Deep Reinforcement Learning to Play Atari Games

Kartik Ahuja

March 22, 2019

Abstract

In this project, we implement deep reinforcement learning algorithms to play Atari Games. We start with simple architectures to play Box2D game, which take a small dimensional state vector (8) as input. Next, we move to RAM version of Atari Games. The RAM version has a larger state vector (128). At the end, we implement the screen version, which takes the image as input, which is a higher dimensional tensor $(210, 160, 3)$. All our implementations are in Google Colab and within a few hours we can train agents that perform much better than randomly acting agents.

1 Introduction

In this project, we will implement deep Q learning algorithms. Before we describe the algorithm, we first provide a brief summary of Q learning. Time is divided into discrete time steps. The agent interacts with the environment. At each time t, the agent makes an observation o_t. We assume that the set of actions are discrete. For instance, move up or down in the game of Pong. The environment at time t is in state s_t. In this section, we assume that the state s_t is completely observable, thus $o_t = s_t$.

If the agent takes the action a_t, then the environment transits into state s_{t+1} and the agent receives reward r_t. The state transition depends on transition kernel $T(s_{t+1}|s_t, a_t)$. We assume that this transition kernel is not known but it is deterministic (for instance, in the games we implement we will have a deterministic environment). We define the total reward from t $R_t = \sum_{k=0}^{\infty} \delta^k r_{t+k}$ with a discount factor $\delta \in (0, 1)$.

The action value $Q^\pi(s, a) = E[R_t|s_t = s, a]$ is the expected return for selecting action a in state s and following policy π. The optimal action value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$. Similarly, the value of state s under policy π is defined as $V^\pi(s) = E[R_t|s_t = s]$ and is simply the expected return for following policy π from state s. The optimal value function is defined as $V^*(s) = \max_\pi V^\pi(s)$. Q learning is a model-free approach to reinforcement learning, i.e., there are no assumptions on the dynamics of the transition. In Q learning, the goal of the agent is to estimate the Q function. There can be many ways to estimate the

Q function. The traditional approaches are non-parametric and update the Q function for each state, action pair separately at each time step. In parametric approaches, the parameters of the model are updated based on the reward received (we explain this step in detail next). In this work, we use a parametric model (for instance, a neural network) to estimate the Q function. We write the parametric model as $Q(s, a; \theta)$. Suppose the system is in state s and the agent follows action a and the agent receives reward r. Suppose θ_k are the parameter values during iteration k.

$$Y = r + \gamma \max_a Q(s', a'; \theta_k) \tag{1}$$

The parameters θ are updated to minimize the squared error between $\frac{1}{2}(Y - Q(s, a; \theta))^2$. The parameters are updated as $\theta_{k+1} = \theta_k + \alpha \nabla_\theta Q(s, a; \theta_k)(Y - Q(s, a; \theta))$. In practice, there are two modifications we need to take care.

- In practice, we replace the Q network in Y in (1) with $Q(s', a'; \theta_k^-)$. The parameters θ_k^- are updated every C iterations as follows $\theta_k^- = \theta_k$
- In online setting, we need to keep a replay memory. The replay memory stores the rewards, states, and observations for the last N_{replay} steps. At each step the update to the parameters θ are based on a randomly sampled batch from the replay memory.

The agent in each step follows an ϵ greedy policy, which works as follows. With ϵ probability the agent follows a random action and with probability $(1 - \epsilon)$ it follows the action prescribed by the current estimate of the Q function.

1.1 Implementation

We provide the code at the end of this draft. We give a detailed description of the structure of the code below.

1.1.1 QNetwork Class

In the QNetwork class we define the architecture we want to use for estimation of the Q function. For instance, we will use a MLP for the non-screen version and convolutional neural network for the screen version. In the QNetwork class, we define the forward pass for the network. Note that we only need to define the forward pass. Backward pass will be easy to do as we will see later. We will instantiate objects from the QNetwork class later.

