AU 332 Artificial Intelligence: Principles and Techniques

By: Bo Yue (517021910825)

HW#: 3

October 31, 2019

I. INTRODUCTION

A. Purpose

The goal of today's lab or homework3 is to render me to be familiar with Reinforcement Learning(RL). There are two environments: the Maze Environment and Atari Game. I have to implement Dyna-Q Learning and Deep-Q Learning, respectively.

B. Environment

The environment for this lab and later labs are listed as follows:

- Pycharm(Python 3.7)
- Ubuntu 16.04 LTS

II. REINFORCEMENT LEARNING IN MAZE ENVIRONMENT

In the maze part of homework3, I will implement an agent based on Dyna-Q Learning, which can find optimal policy in two mazes(a simple one and a hard one), shown in Figure 1.The red rectangle represents the start point and the green circle represents the exit point. You can move upward, downward, leftward and rightward and you should avoid falling into the traps, which are represented by the black rectangles. Finding the exit will give a reward +1 and falling into traps will cause a reward -1, and both of the two cases will terminate current iteration. Moreover, you will get a bonus reward +2 if you find the treasure, which shown as golden diamond.

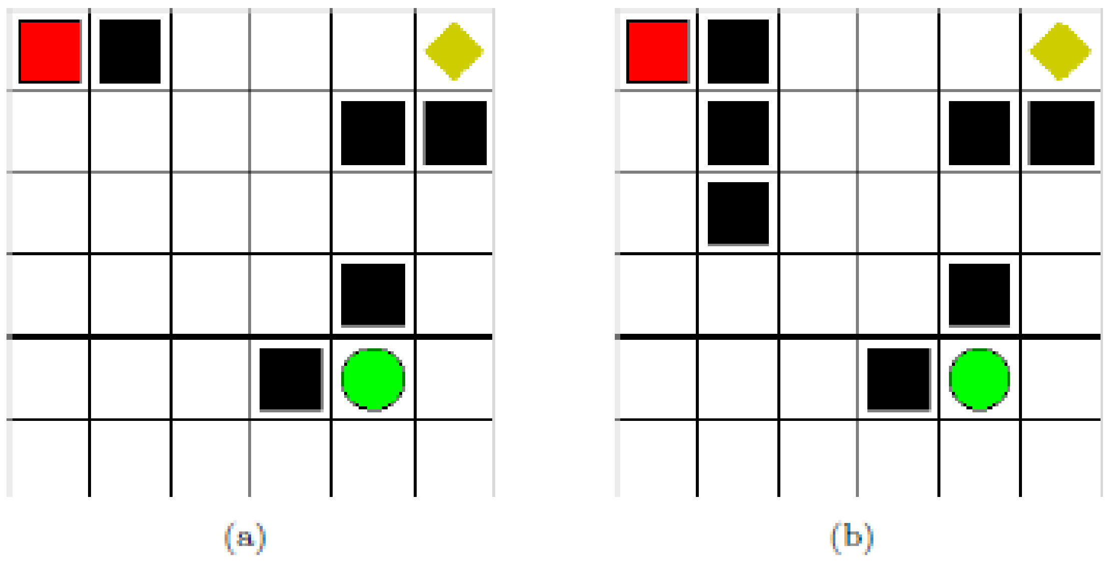


FIG. 1: Maze Environment

In general, besides fully implementing Dyna-Q Learning, I use a combination of undirected Semi-Uniform Distributed Exploration and directed Counter-Based Exploration

A. Dya-Q Learning

Dyna-Q Learning is composed of model-free Q-Learning and model-based Dyna-Q Learning. Model-free part updates Q-value simply based on its expensive experience, while model-based part updates Q-value when backpropagating the experienced fragment of episode.

