UNIVERSITY OF CALIFORNIA
Los Angeles

An Application of Reinforcement Learning Techniques in Traditional Pathfinding

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Applied Statistics

by

Britney Shaohan Brown

2022

ABSTRACT OF THE THESIS

An Application of Reinforcement Learning Techniques in Traditional Pathfinding

by

Britney Shaohan Brown
Master of Applied Statistics
University of California, Los Angeles, 2022
Professor Yingnian Wu, Chair

Modern-day navigation relies on pathfinding algorithms to determine the shortest distance between two locations. These algorithms search graphs robustly, starting at an initial node and analyzing adjacent positions connecting to the destination. Even though this technique consistently finds optimal routes, pathfinding is dependent on prior knowledge of a given environment. Reinforcement learning is a branch of machine learning capable of achieving similar results through efficient exploration, data collection, and exploitation. A form of artificial intelligence, reinforcement learning focuses on understanding the environment through incentives and penalties to make optimal decisions, eventually leading to desired target convergence. This research trains three model-free reinforcement learning techniques, advantage actor-critic (A2C), proximal policy optimization (PPO), and deep Q-network (DQN) on custom maze environments. In comparison with Dijkstra's algorithm, a standard pathfinding approach, results indicate that DQN can find analogous routes, especially when pre-trained with expert-guided behavior to reach these optimal solutions in a time-efficient manner.

The thesis of Britney Shaohan Brown is approved.

Tao Gao

Mark S. Handcock

Yingnian Wu, Committee Chair

University of California, Los Angeles

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATIONS

1 Introduction

In this day and age, it is near impossible to get lost due to global positioning and navigation systems that map us wherever we need to go. Navigational apps have become so advanced that we can personalize travel experiences, getting directions to avoid toll roads, accident sites, and even red-light cameras. While the detailed routing algorithms for these applications are kept secret, pathfinding techniques remain at the core, determining the shortest distance between two locations and cornering the market in optimal graph search.

However, pathfinding algorithms rely heavily on prior knowledge of a deterministic environment. This is not the case for most real-world applications, such as instantaneous navigation, which contains unknown obstacles and is stochastic by nature. Without providing the exact location of targets and barriers for a pathfinding algorithm, it is unable to determine the optimal path and is therefore rendered useless in a dynamic environment. Therefore, this research aims to leverage the advancements in artificial intelligence to test the applicability and efficiency of standard reinforcement learning algorithms in optimal path routing. In an attempt to combat the limitations of pathfinding, these reinforcement learning agents will be trained to solve a maze without prior knowledge of the limitations or goals of the environment.

The following research will generate custom maze environments with OpenAI Gym to test three reinforcement learning algorithms: advantage actor-critic (A2C), proximal policy optimization (PPO), and a deep Q-network (DQN). The on-policy PPO and off-policy DQN approaches successfully converged on an optimal path comparable to a common pathfinding approach, Dijkstra's algorithm. Finally, using an expert policy inspired by the resulting movements of this pathfinding method, research shows that using behavioral cloning to train a DQN agent not only solves the maze but also optimizes movement and training time.

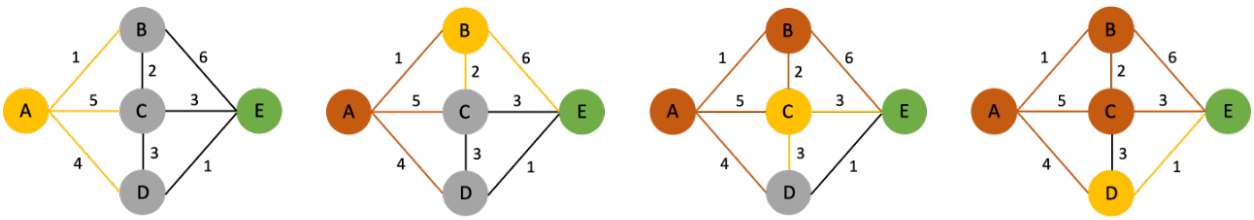
2 Pathfinding

Imagine traveling to an unfamiliar city, tired from a long flight and in search of a decent coffee shop. While it is entirely possible to start at one end of town and test every possible path in hopes of finding an establishment, this method is exhaustive and time-consuming. Thanks to modern-day technological resources, a more convenient method relies on digital mapping services to reach destinations safely and efficiently. These systems typically rely on pathfinding, a computer algorithm that finds the shortest distance between two points. This application is loosely related to the shortest path problem within graph theory, which identifies paths dependent on specific criteria such as speed or difficulty. Pathfinding is not only crucial in navigating complex road systems and obstacles but also in other worldwide necessities such as routing IP network protocols, understanding optimal flight agendas, determining autonomous robotic controls, and even modeling the spread of infectious diseases.

2.1 Dijkstra's Algorithm

In 1956, pathfinding secured its place in computer programming and routing history when Dutch software engineer Edsger Wybe Dijkstra developed a well-known algorithm for finding the shortest distance between nodes (Richards). The eponymously named Dijkstra's algorithm eliminates the use of traversal algorithms such as the Bellman-Ford algorithm which exhaustively searches every possibility, including impossible routes, to eventually determine the shortest path. On the other hand, Dijkstra's algorithm simplifies complex routing problems by utilizing dynamic programming to dissect smaller, solvable sub-problems and create optimal substructures for a combined ideal solution.

As described in “Towards Shortest Path Computation using Dijkstra’s Algorithm,” this approach performs a recursive operation that labels nodes binarily and stores the relationship with predecessor nodes in a priority queue to determine the absolute shortest path between an initial starting node and a target node (Makariye, 2017). Dijkstra’s algorithm begins by selecting the initial node and storing the distance to every directly neighboring node. At this point, all nodes that cannot be reached in a single step are placed at the bottom of a priority queue. Dijkstra’s then labels the initial node as “visited” and selects the nearest node for further investigation. The total distance to the initial node is then recalculated for every “unvisited” neighbor of this newly selected node. These distances are compared to the original, previously determined values; the smaller of the two distances is then saved in the priority queue, which now represents a visiting order for all remaining nodes. After the queue updates, the current node then transitions to a “visited” state and a new, unvisited node is selected from the top of the priority queue. Dijkstra’s algorithm recursively visits nearest neighbors until the target node is chosen from the list of unvisited nodes. At the end of this process, the optimized path is found through the stored order of predeceasing nodes in the priority queue.



(A)	Priority Queue
B	1
D	4
C	5
E	N/A

(B)	Priority Queue
C	3
D	4
E	7

(C)	Priority Queue
D	4
E	6

(D)	Priority Queue
E	5

Figure 1: Dijkstra's Algorithm

Above is an example of Dijkstra's algorithm locating the shortest path from initial node A to target node E. Although E is unreachable from A directly, the distances to neighboring nodes are stored in the priority queue and updated recursively. After evaluating distances through B, C, and D, the algorithm reveals the shortest path from A to the target E is through node D.

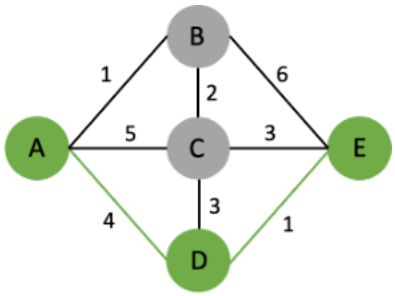


Figure 2: Dijkstra's Optimal Route

However, Dijkstra's algorithm bases its search on a weighted, non-negative, and acyclic graph (Huang et al.). Unlike the example above, these factors are not found in realistic representations of navigational and global positioning systems due to an assortment of unknown obstructions and inconsistences. To remain relevant in modern technological advances, pathfinding approaches need to adapt routes quickly and effectively for dynamic obstacles.

3 Reinforcement Learning

"What we want is a machine that can learn from experience."

~ Alan Turing, the father of artificial intelligence, in 1947

Since the conception of machine learning (ML) in the 1950s, the field has further developed into three primary varieties including supervised, unsupervised, and reinforcement. In supervised learning, the goal is to create a model that accurately describes a labeled dataset, allowing for proper categorization and subsequent prediction of new data (Hayes). On the other hand, unsupervised learning focuses on the underlying structure of an unlabeled dataset, utilizing the relationships between datapoints and their respective characteristics (Bhatt). Both task-driven supervised and data-driven unsupervised learning are commonly used for classification as well as prediction. Finally, unlike these two ML branches, reinforcement learning specializes in interacting and gathering data from an unknown environment- observing consequences, maximizing rewards, and learning appropriate behaviors.

3.1 Definitions

This section contains a brief high-level overview of reinforcement learning (RL) notations and descriptions to provide the necessary background for the following concepts and algorithms. Beginning with a dive into Markov Decision Processes (MDPs), this section will also cover the use of policy, value, quality, and advantage functions as well as how these functions can be estimated using the Bellman equation. This section also covers the architecture of a reinforcement learning episode as well as the discounted reward used to motivate learning.

3.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a stochastic, discrete-time control process that provides the mathematical framework of reinforcement learning algorithms. An extension of Markov chains, MDPs allow a decision maker to choose actions that maximize rewards within a given environment. For each step, this decision maker must choose an available action and the MDP reacts by moving to a new state and producing a positive or negative reward. This process creates a state transition function where each next state is dependent on the current state and action alone, satisfying the Markov property of conditional independence.

