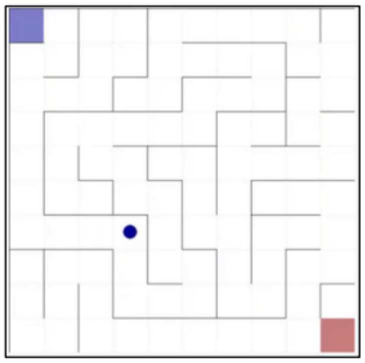
Reinforcement Learning Mid Assignment

By Elad Prager & Shira Moscovitch

General description

In this project we tried to solve the NxN maze problem with different reinforcement learning algorithms.
The objective of the task is to bring the agent from the start point at the left top corner on the maze (in blue) to the bottom right corner (in red) in minimum steps



- There are 4 possible **actions**, moving up, down, right or left ("N", "S", "E", "W").
  When the agent wants to move into a wall, it stays at the same location.
- The **transition model** is stochastic with probability of 0.9, meaning, when the agent wants to move to a specific direction it will move there with probability of 0.9 or will move to another direction with probability of 0.1 (distributed uniformly between all the other 3 directions).
- The **observation space** (the states of the model) is given by the (x,y) coordinates of the maze board, meaning we have NxN states
- The rewards are:
  - 1 → if the agent got to the goal
  - $-0.1/size\ of\ the\ board$ for every other step

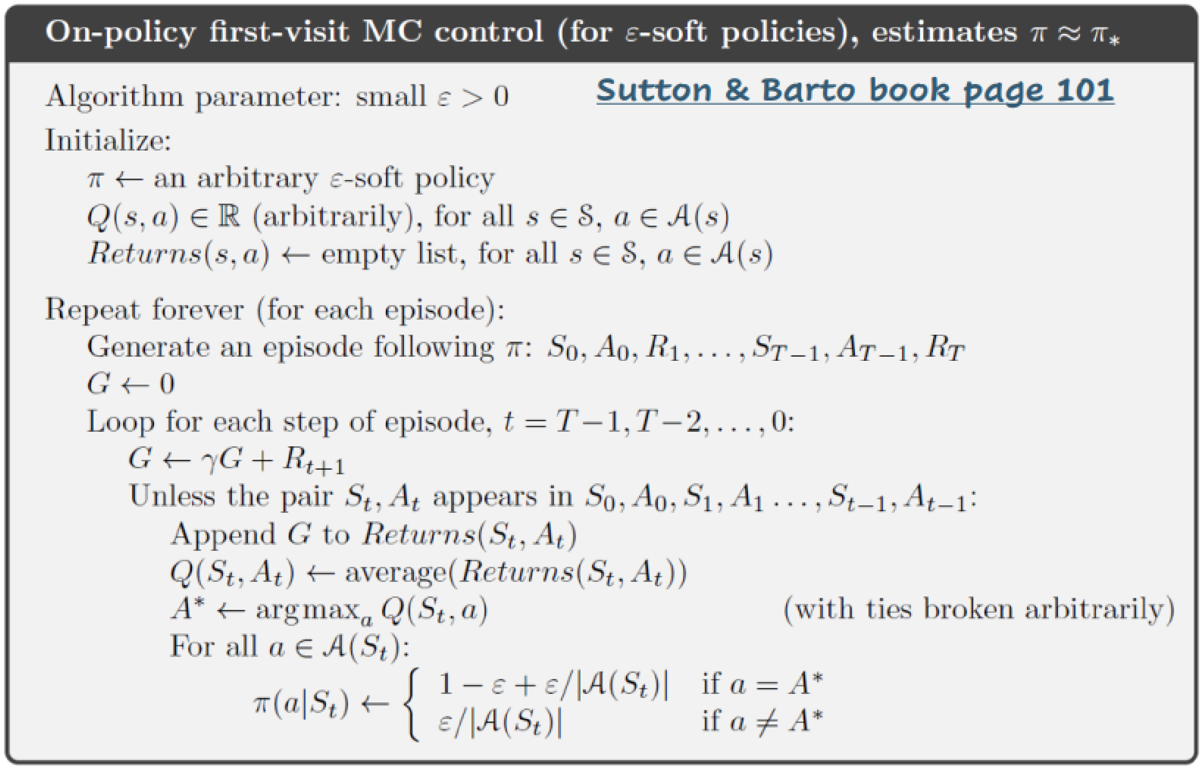
  We experiment different rewards along this project

In this project we assumed no prior information about the environment and all the information is essentially collected by experience during the learning episodes. The project is implemented with OpenAI Gym toolkit in Python.

Monte-Carlo on 15x15 board

The Monte Carlo method for reinforcement learning learns directly from episodes of experience without any prior knowledge of MDP transitions. Every episodic MDP is running fully (until the end state) before we calculate any returns. We don't update after every action but after every episode. We update the value with the mean return of all sample trajectories for each state.

We used on policy first visit monte-carlo with ε-soft policies, which is given by:



On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Sutton & Barto book page 101

```
Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
            $A^* \leftarrow \arg\max_a Q(S_t, a)$                (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

This algorithm was used to solve the 15X15 board problem.

Stochastic transition model

As written in the exercise instructions we added stochasticity to the model. Now, the agent will take a random action in a probability of 13.333%, Since a random action includes all actions (including the action given by the policy), the probability for a the non greedy actions is: 13.333%/4*3=10%, and the probability for a greedy action is: 86.666% + 13.333%/4 = 90%, as requested.
Furthermore, we choose to add this chunk of code within the episode train loop for each algorithm (and not over the MazeEnv.step function), In order to also have the flexibility to perform a deterministic run to more simply examine the optimal route.

```
a = env.ACTION[action]
if is_stochastic:
  n = np.random.uniform(0, 1)
  if n < (4/30):
    a = env.action_space.sample()
```

All the models were trained with a deterministic environment first to stabilize the algorithms.

Data structures

To implement the MC algorithm we used few data structures

For policy we used a dictionary such that every state has its own dict of probabilities per action. The policy was initiated by a uniform distribution for every action for every state with the function

```
#Function for Random Policy
def create_random_policy(env, action_size, state_size):
# we will build a random policy dictionary where the states are the keys
# each key will have a dictionary of probability per action

  policy = {}
  for key in range(state_size):
      p = {} #probability dict
      for action in range(action_size):
            p[action] = 1 / action_size  #start with uniform distribution
      policy[key] = p
  return policy
```

To ensure continual exploration the policy is updated with ε-soft greedy method meaning all actions will have non-zero probability. We choose the action which maximizes the action value function with probability 1 – epsilon and with probability epsilon choose an action at random.

Althose this exploration method was very useful in converging the first episodes, the model failed to converge to the minimal path since we continued high exploration even when we were close to the optimal path, this is the known Exploration-Exploitation Dilemma. As training continues, we need to balance exploitation versus exploration, we want to make sure our agent doesn't get trapped in a cycle going from one square to another, back and forth. We also don't want our agent permanently choosing random values. We'll use the function below to try to balance this:

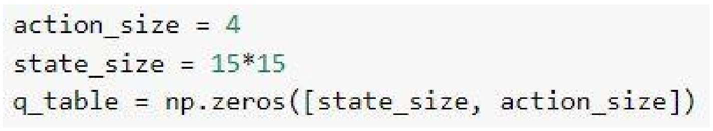
```
def reduce_epsilon(epsilon,epoch, min_epsilon, decay_rate):
    return min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*epoch)
```

The epsilon decay rate is a very delicate parameter that directly influences how fast we reduce our epsilon. Reduce too fast, the agent won't have enough time to learn. Reduce too slow, we will waste time picking random actions. Also we saw it is important to start from high max_epsilon (=1) to converge to the optimal path.

Another data structure is a table that is essentially mapping all the possible state,action pairs and the expected reward for taking an action at a particular state that we will keep updating ($Q(st,at)$). This means we have 4 actions for columns, and 225 possible states (player location on the 15 by 15 grid). So our table will look like:

	'W' - LEFT	'S' - DOWN	'E' - RIGHT	'N' - UP
State 0	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
State 1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
State ...	...	...	...	...
State 224	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)

In practice, we implemented this structure with a np array as following:



```
action_size = 4
state_size = 15*15
q_table = np.zeros([state_size, action_size])
```

($Q(st,at)$) is updated once per episode with the Average reward across episodes

We built the main function to play an episode (play_game) that runs from start of the game to the end and return all the values of the game (states, actions and rewards) for all the moves of the episode.

We experimented with different initializations for each episode, the parameter random_init was used to determine the percentage of starting from a random state. Though promising, it didn't improve results or convergence and therefore we always started at initial state (0,0).

The algorithm was restricted to a maximal number of states, it was useful for initial convergence of the algorithm, but with the correct parameters it was not necessary.

Model Tuning and parameters selection

After the code was stabilized we set our initial hyperparameters as follows:

```
# Initial Hyperparameters
EPOCHS = 1200
GAMMA = 0.95                  # discount rate
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.005            # Exponential decay rate for exploration prob
```

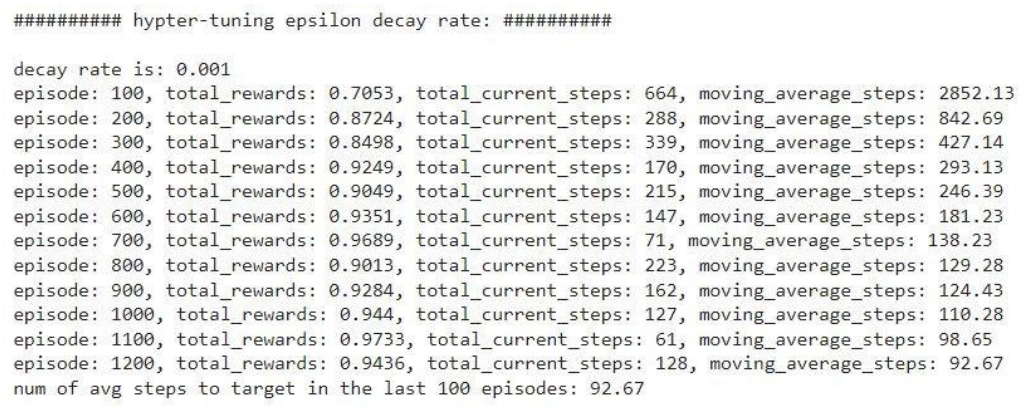
Now, we were ready to run our first initial experiment. This gave us a quick overview before we continue to do more thorough experiments. Fourtantly, that experiment went well, and the model managed to converge.

Next, we considered tuning our initial hyperparameters. Often the best place to choose a good starting value is through experimentation, and therefore, an hyperparameter tuning is required, we consider the following possibilities:

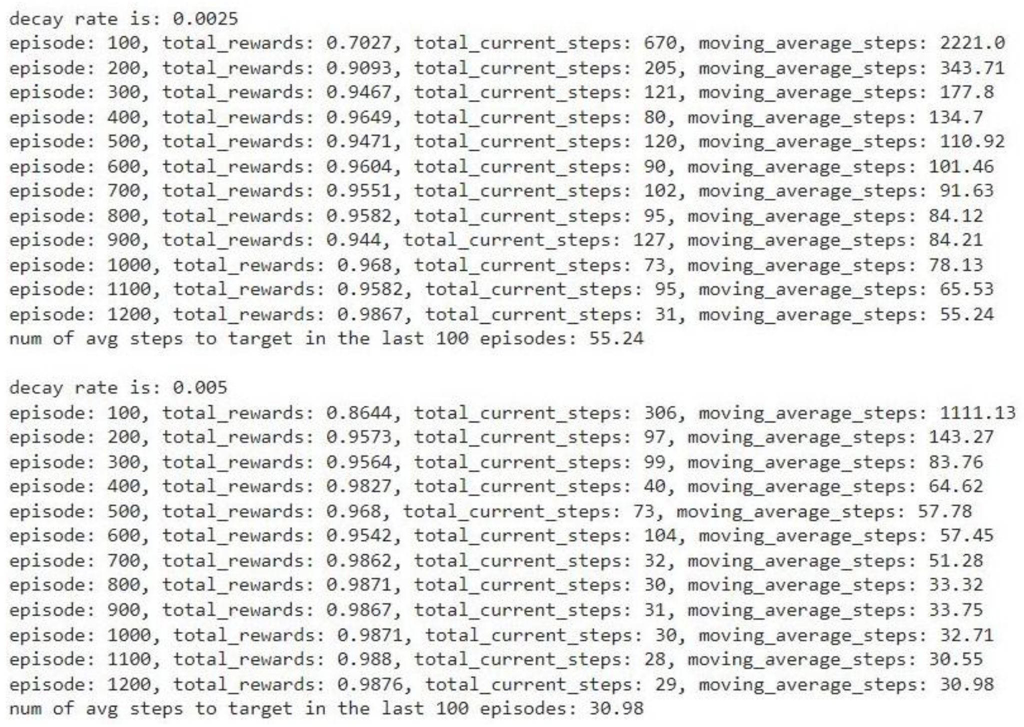
```
# Hyperparameters Tuning
epoch_list = [1000, 1250, 1500, 1750, 2000]
gamma_list = [0.9, 0.925, 0.95, 0.975]
min_epsilon_list = [0.01, 0.03, 0.05, 0.07]
decay_rate_list = [0.001, 0.0025, 0.005, 0.0075]
```

For each of these hyperparameters, we ran a separate experiment, where we track the moving average number of steps for the past 100 episodes. In the next figure we attached an example for the hyperparameter tuning we made for the epsilon decay rate.

As we can see from the last log line in each experiment, there is a significant difference depending on the value of the decay rate. Logically, we choose the values that minimize the number of average steps to the target in the last 100 episodes.



```
########## hypter-tuning epsilon decay rate: ##########

decay rate is: 0.001
episode: 100, total_rewards: 0.7053, total_current_steps: 664, moving_average_steps: 2852.13
episode: 200, total_rewards: 0.8724, total_current_steps: 288, moving_average_steps: 842.69
episode: 300, total_rewards: 0.8498, total_current_steps: 339, moving_average_steps: 427.14
episode: 400, total_rewards: 0.9249, total_current_steps: 170, moving_average_steps: 293.13
episode: 500, total_rewards: 0.9049, total_current_steps: 215, moving_average_steps: 246.39
episode: 600, total_rewards: 0.9351, total_current_steps: 147, moving_average_steps: 181.23
episode: 700, total_rewards: 0.9689, total_current_steps: 71, moving_average_steps: 138.23
episode: 800, total_rewards: 0.9013, total_current_steps: 223, moving_average_steps: 129.28
episode: 900, total_rewards: 0.9284, total_current_steps: 162, moving_average_steps: 124.43
episode: 1000, total_rewards: 0.944, total_current_steps: 127, moving_average_steps: 110.28
episode: 1100, total_rewards: 0.9733, total_current_steps: 61, moving_average_steps: 98.65
episode: 1200, total_rewards: 0.9436, total_current_steps: 128, moving_average_steps: 92.67
num of avg steps to target in the last 100 episodes: 92.67
```

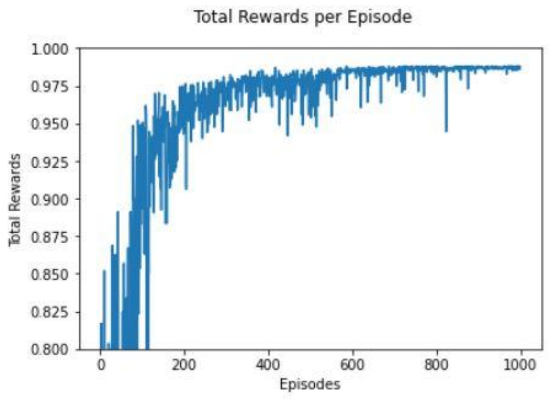


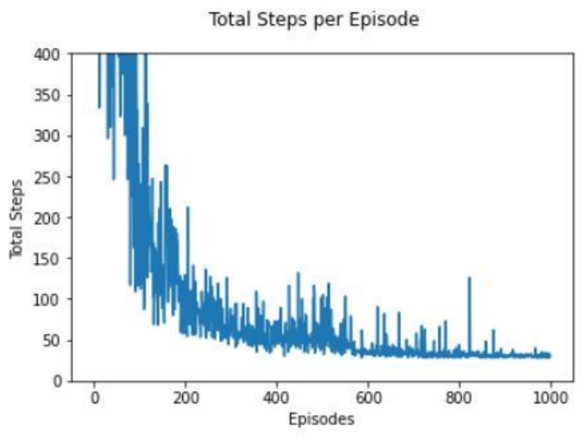
```
decay rate is: 0.0025
episode: 100, total_rewards: 0.7027, total_current_steps: 670, moving_average_steps: 2221.0
episode: 200, total_rewards: 0.9093, total_current_steps: 205, moving_average_steps: 343.71
episode: 300, total_rewards: 0.9467, total_current_steps: 121, moving_average_steps: 177.8
episode: 400, total_rewards: 0.9649, total_current_steps: 80, moving_average_steps: 134.7
episode: 500, total_rewards: 0.9471, total_current_steps: 120, moving_average_steps: 110.92
episode: 600, total_rewards: 0.9604, total_current_steps: 90, moving_average_steps: 101.46
episode: 700, total_rewards: 0.9551, total_current_steps: 102, moving_average_steps: 91.63
episode: 800, total_rewards: 0.9582, total_current_steps: 95, moving_average_steps: 84.12
episode: 900, total_rewards: 0.944, total_current_steps: 127, moving_average_steps: 84.21
episode: 1000, total_rewards: 0.968, total_current_steps: 73, moving_average_steps: 78.13
episode: 1100, total_rewards: 0.9582, total_current_steps: 95, moving_average_steps: 65.53
episode: 1200, total_rewards: 0.9867, total_current_steps: 31, moving_average_steps: 55.24
num of avg steps to target in the last 100 episodes: 55.24

decay rate is: 0.005
episode: 100, total_rewards: 0.8644, total_current_steps: 306, moving_average_steps: 1111.13
episode: 200, total_rewards: 0.9573, total_current_steps: 97, moving_average_steps: 143.27
episode: 300, total_rewards: 0.9564, total_current_steps: 99, moving_average_steps: 83.76
episode: 400, total_rewards: 0.9827, total_current_steps: 40, moving_average_steps: 64.62
episode: 500, total_rewards: 0.968, total_current_steps: 73, moving_average_steps: 57.78
episode: 600, total_rewards: 0.9542, total_current_steps: 104, moving_average_steps: 57.45
episode: 700, total_rewards: 0.9862, total_current_steps: 32, moving_average_steps: 51.28
episode: 800, total_rewards: 0.9871, total_current_steps: 30, moving_average_steps: 33.32
episode: 900, total_rewards: 0.9867, total_current_steps: 31, moving_average_steps: 33.75
episode: 1000, total_rewards: 0.9871, total_current_steps: 30, moving_average_steps: 32.71
episode: 1100, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 30.55
episode: 1200, total_rewards: 0.9876, total_current_steps: 29, moving_average_steps: 30.98
num of avg steps to target in the last 100 episodes: 30.98
```

Eventually, when the experiment test ends, we tuned our hyperparameters accordingly, resulted with the following values:

```
# Convergence after Hyperparameters Tuning
EPOCHS = 1000
GAMMA = 0.9                    # Discount rate
min_epsilon = 0.01             # Minimum exploration probability
decay_rate = 0.005             # Exponential decay rate for exploration prob
```

Now, we were ready to train our model. The tuning improved our results. As we can see from the following figures, we manage to converge. As expected, the total rewards gradually increase over the episodes, and the total steps are gradually decreasing over the episodes.





We made some experiments with different rewards (higher or lower), however they didn't help or ruined convergence and therefore were dropped.

As requested we recorded two videos, one in the middle of training and the second for the last epoch. Since we made many experiments, we used the 'is_tuned' flag to avoid recording during the hyperparameters experiments phase.

```
if episode == (EPOCHS - 1) and is_tuned:
  env = wrap_env(env)
if episode == (EPOCHS/2) and is_tuned and is_stochastic:
  env = wrap_env_middle_of_training(env)
```

Also, we thought it would be nice to record a video with no stochasticity. Then we'll be able to actually see the optimal route, and examine the required minimal number of steps for solving the maze. Therefore, we simply added the 'is_stochastic' flag.

Now we also have the flexibility to perform a deterministic run, and record a final video for the last episode.

Here is the final result for the deterministic experiment:

```
episode: 100, total_rewards: 0.7991, total_current_steps: 453, moving_average_steps: 1108.75
episode: 200, total_rewards: 0.9547, total_current_steps: 103, moving_average_steps: 157.16
episode: 300, total_rewards: 0.9698, total_current_steps: 69, moving_average_steps: 79.34
episode: 400, total_rewards: 0.9742, total_current_steps: 59, moving_average_steps: 60.63
episode: 500, total_rewards: 0.9804, total_current_steps: 45, moving_average_steps: 52.88
episode: 600, total_rewards: 0.9787, total_current_steps: 49, moving_average_steps: 49.75
episode: 700, total_rewards: 0.9729, total_current_steps: 62, moving_average_steps: 50.0
episode: 800, total_rewards: 0.9787, total_current_steps: 49, moving_average_steps: 56.08
episode: 900, total_rewards: 0.9876, total_current_steps: 29, moving_average_steps: 44.79
episode: 1000, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 30.26
episode: 1100, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 29.51
episode: 1200, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.42
episode: 1300, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.32
episode: 1400, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.16
episode: 1500, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.05
episode: 1600, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.02
episode: 1700, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.0
episode: 1800, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.31
episode: 1900, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.0
episode: 2000, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.0
```

```
# Determenistic Policy - Best Policy in 28 steps
```

It’s important to mention that with a stochastic transition model, we are not necessarily getting to the optimal path. One way to think of it is that in every 10 steps the agent is not listening to the policy advice once and makes a mistake that can cost 2 or more steps (back and forth) so we have on average 3 more steps and maybe more as seen above.

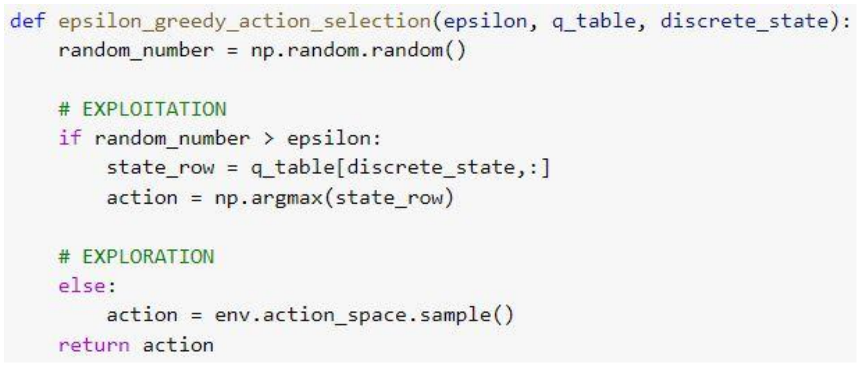
Q-Learning on 15x15 board

Now, we were trying to solve the discrete 15X15 board problem, using the Q-learning model.

Even though the model is different, the data structure is the same as before, this means we have 4 actions for columns, and 225 possible states (player location on the 15 by 15 grid). So our table will look the same as before.

Next, we again took care of the Exploration vs. Exploitation Dilemma. If we simply always select the argmax() q-table value during training, we'll most likely get stuck in an exploitation loop, so we'll use a random value to randomly select an action from time to time, helping the model explore , rather than exploit. The epsilon will help us balance exploration (random choice) versus exploitation (always picking what works for that Q(s,a)).

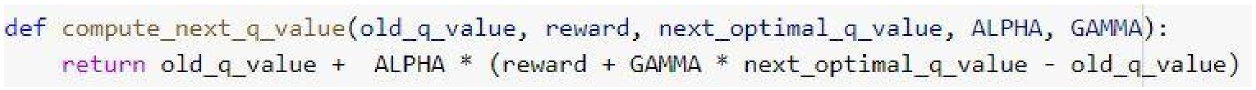
We implement the epsilon greedy action selection in the following function:



```
def epsilon_greedy_action_selection(epsilon, q_table, discrete_state):
    random_number = np.random.random()

    # EXPLOITATION
    if random_number > epsilon:
        state_row = q_table[discrete_state,:]
        action = np.argmax(state_row)

    # EXPLORATION
    else:
        action = env.action_space.sample()
    return action
```

As training continues, we need to balance exploitation versus exploration, we want to make sure our agent doesn't get trapped in a cycle going from one square to another, back and forth. We also don't want our agent permanently choosing random values. For that we again used the reduce_epsilon function.

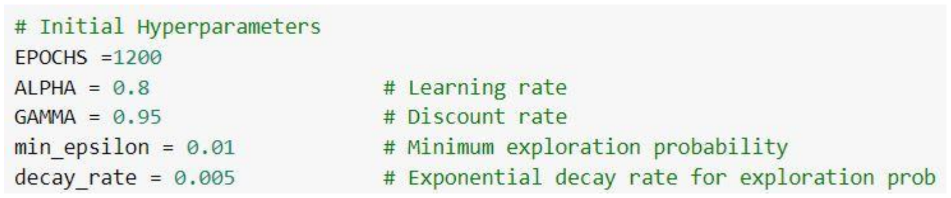
Here we have our main Q-Learning update equation, that function takes in the old q-value, the next optimal q value, the current reward, along with the learning and the discount rates and then updates the next q value accordingly:



```
def compute_next_q_value(old_q_value, reward, next_optimal_q_value, ALPHA, GAMMA):
    return old_q_value +  ALPHA * (reward + GAMMA * next_optimal_q_value - old_q_value)
```

Model Tuning and parameters selection

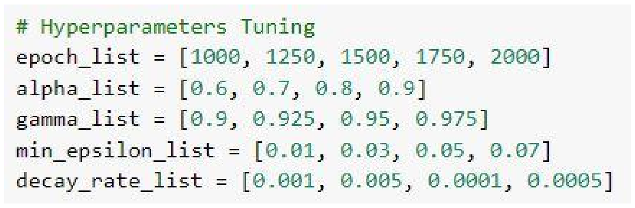
Next, we built the main training function for a given number of episodes, and set our initial hyperparameters as follows:



```
# Initial Hyperparameters
EPOCHS =1200
ALPHA = 0.8                   # Learning rate
GAMMA = 0.95                  # Discount rate
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.005            # Exponential decay rate for exploration prob
```

As in the monte-carlo model we started with some initial parameters to see the model is converging and then continued to experiment with hyperparameter tuning

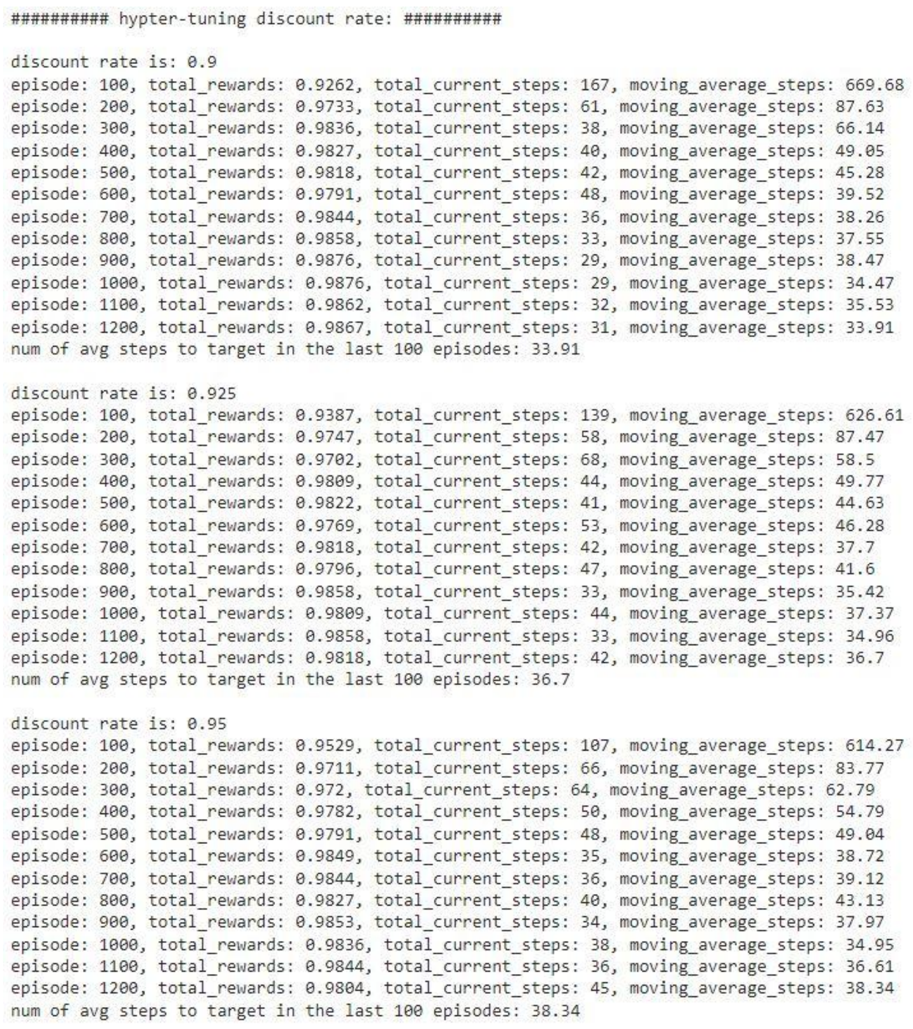
we consider the following possibilities:



```
# Hyperparameters Tuning
epoch_list = [1000, 1250, 1500, 1750, 2000]
alpha_list = [0.6, 0.7, 0.8, 0.9]
gamma_list = [0.9, 0.925, 0.95, 0.975]
min_epsilon_list = [0.01, 0.03, 0.05, 0.07]
decay_rate_list = [0.001, 0.005, 0.0001, 0.0005]
```

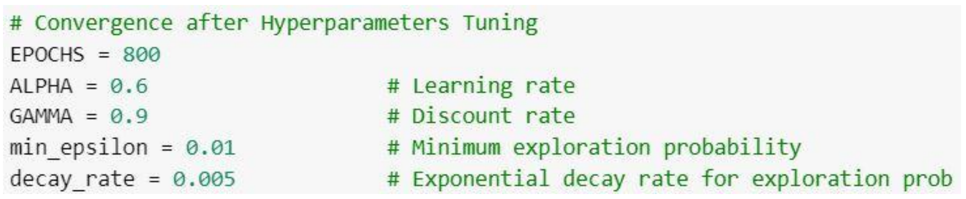
For each of these hyperparameters, we run a separate experiment, where we track the moving average number of steps for the past 100 episodes. In the next figure we attached an example for the hyperparameter tuning we made for the epsilon decay rate.

As we can see from the last log line in each experiment, there is a significant difference depending on the value of the decay rate. We choose the values that minimize the number of average steps to the target in the last 100 episodes.



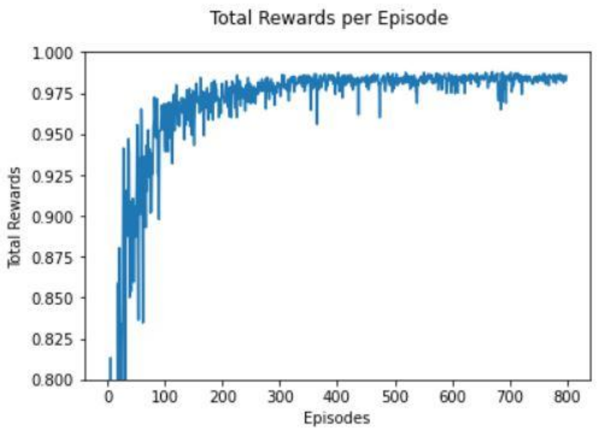
```
########## hypter-tuning discount rate: ##########

discount rate is: 0.9
episode: 100, total_rewards: 0.9262, total_current_steps: 167, moving_average_steps: 669.68
episode: 200, total_rewards: 0.9733, total_current_steps: 61, moving_average_steps: 87.63
episode: 300, total_rewards: 0.9836, total_current_steps: 38, moving_average_steps: 66.14
episode: 400, total_rewards: 0.9827, total_current_steps: 40, moving_average_steps: 49.05
episode: 500, total_rewards: 0.9818, total_current_steps: 42, moving_average_steps: 45.28
episode: 600, total_rewards: 0.9791, total_current_steps: 48, moving_average_steps: 39.52
episode: 700, total_rewards: 0.9844, total_current_steps: 36, moving_average_steps: 38.26
episode: 800, total_rewards: 0.9858, total_current_steps: 33, moving_average_steps: 37.55
episode: 900, total_rewards: 0.9876, total_current_steps: 29, moving_average_steps: 38.47
episode: 1000, total_rewards: 0.9876, total_current_steps: 29, moving_average_steps: 34.47
episode: 1100, total_rewards: 0.9862, total_current_steps: 32, moving_average_steps: 35.53
episode: 1200, total_rewards: 0.9867, total_current_steps: 31, moving_average_steps: 33.91
num of avg steps to target in the last 100 episodes: 33.91

discount rate is: 0.925
episode: 100, total_rewards: 0.9387, total_current_steps: 139, moving_average_steps: 626.61
episode: 200, total_rewards: 0.9747, total_current_steps: 58, moving_average_steps: 87.47
episode: 300, total_rewards: 0.9702, total_current_steps: 68, moving_average_steps: 58.5
episode: 400, total_rewards: 0.9809, total_current_steps: 44, moving_average_steps: 49.77
episode: 500, total_rewards: 0.9822, total_current_steps: 41, moving_average_steps: 44.63
episode: 600, total_rewards: 0.9769, total_current_steps: 53, moving_average_steps: 46.28
episode: 700, total_rewards: 0.9818, total_current_steps: 42, moving_average_steps: 37.7
episode: 800, total_rewards: 0.9796, total_current_steps: 47, moving_average_steps: 41.6
episode: 900, total_rewards: 0.9858, total_current_steps: 33, moving_average_steps: 35.42
episode: 1000, total_rewards: 0.9809, total_current_steps: 44, moving_average_steps: 37.37
episode: 1100, total_rewards: 0.9858, total_current_steps: 33, moving_average_steps: 34.96
episode: 1200, total_rewards: 0.9818, total_current_steps: 42, moving_average_steps: 36.7
num of avg steps to target in the last 100 episodes: 36.7

discount rate is: 0.95
episode: 100, total_rewards: 0.9529, total_current_steps: 107, moving_average_steps: 614.27
episode: 200, total_rewards: 0.9711, total_current_steps: 66, moving_average_steps: 83.77
episode: 300, total_rewards: 0.972, total_current_steps: 64, moving_average_steps: 62.79
episode: 400, total_rewards: 0.9782, total_current_steps: 50, moving_average_steps: 54.79
episode: 500, total_rewards: 0.9791, total_current_steps: 48, moving_average_steps: 49.04
episode: 600, total_rewards: 0.9849, total_current_steps: 35, moving_average_steps: 38.72
episode: 700, total_rewards: 0.9844, total_current_steps: 36, moving_average_steps: 39.12
episode: 800, total_rewards: 0.9827, total_current_steps: 40, moving_average_steps: 43.13
episode: 900, total_rewards: 0.9853, total_current_steps: 34, moving_average_steps: 37.97
episode: 1000, total_rewards: 0.9836, total_current_steps: 38, moving_average_steps: 34.95
episode: 1100, total_rewards: 0.9844, total_current_steps: 36, moving_average_steps: 36.61
episode: 1200, total_rewards: 0.9804, total_current_steps: 45, moving_average_steps: 38.34
num of avg steps to target in the last 100 episodes: 38.34
```

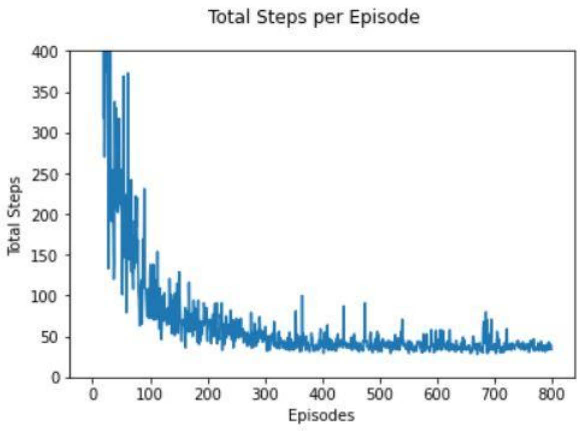
Eventually, when the experiment test ends, we tuned our hyperparameters accordingly, resulted with the following values:



```
# Convergence after Hyperparameters Tuning
EPOCHS = 800
ALPHA = 0.6                   # Learning rate
GAMMA = 0.9                   # Discount rate
min_epsilon = 0.01            # Minimum exploration probability
decay_rate = 0.005            # Exponential decay rate for exploration prob
```

Now, we were ready to train our model. The tuning improved our results. As we can see from the following figures, we manage to converge. As expected, the total rewards gradually increase over the episodes, and the total steps are gradually decreasing over the episodes.





As in the MC model we recorded the middle of training and the last epoch movies along with non-stochastic environment video to show the optimal solution (28 steps)

Here is the final result for the deterministic experiment:

```
episode: 100, total_rewards: 0.9591, total_current_steps: 93, moving_average_steps: 663.4
episode: 200, total_rewards: 0.9707, total_current_steps: 67, moving_average_steps: 59.82
episode: 300, total_rewards: 0.9818, total_current_steps: 42, moving_average_steps: 41.14
episode: 400, total_rewards: 0.9862, total_current_steps: 32, moving_average_steps: 35.18
episode: 500, total_rewards: 0.9867, total_current_steps: 31, moving_average_steps: 32.22
episode: 600, total_rewards: 0.9876, total_current_steps: 29, moving_average_steps: 30.3
episode: 700, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 29.32
episode: 800, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.64
episode: 900, total_rewards: 0.9876, total_current_steps: 29, moving_average_steps: 28.48
episode: 1000, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.37
episode: 1100, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.15
episode: 1200, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.08
episode: 1300, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.04
episode: 1400, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.03
episode: 1500, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.02
episode: 1600, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.02
episode: 1700, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.02
episode: 1800, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.0
episode: 1900, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.0
episode: 2000, total_rewards: 0.988, total_current_steps: 28, moving_average_steps: 28.0
```

```
# Determenistic Policy - Best Policy in 28 steps
```

MC vs Q-learning comparison

The Monte-Carlo algorithm has a strategy to estimate the optimal policy for an unknown model. However, a disadvantage is that the strategy can only be updated after the whole episode. In other words, the Monte Carlo method does not make full use of the MDP learning task structure. The Q-learning on the other hand is making full use of the MDP structure. It converges faster and is more stable than the Monte-Carlo.

30x30 board

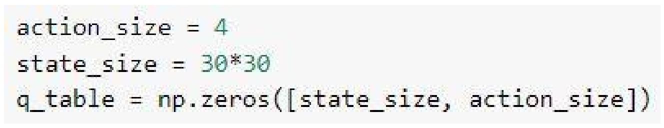
After solving the 15X15 board problem, both on the Monte Carlo and the Q-learning Models and examining the model's results, we've decided to solve the discrete 30X30 board problem with the Q-learning Model, which in our opinion had a smoother convergence.

As stated above, our table is essentially a mapping of all possible states,action pairs and the expected reward for taking an action at a particular state that we will keep updating. $(Q(s_t,a_t))$

For our advanced discrete 30X30 board problem, this means we have 4 actions for columns, and 900 possible states (player location on the 30 by 30 grid). So our table will look like:

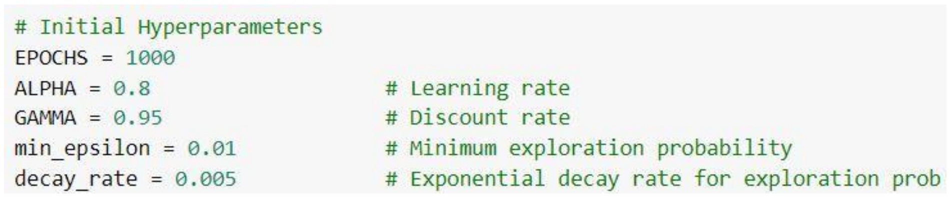
	'W' - LEFT	**'S' - DOWN**	**'E' - RIGHT**	**'N' - UP**
State 0	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
State 1	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)
State ...	...	...	...	...
State 899	Q(s,a)	Q(s,a)	Q(s,a)	Q(s,a)

Technically, we implemented this structure with a np array as following



```
action_size = 4
state_size = 30*30
q_table = np.zeros([state_size, action_size])
```

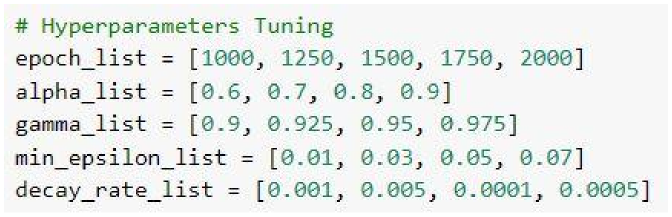
As before, the Q-Learning update functions require hyperparameters. We initialize those has following:



```
# Initial Hyperparameters
EPOCHS = 1000
ALPHA = 0.8                    # Learning rate
GAMMA = 0.95                   # Discount rate
min_epsilon = 0.01             # Minimum exploration probability
decay_rate = 0.005             # Exponential decay rate for exploration prob
```

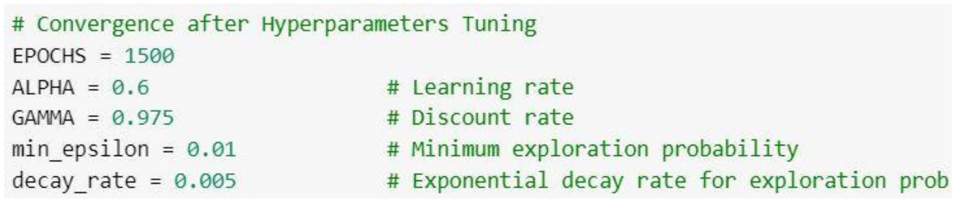
Now, we were ready to run our first initial experiment. Similarly to the 15X15 board problem, this gave us a quick overview before we continue to do more experiments with different hyper-parameters. Fourtantly, the experiment went well, and the model managed to converge.

Next, we tuned our hyperparameters, using the following possibilities:



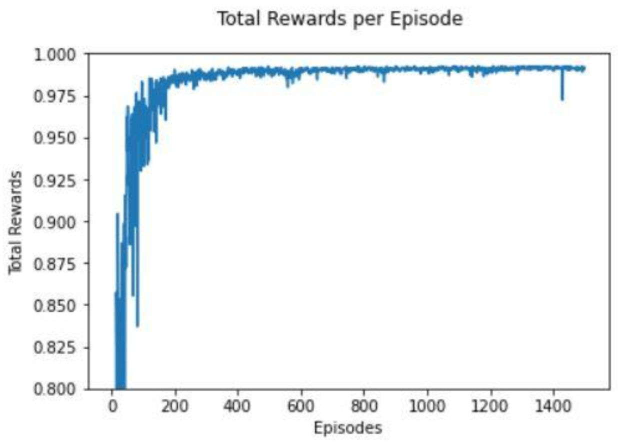
```
# Hyperparameters Tuning
epoch_list = [1000, 1250, 1500, 1750, 2000]
alpha_list = [0.6, 0.7, 0.8, 0.9]
gamma_list = [0.9, 0.925, 0.95, 0.975]
min_epsilon_list = [0.01, 0.03, 0.05, 0.07]
decay_rate_list = [0.001, 0.005, 0.0001, 0.0005]
```

Eventually, when the experiment test ends, we tuned our hyperparameters accordingly, resulted with the following values:

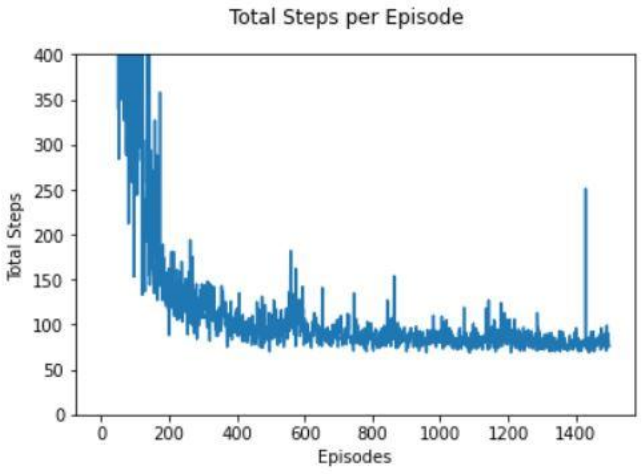


```
# Convergence after Hyperparameters Tuning
EPOCHS = 1500
ALPHA = 0.6                    # Learning rate
GAMMA = 0.975                  # Discount rate
min_epsilon = 0.01             # Minimum exploration probability
decay_rate = 0.005             # Exponential decay rate for exploration prob
```

Now, we were ready to train our model. The tuning improved our results. As we can see from the following figures, we manage to converge. As expected, the total rewards gradually increase over the episodes, and the total steps are gradually decreasing over the episodes



.



As before, we thought it would be nice to record a video with no stochasticity. Therefore, we set the ‘is_stochastic’ flag to False. Now we were able to actually see the optimal route, and examine the required minimal number of steps for solving the maze.

Here is the final result for the deterministic experiment:

```
episode: 100, total_rewards: 0.9683, total_current_steps: 286, moving_average_steps: 2229.72
episode: 200, total_rewards: 0.9846, total_current_steps: 140, moving_average_steps: 240.18
episode: 300, total_rewards: 0.9919, total_current_steps: 74, moving_average_steps: 104.72
episode: 400, total_rewards: 0.9917, total_current_steps: 76, moving_average_steps: 82.91
episode: 500, total_rewards: 0.9923, total_current_steps: 70, moving_average_steps: 74.99
episode: 600, total_rewards: 0.9922, total_current_steps: 71, moving_average_steps: 71.38
episode: 700, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 69.02
episode: 800, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 67.97
episode: 900, total_rewards: 0.9927, total_current_steps: 67, moving_average_steps: 67.15
episode: 1000, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.6
episode: 1100, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.64
episode: 1200, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.26
episode: 1300, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.09
episode: 1400, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.15
episode: 1500, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.03
episode: 1600, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.01
episode: 1700, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.04
episode: 1800, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.0
episode: 1900, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.02
episode: 2000, total_rewards: 0.9928, total_current_steps: 66, moving_average_steps: 66.0
```

```
# Determenistic Policy - Best Policy in 66 steps
```

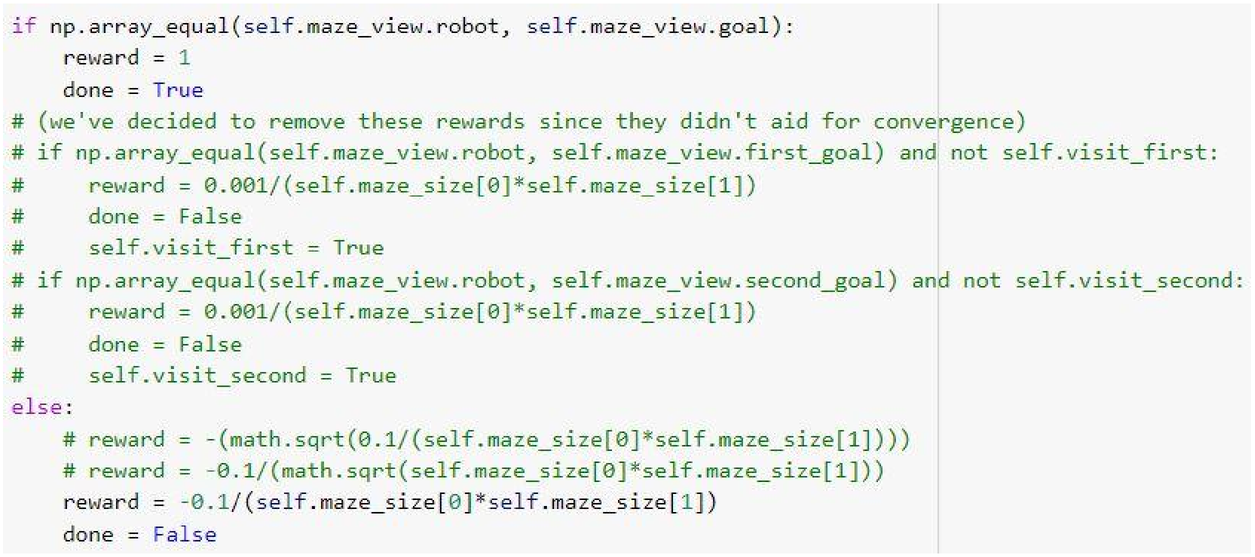
Experimenting with adding a reward cells to help convergence

Finally, we've tried to add two partial goals as suggested in the instructions. Since the optimal route took 66 steps, we thought to put the first partial goal after 22 steps, and the second partial goal after 44 steps in that route. In doing so, our agent will receive a positive reward in every ⅓ of his way to the final goal. Thus, we set the partial goals to the following squares:

```
self.__first_goal = np.array((1, 17))
self.__second_goal = np.array((12, 28))
```

We've made a few experiments such as: trying different reward values, giving a positive reward only for the first visit over the partial goal coordinates, giving a positive reward over these squares along with a negative reward for the extra step, etc. In addition, we've made some experiments regarding the negative rewards value. However, eventually, since we did not see any improvement, we decided to stay with the original reward system.

The experimentation regarding the reward system is attached:



```
if np.array_equal(self.maze_view.robot, self.maze_view.goal):
    reward = 1
    done = True
# (we've decided to remove these rewards since they didn't aid for convergence)
# if np.array_equal(self.maze_view.robot, self.maze_view.first_goal) and not self.visit_first:
#     reward = 0.001/(self.maze_size[0]*self.maze_size[1])
#     done = False
#     self.visit_first = True
# if np.array_equal(self.maze_view.robot, self.maze_view.second_goal) and not self.visit_second:
#     reward = 0.001/(self.maze_size[0]*self.maze_size[1])
#     done = False
#     self.visit_second = True
else:
    # reward = -(math.sqrt(0.1/(self.maze_size[0]*self.maze_size[1])))
    # reward = -0.1/(math.sqrt(self.maze_size[0]*self.maze_size[1]))
    reward = -0.1/(self.maze_size[0]*self.maze_size[1])
    done = False
```

References

https://www.analyticsvidhya.com/blog/2018/11/reinforcement-learning-introduction-monte-carlo-learning-openai-gym/

https://www.youtube.com/watch?v=C9JGEfFF_EQ&list=PL8Ndnh4x737oIVNEQw2KM7FPmGDg2odKK

https://www.udemy.com/course/practical-ai-with-python-and-reinforcement-learning/

https://www.udemy.com/course/artificial-intelligence-reinforcement-learning-in-python/

https://www.youtube.com/watch?v=IXuHxkpO5E8