EE449 HW2

Yusuf Baran

1.

1.1

The functions of the MazeEnvironment class works properly.

1.2

The environment and the agent are ready for the learning.

1.3

In this part, while plotting the convergence curves, I realized that the first iterations are dominating as expected. So, I discarded the first 10 iterations for clarity. The scripts include both plotting methods. One can generate both versions.

I. **alpha 0.001 plots**

- **Utility value function heatmap:**

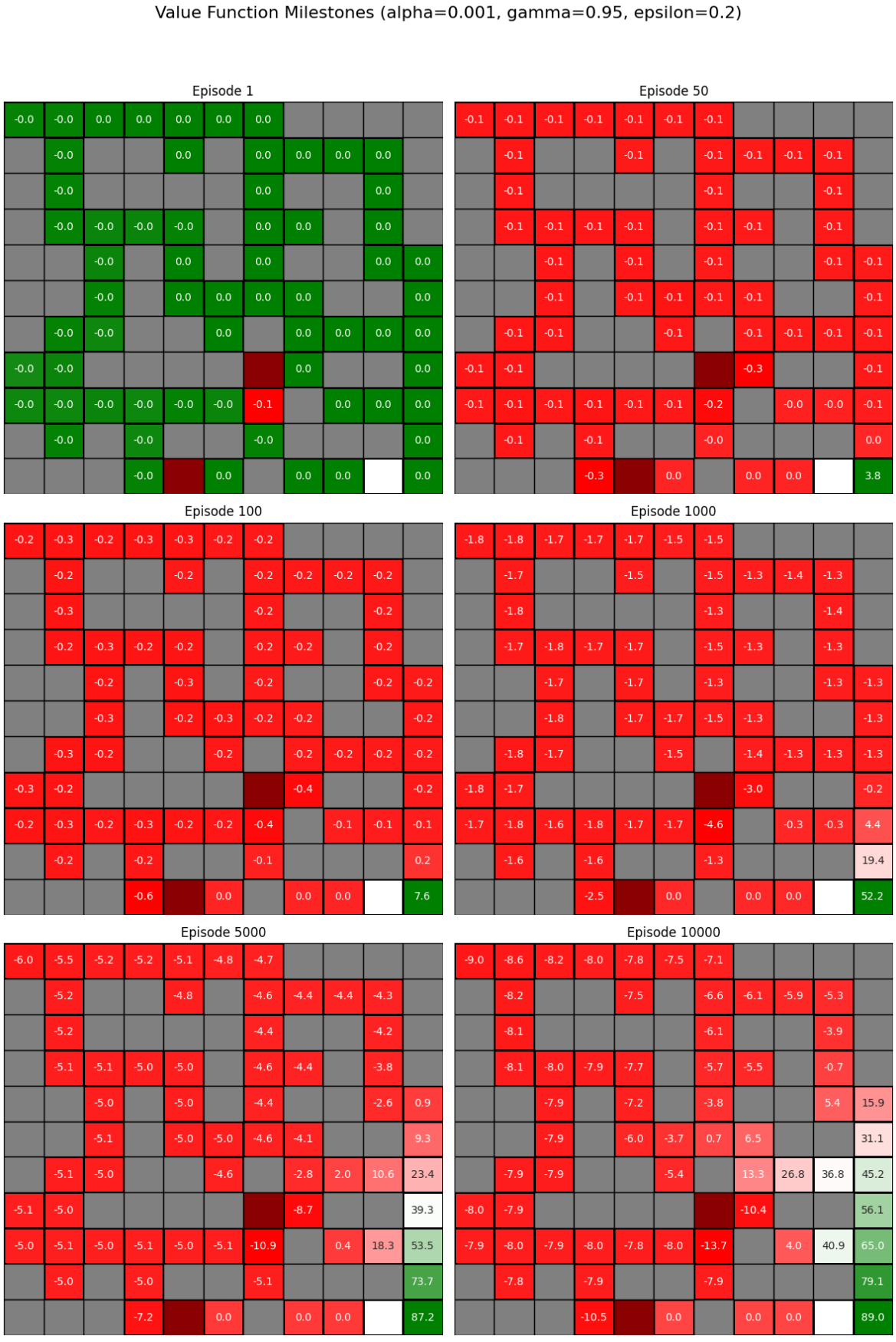

Figure 1.alpha 0.001 value plot

- **Policy:**

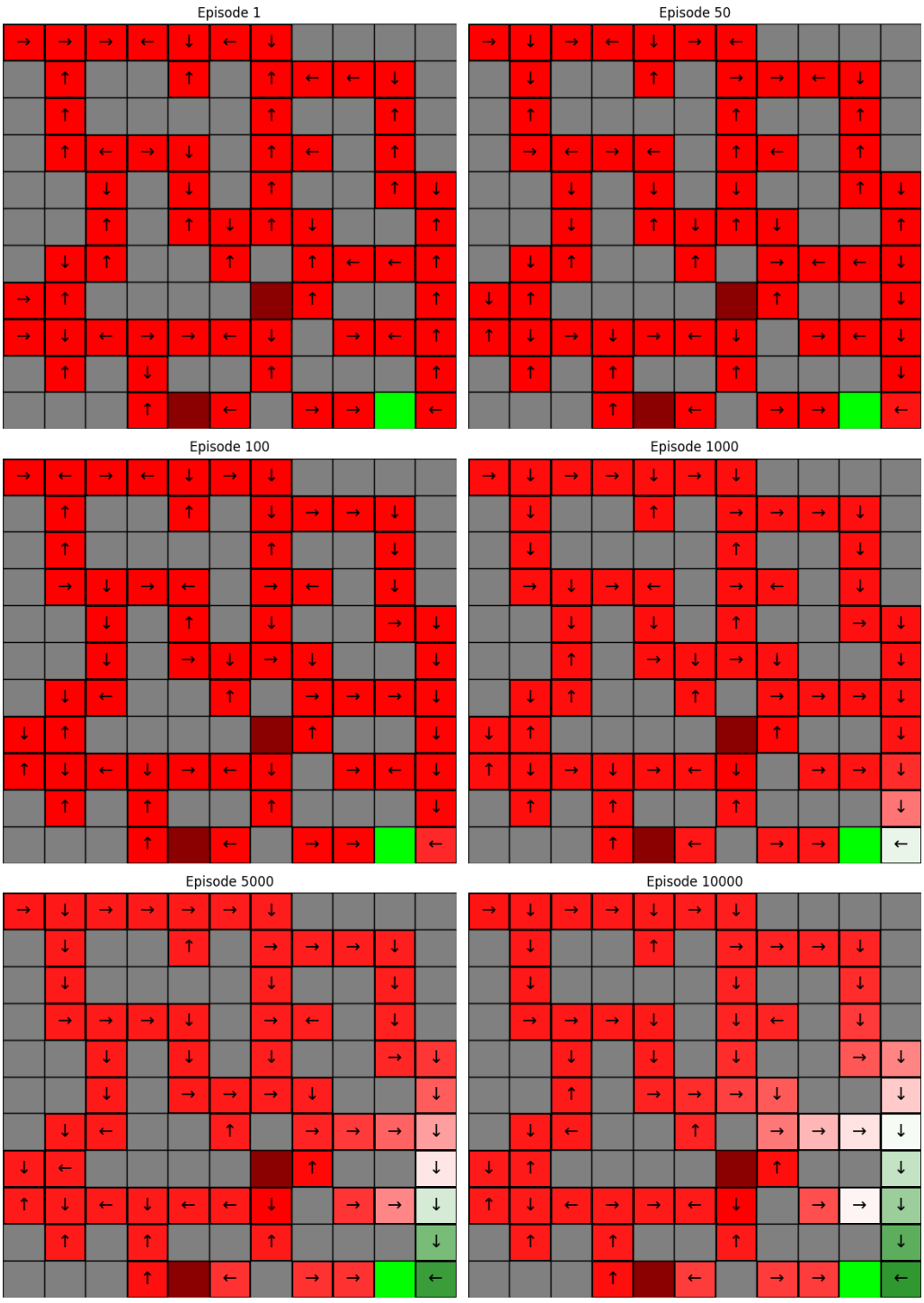

Figure 2.alpha 0.001 policy plot

- **Convergence:**

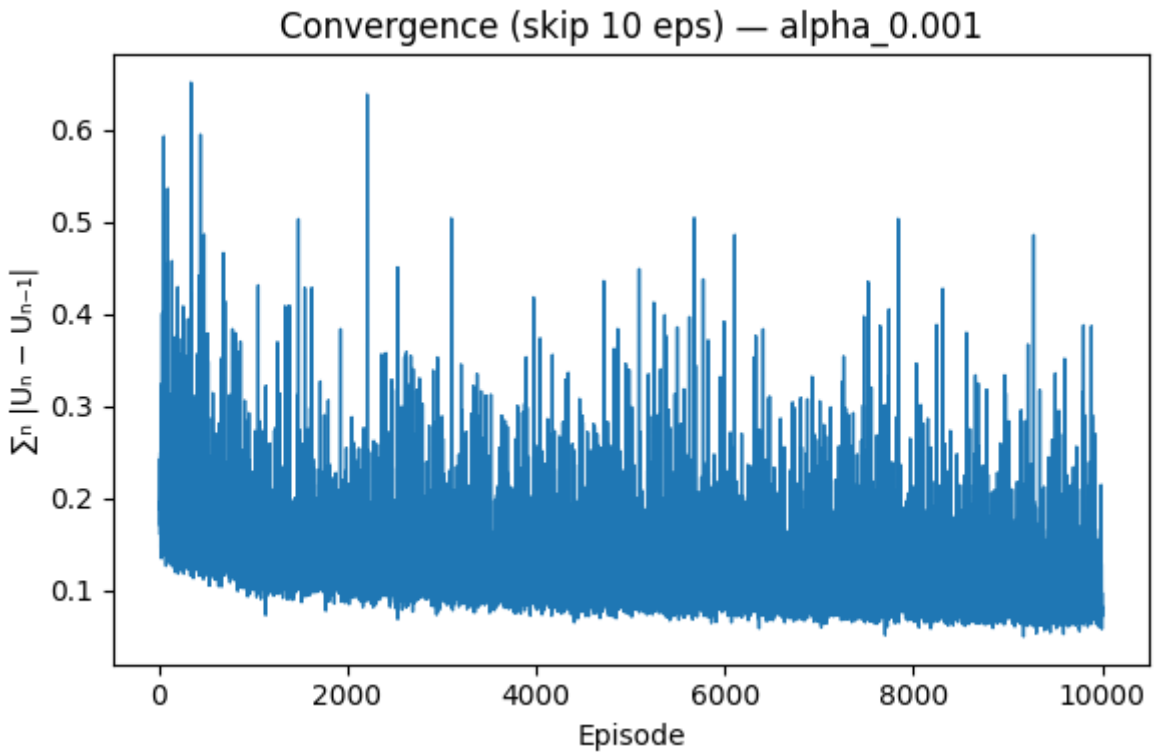

Figure 3.alpha 0.001 convergence plot

II. alpha 0.01 plots

- **Utility value function heatmap:**

Figure 4.alpha 0.01 value plot

- **Policy:**

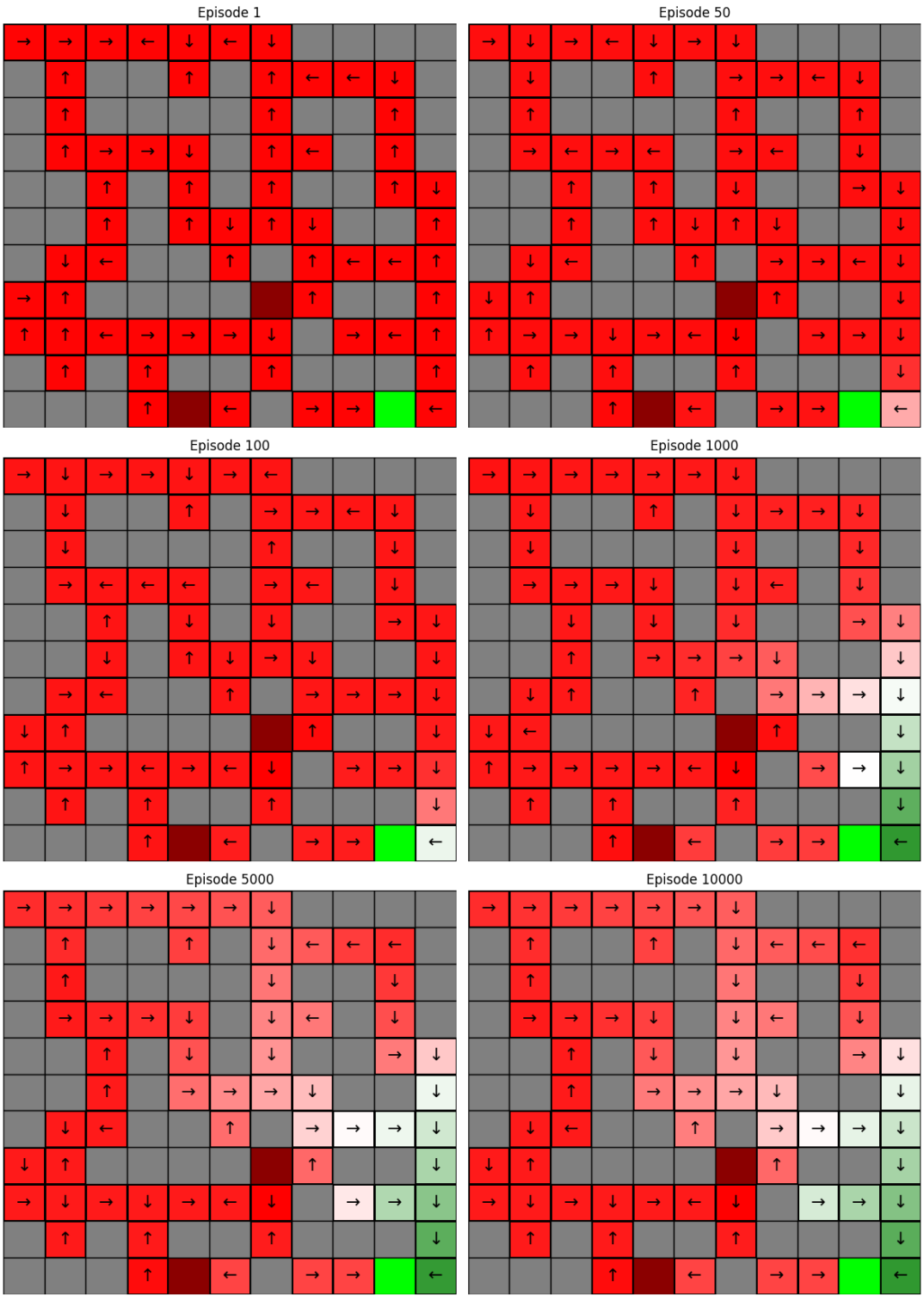

Figure 5.alpha 0.01 policy plot

- **Convergence:**

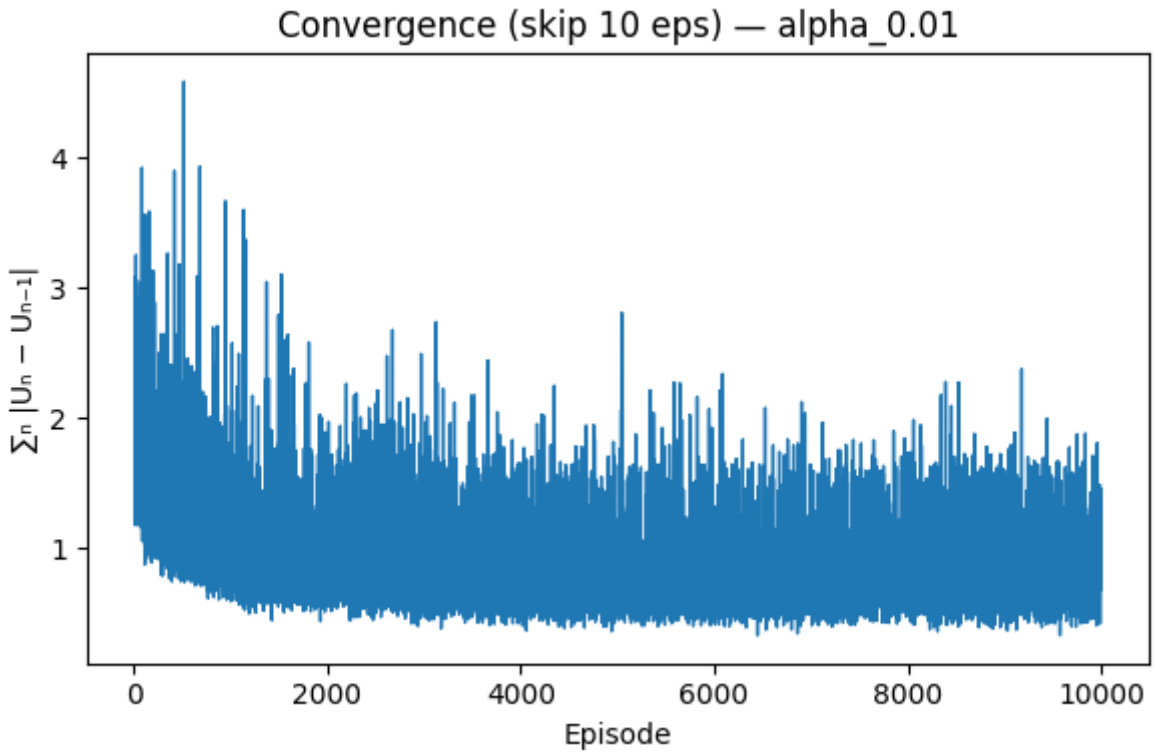

Figure 6.alpha 0.01 convergence plot

III. alpha 0.1 plots

