UNIVERSITY OF TECHNOLOGY OF COMPIÈGNE

MASTER OF SCIENCE

Graph Neural Networks and Reinforcement Learning for Multi-Robot Motion Planning

---

Author:
Hussein LEZZAIK

Supervisors:
Gennaro Notomista
Claudio Pacchierotti

A thesis submitted in fulfillment of the requirements for Master of Science in Robotics

at

University of Technology of Compiègne

RAINBOW Team
Inria

September 3rd, 2021

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan M. Turing

To my Family…

UNIVERSITY OF TECHNOLOGY OF COMPIÈGNE

Abstract

RAINBOW Team
Computer Science Department

Master of Science

Graph Neural Networks and Reinforcement Learning for Multi-Robot Motion Planning

by Hussein LEZZAIK

The emergence of relatively cheap sensing and actuation nodes, capable of short-range communications and local decision-making, has raised a number of new system-level questions concerning how such systems should be coordinated, controlled, and scaled to diverse-large networks. This thesis presents a data-driven decentralized solution to this dilemma. The proposed solution is achieved by designing a deep learning model from scratch, powered by Graph Neural Networks and Reinforcement Learning. The proposed methodology is simulated and experimentally validated on the task of consensus, with diverse communication network topologies and scalable networks.

Keywords: Consensus, Decentralized Controllers, Graph Neural Networks, Reinforcement Learning.

Acknowledgements

First and foremost, I am deeply honored of having had the opportunity of working with my advisor Claudio Pacchierotti. His amazing enthusiasm pushed me to work hard during my stay at Inria, and got me to escape my correlation-causation dilemma with respect to what can be accomplished in a short period of time. Not only did he seed in me how to conduct fundamental research, but also taught me how to be productive and climb the ladder of success in diverse orthogonal dimensions. I will never forget how he was always there when I needed him, from weekends to late nights and even during vacations! Thanks to his consistent advice and unique perspective, I feel that I experienced one of the most profound personal growth which will be reflected on my entire future career.

I've also been very fortunate to work with and learn from Gennaro Notomista, whom I've developed a lot of respect and admiration for and who have become my role model in many respects. His passion to robotics, strong theoretical background and top-notch software skills has shaped my thinking and engineering philosophy in a way that will forever impact my work. I really enjoyed our deep weekly discussions on designing neural networks from scratch, representation learning, latest advances in AI, and best of all software engineering. All of this work wouldn't have been possible without him.

I'm also very grateful for the support I received by the RAINBOW team once I joined and how welcoming they are. I got to interact with some of the greatest scientists and engineers during my professional career, and had several memorable debates on topics like the interpretability of deep learning. I loved the cooperative culture at Inria, the passion to discover new ideas, and discipline to push technology forward. I hope we get to collaborate again in the future.

And last but not least, my most profound thank you goes to my family, my mother, my father, my brother and my sister, who, from too far away—this time, from the other side of the Atlantic Ocean—have always been there, supporting all the choices I made in life, and continuously encouraging me to dream bigger and achieve more. None of my accomplishments would have been possible without you. No words will ever be enough: *I love you*

Contents

List of Figures

List of Abbreviations

ROS	**R**obot **O**perating **S**ystem
DL	**D**eep **L**earning
SL	**S**upervised **L**earning
RL	**R**einforcement **L**earning
IL	**I**mitation **L**earning
AI	**A**rtificial **I**ntelligence
CNN	**C**onvolutional **N**eural **N**etworks
RNN	**R**ecurrent **N**eural **N**etworks
GNN	**G**raph **N**eural **N**etworks
DNN	**D**eep **N**eural **N**etworks
PPO	**P**roximal **P**olicy **O**ptimization
DQN	**D**eep **Q** **N**etworks
i.i.d	**i**ndependet **i**dentical **d**istribution
MDP	**M**arkov **D**ecision **P**rocess
DPL	**D**irect **P**olicy **L**earning
IRL	**I**nverse **R**einforcement **L**earning
MLP	**M**ulti **L**ayer **P**erceptron
SDG	**S**tochastic **G**radient **D**escent

Chapter 1

Introduction

1.1 Motivation

Large scale swarms are composed of multiple agents that collaborate to accomplish a task. In the future, these systems could be deployed to, for example, provide on-demand wireless networks, perform rapid environmental mapping, search after natural disasters, or enable sensor coverage in cluttered and communications denied environments [34].

Effective communication is key to successful, decentralized, multi-robot path planning. Yet, it is far from obvious what information is crucial to the task at hand, and how and when it must be shared among robots [13] [21] [26] [7] [22] [14] [40].

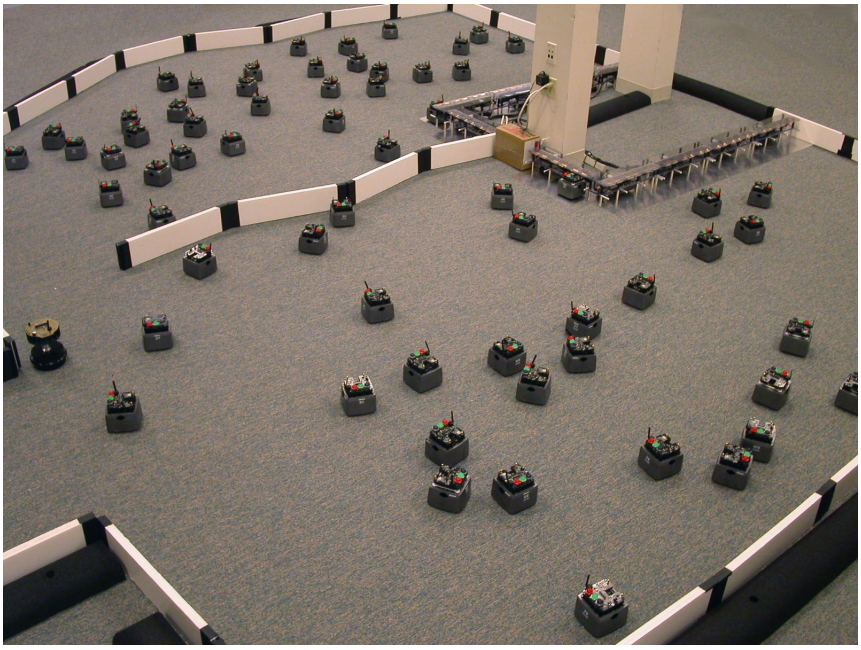

FIGURE 1.1: Swarm Systems, from [15]

Of particular interest to us is the capability of learning-based methods to handle high-dimensional joint state-space representations, useful when planning for large-scale collective robotic systems, by offloading the online computational burden to an offline learning procedure. The fact that each robot must be able to accumulate information from other robots in its neighborhood is key to this learning procedure. From the point of view of an individual robot, its local decision-making system is incomplete, since other agent's unobservable states affect future values. The manner in which information is shared is crucial to the system's performance, yet is not well addressed by current machine learning approaches[11][23].

Graph Neural Networks (GNNs) [9] [3] promise to overcome this deficiency[13]. They capture the relational aspect of robot communication and coordination by modeling the collective robot system as a graph: each robot is a node, and edges represent communication links. Although GNNs have been applied to a number of problem domains, including molecular biology, quantum chemistry, and simulation engines, they have only very recently

been considered within the multi-robot domain, for applications of flocking and formation control .

We consider the problem of finding distributed-decentralized controllers for large networks of mobile robots with interacting dynamics and sparsely available communications. Our approach is to learn local controllers that require only local information and communications at test time by imitating the policy of centralized controllers using global information at training time.

We apply this approach to the problem of consensus to demonstrate performance on communication graphs that change as the robots move. We examine how different communication topologies and bigger networks impact the performance of our approach compared to centralized ones.

1.2 Organization

The rest of this chapter provides background material on graph methods for multi-agent systems, and ends with a brief overview of the contributions of this work. Chapter 2 provides a brief introduction to relevant background literature that our work is based on. Chapter 3 develops the decentralized GNN model for consensus, architecture details and methodology that we used to collect data and train our model. Chapter 4 demonstrates different experiments to evaluate the performance of our model which provides promising results for this framework. Chapter 5 concludes this work and briefly discusses possible future research directions for training our decentralized GNN approach using reinforcement learning.

1.3 Graph Methods for Multi-Robots

Classical approaches to multi-robot path planning can generally be described as either centralized or decentralized:

1.3.1 Centralized Algorithms

Centralized approaches are facilitated by a planning unit that monitors all robot's positions and desired destinations, and returns a coordinated plan of trajectories (or way-points) for all the robots in the system. These plans are communicated to the respective robots, which use them for real-time on-board control of their navigation behavior. Coupled centralized approaches, which consider the joint configuration space of all involved robots, have the advantage of producing optimal and complete plans, yet tend to be computationally very expensive. Indeed, solving for optimality is NP-hard [39], and although significant progress has been made towards alleviating the computational load [6], these approaches still scale poorly in environments with a high number of potential path conflicts.

1.3.2 Decentralized Algorithms

Decentralized approaches provide an attractive alternative to centralized approaches, firstly, because they reduce the computational overhead, and secondly, because they relax the dependence on centralized units. This body of work considers the generation of collision-free paths for individual robots that cooperate only with immediate neighbors [4], or with no other robots at all [35]. In the latter case, coordination is reduced to the problem of reciprocally avoiding other robots (and obstacles), and can generally be solved without the use of communication. Yet, by taking purely local objectives into account, global objectives (such as path efficiency) cannot be explicitly optimized. In the former case, it has been shown that monotonic cost reduction of global objectives can be achieved. This feat, however, relies on strong assumptions (e.g., problem convexity and invariance of communication graph) that can generally not be guaranteed in real robot systems [25].

1.3.3 Learning-Based Algorithms

Learning-based methods have proven effective at designing robot control policies for an increasing number of tasks [25]. The application of learning-based methods to multi-robot motion planning has attracted particular attention due to their capability of handling high-dimensional joint state-space representations, by offloading the online computational burden to an offline learning procedure. The work in [5] proposes a decentralized multi-agent collision avoidance algorithm based on deep reinforcement learning. Their results show that significant improvement in the quality of the path (i.e., time to reach the goal) can be achieved with respect to current benchmark algorithms (e.g., ORCA [35]). Also in recent work, Sartoretti et al. [2] propose a hybrid learning-based method called PRIMAL for multi-agent path-finding that uses both imitation learning (based on an expert algorithm) and multi-agent reinforce- ment learning. It is note-worthy that none of the aforementioned learning-based approaches consider inter-robot communication, and thus, do not exploit the scalability benefits of fully decentralized approaches. Learning what, how, and when to communicate is key to this aim.

1.4 Contributions

This work is inspired by the GNN decentralized models developed in [13] [34] for multi-robot motion planning. Our contributions can be summarized as follows:

1. We decouple a centralized expert consensus algorithm into a decentralized controller.
2. We propose a deep learning framework powered by graph neural nets and reinforcement learning in a imitation learning framework.
3. We validate the performance of our model and provide supporting results that our model works for different communication graph topologies, and size of network.

Chapter 2

Background

In this chapter, we will provide a brief theoretical background of all the main materials that our work was based on. First, we will talk about consensus for multi-agents, then we'll go into deep learning and dive a little deep into what supervised learning actually is. Then we will go into graph neural networks and it's most important applications, and then finally explain what reinforcement learning is, and the most important types of imitation learning.

2.1 Positional Control of Unicycle

One of the important models used in control for mobile robotics is the Hemingwayian cycle, which treats the controlled robot as a single point. Now since our approach is built and trained on two-wheeled differential drive, here we shall provide the important theoretical background on the transformation from unicycle to a two wheeled differential drive.

