IE 534 HW: Reinforcement Learning

> v1, Designed for IE 534/CS 547 Deep Learning, 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/ (https://spinningup.openai.com/en/latest/)
- (4) *Useful code references*: https://github.com/deepmind/bsuite (https://github.com/deepmind/bsuite); https://github.com/openai/baselines (https://github.com/openai/baselines); https://github.com/astooke/rlpyt (https://github.com/astooke/rlpyt);

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

Your NetID: junzhew3

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/ (https://pytorch.org/get-started/locally/)](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:\programdata\anaconda3\lib\site-packages
(0.15.4)
Requirement already satisfied: scipy in c:\programdata\anaconda3\lib\site-packages
(from gym) (1.3.1)
Requirement already satisfied: six in c:\programdata\anaconda3\lib\site-packages
(from gym) (1.12.0)
Requirement already satisfied: pyglet<=1.3.2,>=1.2.0 in c:\programdata\anaconda3\l
ib\site-packages (from gym) (1.3.2)
Requirement already satisfied: opencv-python in c:\programdata\anaconda3\lib\site-
packages (from gym) (4.1.1.26)
Requirement already satisfied: numpy>=1.10.4 in c:\programdata\anaconda3\lib\site-
packages (from gym) (1.16.5)
Requirement already satisfied: cloudpickle~=1.2.0 in c:\programdata\anaconda3\lib
\site-packages (from gym) (1.2.2)
Requirement already satisfied: future in c:\programdata\anaconda3\lib\site-package
s (from pyglet<=1.3.2,>=1.2.0->gym) (0.17.1)
Note: you may need to restart the kernel to use updated packages.
```

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.

1.2 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:

0:00 / 0:02

In [3]:

```
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()
```

1.3 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 (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 [27]:

```
## 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 [28]:

```
## 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!" % eps

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 but got %f!" % eps

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 got %f!" % eps

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 but got %f!" % eps
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 [35]:

```
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([128]), True), ('fc2.weight', torch.Size([3, 128]), True), ('fc2.bias',
torch.Size([3]), True)]

