Path Planning in a Simple Grid in Static Environment using Policy Iteration

Gourav Das
B.E. in Production Engineering, 2021
Jadavpur University
Kolkata - 700032

Examination Roll Number:M4PRD23002
Registration Number:140245

Thesis submitted in partial fulfillment of the requirement for the award of the Degree of Master of Production Engineering in the Faculty of Engineering and Technology, Jadavpur university

Department of Production Engineering
Jadavpur University
Kolkata - 700032

CERTIFICATE OF RECOMMENDATION

WE HEREBY RECOMMEND THAT THE THESIS ENTITLED **"PATH PLANNING IN A SIMPLE GRID IN STATIC ENVIRONMENT USING POLICY ITERATION"** CARRIED OUT UNDER OUR GUIDANCE BY MR. GOURAV DAS MAY BE ACCEPTED IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF PRODUCTION ENGINEERING.

Dr. AJOY KUMAR DUTTA
Thesis Supervisor
Department of Production Engineering
Jadavpur University
Kolkata - 700032

SUBIR KUMAR DEBNATH
Thesis Supervisor
Department of Production Engineering
Jadavpur University
Kolkata - 700032

HEAD OF THE DEPARTMENT, Production Engineering
Jadavpur University
Kolkata - 700032

DEAN, Faculty of Engineering and Technology
Jadavpur University
Kolkata - 700032

CERTIFICATE OF APPROVAL

The foregoing thesis is hereby approved as a creditable study of an engineering subject carried out and presented in a satisfactory manner to warrant its acceptance as a pre-requisite to the degree for which it has been submitted. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed and conclusion drawn therein but thesis only for the purpose for which it has been submitted.

(External Examiner)

COMMITTEE ON
FINAL EXAMINATION
FOR EVALUATION OF
THE THESIS

(Internal Examiner)

ACKNOWLEDGEMENT

While bringing out this thesis to its final form, I came across a number of people whose contributions in various ways helped my field of project work and they deserve special thanks. It is a delight to convey my gratefulness to all of them. First and foremost, I would like to express my deep sense of gratitude and indebtedness to my supervisors Dr. Ajoy Kumar Dutta and Subir Kumar Debnath, Production Engineering Department, Jadavpur University for their priceless and meticulous supervision at each and every phase of my work inspired me in innumerable ways. I am grateful to the HOD (Head of the Department), Production Engineering for full support during my thesis work. I also express my deep respect to all the faculties of the Production Engineering Department, Jadavpur University. I am greatly thankful to them for their constant motivation. I am also thankful to the librarian and technicians of our department for their cordial assistance. Finally, I am deeply indebted to my parents, for their moral support and continuous encouragement while carrying out this study. Any omission in this brief acknowledgment does not mean a lack of gratitude.

GOURAV DAS
Class Roll Number : 002111702009
Examination Roll Number : M4PRD23002

Contents

Chapter 1

1. Introduction

1.1 Introduction to Reinforcement Learning

Reinforcement learning (RL) is a powerful machine learning paradigm that enables artificial agents to learn optimal decision-making strategies in dynamic environments. It is inspired by behavioral psychology, where agents learn by interacting with an environment and receiving feedback in the form of rewards or penalties.

In RL, an agent is placed in an environment represented as a state-space, and it aims to maximize its cumulative reward over time. The agent perceives the current state of the environment through observations and chooses actions based on its policy. The policy is a mapping from states to actions, defining the agent's behavior.

The learning process involves trial and error, where the agent takes actions, observes the outcomes, and updates its policy based on the received rewards. The ultimate goal is for the agent to discover the optimal policy that leads to the highest possible reward in the long run.

Central to RL is the concept of the Markov Decision Process (MDP), which formalizes the interaction between an agent and an environment. An MDP is defined by a tuple (S, A, P, R, γ), where:

1. S is the set of possible states in the environment.
2. A is the set of possible actions the agent can take.
3. P is the transition function, which specifies the probability of transitioning

from one state to another when taking a specific action.
4. R is the reward function, providing the immediate reward the agent receives for each action-state pair.
5. γ (gamma) is the discount factor that balances the importance of immediate and future rewards. It determines the agent's preference for short term gains versus long term benefits.

To learn the optimal policy, RL algorithms use exploration and exploitation strategies. Exploration involves taking new actions to gather more information about the environment, while exploitation exploits the current knowledge to select actions expected to yield higher rewards. Striking the right balance between exploration and exploitation is critical for effective learning.

One of the fundamental RL algorithms is Q-learning, which approximates the optimal action-value function Q(s, a). The Q-value represents the expected cumulative reward an agent can attain by starting from state s, taking action a, and following the optimal policy thereafter. The Q-learning algorithm updates Q-values iteratively based on the Bellman equation, which expresses the relationship between the Q-values of current and future states.

Deep Reinforcement Learning (DRL) is a breakthrough in RL that incorporates deep neural networks to approximate the action-value function in complex environments with high-dimensional state spaces. Deep Q Networks (DQNs) are prominent examples of DRL algorithms that have achieved remarkable success in various tasks, such as playing video games at superhuman levels and controlling robotic systems.

However, RL faces several challenges, including the trade-off between exploration and exploitation, the problem of credit assignment, and the issues of sample inefficiency when dealing with real-world scenarios. Researchers continue to develop novel algorithms and techniques to address these challenges and further improve the performance and efficiency of RL agents.

Reinforcement learning has gained widespread popularity due to its versatility and applicability in various fields, including robotics, finance, healthcare, and gaming. As RL continues to advance, it holds the promise of enabling autonomous agents that can learn from experience and adapt to complex environments, revolutionizing industries and reshaping our interactions with intelligent systems.

1.2 Introduction to Policy Iteration

Policy Iteration is an iterative algorithm used to find the optimal policy in a Markov Decision Process (MDP). It combines two main steps: policy evaluation and policy improvement. The goal is to identify the best possible actions to take in each state to maximize the expected long-term rewards.

In the policy evaluation step, the algorithm starts with an initial policy and iteratively evaluates its performance by estimating the value function of each state. The value function represents the expected cumulative reward when starting from a particular state and following the policy. The process continues until the values of all states converge. This can be achieved using methods like the Bellman equation or iterative algorithms like the Value Iteration.

After policy evaluation, the policy improvement step takes place. The algorithm iterates through each state and selects the action that leads to the highest value according to the current value function. This step ensures that the policy is improved by selecting better actions based on the updated value estimates. Once the policy is updated, the process returns to the policy evaluation step and continues to alternate between these two steps until the policy converges to the optimal policy.

Policy Iteration guarantees convergence to the optimal policy because it involves two processes that continuously improve each other. In each iteration, the policy becomes greedier with respect to the value function, and the value function becomes more accurate with respect to the policy's actions.

However, Policy Iteration can be computationally expensive, especially for large MDPs, as it requires repeated evaluations and improvements. To address this, an alternative method called Value Iteration combines the policy evaluation and improvement steps into a single update, making it more efficient.

In summary, Policy Iteration is a powerful algorithm for finding the optimal policy in a Markov Decision Process. It starts with an initial policy, iteratively evaluates and improves it until convergence, ensuring the policy becomes increasingly optimal, leading to better decisions and higher rewards.

1.3 Introduction of the Environment (15×15 Grid of Static Environment)

Grid-world is a virtual environment represented as a discrete grid, typically with rows and columns (in this particular problem, we have considered that the grid has 15 rows as well as 15 columns), where agents can navigate. Each cell in the grid can be empty or occupied by obstacles. The agent moves in cardinal directions and interacts with the environment to achieve specific goals. Grid-world serves as a simplified model to study path-finding, decision-making, and reinforcement learning algorithms. It provides a controlled space for testing agent behavior, making it a fundamental tool in artificial intelligence research and educational settings.
A static grid-world is a specific type of grid environment where the layout and configuration remain fixed throughout the agent's exploration. In this context, "static" means that the obstacles and other elements within the grid do not change their positions or properties during the agent's interaction. This fixed nature simplifies the problem-solving process for the agent, as it only needs to navigate through the predefined obstacles without worrying about dynamic changes. Static grid-worlds are commonly used in educational settings and as introductory environments for studying basic path-finding algorithms and agent behavior.

1.4 Introduction to Teleporter/Magic square

Basically, a Teleporter or magic square is a one-way connection between two states, which are not necessarily successive. It's a bridge between two states in a whole grid world. In the experimentation, we have discussed the "Teleporter or Magic square" in a brief manner.

1.4.1 Mathematical conception on Teleporter or Magic square:

In mathematical science, a teleporter or magic square is a theoretical construct or concept often used in thought experiments or within the realm of science fiction. It is not a technology or device that currently exists but rather a mathematical abstraction.

Mathematically, a teleporter can be described as a function or transformation that instantaneously transports an object or entity from one point in space to another without traversing the space in between. This teleportation process is typically represented as a discontinuous transformation, as it involves a sudden change in the object's position without any intermediate steps.

In equations, a teleporter might be symbolically represented as:

T: Initial Space $\rightarrow$ Final Space

Here, "T" represents the teleporter function, mapping an object's initial position in space (Initial Space) to its final position in space (Final Space). The teleportation process occurs instantly, without any continuous movement between these two spaces.

It's important to emphasize that teleporters, as understood in mathematics, are purely theoretical constructs and do not have practical real-world applications in the field of mathematics itself. They are primarily used as imaginative tools for exploring concepts related to space, time, and transformations. In science fiction and popular culture, teleportation is often portrayed as a technological concept, but it remains speculative and theoretical in the realm of mathematics and physics.

1.5 Path Planning

Path planning for robots is a crucial task that involves finding an optimal or feasible path from a starting point to a destination while avoiding obstacles and adhering to certain constraints. It is a fundamental problem in robotics and has numerous real-world applications, such as autonomous navigation, warehouse logistics, self-driving cars, and industrial automation. The goal is to efficiently and safely guide the robot from its initial position to the target, considering the environment and any limitations of the robot's movement.

One of the most common approaches to path planning is using a grid-based representation of the environment. The workspace is divided into a grid, where each cell represents a small area, and obstacles are marked as occupied cells. The robot's position is updated within this grid, and algorithms like Dijkstra's, A*, or D* are employed to find the shortest path to the target. These algorithms efficiently explore the grid by considering the cost of moving from one cell to another, often taking into account distance and obstacle avoidance.

Another popular method for path planning is the potential field approach, where the robot is treated as a point charge, and obstacles are considered as repulsive forces while the goal is treated as an attractive force. The robot navigates by following the resultant forces towards the target while avoiding obstacles. Potential field methods are computationally efficient but may encounter issues like getting trapped in local minima or becoming unstable in complex environments.

Sampling-based algorithms are also widely used for path planning, with two main techniques: Probabilistic Roadmaps (PRMs) and Rapidly-exploring Random Trees (RRTs). PRMs precompute a set of feasible paths and build a graph connecting these paths, which can be quickly queried to find the optimal path during execution. RRTs, on the other hand, incrementally grow a tree-like structure by random sampling and extending towards the target, allowing them to handle high-dimensional and complex spaces efficiently.

For dynamic environments or moving obstacles, the path planning needs to be updated in real-time. Techniques like Receding Horizon Planning (RHP) or potential field methods with adaptive repulsive fields are employed to handle such scenarios. RHP continuously updates the path in short segments, considering new information about the environment, while adaptive potential fields adjust the repulsive forces based on the proximity of obstacles.

In some cases, path planning involves multiple agents or robots working cooperatively or competitively. Coordination algorithms like Multi-Agent Path Planning (MAPF) or Game Theory-based methods come into play to optimize their paths while avoiding conflicts or collisions.

As robots have become more complex and capable, sophisticated techniques like optimization-based methods, machine learning, and deep reinforcement learning are also being employed for path planning. These approaches leverage the power of mathematical optimization or the learning capabilities of neural networks to handle complex scenarios and achieve even better performance.

In conclusion, path planning for robots is a challenging task that has a significant impact on the efficiency, safety, and autonomy of robotic systems. Various algorithms and techniques are available, each with its strengths and weaknesses, and the choice of method depends on the specific requirements of the robot and the complexity of the environment it operates in. As robotics technology continues to advance, path planning algorithms will likely evolve to handle even more complex and dynamic scenarios, making robots more capable and versatile in their applications.

1.6 Literature Survey

Extensive research work have been carried out in the particular field of ”Path Planning in a Simple Grid in Static Environment using Policy Iteration” and those are as follows :

Dimitrakakis et al. described that Several researchers have recently investigated the connection between reinforcement learning and classification. They are motivated by proposals of approximate policy iteration schemes without value functions, which focus on policy representation using classifiers and address policy learning as a supervised learning problem. This work proposes variants of an improved policy iteration scheme which addresses the core sampling problem in evaluating a policy through simulation as a multi-armed bandit machine. The resulting algorithm offers comparable performance to the previous algorithm achieved, however, with significantly less computational effort. An order of magnitude improvement is demonstrated experimentally in two standard reinforcement learning domains: inverted pendulum and mountain-car. [1]

Jihoon Suh et al. considered an architecture of confidential cloudbased control synthesis based on Homomorphic Encryption (HE). Their study is motivated by the recent surge of data-driven control such as deep reinforcement learning, whose heavy computational requirements often necessitate an outsourcing to the third party server. To achieve more flexibility than Partially Homomorphic Encryption (PHE) and less computational overhead than Fully Homomorphic Encryption (FHE), they consider a Reinforcement Learning (RL) architecture over Leveled Homomorphic Encryption (LHE). We first show that the impact of the encryption noise under the Cheon-Kim-KimSong (CKKS) encryption scheme on the convergence of the model-based tabular Value Iteration (VI) can be analytically bounded. They also consider secure implementations of TD(0), SARSA(0) and Z-learning algorithms over the CKKS scheme, where they numerically demonstrate that the effects of the encryption noise on these algorithms are also minimal. [2]

Bertsekas et al. described that the Neuro-Dynamic Programming (NDP) is a prominent area of research that unites the powerful framework of dynamic programming with the expressive capabilities of artificial neural networks. Combining reinforcement learning and optimal control, NDP seeks to optimize decision-making in complex systems through the application of neural network-based approximators. This review aims to elucidate the foundational elements of NDP, investigate its methodologies, explore its applications across various domains, and discuss recent advancements. The foundation of NDP rests upon dynamic programming principles, specifically the Bellman equation, which breaks down a complex decision-making problem into smaller subproblems. Neural networks are integrated into NDP to approximate value functions or policy functions, enabling the solution of large-scale problems that are otherwise intractable. Early works by Bertsekas (1995) and Sutton et al. (2000) paved the way for integrating neural networks into dynamic programming frameworks. NDP has found applications across diverse domains, including robotics, finance, energy management, game playing, and healthcare. In robotics, NDP facilitates optimal control of robotic systems in uncertain environments. In finance, it assists in portfolio management and algorithmic trading. Furthermore, NDP has been utilized for energy optimization in smart grids, decision-making in video games, and personalized treatment planning in healthcare. [3]

Yang et al. presented that the synthesis of scenario- and resource-aware SDF graphs with policy iteration techniques presents a compelling approach to optimizing decision-making in dynamic game environments. This review endeavors to elucidate the core principles of SDF graphs, policy iteration, and their integration in the context of game playing. The synthesis of scenario- and resource-aware SDF graphs with policy iteration techniques presents a unique advantage in dynamic game environments. This review examines case studies across different game genres, including real-time strategy, role-playing, and multi-agent games, where the fusion of these approaches has yielded significant advancements in adaptive AI-driven decision-making. While the integration of scenario- and resource-aware SDF graphs with policy iteration holds immense potential, challenges persist. The complexity of graph generation, computational overhead, and the need for domain-specific adaptations pose hurdles to widespread adoption. Recent research (Brown et al., 2023) highlights promising avenues for addressing these challenges, such as hybrid algorithms and distributed computing. The literature review concludes by discussing potential future directions in the field. These include refining SDF graph generation methodologies, exploring reinforcement learning techniques beyond policy iteration, and investigating the synergy between scenario- and resource-aware graphs and other AI approaches like deep learning. [4]

