ISIA 2025/2026 - PL 2
Gymnasium QuickStart

---

1 Gymnasium Quick Start

adapted from Gymnasium documentation, available at https://gymnasium.farama.org/

1.1 Basic Usage

1.1.1 What is Reinforcement Learning?

Before diving into Gymnasium, let's understand what we're trying to achieve. Reinforcement learning is like teaching through trial and error - an agent learns by trying actions, receiving feedback (rewards), and gradually improving its behavior. Think of training a pet with treats, learning to ride a bike through practice, or mastering a video game by playing it repeatedly.

The key insight is that we don't tell the agent exactly what to do. Instead, we create an environment where it can experiment safely and learn from the consequences of its actions.

1.1.2 Why Gymnasium?

Whether you want to train an agent to play games, control robots, or optimize trading strategies, Gymnasium gives you the tools to build and test your ideas. At its heart, Gymnasium provides an API (application programming interface) for all single agent reinforcement learning environments, with implementations of common environments: `cartpole`, `pendulum`, `mountain-car`, `mujoco`, `atari`, and more. This page will outline the basics of how to use Gymnasium including its four key functions: `make()`, `Env.reset()`, `Env.step()` and `Env.render()`.

At the core of Gymnasium is `Env`, a high-level python class representing a markov decision process (MDP) from reinforcement learning theory (note: this is not a perfect reconstruction, missing several components of MDPs). The class provides users the ability start new episodes, take actions and visualize the agent's current state. Alongside `Env`, `Wrapper` are provided to help augment / modify the environment, in particular, the agent observations, rewards and actions taken.

1.1.3 Installation

To install the base Gymnasium library, use `pip install gymnasium`.

This does not include dependencies for all families of environments (there's a massive number, and some can be problematic to install on certain systems). You can install these dependencies for one family like `pip install gymnasium[atari]` or use `pip install gymnasium[all]` to install all dependencies.

1.1.4 Environments

Gymnasium includes the following families of environments along with a wide variety of third-party environments

- Classic Control - These are classic reinforcement learning based on real-world problems and physics.
- Box2D - These environments all involve toy games based around physics control, using box2d based physics and PyGame-based rendering
- Toy Text - These environments are designed to be extremely simple, with small discrete state and action spaces, and hence easy to learn. As a result, they are suitable for debugging implementations of reinforcement learning algorithms.
- MuJoCo - A physics engine based environments with multi-joint control which are more complex than the Box2D environments.
- Atari - Emulator of Atari 2600 ROMs simulated that have a high range of complexity for agents to learn.
- Third-party - A number of environments have been created that are compatible with the Gymnasium API. Be aware of the version that the software was created for and use the apply_env_compatibility in gymnasium.make if necessary.

1.1.5 Initializing Environments

Initializing environments is very easy in Gymnasium and can be done via the make() function:

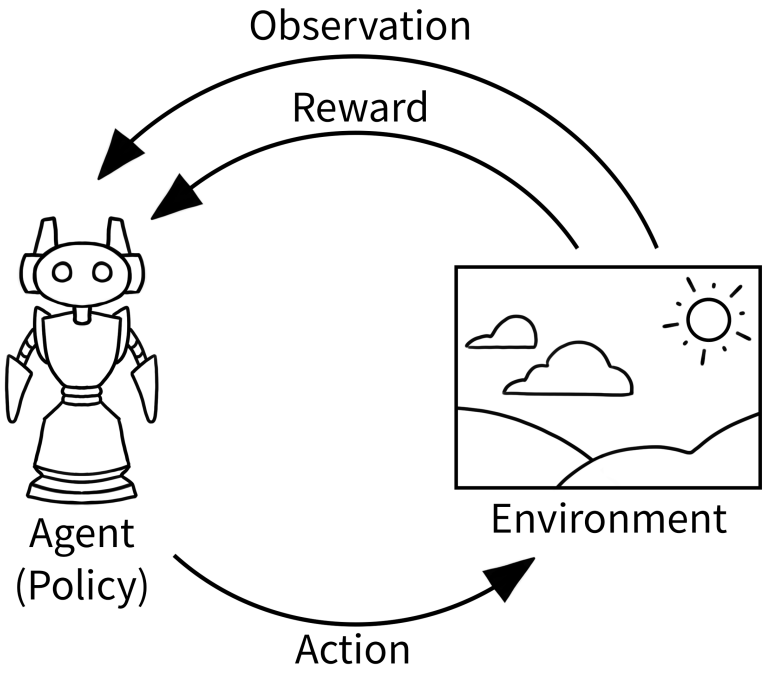
[ ]:
```
import gymnasium as gym

# Create a simple environment perfect for beginners
env = gym.make('CartPole-v1')

# The CartPole environment: balance a pole on a moving cart
# - Simple but not trivial
# - Fast training
# - Clear success/failure criteria
```

This function will return an Env for users to interact with. To see all environments you can create, use `pprint_registry()`. Furthermore, `make()` provides a number of additional arguments for specifying keywords to the environment, adding more or less wrappers, etc. See `make()` for more information. Understanding the Agent-Environment Loop

In reinforcement learning, the classic "agent-environment loop" pictured below represents how learning happens in RL. It's simpler than it might first appear:

1. Agent observes the current situation (like looking at a game screen)
2. Agent chooses an action based on what it sees (like pressing a button)
3. Environment responds with a new situation and a reward (game state changes, score updates)
4. Repeat until the episode ends

This might seem simple, but it's how agents learn everything from playing chess to controlling robots to optimizing business processes.



