breakout DQN

李謦伊

1. Describe my DQN

Part 1: Define the network.

agent = DQN(env)

Part 2: Define state and reshape to (1,128)

Part 3: Choose action

Part 4: Storage state , action , reward, next_state , done

```
def main():
    env = gym.make('Breakout-ram-v0')
    agent = DQN(env)
    Reward = []
    Episodes = []
    cost_time = []
    Total_reward = []

    for episode in range(1,EPISODE+1):
        tStart = time.time()
        total_reward = 0
        total_reward_100 = 0

        # initialize task
        state = env.reset().reshape(1,128)
        for step in range(STEP):
            #env.render()
            action = agent.get_action(state)
            next_state,reward,done,_ = env.step(action)
            next_state = next_state.reshape(1,128)

            total_reward += reward

            agent.percieve(state, action, reward, next_state, done, episode)

            state = next_state

            if done:
                break
        #print ('Episode:', episode, 'Total Point this Episode is:', total_reward )
        Total_reward.append(total_reward)

        if episode % 100 == 0:
            total_reward_100 = sum(Total_reward[episode-100:episode])
            Episodes.append(episode)
            Reward.append(total_reward_100 / 100)

            print ('100 Episodes:', episode / 100  , 'Avg Reward of last 100 episodes:', total_reward_100 / 100)


        tEnd = time.time()
        cost_time.append(tEnd - tStart)
        if episode % 100 == 0:
            print('cost time:', sum(cost_time))
            print('----------------------------------------')

    agent.plot_reward(Episodes , Reward)
    print('mean_cost time:',np.mean(cost_time))

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

DQN :

(i) Define the parameters

```python
def __init__(self, env):
    self.env = env
    self.lr = learning_rate
    self.gamma = gamma
    self.epsilon = initial_epsilon
    self.epsilon_min = final_epsilon
    self.epsilon_decay= epsilon_decay

    # init replay memory
    self.replayMemory = deque()
    # init some parameters
    self.batch_size = batch_size  #每次更新時從memory裡取多少記憶出來
    self.memory_size = memory_size  # 記憶上限
    self.action_dim = self.env.action_space.n
    self.state_dim = self.env.observation_space.shape[0]

    self.learn_steps = 0  # 用來控制什麼時候學習
    self.replace_target_iter = replace_target_iter  # 更換 target_net的步數

    self.evaluate_model = self.create_network()
    self.target_model = self.create_network()
```

```python
# Hyper Parameters
OBSERVE = 25000. # timesteps to observe before training
EXPLORE = 750000. # frames over which to anneal epsilon
TRAIN = 24000000.
initial_epsilon = 1.0
final_epsilon = 0.1
epsilon_decay= (initial_epsilon - final_epsilon) / EXPLORE
learning_rate = 0.00025
memory_size = 200000
batch_size = 32
gamma = 0.99
EPISODE = 5000
STEP = int((OBSERVE + EXPLORE + TRAIN)/ EPISODE)
replace_target_iter = 2500
```

(ii) Create the network

I established five hidden layers and output layer.

Among of them, the number of neurons in the first hidden layer I set to **128**, “**relu**” as an activation function and kernel_initializer is **random_normal**.

The second hidden layer is **64 units** , third hidden layer is **32 units** , fourth hidden layer is **16 units** , fifth hidden layer is **8 units**.

And their activation function are “**relu**”, kernel_initializer are **random_normal**.

The number of neurons in the output layer I set to **action_dim** (action_dim =4) , “**linear**” as an activation function and kernel_initializer is **uniform**.

Finally, I used “**mean_squared_error**” as loss function and optimizer is **Adam** (the parameters are learning rate , epsilon=1e-6) .