- **Utility value function heatmap:**

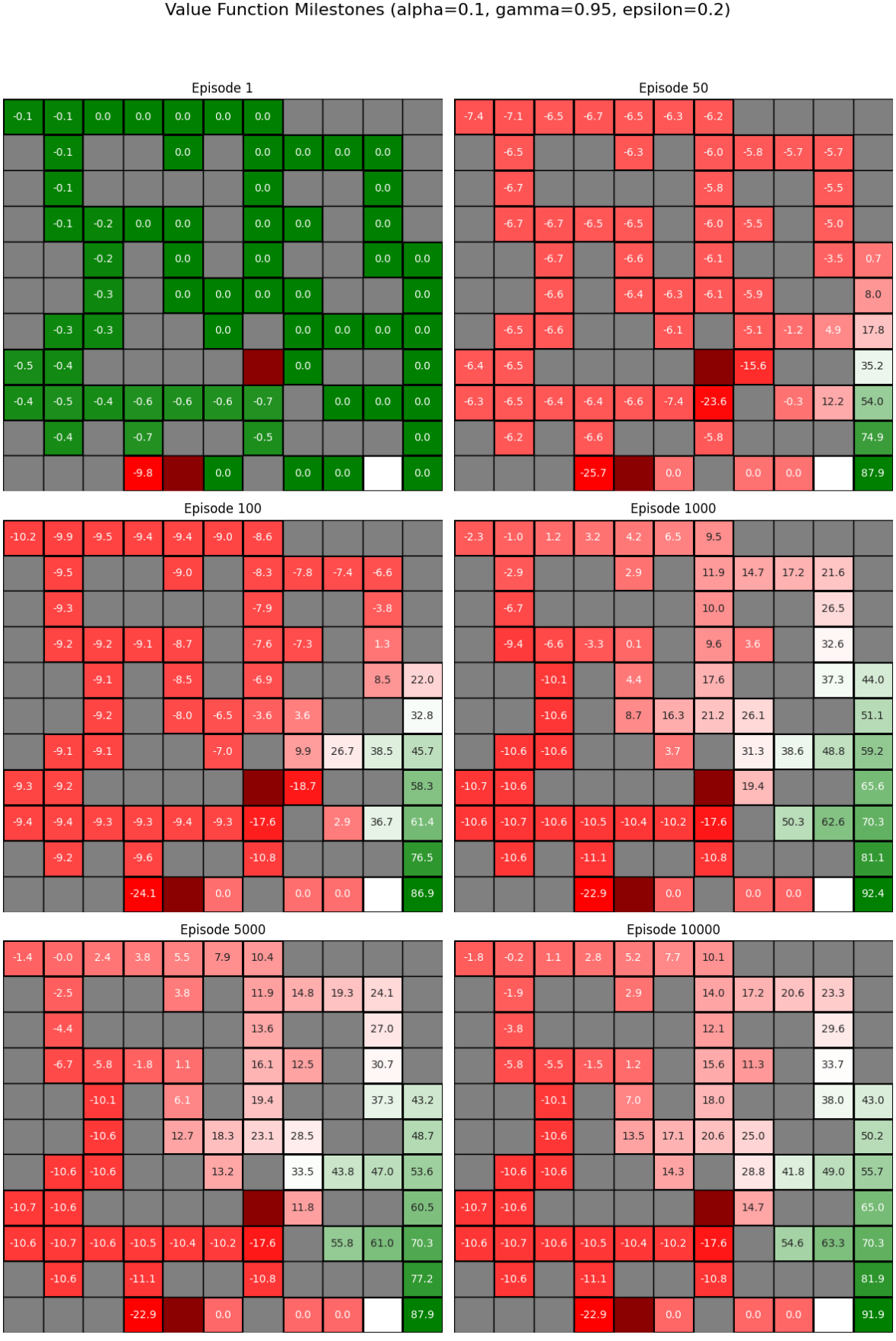

Figure 7.alpha 0.1 value plot

- **Policy:**

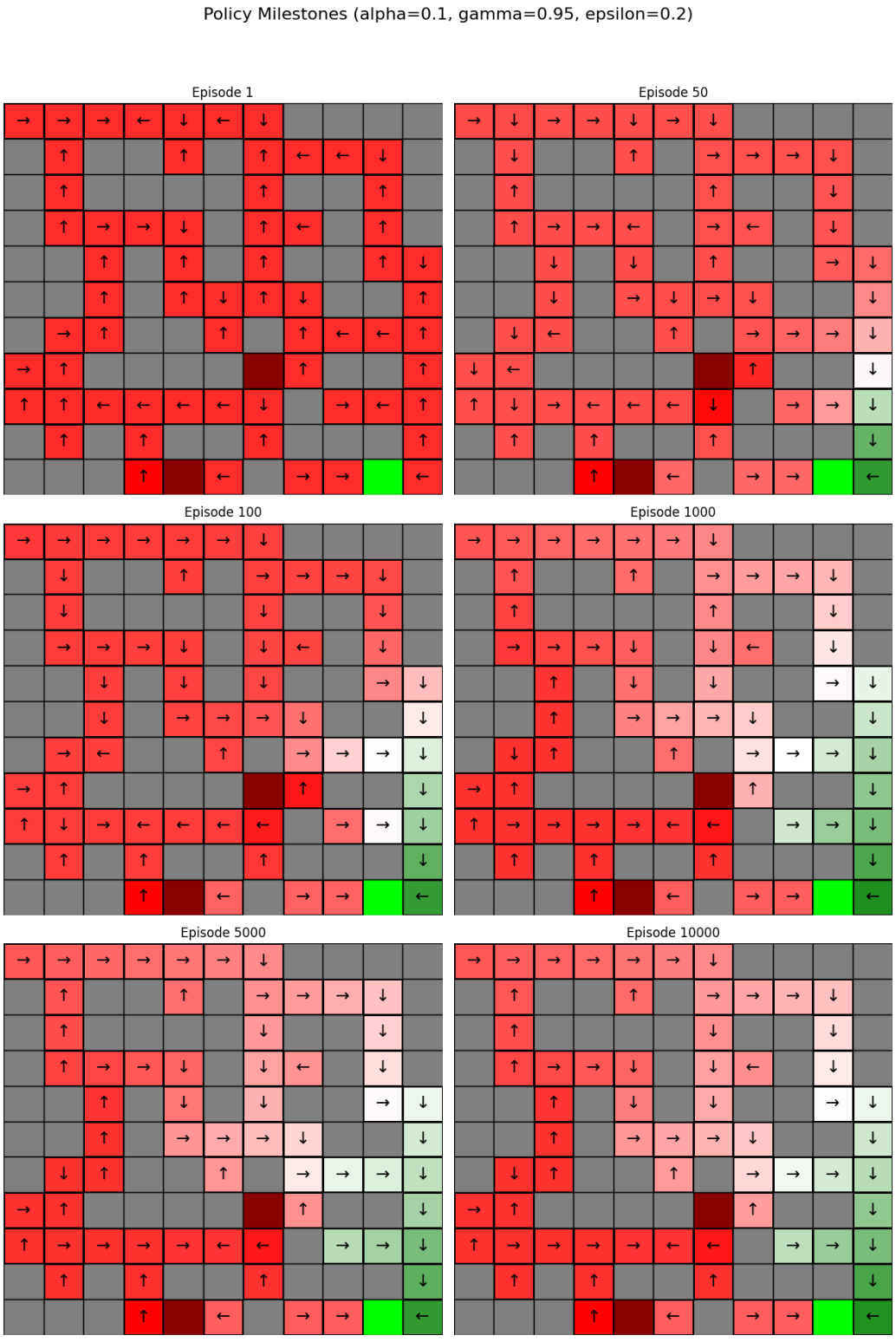

Figure 8.alpha 0.1 policy plot

- **Convergence:**

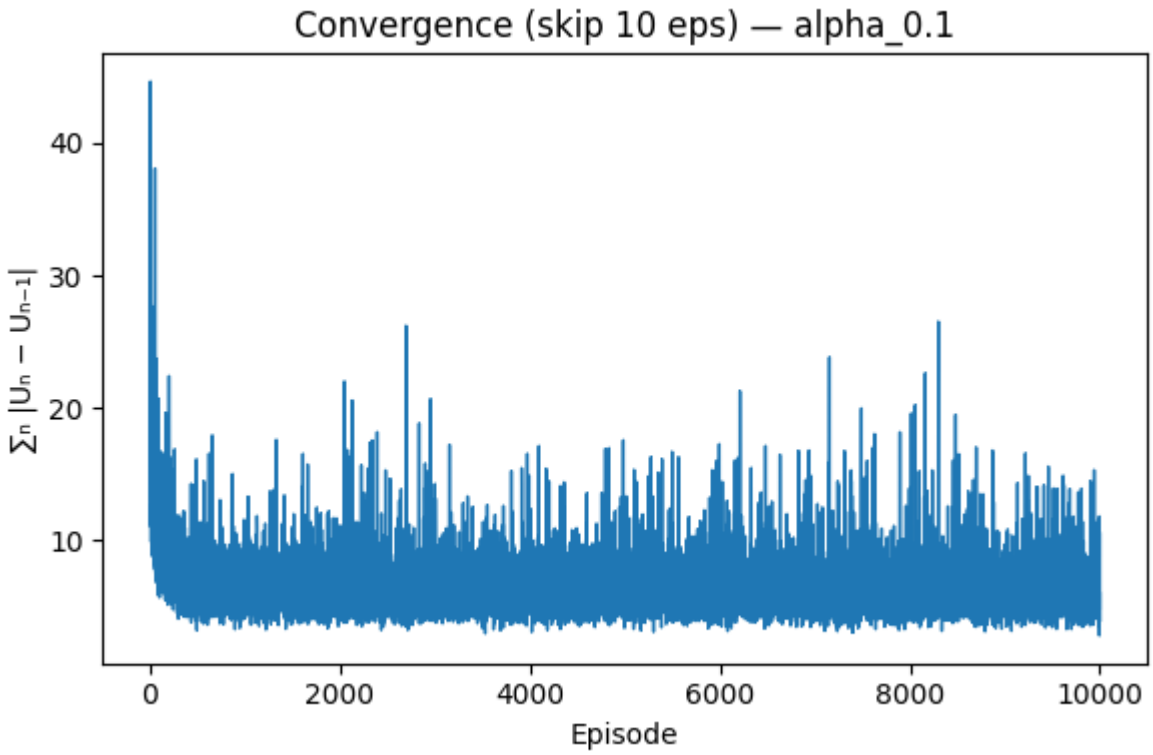

Figure 9.alpha 0.1 convergence plot

IV. alpha 0.5 plots

- **Utility value function heatmap:**

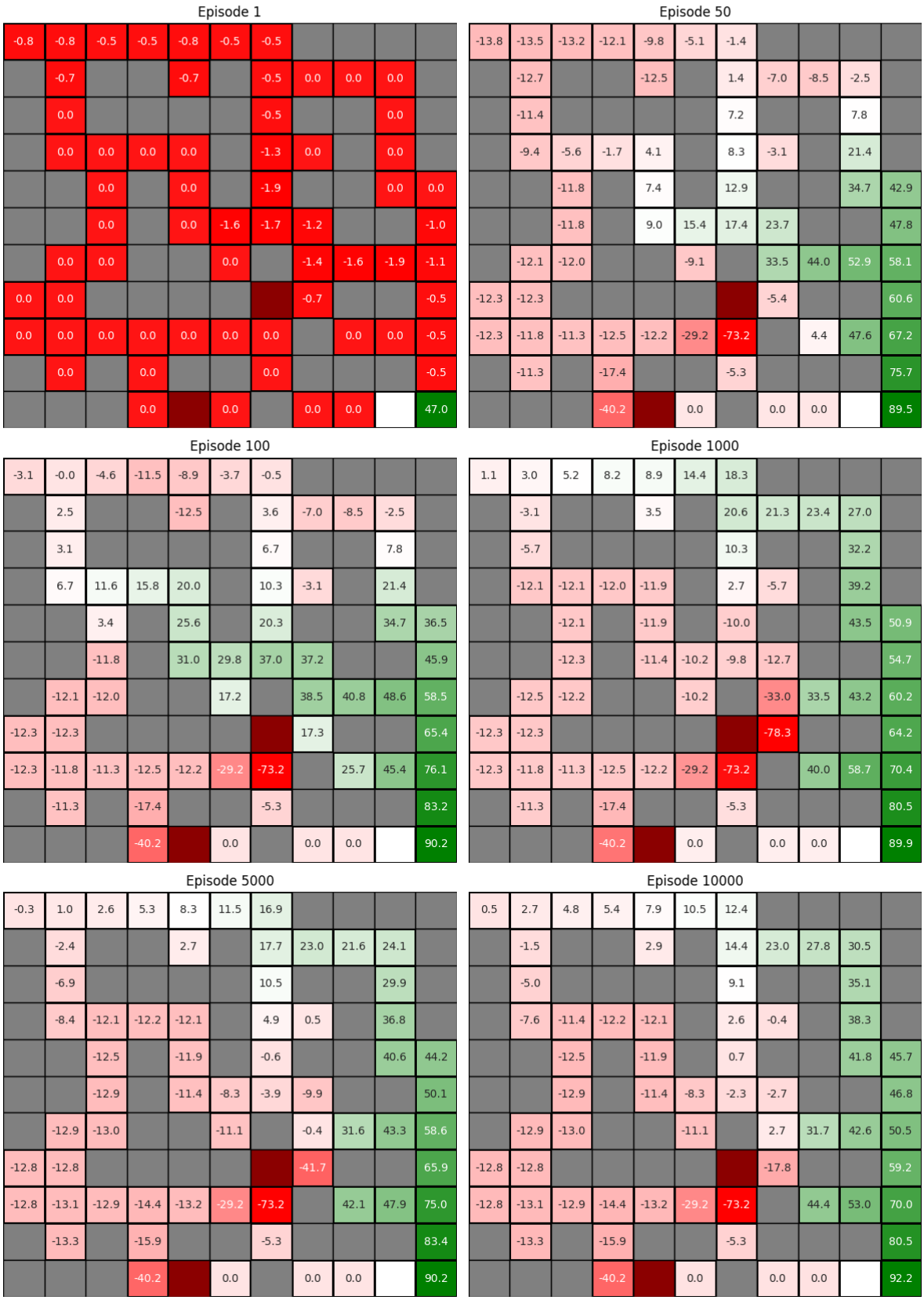

Figure 10.alpha 0.5 value plot

- **Policy:**

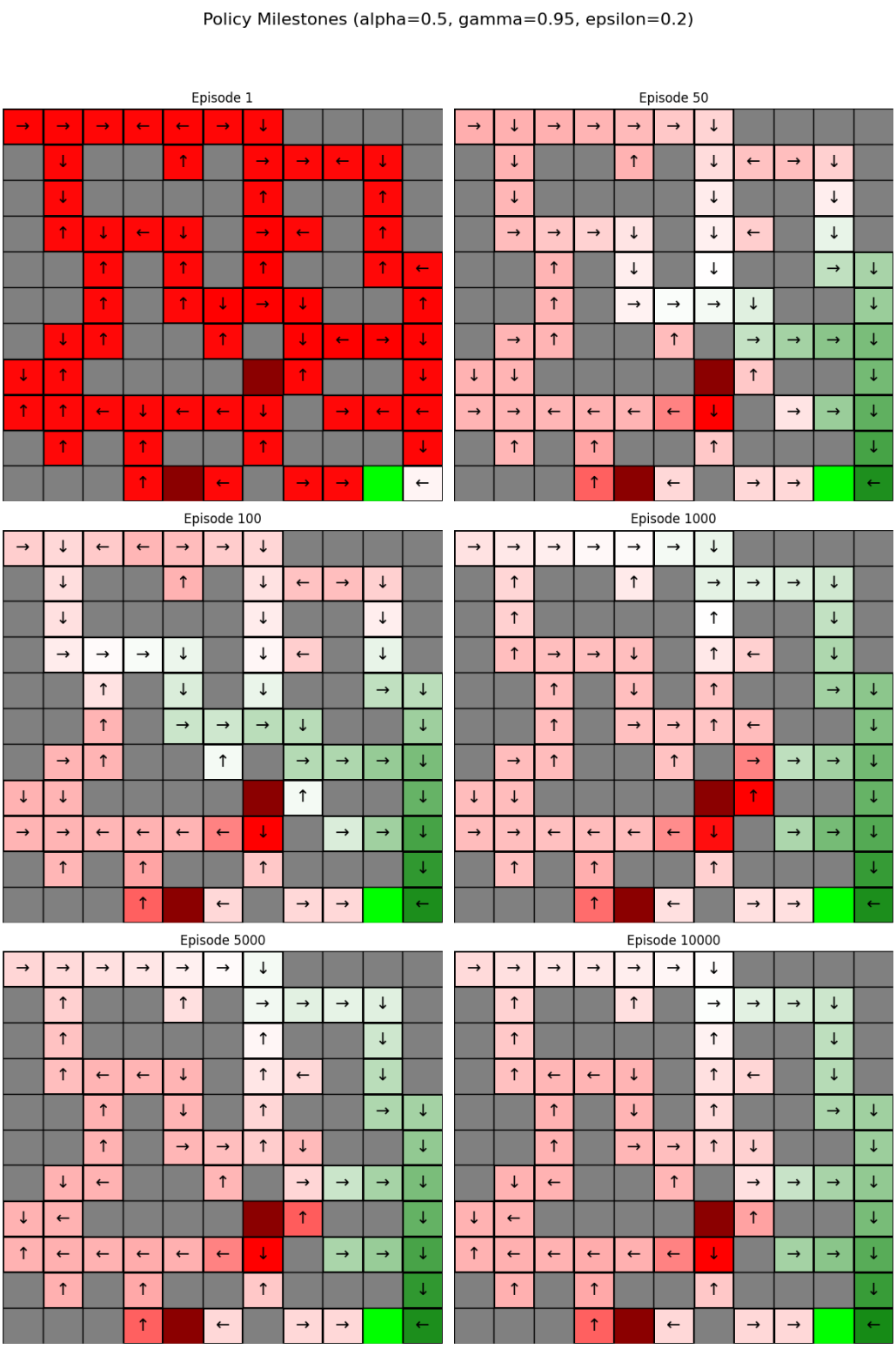

Figure 11.alpha 0.5 policy plot

- **Convergence:**

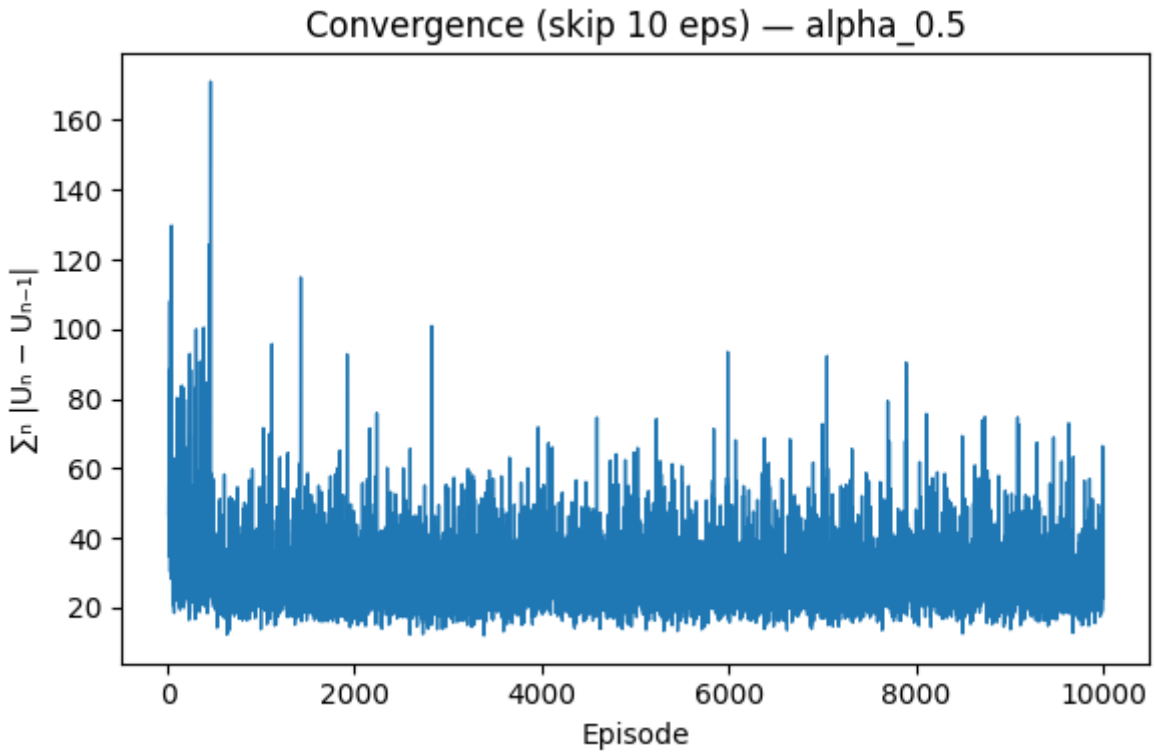

Figure 12.alpha 0.5 convergence plot

V. alpha 1.0 plots

- **Utility value function heatmap:**

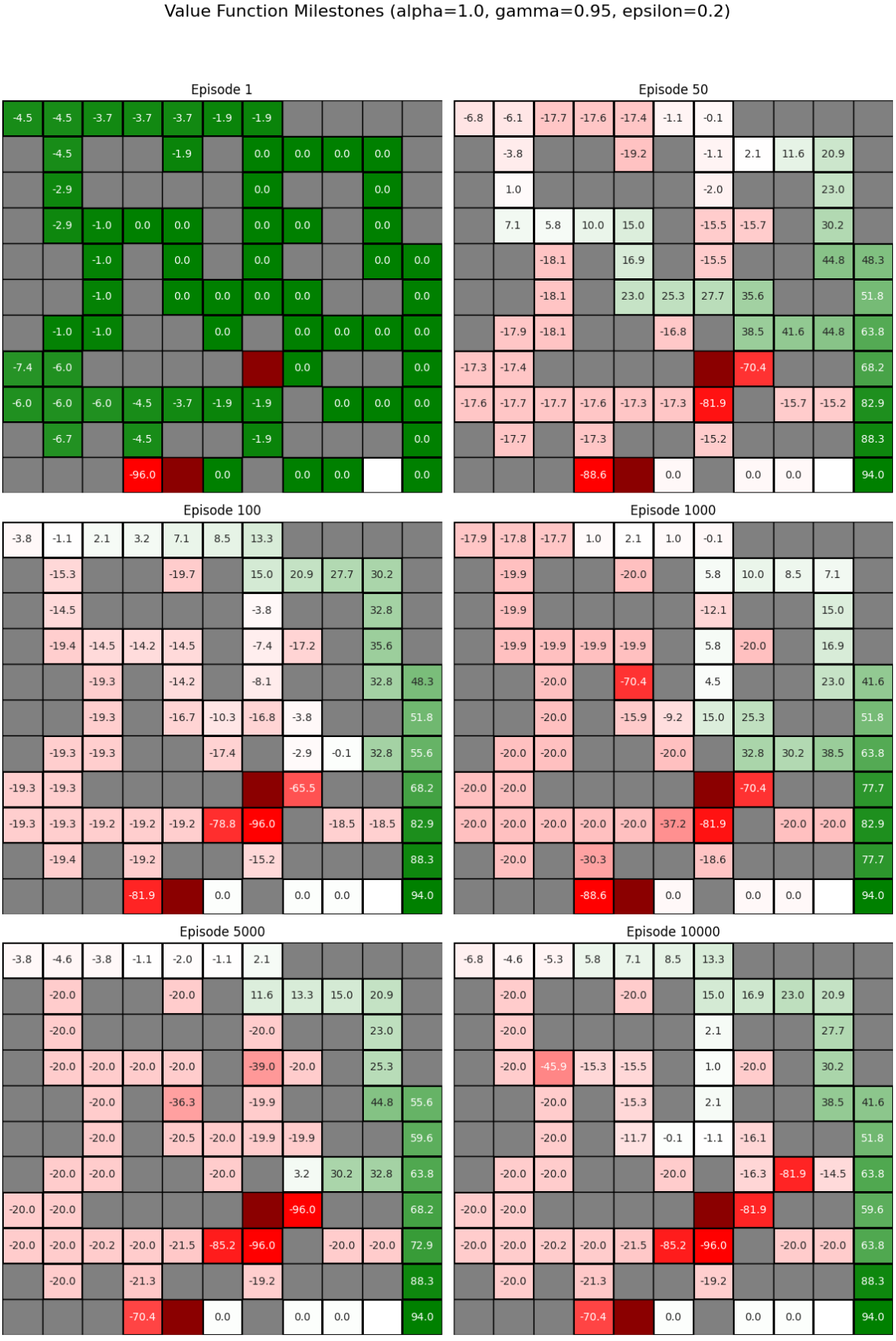

Figure 13.alpha 1.0 value plot

- **Policy:**

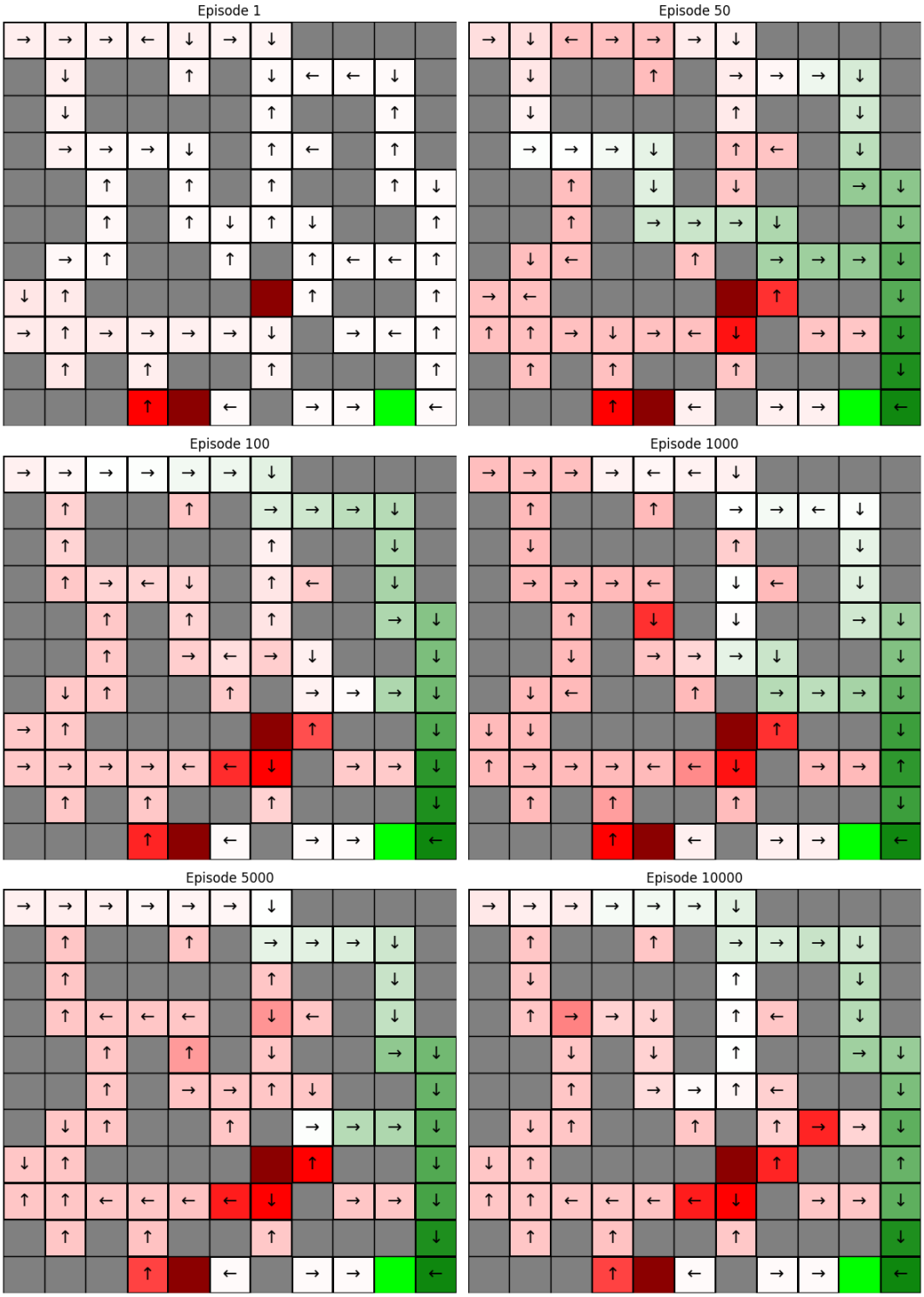

Figure 14.alpha 1.0 policy plot

- **Convergence:**

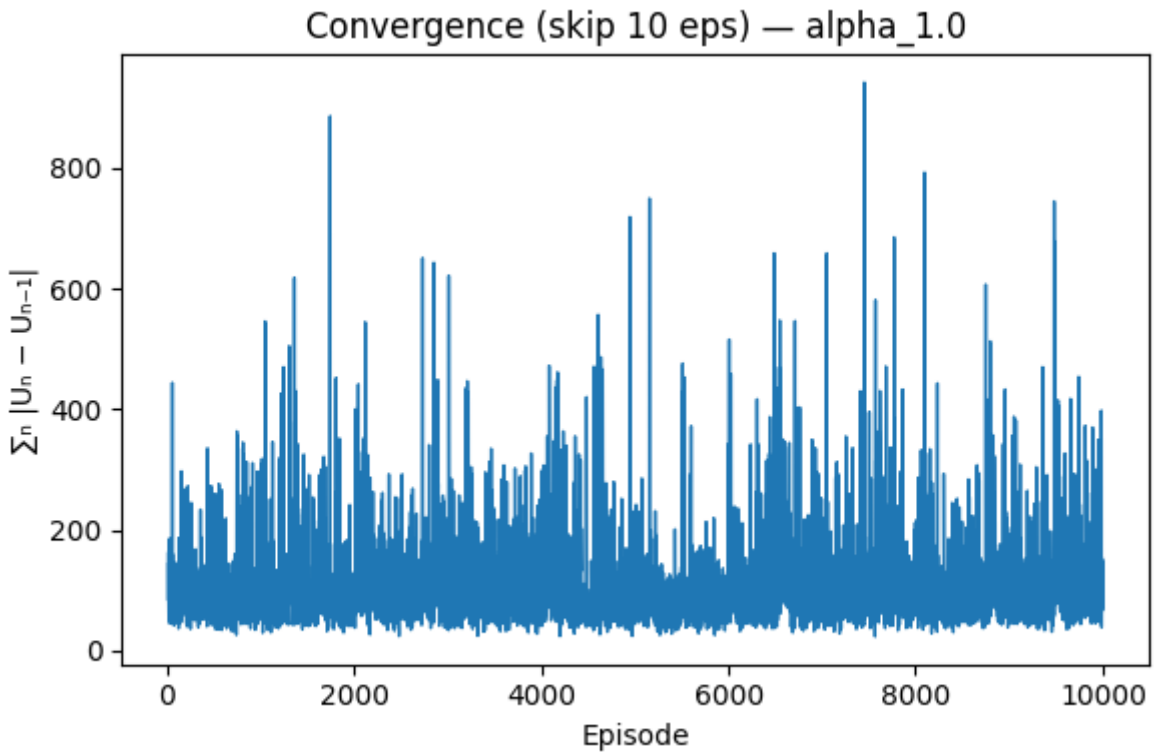

Figure 15.alpha 1.0 convergence plot

VI. epsilon 0.0 plots

- **Utility value function heatmap:**

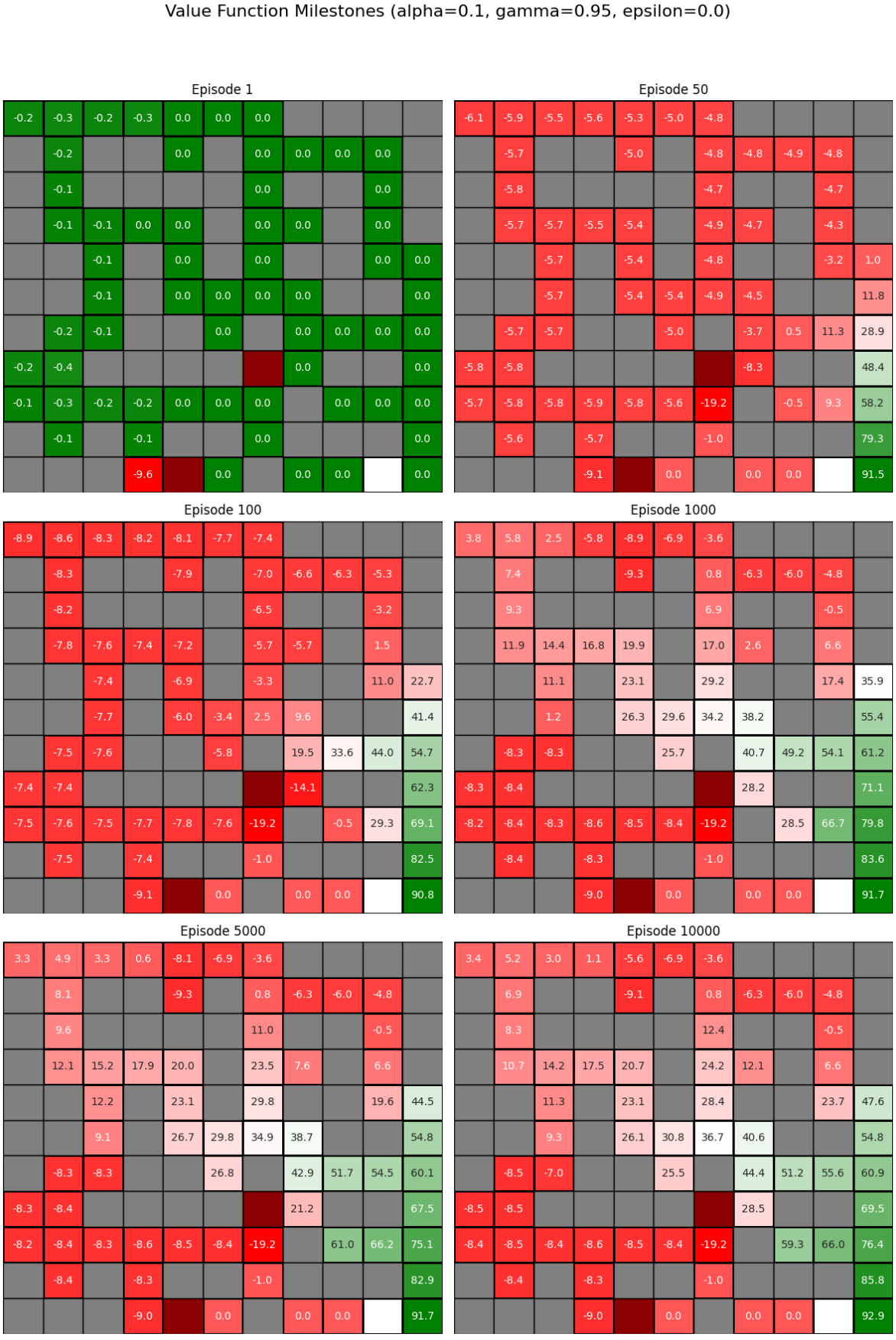

Figure 16.epsilon 0.0 value plot

