DDQN(Double DQN)

Paper Link: Deep Reinforcement Learning with Double Q-learning

Key Features

- The Q-value of original Q-learning and DQN can be overestimated.
- DDQN is an algorithm that reduces the overestimation bias of DQN.
- DDQN updates the target value to $Y_t = R_{t+1} + \gamma Q(S_{t+1}, \arg\max Q(S_{t+1}, a; \theta_t); \theta_t^-)$
- DDQN can find a better policy without additional network structure or parameters in DQN.

Background

In DQN, there is a problem of overestimating the target value rather than the optimal value. DDQN shows that overestimation bias can actually negatively affect performance, and it is better to reduce it.

The problem with Overestimation bias

In Q-learning, the target value can be written as follows.

$$Y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t) \quad (1)$$

As presented in the Thrun and Schwartz (1993) paper, if it is an optimal action value Q, it can be written as

$$\sum_{a} \big(Q_t(s,a) - V_*(s)\big) = 0$$

but, in $Q(s,a)$, an error may occur due to noise in the environment, so it becomes as follows

$$\frac{1}{m}\sum_{a} \big(Q_t(s,a) - V_*(s)\big) = C$$
$$\max_a Q(s,a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}$$
$$\max_a Q(s,a) - V_*(s) \geq \sqrt{\frac{C}{m-1}}$$

For the above reasons, the bias of $\sqrt{\frac{C}{m-1}}$ or higher is continuously overestimated by the difference from optimal V when the target value is calculated using $\max Q(s,a)$ in Q-learning.In fact, the paper shows several results in which Q-learning is overestimation-biased compared to double Q-learning.

Method

Double Q-learning

In the van Hasselt, Double Q-learning (2010) algorithm, in order to prevent overestimation bias, action selection and action evaluation are decoupled into two value functions. For a clear comparison with Q-learning, the paper can first solve the selection and evaluation in Q-learning and rewrite the subject (1) as follows

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t)$$

The Double Q-learning error is written as follows.

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta'_t)$$

The above expression uses a second set of weights θ_t to fairly evaluate the value of the policy. This second set of weights can be updated symmetrically by switching the roles of θ and θ'.

Similarly, in DQN, overestimation can be reduced by decomposing the max operation in the target into action selection and action evaluation. In reference to both Double Q-learning and DQN, we refer to the resulting algorithm as Double DQN. Its update is same as DQN, but replacing the target Y_t^{DQN} with

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t^-)$$

In comparison to Double Q-learning, the weights of the second network θ' are replaced with the weights of the target network θ^- for the evaluation of the current greedy policy. The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network.

Algorithm

DDQN is the same except for Equation (17) that finds target y_j in the existing DQN algorithm.

Initialize replay memory D to capacity N (2)

Intialize action value function Q with random weights θ (3)

Initalize target action value function $\hat{Q}$ with weights $\theta^{-} = \theta$ (4)

For episode $= 1, M$ **do** (5)

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(\mathrm{s}_1)$ (6)

  For $t = 1,\ \mathrm{T}$ **do** (7)

    With probability ϵ select a random action a_t (8)

    otherwise select $a_t = argmaxQ(\phi(s_t), a; \theta)$ (9)

    Execute action a_t in emulator and observe reward r_t and image x_{t+1} (10)

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{\mathrm{t}+1} = \phi(\mathrm{s}_{\mathrm{t}+1})$ (11)

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D (12)

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D (13)

    If episode terminates at step $j + 1$ (14)

      Set $y_j = r_j$ (15)

    Else (16)

      Set $y_j = r_j + \gamma \hat{Q}(\phi_{j+1}, \arg\max Q(\phi_{j+1}, a'; \theta); \theta^{-})$ (17)

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ (18)

    Every C steps reset $\hat{\mathrm{Q}} = \mathrm{Q}$ (19)

  End for (20)

End for (21)

Implementation on JORLDY

- DDQN JORLDY Implementation

```
# DDQN is derived DQN and uses only the learn method overriding.

class Double(DQN):
  def __init__(self, **kwargs):
    super(Double, self).__init__(**kwargs)

  def learn(self):
    ...
```

- When calculating target y with DQN

```
# Set y = r + γmaxQ

def learn(self):

  ...

  with torch.no_grad():
    max_Q = torch.max(q).item()
    next_q = self.target_network(next_state)
    target_q = (
        reward + (1 - done) * self.gamma * next_q.max(1, keepdims=True).values
    )

  ...
```

- When calculating target y with Double-DQN

```
# Set y = r + γQ'(s,argmaxQ(s))

def learn(self):

  ...

  with torch.no_grad():
    max_Q = torch.max(q).item()
    next_q = self.network(next_state)
    max_a = torch.argmax(next_q, axis=1)
    max_one_hot_action = eye[max_a.view(-1).long()]

    next_target_q = self.target_network(next_state)
    target_q = reward + (next_target_q * max_one_hot_action).sum(
        1, keepdims=True
    ) * (self.gamma * (1 - done))

  ...
```

References

Relevant papers

- Issues in Using Function Approximation for Reinforcement Learning
- van Hasselt. Double Q-learning(2010)
- Human-level control through deep reinforcement learning