Reinforcement Learning

November 12, 2019

1 IE 534 HW: Reinforcement Learning

`v1, Designed by Yuanyi Zhong, 2019`

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.,
(2) Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavu

(3) A useful tutorial: https://spinningup.openai.com/en/latest/
(4) *Useful code references*: https://github.com/deepmind/bsuite; https://github.com/openai/base
```

First of all, **enter your NetID here** in the cell below: Your NetID: yuanyiz2

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
```

```
Requirement already satisfied: gym in c:\users\bill\anaconda3\lib\site-packages (0.14.0)
Requirement already satisfied: scipy in c:\users\bill\anaconda3\lib\site-packages (from gym) (1.
Requirement already satisfied: cloudpickle~=1.2.0 in c:\users\bill\anaconda3\lib\site-packages (
Requirement already satisfied: pyglet<=1.3.2,>=1.2.0 in c:\users\bill\anaconda3\lib\site-package
Requirement already satisfied: numpy>=1.10.4 in c:\users\bill\anaconda3\lib\site-packages (from
Requirement already satisfied: six in c:\users\bill\anaconda3\lib\site-packages (from gym) (1.12
Requirement already satisfied: future in c:\users\bill\anaconda3\lib\site-packages (from pyglet<
Note: you may need to restart the kernel to use updated packages.
```

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 action
            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 .505

        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 g

        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 bu
        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
```

```python
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([1

0.28368130326271057 > 0.28000691533088684 ?
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 [7]: %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 cuda
parameters to optimize: [('fc1.weight', torch.Size([128, 4]), True), ('fc1.bias', torch.Size([12

