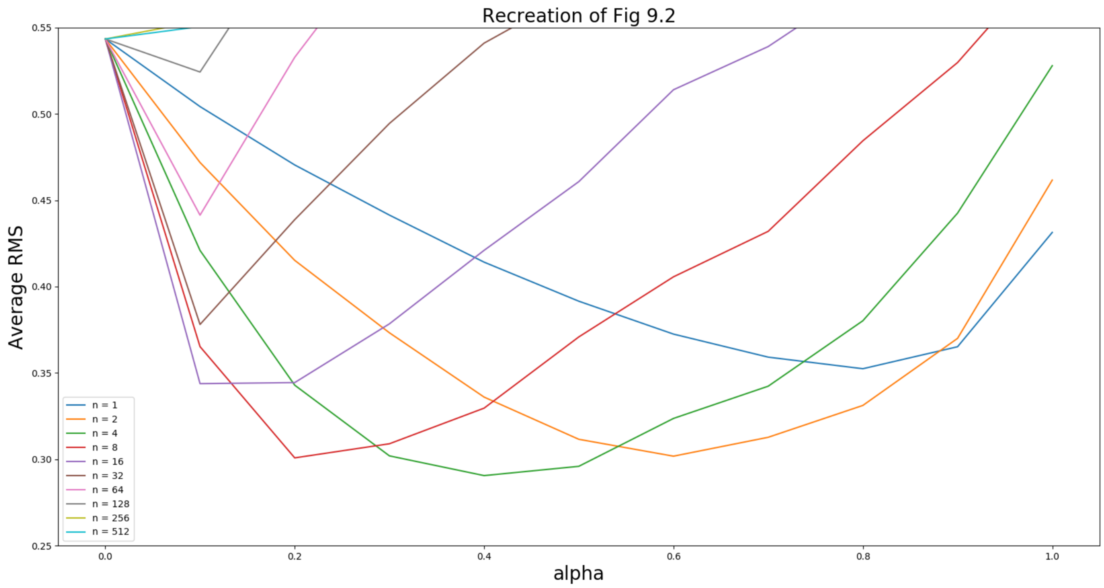
(b) In the case where the environment switches from "Taxi-v4" to "Taxi-v5" after the initial 100 episodes, the length of each episode drastically increased as the algorithm needed more iterations to adjust the model and action-values to the new environment. As the algorithm was able to iterate and better learn the new environment, the length of each episode begins to converge to a constant as represented by a straight line. In order to better account for the change in the environment, I have modified the use of the algorithm to use a lower discount rate (γ). In my case, I used a discout factor of 0.5, which would help in a faster adjustment to the new environment by reducing the weight placed on already learned Q values. As a result, the algorithm is affected less by future expected returns and more by the immediate rewards, which greatly boosts immediate learning to a changing environment. Note that a similar change can be seen by decreasing alpha, the step size, as well as a combination of both.

The plot below displays this difference with the two curves. The red curve is the original algorithm prom part (a) reacting to the change in the environment. The blue curve represents the modified algorithm as described above.



Question 2

Below is the plot produced in an attempt to reproduce the right plot in Figure 9.2 of the textbook.



Appendix: Relevant Code - tjha.py

```
# Tejas Jha
# 5 November 2018
# EECS 498 - Reinforcement Learning HW 3
#
# ------------------------------------------------------------------------------------------
# The code below implements a modified version of "Tabular Dyna-Q" as well as
# implementation modifications to adapt to changes in the environemnt faster.
# This file also contains the code used to implement n-step semi-gradient TD estimation
#
# The functions below expand on the starter code provided that help generate the
# states, actions, and rewards using Taxi-v4 and Taxi-v5 on openAi gym
# ------------------------------------------------------------------------------------------

import numpy as np
import gym
import copy
import mytaxi
```

```python
import math
import matplotlib.pyplot as plt

# Environment for Taxi-v4
ENV4 = gym.make('Taxi-v4').unwrapped
# Einvironment for Taxi-v5
ENV5 = gym.make('Taxi-v5').unwrapped
# Possible actions that can be taken
ACTIONS = [0,1,2,3,4,5]

# Helpers to choose best action given probability distributions
def _greedy(Q,s):
    qmax = np.max(Q[s])
    actions = []
    for i,q in enumerate(Q[s]):
        if q == qmax:
            actions.append(i)
    return actions

def greedy(Q,s):
    return np.random.choice(_greedy(Q,s))

def ep_greedy(Q,s,ep):
    if np.random.rand() < ep:
        return np.random.choice(len(Q[s]))
    else:
        return greedy(Q,s)

