Chapter 2

Policy Gradient Methods

Recall that the goal of reinforcement learning is to solve the following optimization problem:

$$\max_{\pi} V^{\pi}(s_0) := \max_{\pi} \mathbb{E}_{T,R,\pi}\left[\sum_{t=0}^{H-1} r_t\right],$$

where s_0 is a fixed initial state. Policy gradient methods approach this problem by iteratively improving the policy using local gradient-based updates.

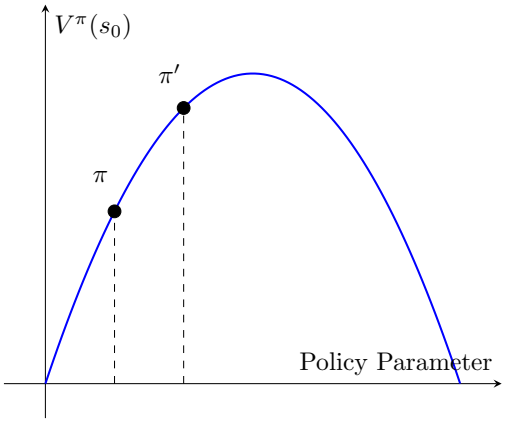

Figure 2.1: Gradient ascent step improves the policy value $V^{\pi}(s_0)$.

Suppose we have a policy π with value $V^{\pi}(s_0)$. Unless π is optimal, there always exists a small perturbation of the policy that leads to a higher expected return. Specifically, for a small change in policy from π to π', we can approximate the change in value as:

$$V^{\pi'}(s_0) \approx V^{\pi}(s_0) + \nabla_{\pi} V^{\pi}(s_0) \cdot (\pi' - \pi),$$

where $\nabla_\pi V^\pi(s_0)$ is the gradient of the value function with respect to the policy parameters. This suggests that we can improve the policy by moving in the direction of the gradient:

$$\pi' = \pi + \alpha \nabla_\pi V^\pi(s_0),$$

where $\alpha > 0$ is a learning rate. This update rule is the foundation of policy gradient methods, which directly optimize the expected return by estimating and following the gradient (see Figure 2.1 for an illustration).

In the rest of this chapter, we introduce a simple gradient-based algorithm, and implement it to solve two environments: CartPole and Pong.

2.1 Environments: CartPole and Pong

In this chapter, we focus on implementing policy gradient methods to solve two benchmark environments in reinforcement learning: **CartPole** and **Pong**. These environments differ in terms of state representation, action space, and the challenges they present for learning-based algorithms. We begin by introducing each environment in detail.

2.1.1 CartPole Environment

The CartPole environment is a classic control problem where the objective is to prevent a pole, hinged on a cart, from falling over. The agent can apply a force of fixed magnitude to the left or right of the cart to balance the pole.

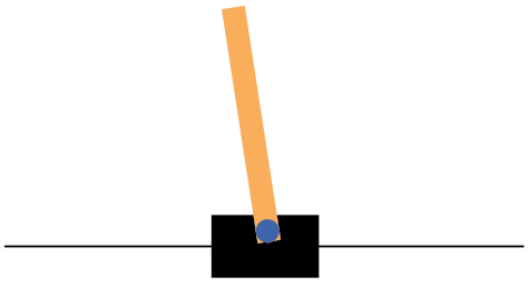

Figure 2.2: Screenshot of the CartPole environment. The agent applies left/right forces to the cart to keep the pole balanced.

- **Observation space:** A 4-dimensional continuous vector:

  $$\text{obs} = (\text{cart position}, \text{cart velocity}, \text{pole angle}, \text{pole angular velocity})$$

  The observation limits are summarized in the table below:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	~ -0.418 rad ($-24°$)	~ 0.418 rad ($24°$)
3	Pole Angular Velocity	$-\infty$	∞

- **Action space:** A discrete set $\{0, 1\}$, where 0 represents pushing the cart to the left, and 1 to the right.
- **Reward:** The agent receives a reward of $+1$ for every timestep that the pole remains upright.
- **Episode termination:** An episode ends when the pole angle exceeds $\pm 24°$ or the cart moves beyond ± 4.8 units from the center or 500 timesteps.

We use the implementation available in the Gymnasium library [1] under the environment name `CartPole-v1`.