```
def create_network(self):

    model = Sequential()

    model.add(Dense(units=128,input_dim=self.state_dim, activation="relu",kernel_initializer='random_normal'))
    model.add(Dense(units=64, activation="relu",kernel_initializer='random_normal'))
    model.add(Dense(units=32, activation="relu",kernel_initializer='random_normal'))
    model.add(Dense(units=16, activation="relu",kernel_initializer='random_normal'))
    model.add(Dense(units=8, activation="relu",kernel_initializer='random_normal'))
    model.add(Dense(self.action_dim, activation="linear",kernel_initializer='uniform'))

    model.compile(loss="mean_squared_error", optimizer=tf.train.AdamOptimizer(self.lr,epsilon=1e-6))

    return model
```

(iii) Choose action

First, define action is None , then :

- If we randomly select a number that is less than epsilon, we randomly select the action,
- Else , predict the reward value based on the current state and choose the action that will give the highest reward.

Second, decreasing the epsilon.

If epsilon > epsilon_min(final_epsilon) and learn_step > our obserse , we will decreasing the epsilon.

And our epsilon decay is (initial_epsilon – final_epsilon) / explore .

```
def get_action(self, state):

    action = None

    if np.random.random() < self.epsilon:
        action = self.env.action_space.sample()
    else:
        action = np.argmax(self.evaluate_model.predict(state)[0])

    if self.epsilon > self.epsilon_min and self.learn_steps > OBSERVE:
        self.epsilon -= self.epsilon_decay


    return action
```

(iv) Train Q-value

First, check if the target_net parameter is replaced.

If learn_step % 2500 = 0 , we update the target_net parameter. (In order to reduce the time our train.)

Second,

Step 1: obtain random minibatch from replay memory.

And setting the zero matrix of the update input and target.

Step 2: calculate y (reward)

- If the game is done , y = reward
- Else, y = reward + gamma * max(target_Q)

Step 3: Train evaluate model. (update input and update target)

```
def train_Q_network(self):  ##Learn
    #檢查是否替換 target_net參數
    if self.learn_steps % self.replace_target_iter == 0:
        self.target_model.set_weights(self.evaluate_model.get_weights())

    # Step 1: obtain random minibatch from replay memory
    minibatch = random.sample(self.replayMemory, self.batch_size)

    update_intput = np.zeros((self.batch_size, self.state_dim))
    update_target = np.zeros((self.batch_size, self.action_dim))

    # Step 2: calculate y
    for i in range(self.batch_size):
        state, action, reward, next_state, done = minibatch[i]
        action = np.where(action == np.max(action)) # 找最好的action
        target = self.evaluate_model.predict(state)[0]  # target Q value for the i-th state
        target_Q = self.target_model.predict(next_state)[0]  # target Q values for the i-th next_state

        # update
        if done:
            target[action] = reward
        else :
            target[action] = reward + self.gamma * np.max(target_Q)

        update_intput[i] = state
        update_target[i] = target

    # Train evaluate model
    self.evaluate_model.fit(update_intput, update_target, batch_size = self.batch_size, epochs =1, verbose=0)
```

(v) Storage state , action , reward, next_state , done

If our memory is full (maximum = memory_size), delete the first memory.

First, Setting the zero matrix to 4 * 4 , in order to store action we choose.

Then storage the formation we got.

Second, if our memory is larger than batch_size(we choose 32) and learn_step % 4 = 0 (In order to reduce the time our train.) and learn_step is larger than our observe , we start to train our network.

Third, Add one to learn_step.

```
def percieve(self, state , action, reward, next_state, done, episode):

    if len(self.replayMemory) == self.memory_size:
        self.replayMemory.popleft()

    one_hot_action = np.zeros(self.action_dim)
    one_hot_action[action] = 1

    self.replayMemory.append((state, one_hot_action , reward , next_state , done))

    if len(self.replayMemory) > batch_size and (self.learn_steps % 4 == 0) and (self.learn_steps > OBSERVE):
        self.train_Q_network()

    self.learn_steps += 1
```

(vi) plot our episode and reward

```
def plot_reward(self,Episodes , Reward):
    plt.plot(Episodes , Reward ,color = 'pink',marker = 'o')
    plt.title('Avg Reward of last 100 episodes')
    plt.ylabel('Reward')
    plt.xlabel('Episodes')
    plt.show()
```

2. The average time I train an episode.

I train an episode time is 3.376 second.

```
mean_cost time: 3.3759921600818634
```

3. The learning curve.