ELEC-8900-57 Special Topic - Machine Learning

Reinforcement Learning: Policy Gradient methods

Presented by:

Saad Ahmed Salim

Ashab Uddin

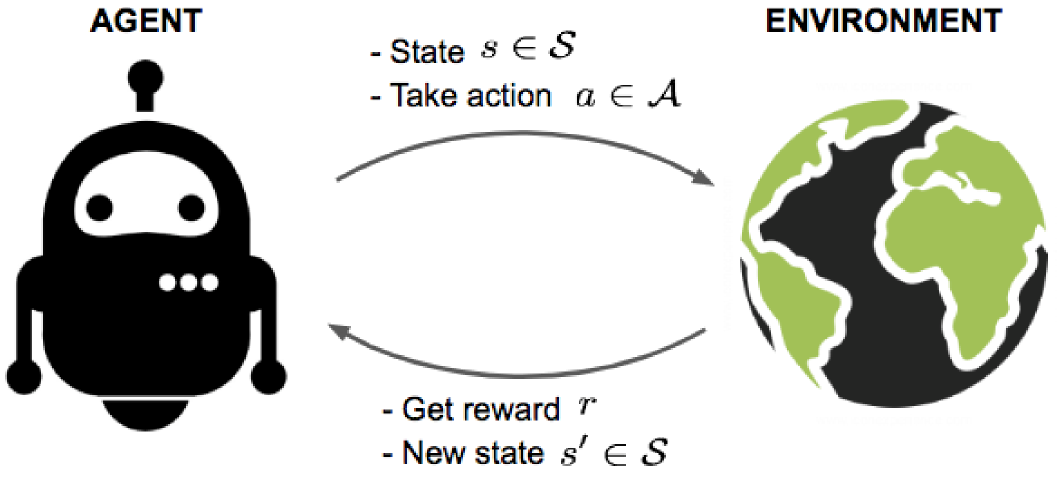
Instructor: Dr. Yasser Alginahi

Date: 17 November 2023

University of Windsor

What is Reinforcement Learning (RL)?

- An agent interacts with the environment, trying to take smart actions according to current state to maximize cumulative rewards.



https://lilianweng.github.io/posts/2018-02-19-rl-overview/?fbclid=

Key Concepts

- To determine the optimal policy to maximize long term reward

How it works

Don’t know anything about the path.

Goal is to find safe path.

No info before, learn from actions.

Reward & Penalty.

Comparison among Supervised, Unsupervised Learning and RL

Supervised	Unsupervised	Reinforcement
Labeled Data	Unlabeled data	An environment
Take actions using data	No clue about data. Take steps to determine the answer	No information before, learn from action
Photo identification, CT scan to detect tumor	Online shop product recommendation, Credit card fraud detection	Self Driving car, Gaming Bot

Markov Property and Markov Process

- Markov Process : A stochastic process has Markov property if conditional probability distribution of future states of process depends only upon present state and not on the sequence of events that preceded. Markov Decision Process: A Markov decision process (MDP) is a discrete time stochastic control process [2].

Definition

A state S_t is *Markov* if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

Definition

A *Markov Process* (or *Markov Chain*) is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$

MDP

A Markov decision process (MDP) is a Markov reward process with decisions. It is an *environment* in which all states are Markov.

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}_{ss'}^{a} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}_s^a = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$
- γ is a discount factor $\gamma \in [0, 1]$.

https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf

Policy

- policy determines how the agent behaves from a specific state.
- There are two types of policies: deterministic policy and stochastic policy.

Deterministic policy

- The deterministic policy output an action with probability one. For instance, In a car driving scenario, consider we have three actions: turn left, go straight, and turn right. The RL agent with deterministic policy always outputs one of the actions with probability 1.

Stochastic policy

- Stochastic policy output the probability distribution over the actions from states. For instance, consider an action steering angle of vehicle: The output of the policy will be a probability distribution with mean and standard deviation say (5,10).

Value Function

- The value function, denoted as *V*(*s*), represents the expected cumulative reward of being in a particular state *s* and following a certain policy. It quantifies how good it is for an agent to be in a specific state. The formal definition is as follows:

$$V(s) = E[\sum_{k=0}^{\infty} \gamma^t R_t |\, s_t = s]$$

- Where γ is discounted factor and R_t is reward in time steps t.

Q-Value Function

- The Q-value function, denoted as Q(s, a), represents the expected cumulative reward of being in state s, taking action a, and then following a certain policy. It reflects the quality of taking an action in a specific state. The formal definition is as follows:

$$Q(s,a) = E[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_t = s, a_t = a]$$

- Where E is expected value operator , a_t is action and s_t is state at time step t.