1.1.6 Print All Registered Environments

To print all registered environments in Farama Gymnasium, you can use the `gymnasium.pprint_registry()` function, which pretty-prints all the available environments in the registry.

[ ]:
```
import gymnasium as gym
import ale_py

gym.pprint_registry()
```

Alternatively, you can directly list the environment IDs with the following code:

[ ]:
```
import gymnasium as gym
import ale_py

for env_id in gym.envs.registry.keys():
    print(env_id)
```

1.1.7 Your First RL Program

Let's start with a simple example using CartPole - perfect for understanding the basics:

[ ]:
```
# Run `pip install "gymnasium[classic-control]"` for this example.
import gymnasium as gym

# Create our training environment - a cart with a pole that needs balancing
```

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

# Reset environment to start a new episode
observation, info = env.reset()
# observation: what the agent can "see" - cart position, velocity, pole angle, etc.
# info: extra debugging information (usually not needed for basic learning)

print(f"Starting observation: {observation}")
# Example output: [ 0.01234567 -0.00987654  0.02345678  0.01456789]
# [cart_position, cart_velocity, pole_angle, pole_angular_velocity]

episode_over = False
total_reward = 0

while not episode_over:
    # Choose an action: 0 = push cart left, 1 = push cart right
    action = env.action_space.sample()  # Random action for now - real agents will be smarter!

    # Take the action and see what happens
    observation, reward, terminated, truncated, info = env.step(action)

    # reward: +1 for each step the pole stays upright
    # terminated: True if pole falls too far (agent failed)
    # truncated: True if we hit the time limit (500 steps)

    total_reward += reward
    episode_over = terminated or truncated

print(f"Episode finished! Total reward: {total_reward}")
env.close()
```

What you should see: A window opens showing a cart with a pole. The cart moves randomly left and right, and the pole eventually falls over. This is expected - the agent is acting randomly!

Explaining the Code Step by Step First, an environment is created using make() with an optional "render_mode" parameter that specifies how the environment should be visualized. See Env.render() for details on different render modes. The render mode determines whether you see a visual window ("human"), get image arrays ("rgb_array"), or run without visuals (None - fastest for training).

After initializing the environment, we Env.reset() the environment to get the first observation along with additional information. This is like starting a new game or episode. For initializing the environment with a particular random seed or options (see the environment documentation for possible values) use the seed or options parameters with reset().

As we want to continue the agent-environment loop until the environment ends (which happens in an unknown number of timesteps), we define episode_over as a variable to control our while loop.

Next, the agent performs an action in the environment. Env.step() executes the selected action (in our example, random with env.action_space.sample()) to update the environment. This action can be

imagined as moving a robot, pressing a button on a game controller, or making a trading decision. As a result, the agent receives a new observation from the updated environment along with a reward for taking the action. This reward could be positive for good actions (like successfully balancing the pole) or negative for bad actions (like letting the pole fall). One such action-observation exchange is called a timestep.

However, after some timesteps, the environment may end - this is called the terminal state. For instance, the robot may have crashed, or succeeded in completing a task, or we may want to stop after a fixed number of timesteps. In Gymnasium, if the environment has terminated due to the task being completed or failed, this is returned by step() as terminated=True. If we want the environment to end after a fixed number of timesteps (like a time limit), the environment issues a truncated=True signal. If either terminated or truncated are True, we end the episode. In most cases, you'll want to restart the environment with env.reset() to begin a new episode.

1.1.8 Action and observation spaces

Every environment specifies the format of valid actions and observations with the action_space and observation_space attributes. This is helpful for knowing both the expected input and output of the environment, as all valid actions and observations should be contained within their respective spaces. In the example above, we sampled random actions via env.action_space.sample() instead of using an intelligent agent policy that maps observations to actions (which is what you'll learn to build).

Understanding these spaces is crucial for building agents: - Action Space: What can your agent do? (discrete choices, continuous values, etc.) - Observation Space: What can your agent see? (images, numbers, structured data, etc.)

Importantly, Env.action_space and Env.observation_space are instances of Space, a high-level python class that provides key functions: Space.contains() and Space.sample(). Gymnasium supports a wide range of spaces:

- Box: describes bounded space with upper and lower limits of any n-dimensional shape (like continuous control or image pixels).
- Discrete: describes a discrete space where {0, 1, ..., n-1} are the possible values (like button presses or menu choices).
- MultiBinary: describes a binary space of any n-dimensional shape (like multiple on/off switches).
- MultiDiscrete: consists of a series of Discrete action spaces with different numbers of actions in each element.
- Text: describes a string space with minimum and maximum length.
- Dict: describes a dictionary of simpler spaces (like our GridWorld example you'll see later).
- Tuple: describes a tuple of simple spaces.
- Graph: describes a mathematical graph (network) with interlinking nodes and edges.
- Sequence: describes a variable length of simpler space elements.

For example usage of spaces, see their documentation along with utility functions.

Let's look at some examples:

[ ]:
```python
import gymnasium as gym

# Discrete action space (button presses)
env = gym.make("CartPole-v1")
print(f"Action space: {env.action_space}")  # Discrete(2) - left or right
print(f"Sample action: {env.action_space.sample()}")  # 0 or 1

