Graph Diffusion for Imitation Learning in Robotics

Exploring Imitation Learning as a Graph Generation Problem using Diffusion-based Methods
Master thesis by Caio Victor Gouveia Freitas (Student ID: 2328654)
Date of submission: June 5, 2024

1. Review: Prof. Dr. Georgia Chalvatzaki
2. Review: Prof. Dr.-Ing. Rolf Findeisen
3. Review: M.Sc. Ali Younes, M.Sc. An Thai Le, M.Sc. Maik Pfefferkorn
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Electrical Engineering and Information Technology Department

Institut für Automatisierungstechnik und Mechatronik

IAS and PEARL Lab

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Caio Victor Gouveia Freitas, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 5. Juni 2024

C. V. G. Freitas

Abstract

In the field of robotics, imitation learning (IL) has emerged as a powerful paradigm, enabling machines to learn increasingly complex behaviors from human demonstrations. While conventional IL methods require careful engineering of observation and action spaces, recent advancements in diffusion models and graph neural networks (GNNs) offer promising methods for more flexible and general learning approaches. This thesis explores the fusion of these advancements by proposing a novel framework that leverages diffusion-based graph generation for IL in robotics. We introduce two graph diffusion policies for IL: GraphDDPM, based on denoising diffusion probabilistic models, and ARGG, which explores the autoregressive graph generation framework. These aim to expand the applications of such models to robotic action generation tasks. Our approaches differ from existing methods by incorporating joint-level proprioception and action representation, treating the environment and manipulator as graphs, and operating directly in the graph domain to utilize structural information. We conduct comprehensive experiments to validate our approaches and their qualities, comparing its performance against traditional IL methods and contemporary state-of-the-art techniques. Additionally, we discuss limitations and future improvements to the developed framework, aiming to incentivize future research applying such models in robotics. Through this research, we aim to bridge the gap between diffusion-based graph generative models and robotics, offering insights into the potential of this approach for future advancements in robotic learning and autonomy.

Zusammenfassung

In der Robotik hat sich *Imitation Learning* (IL) als einflussreiches Paradigma etabliert, das es Maschinen ermöglicht, zunehmend komplexe Verhaltensweisen aus menschlichen Demonstrationen zu erlernen. Während konventionelle IL-Methoden eine sorgfältige Gestaltung der Beobachtungs- und Aktionsräume erfordern, bieten jüngere Fortschritte in Diffusionsmodellen (*Diffusion Models*) und Graph-Neuronalen Netzen (GNNs) vielversprechende Methoden für flexiblere und allgemeinere Lernansätze. Die vorliegende Arbeit untersucht die Fusion dieser Methoden durch einen neuartigen Ansatz, der diffusionsbasierte graphengenerative Modelle für IL in der Robotik nutzt. Wir stellen zwei Ansätze zur Graph-Diffusion für IL vor: GraphDDPM, basierend auf den *denoising diffusion probabilistic models*, und ARGG, das den autoregressiven Graphengenerierungansatz erforscht. Diese Richtlichtlinien zielen darauf ab, die Anwendungen von diffusionsbasierten Modellen auf die Aufgaben der robotischen Aktionsgenerierung auszudehnen. Unsere Ansätze unterscheiden sich von bestehenden Methoden durch die Integration von Propriozeptions- und Aktionsrepräsentation auf Gelenkebene, indem sie die Umgebung und den Manipulator als Graphen behandeln und direkt im Graphenbereich arbeiten, um strukturelle Informationen zu nutzen. Wir führen umfassende Experimente durch, um unsere Ansätze und deren Qualitäten zu validieren, vergleichen ihre Leistung mit traditionellen IL-Methoden und Techniken, die den Stand der Entwicklung repräsentieren. Darüber hinaus diskutieren wir Einschränkungen und Verbesserungen des entwickelten Ansatzes, um künftige Forschende zu motivieren, solche Modelle in der Robotik anzuwenden. Somit zielt die Forschungsarbeit darauf ab, die Kluft zwischen diffusionsbasierten graphengenerativen Modellen und der Robotik zu überbrücken und Einblicke in das Potenzial des Ansatzes für robotergestütztes Lernen und Autonomie zu gewähren.

Contents

1. Introduction

In recent years, imitation learning has emerged as a relevant paradigm in robotics, offering means for machines to learn from human demonstrations and replicate complex behaviors without explicitly instructing actions to the robotic platform [60]. This approach not only accelerates the learning process but also facilitates the transfer of skills from the demonstrator to the robot, allowing it to reproduce the behavior from the recorded data [37].

In most work, the approach is to carefully engineer the observation and action space to have it reduced to a minimal efficient set to facilitate learning [12], affecting the robot proprioception, perception of the objects, as well as the choice of robot actions.

Meanwhile, the landscape of machine learning has witnessed remarkable advancements, particularly in the realm of diffusion models. These models, inspired by processes such as random walks and heat diffusion, have demonstrated exceptional capabilities in capturing complex data distributions and generating realistic samples [16, 8, 46]. While primarily applied in domains like image generation [16], their potential has been recently explored in other domains such as human motion synthesis [11] and robotics [7, 45, 25, 39]. Recently, Diffusion Policy from [10] has significantly improved state-of-the-art performance across several robotics tasks using this framework, being able to precisely imitate and multi-modal actions.

Similarly, graph neural networks (GNNs) have also recently emerged as a powerful tool for learning on complex, non-euclidean data [41]. Given the variety of application domains in which graph representation is adequate, GNNs were a response from the deep learning research community to operate natively in this domain. GNNs have achieved great performance in several domains, such as Bioinformatics [19] and natural language processing [55], and also have shown up in several works in robotics that benefit from it [41].

These graph models have also been paired with diffusion-based techniques, aiming to leverage the benefits of diffusion processes on graphs models, and have been applied in various graph generation tasks [9], especially in the field of molecule generation [18, 23, 17] and traffic forecasting [26].

However, to the best of our knowledge, diffusion-based graph generative models have not yet been applied to robotics. This thesis aims to fill this gap by investigating approaches that leverage such models for learning and generating robot actions as graphs. In doing so, we aim to expand the applications of diffusion-base graph generative models by implementing, testing, and validating a graph diffusion policy for IL and benchmarking the performance of our proposed methods against traditional IL approaches [33] and contemporary state-of-the-art techniques, such as the one from [10].

Our approach differs from the one in [10] in the sense that it 1) uses joint-level proprioception and actions, aiming for embodiment-awareness 2) represents the environment and the manipulator as a graph 3) is defined by a model that operates on the graph domain, aiming to leverage the robot's structural information

This thesis is structured as follows. In Chapter 2, we provide an in-depth introduction to the backgrounds of imitation learning, graph neural networks, and diffusion models. We review related work in each of these areas, focusing particularly on their applications in the field of robotics. Chapter 3 formally introduces the problem under investigation and outlines the explored approaches to graph generation within this context. In Chapter 4, we detail the experimental setup, present our experiments, and discuss the findings. Finally, Chapter 5 summarizes the results of our research, offering insights into its limitations and strengths, and provides a perspective on potential future work.

2. Background and Related Work

This chapter introduces the theoretical foundations and related work relevant to imitation learning, graph neural networks, and diffusion models summarizing the core ideas of each of the frameworks that are explored in this thesis. First, we introduce the IL framework, its broad categories, some of its challenges, common practices, and applications in the field of robotics. After that, the most important properties of GNNs are explored, such as information propagation and pooling, followed by an outline of the related work applying GNNs to robotics, that motivates this work.

Finally, diffusion models are presented along with their most popular approaches, as well as some applications in the field of robotics, also motivating the current work. We finish this chapter by introducing diffusion-based graph generative models (the combination of diffusion models and GNNs) and some of the literature and applications.

2.1. Imitation Learning

In the IL framework, the agent aims to learn to perform a specific task by imitating an expert's demonstrations. That is accomplished by training the agent (model), which can be interpreted as learning the mapping between observations and corresponding actions to accomplish tasks [60].

In the literature, two broad categories of approach can be found: behavioral cloning and inverse reinforcement learning [60, 51]. Furthermore, a more recent development known as imitation from observation (IfO) has emerged as a more natural way to consider learning from an expert, which aims to gather insights solely from observational data without explicit low-level actions [51].

