Lab 6

TA: 何國豪、**鄭余玄**

11/15 12:00

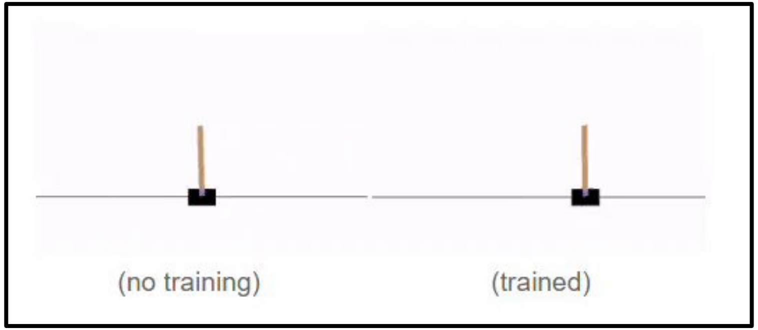
Lab6 Deadline

no demo

Outline

1. Solve CartPole-v1 using DQN
2. Solve Pendulum-v0 using DDPG
3. Modify and Run Sample Code
4. Scoring Criteria
5. Reminders
6. Sample Code Guides

1. CartPole-v1



- Observation [4]
  - Cart Position
  - Cart Velocity
  - Pole Angle
  - Pole Velocity at Tip
- Action [2]
  - Left
  - Right
- Reward
  - **+1** for every time step
- Starting State
  - All observations are assigned a uniform random value between ±0.05
- Episode Termination
  - Pole Angle is more than ±12°
  - Center of the cart reaches the edge of the display
  - Episode length is greater than **500**

Deep Q-Network (DQN)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $Q = Q$

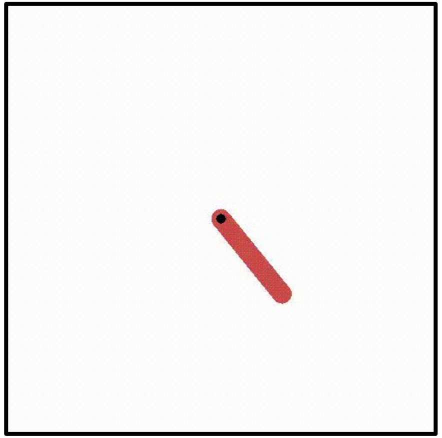
  End For

End For

TODO:

- Construct the neural network
- Select action according to epsilon-greedy
- Construct Q-values and target Q-values
- Calculate loss function
- Update behavior and target network
- Understand deep Q-learning mechanisms

2. Pendulum-v0



- Observation [3]
  - cos(theta) (Angle)
  - sin(theta) (Angle)
  - theta_dot (Angular Velocity)
- Action [1]
  - Joint Effort
- Reward
  - In essence, the goal is to remain at zero angle (vertical), with the least rotational velocity, and the least effort.
- Starting State
  - Random angle from -pi to pi, and random velocity between -1 and 1
- Episode Termination
  - Episode length is greater than 200

Deep Deterministic Policy Gradient (DDPG)

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

TODO:

- Construct neural networks of both actor and critic
- Select action according to the actor and the exploration noise
- Update critic
- Update actor
- Update target network softly
- Understand the mechanism of actor-critic

3. Modify Sample Code

1. Find a `#TODO` comment with hints
2. remove the `raise NotImplementedError`

3. Run Sample Code

- Simply train and test: `python dqn.py`
- Only test and render: `python dqn.py --restore --render`
- Help message: `python dqn.py --help`
- Other usages:
  - Save as different model name: `python dqn.py -m cart_model`
  - Load and test different model: `python dqn.py -m cart_model --restore`
  - Only run on cpu: `python dqn.py -d cpu`
  - Only run on 3rd gpu: `python dqn.py -d cuda:2`
  - Set episode length to 2000: `python dqn.py -e 2000`

4. Scoring Criteria

Show your work, otherwise no credit will be granted.

