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);

Your NetID: sdutta26

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]:
```
conda install pytorch==1.1.0 torchvision==0.3.0 cudatoolkit=10.0 -c pytorch
```

```
Collecting package metadata (repodata.json): ...working... done
Solving environment: ...working... done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

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

```
Requirement already satisfied: gym in c:\users\sayan\anaconda3\lib\site-packages
(0.15.4)
Requirement already satisfied: scipy in c:\users\sayan\anaconda3\lib\site-packag
es (from gym) (1.1.0)
Requirement already satisfied: pyglet<=1.3.2,>=1.2.0 in c:\users\sayan\anaconda3
\lib\site-packages (from gym) (1.3.2)
Requirement already satisfied: opencv-python in c:\users\sayan\anaconda3\lib\sit
e-packages (from gym) (4.1.2.30)
Requirement already satisfied: cloudpickle~=1.2.0 in c:\users\sayan\anaconda3\li
b\site-packages (from gym) (1.2.2)
Requirement already satisfied: six in c:\users\sayan\anaconda3\lib\site-packages
(from gym) (1.12.0)
Requirement already satisfied: numpy>=1.10.4 in c:\users\sayan\anaconda3\lib\sit
e-packages (from gym) (1.14.5)
Requirement already satisfied: future in c:\users\sayan\anaconda3\lib\site-packa
ges (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:

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

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:

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

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

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

   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 [4]:
```
## 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 [5]:
```
## 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 go
t %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 arou
nd .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 .0
1 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 [6]:
```
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.bia
s', torch.Size([128]), True), ('fc2.weight', torch.Size([3, 128]), True), ('fc2.
bias', torch.Size([3]), True)]

0.14874370396137238 > 0.14560644328594208 ?
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='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=2
00, replay_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.bia
s', 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   0.02 |n_ep    16 |ep_len   25.8 |ep_rew  25.84 |raw_ep_rew
25.84 |env_step    400 |time 00:00 rem 00:11
iter    400 |loss   0.00 |n_ep    35 |ep_len   20.2 |ep_rew  20.23 |raw_ep_rew
20.23 |env_step    800 |time 00:00 rem 00:14
iter    600 |loss   0.01 |n_ep    51 |ep_len   18.8 |ep_rew  18.76 |raw_ep_rew
18.76 |env_step   1200 |time 00:01 rem 00:15
iter    800 |loss   0.00 |n_ep    65 |ep_len   28.8 |ep_rew  28.76 |raw_ep_rew
28.76 |env_step   1600 |time 00:01 rem 00:16
iter   1000 |loss   0.00 |n_ep    76 |ep_len   31.1 |ep_rew  31.13 |raw_ep_rew
31.13 |env_step   2000 |time 00:01 rem 00:15
iter   1200 |loss   0.01 |n_ep    87 |ep_len   33.9 |ep_rew  33.93 |raw_ep_rew
33.93 |env_step   2400 |time 00:02 rem 00:15
iter   1400 |loss   0.01 |n_ep   100 |ep_len   34.3 |ep_rew  34.34 |raw_ep_rew
34.34 |env_step   2800 |time 00:02 rem 00:15
iter   1600 |loss   0.02 |n_ep   112 |ep_len   33.1 |ep_rew  33.14 |raw_ep_rew
33.14 |env_step   3200 |time 00:02 rem 00:15
iter   1800 |loss   0.02 |n_ep   126 |ep_len   27.3 |ep_rew  27.30 |raw_ep_rew
27.30 |env_step   3600 |time 00:03 rem 00:15
iter   2000 |loss   0.01 |n_ep   136 |ep_len   34.7 |ep_rew  34.70 |raw_ep_rew
34.70 |env_step   4000 |time 00:03 rem 00:14
iter   2200 |loss   0.03 |n_ep   145 |ep_len   39.4 |ep_rew  39.42 |raw_ep_rew
39.42 |env_step   4400 |time 00:04 rem 00:14
iter   2400 |loss   0.02 |n_ep   148 |ep_len   61.7 |ep_rew  61.69 |raw_ep_rew
61.69 |env_step   4800 |time 00:04 rem 00:13
iter   2600 |loss   0.02 |n_ep   151 |ep_len   71.9 |ep_rew  71.94 |raw_ep_rew
71.94 |env_step   5200 |time 00:04 rem 00:13
iter   2800 |loss   0.03 |n_ep   159 |ep_len   65.2 |ep_rew  65.21 |raw_ep_rew
65.21 |env_step   5600 |time 00:05 rem 00:13
iter   3000 |loss   0.02 |n_ep   164 |ep_len   67.8 |ep_rew  67.76 |raw_ep_rew
67.76 |env_step   6000 |time 00:05 rem 00:13
iter   3200 |loss   0.00 |n_ep   168 |ep_len   84.9 |ep_rew  84.95 |raw_ep_rew
84.95 |env_step   6400 |time 00:05 rem 00:12
iter   3400 |loss   0.01 |n_ep   170 |ep_len   94.5 |ep_rew  94.46 |raw_ep_rew
94.46 |env_step   6800 |time 00:06 rem 00:12
iter   3600 |loss   0.03 |n_ep   172 |ep_len  114.2 |ep_rew 114.18 |raw_ep_rew 1
14.18 |env_step   7200 |time 00:06 rem 00:12
iter   3800 |loss   0.02 |n_ep   173 |ep_len  128.6 |ep_rew 128.56 |raw_ep_rew 1
28.56 |env_step   7600 |time 00:07 rem 00:12
iter   4000 |loss   0.01 |n_ep   174 |ep_len  143.2 |ep_rew 143.21 |raw_ep_rew 1
43.21 |env_step   8000 |time 00:07 rem 00:11
iter   4200 |loss   0.02 |n_ep   176 |ep_len  169.8 |ep_rew 169.85 |raw_ep_rew 1
69.85 |env_step   8400 |time 00:08 rem 00:11
iter   4400 |loss   0.05 |n_ep   178 |ep_len  189.6 |ep_rew 189.63 |raw_ep_rew 1
89.63 |env_step   8800 |time 00:08 rem 00:11
iter   4600 |loss   0.00 |n_ep   179 |ep_len  194.0 |ep_rew 193.96 |raw_ep_rew 1
```