1.1.2 ReplayBuffer Class

We initialize the replay buffer with the size of the memory, the action size, state size, buffer size, an empty list to store the interactions. We also define a named tuple. Different elements of the named tuple are : "state", "next state", "reward", "action", "done". We define a push function, which is used to push the experiences of the agent into the memory list. We define a sampling function, which generates random samples from the memory.

1.1.3 Agent Class

Initialize: In this class, we will define the Agent. The Agent has several components. We initialize the agent with quite a few parameters namely: size of the state (for instance, 8 for Box2d game), action size (for instance, 4 for Lunar Lander), buffer size to store the replay memory, batch size (how many samples to use in learning), learning rate (step for the gradient), update every (how often to update the first network), gamma (discount factor) and tau (for soft update of the target network). For the agent we also initialize two networks from the QNetwork class: a) Q_network and b) Q_network_val. Q_network is updated after update every in time steps. Q_network_val is the target network, which is updated using soft updates. For the agent we also initialize an optimizer. The optimizer is called later in the step function to compute the gradients of the loss function. We also initialize an object of the ReplayBuffer to store the replay memory.

We define some functions for the Agent class as follows.

Act: In this function, the agent acts based on the epsilon greedy policy. There are two parts to the policy. With probability epsilon a random action is taken. With the remaining probability the agent follows the outcome dictated by the deep Q network.

Step: In step function, we push the current samples (sampled from the environment) into the replay memory (we use the ReplayBuffer object we initialized). We push the reward, action, observation, state and next state and indicator if the terminal state has been reached or not into the memory. As soon as the length of the memory becomes larger than the batch size, the agent starts to learn. In the learning part, our goal is to update the Q_network and Q_network_val. We first define the loss function. We use a mean squared error loss function. We define the target value as in (1), which uses the target network Q_network_val as in the RHS of (1). The loss function measures the distance between the target value and the estimate computed based on the Q_network. We use the gradient of the loss function (computed using backward pass) to update the Q_network (the optimizer decides how to update the network based on the gradient value). We use soft updates to update the target network Q_network_val.

2 Architectures and Results

2.1 Architecture for Box2D: LunarLander-v2

In LunarLander-v2, the task for the agent is to perform a perfect landing (See further details for Lunar Lander-v2 on (`https://gym.openai.com/envs/LunarLander-v2/`). LunarLander-v2 is considered solved when the score crosses 200. The state for LunarLander-v2 is a eight dimensional vector. We use the following architecture for LunarLander-v2. We use the following notations: FC-Net: Fully Connected Network

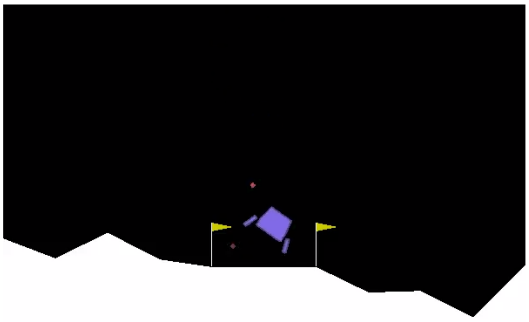

Figure 1: Random agent performing poorly initially in lunar landing task

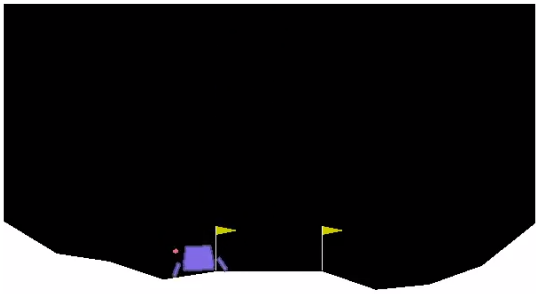

Figure 2: Partially trained agent performing moderately better in lunar landing task

- **Layer 1:** FCNet with 256 outputs → RELU
- **Layer 2:** FCNet with 256 outputs → RELU
- **Layer 3:** FCNet with 4 outputs

In Figure 1 below, we show how a random acting agent performs on LunarLander-v2. In Figure 2, we show how a partially trained agent is able to land but it lands outside the flag. In Figure 3, we show the completely trained agent doing well in landing task. In Figure 4, we show the performance of the method vs number of episodes. The average score for the last 100 epochs is 202.

