EE449 – HW2
Çetincan Önelge - 2375426

EE449 HW2
Reinforcement Learning

1: Temporal Difference (TD) Learning

1.3: Experimental Work

For the experimental work, as stated in the question, I have done 5 runs for each parameter. While changing the single parameter values, I have kept the other two parameter values at defaults. For each run, I have generated 1 convergence plot, 1 utility value function heat map that includes all 6 different episode runs and 1 policy map that includes all 6 different episode runs. So, in this section we will have 3 different figures for each run and 45 figures in total.

Alpha Runs (α=0.001, 0.01, 0.1, 0.5, 1.0 ; γ=0.95 ; ε=0.2)

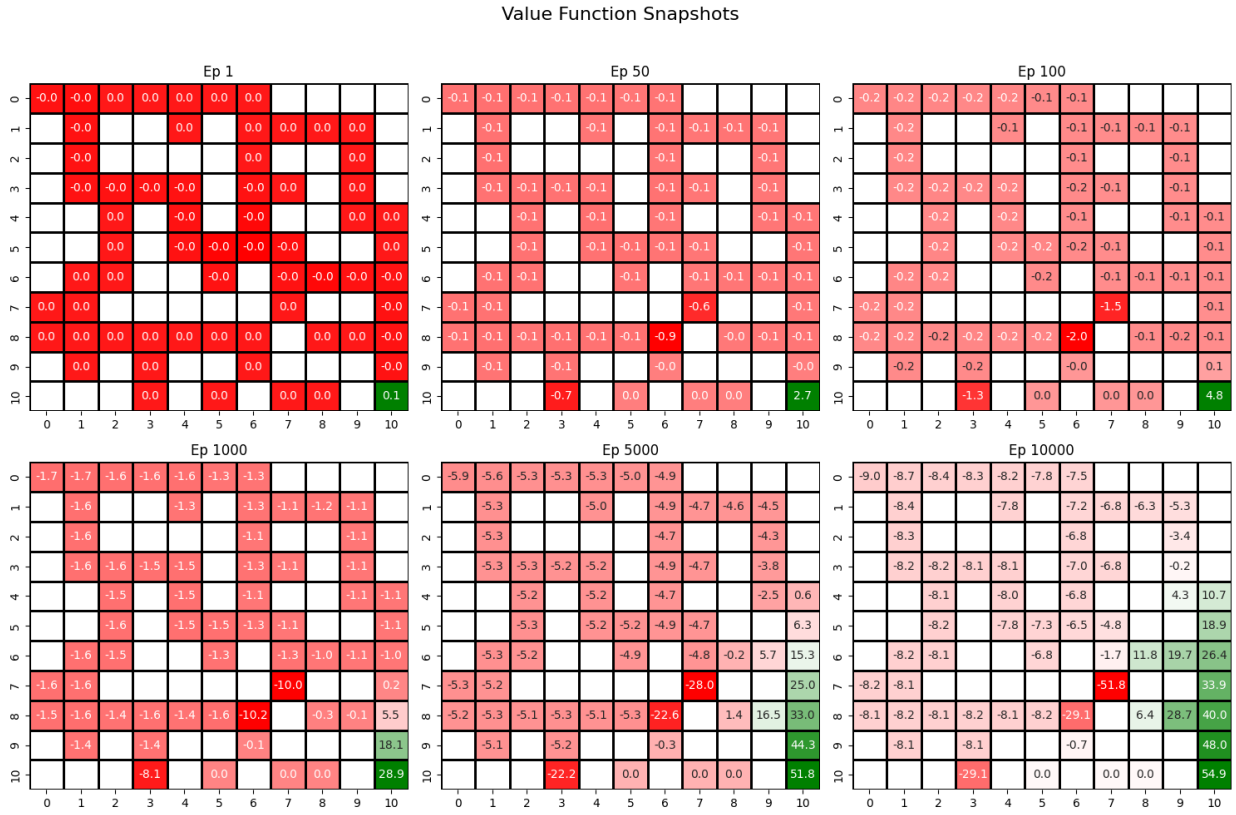

Figure 1: Utility Value Function Heatmap of Parameters (α=0.001, γ=0.95, ε=0.2)

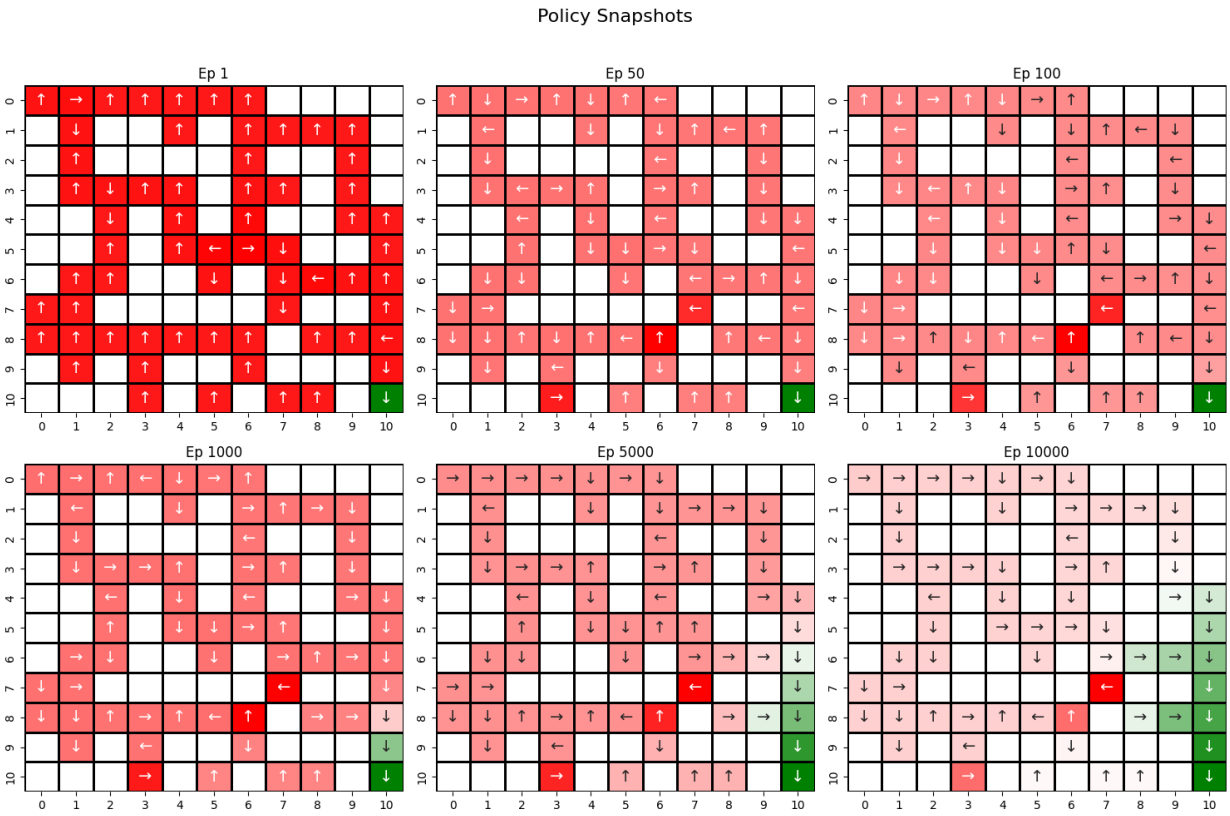

Figure 2: Policy Map of Parameters (α=0.001, γ=0.95, ε=0.2)

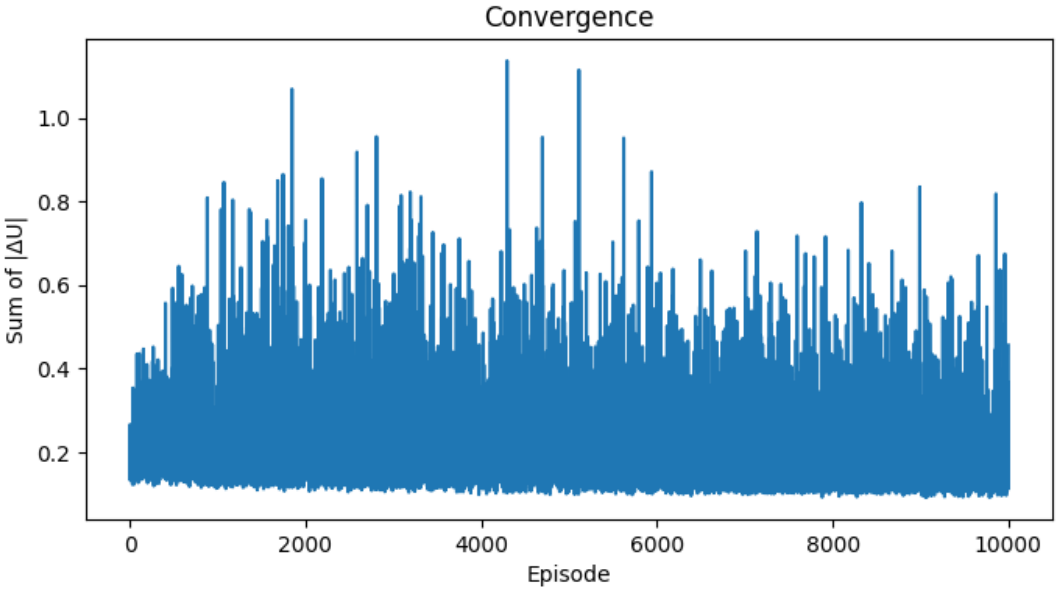

Figure 3: Convergence Plot of Parameters (α=0.001, γ=0.95, ε=0.2)

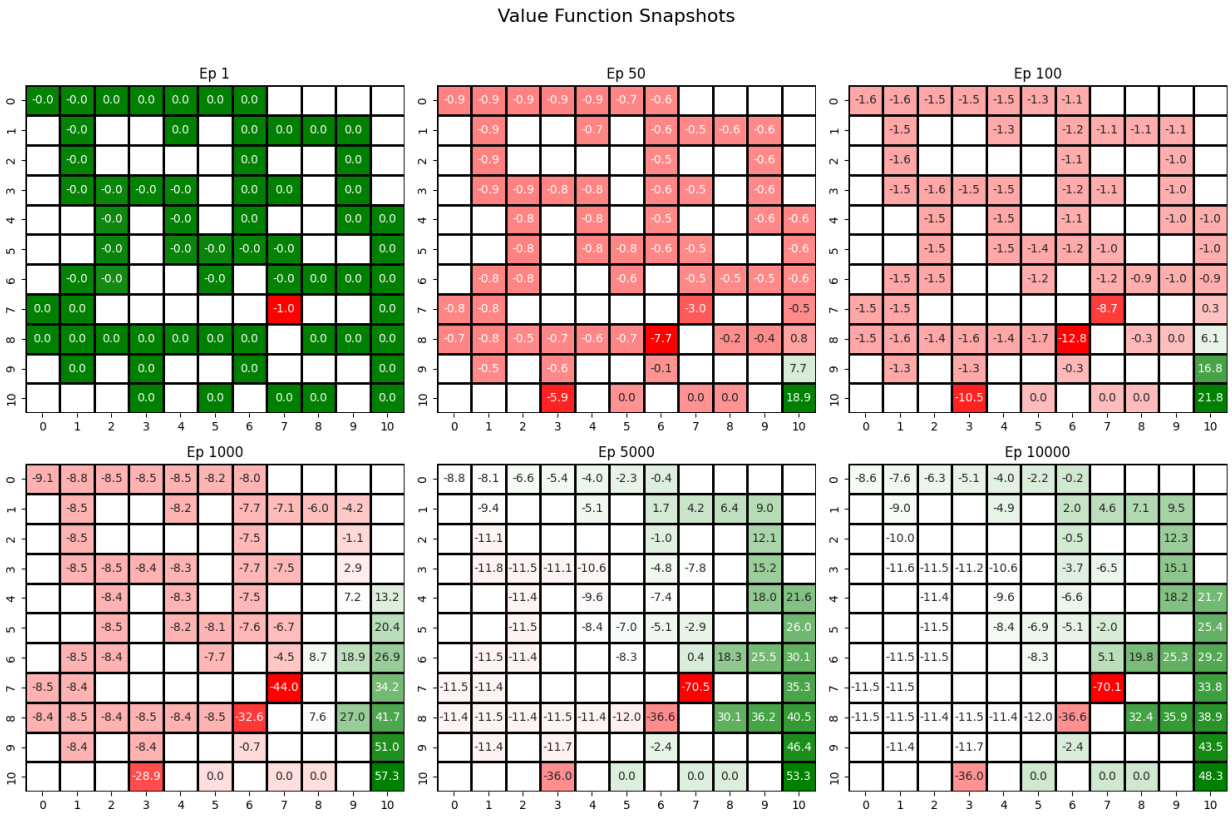

Figure 4: Utility Value Function Heatmap of Parameters (α=0.01, γ=0.95, ε=0.2)

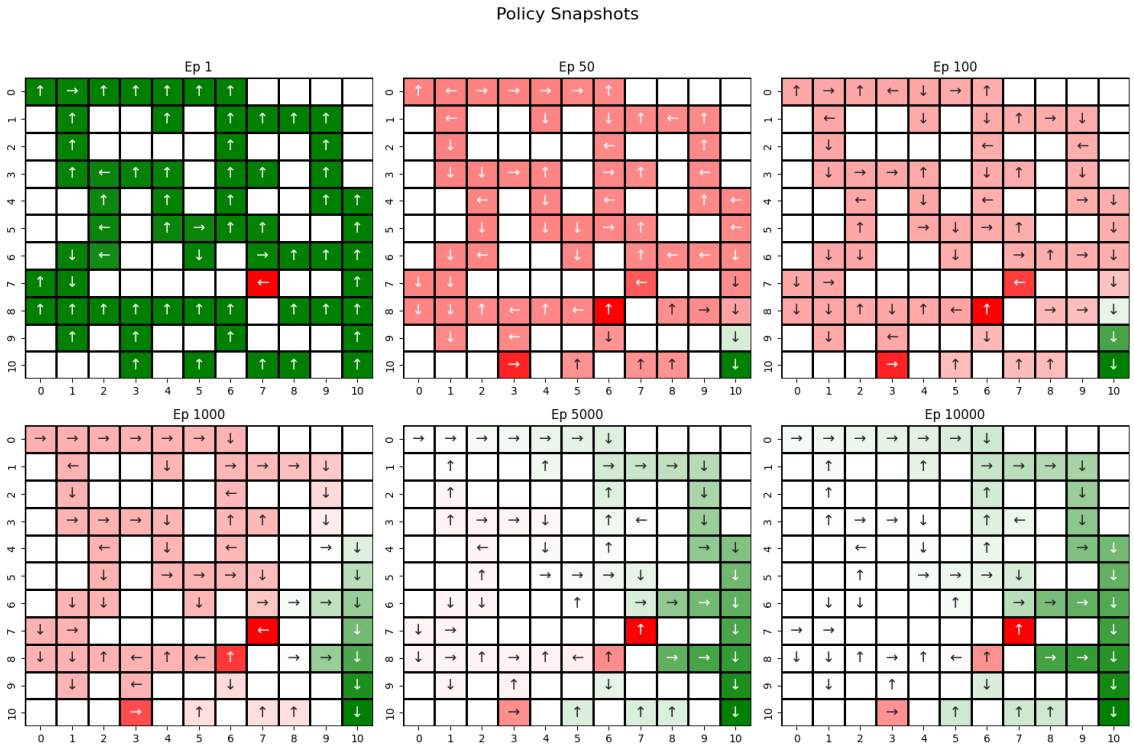

Figure 5: Policy Map of Parameters (α=0.01, γ=0.95, ε=0.2)

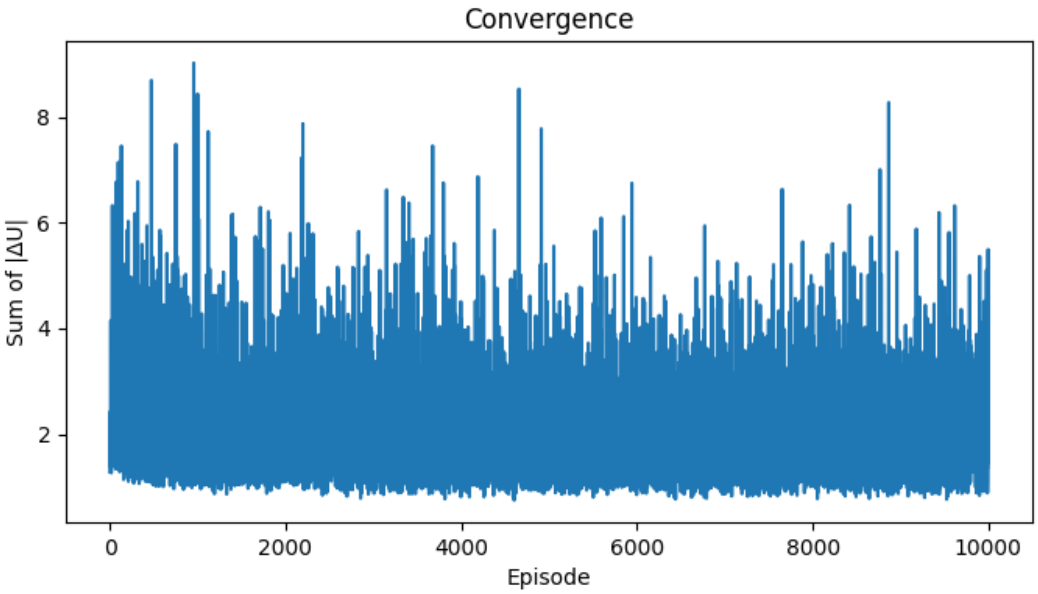

Figure 6: Convergence Plot of Parameters (α=0.01, γ=0.95, ε=0.2)

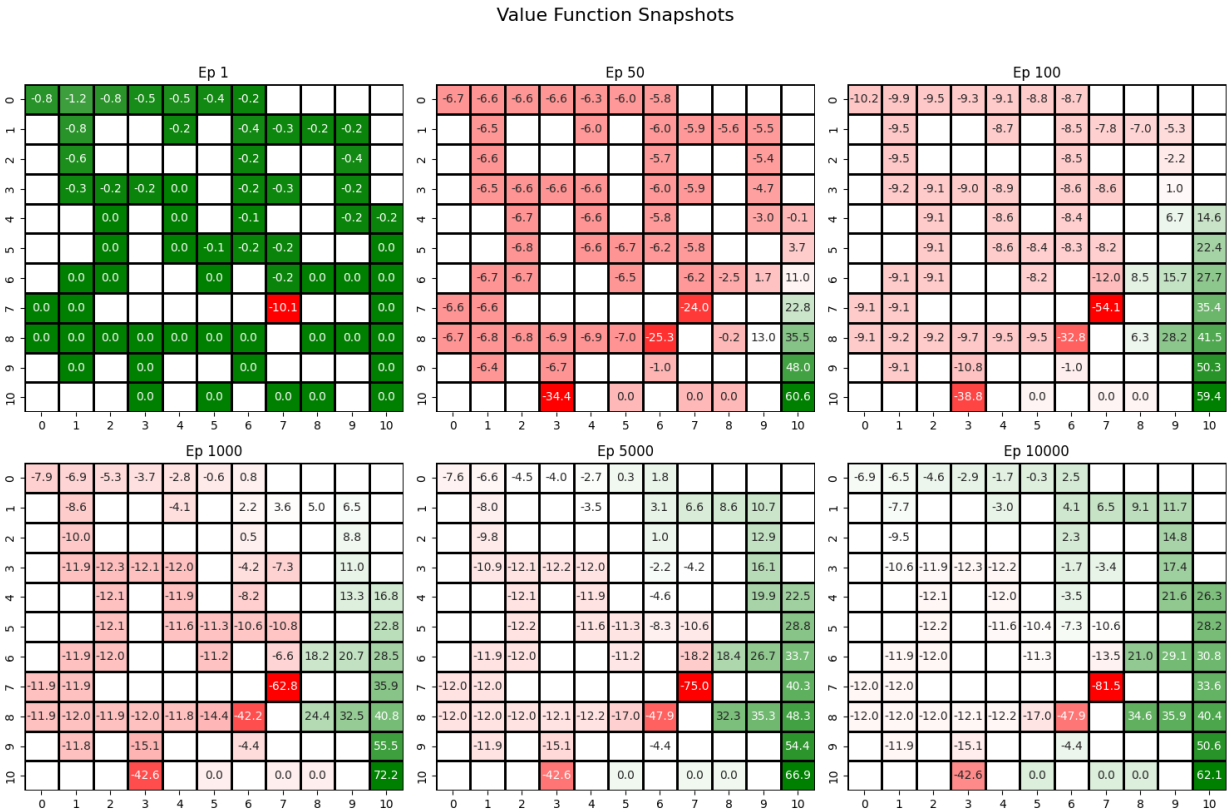

Figure 7: Utility Value Function Heatmap of Parameters (α=0.1, γ=0.95, ε=0.2)

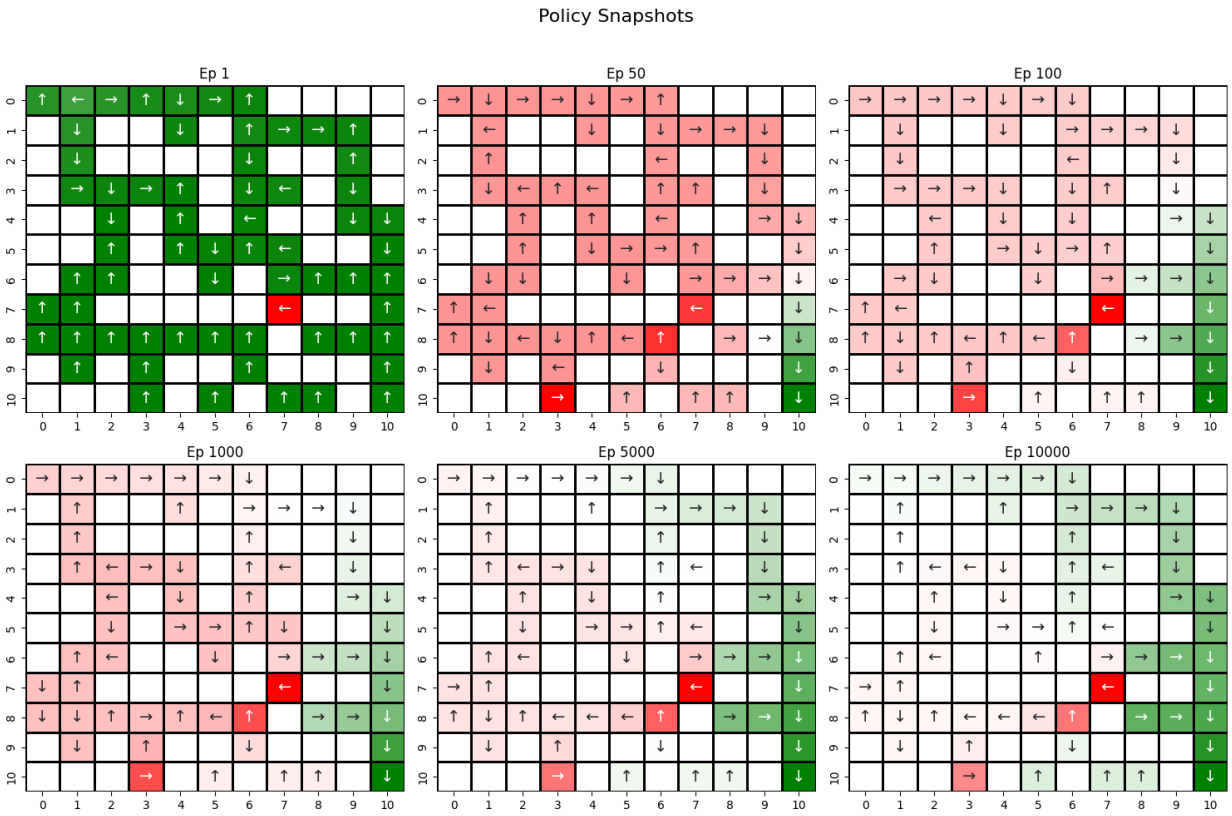

Figure 8: Policy Map of Parameters (α=0.1, γ=0.95, ε=0.2)

