Deep Reinforcement Learning for a video game from OpenAI Gym

Contents:

Step 1: Importing an OpenAI Gym game

The OpenAI Gym offers a set of game simulators that can be used to implement and compare reinforcement learning algorithm without regarding to the structure of agent in game, as well as a good compatibility with another library such and Pytorch or TensorFlow. In this project, we need to import a game of Gym and then play the game by training a deep reinforcement learning model.
To get start, we install the OpenAI Gym

```
!pip install gym
```

then, we install Atari game set from gym library

```
!pip install gym[atari]
```

For rendering the environment, we need to install python virtual display tool

```
!pip install pyvirtualdisplay
!pip install piglet
```

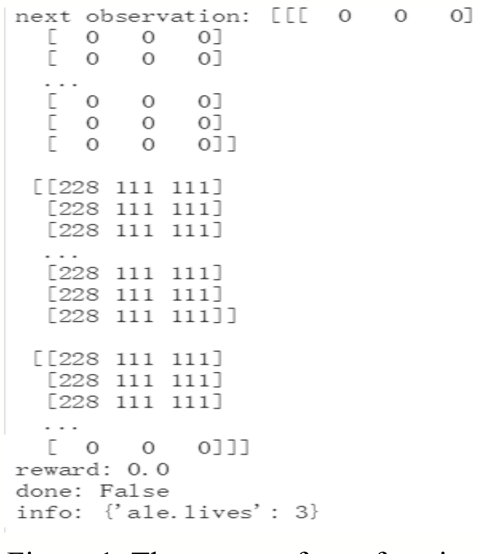
Then we import gym environment which consists of many classical 2-D games, we pick up one. Also, we can reset the environment and get the original observations as follow.

```
import gym
env = gym.make('MsPacman-v0')
env.reset()
```

After initialized game, we can get some attributes of the game by taking randomly one action and fed it in the environment by calling env.step(action), we can know what the action is doing to the environment.

```
action = env.action_space.sample()
obs, r, done, info = env.step(action)
```

As seen from the figure 1, step function returns four values including observations, reward, done and info. Observation shows us what is environment, for instance, pixel data from each frame in board game. Reward represents the amount of reward generated by previous action we taken. Done indicate that whether it is time when we need reset the environment. Info is used for debugging.



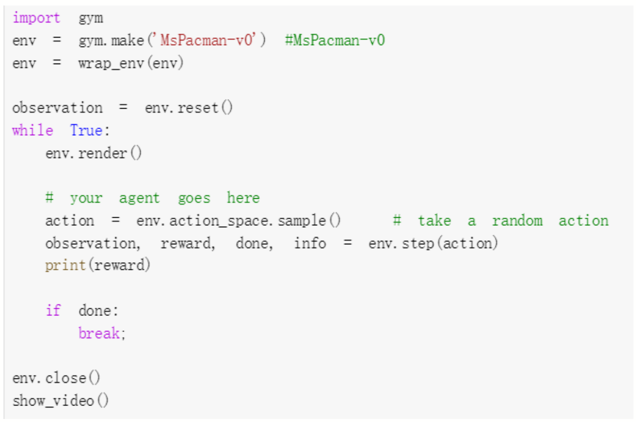
```
next observation: [[[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  0   0   0]
  [  0   0   0]
  [  0   0   0]]

 [[228 111 111]
  [228 111 111]
  [228 111 111]
  ...
  [228 111 111]
  [228 111 111]
  [228 111 111]]

 [[228 111 111]
  [228 111 111]
  [228 111 111]
  ...
  [  0   0   0]]]
reward: 0.0
done: False
info: {'ale.lives' : 3}
```

Figure 1. The output of step function

In addition, observations and actions are defined by two attributes of type, observation_space and action_space in every environment. Observation spaces are multi-dimensional vector of numeric values, and action spaces are a set of discrete numbers that corresponding to each action of an agent (OpenAI, 2020). An example as below:

```
print(env.observation_space)
Box(210, 160, 3)
print(env.action_space)
Discrete(9)
```

As the figure 2 shows, for the sake of illustration for importing game successfully, we can call render function to visualize the interaction process which gets started by calling reset() and then take continually random action until game done.



```
import gym
env = gym.make('MsPacman-v0')  #MsPacman-v0
env = wrap_env(env)

observation = env.reset()
while True:
    env.render()

    # your agent goes here
    action = env.action_space.sample()     # take a random action
    observation, reward, done, info = env.step(action)
    print(reward)

    if done:
        break;

env.close()
show_video()
```