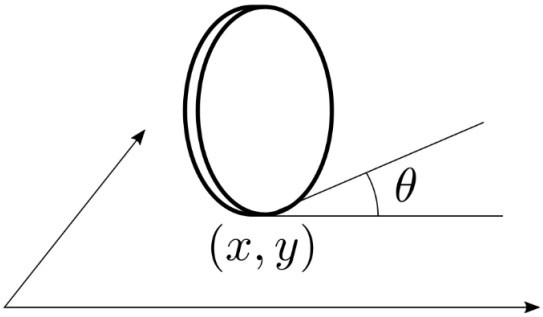

FIGURE 2.1: Rolling Coin, from [20]

Unicycles are used to model a large variety of mobile robotic systems: ground, marine, and even aerial robots are very often abstracted using a rolling coin. The dynamic model of the unicycle is given by:

$$\begin{cases} \dot{x} = v\cos(\theta) \\ \dot{y} = v\sin(\theta) \\ \dot{\theta} = w \end{cases} \tag{2.1}$$

Where:

- x, y are the coordinates of the position of the system in the plane
- θ is its orientation
- v, w are the linear and angular velocity control inputs

An effective way of designing controllers for unicycle models consists in controlling a point p at a distance l in front of the unicycle depicted in the following figure:

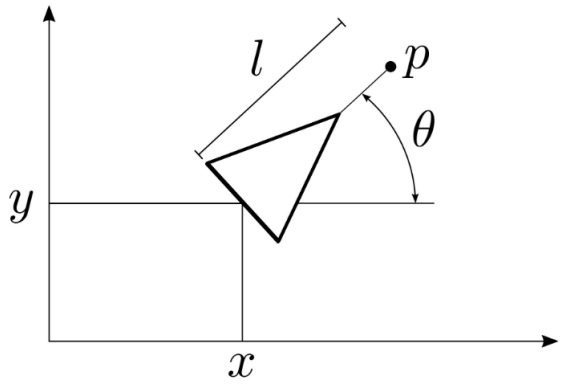

FIGURE 2.2: Point P at a distance l from center of unicycle, from [20]

In fact, it is easy to see that:

$$\dot{p} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & l \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \tag{2.2}$$

So, once a controller for $\dot{p}$ is designed, the inputs v and w to the unicycle can be easily computed as:

$$\begin{bmatrix} v \\ w \end{bmatrix} = L^{-1}(l)R^{T}(\theta)\dot{p} \tag{2.3}$$

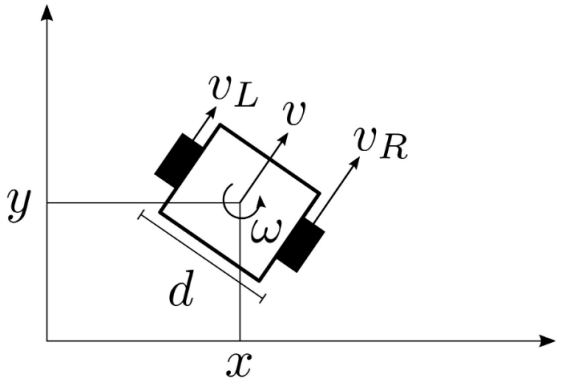

FIGURE 2.3: Differential Drive Robot, from [20]

As in practice robots are not rolling coins, but rather have wheels, the values v and w can be turned into wheel speed in the case of differential-drive robots (shown in the figure above) as follows:

$$\begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ -1/2d & 1/2d \end{bmatrix} \begin{bmatrix} v_L \\ v_R \end{bmatrix} \Leftrightarrow \begin{bmatrix} v_L \\ v_R \end{bmatrix} = D^{-1} \begin{bmatrix} v \\ w \end{bmatrix} \tag{2.4}$$

This equation shall be used for the consensus control algorithm in Chapter 3, to map the control output from a unicycle model to a two wheeled differential drive.

2.2 Consensus Control

In the cooperative consensus control, robots will share their information about positions or speeds with each other. This information may lead to achieve the common group objective, which in our case of study is to meet in at a target location.

Consensus is an important problem in cooperative control, where several agents have to be synchronized to a common value. In the case of multi-robots systems, it can be interesting for the robots to achieve a consensus on positions, like a rendez-vous point, to reach an objective at the same instant. Consensus means having agents come to a global agreement on a state value.

The significance of the agreement protocol is twofold. On one hand, agreement has a close relation to a host of multi-agent problems such as flocking, rendez-vous, swarming, attitude alignment, and distributed estimation. On the other hand, this protocol provides a concise formalism for examining means by which the network topology dictates properties of the dynamic process evolving over it [17].

The agreement protocol involves n dynamic units, labeled $[1, 2, ..., n]$ interconnected via relative information-exchange links. The rate of change of each unit's state is assumed to be governed by the sum of its relative states with respect to a subset of other (neighboring) units. An example of the agreement protocol with three first-order dynamic units is shown in figure below:

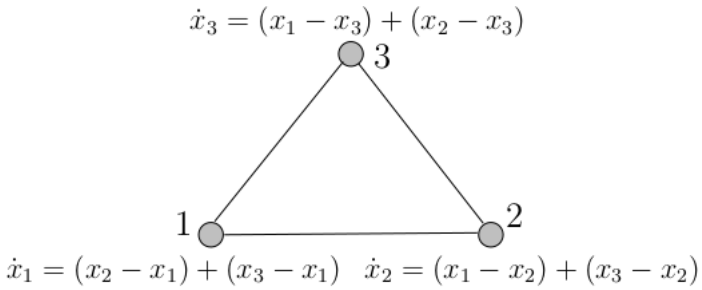

FIGURE 2.4: Agreement protocol over a triangle, from [1]

Denoting the scalar state of unit i as $x_i \in R$, one then has:

$$\dot{x}_i(t) = \sum_{j \in N(i)} (x_j(t) - x_i(t)), \quad i = 1, 2, ..., n \tag{2.5}$$

Where $N(i)$ is the set of units "adjacent to" or neighboring, unit i in the network. When the adopted notion of adjacency is symmetric, the overall system can be represented by:

$$\dot{x}(t) = -L(G)x(t) \tag{2.6}$$

where the positive semi-definite matrix $L(G)$ is the Laplacian of the agent's interaction network G and $x(t) = [x_1(\mathrm{t}), ..., x_n(\mathrm{t})]^T \in R^n$. We refer to 2.6 as the agreement dynamics.

2.3 Deep Learning

Before moving on to the explanation of our deep learning based-approach for multi-robot motion planning, it is paramount here to introduce general knowledge on machine learning. In 1950, after the first computers were developed, Alan Turing devised the concept of Artificial Intelligence (*AI*) and gave it a definition. The revolution was when he asked himself: "Can Machines Think?". He devised a test, called Turing Test, which was intended to distinguish machines from human beings after going through a series of questions and answers. If a computer is able to pass this test, it is considered to be "Intelligent" like a human. With this, it ignited a spark in researchers so in 1956, AI got recognition as an academic subject.

AI was considered as an intelligent agent that perceives the variable environment, learn from it and makes decision based on the knowledge base (*KB*). It was a dream for every researcher to apply it to different areas of research for over 60 years, but it was astonishingly developed in the 21st century. The reason for late development was not the incompetence of the researchers but the limited computational power for such complex problem. With increased capability of computational power, AI has flourished into noticeably popularization of the internet, big data handling, interconnection and fusion of data and many more accomplishments worth discussion.

In this time, AI is part of studies in research areas like reinforcement learning, knowledge reasoning and representation, automated reasoning and most importantly deep learning. The attractions towards machine learning is because of absence of the ability to write complex programs for unstructured problems and rather recognizing patterns and making predictions from the data available. So it is expected that machine learning would find its way in the field of robotics as robots are becoming an important part of current industrial use. Traditionally, machine learning is categorized into three main areas as shown in Figure 2.5.

This chapter provides a brief technical background on deep learning for our work. For a more thorough and slower-paced introduction we recommend the Deep Learning book from Goodfellow et al. [8].

2.3.1 Unsupervised Learning

In this class, the agent is not instructed about the input or output, rather a huge size of data is fed in (unlike supervised learning). So, no label is given to the output. The agent focuses on observing the universal data set to find the patterns, structure and/or features embedded with the data. A most common example of such problem is the news webpage in which various news of similar category are accumulated into one link like sports or weather, etc. These are basically the clusters. With a bird's eye view, the agent is able to recognize the apparent clusters then the agent classifies the similar clusters into the corresponding one for example, you won't find sports in the weather cluster.

2.3.2 Supervised Learning

As the name suggests, the agent is given the data to observe. It is the most commonly used category of machine learning. In this kind of learning, the agent observes given data (the input to output pairs), learns the pattern and it devices a function that maps output from input. The agent is interested in finding the output with a precondition of knowing the input. It is further divided into two problems:

- **Classification Problem**: is the one in which for an input (in the form of an image for e.g.), the devised output is one of the discrete number of possible 'classes'.
- **Regression Problem**: is the one in which the output is the continuous stream of values based on changing scenarios.

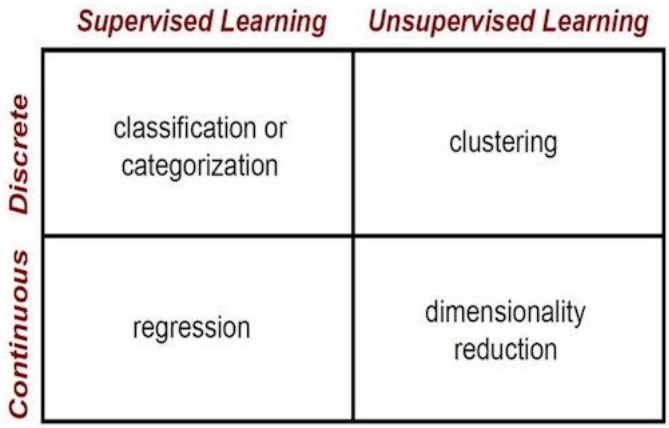

FIGURE 2.5: Supervised Learning vs Unsupervised Learning

Many practical problems can be formulated as requiring a computer to perform a mapping $f : X \to Y$, where X is an input space and Y is an output space. For instance, in visual recognition X could be the space of images and Y could be the interval $[0, 1]$ indicating the probability of a cat appearing somewhere in the image. Unfortunately, in many cases it is difficult to manually specify the function f by conventional means (e.g. it is unclear how one might write down a program that recognizes a cat). The supervised learning paradigm offers an alternative approach that takes advantage of the fact that it is often relatively easy to obtain examples $(x, y) \in X \times Y$ of the desired mapping. In our running example, this would correspond to collecting a dataset of images each labeled with the presence or absence of a cat, as annotated by humans.

The objective: Concretely, we assume a training dataset of n examples $[(x_1 , y_1), \ldots (x_n, y_n)]$ made up of independent and identically distributed $(i.i.d.)$ samples from a data generating distribution D; i.e. $(x_i, y_i) \sim D$ for all i.

We then think about learning the mapping $f : X \to Y$ by searching over a set of candidate functions and finding the one that is most consistent with the training examples. More precisely, we consider some particular class of functions F and choose a scalar-valued loss function $L(\hat{y}, y)$ that measures the disagreement between a predicted label $\hat{y}_i = f(x_i)$ for some $f \in F$ and a true label y_i . Our objective in learning is to find $f^* \in F$ that ideally satisfies:

$$f^* = \underset{f \in F}{\operatorname{argmin}} E_{(x,y)\sim D} L(f(x), y) \tag{2.7}$$

In other words, we seek a function f^* that minimizes the expected loss over the data generating distribution D. In practical applications, once we identify this function we can discard the original training data and only keep the learned function f^*, which we use to map elements of X to Y.

Unfortunately, the optimization problem above is intractable because we do not have access to all possible elements of D and therefore cannot evaluate the expectation or simplify it analytically without making unrealistically strong assumptions about the form of D, L or f . However, under the $i.i.d.$ assumption we can approximate the expected loss in Equation 2.7 above with sampling by averaging the loss over the available training data:

$$f^* \approx \underset{f \in F}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i) \tag{2.8}$$

In other words we optimize the loss only over the available training examples, but the hope is that this is a good proxy objective for the actual objective in Equation 2.7

Summary: In supervised learning we are given a dataset of n datapoints $[(x_1, y_1), \ldots (x_n, y_n)]$ where $(x_i, y_i) \in \mathrm{X} \times \mathrm{Y}$ and we identify three quantities to formalize the problem:

- The search space of functions F, where each $f \in F$ maps X to Y.
- The scalar-valued loss function $L(\hat{y}, y)$ that evaluates the mismatch between a true label y and a predicted label $\hat{y}_i = f(x_i)$.
- The scalar-valued regularization loss $R(f)$ that measures the complexity of a mapping.

We will stop with here as providing a complete background on deep learning is beyond the scope of this thesis.

2.4 Graph Neural Networks

Graphs are a ubiquitous data structure and a universal language for describing complex systems[9] . In the most general view, a graph is simply a collection of objects (i.e., nodes), along with a set of interactions (i.e., edges) between pairs of these objects. For example, to encode a social network as a graph we might use nodes to represent individuals and use edges to represent that two individuals are friends (Figure 2.6). In the biological domain we could use the nodes in a graph to represent proteins, and use the edges to represent various biological interactions, such as kinetic interactions between proteins.