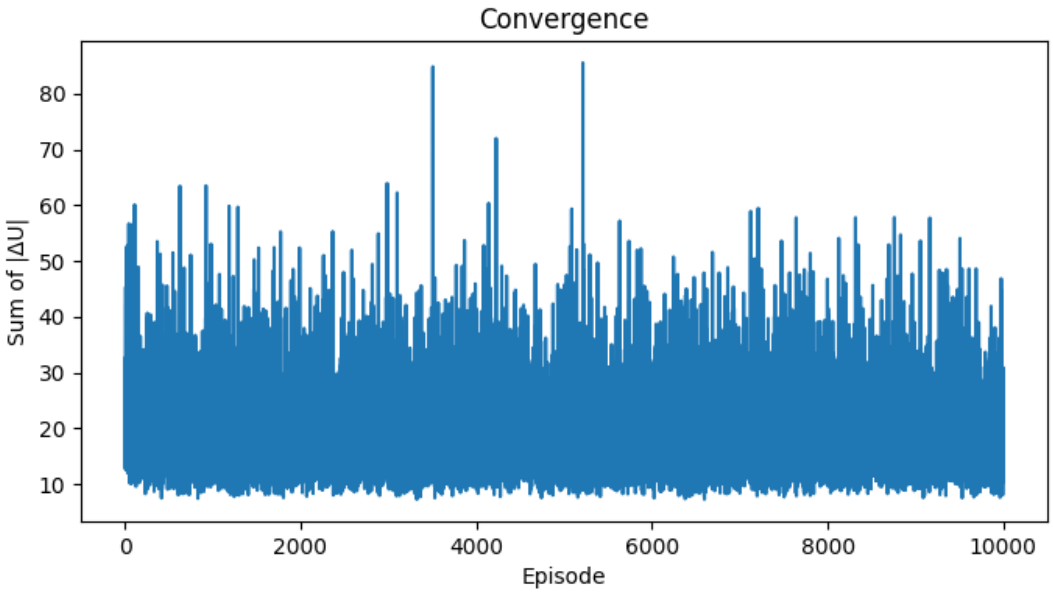

Figure 9: Convergence Map of Parameters (α=0.1, γ=0.95, ε=0.2)

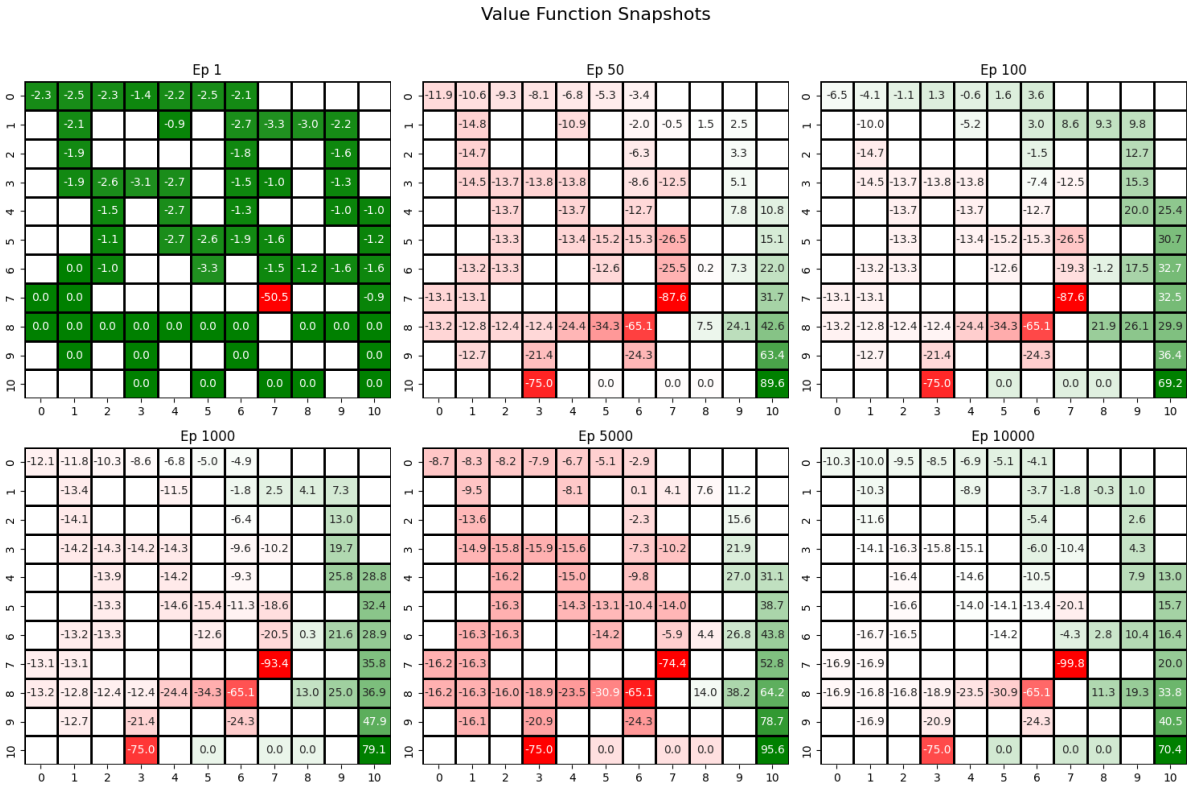

Figure 10: Utility Value Function Heatmap of Parameters (α=0.5, γ=0.95, ε=0.2)

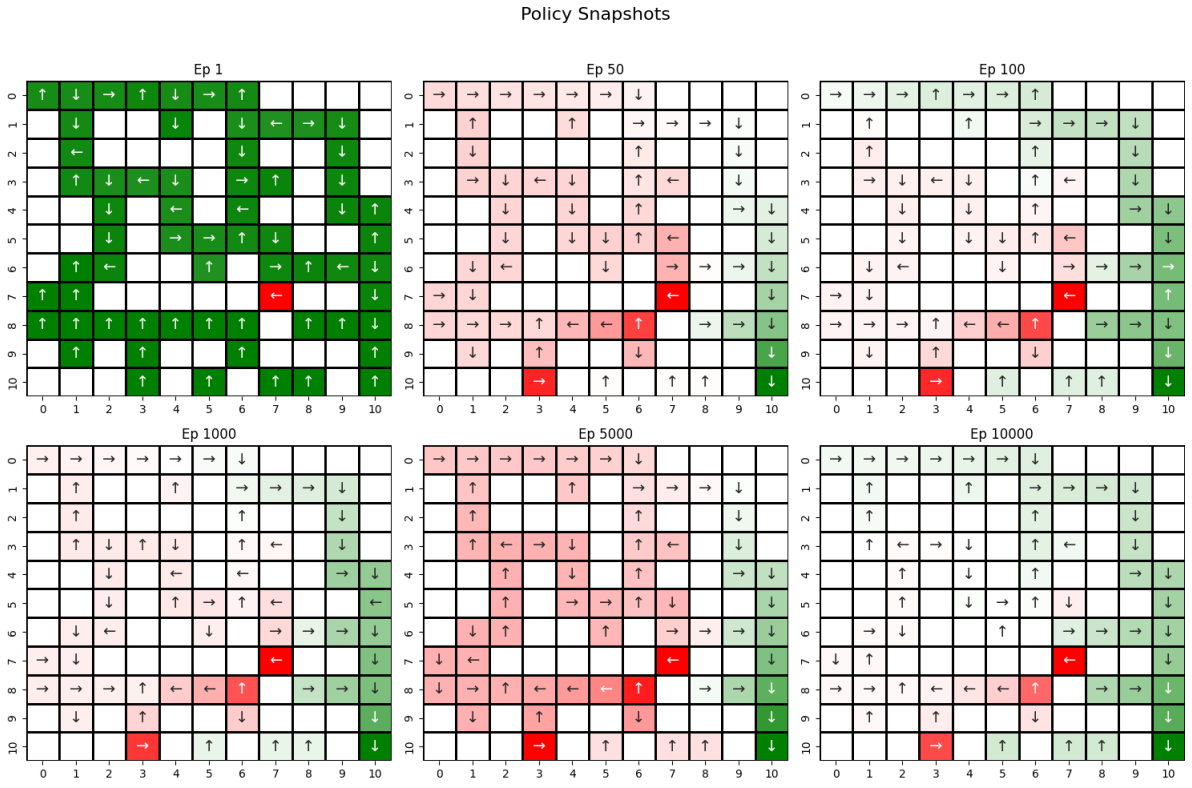

Figure 11: Policy Map of Parameters (α=0.5, γ=0.95, ε=0.2)

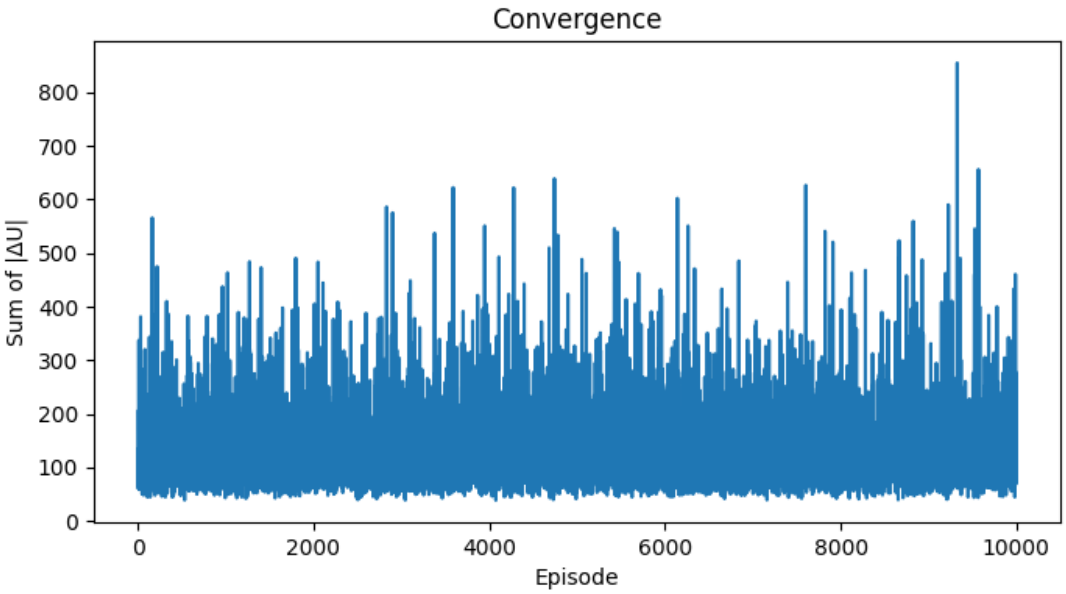

Figure 12: Convergence Plot of Parameters (α=0.5, γ=0.95, ε=0.2)

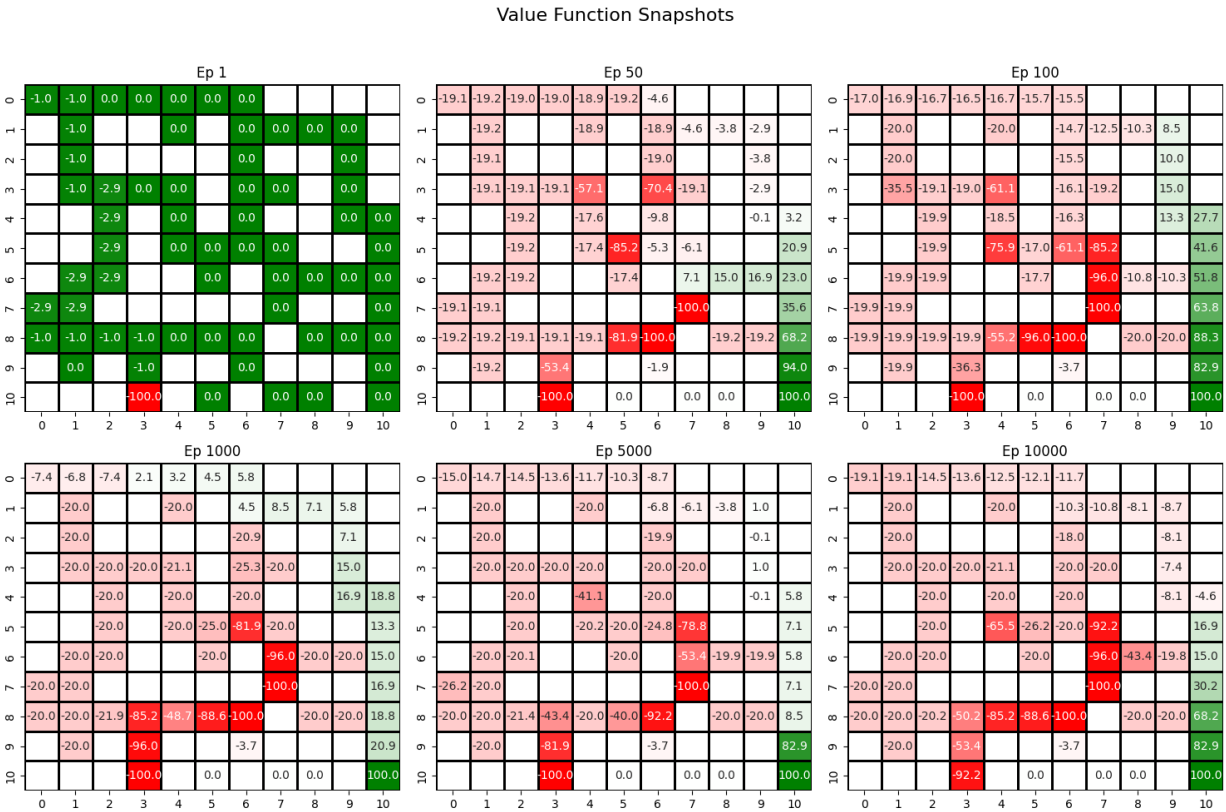

Figure 13: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=0.2)

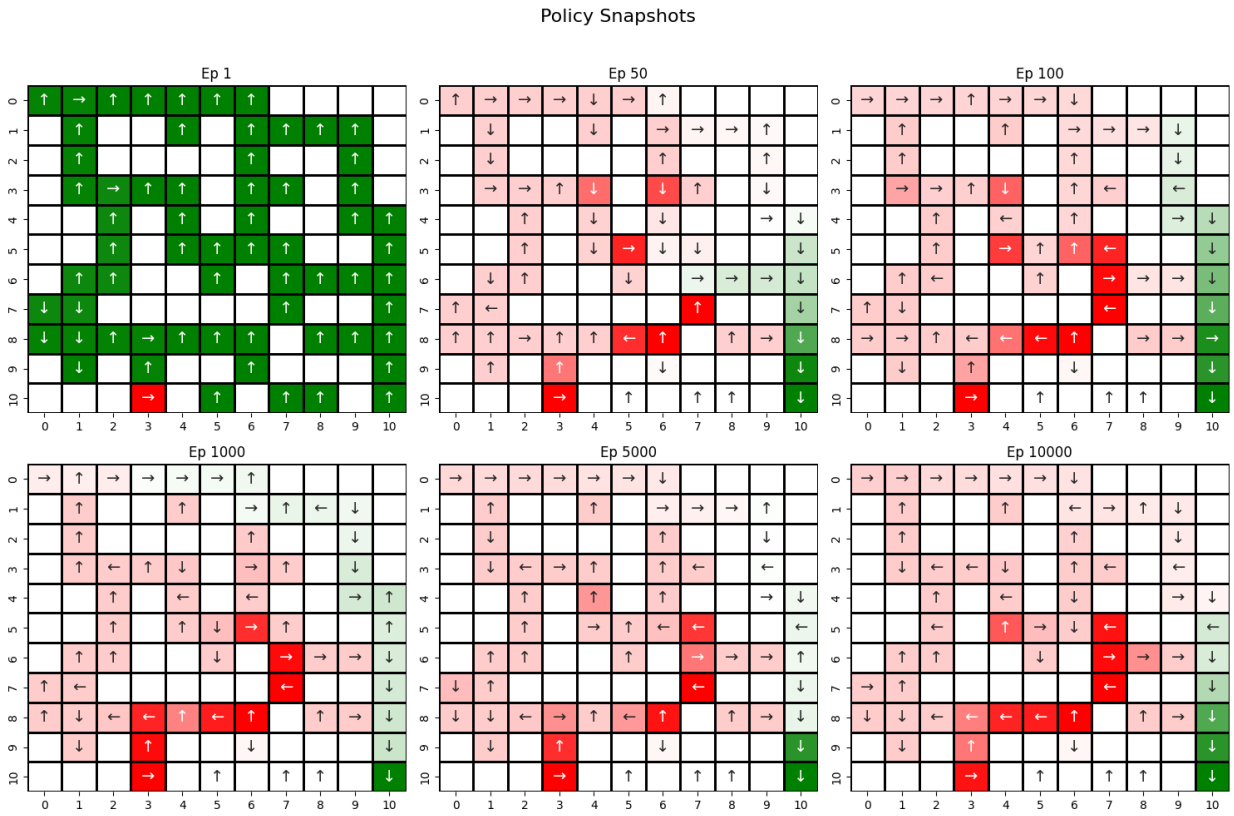

Figure 14: Policy Map of Parameters (α=1.0, γ=0.95, ε=0.2)

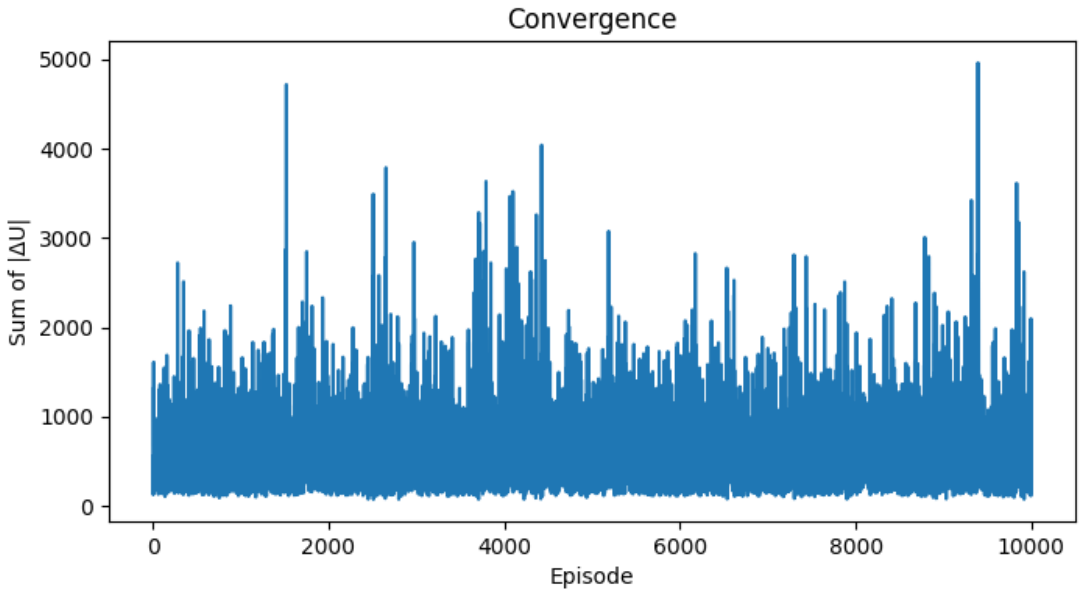

Figure 15: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=0.2)

Gamma Runs (α=0.1 ; γ=0.10,0.25,0.50,0.75,0.95 ; ε=0.2)

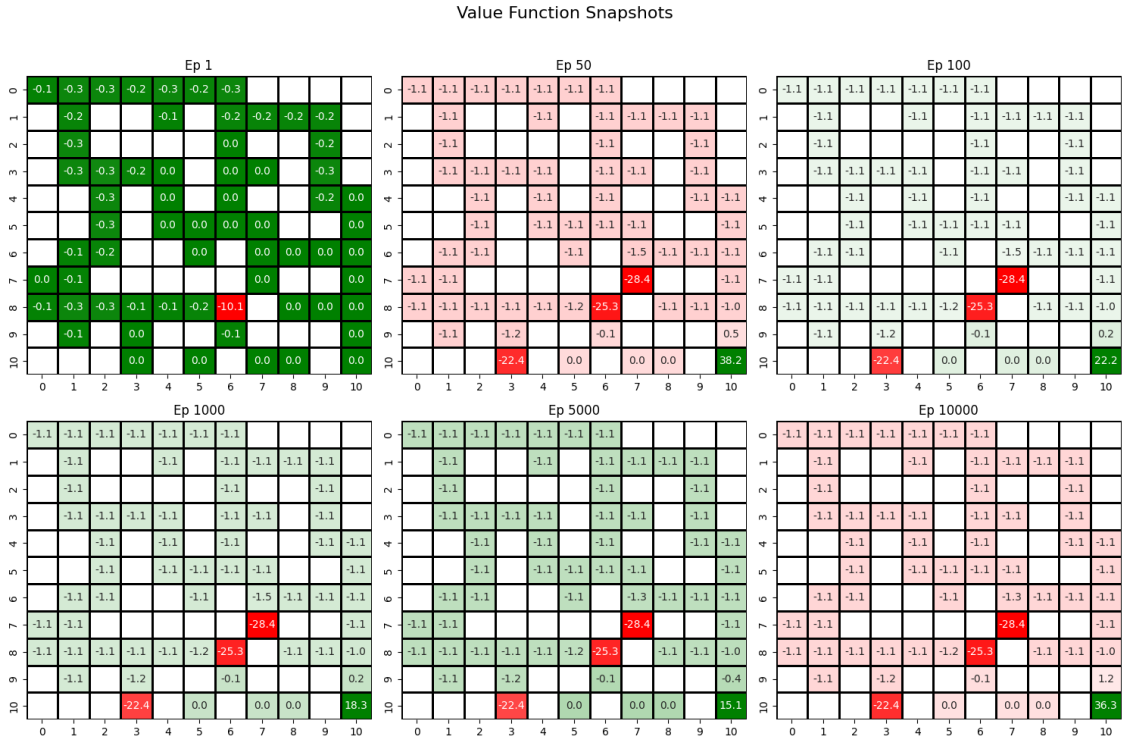

Figure 16: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.10, ε=0.2)