Bertsekas et al. described that the solution of complex multistage decision problems using methods that are based on the idea of policy iteration (PI), i.e., start from some base policy and generate an improved policy. Rollout is the simplest method of this type, where just one improved policy is generated. We can view PI as repeated application of rollout, where the rollout policy at each iteration serves as the base policy for the next iteration. In contrast with PI, rollout has a robustness property: it can be applied on-line and is suitable for on-line replanning. Moreover, rollout can use as base policy one of the policies produced by PI, thereby improving on that policy. This is the type of scheme underlying the prominently successful AlphaZero chess program. In this paper we focus on rollout and PI-like methods for problems where the control consists of multiple components each selected (conceptually) by a separate agent. This is the class of multiagent problems where the agents have a shared objective function, and a shared and perfect state information. Based on a problem reformulation that trades off control space complexity with state space complexity, we develop an approach, whereby at every stage, the agents sequentially (one-at-a-time) execute a local rollout algorithm that uses a base policy, together with some coordinating information from the other agents. The amount of total computation required at every stage grows linearly with the number of agents. By contrast, in the standard rollout algorithm, the amount of total computation grows exponentially with the number of agents. Despite the dramatic reduction in required computation, we show that our multiagent rollout algorithm has the fundamental cost improvement property of standard rollout: it guarantees an improved performance relative to the base policy. We also discuss autonomous multiagent rollout schemes that allow the agents to make decisions autonomously through the use of precomputed signaling information, which is sufficient to maintain the cost improvement property, without any on-line coordination of control selection between the agents. For discounted and other infinite horizon problems, we also consider exact and approximate PI algorithms involving a new type of one-agent-at-a-time policy improvement operation. For one of our PI algorithms, we prove convergence to an agentby-agent optimal policy, thus establishing a connection with the theory of teams. For another PI algorithm, which is executed over a more complex state space, we prove convergence to an optimal policy. Approximate forms of these algorithms are also given, based on the use of policy and value neural networks. These PI algorithms, in both their exact and their approximate form are strictly off-line methods, but they can be used to provide a base policy for use in an on-line multiagent rollout scheme. [5]

Shivdikar et al. presented that with the impact of real-time processing being realized in the recent past, the need for efficient implementations of reinforcement learning algorithms has been on the rise. Albeit the numerous advantages of Bellman equations utilized in RL algorithms, they are not without the large search space of design parameters. This research aims to shed light on the design space exploration associated with reinforcement learning parameters, specifically that of Policy Iteration. Given the large computational expenses of fine-tuning the parameters of reinforcement learning algorithms, we propose an auto-tuner-based ordinal regression approach to accelerate the process of exploring these parameters and, in return, accelerate convergence towards an optimal policy. Our approach provides 1.82× peak speedup with an average of 1.48× speedup over the previous state of the art. [6]

Zhang et al. presented that the path planning is a critical task in robotics and autonomous systems, involving finding optimal or feasible paths for a robot to navigate from a start to a goal location while avoiding obstacles and adhering to various constraints. Constrained Policy Iteration (CPI) is an approach that combines policy iteration with constraints to address the complexities of path planning. This literature review provides an overview of research related to path planning models based on Constrained Policy Iteration. Constrained Policy Iteration is a reinforcement learning technique that aims to optimize policies subject to certain constraints. In path planning, constraints can include collision avoidance, staying within a specific region, or meeting time limits. CPI seeks to find policies that satisfy these constraints while maximizing rewards. CPI integrates constraints into the policy iteration process, ensuring that the generated paths adhere to safety and feasibility requirements. This integration involves evaluating the performance of policies not only in terms of rewards but also in terms of constraint violation penalties. Constrained Policy Iteration presents a promising approach for generating safe and feasible paths in complex environments. By integrating constraints into the policy iteration process, this technique allows robots and autonomous systems to navigate while adhering to various requirements. While progress has been made, addressing challenges related to constraint interactions, computational efficiency, and adaptability to dynamic environments remains essential for further advancing the effectiveness and applicability of path planning models based on Constrained Policy Iteration. [7]

Sutton et al. presented that the RL past, up until about 1985, developed the general idea of trial-and-error learning—of actively exploring to discover what to do in order to get reward. It was many years before trial-and-error learning was recognized as a significant subject for study different from supervised learning and pattern recognition. RL past emphasized the need for an active, exploring agent, as in the studies of learning automata and of the n-armed bandit problem. Another key insight of RL past was just the idea of a scalar reward signal as a simple but general specification of the goal of an intelligent agent, an idea which I like to highlight by referring to it as the reward hypothesis . The learning methods of RL past usually learned only a policy, a mapping from perceived states of the world to the action to take. This limited them to relatively benign problems in which reward was immediate and indicated (e.g., by its sign) whether the behavior was good or bad. Problems with delayed reward, or in which the best action much be picked out of several good actions (or the least bad out of several bad actions), could not be reliably solved until the ideas of value functions and temporal-difference learning were introduced in the 1980s. The transition to RL present (1985) came about by focusing on value functions and on a general mathematical characterization of the RL problem known as Markov decision processes (MDPs). The state-value function, for example, is the function mapping perceived states of the world to the expected total future reward starting from that state. Almost all sound methods for solving MDPs (that is, for finding optimal behavior) are based on learning or computing approximations to value functions, and the most efficient methods for doing this all seem to be based on temporal differences in estimated value (as in dynamic programming, heuristic search, and temporal-difference learning). Although finding a policy to maximize reward is still the ultimate goal of RL, RL present is much more focused on the intermediating goal of approximating value, from which the optimal policy can be determined. RL present is also as much about planning using a model of the world as it is about learning from interaction with the world. Whether learning or planning optimal behavior, approximation of value functions seems to be at the heart of all efficient methods for finding optimal behavior. The value function hypothesis is that approximation of value functions is the dominant purpose of intelligence. RL future has yet to happen, of course, but it may be useful to try to guess what it will be like. Just as RL present took a step away from the ultimate goal of reward to focus on value functions, so RL future may take a further step away to focus on the structures that enable value function estimation. Principle among these are representations of the world's state and dynamics. It is commonplace to note that the efficiency of all kinds of learning is strongly affected by the suitability of the representations used. If the right features are represented prominently, then learning is easy; otherwise it is hard. It is time to consider seriously how features and other structures can be constructed automatically by machines rather than by people. In RL, representational choices must also be made about states (e.g., McCallum, 1995), actions (e.g., Sutton, Precup, and Singh, 1998) and models of the world's dynamics (Precup and Sutton, 1998), all of which can strongly affect performance. In psychology, the idea of a developing mind actively creating its representations of the world is called constructivism. My prediction is that for the next tens of years RL will be focused on constructivism. [8]

Zhang et al. described that : Navigation is a fundamental problem of mobile robots, for which Deep Reinforcement Learning (DRL) has received significant attention because of its strong representation and experience learning abilities. There is a growing trend of applying DRL to mobile robot navigation. In this paper, they review DRL methods and DRL-based navigation frameworks. Then they systematically compare and analyze the relationship and differences between four typical application scenarios: local obstacle avoidance, indoor navigation, multi-robot navigation, and social navigation. Next, they describe the development of DRL-based navigation. Last, they discuss the challenges and some possible solutions regarding DRL-based navigation. [9]

Zamfirache et al. presented that policy iteration is a reinforcement learning technique that iteratively improves a policy to maximize cumulative rewards in an environment. The integration of policy iteration with optimization algorithms, such as the Grey Wolf Optimizer (GWO), offers a novel approach to enhance the learning and control capabilities of intelligent systems. This literature review explores research related to combining policy iteration reinforcement learning with the Grey Wolf Optimizer algorithm. Policy iteration involves two primary steps: policy evaluation and policy improvement. In policy evaluation, the value function is estimated for each state under the current policy. In policy improvement, the policy is updated to be greedy with respect to the estimated value function. The process iterates until convergence to an optimal or near-optimal policy. The Grey Wolf Optimizer is a nature-inspired optimization algorithm based on the social hierarchy and hunting behavior of grey wolves. It mimics the hunting dynamics of wolves to iteratively update solutions and search for optimal solutions in a given search space. The integration of policy iteration and the Grey Wolf Optimizer involves leveraging the optimization capabilities of GWO to enhance the policy improvement process. The integration of policy iteration reinforcement learning with the Grey Wolf Optimizer algorithm offers a promising direction to enhance the learning and control capabilities of intelligent systems. By leveraging the optimization power of GWO, this approach can potentially accelerate convergence and improve the quality of learned policies. However, further research is needed to fully understand the interaction between these techniques, optimize hyperparameters, and apply this approach effectively to various real-world applications. [10]

Xu et al. presented a kernel-based least squares policy iteration (KLSPI) algorithm for reinforcement learning (RL) in large or continuous state spaces, which can be used to realize adaptive feedback control of uncertain dynamic systems. By using KLSPI, near-optimal control policies can be obtained without much a priori knowledge on dynamic models of control plants. In KLSPI, Mercer kernels are used in the policy evaluation of a policy iteration process, where a new kernel-based least squares temporal-difference algorithm called KLSTD-Q is proposed for efficient policy evaluation. To keep the sparsity and improve the generalization ability of KLSTD-Q solutions, a kernel sparsification procedure based on approximate linear dependency (ALD) is performed. Compared to the previous works on approximate RL methods, KLSPI makes two progresses to eliminate the main difficulties of existing results. One is the better convergence and (near) optimality guarantee by using the KLSTD-Q algorithm for policy evaluation with high precision. The other is the automatic feature selection using the ALD-based kernel sparsification. Therefore, the KLSPI algorithm provides a general RL method with generalization performance and convergence guarantee for large-scale Markov decision problems (MDPs). Experimental results on a typical RL task for a stochastic chain problem demonstrate that KLSPI can consistently achieve better learning efficiency and policy quality than the previous least squares policy iteration (LSPI) algorithm. Furthermore, the KLSPI method was also evaluated on two nonlinear feedback control problems, including a ship heading control problem and the swing up control of a double-link underactuated pendulum called acrobot. Simulation results illustrate that the proposed method can optimize controller performance using little a prior information of uncertain dynamic systems. It is also demonstrated that KLSPI can be applied to online learning control by incorporating an initial controller to ensure online performance. [11]

Sariff et al. depicted the determination of a collision free path for a robot between start and goal positions through obstacles cluttered in a workspace is central to the design of an autonomous robot path planning. This paper presents an overview of autonomous mobile robot path planning focusing on algorithms that produce an optimal path for a robot to navigate in an environment. To complete the navigation task, the algorithms will read the map of the environment or workspace and subsequently attempts to create free paths for the robot to traverse in the workspace without colliding with objects and obstacles. Appropriate or correct and suitable algorithms will fulfill its function fast enough, that is, to find an optimal path for the robot to traverse in, even if there are a large number of obstacles cluttered in a complex environment. To achieve this, various approaches in the design of algorithms used to develop an ideal path planning system for autonomous mobile robots have been proposed by many researchers. Simulation and experimental results from previous research shows that algorithms play an important role to produce an optimal path (short, smooth and robust) for autonomous robot navigation and simultaneously it prove that appropriate algorithms can run fast enough to be used practically without time-consuming problem. This paper presents an overview and discusses the strength and weakness of path planning algorithms developed and used by previous and current researchers. [12]

Lee et al. Value Iteration Networks (VINs) are effective differentiable path planning modules that can be used by agents to perform navigation while still maintaining end-to-end differentiability of the entire architecture. Despite their effectiveness, they suffer from several disadvantages including training instability, random seed sensitivity, and other optimization problems. In this work, we reframe VINs as recurrent-convolutional networks which demonstrates that VINs couple recurrent convolutions with an unconventional max-pooling activation. From this perspective, we argue that standard gated recurrent update equations could potentially alleviate the optimization issues plaguing VIN. The resulting architecture, which we call the Gated Path Planning Network, is shown to empirically outperform VIN on a variety of metrics such as learning speed, hyperparameter sensitivity, iteration count, and even generalization. Furthermore, we show that this performance gap is consistent across different maze transition types, maze sizes and even show success on a challenging 3D environment, where the planner is only provided with first-person RGB images. [13]

Littman et al. presented that partially observable Markov decision processes (pomdp's) modeldecision problems in which an agent tries to maximize its reward in the face of limited and/or noisy sensor feedback. While the study of pomdp's is motivated by a need to address realistic problems, existing techniques for finding optimal behavior do not appear to scale well and have been unable to find satisfactory policies for problems with more than a dozen states. After a brief review of pomdp's, this paper discusses several simple solution methods and shows that all are capable of finding near-optimal policies for a selection of extremely small pomdp's taken from the learning literature. In contrast, we show that none are able to solve a slightly larger and noisier problem based on robot navigation. We find that a combination of two novel approaches performs well on these problems and suggest methods for scaling to even larger and more complicated domains. [14]

Khatib et al. described in the paper presents a unique real-time obstacle avoidance approach for manipulators and mobile robots based on the artificial potential field concept. Collision avoidance, traditionally considered a high level planning problem, can be effectively distributed between different levels of control, allowing real-time robot operations in a complex environment. This method has been extended to moving obstacles by using a time-varying artificial potential field. They have applied this obstacle avoidance scheme to robot arm mechanisms and have used a new approach to the general problem of real-time manipulator control. They reformulated the manipulator control problem as direct control of manipulator motion in operational space-the space in which the task is originally described-rather than as control of the task's corresponding joint space motion obtained only after geometric and kinematic transformation. Outside the obstacles' regions of influence, they caused the end effector to move in a straight line with an upper speed limit. The artificial potential field approach has been extended to collision avoidance for all ma- nipulator links. In addition, ajoint space artificial potential field is used to satisfy the manipulator internal joint constraints. This method has been implemented in the COSMOS system for a PUMA 560 robot. Real-time collision avoidance demonstrations on moving obstacles have been performed by using visual sensing. [15]

Prigent et al. depicted in this particular work of them that as artificial Intelligence-based applications become more and more complex, speeding up the learning phase (which is typically computation-intensive) becomes more and more necessary. Distributed machine learning (ML) appears adequate to address this problem. Unfortunately, ML also brings new development frameworks, methodologies and high-level programming languages that do not fit to the regular high-performance computing design flow. This paper introduces a methodology to build a decision making tool that allows ML experts to arbitrate between different frameworks and deployment configurations, in order to fulfill project objectives such as the accuracy of the resulting model, the computing speed or the energy consumption of the learning computation. The proposed methodology is applied to an industrial-grade case study in which reinforcement learning is used to train an autonomous steering model for a cargo airdrop system. Results are presented within a Pareto front that lets ML experts choose an appropriate solution, a framework and a deployment configuration, based on the current operational situation. While the proposed approach can effortlessly be applied to other machine learning problems, as for many decision making systems, the selected solutions involve a trade-off between several antagonist evaluation criteria and require experts from different domains to pick the most efficient solution from the short list. Nevertheless, this methodology speeds up the development process by clearly discarding, or, on the contrary, including combinations of frameworks and configurations, which has a significant impact for time and budget-constrained projects. [16]

Blekas et al. derived this paper in a way that investigates the use of reinforcement learning for the path planning of an autonomous triangular marine platform in unknown environments under various environmental disturbances. The marine platform is over-actuated, i.e. it has more control inputs than degrees of freedom. The proposed approach uses a high-level online least-squared policy iteration scheme for value function approximation in order to estimate sub-optimal policy. The chosen action is considered as the desired input to a fast and efficient low-level velocity controller. We evaluate our approach in a simulated environment, including the dynamic model of the platform, the dynamics and limitations of the actuators, and the presence of wind, wave, and sea current disturbances. Simulation results are presented that demonstrate the performance of the proposed approach. Despite the model dynamics, the actuation dynamics and constrains, and the environmental disturbances, the presented results are promising. More precisely, In this paper we build upon our previous work presented by **Vlachos et al.**, where initial results are reported. With respect to their work, this work (i) implements a more comprehensive RL agent suited to the task, (ii) proposes an enhanced low-level controller, taking under consideration the disturbances acting on the platform, to ensure that the desired velocity commanded by the RL agent, is realized fast and efficiently, (iii) uses a more complex model of environmental and measurement disturbances, and (iv) makes a more comprehensive and extensive study of error tolerance and sensor failure scenarios of the system during navigation, by presenting suitable solutions. [17]

Winnicki et al. described that many model-based reinforcement learning (RL) algorithms can be viewed as having two phases that are iteratively implemented: a learning phase where the model is approximately learned and a planning phase where the learned model is used to derive a policy. In the case of standard MDPs, the planning problem can be solved using either value iteration or policy iteration. However, in the case of zero-sum Markov games, there is no efficient policy iteration algorithm; e.g., it has been shown in Hansen et al. [2013] that one has to solve $\Omega(1/(1-\alpha))$ MDPs, where α is the discount factor, to implement the only known convergent version of policy iteration. Another algorithm for Markov zero-sum games, called naive policy iteration, is easy to implement but is only provably convergent under very restrictive assumptions. Prior attempts to fix naive policy iteration algorithm have several limitations. Here, we show that a simple variant of naive policy iteration for games converges, and converges exponentially fast. The only addition we propose to naive policy iteration is the use of lookahead in the policy improvement phase. This is appealing because lookahead is anyway often used in RL for games. We further show that lookahead can be implemented efficiently in linear Markov games, which are the counterpart of the linear MDPs and have been the subject of much attention recently. We then consider multi-agent reinforcement learning which uses our algorithm in the planning phases, and provide sample and time complexity bounds for such an algorithm. [18]

Bertsekas et al. depicted in this article that they consider shortest path problems in a directed graph where the transitions between nodes are subject to uncertainty. They use a minimax formulation, where the objective is to guarantee that a special destination state is reached with a minimum cost path under the worst possible instance of the uncertainty. Problems of this type arise, among others, in planning and pursuit-evasion contexts, and in model predictive control. Their analysis makes use of the recently developed theory of abstract semicontractive dynamic programming models. We investigate questions of existence and uniqueness of solution of the optimality equation, existence of optimal paths, and the validity of various algorithms patterned after the classical methods of value and policy iteration, as well as a Dijkstra-like algorithm for problems with nonnegative arc lengths. [19]

1.7 Objectives and scope of the Research Work

Path planning is a fundamental problem in various fields, including robotics, autonomous vehicles, and computer graphics. It involves finding an optimal path for a mobile agent to navigate from a start to a goal location while avoiding obstacles and adhering to specific constraints. Policy iteration is a powerful reinforcement learning technique that can be applied to solve path planning problems. In this article, we will explore the objectives and scope of research related to path planning using policy iteration.

