Q-Learning & DQNs (12 regular points + 2 extra credit points for both CS4803 and CS7643)

In this section, we will implement a few key parts of the Q-Learning algorithm for two cases - (1) network which is a single linear layer (referred to in RL literature as "Q-learning with linear functi approximation") and (2) A deep (convolutional) Q-network, for some Atari game environments w the states are images.

Optional Readings:

- **Playing Atari with Deep Reinforcement Learning**, Mnih et. al., https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf)
- **The PyTorch DQN Tutorial** https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html (https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html)

In [20]:

```
%load_ext autoreload
%autoreload 2

import numpy as np
import gym

import torch
import torch.nn as nn
import torch.optim as optim

from core.dqn_train import DQNTrain
from utils.test_env import EnvTest
from utils.schedule import LinearExploration, LinearSchedule
from utils.preprocess import greyscale
from utils.wrappers import PreproWrapper, MaxAndSkipEnv

from linear_qnet import LinearQNet
from cnn_qnet import ConvQNet

if torch.cuda.is_available():
    device = torch.device('cuda', 0)
else:
    device = torch.device('cpu')
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Part 1: Setup Q-Learning with Linear Function Approximation

Training Q-networks using (Deep) Q-learning involves a lot of moving parts. However, for this assignment, the scaffolding for the first 3 points listed below is provided in full and you must only complete point 4. You may skip to point 4 if you only care about the implementation required for assignment.

1. **Environments**: We will use the standardized OpenAI Gym framework for environment API (read through [http://gym.openai.com/docs/ (http://gym.openai.com/docs/)](http://gym.openai.com/docs/) if you want to know more details about this interface). Specifically, we will use a custom Test environment define `utils/test_env.py` for initial sanity checks and then Gym-Atari environments later on.

2. **Exploration**: In order to train any RL model, we require experience or "data" gathered from interacting with the environment by taking actions. What policy should we use to collect this experience? Given a Q-network, one may be tempted to define a greedy policy which alway picks the highest valued action at every state. However, this strategy will in most cases not since we may get stuck in a local minima and never explore new states in the environment which may lead to a better reward. Hence, for the purpose of gathering experience (or "data from the environment, it is useful to follow a policy that deviates from the greedy policy sligh order to explore new states. A common strategy used in RL is to follow an ϵ-greedy policy w with probability $0 < \epsilon < 1$ picks a random action instead of the action provided by the gree policy.

3. **Replay Buffers**: Data gathered from a single trajectory of states and actions in the environm provides us with a batch of highly correlated (non IID) data, which leads to high variance in gradient updates and convergence. In order to ameliorate this, replay buffers are used to ga a set of transitions i.e. (state, action, reward, next state) tuples, by executing multiple traject in the environment. Now, for updating the Q-Network, we will first wait to fill up our replay bu with a sufficiently large number of transitions over multiple different trajectories, and then randomly sample a batch of transitions to compute loss and update the models.

4. **Q-Learning network, loss and update**: Finally, we come to the part of Q-learning that we v implement for this assignment -- the Q-network, loss function and update. In particular, we v implement a variant of Q-Learning called "Double Q-Learning", where we will maintain two C networks -- the first Q network is used to pick actions and the second "target" Q network is u to compute Q-values for the picked actions. Here is some referance material on the same - [1 (https://towardsdatascience.com/double-q-learning-the-easy-way-a924c4085ec3)](https://towardsdatascience.com/double-q-learning-the-easy-way-a924c4085ec3), [Blog 2 (https://medium.com/@ameetsd97/deep-double-q-learning-why-you-should-use-it-bedf660d5295)](https://medium.com/@ameetsd97/deep-double-q-learning-why-you-should-use-it-bedf660d5295), but we will not need to get into the details of Double Q-learning for this assignment. Now, let's walk through the steps required to implement this below.

   - **Linear Q-Network**: In `linear_qnet.py` , define the initialization and forward pass of a C network with a single linear layer which takes the state as input and outputs the Q-value for all actions.
   - **Setting up Q-Learning**: In `core/dqn_train.py` , complete the functions `process_stat` `forward_loss` and `update_step` and `update_target_params` . The loss function for ou Networks is defined for a single transition tuple of (state, action, reward, next state) as follows. $Q(s_t, a_t)$ refers to the state-action values computed by our first Q-network at th current state and and for the current actions, $Q_{target}(s_{t+1}, a_{t+1})$ refers to the state-actio values for the next state and all possible future actions computed by the target Q-Netwo

$$
\begin{aligned}
Q_{sample}(s_t) &= r_t \text{ if done} \\
&= r_t + \gamma \max_{a_{t+1}} Q_{target}\left(s_{t+1}, a_{t+1}\right) \text{ otherwise} \\
\text{Loss} &= \left(Q_{sample}(s_t) - Q(s_t, a_t)\right)^2
\end{aligned}
$$

Deliverable 1 (6 points)

Run the following block of code to train a Linear Q-Network. You should get an average reward c ~4.0, full credit will be given if average reward at the final evaluation is above 3.5

In [27]:

```
from configs.p1_linear import config as config_lin

env = EnvTest((5, 5, 1))

# exploration strategy
exp_schedule = LinearExploration(env, config_lin.eps_begin,
        config_lin.eps_end, config_lin.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_lin.lr_begin, config_lin.lr_end,
        config_lin.lr_nsteps)

# train model
model = DQNTrain(LinearQNet, env, config_lin, device)
model.run(exp_schedule, lr_schedule)
```

```
Evaluating...
Average reward: 4.10 +/- 0.00


 9001/10000 [==========================>...] - ETA: 0s - Loss: 0.3539 - Avg_R: 4.
000 - Max_R: 4.1000 - eps: 0.0100 - Grads: 6.5900 - Max_Q: 2.6928 - lr: 0.0010

Evaluating...



Average reward: 4.10 +/- 0.00

10001/10000 [==============================] - 5s - Loss: 3.6390 - Avg_R: 4.0650
Max_R: 4.1000 - eps: 0.0100 - Grads: 4.4510 - Max_Q: 2.5255 - lr: 0.0010

- Training done.
Evaluating...
Average reward: 4.10 +/- 0.00
```

You should get a final average reward of over 4.0 on the test environment.

Part 2: Q-Learning with Deep Q-Networks

In `cnn_qnet.py`, implement the initialization and forward pass of a convolutional Q-network with architecture as described in this DeepMind paper:

"Playing Atari with Deep Reinforcement Learning", Mnih et. al. (https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf (https://www.cs.toronto.edu/~vmnih/docs/dqn.p

Deliverable 2 (4 points)

Run the following block of code to train our Deep Q-Network. You should get an average reward ~4.0, full credit will be given if average reward at the final evaluation is above 3.5

In [28]:

```
from configs.p2_cnn import config as config_cnn

env = EnvTest((80, 80, 1))

# exploration strategy
exp_schedule = LinearExploration(env, config_cnn.eps_begin,
        config_cnn.eps_end, config_cnn.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_cnn.lr_begin, config_cnn.lr_end,
        config_cnn.lr_nsteps)

# train model
model = DQNTrain(ConvQNet, env, config_cnn, device)
model.run(exp_schedule, lr_schedule)
```

```
Average reward: 2.10 +/- 0.00

 901/1000 [==========================>...] - ETA: 0s - Loss: 4.6039 - Avg_R: 2.97
0 - Max_R: 4.1000 - eps: 0.0100 - Grads: 73.5697 - Max_Q: 0.4820 - lr: 0.0001

Evaluating...



Average reward: 4.10 +/- 0.00

1001/1000 [==============================] - 4s - Loss: 2.2167 - Avg_R: 3.9550 -
ax_R: 4.1000 - eps: 0.0100 - Grads: 52.0332 - Max_Q: 0.5473 - lr: 0.0001

