Project ID: 24/CSE-AIML/1/103

iPark: Intelligent Parking

A PROJECT REPORT
Submitted By

Kushagra
(2000271530029)
Abhinav Sajjan
(2000271530006)
Himanshu Mishra
(2000271530024)
Shreyashi Jaiswal
(2000271530056)

Under the guidance of
Mr Samender Singh

Submitted in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Technology
in
Computer Science and Engineering with
Artificial Intelligence and Machine Learning

to

Department of Computer Science & Engineering
AJAY KUMAR GARG ENGINEERING COLLEGE, GHAZIABAD
DR. A. P. J. ABDUL KALAM TECHNICAL UNIVERSITY, LUCKNOW
May 24, 2024

DECLARATION

We hereby declare that the work presented in this report entitled "iPark: Intelligent Parking", was carried out by us. We have not submitted the matter embodied in this report for the award of any other degree or diploma from any other University or Institute. I have given due credit to the original authors/sources for all the words, ideas, diagrams, graphics, computer programs, experiments, and results, that are not my original contribution. I have used quotation marks to identify verbatim sentences and given credit to the original authors/sources.

I affirm that no portion of my work is plagiarized, and the experiments and results reported in the report are not manipulated. In the event of a complaint of plagiarism and the manipulation of the experiments and results, I shall be fully responsible and answerable.

Name : Kushagra
Roll No. : 2000271530029

Name : Abhinav Sajjan
Roll No. : 2000271530006

Name : Himanshu Mishra
Roll No. : 2000271530024

Name : Shreyashi Jaiswal
Roll No. : 2000271530056

CERTIFICATE

Certified that **Kushagra** (2000 27153 0029), **Abhinav Sajjan** (2000 27153 0006), **Himanshu Mishra** (2000 27153 0024), and **Shreyashi Jaiswal** (2000 27153 0056) has carried out the Project entitled **"iPark: Intelligent Parking"** for the award of **Bachelor Of Technology (B. Tech)** from **Dr. A. P. J. Abdul Kalam Technical University (AKTU), Lucknow** under my supervision. The project embodies the results of original work, the students and the contents carry out studies do not form the basis for awarding any other degree to the candidate or anybody else from this or any other University/Institution.

Mr Samender Singh
Assistant Professor
Department of Computer
Science & Engineering
AKG Engineering College, Ghaziabad

Dr Avdhesh Gupta
Professor Incharge (PI), CSE-AIML
Department of Computer
Science & Engineering
AKG Engineering College, Ghaziabad

Date: May 24, 2024

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout the development of this project.

Firstly, I would like to thank my project guide, Mr Samender Singh, for his invaluable guidance, patience, and encouragement. His insights and suggestions have been crucial to the success of this project. I extend my heartfelt thanks to Dr Avdhesh Gupta, our professor in charge, for providing the necessary resources and support. His expertise and feedback have been instrumental in shaping the direction and quality of this work.

My sincere appreciation goes to my friends and family for their unwavering support and encouragement throughout this journey. Their understanding and patience have been a constant source of motivation. Additionally, I would like to acknowledge the online communities and forums dedicated to machine learning and game development. The shared knowledge and collaborative spirit of these communities have been immensely helpful.

Lastly, I am grateful to God for providing me with the strength and perseverance to complete this project.

Thank you all for your continuous support and encouragement.

Abstract

The constant growth in the Artificial Intelligence and Machine Learning field is changing our daily lives in countless ways. The action of driving is one of the examples, and the redundant task of parking is a waste of our useful time. This project describes how RL agents in the Unity Environment can perform autonomous parking. The goal is to propose a method that uses reinforcement learning techniques offered by the Unity ML-Agents framework within Unity's realistic 3D simulation to solve the requirement for autonomous parking solutions. The suggested solution's design, execution, and assessment are highlighted in the paper. The system offers an adaptive and realistic system for autonomous parking in complex situations. The outcomes of thorough performance testing and comparative analysis highlight the usefulness and promise of the suggested approach in the area of autonomous car parking. Discussing the results, difficulties faced, and prospects for additional study and advancement in autonomous car parking technology round up the report.

Keywords: *Unity ML-Agents, Unity Game Engine, Autonomous Parking System, Reinforcement Learning*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The fusion of artificial intelligence and virtual simulation has ushered in a new era of technological advancements. Within this paradigm, Machine Learning (ML) and, more specifically, Reinforcement Learning (RL), have emerged as transformative tools for training intelligent agents to autonomously perform intricate tasks. This project focuses on harnessing the capabilities of RL within the Unity simulation environment to address the challenging domain of automated car parking.

Unity, with its robust physics engine and realistic rendering, provides an ideal platform for creating simulated environments that closely mimic real-world challenges. As autonomous vehicle technology progresses, the development of intelligent parking systems becomes crucial. This project leverages Unity's capabilities to create a simulated environment where a virtual car equipped with an RL agent can learn to navigate diverse parking scenarios.

Through a comprehensive analysis of the proposed system's design, implementation, and evaluation, this project aims to shed light on the potential of RL-based autonomous parking simulation as a viable approach for addressing the challenges of autonomous vehicle development. Furthermore, the findings and methodologies presented in this project may pave the way for future advancements in autonomous vehicle training, with broader applications in other domains of autonomous systems.

1.2 Identification of the Problem

Automated car parking represents a multifaceted challenge in the field of autonomous vehicle control. The intricacies of spatial awareness, trajectory planning, and real-time

decision-making.

These challenges combined are so significant that a traditional algorithm can not overcome them. This project identifies this problem and aims to train an RL agent to navigate and park a car autonomously, recognizing the dynamic and complex nature of the parking situation. The main reason to opt for the RL technique over other sorts of algorithms, and the Unity Environment is due to the facilities provided, we will take a closer look at the Unity Engine, and the Unity ML-Agents framework (including RL tools provided) in further sections.

The project seeks to address fundamental questions such as:

- How can we construct a system capable of parking a vehicle on its own?
- What factors contribute to successful navigation in various parking scenarios?
- and How efficient will the system be?

By honing in on these challenges, the project aims to contribute to the development of robust and adaptive RL models capable of handling the nuances associated with automated car parking.

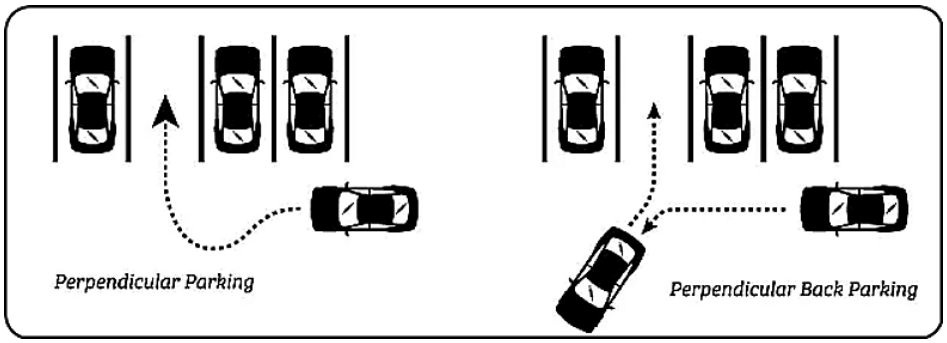

Figure 1.1: Identification of the Parking Problem

1.3 Significance of the Problem

The significance of this project lies in its direct relevance to the advancement of autonomous vehicles. Automated parking systems, an integral part of the broader self-driving technology, demand intelligent agents capable of making swift and accurate decisions in real time. By employing RL techniques, this project endeavours to create a model that not only learns optimal parking strategies but also adapts to diverse scenarios, showcasing the adaptability necessary for real-world applications.

As autonomous vehicles become an increasingly viable mode of transportation, the ability to navigate and park autonomously becomes a defining factor in their widespread adoption. This project addresses the significance of the problem by contributing to the growing body of knowledge surrounding ML applications in simulated environments, with a specific focus on enhancing the capabilities of RL agents in the context of automated car parking.

1.4 Scope of the Project

The scope of this project encompasses the development of an autonomous car parking system using Reinforcement Learning (RL) within the Unity simulation environment. The primary focus is on training a virtual agent to autonomously navigate and park a car in diverse scenarios, emulating real-world challenges. The system will address various aspects of automated parking, including spatial awareness, trajectory planning, and real-time decision-making.

The simulation environment, powered by Unity, provides a flexible and realistic platform for creating parking scenarios that mimic the complexities of urban environments. The scope extends to the exploration of RL algorithms to optimize the learning process, with an emphasis on adaptability and efficiency.

1.5 Project Objectives

Objective 1. Develop an RL model tailored for car parking in Unity, integrating state-of-the-art algorithms to enable effective learning and decision-making.

Objective 2. Design and implement a simulation environment within Unity that encompasses a range of parking scenarios, capturing the complexities of real-world parking challenges.

Objective 3. Train the RL agent to navigate and park a virtual car autonomously, emphasizing adaptability to different parking space configurations and dynamic environments.

Objective 4. Evaluate the performance of the trained RL agent based on key metrics, including success rate, parking accuracy, and computational efficiency.

Objective 5. Contribute insights to the broader field of ML applications in simulated environments, offering solutions and methodologies for training RL agents in complex tasks, specifically in the context of automated car parking.

Figure 1.2: Parking scenario simulated in the Unity Engine

1.6 Outline of the Report

This document outlines the key aspects of the projects and serves as a detailed and essential guide for the "iPark: Intelligent Parking" project. Its overarching purpose is to provide comprehensive insights into the intricacies of the system, offering a roadmap for developers, stakeholders, and all involved parties.

Key Aspects:

1. System Architecture: Detailed exploration of the architectural framework, highlighting the interplay between components, modules, and their respective functionalities.

2. Functionalities: In-depth coverage of the system's functionalities, elucidating how each component contributes to the overall autonomy and efficiency of the car parking system.

3. Design Strategies: An overview of the design strategies employed, focusing on how Unity's simulation environment and Reinforcement Learning techniques synergise to create an intelligent parking model.

4. Development Guidelines: A reference for developers, providing guidelines and insights into the coding, implementation, and integration processes crucial for the successful realization of the system.

5. Decision-Making Reference: Facilitation of informed decision-making among stakeholders by offering a clear understanding of design choices, methodologies, and anticipated outcomes.

By addressing these key aspects, this document aims to be a foundational resource, ensuring clarity, alignment, and efficient display of information on the autonomous car parking system.

Chapter 2

Literature Review

The rapid advancements in Artificial Intelligence (AI) and Machine Learning (ML) have significantly impacted various domains, leading to innovative solutions and enhanced capabilities. This literature review aims to provide a comprehensive overview of the key areas of research relevant to our project on autonomous parking systems. We will explore the historical evolution and foundational concepts of AI, ML, and Reinforcement Learning (RL), examining their applications in autonomous systems.

Additionally, we will review existing studies on autonomous parking, highlighting the challenges and solutions proposed in the literature. Furthermore, we will discuss the tools and frameworks utilized in our project, such as Unity3D and Unity ML-Agents, to contextualize our work within the broader field of intelligent parking systems. This review will serve to underscore the significance of our research and the contributions it makes to the ongoing advancements in autonomous vehicle technology.

2.1 Research Papers Review

2.1.1 Unity: A General Platform for Intelligent Agents by Arthur Juliani et al.

Arthur Juliani and his team introduce Unity, a versatile platform designed for developing and evaluating intelligent agents. Unity provides a rich simulation environment that supports a wide range of applications, from robotics to video games. The platform's integration with the Unity3D engine allows for the creation of complex, interactive environments that are crucial for training and testing intelligent agents.

For our project, the Unity platform serves as the backbone for developing and testing our autonomous parking agent. The flexibility and capabilities of Unity, as described

by Juliani et al., enable us to create a realistic and dynamic simulation environment. The paper highlights key features such as the ease of integrating machine learning frameworks and the support for multi-agent systems, both of which are instrumental in achieving our project goals (Juliani et al., 2018).

2.1.2 Quantifying Generalization in Reinforcement Learning by Karl Cobbe et al.

Karl Cobbe and his team's paper focuses on enhancing the robustness of reinforcement learning algorithms through procedural generation. They propose a novel benchmark, Procgen, which generates a diverse set of environments to evaluate RL agents' generalization capabilities. The core idea is to expose agents to a wide variety of scenarios during training, thereby improving their adaptability and performance in unseen environments.

The relevance of this paper to our project cannot be overstated. The use of procedural generation can significantly enhance the robustness of our parking agent by exposing it to numerous parking scenarios, including edge cases that are difficult to anticipate. By integrating procedural generation into our training pipeline, we can ensure that our agent is not only proficient in common parking situations but also adaptable to rare and complex scenarios (Cobbe et al., 2020).

2.1.3 Proximal Policy Optimization Algorithms by John Schulman et al.

John Schulman and his co-authors introduce the Proximal Policy Optimization (PPO) algorithm, a breakthrough in the field of reinforcement learning. PPO strikes a balance between complexity and performance, offering a simpler yet highly effective alternative to algorithms like Trust Region Policy Optimization (TRPO). The key innovation of PPO lies in its use of a clipped surrogate objective, which stabilizes training and enhances the reliability of policy updates.

This paper is particularly relevant to our project, as PPO is one of the primary algorithms we have utilized for training our parking agent. The algorithm's ability to provide stable and efficient learning is crucial for developing a robust autonomous parking system. The detailed mathematical foundations and empirical results presented by Schulman et al. underscore the algorithm's superiority in handling various RL tasks, making it an ideal choice for our application (Schulman et al., 2017).

2.1.4 Comparative Study of SAC and PPO in Multi-Agent Reinforcement Learning Using Unity ML-Agents by Andres Bayona et al.

Andres Bayona and his colleagues explore the application of adversarial reinforcement learning in urban driving scenarios. The paper presents a framework where an adversarial agent introduces perturbations in the driving environment, challenging the autonomous driving agent to adapt and improve its robustness. This approach ensures that the driving agent can handle a variety of unexpected situations, enhancing its reliability in real-world applications.

The concept of adversarial reinforcement learning is highly relevant to our project, as it can be applied to create challenging parking scenarios that test the limits of our autonomous parking agent. By incorporating adversarial elements, we can ensure that our agent is not only proficient in standard parking tasks but also resilient to unpredictable changes in the environment (Bayona et al., 2021).

2.1.5 Playing FPS Games with Deep Reinforcement Learning by Guillaume Lample and Devendra Singh Chaplot

Guillaume Lample and Devendra Singh Chaplot explore the application of deep reinforcement learning (DRL) in complex environments by training agents to play first-person shooter (FPS) games. Their work highlights the ability of DRL to handle high-dimensional input spaces and make real-time decisions in dynamic and adversarial settings. The authors utilized a combination of convolutional neural networks (CNNs) and recurrent neural networks (RNNs) to enable the agent to perceive the environment, plan actions, and adapt to changing scenarios.

The significance of this paper lies in its demonstration of DRL's capability to learn from raw sensory inputs, a critical aspect for developing autonomous agents in various fields, including robotics and gaming. The challenges addressed, such as partial observability and long-term planning, are directly relevant to our project's goal of developing a sophisticated parking agent. The techniques described can be adapted to improve our simulation environment's realism and the agent's decision-making processes (Lample & Chaplot, 2020).

2.1.6 Intelligent Parking System: A Reinforcement Learning Approach by Fethi Filali et al.

The paper by Fethi Filali and colleagues presents an innovative approach to addressing urban parking challenges using reinforcement learning (RL) techniques. The authors propose a smart parking system that leverages RL to optimize the allocation of parking spaces in real-time, thus enhancing the efficiency of urban mobility. The system utilizes historical parking data, real-time occupancy information, and predictive analytics to manage parking space availability dynamically. The RL model employed is designed to learn optimal policies that minimize the time drivers spend searching for parking, thereby reducing traffic congestion and associated emissions.

One of the key strengths of this study is its practical implementation and validation. The authors conducted extensive simulations using real-world data from a large urban area, demonstrating the effectiveness of their approach in reducing average parking search times by significant margins. The results indicate that the RL-based system outperforms traditional static allocation methods, showcasing the potential of intelligent systems in urban planning.

In the context of our project, this paper provides valuable insights into the application of RL for parking management, reinforcing the feasibility of using RL in our own autonomous parking system. The use of real-time data and predictive analytics aligns with our approach to creating a dynamic parking environment (Filali et al., 2023).

Chapter 3

Tools & Technologies Utilised

3.1 Artificial Intelligence (AI)

Artificial Intelligence (AI) represents a dynamic field of computer science dedicated to developing systems that can perform tasks requiring human-like intelligence. With advancements in machine learning algorithms, increased computational power, and the availability of vast datasets, AI has rapidly evolved, leading to transformative applications across various domains. This section provides a comprehensive overview of AI, encompassing its historical development, key concepts, applications, and future prospects.

Historical Evolution of AI

The roots of AI can be traced back to ancient civilizations, where myths and legends depicted artificial beings with human-like attributes. However, the modern era of AI began in the 1950s, marked by seminal contributions from pioneers like Alan Turing and John McCarthy. Turing introduced the concept of machine intelligence through the Turing Test, while McCarthy coined the term "artificial intelligence" and organized the Dartmouth Conference in 1956, marking the formal inception of the field. Since then, AI research has progressed through various stages, including symbolic AI, connectionism, and the emergence of machine learning paradigms.

Key Concepts in AI

AI encompasses a diverse range of concepts and techniques, including:

- **Machine Learning (ML):** ML focuses on developing algorithms that can learn from data and make predictions or decisions without explicit programming. Su-

pervised learning, unsupervised learning, and reinforcement learning are fundamental paradigms within ML.

- **Deep Learning:** Deep learning employs artificial neural networks with multiple layers to learn complex patterns in data. Its success in tasks such as image recognition, natural language processing, and speech recognition has propelled AI to new heights.
- **Natural Language Processing (NLP):** NLP enables computers to understand, interpret, and generate human language, facilitating applications like machine translation, sentiment analysis, and chat-bots.
- **Computer Vision:** Computer vision enables machines to interpret and understand visual information from the real world, powering applications like image recognition, object detection, and video analysis.
- **Robotics:** Robotics integrates AI with mechanical engineering to create intelligent machines capable of performing physical tasks autonomously or semi-autonomously.

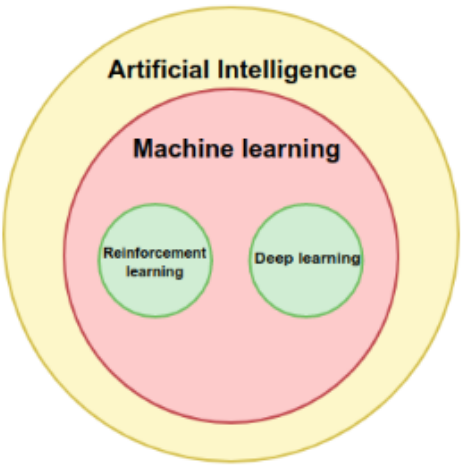

Figure 3.1: AI Domains and Concepts

Applications of AI

AI technologies find applications across diverse domains, including:

- **Healthcare:** AI aids in medical diagnosis, personalized treatment planning, drug discovery, and patient monitoring.
- **Finance:** AI powers algorithmic trading, fraud detection, risk assessment, and customer service chat-bots in the financial sector.

- **Transportation:** Autonomous vehicles, traffic management systems, and predictive maintenance leverage AI technologies to enhance safety and efficiency.
- **Retail:** AI-driven recommendation systems, inventory management, supply chain optimization, and cashier-less stores optimize operations in the retail industry.
- **Education:** AI-enabled adaptive learning platforms, intelligent tutoring systems, and educational games offer personalized learning experiences for students.
- **Entertainment:** AI-driven content recommendation, personalized marketing, content creation, and virtual assistants enhance user experiences in the entertainment industry.

Challenges and Future Directions

Despite its advancements, AI faces challenges such as ethical concerns, privacy issues, and the need for robust and interpretable AI systems. Addressing these challenges requires efforts in areas like explainable AI, ethics frameworks, AI safety, and human-AI collaboration. Moving forward, stakeholders must navigate these challenges responsibly to ensure that AI technologies benefit society while mitigating potential risks.

3.2 Machine Learning (ML)

Machine Learning (ML) is a sub-field of artificial intelligence (AI) that focuses on developing algorithms capable of learning from data and making predictions or decisions without being explicitly programmed. ML techniques enable computers to identify patterns, extract insights, and make informed decisions based on data. This section provides a comprehensive overview of ML, including its fundamental concepts, types of learning, popular algorithms, applications, and future directions.

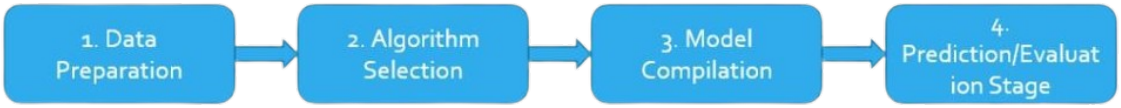

Figure 3.2: Machine Learning Process

Fundamental Concepts of Machine Learning

At the core of machine learning are the following fundamental concepts:

- **Data:** ML algorithms learn from data, which can be structured (e.g., tabular data) or unstructured (e.g., text, images, audio).
- **Features:** Features are the individual variables or attributes present in the data that are used to make predictions or decisions.
- **Models:** ML models are mathematical representations of patterns and relationships within the data.
- **Training:** Training involves feeding data into ML algorithms to adjust model parameters and optimize performance.
- **Evaluation:** Evaluation assesses the performance of ML models on unseen data to ensure generalization and reliability.
- **Prediction:** ML models use learned patterns to make predictions or decisions on new, unseen data.

Types of Machine Learning

Machine learning can be categorized into three main types:

1. **Supervised Learning:** In supervised learning, algorithms learn from labelled data, where each example is associated with a target output. The goal is to learn a mapping from inputs to outputs, enabling the algorithm to make predictions on new, unseen data.
2. **Unsupervised Learning:** Unsupervised learning involves learning from unlabeled data, where the algorithm seeks to identify hidden patterns or structures within the data. Common tasks include clustering, dimensionality reduction, and anomaly detection.
3. **Reinforcement Learning:** Reinforcement learning (RL) is a type of learning where an agent learns to interact with an environment to achieve a goal. The agent receives feedback in the form of rewards or penalties based on its actions, allowing it to learn optimal strategies over time.

Popular Machine Learning Techniques

Machine learning algorithms encompass a wide range of techniques, including:

- **Linear Regression:** A simple algorithm for modelling the relationship between a dependent variable and one or more independent variables.
- **Logistic Regression:** Used for binary classification tasks, logistic regression models the probability of a binary outcome.
- **Decision Trees:** Tree-based algorithms that recursively partition the feature space to make predictions.
- **Random Forest:** An ensemble method that combines multiple decision trees to improve performance and reduce overfitting.
- **Support Vector Machines (SVM):** SVMs are powerful algorithms for classification and regression tasks, particularly in high-dimensional spaces.
- **K-Nearest Neighbours (KNN):** KNN is a non-parametric algorithm that makes predictions based on the similarity of input data points to their nearest neighbours.

Applications of Machine Learning

Machine learning has a wide range of applications across various domains, including:

- **Healthcare:** ML is used for medical diagnosis, patient monitoring, personalized treatment planning, and drug discovery.
- **Finance:** ML powers algorithmic trading, fraud detection, credit scoring, and risk assessment in the financial sector.
- **E-commerce:** ML drives recommendation systems, personalized marketing, demand forecasting, and customer segmentation in e-commerce platforms.
- **Autonomous Vehicles:** ML algorithms enable object detection, path planning, and decision-making in autonomous vehicles.
- **Natural Language Processing:** ML techniques facilitate tasks such as machine translation, sentiment analysis, text summarisation, and chat-bots in NLP applications.

Challenges and Future Directions

While ML has made significant advancements, it faces several challenges, including data quality, interpretability, fairness, and scalability. Addressing these challenges requires interdisciplinary efforts and ongoing research in areas such as explainable AI, fairness and bias mitigation, federated learning, and lifelong learning. Additionally, emerging trends like federated learning, meta-learning, and automated machine learning (AutoML) hold promise for advancing the capabilities and accessibility of ML technologies.

3.3 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the structure and functioning of biological neural networks in the human brain. ANNs consist of interconnected nodes, or neurons, organized in layers, allowing them to learn complex patterns and relationships from data. This section provides a detailed overview of ANNs, covering their architecture, training process, types, activation functions, popular architectures, applications, and future directions.

Architecture of Artificial Neural Networks

The architecture of an artificial neural network typically consists of the following components:

- **Input Layer:** The input layer receives raw data or features and passes them to the network for processing.
- **Hidden Layers:** Hidden layers are intermediate layers between the input and output layers, where neurons perform computations and extract features from the input data.
- **Output Layer:** The output layer produces the final predictions or decisions based on the processed input data.
- **Connections (Weights):** Connections between neurons are represented by weights, which are adjusted during the training process to minimize prediction errors.
- **Activation Functions:** Activation functions introduce non-linearity into the network, enabling it to learn complex relationships and make non-linear trans-

formations of the input data.

Training Process of Artificial Neural Networks

The training process of ANNs involves the following key steps:

1. **Initialization:** Initialize the weights and biases of the network to small random values.
2. **Forward Propagation:** Pass the input data through the network to obtain predictions.
3. **Loss Calculation:** Calculate the difference between the predicted output and the true output using a loss function.
4. **Backpropagation:** Propagate the error backwards through the network and adjust the weights using gradient descent optimization algorithms to minimize the loss.
5. **Iteration:** Repeat the forward and backward propagation steps iteratively until the model converges to optimal weights and output.

Types of Artificial Neural Networks

Artificial Neural Networks can be classified into various types based on their architecture and learning mechanisms, including:

- **Feedforward Neural Networks (FNNs):** FNNs are the simplest type of neural networks, where information flows in one direction from the input layer to the output layer without feedback loops.

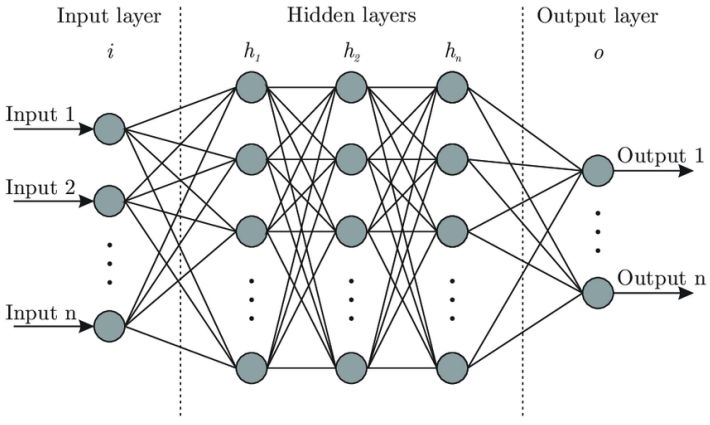

Figure 3.3: An Example of Multi-layered FNN

- **Recurrent Neural Networks (RNNs):** RNNs have connections that form directed cycles, allowing them to process sequences of data with temporal dependencies.

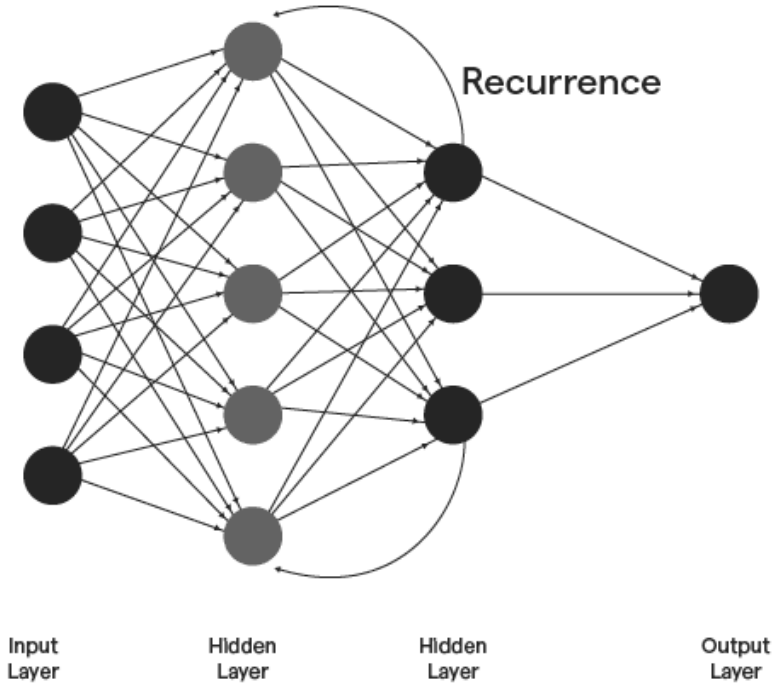

Figure 3.4: An Example of RNN

- **Convolutional Neural Networks (CNNs):** CNNs are specialized for processing grid-like data, such as images, by leveraging convolutional layers that extract spatial features.

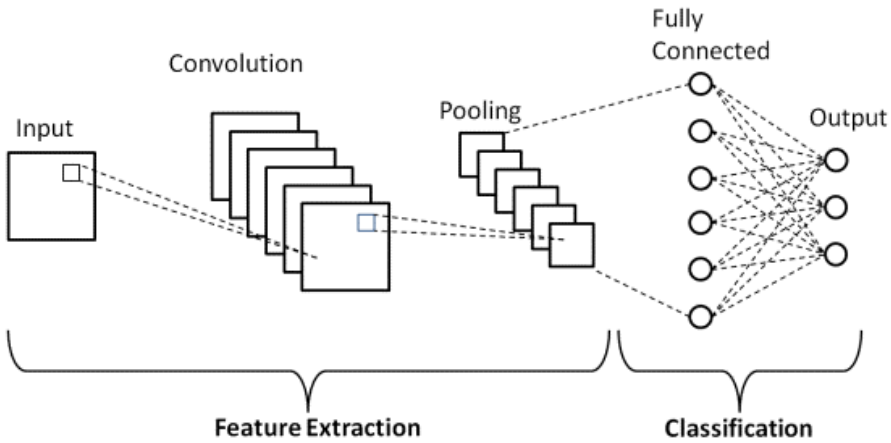

Figure 3.5: An Example of CNN

Activation Functions in Artificial Neural Networks

Activation functions introduce non-linearity into the network, enabling it to learn complex patterns and relationships in the data. Commonly used activation functions include:

1. **Sigmoid Function (Logistic Function):** The sigmoid function squashes the input values into the range [0, 1], making it suitable for binary classification tasks.

   Equation:

   $$f(x) = \frac{1}{1 + e^{-x}}$$

   Graph:

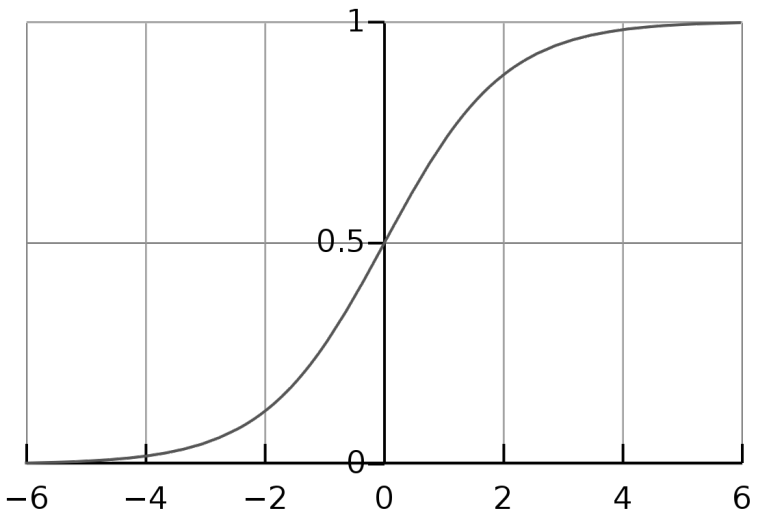

Figure 3.6: Graph for a Sigmoid Function

2. **Hyperbolic Tangent (Tanh) Function:** Similar to the sigmoid function, the tanh function squashes the input values into the range [-1, 1], providing stronger gradients and better convergence properties.

   Equation:

   $$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Graph:

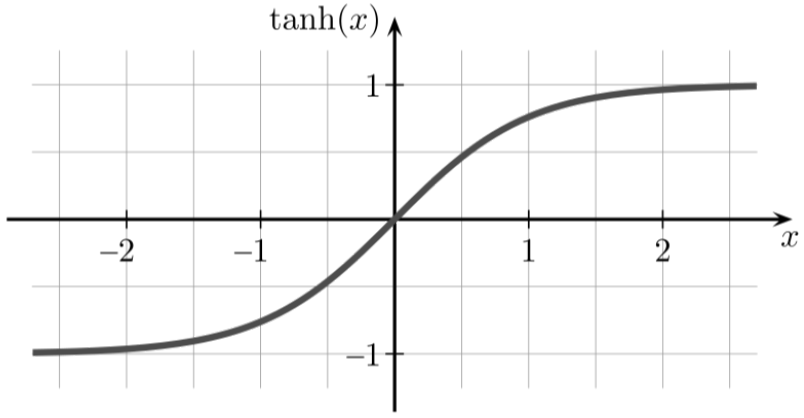

Figure 3.7: Graph for a Tanh Function

3. **Rectified Linear Unit (ReLU):** ReLU sets all negative input values to zero, introducing sparsity and enabling faster training compared to sigmoid and tanh functions.

Equation:

$$f(x) = max(0, x)$$

Graph:

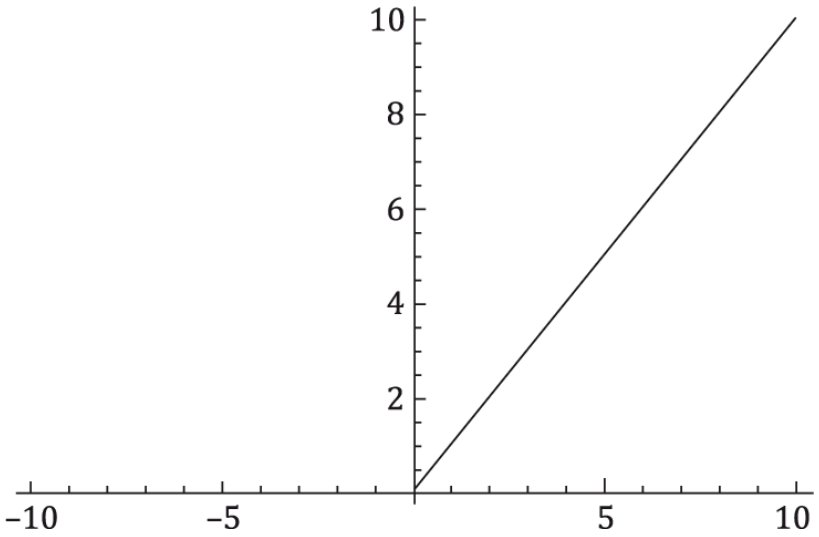

Figure 3.8: Graph for a ReLU Function

Popular Architectures of Artificial Neural Networks

Several popular architectures of artificial neural networks have emerged, including:

- **Multilayer Perceptrons (MLPs):** MLPs consist of multiple layers of neurons with fully connected connections between adjacent layers.
- **Long Short-Term Memory (LSTM) Networks:** LSTMs are a type of recurrent neural network designed to capture long-term dependencies in sequential data.
- **Gated Recurrent Units (GRUs):** GRUs are similar to LSTMs but have a simpler architecture, making them more computationally efficient.
- **Deep Belief Networks (DBNs):** DBNs are hierarchical generative models consisting of multiple layers of stochastic, latent variables.