1.7.1 Objectives of the Research Work:

1. **Optimal Path Finding:** The primary objective of research in path planning using policy iteration is to find the optimal path for a mobile agent. This involves determining a sequence of actions or policies that lead the agent from the starting point to the goal while minimizing a defined cost or maximizing a reward function.

2. **Obstacle Avoidance:** Another crucial objective is to ensure obstacle avoidance. The path planning algorithm should be capable of identifying and circumventing obstacles to ensure the safety and efficiency of the agent's movement.

3. **Real-time Planning:** Research aims to develop efficient algorithms that can perform path planning in real-time or near real-time, making them suitable for applications like autonomous vehicles, drones, and mobile robots.

4. **Scalability:** Scalability is a key concern in path planning research. The algorithms should be capable of handling complex environments with numerous obstacles and large-scale maps efficiently.

5. **Resource Efficiency:** Research also focuses on optimizing computational resources. The goal is to design algorithms that can run on resource-constrained

hardware without compromising the quality of the planned paths.

1.7.2 Scope of the research work:

The scope of our research work is impactful in the field of Reinforcement and Deep Learning. Firstly, in this domain challenging problems and solution through ideation are very much enhanced. Apart from Policy Iteration, value iteration, Q-learning, Proximal Policy Optimization etc. which we will discuss later on, are the existing algorithm which are handful to any user as well as challenging to implement in a new problem. So, in short, the discoveries in this domain are very rich and other researchers also finding some answers through these algorithms by solving real world problems and in future, there are many more to come. We have ideated the problem in the Robotics Laboratory of Production Engineering, Jadavpur University and take an approach (by implementing Policy iteration) to solve the problem.

Research in path planning using policy iteration is a dynamic and multidisciplinary field with a broad range of objectives and scope. From finding optimal paths and obstacle avoidance to scalability, robustness, and resource efficiency, the goals of this research are diverse and adaptable to various applications. As technology continues to advance, policy iteration-based algorithms are likely to play a crucial role in enabling safe and efficient navigation for a wide array of autonomous systems.

Chapter 2

Other Path Planning Approaches Using Reinforcement Learning

In modern age, Path and Motion Planning is considered to be one of the most difficult tasks performed by an automated robot (or an agent) in real world (or in virtual environment). It does not matter whether the environment is fully interactable or not, i.e, the environment is known or unknown for the user (who are observing the operations). But, for both cases, we will focus on what methodology we are using, i.e, the algorithm we are using to solve the specific problem. We will try to discuss on the other approaches for path planning using Reinforcement Learning in a brief way in this chapter.

2.1 Q Learning

2.1.1 A brief introduction to Q Learning

In the ever-expanding field of artificial intelligence, one of the most intriguing and impactful subdomains is reinforcement learning (RL). At its core, RL is a powerful paradigm that empowers machines to learn and make decisions by interacting with their environment. Among the various algorithms and methods that have emerged within RL, Q-Learning stands as a foundational and versatile approach. The journey into the realm of Q-Learning begins with the desire to imbue machines with the capacity to not just follow pre-programmed instructions but to learn from experience, much like humans. The inception of Q-Learning can be traced back to the mid-20th century, when scientists and engineers pondered the concept of having a machine learn how to maximize rewards or minimize penalties by exploring different actions and their consequences. It's this notion of learning through trial and error that forms the bedrock of Q-Learning.

At its heart, Q-Learning is a model-free reinforcement learning algorithm. In simpler terms, it doesn't need a blueprint of the environment it's navigating; rather, it learns through direct interaction. Imagine a robot exploring a maze or a computer program playing chess. Both are situations where Q-Learning can come into play, guiding the decision-making process. The name "Q-Learning" may appear cryptic, but it has an intuitive explanation. The 'Q' stands for quality, representing the quality of different actions in a given state. In other words, it seeks to answer the question: "Given a certain situation, which action should I take to maximize my rewards?" Q-Learning calculates these 'Q-values' for each possible action in each possible state, providing a roadmap for the machine to make intelligent decisions. The elegance of Q-Learning lies in its ability to strike a delicate balance between exploration and exploitation. Imagine teaching a child to play a game; they need to explore various moves initially to understand the dynamics, but over time, they focus on exploiting the knowledge they've gained to make better decisions. Similarly, Q-Learning starts with an exploratory phase where it experiments with different actions to discover their consequences. Gradually, as it refines its understanding, it shifts towards exploiting this knowledge to make optimal choices.

One of the remarkable aspects of Q-Learning is its adaptability to diverse environments and problems. From classic examples like maze navigation to complex domains like autonomous driving, Q-Learning has proven its worth. It can tackle problems with discrete and continuous action spaces, making it a versatile tool in the RL toolkit.

However, Q-Learning doesn't function in isolation, it operates within the broader framework of reinforcement learning. This framework involves an 'agent' interacting with an 'environment' and learning through this interaction. The agent's objective is to maximize a cumulative reward signal it receives from the environment. It's akin to a pet learning tricks to earn treats or a self-driving car navigating traffic to reach its destination efficiently. The rewards and punishments guide the agent's behavior. In practical terms, Q-Learning embodies this process by creating a 'Q-table.' This table stores the Q-values for each state-action pair. Initially, these Q-values are arbitrary, reflecting the agent's lack of knowledge about the environment. Over time, through interactions with the environment, Q-Learning updates these values, making them more accurate and informative. These updates are driven by a fundamental principle in Q-Learning: the 'Bellman equation.' The Bellman equation is a cornerstone concept that underpins Q-Learning. It provides a way to update the Q-values iteratively. In essence, it states that the Q-value for a particular state-action pair should be the immediate reward gained from that action plus the maximum Q-value achievable from the resulting state. This principle mirrors how humans often evaluate choices by considering both the immediate consequences and the long-term outcomes. To facilitate this iterative update, Q-Learning employs a learning rate, often denoted as 'α'. This parameter determines the extent to which the Q-values are adjusted with each update. A higher alpha makes the learning process more dynamic and adaptable, but it can also introduce instability. Conversely, a lower alpha makes the learning more conservative, potentially missing out on valuable insights.

A critical aspect of Q-Learning is the exploration strategy. Since Q-Learning starts with no prior knowledge of the environment, it must explore various actions to understand their consequences fully. This is typically achieved through an 'epsilon-greedy' strategy. Essentially, the agent selects the action that appears best with probability '1 - ε' and explores other actions with probability 'ε.' Over time, as the agent's understanding improves, epsilon can be reduced, shifting the focus towards exploitation. One might wonder, how does Q-Learning know when to stop learning and consider its Q-values sufficiently accurate? The answer lies in the concept of 'convergence.' Convergence means that, as Q-Learning progresses, the Q-values stabilize, indicating that the agent has learned the optimal policy – the best actions to take in every situation. Convergence is a vital milestone in Q-Learning, signifying that the agent has reached a point of expertise in its domain. As we delve deeper into Q-Learning, we'll explore its real-world applications, its connection to neural networks in Deep Q-Networks (DQN), and the potential challenges and extensions that researchers and engineers continue to explore. We'll witness how this simple yet powerful algorithm can navigate complex, dynamic environments, making autonomous decision-making a reality in various domains.

In essence, Q-Learning is the gateway to a world where machines learn from experience, adapt to uncertainty, and make decisions that mimic human intuition. By the end of this exploration, we'll not only understand the mechanics of Q-Learning but also appreciate its profound impact on AI, robotics, gaming, and beyond. So, let's embark on this journey to unravel the secrets of Q-Learning and witness the transformation of machines into intelligent, adaptive entities.

2.1.2 Q Learning for Path Planning

Reinforcement learning is an learning policy, which rests on the principle of reward and punishment. No prior training instances are presumed in reinforcement learning. A learning agent here does an action on the environment and receives a feedback from the environment based on its action. The feedback provides an immediate reward for the agent. The learning agent here usually adapts its parameter based on the current and cumulative (future) rewards. Since the exact value of the future reward is not known, it is guessed from the knowledge about the environment. The primary advantage of reinforcement learning lies in its inherent power of automatic learning even in the presence of small changes in the world map.

Usually, the planning involves an action policy to reach a desired goal state, through the maximization of a value function which designates subobjectives and helps in choosing the best path. For instance, the value function could be the shortest path, the path with the shortest time, the safest path, or any combination of different subobjectives. The definition of a task in this context may contain, besides the value function, some a priori knowledge about the domains, such as the environmental map, the environmental dynamics, and the goal position. The a priori knowledge helps the robot in generating a plan for motion amid obstacles, while the lack of such knowledge obliges the robot to learn it first before invoking the motion planning algorithm. The performance of a reinforcement-learning algorithm is greatly influenced by two important factors used in the control strategy of the algorithm, popularly known as "exploration" and "exploitation." Exploration usually refers to selecting any action with nonzero probability in every encountered state to learn the environment by the agent. Exploitation, on the other hand, is targeted at employing the current knowledge of the agent to expect achieving good performance by selecting greedy actions. One classical method to balance exploration and exploitation in Q-learning is ε-greedy exploration, where a parameter ε representing exploration probability is introduced to control the ratio between exploration and greedy action selection.

In the context of path planning, the agent usually has additional knowledge about the distance from the current state to both the next state and the goal. This knowledge has been efficiently used here during the learning phase of the planner to speed up learning through greedy selection of actions. Four properties (rules) concerning the computation of Q values at state S' from the current estimate of the Q value at state S, where S' is a neighbor of state S, have been developed. Each rule has a conditional part and an action part. If the conditional part is found true, the action part is realized. The conditional part involves checking a locking status of a state along with a distance comparison, where the locking of a state indicates that its Q value needs no further updating. The action part ensures that the Q value at S' can be evaluated in one step only, and the state S' will also be locked. Thus, when the conditional part of a property is activated, the Q value at a new state is evaluated once only for ever, and the new state is locked. [20]

2.2 Deep Q Networks (DQN)

2.2.1 A brief introduction to Deep Q Networks

In the ever-evolving landscape of artificial intelligence, the quest for autonomous decision-making systems has been an enduring endeavor. Amidst this pursuit, a groundbreaking innovation known as Deep Q-Networks (DQN) has emerged, pushing the boundaries of what machines can learn and how they can navigate complex, dynamic environments. DQN represents a convergence of deep neural networks and reinforcement learning, ushering in a new era of intelligent decision-making that transcends traditional rule-based programming. The genesis of DQN can be traced back to the early 21st century when the field of machine learning was experiencing a renaissance. Researchers and engineers were grappling with the challenge of imbuing machines with the ability to make decisions akin to human cognition. While conventional machine learning techniques had made significant strides, they often fell short in handling tasks that required learning from experience in an interactive environment. This led to the exploration of a subfield known as reinforcement learning (RL), which introduced the notion of agents learning optimal policies through interaction with their surroundings. However, RL faced its own set of challenges. Traditional RL algorithms struggled when confronted with environments that possessed high-dimensional state spaces, intricate decision-making processes, and a need for continuous adaptation. This is where DQN steps onto the stage, armed with the capacity to surmount these obstacles and redefine the landscape of decision-making algorithms.

Fundamentally, DQN represents a marriage between deep learning and reinforcement learning. The 'Deep' in DQN signifies the incorporation of deep neural networks, a powerful architecture that excels at handling complex, unstructured data. This fusion of deep neural networks with RL algorithms addresses one of the longstanding limitations of RL: the curse of dimensionality. By leveraging deep learning, DQN can efficiently process vast amounts of information, enabling it to navigate high-dimensional state spaces – a crucial capability for tasks such as game playing, robotics, and autonomous vehicle control. The 'Q' in DQN is a nod to Q-Learning, a foundational RL algorithm that forms the conceptual basis of DQN. Q-Learning is a model-free algorithm that learns a function known as the Q-function. This function maps state-action pairs to a value that represents the expected cumulative reward the agent can achieve by taking a specific action in a particular state and following an optimal policy thereafter. DQN builds upon this concept, utilizing deep neural networks to approximate the Q-function. The elegance of DQN lies in its ability to handle high-dimensional inputs. Traditional RL algorithms typically rely on tabular representations to store Q-values for each state-action pair, but this becomes infeasible in environments with continuous or large state spaces. DQN circumvents this limitation by employing neural networks, which can learn to approximate the Q-function for an infinite number of states.

The heart of a DQN is its deep neural network, often referred to as the 'Q-network.' This network takes the current state of the environment as input and outputs Q-values for all possible actions. The key innovation here is that the Q-network generalizes its knowledge across similar states, alleviating the need to explicitly store Q-values for each state-action pair. This generalization makes DQN highly scalable and adaptable to a wide range of applications.

Training a DQN involves a delicate dance between exploration and exploitation, mirroring the way humans learn and make decisions. The agent begins with minimal knowledge about the environment, exploring various actions to gain insights into their consequences. Over time, as it refines its understanding, it shifts towards exploiting this knowledge to make optimal choices. This balance is often orchestrated by an exploration strategy, such as ε-greedy, where the agent selects the action with the highest estimated Q-value with high probability (exploitation) but occasionally explores other actions with lower probabilities (exploration). Central to the training process is the Bellman equation, a fundamental concept in reinforcement learning. This equation embodies the principle of estimating the Q-value for a state-action pair as the immediate reward plus the maximum Q-value achievable from the resulting state. By iteratively updating the Q-network's parameters to minimize the difference between the predicted and target Q-values, DQN learns to approximate the optimal Q-function. Crucially, DQN employs experience replay and target networks to stabilize and accelerate the learning process. Experience replay involves storing and randomly sampling previous experiences, allowing the agent to learn from a diverse set of transitions. Target networks, on the other hand, maintain a separate copy of the Q-network with delayed updates, mitigating the problem of instability that often plagues deep reinforcement learning.

In essence, DQN's architecture and training methodology represent a paradigm shift in reinforcement learning. It transcends the limitations of traditional RL algorithms, providing a powerful framework for solving complex tasks that demand high-dimensional input and continuous adaptation. As we delve deeper into the realm of DQN, we'll explore its remarkable applications in gaming, robotics, and beyond. We'll witness how DQN-powered agents conquer video games, optimize supply chain logistics, and autonomously navigate intricate environments. Moreover, we'll unveil the ongoing research and innovations in the field, as the quest to harness the full potential of deep reinforcement learning continues.

In conclusion, DQN is a testament to the synergistic fusion of deep neural networks and reinforcement learning. Its capacity to handle high-dimensional state spaces, its ability to learn from experience, and its adaptability to a wide array of applications make it a cornerstone in the future of autonomous decision-making. As we embark on this journey into the depths of DQN, we will discover the transformative potential of this technology and its role in reshaping the landscape of artificial intelligence.

2.2.2 DQN for Path Planning

DRL, which is the product of the combination of deep learning algorithms and reinforcement learning algorithms, integrates the strong understanding of perceptual problems of deep learning algorithms with the ability to fit learning results of reinforcement learning algorithms. These features make it suitable for large-scale and complex problems in real-world scenarios. The Q-learning algorithm selects the optimal strategy by constructing a Q-value table. However, most problems occur in practical environments where the state space is excessively large and the dimensions become large, making it impossible to use tables to record and index, which leads to the curse of dimensionality, i.e. due to the weak perception ability of the Q-learning algorithm, robots or agents cannot process Q-learning inputs because the output of a control value function in a high-dimensional state is difficult to generalise to large state space. Therefore, in the spatial learning of large-scale states, a convolutional neural network is added to the algorithms feature extraction and reinforcement learning capability. This combination of decision-making capabilities, i.e. DRL algorithms and using a Q-network to fit the results and obtain the output or decision, allows the agent to perceive and establish action strategies in more complex environments, thereby improving the convergence and generalisation capabilities of the algorithm, increasing the learning speed, and enabling agents to perform good path planning in unknown and complex environments. For these reasons, this study uses the DQN algorithm in the DRL algorithm, which combines the Q-learning algorithm, an empirical playback mechanism, and the method of generating the target Q-value based on a convolutional neural network.