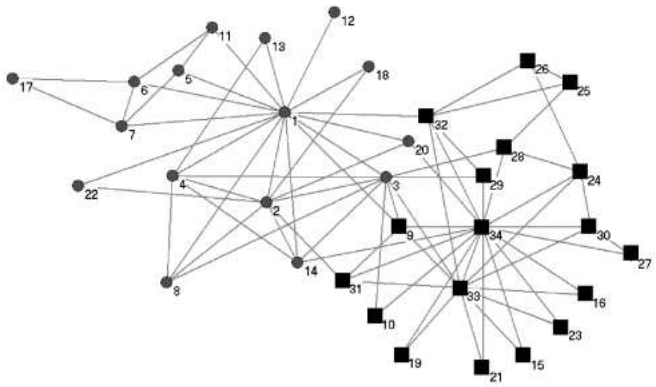

FIGURE 2.6: The famous Zachary Karate Club Network represents the friendship relationships between members of a karate club studied by Wayne W. Zachary from 1970 to 1972. From [9]

The power of the graph formalism lies both in its focus on relationships between points (rather than the properties of individual points), as well as in its generality. The same graph formalism can be used to represent social networks, interactions between drugs and proteins, the interactions between atoms in a molecule, or the connections between terminals in a telecommunications network—to name just a few examples.

Graphs do more than just provide an elegant theoretical framework, how- ever. They offer a mathematical foundation that we can build upon to analyze, understand, and learn from real-world complex systems. In the last twenty-five years, there has been a dramatic increase in the quantity and quality of graph- structured data that is available to researchers. With the advent of large-scale social networking platforms, massive scientific initiatives to model the interac- tome, food webs, databases of molecule graph structures, and billions

of inter- connected web-enabled devices, there is no shortage of meaningful graph data for researchers to analyze.

2.4.1 What is a Graph?

Before we discuss machine learning on graphs, it is necessary to give a bit more formal description of what exactly we mean by "graph data". Formally, a graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes. We denote an edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$. In many cases we will be concerned only with simple graphs, where there is at most one edge between each pair of nodes, no edges between a node and itself, and where the edges are all undirected, i.e., $(u, v) \in E \leftrightarrow (v, u) \in E$.

A convenient way to represent graphs is through an adjacency matrix $A \in R^{|V|\times|V|}$. To represent a graph with an adjacency matrix, we order the nodes in the graph so that every node indexes a particular row and column in the adjacency matrix. We can then represent the presence of edges as entries in this matrix: $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise. If the graph contains only undirected edges then A will be a symmetric matrix, but if the graph is directed (i.e., edge direction matters) then A will not necessarily be symmetric. Some graphs can also have weighted edges, where the entries in the adjacency matrix are arbitrary real-values rather than $[0, 1]$. For instance, a weighted edge in a protein-protein interaction graph might indicated the strength of the association between two proteins.

2.4.2 Machine Learning on Graphs

Machine learning is inherently a problem-driven discipline. We seek to build models that can learn from data in order to solve particular tasks, and machine learning models are often categorized according to the type of task they seek to solve: Is it a supervised task, where the goal is to predict a target output given an input datapoint? Is it an unsupervised task, where the goal is to infer patterns, such as clusters of points, in the data?

Machine learning with graphs is no different, but the usual categories of supervised and unsupervised are not necessarily the most informative or useful when it comes to graphs. In this section we provide a brief overview of the most important and well-studied machine learning tasks on graph data. As we will see, "supervised" problems are popular with graph data, but machine learning problems on graphs often blur the boundaries between the traditional machine learning categories.

2.4.2.1 Node Classification

Suppose we are given a large social network dataset with millions of users, but we know that a significant number of these users are actually bots. Identifying these bots could be important for many reasons: a company might not want to advertise to bots or bots may actually be in violation of the social network's terms of service. Manually examining every user to determine if they are a bot would be prohibitively expensive, so ideally we would like to have a model that could classify users as a bot (or not) given only a small number of manually labeled examples.

This is a classic example of node classification, where the goal is to predict the label y_u —which could be a type, category, or attribute—associated with all the nodes $u \in V$, when we are only given the true labels on a training set of nodes $V_{train} \in V$. Node classification is perhaps the most popular machine learning task on graph data, especially in recent years.

2.4.2.2 Relation Prediction

Node classification is useful for inferring information about a node based on its relationship with other nodes in the graph. But what about cases where we are missing this relationship information? What if we know only some of protein- protein interactions that are present in a given cell, but we want to make a good guess about the interactions we are missing? Can we use machine learning to infer the edges between nodes in a graph?

This task goes by many names, such as link prediction, graph completion, and relational inference, depending on the specific application domain.

The standard setup for relation prediction is that we are given a set of nodes V and an incomplete set of edges between these nodes $E_{train} \in E$. Our goal is to use this partial information to infer the missing edges $E \in E_{train}$. The complexity of this task is highly dependent on the type of graph data we are examining.

2.4.2.3 Clustering and Community Detection

Both node classification and relation prediction require inferring missing information about graph data, and in many ways, those two tasks are the graph analogues of supervised learning. Community detection, on the other hand, is the graph analogue of unsupervised clustering.

Suppose we have access to all the citation information in Google Scholar, and we make a collaboration graph that connects two researchers if they have co-authored a paper together. If we were to examine this network, would we expect to find a dense "hairball" where everyone is equally likely to collaborate with everyone else? It is more likely that the graph would segregate into differ- ent clusters of nodes, grouped together by research area, institution, or other demographic factors. In other words, we would expect this network—like many real-world networks—to exhibit a community structure, where nodes are much more likely to form edges with nodes that belong to the same community.

This is the general intuition underlying the task of community detection. The challenge of community detection is to infer latent community structures given only the input graph $G = (V, E)$. The many real-world applications of community detection include uncovering functional modules in genetic interaction networks and uncovering fraudulent groups of users in financial transaction networks.

2.4.2.4 Graph Classification, Regression, and Clustering

The final class of popular machine learning applications on graph data involve classification, regression, or clustering problems over entire graphs. For instance, given a graph representing the structure of a molecule, we might want to build a regression model that could predict that molecule's toxicity or solubility. Or, we might want to build a classification model to detect whether a computer program is malicious by analyzing a graph-based representation of its syntax and data flow. In these graph classification or regression applications, we seek to learn over graph data, but instead of making predictions over the individual components of a single graph (i.e., the nodes or the edges), we are instead given a dataset of multiple different graphs and our goal is to make independent predictions specific to each graph. In the related task of graph clustering, the goal is to learn an unsupervised measure of similarity between pairs of graphs.

2.4.3 Graph Neural Networks

In this section, we turn our focus to more complex encoder models. We will introduce the graph neural network (GNN) formalism, which is a general framework for defining deep

neural networks on graph data. The key idea is that we want to generate representations of nodes that actually depend on the structure of the graph, as well as any feature information we might have.

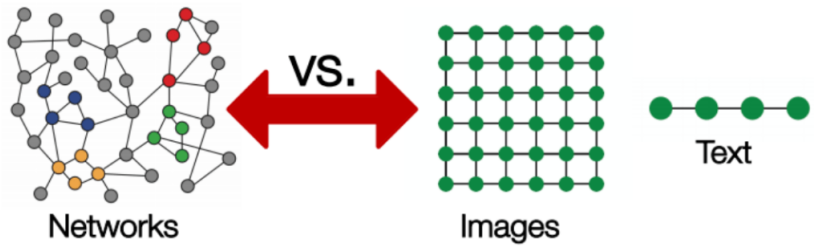

FIGURE 2.7: Graph Networks vs Images/Text, from [16]

The primary challenge in developing complex encoders for graph-structured data is that our usual deep learning toolbox does not apply. For example, convolutional neural networks (CNNs) are well-defined only over grid-structured inputs (e.g., images), while recurrent neural networks (RNNs) are well-defined only over sequences (e.g., text). To define a deep neural network over general graphs, we need to define a new kind of deep learning architecture.

We point the reader to the following resources for diving deeper: [17], [9], [3].

2.4.3.1 Permutation Invariance and Equivariance

One reasonable idea for defining a deep neural network over graphs would be to simply use the adjacency matrix as input to a deep neural network. For example, to generate an embedding of an entire graph we could simply flatten the adjacency matrix and feed the result to a multi-layer perceptron (MLP):

$$Z_G = MLP(A[1] \oplus A[2] \oplus ... \oplus A[|V|]) \tag{2.9}$$

where $A[i] \in R^{|V|}$ denotes a row of the adjacency matrix and we use $\oplus$ to denote vector concatenation. The issue with this approach is that it depends on the arbitrary ordering of nodes that we used in the adjacency matrix. In other words, such a model is not permutation invariant, and a key desideratum for designing neural networks over graphs is that they should permutation invariant (or equivariant) [9].

Ensuring invariance or equivariance is a key challenge when we are learning over graphs.

2.4.3.2 Basics of Deep Learning for Graphs

In graph theory, we implement the concept of Node Embedding. It means mapping nodes to a d- dimensional embedding space (low dimensional space rather than the actual dimension of the graph), so that similar nodes in the graph are embedded close to each other.

Our goal is to map nodes so that similarity in the embedding space approximates similarity in the network. Let's define u and v as two nodes in a graph. x_u and x_v are two feature vectors. Now we'll define the encoder function Enc(u) and Enc(v), which convert the feature vectors to z_u and z_v.

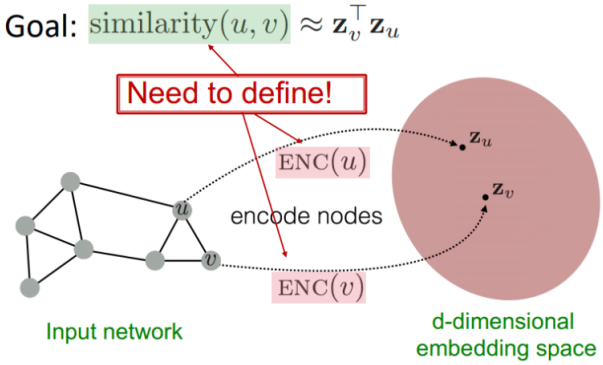

FIGURE 2.8: Similarity Function, from [16]

The encoder function should be able to perform :

- Locality (local network neighborhoods)
- Aggregate information
- Stacking multiple layers (computation)

Locality information can be achieved by using a computational graph. As shown in the graph below, *i* is the red node where we see how this node is connected to its neighbors and those neighbors' neighbors. We'll see all the possible connections, and form a computation graph.

By doing this, we're capturing the structure, and also borrowing feature information at the same time.

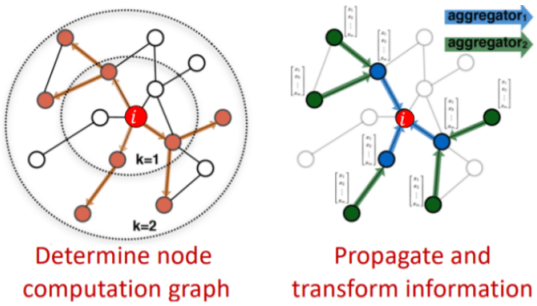

FIGURE 2.9: Neighborhood Exploration, from [16]

Once the locality information preserves the computational graph, we start aggregating. This is basically done using neural networks.

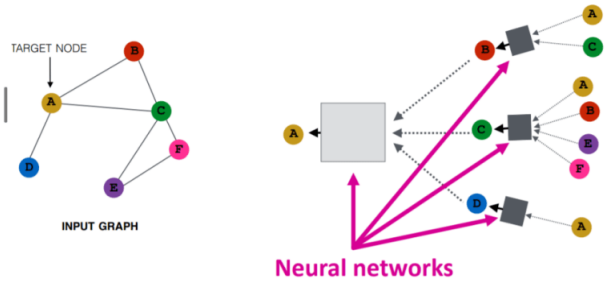

FIGURE 2.10: Neural Networks for Graphs, from [16]

Neural Networks are presented in grey boxes. They require aggregations to be order-invariant, like sum, average, maximum, because they are permutation-invariant functions.

This property enables the aggregations to be performed.

Let's move on to the forward propagation rule in GNNs. It determines how the information from the input will go to the output side of the neural network.

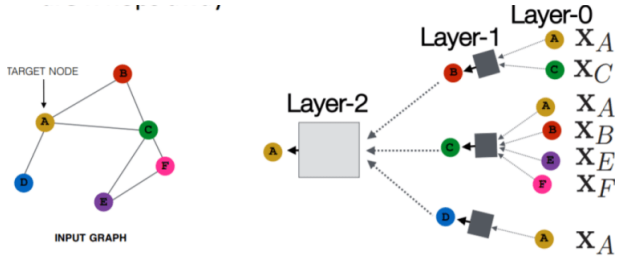

FIGURE 2.11: Deep Model - Many Layers, from [16]

Every node has a feature vector. For example, (X_A) is a feature vector of node A.

The inputs are those feature vectors, and the box will take the two feature vectors (X_A and X_c), aggregate them, and then pass on to the next layer.

Notice that, for example, the input at node C are the features of node C, but the representation of node C in layer 1 will be a hidden, latent representation of the node, and in layer 2 it'll be another latent representation.