Bellman Equation for Value Function

- Bellman Expectation Equation for V-function:
- This equation expresses the expected value of being in state s as the sum of the immediate rewardR_{t+1} and the discounted expected value of the next state s_{t+1}, where γ is the discount factor.

$$V(s) = E[r_t + \gamma V(s_{t+1}) | \mathrm{s_t} = \mathrm{s}]$$

Optimal Value Function:

$$V^*(s) = \max_a E[r_t + \gamma V^*(s_{t+1}) | s_t = s$$

Bellman Equation for Q function

- Bellman Optimality Equation for Q-function:
- This equation expresses the optimal Q-value for taking action a in state s as the sum of the immediate reward R_{t+1} and the discounted maximum Q-value of the next state s_{t+1} over all possible actions a'.

$$\mathrm{Q}(s,a) = E[r_t + \gamma \max_{a'} Q(s_{t+1}, a_{t+1}) | \mathrm{s_t = s, a_t = a}]]$$

Optimal Q Function:

$$\mathrm{Q}^*(s,a) = E[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a_{t+1}) | \mathrm{s_t = s, a_t = a}]]$$

Limitation of Value Based Method

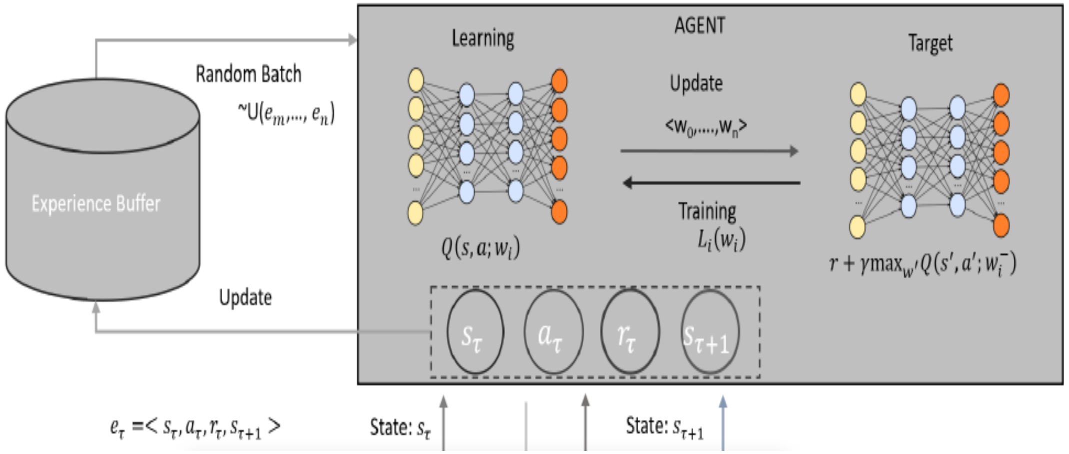
- Difficulty in Handling Continuous Action Spaces
- Overestimation Bias
- Exploration Challenges
- Lack of Convergence Guarantees
- Suboptimal Policies for Stochastic Environments

Advantage of Policy Gradient method

- Better convergence properties
- Effective in high-dimensional or continuous action spaces
- Learning stochastic policies
- Exploration Increase

What we did in DQN [9]

- Action is generated from Learning Net from Q Estimated Q value for each action by : Q_{argmax} or random by epsilon greedy method
- Action goes to environment , get reward and next state
- Create Buffer of s, a ,r, s’, a’
- After Batch Formation: start Training,
- In training , Loss Function: Q_(Estimated) -Q_(target)
- $Q_{target} = r + \gamma Q_{max,a'}(s'a')$



Environment: s, a, r, s’

Policy Gradient Method

- Estimate the probability of action for a given state that will give a long-term reward for the whole trajectory. When an agent follows that actions in each time step that is called policy , that can be defined as $\pi_\theta(\mathrm{s}, a)$ dependent on parameters θ called policy parameters. $\pi_\theta(\mathrm{s}, a)$ is called stochastic policy because it indicates the probability of taking an action a being in state s.

Policy Gradient Mathematics

- $J(\theta) = V^{\pi}(s) = \sum_{a} Q^{\pi}(s, a) * \pi(a|s)$
- Here π is probability distribution of action parameterized by θ [6]

Policy Gradient Mathematics

From Ref[6], Policy Gradient Theorem states that

$$\Delta_\theta J(\theta) = \sum_{s\epsilon S} d^{\pi}(s) \sum_{a\epsilon A} \Delta_\theta (\pi_\theta(a|s) * Q^{\pi}(s,a))$$

$$\Delta_\theta J(\theta) = E_{\pi}[Q^{\pi}(s,a) * \Delta_\theta \ln \pi_\theta(a|s)]$$

University of Windsor

Policy Gradient Mathematics

- $d^{\pi}(s)$ is stationary probability distribution.