1. *Exploration Function*

Undirected Semi-Uniform Distributed Exploration This is the same as epsilon-greedy strategy. $(1 - P_{best})$ equals to ϵ, where one legal action is randomly choosed, including the optimal action. Besides this, the optimal action has another P_{best} probability to be chosen. The theoretical equation is given below:

$$P(a) = \begin{cases} P_{best} + \dfrac{1 - P_{best}}{\#\text{ of actions}} & \text{, if } a \text{ maximizes } f(a) \\ \dfrac{1 - P_{best}}{\#\text{ of actions}} & \text{, otherwise} \end{cases}$$

Directed Counter-Based Exploration Here, I simplify the counter-based exploration strategy to some extent. The principle is that the location with higher expected Q-value and less visited counts should be preferred. This makes sense, as this kind of locations do have the potential to be the black horse.The exploration term is the quotient of the counter value for the current state and the expected counter value for the state that results from taking an action. In the given equation, α is constant factor weighting exploration versus exploitation. The theoretical equation is also given below:

Get Rid of Stuck Here, I have taken into account a kind of scenario, where stuck or repeated situation may occur. That is to say, when the agent get the reward +3, the *qTable* is then updated, and the specific Q(s, a) becomes large enough for s' to go back again to where the bonus does not exist any longer. This is a waste of time and computational power. I use a *qTable* to get rid of this scenario.

$$eval_c(a) = \alpha \cdot f(a) + \frac{c(s)}{E[c \mid s, a]}$$

$$E[c \mid s, a] = \sum_{s} P_{s \to s'}(a) \cdot c(s')$$

Outcomes For the same agent, the learning process(to acquire +4 reward) are 11 iterations for the simple environment and 39 iterations for the hard environment. Refer to Figure 2 & 3.

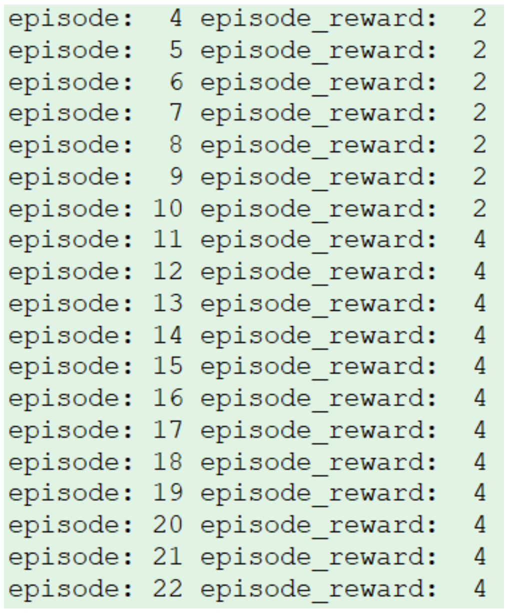

```
episode:  4 episode_reward:  2
episode:  5 episode_reward:  2
episode:  6 episode_reward:  2
episode:  7 episode_reward:  2
episode:  8 episode_reward:  2
episode:  9 episode_reward:  2
episode: 10 episode_reward:  2
episode: 11 episode_reward:  4
episode: 12 episode_reward:  4
episode: 13 episode_reward:  4
episode: 14 episode_reward:  4
episode: 15 episode_reward:  4
episode: 16 episode_reward:  4
episode: 17 episode_reward:  4
episode: 18 episode_reward:  4
episode: 19 episode_reward:  4
episode: 20 episode_reward:  4
episode: 21 episode_reward:  4
episode: 22 episode_reward:  4
```

FIG. 2: outcome 1

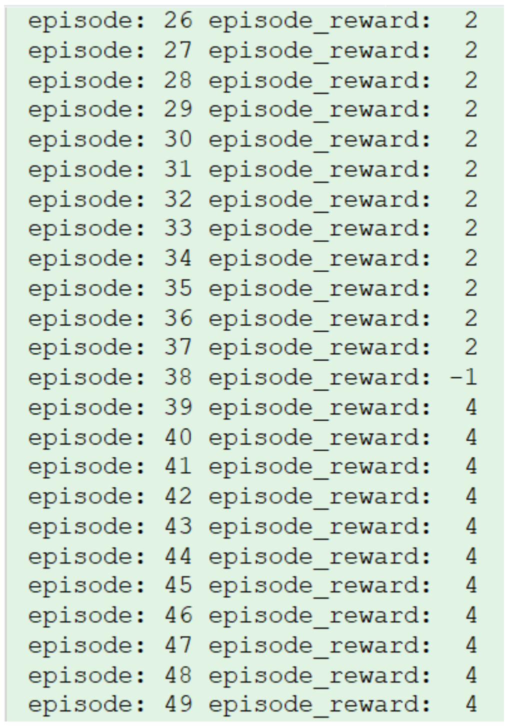

```
episode: 26 episode_reward:  2
episode: 27 episode_reward:  2
episode: 28 episode_reward:  2
episode: 29 episode_reward:  2
episode: 30 episode_reward:  2
episode: 31 episode_reward:  2
episode: 32 episode_reward:  2
episode: 33 episode_reward:  2
episode: 34 episode_reward:  2
episode: 35 episode_reward:  2
episode: 36 episode_reward:  2
episode: 37 episode_reward:  2
episode: 38 episode_reward: -1
episode: 39 episode_reward:  4
episode: 40 episode_reward:  4
episode: 41 episode_reward:  4
episode: 42 episode_reward:  4
episode: 43 episode_reward:  4
episode: 44 episode_reward:  4
episode: 45 episode_reward:  4
episode: 46 episode_reward:  4
episode: 47 episode_reward:  4
episode: 48 episode_reward:  4
episode: 49 episode_reward:  4
```

