STATS 231A: Final Project

Peng-Yu Chen
Graduate Student
Department of Civil Engineering, UCLA
sam75782008@g.ucla.edu

Abstract

The final project of STATS 231A is aiming to implement the Reinforcement Learning (RL) technique for self-playing game. In this report, the **Breakout-Ram-V0** is selected as the demonstration of RL with implementation of the Deep Q-Learning and the Policy Gradient.

1 Introduction of RL

In the world of artificial intelligence, the ultimate goal is to develop agents that can act like human beings. Given from the learning process we have been through, the goal is always to get an optimal reward. Depends on the state, different action can be made. The goal is hence find out the action given the state to maximize the long-term reward. In this project, the video game **Atari Breakout** is selected for implementing the RL. In this game, the action could be either moving left or moving right, and the reward is the score the agent receive when catching the ball. Otherwise, it loose one life, and it only has five life.

1.1 Deep Q-Learning

Let's start from the Q Learning then moving to the deep Q-Learning. In the Q Learning, a Q function is introduced to record the reward given the state and the action, where the Q is shown in e.q.1. α is the learning rate, R is the immediate reward, and γ is the discount factor that adjust the contribution of the future reward.

$$Q(s,a) = Q(s,a) + \alpha \times [R(s,a) + \gamma \times maxQ(s',a) - Q(s,a)] \tag{1}$$

It is worth noted that, if the agent choose action a_1 and get positive reward, it will never choose action a_2. To improve this issue, the $\epsilon - greedy$ approach is introduced so that the agent should explore at probability of ϵ or exploit at a probability of $1 - \epsilon$. For the case that the size of state and action are extreme large, it is difficult to record all the actions and the corresponding rewards. This is where the Deep Q-Learning take into control. Instead of storing all Q values, the neural network structure can approximate the value through million of hidden variables. The neural network use the input frame (i.e., image) that describe the state to compute the possible Q values. Users can easily define the number of fully-connected layers (i.e., 4 in this project). The loss function is shown in e.q.2. The idea is to minimize the difference between the predicted Q values and the initialized Q values with randomness, which can be achieved by the stochastic gradient descent. The gradient is shown in e.q.3, where w is the weights in the neural networks.

$$L = E[(\gamma + \gamma \times maxQ(s',a') - Q(s,a))^2] \tag{2}$$

$$\frac{\partial L(w)}{\partial w} = E[(\gamma + \gamma \times maxQ(s',a') - Q(s,a))\frac{\partial Q(s,a,w)}{\partial w}] \tag{3}$$

When training the deep Q network, it is not feasible to keep adding new man-made game into the network. Instead, experience replay technique is used. To be more specific, a replay buffer is generated to store the past experiences (i.e., current state, actions, reward, next states). To update the weight of the neural networks, the agent will randomly select a portion data of the replay buffer. Another important technique in the deep Q network is to create two separate neural networks. The reason is that when the predicted Q values are closed to the experience, the gradient may not be found, which could causes divergence. Creating two separate networks (one updates the parameters slower than the other) is hence introduced in the algorithm.

1.2 Policy Gradient

The goal of reinforcement learning is to find an optimal behavior strategy for the agent to obtain optimal rewards. The **policy gradient** method target at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function respect to θ, $\pi_\theta(a|s)$. The value of the reward (objective) function depends on this policy and then the goal is to optimize θ for the best reward. The reward function is defined in e.q.4.

$$J(\theta) = \sum_{s \in S} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \pi_\theta(a|s) Q^{\pi}(s,a) \tag{4}$$

It is natural to expect policy-based methods are more useful in the continuous space because there is an infinite number of actions and states to estimate the values for and hence value-based approaches are too computational-expensive. However, using the gradient ascent, we can move θ toward the direction suggested by the gradient $\nabla_\theta J(\theta)$ to find the best θ for π_θ that produces the highest return. The gradient of e.g.4 is in e.q.5.

$$\nabla J(\theta) = E_{\pi_\theta}[\nabla ln \pi(a|s,\theta) Q_{\pi}(s,a)] \tag{5}$$

2 Implementing Detail

2.1 Deep Q-Learning

Fig.1 and 2 show the class of QNetwork where four fully-connected network are implemented in this project with [1024, 1024, 512, 256] dimensions. Each fully-connected layer includes linear combination, batch normaliztion and dropout which used to avoid overfitting. The forward function is to put the state into the network and compute the action.

Fig.3, 4, and 5 show the agent class and the functions in it. With the initialization of state size and action size, two identical networks are developed. One is for the local Q value, the other is for the target Q value. Adam optimizer is used to minimize the loss. State, action, reward and next state are memorized as a truple in the ReplayBuffer and are used in every learning step. In the action function, the action is decided from the forward calculation of the local network, whether to do it is depended on the ϵ probability.

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

    def __init__(self, state_size, action_size, seed, fc1_units=1024, fc2_units=1024, fc3_units=512, fc4_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
            fc3_units (int): Number of nodes in third hidden layer
            fc4_units (int): Number of nodes in fourth hidden layer
        """
        super(QNetwork, self).__init__()
        # fixed random
        self.seed = torch.manual_seed(seed)
        # state and action size
        self.state_size = state_size #128
        self.action_size = action_size #4

        # 4-fully connected network
        self.fc1_units   = fc1_units #1024
        self.fc2_units   = fc2_units #1024
        self.fc3_units   = fc3_units #512
        self.fc4_units   = fc4_units #256

        #FC1: linear comb --> batch normalization --> dropout (avoid overfitting)
        self.layer1 = nn.Linear(self.state_size, self.fc1_units, bias=True)
        self.bn1 = nn.BatchNorm1d(self.fc1_units)
        self.dp1 = nn.Dropout(p=0.5)

        #FC2
        self.layer2 = nn.Linear(self.fc1_units,  self.fc2_units, bias=True)
        self.bn2 = nn.BatchNorm1d(self.fc2_units)
        self.dp2 = nn.Dropout(p=0.5)

        #FC3
        self.layer3 = nn.Linear(self.fc2_units,  self.fc3_units, bias=True)
        self.bn3 = nn.BatchNorm1d(self.fc3_units)
        self.dp3 = nn.Dropout(p=0.5)

        #FC4
        self.layer4 = nn.Linear(self.fc3_units,  self.fc4_units, bias=True)
        self.layer5 = nn.Linear(self.fc4_units,  self.action_size, bias=True)
```

Figure 1: QNetwork: Fully-connected Network

```python
#Given State-->action (Q values)
def forward(self, state):
    """Build a network that maps state -> action values."""
    #return state
    layer1 = F.relu(self.layer1(state))
    layer2 = F.relu(self.layer2(layer1))
    layer3 = F.relu(self.layer3(layer2))
    layer4 = F.relu(self.layer4(layer3))
    action_values = self.layer5(layer4)
    return action_values
```

Figure 2: QNetwork: Forward calculation

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

    def __init__(self, state_size, action_size, seed):
        """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)


        # Q-Network
        # Two Networks are needed because Q learning needs to update network itself
        # The first network updates faster than the second network
        # e.g. first netwrok updates every epoch whereas the second network only updates the parameters
        # every 10 epoches.
        # .to(device) is a command for GPU
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)

        #optimization: Adam optimizer
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory: store the experience(action, state)
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0
```