Behavioral Cloning (BC) faces the problem of learning behaviors as a supervised learning task, where demonstrations by an expert are recorded and mimicked by learning a state-action map. This straightforward and data-centered method operates independently of any task or environmental specifics. It leverages the extensive body of research within the domain of machine learning on supervised learning methodologies for its techniques [60].

However, it is known to suffer from the covariate shift problem that arises when the state distribution observed during testing differs from the ones on the data set, leading to weak generalization to out-of-distribution states [43]. Besides that, the ”copycat problem” arises when the agent learns to cheat by replicating the expert’s previous actions leading to highly correlated actions over time. Behavioral cloning methods can be explicit, trying to map actions directly to observations, or implicit, treating BC as an energy-based modeling problem, and then performing optimization based on the energy model.

Inverse Reinforcement Learning (IRL) involves inferring the reward function underlying observed demonstrations, followed by policy optimization through reinforcement learning (RL).

For all of these, the data collection must be performed in the same domain as the robotic platform, which is what Imitation from Observation (IfO) tries to avoid by learning only from observational data, resembling natural human or animal learning processes [29]. Here, the agent can learn policies without having explicit access to the actions.

Overall, the IL problem has shown to be a very challenging task, especially due to the variability and multi-modality of the human demonstrations, as well as the mismatch between training and evaluation objectives [33] - models are trained to mimic the data, but the effective evaluation is made with high-level, task-related metrics (e.g., success rate).

Related Work

One of the earliest and most famous successful applications of imitation learning was the autonomous driving project by [43], where a neural network was trained to map input image observations to discrete actions to drive the vehicle.

In the last decade, learning from human demonstration has benefited from the developments in deep neural networks and optimization algorithms, allowing the model size and

complexity of applications to scale up. As the first algorithm to beat a top-performing human in the game Go, AlphaGo [49] represented another milestone towards the proficiency of such algorithms against human performance.

In the field of autonomous robotics, conventional programming entails not only a deep understanding of the task but also the ability to articulate it algorithmically. On the other hand, RL techniques rely on crafting reward functions, which also demands prior comprehension of the task. Here, IL presents an alternative paradigm to both traditional programming and RL methodologies, relying on acquiring expertise through demonstrations [60]. This approach represents a significant leap forward in democratizing robotics platforms by overcoming the need for programming expertise. It operates under the premise that human experts can showcase desired behaviors even without adept programming skills [33].

The work from [33] explores the challenges of IL in robotics and provides benchmark policies for several tasks. Their findings indicate that IL policies are highly dependent on the observation space and model hyperparameters. On the observations, they find the performance to be hardly damaged when moving from an end-effector-only model to a more complete robot representation (e.g. 49%-88% relative performance drop when adding robot joints), advocating for extensive engineering of the observation space and the information-hiding paradigm. They also find using a stochastic policy to be highly beneficial in terms of performance.

[65] performs a similar study regarding the robot's actions during imitation, finding the operation space controllers (end-effector control) to facilitate learning in over the use of joint velocity controllers, aligning with the findings from [33].

Although effective in terms of success rate, focusing the robot's proprioception solely on its end-effector imposes a limitation on the range of tasks the policy can effectively execute. In such cases, individual joint commands are derived from inverse kinematics or dynamics models, which in the case of redundant manipulators produce a multiplicity of solutions [50]. The choice of a solution must then be made using extra constraints and can diverge from the original demonstrations. This can pose a challenge in task execution in case the configuration of the whole manipulator is relevant to the task.

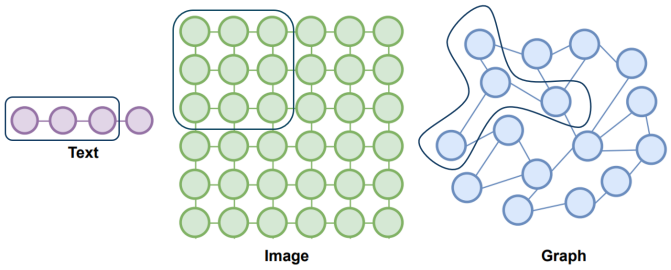


Figure 2.1.: Illustration of text, image, and graph data, highlighting the first neighbor convolution region in each of the representations, edited from [41].

2.2. Graph Neural Networks

Graphs are mathematical elements that represent a collection of entities (nodes) and their relations (edges). They provide a natural framework for modeling relationships between entities and are commonly used to model various elements, from molecules and scientific citations to physical interactions and robotic environments [41]. An undirected, homogeneous graph $\boldsymbol{G}$ with N nodes, is represented as its set of nodes $\boldsymbol{V} = \{v_i\}$ and edges $e_{ij} \in \boldsymbol{E}$ between two nodes v_i and v_j, where $|\boldsymbol{V}| = N$ and edges $|\boldsymbol{E}| = N_E$ as

$$\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E} > \tag{2.1}$$

Each node v_i contains node features $\boldsymbol{x}_i \in \mathbb{R}^F$, that provide information about the entity (e.g. atom type and aromaticity, in a molecule). Similarly, the edges e_{ij} can also hold edge features $\boldsymbol{a}_{ij} \in \mathbb{R}^{F'}$ containing information about the connection between entities (e.g. nature of the atomic bond in a molecule). Here F and F' denote the dimensions of the node and edge features, respectively. Further on we refer to the node feature matrix of the graph $\boldsymbol{x} \in \mathbf{R}^{N \times F}$ as the result of stacking the node features for all nodes.

Further on, the graph connectivity can be represented through the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$, with each element $A_{i,j} = 1$ if an edge $e_{ij} \in \boldsymbol{E}$ exists, and zero otherwise. From $\boldsymbol{A}$, the diagonal degree matrix $\boldsymbol{D}$ can also be derived, where D_{ii} is the sum of all edges with connection to node v_i [41].

Graph neural networks (GNNs) are machine-learning-based methods that operate on the unstructured graph domain. They leverage properties such as permutation invariance

and variable size and structure to provide greater generalization capabilities than regular structured models. GNNs are agnostic to the number of nodes and edges in a graph and are designed to combine and propagate the information in nodes and edges throughout the graph's existing structure without changing it.

While traditional neural networks have excelled in tasks involving Euclidean, grid-like data, many relevant real-world systems or problems can be more naturally expressed with a graph structure [62]. This gap has driven a tide of research on deep learning on graphs, among them GNNs are the most successful in coping with various learning tasks across a large number of application domains [56]. GNNs are neural architectures designed to operate on graph-structure data. They iteratively update node representations with aggregated information from neighboring nodes and their own representation from the previous iteration.

These graph-structured models have performed well in various fields, including natural language processing [55], recommendation systems [57], and, more recently, in robotics [41, 61].

Whether the prediction task is made on node, edge, or graph level (or a combination of them) is a relevant design choice when designing the model, that will determine the output of the last prediction layers.

Built upon classic neural network architectures, GNNs process graph elements in vectorized form, producing outputs within the graph domain. Multiple operators are used to propagate, sample, and extract information from graphs using learnable modules (i.e., regular neural networks). Some of them are presented below:

2.2.1. Information Propagation

Among the operations used to propagate information across nodes is a more general operation of convolution, extending the idea of CNNs that have become so widely used. Convolution is a filtering operation performed by applying a filter (kernel) to the input data, and in the graph domain, it can be applied to graph structures through a spectral or spatial approach. The first is built on top of the graph Fourier transform, which for a graph signal (node features) x is defined as $\mathcal{F}(\boldsymbol{x}) = \mathbf{U}^T\boldsymbol{x}$, where $\mathbf{U}$ is the matrix of eigenvectors from the normalized graph Lagrangian $\boldsymbol{L} \coloneqq I_N - \boldsymbol{D}^{\frac{-1}{2}}\boldsymbol{A}\boldsymbol{D}^{\frac{-1}{2}} = \mathbf{U}\Lambda\mathbf{U}^T$, with Λ diagonal [63].