# Box observation space (continuous values)
print(f"Observation space: {env.observation_space}")  # Box with 4 values
# Box([-4.8, -inf, -0.418, -inf], [4.8, inf, 0.418, inf])
print(f"Sample observation: {env.observation_space.sample()}")  # Random valid observation
```

1.1.9 Modifying the environment

Wrappers are a convenient way to modify an existing environment without having to alter the underlying code directly. Think of wrappers like filters or modifiers that change how you interact with an environment. Using wrappers allows you to avoid boilerplate code and make your environment more modular. Wrappers can also be chained to combine their effects.

Most environments created via gymnasium.make() will already be wrapped by default using TimeLimit (stops episodes after a maximum number of steps), OrderEnforcing (ensures proper reset/step order), and PassiveEnvChecker (validates your environment usage).

To wrap an environment, you first initialize a base environment, then pass it along with optional parameters to the wrapper's constructor:

[ ]:
```python
import gymnasium as gym
from gymnasium.wrappers import FlattenObservation

# Start with a complex observation space
env = gym.make("CarRacing-v3")

env.observation_space.shape
# (96, 96, 3)  # 96x96 RGB image
```

[ ]:
```python
# Wrap it to flatten the observation into a 1D array
wrapped_env = FlattenObservation(env)

wrapped_env.observation_space.shape
# (27648,)  # All pixels in a single array

# This makes it easier to use with some algorithms that expect 1D input
```

Common wrappers that beginners find useful:

- TimeLimit: Issues a truncated signal if a maximum number of timesteps has been exceeded (preventing infinite episodes).
- ClipAction: Clips any action passed to step to ensure it's within the valid action space.

- RescaleAction: Rescales actions to a different range (useful for algorithms that output actions in [-1, 1] but environment expects [0, 10]).
- TimeAwareObservation: Adds information about the current timestep to the observation (sometimes helps with learning).

For a full list of implemented wrappers in Gymnasium, see wrappers.

If you have a wrapped environment and want to access the original environment underneath all the layers of wrappers (to manually call a function or change some underlying aspect), you can use the unwrapped attribute. If the environment is already a base environment, unwrapped just returns itself.

[ ]:
```
wrapped_env
# <FlattenObservation<TimeLimit<OrderEnforcing<PassiveEnvChecker<CarRacing<CarRacing-v3>>>>>>
```

[ ]:
```
wrapped_env.unwrapped
# <gymnasium.envs.box2d.car_racing.CarRacing object at 0x7f04efcb8850>
```

1.1.10 Common Issues for Beginners

Agent Behavior:

- Agent performs randomly: That's expected when using env.action_space.sample()! Real learning happens when you replace this with an intelligent policy
- Episodes end immediately: Check if you're properly handling the reset between episodes

Common Code Mistakes:

```
# ✖ Wrong - forgetting to reset
env = gym.make("CartPole-v1")
obs, reward, terminated, truncated, info = env.step(action)  # Error!

# ✔ Correct - always reset first
env = gym.make("CartPole-v1")
obs, info = env.reset()  # Start properly
obs, reward, terminated, truncated, info = env.step(action)  # Now this works
```

1.2 Training an Agent

When we talk about training an RL agent, we're teaching it to make good decisions through experience. Unlike supervised learning where we show examples of correct answers, RL agents learn by trying different actions and observing the results. It's like learning to ride a bike - you try different movements, fall down a few times, and gradually learn what works.

The goal is to develop a **policy** - a strategy that tells the agent what action to take in each situation to maximize long-term rewards.

1.2.1 Understanding Q-Learning Intuitively

For this tutorial, we'll use Q-learning to solve the Blackjack environment. But first, let's understand how Q-learning works conceptually.

Q-learning builds a giant "cheat sheet" called a Q-table that tells the agent how good each action is in each situation:

- **Rows** = different situations (states) the agent can encounter
- **Columns** = different actions the agent can take
- **Values** = how good that action is in that situation (expected future reward)

For Blackjack:

- **States**: Your hand value, dealer's showing card, whether you have a usable ace
- **Actions**: Hit (take another card) or Stand (keep current hand)
- **Q-values**: Expected reward for each action in each state

The Learning Process

1. Try an action and see what happens (reward + new state)
2. Update your cheat sheet: "That action was better/worse than I thought"
3. Gradually improve by trying actions and updating estimates
4. Balance exploration vs exploitation: Try new things vs use what you know works

Why it works: Over time, good actions get higher Q-values, bad actions get lower Q-values. The agent learns to pick actions with the highest expected rewards.

1.2.2 About the Environment: Blackjack

Blackjack is one of the most popular casino card games and is perfect for learning RL because it has:

- Clear rules: Get closer to 21 than the dealer without going over
- Simple observations: Your hand value, dealer's showing card, usable ace
- Discrete actions: Hit (take card) or Stand (keep current hand)
- Immediate feedback: Win, lose, or draw after each hand

This version uses infinite deck (cards drawn with replacement), so card counting won't work - the agent must learn optimal basic strategy through trial and error.

Environment Details:

- Observation: (player_sum, dealer_card, usable_ace)
  - player_sum: Current hand value (4-21)
  - dealer_card: Dealer's face-up card (1-10)
  - usable_ace: Whether player has usable ace (True/False)
- Actions: 0 = Stand, 1 = Hit
- Rewards: +1 for win, -1 for loss, 0 for draw
- Episode ends: When player stands or busts (goes over 21)

1.2.3 Executing an action

After receiving our first observation from env.reset(), we use env.step(action) to interact with the environment. This function takes an action and returns five important values:

```
observation, reward, terminated, truncated, info = env.step(action)
```

- observation: What the agent sees after taking the action (new game state)
- reward: Immediate feedback for that action (+1, -1, or 0 in Blackjack)
- terminated: Whether the episode ended naturally (hand finished)
- truncated: Whether episode was cut short (time limits - not used in Blackjack)
- info: Additional debugging information (can usually be ignored)

The key insight is that reward tells us how good our immediate action was, but the agent needs to learn about long-term consequences. Q-learning handles this by estimating the total future reward, not just the immediate reward.

1.2.4 Building a Q-Learning Agent

Let's build our agent step by step. We need functions for:

1. Choosing actions (with exploration vs exploitation)
2. Learning from experience (updating Q-values)
3. Managing exploration (reducing randomness over time)

Exploration vs Exploitation This is a fundamental challenge in RL:

- Exploration: Try new actions to learn about the environment
- Exploitation: Use current knowledge to get the best rewards

We use epsilon-greedy strategy:

- With probability epsilon: choose a random action (explore)
- With probability 1-epsilon: choose the best known action (exploit)

Starting with high epsilon (lots of exploration) and gradually reducing it (more exploitation as we learn) works well in practice.

[ ]:
```python
from collections import defaultdict
import numpy as np
import gymnasium as gym


class BlackjackAgent:
    def __init__(
            self,
            env: gym.Env,
            learning_rate: float,
```

```
        initial_epsilon: float,
        epsilon_decay: float,
        final_epsilon: float,
        discount_factor: float = 0.95,
    ):
        """Initialize a Q-Learning agent.

        Args:
            env: The training environment
            learning_rate: How quickly to update Q-values (0-1)
            initial_epsilon: Starting exploration rate (usually 1.0)
            epsilon_decay: How much to reduce epsilon each episode
            final_epsilon: Minimum exploration rate (usually 0.1)
            discount_factor: How much to value future rewards (0-1)
        """
        self.env = env

        # Q-table: maps (state, action) to expected reward
        # defaultdict automatically creates entries with zeros for new states
        self.q_values = defaultdict(lambda: np.zeros(env.action_space.n))

        self.lr = learning_rate
        self.discount_factor = discount_factor  # How much we care about future rewards

        # Exploration parameters
        self.epsilon = initial_epsilon
        self.epsilon_decay = epsilon_decay
        self.final_epsilon = final_epsilon

        # Track learning progress
        self.training_error = []

    def get_action(self, obs: tuple[int, int, bool]) -> int:
        """Choose an action using epsilon-greedy strategy.

        Returns:
            action: 0 (stand) or 1 (hit)
        """
        # With probability epsilon: explore (random action)
        if np.random.random() < self.epsilon:
            return self.env.action_space.sample()

        # With probability (1-epsilon): exploit (best known action)
        else:
            return int(np.argmax(self.q_values[obs]))

    def update(
```

```
            self,
            obs: tuple[int, int, bool],
            action: int,
            reward: float,
            terminated: bool,
            next_obs: tuple[int, int, bool],
    ):
        """Update Q-value based on experience.

        This is the heart of Q-learning: learn from (state, action, reward, next_state)
        """
        # What's the best we could do from the next state?
        # (Zero if episode terminated - no future rewards possible)
        future_q_value = (not terminated) * np.max(self.q_values[next_obs])

        # What should the Q-value be? (Bellman equation)
        target = reward + self.discount_factor * future_q_value

        # How wrong was our current estimate?
        temporal_difference = target - self.q_values[obs][action]

        # Update our estimate in the direction of the error
        # Learning rate controls how big steps we take
        self.q_values[obs][action] = (
                self.q_values[obs][action] + self.lr * temporal_difference
        )

        # Track learning progress (useful for debugging)
        self.training_error.append(temporal_difference)

    def decay_epsilon(self):
        """Reduce exploration rate after each episode."""
        self.epsilon = max(self.final_epsilon, self.epsilon - self.epsilon_decay)
```

Understanding the Q-Learning Update The core learning happens in the update method. Let's break down the math:

```
# Current estimate: Q(state, action)
current_q = self.q_values[obs][action]

# What we actually experienced: reward + discounted future value
target = reward + self.discount_factor * max(self.q_values[next_obs])

# How wrong were we?
error = target - current_q

# Update estimate: move toward the target
new_q = current_q + learning_rate * error
```

This is the famous Bellman equation in action - it says the value of a state-action pair should equal the immediate reward plus the discounted value of the best next action.

1.2.5 Training the Agent

Now let's train our agent. The process is:

1. Reset environment to start a new episode
2. Play one complete hand (episode), choosing actions and learning from each step
3. Update exploration rate (reduce epsilon)
4. Repeat for many episodes until the agent learns good strategy

[ ]:
```
# Training hyperparameters
learning_rate = 0.01  # How fast to learn (higher = faster but less stable)
n_episodes = 100_000  # Number of hands to practice
start_epsilon = 1.0  # Start with 100% random actions
epsilon_decay = start_epsilon / (n_episodes / 2)  # Reduce exploration over time
final_epsilon = 0.1  # Always keep some exploration

