MITx - 686x_Project_5_TextBasedGame

September 8, 2019

0.1 Project 5: Text-Based Game

[1]:
```
import numpy as np
from IPython.display import Image
```

0.2 1. Introduction

In this project, we address the task of learning control policies for text-based games using reinforcement learning. In these games, all interactions between players and the virtual world are through text. The current world state is described by elaborate text, and the underlying state is not directly observable. Players read descriptions of the state and respond with natural language commands to take actions.

For this project you will conduct experiments on a small Home World, which mimic the environment of a typical house.The world consists of a few rooms, and each room contains a representative object that the player can interact with. For instance, the kitchen has an apple that the player can eat. The goal of the player is to finish some quest. An example of a quest given to the player in text is You are hungry now . To complete this quest, the player has to navigate through the house to reach the kitchen and eat the apple. In this game, the room is hidden from the player, who only receives a description of the underlying room. At each step, the player read the text describing the current room and the quest, and respond with some command (e.g., eat apple). The player then receives some reward that depends on the state and his/her command.

In order to design an autonomous game player, we will employ a reinforcement learning framework to learn command policies using game rewards as feedback. Since the state observable to the player is described in text, we have to choose a mechanism that maps text descriptions into vector representations. A naive approach is to create a map that assigns a unique index for each text description. However, such approach becomes difficult to implement when the number of textual state descriptions are huge. An alternative method is to use a bag-of-words representation derived from the text description. This project requires you to complete the following tasks:

- Implement the tabular Q-learning algorithm for a simple setting where each text description is associated with a unique index.
- Implement the Q-learning algorithm with linear approximation architecture, using bag-of-words representation for textual state description.
- Implement a deep Q-network.
- Use your Q-learning algorithms on the Home World game.

Setup: As with the previous projects, please use Python's NumPy numerical library for handling arrays and array operations; use matplotlib for producing figures and plots. 1. Note on software: For the all the projects, we will use python 3.6 augmented with the NumPy numerical toolbox, the matplotlib plotting toolbox. For THIS project, you will also be using PyTorch for implementing Neural Nets 2. Download rl.tar.gz and untar it in to a working directory. The archive contains various data files, along with the following python files: * agent_tabular.py where you will implement an agent using tabular Q-learning * agent_linear.py where you will implement an agent using Q-learning with linear approximation * agent_dqn.py where you will implement an agent using a deep Q-network

Tip: Throughout the whole online grading system, you can assume the NumPy python library is already imported as np. In some problems you will also have access to python's random library, and other functions you've already implemented. Look out for the "Available Functions" Tip before the codebox, as you did in the last project.

This project will unfold both on MITx and on your local machine. However, we encourage you to first implement the functions locally and run the test scripts to validate basic functionality. Think of the online graders as a submission box to submit your code when it is ready. You should not have to use the online graders to debug your code. A good strategy for this project is to first implement all the functions from tab 3 and 4 to check for acceptable performance before submitting your code online.

0.3 2. Home World Game

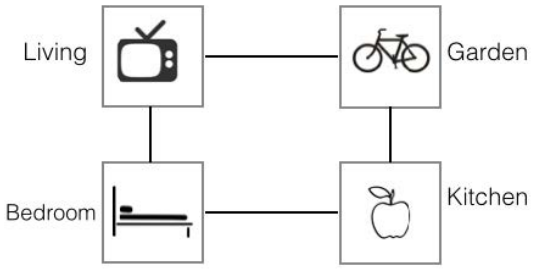
In this project, we will consider a text-based game represented by the tuple $< H, C, P, R, \gamma, \Psi >$. Here H is the set of all possible game states. The actions taken by the player are multi-word natural language commands such as eat apple or go east . In this project we limit ourselves to consider commands consisting of one action (e.g., eat) and one argument object (e.g. apple).

- H is the set of all possible game states.
- S denote the space of all possible text descriptions. The text descriptions s observed by the player are produced by a stochastic function $\Psi : H \rightarrow S$. Assume that each observable state $s \in S$ is associated a unique hidden state, denoted by $h(s) \in H$.
- $C = \{(a, b)\}$ is the set of all commands (action-object pairs).
- $R : H\ x\ C \rightarrow R$ is the deterministic reward function. $R(h, a, b)$ is the immediate reward the player obtains when taking command (a, b) in state h. We consider discounted accumulated rewards where γ is the discount factor. In particular, the game state h is hidden from the player, who only receives a varying textual description.
- $P : H\ x\ C\ x\ H \rightarrow [0, 1]$ is the transition matrix: $P(h'|h, a, b)$ is the probability of reaching state h' if command $c = (a, b)$ is taken in state h.

You will conduct experiments on a small Home World, which mimic the environment of a typical house. The world consists of four rooms- a living room, a bed room, a kitchen and a garden with connecting pathways (illustrated in figure below). Transitions between the rooms are deterministic. Each room contains a representative object that the player can interact with. For instance, the living room has a TV that the player can watch , and the kitchen has an apple that the player can eat. Each room has several descriptions, invoked randomly on each visit by the player.

[2]:
```
Image(filename='C:/root/workspace/MITx_6.86x/07_Projects/rl/images_homeworld.
↪jpg')
```

[2]:



Reward Structure

Positive	Negative
Quest goal: +1	Negative per step: 0.01
.	Invalid command: 0.1

At the beginning of each episode, the player is placed at a random room and provided with a randomly selected quest. An example of a quest given to the player in text is You are hungry now. To complete this quest, the player has to navigate through the house to reach the kitchen and eat the apple (i.e., type in command eat apple). In this game, the room is hidden from the player, who only receives a description of the underlying room.

The underlying game state is given by $h = (r, q)$, where r is the index of room and q is the index of quest.

At each step, the text description s provided to the player contains two part $s = (s_r, s_q)$, where s_r is the room description (which are varied and randomly provided) and s_q is the quest description. The player receives a positive reward on completing a quest, and negative rewards for invalid command (e.g., eat TV). Each non-terminating step incurs a small deterministic negative rewards, which incentives the player to learn policies that solve quests in fewer steps. (see the Table 1) An episode ends when the player finishes the quest or has taken more steps than a fixed maximum number of steps.

Each episode produces a full record of interaction

$$(h_0, s_0, a_0, b_0, r_0, \dots h_t, s_t, a_t, b_t, r_t, h_{t+1}, \dots)$$

where

- $h_0 = (h_{r,0}, h_{q,0}) \sim \Gamma_0$ (Γ_0 denotes an initial state distribution),
- $h_t \sim P(|h_{t-1}, a_{t-1}, b_{t-1})$
- $s_t \sim \Psi(h_t)$
- $r_t = R(h_t, a_t, b_t)$

and all commands (a_t, b_t) are chosen by the player. The record of interaction observed by the player is $(s_0, a_0, b_0, r_0, \dots, s_t, a_t, b_t, r_t, \dots)$.

Within each episode, the quest remains unchanged, i.e., $h_{q,t} = h_{q,0}$ (so as the quest description $s_{q,t} = s_{q,0}$). When the player finishes the quest at time K, all rewards after time K are assumed to be zero, i.e., $r_t = 0$ for $t > K$.

Over the course of the episode, the total discounted reward obtained by the player is $\sum_{t=0}^{\infty} \gamma^t r_t$

We emphasize that the hidden state $h_0, \ldots, h_T$ are unobservable to the player.