2.1.2 Pong Environment

Pong is a visually rich environment based on the Atari 2600 game. The agent controls a paddle and aims to win a game of Pong against a built-in opponent. The objective of the game is to keep deflecting the ball away from your goal and into the opponent's goal.

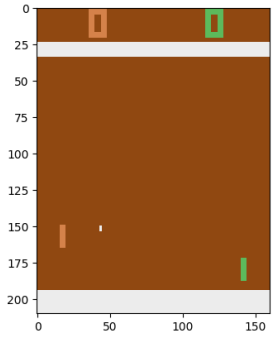

Figure 2.3: Frame from the Pong environment. Green paddle is the agent, orange is the opponent.

- **Observation space:** An RGB image of shape $(210, 160, 3)$ representing the full screen.

- **Action space:** A discrete set of 6 actions, of which we restrict to 3 meaningful ones: `NOOP`, `UP`, and `DOWN`.
- **Reward:** +1 when the agent scores a point, -1 when the agent loses a point, and 0 otherwise.
- **Episode termination:** A game ends when one of the players reaches 21 points.

We use the implementation from the ALE (Arcade Learning Environment) interface in Gymnasium [2], under the environment name `ALE/Pong-v5`.

2.2 Vanilla Policy Gradient Algorithm

In this section, we describe how to solve CartPole and Pong using the policy gradient algorithm with a linear policy. We first formalize the policy representation, then present the algorithm and explain the gradient derivation used to update the policy parameters.

2.2.1 Policy Parameterization

Let the state space be $S \subseteq \mathbb{R}^d$ and the action space be finite, $A = \{1, 2, \ldots, K\}$, where d is the dimension of the observation vector and $K = |A|$ is the number of discrete actions. For CartPole and Pong, (d, K) are $(4, 2)$ and $(210 * 160 * 3 = 100800, 3)$ respectively. We define a policy $\pi_\theta : S \to \Delta(A)$ using a softmax function over a linear function of the state:

$$\pi_\theta(a \mid s) = \frac{\exp(\langle \theta_a, s \rangle)}{\sum_{a' \in A} \exp(\langle \theta_{a'}, s \rangle)},$$

where $\theta = (\theta_1, \ldots, \theta_K)$ is a matrix in $\mathbb{R}^{K \times d}$, and $\theta_a \in \mathbb{R}^d$ is the parameter vector associated with action a. The policy assigns a probability distribution over actions based on the current state.

2.2.2 Algorithm

Our goal is to find a policy π_θ that maximizes expected return from a fixed initial state s_0:

$$\max_\theta V(\theta) \quad \text{where } V(\theta) = \mathbb{E}_{T,R,\pi_\theta}\left[\sum_{t=0}^{H-1} r_t\right].$$

The policy gradient algorithm performs gradient ascent to improve the policy, using the update

$$\theta \to \theta + \nabla_\theta V(\theta)$$

Since, we can only get an estimator for the function $V(\theta)$, we can get an unbiased estimator for the policy gradient: Let $\tau = (s_0, a_0, r_0, s_1, \ldots, s_H)$ be a trajectory generated by following policy π_θ,

$$\nabla_\theta V(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{H-1} \left(\sum_{t=0}^{H-1} r_t \right) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right].$$

We now present the algorithm using this gradient estimate:

Algorithm 1: Vanilla Policy Gradient with Linear Policy

Input: Learning rate α, number of episodes N
Output: Learned policy parameters $\theta \in \mathbb{R}^{K \times d}$

{1} Initialize $\theta \leftarrow 0$
{2} **for** $i = 1$ *to* N **do**
{3} Sample a trajectory $\tau = (s_0, a_0, r_0, \ldots, s_H)$ using policy π_θ
{4} Compute return: $R = \sum_{t=0}^{H-1} r_t$
{5} Update parameters: $\theta \leftarrow \theta + \alpha \cdot \sum_{t=0}^{H-1} R \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
{6} **return** θ

We can replace the total return term $\sum_{t=0}^{H-1} r_t$ with other functions that depend on π_θ, s_t, a_t, such as the state-action value function $Q^{\pi_\theta}(s_t, a_t)$ or the advantage function $A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$. Algorithms like TRPO [3] and PPO [4] use these modifications and typically achieve better performance than the vanilla policy gradient method in practice.