2.2 Architecture for Atari-Ram: Pong-ram-v0

Our goal is to make an agent learn to play Pong. The information that is available to use is a 128 dimensional vector available from the RAM. We use the following architecture for Pong-ram-v0. In Pong there are 6 actions but we only used 4 actions as training with all actions was taking longer. We use the following notations: FCNet: Fully Connected Network, RELU: Rectified Linear Unit

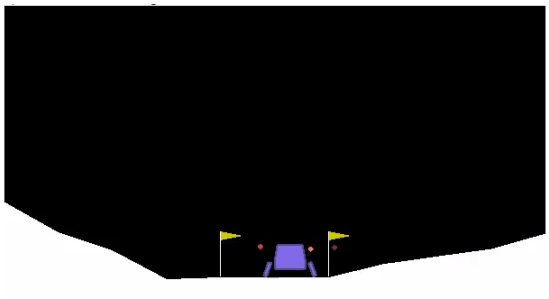

Figure 3: Completely trained agent performing well in lunar landing task

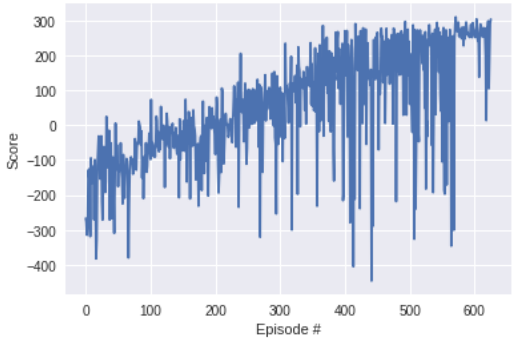

Figure 4: Score of the agent versus episode

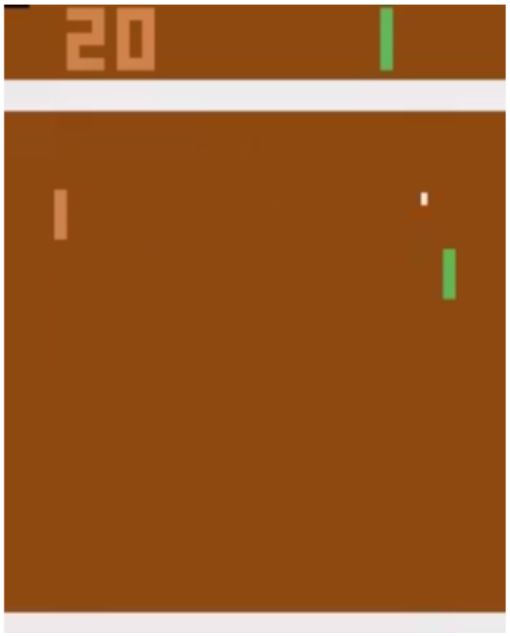

Figure 5: Random agent performing poorly initially in Pong. Score: -19. Green is our agent.

- **Layer 1:** FCNet with 256 outputs → RELU
- **Layer 2:** FCNet with 256 outputs → RELU
- **Layer 3:** FCNet with 256 outputs → RELU
- **Layer 4:** FCNet with 256 outputs → RELU
- **Layer 5:** FCNet with 4 outputs

In Figure 5 (agent in green is our agent), we show how a random acting agent performs poorly in Pong and has a score of -19. In Figure 6, we show how a well trained agent is able to outperform the other agent and has a score of 2. In Figure 7, we show the performance of the method improves vs number of episodes. The average score for the last 100 epochs is -2.5.

2.3 Architecture for Atari: Pong-v0

In this section, we now describe a harder task. Our goal is to make an agent learn to play Pong. The information that is available to use is the screen image, which is a (210, 160, 3) tensor. We use the following architecture for Pong-v0. We use the following notations: Conv2D: a two dimensional convolutional filter.