The learning goal of the player is to find a policy that $\pi : S \to C$ that maximizes the expected cumulative discounted reward

$$E[\sum_{t=0}^{\infty} \gamma^t R(h_t, a_t, b_t) | (a_t, b_t) \sim \pi]$$

where the expectation accounts for all randomness in the model and the player. Let $\pi^\star$ denote the optimal policy. For each observable state $s \in S$, let $h(s)$ be the associated hidden state. The optimal expected reward achievable is defined as

$$V^\star = E_{h \sim \Gamma_0, s \sim \Psi(h)}[V^\star(s)]$$

where Γ_0 denotes an initial state distribution, $\Psi : H \to S$ produces the text descriptions s observed by the player. Assume that each observable state $s \in S$ is associated a unique hidden state, denoted by $h(s) \in H$.

$$V^\star(s) = max E[\sum_{t=0}^{\infty} \gamma^t R(h_t, a_t, b_t) | h_0 = h(s), s_0 = s, (a_t, b_t) \sim \pi]$$

We can define the optimal Q-function as

$$Q^\star(s, a, b) = max E[\sum_{t=0}^{\infty} \gamma^t R(h_t, a_t, b_t) | h_0 = h(s), s_0 = s, a_0 = a, b_0 = b, (a_t, b_t) \sim \pi^\star \; for \; t \geq 1]$$

Note that given $Q^\star(s, a, b)$, we can obtain an optimal policy:

$$\pi^\star(s) = max_{(a,b) \in C} Q^\star(s, a, b)$$

The commands set C contain all (action, object) pairs.

Note that some commands are invalid. For instance, (eat,TV) is invalid for any state, and (eat, apple) is valid only when the player is in the kitchen (i.e., h_r corresponds to the index of kitchen). When an invalid command is taken, the system state remains unchanged and a negative reward is incurred. Recall that there are four rooms in this game. Assume that there are four quests in this game, each of which would be finished only if the player takes a particular command in a particular room. For example, the quest "You are sleepy" requires the player navigates through rooms to bedroom (with commands such as go east/west/south/north) and then take a nap on the bed there. For each room, there is a corresponding quest that can be finished there.

Note that in this game, the transition between states is deterministic. Since the player is placed at a random room and provided a randomly selected quest at the beginning of each episode, the distribution Γ_0 of the initial state h_0 is uniform over the hidden state space H.

0.3.1 Optimal episodic reward

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

[4]:
```
DEBUG = False
DEFAULT_REWARD = -0.01 # Negative reward for each non-terminal step
```

```
JUNK_CMD_REWARD = -0.1 # Negative reward for invalid commands
QUEST_REWARD = 1 # positive reward for finishing quest
STEP_COUNT = 0  #count the number of steps in current episode
MAX_STEPS = 20
```

[5]:
```
rooms = ['Living', 'Garden', 'Kitchen','Bedroom']
living_desc = ['This room has a couch, chairs and TV.',
          'You have entered the living room. You can watch TV here.',
          'This room has two sofas, chairs and a chandelier.',
          'A huge television that is great for watching games.']
garden_desc = ['This space has a swing, flowers and trees.',
          'You have arrived at the garden. You can exercise here',
          'This area has plants, grass and rabbits.',
          'A nice shiny bike that is fun to ride.',]
kitchen_desc = ['This room has a fridge, oven, and a sink.',
           'You have arrived in the kitchen. You can find food and drinks here.
 ↪',
           'This living area has pizza, coke, and icecream.',
           'A red juicy fruit.']
bedroom_desc = ['This area has a bed, desk and a dresser.',
           'You have arrived in the bedroom. You can rest here.',
           'You see a wooden cot and a mattress on top of it.',
           'A nice, comfortable bed with pillows and sheets.']
rooms_desc = {'Living': living_desc, 'Garden': garden_desc, 'Kitchen':␣
 ↪kitchen_desc, 'Bedroom': bedroom_desc}
rooms_desc_map = {}
```

[6]:
```
# --Simple quests
quests = ['You are bored.', 'You are getting fat.', 'You are hungry.','You are␣
 ↪sleepy.']
quests_map = {}
# --(somewhat) complex quests
# -- quests = {'You are not sleepy but hungry.',
# --                      'You are not hungry but sleepy.',
# --                      'You are not getting fat but bored.',
# --                      'You are not bored but getting fat.'}
```

[7]:
```
quest_actions = ['watch', 'exercise', 'eat', 'sleep'] #aligned to quests above
quest_objects = ['tv', 'bike', 'apple', 'bed'] #aligned to quest actions above

actions = ['eat', 'sleep', 'watch', 'exercise', 'go']
objects = ['apple', 'bed', 'tv', 'bike', 'north','south','east','west']
```

[8]:
```
living_valid_act = ['go', 'go', 'watch']
living_valid_obj = ['south', 'west', 'tv']
living_transit = ['Bedroom', 'Garden', 'Living']

garden_valid_act = ['go', 'go', 'exercise']
garden_valid_obj = ['south', 'east', 'bike']
```

```
garden_transit = ['Kitchen', 'Living', 'Garden']

kitchen_valid_act = ['go', 'go', 'eat']
kitchen_valid_obj = ['north', 'east', 'apple']
kitchen_transit = ['Garden', 'Bedroom', 'Kitchen']

bedroom_valid_act = ['go', 'go', 'sleep']
bedroom_valid_obj = ['north', 'west', 'bed']
bedroom_transit = ['Living', 'Kitchen', 'Bedroom']

rooms_valid_acts = {'Living': living_valid_act, 'Garden': garden_valid_act,␣
 ↪'Kitchen': kitchen_valid_act, 'Bedroom': bedroom_valid_act}
rooms_valid_objs = {'Living': living_valid_obj, 'Garden': garden_valid_obj,␣
 ↪'Kitchen': kitchen_valid_obj, 'Bedroom': bedroom_valid_obj}
rooms_transit = {'Living': living_transit, 'Garden': garden_transit, 'Kitchen':␣
 ↪kitchen_transit, 'Bedroom': bedroom_transit}
```

[9]:
```
NUM_ROOMS = len(rooms)
NUM_QUESTS = len(quests)
NUM_ACTIONS = len(actions)
NUM_OBJECTS = len(objects)
NUM_ROOMS,NUM_QUESTS,NUM_ACTIONS,NUM_OBJECTS
```

[9]:
```
(4, 4, 5, 8)
```

[10]:
```
command_is_valid = np.zeros((NUM_ROOMS,NUM_ACTIONS,NUM_OBJECTS)) #4,5,8
#command_is_valid
```

[11]:
```
transit_matrix = np.zeros((NUM_ROOMS,NUM_ACTIONS,NUM_OBJECTS,NUM_ROOMS))␣
 ↪#4,5,8,4
#transit_matrix
```

[12]:
```
def text_to_hidden_state_mapping():
    for i in range(NUM_ROOMS):
        room_name = rooms[i]
        for room_desc in rooms_desc[room_name]:
            rooms_desc_map[room_desc] = i

    for i in range(NUM_QUESTS):
        quest_text = quests[i]
        quests_map[quest_text] = i
```

[13]:
```
def load_game_data():
    # each state:(room, quest), where "room" is a hidden state
    # observable state: (room description, quest)

    for room_name in rooms_valid_acts:

        room_index = rooms.index(room_name)
        valid_acts = rooms_valid_acts[room_name]
```

```
        valid_objs = rooms_valid_objs[room_name]
        transit = rooms_transit[room_name]

        for valid_index, act in enumerate(valid_acts):
            obj = valid_objs[valid_index]
            act_index = actions.index(act)
            obj_index = objects.index(obj)
            # valid commands: A(h,(a,o))=1 if (a,o) is valid for hidden state h.
            command_is_valid[room_index, act_index, obj_index] = 1;

            next_room_name = transit[valid_index]
            next_room_index = rooms.index(next_room_name)
            #deterministic transition
            transit_matrix[room_index, act_index, obj_index, next_room_index] =␣
↪1;

    text_to_hidden_state_mapping()
```