Then, the convolution operation in the spectral domain for a filter $\boldsymbol{g} \in \mathbb{R}^N$ is defined as

$$\mathbf{x} *_G \mathbf{g} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{g})) = \mathbf{U}\left(\left(\mathbf{U}^T\mathbf{x}\right) \odot \left(\mathbf{U}^T\mathbf{g}\right)\right). \tag{2.2}$$

When using a simplified learnable filter $\boldsymbol{g}_\theta = diag(\boldsymbol{U}^T\boldsymbol{g})$, the spectral graph convolution becomes

$$\mathbf{x} *_G \mathbf{g} = \boldsymbol{U}\boldsymbol{g}_\theta\boldsymbol{U}^T\boldsymbol{x}. \tag{2.3}$$

Different methods provide different filter designs for $\boldsymbol{g}_\theta$, using θ as a learnable parameter. Most of them make use of neural network layers as a nonlinear function for parametrizing the update of node features throughout multiple (L) layers, in a general form, taking a graph signal $\boldsymbol{x}$, as well as the adjacency matrix $\boldsymbol{A}$ and returning node-level features $\mathbf{z} \in R^{N \times F^{out}}$, for the number of output features F^{out}.

Graph Convolutional Networks (GCNs) from [22] use an approximation of the graph convolution given by $\mathbf{x} *_G \mathbf{g}_\theta \approx \theta(\boldsymbol{I}_N + \boldsymbol{D}^{\frac{-1}{2}}\boldsymbol{A}\boldsymbol{D}^{\frac{-1}{2}})\boldsymbol{x}$, with θ as a free parameter, ending up with a more compact formulation for the information propagation. Starting from $\boldsymbol{H}^{(0)} = \boldsymbol{x}$, the layers of convolution are defined as

$$H^{(l+1)} = \sigma(\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}), \tag{2.4}$$

where $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_N$ is the adjacency matrix with added self-connections, $\tilde{\boldsymbol{D}}_{ii} = \sum_j \tilde{\boldsymbol{A}}_{jj}$ and $\boldsymbol{W}^{(l)} \in \mathbb{R}^{N \times F}$ is the weight matrix for layer l.

When introducing GCNs, [22] also shows that this structure captures the inherent structure of the graph data based on its connections even without any training, backed by the similarity of this approach with the Weisfeiler-Lehnman algorithm for testing for isomorphism. An illustration of the untrained GCN model's embeddings is presented in Figure 2.2.

The spatial approaches to propagation take inspiration from classic convolutional neural networks (CNNs) but generalize the exchange of information within the spatial neighborhood of the nodes using message passing. Composing a two-phase process consisting of message passing and readout, defining the gathering of the information from neighbor nodes, and updating node representations by combining the gathered information, respectively.

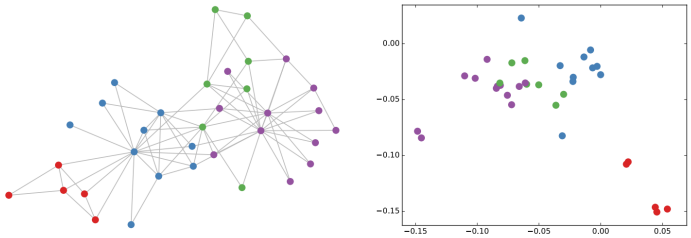


Figure 2.2.: Embeddings generated by untrained, 3-layer GCN on the ”Zachary’s karate club” graph classification Dataset, from [22]. Note that the embeddings closely resemble the input graph structure.

The message passing can be written in a general form as the following chain in the function of the node features and edge attributes as

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \bigoplus_{j \in \mathcal{N}(i)} \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right), \tag{2.5}$$

where $\bigoplus$ is a differentiable, permutation invariant combination function such as *sum*, *mean* or *max*, γ and ϕ represent also differentiable parametrized functions that are commonly modeled by Multilayer Perceptrons (MLPs) [13].

In the notation from [47] a graph convolutional layer can then be defined explicitly calculating the ”messages” $\boldsymbol{m}_{ij}$ and aggregating them as

$$\begin{aligned} \mathbf{m}_{ij} &= \phi_e \left(\mathbf{h}_i^l, \mathbf{h}_j^l, \boldsymbol{a}_{ij} \right) \\ \mathbf{m}_i &= \sum_{j \in \mathcal{N}(i)} \mathbf{m}_{ij} \\ \mathbf{h}_i^{l+1} &= \phi_h \left(\mathbf{h}_i^l, \mathbf{m}_i \right), \end{aligned} \tag{2.6}$$

where $\boldsymbol{h}_i^l \in \mathbb{R}^F$ is the dimension of the node embedding at layer l and $\mathcal{N}(i)$ represents the set of neighbors of node v_i. Here, the models that operate on the data are represented as ϕ_e and ϕ_h, also commonly approximated by MLPs.

Besides message passing and convolution, several other operations can be performed on GNNs, accentuating different properties of the graph signal and structure. For example, recurrent operators are similar to the convolution layers but keep the weights throughout different layers. Skip connections can be used when building deep models to avoid propagating noisy information and over-smoothing. Sampling methods can be useful when dealing with large-scale graphs, among other techniques[63].

E(n) Equivariant Message Passing

To handle geometric data in 3D space, [47] proposes an E(n) Equivariant GNN (EGNN) model, that separately takes into account node features depending on their variance or equivariance to Euclidean translations, rotations, and reflections. In this framework, the node features are separated into $\boldsymbol{x}_i$ and $\boldsymbol{h}_i$, the former being the node coordinates, and the latter node features (node properties invariant to coordinates, such as an atomic type). Both $\boldsymbol{x}_i, \boldsymbol{h}_i$ are updated by Equivariant Graph Convolutional Layers, that operate in these node features maintaining equivariance to $\boldsymbol{x}_i$.

Definition 2.2.1 (Equivariance [47])**.** Let X and Y be G-sets for the same group G and consider a transformation $T_g : X \to X$ on X for $g \in G$. A function $\phi : X \to Y$ is called equivariant to g if there exists an equivalent transformation on the function's output space $S_g : Y \to Y$ such that $\phi(T_g(x)) = S_g(\phi(x))$.

In practical terms, the EGNNs feature three types of equivariance on a set of nodes with coordinates $\boldsymbol{x} \in \mathbb{R}^n$. For the context of $E(3)$, the model is translation equivariant on $\boldsymbol{x}$ for any translation vector $g \in \mathbb{R}^3$, and rotation and reflection equivariant for any orthogonal matrix $Q \in \mathbb{R}^{3\times3}$. That means a transformed input regarding the node coordinates produces an equivariantly transformed output, i.e.

$$[Q\boldsymbol{x}^{l+1} + g\ , \boldsymbol{h}^{l+1}] = \text{EGCL}(Q\boldsymbol{x}^{l} + g, \boldsymbol{h}^{l}). \tag{2.7}$$

The equivariance is achieved by changing the message passing (2.6) to also contain a propagation for the node coordinates, as

$$\boldsymbol{m}_{ij} = \phi_e(\boldsymbol{h}_i^l, \boldsymbol{h}_j^l, ||\boldsymbol{x}_i^l - \boldsymbol{x}_j^l||^2, a_{ij}) \tag{2.8}$$

$$\boldsymbol{x}_i^{l+1} = \boldsymbol{x}_i^l + C \sum_{j \neq i} (\boldsymbol{x}_i^l - \boldsymbol{x}_j^l) \phi_x(\boldsymbol{m}_{ij}) \tag{2.9}$$

$$\boldsymbol{m}_{ij} = \sum_{j \neq i} \boldsymbol{m}_{ij} \tag{2.10}$$

$$\boldsymbol{h}_i^{l+1} = \phi_h(\boldsymbol{h}_{\boldsymbol{i}}^{\boldsymbol{l}}, \boldsymbol{m}_i). \tag{2.11}$$

For their equivariance property and reported performance in tasks such as the dynamical modeling of N-body systems in 3D, the EGNNs are explored in this work as architecture to tackle the imitation learning problem in robotics manipulators.

2.2.2. Pooling

Given that GNNs mostly operate on node-level representations, pooling layers play a pivotal role in tasks such as graph classification [56]. These layers are designed to derive graph-level representations from the node-level ones, compiling the collective node information into a single vectorial representation that captures the entire graph structure.

Mathematically, a pooling layer is a function that maps the set of multiple node-level features to a single, graph-level representation. Given a graph $\boldsymbol{G}$ and its node features $\boldsymbol{x}_i$, the simplest pooling layers perform basic computations such as *sum*, *mean* or pick the maximum node-feature value, e.g.

$$f_{\text{pool}}^{\text{sum}}(\boldsymbol{G}) = \sum_{\boldsymbol{x}_i \in \mathbf{X}} \boldsymbol{x}_i \tag{2.12}$$

for the sum pooling. These simple pooling layers are still widely used in the literature, given their great performance compared to more complex approaches. [35] (and more recently, [5]) argues that the information propagation (i.e. convolution) plays a much greater role in the success of GNNs on most benchmarks than complex learnable layers, encouraging the use of basic pooling for GNNs.

