rl

November 16, 2019

1 IE 534 HW: Reinforcement Learning

`v1, Designed by TIANQI WU, Fall 2019 at UIUC`

In this assignment, we will experiment with the (deep) reinforcement learning algorithms covered in the lecture. In particular, you will implement variants of the popular `DQN` (Deep Q-Network) (1) and `A2C` (Advantage Actor-Critic) (2) algorithms (by the same first author! orz), and test your implementation on both a small example (CartPole problem) and an Atari game (Breakout game). We focus on model-free algorithms rather than model-based ones, because neural nets are easier applicable and more popular nowadays in the model-free setting. (When the system dynamic is known or can be easily inferred, model-based can sometimes do better.)

The assignment breaks into **three parts**:

- **In Part I** (50 pts), you basically need to follow the instructions in this notebook to do a little bit of coding. We'll be able to see if your code trains by testing against the CartPole environment provided by the OpenAI gym package. We'll generate some plots that are required for grading.
- **In Part II** (40 pts), you'll copy your code onto Blue Waters (or actually any good server..), and run a much larger-scale experiment with the Breakout game. Hopefully, you can teach the computer to play Breakout in less than half a day! Share your final game score in this notebook. **This part will take at least a day. Please start early!!**
- **In Part III** (10 pts), you'll be asked to think about a few questions. These questions are mostly open-ended. Please write down your thoughts on them.

Finally, after you finished everything in this notebook **(code snippets C1-C5, plots P1-P5, question answers Q1-Q5)**, please save the notebook, and export to a PDF (or an HTML file), and submit:

1. the **.ipynb notebook and exported .pdf/.html file**, PDF is preferred (I usually do File -> Print Preview -> use Chrome's Save as PDF);
2. your code (**Algo.py, Model.py files**);
3. job artifacts (**.log files** only, pytorch models and images not required)

to Compass 2g for grading.

PS: Remember to save your notebook occasionally as you work through it!

References

- (1) 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), p.529.
- (2) Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, June. Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937).
- (3) A useful tutorial: https://spinningup.openai.com/en/latest/
- (4) *Useful code references*: https://github.com/deepmind/bsuite; https://github.com/openai/baselines; https://github.com/astooke/rlpyt;

First of all, **enter your NetID here** in the cell below: Your NetID: twu38

1.1 Part I: DQN and A2C on CartPole

This part is designed to run on your own local laptop/PC.

Before you start, there are some python dependencies: `pytorch`, `gym`, `numpy`, `multiprocessing`, `matplotlib`. Please install them correctly. You can install `pytorch` following instruction here https://pytorch.org/get-started/locally/. The code is compatible with PyTorch 0.4.x ~ 1.x. PyTorch 1.1 with cuda 10.0 worked for me (`conda install pytorch==1.1.0 torchvision==0.3.0 cudatoolkit=10.0 -c pytorch`).

Please **always** run the code cell below each time you open this notebook, to make sure `gym` is installed and to enable `autoreload` which **allows code changes to be effective immediately**. So if you changed `Algo.py` or `Model.py` but the test codes are not reflecting your changes, restart the notebook kernel and run this cell!!

In [1]:
```
# install openai gym
#%pip install gym
# enable autoreload
%load_ext autoreload
%autoreload 2
```

1.1.1 1.1 Code Structure

The code is structured in 5 python files:

- `Main.py`: contains the main entry point and training loop
- `Model.py`: constructs the torch neural network modules
- `Env.py`: contains the environment simulations interface, based on openai gym
- `Algo.py`: implements the DQN and A2C algorithms
- `Replay.py`: implements the experience replay buffer for DQN
- `Draw.py`: saves some game snapshots to jpeg files

Some parts of the code `Model.py` and `Algo.py` are left blank for you to complete. You are not required to modify the other parts (unless, of course, you want to boost the performance!). This is kind of a minimalist implementation, and might be different from the other code on the internet in details. You're welcomed to improve it, after you've finished all the required things of this assignment.

1.1.2 1.2 OpenAI gym and CartPole environment

OpenAI developed python package `gym` a while ago to facilitate RL research. `gym` provides a common interface between the program and the environments. For instance, the code cell below will create the CartPole environment. A window will show up when you run the code. The goal is to keep adjusting the cart so that the pole stays in its upright position.

A demo video from OpenAI:

`gym` also provides interface to Atari games. However, installing package `atari-py` is not easy on Windows/Mac, so we won't demonstrate it here. More info: http://gym.openai.com/docs/.

In [2]:
```
import time
import gym
env = gym.make('CartPole-v1')
env.reset()
for _ in range(200):
    env.render()
    state, reward, done, _ = env.step(env.action_space.sample()) # take a random actio
    if done: break
    time.sleep(0.15)
env.close()
```

1.1.3 1.3 Deep Q Learning

A little recap on DQN. We learned from lecture that Q-Learning is a model-free reinforcement learning algorithm. It falls into the off-policy type algorithm since it can utilize past experiences stored in a buffer. It also falls into the (approximate) dynamic programming type algorithm, since it tries to learn an optimal state-action value function using time difference (TD) errors. Q Learning is particularly interesting because it exploits the optimality structure in MDP. It's related to the Hamilton–Jacobi–Bellman equation in classical control.

For MDP

$$M = (S, A, P, r, \gamma)$$

where S is the state space, A is the action space, P is the transition dynamic, $r(s, a)$ is a reward function $S \times A \mapsto R$, and γ is the discount factor.

The tabular case (when S, A are finite), Q-Learning does the following value iteration update repeatedly when collecting experience (s_t, a_t, r_t) (η is the learning rate):

$$Q^{new}(s_t, a_t) \leftarrow Q^{old}(s_t, a_t) + \eta \left(r_t + \gamma \max_{a' \in A} Q^{old}(s_t, a') - Q^{old}(s_t, a_t) \right).$$

With function approximation, meaning model $Q(s, a)$ with a function $Q_\theta(s, a)$ parameterized by θ, we arrive at the Fitted Q Iteration (FQI) algorithm, or better known as Deep Q Learning if the function class is neural networks. Q-Learning with neural network as function approximator was known long ago, but it was only recently (year 2013) that DeepMind made this algorithm actually work on Atari games. Deep Q Learning iteratively optimize the following objective:

$$\theta_{new} \leftarrow \arg\min_{\theta} \mathbb{E}_{(s,a,r,s')\sim D} \left(r + \gamma \max_{a' \in A} Q_{\theta_{old}}(s', a') - Q_{\theta}(s, a) \right)^2 .$$

Therefore, with a batch of $\{(s^i, a^i, r^i, s'^i)\}_{i=1}^N$ sampled from the replay buffer, we can build a loss function L in pytorch:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(r^i + \gamma \max_{a' \in A} Q_{\theta_{old}}(s'^i, a') - Q_{\theta}(s^i, a^i) \right)^2 ,$$

and run the usual gradient descent on θ with a pytorch optimizer.

