SwarmNet: A Graph Based Learning Framework for Creating and Understanding Multi-Agent System Behaviors

by

Siyu Zhou

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved November 2020 by the
Graduate Supervisory Committee:

Heni Ben Amor, Co-Chair
Sara Imari Walker, Co-Chair
Paul Davies
Ted Pavlic
Steven Presse

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

A *swarm* describes a group of interacting agents exhibiting complex collective behaviors. Higher-level behavioral patterns of the group are believed to emerge from simple low-level rules of decision making at the agent-level. With the potential application of swarms of aerial drones, underwater robots, and other multi-robot systems, there has been increasing interest in approaches for specifying complex, collective behavior for artificial swarms. Traditional methods for creating artificial multi-agent behaviors inspired by known swarms analyze the underlying dynamics and hand craft low-level control logics that constitute the emerging behaviors. Deep learning methods offered an approach to approximate the behaviors through optimization without much human intervention.

This thesis proposes a graph based neural network architecture, *SwarmNet*, for learning the swarming behaviors of multi-agent systems. Given observation of only the trajectories of an expert multi-agent system, the SwarmNet is able to learn sensible representations of the internal low-level interactions on top of being able to approximate the high-level behaviors and make long-term prediction of the motion of the system. Challenges in scaling the SwarmNet and graph neural networks in general are discussed in detail, along with measures to alleviate the scaling issue in generalization is proposed. Using the trained network as a control policy, it is shown that the combination of imitation learning and reinforcement learning improves the policy more efficiently. To some extent, it is shown that the low-level interactions are successfully identified and separated and that the separated functionality enables fine controlled custom training.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A *swarm* describes a group of interacting agents exhibiting complex collective behaviors. Higher-level behavioral patterns of the group are believed to emerge from simple low-level rules of decision making at the agent-level. With the potential application of swarms of aerial drones, underwater robots, and other multi-robot systems, there has been increasing interest in approaches for specifying complex, collective behavior for artificial swarms.

Traditionally, a common approach to creating swarm behaviors for artificial systems is to manually design the underlying logic through careful analysis and synthesis, sometimes taking inspiration from biological swarms such as a bird flock or honey bee colony. In one of the defining works of swarm systems, Reynolds (1987) describes the rules of interactions of autonomous agents for imitating bird-like flocking behavior. Dubbed *Boids*, these bird-like agents engage in reduced rules of interactions within the flock and with external obstacles that appear to reproduce flocking patterns seen in some natural swarms. A critical insight that is introduced in Reynolds (1987) is the concept of compositionality – complex behavioral patterns of the swarm are generated through a superposition of a set of simple rules. Each rule of interaction causes an acceleration, before all vectors are superposed as the steering acceleration for the boid's change in velocity. Alternatively, particle-system based swarm models exist in different fields of study. For example, Vicsek *et al.* (1995) assumes constant speed and matches the orientation of a particle to the average of those of its neighbors; Balch and Hybinette (2000) models formation of unmanned ground vehicles under the influence of potential field rather than forces; Helbing *et al.* (2000) discusses a more sophisticated repulsive force function for the collision avoidance mechanism in human crowd. More recently, graph-theoretic Mesbahi and Egerstedt (2010) approaches and methods based on cost functions and auctioning processes Lagoudakis *et al.* (2004) have become prevalent.

Accurately describing the interactions rely on handcrafted functions using domain knowledge and observation of the dynamics. From the perspective of applications in the real world, developing control strategies that allow a group of agents to perform a task jointly is considered to be an more complex challenge. This statement is particularly true because mobile agents need to dynamically act in different physical space. However, since the advent of deep learning, neural network as a

general function approximator (Tianping Chen and Hong Chen, 1995) has enabled fitting suitable functions directly from observational data without specific domain knowledge.

Understanding interactions between elements of a complex system plays an important part in determining the dynamics. With an emphasis on the relations and interactions, swarm systems are naturally represented as graphs. Each element being a node in the graph representation, interaction between two elements is denoted by en edge between the two corresponding nodes. The influence from one element to the other is carried by a directional edge from the source node to the target node, whereas undirected edge is sufficient to represent the mutual influence between two nodes if the interaction is undirected, or is the same for both directions.

Under this formulation, evolution of the system states can be manifested of information flow from one node to the other through the edges. What accompany the graph formalism are locality and universality, that each node is only directly affected by the immediate neighborhood, and that nodes of the same type follow the same set of rules within their respective neighborhood across the graph. For example, in a swarm model for a colony of ants, the decision making process for each ant involves only the ants that has close contact with it at any time besides environmental context, and despite that soldiers and workers behave differently, all worker ants react to the interactions in the same way.

In contrast to the manual design approach with preset rules of interactions, our approach allows for behavior generation without reliance on a human expert. Instead, the underlying rules are either acquired through the imitation of demonstrations or iteratively optimized through trial-and-error. In this thesis, we discuss a graph based learning framework *SwarmNet* on modeling of multi-agent swarming behaviors. Compared to simple fully connected neural networks, SwarmNet utilize the graph representation of the systems, and model the interactions between nodes through edges directly. We show that composing the neural network around the structure of the system's underlying dynamics introduces significant understanding of the dynamics by the network. This is shown with the prediction accuracy of the motion of a multi-agent system in Chapter 3. Strong evidence is also presented in Chapter 5 that the SwarmNet components are capable of identifying and separating the low-level behaviors constituting the overall dynamics with exposure to only the top level behaviors.

To begin with, we briefly introduce the historical development of graph neural network (GNN) and key insights in the following Chapter 2. Chapter 3 presents our implementation of a GNN named SwarmNet (Zhou *et al.*, 2019) and discusses its effectiveness in imitating multi-agent behaviors and

predicting their motions. Chapter 4 addresses the challenge in GNN's ability of scaling to the number of nodes, which has been overlooked in research. Chapter 5 we propose a training scheme that combines imitation learning and reinforcement learning and how the inherent understanding of the interactions by the SwarmNet affects the outcome of learning.

Chapter 2

OVERVIEW OF GRAPH NEURAL NETWORKS

2.1 General Formulation of Graph Neural Networks

Graph Neural Network (GNN) is a class of neural network structures that explicitly model interactions within data with a graph structure. Within its relative short history, variants of GNNs have been proposed based on distinct first principles and approximation approaches. Despite the distinctions, these variants draw similarities with each other, and it can be shown that they can be unified by a generic form.

A graph is defined by $\mathcal{G}$ its set of nodes $\mathcal{N}$ and set of edges $\mathcal{E}$, $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Each node in $\mathcal{N}$ is identified by a unique index i, and each directed edge is identified by the two node indices it is connected to $(i, j) \in \mathcal{N} \times \mathcal{N}$, $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. In the scope of this paper, we interpret (i, j) as an edge from i to j. The neighborhood of i, $\mathcal{N}_i$, is defined as all the nodes that are connected to j, $\mathcal{N}_i = \{j | \forall j, (j, i) \in \mathcal{E}\}$. For undirected graphs, $(i, j) = (j, i)$ is assumed, and both must be found in $\mathcal{E}$ if either exists. (See Fig.) State vector attributed to node i is denoted $\mathbf{v}_i$, and state vector associate to edge (i, j) is represented by $\mathbf{e}_{ij}$.

The first GNN concept was proposed by Scarselli *et al.* (2009), in which edge states are treated

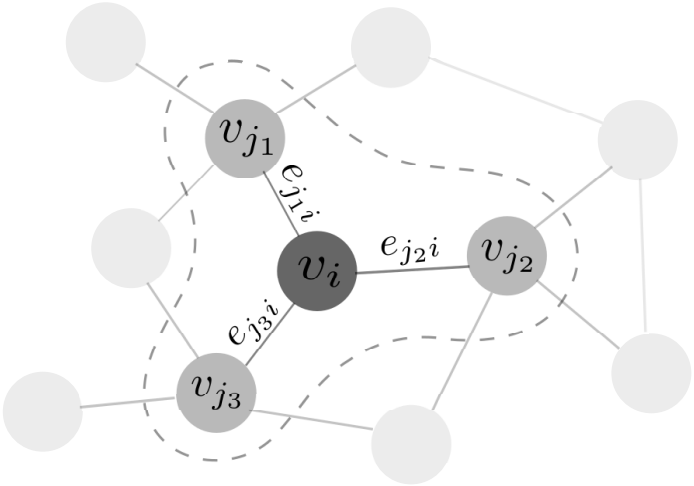

Figure 2.1: Illustration of concepts and notations for nodes, edges and neighborhood in a graph.

as static predetermined labels, and node states are determined by following rules,

$$\mathbf{v}_i = f(\mathbf{V}_i, \mathbf{E}_i) \tag{2.1}$$

where $\mathbf{V}_i$ is the set of the node states of i's neighbor $\mathcal{N}_i$, and $\mathbf{E}_i$ is the set of edge states between i and neighbors $\mathcal{N}_i$, such that $\mathbf{V}_i = \{\mathbf{v}_j | \forall j \in \mathcal{N}_i\}, \mathbf{E}_i = \{\mathbf{e}_{ij} | \forall j \in \mathcal{N}_i\}$. This update rule is applied recursively until convergence from initial states, given the requirement that function f is a contraction map. The node states are then used to read-out output of GNN following

$$\mathbf{o}_i = g(\mathbf{v}_i). \tag{2.2}$$

Both f and g are approximated by a neural network and learned from data. Note that the key assumption for function f and g's being applicable to any node is *universal locality*. That is, the graph is built by nodes that follow the same set of rules within their local neighborhood, with respect to their states.

Allowing edge states to be dynamically updated, Battaglia *et al.* (2018) generalize the update rules to include that of the edges and a global level state vector $\mathbf{u}$:

$$\mathbf{e}'_{ij} = \phi^e(\mathbf{e}_{ij}, \mathbf{v}_i, \mathbf{v}_j, \mathbf{u}) \tag{2.3}$$

$$\mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}_i, \mathbf{v}_i, \mathbf{u}) \tag{2.4}$$

$$\mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}, \mathbf{u}). \tag{2.5}$$

Here, $\mathbf{e}'_{ij}$, $\mathbf{v}'_i$ and $\mathbf{u}'$ represent the updated edge state, node state and global state, respectively. It is important to note that updated states do not necessarily share the same dimension, i.e. they are allowed to have different numbers of features. $\bar{\mathbf{e}}'_i$, $\bar{\mathbf{v}}'$ and $\bar{\mathbf{e}}'$ are introduced as aggregated message, defined as

$$\bar{\mathbf{e}}_i = \rho^e(\mathbf{E}'_i) \tag{2.6}$$

$$\bar{\mathbf{e}}' = \rho^v(\mathbf{E}') \tag{2.7}$$

$$\bar{\mathbf{v}}' = \rho^u(\mathbf{V}') \tag{2.8}$$

where $\mathbf{V}$ simply is the set of all node states and $\mathbf{E}$ is the set of all edge states. In practice, functions $\phi^{\{e,v,u\}}$ are learned by neural networks, and functions $\rho^{\{e,v,u\}}$ are chosen to be simple reduction operations such as summation or average. Detailed discussion on the choice of reduction functions is to be continued in Chapter 4. The global state $\mathbf{u}$ may be interpreted as the context of or external

input to the graph, in which case it is not updated by the graph itself. Note that if global state $\mathbf{u}$ is not involved, in addition to the reduction functions being allowed to be absorbed as part of update functions, there is still a distinction between Eq. (2.1) and Eq. (2.7), which tells whether the previous state of a node is involved in its own update, with Eq. (2.1) being a special case of Eq. (2.7.

The universal local update rules Eq. (2.6)-(2.7) pull 'weighted' information from nodes neighborhood, and is analogous to a sliding filter of convolution when applied across the full graph. Due to this broad resemblance, the series of graph update operations are often branded as Graph Convolution in the literature.

2.2 Implementations of Graph Neural Networks

2.2.1 Spectral Graph Convolution

Earlier works in GNNs take root in graph spectral analysis (e.g. Defferrard *et al.* (2016), Bruna *et al.* (2013)). Spectral-based Graph Convolutions start by decomposing the normalized Laplacian of an undirected graph $\mathbf{L}$ into its eigenvalue matrix $\mathbf{\Lambda} = diag(\lambda)$ and orthonormal eigenvector matrix $\mathbf{U}$.

$$\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T \tag{2.9}$$

Normalized Graph Laplacian is defined as

$$\mathbf{L} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} \tag{2.10}$$

, where $\mathbf{A}$ is the adjacency matrix, symmetrical for undirected graph, and $\mathbf{D}$ is the diagonal degree matrix $\mathbf{D}_{ii} = \sum_k \mathbf{A}_{ik}$. Suppose the node state matrix $\mathbf{V}$ of an order n graph has d features, $\mathbf{V} \in \mathbb{R}^{n \times d}$, and we use $\mathbf{x}_j \in \mathbb{R}^n, 1 \leq j \leq n$ to denote the j-th feature of all nodes. Then for every j, a diagonal filter matrix $\mathbf{g}^j = diag(\theta^j)$, graph convolution updates feature $\mathbf{x}_j$ as follows

$$\mathbf{x}'_j = \mathbf{U}\mathbf{g}^j\mathbf{U}^T\mathbf{x}_j \tag{2.11}$$

The right hand side of Eq. (2.11) shows that node state matrix is Fourier transformed into the spectral space by $\mathbf{U}$ according to the graphs connectivity. The vectors in the spectral space are weighted by filters θ before being transformed back to state space as the updated states. Considering $\mathbf{g}^j$ and $\mathbf{\Lambda}$ both being diagonal matrices, there exists a map f^j such that $\mathbf{g}^j = f^j(\mathbf{\Lambda})$. Immediately, we have

$$\mathbf{x}'_j = \mathbf{U}f^j(\mathbf{\Lambda})\mathbf{U}^T\mathbf{x}_j = \mathbf{U}f^j(\mathbf{U}^T\mathbf{L}\mathbf{U})\mathbf{U}^T\mathbf{x}_j = f^j(\mathbf{L})\mathbf{x}_j \tag{2.12}$$

When viewed as polinomial expansion, each term of $f^j(\mathbf{L})$ can be understood as repeatedly applying graph Laplacian $\mathbf{L}$ to the state matrix. And by virtue of Eq. (2.10), equivalently feature j of node i after one multiplication of $\mathbf{L}$ is transformed to

$$\mathbf{v}'_{i,j} = \mathbf{v}_{i,j} - \frac{1}{\sqrt{d_i}} \sum_{j \in \mathcal{N}_j} \frac{\mathbf{v}_{i,j}}{\sqrt{d_j}} \tag{2.13}$$

where d_i is the ith diagonal element of $\mathbf{D}$, the degree of node i. Eq. (2.13) obviously presents itself as a special linear case of Eq. (2.7), and spectral graph convolution dictated by Eq. (2.11) consists of a mixed number, up to infinity, of "layers" of linear graph convolution under Eq. (2.13).

Nevertheless, Eq. (2.11) forgoes the neat benefit of universal locality, that learned filter $\mathbf{g}^j$ is restricted to graphs of the same order. As pointed out in Kipf and Welling (2016), matrix decomposition in Eq. (2.10) on top of the matrix multiplication on the whole graph is extremely expensive. Instead, $\mathbf{g}^j = f^j(\mathbf{\Lambda})$ can be well approximated by K-th order expansion in Chebyshev polynomials. Kipf and Welling (2016) further truncates the expansion to just 1st order, leaving

$$\mathbf{x}_{j'} = \theta_0^j \mathbf{x}_j - \theta_1^j (\mathbf{L} - \mathbf{I}) \mathbf{x}_j = (\theta_0^{j\prime} \mathbf{I} - \theta_1^{j\prime} \mathbf{L}) \mathbf{x}_j \tag{2.14}$$

Note that this approximation can also be arrived at by any first-order polynomial expansion without resorting to Chebyshev polynomials. It can then be argued that successive application of Eq. (2.14) spawns the higher order terms that may deem essential for accurate approximation.

Eq. (2.14) involves sparse matrix multiplication, as well as restoring the universal locality. In its more generic matrix form, with each of the d features being mapped to d' new features, we have

$$\mathbf{V}' = \sigma(\mathbf{V}\mathbf{\Theta}'_0 - \mathbf{L}\mathbf{V}\mathbf{\Theta}'_1) \tag{2.15}$$

where $\mathbf{\Theta}'_0, \mathbf{\Theta}'_1 \in \mathbb{R}^{d \times d'}$. The non-linear element-wise activation function σ is put in-place to enable the ability to approximate non-linear rules when the equation is applied successively.

2.2.2 Spatial Graph Convolution

As mentioned earlier, one of the motivations in the development is to directly extend the success of locality and transnational invariance in CNNs to non-grid-like graph data structures. These principles led to a class of graph convolutions now commonly regarded as Spatial Graph Convolution. Similar to a filter in CNN, the receptive field of a node contains the closest neighbors defined by graph connectivity. A weighted sum of node states in the neighborhood forms the state of the central node in the receptive field for the next graph layer, much like that the weighted sum of pixels by

a CNN kernel produces a central pixel for the next CNN layer. These implementations in general adopt the following as the update rule for node states:

$$\mathbf{v}_i' = f(\sum_{j \in \mathcal{N}_i^+} \mathbf{v}_j w_{ji}) \tag{2.16}$$

where w_{ji} is the learnable weight associated with the edge (j, i), $\mathcal{N}_i^+ = \mathcal{N}_i + \cup\{i\}$ and f is a learnable function, usually non-linear. For a simple case of f where it can be reduced to a linear transformation $\mathbf{\Theta}$ and an element-wise non-linear activation σ, with all $w_{ji} \equiv 1$, it can be written in matrix form

$$\mathbf{V}' = \sigma(\mathbf{V}\mathbf{\Theta} + \mathbf{A}\mathbf{V}\mathbf{\Theta}) \tag{2.17}$$

Various works incrementally build on top of Eq. (2.16), and introduce inductive biases to weights (e.g. Xu *et al.* (2019)), size of the immediate receptive field (e.g. Tran *et al.* (2018)), or how previous layers contribute to the update for the next layer (e.g. Micheli (2009)).

