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

---

Task 19: Factory Environment v1 (SimpleFactoryEnv)

Environment Overview

- **Environment**: `SimpleFactoryEnv`, Gymnasium-compatible, trained with SB3 (PPO / DQN).
- **Purpose**: First factory-style scheduling environment with two functional units (FUs) and batch-style job flow.

Observation (State) Design

State Representation: Fixed-size, flattened numeric vector (MLP-friendly)

```
[Abusy, Bbusy, Atime, Btime]
```

- `Abusy`, `Bbusy` ∈ {0, 1} indicate whether FU A or B is currently processing a job.
- `Atime`, `Btime` ≥ 0 track elapsed processing time for the current job in each FU.

Design Rationale - Mirrors earlier GridWorld / NumberGuess compact state design. - Explicitly encodes both **availability** and **temporal progress**. - Prepares the environment for Task 20 (SimPy), where time becomes event-driven.

Action Space

Action Space: `Discrete(2)`

- `0 = hold`
- Let both FUs continue processing their current jobs.
- `1 = move`
- Advance jobs between FUs and/or introduce new jobs, depending on state.

Design Insight - A single global decision ("wait vs advance") can still control parallel FUs when the environment manages internal routing. - Keeps the MDP small, debuggable, and stable for PPO.

Order & FU Counters

Tracked internally: - `total_orders`: sampled at reset (e.g., 5–15). - `todo`: orders not yet started. - `doing`: orders currently being processed in A and/or B. - `complete`: orders that have finished processing and exited B.

Step Dynamics (Conceptual Flow)

Each `step(action)` performs the following logic:

1. Time Update

- If a FU is busy, its timer increments (`Atime`, `Btime`).
- If processing exceeds configured durations (`Aduration`, `Bduration`), small negative penalties are applied (over-processing).

2. Completion & Movement Logic

- **B completion**:
- If `Bbusy == 1` and `Btime` reaches its target:
    - Job completes
    - `complete += 1`, `Bbusy = 0`
    - Strong positive reward
- **A → B transfer**:
- If `Abusy == 1` and B is free:
    - Job moves from A to B
    - Update `Abusy`, `Bbusy`, reset timers
    - Small positive reward
- If B is busy and move is attempted:
    - Negative reward

3. Starting New Jobs

- If `todo > 0` and A is free:
- Under `action == 1`, a new job is started in A:
    - `Abusy = 1`, `Atime = 0`
    - `todo -= 1`, `doing += 1`
    - Small positive reward

4. Invalid Actions

- Actions that would overload capacity (e.g., `action == 1` when both FUs are busy):
- Environment state remains unchanged
- Strong negative reward applied

Episode Length & Termination

- `step_count` increments every step.
- **Truncation**:
- If `step_count >= max_steps`
- Episode truncates with negative reward
- **Natural Termination**:

```
terminated = (todo == 0 and doing == 0)
```

- All orders completed and no FU still processing

Reward Structure

Positive Rewards - Large completion spikes (≈ 40–42) for: - B → complete transitions - Valid batch moves - Small positive reward for successfully starting new jobs

Negative Rewards - Small step penalties (≈ −0.5 to −1.0): - Over-processing - Excessive waiting - Large penalties (≈ −10): - Invalid actions - Capacity violations

Design Insight - Sparse reward landscape with strong terminal spikes - Light shaping to discourage idle or pathological waiting

---

PPO Training Behaviour (Observed)

Algorithm: PPO (`MlpPolicy`) on `SimpleFactoryEnv`

Typical stats after ~20k timesteps: - `ep_len_mean`: ~25–30 - `ep_rew_mean`: ~100–150 (dominated by completion rewards) - KL divergence and entropy in stable ranges

Learned Qualitative Policy - Use `action == 1` to: - Start jobs when A is free - Move jobs when FUs are ready - Use `action == 0` to: - Wait while A/B process jobs - Cycle: - start → wait → move/complete → repeat - Finish remaining jobs once `todo == 0`, then terminate

---

Key Design Lessons from Task 19

Observation Design

- Fixed-size numeric vectors integrate cleanly with SB3 MLP policies.
- Explicit time encoding enables smooth transition to event-driven simulation.

Action Semantics

- Coarse actions + rich environment logic simplify learning without losing expressiveness.
- Internal handling of parallelism reduces action-space complexity.

Invalid Actions & Stability

- Penalize illegal actions without changing state.
- Separate learnable mistakes from terminal conditions for PPO stability.

Reward Shaping for Factory RL

- Large completion bonuses drive throughput.