Figure 3: Agent Class

```python
#update and store experience
def step(self, state, action, reward, next_state, done):
    # Save experience in replay memory
    self.memory.push(state, action, reward, next_state, done)#push into truple

    # Learn every UPDATE_EVERY time steps.
    self.t_step = (self.t_step + 1) % UPDATE_EVERY
    if self.t_step == 0:
        # If enough samples are available in memory, get random subset and learn
        if len(self.memory) > BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

#action
def act(self, state, eps=0.):
    """Returns actions for given state as per current policy.

    Params
    ======
        state (array_like): current state
        eps (float): epsilon, for epsilon-greedy action selection
    """
    #comvert the state from numpy data type to pytorch tensor
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    #call Q-Network
    self.qnetwork_local.eval()
    with torch.no_grad():
        action_values = self.qnetwork_local(state)
    self.qnetwork_local.train()

    # Epsilon-greedy action selection
    if random.random() > eps:
        return np.argmax(action_values.cpu().data.numpy())
    else:
        return random.choice(np.arange(self.action_size))
```

Figure 4: Agent: step and action

During the learning, the predicted Q values and the expected Q values are calculated and used to minimize the loss function through the optimizer. The local network is first update during the backward propagation then the target network is updated. That is, the target network is updated later than the local network.

```python
def learn(self, experiences, gamma):
    """Update value parameters using given batch of experience tuples.
    Params
    ======
        experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    #pull out from truple
    states, actions, rewards, next_states, dones = experiences

    # Get max predicted Q values (for next states) from target model
    Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
    # Compute Q targets for current states
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Get expected Q values from local model
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    # Compute loss (mean square error)
    loss = F.mse_loss(Q_expected, Q_targets)
    # Minimize the loss: gradient descent
    self.optimizer.zero_grad()
    loss.backward()#-->update the local network
    self.optimizer.step()

    # ------------------- update target network ------------------- #
    self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

def soft_update(self, local_model, target_model, tau):
    """Soft update model parameters.
    θ_target = τ*θ_local + (1 - τ)*θ_target
    Params
    ======
        local_model (PyTorch model): weights will be copied from
        target_model (PyTorch model): weights will be copied to
        tau (float): interpolation parameter
    """
    for target_param, local_param in zip(target_model.parameters(), local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

Figure 5: Agent: learning and update

Fig.6 shows the ReplayBuffer class which store the state, reward, actions, and the next state. During the learning process, a batch size of experience is selected randomly to update the weights of QNetwork.

```python
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.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def push(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        #randomly select batch_size from experience
        experiences = random.sample(self.memory, k=self.batch_size)
        #corresponding states, actions, rewards, and next states.
        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(d
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).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)
```

Figure 6: Memory Buffer

Fig.7 shows the Deep Q-Learning class where 10,000 episodes are defined for this project. Within each episode, 1000 time step are used to traine the agent with the initial $\epsilon = 1.0$, the end $\epsilon = 0.01$, and the decay $\epsilon = 0.995$. The video is outputed every 1,000 episodes with printing the average score of last 100 time steps. The model will be stored if the score is larger than 10.

```python
#Deep Q-Learning
def dqn(n_episodes=10000, 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 % 1000 == 0:
            env = gym.wrappers.Monitor(gym.make(atari_game), 'output_%d' % i_episode, force=True)
            state = env.reset()
        else:
            state = env.reset()
        score = 0
        for t in range(max_t):
          #based on epsioln to select action
            action = agent.act(state, eps)
            if t%100==0:
                action = 1
            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 % 1000 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
            if render:
                env.close()
                show_video('output_%d' % i_episode)
                env = gym.make(atari_game)
        if np.mean(scores_window)>=10.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode-100, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores

#main train
agent = Agent(state_size=env.observation_space.shape[0], action_size=env.action_space.n, seed=0)
scores = 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()
```

Figure 7: Deep Q-Learning

2.2 Policy Gradient

Policy gradient methods maximize the expected total reward by repeatedly estimate the gradient $g : \nabla_\theta E[\sum_{t=0}^{\infty} \gamma_t]$. There are several different related expression for the policy gradient, which have the form in e.q.6, where the Ψ_t is the total reward of the trajectory.

$$g = E[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta log\pi_\theta(a_t|s_t)] \tag{6}$$

Fig.8 and 9 show the class of policy gradient and the training process. It starts from the initialization of policy parameters, rewards, experience (observation), actions and its probability. Within the training of each episode, the logit probability of action will be calculate as well as its gradient to compute the expectation of the reward. The gradient ascent is implemented to maximize the reward with the direction of the optimal policy.

```python
class LogisticPolicy:

    def __init__(self, θ, α, γ):
        # Initialize paramters θ, learning rate α and discount factor γ
        # Initialization of policy parameters; learning rate and the deduction factor of future uncertainty
        self.θ = θ
        self.α = α
        self.γ = γ

    def logistic(self, y):
        # definition of logistic function
        # logit function for the probability of actions
        return 1/(1 + np.exp(-y))

    def probs(self, x):
        # returns probabilities of two actions

        y = x @ self.θ
        prob0 = self.logistic(y)

        return np.array([prob0, 1-prob0])

    def act(self, x):
        # sample an action in proportion to probabilities

        probs = self.probs(x)
        action = np.random.choice([0, 1], p=probs)

        return action, probs[action]

    def grad_log_p(self, x):
        # calculate grad-log-probs
        y = x @ self.θ
        grad_log_p0 = x - x*self.logistic(y)
        grad_log_p1 = - x*self.logistic(y)

        return grad_log_p0, grad_log_p1

    def grad_log_p_dot_rewards(self, grad_log_p, actions, discounted_rewards):
        # dot grads with future rewards for each action in episode
        # discounted_rewards: predicted rewards with future uncertainty
        return grad_log_p.T @ discounted_rewards

    def discount_rewards(self, rewards):
        # calculate temporally adjusted, discounted rewards

        discounted_rewards = np.zeros(len(rewards))
        cumulative_rewards = 0
        for i in reversed(range(0, len(rewards))):
            cumulative_rewards = cumulative_rewards * self.γ + rewards[i]
            discounted_rewards[i] = cumulative_rewards

        return discounted_rewards

    def update(self, rewards, obs, actions):
        # calculate gradients for each action over all observations
        grad_log_p = np.array([self.grad_log_p(ob)[action] for ob,action in zip(obs,actions)])

        assert grad_log_p.shape == (len(obs), 4)

        # calculate temporaly adjusted, discounted rewards
        discounted_rewards = self.discount_rewards(rewards)

        # gradients times rewards
        dot = self.grad_log_p_dot_rewards(grad_log_p, actions, discounted_rewards)

        # gradient ascent on parameters
        self.θ += self.α*dot
```

Figure 8: Policy Gradient

```python
def run_episode(env, policy, render=False):

    observation = env.reset()
    totalreward = 0
    #nitialization of observation, actions, rewards, and probabilities
    observations = []
    actions = []
    rewards = []
    probs = []

    done = False

    while not done:
        if render:
            env.render()