So in order to perform forward propagation in this computational graph, we need 3 steps:

1. **Initialize the activation units:**

$$h_v^0 = X_v \tag{2.10}$$

2. **Every layer in the network:**

$$h_v^k = \sigma(W_k \Sigma \frac{h_v^{k-1}}{|N(v)|} + B_k h_v^{k-1}) \tag{2.11}$$

   We can notice that there are two parts for this equation: The first part is basically averaging all the neighbors of node v. The second part is the previous layer embedding of node v multiplied with a bias B_k, which is a trainable weight matrix and it's basically a self-loop activation for node v. And σ the non-linearity activation that is performed on the two parts.

3. **The last equation (at the final layer):**

$$z_v = h_v^K \tag{2.12}$$

   It's the embedding after K layers of neighborhood aggregation.

Now, to train the model we need to define a loss function on the embeddings. We can feed the embeddings into any loss function and run stochastic gradient descent to train the weight parameters.

Training can be unsupervised or supervised:

- **Unsupervised Learning:** Use only the graph structure: similar nodes have similar embeddings. Unsupervised loss function can be a loss based on node proximity in the graph, or random walks.
- **Supervised Learning:** Train model for a supervised task like node classification, normal or anomalous node.

To recap, in this section we described a basic idea of generating node embeddings by aggregating neighborhood information.

2.4.3.3 Graph Convolutional Networks

GCNs were first introduced in "Spectral Networks and Deep Locally Connected Networks on Graphs" (Bruna et al, 2014), as a method for applying neural networks to graph-structured data.

The simplest GCN has only three different operators:

- Graph convolution
- Linear layer
- Nonlinear activation

The operations are usually done in this order. Together, they make up one network layer. We can combine one or more layers to form a complete GCN.

2.4.3.4 GraphSAGE Idea

GraphSAGE (Hamilton et al, NIPS 2017) is a representation learning technique for dynamic graphs. It can predict the embedding of a new node, without needing a re-training procedure. To do this, GraphSAGE uses inductive learning. It learns aggregator functions which can induce new node embedding, based on the features and neighborhood of the node.

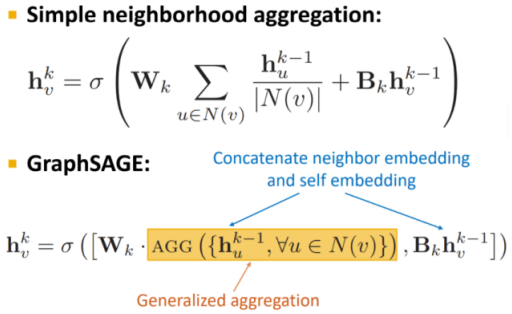

FIGURE 2.12: GraphSAGE, from [16]

We can notice two big differences. Rather than summing two things together and losing track of them, we use a general aggregation function which keeps them separate by concatenating them.

Before, we were using the Mean aggregation function – we simply took the message from the neighbors and added them up, and then normalized that by the number of neighbors. Now, we can also make a pooling type approach, or we can also use a deep neural network like an LSTM

2.4.4 Applications of GNNs

Let's go through some applications across domains where GNN can resolve various challenges.

2.4.4.1 GNNs in Computer Vision

Using regular CNNs, machines can distinguish and identify objects in images and videos. Although there is still much development needed for machines to have the visual intuition of a human. Yet, GNN architectures can be applied to image classification problems.

One of these problems is scene graph generation, in which the model aims to parse an image into a semantic graph that consists of objects and their semantic relationships. Given an image, scene graph generation models detect and recognize objects and predict semantic relationships between pairs of objects.

However, the number of applications of GNNs in computer vision is still growing. It includes human-object interaction, few-shot image classification, and more.

2.4.4.2 GNNs in Natural Language Processing

In NLP, we know that the text is a type of sequential data which can be described by an RNN or an LSTM. However, graphs are heavily used in various NLP tasks, due to their naturalness and ease of representation.

Recently, there has been a surge of interest in applying GNNs for a large number of NLP problems like text classification, exploiting semantics in machine translation, user geolocation, relation extraction, or question answering.

We know that every node is an entity and edges describe relations between them. In NLP research, the problem of question answering is not recent. But it was limited by the existing database. Although, with techniques like GraphSage (Hamilton et al.) [10], the methods can be generalized to previously unseen nodes.

2.4.4.3 GNNs in Traffic

Forecasting traffic speed, volume or the density of roads in traffic networks is fundamentally important in a smart transportation system. We can address the traffic prediction problem by using STGNNs.

Considering the traffic network as a spatial-temporal graph where the nodes are sensors installed on roads, the edges are measured by the distance between pairs of nodes, and each node has the average traffic speed within a window as dynamic input features.

2.4.4.4 GNNs in other domains

The application of GNNs is not limited to the above domains and tasks. There have been attempts to apply GNNs to a variety of problems such as program verification, program reasoning, social influence prediction, recommender systems, electrical health records modeling, brain networks, and adversarial attack prevention.

2.5 Reinforcement Learning

Reinforcement learning is a powerful framework for controlling dynamical systems that can be used to learn control policies with minimal user intervention. A complete review of reinforcement learning is outside the scope of this thesis, but may be found across several resources like [31] [32] [24].

Reinforcement learning (RL) is a general framework where agents learn to perform actions in an environment so as to maximize a reward. The two main components are the environment, which represents the problem to be solved, and the agent, which represents

the learning algorithm.

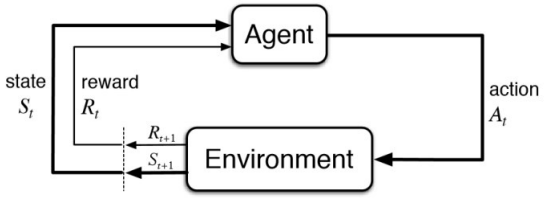

FIGURE 2.13: Reinforcement Learning Cycle, from [31]

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning [31].

A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes are intended to include just these three aspects—sensation, action, and goal—in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems we consider to be a reinforcement learning method.

The agent and environment continuously interact with each other. At each time step, the agent takes an action on the environment based on its policy $\Pi(\frac{a_t}{s_t})$, where s_t is the current observation from the environment, and receives a reward r_{t+1} and the next observation s_{t+1} from the environment. The goal is to improve the policy $\Pi(\frac{a_t}{s_t})$ so as to maximize the sum of rewards $\sum_{t=0}^{T} \gamma^t r_t$. Here γ^t is a discount factor in [0,1] that discounts future rewards relative to immediate rewards. This parameter helps us focus the policy, making it care more about obtaining rewards quickly.

It is important to distinguish between the state of the environment and the observation, which is the part of the environment state that the agent can see, e.g. in a poker game, the environment state consists of the cards belonging to all the players and the community cards, but the agent can observe only its own cards and a few community cards. In most literature, these terms are used interchangeably and observation is also denoted as s.

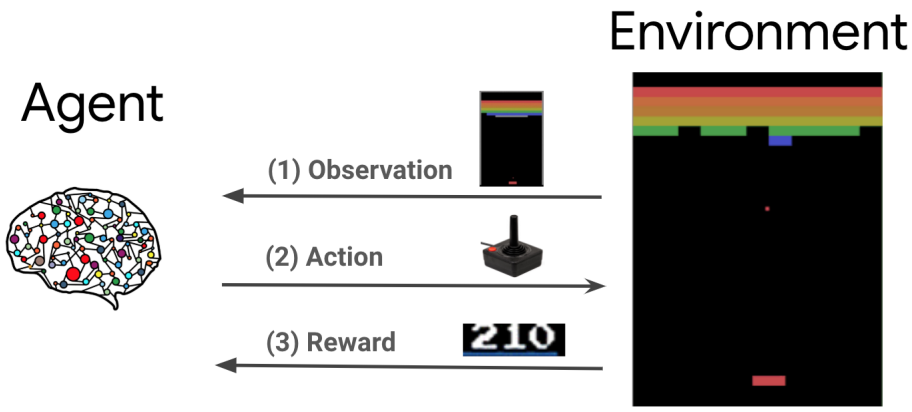

FIGURE 2.14: Reinforcement Learning for Atari Game, from [33]

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward. The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved [31].

Now, we will talk briefly about two methods that are relevant for the scope of our work:

2.5.1 Deep Q-Learning (DQN)

The DQN (Deep Q-Network) algorithm was developed by DeepMind in 2015. It was able to solve a wide range of Atari games (some to superhuman level) by combining reinforcement learning and deep neural networks at scale. The algorithm was developed by enhancing a classic RL algorithm called Q-Learning with deep neural networks and a technique called experience replay.

Q-Learning

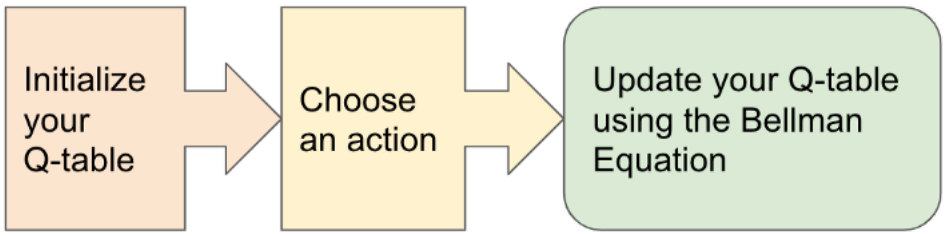

FIGURE 2.15: The Q-Learning Algorithm, from [36]

Q-Learning is based on the notion of a Q-function. The Q-function (a.k.a the state-action value function) of a policy π, $Q^{\pi}(s,a)$, measures the expected return or discounted sum of rewards obtained from state s by taking action a first and following policy π thereafter. We define the optimal Q-function $Q^{*}(s,a)$ as the maximum return that can be obtained starting from observation s, taking action a and following the optimal policy thereafter. The optimal

Q-function obeys the following Bellman optimality equation:

$$Q^*(s,a) = \mathbb{E}[r + \gamma \operatorname*{argmax}_{a'} Q^*(s,a)] \tag{2.13}$$

This means that the maximum return from state s and action a is the sum of the immediate reward r and the return (discounted by γ) obtained by following the optimal policy thereafter until the end of the episode (i.e., the maximum reward from the next state s'). The expectation is computed both over the distribution of immediate rewards r and possible next states s'.

The basic idea behind Q-Learning is to use the Bellman optimality equation as an iterative update $Q_{i+1}(s,a) \longleftarrow \mathbb{E}[r + \gamma \operatorname{argmax}_{a'} Q^*(s,a)]$, and it can be shown that this converges to the optimal Q-function, i.e. $Q_i \to Q^*$ as $i \to \infty$ [18].

Deep Q-Learning

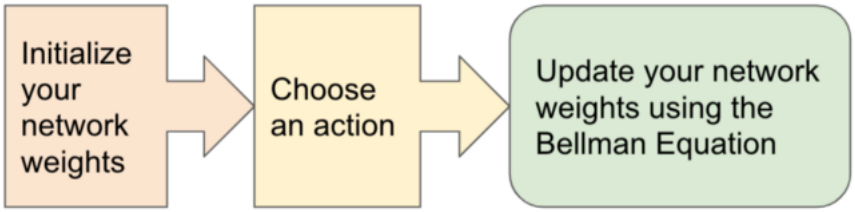

FIGURE 2.16: The Deep Q-Learning Algorithm, from [36]

For most problems, it is impractical to represent the Q-function as a table containing values for each combination of s and a. Instead, we train a function approximator, such as a neural network with parameters θ , to estimate the Q-values, i.e. $Q(s,a;\theta) \approx Q^*(s,a)$. This can done by minimizing the following loss at each step i:

$$L_i(\Theta_{i)} = \mathbb{E}_{s,a,r,s' \sim \rho}[(y_i - Q(s,a;\theta_{i-1}))^2] \tag{2.14}$$

where

$$y_i = r + \gamma \operatorname*{argmax}_{a'} Q(s',a',\theta_{i-1}) \tag{2.15}$$

Here, y_i is called the TD (temporal difference) target, and $y_i - Q$ is called the TD error. ρ represents the behaviour distribution, the distribution over transitions s,a,r,s' collected from the environment.

Note that the parameters from the previous iteration θ_{i-1} are fixed and not updated. In practice we use a snapshot of the network parameters from a few iterations ago instead of the last iteration. This copy is called the target network.

Q-Learning is an off-policy algorithm that learns about the greedy policy $a = \operatorname{argmax}_a Q(s,a;\theta)$ while using a different behaviour policy for acting in the environment/collecting data. This behaviour policy is usually an ε-greedy policy that selects the greedy action with probability $1-\varepsilon$ and a random action with probability ε to ensure good coverage of the state-action space.

Experience-Replay

To avoid computing the full expectation in the DQN loss, we can minimize it using stochastic gradient descent. If the loss is computed using just the last transition s, a, r, s', this reduces to standard Q-Learning.

