Policy Gradient

Just optimize it!

Jaewon Choi (SNUH), 2025.05.28

Goal: Policy Gradient

$$\theta = \theta + \alpha \nabla_{\theta} J(\theta)$$

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{T} \Phi_t \nabla_{\theta} log \pi_{\theta}(A_t | S_t) \right]$$

Problem Statement

$$\tau = (S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_{T+1})$$ Trajectory

$$G(\tau) = R_0 + \gamma R_1 + \gamma^2 R_2 + \ldots + \gamma^T R_T$$

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[G(\tau)\right]$$

기대 장기 보상

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}\left[G(\tau)\right]$$

$$= \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} G(\tau)\, \nabla_\theta log \pi_\theta(A_t \,|\, S_t)\right]$$

Problem Statement

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[G(\tau)\right]$$

$$= \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} G(\tau)\,\nabla_\theta log \pi_\theta(A_t\,|\,S_t)\right]$$

$$\theta = \theta + \alpha\nabla_\theta J(\theta)$$

$$\nabla_\theta log \pi_\theta(A_t\,|\,S_t) = \frac{\nabla_\theta \pi_\theta(A_t\,|\,S_t)}{\pi_\theta(A_t\,|\,S_t)}$$

REINFORCE algorithm

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} G_t \nabla_\theta log \pi_\theta(A_t \,|\, S_t) \right]$$

$$G_t = R_t + \gamma R_{t+1} + \ldots + \gamma^{T-t} R_T$$

REINFORCE algorithm

```python
# 간단한 정책 네트워크
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(PolicyNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim),
            nn.Softmax(dim=-1)
        )

    def forward(self, state):
        return self.net(state)
```

```python
# 학습 루프
for episode in range(1000):
    state = env.reset()
    log_probs = []
    rewards = []

    done = False
    while not done:
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        probs = policy(state_tensor)
        dist = torch.distributions.Categorical(probs)
        action = dist.sample()

        log_prob = dist.log_prob(action)
        log_probs.append(log_prob)

        state, reward, done, _ = env.step(action.item())
        rewards.append(reward)
```

REINFORCE algorithm

```
# 누적 보상 계산
returns = []
R = 0
for r in reversed(rewards):
    R = r + gamma * R
    returns.insert(0, R)
returns = torch.tensor(returns)

# Normalize returns (optional but helpful)
returns = (returns - returns.mean()) / (returns.std() + 1e-9)
```

```
# 손실 계산 및 업데이트
loss = 0
for log_prob, Gt in zip(log_probs, returns):
    loss += -log_prob * Gt  # Gradient ascent -> minimize negative

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

REINFORCE algorithm with baseline

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} G_t \nabla_\theta log \pi_\theta(A_t | S_t) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} (G_t - b(S_t)) \nabla_\theta log \pi_\theta(A_t | S_t) \right]
\end{aligned}
$$

Actor-Critic Model

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} (G_t - b(S_t)) \, \nabla_\theta log \pi_\theta(A_t \,|\, S_t) \right]$$

$$= \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} (G_t - V_w(S_t)) \, \nabla_\theta log \pi_\theta(A_t \,|\, S_t) \right]$$

Actor-Critic Model

```python
# Actor-Critic Network
class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(ActorCritic, self).__init__()
        self.shared = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU()
        )
        self.actor = nn.Linear(hidden_dim, action_dim)
        self.critic = nn.Linear(hidden_dim, 1)

    def forward(self, state):
        x = self.shared(state)
        logits = self.actor(x)
        value = self.critic(x)
        return logits, value
```

```python
# 학습 루프
for episode in range(1000):
    state = env.reset()
    log_probs = []
    values = []
    rewards = []

    done = False
    while not done:
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        logits, value = model(state_tensor)