- **Layer 1:** 32 ConvNet filters, filter size = 3, stride =2, pad=1 → ELU

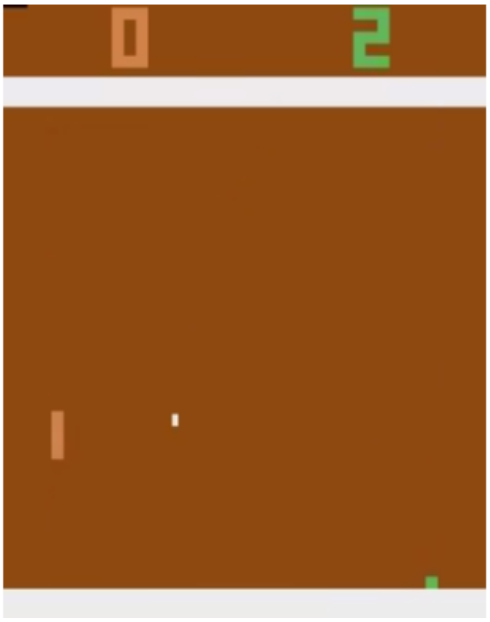

Figure 6: Well trained agent performing much better in Pong. Score: 2

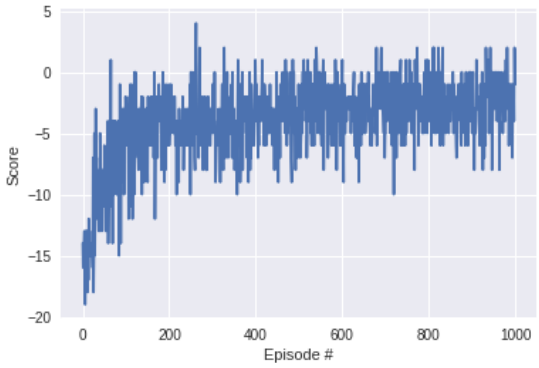

Figure 7: Score of the agent versus episode

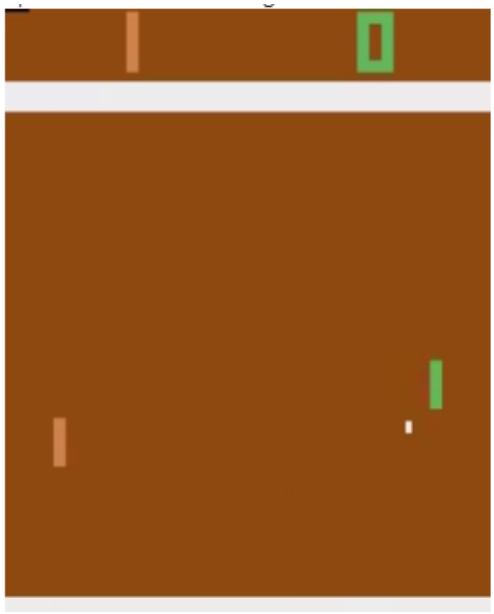

Figure 8: Partially trained agent giving competition to the opponent. Score: -1

- **Layer 2:** 32 ConvNet filters, filter size = 3, stride =2, pad=1 → ELU
- **Layer 3:** 32 ConvNet filters, filter size = 3, stride =2, pad=1 → ELU
- **Layer 4:** 32 ConvNet filters, filter size = 3, stride =2, pad=1 → ELU
- **Layer 5:** GRUcell with output shape of 100
- **Layer 6:** FCNet with output shape of 4

In Figure 5 (agent in green is our agent), we show how a random acting agent performs poorly in Pong and has a score of -19. In Figure 8, we show how a well trained agent is able to reasonably compete with other agent. In Figure 9, we show the performance of the method improves vs number of episodes. The score has improved from -20 to around -4.5. Due to RAM limitations on Google Colab it was not possible to run more episodes, which should have further improved the agent. The average score for the last 100 epochs is -4.5.

3 Conclusion

We provided a brief introduction to deep Q learning. We discussed three different architectures in the increasing order of complexity. These architectures were trained and shown to perform well in their respective tasks. In the next section, we provide the code.