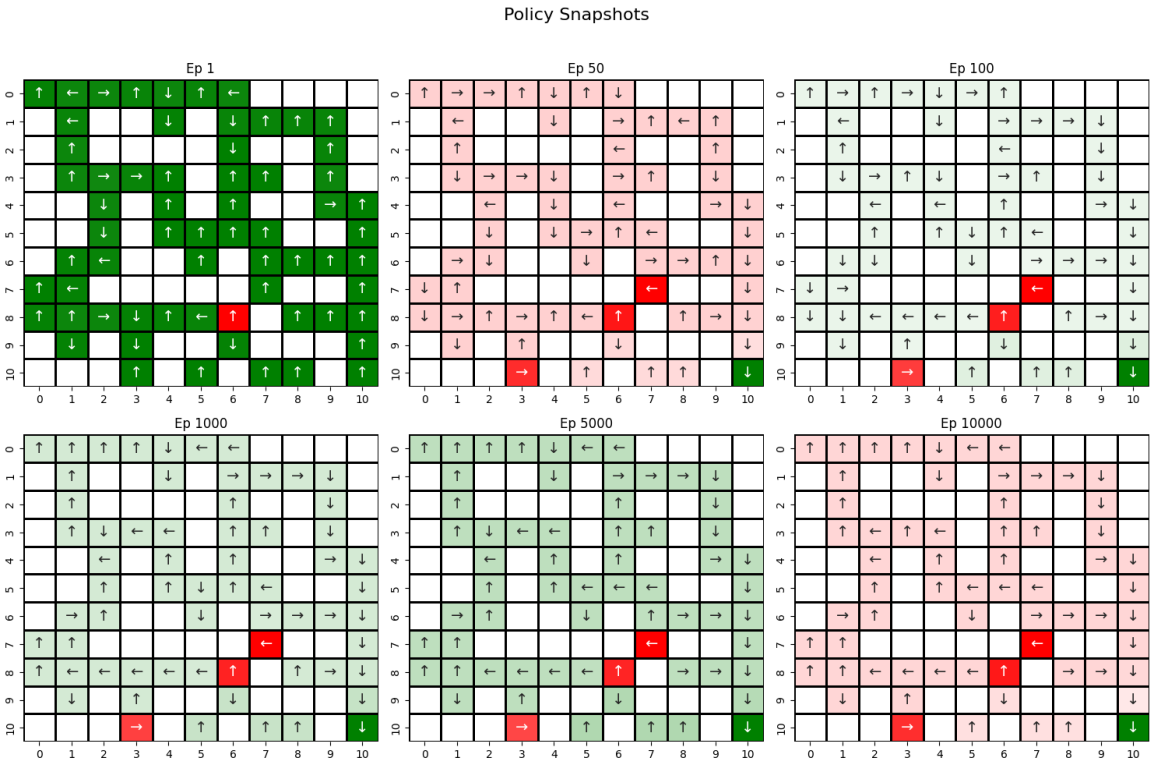

Figure 17: Policy Map of Parameters (α=1.0, γ=0.10, ε=0.2)

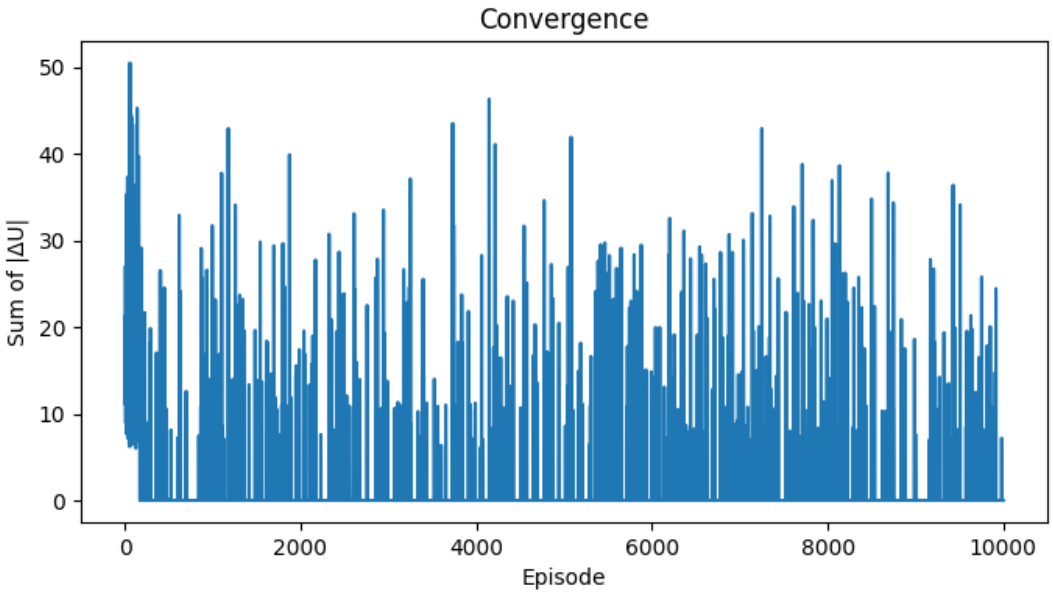

Figure 18: Convergence Plot of Parameters (α=1.0, γ=0.10, ε=0.2)

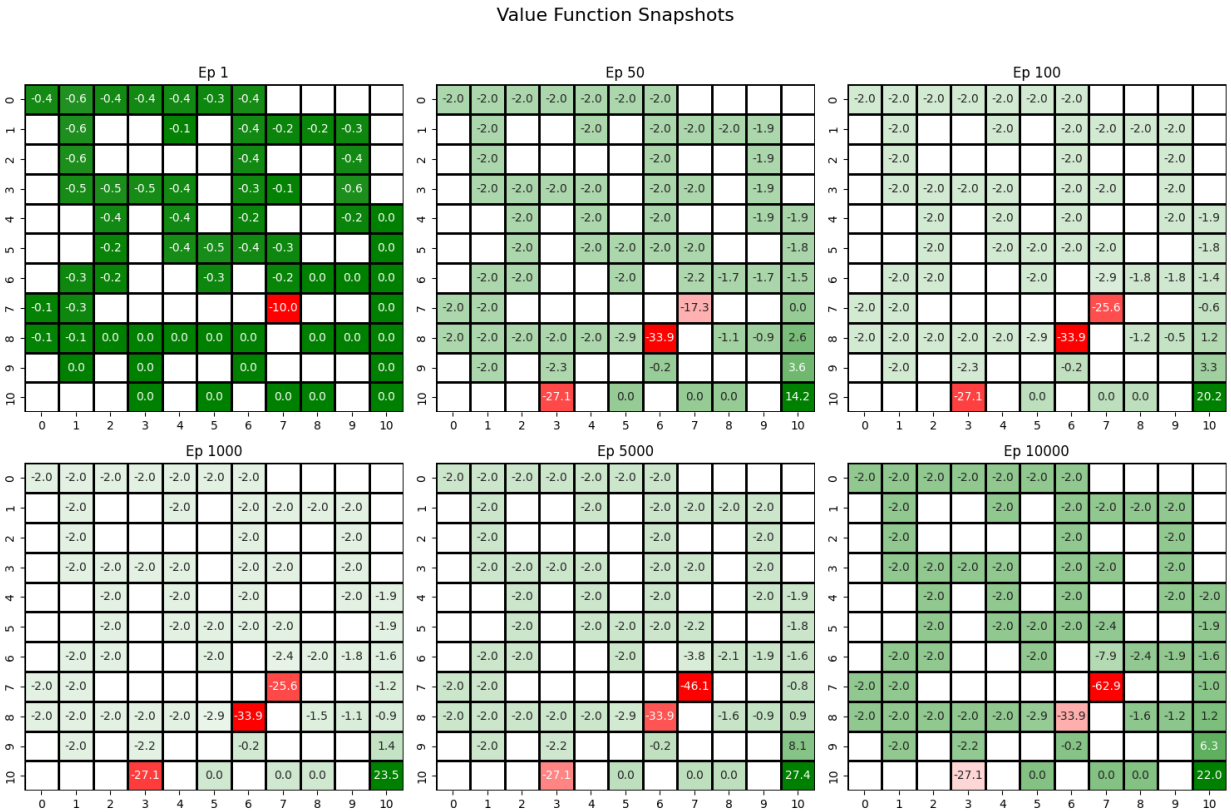

Figure 19: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.50, ε=0.2)

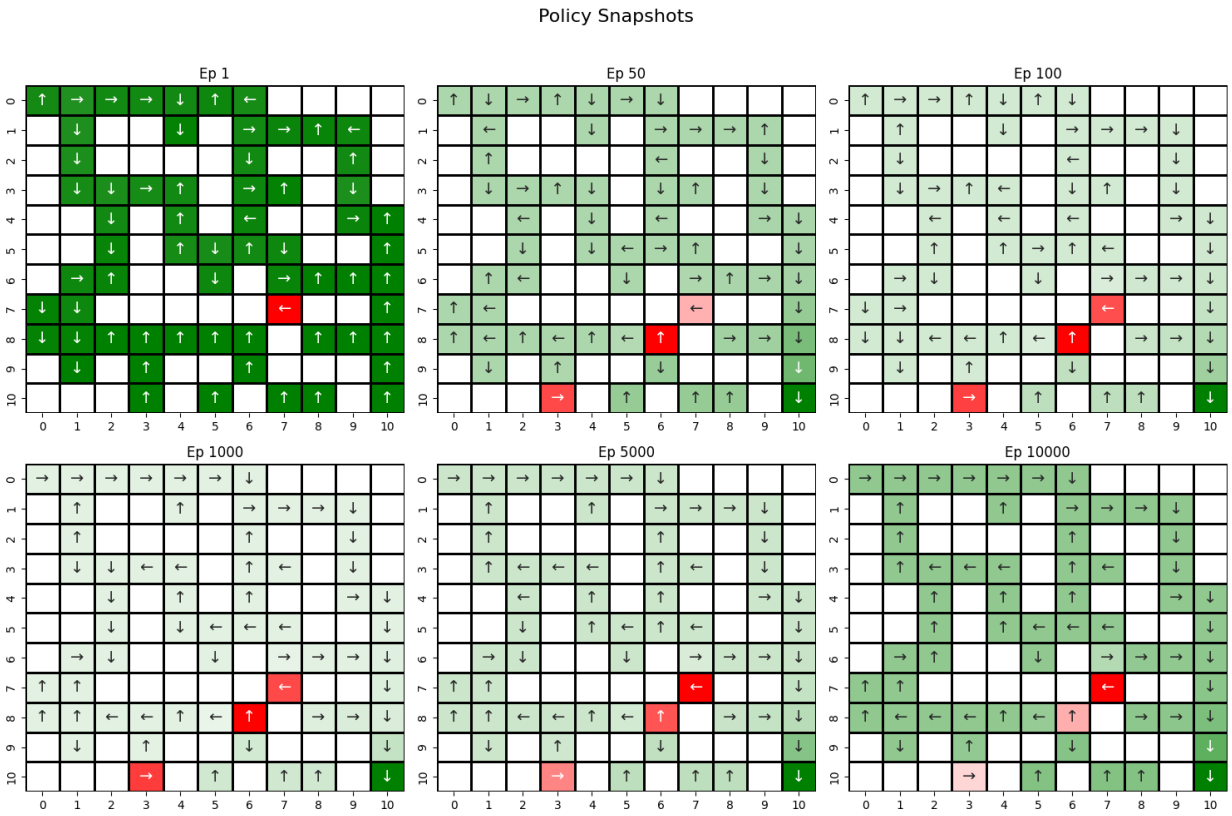

Figure 20: Policy Map of Parameters (α=1.0, γ=0.50, ε=0.2)

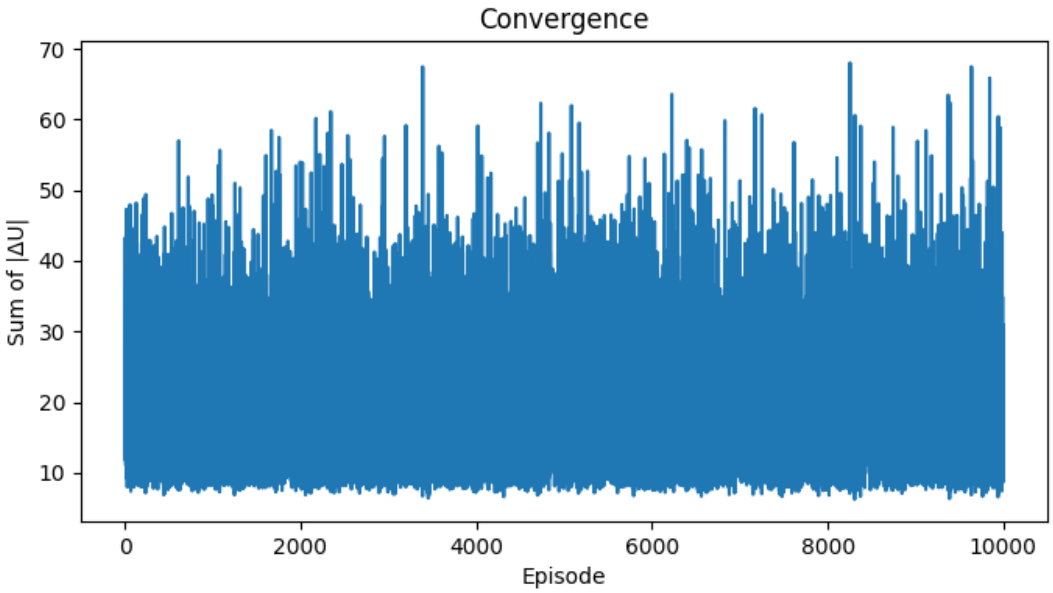

Figure 21: Convergence Plot of Parameters (α=1.0, γ=0.50, ε=0.2)

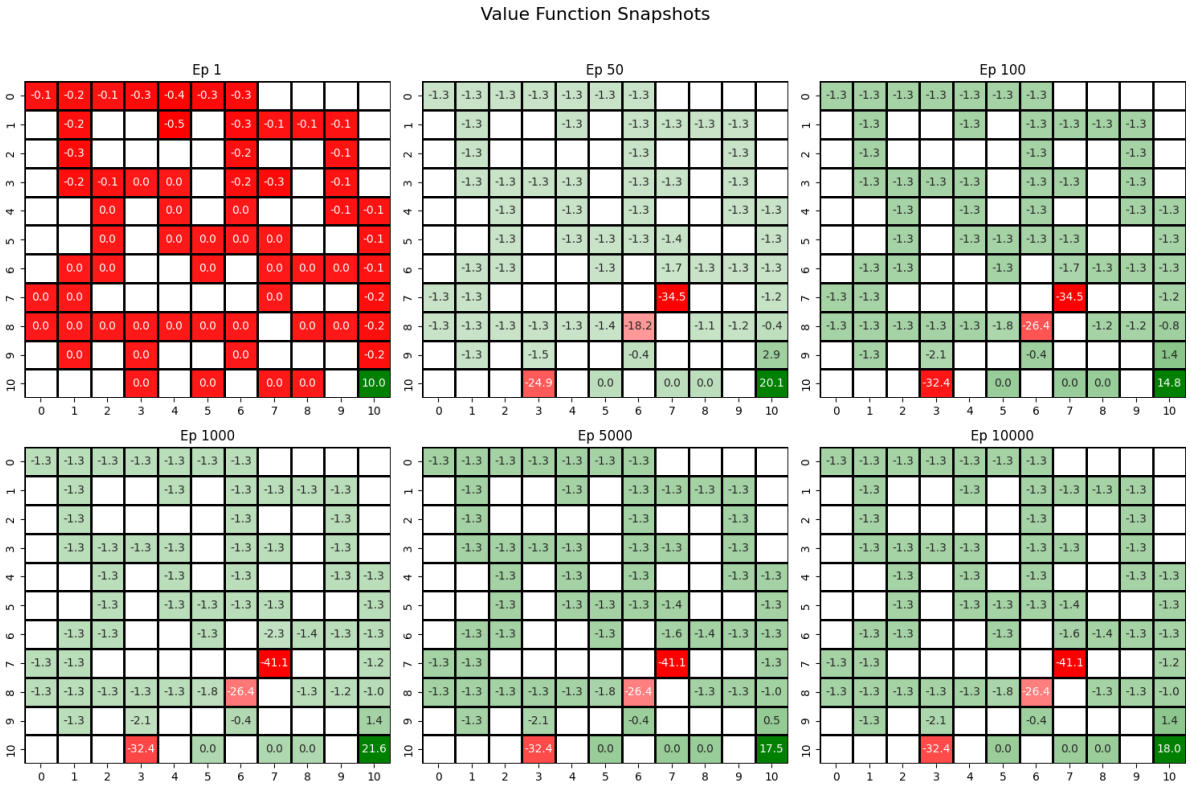

Figure 22: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.25, ε=0.2)

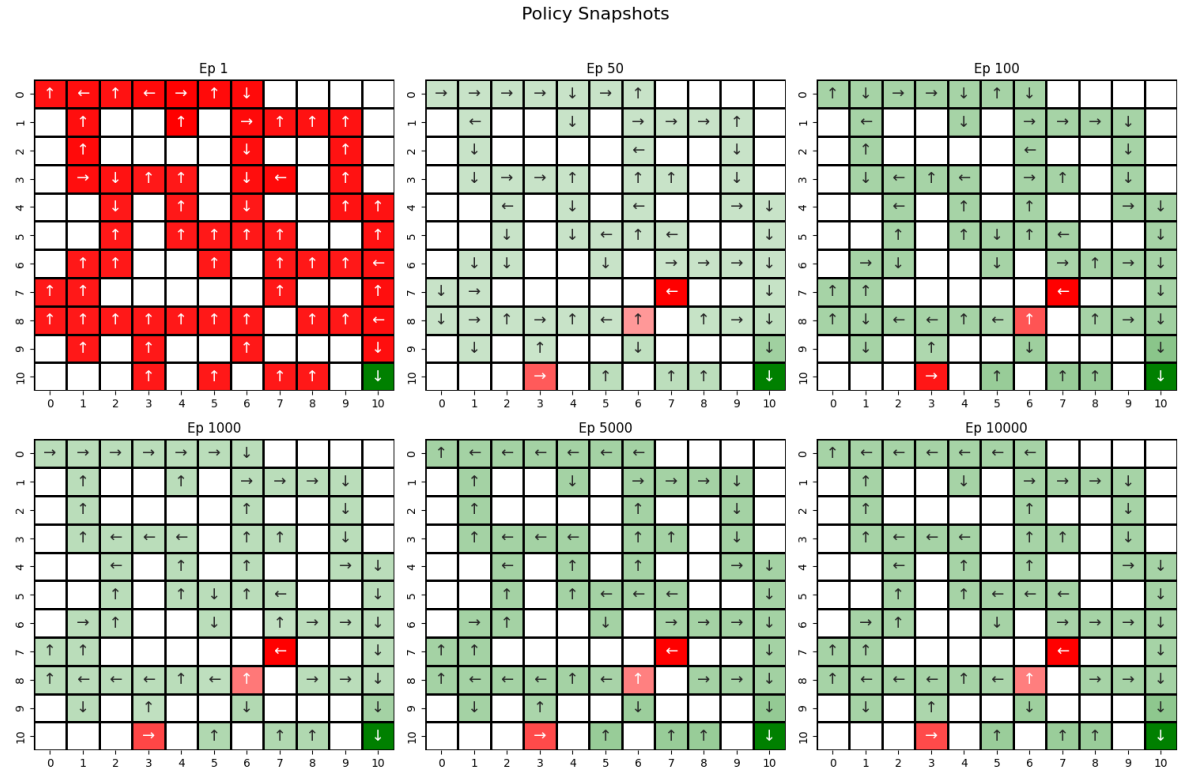

Figure 23: Policy Map of Parameters (α=1.0, γ=0.25, ε=0.2)

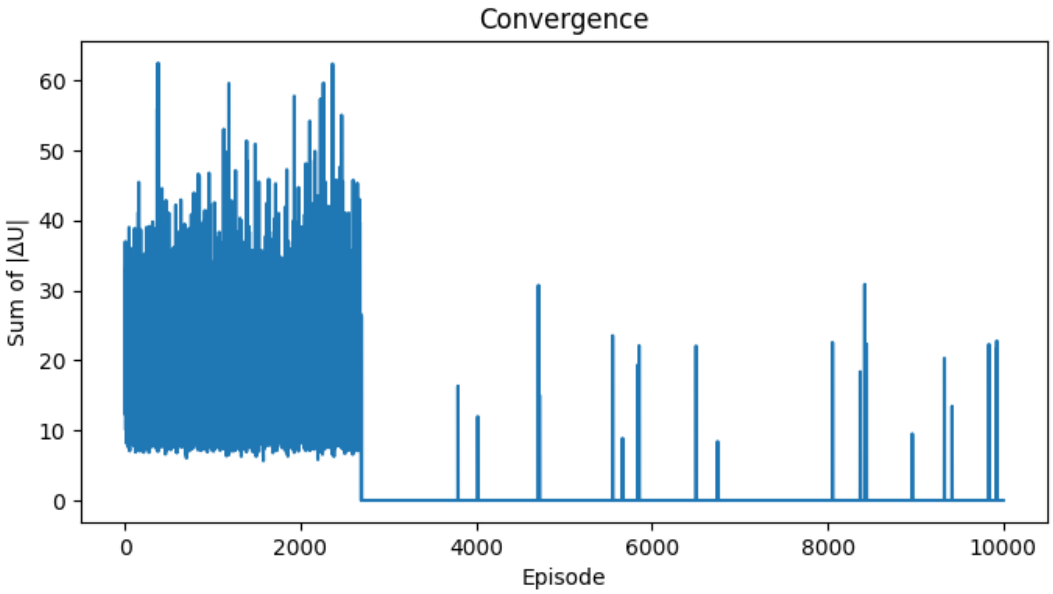

Figure 24: Convergence Plot of Parameters (α=1.0, γ=0.25, ε=0.2)

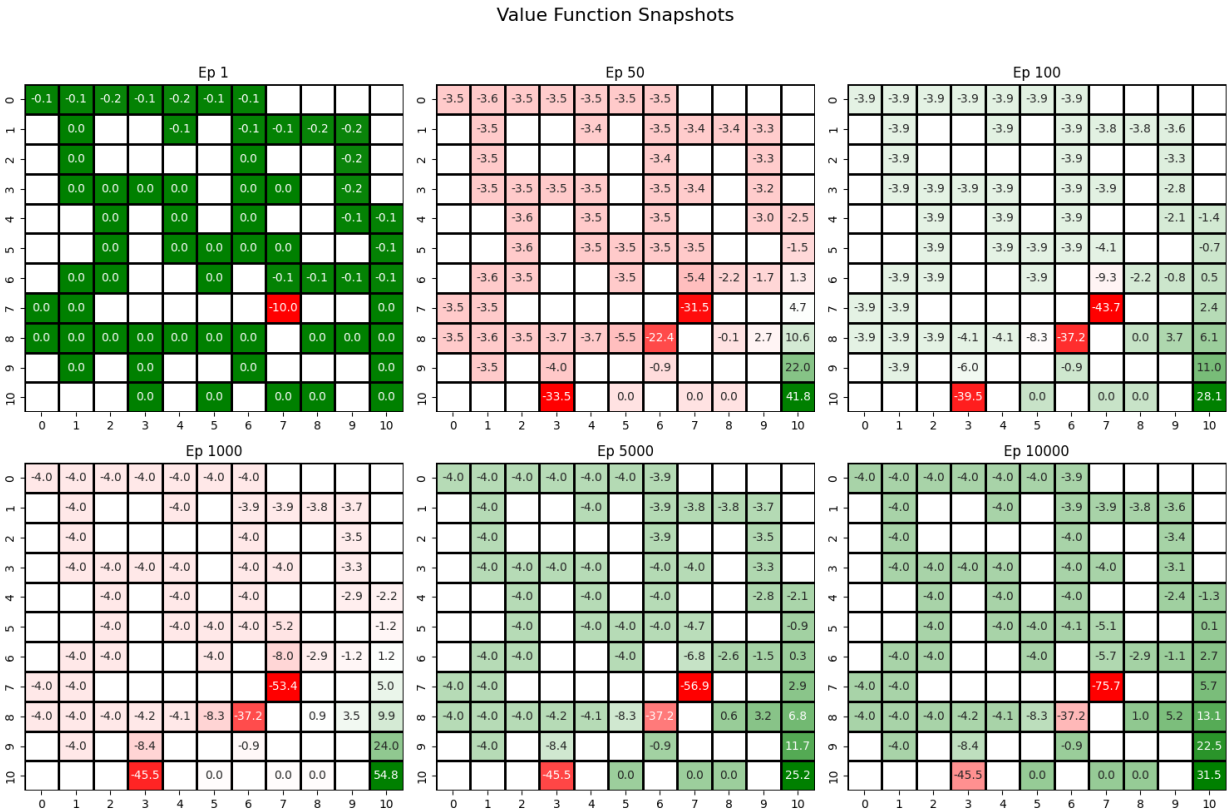

Figure 25: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.75, ε=0.2)