2.2.3. Graph Neural Networks in Robotics

In the field of robotics, using graph representations and models has been shown to be a powerful tool to learn useful spatio-temporal information [41] with a wide range of applications from the modeling of passive bodies for simulation [24] to planning of multi-agent systems [53], action recognition [1], and, most related to the present work, object manipulation tasks. The geometric data that features in robotics tasks have an explicit graph structure and are naturally modeled with graphs [41]. Their ability to model spatial and functional key points of the environment allows for meaningful planning of tasks. Works like [32, 52] focus on environment perception and modeling (particularly of deformable objects), which are shown to significantly benefit from graph representation and graph-domain learning with GNNs, while [48] attempts to extract a graph environment representation images. From a task-agnostic perspective, works like the one from [61] use GNNs for motion retargeting between a human demonstrator and a robotic platform. With a GNN encoder-decoder structure, the networks are optimized based on a loss representing the position gap between the demonstrator and the robot.

Using a scene graph representation with objects and pre-defined goal positions, the method from [27] learns tasks in an efficient and interpretable manner. It employs a high-level Graph Neural Network (GNN)-based policy alongside a low-level pick-and-place primitive to control the manipulator effectively. The high-level policy classifies nodes to determine which object from the scene graph to grasp next and which goal position it should be placed at. Given its simplicity, this approach can be trained with a small amount of demonstration data and still perform well. However, its generality is limited, as the manipulation task is restricted to compositions of pick-and-place primitives.

Similarly, [48] uses GNNs for solving visual imitation as a visual correspondence problem, using a scene graph based on 3D world coordinates of objects and their visual features over time. Here, the visual entities are represented as a hierarchical graph, with one main node per object and other nodes from the subgraph representing object features that can be used as features for the tracking, while edges represent the 3D spatial arrangement between the elements. The policy is then learned by matching similarity across the demonstrator and imitator visual entity graphs across demonstrator and imitator scene graphs along the execution of the task. [59] presents a similar graph scene, but with their coordinates in image space instead, and uses them to predict the effects of actions and use the trained model to plan towards a given goal in a model predictive control (MPC) setting.

2.2.4. Drawbacks

Despite the benefits of high synergy with the graph structure and data efficiency, the use of deep neural networks by stacking multiple layers has been shown to reduce performance due to the problem of over-smoothing, as the GCN is shown to be a special case of Laplacian smoothing [56]. This is observed with the node features becoming too similar after the information propagation.

Besides that, [2] has observed a drop in performance due to an information bottleneck when long-range dependencies between nodes are relevant to the task. That causes exponentially growing information to be squashed into fixed-size vectors and, for the classic GCNs from [22], is observed from a depth of 4 layers on.

2.3. Diffusion Models

Diffusion Models are a class of generative models that gradually perturb the original data sample x_0 to learn the underlying distribution of a dataset $p(x_0)$, from which novel data can be sampled. It is mainly described by a forward process, a reverse process, and a sampling procedure, all consisting of Markovian processes that iteratively transform the data sample [8, 7]. The distribution $p(x_0)$ is modelled as

$$p(x_0) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t), \tag{2.13}$$

for a model with parameters θ.

In the forward diffusion chain, small amounts of noise ϵ_t are added to the sample until at timestep T its distribution is fully corrupted (e.g., $p(x_T)$ is a Gaussian distribution). The forward transition, unlike VAEs, doesn't involve any trainable parameters, and is entirely determined by the noise configuration and the transition chain. Here, noise is added in a series of distribution transitions guided by hyperparameters such as the noise schedule β_t, which regulate noise intensity throughout the process. From the multiple diffusion model frameworks, Denoising Diffusion Probabilistic Models (DDPMs) were presented by [16] and are one of the most popular for discretely formulated diffusion models [8, 10, 7]. It proposes modeling the prior distribution, as well as the distribution of the noise added step-by-step in the forward chain as isotropic Gaussian noise, i.e. $\epsilon_t \sim \mathcal{N}(0, I)$, and has

$$p(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I) \tag{2.14}$$

as a formulation for the forward chain. Using $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \Pi_{k=1}^{t}\alpha_k$, the noised sample at diffusion time t can be written in closed-form in function of the sample x_0 and t as

$$p(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I), \tag{2.15}$$

which allows for parallelized training in the diffusion timestep t.

By controlling the amount of noise ϵ_t to be added to the sample at time t, the noise scheduling β_t is able to balance between exploration and exploitation of the training data [8]. It should be sufficiently big to generalize well and sufficiently small to allow the convergence of the optimization [8]. Besides the appropriate noise schedule, maintaining good performance also requires the terminal distribution $p(x_T)$ not to differ greatly from the original $p(x_0)$, maintaining as many structures as possible from the original distributions [8]. One way to address this is by maintaining a similar mean and variance through an initial normalization of the data. [16] scales all data to the interval $[-1, 1]$, to ensure that the neural network reverse process operates consistently.

In the reverse process, the model θ is trained to iteratively remove noise between two consecutive timesteps in the following chain

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{2.16}$$

for the DDPM method [16], given a sampled t and $\epsilon \sim \mathcal{N}(0, I)$ the model is optimized to approximate the noise ϵ taking gradient descent steps on the simplified loss function

$$\mathcal{L}_{DDPM}(\theta) = \mathbb{E}_{\mathbf{x}_0,\boldsymbol{\epsilon}}||\epsilon - \epsilon_\theta(t, \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t})||^2. \tag{2.17}$$

Once the model parameters θ are optimized, new samples x_T can be pulled from the initially noisy distribution and transformed by the model for T timesteps to generate a new sample $x*_0 \sim p_{\theta*}(x_0) \approx p(x_0)$.

That is done in the following reverse chain

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sigma_t \mathbf{z}, \tag{2.18}$$

with $\mathbf{z} \sim \mathcal{N}(0, I)$ and $\sigma_t = \sqrt{\beta_t}$ [16].

For practical applications, distributions can often better be described as conditional distributions, i.e. $p(x_0|c)$, for a condition $c \in \mathcal{C}$. The conditioning of the distribution acts like adding a guidance mechanism to the pipeline, that allows for change in the denoising direction. In this case, the conditioning information c can be added to the transition step as

$$p(x_0|c) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t, c), \tag{2.19}$$

incurring in adapting equations (2.16), (2.17) and (2.18) accordingly [31].

2.3.1. Graph Diffusion

Graph generation is an important computational task with numerous applications in different fields, including molecule design combined with 3D structure, traffic forecasting, natural language processing, and 3D motion synthesis [9, 26, 28]. Inspired by the success of Denoising Diffusion models in the field of computer vision, this framework has been recently applied to the graph generation problem and led to promising results [11, 18].

Among several diffusion-based graph generative models, [18] managed to outperform previous methods for the task of 3D molecule generation, i.e., predicting the 3D configuration of the atoms in a molecule, by jointly injecting Gaussian noise to the latent variables and node coordinates (in their case, categorical).

Closer to the problem tackled in this work, DiffuPose applies graph-based diffusion models to the Human3.6M dataset, containing 3D joint positions of humans. The authors try to predict the 3D configuration of human joints in space based on a 2D joint representation (image-originated). For that, a Denoising DDPM [16] is used with a Graph Convolutional Network [22] as a denoising function, conditioned in the 2D image coordinates. Their objective, given a fixed graph structure, is to iteratively denoise the node features, i.e. the 3D joint positions, according to the chains (2.15) and (2.16). For each denoising step t, they attempt to generate 3D joints $\boldsymbol{x}_t \in \mathbb{R}^{N\times 3}$, concatenating it with the 2D

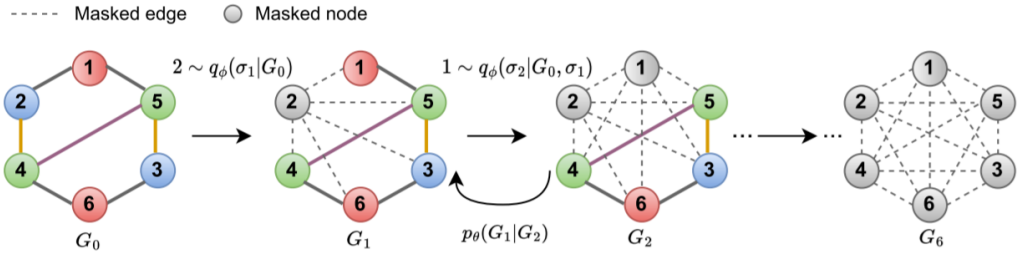


Figure 2.3.: Autoregressive Diffusion Forward (and reverse) chain for molecule graph generation, from [23].

