2025

Reinforcement Learning

FONTYS UNIVERSITY OF APPLIED SCIENCES
ANNA KADURINA

Table of Contents

GitLab

https://git.fhict.nl/I493249/reinforcementlearning.git

FrozenLake with Q-Learning

Introduction

In this report, I explore how a reinforcement learning agent can learn to solve the FrozenLake environment using Q-learning. The goal is to understand how different settings — like hyperparameters, exploration strategies, and reward shaping — affect how well the agent learns to reach the goal. I also test more advanced ideas like injecting successful paths and using experience replay to speed up learning and improve performance. The experiments are done in both deterministic and stochastic environments, and I scale up the difficulty by testing on larger map sizes as well.

FrozenLake is a classic environment from the Gymnasium library. It's a grid-based world where an agent starts at one point (S) and tries to reach the goal (G) by moving up, down, left, or right. The challenge is that some tiles are holes (H) — if the agent steps into one, the episode ends immediately. Depending on the environment setting, the movement can also be slippery, meaning the agent might not go exactly in the direction it chooses. This adds an element of uncertainty, making it a great environment for studying exploration and reward-driven behaviour.

Hyperparameters

Before diving deep into all kinds of fancy strategies and reward tricks, I started off by getting a solid grip on the basics: the hyperparameters. These are the core settings that shape how our Q-learning agent learns.

In this chapter, I explored the effect of tuning the three most important hyperparameters:

- **Alpha (α)** — the learning rate, which controls how quickly the agent updates its Q-values.
- **Gamma (γ)** — the discount factor, which determines how much the agent values future rewards versus immediate ones.
- **Epsilon (ε)** — the exploration rate, which affects how often the agent chooses a random action instead of the best-known one (i.e., the explore-vs-exploit balance).

I tested different values for each of these hyperparameters while keeping the others constant, and ran experiments both in a deterministic environment (where actions always do what you expect) and a stochastic one (slippery=true, where movement is less predictable).

Deterministic Setting

In the deterministic setting, both alpha=0.1 and alpha=0.5 led to successful learning, with higher learning rates helping the agent converge a bit faster. However, with alpha=0.01, the agent barely learned at all — it was too cautious in updating its knowledge.

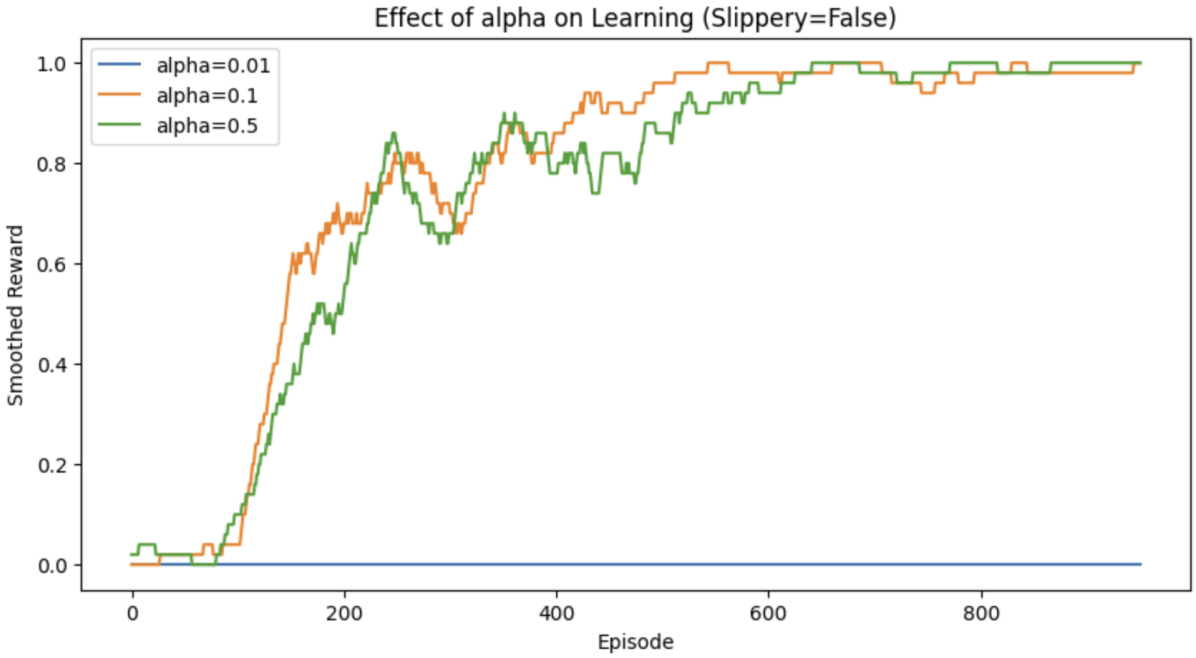

Figure 1 – Effect of alpha on Learning (Slippery=False)

The deterministic tests showed that gamma=0.8 and 0.99 gave the most stable and consistent learning.

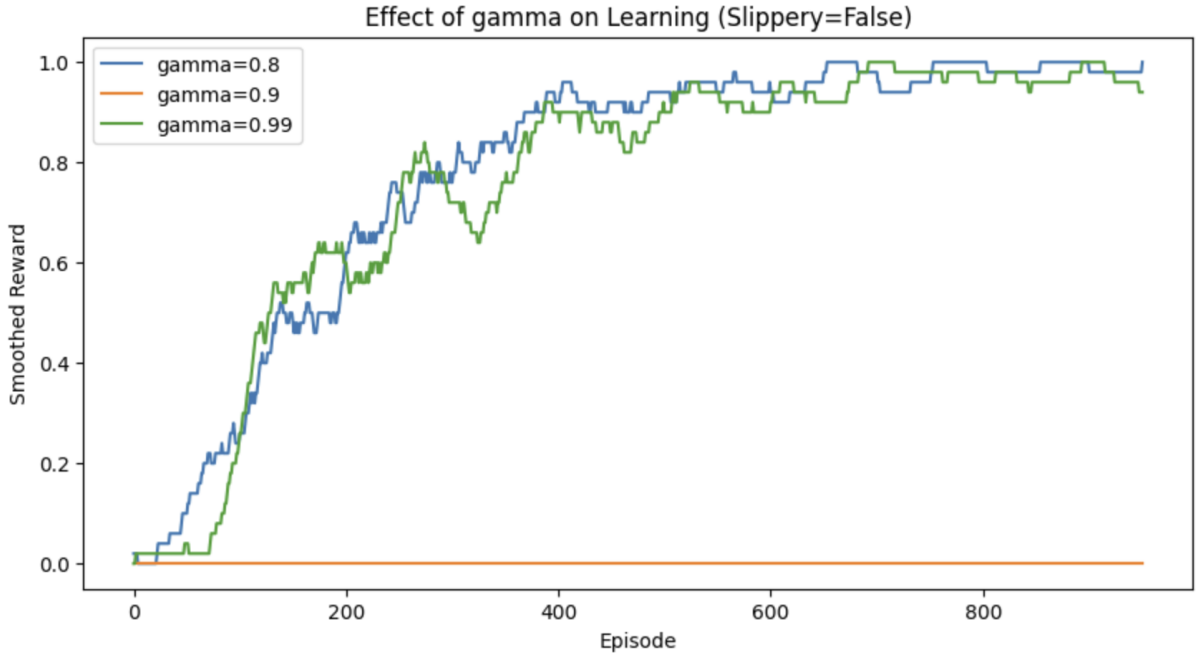

Figure 2 – Effect of gamma on Learning (Slippery=False)

The epsilon was very telling. In deterministic environments, high exploration (epsilon=1.0) allowed the agent to learn efficiently by trying everything, and it steadily improved. Lower exploration (0.1) completely stalled the learning — the agent got stuck doing the wrong thing over and over.

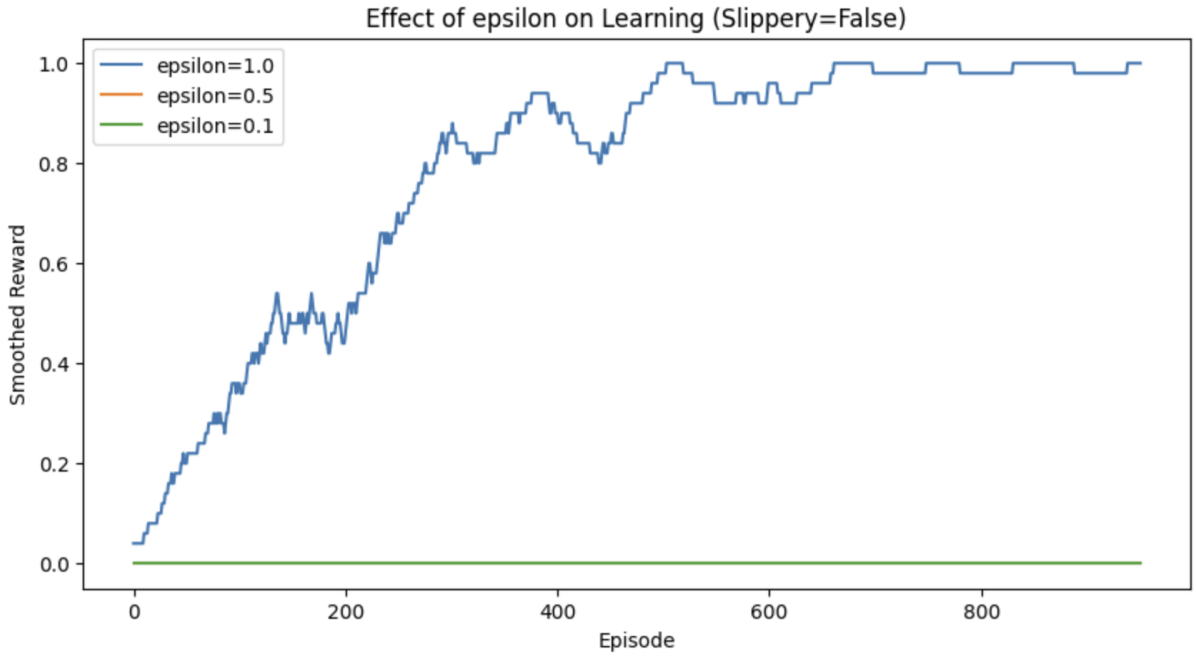

Figure 3 – Effect of epsilon on Learning (Slippery=False)

Stochastic Setting

In the slippery environment, a higher alpha (0.5) was essential to see any progress. Lower alphas led to flat learning curves, meaning the agent couldn't adjust fast enough in a noisy setting.

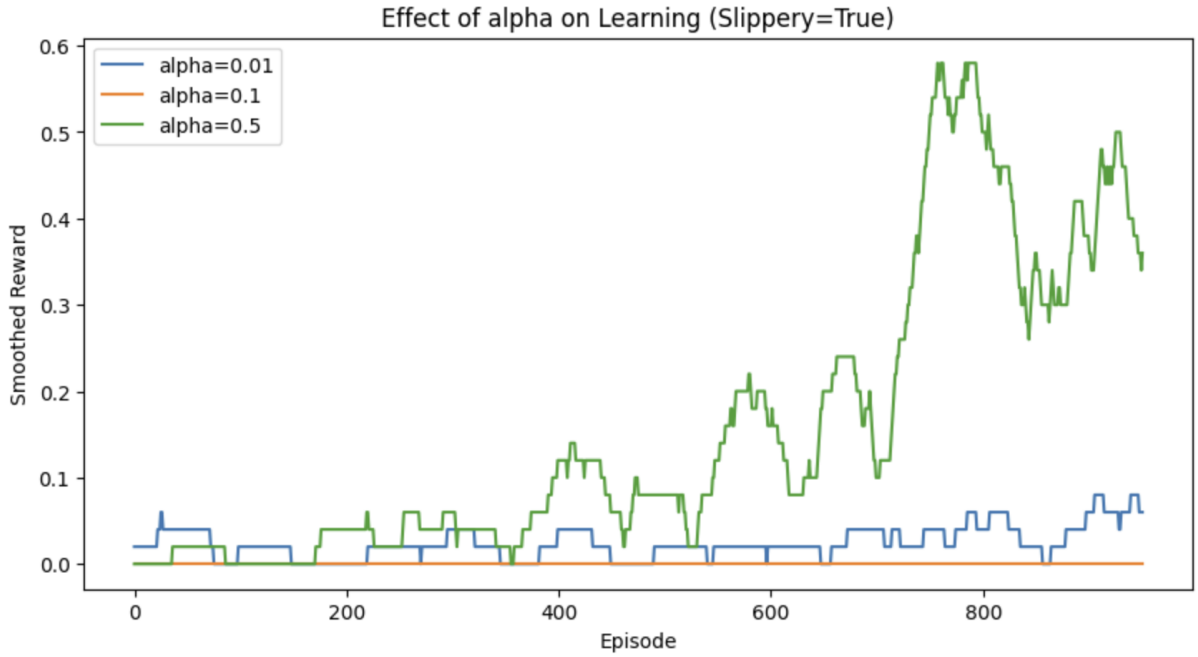

Figure 4 – Effect of alpha on Learning (Slippery=True)

When it is slippery, gamma=0.99 performed best — the agent needed to look further ahead because actions often didn't have immediate predictable effects. Lower gamma values caused it to short-sightedly misjudge situations, especially when uncertainty was involved.

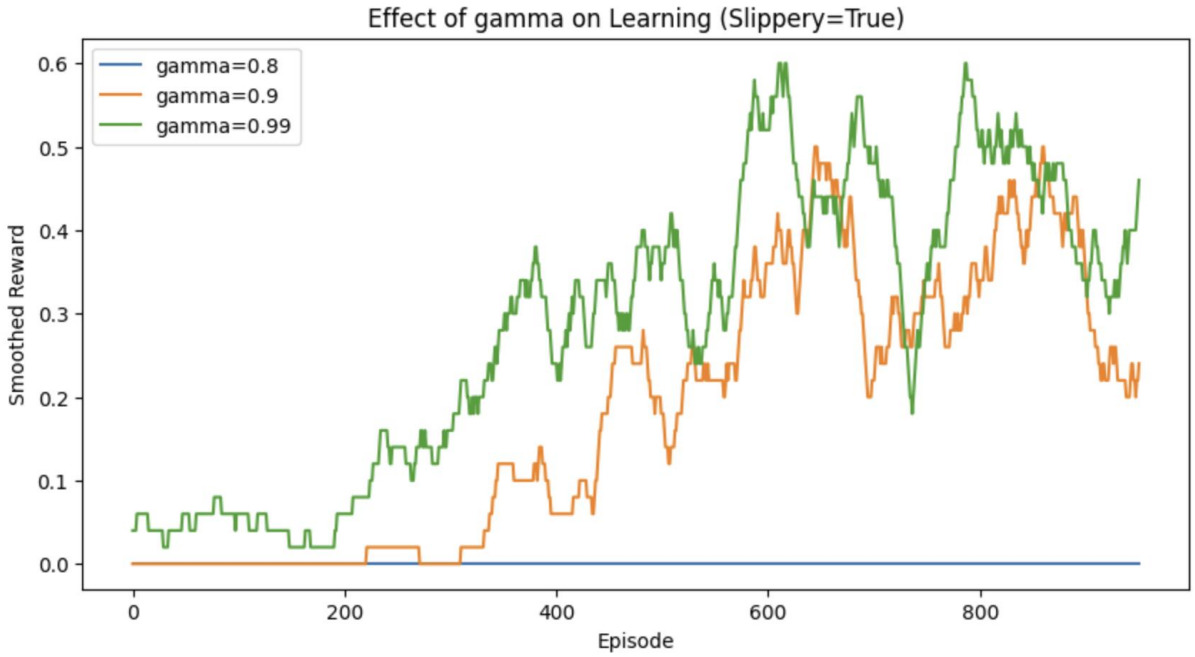

Figure 5 – Effect of gamma on Learning (Slippery=True)

In the slippery case, epsilon=0.5 worked best. It balanced between exploring enough to deal with uncertainty, while still exploiting what it had learned. Both extreme values (1.0 and 0.1) resulted in poor learning — either too random or too repetitive.