0.2180587649345398 > 0.21361640095710754 ?
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 [36]:

```
%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_co
ef=0.01, env='CartPole-v1', eps_decay=4000, frame_skip=1, frame_stack=4, load='',
log='log.txt', lr=0.001, niter=10000, nproc=2, parallel_env=0, print_freq=200, rep
lay_size=20000, save_dir='cartpole_dqn/', target_update=1000, train_freq=1, train_
start=100, value_coef=0.5)
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([128]), True), ('fc2.weight', torch.Size([2, 128]), True), ('fc2.bias',
torch.Size([2]), True)]

obses on reset: 2 x (4,) float32
iter    200 |loss   0.01 |n_ep    16 |ep_len   20.9 |ep_rew  20.92 |raw_ep_rew  2
0.92 |env_step    400 |time 00:00 rem 00:10
iter    400 |loss   0.00 |n_ep    38 |ep_len   18.9 |ep_rew  18.91 |raw_ep_rew  1
8.91 |env_step    800 |time 00:00 rem 00:13
iter    600 |loss   0.01 |n_ep    57 |ep_len   20.4 |ep_rew  20.38 |raw_ep_rew  2
0.38 |env_step   1200 |time 00:00 rem 00:14
iter    800 |loss   0.00 |n_ep    77 |ep_len   19.1 |ep_rew  19.09 |raw_ep_rew  1
9.09 |env_step   1600 |time 00:01 rem 00:14
iter   1000 |loss   0.01 |n_ep    99 |ep_len   17.3 |ep_rew  17.30 |raw_ep_rew  1
7.30 |env_step   2000 |time 00:01 rem 00:14
iter   1200 |loss   0.02 |n_ep   119 |ep_len   19.6 |ep_rew  19.57 |raw_ep_rew  1
9.57 |env_step   2400 |time 00:01 rem 00:14
iter   1400 |loss   0.02 |n_ep   136 |ep_len   21.5 |ep_rew  21.51 |raw_ep_rew  2
1.51 |env_step   2800 |time 00:02 rem 00:14
iter   1600 |loss   0.03 |n_ep   163 |ep_len   15.2 |ep_rew  15.22 |raw_ep_rew  1
5.22 |env_step   3200 |time 00:02 rem 00:13
iter   1800 |loss   0.02 |n_ep   190 |ep_len   13.8 |ep_rew  13.83 |raw_ep_rew  1
3.83 |env_step   3600 |time 00:02 rem 00:13
iter   2000 |loss   0.10 |n_ep   217 |ep_len   14.2 |ep_rew  14.23 |raw_ep_rew  1
4.23 |env_step   4000 |time 00:03 rem 00:13
iter   2200 |loss   0.04 |n_ep   245 |ep_len   13.7 |ep_rew  13.73 |raw_ep_rew  1
3.73 |env_step   4400 |time 00:03 rem 00:12
iter   2400 |loss   0.11 |n_ep   275 |ep_len   13.0 |ep_rew  13.03 |raw_ep_rew  1
3.03 |env_step   4800 |time 00:03 rem 00:12
iter   2600 |loss   0.13 |n_ep   305 |ep_len   12.9 |ep_rew  12.94 |raw_ep_rew  1
2.94 |env_step   5200 |time 00:04 rem 00:12
iter   2800 |loss   0.09 |n_ep   333 |ep_len   14.3 |ep_rew  14.30 |raw_ep_rew  1
4.30 |env_step   5600 |time 00:04 rem 00:11
iter   3000 |loss   0.04 |n_ep   365 |ep_len   12.5 |ep_rew  12.48 |raw_ep_rew  1
2.48 |env_step   6000 |time 00:05 rem 00:11
iter   3200 |loss   0.07 |n_ep   393 |ep_len   14.2 |ep_rew  14.16 |raw_ep_rew  1
4.16 |env_step   6400 |time 00:05 rem 00:11
iter   3400 |loss   0.11 |n_ep   418 |ep_len   16.1 |ep_rew  16.13 |raw_ep_rew  1
6.13 |env_step   6800 |time 00:05 rem 00:11
iter   3600 |loss   0.08 |n_ep   439 |ep_len   19.4 |ep_rew  19.38 |raw_ep_rew  1
9.38 |env_step   7200 |time 00:06 rem 00:10
iter   3800 |loss   0.04 |n_ep   459 |ep_len   19.7 |ep_rew  19.69 |raw_ep_rew  1
9.69 |env_step   7600 |time 00:06 rem 00:10
iter   4000 |loss   0.09 |n_ep   472 |ep_len   27.8 |ep_rew  27.78 |raw_ep_rew  2
7.78 |env_step   8000 |time 00:06 rem 00:10
iter   4200 |loss   0.16 |n_ep   480 |ep_len   33.9 |ep_rew  33.90 |raw_ep_rew  3
3.90 |env_step   8400 |time 00:07 rem 00:09
iter   4400 |loss   0.28 |n_ep   483 |ep_len   50.2 |ep_rew  50.24 |raw_ep_rew  5
0.24 |env_step   8800 |time 00:07 rem 00:09