obses on reset: 2 x (4,) float32
iter    200 |loss   0.01 |n_ep    16 |ep_len   21.6 |ep_rew  21.59 |raw_ep_rew  21.59 |env_step
iter    400 |loss   0.03 |n_ep    33 |ep_len   21.6 |ep_rew  21.57 |raw_ep_rew  21.57 |env_step
iter    600 |loss   0.01 |n_ep    47 |ep_len   25.8 |ep_rew  25.80 |raw_ep_rew  25.80 |env_step
iter    800 |loss   0.02 |n_ep    66 |ep_len   22.1 |ep_rew  22.06 |raw_ep_rew  22.06 |env_step
iter   1000 |loss   0.03 |n_ep    88 |ep_len   17.0 |ep_rew  17.01 |raw_ep_rew  17.01 |env_step
iter   1200 |loss   0.02 |n_ep   109 |ep_len   19.9 |ep_rew  19.91 |raw_ep_rew  19.91 |env_step
iter   1400 |loss   0.02 |n_ep   126 |ep_len   22.0 |ep_rew  21.97 |raw_ep_rew  21.97 |env_step
iter   1600 |loss   0.01 |n_ep   146 |ep_len   21.9 |ep_rew  21.93 |raw_ep_rew  21.93 |env_step
iter   1800 |loss   0.02 |n_ep   168 |ep_len   17.0 |ep_rew  16.98 |raw_ep_rew  16.98 |env_step
iter   2000 |loss   0.01 |n_ep   185 |ep_len   24.6 |ep_rew  24.59 |raw_ep_rew  24.59 |env_step
iter   2200 |loss   0.06 |n_ep   202 |ep_len   21.0 |ep_rew  21.05 |raw_ep_rew  21.05 |env_step
iter   2400 |loss   0.01 |n_ep   215 |ep_len   27.3 |ep_rew  27.29 |raw_ep_rew  27.29 |env_step
iter   2600 |loss   0.00 |n_ep   225 |ep_len   35.8 |ep_rew  35.84 |raw_ep_rew  35.84 |env_step
iter   2800 |loss   0.01 |n_ep   235 |ep_len   33.6 |ep_rew  33.55 |raw_ep_rew  33.55 |env_step
iter   3000 |loss   0.02 |n_ep   247 |ep_len   32.9 |ep_rew  32.94 |raw_ep_rew  32.94 |env_step
iter   3200 |loss   0.09 |n_ep   255 |ep_len   39.5 |ep_rew  39.48 |raw_ep_rew  39.48 |env_step
iter   3400 |loss   0.01 |n_ep   261 |ep_len   58.4 |ep_rew  58.35 |raw_ep_rew  58.35 |env_step
iter   3600 |loss   0.00 |n_ep   266 |ep_len   64.9 |ep_rew  64.94 |raw_ep_rew  64.94 |env_step
iter   3800 |loss   0.08 |n_ep   267 |ep_len   72.6 |ep_rew  72.65 |raw_ep_rew  72.65 |env_step
iter   4000 |loss   0.01 |n_ep   270 |ep_len  103.2 |ep_rew 103.16 |raw_ep_rew 103.16 |env_step
iter   4200 |loss   0.04 |n_ep   273 |ep_len  111.4 |ep_rew 111.41 |raw_ep_rew 111.41 |env_step
iter   4400 |loss   0.02 |n_ep   277 |ep_len  112.7 |ep_rew 112.71 |raw_ep_rew 112.71 |env_step
iter   4600 |loss   0.00 |n_ep   281 |ep_len  112.4 |ep_rew 112.41 |raw_ep_rew 112.41 |env_step
iter   4800 |loss   0.00 |n_ep   283 |ep_len  114.2 |ep_rew 114.22 |raw_ep_rew 114.22 |env_step
iter   5000 |loss   0.08 |n_ep   287 |ep_len  123.7 |ep_rew 123.75 |raw_ep_rew 123.75 |env_step