The Atari DQN work [18] introduced a technique called Experience Replay to make the network updates more stable. At each time step of data collection, the transitions are added to a circular buffer called the replay buffer. Then during training, instead of using just the latest transition to compute the loss and its gradient, we compute them using a mini-batch of transitions sampled from the replay buffer. This has two advantages: better data efficiency by reusing each transition in many updates, and better stability using uncorrelated transitions in a batch.

2.5.2 Proximal Policy Optimization (PPO)

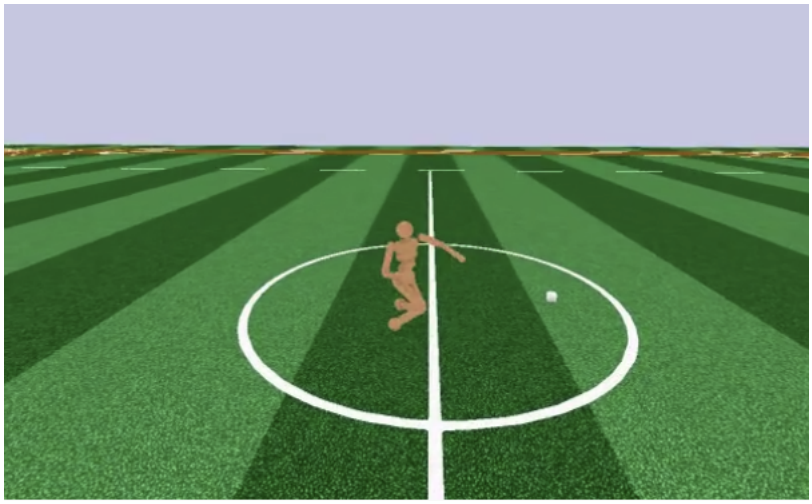

FIGURE 2.17: PPO lets us train AI policies in challenging environments, like the Roboschool one shown above where an agent tries to reach a target (the pink sphere), learning to walk, run, turn, use its momentum to recover from minor hits, and how to stand up from the ground when it is knocked over [29] .

Policy gradient methods are fundamental to recent breakthroughs in using deep neural networks for control, from video games[19], to 3D locomotion [28], to Go [30]. But getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks.

Researchers have sought to eliminate these flaws with approaches like Trust Region Policy Optimization (TRPO) [27] and ACER [37], by constraining or otherwise optimizing the size of a policy update. These methods have their own trade-offs — ACER is far more complicated than PPO, requiring the addition of code for off-policy corrections and a replay buffer, while only doing marginally better than PPO on the Atari benchmark; TRPO — though useful for continuous control tasks — isn't easily compatible with algorithms that share parameters between a policy and value function or auxiliary losses, like those used to solve problems in Atari and other domains where the visual input is significant.

PPO
With supervised learning, we can easily implement the cost function, run gradient descent on it, and be very confident that we'll get excellent results with relatively little hyperparameter tuning [29]. The route to success in reinforcement learning isn't as obvious — the algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

The new variant uses a novel objective function not typically found in other algorithms:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t] \tag{2.16}$$

Where:

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

This objective implements a way to do a Trust Region update which is compatible with Stochastic Gradient Descent, and simplifies the algorithm by removing the KL penalty and need to make adaptive updates. In tests, this algorithm has displayed the best performance on continuous control tasks and almost matches ACER's performance on Atari, despite being far simpler to implement.

2.6 Imitation Learning

RL is one of the most interesting areas of ML, where an agent interacts with an environment by following a policy. In each state of the environment, it takes action based on the policy, and as a result, receives a reward and transitions into a new state. The goal of RL is to learn an optimal policy which maximizes the long-term cumulative rewards.

To achieve this, there are several RL algorithms and methods, which use the received rewards as the main approach to approximate the best policy. Generally, these methods perform really well. In some cases, though the teaching process is challenging. This can be especially true in an environment where the rewards are sparse (e.g. a game where we can only receive a reward when the game is won or lost).

To help with this issue, we can manually design reward functions, which provide the agent with more frequent rewards. Also, in certain scenarios, there isn't any direct reward function (e.g. teaching a self-driving vehicle), thus the manual approach is necessary.

However, manually designing a reward function that satisfies the desired behaviour, cam be extremely complicated.

A feasible solution to this problem is imitation learning (IL). In IL instead of trying to learn from the sparse reward or manually specifying a reward function, an expert (typically a human) provides us with a set of demonstrations. The agent then tries to learn the optimal

policy by following and imitating the experts decisions.

Imitation learning techniques aim to mimic human behavior in a given task. An agent (a learning machine) is trained to perform a task from demonstrations by learning a mapping between observations and actions.

The idea of teaching by imitation has been around for many years, however, the field is gaining attention recently due to advances in computing and sensing as well as rising demand for intelligent applications. The paradigm of learning by imitation is gaining popularity because it facilitates teaching complex tasks with minimal expert knowledge of the tasks.

Generally, imitation learning is useful when its easier for an expert to demonstrate the desired behaviour rather than to specify a reward function which would generate the same behaviour or to directly learn a policy.

The three imitation learning approaches are:

1. Behavioural Cloning
2. Direct Policy Learning
3. Inverse Reinforced Learning

	Direct Policy Learning	Reward Learning	Access to Environment	Interactive Demonstrator	Pre-collected Demonstrations
Behavioral Cloning	Yes	No	No	No	Yes
Direct Policy Learning (Interactive IL)	Yes	No	Yes	Yes	Optional
Inverse Reinforcement Learning	No	Yes	Yes	No	Yes

FIGURE 2.18: Types of IL, from [38]

2.6.1 Behavioural Cloning

The simplest form of imitation learning is behaviour cloning (BC), which focuses on learning the experts policy using supervised learning. (1st done in 1989-1999 Dean Pomerleau et al., for autonomous vehicles).

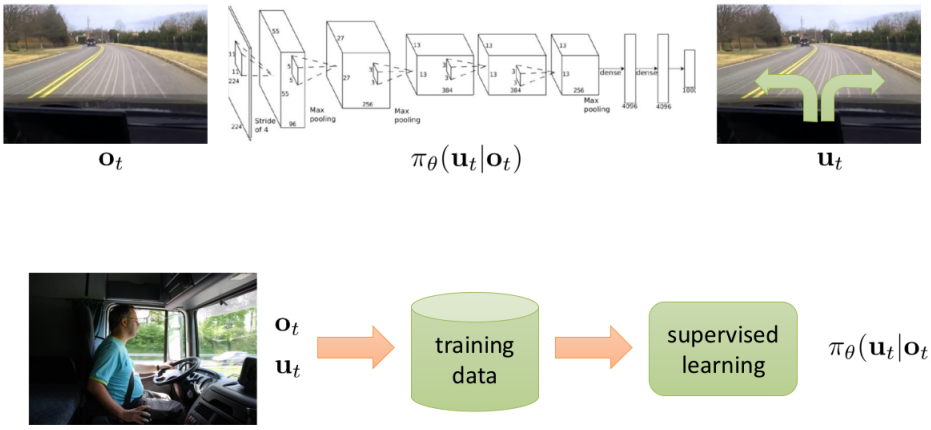

FIGURE 2.19: Bojarski et al. '16 NVIDIA, from [38]

The way BC works is quite simple. Given the experts demonstrations we divide these into state-action pairs, we treat these pairs as i.i.d examples and finally apply a supervised learning algorithm. The loss function can be dependent on the application.

Therefore, the algorithm is as following:

Algorithm 1 Pseudo-code for Behavioural Cloning

1: Collect demonstrations (τ^* trajectories) from expert.
2: Treat the demonstrations as i.i.d state-action pairs: (s_0^*, a_0^*), (s_1^*, a_1^*), ..., (s_n^*, a_n^*)
3: Learn policy using supervised learning by minimizing the loss function $L(a^*, \Pi_\theta(s))$

For some cases, BC can work excellently. For the majority of the cases, though, BC can be quite problematic.
The main reason for this is the i.i.d assumption: while supervised learning assumes that the state-action pairs are distributed i.i.d, in the Markov Decision Process (MDP) an action in a given state induces the next state, which breaks the previous assumption.

This also means, that errors made in different states add up, therefore a mistake made by the agent can easily put it into a state that the expert has never visited and the agent has never been trained on. In such states, the behaviour is undefined, and this can lead to catastrophic failures.

Still, BC can work quite well in certain applications. Its main advantages are its simplicity and efficiency. Suitable applications can be those, where we don't need long-term planning, the experts trajectories can cover the state space, and where commiting an error doesn't lead to fatal consequences. However, we should avoid using BC when any of these characteristics are true.

2.6.2 Direct Policy Learning

Direct Policy Learning (DPL) is basically an improved version of behavioural cloning. This iterative method assumes, that we have access to an interactive demonstrator at training time, who we can query.

Just like in BC, we collect some demonstrations from the expert, and we apply supervised learning to learn a policy. We roll out this policy in our environment, and we query the expert to evaluate the roll-out trajectory. In this way, we get more training data which we feedback to supervised learning. This loop continues until we converge.

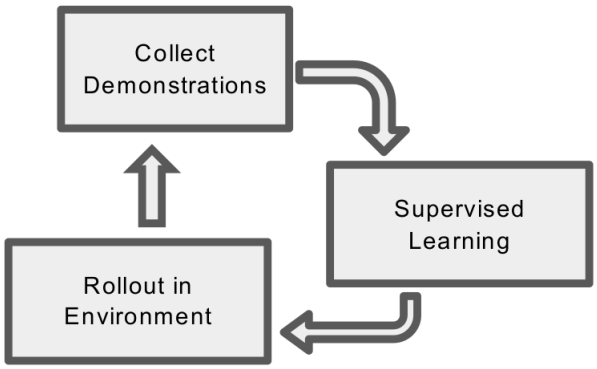

FIGURE 2.20: Direct Policy Learning Cycle, from [38]

The way the general DPL algorithm is the following: First, we start with an initial predictor policy based on the initial expert demonstrations. Then, we execute a loop until we converge. In each iteration, we collect trajectories by rolling out the current policy (which we obtained in the previous iteration), and using these we estimate the state distribution. Then, for every state, we collect feedback form the expert, what would he have done in the same state. Finally, we train a new policy using this feedback.

To make the algorithm work efficiently, it is important to use all the previous training data during the teaching, so that the agent "remembers" all the mistakes it made in the past. There are several algorithms to achieve this, data aggregation and policy aggregation.

Data aggregation trains the actual policy on all the previous training data. Meanwhile, policy aggregation trains a policy on the training data received in the last iteration and then combines this policy with all the previous policies using geometric blending. In the next iteration, we use this newly obtained, blended policy during the roll-out. Both methods are convergent, in the end, a policy which is not much worse than the expert.

The full algorithm is as following:

Algorithm 2 Pseudo-code for Direct Policy Learning

Initial Predictor: π_0
For m=1:
1: Collect trajectories τ by rolling out π_{m-1}
2: Estimate state distribution P_m using $s \in \tau$
3: Collect interactive feedback $[\pi^*(s) \mid s \in \tau\ ]$
4: Data Aggregation (e.g. DAgger)
- Train π_m on $P_1 \cup ... \cup P_m$
5: Policy Aggregation (e.g. SEARN or SMILE)
- Train $\pi_m^{'}$ on P_m
- $\pi_m = \beta\ \pi_m^{'} + (1\text{-}\beta)\ \pi_{m-1}$

Iterative direct policy learning is a very efficient method, which does not suffer from the problems that BC does. The only limitation of this method is the fact, that we need an expert that can evaluate the agent's actions at all times, which is not possible in some applications.

2.6.3 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) is a diferent approach of IL, where the main idea is to learn the reward function of the environment based on the experts demonstrations, and then find the optimal policy (the one that maximizes this reward function) using RL.

In this approach, we start with as set of expert demonstrations (we assume these are optimal), and then we try to estimate the parametrized reward function, that would cause the experts behaviour/policy.

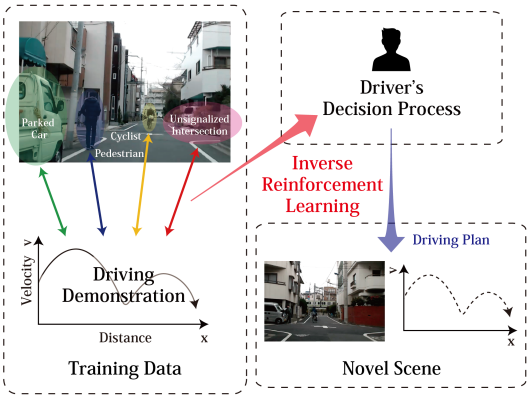

FIGURE 2.21: Modeling risk anticipation and defensive driving on residential roads with inverse reinforcement learning, from [38]

We repeat the following process until we find a good enough policy:

1. We update the reward function parameters
2. Then we solve the reinforced learning problem (given the reward function we try to find the optimal policy).
3. Finally, we compare the newly learned policy with the experts policy.

The general IRL algorithm is the following:

Algorithm 3 Pseudo-code for Inverse Reinforcement Learning

Collect Expert Demonstrations: D = $[\tau_1, \ldots, \tau_m]$
In a loop:
- Learn reward function $r_\theta(s_t, a_t)$.
- Given the reward function r_θ, learn π using RL.
- Compare π with π^* (expert's policy).
- Stop if π is satisfactory.

Depending on the actual problem, there can be two approaches of IRL:

1. The model-based approach
2. The model-free approach

2.6.4 Model-Based vs Model-Free

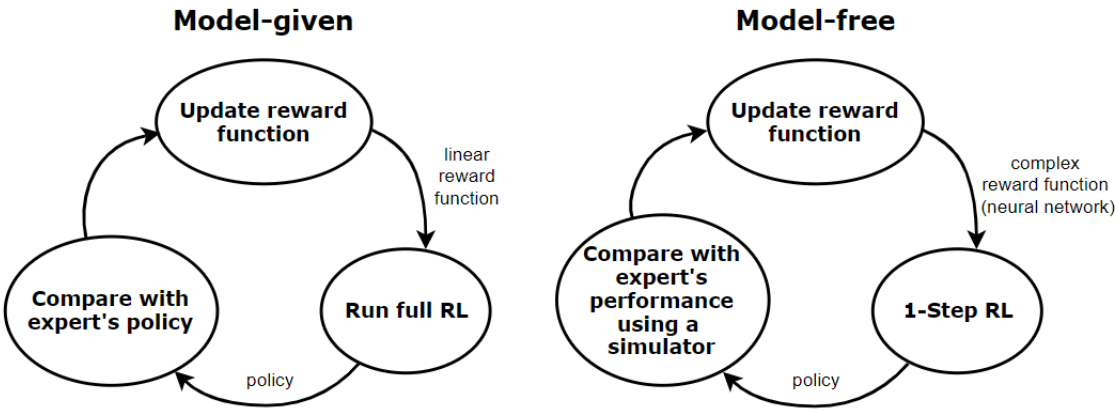

FIGURE 2.22: Model based vs Model free, from [38]

The main difference between model-free and model-based RL is the policy network, which is required for model-based RL and unnecessary in model-free.

Model-based: our reward function is linear. In each iteration, we need to solve the full RL problem, so to be able to do this efficiently, we assume, that the environment's (the MDP's) state space is small. We also suppose that we know the state transition dynamics of the environment. This is needed so that we can compare our learned policy with the expert's one effectively.

Model-free: is a more general case. Here, we assume that the reward function is more complex, which we usually model with a neural network. We also suppose that the state space of the MDP is large or continuous, therefore in each iteration, we only solve a single step of the RL problem. In this case, we don't know the state transitions dynamics of the environment, but we assume, that we have access to a simulator or the environment. Therefore, comparing our policy to the expert's one is trickier.

In both cases, however, learning the reward function is ambiguous. The reason for this is that many rewards functions can correspond to the same optimal policy (the expert's policy). To solve this issue, we can use the maximum entropy principle proposed by Ziebart: we should choose the trajectory distribution with the largest entropy.

2.7 Conclusion

In this section, we talked about the main supplementary background that our thesis is revolved around. Providing a complete material to cover all the concepts of the previously mentioned fields is beyond the scope of this thesis.

Chapter 3

Decentralized Graph Neural Networks

In this chapter, we will talk about how we built our decentralized model to replace centralized algorithms. We explain how centralized algorithms work, how we designed and came up with our decentralized model, how we went about collecting data using an expert consensus algorithm, and conclude about the main take aways from this section.

3.1 Centralized Consensus Algorithm

The topology of our system allows at any time each robot to access information of another robot and to get updates from them. Each robot will regulate its position with respect to its neighbors. The consensus-based control law will specify the exchange of information within the system, between a robot and all its nearby neighbors, and will predict the corresponding control input for each robot.

The control input is described as below:

$$\begin{cases} U_{x_i} = -\sum_{j=1}^{n} A_{ij} K_{ij} * [(x_i - x_j)] \\ U_{y_i} = -\sum_{j=1}^{n} A_{ij} K_{ij} * [(y_i - y_j)] \end{cases} \tag{3.1}$$

Where:

- $\mathbf{x}_i$: The state variable representing the position of robot i at any time t
- $\mathbf{x}_j$: Constant representing the final target tracked by the robot i
- $\mathbf{K}_{ij}$: Gain associated to track final target of robot i
- $\mathbf{A}_{ij}$: Gain correlating communication from robot i to robot j

The most important key characteristic in multi-robot systems is defining the adjacency matrix, that defines how the network of robots communicate with each other and collaborate.

Since in our experiments, we used CoppeliaSim as a physics simulator and ROS for publishing results of our algorithms, below is the pseudo code of the consensus algorithm that we used to generate data and mimic in a imitation learning framework.

Algorithm 4 Pseudo Code of a Centralized Consensus Algorithm

Input: $[x_i]$, $[y_i]$, $[A_{ij}]$
Output: $[U_x]$, $[U_y]$
1: Define adjacency matrix A_{ij}
2: Read poses of robots x_i, y_i, Θ_i
3: Calculate control inputs using equation (3.1)
4: Transform each control input to V_L, V_R
5: Publish Speeds

Now since our decentralized controller runs in the local reference frame of each robot, we transformed both the local pose of each robot as well as the controllers in to the local reference frame using the following formula's:

$$\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}_L = \begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}_G \tag{3.2}$$

$$\begin{bmatrix}U_x\\ U_y\end{bmatrix}_L = \begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}U_x\\ U_y\end{bmatrix}_G \tag{3.3}$$

Where:

- x_d, y_d and θ_d are the coordinates of the reference to be followed.
- x_e, y_e and θ_e represent the longitudinal, lateral and rotational errors of the vehicle with respect to the desired trajectory (in our case, the center lane of the trajectory).

3.2 Decentralized GNN Model

In this section, we discuss our decentralized controller that's powered by deep learning for replacing the centralized algorithm. Now at the beginning, we tried several vanilla-architectures and based on error analysis and performance, we made adjustments to the model design. Initially, we tried a two input two output model; but due to poor performance and generalization power we kept on engineering a lot of experiments (review appendix A for details).

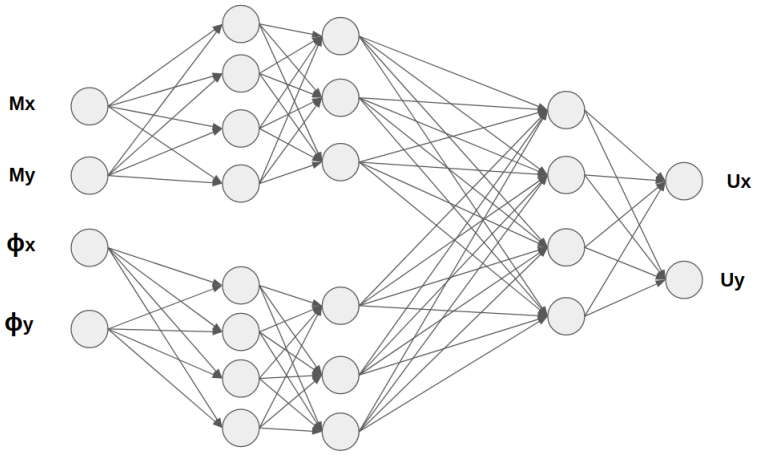

FIGURE 3.1: Ensembled Model Architecture

After running several different experiments, and doing some feature engineering we came up with the final design solution presented in the figure above which has four inputs. You can read A, B for more details.

The equations used to calculate the inputs of this GNN model per robot are:

$$Mx_i = \sum_{j \in N} \frac{A_{ij}(x_j - x_i)}{N-1} \tag{3.4}$$

$$My_i = \sum_{j \in N} \frac{A_{ij}(y_j - y_i)}{N-1} \tag{3.5}$$

$$\Phi_{xi} = \sum_{j \in N} \frac{A_{ij} Mx_i}{N-1} \tag{3.6}$$

$$\Phi_{yi} = \sum_{j \in N} \frac{A_{ij} My_j}{N-1} \tag{3.7}$$

N here is the number of robots within the network. It is important to note that since our solution is decentralized; this model is used independently for each robot, and this is the key power of our approach over centralized algorithms. Diving deeper into these equations, each robot can operate independently without having to communicate with any nearby robot, nor depend on some sort of centralized operator that might fail due to any random reason. The deep learning models built were in PyTorch.[1]

As of some architectural details: we used ReLU functions as activation functions, stochastic gradient descent (SDG) as optimizers, xavier uniform functions for initializing weights of the neural networks, mean-squared error (MSE) as a performance metric since it's a regression problem, 100 epochs, a learning rate of 0.01, and a momentum of 0.9.

Now for integrating these models in real time, we used ROS to publish speeds and predictions from the GNN models, and CoppeliaSim as a physics-based engine to simulate performance.

3.3 Generating Data for Training

In order to develop a decentralized controller that imitates the behaviour of a centralized one, we used behavioural cloning which is the simplest form of imitation learning. In particular, because the problem we are trying to solve can be framed as an imitation learning one, we've ran an expert algorithm to collect labelled data to later train on.

By writing some Python scripts, and using the Python Remote API from CoppeliaSim: we designed an algorithm that changes the initial positions/orientations of each robot per scene, runs an expert consensus algorithm and saves labelled data of the inputs/outputs required to train our GNN model.

Since we want our framework to work at test time on weak communication topologies, we collected data from an expert algorithm on a **fully connected graph**.

Below is the pseudo-code that we used to collect our labelled dataset:

[1] Read code documentation here [12] for more details.

Algorithm 5 Pseudo Code for Data Collection

1: **Read** positions x1, x2; y1, y2; Θ1, Θ2, and calculate distances between robots
2: **If:** distance > robots meeting:
- Run Consensus Algorithm
- Publish Speeds
- Save to 2 CSV files for each robot: relative pose + control input (local frame)
else:
- Stop simulation
- Retrieve object handles
- Set robot positions/orientations
- Start simulation
3: **Stop** code and disconnect from CoppeliaSim after defined nb. of scenes achieved

3.4 Conclusion

In this section, we provided an overview of how we designed our decentralized GNN model from scratch as well as the input features, and then discussed how we generated data to train our model on.

We would also like to note that we tried training our model on several different types of graph datasets, and also tried fusing datasets of both a fully connected graph and a line graph; but didn't notice any enhanced performance, or in other words doesn't really correlate with the usability of this decentralized framework as an fundamental approach of replacing centralized algorithms.

Chapter 4

Testing and Validation

In this chapter, we will talk about how we setted up the simulation environment in CoppeliaSim, how we integrated ROS within the Python code in order to publish speeds and subscribe to poses of each robot, and also how we built/trained our deep learning model in PyTorch. Finally, we provide some experiments that we performed to compare the performance of our decentralized model with an expert centralized one, and on some weak communication topologies.

4.1 Simulation Environment

For our experiments, we used V-Rep also known as CoppeliaSim as a physics engine/robotics simulator to test the performance and efficiency of our models. We integrated ROS within our code in order to interact and control the robots within the simulator.

4.1.1 V-REP

V-REP is a general purpose robot simulator with integrated development environment. It is extremely useful for modelling and simulating various sensors, mechanisms, robots and even all combined. It is widely used for verification of the product (robot) developed and for fast prototyping, controller development and its performance testing in the intended space, Hardware control from external environment with a middle-ware solution, safety monitoring of a devised prototype, convenient presentation of the product developed by any industry for the users, etc.

The simulator has three core elements:

1. **Scene Objects:**
   - To create any evaluation, testing or working space for the subject under consideration including adding any sensors like proximity, force/torque, vision sensor (cam- era), light, mirrors, dummies for static or dynamic environment, and many more.
2. **Calculation Module:**
   - 5 basic algorithms which can be used separately or combined according to the need. 5 basic algorithms are:
     (a) Physics/Dynamics calculation of various robot types.
     (b) Forward / inverse kinematics of various mechanisms.
     (c) Minimum distance calculation of one part of the robot from another or even one whole robot from another.
     (d) Path / motion planning
     (e) Collision detection.
3. **Control Mechanisms:**

- 5 methods of interfaces are available to utilize and 7 languages are supported. The methods can be utilized separately or in combinations. Methods / interfaces with supported languages are:
  (a) Embedded Scripts - over 500 API functions available and further extendable, many Lua extension libraries and interfaces available, etc.
  (b) Plugins - over 500 API functions available, C/C++ interfaces, ROS, etc.
  (c) Add-ons - over 400 API functions available, Lua interfaces supported and customizable simulation environment.
  (d) Remote API Clients - over 100 API functions available, C/C++, Python, Java, Matlab, Octave, Lua and urbi interfaces supported.
  (e) ROS Nodes - It is plugins based, ROS interface.