        # add state
        observations.append(observation)
        # conduct action
        action, prob = policy.act(observation)
        observation, reward, done, info = env.step(action)
        #calculate rewards
        totalreward += reward
        rewards.append(reward)
        actions.append(action)
        probs.append(prob)

    return totalreward, np.array(rewards), np.array(observations), np.array(actions), np.array(probs)

def train(θ, α, γ, Policy, MAX_EPISODES=1000, seed=None, evaluate=False):

    # initialize environment and policy
    env = gym.make('CartPole-v0')
    #env = gym.make('Breakout-ram-v0')
    if seed is not None:
        env.seed(seed)
    episode_rewards = []
    policy = Policy(θ, α, γ)

    # train until MAX_EPISODES
    for i in range(MAX_EPISODES):

        # run a single episode
        total_reward, rewards, observations, actions, probs = run_episode(env, policy)

        # keep track of episode rewards
        episode_rewards.append(total_reward)

        # update policy
        policy.update(rewards, observations, actions)
        print("EP: " + str(i) + " Score: " + str(total_reward) + " ",end="\r", flush=False)

    # evaluation call after training is finished - evaluate last trained policy on 100 episodes
    if evaluate:
        env = Monitor(env, 'pg_cartpole/', video_callable=False, force=True)
        for _ in range(100):
            run_episode(env, policy, render=False)
        env.env.close()

    return episode_rewards, policy
```

Figure 9: Policy Gradient:train

3 Results

Fig.9 and 10 show the demonstration of the playing from the agent and the learning process, where the example showing the average score every 100 step. Although it is not the highest, one can see the highest in the learning curve is around 25. Fig.12 shows the result from the policy gradient, which only used 200 episodes to reach the convergence of the score. It is much efficient than the Deep Q-Learning.

Figure 10: Playing Demonstration

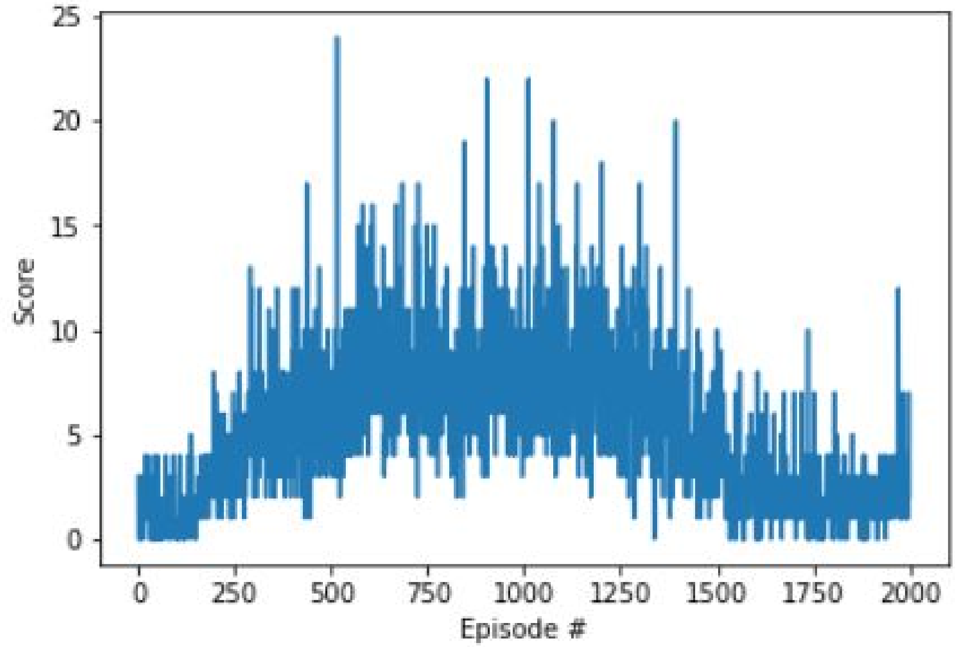


Figure 11: Training Processing

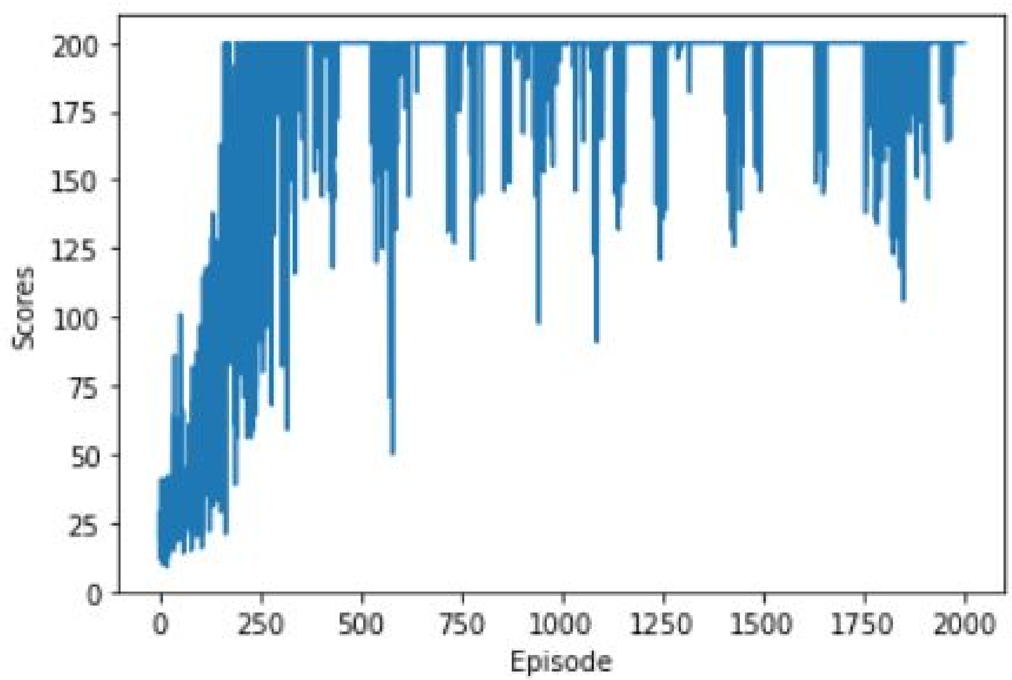


Figure 12: Policy Gradient

References

[1] REINFORCEMENT LEARNING (DQN) TUTORIAL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html [2] Gradient Policy Algorithm: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

2020 Fall STAT 231A --- Final Deep Q-Network (DQN)

Peng-Yu Chen

704732316

12.13.2020

In this notebook, you will try two reinforcement learning algorithm :

1. Deep Q-learning with replay buffer.
2. Policy gradient.

on OpenAI Gym's Atari/box2d game.

I provided all the code necessary. What you have to do is modify the corresponding network structure and hyperparameters. The current network structure are defined to run the game "CartPole-v0", which is the easiest game in GYM. A very good official pytorch tutorial is a good start. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. You are required to choose at least one of the following games. You can choose any atari / box2d game you like under this two webpage:

- [EASY] https://gym.openai.com/envs/#box2d The box2d game state is the smallest. e.g. LunarLander-V2, it has only 8 dims.
- https://gym.openai.com/envs/#atari Each atari game has two kind of input.
  - [MEDIUM] RAM version has a small state of only 128 dims. You can use fully connected layer to train.
  - [HELL] Screen version takes image as state which is around 200*200*3 dims. You need conv layer to train.

The implementation of [EASY] is required. If you make it all right, typical you will train a good agent [illegible] ELL] is optional with bouns. Challange your self on atari game. Screen version need CNN and typically need 10 hour to train.

You have to "solve" the problem to earn full credits. Definition of solved : See https://github.com/openai/gym/wiki/Leaderboard

There are no specific definition of solved for atari game.

Upload two files for coding part in Final.

- A pdf files : Your report. Please write down specific algorithm, implementing detail and result (Include sample game screenshot and reward-epoch plot) Also, attach all the code at the end of the pdf. For implementing detail, you can just comment on the code.
- This ipynb files.
- PS. If you think my implementation is bad, fell free to implement your own. You can use Tensorflow if you prefer to do so. However, please define the same class as this template. Include at least : agent class with act and learn; replay class with push and sample; q-function class with deep network structure; a train function.

1. Import the Necessary Packages