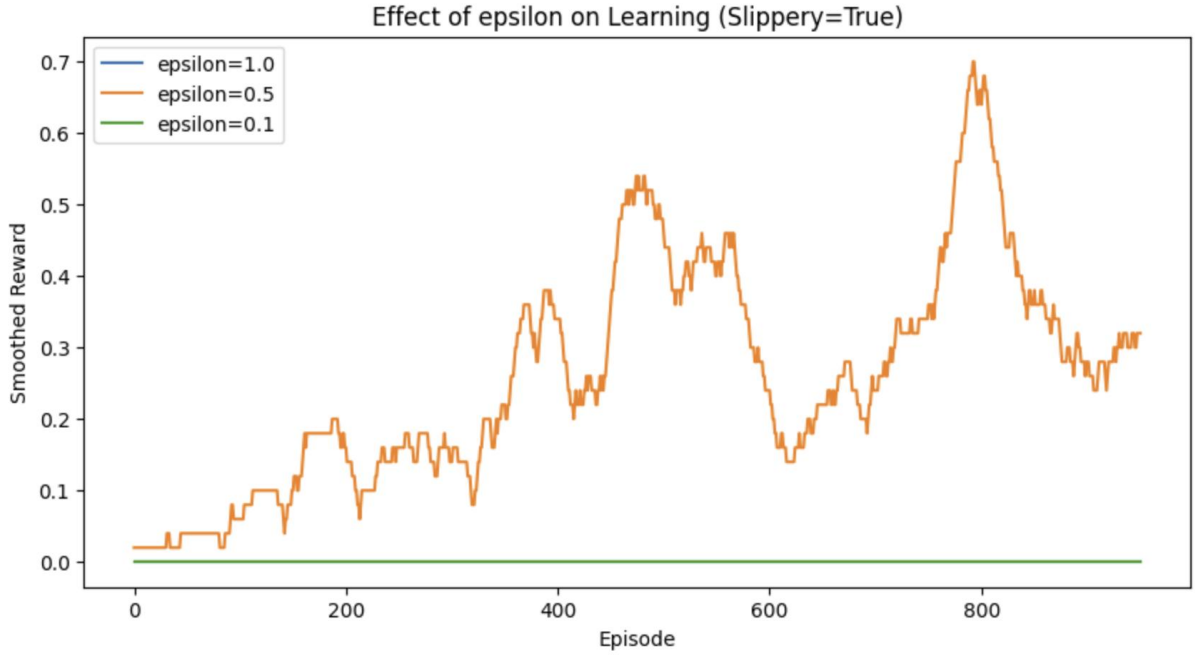

Figure 6 – Effect of epsilon on Learning (Slippery=True)

This chapter highlighted that there's no one-size-fits-all setting — the best hyperparameters depend heavily on the environment. Slippery conditions require more aggressive learning and a cautious balance between exploration and exploitation.

Exploration-Exploitation Trade-off

One of the most important ideas in reinforcement learning is the balance between exploration (trying new actions) and exploitation (sticking to what seems to work). In this part of the project, we wanted to dive deeper into how this trade-off affects learning in the Frozen Lake environment.

Deterministic Setting

In the first experiment, I tested fixed epsilon values (without decay) in a deterministic environment. As shown in the plot, when epsilon was set to 1.0, the agent kept exploring randomly and never really settled on a good policy. With epsilon at 0.1, the agent didn't explore enough and basically got stuck with poor strategies. Interestingly, the epsilon value of 0.5 gave the best results in this setup—it struck a decent balance between exploring and exploiting. This made it clear that while fixed values can work to some extent, they aren't very reliable or flexible.

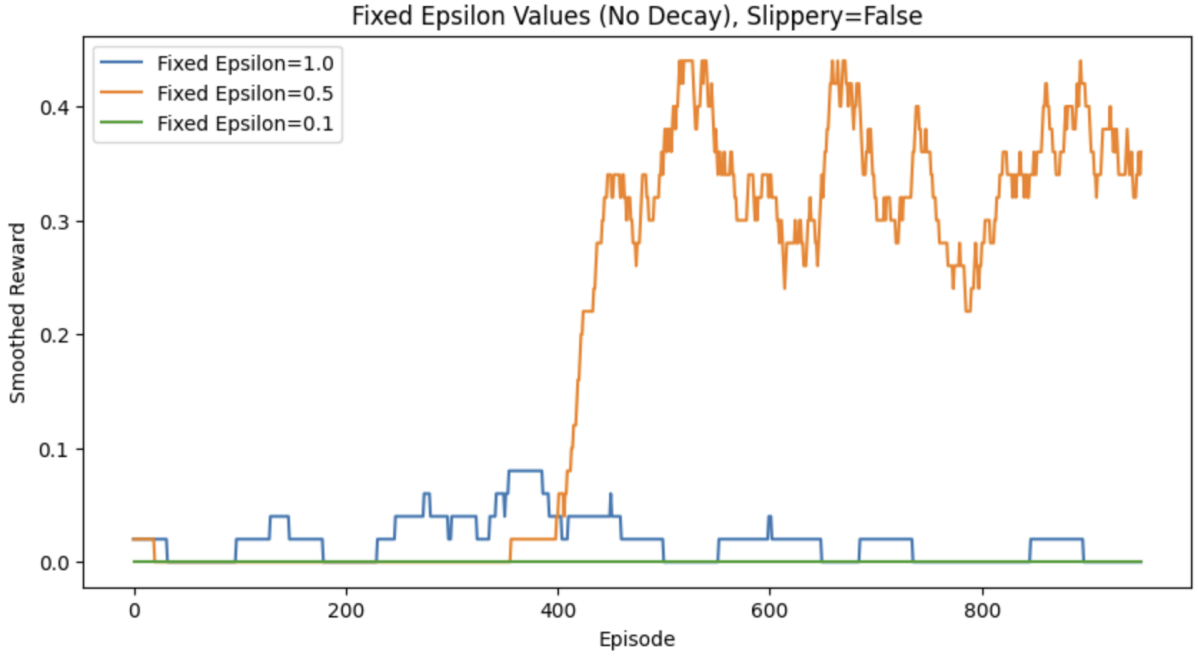

Figure 7 –Fixed Epsilon Values (No Decays), Slippery=False

Next, I moved on to epsilon decay strategies in the deterministic setup. It turned out that a decay of 0.995 gave the best results by far. It allowed the agent to explore thoroughly in the beginning and slowly transition into exploiting what it had learned. A decay of 0.999 was too slow—exploration lasted too long—and 0.97 was too fast, causing the agent to settle early on a suboptimal policy. A decay of 0.99 was in the middle but still less effective than 0.995. This experiment showed that not only should epsilon decay over time, but how fast it decays matters a lot.

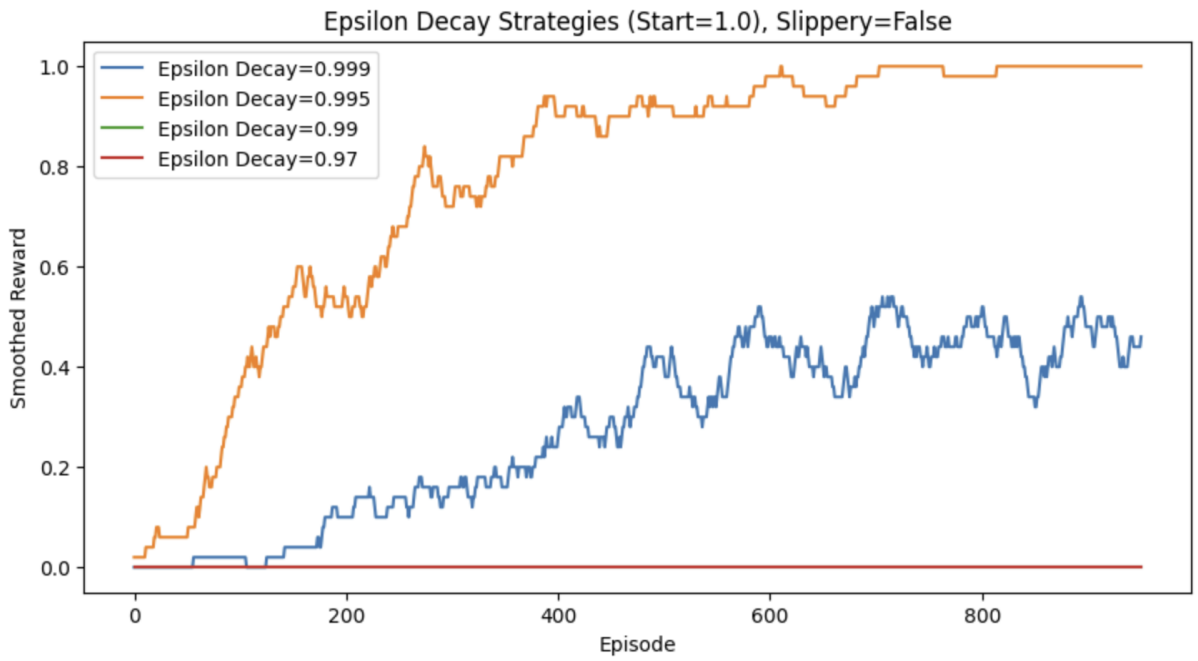

Figure 8 – Epsilon Decay Strategies (Start=1.0), Slippery=False

After testing exponential decay, I compared it with linear decay strategies. In the deterministic environment, linear decay outperformed exponential decay significantly. It decreased epsilon steadily, which gave the agent more predictable control over when to shift from exploration to exploitation.

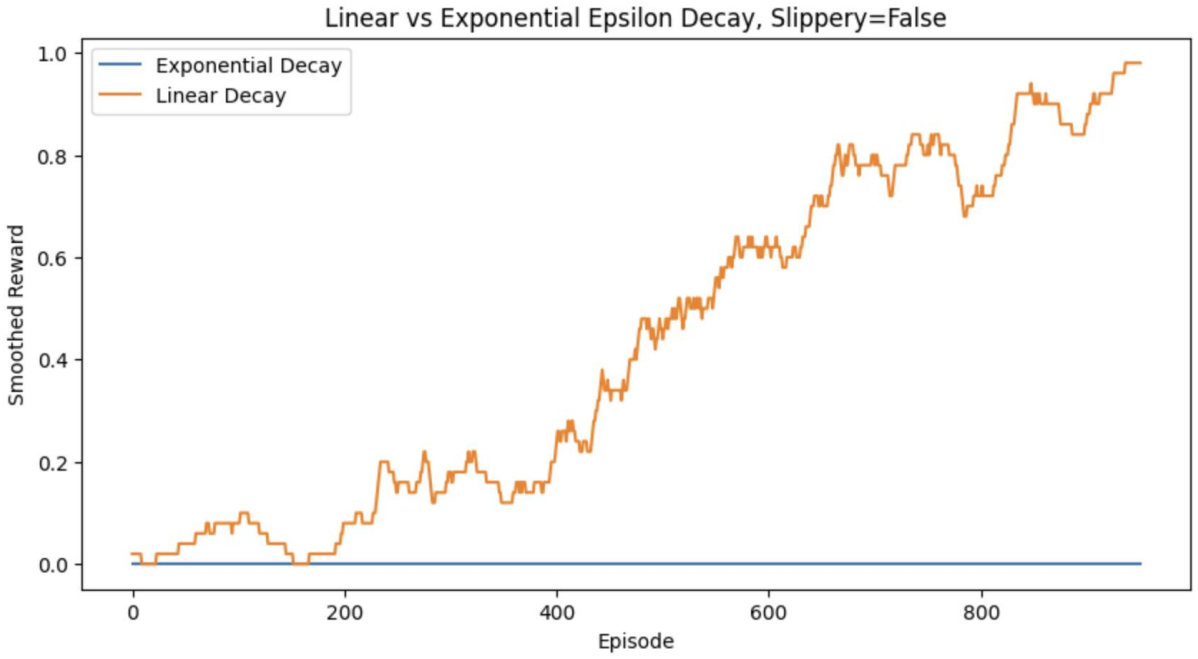

Figure 9 –Linear vs Exponential Epsilon Decay, Slippery=False

Stochastic Setting

As shown in the plot, all fixed epsilon values struggled more in the slippery environment. Since the environment is now stochastic, sticking to one exploration rate just didn't cut it. The agent either explored too much or too little and couldn't adapt to the randomness of the environment.

This emphasized the need for smarter exploration strategies, especially in noisy environments like this.

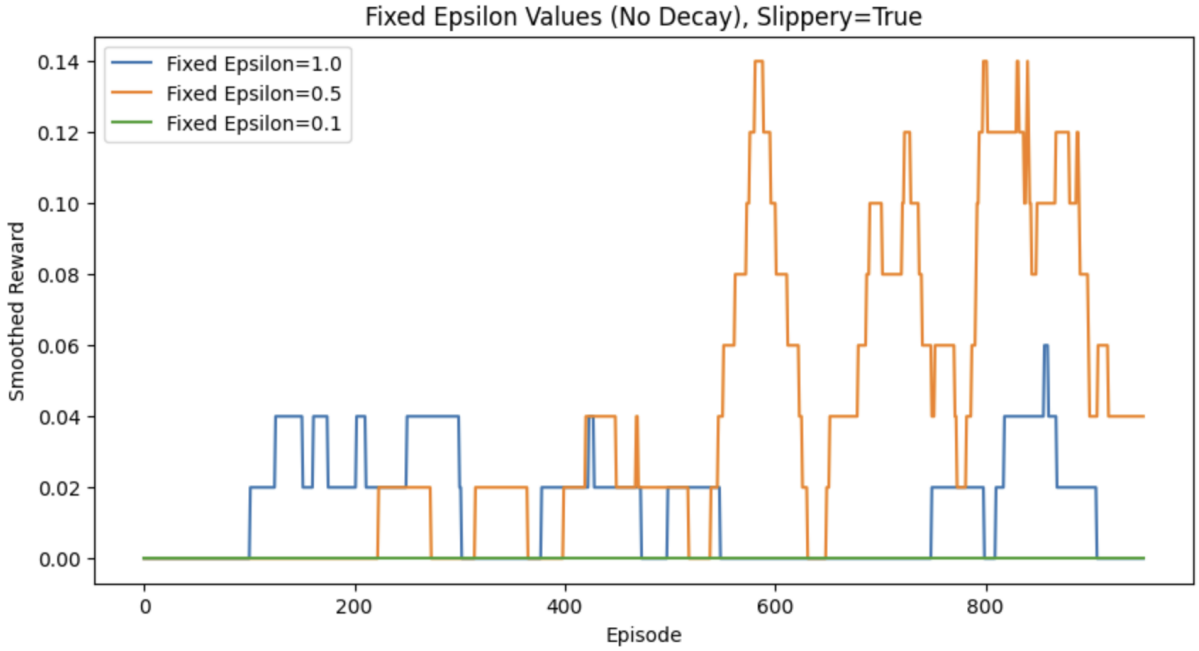

Figure 10 –Fixed Epsilon Values (No Decay), Slippery=True

As seen in the plot below, the decay rate of 0.995 once again gave the most consistent performance. The other decay values, especially 0.97, failed to cope with the randomness of the environment. The agent either became too greedy too fast or kept exploring aimlessly. This confirmed that in both deterministic and stochastic setups, epsilon decay needs to be carefully tuned for optimal performance.

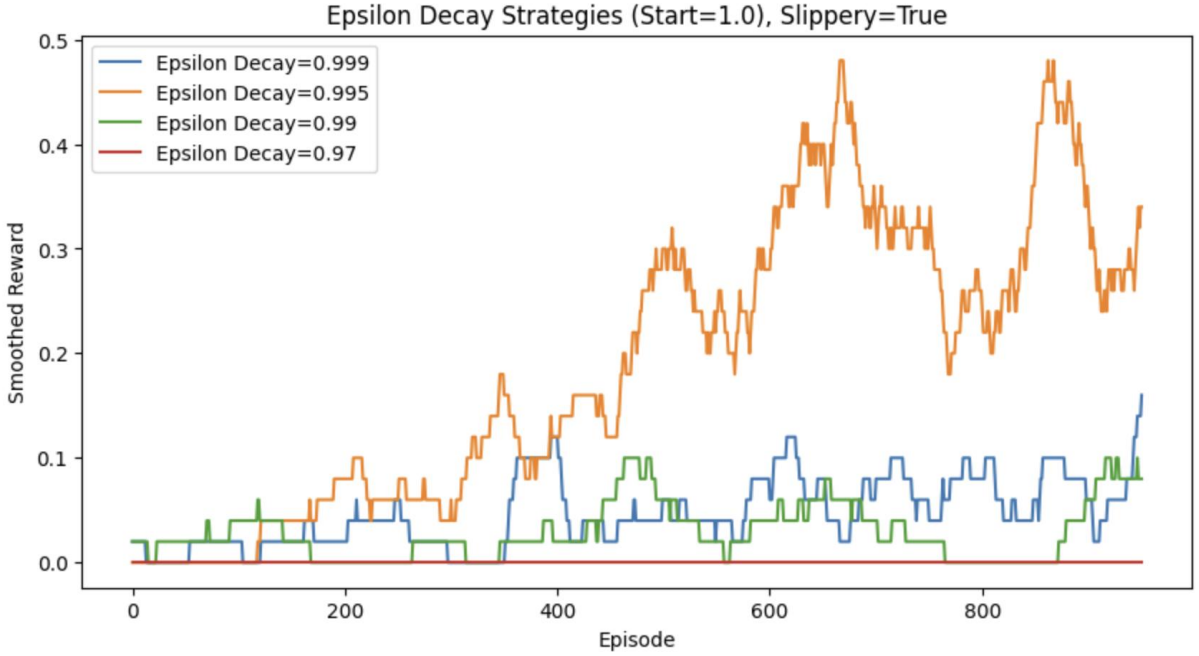

Figure 11 – Epsilon Decay Strategies (Start=1.0), Slippery=True

In the slippery environment, linear decay again showed better performance. It helped the agent adapt to the environment more smoothly, even when randomness made it harder to predict outcomes. So, overall, linear decay proved to be a stronger approach, especially in harder environments.