# Part (a) - Tabular Dyna-Q:
# Function implementation of dynaq to handle the stochastic nature of the environment
# returns Q and cum_steps - list of the cumulatative number of steps counted from the
# first episode to the end of each episode.
def dynaq(env,n=10,gamma=1,alpha=1,epsilon=0.1,runs=1,episodes=100):
    for run in range(runs):
        # Update kept on cumulative number of steps
        cum_steps = np.zeros(episodes)
        Q = np.zeros((env.nS,env.nA))
        # Using deterministic model
        Model = {}
        for s in range(env.nS):
            Model[s] = {}
            for a in range(env.nA):
                Model[s][a] = (-1, s)
        # Loop over episodes
        for idx in range(episodes):
            visited_states = []
```

```
            taken_actions = {}
            s = env.reset()
            visited_states.append(s)
            done = False
            counter = 0
            while not done:
                a = ep_greedy(Q,s,epsilon)
                if s in taken_actions:
                    if a not in taken_actions[s]:
                        taken_actions[s].append(a)
                else:
                    taken_actions[s] = []
                    taken_actions[s].append(a)
                ss, r, done, _ = env.step(a)
                Q[s,a] = Q[s,a] + alpha * (r + gamma * np.max(Q[ss]) - Q[s,a])
                Model[s][a] = (r, ss)
                s = ss
                for i in range(n):
                    rand_s = np.random.choice(visited_states , size=1)
                    rand_s = rand_s[0]
                    rand_a = np.random.choice(taken_actions[rand_s], size=1)
                    rand_a = rand_a[0]
                    tup = Model[rand_s][rand_a]
                    Q[rand_s,rand_a] = Q[rand_s,rand_a] + alpha * (tup[0] + gamma * np.max(Q[tup[1]]) - Q[rand_s, rand_a])
                counter += 1
                visited_states.append(s)
            if idx == 0:
                cum_steps[idx] = counter
            else:
                cum_steps[idx] = counter + cum_steps[idx - 1]
    return Q, cum_steps

# Part (a) - adaptation of qlearn from hw2 to compare with Tabular Dyna-Q
# Key difference is 100 episodes instead of 500 by default
# Also, steps are now everaged through multiple callings of function
def qlearn(env,gamma=1,alpha=0.9,ep=0.05,runs=1,episodes=100):
    #np.random.seed(3)
    #env.seed(5)
    nS = env.nS
    nA = env.nA
    rew_alloc = []
    for run in range(runs):
        Q = np.zeros((nS,nA))
```

```python
        rew_list = np.zeros(episodes)
        cum_steps = np.zeros(episodes)
        for idx in range(episodes):
            s = env.reset()
            done = False
            counter = 0
            cum_rew = 0
            while not done:
                a = ep_greedy(Q,s,ep)
                ss, r, done, _ = env.step(a)
                Q[s,a] = Q[s,a] + alpha * (r + gamma * np.max(Q[ss]) - Q[s,a])
                s = ss
                cum_rew +=  gamma**counter * r
                counter += 1.
            rew_list[idx] = cum_rew
            if idx == 0:
                cum_steps[idx] = counter
            else:
                cum_steps[idx] = counter + cum_steps[idx - 1]
        rew_alloc.append(rew_list)
    rew_list = np.mean(np.array(rew_alloc),axis=0)
    return Q, cum_steps

# Modified versions of the algorithms above for usage in random change to v5 environment after 100 episodes
def original_dynaq(env1, env2, n=10,gamma=1,alpha=1,epsilon=0.1,runs=1,episodes=300):
    for run in range(runs):
        # Update kept on cumulative number of steps
        cum_steps = np.zeros(episodes)
        Q = np.zeros((env1.nS,env1.nA))
        # Using deterministic model
        Model = {}
        for s in range(env1.nS):
            Model[s] = {}
            for a in range(env1.nA):
                Model[s][a] = (-1, s)
        env = env1
        # Loop over episodes
        for idx in range(episodes):
            if idx == 100:
                env = env2
            visited_states = []
            taken_actions = {}
            s = env.reset()
            visited_states.append(s)
            done = False
```

```
            counter = 0
            while not done:
                a = ep_greedy(Q,s,epsilon)
                if s in taken_actions:
                    if a not in taken_actions[s]:
                        taken_actions[s].append(a)
                else:
                    taken_actions[s] = []
                    taken_actions[s].append(a)
                ss, r, done, _ = env.step(a)
                Q[s,a] = Q[s,a] + alpha * (r + gamma * np.max(Q[ss]) - Q[s,a])
                Model[s][a] = (r, ss)
                s = ss
                for i in range(n):