Therefore, every MDP can be broken into four main components: states, actions, rewards, and transitions. The notation S represents the set of finite states describing the current environment while $A(s)$ symbolizes every possible action for a given state. The reward, written as $r = R(s, a)$, is a function that provides a real value to denote the consequence of choosing a certain action at the current state. Finally, the transition model $P(s' | s, a)$ describes the probability of transitioning to a new state s' when selecting action a at current state s. This transition model represents all possible combinations of $s, a,$ and s' for the environment.

3.1.2 Policy Function

A policy function defines how an agent behaves, detailing which action to perform in each state. These policies are either deterministic $a = \pi(s)$ or stochastic $a = \pi(a|s)$. Deterministic policies indicate what action to take for a given state while stochastic policies are functions containing a random probability for action selection (Del Pra). Therefore, a deterministic policy can be viewed as a special stochastic policy containing an exclusive probability of 1 for one of the actions and 0 for the remaining options.

3.1.3 Episode

An episode is a sequence of observed states and actions construed through timesteps, which vary from 0 to a finite or infinite end value. For every timestep t, an agent observes the given state $s_t \in S$ and takes an action $a_t \in A$. The agent then obtains a reward r_t and transitions to a new state s_{t+1} selected by the state transition model $P(s_{t+1} | s_t, a_t)$. Finally, the new state s_{t+1} becomes the current state s_t and this loop continues until the agent reaches a terminal state in which all actions receive a reward of 0 to represent the end of an episode (Zychlinski).

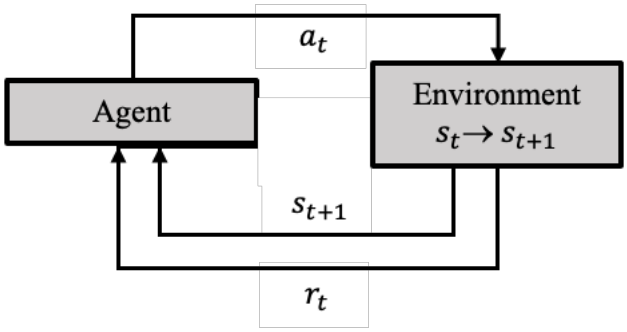


Figure 3: Standard Reinforcement Learning Architecture

3.1.4 Discounted Return

Using the rewards from each timestep t, a long-term cumulative reward, known as the return G_t, is calculated at the end of each episode. In addition to the immediate positive or negative reward for every timestep, r_t, a discount factor $\gamma \in [0\,,\ 1)$ is used to determine the importance of each state (Del Pra). This discounted return is denoted as follows:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots = \sum_{n=0}^{N} \gamma^n r_n \qquad (1)$$

If the discount factor is 0, RL agents only consider the current reward, seeking immediate rewards rather than choosing a less-rewarding action that leads to a significantly larger reward

later in the episode (Jones). If the discount factor approaches 1, an RL agent heavily considers the consequences of current actions in hopes of greater long-term rewards (Torres). For example, imagine two students with respective discount factors near 0 and 1. The first might party one night, being forced to spend all weekend catching up while the later would study during the week, earning a homework-free, party weekend. Similarly, most RL algorithms select larger discount factors to create agents that are heavily inspired by future rewards.

3.1.5 Value and Advantage Functions

The state-value function, denoted simply as the value function $V(s)$, measures the overall expected return G_t. Starting at the current state s_t, the agent performs N timesteps, which represents the total number of states until reaching the terminal state and ending the episode.

$$V(s) = \mathbb{E}\left[G_t \mid s_t\right] = \mathbb{E}\left[\sum_{n=0}^{N} \gamma^n r_n\right] \tag{2}$$

Similarly, the action-value function, also known as the quality function $Q(s)$, measures the expected long-term return when the agent performs action a at current state s.

$$Q(s,a) = \mathbb{E}\left[G_t \mid s_t = s, a_t = a\right] = \mathbb{E}\left[\sum_{n=0}^{N} \gamma^n r_n\right] \tag{3}$$

The difference between the state-value and action-value functions is subtle. At $n = 0$, the value $V(s)$ reward r_0 is the expected reward when the agent is in state s, before the agent performs any action (Zychlinski). The quality $Q(s)$ reward r_0 is the expected reward after playing an action. This distinction is important when measuring the advantage function $A(s,a)$, which demonstrates if a certain action is a good or bad decision. Essentially, this function measures the advantage of selecting particular actions compared to others when an agent is in a fixed state.

$$A(s,a) = Q(s,a) - V(s) = \mathbb{E}[r(s,a) - r(s)] \tag{4}$$

3.1.6 Bellman Equation

The Bellman equation is used to get an estimate for both value functions by breaking them into two parts: the immediate reward and the discounted value of the next step, which is chosen with the transition model *P* (Torres). Rather than finding the cumulative sum over the remaining timesteps, the Bellman equation simplifies this calculation, optimizing a discrete-time solution using dynamic programming (Tanwar). Here, the Bellman equation is expressed in terms of both the state-value and action-value functions respectively:

$$V(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)V(s') \tag{5}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)Q(s',a') \tag{6}$$

In addition, this equation is used to determine the maximum of the action-value function, which is also referred to as the optimal quality function $Q^*(s,a)$:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a'} Q^*(s',a') \tag{7}$$

Subsequently, selecting the action that maximizes $Q^*(s,a)$ results is the optimal policy $\pi^*(s)$:

$$\pi^*(s) = arg \max_{a \in A} Q^*(s,a) \tag{8}$$

3.2 Reward System

Reinforcement learning is well-suited for sequential decision problems, such as navigating a maze, since the ultimate goal is finding the optimal sequence of actions that

maximize long-term gains (Hayes). In 1972, Stanford professor Walter Mischel and his graduate researcher Ebbe Ebbesen conducted a delayed gratification study in which a child is presented with two options: either a small, immediate reward or a better, larger reward if they choose to wait an extended period. Researchers typically offered one marshmallow as the immediate reward and told participants that waiting for 15 minutes would result in two marshmallows, rather than just one. Some children would choose the initial treat while others decided to delay their gift, opting for the larger reward. The latter, reward maximization, is how RL algorithms are taught to learn.

In an initially unknown environment, reinforcement learning agents interact and explore their surroundings. They gather valuable information by implementing a combination of random and informed actions. The environment responds accordingly, giving the agent delayed feedback through rewards. These rewards can be positive to enforce desired behavior, negative to emphasize the severity of mistakes, or zero to represent neutral action choices. This reward system allows for full comprehension of long-term rewards and consequences. Rather than making the best stand-alone decision in the moment, agents learn a sequence of actions that maximize the overall reward, therefore gaining the capability of sequential decision making.

3.3 Exploration-Exploitation Dilemma

Imagine someone moving to a new town. They would need to explore restaurants in the area before determining which places have the best food and quality service, eventually exploiting this knowledge by continuously going to their favorite restaurants. If this person stops searching for new restaurants after finding one place, they may never find another restaurant with amazing food and excellent service. The same dilemma occurs for RL agents, which have two goals when interacting with an environment: exploration and exploitation.

The success of many reinforcement learning agents is entirely dependent on their exploration strategy. If an agent performs the best action for every iteration, it is likely that they never explore the entire environment. By limiting the state space in such a manner, the agent never learns about the rewards and consequences of each possible action, regardless of its effect on the long-term expected return. On the other hand, if the agent only makes random choices, it will never learn a sequence of optimal actions. Therefore, the solution is to combine the ability to explore and train on environmental information while exploiting this knowledge to make better decisions and narrow exploration where greater rewards are more likely.

3.3.1 Epsilon Greedy Strategy

The solution to this suboptimal performance is an exploration vs exploitation trade-off such as epsilon greedy (ε-greedy) action selection. Under this strategy, a random action is chosen with probability ε while the "greedy" action, which is the action that maximizes $Q(s, a)$, is chosen with probability $1 - \varepsilon$. While this may lead to short-term losses, this strategy allows for the agent to collect enough information to make the best, informed decision for the future.

3.3.2 Epsilon Decay

The ε-greedy strategy allows an agent to select the best action at almost every step, with a small probability ϵ of selecting a random action. Since higher epsilon values ε leads to more exploratory agents, many RL methods implement an epsilon decay. By steadily decreasing ε over a training period, environmental exploration is highly encouraged in early iterations but surpassed by exploitative actions to optimize the action-value function $Q^{\pi}(s_t, a_t)$. In other words, as confidence in the learned policy increases, epsilon decay ensures that exploitation becomes a priority over continued exploration.

3.4 Model-Based Algorithms

In most real-world problems where reinforcement learning is applied, the state transition model $P(s' | s, a)$ and reward function $R(s, a)$ are unknown. A RL agent must explore the environment to obtain accurate knowledge of the MDP dynamics, sampling actions to build an estimate of the optimal action-value function $Q^*(s, a)$ and subsequent best policy $\pi^*(s)$. This approach is known as model-based reinforcement learning.

Within these algorithms, environment exploration learns a model to plan optimal controls, simulate new episodes, and obtain the best sequence of actions. During training, these algorithms interact with the environment and build a virtual transition model without waiting for an environmental response. Since most models behave linearly, this model-based process only requires a few samples to effectively learn MDP dynamics.

Every model-based algorithm begins with the initialization of the action-value function $Q(s, a)$ and a parametric model $m(s, a)$. Two of the most common parametric models to represent MDP dynamics are the Gaussian Process and Gaussian Mixture Model. The first makes predictions using the Gaussian distribution while the second is a probabilistic model that assumes the data is a mixture of multiple Gaussian distributions with unknown parameters (Del Pra).

Using the base quality function $Q(s, a)$ and model $m(s, a)$, the algorithm then loops through the following three stages: acting, model learning, and planning. In the first stage, the base policy $\pi_0(a_t, s_t)$ selects and performs an action in the environment, collecting observations $\{s_t, a_t, s_{t+1}\}$ at each timestep. From these sample trajectories, stage two begins with a supervised learning algorithm which trains the model to minimize the least square error between the model's predicted state and the environment's sample trajectory. Finally, the planning stage

uses the parametric model and a random state-action sample to predict the next state and reward, updating the policy and repeating this process *N* times until reaching a terminal state:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[\gamma \max_{a} Q(s',a) - Q(s,a)\right] \tag{9}$$

Once this loop is complete, the agent is theoretically able to utilize the updated model to perform explicitly in the real environment, even though it only used sample trajectories to train.

3.4.1 Model Predictive Control

Unfortunately, the model-based approach described above is vulnerable to drifting since the base policy is usually too small to cover the entire environment before small errors lead to uninformed predictions (Del Pra). This drift occurs when the sample trajectory finds itself in states that the model has yet to encounter, making it impossible for the planning stage to find the optimal controls. The solution is to continuously sample and fit the model throughout training, instead of only learning the model at the start of the algorithm. This continuous feedback from the real environment is known as model predictive control.

After the initial acting, model learning, and planning stages are complete, model predictive control moves onto a fourth stage: execution. In this stage, the algorithm takes a predicted action a given the current state s, and the resulting new state s' is appended to the trajectory dataset. In the next phase, dataset update, the algorithm alternates between either the learning or planning stage for the remainder of training. This process allows the algorithm to refit the model, optimizing the entire trajectory and allowing corrective measures for repeated states (Del Pra).

3.4.2 Disadvantage

In both the root model-based approach and the model predictive control evolution, an optimal policy is found using a pre-determined model, whether or not it reflects the true mechanisms of the environment. If the model is inaccurate, then there is a large risk of the agent learning unwanted or useless behaviors. Therefore, to create an agent that can functionally operate in a complex, real-world environment, model-free reinforcement learning methods are used to combat the cons of estimating MDP environment dynamics.

3.5 Model-Free Algorithms

In model-free reinforcement learning, the optimal policy $\pi^*(s)$ is obtained through direct interactions with the environment. Instead of learning a virtual model of environment dynamics, the agent optimizes the policy by maximizing the expected discounted return G_t. One unique aspect of model-free agents is that they learn efficient state-action pairs without understanding what each observation and action attributes to within the environment. For example, an agent might learn that selecting action 1 has the best reward but is unaware that this action directly correlates to moving south, out of a blocked-in corner of a maze. Fortunately, even without correlating action to real-world motion, model-free algorithms are more than capable of building effective learning agents. Within the model-free algorithm umbrella, there are two main varieties: on-policy and off-policy, also known as policy-based and value-based respectively. The difference in these two approaches is dependent on how information is gathered.

3.5.1 On-Policy Methods

With on-policy algorithms, actions are selected based on the best policy at the time of decision making. At each timestep, information from said actions are then used to continuously

improve the policy and value function $V(s)$. This method guarantees convergence on an optimal policy π^* that performs the best action for each observational state.

3.5.1.1 Policy Gradient

Policy gradient is the baseline of all on-policy methods. It aims to find a vector of parameters θ that maximize the value function $V(s,\theta)$ under a parametric policy $\pi(a,|s,\theta)$. The Softmax policy is typically used for discrete actions since the function can convert the output into a distribution of probabilities, as seen below with the $\emptyset(s,a)$ vector representing the current state-action selection:

$$\pi(a|s,\theta) = \frac{e^{\emptyset(s,a)^T\theta}}{\sum_{k=1}^{N} e^{\emptyset(s,a_k)^T\theta}} \tag{10}$$

On the other hand, the following Gaussian policy is used if the action space is continuous:

$$\pi(a|s,\theta) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(a-\mu)^2}{2\sigma^2}} \tag{11}$$

While policy gradient traditionally uses the Softmax or Gaussian policies shown above, any deterministic or stochastic policy differentiable with respect to θ can be used as the parametric policy $\pi(a|s,\theta)$ to choose an action, generating a sequence of observations (s,a,r) and recalculating the total long-term discounted reward G_t:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \tag{12}$$

These observations are also used to update the vector of parameters θ_t in a traditional gradient descent process with a preselected learning rate $\alpha > 0$:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta V(s, \theta) = \theta_t + \alpha G_t \nabla_\theta \ln \pi(s_t | s_t, \theta) \quad (13)$$

When the algorithm converges, the result approximates the optimal policy π^*. In addition, the simplicity of learning a vector of parameters θ in policy gradient avoids an exponential growth in computational energy and memory usage in the case of a sizeable environment or action space.

However, one of the largest disadvantages with the policy gradient method is high variance, which can lead to vastly different results for identical training runs (Del Pra). The standard solution is to utilize the advantage function $A(s, a)$ which compares the chosen action to the average action at any given state (Karunakaran). When $A(s, a)$ is positive, this indicates that the agent's chosen action resulted in a good reward, so the policy updates accordingly by increasing the probability of selecting this particular action. Using the same logic, a bad action shows a negative advantage value and the policy updates with a subsequent reduction in probability. Unfortunately, it is common for this advantage function approach to have noisy estimates since the value function $V(s)$ and policy have separate neural networks. The step size drastically effects the learning ability of a neural network's line search to determine the direction for optimization. If the size is too small, expected convergence time increases, but if the size is too large, the learning process over-fluctuates and never converges (Karunakaran).

3.5.1.2 Trust Region Policy Optimization

Therefore, when using the advantage function to reduce gradient variance, it is highly recommended to implement Trust Region Policy Optimization (TRPO). Instead of an ineffective line search strategy for $V(s)$, this approach constructs a circular evaluation region using the step size as the radius (Karunakaran). Within this trust region, the algorithm searches for a local maxima and uses this particular point to determine the direction of the next trust region. This dynamic approach ensure that the new, updated policy is not significantly different from the

previous policy, controlling for potential divergence. The TRPO algorithm aims to maximize the following objective subject to the Kullback-Leibler (KL) regularization below:

$$E_{\tau\sim\pi\theta_{old}}\left[\frac{\pi_{new\,\theta}(a|s)}{\pi_{old\,\theta}(a|s)}A_{old\,\theta}(s,a)\right] \tag{14}$$

$$E_{\tau\sim\pi\theta_{old}}\left[\overline{D_{KL}}\left(\pi_{old\,\theta}(.|s),\pi_{new\,\theta}(.|s)\right)\right] \leq \delta \tag{15}$$

This KL constraint penalizes the new policy for diverging but typically adds unnecessary computational overhead, so the TRPO is generally passed over for a simpler actor-critic implementation known as proximal policy optimization.

3.5.1.3 Actor-Critic

Unlike policy gradient which only optimizes the policy, the actor-critic method estimates and updates both the policy and value function. However, the main difference between these two methods is how they combat the high variance in the vector of parameters θ. Policy gradient uses the advantage function while actor-critic algorithms use temporal difference (TD) error which is the subtraction of a baseline $b(s)$ from the total discounted return G_t:

$$\delta = G_t - b(s) \tag{16}$$

In actor-critic algorithms, the critic is an estimate of a value function, such as the state-value function $\hat{V}(s,w)$, and is found through a neural network containing a vector of weights w (Yoon). This estimated value function $\hat{V}(s,w)$ represents the baseline function $b(s)$. The actor of actor-critic algorithms then updates the direction of the policy according to suggestions from the critic.

Within each episode, the actor observes the given state and takes an action based on the current policy $\pi(a, |s, \theta)$. The estimated baseline function $\hat{V}(s, w)$ then uses the reward r and next state s' outputs to calculate the temporal difference for that timestep:

$$\delta \leftarrow r + \gamma \hat{V}(s', w) - \hat{V}(s, w) \tag{17}$$

This resulting error is then used to update the weights for the critic's estimated baseline function $\hat{V}(s, w)$ as well as the actor's policy parameters θ, which are shown below respectively. As noted here, actor-critic algorithms typically have different step sizes α_w and α_θ for the separate weight and parameter updates:

$$w \leftarrow w + \alpha_w \beta \delta \nabla_w \hat{V}(s, w) \tag{18}$$

$$\theta \leftarrow \theta + \alpha_\theta \beta \delta \nabla_\theta \ln \pi(a, |s, \theta) \tag{19}$$

This dual update loop continues for each step within an episode until reaching a terminal state. Once an episode is complete, the updated policy parameters θ and state-value function $\hat{V}(s, w)$ are initialized for the next training episode. The figure below represents an episodic training loop for actor-critic algorithms:

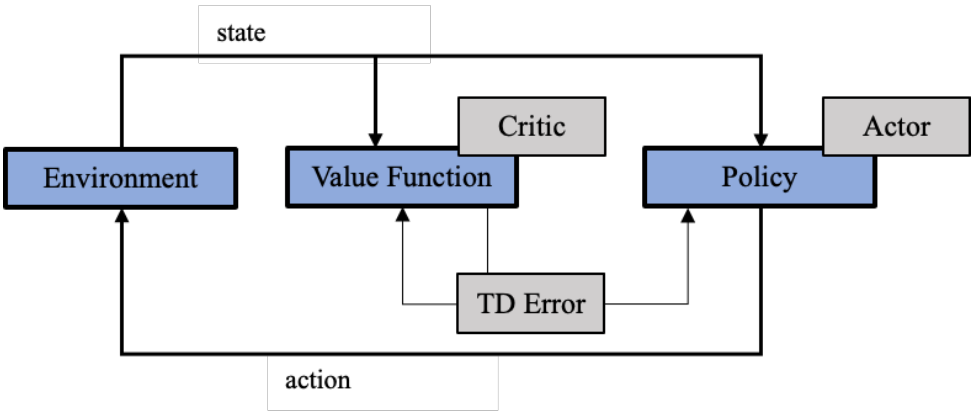


Figure 4: Actor-Critic Algorithm

3.5.1.3.1 Advantage Actor Critic

The most common expansion of the base actor-critic method is the advantage actor critic. Instead of estimating the state-value function, this approach utilizes the advantage function $A(s, a)$ to compare actions at the current state. The advantage function is rewritten below in terms of Bellman's relationship between the estimated action and state values:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \tag{20}$$

There are two variants of the advantage actor critic: A3C and A2C. In asynchronous advantage actor critic (A3C), there are multiple agents in parallel environments training independently to update the global value function (Yoon). In A2C, only one agent is used in a singular environment. While asynchrony is thought to have improved performance through regularization or enhanced exploration, empirical results reveal that both approaches have similar results (Mnih et al.) with A2C reigning supreme in terms of computational efficiency.

3.5.1.3.2 Proximal Policy Optimization

Proximal policy optimization (PPO) defines the following probability ratio $r(\theta)$ between the current and next policy:

$$r(\theta) = \frac{\pi_{new\ \theta}(a|s)}{\pi_{old\ \theta}(a|s)} \tag{21}$$

Similar to the policy gradient approach, this objective function is unstable; if the appropriate step size is not selected, convergence may never occur. Therefore, PPO implements a hyperparameter ϵ to constrain the policy ratio to a small interval $[1 - \epsilon, 1 + \epsilon]$. This clipped policy ratio minimizes the possibility of large policy divergence and guarantees eventual convergence, allowing for stable training and reproducibility:

$$J^{CLIP}(\theta) = \mathbb{E}\left[min(r(\theta)\hat{A}_{old\,\theta}(s,a), clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_{old\,\theta}(s,a))\right] \quad (22)$$

However, if an environment has sparse rewards, it is entirely possible that PPO converges but is unable to solve the environment. In this unfortunate case, PPO operates with little to no learning signal unless it randomly reaches the goal. Since this on-policy approach performs a gradient update at the end of each episode, this rare occurrence is not sufficient for the policy to teach the agent to consistently find the goal state. Therefore, a higher entropy coefficient for PPO agents is typically used to inspire more random actions (AurelianTactics). Entropy directly effects the unpredictability of chosen actions and enhances exploratory behavior to increase the possibility of finding the goal more often during training (Xin et al.).

3.5.2 Off-Policy Methods

In off-policy algorithms, the focus is to approximate the optimal state-value $V(s)$ or action-value $Q(s)$ functions (Del Pra). With this value-based approach, an agent may deliberately choose an action that acts against the known optimal action at the time of decision making. This explorative nature allows an off-policy agent to further interact with the environment to determine if sub-optimal, short-term actions lead to greater long-term rewards.

3.5.2.1 Q-Learning

Q-learning is a common model-free, off-policy reinforcement learning algorithm that uses a table $Q^{\pi}(s_t, a_t)$ to store the quality function's resulting values for all possible state-action pairs. To initialize this Q-table, each state-action value is set to a small random number, except for terminal states which are set to zero to represent episode completion. For each timestep, an action is performed and the sequence of observations (s, a, r, s') is used to iteratively update the prior $Q^{\pi}(s_t, a_t)$ values using an expansion of the Bellman equation which contains a fixed learning rate α and a discount rate γ to control the effect of new information:

$$Q'(s,a) = (1-\alpha) * Q(s,a) + \alpha(r + \gamma\, Q(s', argmax\, a' : Q(s',a'))) \tag{23}$$

As seen above, the updated Q-table $Q'(s,a)$ is a weighted average of the current and new Q-values. The latter is the sum of the immediate reward r and the expected discounted reward Q after choosing the optimal action. Once each state-action pair is sufficiently visited through an exploration strategy and incrementally updated, the optimized table converges to represents the expected discounted reward G_t for every action a performed in state s (Van de Kleut, "The Mathematical Foundations of Reinforcement Learning").

3.5.2.2 Value Function Approximation

The value function approximation method estimates the optimal policy $\pi^*(s)$ by iteratively approximating a parametric action-value function $\hat{Q}(s,a,w)$ with a neural network containing a vector of weights w (Del Pra). For each iteration within an episode, the current state-action pair reveals an environmental reward $R(s,a)$ and new state s'. These outputs are then used to update the weight parameters using gradient descent:

$$\Delta w = \alpha \left(R(s,a) + \gamma \hat{Q}(s',a',w) - \hat{Q}(s,a,w) \right) \nabla_w \hat{Q}(s,a,w) \tag{24}$$

3.5.2.3 Deep Q-Networks

In reinforcement learning, it is rare to have an environment with small, discrete spaces for both the action and state space, which are required for Q-learning. Instead, the more likely scenario is that one or both spaces are continuous. Unfortunately, the time it takes to reach convergence for a massive Q-table in such environments is a significant technical challenge (Choudhary) which can be combatted with more advanced algorithms such as Deep Q-Networks (DQNs). Similar to the value function approximation approach, DQN replaces the state-action table with a deep neural network.

In "Deep Q-Learning with Neural Networks", Van de Kleut explains that Q-Learning, $Q^{\pi}(s_t, a_t)$ is a function over state-action pairs in the form of $S \times A \rightarrow \mathbb{R}$. In DQN, the action space is limited to low-dimensional discrete actions so the Q-table can be rewritten as $Q^{\pi}(\cdot, a_t)$, detailing A continuous functions in the form $S \rightarrow \mathbb{R}$. In DQN, the neural network estimates the Q-table by corresponding each A output node to a differentiable function approximator $Q_{\theta}(s_t, a_t)$ for every action a_t. In addition, since $Q^{\pi}(\cdot, a_t)$ is continuous over *S,* similar states ($s \approx s'$) will have similar Q-values $Q^{\pi}(s, a) \approx Q^{\pi}(s', a)$ so nearby states s_t' improve alongside current states s_t, allowing for generalization of unobserved and untrained states.

In the deep neural network, the estimates for $Q^{\pi}(s_t, a_t)$ will eventually converge as the environmental response (s_t, a_t, r_t, s_{t+1}) is used to repeatedly update the parameters θ with respect to the following loss function:

$$L(\theta) = \left(Q_{\theta}(s_t, a_t) - r_t + (1 - d_t)\gamma \max_{a_{t+1}} Q_{\theta}(s_{t+1}, a_{t+1})\right)^2 \quad (25)$$

However, when updating these parameters θ, the predictions for other state-action pairs also change. On any given step, if θ is modified to reflect a higher estimate, then the update for the next estimate will also be higher. This leads to a runaway effect where every current state influences the next state's prediction until bootstrapping target values from the Q_{θ} training network leads to uncontrollable predictions for all $Q_{\theta}(s_t, a_t)$ estimates. DQN utilizes two techniques to minimize this instability and improve learning: target networks and replay buffers.

To reduce the runaway effect of bootstrapping dynamic targets from a model in training, DQN stores an identical copy of the Q_{θ} network, denoted as the target network Q_{θ}^{-}. This target network only synchronizes parameters with Q_{θ} every n_{θ} timesteps, so Q_{θ}^{-} parameters remain the same between each synchronization (Van de Kleut, "Deep Q-Learning with Neural Networks").

These infrequent updates generate target values for the Q_θ network to minimize loss with a stable constant.

Another common method for mitigating training instability is the use of replay buffers, also known as experience replay. This technique reuses past observations to avoid catastrophic forgetting, which is the unfortunate situation when θ updates lead to unlearning since decreasing loss for the current training samples can increase loss for older training samples (Van de Kleut, "Deep Q-Learning with Neural Networks"). By storing previous transitions (s_t, a_t, r_t, s_{t+1}) in a replay buffer, θ updates with a sample batch, rather than just the most recent transition. This process not only avoids forgetfulness but also reduces overfitting and increases the likelihood of convergence. For further improvement, more advanced exploration strategies are applied to increase the proportion of episodes that reach the goal. With more successful runs stored in the replay buffer, DQN is more likely to converge on an optimal policy.

With a target network and experience replay, DQN is significantly more efficient than the generic off-policy Q-learning algorithm, especially when the action or observation space is large. Here is a visual representation of the DQN training loop described above:

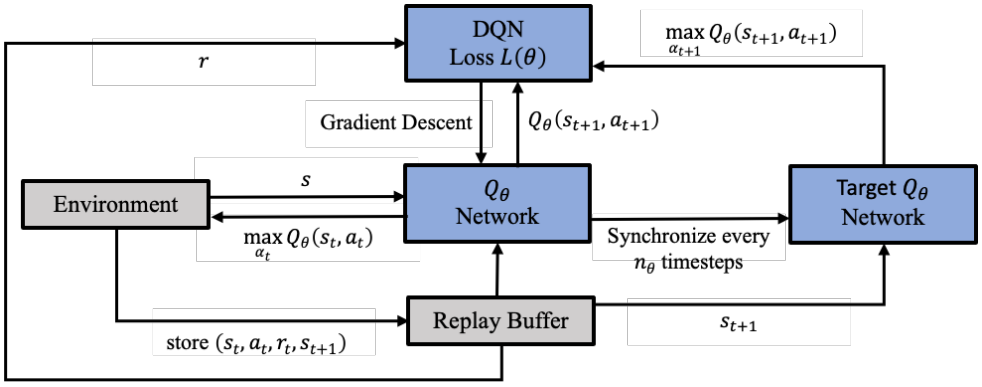


Figure 5: Deep Q-Network Algorithm

3.6 Imitation Learning

Reinforcement learning achieves optimized results in many fields such as robotics, text mining, financial trading, and even healthcare. However, these successes generally occur after millions of iterations and extensive computational power since RL agents are far from perfect during initial training. Due to the unpredictability of visited state spaces, an agent often makes mistakes, learning to behave poorly and performing inadequately in the specified environment.

Unlike computer simulations, humans rarely learn from scratch. We learn to walk by watching our parents, holding their hand, and relying on their guidance, eventually understanding the physical motions necessary to walk by ourselves. If we were told to walk without knowing what walking is, it would take a significantly longer time to learn.

For RL agents, a process known as imitation learning provides an expert to assist agents in "walking", helping them learn through demonstrations and guidance. This approach rewards through examples, rather than assuming an agent explores an environment efficiently enough to obtain a decent policy ("Imitation Learning"). Agents learn good behavior even if rewards are sparse or inaccurately represent the true objectives of an environment. Behavioral cloning is a subclass of imitation learning that uses a set of expert demonstrations to build an expert-imitating policy. This pre-trained policy is then used as the initial parametric policy for an RL agent to build upon. With these expert demonstrations, agents can quickly learn the general rules of an environment, narrowing focus on optimizing the task at hand.

Figure 6: Behavioral Cloning Architecture

4 Experiment

To begin testing the efficiency of reinforcement learning algorithms compared to traditional pathfinding, several custom environments will be generated using OpenAI Gym. To mimic the routing capabilities of navigational applications, these environments will represent a system of buildings and roads. While navigation is typically represented by multi-directional graphs, as seen in *Figure 1*, this research will focus on maze environments, otherwise known as undirected graphs. Unlike Dijkstra's algorithm which chooses neighboring nodes to find the target, RL agents will use a predefined reward system to learn which boundaries to avoid and which paths to take, eventually building a graph in which a path can be optimized.

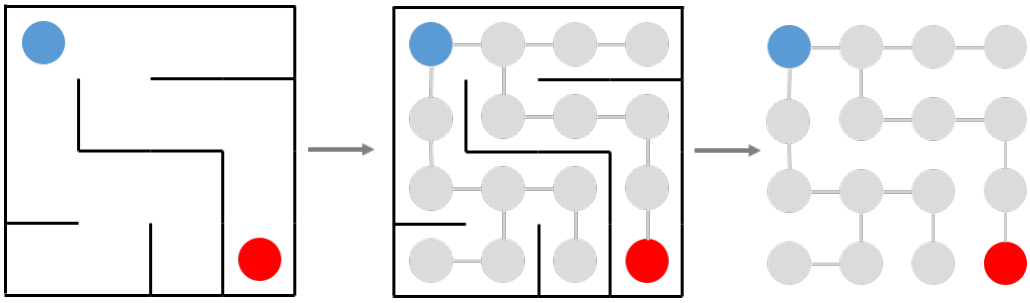


Figure 7: Visual Representation of an Undirected Graph

Finally, this section will train and compare the convergence results and optimal actions found by three reinforcement learning agents and Dijkstra's pathfinding algorithm.

4.1 Custom Environment

One of the largest challenges in training reinforcement learning agents for real-world scenarios is finding virtual environments that accurately represent the complexities of reality. In December of 2015, investors such as Elon Musk, Sam Altman, and Microsoft pledged over one billion US dollars to form OpenAI, an organization dedicated to sharing patents and research

with the general public, freely collaborating with worldwide researchers and institutions (Markoff). On the 27th of April the following year, the organization released a beta version of OpenAI Gym, a platform designed to expand reinforcement learning capabilities (Gershgorn).

Through a simple interface and general-intelligence benchmark for a wide variety of simulated environments, Gym began to set the standard for reproducible environments in AI research publications (Gershgorn). This toolkit is generally used to develop and compare algorithms on a collection of pre-made environments with enormous state or action spaces, such as Atari video games and simulated robotics (Brockman). This research utilizes Gym's capability to create custom maze environments with building-like obstacles to test the efficiency of reinforcement learning algorithms verses current pathfinding standards in navigation.

4.1.1 OpenAI Gym

For customized maze scenarios, each environment must first inherit the base class *gym.Env* which encapsulates the arbitrary dynamics of both partially and fully observed environments (Brockman). In the initialization of any environment, a user must define the *action_space* and *observation_space*. These two variables must be of Gym's *space* class and are typically a one-dimensional discrete type or a n-dimensional box type (Pocs). For this particular use case, the action space is discrete with a length of four to represent the four possible directions of movement: north, south, west, and east. The observation space is defined by the more complex box type with a matrix containing bounded values to represent the four potential maze objects: free space, obstacle, agent, or goal.

Using the defined action space as an input, the *step* method takes a step in the environment and returns a tuple of information in the following order: next observation, numeric reward, terminal state flag, and an optional dictionary of user-specified information ("API").

The terminal state flag is a Boolean value that lets the agent know whether the environment's end state has been reached (Pocs), prompting the *reset* method to restart the environment, returning the agent to the initial starting point.

4.1.2 Obstacle Generation

Drawing inspiration from *mazelab*, a customizable framework for gridworld environments (Zuo), this project builds and stores four mazes of varying difficulty to train RL agents. Scikit-image, an open-source Python package for image processing, contains a *random_shapes* function that is used to generate two-dimensional numpy arrays with rectangular shapes of random sizes, quantities, and overlap. Once coerced into a binary format, these arrays are used to represent the free spaces and obstacles of a maze. Then, a final check guarantees an agent can move from the initial starting point to the end point without crossing an obstacle. This $n \times n$ array is then piped into the custom Gym environment described above.

4.1.3 Maze Environment

The infrastructure for the basic maze is built using a custom abstract base class. Within this class are two abstract methods *size* and *make_objects* which respectively return the dimensions of the maze and the list of defined objects. The *size*, n , is extracted from the above numpy array while the maze objects are built using a custom data class defining data properties such as an object's name, value, RGB tuple, impassable Boolean value, and position. The value, and position are used to define the location of a maze object within the environment. For example, the agent takes on the value 2 at $[1,1]$ while free spaces and obstacles are represented by 0 and 1 respectively. Therefore, the numeric characterization of the upper left side of the maze environment is:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Figure 8: Initial Maze before Action

During training, the agent chooses an action by selecting a value ranging from 0 to 3, representing the four different directions it can take: north, south, west, and east. Using this chosen action, Von Neumann neighborhood motion determines the direction of travel within the matrix such as [1,0], which portrays a movement one unit south. This motion defines a tentative new position for the agent which is then checked for validity in terms of a free space rather than an obstacle. If the selected action results in moving to an impassable obstacle, the current step is marked as invalid, and the agent's position remains the same. However, if the agent's move is valid, the position updates and is parsed through to the environment. For example, if the agent chooses valid action 1, the agent will move south in the environment:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Figure 9: Initial Maze after Action

The final portion of the *step* function assigns a reward using a custom predefined system. When an agent chooses an invalid position, it is assigned a -0.1 reward, which acts as punishment for this unwanted behavior. If the agent selects a valid position but has yet to reach the goal, it is assigned a smaller punishment, -0.001, which validates a good action while encouraging the agent to reach the goal quicker. Finally, once the agent reaches the goal, it receives a large reward, 10, and the Boolean terminal state flag switches to done. With this reward system, an efficient agent would consistently receive a total return slightly under 10.

4.1.4 Visualization

With the above environment mechanics, Gym's *render* method is then used to transform the numpy arrays into a readable format. The agent is set to start in the upper left corner with the goal placed in the bottom right corner. Finally, the following maze environments, generated with varying difficulty levels, are built and ready to train RL agents in the following section:

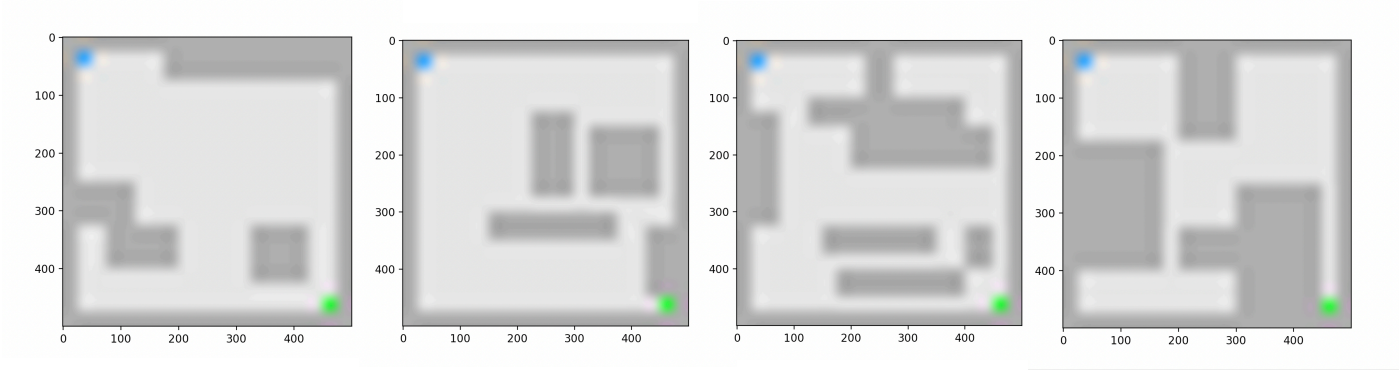


Figure 10: Custom Maze Environments

4.2 Training

This section details the set-up and training for learning agents operating in the maze environments above. Since model-based algorithms tend to prioritize maximizing rewards by performing actions irrespective of consequences, this research will focus on model-free algorithms that learn the consequences of their actions through experience. For the on-policy side of the model-free spectrum, both the A2C and PPO methods are used while DQNs are the only off-policy method explored in the navigation-inspired maze environments.

To find the most optimal path in the maze environments, our research implements Dijkstra's algorithm. Through built-in functions, this pathfinding algorithm will result in an array of optimal actions that will be used as a comparable baseline. These results are also analyzed to build expert demonstrations to enhance learning through behavioral cloning.

4.2.1 Dijkstra's Algorithm

SciPy, the free and open-source Python library for scientific and technical computing, contains a 2-D sparse array package, *scipy.sparse*, that stores compressed sparse row matrices. The two-dimensional impassable array from the predefined gym environment, which represents obstacles through TRUE/FALSE values, is compressed from its original $M \times N$ matrix shape into a flat array of $M * N$ nodes. For example, a 20×20 matrix with a [1,1] starting point will be flattened into a 1-D array of length 400 with node ID 21 acting as the initial point. From here, all free nodes, as defined by the FALSE values of the impassable array, are analyzed for potential movement. For every selected free node, every possible motion including north [-1,0], south [1,0], west [0,-1], and east [0,1] is evaluated and, if the move results in a transition to a passable node, the information is stored in a compressed sparse row matrix. This matrix contains all allowable movements as defined by the restrictions of the given gym environment.

Following the construction of this matrix of motions, the built-in *dijkstra* function of the SciPy *csgraph* module performs Dijkstra's algorithm using Fibonacci heaps ("scipy.sparse.csgraph.dijkstra"). The index for the compressed graph is set to search the initial starting point only, performing Dijkstra's algorithm and creating a predecessor array of the shortest path. Using this exact order of nodes to travel, appropriate actions are determined by taking the difference between the coordinates of a node and its predecessor, matching the [x, y] result to an allowable movement in the M*N x M*N compressed sparse row matrix.

This results in an array of the optimal order of north, south, east, and west actions an agent should take to reach the target in the quickest amount of time.

4.2.2 On-Policy Algorithms

To implement A2C and PPO algorithms, this project utilizes the *stable_baselines3* module, which is based in PyTorch, an open-source machine learning framework. To comply with the existing functionality of this resource, the first step in training each algorithm involves wrapping the custom maze scenario in a dummy vectorized environment. This step also assists in training agents in multiple independent environments to increase training efficiency.

In training these agents, the policy, discount factor, learning rate, and entropy coefficient must be set for optimal training. For both agents, the selected policy is *MlpPolicy,* which is a multi-layer perceptron, a special deep neural network. The definitions and effect of the discount factor, learning rate, and entropy hyperparameters are defined in Chapter 3.

4.2.3 Off-Policy Algorithms

Unlike the above on-policy algorithms, the DQN agent is built using TensorFlow. Utilizing Keras, a neural network is constructed to approximate the state-value function. For this environment, the network is a sequential model with the input layer representing the observation space. The singular hidden layer uses a rectified linear unit (ReLu) activation function while the output layer has a linear activation and consists of four nodes, one for each possible movement. Note that the same neural structure is also used in the DQN's target network.

As described in Chapter 3, the DQN agent uses an epsilon greedy policy to combat the exploration-exploitation dilemma. The agent also uses the built-in *SequentialMemory* function of Keras to build a replay buffer that stores previous states, actions, and rewards. An Adam optimizer, which is based on adaptive estimation of first-order and second-order moments, is used to perform stochastic gradient descent.

4.2.4 Behavioral Cloning

Although the off-policy DQN agent proves to be successful as described below in section 4.3, this research decided to further improve the agent using behavioral cloning, a form of learning through imitation. This method uses a set of expert demonstrations to build a policy for learning agents to mimic. This pre-trained policy was loosely built by the results of Dijkstra's algorithm, which were a sequence of south and east movements, an efficient approach that never spends time moving away from the target.

Through the analysis of these results, a custom expert policy is built using the Keras *policy* class as a baseline. The unique select action feature of this policy selects a south or east action randomly, but the probability of choosing either action is dependent on the prevalence and location of this action in the optimal Dijkstra's sequence. For example, if an agent must avoid an obstacle by moving south, the expert policy is more likely to recommend this motion. The action selection feature is also instructed to make random choices to ensure the policy explores other paths in hopes of finding a sequence similar or better than the shortest path found by Dijkstra's.

After training the Dijkstra expert policy for 5000 steps to construct a functioning Q_θ network, the weights are saved and loaded in to an untrained DQN learning agent with the same parameters defined in section 4.2.3 above. Since the Q_θ network is pretrained to move towards the target by taking south and east actions, this untrained agent's primary focus is on finding the optimal sequence of these actions, rather than exploring how each motion reacts to the reward system. Therefore, the DQN learning agent requires significantly less training steps to reach convergence.

4.3 Results

As shown in the table below, some agents had to train significantly longer than others to accommodate the complexity of each maze. Note that the asterisk represents an agent that converges on a singular path resulting in the maximum 9.967 reward, as found by Dijkstra's algorithm. These training steps are in thousands, for example in maze 0, the A2C agent trained for 50,000 steps while the expert DQN trained in 1,000 steps. On average, A2C requires 65 thousand training steps to reach the goal while the successfully trained PPO agents only used about 50 thousand training steps. Both the DQN and pre-trained DQN learning agents solved each maze in approximately 35 thousand and 2.5 thousand training steps respectively.

Total Training Steps

** represents agents that successfully converged (reward = 9.967)*

	A2C	PPO	DQN	DQN w/ Expert
Maze 0	50	35*	30*	1*
Maze 1	60	40*	30*	2*
Maze 2	70	80*	40*	3*
Maze 3	80	120	35*	4*

Figure 11: Total Training Steps (in thousands)

For all four maze samples, A2C reaches the target but fails to converge while PPO can only converge for the three simpler mazes. In contrast, the DQN agent is consistent at solving every maze and even improves training efficiency when implementing behavioral cloning through Dijkstra's expert policy. Therefore, after training each agent for every maze, results indicate that the off-policy, pre-trained DQN method is most effective in terms of time and accuracy. Below are detailed descriptions and comparisons of each agent's training results.

4.3.1 A2C vs. PPO

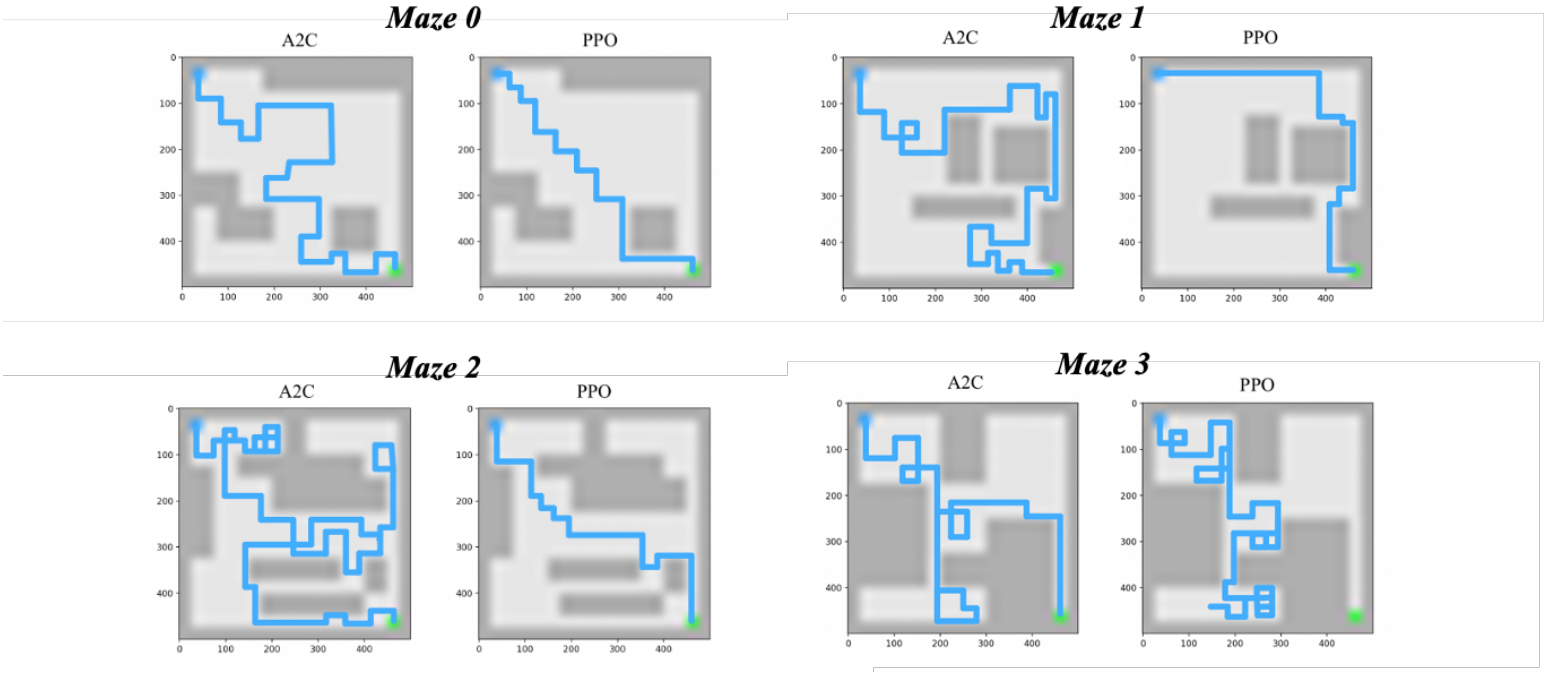


Figure 12: A2C vs. PPO

The A2C agent on the left requires at least 50K training steps before learning to move towards the goal. However, once it reaches the goal, it fails to converge on an optimal sequence, making random movements leading away from the target. This graphic shows A2C struggling to locate the end goal, even after hours of training. Therefore, this implementation and associated hyperparameters indicate that A2C is not the best method to replace pathfinding.

The other on-policy method, PPO, can solve the three easier mazes. However, the random seed had a significant effect on this ability since identical training runs produced diverging results if the target was never found in the exploratory stages. In fact, PPO was trained in parallel environments to increase the chance of the agent hitting the goal and factoring the reward into the policy. In addition, the average successful PPO agent took more than twenty minutes to train, suggesting that this method is not a suitable match for pathfinding considering the significant computational power needed for training with parallel environments.

4.3.2 DQN vs. DQN w/ Behavioral Cloning

On the other hand, the on-policy DQN is precise and computationally efficient. The agent does not require multiple parallel environments to reach convergence since the random seed has little to no effect on its ability to locate the target. In approximately 30 thousand steps, each DQN agent solves the maze without any prior knowledge of the environment by learning the consequences of hitting obstacles or wandering aimlessly around the observation space. Eventually the agent reaches the goal and iteratively builds upon this finding to construct the shortest distance between two points, something pathfinding algorithms can only do when given the destination at the start of the algorithm. Therefore, out of these three explored RL techniques, DQN is the most comparable algorithm to traditional pathfinding.

With an expert to mimic, the same results are found in an average of 2,500 training steps, taking less than a minute to converge. The Dijkstra expert of this behavioral cloning method teaches the agent how to move in terms of optimal directions to take, such as south and east. This pre-training simplifies the learning curve for the DQN agent so it only needs to optimize a route rather than learning the basic rules of the environment.

Like its successor, the pre-trained DQN agent solves each maze in 34 steps and receives a consistent 9.967 reward, converging in less than 10% of the original DQN training time. However, as seen in the graph below, these optimal paths are not the same routes that Dijkstra's algorithm found. Since these environments are 20×20 matrices, there are a limited number of solutions that reach the target. Even with the exact same reward system, two RL agents can find different paths depending on the knowledge they accumulated during training; one agent may choose to hover near obstacles while another might operate in the free space since they receive the same reward for either approach.

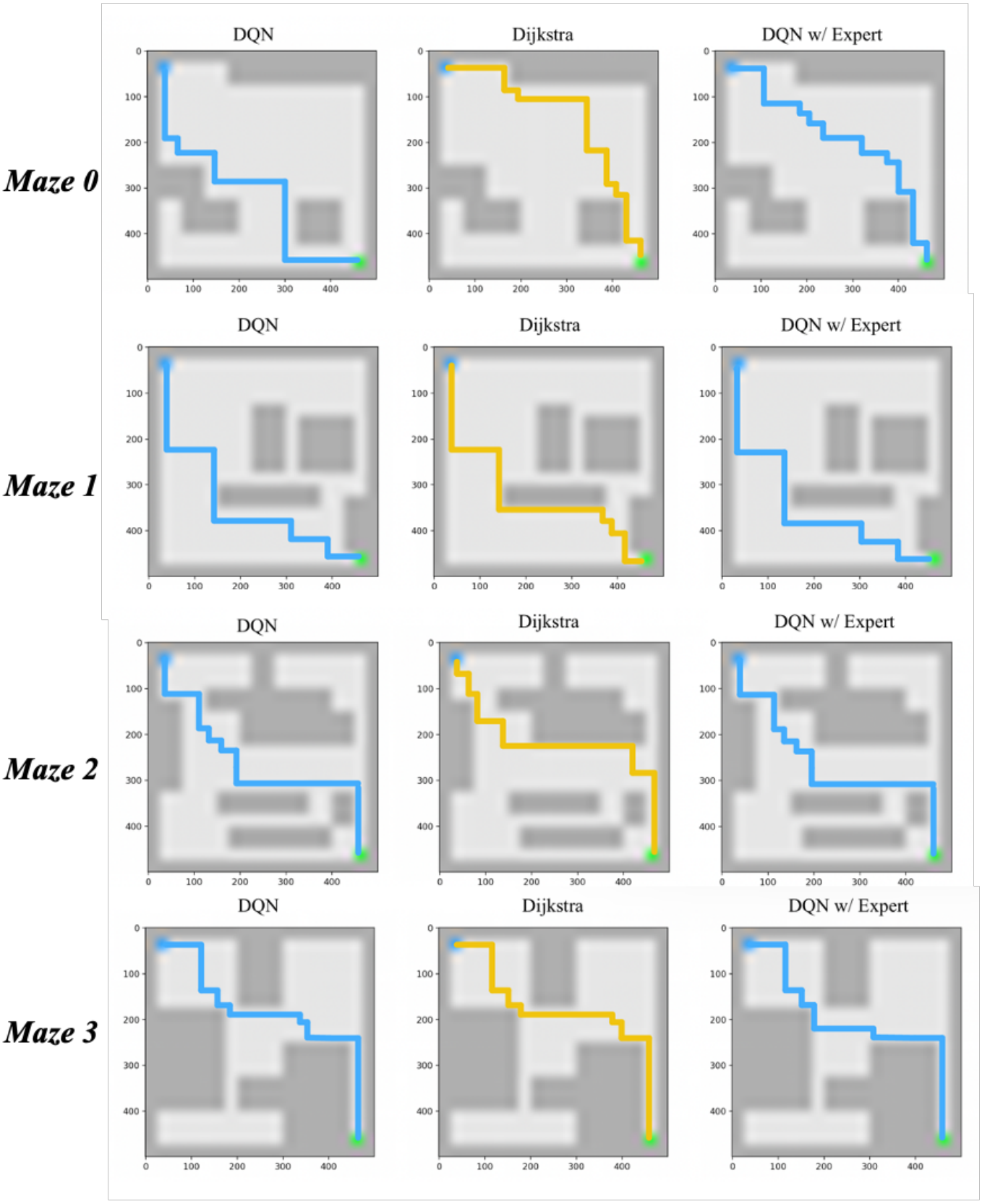


Figure 13: DQN vs. Dijkstra

5 Conclusion

While the advantage actor-critic (A2C) and proximal policy optimization (PPO) agents failed to reach convergence after tens of thousands of training steps, an off-policy deep Q-network (DQN) proved successful. This DQN agent not only solved the maze but created a route comparable to Dijkstra's algorithm, proving that reinforcement learning can blindly enter an environment, explore its surroundings, and iteratively build upon this exploration to construct the shortest distance between two points. Unlike pathfinding, these RL agents build optimal routes without even knowing of the existence, let alone the location, of the target point.

5.1 Contribution

Reinforcement learning is a great representation of true computer imagination in the field of artificial intelligence. By learning through trial and error, RL not only performs but understands what decisions will better solve the problem at hand while simultaneously making note of wrong behaviors. Similar to video games, RL agents gain rewards for completing side quests, earning the toolset necessary to complete the main quest. This robust approach showcases the innate RL ability to make intellectual decisions and establish patterns of behaviors through dynamic adaptability. Unlike traditional machine learning techniques such as supervised learning which trains through a set of correct actions, RL uses the consequences of an environment as signals to iteratively improve positive behavior. Therefore, this research showcases RL's ability to capture complex structures with minimal guidance.

In fact, when provided with additional guidance through pathfinding, RL performance only improves in terms of speed since precision remains the same. Using Dijkstra's solved maze as an expert for behavioral cloning, the DQN learning agent found an optimal route comparable

to pathfinding in terms of both training efficiency and sequence of actions. The selected actions earn the highest reward possible for the environment, similar to the learning agent acting on its own. These results indicate that using pathfinding to train RL behavior is the quintessential method for receiving the benefits of pathfinding's swift computational capabilities and reinforcement learning's ability to gather knowledge outside of the main task. This approach not only connects nodes on a graph, but also understands the rules of an environment to build the absolute shortest and safest route possible.

5.2 Limitations

Since RL agents connect to their respective neural networks through incentives and penalties, successful training relies heavily on modifications to accurately represent an environment. For example, if step sizes are too large, this leads to divergence. In fact, an agent's memory can be wiped clean since disastrous forgetting is common when new information erases knowledge the agent obtained prior.

Another common issue is reinforcement learning agents maximizing rewards without completing the task at hand. This occurs when the reward system does not account for intricate details of the environment, creating a rulebook that can never help the agent win the game. For example, if a dog never gets praised for performing a trick, it will never learn to do it on command. For the learning process of an RL agent to converge in a reasonable time, going from randomly exploring the environment to performing sophisticated tasks, the reward system must be optimized to represent the true mechanics of the environment.

Finally, the current DQN agent can be further optimized with hyperparameter tuning. The above results were achieved with standard discount factor, entropy, and epsilon parameters.

Without hyperparameter tuning, estimated model parameters generally produce suboptimal results since they are not capable of truly minimizing the loss function. Therefore, with even less errors, the agent will likely have better controls for behaviors to reach the computational speed of simple pathfinding algorithm's such as Dijkstra's.