iter   5200 |loss   0.06 |n_ep   289 |ep_len  124.8 |ep_rew 124.79 |raw_ep_rew 124.79 |env_step
iter   5400 |loss   0.06 |n_ep   291 |ep_len  135.8 |ep_rew 135.84 |raw_ep_rew 135.84 |env_step
iter   5600 |loss   0.08 |n_ep   294 |ep_len  140.3 |ep_rew 140.32 |raw_ep_rew 140.32 |env_step
iter   5800 |loss   0.02 |n_ep   295 |ep_len  144.9 |ep_rew 144.89 |raw_ep_rew 144.89 |env_step
iter   6000 |loss   0.03 |n_ep   299 |ep_len  154.5 |ep_rew 154.54 |raw_ep_rew 154.54 |env_step
iter   6200 |loss   0.00 |n_ep   301 |ep_len  158.5 |ep_rew 158.51 |raw_ep_rew 158.51 |env_step
iter   6400 |loss   0.00 |n_ep   303 |ep_len  154.6 |ep_rew 154.60 |raw_ep_rew 154.60 |env_step
iter   6600 |loss   0.01 |n_ep   305 |ep_len  157.7 |ep_rew 157.68 |raw_ep_rew 157.68 |env_step
iter   6800 |loss   0.01 |n_ep   309 |ep_len  157.3 |ep_rew 157.26 |raw_ep_rew 157.26 |env_step
iter   7000 |loss   0.02 |n_ep   310 |ep_len  160.5 |ep_rew 160.54 |raw_ep_rew 160.54 |env_step
iter   7200 |loss   0.00 |n_ep   312 |ep_len  175.1 |ep_rew 175.12 |raw_ep_rew 175.12 |env_step
iter   7400 |loss   0.02 |n_ep   314 |ep_len  170.3 |ep_rew 170.32 |raw_ep_rew 170.32 |env_step
```

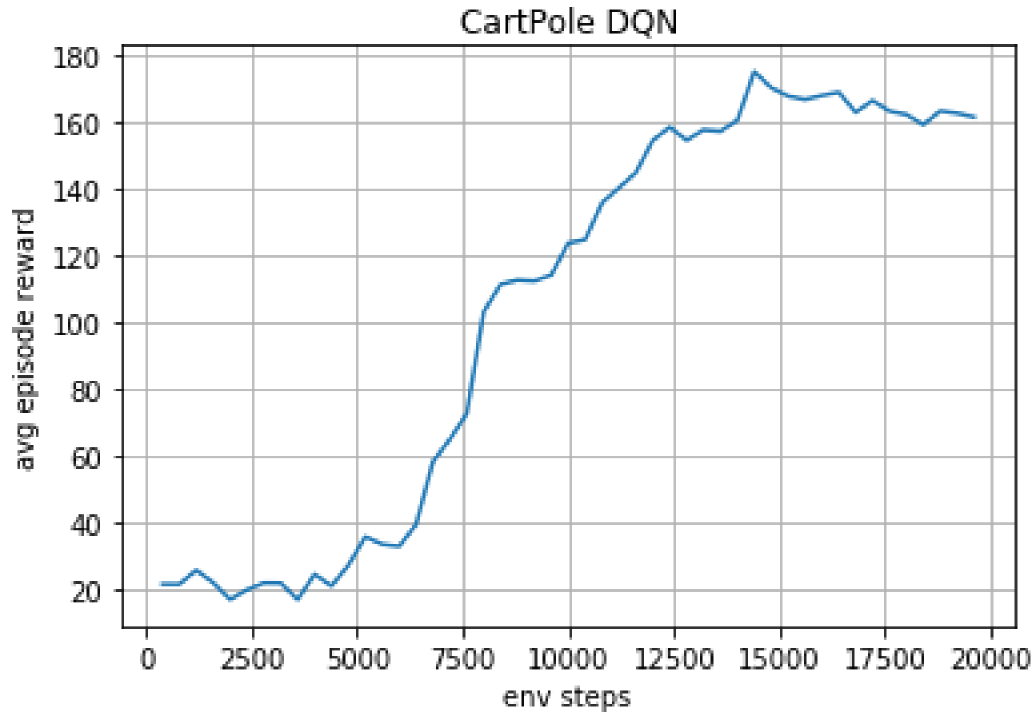
```
iter   7600 |loss   0.01 |n_ep   317 |ep_len  167.8 |ep_rew 167.82 |raw_ep_rew 167.82 |env_step
iter   7800 |loss   0.04 |n_ep   320 |ep_len  166.8 |ep_rew 166.80 |raw_ep_rew 166.80 |env_step
iter   8000 |loss   0.02 |n_ep   322 |ep_len  168.0 |ep_rew 167.99 |raw_ep_rew 167.99 |env_step
iter   8200 |loss   0.09 |n_ep   324 |ep_len  169.0 |ep_rew 168.97 |raw_ep_rew 168.97 |env_step
iter   8400 |loss   0.01 |n_ep   327 |ep_len  163.0 |ep_rew 162.95 |raw_ep_rew 162.95 |env_step
iter   8600 |loss   0.08 |n_ep   330 |ep_len  166.5 |ep_rew 166.48 |raw_ep_rew 166.48 |env_step
iter   8800 |loss   0.23 |n_ep   332 |ep_len  163.3 |ep_rew 163.31 |raw_ep_rew 163.31 |env_step
iter   9000 |loss   0.15 |n_ep   334 |ep_len  162.3 |ep_rew 162.29 |raw_ep_rew 162.29 |env_step
iter   9200 |loss   0.12 |n_ep   337 |ep_len  159.2 |ep_rew 159.23 |raw_ep_rew 159.23 |env_step
iter   9400 |loss   0.16 |n_ep   339 |ep_len  163.4 |ep_rew 163.37 |raw_ep_rew 163.37 |env_step
iter   9600 |loss   0.01 |n_ep   342 |ep_len  162.7 |ep_rew 162.68 |raw_ep_rew 162.68 |env_step
iter   9800 |loss   0.18 |n_ep   344 |ep_len  161.5 |ep_rew 161.54 |raw_ep_rew 161.54 |env_step
save checkpoint to cartpole_dqn/9999.pth
```