Figure 2. The output of step function

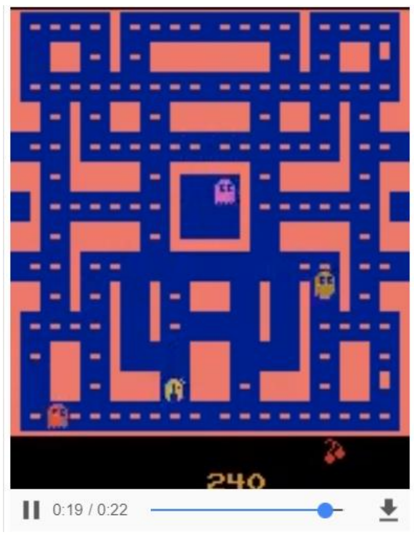


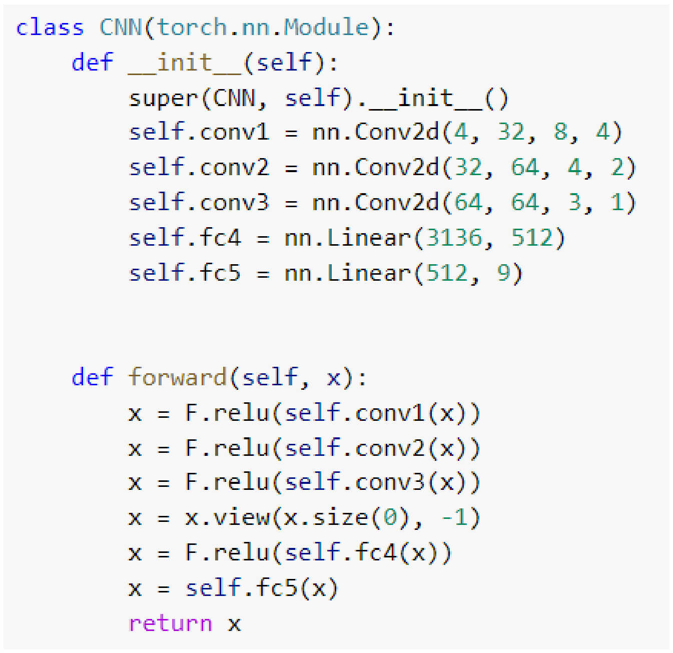
Figure 3. The output of step function

The figure 3 is a screen shot of the game we import. The game is called MsPacman-v0 in which a robot attempts to obtain the highest reward by eating the beans on the way while avoiding the ghosts. So, we have successfully loaded the game of OpenAI Gym.

Step 2: Creating a network

As noted in the above, we attempt to train a reinforcement learning model to play the MsPacman game. Q-learning algorithm can be used to enable our agent get the maxim reward by taking the action with maximum Q value (expected reward of every action at a state, $Q(s_t,a_t)=maxR_{t+1}$), which means Q-learning algorithm will storage Q values on all the explored states and actions and compute the maximum values among these. However, this goal is hard to achieve when playing a game, because there is huge demand on the amount of memory and time, so we would like to get the help of deep neuron network to do this.

We can use torch.nn package in Pytorch language to construct neural networks and use autograd package to differentiate automatically all operations on tensors (like array in Numpy). As the figure 4 shows, the configuration of our CNN based on torch.nn.Module which consist of several layers and a Relu function that returns the computed result.



```python
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(4, 32, 8, 4)
        self.conv2 = nn.Conv2d(32, 64, 4, 2)
        self.conv3 = nn.Conv2d(64, 64, 3, 1)
        self.fc4 = nn.Linear(3136, 512)
        self.fc5 = nn.Linear(512, 9)


    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc4(x))
        x = self.fc5(x)
        return x
```

Figure 4. The configuration of Convolutional Neuron Network

As the table 1 illustrates, our neuron network has the input layer and the output layer, as well as 5 (or 8 if you consider ReLU as layer) hidden layers. The hidden layers consist of three convolutional layers and two full connected layers, ReLU activation function be used in each layer except the last one. In addition, the outputs of each layers can be obtained according to the formula $(N-F)/S + 1$.