Applications of Artificial Neural Networks

Artificial Neural Networks have found applications across various domains, including:

- **Computer Vision:** CNNs are widely used for image classification, object detection, facial recognition, and image generation tasks.
- **Natural Language Processing:** RNNs and transformer-based architectures like BERT are employed for tasks such as machine translation, text generation, sentiment analysis, and named entity recognition.
- **Speech Recognition:** Recurrent and convolutional neural networks are utilized for speech recognition, speaker identification, and speech synthesis applications.
- **Healthcare:** ANNs are applied in medical image analysis, disease diagnosis, drug discovery, and personalized medicine.
- **Finance:** Neural networks are employed for stock market prediction, credit scoring, fraud detection, and algorithmic trading in the financial sector.

Challenges and Future Directions

Despite their success, artificial neural networks face challenges such as overfitting, interpretability, and scalability. Addressing these challenges requires research in areas such as regularization techniques, explainable AI, model compression, and efficient training algorithms. Furthermore, emerging trends like neurosymbolic AI, lifelong learning, and neuromorphic computing hold promise for advancing the capabilities and scalability of artificial neural networks.

3.4 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm concerned with learning how to make sequences of decisions, known as actions, to maximize cumulative rewards in an environment. RL agents learn through trial and error by interacting with the environment, receiving feedback in the form of rewards or penalties, and adjusting their behaviour to achieve long-term goals. This section provides a comprehensive overview of RL, covering its key components, algorithms, applications, and challenges.

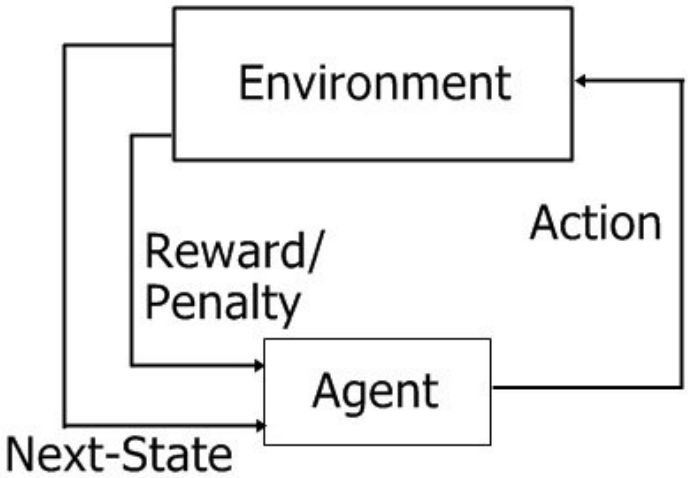

Figure 3.9: Process of Reinforcement Learning

Key Components of Reinforcement Learning

The key components of a reinforcement learning system include:

- **Agent:** The decision-making entity that interacts with the environment.
- **Environment:** The external system with which the agent interacts.
- State: A representation of the environment at a particular time.

- **Action:** The set of possible decisions that the agent can take.
- **Reward:** Feedback from the environment indicating the success or failure of an action.
- **Policy:** The strategy or mapping from states to actions that the agent follows to maximize rewards.
- **Value Function:** An estimate of the expected cumulative rewards obtained by following a particular policy.
- **Model:** An internal representation of the environment's dynamics, used for planning and decision-making.

Reinforcement Learning Techniques

Reinforcement Learning algorithms can be broadly categorized into model-free and model-based approaches. Some popular RL algorithms include:

- **Q-Learning:** A model-free algorithm that learns an action-value function $\mathcal{Q}(\mathbf{s}, \mathbf{a})$ to estimate the expected cumulative rewards of taking action $\mathbf{a}$ in state $\mathbf{s}$.
- **Deep Q-Networks (DQN):** A deep learning-based extension of Q-learning that uses a neural network to approximate the action-value function.
- **Policy Gradient Methods:** Model-free algorithms that directly optimize the policy to maximize rewards, typically using gradient ascent on a performance objective.
- **Actor-Critic Methods:** Model-free algorithms that combine the advantages of policy gradients and value-based methods by maintaining both a policy (actor) and a value function (critic).

Applications of Reinforcement Learning

Reinforcement Learning has found applications in various domains, including:

- **Robotics:** RL is used to train robotic agents to perform complex tasks such as grasping objects, navigation, and manipulation.
- **Game Playing:** RL algorithms have achieved superhuman performance in games like chess, Go, and video games by learning optimal strategies through self-play.

- **Autonomous Vehicles:** RL is employed to train autonomous vehicles to navigate traffic, follow traffic rules, and make safe driving decisions.
- **Finance:** RL is applied in algorithmic trading, portfolio management, and risk assessment to optimize investment strategies and maximize returns.

Challenges and Future Directions

Despite its success, RL faces challenges such as sample inefficiency, exploration-exploitation trade-offs, and generalization to unseen environments. Addressing these challenges requires research in areas such as transfer learning, meta-learning, and hierarchical reinforcement learning. Furthermore, incorporating insights from cognitive science and neuroscience may lead to more robust and human-like learning algorithms.

3.4.1 Markov Decision Process (MDP)

Markov Decision Processes (MDPs) provide a mathematical framework for modelling decision-making in stochastic environments. In an MDP, an agent interacts with an environment in discrete time steps by selecting actions to perform, which transition the system from one state to another. Each action taken by the agent results in a stochastic outcome, where the next state and immediate reward are determined by the current state and action, following certain probabilistic rules.

Components of MDP

An MDP is characterized by the following components:

1. **State Space $\mathcal{S}$**: The set of all possible states that the environment can be in. States represent the relevant aspects of the environment that the agent needs to consider when making decisions.
2. **Action Space $\mathcal{A}$:** The set of all possible actions that the agent can take. Actions are the choices available to the agent at each state and influence the subsequent state transition and reward received.
3. **Transition Probability Function $\mathcal{P}$**: This function defines the probability distribution over the next possible states given the current state and action. Formally, $\mathcal{P}(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ represents the probability of transitioning to state $\mathbf{s}'$ when taking action $\mathbf{a}$ in state $\mathbf{s}$.

4. **Reward Function $\mathcal{R}$:** The reward function specifies the immediate reward received by the agent upon transitioning from one state to another. It maps state-action pairs to real-valued rewards, indicating the desirability of the agent's actions in different states.

5. **Discount Factor $\mathcal{Y}$**: The discount factor determines the importance of future rewards relative to immediate rewards. It is a value between 0 and 1, where 0 indicates that the agent only considers immediate rewards, and 1 indicates that future rewards are equally important.

MDP Dynamics

The dynamics of an MDP can be described by the following equations:

1. **State Transition Probability Function ($\mathcal{P}$):**

$$\mathcal{P}(s_{t+1}|s_t, a_t) = Pr(S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t)$$

   This function represents the probability of transitioning to state $\mathbf{s_{t+1}}$ given that the agent is in state $\mathbf{s_t}$ and takes action $\mathbf{a_t}$.

2. **Reward Function ($\mathcal{R}$):**

$$\mathcal{R}(s_t, a_t) = \mathbb{E}[R_{t+1}|S_t = s_t, A_t = a_t]$$

   This function represents the expected immediate reward received by the agent upon taking action $\mathbf{a_t}$ in state $\mathbf{s_t}$.

Objective in MDP

The goal of the agent in an MDP is to learn a policy π that maximizes the expected cumulative reward, also known as the return. The return is defined as the sum of discounted rewards obtained over time, where the discount factor $\mathcal{Y}$ determines the importance of future rewards. Mathematically, the objective is to find the optimal policy π^* that maximizes the expected return:

$$\pi^* = argmax_\pi \mathbb{E}_\pi[\Sigma_{t=0}^{\infty} \mathcal{Y}^t r_t]$$

where $\mathbb{E}_\pi$ denotes the expectation over trajectories generated by policy π.

Solving MDPs

Various algorithms can be used to solve MDPs and find the optimal policy. Some common approaches include dynamic programming methods such as value iteration and policy iteration, which require knowledge of the complete MDP model. Model-free methods such as Q-learning and policy gradient methods learn directly from experience without explicitly modelling the MDP dynamics.

Applications of MDPs

MDPs have widespread applications in fields such as robotics, autonomous systems, finance, and healthcare. They are used to model sequential decision-making problems where uncertainty and stochasticity are present, enabling agents to make optimal decisions in complex environments.

Understanding Markov Decision Processes is essential for developing and applying reinforcement learning algorithms effectively. By leveraging the formalism provided by MDPs, researchers and practitioners can design intelligent systems that learn to make decisions in uncertain and dynamic environments, leading to advancements in artificial intelligence and autonomous systems.

3.5 Unity3D Game Engine

Unity3D, developed by Unity Technologies, stands as a premier game engine renowned for its versatility and power. Since its inception, Unity3D has continually evolved, shaping the landscape of game development and interactive experiences. Initially released in 2005 as a Mac OS X-exclusive engine, Unity3D quickly gained traction for its user-friendly interface and robust feature set. Over the years, it has grown into a cross-platform powerhouse, supporting development for a multitude of devices and platforms, from desktop and mobile to consoles and virtual reality.

Figure 3.10: Unity3D Logo [Source]

Historical Evolution

Unity3D's journey began with a vision to democratize game development, making it accessible to developers of all backgrounds and skill levels. Its roots trace back to the pioneering work of David Helgason, Joachim Ante, and Nicholas Francis, who sought to create a game engine that combined ease of use with powerful capabilities. The early versions of Unity3D focused on providing a comprehensive toolset for 3D game development, with features such as scene editing, asset management, and scripting support.

Key Features of Unity3D

- **Physics Simulation:** The engine includes a built-in physics engine that enables realistic simulation of object interactions, collisions, and dynamics. Developers can use Unity3D's physics components and constraints to create dynamic and interactive environments where objects behave according to real-world physics principles.
- **Animation and Rigging:** Unity3D provides robust animation tools for creating and controlling character animations, object animations, and complex motion sequences. It supports skeletal animation, blend trees, inverse kinematics (IK), and animation retargeting, allowing developers to bring characters and objects to life with fluid and natural movements.
- **Scripting and Programming:** Unity3D supports scripting and programming in C#, JavaScript (UnityScript), and Boo. Developers can write custom scripts to define game logic, behaviour, and interactions, as well as to extend the engine's functionality through custom editor tools and plugins.
- **Asset Pipeline:** Unity3D features a streamlined asset pipeline that facilitates the import, management, and manipulation of various asset types, including 3D models, textures, audio files, animations, and shaders. It provides built-in tools for organizing assets, optimizing performance, and creating asset bundles for efficient loading and streaming.
- **Editor and Workflow:** The Unity3D editor offers a user-friendly interface with intuitive tools and workflows for designing, prototyping, and iterating game content. It supports real-time editing, instant previews, scene management, and collaborative development, enabling teams to work together seamlessly on projects of any scale.

Technical Advancements in Unity3D

Unity3D's success can be attributed to its robust technical foundation and innovative features. Under the hood, Unity3D utilizes a modern graphics rendering pipeline, leveraging APIs such as DirectX, OpenGL, and Vulkan to achieve high-fidelity graphics and visual effects. Its support for physically-based rendering (PBR), real-time lighting, and post-processing enables developers to create stunning and immersive environments.

In addition to its graphics prowess, Unity3D boasts advanced multithreading capabilities, allowing for efficient utilization of multi-core processors and improved performance. This enables developers to create complex and highly interactive experiences without sacrificing performance or scalability.

Cross-Platform Development

One of Unity3D's defining features is its cross-platform development capabilities. Developers can write code once and deploy their games to a wide range of platforms, including Windows, macOS, Linux, iOS, Android, WebGL, PlayStation, Xbox, and more. Unity3D's platform-agnostic nature simplifies the development process and ensures maximum reach for developers, allowing them to target multiple platforms with minimal effort.

Community and Ecosystem

Unity3D's success is also attributed to its thriving community and ecosystem. The Unity Asset Store serves as a hub for developers to discover, share, and monetize assets, tools, and plugins. From 3D models and textures to scripts and editor extensions, the Asset Store offers a vast library of resources that accelerate development and enhance productivity.

3.5.1 Unity Editor

The Unity Editor Application for Windows stands as the primary tool for developers to create, edit, and manage Unity projects. Offering a comprehensive suite of features and functionalities, it serves as the central hub for game development within the Unity ecosystem. In this detailed exploration, we delve into the core elements, engine physics, and editor interface that define the Unity Editor Application on the Windows platform.

Core Elements

In the Unity Editor Application, several core elements play pivotal roles in the game development process, offering essential functionalities and tools for developers. These elements encompass GameObjects, Components, Scenes, Assets, Prefabs, and Scripts.

- **GameObjects:** GameObjects are the fundamental building blocks of Unity scenes, representing entities within the game world. They can represent characters, props, cameras, lights, and other elements. Each GameObject can have multiple components attached to it, defining its behaviour, appearance, and interactions.

- **Components:** Components are modular units of functionality that can be attached to GameObjects to define their behaviour and appearance. Unity provides a wide range of built-in components, such as Transform, Rigidbody, Collider, Renderer, and Script, as well as custom components created by developers. Components can be added, removed, and manipulated within the Unity Editor, allowing for extensive customization and flexibility.

- **Scenes:** Scenes are self-contained environments within Unity projects that contain GameObjects, assets, and configurations. They represent different levels, menus, or segments of a game and serve as containers for organizing and managing game content. Developers can create, open, save, and manage scenes within the Unity Editor, enabling seamless navigation and editing of game environments.

- **Assets:** Assets are digital files that contain content such as textures, models, audio clips, animations, and scripts used in Unity projects. They are stored in the project's Asset folder and can be imported, exported, and manipulated within the Unity Editor. Assets play a crucial role in shaping the visual, auditory, and interactive aspects of a game, allowing developers to create rich and immersive experiences.

- **Prefabs:** Prefabs are reusable GameObject templates that encapsulate configurations, components, and settings. They enable developers to create and instantiate GameObject instances with predefined properties and behaviours, streamlining the development workflow and promoting consistency and efficiency. Prefabs can be modified, updated, and instantiated dynamically within Unity projects, making them invaluable for rapid prototyping and iteration.

- **Scripts:** Scripts are pieces of code written in programming languages such as C# or JavaScript that define the behaviour and logic of gameobjects and components. They enable developers to implement game mechanics, interactions,

and systems, as well as extend the functionality of Unity Editor. Scripts can be attached to gameobjects as components, allowing for dynamic and interactive behaviours within Unity projects.

Physics Engine

Unity's physics engine plays a crucial role in simulating realistic interactions and behaviours within game worlds. Leveraging advanced physics algorithms and calculations, it enables the realistic simulation of rigid bodies, collisions, forces, and constraints. Some of the key offerings by the Physics Engine are:

- **Collider Components:** Unity provides a variety of collider components, such as BoxCollider, SphereCollider, CapsuleCollider, and MeshCollider, which define the shape and size of collidable objects. These components interact with the physics engine to detect and respond to collisions accurately.
- **Rigidbody Component:** The Rigidbody component adds physics simulation to GameObjects, allowing them to respond to forces such as gravity, friction, and applied forces. It enables dynamic movement, rotation, and collision detection for GameObjects, making them behave like physical objects in the game world.
- **Physics Materials:** Physics materials define the friction and bounciness properties of colliding surfaces, influencing the behaviour of interactions between objects. By adjusting parameters such as friction, bounciness, and friction combined, developers can fine-tune the physics behaviour of GameObjects to achieve desired effects.

Editor Interface

The Unity Editor Interface provides developers with a versatile and intuitive workspace for creating, editing, and managing Unity projects. It comprises various views, windows, and tools that facilitate navigation, manipulation, and customization of game content. The default interface of the Unity Editor is shown in the figure below. This layout can be modified by the user. Let's see the details about the windows/components of this layout.

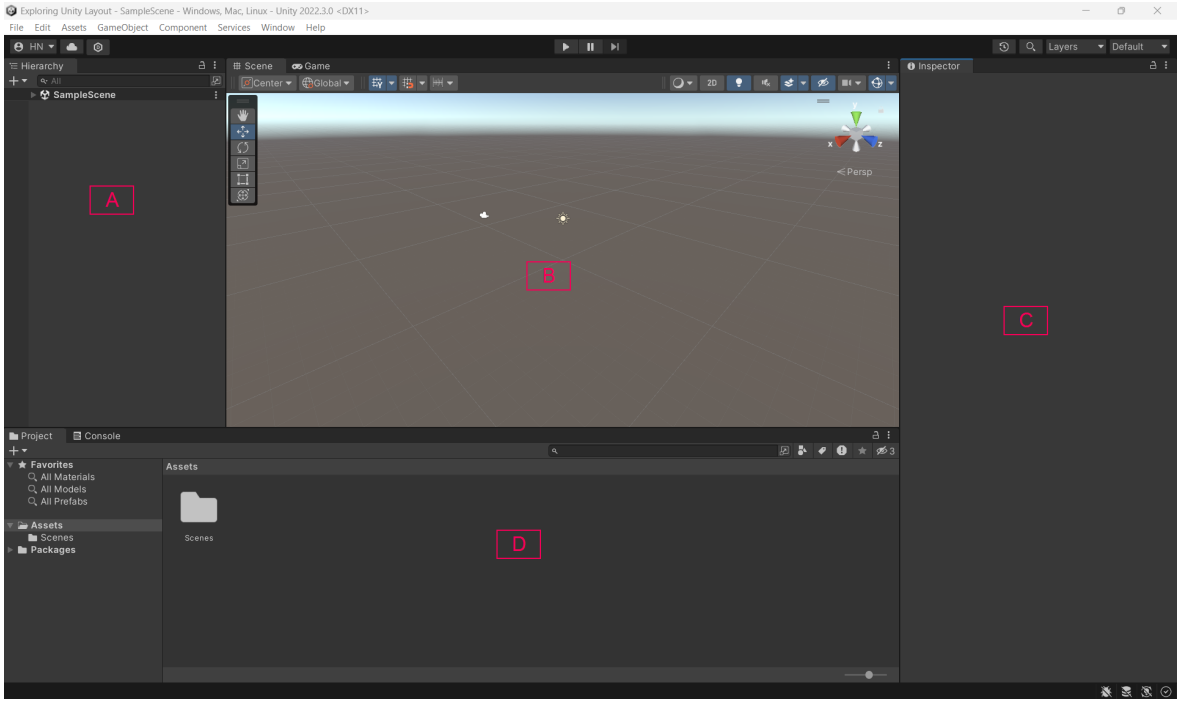

Figure 3.11: The Default Layout of Unity Editor [Source]

(A) **Hierarchy View:** The Hierarchy view displays a hierarchical list of GameObjects present in the current scene, allowing developers to organize, select, and manipulate GameObjects easily. It provides a visual representation of the scene's structure and enables hierarchical editing of GameObjects and their components.

Figure 3.12: The Unity Editor Hierarchy Window

(B) **Scene View & Game View:** The Scene view provides a visual representation of the game world, allowing developers to view, navigate, and edit scenes interac-

tively. It offers tools for manipulating GameObjects, adjusting camera settings, and navigating the scene from different perspectives, facilitating scene creation and editing. The Game view displays the game output and allows developers to preview and play the game within the Unity Editor. It provides real-time feedback on gameplay, graphics, and performance, enabling developers to test and iterate on game mechanics, visuals, and interactions.

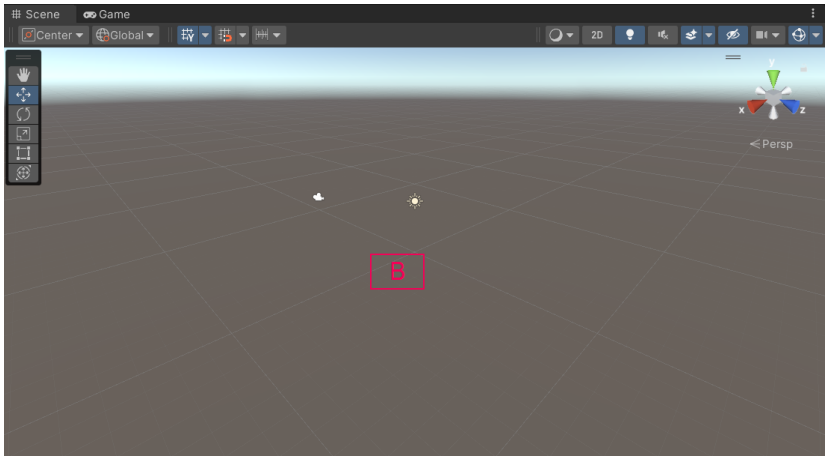

Figure 3.13: The Unity Editor Scene View

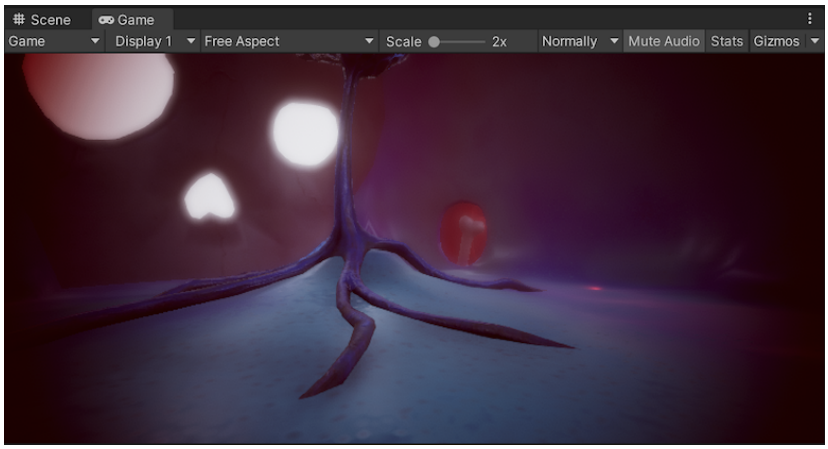

Figure 3.14: The Unity Editor Game View

(C) **Inspector Window:** The Inspector window displays detailed information and controls for selected GameObjects, components, and assets. It allows developers to view and edit properties such as transform values, material settings, script parameters, and asset attributes, providing fine-grained control and customization options.

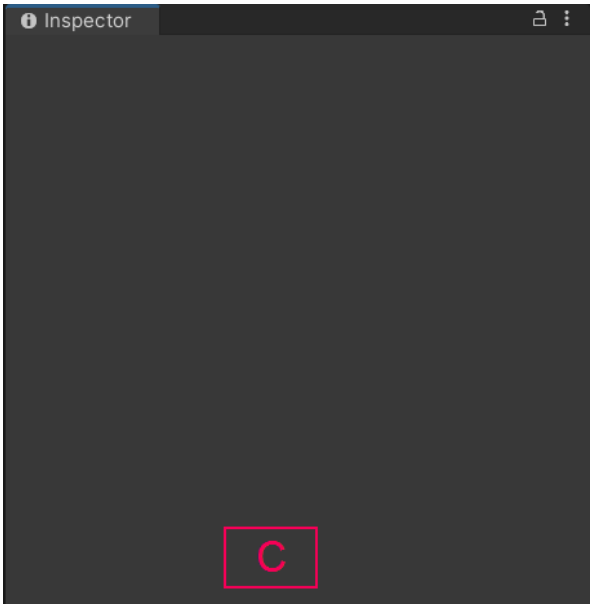

Figure 3.15: The Unity Editor Inspector Window

(D) **Project Window:** The Project window serves as a centralized repository for project assets, allowing developers to browse, import, organize, and manage assets efficiently. It provides features such as searching, filtering, and tagging, as well as asset previews and previews, facilitating asset discovery and management.

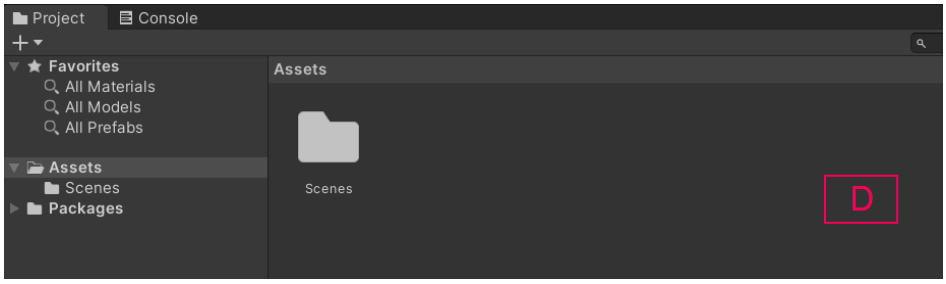

Figure 3.16: The Unity Editor Project Window

3.6 Unity ML-Agents Framework

The Unity ML-Agents Framework stands at the forefront of innovation, revolutionizing the integration of artificial intelligence (AI) and machine learning (ML) within Unity projects. Developed with the purpose of empowering developers to create intelligent, adaptive agents capable of learning and interacting within virtual environments, Unity ML-Agents has rapidly gained traction as a powerful tool for advancing AI research and game development alike.

Originally introduced as an open-source project by Unity Technologies, the Unity ML-Agents Framework has since evolved into a comprehensive toolkit, providing developers with a diverse array of algorithms and tools for training and deploying intelligent agents. Its GitHub repository serves as a hub for collaborative development and community-driven innovation, fostering a vibrant ecosystem of researchers, developers, and enthusiasts dedicated to pushing the boundaries of AI and ML in Unity.

The journey of Unity ML-Agents traces back to its inception in 2017 by Arthur Juliani, driven by the vision of democratizing AI and enabling the creation of lifelike virtual experiences. Since then, it has undergone continuous refinement and enhancement, with regular updates, contributions from the open-source community, and integration of cutting-edge research findings. Today, Unity ML-Agents stands as a testament to the collective efforts of AI researchers, game developers, and Unity enthusiasts worldwide, ushering in a new era of intelligent virtual agents and immersive interactive experiences.

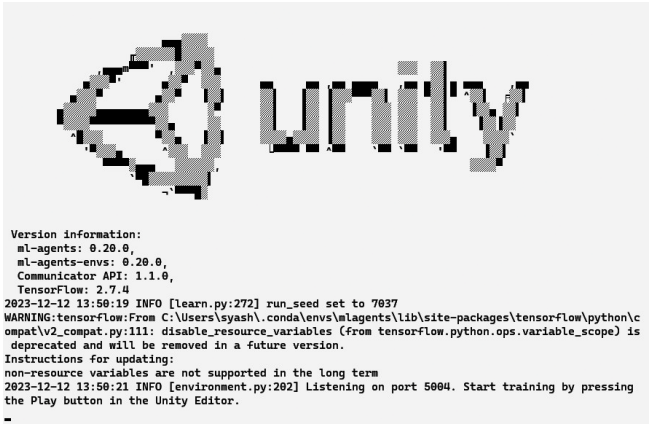

Figure 3.17: Unity ML-Agents in Windows Terminal

Agent Parameters

Agent Parameters define the characteristics and behaviours of ML agents within Unity projects, allowing developers to customize their learning process and interactions with the environment. These parameters encompass settings such as observation space, action space, reward structure, and learning algorithm, providing fine-grained control over agent behaviour and performance. Let's delve into the various aspects of Agent Parameters, including Behavior Parameters, Environment, Actions, and Rewards.

- **Behaviour Parameters:**
  Behaviour Parameters encompass a range of settings that dictate how an agent interacts with its environment and learns from experience. These parameters

include:

  - **Observation Space:** Defines the type and dimensions of the data observed by the agent from its environment. Observations can include visual inputs (e.g., camera images), vector inputs (e.g., position, velocity), or a combination of both. Specifying the observation space allows the agent to perceive and interpret relevant information from its surroundings.
  - **Action Space:** Specifies the possible actions that the agent can take within the environment. Actions can be discrete (e.g., move left, move right) or continuous (e.g., accelerate, brake) and may vary in complexity depending on the task. Defining the action space enables the agent to choose actions based on its current state and policy.
  - **Behaviour Type:** Determines the learning algorithm employed by the agent, such as Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), or Deep Q-Networks (DQN). Selecting an appropriate behaviour type influences the agent's learning process, convergence behaviour, and performance on the task.
  - **Reward Signals:** Specifies the reward structure used to provide feedback to the agent based on its actions and interactions with the environment. Rewards can be sparse or dense, immediate or delayed, and shaped to incentivize desired behaviours. Designing effective reward signals is crucial for guiding the agent towards achieving the desired objectives.

- **Environment:** The Environment defines the virtual world or scenario in which the agent operates and interacts. It encompasses the following aspects:

  - **Scene Setup:** Describes the layout, objects, obstacles, and dynamics of the virtual environment. Scenes may represent realistic or abstract settings, ranging from simple grid worlds to complex 3D simulations.
  - **Simulation Parameters:** Includes settings related to physics, time scale, rendering, and other simulation aspects. Adjusting simulation parameters can influence the realism, performance, and computational requirements of the environment.
  - **Reset Conditions:** Specifies conditions under which the environment resets, such as reaching a terminal state, exceeding a time limit, or encountering a failure condition. Reset conditions enable episodic learning and facilitate repeated trials for the agent to learn from experience.

- **Actions:** Actions refer to the decisions and behaviours that the agent can perform within the environment. They are defined based on the action space specified in the behaviour parameters and may include:
  - **Discrete Actions:** Represent discrete choices or movements that the agent can make, such as selecting from a finite set of options (e.g., move left, move right, stay).
  - **Continuous Actions:** Represent continuous or analogue control signals that allow for fine-grained adjustments and smooth movements (e.g., throttle, steering angle, pitch angle).
  - **Action Constraints:** Define constraints or limits on the range and magnitude of actions to ensure safe and meaningful interactions with the environment.
- **Rewards:** Rewards play a crucial role in shaping the agent's behaviour and learning process by providing feedback on the desirability of its actions. They are defined based on the reward signals specified in the behaviour parameters and may include:
  - **Immediate Rewards:** Given to the agent immediately after performing an action, typically based on the immediate consequences of the action (e.g., reaching a target, avoiding an obstacle).
  - **Delayed Rewards:** Given to the agent after a series of actions or over multiple time steps, reflecting the cumulative impact of its decisions on long-term objectives (e.g., maximizing cumulative score, achieving a task objective).
  - **Sparse Rewards:** Given infrequently or only in response to specific events or achievements, requiring the agent to explore and discover rewarding behaviours.
  - **Shaped Rewards:** Designed to provide additional guidance or incentives to the agent by shaping the reward landscape and highlighting desired behaviours (e.g., penalizing collisions, rewarding progress towards goals).

By carefully defining Behavior Parameters, Environment, Actions, and Rewards, developers can create rich and immersive learning environments for intelligent agents within Unity projects, enabling them to acquire skills, adapt to challenges, and achieve desired objectives through iterative learning and optimization.

Unity ML-Agents Training Process

The training process in Unity ML-Agents represents a pivotal stage in the development of intelligent agents, where machine learning algorithms are employed to teach agents how to perceive, act, and learn from their environment. At the heart of this process lies a sophisticated interplay of components and techniques, orchestrated to enable agents to acquire skills, refine strategies, and adapt to dynamic environments through iterative learning and optimization.

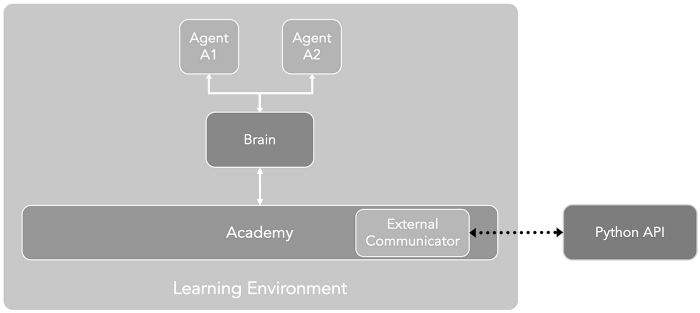

Figure 3.18: Unity ML-Agents Training Process

Central to the training process is the Unity ML-Agents External Communicator, a vital interface that facilitates communication between Unity environments and external training processes. This lightweight and efficient component enables seamless integration with Python-based training frameworks, such as TensorFlow and PyTorch, allowing developers to leverage the full power of state-of-the-art machine learning algorithms and computational resources for training intelligent agents.

The training process typically unfolds in several iterative phases, each aimed at progressively improving the agent's performance and learning capabilities:

1. **Initialization:** The training process begins with the initialization of the Unity environment and the instantiation of the agent(s) within the virtual world. Initial parameters, such as observation space, action space, and reward structure, are defined based on the specifications provided in the Unity ML-Agents Behavior Parameters.

2. **Data Collection:** During this phase, the agent(s) interact with the environment, collecting observations, performing actions, and receiving feedback in the form of rewards. These interactions generate trajectories of experience, which are subsequently used to train the agent's neural network model.

3. **Neural Network Training:** Using the collected trajectories of experience, the agent's neural network model is trained using machine learning algorithms, such as Proximal Policy Optimization (PPO) or Soft Actor-Critic (SAC). Through a process of backpropagation and optimization, the neural network learns to map observations to actions, maximizing cumulative rewards and improving performance over time.

4. **Policy Optimization:** As training progresses, the agent's policy, or strategy for selecting actions, is continuously refined and optimized to maximize expected rewards. This optimization process involves fine-tuning neural network parameters, adjusting exploration-exploitation trade-offs, and balancing between the exploration of new strategies and the exploitation of learned knowledge.

5. **Evaluation and Iteration:** Periodically, the trained agent(s) are evaluated on held-out validation environments or test scenarios to assess their performance and generalization capabilities. Based on evaluation results, training parameters may be adjusted, and the training process iterated to further improve agent performance and robustness.

Algorithms Provided

Unity ML-Agents offers a diverse selection of state-of-the-art machine learning algorithms, empowering developers to train intelligent agents capable of mastering a wide range of tasks and environments. Each algorithm brings unique strengths and characteristics to the table, catering to different learning scenarios and performance requirements. Here's a brief overview of the key algorithms provided by Unity ML-Agents:

- **Proximal Policy Optimization (PPO):**
  - PPO is a popular policy optimization algorithm designed for training deep reinforcement learning agents.
  - It operates by iteratively optimizing the policy parameters to maximize expected rewards while ensuring stable and efficient learning.
  - PPO achieves this by constraining the policy update steps to be "proximal" to the current policy, preventing large policy changes that may destabilize training.
  - Known for its simplicity, stability, and sample efficiency, PPO has emerged as a go-to choice for training agents in a variety of environments, from simple grid worlds to complex 3D simulations.

- **Soft Actor-Critic (SAC):**
  - SAC is an off-policy actor-critic algorithm that leverages the principles of maximum entropy reinforcement learning to achieve robust and exploratory behaviour.
  - Unlike traditional actor-critic methods, SAC incorporates entropy regularization, encouraging agents to explore diverse strategies and learn more efficiently.
  - SAC is particularly well-suited for continuous action spaces and environments with sparse rewards, where exploration is crucial for discovering optimal policies.
  - With its emphasis on stochastic policies and entropy maximization, SAC has demonstrated impressive performance in challenging tasks such as robotic manipulation and locomotion.

Each of these algorithms offers a distinct set of advantages and trade-offs, making them suitable for different learning scenarios and problem domains. By providing a comprehensive suite of algorithms, Unity ML-Agents empowers developers to explore and experiment with various approaches, tailoring their training pipelines to the specific requirements of their projects and domains. Now let's see a detailed explanation of PPO and SAC.

3.6.1 Proximal Policy Optimisation Algorithm (PPO)

Proximal Policy Optimization (PPO) stands as a prominent algorithm in the domain of reinforcement learning, celebrated for its robustness, simplicity, and sample efficiency. Designed to address the challenges of training deep reinforcement learning agents in complex environments, PPO offers a principled and scalable approach to policy optimization, ensuring stable and efficient learning dynamics. Let's delve into the inner workings of PPO, exploring its mathematical foundations, advantages, and disadvantages.

Mathematical Foundations

Proximal Policy Optimization (PPO) is grounded in the principles of policy gradient methods, aiming to iteratively improve the policy of an agent to maximize expected cumulative rewards. At its core, PPO utilizes a surrogate objective function that guides

policy updates while ensuring stable and efficient learning dynamics. Let's delve into the mathematical underpinnings of PPO:

1. **Objective Function:**
   PPO's objective function is defined as follows:

$$J(\theta) = \mathbb{E}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)]$$

   where:

   - $\mathbf{J}(\theta)$ represents the surrogate objective function to be optimized.
   - θ denotes the policy parameters.
   - $\mathbf{r_t}(\theta)$ denotes the probability ratio between the probabilities of actions under the current and old policies.
   - $\mathbf{A_t}$ denotes the advantage estimate, representing the advantage of taking action $\mathbf{A_t}$ in the state $\mathbf{s_t}$ over the baseline value function.
   - ϵ is a hyperparameter that controls the extent of policy updates.