- **Policy:**

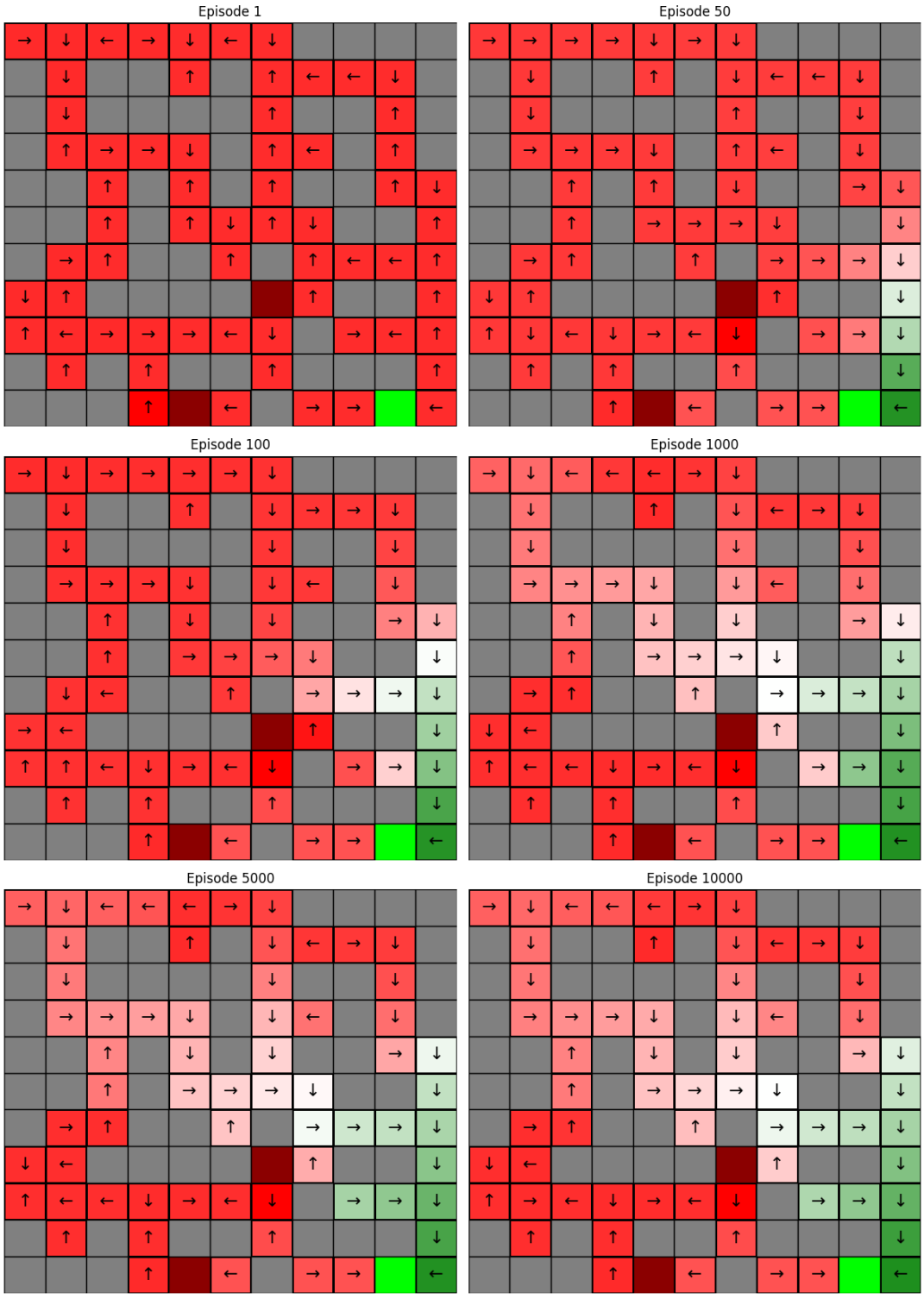

Figure 17.epsilon 0.0 policy plot

- **Convergence:**

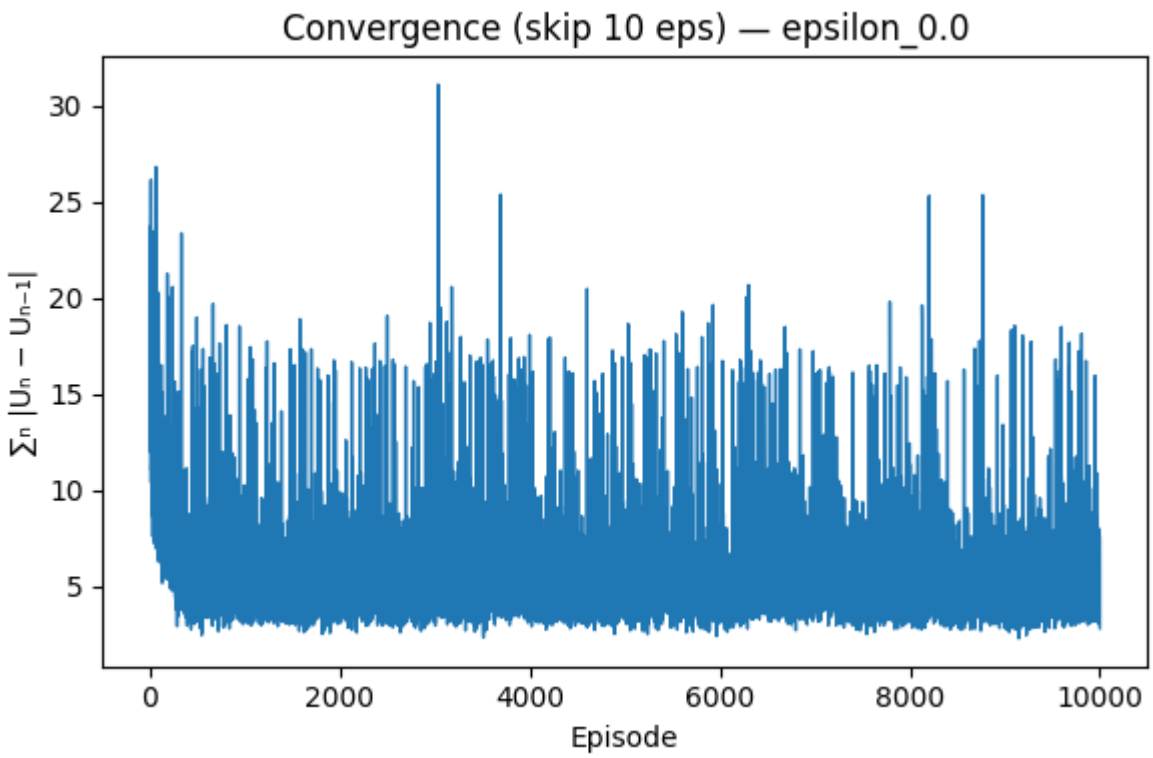

Figure 18.epsilon 0.0 convergence plot

VII. epsilon 0.2 plots

- **Utility value function heatmap:**

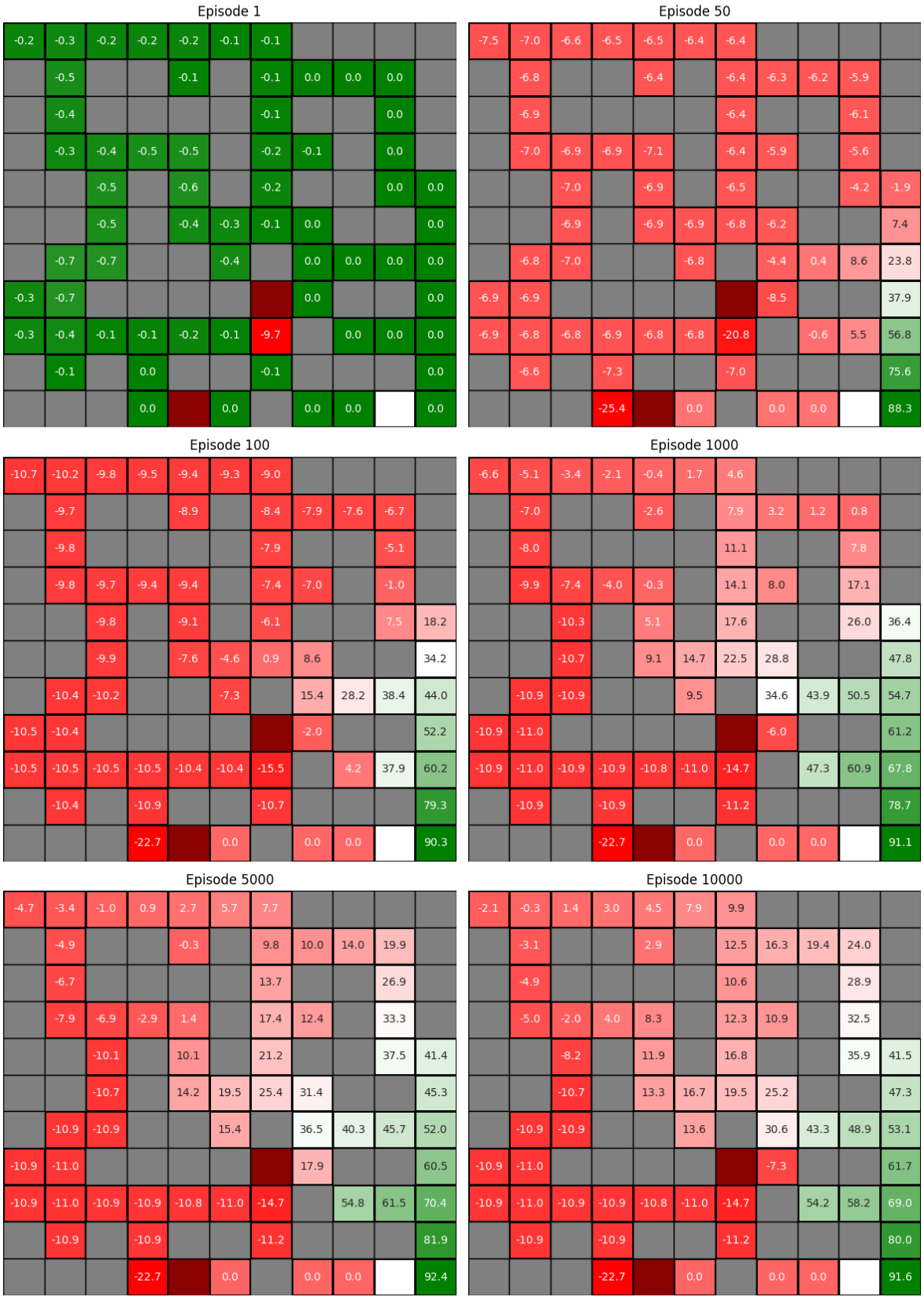

Figure 19.epsilon 0.2 value plot

- **Policy:**

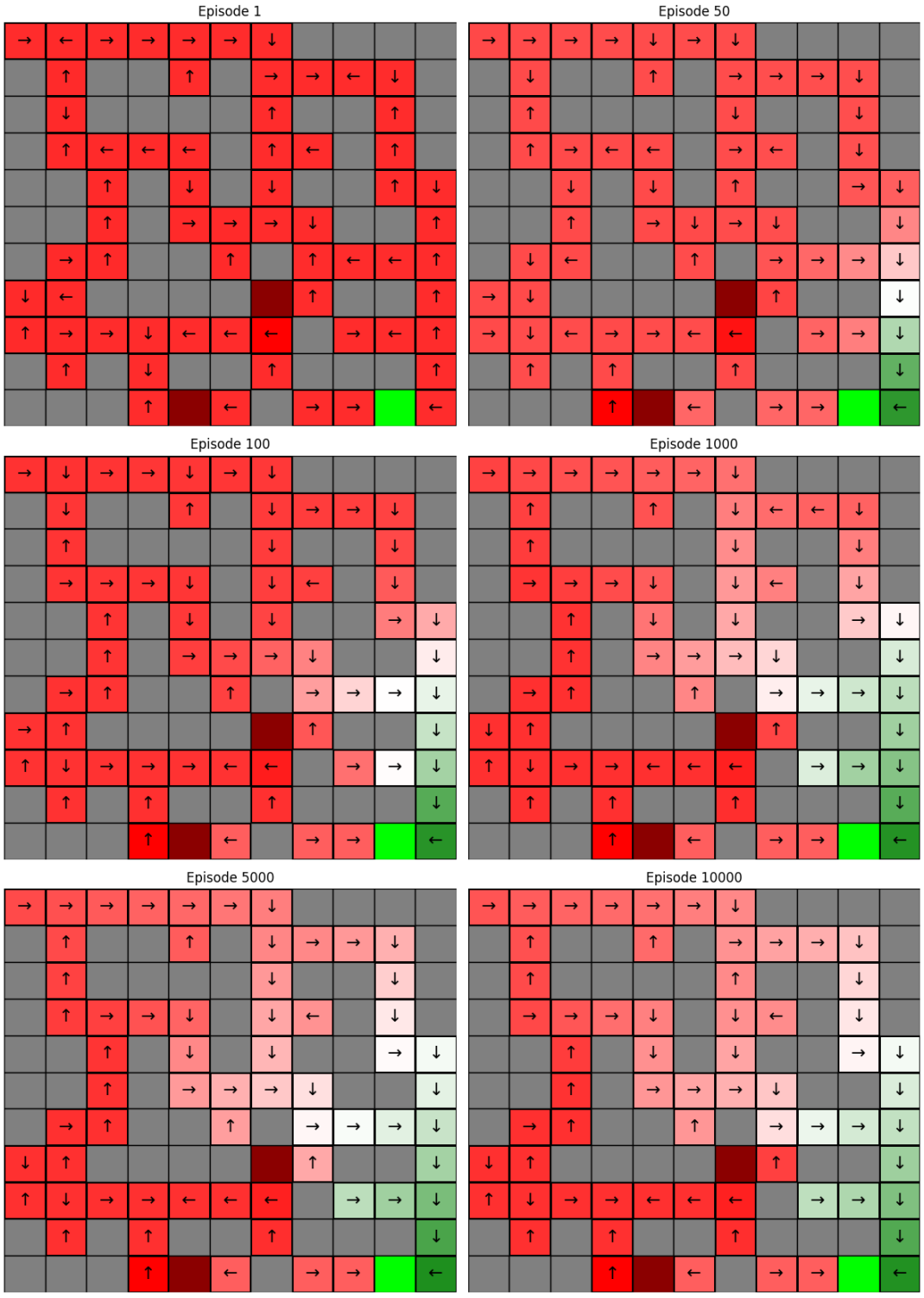

Figure 20.epsilon 0.2 policy plot

- **Convergence:**

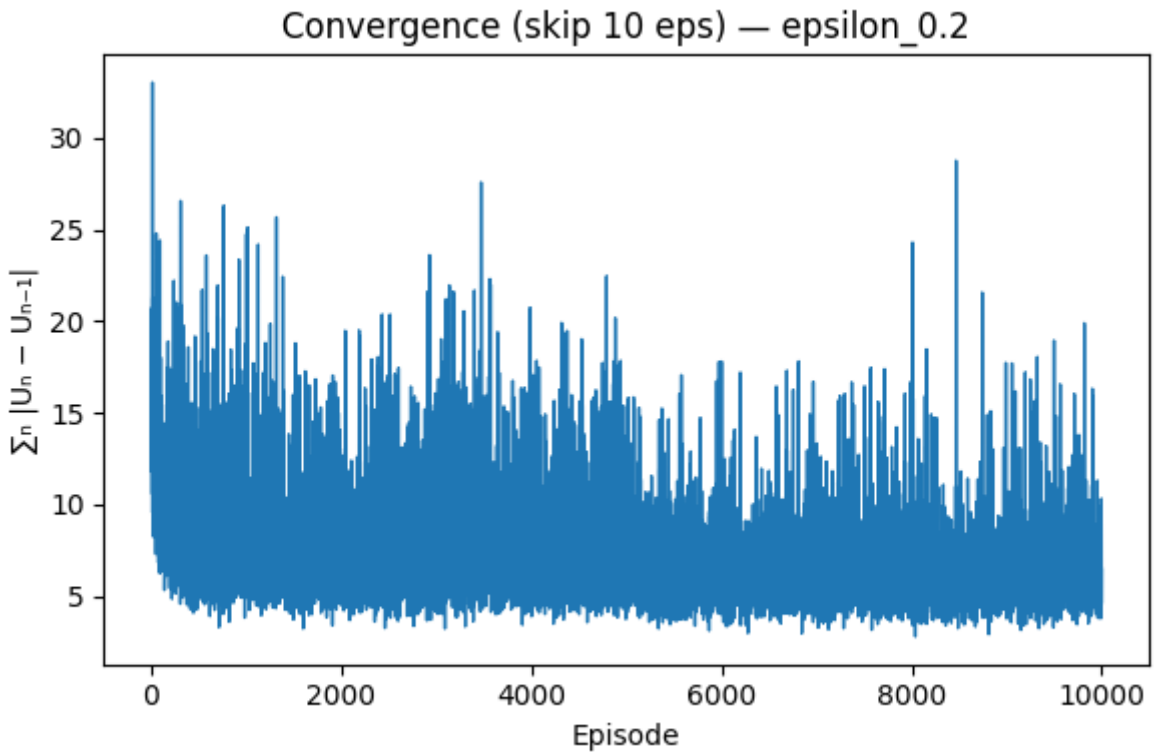

Figure 21.epsilon 0.2 convergence plot

VIII. epsilon 0.5 plots

- **Utility value function heatmap:**

Figure 22.epsilon 0.5 value plot

- **Policy:**

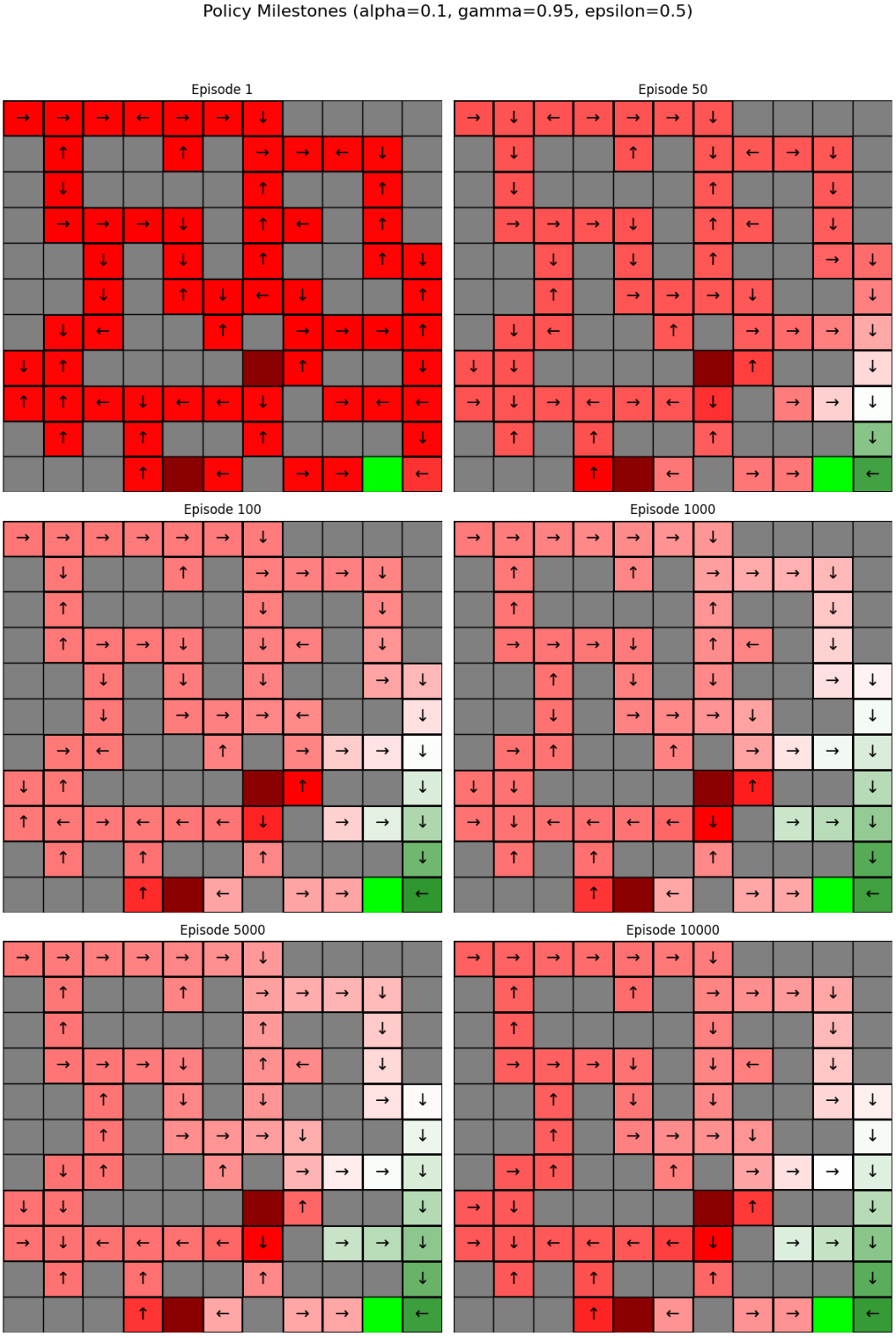

Figure 23.epsilon 0.5 policy plot

- **Convergence:**

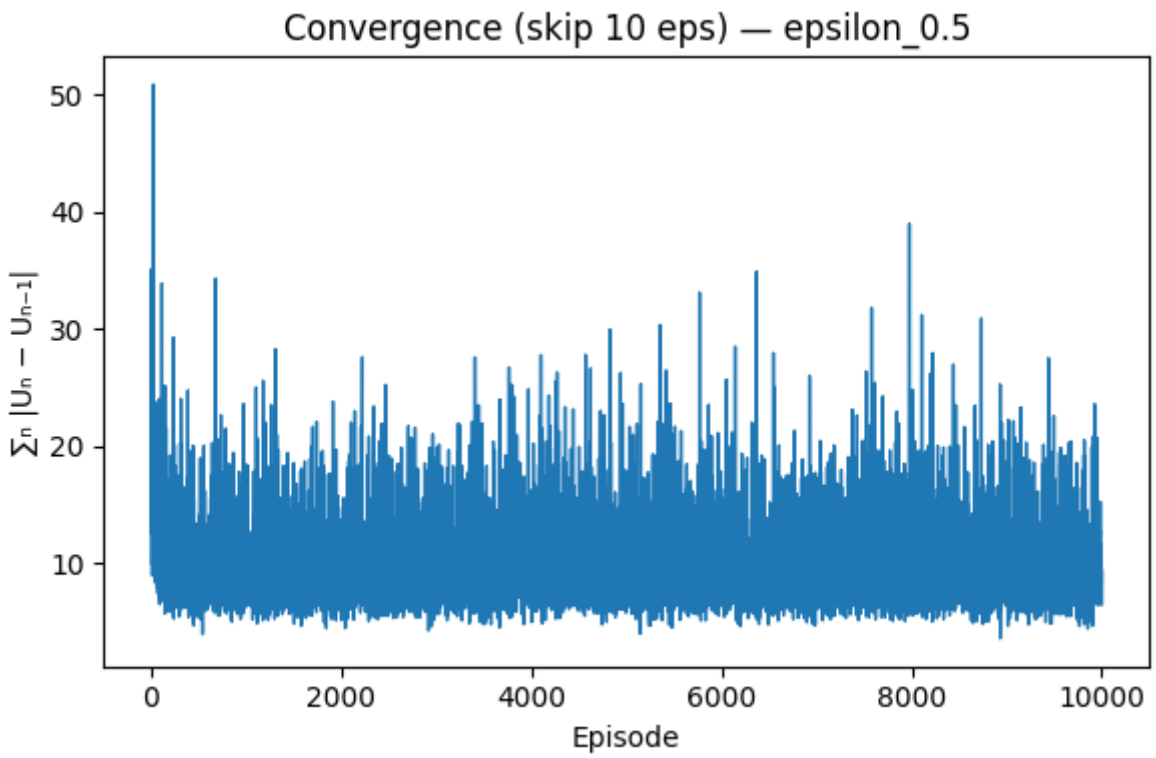

Figure 24.epsilon 0.5 convergence plot

IX. epsilon 0.8 plots

- **Utility value function heatmap:**

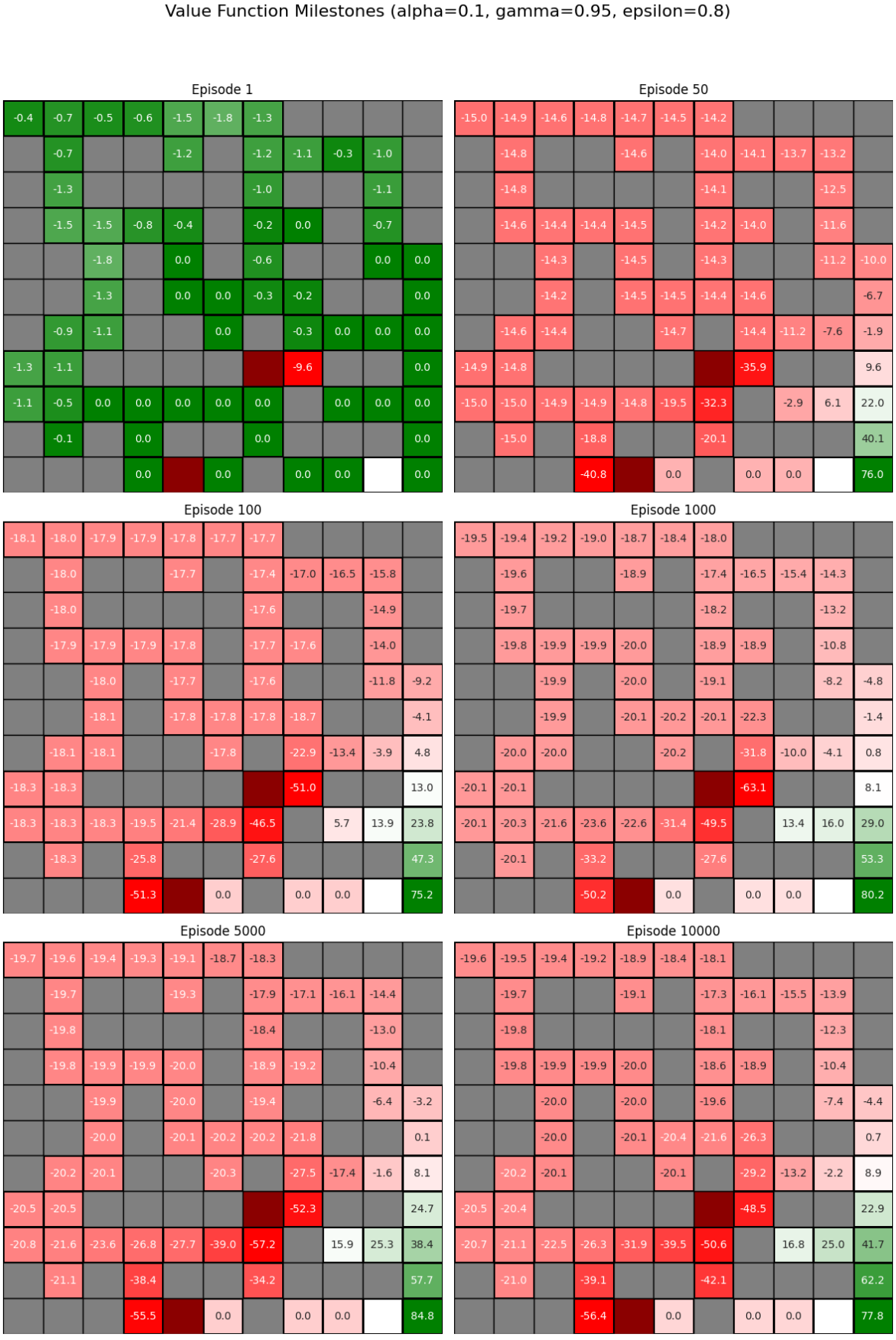

Figure 25.epsilon 0.8 value plot

- **Policy:**

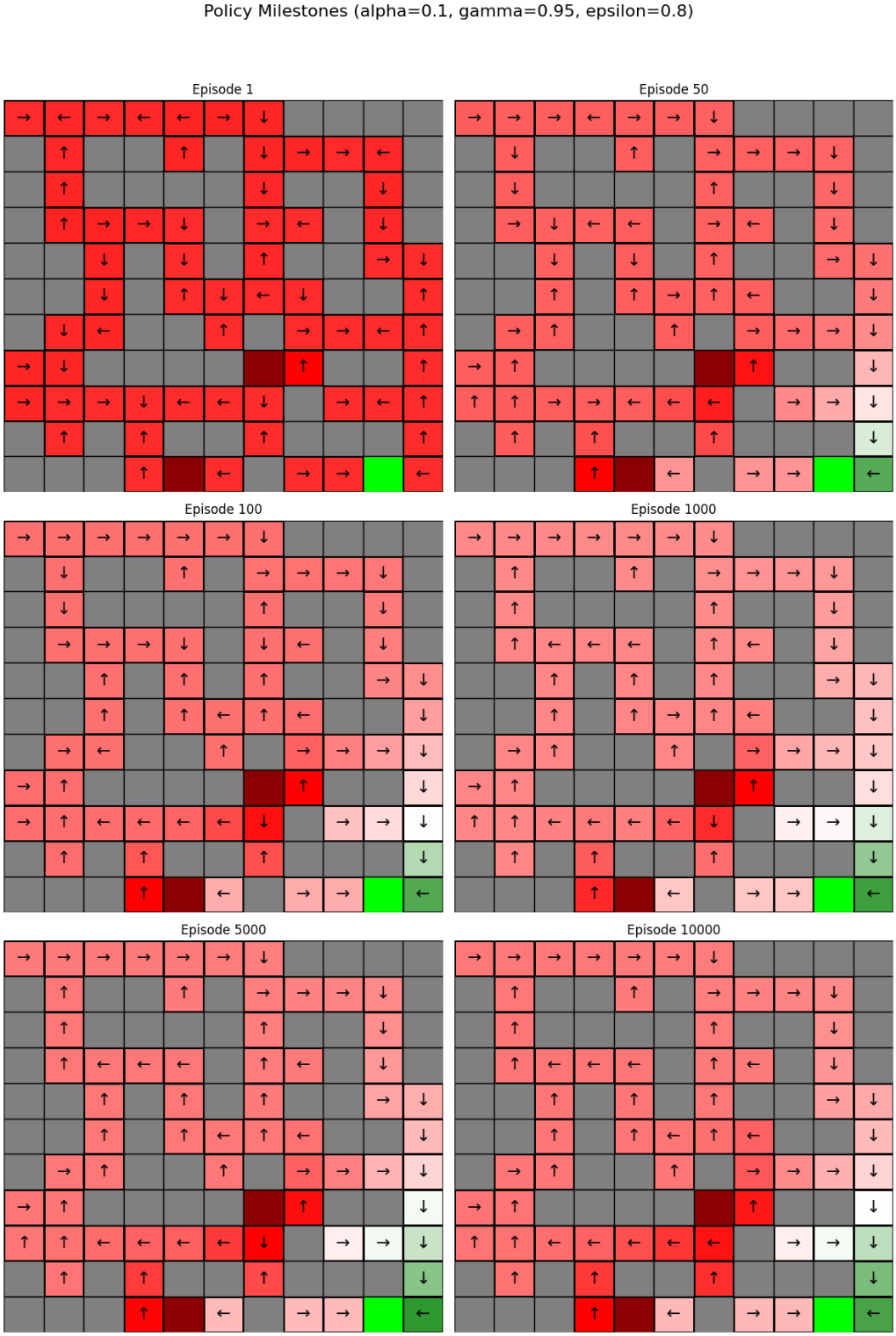

Figure 26.epsilon 0.8 policy plot

