Policy Gradient: Implementation via GridWorld

Thuong Dang

December 2025

Contents

1 Introduction

In this project, we implement the REINFORCE algorithm to train an agent to navigate a gridworld environment. Unlike Q-learning, which learns action values for specific state-action pairs, policy gradient methods directly learn a parameterized policy $\pi_\theta(a \mid s)$ that can generalize across different grid configurations. The gridworld is an 8×8 grid where:

- The **agent** starts at a designated position and must reach the **goal** position
- **Obstacles** (walls) are randomly placed throughout the grid
- The agent can move in four directions: up, down, left, right
- Episodes terminate when the agent reaches the goal or exceeds the maximum step limit

The environment provides the following rewards:

- $+10$ for reaching the goal
- -1 for each step taken (encourages shorter paths)
- -5 for attempting to move into a wall

Our objective is to train a policy network π_θ that:

1. Successfully navigates from start to goal on a fixed set of training grids
2. Generalizes to new, unseen grid configurations
3. Learns to avoid walls and find efficient paths

This document is organized as follows: Section 2 provides a high-level overview of the training process, Section 3 details the REINFORCE algorithm implementation. The GridWorld environment and observation representation will be discussed in Section 4 and Section 5 describes the policy network architecture. In Section 6, we present evaluation results on both training and test grids and in Section 7, we visualize and test the agent interactively.

2 High-level overview

The training process consists of three main steps:

1. **Initialize fixed training grids**: Generate a set of diverse grid configurations that will be used throughout training
2. **Initialize policy network**: Create a neural network π_θ that maps observations to action probabilities

3. **Training loop**: For each update iteration:
   (a) Generate trajectories using the current policy (trajectories not reaching the goal are allowed)
   (b) Compute G_t, the discounted return at each timestep
   (c) Compute the policy gradient using the REINFORCE estimator
   (d) Update network parameters via gradient ascent

Sections 3-5 expand on these steps in detail.

3 REINFORCE algorithm

Because the REINFORCE algorithm is the heart of the project, we will discuss it first before going to the policy network setup. Note that the choice of the policy network is not very essential, as we can use CNN, MLP, or transformers. The most important role of the policy network is to generate the probability distribution of actions given the current state. More details will be discussed in the next section. For now, we assume its existence and its outputs given inputs.

3.1 Assumptions and Prerequisites

The *training grids* will be randomly generated before training. During each episode, we randomly sample one grid from this fixed set. Each grid consists of:

- Obstacles
- Start position
- Goal position

We also assume the existence of the *policy network* π_θ, where θ is the parameter of the neural network. The policy network provides a method

```
a, logp, ent = policy.act(obs),
```

where

1. The input `obs` is the current state representation. In Section 4, we will discuss this, but it can be understood as the agent's current position and the current board.
2. The output `a` is the sampled action from $\pi_\theta(\cdot \mid s)$
3. The output `logp` is the log probability of the sampled action
4. The output `ent` is the entropy of the policy distribution. Note that it is not part of the REINFORCE estimator. We will discuss this later.

For the *environment*, the environment provides

`obs, reward, done, info = env.step(action)`,

where `obs`, `reward`, and `info` are the next state representation, reward, and information about the grid after taking `action`. Furthermore, `done` is a boolean to check if the game is finished or the step limit is reached.

3.2 Trajectory collection

To update the parameters of the policy network with the REINFORCE algorithm and Monte Carlo method, we use the policy network to create trajectories in the training grid set. More precisely, we will collect:

- A trajectory consists of the sequence $(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_T, a_T, r_T)$
- We store rewards (to compute discounted returns), log probabilities (to update the neural network's parameters with REINFORCE), and entropies (for exploration).
- We collect `batch_episodes` trajectories before performing one parameter update.

Here is the implementation code:

```
for _ in range(batch_episodes):  # batch_episodes = 32
    # Load a training grid
    obs = env.reset()
    done = False

    ep_logps = []
    ep_rewards = []
    ep_ents = []

    # Rollout one episode
    while not done:
        a, logp, ent = policy.act(obs)
        next_obs, r, done, _ = env.step(a)

        ep_logps.append(logp)
        ep_rewards.append(r)
        ep_ents.append(ent)

        obs = next_obs
```

3.3 Discounted Return G_t

After collecting rewards via the trajectory $\tau = (s_0, a_0, r_0) \ldots (s_T, a_T, r_T)$, we compute the discounted return from each timestep:

$$G_t = G_t(\tau) = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t} r_T,$$

or recursively,

$$G_t = r_t + \gamma G_{t+1},$$

where $\gamma = 0.99$ is the discount factor.