Exploration Exploration, as the word suggests, refers to explore novel unvisited states in RL. The FQI (or DQN) needs an exploratory datasets to work well. The common way to produce exploratory dataset is through randomization, such as the ϵ-greedy exploration strategy we will implement in this assignment. - ϵ-greedy exploration:

At training iteration it, the agent chooses to play

$$a = \begin{cases} \arg\max_a Q_{\theta}(s, a) & \text{with probability } 1 - \epsilon_{it} , \\ \text{a random action } a \in A & \text{with probability } \epsilon_{it} . \end{cases}$$

And ϵ_{it} is annealed, for example, linearly from 1 to 0.01 as training progresses until iteration it_{decay}:

$$\epsilon_{it} = \max \left\{ 0.01, 1 + (0.01 - 1) \frac{it}{it_{\text{decay}}} \right\}.$$

Two Caveats

1. There's an improvement on DQN called Double-DQN with the following loss L, which has shown to be empirically more stable than the original DQN loss described above. You may want to implement the improved one in your code:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(r^i + \gamma Q_{\theta_{old}}(s'^i, \arg\max_{a' \in A} Q_{\theta}(s'^i, a')) - Q_{\theta}(s^i, a^i) \right)^2 .$$

2. Huber loss (a.k.a smooth L1 loss) is commonly used to reduce the effect of extreme values:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} Huber \left(r^i + \gamma Q_{\theta_{old}}(s'^i, \arg\max_{a' \in A} Q_{\theta}(s'^i, a')) - Q_{\theta}(s^i, a^i) \right)$$

   You can look up the pytorch document here: https://pytorch.org/docs/stable/nn.functional.html#smooth-l1-loss

C1 (5 pts): Complete the code for the two layered fully connected network class `TwoLayerFCNet` in file `Model.py` And run the cell below to test the output shape of your module.

```
In [3]: ## Test code
        from Model import TwoLayerFCNet
        import torch
        net = TwoLayerFCNet(n_in=4, n_hidden=16, n_out=5)
        x = torch.randn(10, 4)
        y = net(x)
        assert y.shape == (10, 5), "ERROR: network output has the wrong shape!"
        print ("Output shape test passed!")
```

```
Output shape test passed!
```

C2 (5 pts): Complete the code for ϵ-greedy exploration strategy in function `DQN.act` in file 'Algo.py' And run the cell below to test it.

```
In [4]: ## Test code
        from Algo import DQN
        class Nothing: pass
        dummy = Nothing()
        dummy.eps_decay = 500000

        dummy.num_act_steps = 0
        eps = DQN.compute_epsilon(dummy)
        assert abs( eps - 1.0 ) < 0.01, "ERROR: compute_epsilon at t=0 should be 1 but got %f!'

        dummy.num_act_steps = 250000
        eps = DQN.compute_epsilon(dummy)
        assert abs( eps - 0.505 ) < 0.01, "ERROR: compute_epsilon at t=250000 should around .5(

        dummy.num_act_steps = 500000
        eps = DQN.compute_epsilon(dummy)
        assert abs( eps - 0.01 ) < 0.01, "ERROR: compute_epsilon at t=500000 should be .01 but

        dummy.num_act_steps = 600000
        eps = DQN.compute_epsilon(dummy)
        assert abs( eps - 0.01 ) < 0.01, "ERROR: compute_epsilon after t=500000 should be .01 l
        print ("Epsilon-greedy test passed!")
```

```
Epsilon-greedy test passed!
```

C3 (10 pts): Complete the code for computing the loss function in `DQN.train` in file `Algo.py` And run the cell below to verify your code decreses the loss value in one iteration.

```
In [5]: import numpy as np
        from Algo import DQN
```

```
class Nothing: pass
dummy_obs_space, dummy_act_space = Nothing(), Nothing()
dummy_obs_space.shape = [10]
dummy_act_space.n = 3

dqn = DQN(dummy_obs_space, dummy_act_space, batch_size=2)

for t in range(3):
    dqn.observe([np.random.randn(10).astype('float32')], [np.random.randint(3)],
                [(np.random.randn(10).astype('float32'), np.random.rand(), False, None)

b = dqn.replay.cur_batch
loss1 = dqn.train()
dqn.replay.cur_batch = b
loss2 = dqn.train()

print (loss1, '>', loss2, '?')
assert loss2 < loss1, "DQN.train should reduce loss on the same batch"

print ("DQN.train test passed!")
```

```
parameters to optimize: [('fc1.weight', torch.Size([128, 10]), True), ('fc1.bias', torch.Size(

0.05873310565948486 > 0.056507933884859085 ?
DQN.train test passed!
```