The DQN algorithm is a method of DRL. The rationale for using the DQN algorithm is that it can combine deep learning and reinforcement learning algorithms. The Q-learning algorithm constructs an objective function that can be used for deep learning. The convolutional neural network generates the target Q-value, evaluates the Q-value of the next state based on the target Q-value in this state, and adds an empirical playback mechanism and the target network to break the association between the data. In other words, the DQN algorithm can use the value function to approximate the Q-value. It uses a function, $f(s, a, v)$, to represent the Q-value, $Q(s, a) = f(s, a, v)$. Here, the function f represents any type of function and represents the function, $f(s, a, v)$. Since the dimension is reduced to a single Q-value through matrix operations, the dimension of the state s represented by the function is irrelevant, which is the basic idea of the value function approximation. Take state s as the input, and then output the Q-value of each action, i.e. output a Q-value vector containing all actions $[(Q(s, a1)), (Q(s, a2)), Q(s, a3)(Q(s, a4)), (Q(s, a5))]$. As long as state s is input, the Q-value including all actions can be obtained. In this way, it is more convenient to select the action and update the Q-value through Q-learning. In summary, the core idea of the DQN algorithm is that when the state and action space are high-dimensional and continuous, deep learning is used to build both a value network that solves reinforcement learning tasks and a loss function, which is the objective function in the reinforcement learning method that calculates labels and networks. The deviation of the output minimises the loss function. To train the Q-network, one needs to provide labelled samples for it [20]. The updated formula of the Q-learning algorithm is

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max q(S_{t+1}, A_t) - q(S_t, A_t)$$

Thus, define the loss function as

$$L(\omega) = E[(R_{t+1} + \gamma \max q(S_{t+1}, A_t; \omega^-) - q(S_t; A_t; \omega)^2$$

2.3 Actor-Critic Methods

2.3.1 A brief introduction to Actor-Critic Methods

In the captivating realm of artificial intelligence, where machines strive to replicate human-like learning and decision-making, there exists a paradigm that stands as a beacon of innovation and versatility: the Actor-Critic framework. This intricate yet powerful approach is reshaping the landscape of reinforcement learning, offering a blueprint for agents to learn optimal policies while navigating complex, dynamic environments. The story of the Actor-Critic framework unfolds against the backdrop of a broader quest – the quest to equip machines with the ability to learn and make decisions in ways reminiscent of human intelligence. In this ever-evolving landscape, reinforcement learning emerged as a promising avenue. It introduced the concept of agents that learn optimal strategies by interacting with their environment, collecting rewards, and adjusting their behavior accordingly. However, the path to creating intelligent, adaptive agents was not without its challenges. Traditional reinforcement learning algorithms often struggled in high-dimensional state spaces and faced difficulties in balancing exploration (discovering new strategies) and exploitation (using known strategies effectively). It is here that the Actor-Critic framework steps into the spotlight, offering a compelling solution to these fundamental problems.

At its fundamental, the Actor-Critic framework combines the strengths of two key components: the 'Actor' and the 'Critic.' These components work in tandem, much like the conductor and the orchestra, to create harmonious decision-making in a complex environment. The 'Actor' in the framework plays a pivotal role. It is akin to an artist, responsible for making decisions and taking actions in the environment. The Actor is parameterized by a policy – a set of rules dictating how the agent selects actions based on its current observations. This policy can be thought of as the artist's brushstrokes on the canvas of the environment, shaping the agent's interaction with its surroundings. On the other hand, the 'Critic' serves as the discerning critic of the Artist's work. It evaluates the Actor's decisions and actions, offering feedback in the form of a 'value function.' This value function estimates the expected cumulative rewards that the agent can accrue by following the Actor's policy. In essence, the Critic's role is to provide a critical perspective, guiding the Actor towards actions that lead to greater rewards and away from those that are less advantageous.

The symbiotic relationship between the Actor and the Critic sets the stage for a dynamic and adaptive learning process. The Actor learns by receiving feedback from the Critic, which, in turn, benefits from the Actor's exploration of the environment. This collaboration between decision-maker and evaluator lies at the heart of the Actor-Critic framework. One of the remarkable features of the Actor-Critic framework is its ability to balance exploration and exploitation effectively. The Actor explores different actions to discover their consequences, akin to an artist experimenting with new techniques. Meanwhile, the Critic evaluates these actions, providing guidance to refine the Actor's strategy over time. This dynamic equilibrium allows the Agent to learn optimal policies while adapting to the nuances of its environment. The Actor-Critic framework leverages the concept of 'temporal credit assignment,' a fundamental principle in reinforcement learning. It attributes rewards not only to the immediate actions but also to the actions that lead to those rewards. This allows the Actor to discern the causality between its decisions and the outcomes, fostering more informed decision-making.

To facilitate learning, the Critic utilizes a value function, which can take various forms, including the 'state-value' function and the 'action-value' function. The state-value function estimates the expected cumulative rewards starting from a given state, while the action-value function estimates the same, but conditioned on taking a specific action first. These value functions serve as the compass guiding the Actor towards states and actions that lead to greater rewards. In practical terms, the Actor-Critic framework embodies a continuous cycle of decision-making and evaluation. The Actor selects actions based on its policy, interacts with the environment, and collects rewards. The Critic then evaluates the chosen actions, updating its value function to reflect the Agent's performance. This feedback loop continues iteratively, with both the Actor and the Critic refining their roles and strategies. The Actor-Critic framework's adaptability and versatility make it a cornerstone in the field of reinforcement learning. Its applications span diverse domains, from robotics and game playing to finance and autonomous vehicles. In robotics, for instance, an autonomous robot can utilize the Actor-Critic framework to learn and refine its control policies, enabling it to perform tasks with precision and adapt to changing conditions.

As we embark on this journey into the depths of the Actor-Critic framework, we will explore its real-world applications, its synergy with deep learning in Deep Actor-Critic networks, and the ongoing research that continues to expand its horizons. We will witness how this collaborative approach, reminiscent of a symphony, orchestrates intelligent decision-making in complex, dynamic environments, offering a promising path toward creating truly autonomous agents.

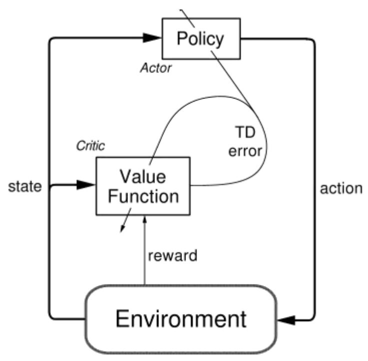


Figure 2.1: Actor-Critic Architecture

2.3.2 Actor-Critic Methods as a tool to find Path

Path planning aims to compute collision-free optimal paths from the starting positions to the goals for the virtual crowd in the situated environment, which has been an important aspect in virtual crowd simulation and widely used in many applications including entertainment, military training, seismic evacuation, and intelligent transportation. There are two types of path planning algorithms: global path planning and local collision avoidance. The global path planning models the environment as a map at first and then generates an optimal path, which is not applicable to a dynamic and unknown environment. The local collision avoidance can make the agent avoid collisions but may fail to generate a path to the final goal. Recently, deep reinforcement learning (DRL) has been studied to deal with path planning in an unknown environment because it combines the excellent perceiving ability of deep learning and decision-making ability of reinforcement learning.

With the increase of the distance between the starting position and the goal, it takes more steps for the agent to explore the environment. However, the positive reward is only given to the agent for reaching the goal. Therefore, it is slow and difficult to train the policy from scratch through extensive trial and error. Imitation learning enables an agent to learn a policy by imitating the behavior of an expert in a reward-sparse environment, but expert demonstrations are difficult to be obtained. The policy which is converged in a local optimum early often outputs the action distribution with low variance and shows less adaptability to the change in the environment. The entropy4 of the policy ensures more random actions are taken during the training, which can avoid getting stuck in a local optimum too early. Proximal policy optimization (PPO) maximizes expected return and uses the entropy as a regularization term to ensure diverse enough actions, while soft actor critic (SAC) simultaneously maximizes expected return and the entropy. Memory8 is crucial for the agent to plan a path in a partially observable environment. Without historical observations,9 the agent has to act only according to the immediate observation and has trouble in approaching the goal far away. The approaches based on velocity obstacle (VO) are widely used for avoiding collisions with dynamic obstacles, which compute a collision region for each agent. By sequencing a set of lessons with increasing difficulties, the SAC agent is able to focus on the easy lesson first and then move on to the difficult lesson and the final lesson. We stack the state information to alleviate the limit of local observation in policy learning. A comprehensive reward function based on VO algorithm is designed to encourage agents to reach their target position as quickly as possible and avoid collisions with other agents and obstacles. The proposed approach makes the network converge to a solution with good generalization performance and adaptability.

We can summarize some points which are as follows : (1) we utilize the SAC algorithm to maximize the entropy of the path planning policy, which improves the adaptability of agents to new changes in a dynamic environment. (2) We adopt curriculum learning[1] to train the neural networks of the SAC algorithm in lessons ranging from easy to difficult, which accelerates the convergence of the policy in an unknown environment and achieves a good generalization capability of the learned policy. (3) We consider the stacked state information as one component of the input of the neural network and design a comprehensive reward function based on the VO concept, which encourages the agents to reach their goals successfully without colliding with static obstacles and other agents. [21]

[1]Curriculum learning is a training strategy that trains the machine learning model starting with a small set of simple examples and finally with the whole training dataset.

2.4 Proximal Policy Optimization (PPO)

2.4.1 Introduction to PPO

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that represents a pivotal shift in addressing these challenges. It doesn't merely aim to teach machines to navigate complex environments; it aspires to do so with a level of stability and safety that surpasses its predecessors. The name 'Proximal Policy Optimization' may sound technical, but it encapsulates the essence of the approach. The term 'Policy' refers to the strategy or behavior an agent employs to make decisions, while 'Proximal' suggests the notion of maintaining a policy close to the existing one. This hint of proximity alludes to the strategy PPO employs to achieve stability – a critical aspect of any learning algorithm. To understand the significance of PPO, it's essential to delve into the intricacies of reinforcement learning. In this framework, an 'agent' interacts with an 'environment,' making decisions and taking actions to maximize the cumulative rewards it receives. These rewards serve as signals guiding the agent's learning process. The agent's objective is to discover the optimal policy – the sequence of actions that yields the most substantial cumulative rewards. PPO sets out to achieve this goal by optimizing the agent's policy – the strategy it uses to select actions based on its observations of the environment. In doing so, it grapples with a fundamental challenge: how to strike the right balance between exploring new strategies and exploiting known ones. Exploration is essential for learning, but excessive exploration can lead to suboptimal performance and inefficiency. Exploitation is vital for leveraging existing knowledge, but over-reliance on it can lead to stagnation and missed opportunities.

Here is where PPO introduces a nuanced approach. It seeks to optimize the policy while ensuring that the policy updates are not too drastic. This notion of controlled policy updates, referred to as the 'proximal' aspect, adds a layer of stability to the learning process. Instead of radically changing the policy, PPO nudges it gently towards improved strategies, incrementally refining its decision-making prowess. A pivotal concept within PPO is the objective function, which quantifies the quality of a policy. This function guides the optimization process, providing a clear direction for policy updates. PPO utilizes a 'proximal objective,' which ensures that the policy updates remain within a certain proximity to the existing policy. This proximal constraint serves as a safety net, preventing the algorithm from making overly aggressive or risky policy changes.

The fundamental understanding of PPO lies in its simplicity and effectiveness. It employs a trust region optimization approach, which limits how far the policy can change in each iteration. This constraint, reminiscent of a safety harness, prevents the algorithm from straying too far from the familiar territory of good policies, ensuring that it remains robust and resistant to catastrophic failures. As PPO iteratively refines the policy, it benefits from a remarkable property – it can be used with deep neural networks, making it applicable to high-dimensional state spaces and complex tasks. This marriage of PPO and deep learning, often referred to as Deep Proximal Policy Optimization, has opened up new frontiers in reinforcement learning, enabling machines to excel in tasks that were previously considered insurmountable.

The applications of PPO are as diverse as the domains it seeks to conquer. From training autonomous robots to navigate unpredictable environments to teaching virtual agents to play complex games, PPO has demonstrated its mettle. In the world of robotics, PPO can facilitate the development of agile, adaptable machines that can assist in search and rescue missions or perform intricate tasks in industrial settings. In gaming, it has led to the creation of AI agents that can rival human players in skill and strategy.

This introduction merely scratches the surface of Proximal Policy Optimization's vast potential and impact. As we delve deeper into the intricacies of this approach, we will uncover its real-world applications, explore the synergy with deep learning, and understand the ongoing research endeavors that seek to push the boundaries of reinforcement learning. We will witness how PPO represents a remarkable fusion of stability and adaptability, reshaping the landscape of artificial intelligence and providing a blueprint for machines to learn and thrive in the dynamic, uncertain world. Proximal Policy Optimization stands as a testament to the relentless pursuit of stability and safety in reinforcement learning. Its ability to combine controlled policy updates with the power of deep learning has opened doors to new possibilities in artificial intelligence. As we navigate the uncharted waters of this approach, we embark on a journey that reveals the transformative potential of PPO in shaping the future of intelligent systems and autonomous decision-making.

2.4.2 The Algorithm

The mathematical equation of PPO is shown below:

$$L^{CLIP}(\theta) = \bar{E}_t[\min(r_t(\theta)\bar{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\bar{A}_t)]$$

where
θ is policy parameter

$\bar{E}_t$ denotes the empirical expectations over timesteps
r_t denotes the ratio of the probabilities under the new and old policies respectively
A_t is estimated advantage at time t
ε is a hyperparameter, usually 0.1 or 0.2

The following important inferences can be drawn from the PPO equation:

1. It is a policy gradient optimization algorithm, that is, in each step there is an update to an existing policy to seek improvement on certain parameters
2.It ensures that the update is not too large, that is the old policy is not too different from the new policy (it does so by essentially "clipping" the update region to a very narrow range)
3. Advantage function is the difference between the future discounted sum of rewards on a certain state and action, and the value function of that policy.
4. Importance Sampling ratio, or the ratio of the probability under the new and old policies respectively, is used for update
5. ε is a hyperparameter denotes the limit of the range within which the update is allowed

This is how the working PPO algorithm looks, in it's entirety when implemented in Actor-Critic style:

```
Input : Intial policy parameters
for iteration = 1,2, ... do
for actor = 1,2, ... do
Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
Compute Advantage estimates $\bar{A}_1, ..., \bar{A}_t$
end for
Optimize surrogate L wrt $\theta$, with K epoches with Minibatch size $M <= NT$
$\theta_{old} \leftarrow \theta$
end for
```

2.4.3 PPO for Path Planning

Ianenko et al. described in their work on Proximal Policy Optimization that complete coverage path planning is a problem of finding a shortest trajectory from an origin point to a target point so that the agent passes through all reachable space points while avoiding collisions with obstacles. This issue is fundamentally important in many robotics applications including demining, farming, cleaning, inspection of complex structures, underwater operations. The task is essentially a special case of covering salesman problem which is an NPhard problem, so the computational time required to solve the problem increases rapidly with problem size. A necessity to handle large unknown dynamic environments with moving obstacles further increases the complexity of the problem and questions the applicability of various classic approaches. Most algorithms developed to solve the coverage path planning do not provide any guarantee of completeness, i.e. there's no way to prove that the algorithm does not leave some regions of the space uncovered. Although this property is critical for some applications (e.g. demining) and several works provide provable complete solutions, heuristics may be preferable in many cases due to limited computational resources and sensor data available to an agent.

Reinforcement learning (RL) methods have shown tremendous success achieving a performance comparable to that of an expert human in complex environments such as in Atari games. Beyond the unmatched performance these methods reach, they remove the necessity to program domain-specific heuristics explicitly. However, to the best of our knowledge, there are quite a few papers exploring the application of reinforcement learning methods for coverage path planning problem. Hence our main objective was to find out whether RL-based approach performed well compared to some classical algorithms.