2. **Probability Ratio:**
   The probability ratio $\mathbf{r_t}(\theta)$ is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

   where:

   - $\pi_\theta(\mathbf{a_t}|\mathbf{s_t})$ represents the probability of taking action $\mathbf{a_t}$ in state $\mathbf{s_t}$ under the current policy parameterized by θ.
   - $\pi_{\theta_{\mathbf{old}}}(\mathbf{a_t}|\mathbf{s_t})$ represents the probability of taking action $\mathbf{a_t}$ in state $\mathbf{s_t}$ under the old policy parameterized by $\theta_{\mathbf{old}}$.

3. **Advantage Estimation:**
   The advantage estimate $\mathbf{A_t}$ quantifies the advantage of taking action $\mathbf{a_t}$ in the state $\mathbf{s_t}$ over the baseline value function. It can be calculated using various methods such as generalized advantage estimation (GAE) or Monte Carlo estimation.

4. **Clipped Surrogate Objective:**
   PPO employs a clipped surrogate objective to prevent large policy updates and ensure stability during training. The clipped objective is defined as the minimum

between two terms: the original objective and a clipped version of it.

$$J^{clip}(\theta) = \mathbb{E}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\theta, 1+\theta)A_t)]$$

where the clip function ensures that the probability ratio $\mathbf{r_t}(\theta)$ remains within a predefined range of $1-\theta$ to $1+\theta$.

By optimizing the surrogate objective function through stochastic gradient ascent, PPO iteratively updates the policy parameters to improve agent performance while maintaining stability and efficiency in learning. This mathematical framework forms the foundation of PPO's effectiveness in training deep reinforcement learning agents across diverse environments and tasks.

Advantages & Disadvantages

Advantages:

- **Sample Efficiency:** PPO exhibits efficient use of samples, leveraging the clipped surrogate objective to prevent large policy updates and stabilize training.
- **Robustness:** PPO's simplicity and robustness make it well-suited for a wide range of environments and learning scenarios, including environments with sparse rewards.
- **Scalability:** PPO's straightforward implementation and lightweight computational requirements enable scalable training across diverse environments and hardware configurations.

Disadvantages:

- **Hyperparameter Sensitivity:** PPO's performance may be sensitive to hyperparameter choices, requiring careful tuning for optimal results.
- **Limited Exploration:** While PPO encourages stable learning, it may exhibit limited exploration capabilities compared to more exploratory algorithms such as Soft Actor-Critic (SAC).
- **Potential for Suboptimal Policies:** In some cases, PPO may converge to suboptimal policies due to the conservative nature of the policy updates enforced by the clipping mechanism.

Despite these limitations, PPO remains a popular and widely-used algorithm in the field of reinforcement learning, prized for its simplicity, stability, and effectiveness in training deep reinforcement learning agents across a variety of environments and tasks.

3.6.2 Soft Actor-Critic Algorithm (SAC)

Soft Actor-Critic (SAC) emerges as a prominent algorithm in the realm of deep reinforcement learning, revered for its capacity to handle continuous action spaces with ease and efficiency. SAC operates under the framework of maximum entropy reinforcement learning, prioritizing not only the maximization of expected cumulative rewards but also the maximization of entropy in the agent's policy distribution.

At its core, SAC combines elements of actor-critic methods with the concept of entropy regularization, aiming to optimize both the policy and value functions simultaneously. SAC maintains three key components: an actor-network parameterizing the policy, a critic network estimating the value function, and a temperature parameter governing the entropy of the policy distribution. The algorithm iteratively updates these components to maximize expected rewards while maximizing policy entropy. Let's delve into the intricacies of SAC, exploring its mathematical formulations, its strengths and limitations:

Mathematical Foundations

Soft Actor-Critic (SAC) leverages the principles of maximum entropy reinforcement learning, blending policy optimization with entropy regularization to achieve robust and versatile learning in continuous action spaces. Let's delve deeper into the mathematical underpinnings of SAC:

1. **Objective Function:**
   SAC's objective function comprises two key components: the expected cumulative rewards and the entropy of the policy distribution. Formally, the objective function $\mathbf{J}(\theta)$ is defined as:

   $$J(\theta) = \mathbb{E}_{\tau:\pi_\theta}[\Sigma_{t=0}^{T} \mathcal{Y}^t (r(s_t, a_t) + \alpha \mathcal{H}(\pi_\theta(a_t|s_t)))]$$

   where:

   - τ represents a trajectory sampled from the policy.
   - π_θ denotes the policy parameterized by θ.
   - $\mathbf{r}(\mathbf{s_t}, \mathbf{a_t})$ denotes the reward obtained by taking action $\mathbf{a_t}$ in state $\mathbf{s_t}$.

- $\mathcal{Y}$ is the discount factor.
- α controls the trade-off between expected rewards and policy entropy.
- $\mathcal{H}(\pi_\theta(\mathbf{a_t}, \mathbf{s_t})$ represents the entropy of the policy distribution.

2. **Entropy Regularization:**
   The entropy term in the objective function encourages exploration by maximizing policy entropy. It is defined as:

$$\mathcal{H}(\pi_\theta(a_t|s_t)) = -\mathbb{E}_{a:\pi_\theta}[log\pi_\theta(a|s_t)]$$

   Maximizing entropy encourages the policy to be more diverse, leading to better exploration and robustness in learning.

3. **Optimization:**
   SAC optimizes the objective function using stochastic gradient ascent, adjusting the policy parameters θ to maximize expected cumulative rewards while maximizing policy entropy. This optimization process involves sampling trajectories from the policy and computing gradients with respect to the policy parameters.

4. **Temperature Parameter:**
   The temperature parameter α controls the influence of entropy regularization on the policy. By adjusting α, practitioners can tune the balance between exploration and exploitation in the learning process.

By incorporating these mathematical elements, SAC provides a principled framework for training deep reinforcement learning agents in continuous action spaces, balancing exploration and exploitation to achieve effective and robust learning outcomes.

Advantages & Disadvantages

Advantages:

- **Continuous Action Spaces:** SAC excels in handling continuous action spaces, offering a natural and efficient approach to learning in such environments.
- **Entropy Regularization:** By maximizing policy entropy, SAC encourages exploration and robustness, leading to more diverse and adaptive policies.
- **Sample Efficiency:** SAC's use of off-policy data and entropy regularization often leads to improved sample efficiency compared to other algorithms.

Disadvantages:

- **Hyperparameter Sensitivity:** SAC's performance may be sensitive to hyperparameter choices, requiring careful tuning for optimal results.
- **Complexity:** Implementing SAC may require additional computational resources and tuning efforts compared to simpler algorithms.

Despite these challenges, SAC stands as a versatile and powerful algorithm in the arsenal of deep reinforcement learning methods, offering a principled and effective approach to training agents in continuous action spaces with exploration and efficiency.

3.6.3 PPO Vs SAC

In reinforcement learning, Proximal Policy Optimization (PPO) stands out as simplicity, stability, and scalability. Developed as a robust and accessible algorithm for training agents in complex environments, PPO has garnered widespread acclaim for its straightforward implementation and reliable performance across a diverse array of tasks. While competing methodologies such as Soft Actor-Critic (SAC) offer theoretical advantages, the practical superiority of PPO is evident in its ease of use, stable training dynamics, and minimal hyperparameter sensitivity. In this comparative analysis, we delve into the strengths and weaknesses of both PPO and SAC, ultimately advocating for the adoption of PPO as the preferred choice for reinforcement learning applications.

PPO over SAC

Proximal Policy Optimization (PPO) Advantages:

- **Simplicity:** PPO boasts a straightforward implementation, making it accessible even to beginners in reinforcement learning.
- **Stability:** With its clipped surrogate objective function, PPO offers stable training dynamics, reducing the likelihood of divergence or oscillation during training.
- **Scalability:** PPO scales seamlessly to complex environments and large-scale systems, demonstrating consistent performance across a wide range of tasks.
- **Ease of Tuning:** Its minimal hyperparameter sensitivity and intuitive tuning process streamline experimentation and deployment, saving valuable time and resources.

Soft Actor-Critic (SAC) Disadvantages:

- **Complexity Overload:** SAC's reliance on entropy regularization and maximum entropy objectives introduces additional complexity to the training process, requiring meticulous hyperparameter tuning and careful implementation.
- **Sensitivity Strain:** The intricate interplay of hyperparameters in SAC can make it challenging to achieve optimal performance, often resulting in suboptimal results and increased computational overhead.
- **Exploration Encumbrance:** While SAC offers enhanced exploration through entropy regularization, this comes at the cost of increased computational complexity and training time, limiting its practical applicability in real-world scenarios.

Contrast and Comparision

Feature	**PPO**	**SAC**
Policy Optimization	Clipped surrogate objective function	Maximum entropy objective with entropy regularization
Exploration	Limited exploration capabilities	Utilizes entropy regularization for exploration
Sample Efficiency	Moderate	High
Hyperparameter Sensitivity	Less sensitive	More sensitive
Complexity	Simple and easy to implement	More complex and requires meticulous tuning
Robustness	Stable training dynamics	Increased complexity may lead to instability
Theoretical Guarantees	Lacks theoretical guarantees	Provides theoretical guarantees on policy improvement and optimality

Table 3.1: Comparing PPO and SAC on different features

Promoting PPO over SAC

In conclusion, while Soft Actor-Critic may offer theoretical benefits, the practical advantages of Proximal Policy Optimization are undeniable. PPO's simplicity, stability, and scalability make it the clear choice for researchers and practitioners seeking a reliable and efficient reinforcement learning algorithm. By prioritizing ease of implementation, robust training dynamics, and minimal hyperparameter sensitivity, PPO

emerges as the superior option for a wide range of applications, empowering users to achieve consistent results with minimal effort and computational overhead. Hence, we have chosen PPO for iPark.

3.6.4 Hyperparameters & Trainer Configuration

Hyperparameters are external configuration variables that data scientists use to manage machine learning model training. Sometimes called model hyperparameters, the hyperparameters are manually set before training a model. They're different from parameters, which are internal parameters automatically derived during the learning process and not set by data scientists.

Examples of hyperparameters include the number of nodes and layers in a neural network and the number of branches in a decision tree. Hyperparameters determine key features such as model architecture, learning rate, and model complexity.

```yaml
behaviors:
    default:
        trainer_type: ppo
        hyperparameters:
            batch_size: 512
            buffer_size: 5120
            learning_rate_schedule: linear
            learning_rate: 3.0e-4
        network_settings:
            hidden_units: 512
            normalize: false
            num_layers: 4
            vis_encode_type: simple
            memory:
                memory_size: 512
                sequence_length: 512
        max_steps: 10.0e5
        time_horizon: 64
        summary_freq: 10000
        reward_signals:
            extrinsic:
                strength: 1.0
                gamma: 0.99
    iPark:
        trainer_type: ppo
        hyperparameters:
            batch_size: 512
            buffer_size: 5120
        network_settings:
            hidden_units: 512
            num_layers: 4
        max_steps: 10.0e6
        time_horizon: 128
```

Figure 3.19: `trainer_config.yaml` file containing Hyperparameters of an iPark Model

Now let's take a look at the `config` variables and hyperparameters provided by the Unity ML-Agents Framework. These are generally configured or changed in a file named `trainer_config.yaml`. When initiating training, this file is passed to the ML-Agents and RL model behaviour depends on these.

Common Trainer Configurations

Setting	Description
`trainer_type`	(default = `ppo`) The type of trainer to use: `ppo`, `sac`, or `poca`
`summary_freq`	(default = `50000`) Number of experiences that need to be collected before generating and displaying training statistics. This determines the granularity of the graphs in Tensorboard
`time_horizon`	(default = `64`) How many steps of experience to collect per-agent before adding it to the experience buffer. When this limit is reached before the end of an episode, a value estimate is used to predict the overall expected reward from the agent's current state. As such, this parameter trades off between a less biased, but higher variance estimate (long time horizon) and more biased, but less varied estimate (short time horizon). In cases where there are frequent rewards within an episode, or episodes are prohibitively large, a smaller number can be more ideal. This number should be large enough to capture all the important behaviour within a sequence of an agent's actions. Typical range: `32 - 2048`
`max_steps`	(default = `500000`) Total number of steps (i.e., observation collected and action taken) that must be taken in the environment (or across all environments if using multiple in parallel) before ending the training process. If you have multiple agents with the same behaviour name within your environment, all steps taken by those agents will contribute to the same `max_steps` count. Typical range: `5e5 - 1e7`

Setting	Description
keep_checkpoints	(default = 5) The maximum number of model checkpoints to keep. Checkpoints are saved after the number of steps specified by the checkpoint_interval option. Once the maximum number of checkpoints has been reached, the oldest checkpoint is deleted when saving a new checkpoint.
even_checkpoints	(default = false) If set to true, ignores checkpoint_interval and evenly distributes checkpoints throughout training based on keep_checkpoints and max_steps, i.e. checkpoint_interval = max_steps / keep_checkpoints. Useful for cataloguing agent behaviour throughout training.
checkpoint_interval	(default = 500000) The number of experiences collected between each checkpoint by the trainer. A maximum of keep_checkpoints checkpoints are saved before old ones are deleted. Each checkpoint saves the .onnx files in results/ folder.
init_path	(default = None) Initialize trainer from a previously saved model. Note that the prior run should have used the same trainer configurations as the current run, and have been saved with the same version of ML-Agents. You can provide either the file name or the full path to the checkpoint
threaded	(default = false) Allow environments to step while updating the model. This might result in a training speedup, especially when using SAC. For best performance, leave setting to false when using self-play.
hyperparameters -> learning_rate	(default = 3e-4) Initial learning rate for gradient descent. Corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable, and the reward does not consistently increase.

Setting	Description
hyperparameters -> batch_size	Number of experiences in each iteration of gradient descent. This should always be multiple times smaller than buffer_size. If you are using continuous actions, this value should be large (on the order of 1000s). If you are using only discrete actions, this value should be smaller (on the order of 10s).
hyperparameters -> buffer_size	(default = 10240 for PPO and 50000 for SAC) PPO: Number of experiences to collect before updating the policy model. This corresponds to how many experiences should be collected before we do any learning or updating of the model. This should be multiple times larger than batch_size. Typically a larger buffer_size corresponds to more stable training updates. SAC: The max size of the experience buffer - on the order of thousands of times longer than your episodes, so that SAC can learn from old as well as new experiences.

Table 3.2: Common Trainer Configurations [Git-URI]

PPO-specific Configurations

Setting	Description
hyperparameters -> beta	(default = 5.0e-3) Strength of the entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. Increasing this will ensure more random actions are taken. This should be adjusted such that the entropy (measurable from TensorBoard) slowly decreases alongside increases in reward. If entropy drops too quickly, increase beta. If entropy drops too slowly, decrease beta

Setting	Description
`hyperparameters -> epsilon`	(default = `0.2`) Influences how rapidly the policy can evolve during training. Corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Setting this value small will result in more stable updates, but will also slow the training process.
`hyperparameters -> beta_schedule`	(default = `learning_rate_schedule`) Determines how beta changes over time. `linear` decays beta linearly, reaching 0 at max_steps, while `constant` keeps beta constant for the entire training run. If not explicitly set, the default beta schedule will be set to `hyperparameters -> learning_rate_schedule`.
`hyperparameters -> lambd`	(default = `0.95`) Regularization parameter (lambda) used when calculating the Generalized Advantage Estimate (GAE). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. Low values correspond to relying more on the current value estimate (which can be high bias), and high values correspond to relying more on the actual rewards received in the environment (which can be high variance). The parameter provides a trade-off between the two, and the right value can lead to a more stable training process.
`hyperparameters -> num_epoch`	(default = `3`) Number of passes to make through the experience buffer when performing gradient descent optimization. The larger the batch_size, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning.
`hyperparameters -> shared_critic`	(default = `False`) Whether or not the policy and value function networks share a backbone. It may be useful to use a shared backbone when learning from image observations.

Table 3.3: PPO-Specific Configurations `[Git-URI]`

Chapter 4

System Overview & Architecture

4.1 Brief Introduction

The project is a comprehensive undertaking aimed at developing a sophisticated system. The system's primary objective is to enable vehicles to navigate and park autonomously, specifically addressing the intricacies of urban parking scenarios. This endeavour is facilitated by Reinforcement Learning, Unity Engine, Unity ML-Agents, and specifically the PPO algorithm, all explained in detail in the previous chapter.

In this chapter, we will take a look at the solution itself. The environment, concept, components, assets (environment models) used, scripts, evaluation, testing, and deployment process in detail.

4.2 Project Environment

The successful development and deployment of any software project hinge upon a robust and well-configured development environment. In the realm of game development and machine learning, where precision, performance, and collaboration are paramount, the choice of tools and technologies plays a pivotal role in shaping project outcomes. In this section, we delve into the foundational components of our project environment, encompassing the versatile Unity Engine, the expressive C# programming language, the powerful Visual Studio integrated development environment (IDE), and the collaborative prowess of GitHub version control. Together, these elements form the bedrock of our development ecosystem, empowering us to harness the full potential of modern software engineering practices and deliver innovative solutions at the intersection of gaming and artificial intelligence.

4.2.1 Unity Engine

We have used Unity3D Game Engine (Chap2:Sect5) for the simulation of our virtual parking scenarios, NPC cars, Car-Agent, evaluation and deployment purposes. Additionally, it is responsible for the optimized working of Unity ML-Agents and the whole training and testing process of the RL Agent. Now let's delve into some project-specific details:

Project Setup:

- **Unity Version:** 2021.3.30f1
- **Scenes:**
    1. **Main Menu:** This scene oversees the majority of user interaction. It's the initial load and offers users the choice to commence the simulation, modify settings and models, and exit the application.
    2. **Training:** This scene is primarily employed for RL Agent training. It comprises 16 parking scenarios, with each scenario constantly evolving with every episode, yet all agents operate synchronously. This accelerates the training process while maintaining the dynamic essence of the model. It's integrated within the Unity Engine and remains unexported in the final application.
    3. **FinalScene:** This serves as the central hub within the application, showcasing the project to the user. Here, various dynamic parking scenarios are simulated, akin to those in the Training Scene, but with agents undergoing testing. Additionally, users can calculate efficiency, facilitated by the corresponding user interface.
- **Models Created:**
    - **Development Models**
        1. *iPark [01] 21-11-2023:* Trained on 21st November for 4hrs. Ended with the reward value of 0.7201
        2. *iPark [02] 28-03-2024:* Trained on 28th March for 3hrs 46min. Ended with the reward value of 0.4385
        3. *iPark [03] 29-03-2024:* Trained on 29th March for 4hrs 33min. Ended with the reward value of 0.7329
        4. *iPark [04] 30-03-2024:* Trained on 30th March for 3hrs 56min. Ended with the reward value of 0.5327

- **Exported Models**
  1. *All Development Models*
  2. *iPark Export [01] 04-05-2024:* Trained on 04th May for 3hrs 48min. Ended with a reward value of 0.5048
  3. *iPark Export [02] 06-05-2024:* Trained on 06th May for 3hrs 28min. Ended with a reward value of 0.5269
  4. *iPark Export [03] 08-05-2024:* Trained on 08th May for 3hrs 36min. Ended with a reward value of 0.7008

- **Unique Assets:**
  1. **Chevrolet Corvette 1980** by *Yarkov DevBlog*. `[SketchFab-URI]`
  2. **Low Poly Tree Pack** by *louieoliva*. `[SketchFab-URI]`

- **C# Scripts Created:**
  1. `AutoParkAgent.cs` : The C# script responsible for the Agent Behaviour, rewards and working of `SimulationManager.cs` .
  2. `ButtonsController.cs` : The C# script responsible for the working for all UI Buttons.
  3. `CameraLook.cs` : The C# script responsible for the camera behaviour in the FinalScene.
  4. `CameraMovement.cs` : The C# script responsible for the camera behaviour in the FinalScene.
  5. `CarController.cs` : The C# script responsible for the Car behaviour and actions such as Drive, Brake, and Steer. it is primarily used by `AutoParkAgent.cs` .
  6. `EfficiencyCal.cs` : The C# script responsible for the calculation of efficiency on per-agent basis.
  7. `EfficiencyCombined.cs` : The C# script responsible for the combined efficiency of all agents through the use of `EfficiencyCal.cs`.
  8. `ParkingLot.cs` : The C# script responsible for rewarding and penalising the agent based on parking. They work with the help of a logical collider.
  9. `SimulationManager.cs` : The C# script responsible for the management and working of the simulation including initiating the episode and changing the scenarios. it is primarily used by `AutoParkAgent.cs`.

- **Packages (In addition to default packages):**
    1. **Cinemachine:** Unity provided package used for advanced camera movement. It is mainly used in the Main Menu dolly camera behaviour.
    2. **TextMeshPro:** Unity provided package used for high quality UI. It is used in all of the UI elements and components.
- **Total Size:** `3.6 GB` including `results` directory.
- **Targeted Platform:** Windows

4.2.2 C-Sharp (C#) Programming Language

C# (pronounced "C sharp") stands as a cornerstone in modern software development, particularly in the realm of game development. Developed by Microsoft, C# is renowned for its versatility, performance, and robustness. It serves as the primary programming language within the Unity Engine, a leading game development platform used by developers worldwide. C# brings a blend of simplicity and power to the table, enabling developers to craft complex applications with relative ease.

As an object-oriented language, C# emphasizes modularity and extensibility, allowing developers to organize their code into reusable components. Its syntax is clean and intuitive, making it accessible to newcomers while offering advanced features for seasoned developers. One of C#'s standout features is its integration with the .NET Framework, providing access to a vast ecosystem of libraries and tools for various programming tasks.

For game development, C# offers a perfect balance between performance and productivity. Its seamless integration with Unity's API (Application Programming Interface) empowers developers to create immersive gaming experiences with relative ease. From defining game mechanics to implementing AI behaviours, C# serves as the backbone of Unity game development, enabling developers to bring their creative visions to life.

In summary, C# plays a pivotal role in the project environment, serving as the primary programming language for developing game logic, behaviours, and interactions within the Unity Engine. Its versatility, performance, and integration capabilities make it an indispensable tool for game developers, driving innovation and creativity in the gaming industry.

A list of all C# script files created by us:

(for details refer to 3.2.1 [Unity Engine]:[Project Setup]:[C# Scripts Created])

1. `AutoParkAgent.cs`
2. `ButtonsController.cs`
3. `CameraLook.cs`
4. `CameraMovement.cs`
5. `CarController.cs`
6. `EfficiencyCal.cs`
7. `EfficiencyCombined.cs`
8. `ParkingLot.cs`
9. `SimulationManager.cs`

4.2.3 Visual Studio

Visual Studio serves as the primary Integrated Development Environment (IDE) for scripting game logic and behaviours within the Unity Engine ecosystem. Developed by Microsoft, Visual Studio offers a comprehensive suite of tools and features specifically tailored to enhance productivity and streamline the development workflow for Unity developers. Its seamless integration with Unity Editor provides a powerful environment for writing, debugging, and deploying C# scripts, enabling developers to create rich and engaging gaming experiences with ease.

Visual Studio's robust feature set empowers Unity developers to write high-quality code efficiently. Its advanced code editing capabilities, including IntelliSense code completion, code navigation, and syntax highlighting, facilitate rapid development and ensure code accuracy. Developers can leverage Visual Studio's integrated debugging tools to identify and fix issues in their scripts, with real-time error highlighting and breakpoint support providing invaluable assistance in the debugging process.

4.2.4 GitHub

GitHub serves as a popular platform for version control and collaborative development, making it an ideal choice for sharing Unity projects among teams. By hosting Unity projects on GitHub repositories (`[iPark-GitHub-URI]`), developers can easily collaborate on code, assets, and scenes, ensuring seamless coordination and version management. GitHub's version control features allow developers to track changes, merge contributions, and resolve conflicts efficiently, facilitating a smooth and organized development process. With its support for branching and pull requests, GitHub enables teams to work on different features or fixes simultaneously while maintaining a centralized repository of the project's history. Overall, GitHub provides a reliable and efficient solution for sharing Unity projects, fostering collaboration and enhancing productivity among developers.

4.3 Project Concept

4.3.1 Working Concept

The project can work in 2 modes inside the Unity Editor: Training and Testing. We will look at how it works in both modes. In the exported application, it can only work in the Testing phase as Unity does not provide training features in build applications.

Unity Editor: Training Mode

During Training mode, the agent operates without any Neural Network guidance, as illustrated in the figure below, initiating the training process.

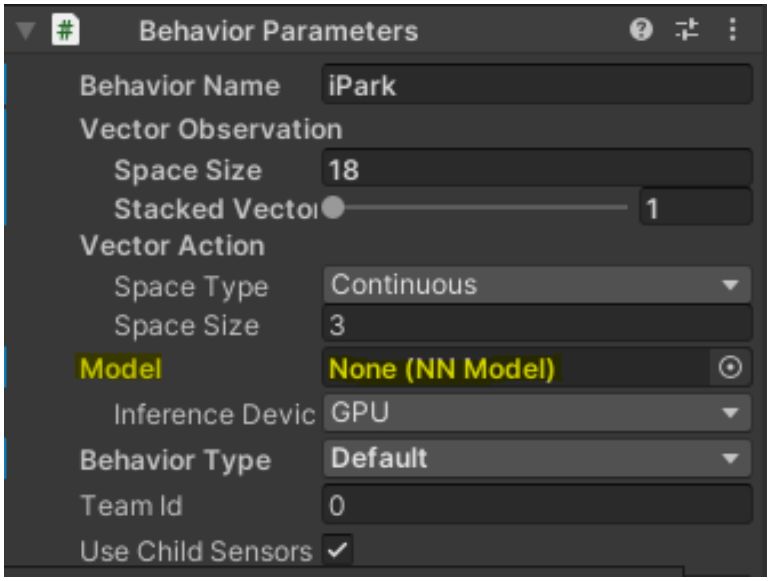

Figure 4.1: Neural Network is not required and not passed while Training

To commence training, the 4.7.1 command must be executed, followed by pressing play in the Unity Editor. Subsequently, the training process launches, with both Unity ML-Agents and the Unity Editor collaborating to train the model. The behaviour during this process is defined by the `trainer_config.yaml`, which dictates actions such as periodic Neural Network exports and checkpoint creation. These actions are crucial for preserving progress and generating reports for tensorboard, providing statistics like Extrinsic Reward, Policy Loss, Episode Length, and more.

During the training process, the RL agent interacts with both the RL training module and the Simulation Environment module to facilitate learning. The agent receives state information from the environment and rewards from the training module. Subsequently, it executes actions within the simulation environment, which in turn generates

rewards and passes relevant statistics to the TensorBoard or evaluation module. Based on these rewards, the RL training module adjusts the agent's settings and neural network properties, optimizing its behaviour for the task at hand. This iterative process of action execution, reward collection, and parameter adjustment repeats for a predetermined number of training steps, typically spanning millions of iterations, to effectively train the agent's decision-making capabilities.

Unity Editor: Testing Mode

During Testing mode within the Unity Editor, a Neural Network is essential, as depicted in the figure below.

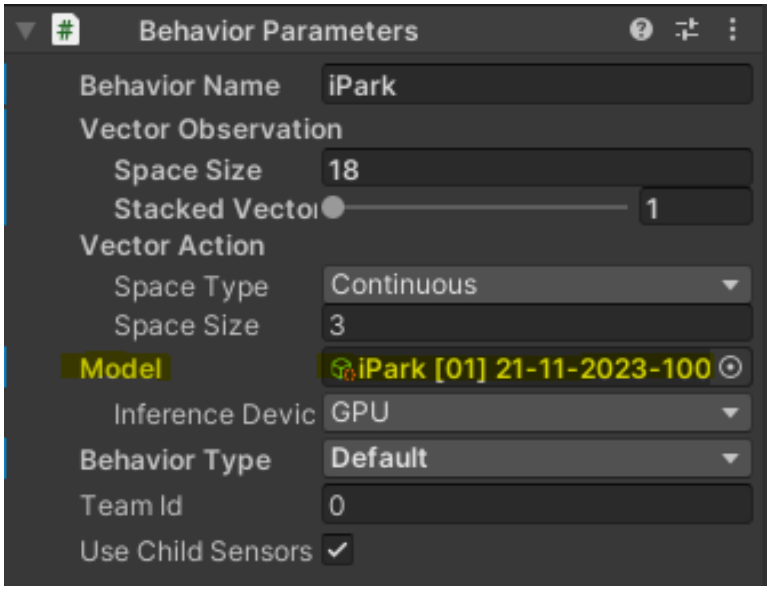

Figure 4.2: Neural Network is required and passed while Testing

This Neural Network, or model, is subsequently put to the test in various parking scenarios. The performance assessment occurs agent by agent, facilitated by `EfficiencyCal.cs`, and collectively for all agents, managed by `EfficiencyCombined.cs`. This process persists endlessly and can only be halted by selecting "Stop" in the Unity Editor.

Throughout testing, the NN interacts solely with the simulation environment, executing actions based on its learned policy. In response, the simulation environment communicates the outcomes of these actions to the evaluation module. This iterative testing procedure provides insights into the NN's effectiveness in navigating dynamic parking environments and guides further refinements in its decision-making capabilities.

The flow chart below shows the working of the project in the Unity Editor in both modes.

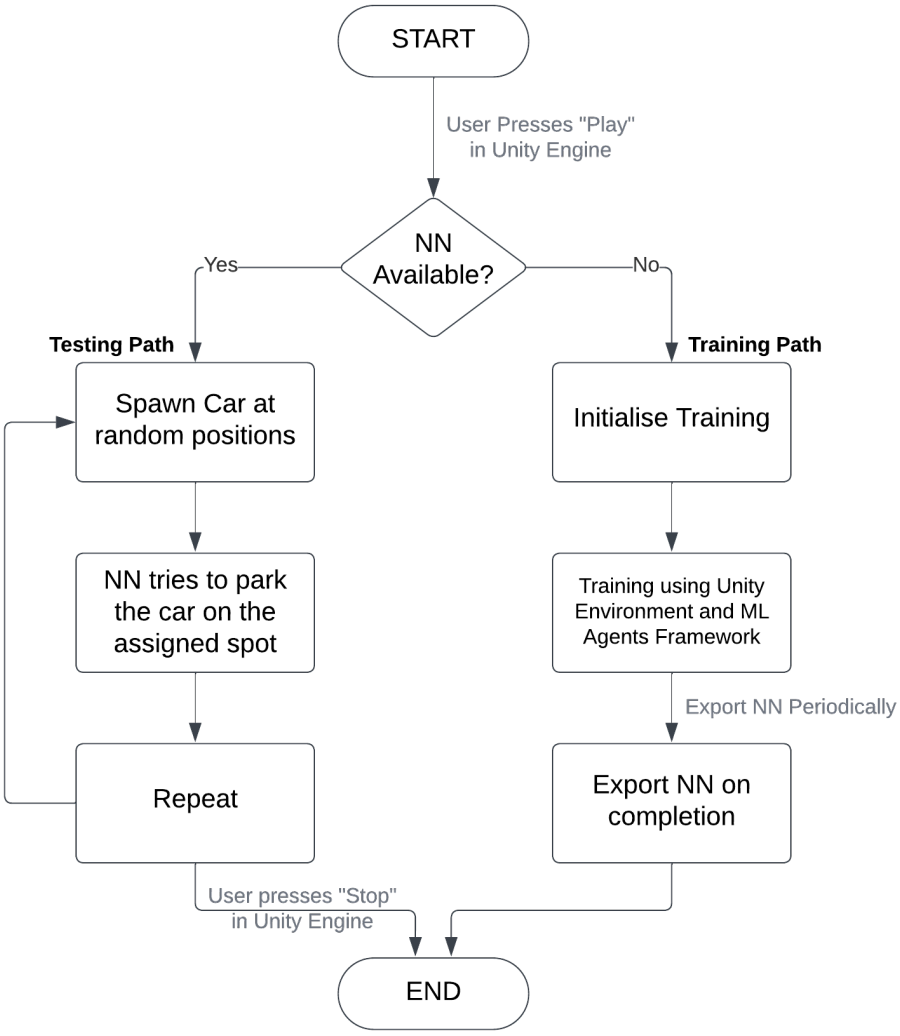

Figure 4.3: Flow chart showing the working in Unity Editor

Exported Application: Testing Phase

In the exported application, testing begins when the user selects “Start” and chooses a model from the list provided. We’ve included a total of 7 models, comprising 4 development models and 3 export models, all saved on the drive. Following this selection, the “Final Scene” is loaded, functioning akin to “Unity Editor: Testing Mode”. Moreover, users can opt to “Reset” the scene, “Go Back” to the main menu to try another model, or “Quit” the application.

The flow chart below shows the general working of the project in the exported application.

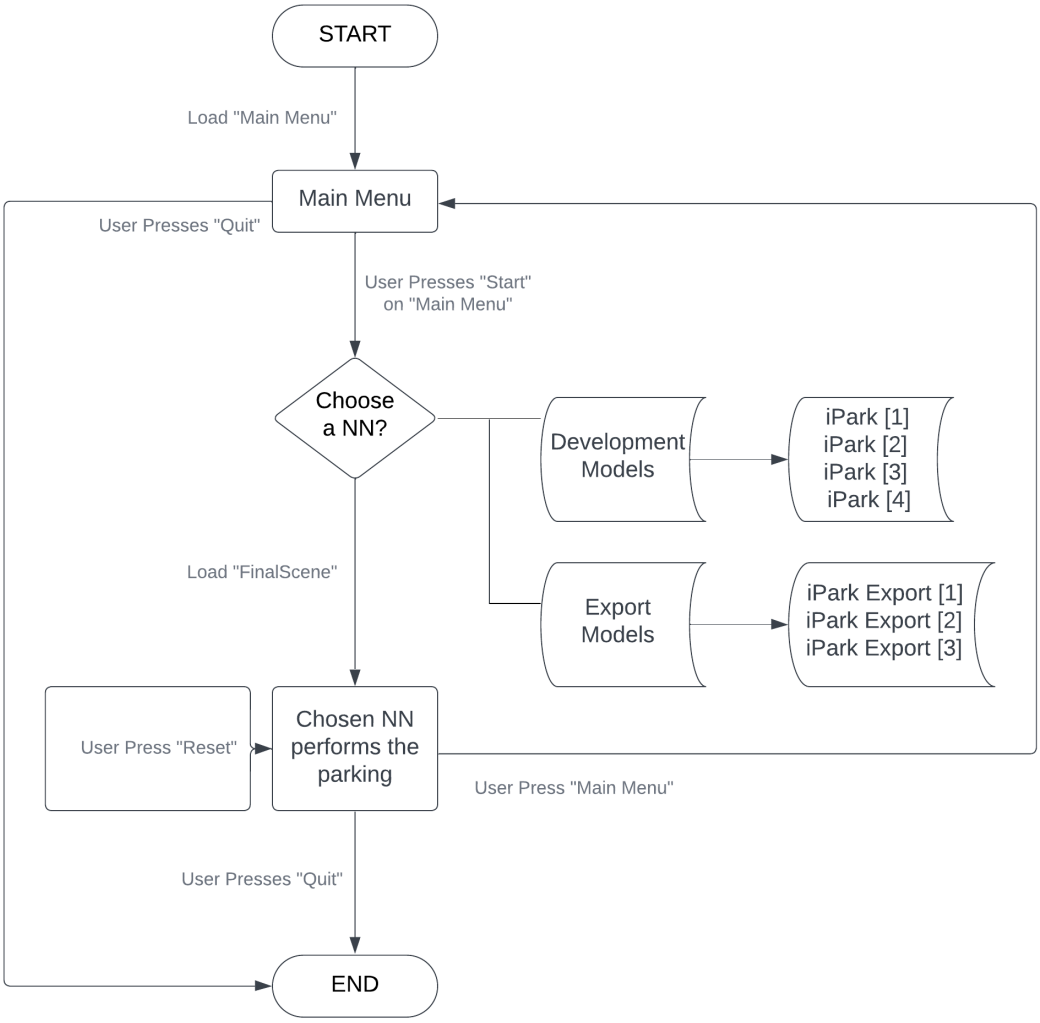

Figure 4.4: Flow chart showing the working in exported Application

The flow chart don't have the following functions for the sake of simplicity:

1. **Developer Evaluation in Main Menu:** It links to an external file displaying the evaluation statistics noted by the developers.