- **Convergence:**

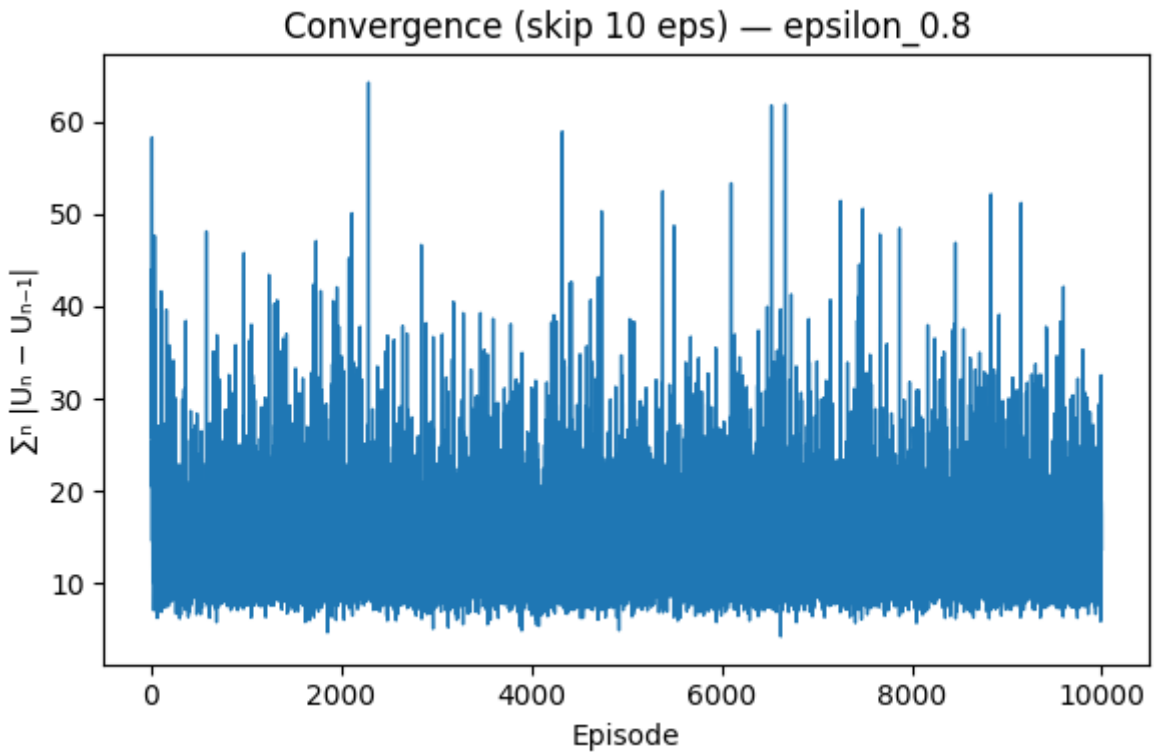

Figure 27.epsilon 0.8 convergence plot

X. epsilon 1.0 plots

- **Utility value function heatmap:**

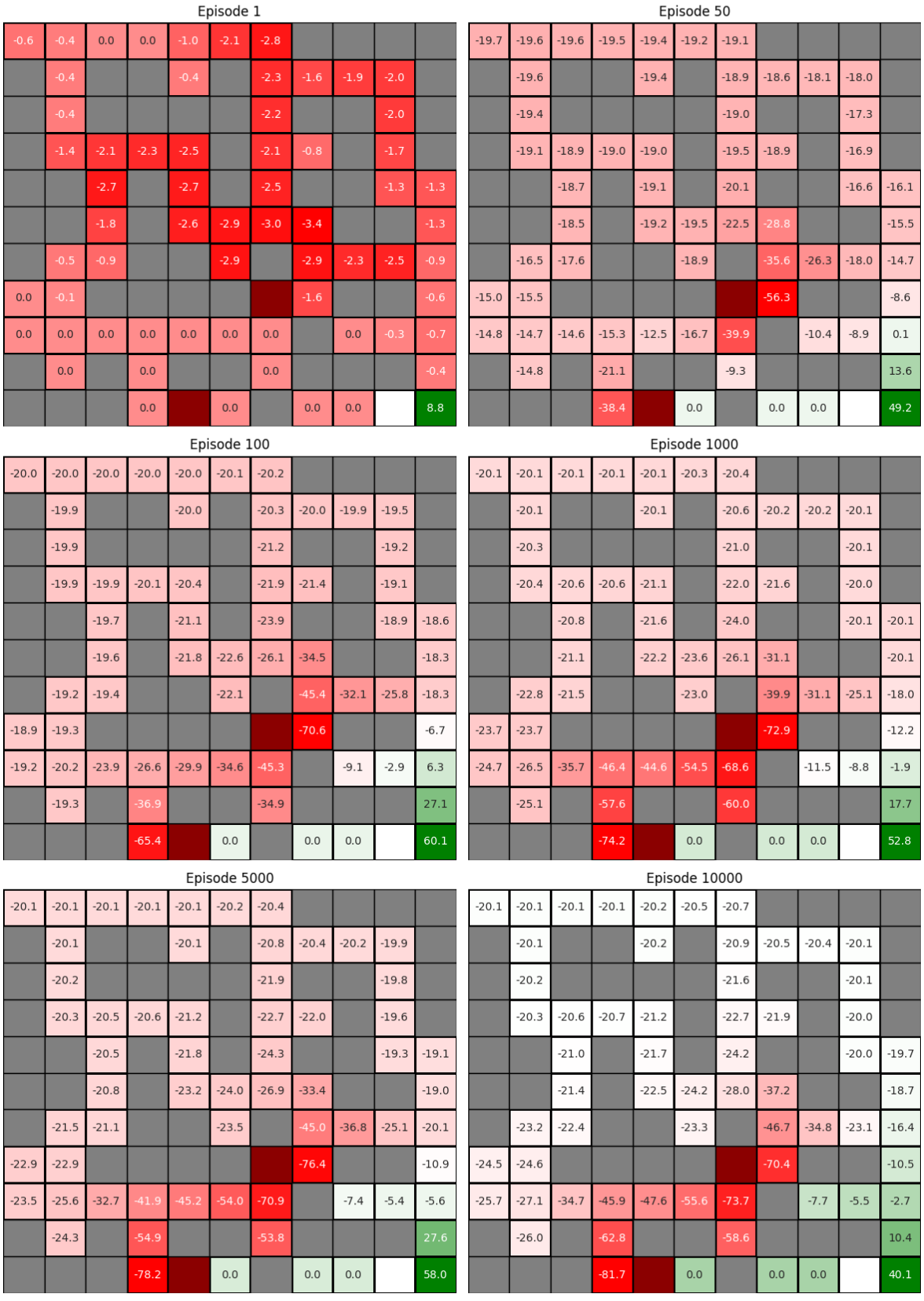

Figure 28.epsilon 1.0 value plot

- **Policy:**

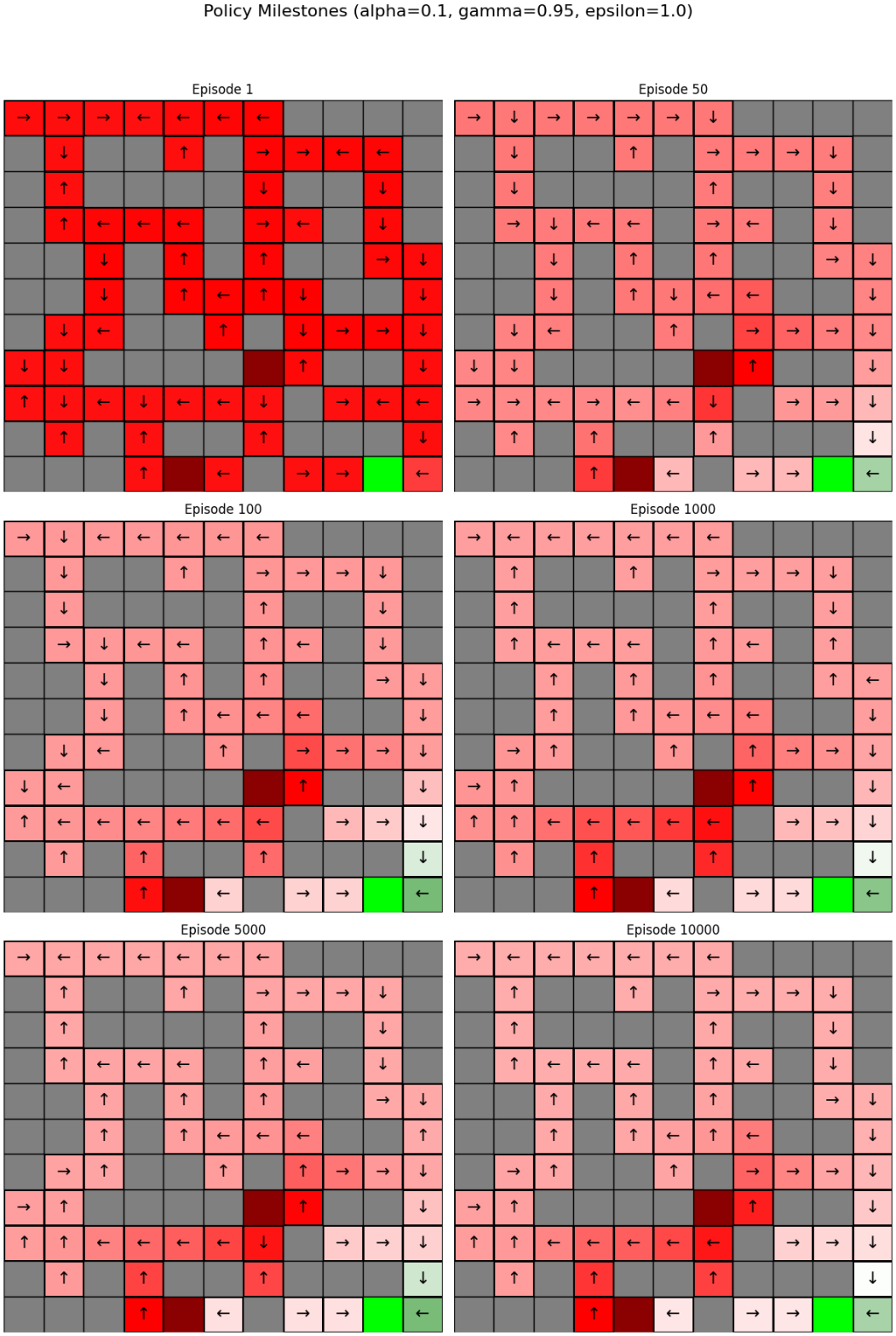

Figure 29.epsilon 1.0 policy plot

- **Convergence:**

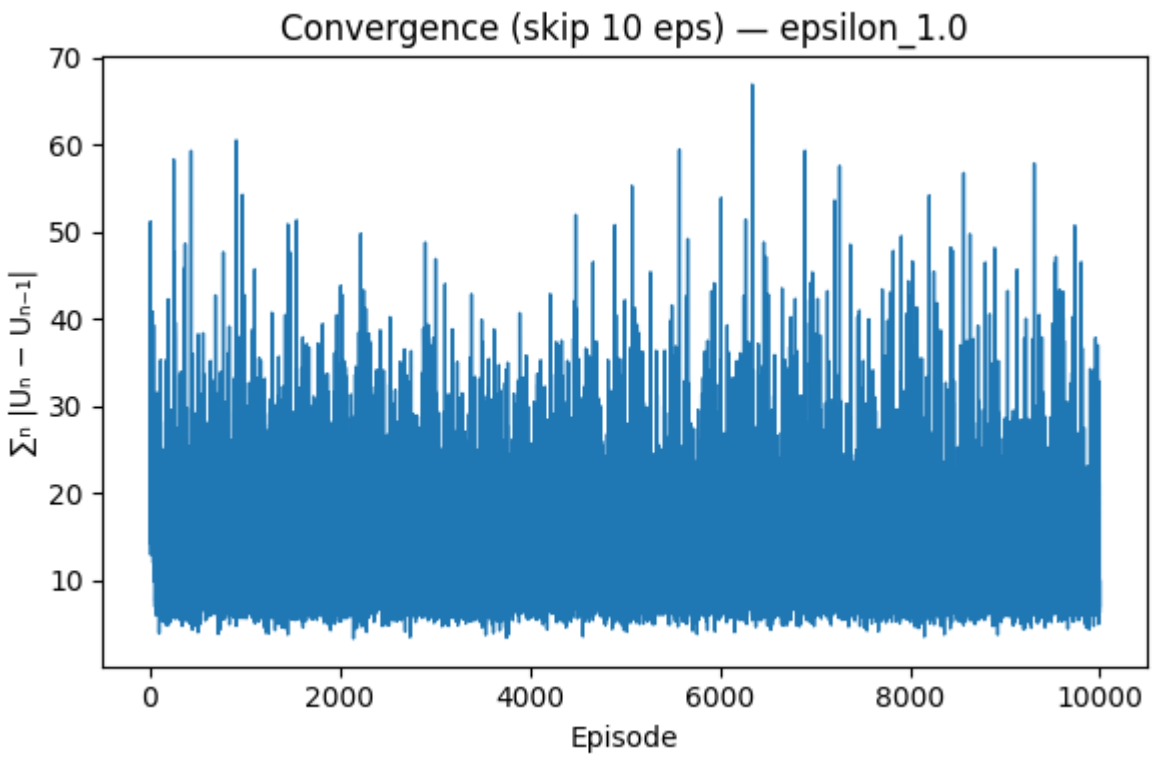

Figure 30.epsilon 1.0 convergence plot

XI. gamma 0.1 plots

- **Utility value function heatmap:**

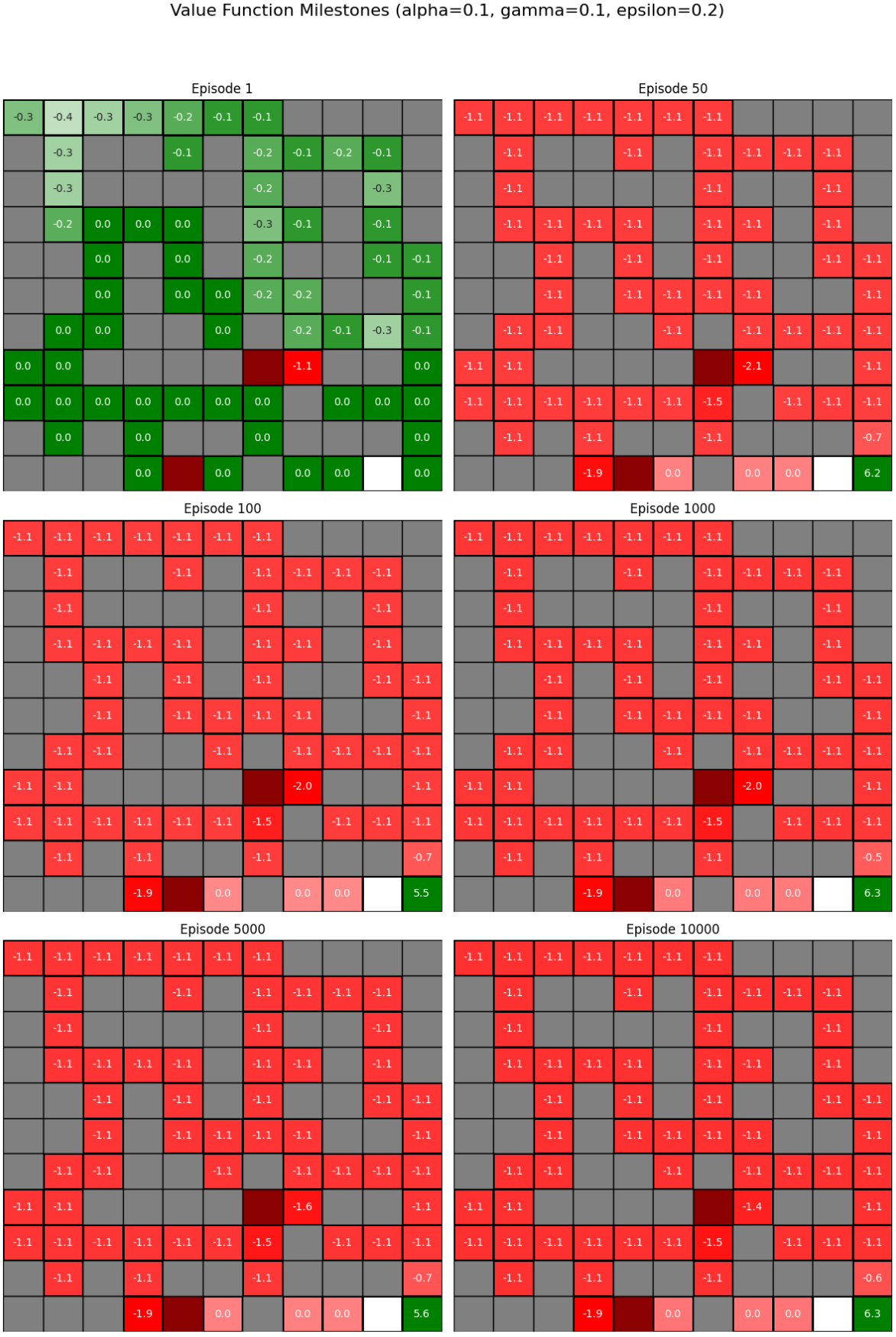

Figure 31.gamma 0.1 value plot

- **Policy:**

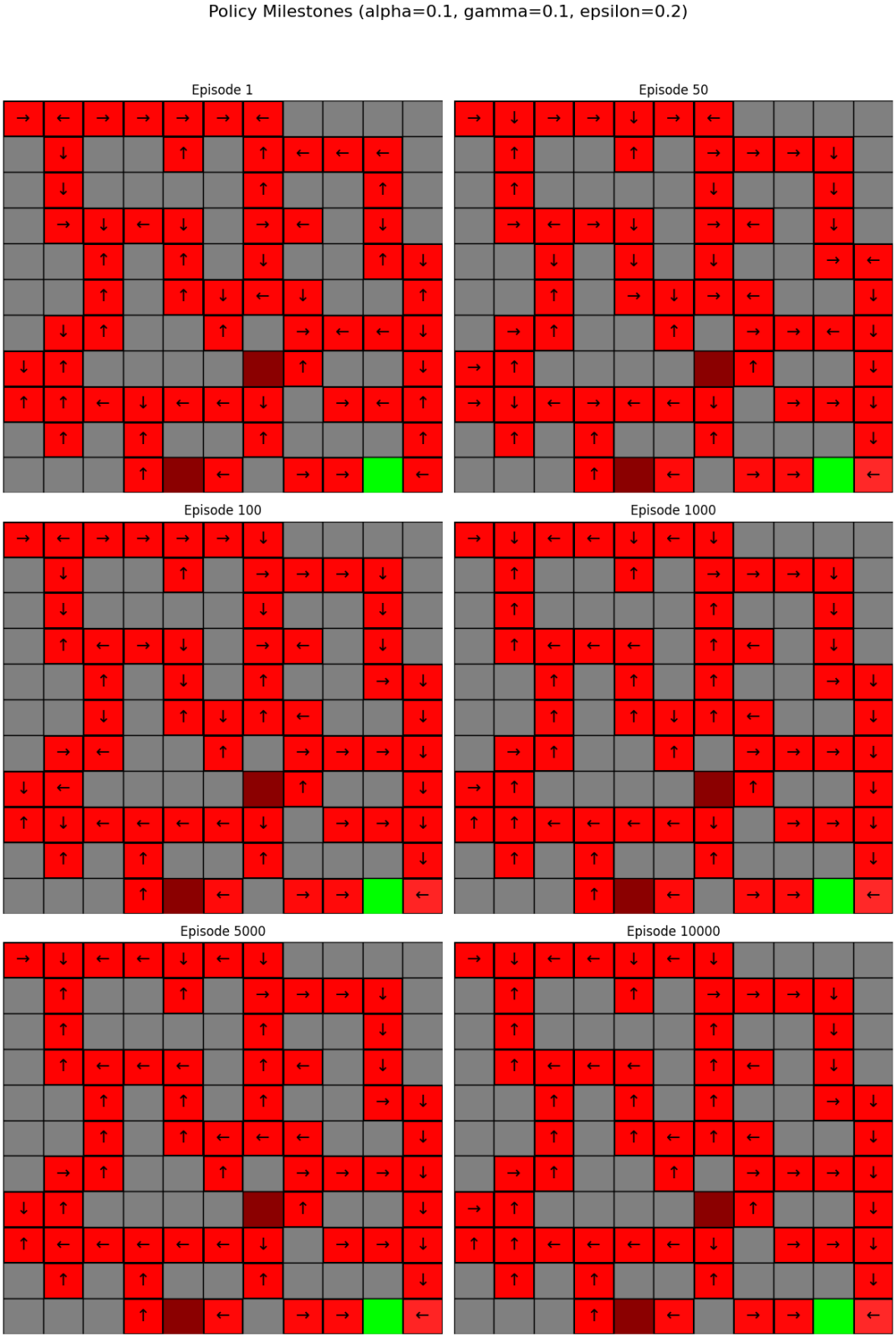

Figure 32.gamma 0.1 policy plot

- **Convergence:**

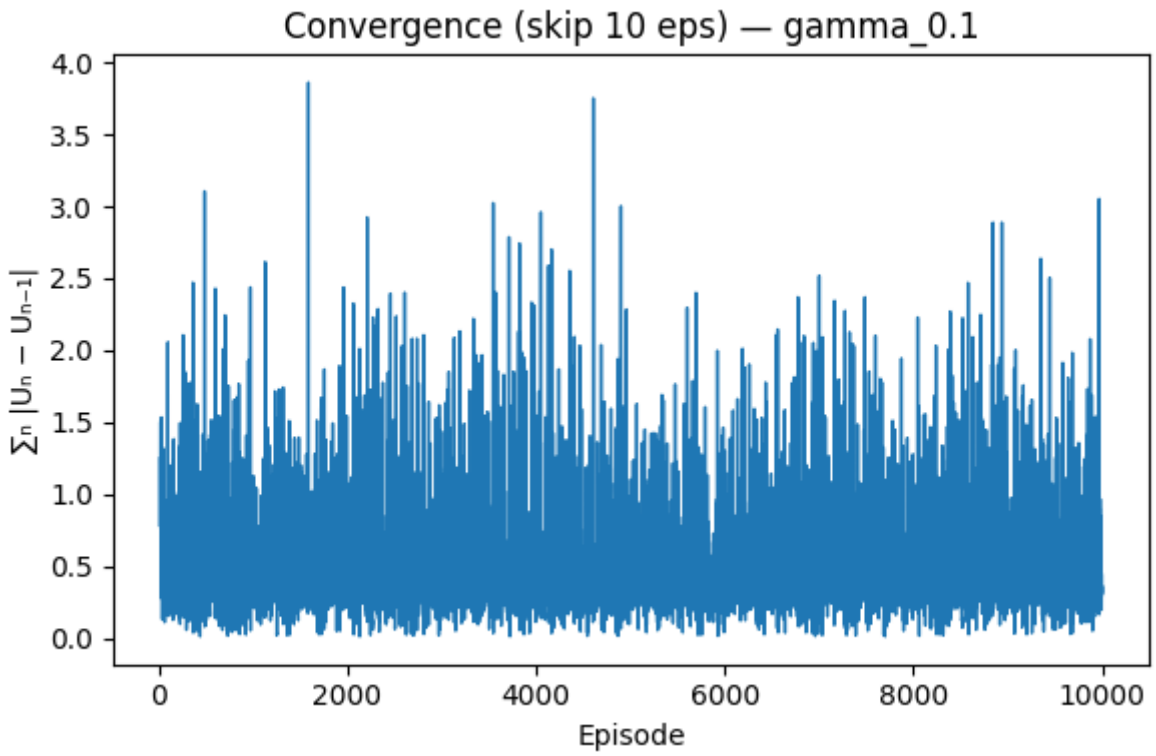

Figure 33.gamma 0.1 convergence plot

XII. gamma 0.5 plots

- **Utility value function heatmap:**

Figure 34.gamma 0.5 value plot

- **Policy:**

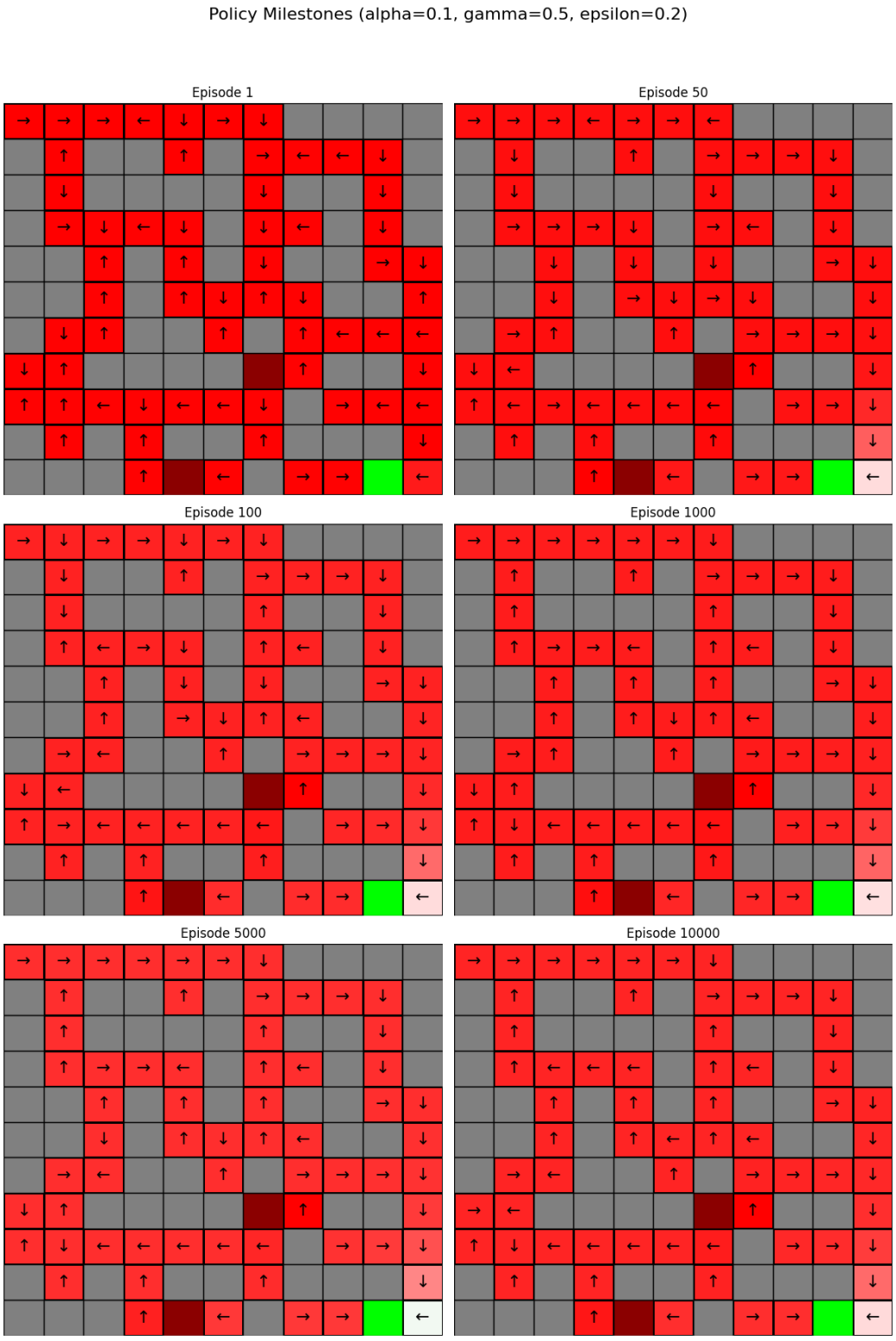

Figure 35.gamma 0.5 policy plot

- **Convergence:**

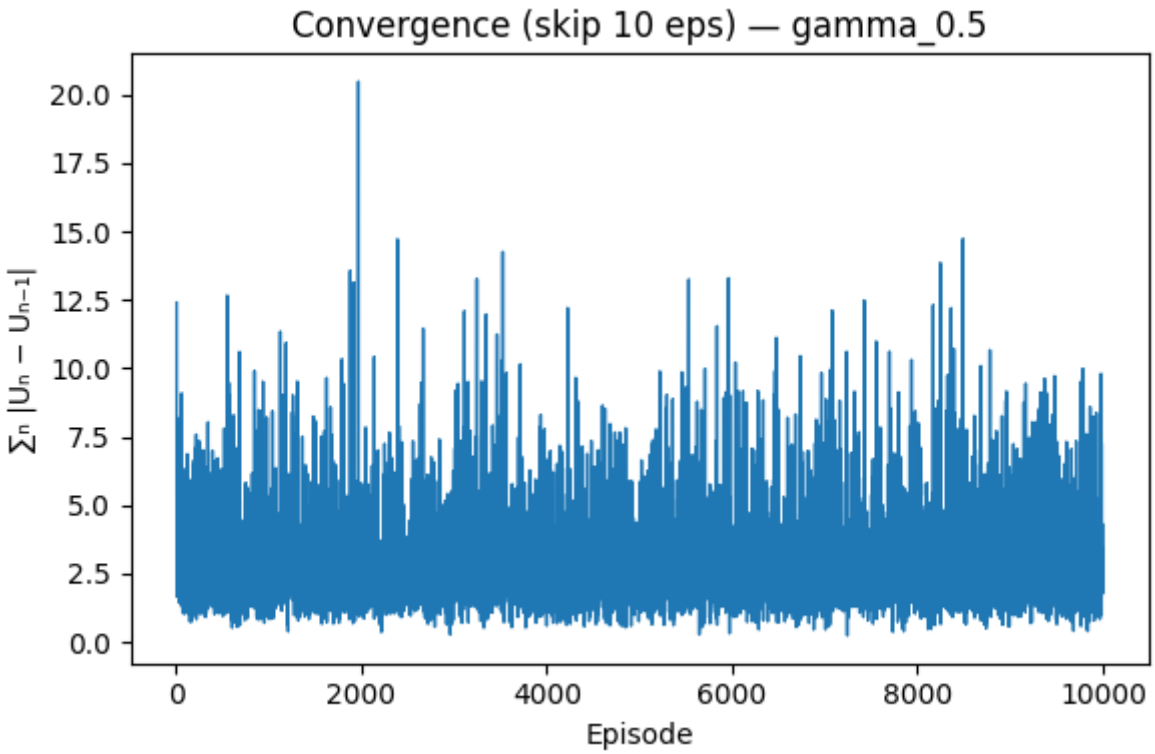

Figure 36.gamma 0.5 convergence plot

XIII. gamma 0.25 plots