Their work presents a reinforcement learning solution for coverage path planning in a 2D simulated grid environment that is able to handle large maps with complex topology and dynamic obstacles. Our experiments show that reinforcement learning can find policies that generate complex behavior that allows agent to successfully cover an unknown dynamic environment with low repetition rate. To handle multi-room environments, we exploit a hierarchical approach. On the first level, we use watershed segmentation algorithm to decompose the environment into subregions and derive the optimal path that connects them with an A* algorithm. On the second level we use RL agent to plan a trajectory of an agent within current subregion. In large environments it is preferable to perform the coverage task with multiple agents for faster coverage and higher fault tolerance, i.e. even if some agent fails others can complete the task and cover the rest of the environment. We present experimental results showing that the policy learned in a single-agent mode can efficiently work when placed in multi-agent setting with no additional training. [22]

2.5 Monte Carlo Tree Search

2.5.1 Introduction to MCTS

Monte Carlo Tree Search (MCTS) is a heuristic search set of rules that has won big attention and reputation within the discipline of synthetic intelligence, specially in the area of choice-making and game playing. It is known for its ability to effectively handle complex and strategic video games with massive search areas, in which traditional algorithms may additionally struggle due to the full-size number of feasible actions or actions. MCTS combines the standards of Monte Carlo strategies, which rely upon random sampling and statistical evaluation, with tree-primarily based search techniques. Unlike traditional search algorithms that rely upon exhaustive exploration of the entire seek area, MCTS specializes in sampling and exploring only promising areas of the hunt area.

The center idea in the back of MCTS is to build a seek tree incrementally by using simulating more than one random performs (regularly known as rollouts or playouts) from the current recreation nation. These simulations are carried out until a terminal state or a predefined intensity is reached. The results of these simulations are then backpropagated up the tree, updating the records of the nodes visited at some stage in the play, which includes the wide variety of visits and the win ratios. As the search progresses, MCTS dynamically balances exploration and exploitation. It selects moves through considering both the exploitation of notably promising movements with high win ratios and the exploration of unexplored or less explored moves. This balancing is finished through the usage of an top confidence sure (UCB) components, which includes the Upper Confidence Bounds for Trees (UCT), to decide which moves or nodes to visit for the duration of the hunt. MCTS has been efficiently implemented in numerous domains, including board games (e.G., Go, chess, and shogi), card video games (e.G., poker), and video games. It has done splendid overall performance in lots of challenging recreation-gambling scenarios, frequently surpassing human understanding. MCTS has also been prolonged and tailored to deal with different trouble domains, which include making plans, scheduling, and optimization.

One of the exquisite blessings of MCTS is its ability to handle video games with unknown or imperfect data, as it relies on statistical sampling as opposed to whole know-how of the game state. Additionally, MCTS is scalable and may be parallelized efficaciously, making it suitable for disbursed computing and multi-core architectures. Monte Carlo Tree Search (MCTS) is a search technique in the field of Artificial Intelligence (AI). It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning. In tree search, there's always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the " exploration-exploitation trade-off ". It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best.

Exploration helps in exploring and discovering the unexplored parts of the tree, which could result in finding a more optimal path. In other words, we can say that exploration expands the tree's breadth more than its depth. Exploration can be useful to ensure that MCTS is not overlooking any potentially better paths. But it quickly becomes inefficient in situations with large number of steps or repetitions. In order to avoid that, it is balanced out by exploitation. Exploitation sticks to a single path that has the greatest estimated value. This is a greedy approach and this will extend the tree's depth more than its breadth. In simple words, UCB formula applied to trees helps to balance the exploration-exploitation trade-off by periodically exploring relatively unexplored nodes of the tree and discovering potentially more optimal paths than the one it is currently exploiting. For this characteristic, MCTS becomes particularly useful in making optimal decisions in Artificial Intelligence (AI) problems.

2.5.2 Why use Monte Carlo Tree Search (MCTS)?

Here are some reasons why MCTS is commonly used:

Handling Complex and Strategic Games: MCTS excels in games with large search spaces, complex dynamics, and strategic decision-making. It has been successfully applied to games like Go, chess, shogi, poker, and many others, achieving remarkable performance that often surpasses human expertise. MCTS can effectively explore and evaluate different moves or actions, leading to strong gameplay and decision-making in such games.

Unknown or Imperfect Information: MCTS is suitable for games or scenarios with unknown or imperfect information. It relies on statistical sampling and does not require complete knowledge of the game state. This makes MCTS applicable to domains where uncertainty or incomplete information exists, such as card games or real-world scenarios with limited or unreliable data.

Learning from Simulations: MCTS learns from simulations or rollouts to estimate the value of actions or states. Through repeated iterations, MCTS gradually refines its knowledge and improves decision-making. This learning aspect makes MCTS adaptive and capable of adapting to changing circumstances or evolving strategies.

Optimizing Exploration and Exploitation: MCTS effectively balances exploration and exploitation during the search process. It intelligently explores unexplored areas of the search space while exploiting promising actions based on existing knowledge. This exploration-exploitation trade-off allows MCTS to find a balance between discovering new possibilities and exploiting known good actions.

Scalability and Parallelization: MCTS is inherently scalable and can be parallelized efficiently. It can utilize distributed computing resources or multi-core architectures to speed up the search and handle larger search spaces. This scalability makes MCTS applicable to problems that require significant computational resources.

Applicability Beyond Games: While MCTS gained prominence in game-playing domains, its principles and techniques are applicable to other problem domains as well. MCTS has been successfully applied to planning problems, scheduling, optimization, and decision-making in various real-world scenarios. Its ability to handle complex decision-making and uncertainty makes it valuable in a range of applications.

Domain Independence: MCTS is relatively domain-independent. It does not require domain-specific knowledge or heuristics to operate. Although domain-specific enhancements can be made to improve performance, the basic MCTS algorithm can be applied to a wide range of problem domains without significant modifications.

2.5.3 The Algorithm

In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of a number of simulations. The process of Monte Carlo Tree Search can be broken down into four distinct steps, viz., selection, expansion, simulation and backpropagation. Each of these steps is explained in details below:

Selection : In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During tree traversal, a node is selected based on some parameters that return the maximum value. The parameters are characterized by the formula that is typically used for this purpose is given below.

$$S_i = x_i + C\sqrt{ln(t)/n_i}$$

where:
S_i = value of a node i
x_i = empirical mean of a node i
C = a constant
t = total number of simulations

When traversing a tree during the selection process, the child node that returns the greatest value from the above equation will be one that will get selected. During traversal, once a child node is found which is also a leaf node, the MCTS jumps into the expansion step.

Expansion : In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.

Simulation : In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved.

Backpropagation : After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it backpropagates from the new node to the root node. During the process, the number of simulation stored in each node is incremented. Also, if the new node's simulation results in a win, then the number of wins is also incremented.

2.5.4 MCTS for Path Planning

Eiffert et al. presented an integrated path planning framework using generative Recurrent Neural Networks (RNNs) and Monte Carlo Tree Search (MCTS). This

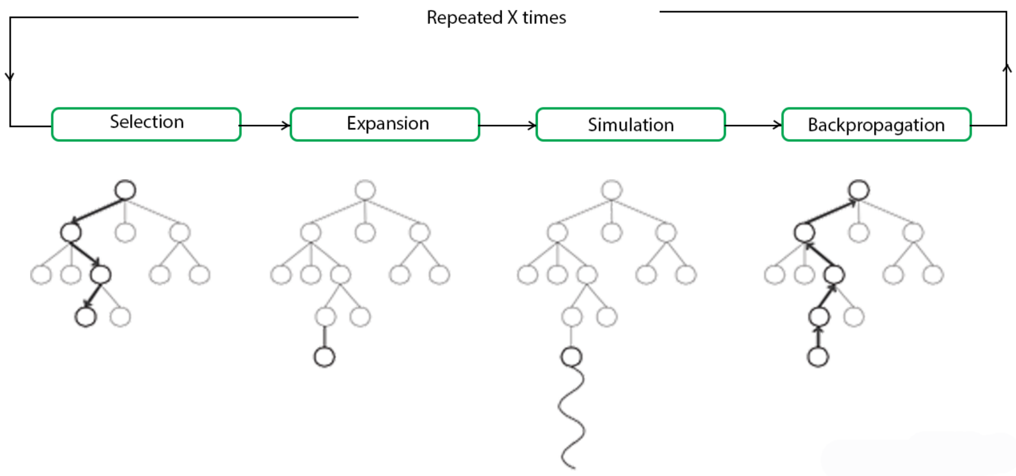


Figure 2.2: Monte Carlo Tree Search

builds upon recent work in motion prediction of crowds. We use RNNs in an encoder-decoder architecture, where the input at each timestep is the current position of agent and the future position of the robot, which is known during training and sampled during planning. This approach allows the encoding of observed sequences of agent positions and robot actions, which can be used by the decoder stage to generate likely responses of all agents to a robot's action. The generative model is used within the MCTS to simulate state transitions for sampled actions during a tree search of the robot's action space.

They validate the proposed approach on three varied datasets, including interactions between pedestrians and a vehicle, livestock and a mobile robot, and simulated interactions of pedestrians generated using the optimal reciprocal collision avoidance (ORCA) model of motion. The performance of their path planning method is compared to existing approaches including a reactive potential field and deep reinforcement learning. The results demonstrate that not only does our planning algorithm perform comparably to state of the art methods for collision avoidance, but more importantly, it is able to direct the future states of nearby individuals using a motion model learnt from real world data, allowing application to tasks such as planning paths that maneuver nearby individuals, or herding of livestock towards a goal. [23]

Chapter 3

Software Used in the Research Work and Related concepts

For preparing the environment and setting up the necessaries to do the particular experiment for **"Path Planning using Policy Iteration in a simple grid with static environment"**, we used **Python** as the programming language and **Pycharm** as the IDE (Integrated Development Environment)

3.1 About Python

Python, a versatile and high-level programming language, was created by Guido van Rossum and first released in 1991. Its design emphasizes code readability, making it an ideal choice for both beginners and experienced developers. Python's simplicity is evident in its clean and consistent syntax, using indentation to define code blocks, rather than relying on braces or keywords.

Python's popularity has soared due to its wide range of applications. It is extensively used in web development, with frameworks like Django and Flask simplifying the creation of robust web applications. In data science, Python shines with libraries such as NumPy, pandas, and Matplotlib for data manipulation, analysis, and visualization. Furthermore, Python plays a crucial role in artificial intelligence and machine learning through TensorFlow, PyTorch, and scikit-learn.

Its versatility extends to scripting, automation, scientific computing, and more. Python's cross-platform compatibility allows code to run seamlessly on various operating systems. The active Python community ensures continuous development, and its extensive standard library simplifies tasks ranging from file handling to networking.

In recent years, Python's popularity has skyrocketed, making it one of the top programming languages globally. Its readability, extensive ecosystem, and ease of learning make it an invaluable tool for developers across diverse domains.

Today, Python is ubiquitous in the software development landscape. It serves as the foundation for web development, with frameworks like Django and Flask empowering developers to create robust web applications. In data science, Python is the language of choice due to libraries such as NumPy, pandas, and SciPy, facilitating data manipulation, analysis, and statistical modeling. Python's machine learning libraries, including TensorFlow and PyTorch, have revolutionized AI and deep learning.

Python's versatility extends beyond these domains. It's used for scripting, automation, scientific computing, and more. Its cross-platform compatibility ensures code portability, running smoothly on various operating systems.

The Python community, characterized by its welcoming nature and active development, contributes to Python's sustained growth. This programming language's enduring appeal lies in its simplicity, versatility, and extensive ecosystem, cementing its status as a dominant force in modern software development.

3.2 Why Python in this particular research work?

Python, a versatile and dynamic programming language, has gained immense popularity in the realm of research across various disciplines. Whether you are exploring complex data sets, simulating physical phenomena, conducting statistical analyses, or developing machine learning models, Python offers a multitude of libraries, ease of use, and a supportive community that makes it an indispensable tool for researchers. In this essay, we will delve into the reasons why Python is a preferred choice for research work.

Open Source and Accessibility: Python's open-source nature means it is free to use and can be easily installed on multiple platforms. This accessibility breaks down financial barriers, enabling researchers worldwide to utilize it without incurring hefty software licensing costs.

Rich Ecosystem of Libraries: Python boasts an extensive ecosystem of specialized libraries and frameworks. Libraries like NumPy, SciPy, pandas, and Matplotlib provide powerful tools for data manipulation, numerical computing, and data visualization, making Python particularly valuable for data-centric research.

Machine Learning and Deep Learning: Python is the go-to language for machine learning and deep learning research. Libraries like TensorFlow, PyTorch, and scikit-learn simplify the development and implementation of complex algorithms, allowing researchers to tackle intricate problems in fields ranging from natural language processing to computer vision.

Community and Documentation: Python's thriving community ensures continuous development and maintenance. Researchers can easily find help through forums, tutorials, and extensive documentation. This support system is invaluable when troubleshooting issues or optimizing code for performance.

Interoperability: Python can seamlessly interface with other languages like C, C++, and Java. This flexibility is essential when integrating specialized, high-performance libraries or utilizing existing codebases in research projects.

Data Analysis and Visualization: In research, data analysis and visualization are crucial. Python's libraries simplify the process, allowing researchers to explore and communicate their findings effectively. Jupyter Notebooks, for example, provide an interactive environment for combining code, data, and visualizations in a single document.

Reproducibility and Transparency: Python's code readability and ease of sharing make research more reproducible and transparent. Colleagues and peers can easily review, replicate, and build upon research findings, fostering collaboration and trust within the scientific community.

High-Performance Computing: Python can harness the power of high-performance computing clusters through libraries like Dask and mpi4py. This capability is indispensable for researchers working on computationally intensive tasks such as climate modeling or simulating physical processes.

Cross-Disciplinary Applications: Python's versatility transcends disciplines. Researchers from diverse fields such as biology, physics, economics, and social sciences can utilize Python for data analysis, modeling, and visualization, promoting interdisciplinary collaboration and innovation.

Customization and Extensibility: Python allows researchers to tailor solutions to their specific needs. They can easily create custom functions, modules, or even entire packages to address unique research challenges, providing a significant degree of flexibility.

Integration with Big Data Technologies: In the era of big data, Python integrates seamlessly with big data technologies like Apache Hadoop and Apache Spark. This enables researchers to process and analyze vast datasets efficiently, opening new avenues for research.

Natural Language Processing (NLP): With the rise of textual data in research, Python's NLP libraries like NLTK and spaCy are invaluable for text analysis, sentiment analysis, and language modeling, making it a preferred choice in fields such as linguistics and social sciences.

Robust Testing and Debugging Tools: Python offers a wide range of testing and debugging tools that facilitate the development of reliable and error-free research code. This ensures that research results are based on sound programming practices.

Scalability: Python's scalability allows researchers to start with small-scale experiments and easily scale up to larger projects as needed. This flexibility is especially advantageous when the scope of research expands.

Ethical Considerations: Python's emphasis on readability and maintainability aligns with ethical considerations in research. Well-documented and clear code helps ensure the ethical treatment of research subjects and data.

In conclusion, Python's prominence in research is not arbitrary. Its open-source nature, extensive libraries, community support, and versatility empower researchers to tackle complex problems, analyze vast datasets, and develop innovative solutions in a cost-effective and transparent manner. As research continues to evolve, Python's adaptability and comprehensive toolset position it as an indispensable tool for scientists and researchers across the globe, advancing our understanding of the world and driving innovation in countless domains.

3.3 Machine Learning, Deep Learning and Python

In today's data-driven world, the fields of Machine Learning (ML) and Deep Learning (DL) stand as transformative forces reshaping industries, research, and everyday life. These technologies have progressed far beyond their conceptual beginnings, becoming integral in fields as diverse as healthcare, finance, autonomous vehicles, and natural language processing. This essay explores the fundamental concepts, applications, and the transformative potential of ML and DL.

3.3.1 Understanding Machine Learning:

Machine Learning is a subset of artificial intelligence that focuses on creating algorithms capable of learning from data and making predictions or decisions without being explicitly programmed. It relies on the principle that systems can automatically learn and improve from experience. Key components of ML include:

1. Data: Data is the lifeblood of machine learning. Algorithms require vast amounts of data to identify patterns, trends, and make predictions.

2. Algorithms: These are mathematical models designed to process data and make decisions. Popular ML algorithms include decision trees, random forests, and support vector machines.

3. Training: ML models are 'trained' on labeled datasets. During this process, the model learns to make predictions by recognizing patterns in the data.

4. Evaluation: After training, models are tested on new, unseen data to evaluate their performance and accuracy.

3.3.2 Applications of Machine Learning:

1. Image Recognition: ML has revolutionized image recognition technology. Deep learning models like Convolutional Neural Networks (CNNs) are now capable of identifying objects in images with astonishing accuracy.

2. Natural Language Processing (NLP): ML and DL models have transformed the way machines understand and generate human language. Chatbots, language translation, and sentiment analysis are examples of NLP applications.

3. Healthcare: ML is used to predict disease outbreaks, diagnose medical conditions, and personalize treatment plans. DL is especially promising in analyzing medical images like X-rays and MRIs.

3.3.3 Understanding Deep Learning:

Deep Learning is a subset of ML that focuses on neural networks with multiple layers, known as deep neural networks. These networks attempt to mimic the human brain's structure, allowing them to process vast amounts of data with remarkable accuracy. Key components of DL include:

1. Neural Networks: These are interconnected layers of nodes, or neurons, which process and transmit information. Deep networks have multiple hidden layers that enable them to model complex relationships in data.

