Reinforcement Learning – Teaching Reference (Linear Pathway)

Purpose: This document is a distilled, structured reference of completed tasks and learned concepts. It is intended to be provided to *Perplexity* (or another tutor system) as **context** so that future explanations, examples, and tasks build directly on what has already been learned.

Learning Status: Linear Pathway

How to Use This Reference (Instruction for the Tutor)

- Assume the learner **already understands and has implemented** the items below.
- Do **not** re-explain basics unless explicitly asked.
- When teaching new tasks, **reuse patterns, conventions, and terminology** listed here.
- Prefer extensions, variations, debugging depth, and conceptual reinforcement over repetition.

Completed Tasks & Core Knowledge

Task 10–11: Tabular Q-Learning (GridWorld)

What Was Implemented - Q-table stored as a Python dictionary keyed by `(row, col, action)`. - Safe Q-value access via `get_Q(state, action)` with default `0.0`. - Standard Q-learning update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$$

Environment Design - 2D GridWorld with: - Valid cells - Obstacles - Terminal goal state - Reward scheme: - Negative reward for invalid moves - Positive reward for reaching the goal - Small step reward otherwise

Policy & Training Loop - ε-greedy action selection: - ε → random action (exploration) - 1−ε → greedy action (exploitation) - Episode structure: - Fixed start `(0,0)` - Loop: select action → step → update Q → terminate on terminal state - Logged episode length to measure convergence

Key Insights - Convergence visible as episode length → optimal path length - Hyperparameter effects: - High α: faster but noisier learning - Lower γ: short-term reward focus - Verified learning via greedy rollouts and Q-table inspection

---

Task 14: DQN – Hyperparameter Tuning (Conceptual)

Explored Hyperparameters - Learning rate (α) - Discount factor (γ) - Exploration schedule (ε decay)

Evaluation Metrics - Average episode length - Variance of episode length - Goal-reaching success rate

Core Understanding - No universal best hyperparameters - Optimal values are **task-dependent** and must be determined empirically - Stability vs speed trade-off is central in deep RL

Task 17: Policy & Value Visualization / Debugging

Policy Inspection - Generated deterministic greedy policy (ε = 0) - Rendered policy as: - Text grid of arrows (↑ ↓ ← →) - X for obstacles, G for goal

Visualizations Built - Quiver plots of policy directions - Heatmap of state values:

$$V(s) = \max_a Q(s, a)$$

Rendering Conventions - Coordinate system fixed as `(0,0)` = top-left - Resolved plotting mismatches using: - `imshow(origin=...)` - `invert_yaxis()` for quiver plots

Debugging Insights - Value gradients should smoothly increase toward the goal - Visualizations expose bugs invisible in raw numbers - Boundary errors - Coordinate mismatches - Invalid transitions

---

Task 18: Consolidated Knowledge for Future Work

1. Gymnasium Environment Design

Custom Environments Implemented - `GridWorldEnv` (2D grid) - `NumberGuessEnv` (1D number line)

Correct Gymnasium API Usage - `action_space`, `observation_space` - `reset(seed=None, options=None) -> (obs, info)` - `step(action) -> (obs, reward, terminated, truncated, info)`

Termination Semantics - `terminated`: natural terminal condition (goal, exact guess) - `truncated`: forced cutoff (e.g., max steps)

---

2. MDP Design: State, Action, Reward

State Representations - GridWorld: `(row, col)` - NumberGuess: `[last_guess, difference]`

Action Mappings - GridWorld: - 0=up, 1=down, 2=left, 3=right - NumberGuess: - Discrete index mapped to numeric guess

Reward Shaping Experience - GridWorld: penalties for invalid moves, sparse terminal reward - NumberGuess: experimented with distance-based rewards

---

3. Rendering & Visualization Patterns

Design Principle - Environment logic ≠ rendering logic

Rendering Implementations - GridWorld: - ASCII grid - Matplotlib grid with colormap and agent marker - NumberGuess: - Number line - Target marker - Guess marker + arrow annotation

Interactive Matplotlib Pattern - Persistent `fig, ax` - `plt.ion()` - Clear and redraw per step - `plt.pause()` for animation

---

4. Stable-Baselines3 Integration (DQN)

Training Pattern

```
env = CustomEnv()
model = DQN("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=...)
```

Evaluation Pattern - Deterministic predictions - Manual step loop with render - Reset on `terminated or truncated`

Debugging Techniques - Print internal state during evaluation - Interpret SB3 logs: - `ep_len_mean` - `ep_rew_mean` - `exploration_rate`

---

5. Debugging & API Discipline

Common Bugs Fixed - Incorrect `step()` signatures - State reinitialization inside `step` - Incorrect use of `render_mode`

Debug Strategy - Print state transitions - Cross-check logic vs visualization - Isolate logic bugs from rendering bugs

---

High-Level Takeaways

The learner can now: - Design Gymnasium-compliant custom environments - Encode states/actions suitable for function approximation - Apply reward shaping intentionally - Debug agents using visualization, not just metrics - Use SB3 as a standardized agent layer

These patterns are **explicitly intended** to be reused and extended in: - Factory scheduling environments - SimPy-based discrete event simulations - Multi-objective and time-dependent RL problems

---

Guidance for Next Tasks

When introducing **factory environments, SimPy integration, or advanced scheduling**: - Assume fluency with Gymnasium + SB3 - Focus on: - Time-based state evolution - Event-driven transitions - Deadline-sensitive rewards - Emphasize visualization and debugging tools early

End of Teaching Reference