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: afausti2

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:\users\penti\miniconda3\envs\cs547\lib\site-packages (0.15.4)
Requirement already satisfied: six in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from gym) (1.12.0)
Requirement already satisfied: pyglet<=1.3.2,>=1.2.0 in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from gym) (1.3.2)
Requirement already satisfied: scipy in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from gym) (1.3.2)
Requirement already satisfied: cloudpickle~=1.2.0 in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from gym) (1.2.2)
Requirement already satisfied: numpy>=1.10.4 in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from gym) (1.17.2)
Requirement already satisfied: opencv-python in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from gym) (4.1.1.26)
Requirement already satisfied: future in c:\users\penti\miniconda3\envs\cs547\lib\site-packages (from pyglet<=1.3.2,>=1.2.0->gym) (0.18.2)
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 [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 rand
om 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)](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!" % 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 [5]:
```
import numpy as np
from Algo import DQN
class Nothing: pass
dummy_obs_space, dummy_act_space = Nothing(), Nothing()
dummy_obs_space.shape = [10]
dummy_act_space.n = 3

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

for t in range(3):
    dqn.observe([np.random.randn(10).astype('float32')], [np.random.randint(3
)],
                [(np.random.randn(10).astype('float32'), np.random.rand(), Fal
se, 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.b
ias', torch.Size([128]), True), ('fc2.weight', torch.Size([3, 128]), True),
('fc2.bias', torch.Size([3]), True)]

0.1998388022184372 > 0.19550158083438873 ?
DQN.train test passed!
```

P1 (10 pts): Run DQN on CartPole and plot the learning curve (i.e. averaged episodic reward against env steps).

Your code should be able to achieve **>150** averaged reward in 10000 iterations (20000 simulation steps) in only a few minutes. This is a good indication that the implementation is correct. It's ok that the curve is not always monotonically increasing because of randomness in training.

In [6]:
```
%run Main.py  \
    --niter 10000   \
    --env CartPole-v1   \
    --algo dqn  \
    --nproc 2   \
    --lr 0.001  \
    --train_freq 1  \
    --train_start 100   \
    --replay_size 20000 \
    --batch_size 64     \
    --discount 0.996    \
    --target_update 1000    \
    --eps_decay 4000    \