```
!pip install box2d-py
!apt-get install -y xvfb python-opengl > /dev/null 2>&1
!pip install gym pyvirtualdisplay > /dev/null 2>&1
```

```
Collecting box2d-py
  Downloading https://files.pythonhosted.org/packages/06/bd/6cdc3fd994b0649dcf5d9bad85b
     |████████████████████████████████| 450kB 12.3MB/s
Installing collected packages: box2d-py
Successfully installed box2d-py-2.3.8
```

```
#import package
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
```

Saved successfully!

```
hondisplay
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="h
        <source src="data:video/mp4;base64,{0}" type="video/mp4" /> </video>'''.format(encode

display = Display(visible=0, size=(400, 300))
```

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

```
<pyvirtualdisplay.display.Display at 0x7f39bba8e0b8>
```

2. Try it

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

Let's work on "Breakout-ram: Maximize the score."

```
atari_game = "Breakout-ram-v0"
# atari_game = "LunarLander-v2"
#atari_game = "CartPole-v0"

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')
```

Saved successfully!

```
State shape:  (128,)
Number of actions:  4
 2.00000
```

3. Define QNetwork, agent and replay buffer

```
#Define hyperparameters
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
GAMMA = 0.98            # discount factor
TAU = 0.9e-3              # for soft update of target parameters
LR = 5e-5               # learning rate
UPDATE_EVERY = 1        # how often to update the network

#running GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


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

    def __init__(self, state_size, action_size, seed, fc1_units=256, fc2_units=256, fc3_units
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
            fc3_units (int): Number of nodes in third hidden layer
            fc4_units (int): Number of nodes in fourth hidden layer
        """
                                              t__()

        self.seed = torch.manual_seed(seed)
        # state and action size
        self.state_size = state_size #128
        self.action_size = action_size #4

        # 4-fully connected network
        self.fc1_units   = fc1_units #1024
        self.fc2_units   = fc2_units #1024
        self.fc3_units   = fc3_units #512
        self.fc4_units   = fc4_units #256
```

Saved successfully!

```
        #FC1: linear comb --> batch normalization --> dropout (avoid overfitting)
        self.layer1 = nn.Linear(self.state_size, self.fc1_units, bias=True)
        #self.bn1 = nn.BatchNorm1d(self.fc1_units)
        #self.dp1 = nn.Dropout(p=0.5)

        #FC2
        self.layer2 = nn.Linear(self.fc1_units,  self.fc2_units, bias=True)
        #self.bn2 = nn.BatchNorm1d(self.fc2_units)
        #self.dp2 = nn.Dropout(p=0.5)

        #FC3
        self.layer3 = nn.Linear(self.fc2_units,  self.fc3_units, bias=True)
        #self.bn3 = nn.BatchNorm1d(self.fc3_units)
        #self.dp3 = nn.Dropout(p=0.5)

        #FC4
        self.layer4 = nn.Linear(self.fc3_units,  self.fc4_units, bias=True)
        self.layer5 = nn.Linear(self.fc4_units,  self.action_size, bias=True)

    #Given State-->action (Q values)
    def forward(self, state):
        """Build a network that maps state -> action values."""
        #return state
        layer1 = F.relu(self.layer1(state))
        layer2 = F.relu(self.layer2(layer1))
        layer3 = F.relu(self.layer3(layer2))
        layer4 = F.relu(self.layer4(layer3))
        action_values = self.layer5(layer4)
        return action_values


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

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params

  Saved successfully!                   sion of each state
                                        nsion of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)


        # Q-Network
        # Two Networks are needed because Q learning needs to update network itself
        # The first network updates faster than the second network
```

```
        # e.g. first netwrok updates every epoch whereas the second network only updates the
        # every 10 epoches.
        # .to(device) is a command for GPU
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)

        #optimization: Adam optimizer
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory: store the experience(action, state)
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    #update and store experience
    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.push(state, action, reward, next_state, done)#push into truple

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    #action
    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        #comvert the state from numpy data type to pytorch tensor
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        #call Q-Network
```



```
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples
```

```
        Update value parameters using given batch of experience tuples.
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        #pull out from truple
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss (mean square error)
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss: gradient descent
        self.optimizer.zero_grad()
        loss.backward()#-->update the local network
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target
        Params
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(), local_model.parameter
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

Saved successfully!

```
xperience 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.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward",
        self.seed = random.seed(seed)

    def push(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        #randomly select batch_size from experience
        experiences = random.sample(self.memory, k=self.batch_size)
        #corresponding states, actions, rewards, and next states.
        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None]))
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None]
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None]
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is r
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).as

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


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

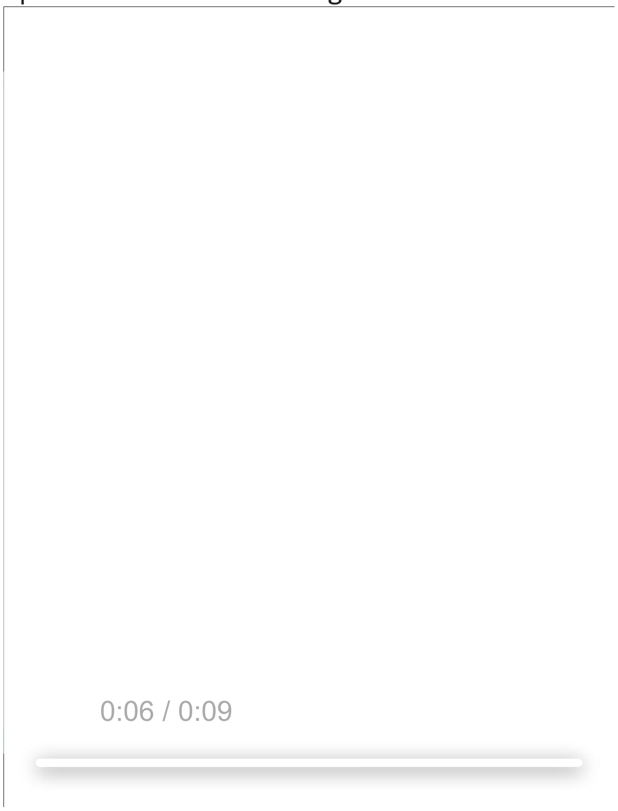
3. Train the Agent with DQN

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

    Params
    ======
                                    umber of training episodes
                                     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
```