```
In [1]: 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 [9]: 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.

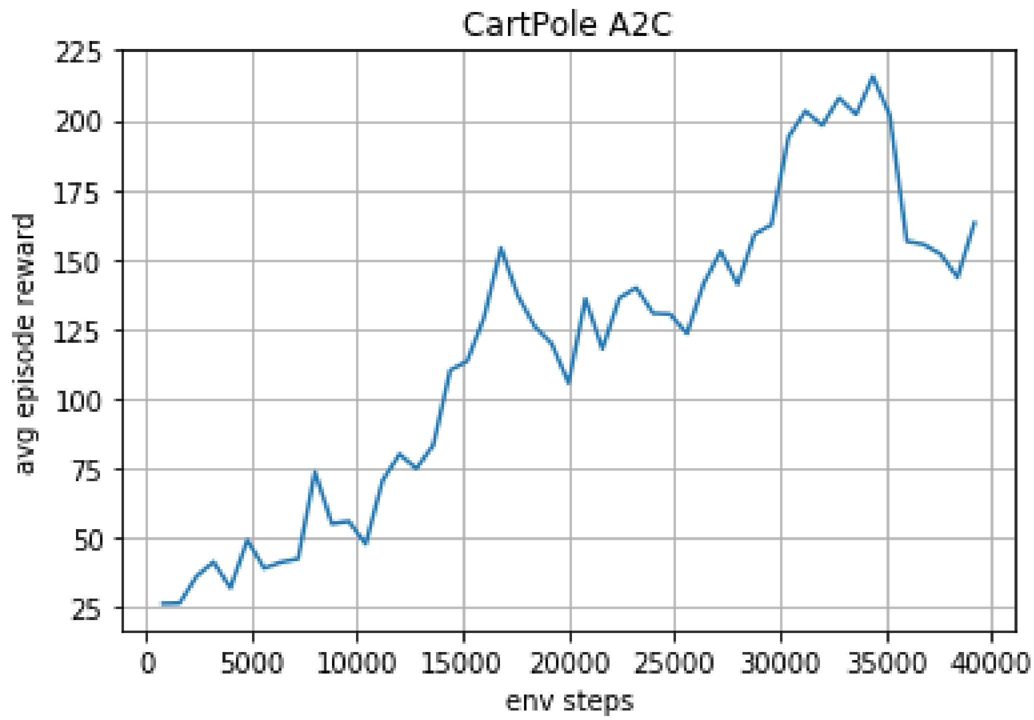
```
In [10]: %run Main.py  \
             --niter 10000   \
             --env CartPole-v1   \
             --algo a2c  \
             --nproc 4   \
             --lr 0.001  \
             --train_freq 16 \
             --train_start 0 \
             --batch_size 64     \
             --discount 0.996    \
             --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 cuda
shared net = False, parameters to optimize: [('fc1.weight', torch.Size([128, 4]), True), ('fc1.b

obses on reset: 4 x (4,) float32
iter    200 |loss   0.80 |n_ep    25 |ep_len   26.5 |ep_rew  26.49 |raw_ep_rew  26.49 |env_step
iter    400 |loss   0.87 |n_ep    54 |ep_len   26.7 |ep_rew  26.68 |raw_ep_rew  26.68 |env_step
iter    600 |loss   1.03 |n_ep    76 |ep_len   36.4 |ep_rew  36.36 |raw_ep_rew  36.36 |env_step
iter    800 |loss   0.72 |n_ep    98 |ep_len   41.5 |ep_rew  41.48 |raw_ep_rew  41.48 |env_step
iter   1000 |loss   0.85 |n_ep   122 |ep_len   32.2 |ep_rew  32.18 |raw_ep_rew  32.18 |env_step
iter   1200 |loss   0.82 |n_ep   137 |ep_len   49.3 |ep_rew  49.29 |raw_ep_rew  49.29 |env_step
iter   1400 |loss   0.75 |n_ep   156 |ep_len   39.2 |ep_rew  39.23 |raw_ep_rew  39.23 |env_step
iter   1600 |loss   0.86 |n_ep   173 |ep_len   41.4 |ep_rew  41.40 |raw_ep_rew  41.40 |env_step
iter   1800 |loss   0.95 |n_ep   191 |ep_len   42.6 |ep_rew  42.61 |raw_ep_rew  42.61 |env_step
iter   2000 |loss   0.51 |n_ep   201 |ep_len   73.8 |ep_rew  73.81 |raw_ep_rew  73.81 |env_step
iter   2200 |loss   0.66 |n_ep   217 |ep_len   55.2 |ep_rew  55.19 |raw_ep_rew  55.19 |env_step
iter   2400 |loss   0.81 |n_ep   232 |ep_len   56.1 |ep_rew  56.10 |raw_ep_rew  56.10 |env_step
iter   2600 |loss   0.92 |n_ep   243 |ep_len   47.9 |ep_rew  47.86 |raw_ep_rew  47.86 |env_step
iter   2800 |loss   1.03 |n_ep   254 |ep_len   70.7 |ep_rew  70.71 |raw_ep_rew  70.71 |env_step
iter   3000 |loss   1.00 |n_ep   262 |ep_len   80.1 |ep_rew  80.12 |raw_ep_rew  80.12 |env_step
iter   3200 |loss   0.57 |n_ep   275 |ep_len   75.1 |ep_rew  75.05 |raw_ep_rew  75.05 |env_step
iter   3400 |loss   1.00 |n_ep   282 |ep_len   83.4 |ep_rew  83.38 |raw_ep_rew  83.38 |env_step
iter   3600 |loss   0.96 |n_ep   288 |ep_len  110.2 |ep_rew 110.25 |raw_ep_rew 110.25 |env_step
iter   3800 |loss   0.67 |n_ep   294 |ep_len  113.5 |ep_rew 113.49 |raw_ep_rew 113.49 |env_step
```

```
iter   4000 |loss   0.83 |n_ep   297 |ep_len  129.2 |ep_rew 129.21 |raw_ep_rew 129.21 |env_step
iter   4200 |loss   0.97 |n_ep   304 |ep_len  154.1 |ep_rew 154.09 |raw_ep_rew 154.09 |env_step
iter   4400 |loss   0.28 |n_ep   311 |ep_len  137.0 |ep_rew 137.03 |raw_ep_rew 137.03 |env_step
