Jeff McCrea

CSS 595 – Term Report

Professor Fukuda

Fall 2023

ENHANCEMENT OF AGENT PERFORMANCE WITH MACHINE LEARNING - FALL TERM REPORT

# Introduction

This project seeks to further extend the development of agent movement over structured data by augmenting agent navigation with machine learning. Currently, applications developed with MASS in computational geometry and graph-based problem solving propagate agents over structured data and move them based on traditional search or heuristic-based algorithms like Ant Colony Optimization and Particle Swarm Optimization. This quarter's goal was to survey ML agent integrations for the Shortest Path Problem, benchmark the current implementation of Shortest Path in MASS Java, and design and implement ML agents to solve the Shortest Path problem in MASS Java. This paper outlines the work completed this quarter and provides a perspective on the work to be completed next quarter.

# Overview

## Agents in AI:

When looking at agents in AI systems broadly, agents comprise a program or system designed to perceive the environment it is part of, use that perception to make decisions, and use those decisions to take actions in the furtherance of some object or goal. In most cases, the agent is designed to operate autonomously without intervention from the developer.

When surveying agents in AI, five main types are commonly present in academic literature. These different agent types can be categorized based on their characteristics and how they respond to environmental stimuli. While the different agent types may have some characteristics that overlap, each agent tends to build upon the last to enhance its capabilities, most notably to make the agent more autonomous from the control of the developer or main program.

As mentioned, agents in AI can be grouped into five distinct classes based on their intelligence and capabilities:

*Simple Reflex Agents*

Simple reflex agents, as the name implies, are agents that respond only to the current environmental stimuli. The agents only know how to operate based on the simple condition-action rule that maps a state to an available action. Given this condition-action rule policy, the agents must be able to fully observe the environment to understand what actions will be optimal in a given state.

Reflex agents come with the benefit of being relatively simple to design and implement. The agents operate on an easy-to-understand condition-action policy and do not require in-depth knowledge of agent-based programming or AI systems. While the simplicity may be appealing, reflex agents do come with some significant limitations. The main limitation of simple reflex agents is that they have very limited intelligence. The agent is limited to only what it can perceive in the current environment, lacks the ability to understand anything beyond what is perceptible at a given state, and struggles to adapt to changes in the environment that are not captured in the condition-action policy.

*Model-Based Reflex Agents*

Model-Based Reflex Agents build on Simple Reflex Agents by providing a mechanism for agents to identify a condition that matches their current state. The agent keeps track of its internal state, which is updated based on its perception of the current environment as well as its individual perception history. Tracking the agent's state history allows the agent to have some insight into the part of the environment that is not currently observable. The inclusion of this mechanism allows for a more intelligent agent that can now operate in partially observable environments.

*Goal-Based Agents*

Unlike the previous two types of agents, which operate on identifying an action based on a given state, Goal-Based Agents make decisions based on how far they are from a defined goal. The agents' goal can be a single objective, like reaching an endpoint or a set of goals. The agent decides which action to take by determining which action will get it closer to its goal. The goal serves as a way for the agent to determine which action to take to reach its goal more efficiently.

*Utility-Based Agents*

Utility-based agents are designed around their end purpose and choose actions based on a utility metric for each state. While Goal-based Agents determine actions based on a desired goal, simply achieving that goal may not fully capture the complexity of the environment or the problem to be solved. The utility metric provides a means to evaluate the agent's performance in a given state. The agent selects an action that it believes will maximize its expected utility. The utility function maps a state to a real number associated with the agent's performance.

*Learning Agent*

Finally, learning agents possess the capability to learn from their past experiences or are in some way endowed with the ability to learn. Agents typically possess elementary knowledge of their environment and some baseline functionality to navigate and understand how the environment responds to given actions. This ability to learn enables the agent to adapt autonomously through its learned experiences. Learning agents are comprised of four key components shown in Figure 1:

1. Learning Element: Enables the agent to improve decision-making by learning from interactions with the environment.
2. Critic: While the learning element is responsible for improving the agent's decision-making based on past experiences, the critic is responsible for quantifying how well an agent is doing concerning a fixed performance metric.
3. Performance Element: The performance element is responsible for how the agent selects its actions. Given the feedback from the critic and the learned optimization from the learning element, the performance element selects the optimal action in the current environment.
4. Problem Generator: The problem generator determines what actions are available to an agent at any given time. In addition to selecting actions, the problem generator balances the need for an agent to explore new paths with the tendency to exploit already learned paths.