[14]:
```
# take a step in the game
def step_game(current_room_desc, current_quest_desc, action_index,␣
↪object_index):
    global STEP_COUNT
    STEP_COUNT = STEP_COUNT+1
    terminal = (STEP_COUNT >= MAX_STEPS)
    #print('Step=%d' %(STEP_COUNT))
    #print(terminal)

    # room_index: the hidden state.
    current_room_index = rooms_desc_map[current_room_desc]
    quest_index = quests_map[current_quest_desc]

    if (command_is_valid[current_room_index, action_index, object_index]==1):
        # quest has been finished
        if ((actions[action_index]==quest_actions[quest_index]) and␣
↪(objects[object_index]==quest_objects[quest_index])):
            terminal = True
            reward = QUEST_REWARD

            if DEBUG:
                print('Finish quest: %s at Room %s with command %s %s'␣
↪%(current_quest_desc, current_room_desc,␣
↪actions[action_index],objects[object_index]))

        else:
            reward = DEFAULT_REWARD

        # probability distribution of next room.
```

```
        next_room_dist = transit_matrix[current_room_index, action_index,␣
↪object_index, :]
        next_room_index = np.random.choice(NUM_ROOMS, p=next_room_dist)
        next_room_name = rooms[next_room_index]
        next_room_desc_index = np.random.
↪randint(len(rooms_desc[next_room_name]))
        next_room_desc = rooms_desc[next_room_name][next_room_desc_index]
        #if DEBUG:
            #print('Reward: %1.3f' % (reward,))
            #print('Transit to Room %d:%s. %s' %(next_room_index,␣
↪rooms[next_room_index],rooms_desc[next_room_name][next_room_desc_index]))

    else:
        # penalty for invalid command
        reward = DEFAULT_REWARD + JUNK_CMD_REWARD
        # state remains the same when invalid command executed
        next_room_desc = current_room_desc

        # if DEBUG:
        #     print('Invalid command!')
        #     print('Reward: %1.3f' % (reward,))
        #     print('Remain in Room %d:%s' %(next_room_index,␣
↪rooms[next_room_index],))

    # quest remains the same during each episode
    next_quest_desc = current_quest_desc
    return (next_room_desc, next_quest_desc, reward, terminal)
```

[15]:
```
# start a new game
def newGame():
    global STEP_COUNT
    STEP_COUNT = 0
    # random initial state: room_index + quest_index
    room_index = np.random.randint(NUM_ROOMS)
    room_name = rooms[room_index]
    room_desc_index = np.random.randint(len(rooms_desc[room_name]))
    room_desc = rooms_desc[room_name][room_desc_index]

    quest_index = np.random.randint(len(quests))
    quest_desc = quests[quest_index]

    terminal = False
    if DEBUG:
        print('Start a new game')
        print('Start Room %d: %s. %s' % (room_index, room_name, room_desc,))
        print('Start quest: %s' % (quest_desc,))
```

```
    return (room_desc, quest_desc, terminal)
```

[16]:
```
def get_actions():
    return (actions)
```

[17]:
```
def get_objects():
    return (objects)
```

[18]:
```
def make_all_states_index():
    """
    Returns tow dictionaries:
    1: one for all unique room descriptions occur in the game
    2: one for all unique quests in the game
    """
    dictionary_room_desc = {}
    dictionary_quest_desc = {}
    for room in rooms_desc:
        for desc in rooms_desc[room]:
            if desc not in dictionary_room_desc:
                dictionary_room_desc[desc] = len(dictionary_room_desc)

    for quest in quests:
        if quest not in dictionary_quest_desc:
            dictionary_quest_desc[quest] = len(dictionary_quest_desc)

    return (dictionary_room_desc, dictionary_quest_desc)
```

[19]:
```
from string import punctuation, digits
import csv
import numpy as np
import matplotlib.pyplot as plt

import sys

if sys.version_info[0] < 3:
    PYTHON3 = False
else:
    PYTHON3 = True

def load_data(path_data):
    """Return a dictionary for the state descriptions displayed to player"""
    global PYTHON3

    data = []
    if PYTHON3:
        f_data = open(path_data, encoding="latin1")
    else:
        f_data = open(path_data)
```

```
    reader = csv.reader(f_data, delimiter='\t')

    for row in reader:
            data.append(row)

    f_data.close()

    return data


def ewma(a, alpha=0.9):
    """Computes the exponentially weighted moving average of a"""
    b = np.array(a)
    n = b.size
    w0 = np.ones(n) * alpha
    p = np.arange(n - 1, -1, -1)
    return np.average(b, weights=w0 ** p)


def extract_words(input_string):
    """
    Helper function for bag_of_words()
    Inputs a text string
    Returns a list of lowercase words in the string.
    Punctuation and digits are separated out into their own words.
    """
    for c in punctuation + digits:
        input_string = input_string.replace(c, ' ' + c + ' ')
    return input_string.lower().split()


def bag_of_words(texts):
    """
    Inputs a list of string descriptions
    Returns a dictionary of unique unigrams occurring over the input
    """
    dictionary = {}  # maps word to unique index
    for text in texts:
        word_list = extract_words(text[0])
        for word in word_list:
            if word not in dictionary:
                dictionary[word] = len(dictionary)
    return dictionary


def extract_bow_feature_vector(state_desc, dictionary):
    """
```

```
    Inputs a string state description
    Inputs the dictionary of words as given by bag_of_words
    Returns the bag-of-words vector representation of the state
    The returned vector is of dimension m, where m the total number of entries␣
↪in the dictionary.
    """
    state_vector = np.zeros([len(dictionary)])
    word_list = extract_words(state_desc)
    for word in word_list:
        if word in dictionary:
            state_vector[dictionary[word]] += 1

    return state_vector
```

3. Q-learning Algorithm

In this section, you will implement the Q-learning algorithm, which is a model-free algorithm used to learn an optimal Q-function. In the tabular setting, the algorithm maintains the Q-value for all possible state-action pairs. Starting from a random Q-function, the agent continuously collects experiences (s,c,R(s,c),s) and updates its Q-function.

From now on, we will refer to c=(a,b) as "an action" although it really is an action with an object.

Q-learning Algorithm

- The agent plays an action c at state s , getting a reward R(s,c) and observing the next state s .
- Update the single Q-value corresponding to each such transition:

$$Q(s,c) \leftarrow (1\alpha)Q(s,c) + \alpha[R(s,a) + \gamma max_{c' \in C'} Q(s',c')]$$

Tip: We recommend you implement all functions from this tab and the next one before submitting your code online. Make sure you achieve reasonable performance on the Home World game

Single step update Write a function tabular_q_learning that updates the single Q-value, aiven the transition date $(s, c, R(s,c), s')$.