P1 (10 pts): Run DQN on CartPole and plot the learning curve (i.e. averaged episodic reward against env steps). Your code should be able to achieve **>150** averaged reward in 10000 iterations (20000 simulation steps) in only a few minutes. This is a good indication that the implementation is correct. It's ok that the curve is not always monotonically increasing because of randomness in training.

```
In [6]: %run Main.py  \
            --niter 10000   \
            --env CartPole-v1   \
            --algo dqn  \
            --nproc 2   \
            --lr 0.001  \
            --train_freq 1  \
            --train_start 100   \
            --replay_size 20000 \
            --batch_size 64     \
            --discount 0.996    \
            --target_update 1000    \
            --eps_decay 4000    \
            --print_freq 200    \
            --checkpoint_freq 20000 \
```

```
        --save_dir cartpole_dqn \
        --log log.txt \
        --parallel_env 0
```

```
Namespace(algo='dqn', batch_size=64, checkpoint_freq=20000, discount=0.996, ent_coef=0.01, env=
observation space: Box(4,)
action space: Discrete(2)
running on device cpu
parameters to optimize: [('fc1.weight', torch.Size([128, 4]), True), ('fc1.bias', torch.Size([

obses on reset: 2 x (4,) float32
iter    200 |loss   0.01 |n_ep    15 |ep_len   21.4 |ep_rew  21.38 |raw_ep_rew  21.38 |env_step
iter    400 |loss   0.00 |n_ep    29 |ep_len   27.0 |ep_rew  27.01 |raw_ep_rew  27.01 |env_step
iter    600 |loss   0.00 |n_ep    44 |ep_len   29.3 |ep_rew  29.33 |raw_ep_rew  29.33 |env_step
iter    800 |loss   0.00 |n_ep    63 |ep_len   21.6 |ep_rew  21.60 |raw_ep_rew  21.60 |env_step
iter   1000 |loss   0.00 |n_ep    86 |ep_len   16.0 |ep_rew  15.95 |raw_ep_rew  15.95 |env_step
iter   1200 |loss   0.03 |n_ep   110 |ep_len   17.6 |ep_rew  17.64 |raw_ep_rew  17.64 |env_step
iter   1400 |loss   0.05 |n_ep   137 |ep_len   14.7 |ep_rew  14.67 |raw_ep_rew  14.67 |env_step
iter   1600 |loss   0.05 |n_ep   165 |ep_len   15.3 |ep_rew  15.29 |raw_ep_rew  15.29 |env_step
iter   1800 |loss   0.02 |n_ep   186 |ep_len   16.4 |ep_rew  16.38 |raw_ep_rew  16.38 |env_step
iter   2000 |loss   0.04 |n_ep   211 |ep_len   15.5 |ep_rew  15.48 |raw_ep_rew  15.48 |env_step
iter   2200 |loss   0.06 |n_ep   241 |ep_len   13.9 |ep_rew  13.92 |raw_ep_rew  13.92 |env_step
iter   2400 |loss   0.05 |n_ep   266 |ep_len   16.6 |ep_rew  16.58 |raw_ep_rew  16.58 |env_step
iter   2600 |loss   0.06 |n_ep   281 |ep_len   26.5 |ep_rew  26.55 |raw_ep_rew  26.55 |env_step
iter   2800 |loss   0.04 |n_ep   301 |ep_len   20.6 |ep_rew  20.57 |raw_ep_rew  20.57 |env_step
iter   3000 |loss   0.06 |n_ep   319 |ep_len   24.1 |ep_rew  24.07 |raw_ep_rew  24.07 |env_step
iter   3200 |loss   0.08 |n_ep   329 |ep_len   29.9 |ep_rew  29.91 |raw_ep_rew  29.91 |env_step
iter   3400 |loss   0.01 |n_ep   335 |ep_len   45.9 |ep_rew  45.93 |raw_ep_rew  45.93 |env_step
iter   3600 |loss   0.14 |n_ep   339 |ep_len   67.6 |ep_rew  67.63 |raw_ep_rew  67.63 |env_step
iter   3800 |loss   0.08 |n_ep   346 |ep_len   67.1 |ep_rew  67.13 |raw_ep_rew  67.13 |env_step
iter   4000 |loss   0.02 |n_ep   351 |ep_len   59.1 |ep_rew  59.12 |raw_ep_rew  59.12 |env_step
iter   4200 |loss   0.22 |n_ep   357 |ep_len   61.8 |ep_rew  61.77 |raw_ep_rew  61.77 |env_step
iter   4400 |loss   0.08 |n_ep   360 |ep_len   76.3 |ep_rew  76.28 |raw_ep_rew  76.28 |env_step
iter   4600 |loss   0.06 |n_ep   364 |ep_len  100.8 |ep_rew 100.85 |raw_ep_rew 100.85 |env_step
iter   4800 |loss   0.07 |n_ep   366 |ep_len  108.9 |ep_rew 108.87 |raw_ep_rew 108.87 |env_step
iter   5000 |loss   0.07 |n_ep   368 |ep_len  113.0 |ep_rew 113.02 |raw_ep_rew 113.02 |env_step
iter   5200 |loss   0.03 |n_ep   370 |ep_len  133.4 |ep_rew 133.39 |raw_ep_rew 133.39 |env_step
iter   5400 |loss   0.19 |n_ep   372 |ep_len  149.3 |ep_rew 149.27 |raw_ep_rew 149.27 |env_step
iter   5600 |loss   0.07 |n_ep   374 |ep_len  160.2 |ep_rew 160.25 |raw_ep_rew 160.25 |env_step
iter   5800 |loss   0.07 |n_ep   377 |ep_len  158.4 |ep_rew 158.44 |raw_ep_rew 158.44 |env_step
iter   6000 |loss   0.02 |n_ep   378 |ep_len  162.5 |ep_rew 162.50 |raw_ep_rew 162.50 |env_step
iter   6200 |loss   0.14 |n_ep   379 |ep_len  179.8 |ep_rew 179.85 |raw_ep_rew 179.85 |env_step
iter   6400 |loss   0.05 |n_ep   381 |ep_len  183.7 |ep_rew 183.67 |raw_ep_rew 183.67 |env_step
iter   6600 |loss   0.22 |n_ep   383 |ep_len  204.8 |ep_rew 204.81 |raw_ep_rew 204.81 |env_step
iter   6800 |loss   0.02 |n_ep   385 |ep_len  199.2 |ep_rew 199.18 |raw_ep_rew 199.18 |env_step
iter   7000 |loss   0.09 |n_ep   387 |ep_len  199.6 |ep_rew 199.55 |raw_ep_rew 199.55 |env_step
iter   7200 |loss   0.03 |n_ep   389 |ep_len  206.9 |ep_rew 206.95 |raw_ep_rew 206.95 |env_step
iter   7400 |loss   0.06 |n_ep   391 |ep_len  199.9 |ep_rew 199.87 |raw_ep_rew 199.87 |env_step
```

