AI Applications Capstone Project

Problem Statement

This project aims to develop and implement a self-navigating agent capable of finding optimal paths through arbitrarily sized maze environments, where complete knowledge of state transitions and rewards is unavailable.

Background and Context

The course

During this course we have covered the Basics of Reinforcement Learning and the following methods:

- Markov Decision Process/Property
- Dynamic Programming
- Monte Carlo
- Temporal Difference
- SARSA (State-Action-Reward-Action)
- Q-Learning

The following subchapters of Background and Context will go through the origins of these methods and, in increasingly greater detail, how the methods are applied.

Defining Artificial Intelligence and Reinforcement Learning

John McCarthy is documented to be the person that coined the term Artificial Intelligence. He, along with Marvin Minsky, Nathaniel Rochester, and Claude Shannon, submitted a proposal, called A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence[1], in August 1955 for the Dartmouth College Conference. The start of the proposal states

> The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.

Six years later, in 1961, the term reinforcement learning was coined by Minsky in 1961 in his paper titled Steps toward Artificial Intelligence[2].

> The problems of heuristic programming-of making computers solve really difficult problems-are divided into five main areas: Search, Pattern-Recognition, Learning, Planning, and Induction.

> A computer can do, in a sense, only what it is told to do. But even when we do not know how to solve a certain problem, we may program a machine (computer) to Search through some large space of solution attempts. Unfortunately, this usually leads to an enormously inefficient process.
>
> With Pattern-Recognition techniques, efficiency can often be improved, by restricting the application of the machine's methods to appropriate problems.
> Pattern-Recognition, together with Learning, can be used to exploit generalizations based on accumulated experience, further reducing search. By analyzing the situation, using Planning methods, we may obtain a fundamental improvement by replacing the given search with a much smaller, more appropriate exploration. To manage broad classes of problems, machines will need to construct models of their environments, using some scheme for Induction.
>
> Wherever appropriate, the discussion is supported by extensive citation of the literature and by descriptions of a few of the most successful heuristic (problem-solving) programs constructed to date.

In this paper he goes into what is a Learning System, here is the summary;

> In order to solve a new problem, one should first try using methods similar to those that have worked on similar problems. To implement this "basic learning heuristic" one must generalize on past experience, and one way to do this is to use success-reinforced decision models. These learning systems are shown to be averaging devices. Using devices which learn also which events are associated with reinforcement, i.e., reward, we can build more autonomous "secondary reinforcement" systems. In applying such methods to complex problems, one encounters a serious difficulty-in distributing credit for success of a complex strategy among the many decisions that were involved. This problem can be managed by arranging for local reinforcement of partial goals within a hierarchy, and by grading the training sequence of problems to parallel a process of maturation of the machine's resources.

With that there is a reference to Skinner’s work and its importance in the role of reinforcement learning.

> The analogy is with "reward" or "extinction" (not punishment) in animal behavior. The important thing about this kind of process is that it is "operant" (a term of Skinner [44]); the reinforcement operator does not initiate behavior, but merely selects that which the Trainer likes from that which has occurred

Here is the original schema for

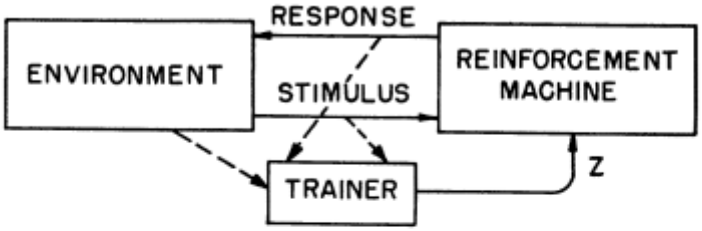


Fig. 8—Parts of an "operant reinforcement" learning system. In response to a stimulus from the environment, the machine makes one of several possible responses. It remembers what decisions were made in choosing this response. Shortly thereafter, the Trainer sends to the machine positive or negative reinforcement (reward) signal; this increases or decreases the tendency to make the same decisions in the future. Note that the Trainer need not know how to solve problems, but only how to detect success or failure, or relative improvement; his function is selective. The Trainer might be connected to observe the actual stimulus-response activity or, in a more interesting kind of system, just some function of the state of the environment.

From animals to agents

Sutton and Barto see the history of reinforcement learning as two threads, one starting with animal psychology and the other in "optimal control". Whilst I fully agree, I am given a relatively short background and context here so I treat them as one, joining them immediately with Minsky's reference to Skinner. Links are provided for further information on the respective threads.

Animal psychology through to human behaviour

Clearly Minsky was influenced by Skinner and his 1953 book, Science and Human Behaviour (a version from 2014 is available at the BK Skinner Foundation[3]). Influences can be traced back to work in animal psychology and behaviour by Edward Throndike (late 1890s) [source is The Alignment Problem by Brian Christian[4]].

Edward Thorndike was quite a character, he lived with the animals he studied, and his work bridged the gap between psychology and behaviourism (later developed by Skinner and others). He defined The Law of Effect[5], which can clearly be linked to modern day reinforcement learning

> … responses that produce a satisfying effect in a particular situation become more likely to occur again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation.

Taking a step back, the Mathematics of the field is rooted in Markov Chains[6] developed by a Russian Mathematician Andrey Markov in 1906. The key property of a Markov Chain is that the probability of each event only depends on the probability of the previous event. This can be rephrased to "what happens next is dependent only on the current state".

Dynamic Programming

In the 1950s, before McCarthy and Minsky respectively coined the terms Artificial Intelligence and Reinforcement Learning, Richard Bellman furthered the field of Control Theory[7] with Dynamic Programming[8].

Dynamic Program allows for the decomposition of problems into subproblems and then solving them in a recursive manner. In which the eponymous Bellman Equation[9] is critical to understand the value of a subproblem and therefore all other subproblems and the value of the whole problem.

The Bellman Equation

The Bellman Equation defines the value of a state as the maximum possible sum of immediate reward and discounted future values, weighted by their transition probabilities.

$$V(s) = \max_a[R(s,a) + \gamma\sum P(s'|s,a)V(s')]$$

Where:

- V(s): Value function for state s
- γ: Discount factor
- $\max_a$: Optimization

Components:

- Immediate reward: R(s,a)
- Future value: γ∑P(s'|s,a)V(s')
- Transition probabilities: P(s'|s,a)

The Bellman Equation is critical to Dynamic Programming as it is a formal Mathematical method for calculating the value of the next state. In an optimal system, that is one where all of the environment is known, a Markov Decision Process[10] can be applied to calculate the value of all states.

Markov Decision Process

A mathematical framework for modeling decision-making where outcomes are partly random and partly under the control of a decision maker.

Where:

- S: Set of states
- A: Set of actions
- P(s'|s,a): State transition probability
- R(s,a): Reward function

Components:

- State space: All possible states
- Action space: Available actions
- Transition model: P(s'|s,a)
- Reward function: R(s,a,s')

Key elements of Reinforcement Learning's application of the Markov Decision Process are the Value Iteration[11] and Policy Iteration[12]. The value iteration algorithm will iteratively update the value function for each state until it converges on the optimal value function. The Policy Iteration alternates between policy evaluation and policy improvement, these are repeated until the policy converges to the optimal policy.

Value Iteration

Iteratively computes optimal state values by maximizing immediate rewards plus discounted future values.

$$V_{k+1}(s) = \max_a[R(s,a) + \gamma\sum P(s'|s,a)V\_k(s')]$$

Where:

- $V_{k+1}(s)$: Value at iteration k+1
- $\max_a$: Optimal action selection
- k: Iteration step

Components:

- Current estimate: $V_k(s)$
- Next estimate: $V_{k+1}(s)$
- Action-value: R(s,a)

Policy Iteration

Alternates between evaluating current policy and improving it until convergence.

$$\underline{\text{Evaluation}}: V_{k+1}(s) = R(s,\pi(s)) + \gamma\sum P(s'|s,\pi(s))V\_k(s')$$

$$\underline{\text{Improvement}}: \pi'(s) = \text{argmax}_a[R(s,a) + \gamma\sum P(s'|s,a)V(s')]$$

Where:

- $\pi(s)$: Current policy
- $\pi'(s)$: Improved policy
- $V_k(s)$: Value estimate at step k

Components:

- Policy evaluation term: $R(s,\pi(s))$
- Future value term: $\gamma\sum P(s'|s,\pi(s))V_k(s')$
- Policy selection: $\text{argmax}_a[...]$

So here's a reminder of the problem statement

> This project aims to develop and implement a self-navigating agent capable of finding optimal paths through arbitrarily sized maze environments, where complete knowledge of state transitions and rewards is unavailable.

Is Dynamic Programming enough to solve our problem? No.

The approaches are insightful and build a vital understanding of Reinforcement Learning however there are two key elements of the problem statement that the Dynamic Programming approaches (MDP, Value/Policy Iteration) we have looked at so far cannot achieve.

Partially observable

To achieve successful learning with a model-based approach:

- Complete knowledge of state transitions P(s'|s,a)
- Full reward function R(s,a)
- All possible states and actions

However the problem statement does not include any uncertainty, so whilst the above methods are not able to achieve this, it would not be a blocker for using them.

Scaling

The problem statement clearly references arbitrarily sized maze environments. With Dynamic Programming and Model-based methods, when the State and Action spaces become large the computational complexity grows rapidly making it less suitable for large-scale problems. This excludes them from the solution.

This leads us to model-free methods: SARSA, and Q-Learning, which learn directly from experience rather than requiring a complete model. Before that I'll take a moment to discuss two leading figures in reinforcement learning and the basics that have come from their work.

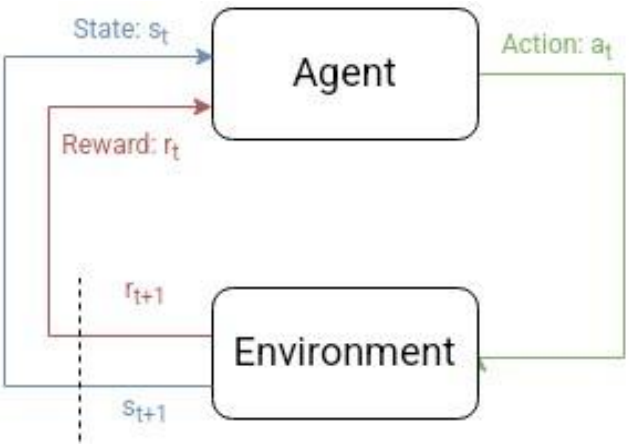
Reinforcement learning's founding fathers

Richard Sutton[13] and Andrew Barto have had a long collaboration which has defined the field of reinforcement learning. In 1998 they wrote the "bible" for Reinforcement learning, they have since updated it with a second edition in 2018, it is available on Sutton's site Reinforcement Learning: An Introduction - second edition[14].

Basics of Reinforcement learning

Reinforcement learning includes the following components:

- **Agent**: a model that perceives the environment
- **Environment**: real or virtual world in which the agent acts.
- **Action**: interactions by the agent on the environment
- **State**: configurations of the environment that the agent can perceive.
- **Reward**: is the value that the agent tries to maximise not immediately but in the future.



These components interact in a continuous manner, forming what is known as the Agent-Environment interface, as shown in figure above. At each time step t the Agent:

- Receives a state (s_t) from the Environment
- Choses an action based on s_t
- Receives reward r_{t+1} and a new state s_{t+1}
- Updates its knowledge based on its experience

Importance of reward

The reward hypothesis, fundamental to reinforcement learning, states that all goals can be described by maximisation of expected cumulative reward. This relates directly to Thorndike's Law of Effect - actions leading to positive outcomes become more likely.

The reward function must be carefully designed as it directs the agent's behaviour. A poorly designed reward can lead to unexpected behaviour and undesired outcomes.

Policy

Policies (π) are referred to a lot in reinforcement learning, to add context I share the definition from Sutton and Barto:

> A policy defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states.

Policies can be:

- Deterministic
- Stochastic
- Optimal

Of course a policy can be a mix of Optimal and Deterministic or Stochastic.

Episodes

Episodes are an important part of reinforcement learning, as learning occurs over multiple discrete episodes. An episode represents a series of interactions in the Agent-Environment interface for each time step t (as described above), starting with the initial state and finishing in a terminal state (success or failure).

The episodic approach to learning allows the Agent to:

- Reset the environment to its original state
- Learn from both successful and unsuccessful episodes
- Persistently store its learning from each episode
- Develop increasingly optimal strategies

Exploration vs Exploitation

The balance between exploration (looking for new rewards) and exploitation (taking known rewards) is very important in reinforcement learning. Simply, if an agent always takes the best known reward there is no guarantee that it will find the optimal solution. This is like the local minima problem in gradient descent, there could be a better path just out of reach.

The epsilon-greedy strategy addresses this by:

- Taking random action with a probability ε (typically 5-10% of the time)
- Following the best known solution with a probability 1-ε.
- Gradually decreasing ε over time (decay) to favor exploitation over exploration

On-Policy and Off-Policy Learning

On-Policy Learning

- Agent learns the value of the policy it's currently following
- Updates are based on actions actually taken
- More stable but potentially slower to converge
- Example: SARSA (State-Action-Reward-State-Action)

Off-Policy Learning

- Agent learns about optimal policy while following different behavioral policy
- Updates based on hypothetical actions (what could have been optimal)
- Can learn optimal policy while exploring
- Example: Q-Learning

Temporal-Difference (TD) Learning

Temporal-difference learning is comparatively small compared to other aspects we have explored here, however its impact is not proportional to its size! It has had a significant impact on not just the fields of Artificial Intelligence and Reinforcement Learning but also the fields of Neuroscience and Psychology.

In 1996 Dayan, Montague, and Sejnowski[15] connected Wolfram Schultz's work with Dopamine release in monkeys to the Temporal-difference algorithm. There has been further papers on the connection of Dopamine as the "Temporal-difference error" including one from Deepmind in 2020:

- Dopamine and temporal difference learning: A fruitful relationship between neuroscience and AI[16] (Deepmind 2020)

Sutton and Barto stated this about temporal-difference in their book

> Temporal-difference learning methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity—for example, of the probability of winning in the tic-tac-toe example.

An analogy I like is the Estimated Time of Arrival (ETA) on an application like Google Maps. The closer to the destination you get the more confident you can be in the predicted ETA.