Saved successfully!

```
    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=T
            state = env.reset()
        else:
            state = env.reset()
        score = 0
        for t in range(max_t):
          #based on epsioln to select action
            action = agent.act(state, eps)
            if t%100==0:
                action = 1
            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))
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_winc
            if render:
                env.close()
                show_video('output_%d' % i_episode)
                env = gym.make(atari_game)
        if np.mean(scores_window)>=10.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_ep
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores

#main train
agent = Agent(state_size=env.observation_space.shape[0], action_size=env.action_space.n, seec
scores = dqn()

Saved successfully!

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

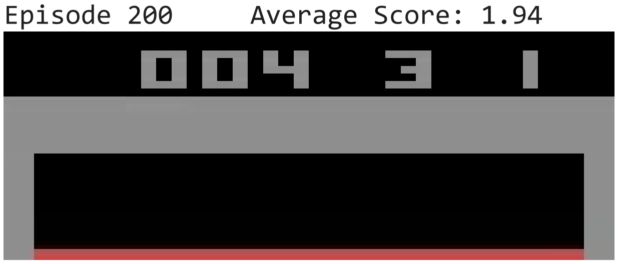
```
Episode 0       Average Score: 3.00
```

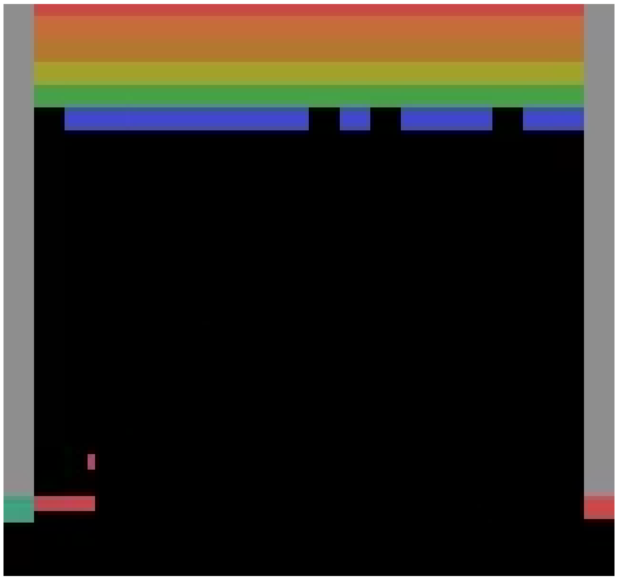



```
Episode 100     Average Score: 1.31
```

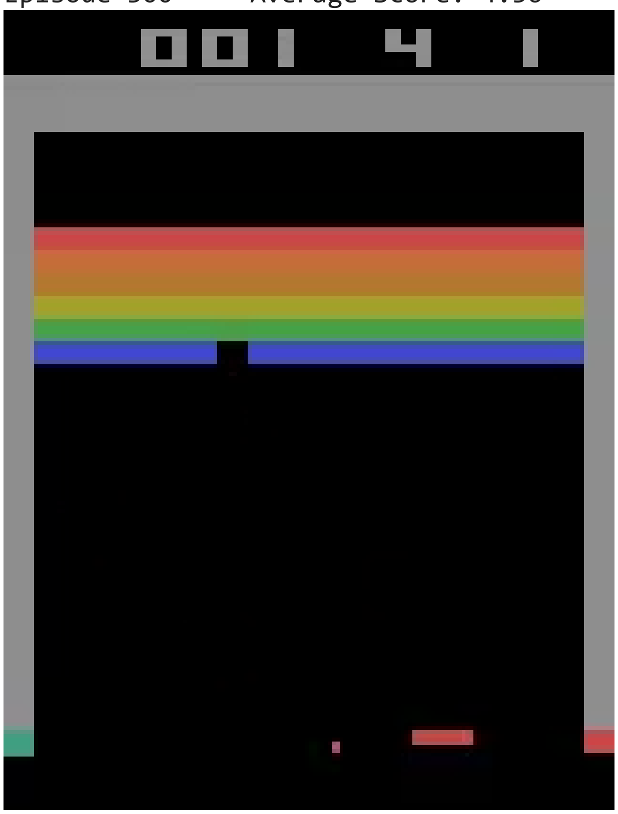
Saved successfully!

```
Episode 200     Average Score: 1.94
```





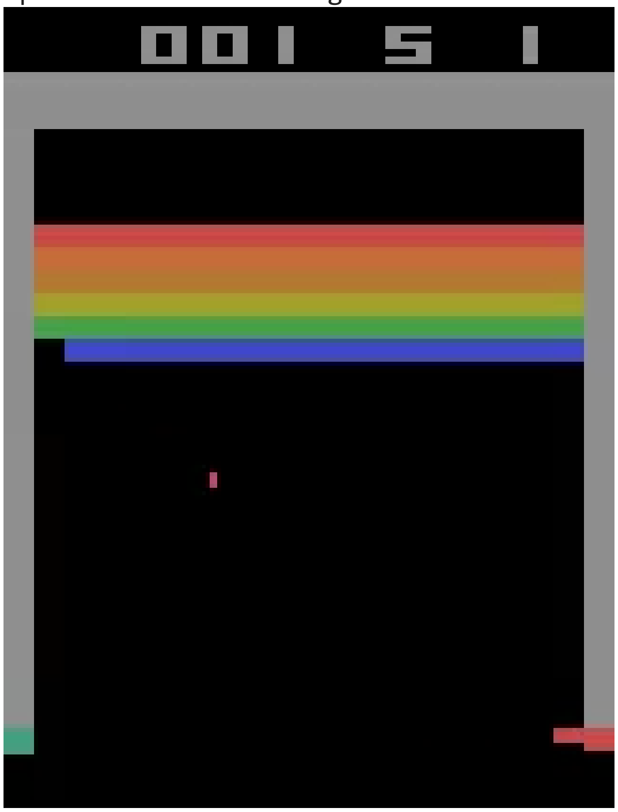
```
Episode 300     Average Score: 4.36
```




