

Introduction to policy gradient

DEEP REINFORCEMENT LEARNING IN PYTHON



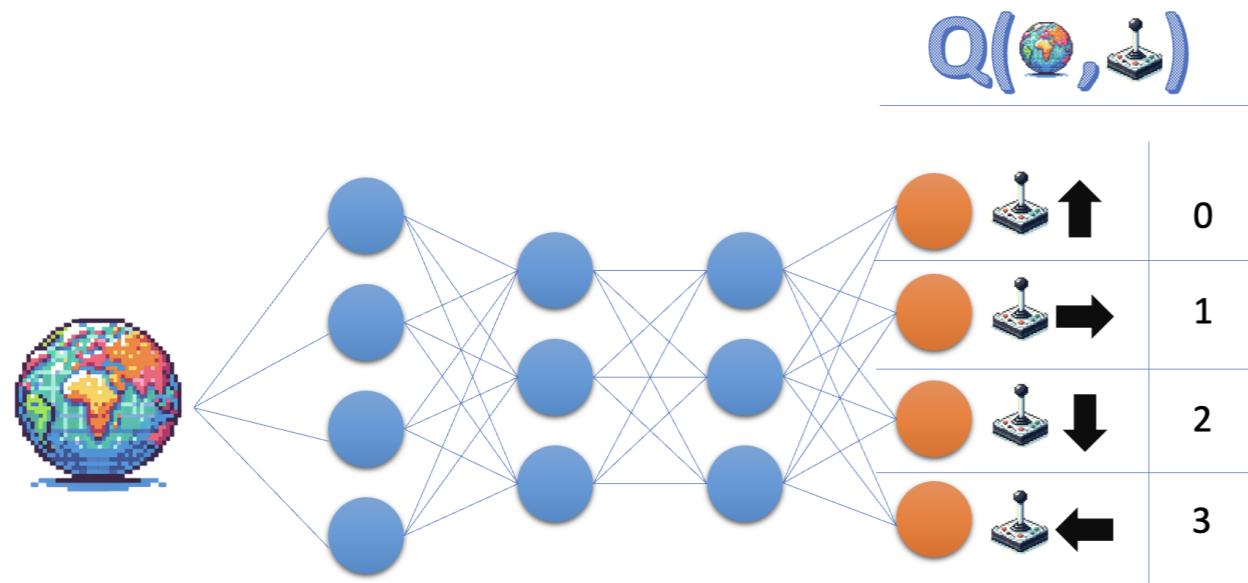
Timothée Carayol

Principal Machine Learning Engineer,
Komment

Introduction to Policy methods in DRL

Q-learning:

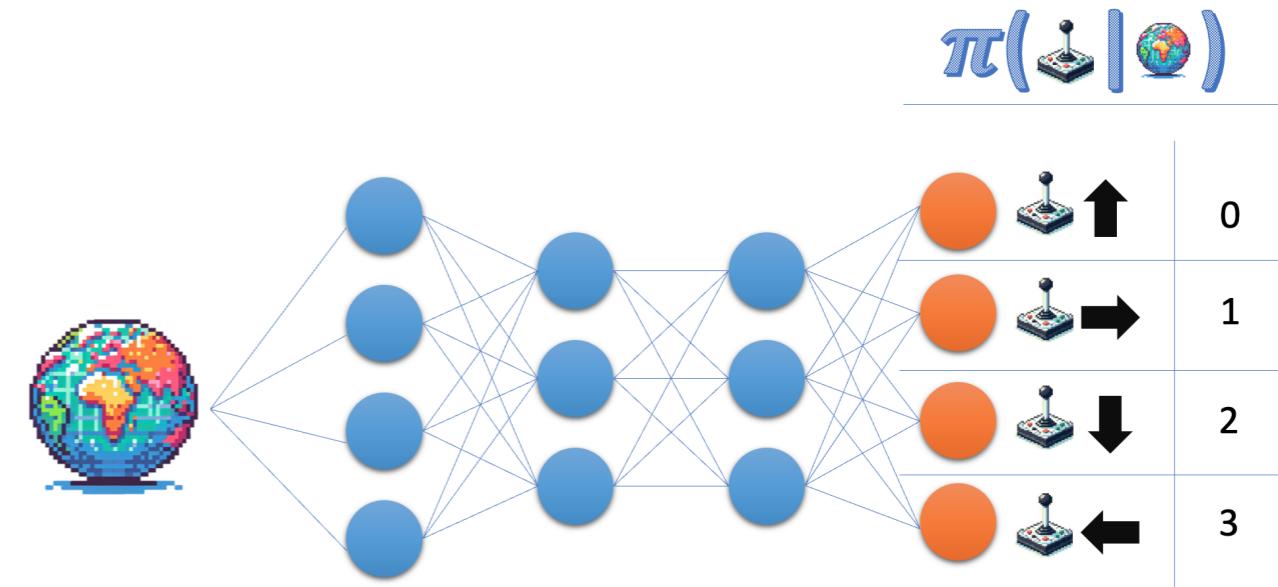
- Learn the action value function Q



- Policy: select action with highest value

Policy learning:

- Learn the policy directly



Policy learning

- Can be stochastic
 - Handle continuous spaces
 - Directly optimize for the objective
 - High variance
 - Less sample efficient
-
- In Deep-Q learning: policies are deterministic

$$\pi_{\theta}(a_t | s_t):$$

- Probability distribution for a_t in state s_t , with:
 - a_t, s_t : action and state at step t
 - θ : policy parameters (network weights)

The policy network (discrete actions)

```
class PolicyNetwork(nn.Module):
    def __init__(self, state_size, action_size):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

    def forward(self, state):
        x = torch.relu(self.fc1(torch.tensor(state)))
        x = torch.relu(self.fc2(x))
        action_probs = torch.softmax(self.fc3(x), dim=-1)
        return action_probs
```

```
action_probs = policy_network(state)
print('Action probabilities:', action_probs)
```

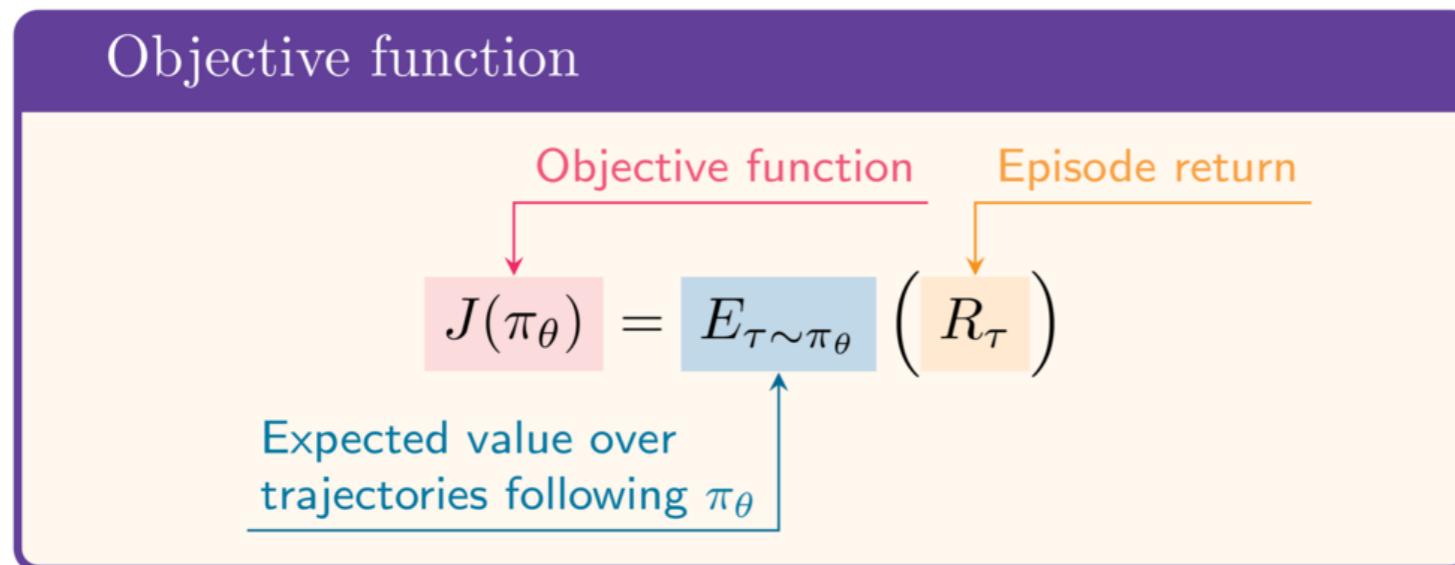
```
Action probabilities: tensor([0.21, 0.02, 0.74, 0.03])
```