3.4 Computing the Policy Gradient

Recall that in the discounted reward setting, we can use the Monte Carlo method to update the parameters by the formula

$$\nabla_\theta J_\gamma(\theta) \approx \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t(\tau).$$

In batch updates with N trajectories, we have

$$\nabla_\theta J_\gamma(\theta) \approx \frac{1}{N} \sum_{\tau} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t(\tau) \right).$$

In practice, to avoid premature convergence, we add entropy regularization. Its role is analogous to ϵ-greedy in Q-learning, but works differently: instead of explicitly scheduling exploration, it encourages the policy to maintain stochasticity throughout training. It also prevents the policy network from getting stuck in local optima. The objective in this case is

$$J_{\text{total}} = J_{\text{REINFORCE}} + \alpha H(\pi_\theta),$$

where $\alpha = 0.01$ is the `entropy_coef`. In Q-learning, at early stages, we want to explore, and at later stages, we want to exploit. These actions can be manually controlled by decreasing ϵ for randomly chosen actions. In policy gradient, the `entropy_coef` is set to be small and fixed, and it provides *automatic adaptation*.

How does it naturally shift from exploration to exploitation? At early stages, the network produces actions with a nearly uniform distribution over all actions, so the entropy is high. The entropy term therefore encourages the policy to remain stochastic while the REINFORCE term starts to shape the policy toward rewarding actions. Over time, the network learns via the REINFORCE estimator, and the policy becomes more decisive. The entropy naturally decreases, but this regularization term prevents the policy from collapsing too quickly, allowing the policy to retain some stochasticity and continue exploring when needed.

We can implement the loss as follows:

```
# Concatenate data from all episodes in the batch
returns_t = torch.cat(all_returns)      # All G_t values
logps_t = torch.stack(all_logps)        # All log pi_theta(a_t|s_t)
ents_t = torch.stack(all_ents)          # All H(pi_theta(·|s_t))

# Standardize returns for training stability
```

```
returns_t = (returns_t - returns_t.mean()) / (returns_t.std() + 1e-8)

# REINFORCE policy gradient estimator
policy_loss = -(logps_t * returns_t).mean()

# Entropy regularization (NOT part of REINFORCE)
entropy_bonus = ents_t.mean()

# Total loss
loss = policy_loss - entropy_coef * entropy_bonus
```

3.5 Parameter Update

Finally, we update the policy network parameters using gradient descent:

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

Where:

1. `zero_grad()`: Clear any existing gradients
2. `backward()`: Compute ∇_θloss via backpropagation
3. `step()`: Update parameters $\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_\theta \text{loss}$, where η is the learning rate

This update increases the probability of actions that led to high returns and decreases the probability of actions that led to low returns.

4 GridWorld Environment

The GridWorld environment is an 8×8 grid navigation task where an agent must reach a goal position while avoiding obstacles. The environment is implemented to support training on multiple grid configurations with varying obstacle placements.

4.1 Grid Configuration

Each grid configuration consists of:

- **Grid size**: 8×8 cells
- **Obstacles**: Randomly placed with 20% probability
- **Start position**: Randomly sampled from free (non-obstacle) cells
- **Goal position**: Randomly sampled from free cells (different from start)

- **Connectivity guarantee**: A breadth-first search (BFS) ensures that a wall-free path exists from start to goal

The connectivity guarantee is important because although the agent *can* move through obstacles (with a heavy penalty), we ensure that every grid has at least one obstacle-free solution. This allows the agent to learn to avoid obstacles rather than being forced through them.

4.2 Action Space

The agent has four discrete actions (UP, DOWN, LEFT, RIGHT). The agent can attempt any action at any time. The environment handles three cases:

1. **Out of bounds**: If the action would move the agent outside the 8×8 grid, the agent stays in place and receives the step penalty.
2. **Move into obstacle**: The agent moves to the obstacle cell and receives the wall penalty. Unlike traditional gridworlds, obstacles do not block movement but impose a cost.
3. **Move to empty cell**: The agent moves to the new position and receives the step penalty.

4.3 Reward Structure

The environment provides the following rewards. Note that these rewards are used only during training to prevent large total rewards. For human-friendly view, we $\times 10$ the rewards.