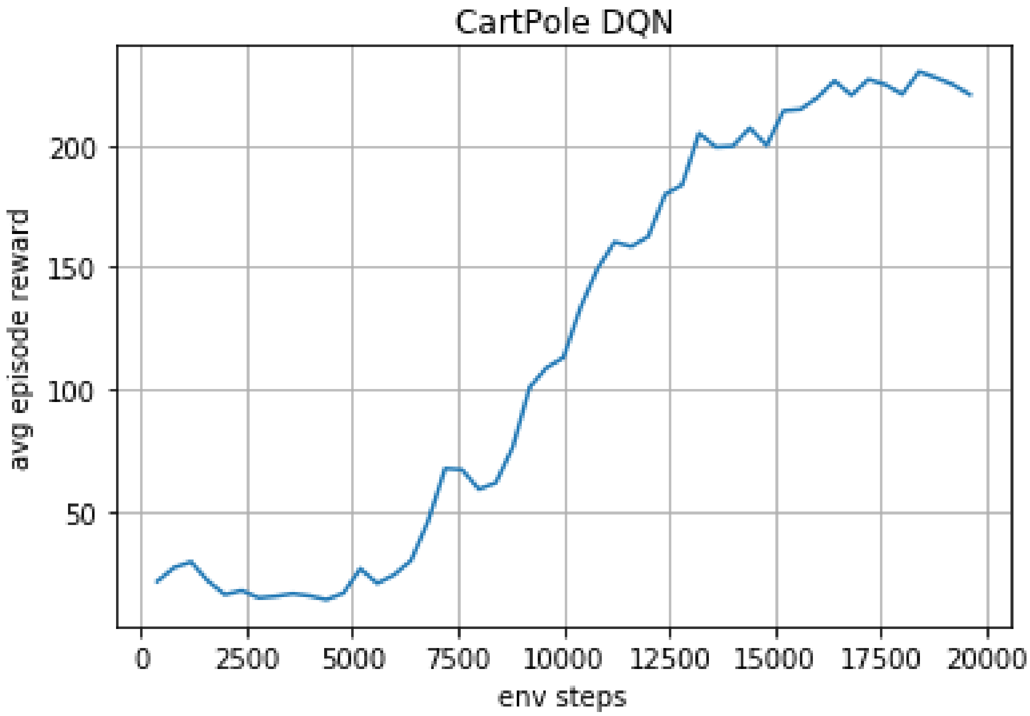
```
iter   7600 |loss   0.51 |n_ep    393 |ep_len  214.0 |ep_rew 214.02 |raw_ep_rew 214.02 |env_step
iter   7800 |loss   0.09 |n_ep    394 |ep_len  214.6 |ep_rew 214.62 |raw_ep_rew 214.62 |env_step
iter   8000 |loss   0.02 |n_ep    395 |ep_len  219.5 |ep_rew 219.46 |raw_ep_rew 219.46 |env_step
iter   8200 |loss   0.09 |n_ep    397 |ep_len  226.3 |ep_rew 226.30 |raw_ep_rew 226.30 |env_step
iter   8400 |loss   0.43 |n_ep    399 |ep_len  220.4 |ep_rew 220.38 |raw_ep_rew 220.38 |env_step
iter   8600 |loss   0.28 |n_ep    401 |ep_len  226.9 |ep_rew 226.87 |raw_ep_rew 226.87 |env_step
iter   8800 |loss   0.07 |n_ep    403 |ep_len  224.7 |ep_rew 224.74 |raw_ep_rew 224.74 |env_step
iter   9000 |loss   0.03 |n_ep    405 |ep_len  220.8 |ep_rew 220.81 |raw_ep_rew 220.81 |env_step
iter   9200 |loss   0.05 |n_ep    407 |ep_len  230.0 |ep_rew 230.04 |raw_ep_rew 230.04 |env_step
iter   9400 |loss   0.10 |n_ep    408 |ep_len  227.5 |ep_rew 227.54 |raw_ep_rew 227.54 |env_step
iter   9600 |loss   0.04 |n_ep    410 |ep_len  224.8 |ep_rew 224.76 |raw_ep_rew 224.76 |env_step
iter   9800 |loss   0.04 |n_ep    412 |ep_len  220.6 |ep_rew 220.61 |raw_ep_rew 220.61 |env_step
save checkpoint to cartpole_dqn/9999.pth
```

In [7]:
```python
import matplotlib.pyplot as plt

def plot_curve(logfile, title=None):
    lines = open(logfile, 'r').readlines()
    lines = [l.split() for l in lines if l[:4] == 'iter']
    steps = [int(l[13]) for l in lines]
    rewards = [float(l[11]) for l in lines]
    plt.plot(steps, rewards)
    plt.xlabel('env steps'); plt.ylabel('avg episode reward'); plt.grid(True)
    if title: plt.title(title)
    plt.show()
```

The log is saved to `'cartpole_dqn/log.txt'`. Let's plot the running averaged episode reward curve during training:

In [8]:
```python
plot_curve('cartpole_dqn/log.txt', 'CartPole DQN')
```



1.1.4 1.4 Actor-Critic Algorithm

Policy gradient methods are another class of algorithms that originated from viewing the RL problem as a mathematical optimization problem. Recall that the objective of RL is to maximize the expected cumulative reward the agent gets, namely

$$\max_{\pi} J(\pi) := \mathbb{E}_{(s_t,a_t,r_t)\sim D^{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

where D^{π} is the distribution of trajectories induced by policy π, and inside the expectation is the random variable representing the discounted cumulative reward and J is the reward (or cost) functional. Essentially, we want to optimize the policy π.

The most straightforward way is to run gradient update on the parameter θ of a *parameterized* policy π_{θ}. One such algorithm is the so-called `Advantage Actor-Critic (A2C)`. A2C is an on-policy policy optimization type algorithm. While collecting on-policy data, we iteratively run gradient ascent:

$$\theta_{new} \leftarrow \theta_{old} + \eta \hat{\nabla}_{\theta} J(\pi_{\theta_{old}})$$

with a Monte Carlo estimate $\hat{\nabla}_{\theta} J$ of the true gradient $\nabla_{\theta} J$. The true gradient writes as (by Policy Gradient Theorem and some manipulations):

$$\nabla_{\theta} J(\pi_{\theta_{old}}) = \mathbb{E}_{(s_t,a_t,r_t)\sim D^{\pi_{\theta_{old}}}} \sum_{t=0}^{\infty} \left(\nabla_{\theta} \log \pi_{\theta_{old}}(s_t, a_t) \left(\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} - V^{\pi_{\theta_{old}}}(s_t) \right) \right).$$

The quantity in the inner-most parentheses $A(s_t, a_t) = Q(s_t, a_t) - V(s_t) = (\mathbb{E} \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}) - V(s_t)$ is called the *Advantage* function (not very rigoriously speaking...). That's why it's called **Advantage** Actor-Critic. More on A2C: https://arxiv.org/abs/1506.02438.