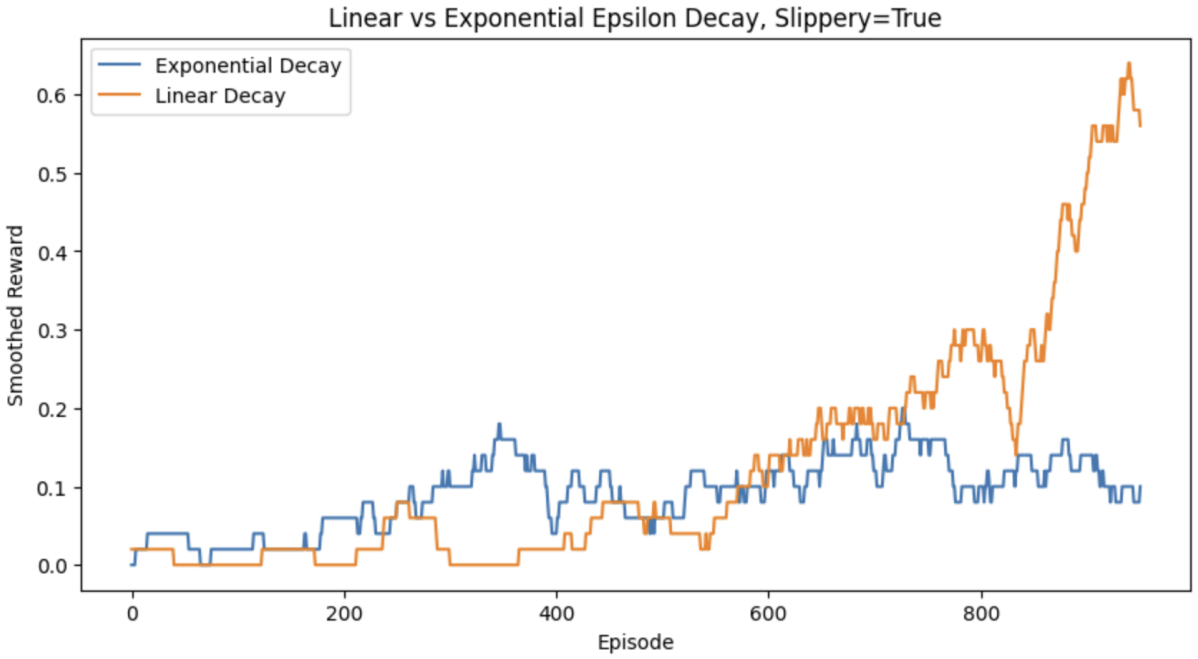

Figure 12 – Linear vs Exponential Epsilon Decay, Slippery=True

Boltzmann vs Epsilon-Greedy

In this part of the project, I wanted to compare two different strategies that guide how the agent balances exploration (trying new actions) and exploitation (using what it already knows). The two strategies I used were the traditional epsilon-greedy approach and the Boltzmann (or softmax) policy. Epsilon-greedy randomly chooses a random action with probability ε and the best-known action otherwise. Boltzmann sampling, on the other hand, uses a probability distribution over actions based on their Q-values and a temperature parameter that controls how "soft" or "sharp" that preference is. Lower temperatures make the agent pick the best actions more deterministically, while higher temperatures make it more exploratory.

Deterministic Setting

In the deterministic version of FrozenLake, the results were very clear. Firstly, I varied the Boltzmann temperature and found that a very low temperature (0.1) performed exceptionally well, reaching a smoothed reward of 1.0 in under 400 episodes. This makes sense because in a stable environment, once the agent finds a good policy, it should stick to it—which is exactly what low temperature does by favoring the action with the highest Q-value.

As temperature increases, the agent becomes more random in its action selection, and the learning process slows down. For example, temperature=2.0 kept the agent stuck in a very exploratory mode, which caused it to perform poorly. The agent never settled on the optimal path and kept exploring, even when it already had a good idea of what to do.

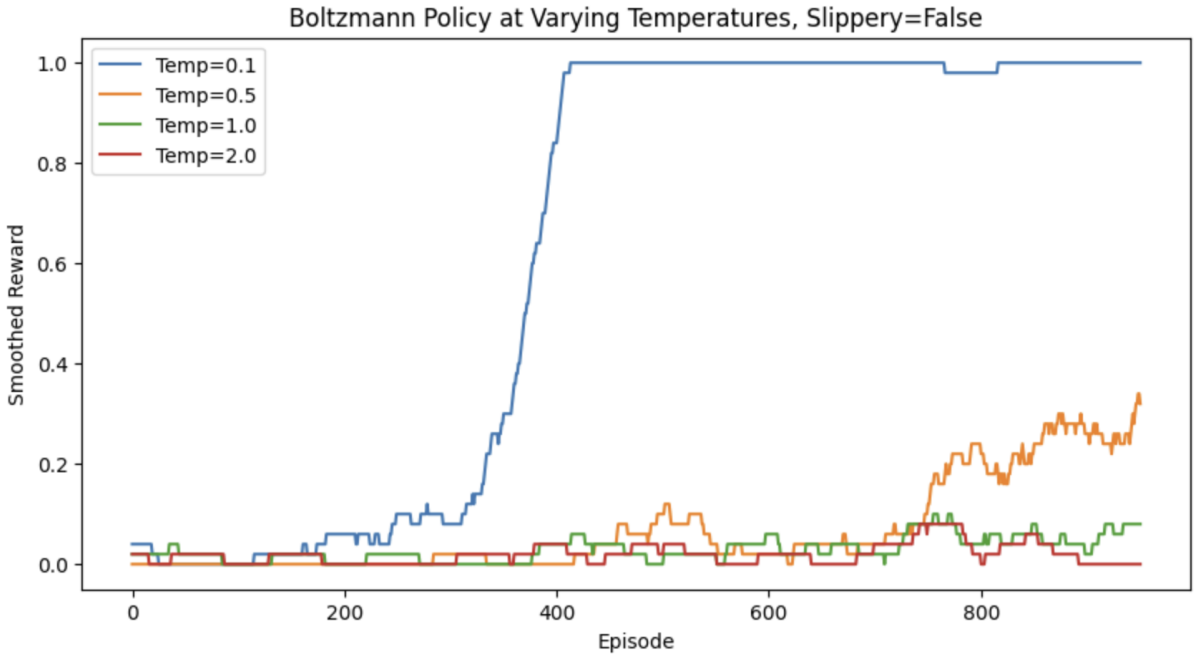

Figure 13 –Boltzmann Policy at Varying Temperatures, Slippery=False

Secondly, I compared Boltzmann (Temp=1.0) with standard epsilon-greedy. Here, epsilon-greedy clearly outperformed Boltzmann in the deterministic setup. The epsilon-greedy agent reached optimal performance quickly and remained consistent, while Boltzmann with Temp=1.0 struggled to converge, even with a decent temperature value. This indicates that Boltzmann may not be the most effective choice for environments where outcomes are predictable and stable.

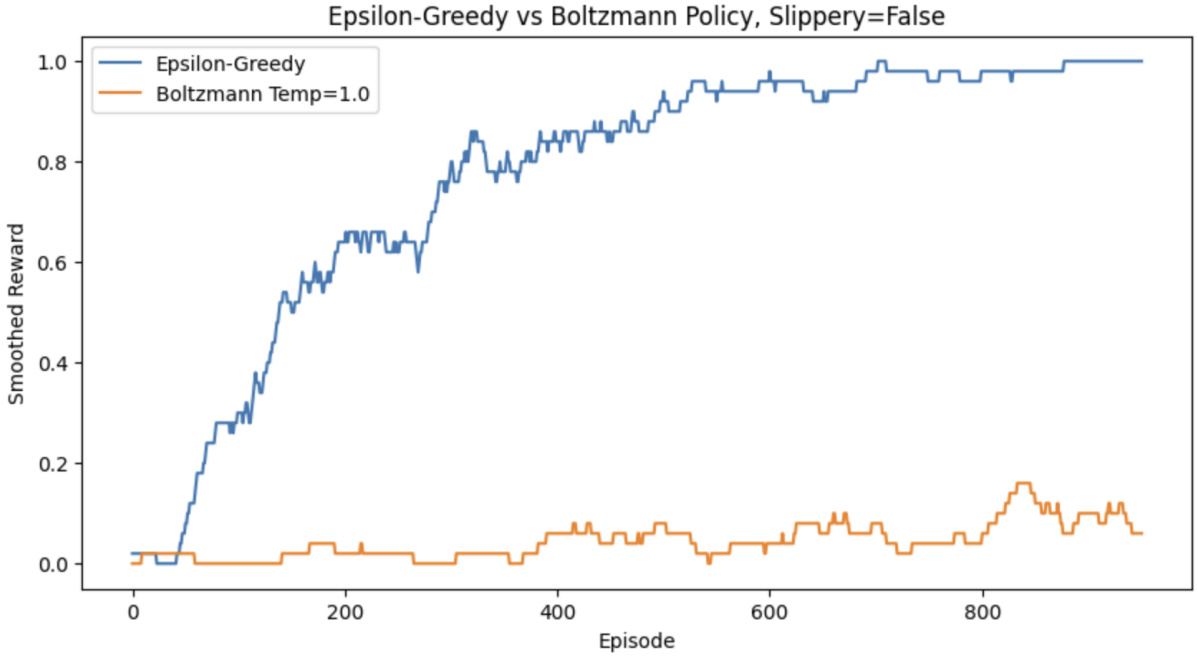

Figure 14 –Epsilon-Greedy vs Boltzmann Policy, Slippery=False

Stochastic Setting

In the stochastic version of FrozenLake, things became more unpredictable. In the Boltzmann temperature experiment, none of the temperature values led to consistent success. The curves were jagged and rewards fluctuated a lot. This reflects the difficulty of learning in an environment where the agent's chosen action doesn't always result in the expected outcome. In such cases, even if the agent picks the best-known action, the randomness in the environment may make it fail anyway.

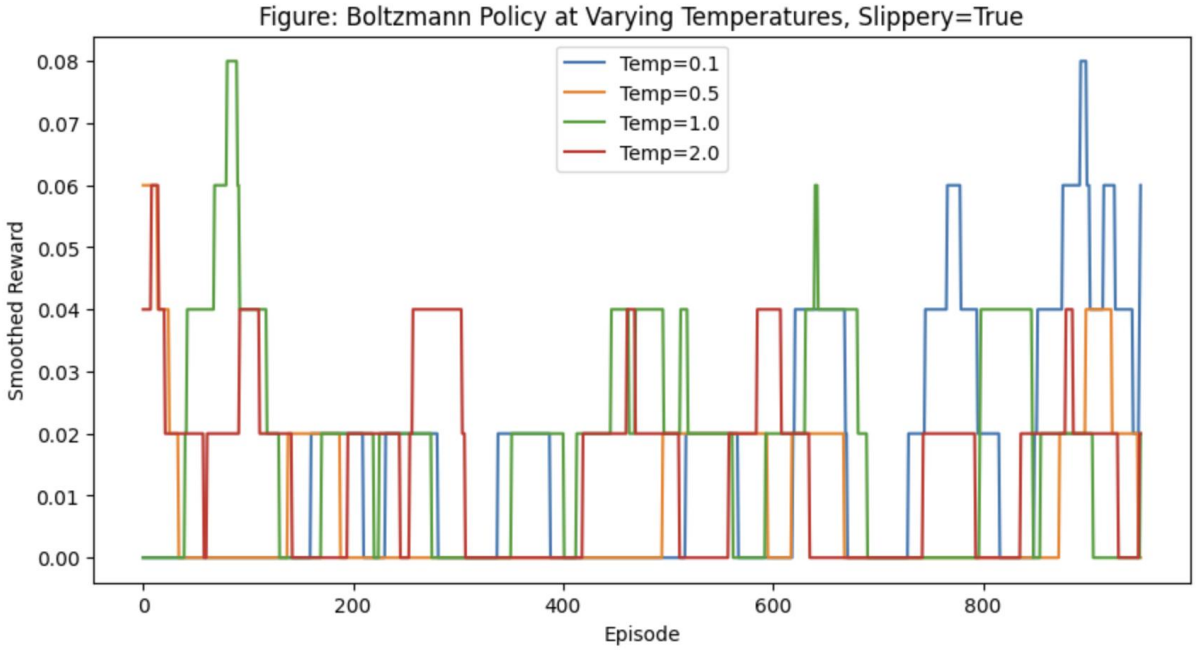

Figure 15 –Boltzmann Policy at Varying Temperatures, Slippery=True

In the comparison between epsilon-greedy and Boltzmann, the epsilon-greedy agent still performed significantly better. It managed to gradually improve its policy over time and eventually reached a smoothed reward above 0.7. In contrast, the Boltzmann agent struggled to make progress and remained stuck below 0.1 in most episodes. This highlights that in slippery environments, structured decay (like with epsilon-greedy) gives the agent a better shot at adapting its behavior over time, compared to Boltzmann's more probabilistic and soft decision-making.

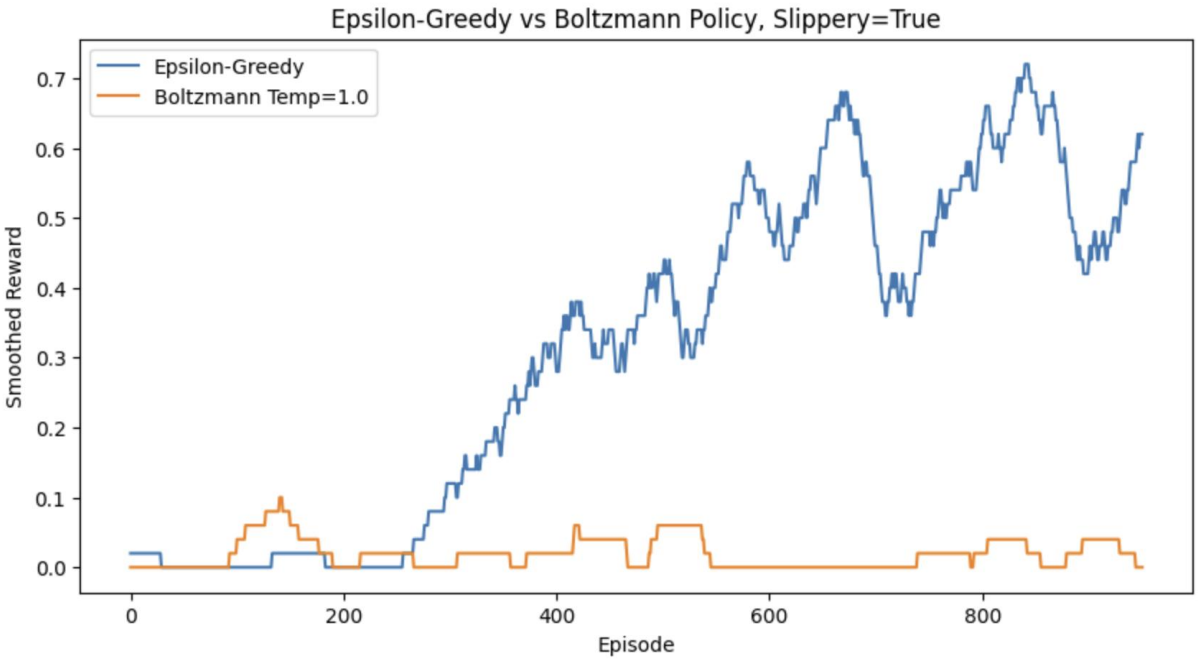

Figure 16 –Epsilon-Greedy vs Boltzmann Policy, Slippery=True

Policy Evaluation

After experimenting with various hyperparameters and exploration strategies, I trained two agents using the best settings discovered in earlier chapters. One agent was trained in the deterministic version of FrozenLake (slippery=False) and the other in the stochastic version (slippery=True). In both cases, I used the optimal combination of learning rate, discount factor, and epsilon decay for epsilon-greedy exploration, which consistently outperformed other configurations.

```python
best_config_d = {
    "alpha": 0.01,
    "gamma": 0.8,
    "epsilon": 1.0,
    "epsilon_decay": 0.995,
    "epsilon_min": 0.1
}

best_config_s = {
    "alpha": 0.5,
    "gamma": 0.9,
    "epsilon": 1.0,
    "epsilon_decay": 0.995,
    "epsilon_min": 0.1
}

env = create_env(is_slippery=False)
agent = QLearningAgent(env.observation_space.n, env.action_space.n, **best_config_d)
train_agent(env, agent)

display_policy(agent.Q, shape=(4, 4), title="Final Policy (Deterministic Environment)")
print_q_table(agent.Q, shape=(4, 4))

env_slip = create_env(is_slippery=True)
agent_slip = QLearningAgent(env_slip.observation_space.n, env_slip.action_space.n, **best_config_s)
train_agent(env_slip, agent_slip)

display_policy(agent_slip.Q, shape=(4, 4), title="Final Policy (Stochastic Environment)")
print_q_table(agent_slip.Q, shape=(4, 4))
```

Figure 17 –Best hyperparameters derived, policy and q-table display

Deterministic Setting

In the deterministic environment, the arrows indicate that the agent prefers to move down and then left from the start, and finally transitions to moving right and down as it approaches the goal. This is a sensible strategy given the structure of the environment. The Q-table confirms that the agent is assigning increasingly higher values to states as it gets closer to the goal. The values rise steadily from 0.25 at the starting position to a perfect 1.00 just before the terminal state. This smooth gradient of Q-values suggests that the agent has effectively learned which states are more valuable and is able to reliably reach the goal by following its learned policy.

```
Final Policy (Deterministic Environment)
[['↓' '←' '←' '←']
 ['↓' '←' '←' '←']
 ['→' '↓' '↓' '']
 ['' '' '' '']]
Q-Table (max Q-values per state):
0.25 | 0.01 | 0.00 | 0.00
0.36 | 0.00 | 0.00 | 0.00
0.48 | 0.62 | 0.24 | 0.00
0.00 | 0.79 | 1.00 | 0.00
```

Figure 18 –Policy and Q-table for Deterministic Environment

Stochastic Setting

In the stochastic environment, things are more chaotic, as expected. The policy map reflects more mixed behaviors, with some arrows pointing up, left, or right, showing that the agent is trying to adapt to the randomness in the environment. The path isn't as straightforward as in the deterministic case, but there is still some directionality—especially the arrows pointing right from the start, indicating that the agent is at least attempting to move toward the goal. The Q-values in this version are significantly lower, with the highest being 0.58 near the goal.