        # 정책으로부터 행동 선택
        dist = torch.distributions.Categorical(logits=logits)
        action = dist.sample()
        log_prob = dist.log_prob(action)

        next_state, reward, done, _ = env.step(action.item())

        log_probs.append(log_prob)
        values.append(value)
        rewards.append(reward)

        state = next_state
```

Actor-Critic Model

```python
# 마지막 상태의 value는 0으로 가정 (terminal)
returns = []
R = 0
for r in reversed(rewards):
    R = r + gamma * R
    returns.insert(0, R)

returns = torch.tensor(returns)
values = torch.cat(values).squeeze()
log_probs = torch.stack(log_probs)
```

```python
# Advantage 계산
advantage = returns - values.detach()

# 손실 함수
actor_loss = -(log_probs * advantage).mean()
critic_loss = nn.MSELoss()(values, returns)
loss = actor_loss + critic_loss

# 경사 하강
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Actor-Critic Model

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} (R_t + \gamma V_w(S_{t+1}) - V_w(S_t)) \, \nabla_\theta log \pi_\theta(A_t \,|\, S_t) \right]$$

Policy Gradient

- Pros
  - Direct optimization
  - Works for continuous action space
  - Stochastic policy
- Cons
  - High variance
  - Sample inefficiency
  - Sensitive to hyper parameters

