Project 4

Prameth Gaddale, pqg5273@psu.edu

ABSTRACT

This project deals with use of reinforcement learning algorithms such as Q-Learning and Deep Q-Learning for learning optimal policies in solving the agent's motion in various challenging maze configurations. Generally, reinforcement learning is a machine learning methodology that focuses on training agents to make decisions in an environment to maximize a cumulative predefined reward. In this approach, an agent interacts with its environment, learning through trial and error to take actions that lead to the highest reward. The agent receives feedback from the environment in the form of a reward signal and uses this feedback to adjust its behaviour. Reinforcement Learning algorithms typically involve an exploration-exploitation trade-off, where the agent must balance the desire to exploit its current knowledge with the need to explore new actions that may lead to even higher rewards. The output signal is very sparse compared to other machine learning strategies such as supervised learning or unsupervised learning as the learning is happening through the reward that the agent receives, so choosing a good reward function and policy is crucial for the functioning of a reinforcement learning algorithms. The field of Reinforcement Learning has applications in a wide range of areas, including robotics, gaming, and autonomous systems.

Contents

ABSTRACT 1
Reinforcement Learning 2
  Markov-Decision Process 2
  Q Learning 2
  Algorithms 3
    Value Iteration 3
    Policy Iteration 3
  OpenAI Gym: Deep Maze 4
  Learning 5
  Results 5
  Comparison 9
  Summary 9
Deep Reinforcement Learning 10
  Algorithms 10
  Deep Q-Networks 11
  Results 11
  Observations 15
  Comparison 17
  Summary 17
SOTA: Maze Solving 18
  Dreaming Variational Autoencoder 18
    Proximal Policy Optimization 19
    Results 20
References 20

Reinforcement Learning

Markov-Decision Process

A Markov Decision Process (MDP) is a mathematical framework used in Reinforcement Learning for modelling decision-making problems in which the expected/received outcome depends on both the random events and the actions taken by the agent. MDPs are widely used in applications such as robotics, game playing, and decision-making under uncertainty.

Typically, an MDP is defined by a set of states, actions, rewards, and transition probabilities. At each time step, the agent is in a particular state, and can take an action to transition to a new state. The outcome of each action is probabilistic and depends on the current state and the action taken. The agent receives a reward based on the transition to the new state, and its goal is to maximize the cumulative reward over time. The transition probabilities specify the probability of transitioning from one state to another state, given a particular action. The reward function specifies the reward received for each transition from one state to another. The objective of the agent is to find a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time.

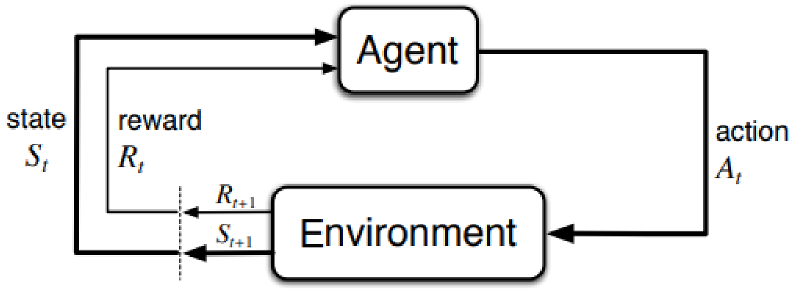


Figure: General schematic of a Markov Decision Process.

The solution to an MDP is a value function or a Q-function that assigns a value to each state or state-action pair, respectively, representing the expected cumulative reward starting from that state or state-action pair and following the optimal policy. These value functions can be computed using dynamic programming or Reinforcement Learning algorithms, such as Q-learning. MDPs are a powerful tool for modelling and solving decision-making problems under uncertainty, and are widely used in artificial intelligence, control theory, and operations research.

Q Learning

Q-learning is a popular algorithm used in Reinforcement Learning for finding the optimal action-selection policy in a Markov Decision Process (MDP). It is a model-free algorithm, which means that it does not require a model of the environment and its dynamics to make decisions [1].

The key idea behind Q-learning is to learn an action-value function ***Q(s,a)***, which estimates the expected future reward for taking a particular action, **a** in a particular state, **s**. The action-value function is updated iteratively using the Bellman equation, which expresses the relationship between the value of a state and the values of its neighbouring states.

At each time step, the agent observes the current state, takes an action based on its current policy, receives a reward, and observes the next state. It then updates its estimate of the action-value function for the current state-action pair using the reward and the maximum expected future reward from the next state, according to the Bellman equation.

Over time, the action-value function converges to the optimal action-value function, which specifies the highest expected future reward for each state-action pair. The optimal policy can then be derived from the optimal action-value function, by selecting the action with the highest expected future reward for each state. Q-learning has been successfully applied to a wide range of problems, including game playing, robotics, and autonomous navigation.

Algorithms

Value Iteration

In Q-learning, value iteration is used to update the Q-values, which represent the expected reward of taking an action in each state. The basic idea behind value iteration is to iteratively update the Q-values using the Bellman equation until they converge to their optimal values. The Bellman equation expresses the value of a state-action pair as the sum of the immediate reward and the discounted expected value of the next state. By iteratively applying the Bellman equation to all state-action pairs, the Q-values are gradually updated until they converge to their optimal values.

The value iteration algorithm starts by initializing the Q-values randomly or with some initial estimate. Then, at each iteration, it updates the Q-values for all state-action pairs using the Bellman equation. The algorithm repeats this process until the Q-values converge to their optimal values. In Q-learning, value iteration is used to update the Q-values after each action taken by the agent. The Q-value for a state-action pair is updated using the observed reward and the estimated value of the next state, which is computed using the Q-values of the next state and the available actions.

Algorithm 1 Value Iteration

```
Initialize $V^{(0)}$.
for $n=1,2,\ldots$ do
    for $s \in S$ do
        $V^{(n)}(s) = \max_a \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V^{(n-1)}(s'))$
    end for
    ▷ The above loop over $s$ could be written as $V^{(n)} = TV^{(n-1)}$
end for
```

Figure: General algorithm for solving Q-Learning problems using Value Iteration. [2]

Value iteration is a powerful algorithm for solving MDPs and finding the optimal policy in Reinforcement Learning. However, it can be computationally expensive for large state and action spaces and may require a lot of training data to converge to the optimal values.

Policy Iteration

In Q-learning, policy iteration can be used to update the policy, which is a mapping from states to actions that maximizes the expected cumulative reward over time. The basic idea behind policy iteration is to iteratively improve the policy by alternating between two steps: policy evaluation and policy improvement. In the policy evaluation step, the algorithm evaluates the value function of the current policy by iteratively applying the Bellman equation until it converges to its optimal value. In the policy improvement step, the algorithm updates the policy by selecting the action that maximizes the expected value of the next state, based on the current value function.

The policy iteration algorithm starts by initializing the policy randomly or with some initial estimate. Then, at each iteration, it alternates between policy evaluation and policy improvement until the policy converges to the optimal policy.

Algorithm 2 Policy Iteration

```
Initialize $\pi^{(0)}$.
for $n = 1, 2, \ldots$ do
    $V^{(n-1)} = \text{Solve}[V = T^{\pi^{(n-1)}} V]$
    for $s \in S$ do
        $\pi^{(n)}(s) = \text{argmax}_a \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma V^{(n-1)}(s')]$
                  $= \text{argmax}_a Q^{\pi^{(n-1)}}(s,a)$
    end for
end for
```

Figure: General algorithm for solving Q-Learning problems using Policy Iteration. [2]

In Q-learning, policy iteration can be used to update the policy after each action taken by the agent. The policy is updated by selecting the action that maximizes the Q-value of the current state, based on the current estimates of the Q-values. Policy iteration can be more computationally efficient than value iteration, as it converges to the optimal policy in fewer iterations. However, it may still be computationally expensive for large state and action spaces.

OpenAI Gym: Deep Maze

DeepMaze is a maze environment for reinforcement learning, which is part of the OpenAI Gym toolkit [3]. The environment consists of a randomly generated maze with a start and end point, and the goal is for the agent to navigate through the maze to reach the end point. In DeepMaze, the agent observes the current location of the maze, and can take actions such as moving up, down, left, or right. The agent receives a reward of +1 for reaching the end point and a reward of -1 for hitting a wall or moving out of bounds.