2.3 Implementing in Python

We now describe how to implement the vanilla policy gradient algorithm from Algorithm 1 in Python, using PyTorch and Gymnasium.

2.3.1 Environment Setup and Dependencies

We recommend using a virtual environment to keep dependencies isolated.

Step 1: Create a virtual environment

```
python3 -m venv .venv
source .venv/bin/activate  # on Linux/macOS
.venv\Scripts\activate.bat # on Windows
```

Running the above code for the first time will lead to an error requiring Xcode developer tools, followed by a prompt to install the necessary tools. Since this installation requires a stable internet connection, it's best to complete it a day in advance.

Step 2: Install required packages

```
pip install --upgrade pip
pip install torch
pip install "gymnasium[classic-control]"
```

Step 3: Verify installation Check that PyTorch and Gymnasium work correctly:

```
python -c "import torch; print(torch.__version__)"
python -c "import gymnasium as gym; env = gym.make('CartPole-v1'); print(env)"
```

2.3.2 Training on CartPole

Below is a full implementation using a linear softmax policy for CartPole:

```python
# Libraries to help construct neural networks
# and train them.
import torch
import torch.nn as nn
import torch.optim as optim

# Library to create and interact with environments.
import gymnasium as gym

# Define the policy network
# It first applies a linear mapping from the state space to hidden layer,
# and then applies a softmax activation to obtain the action probabilities.

class LinearPolicy(nn.Module):
    def __init__(self, obs_dim, n_actions):
        super().__init__()
        # This has obs_dim x n_actions many parameters.
        # Useful to inutively see how much data you need to train this.
        self.linear = nn.Linear(obs_dim, n_actions)

    def forward(self, x):
        # Basically if self.linear(x) = [2, 3, -4]
        # softmax(self.linear(x)) will normalize it to  [0.4, 0.5, 0.1]
        # And if we do batch, then x is [batch_size, obs_dim]
        # self.linear(x) will be [batch_size, n_actions]
        # softmax(self.linear(x)) will be [batch_size, n_actions]
        return torch.softmax(self.linear(x), dim=-1)

# Next, we create our Cartpole environment
env = gym.make("CartPole-v1")

# Here is how the observation and action space looks like
obs_dim = env.observation_space.shape[0]
```

```
n_actions = env.action_space.n
print(f"S = R^{obs_dim} and A = {list(range(n_actions))}")

policy = LinearPolicy(obs_dim, n_actions)
optimizer = optim.Adam(policy.parameters(), lr=1e-2)

# We train for 1000 episodes
for episode in range(1000):
    # We get our starting state from the environment
    obs, _ = env.reset()

    # We need to store log probabilities and rewards for computing gradients
    log_probs = []
    rewards = []
    done = False

    # We run our current policy until the end of episode
    while not done:
        # Turn our state into a tensor (because nn uses tensors)
        obs_tensor = torch.tensor(obs, dtype=torch.float32)

        # Compute action probabilities under our policy
        # Sample an action.
        action_probs = policy(obs_tensor)
        dist = torch.distributions.Categorical(action_probs)
        action = dist.sample()

        # Execute the action in the environment, collect reward, new state
        obs, reward, terminated, truncated, _ = env.step(action.item())

        # We need log probabilities and rewards to compute gradient, so lets save that
        log_probs.append(dist.log_prob(action))
        rewards.append(reward)

        done = terminated or truncated

    # We need to turn rewards into tensors before using them.
    rewards = torch.tensor(rewards, dtype=torch.float32)

    # Compute the loss to propagate the gradients
    loss = -sum(log_probs)*sum(rewards)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if episode % 50 == 0:
        print(f"Episode {episode}, return = {sum(rewards)}")
```

2.3.3 Visualizing the Learned CartPole Policy

To play back the trained policy:

```
env = gym.make("CartPole-v1", render_mode="human")

for episode in range(5):
    obs, _ = env.reset()
    done = False
    while not done:
        obs_tensor = torch.tensor(obs, dtype=torch.float32)
        action_probs = policy(obs_tensor)
        action = torch.argmax(action_probs).item()
        obs, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated

env.close()
```