FIG. 3: outcome 2

B. Code Part

The code in main.py and agent.py are appended as follows:
main.py

```
from agent import Agent, Model #import class Agent from agent
import time

maze = '1'

if maze == '1':
    from maze_env1 import Maze
elif maze == '2':
    from maze_env2 import Maze


if __name__ == "__main__":
    ### START CODE HERE ###
    # This is an agent with random policy. You can learn how to interact with the
                                                  environment through the code below.
    # Then you can delete it and write your own code.

    episodes              = 100
    model_based_episodes = 5
    env    = Maze()
    model  = Model(actions=list(range(env.n_actions)))
    agent  = Agent(actions=list(range(env.n_actions)))       #   range  (4)      0      ,1,2,
                                                               3
    bonus = False
    for episode in range(episodes):                         #

        s = env.reset()
        episode_reward = 0
        while True:
            #env.render()                 # You can comment all render() to turn off the
                                                          graphical interface in training
                                                          process to accelerate your code.

            # move one step

            a = agent.choose_action(str(s),bonusflag=bonus)

            s_, r, done = env.step(a)
            if r == 3:
                bonus = True
            # update Q model-free
            agent.learn(str(s), a, r, str(s_), done)

            model.store_transition(str(s), a, r, s_)

            # update Q model-based
            for n in range(model_based_episodes):
                ss, sa  = model.sample_s_a()
                sr, ss_ = model.get_r_s_(ss, sa)
                agent.learn(ss, sa, sr, str(ss_), done)

            episode_reward += r
            s = s_

            if done:
                #env.render()
                time.sleep(0.5)
                break
        # format output
        print('episode: %2s'%episode, 'episode_reward: %2s'%episode_reward)

    ### END CODE HERE ###

    print('\ntraining over\n')
```