Reminder: You should implement this function locally first. You can read through the next tab to understand the context in which this function is called

Available Functions: You have access to the NumPy python library as np. You should also use constants ALPHA and GAMMA in your code

[20]:
```
"""Tabular QL agent"""
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

DEBUG = False

GAMMA = 0.5  # discounted factor
TRAINING_EP = 0.5  # epsilon-greedy parameter for training
TESTING_EP = 0.05  # epsilon-greedy parameter for testing
NUM_RUNS = 10
```

```
NUM_EPOCHS = 200
NUM_EPIS_TRAIN = 25  # number of episodes for training at each epoch
NUM_EPIS_TEST = 50  # number of episodes for testing
ALPHA = 0.1  # learning rate for training

ACTIONS = get_actions()
OBJECTS = get_objects()
NUM_ACTIONS = len(ACTIONS)
NUM_OBJECTS = len(OBJECTS)
```

[21]:
```
# pragma: coderesponse template
def tabular_q_learning(q_func, current_state_1, current_state_2, action_index,
                       object_index, reward, next_state_1, next_state_2,
                       terminal):
    """Update q_func for a given transition (s, c, R(s, c), s')

       Q(s,c) <- (1-a) Q(s,c) + a [R(s,a) + r maxQ(s',c')]

    Args:
        q_func (np.ndarray): current Q-function
        current_state_1, current_state_2 (int, int): two indices describing the␣
↪current state
        action_index (int): index of the current action
        object_index (int): index of the current object
        reward (float): the immediate reward the agent recieves from playing␣
↪current command
        next_state_1, next_state_2 (int, int): two indices describing the next␣
↪state
        terminal (bool): True if this epsiode is over

    Returns:
        None
    """
    # TODO Your code here

    if terminal:
        q_func[current_state_1, current_state_2, action_index, object_index] *=␣
↪1-ALPHA
        q_func[current_state_1, current_state_2, action_index, object_index] +=␣
↪ALPHA * reward
    else:
        # next_reward = -0.01
        next_reward = np.amax(q_func[next_state_1, next_state_2])
        q_func[current_state_1, current_state_2, action_index, object_index] *=␣
↪1-ALPHA
        q_func[current_state_1, current_state_2, action_index, object_index] +=␣
↪ALPHA * (reward + next_reward * GAMMA)
```

```
    return None  # This function shouldn't return anything
```

0.3.2 Epsilon-greedy exploration

Note that the Q-learning algorithm does not specify how we should interact in the world so as to learn quickly. It merely updates the values based on the experience collected. If we explore randomly, i.e., always select actions at random, we would most likely not get anywhere. A better option is to exploit what we have already learned, as summarized by current Q-values. We can always act greedily with respect to the current estimates, i.e., take an action $\pi(s) = argmax_{c \in C} Q(s, c)$. Of course, early on, these are not necessarily very good actions. For this reason, a typical exploration strategy is to follow a so-called -greedy policy: with probability take a random action out of C with probability 1 follow $\pi(s) = argmax_{c \in C} Q(s, c)$. The value of here balances exploration vs exploitation. A large value of means exploring more (randomly), not using much of what we have learned. A small , on the other hand, will generate experience consistent with the current estimates of Q-values.

Now you will write a function epsilon_greedy that implements the -greedy exploration policy using the current Q-function.

Reminder: You should implement this function locally first. You can read through the next tab to understand the context in which this function is called

Available Functions: You have access to the NumPy python library as np. Your code should also use constants NUM_ACTIONS and NUM_OBJECTS.

[22]:
```
# pragma: coderesponse template
def epsilon_greedy(state_1, state_2, q_func, epsilon):
    """Returns an action selected by an epsilon-Greedy exploration policy

    Args:
        state_1, state_2 (int, int): two indices describing the current state
        q_func (np.ndarray): current Q-function
        epsilon (float): the probability of choosing a random command

    Returns:
        (int, int): the indices describing the action/object to take
    """
    # TODO Your code here

    action_index, object_index = None, None

    greedy = True if np.random.random() <= epsilon else False
    if greedy:
        action_index = np.random.randint(NUM_ACTIONS)
        object_index = np.random.randint(NUM_OBJECTS)
    else:
        _, _, action_index, object_index = np.
↪unravel_index(q_func[state_1,state_2].argmax(), q_func.shape)
```

```
    return (action_index, object_index)
```

0.4 4. Tabular Q-learning for Home World game

In this section you will evaluate the tabular Q-learning algorithms for the Home world game. Recall that the state observable to the player is described in text. Therefore we have to choose a mechanism that maps text descriptions into vector representations.

In this section you will consider a simple approach that assigns a unique index for each text description. In particular, we will build two dictionaries:

- dict_room_desc that takes the room description text as the key and returns a unique scalar index
- dict_quest_desc that takes the quest description text as the key and returns a unique scalar index.

For instance, consider an observable state $s = (s_r, s_q)$, where sr and sq are the text descriptions for the current room and the current request, respectively. Then i_r =dict_room_desc$[s_r]$ gives the scalar index for s_r and i_q = dict_quest_desc$[s_q]$ gives the scalar index for s_q. That is, the textual state $s = (s_r, s_q)$ is mapped to a tuple $I = (i_r, i_q)$.

Normally, we would build these dictionaries as we train our agent, collecting descriptions and adding them to the list of known descriptions. For the purpose of this project, these dictionaries will be provided to you.

0.4.1 Evaluating Tabular Q-learning on Home World

The following python files are provided:

- framework.py contains various functions for the text-based game environment that the staff has implemented for you. Some functions that you can call to train and testing your reinforcement learning algorithms:
  - newGame()
    - Args: None
    - Return: A tuple where the first element is a description of the initial room, the second element is a description of the quest for this new game episode, and the last element is a Boolean variable with value False implying that the game is not over.
  - step_game()
    - Args:
      - room_index: An integer indicating the room index of the current state
      - room_desc_index: An integer representing the room description index of the current state
      - quest_index: An integer indicating the quest index of the current state
      - action_index: An integer used to represent the index of the selected action
      - object_index: An integer used to indicate the index of the selected object
    - Return: the system next state when the selected command is applied at the current state.
      - next_room_desc: The description of the room of the next state

- next_quest_desc: The description of the next quest
- reward: A real valued number representing the one-step reward obtained at this step
- terminal: A boolean valued number indicating whether this episode is over (either quest is finished, or the number of steps reaches the maximum number of steps for each episode).

- agent_tabular_QL.py contains various function templates that you will use to implement your learning algorithm.

In this section, you will evaluate your learning algorithm for the Home World game. The metric we use to measure an agent's performance is the cumulative discounted reward obtained per episode averaged over the episodes.

The evaluation procedure is as follows. Each experiment (or run) consists of multiple epochs (the number of epochs is NUM_EPOCHS). In each epoch:

1. You first train the agent on NUM_EPIS_TRAIN episodes, following an -greedy policy with =TRAINING_EP and updating the Q values.
2. Then, you have a testing phase of running NUM_EPIS_TEST episodes of the game, following an -greedy policy with =TESTING_EP, which makes the agent choose the best action according to its current Q-values 95% of the time. At the testing phase of each epoch, you will compute the cumulative discounted reward for each episode and then obtain the average reward over the NUM_EPIS_TEST episodes.

Finally, at the end of the experiment, you will get a sequence of data (of size NUM_EPOCHS) that represents the testing performance at each epoch.

Note that there is randomness in both the training and testing phase. You will run the experiment NUM_RUNS times and then compute the averaged reward performance over NUM_RUNS experiments.

Most of these operations are handled by the boilerplate code provided in the agent_tabular_QL.py file by functions run, run_epoch and main, but you will need to complete the run_episode function.

Write a run_episode function that takes a boolean argument (whether the epsiode is a training episode or not) and runs one episode.