iter   4600 |loss   0.92 |n_ep   317 |ep_len  125.9 |ep_rew 125.93 |raw_ep_rew 125.93 |env_step
iter   4800 |loss   0.86 |n_ep   325 |ep_len  119.8 |ep_rew 119.81 |raw_ep_rew 119.81 |env_step
iter   5000 |loss   0.83 |n_ep   332 |ep_len  105.7 |ep_rew 105.68 |raw_ep_rew 105.68 |env_step
iter   5200 |loss   0.29 |n_ep   337 |ep_len  135.9 |ep_rew 135.90 |raw_ep_rew 135.90 |env_step
iter   5400 |loss   0.33 |n_ep   345 |ep_len  117.9 |ep_rew 117.90 |raw_ep_rew 117.90 |env_step
iter   5600 |loss   1.00 |n_ep   350 |ep_len  136.1 |ep_rew 136.14 |raw_ep_rew 136.14 |env_step
iter   5800 |loss   1.00 |n_ep   355 |ep_len  139.9 |ep_rew 139.90 |raw_ep_rew 139.90 |env_step
iter   6000 |loss   0.18 |n_ep   361 |ep_len  130.7 |ep_rew 130.68 |raw_ep_rew 130.68 |env_step
iter   6200 |loss   0.72 |n_ep   368 |ep_len  130.4 |ep_rew 130.41 |raw_ep_rew 130.41 |env_step
iter   6400 |loss   0.49 |n_ep   375 |ep_len  123.4 |ep_rew 123.42 |raw_ep_rew 123.42 |env_step
iter   6600 |loss   0.86 |n_ep   379 |ep_len  141.5 |ep_rew 141.55 |raw_ep_rew 141.55 |env_step
iter   6800 |loss   0.82 |n_ep   383 |ep_len  152.9 |ep_rew 152.87 |raw_ep_rew 152.87 |env_step
iter   7000 |loss   0.31 |n_ep   389 |ep_len  141.1 |ep_rew 141.10 |raw_ep_rew 141.10 |env_step
iter   7200 |loss   0.70 |n_ep   393 |ep_len  159.0 |ep_rew 159.01 |raw_ep_rew 159.01 |env_step
iter   7400 |loss   1.11 |n_ep   397 |ep_len  162.5 |ep_rew 162.48 |raw_ep_rew 162.48 |env_step
iter   7600 |loss   0.16 |n_ep   401 |ep_len  193.9 |ep_rew 193.87 |raw_ep_rew 193.87 |env_step
iter   7800 |loss   0.66 |n_ep   404 |ep_len  203.2 |ep_rew 203.21 |raw_ep_rew 203.21 |env_step
iter   8000 |loss   0.28 |n_ep   409 |ep_len  198.1 |ep_rew 198.14 |raw_ep_rew 198.14 |env_step
iter   8200 |loss   0.99 |n_ep   410 |ep_len  207.9 |ep_rew 207.93 |raw_ep_rew 207.93 |env_step
iter   8400 |loss   0.87 |n_ep   416 |ep_len  202.1 |ep_rew 202.08 |raw_ep_rew 202.08 |env_step
iter   8600 |loss  -0.08 |n_ep   420 |ep_len  215.6 |ep_rew 215.61 |raw_ep_rew 215.61 |env_step
iter   8800 |loss   0.79 |n_ep   425 |ep_len  201.7 |ep_rew 201.66 |raw_ep_rew 201.66 |env_step
iter   9000 |loss   0.74 |n_ep   431 |ep_len  156.5 |ep_rew 156.51 |raw_ep_rew 156.51 |env_step
iter   9200 |loss   0.30 |n_ep   434 |ep_len  155.4 |ep_rew 155.39 |raw_ep_rew 155.39 |env_step
iter   9400 |loss   0.61 |n_ep   442 |ep_len  151.9 |ep_rew 151.88 |raw_ep_rew 151.88 |env_step
iter   9600 |loss   0.64 |n_ep   445 |ep_len  143.6 |ep_rew 143.59 |raw_ep_rew 143.59 |env_step
iter   9800 |loss   0.91 |n_ep   451 |ep_len  163.1 |ep_rew 163.09 |raw_ep_rew 163.09 |env_step
save checkpoint to cartpole_a2c/9999.pth
```

```
In [11]: 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 [13]: 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.b
```