Sarsa (State-Action-Reward-State-Action)

Sarsa is an on-policy temporal-difference learning algorithm. It updates Q-values based on actions actually taken by the current policy.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$$

Where:

- $Q(s_t,a_t)$: Current state-action value
- α: Learning rate
- γ: Discount factor
- R_{t+1}: Immediate reward
- $Q(s_{t+1},a_{t+1})$: Next state-action value

Q-Learning

Q-Learning is an off-policy temporal-difference learning algorithm that updates the Q-values with the maximum possible value for the next state.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[R_{t+1} + \gamma \cdot \max_a[Q(s_{t+1},a)] - Q(s_t,a_t)]$$

The main difference between Sarsa and Q-Learning make Q-Learning a strong candidate for solving the problem. The advantages that Q-Learning has are:

- Uses the max future Q-value regardless of policy
- Can learn optimal policy whilst following an exploratory policy
- More efficient at finding the optimal policy

It may however be less stable than Sarsa during training.

Q-Learning was created by Chris Watkins in 1989 as part of his PhD dissertation Learning from Delayed Rewards[17].

With that in mind I will finish this section with Chris Watkins words on Q-Learning

> It occurred to me to consider reinforcement learning as a form of incremental dynamic programming. My old 1989 PhD thesis "Learning from Delayed Rewards", introduced a model of reinforcement learning (learning from rewards and punishments) as incrementally optimising control of a Markov Decision Process (MDP), and proposed a new algorithm – which was dubbed "Q-learning" – that could in principle learn optimal control directly without modelling the transition probabilities or expected rewards of the MDP. The first rigorous proof of convergence was in Watkins and Dayan (1992). These innovations helped to stimulate much subsequent research in reinforcement learning. The notion that animals, or 'learning agents' inhabit a MDP, or a POMDP, and that learning consists of finding an optimal policy, has been dominant in reinforcement learning research since

A journey that started with Edward Thorndike observing animals in his apartment through to computer scientists working on Control Theory, has come full circle back to the way that animals learn.

Other options not considered

The Q-Learning learning algorithm dates back to 1989 and there have been significant advancements in reinforcement learning since then. This is particularly true with Deep Reinforcement Learning, the pairing of Neural Networks with algorithms like Q-Learning. This is outside the scope of the course syllabus so has not been considered.

I have not included any detail on the Monte Carlo method. The reason is simple, from my readings and learnings on the Monte Carlo Markov Chain (the method relevant to this course), I have found that it is less appealing for the problem. In general terms, I have seen the process as like a sledgehammer where the agent runs through every possible approach and assigns a score. Q-Learning seems more efficient and elegant for this problem.

This does lead me to think that, time permitting, I should try it with a MCMC in the future.

Objectives

- Develop a Q-Learning agent for maze navigation
- Create a scalable maze environment
- Evaluate the agent's learnings:
    1. Baseline - untrained compared to trained agents
    2. Hyperparameters - Experiment with changes to the model's hyperparameters (learning rate, epsilon-greedy value, discount rate of future rewards).
    3. Scale - maze sizes from 5 to 50
- Analyse the results
- Present findings

Methodology

This section outlines the experiment's approach to validate Q-Learning for maze navigation.

Experiment design

The experiments follow a systematic approach to evaluate the agent's performance across different maze sizes and hyperparameters. The first thing to verify is that a trained agent is better than an untrained agent.

Control Tests

- Baseline runs with untrained agent
- Optimal path calculation by hand for comparison (when feasible)
- Multiple episodes per configuration to account for randomness

Training evaluation

Next will be evaluation of training an agent with different parameters - changing either the training algorithm's hyperparameters or the environment's size.

The following data will be collected:

- Steps to reach the goal
- Episodes until convergence
- Final path success

Hyperparameter Testing

To evaluate which are the optimal hyperparameters, tests will be run changing the following values from a hyperparameter matrix, thus ensuring that each combination is tested.

- Evaluation of training for learning rates: [0.1, 0.01, 0.001]
- Evaluation of training for discount factors: [0.9, 0.95, 0.99]
- Evaluation of training for epsilon-greedy values: [0.1, 0.2, 0.3]

Scalability Testing

To evaluate that the agent is able to manage increasing larger environments the tests include mazes of increasing size.

- Evaluation of training for mazes of sizes: [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

Success Criteria

The agent's performance will be considered successful if it:

- Converges on a stable policy
- Consistently finds the optimal path to the goal

Data Collection

Each experiment captures:

- Size of the environment
- Number of episodes
- Steps per episode
- Reward per episode
- Training duration
- Hyperparameters used

Implementation

Reinforcement learning algorithm

Q-learning (Off-policy TD Control) for Estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+$, $a \in A(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:
1. Initialize S
2. Loop for each step of episode:
    - Choose A from S using policy derived from Q (e.g., ε-greedy)
    - Take action A, observe R, S'
    - Update Q: $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
    - $S \leftarrow S'$
3. Until S is terminal

Environment design

In the first set of experiments I will use a fully observable environment. This will be a generated and solvable maze. The Agent will have access to the next state for a given action from its current state. That will either be:

- 0 for open space
- 1 for a wall

The agent will know it has finished when it receives the goal reward.

Using Mazelib

Mazelib[18] is a well documented python library that is capable of using various algorithms to create solvable mazes.

The Sidewinder algorithm has been chosen for maze generation as it *quickly* creates varied, solvable mazes with configurable dimensions (see Appendix A for the test details).

Key properties:
- Binary grid representation (0=path, 1=wall)
- Single valid path between start/goal
- Customizable size
- Can set the start to the first valid cell in the grid
- Can set the goal to the last valid cell in the grid
- It favours a corridor at the top, when you view the whole maze it is easier to solve, making manual validation easier.

Agent design

Q-learning agent with:

- State: Current position in maze
- Actions: Up, down, left, right
- Reward structure (amounts set in the experiment):
    - Goal reached
    - Wall collision
    - Movement cost
- Learning parameters (amounts set in the experiment):
    - Epsilon-greedy exploration
    - Learning rate α for Q-value updates
    - Discount factor γ for future rewards
- Q-table: Maps state-action pairs to expected returns

Agent-Environment Control

The third class is Q-Learning Agent Control. It will be responsible for:

- Episodes
- Agent Training
- Agent Testing
- Visualisation of the training

Automated testing

The final class is for running the tests and collating the results.

Python Class Overview

The following are the “headers” that cover the functionality required for the solution. The full code is in the appendix.

```
class Config:
    """Configuration parameters for maze navigation system"""

    maze_params:
    - maze_size: int  # Grid dimensions

    agent_params:
    - learning_rate: float  # Step size α
    - discount_factor: float  # γ
    - epsilon: float  # Exploration rate
    - num_episodes: int  # Training episodes

    rewards:
    - goal_reward: float
```

```
    - wall_penalty: float
    - step_penalty: float

class MazeEnvironment:
    """Handles maze generation, state management and visualization"""

    Properties:
    - grid: np.array  # Binary maze from Sidewinder algorithm
    - start: tuple  # Fixed (1,1) starting position
    - end: tuple  # Goal at (height-2, width-2)
    - config: Config

    Methods:
    - generate()  # Creates maze using Sidewinder
    - get_state(position)  # Returns current position
    - get_reward(state, next_state)  # Calculates transition reward
    - is_valid_move(state)  # Checks move legality
    - visualize(path)  # Displays maze with optional path

class QLearningAgent:
    """Implements Q-learning algorithm"""

    Properties:
    - q_table: np.array  # Shape (height, width, 4) for actions
    - actions: List[Tuple]  # [(Up), (Down), (Left), (Right)]
    - config: Config
    - exploration_rate: float

    Methods:
    - get_action(state, training)  # ε-greedy action selection
    - update(state, action, reward, next_state)  # Q-value update

class AgentControl:
    """Manages training, testing and metrics visualization"""

    Properties:
    - env: MazeEnvironment
    - agent: QLearningAgent
    - config: Config
    - metrics: Dict  # Tracks rewards, steps, stability etc.

    Methods:
    - run_episode(training)  # Executes single episode
    - train(save_path)  # Training with live visualization
    - test(display)  # Tests trained agent
    - test_consistency(num_tests)  # Multiple test runs
    - _update_training_plots()  # Real-time metrics plotting
```

```
class ExperimentRunner:
    """Manages automated testing and reporting"""

    Properties:
    - base_config: Config
    - results: List
    - experiment_id: str  # Timestamp-based ID
    - base_path: Path  # Output directory

    Methods:
    - run_baseline_comparison(num_trials)  # Trained vs untrained
    - run_hyperparameter_sweep()  # Tests parameter combinations
    - run_scalability_test(sizes)  # Tests different maze sizes
    - save_report(results)  # Exports results to Excel
    - _run_untrained_tests(num_trials)  # Helper for baseline
```

Results

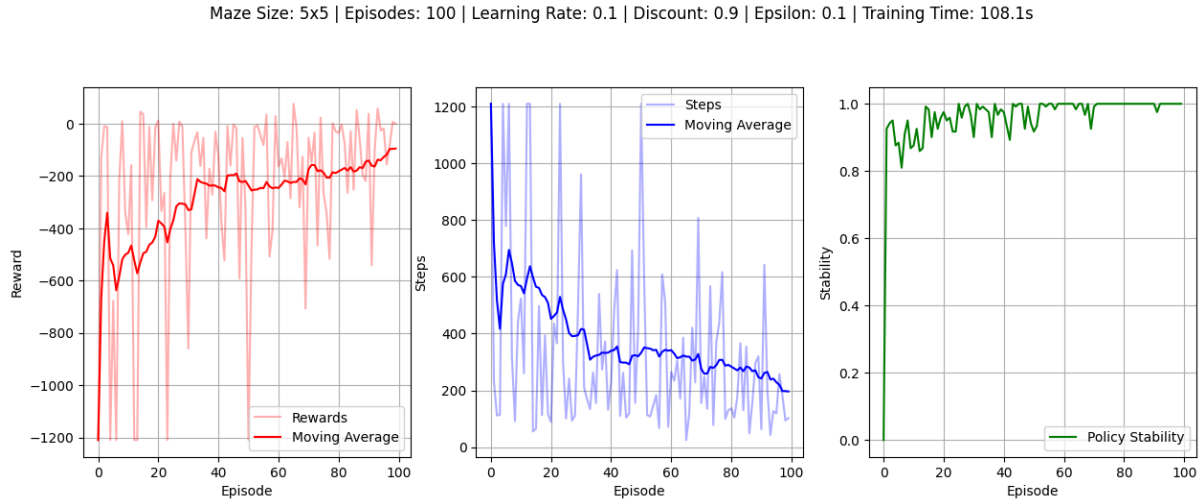
Control Test Results

To meet the experiment design requirements, the `test_consistency` method of the AgentControl class runs the tests X number of times and records the mean and standard deviation on the number of steps taken.

Results

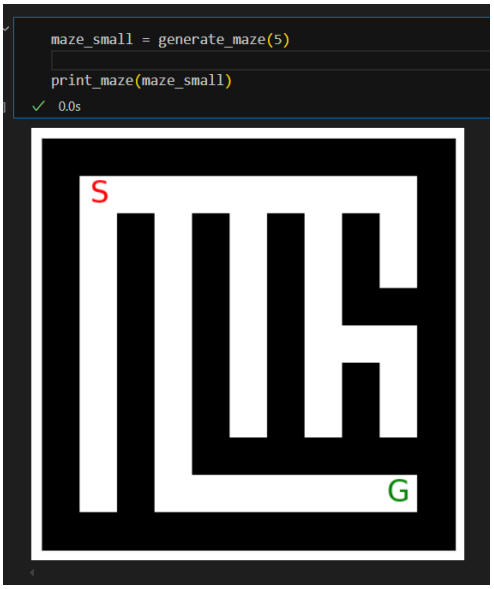
Metric	Untrained	Trained
Success Rate	0	1
Average Steps	1210	16
Std Steps	0	0

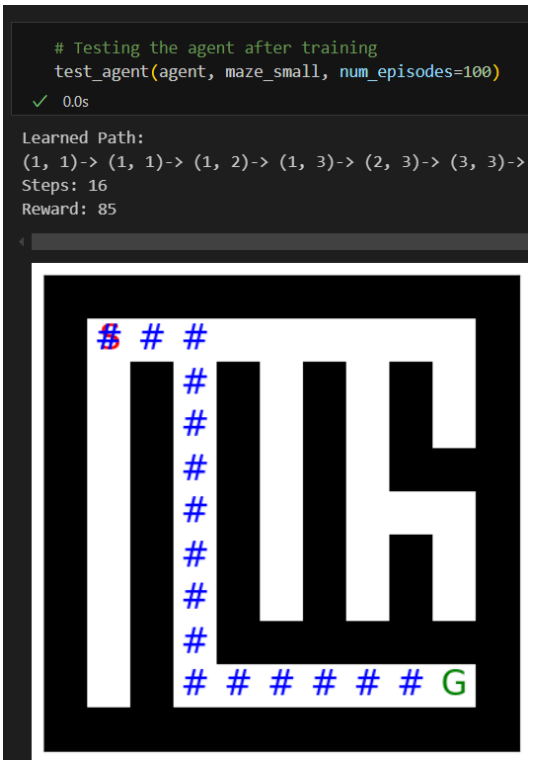
Training results for an agent in a 5x5 maze



Confirming the optimal path

The actual maze in the automated test is not available (a point to pick up on for improvements) it is however possible to review similar shaped mazes for confirmation that the route taken is indeed to the optimal one.





Hyperparamater testing

Maze size is 5x5

learning rate	discount factor	epsilon	success	avg steps	final policy stability
0.1	0.9	0.1	1	16	1.00
0.1	0.9	0.2	1	16	1.00
0.1	0.9	0.3	1	16	1.00
0.1	0.95	0.1	1	16	1.00
0.1	0.95	0.2	1	16	1.00
0.1	0.95	0.3	1	16	1.00
0.1	0.99	0.1	1	24	0.99
0.1	0.99	0.2	1	20	0.99
0.1	0.99	0.3	1	16	1.00
0.01	0.9	0.1	0	1210	0.96
0.01	0.9	0.2	0	1210	0.98

learning rate	discount factor	epsilon	success	avg steps	final policy stability
0.01	0.9	0.3	0	1210	0.92
0.01	0.95	0.1	0	1210	0.95
0.01	0.95	0.2	0	1210	0.89
0.01	0.95	0.3	0	1210	0.91
0.01	0.99	0.1	0	1210	0.98
0.01	0.99	0.2	0	1210	0.88
0.01	0.99	0.3	0	1210	0.90
0.001	0.9	0.1	0	1210	0.90
0.001	0.9	0.2	0	1210	0.89
0.001	0.9	0.3	0	1210	0.90

Scale testing

Experiments were conducted with the following hyperparameters:

- Learning rate: 0.1
- Discount rate: 0.9
- Epsilon value: 0.1

maze size	num of episodes	training time (seconds)	success rate	avg steps
5	25	29	0	1210
10	100	109	1	36
15	225	250	1	60
20	400	467	1	108
25	625	723	1	104
30	900	1080	1	128
35	1225	1482	1	168
40	1600	2011	1	204
45	2025	2647	1	228
50	2500	3259	1	268

Discussion

Control test

Clearly the training of the agent has a statistically positive effect on the agent's capability to navigate the maze.