2. Backpropagation: This is the primary training technique for deep neural networks. It adjusts the network's weights and biases to minimize the difference between predicted and actual outcomes.

3. Activation Functions: These functions introduce non-linearity into neural networks, allowing them to approximate complex functions. Popular activation functions include ReLU and Sigmoid.

3.3.4 Applications of Deep Learning:

1. Autonomous Vehicles: DL is crucial for self-driving cars. These systems use neural networks to process real-time data from sensors and make driving decisions.

2. Voice and Speech Recognition: Services like Siri and Alexa rely on DL models to understand and respond to human speech.

3. Recommendation Systems: Streaming platforms like Netflix use DL algorithms to personalize content recommendations based on user preferences.

3.3.5 The Transformative Potential:

Machine Learning and Deep Learning are not just tools but catalysts for change across numerous industries:

1. Healthcare: DL can analyze medical images faster and with greater accuracy than human experts, potentially revolutionizing diagnosis and treatment.

2. Finance: ML models can predict market trends and detect fraudulent transactions in real-time, enhancing financial security and investment strategies.

3. Manufacturing: Predictive maintenance models based on ML can reduce downtime by identifying potential equipment failures before they occur.

4. Education: Personalized learning systems driven by ML can adapt to individual student needs, making education more effective and engaging.

5. Environmental Conservation: ML can analyze climate data, monitor wildlife populations, and help in conservation efforts.

Machine Learning and Deep Learning have come a long way from their theoretical foundations to practical, transformative tools. As they continue to evolve, their potential to reshape industries and improve our lives remains vast. However, ethical considerations, such as data privacy and algorithm bias, must be carefully addressed to ensure these technologies benefit society as a whole. The journey of Machine Learning and Deep Learning is far from over, promising exciting developments that will continue to redefine what's possible in the world of artificial intelligence.

3.4 Libraries used in Machine Learning and Deep Learning Regularly

3.4.1 Libraries Used in Machine and Deep Learning majorly:

Machine learning and deep learning have rapidly evolved in recent years, enabling computers to learn from data and make intelligent decisions. This progress has been driven in large part by the development and availability of powerful libraries that provide the necessary tools and frameworks for researchers and developers. These libraries are essential for building, training, and deploying machine learning and deep learning models efficiently. Here's an overview of some of the most widely used libraries in these fields:

1. **TensorFlow:** TensorFlow, developed by Google Brain, is one of the most popular deep learning libraries. It provides a comprehensive ecosystem for developing machine learning and deep learning models, supporting both CPU and GPU acceleration. TensorFlow offers high-level APIs like Keras for easy model building and lower-level APIs for maximum flexibility.

2. **PyTorch:** PyTorch, developed by Facebook's AI Research lab (FAIR), has gained immense popularity due to its dynamic computation graph, which makes it more intuitive for researchers and easier to debug. It's known for its flexibility and has become a top choice for deep learning research.

3. **scikit-learn:** Scikit-learn is a widely-used library for traditional machine learning algorithms. It offers a user-friendly interface for classification, regression, clustering, dimensionality reduction, and more. It's a great choice for tasks that don't require deep neural networks.

4. **Keras:** Initially developed as a high-level API running on top of other deep learning libraries like TensorFlow and Theano, Keras has now been integrated directly into TensorFlow. It's renowned for its simplicity and ease of use, making it a fantastic choice for beginners and rapid prototyping.

5. **MXNet:** Apache MXNet is an open-source deep learning framework that's known for its scalability and efficiency, particularly on distributed systems. It provides both symbolic and imperative APIs and is suitable for a wide range of deep learning tasks.

6. CNTK (Microsoft Cognitive Toolkit): CNTK is Microsoft's deep learning library, designed for training deep neural networks efficiently. It's known for its speed and scalability and is commonly used in research and production settings.

7. Theano (Deprecated): Although it's no longer actively developed, Theano played a crucial role in the early days of deep learning. It was one of the first libraries to provide efficient GPU support and symbolic differentiation for building neural networks.

8. XGBoost: XGBoost is a gradient boosting library that excels in structured data problems, such as classification and regression. It's known for its performance and has won numerous machine learning competitions.

9. LightGBM: LightGBM is another gradient boosting library known for its efficiency and speed. It's particularly well-suited for large datasets and has become popular in both competitions and production environments.

10. Pandas: While not a machine learning library per se, Pandas is a critical library for data preprocessing and manipulation. It provides data structures and functions that simplify data cleaning and transformation, making it a foundational tool in the data preprocessing stage.

These libraries have greatly accelerated the development and adoption of machine learning and deep learning techniques. Depending on your specific needs, you may choose one or more of these libraries to build, train, and deploy your models effectively. The choice often depends on factors like the complexity of your project, your familiarity with the library, and the community support available.

3.5 Integrated Development Environment (IDE)

In the ever-evolving landscape of software development, the role of Integrated Development Environments (IDEs) cannot be overstated. IDEs represent a technological marvel, seamlessly bringing together a myriad of tools and functionalities into a unified platform. These robust software applications have transformed the way developers work, fostering efficiency, collaboration, and innovation. In this essay, we explore the significance of IDEs, their evolution, and their profound impact on the world of software development.

The Genesis of IDEs

The early days of programming were marked by laborious and error-prone processes. Developers had to navigate through multiple standalone tools like text editors, compilers, and debuggers, which hindered productivity. The birth of IDEs in the 1980s changed this paradigm by integrating these tools into a single environment. The pioneering IDE, Turbo Pascal, allowed programmers to edit, compile, and debug their code within a unified workspace. This marked the beginning of a revolution that continues to shape the software development industry today.

Streamlining Development Workflows

IDEs excel at simplifying complex development workflows. They provide a single interface where developers can write, test, and deploy code seamlessly. The code editors in IDEs offer features like syntax highlighting, auto-completion, and code navigation, which expedite the coding process. Moreover, IDEs are equipped with powerful debugging tools that enable developers to identify and rectify errors quickly. This streamlining of development processes saves time and reduces the likelihood of introducing bugs, ultimately leading to more reliable software.

Enhancing Collaboration

Collaboration is at the heart of modern software development, and IDEs play a crucial role in facilitating it. With features like version control system integration and real-time collaboration tools, IDEs enable multiple developers to work on the same codebase simultaneously. This fosters teamwork, accelerates project development, and ensures code consistency. In addition, IDEs often support code review workflows, allowing developers to provide feedback and make improvements collaboratively.

Cross-Platform Development

As software development has expanded to encompass various platforms and languages, IDEs have evolved to support this diversity. Many IDEs now offer cross-platform development capabilities, enabling developers to write code for multiple operating systems and devices. Whether it's web development, mobile app development, or desktop application development, IDEs provide the necessary tools and libraries to cater to diverse development needs.

Supporting a Plethora of Languages

Another remarkable aspect of IDEs is their versatility in supporting a wide array of programming languages. From Python and Java to C++ and Ruby, IDEs offer language-specific tools and integrations, making it easier for developers to work in their preferred language. This adaptability promotes innovation by empowering developers to choose the language best suited for their project, rather than being restricted by the limitations of their tools.

Automation and Integration

Automation is a cornerstone of modern software development, and IDEs have embraced this trend wholeheartedly. They incorporate features like code generation, automated testing, and continuous integration and deployment (CI/CD) pipelines. These automation capabilities not only reduce the manual workload but also enhance code quality by enforcing best practices and standards.

In conclusion, Integrated Development Environments have revolutionized the world of software development. They have evolved from simple code editors to multifaceted platforms that streamline workflows, foster collaboration, and support diverse programming languages and platforms. IDEs are not just tools; they are enablers of innovation. They empower developers to focus on creativity and problem-solving rather than the intricacies of tool integration. As the software development landscape continues to evolve, IDEs will undoubtedly remain a vital catalyst for innovation in the field, nurturing the seeds of creativity and enabling developers to push the boundaries of what is possible.

3.6 PyCharm, The IDE used in this research work

3.6.1 PyCharm, a Python IDE

PyCharm, a name synonymous with innovation and efficiency in the world of software development, has emerged as the go-to Integrated Development Environment (IDE) for Python enthusiasts and professionals alike. In an era where programming languages like Python have become the driving force behind technological advancements, PyCharm stands as a testament to the evolution of tools designed to elevate the development experience.

The story of PyCharm begins with JetBrains, a company renowned for its commitment to developing powerful IDEs. With PyCharm, they have not only crafted a tool but also an ecosystem that caters exclusively to Python development. This ecosystem comprises two editions: PyCharm Community, which is open-source and free, and PyCharm Professional, a premium version packed with features designed to meet the demands of professional software development.

One of the standout features of PyCharm is its intuitive and user-friendly interface. The developers at JetBrains have paid meticulous attention to detail, creating an environment that seamlessly integrates code editing, debugging, testing, and project management. The result is a fluid and harmonious workflow that allows developers to focus on their creative process without the distraction of cumbersome tool management.

PyCharm's code editor is a masterpiece in itself. It offers features such as intelligent code completion, real-time error highlighting, and advanced code navigation that significantly boost productivity. The code analysis capabilities are particularly noteworthy, as they assist developers in writing clean and maintainable code by suggesting improvements and adhering to Python's PEP 8 coding style guidelines.

For those diving into web development with Python, PyCharm is equipped with outstanding Django support. It provides integrated tools for managing Django projects, templates, and migrations, simplifying the complexities of web development and enabling developers to create web applications efficiently.

Another key feature that sets PyCharm apart is its exceptional support for version control systems like Git. Developers can perform all Git operations right from within the IDE, streamlining collaboration and ensuring version control best practices are effortlessly integrated into the development process.

PyCharm also boasts a rich plugin ecosystem, allowing developers to extend its functionality to suit their specific needs. Whether it's integrating third-party libraries, customizing code inspections, or enhancing the user interface, the plugin system offers endless possibilities for tailoring PyCharm to individual preferences.

Moreover, PyCharm supports various project types, from data science projects with Jupyter notebooks to scientific computing with NumPy and SciPy. It effortlessly adapts to the diverse needs of the Python community, making it an invaluable tool for a wide range of professionals, from data scientists to web developers to machine learning engineers.

In essence, PyCharm is more than just an IDE; it's a creative canvas where Python developers can bring their ideas to life. It encapsulates the spirit of innovation and collaboration that defines the Python community. As the Python ecosystem continues to expand and evolve, PyCharm remains at the forefront, providing developers with the tools they need to turn their visions into reality. It is a testament to the power of thoughtful design, user-centric development, and the enduring importance of IDEs in shaping the future of software development.

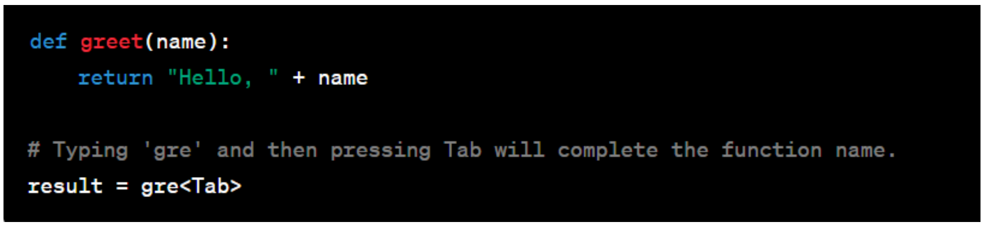
3.6.2 PyCharm, working functionalities with examples

PyCharm, as an Integrated Development Environment (IDE) for Python, offers a robust set of features to facilitate the software development process. Let's delve into how PyCharm works with examples to illustrate its functionality.

1. Code Editing: PyCharm provides an intuitive code editor that supports features like syntax highlighting, code completion, and intelligent code analysis. For example, as you type, it can suggest code completions based on the context:

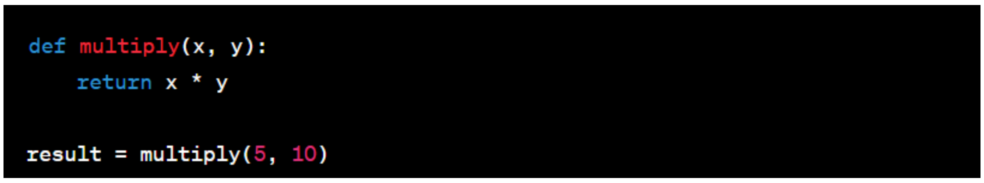
2. Code Navigation: Developers can easily navigate through code with features like "Go to Definition" and "Find Usages." For instance, you can quickly jump to the definition of a function by placing the cursor on it and using a shortcut or right-clicking:

3. Integrated Debugger: PyCharm offers a powerful integrated debugger



```python
def greet(name):
    return "Hello, " + name

# Typing 'gre' and then pressing Tab will complete the function name.
result = gre<Tab>
```

Figure 3.1: Completion of code as suggestion



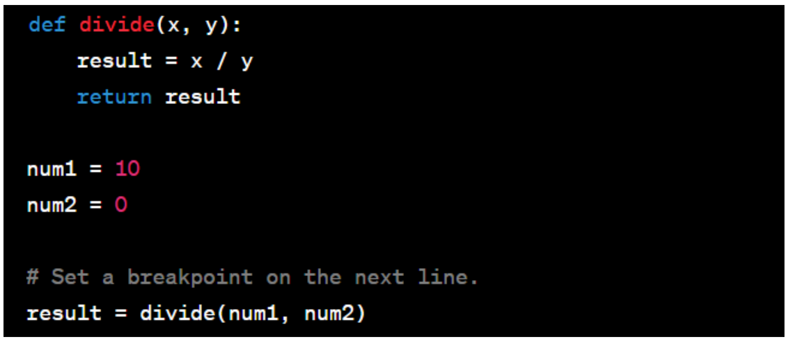
```python
def multiply(x, y):
    return x * y

result = multiply(5, 10)
```

Figure 3.2: Pycharm as code navigator

that helps identify and fix issues in your code. You can set breakpoints, inspect variables, and step through code execution:

4. Version Control: PyCharm has built-in support for version control systems like Git. You can commit changes, view history, and resolve merge conflicts directly from the IDE:

5. Project Management: It allows you to create and manage projects effortlessly. You can organize your code into directories, manage dependencies using the integrated package manager, and create virtual environments:

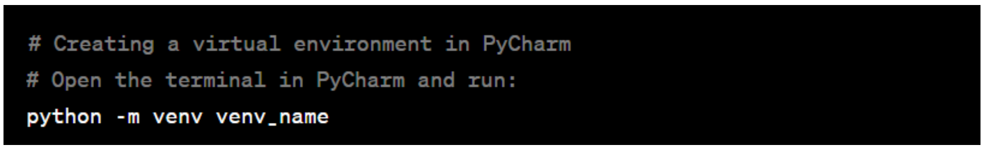


```python
def divide(x, y):
    result = x / y
    return result

num1 = 10
num2 = 0

# Set a breakpoint on the next line.
result = divide(num1, num2)
```

Figure 3.3: Pycharm as Integrated Debugger

```
# Making changes to a Python file
```

Figure 3.4: Version Control



```
# Creating a virtual environment in PyCharm
# Open the terminal in PyCharm and run:
python -m venv venv_name
```

Figure 3.5: Project Management

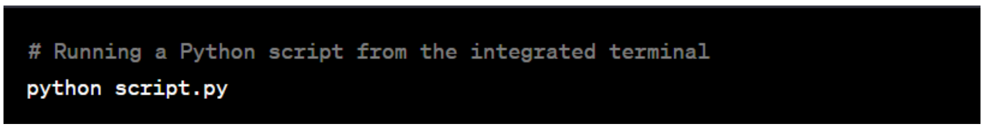
6. Integrated Terminal: PyCharm provides an integrated terminal where we can execute shell commands, manage virtual environments, or run Python scripts.

7. Web Development Support: PyCharm excels in web development with support for frameworks like Django and Flask. It provides project templates, code completion, and integrated web server support.

8. Testing and Profiling: You can run unit tests and profile your code's performance directly from the IDE. PyCharm assists in writing and running tests, ensuring code quality:

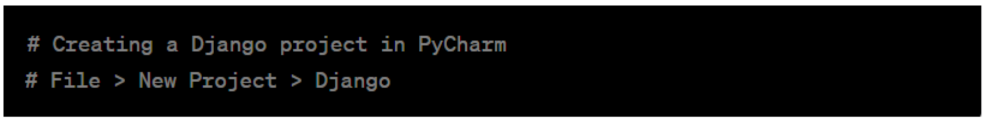
9. Plugins and Extensions: PyCharm's rich plugin ecosystem allows you to extend its functionality. You can install plugins for frameworks, libraries, or custom tools to enhance your development environment.