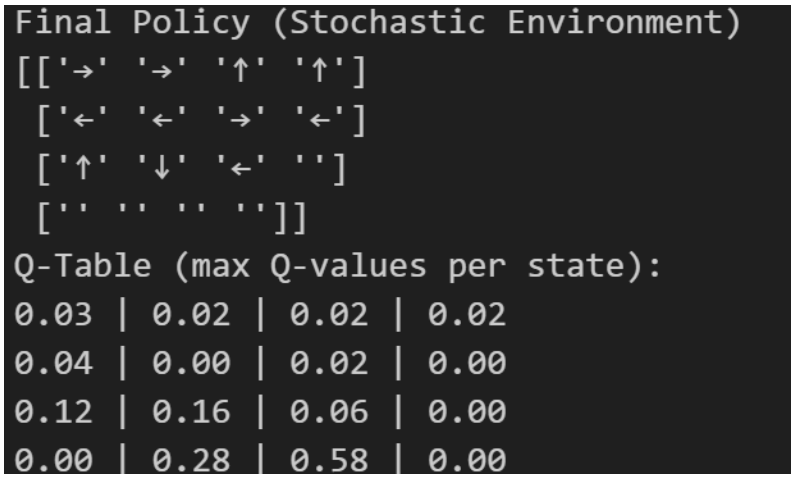

Figure 19 –Policy and Q-table for Stochastic Environment

Reward Shaping

In this part of the project, I focused on reward shaping—a technique used to guide an agent's learning by modifying the reward function. The goal was to help the agent learn more efficiently, especially in large or sparse environments like Frozen Lake, where successful actions might be rare and delayed. Normally, in Frozen Lake, the agent only gets a positive reward when it reaches the goal and receives nothing or falls into a hole otherwise. With reward shaping, I added a small penalty for each step to encourage the agent to find the shortest path and a larger penalty when falling into a hole to reinforce the idea that holes are bad.

Deterministic Setting

In the figure below, we can see that using shaped rewards with a step penalty of -0.05 and a hole penalty of -2.0 leads to a much slower start compared to default rewards. However, the agent does eventually reach a decent policy. This tells us that reward shaping introduces a cost to exploration early on, but with enough training, the agent compensates and learns well.

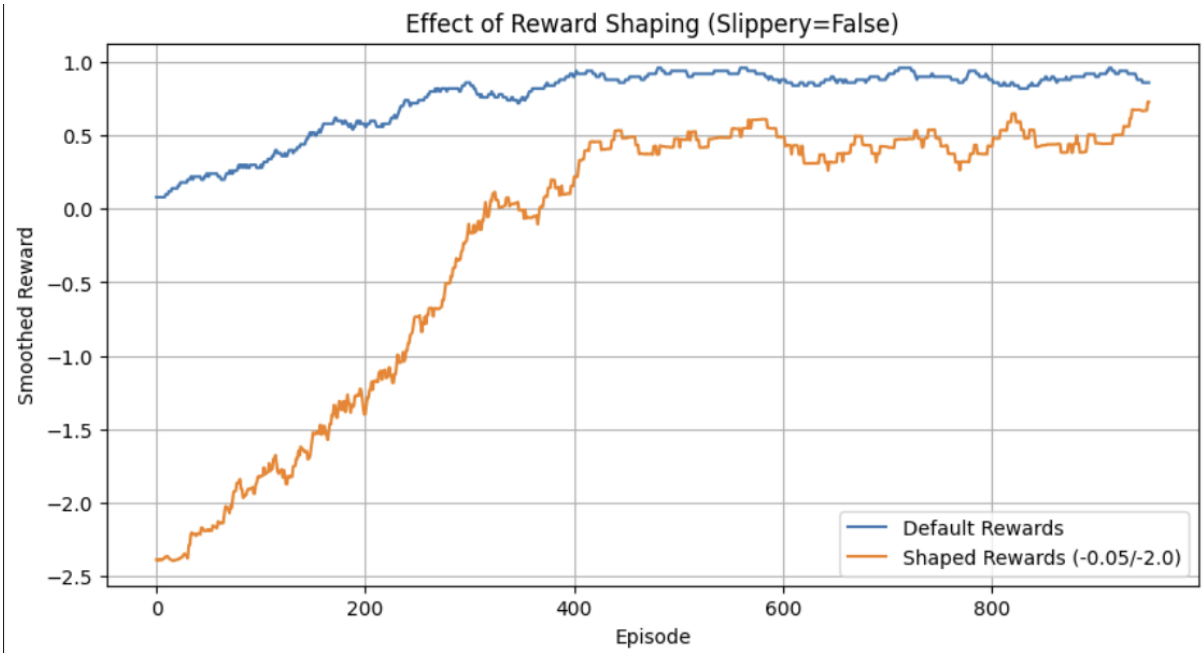

Figure 20 –Effect of Reward Shaping (Slippery=False)

Then I tested three shaping configurations: light (-0.01/-1.0), medium (-0.05/-2.0), and harsh (-0.1/-5.0). Unsurprisingly, the light configuration performed best, maintaining high reward levels, while the harsh penalty slowed learning drastically. This suggests that shaping works best when the penalties are just enough to nudge the agent, but not so harsh that they discourage any form of exploration.

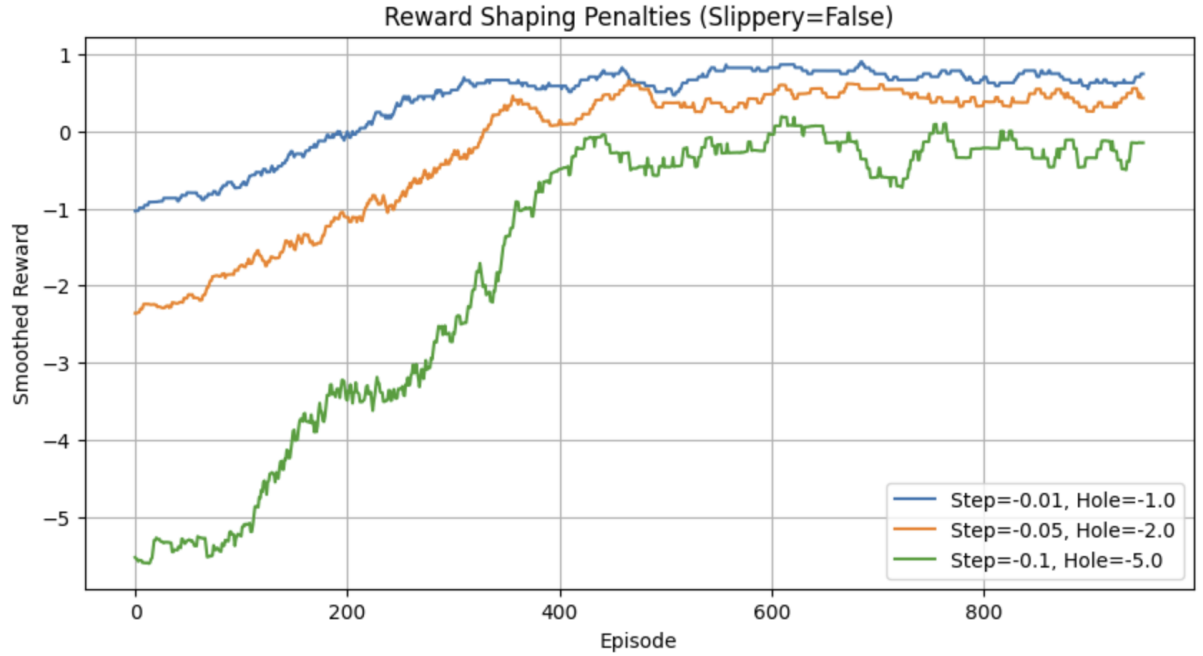

Figure 21 –Reward Shaping Penalties (Slippery=False)

The final policy shows a mostly optimal route to the goal, and the agent successfully learns to avoid holes. However, some Q-values are still negative—this is because even though the agent is reaching the goal, the step penalties add up over time, slightly lowering the overall reward unless the path is extremely efficient.

```
Policy After Shaping (Slippery=False)
[['↓' '→' '↓' '←']
 ['↓' '←' '↓' '←']
 ['→' '→' '↓' '']
 ['' '' '' '']]
Q-Table (max Q-values per state):
0.06 | -0.05 | 0.04 | -0.03
0.19 | 0.00 | 0.29 | 0.00
0.35 | 0.53 | 0.74 | 0.00
0.00 | 0.21 | 1.00 | 0.00
```

Figure 22 –Policy and Q-table after shaping (Slippery=False)

I also plotted the average reward per episode. We can clearly see that while the average reward starts off very negative, due to the step and hole penalties, the agent steadily improves its behaviour. Around episode 300, it begins crossing into positive territory, indicating it's successfully navigating to the goal more often than falling into holes or wandering inefficiently. By the later episodes, the average reward stabilizes above 0.5, meaning the agent has learned to avoid penalties and efficiently reach the goal.

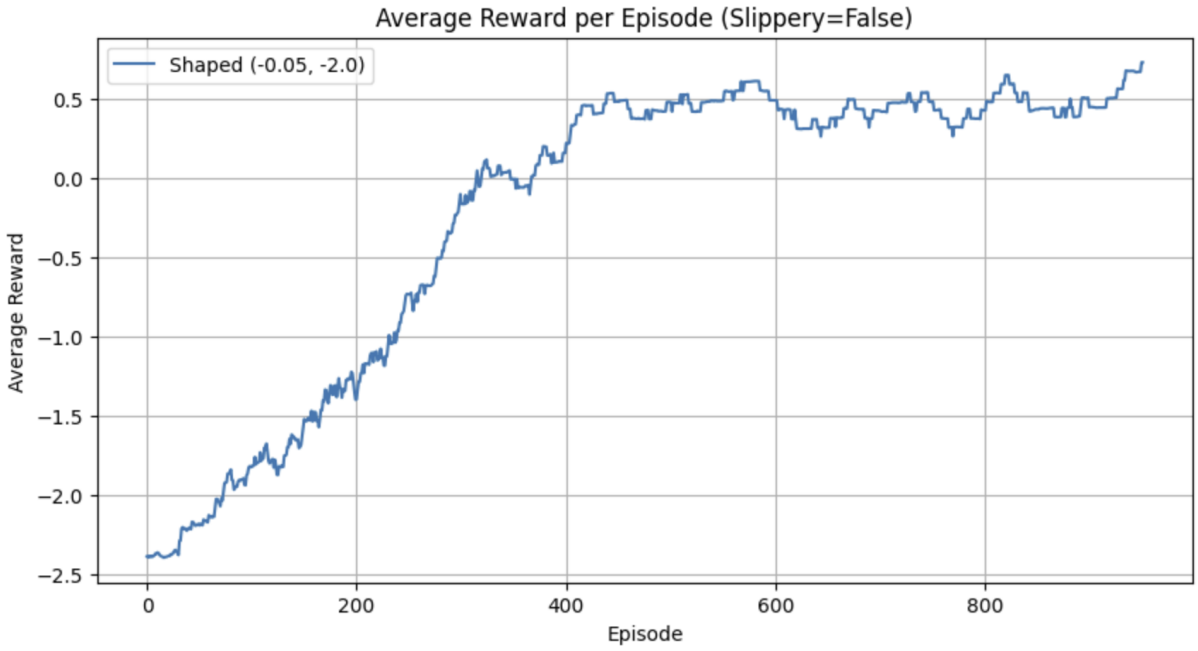

Figure 23 –Average Reward per Episode (Slippery=False)

Stochastic Setting

In the stochastic environment (slippery = True), reward shaping turned out to be more difficult. In the figure below comparing shaped vs default rewards, default rewards actually performed better overall, with the shaped version struggling to climb above a -2.0 average.

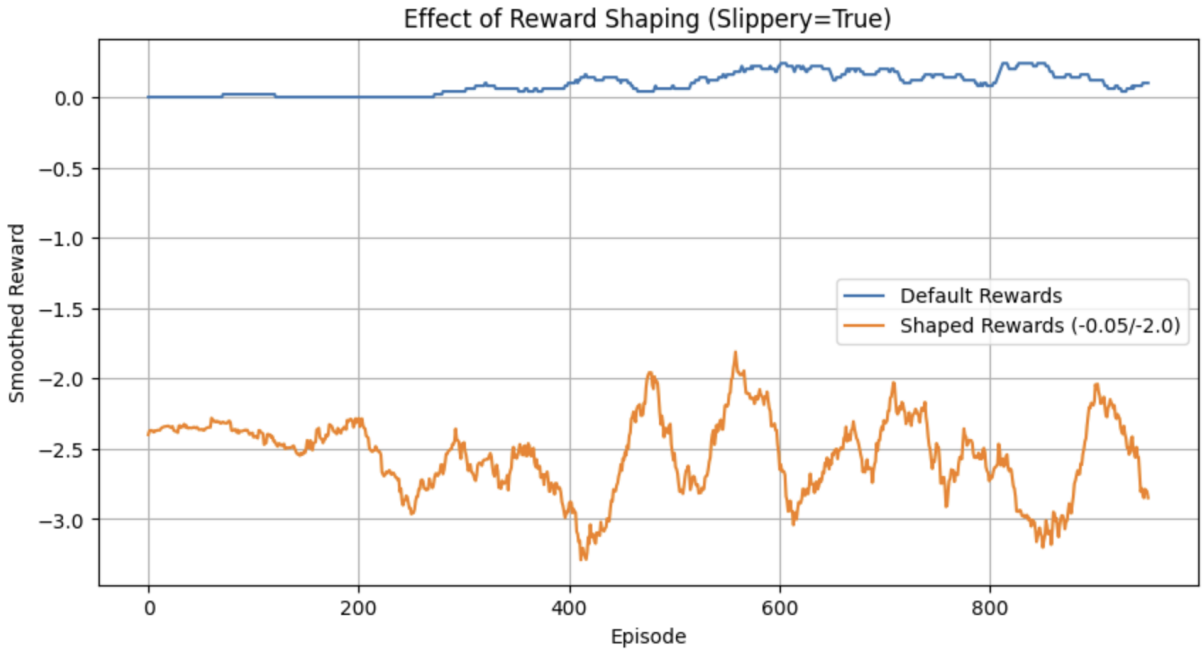

Figure 24 –Effect of Reward Shaping (Slippery=True)

Below we can see a very clear trend: the stronger the penalties, the worse the agent performs. With the harshest setting (-0.1/-5.0), the average reward drops all the way to -6.0, suggesting the agent is being too heavily punished to effectively explore and learn. This reinforces the idea that shaping needs to be used with caution in stochastic environments—while it can help guide learning, it can also get in the way if overdone.