Reminder: You should implement this function locally first. Make sure you can achieve reasonable performance on the Home World game before submitting your code

Available Functions: You have access to the NumPy python library as np, framework methods framework.newGame() and framework.step_game(), constants TRAINING_EP and TESTING_EP, GAMMA, dictionaries dict_room_desc and dict_quest_desc and previously implemented functions epsilon_greedy and tabular_QLearning

[23]:
```
# pragma: coderesponse template
def run_episode(for_training):
    """ Runs one episode
    If for training, update Q function
    If for testing, computes and return cumulative discounted reward

    Args:
        for_training (bool): True if for training
```

```
    Returns:
        None
    """

    epsilon = TRAINING_EP if for_training else TESTING_EP

    epi_reward = 0
    # initialize for each episode
    # TODO Your code here
    steps = 0

    (current_room_desc, current_quest_desc, terminal) = framework.newGame()
    #import pdb; pdb.set_trace()

    while not terminal:
        # Choose next action and execute
        # TODO Your code here
        steps += 1

        current_room_index = dict_room_desc[current_room_desc]
        quest_index = dict_quest_desc[current_quest_desc]

        action_index, object_index = epsilon_greedy(current_room_index,␣
↪quest_index, q_func, epsilon)
        next_room_desc, next_quest_desc, reward, terminal = framework.
↪step_game(current_room_desc,
                                                                        ␣
↪current_quest_desc,
                                                                        ␣
↪action_index,
                                                                        ␣
↪object_index)
        next_room_index = dict_room_desc[next_room_desc]
        next_quest_index = dict_quest_desc[next_quest_desc]

        if for_training:
            # update Q-function.
            # TODO Your code here
            tabular_q_learning(q_func, current_room_index, quest_index,␣
↪action_index,
                               object_index, reward, next_room_index,␣
↪next_quest_index,
                               terminal)

        if not for_training:
            # update reward
```

```
            # TODO Your code here
            epi_reward += GAMMA**(steps-1) * reward

        # prepare next step
        # TODO Your code here
        current_room_desc = next_room_desc
        current_quest_desc = next_quest_desc

    if not for_training:
        return epi_reward
```

0.4.2 Report performance

In your Q-learning algorithm, initialize Q at zero. Set NUM_RUNS = 10, NUM_EPIS_TRAIN = 25, NUM_EPIS_TEST = 50, = 0.5, TRAINING_EP = 0.5, TESTING_EP = 0.05 and the learning rate $\alpha = 0.1$.

Please enter the number of epochs when the learning algorithm converges. That is, the testing performance become stable.

- 21

Please enter the average episodic rewards of your Q-learning algorithm when it converges.

- 0.504442786336921

[24]:
```
def run_epoch():
    """Runs one epoch and returns reward averaged over test episodes"""
    rewards = []

    for _ in range(NUM_EPIS_TRAIN):
        run_episode(for_training=True)

    for _ in range(NUM_EPIS_TEST):
        rewards.append(run_episode(for_training=False))

    return np.mean(np.array(rewards))
```

[25]:
```
def run():
    """Returns array of test reward per epoch for one run"""
    global q_func
    q_func = np.zeros((NUM_ROOM_DESC, NUM_QUESTS, NUM_ACTIONS, NUM_OBJECTS))

    single_run_epoch_rewards_test = []
    pbar = tqdm(range(NUM_EPOCHS), ncols=80)
    for _ in pbar:
        single_run_epoch_rewards_test.append(run_epoch())
        pbar.set_description(
            "Avg reward: {:0.6f} | Ewma reward: {:0.6f}".format(
```

```
                np.mean(single_run_epoch_rewards_test),
                ewma(single_run_epoch_rewards_test)))
    return single_run_epoch_rewards_test
```

[ ]:
```
if __name__ == '__main__':
    # Data loading and build the dictionaries that use unique index for each␣
 ↪state
    (dict_room_desc, dict_quest_desc) = make_all_states_index()
    NUM_ROOM_DESC = len(dict_room_desc)
    NUM_QUESTS = len(dict_quest_desc)

    # set up the game
    load_game_data()

    epoch_rewards_test = []  # shape NUM_RUNS * NUM_EPOCHS

    for _ in range(NUM_RUNS):
        epoch_rewards_test.append(run())

    epoch_rewards_test = np.array(epoch_rewards_test)

    x = np.arange(NUM_EPOCHS)
    fig, axis = plt.subplots()
    axis.plot(x, np.mean(epoch_rewards_test,
                         axis=0))  # plot reward per epoch averaged per run
    axis.set_xlabel('Epochs')
    axis.set_ylabel('reward')
    axis.set_title(('Tablular: nRuns=%d, Epilon=%.2f, Epi=%d, alpha=%.4f' %
                    (NUM_RUNS, TRAINING_EP, NUM_EPIS_TRAIN, ALPHA)))
    plt.show()
```

0.5 5. Parameter Tuning

Effects of adjusting epsilon In this question, you will investigate the impact of ε on the convergence of Q-learning algorithm. Fix the learning rate and do the experiments with different values of the training $TRAININGEP \in [0, 1]$. What you have observed?

- Y For very large ε (say ε=1), the algorithm converges slower compared to ε=0.5
- N For very large ε (say ε=1), the algorithm converges faster compared to ε=0.5
- N For very small ε (say ε=0.00001), the algorithm converges slower compared to ε=0.5
- Y For very small ε (say ε=0.00001), the algorithm converges faster compared to ε=0.5

0.5.1 Effects of alpha

In this question, you will investigate the impact of α on the convergence of Q-learning algorithm. Fix the exploration parameter ε=0.5 and do the experiments with different values of the training $\alpha \in [10^6, 1]$. What you have bserved?

- N The algorithm converges for all values of α

- Y The algorithm does not converge for all values of α
- N The smaller α, the slower the convergence
- N The smaller α, the faster the convergence

0.6 6. Q-learning with linear function approximation

Since the state displayed to the agent is described in text, we have to choose a mechanism that maps text descriptions into vector representations. A naive way is to create one unique index for each text description, as we have done in previous part. However, such approach becomes infeasible when the state space becomes huge. To tackle this challenge, we can design some representation generator that does not scale as the original textual state space. In particular, a representation generator $\Psi_R(\cdot)$ reads raw text displayed to the agent and converts it to a vector representation $v_s = \Psi_R(s)$. One approach is to use a bag-of-words representation derived from the text description.

In large games, it is often impractical to maintain the Q-value for all possible state-action pairs. One solution to this problem is to approximate $Q(s, c)$ using a parametrized function $Q(s, c; \theta)$.

In this section we consider a linear parametric architecture:

$$Q(s, c; \theta) = \phi(s, c)^T \theta = \sum_{i=1}^{d} \phi_i(s, c)\theta_i,$$

where $\phi(s, c)$ is a fixed feature vector in Rdfor state-action pair (s, c) with i-th component given by $\phi_i(s, c)$, and $\theta \in R_d$ is a parameter vector that is shared across state-action pairs. The key challenge here is to design the feature vectors $\phi(s, c)$. Note that given a textual state s, we first translate it to a vector representation v_s using $\Psi_R(s)$. So the question here is how to design a mapping function convert $\Psi_R(s), c)$ into a vector representation in R^d. Assume that the size of action space is d_C, and the dimension of the vector space for state representation is d_R

0.6.1 Feature engineering

Exercise: Consider the following feature engineering. Define a function $\Psi_c : C \to R^{d_C}$ where the j-th component $\Psi_{C_j}(c)$ is given as follows:

$$\Psi_{C,j} = one\ if\ j = c\ zero\ else$$

The feature vector is defined as
$\phi(s, c) = [\Psi_R(s), \Psi_C(c)]^T$.
Will it work?
No
Alternatively, consider the following feature map: $\phi(s, c) \in R^{d_C - d_R}$, where $\phi_i(s, c) = 0$ for all $i \notin [(c1)d_R + 1, cd_R]$, and for $i \in [(c1)d_R + 1, cd_R], \phi_i(s, c) = \Psi_R, i - (c - 1)d_R{}^{(s)}$. That is,
$\phi(s, c) = [0, \ldots, \Psi_R(s), \ldots, 0]^T$
You will implement this feature map in the next tab.