It is possible to manually check that the agent is choosing the optimal path.

Due to the way the tests were implemented the check of optimal path needed to be done on a different maze. In a future experiment I would add the saving of the maze and path to the test results.

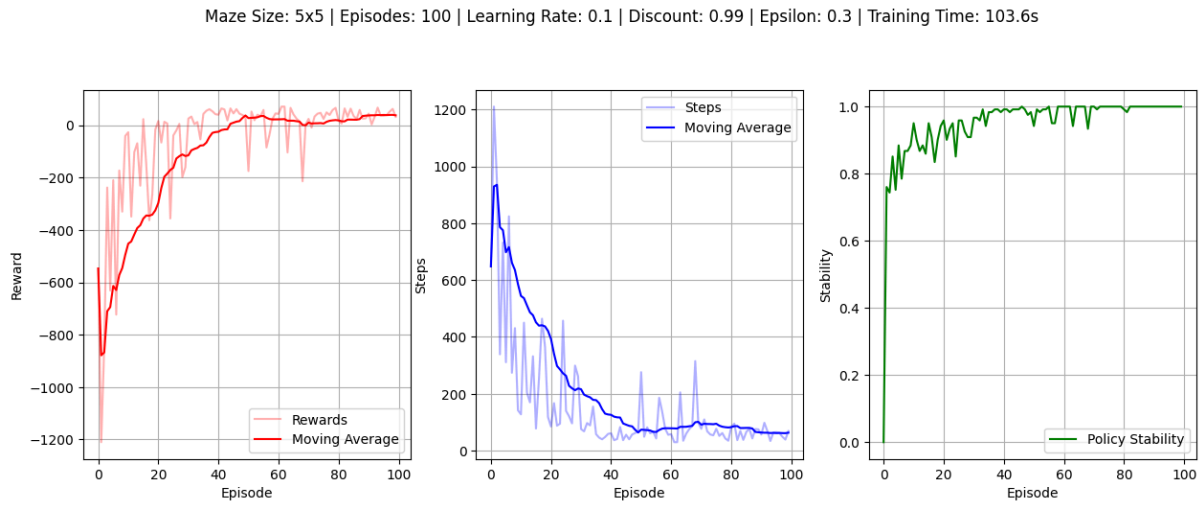
Hyperparamater testing

I include important visualisations here for ease of reference. Not all points have them, the visualisations are available in Appendix B.

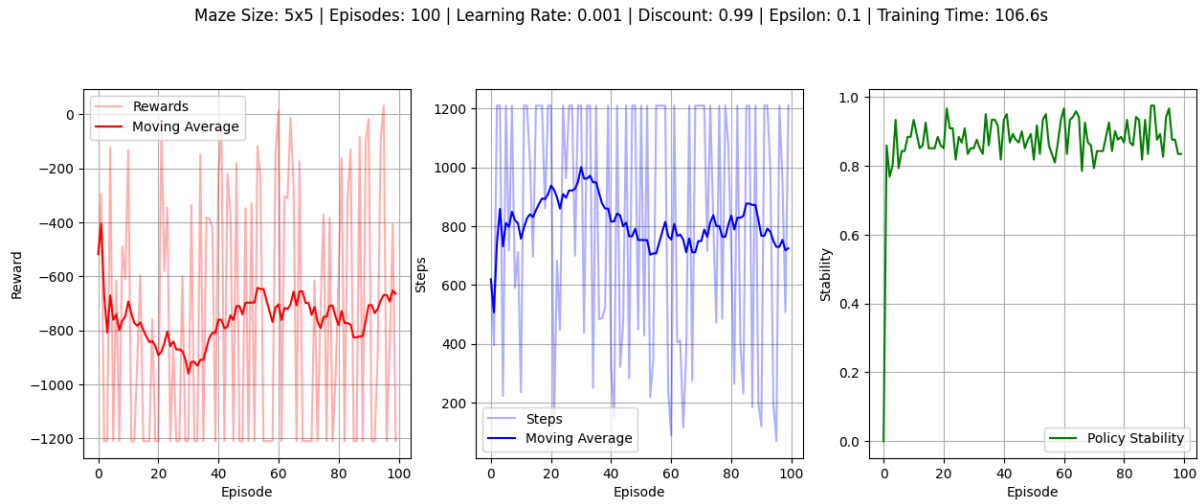
Learning rate

- A learning rate of 0.1 is the most optimal
- Other learning rates do not converge (within 100 episodes)

A training run for an agent that learnt the maze



A training run for an agent that did not learnt the maze

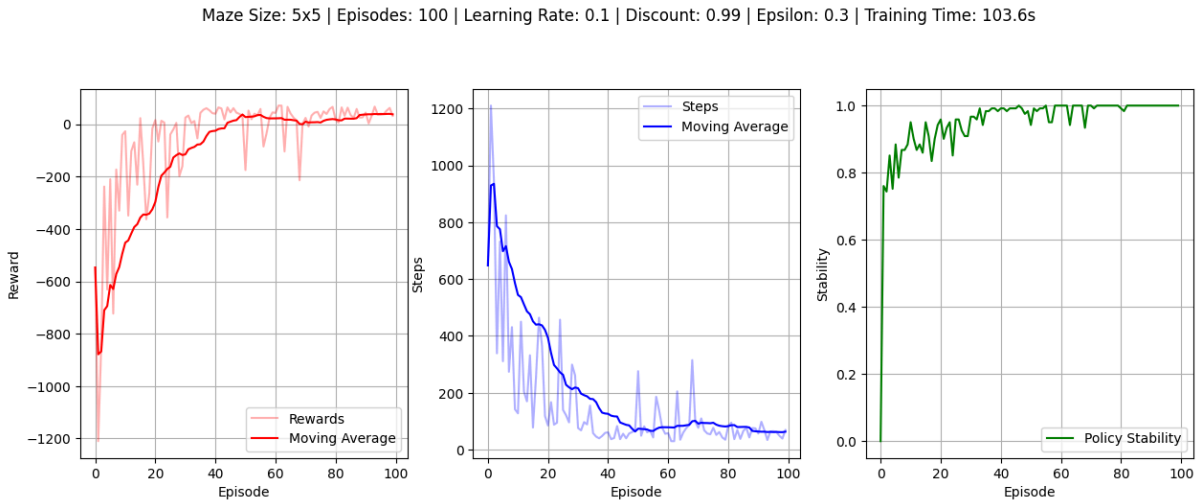


Discount rate

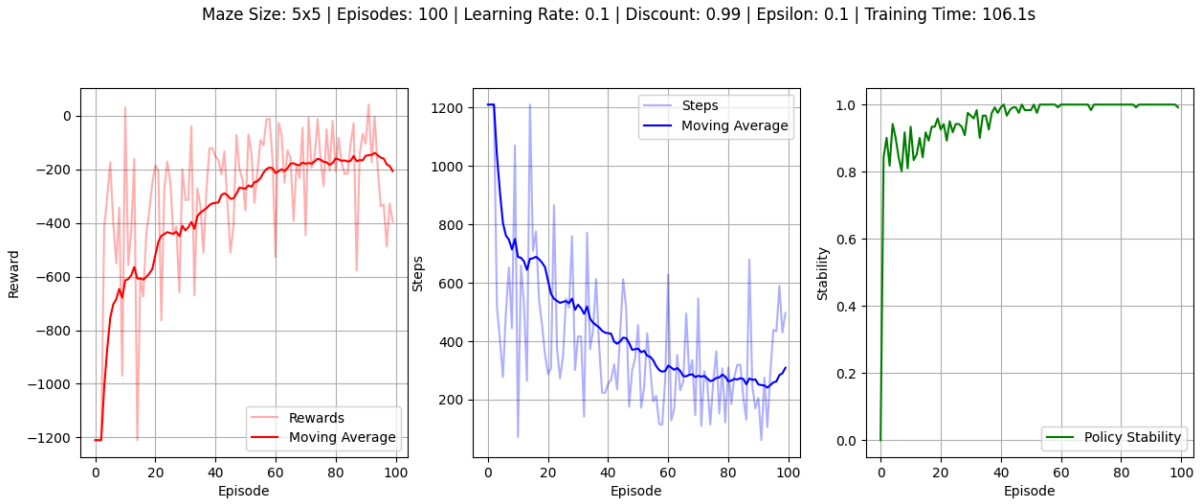
It is difficult to draw any conclusions from agents that were trained with a learning rate of 0.01 and 0.001. There is not sufficient data. As such the below analysis refers to results with a learning rate of 0.1

High discount (0.99)

- With high exploration (epsilon = 0.3) is quicker to converge (after approx 50 episodes of training)

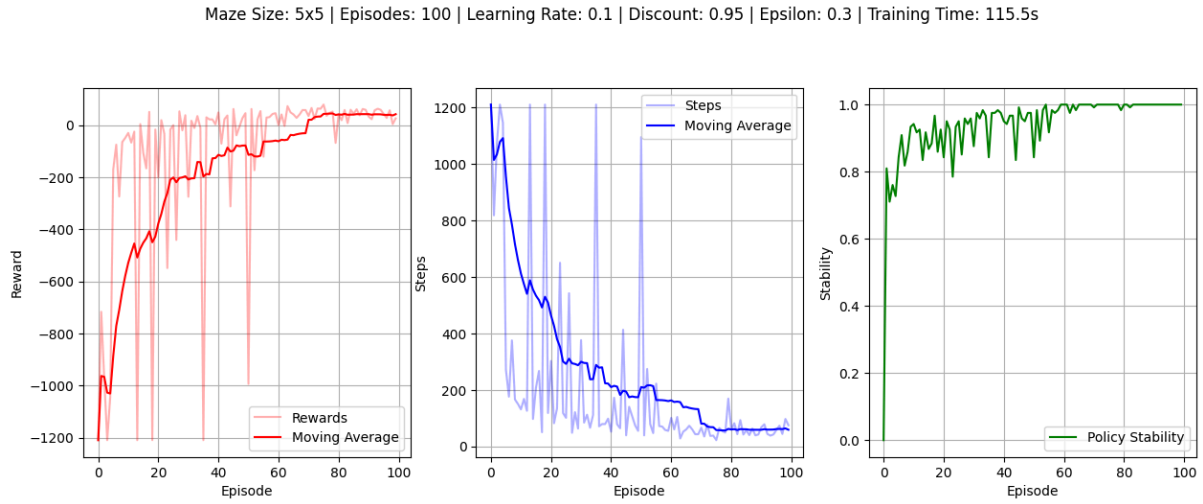


- With medium and low exploration (epsilon < 0.3) the agent does not converge on an optimal path within 100 episodes (it converges on a path of 20 and 24 steps respectively)



Medium discount (0.95)

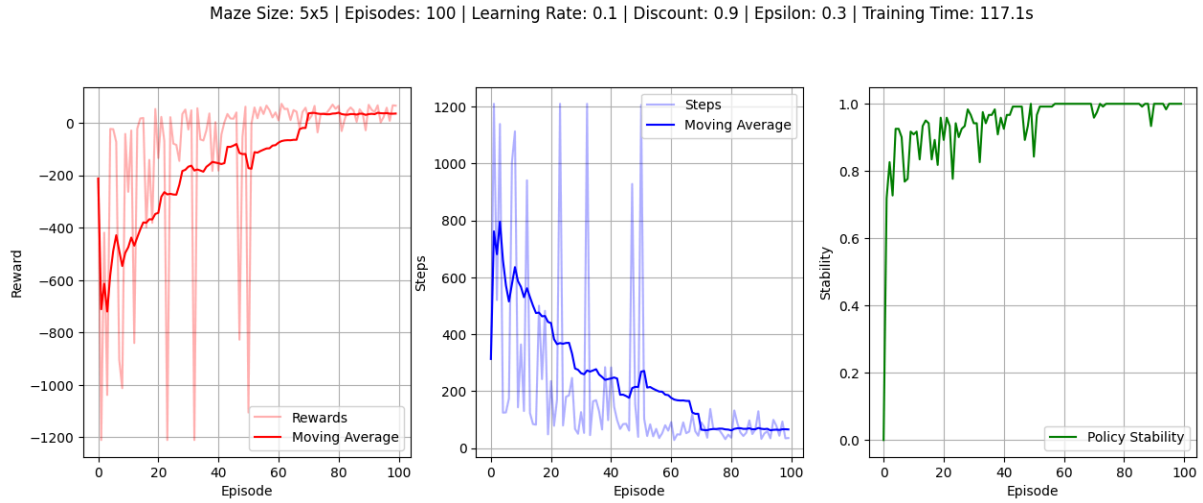
- With high exploration (epsilon = 0.3) the agent converges but slower than with a higher discount (0.99) (after approx 70 episodes of training)



