Bellman Equations

Reward

$$R_t = \sum_{k=0}^{\inf} \gamma^k r_{t+k+1}$$

Policy

Policy is a function Π(s, a) of the state and the current action. It returns the probability of taking action a in state s.

State value function

$$V^{\pi}(s) = E_{\pi}[R_t | s_t = s]$$

It is the expected return when starting from state s according to policy π

Action value function

$$Q^{\pi}(s, a) = E_{\pi}[R_t | s_t = s, a_t = a]$$

It is the expected return given s and a under π

Bellman equation for state value function

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s^{'}} P^{a}_{ss^{'}} [R^{a}_{ss^{'}} + \gamma V^{\pi}(s^{'})]$$

Bellman equation for action value function

$$Q^{\pi}(s, a) = \sum_{s^{'}} P^{a}_{ss^{'}} [R^{a}_{ss^{'}} + \gamma \sum_{a^{'}} \pi(s^{'}, a^{'} Q^{\pi}(s^{'}, a^{'}))]$$

Training algorithms

Q-learning

Deterministic Bellman

$$Q(s,a) = r + \gamma * max(Q(s',a'))$$

```
# Randomize current QValues
rand_qvals = Q[state] + torch.rand(1,number_of_actions)/1000
# Get an action given the current QValues
action = torch.max(rand_qvals, 1)[1][0].item()
# Produce a new state given the chosen action
new_state, reward, done, info = env.step(action)
# Update QValues given current reward and QValue
Q[state, action] = reward + gamma * torch.max(Q[new_state])
state = new_state
```

Stochastic Q-Learning

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma * max(Q(s',a'))]$$

```
rand_qvals = Q[state] + torch.rand(1,number_of_actions)/1000
action = torch.max(rand_qvals, 1)[1][0].item()
new_state, reward, done, info = env.step(action)
Q[state, action] = (1 - learning_rate) * Q[state, action] + learning_rate *
(reward + gamma * torch.max(Q[new_state]))
state = new_state
```

E-Greedy decay

```
random_for_egreedy = torch.rand(1)[0].item()
if random_for_egreedy > egreedy:
    rand_qvals = Q[state] + torch.rand(1,number_of_actions)/1000
    action = torch.max(rand_qvals, 1)[1][0].item()
else:
    action = env.action_space.sample()

if egreedy > egreedy_final:
    egreedy *= egreedy_decay

new_state, reward, done, info = env.step(action)
Q[state, action] = reward + gamma * torch.max(Q[new_state])
state = new_state
```

Gradient descent - Neural Network

$$Q(s,a) = f(s,a)$$

f(.) is approximated using a neural network

Standard optimization

```
def optimize(self, state, action, new_state, reward, done):
    state = torch.Tensor(state).to(device)
    new_state = torch.Tensor(new_state).to(device)
    reward = torch.Tensor([reward]).to(device)

    if done:
        target_value = reward
    else:
        new_state_values = self.nn(new_state).detach()
        max_new_state_values = torch.max(new_state_values)
        target_value = reward + gamma * max_new_state_values

    predicted_value = self.nn(state)[action]
    loss = self.loss_func(predicted_value, target_value)
    self.optimizer.zero_grad()
    loss.backward()
        self.optimizer.step()
```

Experience replay

```
def optimize(self):

    if len(memory) < batch_size:
        return

    state, action, new_state, reward, done = memory.sample(batch_size)
    state = torch.Tensor(state).to(device)
    new_state = torch.Tensor(new_state).to(device)
    reward = torch.Tensor(reward).to(device)
    action = torch.LongTensor(action).to(device)
    done = torch.Tensor(done).to(device)

    new_state_values = self.nn(new_state).detach()
    max_new_state_values = torch.max(new_state_values, 1)[0]
    target_value = reward + (1 - done) * gamma * max_new_state_values

    predicted_value = self.nn(state).gather(1, action.unsqueeze(1)).squeeze(1)
    loss = self.loss_func(predicted_value, target_value)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Target Net

```
def optimize(self):

    if len(memory) < batch_size:
        return

    state, action, new_state, reward, done = memory.sample(batch_size)
    state = torch.Tensor(state).to(device)
    new_state = torch.Tensor(new_state).to(device)
    reward = torch.Tensor(reward).to(device)
    action = torch.LongTensor(action).to(device)
    done = torch.Tensor(done).to(device)

    new_state_values = self.target_nn(new_state).detach()
    max_new_state_values = torch.max(new_state_values, 1)[0]
    target_value = reward + (1 - done) * gamma * max_new_state_values

    predicted_value = self.nn(state).gather(1, action.unsqueeze(1)).squeeze(1)
```

```
        loss = self.loss_func(predicted_value, target_value)
        self.optimizer.zero_grad()
        loss.backward()
        if clip_error:
            for param in self.nn.parameters():
                param.grad.data.clamp_(-1,1)
                self.optimizer.step()

                if self.update_target_counter % update_target_frequency == 0:
                    self.target_nn.load_state_dict(self.nn.state_dict())

                    self.update_target_counter + 1
```