                    rand_s = np.random.choice(visited_states, size=1)
                    rand_s = rand_s[0]
                    rand_a = np.random.choice(taken_actions[rand_s], size=1)
                    rand_a = rand_a[0]
                    tup = Model[rand_s][rand_a]
                    Q[rand_s,rand_a] = Q[rand_s,rand_a] + alpha * (tup[0] + gamma * np.max(Q[tup[1]]) - Q[rand_s,rand_a])
                counter += 1
                visited_states.append(s)
            if idx == 0:
                cum_steps[idx] = counter
            else:
                cum_steps[idx] = counter + cum_steps[idx - 1]
    return Q, cum_steps

# Modified improvement for dynaq to account for environment change
def modified_dynaq(env1, env2, n=10,gamma=0.5,alpha=1,epsilon=0.1,runs=1,episodes=300):
    for run in range(runs):
        # Update kept on cumulative number of steps
        cum_steps = np.zeros(episodes)
        Q = np.zeros((env1.nS,env1.nA))
        # Using deterministic model
        Model = {}
        for s in range(env1.nS):
            Model[s] = {}
            for a in range(env1.nA):
                Model[s][a] = (-1, s)
        env = env1
        # Loop over episodes
        for idx in range(episodes):
```

```
            if idx == 100:
                env = env2
            visited_states = []
            taken_actions = {}
            s = env.reset()
            visited_states.append(s)
            done = False
            counter = 0
            while not done:
                a = ep_greedy(Q,s,epsilon)
                if s in taken_actions:
                    if a not in taken_actions[s]:
                        taken_actions[s].append(a)
                else:
                    taken_actions[s] = []
                    taken_actions[s].append(a)
                ss, r, done, _ = env.step(a)
                Q[s,a] = Q[s,a] + alpha * (r + gamma * np.max(Q[ss]) - Q[s,a])
                Model[s][a] = (r, ss)
                s = ss
                for i in range(n):
                    rand_s = np.random.choice(visited_states, size=1)
                    rand_s = rand_s[0]
                    rand_a = np.random.choice(taken_actions[rand_s], size=1)
                    rand_a = rand_a[0]
                    tup = Model[rand_s][rand_a]
                    Q[rand_s,rand_a] = Q[rand_s,rand_a] + alpha * (tup[0] + gamma * np.max(Q[tup[1]]) - Q[rand_s,rand_a])
                counter += 1
                visited_states.append(s)
            if idx == 0:
                cum_steps[idx] = counter
            else:
                cum_steps[idx] = counter + cum_steps[idx - 1]
    return Q, cum_steps

class ValueFunction:
    def __init__(self, num_of_groups):
        self.num_of_groups = num_of_groups
        self.group_size = 1000 // num_of_groups
        self.params = np.zeros(num_of_groups)

    def value(self, state):
        if state in [0,1001]:
            return 0
```

```
            group_index = (state - 1) // self.group_size
            return self.params[group_index]

    def update(self, delta, state):
        group_index = (state - 1) // self.group_size
        self.params[group_index] += delta

def get_action():
    if np.random.binomial(1, 0.5) == 1:
        return 1
    return -1

def step(state, action):
    step = np.random.randint(1, 100 + 1)
    step *= action
    state += step
    state = max(min(state, 1001), 0)
    if state == 0:
        reward = -1
    elif state == 1001:
        reward = 1
    else:
        reward = 0
    return state, reward

# other
def randomWalk2(value_function, n, alpha):
    state = 500
    states = [state]
    rewards = [0]
    time = 0

    # the length of this episode
    T = float('inf')
    while True:
        # go to next time step
        time += 1

        if time < T:
            # choose an action randomly
            action = get_action()
            next_state, reward = step(state, action)

            # store new state and new reward
            states.append(next_state)
            rewards.append(reward)

            if next_state in [0, 1001]:
```

```
                T = time

        # get the time of the state to update
        update_time = time - n
        if update_time >= 0:
            returns = 0.0
            # calculate corresponding rewards
            for t in range(update_time + 1, min(T, update_time + n) + 1):
                returns += rewards[t]
            # add state value to the return
            if update_time + n <= T:
                returns += value_function.value(states[update_time + n])
            state_to_update = states[update_time]
            # update the value function
            if not state_to_update in [0,1001]:
                delta = alpha * (returns - value_function.value(state_to_update))
                value_function.update(delta, state_to_update)
        if update_time == T - 1:
            break
        state = next_state


if __name__ == '__main__':

    # Part (a)
    print("Training using Tabular Dyna-Q for 100 episodes using Taxi-v4")
    # Average results over 20 runs
    dynaq_Q_avg = 0
    qlearn_Q_avg = 0

    dynaq_cum_steps_avg = np.zeros(shape=(100,))
    qlearn_cum_steps_avg = np.zeros(shape=(100,))

    for i in range(20):
        print("Performing run: " + str(i + 1))
        # Randomize seeds so runs are independent
        np.random.seed(i)
        ENV4.seed(i)
        _, dynaq_cum_steps = dynaq(ENV4)
        _, qlearn_cum_steps = qlearn(ENV4)
        # Update averages
        #dynaq_Q_avg += dynaq_Q / 20.0
        #qlearn_Q_avg += qlearn_Q / 20.0
        dynaq_cum_steps_avg += np.divide(dynaq_cum_steps, 20.0)
```

```python
            qlearn_cum_steps_avg += np.divide(qlearn_cum_steps, 20.0)

        # Compare results with Q learning implementation used in hw2 in plot
        # Generate plots for Question 1 Part(a)
        episodes = np.arange(len(qlearn_cum_steps_avg))
        plt.plot(episodes,qlearn_cum_steps_avg, 'r')
        plt.plot(episodes, dynaq_cum_steps_avg, 'b')
        plt.xlabel("episodes", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
        plt.ylabel("cum_steps", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
        plt.title("Dynaq and Q-learn cum_steps vs. episodes", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
        plt.savefig("Figure1")


        dynaq_cum_steps_avg1 = np.zeros(shape=(300,))
        dynaq_cum_steps_avg2 = np.zeros(shape=(300,))
        for i in range(5):
            print("Performing run: " + str(i + 1))
            # Randomize seeds so runs are independent
            np.random.seed(i)
            ENV4.seed(i)
            ENV5.seed(i)
            _, dynaq_cum_steps1 = original_dynaq(ENV4, ENV5)
            _, dynaq_cum_steps2 = modified_dynaq(ENV4, ENV5)
            # Update averages
            #dynaq_Q_avg += dynaq_Q / 20.0
            #qlearn_Q_avg += qlearn_Q / 20.0
            dynaq_cum_steps_avg1 += np.divide(dynaq_cum_steps1, 5.0)
            dynaq_cum_steps_avg2 += np.divide(dynaq_cum_steps2, 5.0)

        episodes = np.arange(len(dynaq_cum_steps_avg1))
        plt.plot(episodes,dynaq_cum_steps_avg1, 'r')
        plt.plot(episodes, dynaq_cum_steps_avg2, 'b')
        plt.xlabel("episodes", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
        plt.ylabel("cum_steps", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
        plt.title("Original/Modified DynaQ cum_steps vs. episodes", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
        plt.savefig("Figure2")

        plt.figure(figsize=(20,10))
        alphas = np.arange(0,1.1, 0.1)
        ns = np.power(2,np.arange(0,10))
        tsv = np.load('trueStateValue.npy')
```

```

    print(ns)
    print(alphas)

    # # dynaq_cum_steps_avg = np.zeros(shape=(100,))
    # # episodes = np.arange(len(dynaq_cum_steps_avg))
    # # for n in ns:
    # #     print(n)
    # #     lab = "n=" + str(n)
    # #     errorset = []
    # #     for alph in alpha:
    # #         print(alph)
    # #         rms = randomWalk(alph, n, tsv)
    # #         errorset.append(rms)
    # #     plt.plot(alpha, errorset, label=lab)
    # #     break
    # # plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    # # plt.savefig("FigureX")


    #track the errors for each (step, alpha) combination
    errors = np.zeros((len(ns), len(alphas)))
    for run in range(100):
        for step_ind, n in zip(range(len(ns)), ns):
            for alpha_ind, alpha in zip(range(len(alphas)), alphas):
                # we have 20 aggregations in this example
                value_function = ValueFunction(20)
                for ep in range(0, 10):
                    randomWalk2(value_function, n, alpha)
                    # calculate the RMS error
                    state_value = np.asarray([value_function.value(i) for i in np.arange(1, 1001)])
                    errors[step_ind, alpha_ind] += np.sqrt(np.sum(np.power(state_value - tsv[1: -1], 2)) / 1000)
        print(run)
    # take average
    errors /= 10 * 100
    # truncate the error
    for i in range(len(ns)):
        plt.plot(alphas, errors[i, :], label='n = ' + str(ns[i]))
    plt.xlabel("alpha", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
    plt.ylabel("Average RMS", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
    plt.title("Recreation of Fig 9.2", fontdict={'fontname':'DejaVu Sans', 'size':'20'})
    plt.ylim([0.25, 0.55])
    plt.legend()
```

```
plt.savefig("FigureY")
```