5.3 Future Work

The greatest advantage of a reinforcement learning approach to optimal routing systems is the accelerated advancement and rapid enhancements within this learning methodology. OpenAI Gym was released in 2016, and in less than 10 years on the market, it has exponentially increased the use of RL techniques in optimizing processes, simulations, and even controls for autonomous systems related to nearly every industry including robotics, automobile, and even healthcare. Specifically in the navigation industry, machine learning is already being used to create guidance systems for people to personalize their driving experience to avoid traffic areas and accident sites. Consequently, it is no big leap to propose a walking navigation app that specifically benefits from the results of this study.

When moving to a new city, a lot of people worry about their safety, opting for convenient locations near their residence or place of work. Most people rely on navigation apps when going out of these known comfort zones to a new venue, wanting to get there in the least amount of time. However, there is no guarantee that the shortest route is the safest. Therefore, reinforcement learning can be used to build a map of safe zones and avoidance areas utilizing available sources such as criminal reports and social media posts. These RL agents will not only navigate someone to their destination, but also help them successfully avoid danger zones such as rowdy bars.

Rather than only learning to avoid impassable building obstacles, the custom environments above can expand to simulate danger zones in passable observational states. When agents receive the significant penalties, they will not only learn to avoid walking into obstacles but also avoid entering dangerous spaces. In addition, these RL agents can be pre-trained with pathfinding cloning techniques to quickly learn a baseline route, building a map of the environment to be further optimized into the shortest, safe route. With this latest and greatest machine learning technology inside a dynamic walking navigation app, users will be able to input real-time issues to help the community take evening strolls, go to museums, and eat at local restaurants without having to worry about their safety.

Appendix

Code Repository

The custom environment generation, training, and saved weights for RL policies can be found in the code repository through this link:

https://github.com/bbrown1021/UCLA-MAS-Thesis

Video of Training Results

To view the solved algorithms, visit this link:

https://drive.google.com/file/d/1Egpwi1HS9AAGzSfrF6lGIGhYzd4Bsaqc/view?usp=sharing

The video contains two parts:

1. A comparison of the optimal, successful routes of Dijkstra's algorithm as well as the original and pre-trained DQN agents.
2. A demonstration of the pathfinding algorithm and reinforcement learning agents performing in each maze environment. Note that each of the RL agents are not being trained in real-time, the videos were made using the saved trained policy weights and evaluating the agent for five episodes.

References

"API." *Gym Documentation*, https://www.gymlibrary.ml/content/api/.

"Easy Reinforcement Learning (DQN) with Keras-RL." *Linuxtut*, 14 Jan. 2017, https://www.linuxtut.com/en/e63ade6f21766c7c2393/.

"Imitation Learning." *Principles of Robot Autonomy*, Stanford University, https://web.stanford.edu/class/cs237b/pdfs/lecture/lecture_10111213.pdf.

"scipy.sparse.csgraph.dijkstra." *Scipy.sparse.csgraph.dijkstra - SciPy v1.8.0 Manual*, The SciPy Community, https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.dijkstra.html.

Argerich, Mauricio Fadel. "Tutoring Reinforcement Learning." *Medium*, Towards Data Science, 19 June 2020, https://towardsdatascience.com/tutoring-reinforcement-learning-a52186306d6d.

AurelianTactics. "PPO Hyperparameters and Ranges." *Medium*, Towards Data Science, 28 July 2018, https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe.

Bhatt, Shweta. "Reinforcement learning 101." *Medium*, Towards Data Science, 19 Apr. 2019, https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

Brockman, Greg, et al. "OpenAI/Gym." *GitHub*, OpenAI, 26 Apr. 2016, https://github.com/openai/gym/blob/master/gym/core.py.

Causevic, Siwei. “A Structural Overview of Reinforcement Learning Algorithms.” *Medium*, Towards Data Science, 13 Aug. 2021, https://towardsdatascience.com/an-overview-of-classic-reinforcement-learning-algorithms-part-1-f79c8b87e5af.

Choudhary, Ankit. “Deep Q-Learning: An Introduction to Deep Reinforcement LearningAnkit Choudhary.” *Analytics Vidhya*, 18 Apr. 2019, https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

Del Pra, Marco. “Introduction to Reinforcement Learning.” *Medium*, Towards Data Science, 2 Nov. 2020, https://towardsdatascience.com/introduction-to-reinforcement-learning-c99c8c0720ef.

Gershgorn, Dave. “Elon Musk's Artificial Intelligence Group Opens a 'Gym' to Train A.I.” *Popular Science*, 27 Apr. 2016, http://www.popsci.com/elon-musks-artificial-intelligence-group-opens-gym-to-train-ai.

Hayes, Genevieve. “The other type of machine learning.” *Medium*, Towards Data Science, 24 Dec. 2019, https://towardsdatascience.com/the-other-type-of-machine-learning-97ab81306ce9.

Huang, Chung-Yang, et al. “Dijkstra Algorithms.” *Fundamentals of Algorithms*, ScienceDirect Topics, 2009, https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms.

Jones, Tim. “Train a Software Agent to Behave Rationally with Reinforcement Learning.” *Artificial Intelligence*, IBM, 11 Oct. 2017, https://developer.ibm.com/articles/cc-reinforcement-learning-train-software-agent/.

Karunakaran, Dhanoop. “Proximal Policy Optimization(PPO)- a Policy-Based Reinforcement Learning Algorithm.” *Medium*, Intro to Artificial Intelligence, 4 Dec. 2020, https://medium.com/intro-to-artificial-intelligence/proximal-policy-optimization-ppo-a-policy-based-reinforcement-learning-algorithm-3cf126a7562d.

Markoff, John. “Artificial-Intelligence Research Center Is Founded by Silicon Valley Investors.” *The New York Times*, The New York Times, 11 Dec. 2015, https://www.nytimes.com/2015/12/12/science/artificial-intelligence-research-center-is-founded-by-silicon-valley-investors.html.

Mischel, Walter and Ebbe B. Ebbesen. “Attention in delay of gratification.” *Journal of Personality and Social Psychology* 16 (1970): 329-337.

Mnih, Volodymyr, et al. “Asynchronous Methods for Deep Reinforcement Learning.” *ArXiv*, 16 June 2016, https://arxiv.org/abs/1602.01783.

Mnih, Volodymyr, et al. “Playing Atari with Deep Reinforcement Learning.” *ArXiv*, 19 Dec. 2013, https://arxiv.org/abs/1312.5602.

N. Makariye, "Towards shortest path computation using Dijkstra algorithm," *2017 International Conference on IoT and Application (ICIOT)*, 2017, pp. 1-3, doi: 10.1109/ICIOTA.2017.8073641.

Nannapaneni, Rajasekhar. “Optimal Path Routing Using Reinforcement Learning.” *Dell EMC*, Dell Technologies, 2020, https://education.dellemc.com/content/dam/dell-emc/documents/en-us/2020KS_Nannapaneni-Optimal_path_routing_using_Reinforcement_Learning.pdf.

Plappert, Matthias. “Keras-RL: Deep Reinforcement Learning for Keras.” *GitHub*, GitHub, 2016, https://github.com/keras-rl/keras-rl.

Pocs, Mate. “Beginner's Guide to Custom Environments in OpenAI's Gym.” *Medium*, Towards Data Science, 16 Dec. 2020, https://towardsdatascience.com/beginners-guide-to-custom-environments-in-openai-s-gym-989371673952.

Raffin, Antonin, et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations.” *GitHub*, Journal of Machine Learning Research, 2021, https://github.com/DLR-RM/stable-baselines3.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*.

Richards, Hamilton. “Edsger Wybe Dijkstra.” *Edsger W. Dijkstra - A.M. Turing Award Laureate*, https://amturing.acm.org/award_winners/dijkstra_1053701.cfm.

Tanwar, Sanchit. “Bellman Equation and dynamic programming.” *Medium*, Analytics Vidhya, 7 Jan. 2022, https://medium.com/analytics-vidhya/bellman-equation-and-dynamic-programming-773ce67fc6a7.

Torres, Jordi. “The Bellman Equation.” *Medium*, Towards Data Science, 24 Sept. 2021, https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7.

Van de Kleut, Alexander. “Deep Q-Learning with Neural Networks.” *AV - Alexander Van De Kleut*, 31 May 2020, https://avandekleut.github.io/dqn/.

Van de Kleut, Alexander. “The Mathematical Foundations of Reinforcement Learning.” *AV - Alexander Van De Kleut*, 26 May 2020, https://avandekleut.github.io/q-learning/.

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, Joelle Pineau. *An Introduction to Deep Reinforcement Learning.*

Xin, Bo, et al. “Exploration Entropy for Reinforcement Learning.” *Mathematical Problems in Engineering*, Hindawi, 9 Jan. 2020, https://www.hindawi.com/journals/mpe/2020/2672537/.

Yoon, Chris. “Understanding Actor Critic Methods.” *Medium*, Towards Data Science, 17 July 2019, https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f.

Zhang, Yihao. “Deep Reinforcement Learning on 1-Layer Circuit Routing Problem.” *IDEALS @ Illinois*, 2018, https://www.ideals.illinois.edu/handle/2142/102796.

Zuo, Xingdong. “Mazelab: A Customizable Framework to Create Maze and Gridworld Environments.” *GitHub*, GitHub, 2018, https://github.com/zuoxingdong/mazelab.

Zychlinski, Shaked. “The Complete Reinforcement Learning Dictionary.” *Medium*, Towards Data Science, 24 Nov. 2019, https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e.