It can be readily seen that when $\mathbf{\Theta}_0 = -\mathbf{\Theta}_1$, Eq. (2.15) and Eq. (2.17) differ only by the detail that adjacency matrix in Eq. (2.15) is normalized. However, learnable parameters in Eq. (2.17) may adapt to automatically compensate the scaling factors. Although spectral graph convolution and spatial graph convolution were developed out of distinct principles, the resemblance in the linear operations between their approximate formulations strikes an impression that the distinction between the spectral and the spatial is no longer necessary in practice, and we now refer both as graph convolution.

2.2.3 Graph Attention Network

Attention mechanism has been known as a mechanism to assign weights to features according to their relative "importance" (Vaswani *et al.* (2017)). For a set of vectors $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, we define function $a(\mathbf{x}_i, \mathbf{x}_j)$, $a : \mathbb{R}^{2d} \mapsto \mathbb{R}$, as the importance of $\mathbf{x_j}$ with respect to $\mathbf{x_i}$, or more commonly, $\mathbf{x}_i$'s attention on $\mathbf{x}_j$. When attempting to update node state $\mathbf{v}_i$ according to Eq. (2.16), Graph Attention Network (GAT) proposed by Veličković *et al.* (2018) replaces the weights w_{ji} with attention:

$$w_{ji} = a(\mathbf{v}_i, \mathbf{v}_j) \tag{2.18}$$

An additional requirement is imposed in GAT that all attention of the same node be summed to 1. Thus, w_{ji} for all $j \in \mathcal{N}_i^+$ are regulated by a *softmax* operation.

$$\alpha_{ij} = \text{softmax}(w_{ji})|_i = \frac{\exp(w_{ji})}{\sum_{j \in \mathcal{N}_i^+} \exp(w_{ji})} \tag{2.19}$$

with the update rule Eq. (2.16) being substituted by

$$\mathbf{v}_i' = f(\sum_{j \in \mathcal{N}_i^+} \mathbf{v}_j \alpha_{ij}) \tag{2.20}$$

Concretely, the original GAT implements a minimal non-linear function as a, parameterized by a learnable matrix $\mathbf{W}$ and a learnable projection vector $\mathbf{a}$.

$$a(\mathbf{v}_i, \mathbf{v}_j) = \sigma(\mathbf{a}^T[\mathbf{W}\mathbf{v}_1 || \mathbf{W}\mathbf{v}_2]) \tag{2.21}$$

where $||$ stands for vector concatenation, and σ is an element-wise non-linear activation.

The GAT network also draws inspiration from Vaswani *et al.* (2017) to employ *multi-headed* attention, in which multiple independent sets of K functions $a^k, 1 \leq k \leq K$ and corresponding parameters are created for each layer. The K instances of updated states are concatenated,

$$\mathbf{h}_i' = ||_{k=1}^K f(\sum_{j \in \mathcal{N}^+} \alpha_{ij}^k \mathbf{v}_j) \tag{2.22}$$

Here the symbol $||_{k=1}^K$ denotes concatenation for vectors with $k \in \{1, 2, ..., K\}$. And the updated state $\mathbf{v}_i'$ is reduced from $\mathbf{h}_i'$ by another learnable function g,

$$\mathbf{v}_i' = g(\mathbf{h}_i') \tag{2.23}$$

which in the case Veličković *et al.* (2018) is simply a mean reduction function,

$$\mathbf{v}_i' = \frac{1}{K} \sum_{k=1}^K f(\sum_{j \in \mathcal{N}^+} \alpha_{ij}^k \mathbf{v}_j) \tag{2.24}$$

2.2.4 Message Passing Neural Network

When comparing spatial graph convolution to the generic form of graph update Eq. (2.6)-(2.7), we notice that spatial graph convolution does not contain an explicit edge state. Rather, $\mathbf{v}_j w_{ji}, j \in \mathcal{N}_i^+$ pretends to be the state of edge (j, i), a message sent from node j of its weighted node state. Message Passing Neural Network (MPNN) (Gilmer *et al.* (2017)) grows on this idea of *message passing* between nodes, and interprets edge state as message sent between two connected nodes, whereas node update being message aggregation from incoming edges.

$$\mathbf{e}_{ij}' = \phi^e(\mathbf{e}_{ij}, \mathbf{v}_i, \mathbf{v}_j) \tag{2.25}$$

$$\mathbf{v}_i' = \phi^v(\mathbf{v}_i, \sum_{j \in \mathcal{N}_i} \mathbf{e}_{ji}) \tag{2.26}$$

MPNN arguably enjoys more expressive power as it is able to describe more generic interactions by recognizing that interaction between two nodes should depend on the states of both. In that regard, the modeling of neighborhood influence by graph convolution such as Eq. (2.16) regardless the central node is possibly less accurate. Despite the involvement of the central node in attention computation, the outcome of the attention is a scalar that linearly rescales only the state of the source node. Although the advantage of expressive capacity of MPNN has not been thoroughly investigated, significant in performance gain on relevant tasks over simpler multi-layer graph attention convolution is not evident in the literature. One possible explanation could attribute this lack of preference to the fact that stacked layers GAT with multi-headed attention might help compensate the relatively weak non-linearity of Eq. (2.20).

In the scope of this paper, we use MPNN's formulation as our graph convolution of choice, due to its intuitive description of interaction in the physical systems of our interests, as well as it being the closest reproduction of the general form of graph update Eq. (2.6)-(2.7).

Chapter 3

MODELING MULTI-AGENT SYSTEMS WITH GRAPH NEURAL NETWORK WITH IMITATION LEARNING

A swarm system consists of a large number of agents which perform tasks through interactions between agents and interactions with the environment. Complex high-level collective behaviors of the group are believed to emerge from simple low-level rules of interactions and decision making processes on agent-level. Nature presents ample examples of swarm systems, such as ant colonies, fish schools, bird flocks, etc. And engineering of artificial swarm systems often take inspiration from the biological counterparts.

In the Boid model, Reynolds (1987) describes the rules of interactions of autonomous agents for imitating the collective flocking behaviors of a bird flock. Dubbed *Boids*, these bird like agents engage in reduced rules of interactions within the flock and with external obstacles. Similar to particle simulation, each rule of interaction causes an acceleration, before all vectors are superposed as the steering acceleration for the boid's change in velocity. Confronting the obstacle avoidance behavior, they present the choice of using a force field model versus a "steering-to-avoid" one, of which the latter is preferred as a more realistic simulation, though the former is reserved for modeling inter-agent separation. Reynolds (1999) further extends the rule set in an attempt to include more elements (e.g. the roles of predator and prey) as well as to address more subtle underlying interactions. Because of the superposition principle employed in the particle system, one may select a subset of the rules and form a less complex approximated model. Of particular interest to the scope of research presented in this thesis are environments consisting of numerous boids, external obstacles, and a goal the agents seek to reach. Thus the low-level behaviors involved according to the Boid model include, agent cohesion, alignment, separation (collision avoidance), obstacle avoidance and goal seeking. (See Fig. 3.1.)

Alternative particle system based swarm models exist in different areas of application. For example, Vicsek *et al.* (1995) assumes constant speed and matches the orientation of a particle to the average of those of its neighbors; Balch and Hybinette (2000) models formation of unmanned ground vehicles under the influence of potential field rather than forces; Helbing *et al.* (2000) engineers a more sophisticated function for the collision avoidance mechanism in human crowd. However, Reynolds

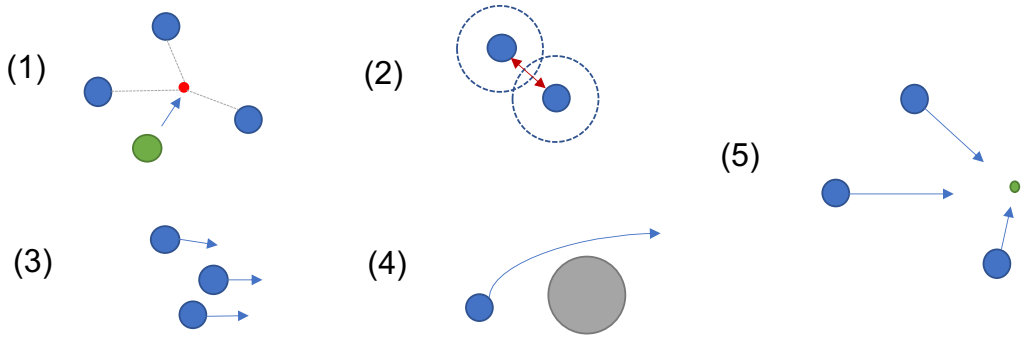

Figure 3.1: Low level behaviors included in the Boid model. (1) Cohesion: agents try to converge to neighbors' average position. (2) Separation: agents try to keep a certain distance away from each other to avoid collision. (3) Alignment: agents try to match the velocity of the neighbors. (4) Obstacle avoidance: agent steers away from obstacles. (5) Goal seeking: individual agents are attracted to the common goal.

(1999) is arguably the most generic framework, as the above mentioned models either focus on a subset of the interactions, or introduce modification to the actual interaction functions.

In the perspective of application in real world, developing control strategies that allow a group of agents to perform a task jointly is considered to be a complex challenge. Different from the manual design approach with preset rules of interactions, the *machine learning* approach allows behavior generation without reliance on human intervention. Instead, the desired underlying rules are either approximated by a program from the observation of demonstrations, or optimized iteratively through self-exploration under the guidance of an evaluation metric. The former, being called *imitation learning*, is a class of supervised learning. Behaviors generated by the program is compared with a collection of demonstrations as *ground truth*, and the parameters are optimized to achieve a minimal discrepancy between the two. The latter falls under the banner of *unsupervised learning*, with common implementations such as *evolutionary algorithms* and *reinforcement learning*. Learning of swarm systems presented in this thesis approaches the solution through both imitation learning and reinforcement learning.

In this part, we present an approach for learning swarm coordination through imitation. Rather than specifying the system dynamics, the designer only has to provide demonstrations of successful coordination behavior. In turn, this data set is used to learn a neural network representation of the underlying dynamics and rules of interaction. More specifically, we leverage recent insights regarding graph neural networks to learn a structured model of the dynamics. An implementation example of GNN can be seen in Kipf *et al.* (2018), where the ability of GNN to better predict the motion of a

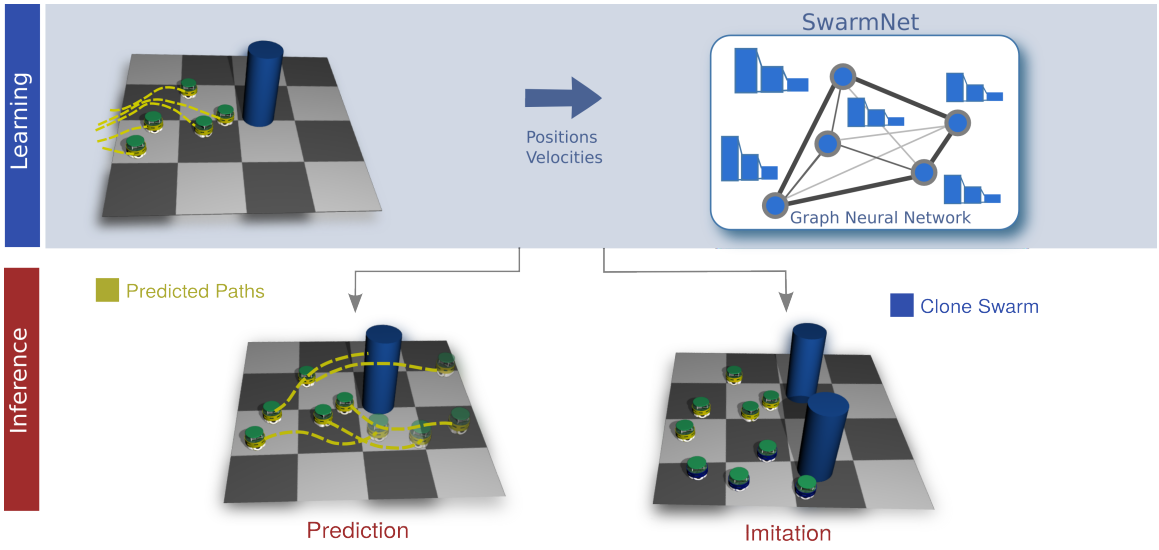

Figure 3.2: Overview of the proposed methodology: we collect training data from an observed swarm to learn a compact graph neural network representation called SwarmNet. In turn, it can be used to predict future behavior, augment an existing swarm with more agents, or create a clone of the swarm with similar behavior.

simple physically interacting particles is highlighted. We show that the specific GNN implemented in this paper can accurately capture the dynamics of more complex swarm motion behaviors given a set of demonstrations. We also discuss and analyze difficulties in scaling learned models up to a swarm with a larger set of agents. Based on this discussion, we then propose a refinement training procedure to address these issues. The refinement procedure helps tuning a learned model to a system with a different number of agents, i.e., scalability along the size of the swarm.

3.1 Model Implementation

In this section, we describe our methodology for imitation learning of swarm behavior. Fig. 3.2 depicts an overview of both the training and inference process for swarm. The input to the learning process is a set of demonstration trajectories, or execution traces, of the agents in the swarm. Execution traces are discretely sampled trajectories specifying the position and velocity of each agent at time step t. These trajectories are used to train a graph neural network to model the group behavior observed in the swarm. Our network takes the time series as input and generates the predicted next steps in the time series. The prediction error is measured by the mean squared error between output states and expected states from the true time series. The task of our network is to

model the motion dynamics of the swarm with minimum prediction error. Without the assumption that the dynamics is first-order Markovian, states earlier than the present state carry predictive power for future states and must be accounted for. Historical states and their higher order relations with the current states, for example, velocity and acceleration, are necessary for making accurate predictions. Therefore, we apply a series of one-dimensional convolutional layers to process a history states for the agents. The 1D convolution acts along the time steps of each agent and abstracts a representation of its past as the starting point for prediction. This representation will then be taken as the input of the GNN.

The interaction of the swarm system being embedded in a graph, the GNN directly models the information propagation within the graph by explicitly computing the interaction between nodes and account for the influence from the interactions to each node to update the graph's state. The procedures can in general be identified as 3 steps. Firstly, interactions between all pairs of nodes connected through an edge are computed by a function universal across all edges. The interaction, often called edge state, is determined only by its current state and the two nodes at both ends. Secondly, centered on each individual node, all interactions directed to it are aggregated by another function universal to all nodes to form the local influence to the node. This step is sometimes called edge aggregation. Lastly, a third function universal to all nodes updates each node state based on the node's current state and aggregated influence, thus completing the state update of the entire network. Evolution of graph dynamics is fully described by applying the procedures repeatedly. Details of structure and training procedures will be explained in the following subsections.

3.1.1 Data Structure

The input time series of motion states from an instance of swarm simulation are arranged to have the shape of $T \times N \times D$, where T is the number of time steps, N is the number of agents, and D is the dimension of the state vector $s_i(t)$ of agent i at step t, where $i, t \in \mathcal{N}$, $i < N, t < T$. The state vector contains both the position and velocity, i.e., $\boldsymbol{s}_i(t) = [\mathbf{x}_i(t), \dot{\mathbf{x}}_i(t)]$, where the square brackets stand for concatenation of the position $\mathbf{x}_i(t)$ and velocity $\dot{\mathbf{x}}_i(t)$. In all subsequent experiments presented in this paper, all agents live in a two-dimensional space, which lets the dimension of state vectors $\mathcal{D} = 4$. A sequence with window length T_w of history states $\boldsymbol{s}_i(t - T_w + 1), \boldsymbol{s}_i(t - T_w + 2), ..., \boldsymbol{s}_i(t - 1), \boldsymbol{s}_i(t)$ are processed to output the next future state $\boldsymbol{s}_i(t + 1)$. Long term prediction is realized by repeatedly appending the predicted new step to the sequence of history state, dropping the earlier states and

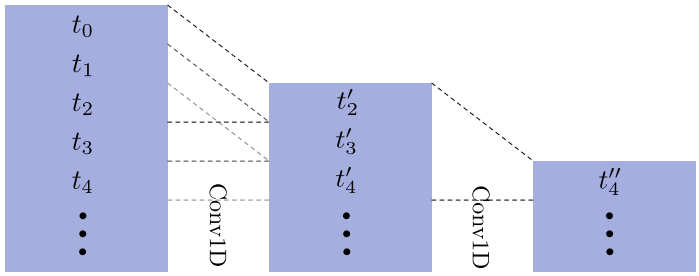

Figure 3.3: 1D Convolution: for a kernel of size $K = 3$, the time series is transformed by replacing each time step with the weighted sum of itself and its 2 previous steps. Since the time series is not padded beyond step 0, every convolution layer removes 2 steps from the beginning. A filter with weights $(0, 0, 1)$ ignores earlier histories and only considers the present time step, thereby implementing a Markov assumption. A filter with weights $(0, -1, 1)$ effectively approximates the first order derivative, and one with $(1, -2, 1)$ approximates the second order derivative, etc. A group of such filters would be sufficient to capture the dynamics of histories.

feeding the new sequence of length T_w as input.