- **Utility value function heatmap:**

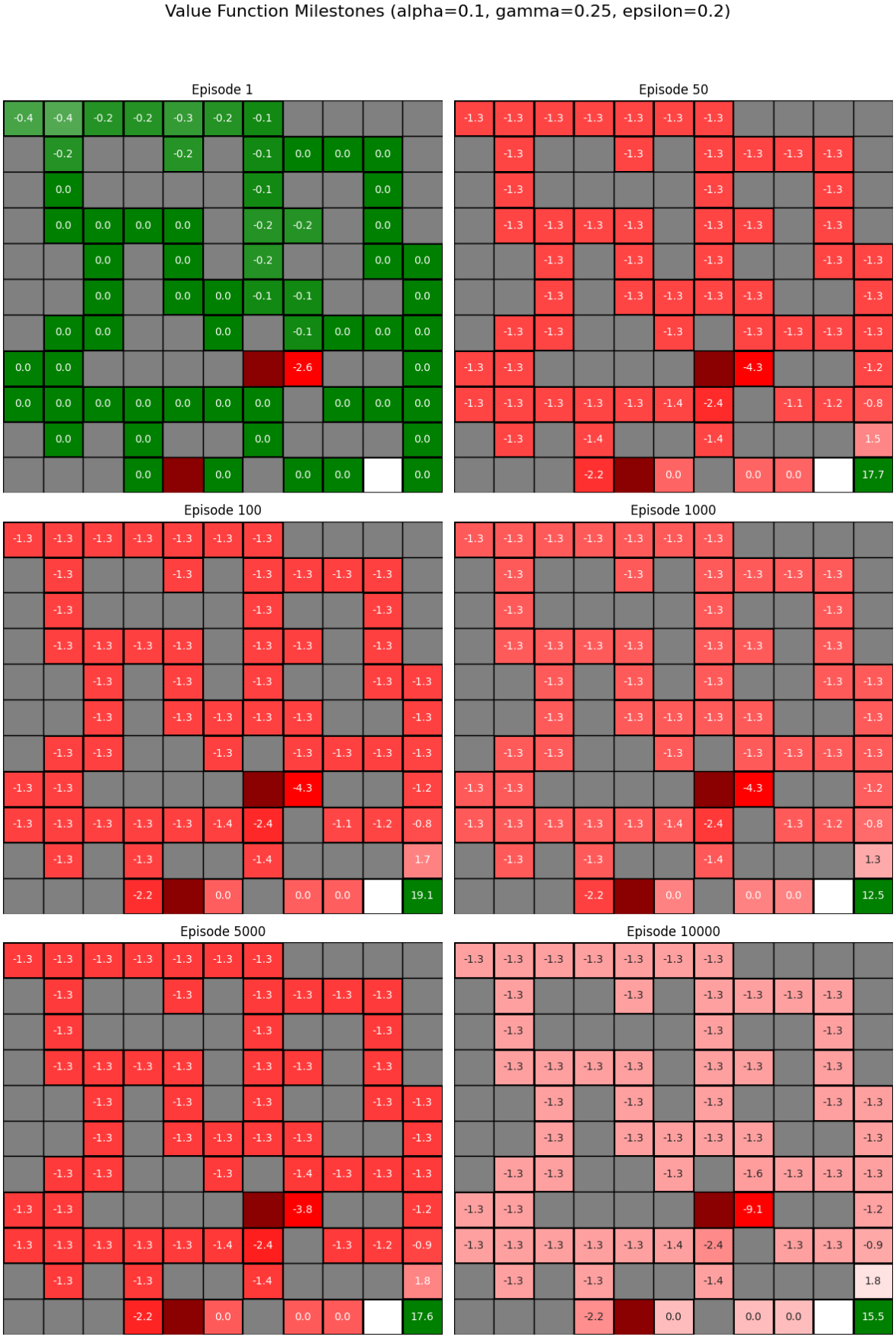

Figure 37.gamma 0.25 value plot

- **Policy:**

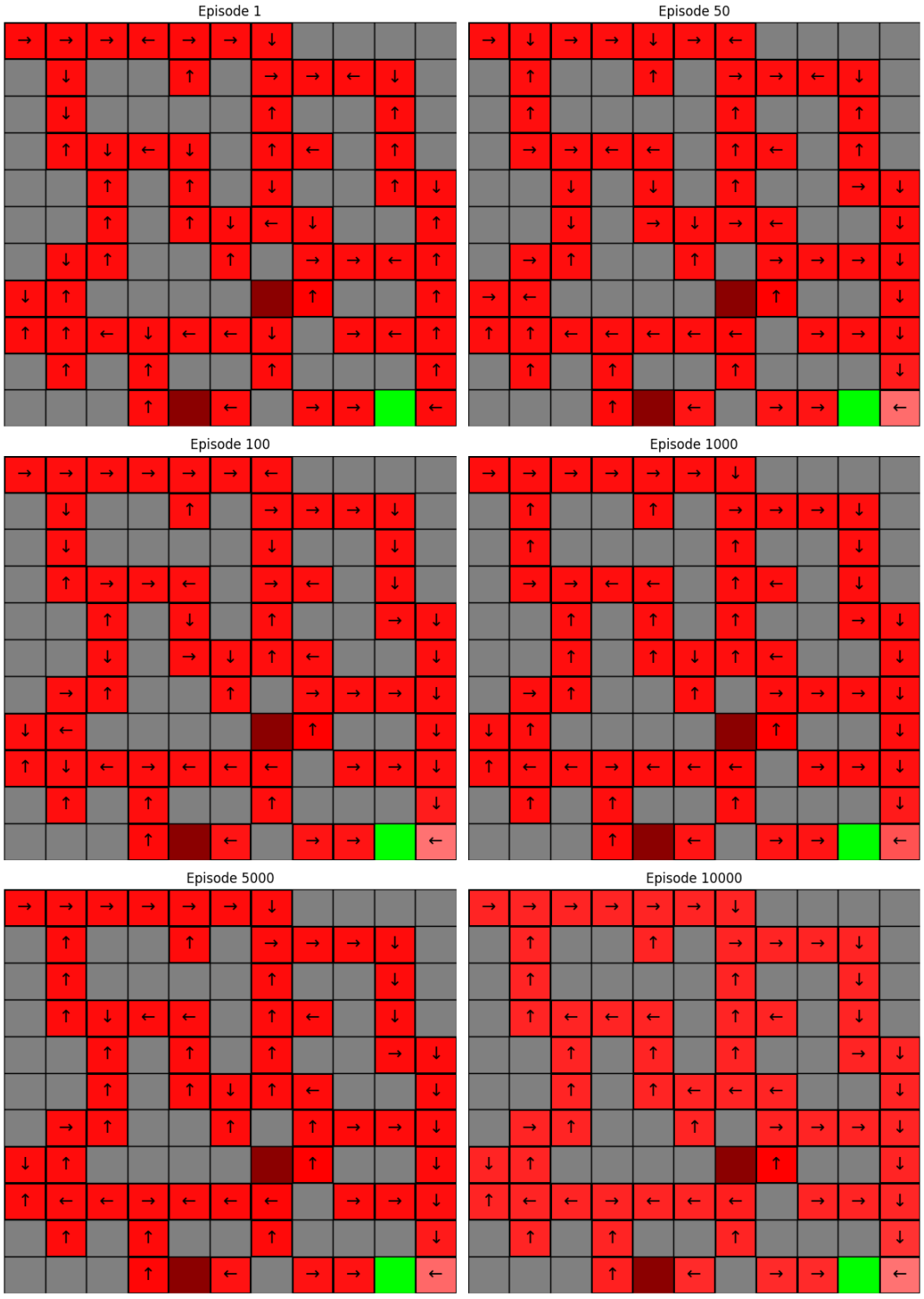

Figure 38.gamma 0.25 policy plot

- **Convergence:**

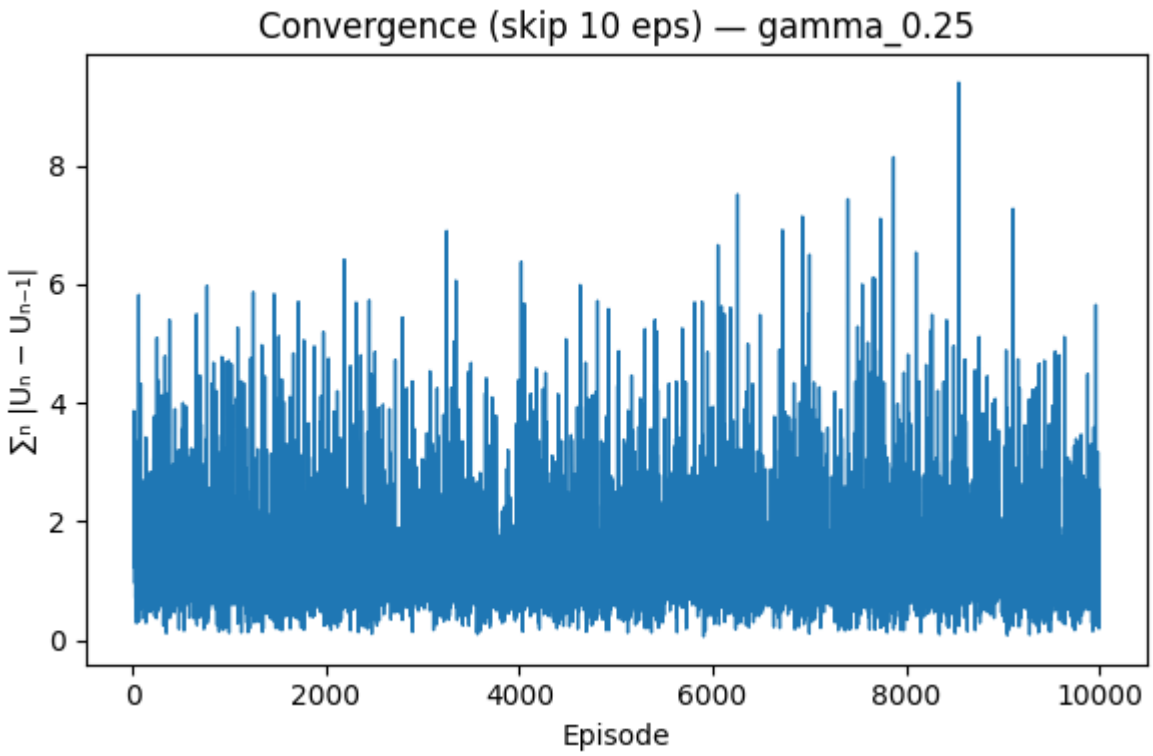

Figure 39.gamma 0.25 convergence plot

XIV. gamma 0.75 plots

- **Utility value function heatmap:**

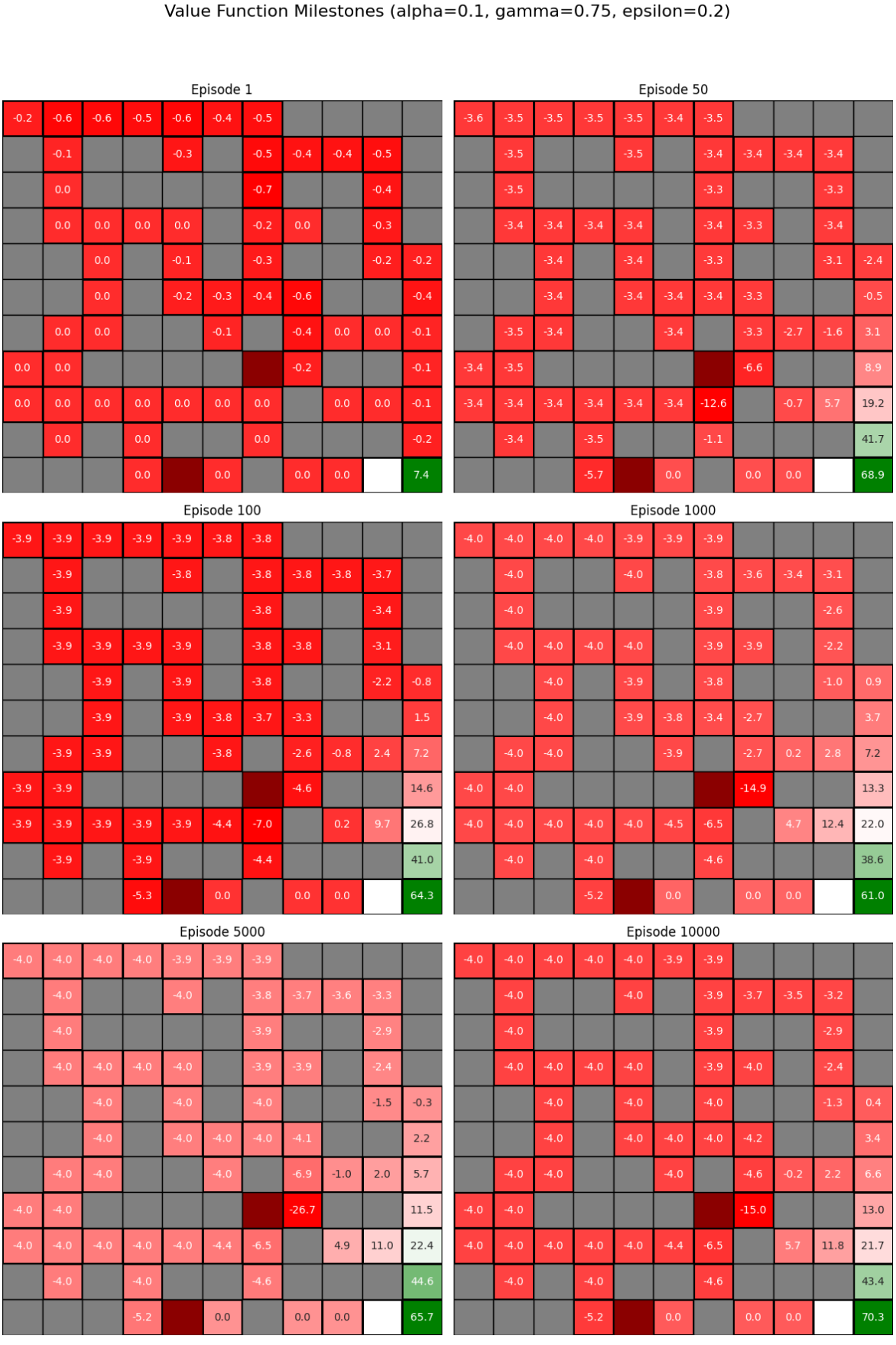

Figure 40.gamma 0.75 value plot

- **Policy:**

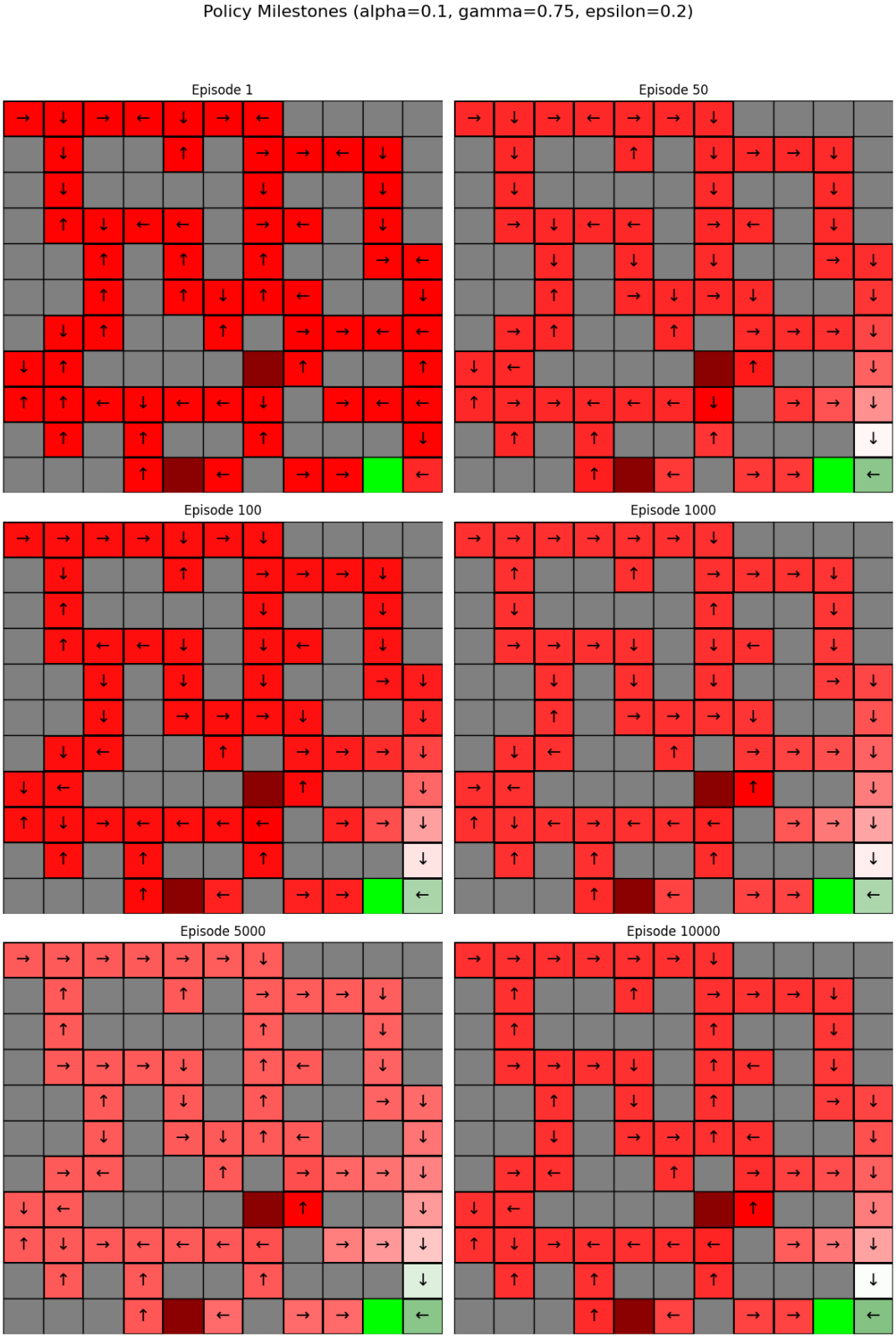

Figure 41.gamma 0.75 policy plot

- **Convergence:**

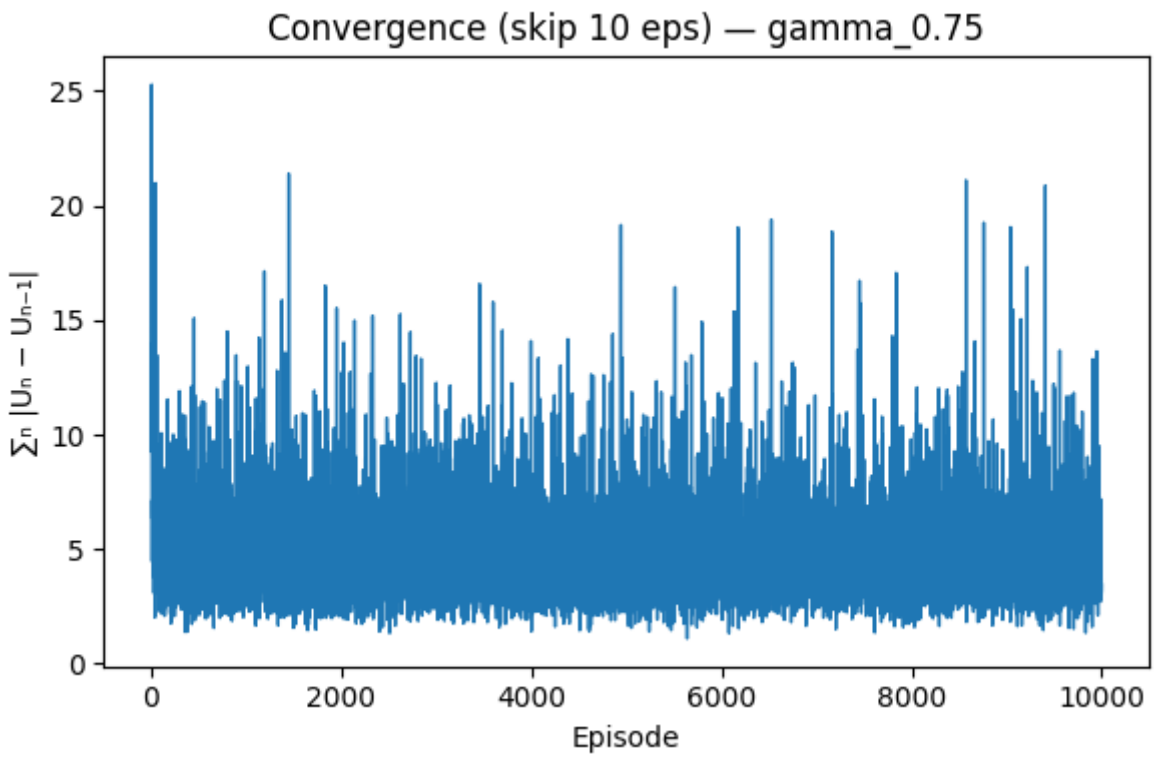

Figure 42.gamma 0.75 convergence plot

XV. gamma 0.95 plots

- **Utility value function heatmap:**

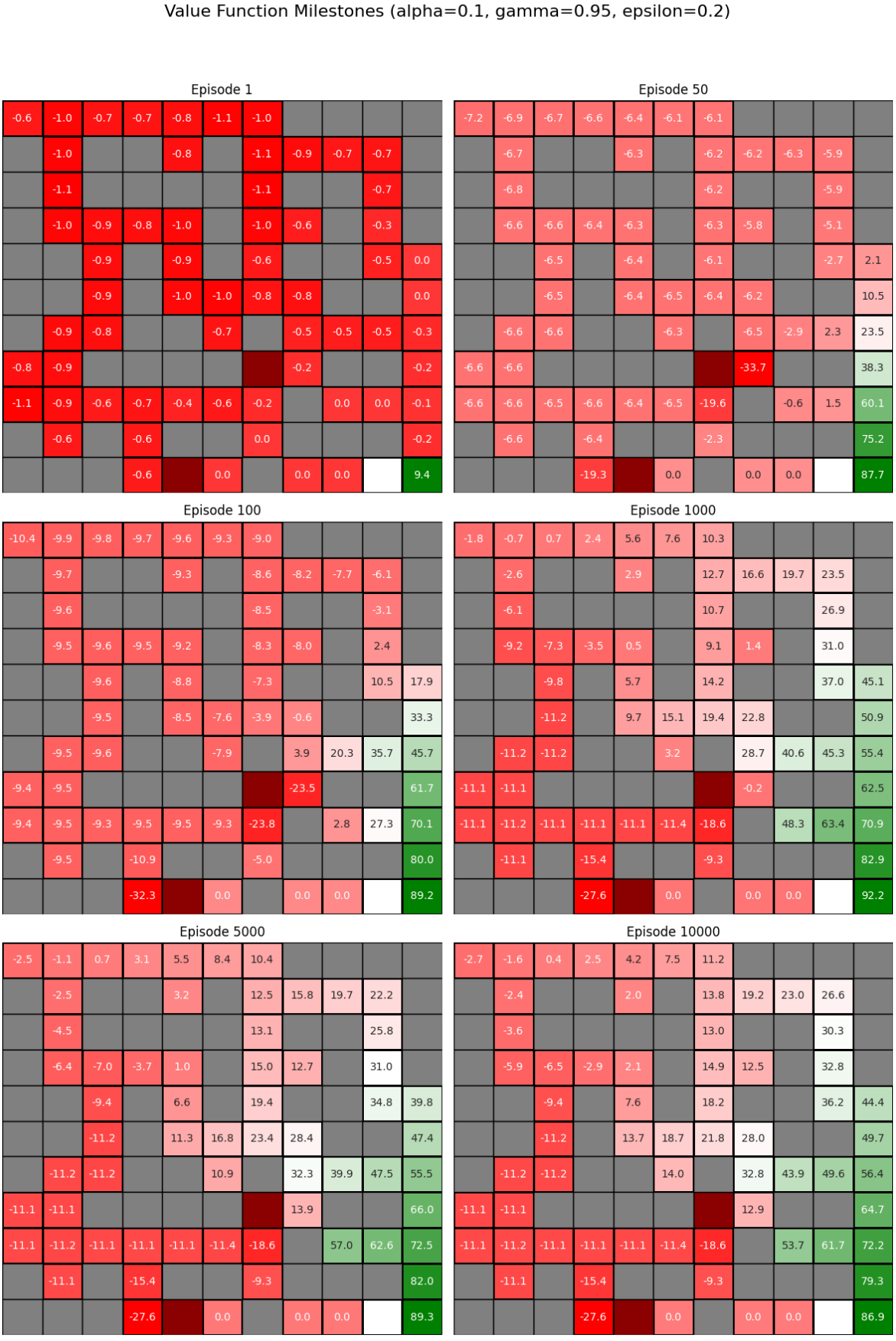

Figure 43.gamma 0.95 value plot

- **Policy:**

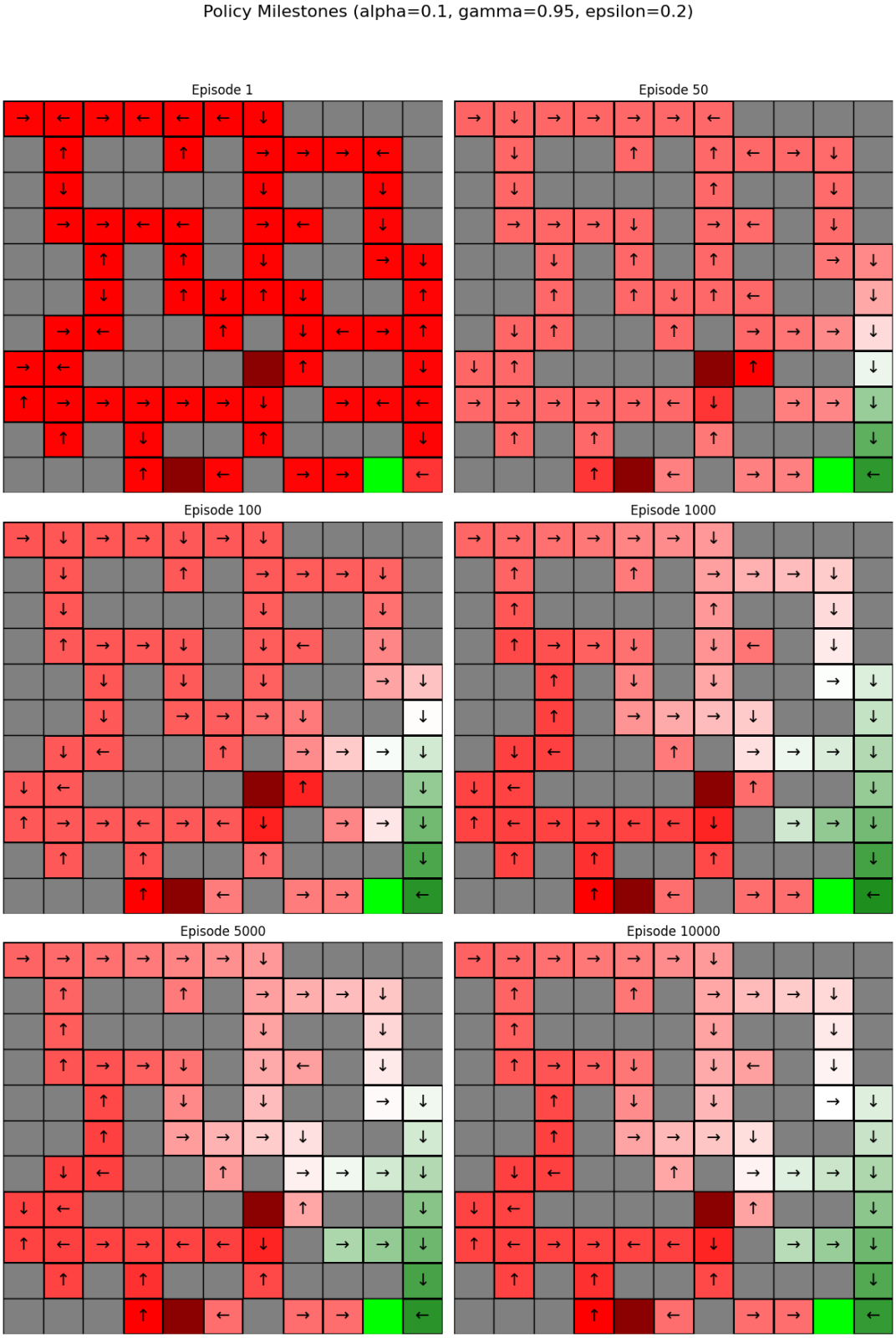

Figure 44.gamma 0.95 policy plot

- **Convergence:**

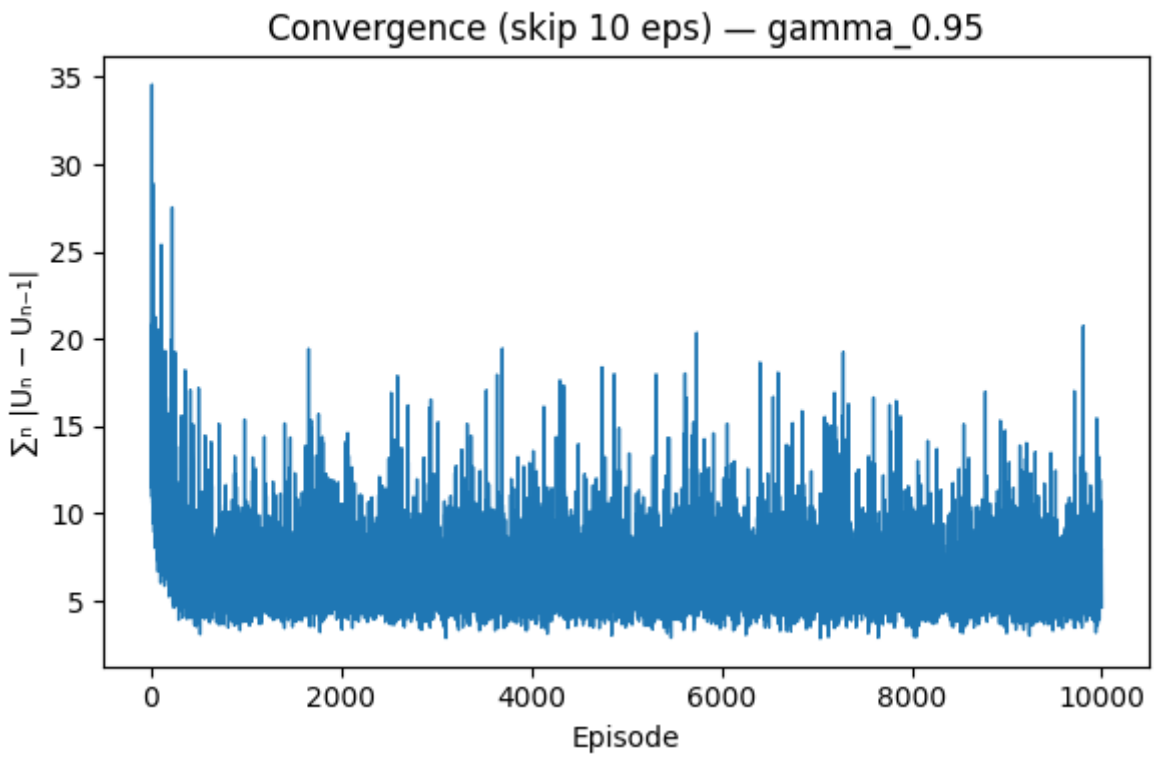

Figure 45.gamma 0.95 convergence plot

1.4

1.) I chose to have the agent move in its chosen direction with relatively high probability (for example, 0.75) but also to "slip" into one of the side or backward directions with small probabilities (for example, 0.10 and 0.05). This stochasticity reflects the uncertainty you often encounter in real environments. By doing so, the agent cannot merely memorize a single path; it must learn a policy that is robust to occasional unintended moves. Similarly, the reward function penalizes every step slightly (for instance, −1 per transition) to discourage needless moves and to drive the agent toward shorter routes, while offering a large positive reward (for instance, +100) upon reaching the goal to strongly reinforce successful completion. On the other hand, imposing a large negative penalty (for instance, −100) for landing in a trap state teaches the agent to avoid dangerous or highly suboptimal regions. To sum up these design choices shape behaviour so that the agent seeks to minimize time spent, learns to avoid from traps, and develops a policy capable of handling the occasional stochastic deviation from its intended action.