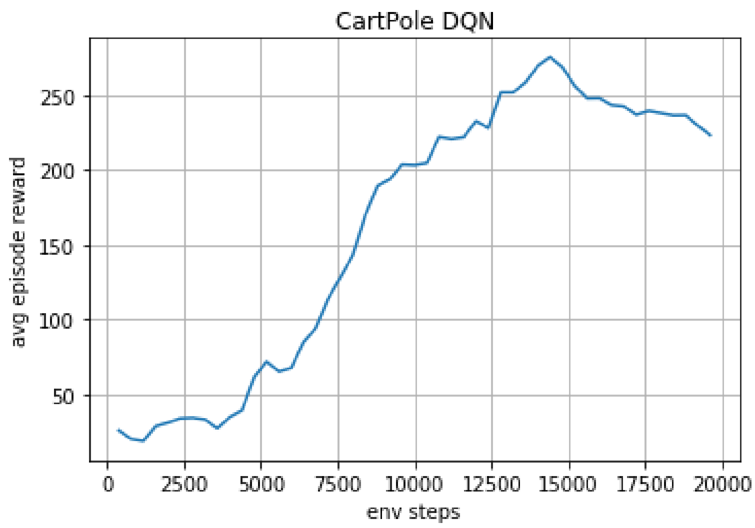
```
93.96 |env_step   9200 |time 00:09 rem 00:10
iter   4800 |loss   0.01 |n_ep   181 |ep_len  203.8 |ep_rew 203.82 |raw_ep_rew 2
03.82 |env_step   9600 |time 00:09 rem 00:10
iter   5000 |loss   0.07 |n_ep   183 |ep_len  203.4 |ep_rew 203.39 |raw_ep_rew 2
03.39 |env_step  10000 |time 00:10 rem 00:10
iter   5200 |loss   0.01 |n_ep   184 |ep_len  204.8 |ep_rew 204.76 |raw_ep_rew 2
04.76 |env_step  10400 |time 00:10 rem 00:10
iter   5400 |loss   0.04 |n_ep   186 |ep_len  222.5 |ep_rew 222.46 |raw_ep_rew 2
22.46 |env_step  10800 |time 00:11 rem 00:09
iter   5600 |loss   0.01 |n_ep   188 |ep_len  221.0 |ep_rew 220.97 |raw_ep_rew 2
20.97 |env_step  11200 |time 00:11 rem 00:09
iter   5800 |loss   0.08 |n_ep   189 |ep_len  222.3 |ep_rew 222.28 |raw_ep_rew 2
22.28 |env_step  11600 |time 00:12 rem 00:08
iter   6000 |loss   0.00 |n_ep   191 |ep_len  232.9 |ep_rew 232.87 |raw_ep_rew 2
32.87 |env_step  12000 |time 00:12 rem 00:08
iter   6200 |loss   0.00 |n_ep   192 |ep_len  228.3 |ep_rew 228.29 |raw_ep_rew 2
28.29 |env_step  12400 |time 00:13 rem 00:08
iter   6400 |loss   0.01 |n_ep   194 |ep_len  252.1 |ep_rew 252.12 |raw_ep_rew 2
52.12 |env_step  12800 |time 00:13 rem 00:07
iter   6600 |loss   0.03 |n_ep   194 |ep_len  252.1 |ep_rew 252.12 |raw_ep_rew 2
52.12 |env_step  13200 |time 00:14 rem 00:07
iter   6800 |loss   0.01 |n_ep   196 |ep_len  258.7 |ep_rew 258.66 |raw_ep_rew 2
58.66 |env_step  13600 |time 00:14 rem 00:06
iter   7000 |loss   0.03 |n_ep   197 |ep_len  269.8 |ep_rew 269.79 |raw_ep_rew 2
69.79 |env_step  14000 |time 00:15 rem 00:06
iter   7200 |loss   0.00 |n_ep   198 |ep_len  275.7 |ep_rew 275.71 |raw_ep_rew 2
75.71 |env_step  14400 |time 00:15 rem 00:06
iter   7400 |loss   0.00 |n_ep   200 |ep_len  269.0 |ep_rew 268.99 |raw_ep_rew 2
68.99 |env_step  14800 |time 00:16 rem 00:05
iter   7600 |loss   0.01 |n_ep   202 |ep_len  256.6 |ep_rew 256.57 |raw_ep_rew 2
56.57 |env_step  15200 |time 00:16 rem 00:05
iter   7800 |loss   0.00 |n_ep   204 |ep_len  248.1 |ep_rew 248.13 |raw_ep_rew 2
48.13 |env_step  15600 |time 00:17 rem 00:04
iter   8000 |loss   0.08 |n_ep   206 |ep_len  248.2 |ep_rew 248.25 |raw_ep_rew 2
48.25 |env_step  16000 |time 00:17 rem 00:04
iter   8200 |loss   0.03 |n_ep   207 |ep_len  243.7 |ep_rew 243.72 |raw_ep_rew 2
43.72 |env_step  16400 |time 00:18 rem 00:03
iter   8400 |loss   0.01 |n_ep   209 |ep_len  242.8 |ep_rew 242.84 |raw_ep_rew 2
42.84 |env_step  16800 |time 00:18 rem 00:03
iter   8600 |loss   0.00 |n_ep   211 |ep_len  237.1 |ep_rew 237.11 |raw_ep_rew 2
37.11 |env_step  17200 |time 00:19 rem 00:03
iter   8800 |loss   0.01 |n_ep   213 |ep_len  239.7 |ep_rew 239.72 |raw_ep_rew 2
39.72 |env_step  17600 |time 00:19 rem 00:02
iter   9000 |loss   0.01 |n_ep   215 |ep_len  238.3 |ep_rew 238.27 |raw_ep_rew 2
38.27 |env_step  18000 |time 00:20 rem 00:02
iter   9200 |loss   0.01 |n_ep   216 |ep_len  236.7 |ep_rew 236.74 |raw_ep_rew 2
36.74 |env_step  18400 |time 00:20 rem 00:01
iter   9400 |loss   0.01 |n_ep   218 |ep_len  236.9 |ep_rew 236.86 |raw_ep_rew 2
36.86 |env_step  18800 |time 00:21 rem 00:01
iter   9600 |loss   0.00 |n_ep   220 |ep_len  229.8 |ep_rew 229.82 |raw_ep_rew 2
29.82 |env_step  19200 |time 00:21 rem 00:00
iter   9800 |loss   0.05 |n_ep   222 |ep_len  223.6 |ep_rew 223.59 |raw_ep_rew 2
23.59 |env_step  19600 |time 00:22 rem 00:00
save checkpoint to cartpole_dqn/9999.pth
```