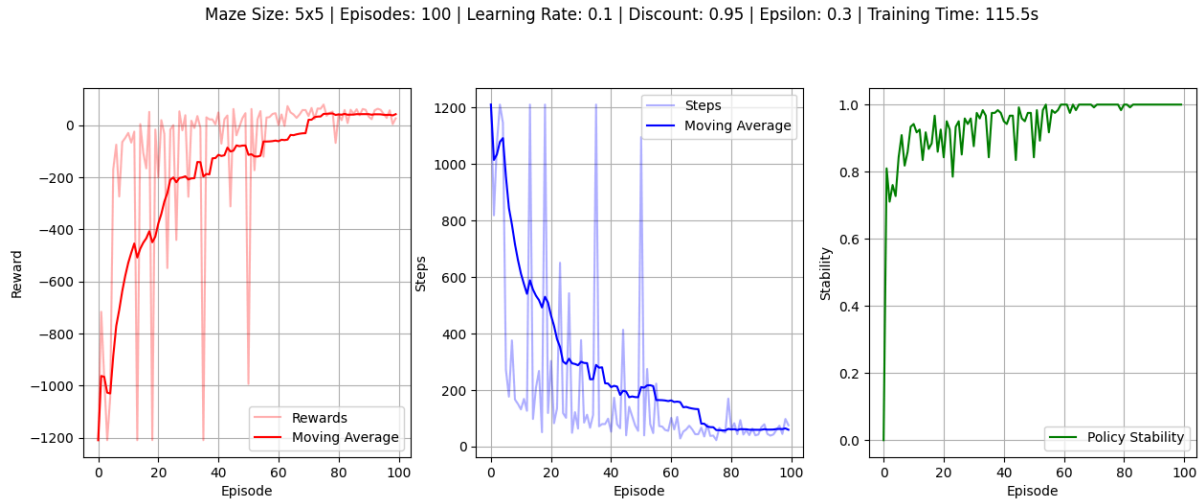
- With a medium exploration (epsilon = 0.2) the agent appears to converge within the 100 episodes but it is not conclusive.
- The agent does not converge with a low exploration (0.1)

Low discount (0.9)

- With high exploration (epsilon = 0.3) the agent converges but slower than with a higher discount (0.99) (after approx 70 episodes of training)



- The most interesting result appears here with a medium exploration rate (epsilon = 0.2). It appears that there is a convergence at approx. 40 episodes



- As with the medium discount, the agent does not converge with a low exploration (0.1)

Epsilon value

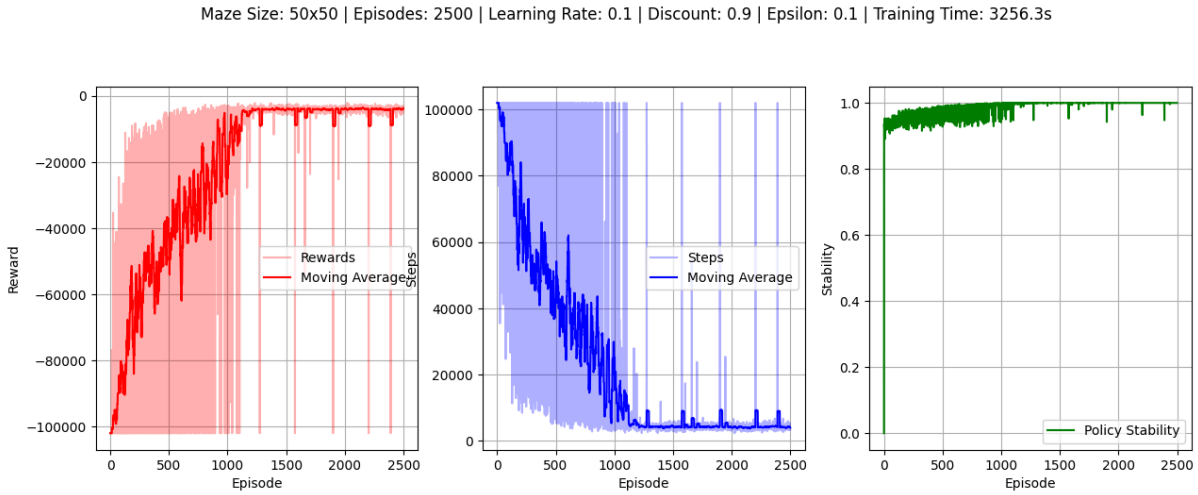
- In those training runs that did not converge there appears to be an inverse correlation between the epsilon value and the policy stability. This has a logic to it as the higher there is more exploration and the exploitation is sub-optimal and varies until the q-table has the optimal actions stored in it.
- Generally it seems that a higher exploration value equates to a quicker conversion on the optimal path. This implies that an agent learns quicker by exploring, some that has a common sense logic to it. It would be interesting, in future experiments, to increase the epsilon value until the convergence starts to come back out.
- There appears to be an exception to the above, the quickest apparent convergence was when the discount was low and exploration was medium (approx 40 steps).

Scale testing

The scale testing, by successfully solving the mazes up until size 50, has shown that the solution (agent, environment, and agent-environment control) scales on a laptop.

Other than concluding it does scale from 5 to 50, there is little to be drawn from this data. The reason is that the number of episodes is the size squared and this is what directly impacts the training time. As included below, stopping when the optimal path has been found and is continued to be used would be advantageous.

As we can see from below the agent converged on the optimal path for the 50x50 maze around episode 1200. Anecdotally, the training appears to be $O(n^2/2)$.



Overall

The results appear to show the following:

- A higher learning rate equates to quicker convergence (learning!)
- Exploration is an important factor in learning quickly
- There may be a sweet spot between discounting future returns and exporation

There are other factors that would be interesting to bring in to a future experiment, including:

- Experimenting with a higher exploration value
- A metric that tracks when the optimal path is found (i.e. when it converges)
- An early stopping mechanism once it has converged
- Measure the performance of the agents training as the maze scales
- A decaying epsilon value (i.e. high to start with and low as the training progresses)

Conclusion

Problem and objective revisited

The problem statement

> This project aims to develop and implement a self-navigating agent capable of finding optimal paths through arbitrarily sized maze environments, where complete knowledge of state transitions and rewards is unavailable.

The solution shows a successful implementation of a self-navigating Q-Learning agent. The agent learnt to navigate a maze without any prior knowledge. This was shown to work for mazes of size 50x50. However, the limits of the solution were not tested.

Objectives

- Develop a Q-Learning agent for maze navigation
- Create a scalable maze environment
- Evaluate the agent's learnings:
    1. Baseline - untrained compared to trained agents
    2. Hyperparameters - Experiment with changes to the model's hyperparameters (learning rate, epsilon-greedy value, discount rate of future rewards).
    3. Scale - maze sizes from 5 to 50
- Analyse the results
- Present findings

The success criteria

> The agent's performance will be considered successful if it:
- Converges on a stable policy
- Consistently finds the optimal path to the goal

There are two key elements that highlight agents that have met the success criteria:
- a policy stability of 1.0 for agents that have found the optimal path.
- agents trained with a learning rate of 0.1 (except a discount of 0.99 and opsilon of <0.3) consistently found the optimal path.

Key findings

- Learning rate of 0.1 provided optimal for convergence
- Higher exploration rates (epsilon = 3) generally led to faster learning
- The solutions scaled successfully to a 50 x 50 maze
- Training appears to have followed $O(n^2/n)$ complexity

Limitations

- The number of episodes is fixed at the start of training.
- Not able to view the trained maze in the experiment datasets

Future research

- Experimenting with a higher exploration value
- A metric that tracks when the optimal path is found (i.e. when it converges)
- An early stopping mechanism once it has converged
- Measure the performance of the agents training as the maze scales
- A decaying epsilon value (i.e. high to start with and low as the training progresses)
- Add uncertainty to the environment (e.g. probabilistic wind/slope in some states which move the agent or dirty sensor that gives the agent incorrect location information).

Appendices

Appendix A: Maze algorithm testing

BinaryTree, Sidewinder, Kruskal, Prims

Sidewinder is the quickest

Final Results:

BinaryTree:
  10x10: 0.00s
  100x100: 0.02s

Sidewinder:
  10x10: 0.00s
  100x100: 0.01s

Kruskal:
  10x10: 0.00s
  100x100: 25.47s

Prims:
  10x10: 0.00s
  100x100: 0.19s

Python code for testing the algorithms

```
import time
from typing import Type, List
from mazelib import Maze
from mazelib.generate.Prims import Prims
from mazelib.generate.Kruskal import Kruskal
from mazelib.generate.BinaryTree import BinaryTree
from mazelib.generate.Sidewinder import Sidewinder

def test_algorithm(algo_class: Type, size: int, trials: int = 1) ->
float:
    times = []
    print(f"\nTesting {algo_class.__name__} at size {size}x{size}")
    for trial in range(trials):
        print(f"  Trial {trial + 1}/{trials}...", end=" ", flush=True)
        m = Maze()
        m.generator = algo_class(size, size)
        start = time.perf_counter()
```

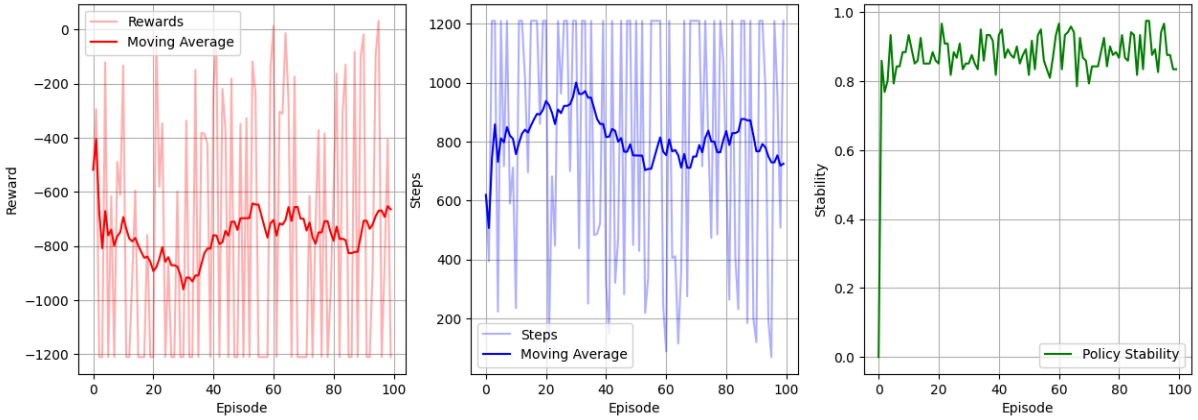
```
        m.generate()
        m.generate_entrances()
        trial_time = time.perf_counter() - start
        times.append(trial_time)
        print(f"{trial_time:.2f}s")
    avg_time = sum(times) / len(times)
    print(f"  Average: {avg_time:.2f}s")
    return avg_time

algorithms = [BinaryTree, Sidewinder, Kruskal, Prims]
sizes = [10, 100]

results = {}
for size in sizes:
    print(f"\n{'='*50}")
    print(f"Testing size {size}x{size}")
    for algo in algorithms:
        if algo.__name__ not in results:
            results[algo.__name__] = []
        try:
            avg_time = test_algorithm(algo, size)
            results[algo.__name__].append((size, avg_time))
        except Exception as e:
            print(f"Error with {algo.__name__}: {e}")
            continue

print("\nFinal Results:")
for algo, data in results.items():
    print(f"\n{algo}:")
    for size, time in data:
        print(f"  {size}x{size}: {time:.2f}s")
```

Appendix B: Visualisations of the results

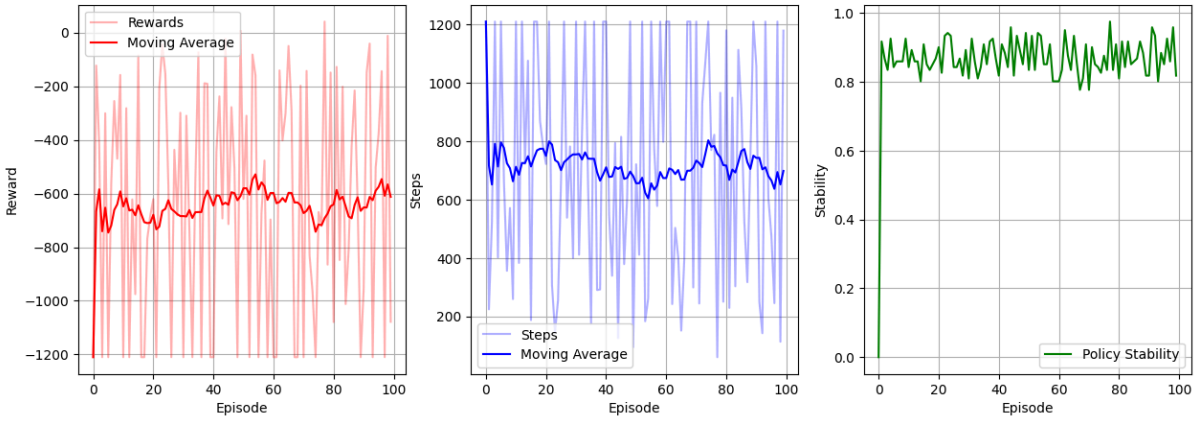
Hyperparameter testing

Learning Rate 0.001

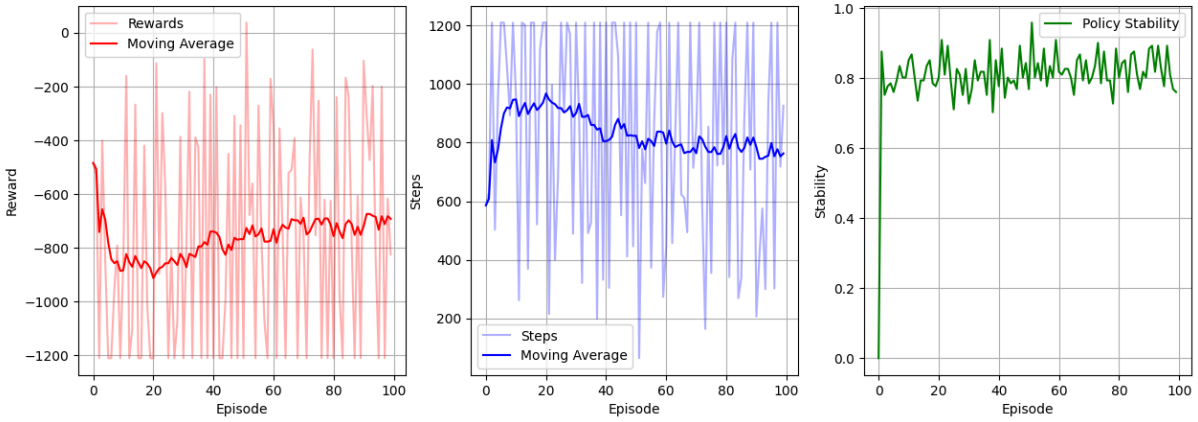
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.99 | Epsilon: 0.1 | Training Time: 106.6s