iter   4600 |loss   0.07 |n_ep   487 |ep_len   79.5 |ep_rew  79.45 |raw_ep_rew  7
9.45 |env_step   9200 |time 00:07 rem 00:09
iter   4800 |loss   0.06 |n_ep   490 |ep_len   89.4 |ep_rew  89.43 |raw_ep_rew  8
9.43 |env_step   9600 |time 00:08 rem 00:08
```

```
iter   5000 |loss   0.07 |n_ep   494 |ep_len  101.6 |ep_rew 101.61 |raw_ep_rew 10
1.61 |env_step  10000 |time 00:08 rem 00:08
iter   5200 |loss   0.71 |n_ep   498 |ep_len   86.4 |ep_rew  86.35 |raw_ep_rew  8
6.35 |env_step  10400 |time 00:08 rem 00:08
iter   5400 |loss   0.21 |n_ep   502 |ep_len  109.6 |ep_rew 109.57 |raw_ep_rew 10
9.57 |env_step  10800 |time 00:09 rem 00:07
iter   5600 |loss   0.07 |n_ep   504 |ep_len  122.6 |ep_rew 122.59 |raw_ep_rew 12
2.59 |env_step  11200 |time 00:09 rem 00:07
iter   5800 |loss   0.24 |n_ep   506 |ep_len  134.7 |ep_rew 134.74 |raw_ep_rew 13
4.74 |env_step  11600 |time 00:09 rem 00:07
iter   6000 |loss   0.07 |n_ep   508 |ep_len  143.5 |ep_rew 143.48 |raw_ep_rew 14
3.48 |env_step  12000 |time 00:10 rem 00:06
iter   6200 |loss   0.64 |n_ep   509 |ep_len  145.7 |ep_rew 145.73 |raw_ep_rew 14
5.73 |env_step  12400 |time 00:10 rem 00:06
iter   6400 |loss   0.07 |n_ep   511 |ep_len  169.8 |ep_rew 169.81 |raw_ep_rew 16
9.81 |env_step  12800 |time 00:10 rem 00:06
iter   6600 |loss   0.09 |n_ep   513 |ep_len  177.4 |ep_rew 177.44 |raw_ep_rew 17
7.44 |env_step  13200 |time 00:11 rem 00:05
iter   6800 |loss   0.07 |n_ep   515 |ep_len  183.3 |ep_rew 183.26 |raw_ep_rew 18
3.26 |env_step  13600 |time 00:11 rem 00:05
iter   7000 |loss   0.79 |n_ep   517 |ep_len  189.4 |ep_rew 189.43 |raw_ep_rew 18
9.43 |env_step  14000 |time 00:11 rem 00:05
iter   7200 |loss   1.45 |n_ep   519 |ep_len  196.1 |ep_rew 196.05 |raw_ep_rew 19
6.05 |env_step  14400 |time 00:12 rem 00:04
iter   7400 |loss   0.06 |n_ep   520 |ep_len  195.5 |ep_rew 195.55 |raw_ep_rew 19
5.55 |env_step  14800 |time 00:12 rem 00:04
iter   7600 |loss   0.12 |n_ep   522 |ep_len  208.8 |ep_rew 208.77 |raw_ep_rew 20
8.77 |env_step  15200 |time 00:12 rem 00:04
iter   7800 |loss   0.35 |n_ep   523 |ep_len  216.8 |ep_rew 216.80 |raw_ep_rew 21
6.80 |env_step  15600 |time 00:13 rem 00:03
iter   8000 |loss   0.82 |n_ep   525 |ep_len  228.1 |ep_rew 228.06 |raw_ep_rew 22
8.06 |env_step  16000 |time 00:13 rem 00:03
iter   8200 |loss   1.10 |n_ep   527 |ep_len  227.7 |ep_rew 227.70 |raw_ep_rew 22
7.70 |env_step  16400 |time 00:14 rem 00:03
iter   8400 |loss   0.20 |n_ep   527 |ep_len  227.7 |ep_rew 227.70 |raw_ep_rew 22
7.70 |env_step  16800 |time 00:14 rem 00:02
iter   8600 |loss   0.04 |n_ep   529 |ep_len  237.6 |ep_rew 237.58 |raw_ep_rew 23
7.58 |env_step  17200 |time 00:14 rem 00:02
iter   8800 |loss   0.37 |n_ep   531 |ep_len  247.8 |ep_rew 247.82 |raw_ep_rew 24
7.82 |env_step  17600 |time 00:15 rem 00:02
iter   9000 |loss   0.20 |n_ep   532 |ep_len  248.9 |ep_rew 248.94 |raw_ep_rew 24
8.94 |env_step  18000 |time 00:15 rem 00:01
iter   9200 |loss   0.84 |n_ep   533 |ep_len  248.4 |ep_rew 248.44 |raw_ep_rew 24
8.44 |env_step  18400 |time 00:15 rem 00:01
iter   9400 |loss   0.16 |n_ep   535 |ep_len  247.5 |ep_rew 247.52 |raw_ep_rew 24
7.52 |env_step  18800 |time 00:16 rem 00:01
iter   9600 |loss   0.07 |n_ep   537 |ep_len  249.5 |ep_rew 249.53 |raw_ep_rew 24
9.53 |env_step  19200 |time 00:16 rem 00:00
iter   9800 |loss   0.22 |n_ep   539 |ep_len  250.2 |ep_rew 250.22 |raw_ep_rew 25
0.22 |env_step  19600 |time 00:16 rem 00:00
save checkpoint to cartpole_dqn/9999.pth
```