In [8]:
```
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 [10]:
```
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

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

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):

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

The quantity in the inner-most parentheses $A(s_t, a_t) = Q(s_t, a_t) - V(s_t) = (\mathrm{E} \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}) - V(s_t)$ 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:

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

   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.

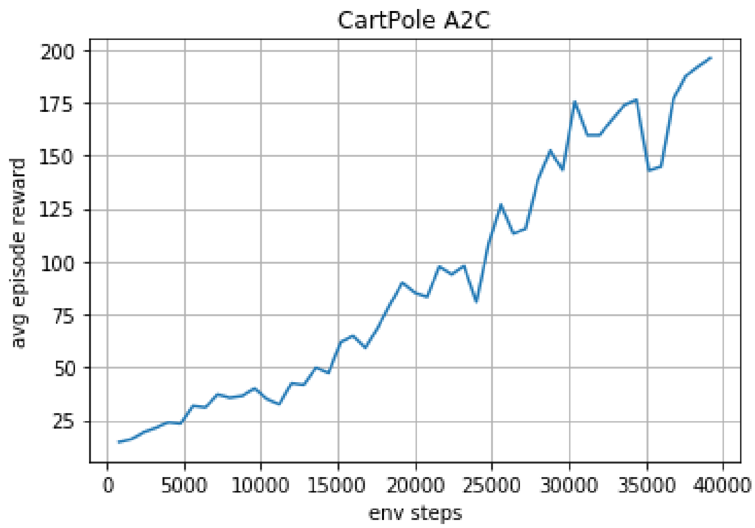
In [11]:
```
%run Main.py  \
    --niter 10000   \
    --env CartPole-v1   \
    --algo a2c  \
    --nproc 4   \
    --lr 0.001  \
    --train_freq 16 \
    --train_start 0 \
    --batch_size 64     \
    --discount 0.996    \
    --value_coef 0.01    \
    --print_freq 200    \
    --checkpoint_freq 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='CartPole-v1', eps_decay=200000, frame_skip=1, frame_stack=4, loa
d='', log='log.txt', lr=0.001, niter=10000, nproc=4, parallel_env=0, print_freq=
200, replay_size=1000000, save_dir='cartpole_a2c/', target_update=2500, train_fr
eq=16, 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([12
8, 4]), True), ('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   0.86 |n_ep    48 |ep_len   14.7 |ep_rew  14.66 |raw_ep_rew
14.66 |env_step    800 |time 00:00 rem 00:23
iter    400 |loss   0.91 |n_ep    90 |ep_len   16.0 |ep_rew  16.03 |raw_ep_rew
16.03 |env_step   1600 |time 00:00 rem 00:21
iter    600 |loss   0.91 |n_ep   130 |ep_len   19.3 |ep_rew  19.30 |raw_ep_rew
19.30 |env_step   2400 |time 00:01 rem 00:19
iter    800 |loss   0.71 |n_ep   169 |ep_len   21.4 |ep_rew  21.39 |raw_ep_rew
21.39 |env_step   3200 |time 00:01 rem 00:19
iter   1000 |loss   0.80 |n_ep   204 |ep_len   24.0 |ep_rew  23.96 |raw_ep_rew
23.96 |env_step   4000 |time 00:02 rem 00:19
iter   1200 |loss   0.89 |n_ep   235 |ep_len   23.3 |ep_rew  23.32 |raw_ep_rew
23.32 |env_step   4800 |time 00:02 rem 00:18
iter   1400 |loss   1.04 |n_ep   262 |ep_len   31.8 |ep_rew  31.76 |raw_ep_rew
31.76 |env_step   5600 |time 00:02 rem 00:18
iter   1600 |loss   0.64 |n_ep   287 |ep_len   30.9 |ep_rew  30.93 |raw_ep_rew
30.93 |env_step   6400 |time 00:03 rem 00:17
iter   1800 |loss   1.01 |n_ep   310 |ep_len   37.1 |ep_rew  37.12 |raw_ep_rew
37.12 |env_step   7200 |time 00:03 rem 00:17