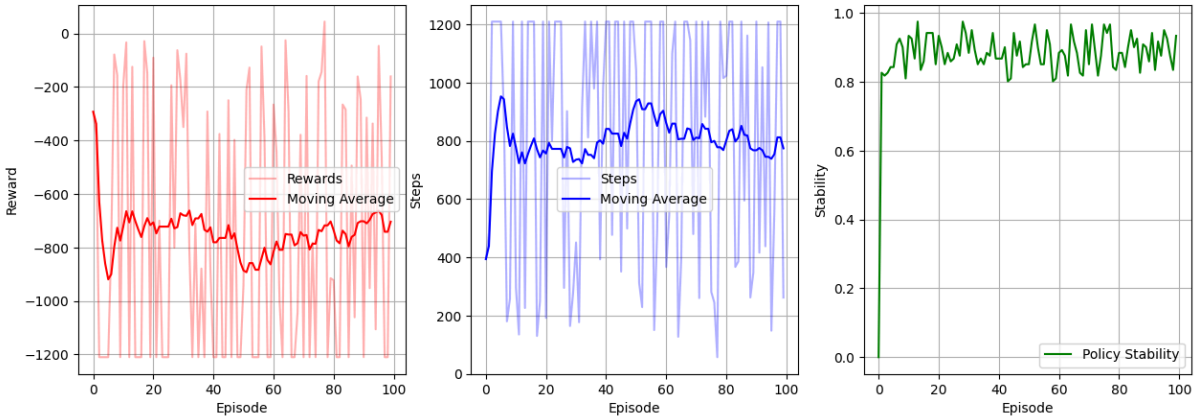
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.99 | Epsilon: 0.2 | Training Time: 107.2s



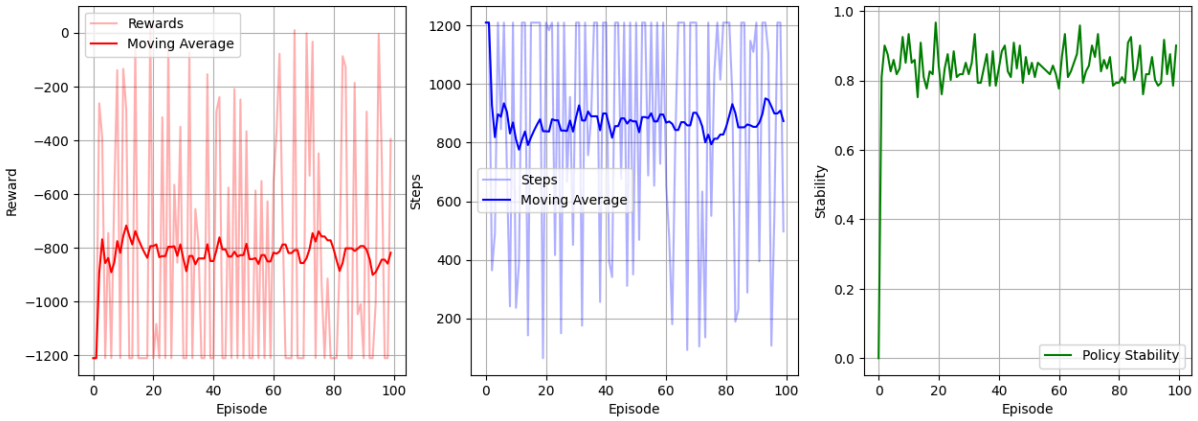
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.99 | Epsilon: 0.3 | Training Time: 107.0s



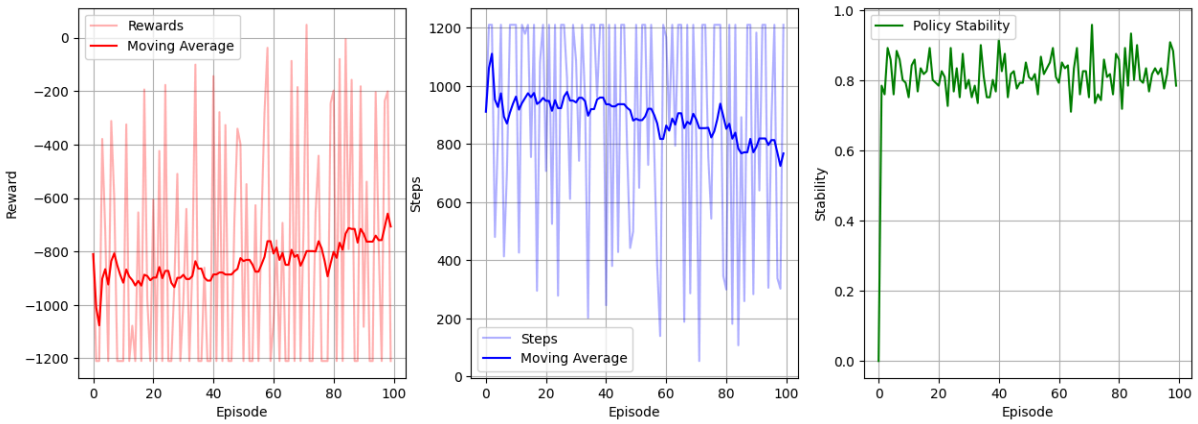
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.95 | Epsilon: 0.1 | Training Time: 107.4s



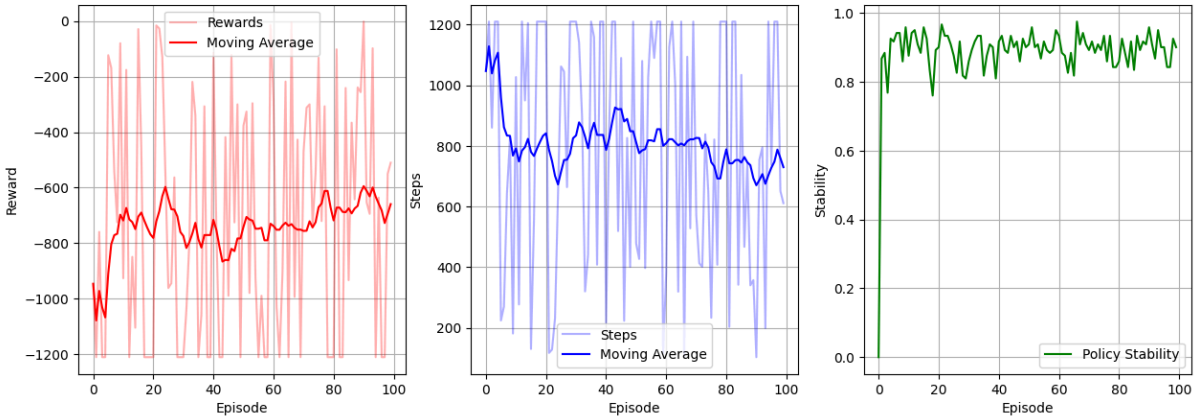
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.95 | Epsilon: 0.2 | Training Time: 107.8s



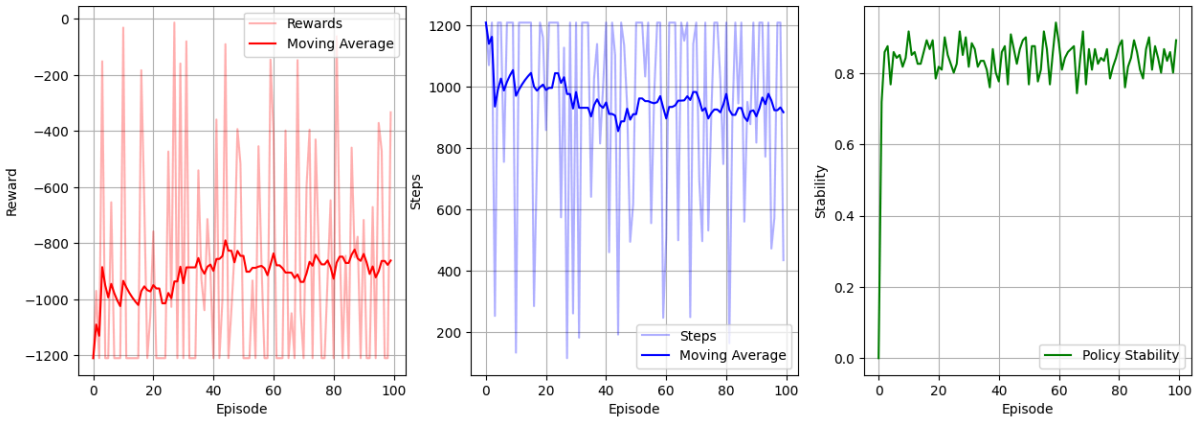
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.95 | Epsilon: 0.3 | Training Time: 106.9s



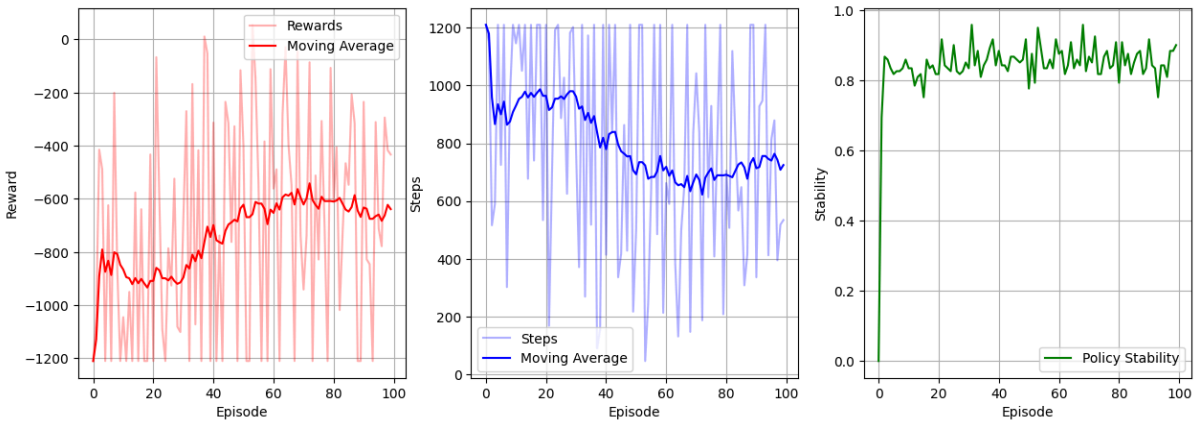
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 106.7s



Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.9 | Epsilon: 0.2 | Training Time: 105.7s

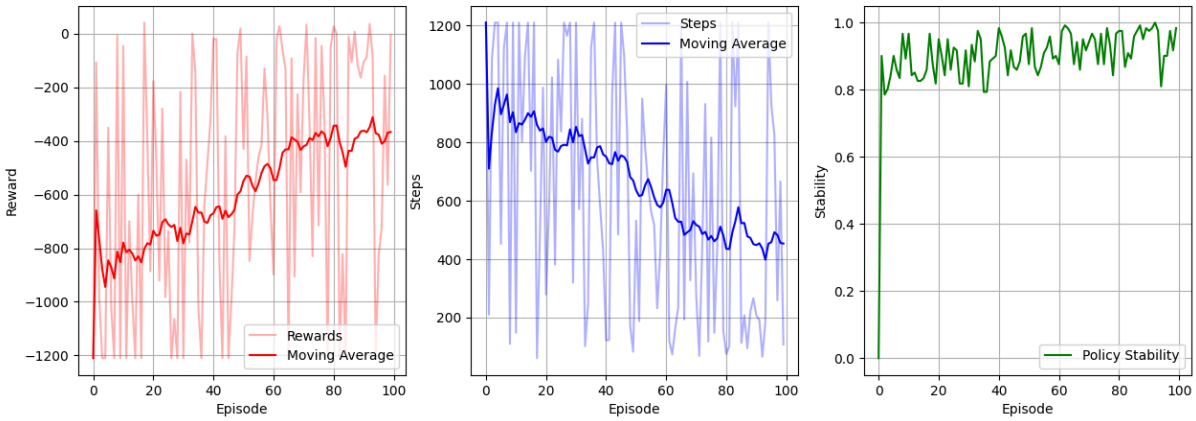


Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.001 | Discount: 0.9 | Epsilon: 0.3 | Training Time: 105.9s

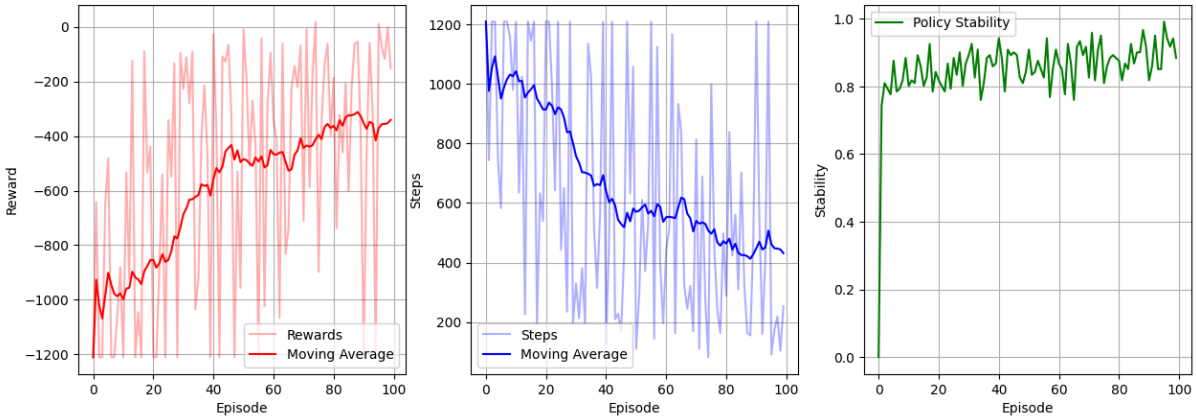


Learning Rate 0.01

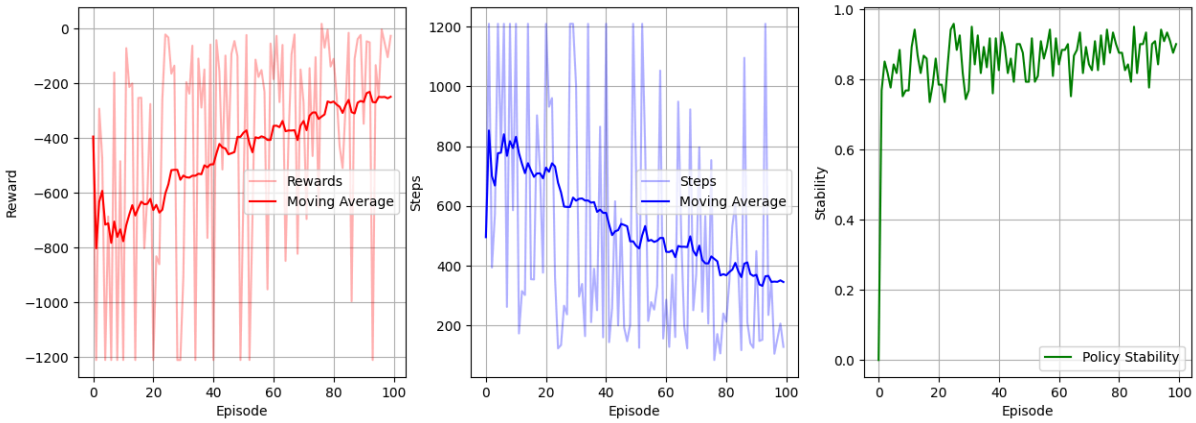
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.99 | Epsilon: 0.1 | Training Time: 106.6s