agent.py

```
import numpy as np
import pandas as pd
import math

class Agent:
    ### START CODE HERE ###

    #                      actionsepsilon
    def __init__(self, actions, alpha=0.1, gamma=0.7, epsilon=1):
        self.actions = actions
        self.alpha   = alpha                              # learning_rate
        self.gamma   = gamma                              # reward_decay
        self.epsilon = epsilon
        self.qTable  = pd.DataFrame(columns=self.actions) # tabular q-value
        self.qTable_ = pd.DataFrame(columns=self.actions) # tabular q-value without bonus
        self.cTable  = pd.DataFrame(columns=self.actions) # tabular counter value

    def choose_action(self, observation, bonusflag):
        self.state_exist(observation)

        # tune epsilon
        self.epsilon = 0.01
        # in case stuck or repeated situations
        if not bonusflag:
            # epsilon-greedy method
            if np.random.uniform() < self.epsilon:
                # random
                action = np.random.choice(self.actions)
            else:
                # get all potential actions
                action_to_be = self.qTable.loc[observation]
                # if more than one
                action = np.random.choice(action_to_be[action_to_be == max(action_to_be)].
                                                            index)
        else:
            if np.random.uniform() < self.epsilon:
                # random
                action = np.random.choice(self.actions)
            else:
                # get all potential actions
                action_to_be = self.qTable_.loc[observation]
                # if more than one
                action = np.random.choice(action_to_be[action_to_be == max(action_to_be)].
                                                            index)

        return action

    def learn(self, s, a, r, s_, done):
        k = 1
        flag = False
        self.state_exist(s_)

        q_predict = self.qTable.loc[s, a]
        q_predict_ = self.qTable_.loc[s, a]
        if not done:
            # in case stuck or repeated situation
            if r == 3 and flag == False:
                flag = True
                self.qTable_ = self.qTable
                q_target_ = self.gamma * max((self.qTable.loc[s_] + k / (1+self.cTable.loc[
                                                              s_].sum())))
                self.qTable_.loc[s, a] += self.alpha * (q_target_ - q_predict_)
            if flag == True:
                q_target_ = r + self.gamma * max((self.qTable.loc[s_] + k / (1 + self.cTable
                                                              .loc[s_].sum())))
                self.qTable_.loc[s, a] += self.alpha * (q_target_ - q_predict_)
        else:
            q_target_ = r
            self.qTable_.loc[s, a] += self.alpha * (q_target_ - q_predict_)

        if not done:
            q_target = r + self.gamma * max((self.qTable.loc[s_] + k / (1+self.cTable.loc[s_
                                                          ].sum())))
        else:
            q_target = r

        self.cTable.loc[s, a] += 1
        self.qTable.loc[s, a] += self.alpha * (q_target - q_predict)

    def state_exist(self, state):
    # if state not exist, initialize all 0
        if state not in self.qTable.index:
            self.qTable = self.qTable.append(pd.Series(
                [0]*len(self.actions),
                index=self.qTable.columns,
                name=state))
        if state not in self.qTable_.index:
            self.qTable_ = self.qTable_.append(pd.Series(
                [0]*len(self.actions),
                index=self.qTable_.columns,
                name=state))

        if state not in self.cTable.index:
            self.cTable = self.cTable.append(pd.Series(
                [0]*len(self.actions),
                index=self.cTable.columns,
                name=state))

class Model:
    '''
    Model(s,a) is utilized to implement model-based RL:
    1. Learn from interaction
    2. Update Q(s,a) after each learning process
    '''
    def __init__(self, actions):
        self.actions = actions
        self.storage = pd.DataFrame(columns=actions, dtype=np.object)

    def store_transition(self, s, a, r, s_):
        if s not in self.storage.index:
            self.storage = self.storage.append(
                pd.Series([None] * len(self.actions),
                index = self.storage.columns,
                name = s)
            )

        self.storage.loc[s, a] = (r, s_)

    def sample_s_a(self):
        s = np.random.choice(self.storage.index)
        a = np.random.choice(self.storage.loc[s].dropna().index)
        return s, a

    def get_r_s_(self, s, a):
        r, s_ = self.storage.loc[s, a]
        return r, s_

# global variables storing last iteration's information
np.random.seed(100)
    ### END CODE HERE ###
```

III. REINFORCEMENT LEARNING ON ATARI GAME

A. Method

Later Reward This is an important thing to be noticed. When you choose an action to rebound the ball, the reward appears 30-38 steps afterwards. Therefore, there exists a delay. We use afterwards reward, and add up the reward between 30-38 step behind for a single action.

Pre-training We acquire this idea from our Pro. Yue Gao. We first play atari on ourselves and use the data to train our agent supervisely at the beginning. This is a lot better than randomly pre-trained.

Meaningful-training We do not use all acquired training data to update the fixed-Q target. We use agent which can survive more steps to tune our agent.

Less Features We use 128 inputs instead of 512 inputs. This is the same principle as dropout.

Tensorboard Visualize We use tensorboard to monitor our agent behavior dynamically.

Less Rendering When training, we do not use environment rendering to save time.

B. Tuning Process

For Later Reward We use a fixed length(=38) list to store $(s, a, r, s^{'}, done)$. For each $(s, a, s^{'}, done)$, $r_{\_} = sum(r[30:38])$

For Pre-training My parterner Chi Zhang plays the game for rewards better than 10 and move to target step by step, which makes it easy for agent to learn.

For Meaningful-training We tried several times to observe the relationship between rewards acquired and steps. Finally, we choose episode with steps more than 300 for further training.

For Less Features We use 128 input instead of 512.