And the Monte Carlo estimate of the gradient is

$$\hat{\nabla}_\theta J(\pi_{\theta_{old}}) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T} \left(\nabla_\theta \log \pi_{\theta_{old}}(s_t^i, a_t^i) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^i - V_{\phi_{old}}(s_t^i) \right) \right)$$

where $V_{\phi_{old}}$ is introduced as a *parameterized* estimate for $V^{\pi_{\theta_{old}}}$, which can also be a neural network. So V_ϕ is the **critic** and π_θ is the **actor**. We can construct a specific loss function in pytorch that gives $\hat{\nabla}_\theta J$. $V_{\phi_{old}}$ is trained with SGD on a L2 loss function. It's further common practice to add an entropy bonus loss term to encourage maximum entropy solution, to facilitate exploration and avoid getting stuck in local minima. We shall clarify these loss functions in the following summarization.

Summarizing a variant of the A2C algorithm:

For many iterations repeat: 1. Collect N independent trajectories $\{(s_t^i, a_t^i, r_t^i)_{t=0}^{T}\}_{i=1}^{N}$ by running policy π_θ for maximum T steps; 2. Compute the loss function for the policy parameter θ:

$$L_{policy}(\theta) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T} \left(\log \pi_\theta(s_t^i, a_t^i) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^i - V_\phi(s_t^i) \right) \right)$$

Compute the entropy term for θ:

$$L_{entropy}(\theta) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T} \left(- \sum_{a \in A} \pi_\theta(s_t^i, a) \log \pi_\theta(s_t^i, a) \right)$$

Compute the loss for value function parameter ϕ:

$$L_{value}(\phi) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=0}^{T} \left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^i - V_\phi(s_t^i) \right)^2$$

3. Use pytorch auto-differentiation and optimizer to do one gradient step on (θ, ϕ) with the overall loss:

$$L(\theta, \phi) = -L_{policy} - \lambda_{ent} L_{entropy} + \lambda_{val} L_{value}$$

where λ_{ent} and λ_{val} are coefficients to balances the loss terms.

C4 (10 pts): Complete the code for computing the advantange, entropy and loss function in `A2C.train` in file `Algo.py`

```
In [ ]:
```

P2 (10 pts): Run A2C on CartPole and plot the learning curve (i.e. averaged episodic reward against training iteration). Your code should be able to achieve **>150** averaged reward in 10000 iterations (40000 simulation steps) in only a few minutes. This is a good indication that the implementation is correct.