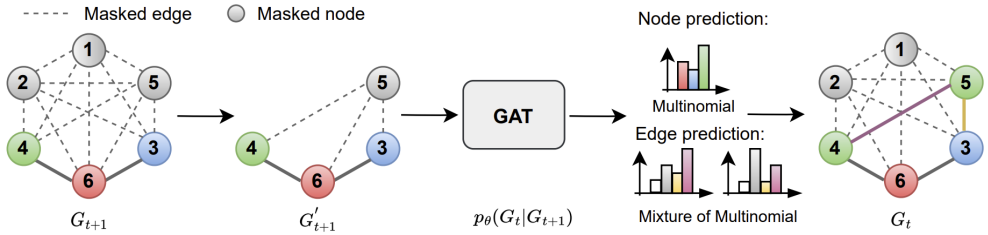


Figure 2.4.: Generative chain of the Autoregressive Diffusion Model for Graph Generation, from [23].

image positions $\boldsymbol{y} \in \mathbb{R}^{J\times 2}$, used as conditioning. Due to the explicit consideration of the connectivity by the GCN model, this setup is able to outperform state-of-the-art 3D Human Pose Estimation methods.

[23] builds on top of the Autoregressive Diffusion Models from [17] to design Autoregressive Diffusion Models for Graph Generation (GraphARM) as an efficient method for molecule generation consisting of a graph with categorical node and edge features. The autoregressive diffusion features the idea of absorbing discrete diffusion from [3] - i.e. a Markov destruction process, where at each timestep t one dimension in the data is decayed into an absorbing state (a mask token). This diffusion process converges into a stationary distribution, with all mass on the absorbing state [23]. GraphARM uses an autoregressive diffusion process to corrupt the graph signal by masking one node per diffusion step until the graph is fully masked, as in Figure 2.4.

Definition 2.3.1 (Absorbed Node [23])**.** When a node $v_i \in G$ in the absorbing state, it will have its node features masked and it will be connected to all other nodes in $\boldsymbol{G}$ by masked edges.

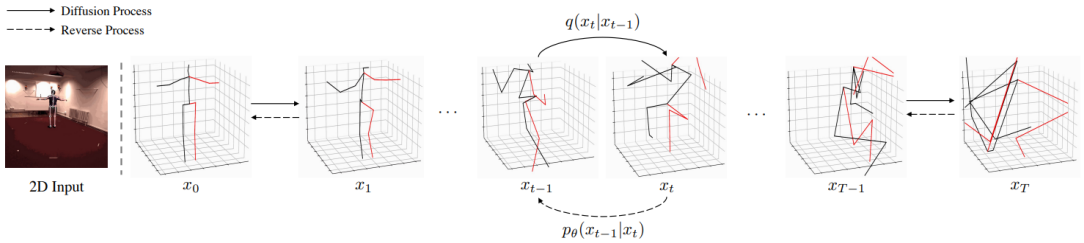


Figure 2.5.: Graph Diffusion Chain for human 3D motion prediction[11].

Regarding the order in which nodes are absorbed, called the node decay ordering σ_t, GraphARM introduces a diffusion ordering network $q_\phi(\sigma_t|\boldsymbol{G}_0, \sigma_{<t})$ to predict the next node to be sampled at each step t. The diffusion ordering network is then trained using the REINFORCE algorithm [54] using the negative likelihood of the data as a reward function. In the reverse process, the (discrete) value for each node is sampled from a categorical distribution given by the denoising network $p_\theta(\boldsymbol{G_t}|\boldsymbol{G_{t+1}})$. Graph Attention Networks (GATs) are used both for the diffusion ordering and the denoising models, which are trained to maximize the expected likelihood of the underlying data.

2.3.2. Diffusion Models in Robotics

In the work from [7], diffusion models are shown to be effective motion planners for robots in long-horizon settings, effectively encoding multimodality of the underlying data. Here, the planner is faced from an imitation learning perspective, learning the trajectory prior distribution from expert data while also incorporating optimization regarding relevant cost functions for the planner, e.g. to ensure trajectory smoothness and that the initial and goal states are respected along the whole diffusion process. Sampling with guidance is also explored, estimating a task-conditioned posterior $p(\tau|\mathcal{O})$ (where $\mathcal{O}$ is an encoding for the task).

More recently, [10], a visuomotor policy is defined as a conditional denoising diffusion process. This model learns to map visual and vector observations to robot actions, encompassing end-effector position, orientation, and gripper position. They make use of DDPM models on top of convolutional Unet models and Transformers. Integration into the robotic platform occurs via a receding-horizon control scheme. While the model generates long-horizon predictions, re-planning is triggered after shorter action horizons to maintain responsiveness.

Applied to behavioral cloning, Diffusion Policy is introduced as an extension of classical explicit and implicit policies. It addresses limitations inherent in these traditional approaches, such as the absence of multi-modal behavior, the inability to execute high-precision tasks using direct observation-action mapping models, and training instability, particularly in optimizing implicit Energy-Based Models.

3. Methodology and Approach

In this chapter, we present the comprehensive methodology and approach adopted in this thesis to address the imitation learning (IL) problem in robotics as a graph generation problem. The core problem and hypothesis are exposed, emphasizing the need for an algorithm that not only learns from demonstrations but also exhibits awareness of the robot's embodiment and environment throughout task execution. Subsequently, we introduce the specifics of the graph data representation alongside the control setting for the simulated robot environment and benchmarking methodology against state-of-the-art methods (Section 4.1). Finally, we present two distinct approaches - GraphDDPM and ARGG Policies, along with their respective components and architectures, for tackling the IL problem within a graph-based generative framework.

3.1. Problem Statement and Hypothesis

The imitation learning (IL) problem introduced in Section 2.1 consists of training an agent to mimic the behavior of an expert demonstrator by learning a policy that maps observations to actions, thus enabling a robotic platform to perform tasks based on demonstrations. Under this approach, our system consists of an environment and an agent whose actions on the environment are determined by a policy. The policy, typically represented by neural networks in IL, is responsible for generating the robot's actions based on observations from the environment.

The main challenges encompass defining a graph representation, creating a corresponding graph-structured dataset and environment, and devising a policy capable of learning from demonstrations within this graph framework and executing tasks.

Our approach aims at a policy endowed with awareness of the robot topology and its environment, using that information not only to learn (supervision) but also when performing

a task. Contrary to most high-performing policies mentioned in Chapter 2, which only account for the end-effector, we seek to enable the robot to learn the distribution over joint velocity trajectories and leverage the connectivity of the robot's components through the graph representation. Further on, we intend the designed policy to allow for a flexible environment as input, with a variable number of objects, which is possible when using GNNs in the scene representation. It is also desired that, as in [10, 7], the policy can capture multimodality in the underlying distribution over the demonstrated trajectories.

At last, there is also potential for further development towards an embodiment-agnostic method, capable of imitating behavior across slightly different topologies, thus approaching the IfO problem.

3.1.1. Receding Horizon Control Setting

The policy operates in a closed-loop setting, following a receding horizon control scheme [34]. Here, the policy $\boldsymbol{x}_{A_{t,t+1,\dots t+H_{pred}}} = \pi_\theta(\boldsymbol{x}_{O_{t,t-1,\dots,t-H_{obs}}})$ is optimized offline based on the demonstrations to minimize the cost

$$\mathcal{L} = ||\hat{\boldsymbol{x}}_{A_{t,t+1,\dots t+H_{pred}}} - \boldsymbol{x}_{A_{t,t+1,\dots t+H_{pred}}}||_2^2 \tag{3.1}$$

using stochastic gradient descent (AdamW), where the vector $\boldsymbol{x}_{A_t}$ encapsulates the joint actions (i.e. joint velocities) at time t and $\boldsymbol{x}_{O_t}$ the respective joint observations.