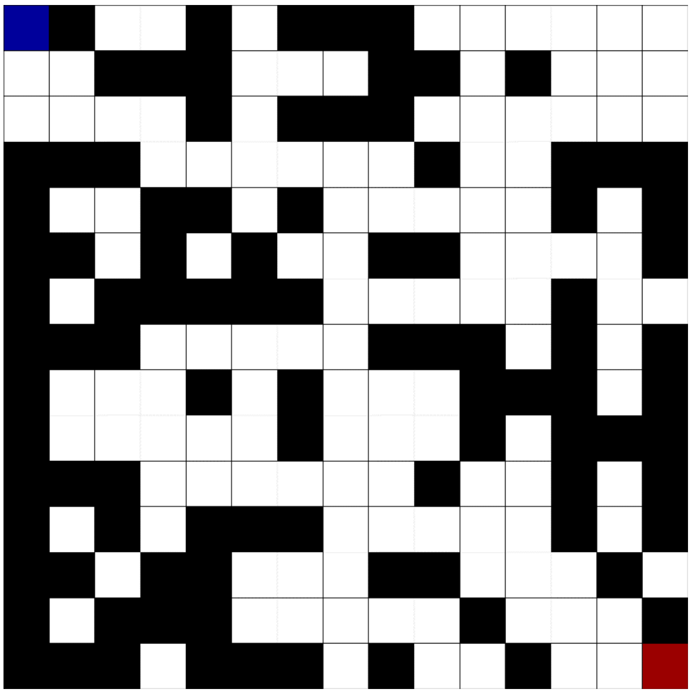


Figure: Representation of a 15X15 OpenAI Gym DeepMaze maze configuration.

DeepMaze is designed to be a challenging environment for reinforcement learning agents, as the maze can be generated with a variety of different configurations and layouts. This requires the agent to learn a general strategy for navigating through mazes rather than simply memorizing a specific path.

The OpenAI Gym provides a testing and evaluation interface for reinforcement learning environments, which makes it easy to train and compare different reinforcement learning algorithms. With the availability of DeepMaze in the OpenAI Gym, researchers and practitioners can easily use it as a benchmark environment to evaluate their reinforcement learning algorithms, and to develop new and more effective algorithms.

Learning

The agents make their way through the different maze configurations without the knowledge of the entire map, but they just get provided with the exploration and the reward signal they obtain. However, this only makes sense if, after every run through the maze configuration, out agent looks back at the path it followed and learns something from it. Simplifying things a bit, after enough traversals, the agent will hopefully converge to the optimal path to the end point specified in the settings.

To use Q-learning for maze navigation, we start by setting the Q-values to a set of arbitrary values. Hence, we then repeatedly choose an action in the current state using an exploration-exploitation strategy (such as epsilon-greedy), take the action which we got from the policy, observe the reward and the next state, and update the Q-values using the Bellman equations. As the Q-values are updated, the agent learns the optimal policy, which is the sequence of actions that leads to the goal with the highest expected reward. The policy can be derived from the Q-values by choosing the action with the highest Q-value in each state.

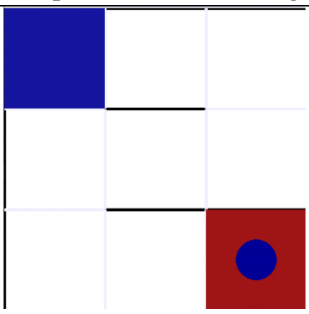
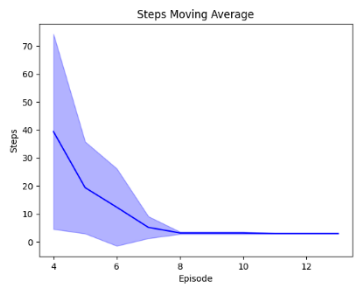
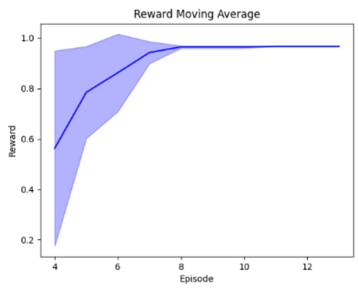
Results

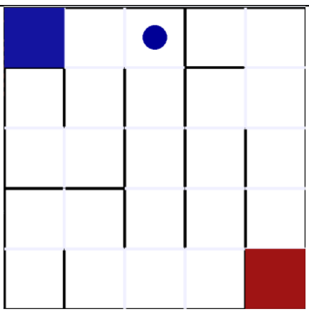
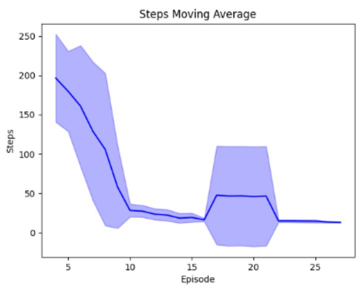
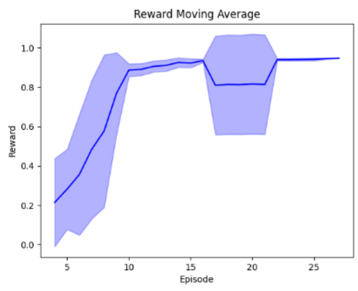
Maze configurations, in general are extremely useful for testing and evaluating the performance of reinforcement learning algorithms, such as Q-learning. In the context of maze configurations, the agent (or learner) navigates through the maze and receives rewards (or penalties) based on its actions. The goal is for the agent to learn the optimal path through the maze that leads to the highest reward.

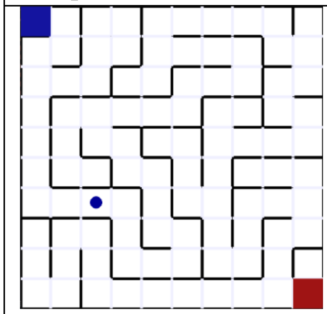
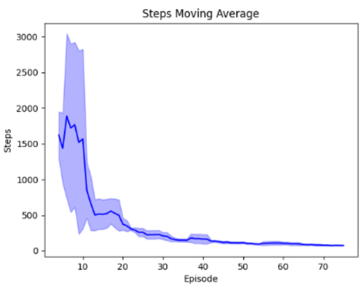
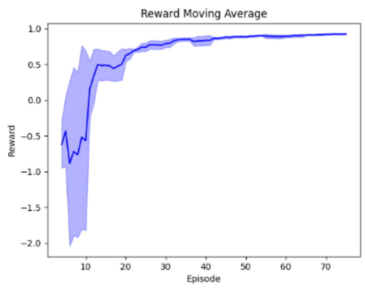
Observations in Q-learning refer to the information that the agent receives about the environment, which it uses to make decisions. In the context of maze configurations, observations may include information about the agent's current location, the locations of walls or obstacles, and the locations of rewards or penalties.

Overall, maze configurations can provide a useful framework for testing and evaluating Q-learning algorithms, as they allow researchers to control and manipulate the environment in a systematic way. This can help to isolate specific factors that may impact the performance of the algorithm and provide insights into how to design more effective learning systems.

Let's look at the results we get from running the Q-Learning algorithm on the sample configuration.

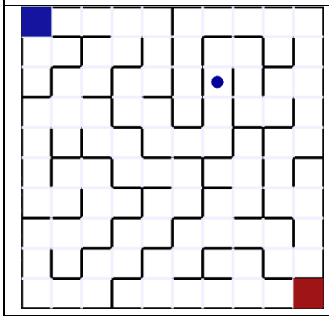
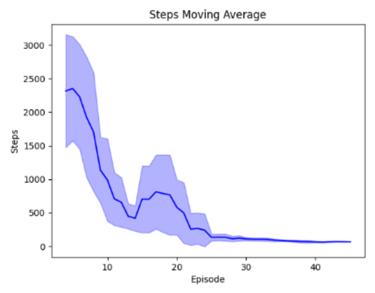
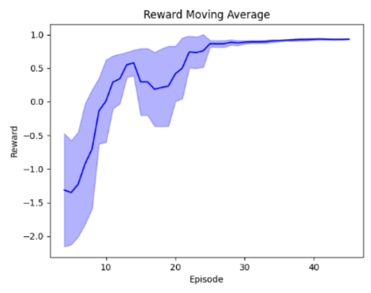
Sample 3x3 Maze Configuration		
		

Sample 5x5 Maze Configuration		
		

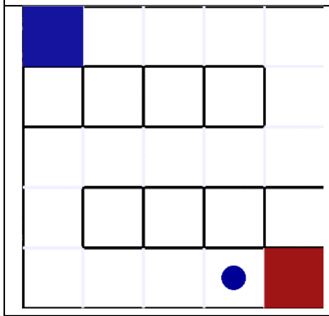
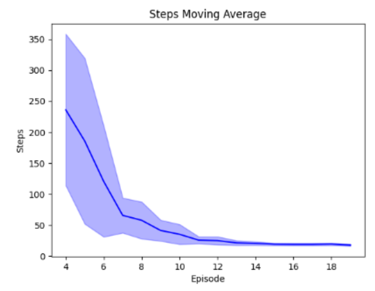
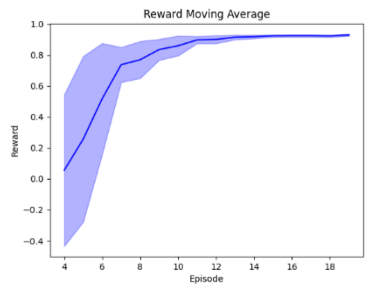
Sample 10x10 Maze Configuration		
		

- Looking at the sample maze configurations, the results start to form a pattern, where the agent takes lot of steps in the starting sequence of episodes which is expected to be happening.
- But however, for a simple maze structure defined in the Sample 3X3 maze, which took about 3 timesteps to reach the end point specified in the learning code.