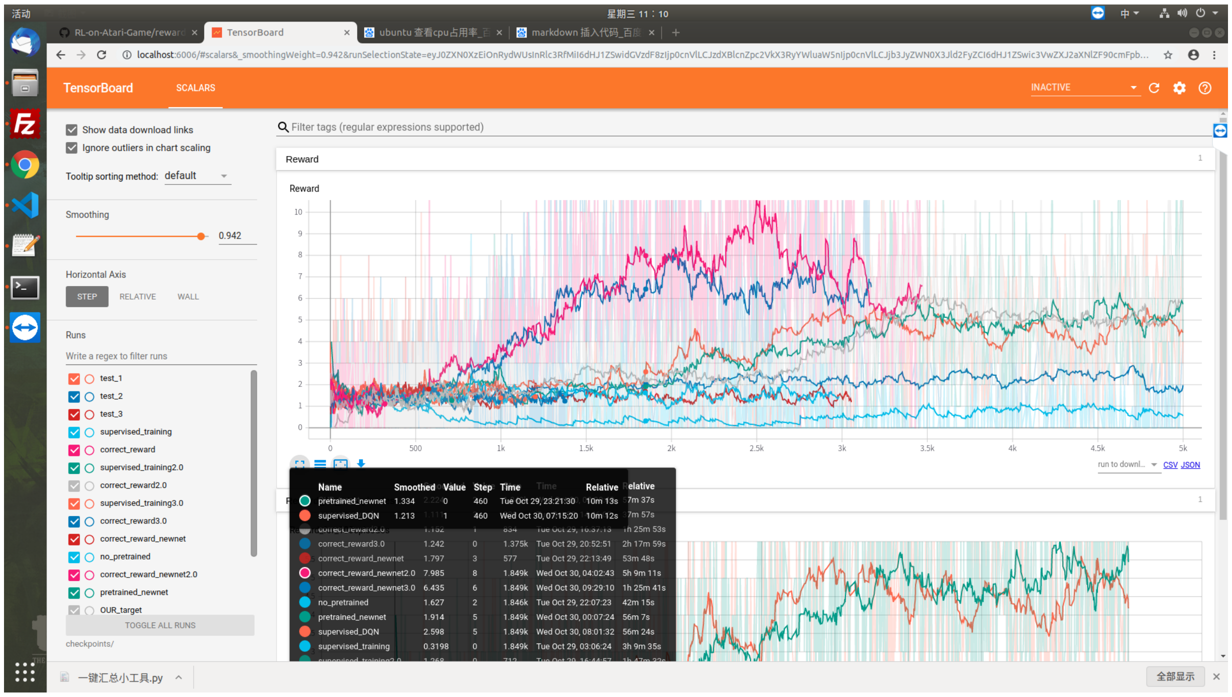
For Other Hyperparameters We increase randomness at the vary beginning, and more layers of our NN. Besides, we use a better optimizer Adam.

1. *Trial 1: pre-train the model with labeled data*

We played the atari step by step and saved the actions and states during the process into a csv file. We played about 5000 steps and at last we select 500 steps as our labeled dataset.

At the beginning of the train, we use the label dataset as ground truth to supervised the training of model, the function is written as:

```
    def train_with_gt(self):
        csv_file=open('labeled_dataset.csv')
        csv_reader_lines = csv.reader(csv_file)
        for line in csv_reader_lines:
            line = list(map(int, line))
            update_input = np.array(line[:-4]).reshape(1,128)
            update_target = np.array(line[-4:]).reshape(1,4)
            self.evaluation_model.fit(update_input, update_target, batch_size=1, epochs=1,
                                                   verbose=0)
```



Show data download links ✓
Ignore outliers in chart scaling ✓
Tooltip sorting method: default
Smoothing 0.942
Horizontal Axis: [✓ STEP] RELATIVE WALL

Runs: ✓ test_1, ✓ test_2, ✓ test_3, ✓ supervised_training, ✓ correct_reward, ✓ supervised_training2.0, ✓ correct_reward2.0, ✓ supervised_training3.0, ✓ correct_reward3.0, ✓ correct_reward_newnet, ✓ no_pretrained, ✓ correct_reward_newnet2.0, ✓ pretrained_newnet, ✓ OUR_target

checkpoints/

	Name	Smoothed	Value	Step	Time	Relative
●	pretrained_newnet	1.334	0	460	Tue Oct 29, 23:21:30	10m 13s
●	supervised_DQN	1.213	1	460	Wed Oct 30, 07:15:20	10m 12s