iter   2000 |loss   0.91 |n_ep   333 |ep_len   35.6 |ep_rew  35.60 |raw_ep_rew
35.60 |env_step   8000 |time 00:04 rem 00:16
iter   2200 |loss   0.60 |n_ep   356 |ep_len   36.5 |ep_rew  36.48 |raw_ep_rew
36.48 |env_step   8800 |time 00:04 rem 00:16
iter   2400 |loss   0.59 |n_ep   376 |ep_len   39.9 |ep_rew  39.89 |raw_ep_rew
39.89 |env_step   9600 |time 00:05 rem 00:15
iter   2600 |loss   0.84 |n_ep   400 |ep_len   35.1 |ep_rew  35.08 |raw_ep_rew
35.08 |env_step  10400 |time 00:05 rem 00:15
iter   2800 |loss   0.56 |n_ep   421 |ep_len   32.4 |ep_rew  32.40 |raw_ep_rew
32.40 |env_step  11200 |time 00:05 rem 00:14
iter   3000 |loss   0.68 |n_ep   440 |ep_len   42.5 |ep_rew  42.45 |raw_ep_rew
42.45 |env_step  12000 |time 00:06 rem 00:14
iter   3200 |loss   0.62 |n_ep   457 |ep_len   41.7 |ep_rew  41.69 |raw_ep_rew
41.69 |env_step  12800 |time 00:06 rem 00:14
iter   3400 |loss   0.49 |n_ep   471 |ep_len   49.8 |ep_rew  49.84 |raw_ep_rew
49.84 |env_step  13600 |time 00:07 rem 00:13
iter   3600 |loss   0.56 |n_ep   487 |ep_len   47.2 |ep_rew  47.16 |raw_ep_rew
47.16 |env_step  14400 |time 00:07 rem 00:13
iter   3800 |loss   0.62 |n_ep   498 |ep_len   61.9 |ep_rew  61.86 |raw_ep_rew
61.86 |env_step  15200 |time 00:07 rem 00:12
iter   4000 |loss   0.91 |n_ep   510 |ep_len   64.7 |ep_rew  64.72 |raw_ep_rew
64.72 |env_step  16000 |time 00:08 rem 00:12
iter   4200 |loss   1.05 |n_ep   524 |ep_len   59.1 |ep_rew  59.09 |raw_ep_rew
59.09 |env_step  16800 |time 00:08 rem 00:11
iter   4400 |loss   0.95 |n_ep   534 |ep_len   68.6 |ep_rew  68.63 |raw_ep_rew
```

```
68.63 |env_step  17600 |time 00:09 rem 00:11
iter   4600 |loss   0.48 |n_ep   545 |ep_len   79.7 |ep_rew  79.66 |raw_ep_rew
79.66 |env_step  18400 |time 00:09 rem 00:11
iter   4800 |loss   0.94 |n_ep   553 |ep_len   89.8 |ep_rew  89.83 |raw_ep_rew
89.83 |env_step  19200 |time 00:09 rem 00:10
iter   5000 |loss   0.39 |n_ep   564 |ep_len   85.1 |ep_rew  85.13 |raw_ep_rew
85.13 |env_step  20000 |time 00:10 rem 00:10
iter   5200 |loss   0.03 |n_ep   575 |ep_len   83.2 |ep_rew  83.21 |raw_ep_rew
83.21 |env_step  20800 |time 00:10 rem 00:09
iter   5400 |loss   0.30 |n_ep   580 |ep_len   97.4 |ep_rew  97.42 |raw_ep_rew
97.42 |env_step  21600 |time 00:11 rem 00:09
iter   5600 |loss   0.47 |n_ep   589 |ep_len   93.7 |ep_rew  93.69 |raw_ep_rew
93.69 |env_step  22400 |time 00:11 rem 00:08
iter   5800 |loss   0.80 |n_ep   597 |ep_len   97.7 |ep_rew  97.74 |raw_ep_rew
97.74 |env_step  23200 |time 00:11 rem 00:08
iter   6000 |loss   0.69 |n_ep   605 |ep_len   80.7 |ep_rew  80.75 |raw_ep_rew
