Lab 3: Deep Q-Learning

Deep Reinforcement Learning Bootcamp

August 26-27, Berkeley CA

1 Introduction

In lab 3, you will implement Deep Q Networks (DQN) [1].

2 Environment Setup

You should have your environment set up as specified in the pre-lab setup instructions.

3 DQN

3.1 Background

This section explains the basics of DQN. The goal of a DQN agent is to maximize the future discounted return at each timestep t, namely

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}, \tag{1}$$

assuming the environment episode ends at timestep T. The optimal action-value function $Q^*(s, a)$ defines the maximum discounted return achievable when following an optimal policy π^*, thus

$$Q^*(s, a) = \max_{\pi} \mathbb{E}_{\pi} \left[R_t | s_t = s, a_t = a \right]. \tag{2}$$

This optimal action-value function defines a recursive relationship. In other words, the optimal Q-value function conforms to the Bellman optimality equation

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{S}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]. \tag{3}$$

Generally, we can estimate this optimal Q-function by updating the Q-value function in an iterative fashion as

$$Q_{i+1}(s, a) = \mathbb{E}_{s' \sim \mathcal{S}} \left[r + \gamma \max_{a'} Q_i(s', a') | s, a \right], \tag{4}$$

Algorithm 1: DQN

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode 1 to M **do**
  for t from 1 to T **do**
    With probability ϵ select random action a_t otherwise select $a_t = \max_a Q^*(\phi(s_t), a, ; \theta)$
    Execute action a_t in emulator and observe reward r_t, state s_{t+1}, and episode termination signal d_t
    Set $s_{t+1} = s_t$, a_t, d_t and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1}, d_t)$ in $\mathcal{D}$
    Sample minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1}, d_j)$ from $\mathcal{D}$
    Set

$$y_j = \begin{cases} r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta') & \text{for non-terminal transition} \\ r_j & \text{for terminal transition} \end{cases}$$

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    Every C steps, update target network $\theta' \leftarrow \theta$
  end
end

which ultimately converges to Q^* as i goes to infinity. In DQN we use a function approximator to represent the Q-value function, rather than a lookup table. This function $Q(s, a; \theta)$ is trained to approximate $Q^*(s, a)$ over time using a loss function defined as

$$\mathbb{E}_{s,a,s'} \left[(y - Q(s, a; \theta))^2 \right], \tag{5}$$

in which

$$y = r + \gamma \max_{a'} Q(s', a'; \theta') \tag{6}$$

where the parameters θ' are updated at specific interval. The network $Q(s, a; \theta')$ is called the target network.

DQN is inherently off-policy, which means that we can update the agent towards the goal behavior through using data that is sampled from arbitrary behavior. Therefore, all sampled (s, a, s', r) tuples are stored in a replay buffer. At specific intervals, we sample from this replay buffer in order to update the aforementioned loss function. The complete algorithm is given in Algorithm 1.

3.2 Implementing the DQN Update

First we will implement DQN's update and test our implementation on a simple gridworld environment. If you try running

```
./docker_run.sh simpledqn/main.py GridWorld-v0
```

you will notice that it crashes with "Warning: test for `compute_q_learning_loss` didn't pass!".

In this exercise you will implement the computation of the Q-learning loss for a batch of sampled states, actions, rewards, next states, and termination signals according to

$$\mathbb{E}_{s,a,s'}\left[(y - Q(s,a;\theta))^2\right] \tag{7}$$

in which

$$y = \begin{cases} r + \gamma \max_{a'} Q(s',a';\theta') & \text{for non-terminal transition} \\ r & \text{for terminal transition} \end{cases} \tag{8}$$

In `simpledqn/main.py`, you should fill in `compute_q_learning_loss` with an implementation of the Q-learning loss, which takes in a batch of tuples (s, a, r, d, s') and returns a Chainer variable that contains the Q-learning loss[1]. The current Q network $Q(s,a;\theta)$ is stored as Python variable `self._q`, while the target Q network $Q(s,a;\theta')$ corresponds to the Python variable `self._qt`. Although the Q-function is defined as taking two arguments, namely s and a, the function is implemented as taking as input a single state s, while outputting a vector of Q-values, with each element representing the value for a particular action a.

After you implement this function, the test that previously failed should now pass with "Test for `compute_q_learning_loss` passed!".

During any point of training, you can visualize your policy by running the following command in a separate terminal:

```
./docker_run.sh simpledqn/main.py --render True GridWorld-v0
```

You can also visualize the learning curve by running the following command:

```
./docker_run.sh viskit/frontend.py data/local/dqn_gridworld
```

You should see a line similar to the following:

```
Done! View http://localhost:5000 in your browser
```

Follow the instruction and visit the address in your browser.