Some works about image preprocessing needs to do before imputing. At beginning, we crop the image to the size of 84x84, which lead to removing on irrelative edge areas and reduced the size of images. Moreover, since our game has a low dependency on the color and more attention on the marginal gradients, it is a good way that the cropped images are transferred from RGB (colorful) to grayscale (black-and-white) by calling cv2 of OpenCV, which lead to a decrease on the image dimension and drastic increase on the computation without lack of gradient information. Furthermore, we replicate the image to four copies and get an image of 84x84x4. Finally, feeding the preprocessed image to the network, the neuron network will output 9 values which corresponding to all possible actions of the game. The Graph 5 argue that the input and output of the network.

Hidden layer	**Input**	**Number of filters**	**Filter size**	**Stride**	**Activation**	**Output**
Conv1	84x84x4	32	8x8	4	ReLU	(84-8)/4+1=20, 20x20x32
Conv2	20x20x32	64	4x4	2	ReLU	(20-4)/2+1=9, 9x9x64
Conv3	9x9x64	64	3x3	1	ReLU	(9-3)/1+1=7, 7x7x64
FC1	7x7x64	512			ReLU	512
FC2	512	2			Linear	9

Table1. Size of each stage of Convolutional Neuron Network

Convolutional layer: this layer is the most significant layer of CNN and some learnable filters (or mask or kernel) are made of whose parameters. Convolutional layer do some convolution operation to the input by calling nn.Conv2d(). We use this function with four parameters. The first item is in_channels that represent number of channels of each input sample (3 for images with 3 channels RGB, 1 for gray image). The second item is output_channels that means the number of channels of each output generated by the convolution. The third is kernel_size that shows the size of the convolving kernel. The final parameter is the size of the stride that describes the moving steps of the filers. We should notice that the size of the output of each convolutional layer should be equal to that of its previous layer.

ReLU: the full name is rectified linear unit, it is one of activation functions and introduce the non-linearities that making the neuron network is suitable for all kind of functions rather than only linear ones (Developers, 2020). In addition, ReLU avoid vanishing gradient problem that preventing the change on the weight. The form of ReLu function is ReLU = max(*threshold*, *x*), which means output is 0 when $x < threshold$, otherwise, linear process with slope 1. We can do this operation by calling Function.relu().

Full connection: the function is nn.Linear(), which does an affine operation on the inputs with the form of $y = Wx + b$, W and b represents two parameters of linear regression, weight and bias (similar with function in Perceptron). Similar to convolution layer, the input of the next FC should equal to that of its previous FC.

We can use a forward method to implement the layers that have different functions. The figure 5 illustrates the schematic illustration of our convolution neuron network.

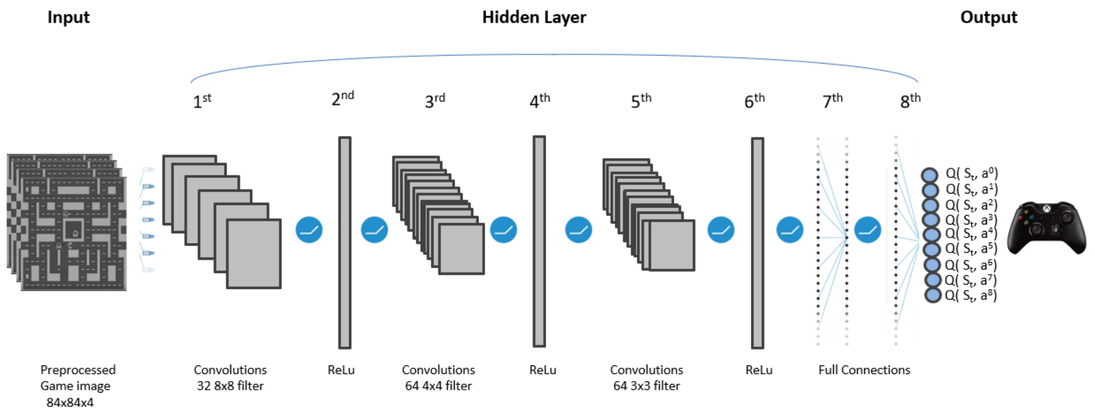


Figure 5. The Schematic illustration of the Convolutional Neuron Network

In conclusion, firstly, we feed the game image as the inputs to our network. Secondly, data experienced a series of hidden layers where convolving the inputs with different number and size of filter, whose goal is to do feature extraction from the images. Thirdly, the output of the previous layer was flattened, for instance, the data with a shape of (5, 5, 32) should be flattened to the size of 5*5*32=800. Finally, fully connecting our neuron network with the function *f(wx+b)*, which result in classification of features generated by previous layers and produced the outputs.