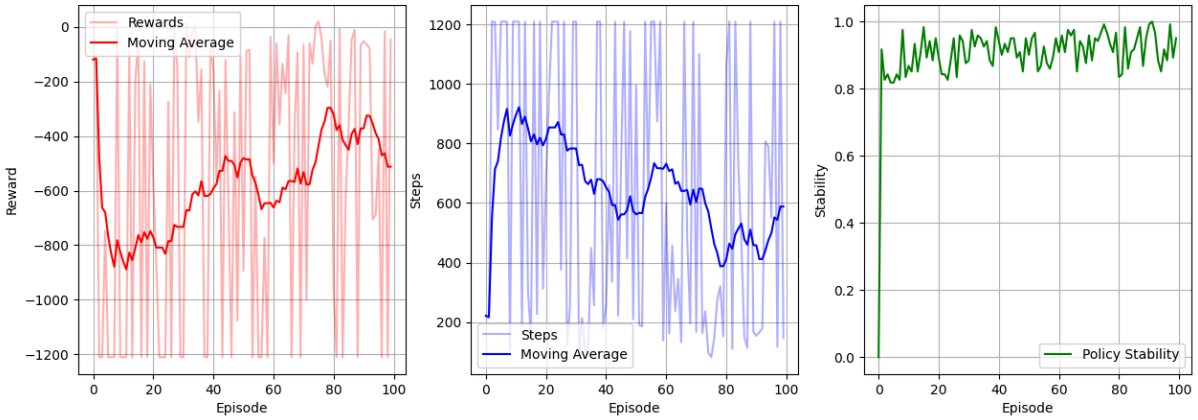
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.99 | Epsilon: 0.2 | Training Time: 105.8s



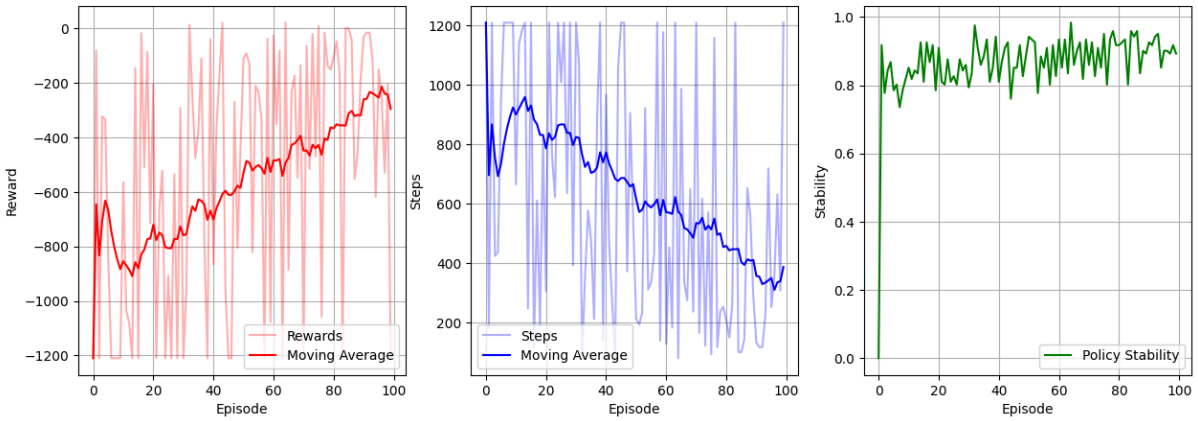
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.99 | Epsilon: 0.3 | Training Time: 106.7s



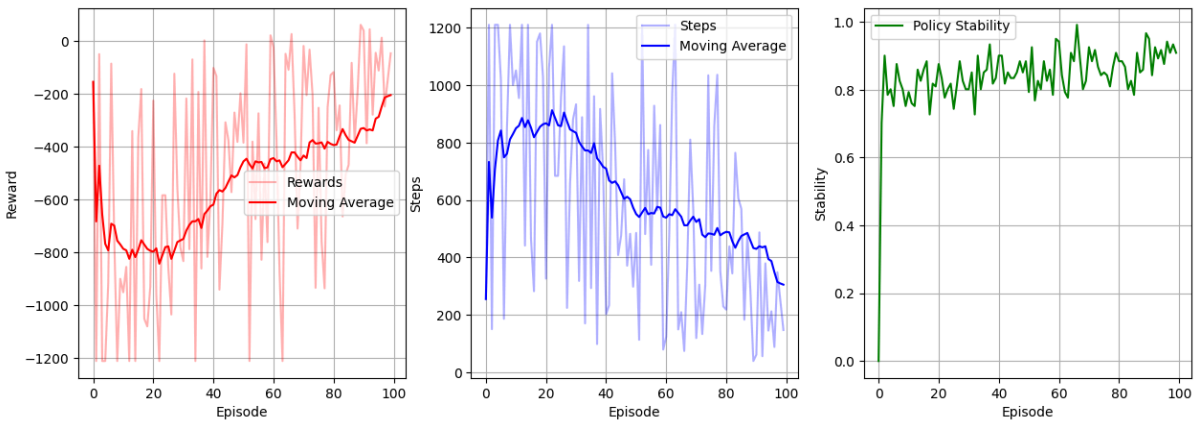
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.95 | Epsilon: 0.1 | Training Time: 109.2s



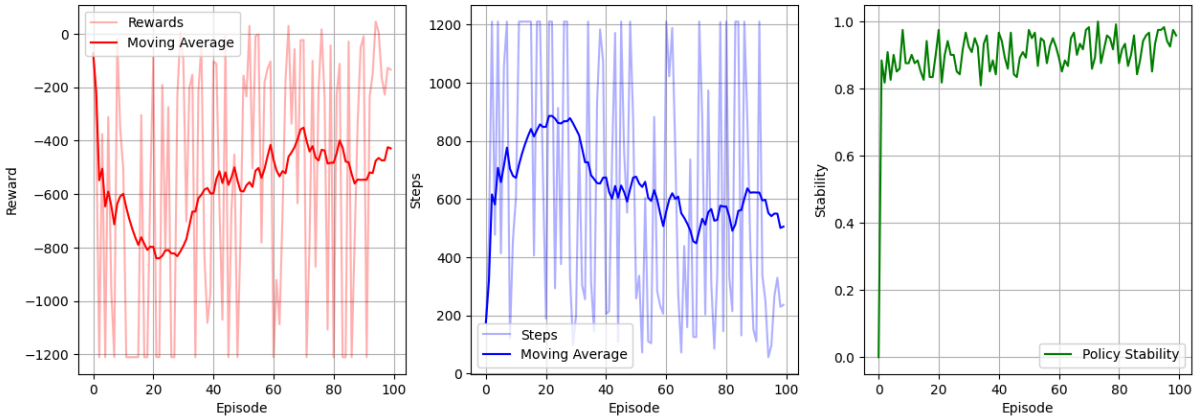
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.95 | Epsilon: 0.2 | Training Time: 108.5s



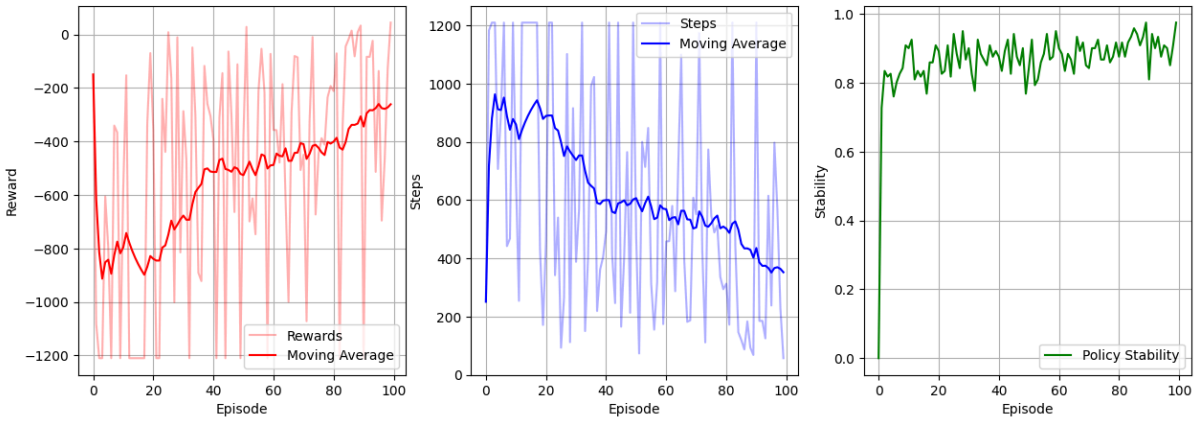
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.95 | Epsilon: 0.3 | Training Time: 106.2s



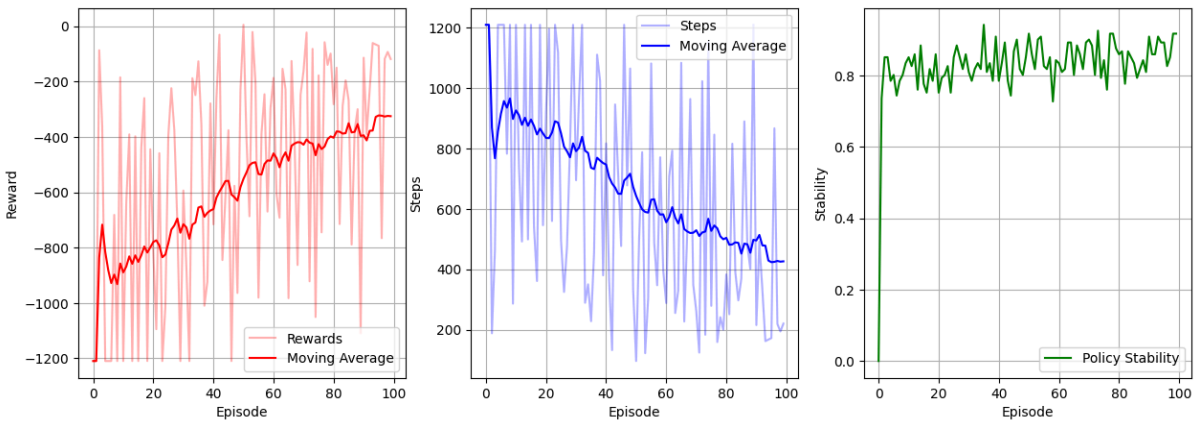
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 109.2s



Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.9 | Epsilon: 0.2 | Training Time: 107.7s

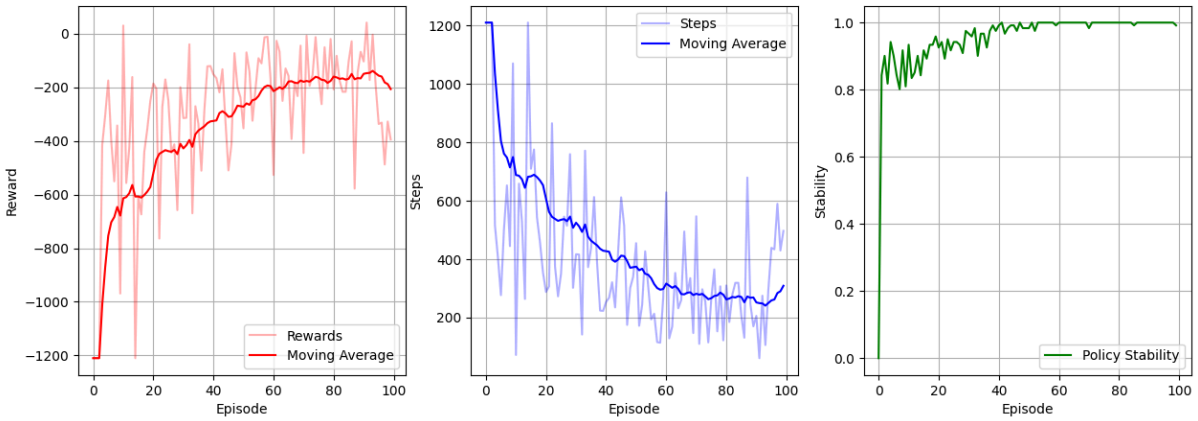


Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.01 | Discount: 0.9 | Epsilon: 0.3 | Training Time: 107.7s