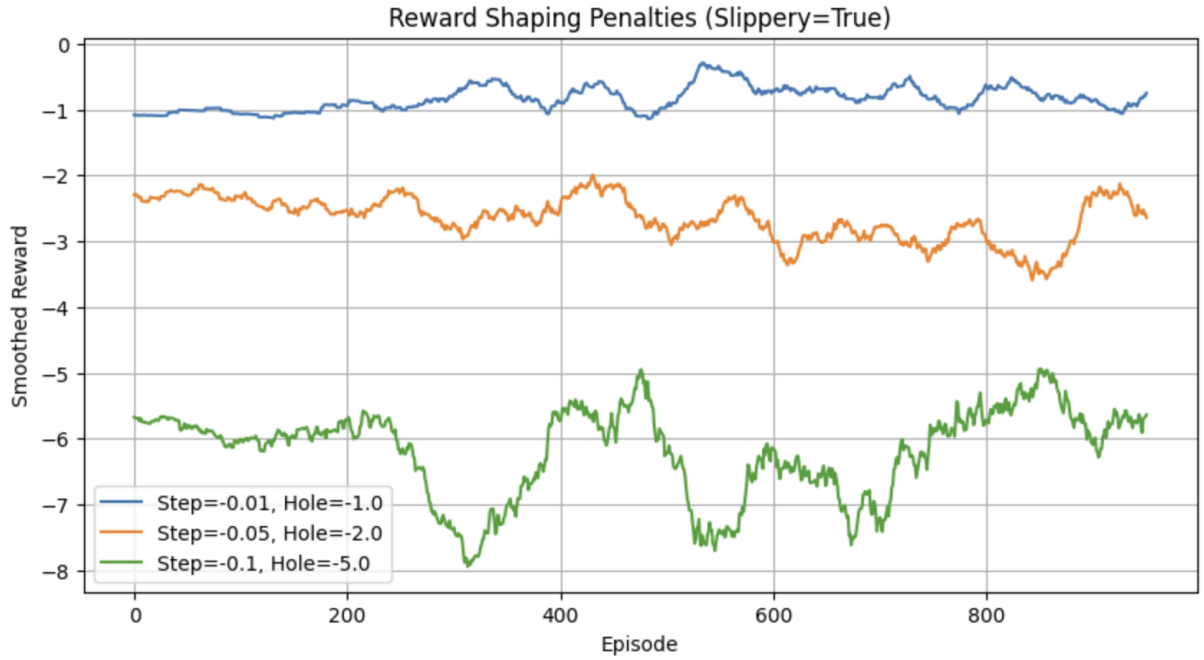

Figure 25 –Reward Shaping Penalties (Slippery=True)

As seen in the final policy, the agent still managed to reach the goal but had much lower Q-values overall. The randomness introduced by slipperiness makes it harder for the agent to associate actions with consistent outcomes. Combined with negative shaping penalties, this leads to slower and less stable learning.

```
Policy After Shaping (Slippery=True)
[['↓' '→' '↓' '↓']
 ['↓' '←' '↓' '←']
 ['→' '→' '↓' '']
 ['' '' '' '']]
Q-Table (max Q-values per state):
-0.15 | -0.07 | -0.13 | -0.07
-0.09 | 0.00 | -0.05 | 0.00
-0.05 | -0.01 | 0.11 | 0.00
0.00 | 0.34 | 0.89 | 0.00
```

Figure 26 –Policy and Q-table after shaping (Slippery=True)

In the plot below things are less stable than when slippery=False. The average reward remains negative throughout and fluctuates quite a bit. This is expected because the stochastic environment introduces randomness in the movement, making it harder for the agent to follow an optimal policy, even after learning. Still, we can observe some peaks where the agent starts performing better, but overall, the shaped penalties may be too strict in this slippery context, preventing consistent improvement. This highlights that while reward shaping works well in deterministic settings, it may need to be fine-tuned for environments with higher uncertainty.

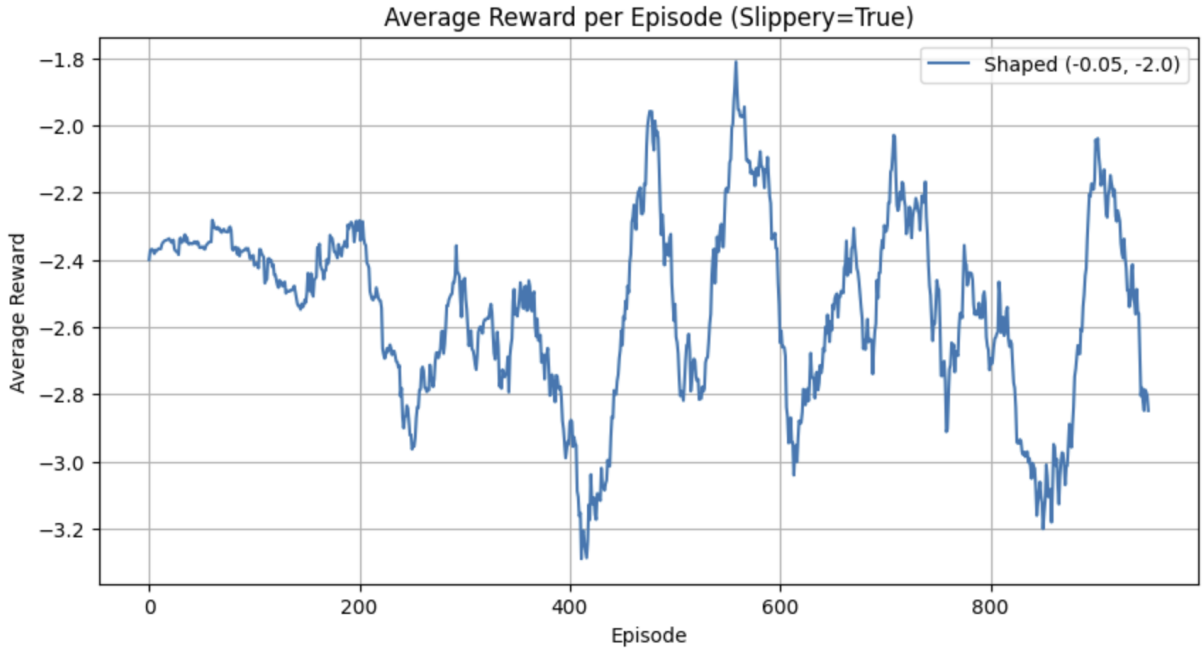

Figure 27 –Average Reward per Episode (Slippery=True)

Stochastic vs Deterministic

Before we continue, let's do a fair comparison between the agent's learning performance and final policies in both deterministic and stochastic versions of Frozen Lake. Looking at the learning curves in the comparison plot, the difference is immediately obvious. In the deterministic setting, the agent steadily improves and reaches near-optimal performance quite quickly, with the reward consistently approaching 1.0 after around 400 episodes. On the other hand, in the stochastic environment, learning is slower and much more unstable. Although the

agent does improve gradually, it never reaches the same level of confidence or consistency, plateauing around a reward of 0.3. This illustrates the challenge of learning under uncertainty—even with the same algorithm and hyperparameters, stochasticity significantly hinders progress.

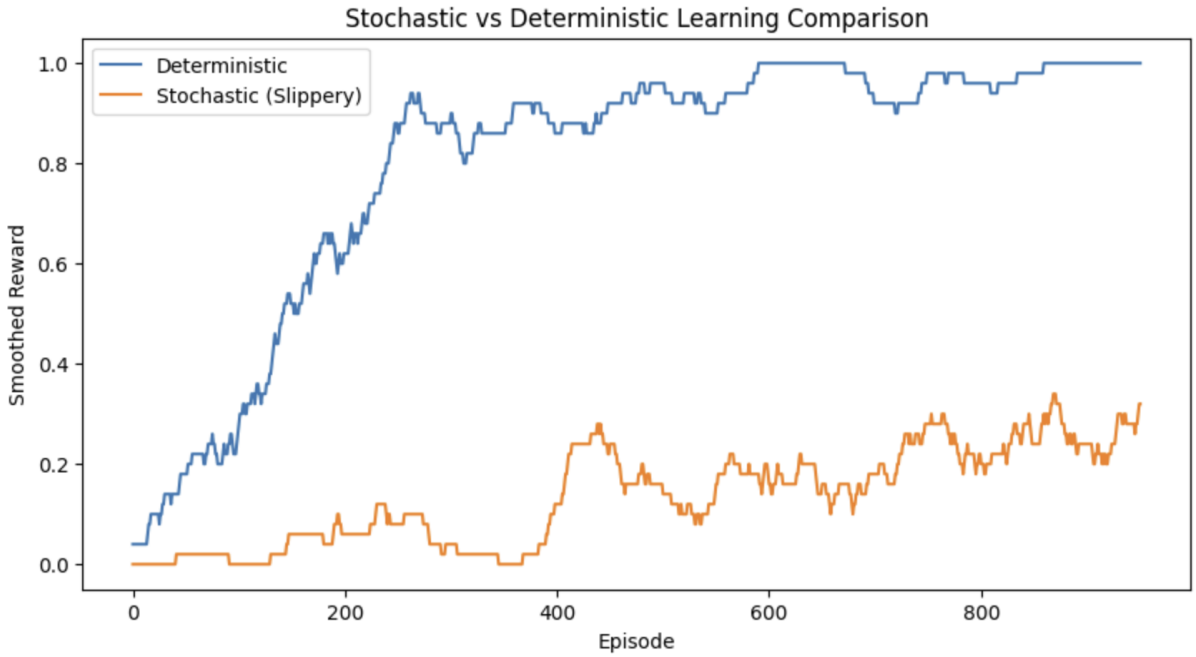

Figure 28 –Stochastic vs Deterministic Learning Comparison

This pattern is also reflected in the learned policies and Q-tables. The deterministic policy is very clear and direct: the agent consistently moves toward the goal using optimal directions like right and down. The Q-values for most of the states are high, often close to or exactly 1, indicating strong confidence in the chosen actions. The stochastic policy, in contrast, appears less structured and more scattered. The agent doesn't stick to one clear path, and the Q-values are much lower, reflecting a lack of confidence due to the randomness in action outcomes.

```
Deterministic Policy:
Policy (Deterministic)
[['↓' '←' '←' '←']
 ['↓' '←' '←' '←']
 ['→' '↓' '↓' '']
 ['' '' '' '']]
Stochastic Policy:
Policy (Stochastic)
[['←' '←' '←' '←']
 ['←' '←' '↓' '←']
 ['↑' '→' '←' '']
 ['' '' '' '']]
Q-Table (Deterministic):
Q-Table (max Q-values per state):
0.95 | 0.82 | 0.20 | 0.00
0.96 | 0.00 | 0.00 | 0.00
0.97 | 0.98 | 0.97 | 0.00
0.00 | 0.99 | 1.00 | 0.00
Q-Table (Stochastic):
Q-Table (max Q-values per state):
0.23 | 0.07 | 0.00 | 0.00
0.24 | 0.00 | 0.13 | 0.00
0.25 | 0.29 | 0.44 | 0.00
0.00 | 0.38 | 0.79 | 0.00
```

Figure 29 – Policy and Q-table of both Deterministic and Stochastic Settings

Craving Behaviour (Q-value Heatmap)

Now, we are going to explore the craving behaviour by visualizing the agent's learned Q-values in a heatmap. The idea is to see how strongly the agent "craves" certain actions based on how valuable it believes they are. In other words, the Q-value heatmap doesn't just show where the agent prefers to go—it reveals how confident it is about those choices. The closer the agent is to the goal, the more confident (and greedy) it usually becomes, because it's about to receive a reward.

Deterministic Setting

Looking at the deterministic heatmap, we can clearly observe that the Q-values increase as the agent moves closer to the goal. The bottom row has the most saturated cells, especially the cell right before the goal, which reaches a Q-value close to 1. This strong contrast indicates that the agent has fully understood the value of getting close to the goal and has developed a very greedy and confident policy near the finish line. The directions the agent chooses are clean and consistently point toward the optimal path.

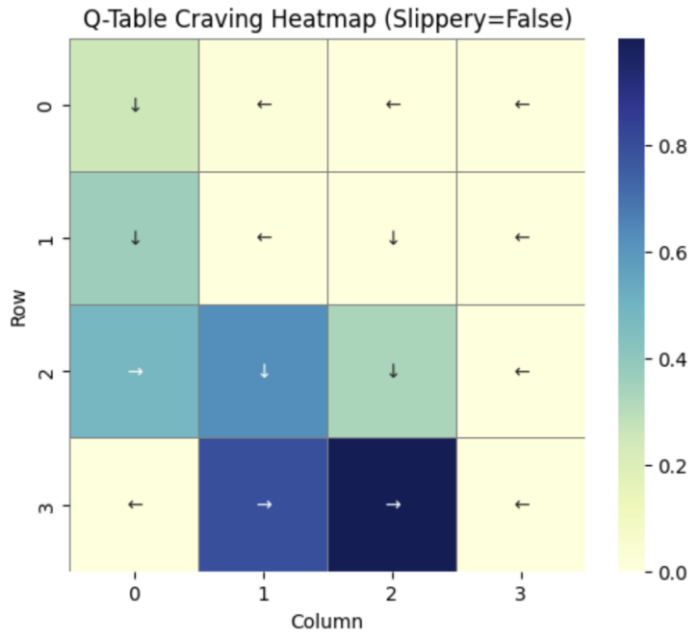

Figure 30 – Q-Table Craving Heatmap (Slippery=False)

Stochastic Setting

In contrast, the stochastic heatmap tells a different story. While there's still a gradual increase in Q-values as we approach the goal, the values are generally lower and more spread out. The policy arrows also appear more scattered and less direct compared to the deterministic case, which reflects the added challenge of planning under uncertainty.

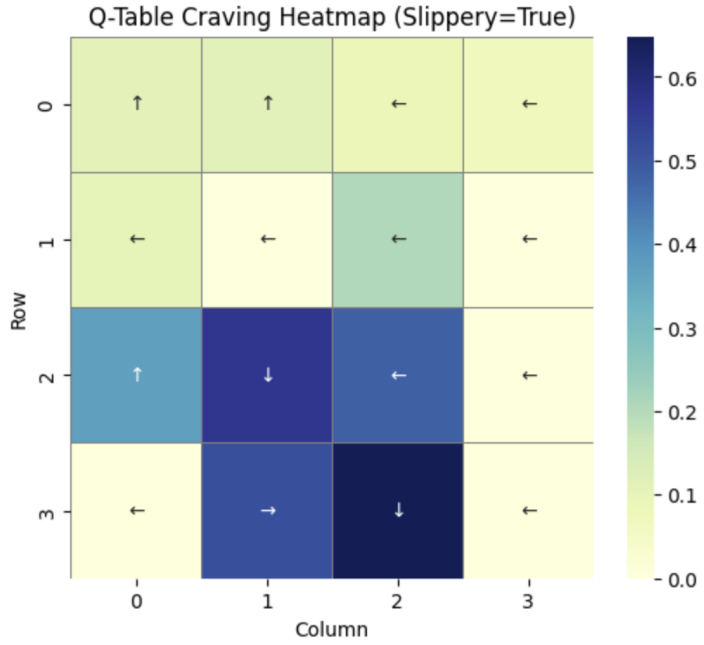

Figure 31 – Q-Table Craving Heatmap (Slippery=True)

Experience Replay

Now, let's explore Experience Replay, a technique commonly used in reinforcement learning to improve the efficiency and stability of training agents. Rather than updating the Q-table

based only on the most recent experience, Experience Replay involves storing past experiences in a buffer and sampling from this buffer to train the agent. The goal was to investigate whether this added complexity results in better learning performance, especially under both deterministic and stochastic conditions.

Deterministic Setting

Starting with the Slippery=False (Deterministic) environment, the results are striking. The agent using Experience Replay outperforms the regular Q-learning agent by a large margin.

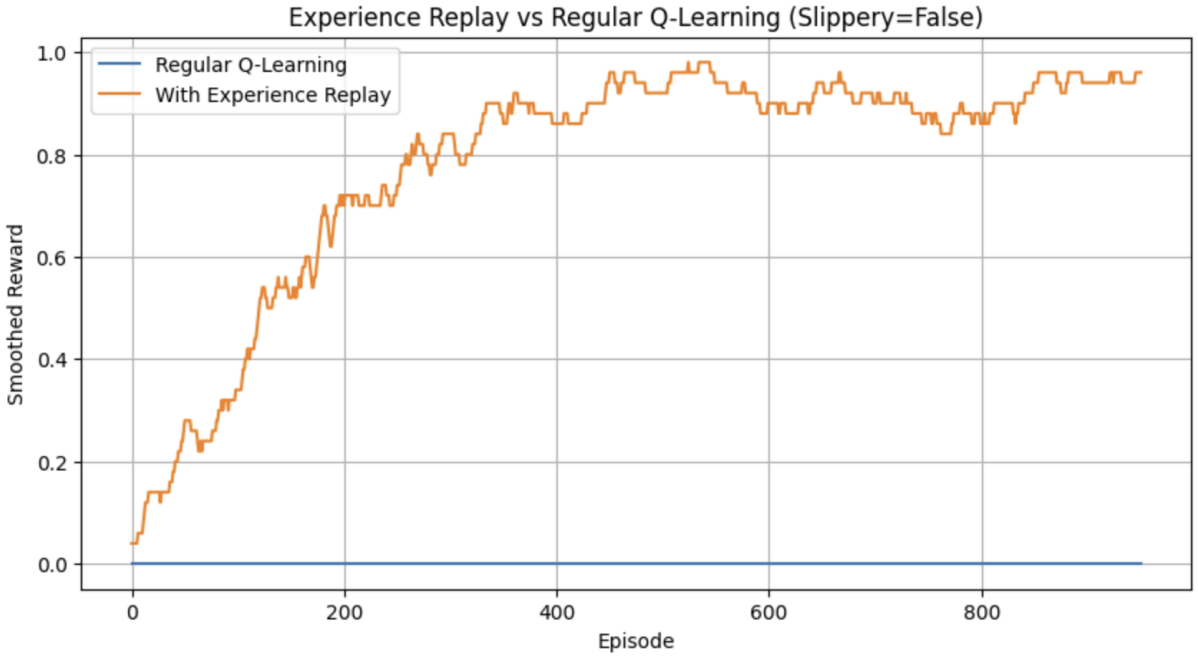

Figure 32 – Experience Replay vs Regular Q-Learning (Slippery=False)

The corresponding policy learned by the agent is highly optimal, with clear directional actions leading toward the goal, and the Q-table confirms this with high values concentrated along the expected shortest path. This demonstrates the power of replay in stabilizing and accelerating learning in less noisy environments.

```
Policy (With Experience Replay)
[['→' '→' '↓' '←']
 ['↓' '←' '↓' '←']
 ['→' '→' '↓' '']
 ['' '' '' '']]
Q-Table (max Q-values per state):
0.33 | 0.41 | 0.51 | 0.41
0.41 | 0.00 | 0.64 | 0.00
0.51 | 0.64 | 0.80 | 0.00
0.00 | 0.79 | 1.00 | 0.00
```

Figure 33 – Policy and Q-table with Experience Replay (Slippery=False)

Stochastic Setting

In the Slippery=True (Stochastic) setting, we again observe that Experience Replay provides a notable advantage. Although its smoothed reward remains modest—peaking around 0.2—the difference is still substantial.