- $+1.0$: Reaching the goal position
- -0.1: Each step taken on an empty cell
- -0.5: Moving through an obstacle cell

The step penalty encourages the agent to find shorter paths. The large obstacle penalty teaches the agent to avoid obstacles when possible. When the agent reaches the goal, the goal reward overrides any step or wall penalty for that transition.

4.4 Observation Representation

The observation is a 3-dimensional tensor of shape $(3, 8, 8)$ with binary values

```
obs = np.stack([walls, agent, goal], axis=0)
```

and it consists of

- **Channel 0 (walls)**: Binary mask where 1.0 indicates an obstacle, 0.0 indicates free space

- **Channel 1 (agent)**: Binary mask with 1.0 at the agent's current position, 0.0 elsewhere
- **Channel 2 (goal)**: Binary mask with 1.0 at the goal position, 0.0 elsewhere

Why three separate channels? Our choice for policy network is CNN, which is stateless, each call to `policy.act(obs)` receives only the current observation without memory of previous states or knowledge of which grid configuration is being used. Therefore, each observation must be self-contained and provide all information needed for decision-making.

We use separate binary channels rather than a single channel with different values (e.g., 0=empty, 1=wall, 2=agent, 3=goal) because CNNs naturally process multi-channel inputs where each channel has clear semantic meaning. Each convolutional filter processes all input channels simultaneously, making it easier to learn spatial relationships like "agent near goal" or "agent near obstacle" when these entities are in separate channels.

4.5 Rendering

While observation representation is machine-friendly, our `render()` method is human-friendly view of the grid. Its output is something like this

```
#.#....#
..#..A..
##...#..
....#..G
.#......
```

4.6 Episode Termination

An episode terminates when either:

- **Success**: The agent reaches the goal position
- **Timeout**: The agent exceeds the maximum number of steps (64 steps)

4.7 Training Grid Generation

Before training begins, we generate a fixed set of 400 training grids. Each grid is generated using `reset()` and stored with its obstacle configuration, start position, and goal position. During training, episodes randomly sample from this fixed set rather than generating new grids each time.

5 Policy Network

Our choice of policy network is CNN. It is standard for board games, or more general, 2D RL environment. The reason is the environment is spatial

- Objects have locations
- Relative positions matter (agent vs obstacles, agent vs goal)
- Actions depend on local pattern near the agent

The network is quite simple, with two convolutional layers, a flattern layer, and two linear layers at the end. Our input is of shape `(B, 3, 8, 8)` where B is the batch size, 8×8 is the board side, and we have 3 channels as discussed in Section 4.4. The output is of shape `(B, 4)` for the probability of the four actions with batch size `B`.

The key method of this class is `act()`. Its main input is the observation `obs`. The method converts the raw logits into a categorical action distribution $\pi_\theta(a \mid s)$, samples an action $a \sim \pi_\theta(a \mid s)$, and returns the sampled action, its log-probability, and the entropy of the policy distribution. These terms are required for REINFORCE algorithm in Section 3.

For evaluation, we choose action $a = \text{argmax}_a \, \pi(a \mid s)$.

6 Evaluation

We evaluate the trained policy on two levels: continuous monitoring during training and final generalization testing.

6.1 Training Progress Monitoring

During training, we track a rolling success rate to monitor learning progress if the agent can reach the goal:

```
# After each batch of episodes
batch_success = []
for each episode in batch:
    success = 1.0 if (env.agent == env.goal) else 0.0
    batch_success.append(success)

running_success.extend(batch_success)

# Log every 10 updates
if upd % 10 == 0:
    avg_success_recent = mean(running_success[-320:])
    print(f"avg_success_recent={avg_success_recent:.3f}")
```

This metric computes the success rate over the last 320 training episodes (10 updates $\times$ 32 episodes per update).