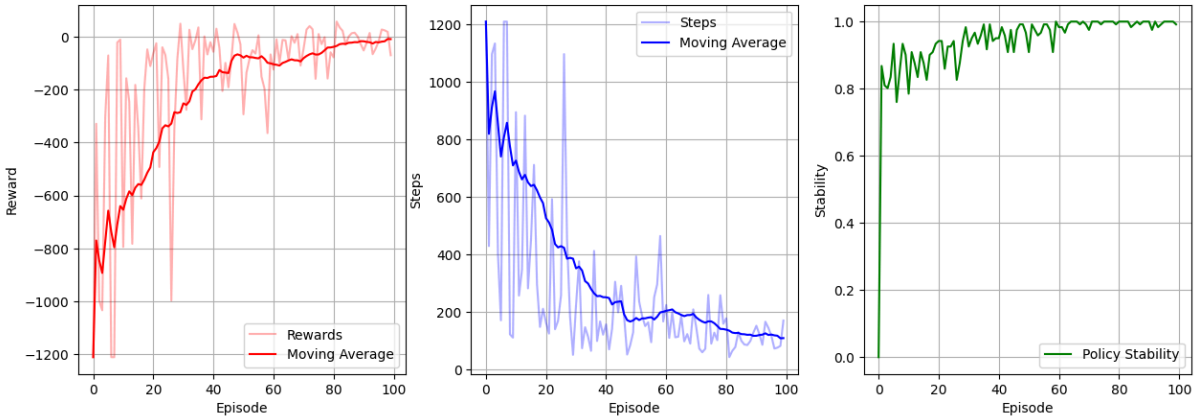


Learning rate: 0.1

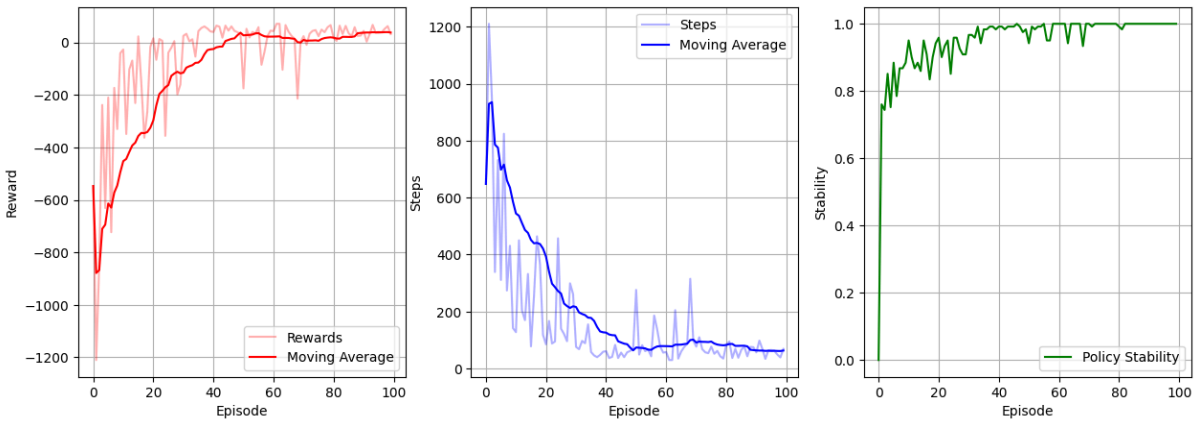
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.99 | Epsilon: 0.1 | Training Time: 106.1s



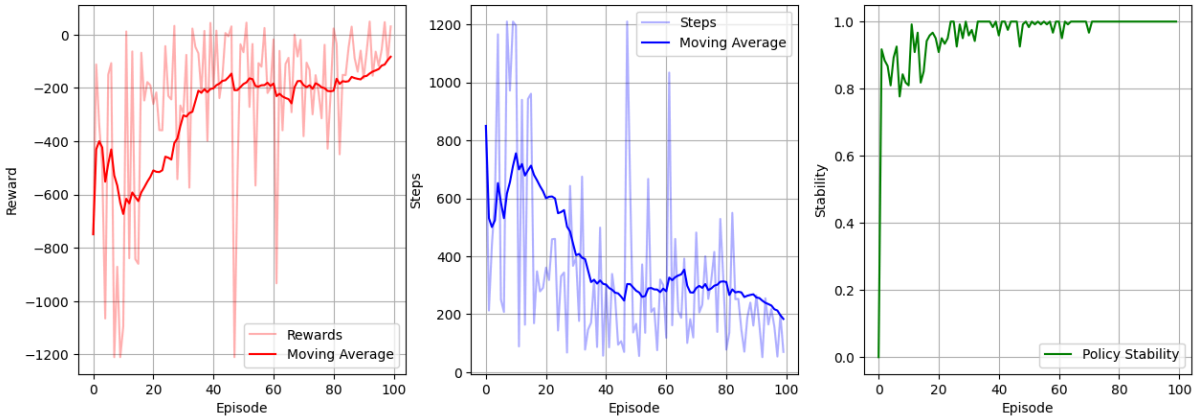
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.99 | Epsilon: 0.2 | Training Time: 103.2s



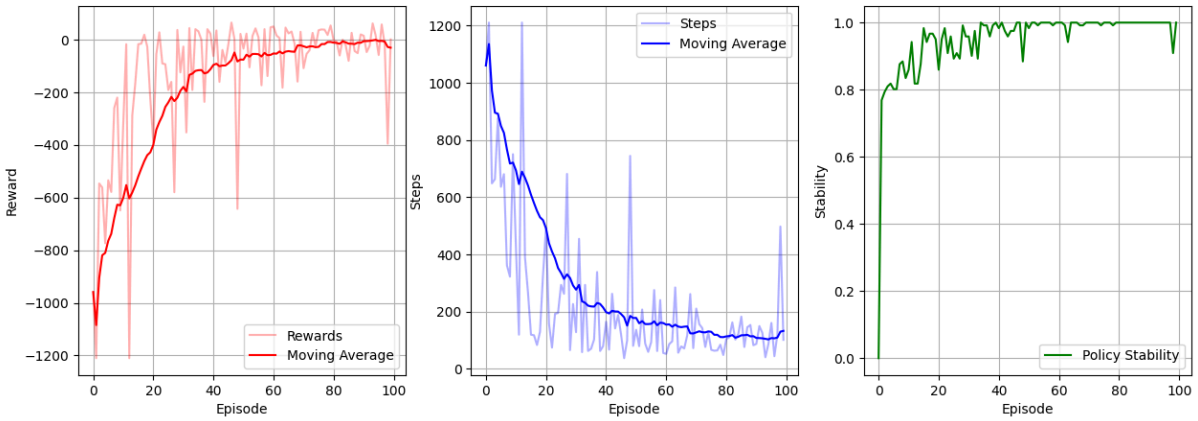
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.99 | Epsilon: 0.3 | Training Time: 103.6s



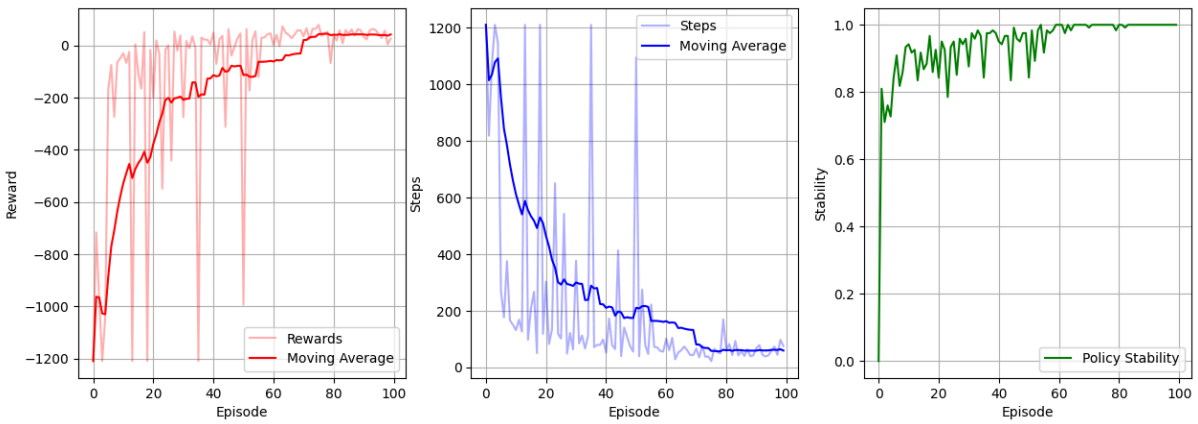
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.95 | Epsilon: 0.1 | Training Time: 116.5s



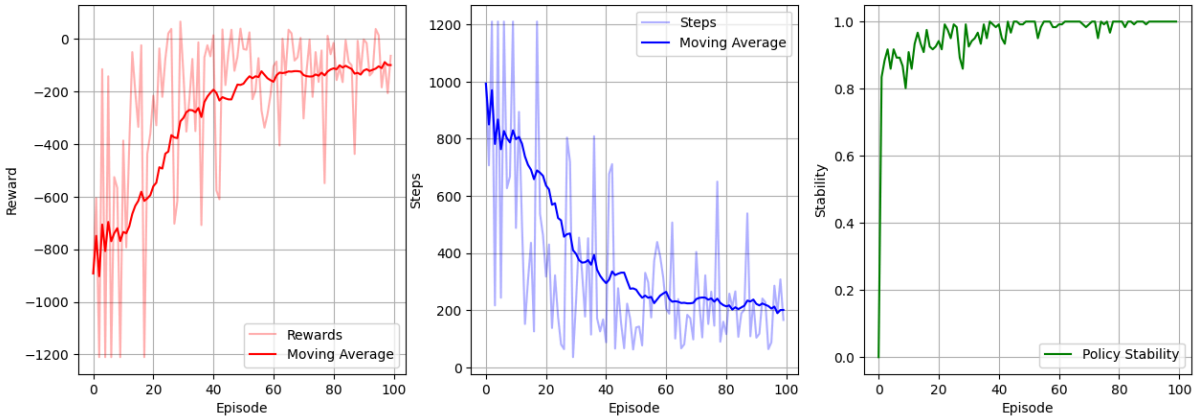
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.95 | Epsilon: 0.2 | Training Time: 112.1s



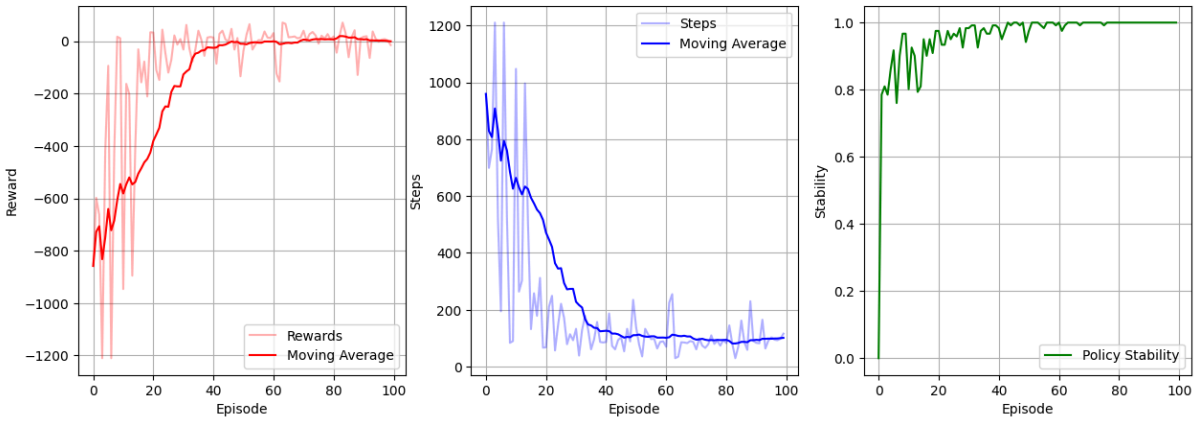
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.95 | Epsilon: 0.3 | Training Time: 115.5s



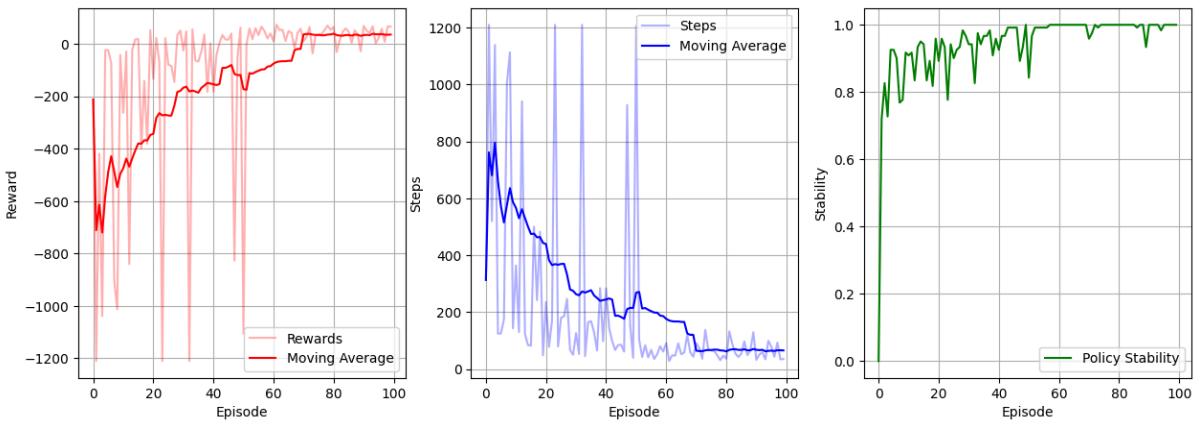
Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 108.4s



Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.2 | Training Time: 109.7s

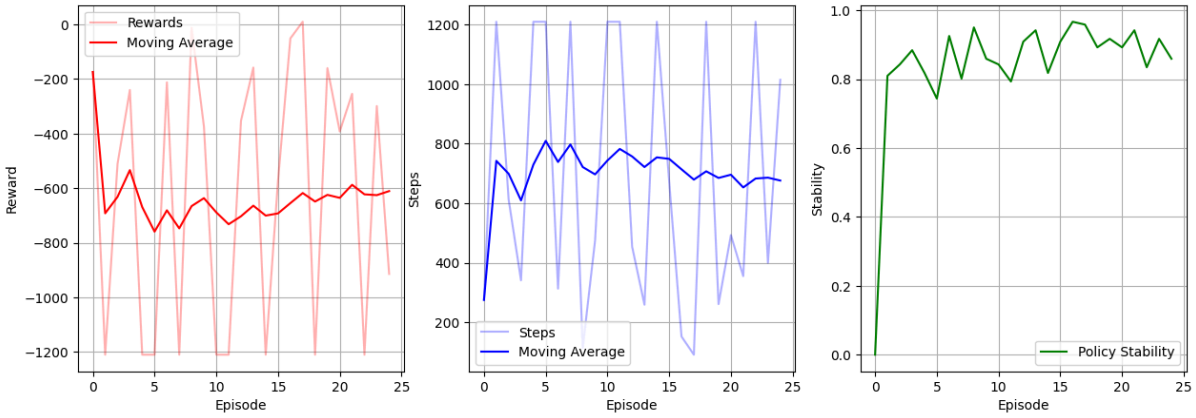


Maze Size: 5x5 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.3 | Training Time: 117.1s