80.75 |env_step  24000 |time 00:12 rem 00:08
iter   6200 |loss   0.83 |n_ep   609 |ep_len  108.2 |ep_rew 108.20 |raw_ep_rew 1
08.20 |env_step  24800 |time 00:12 rem 00:07
iter   6400 |loss   0.93 |n_ep   616 |ep_len  126.9 |ep_rew 126.87 |raw_ep_rew 1
26.87 |env_step  25600 |time 00:12 rem 00:07
iter   6600 |loss   0.36 |n_ep   624 |ep_len  113.0 |ep_rew 113.02 |raw_ep_rew 1
13.02 |env_step  26400 |time 00:13 rem 00:06
iter   6800 |loss   0.95 |n_ep   628 |ep_len  115.4 |ep_rew 115.36 |raw_ep_rew 1
15.36 |env_step  27200 |time 00:13 rem 00:06
iter   7000 |loss   0.77 |n_ep   633 |ep_len  138.4 |ep_rew 138.38 |raw_ep_rew 1
38.38 |env_step  28000 |time 00:14 rem 00:06
iter   7200 |loss   0.79 |n_ep   639 |ep_len  152.3 |ep_rew 152.27 |raw_ep_rew 1
52.27 |env_step  28800 |time 00:14 rem 00:05
iter   7400 |loss   0.72 |n_ep   642 |ep_len  143.1 |ep_rew 143.07 |raw_ep_rew 1
43.07 |env_step  29600 |time 00:14 rem 00:05
iter   7600 |loss   0.97 |n_ep   647 |ep_len  175.3 |ep_rew 175.35 |raw_ep_rew 1
75.35 |env_step  30400 |time 00:15 rem 00:04
iter   7800 |loss  -0.15 |n_ep   652 |ep_len  159.5 |ep_rew 159.46 |raw_ep_rew 1
59.46 |env_step  31200 |time 00:15 rem 00:04
iter   8000 |loss  -0.08 |n_ep   660 |ep_len  159.5 |ep_rew 159.46 |raw_ep_rew 1
59.46 |env_step  32000 |time 00:16 rem 00:04
iter   8200 |loss   0.13 |n_ep   663 |ep_len  166.7 |ep_rew 166.66 |raw_ep_rew 1
66.66 |env_step  32800 |time 00:16 rem 00:03
iter   8400 |loss   0.79 |n_ep   667 |ep_len  173.6 |ep_rew 173.61 |raw_ep_rew 1
73.61 |env_step  33600 |time 00:16 rem 00:03
iter   8600 |loss   0.09 |n_ep   671 |ep_len  176.3 |ep_rew 176.25 |raw_ep_rew 1
76.25 |env_step  34400 |time 00:17 rem 00:02
iter   8800 |loss   0.75 |n_ep   678 |ep_len  142.8 |ep_rew 142.82 |raw_ep_rew 1
42.82 |env_step  35200 |time 00:17 rem 00:02
iter   9000 |loss   0.77 |n_ep   682 |ep_len  144.6 |ep_rew 144.64 |raw_ep_rew 1
44.64 |env_step  36000 |time 00:18 rem 00:01
iter   9200 |loss   1.07 |n_ep   685 |ep_len  176.5 |ep_rew 176.55 |raw_ep_rew 1
76.55 |env_step  36800 |time 00:18 rem 00:01
iter   9400 |loss   0.04 |n_ep   690 |ep_len  187.5 |ep_rew 187.47 |raw_ep_rew 1
87.47 |env_step  37600 |time 00:18 rem 00:01
iter   9600 |loss   0.94 |n_ep   692 |ep_len  191.9 |ep_rew 191.87 |raw_ep_rew 1
91.87 |env_step  38400 |time 00:19 rem 00:00
iter   9800 |loss   0.50 |n_ep   698 |ep_len  195.8 |ep_rew 195.81 |raw_ep_rew 1
95.81 |env_step  39200 |time 00:19 rem 00:00
save checkpoint to cartpole_a2c/9999.pth
```