2.) At episode 1 the heatmap is almost uniform: nearly every free cell shows a value close to zero (or the small negative from the step cost), meaning the agent has no useful information yet. By episode 50 you start to see the first ripples of learning: cells adjacent

to the goal brighten slightly (less negative) and those near traps deepen (more negative), but most of the maze remains a flat red. Around episode 100 a clearer gradient emerges, with a band of relatively high values tracing out the shortest routes toward the goal and low valleys are observable near the trap walls. By episode 1000 the main corridors stand out sharply—the good states leading from start to finish is now well defined, and regions far from any reward are uniformly low. By episode 5000 the map has largely smoothed into its final shape: values have propagated almost fully backward from the goal, creating a steady utility landscape that rewards moving along the optimal path and punishes circles or traps. At episode 10000 the heatmap is essentially stable, with only tiny adjustments occurring; at this point the value function has effectively converged and closely matches the true expected return for each state under the learned policy.

3.) By tracking the sum of absolute differences between successive value estimates, we observed a rapid decline in update magnitude over the first few hundred episodes and a flattening of the convergence curve thereafter. In our default run ($\alpha = 0.1$, $\gamma = 0.95$, $\varepsilon = 0.2$), the total change dropped very low around episode 3500 and the heatmap at episode 5000 was essentially identical to later snapshots. The convergence plot—showing the sum of updates every ten episodes—confirms that after roughly 1000 episodes the curve behaves very close to its global minimum, and beyond that point further learning is not that effective. Overall, the utility function can be said to have effectively converged between 1000 and 3500 episodes under these settings. But this generalization is very faulty since for some extreme cases the curve might not even converges. Similarly for small alpha values it might take around 10000 episodes to converge. Epsilon equals to 0.2 also gets better and better until episode 10000.

4.) The TD(0) process proved quite sensitive to the choice of learning rate α. when α was very small ($\alpha=0.001$), the update magnitudes were tiny and the convergence curve remained a slowly decaying band of low-amplitude noise even after thousands of episodes, so it took on the order of 5000–10000 episodes before the value function stabilized. By contrast, a moderate α (around 0.01–0.1) produced the best trade-off: large initial updates that rapidly drove the curve down in the first few hundred episodes, then a smooth settling into a narrow fluctuation band by roughly episode 1000. when α was pushed high ($\alpha \geq 0.5$), the convergence plots showed huge, persistent spikes—updates hundreds of times larger than for lower α—and the value estimates never settled cleanly. At $\alpha=1.0$ the process was essentially unstable, with the sum of changes remaining wildly variable throughout training.

When $\alpha = 0.001$ the value heatmaps remain almost flat for a very long time: even by episode 100 you see only slight dips (around –0.2 to –0.3) near traps and tiny rises near the goal, and it isn't until well past episode 5000 that a recognizable gradient emerges. Increasing to $\alpha = 0.01$ produces noticeably faster propagation: by episode 100 the cells along likely routes toward the goal have values around –1 to –2, and by episode 1000 the

main corridors show better values. At α = 0.1 the learning is very good—by episode 50 you already see a clear gradient, by episode 100 a smooth path leads from start to goal, and by episode 1000 the values have largely converged into their final shape. Pushing α higher to 0.5 causes overshooting and oscillations: The heatmaps at early milestones exhibit wildly varying positives and negatives (even spikes above 30 or below –70), and the eventual utility landscape remains noisy. At α = 1.0 the updates are so large that many cells take on extreme values (e.g. –96 or +94) and the heatmaps never fully settle into a stable gradient.

The policy milestones mirror these trends. with α = 0.001 the arrows remain effectively random until very late, since the value function offers almost no guidance; even by episode 1000 most actions still point wrong. At α = 0.01 the policy begins to act with sensible moves by episode 100, and by episode 1000 most arrows reliably point along the optimal path. The case α = 0.1 yields the fastest, cleanest convergence: clear, consistent arrows appear by episode 100, and the almost full optimal policy is in place by episode 1000. At α = 1.0 policy is unstable: the learned policy never stabilizes, with arrows swapping erratically throughout all snapshots and failing to form a coherent route.

For very small γ (like 0.1), you barely see any ripples in the convergence plot: updates stay small from the start and the value heatmap hardly spreads reward information away from the goal, so the agent learns only “right next” moves and ignores long-term planning. This phenomenon is still valid for lower gamma values than 0.95. This is observable for value and policy plots. Since the gamma values are very low agent is not planning future in a good way which ends with poor performance. This is also observable from convergence curve. It is not getting better for lower gamma values. However, training is obvious for gamma 0.95 since convergence curves smoothly decreases. Setting γ very high (around 0.95) makes rewards flow far across the maze, giving you richer, smoother gradients in the final heatmap.

For γ = 0.1 the value heatmaps stay almost uniform after the first few episodes. By episode 50 virtually every free cell is about –1.1, and that barely changes through episodes 100, 1000, and even 10000. The agent only cares about the one-step penalty and never builds up any gradient back to the goal. accordingly, the policy snapshots remain largely random or “right-next” throughout: arrows don’t coalesce into a meaningful route even by episode 10000, since the value function never contains long-term guidance.

For γ = 0.25 you get slightly more signal but still a very shallow look-ahead. The value maps at episode 50 are nearly flat around –1.3, and only by episode 100 do you see any cell rise above –1.3 (up to about +1.7 near the goal). even at episode 10000 most cells are stuck around –1.3, with only a handful of “bright” goal-neighbours. The policy shows a

similar lag: at episode 100 arrows are still scattered, and only slowly by episodes 5000–10000 do you catch patches of correct moves near the goal. the agent is still largely short-sighted.

Setting γ = 0.5 behaves very similar. However, for this case you reach good policy for episode 1000.

With γ = 0.75 the agent plans further ahead, so the values evolve more slowly but end up richer. The heatmap at episode 50 shows values around –3.5, at episode 100 about –3.8 everywhere, and only by episode 1000 does a clear gradient appear (–4 at far states up to +7 near the goal). By episode 5000 the corridor values climb into double digits, and the final map at 10000 is a smooth slope. policy wise, you see arrows slowly align: the first faint path emerges around episode 100, most critical cells point correctly by episode 1000, and by episode 5000 the policy is clean, with only very minor tweaks by episode 10000.

Finally, γ = 0.95 yields the deepest look-ahead and the slowest settling. at episode 50 value cells range from –6 to +23, at episode 100 from –10 up to +18, and only by episode 1000 do values settle into a coherent gradient. The curve still shifts out past episode 5000 before smoothing by episode 10000. Policy snapshots reflect this drawn-out process: you see some correct arrows around the goal early on, but the full optimal policy gets better and better between episodes 1000 and 5000, with tiny adjustments continuing through episode 10000. In other words, high γ gives a powerful, far-reaching value function.

5.) Implementing TD(0) taught me that the simplest updates can hide tricky edge cases. For example, when the agent “bumps” into a wall or tries to move off the grid, it needs to remain in the same state and still incur the step penalty—if you don’t catch those invalid moves before the update, your code will either crash with an out-of-bounds error or let the agent exploit the boundary by avoiding penalties altogether. I solved this by explicitly checking the proposed next-state coordinates and, if they fell outside the maze or into a wall, overwriting them with the current state before computing the reward and TD update. Another stumbling block was setting the learning rate: with α too large, the value estimates would swing wildly and never settle, but with α too small, the agent barely learned anything even after thousands of episodes. I finally settled on an α in the 0.1. Tuning exploration was equally important running with a fixed high epsilon kept the agent dithering, while a fixed low epsilon caused it to lock into poor shortcuts. Finally, recognizing true convergence isn’t just a matter of looking at heatmaps, so I tracked the sum of absolute value changes each episode.

6.) With epsilon=0.0 the agent never explores and simply follows its current greedy policy, so the convergence metric drops very quickly and remains at a low, nearly constant level (albeit potentially to a suboptimal policy). With a small exploration rate like

epsilon=0.2 there are moderate early fluctuations and a slightly slower decline, but the algorithm still settles into a stable value-estimate with small oscillations. At epsilon=0.5 the agent explores half the time, which slows convergence further and keeps the deviation measure higher and more erratic over time. At very high rates such as epsilon=0.8 and epsilon=1.0 the updates remain large and noisy even late in training, reflecting almost pure random action selection that prevents the value estimates from stabilizing quickly. In short, increasing epsilon boosts exploration—useful for discovering better policies—but also slows convergence and increases variance in the update magnitudes; reducing epsilon speeds up stabilization but risks getting stuck without adequate exploration.

With ε=0.0 the agent never takes random actions and simply follows its current greedy choice, so after a few tie-break explorations it quickly settles on a single path and its values along that route converge rapidly to high positive numbers while all other state-values remain flat and unvisited. Introducing light exploration (ε≈0.2) injects occasional randomness, which causes slightly more fluctuation and a slower early decline in update magnitudes but still allows the value estimates to form a clear gradient toward the goal by a few hundred episodes and lock in an optimal policy by a few thousand. At ε≈0.5 the agent explores half the time, so convergence of both the values and the policy is much slower and noisier—only by several thousand episodes does the funnel of high values emerge and the arrows overwhelmingly point toward the goal. With very high rates (ε=0.8 or ε=1.0) most steps are random noise, so the learned values stabilize around the expected return of a random walk (deep negatives with only slight positive bumps near the goal) and the greedy policy remains essentially meaningless even after ten thousand episodes. In short, increasing ε improves coverage of the state–action space but at the cost of slower propagation of rewards and higher variance, whereas too little exploration risks never discovering the optimal path—hence a small ε likely 0.2 usually yields the best balance for off-policy learning.

7.) Beyond the constant per-step cost already in place, we can accelerate learning or crank up the difficulty by reshaping the environment in various ways: for example, sprinkle small “checkpoint” bonuses (+1 or so) at key waypoints to smooth out the reward gradient; randomize the start and goal positions each episode so the agent must learn a general navigation strategy rather than memorizing one map; enlarge the grid and pepper it with extra dead-ends, one-way corridors or locked doors (with collectible keys) to explode the state-space; introduce timed or random wall openings/closings to make the layout non-stationary; inject more transition noise so actions aren’t perfectly reliable; even employ a curriculum of procedurally generated mazes that grow in size or branching factor over time. Each tweak either provides richer shaping signals or forces far more exploration, strengthening the learned policy’s robustness.

8.) One simple way is to treat α and ε as schedules rather than fixed constants. For α you might start at 1 and then halve it every few hundred episodes so that your updates are big at first and then get finer as you learn. For ε-greedy you could begin with ε=1.0 (pure exploration) and decay it linearly or exponentially down toward 0.05 so that the agent gradually shifts from exploring random moves to exploiting what it's learned. Sprinkling "checkpoints" into the maze by giving a small bonus (for example +5) whenever the agent passes a key waypoint breaks the task into subgoals and speeds up propagation of useful reward signals. By simply annealing α and ε over time and peppering the environment with checkpoint rewards, you'll often see faster, more reliable convergence without adding heavy-duty algorithms.

2.

2.1

No. Because the state space in LunarLander-v3 is an 8-dimensional continuous vector, there are effectively infinitely many distinct states, so a lookup table for values or Q-values would either require infinite memory or force a very coarse discretization that destroys the problem's dynamics. Tabular methods rely on visiting each discrete state enough times to estimate its value, but here you would never see the same exact state twice. Instead, you need a function approximator—such as a neural network in a deep Q-network or a linear basis with tile coding—that can generalize value estimates across similar continuous states. This lets the agent learn from limited samples and handle the continuous, high-dimensional nature of the lunar lander task.

2.2

The environment and the agent are ready for the learning.

2.3

- **Different alpha values:**

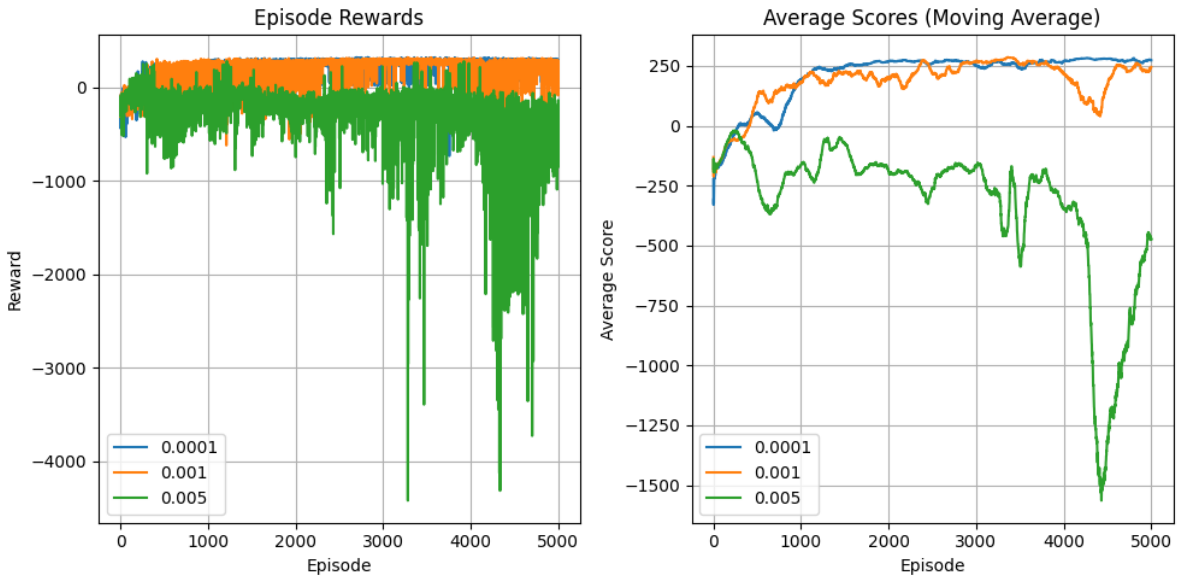

Figure 46. Experiment 2 result for different alpha values

- **Different gamma values:**

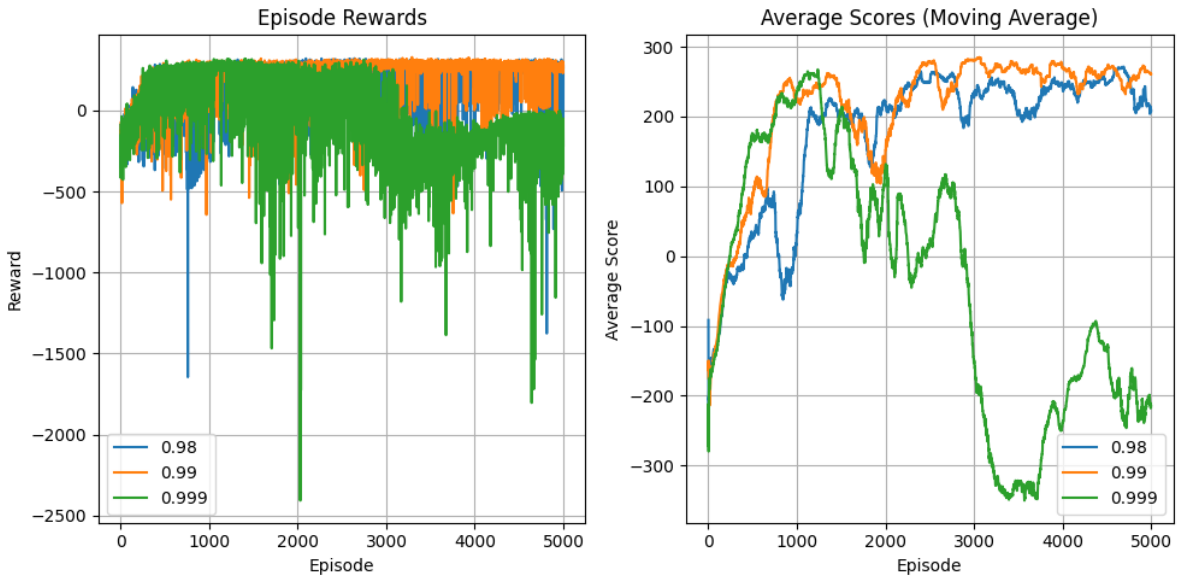

Figure 47.Experiment 2 result for different gamma values

- **Different epsilon values:**

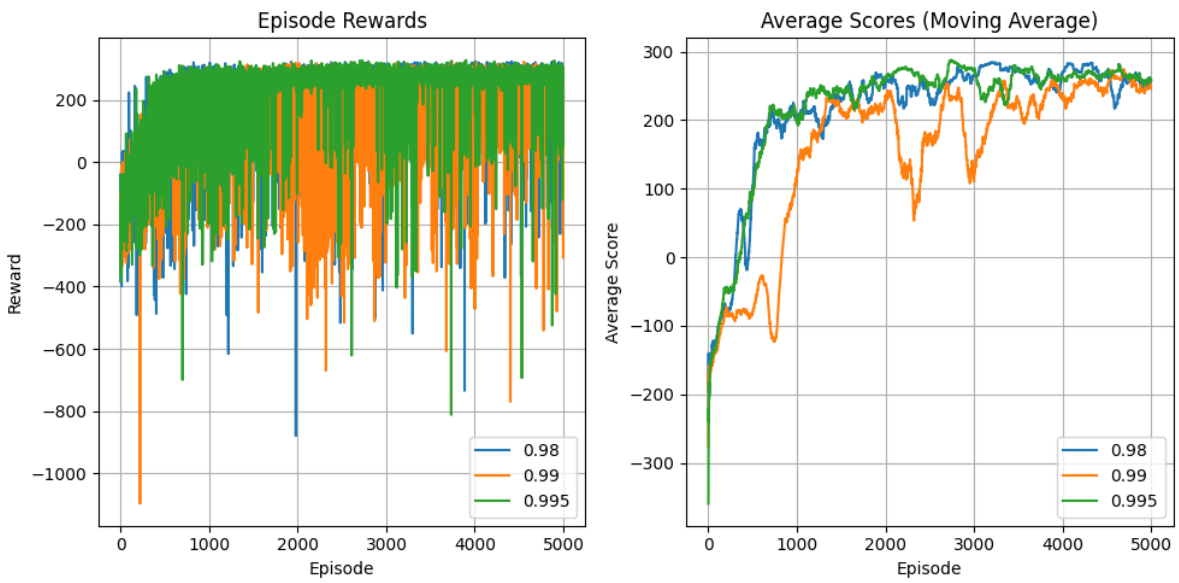

Figure 48. Experiment 2 result for different epsilon values

- **Different f values:**

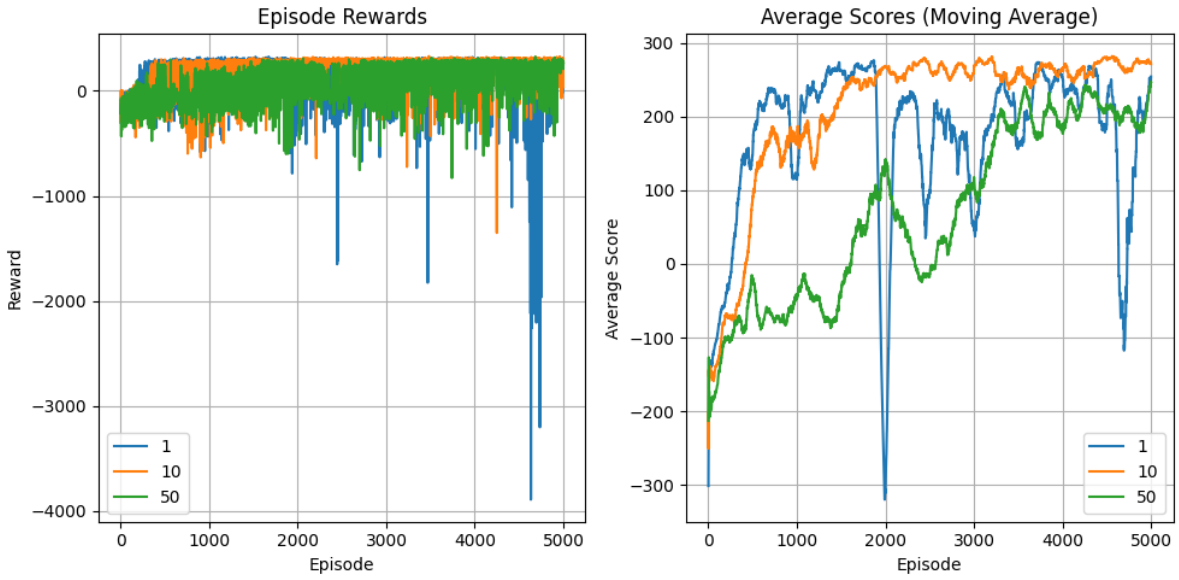

Figure 49. Experiment 2 result for different f values

- **Different hidden dimensions:**

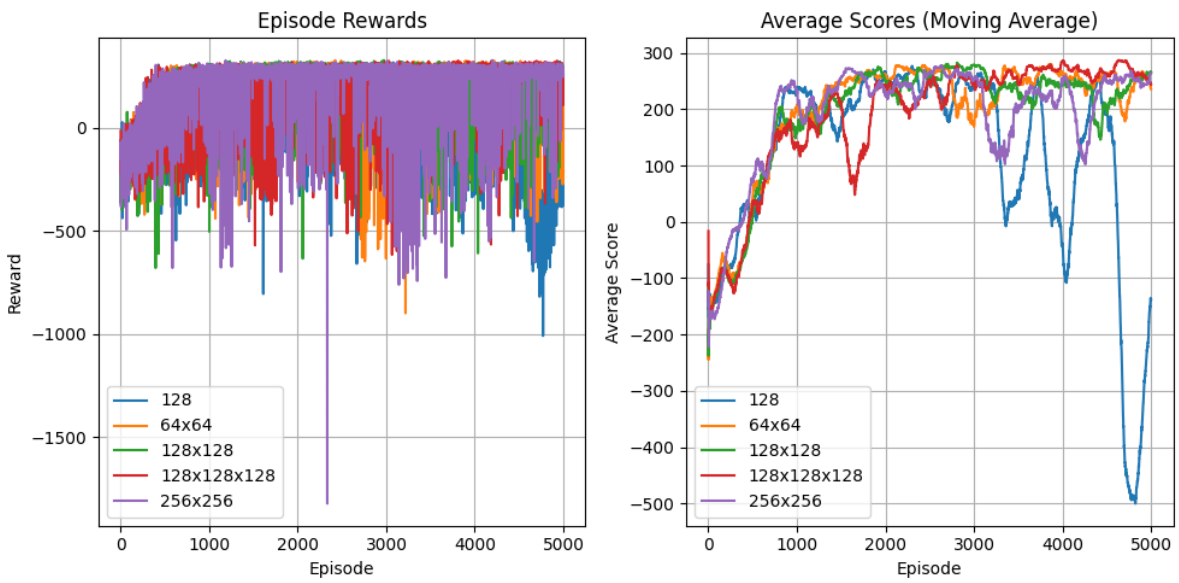

Figure 50.Experiment 2 result for different hidden dimensions

- **Solved episodes:**

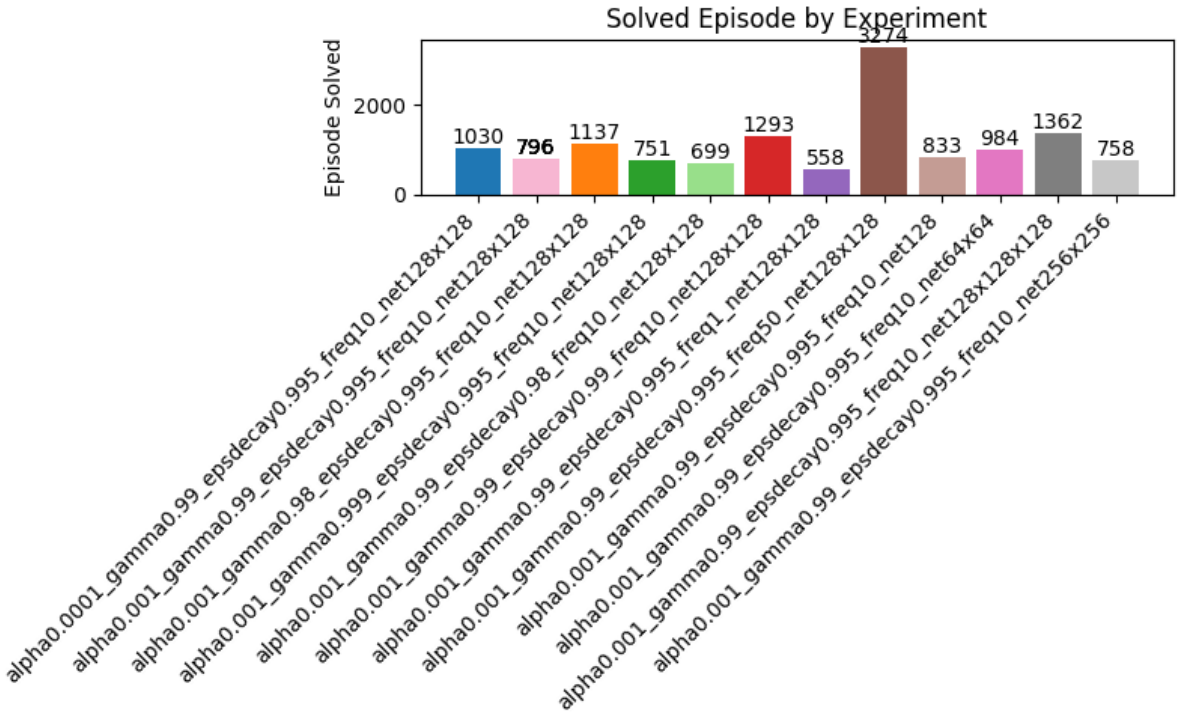

Figure 51. Solved episodes

2.4

1.) For the smallest rate (0.0001) the lander's average score climbs very gradually and almost monotonically, showing virtually no wild swings but requiring many episodes to approach its plateau. The medium rate (0.001) learns much faster – it reaches high average scores within a few hundred to a thousand episodes – and then settles into a relatively stable band, making it the best overall in terms of final performance and speed. The highest rate (0.005) initially jumps up quickly, but the moving average then oscillates wildly and even collapses into deep negative territory, indicating severe overshooting and instability. In short, 0.0001 trades speed for stability, 0.005 trades volatility for speed (and ultimately poor results), while 0.001 hits the sweet spot of reasonably fast convergence with good, stable final scores.

