## I. INTRODUCTION

### A. Purpose

This part is completed by Chai Zihao and Xu yanxuan. The goal of this homework is to implement a DQN agent to play atari game pacman and adjust itself by training to get higher scores.It will exemplify the DQN algorithm.

Pacman is one of the classic and leading games. We need to guide the pac-Man to eat all the dots and avoid the ghosts. In this assignment, we are asked to design a DQN agent to learn control policies directly from the visual information of the game.


(a) MsPacman-ram-v0 gym

As shown in the figure, the MsPacman-ram-v0 gym environment is utilized as the training environment. This environment provides the ram(128 bytes) of the atari console as model input. Each time, the agent should choose an action from 9 available actions, corresponding to the 8 buttons on the handle and "do nothing".

### B. Environment

There is a minimal amount of equipment to be used in this homework. A few requirements are listed below:

- Python 3.7
- Library: gym, tensorflow, keras ,collections, numpy, math
- Complier: Pycharm 2020.2
- Server: Baidu Ai Studio, Ali Tianchi

## II. IMPLEMENTATION

### A. DQN algorithm

DQN is an approximate value function obtained by neural network. Input a state, to network then get an output Q(s,a), and use $-$greedy strategy to make decisions by output action. Then Update the parameters of the function network according to the reward, and loop the above process until a good function network is trained well.

DQN separates the whole framework into a target network and an eval network. One is Q network which is used to update the Q value synchronously, the other is the target network used to calculate the target Q value. The weight is synchronized to the target network.

The following is Dyna-Q algorithm's pseudo-code.

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

(b) DQN-Q learning

Reinforcement learning is not exactly the same as supervised learning or unsupervised learning. Each sample in supervised learning has a one-to-one corresponding label, but unsupervised learning does not. Reinforcement learning has a special label-(reward). As the game progresses, these labels have a certain connection in the time dimension. It is based on these rewards that the agent can learn to perform corresponding operations in various situations to achieve the goal.

### B. Improvement of DQN

1. Adjust the hyper-parameters

   In order to achieve a better score, we adjust many parameters, such as discount_factor, learning_rate, memory_maxlen and so on. In the end, we choose the parameters with better performance and more reasonable to submit.

2. Network Improvement

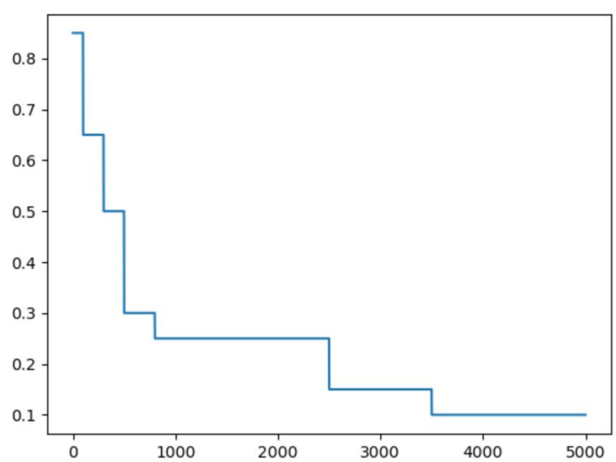
   In order to obtain a prominent training result, we design a effective neural network for the training process. This network is consist of three kinds of layers. The first is input layer, which only contain one

Dense layer to pre-process the input data. What links to input layer is hidden layer before a Batch-Normalization layer, which contains three Dense layers to extract features of input and do a regression operation for training. After that, the data from the output of hidden layer will pass through the last output layer constituted by one Dense layer. The following is the framework of this network:

$$\begin{gathered} Input \\ \downarrow \\ Dense \quad Layer \\ \downarrow \\ BatchNormalization \quad Layer \\ \downarrow \\ Dense \quad Layer \\ \downarrow \\ Dense \quad Layer \\ \downarrow \\ Dense \quad Layer \\ \downarrow \\ Dense \quad Layer \end{gathered}$$

3. Epsilon Greedy Algorithm

   The most important thing for the epsilon-greedy algorithm is the epsilon attenuation strategy. After trying a variety of different methods and strategies, we finally choose the Piecewise function, which decreases rapidly at the beginning and slowly decreases afterwards to obtain better results.




(c) Epsilon Algorithm

4. Death penalty

   To get a better score, we first need to make Pac-Man survive. Therefore, we chose to add the death penalty. When Pac-Man dies, the reward of this action becomes a relatively large negative number (-100). In this way, the agent will survive longer and get a higher score

5. Normalization of the model input

   We used BatchNormalization() to normalize the model input. In the process of deep neural network training, normalization can make the input of each layer of neural network maintain the same distribution.In addition, batchNormalization can also greatly increase the training speed and accelerate the convergence process.