In [2]:

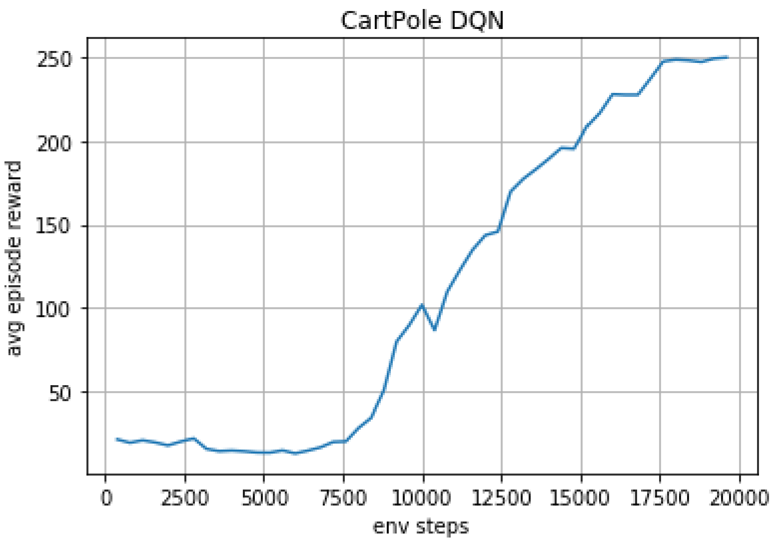
```
import matplotlib.pyplot as plt

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 [38]:

```
plot_curve('cartpole_dqn/log.txt', 'CartPole DQN')
```



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 (https://arxiv.org/abs/1506.02438)](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:
$$L_{policy}(\theta) = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=0}^{T}\left(\log \pi_\theta(s_t^i, a_t^i)\left(\sum_{t'=t}^{T}\gamma^{t'-t} r_{t'}^i - V_\phi(s_t^i)\right)\right)$$

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
```