That satisfies the following Balance equation:

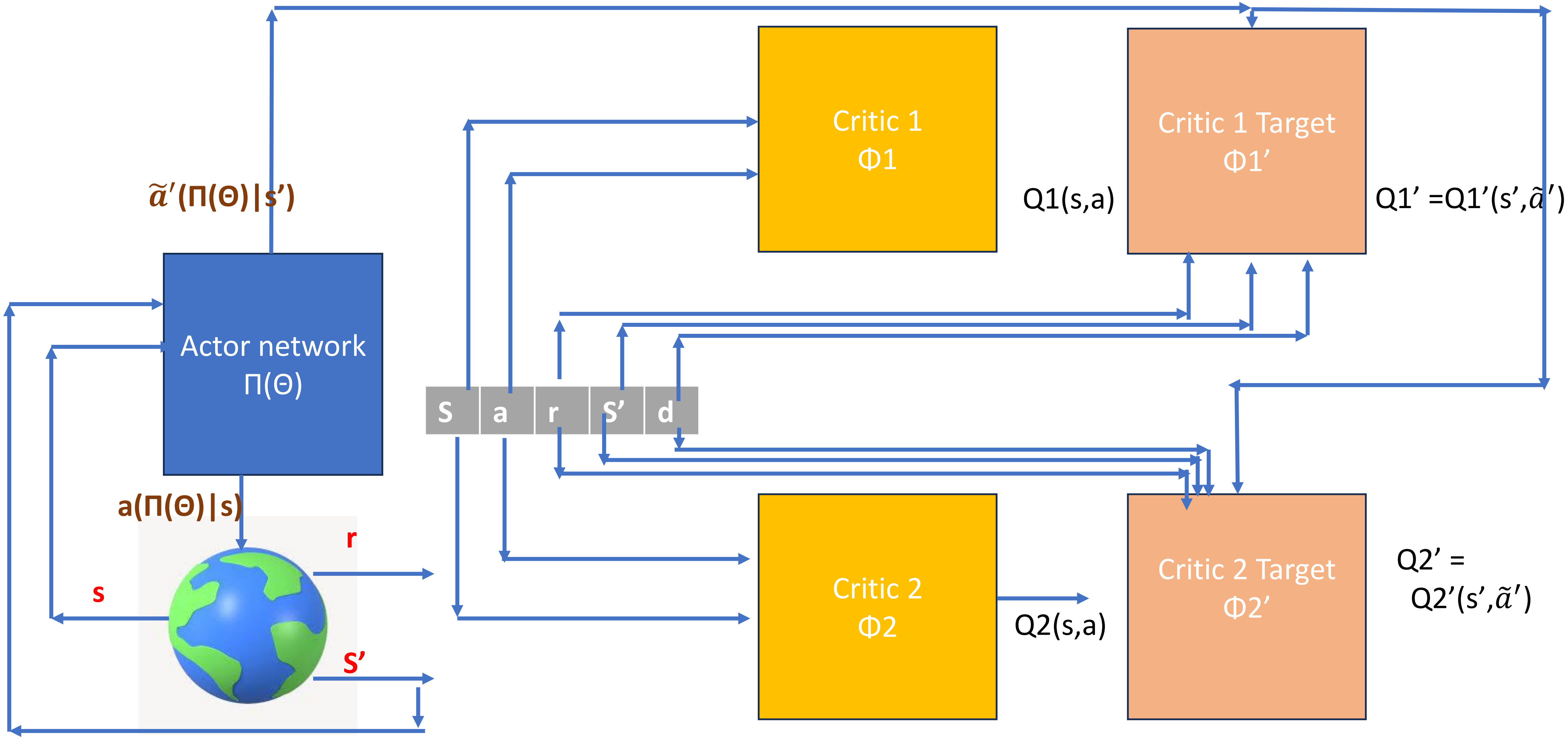
$d^{\pi}(i) * P(i \leftarrow j) = d^{\pi}(j) * P(j \leftarrow i)$.

After training and convergence, the $d^{\pi}(s)$ becomes stable.