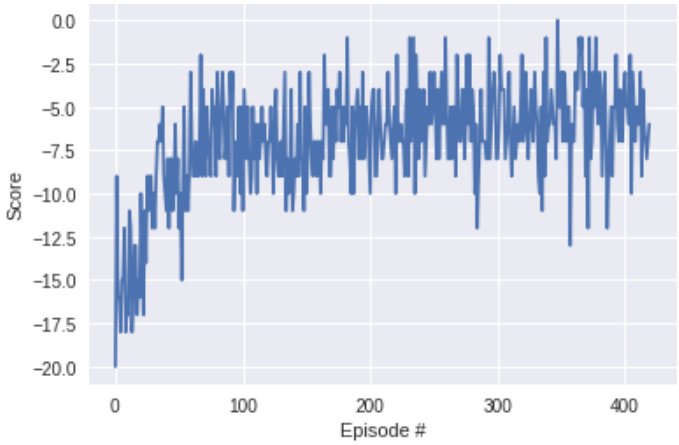

Figure 9: Score of the agent versus episode

In [ ]:
```
import gym
from gym import wrappers
import random
import torch
import numpy as np
from collections import deque, namedtuple
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch.nn as nn
import torch.optim as optim
import glob
import io
import base64
from IPython.display import HTML
from IPython import display as ipythondisplay
from pyvirtualdisplay import Display
# %matplotlib inline

def show_video(folder):
    mp4list = glob.glob('%s/*.mp4' % folder)
    if len(mp4list) > 0:
        encoded = base64.b64encode(io.open(mp4list[0], 'r+b').read())
        ipythondisplay.display(HTML(data='''<video alt="test" autoplay loop controls style="height: 400px;">
        <source src="data:video/mp4;base64,{0}" type="video/mp4" /> </video>'''.
                                    format(encoded.decode('ascii'))))

display = Display(visible=0, size=(400, 300))
display.start()

"""### 2. Try it

The following code will output a sample video whose action is random sampled.
"""

atari_game = "LunarLander-v2"
env = gym.wrappers.Monitor(gym.make(atari_game), 'sample', force=True)
env.seed(0)
print('State shape: ', env.observation_space.shape)
print('Number of actions: ', env.action_space.n)

state = env.reset()
```

```
cr = 0
for j in range(2000):
    action = env.action_space.sample()
    env.render()
    state, reward, done, _ = env.step(action)

    cr += reward
    print('\r %.5f' % cr, end="")
    if done:
        break
env.close()
show_video('sample')

print (state.shape)

"""### 3. Define QNetwork, agent and replay buffer"""

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 1e-3               # learning rate
UPDATE_EVERY = 5        # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print (device)

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=256, fc2_units=256):
        super(QNetwork, self).__init__()
        self.seed   = torch.manual_seed(seed)
        self.state_size  = state_size
        self.action_size = action_size
        self.fc1_units   = fc1_units
        self.fc2_units   = fc2_units
        self.layer1 = nn.Linear(self.state_size, self.fc1_units, bias=True)
        self.layer2 = nn.Linear(self.fc1_units,  self.fc2_units, bias=True)
        self.layer3 = nn.Linear(self.fc2_units,  self.action_size, bias=True)

    def forward(self, state):
        """Build a network that maps state -> action values."""
```

```
        layer1 = F.relu(self.layer1(state))
        layer2 = F.relu(self.layer2(layer1))
        layer3 = self.layer3(layer2)
        return layer3

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed, buffer_size, batch_size,
                 learning_rate, update_every,gamma, tau):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.buffer_size = buffer_size
        self.batch_size  = batch_size
        self.learning_rate = learning_rate
        self.update_every = update_every
        self.gamma        = gamma
        self.tau          = tau
        self.Q_network     = QNetwork(self.state_size, self.action_size, seed )
        self.Q_network_val = QNetwork(self.state_size, self.action_size, seed )

        # Replay memory
        self.memory        = ReplayBuffer(self.action_size, self.buffer_size, self.batch_size, self.seed)
        self.optimizer     = optim.Adam(self.Q_network.parameters(), lr=self.learning_rate)

        self.steps_until_update = 0

    def step(self, state, action, reward, next_state, done):

        # Save experience in replay memory
        self.memory.push(state, action, reward, next_state, done)
        self.steps_until_update = (self.steps_until_update + 1)%self.update_every

        if(self.steps_until_update==0):
```

```
        if(self.memory.__len__()>self.batch_size):
          sample = self.memory.sample()
          states, actions, rewards, next_states, dones = sample
          self.Q_network_val.eval()
          with torch.no_grad():
            target_rewards = rewards + self.gamma*(torch.max(self.Q_network_val.forward(next_states),
                                                             dim=1, keepdim=True)[0])*(1-dones)

          self.Q_network.train()