```
Episode 400     Average Score: 5.35
```

Saved successfully!

Episode 500 Average Score: 6.18

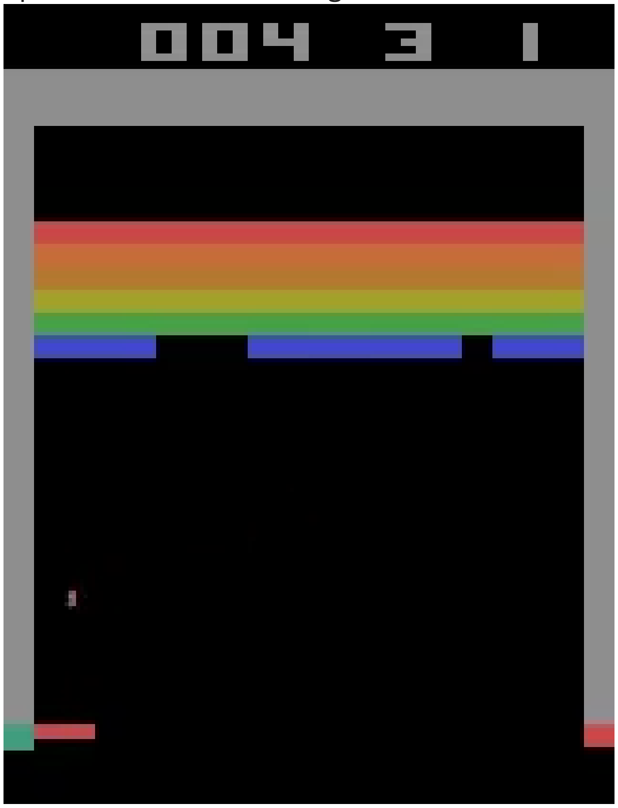


Episode 600 Average Score: 7.26

Saved successfully!

0:21 / 0:24

```
Episode 700	Average Score: 8.14
```

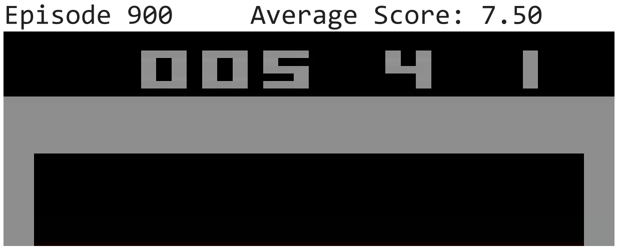


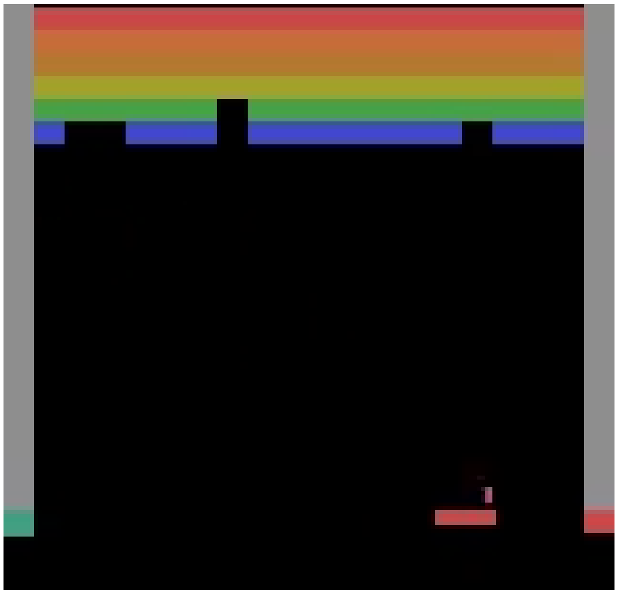

```
Episode 800	Average Score: 7.92
```

Saved successfully!

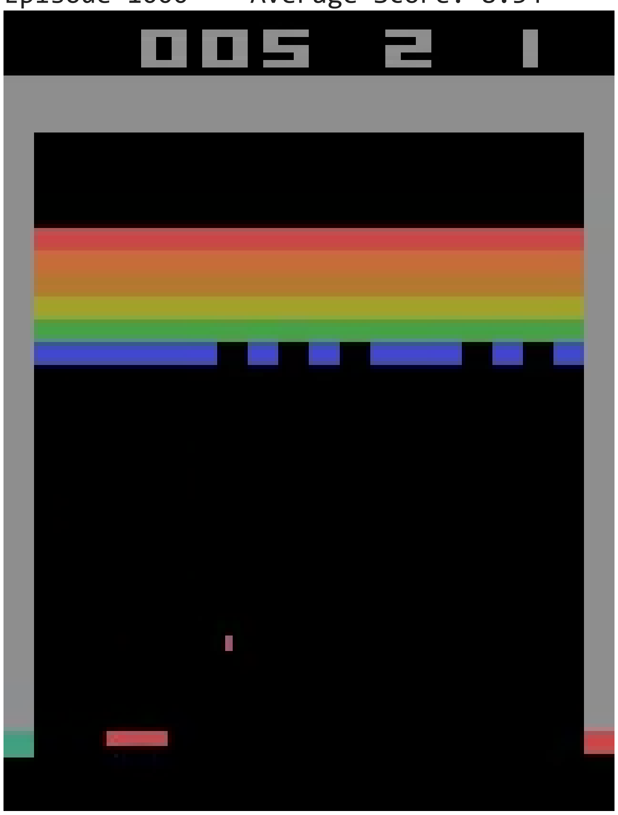
0:11 / 0:14