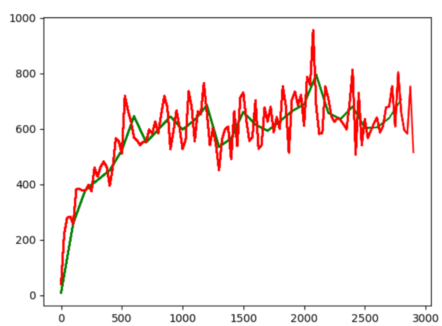
6. Some other idea of failure

   In addition, we tried to make many improvements, but in the end none of them achieved good results.

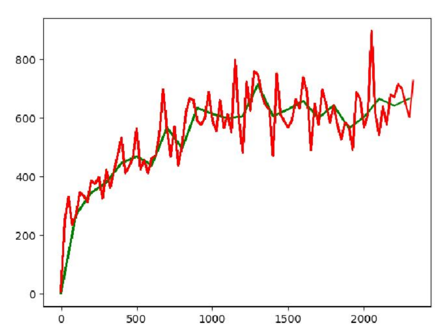
   - Modify the network optimizer to RMSProp, SGD, etc.
   - Imitate two fully connected network and two convolutional layers from papers in the Nature.
   - Join a deeper network and modify the activation function of each layer of the network.
   - Adjust the batch_size of the network.

### C. Result

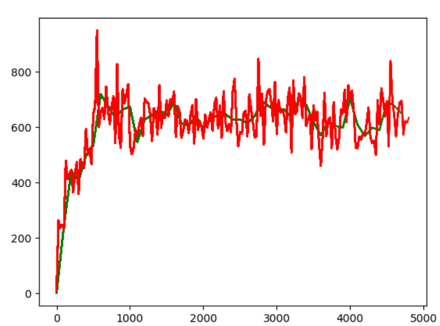
Here are some figures of us. As we can see, after tyring to train the agent by different parameters and model frameworks, finally the score of the agent can easily reach 600 within 1000 episodes, but then its rising speed begins to slow down and scores begin to oscillate.



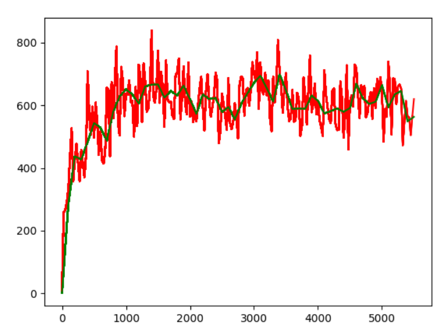

(d) lab1




(e) lab2




(f) lab3




(g) lab4

## III. CONCLUSION

We complete the remaining part of the code and improved on the basis. From the test in our PC and server, we found our code could get 600 scores easily within 1000 episodes.However, there's still some unsolved problem to be improved.

What's more, I think the lab spend us too much time to wait for the result.We try to use Baidu Ai Studio and Ali Tianchi Server to test multiple sets of parameters at the same time, but even so, 5000 episodes require a PC or server about 6 hours of computing time. I think an experimental subject that can see the results of the operation in a shorter time or a high-performance server can help us better understand knowledge and save time at the same time.

Now is the ending time, first we want to thank for the help from teaching assistants and our professor Yue Gao. Through this homework, we have attained lots of skills and experiences. We have almost mastered the DQN algorithm and learned some deep learning tuning techniques. The heuristic function give us great inspiration. We gain a lot.

I have added an adjunct of source codes with this test report, Thanks again!

## IV. EXPERIMENT CODE

This section contains our code block of this homework.