    --print_freq 200    \
    --checkpoint_freq 20000 \
    --save_dir cartpole_dqn \
    --log log.txt \
    --parallel_env 0
```

```
Namespace(algo='dqn', batch_size=64, checkpoint_freq=20000, discount=0.996, e
nt_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=200, replay_size=20000, save_dir='cartpole_dqn/', target_update=1000, t
rain_freq=1, train_start=100, value_coef=0.5)
observation space: Box(4,)
action space: Discrete(2)
running on device cuda
parameters to optimize: [('fc1.weight', torch.Size([128, 4]), True), ('fc1.bi
as', 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    20 |ep_len   19.6 |ep_rew  19.60 |raw_ep_re
w  19.60 |env_step    400 |time 00:00 rem 00:47
iter    400 |loss   0.00 |n_ep    39 |ep_len   20.9 |ep_rew  20.89 |raw_ep_re
w  20.89 |env_step    800 |time 00:02 rem 01:01
iter    600 |loss   0.00 |n_ep    59 |ep_len   20.9 |ep_rew  20.94 |raw_ep_re
w  20.94 |env_step   1200 |time 00:04 rem 01:04
iter    800 |loss   0.00 |n_ep    79 |ep_len   18.9 |ep_rew  18.89 |raw_ep_re
w  18.89 |env_step   1600 |time 00:05 rem 01:05
iter   1000 |loss   0.00 |n_ep    99 |ep_len   19.1 |ep_rew  19.08 |raw_ep_re
w  19.08 |env_step   2000 |time 00:07 rem 01:05
iter   1200 |loss   0.02 |n_ep   120 |ep_len   19.1 |ep_rew  19.13 |raw_ep_re
w  19.13 |env_step   2400 |time 00:08 rem 01:04
iter   1400 |loss   0.03 |n_ep   138 |ep_len   22.5 |ep_rew  22.53 |raw_ep_re
w  22.53 |env_step   2800 |time 00:10 rem 01:03
iter   1600 |loss   0.02 |n_ep   163 |ep_len   15.7 |ep_rew  15.69 |raw_ep_re
w  15.69 |env_step   3200 |time 00:11 rem 01:02
iter   1800 |loss   0.02 |n_ep   190 |ep_len   14.9 |ep_rew  14.94 |raw_ep_re
w  14.94 |env_step   3600 |time 00:13 rem 01:01
iter   2000 |loss   0.03 |n_ep   214 |ep_len   17.6 |ep_rew  17.58 |raw_ep_re
w  17.58 |env_step   4000 |time 00:15 rem 01:00
iter   2200 |loss   0.03 |n_ep   237 |ep_len   17.5 |ep_rew  17.50 |raw_ep_re
w  17.50 |env_step   4400 |time 00:16 rem 00:59
iter   2400 |loss   0.03 |n_ep   249 |ep_len   29.5 |ep_rew  29.47 |raw_ep_re
w  29.47 |env_step   4800 |time 00:18 rem 00:58
iter   2600 |loss   0.03 |n_ep   267 |ep_len   23.2 |ep_rew  23.21 |raw_ep_re
w  23.21 |env_step   5200 |time 00:20 rem 00:56
iter   2800 |loss   0.02 |n_ep   280 |ep_len   33.5 |ep_rew  33.50 |raw_ep_re
w  33.50 |env_step   5600 |time 00:21 rem 00:55
iter   3000 |loss   0.02 |n_ep   293 |ep_len   33.2 |ep_rew  33.17 |raw_ep_re
w  33.17 |env_step   6000 |time 00:23 rem 00:54
iter   3200 |loss   0.04 |n_ep   300 |ep_len   35.6 |ep_rew  35.65 |raw_ep_re
w  35.65 |env_step   6400 |time 00:25 rem 00:53
iter   3400 |loss   0.02 |n_ep   304 |ep_len   58.1 |ep_rew  58.14 |raw_ep_re