3.1.2 1D Convolutions

The goal of one-dimensional convolutions is to abstract a concise representation of the historical states. The 1D convolutions with no padding condense the time sequences for each agent, until the output after the last 1D convolution layer is reduced to a $1 \times N \times C$ shaped tensor, with C being the number of channels of the last layer. For history window length T_w and a universal kernel size of K, this would require at least $\lceil (T-1)/(K-1) \rceil$ layers since each layer condenses the length by $K - 1$. We typically choose the number of 1D convolution layers L and T_w such that they satisfy $T_w = L(K-1)+1$. $K = 3$ is what we have used throughout the experiments. As the 1D convolutional filters slide through the entire time series of trajectories of length T, $T_s = T + 1 - T_w$ sub-sequences are condensed, each serving as the basis for prediction, and are sent to the following Graph Neural Network module to produce the predicted states for their corresponding next steps.

3.1.3 Graph Neural Network

Swarms along with their interactions are embedded in a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes which represent the agents, and $\mathcal{E} = \{(i, j) | i, j \in \mathcal{N},\ i \neq j\}$ is the set of edges whose element (i, j) represents the influence from agent i to agent j through interaction. The state of the node $i \in \mathcal{N}$ is a vector of dimension d_v, as $\boldsymbol{v}_i \in \mathbb{R}^{d_v}$, while the state of an edge $(i, j), i, j \in \mathcal{N}\, i \neq j$ is a vector of dimension d_e, that is, $\boldsymbol{e}_{ij} \in \mathbb{R}^{d_e}$. In swarm motion, such interactions include pulling, pushing and steering, and the edge states may be interpreted as an abstraction of the physical forces.

For each of the T_s slices from the output of 1D convolution module, the GNN module takes the $N \times C$ tensor as states of the N nodes. The node state of the next step inevitably is influenced by the interactions the node experiences. The interactions depend on the current and previous states of the involved nodes. As introduced earlier, this information passing process is realized by GNN, formulated by 3 functions below:

$$\boldsymbol{e}_{ij} = \phi^e(\boldsymbol{v}_i, \boldsymbol{v}_j), \tag{3.1}$$

$$\bar{\boldsymbol{e}}_i = \psi^{\bar{e}}(\sum_{j \in \mathcal{N}_i} \boldsymbol{e}_{ji}), \tag{3.2}$$

$$\boldsymbol{v}'_i = \phi^v(\boldsymbol{v}_i, \bar{\boldsymbol{e}}_i), \tag{3.3}$$

As a special case of Eq. (2.25), function ϕ^e, computes the influence from node i to node j. Function $\psi^{\bar{e}}$ aggregates the influences from all sources for node i as the total influence $\bar{\boldsymbol{e}}_i$. And finally, function ϕ^v transforms the total influence and the node's historical states into the prediction of the next step. Note we introduce the extra function $\psi^{\bar{e}}$ to create a more generic aggregation operation than a sum reduction. Although it can be argued that $\psi^{\bar{e}}$ may be absorbed as part of ϕ^v mathematically, we prefer to separate the two semantically different parts for easier scaling analysis. All three functions are each approximated by a multi-layer perceptron (MLP).

In principle, the group of GNN operations may be stacked one after another just like layers of multi-layer perceptron, where the output vector $\boldsymbol{v}'_i$ of intermediate layers would serve as hidden states and only the output of the last layer is considered the update for the next step. In that case, interaction horizon gets expanded from the nearest neighbors, since information propagates further down the edges with each GNN layer added. Nevertheless, in this paper, we only employ one GNN layer in our model, as is appropriate for the physical settings of a swarm system.

It is worth pointing out that $\mathbf{v}i$ and $\mathbf{v}'_i$, despite both being the "node state" before and after GNN operations, may carry different meanings, as in the practice of this part where we treat the output $\mathbf{v}'_i$ as the predicted *change* between the current state vector $\mathbf{s}_i(t)$ and future state vector $\mathbf{s}_i(t+1)$ of agent i, $\mathbf{s}_i(t+1) = \mathbf{v}' + \mathbf{s}_i(t)$, while we assign the representation of history states to $\mathbf{v}_i$. The core algorithm of our GNN is presented in Algorithm 1. Data flow is shown in Fig. 3.4. We name the architecture of the neural network *SwarmNet*.

3.1.4 Loss Function and Training

Prediction error is measured as the mean squared error (MSE) between the predicted states $\mathbf{s}_i(t+1)$ and the ground truth $\mathbf{s}^*_i(t+1)$ from motion data. Supervised training is performed to

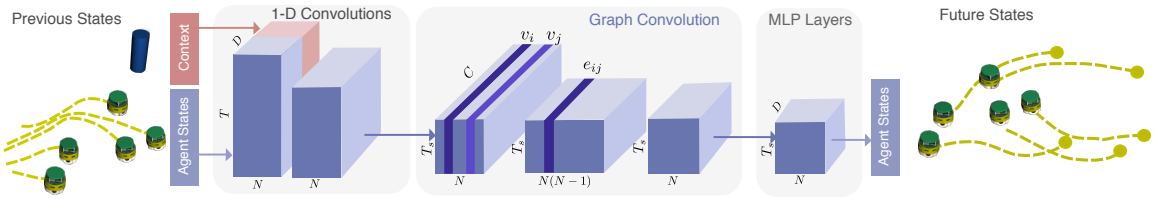

Figure 3.4: Data flow through modules of the neural network. Letters along sides indicate dimensions. T is the number of time steps, N is the number of agents in the system, and D is the length of the state vector. H is the size of the encoded state vector, which is the size of the last layer of an MLP. Node states v_i and v_j are passed through Graph Convolution to produce interactions e_{ij}

Data: History states $\mathbf{S}_{(1:T_w,1:N,1:D)}$, Next state $\boldsymbol{S}^*_{(1:N,1:D)}$, Edge type one-hot labels $\mathbf{E}_{(1:N,1:N,1:T_E)}$

Result: Conv1D parameters w, Function $\phi^e, \psi^{\bar{e}}, \phi^v$ parameters $\theta_1, \theta_2, \theta_3$

Random initialize $w, \theta_1, \theta_2, \theta_3$ $\boldsymbol{S}'_{(1:N,1:D)} \leftarrow \mathbf{S}_{(T_w,1:N,1:D)}$ `// Current state`

$\boldsymbol{V}_{(1:N,1:D)} \leftarrow \text{Conv1D}(\mathbf{S}_{(1:T_w,1:N,1:D;w)})$ **for** i *in* $1:N$ **do**

  $\boldsymbol{v} \leftarrow \boldsymbol{V}_{(i,1:D)}$ $\bar{\boldsymbol{e}} \leftarrow \mathbf{0}$

  for j *in* $1:N$ **do**

    $\boldsymbol{u} \leftarrow \boldsymbol{V}_{(j,1:D)}$ **for** k *in* $1:T_E$ **do**

      $\boldsymbol{e} \leftarrow \phi^e(\boldsymbol{v}, \boldsymbol{u}) * \mathbf{E}_{(i,j,k)}$

      $\bar{\boldsymbol{e}} \leftarrow \bar{\boldsymbol{e}} + \boldsymbol{e}$

    end

    $\bar{\boldsymbol{e}} \leftarrow \psi^{\bar{e}}(\bar{\boldsymbol{e}})$

  end

  $\boldsymbol{v}' \leftarrow \phi^v(\boldsymbol{v}, \bar{\boldsymbol{e}})$

  $\boldsymbol{S}'_{(i,1:D)} \leftarrow \boldsymbol{S}'_{(i,1:D)} + \boldsymbol{v}'$

end

$L \leftarrow \text{MSE}(\boldsymbol{S}'_{(1:N,1:D)}, \boldsymbol{S}^*_{(1:N,1:D)})$

$(w, \theta_1, \theta_2, \theta_3) \leftarrow \text{Adam}(L, (w, \theta_1, \theta_2, \theta_3))$

Algorithm 1: A training step on swarm dataset

adjust the parameters in the functions by minimizing the MSE between $\mathbf{s}_i(t+1)$ and $\mathbf{s}^*_i(t+1)$ over all agents and all time steps.

Given that the simulated motion data is discretely sampled from trajectories, how fine grained the sample rate is affects the manifest of the error. Because of this, the error is normalized by the average "natural skip" in the ground truth data, i.e., the MSE of state vectors between two

consecutive steps in the ground truth trajectories, $\bar{L}$,

$$\bar{L} = \frac{1}{2DN(T-1)} \sum_{t=1}^{T-1} \sum_{i=1}^{N} \left(\boldsymbol{s}_i^*(t+1) - \boldsymbol{s}_i^*(t)\right)^2 \tag{3.4}$$

Normalized error $L_{norm} = \frac{L}{\bar{L}}$ rectifies the dependence of prediction error on the scale of the intrinsic spacing in the ground-truth data.

Multistep prediction is enabled by appending the predicted state $\mathbf{s}_i(t+1)$ back to the sequence of T_w states and taking the new last T_w states as the history ground for future prediction. Multi-step prediction builds up predictions in the far future on the predictions in the near future, and thus imposes a higher demand on the accuracy and encourages the model to learn the true mechanisms better, despite harder to train. We gradually increase the difficulty of the prediction task by increasing the required number of prediction steps. This curriculum learning (Bengio *et al.* (2009)) starting with short term prediction helps to guide the model faster in the earlier stage while still granting good long term prediction power later.

3.1.5 Uncertainty, Noise and Nondeterminism

Swarms and multi-robot systems are typically acting in nondeterministic environments in which perceptual data is noisy and the effect of actions uncertain. We extend our methodology to nondeterministic environments by leveraging recent theoretical insights connected to *Bayesian* deep learning. In particular, the work in Gal (2016) shows that estimates of model uncertainty can be generated from a neural network using the Dropout Srivastava *et al.* (2014) algorithm. Dropout is a training algorithm in which connections of a neural network are randomly activated and disactivated. This is achieved using a dropout probability p which describes the probability of an input activation being dropped. After training a network with Dropout, we can use the same technique to generate different outputs for the same input, i.e., disactivate layer input with probablity p. In such a case, each forward pass through the network is called a *stochastic forward pass* and can be seen as a sample from the underlying probability distribution. According to Gal (2016), such stochastic forward passes in a deep network will be an approximation of variational inference in a Gaussian process. In our case, the set of stochastic forward passes $\mathcal{S} = \{\hat{\boldsymbol{s}}_i^1(t+1), ..., \hat{\boldsymbol{s}}_i^S(t+1)\}$ represents S samples from the probability distribution over the future states of agent i, i.e., the distribution defining the range of different states the agent can be as a result of nondeterminism. Recursively sampling for longer horizons of the future generates the anticipated trajectories for all agents along with inherent

	5 Steps		
Method	Boids	Helbing	Chaser
Kipf's GNN	0.4288 ± 0.0182	0.4374 ± 0.0179	0.1391 ± 0.0006
LSTM	0.8992 ± 0.0098	1.2241 ± 0.0098	0.2013 ± 0.0020
SwarmNet	0.2813 ± 0.0012	0.1000 ± 0.0050	$\mathbf{0.0152 \pm 0.0002}$
SwarmNet (Context)	$\mathbf{0.2338 \pm 0.0024}$	$\mathbf{0.0317 \pm 0.0019}$	$\mathbf{0.0152 \pm 0.0002}$
	40 Steps		
Method	Boids	Helbing	Chaser
Kipf's GNN	17.45 ± 1.00	19.61 ± 0.43	14.23 ± 0.05
LSTM	41.20 ± 0.24	42.88 ± 0.75	126.3 ± 3.5
SwarmNet	10.47 ± 0.91	3.855 ± 0.305	$\mathbf{3.691 \pm 0.042}$
SwarmNet (Context)	$\mathbf{2.778 \pm 0.066}$	$\mathbf{0.484 \pm 0.023}$	$\mathbf{3.691 \pm 0.042}$

Table 3.1: Normalized MSE loss for short term prediction (top, 5 steps) and long term prediction (bottom, 40 steps)

uncertainties. In the remainder of this paper, we will refer to versions of our network that use this approach as *SwarmNet*$^+$. It is important to note, that SwarmNet$^+$ can generate multiple different, potentially conflicting or bifurcating predictions for the same input state to the network – a property that is typically not possible in standard neural network training approaches.

3.2 Experiments and Results

We generated training data using popular techniques for synthesizing swarm behavior. In turn, we compared different versions of SwarmNet to the graph neural network (GNN) in Kipf *et al.* (2018) and the LSTM network method in Hochreiter and Schmidhuber (1997). Since SwarmNet shares strong similarities with GNN approaches such as Kipf *et al.* (2018), we tested different ablations and additions to Kipf *et al.* (2018) based on the insights in this paper. In addition, we performed a variety of experiments to investigate sample-efficiency and other critical aspects of SwarmNet.

3.2.1 Swarm Data Sets

Our data set consists of motion data produced using the Boids model (Reynolds, 1999), the Helbing model (Helbing *et al.*, 2000), and a simple chaser model. In the Boids model, $N = 5$ agents

demonstrate flocking behaviors while approaching a common goal location and avoiding obstacles. The Helbing model (Helbing *et al.*, 2000) generates swarm behavior via repulsive forces only. The final model, called chaser, places a set of agents on a circle. Each of these agents generates steering actions that make it chase another agent of the swarm. For each of the above models we generate data sets by running the simulation for about 50 time steps while recording the agent positions and velocities, as well as the environmental variables. In both the Boids and Helbing model, the environmental variable is the positions of obstacles.

3.2.2 Prediction Accuracy

The first experiment focuses on the prediction accuracy when compared to other methods, in particular the method in Kipf *et al.* (2018) and LSTMs (Hochreiter and Schmidhuber, 1997). Table 3.1 summarizes the loss of the methods across the three data sets. We performed the experiment for prediction horizons of 5 steps and 40 steps. All methods were trained with 50K demonstrations of the corresponding swarm behavior. The best performance is achieved by SwarmNet with contextual inputs. Removing the context information has a significant impact on the SwarmNet performance, in particular in the long-term prediction case (right side of the table). On the Boids data set, the error for 40 step predictions jumps from 2.778 to about 10.47 when contextual information is omitted. In general, SwarmNet outperforms all other methods on all data sets and in both conditions. An interesting aspect of these results is that SwarmNet was only trained on data for predictions of up to 10 steps, as previously described in Sec. 3.1.4. Despite that, it correctly learned to make accurate predictions for 40 steps and beyond indicating that it focused on the true dynamics of the task.

Fig. 3.5 shows the prediction results for different locations of the obstacle (black circle). We see that the SwarmNet predictions are qualitatively implementing the correct swarming and avoidance behavior, even if small deviations from the ground truth occur. The rightmost example in Fig. 3.5 shows an interesting behavior in which the orange agent is first trying to circumvent the object on the right side and then turns to head back to goal location. While unintuitive, this behavior is also existent in the ground truth data.

Next, we investigated the influence of the *training set size* and the *prediction horizon* on the prediction results. Fig. 3.6 shows the MSE for five different horizon lengths (1-40 steps into the future) trained with data sets ranging from 100K demonstrations down to only 1K demonstrations.

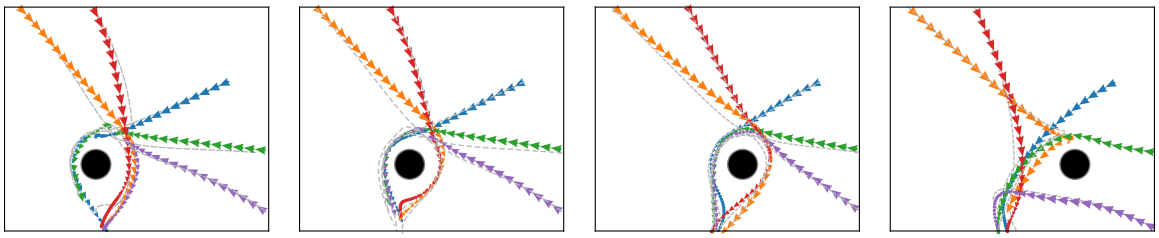

Figure 3.5: From left to right: four different predictions of the swarm behavior using SwamNet. In each one of the four experiments, the obstacle is place slightly more to the right. The colored arrows show the movement of 5 boids, while the gray dashed lines are the ground truth trajectories. The black circle represents the obstacle.

It is interesting to note that, in the case of long-term prediction, only when trained with less than 5K demonstrations does the MSE deteriorate. Even in the case of training on only 1K samples, SwarmNet still generates (qualitatively) reasonable swarm behavior that executes the intended task. Fig. 3.7 shows the behavior of a network trained on 1K examples of the Boids task. The generated predictions still follow the trend of the ground truth data, with one exception in which an agent trajectory (orange) is predicted to go through the obstacle.

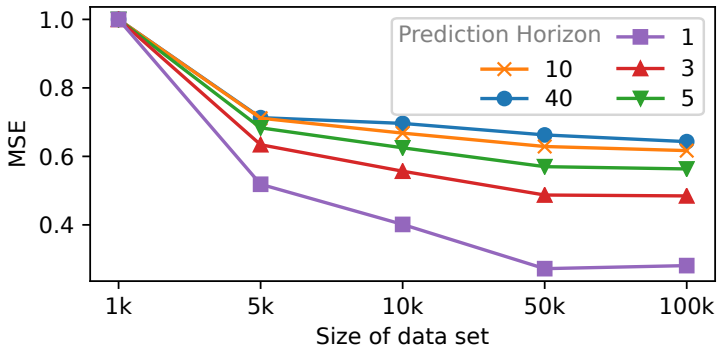

Figure 3.6: Prediction error for different horizon lengths and sizes of training set with boids. MSE errors are normalized by the error with 1k data.