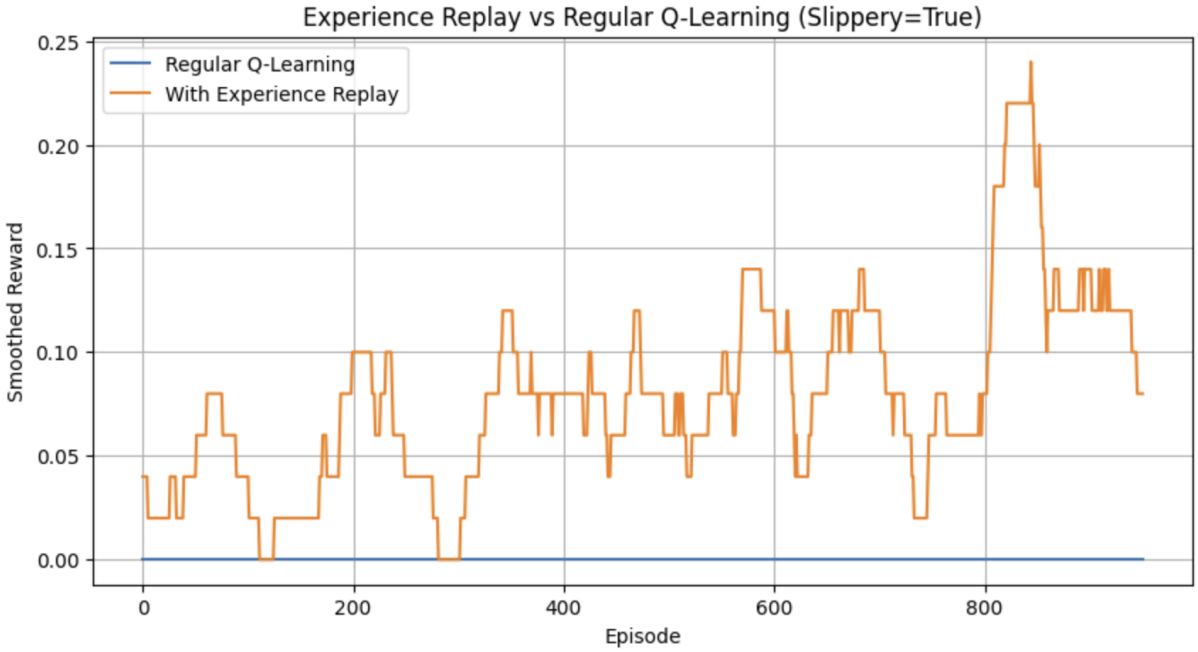

Figure 34 – Experience Replay vs Regular Q-Learning (Slippery=True)

The learned policy contains a coherent path to the goal, and the Q-table, though lower in magnitude than in the deterministic case, reveals well-formed estimates guiding the agent in the correct direction.

```
Policy (Replay Agent, Slippery=True)
[['→' '→' '↓' '↑']
 ['←' '←' '←' '←']
 ['↓' '↓' '←' '']
 ['' '' '' '']]
Q-Table (max Q-values per state):
0.21 | 0.22 | 0.29 | 0.25
0.21 | 0.00 | 0.19 | 0.00
0.26 | 0.47 | 0.53 | 0.00
0.00 | 0.64 | 0.75 | 0.00
```

Figure 35 – Policy and Q-table with Experience Replay (Slippery=True)

Injected Trajectories

In this chapter, we explore the concept of injected trajectories, a strategy where we provide the agent with a set of successful state-action sequences—effectively showing it a path to the goal. This technique acts like a form of offline expert demonstration, helping the agent get a better idea of the reward structure and optimal decisions early in training. I chose to test this only in

the slippery=False (deterministic) setting, since hardcoding an optimal path for a stochastic environment is unreliable: the agent may slip and deviate from the trajectory, making the injection ineffective or misleading. Here, the goal was simply to evaluate whether this strategy can boost performance when the environment is predictable.

The plot below compares Q-learning performance in the deterministic environment with and without an injected successful trajectory. The agent that receives a predefined trajectory learns faster, achieving higher rewards earlier than the baseline agent. Although both eventually converge to similar performance, the injected trajectory clearly accelerates early learning.

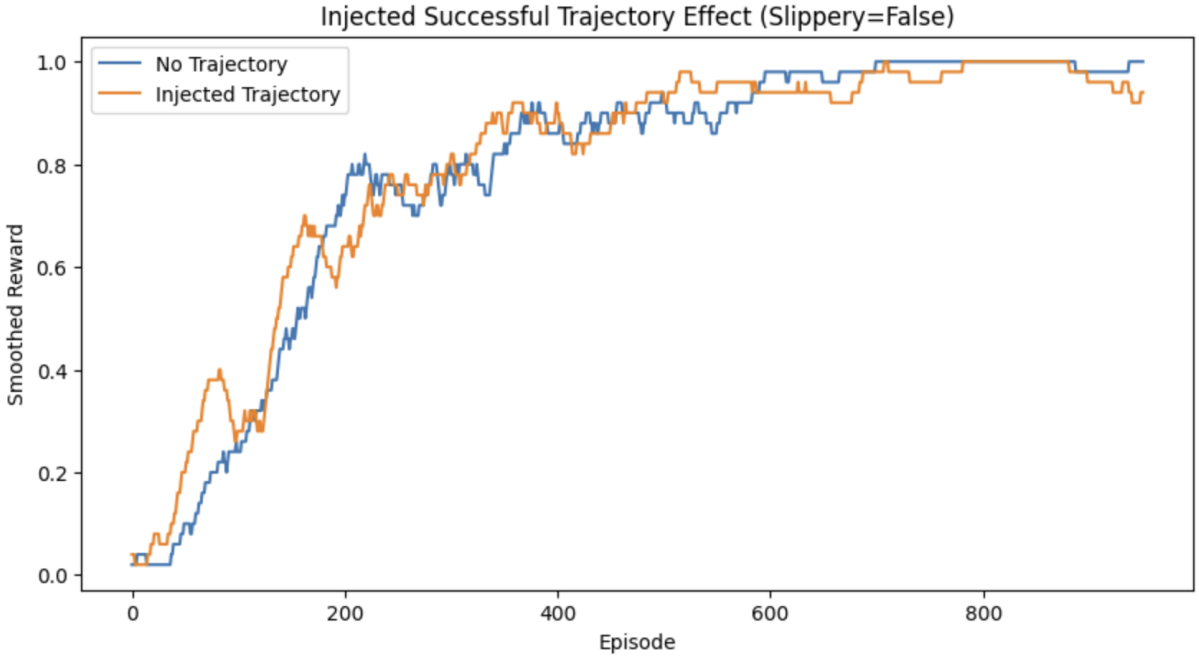

Figure 36 – Injected Successful Trajectory Effect (Slippery=False)

Now, I decided to dive deeper by varying the discount factor γ to observe how much the agent values future rewards when learning from the injected trajectory. Lower values like γ=0.5 and γ=0.7 still show decent performance but are slightly more short-sighted, which causes slower convergence and slightly less optimal asymptotic behavior. γ=0.9 performs well and converges quickly to an optimal policy. However, γ=0.99 surprisingly fails completely. his result suggests that when the discount factor is too high, and if the rewards are sparse (which is the case in FrozenLake), the agent might overly rely on distant rewards and ignore the immediate structure.

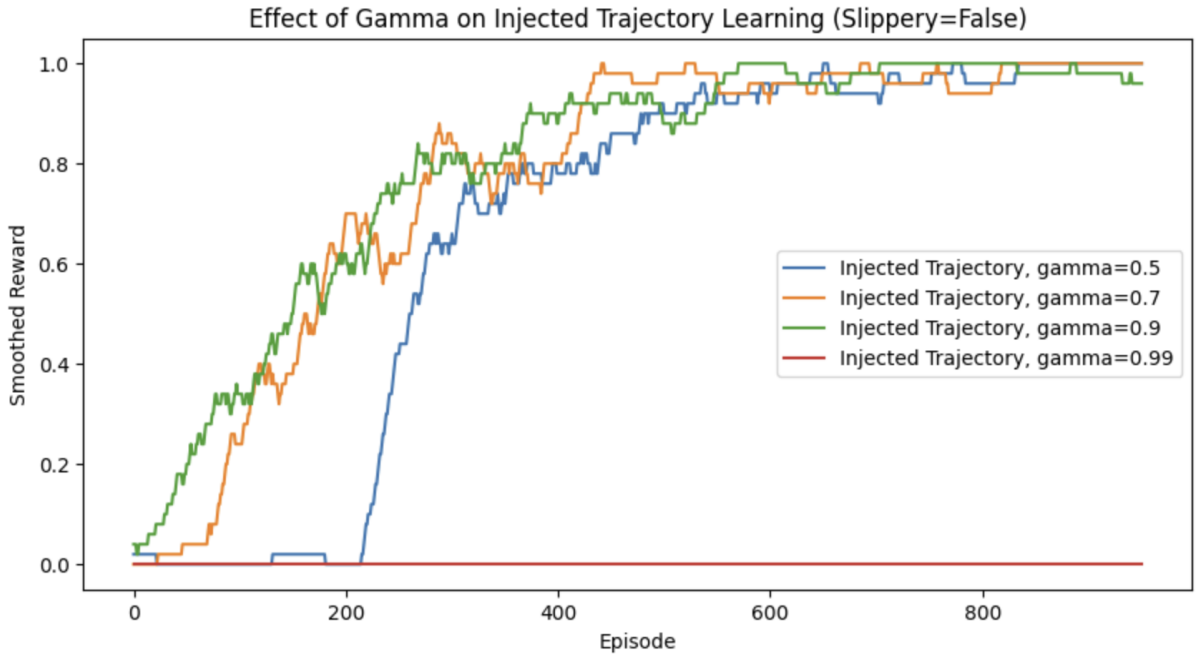

Figure 37 – Effect of Gamma on Injected Trajectory Learning (Slippery=False)

Learning Curves Summary

This summary plot provides a consolidated view of all the learning strategies tested in the deterministic environment. Epsilon-Greedy and Experience Replay are the top-performing methods, both achieving high reward levels consistently after around 400 episodes. Experience Replay shows slightly faster learning in early stages. Injected Trajectories also lead to stable high rewards. Boltzmann exploration with T=1.0 fails to show meaningful improvement in this setting, while Penalty Shaping significantly underperforms due to the added negative rewards, slowing down learning but eventually improving. This comparison highlights that, in the deterministic FrozenLake setup, strategies enhancing sample efficiency and encouraging successful exploration—like Experience Replay and Injected Trajectories—are the most effective.

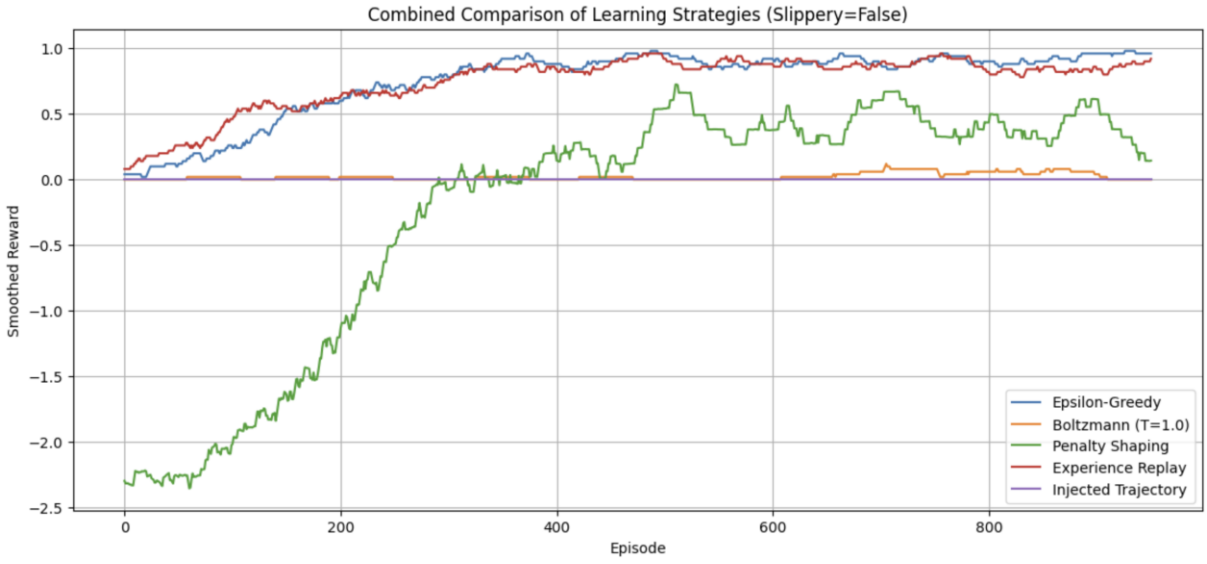

Figure 38 – Combined Comparison of Learning Strategies (Slippery=False)

Scaling to Larger Maps

Now, that we have explored the way the agent is working, tested out different strategies and saw results in plots and in the policies, we are going to move on to working with larger maps, specifically 8x8 and 16x16 FrozenLake environments. While the 8x8 map is supported natively by Gym, the 16x16 version had to be custom-built with a start ('S'), goal ('G'), and randomized hole placements ('H'), ensuring that the path is challenging but solvable (turned out to be too challenging). To guide learning in these larger, sparse environments, I use reward shaping (step penalties and hole penalties) and inject handcrafted successful trajectories into the agent's Q-table prior to training.

The plot below displays the learning curve on the 8x8 FrozenLake. After trajectory injection and sufficient training episodes, the agent quickly converges to a high reward, showing strong and consistent performance.