- Training done.
Evaluating...
Average reward: 4.10 +/- 0.00
```

You should get a final average reward of over 4.0 on the test environment, similar to the previous case.

Part 3: Playing Atari Games from Pixels - using Linear Function Approximation

Now that we have setup our Q-Learning algorithm and tested it on a simple test environment, we shift to a harder environment - an Atari 2600 game from OpenAI Gym: Pong-v0 (https://gym.openai.com/envs/Pong-v0/ (https://gym.openai.com/envs/Pong-v0/)), where we will RGB images of the game screen as our observations for state.

No additional implementation is required for this part, just run the block of code below (will take around 1 hour to train). We don't expect a simple linear Q-network to do well on such a hard environment - full credit will be given simply for running the training to completion irrespective of final average reward obtained.

You may edit `configs/p3_train_atari_linear.py` if you wish to play around with hyperparamete for improving performance of the linear Q-network on Pong-v0, or try another Atari environment changing the `env_name` hyperparameter. The list of all Gym Atari environments are available he https://gym.openai.com/envs/#atari (https://gym.openai.com/envs/#atari)

Deliverable 3 (2 points)

Run the following block of code to train a linear Q-network on Atari Pong-v0. We don't expect the linear Q-Network to learn anything meaingful so full credit will be given for simply running this training to completion (without errors), irrespective of the final average reward.

In [29]:
```
from configs.p3_train_atari_linear import config as config_lina

# make env
env = gym.make(config_lina.env_name)
env = MaxAndSkipEnv(env, skip=config_lina.skip_frame)
env = PreproWrapper(env, prepro=greyscale, shape=(80, 80, 1),
                    overwrite_render=config_lina.overwrite_render)

# exploration strategy
exp_schedule = LinearExploration(env, config_lina.eps_begin,
        config_lina.eps_end, config_lina.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_lina.lr_begin, config_lina.lr_end,
        config_lina.lr_nsteps)

# train model
model = DQNTrain(LinearQNet, env, config_lina, device)
print("Linear Q-Net Architecture:\n", model.q_net)
model.run(exp_schedule, lr_schedule)
```

```
Evaluating...

Linear Q-Net Architecture:
 LinearQNet(
  (fullCnct): Linear(in_features=25600, out_features=6, bias=True)
)

Average reward: -21.00 +/- 0.00

250001/500000 [==============>...............] - ETA: 1160s - Loss: 2.1385 - Avg_
20.4800 - Max_R: -18.0000 - eps: 0.7750 - Grads: 282.3471 - Max_Q: 8.2338 - lr: 0
1

Evaluating...



Average reward: -20.92 +/- 0.05

500001/500000 [==============================] - 2370s - Loss: 0.9164 - Avg_R: -2
00 - Max_R: -19.0000 - eps: 0.5500 - Grads: 173.4326 - Max_Q: 8.5056 - lr: 0.0001

- Training done.
Evaluating...



Average reward: -20.98 +/- 0.02
```

Part 4: Playing Atari Games from Pixels - using Deep Q-Networks

This part is extra credit and worth 5 bonus points. We will now train our deep Q-Network from Pa
on Pong-v0.

Again, no additional implementation is required but you may wish to tweak your CNN architectur `cnn_qnet.py` and hyperparameters in `configs/p4_train_atari_cnn.py` (however, evaluation v be considered at no farther than the default 5 million steps, so you are not allowed to train for longer). Please note that this training may take a very long time (we tested this on a single GPU it took around 6 hours).

The bonus points for this question will be allotted based on the best evaluation average reward (EAR) before 5 million time stpes:

1. EAR >= 0.0 : 4/4 points
2. EAR >= -5.0 : 3/4 points
3. EAR >= -10.0 : 3/4 points
4. EAR >= -15.0 : 1/4 points

Deliverable 4: (2 points. Extra Credit for both CS4803 and CS7643)

Run the following block of code to train your DQN:

In [ ]:

```
from configs.p4_train_atari_cnn import config as config_cnna


# make env
env = gym.make(config_cnna.env_name)
env = MaxAndSkipEnv(env, skip=config_cnna.skip_frame)
env = PreproWrapper(env, prepro=greyscale, shape=(80, 80, 1),
                    overwrite_render=config_cnna.overwrite_render)

# exploration strategy
exp_schedule = LinearExploration(env, config_cnna.eps_begin,
        config_cnna.eps_end, config_cnna.eps_nsteps)

# learning rate schedule
lr_schedule  = LinearSchedule(config_cnna.lr_begin, config_cnna.lr_end,
        config_cnna.lr_nsteps)

# train model
model = DQNTrain(ConvQNet, env, config_cnna, device)
print("CNN Q-Net Architecture:\n", model.q_net)
model.run(exp_schedule, lr_schedule)
```