During execution, at time step t the policy takes the latest H_{obs} steps of observation data $\boldsymbol{x}_{O_t}$ and predicts H_{pred} steps of actions, of which H_{act} steps of actions are executed on the robot without re-planning. We will then follow the nomenclature from [10] and call H_{obs} the observation horizon, H_{pred} the prediction horizon and H_{act} action horizon. This encourages temporal action consistency while maintaining responsivity.

As illustrated in Figure 3.1, the policy (represented by its parameters θ) recomputes the actions for the whole prediction horizon H_{pred} after executing H_{act} steps. Parallel to classic MPC [44], there is no online optimization but only a policy optimized offline to reproduce the demonstrations.

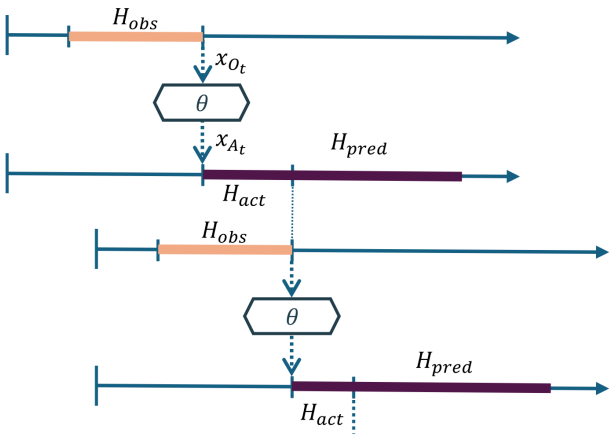


Figure 3.1.: Illustration of the recomputation of the actions by the policy, concerning the observation, action and prediction horizons.

3.1.2. Representing Rotations

The work from [64] tackles the discontinuity problem of the quaternion and Euler rotation representations to design a 6D [1] continuous representation and empirically prove its performance gains for neural networks in several applications. [10] extends this to unit quaternions and implements transformers to and from the 6D continuous rotation, which is also utilized for the transformation from quaternions to 6D in this work.

The mapping g_{CS} performs the conversion between a rotation matrix and the 6D continuous representation. It operates on a rotation matrix $X \in SO(3)$. The transformation to the 6D continuous is rather straightforward, dropping the last vector from the rotation matrix as

$$g_{GS}\left(\begin{bmatrix} | & | & | \\ a_1 & a_2 & a_3 \\ | & | & | \end{bmatrix}\right) = \begin{bmatrix} | & | \\ a_1 & a_2 \\ | & | \end{bmatrix}, \tag{3.2}$$

where $a_i, i = 1, 2, 3$ are column vectors.

[1] More generally, the authors from [64] present a $n^2 - n$-dimensional continuous representation for n dimensional translation and rotations.

The reverse mapping f_{GS} to a rotation matrix, however, requires the reconstruction of that third basis vector through a Gram-Schmidt-like orthogonalization, i.e.

$$
\begin{aligned}
f_{GS}\left(\begin{bmatrix} | & | \\ a_1 & a_2 \\ | & | \end{bmatrix}\right) &= \begin{bmatrix} | & | & | \\ b_1 & b_2 & b_3 \\ | & | & | \end{bmatrix} \\
b_i &= \begin{bmatrix} \begin{cases} N(a_1) & \text{if } i = 1 \\ N(a_2 - (b_1 \cdot a_2)\, b_1) & \text{if } i = 2 \\ b_1 \times b_2 & \text{if } i = 3 \end{cases} \end{bmatrix}^T ,
\end{aligned}
\tag{3.3}
$$

where $N(v) = v/\|v\|$ is a normalization operator and b_i are also column vectors.

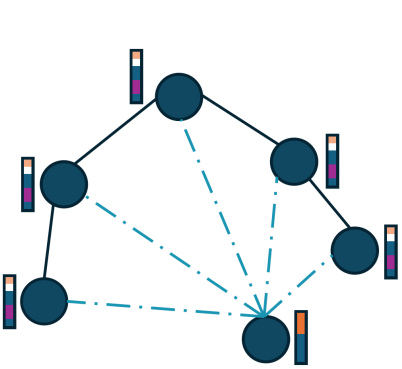
The rotations in the designed graph representation are transformed according to (3.2)-(3.3), to avoid the above-mentioned discontinuities.

3.1.3. Graph Representation

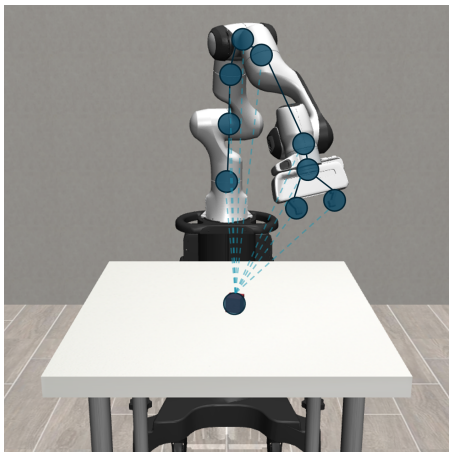
Given its geometric nature, a robot and its surrounding environment can be very naturally modeled as a graph structure, as demonstrated by various authors in recent literature [41]. For instance, [27] introduces a scene graph with nodes being objects or predefined goal positions, and uses such a representation and annotated graph data to train a Graph classifier used as a high-level policy of what object to pick and where to place it. [48, 59] represent the objects in the scene and the robot's end-effector using their coordinates to predict actions and future steps from the scene, respectively. These are more similar to our approach, which will also represent the scene and end-effector and predict future states, but also contain the robot's embodiment with all its joints. Key considerations in designing such graphs include selecting appropriate node and edge features and defining temporal horizons within the data, specifically in imitation learning.

Let $\boldsymbol{G} =< \boldsymbol{V}, \boldsymbol{E} >$ be a graph with nodes $v_i \in \boldsymbol{V}$ and undirected edges $e_{ij} \in \boldsymbol{E}$, where each node v_i represents either a robot joint or an object in the scene.

We denote the group of robot joint nodes as $\mathcal{J} \subset \mathbf{V}$ and the object nodes as $\mathcal{O} \subset \mathbf{V}$ such that $\mathcal{J} \cup \mathcal{O} = \boldsymbol{V}$ and $\mathcal{J} \cap \mathcal{O} = \{\}$. Each v_i contains node features $\boldsymbol{x}_i$ and coordinates $\boldsymbol{p}_i$ in the 3D scene. Each edge $e_{ij} \in E$ contains an edge feature a_{ij} representing either a robot link or a physical interaction, depending on which nodes it connects. Specifically, if $v_i, v_j \in \mathcal{J}$, then a_{ij} signifies a link between these components; otherwise, it denotes a



(a) Illustration of a generic scene graph with 5 robot joints and an object, represented with illustrative node features and different edge types.



(b) Illustration of scene graph representation for the Lift Robosuite [65] environment

Figure 3.2.: Graph representations of the robot and environment

physical interaction. The node task-space coordinates p_i are responsible for providing a geometric description of the scene, and are further used to provide E(3) equivariance to our graph models extending on [47].

The node-level feature $\boldsymbol{x}_i$ provides the relevant information for the description of that node. Therefore it contains joint values in the case of robot joints and rotations in the case of objects. To ensure the homogeneity of the graph, the dimensions of the node features are chosen after the node that requires the most dimensions, masking the unused dimensions as zero.

A distinction is made between the observed node features $\boldsymbol{x}_{O_t}$ and the action node features $\boldsymbol{x}_{A_t}$ at time t regarding (especially, but not only) the time horizon of the node features. $\boldsymbol{x}_{O_t}$ contains node-level observations for the previous H_{obs} steps from the inference time t, whereas the actions refer to the future H_{pred} steps.

We refer to $\boldsymbol{G}_{O_t}$ as the observation graph and to $\boldsymbol{G}_{A_t}$ as the action graph at time t. They differ only on the contents of their node-level features but present the same connectivity.

Action Node Features

The action node features $\boldsymbol{x}_{A_t} \in \mathbb{R}^{N \times H_{pred}}$ contain the desired action values, during the prediction horizon. For each of the nodes corresponding to robot joints $v_i \in \mathcal{J}$, these consist of the individual joint velocities. Regarding the objects, we don't see the need to represent any action, so the node features respective to $v_i \in \mathcal{O}$ are masked with zeros, only to maintain the homogeneity of the graph.