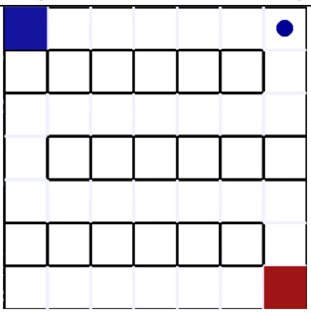
- Comparing in terms of the steps taken to achieve the goal state, in the sample 3X3 case, a steady decrease in the steps taken to achieve the goal state was decreasing with the agent being able to learn the maze configuration pretty well.
- But it took about 14 episodes to discover the reward signal correlation and adjust the positioning of the moves being taken.
- In an unexpected situation, the sample 5X5 configuration was seen to be having trouble in the update in the episodes 17-22 range.
    - This corresponded to be having a worse variance around the position of the steps taken.
    - It was also seen in the case of reward signal behaviour as they both are interlinked with each other. More the steps taken, impact the reward signal because we have used discounting mechanism for the future rewards.
    - This means that, more the steps taken, the sum of cumulative rewards earned by the agent goes down pretty quickly with the factor less than one is multiplied multiple times to the future reward scalar values.
- In the case of 10X10 maze configuration, the reward and step moving averages showed a more noisy behaviour. This is because, agent has more routes to travel to reach the end point for the maximum rewards.
- In the end episodes ranges, the steps ordinate wasn't varying much as the reward signal which is good situation for the agent to receive a greater cumulative reward.

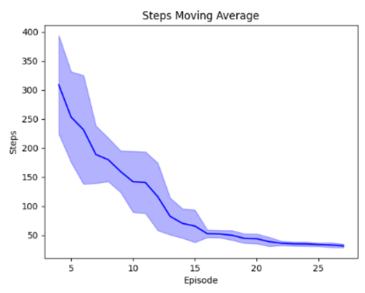
Random 10x10 Maze Configuration		
		

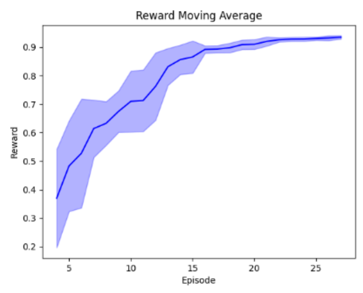
- This random configuration has more twists and turns compared to the sample maze configurations, also, in the later stages of the maze.
- This is pretty evident from the behaviour in the plots, where the agent took more than 2200 steps in the initial successful winning event, and until 25 episodes where the agent comes to terms with the maze reward system and complete with less than 100 steps.
- In the end, the agent took about 69 steps to finish the maze, also with about +1 overall reward, starting out with a negative reward.
- In the end, the agent kinds of level outs to an optimum step and reward count.

Regular 5x5 Maze Configuration		
		

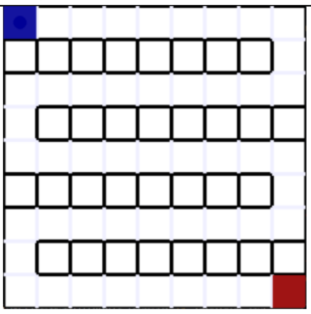
Regular 7x7 Maze Configuration

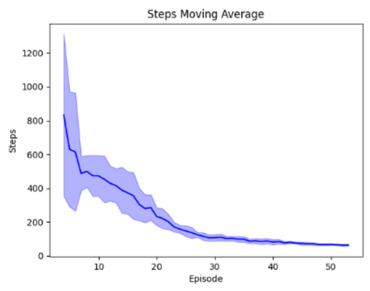


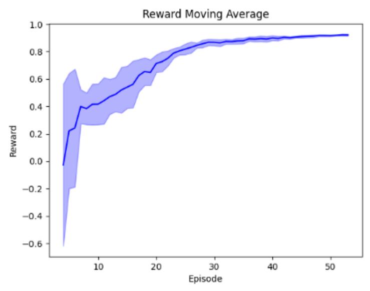




Regular 9x9 Maze Configuration







Coming to the most interesting part of the maze configuration, here are the observations of the regular mazes:

- The regular maze configuration doesn’t seem that hard for a human when its visually seen. Most of us, naturally glance through it and observe the expected optimal behaviour from the agent.
- However, the regular mazes are the ones which require the agent to go through each and every open box in the maze to reach the end goal, and eventually end up with highest amount of steps in the lot. Because of that reason alone, direct comparison between the regular maze and sample/random maze configurations cannot be drawn as the agent cannot directly observe the complete action state space as a human could. It can be made possible in the case of more advanced algorithms, where DeepMind used Deep Reinforcement Learning to beat the human baselines in various Atari games with feeding in the pixels through deep convolutional neural networks.
- In this case, with the traditional Q-Learning algorithm, regular mazes surprisingly took less steps compared to the sample maze configuration, and had a less noisy reward average across the steps. Also, in the case of few sample/random mazes, the model performance in terms of lower steps and higher rewards was altered with worse outcomes, which is not the case in terms of regular mazes.
- There is less variance observed in the case of regular mazes, which is expected as the action space considered by the agent is hugely dependent on the previous reward/time step duration. Once the agent is able to figure out the direction of change to maximize reward, the policy to the future reward is ameliorated to lower rates and the path with lower timesteps is chosen.
- In regular maze, this could be attributed to the fact that the agent kind of moves back and forth which is more redundant in nature and unnecessary in terms of maximizing the overall reward.

Comparison

Table: Maze configurations comparison with the number of moves and the total runtime.

Maze Variant	**Number of Timesteps**	**Number of Episodes**	**Total Reward**
Sample 3x3	3	14	0.967
Sample 5x5	13	28	0.948
Sample 10x10	73	76	0.927
Random 10x10	69	46	0.931
Regular 5x5	16	20	0.936
Regular 7x7	28	28	0.942857
Regular 9x9	59	54	0.927160

Summary

- The agent was trained on various set of maze configurations and reward, step stats were analysed.
    - The maze configurations selected were from the baseline samples provided, random mazes generated, and the regular maze configurations generated with a straightforward routed with 180 degree turns to sort of confuse the agent to make redundant steps to fig out the optimal direction of moves.
- The agents trained in the case of Q-Learning were observed to be having a noisier reward signal compared to some simpler maze configurations. The reward signal can be noisy, meaning that it contains errors or fluctuations that are not related to the actual performance of the agent.
- Noisy reward signals can make it difficult for RL agents to learn the optimal behaviour since the agent may be incorrectly rewarded for actions that do not actually lead to a good outcome. In some cases, noisy rewards can even lead the agent to learn the wrong behaviour entirely.

Deep Reinforcement Learning

Unlike the use of policy iterative Q-value updating methods being used in the Q-Learning strategies, Deep reinforcement learning uses neural networks to approximate the value function and policy function that are used in reinforcement learning.