For this thesis, the objectives are to test the behavior of multiple robots in a team cooperating with each other with very specific requirements related to the global connectivity and flexibility in terms of increasing the number of agents in the team but the environment for their operations does not include obstacles or autonomous navigation in a dynamic environment so setting up the scene in v-rep is not exactly a necessity.

Now while choosing a suitable scene to experiment our work on, we chose the BubbleRob scene since it's already open sourced. We did some small modifications in the Lua code, and gradually increased the number of robots in order to prove the performance of our model on bigger networks.

Also, while our data was generated on bubblerob robots we can easily extend this framework for different types of robots by just applying the same methodology to different robots. We considered working on some kind of aerial vehicles, but due to the non linearity of the dynamics of quadcopters we decided that it's not the best platform to build our model around.

4.1.2 ROS

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. It is an open source collection of software frameworks supported on Ubuntu, Debian and Windows. There are many online communities/forums to help beginners.

We used ROS as an interface in order to interact with the robots from within the simulator, mainly using a single node that subscribes to read the poses of each robot and then publish the local speed inputs of each robot. For a complete documentation of the code, classes, and how we integrated ROS within our framework read [12].

4.2 Experimental Results

Now in order to prove the usability of our framework on the problem of consensus, we ran several experiments on well-used types of graphs for multi-robots and plotted their convergence rate to measure the success rate of our approach compared to centralized ones.
As a stopping condition, we chose this distance metric which represents the global measure of how close all robots are to each other:

$$d = \sum_{j \in N} |(x_1 - x_j) + (y_1 - y_j)| \tag{4.1}$$

In other words, in order for a consensus scene to be considered successful the value of the global disk distance d must be around 0.2 which has been found empirically. Now since the goal of this thesis is to propose a decentralized deep learning solution, demonstrating the efficiency of this framework on the problem of consensus is sufficient enough to prove

that it can be scaled to different types of motion planning. Since in general, the only difference would be the expert algorithm that we'll use to generate data to train on as well as the reward function.

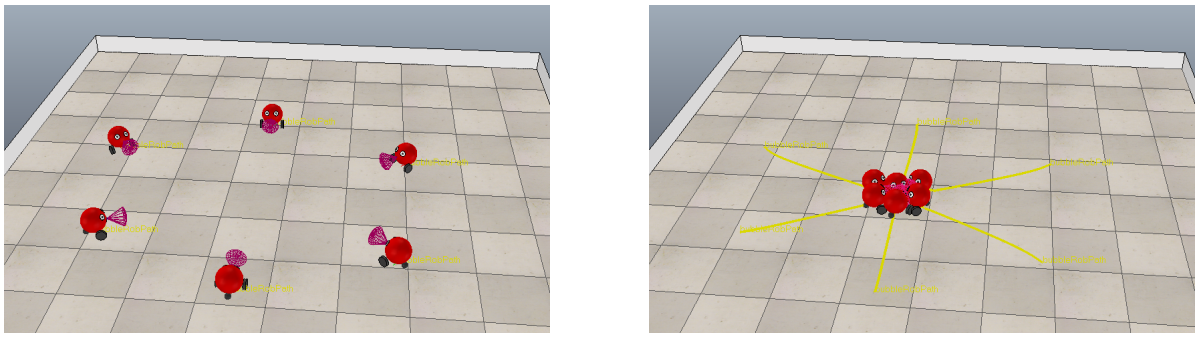

FIGURE 4.1: Consensus of Six BubbleRob Robots

4.2.1 Centralized Consensus

By running a centralized controller for consensus, below is the graph plot of the convergence rate of the disk distance d with respect to time for a network of six robots and a fully connected graph. We can notice that this curve can be fitted by an inverse exponential function which shall be demonstrated later that the same pattern shall be repeated with the decentralized ones.

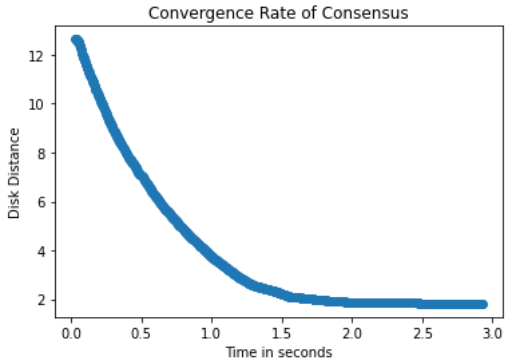

FIGURE 4.2: Convergence Rate of a Centralized Controller

By assuming that consensus is fully completed when d is approximately 0.2, we can notice from the above graph that a centralized controller took around 1.5 seconds for it to converge over a network of six robots of a fully connected graph.

4.2.2 Decentralized Consensus

Now here comes the different experiments that we did to test our decentralized GNN model. As mentioned earlier, this GNN model was trained on a fully connected graph, but in order to prove that it works on more weaker communication topologies we tested it on the following graphs:

1) Fully Connected Graph
2) Cyclic Graph
3) Line Graph
4) Random Weak Graph

4.2.3 Fully Connected Graph

For a fully connected graph, the adjacency matrix usually has the following formula:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Below is the convergence rate of the global disk distance for a fully connected graph:

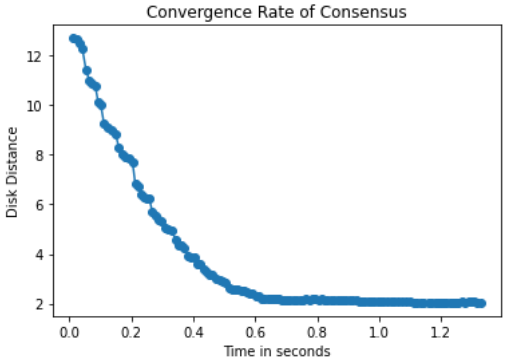

FIGURE 4.3: Convergence rate of GNN controller for a fully connected graph

Notice that the curve is very similar to the centralized controller, with a smaller convergence rate since the GNN was trained on a fully connected graph and thus at test time it actually isn't doing any real predictions.

4.2.4 Cyclic Graph

For a cyclic graph, the adjacency matrix usually has the following formula:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Below is the convergence rate of the global disk distance for a cyclic graph:

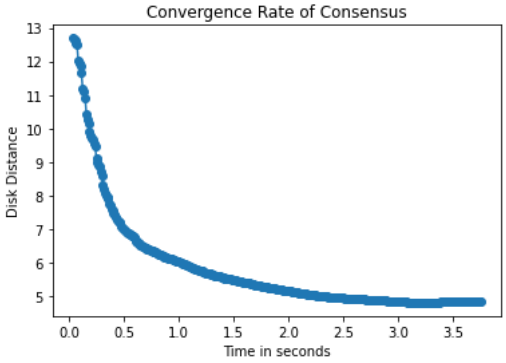

FIGURE 4.4: Convergence rate of GNN controller for a cyclic graph

Note that although the communication graph is quite weak, the decentralized GNN model was still able to converge in a pretty close period amount of time of a centralized controller.

4.2.5 Line Graph

For a line graph, the adjacency matrix usually has the following formula:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Below is the convergence rate of the global disk distance for a line graph:

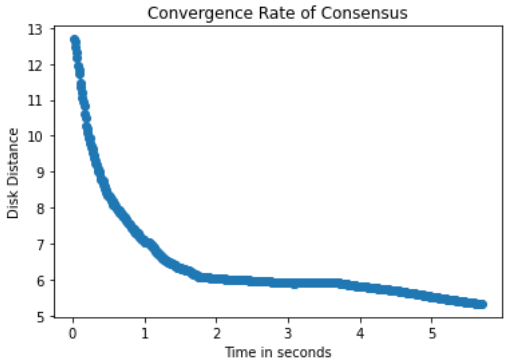

FIGURE 4.5: Convergence rate of GNN controller for a line graph

Note that for this particular case, since in a line graph the communication between robots is very weak the convergence rate took much more time in order to converge. But this result is actually expected since even in the real world with a centralized controller, having a line graph implies very weak communication.

4.2.6 Random Weak Graph

After experimenting with both strong and weak graphs, we decided to try a random graph with very weak connections in order to visualize how it might perform. Below is a random adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Below is the convergence rate of the global disk distance for a weak communication graph:

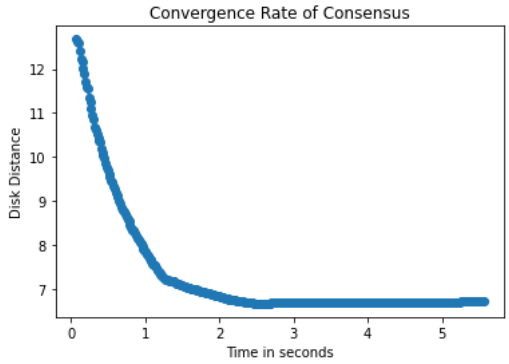

FIGURE 4.6: Convergence rate of GNN controller for a random graph

Note that this case is pretty similar to the line graph case, i.e the communication links between the robots is weak thus it's hard to capture the total amount of time needed in order for the network to converge. However, the important feature to notice is that the graph has the same form of all other types of graphs and does follow the convergence pattern.

4.3 Performance Discussion

As an overview, our decentralized GNN model has proved it's efficiency and stability in terms of convergence with a comparable performance with that of a centralized controller.

Although convergence rates vary depending on how strong the communication links are, we can confidently say that this framework is robust and strong enough to perform well on all types of graphs just as centralized models.

Plus, we demonstrated on the consensus task that our model can work, and be adaptive to both strong and weak communication graphs paving the way to conduct more research on improving the performance of this model and using it to solve more types of motion planning.

Studying the stability of this GNN model in a Lyapunov-theory equivalent is quite hard to do, due to the un-interpretability of deep learning models and lacking enough theory to support such analysis. However, it's safe to say by studying the data distribution, looking at the performance of regression metrics, and overall the scenario's of where this model might be implemented that it's stable with a big confidence interval.

Finally, we would like to point out that the consensus demonstration is rather a proof of concept, and not the ultimate goal. Thus, these results show very promising potential for this framework to solve other types of important multi-robot motion planning where centralized algorithms perform poorly.

Chapter 5

Conclusion and Future Work

5.1 Overview and Conclusion

This chapter summarizes the main points addressed by the thesis, gives conclusions about the presented results, and what potential they carry, and provides insights on future work than can be built upon this work.

The main objective of this thesis was to propose a decentralized data-driven control strategy for multi-robot motion planning. Centralized algorithms are computationally expensive, aren't robust to weak communication topologies, and don't scale well with big amounts of robots.

Our approach revolved around building a model to perform consensus as a use case demonstration. In order to mimic the performance of a centralized algorithm, we decided to build a model and learn in a imitation learning framework (BC). We ran an expert handcrafted consensus algorithm to collect data, and decided to use a fully-connected graph as our communication topology as it captures strong information exchange between robots which makes the deep learning model generalize to more weak communications.

We then designed several versions of GNN models from scratch, by doing some feature engineering and error analysis. After coming up with an effective model that performed well on the test set having the same distribution of data of the training set (fully connected graph), we set up several experiments to demonstrate the performance of our approach.

Results in chapter 4 showed the convergence of our decentralized GNN framework on different types of communication graphs, with very weak connections like a line graph. Although the convergence rate varied depending on the connectivity of the graph, globally all networks converged after a small amount of time. These experiments empirically prove the power of our approach on the consensus problem, and thus show promising potential to solve other types of multi-robot motion planning problems where decentralized algorithms are necessary.

5.2 Future Work and Perspective

After presenting our decentralized GNN model for consensus, training this model with reinforcement learning is an obvious step forward to improve the performance of our model. Since most motion planning problems have straight forward objective functions like the case of consensus, using reinforcement learning to train our GNN model can dramatically improve the performance of our model and learn more features.

One effective algorithm that can be used to train our GNN model is using Q-Learning with our GNN model integrated within this network. We developed a custom environment from scratch using Gym library from OpenAI in order to train our models, since there wasn't any off the shelf libraries online for our specific problem and robots. However, due to the limited time we stopped our work which shall be investigated by the team in the future.

Appendix A

Graph Neural Network v1

This is the first initial architecture of the MLP model that we designed. The model takes two inputs and gives two outputs. The inputs Mx, My represent the global relative pose information of each robot with respect to it's neighboring robots. The outputs Ux, Uy are the local control inputs that we want to learn and predict at inference time that are fed into each robot in a decentralized manner. These controllers are then transformed into local speeds left and right using the transformation equations in section 2.