Observation Node Features

The observation node features $\boldsymbol{x}_{O_t}$ contain the observed values accumulated for the observation horizon H_{obs} at time t. For each object $v_i \in \mathcal{O}$, the object's orientation is used as an observation feature together with a unique node ID i [2]. Here, the 6D representation exposed in Subsection 3.1.2 is used, resulting in the node features $\boldsymbol{x}_{O_t} \in \mathbb{R}^{N \times H_{obs} \times 7}$.

The node-level observations for $v_i \in \mathcal{J}$ consist only of their observed joint angles, as well as the unique node ID. The orientations for the individual robot joints are considered irrelevant to the performance of the task, given that the node coordinates $\boldsymbol{p}_i$ for the individual gripper fingers already encode the information of end-effector orientation. To guarantee homogeneity, the robot joint node features are masked with zeros in the (5) remaining dimensions.

Graph-structured Dataset

The graph-structured dataset is then created following the representation described above, based on the Robomimic [33] datasets in the *lowdim* modality (not containing images). More information about the contents of the resulting graph datasets is presented in Table 3.1.

[2] Despite not being invariant to E(3) rotations, the object's rotation is used as a node feature in the EGNN networks as an object property, to assure it's being considered by the policy.

Table 3.1.: Summary of the created graph dataset for all tasks.

	# Nodes	# Edges	Action dim	Obs dim	Edge dim	# Graphs ph	# Graphs mh
Lift	10	17	H_{pred}	$[H_{obs}, 7]$	1	5466	24827
Square	10	17	H_{pred}	$[H_{obs}, 7]$	1	25954	74431
Transport	21	71	H_{pred}	$[H_{obs}, 7]$	1	89552	189500

3.2. Approaches and Procedures

The literature on graph generative models presents two distinct approaches regarding the generation patterns: autoregressive and one-shot generation [28] [3]. Both approaches involve decomposing the graph generation problem into multiple classification or regression tasks. For instance, in autoregressive graph generation, node and edge features are predicted at each generation step. In comparison, denoising diffusion models approximate node feature noise at each step during denoising.

This section handles both an autoregressive and a one-shot diffusion approach, focusing on their implementations and nuances in addressing the challenge of generating graph-structured trajectories for robotic manipulation within the context of IL.

3.2.1. Graph Encoder

When processing an observation from the environment, represented as a scene graph $\boldsymbol{G}_{o_t}$, the graph encoder plays a crucial role in transforming it into a global-level feature that captures the current state of the environment. This encoded feature enables the policy to make informed decisions about subsequent actions.

The graph encoder primarily consists of an EGNN [47] architecture, comprising L equivariant graph convolution layers (EGCL), that implement message passing as outlined in (2.8)-(2.11). Following these convolutional layers, a global pooling layer generates a feature of fixed size per graph object, and a linear layer is applied to adjust the output dimension.

[3] To clarify the nomenclature, the one-shot generation predicts in a single shot only one denoising step for all data dimensions. It, however, does not provide a final sample after a single prediction.

Additionally, the node IDs contained within the observation node features undergo an Embedding layer transformation. These embeddings are then concatenated with the input observation node features before being fed into the EGCL layers, as illustrated in Algorithm 1.

Algorithm 1 Graph Encoder

Require: $\boldsymbol{G}_{o_t}$
1: $\mathbf{h}_{v_t}^0 \leftarrow [\boldsymbol{x}_O, \text{Embedding}(\boldsymbol{x}_{id})]$
2: **for** $l \leftarrow 1$ to L **do**
3: $\quad \mathbf{h}_{v_t}^l, \boldsymbol{p}_{v_t} \leftarrow \text{EGCL}(\boldsymbol{h}_{v_t}^{l-1}, \boldsymbol{p}_{v_t}, \boldsymbol{h}_e)$
4: **end for**
5: FiLM $\leftarrow f_{pool}^{mean}(\mathbf{h}_{v_t}^L)$
6: **return** FiLM

In this case, the encoder outputs the "FiLM Generator" function as described by [40], meaning scale and bias vectors to modulate the neural network's channel-wise activation for all message-passing (convolution) layers.

3.2.2. GraphDDPM

Inspired by the application of diffusion models for trajectory generation in [7, 10] and the work from [11] estimating 3D human positions, we aim to leverage the same advantages in our work, such as the high-quality samples for long horizons trajectories and ability to learn multimodal behavior. This approach builds on top of one of the most widely used frameworks within diffusion models, the Denoising Diffusion Probabilistic Models from [16], using the DDPM framework to denoise node features in the same fashion as [11]. Here, at each diffusion step k, Gaussian noise $\boldsymbol{\epsilon}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is injected into the node features of the action graph $\boldsymbol{x}_{A_t}$ described in 3.1.3, and a Graph model is then trained to iteratively remove the node-level added noise. The reverse (denoising) process is then conditioned on the observation graphs $\boldsymbol{G}_{O_t}$, directing the final action according to the last observations of the scene graph.

Forward Diffusion

In the forward diffusion chain, noise is injected into the original node features $\boldsymbol{x}_t^0 \coloneqq \boldsymbol{x}_{A_t}$ [4] using the DDPM Noise Scheduler with a variance schedule β_k following

$$\mathbf{x}_t^k \sim \mathcal{N}(\sqrt{\beta_k}\mathbf{x}_t^{k-1}, \beta_k \mathbf{I}) \tag{3.4}$$

in a Markov chain, gradually adding noise to the data.

Here, no optimization is done to the model [16]. As mentioned in Section 2.3, the DDPM forward chain can be written in closed form to admit sampling $\boldsymbol{x}_t^k$ arbitrarily to k, using the form in (2.15). This allows the training process to be parallelized by supervising the noise prediction at multiple denoising steps k.

During the forward whole chain, both actions and observations are normalized to be contained in the $[-1, 1]$ interval, as done in [16, 10]. That's done to avoid a too-large discrepancy between the terminal distribution $p(\boldsymbol{x}^K)$ and $p(\boldsymbol{x}^0)$, which is a cause for suboptimal learning, inefficiency, and slow convergence [8].

The noise schedule is responsible for controlling the variance of the noise added to the samples in the diffusion forward chain. The one used to train this model is the Squared Cosine Schedule from iDDPM [36], as it presented the best results in the proposed tasks in [10]. The noise schedules α_k, β_k are by definition related to each other via $\beta_k = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}$, cf. Section 2.3.

this cosine schedule consists of a smoother alternative to a linear noise scheduler from [16], proposed to allow more uniform destruction of the information. The noise schedule is defined as

$$\bar{\alpha}_k = \frac{f(k)}{f(0)}, f(k) = \cos(\frac{t/T + s}{1 + s} \cdot \frac{\pi}{2})^2, \tag{3.5}$$

where s is a small offset to prevent β_k from being too small.

[4] Here the subscript A is set aside to simplify the notation, the node features updated in the diffusion chains are always the action node features.

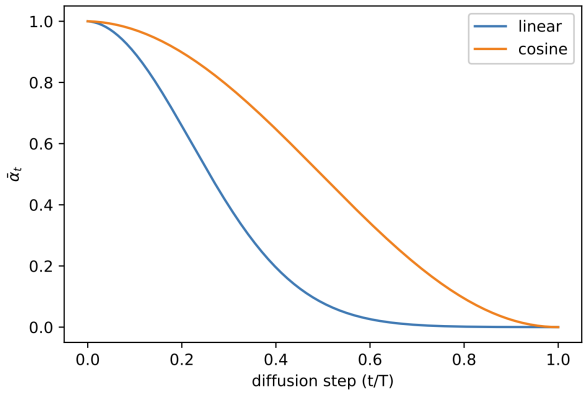


Figure 3.3.: Squared sosine schedule from [36] in comparison to the linear schedule from the original DDPM [16].