	Name	Smoothed	Value	Step	Time	Relative
●	correct_reward2.0	1.152	1	834	Tue Oct 29, 16:37:13	1h 25m 53s
●	correct_reward3.0	1.242	0	1.375k	Tue Oct 29, 20:52:51	2h 17m 59s
●	correct_reward_newnet	1.797	3	577	Tue Oct 29, 22:13:49	53m 48s
●	correct_reward_newnet2.0	7.985	8	1.849k	Wed Oct 30, 04:02:43	5h 9m 11s
●	correct_reward_newnet3.0	6.435	6	1.849k	Wed Oct 30, 09:29:10	1h 25m 41s
●	no_pretrained	1.627	2	1.846k	Tue Oct 29, 22:07:23	42m 15s
●	pretrained_newnet	1.914	5	1.849k	Wed Oct 30, 00:07:24	56m 7s
●	supervised_DQN	2.598	5	1.849k	Wed Oct 30, 08:01:32	56m 24s
●	supervised_training	0.3198	0	1.849k	Tue Oct 29, 03:06:24	3h 9m 35s

FIG. 4: Ablation Study we made

The pre-train method can improve the performance at the beginning of training, however, it has a bad effects for the latter training. That is because the labeled data is not accurate enough and it will miss lead the training. So this method is not involved in our final training.

2. *Trial 2: later reward*

C. Outcome

Required Environment

- Ubuntu 18.04
- tensorflow-gpu
- CUDA 10
- tensorboard

How to Train: run the following command to start the training:

```
python atariDQN.py  --name TestName
```

the data will be saved in *checkpoints* folder, which includes the model named *my_model.h5* and files that are used in tensorboard for visualization. You can check the training process with command:

```
tensorboard --logdir=checkpoints/
```

and visualize the training progress by opening https://localhost:6006 on your browser.

Our final outcomes is reward 16.0 for 2.1k episodes , which can be seen in Figure 5