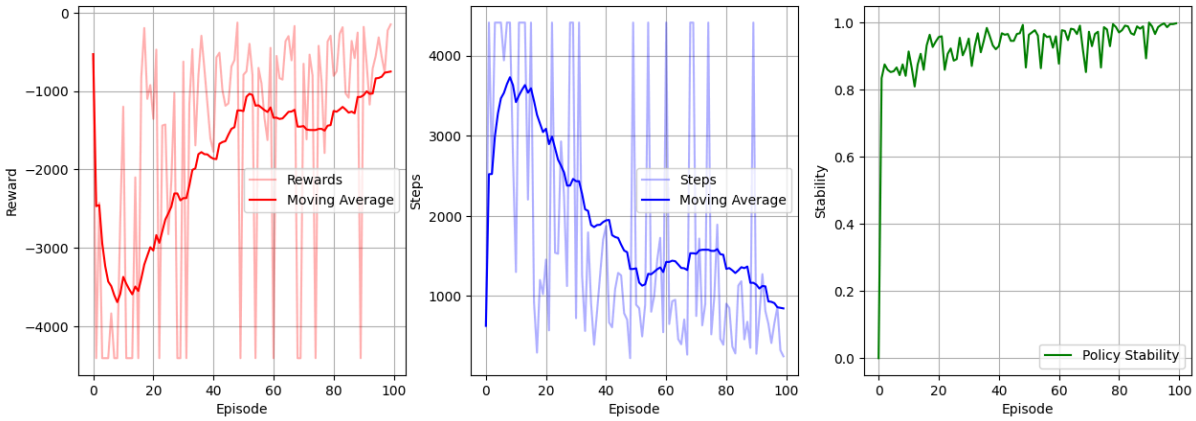


Scale testing

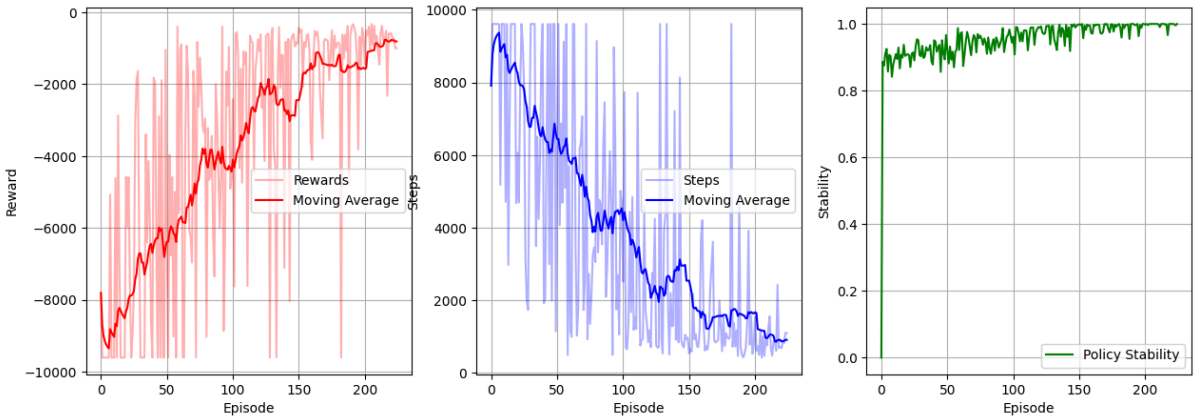
Maze Size: 5x5 | Episodes: 25 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 26.7s



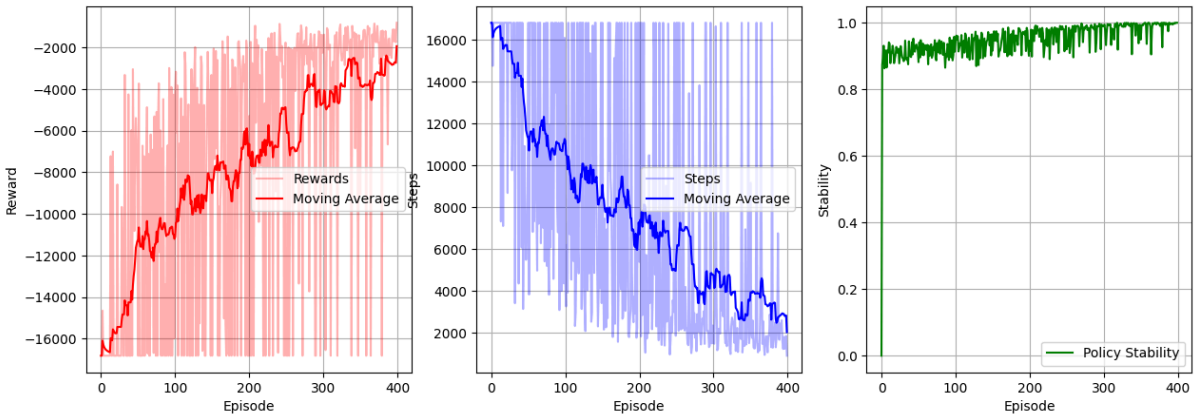
Maze Size: 10x10 | Episodes: 100 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 106.8s



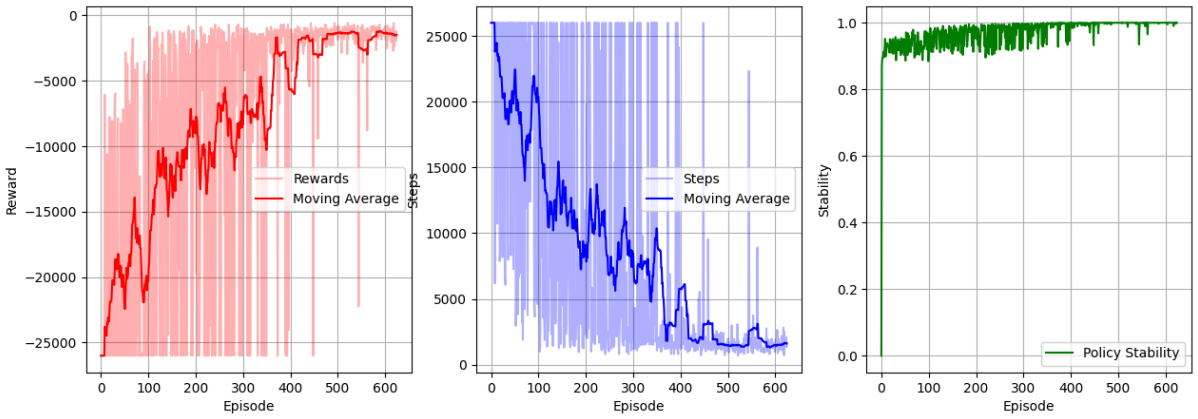
Maze Size: 15x15 | Episodes: 225 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 247.6s



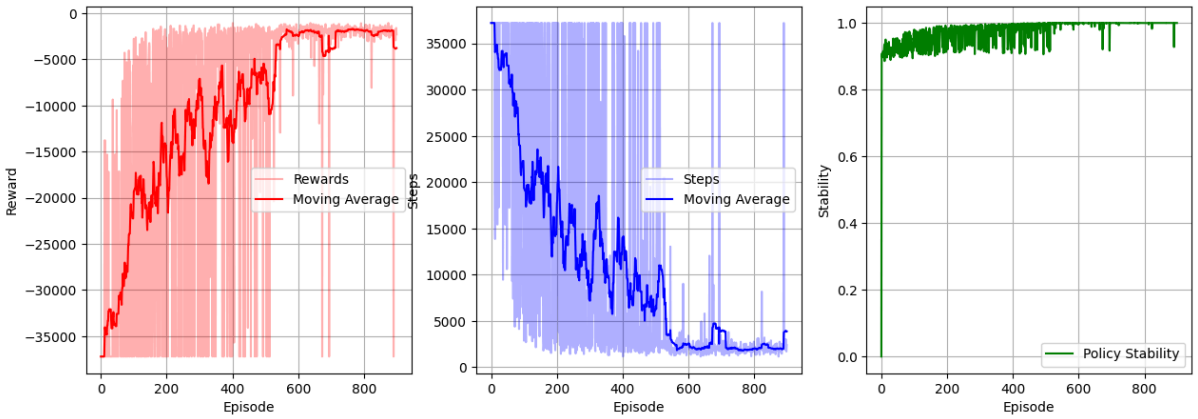
Maze Size: 20x20 | Episodes: 400 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 465.3s



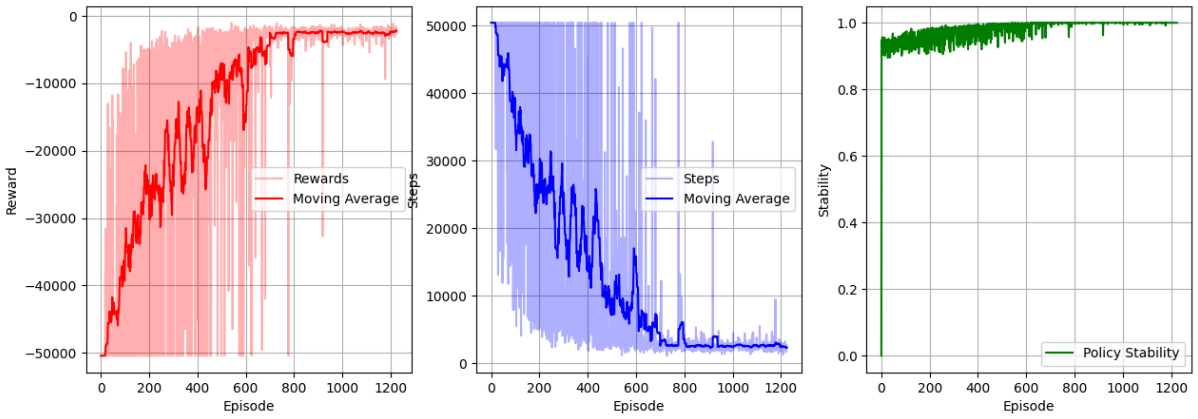
Maze Size: 25x25 | Episodes: 625 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 720.2s



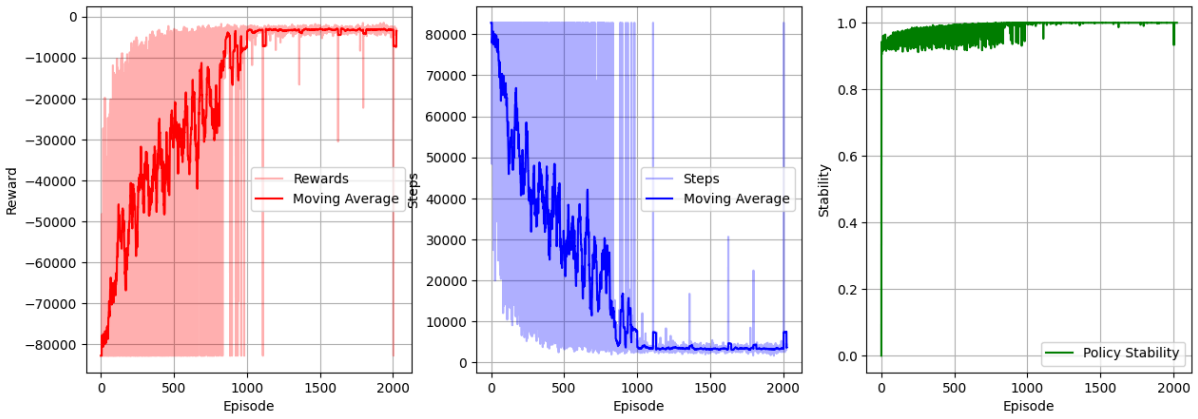
Maze Size: 30x30 | Episodes: 900 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 1078.3s



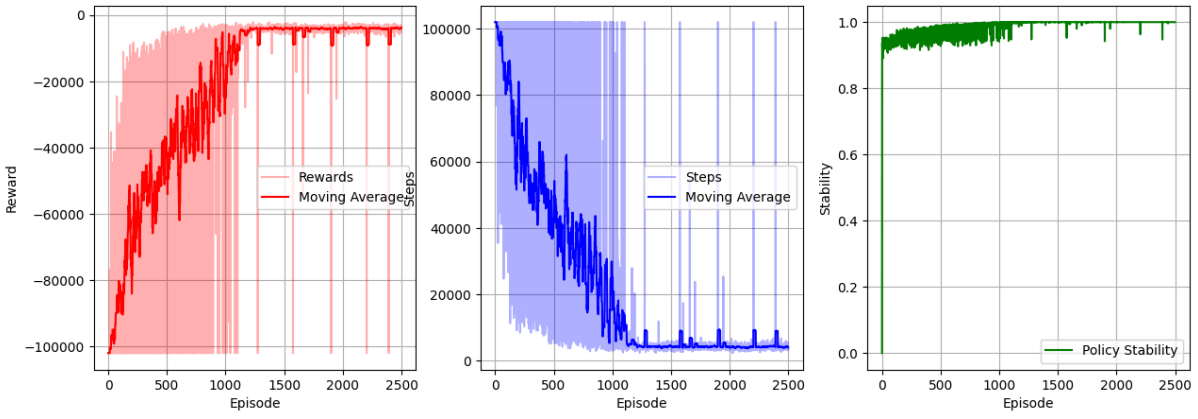
Maze Size: 35x35 | Episodes: 1225 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 1480.2s



Maze Size: 45x45 | Episodes: 2025 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 2645.2s



Maze Size: 50x50 | Episodes: 2500 | Learning Rate: 0.1 | Discount: 0.9 | Epsilon: 0.1 | Training Time: 3256.3s



Links and References

[1] http://jmc.stanford.edu/articles/dartmouth/dartmouth.pdf
[2] https://ieeexplore.ieee.org/document/4066245

[3] https://www.bfskinner.org/newtestsite/wp-content/uploads/2014/02/ScienceHumanBehavior.pdf
[4] https://en.wikipedia.org/wiki/The_Alignment_Problem
[5] https://en.wikipedia.org/wiki/Law_of_effect
[6] https://en.wikipedia.org/wiki/Markov_chain
[7] https://en.wikipedia.org/wiki/Control_theory
[8] https://en.wikipedia.org/wiki/Dynamic_programming
[9] https://en.wikipedia.org/wiki/Bellman_equation
[10] https://en.wikipedia.org/wiki/Markov_decision_process
[11] https://en.wikipedia.org/wiki/Markov_decision_process#Value_iteration
[12] https://en.wikipedia.org/wiki/Markov_decision_process#Policy_iteration
[13] http://www.incompleteideas.net/
[14] http://www.incompleteideas.net/book/the-book.html
[15] https://princetonuniversity.github.io/PsyNeuLink/MontagueModel.html
[16] https://deepmind.google/discover/blog/dopamine-and-temporal-difference-learning-a-fruitful-relationship-between-neuroscience-and-ai/
[17] https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf
[18] https://github.com/john-science/mazelib/