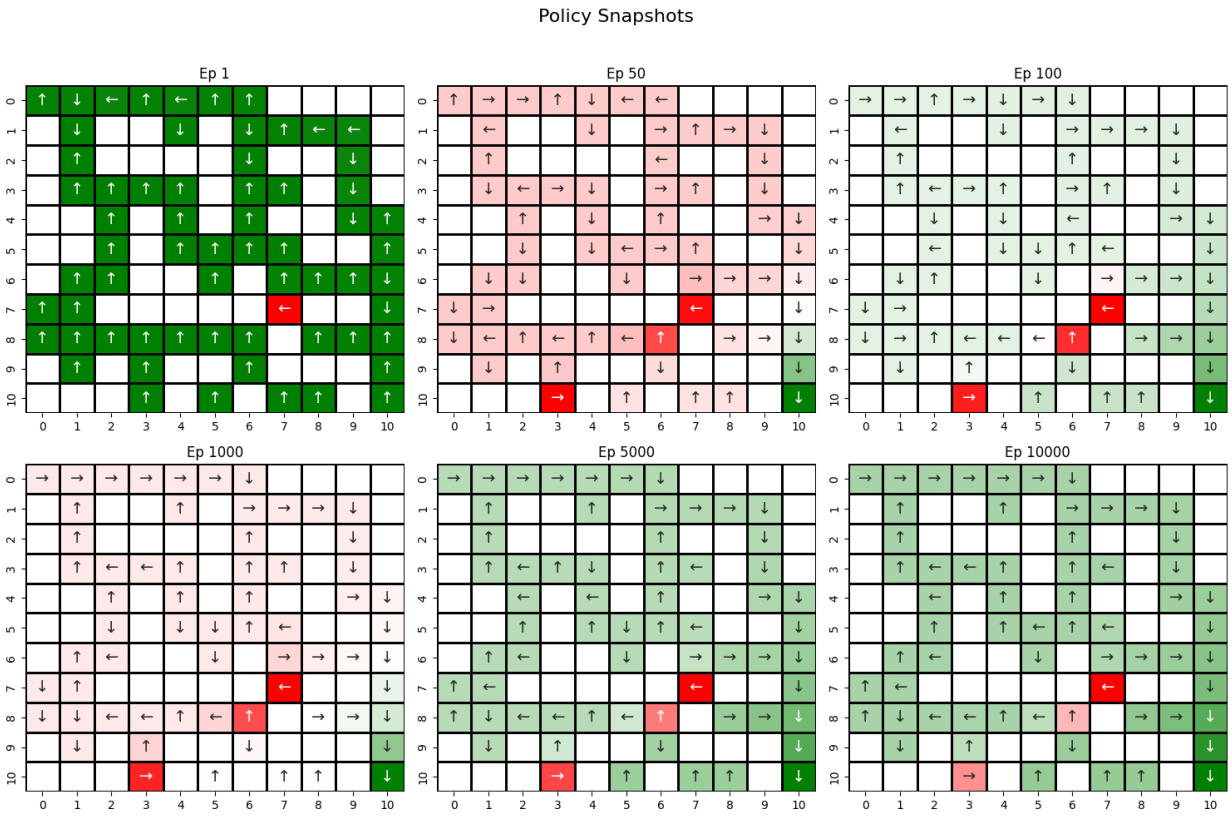

Figure 26: Policy Map of Parameters (α=1.0, γ=0.75, ε=0.2)

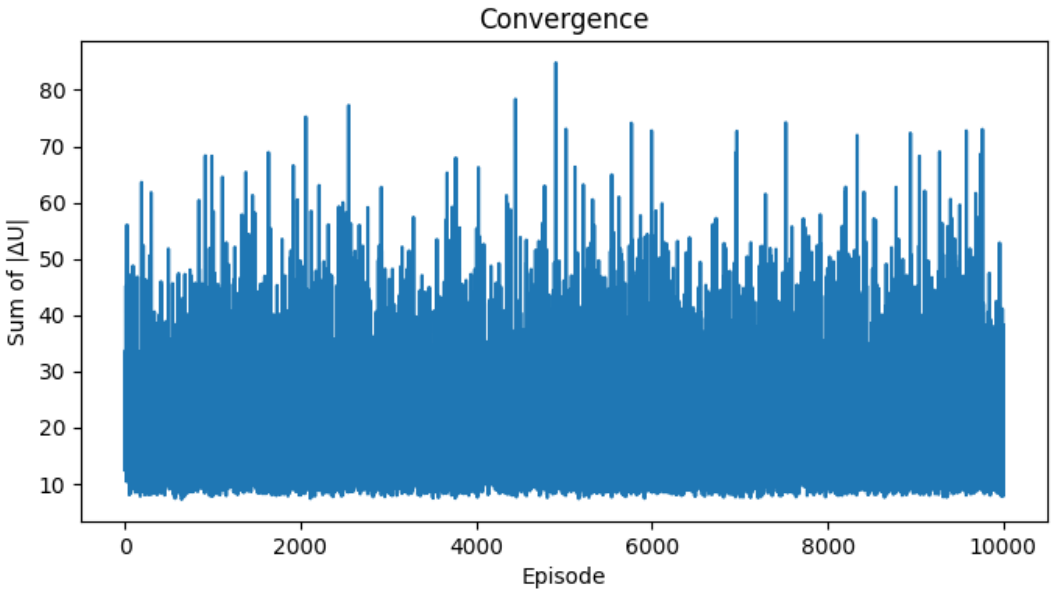

Figure 27: Convergence Plot of Parameters (α=1.0, γ=0.75, ε=0.2)

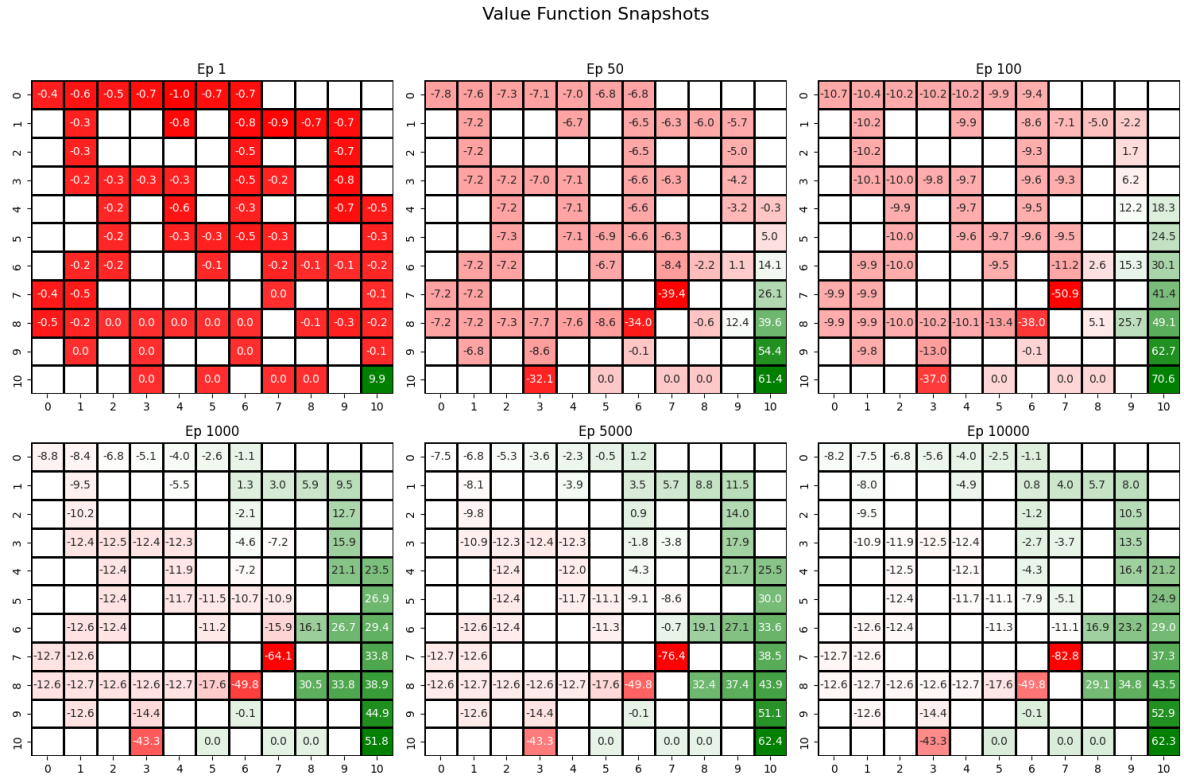

Figure 28: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=0.2)

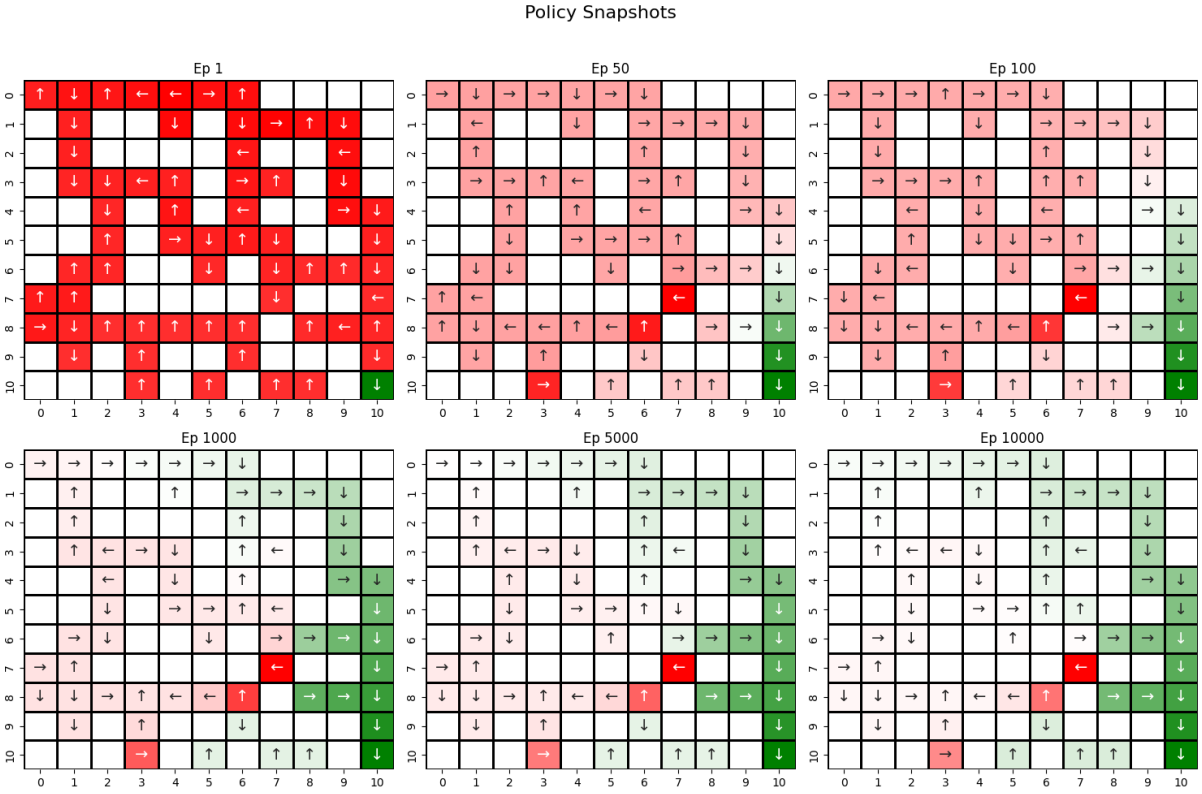

Figure 29: Policy Map of Parameters (α=1.0, γ=0.95, ε=0.2)

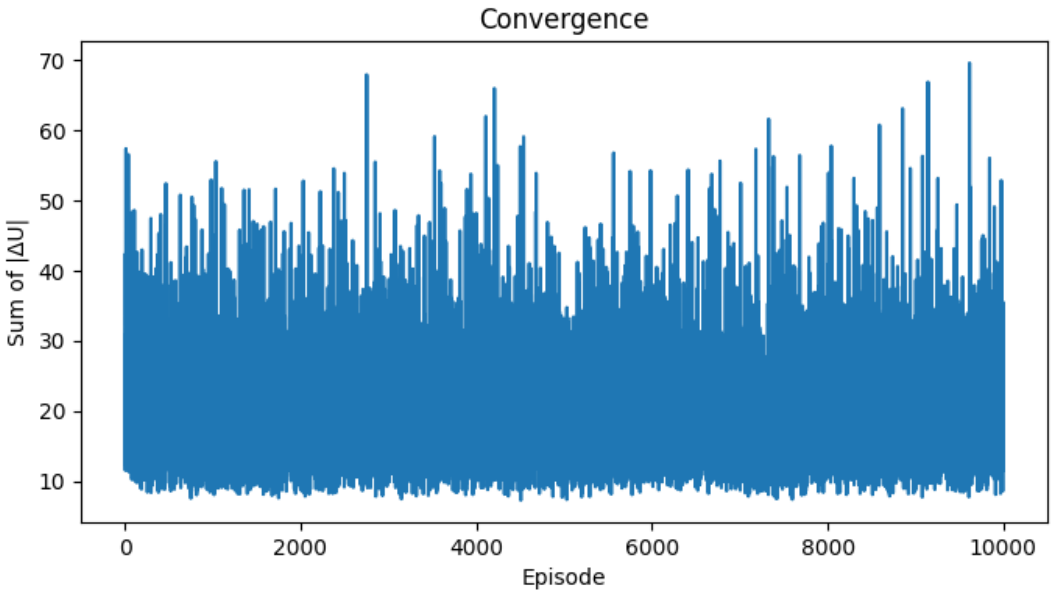

Figure 30: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=0.2)

Epsilon Runs (α=0.1 ; γ=0.95 ; ε=0, 0.2, 0.5, 0.8, 1.0)

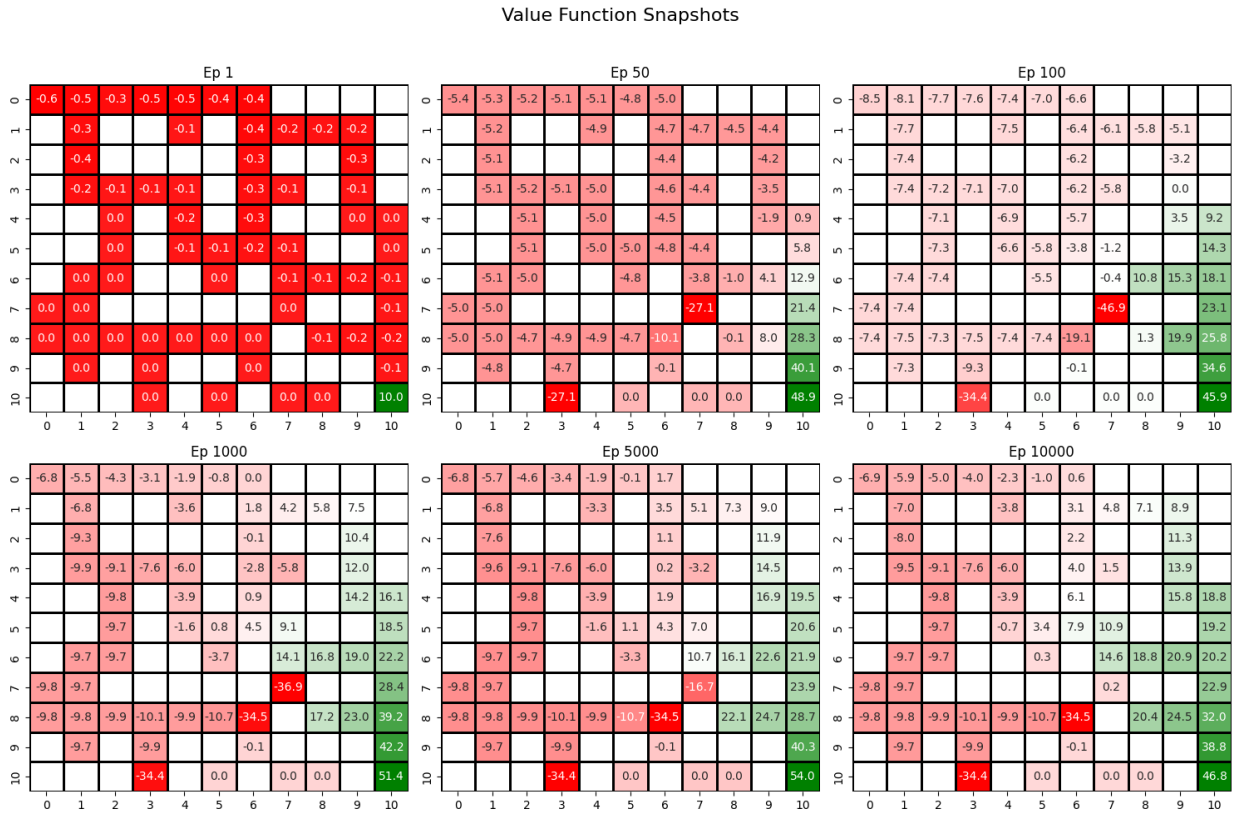

Figure 31: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=0)

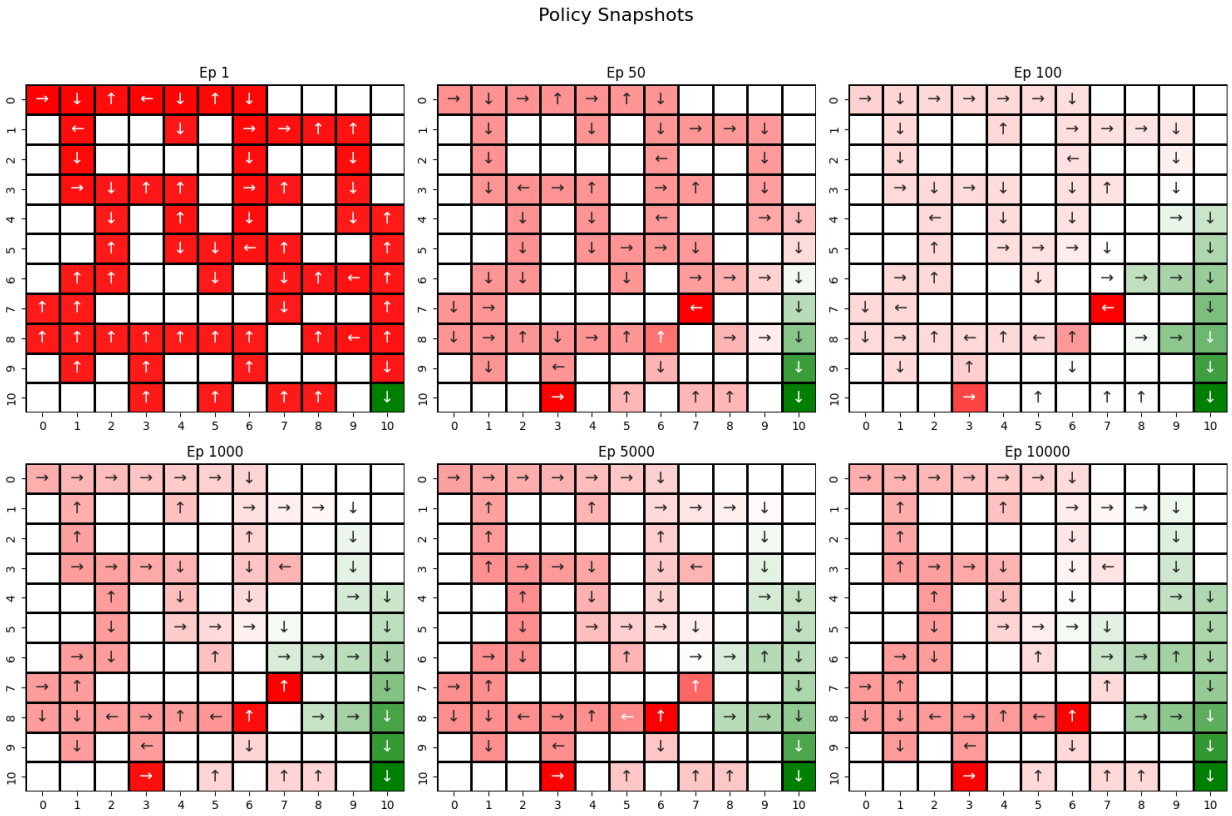

Figure 32: Policy Map of Parameters (α=1.0, γ=0.95, ε=0)

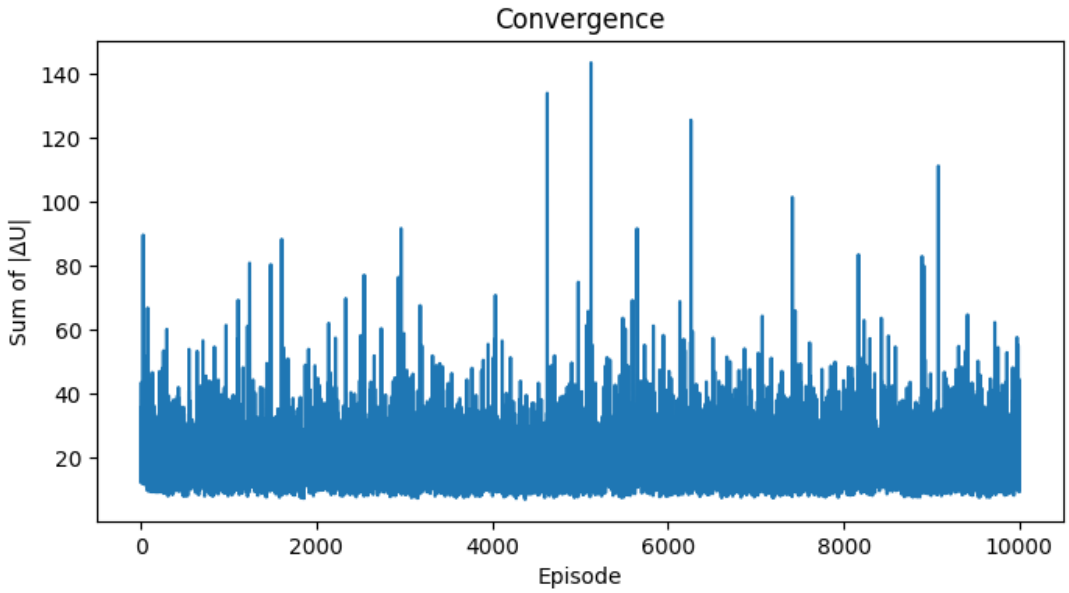

Figure 33: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=0)