# Create environment and agent
env = gym.make("Blackjack-v1", sab=False)
env = gym.wrappers.RecordEpisodeStatistics(env, buffer_length=n_episodes)

agent = BlackjackAgent(
    env=env,
    learning_rate=learning_rate,
    initial_epsilon=start_epsilon,
    epsilon_decay=epsilon_decay,
    final_epsilon=final_epsilon,
)
```

The Training Loop

[ ]:
```
from tqdm import tqdm  # Progress bar

for episode in tqdm(range(n_episodes)):
    # Start a new hand
    obs, info = env.reset()
    done = False

    # Play one complete hand
    while not done:
        # Agent chooses action (initially random, gradually more intelligent)
        action = agent.get_action(obs)

        # Take action and observe result
        next_obs, reward, terminated, truncated, info = env.step(action)
```

```
        # Learn from this experience
        agent.update(obs, action, reward, terminated, next_obs)

        # Move to next state
        done = terminated or truncated
        obs = next_obs

    # Reduce exploration rate (agent becomes less random over time)
    agent.decay_epsilon()
```

What to Expect During Training Early episodes (0-10,000):

- Agent acts mostly randomly (high epsilon)
- Wins about 43% of hands (slightly worse than random due to poor strategy)
- Large learning errors as Q-values are very inaccurate

Middle episodes (10,000-50,000):

- Agent starts finding good strategies
- Win rate improves to 45-48%
- Learning errors decrease as estimates get better

Later episodes (50,000+):

- Agent converges to near-optimal strategy
- Win rate plateaus around 49% (theoretical maximum for this game)
- Small learning errors as Q-values stabilize

1.2.6 Analyzing Training Results

Let's visualize the training progress:

[ ]:
```
import numpy as np
from matplotlib import pyplot as plt


def get_moving_avgs(arr, window, convolution_mode):
    """Compute moving average to smooth noisy data."""
    return np.convolve(
        np.array(arr).flatten(),
        np.ones(window),
        mode=convolution_mode
    ) / window


# Smooth over a 500-episode window
rolling_length = 500
```

```
fig, axs = plt.subplots(ncols=3, figsize=(12, 5))

# Episode rewards (win/loss performance)
axs[0].set_title("Episode rewards")
reward_moving_average = get_moving_avgs(
    env.return_queue,
    rolling_length,
    "valid"
)
axs[0].plot(range(len(reward_moving_average)), reward_moving_average)
axs[0].set_ylabel("Average Reward")
axs[0].set_xlabel("Episode")

# Episode lengths (how many actions per hand)
axs[1].set_title("Episode lengths")
length_moving_average = get_moving_avgs(
    env.length_queue,
    rolling_length,
    "valid"
)
axs[1].plot(range(len(length_moving_average)), length_moving_average)
axs[1].set_ylabel("Average Episode Length")
axs[1].set_xlabel("Episode")

# Training error (how much we're still learning)
axs[2].set_title("Training Error")
training_error_moving_average = get_moving_avgs(
    agent.training_error,
    rolling_length,
    "same"
)
axs[2].plot(range(len(training_error_moving_average)), training_error_moving_average)
axs[2].set_ylabel("Temporal Difference Error")
axs[2].set_xlabel("Step")

plt.tight_layout()
plt.show()
```

Interpreting the Results Reward Plot: Should show gradual improvement from ~-0.05 (slightly negative) to ~-0.01 (near optimal). Blackjack is a difficult game - even perfect play loses slightly due to the house edge.

Episode Length: Should stabilize around 2-3 actions per episode. Very short episodes suggest the agent is standing too early; very long episodes suggest hitting too often.

Training Error: Should decrease over time, indicating the agent's predictions are getting more accurate. Large spikes early in training are normal as the agent encounters new situations.

1.2.7 Common Training Issues and Solutions

Agent Never Improves

Symptoms: Reward stays constant, large training errors Causes: Learning rate too high/low, poor reward design, bugs in update logic Solutions:

- Try learning rates between 0.001 and 0.1
- Check that rewards are meaningful (-1, 0, +1 for Blackjack)
- Verify Q-table is actually being updated

Unstable Training

Symptoms: Rewards fluctuate wildly, never converge Causes: Learning rate too high, insufficient exploration Solutions:

- Reduce learning rate (try 0.01 instead of 0.1)
- Ensure minimum exploration (final_epsilon 0.05)
- Train for more episodes

Agent Gets Stuck in Poor Strategy

Symptoms: Improvement stops early, suboptimal final performance Causes: Too little exploration, learning rate too low Solutions:

- Increase exploration time (slower epsilon decay)
- Try higher learning rate initially
- Use different exploration strategies (optimistic initialization)

Learning Too Slow

Symptoms: Agent improves but very gradually Causes: Learning rate too low, too much exploration Solutions:

- Increase learning rate (but watch for instability)
- Faster epsilon decay (less random exploration)
- More focused training on difficult states

1.2.8 Testing Your Trained Agent

Once training is complete, test your agent's performance:

[ ]:
```python
# Test the trained agent
def test_agent(agent, env, num_episodes=1000):
    """Test agent performance without learning or exploration."""
    total_rewards = []

    # Temporarily disable exploration for testing
    old_epsilon = agent.epsilon
    agent.epsilon = 0.0  # Pure exploitation
```

```
    for _ in range(num_episodes):
        obs, info = env.reset()
        episode_reward = 0
        done = False

        while not done:
            action = agent.get_action(obs)
            obs, reward, terminated, truncated, info = env.step(action)
            episode_reward += reward
            done = terminated or truncated

        total_rewards.append(episode_reward)