Action	index	$\pi(\cdot s)$
 	0	0.21
 	1	0.02
 	2	0.74
 	3	0.03

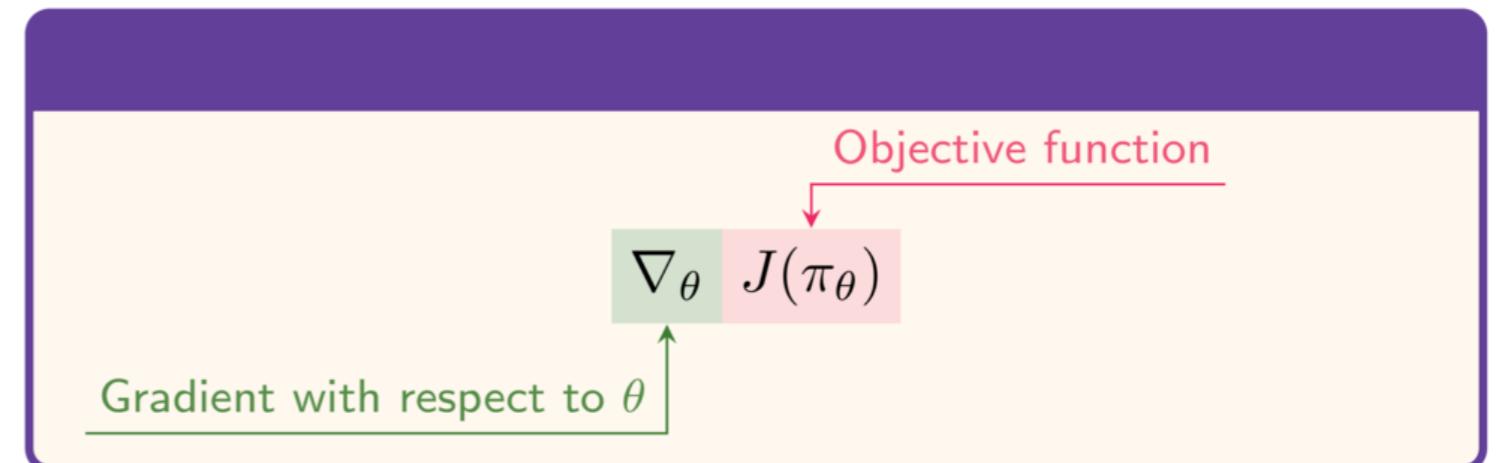
```
action_dist = (
    torch.distributions.Categorical(action_probs))
action = action_dist.sample()
```

The objective function

- Policy must maximize expected returns
 - Assuming the agent follows π_θ
 - By optimizing policy parameter θ
- Objective function:

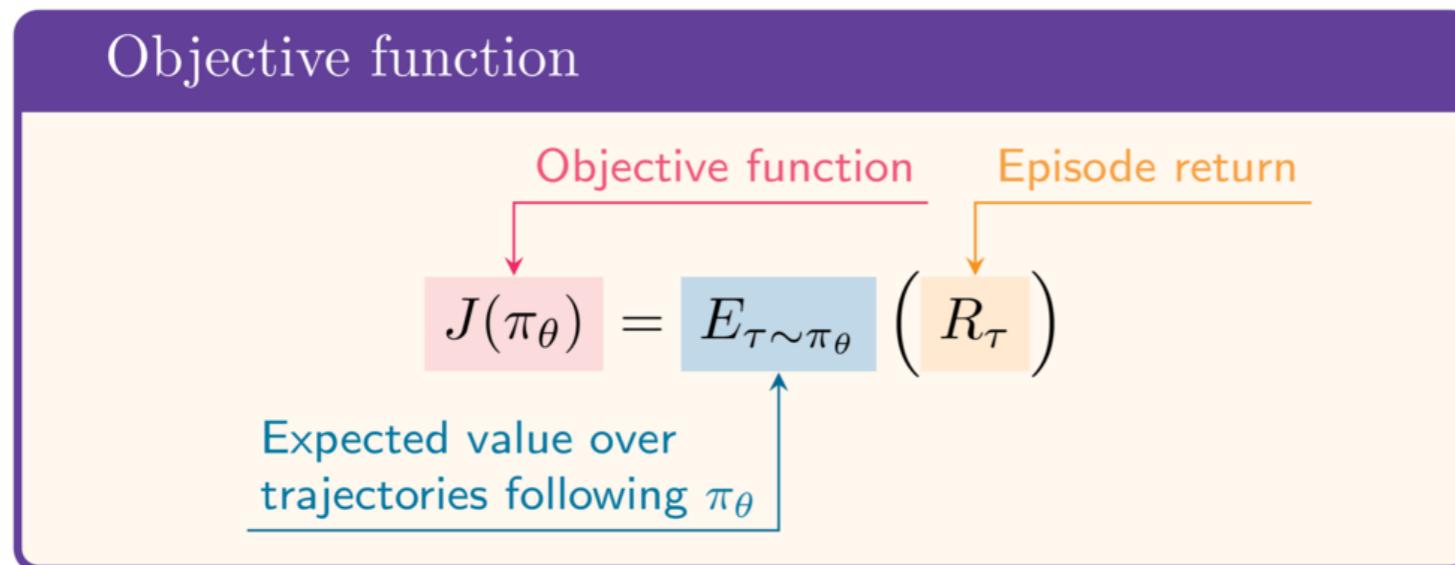


- To maximize J : need gradient with respect to θ :

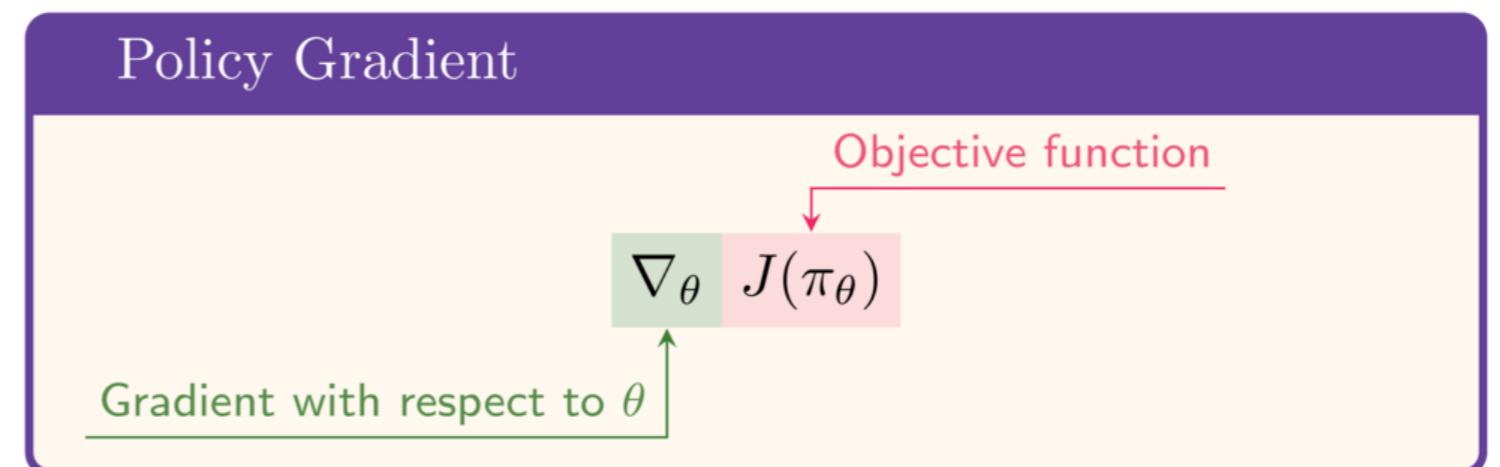


The objective function

- Policy must maximize expected returns
 - Assuming the agent follows π_θ
 - By optimizing policy parameter θ
- Objective function:

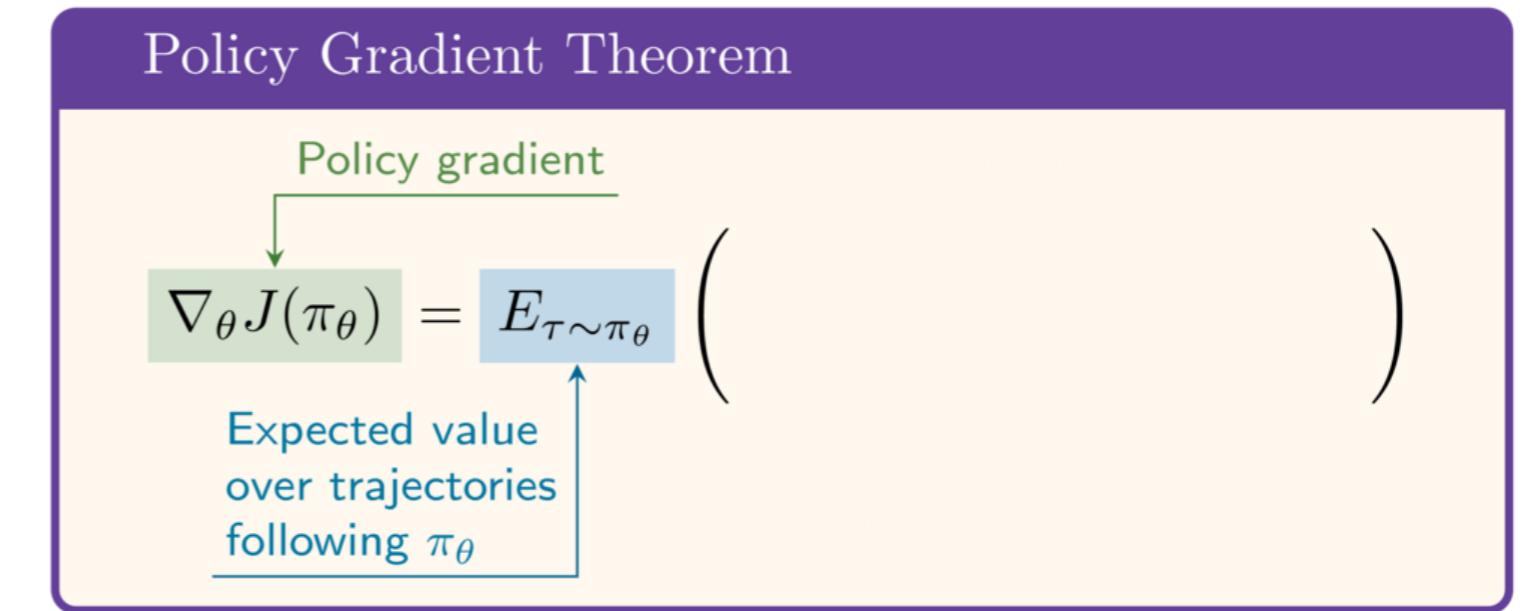


- To maximize J : need gradient with respect to θ :



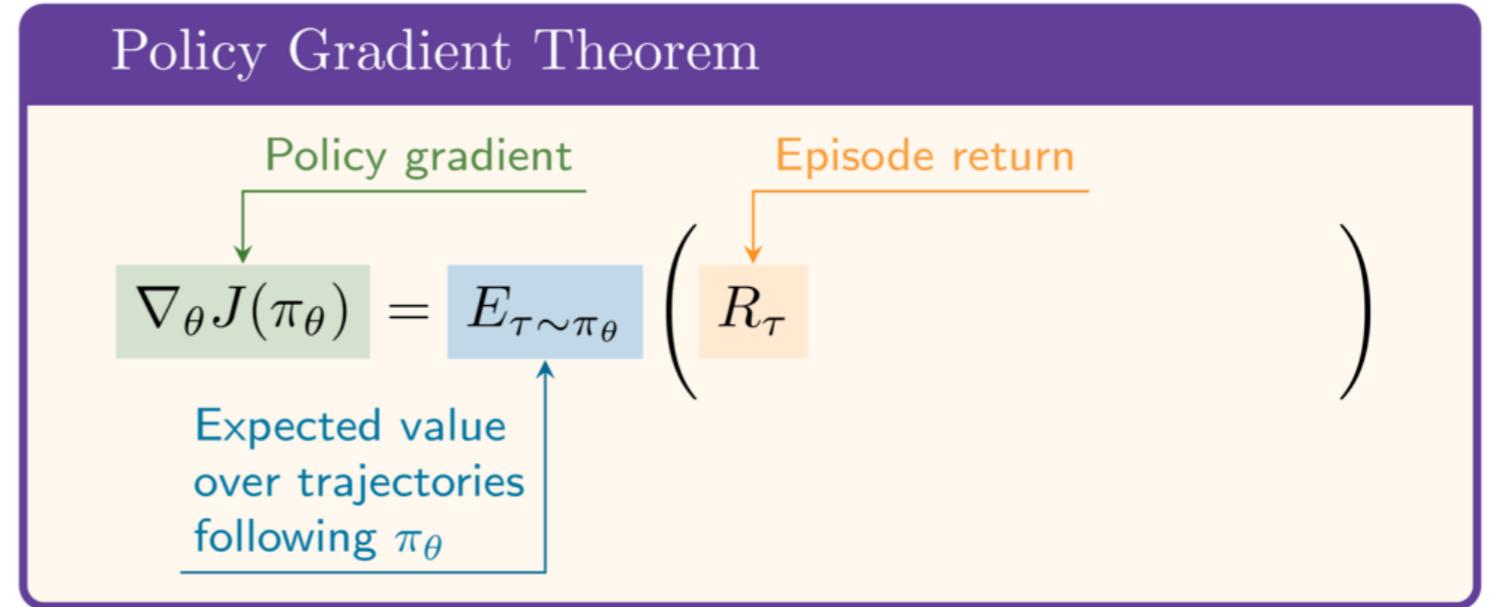
The policy gradient theorem

- Gives a tractable expression for $\nabla_{\theta} J(\pi_{\theta})$
- Expectation over trajectories following π_{θ}
 - Collect trajectories and observe the returns