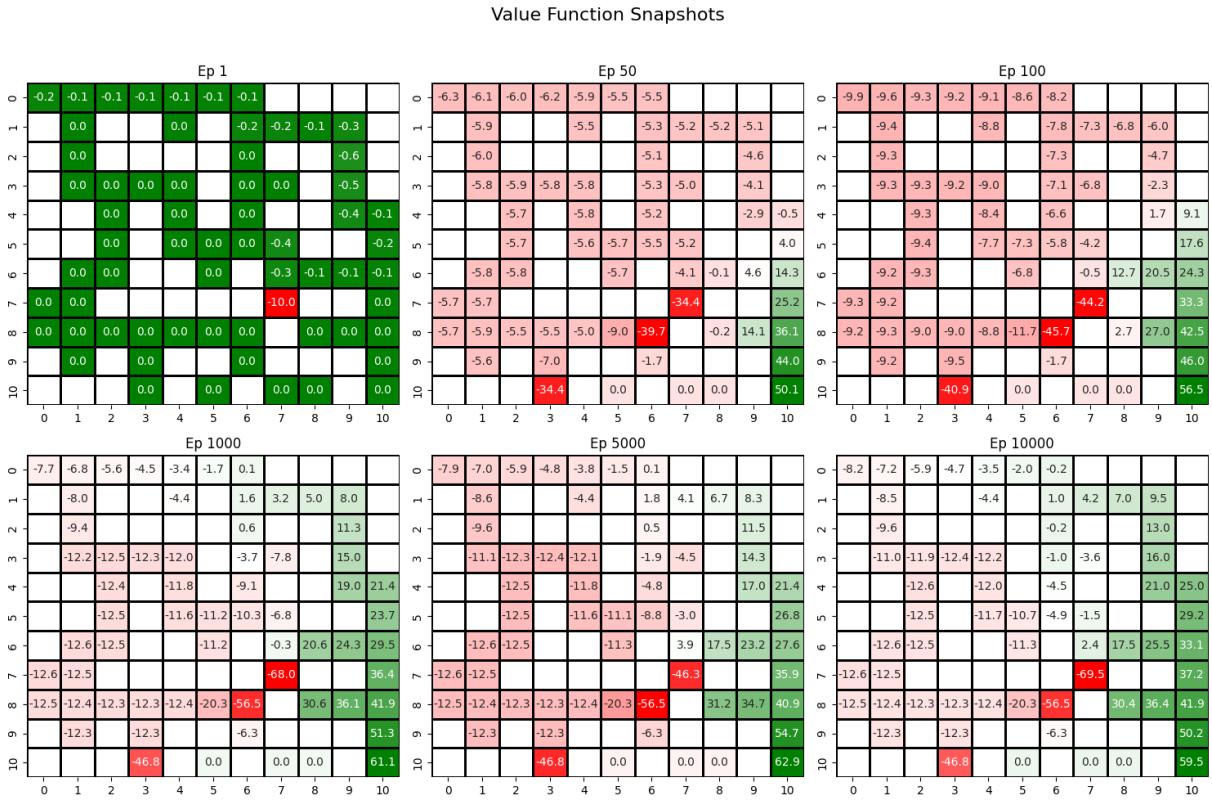

Figure 34: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=0.2)

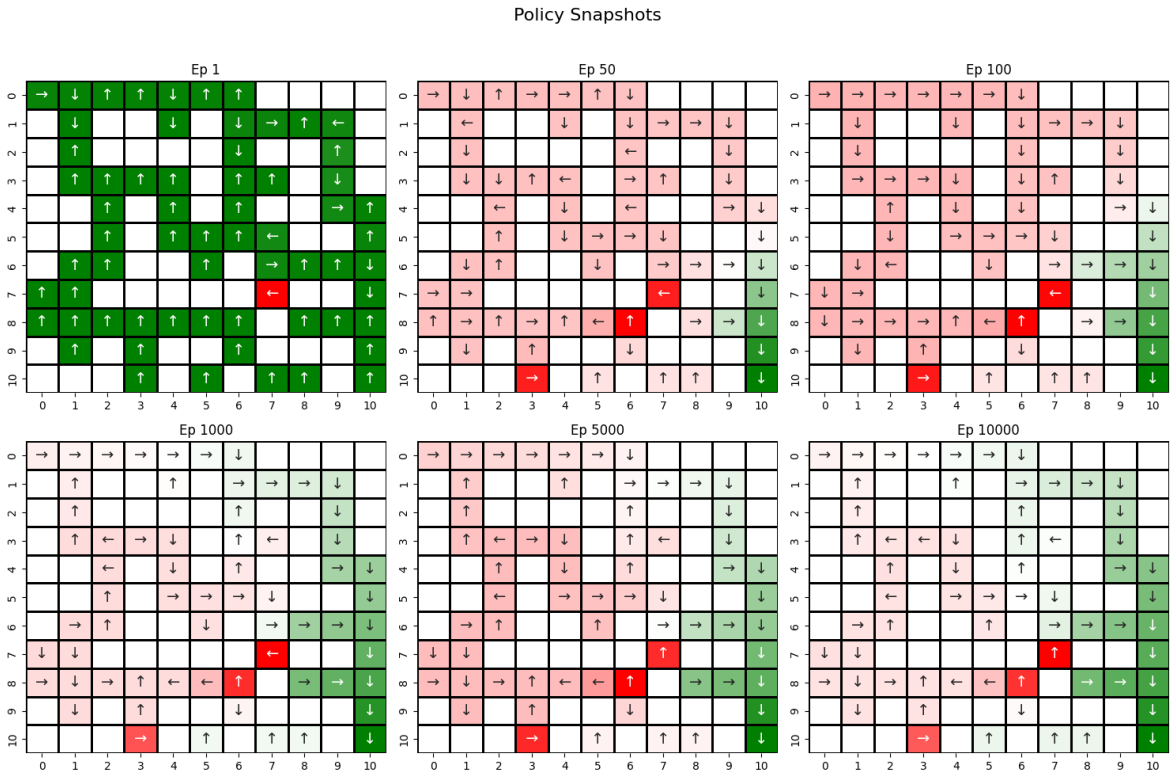

Figure 35: Policy Map of Parameters (α=1.0, γ=0.95, ε=0.2)

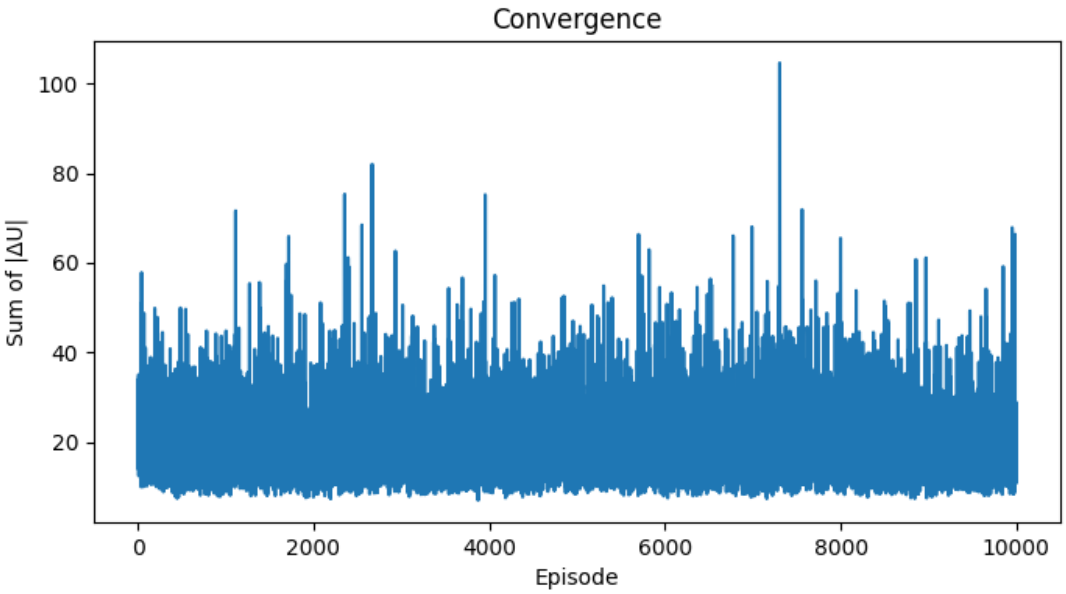

Figure 36: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=0.2)

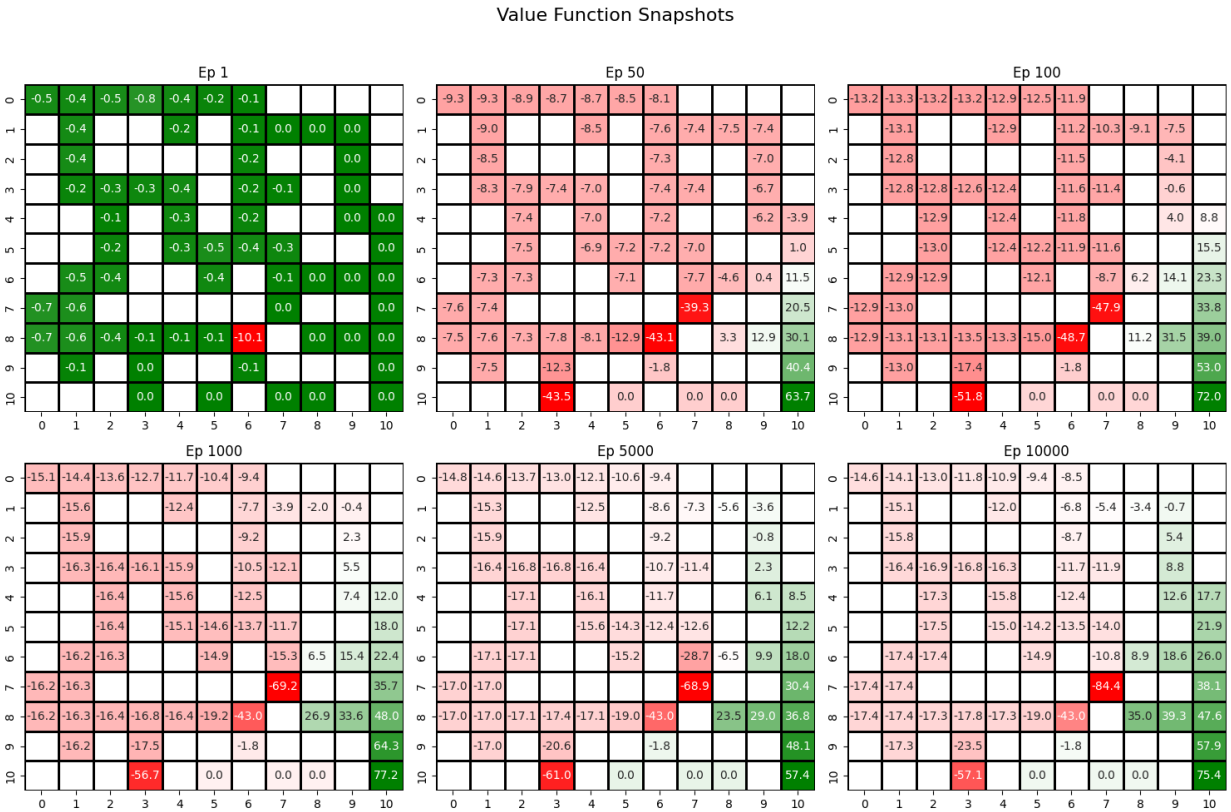

Figure 37: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=0.5)

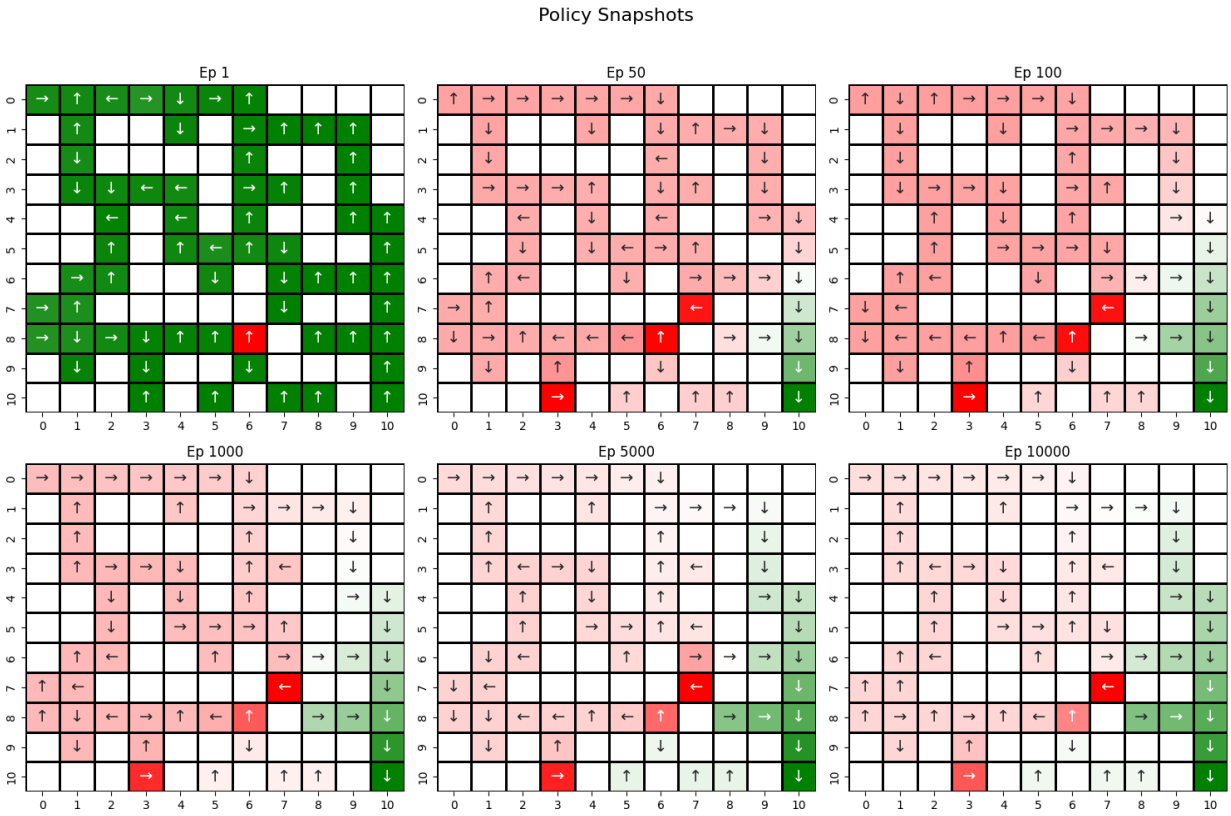

Figure 38: Policy Map Heatmap of Parameters (α=1.0, γ=0.95, ε=0.5)

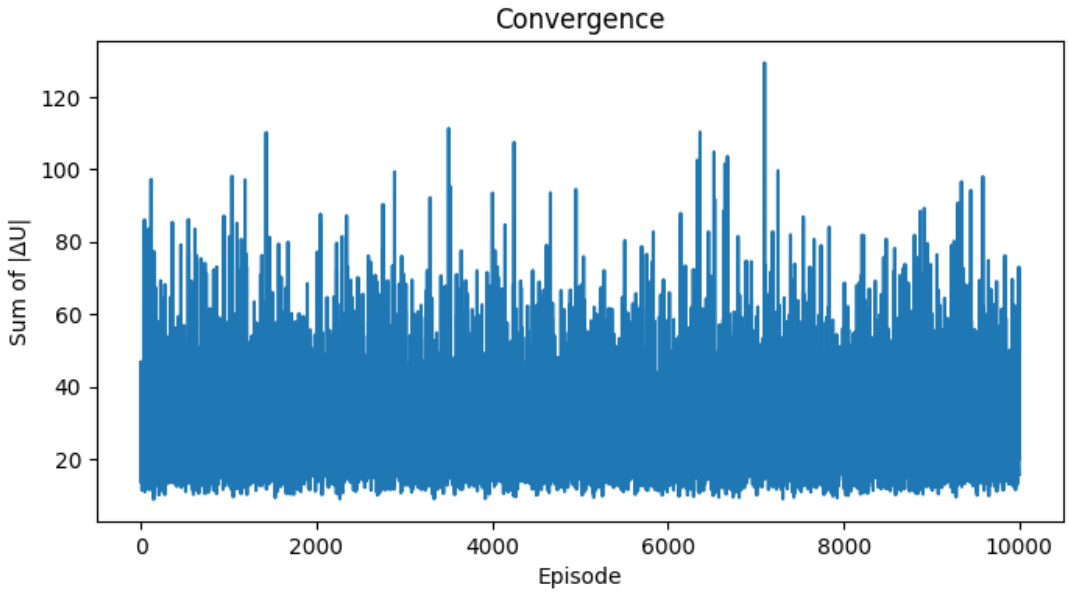

Figure 39: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=0.5)

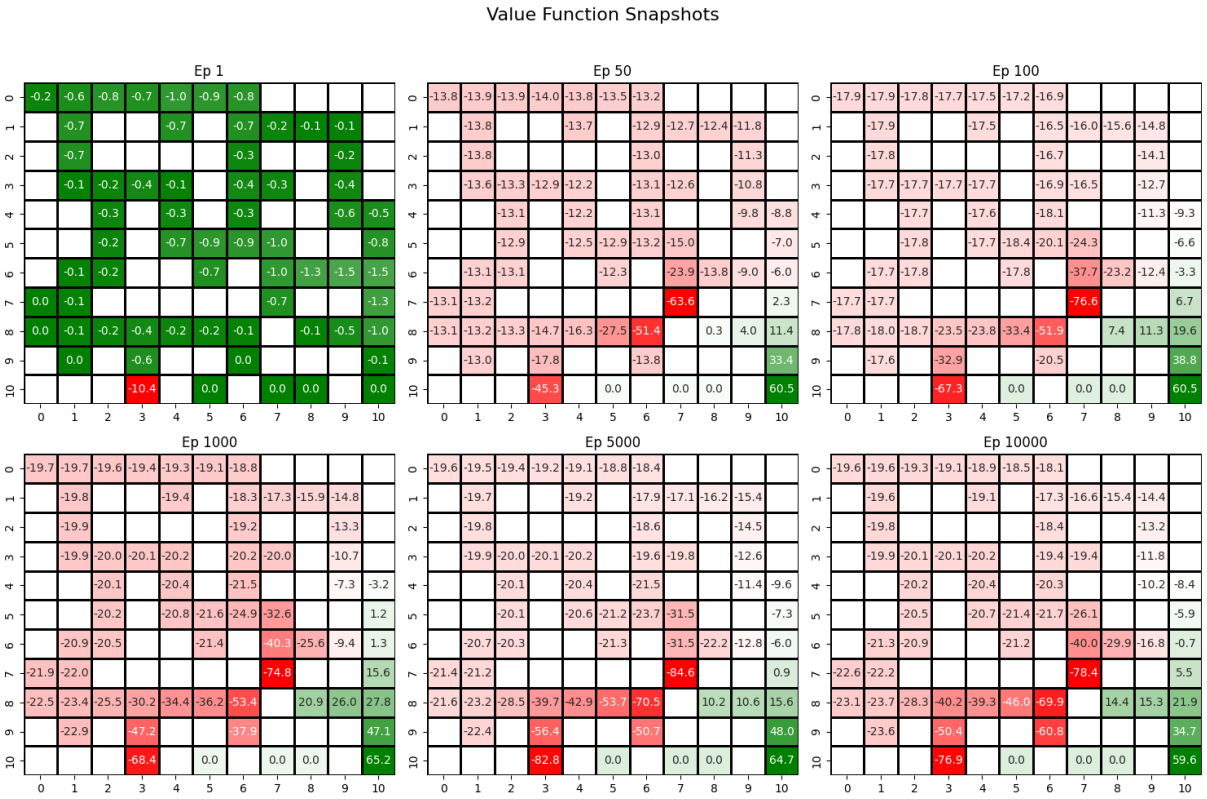

Figure 40: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=0.8)

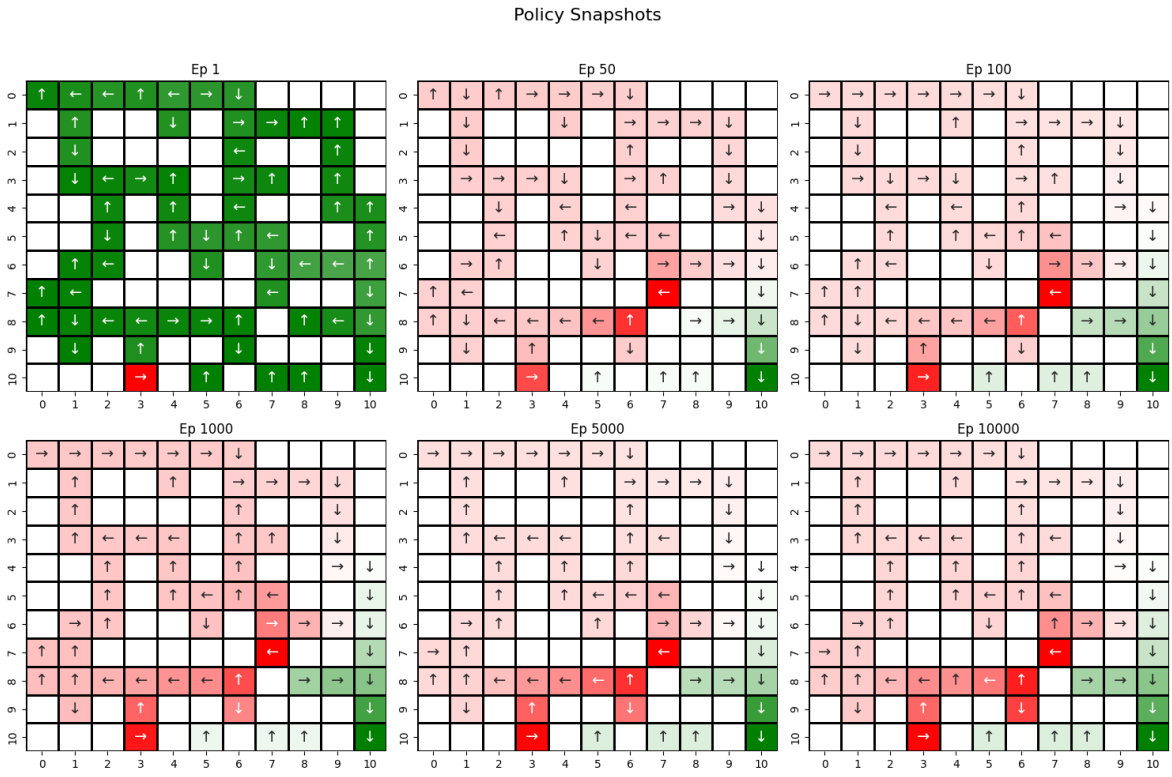

Figure 41: Policy Map of Parameters (α=1.0, γ=0.95, ε=0.8)