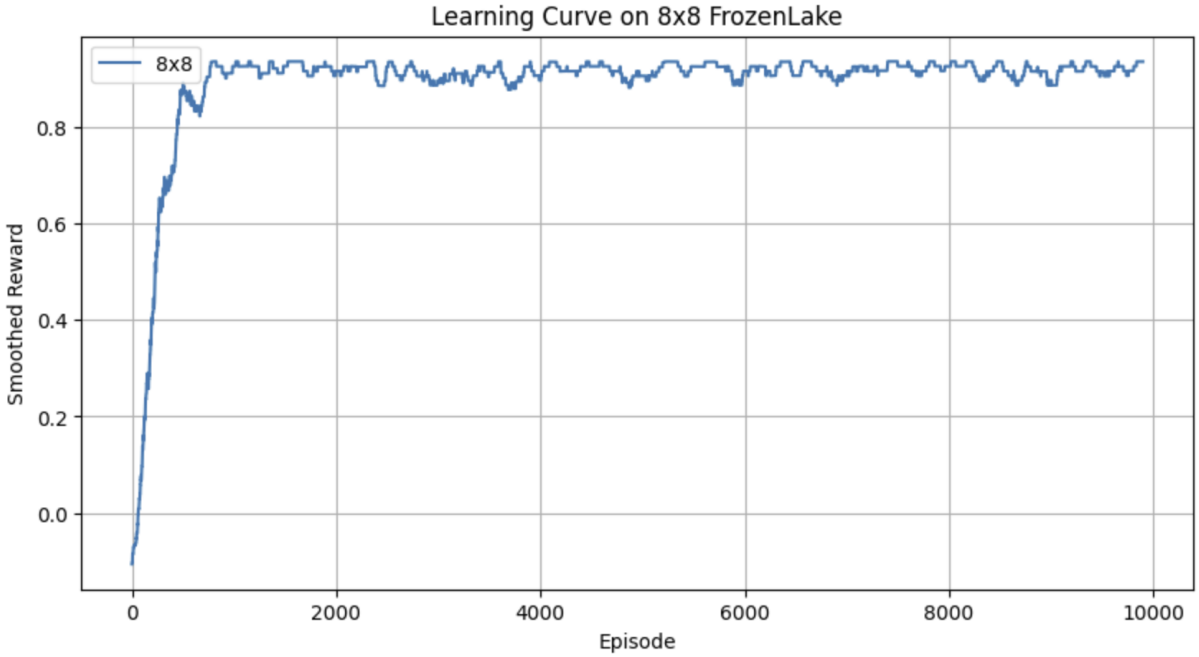

Figure 39 – Learning Curve on 8x8 FrozenLake

Now we see a snippet of the learned Q-values for each state and action in the 8x8 grid. This data reflects the agent's confidence in each possible move. The values vary, but that is normal in a complex environment.

```
State (0, 0) | Q-values: {0: 0.42506507749421213, 1: 0.4646762916074504, 2: 0.4468968671475586, 3: 0.4317980921407614}
State (0, 1) | Q-values: {0: 0.0731479691495689, 1: 0.4938150136591965, 2: 0.12801551133702718, 3: 0.11986489295438225}
State (0, 2) | Q-values: {0: 0.07843991134286449, 1: 0.5154377115704072, 2: 0.055951394983773425, 3: 0.10586815281882711}
State (0, 3) | Q-values: {0: 0.019774486138909437, 1: 0.13054505024229848, 2: 0.5439965826161638, 3: 0.0605584545822173}
State (0, 4) | Q-values: {0: 0.08449456154558906, 1: 0.5929779701470229, 2: 0.08653941679254618, 3: 0.24232660057304634}
State (0, 5) | Q-values: {0: 0.2785766564568371, 1: 0.006772522290504822, 2: -0.00045455577320801784, 3: 0.06153188512776831}
State (0, 6) | Q-values: {0: 0.014103718873313019, 1: 0.11736882882270115, 2: -0.0010949049572666423, 3: 0.008816424940487459}
State (0, 7) | Q-values: {0: 0.014514520525702441, 1: -0.0005073022174205345, 2: -0.0024126489471461374, 3: -0.0017745462915865168}
State (1, 0) | Q-values: {0: 0.4603941753357838, 1: 0.10839158742839898, 2: 0.4943960964288955, 3: 0.4331461571678401}
State (1, 1) | Q-values: {0: 0.4555939253386126, 1: 0.4449485528483602, 2: 0.5256801015041008, 3: 0.4303985523644371}
State (1, 2) | Q-values: {0: 0.48291974140207455, 1: 0.47954481402098614, 2: 0.558610633162212, 3: 0.45171657719567165}
State (1, 3) | Q-values: {0: 0.5127713376769159, 1: -0.004974231123963399, 2: 0.5932743506970659, 3: 0.47020074520527755}
State (1, 4) | Q-values: {0: 0.5524048662196691, 1: 0.6297624744179646, 2: 0.5479304570729995, 3: 0.5382954559610691}
State (1, 5) | Q-values: {0: 0.5932023083032278, 1: 0.18197897569719274, 2: 0.14026651621264535, 3: 0.08755256027905153}
State (1, 6) | Q-values: {0: 0.10647763985485428, 1: 0.6856582940574285, 2: 0.0007391190261692227, 3: 0.028698227436920753}
State (1, 7) | Q-values: {0: 0.0025624892278923134, 1: 0.1671801749789919, 2: 0.0002621977825794655, 3: -0.0012945244752520736}
State (2, 0) | Q-values: {0: 0.0960791057965454, 1: 0.1103086721473621, 2: 0.07072348904396449, 3: 0.024730935004418465}
State (2, 1) | Q-values: {0: 0.06499574764938754, 1: 0.08129345556589826, 2: 0.02282199085934959, 3: 0.4923567539011421 4}
State (2, 2) | Q-values: {0: 0.0262586226332576, 1: 0.004385038308575529, 2: -0.0038561603772519497, 3: 0.522826211406606}
State (2, 4) | Q-values: {0: -0.004994694416940017, 1: 0.560549112933298, 2: 0.6681710257031213, 3: 0.5777713391745364}
State (2, 5) | Q-values: {0: 0.619692555464503, 1: -0.004974231123963399, 2: 0.7086010796874966, 3: 0.5519313150322481}
State (2, 6) | Q-values: {0: 0.6639062713157834, 1: 0.751159031249997, 2: 0.6125965399724242, 3: 0.5666247596775439}
State (2, 7) | Q-values: {0: 0.6977370547484119, 1: 0.11639594154969622, 2: 0.013011372809054088, 3: 0.020259018128304464}
State (3, 0) | Q-values: {0: 0.10850733974010425, 1: 0.10232972735890214, 2: 0.15389093789519712, 3: 0.1080309784060963}
State (3, 1) | Q-values: {0: 0.10983855549983797, 1: 0.071360537054965 2, 2: 0.009198794554454387, 3: 0.2921875825048028 7}
...
State (7, 1) | Q-values: {0: 0.08771627775667534, 1: 0.08809839089399807, 2: 0.08716363663457312, 3: -0.00343094701955}
```

Figure 40 – Snippet of Q-table for 8x8

The next screenshot presents the learned policy grid derived from the Q-table. Each cell shows the direction (→ ↓ ↑ ←) corresponding to the action with the highest Q-value in that state. This visualization confirms that the agent has learned a coherent path through the environment from the start to the goal, avoiding holes.

```
Policy Grid (best action per state):
[['↓' '↓' '↓' '→' '↓' '←' '↓' '←']
 ['→' '→' '→' '→' '↓' '←' '↓' '↓']
 ['↓' '↑' '↑' '←' '→' '→' '↓' '←']
 ['→' '↑' '←' '→' '↑' '←' '↓' '←']
 ['↓' '←' '←' '←' '↑' '→' '→' '↓']
 ['←' '←' '←' '↓' '←' '→' '←' '↓']
 ['↓' '↓' '←' '←' '←' '→' '←' '↓']
 ['↑' '↓' '←' '←' '↓' '→' '←' '←']]
```

Figure 41 – Policy Grid for 8x8

Next, we see the Average Reward per Episode. It demonstrates that the agent quickly learns an optimal policy: the reward rapidly increases in the early episodes and then stabilizes around 0.9, meaning the agent successfully reaches the goal in most episodes.

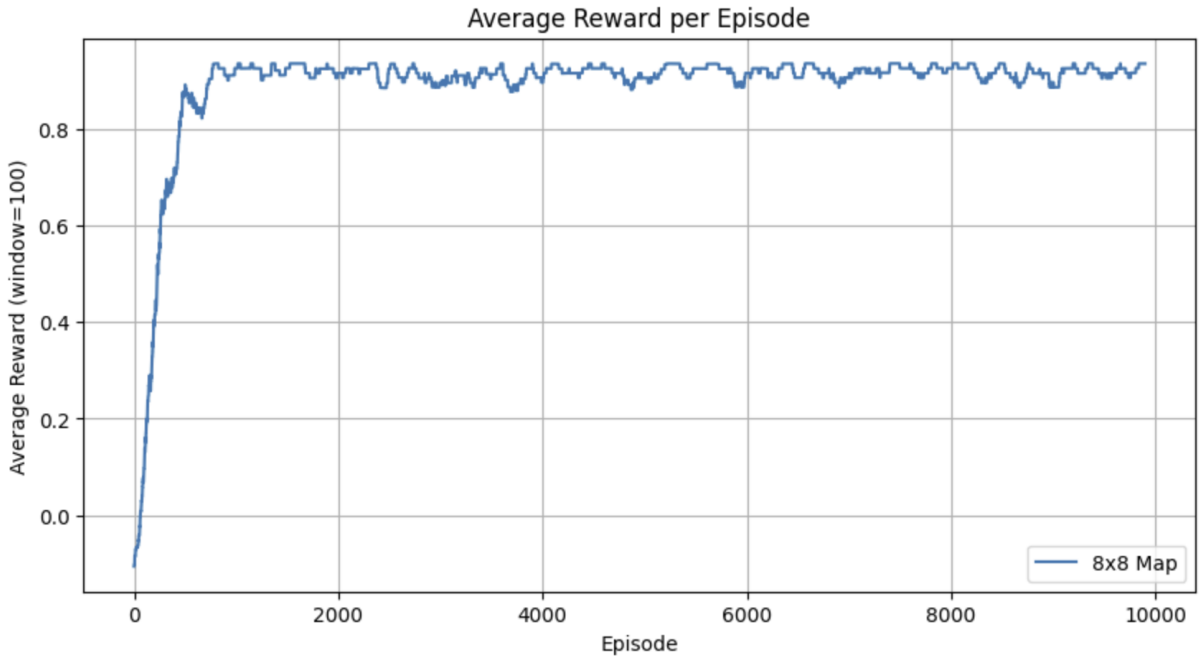

Figure 42 – Average Reward per Episode for 8x8

The next 2 plots depict the learning curve and average reward for the custom 16x16 map. Despite applying the same methods as in the 8x8 case (reward shaping and trajectory injection (different trajectory)), the agent struggles. The reward remains low and plateaus quickly, indicating that while it learns to avoid some holes, it rarely reaches the goal.

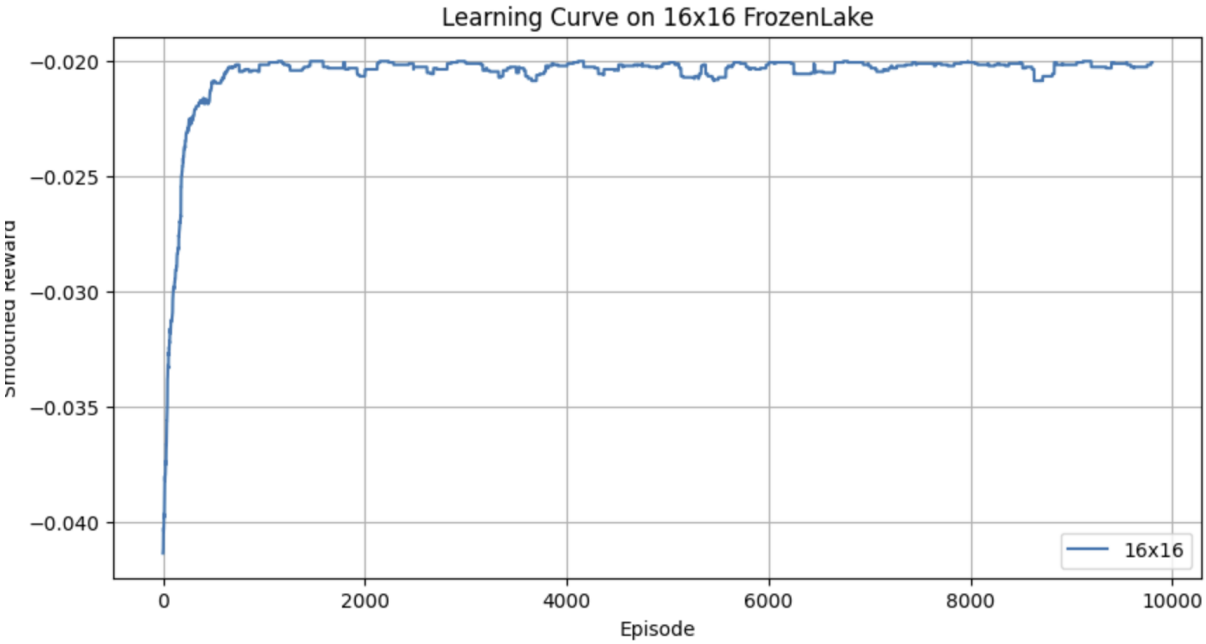

Figure 43 – Learning Curve on 16x16 FrozenLake

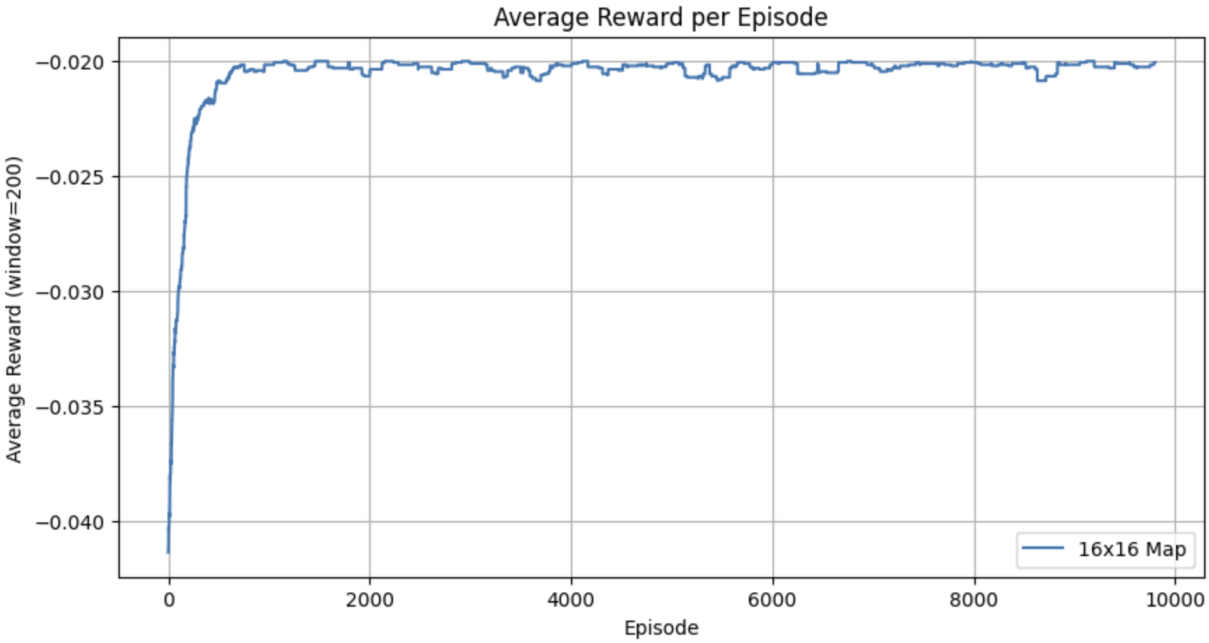

Figure 44 – Average Reward per Episode for 16x16

The larger maps present more challenges. After trajectory injection, rewards and penalties, the 8x8 map was manageable and success was achieved. However, with the 16x16, using again rewards and penalties, and a custom trajectory, the goal is not achieved in the time being. A larger environment presents more difficulty as expected. During my experiments, I tried with a variety of maps for the 16x16 (as they are created manually) and when I created a really simple path for the agent, it was able to go to the gift, but then it is not challenging at all. I tried applying what I had learned in the 4x4 map and I think it went well for the time available.

Agent Behaviour Videos

Lastly, I want to highlight videos of the agents when successfully going for the gift. All these videos can be found in the repository as well. In different sections of the notebook, I exported a video of the agent going to the gift. In the results folder in the repository, you can see a few of the videos generated through the exploration process.

Conclusion

Throughout this project, I explored how a reinforcement learning agent can learn to navigate the FrozenLake environment using Q-learning. Starting from tuning basic hyperparameters like alpha, gamma, and epsilon, I progressively introduced more advanced techniques such as reward shaping, experience replay, and injected trajectories. I also compared different exploration strategies like epsilon-greedy versus Boltzmann sampling and analyzed how deterministic and stochastic (slippery) settings affect the learning process. These experiments made it clear that the agent's performance is highly sensitive to both environment complexity and parameter settings. In simpler, non-slippery settings, the agent could quickly learn optimal policies with the right exploration decay or replay mechanisms. However, in slippery or larger environments, strategies like shaping rewards and injecting successful trajectories became essential to overcome challenges like sparse rewards or random transitions.

I used visual tools such as policy maps, Q-value heatmaps, and learning curves to interpret the results and understand the agent's behavior more deeply. Each plot and figure in this report was generated directly from the accompanying code notebook and is labeled with a number in both the Word document and the relevant code cell, helping to keep everything traceable and organized. Overall, this project helped me understand the challenges and techniques in reinforcement learning and gave me hands-on insight into how agents truly learn through trial and error.

Deep Q-Learning on CartPole

Introduction

In this section, I am going to explore the application of deep Q-learning (DQL) to the classic CartPole-v1 environment provided by Gymnasium. CartPole is a benchmark control task in which the agent must learn to balance a pole on a moving cart by applying left or right forces. The environment provides a continuous state space (position, velocity, angle, and angular velocity) and a discrete action space with two actions: move left or move right.

To solve this task, I employ a Deep Q-Network (DQN), which approximates the Q-values using a neural network instead of a traditional Q-table. The agent learns through experience replay and updates its Q-values.

Setup and Architecture

The experiments are conducted using the CartPole-v1 environment. The agent's task is to balance a pole on a cart by choosing 2 actions: moving the cart to the left or to the right. Each episode ends when the pole angle exceeds a certain threshold or the cart moves out of bounds, or after 500 steps if the pole is successfully balanced.

To approximate the Q-values, a fully connected feedforward neural network is used. The model is really simple, as the main goal of this exploration is to see how deep Q-learning works with different strategies and hyperparameters.

```python
class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 24),
            nn.ReLU(),
            nn.Linear(24, 24),
            nn.ReLU(),
            nn.Linear(24, output_dim)
        )

    def forward(self, x):
        return self.model(x)
```

Figure 45 – DQN code

As you can see, the model is simple with 2 hidden layers with 24 neurons with ReLU activation. Less neurons is suitable for "easier" tasks like CartPole, so the architecture is intentionally lightweight to match the simplicity of CartPole and to allow fast experimentation.

Then, I needed a Replay Buffer. The Replay Buffer stores state, action, reward, next_state, done. It ensures that updates are based on randomized, de-correlated samples.