- Small per-step penalties create urgency.
- Intermediate rewards help PPO learn multi-step production cycles.

Debugging Patterns (Reused)

- Step-level logging:
- State, action, counters, reward, terminated, truncated
- Cross-check logs with:
- `ep_rew_mean`
- `ep_len_mean`
- Same philosophy as GridWorld policy/value visualization

Positioning for Next Task

Task 19 establishes the **state, action, and reward contract** for the factory domain.

Task 20 (SimPy integration) should: - Replace hand-rolled time increments with event-driven processes - Preserve observation structure and reward logic - Shift from fixed-step time to simulation time without changing agent semantics

---

Task 20: Discrete Event Simulation with SimPy

Goal of the Task

Learn to model **time-based systems** using **discrete event simulation (DES)**, replacing manual step counters with an event-driven notion of time. This was done by building and experimenting with a simple **single-server queue** in SimPy.

Core SimPy Concepts Learned

Simulation Environment

- Used `simpy.Environment` as the global simulation engine.
- `env.now` represents the current simulated time.
- The environment maintains the future event queue and advances time automatically.
- This replaces the explicit `step_count += 1` pattern used in Gymnasium environments.

Processes

- System components are implemented as **generator functions** (e.g., `job`, `job_generator`).
- Processes evolve over time by yielding events back to the environment.
- This pattern represents entities acting, waiting, and resuming over simulated time.

Timeouts

- Used `yield env.timeout(t)` to represent the passage of simulated time.
- Applied to:

- Service durations
- Interarrival times
- Time advances only when events occur, not in fixed increments.

Resources and Queues

- Used `simpy.Resource(env, capacity=1)` to represent a single machine/server.
- Requests automatically form a FIFO queue when the resource is busy.
- Jobs block until the resource becomes available, without manual queue logic.

Queue Model Implemented

A basic **1-server queueing system** was simulated: - Jobs arrive over time. - Each job requests service from the server. - After service, the job departs the system.

Job Process Lifecycle

Each job process: 1. Records `arrival_time = env.now` on entry. 2. Requests the machine resource. - If busy, waits in the resource queue. 3. Records `start_service_time` when the resource is acquired. 4. Waits for processing via `yield env.timeout(SERVICE_TIME)`. 5. Records `departure_time` when service completes.

Metrics and Analysis

A `Metrics` helper was used to store per-job timestamps: - `arrival_time` - `start_service_time` - `departure_time`

From these, the following were computed:

- `waiting_time = start_service_time − arrival_time`
- `system_time = departure_time − arrival_time`

After each run, summary statistics were calculated: - Number of jobs processed - Average and maximum waiting time - Average and maximum system time

Conceptual Parallel - Mirrors earlier RL diagnostics: - Episode length ↔ system time - Reward statistics ↔ throughput / latency trade-offs

---

Experiments Performed

- Varied `INTER_ARRIVAL` and `SERVICE_TIME` to study load effects:
- Low load → short queues, small waiting times

- High load → long queues, large waiting times

- Introduced randomness:

- Exponential interarrival times
- Random service times

Observed Effects - Increased variability raises maximum waiting and system times. - Average metrics may stay similar while tail behavior worsens.

Connections to Previous Tasks

- Earlier environments (GridWorld, NumberGuess, SimpleFactoryEnv):

- Time advanced one step per RL action.

- SimPy introduces:

- Event-driven time
- Asynchronous processing
- Natural queueing behavior

This aligns more closely with real factory and scheduling systems.

Positioning for Next Task (Task 21)

With this task, the learner is now fluent in: - `simpy.Environment` - Process-based modeling - `env.timeout` - Resource-backed queues

Next step: Embed SimPy inside the factory environment so that: - RL selects scheduling/routing actions - SimPy governs arrivals, processing, waiting, and time progression - The existing factory **state, action, and reward contract** is preserved

---

End of Teaching Reference

Task 21 – What Was Learned (Duplicate Consolidated Notes)

Environment Summary

You built a **Gymnasium-compatible RL environment** wrapping a **SimPy discrete-event simulation** of a 2-FU factory (A → B).

- Time is governed by SimPy (`self.env`, `env.timeout`, `env.run(until=...)`).
- Jobs spend **2 units at A** and **5 units at B**; `info["time"]` matches true simulated time.

Action Semantics (Stable Interface)

- **action = 0**: hold (idle/penalty logic)
- **action = 1**: start job on A if free and `to_do > 0`
- **action = 2**: move completed job A → B if B free and A finished
- **action = 3**: start both A and B under strict conditions (A finished, B free, orders available)

These form the core scheduling interface going forward.

