Q学习

一种异策略的时序查分学习算法。

时序查分学习算法与蒙特卡洛方法不同：

蒙特卡洛需要完整一个路径完成才能知道总回报，不依赖马尔科夫性质。

而时序查分学习只需要一步，总回报需要依赖马尔科夫性质来进行近似估计。

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big),$$

深度Q网络

为了在连续的状态和动作空间中计算值函数$Q^{\pi}(s,a)$,可以用

$$Q_{\phi}(\mathbf{s},\mathbf{a}) \approx Q^{\pi}(s,a),$$

来近似计算，称为值函数近似。

这个函数通常为一个神经网络，输出为一个实数。

$$Q_{\phi}(\mathbf{s}) = \begin{bmatrix} Q_{\phi}(\mathbf{s},a_1) \\ \vdots \\ Q_{\phi}(\mathbf{s},a_m) \end{bmatrix} \approx \begin{bmatrix} Q^{\pi}(s,a_1) \\ \vdots \\ Q^{\pi}(s,a_m) \end{bmatrix}.$$

所以网络要学习参数来使得函数$Q_{\phi}(s,a)$ 可以逼近值函数$Q^{\pi}(s,a)$

如果用蒙特卡洛方法，直接让$\mathbb{Q}_{\phi}(s,a)$ 去逼近平均的总回报$Q^{\pi}(s,a)$

如果用时序查分，让$\mathbb{Q}_{\phi}(s,a)$ 去逼近$E_{s',a'}[r + \gamma Q_{\phi}(s',a')]$

以 Q 学习为例，采用随机梯度下降，目标函数为

$$\mathcal{L}(s,a,s'|\phi) = \left(r + \gamma \max_{a'} Q_\phi(\mathbf{s}',\mathbf{a}') - Q_\phi(\mathbf{s},\mathbf{a})\right)^2,$$

目标函数存在的问题：

1.参数学习的目标依赖于参数本身，目标不稳定

2.样本之间有很强的相关性

深度**Q**网络

1.目标网络冻结，一个时间段内固定目标中的参数，来稳定学习目标

2.经验回放，构建一个经验池来去除数据相关性。经验池是智能体最近的经历组成的数据集

训练的时候，随机从经验池中抽取样本来代替当前的样本用来进行训练，这样可以打破和相邻训练样本的相似性。

经验回放在一定程度上类似于监督学习，先手机样本，然后在这些样本上进行训练。

算法 14.5: 带经验回放的深度Q网络

输入: 状态空间$\mathcal{S}$，动作空间$\mathcal{A}$，折扣率γ，学习率α

初始化经验池$\mathcal{D}$，容量为N;

随机初始化Q网络的参数ϕ;

随机初始化目标Q网络的参数$\hat{\phi} = \phi$;

repeat

  初始化起始状态s;

  repeat

    在状态s，选择动作$a = \pi^{\epsilon}$;

    执行动作a，观测环境，得到即时奖励r和新的状态s';

    将s, a, r, s'放入$\mathcal{D}$中;

    从$\mathcal{D}$中采样ss, aa, rr, ss';

$$y = \begin{cases} rr, & ss' \text{为终止状态}, \\ rr + \gamma \max_{a'} Q_{\hat{\phi}}(\mathbf{ss}', \mathbf{a}'), & \text{否则} \end{cases};$$

    以$\left(y - Q_{\phi}(\mathbf{ss}, \mathbf{aa})\right)^2$为损失函数来训练$Q$网络;

    $s \leftarrow s'$;

    每隔C步，$\hat{\phi} \leftarrow \phi$;

  until s为终止状态;

until $\forall s, a$，$Q_{\phi}(\mathbf{s}, \mathbf{a})$收敛;

输出: Q网络$Q_{\phi}(\mathbf{s}, \mathbf{a})$

代码

Q网络

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64,
fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
```

```
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

对应于上述算法的学习代码

```
   def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done)
 tuples
            gamma (float): discount factor
        """

        # 从D中采样
        states, actions, rewards, next_states, dones = experiences
        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)
 [0].unsqueeze(1)

        # 如果是终止状态，则Q_next 为rewards
        Q_next = rewards + (gamma * Q_targets_next * (1-dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # 平方误差
        loss = F.mse_loss(Q_next,Q_expected)
        #清0
        self.optimizer.zero_grad()
        #反向传播
        loss.backward()
        #更新参数
        self.optimizer.step()
```

```
        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

完整代码

```
import numpy as np
import random
from collections import namedtuple, deque

from model import QNetwork

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size,
seed).to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
```

```
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and
learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done)
tuples
            gamma (float): discount factor
        """

        # 从D中采样
        states, actions, rewards, next_states, dones = experiences
        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)
[0].unsqueeze(1)
```

```
        # 如果是终止状态，则Q_next 为rewards
        Q_next = rewards + (gamma * Q_targets_next * (1-dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # 平方误差
        loss = F.mse_loss(Q_next,Q_expected)
        #清0
        self.optimizer.zero_grad()
        #反向传播
        loss.backward()
        #更新参数
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)


    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target

        Params
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(),
local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.0-
tau)*target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
```

```
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state",
"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e
is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e
is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e
is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in
experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is
not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```