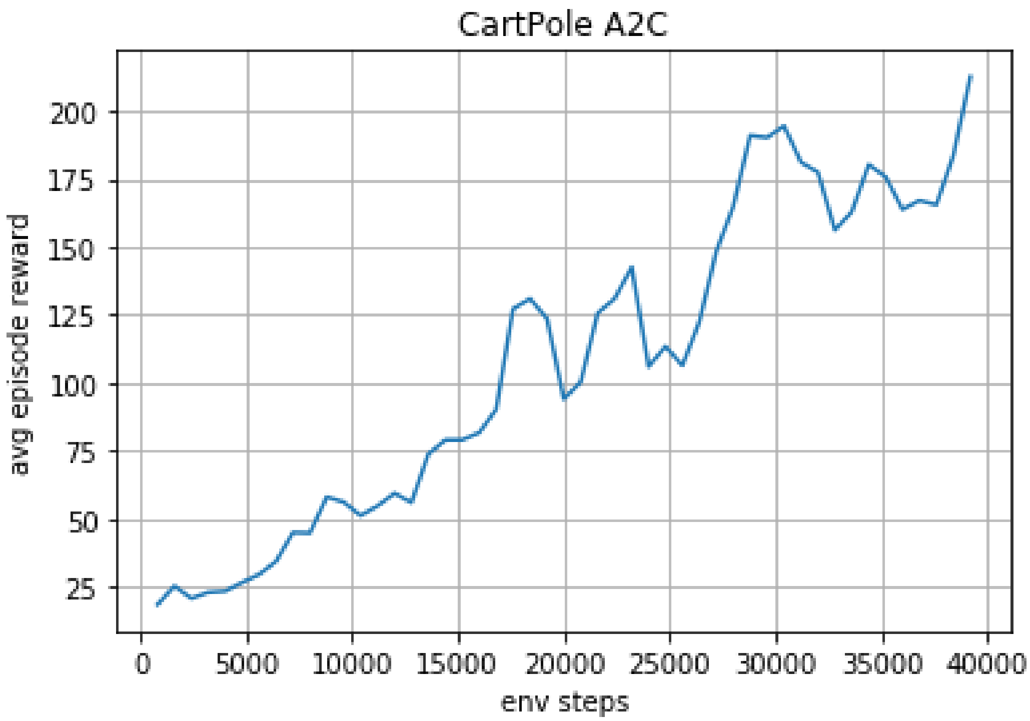
```
In [9]: %run Main.py  \
            --niter 10000   \
            --env CartPole-v1   \
            --algo a2c  \
            --nproc 4   \
            --lr 0.001  \
            --train_freq 16 \
            --train_start 0 \
            --batch_size 64     \
            --discount 0.996    \
            --value_coef 0.01    \
            --print_freq 200    \
            --checkpoint_freq 20000 \
            --save_dir cartpole_a2c \
            --log log.txt \
            --parallel_env 0
```

```
Namespace(algo='a2c', batch_size=64, checkpoint_freq=20000, discount=0.996, ent_coef=0.01, env=
observation space: Box(4,)
action space: Discrete(2)
running on device cpu
shared net = False, parameters to optimize: [('fc1.weight', torch.Size([128, 4]), True), ('fc1

obses on reset: 4 x (4,) float32
iter    200 |loss   0.87 |n_ep    38 |ep_len   18.4 |ep_rew  18.43 |raw_ep_rew  18.43 |env_ste
iter    400 |loss   0.94 |n_ep    77 |ep_len   25.3 |ep_rew  25.26 |raw_ep_rew  25.26 |env_ste
iter    600 |loss   0.79 |n_ep   116 |ep_len   20.7 |ep_rew  20.73 |raw_ep_rew  20.73 |env_ste
iter    800 |loss   0.77 |n_ep   155 |ep_len   22.9 |ep_rew  22.95 |raw_ep_rew  22.95 |env_ste
iter   1000 |loss   0.65 |n_ep   186 |ep_len   23.4 |ep_rew  23.38 |raw_ep_rew  23.38 |env_ste
iter   1200 |loss   0.67 |n_ep   219 |ep_len   26.6 |ep_rew  26.57 |raw_ep_rew  26.57 |env_ste
iter   1400 |loss   0.67 |n_ep   246 |ep_len   29.5 |ep_rew  29.50 |raw_ep_rew  29.50 |env_ste
iter   1600 |loss   0.80 |n_ep   268 |ep_len   34.3 |ep_rew  34.31 |raw_ep_rew  34.31 |env_ste
iter   1800 |loss   0.85 |n_ep   288 |ep_len   45.0 |ep_rew  44.96 |raw_ep_rew  44.96 |env_ste
iter   2000 |loss   1.02 |n_ep   303 |ep_len   44.8 |ep_rew  44.79 |raw_ep_rew  44.79 |env_ste
iter   2200 |loss   0.64 |n_ep   319 |ep_len   58.1 |ep_rew  58.08 |raw_ep_rew  58.08 |env_ste
iter   2400 |loss   0.93 |n_ep   332 |ep_len   56.1 |ep_rew  56.14 |raw_ep_rew  56.14 |env_ste
iter   2600 |loss   0.95 |n_ep   346 |ep_len   51.2 |ep_rew  51.18 |raw_ep_rew  51.18 |env_ste
iter   2800 |loss   1.03 |n_ep   359 |ep_len   54.8 |ep_rew  54.75 |raw_ep_rew  54.75 |env_ste
iter   3000 |loss   0.93 |n_ep   372 |ep_len   59.4 |ep_rew  59.40 |raw_ep_rew  59.40 |env_ste
iter   3200 |loss   0.59 |n_ep   387 |ep_len   56.0 |ep_rew  56.01 |raw_ep_rew  56.01 |env_ste
iter   3400 |loss   0.66 |n_ep   396 |ep_len   73.8 |ep_rew  73.80 |raw_ep_rew  73.80 |env_ste
iter   3600 |loss   0.99 |n_ep   403 |ep_len   79.0 |ep_rew  79.02 |raw_ep_rew  79.02 |env_ste
iter   3800 |loss   0.67 |n_ep   415 |ep_len   79.0 |ep_rew  79.05 |raw_ep_rew  79.05 |env_ste
```

```
iter   4000 |loss   0.99 |n_ep   423 |ep_len   81.7 |ep_rew   81.68 |raw_ep_rew   81.68 |env_ste
iter   4200 |loss   0.40 |n_ep   431 |ep_len   90.3 |ep_rew   90.26 |raw_ep_rew   90.26 |env_ste
iter   4400 |loss   0.96 |n_ep   434 |ep_len  127.3 |ep_rew  127.26 |raw_ep_rew  127.26 |env_ste
iter   4600 |loss   0.35 |n_ep   440 |ep_len  131.0 |ep_rew  130.98 |raw_ep_rew  130.98 |env_ste
iter   4800 |loss   0.45 |n_ep   448 |ep_len  123.7 |ep_rew  123.73 |raw_ep_rew  123.73 |env_ste
iter   5000 |loss   0.57 |n_ep   458 |ep_len   94.1 |ep_rew   94.08 |raw_ep_rew   94.08 |env_ste
iter   5200 |loss   0.30 |n_ep   464 |ep_len  100.4 |ep_rew  100.37 |raw_ep_rew  100.37 |env_ste
iter   5400 |loss   0.85 |n_ep   471 |ep_len  125.7 |ep_rew  125.67 |raw_ep_rew  125.67 |env_ste
iter   5600 |loss   1.07 |n_ep   475 |ep_len  131.1 |ep_rew  131.09 |raw_ep_rew  131.09 |env_ste
iter   5800 |loss   0.85 |n_ep   480 |ep_len  142.6 |ep_rew  142.61 |raw_ep_rew  142.61 |env_ste
iter   6000 |loss   0.49 |n_ep   491 |ep_len  106.0 |ep_rew  106.02 |raw_ep_rew  106.02 |env_ste
iter   6200 |loss   0.07 |n_ep   496 |ep_len  113.4 |ep_rew  113.37 |raw_ep_rew  113.37 |env_ste
iter   6400 |loss   0.88 |n_ep   504 |ep_len  106.4 |ep_rew  106.40 |raw_ep_rew  106.40 |env_ste
iter   6600 |loss   0.19 |n_ep   508 |ep_len  122.5 |ep_rew  122.50 |raw_ep_rew  122.50 |env_ste
iter   6800 |loss  -0.07 |n_ep   514 |ep_len  148.4 |ep_rew  148.37 |raw_ep_rew  148.37 |env_ste
iter   7000 |loss   0.73 |n_ep   517 |ep_len  164.9 |ep_rew  164.85 |raw_ep_rew  164.85 |env_ste
iter   7200 |loss   0.90 |n_ep   521 |ep_len  191.0 |ep_rew  191.01 |raw_ep_rew  191.01 |env_ste
iter   7400 |loss   0.03 |n_ep   526 |ep_len  190.3 |ep_rew  190.25 |raw_ep_rew  190.25 |env_ste
iter   7600 |loss  -0.03 |n_ep   527 |ep_len  194.5 |ep_rew  194.53 |raw_ep_rew  194.53 |env_ste
iter   7800 |loss   0.61 |n_ep   533 |ep_len  181.2 |ep_rew  181.21 |raw_ep_rew  181.21 |env_ste
iter   8000 |loss   0.64 |n_ep   538 |ep_len  177.5 |ep_rew  177.53 |raw_ep_rew  177.53 |env_ste
iter   8200 |loss  -0.07 |n_ep   544 |ep_len  156.2 |ep_rew  156.22 |raw_ep_rew  156.22 |env_ste
iter   8400 |loss   0.74 |n_ep   547 |ep_len  162.9 |ep_rew  162.94 |raw_ep_rew  162.94 |env_ste
iter   8600 |loss   0.13 |n_ep   553 |ep_len  180.4 |ep_rew  180.35 |raw_ep_rew  180.35 |env_ste
iter   8800 |loss   0.75 |n_ep   557 |ep_len  175.9 |ep_rew  175.87 |raw_ep_rew  175.87 |env_ste
iter   9000 |loss   0.14 |n_ep   562 |ep_len  163.8 |ep_rew  163.84 |raw_ep_rew  163.84 |env_ste
iter   9200 |loss  -0.07 |n_ep   568 |ep_len  167.1 |ep_rew  167.14 |raw_ep_rew  167.14 |env_ste
iter   9400 |loss   0.73 |n_ep   570 |ep_len  165.6 |ep_rew  165.58 |raw_ep_rew  165.58 |env_ste
iter   9600 |loss   0.76 |n_ep   573 |ep_len  183.6 |ep_rew  183.61 |raw_ep_rew  183.61 |env_ste
iter   9800 |loss   0.71 |n_ep   576 |ep_len  212.7 |ep_rew  212.71 |raw_ep_rew  212.71 |env_ste
save checkpoint to cartpole_a2c/9999.pth
```

```
In [10]: plot_curve('cartpole_a2c/log.txt', 'CartPole A2C')
```