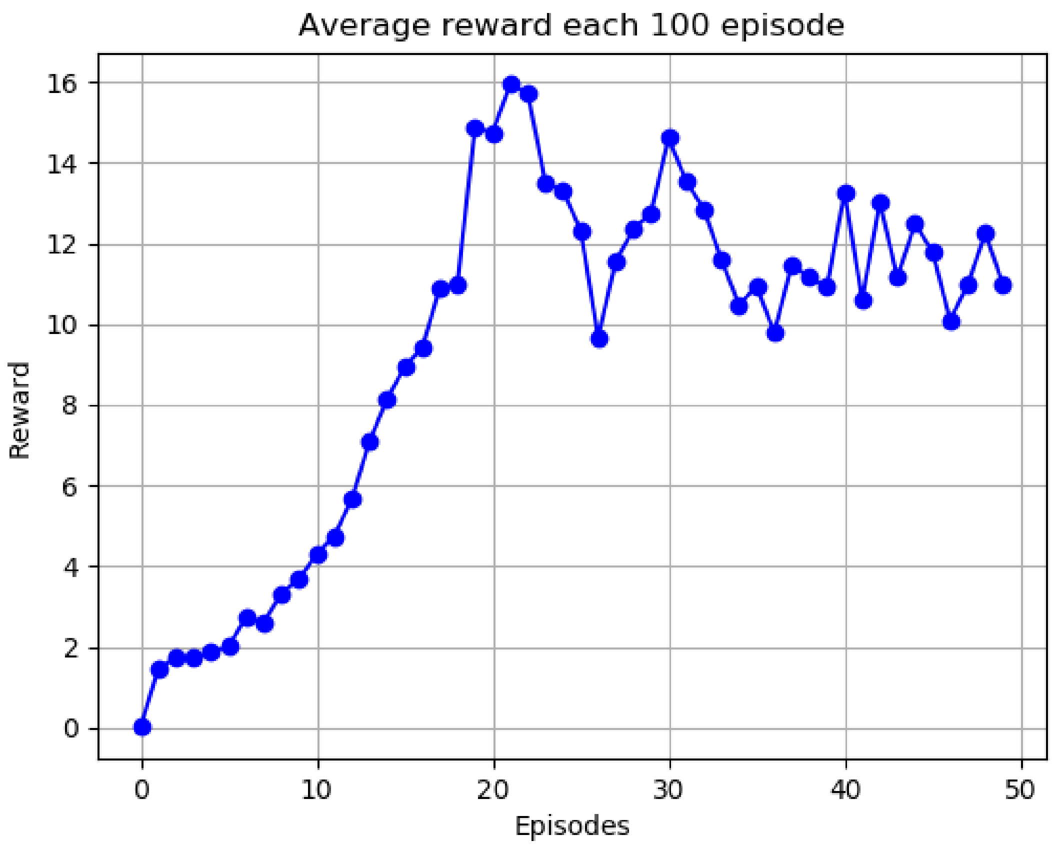


FIG. 5: Ablation Study we made

D. source code

```
import argparse
import sys
import os
from path import Path
import copy


import random
import numpy as np
import pandas as pd
import tensorflow as tf
import gym
from collections import deque
from keras import optimizers
from keras.models import load_model
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Activation, Flatten, Conv1D, MaxPooling1D,Reshape
import matplotlib.pyplot as plt
from tensorboardX import SummaryWriter
from tqdm import tqdm
from queue import Queue
'''
parser = argparse.ArgumentParser(description='DQN for pixel game',
                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--pretrained-model', dest='pretrained_model', default=None, metavar='
                                             PATH',
                    help='path to pre-trained net model')
parser.add_argument('--name', dest='name', type=str, default='demo', required=True,
                    help='name of the experiment, checpoints are stored in checpoints/name')
'''
class DQN:
    ### TUNE CODE HERE ###
    def __init__(self, env):
        """
        env = gym.make('Breakout-ram-v0')
        env = env.unwrapped
        """
        self.env = env
        self.memory = deque(maxlen=400000) #
        self.gamma = 0.9
        self.epsilon = 1.0
        self.epsilon_min = 0.005
        self.epsilon_decay =  self.epsilon / 1000000 #

        self.batch_size = 32
        self.train_start = 20000
        self.state_size = self.env.observation_space.shape[0]
        self.action_size = self.env.action_space.n
        self.learning_rate = 0.00025

        self.evaluation_model = self.create_model()
        self.target_model = self.create_model()

    def create_model(self):
        model = Sequential() # The Sequential model is a linear stack of layers.
        model.add(Dense(128, input_dim=self.state_size,activation='relu'))
        model.add(Dense(128*4, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(64, activation='relu'))
        model.add(Dense(self.env.action_space.n, activation='linear'))# self.env.
                                                   action_space.n = 4
        model.compile(loss='mean_squared_error', optimizer=optimizers.Adam(lr=self.
                                                   learning_rate))
        return model

    def choose_action(self, state, steps):
        if steps > 10000:
            if self.epsilon > self.epsilon_min:
                self.epsilon -= self.epsilon_decay
        if np.random.random() < self.epsilon: #
                                                                           epsilon

            return self.env.action_space.sample()
            # self.evaluation_model.predict(state)                  4

        #                       a
        return np.argmax(self.evaluation_model.predict(state)[0])

    def remember(self, cur_state, action, reward, new_state, done):
        '''

        '''
        if not hasattr(self, 'memory_counter'):
            self.memory_counter = 0
        transition = (cur_state, action, reward, new_state, done)
        self.memory.extend([transition]) # add to    r    i    g    h    t
        # ([(1, 2, 3), (2, 3, 4), (10, 11, 12)], maxlen=400000)
        self.memory_counter += 1

    def replay(self):
        '''

        '''
        if len(self.memory) < self.train_start:
            #                    1000
            return

        mini_batch = random.sample(self.memory, self.batch_size) #
                                                                                      (
                                                              cur_state, action, reward, new_state,
                                                              done)

        update_input = np.zeros((self.batch_size, self.state_size))
        update_target = np.zeros((self.batch_size, self.action_size))

        for i in range(self.batch_size):
            state, action, reward, new_state, done = mini_batch[i]
            target = self.evaluation_model.predict(state)[0]

            if done:
                target[action] = reward
            else:
                target[action] = reward + self.gamma * np.amax(self.target_model.predict(
                                                                      new_state)[0])

            update_input[i] = state
            update_target[i] = target

        self.evaluation_model.fit(update_input, update_target, batch_size=self.batch_size,
                                                              epochs=1, verbose=0)

    def target_train(self):
        self.target_model.set_weights(self.evaluation_model.get_weights())
        return

    def binary_encoding(self,decimal_state):
        binary_state = [np.binary_repr(x,width=8) for x in decimal_state[0]]
        parameter_list = []
        for parameter in binary_state:
            parameter_list.append(int(parameter[:2],2))