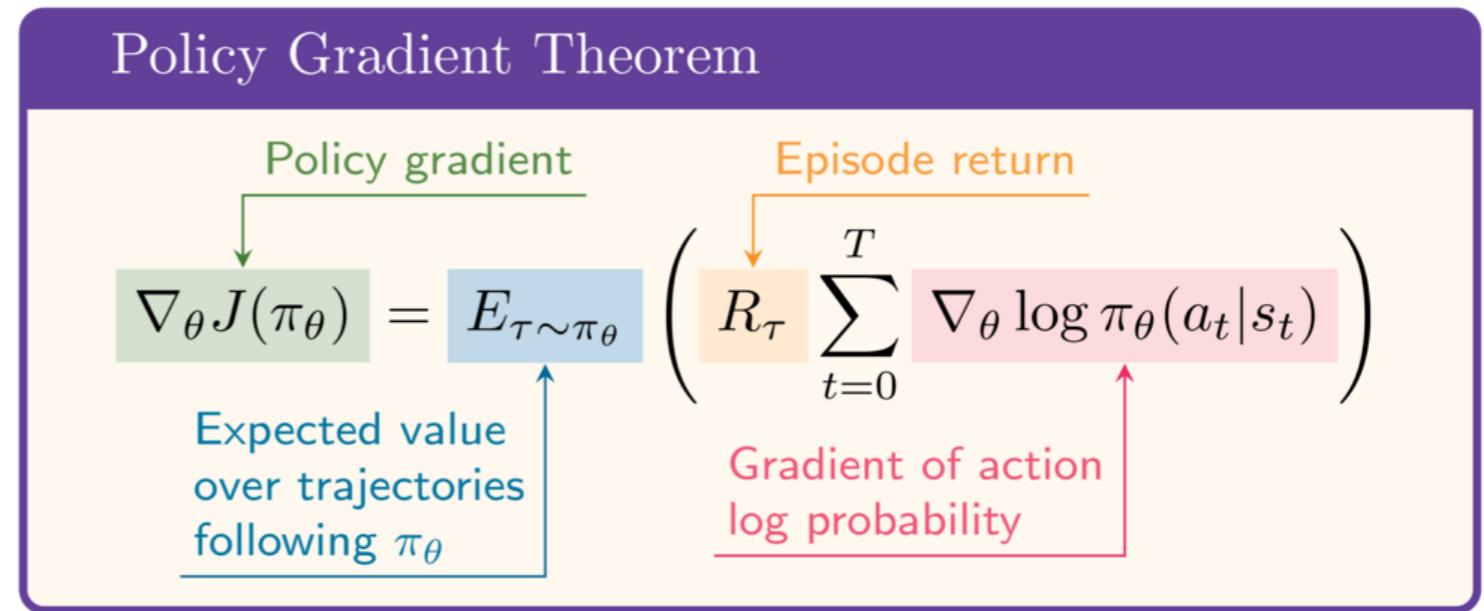
The policy gradient theorem

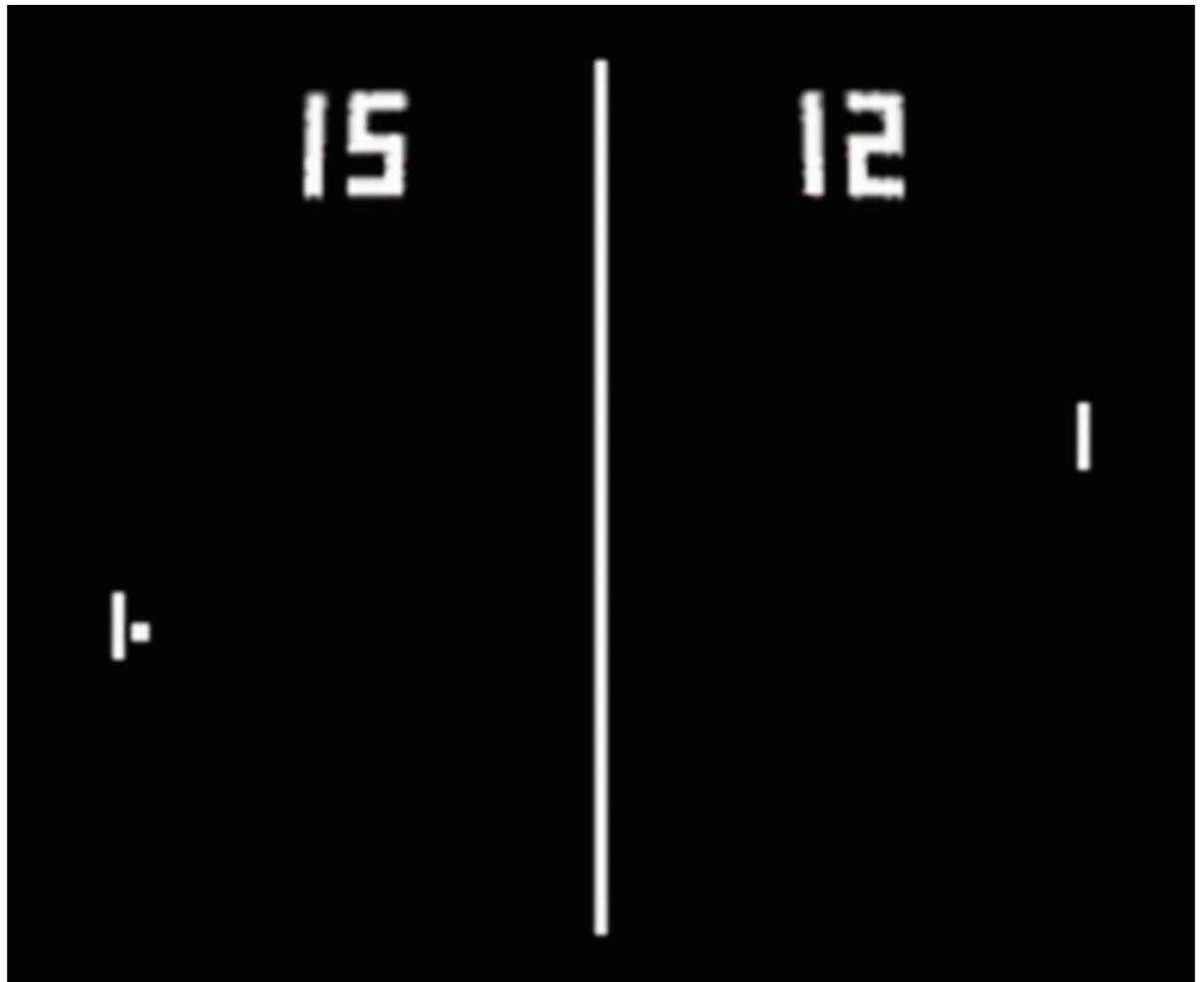
- Gives a tractable expression for $\nabla_{\theta} J(\pi_{\theta})$
- Expectation over trajectories following π_{θ}
 - Collect trajectories and observe the returns
- For each trajectory: consider return R_{τ}



The policy gradient theorem

- Gives a tractable expression for $\nabla_{\theta} J(\pi_{\theta})$
- Expectation over trajectories following π_{θ}
 - Collect trajectories and observe the returns
- For each trajectory: consider return R_{τ}
- Multiply by the sum of gradients of log probabilities of selected actions
- Intuition: nudge theta in ways that increase probability of all actions taken in a 'good' episode





Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

Policy gradient and **REINFORCE**

DEEP REINFORCEMENT LEARNING IN PYTHON

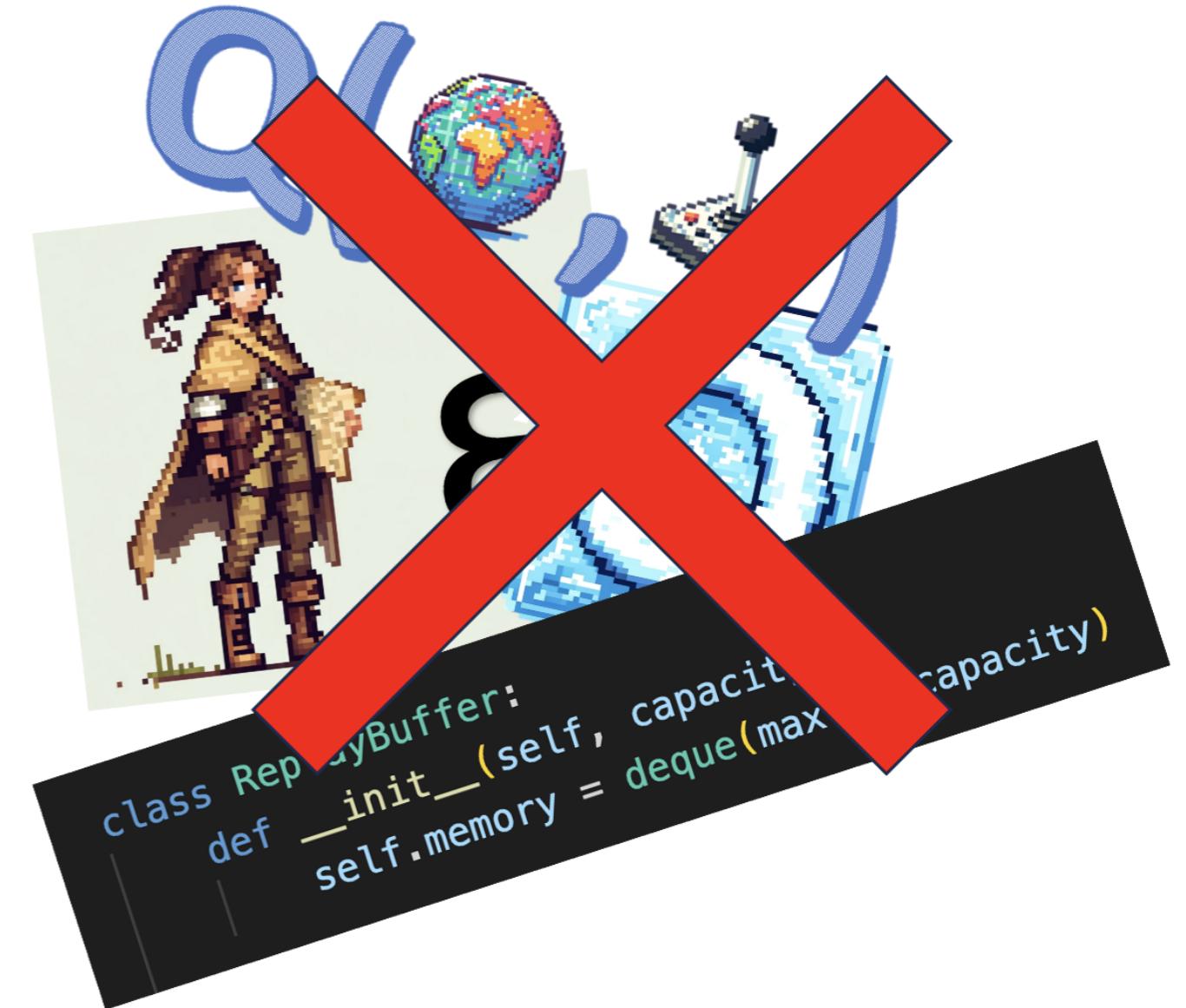


Timothée Carayol

Principal Machine Learning Engineer,
Komment

Differences with DQN

- REINFORCE: Monte-Carlo, not Temporal Difference
 - Update at the end of the episode, *not* at every step
 - Can update after several episodes instead
- No value function
- No target network
- No epsilon-greediness
- No experience replay



The REINFORCE training loop structure

```
for episode in range(num_episodes):
    # 1. Initialize episode
    while not done:
        # 2. Select action
        # 3. Play action and obtain next state and reward
        # 4. Add (discounted) reward to return
        # 5. Update state
        # 6. Calculate loss
        # 7. Update policy network by gradient descent
```

Action Selection

```
from torch.distributions import Categorical

def select_action(policy_network, state):
    action_probs = policy_network(state)
    action_dist = Categorical(action_probs)
    action = action_dist.sample()
    log_prob = action_dist.log_prob(action)
    return action.item(), log_prob.reshape(1)

action, log_prob = select_action(
    policy_network, state)
```

- Obtain probabilities from network
- Sample one action
- Return action and corresponding log probabilities

Sampled action index: 1
Log probability of sampled action: -1.38

Loss Calculation

Recall the policy gradient theorem:

Policy Gradient Theorem

$$\nabla_{\theta} J(\pi_{\theta}) = E_{\tau \sim \pi_{\theta}} \left(R_{\tau} \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right)$$

Policy gradient
Episode return
Expected value over trajectories following π_{θ}
Gradient of action log probability

REINFORCE Loss function

$$\text{Loss} \quad \text{Episode return}$$
$$\mathcal{L}(\theta) = - R_{\tau} \sum_{t=0}^T \log \pi_{\theta}(a_t | s_t)$$

Sum of action log probabilities

In Python:

- R_{τ} as `episode_return`
- Vector of $\log \pi_{\theta}(a_t | s_t)$ as `episode_log_probs`

```
loss = -episode_return * episode_log_probs.sum()
```

The REINFORCE training loop

```
for episode in range(50):
    state, info = env.reset(); done = False; step = 0;
    episode_log_probs = torch.tensor([])
    R = 0
    while not done:
        step += 1
        action, log_prob = select_action(policy_network, state)
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        R += (gamma ** step) * reward
        episode_log_probs = torch.cat((episode_log_probs, log_prob))
        state = next_state
    loss = - R * episode_log_probs.sum()
    optimizer.zero_grad(); loss.backward(); optimizer.step()
```

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

Advantage Actor Critic

DEEP REINFORCEMENT LEARNING IN PYTHON

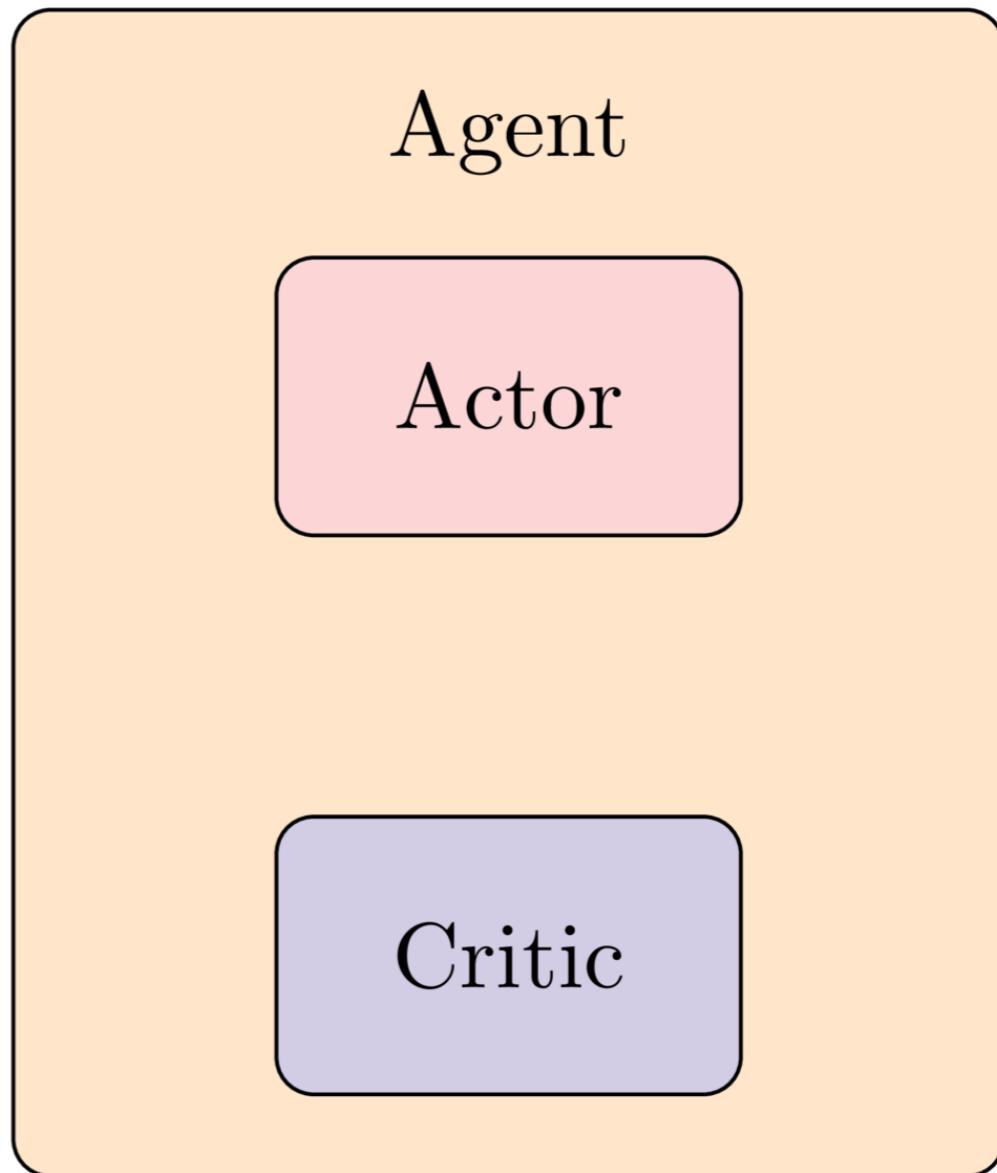


Timothée Carayol

Principal Machine Learning Engineer,
Komment

Why actor critic?