Figure 1 - Learning Agent Model [9]

*Application of AI Agents*

Intelligent agents provide a powerful tool for addressing a number of real-world scenarios. Agents can be trained to understand complex situations and leverage their learned knowledge to optimally guide systems without explicit programming. Some notable examples include:

* Robotics:
	+ Agents can be placed in an environment where they control different robotic instruments, learn how to optimally utilize those instruments to complete tasks, and help automate industrial functions.
* Smart Buildings
	+ With the rise in connected systems present in homes and commercial buildings, agents can be used to learn and understand the most efficient way to control systems like heating and cooling to deliver the maximum amount of comfort to the resident while minimizing energy costs.
* Transportation Systems:
	+ Agents can be used to learn and optimize traffic flows, plan vehicle routes, and improve logistical supply chain flow.

While these are only a few examples of where intelligent agents have been applied, they serve to illustrate the versatility and strength of these agents in solving a multitude of multidisciplinary problems.

## Reinforcement Learning

In machine learning, a common obstacle encountered is the large amount of data required to train models to solve most problems. As problems get more complex, the models also rise in complexity, requiring even more data. Even if an adequate amount of data is available, the data must be vetted to ensure it is accurate, complete, and reliable. Reinforcement Learning (RL) seeks to overcome the requirement of vast troves of data by enabling an agent to explore and learn from their environment and determine the best way to navigate through dynamic scenarios.

Reinforcement Learning is a type of machine learning where agents learn through interactions with the environment they occupy. Learning occurs through the process of trial and error, where agents are provided feedback based on their actions and the experience gleaned from taking those actions in their environment. Unlike supervised learning, where labeled examples of inputs and output are used to train a model, or unsupervised learning, which aims to discover patterns in unlabeled data, RL focuses solely on learning how agents should behave in a given environment. The ability to train intelligent agents to act autonomously in dynamic environments without full visibility presents a powerful tool for solving problems in new and efficient ways.

At a high level, RL is modeled by the natural learning processes all humans and animals utilize. As a person takes actions in life and gains experiences, they can hopefully leverage those experiences to make better decisions in the future. A common real-world illustration of this process is the age-old scenario of a child near a hot stove. While the child may not initially know that it is a bad idea to touch the hot stove if the child does take that action, they will learn rather quickly that the experience of being burned is something to avoid in the future and a good way to avoid being burned is not to touch hot items.

In reinforcement learning, the learning process involves two key components: an agent(s) and the environment. The agent represents the learning component in RL and selects actions to take within its environment, while the environment provides feedback to the agents through rewards or punishments based on its interaction. The agent's primary objective is to learn a strategy or policy that maximizes the cumulative rewards over time.

Reinforcement Learning can be formalized as a Markov Decision Process (MDP), which is a mathematical framework used to model situational decision-making in partially observable and controllable scenarios. A Markov Decision Process is characterized by several components pertinent to RL:

1. States (S): A set of unique situations that the given environment or system can take. In finite environments, the states encompass all states an environment can be in at any given timestep.
2. Actions (A): A set of possible actions or decisions that can be taken in the given environment and can vary based on the state of the environment or system.
3. Transition Probabilities (P): This represents the probability of transitioning from one state to a different state when a given action is selected.
4. Rewards (R): The numerical value of a given state-action pair. Rewards provide the agent with feedback when taking a specific action in a given state.
5. Policy (π): Represents the agent’s strategy to determine each state's optimal action.

Much like human learning, MDPs occur over a set amount of timesteps. At each time step, the agent assesses its current state, determines the action to take based on the defined policy, gains experience from the action by transitioning to the associated state, and receives a reward or punishment. The relationship between Markov Decision Processes and RL is symbiotic. Reinforcement Learning algorithms are designed to utilize the mathematical modeling of decision-making that MDPs provide and enable the creation of intelligent agents that can operate in dynamic environments.