In [12]:
```
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 [31]:
```
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()
reward_count = 0
for _ in range(250):
    env.render()
    state, reward, done, _ = env.step(agent.act([state])[0])
    reward_count += 1
    if done: break
    time.sleep(0.1)
env.close()
print('reward_count: ', reward_count)
```

```
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([12
8, 4]), True), ('fc1.bias', torch.Size([128]), True), ('fc2.weight', torch.Size
([1, 128]), True), ('fc2.bias', torch.Size([1]), True)]

reward_count:  250
```

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 [37]:

```python
## 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!
```

In [38]:

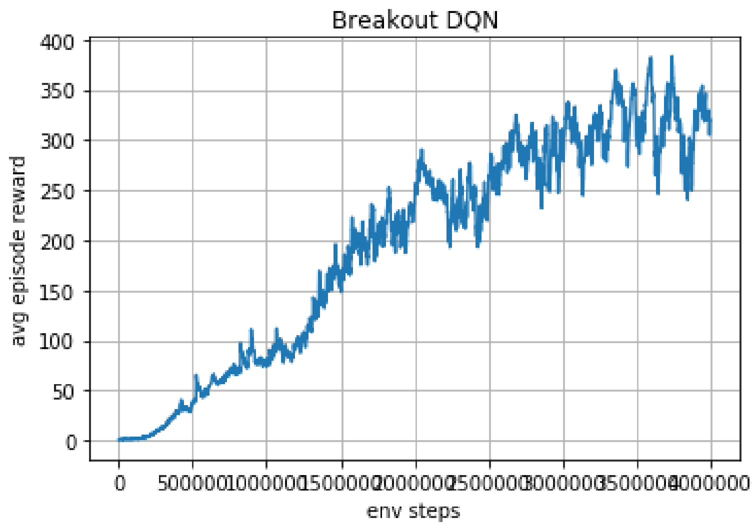
```python
print(net)
```