The network we design is a deep neuron network, because it has more than one hidden layer. We feed the image as the inputs *x* to the network, and an output *y* can be obtained. The learning process is occurring in the hidden layers and it use backward propagation algorithm to achieve the parameters' learning. Backward propagation contains forward pass and backward pass. The forward pass means that we create a network using forward method in which the input *x* be given and the output *y* be outputted. The backward pass means that we compute the loss between the prediction and the ground truth and then doing the gradient descending on the update function, from which the learnable parameters are tuned with respect to the loss. Below is the code showing how to do this, we use mean squared error as loss function and select stochastic gradient descent algorithm as optimizer to minimizes the loss function.

```
criterion = nn.MSELoss()
optimizer = optim.SGD(policy_net.parameters(), lr=0.001, momentum=0.9)
loss = criterion (x, y)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Step 3: Connection of the game to the network

To begin, when starting to play the game, we need to initialize it's environment by calling the environment's reset() function state = env.reset()., which returns the value of original observation. As noted in the above, observation means the state of the environment which shows the next image of game when taking an action, so we can feed the state as input to the network.

Moreover, after feeding the state to the network by calling net:

act_values = self.net(state), the final output produced by the above cnn network contains 9 values from Q(S_t, a^0) to Q(S_t, a^8), which corresponding to 9 actions of the agent in the game. Therefore, we can know that the agent's next action is produced by our network.

Furthermore, we feed the states and next_states to the neuron network and optimize the DQN model for the sake of high scores in the future.

Step 4: Deep reinforcement learning model

In this project, we implemented a Deep Q-Network (DQN) for the game. DQN is a model that combining the reinforcement learning and deep algorithm. In reinforcement learning, an agent of an environment attempts to maximizing some sort of reward. The agent adopts an action to the environment and acquired the reward correspond to that change. Then it goes to new state and take next action, repeating the process until the environment ends (Grigsby, 2018). We call this decision-making loop Markov decision process (MDP). As we said before, when Q-learning algorithm in reinforcement learning be used to play a game, there are necessary demands on a large amount of memory to storage state-action pairs, as well as high time cost for computation. However, DQN will makes Q-learning more data-efficient (Lilian Weng, 2018).

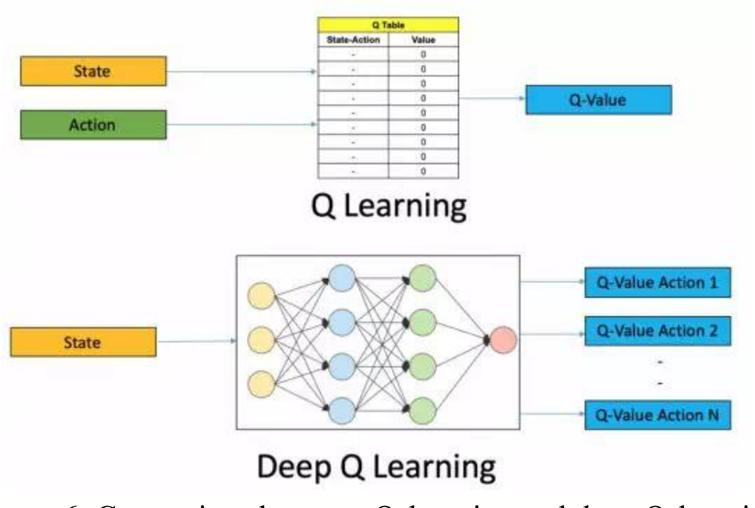


Figure 6. Comparison between Q-learning and deep Q-learning

The figure 6 illustrates the difference between Q-Learning and Deep Q-Learning. In naive Q-learning, we construct a Q table and take the pair of state and action of an agent as the inputs, then learning and updating the Q-value through the Bellman equation, Q(s,a)=r+γmax$_{a'}$Q(s′,a′), as the outputs. Nevertheless, a DQN is established to estimates the Q value in which the state (observation) of this game scenario as the inputs and the Q values for all available actions as the outputs.

In the DQN model, there are two significant parts containing experience replay and the model updates:

Experience replay: It stores the past experiences or episode steps in the form of [state, action, reward, next state] in the replay memory. Radom sample rather than the most recent transition be drawn from the replay memory to form a minibatch and one sample can be used multiple times. The minibatch be used during Q-learning updates, which avoiding the effect of data correlation and increased data efficiency (Jiqizhixin, 2019).

Network updates: We can use two networks to train our model, target network and policy or prediction network. Both they have same structure; the parameters of them need to be updated periodically as the figure 7 shows.

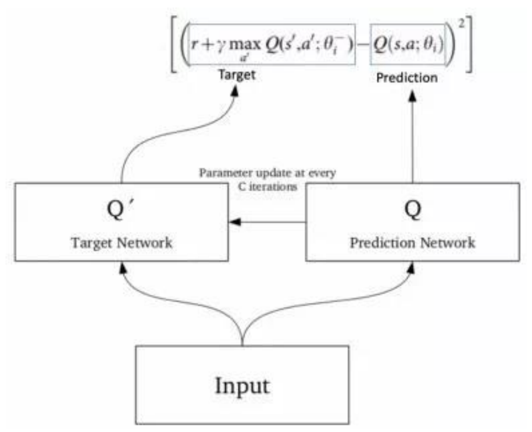
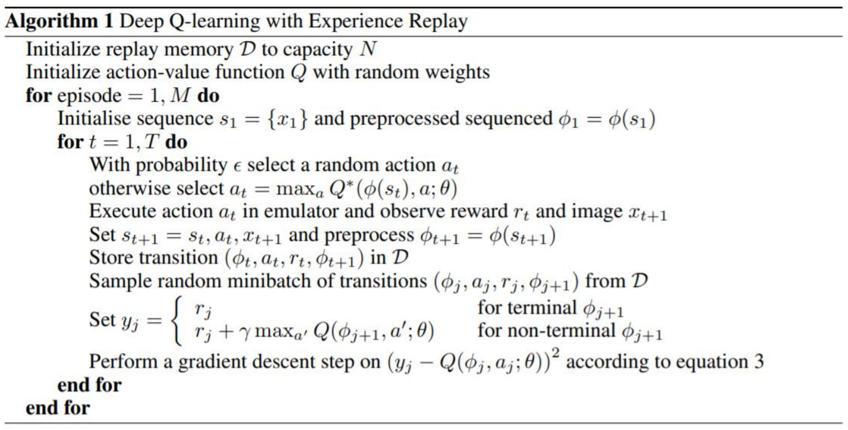

$\left[\left(\boxed{r+\gamma \max_{a'} Q(s',a';\theta_i^-)} - \boxed{Q(s,a;\theta_i)}\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

Figure 7. DQN with two networks (Jiqizhixin, 2019)

The figure 8 illustrates the pseudo code of Deep Q-learning with experience replay.



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
```