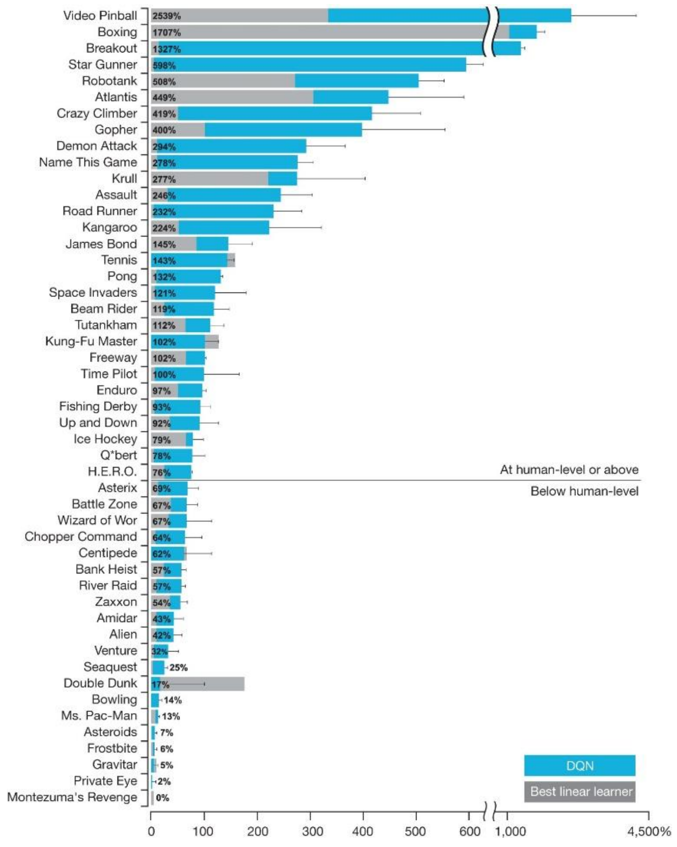


Figure: DQN has been a significant breakthrough in the reinforcement learning platform. The original paper mentions its dominant performance compared to a best linear learner (traditional), with human as a baseline [4].

Algorithms

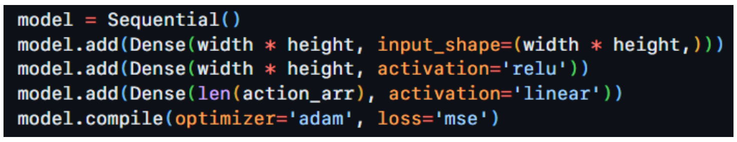
Some of the most widely used deep reinforcement learning algorithms include:

- **Deep Q-Networks (DQN):** DQN is a deep neural network that uses Q-learning, a classic reinforcement learning algorithm, to learn the optimal policy for a given task.
- **Actor-Critic Methods:** Actor-critic methods combine value-based and policy-based approaches by using two neural networks: an actor network that learns the policy and a critic network that learns the value function. Examples of actor-critic methods include A2C (Advantage Actor-Critic) and A3C (Asynchronous Advantage Actor-Critic).
- **Proximal Policy Optimization (PPO):** PPO is a policy optimization method that trains a policy function by updating the parameters in small batches, which makes it more stable and less prone to overfitting [5]. This algorithm was implemented by OpenAI to beat the best DOTA-2 players. It was also used to train the robot hand to solve the Rubik's cube.
- **Trust Region Policy Optimization (TRPO):** TRPO is a policy optimization method that uses a trust region constraint to limit the size of the policy update. This helps prevent the policy from changing too much and destabilizing the learning process. It was used by the Pieter Abbeel's group at UC Berkeley to solve the 2-D locomotion problem [6].

- **Deep Deterministic Policy Gradient (DDPG)**: DDPG is an actor-critic method that uses a deterministic policy function to learn the optimal policy. It has been used to train robots to perform complex manipulation tasks [7].

Deep Q-Networks

The model in the case of Deep Q-Network is expected to approximate the action-value function (Q-Function) using neural networks rather than updating the Q-Table. During training, the network is updated to minimize the difference between the predicted Q-values and the true Q-values, which are obtained using the Bellman equation. The Bellman equation is a recursive formula that expresses the Q-value for a state-action pair as the sum of the immediate reward and the discounted Q-value of the next state. One of the key advantages of DQN is that it can learn directly from raw sensory input, such as pixels in an image, without the need for manual feature engineering. It has been used to achieve human-level performance in Atari games [8].

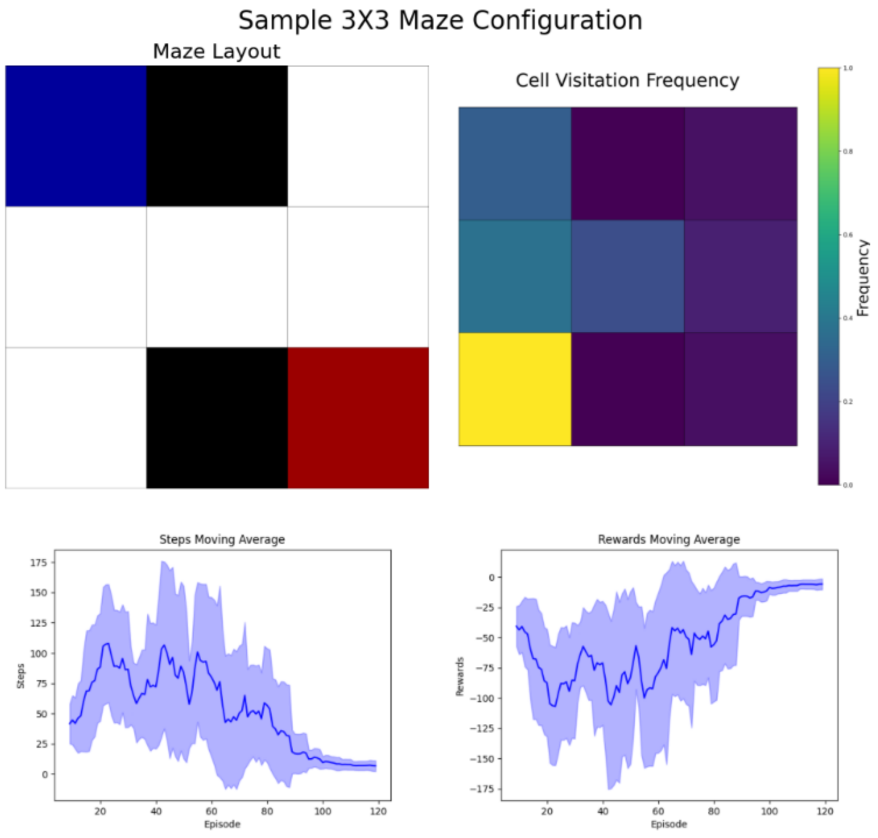


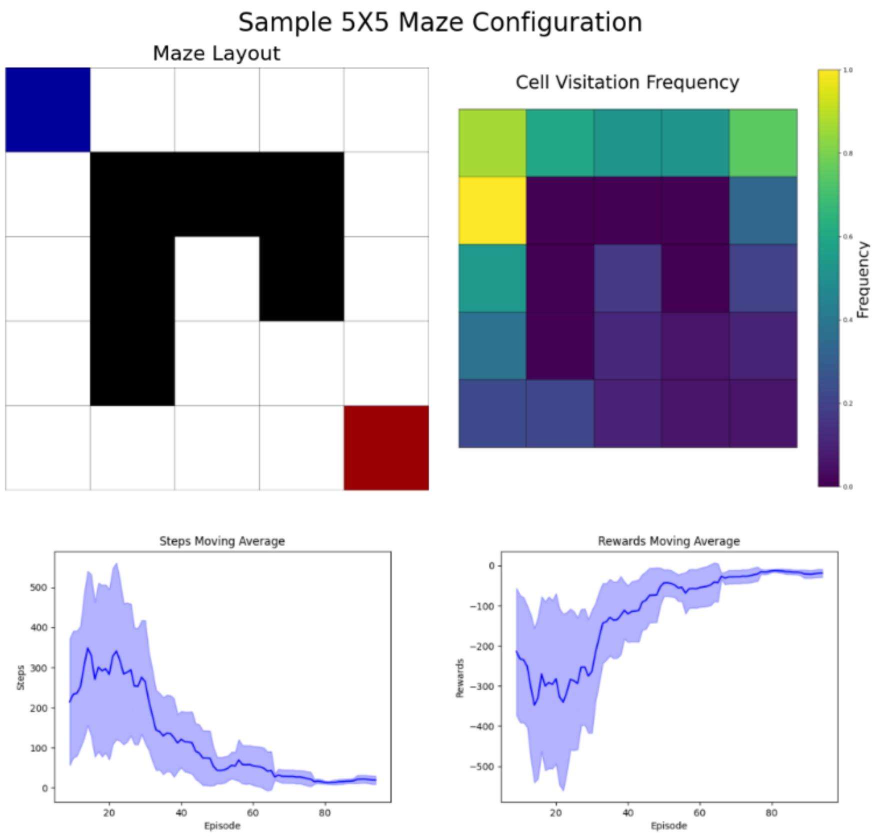
```
model = Sequential()
model.add(Dense(width * height, input_shape=(width * height,)))
model.add(Dense(width * height, activation='relu'))
model.add(Dense(len(action_arr), activation='linear'))
model.compile(optimizer='adam', loss='mse')
```

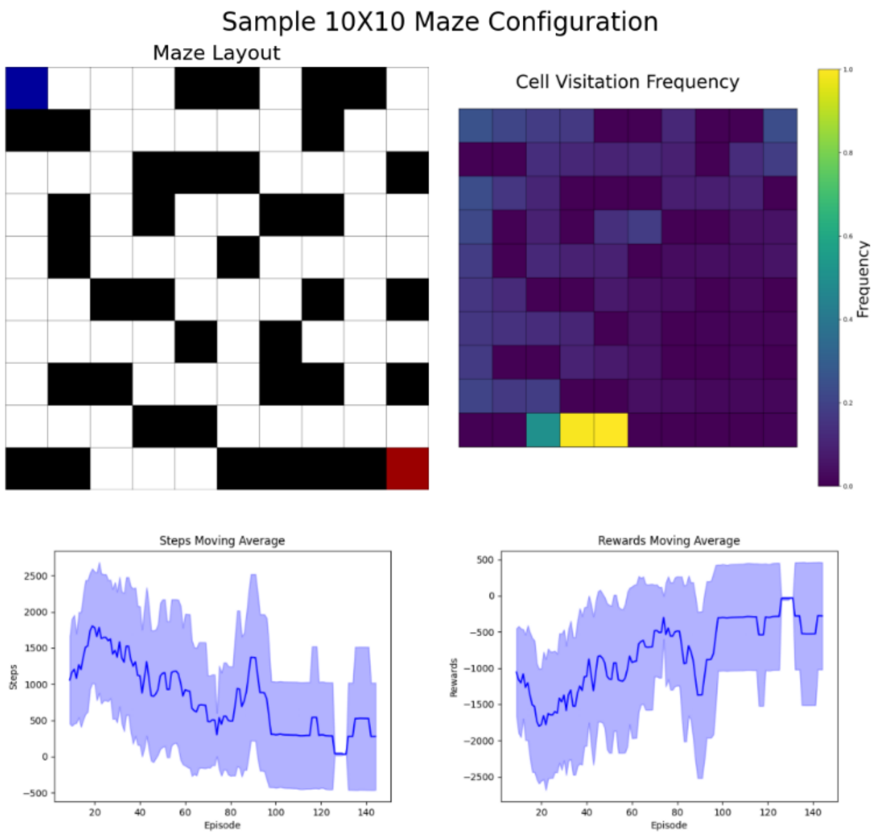
Figure: Deep Q-Network model implementation details.

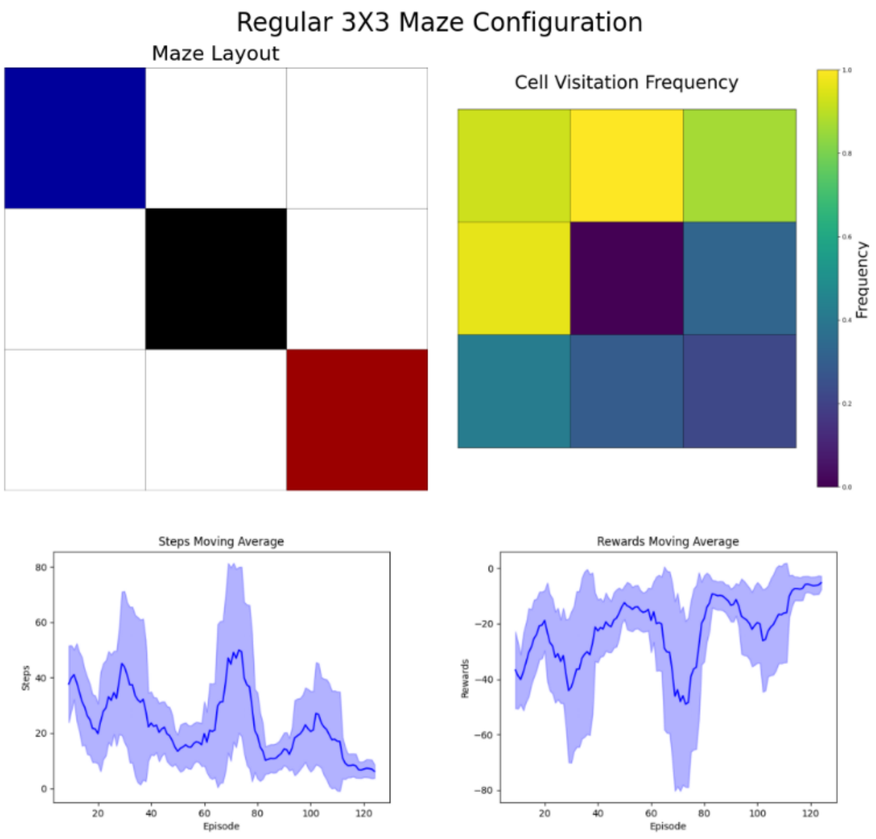
The Deep Q-Network used for training the agent uses a simple full connected layer network with a single hidden layer. The output layer is with a linear activation function. The model architecture in terms of the number of neurons/units in each layer is dependent on the maze dimensions specified in the training procedure.

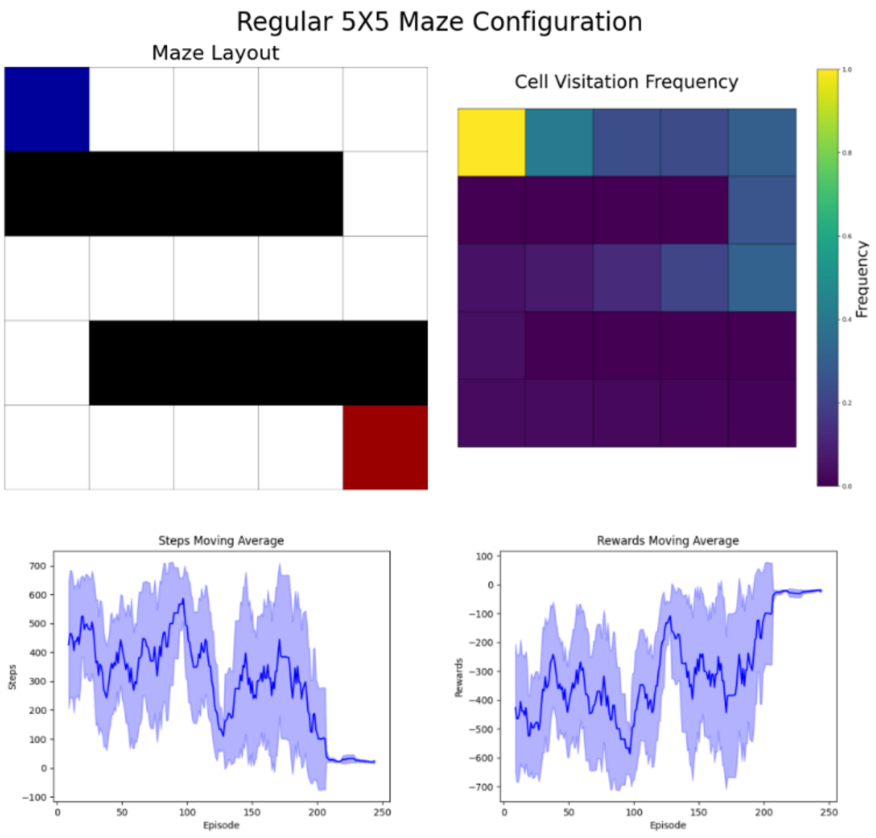
Results

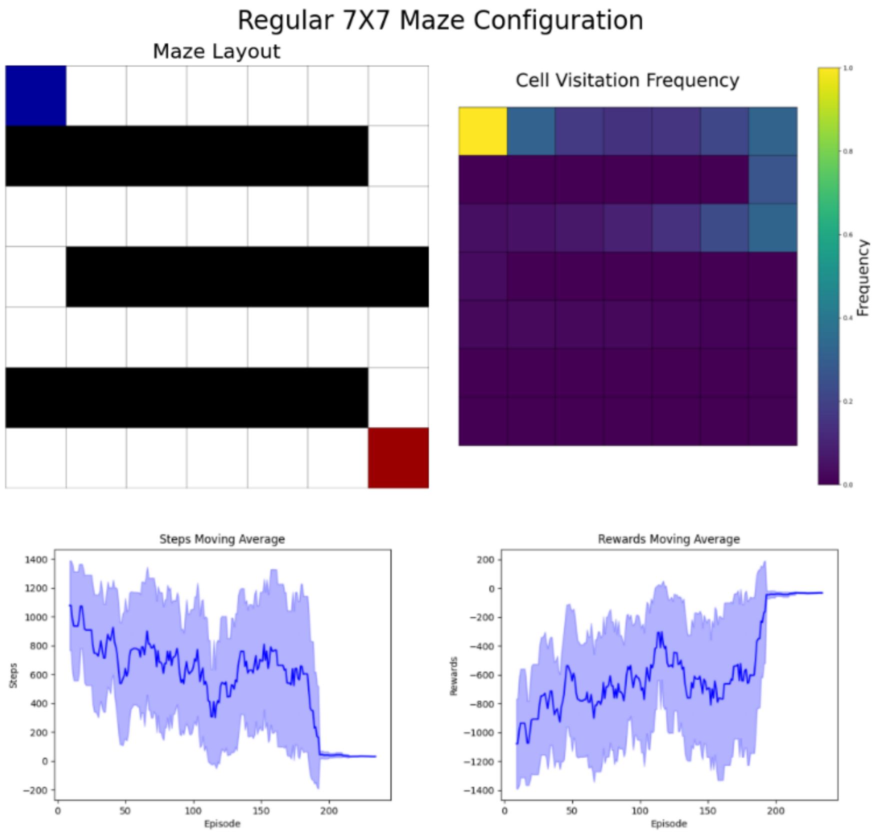


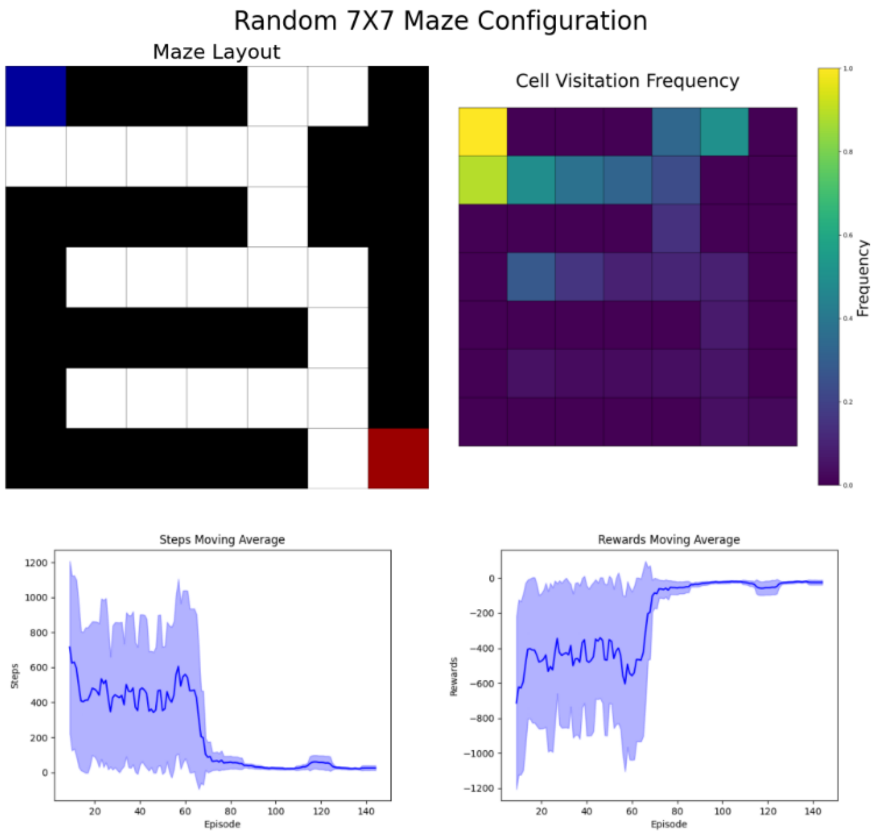

Sample 5X5 Maze Configuration
Maze Layout
Cell Visitation Frequency
Frequency
Steps Moving Average
Steps
Episode
Rewards Moving Average
Rewards
Episode

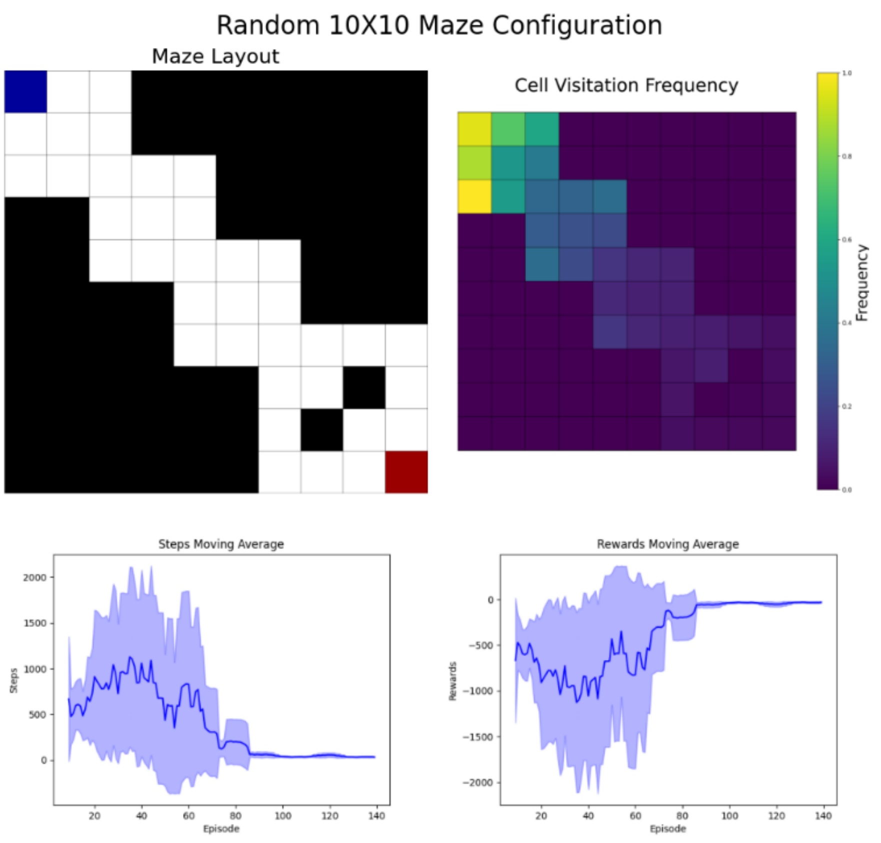

Sample 10X10 Maze Configuration
Maze Layout
Cell Visitation Frequency
Frequency
Steps Moving Average
Steps
Episode
Rewards Moving Average
Rewards
Episode


Regular 3X3 Maze Configuration
Maze Layout
Cell Visitation Frequency
Frequency
Steps Moving Average
Steps
Episode
Rewards Moving Average
Rewards
Episode


Regular 5X5 Maze Configuration
Maze Layout
Cell Visitation Frequency
Frequency
Steps Moving Average
Steps
Episode
Rewards Moving Average
Rewards
Episode


Regular 7X7 Maze Configuration
Maze Layout
Cell Visitation Frequency
Frequency
Steps Moving Average
Steps
Episode
Rewards Moving Average
Rewards
Episode


Random 7X7 Maze Configuration
Maze Layout
Cell Visitation Frequency
Frequency
Steps Moving Average
Steps
Episode
Rewards Moving Average
Rewards
Episode



Observations

- Compared to the Q-Learning, deep reinforcement learning methods applied to the maze configurations took unusually long episodes to train the agent to the optimum stage to get the maximum reward.
- This is expected to happen as the model is optimized with traditional mean squared loss and gradient based update rule. This optimization procedure could take longer time to reach the optimum value and is highly dependent on the hyperparameters chosen in the training the algorithm. For example, we know that the updates observed which running a stochastic gradient descent update mechanism is highly noisy while updating the parameters with respect to the criterion/loss function. It's also similar in this case, where the steps and reward signal are very noisy compared to the Q-table procedure.
- Going into more specifics, sample mazes took more episodes for the agent to reach the optimum number of steps. Also, the total runtime is random in the case of sample mazes, as the 5X5 maze took less time compared to the 3X3 maze.
- In terms of the noisy behaviour reasoning could be attributed to:
    - The number of steps required for an agent to learn the optimal policy can be noisy, meaning that the performance of the agent may vary significantly from episode to episode.
    - This can happen for a variety of reasons, including the stochasticity of the environment, the exploration-exploitation trade-off, or the learning rate used by the algorithm.
    - The agent may start with poor performance as it explores the environment and tries to learn the optimal policy. As it gains experience and refines its policy, its performance may improve.

    - However, as the agent becomes more confident in its actions, it may start to exploit its knowledge too much, leading to a decrease in performance. This trade-off between exploration and exploitation is a key challenge in reinforcement learning.
    - Various techniques have been developed, such as using experience replay to reduce the variance in the updates, using annealing schedules updates to adjust the learning rate, or using exploration strategies such as epsilon-greedy or Boltzmann exploration to encourage the provided agent to explore more in case of maze configurations.
- In the case of regular mazes, the agent took more time than the sample counterparts, which is expected as the number of action/steps required has increased by a large factor and the agent needs to visit each and every open space.
    - Similar to Q-Learning, the agent is observed to cut down the redundant steps of moving back and forth along the straight maze routes with increasing episodes.
    - This is attributed to the fact that agent is learning from the neural network and action is provides.
    - Also, I observed that 9X9 regular maze search took about 45-50 min and the model wasn't able to converge to the end point. This is unexpected in this case, as the regular maze behaviour should be similar to the ones with different numbers of moves and higher runtime.
- The agent not able to converge to a optimum point could be reasoned as follows:
    - The exploration strategy may not be appropriate. If the agent is only exploring a small portion of the maze, it may be missing important parts of the environment that are necessary to learn the optimal policy. In this case, adjusting the exploration strategy, such as increasing the exploration rate or using a more sophisticated exploration method, can help the agent to discover new parts of the maze and improve its performance.
        - This shouldn't be the case, as the agent was able to figure out the 5X5 and 7X7 regular mazes in a reasonable runtime and moves.
    - Another reason could be that the network architecture may not be suitable.
        - If the network is too simple, it may not be able to capture the complex relationships between the agent's actions and the environment.
        - In this case, increasing the network size or using a more sophisticated architecture, such as a recurrent neural network or a convolutional neural network, can help the agent to learn the optimal policy.
- Compared to the regular mazes, specialized random mazes were found to be solvable by the agent in relatively low number of episodes (in the range of 140-160 compared to >220 in the case of regular mazes). This could be because that they have more broader route space for the agent to move around and figure out the optimum set of actions.
- Even, in the rewards and steps plots, a sharp rise in the agent performance is observed.
    - One possible reason could be that the exploration strategy used by the agent. If the agent is using a strategy that encourages exploration, such as epsilon-greedy or Boltzmann exploration, it may eventually discover a new strategy or state that leads to a significant increase in performance. But that's not true in this case.
    - Another possibility could be since, the experience replay procedure has kicked in the agent actions. Experience replay allows the agent to learn from past experiences and update its policy based on a sample of previous transitions. This can help the agent to discover new strategies that it may not have encountered during its initial exploration of the environment. This is inherently built into the Q-Network algorithm, so this might be a possible reason for the observed behaviour.

Comparison

Table: Maze configurations comparison with the number of moves and the total runtime.

File Name	**Number of Moves**	**Total Runtime (min)**
Maze-sample-3x3.csv	4	0.556
Maze-sample-5x5.csv	8	0.462
Maze-sample-10x10.csv	18	0.993
Maze-regular-3x3.csv	4	0.583
Maze-regular-5x5.csv	16	1.281
Maze-regular-7x7.csv	24	0.896
Maze-regular-9x9.csv	DID NOT CONVERGE	DID NOT CONVERGE
Maze-random-7x7.csv	12	0.753
Maze-random-10x10.csv	18	0.820
Maze-random-15x15.csv	28	4.750

Summary

- Like Q-Learning, Deep Reinforcement Learning suffered with a noisy reward signal from each of the agent trained on the different maze configurations.
- It was more pronounced in the cases of larger maze configurations, as the agent needs to explore through various routes to get to the winning position.
    - To address this problem, there are several techniques that can be used in RL to mitigate the effects of noisy rewards. One approach is to use smoothing or filtering techniques to reduce the impact of noise in the reward signal. Another approach is to use techniques such as temporal difference learning, which can help the agent to learn from experience and adjust its behaviour even in the presence of noisy rewards.
    - Another technique to mitigate the effects of noisy rewards is to use more robust RL algorithms such as distributional RL or uncertainty-aware RL, which can handle uncertainties in the reward signal more effectively.
    - Overall, dealing with noisy reward signals in RL is an important research area, and there are many ongoing efforts to develop more effective methods for training agents in noisy environments.
- Also, we observed that there was sharp rise in the agent performance at a certain level of episode range.
    - Sharp rise in agent’s performance may not always be indicative of a successful learning process.
    - It's possible that the agent has overfit to a particular state or strategy, which may not generalize well to other situations.
    - It's important to monitor the performance of the agent over a longer period of time and ensure that it is learning a robust and generalizable policy, rather than just random behaviour is being encoded.

SOTA: Maze Solving

For maze solving, we have explored the use of classical Q-Learning method and Deep Q-Learning using Deep Neural Networks for optimizing the Q-Function specified. But these methods are not considered to be state-of-the-art methods that could be having a wide range of impact on various reinforcement learning tasks. Considering the scope of reinforcement learning being applied to various tasks that involve large amounts of reliable output in the feedback loop to the environment in terms of actions from the right policy function point of view, advanced training strategies have been implemented. Adding to the fact that, we're currently comparing the implemented algorithms to the SOTA, the algorithm and the maze solving tasks benchmarks are reported as follows.

The state of the art in Deep Maze solving is by using a Dreaming Variational Autoencoder with the Proximal Policy Optimization Algorithm.

Dreaming Variational Autoencoder

The Dreaming Variational Autoencoder (DVAE) is a deep neural network based on generative modelling for exploring in environments that provide a sparse reward [9].

Maze solving in general, is sort of a tasks that provides the user with a limited output, just a scalar value of the reward without giving the opportunity to debug the output provided at the extent that unsupervised learning algorithms provide through the thorough visualization of the lagging model output.

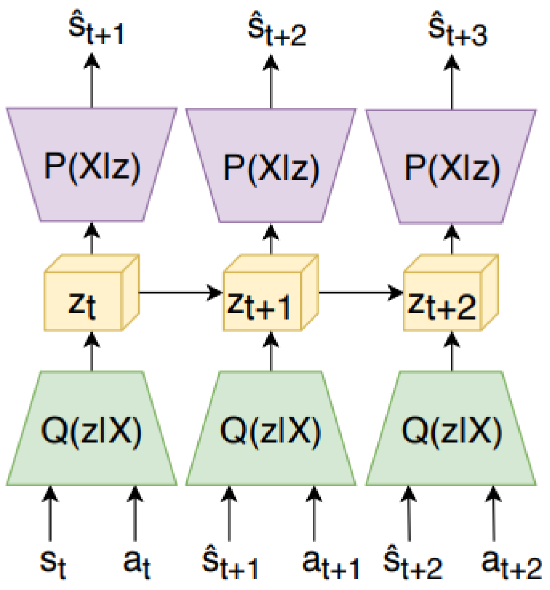
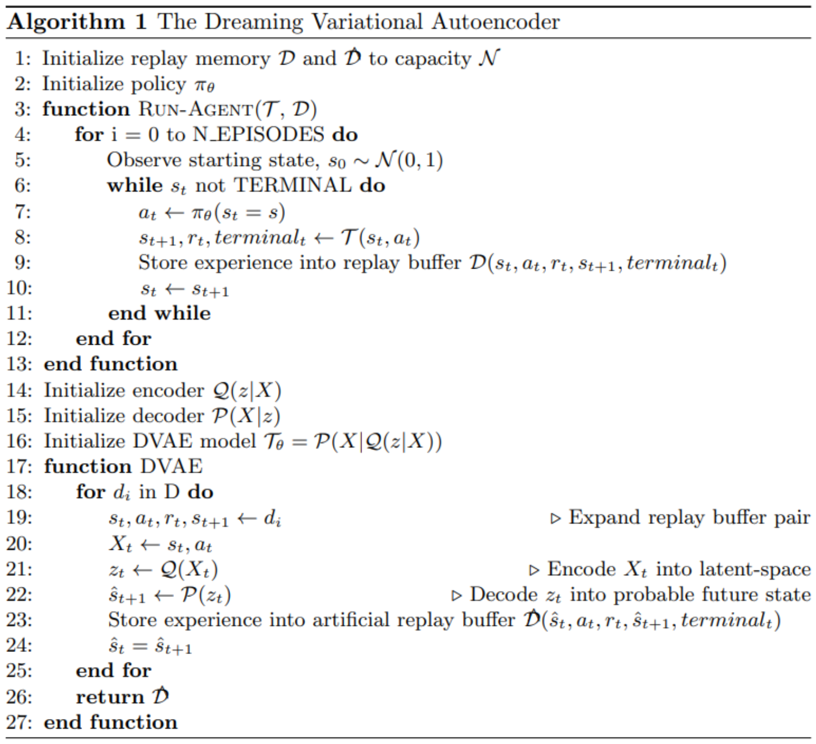


Figure: Illustration of the Dreaming Variational Autoencoder model.

The DVAE model consumes a state and action pairs, yielding the input encoded in a latent space (Q in the Figure is the Encoder). The encoder, Q transforms the input space to a lower-dimensional latent space. The latent space can then be decoded to a probable future state with the help of a decoder, P. The DVAE model could also be implemented with the use of Long Short Term Memory networks (LSTM) [10] to better learn the longer sequences in continuous time domain spaces.



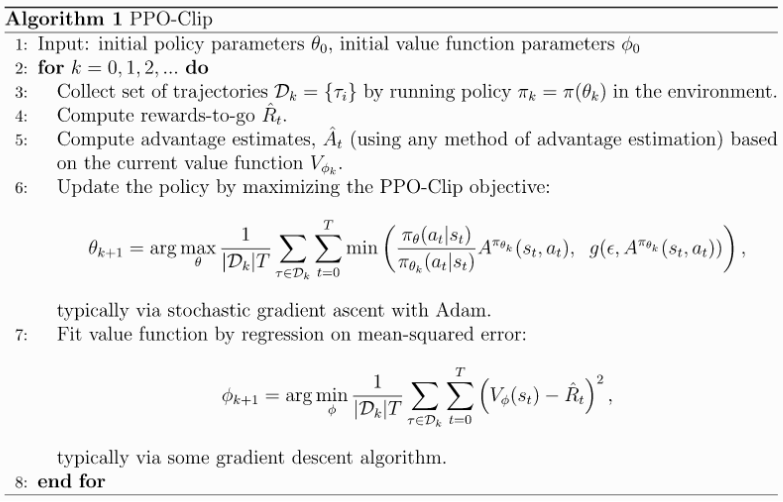
```
Algorithm 1 The Dreaming Variational Autoencoder
1: Initialize replay memory $\mathcal{D}$ and $\hat{\mathcal{D}}$ to capacity $\mathcal{N}$
2: Initialize policy $\pi_\theta$
3: function RUN-AGENT($\mathcal{T}$, $\mathcal{D}$)
4:     for i = 0 to N_EPISODES do
5:         Observe starting state, $s_0 \sim \mathcal{N}(0,1)$
6:         while $s_t$ not TERMINAL do
7:             $a_t \leftarrow \pi_\theta(s_t = s)$
8:             $s_{t+1}, r_t, terminal_t \leftarrow \mathcal{T}(s_t, a_t)$
9:             Store experience into replay buffer $\mathcal{D}(s_t, a_t, r_t, s_{t+1}, terminal_t)$
10:            $s_t \leftarrow s_{t+1}$
11:        end while
12:    end for
13: end function
14: Initialize encoder $\mathcal{Q}(z|X)$
15: Initialize decoder $\mathcal{P}(X|z)$
16: Initialize DVAE model $\mathcal{T}_\theta = \mathcal{P}(X|\mathcal{Q}(z|X))$
17: function DVAE
18:    for $d_i$ in D do
19:        $s_t, a_t, r_t, s_{t+1} \leftarrow d_i$                    ▷ Expand replay buffer pair
20:        $X_t \leftarrow s_t, a_t$
21:        $z_t \leftarrow \mathcal{Q}(X_t)$                          ▷ Encode $X_t$ into latent-space
22:        $\hat{s}_{t+1} \leftarrow \mathcal{P}(z_t)$               ▷ Decode $z_t$ into probable future state
23:        Store experience into artificial replay buffer $\hat{\mathcal{D}}(\hat{s}_t, a_t, r_t, \hat{s}_{t+1}, terminal_t)$
24:        $\hat{s}_t = \hat{s}_{t+1}$
25:    end for
26:    return $\hat{\mathcal{D}}$
27: end function
```

Figure: Algorithm for training the DVAE Model.

Proximal Policy Optimization

PPO (Proximal Policy Optimization) is a reinforcement learning algorithm that is used to optimize the policy of an agent in an environment. The PPO algorithm belongs to the family of actor-critic algorithms, which means it has both an actor and a critic-component. Generally, in PPO, the actor component is responsible for selecting actions based on the current state of the environment, and the critic-component evaluates the quality of the actions taken by the actor. The goal of PPO is to find the optimal policy that maximizes the expected reward received by the agent over time.

One of the key features of PPO is the use of a clipping mechanism to ensure that the new policy does not deviate too much from the previous policy. This helps to stabilize the training process and prevent the agent from making large policy updates that could lead to an unusual and catastrophic performance.

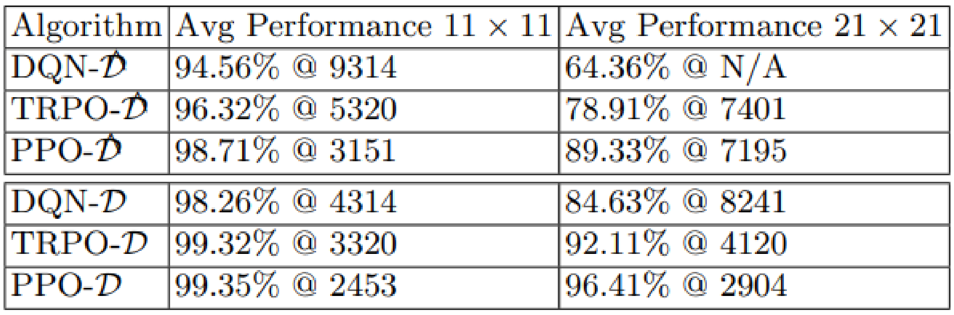


```
Algorithm 1 PPO-Clip
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
       $$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
       $$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$
       typically via some gradient descent algorithm.
8: end for
```

Figure: PPO-Clip algorithm. Gradient clipping is applied in this case to ensure policy doesn't get trapped in local optima.

PPO has been used successfully in a variety of applications, including robotics, game playing, and natural language processing. It is particularly well-suited for problems with continuous action spaces, where traditional RL algorithms like Q-learning may struggle.

Results



Algorithm	Avg Performance 11×11	Avg Performance 21×21
DQN-$\not\mathcal{D}$	94.56% @ 9314	64.36% @ N/A
TRPO-$\not\mathcal{D}$	96.32% @ 5320	78.91% @ 7401
PPO-$\not\mathcal{D}$	98.71% @ 3151	89.33% @ 7195
DQN-$\mathcal{D}$	98.26% @ 4314	84.63% @ 8241
TRPO-$\mathcal{D}$	99.32% @ 3320	92.11% @ 4120
PPO-$\mathcal{D}$	99.35% @ 2453	96.41% @ 2904

Figure: The results of the various Deep Maze configurations.

These results are obtained for 2 Deep Maze configurations, with the agent trained on Vanilla Deep-Q Network, Trust Region Policy Optimization, and Proximal Policy Optimization algorithms. The optimal path yields a 100% while having no solution through convergence leads to 0%. Each of the algorithms ran for 10000 episodes for both the map configuration. In the results section, first value is the average performance achieved by the agent compared to the most optimal pathway and the second value represents the number of episodes used by the agent to get to that greatest performance.

References

[1] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning,* vol. 8, no. 3, pp. 279-292, 1992/05/01 1992, doi: 10.1007/BF00992698.

[2] J. Schulman, "Deep Reinforcement Learning Bootcamp," 2015. [Online]. Available: https://rll.berkeley.edu/deeprlcourse-fa15/docs/mdp-cheatsheet.pdf.

[3] G. Brockman *et al.*, "Openai gym," *arXiv preprint arXiv:1606.01540,* 2016.

[4] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature,* vol. 518, no. 7540, pp. 529-533, 2015/02/01 2015, doi: 10.1038/nature14236.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347,* 2017.

[6] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015: PMLR, pp. 1889-1897.

[7] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971,* 2015.

[8] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602,* 2013.

[9] P.-A. Andersen, M. Goodwin, and O.-C. Granmo, "The dreaming variational autoencoder for reinforcement learning environments," in *Artificial Intelligence XXXV: 38th SGAI International Conference on Artificial Intelligence, AI 2018, Cambridge, UK, December 11–13, 2018, Proceedings 38*, 2018: Springer, pp. 143-155.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation,* vol. 9, no. 8, pp. 1735-1780, 1997.