Policy Gradient: Soft Actor Critic Method

- Entropy : how random a random variable is. If a coin is weighted so that it almost always comes up heads, it has low entropy; if it's evenly weighted and has a half chance of either outcome, it has high entropy.
- For example, in stochastic policy-based actor network : in state s1, a1(μ,σ) =(0,10) and in s2 , a2 (μ,σ) =(0,20). Which action will have more variance in sampling:
- Action a2 will have more exploring capability to find action because the standard deviation is 20.

Soft Actor Critic Network Architecture



University of Windsor

Soft Actor Critic Reward

Now Reward=reward + Entrophy term [7]

- $V^{\pi}(s_t) = E_{a_t \sim \pi_\theta(s_t)}\,[Q_\emptyset\,(s_t, a_t)]$+Entrophy Term

For policy π

- $V(s_t) = E_{a \sim \pi_\theta(s_t)}\,[Q_\emptyset\,(s_t, a_t) + \alpha H(\pi_\theta(.|s_t)] = E_{a \sim \pi_\theta(s_t)}\,[Q_\emptyset\,(s_t, a_t) - \alpha log\,(\pi_\theta(.|s_t)]$

Action Generation and Buffer Formation

- Action is generated from actor network and give a stochastic action (mean, standard deviation) parametrized by θ, Then get reward , next state and next action ..
- Create a buffer of individual batch of (s, a, r) and (s', a')
- Then Start Training

Critic Update

- The cumulative Target Q derived from network using min operator two target Q:

As our object is target should be lower as possible to control the overestimate of Q

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\emptyset_{targ,i}}(s', \widetilde{a}') - \ \alpha \log \pi_\theta(\widetilde{a}', s')),\right), \widetilde{a}' \sim (\pi_\theta(\ \cdot | \ s')$$

From Bellman Equation $Q(s_t, a_t) = r + \gamma(1-d)V'(s_{t+1})$

- Loss Calculation: $\frac{1}{|B|} * \sum_{(s,a,r,s',s)\epsilon\ B}^{n} (Q_{\emptyset_i}(s, a) - y(r, s', d))$^2 for i=1,2

- Φ1 and Φ2 update: as loss minimizing, gradient descent is required:

$$\Delta_{\emptyset_i} \frac{1}{|B|} * \sum_{(s,a,r,s',s)\epsilon\ B}^{n} (Q_{\emptyset_i}(s, a) - y(r, s', d))\text{^2 for i=1,2}$$

Actor Update

- Expected Reward Calculation: This focuses on minimizing overestimate of Q as well as randomness of policy

$$: \quad \frac{1}{|B|} * \sum_{s\epsilon\, B}^{n} \min_{i=1,2} Q_{\emptyset_i}(s, \tilde{a})\,) - \alpha \log \pi_\theta(\,\tilde{a}|s))$$

- Θ Update : It requires gradient ascent as we need maximize expected reward:

$$\Delta_\theta \frac{1}{|B|} * \sum_{s\epsilon\, B}^{n} \min_{i=1,2} Q_{\emptyset_i}(s, \tilde{a})\,) - \alpha \log \pi_\theta(\,\tilde{a}|s))$$

Critic Target Update

- Update Φ1' and Φ2' using soft update : $\emptyset_{targ,i} = \rho^* \, \emptyset_{targ,i}$+(1- $\rho)^* \, \emptyset_i$

Pseudo Code SAC [7]

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
    Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
    Execute a in the environment
    Observe next state s', reward r, and done signal d to indicate whether s' is terminal
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
    If s' is terminal, reset environment state.
    if it's time to update **then**
        for j in range(however many updates) **do**
            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
            Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

            Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2 \qquad \text{for } i = 1, 2$$

            Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

Soft Actor Critic Example with Code

- Sample Code

  https://github.com/openai/spinningup/blob/master/spinup/algos/pytorch/sac/sac.py

- Implementation Code

  https://github.com/zhihanyang2022/pytorch-sac

SAC Sample Code:

```python
# Define the actor network
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3_mean = nn.Linear(32, action_dim)
        self.fc3_stddev = nn.Linear(32, action_dim)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        action_mean = self.fc3_mean(x)
        action_stddev = torch.exp(self.fc3_stddev(x))
        return action_mean, action_stddev
```

```python
# Define the critic network
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim + action_dim, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, state, action):
        x = torch.cat([state, action], dim=1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        q_value = self.fc3(x)
        return q_value
```

```python
# Define the SAC agent
class SACAgent:
    def __init__(self, state_dim, action_dim):
        self.actor = Actor(state_dim, action_dim)
        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr=0.001)
        self.critic1 = Critic(state_dim, action_dim)
        self.critic2 = Critic(state_dim, action_dim)
        self.critic1_optimizer = optim.Adam(self.critic1.parameters(), lr=0.001)
        self.critic2_optimizer = optim.Adam(self.critic2.parameters(), lr=0.001)
        self.target_entropy = -action_dim  # Target entropy for entropy regularization
        self.alpha = 0.2  # Initial temperature coefficient
        self.log_alpha = torch.tensor(np.log(self.alpha), requires_grad=True)
        self.alpha_optimizer = optim.Adam([self.log_alpha], lr=0.001)
        self.replay_buffer = []

    def select_action(self, state):
        state = torch.FloatTensor(state)
        action_mean, action_stddev = self.actor(state)
        normal_distribution = torch.distributions.Normal(action_mean, action_stddev)
        action = normal_distribution.sample()
        return action.detach().numpy()
```

```python
def update(self, batch_size):
    if len(self.replay_buffer) < batch_size:
        return

    # Sample a batch of transitions from the replay buffer
    batch = random.sample(self.replay_buffer, batch_size)
    state_batch, action_batch, reward_batch, next_state_batch = zip(*batch)

    state_batch = torch.FloatTensor(state_batch)
    action_batch = torch.FloatTensor(action_batch)
    reward_batch = torch.FloatTensor(reward_batch)
    next_state_batch = torch.FloatTensor(next_state_batch)

    # Compute the target Q-values
    with torch.no_grad():
        next_action_mean, next_action_stddev = self.actor(next_state_batch)
        next_normal_distribution = torch.distributions.Normal(next_action_mean, next_action_stddev)
        next_action = next_normal_distribution.sample()
        next_q1 = self.critic1(next_state_batch, next_action)
        next_q2 = self.critic2(next_state_batch, next_action)
        next_q = torch.min(next_q1, next_q2)
        target_q = reward_batch + 0.99 * (next_q - self.alpha * next_normal_distribution.log_prob(next_action))
```

```python
# Update the critic networks
q1_pred = self.critic1(state_batch, action_batch)
q2_pred = self.critic2(state_batch, action_batch)
critic1_loss = F.mse_loss(q1_pred, target_q)
critic2_loss = F.mse_loss(q2_pred, target_q)
self.critic1_optimizer.zero_grad()
self.critic2_optimizer.zero_grad()
critic1_loss.backward()
critic2_loss.backward()
self.critic1_optimizer.step()
self.critic2_optimizer.step()
```

```python
# Update the actor network and temperature coefficient
action_mean, action_stddev = self.actor(state_batch)
normal_distribution = torch.distributions.Normal(action_mean, action_stddev)
action = normal_distribution.sample()
q1 = self.critic1(state_batch, action)
q2 = self.critic2(state_batch, action)
min_q = torch.min(q1, q2)
actor_loss = (self.alpha * normal_distribution.log_prob(action) - min_q).mean()
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()
```

Reference

1. *A3C-GS: Adaptive moment gradient sharing with locks for asynchronous Actor–Critic agents*. (2021, March 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/document/9063667

2. https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf

3. De Harder, H. (2023, July 18). Techniques to improve the performance of a DQN agent. *Medium*. https://towardsdatascience.com/techniques-to-improve-the-performance-of-a-dqn-agent-29da8a7a0a7e

4. Makone, A. (2022, January 5). Reinforcement Learning 8: Pick and place robot in an E-Commerce store warehouse i.e. Q-Learning in action. *Medium*. https://ashutoshmakone.medium.com/reinforcement-learning-8-pick-and-place-robot-in-an-e-commerce-store-warehouse-i-e-78d7af7e60c8

5. *Multi-Agent Reinforcement Learning using the Deep Distributed Distributional Deterministic Policy Gradients Algorithm*. (2020, December 20). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/9311945

6. BartoSutton.pdf (cmu.edu)

7. *Soft Actor-Critic — Spinning Up documentation*. (n.d.). https://spinningup.openai.com/en/latest/algorithms/sac.html

8. Weng, L. (2018a, February 19). A (Long) Peek into Reinforcement Learning. *Lil'Log*. https://lilianweng.github.io/posts/2018-02-19-rl-overview/?fbclid=IwAR161iWjfQMpTn0Z4y8MdXb3rYhPum89XD2wAS05HP49aRQeYNyjIRf9Nyk#key-concepts

9. L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu and L. Li, "Delay-Aware and Energy-Efficient Computation Offloading in Mobile-Edge Computing Using Deep Reinforcement Learning," in IEEE Transactions on Cognitive Communications and Networking, vol. 7, no. 3, pp. 881-892, Sept. 2021, doi: 10.1109/TCCN.2021.3066619.