Figure 8. DQN with experience replay (Mnih et al., 2013)

Below is the process of how the DQN work in this project.

(1) Preprocessing the observation (or state) of the game and then feed it to the DQN, DQN will returns the Q values of all actions when the game at this state.

```
act_values = self.policy_net(state).detach().numpy()
```

(2) Selecting an action with epsilon-greedy policy, picking up an action with probability epsilon and choose the action that has max Q values with probability with 1-epsilon.

```
def act(self, state):
     if np.random.rand() <= self.eps_start:
         return random.randrange(self.action_size)
    act_values = self.policy_net(state).detach().numpy()
    return np.argmax(act_values[0])
```

(3) A reward can be obtained when taking the action at the states. Also, the game will go to next state which is the preprocessing image of the game screen. We store the transition to the memory with the form < state, action, reward, next_state, done>

```
def remember(self, state, action, reward, next_state, done):
        if reward == 0:
                self.memory_p.append((state, action, reward, next_state, done))
        else:
                self.memory_n.append((state, action, reward, next_state, done))
```

(4) Replay experience. Sampling randomly the mini batch from the memory we construct at previous step. Then feed it to the DQN model.

```
 if len(agent.memory_n) > batch_size / 2:
        minibatch_n = random.sample(self.memory_n, 5)
        minibatch_p = random.sample(self.memory_p, 59)
        minibatch = random.sample((minibatch_p+minibatch_n), batch_size)
 else:
        minibatch = random.sample(self.memory_p, batch_size)
```