In conclusion, PyCharm serves as an all-encompassing environment that simplifies Python development. It streamlines coding, debugging, version control, and project management, allowing developers to focus on crafting high-quality software. Its continuous evolution and community-driven plugins make it a vital



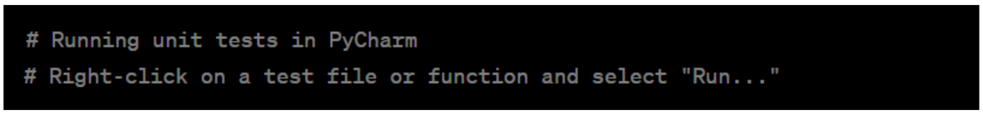
```
# Running a Python script from the integrated terminal
python script.py
```

Figure 3.6: Integrated Terminal



```
# Creating a Django project in PyCharm
# File > New Project > Django
```

Figure 3.7: Pycharm as a support of Web Development



```
# Running unit tests in PyCharm
# Right-click on a test file or function and select "Run..."
```

Figure 3.8: Testing and Profiling

tool for Python developers across various domains.

Chapter 4

Experimentation in a Simple Grid in Static Environment Using Policy Iteration in Pycharm IDE

4.1 Environment Description

In reinforcement learning, Environment is the Agent's world in which it lives and interacts. The agent can interact with the environment by performing some action but cannot influence the rules or dynamics of the environment by those actions. This means if humans were to be the agent in the earth's environments then we are confined with the laws of physics of the planet. We can interact with the environment with our actions but cannot change the physics of our planet.

When an agent performs an action in the environment, the environment returns a new state of the environment making the agent move to this new state. The environment also sends a Reward to the agent which is a scalar value that acts as feedback for the agent whether its action was good or bad.

The whole experimentation is in Python (Programming language) and we have choosen PyCharm as the Integrated Development Environment (IDE) to perform our experimentation as we have found it easier to use Python for having access to numerous number of built-in commands and customize libraries like Numpy, Matplotlib, PyTorch etc. which is not only preferable in Reinforcement Learning (RL), but indeed preferred in as a whole Machine and Deep Learning domain. Firstly, we have created a project in PyCharm to set a **venv** (Virtual Environment) which would be saved in the specific computer where the experimentation took place in a certain location (for example, our file location path in the computer is :

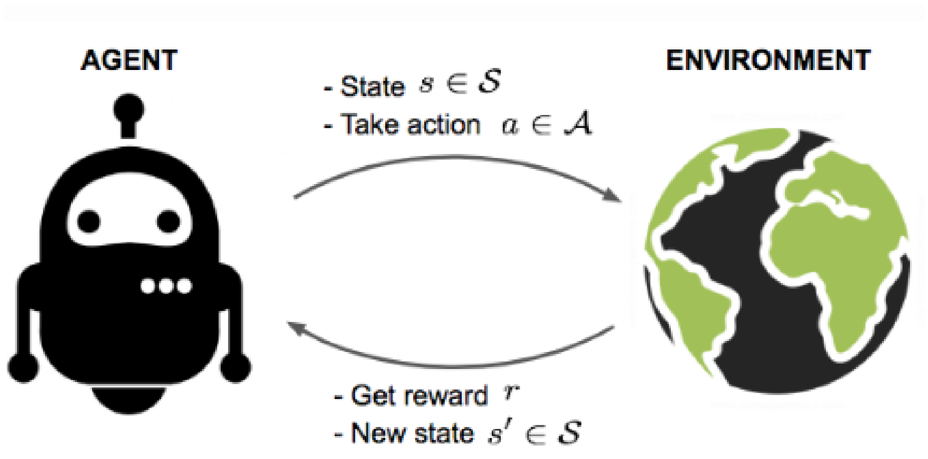

AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 4.1: Relation between the environment and the agent

$"C : \backslash Users \backslash Gourav \backslash PycharmProjects \backslash ThesisWorkonPathPlanning")$.

In this **venv** we have created a file **main.py** where we have written the Python code for experimentation on Path Planning. We have **Numpy** library for various mathematical operations and **Matplotlib** for plotting diagrams other things for plotting result outputs. In the **main.py** Python script, we have created a 15×15 matrix which we named the **Gridworld** (the class name in the script is **Gridworld** as well). The termination state where the path ends is the last block of the grid (224)[1]. In this grid, some obstacle or blocks are placed in the states $70, 71, 85, 86, 100, 101, 115, 116, 130, 131, 136, 137, 138, 139, 151, 152, 153, 154$ respectively and there is a one-way **teleporter or magic square** placed in the grid which connects two state, i.e, state 16 and state 209 respectively.

The reward distribution for the agent to interact with the environment is given below :

1. For every succession i.e, from one state to next state, the reward $\rightarrow -1$
2. On reaching to the terminal state from a non-terminal state, the reward $\rightarrow 0$
3. On making a move from any state except the terminal state to a state where obstacle is placed $\rightarrow -100$

[1]Every state in the 15×15 grid are named $0, 1, ..., 224$

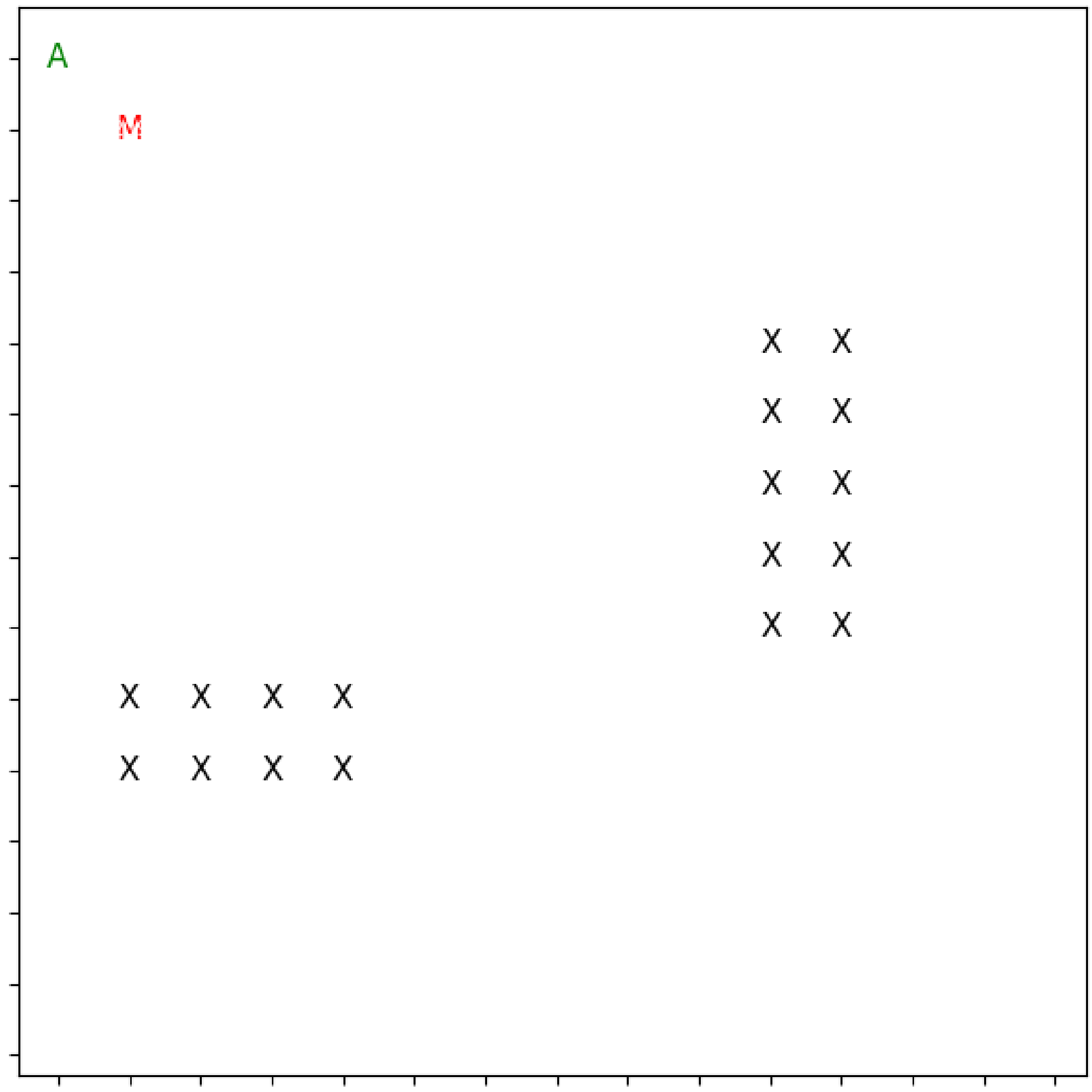


Figure 4.2: A: Agent, M: Teleporter, X: Obstacles, a representation of the environment

4.2 The Bellman Update Equation

According to the Bellman Equation, long-term- reward in a given action is equal to the reward from the current action combined with the expected reward from the future actions taken at the following time. Bellman equation is the basic block of solving reinforcement learning and is omnipresent in RL. It helps us to solve MDP. To solve means finding the optimal policy and value functions. The optimal value function $V^*(S)$ is one that yields maximum value.

The value of a given state is equal to the max action (action which maximizes the value) of the reward of the optimal action in the given state and add a discount factor multiplied by the next state's Value from the Bellman Equation.

$$V(s) = \max_a(R(s,a) + \gamma V(s'))$$

where, $V(s)$ is the value for being in a certain state. $V(s')$ is the value for being in the next state that we will end up in after taking action a. $R(s,a)$ is the reward we get after taking action a in state s. As we can take different actions so we use maximum because our agent wants to be in the optimal state. γ is the discount factor which lies between 0-1. This is the bellman equation in the deterministic environment. It will be slightly different for a non-deterministic environment or stochastic environment.

$$V(s) = \max_a(R(s,a) + \gamma \Sigma_{s'} P(s,a,s')V(s'))$$

In a stochastic environment when we take an action it is not confirmed that we will end up in a particular next state and there is a probability of ending in a particular state. P(s, a, s') is the probability of ending is state s' from s by taking action a. We can solve the Bellman equation using a special technique called dynamic programming.

4.3 Policy Iteration Algorithm

It is the algorithm that we have used to find the optimal path for the agent which is interacting in the 15×15 grid with placing the teleporter as well as the obstacles. Policy iteration algorithm can be discussed in two parts, first is the policy evaluation and the other one is the policy improvement.

4.3.1 Policy Evaluation

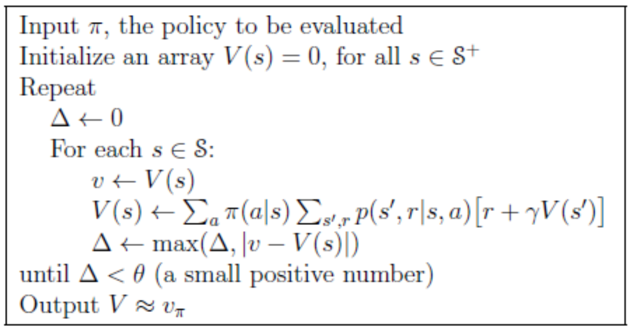
First we consider how to compute the state-value function v_π for an arbitrary policy π. This is called policy evaluation. The state-value function v_π can be written as

$$v_\pi(s) = \Sigma_a \pi(a|s) \Sigma_{s',a} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π, and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π. If the environment's dynamics are completely known, then the above mentioned equation is a system of δ simultaneous linear equations in δ unknowns (the $v_\pi(s)$, $s \in \delta$). In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions $v_1, v_2, v_3, ...$, each mapping δ^+ to $\mathbb{R}$. The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π as an update rule:

$$v_{k+1}(s) = \Sigma_a \pi(a|s) \Sigma_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

for all $s \in \delta$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence v_k can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called iterative policy evaluation.



```
Input $\pi$, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$
```

Figure 4.3: Iterative Policy Evaluation

4.3.2 Policy Improvement

Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s that is $v_\pi(s)$ but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter following the existing policy, π. The value of this way of behaving is

$$q_\pi(s,a) = \Sigma_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

The key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater, that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time—then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall. That this is true is a special case of a general result called the policy improvement theorem.