    # Restore original epsilon
    agent.epsilon = old_epsilon

    win_rate = np.mean(np.array(total_rewards) > 0)
    average_reward = np.mean(total_rewards)

    print(f"Test Results over {num_episodes} episodes:")
    print(f"Win Rate: {win_rate:.1%}")
    print(f"Average Reward: {average_reward:.3f}")
    print(f"Standard Deviation: {np.std(total_rewards):.3f}")


# Test your agent
test_agent(agent, env)
```

Good Blackjack performance:

- Win rate: 42-45% (house edge makes >50% impossible)
- Average reward: -0.02 to +0.01
- Consistency: Low standard deviation indicates reliable strategy

---

1.3 Create a Custom Environment

1.3.1 Before You Code: Environment Design

Creating an RL environment is like designing a video game or simulation. Before writing any code, you need to think through the learning problem you want to solve. This design phase is crucial - a poorly designed environment will make learning difficult or impossible, no matter how good your algorithm is.

Key Design Questions Ask yourself these fundamental questions:

What skill should the agent learn?

- Navigate through a maze?
- Balance and control a system?
- Optimize resource allocation?
- Play a strategic game?

What information does the agent need?

- Position and velocity?
- Current state of the system?
- Historical data?
- Partial or full observability?

What actions can the agent take?

- Discrete choices (move up/down/left/right)?
- Continuous control (steering angle, throttle)?
- Multiple simultaneous actions?

How do we measure success?

- Reaching a specific goal?
- Minimizing time or energy?
- Maximizing a score?
- Avoiding failures?

When should episodes end?

- Task completion (success/failure)?
- Time limits?
- Safety constraints?

GridWorld Example Design For our tutorial example, we'll create a simple GridWorld environment:

- Skill: Navigate efficiently to a target location
- Information: Agent position and target position on a grid
- Actions: Move up, down, left, or right
- Success: Reach the target in minimum steps
- End: When agent reaches target (or optional time limit)

This provides a clear learning problem that's simple enough to understand but non-trivial to solve optimally.

We will implement our GridWorld game as a 2-dimensional square grid of fixed size. The agent can move vertically or horizontally between grid cells in each timestep, and the goal is to navigate to a target that has been placed randomly at the beginning of the episode.

1.3.2 Environment __init__

Like all environments, our custom environment will inherit from gymnasium.Env that defines the structure all environments must follow. One of the requirements is defining the observation and action spaces, which declare what inputs (actions) and outputs (observations) are valid for this environment.

As outlined in our design, our agent has four discrete actions (move in cardinal directions), so we'll use Discrete(4) space.

For our observation, we have several options. We could represent the full grid as a 2D array, or use coordinate positions, or even a 3D array with separate "layers" for agent and target. For this tutorial, we'll use a simple dictionary format like {"agent": array([1, 0]), "target": array([0, 3])} where the arrays represent x,y coordinates.

This choice makes the observation human-readable and easy to debug. We'll declare this as a Dict space with the agent and target spaces being Box spaces that contain integer coordinates.

For a full list of possible spaces to use with an environment, see spaces

[ ]:
```python
import numpy as np
import gymnasium as gym


class GridWorldEnv(gym.Env):

    def __init__(self, size: int = 5):
        # The size of the square grid (5x5 by default)
        self.size = size

        # Initialize positions - will be set randomly in reset()
        # Using -1,-1 as "uninitialized" state
        self._agent_location = np.array([-1, -1], dtype=np.int32)
        self._target_location = np.array([-1, -1], dtype=np.int32)

        # Define what the agent can observe
        # Dict space gives us structured, human-readable observations
        self.observation_space = gym.spaces.Dict(
            {
                "agent": gym.spaces.Box(0, size - 1, shape=(2,), dtype=int),  # [x, y] coordinates
                "target": gym.spaces.Box(0, size - 1, shape=(2,), dtype=int),  # [x, y] coordinates
            }
        )

        # Define what actions are available (4 directions)
        self.action_space = gym.spaces.Discrete(4)
```

```
        # Map action numbers to actual movements on the grid
        # This makes the code more readable than using raw numbers
        self._action_to_direction = {
            0: np.array([1, 0]),  # Move right (positive x)
            1: np.array([0, 1]),  # Move up (positive y)
            2: np.array([-1, 0]),  # Move left (negative x)
            3: np.array([0, -1]),  # Move down (negative y)
        }
```

1.3.3 Constructing Observations

Since we need to compute observations in both Env.reset() and Env.step(), it's convenient to have a helper method _get_obs that translates the environment's internal state into the observation format. This keeps our code DRY (Don't Repeat Yourself) and makes it easier to modify the observation format later.

```
    def _get_obs(self):
        """Convert internal state to observation format.

        Returns:
            dict: Observation with agent and target positions
        """
        return {"agent": self._agent_location, "target": self._target_location}
```

We can also implement a similar method for auxiliary information returned by Env.reset() and Env.step(). In our case, we'll provide the Manhattan distance between agent and target - this can be useful for debugging and understanding agent progress, but shouldn't be used by the learning algorithm itself.

```
    def _get_info(self):
        """Compute auxiliary information for debugging.