2. **Evaluation in Final Scene:** The Final Scene has the option to calculate the combined efficiency of all the acting agents at any point in time. It can be triggered by the user upon pressing "E".

4.3.2 Design & Development of Components

This subsection is focused on the design and development of the system components crucial for the project. We will start by providing details on each component.

RL Model Training Component

The RL Model Training Module serves as the cornerstone for preparing the reinforcement learning model. It interfaces with key components such as the Simulation Environment Module and the UI Module. This module encapsulates the logic for training the model using Unity's simulation environment, ensuring exposure to various parking scenarios. This module is responsible for training the reinforcement learning model. it utilises Unity's simulation environment to simulate diverse parking scenarios and provide a real-world environment to the Agent. It also uses the ML-Agents Framework for the actual training process and Neural Network Creation.

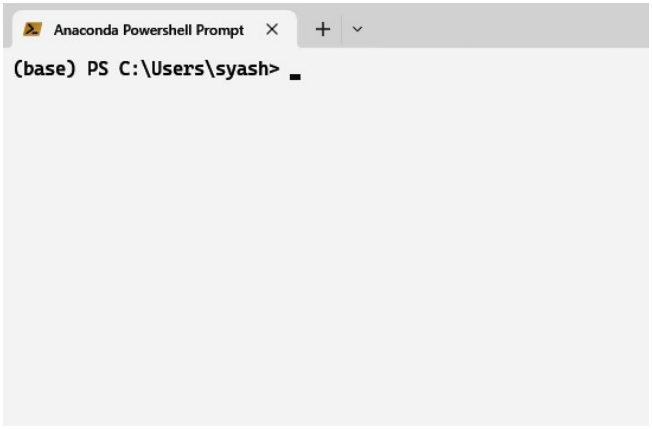

Figure 4.5: Anaconda Command Prompt

The Model training can be initiated by first activating the correct anaconda environment in the anaconda command prompt. The environment name is `mlagents` (not to confuse with Unity ML-Agents) in our case.

```
#activate an anaconda environment: conda activate "env-
   name"
    conda activate mlagents
```

Following that, we need to order the `unity-mlgents` framework training by the following command:

```
#start unity-mlagents training
    mlagents-learn --run-id="modelNameOrId"
```

The `run-id` specifies the name of the model directory, it should be unique. If the `run-id` passed already exists, the command can be appended with `--resume` or `--force` to resume or reset the present information respectively. Upon successfully executing this, Fig 3.17 will show up which will inform us to press "Play" in the unity editor to

start the training phase.

Input: User command for the initialisation alongside Scene information from the Unity Engine.
Output: It outputs a trained RL Model, precisely a Neural Network with programmed properties and behaviour.
Interaction: Communicates with the Simulation Environment Module for state information and Car properties and actions.

RL Model Component

The trained RL Model embodies the learned behaviours, enabling the autonomous decision-making process during parking manoeuvres. This component communicates with the UI Module to display real-time visualizations of the parking process in order to provide feedback to the user. It is an outcome of the Training phase and can only be used by us after training is completed. This module encompasses the core functionality of the system, housing the Reinforcement Learning model responsible for decision-making in the car parking task.

It is stored in the `results` directory with the name of the passed `run-id`. The image below shows the `results` directory of iPark with all development and one export model(s).

Input: User Commands for the initialisation and state information from the Simulation Environment Module.
Output: It outputs Parking decisions and visualizations for the User.
Interaction: Communicates with the Simulation Environment Module visualisation and decisions, It also interacts with the performance metric module for statistics.

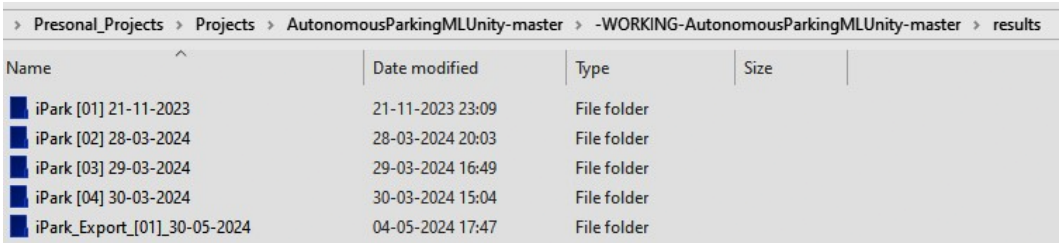

Figure 4.6: Results directory of iPark

Simulation Environment Component

Leveraging the capabilities of Unity, the Simulation Environment Module creates a dynamic and realistic parking simulation. It plays a crucial role in the training phase, providing real-time state information to the RL Model. This module encapsulates the environmental aspects essential for effective model training. It is also crucial during the testing phase as it provides a dynamic parking scenario. During both testing and training, it communicates with the evaluation module to provide details on model statistics.

Input: None (internal, pre-programmed process)
Output: It outputs visualizations and information for the RL Model.
Interaction: It interfaces with the RL Model Training Module to provide training data.

Figure 4.7: Simulation Environment of iPark

User Interface (UI) Component

The User Interface (UI) component serves as a pivotal element across various scenes, including the main menu and final scene, facilitating seamless interaction and navigation within the application. Its primary functionalities encompass scene loading, evaluation display, and user interaction management. The UI component initiates the testing of the model by loading the final scene, where the trained agent's performance is assessed in dynamic parking scenarios. Additionally, it provides users with the option to access the developer evaluation page via the main menu, enabling them to gain insights into the training and testing processes. In the final scene, the UI component offers real-time evaluation feedback, allowing users to monitor the agent's performance

dynamically. Moreover, it ensures a streamlined user experience by incorporating buttons for returning to the main menu and exiting the application, thereby closing the interaction loop and enhancing usability across all scenes.

Input: User commands for initiating and monitoring testing.
Output: It outputs visualizations, feedback, and information about the RL Model via UI elements.
Interaction: It interfaces with the RL Model, initiates testing, and displays results with the interaction from the performance metric module.

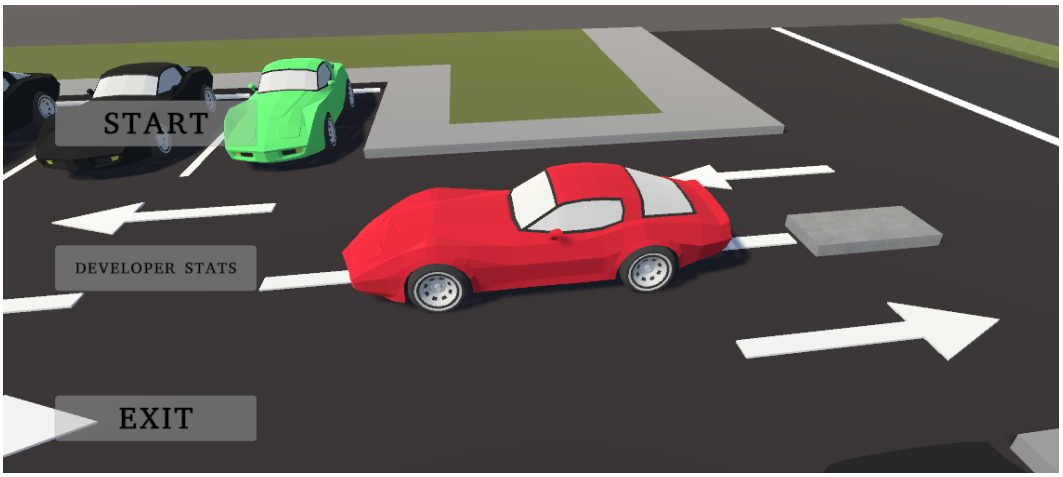

Figure 4.8: UI present on the Main Menu

Performance Metrics Component

The Performance Metrics Component plays a pivotal role in monitoring the efficiency and effectiveness of the trained model throughout both the training and testing phases. It receives performance data directly from the simulation environment, capturing key metrics such as training progress, agent success rates, and evaluation scores. Through seamless interaction with the UI component, it displays this information to users, providing real-time insights into the model's performance dynamics. Furthermore, the Performance Metrics Component is tasked with storing this data persistently while the application is running, ensuring that valuable performance metrics are retained for analysis and review. Upon exiting the application, it consolidates the accumulated data and stores it in a text file, facilitating further analysis and informing future iterations of model training and testing. This comprehensive approach to performance monitoring and data management enhances the transparency and effectiveness of the training and evaluation processes, empowering users to make informed decisions and optimizations to the trained model.

Input: Data from the Simulation Environment Module.
Output: It outputs Performance metrics data through UI elements. It also creates a text file which user can access afterwards.
Interaction: It evaluates and stores metrics for analysis by interacting with simulation environment. It also interacts with the UI component in order to display the values. Lastly, it communicates with the Windows File Manager, in order to create the text file.

```
2024-03-29 02:47:52 (Training Model 28/03)
Efficiency 74.95232%
29-03-2024 16:49:04
Efficiency 58.30884%
29-03-2024 18:10:13
Efficiency 76.28129%
30-03-2024 13:48:07
Efficiency 84.70142%
30-03-2024 14:46:54 (Training Model 29-03)
Efficiency 73.33334%
30-03-2024 23:32:12 (Training Model 30-03)
Efficiency 79.63393%

Efficiency 78.97898%
31-03-2024 00:06:38 (Testing Model)

31-03-2024 00:11:18 (Testing Model)
Efficiency 80.34483%
02-04-2024 01:14:52 (Testing Model)
Efficiency 79.93255%
```

Figure 4.9: An example `Efficiency.txt` file created by the Performance Metrics Component

4.3.3 Amalgamation of Components

Now we will see how these components work cohesively for the Training and Testing task.

Training

In the Training phase, the initiation is primarily managed by the Unity Editor and Unity ML-Agents. Once initiated, the RL Training module takes charge of reward management, model updates, and transmitting essential information to ML-Agents. Within the Simulation Environment, the RL model executes actions, while the environment relays state information back to the RL training module for reward computation and model updates. Additionally, the environment provides parking percentage or efficiency data to the performance metrics module, constituting the Training Efficiency of the model. The performance metrics module is tasked with preserving this data and exporting it to a text file. It also receives model statistics from ML-Agents, albeit

exclusively for Tensorboard purposes.

The figure below shows a data flow chart of this process.

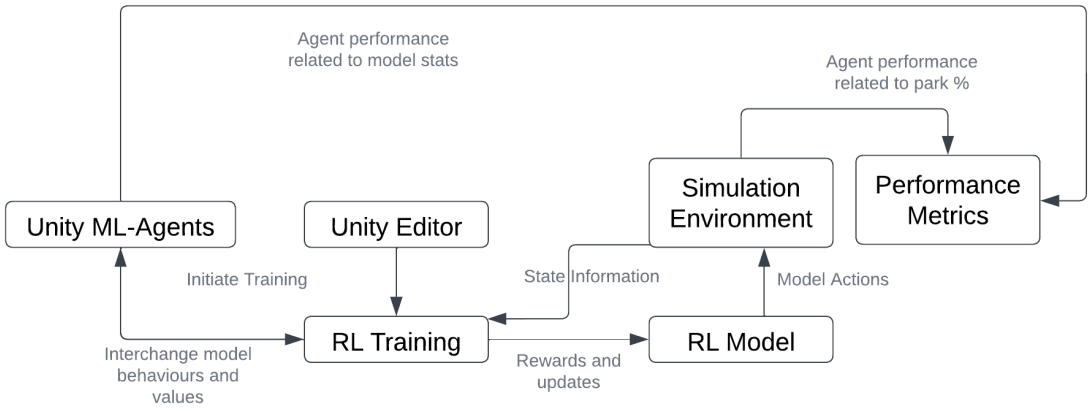

Figure 4.10: Components working during Training

Testing

The testing phase is triggered by the UI component. Upon launching the application, the main menu appears, and when the user selects "Start" and chooses a model from the list, it triggers the loading of the "Final Scene", marking the commencement of testing. During testing, the RL model executes actions within the simulation environment, which, notably, doesn't interact with the RL Training component. Instead, the environment directly provides new dynamic scenarios to the model. Furthermore, the simulation environment communicates with the performance metrics component for evaluation purposes. This cyclic process continues until the user decides to quit the application or return to the main menu.

The figure below shows a data flow chart of this process.

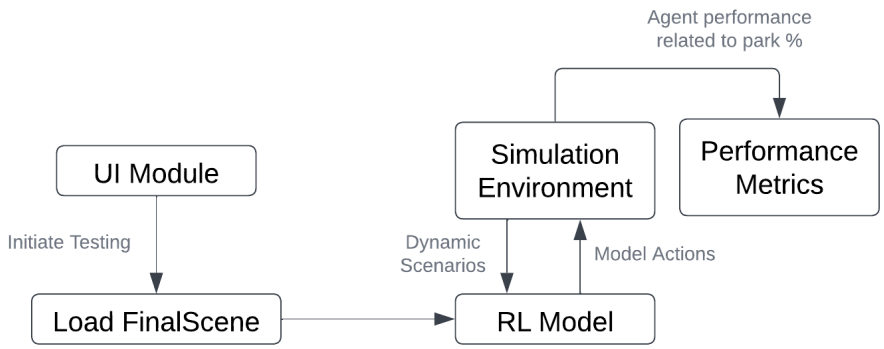

Figure 4.11: Components working during Training

4.4 Assets (Environment-Models) Used

We have used only 2 external assets or environment-models for this project. Now, we will discuss on them.

1. **Chevrolet Corvette 1980** by *Yarkov DevBlog*: It is the car model we have used for Agent and the NPC cars. Its available on SketchFab `[SketchFab-URI]`.

2. **Low Poly Tree Pack** by *louieoliva*: It is rest of the environment models (trees) we have utilised in the project. Its available on SketchFab `[SketchFab-URI]`.

4.5 Key Modules Overview

Let's delve into the fundamental modules of the project. A "Module" within this context refers to a Unity component or a collection of components that either represents a project system component or executes a vital task. These modules play integral roles in ensuring the functionality and success of the project.

4.5.1 Parking Environment Module

This module embodies the "Simulation Environment" system component and fulfills the task of generating dynamic parking scenarios. Moreover, it shoulders the responsibility of overseeing the simulation in accordance with the actions of the agents. Let's look at some parts of this module:

Parking Area

The "Parking Lot" serves as the central stage for the entire operation. It encompasses a designated area comprising _ parking slots. At the outset of each episode, these slots are randomly occupied by NPC cars, a process orchestrated by the `SimulationManager.cs`. Each slot is equipped with a logical collider essential for verifying whether an agent has successfully parked in it. Additionally, the orientation of the agent is recorded, facilitating the incentivization of reverse parking over conventional front parking maneuvers.

The figure below represents the parking lot with the outlined colliders.

Figure 4.12: Parking Lot

NPC Cars

The NPC Cars are the designated car models employed to occupy parking spaces in a randomized fashion within the parking lot. They are devoid of any specific logic or script but are outfitted with colliders. These colliders facilitate the detection of collisions with agents, enabling appropriate actions such as penalization in response.

The figure below shows an NPC Car with an outlined collider.

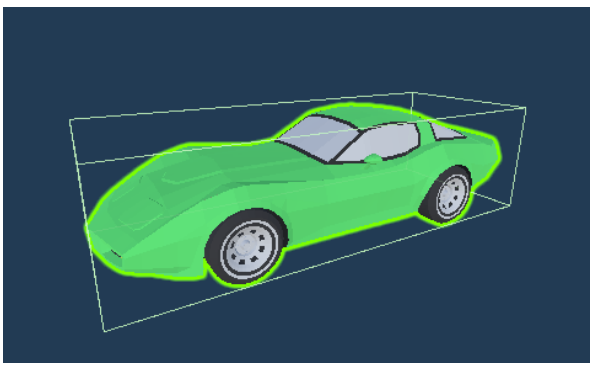

Figure 4.13: An NPC Car

Parking Slots

Parking Slots are the designated spaces within the parking lot, which can either be empty or occupied. These slots are filled randomly by a C# script, `SimulationManager.cs`. They incorporate the logic of rewarding the agent upon successful parking, facilitated by a logical collider. Each slot is equipped with its own collider and script (`ParkingLot.cs`), which orchestrates this functionality.

The figure below shows an empty parking slot with the outlined collider.

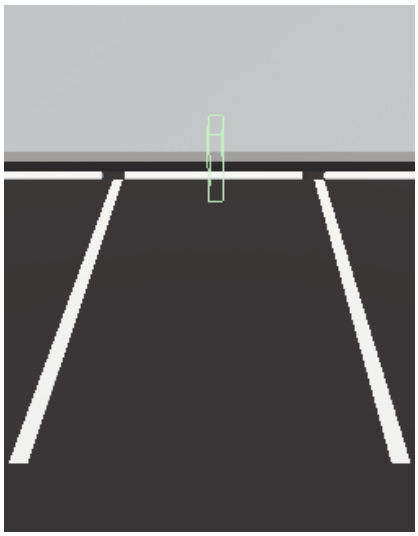

Figure 4.14: An empty parking slot

4.5.2 Car Agent

The car agent acts as the primary agent for our RL model, receiving input from Lidar sensors, specifically the `RayPerceptionSensor3D` component provided by Unity. It is equipped with a rigidbody component, a box collider, and a vehicle model. Additionally, it encompasses several scripts for various functionalities:

- `CarController.cs` script governs the actions of the car, including driving, steering, and braking.
- `AutoParkAgent.cs` script handles the actual RL agent tasks.
- `BehaviourParameters` serves as a subset of the `AutoParkAgent.cs` and is utilised for the model parameters and settings in Unity Editor.
- `SimulationManager.cs` enables interaction with the environment.

- `EfficiencyCal.cs` script is responsible for calculating the efficiency of the agent instance-wise.

The figure below shows the car agent with its Lidar sensors.

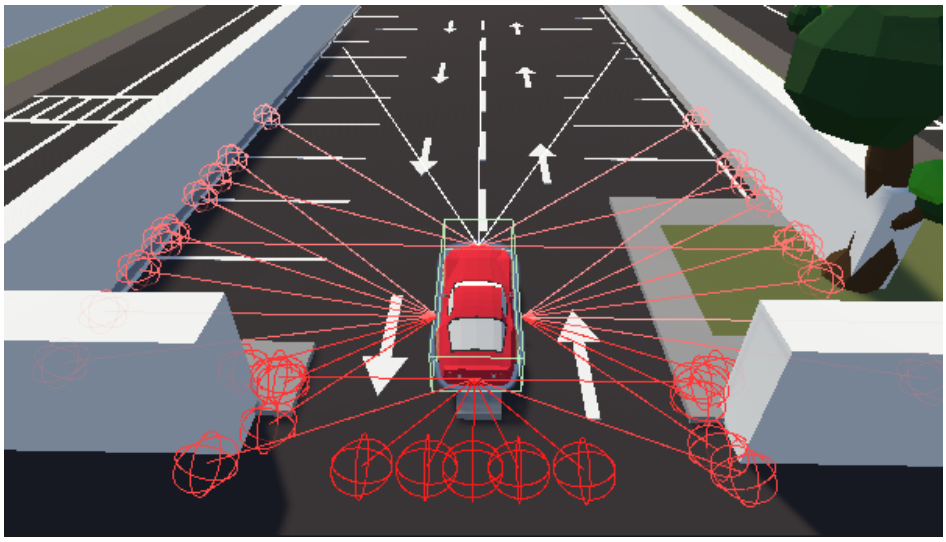

Figure 4.15: The Car Agent with input sensors

4.6 Scripts & Code

4.6.1 Visual Studio Setup

To connect Unity and Visual Studio for streamlined development, developers can follow a straightforward process facilitated by Unity's built-in integration tools. Within the Unity Editor, developers can navigate to the "Edit" menu and select "Preferences." From there, they can choose "External Tools" and designate Visual Studio as the preferred external script editor. This establishes a seamless connection between Unity and Visual Studio, enabling developers to seamlessly transition between the Unity Editor and Visual Studio IDE with a simple click. Once the connection is established, developers can proceed to set up Visual Studio for C# development and game development within the Unity environment. This involves configuring Visual Studio settings and extensions to optimize the development experience for Unity-specific requirements. Developers can start by selecting the appropriate settings and extensions tailored for Unity development within Visual Studio.

For C# development in Unity, developers can enable IntelliSense for Unity-specific APIs, ensuring accurate code completion and syntax highlighting for Unity scripting. Additionally, developers can leverage Unity-specific project templates and scaffolding

features provided by Visual Studio to expedite the creation of new Unity projects and streamline the development process. Furthermore, Visual Studio offers a range of tools and extensions designed to enhance game development workflow efficiency. Developers can access built-in debugging capabilities to troubleshoot and resolve errors effectively, as well as leverage Unity-specific documentation and resources within the Visual Studio environment to aid in the development process.

4.6.2 Environment & Car Agent Scripts

Let's take a look at the Simulation Environment and Car Agent script files. We will take a look at the key components of the script files.

`SimulationManager.cs`

```
0  //Function used to initialise the simulation
1      public void InitializeSimulation()
2      {
3          _initComplete = false;
4          StartCoroutine(OccupyParkingSlotsWithRandomCars());
5          RepositionAgentRandom();
6      }
```

The code starts with the `_initComplete` bool check for the initialisation process, then we start to occupy the parking slots randomly with the NPC cars. This is done in line 4. Lastly, we reposition the agent so that it can be trained for all possible positions.

The functions `RepositionAgentRandom()` and `OccupyParkingSlotsWithRandomCars()` position the agent and random NPC cars in the parking lot respectively based on the position, and rotation of the car and NPCs, and the occupancy of the parking slot.

The code of `RepositionAgentRandom()` and `OccupyParkingSlotsWithRandomCars()` Function can be found in the Appendix.

```
0  //Function used to reset the simulation
1      public void ResetSimulation()
2      {
3          foreach (GameObject parkedCar in parkedCars)
4          {
5              Destroy(parkedCar);
```

```
6         }
7
8         foreach (ParkingLot parkingLot in parkingLots)
9         {
10            parkingLot.IsOccupied = false;
11        }
12        parkedCars.Clear();
13    }
```

The code for resetting the simulation starts by destroying all of the parked NPC Cars (lines 3 to 6) and ends with clearing the occupying status of the parking lots (lines 8 to 11). The list `parkedCars` is also cleared so that it can be used in the next simulation cycle.

AutoParkAgent.cs

```
0  //Function used to Initialise the agent, it is called when the agent
       is first enabled
1      public override void Initialize()
2      {
3          _rigidBody = GetComponent<Rigidbody>();
4          _controller = GetComponent<CarController>();
5          _simulationManager = GetComponent<SimulationManager>();
6          _simulationManager.InitializeSimulation();
7      }
```

The code for initialising the agent is responsible for fetching the required `Rigidbody`, `CarController`, and `SimulationManager` components. it also calls the `InitializeSimulation()` method from the `SimulationManager` script.

```
0  //Function called at the start of each episode
1      public override void OnEpisodeBegin()
2      {
3          _simulationManager.ResetSimulation();
4          _simulationManager.InitializeSimulation();
5          _nearestLot = null;
6      }
```

This function is called at the start of every episode, we start with resetting the simulation environment using the `ResetSimulation()` method. Then we initialise the simulation environment in line 4 and at last, we clear the `_nearestLot` so we can store

a new instance of the `ParkingLot` class.

```
0  //Function called upon receiving actions
1      public override void OnActionReceived(float[] vectorAction)
2      {
3          _lastActions = vectorAction;
4          _controller.CurrentSteeringAngle = vectorAction[0];
5          _controller.CurrentAcceleration = vectorAction[1];
6          _controller.CurrentBrakeTorque = vectorAction[2];
7      }
```

The function `OnActionReceived()` is called when the agent receives the actions from the `DecisionRequester` subcomponent. it maps out the actions of steering, acceleration, and brakes in lines 4 to 6. It also stores these actions in `_lastActions` so they can used later.

```
0  //Function used to collect the information from the Lidar sensors
1      public override void CollectObservations(VectorSensor sensor)
2      {
3          if (_lastActions != null && _simulationManager.InitComplete)
4          {
5              if(_nearestLot == null)
6                  _nearestLot = _simulationManager.
                       GetRandomEmptyParkingSlot();
7              Vector3 dirToTarget = (_nearestLot.transform.position -
                   transform.position).normalized;
8              sensor.AddObservation(transform.position.normalized);
9              sensor.AddObservation(
10                 this.transform.InverseTransformPoint(_nearestLot.
                       transform.position));
11             sensor.AddObservation(
12                 this.transform.InverseTransformVector(_rigitBody.
                       velocity.normalized));
13             sensor.AddObservation(
14                 this.transform.InverseTransformDirection(dirToTarget));
15             sensor.AddObservation(transform.forward);
16             sensor.AddObservation(transform.right);
17             float velocityAlignment = Vector3.Dot(dirToTarget,
```

```
                _rigitBody.velocity);
18              AddReward(0.001f * velocityAlignment);
19          }
20          else
21          {
22              sensor.AddObservation(new float[18]);
23          }
24      }
```

The code above is responsible for collecting all the observations from the added Lidar sensors on the agent. it starts with confirming the simulation initialisation and actions (line 3). If yes, it collects a random empty parking lot from all available lots using the function `GetRandomEmptyParkingSlot()`. After that, we collect the observations from each sensor (lines 8 to 16).

```
0  //Function used to check agent collision with other objects
1      private void OnCollisionEnter(Collision other)
2      {
3          if (other.gameObject.CompareTag("barrier") || other.gameObject
               .CompareTag("car") ||
4              other.gameObject.CompareTag("tree"))
5          {
6              this.gameObject.GetComponent<EfficiencyCal>().collisions
                   ++;
7              this.gameObject.GetComponent<EfficiencyCal>().
                   CalEfficiency();
8
9              AddReward(-0.01f);
10             EndEpisode();
11         }
12     }
```

The `OnCollisionEnter()` method is called when the collider of the agent gets in contact with another collider. We check the tag of the other object (line 3) and if it is a barrier, other cars, or a tree we add 1 in the `collisions` count in `EfficiencyCal`, calculate the new efficiency, and penalise the agent, lines 6, 7, and 9 respectively. Lastly, the episode is ended using the `EndEpisode()` function.

```
0  //Function used to reward the agent on parking
1      public void JackpotReward(float bonus)
2      {
3          AddReward(0.2f + bonus);
4
5          EndEpisode();
6      }
```

The function above is called when the agent has successfully parked the car. It adds the rewards with any coming bonus parameter. Lastly, end the episode.

4.6.3 Evaluation System Scripts

`EfficiencyCal.cs`

```
0  //Function used to get the training or testing information
1      void Start()
2      {
3          if (this.GetComponent<BehaviorParameters>().Model)
4          {
5              isTraining = false;
6          }
7      }
```

The `Start()` method is called at the start of the first frame and checks if the agent has a model with it or not (line 3). If yes, it means that the agent is not in the training phase and will use that model for testing. So we set the `isTraining` bool to false. This will be used by the `EfficiencyCombined.cs` file, and mentioned in the output text file.

```
0  //Function used to calculate the efficiency
1      public void CalEfficiency()
2      {
3          total = parked + collisions;
4          efficiency = (parked / total) * 100;
5      }
```

This function is used to calculate efficiency using the total cases, parked cases, collided cases data, and simple percentage maths. It is called when the agent either collides or parks the car.

```
0  //Function used to get the EfficiencyCal scripts from all agent
       instances
1      private void Awake()
2      {
3          efficiencyCal = new List<EfficiencyCal>();
4          efficiencyCalGO = GameObject.FindGameObjectsWithTag("agent");
5
6          for(int i = 0; i < efficiencyCalGO.Length; i++)
7          {
8              efficiencyCal.Add(efficiencyCalGO[i].GetComponent<
                   EfficiencyCal>());
9              i++;
10         }
11     }
```

The code above is used to get all the instances of the `EfficiencyCal` in the `Efficiency Combined` script for the combined efficiency calculation. We first gather all of the agent instances in line 4 and then extract the `EfficiencyCal` script using a "for" loop, lines 6 to 10.

```
0  //Function used to calculate the final combined efficiency of all the
       agent instances and create the .txt file
1      void EfficiencyFinal()
2      {
3          int totPark = 0;
4          int totCollision = 0;
5          int totCases = 0;
6
7          trainingPhase = efficiencyCal[0].isTraining;
8
9          foreach(EfficiencyCal cal in efficiencyCal)
10         {
11             trainingPhase = trainingPhase || cal.isTraining;
12             totPark += cal.parked;
13             totCollision += cal.collisions;
14         }
15
```

```
16          totCases = totPark + totCollision;
17
18          float eff = 0;
19          eff = (totPark/(float)totCases) * 100;
20
21          if (trainingPhase)
22          {
23              txtString.Add("\n" + DateTime.Now.ToString() + "␣(Training
                    ␣Model)");
24          }
25          else
26          {
27              txtString.Add("\n" + DateTime.Now.ToString() + "␣(Testing␣
                    Model␣" + "\"" + efficiencyCal[0].GetComponent<
                    BehaviorParameters>().Model + "\"" + ")");
28          }
29          txtString.Add("Efficiency␣" + eff + "%");
30          txtString.Add("Total␣Park␣" + totPark);
31          txtString.Add("Total␣Collision␣" + totCollision);
32          txtString.Add("Total␣Cases␣" + totCases);
33          File.AppendAllLines(EffTxtFilePath, txtString);
34      }
```

The `EfficiencyFinal()` method calculates combined efficiency using the data from each `EfficiencyCal` instance. The "foreach" loop, lines 9 to 14, is used to get the data and the training bool. Then, we calculate the combined efficiency, line 19. In the end, we write the text file with training or testing phrases, efficiency data, total park data, total collision data, and total cases data, lines 21 to 33. This function is called when the user either presses "E" in the Final Scene or exits the Final Scene.

4.7 Training Process

4.7.1 Unity ML-Agents Framework Setup

The Model training can be initiated by first activating the correct anaconda environment in the anaconda command prompt. The environment name is mlagents (not to confuse with Unity ML-Agents) in our case.

```
#activate an anaconda environment: conda activate "env-name"
    conda activate mlagents
```

Following that, we need to order the `unity-mlgents` framework training by the following command:

```
#start unity-mlagents training
    mlagents-learn --run-id="modelNameOrId"
```

The `run-id` specifies the name of the model directory, it should be unique. If the `run-id` passed already exists, the command can be appended with `--resume` or `--force` to resume or reset the present information respectively. Upon successfully executing this, Fig 3.17 will show up which will inform us to press "Play" in the unity editor to start the training phase.

4.7.2 Multi Model Training

The decision to embrace multi-model training was underpinned by the acknowledgement of its potential to deliver superior outcomes over traditional single-model methodologies. By concurrently training multiple models, we unlock a nuanced understanding of how various algorithms behave across diverse contexts and scenarios. This diversity in training not only enriches our insights into algorithmic performance but also empowers us to explore a wider range of potential solutions.

Moreover, the utilization of multi-model training offers a safeguard against overfitting and model bias by leveraging the collective intelligence of multiple models. This ensemble approach fosters robustness and adaptability, enabling us to navigate complex problem spaces with greater efficacy. Through the collaborative efforts of these diverse models, we not only enhance our problem-solving capabilities but also cultivate a more comprehensive understanding of the underlying dynamics driving the task at hand. As a result, we are better equipped to address real-world challenges and optimize performance across a spectrum of applications.

A list of the models we trained for this project:

- **Development Models**

    1. *iPark [01] 21-11-2023:* Trained on 21st November for 4hrs. Ended with a reward value of 0.7201
    2. *iPark [02] 28-03-2024:* Trained on 28th March for 3hrs 46min. Ended with a reward value of 0.4385
    3. *iPark [03] 29-03-2024:* Trained on 29th March for 4hrs 33min. Ended with a reward value of 0.7329

4. *iPark [04] 30-03-2024:* Trained on 30th March for 3hrs 56min. Ended with a reward value of 0.5327

- **Exported Models**

1. *All Development Models*
2. *iPark Export [01] 04-05-2024:* Trained on 04th May for 3hrs 48min. Ended with a reward value of 0.5048
3. *iPark Export [02] 06-05-2024:* Trained on 06th May for 3hrs 28min. Ended with a reward value of 0.5269
4. *iPark Export [03] 08-05-2024:* Trained on 08th May for 3hrs 36min. Ended with a reward value of 0.7008

4.8 Model Evaluation Process

The model evaluation is overseen by the `EfficiencyCal` and `Efficiency- Combined` script files. These scripts facilitate evaluation on both a per-agent instance basis and a combined assessment of all agents. Let's delve into the specifics of each:

1. **Agent Instance:** In each agent instance, the `EfficiencyCal` script is implemented. This script is responsible for documenting key metrics during the test period, including the total number of scenarios encountered by the agent instance, the total number of successful parking manoeuvres executed by the agent instance, and the total number of collisions or unsuccessful parking attempts.

2. **Combined:** The `EfficiencyCombined` script assumes the responsibility of amalgamating the efficiencies of all agent instances. It aggregates the values of `totCases`, `totParks`, and `totCollided` obtained from all `EfficiencyCal` scripts. Subsequently, it computes the overall efficiency using the following formula:

$$\sigma = \frac{P}{S} X 100\%$$

where:

- σ: Combined Efficiency of all agent instances.
- P: Total successful parking done by the agent.
- S: Total scenarios agent performed.

4.9 Project Testing & Quality Assurance

The testing and quality assurance phase for iPark: Intelligent Parking is critical to ensure the reliability, functionality, and performance of the application. This section provides an overview of the testing plan, outlining key strategies, methodologies, and areas of focus.

4.9.1 Testing Strategy

Testing Objectives The primary objectives of testing iPark include:

1. Validating the correct functionality of the RL Model in diverse parking scenarios.
2. Ensuring the stability and robustness of the application across various Windows environments.
3. Verifying that user interactions with the UI are intuitive and error-free.

Testing Phases

1. **Unit Testing:**
    - **Objective:** Validate the functionality of individual components, ensuring they perform as expected.
    - **Focus Areas:** Methods, functions, and modules within the Car Controller, RL Model, and UI components.
    - **Tools:** Unity Test Framework for Unity-specific components, standard testing frameworks for other modules.
2. **Integration Testing:**
    - **Objective:** Verify the collaboration and interaction between different modules within the iPark system.
    - **Focus Areas:** Ensure seamless communication between the Car Controller, RL Model, and UI components.
    - **Tools:** Unity Test Framework, standard integration testing tools.
3. **System Testing:**

- **Objective:** Assess the overall system behaviour by testing the integrated components as a complete unit.
- **Focus Areas:** Verify end-to-end functionality, including user interactions, RL model integration, and Unity physics engine interactions.
- **Tools:** Unity Play Mode Tests, external testing tools for non-Unity components.

4. **User Acceptance Testing (UAT):**
    - **Objective:** Evaluate the iPark system's usability and effectiveness in meeting user requirements.
    - **Focus Areas:** Gather feedback on the user interface, car behavior, and overall user experience.
    - **Tools:** User feedback sessions, surveys, and usability testing tools.

Test Types

1. **Functional Testing:**
    - **Objective:** Confirm that each functional requirement specified in the iPark system is met.
    - **Activities:** Car movement validation, RL agent decision-making tests, UI button functionality tests.
    - **Tools:** Unity Play Mode Tests, external tools for non-Unity components.
2. **Performance Testing:**
    - **Objective:** Assess the system's responsiveness, stability, and scalability under varying conditions.
    - **Activities:** Evaluate RL model performance, measure UI responsiveness, and assess system behaviour under simulated load conditions.
    - **Tools:** Profiling tools for Unity, external performance testing tools.
3. **Security Testing:**
    - **Objective:** Identify and address potential security vulnerabilities within the iPark system.
    - **Activities:** Review code for security best practices, test for common security threats, and ensure secure communication between components.

- **Tools:** Code review tools, security testing tools, penetration testing tools.

4. **Usability Testing:**

   - **Objective:** Evaluate the iPark system's user interface for intuitiveness and user-friendliness.
   - **Activities:** Conduct user interaction tests, gather user feedback on UI design, and assess the ease of use for different features.
   - **Tools:** Usability testing tools, user feedback sessions.

5. **Regression Testing:**

   - **Objective:** Ensure that new changes or additions to the iPark system do not negatively impact existing functionalities.
   - **Activities:** Re-run previously validated test cases after each system update or modification.
   - **Tools:** Automated testing tools, version control systems.

4.10 Project Deployment

The deployed application setup can be found here: `iParkSetup.exe`

4.10.1 Deployment Strategy

The deployment of iPark: Intelligent Parking on a Windows system will be executed with careful consideration to ensure a seamless and efficient installation process for end-users.

1. **User Documentation:** Developing comprehensive user documentation will be a key aspect of the deployment strategy. This documentation will accompany the setup file, providing users with detailed instructions, system requirements, and insights into the application's functionalities.

2. **Inno Setup Compiler:** The deployment will leverage the Inno Setup Compiler to create a setup executable file (.exe) for iPark. This powerful tool offers a user-friendly interface for configuring the installation process, facilitating the generation of an installer that streamlines deployment.

3. **Offline Installation:** To cater to diverse user scenarios, the deployment plan includes provisions for offline installations. All necessary dependencies and resources will be encapsulated within the setup file, allowing users to install iPark without requiring an active internet connection.

4. **Desktop Shortcut and Start Menu Integration:** To enhance user convenience, the setup will be configured to create desktop shortcuts and integrate iPark into the Start Menu. This facilitates easy access to the application post-installation.

5. **Uninstalling Support:** The deployment plan emphasizes user-friendly uninstalling. iPark will provide an option for users to uninstall the application effortlessly, with the uninstaller ensuring the clean removal of all installed components and dependencies.

4.10.2 Installation Instructions

To install iPark on your Windows system, follow these steps:

1. **Download Setup File:** Download the setup file (e.g., iParkSetup.exe) from the designated distribution platform or the GitHub Page.

2. **Run Setup:** Double-click on the setup file to initiate the installation process.

3. **System Compatibility Check:** The installer will perform a system compatibility check. Ensure your system meets the minimum requirements.

4. **Choose Installation Location:** Specify the directory where iPark will be installed. The default directory is recommended for most users.

5. **Start Installation:** Click "Install" to begin the installation process. The installer will copy files and set up the necessary components.

6. **Desktop Shortcut and Start Menu Integration:** Choose whether to create a desktop shortcut and integrate iPark into the Start Menu.

7. **Complete Installation:** Once the installation is complete, click "Finish" to exit the installer.

8. **Launch iPark:** Double-click on the iPark desktop shortcut or locate it in the Start Menu to launch the application.

9. **Uninstallation (Optional):** If needed, use the provided uninstaller in the Control Panel to remove iPark from your system.

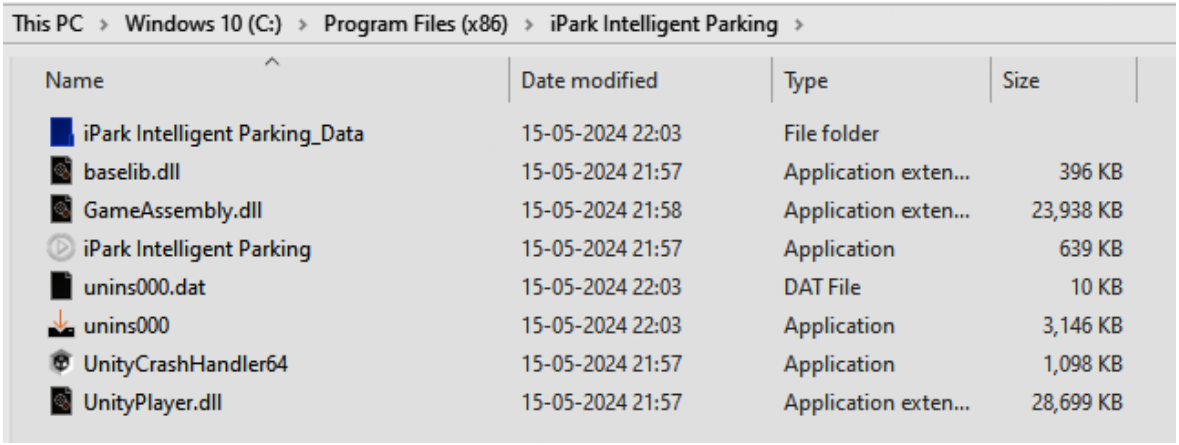

Figure 4.16: iPark installed in the default directory (C:\Program Files (x86)\)

Chapter 5

Results

5.1 Terminologies Used

1. **Cumulative Reward:** Cumulative Reward in TensorBoard graphs tracks the total reward obtained by the agent during training or evaluation. It offers a quick overview of the agent's overall performance and its ability to achieve goals within the environment.

2. **Episode Length:** Episode Length in TensorBoard graphs represents the duration of each episode during training. It indicates how long the agent interacts with the environment before reaching a terminal state or completing a task. Tracking episode length helps monitor the efficiency and effectiveness of the agent's decision-making process over time.

3. **Policy Loss:** Policy Loss in TensorBoard graphs reflects the discrepancy between the predicted actions of the agent and the optimal actions determined by the policy during training. It measures how well the agent's policy approximates the desired behaviour and provides insights into the training progress and stability of the reinforcement learning algorithm.

4. **Value Loss:** In TensorBoard, the Value Loss metric tracks the error between predicted and observed returns during training. A good performance shows a decreasing trend over time, indicating improved accuracy in predicting future rewards. Fluctuations may occur, but overall, the curve should converge to a low and stable level, signalling successful learning by the agent.

5. **Policy Entropy:** Policy Entropy in TensorBoard measures the uncertainty or randomness of the agent's action selection. A good agent should maintain a moderate level of entropy to encourage exploration and prevent premature convergence to suboptimal policies. An ideal scenario shows a decreasing trend in

entropy as the agent learns to make more confident and informed decisions over time, but without diminishing too quickly, ensuring a balance between exploration and exploitation.

5.2 Model-wise Training Results

We've meticulously documented the training outcomes and model parameters on a per-model basis. This section will thoroughly explore these findings, providing insights into the model parameters and parking efficiency attained during training.

5.2.1 iPark [01] 21-11-2023

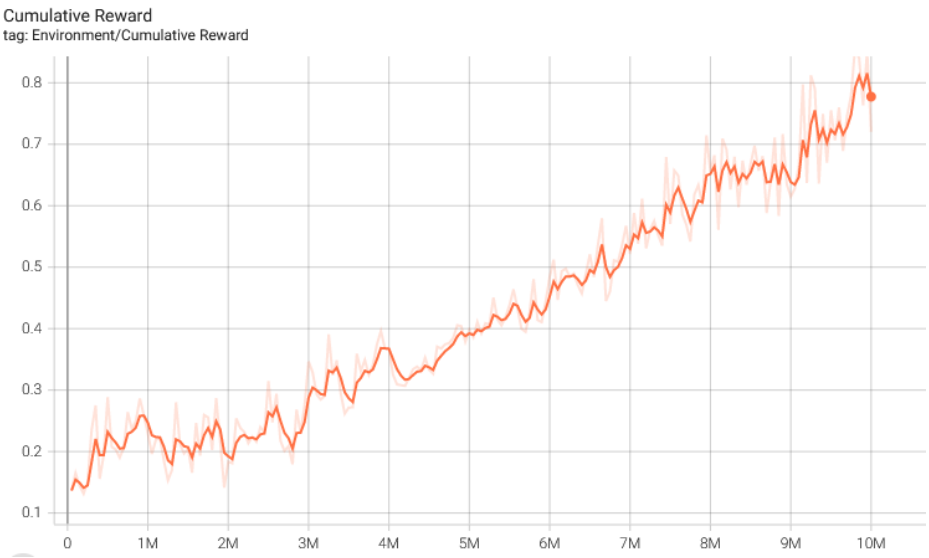

Figure 5.1: Cumulative Reward for iPark [01] 21-11-2023

The cumulative reward graph starts at 50k steps with a reward value of 0.1365, and ends at 10M steps with a value of 0.7201 in 4 hours. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period, which shows that the model was learning new behaviours and did not mature early on sub-optimal rules.

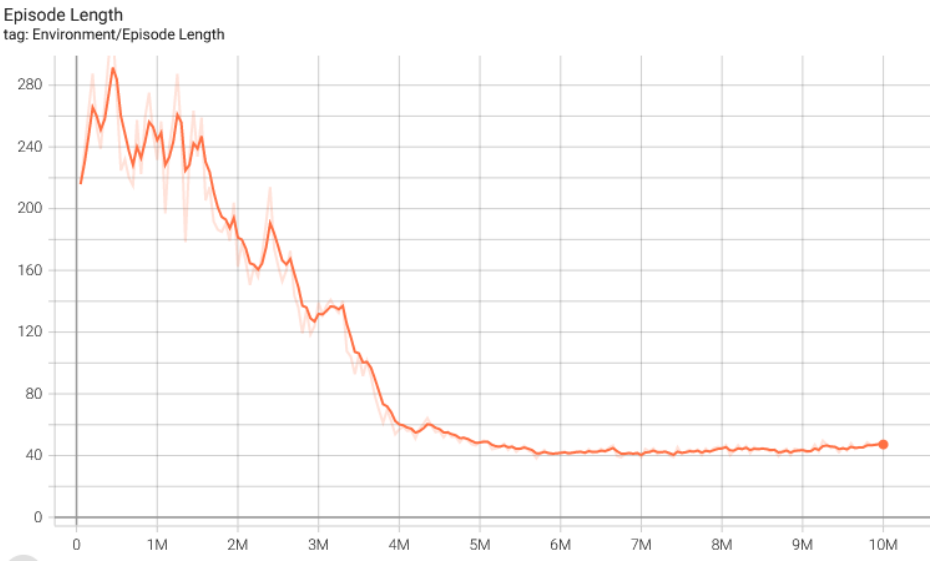

Figure 5.2: Episode Length for iPark [01] 21-11-2023

The episode length graph starts at 50k steps with an episode length of almost 216 (215.8) and ends at 10M steps with a length value of 47. This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each episode. Notably, the graph initially increased from 215.8 (50k steps) to 316.6 (450k steps) and then continuously declined. It remained almost flat from 6M steps (42.07) to 10M steps(47).

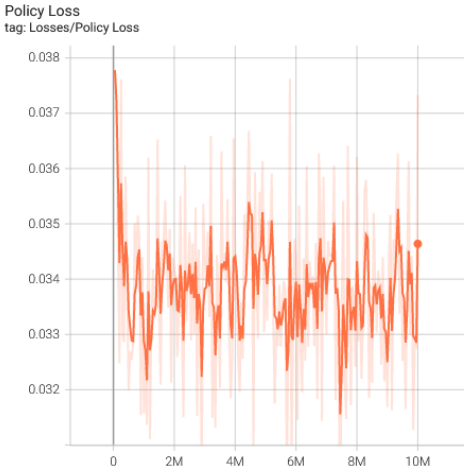

Figure 5.3: Policy Loss for iPark [01] 21-11-2023

The policy loss graph starts at 50k steps with a value of 0.03778 and ends at 10M steps with a value of 0.03732. This decline in value shows that the agent's policy is converging towards an optimal solution. The fluctuations show that the agent is learning from the environment.

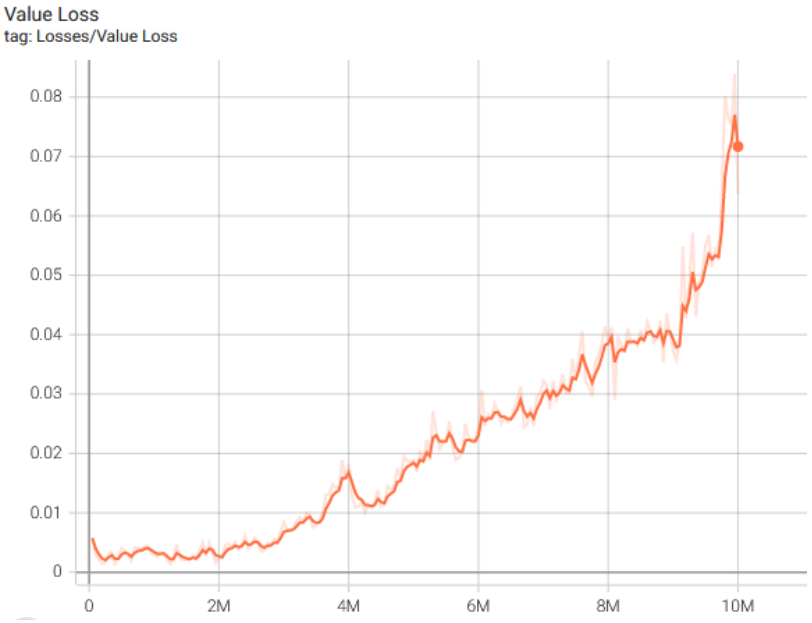

Figure 5.4: Value Loss for iPark [01] 21-11-2023

The value loss graph starts at 50k steps with a value of 5.752e-3 (0.005752) and ends at 10M steps with a value of 0.06366. Value loss shows the difference between the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph but this increasing behaviour shows that the model is not being able to predict the returns of its actions. The model is not at optimal behaviours.

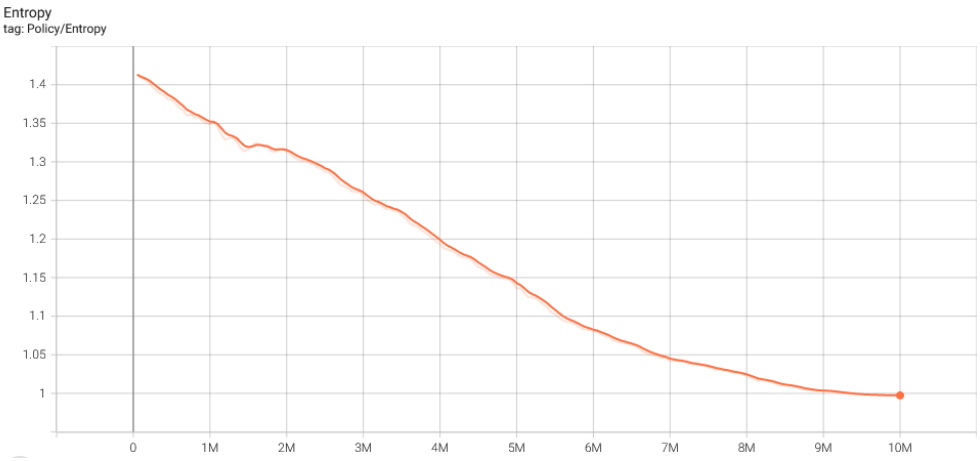

Figure 5.5: Policy Entropy for iPark [01] 21-11-2023

The policy entropy graph starts at 50k steps with a value of 1.413 and ends at 10M steps with a value of 0.9973. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

5.2.2 iPark [02] 28-03-2024

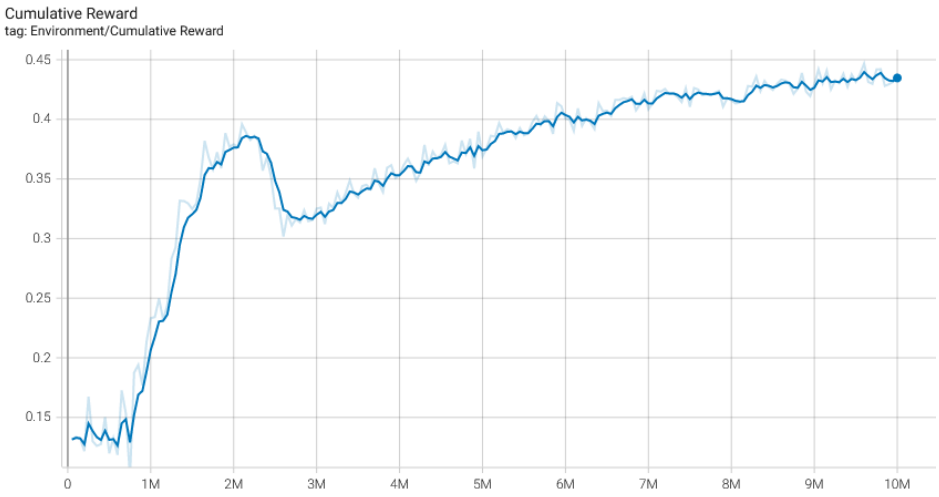

Figure 5.6: Cumulative Reward for iPark [02] 28-03-2024

The cumulative reward graph starts at 50k steps with a reward value of 0.1315, and ends at 10M steps with a value of 0.4385 in 3 hours 46 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period, except a dip at around 2.2M steps. This shows that the model was learning new behaviours and did not mature early on sub-optimal rules. The almost flat graph at the end shows that the model was able to find the optimal settings and the training period was sufficient.

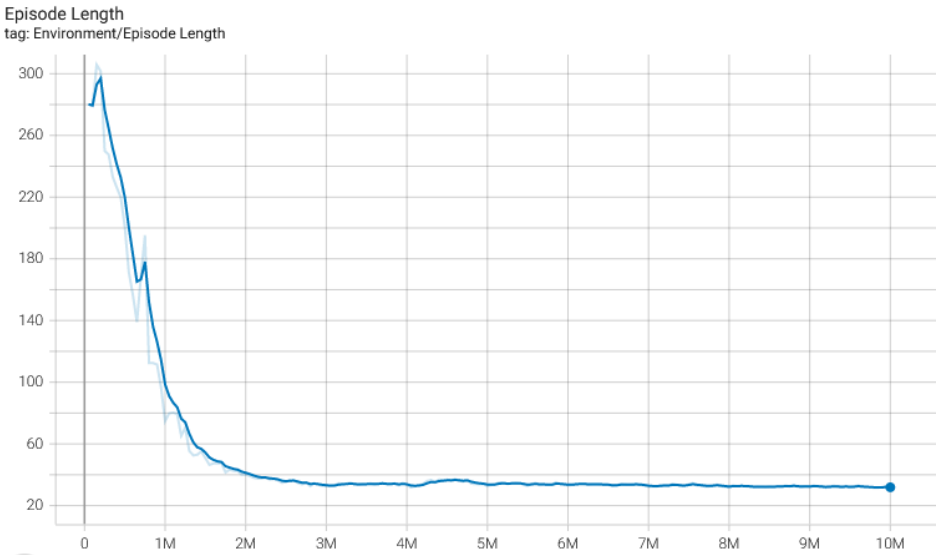

Figure 5.7: Episode Length for iPark [02] 28-03-2024

The episode length graph starts at 50k steps with an episode length of almost 280

(280.2) and ends at 10M steps with a length value of 32 (31.68). This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each episode. It remained almost flat from 3M steps (32.84) to 10M steps (31.68) which shows that the model was able to find the optimal action set very early.

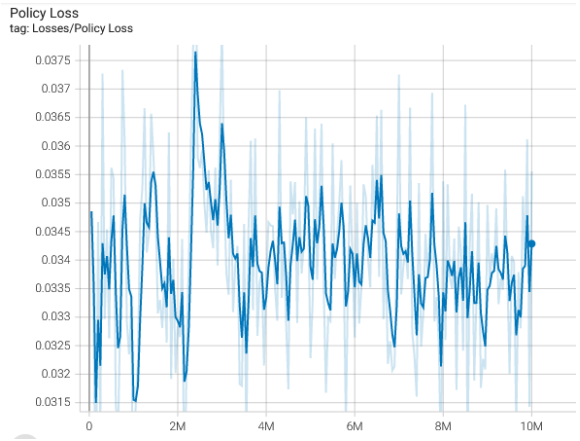

Figure 5.8: Policy Loss for iPark [02] 28-03-2024

The policy loss graph starts at 50k steps with a value of 0.03486 and ends at 10M steps with a value of 0.03556. This increase in value might show that the agent's policy is converging away from an optimal solution. However, the graph is constantly declining from 2.4M steps (0.04704), this shows that the model was moving towards an optimal solution but the training period expired. The fluctuations show that the agent is learning from the environment.

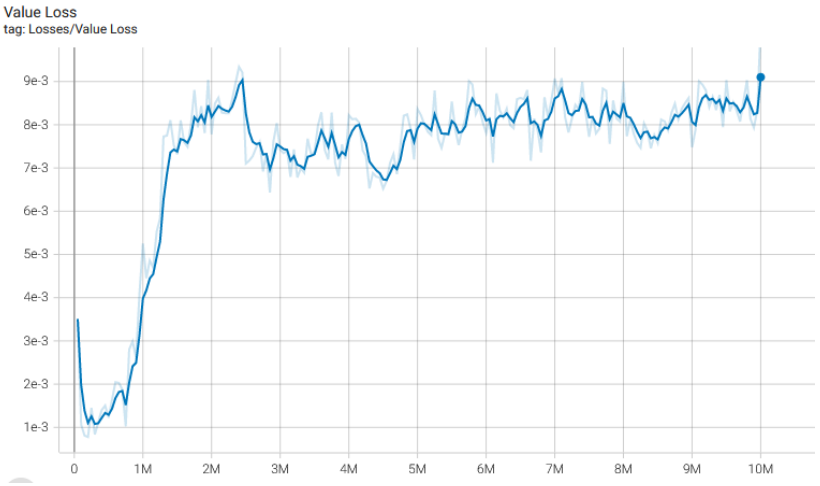

Figure 5.9: Value Loss for iPark [02] 28-03-2024

The value loss graph starts at 50k steps with a value of 3.5087e-3 (0.0035087) and ends at 10M steps with a value of 0.01034. Value loss shows the difference between

the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph. The graph initially increased steeply but is almost constant with fluctuations from 5M steps which shows that the model is converging to a optimal solution.

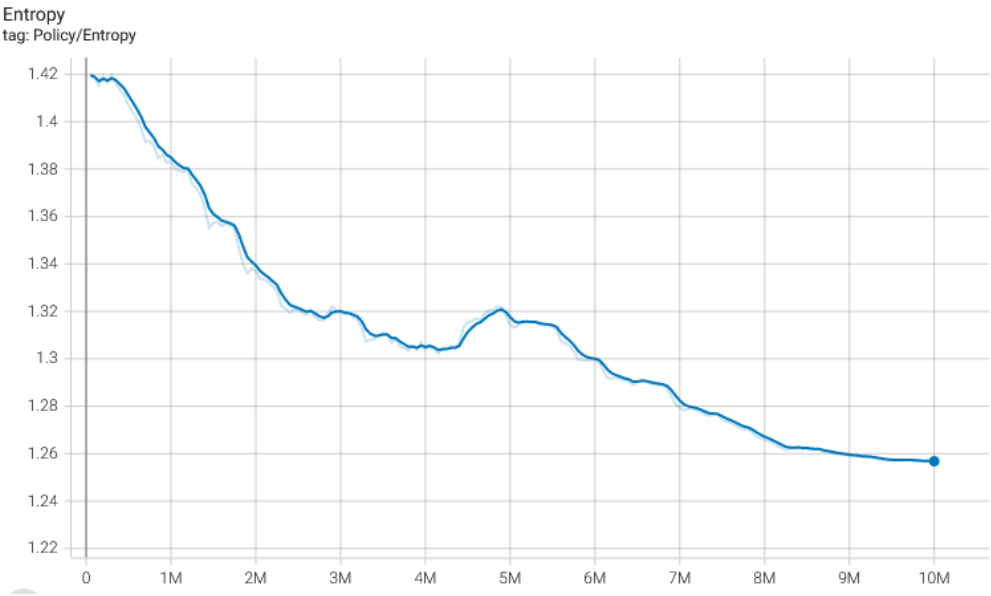

Figure 5.10: Policy Entropy for iPark [02] 28-03-2024

The policy entropy graph starts at 50k steps with a value of 1.42 and ends at 10M steps with a value of 1.257. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution. Also the graph was almost constant from 8.5M steps, this shows that model was able to come to an optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    2024-03-29 02:47:52 (Training Model 28/03)
    Efficiency 74.95232%
```

The `EfficiencyCal` and `EfficiencyCombined` script reported this efficiency of the model in the created text file. This is for whole of the training period and will differ from real testing.

5.2.3 iPark [03] 29-03-2024

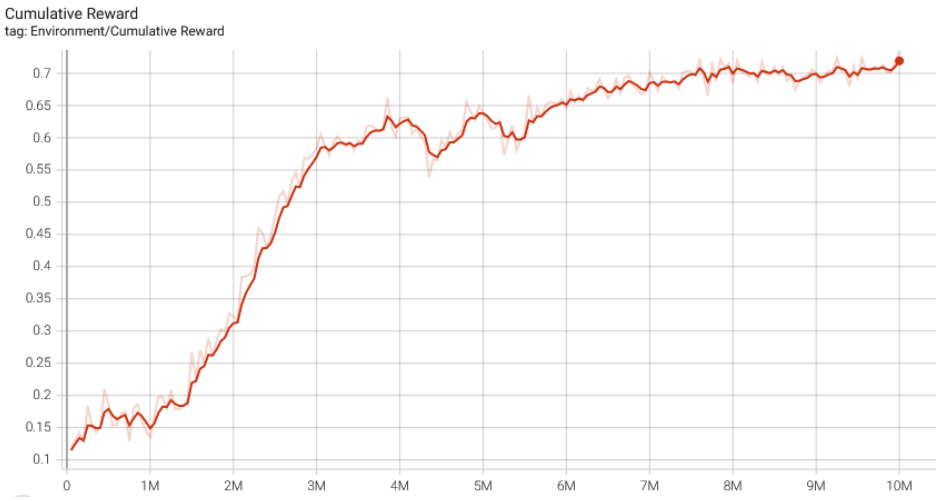

Figure 5.11: Cumulative Reward for iPark [03] 29-03-2024

The cumulative reward graph starts at 50k steps with a reward value of 0.1149, and ends at 10M steps with a value of 0.7329 in 4 hours 33 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period, except a slight dip at around 4M steps. This shows that the model was learning new behaviours and did not mature early. The almost flat graph at the end (from 8M steps) shows that the model was able to find the optimal settings and the training period was sufficient.

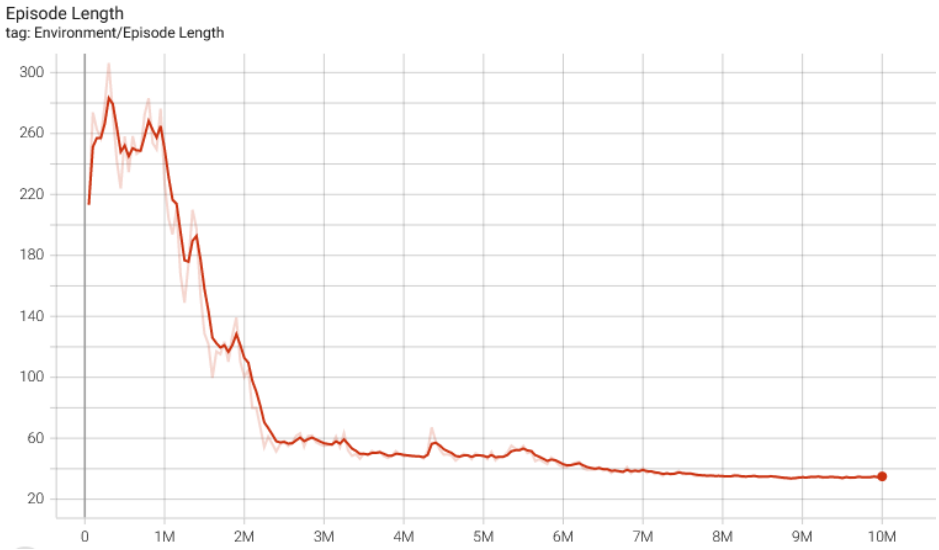

Figure 5.12: Episode Length for iPark [03] 29-03-2024

The episode length graph starts at 50k steps with an episode length of almost 213

(213.2) and ends at 10M steps with a length value of 36 (35.64). This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each episode. It remained almost flat from 8M steps (34.86) to 10M steps (35.64) which shows that the model was able to find the optimal action set.

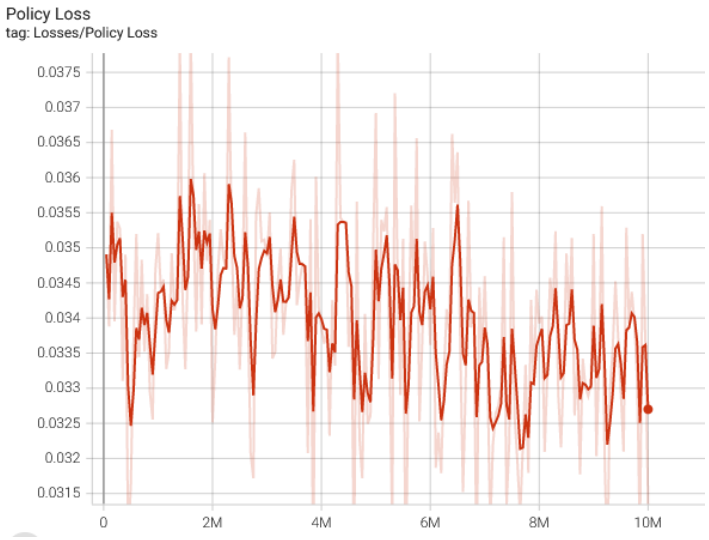

Figure 5.13: Policy Loss for iPark [03] 29-03-2024

The policy loss graph starts at 50k steps with a value of 0.03491 and ends at 10M steps with a value of 0.0327. This decrease in value show that the model was able to find a optimal policy function. Moreover, the graph is declining continuously, meaning that the model was improving throughout the training period. The fluctuations show that the agent is learning from the environment.

Figure 5.14: Value Loss for iPark [03] 29-03-2024

The value loss graph starts at 50k steps with a value of 9.7516e-3 (0.0097516) and ends at 10M steps with a value of 0.0135. Value loss shows the difference between the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph. The graph increased steeply from 1M steps but is almost constant with fluctuations from 5M steps which shows that the model is converging to a optimal solution.

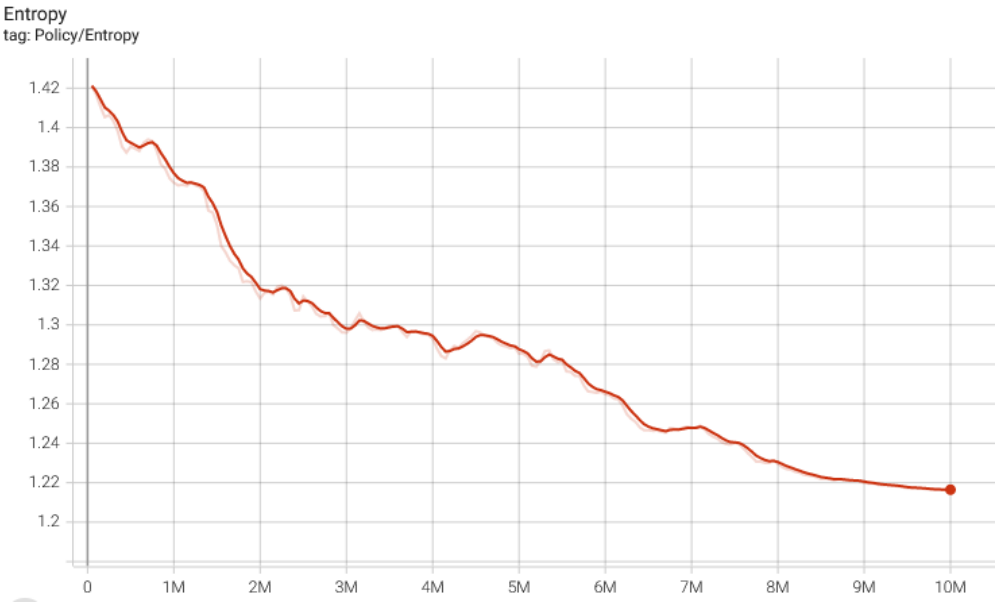

Figure 5.15: Policy Entropy for iPark [03] 29-03-2024

The policy entropy graph starts at 50k steps with a value of 1.421 and ends at 10M steps with a value of 1.216. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    30-03-2024 13:48:07 (Training Model 29-03)
    Efficiency 84.70142%
```

The `EfficiencyCal` and `EfficiencyCombined` script reported this efficiency of the model in the created text file. This is for whole of the training period and will differ from real testing.

5.2.4 iPark [04] 30-03-2024

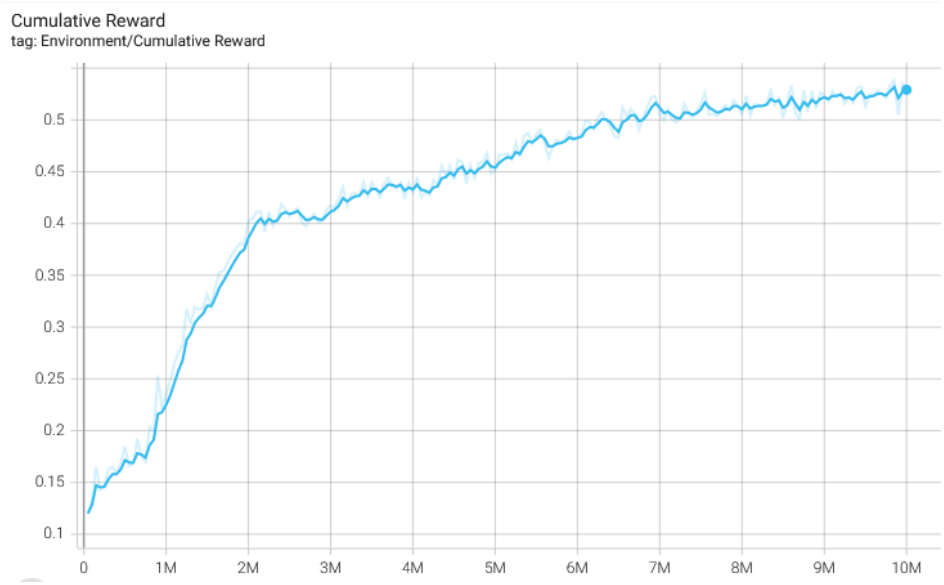

Figure 5.16: Cumulative Reward for iPark [04] 30-03-2024

The cumulative reward graph starts at 50k steps with a reward value of 0.1199, and ends at 10M steps with a value of 0.5327 in 3 hours 56 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period. This shows that the model was learning new behaviours and did not mature early. The almost flat graph at the end (from 8M steps) shows that the model was able to find the optimal settings and the training period was sufficient.