(5) Computing the loss between the target and the prediction, optimizing the parameters of the DQN with gradient descent algorithm. It is noticed that using detach() in target_net, which is declared not to need gradients, because we just use the parameters of policy_net.

```
for state, action, reward, next_state, done in minibatch:
         next_state_values = self.target_net(next_state).detach().numpy()
         # Compute the expected Q values
         expected_state_action_values = (next_state_values * self.gamma) + reward
        #train the network
        expected_state_action_values = torch.from_numpy(expected_state_action_values).float()
        state_action_values = 0
         for t in range(1):
                state_action_values = policy_net(state)
                loss =  criterion(state_action_values, expected_state_action_values) #compute the loss
                optimizer.zero_grad()
                loss.backward()  #backward propagation
                optimizer.step()  # optimize the model
```

(6) Parameters updation.

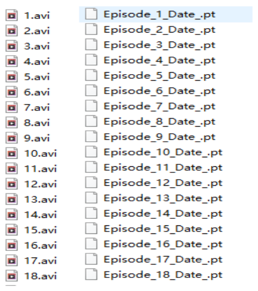
```
    target_net.load_state_dict(policy_net.state_dict())
    target_net.eval()
```

To sum up, the agent's actions are selected either randomly or from the output of policy_net according to epsilon-greedy policy, which lead to the next pace can be obtained from the gym. We save the results in a replay memory, picking a random minibatch from it to train the DQN model. Optimizer are used on each iteration to get the new policy; target_net is used to get the expected Q values and is updated occasionally in order to synchronization across two networks.

Step 5: Experimental results and discussion

We can record the process of playing the game with DQN model and the parameters of CNN in each episode as below:

```
for e in range(EPISODES):
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    vw = cv2.VideoWriter('gdrive/My Drive/Pacman/' + str(e) + '.avi', fourcc, 4, (160,210))
        for time in range(1000000000):
            if done:
                vw.release()
                agent.save('gdrive/My Drive/Pacman/' + "_Episode_" +str(e) + "_Date_" + '.pt')
                break
```



1.avi	Episode_1_Date_.pt
2.avi	Episode_2_Date_.pt
3.avi	Episode_3_Date_.pt
4.avi	Episode_4_Date_.pt
5.avi	Episode_5_Date_.pt
6.avi	Episode_6_Date_.pt
7.avi	Episode_7_Date_.pt
8.avi	Episode_8_Date_.pt
9.avi	Episode_9_Date_.pt
10.avi	Episode_10_Date_.pt
11.avi	Episode_11_Date_.pt
12.avi	Episode_12_Date_.pt
13.avi	Episode_13_Date_.pt
14.avi	Episode_14_Date_.pt
15.avi	Episode_15_Date_.pt
16.avi	Episode_16_Date_.pt
17.avi	Episode_17_Date_.pt
18.avi	Episode_18_Date_.pt

Figure 9. DQN with two networks

The figure 9 shows the videos and corresponding network models. We trained the DQN model and update continuously the parameters of CNN, saving the neuron network at each episode and reload the network by calling agent.save() and agent.load().

The video shows that when the game is initialized and playing, the agent will stay at the original for a while, then it performs an action proposed by the network including left and left, right, up and down moving, keeping static, eating the beans and so on. At meanwhile, the agent needs to avoid the swallowing from the ghosts or swallowing inversely the ghosts to get a big reward. And then, the game goes to next frame and return rewards associated with that change, the accumulated rewards are displayed at the bottom at the game screen.

The agent attempts to maximize the cumulative rewards through adopting the optimal policy given by the combination of reinforcement learning and deep learning. In general, as the increase on the number of training times, the parameters will be forward to less error and the agent will performs better. However, as the limitation on the performing hardware and time, we cannot to train for a long term, which result in lack of a clear trend on the rewards during the episodes as table 2 shows.