```
SimpleCNN(
  (conv_layers): Sequential(
    (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc_layers): Sequential(
    (0): Linear(in_features=3136, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=4, bias=True)
  )
)
```

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.

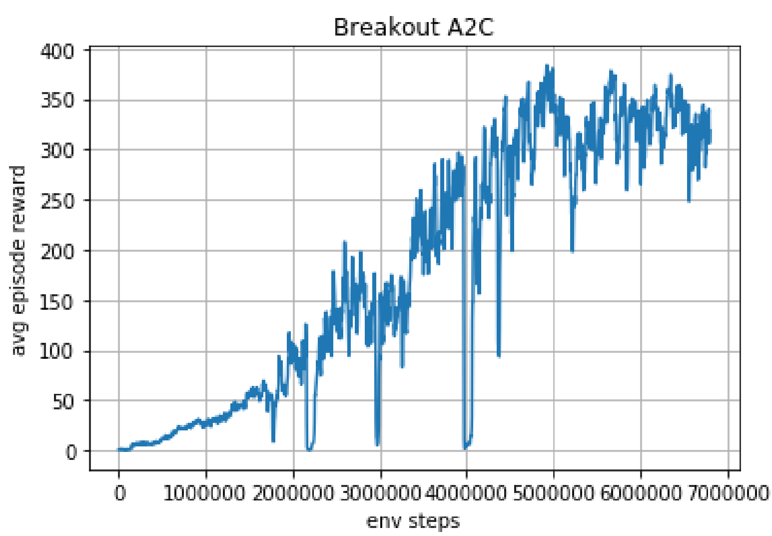
In [39]:
```
plot_curve('log_breakout_dqn.txt', 'Breakout DQN')
```



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 [40]:
```
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.

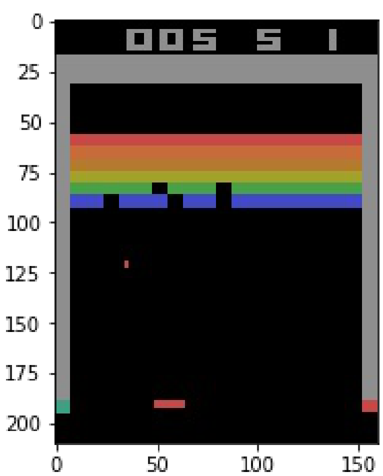
(1) `module load python/2.0.0` and run `Draw.py` on BlueWaters (it's ok to run this locally, no need to start a job).