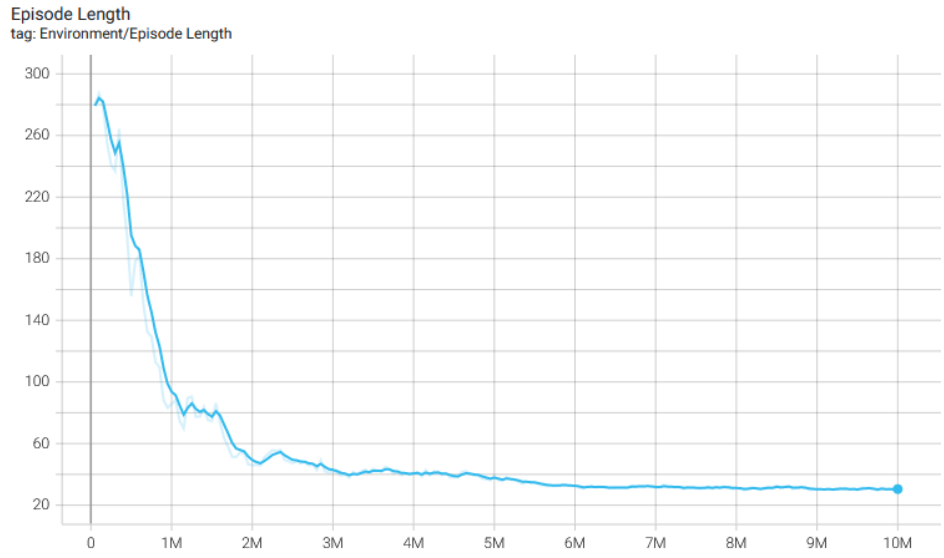

Figure 5.17: Episode Length for iPark [04] 30-03-2024

The episode length graph starts at 50k steps with an episode length of almost 279 (279.3) and ends at 10M steps with a length value of 31 (30.7). This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each episode. It remained almost flat from 6M steps (32.46) to 10M steps (30.7) which shows that the model was able to find the optimal action set fairly early.

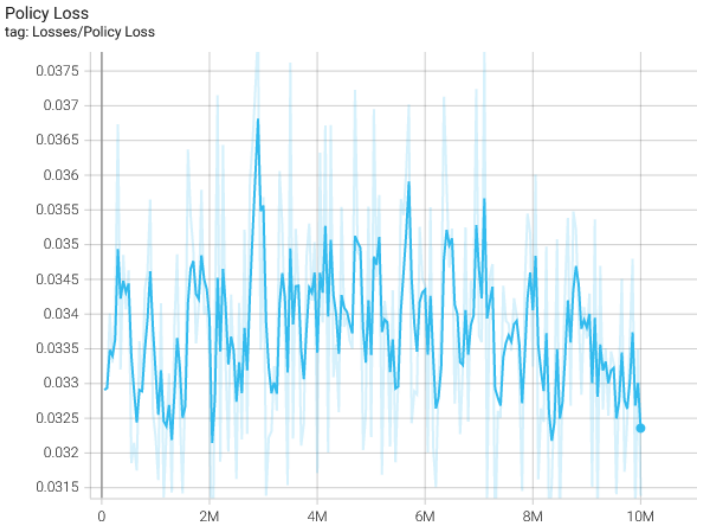

Figure 5.18: Policy Loss for iPark [04] 30-03-2024

The policy loss graph starts at 50k steps with a value of 0.03291 and ends at 10M steps with a value of 0.03138. This decrease in value show that the model was able to find a optimal policy function. Moreover, the graph is declining from 4M, meaning that the model was improving throughout the majority of the training period. The fluctuations show that the agent is learning from the environment.

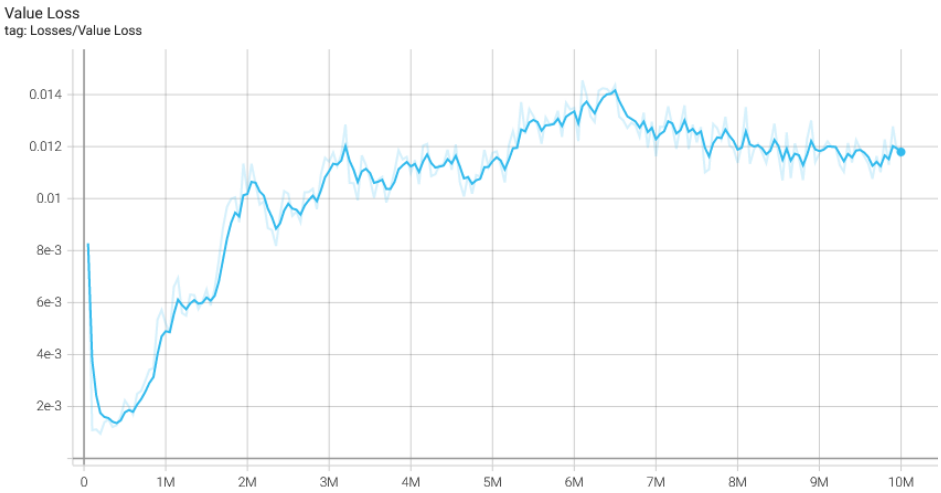

Figure 5.19: Value Loss for iPark [04] 30-03-2024

The value loss graph starts at 50k steps with a value of 8.2821e-3 (0.0082821) and ends at 10M steps with a value of 0.01159. Value loss shows the difference between the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph. The graph increased steeply from 400k steps but is constantly declining with fluctuations from 6.5M steps which shows that the model is converging to a optimal solution.

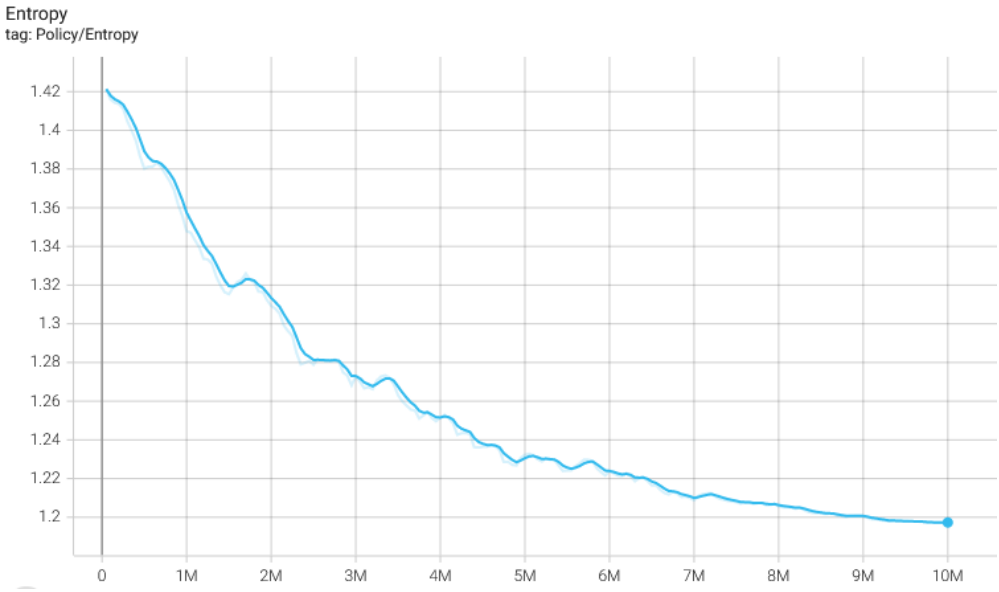

Figure 5.20: Policy Entropy for iPark [04] 30-03-2024

The policy entropy graph starts at 50k steps with a value of 1.421 and ends at 10M steps with a value of 1.197. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    30-03-2024 23:32:12 (Training Model 30-03)
    Efficiency 79.63393%
```

The `EfficiencyCal` and `EfficiencyCombined` script reported this efficiency of the model in the created text file. This is for whole of the training period and will differ from real testing.

5.2.5 iPark Export [01] 04-05-2024

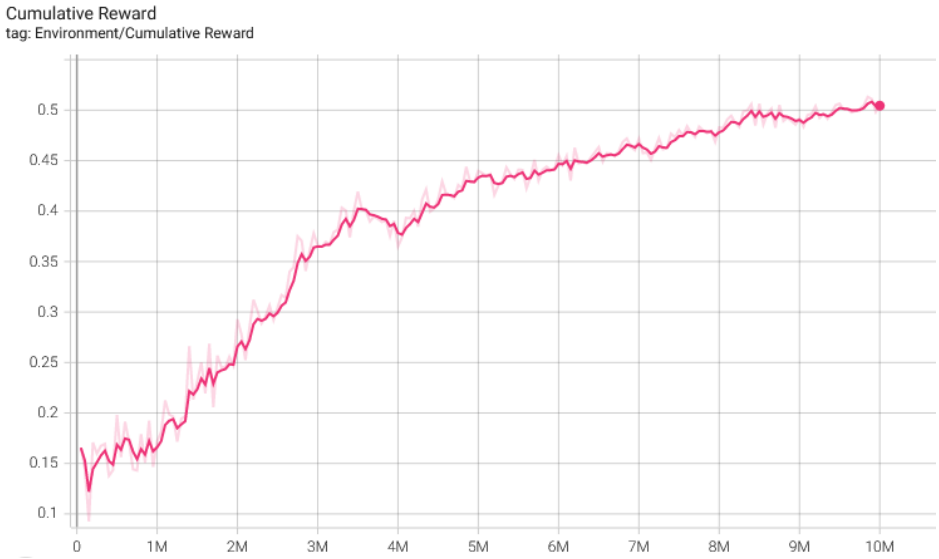

Figure 5.21: Cumulative Reward for iPark Export [01] 04-05-2024

The cumulative reward graph starts at 50k steps with a reward value of 0.1656, and ends at 10M steps with a value of 0.5048 in 3 hours 48 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period. This shows that the model was learning new behaviours and did not mature early.

Figure 5.22: Episode Length for iPark Export [01] 04-05-2024

The episode length graph starts at 50k steps with an episode length of almost 242 (241.5) and ends at 10M steps with a length value of 32 (32.22). This shows that the

model was learning new optimal behaviours and was able to park with fewer steps in each episode.

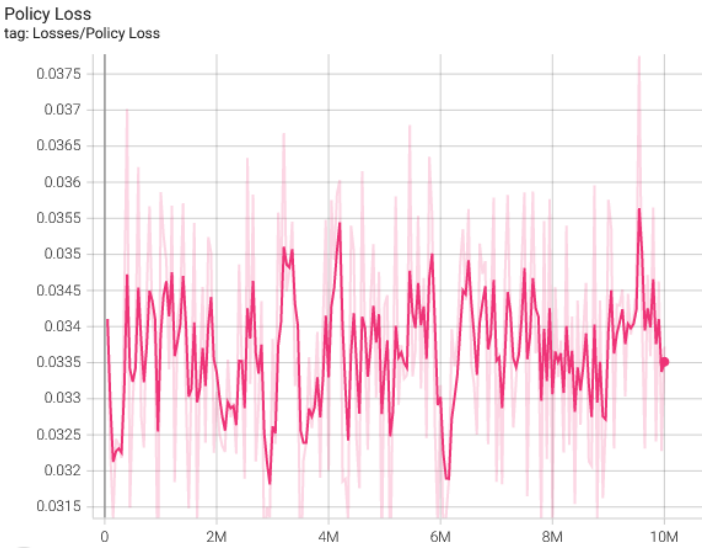

Figure 5.23: Policy Loss for iPark Export [01] 04-05-2024

The policy loss graph starts at 50k steps with a value of 0.03411 and ends at 10M steps with a value of 0.03372. This decrease in value show that the model was able to find a optimal policy function. Moreover, the graph is declining from 9.55M, meaning that the model was improving at the end of the training period. The fluctuations show that the agent is learning from the environment.

Figure 5.24: Value Loss for iPark Export [01] 04-05-2024

The value loss graph starts at 50k steps with a value of 5.965e-3 (0.005965) and ends at 10M steps with a value of 0.01262. Value loss shows the difference between

the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph. The graph initially increased steeply till 5.5M steps, but then the rate of increase was very low which shows that the model was able to find good rules after 5.5M steps.

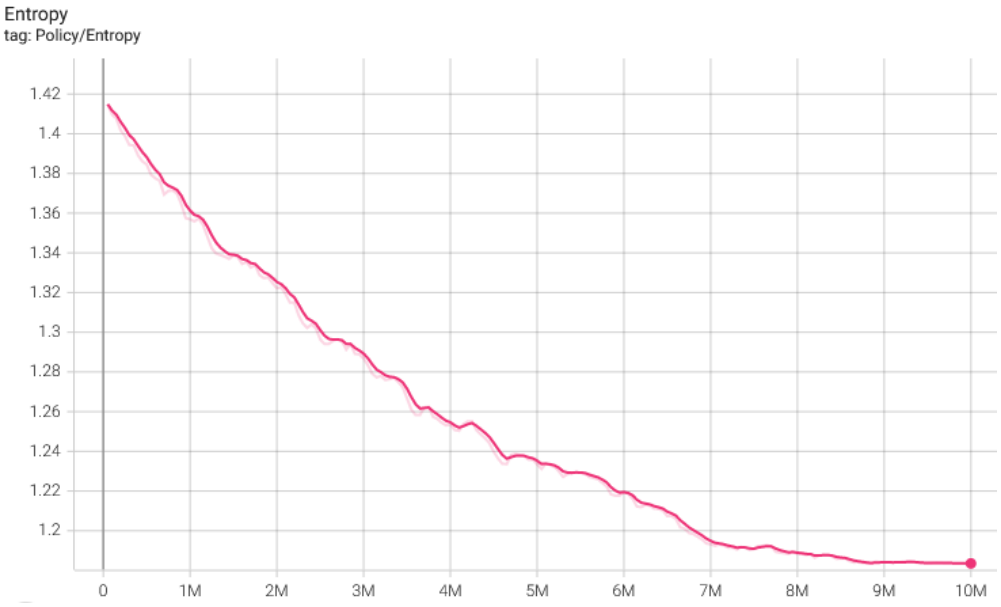

Figure 5.25: Policy Entropy for iPark Export [01] 04-05-2024

The policy entropy graph starts at 50k steps with a value of 1.415 and ends at 10M steps with a value of 1.183. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component
    04-05-2024 21:41:02 (Training Model)
    Efficiency 75.3954%
    Total Park 143200
    Total Collision 46732
    Total Cases 189932
```

The `EfficiencyCal` and `EfficiencyCombined` script reported this efficiency of the model in the created text file. This is for whole of the training period and will differ from real testing.

5.2.6 iPark Export [02] 06-05-2024

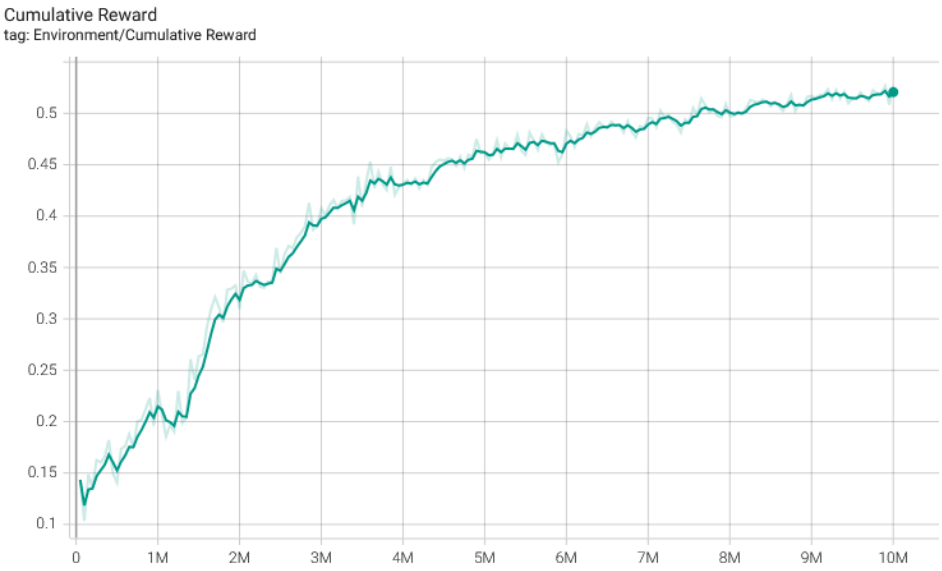

Figure 5.26: Cumulative Reward for iPark Export [02] 06-05-2024

The cumulative reward graph starts at 50k steps with a reward value of 0.1434, and ends at 10M steps with a value of 0.5269 in 3 hours 28 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period. This shows that the model was learning new behaviours and did not mature early.

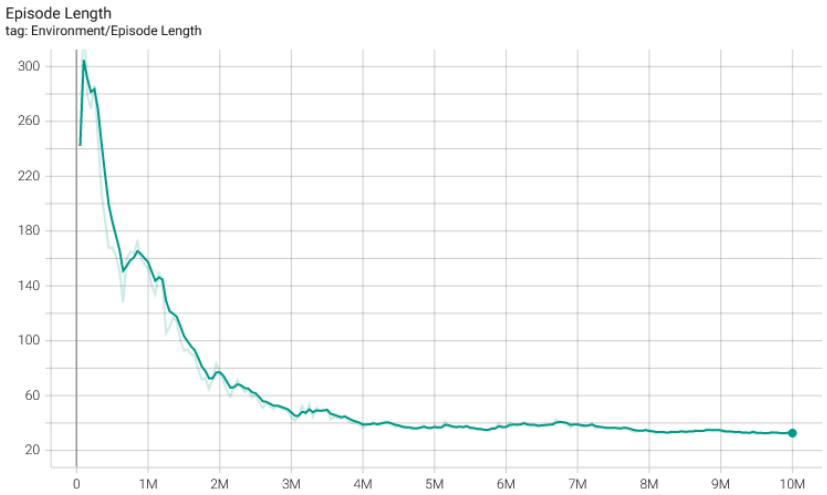

Figure 5.27: Episode Length for iPark Export [02] 06-05-2024

The episode length graph starts at 50k steps with an episode length of 242 and ends at 10M steps with a length value of 32 (32.37). This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each episode. Moreover, the graph was almost flat from 4M steps (36.35) which shows that the model

was able to find optimal settings very early.

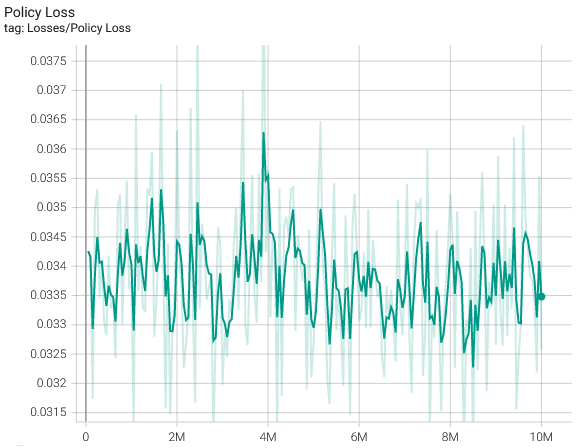

Figure 5.28: Policy Loss for iPark Export [02] 06-05-2024

The policy loss graph starts at 50k steps with a value of 0.03426 and ends at 10M steps with a value of 0.03257. This decrease in value show that the model was able to find a optimal policy function. Moreover, the graph is constantly declining, meaning that the model was improving throughout the training period. The fluctuations show that the agent is learning from the environment.

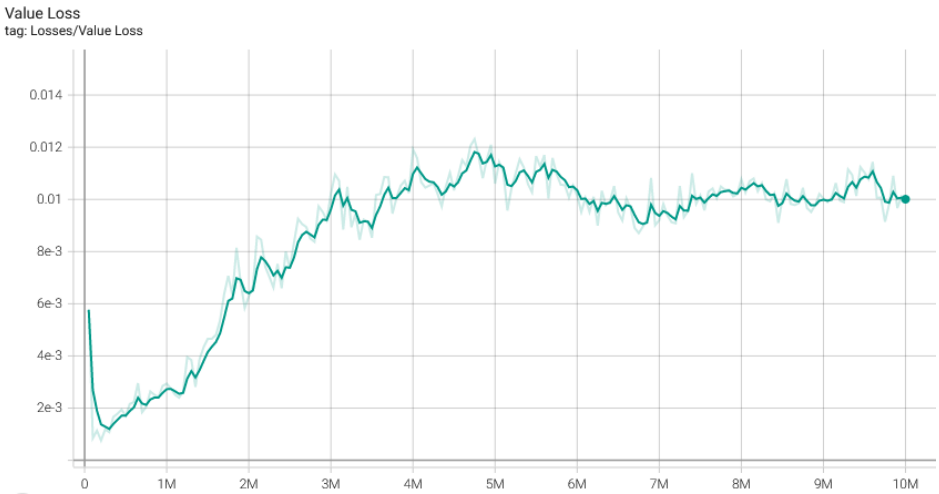

Figure 5.29: Value Loss for iPark Export [02] 06-05-2024

The value loss graph starts at 50k steps with a value of 5.768e-3 (0.005768) and ends at 10M steps with a value of 9.9214e-3 (0.0099214). Value loss shows the difference between the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph. The graph initially increased till 4.75M steps, but then

it declined and stayed continuous from 7M steps. Morover, the overall increase was very low (0.0041534) which shows the model really performed well in value loss and find an optimal solution.

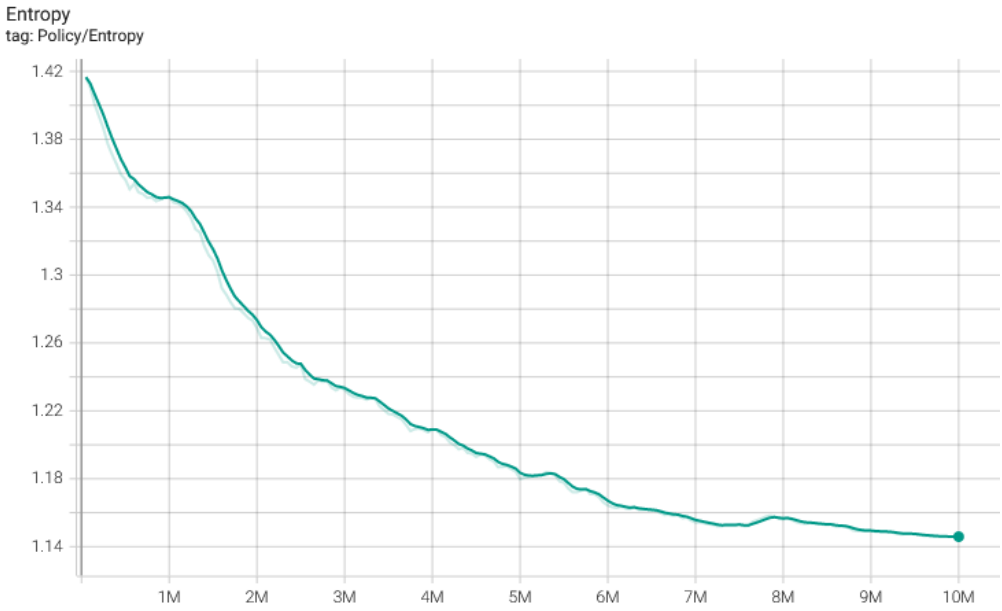

Figure 5.30: Policy Entropy for iPark Export [02] 06-05-2024

The policy entropy graph starts at 50k steps with a value of 1.417 and ends at 10M steps with a value of 1.146. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    08-05-2024 15:30:55 (Training Model)
    Efficiency 79.27053%
    Total Park 170392
    Total Collision 44558
    Total Cases 214950
```

The `EfficiencyCal` and `EfficiencyCombined` script reported this efficiency of the model in the created text file. This is for whole of the training period and will differ from real testing.

5.2.7 iPark Export [03] 08-05-2024

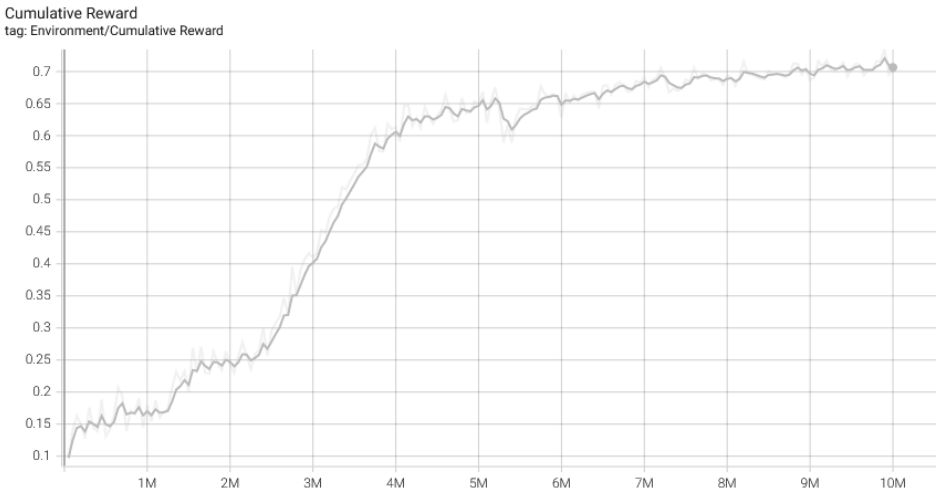

Figure 5.31: Cumulative Reward for iPark Export [03] 08-05-2024

The cumulative reward graph starts at 50k steps with a reward value of 0.09712, and ends at 10M steps with a value of 0.7008 in 3 hours 36 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period. showing that the model was learning new behaviours and did not mature early. The graph stayed almost constant from 9M steps showing that the model was near to optimal settings.

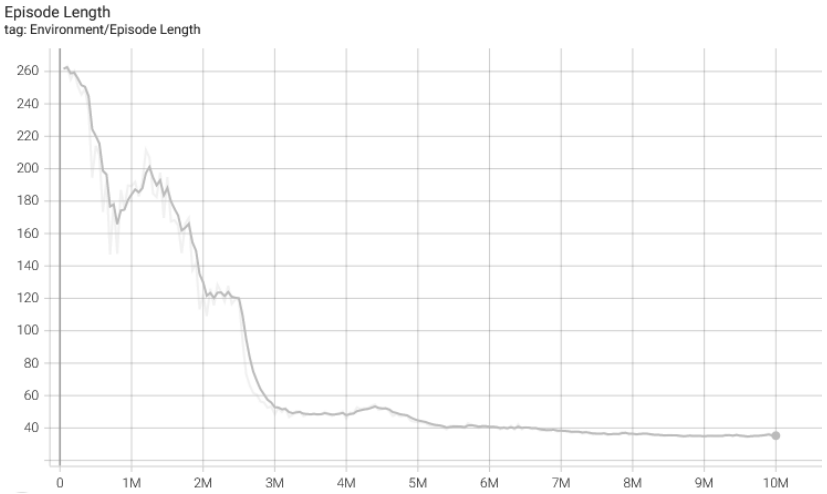

Figure 5.32: Episode Length for iPark Export [03] 08-05-2024

The episode length graph starts at 50k steps with an episode length of 262 (261.6) and ends at 10M steps with a length value of 35 (34.97). This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each

episode. Moreover, the graph was almost flat from 7.5M steps (36.51) which shows that the model was able to find optimal settings.

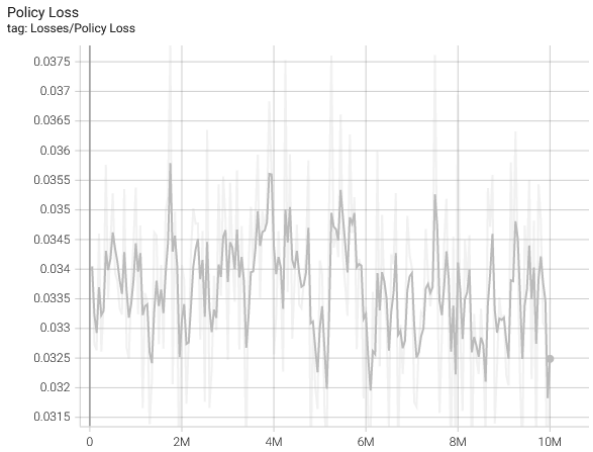

Figure 5.33: Policy Loss for iPark Export [03] 08-05-2024

The policy loss graph starts at 50k steps with a value of 0.03405 and ends at 10M steps with a value of 0.03249. This decrease in value show that the model was able to find a optimal policy function. Moreover, the graph is constantly declining, meaning that the model was improving throughout the training period. The fluctuations show that the agent is learning from the environment.

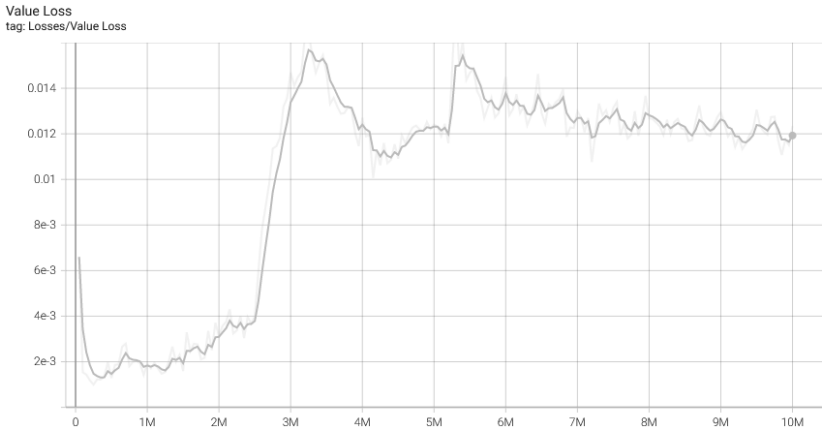

Figure 5.34: Value Loss for iPark Export [03] 08-05-2024

The value loss graph starts at 50k steps with a value of 6.5978e-3 (0.0065978) and ends at 10M steps with a value of 0.01234. Value loss shows the difference between the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a decreasing or constant graph. The graph initially increased till 5.4M steps, but then it declined

continuously, which shows the model was able to find an optimal solution.

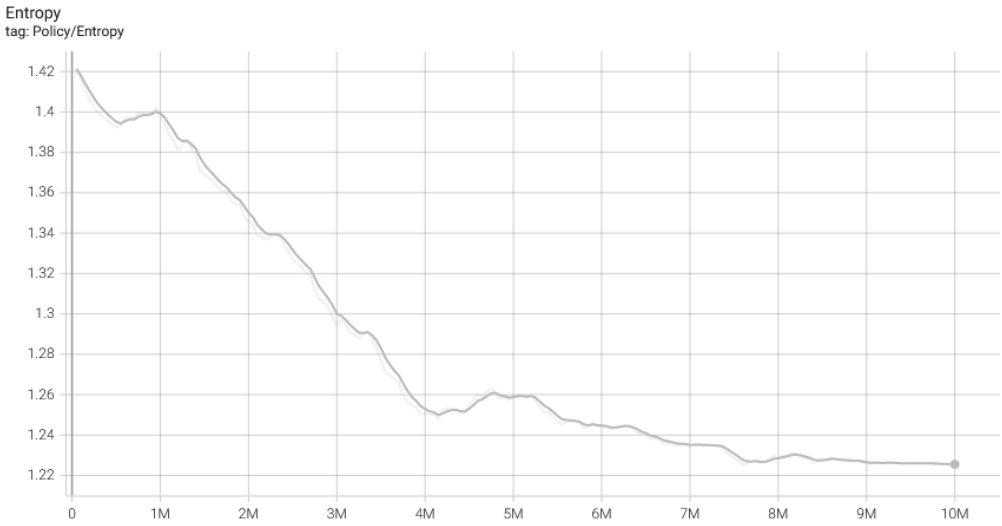

Figure 5.35: Policy Entropy for iPark Export [03] 08-05-2024

The policy entropy graph starts at 50k steps with a value of 1.421 and ends at 10M steps with a value of 1.225. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    08-05-2024 22:40:04 (Training Model)
    Efficiency 79.25703%
    Total Park 150414
    Total Collision 39366
    Total Cases 189780
```

The `EfficiencyCal` and `EfficiencyCombined` script reported this efficiency of the model in the created text file. This is for whole of the training period and will differ from real testing.

5.3 Final Training Results

5.3.1 Graph Analysis

iPark [01] 21-11-2023

The cumulative graph demonstrates significant progress, with the final value ranking among the highest (0.7201). However, this alone cannot determine the model's performance in test scenarios. While the episode length graph appears somewhat ideal, the final value is relatively high, indicating that the model struggled to optimize episode length effectively. Additionally, although the policy loss graph exhibits a decline, the steeply rising value loss is concerning and suggests suboptimal policy performance. On a positive note, the policy entropy remains ideal, indicating that the Proximal Policy Optimization (PPO) algorithm is functioning as expected. Further insights into the model's performance will be revealed during the testing phase.

iPark [02] 28-03-2024

The cumulative graph demonstrates notable progress, with the final value deemed acceptable (0.4385). However, the dip around 2.2M is concerning and warrants attention. On a positive note, the episode length graph appears ideal, with the final value ranking among the lowest, indicating successful optimization of episode length by the model. Conversely, the policy loss graph exhibits an increasing trend despite the decrease in value, which is not optimal. However, the value loss remains acceptable, as it did not continue to increase and instead remained relatively constant. The policy entropy is close to ideal, indicating proper functionality of the Proximal Policy Optimization (PPO) algorithm. It's important to note that efficiency data recorded throughout the training period (74.95232%) may not fully represent the model's performance, and more insights will be gleaned during the testing phase.

iPark [03] 29-03-2024

The cumulative graph displays remarkable progress, with the final value reaching the highest point (0.7329). However, relying solely on this metric may not accurately predict the model's performance in test scenarios. The episode length graph also appears ideal, with the final value deemed acceptable, indicating successful optimization of episode length by the model. Conversely, while the policy loss graph shows a contin-

uous decline, the value loss presents concerns. Although the value loss initially rose before stabilizing, the final value remained quite high. Despite this, the policy entropy remains somewhat ideal, indicating proper functionality of the Proximal Policy Optimization (PPO) algorithm. The efficiency value recorded during the training phase was decent (84.70142%), but its correlation with testing performance remains uncertain. Further insights into the model's performance will be gained during the testing phase.

iPark [04] 30-03-2024

The cumulative graph illustrates significant progress, with the final value reaching a decent level (0.5327). However, relying solely on this metric may not accurately predict the model's performance in test scenarios. Similarly, the episode length graph portrays an ideal trend, with the final value ranking among the lowest (31), indicating successful optimization of episode length by the model. While the policy loss graph demonstrates a decline for most of the training period, the steep rise in value loss is concerning. Nonetheless, the value loss stabilized later on, suggesting that the model learned optimal settings over time. The policy entropy remains ideal, indicating proper functionality of the Proximal Policy Optimization (PPO) algorithm. Additionally, the efficiency value recorded during the training phase is respectable (79.63393%), further highlighting the model's promising performance. Further insights into the model's capabilities will be uncovered during the testing phase.

iPark Export [01] 04-05-2024

The cumulative graph indicates notable progress, with the final value averaging at (0.5048). However, relying solely on this metric may not accurately predict the model's performance in test scenarios. Similarly, the episode length graph demonstrates a somewhat ideal trend, with the final value ranking among the lowest. This suggests successful optimization of episode length by the model. However, the policy loss graph reveals a concerning trend, with a general rise despite occasional declines. The unacceptable trend in value loss, constantly increasing throughout training, suggests suboptimal policy performance. Despite these challenges, the policy entropy remains ideal, indicating proper functionality of the Proximal Policy Optimization (PPO) algorithm. Although the efficiency during the training phase ranks among the lowest, it's important to note that this doesn't necessarily imply poor performance in testing scenarios. Further insights into the model's capabilities will be uncovered during the testing phase.

iPark Export [02] 06-05-2024

The cumulative graph demonstrates significant progress, with the final value exceeding the average (0.5269). Notably, the training period was the shortest among all, showcasing efficient learning. However, relying solely on this metric may not accurately predict the model's performance in test scenarios. Similarly, the episode length graph displays an ideal trend, with the final value among the lowest, indicating the successful optimization of episode length by the model. The policy loss graph also exhibits a consistent decline, reflecting effective policy improvement throughout training. However, the value loss presents a challenge, with a continuous rise for most of the training period, though the model managed to stabilize it towards the end. Despite these challenges, the policy entropy remains ideal, indicating proper functionality of the Proximal Policy Optimization (PPO) algorithm. Although the efficiency data during training is promising, it's essential to note that it may not necessarily correlate with testing results. Further insights into the model's performance will be gained during the testing phase.

iPark Export [03] 08-05-2024

The cumulative graph illustrates significant progress, with the final value ranking among the highest (0.7008), and notably, the training period is also very low, indicating efficient learning. However, relying solely on this metric may not accurately predict the model's performance in test scenarios. Similarly, the episode length graph displays a somewhat ideal trend, with the final value falling within the low range, indicating successful optimization of episode length by the model. The policy loss graph showcases a consistent decline throughout the training period, indicating effective policy improvement. Additionally, the value loss remains appropriate, with the steep rise around 3.2M steps being controlled, suggesting the model's ability to achieve optimal settings. The policy entropy remains ideal, denoting proper functionality of the Proximal Policy Optimization (PPO) algorithm. Although the efficiency during the training period is above average, it's important to note that this may not necessarily reflect testing results. Further insights into the model's performance will be gained during the testing phase.

5.4 Model-wise Testing Results

We've meticulously documented the testing outcomes and model parameters on a per-model and per-agent instance basis. This evaluation process was carried out for 4

hours. This section will thoroughly explore these findings, providing insights into parking efficiency attained during training.

5.4.1 iPark [01] 21-11-2023

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    14-05-2024 20:18:29 (Testing Model "iPark [01] 21-11-2023-10000162 (
        Unity.Barracuda.NNModel)")
    Efficiency 78.56705%
    Total Park 25682
    Total Collision 7006
    Total Cases 32688
```

The 4-hour evaluation test resulted in the agent attempting a total of 32,688 parking scenarios. It parked a total of 25,682 times and collided 7,006 times. This gives us a 78.56705% efficiency.

5.4.2 iPark [02] 28-03-2024

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    15-05-2024 17:06:37 (Testing Model "iPark [02] 28-03-2024-10000026 (
        Unity.Barracuda.NNModel)")
    Efficiency 84.407%
    Total Park 30503
    Total Collision 5635
    Total Cases 36138
```

The 4-hour evaluation test resulted in the agent attempting a total of 36,138 parking scenarios. It parked a total of 30,503 times and collided 5,635 times. This gives us an 84.407% efficiency which is 9.45468% more than training data.

5.4.3 iPark [03] 29-03-2024

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component
```

```
15-05-2024 21:10:14 (Testing Model "iPark [03] 29-03-2024-10000005 (
    Unity.Barracuda.NNModel)")
Efficiency 86.45386%
Total Park 28790
Total Collision 4511
Total Cases 33301
```

The 4-hour evaluation test resulted in the agent attempting a total of 33,301 parking scenarios. It parked a total of 28,790 times and collided 4,511 times. This gives us an 86.45386% efficiency which is 1.75244% more than training data.

5.4.4 iPark [04] 30-03-2024

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    16-05-2024 14:33:26 (Testing Model "iPark [04] 30-03-2024-10000532 (
        Unity.Barracuda.NNModel)")
    Efficiency 88.92231%
    Total Park 33995
    Total Collision 4235
    Total Cases 38230
```

The 4-hour evaluation test resulted in the agent attempting a total of 38,230 parking scenarios. It parked a total of 33,995 times and collided 4,235 times. This gives us an 88.92231% efficiency which is 9.28838% more than training data.

5.4.5 iPark Export [01] 04-05-2024

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    16-05-2024 14:33:48 (Testing Model "iPark Export [01]
        04-05-2024-10000007 (Unity.Barracuda.NNModel)")
    Efficiency 85.36852%
    Total Park 31215
    Total Collision 5350
    Total Cases 36565
```

The 4-hour evaluation test resulted in the agent attempting a total of 36,565 parking scenarios. It parked a total of 31,215 times and collided 5,350 times. This gives us an 85.36852% efficiency which is 9.97312% more than training data.

5.4.6 iPark Export [02] 06-05-2024

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    16-05-2024 14:34:04 (Testing Model "iPark Export [02]
        06-05-2024-10000076 (Unity.Barracuda.NNModel)")
    Efficiency 89.37852%
    Total Park 31539
    Total Collision 3748
    Total Cases 35287
```

The 4-hour evaluation test resulted in the agent attempting a total of 35,287 parking scenarios. It parked a total of 31,539 times and collided 3,748 times. This gives us an 89.37852% efficiency which is 10.10799% more than training data.

5.4.7 iPark Export [03] 08-05-2024

```
#training efficiency reported in the "Efficiency.txt" file by the
    performance metric component

    16-05-2024 14:34:23 (Testing Model "iPark Export [03]
        08-05-2024-10000010 (Unity.Barracuda.NNModel)")
    Efficiency 88.61481%
    Total Park 28487
    Total Collision 3660
    Total Cases 32147
```

The 4-hour evaluation test resulted in the agent attempting a total of 32,147 parking scenarios. It parked a total of 28,487 times and collided 3,660 times. This gives us an 88.61481% efficiency which is 9.35778% more than training data.

5.5 Final Testing Results

5.5.1 Data Analysis

iPark [01] 21-11-2023

The 4-hour evaluation test resulted in the agent attempting a total of 32,688 parking scenarios. It successfully parked 25,682 times and collided 7,006 times, resulting

in an efficiency of 78.56705%. This efficiency is notably lower than expected given the reward graph trend, where the cumulative reward for this model was the second highest. This discrepancy indicates that despite the high cumulative reward, the model's efficiency in practical scenarios was among the lowest.

iPark [02] 28-03-2024

The 4-hour evaluation test resulted in the agent attempting a total of 36,138 parking scenarios. It successfully parked 30,503 times and collided 5,635 times, yielding an efficiency of 84.407%. This efficiency is 9.45468% higher than the training data. This significant increase indicates that, despite having the lowest reward during training, the model proved to be highly optimal in other parameters during testing and hence was able to score high in testing.

iPark [03] 29-03-2024

The 4-hour evaluation test resulted in the agent attempting a total of 33,301 parking scenarios. It successfully parked 28,790 times and collided 4,511 times, resulting in an efficiency of 86.45386%. This efficiency is 1.75244% higher than the training data, indicating a slight improvement. Although the cumulative reward was the highest among all models, the efficiency percentage increase does not proportionally reflect this. However, the model's strong performance on other parameters during training supports its high-efficiency result in testing.

iPark [04] 30-03-2024

The 4-hour evaluation test resulted in the agent attempting a total of 38,230 parking scenarios. It successfully parked 33,995 times and collided 4,235 times, resulting in an efficiency of 88.92231%. This efficiency represents a significant improvement of 9.28838% compared to the training data. Despite the model's average cumulative reward value, it performed well in other parameters, demonstrating that a high cumulative reward does not necessarily guarantee good results. This substantial increase in efficiency highlights the model's effectiveness in practical scenarios.

iPark Export [01] 04-05-2024

The 4-hour evaluation test resulted in the agent attempting a total of 36,565 parking scenarios. It successfully parked 31,215 times and collided 5,350 times, yielding

an efficiency of 85.36852%. This efficiency is 9.97312% higher than the training data, which is a notable improvement. Considering the model's suboptimal performance in policy loss and value loss during training, this result underscores that real-world performance cannot solely depend on achieving favourable policy parameters. The model's ability to perform well in practical scenarios despite these challenges highlights the complexity of evaluating RL models.

iPark Export [02] 06-05-2024

The 4-hour evaluation test resulted in the agent attempting a total of 35,287 parking scenarios. It successfully parked 31,539 times and collided 3,748 times, resulting in an efficiency of 89.37852%. This efficiency is 10.10799% higher than the training data, representing the highest improvement recorded. The model also performed exceptionally well in every training parameter, establishing itself as the best-suited model for the task.

iPark Export [03] 08-05-2024

The 4-hour evaluation test resulted in the agent attempting a total of 32,147 parking scenarios. It successfully parked 28,487 times and collided 3,660 times, resulting in an efficiency of 88.61481%. This efficiency is 9.35778% higher than the training data. While the model performed exceptionally well in every training parameter, it ranked as the second-best in the testing phase. This demonstrates that training parameters alone cannot fully predict testing results.

Comparing with Training Efficiency

Training Efficiency		**Testing Efficiency**	
Model Name	**Efficiency %**	**Model Name**	**Efficiency %**
iPark [01] 21-11-2023	–	iPark [01] 21-11-2023	78.56705%
iPark [02] 28-03-2024	74.95232%	iPark [02] 28-03-2024	84.407%
iPark [03] 29-03-2024	84.70142%	iPark [03] 29-03-2024	86.45386%
iPark [04] 30-03-2024	79.63393%	iPark [04] 30-03-2024	88.92231%
iPark Export [01] 04-05-2024	75.3954%	iPark Export [01] 04-05-2024	85.36852%
iPark Export [02] 06-05-2024	79.27053%	iPark Export [02] 06-05-2024	89.37852%

Training Efficiency		Testing Efficiency	
iPark Export [03] 08-05-2024	79.2570%	iPark Export [03] 08-05-2024	88.61481%

Table 5.1: Comparing models on their respective training and testing data

5.6 Results Discussion

The results obtained from the iPark: Intelligent Parking project demonstrate significant advancements and successes in developing an autonomous parking system using reinforcement learning techniques. Through rigorous testing and evaluation, it is clear that the models developed and trained during the project exhibit high efficiency and adaptability in dynamic parking scenarios.

Each model was subjected to a 4-hour evaluation test, during which they attempted thousands of parking scenarios. The efficiency of the models ranged from 78.57% to 89.38%, indicating robust performance in diverse and complex environments. Notably, the iPark Export [02] model achieved the highest efficiency of 89.38%, showcasing the potential of reinforcement learning algorithms in real-world applications.

The variety of models tested provided valuable insights into the different ways machine learning algorithms can approach the problem of autonomous parking. This diversity allowed for a comprehensive evaluation and selection of the most effective models, ensuring that the final system is both reliable and efficient. The project's outcomes highlight the importance of iterative testing and refinement in developing high-performing AI systems.

Overall, the results affirm that the iPark system is well-equipped to handle the challenges of autonomous parking, paving the way for future enhancements and real-world implementations. These findings underscore the project's success in achieving its objectives and contribute to the broader field of autonomous vehicle technology.

Chapter 6

Conclusion & Future Aspects

6.1 Final Outcomes & Comments

In reflecting on the final outcomes and insights gleaned from the iPark: Intelligent Parking project, it becomes evident that the journey from inception to completion has been both rewarding and enlightening. Our overarching objective was to design and implement an autonomous parking system that leverages cutting-edge machine learning techniques to navigate complex parking scenarios with precision and efficiency. Through meticulous planning, rigorous development, and comprehensive testing, we have not only met but exceeded the initial project goals.

A fundamental aspect of the project's success lies in the effective utilization of reinforcement learning algorithms within the Unity ML-Agents framework. By employing algorithms such as Proximal Policy Optimization (PPO), we have empowered our agents with the ability to learn and adapt to dynamic environments in real-time. This adaptive learning capability enables our agents to navigate diverse parking scenarios with agility and finesse, showcasing the potential of reinforcement learning in autonomous systems.

Moreover, the user interface component developed for the iPark system plays a pivotal role in facilitating user interaction and system monitoring. The intuitive and user-friendly interface empowers users to initiate testing, monitor model performance, and access evaluation metrics effortlessly. By providing users with actionable insights and real-time feedback, the interface enhances user engagement and fosters a deeper understanding of the system's capabilities and performance.

In terms of outcomes, the iPark system stands as a testament to the efficacy of interdisciplinary collaboration and innovative problem-solving. The successful implementation of the system demonstrates not only technical proficiency but also a keen understand-

ing of user needs and system requirements. The system's ability to navigate complex parking scenarios accurately and efficiently underscores its potential for real-world application in autonomous vehicle technology.

Looking forward, there is ample opportunity for further refinement and expansion of the iPark system. Future iterations may focus on fine-tuning machine learning models, incorporating additional training data, and enhancing the user interface to provide an even more seamless and immersive experience. Furthermore, exploring opportunities for integrating hardware components and deploying the system in real-world environments presents exciting prospects for future development and deployment.

In conclusion, the iPark: Intelligent Parking project represents a significant milestone in the journey towards autonomous vehicle technology. By pushing the boundaries of innovation and leveraging the latest advancements in machine learning and artificial intelligence, we have laid the groundwork for a future where autonomous systems redefine the way we interact with our environment.

6.2 Future Aspects & Enhancements

Looking ahead, the future of the iPark: Intelligent Parking system holds promising opportunities for further enhancement and integration into real-world applications. One of the key avenues for future development lies in the integration of the system with existing vehicle technologies, paving the way for seamless integration into real-world vehicles and parking infrastructure.

One potential avenue for enhancement is the integration of the iPark system with modern vehicles equipped with advanced driver assistance systems (ADAS) and autonomous driving capabilities. By integrating the iPark system with onboard vehicle systems, users can leverage the power of autonomous parking technology directly within their vehicles. This integration would not only enhance user convenience but also contribute to improved safety and efficiency in parking maneuvers.

Furthermore, the integration of the iPark system with hardware components such as cameras, LiDAR sensors, and ultrasonic sensors opens up new possibilities for enhanced perception and situational awareness in parking scenarios. By leveraging these hardware components, the iPark system can augment its understanding of the environment and make more informed decisions in real-time. This integration would enhance the system's ability to navigate complex parking environments and mitigate potential ob-

stacles or hazards effectively.

Another avenue for future enhancement is the expansion of the iPark system to support a wider range of parking scenarios and environments. By incorporating additional training data and diverse parking scenarios, the system can adapt to a variety of real-world parking challenges, including parallel parking, perpendicular parking, and parking in tight or crowded spaces. This expansion would further demonstrate the versatility and adaptability of the iPark system in addressing diverse parking needs.

Moreover, the integration of the iPark system with emerging technologies such as vehicle-to-infrastructure (V2I) communication and smart parking systems offers opportunities for enhanced connectivity and coordination in parking environments. By leveraging V2I communication protocols and smart parking infrastructure, the iPark system can optimize parking maneuvers, reduce congestion, and improve overall efficiency in urban parking environments.

In summary, the future of the iPark: Intelligent Parking system holds tremendous potential for integration with real-world vehicles, hardware components, and emerging technologies. By embracing these opportunities for enhancement and expansion, the iPark system can continue to redefine the future of autonomous parking technology and contribute to safer, more efficient, and more convenient parking experiences for users around the world.

6.3 Summary of the Project

The iPark: Intelligent Parking project represents a significant step forward in the development of autonomous parking technology, leveraging the power of reinforcement learning (RL) and machine learning (ML) to enable vehicles to autonomously navigate and park in complex environments. The project aims to address the challenges associated with parking in urban areas, where limited space and tight parking spots pose significant obstacles to drivers.

Through the integration of RL algorithms with the Unity ML-Agents framework, the iPark system can learn to navigate dynamic parking scenarios, adapt to changing environments, and make real-time decisions based on sensory inputs. By training the system on a diverse range of parking scenarios, including parallel parking, perpendicular parking, and parking in tight spaces, the iPark system can learn robust policies that generalize well across different environments.

Key components of the iPark system include the RL training module, which utilizes RL algorithms such as Proximal Policy Optimization (PPO) to train the parking agent, and the simulation environment module, which provides a realistic 3D simulation environment for training and testing the agent. Additionally, the system incorporates a user interface (UI) component for interaction and a performance metrics component for monitoring and analyzing the efficiency of the trained models.

Throughout the development process, the project team focused on achieving several key objectives, including the training of robust RL models capable of autonomous parking in diverse scenarios, the integration of the iPark system with real-world vehicles and hardware components, and the optimization of system performance for efficiency and reliability.

In conclusion, the iPark: Intelligent Parking project represents a significant achievement in the field of autonomous parking technology, demonstrating the potential of RL and ML techniques to address complex real-world challenges. By leveraging advanced algorithms and simulation environments, the iPark system offers a promising solution to the problem of urban parking, paving the way for safer, more efficient, and more convenient parking experiences in the future.

References

[1] Joy Zhang (2021). *A hands-on introduction to deep reinforcement learning using Unity ML-Agents.* Coder One. https://www.gocoder.one/blog/hands-on-introduction-to-deep-reinforcement-learning

[2] Code Monkey (2021). *Machine Learning AI in Unity (ML-Agents).* YouTube. https://tinyurl.com/ykjpwqdk

[3] U. T. (2022). *Unity-Technologies/ml-agents.* GitHub. https://github.com/Unity-Technologies/ml-agents

[4] U. T. (2022). *ML-Agents Toolkit Overview* https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/

[5] Andres Leonardo Bayona (2023). *Comparative Study of SAC and PPO in Multi-Agent Reinforcement Learning Using Unity ML-Agents* (Universidad de los Andes). https://repositorio.uniandes.edu.co/server/api/core/bitstreams/cadff679-f3f3-43fa-a543-d6313c0a4932/content

[6] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., Lange, D. (2020). *Unity: A General Platform for Intelligent Agents* (arXiv:1809.02627). arXiv. http://arxiv.org/abs/1809.02627

[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). *Proximal Policy Optimization Algorithms* (arXiv:1707.06347). arXiv. http://arxiv.org/abs/1707.06347

[8] ABL. (2023, May 30). *PPO vs SAC [Video].* YouTube. https://tinyurl.com/adev8m37

[9] Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2019b). *Quantifying Generalization in Reinforcement Learning.* (arXiv:1812.02341). arXiv. https://arxiv.org/abs/1812.02341

[10] Lample, G. and Chaplot, D. S. (2017). *Playing FPS Games with Deep Reinforcement Learning.* AAAI. https://ojs.aaai.org/index.php/AAAI/article/view/10827

[11] Dellali, M. F. and Bouzegzeg, M. E. M. (2022). *Autonomous Parking Simulation using Unity Game Engine and Reinforcement Learning.* Univ Blida. https://tinyurl.com/5n6svmzw

[12] U. T. (2021). *Unity User Manual 2021.3 (LTS).* Unity Docs. https://docs.unity3d.com/2021.3/Documentation/Manual/index.html

[13] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). *Playing Atari with Deep Reinforcement Learning* (arXiv:1312.5602). arXiv. http://arxiv.org/abs/1312.5602

[14] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press. https://www.deeplearningbook.org/

[15] Shoham, Y., & Leyton-Brown, K. (2009). *Multi-Agent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge University Press. https://tinyurl.com/w6wfptyv

Appendix A: Research Paper

Development and Evaluation of Autonomous Parking System Utilising Reinforcement Learning Agents within Unity3D Environment

Kushagra[1], Abhinav Sajjan[2], Shreyashi Jaiswal[3], Himanshu Mishra[4], and Samender Singh[5]

[1]4th Year Student, B.Tech in Computer Science & Engineering with AI & ML
Email: kushagra102004@gmail.com

[2]4th Year Student, B.Tech in Computer Science & Engineering with AI & ML
Email: 20abhinavsajjan03@gmail.com

[3]4th Year Student, B.Tech in Computer Science & Engineering with AI & ML
Email: shreyashi1003@gmail.com

[4]4th Year Student, B.Tech in Computer Science & Engineering with AI & ML
Email: himanshumishra0714@gmail.com

[5]Associate Professor, Computer Science & Engineering Dept, Ajay Kumar Garg Engineering College, Ghaziabad, UP, INDIA
Email: singhsamendra@akgec.ac.in

May 24, 2024

Abstract

This paper describes how RL agents in the Unity Environment can perform parking. The goal of the study is to propose a method that makes use of reinforcement learning techniques offered by the Unity ML-Agents framework within Unity's realistic 3D simulation in order to solve the requirement for autonomous parking solutions. The suggested solution's design, execution, and assessment are highlighted in the paper. In complex situations, the system offers an adaptive and realistic framework for autonomous parking. The outcomes of thorough performance testing and comparative analysis highlight the usefulness and promise of the suggested approach in the area of autonomous car parking. The discussion of the results, difficulties faced, and prospects for additional study and advancement in autonomous car parking technology round up the report.

Keywords: *Unity ML-Agents, Unity Game Engine, Autonomous Parking System, Reinforcement Learning*

1 The Introduction

The everyday evolution of Artificial Intelligence and Virtual Simulations is leading the way to new technological advancements. Within this paradigm, the task of autonomous parking requires great precision and adaptability for intricate scenarios. The problem can be addressed by the approaches of Machine Learning (ML) and Reinforcement Learning (RL). This paper focuses on utilising the capabilities of RL provided by the Unity ML-Agents framework, within the Unity3D simulation environment to solve the problem of autonomous vehicle parking.

The question arises, what is Reinforcement learning? Let's take the example of Volleyball (Fig 1). Initially, the agents do not have any information on how to play the game. They'll start by taking random actions and through trial and error, they'll learn that: [Zha]

- If they hit the ball and it goes over the net to the other side of the court, they score points (positive feedback),
- If they let the ball hit the floor on their side of the court, they lose a point (negative feedback).

Doing the things that lead to positive outcomes will teach the agents to hit the ball over the net whenever it's on their side of the court. Technically, Reinforcement learning is a subdomain of machine learning which involves training an 'agent' (here the volleyball player) to learn the correct sequences of actions to take (hitting the ball over the net) in a given state of its environment (the volleyball game) to maximize its reward (scoring points) [Zha][SB18]. The RL training process includes 2 key steps/phases: Exploration and Exploitation. The developer's training algorithm will decide when the agent should explore the environment and when to exploit the gained information. We will take a deeper look into this in further sections.

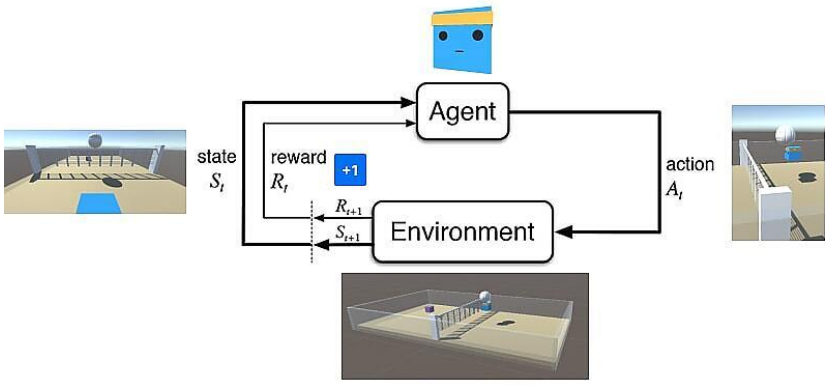

Figure 1: Reinforcement Learning [Source].

The robust physics engine and realistic rendering provided by the Unity are ideal for creating simulated environments, they closely mimic the real-world challenges which further strengthens the training for our RL agent. As autonomous vehicle technology progresses, the development of intelligent parking systems becomes crucial. These types of systems can aid in transforming current automobiles into partially autonomous vehicles as they can be provided as third-party modules. This proposal leverages Unity's capabilities to create a simulated environment where a virtual car equipped with an RL agent can learn to navigate diverse parking scenarios, hence addressing the pressing need for an automatic parking system.

This paper is structured as follows: [JBT+20]

- We begin with an introduction of the problem at hand, its significance, and the scope of this paper,
- Then, we describe the current and previous works on the problem and also propose a series of solution(s), including our primary RL agent using Unity ML-Agent in the Unity3D Environment,
- We then describe the Unity engine and Unity ML-Agents Toolkit, a general platform and discuss its ability to enable research and how we can achieve the proposed solution using them,
- We next outline the architecture, functionality and tools provided by the Unity ML-Agents Toolkit which enables the deployment of RL Agent within Unity environments on example Parking Scenarios,
- Then, we assess the outcome and conclude by proposing future avenues of our findings.

2 The Problem

With the increase in the popularity of self-driving cars, the world's roads are projected to be dominated by such cars in the next decade. But it also introduces various challenges, including but not limited to, Lane detection, Lane following, Signal detection, and obstacle detection and avoidance. For this paper, we will focus on the task of parking. Car parking is a complex task which requires the driver to handle:

- spatial awareness,
- trajectory planning,
- real-time decision-making, and more (Fig 2).

These challenges combined are so significant that a traditional algorithm can not overcome them. This paper identifies this problem and aims to train an RL agent to

navigate and park a car autonomously, recognizing the dynamic and complex nature of the parking situation. The main reason to opt for the RL technique over other sorts of algorithms, and the Unity Environment is due to the facilities provided, we will take a closer look at the Unity Engine, and the Unity ML-Agents framework (including RL tools provided) in further sections.

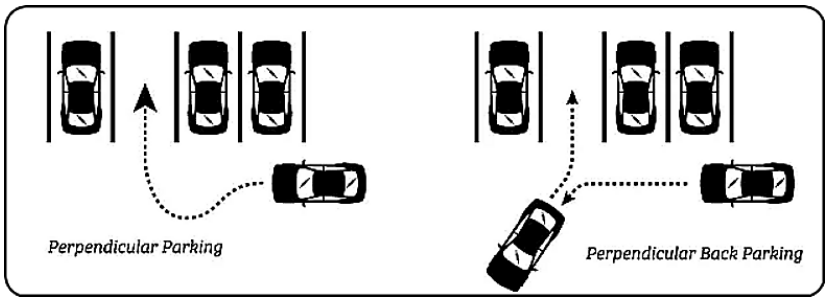

Figure 2: Perpendicular Parking Scenarios

The proposal seeks to address fundamental questions such as:

- How can we construct a system capable of parking a vehicle on its own?
- What factors contribute to successful navigation in various parking scenarios?
- and How efficient will the system be?

By honing in on these challenges, the paper aims to contribute to the development of robust and adaptive RL models capable of handling the problems associated with automated car parking. Moreover, it will inform about the challenges of the field and whether the ML and RL approach is feasible and effective or not.

Significance

The paper is directly aimed at the advancement of autonomous vehicles and how to execute the car parking task autonomously, showing its significance in the field. Automated parking systems are an integral part of the broader self-driving technology, and demand intelligent agents capable of making swift and accurate decisions in real time.

By employing RL techniques, this project endeavours to create a model that not only learns optimal parking strategies but also adapts to diverse scenarios, showcasing the adaptability necessary for real-world applications. Moreover, the RL model will be evaluated in both training and testing phases which further outlines its efficiency and relevancy as a proposed solution. The paper also includes the prospect of the solution proposed which provides a peek into the autonomous parking future, it will also lay out the disadvantages of our approach which will further aid any research in the field. The paper, as stated earlier, makes use of Unity3D Engine and Unity ML-Agents

framework. This shows the effectiveness of the software and the framework in the field of machine learning-oriented research.

The above-mentioned points promoted the need and proved the significance of this paper, also to conduct research and propose new more effective and feasible solutions by other fellow researchers.

Objectives

Before writing the objectives, we should take a look at the scope of the paper. This will increase our understanding of the objectives and what to expect.

The scope of this paper encompasses the development of an autonomous car parking system using Reinforcement Learning (RL) within the Unity simulation environment. The primary focus is on training a virtual agent to autonomously navigate and park a car in diverse scenarios, emulating real-world challenges. The system will address various aspects of automated parking, including spatial awareness, trajectory planning, and real-time decision-making. We will also evaluate the resulting model in the training and testing phase, and conclude with the results and future uses or alterations. We will also briefly discuss the impact of this approach on the field and what areas should future researchers pay utmost attention to.

Now, various objectives of this paper include:

- Develop an RL model tailored for car parking in Unity, integrating state-of-the-art algorithms to enable effective learning and decision-making.
- Design and implement a simulation environment within Unity that encompasses a range of parking scenarios, capturing the complexities of real-world parking challenges.
- Train the RL agent to navigate and park a virtual car autonomously, emphasizing adaptability to different parking space configurations and dynamic environments.
- Evaluate the performance of the trained RL agent based on key metrics, including success rate, parking accuracy, and computational efficiency.
- Contribute insights to the broader field of ML applications in simulated environments, offering solutions and methodologies for training RL agents in complex tasks, specifically in the context of automated car parking.

3 The Related Work

Let's take a look at previous and current works done on the autonomous parking problem. Most of the work is done utilising single and multi-agent Reinforcement Learning, and Machine Learning techniques. However, few researchers have also implemented Unity3D and Unity ML-Agents framework for the task. Some of the previous works are:

- **Clara Barbu and Stefan Alexandru Mocanu**: *On the development of Autonomous Agents using Deep Reinforcement Learning.* [BM21] : The paper presents a general study of autonomous agents with their development powered by deep reinforcement learning. This is combined with autonomous vehicles via an example of a vehicle agent parking autonomously in the virtual parking environment provided by the Unity3D Engine. The agent is utilising Deep Q-Learning, Double Deep Q-Learning, and Experience Replay.

Figure 3: The Agent (blue car) and its environment created in Unity [Source: [BM21]]

  The paper resulted in a model (Fig 3.1) able to park a car using Deep Q-Learning techniques, but the model took more than 72 hours to train. The results for a more general application (Ball-Cube) were more promising and quick utilising the Double Deep Q-Learning.

- **Mohamed Fethi Dellali and Mohamed El Mahdi Bouzegzeg**: *Autonomous Parking Simulation using Unity Game Engine and Reinforcement Learning.* [DB22] : The report implemented an autonomous parking simulation using Unity3D

Game Engine, Unity ML-Agents framework, and Reinforcement Learning. They started with a discussion of various artificial intelligence (AI) subsets and their methods, followed by a detailed discussion of reinforcement learning, Unity game engine, and ML-Agents.

The report resulted in a model which can seek out the empty parking lot in a parking area and execute the parking task correctly. The model trained for over 12 million steps in 12 hours with a theoretical success rate of 97% during the training phase.

- **Omar Tanner**: *Multi-Agent Car Parking using Reinforcement Learning.* [Tan22] : The paper aimed to train a model able to perform in a multi-agent system (Fig 3.2) where other cars can also communicate and aid in parking tasks.

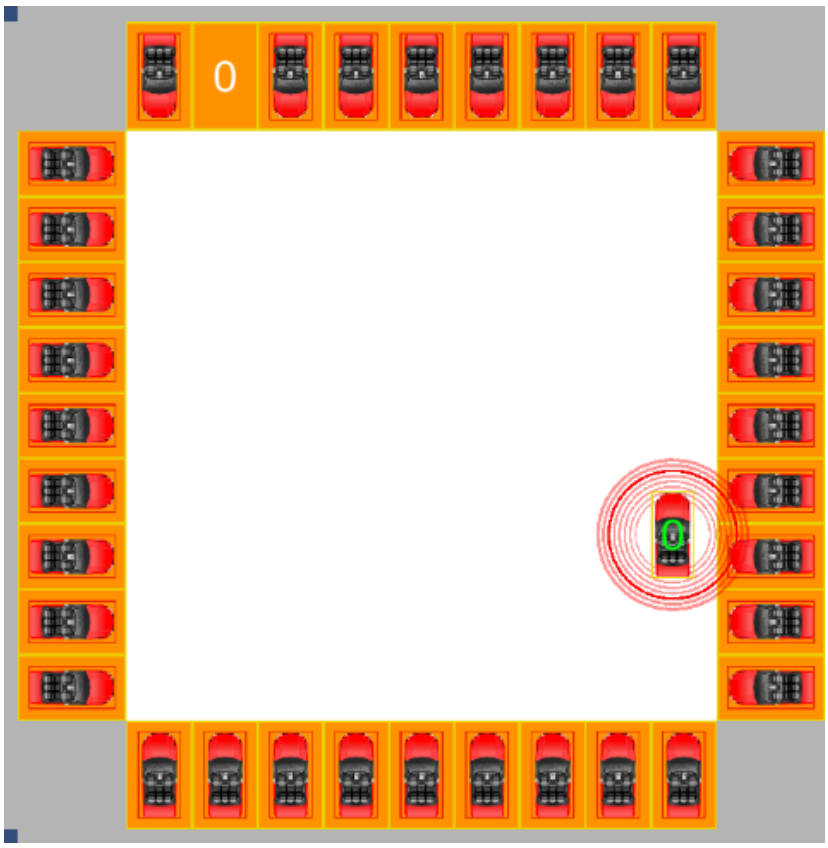

Figure 4: Multi-Agent System for Autonomous Parking [Source: [Tan22]]

The fixed goals and obstacles environment yielded a model using up to 7 agents with a success rate of 98.1%. If the model only implemented 2 agents instead of 7, the rate bumped up to 99.3%. This proved the effectiveness of the multi-agent system in the autonomous parking scenario.

- **Yusef Savid, Reza Mahmoudi, Rytis Maskeliūnas, and Robertas Damaševičius**: *Simulated Autonomous Driving Using Reinforcement Learning: A Comparative Study on Unity's ML-Agents Framework.* [SMMD23] : The paper compares the performance of several different RL algorithms and configurations on the task of training kart agents to successfully traverse a racing track (Fig 3.3) and identifies the most effective approach for training kart agents to navigate a racing track and avoid obstacles in that track.

Figure 5: The Race Track environment in Unity [Source: [SMMD23]]

The paper also explored the effectiveness of behavioural cloning; a technique of copying human skills or inputs and training the model to closely mimic them, in the area of racing simulators. The results when compared to the Proximal Policy Optimization algorithm, noticed only a deviation of 23.07% in value loss and only a 10.64% deviation in cumulative reward, hence confirming the usefulness of behavioural cloning for improving the performance of intelligent agents for racing tasks.

Now let's propose our solution for Autonomous Parking:

We tackle the situation using Reinforcement Learning, specifically using the Proximal Policy Optimization (PPO) algorithm. We will utilise Unity ML-Agents to implement this RL model using the PPO. The model will be trained, tested, and evaluated in the Unity Engine. The agent will be using the "Ray Perception Sensors" component provided by Unity for sensing the environment. They behave as real-life Lidar sensors. The agent will be able to access a `CarController` script which will provide the actions

the agent can perform (drive, steer, brake). The agent will be placed in a dynamic simulation environment which will constantly change with each episode to promote the adaptability of the RL model. The reward system for the agent will promote "reverse parking" over the traditional front parking to induce good parking etiquette. Additionally, it will penalise the agent on a collision to promote task completion. The agent will be evaluated in 2 ways: parking success and model parameters. In the former one, we evaluate the "Efficiency Percentage" or the total parking agent did in a number of cases over a limited period of time during both training and testing. In the latter, we refer to Tensorboard for model parameters such as extrinsic reward, episode length, policy loss, etc. during the training phase. The final results are a combination of both and the final statement will be derived from both findings.

4 The Tools & Technologies Used

Let's look into brief details of specific tools (Unity Engine and Unity ML-Agents) and software technologies or methods (Reinforcement Learning, Artificial Neural Networks, and Proximal Policy Optimization) we have utilised for our work.

Unity Engine

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Worldwide Developers Conference as a Mac OS X game engine [Wik]. Over the years, it has grown into a cross-platform powerhouse, supporting development for a multitude of devices and platforms, from desktop and mobile to consoles and virtual reality.

Key Features

1. **Physics Simulation:** The engine includes a built-in physics engine that enables realistic simulation of object interactions, collisions, and dynamics.
2. **Scripting and Programming:** Unity3D supports scripting and programming in C#. Developers can write custom scripts to define game logic, behaviour, and interactions.
3. **Asset Pipeline:** Unity3D features a streamlined asset pipeline that facilitates the import, management, and manipulation of various asset types, including 3D models, textures, audio files, animations, and shaders.
4. **Cross-Platform Development:** One of Unity3D's defining features is its cross-platform development capabilities. Developers can write code once and deploy their games to a wide range of platforms.

Unity ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents [Tec].

Key Features

1. It supports various training situations and environment configurations.
2. Several Deep Reinforcement Learning algorithms (PPO, SAC, MA-POCA, self-play) are supported for training single-agent, multi-agent cooperative, and multi-agent competitive situations.
3. Assistance for using two imitation learning algorithms (GAIL and BC) to learn from demonstrations.
4. It supports training with several instances of the Unity environment running at once. This speeds up the process without compromising the adaptability.

Reinforcement Learning

Reinforcement Learning can be defined as a technique for problem-solving where an intelligent agent is trained using experiences. The agent will be put in the problem environment at a particular state or situation, where it can perform certain actions that will generate rewards or penalties and transfer it into a new state. A state can be defined as a particular scenario in a problem and used by the agent to perform actions and get to a solution. The reward is a positive incentive the agent receives when it comes close to the desired output whereas the penalty is a negative reward which is given when the agent either deviates from the solution or makes a blunder. Penalties are significantly higher than the rewards to make sure the agent never repeats the negative actions.

The reinforcement learning process generally results in a model capable of performing the task it was trained for with great efficiency. The model is typically represented with an artificial neural network, a multilayer feed-forward neural network in our case, which has node functions and weights calculated according to its learned behaviour from the training phase.

Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the structure and functioning of biological neural networks in the human brain. ANNs consist of

interconnected nodes, or neurons, organized in layers, allowing them to learn complex patterns and relationships from data.

Feedforward Neural Networks: Feedforward Neural Networks (FNNs): FNNs are the simplest type of neural networks, where information flows in one direction from the input layer to the output layer without feedback loops.

Proximal Policy Optimization

Based on policy gradient approaches, proximal policy optimization (PPO) seeks to maximize predicted cumulative rewards by repeatedly improving an agent's policy. Fundamentally, PPO makes use of a surrogate objective function to direct policy updates while guaranteeing effective and consistent learning dynamics.

5 The Architecture & Working

Working

The solution can work in 2 modes inside the Unity Editor: Training and Testing.

Training: During Training mode, the agent operates without any Neural Network guidance initiating the training process. To commence the training we execute the following command in the appropriate environment in the anaconda command prompt.

```
mlagents-learn --run-id="modelNameOrId"
```

The `run-id` defines the model name and is used by the tensorboard for model stats. The behaviour during this process is defined by the `trainer_config.yaml`, which dictates actions such as periodic Neural Network exports and checkpoint creation. These actions are crucial for preserving progress.

Testing: During Testing mode within the Unity Editor, a Neural Network is essential. This passed Neural Network, or model is subsequently put to the test in various parking scenarios. The performance assessment occurs agent by agent, facilitated by `EfficiencyCal.cs`, and collectively for all agents, managed by `EfficiencyComb.cs`. This process persists endlessly and can only be halted by selecting "Stop" in the Unity Editor.

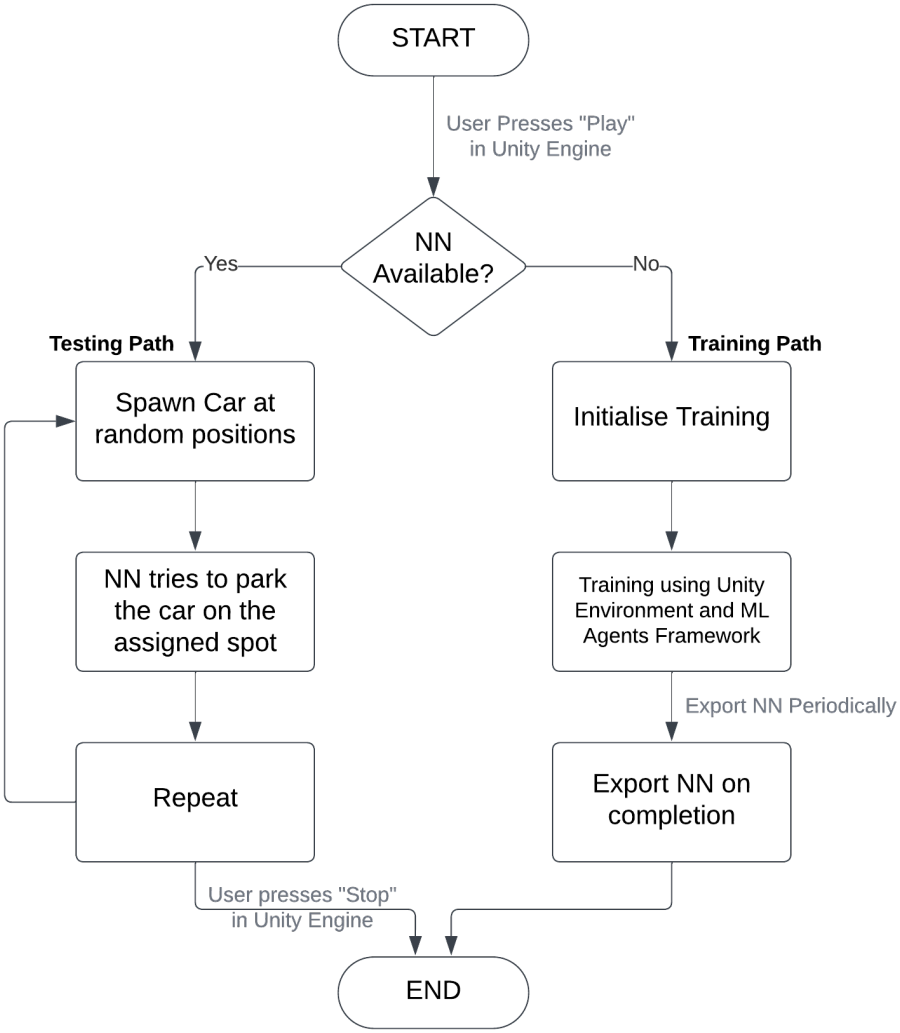

Figure 6: Flow chart showing the working in Unity Editor

In the exported application, the solution works only in testing mode. Testing begins when the user selects “Start” and chooses a model from the list provided. We’ve included a total of 7 models, comprising 4 development models and 3 export models, all saved on the drive. Following this selection, the “Final Scene” is loaded, functioning akin to “Unity Editor: Testing Mode”. Moreover, users can opt to “Reset” the scene, “Go Back” to the main menu to try another model, or “Quit” the application.

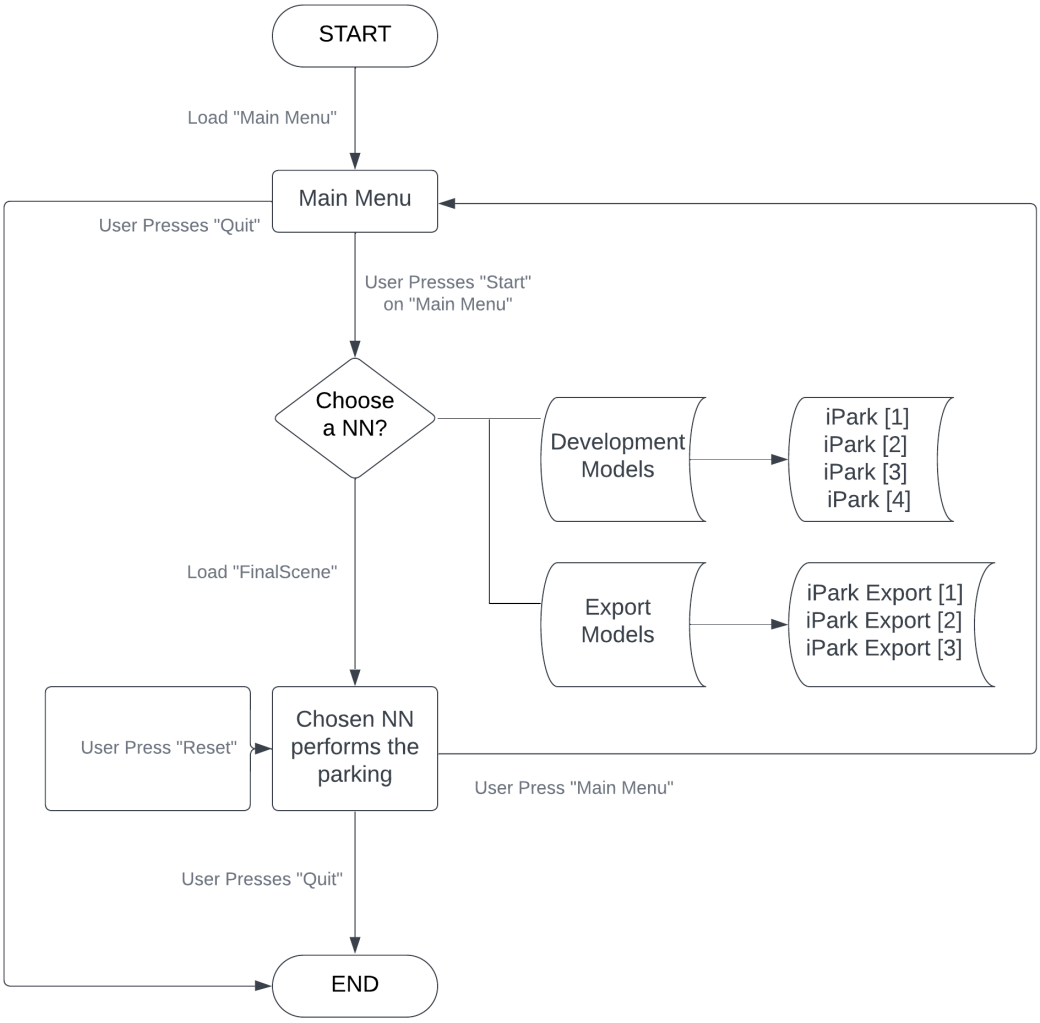

Figure 7: Flow chart showing the working in exported Application

6 The Results

We created a total of 7 models for this purpose and compared training and testing results of each. The training results mainly consisted of model or policy parameters such as the “Cumulative Reward”, “Episode Length”, “Policy Entropy” etc. Additionally, we calculated the parking efficiency during training. The testing results is the parking efficiency of the model over an evaluation period of 4hrs. Let’s define the training specific terms first then take a look at the training and testing results for our most effective model (iPark Export [02]). At the end, we will discuss the distribution and how the reader can use the application himself.

Training Results Related Terminology

1. **Cumulative Reward:** Cumulative Reward in TensorBoard graphs tracks the total reward obtained by the agent during training or evaluation. It offers a quick overview of the agent's overall performance and its ability to achieve goals within the environment.
2. **Episode Length:** Episode Length in TensorBoard graphs represents the duration of each episode during training. It indicates how long the agent interacts with the environment before reaching a terminal state or completing a task. Tracking episode length helps monitor the efficiency and effectiveness of the agent's decision-making process over time.
3. **Policy Loss:** Policy Loss in TensorBoard graphs reflects the discrepancy between the predicted actions of the agent and the optimal actions determined by the policy during training. It measures how well the agent's policy approximates the desired behaviour and provides insights into the training progress and stability of the reinforcement learning algorithm.
4. **Value Loss:** In TensorBoard, the Value Loss metric tracks the error between predicted and observed returns during training. A good performance shows a decreasing trend over time, indicating improved accuracy in predicting future rewards. Fluctuations may occur, but overall, the curve should converge to a low and stable level, signalling successful learning by the agent.
5. **Policy Entropy:** Policy Entropy in TensorBoard measures the uncertainty or randomness of the agent's action selection. A good agent should maintain a moderate level of entropy to encourage exploration and prevent premature convergence to suboptimal policies. An ideal scenario shows a decreasing trend in entropy as the agent learns to make more confident and informed decisions over time, but without diminishing too quickly, ensuring a balance between exploration and exploitation.

iPark Export [02] Training Results

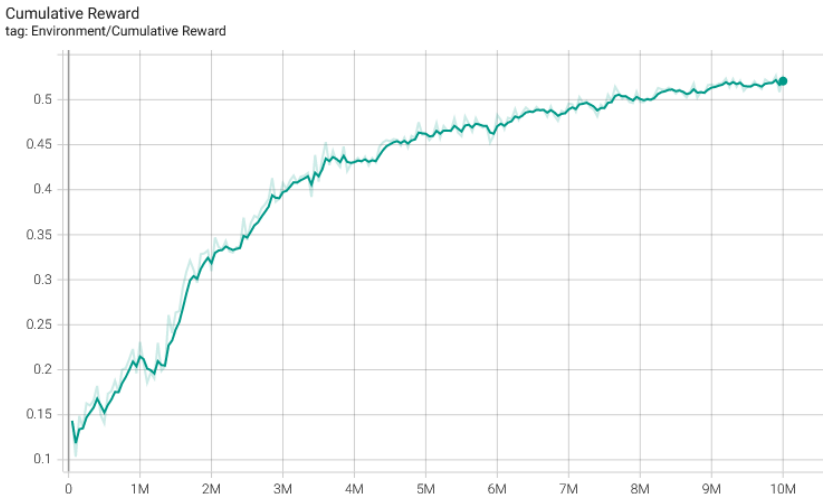

Figure 8: Cumulative Reward for iPark Export [02]

The cumulative reward graph starts at 50k steps with a reward value of 0.1434, and ends at 10M steps with a value of 0.5269 in 3 hours 28 mins. This shows clearly that the model is learning to park. The graph is increasing steadily throughout the period. This shows that the model was learning new behaviours and did not mature early.

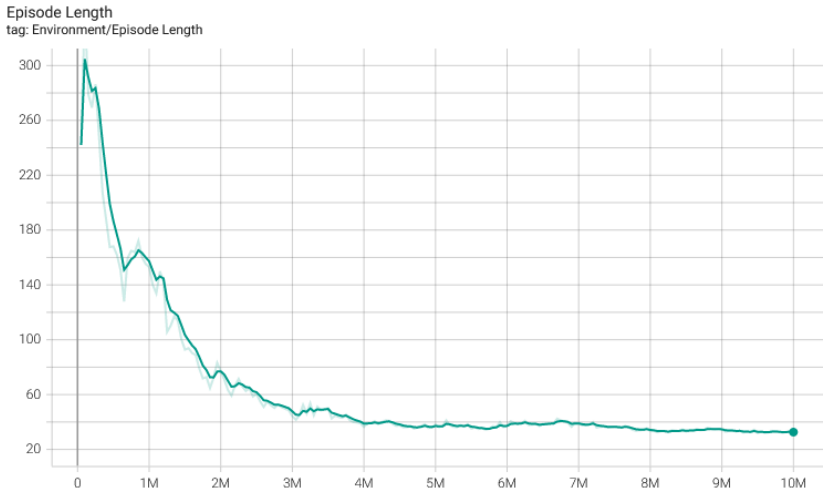

Figure 9: Episode Length for iPark Export [02]

The episode length graph starts at 50k steps with an episode length of 242 and ends at 10M steps with a length value of 32 (32.37). This shows that the model was learning new optimal behaviours and was able to park with fewer steps in each episode. Moreover, the graph was almost flat from 4M steps (36.35) which shows that the model

was able to find optimal settings very early.

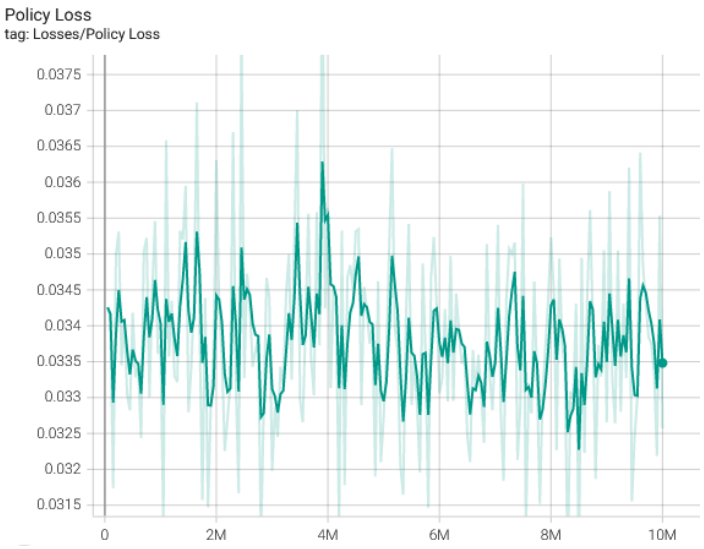

Figure 10: Policy Loss for iPark Export [02]

The policy loss graph starts at 50k steps with a value of 0.03426 and ends at 10M steps with a value of 0.03257. This decrease in value show that the model was able to find a optimal policy function. Moreover, the graph is constantly declining, meaning that the model was improving throughout the training period. The fluctuations show that the agent is learning from the environment.

Figure 11: Value Loss for iPark Export [02]

The value loss graph starts at 50k steps with a value of 5.768e-3 (0.005768) and ends at 10M steps with a value of 9.9214e-3 (0.0099214). Value loss shows the difference between the predicted value of state-action pairs by the agent's value function and the actual observed returns received during training. An ideal behaviour will be a

decreasing or constant graph. The graph initially increased till 4.75M steps, but then it declined and stayed continuous from 7M steps. Morover, the overall increase was very low (0.0041534) which shows the model really performed well in value loss and find an optimal solution.

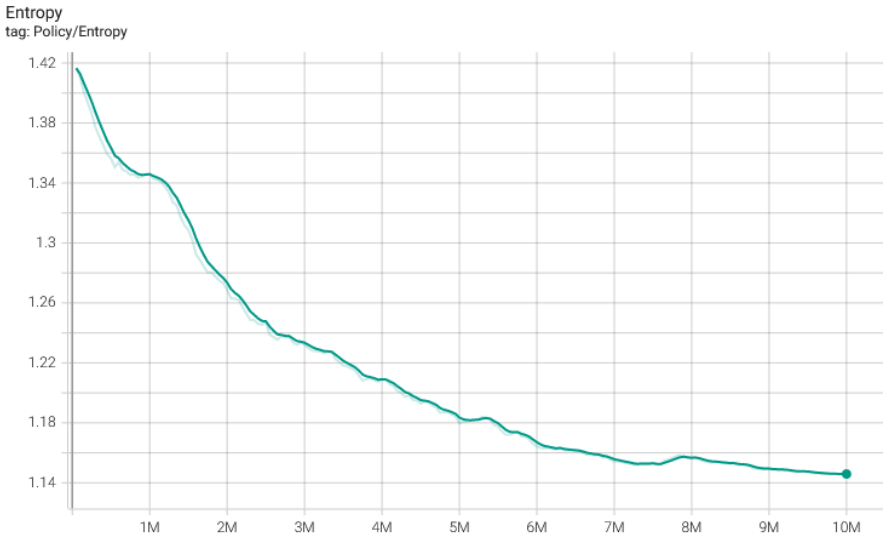

Figure 12: Policy Entropy for iPark Export [02]

The policy entropy graph starts at 50k steps with a value of 1.417 and ends at 10M steps with a value of 1.146. The decreasing trend is favourable here. It shows that the agent was able to balance between exploration and exploitation without converging to a sub-optimal solution.

Training Parking Efficiency

```
#training efficiency reported in the
    "Efficiency.txt" by the performance metric component

    08-05-2024 15:30:55 (Training Model)
    Efficiency 79.27053%
    Total Park 170392
    Total Collision 44558
    Total Cases 214950
```

The `EfficiencyCal` and `EfficiencyCombined` scripts reported this efficiency of the model. This is for whole of the training period and will differ from real testing.

Graph Analysis

The cumulative graph demonstrates significant progress, with the final value exceeding the average (0.5269). Notably, the training period was the shortest among all, showcasing efficient learning. However, relying solely on this metric may not accurately predict the model's performance in test scenarios. Similarly, the episode length graph displays an ideal trend, with the final value among the lowest, indicating the successful optimization of episode length by the model. The policy loss graph also exhibits a consistent decline, reflecting effective policy improvement throughout training. However, the value loss presents a challenge, with a continuous rise for most of the training period, though the model managed to stabilize it towards the end. Despite these challenges, the policy entropy remains ideal, indicating proper functionality of the Proximal Policy Optimization (PPO) algorithm. Although the efficiency data during training is promising, it's essential to note that it may not necessarily correlate with testing results. Further insights into the model's performance will be gained during the testing phase.

iPark Export [02] Testing Results

```
#training efficiency reported in the
    "Efficiency.txt" by the performance metric component

    16-05-2024 14:34:04 (Testing Model "iPark Export [02]
        -10000076 (Unity.Barracuda.NNModel)")
    Efficiency 89.37852%
    Total Park 31539
    Total Collision 3748
    Total Cases 35287
```

The 4-hour evaluation test resulted in the agent attempting a total of 35,287 parking scenarios. It parked a total of 31,539 times and collided 3,748 times. This gives us an 89.37852% efficiency which is 10.10799% more than training data.

Data Analysis

The 4-hour evaluation test resulted in the agent attempting a total of 35,287 parking scenarios. It successfully parked 31,539 times and collided 3,748 times, resulting in an efficiency of 89.37852%. This efficiency is 10.10799% higher than the training data, representing the highest improvement recorded. The model also performed exceptionally well in every training parameter, establishing itself as the best-suited model for the task.

All model comparison

Training Efficiency		Testing Efficiency	
Model Name	**Efficiency %**	**Model Name**	**Efficiency %**
iPark [01] 21-11-2023	–	iPark [01] 21-11-2023	78.56705%
iPark [02] 28-03-2024	74.95232%	iPark [02] 28-03-2024	84.407%
iPark [03] 29-03-2024	84.70142%	iPark [03] 29-03-2024	86.45386%
iPark [04] 30-03-2024	79.63393%	iPark [04] 30-03-2024	88.92231%
iPark Export [01] 04-05-2024	75.3954%	iPark Export [01] 04-05-2024	85.36852%
iPark Export [02] 06-05-2024	79.27053%	iPark Export [02] 06-05-2024	89.37852%
iPark Export [03] 08-05-2024	79.2570%	iPark Export [03] 08-05-2024	88.61481%

Table 1: Comparing models on their respective training and testing data

Deployment

The deployed application setup can be found here:

`iParkSetup.exe`
`[https://github.com/KushagraYashu/iPark/releases/download/setup/iParkSetup.exe]`

After downloading, the program can be installed by running and following the instructions in the setup.

7 The Conclusion

In conclusion, this research successfully demonstrates the viability and effectiveness of employing Reinforcement Learning (RL) agents within the Unity3D environment to tackle the complex problem of autonomous parking. By integrating the Unity ML-Agents framework, we have shown that virtual agents can be trained to navigate and park vehicles autonomously in a variety of scenarios that closely mimic real-world conditions. The RL approach, particularly within the robust and versatile Unity simulation, proved to be a powerful method for developing adaptive and intelligent parking systems.

Throughout this study, the RL agents displayed significant capabilities in spatial awareness, trajectory planning, and real-time decision-making. The performance metrics,

including success rate and parking accuracy, indicated that the RL agents could consistently and efficiently execute parking maneuvers across different configurations and dynamic environments. These results underscore the potential of RL techniques in advancing autonomous vehicle technologies, particularly in enhancing the functionality and reliability of self-parking systems.

Moreover, this research contributes valuable insights into the broader application of machine learning in simulated environments, offering a blueprint for future studies aiming to train RL agents for complex tasks. The use of Unity3D as a simulation platform not only provided a realistic training ground but also facilitated the exploration of various parking scenarios, thereby enhancing the agent's learning process.

Looking ahead, there are several avenues for further research. Future work could explore the integration of additional sensors and real-world data to improve the realism and robustness of the simulation. Additionally, investigating other RL algorithms and hybrid approaches could further enhance the efficiency and effectiveness of the autonomous parking system. The insights gained from this study pave the way for more sophisticated and adaptable autonomous systems, contributing to the ongoing evolution of intelligent transportation solutions.

In summary, the development and evaluation of the autonomous parking system utilizing RL agents within the Unity3D environment demonstrate a promising step forward in the realm of autonomous vehicle technology. The findings from this research highlight the practical applications of RL and set the stage for future innovations in automated parking and beyond.

References

[BM21] Clara Barbu and Stefan Alexandru Mocanu. On the development of Autonomous Agents using Deep Reinforcement Learning. *Scientific Bulletin, University Politehnica of Bucharest*, Vol 83(Series C):97–115, 2021.

[DB22] Mohamed Fethi Dellali and Mohamed El Mahdi Bouzegzeg. Autonomous Parking Simulation using Unity Game Engine and Reinforcement Learning. *Univ Blida*, 2022.

[Gre93] George D. Greenwade. The Comprehensive Tex Archive Network (CTAN). *TUGBoat*, 14(3):342–351, 1993.

[JBT+20] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. *arXiv:1809.02627v2*, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second Edition. MIT Press, 2018.

[SMMD23] Yusef Savid, Reza Mahmoudi, Rytis Maskeliūnas, and Robertas Damaševičius. Simulated Autonomous Driving using Reinforcement Learning: A Comparative Study on Unity's ML-Agents Framework. *Information*, 14(5):290, 2023.

[Tan22] Omar Tanner. Multi-Agent Car Parking using Reinforcement Learning. *Book of Abstracts, ICUR 2022, ArXiv 2206.13338*, 2022.

[Tec] Unity Technologies. Unity ML-Agents Toolkit.

[Wik] Wikipedia. Unity (game engine).

[Zha] Joy Zhang. A hands-on introduction to deep reinforcement learning using Unity ML-Agents.

Appendix B: CVs

Kushagra .

Game Developer

Undergraduate student with penchant for Video game designing and development. Experience conceptualizing and developing several gaming projects. Exhibits fast-learning capabilities and an initiative driven attitude. Interested in making a difference in the world of gaming.

kushagra102004@gmail.com

kushagrayashu.github.io/_wPortfolio/

linkedin.com/in/kushagrabb

github.com/KushagraYashu

EDUCATION

Undergraduate (B.Tech, CSE AI&ML)
A. P. J Abdul Kalam Technical University

10/2020 - Present *Ghaziabad, Uttar Pradesh*

- **CSE AI&ML, 8.204 CGPA (up-to 7th Sem)**
- **Transcript Link**

High School (CBSE, 12th)
Greenway Modern School

04/2018 - 07/2020 *Dilshad Garden, New Delhi*

- **XII, PCM CS, 90.6%**
- **Transcript Link**

WORK EXPERIENCE

Unity Developer
VYORIUS DRONES PRIVATE LIMITED

07/2023 - 12/2023

- Developed and maintained the Drone Flight Simulator, Drone Test Simulator, and Drone Soccer to support drone pilots training in drone manoeuvrability.
- Operated tools such as Unity Engine, Terrain tools, Custom Input Managers, Physics, and Animations.
- For more details, please refer to the attached Driver Folder.

Reference : Pankaj Kumar (Co-Founder and CTO) - +91 88021 12520
Amulya Gupta (Tech Lead) - +91 96508 24906

Python Content Creator
Alphadynamic

05/2022 - 07/2022

- Devised multiple small games to educate children about Python Language.
- Engineered AI models and AI apps using web services like Nyckel and MIT App Inventor.

PARTICIPATION & ACHIEVEMENTS

Brackeys Game Jam 2023.2
-A Week long jam to create a game using the theme "Diving Deeper". -The Submitted game can be found on the project page or this repository.

Top 90 in IcoJam 2023
-Devise a functional game in 72 hours with the theme of 3.... 2.... 1.... -The submitted game can be found on the project page or this repository.

1-Bit Jam 2023
-Create full game using only 2 colors with theme of Light and Dark. -The submitted game can be found on the project page or this repository.

SKILLS

Unity 3D | C# | C++ | Problem Solving | GitHub | Design Thinking | Social Engineering

PROJECTS

REVV Car Simulation (In Progress)
- **UNITY ENGINE | NAVMESH | NAVIGATION AI | PHYSICS ENGINE**
- REVV is a car game, inspired heavily by the Forza Horizon series, It will include 5 different cars on 5 different tracks and the player has to complete each track with the respective car to complete the game. The car systems are different from each other and tracks are self-sculpted terrains.

Gunner FPS Game
- **UNITY ENGINE | C# | UNITY ANIMATIONS | PARTICLE SYSTEMS**
- GUNNER game is a VALORANT range-inspired indie project to learn Game Development using Unity and sharpen my skills in the said subject. The Game contains several practice options and weapons with decent movement abilities.

Pakdam Pakdai (Multi-user)
- **UNITY ENGINE | MULTI-USER INPUT | MULTI-CAMERA SYSTEM**
- A multi-user game in which 2 players can play at a time using WASD and ARROW keys and the objective of the RED is to catch the other player BLUE within time, and BLUE has to escape or hide for the given time. The players also have their independent cameras for easy play and look.

All Projects
- Link to the project portfolio page.

CERTIFICATES & COURSES

CS50's Introduction to Game Development by Harvard University
- Harvard's Game Development Course

Unity C# Scripting Fundamentals
- Fundamentals of C# Scripting for Unity Engine by Pluralsight.

- All Courses & Certificates
Link to a Google Drive Folder containing all certificates & courses info.

PUBLICATION

Video Games and Brain Development (Oct 2022)
- Published an article in College Magazine on the topic of the Effects of Games on the Human Brain. (Page 33)

Practices of Implementing Agile Procedures & DevOps Methodologies in the development of Web Application (Jan - Jun 2022)
- Co-Author on a paper targeting Agile and DevOps methodologies in the development of Web Applications according to IT standards, published in the GLIMPSE - Journal of Computer Science (Vol 1[1], Jan-Jun 2022, pg 30-32).

ABHINAV SAJJAN

ABOUT ME

Passionate and dedicated fresher aspiring to apply my skills and knowledge in real-world challenges and I consistently embrace obstacles as an opportunity for growth.

7704082544

20abhinavsajjan03@gmail.com

EDUCATION

HIGH SCHOOL
CITY MONTESSORI SCHOOL
2018
91%

INTERMEDIATE
CITY MONTESSORI SCHOOL
2020
75%

B.TECH
AKGEC
2020-current

SKILLS

- 2D designing
- Front-end
- DSA
- Analytics
- Communication
- Project management
- Development

LANGUAGE

- HTML & CSS
- JAVA
- C
- SQL
- PYTHON
- JAVASCRIPT

WORK EXPERIENCE

EZmotion 2021

front-end developer

Done internship for 1 month as a front-end developer

RUBARU 2023

SRM IST CAMPUS ANNUAL FEST

- me and my team made a augmented reality of SRM campus .

CERTIFICATION

- **Python Development**
  UPSKILLS
- **React**
  COMPLAIN

STRENGTH

- Problem Solving
- Leadership
- Teamwork
- Time Management

Himanshu Mishra

Email: himanshumishra0714@gmail.com
Mobile: +91-9026-028226
Linkedin:himanshu-mishra-447225201

Education

- **Ajay Kumar Garg Engineering College** — Ghaziabad, India
  Bachelor of Technology - Computer Science and Engineering(AI and ML); GPA: 7.84 — *Dec 2020 - present*
  Courses: *Operating Systems, Data Structures, Analysis Of Algorithms, Artificial Intelligence, Machine Learning, , Database management System,Computer Networks,Data Warehouse and Data mining,Cloud Computing*

Skills Summary

- **Languages**: C, , C++, Dart,Python SQL, JAVA
- **Frameworks**: Provider, Riverpod, GetX, Flutter Bloc, MobX, Flutter Hooks
- **Tools**: Android Studio, Firebase, VS code, Devtools, MySQL
- **Soft Skills**: Leadership, Event Management, Writing, Public Speaking, Time Management

Projects

- **BMI Calculator App using Flutter**: The BMI Calculator app is a mobile application designed to help users calculate their Body Mass Index (BMI). BMI is a measure of body fat based on an individual's weight and height. The app provides a quick and easy way for users to assess whether their weight is within a healthy range.
- **Flash Chat App using Flutter** : The Flash Chat app is a real-time messaging application that allows users to send and receive text messages instantly. It provides a simple and intuitive interface for users to connect and communicate with others in real-time. The app is designed to be visually appealing and user-friendly, making it easy for users to start conversations and stay connected.
- **TIC-TAC-TOE game using Flutter**: Tic Tac Toe is a very popular paper-pencil game often played in classrooms on the last page of the notebook. In this game, two players mark X or 0 one by one in a 3x3 grid. To win the game, one has to complete a pair of 3 symbols in a line, and that can be a horizontal line, a vertical line, or a diagonal line
- **iParking System(Work in progress)**: Building an intelligent parking system that focusses on training a virtual agent to autonomously navigate and park a car in diverse scenarios, emulating real-world challenges.Addresses aspects like spatial awareness, trajectory planning, and real-time decision-making. Technologies used: Machine Learning, Reinforcement Learning, Unity engine.

COURSE WORK

Hands On Essentials-Data Warehouse-April,2023

Hands On Essentials-Cloud Computing-May,2023

NPTEL Online Certification-Soft Skills

Extra Curricular

- College Cricket Team Captain
- Winners of RANN'24 event of Cricket organised at KIET group of institutions ,Ghaziabad
- Winners of LAKSHYA'24 event of Cricket organised at Noida International University,Greater Noida

Volunteer Experience

- **UDDESHYA NGO** — Ghaziabad, India
  Volunteer at the NGO — *Sep 2021 - Present*
  - **About**: Volunteered collection drives,participated in donation efforts,and initiated 4+awareness campaigns on more than 10 social welfare issues.
  - **Impact**: More than 10+ donation drives organised,4+collection drives for clothes and books.More than 200 people including senior citizens and children benefited.

- **Coordinator** — Ghaziabad, India
  Table Tennis Tournament organised at college — *3-4 February 2024*

SHREYASHI JAISWAL

- Ghaziabad, Uttar Pradesh, India
- +(91) 7007926369
- shreyashi1003@gmail.com

SUMMARY

Proactive B.Tech Computer Science graduate with good programming skills and effective communication. Driven by a passion for innovation, I am eager to contribute to a dynamic tech team and make a meaningful impact in the field of technology.

EDUCATION

Bachelor of Technology in Computer Science and Engineering — **12/2020 - Present**
Ajay Kumar Garg Engineering College, Ghaziabad — CGPA - 7.99
- Specialization in Artificial Intelligence and Machine Learning.

Senior Secondary School — **03/2018 - 05/2019**
Glorious Academy, Varanasi — 88.6 %

Secondary School — **03/2016 - 05/2017**
Casterbridge School, Ballia — 87 %

TECHNICAL SKILLS

- **Programming Languages:** C, C++, Python
- **Development:** HTML, CSS
- **Softwares:** Photoshop, Snapseed, Adobe Lightroom

PROJECTS

iParking System — **09/2023 - Present**
- building an intelligent parking system that focusses on training a virtual agent to autonomously navigate and park a car in diverse scenarios, emulating real-world challenges.
- addresses aspects like spatial awareness, trajectory planning, and real-time decision-making.
- technologies used: Machine Learning, Reinforcement Learning, Unity engine.

Jarvis Voice Assistant — **09/2022 - 11/2022**
- processes voice input and provides output through voice and on-screen text.
- Voice Assistant works in three steps -
  1. Speech Recognition - *Recognition of voice using the microphone of the device and converting it into text.*
  2. Speech Manipulation -*Transforming text input into actionable instructions for the computer.*
  3. Interpreting Commands - *Executing the command according to the specific function it is called for.*
- Stops on commands like “quit”, “stop”, or “exit”.
- technologies used: Python, Speech Recognition, Natural Language Processing(NLP), Web Scraping

Calculator — **10/2021 - 11/2021**
- GUI based calculator
- technologies used: Python, Tkinter.

CERTIFICATES

- ICT Academy Honeywell (Cyber Security) (03/2024)
- IBM SkillsBuild (Artificial Intelligence and Data Science) (02/2024)
- Infosys Springboard (Python) (03/2023)
- Remarkskill Education (Python) (10/2021-11/2021)

POSITION OF RESPONSIBILITY

- **HEAD COORDINATOR** of the *Photography and Media Club*(AKGEC). (07/2021-Present)
  1. Sports Photographer - Proficient in capturing diverse sports events using cameras, tripod and gimbal.
  2. Event Photographer - Specialized in covering events like: conferences, meetings, parties, workshop, etc.
  3. Content Writer - Profound in creating various types of contents for social media posts and articles.
- Created College Reports for three consecutive years(2021, 2022, 2023)and covered more than 30 events.
- Coordinated large events like: “**TABLE TENNIS TOURNAMENT”(2022), Freshers’ Party(2023)** at our College.

ADDITIONAL INFORMATION

- **Languages:** Hindi (native), English (professional working proficiency).
- **Extra-Curricular Activities:** Photography, Content Writing.
- **Interests:** Photography, Videography, Content Writing, Editing, Singing, Sketching, Badminton.