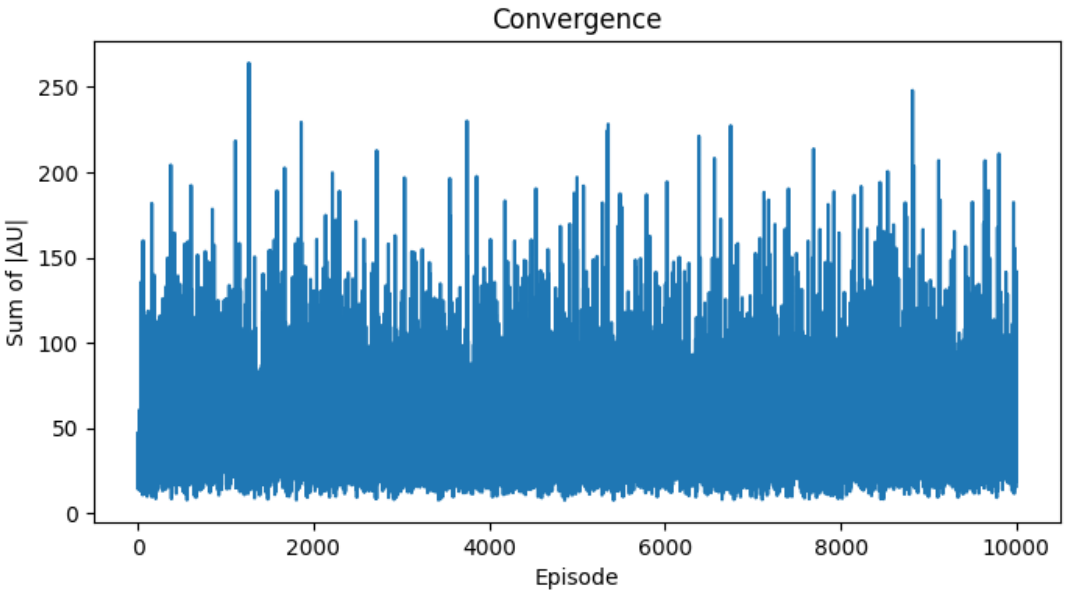

Figure 42: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=0.8)

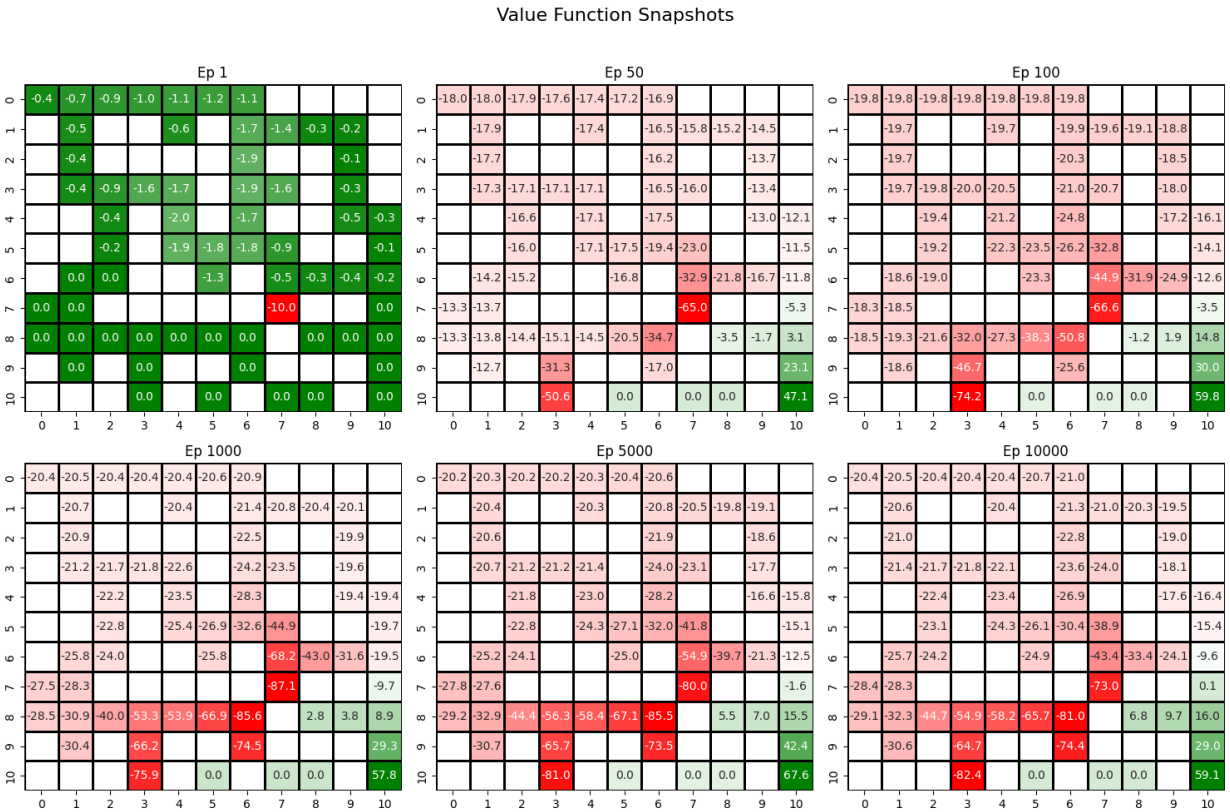

Figure 43: Utility Value Function Heatmap of Parameters (α=1.0, γ=0.95, ε=1.0)

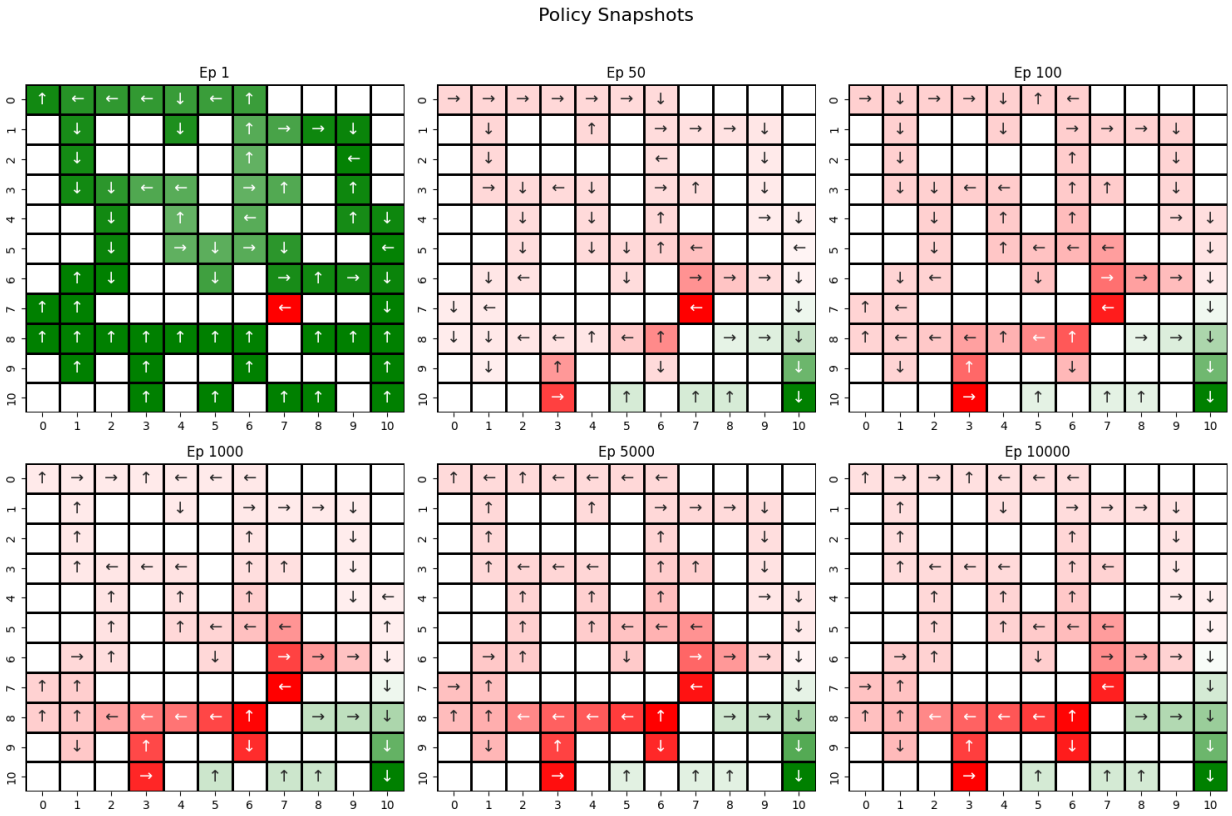

Figure 44: Policy Map of Parameters (α=1.0, γ=0.95, ε=1.0)

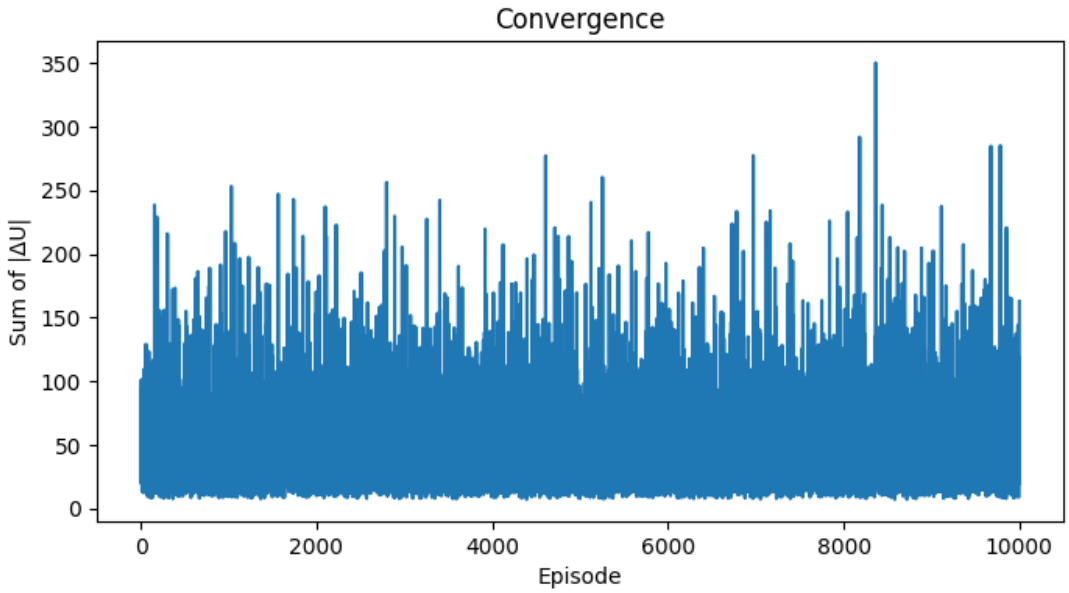

Figure 45: Convergence Plot of Parameters (α=1.0, γ=0.95, ε=1.0)

1.4: Discussions

1. Rationale behind transition probabilities and reward function

We used a 75/5/10/10 split (intended/ opposite/ perpendicular) to inject **mild stochasticity**—enough that the agent can't memorize a single path, yet deterministic enough that utility gradients remain meaningful.

- **Step penalty (–1)**: discourages long, aimless wandering.
- **Goal reward (+100)**: creates a strong uphill gradient toward the green goal cell.
- **Trap penalty (–100)**: deep valleys at each red trap cell force the agent to learn to avoid them.

Together, these shape a utility landscape where the correct navigation corridor stands out clearly over time.

2. Evolution of the utility value functions

- **`alpha_0.1_values (Figure 7)`**
  - At **Ep 1** the grid is nearly uniform.
  - By **Ep 50**, the cells adjacent to the goal (bottom-right corner) brighten (higher utility) and trap regions darken (lower utility).
  - At **Ep 1000**, a smooth gradient from the start at (0,0) down to the goal emerges.
  - By **Ep 10000**, it has fully converged into crisp contour lines leading right then down.
- **`gamma_0.1_values (Figure 16)` VS. `gamma_0.95_values (Figure 28)`**
  - Low γ=0.1 yields a very patchy heatmap even at Ep 10000—values barely propagate far from immediate neighbors.
  - High γ=0.95 shows a clear, correct gradient by Ep 5000 even by Ep 1000.
  - This observation clearly shows that under a small discount factor utility value functions have trouble of evolving to a proper convergence.
- **`epsilon_1.0_values (Figure 43)` VS. `epsilon_0.2_values (Figure 34)`**
  - With pure random (ε=1.0), the "gradient" remains noisy even after 10000 episodes, since you never exploit. Due to the high exploration rate the optimal path is not fully clear.
  - At ε=0.2, a stable gradient appears by Ep 1000.
  - We can easily observe that under lower exploration rate the optimal path is much easier to appear.

3. Convergence of the utility function

Every utility value function shows clear convergence around goal and trap squares. For specific parameters we can summarize convergence of the utility value functions such as:

- Discount Factor: It is important to point out that low discount factor (gamma) <0.95 runs have trouble converging to a point where the most optimal path is easily observable from start to goal. This discount factor issue can be easily observed from Figures 16,19,22,25.

However, the convergence of utility values around the goal and the traps are observable. 0.95 case converges around episode 1000, other cases of discount factors are having trouble converging even at episode 10000.

- Initial Exploration Rate: Inspecting Figures 43,40,37,34,31 yields that lower initial exploration rate results in better optimal path convergence. However, the convergence success between different epsilon parameters is not that drastic like the discount factor. Figure 43 (epsilon=1) is the only figure where it is hard to see the optimal path, in other Figures it is easily observable. Almost all of the epsilon runs converges around Episode 1000.
- Learning Rate: Inspecting Figures 1,4,7,10,13 yields that alpha values of 0.5 and 1 are the best ones. We can say that small learning rate runs have trouble revealing the optimal path to goal, while learning rates of 0.5 & 1 clearly show the optimal path to goal. However, again in every figure convergence around goal and traps are easily observable. At learning rate = 1, map converges very fast around Episode 50, as we lower the learning rate convergence happens at a later episode. For example, at learning rate = 0.001, map is still converging at episode 10000.

4. Sensitivity to α and γ

- **α (learning rate)**
  - `alpha_0.001(Figure 1)`: convergence remains slow—ΔU stays high even at 10000 episodes.
  - `alpha_1.0(Figure 13)`: large oscillations/spikes in ΔU indicate overshooting (utility updates too aggressive).
  - **Sweet spot**: α≈0.01–0.1 yields the fastest, smoothest convergence (`alpha_0.01(Figure 4)`, `alpha_0.1(Figure 7)`).
- **γ (discount factor)**
  - **Low γ** (0.10,0.25): utilities propagate only a few steps from the goal, so the gradient looks flat in many areas—even after 10000 episodes.
  - **High γ** (0.75,0.95): full horizon lookahead yields crisp utility gradients.
  - **Convergence timing** also tracks with γ: higher γ → faster convergence in ΔU.
  - Discount factor that is close to 1 is desired as the figures show.

5. Implementation challenges

- **Mapping stochastic slips** (opposite vs. perpendicular) without indexing errors.
- **Masking invalid moves** so wall-hits do not update off-grid cells.
- **Runtime blow-up** for low γ runtime is really long (6-8hrs)— Because a low discount factor (e.g. γ = 0.1) makes the future goal reward effectively vanish, the agent sees only the –1 step penalties and drifts randomly to avoid traps—so episodes routinely run out to 10 000-step cap. Each of those long episodes does thousands of TD updates, and multiplying by 10 000 episodes means on the order of 10^8 updates, which in pure Python can take many hours. By capping each episode at 500 steps for the low-γ sweeps, we

immediately limited the worst-case updates per episode from 10 000 down to 500, bringing those runs back into the realm of minutes instead of overnight.

6. Impact of ε-greedy exploration

- **`epsilon_1.0`** (exploration): remains a sea of arrows pointing every which way—no coherent policy emerges.
- **`epsilon_0.8`**: early exploration helps avoid traps, but policy only smooths out around Ep 5000.
- **`epsilon_0.2`**: balances exploration/exploitation—policy corridor to goal solidifies by Ep 1000.
- **`epsilon_0.0`** (exploitation): very fast "convergence" of a policy, but prone to **locking in** a suboptimal path if early utilities are noisy.

7. Maze design modifications

- **Moving traps or walls** to test adaptability.
- **Multiple goal cells** with varied rewards to encourage multi-modal routing.
- **Noisy rewards** (reward ± some noise) to simulate sensor error.
- **Larger mazes** (e.g. 20×20) to test scaling of tabular methods vs. function approximation.

8. Algorithmic enhancements

- **Annealed ε-greedy**
  Start with a high ε for broad exploration, then gradually reduce it (for instance, $\varepsilon_t = \varepsilon_0 \cdot \text{decay}^t$) so the agent naturally shifts from exploring new paths to exploiting its learned policy.
- **State-action adaptive α**
  Use a learning-rate schedule that decays α for each state–action pair based on how often it's been updated (e.g. $\alpha_{sa} = 1/(1 + \text{visits}_{sa})$), so early visits adjust quickly but later updates fine-tune more gently.
- **Eligibility traces (TD(λ))**
  Integrate multi-step backups by keeping a short "memory" of recently visited states, which speeds up credit assignment and smooths learning.
- **Prioritized sweeping**
  Rather than backing up every visited state equally, maintain a priority queue of states whose value changes most and focus updates where they'll have the greatest ripple effects.

2: Deep Q-Learning

2.3: Experimental Work

Learning Rate (α):

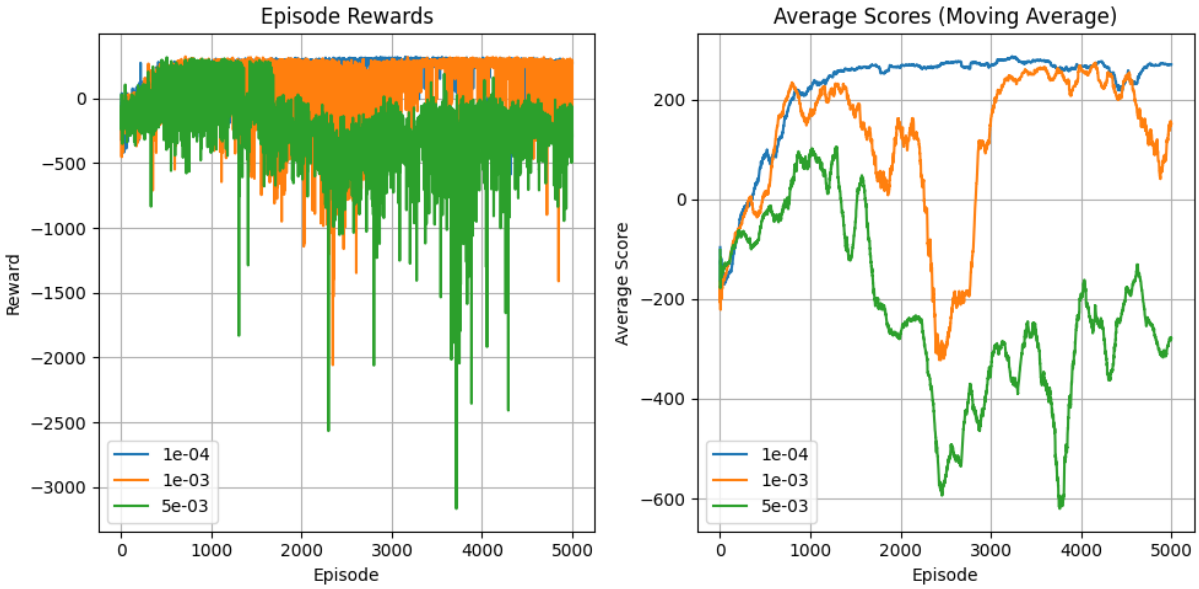

Figure 46: Learning Curve Plot for Learning Rate

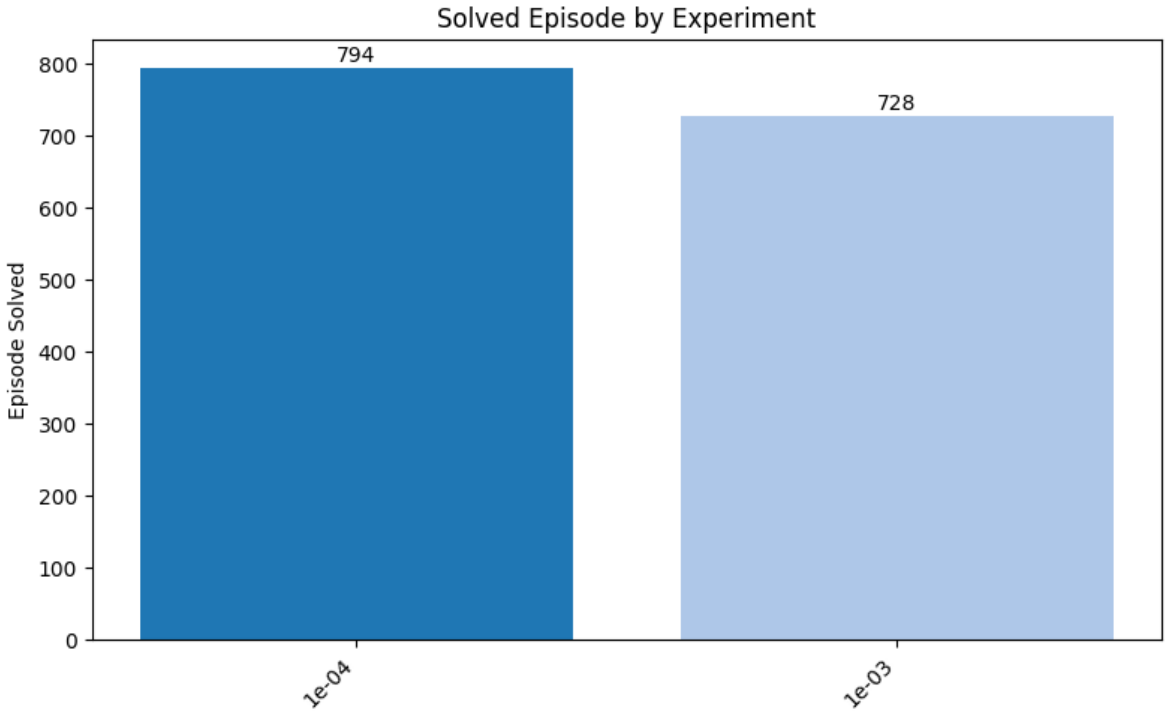

Figure 47: First Solved Episode for Learning Rate

Discount factor (γ):