w  58.14 |env_step   6800 |time 00:26 rem 00:52
iter   3600 |loss   0.05 |n_ep   308 |ep_len   78.3 |ep_rew  78.26 |raw_ep_re
w  78.26 |env_step   7200 |time 00:28 rem 00:50
iter   3800 |loss   0.04 |n_ep   310 |ep_len   94.1 |ep_rew  94.07 |raw_ep_re
w  94.07 |env_step   7600 |time 00:30 rem 00:49
iter   4000 |loss   0.04 |n_ep   313 |ep_len  109.5 |ep_rew 109.51 |raw_ep_re
w 109.51 |env_step   8000 |time 00:32 rem 00:48
iter   4200 |loss   0.05 |n_ep   315 |ep_len  124.4 |ep_rew 124.41 |raw_ep_re
w 124.41 |env_step   8400 |time 00:34 rem 00:47
iter   4400 |loss   0.03 |n_ep   317 |ep_len  144.5 |ep_rew 144.51 |raw_ep_re
w 144.51 |env_step   8800 |time 00:36 rem 00:45
```

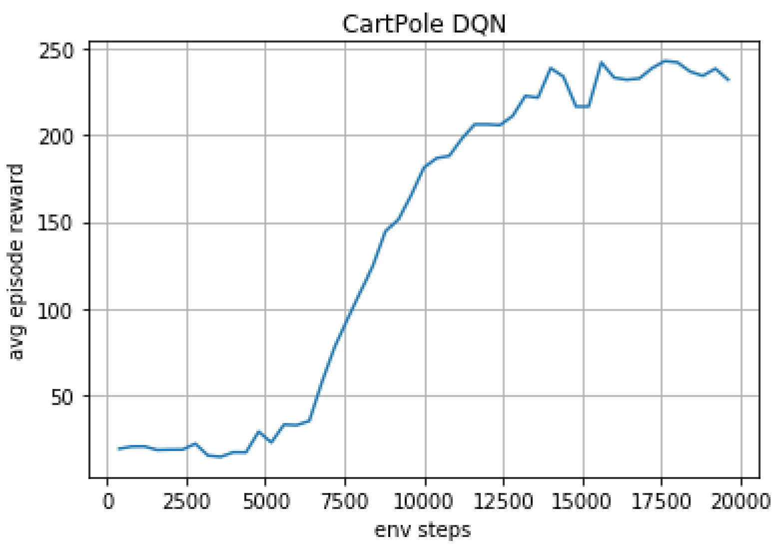
```
iter   4600 |loss   0.06 |n_ep   318 |ep_len  151.5 |ep_rew 151.46 |raw_ep_re
w 151.46 |env_step   9200 |time 00:37 rem 00:44
iter   4800 |loss   0.02 |n_ep   319 |ep_len  165.4 |ep_rew 165.41 |raw_ep_re
w 165.41 |env_step   9600 |time 00:39 rem 00:42
iter   5000 |loss   0.04 |n_ep   321 |ep_len  181.2 |ep_rew 181.19 |raw_ep_re
w 181.19 |env_step  10000 |time 00:41 rem 00:41
iter   5200 |loss   0.02 |n_ep   323 |ep_len  186.6 |ep_rew 186.65 |raw_ep_re
w 186.65 |env_step  10400 |time 00:43 rem 00:40
iter   5400 |loss   0.08 |n_ep   325 |ep_len  187.9 |ep_rew 187.94 |raw_ep_re
w 187.94 |env_step  10800 |time 00:44 rem 00:38
iter   5600 |loss   0.08 |n_ep   327 |ep_len  197.9 |ep_rew 197.91 |raw_ep_re
w 197.91 |env_step  11200 |time 00:46 rem 00:36
iter   5800 |loss   0.11 |n_ep   329 |ep_len  206.1 |ep_rew 206.10 |raw_ep_re
w 206.10 |env_step  11600 |time 00:48 rem 00:34
iter   6000 |loss   0.07 |n_ep   331 |ep_len  206.1 |ep_rew 206.10 |raw_ep_re
w 206.10 |env_step  12000 |time 00:50 rem 00:33
iter   6200 |loss   0.01 |n_ep   332 |ep_len  205.7 |ep_rew 205.69 |raw_ep_re
w 205.69 |env_step  12400 |time 00:51 rem 00:31
iter   6400 |loss   0.01 |n_ep   333 |ep_len  210.9 |ep_rew 210.92 |raw_ep_re
w 210.92 |env_step  12800 |time 00:53 rem 00:29
iter   6600 |loss   0.03 |n_ep   335 |ep_len  222.5 |ep_rew 222.47 |raw_ep_re
w 222.47 |env_step  13200 |time 00:54 rem 00:28
iter   6800 |loss   0.19 |n_ep   336 |ep_len  221.5 |ep_rew 221.52 |raw_ep_re
w 221.52 |env_step  13600 |time 00:56 rem 00:26
iter   7000 |loss   0.01 |n_ep   338 |ep_len  238.5 |ep_rew 238.48 |raw_ep_re
w 238.48 |env_step  14000 |time 00:58 rem 00:24
iter   7200 |loss   0.09 |n_ep   340 |ep_len  233.7 |ep_rew 233.66 |raw_ep_re
w 233.66 |env_step  14400 |time 00:59 rem 00:23
iter   7400 |loss   0.06 |n_ep   343 |ep_len  216.4 |ep_rew 216.39 |raw_ep_re
w 216.39 |env_step  14800 |time 01:01 rem 00:21
iter   7600 |loss   0.01 |n_ep   343 |ep_len  216.4 |ep_rew 216.39 |raw_ep_re