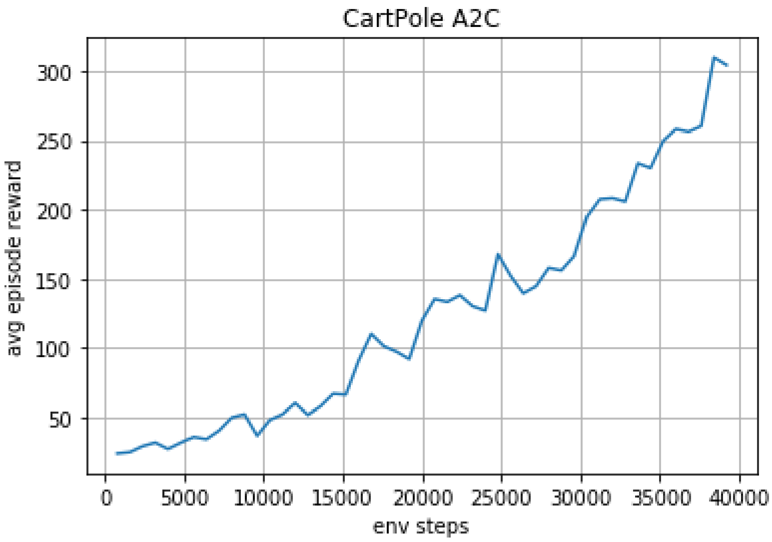
```
Namespace(algo='a2c', batch_size=64, checkpoint_freq=20000, discount=0.996, ent_co
ef=0.01, env='CartPole-v1', eps_decay=200000, frame_skip=1, frame_stack=4, load
='', log='log.txt', lr=0.001, niter=10000, nproc=4, parallel_env=0, print_freq=20
0, replay_size=1000000, save_dir='cartpole_a2c/', target_update=2500, train_freq=1
6, train_start=0, value_coef=0.01)
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.bias', torch.Size([128]), True), ('fc2.weight', torch.Size([2, 128]),
True), ('fc2.bias', torch.Size([2]), True), ('fc1.weight', torch.Size([128, 4]), T
rue), ('fc1.bias', torch.Size([128]), True), ('fc2.weight', torch.Size([1, 128]),
True), ('fc2.bias', torch.Size([1]), True)]

obses on reset: 4 x (4,) float32
iter    200 |loss   0.91 |n_ep    35 |ep_len   23.8 |ep_rew  23.77 |raw_ep_rew  2
3.77 |env_step    800 |time 00:00 rem 00:08
iter    400 |loss   0.85 |n_ep    64 |ep_len   24.7 |ep_rew  24.71 |raw_ep_rew  2
4.71 |env_step   1600 |time 00:00 rem 00:07
iter    600 |loss   0.87 |n_ep    93 |ep_len   29.0 |ep_rew  28.97 |raw_ep_rew  2
8.97 |env_step   2400 |time 00:00 rem 00:07
iter    800 |loss   0.75 |n_ep   121 |ep_len   31.6 |ep_rew  31.58 |raw_ep_rew  3
1.58 |env_step   3200 |time 00:00 rem 00:07
iter   1000 |loss   0.82 |n_ep   149 |ep_len   27.0 |ep_rew  26.97 |raw_ep_rew  2
6.97 |env_step   4000 |time 00:00 rem 00:07
iter   1200 |loss   0.89 |n_ep   172 |ep_len   31.4 |ep_rew  31.43 |raw_ep_rew  3
1.43 |env_step   4800 |time 00:00 rem 00:07
iter   1400 |loss   0.99 |n_ep   192 |ep_len   35.6 |ep_rew  35.58 |raw_ep_rew  3
5.58 |env_step   5600 |time 00:01 rem 00:06
iter   1600 |loss   0.78 |n_ep   216 |ep_len   34.0 |ep_rew  33.98 |raw_ep_rew  3
3.98 |env_step   6400 |time 00:01 rem 00:06
iter   1800 |loss   0.75 |n_ep   236 |ep_len   40.2 |ep_rew  40.22 |raw_ep_rew  4
0.22 |env_step   7200 |time 00:01 rem 00:06
iter   2000 |loss   0.68 |n_ep   253 |ep_len   49.6 |ep_rew  49.56 |raw_ep_rew  4
9.56 |env_step   8000 |time 00:01 rem 00:06
iter   2200 |loss   0.77 |n_ep   270 |ep_len   51.9 |ep_rew  51.91 |raw_ep_rew  5
1.91 |env_step   8800 |time 00:01 rem 00:06
iter   2400 |loss   0.61 |n_ep   289 |ep_len   36.5 |ep_rew  36.46 |raw_ep_rew  3
6.46 |env_step   9600 |time 00:01 rem 00:06
iter   2600 |loss   0.74 |n_ep   305 |ep_len   47.8 |ep_rew  47.81 |raw_ep_rew  4
7.81 |env_step  10400 |time 00:02 rem 00:05
iter   2800 |loss   0.98 |n_ep   316 |ep_len   51.9 |ep_rew  51.89 |raw_ep_rew  5
1.89 |env_step  11200 |time 00:02 rem 00:05
iter   3000 |loss   0.50 |n_ep   331 |ep_len   60.1 |ep_rew  60.08 |raw_ep_rew  6
0.08 |env_step  12000 |time 00:02 rem 00:05
iter   3200 |loss   0.41 |n_ep   349 |ep_len   51.4 |ep_rew  51.42 |raw_ep_rew  5
1.42 |env_step  12800 |time 00:02 rem 00:05
iter   3400 |loss   0.96 |n_ep   360 |ep_len   57.7 |ep_rew  57.75 |raw_ep_rew  5
7.75 |env_step  13600 |time 00:02 rem 00:05
iter   3600 |loss   0.51 |n_ep   371 |ep_len   66.8 |ep_rew  66.77 |raw_ep_rew  6
6.77 |env_step  14400 |time 00:02 rem 00:05
iter   3800 |loss   0.65 |n_ep   381 |ep_len   66.1 |ep_rew  66.08 |raw_ep_rew  6
6.08 |env_step  15200 |time 00:03 rem 00:04
iter   4000 |loss   0.96 |n_ep   388 |ep_len   90.6 |ep_rew  90.56 |raw_ep_rew  9
0.56 |env_step  16000 |time 00:03 rem 00:04
iter   4200 |loss   0.31 |n_ep   395 |ep_len  110.0 |ep_rew 110.00 |raw_ep_rew 11
0.00 |env_step  16800 |time 00:03 rem 00:04
iter   4400 |loss  -0.08 |n_ep   404 |ep_len  101.1 |ep_rew 101.06 |raw_ep_rew 10