Lastly, I implemented a training function that handles the learning process of the agent by repeatedly interacting with the environment and updating the neural network. During training, the agent stores its experiences in the replay buffer and begins learning only after the buffer has enough samples. To ensure stable learning, gradient clipping is applied to prevent large updates, and a separate target network is used to generate consistent targets. This target network

is synchronized with the main network at regular intervals. The function also tracks the training loss and average Q-values per episode, and supports both epsilon-greedy and Boltzmann exploration strategies. (All code can be found in the DQL.ipynb notebook in the repository)

Exploration Strategies and Hyperparameters

I decided on exploring 2 strategies – Epsilon-Greedy and Boltzmann across multiple configurations of hyperparameters to find the best one. The hyperparameters searches are run with 300 episodes to enable faster exploration.

Epsilon-Greedy

In the epsilon-greedy strategy, the agent balances exploration and exploitation by selecting a random action with probability ε (epsilon), and the best-known action (with the highest Q-value) otherwise. To explore better, I tested with different configurations and plotted the average reward per episode. I also ran a hyperparameter search to find the best configuration.

```
configs_eps = [
    # config 1
    {'learning_rate': 0.01, 'batch_size': 32, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.995, 'gamma': 0.99, 'tau': 1.0},
    # config 2
    {'learning_rate': 0.001, 'batch_size': 64, 'epsilon_start': 1.0, 'epsilon_min': 0.1, 'epsilon_decay': 0.99, 'gamma': 0.98, 'tau': 1.0},
    # config 3
    {'learning_rate': 0.005, 'batch_size': 16, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.98, 'gamma': 0.99, 'tau': 1.0},
    # config 4
    {'learning_rate': 0.002, 'batch_size': 64, 'epsilon_start': 1.0, 'epsilon_min': 0.1, 'epsilon_decay': 0.97, 'gamma': 0.97, 'tau': 1.0},
    # config 5
    {'learning_rate': 0.0005, 'batch_size': 32, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.99, 'gamma': 0.95, 'tau': 1.0},
    # config 6
    {'learning_rate': 0.005, 'batch_size': 48, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.98, 'gamma': 0.985, 'tau': 1.0}
]
```

Figure 46 – Configurations for hyperparameters for Epsilon-Greedy

As you can see from the figure above, each configuration varied in all values. I then plotted the training results for each configuration. The figure below shows the results.

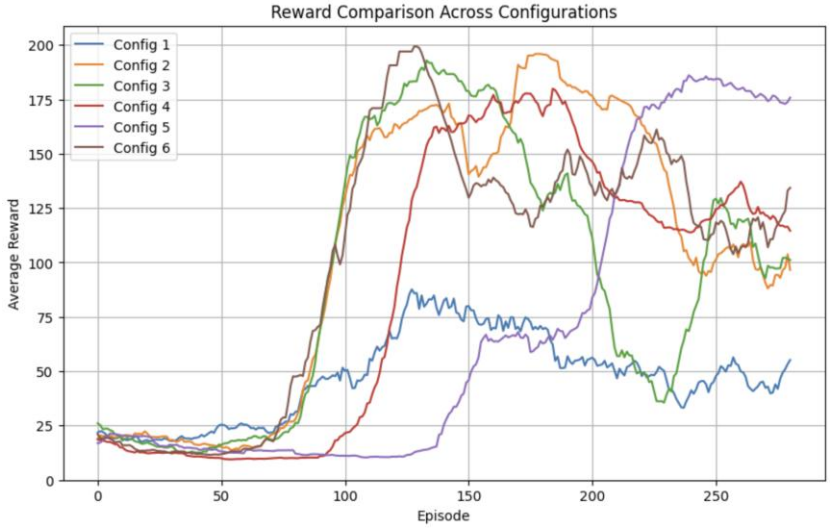

Figure 47 – Reward Comparison Across Configurations for Epsilon-Greedy

From the comparison graph, config 5 ({'learning_rate': 0.0005, 'batch_size': 32, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.99, 'gamma': 0.95, 'tau': 1.0}), ultimately achieves the most stable and high performance, despite a delayed start compared to the others. This outcome shows that early performance does not always indicate long-term learning quality.

Boltzmann

Boltzmann exploration uses a softmax function over Q-values to derive a probability distribution over actions. This means the agent is more likely to choose actions with higher Q-values, but it still occasionally explores suboptimal actions, especially when the temperature parameter τ (tau) is high. I again experimented with 6 different configurations and found the best performing one.

```
configs_boltz = [
    # config 1
    {'learning_rate': 0.01, 'batch_size': 32, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.995, 'gamma': 0.99, 'tau': 1.0},
    # config 2
    {'learning_rate': 0.005, 'batch_size': 64, 'epsilon_start': 1.0, 'epsilon_min': 0.1, 'epsilon_decay': 0.99, 'gamma': 0.98, 'tau': 0.5},
    # config 3
    {'learning_rate': 0.001, 'batch_size': 16, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.98, 'gamma': 0.99, 'tau': 0.7},
    # config 4
    {'learning_rate': 0.002, 'batch_size': 64, 'epsilon_start': 1.0, 'epsilon_min': 0.1, 'epsilon_decay': 0.97, 'gamma': 0.97, 'tau': 0.8},
    # config 5
    {'learning_rate': 0.0005, 'batch_size': 32, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.99, 'gamma': 0.95, 'tau': 0.6},
    # config 6
    {'learning_rate': 0.005, 'batch_size': 48, 'epsilon_start': 1.0, 'epsilon_min': 0.05, 'epsilon_decay': 0.98, 'gamma': 0.985, 'tau': 0.9}
]
```

Figure 48 – Configurations for hyperparameters for Boltzmann

As you can see from the figure above with the configurations, here also the temperature is different to experiment with low and high tau values.

In the resulting comparison plot, config 2 ({'learning_rate': 0.005, 'batch_size': 64, 'epsilon_start': 1.0, 'epsilon_min': 0.1, 'epsilon_decay': 0.99, 'gamma': 0.98, 'tau': 0.5}) demonstrated the best learning progress out of them all, slowly increasing and not fluctuating as much as the other configurations. This experiment helped me see how Boltzmann reacts to different tau values and the importance of them. Here, the best configuration consists with 0.5 tau value, which suggests that it leans more toward exploitation than exploration, but still keeps some stochasticity. It suggests that the agent benefits from being more confident in high-value actions, but not going fully greedy (like epsilon = 0).

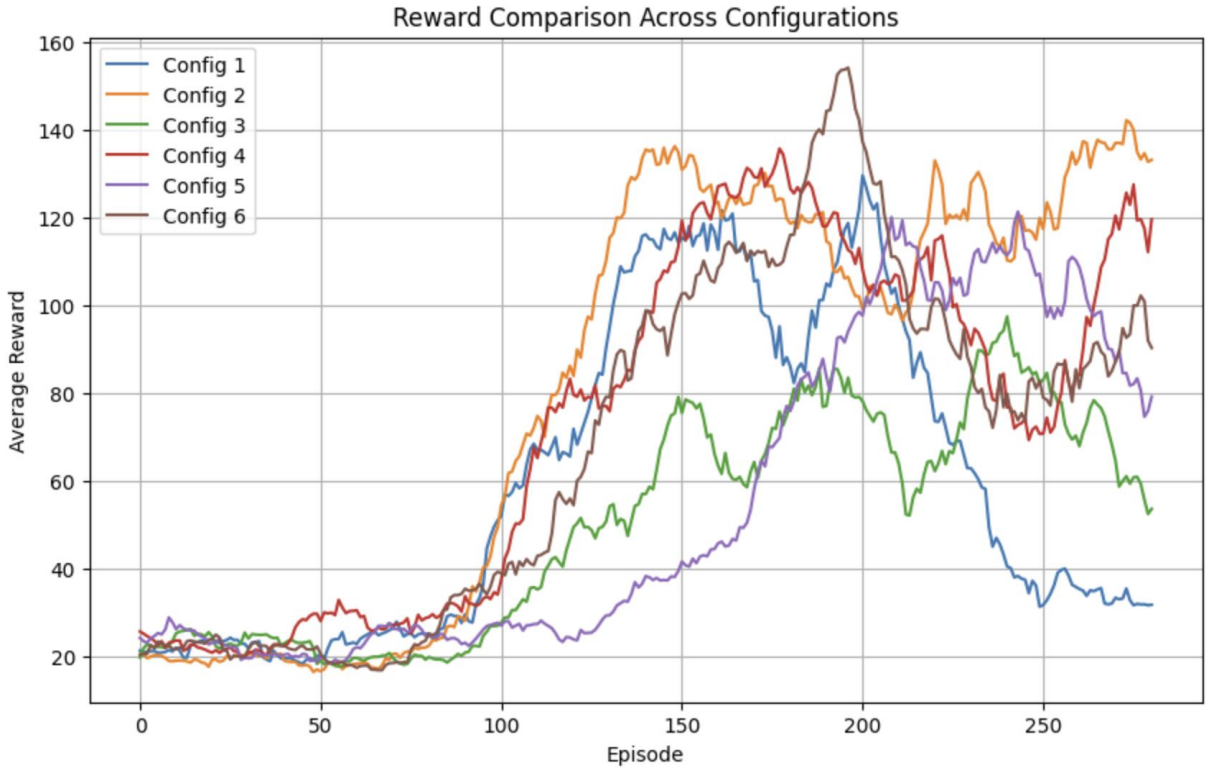

Figure 49 – Reward Comparison Across Configurations for Boltzmann

Results and Analysis

After finding the most optimal configuration parameters for Epsilon-Greedy and Boltzmann, I trained 1 new model for each strategy, but this time on 1000 episodes, as these are the final models we are going to have in this experiment and this way we can assess long-term performance.

First, I plotted again the average reward on the y-axis and the episode on the x-axis. Both agents improved over time, but with notable differences in consistency. The Epsilon-Greedy agent reached stable performance between 150–200 average reward, while the Boltzmann agent initially achieved similar levels but eventually showed instability and fluctuations in reward.

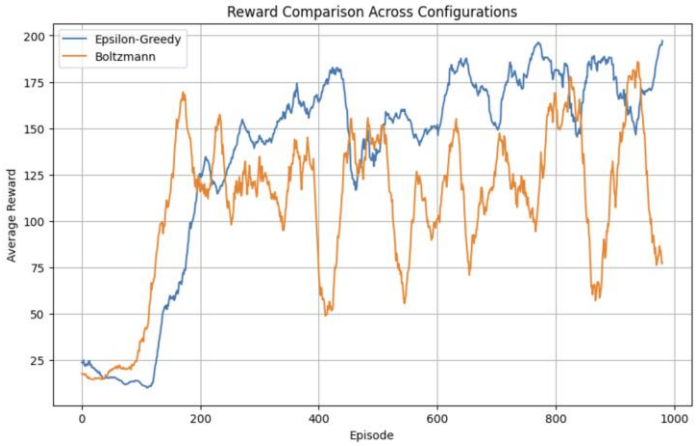

Figure 50 – Reward Comparison Across Configurations

The training loss plot below highlights a clear difference in stability. The Epsilon-Greedy agent maintained relatively low and stable loss values, while the Boltzmann agent experienced a prolonged increase in loss magnitude. This may suggest that the Boltzmann strategy, especially with lower temperature, encourages more exploitation, which can cause aggressive updates and instability.

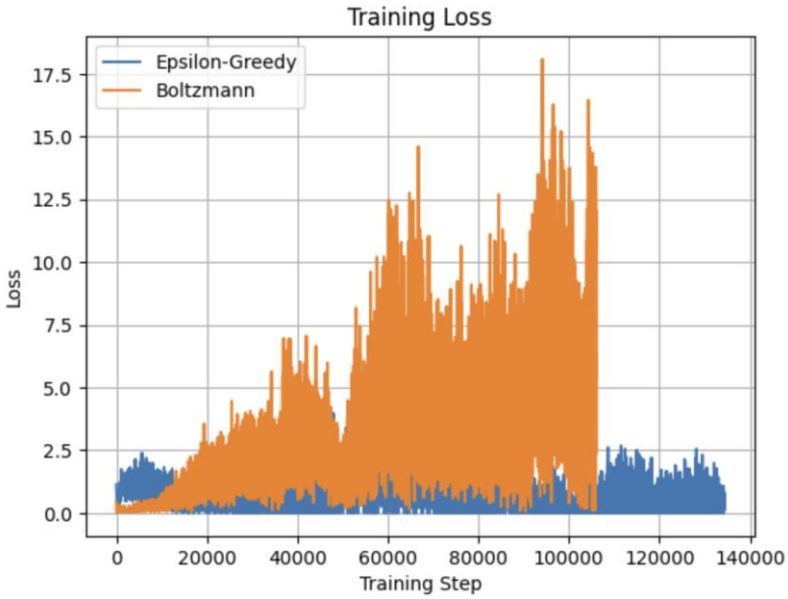

Figure 51 – Training Loss

Next, I plotted the average Q-values per episode for both agents. Both agents begin with low Q-values due to the lack of prior knowledge, but over time, their estimates improve. The Epsilon-Greedy agent shows a steady and conservative increase. In contrast, the Boltzmann agent exhibits a much steeper and continuous rise, eventually surpassing an average Q-value of 40. This rapid increase indicates that the Boltzmann agent gains more confidence in its learned policy.

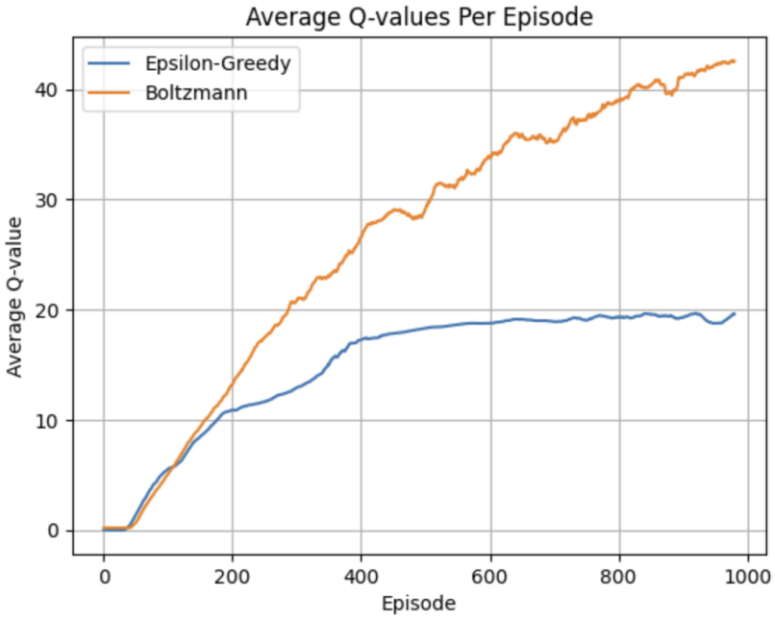

Figure 52 – Average Q-Values Per Episode

Finally, I created a function that evaluates the agents. During training we use exploration strategies like Epsilon-Greedy and Boltzmann to discover new actions and states, helping the agent learn better policies. But during evaluation, we want to measure how well the agent performs when it follows its learned policy without randomness.

The Epsilon-Greedy agent performed well, achieving rewards between 161 and 347 across 10 episodes, indicating that it learned a reasonably strong policy, though not optimal. In contrast, the Boltzmann agent achieved perfect scores of 500 in all 10 evaluation episodes. This shows that the agent not only learned the optimal policy but was also able to execute it consistently and with high confidence. However, it is important to note that during training, the Boltzmann agent showed signs of Q-value overestimation, which could lead to instability in more complex environments. While this did not negatively affect its performance in CartPole, applying additional techniques like Double DQN could help reduce overestimation and improve the robustness of the agent.

```
Epsilon-Greedy Agent:
Episode 1: Total Reward = 218.0
Episode 2: Total Reward = 265.0
Episode 3: Total Reward = 347.0
Episode 4: Total Reward = 279.0
Episode 5: Total Reward = 188.0
Episode 6: Total Reward = 194.0
Episode 7: Total Reward = 216.0
Episode 8: Total Reward = 161.0
Episode 9: Total Reward = 311.0
Episode 10: Total Reward = 300.0

Boltzmann Agent:
Episode 1: Total Reward = 500.0
Episode 2: Total Reward = 500.0
Episode 3: Total Reward = 500.0
Episode 4: Total Reward = 500.0
Episode 5: Total Reward = 500.0
Episode 6: Total Reward = 500.0
Episode 7: Total Reward = 500.0
Episode 8: Total Reward = 500.0
Episode 9: Total Reward = 500.0
Episode 10: Total Reward = 500.0
```

Figure 53 – Final Agent Evaluation Results

Overall, both agents successfully learned to balance the pole, with Boltzmann outperforming Epsilon-Greedy in terms of final evaluation. However, Epsilon-Greedy offered a more interpretable and stable learning process. This suggests that exploration strategies should be selected not only based on performance but also based on training robustness and the problem environment.

Conclusion

Through this experiment, I gained hands-on experience with deep Q-learning by using a neural network and a replay buffer and explored how different strategies and hyperparameter configurations influence an agent's learning behaviour. I learned the importance of balancing exploration and exploitation, and how tuning parameters like learning rate, gamma, and temperature directly affect performance and stability.