Remy the Maze Solver

Aidan Fox-Tierney, Angela Ni, Ching-Chia Lee, Chris Cruz, Matthew Martel, Sameerah Helal, Vu Pham, Xian Shi, Yibo Yan, Yudong Wang, Yunbo Lu

Introduction

In the age of COVID-19, the public health sphere experiences renewed hope with the introduction of a vaccine in its early stage of testing. With a fast tracked timeline, public health officials face a new critical problem: how to optimize vaccine distribution while the vaccine requires sub-zero temperatures and other crucial, specific conditions to ensure its effectiveness. In the upcoming months, vaccine distributors will need to address the question of accessibility for even the most remote people in the world. With the rise of technology in the daily lives of people anywhere and everywhere, path-finding algorithms have become a necessity in developing the tools and services that solve everyday problems. These path-finding algorithms, however, cannot remain fixed. As the world grows and changes, so do these path-finders – and thus, machine learning models can assist these path-finders in becoming smart-learners as well. Whether the objective is delivering a COVID-19 vaccine to a densely forested area, preparing a detailed set of instructions to guide a driver from point A to point B safely and efficiently, or designing an optimal path through a maze, path-finders with high performance, accuracy, and optimality have a wide-range of applications to the real world.

In this project, we utilize machine learning to address an increasingly popular problem in Computer Science: how to find an optimal solution to a maze. Over the course of the quarter, our team of Computer Science, Mathematics and Statistics, and various Engineering students have delegated tasks to create a reinforcement learning model that trains and tests on iterations of generated mazes. In the process, we have scripted a maze-generator that can generate thousands of valid mazes of different sizes in a matter of seconds, referenced Q-learning literature to aid in the process of creating our model, and developed a reinforcement model that solves the problem at hand.

Within this report, we have outlined the frameworks that this project utilizes in order to describe how the model was constructed and how it operates. The literature review highlights how our team integrated pre-existing programs and concepts to create a new model with differentiated features. The proposed solution and experimental results illustrate how we implemented the solution, the reasoning behind specific machine learning decisions, and the consequences of those decisions.

Literature Review

The Q-Maze Learning Algorithm[1] was the inspiration for our project idea. It releases a mouse agent into a maze matrix populated with walls, which the agent is trained to navigate and solve; i.e. reach a target gridpoint. It then sets the agent on new mazes that it must solve using its training.

The learner's actions in Q-Maze's model are defined by a Markov decision process, a discrete-time stochastic control process[7] where each state possesses the *Markov property* that the next state depends only on the current one. The learner, from interacting with the environment, may take on one of a finite, discrete set of states; depending on which they may perform one of a finite, discrete set of actions to take them to a different state.[8]

Reinforcement learning is a type of machine learning algorithm in which a learner takes actions and is informed, or taught, the appropriateness of each action[3]. The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. The Q-Maze model specifically uses a value-based subset of reinforcement learning called *Q-Learning*, in which each state action pair is assigned a value, creating a stochastic-type matrix with dimensions [number of possible states] $\times$ [number of possible actions][4]. Q-Learning iteratively updates the value of a best-utility or best-quality Q-function (the aforementioned matrix) using the *Bellman equation* until it reaches the optimum[5]. The Bellman equation is defined as

$$Q(s,a) = R(s,a) + \max_{i \in \{0,\ldots,n-1\}} Q(s', a_i),$$

where n is the number of possible actions a_i, $R(s,a)$ is the immediate reward, and s' is the next state. While training, the agent is supplied with reinforcement and next-state instructions from respectively named functions to the end of helping it solve the maze.

The main objective of Q-training is to develop a policy π for navigating the maze successfully, the best policy being associated with the optimal value of the best-quality function. This function is found using the the Bellman equation by $\pi(s) = \text{argmax}Q(s, a_i)$ for $i \in \{0, \ldots, n-1\}$, where s is the current state and each a_i is an action. This policy gives the agent a rule for what action to take in every possible state. In training, the agent may either exploit the current π to decide its action, or, more rarely, take a random action to gain experience and explore.

As in the Q-Maze project, our project also includes an agent solving a maze of a similar format. However, our agent trains on more than one maze. Our interface also differs because we allow a user to select mazes, which they may watch the agent solve. Both projects use Python's Tensorflow library in the building of their models.

The Human-level control through deep reinforcement learning paper[2] presents an implementation of Q-learning that trains an agent to compete in "the challenging domain of classic Atari 2600 games". This model uses *experience replay*, which is a scheme for sampling non-uniformly from the *replay buffer*, the store of the learner's past experiences[6]; and has been shown to improve sample efficiency and stability by storing a fixed number of the most recently collected transitions for training. Including experience replay was shown in the paper to dramatically improve the performance or likelihood of winning of the learner in many of the games.

Based on the results of this paper we implemented experience replay in our own model to improve the performance of our agent in solving the mazes. The Q-Maze model does not include this feature. As well, our model trains a learner on our own mazes as opposed to different video games as in the paper's model.

Dataset Description

Overview

The dataset of this project was a set of mazes that the machine learning model trained and tested on. While there was no readily available public datasets of mazes, the process of generating mazes was a rather straightforward one. The maze-generating script evolved from a basic implementation that can create a simple CSV maze in 0.1 seconds to a scalable maze-generator that can produce thousands of mazes in 3.5 seconds. Each CSV file consists of 1's and 0's indicating the existence of a maze wall at specified coordinates.

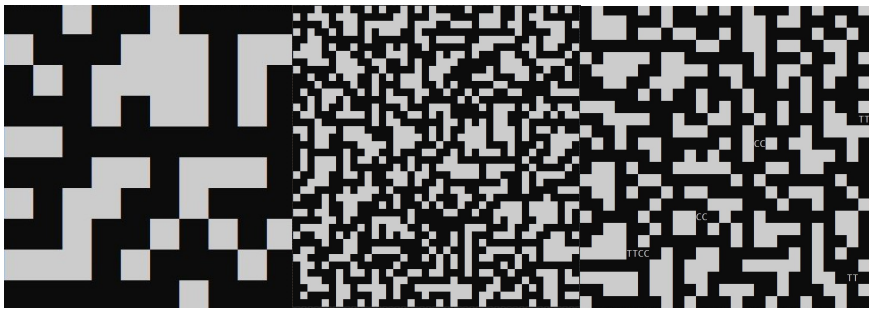

Figure 1: Basic iteration biased to cluster; scalable iteration; refined iteration with random cheese, traps.

Maze Validity and Duplicate Detection

It is important to note that the maze-generator did not initially produce consecutive horizontal walls to avoid wall offs and ensure mazes had valid paths. This feature was later modified to fill unconnected cells while maintaining the validity of the maze.

The maze-generating script also prevents the creation of duplicate mazes by hashing the outputs and running an additional check in the hashmap to confirm that the generated maze does not already exist.

Proposed Solution and Experimental Results

Solution Overview

The proposed solution involves a model similar to the Q-Maze algorithm. With some adjustments and the inclusion of some aspects of the Human-level control through deep reinforcement learning model, our team hoped to train an agent to quickly and successfully solve a variety of mazes. The outline of our solution was as follows: we would generate mazes encoded in .csv files, train our agent on them using Q-learning, then release the agent into previously unseen mazes to see if it can navigate the mazes from start to finish in a time-efficient manner.

Reward System

Reinforcement learning algorithms are largely defined by their reward systems, so our solution implements several reward functions in the training process to push the mouse agent towards finding the path to solve the maze as fast as possible. Since our model utilizes the Bellman equation as its Q-function, all of the rewards, which we will call points, are numerical; and the agent aims to maximize its reward. In addition, our team included an initial lower bound of -10 points for the minimum reward quantity an agent can accumulate; if the reward value drops below this threshold, the training for that maze terminates and the agent moves on to the next sample maze.

In training, most actions result in a negative reward with the exception of reaching the target block (i.e. successfully completing the maze). Taking an invalid step, like attempting to step on a wall or beyond the map costs -3 points. We also have measures in place to prevent the agent from wandering aimlessly through the maze: the model penalizes an agent by -2 points if it repeats a specific path and -3 points if it creates cycles throughout the maze, providing incentive for the agent to progress through the maze faster and more efficiently.

Our model incorporates a positive reward based on the *Manhattan distance* of the agent from the desired endpoint to ensure that the agent is rewarded for getting closer to solving the maze: if the distance decreases, then the reward increases. Note that this distance function is not implemented linearly.

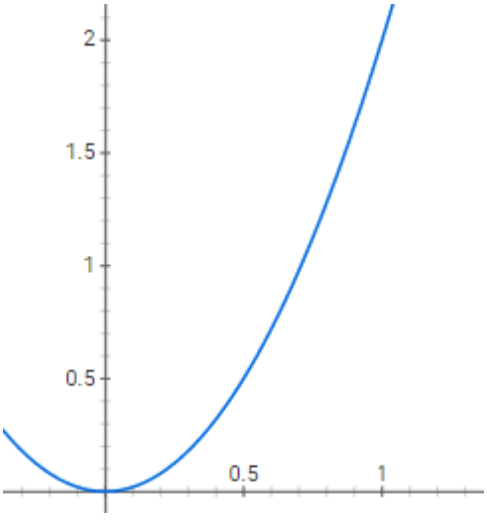

Figure 2: The closer the agent is to the finish block, the more extreme the reward is.

We utilize the Manhattan distance to calculate our model's evaluation reward, the reward for when the agent traverses a new coordinate in the maze or when the agent reaches the end of the maze. We assume our agent begins at the top left corner of the maze and the final destination is at the bottom right corner of the maze. The maximum Manhattan distance can then be calculated as `self.width + self.height - 2`. Then, we can get a ratio of how far the agent approached the final position by calculating

$$\texttt{ratio} = 1 - \frac{\texttt{curr\_distance}}{\texttt{max\_distance}}.$$

This ratio ranges from $[0, 1]$, so the higher `ratio` is, the closer the agent is to the final position. The final reward is given by $\texttt{R\_final} = W \cdot r$, where $W = 2 \cdot \texttt{ratio}^2$ is the weight calculated based on the ratio, and r is a fixed value (2.5 by default). W is the weight function because the increment of weight also increases when the ratio increases. Lastly, in the special case where the agent is one step away from the final position and decides not to go in the direction of the final position, a -3 penalty is directly applied.

Replay Buffer

Our model's time overhead is significantly optimized by one aspect of the Human-level control through deep reinforcement learning paper: the replay buffer. The replay buffer, also known as the experience, buffer, stores the most recent experience of the agent to enhance the agent's learning and improve performance. The objective of this replay buffer is to boost training performance. In a traditional reinforcement learning session, the agent only passes the current state into a neural network and learns the weights from the current state to the next state as follows from the Q-learning update code, in file replay.py, from line 82-87.

```
Q_sa_prev = np.max(self.predict(episode.get_next_state()))
action_idx = int(episode.get_action())
if episode.get_mode()==Mode.END:
    outputs[idx,action_idx]=episode.get_reward()
else:
    # gamma is the discount rate, we use 0.9 in our training
    outputs[idx,action_idx]=episode.get_reward()+self.gamma*Q_sa_prev
```

With this experience buffer, however, our agent can simultaneously learn multiple state transitions. At the end of each move in the maze, the current state is saved in the buffer, and a random sampling overwrites all previously saved states to feed these states into the neural network. While the addition of this buffer feature requires more space, it is capable of decreasing the total training time and increasing the training performance.

Epsilon Decay

The model we use also incorporates an optimization on a variable ε, the ratio of exploration vs. exploitation. Higher ε prefers exploration, or randomly choosing a direction despite previously learned information; while lower ε prefers exploitation, or specifically choosing the ideal direction learned from previous training. With these two strategies in mind, a proper ϵ allows a balance between the exploration and exploitation, giving our agent the chance to explore the global optimal path rather than relying on a learned pattern. Setting a constant ϵ is not ideal because, while the initial training may benefit from a higher value that prefers exploration over exploitation; towards the end of training, the model might be better served with a lower value to avoid aimless wandering around the maze. Due to these fluctuating needs for different ϵ values, a decaying ϵ is implemented: the calculation for `epsilon` is based on the `win_rate`, which ranges from $[0, 1]$. This formal equation for this parameter is

$$\epsilon = 0.9 \cdot \frac{e^{-x}}{(x+1)^4},$$

where e is the natural number, x is the `win_rate`, and ε is `epsilon`.

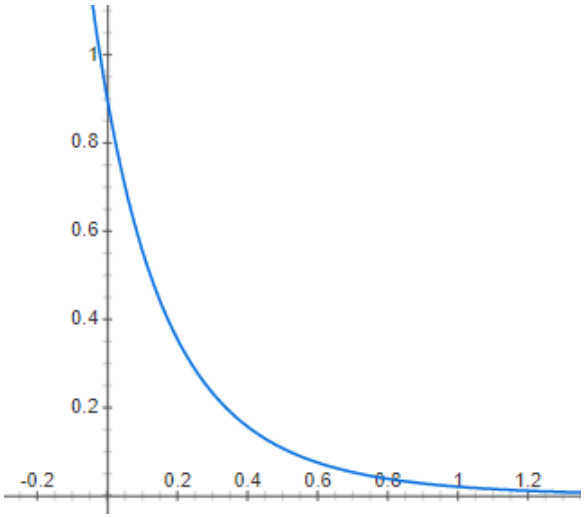

Figure 3: The value of `epsilon` decays, starting with an initial preference for exploration that transitions to a preference for exploitation as training progresses.

Experimental Results

When evaluating the performance of this model, our team had to determine which metrics to use first. Using raining times would take a non-trivial amount of time to collect (approximately 8-10 hours to run the model on 50 maps). As well, with random values between 6 to 8 minutes for 10 x 10 mazes, the training times do not reflect a direct correlation with performance. The randomness of training times may be a result of the various techniques implemented in the model training. For example, our agent is sometimes placed in a random position in the maze, and the sampling from the replay buffer is random. We can detect, at least, that the size of the maze has an exponential relationship with training time: if the size of the map increases, the training time increases exponentially at the minimum. However, training time on a set of maps of the same size is random. Because of this, we determined that it would be inefficient to rerun all maps to collect training times, so we disregarded this metric and looked to win rates instead. When analyzing the relationship between our model's win rate with the number of epochs run, there is a positive correlation with win rates growing drastically with more epochs run. With enough epochs, this relationship exists for all ten mazes run. Our team saw this as an indicator of the model performing well with the equations and biases we set within the training model to make the agent traverse these mazes.

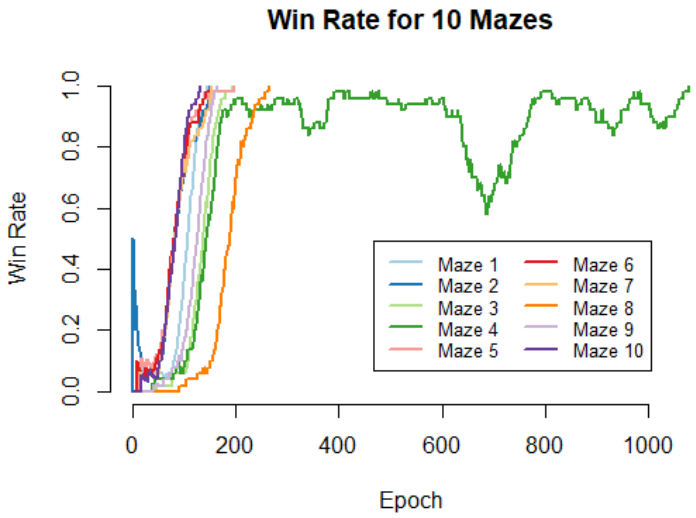

Figure 4: Over the epochs, the win rates for these 10 mazes we trained eventually reach 1.0 or 100%.

User Interface

We used Django to handle the front and back end of our site, since it is Python based and was an easier framework for our team to pick up. Using Bootstrap, we created the Navbar where the user is presented with the different features of the solver. The maze select button gives the user the option to choose one of ten pre-made mazes to have the model solve; the run-algo button, using TensorFlow JS, runs the model and has the agent solve the maze; and the reset-maze refreshes the webpage so the user can pick another maze to solve. The legend displays Remy at the starting point, cheese at the end point, and different spans for different parts of the maze (empty, maze wall, and visited).

To run a game based on a pre-trained model directly on the website, we used TensorFlow JS to process our model on the client side instead of server side. When we train the model locally, we use TensorFlow JS to export a JSON version of our model and weights, which TensorFlow JS interprets with JavaScript. When a user loads the maze map, the website fetches the JSON version of the matched model from the server side. After acquiring the model, on the client side, we loop through

1. Generate a flatten state based on the current game episode, where the length is equal to `self.height` × `self.width`.0 to represents a free path, 1 is a wall, 0.9 is the end, and 0.3 is the current location of the agent.

2. Feed the generated state into the predict function of our model by calling TensorFlow JS. We now have a list of 4 elements, each with 4 rewards for choosing a direction (indices 0 to 3 for left, up, right, and down, respectively).

3. Have the algorithm select the index of the element with the highest value among these 4 values. Higher value represents higher reward and therefore a more ideal direction is more ideal, compared to other directions based on the previous training session.

4. After determining the ideal direction, perform the action of moving in that direction, and update the UI and current state accordingly. This loop continues until the agent solves the maze.

Conclusion and Discussion

We faced many challenges throughout the process of building and training our models. Our initial goal of training an agent on several mazes proved to be infeasible when we found, late into the building process, that our model architecture would have to be changed in order to accommodate the features that we wanted. Because of this, our project turned out to be more similar to the Q-maze project than we had initially anticipated. Regardless, we were able to implement many of our own features into the project, including ones like the experience/replay buffer to better our model.

Through the ambitiousness of our project and ambition of our team, we were able to learn about many aspects of reinforcement learning and different implementations of it, even if we did not get the chance to implement every aspect ourselves. Reinforcement learning was not one of the types of machine learning covered in our class, and while we had a variety of talent on our team, not all aspects of the project were familiar to all of the members of our team. Thus, we allocated many hours and consulted numerous sources to create our final product.

In future versions of this project, we suggest implementing our initial project idea of efficiently training an agent on mazes such that the model accepts a user-generated maze, complete with visible rewards (cheese) and penalties (mouse traps) that the agent would be able to solve. This would involve a pixel-based encoding of the map and allowing the agent to view the map in its entirety instead of just its local environment.

Our code can be found in our our GitHub repository at `https://github.com/chriscruz-1/Remy_Maze_Solver` and our demo video at `https://www.youtube.com/watch?v=9OYMIYeWSEg&feature=youtu.be&ab_channel=AidanFox-Tierney`.

Works Cited

[1] Zafrany, Samy. *Deep Reinforcement Learning for Maze Solving*. https://www.samyzaf.com/ML/rl/qmaze.html.

[2] Mnih, Volodymyr et al. *Human-level control through deep reinforcement learning*. 2015, Nature. http://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf.

[3] Gasser, Michael. *Introduction to Reinforcement Learning*. 2018. https://legacy.cs.indiana.edu/~gasser/Salsa/rl.html.

[4] *Q-Learning*. Wikipedia. https://en.wikipedia.org/wiki/Q-learning.

[5] *An introduction to Q-Learning: reinforcement learning*. 2018, freeCodeCamp. https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/.

[6] Fedus, William et al. *Revisiting Fundamentals of Experience Replay*. 2020, ICML. https://acsweb.ucsd.edu/~wfedus/pdf/replay.pdf.

[7] *Markov Decision Process*. Wikipedia. https://en.wikipedia.org/wiki/Markov_decision_process.

[8] Ashraf, Mohammmad. *Reinforcement Learning Demystified: Markov Decision Processes (Part 1)*. 2018, Medium.com. https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690.