3.2.3 Comparison to the Kipf Model

SwarmNet builds upon recent developments and advances in the field of graph neural networks (Battaglia *et al.*, 2018). In particular, it shares similarities with the method proposed by Kipf *et al.* (2018). However, our method incorporates new components, in particular the 1D convolutions, use of context, and curriculum-based training. To better understand the effects of the individual components of our network, we make changes (ablations and additions) to the original Kipf method to see how they affect performance. First, we noted that SwarmNet can be seen as a

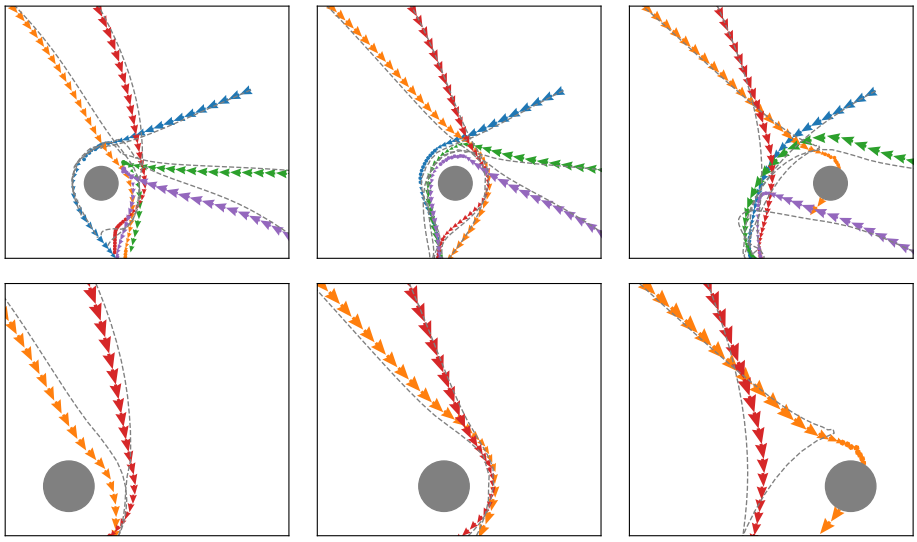

Figure 3.7: Predictions for the behavior of a swarm generated from a network trained on 1000 demonstrations of boids simulation. The top row shows the movements of all agents. The bottom row highlights (for visibility) the movements of only two agents.

variant of only the decoder part of Kipf's approach. Hence, we used the decoder as a starting point and then performed repeated experiments in which we added (a) MLP layers for edge aggregation and context (b) 1D convolutions. Tab. 3.2, shows the results of this comparison. We can see that the decoder of Kipf's model, when combined with both 1D convolutions and context variables, yields MSE error values comparable to our results. The different neural network architectures shown in Tab. 3.2 can be seen as a (discrete) spectrum that shows the effects of gradually transitioning from the original Kipf GNN to our SwarmNet model. Our model still slightly outpaces Kipf's decoder augmented with Conv1D and context. This last difference is due to the curriculum learning approach to training. For a qualitative comparison between SwarmNet predictions and the Kipf GNN predictions, see Fig. 3.8.

3.2.4 Uncertainty

Finally, we also investigated how well the SwarmNet$^+$ extension can deal with uncertainty, noise and nondeterminism. In particular, we simulated both perceptual noise, as well as actuation noise. Perceptual noise was incorporated by adding a value sampled from a univariate normal distribution $\mathcal{N}(0, 1)$ to input for each dimension separately. Actuation noise is simulated by randomly dropping out neurons in test time. Fig. 3.9 depicts the stochastic outputs of our SwarmNet$^+$ model for the

Method	5 Steps	40 Steps
Kipf's GNN	0.4288 ± 0.0182	17.45 ± 1.00
Decoder	0.2654 ± 0.0070	3.639 ± 0.075
Decoder (Context)	0.2717 ± 0.0034	3.873 ± 0.090
Decoder+Conv1D	0.2584 ± 0.0040	3.916 ± 0.388
Decoder+Conv1D (Context)	0.2491 ± 0.0018	3.101 ± 0.046
SwarmNet (Context)	$\mathbf{0.2338 \pm 0.0024}$	$\mathbf{2.778 \pm 0.066}$

Table 3.2: Comparison of MSE error on Boids data of SwarmNet, the original Kipf model, as well as different new models that are created from ablations and additions to the decoder part of the network in Kipf *et al.* (2018).

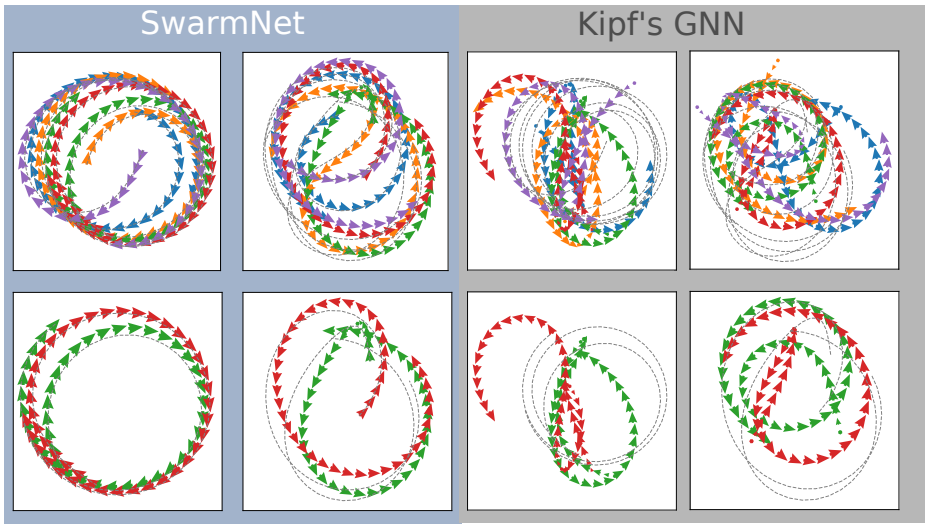

Figure 3.8: Qualitative comparison of SwarmNet and the approach in Kipf *et al.* (2018). Two samples of predicted trajectories (scattered arrows) against the ground truth trajectories (grey dashed lines) for a 5 agent system performing chasing motions. Top row: traces of all 5 agents. Bottom: traces of only two agents for visibility. Only the starting T_{seg} states are provided to the model, and the prediction is done consecutively to the end. All graphs use data from test set.

Boids data. We can see in Fig. 3.9 (left) that the predictions now form envelopes according to the uncertainty at different time steps in the future. In general, the uncertainty appears to grow with larger prediction horizons. In the case of the red agent, the predictions slightly bifurcate around the obstacle. This clearly shows that the model is able to predict multiple potential futures of an agent (and the swarm) based on the inherent uncertainty of the task and environment. On the right, we see the probability distributions for discretized x- and y-coordinates of all agents. Again, some distributions are multimodal, which reinforces the insight that our predictions can produce multiple, diverse, and potentially conflicting future states.

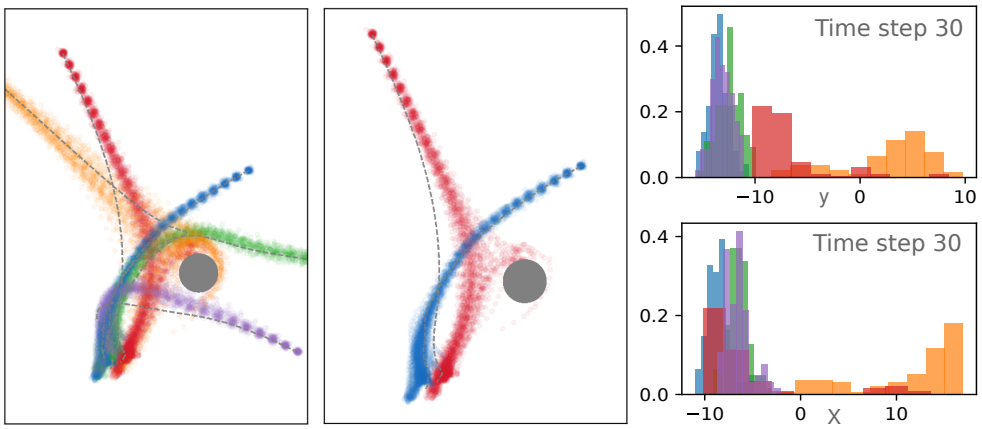

Figure 3.9: Predictions for the *nondeterministic* behavior of a swarm with 5 boids in the presence of uncertainty. Left graph: the stochastic output predictions of the SwarmNet$^+$ network for all agents. Middle: the predictions for only the blue and the red agent, for visual clarity. Right: The probability distributions over x and y coordinates of all agents at time step 30. We can see that the predictions for the red agent bifurcate to the left and right side of the obstacle.

3.3 Conclusion

In this part, we implemented SwarmNet, a GNN, that predicts and imitates the motion behaviors from observed swarm trajectory data. The network is combined with curriculum learning to achieve high accuracy prediction for arbitrary time steps. We demonstrated the network's ability to capture interaction dynamics in swarms through transfer learning. Lastly, we presented the model's built-in ability to simulate actuation uncertainty and its robustness to arbitrary changes in the environment and observational states.

Chapter 4

TOWARDS SCALABLE IMITATION LEARNING FOR MULTI-AGENT SYSTEMS WITH GRAPH NEURAL NETWORKS

In the previous chapter, the boid trajectories the SwarmNet is trained on contain only a fixed number of agents. Since the rules governing the behaviors of each agent is local and universal, independent of the swarm size, the multi-agent system must naturally scale with the number of agents in the system. Thus, whether the trained GNN based SwarmNet also scales is called into question.

We re-iterate the locality and universality characters of GNN's update rules Eq. (3.1)-(3.3): all 3 equations operate locally on the neighborhood of nodes. In other words, global attributes such as the number of nodes are not involved. So long as swarm systems share the same set of mechanisms, manifested by the same set of functions ϕ^e, $\psi^{\bar{e}}$ and ϕ^v in their underlying interaction graphs, the same GNN should be applicable to all the systems without the need for modification to the parameters. Although we argue later that Eq. (3.2) and (3.3) are not directly transferable, the statement still stands true that no change to the structure of the GNN is required.

Indeed, closer inspection of Eq. (3.2) raises suspicion that the number of edges connected to a node i may be implicitly correlated with the number of nodes in the graph. In reality, it is reasonable to expect agents to interact with more neighbors in a larger swarm. In spite that the swarms share the same set of functions, when transferring to a swarm of a different size, function $\psi^{\bar{e}}$ may be exposed to an unseen range simply due to more edges to be aggregated. Adding to the fact

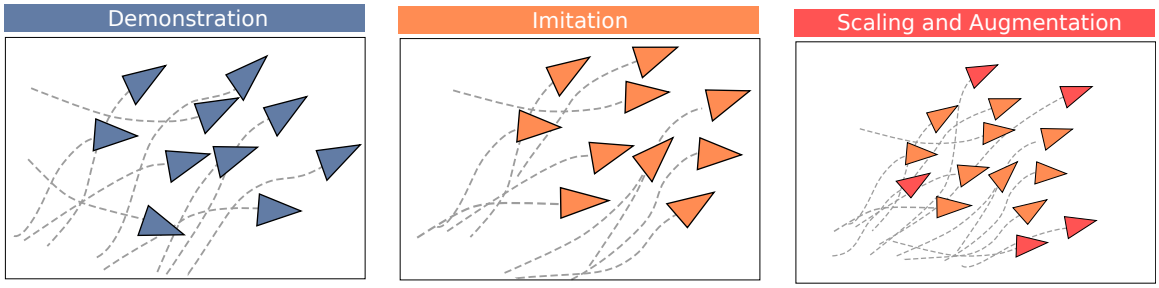

Figure 4.1: A depiction of the training and inference. For training the trajectories of a set of agents is recorded as demonstrations. After we train SwarmNet model, it can be used to create a copy of the swarm. In addition, it is possible to create larger and smaller swarms by changing the number of agents.

that MLPs are poor at extrapolation (Hettiarachchi *et al.*, 2005), function $\psi^{\bar{e}}$ may fail to respond correctly, leading to a drop in accuracy of the GNN.

4.1 Issue with Scalability

The scalability issue caused by poor extrapolation ability by MLP of Eq. (3.2) and Eq. (3.3) would become prominent in more realistic swarm models. In realistic swarms, not only does the size of agent neighborhood change with the size of the swarm, but the neighborhood may change dynamically as well. When long interaction range is assumed, agents are often effectively fully connected, extending the local neighborhood to the entire connectivity graph. The strong correlation between local connectivity and swarm size under this assumption makes trans-swarm application of a trained GNN almost impossible.

It is plausible that the simple summation in Eq. (3.1) is a bad choice. One obvious alternative appears to be the average reduction, which automatically eliminates the shift in range, since the aggregation is normalized by the number of agents. However, one may argue that particle simulation based models of swarms such as Boid (Reynolds, 1999) and Helbing (Helbing *et al.*, 2000) that achieve high level of authenticity are partly or strictly physics based and are built on superposition of accelerations caused by pair wise interactions, realized by simple summation. In other words, average reduction violates the underlying rules. A counter example would be the Vicsek model (Vicsek *et al.*, 1995), in which an agent's state is influenced by the average of its neighbors by design. A self-driving vehicle may prioritize its response to the most imminent danger, in which case a rank pooling aggregation is more desirable. In the scope of this paper, without deviating away from the proposed update rules of GNN, we propose a layer-wise tuning method realized by data padding as an attempt to reduce GNN's difficulty to scale.

An intuitive answer to treat the lack of responsiveness is to adjust the functions for edge aggregation and node update through transfer learning. To avoid catastrophic forgetting (Kirkpatrick *et al.*, 2017), however, the model has to constantly revisit previous data set. Although GNN in principle can freely adapt to different numbers of nodes, having a mixed dimension of input data poses problems to tensor based training frameworks. We note that a node with no interaction with any other node in a graph exerts no influence on or be affected by the rest of the graph, and in reverse, adding an isolated node to a graph does not change the dynamics of the members of the original graph. This immediately implies that the input data of a certain swarm can be padded

MODEL	LOSS					
	5 Boids	10 Boids	20 Boids	5 Helbings	10 Helbings	20 Helbings
Seq2Seq	-	1.45±0.14	-	-	1.50±0.13	-
Kipf's GNN	111±16	11.7±0.88	$\sim 10^4$	$\sim 10^4$	10.6±0.6	$\sim 10^6$
SwarmNet	0.93±0.09	0.41±0.14	1.69±0.15	0.24±0.02	0.05±0.03	0.32±0.04

Table 4.1: Model comparison for prediction error and scalability between Seq2Seq, Kipf's GNN and SwarmNet

with an arbitrary number "ghost" agents with 0 edges in the connectivity matrix. Through state update of the graph, the edge message between a "ghost" and any other node is never accounted for, so the GNN works equally well on the real agents. In this way, input data may have a uniform shape in the dimension of N when mixed with swarms of different sizes. We choose to initialize the states of "ghost" with zeros. Note that history states $\mathbf{S}_{(1:T_w,1:N,1:D)}$ in Algorithm 1 is padded to the maximum N in dataset with mixed N's. During transfer learning, w, θ_1, θ_2 are locked, and only θ_3 is to be updated in the last line.

4.2 Experiments

Before exploring the solutions to scalability, we first show that our model performs better than other models discussed in Chapter 3 in motion prediction on swarms of the size it is not trained on (Table 4.1). We test the accuracy of models' prediction on simulated Boid data and Helbing data. All three models are trained on both the Boid and the Helbing data, with 10 agents only for each model, but are tested on respective models of different sizes. Table 4.1 shows the MSE on test set for long term prediction upto 40 steps. The table also illustrate how non-GNN base model such as Seq2Seq (Sutskever *et al.*, 2014) is not able to be applied to swarms of different sizes. Kipf's GNN (Kipf *et al.*, 2018) being tasked with inferring the latent edge types, comes with the burden of a more complex structure, and is hard to train. SwarmNet, however, has the advantage of knowing the type labels of edges and a cleaner structure, can more easily capture the dynamics of the interactions, and produce accurate long term predictions. Not only that, it already shows better scalability on data of unseen sizes, possibly due to its slimmer structure. Samples of predicted long term trajectories are shown in the left pane of Fig. 4.2.

Secondly, following earlier discussion in Section 4.1 to prove the hypothesis that GNN naturally

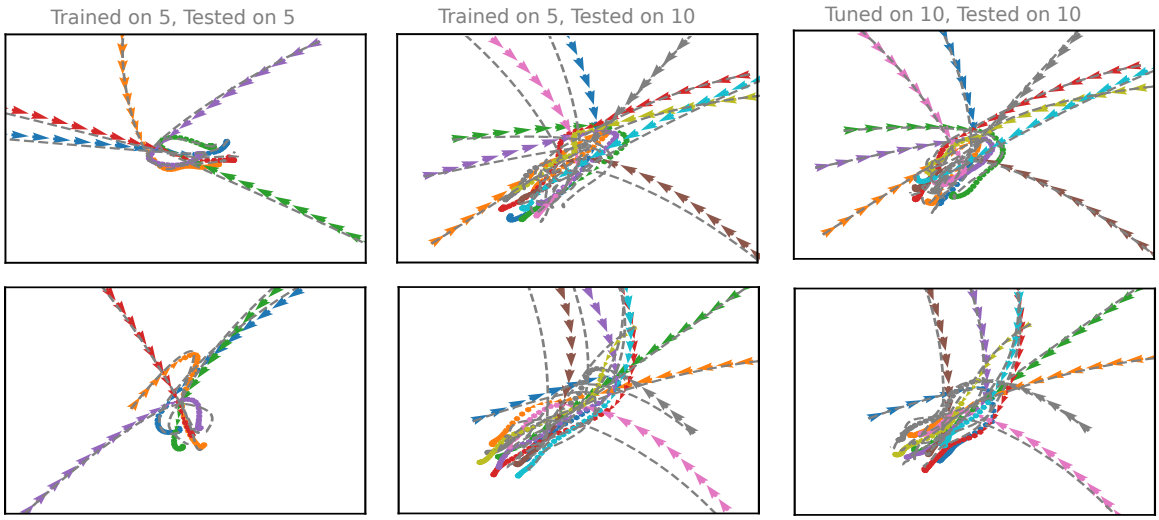

Figure 4.2: Predicted trajectories (colored arrows) overlaid on ground truth (gray dash lines). Only T_w initial states are supplied, and the prediction is done repeatedly upto 40 steps in future. *Left*: Trajectories of 5 boids by a network trained with data of 5 boids. *Middle*: Trajectories of 10 boids by the same network used in the left pane. *Right*: Trajectories of 10 boids by the network after tuning on padded data of 5 and 10 boids.

scales with N, we designed a simplistic swarm model called *chaser* to run experiments on as follows. A chaser is an agent that chases another particle with acceleration proportional to its displacement from its target. In a chaser swarm, each particle is only influenced by one other particle as its target, which means the interaction graph has only one edge attached towards each agent. The summation in Eq. (3.2) becomes trivial as it equals the single edge state associated with agent i, and function $\psi^{\bar{e}}$ can be absorbed into function ϕ^e when chained together. The second input to Eq. (3.3), which is the aggregated edge message $\bar{\mathbf{e}}$, directly inherits the single edge message. So, just like each chaser is oblivious to the rest of the group except for its target, the update rules are independent of N.

We trained SwarmNet on the simulated motion data from a chaser swarm of only a minimal number of agent, $N = 3$. Without any modification to the trained GNN, the test results on chaser swarms of various sizes $N = 3, 5, 10, 20, 100$ show perfect replication of the ground truth trajectory (Fig. 4.3), given only one window length of T_w initial steps as starting points. This simplistic case also seems to indicate that a learned function ϕ^e that governs the universal pair-wise interactions is transferable across swarms of the same type.

To investigate how scalability is challenged by the change in size of the neighborhoods through summation in Eq. (3.2), we modify the dynamic rules of the chaser model, such that each agent

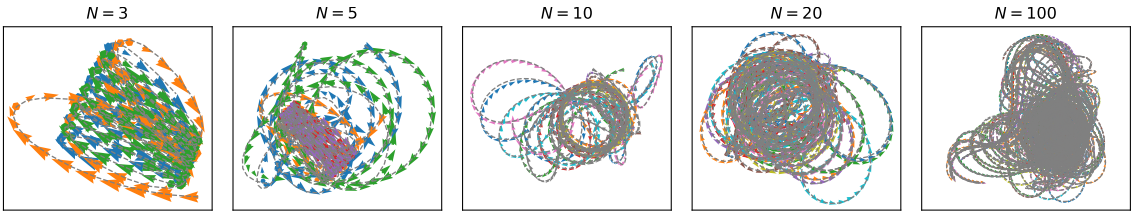

Figure 4.3: Predicted trajectories for chaser swarms of different sizes N. Colored arrows mark the predicted velocities and positions, against the grey dashed lines as the ground truth trajectories. Every agent in these systems chases only 1 other agent. Training was done only on systems with $N = 3$.

may now chase more than 1 target. We again train our model only on a chaser swarm with $N = 3$ and $M = 1$, M being the number of targets each chaser has. The trained model is tested on various combinations of N and M. It is immediate to notice the accuracy degradation in the first row of Table 4.2. To illustrate the shift in the input range to function $\psi^{\bar{e}}$ and subsequently function ϕ^v, we plot the distribution of the norm of output vectors of the functions' preceding components (Fig. 4.4). With the output of first function sharing similar distributions across systems, it is unsurprising that the ones with higher M's have the distribution of summation shifted to the right. Less overlapping is shared by the distributions when the distinction in M is larger, though same M still guarantees almost identical distribution. The strong resemblance by the distributions of function ϕ^e's output favors the presumption that the learned function ϕ^e in GNN from one swarm is readily transferable to other swarms with shared interaction rules.

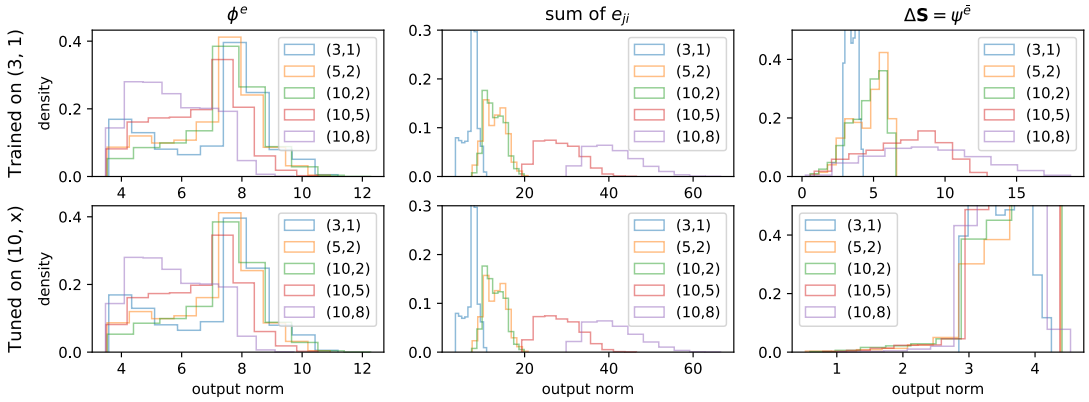

Figure 4.4: Distribution of output vector norms of critical components in SwarmNet, before and after tuning, on data sets with various (N, M). *Left*: Output of function ϕ^e. *Middle*: Output of edge message summation before entering function $\psi^{\bar{e}}$. *Right*: Final output of the network from function ϕ^v, as changes to agent states.

Next we show in Table 4.2, how tuning only the Eq. (3.2) and Eq. (3.3) with padded data extend the performance on swarms of larger neighborhoods. For the chaser model, after training only on

a dataset with 3 agent, each with 1 target ($N = 3, M = 1$), only the validation set with the same size of neighborhood sees the same level of performance (e.g. $N = 3, M = 1$ and $N = 5, M = 1$). We tune the model with Eq. (3.1) locked on a mixed set of chasers with $N = 10$, and M varying between 1 and 9. Note at this moment, padding is not necessary, because the nature of chaser model also allows a variety of neighborhood sizes without changing the swarm size. After tuning, performance on M less than 9 achieves similar levels, and is also greatly improved on larger swarms with larger neighborhoods. We see in Fig. 4.4 that tuning has corrected functions' response to unseen ranges caused by aggregation on higher number of edge messages. The apparent resemblance of the corrected distributions is due to the design of the chaser model, in which a particle chases the average position of all its targets, thus resulting in similar changes regardless of M. For other partly physics based models, the bias is not applicable, since having a higher number of interactions is often coupled with stronger changes. Fig. 4.5 shows the quality of the prediction even for larger unseen neighborhoods.

TRAIN	**VALIDATION SET**								
(N, M)	(3,1)	(5,1)	(5,2)	(10,2)	(10,5)	(10,8)	(20,9)	(20,12)	(20,17)
(3, 1)	0.051	0.036	17.02	14.27	104.93	275.3	350.7	701.4	1932
(10,x)	0.381	0.223	0.0939	0.0799	0.142	0.278	0.291	0.608	1.424

Table 4.2: Chaser evaluation before and after tuning

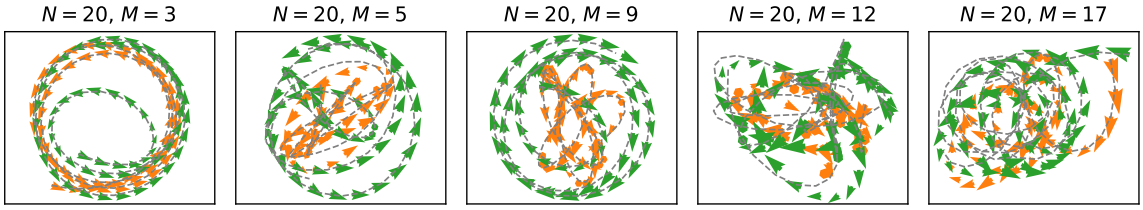

Figure 4.5: Predicted trajectories after tuning against the ground truth for chaser model with various swarm sizes N and neighborhood sizes M. Only the trajectories of 2 agents are shown for legibility.

The ground for tuning only Eq. (3.2) and Eq. (3.3) is based on the premises that training on one dataset, the network has already learned Eq. (3.1) as interaction rules universal to all sizes of the same swarms. To show the importance of a learned function ϕ^e and rule out the suspicion that the performance is solely determined by subsequent functions, we lock function ϕ^e right after random initialization and train the other functions from scratch, the training error is never able to

be brought down below 1 even for 1 step prediction, let alone long term prediction.

For Boid, we assume the agents are fully connected. The number of interaction edges, i.e. the size of neighborhood, is equivalent to the size of the swarms. We initially train the network from scratch on 5 boids and tune it with parameters of Eq. (3.1) locked on a mix of padded data of 5 and 10 boids (Table 4.3). Compared to networks that is trained only on 5 boids or 10 boids from scratch, this tuning and padding method preserves good performance on both data sizes and ones in between, while also avoiding catastrophic forgetting that appears in other transfer learning processes without mixed padded data. Trajectories in Fig. 4.2 show the improved performance from untuned network to a tuned one. As comparison, catastrophic forget occurs when revisiting of old data is not available. For instance in Table 4.3, the network that is trained first with 5 boids and tuned only with 10 boids, performance peak has quickly moved from 5 boids to 10 boids. After tuning on 20 boids, the peak moved to 20 boids as well.

Table 4.3: Comparison between training on fixed data and mixed data with tuning.

METHOD	**LOSS**				
	3	5	7	10	20
5	0.58±0.06	0.14±0.03	0.48±0.08	1.66±0.28	9.6±2.0
10	1.96±0.45	0.93±0.09	0.50±0.11	0.41±0.14	1.69±0.15
20	3.01±0.403	1.78±0.11	1.31±0.06	0.86±0.03	0.64±0.01
5→10	9.04±8.46	1.50±0.85	0.42±0.04	0.29±0.02	1.68±0.31
5→5-10	**0.69±0.13**	**0.13±0.02**	**0.33±0.08**	**0.28±0.02**	**1.74±0.27**
10→20	13.1±13.7	6.21±6.16	3.41±2.90	1.36±0.71	0.62±0.03

4.3 Conclusion

In this chapter, we discussed the challenge to SwarmNet and a class of GNNs' ability to scale. We showed that while the network is not hard constrained to application on systems of a fixed size, and that it indeed retains scaling ability to some extent compared to non-graph models. We discussed the availability and challenges in the scalability of GNN, and identified the root of the problem in the aggregation function. Finally, we proposed a method to improve it, by using layer-wise tuning and mixing of data enabled by padding.

Chapter 5

IMPROVING MULTI-AGENT BEHAVIORS WITH REINFORCEMENT LEARNING

In previous chapters, we demonstrated the SwarmNet's ability to faithfully replicate motion behaviors of a Boid system with moderate scalability. Although the trajectories are the results of the combined efforts on cohesion, collision avoidance, obstacle avoidance and goal seeking, there is no guarantee that the learned motion is collision free. Due to uncertainty both in the learning process and the execution of prediction in a realistic setting, predicted trajectories are not perfect. Since the error is measured purely by the relative distance from the prediction to the ground truth trajectories, the model makes no distinction between a deviation that cuts corners towards possible collision or one that offers more generous safety room. We aim to adjust the parameters in the way that imposes a greater punishment on the former which would reduce the chance of collision.

Reinforcement learning (RL) is an unsupervised machine learning paradigm built on the feedback reward and punishment. In contrast to the "imitating the demonstration from data" approach discussed in the previous chapter, no ground truth is provided as reference, and the model instead attempts to collect maximal reward based on the feedback.

Once our graph neural network has become accustomed to the underlying dynamics through learning from the demonstrations, we hope the extra punishment in the event of collision introduced by RL would help correct catastrophic errors. We plan to explore possible remedy in this area in the following sections.

5.1 Key Concepts and Mathematical Formulation in Reinforcement Learning

Reinforcement Learning is a learning scheme that attempts to maximize feedback received from environment by "trail and error". In the context of an agent exploring an environment through a known set of actions, the name "reinforcement" comes from the learning practice that the tendency of an action resulting in positive feedback is reinforced. The feedback often manifests as a *reward* value informed by the environment when responding to an action. Thus the effect of reinforcement learning is to maximize the expectation of agent's ability to receive reward.

Suppose S as the set of all possible states in the environment, and A is the set of all possible actions by the agent. In a Markovian decision process, at time step t, the agent chooses an action

according to its current state s_t. We call the function $\pi : S \times A \mapsto [0, 1] \subset \mathbb{R}$ *policy*, which defines the probability of taking action a_t at s_t by the agent as $\pi(a_t|s_t)$. Given the probability of state transition from s_t to s_{t+1} due to action a_t as $p(s_{t+1}|s_t, a_t)$, the probability for the agent to traverse through state sequence $(s_0, s_1, ..., s_T)$ by action sequence $(a_0, a_1, ..., a_{T-1})$ is

$$p(s_0, a_0, s_1, a_1, ..., a_{T-1}, s_T) = p(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{5.1}$$

where $p(s_0)$ is the probability the agent starts at state s_0.

The agent receives feedback reward r, from the environment by its choice of a_t at state s_t, $r : S \times A \mapsto \mathbb{R}$. Thus the goal is to optimize policy π for maximal expected reward.

$$\pi^* = \arg\max_{\pi} E_{\tau \sim p_\pi(\tau)} R(\tau) \tag{5.2}$$

where we have substituted state action sequence $(s_0, a_0, s_1, a_1, ...)$ with symbol τ, and $p_\pi(\tau)$ represents the probability under policy π. $R(\tau)$ is the total reward acquired through sequence τ, namely $R(\tau) = \sum_t r(s_t, a_t)$.

With policy being parameterized by θ, expected reward which is the objective of the optimization becomes a function of θ.

$$J(\theta) = E_{\tau \sim p_{\pi_\theta}(\tau)} R(\tau) \tag{5.3}$$

The task is then equivalent to solving for parameters θ that maximize $J(\theta)$. At extremum, θ obviously satisfies

$$\nabla J(\theta)\big|_{\theta^*} = 0 \tag{5.4}$$

However, solving for θ^* that nullifies the gradient is never practical. Nether is finding the $J(\theta)$ where the gradient is 0 the direct goal. Instead, algorithms that iteratively approach the maximal expected reward $J(\theta)$ are applied in the search for optimal parameters. For instance, *stochastic hill climbing*(Kimura *et al.*, 1995) perturbs the parameters randomly and selects the modification that locally increases the reward, in an analogy to "climb up" the hill of expected reward.

Still, the most popular methods guide policy parameter search with the gradient $\nabla J(\theta)$ in what is called *gradient ascent*. Similar to gradient descent in supervised learning learning, where the parameters move in the direction that reduces the prediction loss, gradient ascent in reinforcement learning increases the designated objective that is J. As such, the parameters θ are updated in the direction that the derivative ∇J is positive.

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{5.5}$$

where α is the learning rate which controls the rate of change.

On the other hand, optimization process in gradient based reinforcement learning differs with supervised learning in another critical aspect. Data samples in supervised learning is independent of the parameters being optimized, thus having a fixed optimization target. Since the expected reward return in Eq. (5.3) is determined by the state-action trajectory distribution $\tau \sim p_{\pi_\theta}(\tau)$, the data sampled under the new distribution change after parameters θ updates. The re-evaluation of optimization objective J is no longer a matter of substitution of the new parameters θ, but that the data used for evaluation must be re-sampled under the new parameters. This character of alternating between "policy iteration" and "value estimation" for policy optimization is summarized in Sutton and Barto (2018).

Intensive research has been conducted to address the high variance issue caused by the moving distribution. Examples include methods to stabilize value estimation (Schulman *et al.*, 2016), data sampling in parallel to smooth out randomness (Mnih *et al.*, 2016) and means to discourage drastic parameter shift (Schulman *et al.*, 2017) (Fujimoto *et al.*, 2018). Two of the recurring themes in reinforcement learning research is the trade-off of *on-policy* learning and *off-policy* learning, and the separation of policy searching and value estimation.

On-policy learning categorizes the class of reinforcement learning processes during which L is evaluated on the data sampled from the distribution under the current control policy $\pi(\theta)$. In contrast to re-evaluating J using entirely new data every time policy gets updated, off-policy breaks away from the strict "policy iteration" then "value estimation" paradigm a bit by allowing reuse of data samples of collected from past policies. The mismatch of trajectory distribution between outdated policy and current policy would introduce a bias to the value estimation for the current policy. The bias may be eliminated by factoring the old samples with *importance weight* (Wang *et al.*, 2016), which is the relative probability of old samples occurring under the current policy. Instead of discarding previously sampled trajectories immediately after policy update, the ability to reuse previously sampled trajectories effectively increases sample efficiency, as old data still offer relevant information on the advantage or disadvantage of past actions on certain states. Named *experiences*, the sampled trajectories are stored in memory and mixed with current trajectories to produce gradient for policy update. The term *experience replay* is coined to make the analogy to human learning from past experiences by replaying them in the conscious.

The separation of policy searching and value estimation further decouples the sensitive reliance on

unbiased and stable estimation of values from the ever changing policy during optimization. Instead of using the true values of rewards collected from execution of the current policy, the learning algorithm maintains a function V (or Q) that estimates the future expected reward from a given state s, $V(s)$ (or from a chosen action on the state $Q(s, a)$). The value function gets gradually updated according to newly acquired true rewards. This extra function is dubbed *critic*, in the sense that when substituting the true rewards of part of the trajectory τ in Eq. (5.3),

$$R(\tau) = \sum_{t=0}^{n-1} r(s_t, a_t) + Q(s_{t+n}, a_{t+n}) \tag{5.6}$$

or

$$R(\tau) = \sum_{t=0}^{n-1} r(s_t, a_t) + \sum_{s_{t+n} \in S} p(s_{t+n}|s_{t+n-1}, a_{t+n-1}) V(s_{t+n}) \tag{5.7}$$

it helps assess the merit of the policy as an authority without the necessity for the policy to be exposed to the roll-out of the episode. Here n is the number of time steps of true reward to be used for value assessment. Apart from enabling the learning algorithm to update the policy on the fly before seeing the end of τ, thus speeds up policy iteration, the fact that value functions are learnable estimate of the rewards anneals the inherent high variance in trajectory sampling due to policy change, stochasticity and exploration. In tandem with the name *critic*, the policy is called *actor*, and almost all modern reinforcement learning algorithms embodies the *actor-critic* paradigm.

In the following sections, we briefly introduce the background of policy gradient methods that are relevant to the learning of multi-agent behaviors laid out in the later chapter.

5.2 Policy Gradient Methods

As discussed earlier, Eq. (5.5) is an intuitive method for optimizing policy parameters θ to receive better reward. Explicitly writing the expectation as the sum over all possible trajectories weighted by probability and apply derivative with respect to θ,

$$\nabla_\theta J(\theta) = \sum_\tau R(\tau) \nabla_\theta p_{\pi_\theta}(\tau) \tag{5.8}$$

$$= \sum_\tau R(\tau) p_{\pi_\theta}(\tau) \frac{\nabla_\theta p_{\pi_\theta}(\tau)}{p_{\pi_\theta}(\tau)} \tag{5.9}$$

$$= \sum_\tau R(\tau) p_{\pi_\theta}(\tau) \nabla_\theta \log p_{\pi_\theta}(\tau) \tag{5.10}$$

Expanding derivative $\nabla_\theta \log p_{\pi_\theta}(\tau)$ in terms of the policy π_θ and a sequence of state action pairs according to Eq. (5.1), one gets

$$\nabla_\theta \log p_{\pi_\theta}(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{5.11}$$

since prior $p(s_0)$ and state transition probability $p(s_{t+1}|s_t, a_t)$ are independent of parameters θ. This leads us to the form

$$\nabla_\theta J(\theta) = E_{\tau \sim p_{\pi_\theta}(\tau)} \left[\left(\sum_{t=0}^{T-1} r(s_t, a_t) \right) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{5.12}$$

$$= E_{\tau \sim p_{\pi_\theta}(\tau)} \left[\sum_{t=0}^{T-1} \left(\sum_{t'=0}^{T-1} r(s'_t, a'_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{5.13}$$

Considering the fact that steps proceeding time step t bear no effect on the transition probability due to the Markovian presumption, it can be proved that the contribution from the reward up to time step t to the derivative $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is nullified, resulting in

$$\nabla_\theta J(\theta) = E_{\tau \sim p_{\pi_\theta}(\tau)} \left[\sum_{t=0}^{T-1} \left(\sum_{t'=t}^{T-1} r(s_{t'}, a_{t'}) \right) \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{5.14}$$

This is the formula for policy gradient which is the foundation for all modern policy gradient methods.

In practice, it is impossible to obtain directly the global distribution of τ, so Monte-Carlo sampling is applied. Specifically, the agent actively samples trajectories under its current policy as experiences and use the average of occurrence frequency to approximate the probability distribution:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T^i-1} R(\tau_t^i) \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \right] \tag{5.15}$$

where i represents the i-th sample and $R(\tau_t^i) = \sum_{t'=t}^{T^i-1} r(s_{t'}^i, a_{t'}^i)$ is called *reward-to-go*, which is the reward only since step t. Note that this is indeed the implementation of the classic REINFORCE Sutton *et al.* (2000) model. With the introduction of critic to replace part of reward-to-go in Eq. (5.15) according to Eq. (5.6) and Eq. (5.7), a series of modern reinforcement learning models have been developed as variants of the Actor-Critic framework. See DDPG (Lillicrap *et al.*, 2016), Actor-Critic Konda and Tsitsiklis (2000), A3C (Mnih *et al.*, 2016), TD3 Fujimoto *et al.* (2018), TRPO (Schulman *et al.*, 2015), PPO (Schulman *et al.*, 2017)), for more details of each model.

Since the critic value function must also be learned, it is expected to form a more accurate representation as the agent is exposed to the true reward collected for state-action pairs $r(s_t, a_t)$ during exploration. The value function is in turn updated to close the discrepancy between the

current estimated value and the collected reward-to-go. In the case of Q function,

$$\delta = R(\tau_t) - Q_\psi(s_t, a_t) \tag{5.16}$$

Then parameters ψ are updated by

$$\psi \leftarrow \psi + \delta \nabla_\psi Q_\psi(s_t, a_t) \tag{5.17}$$

Here δ is called the temporal-difference error, and as discussed earlier, the reward-to-go may be partly substituted by value estimation. For example, in n-step rollout, where expected reward beyond step $t+n$ is substituted by $Q(s_{t+n}, a_{t+n})$:

$$R(\tau_t) = \sum_{i=0}^{n-1} r(s_{t+i}, a_{t+i}) + Q(s_{t+n}, a_{t+n}) \tag{5.18}$$

Note that a convenient relation between V function and Q function exists by their definition, as

$$V(s_t) = \sum_{a_t \in \mathcal{A}} Q(s_t, a_t) \tag{5.19}$$

With the exception of REINFORCE, all other models employ the actor-critic framework. These classic implementations models of policy gradient method do not all follow the same choice of on-policy or off-policy training. Although some of the models strictly only support on-policy learning at the time of their inception, sample efficient off-policy extensions have been introduced later with few modifications.

5.3 Proximal Policy Optimization

One of the most popular policy gradient method today is the Proximal Policy Optimization (PPO) (Schulman *et al.*, 2017). It was proposed to rectify potentially drastic change in the parameter space in off-policy learning. Since offline learning recalls experiences from memory collected under the policy with older parameters θ', the gradients must be re-weighted by the relative ratio between the old policy $\pi_{\theta'}$ and current policy π_θ

$$\nabla_\theta J(\theta) = E_{\tau \sim p_{\pi_{\theta'}}(\tau)} \left[\left(\prod_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} \right) \sum_{t=0}^{T-1} R(\tau_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{5.20}$$

The policy probability $\pi_\theta(a_t|s_t)$ can undergo drastic changes within a short amount of training epochs for numerous reasons, for instance, that the policy π_θ is sensitive to parameter change, or that high variance in reward-to-go is experienced. Because of this, large ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}$ may be seen during training, and coupled with the product over all time steps, may result in a extremely

large gradient. Aside from possibly numerically unstable, the large gradient may cause the policy to land in an unrecoverable state. To mitigate the possible divergence, modifications to the vanilla policy gradient have been developed, with the most prominent ones being TRPO (Schulman *et al.*, 2015) and PPO (Schulman *et al.*, 2017). TRPO imposes that when learning on old trajectories, the parameter update be small enough so that the expected KL-divergence from old policy to updated policy is under certain threshold ϵ:

$$E_{\tau\sim p_{\pi_{\theta'}}(\tau)}\left[\sum_{t=0}^{T-1} D_{\mathrm{KL}}\left(\pi_\theta(a_t|s_t)\|\pi_{\theta'}(a_t|s_t)\right)\right] < \epsilon \tag{5.21}$$

PPO improves the overly expensive cost of solving gradients with KL-divergence as hard condition. Instead, it uses the ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)}$ as first order approximation of the KL-divergence and clip value the ratio between $(1-\epsilon, 1+\epsilon)$, effectively limiting oscillation by the ratio. It should be noted that for conciseness, other modifications to Eq. (5.15) inherited by PPO from previous work and other technical details are left out, for example, the use of *advantage function* instead of value functions (See Schulman *et al.* (2016)).

In this work, we use PPO as the choice of our RL optimization method. With a trained SwarmNet on Boid's behaviors for the multi-agent system, we show the desired individual low-level behaviors can be tuned in a controlled manner using RL.

5.4 Related Work

Reinforcement Learning has been widely used for many tasks where traditional control methods are not able to produce results which can adapt to uncertainty that is inherent in the environment. Classic works on multi-agent reinforcement learning such as MADDPG (Lowe *et al.*, 2017) and COMA (Foerster *et al.*, 2017) are based on policy gradient methods. Their implementations of the system states do not use the inherent graph structure of participating entities. Zambaldi *et al.* (2019) shows that accounting for relational information greatly improves performance of some tasks. Huttenrauch *et al.* (2017) follows an actor-critic approach to train cooperative agents for performing tasks such as target location and graph-based formation control. Zhou *et al.* (2019) shows the performance of GNN in multi-agent behavior cloning. Khan *et al.* (2020) use a Graph Convolutional Network (GCN) in their RL control policy which exploits the graph structure for the robots.

5.5 A Graph Neural Network for Imitation and Reinforcement Learning

Our approach is based on training a graph neural network to represent the control policy for a set of interacting agents. We assume centralized control of all agents via a single neural network. However, the approach can potentially also be applied in a decentralized setting by having each agent use a copy of the neural network.

The objective is to learn a neural network f which describes the swarm dynamics – the transition of the swarm from a given state to the state in the next time step. Here, the state of the entire system is made up of the state vector of every single agent (or particle). Accounting for homogeneity, the system is permutation invariant with respect to the agent states. In other words, the order by which each agent state is reported must not affect the representation of the system state. Traditionally, f is implemented as a fully-connected deep neural network, with agent states concatenated in an arbitrary order as input to f (e.g. (Lowe *et al.*, 2017)). This global computation not only limits f's ability to adapt to new systems, but also makes it impossible to scale to system with different number of participating agents. Graph Neural Networks (GNNs) on the other hand, eliminate these two problems by taking a local approach. Embedding the swarm system in a graph, each node represents an agent, with interaction between two agents represented by an edge. It proceeds to find the correspondence between states of consecutive time steps with the following method: (a) individual agents interact with neighboring agents independently; (b) all pair-wise interactions an agent is involved in are aggregated by superposition as the overall influence from its neighborhood; (c) agent update their state according to the neighborhood influence. Instead of attempting to directly discover the high-level mapping using one universal function approximator, GNNs recognize and dissect the dynamical components of the underlying physical models, and assign separate function approximators for the components.

For clarity, we reiterate the definition of the GNN update rules here, along with a readout function here. For a multi-agent system of N agents with state $s = \{\mathbf{x}_1^t, \mathbf{x}_2^t, ..., \mathbf{x}_N^t\}$ at a given time

step t,

$$\mathbf{e}_{ij}^t = \phi^e(\mathbf{x}_i^t, \mathbf{x}_j^t) \tag{5.22}$$

$$\bar{\mathbf{e}}_i^t = \sum_{j \in \mathcal{N}_i} \mathbf{e}_{ij}^t \tag{5.23}$$

$$\mathbf{h}_i^t = \phi^v(\mathbf{x}_i^t, \bar{\mathbf{e}}_i^t) \tag{5.24}$$

$$\mathbf{u}_i^t = \mu(\mathbf{h}_i^t) \tag{5.25}$$

where $\mathbf{e}_{ij}^t$ represents the interaction between node (agent) i and j. The summation in Eq. (5.23) is taken over the neighborhood of i, $\mathcal{N}_i$. ϕ^e and ϕ^v are called *edge functions* and *node functions*, respectively. *Read-out* function ψ maps the *node embedding* $\mathbf{h}_i^t$ to the desired output $\mathbf{u}_i^t$. Following conventions, we use a multi-level perceptron (MLP) to learn each function. We point out subtle differences between Eq. (3.1)-(3.3) and Eq. (5.22)-(5.24). Firstly, we simplify the aggregation function to a summation reduction, and neglect the function $\psi^{\bar{e}}$. Besides reducing the number of training parameters, the removal of $\psi^{\bar{e}}$ does not lose generality as the functionality may be absorbed into the downstream ϕ^v. Secondly, we intentionally use different notations for node state before update as $\mathbf{x}_t^i$, and the output $\mathbf{h}_t^i$ of Eq. (5.24), since the node encoding is not used as the updated node state $\mathbf{x}_i^{t+1}$. Thirdly, we explicitly add a readout function μ to further transform the result of graph convolution $\mathbf{h}_i^t$ to the control signal $\mathbf{u}_i^t$, which finally interfaces with the simulation environment to update to the next state $\mathbf{x}_i^{t+1}$.

In an actor-critic model, it is not uncommon to have the policy and the value function share the encoding of the system state. In our implementation both the policy and the value function take the multi-agent state, embed it in a graph, and produces action and value estimation for each agent. We let the policy and value function share the same encoding until after $\mathbf{h}_i^t$, with their own readout functions. Note that the readout function for value function produces a scalar, consistent with definition of value estimation, $V_i^t = \nu(\mathbf{h}_i^t)$.

Imitation Learning: With access to observed demonstrations of a multi-agent system, imitation learning enables us to clone the high-level behaviors by matching the system's input-output mapping with these observed demonstrations. By minimizing the mean-squared-error (MSE) between the policy outputs and demonstrated targets, the components in a GNN work in concert to form a suitable representation of the underlying interactions and update rules. In this setup, training can be performed using traditional supervised learning and gradient descent.

Reinforcement Learning: Although the trajectories are the results of the combined efforts

on cohesion, collision avoidance, obstacle avoidance and goal seeking, there is no guarantee that the learned motion is collision free. Due to uncertainty both in the learning process and the execution of prediction in a realistic setting, predicted trajectories are not perfect. Since the error is measured purely by the relative distance from the prediction to the ground truth trajectories, the model makes no distinction between a deviation that cuts corners towards possible collision or one that offers more generous safety room. To overcome this, we use Reinforcement learning (RL) to iteratively improve upon the networks weights extracted via imitation. One straight-forward approach is to apply a Policy Search method to all of the weights of the network. However, in the case of GNNs, structural information is available which allows us to focus the RL process on just a subset of the overall network weights.

Structured Learning and Compositionality: During RL, we take advantage of GNN's ability to directly learn the low-level interactions. The different types of interactions, namely from agents, obstacles and the goal to agents are assigned separate function approximators, $\phi^e_{o\to a}$, $\phi^e_{g\to a}$ and $\phi^e_{a\to a}$. Here $x \to a, x \in \{o, g, a\}$ denotes the directed edge from an obstacle (o), the goal (g) and another agent to the central agent of the neighborhood, respectively. During imitation learning from observed data, we expect GNN to learn a unique function for each type of interaction. Although these function approximators are randomly initialized in the same manner, when the model runs through data pieced from systems of various numbers of agents and obstacles, we argue that automatic separation of functions is encouraged by the variety in data, much like the separation of filter functionality in convolution neural networks (Zeiler and Fergus, 2014).

In light of this insight, the separation of functionality provides us an opportunity to adjust specific low-level behaviors through transfer learning by only tuning the corresponding functions while freezing others. For instance, since we are concerned with possible collision with other agents and obstacles, we choose to adjust functions $\phi^e_{o\to a}$ and $\phi^e_{g\to a}$ according to the feedback score from reinforcement learning. Further, it is also possible to freeze function ϕ^v in Eq. (5.24) if we are confident that it sufficiently captures the response of central node to the overall influence from the neighborhood. Doing so would greatly reduce the number of parameters which need to be trained.

5.6 Experiments and Results

5.6.1 Multi-Agent Environment

We consider a 2D environment consisting of N particle agents $\{i|1 \leq i \leq N\}$, a spherical obstacle o and a point goal g. The environment expects acceleration for every agent as action input and updates agents' position and velocity.

The task of the agents is to navigate towards the goal as efficiently as possible while maintaining tight formation and avoiding collisions amongst each other or with the obstacle. To reflex this purpose, we design the reward function for these three aspects of the task. Namely, r_a is a function of d_{ij} (distance between an agent i and another agent j), r_o is a function of d_{io} (distance between an agent i and the obstacle o), and r_g is a function of d_{ig} (distance between an agent i and the goal g).

$$r_a(d_{ij}) = \begin{cases} -c, & d_{ij} < 2a \\ -\rho d_{ij}, & d_{ij} \geq 2a \end{cases}, \; r_o(d_{io}) = \begin{cases} -c, & d_{io} < a + a_o \\ 0, & d_{io} \geq a + a_o \end{cases}, \; r_g(d_{ig}) = \frac{\gamma}{d_{ig} + \alpha} + \beta \tag{5.26}$$

where $c > 0$ is a large cost for collision, a and a_o are the radii of the agent and the obstacle. $\rho, \alpha, \beta, \gamma > 0$ are coefficients that shape the relative importance of the subtasks. Finally, the reward collected in the environment from all the three sources is fed to each of the agents.

$$r_i = r_o(d_{io}) + r_g(d_{ig}) + \sum_{j \in \mathcal{N}_i} r_a(d_{ij}) \tag{5.27}$$

We train a GNN policy that controls the group of agents in a centralized manner. Each of the elements including the obstacle and the goal is represented as a node in the graph, with the node state vector $\mathbf{x}_i = (p_{xi}, p_{yi}, v_{xi}, v_{yi})^T$ being the concatenated position and velocity of entity, where $i \in \{1, 2, ..., N, o, g\}$. We interpret the immediate output $\mathbf{y}_i^t$ in Eq. (5.24) as the acceleration action for the environment. Note that because the static obstacle and the goal are also treated as nodes, action output for these two nodes are spurious and dropped when interfacing with the environment.