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…

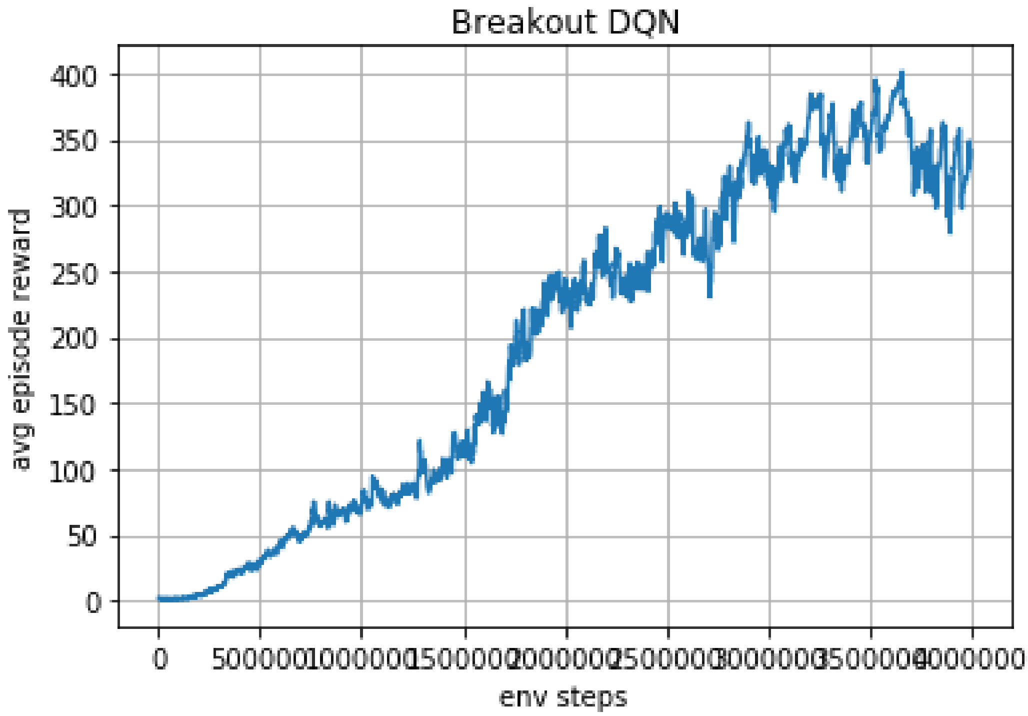
C5 (10 pts): Complete the code for the CNN with 3 conv layers and 3 fc layers in class `SimpleCNN` **in file** `Model.py` And verify the output shape with the cell below.

```
In [6]: ## 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!
```