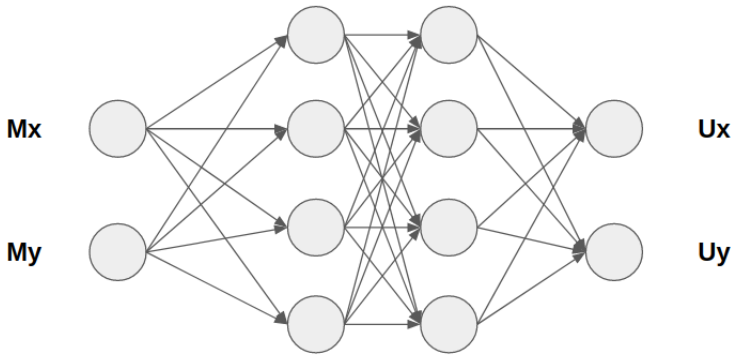

FIGURE A.1: First version of the GNN model

The equations used to calculate the inputs of this network are:

$$Mx_i = \sum_{j \in N} \frac{A_{ij}(x_j - x_i)}{N-1} \tag{A.1}$$

$$My_i = \sum_{j \in N} \frac{A_{ij}(y_j - y_i)}{N-1} \tag{A.2}$$

Note that these inputs are calculated in a decoupled way, i.e. they are exclusive for each robot depending on the relative pose of neighboring robots.

We trained this network on data collected from a fully connected graph consensus algorithm, in a imitation learning framework (BC) and noticed the following downsides:

1. Convergence wasn't fast, and more like a line graph at test time
2. Lacks generalization power to different communication topologies

Appendix B

Graph Neural Network v2

This is the second MLP architecture that we designed, this time trying to introduce two extra inputs Φ_x and Φ_y in order to capture additional global information of relative robots and topology in order for each robot to make better independent decentralized decisions.

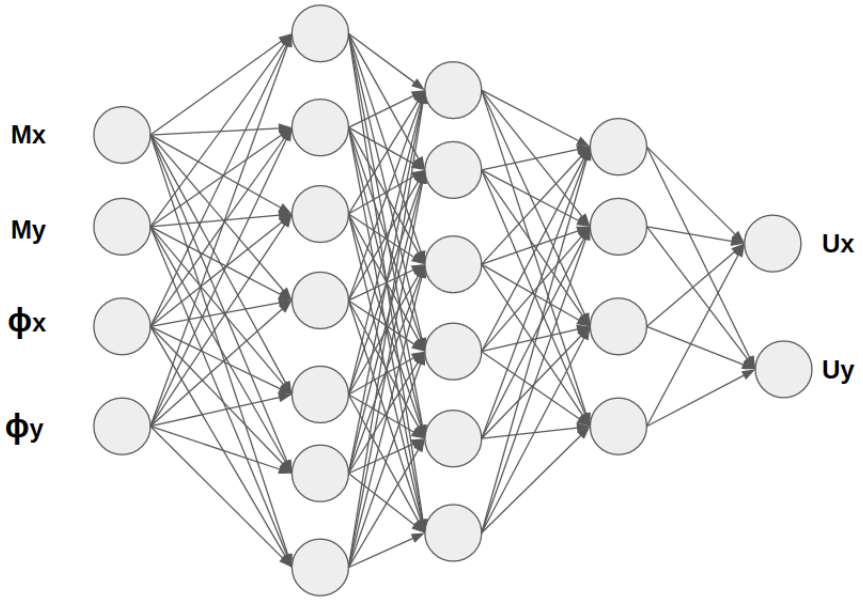

FIGURE B.1: Second version of the GNN model

The equations used to calculate the two additional inputs Φ_x and Φ_y are:

$$\Phi_{xi} = \sum_{j \in N} \frac{A_{ij} M x_j}{N-1} \tag{B.1}$$

$$\Phi_{yi} = \sum_{j \in N} \frac{A_{ij} M y_j}{N-1} \tag{B.2}$$

Note that these two additional terms enhanced the performance of our GNN model by a large margin, however after doing some data visualization we noticed that the values of these inputs are very correlated, and thus decoupled the two independent inputs which dramatically improved the performance of our model.

Appendix C

Gym Custom Environment

In order to train our robots using Deep Q-Learning, we created a custom environment from scratch using Gym from OpenAI and integrated it with V-Rep.

There's several functions of our custom Gym-Environment class that help the agent interact with the environment. Please read the github README.md file here for more [12].

We setup the environment in a way that robot one (i.e. our agent) uses our developed GNN model as the deep neural network in the DQN framework. And thus, the GNN model will be trained using RL to generalize to more complex topologies.

Three important functions are:

1. **Reset(self):**
   - This function returns the observation vector of the environment corresponding to the initial state, which is in our case is: observation-DQN = [Mx_1, My_1, Φx_1, Φy_1]
   - This function resets the environment to its initial state.
2. **Step(self, action):**
   - This function takes an action as an input and applies it to the environment, which leads to the environment transitioning to a new state.
   - The Step function returns 4 things:
     (a) **Observation:** The observation of the state of the environment [Mx_1, My_1, Φx_1, Φy_1].
     (b) **Reward:** The reward that you can get from the environment after executing the action.
     (c) **Done:** Whether the episode has been terminated (which is a Boolean).
     (d) **Info:** Useful info for debugging (which is a Dictionary).
3. **Init(self):**
   - Distance Threshold: that robots meet at.
   - Action Space: which is a discrete 4 box (up, down, left, right).
   - Observation Space: which is a continuous vector which represents the inputs to our GNN model.

Appendix D

GNN integrated within DQN

As a further future direction, in order to improve the performance of our proposed GNN model training it using reinforcement learning is an obvious starting point. Since the reward function of performing consensus for multi-robots is straight forward, and also exposing the network to a lot of different graph topologies will practically expose the GNN model into more diverse, bigger data distribution: integrating the GNN within the DQN is an interesting starting point to improve it's performance.

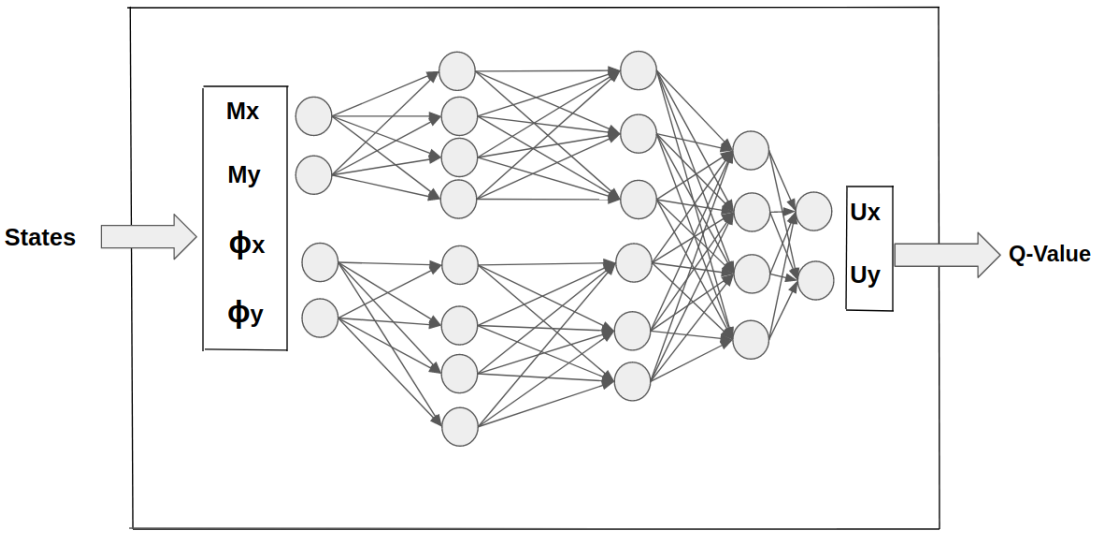

FIGURE D.1: DQN with MLP Architecture

As shown in the figure, since usually DQN models have an embedded neural network within it we can integrate the developed GNN model and train it. In this thesis, as mentioned in the previous appendix C we developed the custom Gym environment from OpenAI in order to do the training as well as wrote the main function to train the GNN.

There are other possible reinforcement learning algorithms to be used, however Q-Learning would be best to start with.

Bibliography

[1] F. Bullo. *Lectures on Network Systems*. 1 edition, 2020.

[2] M. Damani, Z. Luo, E. Wenzel, and G. Sartoretti. Primal$_2$: Pathfinding via reinforcement and imitation multi-agent learning - lifelong. *IEEE Robotics and Automation Letters*, 6(2): 2666–2673, 2021. doi: 10.1109/LRA.2021.3062803.

[3] E. David and K. Jon. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, USA, 2010. ISBN 0521195330.

[4] V. R. Desaraju and J. How. Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees. *2011 IEEE International Conference on Robotics and Automation*, pages 4956–4961, 2011.

[5] M. Everett, Y. Chen, and J. How. Motion planning among dynamic, decision-making agents with deep reinforcement learning. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3052–3059, 2018.

[6] C. Ferner, G. Wagner, and H. Choset. Odrm* optimal multirobot path planning in low dimensional search spaces. In *2013 IEEE International Conference on Robotics and Automation*, pages 3854–3859, 2013. doi: 10.1109/ICRA.2013.6631119.

[7] P. Frasca, R. Carli, F. Fagnani, and S. Zampieri. Average consensus on networks with quantized communication. *International Journal of Robust and Nonlinear Control*, 19:1787–1816, 2009.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[10] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[11] M. Hüttenrauch, A. oić, and G. Neumann. Deep reinforcement learning for swarm systems. *J. Mach. Learn. Res.*, 20:54:1–54:31, 2019.

[12] H. Lezzaik and G. Notomista. *Code for GNN and RL for Multi-agent Systems*. Github, 2021. https://github.com/HusseinLezzaik/Multi-agent-path-planning.

[13] Q. Li, F. Gama, A. Ribeiro, and A. Prorok. Graph neural networks for decentralized multi-robot path planning. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11785–11792, 2020.

[14] W. Lu, F. Atay, and J. Jost. Consensus and synchronization in discrete-time networks of multi-agents with stochastically switching topologies and time delays. *Networks Heterog. Media*, 6:329–349, 2011.

[15] J. McLurkin. *Multi-Robot Systems Engineering*. https://people.csail.mit.edu/jamesm/images/swarm/images/theswarm.jpg.

[16] A. Menzli. *Graph Neural Network and Some of GNN Applications – Everything You Need to Know*. https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications.

[17] M. Mesbahi and M. Egerstedt. Graph theoretic methods in multiagent networks. In *Princeton Series in Applied Mathematics*, 2010.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[20] G. Notomista. *Hemingwayian unicycle*. https://www.gnotomista.com/ancillaries.html#hemingwayianunicycle.

[21] R. Olfati-Saber and R. Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on Automatic Control*, 49:1520–1533, 2004.

[22] R. Olfati-Saber, J. Fax, and R. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95:215–233, 2007.

[23] J. Paulos, S. W. Chen, D. Shishika, and V. R. Kumar. Decentralization of multiagent policies by learning what to communicate. *2019 International Conference on Robotics and Automation (ICRA)*, pages 7990–7996, 2019.

[24] W. B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley-Interscience, USA, 2007. ISBN 0470171553.

[25] M. Rotkowitz and S. Lall. A characterization of convex problems in decentralized control$^a st$. *IEEE Transactions on Automatic Control*, 51(2):274–286, 2006. doi: 10.1109/TAC.2005.860365.

[26] R. O. Saber and R. M. Murray. Consensus protocols for networks of dynamic agents. 2003.

[27] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/schulman15.html.

[28] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2016.

[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[30] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[31] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

[32] C. Szepesvari. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010. ISBN 1608454924.

[33] TensorFlow. *Introduction to RL and Deep Q Networks*. https://www.tensorflow.org/agents/tutorials/0_intro_rl.

[34] E. V. Tolstaya, F. Gama, J. Paulos, G. J. Pappas, V. R. Kumar, and A. Ribeiro. Learning decentralized controllers for robot swarms with graph neural networks. In *CoRL*, 2019.

[35] J. van den Berg, M. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *2008 IEEE International Conference on Robotics and Automation*, pages 1928–1935, 2008. doi: 10.1109/ROBOT.2008.4543489.

[36] M. Wang. *Deep Q-Learning Tutorial: minDQN*. https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc.

[37] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. D. Freitas. Sample efficient actor-critic with experience replay. *ArXiv*, abs/1611.01224, 2017.

[38] H. L. Yisong Yue. *Imitation Learning Workshop*. ICML, 2018. https://sites.google.com/view/icml2018-imitation-learning/.

[39] J. Yu and S. M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, page 1443–1449. AAAI Press, 2013.

[40] W. Yu, G. Chen, and M. Cao. Consensus in directed networks of agents with nonlinear dynamics. *IEEE Transactions on Automatic Control*, 56(6):1436–1441, 2011.