w 216.39 |env_step  15200 |time 01:03 rem 00:19
iter   7800 |loss   0.10 |n_ep   345 |ep_len  241.8 |ep_rew 241.80 |raw_ep_re
w 241.80 |env_step  15600 |time 01:04 rem 00:18
iter   8000 |loss   0.05 |n_ep   347 |ep_len  233.0 |ep_rew 233.03 |raw_ep_re
w 233.03 |env_step  16000 |time 01:06 rem 00:16
iter   8200 |loss   0.02 |n_ep   349 |ep_len  231.8 |ep_rew 231.82 |raw_ep_re
w 231.82 |env_step  16400 |time 01:07 rem 00:14
iter   8400 |loss   0.17 |n_ep   350 |ep_len  232.5 |ep_rew 232.54 |raw_ep_re
w 232.54 |env_step  16800 |time 01:09 rem 00:13
iter   8600 |loss   0.03 |n_ep   352 |ep_len  238.5 |ep_rew 238.48 |raw_ep_re
w 238.48 |env_step  17200 |time 01:11 rem 00:11
iter   8800 |loss   0.14 |n_ep   354 |ep_len  242.6 |ep_rew 242.61 |raw_ep_re
w 242.61 |env_step  17600 |time 01:12 rem 00:09
iter   9000 |loss   0.12 |n_ep   355 |ep_len  241.9 |ep_rew 241.85 |raw_ep_re
w 241.85 |env_step  18000 |time 01:14 rem 00:08
iter   9200 |loss   0.00 |n_ep   357 |ep_len  236.5 |ep_rew 236.53 |raw_ep_re
w 236.53 |env_step  18400 |time 01:15 rem 00:06
iter   9400 |loss   0.12 |n_ep   358 |ep_len  234.1 |ep_rew 234.08 |raw_ep_re
w 234.08 |env_step  18800 |time 01:17 rem 00:04
iter   9600 |loss   0.08 |n_ep   360 |ep_len  238.2 |ep_rew 238.19 |raw_ep_re
w 238.19 |env_step  19200 |time 01:19 rem 00:03
iter   9800 |loss   0.16 |n_ep   362 |ep_len  231.8 |ep_rew 231.84 |raw_ep_re
w 231.84 |env_step  19600 |time 01:20 rem 00:01
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 [8]:
```
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:
$$L_{policy}(\theta) = \frac{1}{NT}\sum_{i=1}^{N}\sum_{t=0}^{T}\left(\log \pi_\theta(s_t^i, a_t^i)\left(\sum_{t'=t}^{T}\gamma^{t'-t} r_{t'}^i - V_\phi(s_t^i)\right)\right)$$
Compute the entropy term for θ:

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.

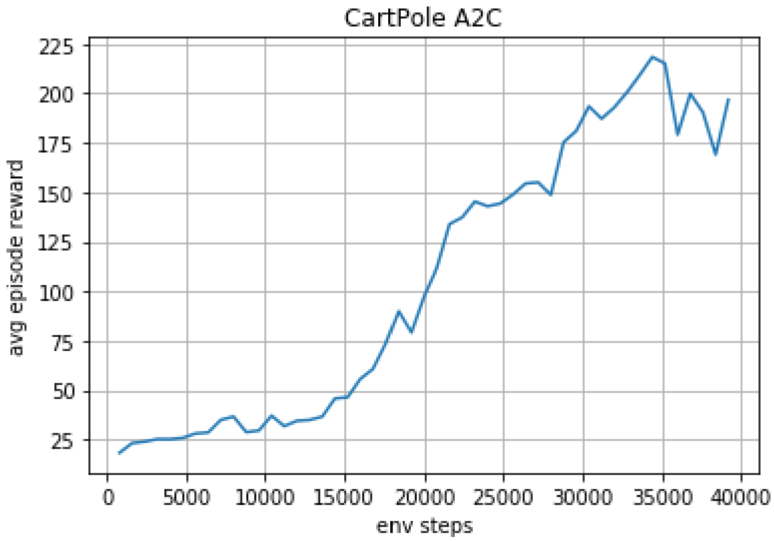
In [9]:
```
%run Main.py  \
    --niter 10000   \
    --env CartPole-v1   \
    --algo a2c  \
    --nproc 4   \
    --lr 0.001  \
    --train_freq 16 \
    --train_start 0 \
    --batch_size 64     \
    --discount 0.996    \
    --value_coef 0.01    \
    --print_freq 200    \
    --checkpoint_freq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

```
Namespace(algo='a2c', batch_size=64, checkpoint_freq=20000, discount=0.996, e
nt_coef=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, pr
int_freq=200, replay_size=1000000, save_dir='cartpole_a2c/', target_update=25
00, train_freq=16, train_start=0, value_coef=0.01)
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.bias', torch.Size([128]), True), ('fc2.weight', torch.Size
([2, 128]), True), ('fc2.bias', torch.Size([2]), True), ('fc1.weight', torch.