        Returns:
            dict: Info with distance between agent and target
        """
        return {
            "distance": np.linalg.norm(
                self._agent_location - self._target_location, ord=1
            )
        }
```

Sometimes info will contain data that's only available inside Env.step() (like individual reward components, action success/failure, etc.). In those cases, we'd update the dictionary returned by _get_info directly in the step method.

1.3.4 Reset function

The reset() method starts a new episode. It takes two optional parameters: seed for reproducible random generation and options for additional configuration. On the first line, you must call super().reset(seed=seed) to properly initialize the random number generator.

In our GridWorld environment, reset() randomly places the agent and target on the grid, ensuring they don't start in the same location. We return both the initial observation and info as a tuple.

```
    def reset(self, seed: Optional[int] = None, options: Optional[dict] = None):
        """Start a new episode.

        Args:
            seed: Random seed for reproducible episodes
            options: Additional configuration (unused in this example)

        Returns:
            tuple: (observation, info) for the initial state
        """
        # IMPORTANT: Must call this first to seed the random number generator
        super().reset(seed=seed)

        # Randomly place the agent anywhere on the grid
        self._agent_location = self.np_random.integers(0, self.size, size=2, dtype=int)

        # Randomly place target, ensuring it's different from agent position
        self._target_location = self._agent_location
        while np.array_equal(self._target_location, self._agent_location):
            self._target_location = self.np_random.integers(
                0, self.size, size=2, dtype=int
            )

        observation = self._get_obs()
        info = self._get_info()

        return observation, info
```

from typing import Optional ### Step function

The step() method contains the core environment logic. It takes an action, updates the environment state, and returns the results. This is where the physics, game rules, and reward logic live.

For GridWorld, we need to: 1. Convert the discrete action to a movement direction 2. Update the agent's position (with boundary checking) 3. Calculate the reward based on whether the target was reached 4. Determine if the episode should end 5. Return all the required information

```
    def step(self, action):
        """Execute one timestep within the environment.

        Args:
            action: The action to take (0-3 for directions)

        Returns:
            tuple: (observation, reward, terminated, truncated, info)
        """
        # Map the discrete action (0-3) to a movement direction
```

```
        direction = self._action_to_direction[action]

        # Update agent position, ensuring it stays within grid bounds
        # np.clip prevents the agent from walking off the edge
        self._agent_location = np.clip(
            self._agent_location + direction, 0, self.size - 1
        )

        # Check if agent reached the target
        terminated = np.array_equal(self._agent_location, self._target_location)

        # We don't use truncation in this simple environment
        # (could add a step limit here if desired)
        truncated = False

        # Simple reward structure: +1 for reaching target, 0 otherwise
        # Alternative: could give small negative rewards for each step to encourage efficiency
        reward = 1 if terminated else 0

        observation = self._get_obs()
        info = self._get_info()

        return observation, reward, terminated, truncated, info
```

1.3.5 Common Environment Design Pitfalls

Now that you've seen the basic structure, let's discuss common mistakes beginners make:

Reward Design Issues Problem: Only rewarding at the very end (sparse rewards)

```
# This makes learning very difficult!
reward = 1 if terminated else 0
```

Better: Provide intermediate feedback

```
# Option 1: Small step penalty to encourage efficiency
reward = 1 if terminated else -0.01

# Option 2: Distance-based reward shaping
distance = np.linalg.norm(self._agent_location - self._target_location)
reward = 1 if terminated else -0.1 * distance
```

State Representation Problems Problem: Including irrelevant information or missing crucial details

```
# Too much info - agent doesn't need grid size in every observation
obs = {"agent": self._agent_location, "target": self._target_location, "size": self.size}

# Too little info - agent can't distinguish different positions
obs = {"distance": distance}  # Missing actual positions!
```

Better: Include exactly what's needed for optimal decisions

```
# Just right - positions are sufficient for navigation
obs = {"agent": self._agent_location, "target": self._target_location}
```

Action Space Issues Problem: Actions that don't make sense or are impossible to execute

```
# Bad: Agent can move diagonally but environment doesn't support it
self.action_space = gym.spaces.Discrete(8)  # 8 directions including diagonals

# Bad: Continuous actions for discrete movement
self.action_space = gym.spaces.Box(-1, 1, shape=(2,))  # Continuous x,y movement
```

Boundary Handling Errors Problem: Allowing invalid states or unclear boundary behavior

```
# Bad: Agent can go outside the grid
self._agent_location = self._agent_location + direction  # No bounds checking!

# Unclear: What happens when agent hits wall?
if np.any(self._agent_location < 0) or np.any(self._agent_location >= self.size):
    # Do nothing? Reset episode? Give penalty? Unclear!
```

Better: Clear, consistent boundary handling

```
# Clear: Agent stays in place when hitting boundaries
self._agent_location = np.clip(
    self._agent_location + direction, 0, self.size - 1
)
```

1.3.6 Registering and making the environment

While you can use your custom environment immediately, it's more convenient to register it with Gymnasium so you can create it with gymnasium.make() just like built-in environments.

The environment ID has three components: an optional namespace (here: gymnasium_env), a mandatory name (here: GridWorld), and an optional but recommended version (here: v0). You could register it as GridWorld-v0, GridWorld, or gymnasium_env/GridWorld, but the full format is recommended for clarity.

Since this tutorial isn't part of a Python package, we pass the class directly as the entry point. In real projects, you'd typically use a string like "my_package.envs:GridWorldEnv".

