Lab 6

TA: 何國豪、**鄭余玄**

12/09 12:00

Lab6 Deadline

no demo

In this lab,

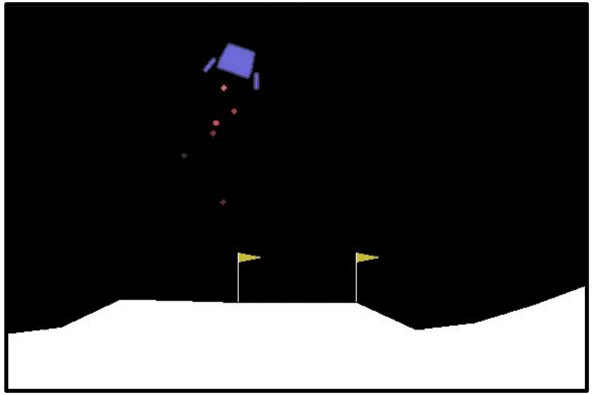
Must use sample code, otherwise no credit.

Outline

A. Specs
   1. Solve LunarLander-v2 using DQN
   2. Solve LunarLanderContinuous-v2 using DDPG
   3. Modify and Run Sample Code
   4. Scoring Criteria
B. Report
C. Sample Code

Specs

LunarLander-v2



- Observation [8]
  1. Horizontal Coordinate
  2. Vertical Coordinate
  3. Horizontal Speed
  4. Vertical Speed
  5. Angle
  6. Angle Speed
  7. If first leg has contact
  8. If second leg has contact
- Action [4]
  1. No-op
  2. Fire left engine
  3. Fire main engine
  4. Fire right engine

- Action [2] (Continuous)
  - Main engine: -1 to 0 off, 0 to +1 throttle from 50% to 100% power. Engine can't work with less than 50% power
  - Left-right: -1.0 to -0.5 fire left engine, +0.5 to +1.0 fire right engine, -0.5 to 0.5 off

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

NCTU CGI Lab

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

CGI Lab

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 --test_only --render`
- Help message: `python dqn.py --help`

4. Scoring Criteria

Show your work, otherwise no credit will be granted.

- Report (80%)
  - (DO explain; do not only copy and paste your codes.)
- Report Bonus (20%)
  - Implement and Experiment on Double-DQN (10%)
  - Extra hyperparameter tuning, e.g., Population Based Training. (10%)
- Performance (20%)
  - [LunarLander-v2] Average reward of 10 testing episodes: Average ÷ 30
  - [LunarLanderContinuous-v2] Average reward of 10 testing episodes: Average ÷ 30

4. Scoring Criteria

- Elaborate **YOUR** idea
  - any references should be cited
- 50% of score is counted for each game
  - that is, if you fail to train a game, your score is at most 50.

Report

Turn in:

1. Experiment report (.pdf)
2. Source code [NOT including model weights]

Notice: zip all files with name "DLP_LAB6_StudentId_Name.zip",

e.g.: 「DLP_LAB6_0856032_鄭余玄.zip」

```
DLP_LAB6_0856032_鄭余玄
├── dqn.py
├── ddpg.py
└── report.pdf
```

3. Email title: MTK_DLP_LAB6_StudentId_Name
4. Email to: chengscott.cs08g@nctu.edu.tw

(Wrong format deduction: -5pts; Multiple deductions may apply.)

4. Describe differences between your implementation and algorithms

- If your answer is ***no difference***, you must receive a zero score.
- The question is mainly asking for what techniques can help training in your implementation, and that technique is not mentioned in the algorithm explicitly.

5. Describe your implementation and the gradient of actor updating

1. explain the gradient of actor in the theory
2. explain your implementation of the gradient of actor
3. Is your implementation the same as in the theory? Why or why not?

Not an answer: I use PyTorch to perform gradient descent, so the result is the same.

Explain XXX of YYY

- Please focus on XXX rather than YYY. That is, you should explain more about XXX than describing YYY.

E.g.: Explain effects of the discount factor.

Not an answer: Discount factor is part of MDP, so it is important.

Performance (20%)

Please attach a screenshot of testing results.

Sample Code

Sample Code

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

dqn-example.py

`class ReplayMemory`	Replay memory (used to store transitions)
`class Net`	PyTorch NN
`class DQN`	Algorithm
`def train`	Training loop
`def test`	Testing loop
`def main`	Entry function

Replay Memory

```python
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

Epsilon-Greedy Action Selection

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

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

Neural Network

```
class DQN(nn.Module):
 def __init__(self, state_dim=8, action_dim=4, hidden_dim=32):
   super().__init__()
   ## TODO ##
   raise NotImplementedError

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

Update Behavior Network

```
def _update_behavior_network(self, gamma):
    # sample a minibatch of transitions
    state, action, reward, next_state, done = self._memory.sample(self.batch_size, self.device)
    ## TODO ##
    # 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

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

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

ddpg-example.py

`class GaussianNoise`	n-dim Gaussian Noise
`class ReplayMemory`	Replay memory (used to store transitions)
`class ActorNet`	PyTorch NN
`class CriticNet`	PyTorch NN
`class DDPG`	Algorithm
`def train`	Training loop
`def test`	Testing loop
`def main`	Entry function

Actor Action Selection

```
def select_action(self, state, noise=True):
 """based on the behavior (actor) network and exploration noise"""
 ## TODO ##
 # with torch.no_grad():
 #   action = ? + ?
 # return action
 raise NotImplementedError
```

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

Random Process

```python
class GaussianNoise:
  def __init__(self, dim, mu=None, std=None):
    self.mu = mu if mu else np.zeros(dim)
    self.std = std if std else np.ones(dim) * 0.1

  def sample(self):
    return np.random.normal(self.mu, self.std)
```

used only during training

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

Reminders

- Your network architecture and hyper-parameters can differ from the defaults.
- 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.

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.