Size([128, 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   1.00 |n_ep    39 |ep_len   18.2 |ep_rew  18.23 |raw_ep_re
w  18.23 |env_step    800 |time 00:00 rem 00:27
iter    400 |loss   0.93 |n_ep    70 |ep_len   23.1 |ep_rew  23.15 |raw_ep_re
w  23.15 |env_step   1600 |time 00:01 rem 00:25
iter    600 |loss   0.91 |n_ep   104 |ep_len   23.9 |ep_rew  23.92 |raw_ep_re
w  23.92 |env_step   2400 |time 00:01 rem 00:25
iter    800 |loss   0.83 |n_ep   135 |ep_len   25.2 |ep_rew  25.20 |raw_ep_re
w  25.20 |env_step   3200 |time 00:02 rem 00:24
iter   1000 |loss   0.81 |n_ep   166 |ep_len   25.2 |ep_rew  25.15 |raw_ep_re
w  25.15 |env_step   4000 |time 00:02 rem 00:23
iter   1200 |loss   0.78 |n_ep   198 |ep_len   25.8 |ep_rew  25.80 |raw_ep_re
w  25.80 |env_step   4800 |time 00:03 rem 00:23
iter   1400 |loss   0.76 |n_ep   227 |ep_len   28.0 |ep_rew  28.01 |raw_ep_re
w  28.01 |env_step   5600 |time 00:03 rem 00:22
iter   1600 |loss   0.73 |n_ep   254 |ep_len   28.5 |ep_rew  28.54 |raw_ep_re
w  28.54 |env_step   6400 |time 00:04 rem 00:22
iter   1800 |loss   0.76 |n_ep   276 |ep_len   34.9 |ep_rew  34.91 |raw_ep_re
w  34.91 |env_step   7200 |time 00:04 rem 00:21
iter   2000 |loss   0.63 |n_ep   299 |ep_len   36.6 |ep_rew  36.59 |raw_ep_re
w  36.59 |env_step   8000 |time 00:05 rem 00:21
iter   2200 |loss   0.80 |n_ep   327 |ep_len   28.7 |ep_rew  28.71 |raw_ep_re
w  28.71 |env_step   8800 |time 00:05 rem 00:20
iter   2400 |loss   0.58 |n_ep   352 |ep_len   29.5 |ep_rew  29.52 |raw_ep_re
w  29.52 |env_step   9600 |time 00:06 rem 00:20
iter   2600 |loss   0.56 |n_ep   371 |ep_len   37.0 |ep_rew  37.05 |raw_ep_re
w  37.05 |env_step  10400 |time 00:06 rem 00:19
iter   2800 |loss   0.94 |n_ep   392 |ep_len   31.7 |ep_rew  31.74 |raw_ep_re
w  31.74 |env_step  11200 |time 00:07 rem 00:19
iter   3000 |loss   0.65 |n_ep   413 |ep_len   34.4 |ep_rew  34.44 |raw_ep_re
w  34.44 |env_step  12000 |time 00:08 rem 00:18
iter   3200 |loss   0.89 |n_ep   434 |ep_len   34.9 |ep_rew  34.89 |raw_ep_re
w  34.89 |env_step  12800 |time 00:08 rem 00:18
iter   3400 |loss   0.59 |n_ep   454 |ep_len   36.6 |ep_rew  36.58 |raw_ep_re
w  36.58 |env_step  13600 |time 00:09 rem 00:17
iter   3600 |loss   0.11 |n_ep   472 |ep_len   45.8 |ep_rew  45.75 |raw_ep_re
w  45.75 |env_step  14400 |time 00:09 rem 00:17
iter   3800 |loss   0.63 |n_ep   484 |ep_len   46.4 |ep_rew  46.38 |raw_ep_re
w  46.38 |env_step  15200 |time 00:10 rem 00:16
iter   4000 |loss   0.54 |n_ep   499 |ep_len   55.7 |ep_rew  55.65 |raw_ep_re