- REINFORCE limitations:
 - High variance
 - Poor sample efficiency
- Actor Critic methods introduce a critic network, enabling Temporal Difference learning



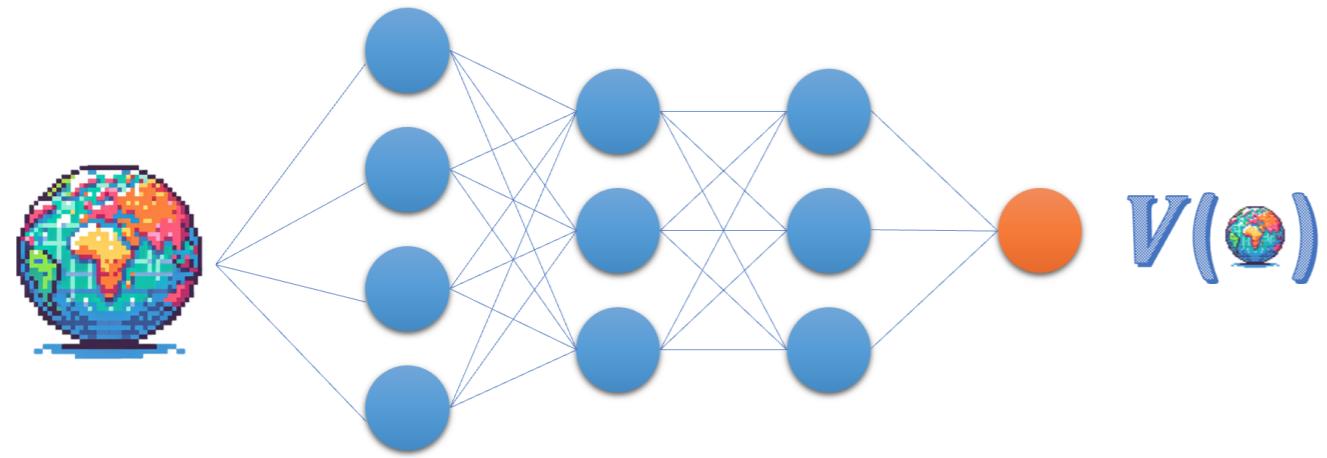
The intuition behind Actor Critic methods



- Actor network:
 - Makes decisions
 - Cannot evaluate them
- Critic network:
 - Provides feedback to actor at every step

The Critic network

- Critic approximates the state value function

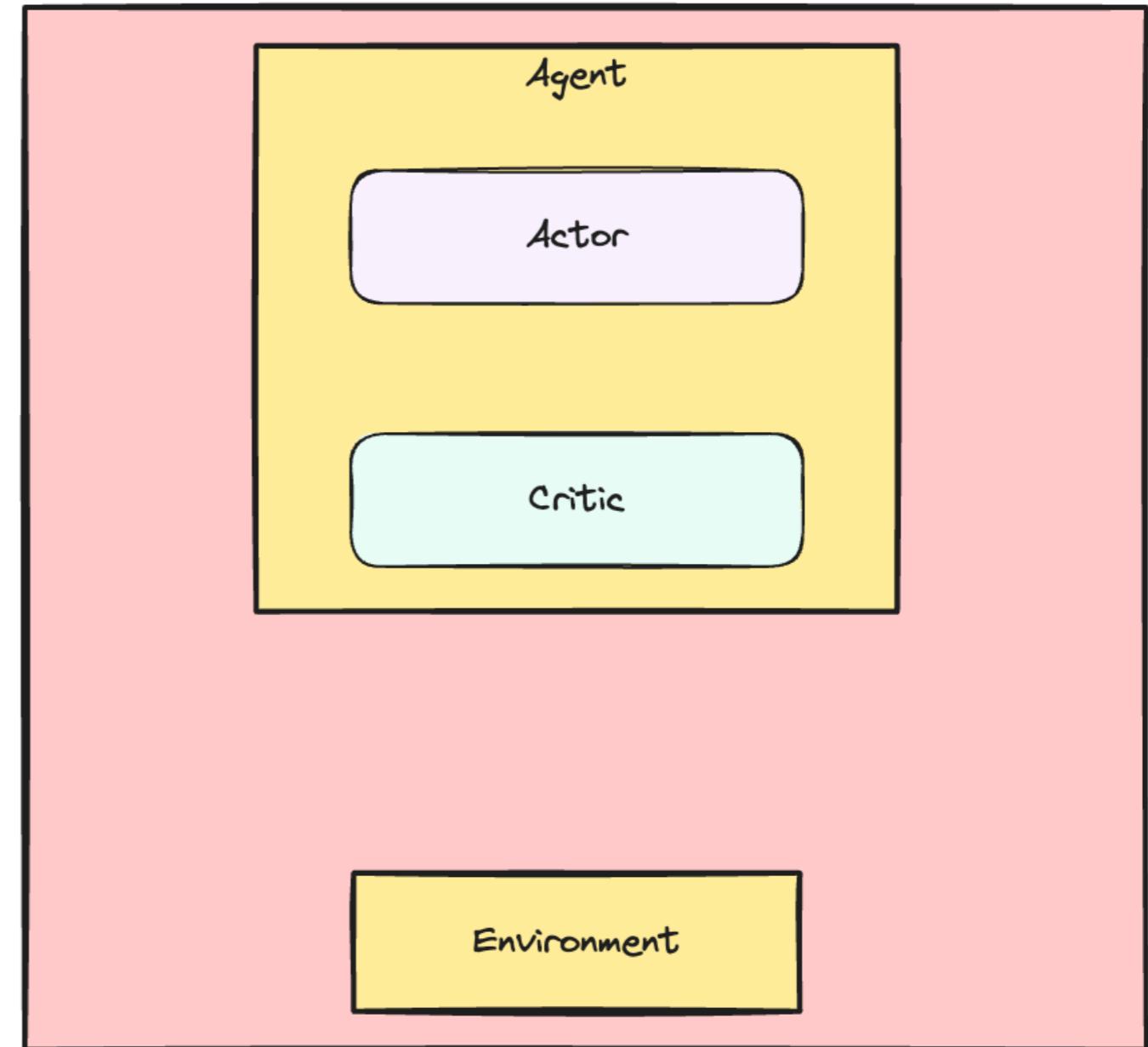


- Judges action a_t based on the advantage or TD-error

```
class Critic(nn.Module):  
    def __init__(self, state_size):  
        super(Critic, self).__init__()  
        self.fc1 = nn.Linear(state_size, 64)  
        self.fc2 = nn.Linear(64, 1)  
  
    def forward(self, state):  
        x = torch.relu(self.fc1(torch.tensor(state)))  
        value = self.fc2(x)  
        return value  
  
critic_network = Critic(8)
```