4.3.3 The Integration - Policy iteration

Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

Figure 4.4: Reaching to optimal policy by iterative evaluation and improvement

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$  (a small positive number)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       $a \leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
       If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V$ and $\pi$; else go to 2
```

Figure 4.5: Policy Iteration combining evaluation and improvement of the policy

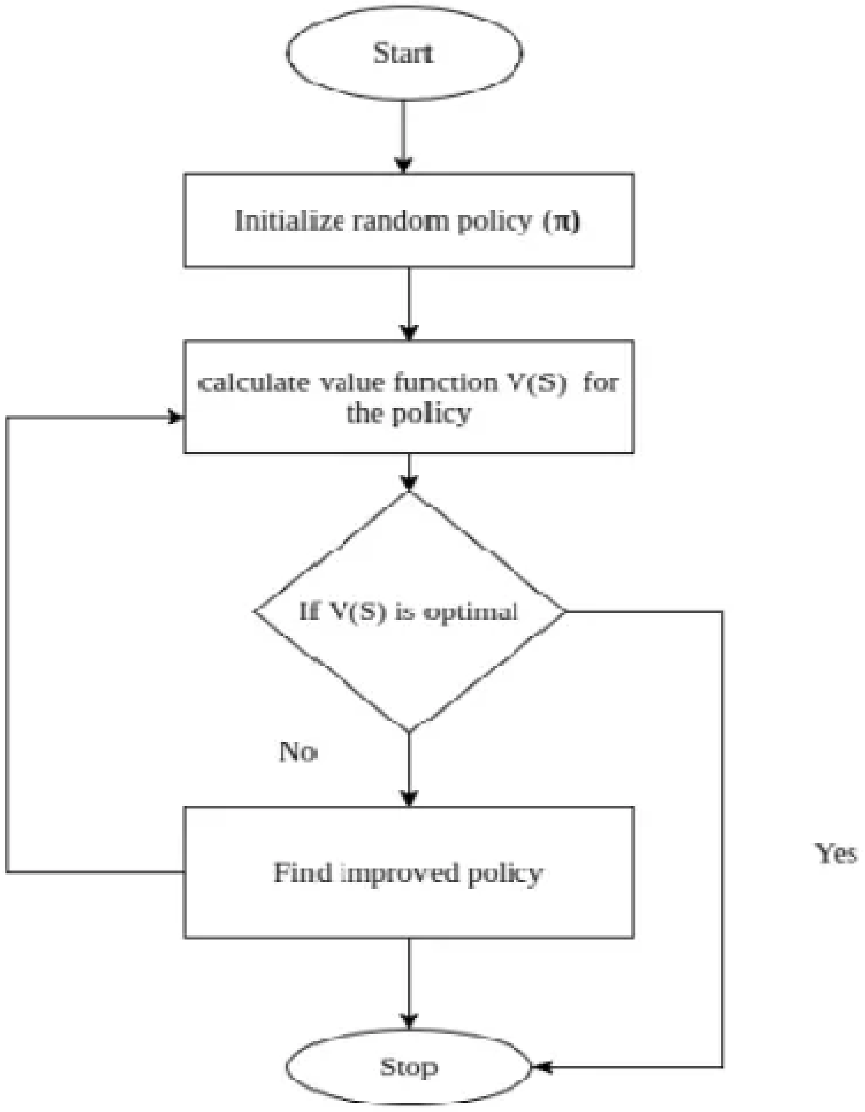

Start
Initialize random policy (π)
calculate value function V(S) for the policy
If V(S) is optimal
No
Yes
Find improved policy
Stop

Figure 4.6: Flow-chart representation of Policy Iteration

4.4 Problem Statement

As we have discussed about the environment earlier, it is very much clear that our goal is to reach to the terminal state without facing any obstacle.

Now, there is an agent which is going to interact with the environment to fulfill the objective of the experiment, i.e, to find not only **any possible path**, to find the **optimal path**. In the environment, our agent can go from one state to the next state in a total 8 specific directions and those are as follows :

N	North
S	South
E	East
W	West
NE	North East
NW	North West
SE	South East
SW	South West

We have a total 225 (15 $\times$ 15) number of states (0, 1, 2, ... , 224). We have placed some obstacles in some specific states in the environment which we have discussed earlier. We have also discussed about the reward distribution, which will come from the environment through interaction earlier. We have placed a "magic square" or "teleporter" which is a one way connector which we also have discussed earlier.

We have restricted "off grid move", i.e, the agent goes in a direction and make contact with grid wall. The environment is static which means the states where the obstacles are placed are not changable in any iteration.

$\Delta = \mid V(s) - V(s') \mid$ is the convergence parameter. In clear words, convergence in this iterative method means there will be a certain state (or iteration) after which if we iterate the policy, we will get the same values. This threshold is called θ and it is considered as constant. In this experiment, we assumed that θ is **1e-6** (10^-6). When in our experimentation $\Delta < \theta$ comes, we have reached to the convergence, otherwise the iterative evaluation will be continued. Through this iterative method, we can reach to the optimal value function and the policy before the convergence or untill the convergence reaches.

Further, we have designed the problem in a way that whatever may be the start point, after reaching to the optimal policy, from wherever we want to start except the terminal state, we will reach to the terminal state as well.

We have considered γ (Discount factor) to be 1.0 in our experimentation.

As expectation, from picking up a random policy, evaluating its every states' value, i.e, the value function and through iterative evaluation and improvement of the policy, we are going to see whether our agent will reach to the terminal state or not. As we are using Reinforcement Learning as a primary tool, it will be a positive thing for the agent to explore all the possibilities (epsilon-greedy algorithm) in the environment.

4.5 Results and Discussions

Implementing the "Policy Iteration" algorithm in the specified problem, we have seen the observations and those are as follows:

1. The initial value function $V(s)$ before the iterative method start was a 15×15 matrix with 0 in every state. After 83636 sweeps of state space for Policy evaluation, the convergence parameter meets with the convergence criterion. The final value function after convergence is shown in figure 4.7.

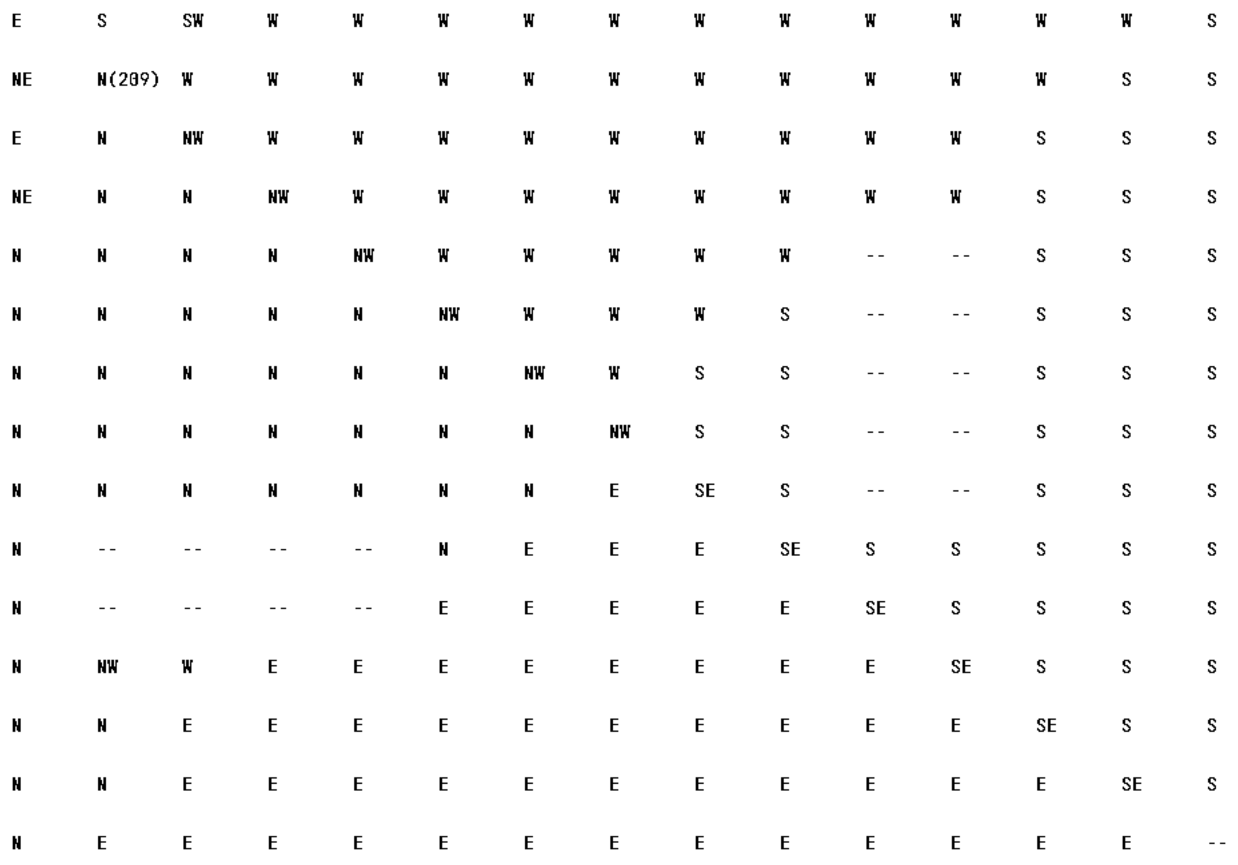
2. After 412 sweeps of state space for policy improvement and 3090 sweeps of state space for policy evaluation we get the optimal value function as well as the optimal policy shown in the figure 4.8 and 4.9 respectively.

```
-121.22 -111.49 -114.42 -130.47 -140.20 -146.02 -148.96 -149.80 -149.26 -148.15 -147.20 -146.91 -147.40 -148.40 -149.66

-122.94 -121.40 -118.45 -131.25 -140.04 -145.22 -147.70 -148.07 -146.99 -145.32 -143.97 -143.63 -144.36 -145.68 -146.90

-129.40 -121.67 -123.62 -134.21 -140.57 -144.41 -145.87 -145.21 -142.73 -139.41 -136.82 -136.35 -138.02 -140.47 -142.27

-132.71 -132.26 -133.56 -137.13 -141.03 -143.28 -143.54 -141.58 -137.21 -130.05 -124.21 -123.61 -128.21 -133.83 -136.50

-136.22 -136.01 -136.90 -138.65 -140.52 -141.50 -140.77 -137.70 -131.58 -121.45 0.00    0.00    -119.23 -127.19 -130.72

-136.22 -135.88 -136.33 -137.34 -138.41 -138.81 -137.66 -134.08 -127.16 -115.40 0.00    0.00    -112.93 -122.10 -125.98

-132.74 -132.25 -132.42 -133.20 -134.30 -135.00 -134.29 -131.11 -124.40 -113.60 0.00    0.00    -110.92 -119.03 -122.84

-126.04 -125.29 -125.09 -125.92 -127.79 -129.92 -130.79 -128.99 -123.47 -113.35 0.00    0.00    -110.38 -117.88 -121.22

-116.61 -115.30 -113.89 -114.49 -118.22 -123.25 -127.73 -127.99 -124.17 -116.18 0.00    0.00    -112.32 -117.86 -120.48

-109.44 0.00    0.00    0.00    0.00    -118.41 -126.18 -128.30 -126.29 -120.77 -115.41 -113.37 -115.12 -117.95 -119.50

-109.31 0.00    0.00    0.00    0.00    -118.82 -127.05 -129.90 -129.07 -125.95 -122.09 -118.94 -117.10 -116.45 -116.76

-116.21 -114.91 -113.59 -114.37 -118.51 -124.31 -130.09 -132.39 -131.84 -129.13 -125.01 -120.13 -115.29 -111.57 -110.32

-125.27 -124.53 -124.41 -125.49 -127.93 -131.18 -133.93 -135.11 -134.14 -131.04 -125.93 -118.84 -109.85 -101.43 -97.38

-131.62 -131.07 -131.18 -132.12 -133.78 -135.71 -137.18 -137.43 -135.91 -132.25 -126.12 -116.90 -103.48 -83.13  -72.44

-134.98 -134.24 -134.36 -135.20 -136.58 -138.06 -139.07 -138.94 -137.14 -133.19 -126.54 -116.25 -100.20 -73.25  0.00
```

Figure 4.7: Value function after convergence

```
-2.00   -1.00   -1.00   -2.00   -3.00   -4.00   -5.00   -6.00   -7.00   -8.00   -9.00   -10.00  -11.00  -12.00  -13.00

-2.00   -2.00   -1.00   -2.00   -3.00   -4.00   -5.00   -6.00   -7.00   -8.00   -9.00   -10.00  -11.00  -12.00  -12.00

-2.00   -1.00   -1.00   -2.00   -3.00   -4.00   -5.00   -6.00   -7.00   -8.00   -9.00   -10.00  -11.00  -11.00  -11.00

-2.00   -2.00   -2.00   -2.00   -3.00   -4.00   -5.00   -6.00   -7.00   -8.00   -9.00   -10.00  -10.00  -10.00  -10.00

-3.00   -3.00   -3.00   -3.00   -3.00   -4.00   -5.00   -6.00   -7.00   -8.00   0.00    0.00    -9.00   -9.00   -9.00

-4.00   -4.00   -4.00   -4.00   -4.00   -4.00   -5.00   -6.00   -7.00   -8.00   0.00    0.00    -8.00   -8.00   -8.00

-5.00   -5.00   -5.00   -5.00   -5.00   -5.00   -5.00   -6.00   -7.00   -7.00   0.00    0.00    -7.00   -7.00   -7.00

-6.00   -6.00   -6.00   -6.00   -6.00   -6.00   -6.00   -6.00   -6.00   -6.00   0.00    0.00    -6.00   -6.00   -6.00

-7.00   -7.00   -7.00   -7.00   -7.00   -7.00   -7.00   -6.00   -5.00   -5.00   0.00    0.00    -5.00   -5.00   -5.00

-8.00   0.00    0.00    0.00    0.00    -8.00   -7.00   -6.00   -5.00   -4.00   -4.00   -4.00   -4.00   -4.00   -4.00

-9.00   0.00    0.00    0.00    0.00    -8.00   -7.00   -6.00   -5.00   -4.00   -3.00   -3.00   -3.00   -3.00   -3.00

-10.00  -10.00  -11.00  -10.00  -9.00   -8.00   -7.00   -6.00   -5.00   -4.00   -3.00   -2.00   -2.00   -2.00   -2.00

-11.00  -11.00  -11.00  -10.00  -9.00   -8.00   -7.00   -6.00   -5.00   -4.00   -3.00   -2.00   -1.00   -1.00   -1.00

-12.00  -12.00  -11.00  -10.00  -9.00   -8.00   -7.00   -6.00   -5.00   -4.00   -3.00   -2.00   -1.00   0.00    0.00
```

Figure 4.8: The optimal value function



```
E       S       SW      W       W       W       W       W       W       W       W       W       W       W       S

NE      N(209)  W       W       W       W       W       W       W       W       W       W       W       S       S

E       N       NW      W       W       W       W       W       W       W       W       W       S       S       S

NE      N       N       NW      W       W       W       W       W       W       W       W       S       S       S

N       N       N       N       NW      W       W       W       W       W       --      --      S       S       S

N       N       N       N       N       NW      W       W       W       S       --      --      S       S       S

N       N       N       N       N       N       NW      W       S       S       --      --      S       S       S

N       N       N       N       N       N       N       NW      S       S       --      --      S       S       S

N       N       N       N       N       N       N       E       SE      S       --      --      S       S       S

N       --      --      --      --      N       E       E       E       SE      S       S       S       S       S

N       --      --      --      --      E       E       E       E       E       SE      S       S       S       S

N       NW      W       E       E       E       E       E       E       E       E       SE      S       S       S

N       N       E       E       E       E       E       E       E       E       E       E       SE      S       S

N       N       E       E       E       E       E       E       E       E       E       E       E       SE      S

N       E       E       E       E       E       E       E       E       E       E       E       E       E       --
```

Figure 4.9: The optimal policy to reach to the terminal state

Chapter 5

Conclusions

Based on the foregoing analysis, program development, experimentations and results on **Path Planning in a Simple Grid in Static Environment using Policy Iteration**, we can conclude the following observations and those are as follows :

1. As the Policy evaluation and improvement are taking place one after another (two consecutive steps) for Policy iteration in every iteration, the computational time requires more but we can get some strong results to find our optimal policy.

2. The computational time that the PyCharm IDE took for the experiment was around 12-15 seconds.

3. As our agent explored every possibility in the complex static environment, it not only gives the optimal path from 0^{th} state, it gives the optimal path from every possible state in the 15×15 grid except the terminal state to the terminal state.

4. Apart from path planning, The policy iteration algorithm can be used in more or less every domain where there is a situation on which a decision has to be taken.

5. Python and PyCharm (IDE) are easier to deal with and Python is very much preferred in these domains, broadly in machine learning.

6. Policy iteration can be used in dynamic environment as well.

6 References

[1] C. Dimitrakakis and M. G. Lagoudakis, "Rollout sampling approximate policy iteration," *Machine Learning*, vol. 72, pp. 157–171, 2008.

[2] J. Suh and T. Tanaka, "Encrypted value iteration and temporal difference learning over leveled homomorphic encryption," in *2021 American control conference (ACC)*, pp. 2555–2561, IEEE, 2021.

[3] D. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic programming.* Athena Scientific, 1996.

[4] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Playing games with scenario-and resource-aware sdf graphs through policy iteration," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 194–199, IEEE, 2012.

[5] D. Bertsekas, "Multiagent reinforcement learning: Rollout and policy iteration," *IEEE/CAA Journal of Automatica Sinica*, vol. 8, no. 2, pp. 249–272, 2021.

[6] S. Shivdikar and J. Nirmal, "Path planning using reinforcement learning: A policy iteration approach," *arXiv preprint arXiv:2303.07535*, 2023.

[7] X. Zhang, Y. Xu, Y. Chen, and D. Li, "Path planning model based on constrained policy iteration," in *2022 41st Chinese Control Conference (CCC)*, pp. 5518–5523, IEEE, 2022.

[8] R. S. Sutton, "Reinforcement learning: Past, present and future," in *Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL'98 Canberra, Australia, November 24–27, 1998 Selected Papers 2*, pp. 195–197, Springer, 1999.

[9] K. Zhu and T. Zhang, "Deep reinforcement learning based mobile robot navigation: A review," *Tsinghua Science and Technology*, vol. 26, no. 5, pp. 674–691, 2021.

[10] I. A. Zamfirache, R.-E. Precup, R.-C. Roman, and E. M. Petriu, "Policy iteration reinforcement learning-based control using a grey wolf optimizer algorithm," *Information Sciences*, vol. 585, pp. 162–175, 2022.

[11] X. Xu, D. Hu, and X. Lu, "Kernel-based least squares policy iteration for reinforcement learning," *IEEE transactions on neural networks*, vol. 18, no. 4, pp. 973–992, 2007.

[12] N. Sariff and N. Buniyamin, "An overview of autonomous mobile robot path planning algorithms," in *2006 4th student conference on research and development*, pp. 183–188, IEEE, 2006.

[13] L. Lee, E. Parisotto, D. S. Chaplot, E. Xing, and R. Salakhutdinov, "Gated path planning networks," in *International Conference on Machine Learning*, pp. 2947–2955, PMLR, 2018.

[14] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling, "Learning policies for partially observable environments: Scaling up," in *Machine Learning Proceedings 1995*, pp. 362–370, Elsevier, 1995.

[15] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The international journal of robotics research*, vol. 5, no. 1, pp. 90–98, 1986.

[16] C. Prigent, L. Cudennec, A. Costan, and G. Antoniu, "A methodology to build decision analysis tools applied to distributed reinforcement learning," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1053–1062, IEEE, 2022.

[17] K. Blekas and K. Vlachos, "Rl-based path planning for an over-actuated floating vehicle under disturbances," *Robotics and Autonomous Systems*, vol. 101, pp. 93–102, 2018.

[18] A. Winnicki and R. Srikant, "A new policy iteration algorithm for reinforcement learning in zero-sum markov games," *arXiv preprint arXiv:2303.09716*, 2023.

[19] D. P. Bertsekas, "Robust shortest path planning and semicontractive dynamic programming," *Naval Research Logistics (NRL)*, vol. 66, no. 1, pp. 15–37, 2019.

[20] Y. Yang, L. Juntao, and P. Lingling, "Multi-robot path planning based on a deep reinforcement learning dqn algorithm," *CAAI Transactions on Intelligence Technology*, vol. 5, no. 3, pp. 177–183, 2020.

[21] L. Sun, J. Yan, and W. Qin, "Path planning for multiple agents in an unknown environment using soft actor critic and curriculum learning," *Computer Animation and Virtual Worlds*, vol. 34, no. 1, p. e2113, 2023.

[22] A. Ianenko, A. Artamonov, G. Sarapulov, A. Safaraleev, S. Bogomolov, and D.-k. Noh, "Coverage path planning with proximal policy optimization in

a grid-based environment," in *2020 59th IEEE Conference on Decision and Control (CDC)*, pp. 4099–4104, IEEE, 2020.

[23] S. Eiffert, H. Kong, N. Pirmarzdashti, and S. Sukkarieh, "Path planning in dynamic environments using generative rnns and monte carlo tree search," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 10263–10269, IEEE, 2020.