5.6.2 Behavior Cloning from the Boid Model

We generate data demonstrating flocking behaviors with collision avoidance using a Boid model. The data consists of sequences of motion trajectories in terms of $\mathbf{x}_i^t$. Knowing the length of timestep

δt, acceleration output by Eq. (5.24) is transformed by

$$\hat{\mathbf{x}}_i^{t+1} = \begin{pmatrix} \mathbf{I} & \delta t\mathbf{I} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \mathbf{x}_i^t + \begin{pmatrix} \frac{\delta t^2}{2}\mathbf{I} \\ \delta t\mathbf{I} \end{pmatrix} \mathbf{u}_i^t \tag{5.28}$$

Parameters of the MLPs in Eq. (5.22) and Eq. (5.24) are optimized to reduce the mean-squared error $||\hat{\mathbf{x}}_i^{t+1} - \mathbf{x}_i^{t+1}||$. To further challenge the model's ability to accurately predict the actions, we enable multi-step prediction during training by feeding the predicted state $\hat{\mathbf{x}}_i^{t+1}$ back as input repeatedly. We employ curriculum learning by gradually increase the prediction horizon up to 10 steps. Fig. 5.1(a) shows the trajectories of 50 steps predicted by our GNN against the ground truth in the demonstrations given only the initial states.

We deliberately prepared our data to include a variety of compositions of the multi-agent system: (a) the number of agents N varies between 1 and 5, and (b) the obstacle is absent in a random sample of trajectories. We observe that the varied composition grants the GNN the ability to focus on different aspects of the dynamics, and edge functions $\phi^e_{o\to a}$, $\phi^e_{g\to a}$ and $\phi^e_{a\to a}$ have been tuned to represent the correct individual interactions. Fig. 5.1 shows with only a subset of edge functions selectively turned on or off, the GNN is successful at predicting some of the expected dynamics with *zero* change to the parameters.

Since we also have full control of the Boid model, we are able to selectively turn off the corresponding interactions between relevant entities to generate the ground truth (shown as gray dashed lines). Column (a) shows the prediction against the ground truth of the full dynamics. Column (b) and (c) show impressive reproduction of the converging motion caused by the goal seeking behavior alone or by both goal seeking and inter-agent interactions. In the two examples shown in Column (d), although there is one agent in each case that deviates from the ground truth, overall, the agent manages to demonstrate sensible obstacle avoidance while converging towards the goal; all other agents replicate the ground truth. It is seen that once the goal-agent edge is dropped, the agents controlled by the GNN no longer follow the ground truth, even though they perpetuate convergence in their own style. We speculate that because a good variation in the number of agents and that of the obstacles, the GNN has partly learned to assign the correct functionality to the edge functions $\phi^e_{a\to a}$ and $\phi^e_{o\to a}$ controlling the interactions, as long as the goal-agent edge $\phi^e_{g\to a}$ is present. But, because of the way the environment is designed, there is always a goal that guides the agents. Due to the constant presence of the goal-agent edge, functionality associated with $\phi^e_{a\to a}$ and $\phi^e_{o\to a}$ is never able to be fully separated from that of $\phi^e_{g\to a}$. In a controlled simulation, it is possible to generate

demonstrations with the goal occasionally turned off to force the $\phi^e_{g\to a}$ function to adapt. However, we argue that this approach would lose its generalization to a realistic imitation learning setup with only observation of systems not in our control. Extra measures for encouraging further separation of functionality is the interest of future study.

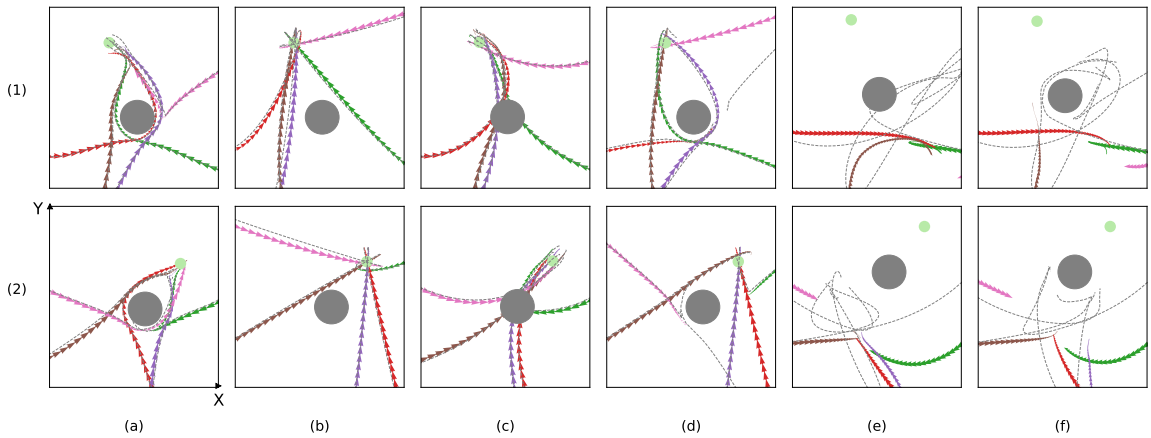

Figure 5.1: Separation of edge functions in GNN. Colored arrows show the trajectories of agents predicted by the GNN, moving around the grey circular obstacle towards the goal marked as a green dot. Dashed lines underneath show the expected trajectories by the Boid simulator. Row (1) and Row (2) provide 2 examples. Column (a) shows the full Boid model. Functions present in the rest of columns are as follows: (b) $\phi^e_{g\to a}$; (c) $\phi^e_{g\to a}$, $\phi^e_{a\to a}$; (d) $\phi^e_{g\to a}$, $\phi^e_{o\to a}$; (e) $\phi^e_{a\to a}$; (f) $\phi^e_{o\to a}$, $\phi^e_{a\to a}$. While isolated functions work well with $\phi^e_{g\to a}$ present, failure to fulfill expected behaviors shown in (e) and (f) suggests that part of functionalities are still mixed.

5.6.3 Improving the Behaviors via Reinforcement Learning

Once the GNN is trained on 10k trajectories, we transfer the GNN to RL as a control policy. Due to uncertainty and imperfect replication of the demonstrated behaviors, it is possible that agents encounter agent-agent collision or agent-obstacle collision. With the introduced negative reward for collision in Eq. (5.26), we expect RL to be able to repair the collision avoidance behaviors.

Despite the policy being deterministic during the IL stage, we reformulate the action output from IL as the mean of a Gaussian distribution with independent adjustable parameters σ_i to enable exploration. PPO (Schulman *et al.*, 2017) is our choice of RL algorithm. The value function of the actor-critic network shares the node embedding $\mathbf{h}^t_i$ with a separate read-out function ψ'. Before transitioning from IL to RL, the parameters of ψ' is tuned in the RL environment to keep it up-to-date with the IL trained policy.

Compared to RL from scratch (Fig. 5.2), the IL trained model clearly has a head-start. Not only is it able to reach a higher reward levels but is also optimized at a much faster rate. The necessary

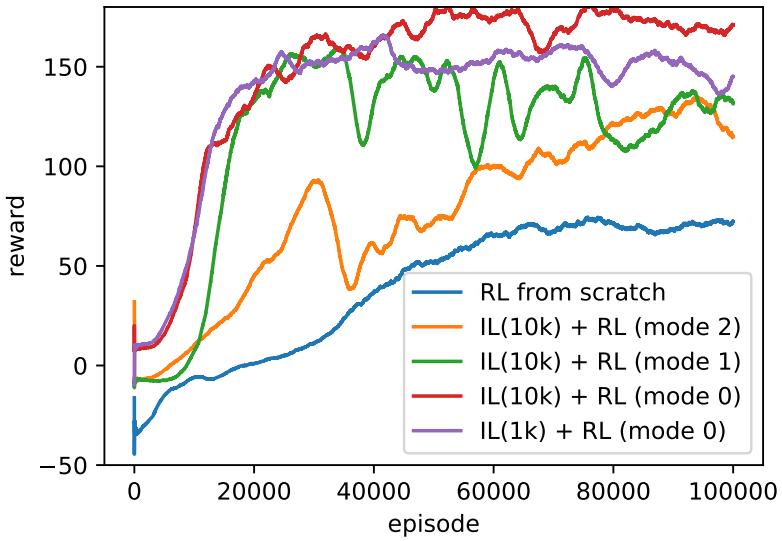

Figure 5.2: Smoothed reward history for training methods. The blue line shows RL from scratch. In Mode 0 RL, all parameters are free to be trained. In Mode 1, function $\phi^e_{g\to a}$ is frozen. In Mode 2, only $\phi^e_{o\to a}$ and $\phi^e_{a\to a}$ in the GNN policy are free.

number of episodes spent in the RL stage overwhelms the number of demonstrations used for IL. We are interested in how number of demonstrations for IL impact RL efficiency. The number of demonstrations in IL is limited to 1k. The reward history in the RL stage is presented in Fig. 5.2. It is clear that the RL efficiency does not drop even when given only 1k demonstrations in the IL stage. Fig. 5.3 demonstrates the improved policy zero-shot tested on a larger system with 10 agents.

Taking advantage of the separation of low-level functions discussed in Section 5.6.2, we explore how selectively turning functions off affects the RL stage. By doing so, we trust the inactive components have already obtained fair representations of the underlying mechanism. Specifically, to refine the collision free agent-agent and agent-obstacle interactions, one need only focus on tuning the parameters in functions $\phi^e_{o\to a}$ and $\phi^e_{g\to a}$. Depending on the relative magnitudes of the coefficients in the reward functions Eq. (5.26), RL may favor one type of behavior (e.g. more aggressive goal seeking when γ is higher), which means the bias caused by fixing components of our model may impair its potential to reach higher reward. We refer to training with all parameters free as *Mode 0* (see Fig. 5.2), training with $\phi^e_{g\to a}$ frozen as *Mode 1*, and tuning only $\phi^e_{o\to a}$ and $\phi^e_{a\to a}$ with all other functions locked as *Mode 2*.

However, we emphasize that the objective of the policy is *not* to solely maximize the expected reward, but to acquire the flocking behaviors described at the beginning of Section 5.6.1. Fixing the components has the benefit of retaining behaviors closer to those learned through IL. We show the

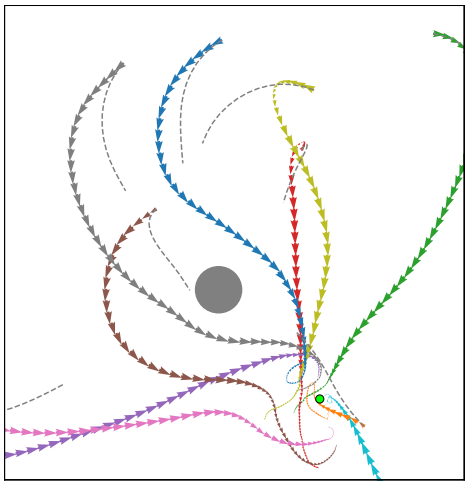

Figure 5.3: Zero-shot test on 10 agents. Grey dashed lines show the unimproved IL policy terminates early due to collision with the obstacle. Colored arrows show that the refined policy successfully avoids collision.

qualitative difference between trajectories under different modes of training and the policy obtained by IL in Fig. 5.4. Besides all RL modes manage to repair the obstacle collision in the IL policy, Mode 2 deviates from the IL policy the least.

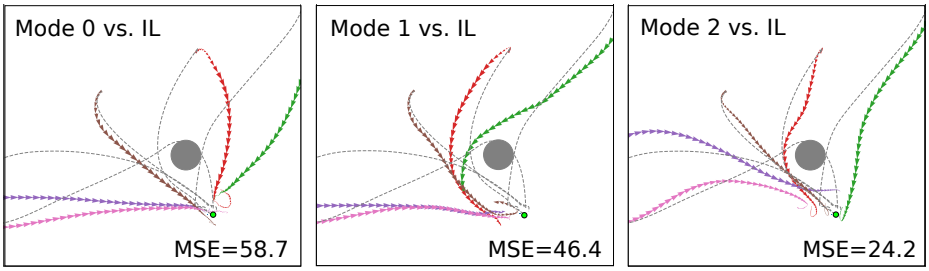

Figure 5.4: RL policies of different modes compared to the original IL trained policy. Grey dashed lines show trajectories of agents under the IL policy. Colored arrows mark the trajectories under RL tuned polices in the corresponding panes. The MSEs between trajectories qualitatively measure how far the RL policies have moved from the IL policy.

5.7 Conclusion

In this chapter, we discuss our results on learning and improving multi-agent swarm behavior and argue that a graph neural network approach is particularly helpful in uncovering group dynamics. We have shown that, after using imitation learning, it is technically possible to investigate the individual components from which the group behavior is composed. This ability could substantially help in uncovering building blocks for emergent behavior from data alone. We have also shown that

we can further improve upon a learned policy using reinforcement learning. In this case, too, we can exploit the structural information inherent to GNNs in order to freeze or enable specific parts of the group behavior. We have also shown early results which indicate that our methodology allows for the zero-shot transfer of learned policy to a larger swarm with more agents. For future work, we intend to expand on these results by performing a wider range of experiments using realistic robot swarm simulators as well as physical robots. Additionally, we want to further investigate the possibility of using GNN-specific reinforcement learning algorithms that aim at fast and robust improvement of swarm behavior.

Chapter 6

SAFETY MEASURE WITH CONSTRAINED OPTIMIZATION

In the previous chapter, we demonstrated the combination with reinforcement learning for controlled policy improvement on collision avoidance. With the negative feedback injected in case of collision, the learned policy under previous imitation would face punishment whenever causing collision due to uncertainty or error imperfect cloning of observed behaviors. Such feedback guides the optimization process to adjust parameters in the direction that reduces chance for the trajectories to enter the obstacle. Since RL is determined to be very computationally expensive (Fig. 5.2) in terms of iterative exploration through episodes, we are hereby interested in alternative solutions that enables us to "repair" the trained model with a more direct approach. We observe that as the negative feedback c received in Eq. (5.26) is increased to infinity, the collision branches of the reward functions $r_a(d_{ij})$ and $r_o(d_{io})$ approximate the hard constraints

$$d_{ij} \geq 2a, \quad d_{io} \geq a + a_o \tag{6.1}$$

Instead of using negative reward as an implicit and soft constraint during parameter optimization, we seek to implement hard constraints which require Eq. (6.1) to be always true. Although conventionally, additional constraints could be incorporated to the optimization objective (MSE loss in this case) via Lagrangian multipliers, such constraints would still function as soft regulators and not guaranteed to be satisfied during training. On the other hand, *constrained optimization* has seen recent success in verification of neural networks (e.g. Tjeng *et al.* (2019)), directly solving the neural networks subject to additional constraints has received limited attention (see e.g. Fischetti and Jo (2017)). In this chapter, we investigate the potential of mixed integer programming for repair the SwarmNet with Eq. 6.1 as hard constraints.

6.1 Mathematical Formulation of Neural Network as Mix Integer Program

We begin by introducing the mathematical formulation of a neural network layer as a mixed integer linear program. Of particular interest is a layer with ReLU activation.

$$\mathbf{h}^{l+1} = \text{ReLU}(\mathbf{W}^l \mathbf{h}^l + \mathbf{b}^l) \tag{6.2}$$

where $\mathbf{h}^l$ is the activation of layer l, $\mathbf{W}^l$ and $\mathbf{b}^l$ being the weights and bias of layer l. Considering that ReLU is a piece-wise linear function

$$\mathrm{ReLU}(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases} \tag{6.3}$$

the constraint by Eq. (6.2) can be formulated using a binary indicator $\theta = \{0, 1\}$

$$\mathbf{W}^l\mathbf{h}^l + \mathbf{b}^l = \mathbf{h}^{l+1} - \mathbf{h}_-^{l+1} \tag{6.4}$$

$$\mathbf{h}^{l+1} \geq 0 \tag{6.5}$$

$$\mathbf{h}_-^{l+1} \geq 0 \tag{6.6}$$

$$\begin{cases} \mathbf{h}^{l+1} \leq 0, \boldsymbol{\theta}^{l+1} = 0 \\ \mathbf{h}_-^{l+1} \leq 0, \boldsymbol{\theta}^{l+1} = 1 \end{cases} \tag{6.7}$$

Here the equality and inequality are all element-wise, that is, in Ineq. 6.5-6.6 all elements of $\mathbf{h}^{l+1}$ and $\mathbf{h}^l$ are non-negative. $\mathbf{h}^{l+1}$, $\mathbf{h}_-^{l+1}$ and θ^{l+1} are of the same dimensions, and branches in Ineq. 6.7 read that whether an element in the binary indicator θ^{l+1} is 0 or 1 determines whether the element at the same set of index in $\mathbf{h}^{l+1}$ or $\mathbf{h}_-^{l+1}$ is non-positive.