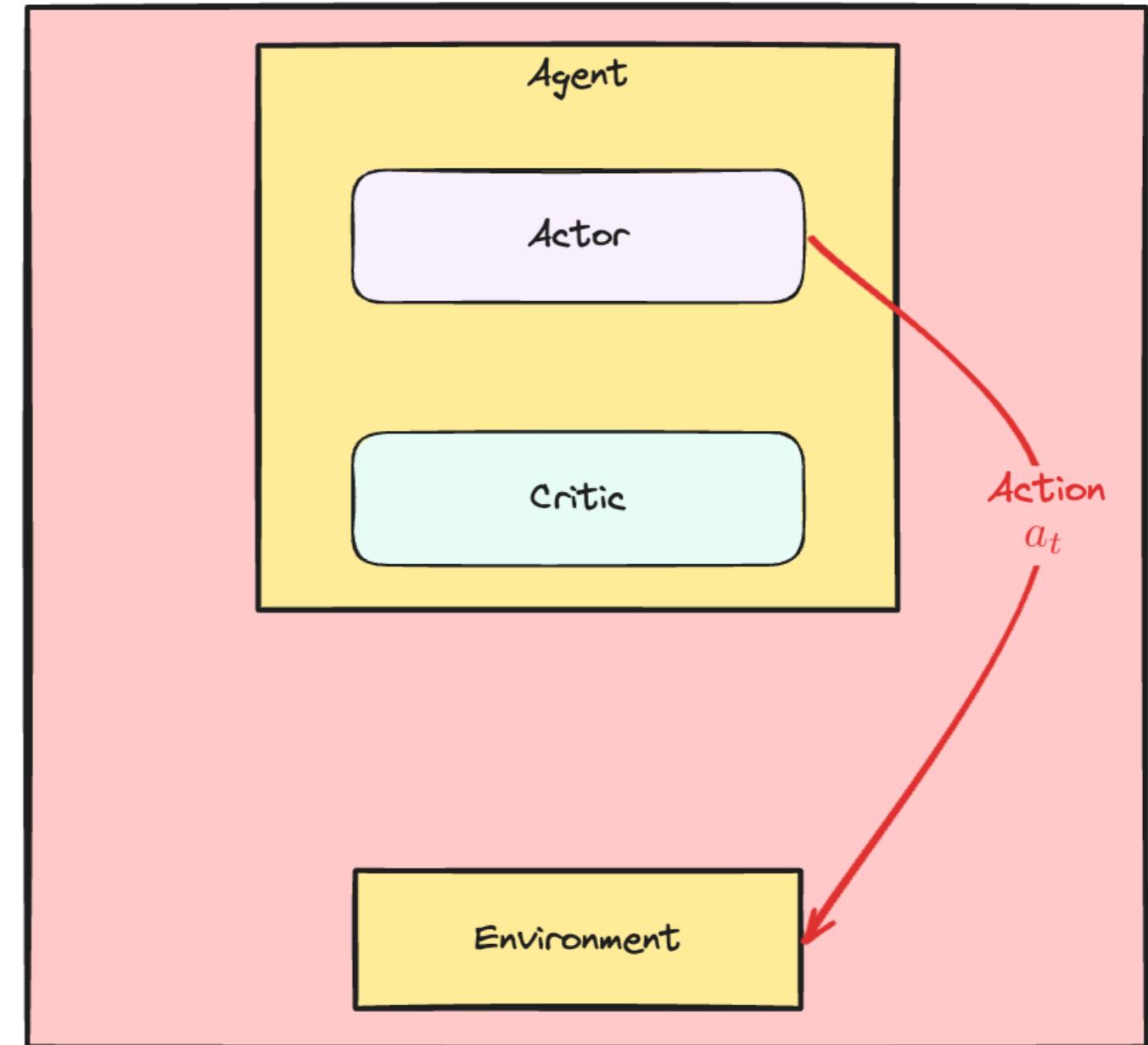
The Actor Critic dynamics

- At every step:
 - Actor chooses action (same as policy network in REINFORCE)



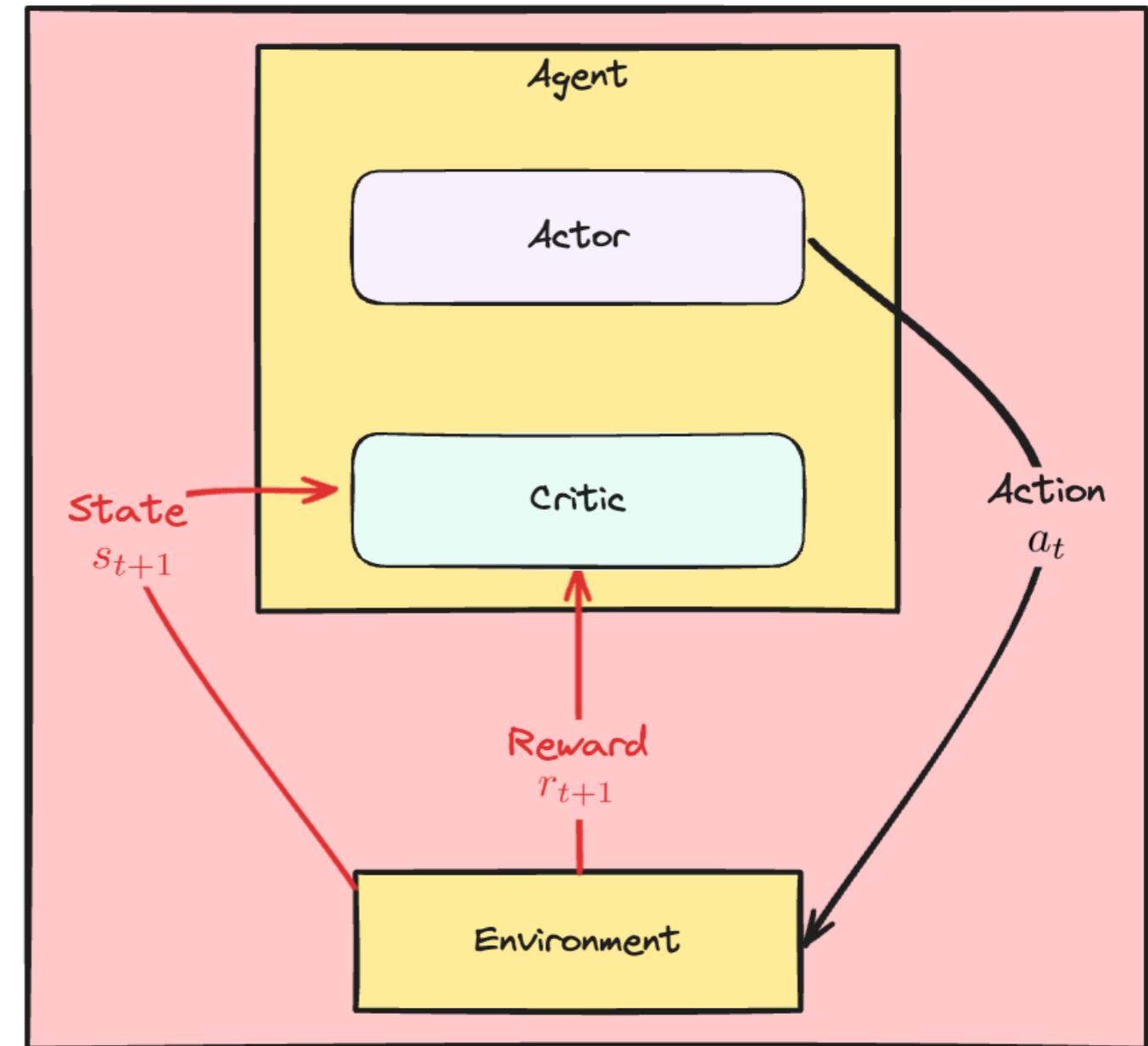
The Actor Critic dynamics

- At every step:
 - Actor chooses action (same as policy network in REINFORCE)
 - Critic observes reward and state