Now let's play a little bit with the trained agent. The neural net parameters are saved to the `cartpole_dqn` and `cartpole_a2c` folders. The cell below will open a window showing one episode play.

```
In [11]: import time
         import gym
         import Algo
         env = gym.make('CartPole-v1')
         agent = Algo.ActorCritic(env.observation_space, env.action_space)
         agent.load('cartpole_a2c/9999.pth')
         state = env.reset()
         for _ in range(120):
             env.render()
             state, reward, done, _ = env.step(agent.act([state])[0])
             if done: break
             time.sleep(0.1)
         env.close()
```

```
shared net = False, parameters to optimize: [('fc1.weight', torch.Size([128, 4]), True), ('fc1
```

1.2 Part II: Solve the Atari Breakout game

In this part, you'll train your agent to play Breakout with the BlueWaters cluster. I have provided the job scripts for you. Please upload your `Algo.py` and `Model.py` completed in **Part I** to your BlueWaters folder. And submit the following two jobs respectively:

```
qsub run_dqn.pbs
qsub run_a2c.pbs
```

The jobs are set to run for at most **14 hours**. **Please start early!!** You might be able to reach the desired score (>= 200 reward) before 14 hours - You can stop the training early if you wish. Then please collect the resulting `breakout_dqn/log.txt` and `breakout_a2c/log.txt` files into the same folder as this Jupyter notebook's. Rename them as `log_breakout_dqn.txt` and `log_breakout_a2c.txt`.

BTW, there's an Atari PC simulator: https://stella-emu.github.io/ I spent a lot of time playing them...

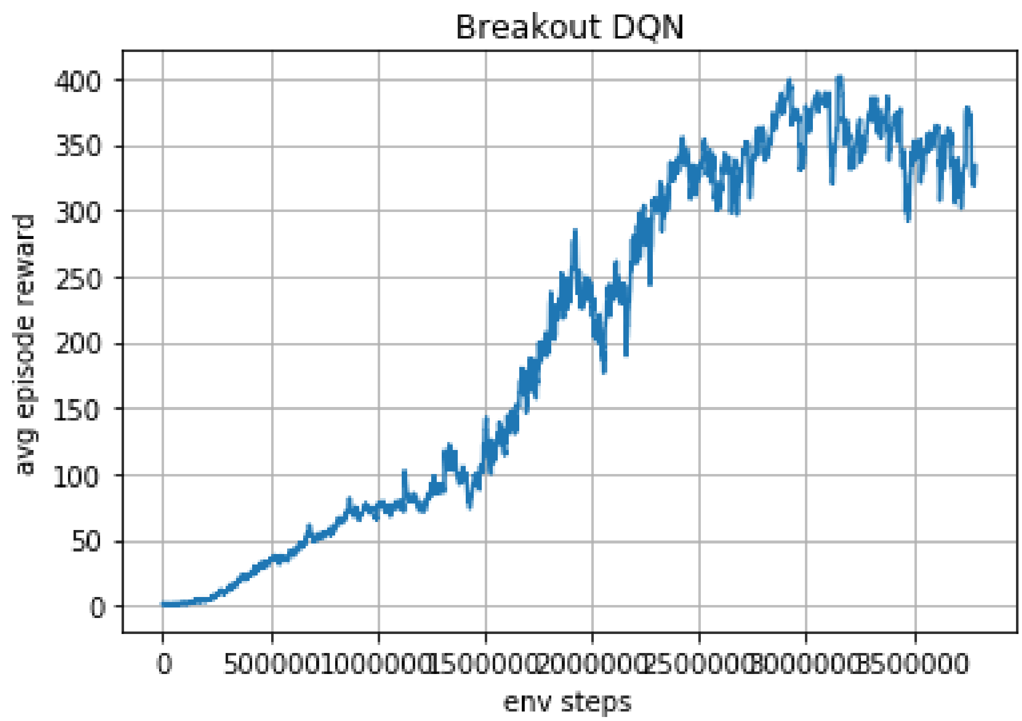
C5 (10 pts): Complete the code for the CNN with 3 conv layers and 3 fc layers in class `SimpleCNN` in file `Model.py` And verify the output shape with the cell below.

```
In [12]: ## Test code
         from Model import SimpleCNN
         import torch
         net = SimpleCNN()
         x = torch.randn(2, 4, 84, 84)
         y = net(x)
         assert y.shape == (2, 4), "ERROR: network output has the wrong shape!"
         print ("CNN output shape test passed!")
```

```
CNN output shape test passed!
```

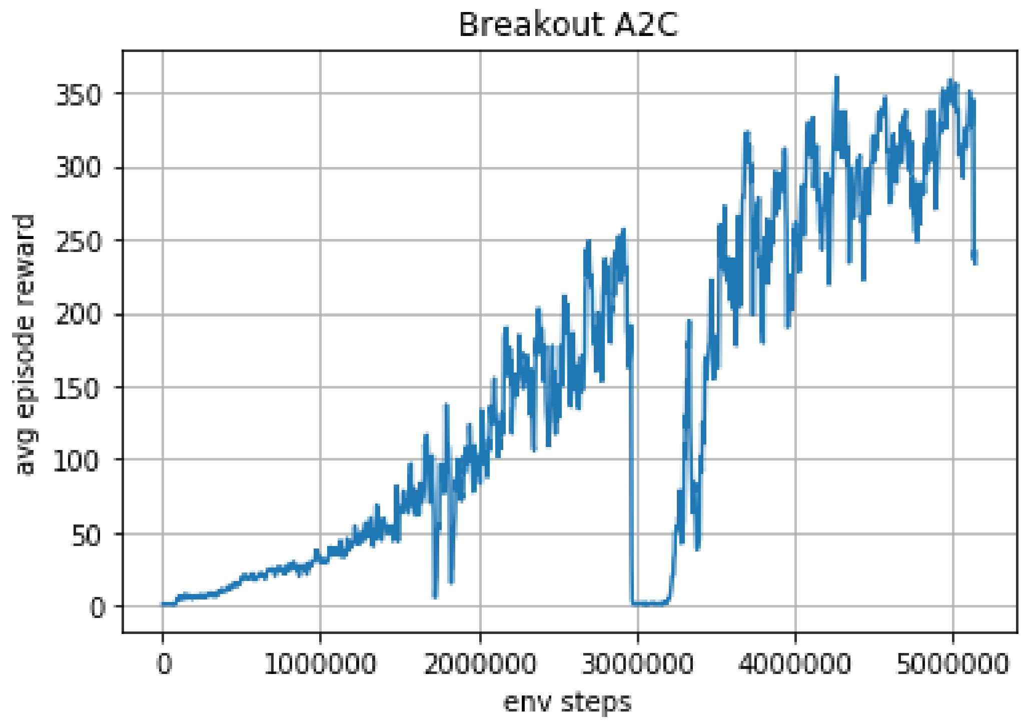
P3 (10 pts): Run the following cell to generate a DQN learning curve. The *maximum* average episodic reward on this curve should be larger than 200 for full credit. (It's ok if the final reward is not as high.) The typical value is around 300. You get 70% credit if $100 \leq$ average episodic reward < 200, 50% credit if $50 \leq$ average episodic reward < 100.

```
In [13]: plot_curve('log_breakout_dqn.txt', 'Breakout DQN')
```