(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 [42]:

```
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: First issue with tabular algorithm is set of states. During training we build our knowledge of our environment and update our q-table. But in making decisions we are limited to the information given in the table. In practice in realistic environment we never experience exactly the same state multiple times. So if we see completely new state we will get lost in tabular setting and not get any information on the action we need to perform. Another issue is resource and performance. In large scale environments, number of states are very large. Simple games like Tetris has 10^60 possible states. Atari games have astronomical number of possible states. So putting everything in table is not feasible. So we need to find single general function which will give values for all possible, even for those which we have never seen before. Neural networks are the best approach for function approximation, they have the ability to compute any function. If the number of hidden units is made sufficiently large, under certain conditions a neural network trained with stochastic gradient descent does converge to the global minimum.

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: Q-learning is a value based method, we calculate the Q-values for all possible actions in action space for a given state and we pick the max value and its corresponding action. But in high dimensional/continuous action space, where the number of actions is a lot or not imaginable, like a robot walking, we need to learn the optimal policy for higher dimensional action space, which policy based methods exactly do. There is no straightforward way to handle continuous actions in Q-Learning. In policy based methods like Policy Gradient, we directly learn our policy function π without worrying about a value function, which means we can choose actions without calculate the Q(S,A) values. Policy gradient methods work better than value-based methods (like DQN) with continuous action spaces. Policy based methods in non-deterministic environment it can learn the stochastic policy (outputs the probabilities for every action) which is useful for handling the exploration/exploitation trade off.

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: Policy gradient algorithm (A2C) can be extended to continuous action space. In discrete action space, the policy gradient algorithm generates logits which pass through softmax function to give probabilities between 0 and 1 which all add up to 1. This is then used as a probability distribution to pick a random action ensuring exploration, and the more the network gets trained and becomes confident about the correct action the exploitation rate increases and likewise exploration rate decreases. Continuous action tasks rather than discrete probabilities take floating point inputs in a certain range say -1 to +1. So in this case in order to facilitate exploration sample values will be drawn from normal probability distribution. In this case the policy network will have two output heads instead of one, 'mean' and 'standard deviation' of the probability distribution function. As the network gets more and more certain about the optimum value of the output, standard deviation gets smaller and smaller, which indicates we are expoiting instead of exploring. With discrete actions loss functions was based on log probability, and continuous action space will have log probability of normal distribution, ebven though the equation will be little different. So for actor-critic to work continuous action space, first we have to modify the policy head to mean and standard deviation of each continuous action. Then we have to modify the loss function to the negative log probability of the normal distribution. Lastly we have to modify the entropy bonus to entropy of a normal distribution, and we can use stochastic gradient descent on the modified loss function.

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: In on-policy methods, training samples are collected according to the target policy — the very same policy that we are trying to optimize for. A2c can be extended to off-policy data, the off-policy methods does not require full trajectories and can reuse any past episodes like experience replay etc. for much better sample efficiency. The sample collection follows a behavior policy different from the target policy and perform better exploration. In off-policy, Qπ the action-value function is estimated with respect to target policy π, and not behaviour policy. It is very hard to compute gradient of Qπ(s,a) in practical scenarios. But if we use an approximated gradient with the gradient of Q, there is still guarantee of the policy improvement and eventually achieve the true local minimum as was proved by (Degris, White & Sutton, 2012). So we can say that when extending policy gradient in the off-policy setting, we can adjust it with a weighted sum, and the weight is the ratio of the target policy to the behavior policy. weighted sum = (target_policy)/(behaviour_policy)

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: Lets discuss below different RL algorithms. (1) There are model-free and model-based algorithms. Model-free algorithms rely on trial-and-error to update its knowledge. They learns directly from experience, perform actions in the real world, then collect reward from the environment, and update their value functions. As a result, it does not require space to store all the combination of states and actions, and are a reliable choice as the state-space and action-space grows. Conversely Model-Based algorithm aims to construct a model based on the real-world interactions, and then use this model to simulate the further episodes, not in the real environment but by applying them to the constructed model and get the results returned by that model. This gives the advantage of speeding the learning, since there is no need to wait for the environment to respond nor to reset the environment to some state in order to resume learning. On the negative side however, if the model is inaccurate, we risk learning something completely different from the reality. Also model-based algorithms become impractical as the state-space and action-space grows. (2) There are on-policy and off-policy RL algorithms. There are two phases of an RL algorithm: the learning (or training) phase and the inference (or behaviour) phase (after the training phase). The distinction between on-policy and off-policy algorithms only concerns the training phase. Off-policy algorithm during training uses a behaviour policy (policy it uses to select actions) that is different than the optimal policy it tries to estimate (the optimal policy). For example, Q-Learning is an off-policy, model-free RL algorithm based on the Bellman Equation. It updates its Q-values using the Q-value of the next state s′ and the greedy action a′. It estimates the total discounted future reward for state-action pairs assuming a greedy policy were followed despite the fact that it's not following a greedy policy. On otherhand, an on-policy algorithm that during training chooses actions using a policy that is derived from the current estimate of the optimal policy, while the updates are also based on the current estimate of the optimal policy. For example, SARSA is on-policy, it updates its Q-values using the Q-value of the next state s′ and the current policy's action a″. It estimates the return for state-action pairs assuming the current policy continues to be followed. Because SARSA follows the action which is actually being taken in the next step, it has the advantage that the policy that it follows will be more optimal and learning will be faster. Off-policy learning in general has higher per-sample variance than On-policy, and may suffer from problems converging as a result, for example while training neural networks via Q-learning. On-policy algorithm like SARSA will approach convergence allowing for possible penalties from exploratory moves, while off-policy algorithm like Q-learning will ignore them. That makes On-policy more conservative - if there is risk of a large negative reward close to the optimal path, Q-learning will tend to trigger that reward whilst exploring, whilst SARSA will tend to avoid a dangerous optimal path and only slowly learn to use it when the exploration parameters are reduced. To make a choice between on and off policy, there are many real-world constraints. For example, If there is a robot and huge money is at stake, then it would be a good idea train the robot in simulation and to prefer more conservative learning algorithm that avoids high risk. If my goal is to train an optimal agent in simulation, or in a low-cost and fast-iterating environment,then off-policy algorithm Q-learning is a good choice, due to the ability to learn optimal policy directly. If my agent learns online, and I want rewards gained whilst learning, then on-policy algorithm like SARSA may be a better choice. (3)Lets discuss DQN algorithm.The main weakness of Q-learning is lack of generality. For states that the Q-learning agent has not seen before, it has no clear path to guide its next action, that is it does not have the ability to estimate values in unseen states. It is in this area that DQN excels, DQN leverages a Neural Network to estimate the Q-value function. In DQN, if training data is highly correlate and less data-efficient then 'experience replay' technique is used for training DQN. There is another technique, 'separate target network', in which it separates Target Network from Learning Network so that variance and fluctuations becomes less severe and stability in training increases. (4) Although DQN achieved high success in higher dimensional problem, but its action space is still discrete and in many physical control tasks if we discretize the action space too finely, we wind up having an action space that is too large, and it becomes extremely hard to converge. Deep Deterministic Policy Gradient (DDPG) is can be used in that scenario, as it borrows the ideas of experience replay and separate target network from DQN.