Beyond the MDP component of Reinforcement Learning, there are several components key to the framework[1]:

1. Exploration & Exploitation: Agents in RL systems have to balance exploration, which is when the agent explores new actions to learn from, and exploitation, which is when the agent selects an action they know yields good rewards. Ensuring that exploration and exploitation are properly balanced is essential for effective learning. If exploration or exploitation are favored too heavily, agents may not find an optimal path or can fall into a local optimum.
2. Discount Factor: The discount factor represents the tendency to favor future rewards when an agent makes an action decision. When the discount factor is higher, agents will tend towards actions that maximize their long-term rewards. On the other hand, a low discount factor will encourage the agent to seek immediate rewards.
3. Value Functions: For an agent to learn from actions they take in the environment, it must be able to quantify the value of a given state-action pair. In RL, there are two common value functions: state-value functions measure the value of being in a given state, and action-value functions measure the value of taking a specific action in a given state. These value functions provide an agent with the means to compare and contrast different actions and, ideally, learn an effective policy for the environment.

Reinforcement Learning has found use in a wide range of different domains and has been proven successful in solving complex problems without requiring massive amounts of data or excessive computation time.

## Q-learning

In the field of RL, one of the more prevalent and popular algorithms is Q-learning. Q-learning is a popular algorithm mainly due to its ability to tackle scenarios that can be modeled as a Markov Decision Process and its flexibility to learn optimal policies without fully understanding all environmental dynamics.

Let’s start off by looking at a simple example of Q-learning before delving into the specifics. Consider an agent placed in a maze that must reach the end point while avoiding potential obstacles. The agent is able to move one tile in the maze at a time, and if it encounters an obstacle, it fails, and the agent instance is killed. The agent's goal is to reach the end in the shortest amount of time while avoiding potential obstacles. Now the question is how do we solve this problem without explicitly programming all the environmental details? Here enters q-learning.

Q-learning is a model-free RL algorithm used to find the optimal agent policy for a given finite MDP [4]. In the context of Q-learning, model-free is defined as an algorithm that doesn’t estimate the transition probably associated with the MDP. More simply, model-free algorithms in RL can be thought of as pure trial-and-error algorithms. Given its model-free nature, Q-learning is commonly utilized in situations where the environment isn’t fully observable and must be learned through trial-and-error exploration of the environment.

Q-learning is an iterative process that allows an agent to learn and improve over time by learning which actions lead to a more optimal action selection policy. The Q-learning algorithm is made up of six components similar to those of an MDP:

* Agents: The entity that takes actions and operates within a given environment
* States (S): All the available states or configurations the agent may encounter. States are the main input to the Q-learning algorithm.
* Actions (A): All available actions that an agent can take in a given state. As mentioned, Q-learning aims to learn a Q-value for each state-action pair.
* Q-Value: The q-value represents the value of taking an action in a given state. In most cases, Q-values are updated during the learning process.
* Bellman Equation: To update the Q-value, the Bellman equation, shown below, is typically used:

Q(s,a) ← Q(s,a) +α[r + γmaxa ​Q(s′,a)−Q(s,a)]

 Where:

* r is the immediate reward received after taking action a in state s
* s’ is the next stage
* maxa ​Q(s′,a) represents the estimated maximum future Q-value for the next state.
* α represents the learning rate. This influences the preference towards new information versus already learning information.
* γ is the discount factor and controls the agent’s tendency to immediate rewards over future rewards.
* Q Table: The Q table is one of the elements that differentiates Q-learning from other RL algorithms. The Q table is a matrix where each entry of a given state and action represents an estimated Q-value for taking action a in state s. When the Q-table is created, values are typically initialized to zero or other random non-zero values.

Now that we have established the baseline operation and components in Q-learning, let’s jump back to the agent in a maze example; how can Q-learning be used to allow the agent to learn to navigate the maze efficiently?

The first step is to initialize the Q-table, which is the m x n matrix where n is the number of actions and m is the number of states. Initially, the agent has no knowledge of the q-value of each state-action pair, so all values state at zero, as shown inFigure 2**.**

Next, at each timestep, the agent has to choose an action in its given state based on the Q-table. Currently, the Q-table has all 0 values, which is where the exploration and exploitation concept is utilized. While there are several ways to balance exploration, the most common approach is the epsilon greedy strategy. At inception, the exploration value will be higher, causing the agent to tend to explore the environment randomly. As the agent explores, the epsilon value decreases, and the agent tends more toward exploiting the information in the Q-table. As the agent moves, the Q-value is updated based on the experiences encountered by the agent based on the result from the Bellman equation.



Figure 2 - Q-table Navigation [10]

Each training instance where the agent completes a path, either successfully or unsuccessfully, is known as an episode. Agents are trained over numerous episodes until the number of specified episodes is reached or the total q-value for a given path converges.

## RL SP in MASS

## System Design

*Environment*

In Q-learning Shortest Path, the network's different vertices naturally form a graph. In MASS Java, students have employed a number of different techniques to represent distributed graphs within the MASS framework. These graphs can simply extend the Place base class and implement their own graph structure over the distributed place array. While that may be a simple way to proceed, it fails to take advantage of new MASS features.

The most optimal solution is to utilize the *GraphPlaces*() class to read and store the graph data in parallel across the cluster. The first step to creating the MASS Java graph is to define a Node class that extends the *VetertexPlace*() class. The VertexPlace() class contains a number of different graph-based functions that make working with graphs much more intuitive and saves the developer from having to implement the functionality on their own.

Once we have defined our Node class, we can create the initial *GraphPlaces* object to store our nodes. The graphplaces object is created with the command *GraphPlaces network = new GraphPlaces(1, Node.class.getName()).* The current version of MASS SP utilizes graphs randomly generated on the fly, but for my project, the graph data will largely be imported from a file. While data reading can be done sequentially by the master node and distributed based on a user-defined scheme, students have previously tackled the problem and created a parallel file reading method. The GraphPlaces class has the ability to load graphs from a number of different file formats, including MATSIM XML, HIPPIPE, SAR, and DSL [7]. The DSL format is a modified CSV style format, where each line represents a given node, its neighbors, and the weight separated by a semicolon; an example graph is shown in Figure 3. Due to the fact that each line represents a different node, each line can be read in parallel by each computing node and enables the graph to be distributed evenly across all the computing nodes. By invoking the GraphPlaces *loadDSLFile*() method and providing the file path to the DSL file, we are able to create a distributed graph efficiently.



*Agent*

The *RLAgent* class is designed to navigate autonomously over the distributed graph, calculate the Q-value for each encountered state and action, and populate the Q-table based on the Q-values for a given route.

To create the agent, we extend the *SmartAgent* class and track the Q-table values and path taken by the agent within the RLAgent class. Extending the SmartAgent class allows us to access some of the autonomous agent navigation methods created by former student Vishnu Mohan [6]. Vishnu added functionality, including the migrateRandom() function, which enables the autonomous RLAgent to migrate randomly when initially exploring the graph[6]. In addition, the *SmartAgent* class contains other useful methods for finding the minimum and maximum weight nodes and getting neighbors and their weights, among other helpful methods.

In the initial single-agent Q-learning design, the agent is spawned on the initial source node specified in the main program and traverses the graph, at first random and then based on the Q-table to find the shortest route from a given source node to the destination node. If an agent encounters an invalid, the agent instance is terminated and respawns at the source node to complete another training episode.

# Goals

During the winter quarter, I had five goals I hoped to achieve. The motivation behind the goals set for this quarter was to gain an understanding of the current state-of-the-art in machine learning agents for the Shortest Path Problem, identify suitable ML frameworks for implementation in MASS, retest the current MASS Shortest Path implementation, and compare the performance of the newly implemented ML agents. The goals for this quarter were as follows:

* Survey agent-based machine learning implementations for solving the Shortest Path Problem
* Generate new benchmarks for the previous strings-and-pins Shortest Path implementation currently present in the MASS Java benchmark applications
* Design an ML system for use in MASS JAVA
* Implement and train ML agents to complete the Shortest Path Problem
* Benchmark ML agent performance against the current MASS JAVA SP implementation

# Achievements

While this quarter may not have been as productive as I would have liked, there was some notable progress made that should aid in completing the remaining work from this quarter and setting the following two quarters to be more productive.

*Survey Agent-based Machine Learning Shortest Path*

One of my primary goals for this quarter was to identify a suitable machine-learning framework that would enable intelligent agents to identify and produce the shortest path in a given weighted graph more efficiently. While I have some background and experience in machine learning, prior to starting this project, I had little familiarity with how agents, coupled with machine learning algorithms, could be used to solve problems more efficiently. The survey allowed me to understand what others have completed, how that work could be adapted into MASS Java, and how that integration could produce more effective and efficient agents.

*Generate New Strings-and-Pins Shortest Path Benchmarks*

In order to gauge the performance of the ML agents in MASS Java, it was necessary to identify the current performance of the already implemented MASS Java SP. While benchmarks for the current MASS SP implementation have been generated previously, MASS has undergone significant development since the original benchmarks were run. The benchmarks not only provide a means to evaluate the performance of ML agents but also provide an insight into how changes in MASS Java have affected the performance of the MASS SP implementation.

To generate the benchmarks for MASS SP, I first identified and cloned the most recently updated version of the program. Initially, I ran into some issues running the program as I was using the wrong version, but after Professor Fukuda's assistance, I was able to find the correct version. The previous benchmark measured performance on MASS 1Node with 1-4 threads and MASS 1-8 nodes with a single thread with the following number of vertices: 125, 250, 500, 1000, 2000, 4000, 8000, 16000.

To complete the benchmarking efficiently, I developed a runner program that given an input, ran the desired test 10 times, and saved the output in a text file. Utilizing the runner program allowed me to streamline the testing procedure while the multiple iterations ensured that there were multiple runs to average off of. The initial benchmarking was done purely on randomly generated graphs provided by the built-in graph generator.

*Design an ML system for use in MASS Java*

Beyond the survey and benchmark work, one of my key goals was to identify and design an applicable ML framework to create and train intelligent agents. Upon surveying work on ML agents for the shortest path, I found that Q-learning-based reinforcement learning was the most commonly used framework.

My initial design centered on exploiting MASS Java's power to store large distributed datasets, like graphs. Many of the works I surveyed focused on a single-node system and seemed to exclude testing on large-scale datasets. While many of these papers did a good job outlining how Q-learning works, they lacked the validation of real large-scale graph data sets. Real-world city graphs can encompass 100,000+ nodes, rendering many of these systems incapable of storing and processing such large graphs.

While the design has yet to be fully implemented, the understanding I gained from my initial foray into implementing my design in MASS has prepared me to complete the project and to have an easier implementation of the other applications.

# Results

*SP Benchmarking Results*

Understanding how the current implementation of MASS SP performs is an important element in developing an effective RL-based solution. As illustrated by the 4-node results below, MASS SP utilizing MASS-core 1.4.3 still performs fairly well but is significantly slower than the earlier MASS-core 1.0 version.

The decrease in execution performance is most likely due to the improvements made on MASS-core over the years to include more features, data members, and other help inclusions that aid in easing development and making the system consistent. The benchmark results are consistent with former student Caroline Tsui's results when she benchmarked other MASS Java graph programs [2].



*RL SP Benchmarking Results*

While one of the goals was to complete and benchmark the RL SP implementation and compare it against the current SP benchmarks, I was not able to complete the implementation in time to generate the results. In the following plan section, I outlined my plan to complete the implementation over the winter quarter, benchmark the program, and update this paper.

# Next quarter's plan

**Winter Break**

In light of the remaining work that is left unfinished, I plan to complete the work on the RL SP implementation over the winter break in preparation for next quarter’s presentation in winter week 3. The schedule for winter break is as follows:

Week 1 (12/15 – 12/22):

*Tasks:*

* Identify and mitigate impediments to agent graph navigation in MASS Java
* Add Q-learning migration method
* Add distributed Q-table into place class
* Complete initial testing

*Deliverables:*

* Initial working version of Q-learning-based single agent navigation
* Small graph benchmarks

Week 2 (12/22 – 12/29):

*Tasks:*

* Debug any remaining issues with Q-learning-based single-agent navigation
* Complete testing on all graph sizes
* Test on real-world graph
* Begin work on multi-agent Q-learning-based navigation

*Deliverables:*

* Completed and tested version of single-agent Q-learning-based SP
* Completed benchmark on all graph sizes
* Comparison between MASS JAVA SP and MASS JAVA Q-learning SP
* Updated term report to include completed remaining work submitted to Professor Fukuda

Week 3 (12/29 – 1/5)

*Tasks:*

* Debug and test initial version of multi-agent Q-learning-based navigation
* Evaluate initial performance of multi-agent Q-learning-based navigation
* Complete benchmarking on multi-agent Q-learning
* Begin survey on Euclidean Shortest Path ML implementations

*Deliverables:*

* Completed initial version of multi-agent Q-learning-based SP
* Minimum:
	+ Initial benchmark on multi-agent Q-learning-based SP
* Ideal:
	+ Complete benchmarks on multi-agent Q-learning-based SP

**Winter Quarter:**

Week 0 (1/3 – 1/10):

*Tasks:*

* Begin survey on Euclidean Shortest Path ML implementations

*Deliverables: N/A*

Week 1 (1/10 – 1/17):