P4 (10 pts): Run the following cell to generate an A2C learning curve. The *maximum* average episodic reward on this curve should be larger than 150 for full credit. (It's ok if the final reward is not as high.) The typical value is around 250. You get 70% credit if 50 $\leq$ average episodic reward $<$ 150, and 50% credit if 20 $\leq$ average episodic reward $<$ 50.

```
In [14]: plot_curve('log_breakout_a2c.txt', 'Breakout A2C')
```



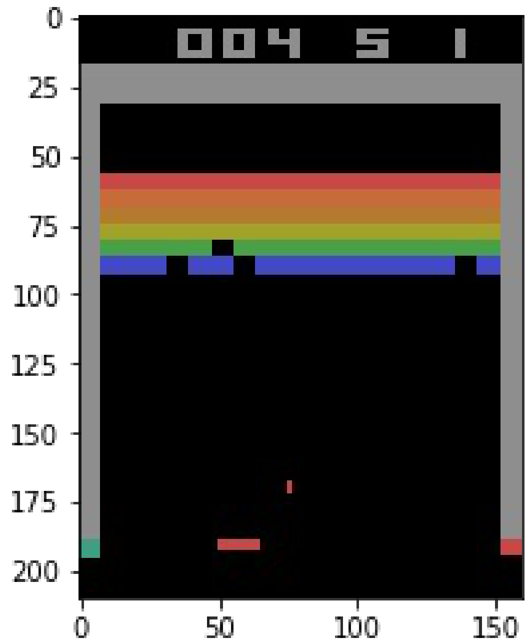
P5 (10 pts): Collect and visualize some game frames by running the script `Draw.py` on BlueWaters.

(1) `module load python/2.0.0` and run `Draw.py` on BlueWaters (it's ok to run this locally, no need to start a job).

(2) Download the result `breakout_imgs` folder from BlueWaters to the folder containing this Jupyter notebook, and run the following cell. You should see some animation of the game.

```
In [15]: import os
         imgs = sorted(os.listdir('breakout_imgs'))
         #imgs = [plt.imread('breakout_imgs/' + img) for img in imgs]

         %matplotlib inline
         import matplotlib.pyplot as plt
         from IPython import display
         pimg = None
         for img in imgs:
             img = plt.imread('breakout_imgs/' + img)
             if pimg:
                 pimg.set_data(img)
             else:
                 pimg = plt.imshow(img)
             display.display(plt.gcf())
             display.clear_output(wait=True)
```



1.3 Part III: Questions (10 pts)

These are open-ended questions. The purpose is to encourage you to think (a bit) more deeply about these algorithms. You get full points as long as you write a few sentences that make sense and show some thinking.

Q1 (2 pts): Why would people want to do function approximation rather than using tabular algorithm (on discretized S,A spaces if necessary)? Bringing function approximation has caused numerous problems theoretically (e.g. not guaranteed to converge), so it seems not worth it... Your answer: Deep Q learning as function approximator deals efficiently with curse of dimensionality. Using CNN as components of RL, we can directly learn from raw, high-dimensional visual inputs.

Q2 (2 pts): Q-Learning seems good... it's theoretically sound (at least seems to be), the performance is also good. Why would many people actually prefer policy gradient type algorithms in some practical problems? Your answer: Sometimes, Q function is too complex to be learned. Policy gradient would be still capable since it directly operates in the policy space. Also, Policy gradient usually converges faster and it may also learn the stochastic policies. Moreover, since the policy network is designed to model probability distribution, it is easy to apply to model continuous action space.

Q3 (2 pts): Does the policy gradient algorithm (A2C) we implemented here extend to continuous action space? How would you do that? Hint: What is a reasonable distribution assumption for policy $\pi_\theta(a|s)$ if a lives in continuous space? Your answer: Our implementation does not extend to continous action space. To do that, instead of let the network output the parameters for a categorical distribution, we may output the parameters for a Gaussion distribution with mean and std.

Q4 (2 pts): The policy gradient algorithm (A2C) we implemented uses on-policy data. Can you think of a way to extend it to utilize off-policy data? Hint: Importance sampling, needs some approximation though Your answer: Utilizing off-policy data means that we can use any actions to improve your value/action-value functions instead of only using the actions generated by the policy. However, we would get a lot of samples that are not part of distribution that we are interested in, which causes high variances. To filter out those unrelated samples, we can use importance sampling with approximation to determine how important the samples generated are to samples that the target policy may have made.

Q5 (2 pts): How to compare different RL algorithms? When can I say one algorithm is better than the other? Hint: This question is quite open. Think about speed, complexity, tasks, etc. Your answer: - Considering sample efficiency from less to more efficient: on-policy policy gradient algorithms -> actor-critic style methods -> off-policy Q-function learning -> model-based.

- Value function fitting(Q-learning, DQN) may minimize error of fit at best and doesnt optimize anything at worst. The complexity could be verty high. Q-learning uses fixed point iteration that may not converge.

- Model-based RL guarantees to converges and minimizes error of fit but does not guarantee that better model is better policy. It may not optimize for expected reward.

- Policy gradient is the only one that actually performs gradient descent on the true objective. But it is also often the least efficient.