If your implementation is correct, you should observe steady improvement in the `AverageReturn` metric, achieving an average return of 1 after about 10 iterations, similar to Figure 1.

You can toggle the "Y-Axis Attribute" to inspect other logged quantities.

Explanation of the various logged quantities:

- `Steps`: The number of environment steps taken so far.

[1] Hint: make use of the chainer mathematical functions as described in https://docs.chainer.org/en/stable/reference/functions.html#mathematical-functions.

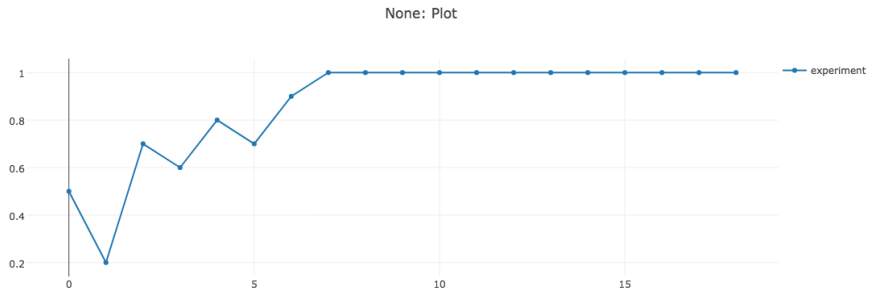

Figure 1: GridWorld-v0 learning curve, return over time

- `Epsilon`: The ϵ-value in the ϵ-greedy action selection regime.
- `Episodes`: The number of environment episodes finished so far.
- `AverageReturn`: The return averaged over the last 10 episodes
- `AverageDiscountedReturn`: The discounted return averaged over the last 10 episodes.
- `TDError^2`: The squared Bellman error (Q-learning loss) averaged over the last 50 Q-function loss updates.

3.3 Implementing the Double DQN Update

Next we will implement a popular variant of DQN: Double DQN. Double DQN reduces the over-estimation problem in regular Q-learning and improves performance [2].

We can switch on the Double DQN mode by turning on a command-line flag:

```
./docker_run.sh simpledqn/main.py --double True GridWorld-v0
```

If you run this command, you will find that it crashes with "Warning: test for `compute_double_q_learning_loss` didn't pass!" In this exercise, we will fill in the function `compute_double_q_learning_loss` to implement the computation of the Double DQN learning loss:

$$\mathbb{E}_{s,a,s'}\left[(y - Q(s,a;\theta))^2\right] \tag{9}$$

in which

$$y = \begin{cases} r + \gamma Q(s', \arg\max_{a'} Q(s',a';\theta);\theta') & \text{for non-terminal transition} \\ r & \text{for terminal transition} \end{cases} \tag{10}$$

where again, θ' is the parameter of the target network. After you implement this function, the test that previously failed should now pass with "Test for `compute_double_q_learning_loss` passed!". Although double Q-learning is more stable, its performance gain on this simple Gridworld problem is not obvious.

3.4 Learning Pong with RAM Observation

Now we will test our double DQN implementation on Atari Pong. Since training DQN with an observation space of raw pixels requires significant computational resources, we will use the emulator's memory state, RAM, as input to our Q network.

Run the Atari game Pong via

```
./docker_run.sh simpledqn/main.py --double True Pong-ram-v0
```

The launcher for Pong-ram-v0 automatically warm-starts the learning process with a filled replay buffer and a set of trained Q-value function weights to speed up learning. The AverageReturn should go from approximately -18 to approximately -11 in 50 iterations, but you should start to see learning progress in the first couple minutes (see Figure 2). Try different hyperparameters to see if you can make it learn faster!

Plot the learning data via

```
./docker_run.sh viskit/frontend.py data/local/dqn_pong
```

Visualize the learned behavior (the agent on the right is controlled by DQN):

```
./docker_run.sh simpledqn/main.py --render True Pong-ram-v0
```

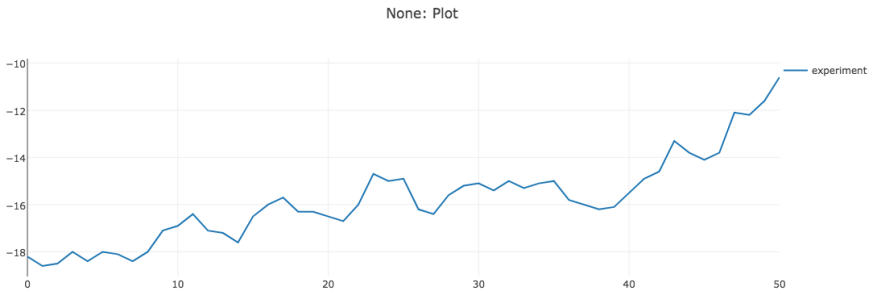

Figure 2: Pong-ram-v0 learning curve, return over time

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016.