w  55.65 |env_step  16000 |time 00:10 rem 00:16
iter   4200 |loss   0.98 |n_ep   507 |ep_len   60.7 |ep_rew  60.73 |raw_ep_re
w  60.73 |env_step  16800 |time 00:11 rem 00:15
```

```
iter   4400 |loss   0.55 |n_ep   518 |ep_len   73.6 |ep_rew  73.60 |raw_ep_re
w  73.60 |env_step  17600 |time 00:11 rem 00:14
iter   4600 |loss   0.64 |n_ep   528 |ep_len   89.9 |ep_rew  89.91 |raw_ep_re
w  89.91 |env_step  18400 |time 00:12 rem 00:14
iter   4800 |loss   0.49 |n_ep   538 |ep_len   79.2 |ep_rew  79.20 |raw_ep_re
w  79.20 |env_step  19200 |time 00:12 rem 00:13
iter   5000 |loss   0.90 |n_ep   545 |ep_len   97.5 |ep_rew  97.45 |raw_ep_re
w  97.45 |env_step  20000 |time 00:13 rem 00:13
iter   5200 |loss   0.31 |n_ep   551 |ep_len  111.7 |ep_rew 111.74 |raw_ep_re
w 111.74 |env_step  20800 |time 00:13 rem 00:12
iter   5400 |loss   0.85 |n_ep   555 |ep_len  133.8 |ep_rew 133.83 |raw_ep_re
w 133.83 |env_step  21600 |time 00:14 rem 00:12
iter   5600 |loss   0.17 |n_ep   563 |ep_len  137.2 |ep_rew 137.24 |raw_ep_re
w 137.24 |env_step  22400 |time 00:15 rem 00:11
iter   5800 |loss   0.82 |n_ep   566 |ep_len  145.3 |ep_rew 145.26 |raw_ep_re
w 145.26 |env_step  23200 |time 00:15 rem 00:11
iter   6000 |loss   0.89 |n_ep   572 |ep_len  142.9 |ep_rew 142.87 |raw_ep_re
w 142.87 |env_step  24000 |time 00:16 rem 00:10
iter   6200 |loss   0.09 |n_ep   577 |ep_len  144.2 |ep_rew 144.23 |raw_ep_re
w 144.23 |env_step  24800 |time 00:16 rem 00:10
iter   6400 |loss  -0.03 |n_ep   582 |ep_len  148.8 |ep_rew 148.76 |raw_ep_re
w 148.76 |env_step  25600 |time 00:17 rem 00:09
iter   6600 |loss   0.90 |n_ep   588 |ep_len  154.4 |ep_rew 154.39 |raw_ep_re
w 154.39 |env_step  26400 |time 00:17 rem 00:09
iter   6800 |loss   0.71 |n_ep   593 |ep_len  155.0 |ep_rew 154.98 |raw_ep_re
w 154.98 |env_step  27200 |time 00:18 rem 00:08
iter   7000 |loss   0.14 |n_ep   597 |ep_len  148.6 |ep_rew 148.61 |raw_ep_re
w 148.61 |env_step  28000 |time 00:18 rem 00:08
iter   7200 |loss   0.83 |n_ep   601 |ep_len  175.3 |ep_rew 175.26 |raw_ep_re
w 175.26 |env_step  28800 |time 00:19 rem 00:07
iter   7400 |loss   0.74 |n_ep   604 |ep_len  181.0 |ep_rew 181.02 |raw_ep_re
w 181.02 |env_step  29600 |time 00:20 rem 00:07
iter   7600 |loss   0.92 |n_ep   608 |ep_len  193.5 |ep_rew 193.48 |raw_ep_re
w 193.48 |env_step  30400 |time 00:20 rem 00:06
iter   7800 |loss   0.32 |n_ep   614 |ep_len  187.2 |ep_rew 187.20 |raw_ep_re
w 187.20 |env_step  31200 |time 00:21 rem 00:05
iter   8000 |loss   0.04 |n_ep   618 |ep_len  192.9 |ep_rew 192.93 |raw_ep_re