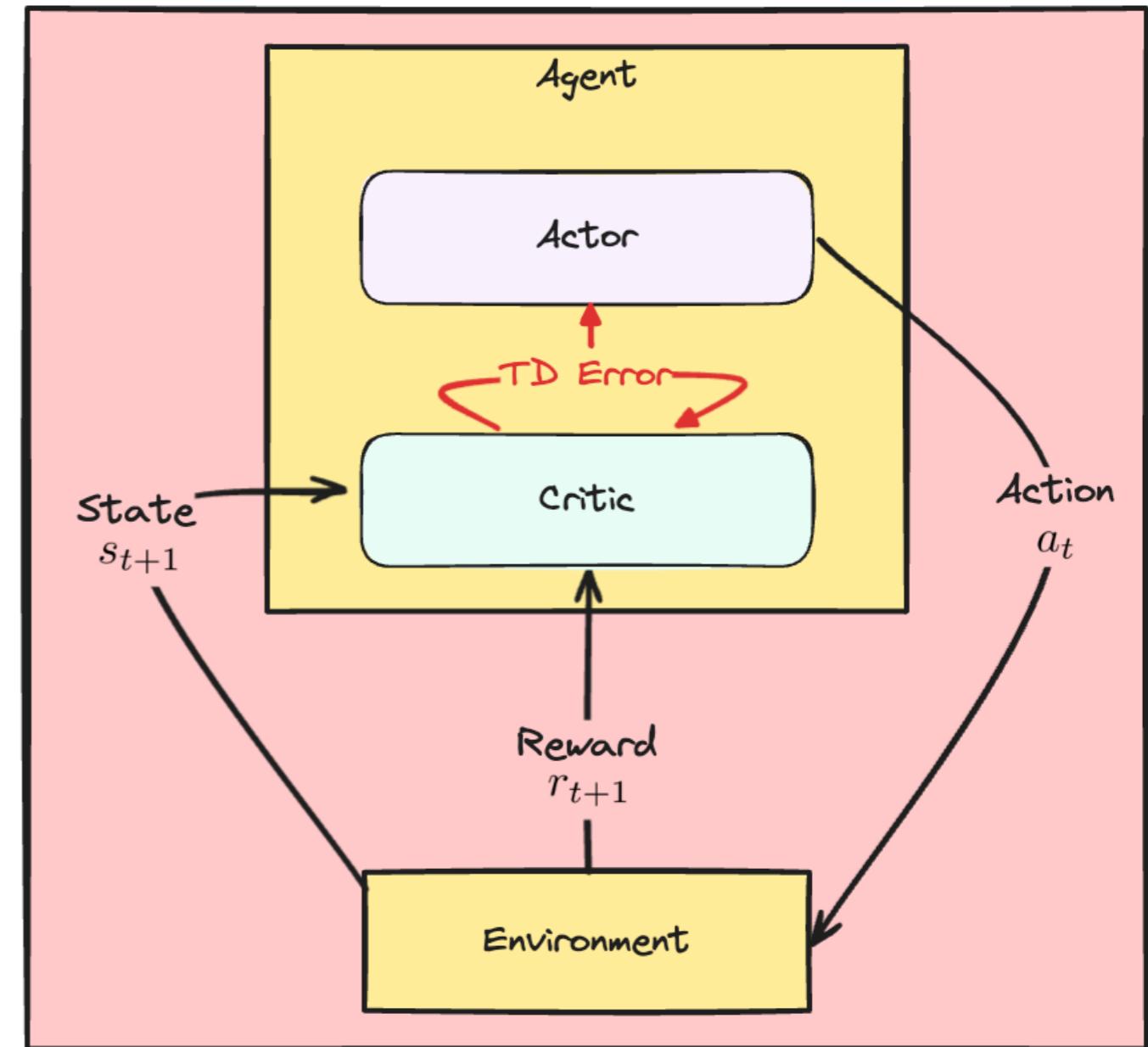
The Actor Critic dynamics

- At every step:
 - Actor chooses action (same as policy network in REINFORCE)
 - Critic observes reward and state
 - Critic evaluates TD Error
 - Actor and Critic use TD Error to update weights



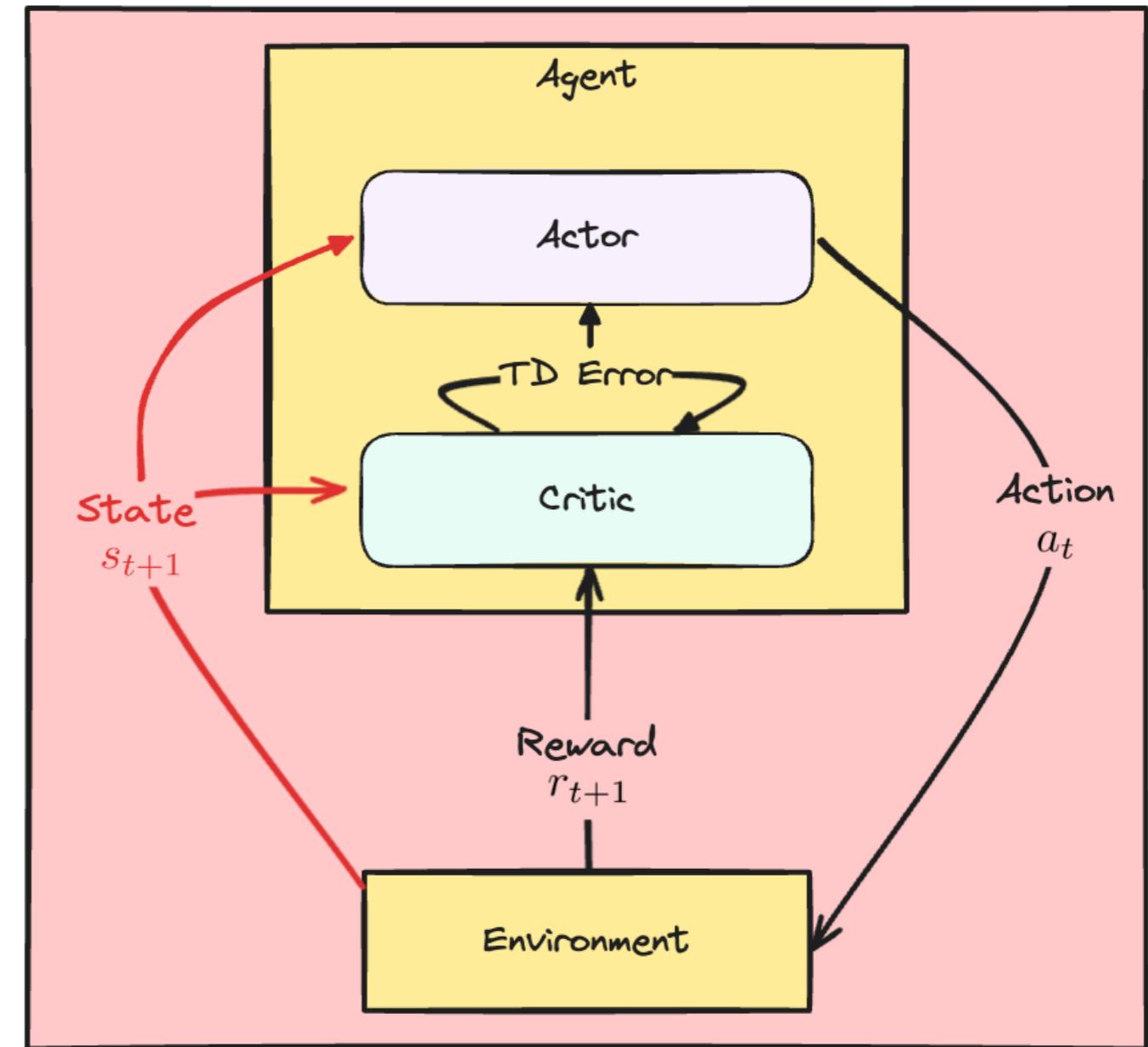
The Actor Critic dynamics

- At every step:
 - Actor chooses action (same as policy network in REINFORCE)
 - Critic observes reward and state
 - Critic evaluates TD Error
 - Actor and Critic use TD Error to update weights
 - Updated Actor observes new state



The Actor Critic dynamics

- At every step:
 - Actor chooses action (same as policy network in REINFORCE)
 - Critic observes reward and state
 - Critic evaluates TD Error
 - Actor and Critic use TD Error to update weights
 - Updated Actor observes new state
- ... start over



The A2C losses

Critic

Critic loss function

Use the Squared TD Error for the Critic:

$$L_c(\theta_c) = \left((r_t + \gamma V_{\theta_c}(s_{t+1})) - V_{\theta_c}(s_t) \right)^2$$

TD error / Advantage

Parameters of the critic

Actor

Actor loss function

It can be shown that we can use, at every step t , this loss function for the Actor:

$$L(\theta) = - \log \pi_\theta(a_t | s_t) \left((r_t + \gamma V(s_{t+1})) - V(s_t) \right)$$

Action log probability

Parameters of the actor

TD error / Advantage

- Critic loss: squared TD error
- TD error captures critic rating
- Increase probability of actions with positive TD error

Calculating the losses

```
def calculate_losses(critic_network, action_log_prob,
                     reward, state, next_state, done):
    # Critic provides the state value estimates
    value = critic_network(state)
    next_value = critic_network(next_state)
    td_target = (reward + gamma *
                 next_value * (1-done))
    td_error = td_target - value

    # Apply formulas for actor and critic losses
    actor_loss = -action_log_prob * td_error.detach()
    critic_loss = td_error ** 2

    return actor_loss, critic_loss
```

- Calculate TD-Error
- Calculate actor loss
 - Use `.detach()` to stop gradient propagation to critic weights
- Calculate critic loss

The Actor Critic training loop

```
for episode in range(10):
    state, info = env.reset()
    done = False
    while not done:
        # Select action
        action, action_log_prob = select_action(actor, state)
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        # Calculate losses
        actor_loss, critic_loss = calculate_losses(critic, action_log_prob, reward, state, next_state, done)
        # Update actor
        actor_optimizer.zero_grad(); actor_loss.backward(); actor_optimizer.step()
        # Update critic
        critic_optimizer.zero_grad(); critic_loss.backward(); critic_optimizer.step()
    state = next_state
```

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON