Solving a 2D Maze with a Deep Q-Network Model using OpenAI's Gym and ENSTA ParisTech's Stable Baselines3

Jake Miu, Alex Flitter

Introduction and Motivation

Reinforcement learning describes a developmental process for an agent in an environment where an agent is rewarded or punished for its actions in the environment. These agents can be recognized to be living entities with a simple decision making structure. The entities explore their environment, learning which actions are rewarded or punished and where said actions occur. Take a toddler, for example, exploring the house of his parents as he learns to walk around. The office space in the house is off limits and the toddler is punished for attempting to enter the room. While the rationale behind why the toddler cannot enter the room is unclear, what the toddler learns is the action of moving from outside to inside that room will receive him a punishment. Varying rooms accumulate varying punishments and rewards for the toddler and its decision making for the purpose of exploration is affected accordingly. This relates to the agent we will develop as our agent will not understand why the punishments are accrued but rather where and how severe the punishments will be. More importantly, the agent will have an objective to maximize the rewards gained in its environment, effectively avoiding all learned areas in the environment where severe punishments are received. The type of reinforcement learning we implement is Q-learning via the bellman equation in a deep q-network (DQN).

In order to construct our environment we utilized the tools published by Elon Musk's AI research and deployment company, namely, OpenAI. This company has been around for approximately seven years and has been on the forefront of representing reinforcement learning scenarios in Python. We specifically used their Gym Python library to implement our DQN and write our environment such that the model and environment are compatible. Note that Gym from OpenAI is an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments. Since Gym from OpenAI is open source, all their code is published on GitHub, and there are some notable youtubers that walk through some of OpenAI's projects that utilize the Gym library. One such youtuber, Nicholas Renotte, posted a video detailing how to create a custom reinforcement learning environment for personal projects such that the environments are compatible with learning models from Stable Baselines3. At ENSTA ParisTech, Stable Baselines3 was developed as a set of improved implementations of reinforcement learning algorithms based on OpenAI's baseline package. Hence, we intended to use OpenAI's Gym library and ENSTA ParisTech's Stable Baselines3's learning models to simulate a DQN agent learning in a custom environment, specifically to solve a 2D-Maze.

Problem Definition

Our goal was to take any square maze with a fixed end and have an agent learn upon the maze such that it could identify an optimal path to the fixed end from any starting position. We decided to use a deep q-network(DQN) because the action space for solving a maze is fixed and discrete. Other kinds of networks such as proximal policy optimization(PPO) seemed overkill for solving a maze since they are generally used for non-discrete action spaces. We also wanted to render our maze so visualization of how the agent behaved during a single episode would be more clear.

Proposed Method

Our first step was to create a custom environment for our agent to learn upon. We selected Stable Baselines3 to develop our DQN model as it was a reinforcement learning representation we were already familiar with. In order to create an environment to be compatible with Stable Baselines3, we were required to implement four member functions for our Python class, namely, init, step, render and reset.

When thinking about how our environment would be able to visualize a maze we realized that an "N by N" maze holds more than "N by N" bits of information. This is because some of the walls are horizontal while others are vertical, so identifying a specific wall in a data structure that can only hold "N by N" bits of information is problematic. What we need is an "N+1 by 2N+1" array to represent our maze. The "2N+1" would start at the left-most wall and then alternate looking at vertical walls and horizontal walls. For example, the 0th index represents the first vertical wall, the 1st index the first horizontal wall, and the 2nd index the

second vertical wall. The "N+1" holds the vertical position of the wall, counting from the bottom to the top of the maze. Note that an "N by N" maze has "N+1" horizontal walls in each column but only "N" vertical walls in each column; thus the top position of the vertical wall columns are filled with an inconsequential value.

init is the constructor for the environment defining most of the important variables that the other three member functions will reference. We decided that the maze would only have four discrete actions, specifically, moving one space North, South, East, or West. The environment would also have to keep track of an agent's current state and past state in the environment so we defined these as integer tuples. As our maze was two-dimensional, our use of tuples enabled us to have our first element in the tuple referred to the x-axis and the second element referred to the y-axis. Further, we created a variable called steps which kept track of how many actions had been taken in that episode and max-steps which was the step count where the code would terminate.

The step function considers an agent's current position and cross-references the wall array in the environment to see if the provided action is valid. If the agent attempts to move into a wall then the agent's current state will not change, functioning as a disincentive since our gamma value will make future rewards smaller. To determine if a move is valid we first identify which cordial direction the agent has elected to move towards. Then using the current position and selected action we reference the correct location in the walls array to resolve whether a wall exists in the agent's path or not. If a wall prevents an agent from moving then we do not update the agent's current position. If there is no wall, however, then the agent's current position would be updated accordingly and the agent accrues a proportionally small negative reward. For the purpose of accelerating the learning process for an agent to not to move into walls, we've specified the negative reward of an action into a wall to be ten times the magnitude compared to an action into free space. Next, the function increments the number of steps the agent has taken in the environment by one and compares the value for the environment's current steps versus the max steps. If the two variables are equal in value then the episode terminates and the current reward is returned.

For our render function, we decided to integrate Pygame. Pygame is a python implementation that can create windows with mutable frames and can update extremely quickly. Our render function code interprets and displays our maze visually in a pygame window. It then draws a red circle at the agent's current position, a blue circle at the agent's past position, and a green circle at the fixed end. This trail of blue dots is important for our visualization because Pygame occasionally struggles to keep up with Stable Baselines3. Hence the blue path is where the agent maneuvered in-between frames.

The reset function is called when an episode is finished and returns an environment to its initial state by assigning its member variables to their original values. Meaning, we set the current and past state to the starting position, we delete all the drawings on the rendered window, and set the steps counter back to zero. This concludes our custom environment implementation.

Once the environment implementation was complete we were able to test its functionality. We opted to mimic an agent by passing through a six-by-six maze with randomly selected actions instead of predicted actions. Meaning the paths chosen would vary on each episode with no relation to the previous paths, and the exploration around the maze could identify edge cases where actions break our execution. The random actions generally ranged from 300 to 4000 steps to solve the maze, clearly denoting the environment was functioning as intended across all cells of the maze. From this point onward we continued to work with a six-by-six maze. We felt that this size was large enough to demonstrate whether our DQN would behave as intended in our custom environment or not, while also not being too large as to make the model's training time incredibly lengthy.

Our next step was to instantiate the DQN model from Stable Baselines3 and have it learn in our environment for a specific amount of time. The amount of time the model needed to learn the maze would depend on the complexity and size of the maze. Once the model had learned over our specified training time we saved the DQN's decision table to avoid retraining in future iterations.

We then tested the model in a singular episode of prediction a path through the maze for performance purposes. The model used its predict function to sort through its DQN decision table to select an optimal action based on its current state that maximizes its future reward gains. After each action was taken by the model the render function would display its path through the environment for us to measure its performance. Once we were confident that our model had sufficiently learned upon the maze, we used Stable Baselines3's evaluate policy function to test the average score of our model's performance in the maze over a large number of episodes.

Intuition

One of the reasons we had confidence that we could develop a working project is because Stable Baselines3 is a massive resource for reinforcement learning projects with a plethora of documentation. Additionally, Nicholas Renotte, a youtuber who presents OpenAI's projects and their implementation, posted a sufficiently in-depth video about how to create your own environment to be compatible with Sta-

ble Baselines3's models. His guidance aided us in breaking down our project into manageable parts and learning the syntax of the library. By watching Nicholas work on projects with equal complexity we were given the confidence that our project could be developed using Stable Baselines3 and a custom reinforcement learning environment.

At the start we had no intuition as to how to implement our environment's rendering, but after searching through OpenAI's posted GitHub projects with working rendering functions, it seemed that Pygame was the clear choice. By using online documentation and a video from Tech With Tim we started to familiarize ourselves with the pros and cons of working with Pygame. While Pygame was capable of rendering our maze, we had to be particularly careful when manually exiting out of a window created by Pygame. This was a challenge as manually closing the window would crash our kernels, but after getting more familiar with Pygame's syntax we figured out how to solve this problem and implemented its solution accordingly.

Once we had confidence that our project could be developed by using Stable Baselines3 and Pygame we built the environment and made all three components function in tandem. We mimicked the general flow of control of the GitHub projects where a custom environment made by us and a model from Stable Baselines3 were instantiated then rendered by Pygame. It is worth noting that we had confidence that we could figure out the logic aspect of the maze in a reasonable amount of time.

We believed that this project was doable with the given amount of time, our previous experience, our group work ethic, and our collective ambition. Although there were unforeseeable challenges, the main reasons why we thought we would prevail (Stable Baselines, Pygame, Github examples) kept our hopes up for success.

Experiments

Once our project was fully functional, we developed two key questions to test our DQN model. The first question was to identify the differences in learning time for the DQN model based on the complexity of its starting position in the maze. We defined the complexity of the starting position to be determined by the amount of walls surrounding the starting cell; meaning, zero walls around the starting cell was the least complex scenario and three walls surrounding the cell was the most complex scenario.

The second question was to measure the performance of DQN models with varying amounts of training. We trained five DQN models with timestep values increasing by 100k starting at 100k (range is 100k to 500k timesteps).

Table 1.

Model Number	Training Timesteps	Actions Taken	Into-Wall Actions
DQN 1	100k	>1000	>1000
DQN 2	200k	206	175
DQN 3	300k	40	21
DQN 4	400k	17	1
DQN 5	500k	16	0

Details and Observations

For the first question, we tested two scenarios of the DQN model learning upon its environment. Specifically, we modified the DQN model's starting position in the maze to represent three complexity levels. The three scenarios include the starting position being the least complex with zero walls surrounding the starting cell, being somewhat complex with one wall surrounding the starting cell, and being the most complex with three walls surrounding the starting cell. The rationale behind this complexity exists in the DQN's ability to continue to run into walls after learning an action into a wall will not only not change its state, but will still accrue a negative reward.

In the least complex scenario, the agent does not struggle to learn the correct path from its starting state as there are no walls to run into. The agent's learning time in this scenario is optimized. In the somewhat complex scenario, the agent takes longer to learn upon its environment as the agent decides to take multiple actions into the singular surrounding wall. The agent, however, quickly learns that the three alternate options of moving into cells without a wall in their path provide the best opportunity to maximize the agent's reward. In the most complex scenario, the agent's learning time is extraordinarily longer than the past two scenarios as the agent visually runs into the three surrounding walls over and over again, even with the negative reward being decupled for actions taken into walls. See the learning time represented via Tensorboard in ep-len-mean (mean episode length) for the least and most complex scenarios in Figure 1-1 and Figure 2-1 respectively below.

For the second question, we tested five DQN models with training timesteps on the environment ranging from 100k to 500k in value. In these scenarios, the starting and ending cells of the maze are fixed. The training timesteps represent the DQN model exploring and exploiting its environment over time. As the first question reveals an agent struggles to learn optimal pathing choices in cells with walls surrounding them, we test the agent's ability to learn and explore across the entire maze in a constrained amount of time. See Table 1 above for the results of these scenarios.

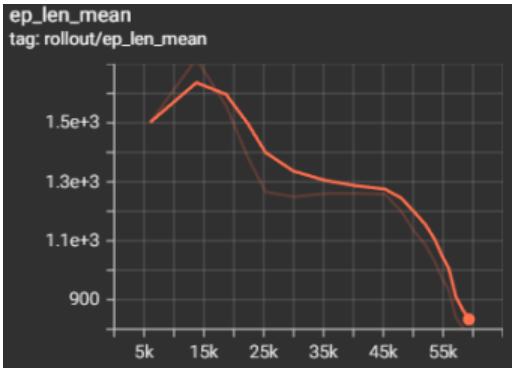

Figure 1. Figure 1-1

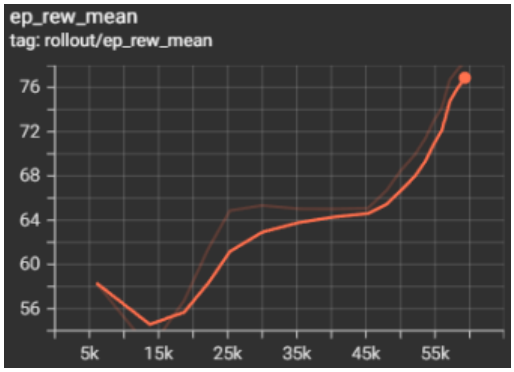

Figure 2. Figure 1-2

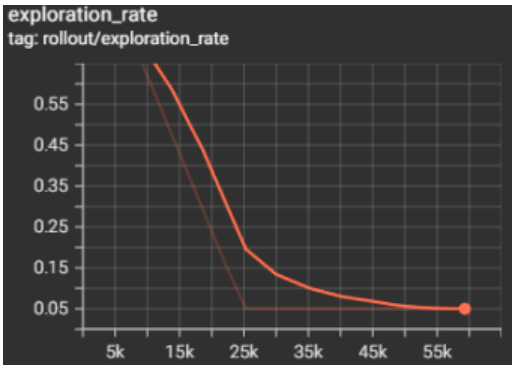

Figure 3. Figure 1-3

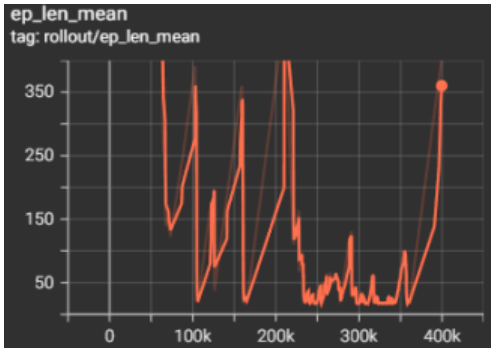

Figure 4. Figure 2-1

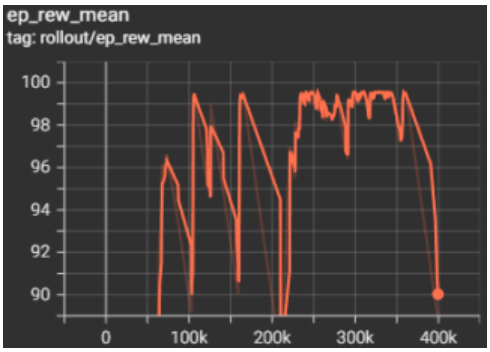

Figure 5. Figure 2-2

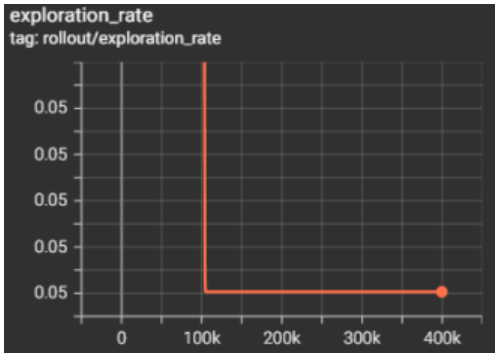

Figure 6. Figure 2-3

Conclusions

Recall our goal was to simulate a StableBaseline3's DQN model within a custom Gym environment and render its behavior using Pygame. Following the proposed method above, we were successfully able to accomplish our goal. We learned that the way we implemented our wall structure, even with decupling the negative accrued reward for moving into walls, proved to present a problem in the DQN model's performance with varying training times (Table 1). We also learned that the initial complexity of the starting cell for the DQN model drastically impacted the learning time and produced a current phenomenon in the reinforcement learning research world; to specify, we were able to simulate a model with staggering differences in performance across its learning process (Figure 2-1, 2-2, 2-3). However, with a starting cell of low complexity, and given sufficient training time, we were able to generate a DQN model that could solve our maze without running into walls (Figure 1-1, 1-2, 1-3). To optimize our implementation in the future we could introduce callbacks for our training procedure to stop training early once the mean reward per episode reached the upper threshold of solving the maze without running into walls. In summary, we demonstrated the usage of OpenAI's Gym library and ENSTA ParisTech's Stable Baselines3's DQN learning model to simulate an agent learning in a custom 2D-Maze.