Day 3. Deep Q-Network

NPEX Reinforcement Learning

2020.08.31

Jaeuk Shin

CORE
Control + Optimization Research Lab

Deep Q-Network - Review

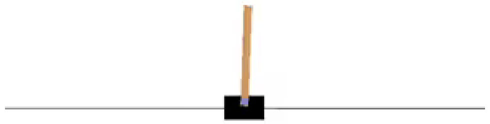
Deep Q-Network - Review

MDP $M = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$, where

state space $\mathcal{S}$ (possibly continuous)

action space $\mathcal{A} = \{0, \cdots, m-1\}$

transition probability P, reward function r, discount factor $\gamma \leq 1$



We will use **CartPole** environment for test purpose!

CORE
Control + Optimization Research Lab

Deep Q-Network - Review

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**

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

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

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

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

Why?

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

$\boxed{\text{Every } C \text{ steps reset } \hat{Q} = Q}$

Update target Q-network $(\theta^- \leftarrow \theta)$

End For

End For

Deep Q-Network - Review

Some Popular Variants of DQN:

Try this!(need little effort)

- Double DQN (van Hasselt, 2015)
- DQN with Prioritized Replay (Schaul, 2016)
- DQN with Dueling Architecture (Wang, 2016)
- C51(Bellemare, 2017)

Deep Q-Network - Implementation

Deep Q-Network - Implementation

What should our agent do? (see **dqn_agent.py**)

First sample $U \sim \mathscr{U}(0,1)$, and select an action as follows:

$$a_t \begin{cases} = \arg\max_a' Q(s_t, a; \theta) & \text{if } U \geq \epsilon \\ \sim \mathscr{U}(\mathcal{A}) & \text{if } U < \epsilon. \end{cases}$$

In other words, with probability ϵ, we sample a random action, and **greedy** action otherwise (ϵ-greedy).

During evaluation, ϵ is set to 0 (why?).

Deep Q-Network - Implementation

How to represent Q-function? (In discrete action case)

Given $\mathcal{A} = \{a_0, \cdots a_{m-1}\}$,

1. we may convert a_j into **one-hot vector** $\mathbf{e}_j$, and feed $(s, \mathbf{e}_j)$,
2. or, just let our network returns m-values as follows:

$$\mathbf{NN}(s;\theta) = (Q(s, a_0;\theta), Q(s, a_1;\theta), \cdots Q(s, a_{m-1};\theta))^\top.$$

see **dqn_model.py**

Deep Q-Network - Implementation

```python
def get_action(self, state, eps):

    self.Q.eval()
    dimS = self.dimS
    nA = self.nA

    s = torch.tensor(state, dtype=torch.float).view(1, dimS)

    q = self.Q(s)

    # simple implementation of \epsilon-greedy method
    if np.random.rand() < eps:
        a = np.random.randint(nA)
    else:
        # greedy selection
        a = np.argmax(q.cpu().data.numpy())

    return a
```

Remark. shape of input tensor?

Remark. numpy array to torch tensor, and vice versa?

```python
class Critic(nn.Module):
    """
    implementation of critic network Q(s, a)
    """
    def __init__(self, state_dim, num_action, hidden_size1, hidden_size2):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, num_action)


    def forward(self, state):
        # given a state s, the network returns a vector Q(s,) of length |A|
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        q = self.fc3(x)

        return q
```

Deep Q-Network - Train

Deep Q-Network - Train

1. Sample a batch $\{(s_i, a_i, r_i, s_i')\}_{i=1}^{N}$, and construct a loss as follows:

$$\frac{1}{N}\sum_{i=1}^{N}\|Q(s_i, a_i; \theta) - y_i\|^2$$

2. Perform one-step gradient (possible choice : Adam, RMSprop, etc.), and update Q-network

3. Update target Q-network

Deep Q-Network - Train

What do we need?

1. replay buffer to store samples collected
2. optimizer
3. target network

```python
# set networks
self.Q = Critic(dimS, nA, hidden_size1=hidden1, hidden_size2=hidden2)
self.target_Q = copy.deepcopy(self.Q)

# freeze the target network
for p in self.target_Q.parameters():
    p.requires_grad_(False)

self.optimizer = Adam(self.Q.parameters(), lr=lr)

self.gamma = gamma
self.tau = tau

self.buffer = ReplayBuffer(dimS, buffer_size)
self.batch_size = batch_size

self.render = render
```

Deep Q-Network - Train

```python
def train(self):

    self.Q.train()
    gamma = self.gamma
    batch = self.buffer.sample_batch(self.batch_size)

    # unroll batch
    with torch.no_grad():
        observations = torch.tensor(batch['state'], dtype=torch.float)
        actions = torch.tensor(batch['action'], dtype=torch.long)
        rewards = torch.tensor(batch['reward'], dtype=torch.float)
        next_observations = torch.tensor(batch['next_state'], dtype=torch.float)
        terminals = torch.tensor(batch['done'], dtype=torch.float)

        mask = 1.0 - terminals

        next_q = torch.unsqueeze(self.target_Q(next_observations).max(1)[0], 1)
        target = rewards + gamma * mask * next_q

    out = self.Q(observations).gather(1, actions)

    loss_ftn = MSELoss()
    loss = loss_ftn(out, target)

    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    self.target_update()

    return
```

CORE
Control + Optimization Research Lab

Deep Q-Network - Train

Given (s, a, s', r), the target is computed as follows:

$$y = \begin{cases} r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{if } s' : \text{terminal state} \\ r & \text{if } s' : \text{non-terminal state} \end{cases}$$

```
next_q = torch.unsqueeze(self.target_Q(next_observations).max(1)[0], 1)
target = rewards + gamma * mask * next_q
```

Shape of each tensor?

Compute $Q(s, a)$ on batch?

```
out = self.Q(observations).gather(1, actions)
```

Deep Q-Network - Implementation

Target Network Update: How?

```
self.target_update()
```

Two options for you:

- Hard Target Update
- Soft Target Update

perform this periodically: $\theta^- \leftarrow \theta$

```
def hard_target_update(self):
    # hard target update
    # this will not be used in our implementation
    self.target_Q.load_state_dict(self.Q.state_dict())
    return


def target_update(self):
    # soft target update
    # when \tau = 1, this is equivalent to hard target update
    for p, target_p in zip(self.Q.parameters(), self.target_Q.parameters()):
        target_p.data.copy_(self.tau * p.data + (1.0 - self.tau) * target_p.data)
    return
```

or in each step, $\theta^- \leftarrow (1-\tau)\theta^- + \tau\theta$

CORE
Control + Optimization Research Lab

Deep Q-Network - Test

see test.ipynb

Deep Q-Network - Plot

see plot.ipynb

Thank you!