```
Episode 900	Average Score: 7.50
```





Episode 1000   Average Score: 8.34

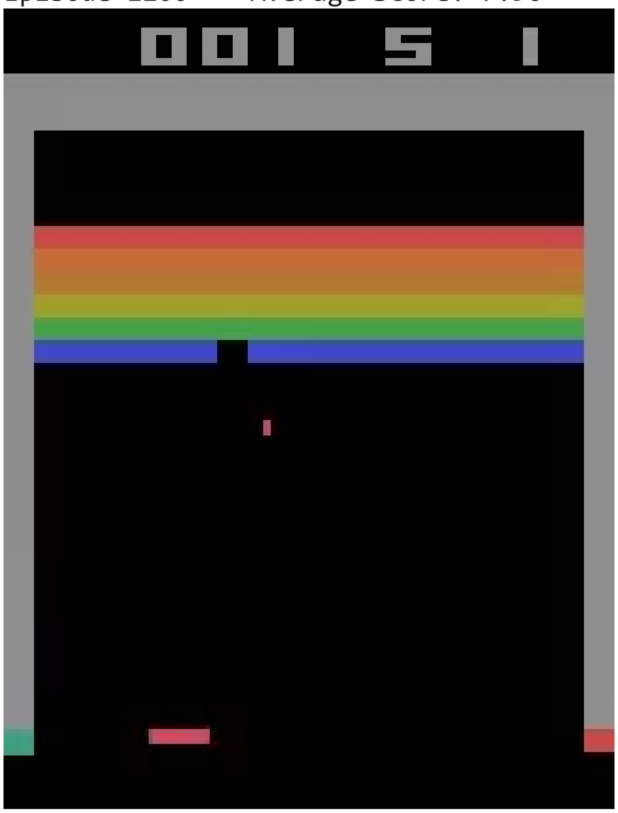


Episode 1100   Average Score: 8.21

Saved successfully!

0:12 / 0:17

Episode 1200   Average Score: 7.96

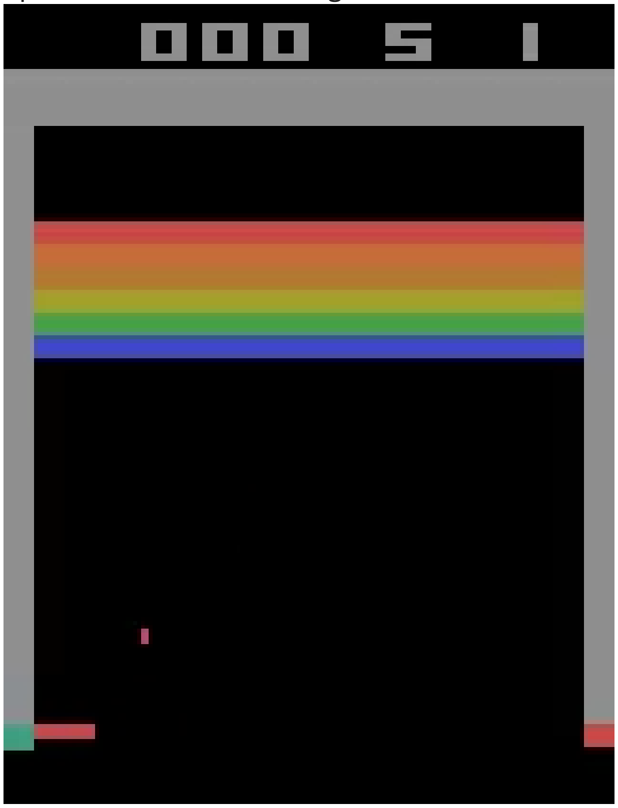


Episode 1300   Average Score: 7.44

Saved successfully!

0:08 / 0:29

```
Episode 1400    Average Score: 6.51
```




```
Episode 1500    Average Score: 4.54
```

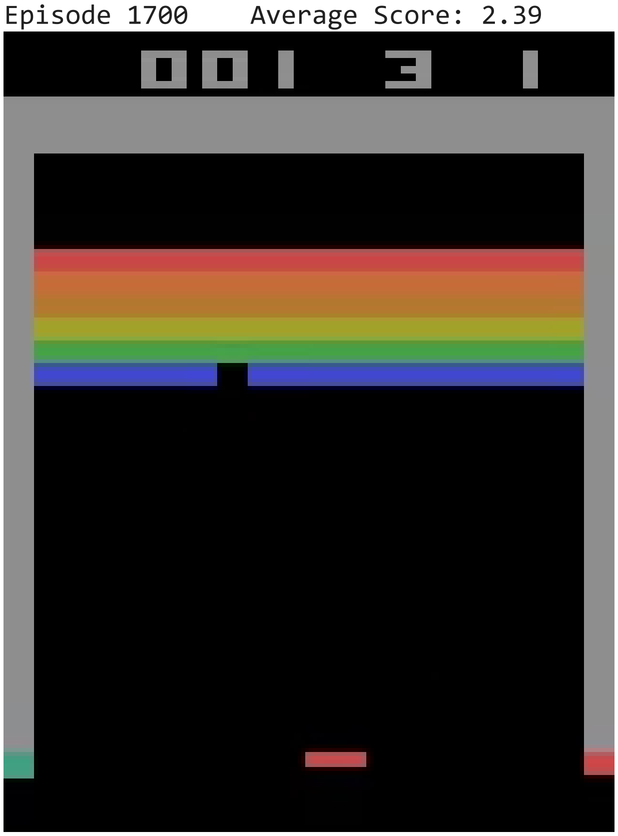
Saved successfully!

0:03 / 0:16

```
Episode 1600    Average Score: 2.87
```

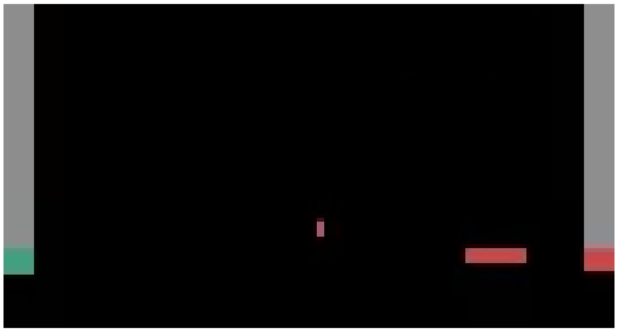
0:03 / 0:06

Episode 1700   Average Score: 2.39

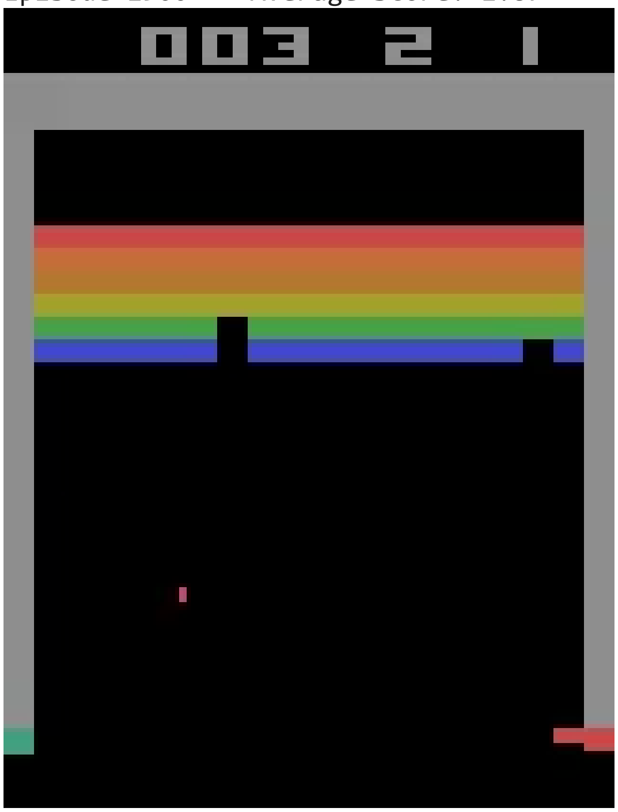


Episode 1800   Average Score: 1.91

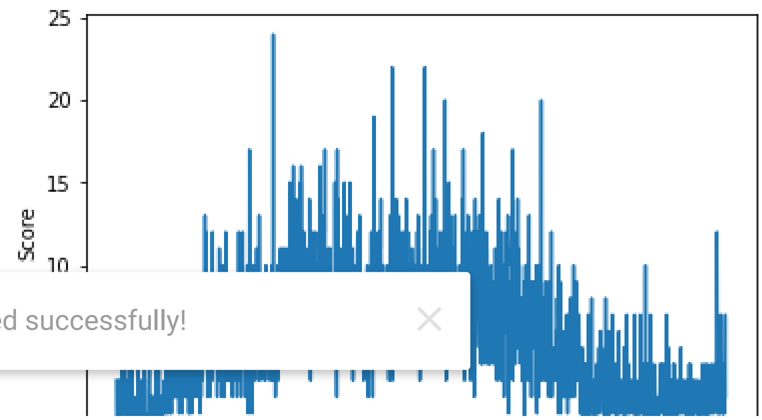
Saved successfully!