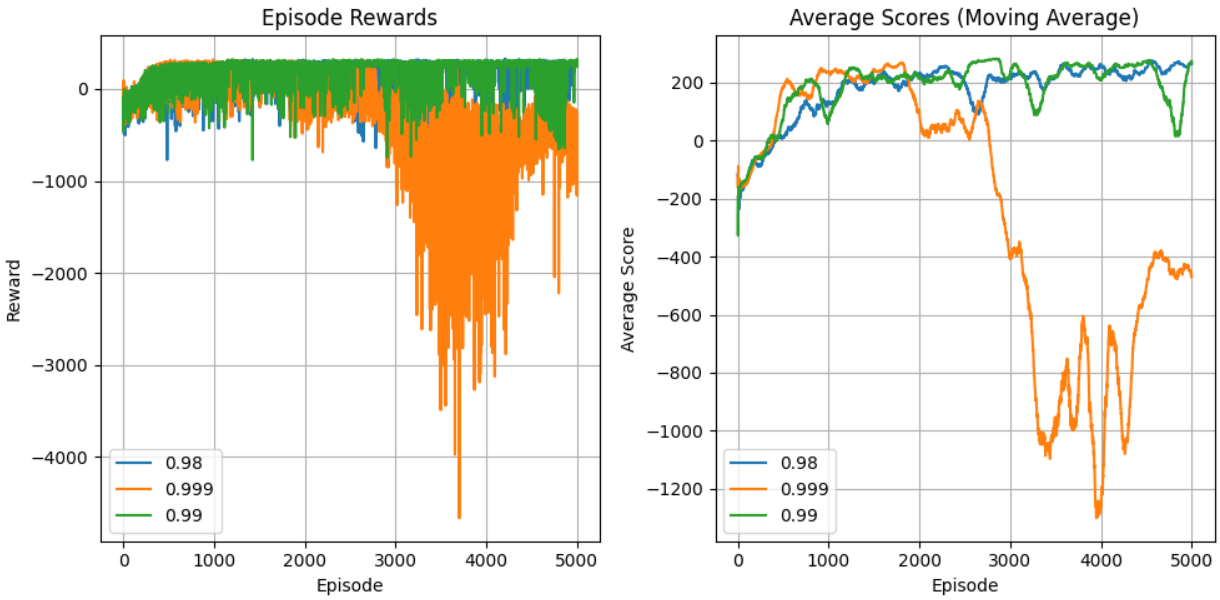

Figure 48: Learning Curve Plot for Discount Factor

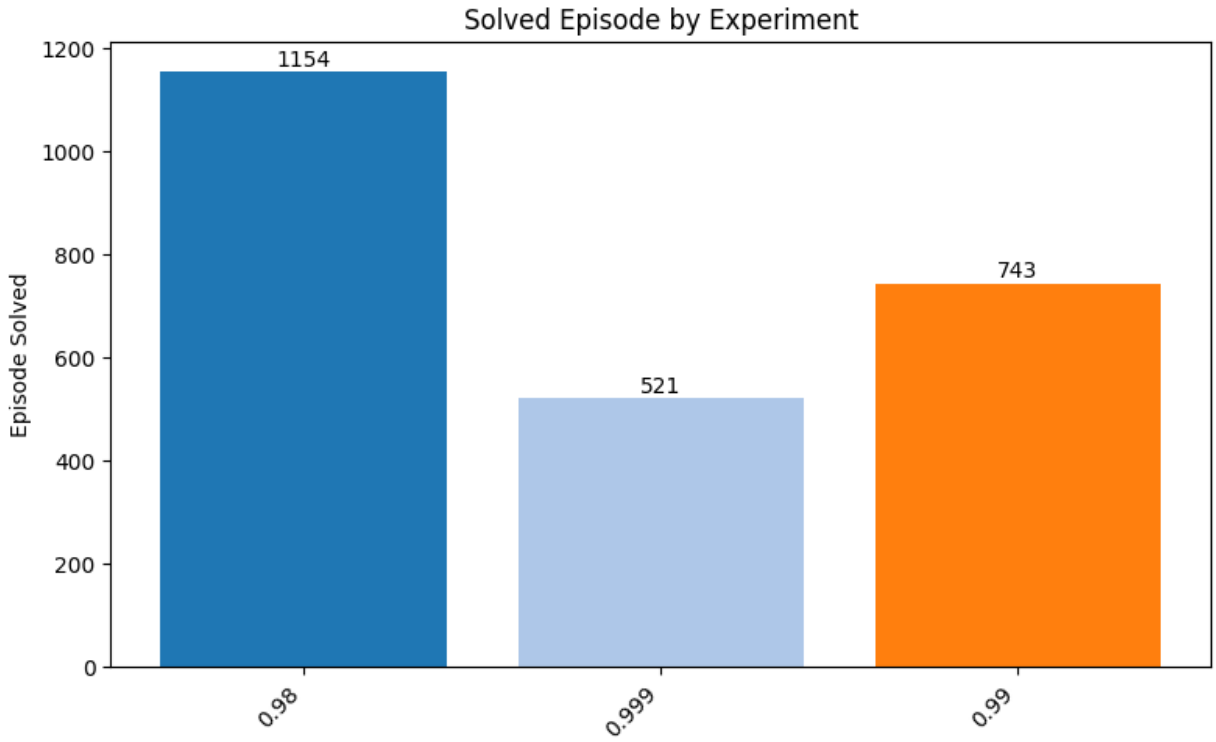

Figure 49: First Solved Episode for Discount Factor

Exploration Decay Rate (ε-decay):

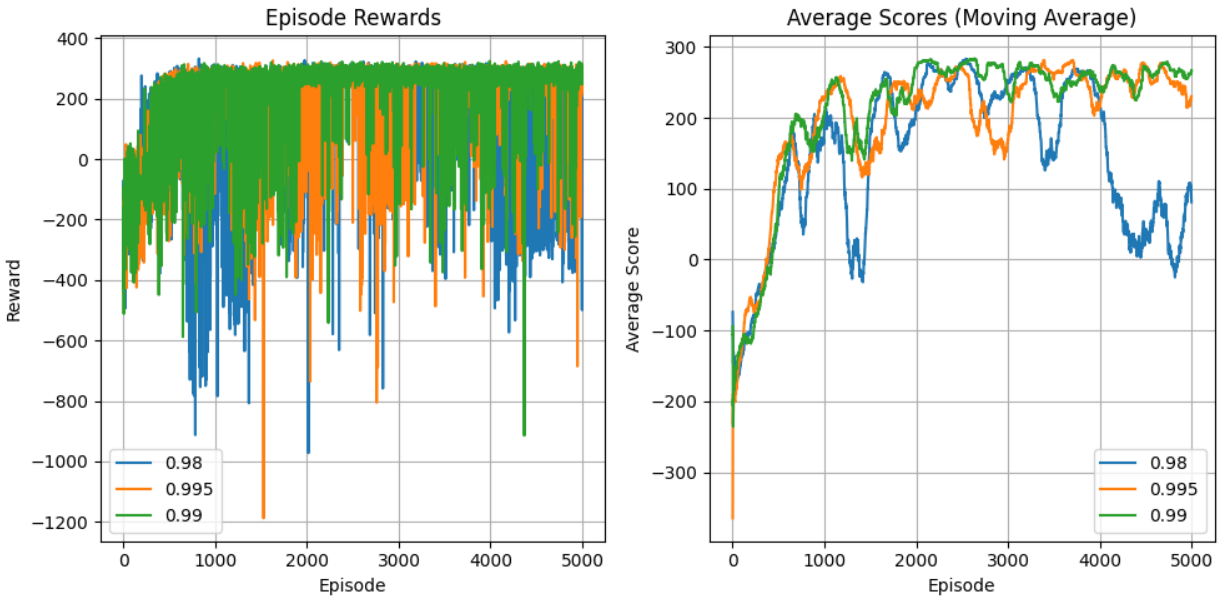

Figure 50: Learning Curve Plot for Exploration Decay Rate

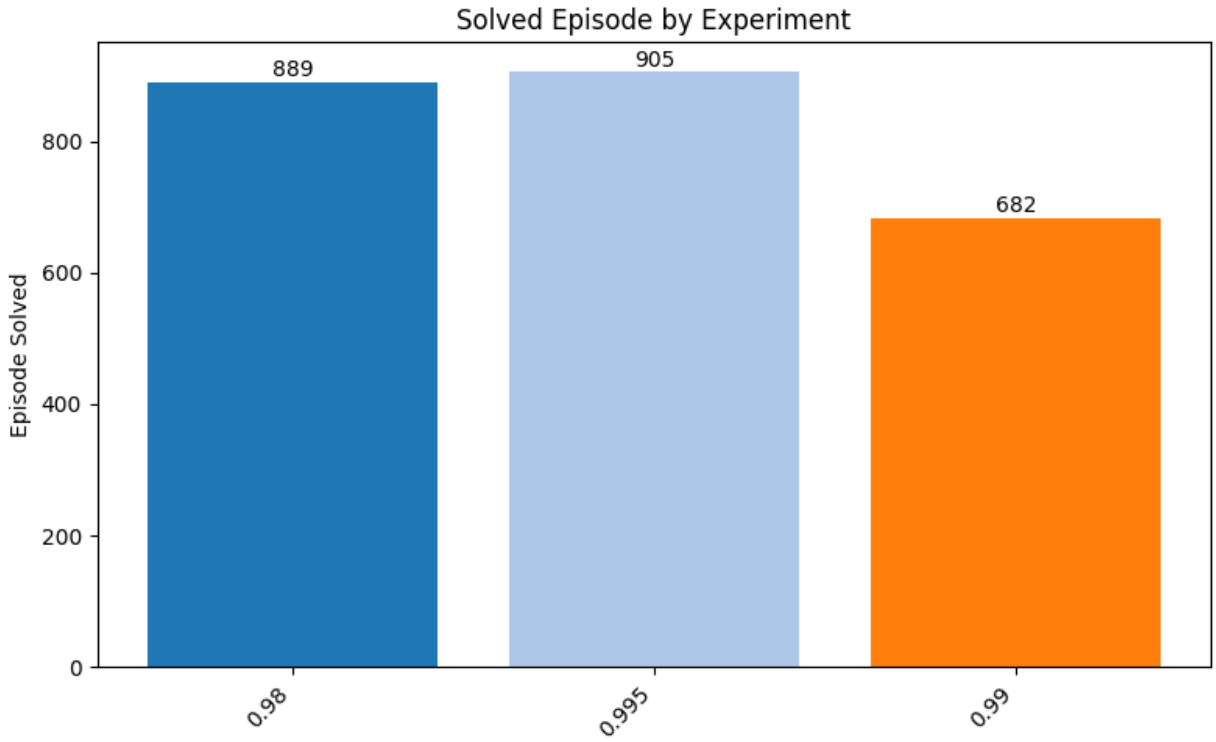

Figure 51: First Solved Episode for Exploration Decay Rate

Target Network Update Frequency (f):

Figure 52: Learning Curve Plot for Target Network Update Frequency

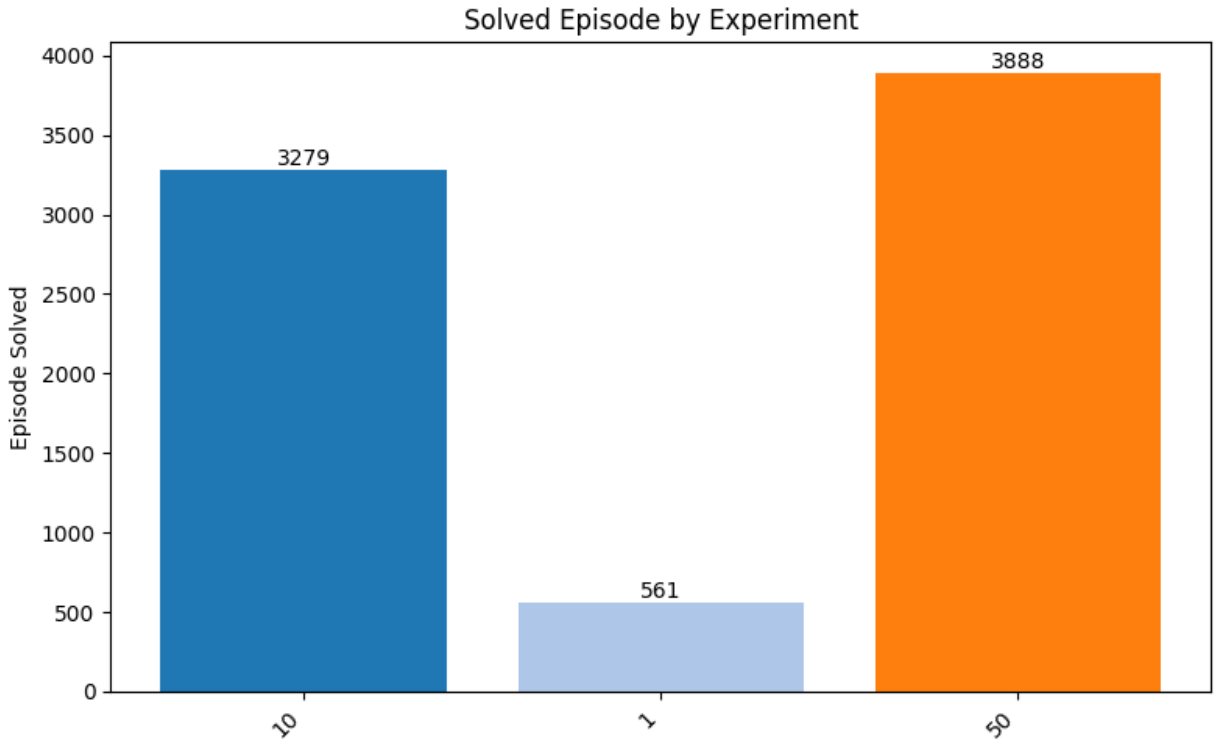

Figure 53: First Solved Episode for Target Network Update Frequency

Q-Network Architecture (Net):

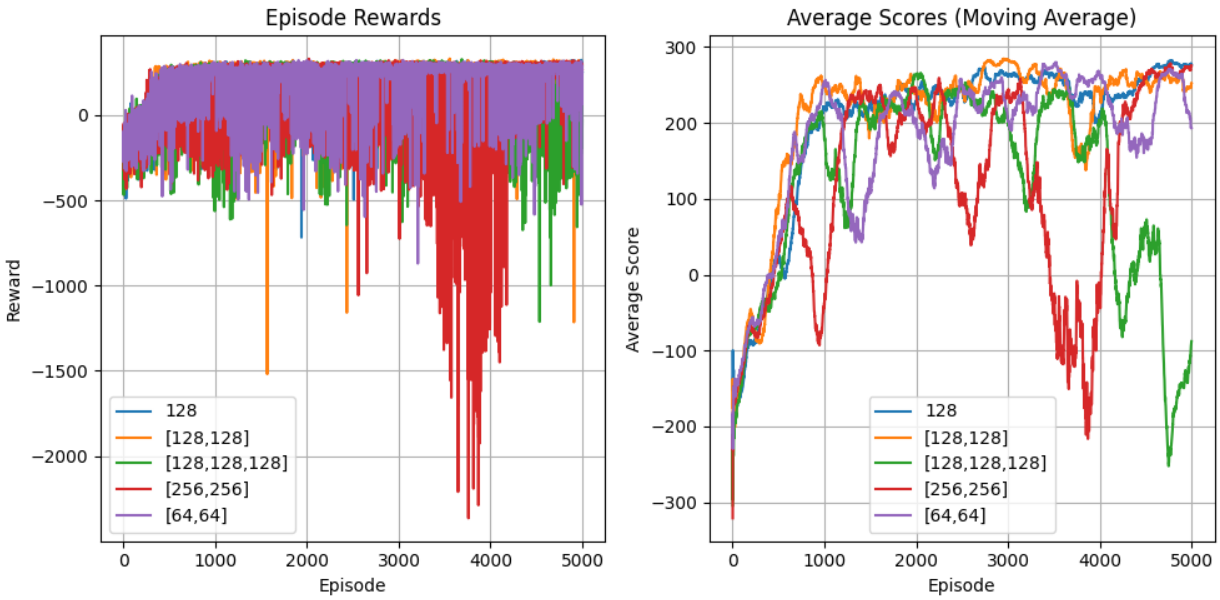

Figure 54: Learning Curve Plot for Q-Network Architecture

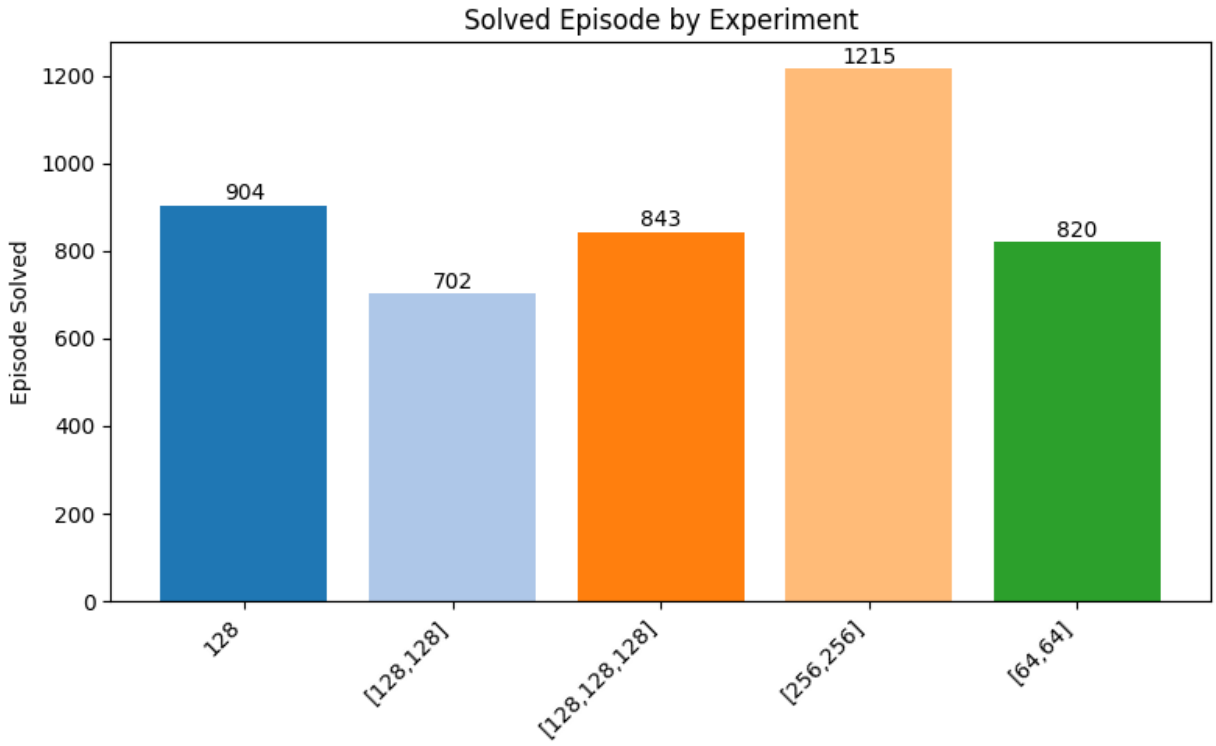

Figure 55: First Solved Episode for Q-Network Architecture

2.4: Discussion

1. How did the three learning rates impact the agent's learning speed and stability? For example, did a higher learning rate converge faster or overshoot (causing instability)? Which learning rate achieved the best final performance, and what trade-offs did you notice (e.g. overshooting vs. slow learning)?

Learning rate	Episode where 100-episode moving average first reached ≥ 200 ("solved")	Long-run trend of the moving-average curve	Stability of raw rewards	Overall verdict
1×10^{-4}	**794**	Plateaus at ~ 260–270 and remains flat	Occasional shallow dips only	**Very stable but slowest to solve**
1×10^{-3}	**728** (≈ 8 % faster)	Peaks ≈ 250, two large drops (≈ 2 k & 3.5 k) then recovers to ~ 250	Deep negative spikes to –1 000 … –2 000	**Fastest early learning but oscillatory**
5×10^{-3}	*Never*	Fluctuates between –400 and –50, no upward trend	Frequent crashes to –3 000 … –3 500	**Divergent — fails to learn**

- **Higher LR accelerates early progress**
  - 1×10^{-3} reaches the solve threshold ~ 70 episodes sooner than 1×10^{-4}.
  - 5×10^{-3} destabilizes learning so much that the agent never solves.
- **Trade-off between speed and stability**
  - **1×10^{-4}** → smaller updates → slower climb but rock-solid convergence.
  - **1×10^{-3}** → larger updates → quick initial gains **and** periodic overshoot/ recovery cycles.
  - **5×10^{-3}** → updates so large that Q-values oscillate wildly, and the policy cannot stabilize.

Conclusion

- **Best final performance:** 1×10^{-4} (highest, smoothest plateau).
- **Fastest time-to-solve:** 1×10^{-3}, but with risk of temporary collapses.
- **Too aggressive:** 5×10^{-3} overshoots the optimum and never recovers.

Practical choice: use **1×10^{-3}** if training time is critical **and** you monitor for divergence (or pair it with additional stabilizers); otherwise, **1×10^{-4}** is the safer default.

2. Compare the results for different γ values. How did a lower γ (favoring short-term rewards) differ from a higher γ (favoring long-term rewards) in terms of convergence and final scores? Which γ led to solving the environment fastest, and why do you think that is?

γ value	Episode where 100-episode moving average first reached ≥ 200 ("solved")	Behaviour of the moving-average curve	Overall judgement
0.98	1 154 (slowest)	Smooth climb; stabilises around 220 – 230 and stays there	**Very stable**, but slower to learn
0.99	**743**	Reaches ≈ 250; stays high with only minor dips	**Best balance** of speed and stability
0.999	**521 (fastest)**	Peaks ≈ 260, then collapses into a long valley below –1 000 before partial recovery	**Highly unstable**; worst final score

- **Lower γ (0.98)**
  - Emphasises short-term rewards → smaller TD-targets and gradients.
  - Learning remains steady with no dramatic swings, but discovering the full landing sequence takes longer.
- **Intermediate γ (0.99)**
  - Extends the planning horizon just enough to speed up discovery of a good policy while keeping targets well-behaved.
  - Delivers both a quick solve (episode 743) **and** a robust final average around 230 – 250.
- **Very high γ (0.999)**
  - Strong focus on distant future rewards makes Q-targets large and highly sensitive to small prediction errors.
  - Combined with the existing learning-rate, this causes weight overshoot: the agent solves quickly (episode 521) but later diverges, producing the deep negative valley seen in the plots.

Conclusion

- **Fastest initial convergence:** γ = 0.999, but the policy later becomes unstable.
- **Most reliable long-term performance:** γ = 0.99 (quick enough to solve and maintains high rewards).
- **Safest and Best Performance but slowest:** γ = 0.98, suitable when absolute stability is more important than training speed.

3. Examine the curves for the different exploration (ε) decay schemes. How did the rate at which exploration was reduced affect the agent's ability to find a good policy? Did a slower decay (exploring for longer) result in better final performance, or was a faster decay sufficient? Discuss the balance between exploration and exploitation you observed.

Effect of ε-decay rate on exploration ↔ exploitation

ε-decay	Episode solved	Long-run average-score trend	Behavioural takeaway
0.98 (slowest decay → longest exploration)	889	Climbs to ≈ 260, then drifts down to ≈ 120 after episode 3500	*Too much exploration*: random thrusts keep degrading a once-good policy.
0.995 (fastest decay)	905	Peaks near 280; settles around 250 with small ripples	*Exploration shuts off early*: learns a solid policy but needs more episodes before discovery.
0.99 (middle ground)	**682 (fastest)**	Rises to ≈ 260, small dips, finishes ≈ 250	*Best balance*: enough early exploration to find a good policy, then shifts to exploitation before random actions become harmful.

- **Slower decay (ε = 0.98)** keeps the agent exploring well past the point where a workable landing sequence is known.
  Result: training solves the task, but continued randomness injects crashes and drags the moving average down in late episodes.
- **Faster decay (ε = 0.995)** shuts exploration off quickly. The agent spends more time exploiting an emerging policy, so convergence is smooth and the final score is high and stable. The price is a longer "discovery" phase—solve time is ~200 episodes slower than the 0.99 run.
- **Moderate decay (ε = 0.99)** achieves the best of both worlds: it maintains high exploration just long enough to stumble upon a good strategy (solving by episode 682) and then exploits aggressively, holding a top-tier average without the late-episode drop seen at ε = 0.98.