            parameter_list.append(int(parameter[2:4],2))
            parameter_list.append(int(parameter[4:6],2))
            parameter_list.append(int(parameter[6:],2))
        return np.array(parameter_list)

    def visualize(self, reward, episode):
        plt.plot(episode, reward, 'ob-')
        plt.title('Average reward each 100 episode')
        plt.ylabel('Reward')
        plt.xlabel('Episodes')
        plt.grid()
        plt.show()
    ### END CODE HERE ###


def main():
    '''
    global args
```

```
args = parser.parse_args()
save_path = Path(args.name)
save_path = 'checkpoints'/save_path #/timestamp
save_path.makedirs_p()
training_writer = SummaryWriter(save_path)
'''

env = gym.make('Breakout-ram-v0')
env = env.unwrapped

episodes = 5000
trial_len = 10000

tmp_reward=0
sum_rewards = 0

graph_reward = []
graph_episodes = []

dqn_agent = DQN(env=env)

####### Training ######
### START CODE HERE ###
dqn_agent.create_model()
current_step = 0
reward_in_episode_100 = 0.0
index100 = 0
for episode in range(episodes):
    prev_state = env.reset().reshape(1,128)
    prev_state = prev_state/128.0-1.0

    # prev_state = dqn_agent.binary_encoding(prev_state).reshape(1,512)
    # print(prev_state)
    prev_state_, action_,reward_,current_state_,done_ = [],[],[],[],[]
    reward_in_episode = 0
    update_count = 0
    prev_lives = {"ale.lives":5}
    lives = {}

    print('episode:',episode)
    for i in range(trial_len):
        # env.render()
        action = dqn_agent.choose_action(prev_state,current_step)

        #action = input()
        #action = np.int64(action)
        current_state, reward, done, lives = env.step(action)  #  [1,128]
        if (prev_lives['ale.lives']>lives['ale.lives']):
            prev_lives = copy.deepcopy(lives)
            action = np.int64(1)
            current_state, reward, done, lives = env.step(action)
        # print(_)
        current_state = current_state.reshape(1, 128)
        current_state = current_state/128.0-1.0
        # current_state = dqn_agent.binary_encoding(current_state).reshape(1,512)

        current_step += 1
        update_count += 1
        reward_in_episode += reward
        reward_in_episode_100 += reward
        # print("*****************",reward)
        if i < 33:
            prev_state_.append(copy.deepcopy(prev_state))
            action_.append(action)
            reward_.append(reward)
```

```
                current_state_.append(copy.deepcopy(current_state))
                done_.append(done)
            else:
                prev_state_.append(copy.deepcopy(prev_state))
                action_.append(action)
                reward_.append(reward)
                current_state_.append(copy.deepcopy(current_state))
                done_.append(done)

                reward = np.sum(reward_[30:35])

                dqn_agent.remember(copy.deepcopy(prev_state_[0]), action_[0], reward,copy.
                                                                       deepcopy( current_state_[0]),
                                                                       done_[0])
                prev_state_.remove(prev_state_[0])
                action_.remove(action_[0])
                reward_.remove(reward_[0])
                current_state_.remove(current_state_[0])
                done_.remove(done_[0])

            if  current_step%4==0:
                dqn_agent.replay()


            prev_state = copy.deepcopy(current_state)


            if done:
                if(update_count>=300) or current_step%2500==0:
                    dqn_agent.target_train()
                # env.render()
                break


        # if episode%500 ==0:
            # dqn_agent.target_model.save(save_path + '/my_model.h5')

        # visualization
        # training_writer.add_scalar('Reward', reward_in_episode, episode)
        if episode%100 ==0:
            # training_writer.add_scalar('Reward 100', reward_in_episode_100/100, index100)
            graph_reward.append(reward_in_episode_100/100)
            print("average reward in 100 episodes:", reward_in_episode_100/100)
            reward_in_episode_100 = 0.0
            graph_episodes.append(index100)

            index100+=1

        del prev_state_, action_,reward_,current_state_,done_


    ### END CODE HERE ###
    # dqn_agent.target_model.save(save_path + '/my_model.h5')
    dqn_agent.visualize(graph_reward, graph_episodes)

if __name__ == '__main__':
    main()
```

IV. CONCLUSION

I really have learned a lot from the two problems concerning reinforcement learning.

In the maze part, I implement dyna-Q learning, and also absorb some efficient and effective algorithms from the paper.

In the atari part, my partner Chi Zhang and I have learned the DQN and have a general idea of the combination between dyna-Q learning and deep neural network.

I appreciate TAs and my professor for issuing such a meaningful homework3.