          expected_rewards = self.Q_network.forward(states).gather(1, actions)
          loss             = F.mse_loss(expected_rewards, target_rewards)
          self.optimizer.zero_grad()
          loss.backward()
          self.optimizer.step()

          for Q_network_val_parameters, Q_network_parameters in zip(self.Q_network_val.parameters(),
                                                                    self.Q_network.parameters()):
              Q_network_val_parameters.data.copy_(self.tau * Q_network_parameters.data +
                                                  (1.0 - self.tau) * Q_network_val_parameters.data)

    def act(self, state, eps=0.1):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """

        state = torch.from_numpy(state).float().unsqueeze(0)
        with torch.no_grad():
            action_values = self.Q_network(state)
        uniform_random = random.random()

        if(uniform_random >eps):
          action = np.argmax(action_values.cpu().data.numpy())
        else:
          action = np.random.randint(self.action_size)
        return action
```

```
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.
        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.buffer_size = buffer_size
        self.batch_size  = batch_size
        self.seed        = random.seed(seed)
        self.position    = 0
        self.memory      = []
        self.transition = namedtuple("Transition",
                                     field_names=["state", "action", "reward",
                                                  "next_state", "done"])

    def push(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        self.current_transition = self.transition(state, action, reward, next_state, done)
        if(len(self.memory)< self.buffer_size):
          self.memory.append(None)
        self.memory[self.position] = self.current_transition
        self.position = (self.position + 1) % self.buffer_size


    def sample(self):
        """Randomly sample a batch of experiences from memory."""

        sample = random.sample(self.memory, self.batch_size)
        sample_list = self.transition(*zip(*sample))
        states = torch.from_numpy(np.vstack(sample_list.state)).float()
        actions = torch.from_numpy(np.vstack(sample_list.action)).long()
        rewards = torch.from_numpy(np.vstack(sample_list.reward)).float()
        next_states = torch.from_numpy(np.vstack(sample_list.next_state)).float()
        dones = torch.from_numpy(np.vstack(sample_list.done).astype(np.uint8)).float()

        return (states, actions, rewards, next_states, dones)
```

```
    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)

"""### 3. Train the Agent with DQN"""

def train_dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon

    env = gym.wrappers.Monitor(gym.make(atari_game), 'output', force=True)

    render = True
    for i_episode in range(0, n_episodes):
        if render and i_episode % 100 == 0:
            env = gym.wrappers.Monitor(gym.make(atari_game), 'output_%d' % i_episode, force=True)
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            if render and i_episode % 100 == 0:
                env.render()
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
```

```
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="")
        if i_episode % 100 == 0:
            if render:
                env.close()
                show_video('output_%d' % i_episode)
                env = gym.make(atari_game)
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
        if np.mean(scores_window)>=200.0: # You can change for different game
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.
                  format(i_episode-100, np.mean(scores_window)))
            torch.save(agent.Q_network.state_dict(), 'checkpoint.pth')
            break
    return scores
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 1        # how often to update the network
agent = Agent(state_size=8, action_size=4, seed=0, buffer_size= BUFFER_SIZE, batch_size= BATCH_SIZE,
              learning_rate= LR, update_every = UPDATE_EVERY, gamma= GAMMA, tau=TAU)

scores = train_dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```