Conclusion

Too-slow decay lets exploration noise erode performance; too-fast decay delays discovery but can still converge; a mid-range decay (0.99 here) delivers the quickest solve **and** a strong, stable final policy.

4. What differences do you see between frequent vs. infrequent target network updates? Did updating the target network more often lead to more stable learning or quicker convergence, or did it cause more oscillation? Explain how the update frequency might be affecting the stability of Q-value estimates.

Update every f steps	Episode solved	Shape of moving-average curve	Stability insight
1 (update **every step**)	**561 → fastest**	Shoots to ≈ 260 by episode 300, then a dramatic plunge below –500 at ≈ 4 000 before partial recovery	**Most oscillatory** – policy net chases a constantly moving target, so Q-values overshoot whenever a big TD-error appears.
10 (baseline value)	3 279	Very slow climb; multiple deep valleys (worst at ≈ 2 100) before finally breaking 200	Still unstable and now **slow** because each overshoot must be re-corrected ten times before the target changes.
50 (update rarely)	3 888 (slowest)	Smooth, monotonic rise to ≈ 230; no catastrophic drops	**Most stable** – a fixed target lets the policy network converge toward a coherent value surface, but progress is gradual.

- **Frequent updates (f = 1)**
  - Target net almost equals policy net ⇒ every big gradient immediately shifts the target, so the policy is always "chasing its own tail."
  - Result: quickest early success, **but** highly susceptible to sudden divergence once value estimates grow large.
- **Moderate updates (f = 10)**
  - Still frequent enough that the target moves before the policy has fully minimised the last TD-error, producing repeated overshoot–correction cycles.
  - Neither fast **nor** stable in this run.
- **Infrequent updates (f = 50)**
  - Target remains static for long stretches, giving the policy network time to reduce TD-error w.r.t. a consistent objective.
  - Learning is slower to reach the solve threshold, yet the final policy is robust and free of large collapses.

Conclusion

The target network provides a **fixed reference** for Q-targets.

- **Too frequent** updates ⇒ the reference moves with each gradient step, inflating variance of Q-value estimates and causing oscillation.
- **Too infrequent** updates ⇒ targets become stale, so gradient directions are accurate but point to an outdated optimum, slowing convergence.
- In this task, a **slow update (f ≳ 50)** yielded the best long-term stability, while **f = 1** maximized speed at the cost of late-stage instability.

5. How did the various neural network architectures (different hidden layer sizes/counts) perform relative to each other? Did a larger network (more neurons or layers) learn faster or achieve higher rewards than a smaller network? Consider the potential reasons, such as function approximation capability vs. overfitting or learning complexity. Which architecture seems to hit the solve criteria fastest, and which struggled?

Architecture (hidden layers)	Episode solved	Shape of moving-average curve	Take-away
128 (neurons) × 1 layer	904	Smooth rise; plateaus ≈ 260	Baseline capacity – learns reliably but not the fastest.
[64 , 64]	820	Slightly noisier than 128-single; final avg ≈ 250	A bit more expressive than one wide layer; converges a little sooner.
[128 , 128]	**702 (fastest)**	Peaks ≈ 280; minor dips but finishes high	Extra layer gives enough capacity to model the lander dynamics yet remains easy to optimise – best speed / quality trade-off.
[128 , 128 , 128]	843	Good early climb, then a sharp collapse near episode 3 600 before recovery	Depth = 3 makes optimisation harder; network temporarily diverges.
[256 , 256]	**1 215 (slowest)**	Very noisy rewards; moving avg hovers 150–200 until late	Large weight count → bigger gradients & over-fitting to early replay samples; slows stable convergence.

- **Moderate width + two layers (128,128) is the sweet-spot.**
  - Enough representational power to fit the non-linear value surface.
  - Still shallow, so gradients propagate cleanly → quickest reach of the ≥ 200 criterion.
- **Going *deeper* to three layers or *wider* to 256-units increases optimisation difficulty** – larger parameter space, higher gradient variance, greater risk of over-fitting sparse early experiences. Both variants show big negative spikes and need many more episodes to stabilise.
- **Smallest networks (128-single or 64,64) remain stable** – limited capacity keeps updates well-behaved – but they require extra episodes to squeeze out the last 20–30 reward points because they approximate the value function less precisely.

Conclusion

A medium-sized, two-layer network (128-128) solves LunarLander the fastest and finishes with the highest sustained reward, whereas the very large **256-256** architecture struggles most to converge.

6. Among all the hyperparameters you experimented with, which one had the most pronounced effect on the agent's performance? Provide examples from your results (refer to the plots or solved episode counts) to support your reasoning. Also, discuss any interactions you suspect between hyperparameters (for instance, could a certain learning rate work better with a certain epsilon schedule? though you tested them independently, think about possible combinations).

Hyper-parameter with the strongest impact: Target-network update frequency (*f*)

Hyper-param	Solve episode (best → worst)	Range	Key figure(s)
Target update *f*	**561** (*f = 1*) → **3 888** (*f = 50*)	**3 327**	Figure 53
γ	521 → 1 154	633	Figure 49
Learning rate	728 → > 5 000 (5 e-3 never solves)	∞	Figure 47
ε-decay	682 → 905	223	Figure 51
Hidden dims	702 → 1 215	513	Figure 55

- **f = 1** – fastest early success, but a dramatic dive near episode 4 000 (target moves every step → value oscillation).
- **f = 10** – chronic overshoot/undershoot cycles; slow climb.
- **f = 50** – very smooth learning, yet needs > 3 800 episodes to solve.

No other single knob produced such a broad swing **for all tested values**, hence *f* dominates overall performance.

- **High lr × frequent updates**
  lr = 5 e-3 explodes on its own; pairing it with *f = 1* would compound the instability. Large step sizes need **slower** target refresh (e.g., *f ≥ 30*).
- **High γ × high lr**
  γ = 0.999 collapses even at 1 e-3; lowering lr or using a gentler ε-decay would likely stabilise it.
- **Network size × ε-decay**
  The wide [256,256] net benefits from longer exploration (ε-decay = 0.98) to avoid early over-fitting.
- **Recommended combo** (from plots):
  lr ≈ 1 e-4, γ = 0.99, ε-decay = 0.99, f ≈ 30–50, net = [128, 128] — fast enough to solve, yet stable in the late game.

Take-away
Target-update frequency is the primary driver of both **speed** and **stability**; tune it first, then adjust lr, γ, and ε-schedule to fine-balance exploration and gradient size.

7. Look at the shape of the learning curves (the reward trajectories over time). Do some configurations have more volatile learning (lots of spikes and dips) while others are smoother? What might cause those differences in training dynamics? Relate this to the hyperparameter settings (e.g., a high learning rate or infrequent target updates might cause more volatility).

Curve examples	Visual behaviour	Likely hyper-parameter cause
lr = 5 e-3 (learning_rate_curve)	Tall negative spikes (–3 000 to –3 500) and no upward trend	**Step size too large** – weight updates overshoot the TD target; Q-values oscillate.
f = 1 (target_update_freq_curve, orange)	Quick surge to +260, then a huge plunge below –500 around ep 4 000	**Target updated every step** – policy net continually chases a moving target, amplifying any sudden TD-error.
γ = 0.999 (gamma_curve, orange)	Stable until ~ep 2 400, then collapses to –1 200	**High discount factor** inflates Q-targets; small prediction errors become large gradients that destabilise learning.
[256, 256] net (hidden_dims_curve, red)	Repeated deep dips; slow recovery	**Excess capacity** → larger gradients & greater chance of over-fitting early, noisy replay samples.
ε-decay = 0.98 (epsilon_decay_curve, blue)	Late-stage drift from +260 down to +120	**Continued exploration** – random actions after convergence pull the average down.
f = 50 (target_update_freq_curve, green)	Smooth monotonic rise, no major crashes	**Stable target network** lets the policy net converge on a fixed objective before it shifts.
lr = 1 e-4 (learning_rate_curve, blue)	Gentle, ripple-free plateau at ≈ 270	**Small step size** keeps updates well within the local basin; minimal oscillation.

- **Learning rate (α)** – larger α multiplies gradient noise, so any outlier TD-error creates an outsized weight change → spikes.
- **Target-update frequency (f)** – if the target moves too often, the error surface itself shifts each step, turning steady descent into zig-zagging.
- **Discount factor (γ)** – high γ lengthens the effective horizon; value estimates become large, so the same absolute error yields bigger gradients.
- **Network capacity** – deeper / wider nets have more parameters to push in conflicting directions, making them sensitive to replay-buffer noise.
- **Exploration schedule (ε)** – high or lingering ε injects random actions even after the policy has converged, lowering moving-average reward without changing the underlying network.

Bottom line
Spiky curves correspond to **aggressive settings** (high α, f = 1, γ ≈ 1, oversized nets) that magnify gradient variance or keep the learning target in flux. Smoother trajectories arise from **conservative settings** (small α, infrequent exploration, slow target refresh) that give the policy network a stable objective and modest updates.

Appendix 1: Temporal Difference (TD) Learning

```
import numpy as np
import matplotlib.pyplot as plt
from utils import plot_value_snapshots, plot_policy_snapshots
from tqdm import trange

class MazeEnvironment:
    def __init__(self):
        # 0=free, 1=obstacle, 2=trap, 3=goal (11×11 layout)
        self.layout = np.array([
            [0,0,0,0,0,0,0,1,1,1,1],
            [1,0,1,1,0,1,0,0,0,0,1],
            [1,0,1,1,1,1,0,1,1,0,1],
            [1,0,0,0,0,1,0,0,1,0,1],
            [1,1,0,1,0,1,0,1,1,0,0],
            [1,1,0,1,0,0,0,0,1,1,0],
            [1,0,0,1,1,0,1,0,0,0,0],
            [0,0,1,1,1,1,2,0,1,1,0],
            [0,0,0,0,0,0,0,1,0,0,0],
            [1,0,1,0,1,1,0,1,1,1,0],
            [1,1,1,0,2,0,1,0,0,3,0]
        ])
        self.start_pos = (0, 0)
        self.current_pos = self.start_pos
        self.state_penalty = -1
        self.trap_penalty = -100
        self.goal_reward = 100
        # Actions: up, down, left, right
        self.actions = {0:(-1,0), 1:(1,0), 2:(0,-1), 3:(0,1)}
        # Stochastic outcome maps
        self.opposite = {0:1,1:0,2:3,3:2}
        self.perpendicular = {0:[2,3],1:[2,3],2:[0,1],3:[0,1]}

    def reset(self):
        self.current_pos = self.start_pos
        return self.current_pos

    def step(self, action):
        probs = [0.75,0.05,0.10,0.10]
        choices = [action,
                   self.opposite[action],
                   self.perpendicular[action][0],
                   self.perpendicular[action][1]]
```

```
        move = np.random.choice(choices, p=probs)
        dr,dc = self.actions[move]
        r,c = self.current_pos
        nr,nc = r+dr, c+dc
        if not (0<=nr<self.layout.shape[0] and 0<=nc<self.layout.shape[1]) or
self.layout[nr,nc]==1:
            nr,nc = r,c
        self.current_pos = (nr,nc)
        cell = self.layout[nr,nc]
        if cell==2: return (nr,nc), self.trap_penalty, True
        if cell==3: return (nr,nc), self.goal_reward, True
        return (nr,nc), self.state_penalty, False

class MazeTD0(MazeEnvironment):
    def __init__(self, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000):
        super().__init__()
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.episodes = episodes
        self.utility = np.zeros_like(self.layout, dtype=float)
        self.convergence = []
        self.record_eps = [1,50,100,1000,5000,episodes]
        self.snapshots = {}

    def get_valid_actions(self, state):
        r,c = state
        valid = []
        for a,(dr,dc) in self.actions.items():
            nr,nc = r+dr, c+dc
            if 0<=nr<self.layout.shape[0] and 0<=nc<self.layout.shape[1] and
self.layout[nr,nc]!=1:
                valid.append(a)
        return valid or list(self.actions.keys())

    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.get_valid_actions(state))
        best_val = -np.inf
        best_actions = []
        for a in self.get_valid_actions(state):
            r,c = state
            dr,dc = self.actions[a]
            nr,nc = r+dr,c+dc
```

```
            if not (0<=nr<self.layout.shape[0] and 0<=nc<self.layout.shape[1]) or
self.layout[nr,nc]==1:
                nr,nc = r,c
            val = self.utility[nr,nc]
            if val>best_val:
                best_val = val
                best_actions = [a]
            elif val==best_val:
                best_actions.append(a)
        return np.random.choice(best_actions)

    def update_value(self, s, reward, s2):
        old = self.utility[s]
        nxt = self.utility[s2]
        self.utility[s] = old + self.alpha*(reward + self.gamma*nxt - old)

    def run_episodes(self, max_steps=None):
        for ep in trange(1, self.episodes+1,
                         desc=f"α={self.alpha} γ={self.gamma} ε={self.epsilon}"):
            s = self.reset()
            done = False
            delta = 0.0
            steps = 0
            while not done and (max_steps is None or steps < max_steps):
                a = self.choose_action(s)
                s2,r,done = self.step(a)
                old = self.utility[s]
                self.update_value(s, r, s2)
                delta += abs(self.utility[s]-old)
                s = s2
                steps += 1
            self.convergence.append(delta)
            if ep in self.record_eps:
                self.snapshots[ep] = self.utility.copy()
        return self.snapshots, self.convergence


def save_convergence(conv, filename, window=100):
    import numpy as _np
    arr = _np.array(conv)
    if len(arr)>=window:
        ma = _np.convolve(arr, _np.ones(window)/window, mode='valid')
    else:
        ma = arr.copy()
    fig, ax = plt.subplots(figsize=(8,4))
```

```
    ax.plot(range(1,len(arr)+1), arr, alpha=0.3, label='raw ΔU')
    ax.plot(range(window, window+len(ma)), ma, color='red', label=f'{window}-
episode MA')
    ax.set(xlabel="Episode", ylabel="Sum of |ΔU|", title="Convergence
(smoothed)")
    ax.legend()
    fig.savefig(filename)
    plt.close(fig)

if __name__ == "__main__":
    # default parameters
    default_alpha = 0.1
    default_gamma = 0.95
    default_epsilon = 0.2
    episodes = 10000

    experiments = []
    # vary alpha
    for a in [0.001, 0.01, 0.1, 0.5, 1.0]:
        experiments.append(("alpha", a, default_gamma, default_epsilon, None))
    # vary gamma (cap steps to 500)
    for g in [0.10, 0.25, 0.50, 0.75, 0.95]:
        experiments.append(("gamma", default_alpha, g, default_epsilon, 500))
    # vary epsilon
    for e in [0.0, 0.2, 0.5, 0.8, 1.0]:
        experiments.append(("epsilon", default_alpha, default_gamma, e, None))

    for name, alpha, gamma, epsilon, cap in experiments:
        prefix = f"{name}_{alpha if name=='alpha' else (gamma if name=='gamma'
else epsilon)}"
        print(f"Running {prefix}, cap={cap}...")
        agent = MazeTD0(alpha=alpha, gamma=gamma, epsilon=epsilon,
episodes=episodes)
        snapshots, conv = agent.run_episodes(max_steps=cap)
        save_convergence(conv, f"{prefix}_convergence.png")
        plot_value_snapshots(snapshots, agent.layout,
filename=f"{prefix}_values.png")
        plot_policy_snapshots(snapshots, agent.layout,
filename=f"{prefix}_policies.png")
```

Appendix 2: Deep Q-Learning

```python
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import random
import json
from collections import deque
from datetime import datetime           #  ← NEW
from utils import plot_learning_curves, plot_solved_episodes
from tqdm import trange

# ──────────────────────────────────────────────
# helpers ──────────────────────────────────────
# ──────────────────────────────────────────────

def _val_to_str(val):
    """
    Convert *any* hyper-parameter value into a clean, filename-safe string.

    • Floats:  0.001  →  0p001
    • Lists:   [128,128]  →  128x128
    • Others:  str(val)
    """
    if isinstance(val, (list, tuple)):
        val_str = "x".join(map(str, val))
    else:
        val_str = str(val)

    # replace characters that do not play well with filenames
    val_str = (
        val_str.replace(" ", "")
               .replace("[", "")
               .replace("]", "")
               .replace(",", "x")
               .replace(".", "p")
    )
    return val_str


def _make_results_name(param_name, val):
    """
    Build a UNIQUE results filename.
```

```
    Adds a UTC timestamp so running the same sweep twice won't overwrite
    the previous JSON.
    """
    safe_val = _val_to_str(val)
    ts       = datetime.utcnow().strftime("%Y%m%dT%H%M%SZ")
    return f"results_{param_name}_{safe_val}_{ts}.json"


# ──────────────────────────────────────────────
# neural net / replay buffer ──────────────────
# ──────────────────────────────────────────────

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dims):
        super().__init__()
        layers = []
        in_dim = state_dim
        for h in hidden_dims:
            layers.append(nn.Linear(in_dim, h))
            layers.append(nn.ReLU())
            in_dim = h
        layers.append(nn.Linear(in_dim, action_dim))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)


class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer   = []
        self.pos      = 0

    def push(self, s, a, r, s2, d):
        if len(self.buffer) < self.capacity:
            self.buffer.append((s, a, r, s2, d))
        else:
            self.buffer[self.pos] = (s, a, r, s2, d)
        self.pos = (self.pos + 1) % self.capacity
```

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

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


# ──────────────────────────────────────────────
# DQN agent ────────────────────────────────────
# ──────────────────────────────────────────────

class DQNAgent:
    def __init__(
        self,
        state_dim,
        action_dim,
        memory_size        = 50_000,
        batch_size         = 64,
        gamma              = 0.99,
        lr                 = 1e-3,
        epsilon_start      = 1.0,
        epsilon_min        = 0.01,
        epsilon_decay      = 0.995,
        target_update_freq = 10,
        hidden_dims        = [128, 128],
    ):
        self.action_dim = action_dim
        self.gamma      = gamma
        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, hidden_dims).to(device)
        self.target_net = QNetwork(state_dim, action_dim, hidden_dims).to(device)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)
        self.target_update_freq = target_update_freq
        self.solved_score       = 200.0

    # ε-greedy
```

```
    def get_action(self, state):
        if random.random() < self.epsilon:
            return random.randrange(self.action_dim)
        state_t = torch.tensor(state, dtype=torch.float32,
device=device).unsqueeze(0)
        with torch.no_grad():
            q_vals = self.policy_net(state_t)
        return int(q_vals.argmax(dim=1).item())

    # one SGD step
    def train_step(self):
        if len(self.memory) < self.batch_size:
            return

        batch       = self.memory.sample(self.batch_size)
        states_np   = np.array([b[0] for b in batch], dtype=np.float32)
        next_np     = np.array([b[3] for b in batch], dtype=np.float32)

        states      = torch.from_numpy(states_np).to(device)
        next_states = torch.from_numpy(next_np).to(device)

        actions = torch.tensor([b[1] for b in batch], dtype=torch.int64,
                               device=device).unsqueeze(1)
        rewards = torch.tensor([b[2] for b in batch], dtype=torch.float32,
                               device=device).unsqueeze(1)
        dones   = torch.tensor([b[4] for b in batch], dtype=torch.float32,
                               device=device).unsqueeze(1)

        q_curr  = self.policy_net(states).gather(1, actions)
        q_next  = self.target_net(next_states).max(1)[0].unsqueeze(1)
        q_targ  = rewards + (1 - dones) * self.gamma * q_next

        loss = nn.MSELoss()(q_curr, q_targ)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    def update_target(self):
        self.target_net.load_state_dict(self.policy_net.state_dict())

    def decay_epsilon(self):
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)


# ————————————————————————————————————————
```

```
# experiment runner ──────────────────────────────
# ────────────────────────────────────────────────

def run_sweep(param_name, param_values, agent_kwargs,
              num_episodes=5000):
    result_paths = []

    for val in param_values:
        # ------------------------------------------------
        # prepare hyper-parameters for this agent
        # ------------------------------------------------
        local_kwargs          = agent_kwargs.copy()
        local_kwargs[param_name] = val
        agent = DQNAgent(state_dim, action_dim, **local_kwargs)

        rewards, avgs = [], []
        window        = deque(maxlen=100)
        solved_ep     = None

        for ep in trange(1, num_episodes + 1,
                         desc=f"{param_name}={val}"):
            state, _ = env.reset()
            total_r  = 0
            done     = False

            while not done:
                a = agent.get_action(state)
                s2, r, term, trunc, _ = env.step(a)
                done = term or trunc
                agent.memory.push(state, a, r, s2, done)
                agent.train_step()
                state   = s2
                total_r += r

            agent.decay_epsilon()
            if ep % agent.target_update_freq == 0:
                agent.update_target()

            rewards.append(total_r)
            window.append(total_r)
            avgs.append(sum(window) / len(window))

            if solved_ep is None and avgs[-1] >= agent.solved_score:
                solved_ep = ep
```

```
        # ----------------------------------------------------
        # record results
        # ----------------------------------------------------
        solved_ep_val = solved_ep
        fname         = _make_results_name(param_name, val)
        with open(fname, "w") as f:
            json.dump(
                {
                    "episode_rewards": rewards,
                    "average_scores":  avgs,
                    "hyperparameters": local_kwargs,
                    "solved_episode":  solved_ep_val,
                },
                f,
            )
        result_paths.append(fname)

    # ----------------------------------------------------
    # plots for this sweep
    # ----------------------------------------------------
    plot_learning_curves(result_paths,
                         output_file=f"{param_name}_learning_curves.png")
    plot_solved_episodes(result_paths,
                         output_file=f"{param_name}_solved_episodes.png")


# ————————————————————————————————————————
# main experiment script ———————————————————
# ————————————————————————————————————————

if __name__ == "__main__":
    env = gym.make("LunarLander-v3")
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n

    # hyper-parameter grids
    lr_list          = [1e-4, 1e-3, 5e-3]
    gamma_list       = [0.98, 0.99, 0.999]
    eps_decay_list   = [0.98, 0.99, 0.995]
    target_freq_list = [1, 10, 50]
    archs            = [[128], [64, 64], [128, 128],
                        [128, 128, 128], [256, 256]]

    # 1) learning-rate sweep
    run_sweep(
```

```
        "lr",
        lr_list,
        {
            "gamma": 0.99,
            "epsilon_decay": 0.995,
            "target_update_freq": 10,
            "hidden_dims": [128, 128],
        },
    )

    # 2) discount-factor sweep
    run_sweep(
        "gamma",
        gamma_list,
        {
            "lr": 1e-3,
            "epsilon_decay": 0.995,
            "target_update_freq": 10,
            "hidden_dims": [128, 128],
        },
    )

    # 3) ε-decay sweep
    run_sweep(
        "epsilon_decay",
        eps_decay_list,
        {
            "lr": 1e-3,
            "gamma": 0.99,
            "target_update_freq": 10,
            "hidden_dims": [128, 128],
        },
    )

    # 4) target-network-frequency sweep
    run_sweep(
        "target_update_freq",
        target_freq_list,
        {
            "lr": 1e-3,
            "gamma": 0.99,
            "epsilon_decay": 0.995,
            "hidden_dims": [128, 128],
        },
    )
```

```
# 5) network-architecture sweep
run_sweep(
    "hidden_dims",
    archs,
    {
        "lr": 1e-3,
        "gamma": 0.99,
        "epsilon_decay": 0.995,
        "target_update_freq": 10,
    },
)

env.close()
```