atariDQN code:

```
# -*- coding:utf-8 -*-
# DQN homework.
import os
import sys
import gym
import pylab
import random
import numpy as np
import math
from collections import deque
from keras.layers import Dense
from keras.optimizers import Adam
from keras.models import Sequential
from gym import wrappers
from utils import *
import tensorflow as tf

from keras.layers import Reshape
from keras.layers import Dense, BatchNormalization, Conv2D, Flatten, Conv1D,
    MaxPooling2D, MaxPooling1D, Permute

# hyper-parameter.
EPISODES = 5000


class DQNAgent:
    def __init__(self, state_size, action_size):
        # if you want to see MsPacman learning, then change to True
        self.render = False

        # get size of state and action
        self.state_size = state_size
        self.action_size = action_size

        # These are hyper parameters for the DQN
        self.discount_factor = 0.9   # 0.5,0.90
        self.learning_rate = 0.05   # 0.1£¬0.05
        self.batch_size = 128
        self.train_start = 1000

        # create replay memory using deque
        self.maxlen = 40000
        self.memory = deque(maxlen=self.maxlen)

        # create main model
        self.model_target = self.build_model()
        self.model_eval = self.build_model()

    # approximate Q function using Neural Network
    # you can modify the network to get higher reward.
    def build_model(self):
        model = Sequential()
```

```
        model.add(Dense(256, input_dim=self.state_size, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(BatchNormalization())
        model.add(Dense(256, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(256, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(256, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(self.action_size,
                        kernel_initializer='he_uniform'))
        model.summary()
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        return model

    # get action from model using epsilon-greedy policy
    def get_action(self, state, episode):

        if episode < 100:
            self.epsilon = 0.85
        elif episode < 300:
            self.epsilon = 0.65
        elif episode < 500:
            self.epsilon = 0.50
        elif episode < 800:
            self.epsilon = 0.30
        elif episode < 2500:
            self.epsilon = 0.25
        elif episode < 3500:
            self.epsilon = 0.15
        else:
            self.epsilon = 0.10

        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        else:
            q_value = self.model_eval.predict(state)
            return np.argmax(q_value[0])

    # save sample <s,a,r,s'> to the replay memory
    def append_sample(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    # pick samples randomly from replay memory (with batch_size)
    def train_model(self):
        if len(self.memory) < self.train_start:
            return
        batch_size = min(self.batch_size, len(self.memory))
        mini_batch = random.sample(self.memory, batch_size)

        update_input = np.zeros((batch_size, self.state_size))
        update_target = np.zeros((batch_size, self.state_size))
        action, reward, done = [], [], []

        for i in range(self.batch_size):
            update_input[i] = mini_batch[i][0]
            action.append(mini_batch[i][1])
            reward.append(mini_batch[i][2])
            update_target[i] = mini_batch[i][3]
```

```
            done.append(mini_batch[i][4])

        target = self.model_eval.predict(update_input)
        target_val = self.model_target.predict(update_target)

        for i in range(self.batch_size):
            # Q Learning: get maximum Q value at s' from model
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + self.discount_factor * (
                    np.amax(target_val[i]))

        # and do the model fit!
        self.model_eval.fit(update_input, target, batch_size=self.batch_size,
                            epochs=1, verbose=0)

    def eval2target(self):
        self.model_target.set_weights(self.model_eval.get_weights())


if __name__ == "__main__":
    # load the gym env
    env = gym.make('MsPacman-ram-v0')
    # set random seeds to get reproduceable result(recommended)
    set_random_seed(0)
    # get size of state and action from environment
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    # create the agent
    agent = DQNAgent(state_size, action_size)
    # log the training result
    scores, episodes = [], []
    graph_episodes = []
    graph_score = []
    avg_length = 20
    sum_score = 0

    # train DQN
    for episode in range(EPISODES):
        done = False
        score = 0
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        lives = 3
        step = 0
        while not done:
            dead = False
            while not dead:
                step = step + 1
                # render the gym env
                if agent.render:
                    env.render()
                # get action for the current state
                action = agent.get_action(state, episode)

                # take the action in the gym env, obtain the next state
                next_state, reward, done, info = env.step(action)
                next_state = np.reshape(next_state, [1, state_size])
```

```python
                # judge if the agent dead
                dead = info['ale.lives'] < lives
                lives = info['ale.lives']

                # update score value
                score = score + reward

                # add penalty factor for dead
                if dead:
                    reward = -100

                # save the sample <s, a, r, s'> to the replay memory
                agent.append_sample(state, action, reward, next_state, done)

                # train the evaluation network
                if step % 500 == 0:
                    agent.eval2target()
                # go to the next state
                state = next_state
                # update the target network after some iterations.
                if step % 4 == 0 or dead or reward > 5:
                    agent.train_model()

        # print info and draw the figure.
        if done:
            scores.append(score)
            sum_score += score
            episodes.append(episode)
            # plot the reward each episode
            # pylab.plot(episodes, scores, 'b')
            print("episode:", episode, "  score:", score, "  memory length:",
                  len(agent.memory), "  epsilon:", agent.epsilon, "  step", step)
        if episode % avg_length == 0:
            graph_episodes.append(episode)
            graph_score.append(sum_score / avg_length)
            sum_score = 0
            # plot the reward each avg_length episodes
            pylab.plot(graph_episodes, graph_score, 'r')
            pylab.savefig("./pacman_avg.png")
        # save the network if you want to test it.
```