6.2 Evaluation Protocol

The generalization evaluation differs from training in several key ways:

Grid Generation:

- Use a different random seed (`seed + 1000`) to generate grids
- Each evaluation episode uses a freshly generated grid
- These grids have never been seen during training
- Same generation parameters (8×8 size, 20% obstacle probability, connectivity guarantee)

Action Selection:

- **Training**: Actions sampled stochastically from $\pi_\theta(a|s)$ for exploration as discussed in Section 3
- **Evaluation**: Actions selected greedily: $a = \text{argmax}_a \, \pi_\theta(a|s)$ as discussed in Section 5
- Action masking is applied to prevent out-of-bounds moves

6.3 Evaluation Metrics

Primary Metric - Success Rate:

An episode is considered successful if the agent reaches the goal position before exceeding the maximum step limit (64 steps):

```
success = (env.agent == env.goal)
success_rate = (successful episodes) / (total episodes)
```

With the reward structure (+1.0 for goal, −0.1 per step, −0.5 per wall), even optimal paths result in negative total rewards. For example, a 20-step path with no wall hits yields $1.0 - 0.1 \times 20 = -1.0$. Therefore, we use goal achievement rather than a reward threshold as the success criterion.

Secondary Metrics:

For successful episodes, we also compute:

- **Average path length**: Mean number of steps to reach the goal. Shorter paths indicate more efficient navigation.
- **Average wall hits**: Mean number of obstacle collisions per episode. Fewer collisions indicate better obstacle avoidance.

These metrics are only computed over successful episodes, as they are not meaningful for episodes that fail to reach the goal.

6.4 Results

We evaluate the trained policy on 100 randomly generated grids. The results are:

```
Success rate: 79.00%
Average path length: 4.6 steps
Average wall hits: 0.8
```

7 Visualization and Interactive Testing

To observe how the trained agent performs on actual grids in real-time, we provide a visualization script `test_rl.py` that renders the agent's navigation step-by-step. The testing script follows the same evaluation protocol as the generalization test:

- **Action selection**: Greedy with $a = \text{argmax}_a \pi_\theta(a|s)$
- **Action masking**: Applied to prevent out-of-bounds moves
- **Grid generation**: New random grids using a different seed (`seed = 101`, different from training `seed = 42`)
- **Visualization**: ASCII rendering after each step with configurable delay

7.1 Key Features

The test script provides:

- **Step-by-step visualization**: Displays the grid after each action with the agent's position, chosen action, and reward received
- **Move validation**: Shows whether the agent successfully moved or stayed in place (e.g., boundary collision)
- **Episode summary**: Reports success/failure, total steps, and cumulative reward
- **Aggregate statistics**: Success rate, average steps, and average reward across all test episodes

7.2 Example Output

Here is a full test run:

```
================================================
EPISODE 5/5
================================================
```

```
Initial Grid:
..#....#
####....
....##.#
.#..G#..
...#....
........
...##...
..#...A.

Agent: A | Goal: G | Wall: # | Empty: .
Start: (7, 6) | Goal: (3, 4)


Step 1: Action = UP | Reward = -1.00 | Position = (6, 6) | moved
..#....#
####....
....##.#
.#..G#..
...#....
........
...##.A.
..#.....

Step 2: Action = UP | Reward = -1.00 | Position = (5, 6) | moved
..#....#
####....
....##.#
.#..G#..
...#....
......A.
...##...
..#.....

Step 3: Action = LEFT | Reward = -1.00 | Position = (5, 5) | moved
..#....#
####....
....##.#
.#..G#..
...#....
.....A..
...##...
..#.....

Step 4: Action = UP | Reward = -1.00 | Position = (4, 5) | moved
..#....#
```

```
####....
....##.#
.#..G#..
...#.A..
........
...##...
..#.....

Step 5: Action = LEFT | Reward = -1.00 | Position = (4, 4) | moved
..#....#
####....
....##.#
.#..G#..
...#A...
........
...##...
..#.....

Step 6: Action = UP | Reward = 10.00 | Position = (3, 4) | moved
..#....#
####....
....##.#
.#..A#..
...#....
........
...##...
..#.....

SUCCESS! Reached goal in 6 steps!
Total Episode Reward: 5.00

==================================================
FINAL STATISTICS
==================================================
Success Rate: 5/5 (100.0%)
Average Steps: 4.8 ± 2.1
Average Reward: 6.20 ± 2.14
Average Steps (successful episodes only): 4.8
```

7.3 Observations from Testing

During testing with `seed = 101` (different from the training seed), the agent demonstrated:

- **Perfect success rate**: Successfully navigated all 5 test grids to the goal
- **Obstacle avoidance**: Did not collide with any walls, achieving optimal

paths

- **Efficient pathfinding**: Average path length of 4.8 steps indicates the agent learned to find short routes
- **Robust generalization**: Handled various grid layouts with different obstacle configurations