0.6.2 Computing theta update rule

The Q-learning approximation algorithm starts with an initial parameter estimate of θ. As the tabular Q-learning, upon observing a data tuple $(s, c, R(s, c), s')$, the target value y for the Q-value of (s, c) is defined as the sampled version of the Bellman operator, $y = R(s, c) + \gamma max_{c'} Q(s', c', \theta)$.

Then the parameter is simply updated by taking a gradient step with respect to the squared loss $L(\theta) = \frac{1}{2}(yQ(s,c,\theta))^2$.

The negative gradient can be computed as follows:

$g(\theta) = -\frac{\delta}{\delta\theta}L(\theta) = (y - Q(s,c,\theta))\phi(s,c)$

Hence the update rule for θ is :

$\theta \leftarrow \theta + \alpha g(\theta) = \theta + \alpha[R(s,c) + \gamma max_{c'}Q(s',c',\theta) - Q(s,c,,\theta)]\phi(s,c)$

where α is the learning rate.

0.7 7. Linear Q-Learning

In this tab, you will implement the Q-learning algorithm with linear function approximation. Recall the linear approximation we chose. $Q(s,c,\theta) = \phi(s,c)^T\theta$

with $\phi(s,c) = [0,\ldots,0,\Psi_R(s),0,\ldots,0]^T$

Now, define $\hat{\theta}_i$ for $i \in d_C$

so that: \$$\theta = [\hat{\theta}_1,\ldots,\hat{\theta}i,\ldots,\{d_A\}$

With this notation, we get:

$$Q(s,c,\theta) = \Phi_R(s)^T\hat{\theta}_c$$

In practice, we can implement \$ as a 2D array, so that

$$[Q(s,1,\theta,\ldots,Q(s,d_C,\theta]^T = [\hat{\theta}_1^T,\ldots,\hat{\theta}_{d_C}^T]^T$$

[26]:
```
"""Linear QL agent"""
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import framework
import utils

DEBUG = False
```

[27]:
```
GAMMA = 0.5  # discounted factor
TRAINING_EP = 0.5  # epsilon-greedy parameter for training
TESTING_EP = 0.05  # epsilon-greedy parameter for testing
NUM_RUNS = 10
NUM_EPOCHS = 600
NUM_EPIS_TRAIN = 25  # number of episodes for training at each epoch
NUM_EPIS_TEST = 50  # number of episodes for testing
ALPHA = 0.001  # learning rate for training

ACTIONS = framework.get_actions()
OBJECTS = framework.get_objects()
NUM_ACTIONS = len(ACTIONS)
NUM_OBJECTS = len(OBJECTS)
```

[28]:
```
def tuple2index(action_index, object_index):
    """Converts a tuple (a,b) to an index c"""
    return action_index * NUM_OBJECTS + object_index
```

[29]:
```
def index2tuple(index):
    """Converts an index c to a tuple (a,b)"""
    return index // NUM_OBJECTS, index % NUM_OBJECTS
```

0.7.1 Epsilon-greedy exploration

Now you will write a function epsilon_ greedy that implements the ε-greedy exploration policy using the current Q-function.

Hint: You can access $Q(s, c, \theta)$ using q_value = (theta @ state_vector)[tuple2index(action_index, object_index)]

Available Functions: You have access to the NumPy python library as np and functions tuple2index and index2tuple. Your code should also use constants NUM_ACTIONS and NUM_OBJECTS

[30]:
```
def epsilon_greedy(state_vector, theta, epsilon):
    """Returns an action selected by an epsilon-greedy exploration policy

    Args:
        state_vector (np.ndarray): extracted vector representation
        theta (np.ndarray): current weight matrix
        epsilon (float): the probability of choosing a random command

    Returns:
        (int, int): the indices describing the action/object to take
    """
    # TODO Your code here
    action_index, object_index = None, None

    greedy = True if np.random.random() <= epsilon else False
    if greedy:
        action_index = np.random.randint(NUM_ACTIONS)
        object_index = np.random.randint(NUM_OBJECTS)
    else:
        next_index = (theta @ state_vector).argmax()
        action_index, object_index = index2tuple(next_index)
        #q_value = (theta @ current_state_vector)[tuple2index(action_index,␣
↪object_index)]

    return (action_index, object_index)
```

0.7.2 Linear Q-learning

Write a function linear_q_learning that updates the theta weight matrix, given the transition date $(s, a, R(s, a), s')$.

Reminder: You should implement this function locally first. You should test this function along with the next one and make sure you achieve reasonable performance

Hint: You can access $Q(s, a, \theta)$ using q_value = (theta @ state_vector)[tuple2index(action_index, object_index)]

Available Functions: You have access to the NumPy python library as np. You should also use constants ALPHA and GAMMA in your code

[31]:
```
def linear_q_learning(theta, current_state_vector, action_index, object_index,
                      reward, next_state_vector, terminal):
    """Update theta for a given transition

    Args:
        theta (np.ndarray): current weight matrix
        current_state_vector (np.ndarray): vector representation of current␣
↪state
        action_index (int): index of the current action
        object_index (int): index of the current object
        reward (float): the immediate reward the agent recieves from playing␣
↪current command
        next_state_vector (np.ndarray): vector representation of next state
        terminal (bool): True if this epsiode is over

    Returns:
        None
    """
    # TODO Your code here

    tuple_index = tuple2index(action_index, object_index)
    q_value = (theta @ current_state_vector)[tuple_index]

    if terminal:
        next_reward  = 0
        g_theta = (reward - q_value) * current_state_vector
    else:
        next_reward = np.max(theta @ next_state_vector)
        g_theta = (reward - q_value + GAMMA * next_reward) *␣
↪current_state_vector

    theta[tuple_index] = theta[tuple_index] + ALPHA * g_theta

    return None
```

[32]:
```
def run_episode(for_training):
    """ Runs one episode
    If for training, update Q function
    If for testing, computes and return cumulative discounted reward

    Args:
        for_training (bool): True if for training

    Returns:
        None
```

```
    """
    epsilon = TRAINING_EP if for_training else TESTING_EP
    epi_reward = 0
    # initialize for each episode
    # TODO Your code here
    steps = 0

    (current_room_desc, current_quest_desc, terminal) = framework.newGame()
    while not terminal:

        steps += 1

        # Choose next action and execute
        current_state = current_room_desc + current_quest_desc
        current_state_vector = utils.extract_bow_feature_vector(current_state,␣
↪dictionary)
        # TODO Your code here

        action_index, object_index = epsilon_greedy(current_state_vector,␣
↪theta, epsilon)
        next_room_desc, next_quest_desc, reward, terminal = framework.
↪step_game(current_room_desc,
                                                                          ␣
↪current_quest_desc,
                                                                          ␣
↪action_index,
                                                                          ␣
↪object_index)
        next_state = next_room_desc + next_quest_desc
        next_state_vector = utils.extract_bow_feature_vector(next_state,␣
↪dictionary)

        if for_training:
            # update Q-function.
            # TODO Your code here
            linear_q_learning(theta, current_state_vector, action_index,␣
↪object_index,
                              reward, next_state_vector, terminal)


        if not for_training:
            # update reward
            # TODO Your code here
            epi_reward += GAMMA**(steps-1) * reward

        # prepare next step
```

```
        # TODO Your code here
        current_room_desc = next_room_desc
        current_quest_desc = next_quest_desc

    if not for_training:
        return epi_reward
```

[33]:
```
def run_epoch():
    """Runs one epoch and returns reward averaged over test episodes"""
    rewards = []

    for _ in range(NUM_EPIS_TRAIN):
        run_episode(for_training=True)

    for _ in range(NUM_EPIS_TEST):
        rewards.append(run_episode(for_training=False))

    return np.mean(np.array(rewards))
```

0.7.3 Evaluate linear Q-learning on Home World game

Adapt your run_episode function to call linear_Q_learning and evaluate your performance using hyperparmeters: Set NUM_RUNS =5, NUM_EPIS_TRAIN=25, NUM_EPIS_TEST=50, =0.5, TRAINING_EP=0.5, TESTING_EP=0.05 and the learning rate =0.001.

Please enter the average episodic rewards of your Q-learning algorithm when it converges. 0.37

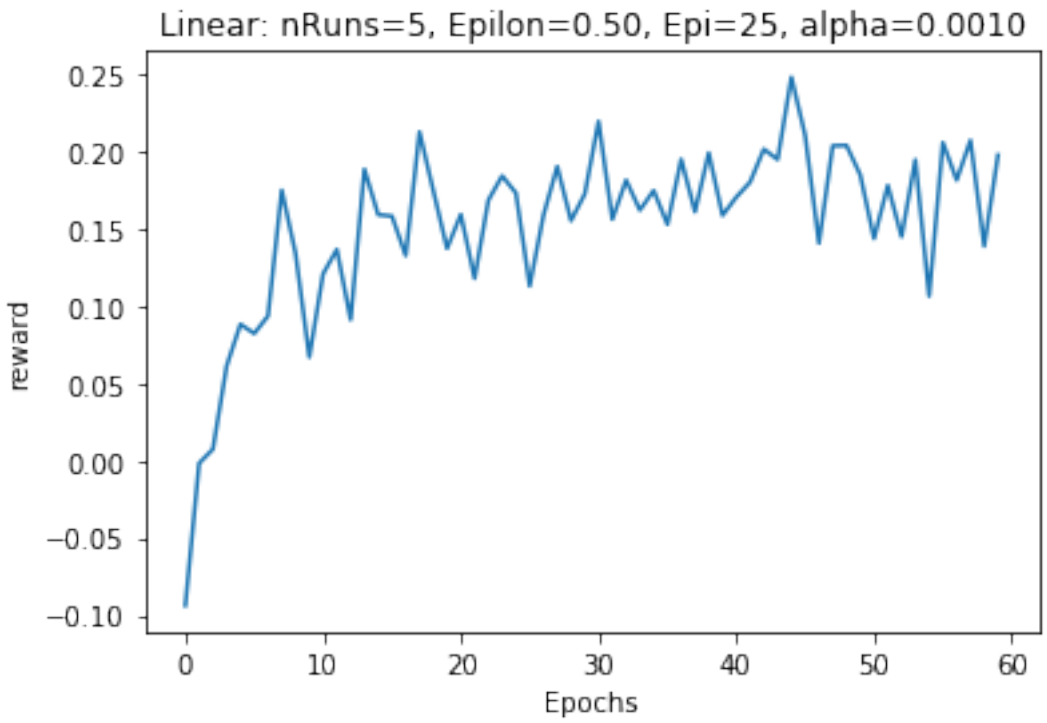
[34]:
```
GAMMA = 0.5  # discounted factor
TRAINING_EP = 0.5  # epsilon-greedy parameter for training
TESTING_EP = 0.05  # epsilon-greedy parameter for testing
NUM_RUNS = 5
NUM_EPOCHS = 60
NUM_EPIS_TRAIN = 25  # number of episodes for training at each epoch
NUM_EPIS_TEST = 50  # number of episodes for testing
ALPHA = 0.001  # learning rate for training
```

[35]:
```
ACTIONS = framework.get_actions()
OBJECTS = framework.get_objects()
NUM_ACTIONS = len(ACTIONS)
NUM_OBJECTS = len(OBJECTS)
```

[36]:
```
def run():
    """Returns array of test reward per epoch for one run"""
    global theta
    theta = np.zeros([action_dim, state_dim])

    single_run_epoch_rewards_test = []
    pbar = tqdm(range(NUM_EPOCHS), ncols=80)
    for _ in pbar:
```

```
        single_run_epoch_rewards_test.append(run_epoch())
        pbar.set_description(
            "Avg reward: {:0.6f} | Ewma reward: {:0.6f}".format(
                np.mean(single_run_epoch_rewards_test),
                utils.ewma(single_run_epoch_rewards_test)))
    return single_run_epoch_rewards_test


if __name__ == '__main__':
    state_texts = utils.load_data('game.tsv')
    dictionary = utils.bag_of_words(state_texts)
    state_dim = len(dictionary)
    action_dim = NUM_ACTIONS * NUM_OBJECTS

    # set up the game
    framework.load_game_data()

    epoch_rewards_test = []  # shape NUM_RUNS * NUM_EPOCHS

    for _ in range(NUM_RUNS):
        epoch_rewards_test.append(run())

    epoch_rewards_test = np.array(epoch_rewards_test)

    x = np.arange(NUM_EPOCHS)
    fig, axis = plt.subplots()
    axis.plot(x, np.mean(epoch_rewards_test,
                         axis=0))  # plot reward per epoch averaged per run
    axis.set_xlabel('Epochs')
    axis.set_ylabel('reward')
    axis.set_title(('Linear: nRuns=%d, Epilon=%.2f, Epi=%d, alpha=%.4f' %
                    (NUM_RUNS, TRAINING_EP, NUM_EPIS_TRAIN, ALPHA)))
```

```
Avg reward: 0.138248 | Ewma reward: 0.172529: 100%|| 60/60 [00:04<00:00,
12.90it/s]
Avg reward: 0.187988 | Ewma reward: 0.194078: 100%|| 60/60 [00:04<00:00,
12.85it/s]
Avg reward: 0.143033 | Ewma reward: 0.180211: 100%|| 60/60 [00:04<00:00,
13.07it/s]
Avg reward: 0.118791 | Ewma reward: 0.148852: 100%|| 60/60 [00:04<00:00,
12.21it/s]
Avg reward: 0.171781 | Ewma reward: 0.189587: 100%|| 60/60 [00:04<00:00,
13.08it/s]
```



0.8 8. Deep Q-network

As you have observed in the previous tab, a linear model is not able to correctly approximate the Q-function for our simple task.

In this section, you will approximate $Q(s, c)$ with a neural network. You will be provided with a DQN that takes the state representation (bag-of-words) and outputs the predicted Q values for the different "actions" and "objects".

0.8.1 Deep Q network

Complete the function deep_q_learning that updates the model weights, given the transition date $(s, c, R(s, c), s')$.

Please enter the average episodic rewards of your Q-learning algorithm when it converges.
0.50

[39]:
```
"""Tabular QL agent"""
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import framework
```

```
import utils

DEBUG = False

GAMMA = 0.5  # discounted factor
TRAINING_EP = 0.5  # epsilon-greedy parameter for training
TESTING_EP = 0.05  # epsilon-greedy parameter for testing
NUM_RUNS = 10
NUM_EPOCHS = 300
NUM_EPIS_TRAIN = 25  # number of episodes for training at each epoch
NUM_EPIS_TEST = 50  # number of episodes for testing
ALPHA = 0.1  # learning rate for training

ACTIONS = framework.get_actions()
OBJECTS = framework.get_objects()
NUM_ACTIONS = len(ACTIONS)
NUM_OBJECTS = len(OBJECTS)

model = None
optimizer = None
```

[38]:
```
class DQN(nn.Module):
    """A simple deep Q network implementation.
    Computes Q values for each (action, object) tuple given an input state␣
 ↪vector
    """

    def __init__(self, state_dim, action_dim, object_dim, hidden_size=100):
        super(DQN, self).__init__()
        self.state_encoder = nn.Linear(state_dim, hidden_size)
        self.state2action = nn.Linear(hidden_size, action_dim)
        self.state2object = nn.Linear(hidden_size, object_dim)

    def forward(self, x):
        state = F.relu(self.state_encoder(x))
        return self.state2action(state), self.state2object(state)
```