2.) With a lower gamma like 0.98 the agent places more weight on immediate step penalties and rewards, so it learns a stable but somewhat myopic policy: you see its moving average climb steadily past the solved threshold within a few hundred episodes, but it plateaus at a slightly lower final score and still experiences occasional dips. At the other extreme, gamma = 0.999 heavily emphasises far-future returns, which leads to high variance in the updates: initial progress looks promising but then the moving average collapses into deep negatives and never stabilises. The middle ground, gamma = 0.99, proved best: it discounts the distant future just enough to propagate the landing bonus back through the state-space without blowing up variance, so it reaches an average score above 200 in the fewest episodes and maintains the highest, most reliable final performance.

3.) When ε is decayed very quickly (for example multiplied by 0.98 each episode), the agent almost stops exploring after a few hundred episodes, so it can exploit its early findings and climb above the solved-threshold in under 500 episodes. That fast decay yields rapid gains but risks locking into a suboptimal policy if the early exploration was unlucky. With a moderate decay (ε×=0.99) the agent explores longer, which smooths out its learning curve but also delays its first success and leads to more score dips as it keeps trying random moves late in training. The slowest decay (ε×=0.995) maintains exploration the longest, helping the agent avoid early traps and leading to a steadier rise and a very

robust final plateau—even it did not cost too many iterations. To conclude, it is better to not be in rush to use eager policies.

4.) When the target network is updated every episode (frequency = 1), you get the quickest early rise in average score but also the biggest sudden collapses—after climbing above 250 it plummets below –300 around episode 2000 and again later, showing severe oscillation. Updating every 10 episodes strikes a sweet spot: the agent still reaches high scores within a few hundred episodes and then stays in a tight band with only minor dips, combining fast convergence with stability. With very infrequent updates (every 50 episodes), learning is noticeably slower—the moving average doesn't get close to 200 until episode 3000—but once it does the curve is almost bump-free, since the averaged targets act as a strong regularizer. In essence, even more frequent changes are fast, they amplify variance in the bootstrapped Q-values (leading to oscillations), while less frequent updates smooth out that variance at the cost of slower information propagation.

5.) Among the tested architectures, the 64×64 and 128×128 networks delivered the best trade-off between capacity and trainability: both crossed the +200 moving-average threshold by roughly 800–1000 episodes and then stayed in a tight performance band. The single-layer 128 net showed an even quicker early rise but later suffered dramatic collapses, suggesting that its limited depth couldn't fully capture the environment's dynamics and led to unstable value estimates. The deeper models (128×128×128 and especially 256×256) had enough parameters to represent complex functions but exhibited higher variance in their updates, causing occasional large score dips and somewhat slower overall convergence. In short, a moderate-sized network hit the solve criteria fastest and held its gains most reliably, while both the smallest and the largest architectures struggled—one through under-capacity and the other through over-capacity and increased learning complexity.

6.) Across all our experiments, changing how often we updated the target network had the biggest impact on solve times. When we updated the target every single episode, the agent reached an average reward ≥200 in about 558 episodes; switching to an update every 10 episodes slowed it down to roughly 833 episodes; and pushing that out to every 50 episodes delayed solving until about 3274 episodes. That's a swing of over 2700 episodes just from the update frequency. By contrast, tweaking α, γ or ε-decay moved solve-times by only a few hundred episodes at most. This makes sense because frequent target updates reduce time it takes to training in the Q-value bootstrap but amplify variance, letting the agent learn quickly but risking oscillations, whereas infrequent updates smooth out the variance at the cost of slower propagation of new information. One could imagine pairing a fast update schedule (freq=1) with a modest learning rate ($\alpha \approx 0.001$) to temper variance or conversely using a slower update (freq=50) if you want to crank up α or carry out more aggressive exploration. You might also expect interactions—for example, a high α (say 0.005) combined with a very fast ε-decay could amplify

instability, whereas pairing a moderate α (0.001) with a gentler decay (0.995) gives the agent enough "breadth" of experience before it locks in its policy. Likewise, a big network (256×256) might benefit from a slower decay so it can gather more diverse samples before fitting its extra capacity.

7.) You can clearly see that some curves are much spikier than others. For example, the high learning rate (α=0.005) run on the alpha plot has huge downward swings past –1500, whereas the low-α (0.0001) curve is almost a gentle slope upward. Likewise, with target updates every episode (freq=1) you get sharp collapses around episode 2000, while the very stale targets (freq=50) give a much smoother but slower rise. The biggest networks (128×128×128 and 256×256) also show more erratic dips compared to the moderate 64×64 or 128×128 models, and the extreme discount (γ=0.999) trace bounces all over versus the steadier γ=0.99 line. These differences come down to variance in the bootstrap targets and weight updates: high α or frequent target swaps amplify TD error noise, large nets amplify fitting jitter, and aggressive discounting extends noisy future returns. in contrast, small α, moderate update intervals, balanced γ, and right-sized networks all damp out those spikes and yield smoother training curves.

CODE

1.

HW_2_Q1_1.py

```
import numpy as np
from matplotlib import pyplot as plt

class MazeEnvironment:
  def __init__(self):
    # Start position
    self.start_pos = (0, 0)
    self.current_pos = self.start_pos

    # Rewards
    self.state_penalty = -1
    self.trap_penalty  = -100
    self.goal_reward   = 100

    # Action deltas: up, down, left, right
    self.actions = {
      0: (-1,  0),   # up
      1: ( 1,  0),   # down
```

```
        2: ( 0, -1),   # left
        3: ( 0,  1)    # right
    }

    # For stochastic transitions
    self._opposite = {0:1, 1:0, 2:3, 3:2}
    self._perps    = {0:[2,3], 1:[2,3], 2:[0,1], 3:[0,1]}

    # Placeholder for the maze layout; set in subclass or afterwards
    self.maze = None

def reset(self):
    """Return the start state and reset the agent."""
    self.current_pos = self.start_pos
    return self.current_pos

def step(self, action):
    """
    Take the integer action (0–3), sample the actual move according to:
     - 0.75 intended direction
     - 0.05 opposite
     - 0.10 each perpendicular
    Bounce off walls/obstacles (stay in place), then return (state, reward, done).
    """
    if self.maze is None:
        raise ValueError("You must set `self.maze` before calling step().")

    # sample stochastic outcome
    p = np.random.random()
    if p < 0.75:
        a = action
    elif p < 0.80:
        a = self._opposite[action]
    else:
        a = np.random.choice(self._perps[action])

    # compute tentative next position
    dr, dc = self.actions[a]
    r, c = self.current_pos
    nr, nc = r + dr, c + dc

    # check bounds and obstacles
    rows, cols = self.maze.shape
    if not (0 <= nr < rows and 0 <= nc < cols) or self.maze[nr, nc] == 1:
        nr, nc = r, c  # bounce: stay in place

    # determine reward and done
    cell = self.maze[r, c]
    if cell == 3:         # goal
```

```
            reward, done = self.goal_reward, True
        elif cell == 2:       # trap
            reward, done = self.trap_penalty, True
        else:                 # normal free cell
            reward, done = self.state_penalty, False

        # update state
        self.current_pos = (nr, nc)
        return self.current_pos, reward, done

# --- now encode Figure 1 layout into a 10×10 array: ---
#   0 = free, 1 = obstacle (grey), 2 = trap (red), 3 = goal (green)

layout = np.array([
    # row 0
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    # row 1
    [1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1],
    # row 2
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1],
    # row 3
    [1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
    # row 4
    [1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0],
    # row 5
    [1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0],
    # row 6
    [1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0],
    # row 7
    [0, 0, 1, 1, 1, 1, 2, 0, 1, 1, 0],
    # row 8
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    # row 9
    [1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0],
    # row 10
    [1, 1, 1, 0, 2, 0, 1, 0, 0, 3, 0],
])
```

HW_2_Q1_2.py

```
import numpy as np
from HW_2_Q1_1 import *

class MazeTD0(MazeEnvironment):
    def __init__(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000):
        super().__init__()
        self.maze = maze
        self.alpha = alpha       # learning rate
```

```
    self.gamma = gamma       # discount factor
    self.epsilon = epsilon   # exploration rate
    self.episodes = episodes
    max_return = 0
    self.utility = np.full(self.maze.shape, max_return, dtype=float)
    self.utility[self.maze == 1] = -np.inf

  def choose_action(self, state):
    """ε-greedy over neighboring utilities, bouncing invalid moves."""
    r, c = state
    # exploration
    if np.random.random() < self.epsilon:
      return np.random.choice(list(self.actions.keys()))
    # exploitation: pick action that leads to highest utility
    best_value = -np.inf
    best_action = None
    rows, cols = self.maze.shape
    for a, (dr, dc) in self.actions.items():
      nr, nc = r + dr, c + dc
      # if off-grid or obstacle, treat as staying put
      if not (0 <= nr < rows and 0 <= nc < cols) or self.maze[nr, nc] == 1:
        val = -np.inf
      else:
        val = self.utility[nr, nc]
      if best_action is None or val > best_value:
        best_value = val
        best_action = a
    if best_action is None:
      best_action = np.random.choice(list(self.actions.keys()))
    return best_action

  def update_utility_value(self, current_state, reward, new_state):
    """Apply the one-step TD(0) update."""
    cr, cc = current_state
    nr, nc = new_state
    current_value = self.utility[cr, cc]
    next_value    = self.utility[nr, nc]
    # TD target
    target = reward + self.gamma * next_value
    # update rule
    self.utility[cr, cc] += self.alpha * (target - current_value)
    self.utility[self.maze == 2] = self.trap_penalty
    self.utility[self.maze == 3] = self.goal_reward

  def run_episodes(self):
    """Run self.episodes of TD(0), return final utility grid."""
    for _ in range(self.episodes):
      state = self.reset()
      done = False
```

```
        while not done:
            action = self.choose_action(state)
            new_state, reward, done = self.step(action)
            self.update_utility_value(state, reward, new_state)
            state = new_state
        return self.utility
# Create an instance of the TD-0 agent on your maze
maze = layout
maze_td0 = MazeTD0(maze,
                   alpha=0.1,
                   gamma=0.95,
                   epsilon=0.2,
                   episodes=10000)

# Run learning and grab the final utility estimates
final_values = maze_td0.run_episodes()
```

HW_2_Q1_3.py

```
# HW_2_Q1_3.py
import os, json
import numpy as np
import matplotlib.pyplot as plt
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Running on device: {device}")

# import your modules from the previous parts
from HW_2_Q1_1 import layout
from HW_2_Q1_2 import MazeTD0
from utils import plot_value_function, plot_policy

# make output directory
os.makedirs("results_td0", exist_ok=True)

# hyperparameter grids
alphas   = [0.001, 0.01, 0.1, 0.5, 1.0]
gammas   = [0.10, 0.25, 0.50, 0.75, 0.95]
epsilons = [0.0, 0.2, 0.5, 0.8, 1.0]

# default (bolded in Table 1)
default = dict(alpha=0.1, gamma=0.95, epsilon=0.2)
milestones = [1, 50, 100, 1000, 5000, 10000]

def run_and_record(name, **params):
    """
    Runs TD(0) with given params, records:
```

```
 - utilities at each milestone
 - convergence diffs every episode
 then saves JSON and produces plots.
"""
print(f"\n--- Experiment: {name} ({params}) ---")
agent = MazeTD0(layout,
            alpha=params['alpha'],
            gamma=params['gamma'],
            epsilon=params['epsilon'],
            episodes=10000)
prev_U = agent.utility.copy()
diffs = []
recorded = {}

for ep in range(1, agent.episodes+1):
  if ep % 1 == 0:
    print(f"  → Completed episode {ep}/{agent.episodes}")
  state = agent.reset()
  done = False
  while not done:
    a = agent.choose_action(state)
    ns, r, done = agent.step(a)
    agent.update_utility_value(state, r, ns)
    state = ns

  # record utility snapshots
  if ep in milestones:
    recorded[ep] = agent.utility.copy()

  # convergence measure
  diff = np.abs(agent.utility - prev_U).sum()
  diffs.append(diff)
  prev_U = agent.utility.copy()

# save JSON
out = {
  "hyperparameters": params,
  "utilities_over_time": {str(k): v.tolist() for k, v in recorded.items()},
  "convergence": diffs
}
fname = f"results_td0/{name}.json"
with open(fname, 'w') as f:
  json.dump(out, f)

save_dir = f"results_td0/{name}_plots"
os.makedirs(save_dir, exist_ok=True)

# 2) Plot & save the 3×2 milestone grids in one shot
print(f"Saving value-function milestones to {save_dir}/{name}_value_milestones.png")
```

```
    plot_value_function(recorded, layout, name, params, save_dir)

    print(f"Saving policy milestones to {save_dir}/{name}_policy_milestones.png")
    plot_policy(recorded, layout, name, params, save_dir)




    # Plot convergence curve
    plt.figure(figsize=(6,4))
    plt.plot(range(1, agent.episodes+1), diffs, label=name)
    plt.title(f"Convergence — {name}")
    plt.xlabel("Episode")
    plt.ylabel("∑ |U_t − U_{t−1}|")
    plt.legend()
    plt.tight_layout()

    # Save the convergence plot
    plt.savefig(f"results_td0/{name}_convergence.png")
    plt.close()

# 1) Sweep α (others at defaults)
for α in alphas:
    run_and_record(f"alpha_{α}", alpha=α,
                   gamma=default['gamma'],
                   epsilon=default['epsilon'])

# 2) Sweep γ (others at defaults)
for γ in gammas:
    run_and_record(f"gamma_{γ}", alpha=default['alpha'],
                   gamma=γ,
                   epsilon=default['epsilon'])

# 3) Sweep ε (others at defaults)
for ε in epsilons:
    run_and_record(f"epsilon_{ε}", alpha=default['alpha'],
                   gamma=default['gamma'],
                   epsilon=ε)
```

HW2_2_Q1_4.py

```
# plot_convergence_curves.py

import os
import glob
import json
import matplotlib.pyplot as plt

# where your JSONs live
INPUT_DIR = r"C:\EE449\EE449_HW_2\results_td0"
```

```
# where to put the new convergence plots
OUTPUT_DIR = "results_convergence_plots"
os.makedirs(OUTPUT_DIR, exist_ok=True)

# how many early episodes to skip in the zoomed plot
SKIP_EPISODES = 10

for json_file in glob.glob(os.path.join(INPUT_DIR, "*.json")):
  name = os.path.splitext(os.path.basename(json_file))[0]
  with open(json_file, "r") as f:
    data = json.load(f)
  diffs = data["convergence"]
  episodes = list(range(1, len(diffs) + 1))

  # optionally drop the first SKIP_EPISODES points
  ep_plot = episodes[SKIP_EPISODES:]
  diffs_plot = diffs[SKIP_EPISODES:]

  plt.figure(figsize=(6, 4))
  plt.plot(ep_plot, diffs_plot, lw=1)
  plt.title(f"Convergence (skip {SKIP_EPISODES} eps) — {name}")
  plt.xlabel("Episode")
  plt.ylabel("∑ₙ |Uₙ − Uₙ₋₁|")
  plt.tight_layout()
  out_path = os.path.join(OUTPUT_DIR, f"{name}_convergence.png")
  plt.savefig(out_path)
  plt.close()
  print(f"Saved {out_path}")
```

HW2_2_Q2_2.py

```
import gymnasium as gym
import numpy as np
import torch
import os
import json
import torch.nn.functional as F
from utils import plot_learning_curves, plot_solved_episodes
import torch.nn as nn
from collections import deque
import random
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Running on device: {device}")


# 1) Q-Network with two hidden layers
class QNetwork(nn.Module):
  def __init__(self, state_dim, action_dim, hidden_dims=(128,128)):
    super().__init__()
    dims = [state_dim] + list(hidden_dims) + [action_dim]
```

```
        self.layers = nn.ModuleList(
            nn.Linear(dims[i], dims[i+1]) for i in range(len(dims)-1)
        )
    def forward(self, x):
        for layer in self.layers[:-1]:
            x = F.relu(layer(x))
        return self.layers[-1](x)


# 2) Replay memory (as given)
class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)


# 3) DQN Agent
class DQNAgent:
    def __init__(self, state_dim, action_dim,
                 memory_size=50000, batch_size=64,
                 hidden_dims=(128, 128),  # ← New parameter
                 gamma=0.99, alpha=1e-3,
                 epsilon_start=1.0, epsilon_min=0.01, epsilon_decay=0.995,
                 target_update_freq=10):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma
        self.hidden_dims = hidden_dims
        self.batch_size = batch_size

        self.epsilon = epsilon_start
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay

        self.memory = ReplayMemory(memory_size)
```

```
        self.policy_net = QNetwork(state_dim, action_dim)
        self.target_net = QNetwork(state_dim, action_dim)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()

        self.optimizer = torch.optim.Adam(self.policy_net.parameters(), lr=alpha)
        self.target_update_freq = target_update_freq

        self.solved_score = 200.0
        self.solved_window = 100
        self.rewards_window = deque(maxlen=self.solved_window)

    def get_action(self, state):
        # ε-greedy action selection
        if random.random() < self.epsilon:
            return random.randrange(self.action_dim)
        state_t = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
        with torch.no_grad():
            qvals = self.policy_net(state_t)
        return int(qvals.argmax(dim=1).item())

    def train_step(self):
        if len(self.memory) < self.batch_size:
            return

        batch = self.memory.sample(self.batch_size)
        states = torch.tensor(np.array([exp[0] for exp in batch]), dtype=torch.float32)
        actions = torch.tensor([exp[1] for exp in batch], dtype=torch.int64).unsqueeze(1)
        rewards = torch.tensor([exp[2] for exp in batch], dtype=torch.float32).unsqueeze(1)
        next_states = torch.tensor(np.array([exp[3] for exp in batch]), dtype=torch.float32)
        dones = torch.tensor([exp[4] for exp in batch], dtype=torch.float32).unsqueeze(1)

        # Current Q-values
        curr_Q = self.policy_net(states).gather(1, actions)
        # Next Q-values from target network
        next_Q = self.target_net(next_states).max(dim=1)[0].unsqueeze(1)
        # TD target: r + γ·maxQ'·(1−done)
        target_Q = rewards + (self.gamma * next_Q * (1 - dones))

        # MSE loss and backprop
        loss = F.mse_loss(curr_Q, target_Q)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def update_target(self):
        # Copy weights
        self.target_net.load_state_dict(self.policy_net.state_dict())
```

```
    def decay_epsilon(self):
        # Decay with floor
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)



# ——— 4) Experiment Runner ————————————————————————————————————————
def run_experiment(params, num_episodes=5000, save_dir="results_dqn"):
    os.makedirs(save_dir, exist_ok=True)
    # unpack hyperparameters
    α  = params["alpha"]
    γ  = params["gamma"]
    εd = params["epsilon_decay"]
    f  = params["target_update_freq"]
    h  = params["hidden_dims"]

    # make a fresh env & agent
    env = gym.make("LunarLander-v3")
    s_dim = env.observation_space.shape[0]
    a_dim = env.action_space.n

    agent = DQNAgent(
        state_dim=s_dim, action_dim=a_dim,
        hidden_dims=h,
        alpha=α, gamma=γ,
        epsilon_start=1.0, epsilon_min=0.01, epsilon_decay=εd,
        target_update_freq=f
    )

    episode_rewards = []
    average_scores = []
    solved_episode = None

    for ep in range(1, num_episodes+1):
        state, _ = env.reset()
        max_steps_per_episode = 1000
        total_reward = 0.0
        done = False

        for t in range(max_steps_per_episode):
            # 1) pick an action
            action = agent.get_action(state)

            # 2) step the env
            next_state, reward, done, _, _ = env.step(action)

            # 3) store the transition
            agent.memory.push(state, action, reward, next_state, float(done))
```

```
        # 4) update the network
        agent.train_step()

        # 5) move to the next state
        state = next_state

        # 6) accumulate reward
        total_reward += reward

        # 7) if the episode is over, stop early
        if done:
            break

    # end episode bookkeeping
    episode_rewards.append(total_reward)
    agent.decay_epsilon()
    if ep % agent.target_update_freq == 0:
        agent.update_target()

    # moving average over the last `solved_window`
    recent_rewards = episode_rewards[-agent.solved_window:]
    moving_avg = np.mean(recent_rewards)
    average_scores.append(moving_avg)
    if solved_episode is None and len(recent_rewards) == agent.solved_window and moving_avg >=
agent.solved_score:
        solved_episode = ep

  # close environment after all episodes
  env.close()

  # save results to JSON
  results = {
    "episode_rewards": episode_rewards,
    "average_scores": average_scores,
    "hyperparameters": params,
    "solved_episode": solved_episode
  }
  # filename uses descriptive parameter keys
  filename = (
      f"{save_dir}/alpha{params['alpha']}_gamma{params['gamma']}_"
      f"epsdecay{params['epsilon_decay']}_freq{params['target_update_freq']}_"
      f"net" + "x".join(str(d) for d in params['hidden_dims']) + ".json"
  )
  with open(filename, 'w') as json_file:
    json.dump(results, json_file)
  return filename

# 1) Defaults (the bold values in your table)
```

```
defaults = {
  "alpha":        1e-3,
  "gamma":        0.99,
  "epsilon_decay":  0.995,
  "target_update_freq": 10,
  "hidden_dims":    (128, 128),
}

# 2) Sweep definitions
sweeps = {
  "alpha":        [1e-4, 1e-3, 5e-3],
  "gamma":        [0.98,  0.99, 0.999],
  "epsilon_decay":  [0.98,  0.99, 0.995],
  "target_update_freq": [1, 10, 50],
  # for the "Net*" experiments, replace these tuples with your actual layer sizes:
  "hidden_dims": [
    (128,),
    (64, 64),
    (128, 128),
    (128, 128, 128),
    (256, 256),
  ],
}

all_jsons = []

for param, values in sweeps.items():
  json_paths = []
  labels    = []
  for v in values:
    # build this run's hyperparameter dict
    params = defaults.copy()
    params[param] = v

    print(f">>> Running sweep {param} = {v}")
    fname = run_experiment(params,
                num_episodes=5000,
                save_dir="results_dqn")
    json_paths.append(fname)
    all_jsons.append(fname)

    # pretty label
    if isinstance(v, tuple):
      labels.append("x".join(map(str, v)))
    else:
      labels.append(str(v))

  # 3) Plot learning curves for this group
  out_curve = f"results_dqn/{param}_learning_curves.png"
```

```
    plot_learning_curves(json_paths,
                         labels=labels,
                         output_file=out_curve)

# 4) One bar chart of solved episodes across *all* experiments
out_solved = "results_dqn/solved_episodes.png"
plot_solved_episodes(all_jsons,
                     labels=None,
                     output_file=out_solved)

print("\nDone.  Look in results_dqn/ for JSONs and plots.")
```