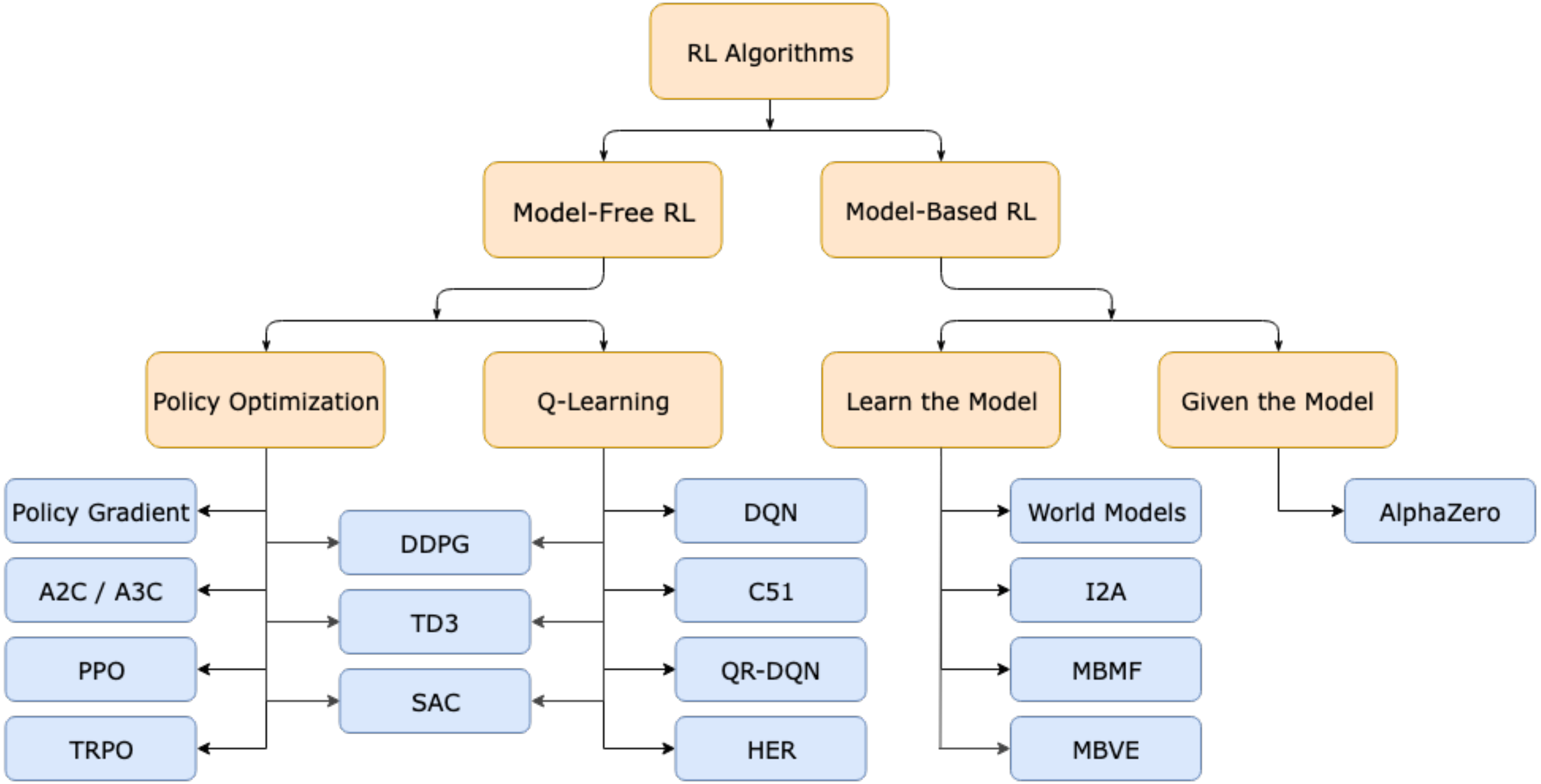
Goal: Policy Gradient

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \Phi_t \nabla_\theta log \pi_\theta(A_t \mid S_t) \right]$$

Taxonomy of Reinforcement Learning algorithms

By Learning Paradigm



Type of policy

- Value-based (Q-learning, DQN)
  - 상태의 가치를 추정하여 행동 선택
- Policy-based (REINFORCE)
  - 정책을 직접 학습
- Actor-Critic (A3C, PPO, DDPG)
  - 정책 + 가치 함수 동시 학습

Value usage

- Value-based: 가치 함수 사용 (e.g., Q-function)
- Policy-based: 정책 함수만 사용
- Hybrid: 둘 다 사용 (Actor-Critic)
- 용도에 따라 선택
  - Value-based 는 탐험 효율이 높음
  - Policy-based는 연속 액션 공간에 강함

Action 공간

- Discrete: 행동이 명확히 나뉘어 있음 (Q-learning, DQN)
- Continuous: 행동이 연속 값 (DDPG, SAC, TD3)
  - Discretization을 하는 것이 나은지 Policy gradient를 하는게 나은지 잘 선택해야 함

On-policy vs Off-policy

- On-policy
  - 정책 안정성
  - Sample inefficiency
  - Exploration이 필수적
- Off-policy
  - 데이터 효율성 (replay buffer 사용 가능)
  - 실제 데이터 + 로그 학습에 유리
  - 배치 학습 (offline RL) 가능
  - Distribution mismatch 위험

Model usage

- Model-free (DQN, PPO)
  - 환경 모델 없이 직접 상호작용
- Model-based (MuZero, Dreamer)
  - 환경 모델을 학습 후 사용
  - Model-based는 예측된 시퀀스를 생성해 planning

Online vs Offline

- Online
  - 실시간으로 샘플 수집 및 업데이트
  - 현실 적용 쉬움, 샘플 효율 낮음
  - e.g., 게임 플레이, 로봇 제어, 시뮬레이션 기반 전략 학습 등
- Offline (Batch RL)
  - 기존 데이터셋으로만 학습
  - 데이터 효율 높음, 안정성 문제
  - 병원 환자 기록, 자율주행 로그, 금융 거래 내역 등

환경 특성에 따른 분류

- Markov Decision Process (MDP) vs Partially Observable MDP (POMDP)
- Single-agent vs Multi-agent
- Stationary vs Non-stationary environment
- Deterministic vs Stochastic dynamics
- E.g.,
  - AlphaStar: Multi-agent + POMDP + Stochastic
  - Robotics: Continuous action + Model-based + POMDP

Conclusion

- 강화학습 방법은 굉장히 많고 comprehensive coverage 를 갖는다
- 원리는 우리가 충분히 배웠으니, “상황을 이해해서” 어떤 환경과 모델을 적용할지 잘 골라 보자
- 고생 많으셨음!