- Report (80%)
  - (DO **explain**; do not copy and paste your codes.)
- Report Bonus (10%)
  - Implement and perform experiments on **Double-DQN**. (5%)
  - **Extra experiments**; e.g., random process comparison (5%)
- Performance (20%)
  - [**CartPole-v1**] Average reward of 10 testing episodes: Average ÷ 5
  - [Pendulum-v0] Average reward of 10 testing episodes: (Average + 700) ÷ 5

5. Reminders

- Your network architecture and hyper-parameters can differ from the defaults.
- Be careful of the target q-value at the end of an episode in DQN.
- Ensure the shape of tensors all the time especially when calculating the loss.
- `with no_grad():` scope is the same as `xxx.detach()`
- Be aware of the indentation of hints.
- When testing DDPG, action selection need NOT include the noise.

Sample Code

Sample Code

- dqn-example.py
- ddpg-example.py

Replay Memory

```
class ReplayMemory:
 def __init__(self, capacity):
   self._buffer = deque(maxlen=capacity)

 def __len__(self):
   return len(self._buffer)

 def append(self, *transition):
   # (state, action, reward, next_state, done)
   self._buffer.append(tuple(map(tuple, transition)))

 def sample(self, batch_size=1):
   return random.sample(self._buffer, batch_size)
```

Usage in algorithm:

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

Greedy Action Selection

```python
def select_action(epsilon, state, action_dim=2):
 """epsilon-greedy based on behavior network"""
 ## TODO ##
 raise NotImplementedError
```

With probability ε select a random action a_t
otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

DQN

```python
class DQN(nn.Module):
 def __init__(self, state_dim=4, action_dim=2, hidden_dim=24):
   super().__init__()
   ## TODO ##
   raise NotImplementedError

 def forward(self, x):
   ## TODO ##
   raise NotImplementedError
```

Update Behavior Network

```python
def update_behavior_network():
    def transitions_to_tensors(transitions, device=args.device):
        """convert a batch of transitions to tensors"""
        return (torch.Tensor(x).to(device) for x in zip(*transitions))

    # sample a minibatch of transitions
    transitions = memory.sample(args.batch_size)
    state, action, reward, next_state, done = transitions_to_tensors(transitions)
    # TODO: loss
    # q_value = ?
    # with torch.no_grad():
    #    q_next = ?
    #    q_target = ?
    # loss = criterion(q_value, q_target)
    raise NotImplementedError
```

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

$$\text{Set } y_j = \begin{cases} \boxed{r_j \quad \text{if episode terminates at step } j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) \quad \text{otherwise} \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

Actor Action Selection

```
def select_action(state, low=-2, high=2):
  """based on the behavior (actor) network and exploration noise"""
  ## TODO ##
  # with torch.no_grad():
  #   action = ? + ?
  #   return max(min(action, high), low)
  raise NotImplementedError
```

Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

Random Process

```
class OrnsteinUhlenbeckProcess:
 """1-dimension Ornstein-Uhlenbeck process"""
 def sample(self, mu=0, std=.2, theta=.15, dt=1e-2, sqrt_dt=1e-1):
   self.x += theta * (mu - self.x) * dt + std * sqrt_dt * random.gauss(0, 1)
   return self.x

 def reset(self, x0=0):
   self.x = x0
```

used only during training

Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

References

1. Mnih, Volodymyr et al. “Playing Atari with Deep Reinforcement Learning.” ArXiv abs/1312.5602 (2013).
2. Mnih, Volodymyr et al. “Human-level control through deep reinforcement learning.” Nature 518 (2015): 529-533.
3. Van Hasselt, Hado, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning.” AAAI. 2016.
4. Lillicrap, Timothy P. et al. “Continuous control with deep reinforcement learning.” CoRR abs/1509.02971 (2015).
5. Silver, David et al. “Deterministic Policy Gradient Algorithms.” ICML (2014).
6. OpenAI. “OpenAI Gym Documentation.” Retrieved from Getting Started with Gym: https://gym.openai.com/docs/.
7. OpenAI. “OpenAI Wiki for Pendulum v0.” Retrieved from Github: https://github.com/openai/gym/wiki/Pendulum-v0.
8. PyTorch. “Reinforcement Learning (DQN) Tutorial.” Retrieved from PyTorch Tutorials: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.