w 192.93 |env_step  32000 |time 00:21 rem 00:05
iter   8200 |loss   1.04 |n_ep   620 |ep_len  200.5 |ep_rew 200.54 |raw_ep_re
w 200.54 |env_step  32800 |time 00:22 rem 00:04
iter   8400 |loss   0.58 |n_ep   623 |ep_len  209.2 |ep_rew 209.23 |raw_ep_re
w 209.23 |env_step  33600 |time 00:22 rem 00:04
iter   8600 |loss   0.67 |n_ep   628 |ep_len  218.5 |ep_rew 218.47 |raw_ep_re
w 218.47 |env_step  34400 |time 00:23 rem 00:03
iter   8800 |loss  -0.05 |n_ep   632 |ep_len  215.2 |ep_rew 215.19 |raw_ep_re
w 215.19 |env_step  35200 |time 00:23 rem 00:03
iter   9000 |loss  -0.19 |n_ep   638 |ep_len  179.1 |ep_rew 179.12 |raw_ep_re
w 179.12 |env_step  36000 |time 00:24 rem 00:02
iter   9200 |loss   0.87 |n_ep   640 |ep_len  199.9 |ep_rew 199.91 |raw_ep_re
w 199.91 |env_step  36800 |time 00:24 rem 00:02
iter   9400 |loss  -0.05 |n_ep   646 |ep_len  190.4 |ep_rew 190.37 |raw_ep_re
w 190.37 |env_step  37600 |time 00:25 rem 00:01
iter   9600 |loss   0.20 |n_ep   649 |ep_len  169.0 |ep_rew 169.01 |raw_ep_re
w 169.01 |env_step  38400 |time 00:25 rem 00:01
iter   9800 |loss  -0.14 |n_ep   655 |ep_len  196.9 |ep_rew 196.85 |raw_ep_re
w 196.85 |env_step  39200 |time 00:26 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 [11]:
```
import time
import gym
import Algo
env = gym.make('CartPole-v1')
agent = Algo.ActorCritic(env.observation_space, env.action_space)
agent.load('cartpole_a2c/9999.pth')
state = env.reset()
for _ in range(120):
    env.render()
    state, reward, done, _ = env.step(agent.act([state])[0])
    if done: break
    time.sleep(0.1)
env.close()
```

```
shared net = False, parameters to optimize: [('fc1.weight', torch.Size([128,
4]), True), ('fc1.bias', torch.Size([128]), True), ('fc2.weight', torch.Size
([2, 128]), True), ('fc2.bias', torch.Size([2]), True), ('fc1.weight', torch.
Size([128, 4]), True), ('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/) I spent a lot of time playing them...

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

And verify the output shape with the cell below.

In [12]:
```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!
```

P3 (10 pts): Run the following cell to generate a DQN learning curve.

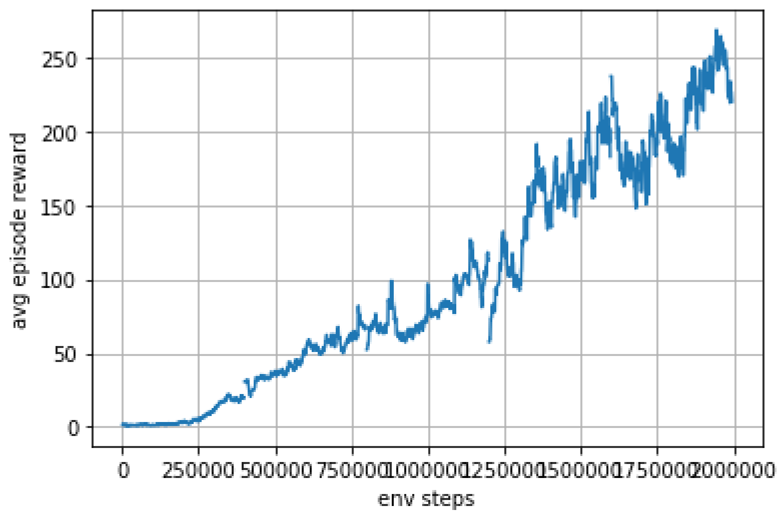
The *maximum* average episodic reward on this curve should be larger than 200 for full credit. (It's ok if the final reward is not as high.) The typical value is around 300. You get 70% credit if $100 \leq$ average episodic reward < 200, 50% credit if $50 \leq$ average episodic reward < 100.