[ ]:
```
def epsilon_greedy(state_vector, epsilon):
    """Returns an action selected by an epsilon-greedy exploration policy

    Args:
        state_vector (torch.FloatTensor): extracted vector representation
        epsilon (float): the probability of choosing a random command

    Returns:
        (int, int): the indices describing the action/object to take
    """
    # TODO Your code here
```

```
    action_index, object_index = None, None

    greedy = True if np.random.random() <= epsilon else False
    if greedy:
        action_index = np.random.randint(NUM_ACTIONS)
        object_index = np.random.randint(NUM_OBJECTS)
    else:
        next_index = (theta @ state_vector).argmax()
        action_index, object_index = index2tuple(next_index)


    return (action_index, object_index)
```

[ ]:
```
# pragma: coderesponse template
def deep_q_learning(current_state_vector, action_index, object_index, reward,
                    next_state_vector, terminal):
    """Updates the weights of the DQN for a given transition

    Args:
        current_state_vector (torch.FloatTensor): vector representation of␣
 ↪current state
        action_index (int): index of the current action
        object_index (int): index of the current object
        reward (float): the immediate reward the agent recieves from playing␣
 ↪current command
        next_state_vector (torch.FloatTensor): vector representation of next␣
 ↪state
        terminal (bool): True if this epsiode is over

    Returns:
        None
    """
    with torch.no_grad():
        q_values_action_next, q_values_object_next = model(next_state_vector)
    maxq_next = 1 / 2 * (q_values_action_next.max()
                         + q_values_object_next.max())

    q_value_cur_state = model(current_state_vector)

    # TODO Your code here

    loss = None

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
# pragma: coderesponse end
```

[ ]:
```
def run_episode(for_training):
    """
        Runs one episode
        If for training, update Q function
        If for testing, computes and return cumulative discounted reward
    """
    epsilon = TRAINING_EP if for_training else TESTING_EP
    epi_reward = 0

    # initialize for each episode
    # TODO Your code here

    steps = 0

    (current_room_desc, current_quest_desc, terminal) = framework.newGame()
    while not terminal:

        steps += 1

        # Choose next action and execute
        current_state = current_room_desc + current_quest_desc
        current_state_vector = torch.FloatTensor(
            utils.extract_bow_feature_vector(current_state, dictionary))

        # TODO Your code here
        action_index, object_index = epsilon_greedy(current_state_vector,␣
↪epsilon)
        next_room_desc, next_quest_desc, reward, terminal = framework.
↪step_game(current_room_desc,
                                                                          ␣
↪current_quest_desc,
                                                                          ␣
↪action_index,
                                                                          ␣
↪object_index)

        next_state = next_room_desc + next_quest_desc
        next_state_vector = torch.FloatTensor(
            utils.extract_bow_feature_vector(next_state, dictionary))

        if for_training:
            # update Q-function.
            # TODO Your code here
            deep_q_learning(current_state_vector, action_index, object_index,
                            reward, next_state_vector, terminal)

        if not for_training:
```

```
            # update reward
            # TODO Your code here
            epi_reward += GAMMA**(steps-1) * reward

        # prepare next step
        # TODO Your code here
        current_room_desc = next_room_desc
        current_quest_desc = next_quest_desc

    if not for_training:
        return epi_reward
```

[ ]:
```
def run_epoch():
    """Runs one epoch and returns reward averaged over test episodes"""
    rewards = []

    for _ in range(NUM_EPIS_TRAIN):
        run_episode(for_training=True)

    for _ in range(NUM_EPIS_TEST):
        rewards.append(run_episode(for_training=False))

    return np.mean(np.array(rewards))
```

[ ]:
```
def run():
    """Returns array of test reward per epoch for one run"""
    global model
    global optimizer
    model = DQN(state_dim, NUM_ACTIONS, NUM_OBJECTS)
    optimizer = optim.SGD(model.parameters(), lr=ALPHA)

    single_run_epoch_rewards_test = []
    pbar = tqdm(range(NUM_EPOCHS), ncols=80)
    for _ in pbar:
        single_run_epoch_rewards_test.append(run_epoch())
        pbar.set_description(
            "Avg reward: {:0.6f} | Ewma reward: {:0.6f}".format(
                np.mean(single_run_epoch_rewards_test),
                utils.ewma(single_run_epoch_rewards_test)))
    return single_run_epoch_rewards_test


if __name__ == '__main__':
    state_texts = utils.load_data('game.tsv')
    dictionary = utils.bag_of_words(state_texts)
    state_dim = len(dictionary)

    # set up the game
```

```
    framework.load_game_data()

    epoch_rewards_test = []  # shape NUM_RUNS * NUM_EPOCHS

    for _ in range(NUM_RUNS):
        epoch_rewards_test.append(run())

    epoch_rewards_test = np.array(epoch_rewards_test)

    x = np.arange(NUM_EPOCHS)
    fig, axis = plt.subplots()
    axis.plot(x, np.mean(epoch_rewards_test,
                         axis=0))  # plot reward per epoch averaged per run
    axis.set_xlabel('Epochs')
    axis.set_ylabel('reward')
    axis.set_title(('Linear: nRuns=%d, Epilon=%.2f, Epi=%d, alpha=%.4f' %
                    (NUM_RUNS, TRAINING_EP, NUM_EPIS_TRAIN, ALPHA)))
    plt.show()
```

[ ]:
```
Please enter the average episodic rewards of your Q-learning algorithm when it␣
↪converges.
```

[ ]:

[ ]:

[ ]:
```
NUM_ROOM_DESC
```

[ ]:

[ ]:

[ ]:
```
a = np.array([11, 12, 13, 14, 15, 16, 17, 15, 11, 12, 14, 15]).reshape(3,4)
a
```

[ ]:
```
row_index = np.where((a[:,0] == 11) * (a[:,1] == 12) * (a[:,2] == 14))
a[row_index, :]
```

[ ]:
```
np.where(a[np.where(a[:,0] == 11),1]==12)
```

[ ]:

[ ]:
```
(dict_room_desc, dict_quest_desc) = make_all_states_index()
NUM_ROOM_DESC = len(dict_room_desc)
NUM_QUESTS = len(dict_quest_desc)
```

[ ]:
```
q_func = np.zeros((NUM_ROOM_DESC, NUM_QUESTS, NUM_ACTIONS, NUM_OBJECTS))
```

[ ]:
```
np.where((q_func[:,0] == 1) * (q_func[:,1] == 2))
```

[ ]:
```
np.argmax(q_func[0,0])
```

[ ]:
```
q_func[0,0,1,3]=9
```

[ ]:
```
i,j,k,l = np.unravel_index(q_func.argmax(), q_func.shape)
i,j,k,l
```

[ ]:
```
i,j,k,l = np.unravel_index(q_func[0,0].argmax(), q_func.shape)
i,j,k,l
```

[ ]:
```
np.amax(q_func[0,0])
```

[ ]:
```

```

[ ]:
```
H = np.array([(r, q) for r in range(4) for q in range(4)])
H
```

[ ]:
```
R = np.array([1, -0.01, -0.1])
R
```

[ ]:
```
from random import randint
h0 = np.array([randint(0, 3), randint(0,3)])
h0
```

[ ]:
```
def R(s, c):
    # s: observable state, c: (a, b), a: action; b: object
    if 0==0:
        return 1
    elif 1==1:
        return -0.01
    else:
        return -0.1
```

[ ]:
```
def H2S():

```

[ ]:
```

```