Conditions (6.4)-(6.7) establish the transformation from $\mathbf{h}^l$ to $\mathbf{h}^{l+1}$ the same way as the functional form Eq. (6.2). The $\boldsymbol{\theta}^{l+1} = 0$ branch represents the case when an element of the left-hand side of Eq. (6.4) is negative and the corresponding element of $\mathbf{h}^{l+1}$ is 0, while the positive case is covered by the $\boldsymbol{\theta}^{l+1} = 1$ branch in which the element in $\mathbf{h}_-^{l+1}$ is set to 0 and ignored. As suggested by the notation in Eq. (6.4), only $\mathbf{h}^{l+1}$ is passed as the input to the next layer transformation.

MLP layers are composed by transforming the layer activation and imposing constraints following (6.4)-(6.7). In turn, the entire SwarmNet can be translated into this mixed integer program. Similar to the training of SwarmNet with the conventional back-propagation method, an objective that measures the difference from the prediction to the ground truth is needed. Note the last layer of SwarmNet only contains linear transformation without non-linear activation. Since we prefer a linear program over a non-linear one, we intend to replace the mean-squared-error with a mean-absolute-error for its piece-wise linearity.

$$\min |\hat{\mathbf{y}} - \mathbf{y}'| \tag{6.8}$$

Minimizing absolute error between the read-out of SwarmNet $\hat{\mathbf{y}}$ and ground truth $\mathbf{y}'$ in the MIP

paradigm goes as follows:

$$\min \boldsymbol{\Delta}_+ + \boldsymbol{\Delta}_- \tag{6.9}$$

$$\text{s.t.} \boldsymbol{\Delta}_+ - \boldsymbol{\Delta}_- = \hat{\mathbf{y}} - \mathbf{y}' \tag{6.10}$$

$$\Delta_+ \geq 0 \tag{6.11}$$

$$\Delta_- \geq 0 \tag{6.12}$$

Similar to Eq. (6.4), the discrepancy $\hat{\mathbf{y}} - \mathbf{y}'$ needs both two terms to account for the value due to absolute value's piece-wise linearity with two branches. Binary indicator variable is not necessary as minimizing both branches Δ_+ and Δ_- ensures that one becomes zero.

6.2 Additional Constraint for Collision Avoidance

As discussed at the beginning of the chapter, avoidance collision is to be enforced by the additional hard constraint Eq. (6.1) during optimization. For simplicity, we neglect the source of collision from agents and only focus on the that from the spherical obstacle. Given the obstacle state $\mathbf{x}_o$ in 2D environment, the first two elements in the vector record the spatial coordinates of it. We use the notation $\mathbf{x}_{o,1:2}$ to represent the position of the obstacle and $\mathbf{x}_{i,1:2}$ for the position of agent i. The condition $d_{io} \geq a + a_o$ would mean

$$\|\mathbf{x}_{i,1:2} - \mathbf{x}_{o,1:2}\|_2 \geq a + a_o \tag{6.13}$$

where $\|\cdot\|_2$ is the L2 norm. However, because L2 norm breaks the piece-wise linearity of the program, a compromise has to be made to implement the constraint. We choose the L1 norm, acknowledging it defines a cube rather than a sphere. Naturally, an option for the size of the cube would be such that the sphere is inscribed in the cube. Specifically, we replace condition (6.13) with the following:

$$|x_{i,1} - x_{o,1}| + |x_{i,2} - x_{o,2}| \geq \sqrt{2}(a + a_o) \tag{6.14}$$

And in the MIP formulation, the condition on L1 distance is written as

$$\mathbf{x}_{i,1:2} - \mathbf{x}_{o,1:2} = \mathbf{d}_+ - \mathbf{d}_- \tag{6.15}$$

$$\mathbf{d}_+ \geq 0 \tag{6.16}$$

$$\mathbf{d}_- \geq 0 \tag{6.17}$$

$$\begin{cases} \mathbf{d}_+ \leq 0, \boldsymbol{\xi} = 0 \\ \mathbf{d}_- \leq 0, \boldsymbol{\xi} = 1 \end{cases} \tag{6.18}$$

$$d_{+,1} + d_{-,1} + d_{+,2} + d_{-,2} \geq \sqrt{2}(a + a_o) \tag{6.19}$$

where $\boldsymbol{\xi}$ is a 2D binary vector indicator and again the conditions are meant to be element-wise. $\mathbf{d}_+$ and $\mathbf{d}_-$ denote the absolute value of the displacement $\mathbf{x}_{i,1:2} - \mathbf{x}_{o,1:2}$ in its positive branch and negative branch, respectively.

6.3 Preliminary Results

We are able to construct the entire SwarmNet as an MIP. Ideally, we could directly solve for the optimal policy with safety constraints if the MIP is feasible, which is less likely when the network extends deeper. With this in mind, we propose a layer-wise tuning procedure. Each time, only the parameters of one layer is adjustable by the MIP, and the free layer is iteratively chosen among all layers until the objective converges.

In this section, we present our preliminary results of layer-wise tuning. The SwarmNet is first trained with IL on Boids data of 5 agents and 1 obstacle. For simplicity, we restrict the choice of layer to be only the last layer. Given that the number of samples M in training data also affects the number of branches to be explored by the solver, the time complexity scales proportional to 2^{MN}, where N is the number of adjustable neurons. We are able to solve the constrained optimization as MIP using Gurobi (Gurobi Optimization, 2020) with 1000 samples. We counted the number of time-steps as well as the number of trajectories that contain collision for both train and test sets. Although we used L1 distance in the safety constraint, collision under the normal criteria measured by L2 distance consistently dropped for both the training data and test data, shown in Table 6.1. It is expected that the predictive accuracy drops by enforcing the L1 safety measure, as seen by the increase in MSE.

	Train set			Test set		
	No. steps	No. trajs	MSE	No. steps	No. trajs	MSE
IL	25/1000	5/100	4.62	51/2500	13/250	4.95
Repair	6/1000	2/100	20.09	25/2500	7/250	19.26

Table 6.1: Collision frequency before and after repair. The last layer is tuned with 1000 steps as the train set, and tested on 2500 steps. Collision in both the number of steps and number of trajectories is counted. MSE between original IL policy prediction and repaired policy prediction is measured.

6.4 Discussion

Our preliminary results show promise of incorporating the explicit constraint in an MIP. We recognize the challenges that remain to be tackled. These challenges include the fact that the scaling of time complexity prevents us from having a larger training set, limiting the generalization ability of the repaired network. Although the last layer the the most sensitive to the constraint on the output, it is the respective edge function that needs the repair work. At the current stage, the edge function being farther away from the output has trouble responding to the repair constraints due to few samples with collision in the training set. Lastly, we also need solid investigation in the convergence of layer-wise tuning. Thorough study of these challenges is left to future work.

Chapter 7

SUMMARY

The motivation of the thesis is to propose a deep learning approach to learn multi-agent swarming behaviors with a graph based neural network we call SwarmNet. Contrary to the manual design approach, the graph network deep learning approach is able to form sensible understanding of low-level behaviors as well as replicating the top level behaviors observed, without human intervention.

SwarmNet, built upon previous development on GNNs, treats agents as nodes of the graph, and describes interactions of involved entities with edge functions. Under this formulation, evolution of the system states is interpreted as information exchange from one node to the other through the edges. The edge functions dynamically computes representations of the interactions, before they are superposed to form the overall influence to each agent from their respective neighborhood. The decision making or state update process is then approximated by another component of GNN which is a node function.

Structurally, locality and universality presides SwarmNet and GNNs, Each node is only directly affected by the immediate neighborhood, and nodes of the same type follow the same set of rules within their respective neighborhood across the graph. These principles allude to the fact that functionalities of GNNs are independent of the graph size, and in turn the size of the swarm in application. However, we show that the inevitable coupling of the output range of the summation reduction in edge aggregation with the size of the neighborhood prevents accurate generalization to systems with different size of the neighborhood. This drawback is particularly pronounced when the embedded graph is fully connected. We stress the importance of variety in the size of neighborhood during training for better generalization, and alleviate the overfit for fully connected graphs via padding in batch training.

Additionally, learning of multi-agent behaviors is not limited to imitation learning. Reinforcement learning is introduced to further refine the swarming behaviors with a stronger push for safety. We demonstrate a training scheme that combines imitation learning and reinforcement learning produces significantly better control policy from reinforcement learning from scratch. The potential to uncover and separate low-level behaviors is another advantage of graph structured neural network. When the training data contain systems of various compositions, the GNNs respond to the variation

in the composition by adapting the edge functions accordingly. As a result, functions are aligned with the dynamical systems underlying interaction mechanisms. This can be utilised to create fine controlled custom policy.

In future work, we plan on testing our methodology with real drone experiments. Several open-source quadrotor drone simulators (Shah *et al.*, 2017)(Song *et al.*, 2020) are available for training and simulation purposes. It is possible to program heuristic interaction functions like those in the Boid model to create demonstrations of a weak expert for imitation learning. The simulators then support reinforcement learning so that the SwarmNet policy that has cloned the heuristic behaviors is able to improve safe swarming navigation using the feedback collected from physics based simulations. At this step, the centralized policy should be converted to copies of decentralized policies for each agent, and further model tuning may be required. And finally, we want to install the SwarmNet to physical drones to test the viability in real application.

We note that techniques demonstrated in the SwarmNet is not limited to motion prediction and planning of multi-agent systems. It in principle applies to dynamics of any system that may be represented as a graph. We are interested in exploring possible applications on other systems.

Balch, T. and M. Hybinette, "Social potentials for scalable multi-robot formations", Proceedings-IEEE International Conference on Robotics and Automation **1**, February 2000, 73–80 (2000).

Battaglia, P. W., J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li and R. Pascanu, "Relational inductive biases, deep learning, and graph networks", pp. 1–40, URL `http://arxiv.org/abs/1806.01261` (2018).

Bengio, Y., J. Louradour, R. Collobert and J. Weston, "Curriculum learning", Proceedings of the 26th annual international conference on machine learning pp. 41–48 (2009).

Bruna, J., W. Zaremba, A. Szlam and Y. LeCun, "Spectral Networks and Locally Connected Networks on Graphs", pp. 1–14, URL `http://arxiv.org/abs/1312.6203` (2013).

Defferrard, M., X. Bresson and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering", Advances in Neural Information Processing Systems pp. 3844–3852 (2016).

Fischetti, M. and J. Jo, "Deep Neural Networks as 0-1 Mixed Integer Linear Programs: A Feasibility Study", URL `http://arxiv.org/abs/1712.06174` (2017).

Foerster, J., G. Farquhar, T. Afouras, N. Nardelli and S. Whiteson, "Counterfactual Multi-Agent Policy Gradients", URL `http://arxiv.org/abs/1705.08926` (2017).

Fujimoto, S., H. Van Hoof and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods", 35th International Conference on Machine Learning, ICML 2018 **4**, 2587–2601 (2018).

Gal, Y., *Uncertainty in Deep Learning*, Ph.D. thesis, University of Cambridge (2016).

Gilmer, J., S. S. Schoenholz, P. F. Riley, O. Vinyals and G. E. Dahl, "Neural message passing for quantum chemistry", 34th International Conference on Machine Learning, ICML 2017 **3**, 2053–2070 (2017).

Gurobi Optimization, L., "Gurobi optimizer reference manual", URL `http://www.gurobi.com` (2020).

Helbing, D., I. Farkas and T. Vicsek, "Simulating dynamical features of escape panic", Nature **407**, 6803, 487–490 (2000).

Hettiarachchi, P., M. J. Hall and A. W. Minns, "The extrapolation of artificial neural networks for the modelling of rainfall–runoff relationships", Journal of Hydroinformatics **7**, 4, 291–296 (2005).

Hochreiter, S. and J. Schmidhuber, "Long short-term memory", Neural computation **9**, 8, 1735–1780 (1997).

Huttenrauch, M., A. Sosic and G. Neumann, "Guided deep reinforcement learning for swarm systems. arxiv preprint (2017)", arXiv preprint arXiv:1709.06011 (2017).

Khan, A., E. Tolstaya, A. Ribeiro and V. Kumar, "Graph policy gradients for large scale robot control", in "Conference on Robot Learning", pp. 823–834 (PMLR, 2020).

Kimura, H., M. Yamamura and S. Kobayashi, "Reinforcement Learning by Stochastic Hill Climbing on Discounted Reward", pp. 295–303 (Morgan Kaufmann, 1995), URL `http://www.sciencedirect.com/science/article/pii/B978155860377650044X`.

Kipf, T., E. Fetaya, K.-c. W. Max and W. Richard, "Neural Relational Inference for Interacting Systems", (2018).

Kipf, T. N. and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks", pp. 1–14, URL http://arxiv.org/abs/1609.02907 (2016).

Kirkpatrick, J., R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran and R. Hadsell, "Overcoming catastrophic forgetting in neural networks", Proceedings of the National Academy of Sciences **114**, 13, 3521–3526 (2017).

Konda, V. R. and J. N. Tsitsiklis, "Actor-critic algorithms", Advances in Neural Information Processing Systems pp. 1008–1014 (2000).

Lagoudakis, M. G., M. Berhault, S. Koenig, P. Keskinocak and A. J. Kleywegt, "Simple auctions with performance guarantees for multi-robot task allocation", in "2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)", vol. 1, pp. 698–705 (IEEE, 2004).

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous Control with Deep Reinforcement Learning", (2016).

Lowe, R., Y. Wu, A. Tamar, J. Harb, P. Abbeel and I. Mordatch, "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments", Nips , Nips, 6379–6390, URL http://arxiv.org/abs/1706.02275 (2017).

Mesbahi, M. and M. Egerstedt, *Graph theoretic methods in multiagent networks*, vol. 33 (Princeton University Press, 2010).

Micheli, A., "Neural network for graphs: A contextual constructive approach", IEEE Transactions on Neural Networks **20**, 3, 498–511 (2009).

Mnih, V., A. P. Badia, L. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", 33rd International Conference on Machine Learning, ICML 2016 **4**, 2850–2869 (2016).

Reynolds, C. W., "Flocks, herds and schools: A distributed behavioral model", ACM SIGGRAPH Computer Graphics **21**, 4, 25–34, URL http://portal.acm.org/citation.cfm?doid=37402.37406 (1987).

Reynolds, C. W., "Steering behaviors for autonomous characters", Game Developers Conference pp. 763–782 (1999).

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini, "The graph neural network model.", IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council **20**, 1, 61–80 (2009).

Schulman, J., S. Levine, P. Moritz, M. Jordan and P. Abbeel, "Trust Region Policy Optimization", (2015).

Schulman, J., P. Moritz, S. Levine, M. I. Jordan and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation", 4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings pp. 1–14 (2016).

Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms", Current Opinion in Neurobiology **17**, 4, 456–464, URL http://arxiv.org/abs/1707.06347 (2017).

Shah, S., D. Dey, C. Lovett and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles", in "Field and Service Robotics", (2017), URL https://arxiv.org/abs/1705.05065.

Song, Y., S. Naji, E. Kaufmann, A. Loquercio and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator", arXiv preprint arXiv:2009.00563 (2020).

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", J. Mach. Learn. Res. **15**, 1, 1929–1958 (2014).

Sutskever, I., O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks", Nips p. 9 (2014).

Sutton, R. S. and A. G. Barto, *Reinforcement Learning: An Introduction* (A Bradford Book, Cambridge, MA, USA, 2018).

Sutton, R. S., D. McAllester, S. Singh and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation", Advances in Neural Information Processing Systems pp. 1057–1063 (2000).

Tianping Chen and Hong Chen, "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems", IEEE Transactions on Neural Networks **6**, 4, 911–917 (1995).

Tjeng, V., K. Xiao and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming", pp. 1–21 (2019).

Tran, D. V., N. Navarin and A. Sperduti, "On filter size in graph convolutional networks", in "2018 IEEE Symposium Series on Computational Intelligence (SSCI)", pp. 1534–1541 (2018).

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need", URL `http://arxiv.org/abs/1706.03762` (2017).

Veličković, P., A. Casanova, P. Liò, G. Cucurull, A. Romero and Y. Bengio, "Graph attention networks", 6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings pp. 1–12 (2018).

Vicsek, T., A. Czirók, E. Ben-Jacob, I. Cohen and O. Shochet, "Novel Type of Phase Transition in a System of Self-Driven Particles", Physical Review Letters **75**, 6, 1226–1229, URL `https://link.aps.org/doi/10.1103/PhysRevLett.75.1226` (1995).

Wang, Z., V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu and N. de Freitas, "Sample Efficient Actor-Critic with Experience Replay", , 2016, URL `http://arxiv.org/abs/1611.01224` (2016).

Xu, K., S. Jegelka, W. Hu and J. Leskovec, "How powerful are graph neural networks?", 7th International Conference on Learning Representations, ICLR 2019 pp. 1–17 (2019).

Zambaldi, V., D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals and P. Battaglia, "Deep Reinforcement Learning With Relational Inductive Biases", International Conference on Learning Representations p. 2019 (2019).

Zeiler, M. D. and R. Fergus, "Visualizing and understanding convolutional networks", Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **8689 LNCS**, PART 1, 818–833 (2014).

Zhou, S., M. J. Phielipp, J. A. Sefair, S. I. Walker and H. B. Amor, "Clone Swarms: Learning to Predict and Control Multi-Robot Systems by Imitation", in "2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)", pp. 4092–4099 (IEEE, 2019), URL `https://ieeexplore.ieee.org/document/8967824/`.

APPENDIX A

CODE REPOSITORIES

Simulation of Boid, Helbing and Chaser models: `https://github.com/siyuzhou/swarms`
SwarmNet: `https://github.com/siyuzhou/swarmnet`
Reinforcement Learning with SwarmNet and Boids: `https://github.com/siyuzhou/swarmrl`