In [26]:
```
# get action steps and average rewards from training checkpoint logs
import os
import fnmatch

log_files = [file for file in os.listdir('breakout_dqn/') if fnmatch.fnmatch(f
ile, 'cp*')]

steps = []
rewards = []
for i, log_file in enumerate(log_files):
    log_file = 'breakout_dqn/' + log_file
    lines = open(log_file, 'r').readlines()
    lines = [l.split() for l in lines if l[:4] == 'iter' and l[11] != 'nan']
    steps.extend([int(l[13]) + i*399000 for l in lines])
    rewards.extend([float(l[11]) for l in lines])

plt.plot(steps, rewards)
plt.xlabel('env steps'); plt.ylabel('avg episode reward'); plt.grid(True)
plt.show()
```

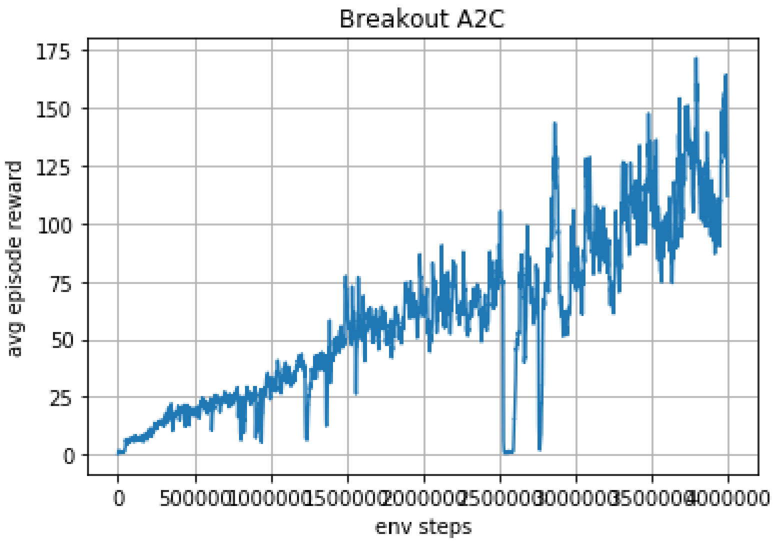


Because of memory limitations on my personal machine, I had to run the training in 200000 iteration long increments. This leads to breaks in the rolling average so the plot has some discontinuities. It's also the reason why I have multiple log files for this part.

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')
```



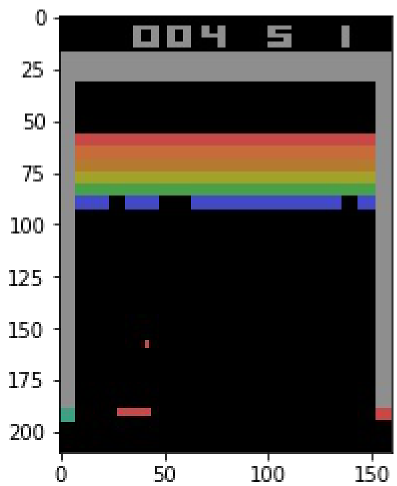
P5 (10 pts): Collect and visualize some game frames by running the script `Draw.py` on BlueWaters.

(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 [3]:
```
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...

As discrete state/action spaces get larger they become impractical to represent as tables in terms of memory. Games like chess and go are good examples of this.

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?

One advantage is that by learning the optimal policy directly, policy gradient methods can learn a stochastic policy.

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?

You could assume $\pi_\theta(a|s)$ is normally distributed.

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

Algorithms like [ACER](https://arxiv.org/abs/1611.01224) accomplish this

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.

I think this just depends on what resources you have available to you. For Google Brain, having AutoRL use 100 GPUs for a few weeks is better than having an engineer tune the hyper parameters for two days on 10 GPUs. From my perspective the better RL algorithm is the one that can be implemented to solve the problem.