Observation Vector (10D)

- Global: `[to_do, complete]`
- Per FU (A then B): `[busy_flag, avg_wait_time, avg_system_time, total_working_time]`

Aggregated from job logs: - `arrival_times` - `start_service_times` - `departure_times`

Reward Shaping

- `+0.5` start job
- `+1` complete at any FU
- `+5` complete at final FU (B)
- `+200` all orders completed
- Small penalties for invalid/ineffective actions

Encodes throughput-maximization with efficiency constraints.

PPO Behaviour

- Stable learning: `ep_len_mean ≈ 20–30`, `ep_rew_mean ≈ 260–280`.
- Repeated successful full-completion episodes.

Design Patterns to Reuse

Separation of Concerns

- **Gym layer**: reset, step, actions, observations, rewards.
- **SimPy layer**: time, resources, FU processes via `env.timeout`.

State Design Strategy

- Fixed-size numeric summaries (averages/totals/flags).
- Do **not** expose raw per-job histories.

Scheduling Logic Pattern

- Strict lifecycle: **A → B**.
- A cannot push to B if B busy and no buffer.
- Enforced via SimPy Resources + action guards.

Time Advancement

- Use `remaining_times` and `self.working`.
- Advance with `env.run(until=min_remaining_time)` when active.
- RL steps represent **decision points**, not fixed time ticks.

Reward Design Philosophy for Future Tasks

- Large positive spikes for meaningful milestones.
- Small penalties for wasted actions.
- Add lateness/deadlines as **extra reward terms**, not replacements.

Guidance for Future Tasks (22+)

Future extensions (deadlines, buffers, failures, multi-stream arrivals) should:

- Preserve Gym API: `reset → (obs, info)`, `step → (obs, reward, terminated, truncated, info)`.
- Keep the fixed observation-vector philosophy.
- Reuse SimPy core (resources, job processes, timing via `env.run`).
- Extend (not replace) action semantics; complexity should live in job data, reward logic, and observation features.

Task 22 – What I Learned

1) Designing a Factory RL Environment (SimPy + Gym)

- Built **WorkshopEnv**, wrapping a SimPy `Environment` inside a Gymnasium environment so PPO can control a discrete-event factory.
- Modelled FUs **A, B, C** as SimPy `Resource`s with service times.
- Modelled orders with **size, random release times, due dates, and random routes** through FUs.
- Defined a **MultiDiscrete action space** per FU (hold / pull from previous FU / start order), converting scheduling into an MDP.

---

2) Observation Space Design (Job + Machine Centric)

- Per-order features:
- Fraction remaining (`to_do / size`)

- Fraction completed
- Normalised time-to-deadline (`time_to_deadline / order_length`, clipped to [-1, 1])
- Route encoding via **one-hot FU start indicators**.
- Per-FU features:
- Busy flag
- Normalised working time (utilisation)
- Normalised waiting time

Goal: give the agent both job-level and machine-level state.

---

3) Reward Shaping for Scheduling

- Positive rewards for starting and completing jobs at FUs.
- Larger bonus when an order finishes its **final FU** (throughput objective).
- **Lateness penalty** proportional to how far past the due date the system is (only after order release).
- **Terminal bonus** scaled by average machine utilisation when all orders finish.

Encodes throughput + timeliness + utilisation.

---

4) Time Control in a Discrete-Event RL Loop

- Advanced time using `env.run(until=env.now + service_time)` when processing occurs.
- Used +1 time-step advances when idle but jobs remain, bridging event-driven DES and step-based RL.
- Tracked per-FU `remaining_time` and advanced to the **next completion event** using `min(active_times)`.
- Decremented remaining times after each advance → emulates next-event simulation inside each RL step.

---

5) Gantt-Style Schedule Logging and Export

- Implemented `_log_gantt(start_t, end_t)` to record FU activity each integer second.
- Stored:
- `time_log`: list of timestamps
- `fu_log`: dict keyed by FU name, storing which job was processed
- Recorded `working_on` identifiers (e.g., `O1-P?`) for traceability.
- Exported logs to **CSV (**`gantt_log.csv`**)** for Excel visualisation.

---

6) PPO Integration and Debugging

- Integrated **SB3 PPO** with tuned hyperparameters (LR, gamma, network size).
- Verified training via:
- `ep_len_mean`
- `ep_rew_mean`

- `explained_variance`
- Debugged with sanity prints:
- `env.env.now`
- `FUs["A"]["working_on"]`
- sample `time_log` and `fu_log` entries

Ensured logged Gantt data matches environment step-by-step behaviour.