Episodes	1	2	3	4	5	6	7	8	9
Rewards	260	370	230	390	310	1010	180	170	190
Episodes	10	11	12	13	14	15	16	17	18
Rewards	380	160	250	610	260	120	220	290	370

Table 2. Rewards of each episode

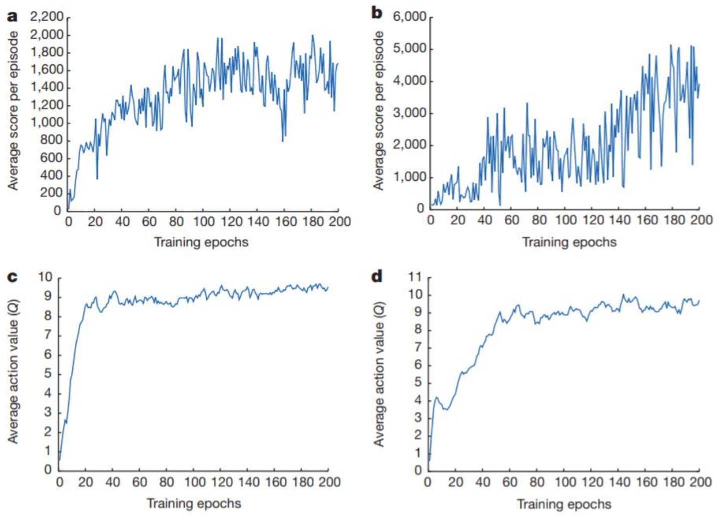


Figure 10. Training curves of DQN (Mnih et al., 2015)

As we have no the experiment data on the large number of episodes, we can refer to the study of Mnih et al (2015) as the figure 10 shows. The graphs give the result of two learnings consist of the agent's average score in each episode and the average predicted Q-values over the set of states. Graph a and c for the Atari game Space Invaders, b and d for Seaquest.

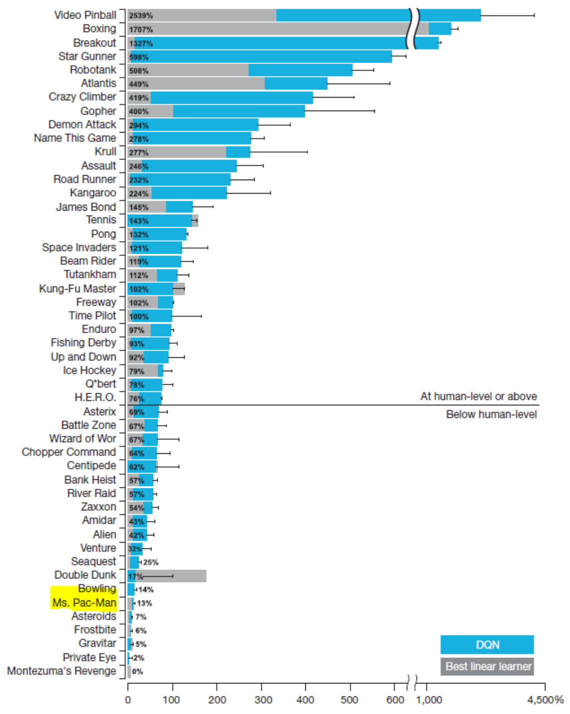


Figure 11. Comparison of DQN and best linear learner (Mnih et al., 2015)

As seen from figure 11, Mnih et al (2015) found that when playing 43 of Atari 2600 games, the DQN behaved at a level that was comparable or superior to the performance of a professional human players, attaining more than 75% of the scores of humans on the majority of the games though its performance lower than that of human on the game Pac Man in our project.

References:

Developers (2020) *Schooling Flappy Bird: A Reinforcement Learning Tutorial* Available at: https://www.toptal.com/deep-learning/pytorch-reinforcement-learning-tutorial (Accessed: 13 April 2020)

Grigsby Jake (2018) *Advanced DQNs: Playing Pac-man with Deep Reinforcement Learning* Available at: https://towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814 (Accessed: 10 April 2020)

Jiqizhixin (2019) *Introduction to implementation of Deep Q-Learning on OpenAI Gym* https://www.jiqizhixin.com/articles/2019-05-20-4 (Accessed: 15 April 2020)

Lilian Weng (2018) *Implementing Deep Reinforcement Learning Models with*

Tensorflow + OpenAI Gym Available at: https://lilianweng.github.io/lil-log/2018/05/05/implementing-deep-reinforcement-learning-models.html (Accessed: 7 April 2020)

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. *Nature*, *518*(7540), pp.529-533.

OpenAI Gym (2020) *Getting Started with Gym* Available at: https://gym.openai.com/docs/ (Accessed: 10 April 2020)