```
# Register the environment so we can create it with gym.make()
gym.register(
    id="gymnasium_env/GridWorld-v0",
    entry_point=GridWorldEnv,
    max_episode_steps=300,  # Prevent infinite episodes
)
```

Once registered, you can check all available environments with gymnasium.pprint_registry() and create instances with gymnasium.make(). You can also create vectorized versions with gymnasium.make_vec().

```
import gymnasium as gym

# Create the environment like any built-in environment
>>> env = gym.make("gymnasium_env/GridWorld-v0")
<OrderEnforcing<PassiveEnvChecker<GridWorld<gymnasium_env/GridWorld-v0>>>>

# Customize environment parameters
>>> env = gym.make("gymnasium_env/GridWorld-v0", size=10)
>>> env.unwrapped.size
10

# Create multiple environments for parallel training
>>> vec_env = gym.make_vec("gymnasium_env/GridWorld-v0", num_envs=3)
SyncVectorEnv(gymnasium_env/GridWorld-v0, num_envs=3)
```

1.3.7 Debugging Your Environment

When your environment doesn't work as expected, here are common debugging strategies:

Check Environment Validity

```
from gymnasium.utils.env_checker import check_env

# This will catch many common issues
try:
    check_env(env)
    print("Environment passes all checks!")
except Exception as e:
    print(f"Environment has issues: {e}")
```

Manual Testing with Known Actions

```
# Test specific action sequences to verify behavior
env = gym.make("gymnasium_env/GridWorld-v0")
obs, info = env.reset(seed=42)  # Use seed for reproducible testing

print(f"Starting position - Agent: {obs['agent']}, Target: {obs['target']}")

# Test each action type
actions = [0, 1, 2, 3]  # right, up, left, down
for action in actions:
    old_pos = obs['agent'].copy()
    obs, reward, terminated, truncated, info = env.step(action)
    new_pos = obs['agent']
    print(f"Action {action}: {old_pos} -> {new_pos}, reward={reward}")
```

Common Debug Issues

```python
# Issue 1: Forgot to call super().reset()
def reset(self, seed=None, options=None):
    # super().reset(seed=seed)  # ❌ Missing this line
    # Results in: possibly incorrect seeding

# Issue 2: Wrong action mapping
self._action_to_direction = {
    0: np.array([1, 0]),   # right
    1: np.array([0, 1]),   # up - but is this really "up" in your coordinate system?
    2: np.array([-1, 0]),  # left
    3: np.array([0, -1]),  # down
}

# Issue 3: Not handling boundaries properly
# This allows agent to go outside the grid!
self._agent_location = self._agent_location + direction  # ❌ No bounds checking
```

1.3.8 Using Wrappers

Sometimes you want to modify your environment's behavior without changing the core implementation. Wrappers are perfect for this - they let you add functionality like changing observation formats, adding time limits, or modifying rewards without touching your original environment code.

```
>>> from gymnasium.wrappers import FlattenObservation

>>> # Original observation is a dictionary
>>> env = gym.make('gymnasium_env/GridWorld-v0')
>>> env.observation_space
Dict('agent': Box(0, 4, (2,), int64), 'target': Box(0, 4, (2,), int64))

>>> obs, info = env.reset()
>>> obs
{'agent': array([4, 1]), 'target': array([2, 4])}

>>> # Wrap it to flatten observations into a single array
>>> wrapped_env = FlattenObservation(env)
>>> wrapped_env.observation_space
Box(0, 4, (4,), int64)

>>> obs, info = wrapped_env.reset()
>>> obs
array([3, 0, 2, 1])  # [agent_x, agent_y, target_x, target_y]
```

This is particularly useful when working with algorithms that expect specific input formats (like neural networks that need 1D arrays instead of dictionaries).

1.3.9 Advanced Environment Features

Once you have the basics working, you might want to add more sophisticated features:

Adding Rendering

```
def render(self):
    """Render the environment for human viewing."""
    if self.render_mode == "human":
        # Print a simple ASCII representation
        for y in range(self.size - 1, -1, -1):  # Top to bottom
            row = ""
            for x in range(self.size):
                if np.array_equal([x, y], self._agent_location):
                    row += "A "  # Agent
                elif np.array_equal([x, y], self._target_location):
                    row += "T "  # Target
                else:
                    row += ". "  # Empty
            print(row)
        print()
```

Parameterized Environments

```
def __init__(self, size: int = 5, reward_scale: float = 1.0, step_penalty: float = 0.0):
    self.size = size
    self.reward_scale = reward_scale
    self.step_penalty = step_penalty
    # ... rest of init ...

def step(self, action):
    # ... movement logic ...

    # Flexible reward calculation
    if terminated:
        reward = self.reward_scale  # Success reward
    else:
        reward = -self.step_penalty  # Step penalty (0 by default)
```

1.3.10 Real-World Environment Design Tips

Start Simple, Add Complexity Gradually

- First: Get basic movement and goal-reaching working
- Then: Add obstacles, multiple goals, or time pressure
- Finally: Add complex dynamics, partial observability, or multi-agent interactions

Design for Learning

- Clear Success Criteria: Agent should know when it's doing well
- Reasonable Difficulty: Not too easy (trivial) or too hard (impossible)
- Consistent Rules: Same action in same state should have same effect

- Informative Observations: Include everything needed for optimal decisions

Think About Your Research Question

- Navigation: Focus on spatial reasoning and path planning
- Control: Emphasize dynamics, stability, and continuous actions
- Strategy: Include partial information, opponent modeling, or long-term planning
- Optimization: Design clear trade-offs and resource constraints