*Tasks:*

* Verify functionality of MASS Java Euclidean Shortest Path
* Benchmark ESP implementation
* Complete ML ESP survey and discuss with Professor Fukuda
* Start presentation slide deck

*Deliverables:*

* MASS ESP benchmark
* ESP ML Survey

Week 2 (1/17 – 1/24):

*Tasks:*

* Project Presentation on RL SP results and perspective for the upcoming quarter
* Being ESP ML Agent Outline

*Deliverables:*

* Presentation slide deck
* Presentation

Week 3 (1/24 – 1/31):

*Tasks:*

* Discuss ESP ML agent outline with Professor Fukuda
* Being ESP ML Agent implementation

*Deliverables:*

* ESP ML Agent outline

Week 4 (1/31 – 2/7):

*Tasks:*

* Verify functionality and correctness of ESP ML Agent
* Generate initial ML agent performance benchmarks
* Debug and tune ML agent as needed

*Deliverables:*

* Initial ESP ML agent implementation
* Initial performance statics on ESP ML Agent
* Discussion with Professor Fukuda on performance and other considerations

Week 5 (2/7 – 2/14):

*Tasks:*

* As needed, tune ML agent to optimize performance and accuracy

*Deliverables:*

* Updated performance benchmarks

Week 6 (2/14 – 2/21):

*Tasks:*

* Generate initial performance comparison with current MASS Java ESP implementation
* Tune ML agent based on results

*Deliverables:*

* MASS Java ESP and ML agent ESP performance comparison

Week 7 (2/21 – 2/28):

*Tasks:*

* Finalize ESP ML agent implementation
* Fully comment and document code before uploading to BitBucket
* Complete ML ESP agent documentation to include in the repository

*Deliverables:*

* ML ESP agent implementation
* Documented and uploaded code to BitBucket

Week 8 (2/28 – 3/6):

*Tasks:*

* Generate final performance statistics
* Begin term report

*Deliverables:*

* Final performance statistics; discuss with Professor Fukuda

Week 9 (3/6 – 3/15):

*Tasks:*

* Complete term report

*Deliverables:*

* Term report

# Conclusion

The work completed over this quarter has enabled me to better understand ML agent implementations, specifically Q-learning-based implementations, and how agents, coupled with ML, can be used to solve a variety of common computer science problems. While I was not able to complete all my goals set for this quarter at the time of writing this paper, I feel that the work completed has laid a solid foundation for completing the remainder over the winter break and for a more productive winter quarter. I look forward to continuing work on this project and working with Professor Fukuda and the DSL group.

[1]

“3.5 The Markov Property.” Accessed: Dec. 13, 2023. [Online]. Available: <http://www.incompleteideas.net/book/ebook/node32.html>

[2]

Y. Cui and C. Tsui, “Agent-based Graph Applications in MASS Java and Comparison with Spark”.

[3]

D. Blashaw and M. Fukuda, “An Interactive Environment to Support Agent-based Graph Programming (Draft Paper)”.

[4]

“An introduction to Q-Learning: reinforcement learning,” freeCodeCamp.org. Accessed: Dec. 13, 2023. [Online]. Available: <https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/>

[5]

Z. Sun, “Applying Reinforcement Learning for Shortest Path Problem,” in *2022 International Conference on Big Data, Information and Computer Network (BDICN)*, Jan. 2022, pp. 514–518. doi: [10.1109/BDICN55575.2022.00100](https://doi.org/10.1109/BDICN55575.2022.00100).

[6]

V. Mohan, “Automated Agent Migration Over Structured Data”.

[7]

B. Luger and D. M. Fukuda, “Distributed Data Structures”.

[8]

X. Wang, L. Jin, and H. Wei, “The Shortest Path Planning Based on Reinforcement Learning,” *J. Phys.: Conf. Ser.*, vol. 1584, no. 1, p. 012006, Jul. 2020, doi: [10.1088/1742-6596/1584/1/012006](https://doi.org/10.1088/1742-6596/1584/1/012006).

[9]

“What is Q-Learning: Everything you Need to Know | Simplilearn,” Simplilearn.com. Accessed: Dec. 13, 2023. [Online]. Available: <https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-q-learning>

[10]

“What is Q-learning?,” Enterprise AI. Accessed: Dec. 13, 2023. [Online]. Available: <https://www.techtarget.com/searchenterpriseai/definition/Q-learning>