```
Episode 1900	Average Score: 1.87
```




```
Episode 1999	Average Score: 2.32
```



Saved successfully!

You can load the parameter by this line.

```
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))

for i in range(3):
    state = env.reset()
    for j in range(200):
```

```
        action = agent.act(state)
        env.render()
        state, reward, done, _ = env.step(action)
        if done:
            break
```

Policy gradient

This one is implemented in pure Python.

Define PG functions

```
import gym
#atari_game = "CartPole-v0"
atari_game = "Breakout-ram-v0"
env = gym.make(atari_game)

import numpy as np

class LogisticPolicy:

    def __init__(self, θ, α, γ):
        # Initialize paramters θ, learning rate α and discount factor γ
        # Initialization of policy parameters; learning rate and the deduction factor of futu
        self.θ = θ
        self.α = α
        self.γ = γ

    def logistic(self, y):
        # definition of logistic function
        # logit function for the probability of actions
        return 1/(1 + np.exp(-y))

    def probs(self, x):
        # returns probabilities of two actions

        y = x @ self.θ
```



```
        return np.array([prob0, 1-prob0])

    def act(self, x):
        # sample an action in proportion to probabilities

        probs = self.probs(x)
        action = np.random.choice([0, 1], p=probs)

        return action, probs[action]
```

```
    def grad_log_p(self, x):
        # calculate grad-log-probs
        y = x @ self.θ
        grad_log_p0 = x - x*self.logistic(y)
        grad_log_p1 = - x*self.logistic(y)

        return grad_log_p0, grad_log_p1

    def grad_log_p_dot_rewards(self, grad_log_p, actions, discounted_rewards):
        # dot grads with future rewards for each action in episode
        # discounted_rewards: predicted rewards with future uncertainty
        return grad_log_p.T @ discounted_rewards

    def discount_rewards(self, rewards):
        # calculate temporally adjusted, discounted rewards

        discounted_rewards = np.zeros(len(rewards))
        cumulative_rewards = 0
        for i in reversed(range(0, len(rewards))):
            cumulative_rewards = cumulative_rewards * self.γ + rewards[i]
            discounted_rewards[i] = cumulative_rewards

        return discounted_rewards

    def update(self, rewards, obs, actions):
        # calculate gradients for each action over all observations
        grad_log_p = np.array([self.grad_log_p(ob)[action] for ob,action in zip(obs,actions)]

        assert grad_log_p.shape == (len(obs), 4)

        # calculate temporaly adjusted, discounted rewards
        discounted_rewards = self.discount_rewards(rewards)

        # gradients times rewards
        dot = self.grad_log_p_dot_rewards(grad_log_p, actions, discounted_rewards)

        # gradient ascent on parameters
        self.θ += self.α*dot

                      =False):
```

Saved successfully!

```
    totalreward = 0
    #nitialization of observation, actions, rewards, and probabilities
    observations = []
    actions = []
    rewards = []
    probs = []

    done = False

    while not done:
```

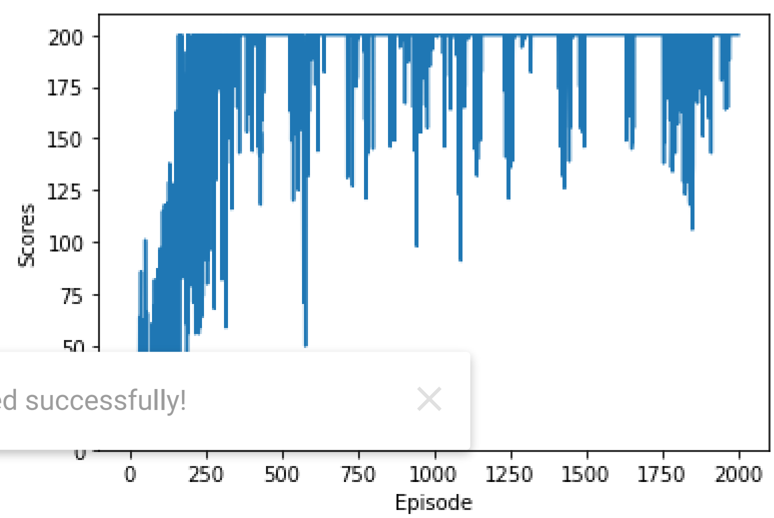
```
        if render:
            env.render()

        # add state
        observations.append(observation)
        # conduct action
        action, prob = policy.act(observation)
        observation, reward, done, info = env.step(action)
        #calculate rewards
        totalreward += reward
        rewards.append(reward)
        actions.append(action)
        probs.append(prob)

    return totalreward, np.array(rewards), np.array(observations), np.array(actions), np.arra

def train(θ, α, γ, Policy, MAX_EPISODES=1000, seed=None, evaluate=False):

    # initialize environment and policy
    #env = gym.make('CartPole-v0')
    atari_game = 'CartPole-v0'
    env = gym.wrappers.Monitor(gym.make(atari_game), 'sample', force=True)
    env.seed(0)
    if seed is not None:
        env.seed(seed)
    episode_rewards = []
    policy = Policy(θ, α, γ)

    # train until MAX_EPISODES
    for i in range(MAX_EPISODES):

        # run a single episode
        total_reward, rewards, observations, actions, probs = run_episode(env, policy)

        # keep track of episode rewards
        episode_rewards.append(total_reward)

        # update policy
        policy.update(rewards, observations, actions)
                                    core: " + str(total_reward) + " ",end="\r", flush=False)

    # evaluation call after training is finished - evaluate last trained policy on 100 episod
    if evaluate:
        env = Monitor(env, 'pg_cartpole/', video_callable=False, force=True)
        for _ in range(100):
            run_episode(env, policy, render=False)
        env.env.close()

    return episode_rewards, policy


from gym.wrappers.monitor import Monitor, load_results
```

Saved successfully!

```
from gym.wrappers.monitor import Monitor, load_results

# for reproducibility
GLOBAL_SEED = 0
np.random.seed(GLOBAL_SEED)

episode_rewards, policy = train(θ=np.random.rand(4),
                                α=0.002,
                                γ=0.99,
                                Policy=LogisticPolicy,
                                MAX_EPISODES=2000,
                                seed=GLOBAL_SEED,
                                evaluate=True)
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(episode_rewards);
results = load_results('pg_cartpole')
plt.hist(results['episode_rewards'], bins=20);
```

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.plot(episode_rewards);
results = load_results('pg_cartpole')
plt.hist(results['episode_rewards'], bins=20);
plt.xlabel('Episode')
plt.ylabel('Scores')
```

```
Text(0, 0.5, 'Scores')
```



Saved successfully!

Saved successfully!