Task 10–11: Tabular Q-Learning (Gridworld Prototype)

- Represented Q-values in a Python dict keyed by (row, col, action).
- Wrote get_Q helper to safely read Q-values with default 0.0.
- Implemented the Q-learning update:
  $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max{}'_a Q(s',a') - Q(s,a)]$.
- Designed a simple gridworld: valid cells, obstacles, goal, rewards for move/goal/invalid.
- Built an epsilon-greedy policy (choose_action) that:
  - With probability ε picks a random action (explore).
  - Otherwise chooses the action with highest Q at the current state (exploit).
- Implemented an episode loop:
  - Start at (0,0), repeatedly choose actions, call step, update Q, stop on terminal state.
- Logged steps per episode to see learning progress; understood that convergence means episode lengths approaching the optimal path length.
- Observed effect of hyperparameters:
  - Higher α → faster but noisier learning.
  - Lower γ focused more on near-term rewards.
- Learned to interpret Q-tables and greedy rollouts to verify that the agent found the optimal route around obstacles.

Task 14: DQN / Hyperparameter Tuning (Conceptually)

- Understood that learning rate, discount factor, and exploration schedule strongly affect:
  - Speed of convergence.
  - Stability of learning and final performance.
- Practiced systematic sweeps:
  - Tried multiple learning rates (e.g. α = 0.1, 0.3, 0.5) and compared average episode length.

- Tried several γ values and saw how far-sightedness changes behaviour in the grid.

- Learned to evaluate runs using:
    - Average steps per episode.
    - Variance in episode length.
    - Percentage of episodes reaching the goal.
- Built intuition that “best” hyperparameters are task-dependent and must be chosen empirically, not by default.

Task 17: Policy/Value Visualisation & Debugging

- Generated a deterministic greedy path from the learned Q-table (ε=0) to inspect the policy.
- Printed a text grid of arrows:
    - ↑, ↓, ←, → for best action in each non-terminal, non-obstacle cell.
    - X for obstacles, G for goal.
- Created Matplotlib visualisations:
    - Quiver plot of arrows over the grid.
    - Points (red) for obstacles and (green) for goal.
    - Heatmap of state values $V(s) = \max_a Q(s, a)$ to show high-value regions near the goal.
- Debugged coordinate issues:
    - Decided on a consistent convention: (0,0) at top-left.
    - Fixed mismatches by controlling imshow origin and invert_yaxis in quiver plots.
- Used heatmap to see:
    - Value gradient towards the goal.
    - Effect of obstacles and dead-ends on state value.
- Learned how visualisations reveal bugs (inconsistent paths, wrong boundaries) that are not obvious from raw numbers.

Task 18: What You've Learned (for Future You + Collaborators)

1. Gymnasium Environment Design

- **Implemented two custom environments: GridWorldEnv (2D grid) and number_guesser_env (1D number line).**
- **Correctly used the Gymnasium API:**
  - **Defined action_space and observation_space using spaces.Discrete and spaces.Box.**
  - **Implemented reset(self, seed=None, options=None) returning (obs, info).**
  - **Implemented step(self, action) returning (obs, reward, terminated, truncated, info).**
- **Understood terminated vs truncated:**
  - **terminated: natural episode end (goal reached, invalid move, exact guess, etc.).**
  - **truncated: forced end (e.g., max_steps reached) without natural terminal condition.**

2. State, Action, Reward Design (MDP thinking)

- **Chose compact, meaningful state representations:**
  - **GridWorld: current grid coordinates (row, col).**
  - **NumberGuess: [last_guess, difference] where difference = guess - target.**
- **Mapped discrete actions to domain semantics:**
  - **GridWorld: 0=up, 1=down, 2=left, 3=right, with boundary/obstacle checks.**
  - **NumberGuess: guess = low + action to map discrete index to actual number.**
- **Practised reward shaping:**
  - **GridWorld: large negative reward for invalid moves, positive for reaching goal, small positive otherwise.**
  - **NumberGuess: experimented with distance-based rewards (exponential) and discussed alternatives (-abs(diff), squared penalties, big success bonus) to encourage hitting the exact target.**

3. Custom Rendering and Visualisation

- Learned to separate environment logic from rendering:
    - step only updates state and returns signals.
    - render visualises current state in different modes.
- GridWorld rendering:
    - ASCII: printed a grid with indices, obstacles, goal, and agent.
    - Matplotlib: used imshow with a custom colormap, grid lines, and a blue dot for the agent; handled origin, clearing axes, and persistent fig/ax.
- NumberGuess rendering:
    - Implemented a number line visual:
        - Horizontal line from low to high.
        - “X” at target.
        - Red dot at last_guess.
        - Arrow from guess to target using annotate.
- Understood the pattern for interactive Matplotlib in envs:
    - Create figure once (self.fig, self.ax), call plt.ion(), clear and redraw each frame, plt.pause(...).

4. Integration with Stable-Baselines3 (DQN)

- Successfully plugged custom Gymnasium envs into SB3:
    - Confirmed that a custom env with correct API works with SB3’s DQN.
- Training pattern:
    - env = GridWorldEnv() / number_guesser_env()
    - model = DQN("MlpPolicy", env, verbose=1)
    - model.learn(total_timesteps=...).
- Evaluation pattern:
    - Created a separate eval_env = number_guesser_env(render_mode="human").

- Used model.predict(obs, deterministic=True) inside a loop with env.step, env.render, and if terminated or truncated: reset.

- Debugging learned behaviour:
  - Printed (target, last_guess, reward, done, truncated) to understand what the agent is actually doing.
  - Interpreted SB3 logs (ep_len_mean, ep_rew_mean, exploration_rate) to assess learning progress.

5. Debugging and API Discipline

- Fixed several API-level bugs that are common in custom envs:
  - Step signature mismatch (step(self, action) vs extra arguments) and resulting SB3 TypeError.
  - Ensured state variables (last_guess, target) are updated in the correct places and not reinitialised inside step.
  - Ensured render uses self.render_mode, not a bare render_mode.
- Learned to debug by:
  - Printing current_state / next_state in GridWorld and (target, last_guess, difference) in NumberGuess.
  - Checking that visual behaviour matches printed state (i.e., distinguishing env logic bugs from rendering bugs).

6. Conceptual Takeaways for Future Tasks

- You now know how to:
  - Design custom environments that respect Gymnasium's API and integrate cleanly with RL libraries.
  - Choose and encode state/action/reward in a way that's suitable for function approximation (DQN).
  - Build intuitive visualisations to inspect agent behaviour, not just rely on scalar metrics.
  - Use SB3 as a "standard agent layer" on top of your custom envs, which scales directly to your factory scheduling environments in later tasks.

For the next tasks (factory env, SimPy integration), these patterns—clean API, explicit state design, careful reward shaping, and visual debugging—are the key tools you'll reuse and extend.