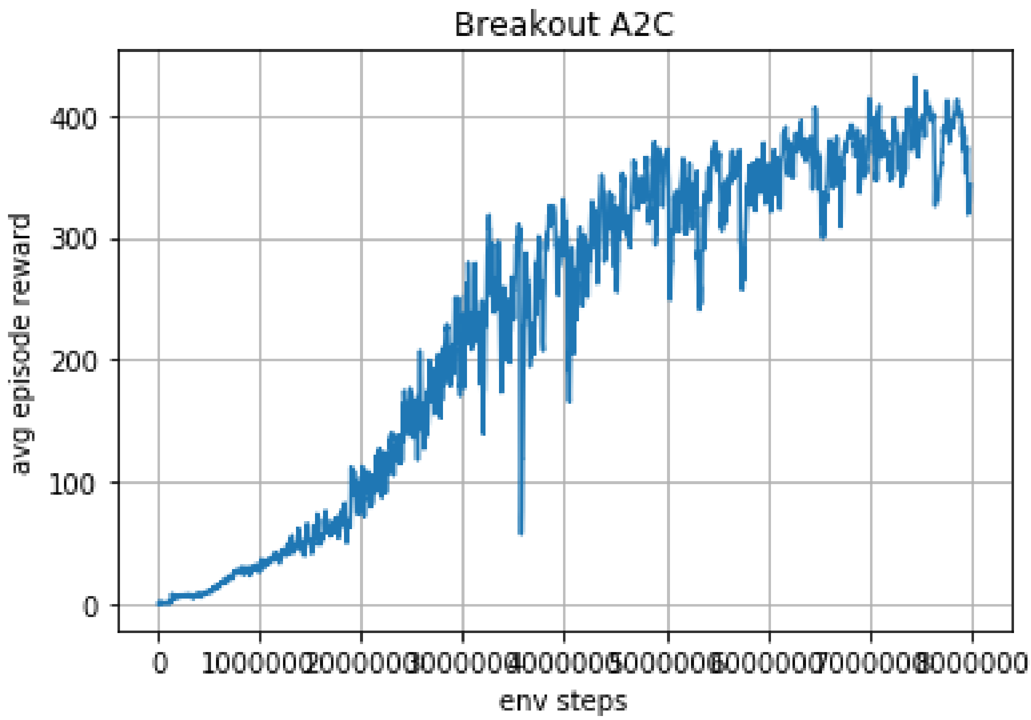
P3 (10 pts): Run the following cell to generate a DQN learning curve. The *maximum* average episodic reward on this curve should be larger than 200 for full credit. (It's ok if the final reward is not as high.) The typical value is around 300. You get 70% credit if $100 \leq$ average episodic reward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

```
In [14]: 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 [2]: plot_curve('log_breakout_a2c.txt', 'Breakout A2C')
```



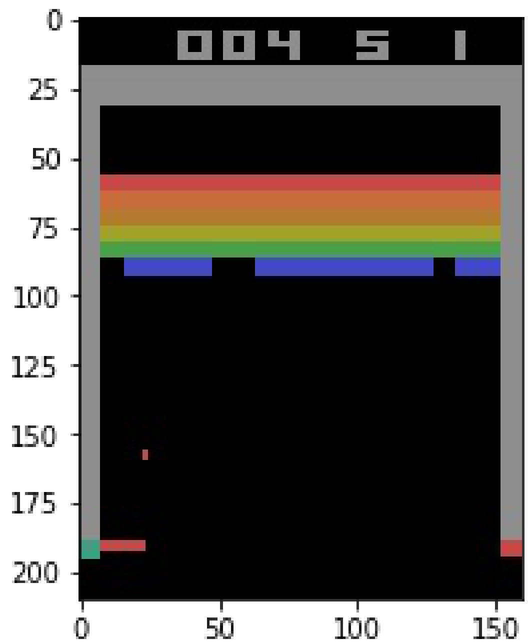
P5 (10 pts): Collect and visualize some game frames by running the script `Draw.py` on BlueWaters.

(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 [16]:
```
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: I don't know. People enjoy "neuralizing" things I guess..

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: I don't know. I like Q learning. The name is cute. Anyone watch StarTrek?

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: I don't know. Maybe normalizing flow?? OK, people really do this..(arXiv:1905.06893) Hot area + hot area

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: I don't know. Do random math tricks or pray?

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: I don't know. Just pick one you like, they're equally bad..