1.06 |env_step  17600 |time 00:03 rem 00:04
iter   4600 |loss   0.46 |n_ep   413 |ep_len   97.1 |ep_rew  97.08 |raw_ep_rew  9
7.08 |env_step  18400 |time 00:03 rem 00:04
```

```
iter   4800 |loss   0.51 |n_ep   422 |ep_len   91.7 |ep_rew  91.74 |raw_ep_rew  9
1.74 |env_step  19200 |time 00:03 rem 00:04
iter   5000 |loss   0.62 |n_ep   426 |ep_len  119.6 |ep_rew 119.60 |raw_ep_rew 11
9.60 |env_step  20000 |time 00:03 rem 00:03
iter   5200 |loss   1.06 |n_ep   431 |ep_len  135.2 |ep_rew 135.18 |raw_ep_rew 13
5.18 |env_step  20800 |time 00:04 rem 00:03
iter   5400 |loss   0.57 |n_ep   439 |ep_len  133.1 |ep_rew 133.14 |raw_ep_rew 13
3.14 |env_step  21600 |time 00:04 rem 00:03
iter   5600 |loss   0.93 |n_ep   442 |ep_len  138.0 |ep_rew 137.98 |raw_ep_rew 13
7.98 |env_step  22400 |time 00:04 rem 00:03
iter   5800 |loss   0.28 |n_ep   450 |ep_len  130.0 |ep_rew 130.03 |raw_ep_rew 13
0.03 |env_step  23200 |time 00:04 rem 00:03
iter   6000 |loss   0.84 |n_ep   454 |ep_len  126.8 |ep_rew 126.84 |raw_ep_rew 12
6.84 |env_step  24000 |time 00:04 rem 00:03
iter   6200 |loss   0.24 |n_ep   459 |ep_len  167.7 |ep_rew 167.69 |raw_ep_rew 16
7.69 |env_step  24800 |time 00:04 rem 00:02
iter   6400 |loss   0.77 |n_ep   464 |ep_len  152.0 |ep_rew 151.98 |raw_ep_rew 15
1.98 |env_step  25600 |time 00:05 rem 00:02
iter   6600 |loss   0.70 |n_ep   471 |ep_len  139.3 |ep_rew 139.34 |raw_ep_rew 13
9.34 |env_step  26400 |time 00:05 rem 00:02
iter   6800 |loss   0.43 |n_ep   477 |ep_len  144.4 |ep_rew 144.43 |raw_ep_rew 14
4.43 |env_step  27200 |time 00:05 rem 00:02
iter   7000 |loss   0.08 |n_ep   482 |ep_len  157.7 |ep_rew 157.74 |raw_ep_rew 15
7.74 |env_step  28000 |time 00:05 rem 00:02
iter   7200 |loss   0.72 |n_ep   486 |ep_len  155.9 |ep_rew 155.90 |raw_ep_rew 15
5.90 |env_step  28800 |time 00:05 rem 00:02
iter   7400 |loss   0.75 |n_ep   490 |ep_len  166.2 |ep_rew 166.16 |raw_ep_rew 16
6.16 |env_step  29600 |time 00:05 rem 00:02
iter   7600 |loss   0.91 |n_ep   493 |ep_len  195.0 |ep_rew 195.04 |raw_ep_rew 19
5.04 |env_step  30400 |time 00:05 rem 00:01
iter   7800 |loss   0.94 |n_ep   496 |ep_len  207.5 |ep_rew 207.46 |raw_ep_rew 20
7.46 |env_step  31200 |time 00:06 rem 00:01
iter   8000 |loss   0.85 |n_ep   498 |ep_len  208.2 |ep_rew 208.23 |raw_ep_rew 20
8.23 |env_step  32000 |time 00:06 rem 00:01
iter   8200 |loss   0.65 |n_ep   503 |ep_len  205.8 |ep_rew 205.81 |raw_ep_rew 20
5.81 |env_step  32800 |time 00:06 rem 00:01
iter   8400 |loss   0.90 |n_ep   506 |ep_len  233.4 |ep_rew 233.42 |raw_ep_rew 23
3.42 |env_step  33600 |time 00:06 rem 00:01
iter   8600 |loss   0.74 |n_ep   509 |ep_len  230.1 |ep_rew 230.06 |raw_ep_rew 23
0.06 |env_step  34400 |time 00:06 rem 00:01
iter   8800 |loss   1.03 |n_ep   511 |ep_len  249.6 |ep_rew 249.56 |raw_ep_rew 24
9.56 |env_step  35200 |time 00:06 rem 00:00
iter   9000 |loss  -0.14 |n_ep   515 |ep_len  258.1 |ep_rew 258.07 |raw_ep_rew 25
8.07 |env_step  36000 |time 00:07 rem 00:00
iter   9200 |loss   1.02 |n_ep   518 |ep_len  256.1 |ep_rew 256.15 |raw_ep_rew 25
6.15 |env_step  36800 |time 00:07 rem 00:00
iter   9400 |loss   0.77 |n_ep   519 |ep_len  260.2 |ep_rew 260.23 |raw_ep_rew 26
0.23 |env_step  37600 |time 00:07 rem 00:00
iter   9600 |loss   0.82 |n_ep   522 |ep_len  309.8 |ep_rew 309.77 |raw_ep_rew 30
9.77 |env_step  38400 |time 00:07 rem 00:00
iter   9800 |loss  -0.16 |n_ep   524 |ep_len  304.3 |ep_rew 304.25 |raw_ep_rew 30
4.25 |env_step  39200 |time 00:07 rem 00:00
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 [5]:

```
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.bias', torch.Size([128]), True), ('fc2.weight', torch.Size([2, 128]),
True), ('fc2.bias', torch.Size([2]), True), ('fc1.weight', torch.Size([128, 4]), T
rue), ('fc1.bias', torch.Size([128]), True), ('fc2.weight', torch.Size([1, 128]),
True), ('fc2.bias', torch.Size([1]), True)]
```

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/ (https://stella-emu.github.io/)](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 [2]:

```
## 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!")
```

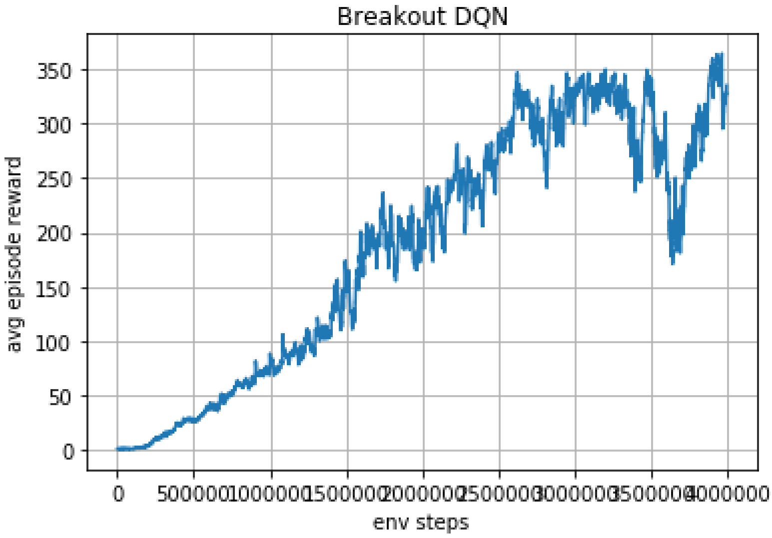
```
CNN output shape test passed!
```

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 [3]:

```
plot_curve('log_breakout_dqn.txt', 'Breakout DQN')
```

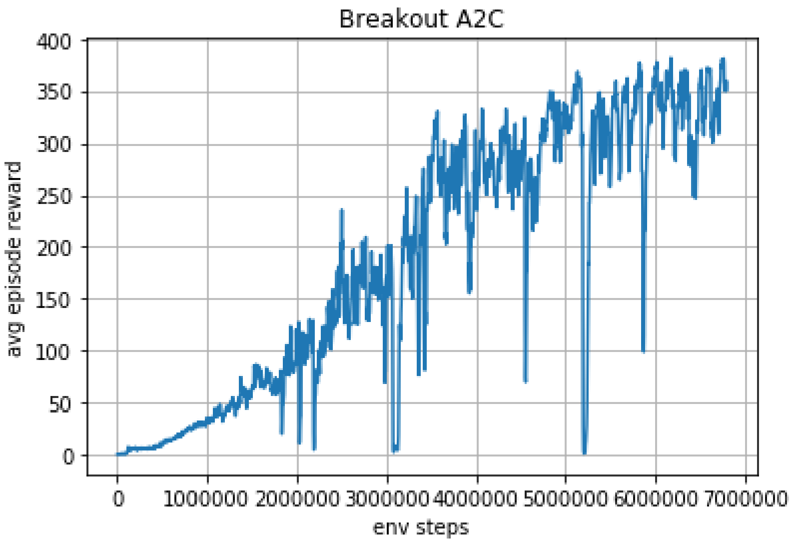


P4 (10 pts): Run the following cell to generate an A2C learning curve.

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 [4]:

```
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.

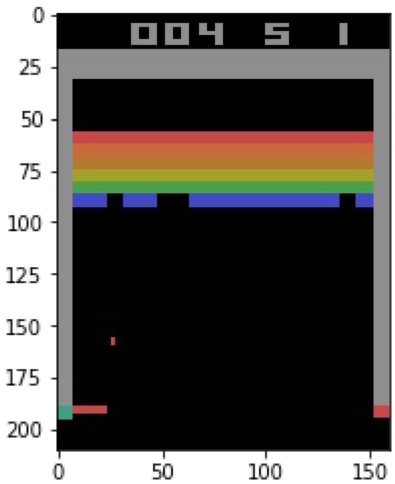
(1) `module load python/2.0.0` and run `Draw.py` on BlueWaters (it's ok to run this locally, no need to start a job).

(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 [6]:

```
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)
```



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: The tabular algorithm might works well in samll space like the 4X4 grid world. But when game space is larger, such as chess, the total number of discretized states will be very large if we take the moving direction, possibilty and the kings of chess into consideration. It is unrealistic to compute and store these large states.

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: If we use policy gradient type algorithms, we don't really need to know about the ergodic distribution of states P nor the environment dynamics p. This is crucial because for most practical purposes, it hard to model both these variables.On occasions when the Q function is too complex to be learned, DQN will fail miserably. On the other hand, Policy Gradients is still capable of learning a good policy since it directly operates in the policy space. Furthermore, Policy Gradients usually show faster convergence rate than DQN.

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: Yes, we can assume the values for actions are Gaussian distributed and the policy is defined using a Gaussian distribution with means computed from a deep network.

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: Yes, we can predefine behavior policy for collecting samples as a known policy and add the ratio of two policies called importance weight into the gradient. We can also use an approximated gradient with the gradient of Q ignored. In this way, we still guarantee the policy improvement and eventually achieve the true local minimum.

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: 1. It should be able to apply to a wider range of problems. For example, it can be easily applied to model continuous action space. 2. The faster convergence rate 3. The tendency to converge to a global optimal.