Reverse Diffusion

In the reverse process, the noise ϵ_k added to the node features $\mathbf{x}_0$ is estimated by the model $\boldsymbol{\epsilon}_\theta(\boldsymbol{G}_{O_t}, \mathbf{x}_t^k, k)$ conditioned in the observations in the reverse chain can be obtained by extending equation 2.16 to account for the observation features as

$$\boldsymbol{x}_t^{k-1} = \frac{1}{\sqrt{\bar{\alpha}_k}}(\boldsymbol{x}_t^k - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}}\boldsymbol{\epsilon}_\theta(\boldsymbol{G}_{o_t}, \boldsymbol{x}_t^k, k)) + \sigma_k \mathbf{z}. \tag{3.6}$$

The model is then trained to reduce the noise prediction error, using it as a loss function, as the simplified DDPM objective function in

$$\mathcal{L}_{GraphDDPM}(\theta) = \mathbb{E}_{\boldsymbol{x}_0, \boldsymbol{G}_{o_t}, \boldsymbol{\epsilon}} ||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\boldsymbol{G}_{o_t}, k, \sqrt{\bar{\alpha}_k}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_k}\boldsymbol{\epsilon})||^2 \tag{3.7}$$

The reverse (generative) process for the GraphDDPM model is illustrated in Figure 3.4

The minimization of this loss function is performed employing the Adam optimizer with decoupled weight decay (AdamW) [30]. During training, the model's parameters are iteratively updated based on the gradients from the loss function in Equation (3.7). The optimizer AdamW is based on Adaptive Moment Estimation (Adam) [21] but with the weight decay decoupled from the number of optimization steps taken with respect to the loss function [30].

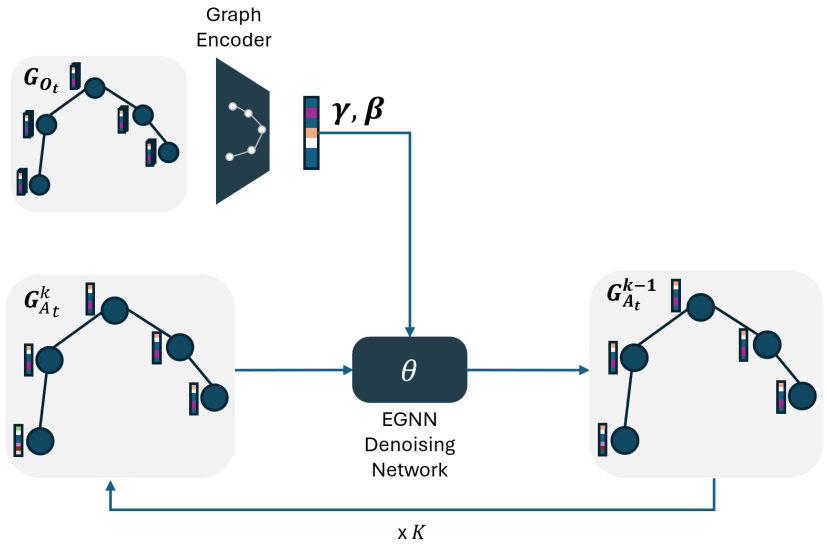


Figure 3.4.: Reverse diffusion chain for GraphDDPM Policy for conditional trajectory generation based on the observation graph.

To improve training stability, a learning rate scheduler is also used. This consists of a cosine learning rate scheduler with a linear warmup of 500 gradient steps, as presented in Appendix A.1.

Graph Conditional Denoising Network

The Graph Conditional Denoising Model $\epsilon_\theta(\boldsymbol{G}_{O_t}, \mathbf{x}_t^k, k)$ essentially projects all information to an embedding space, creates a graph-level conditioning tensor from the observation graph, propagates the information through the graph with message passing and returns the node features to the desired output dimension with linear layers.

First, a diffusion step encoder (2-layer fully connected U-Net) is used to account for the diffusion step k in the predictions, besides that, an embedding layer is responsible for projecting the node IDs into a continuous embedding space. The node and edge features are then projected to the same continuous embedding space using single-layer MLPs, as in

$$\boldsymbol{h}_x^0 = \text{MLP}_1(\boldsymbol{x}_O),\ \boldsymbol{h}_e = \text{MLP}_1(e_{ij}),\ \boldsymbol{h}_k = \text{MLP}_2(k). \tag{3.8}$$

The graph encoder, as described in 3.2.1, is part of this model. It is responsible for generating the conditioning tensor from the graph observations incoming from the environment. Following that, L message-passing layers led by conditioning layers are responsible for the information propagation and guidance of the denoising process on the graph-level conditioning embedding. The conditioning is made using Feature-wise Linear Modulation (FiLM) [40], in the same fashion as [10]. Here, the FiLM conditioning is done between message-passing rounds in the graph neural network, as described in Algorithm 2.

Each round of message-passing consists of an equivariant graph convolutional layer, described in (2.8)-(2.11), and uses the node coordinates, connections and edge attributes from the observation graph $\boldsymbol{G}_{O_t}$.

Finally, a node prediction layer (2-layer MLP) takes the final node embeddings and outputs node-level values supervised as the diffusion step noise ϵ_k as

$$\epsilon_\theta(\boldsymbol{G}_{O_t}, \boldsymbol{x}^k, k) = \text{MLP}_2(\boldsymbol{h}_{v_i}^L). \tag{3.9}$$

Algorithm 2 FiLM Conditioning Between Message Passing Layers

Require: $\boldsymbol{G}_{o_t}$, $\mathbf{h}_{v_t}^k$
1: FiLM $\leftarrow \phi_{encoder}(\boldsymbol{G}_{o_t})$
2: $\boldsymbol{\gamma}$, $\boldsymbol{\beta} \leftarrow$ FiLM
3: **for** $l \leftarrow 1$ to L **do**
4: $\quad \mathbf{h}_{v_t}^k \leftarrow \gamma_l * \boldsymbol{h}_{v_t}^k + \beta_l$
5: $\quad \mathbf{h}_{v_t}^k, \boldsymbol{p}_{v_t} \leftarrow$ EGCL($\boldsymbol{h}_{v_t}^k$, $\boldsymbol{p}_{v_t}$, $\boldsymbol{h}_e$)
6: **end for**
7: **return** $\mathbf{h}_{v_t}^L$

Sampling Procedure

During generation, the action node features are sampled from a standard Gaussian distribution $\boldsymbol{x}_t^K \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, and go through a chain of K denoising steps, following the DDPM noise schedule scheme [16]. In the graph conditional denoising network, the graph structure is taken from the observation graph $\boldsymbol{G}_{o_t}$, along with the noised node features $\boldsymbol{x}_t^K$ to predict the step noise ϵ_θ^k. The step noise is iteratively removed from the node features in the chain

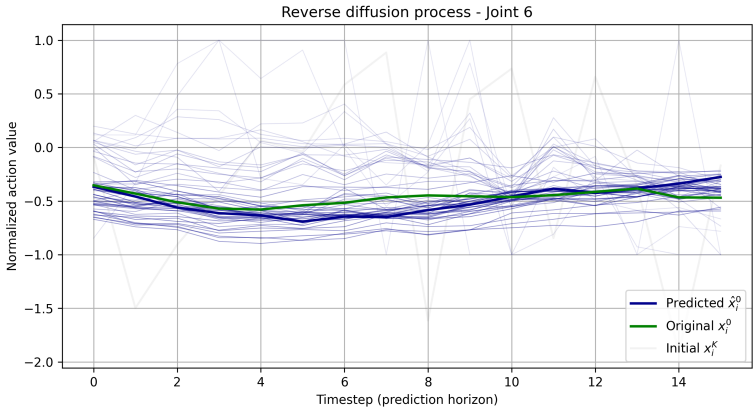


Figure 3.5.: Reverse Process for a single joint using the trained GraphDDPM model. One single action sample is generated from an observation in the Lift (ph) dataset, and displayed for multiple diffusion steps k. The equivalent for all joints is presented in Appendix A.5.

$$\boldsymbol{x}_t^{k-1} = \frac{1}{\sqrt{\alpha_t}}(\boldsymbol{x}_t^k - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\boldsymbol{G}_{o_t}, \boldsymbol{x}_t^k, k)) + \sigma_t \mathbf{z}. \tag{3.10}$$

The result after K diffusion steps is a sampled $\boldsymbol{x}_t^0$, consisting of the node features for the action $\boldsymbol{G}_{A_t}$ at time t. The denoising chain process of a trained GraphDDPM model is presented in Figure 3.5.

Mini-Batch Optimization

In deep learning, a distinction is made between batch gradient optimization and stochastic gradient algorithms. The first involves accumulating gradient computations over a single pass through the entire training dataset before updating parameters. In contrast, most of the methods opt for the stochastic approach (also referred to as online methods), taking optimization steps from gradient averages computed over small subsets of the complete training set, known as mini-batches [14]. Mini-batch training aims to reduce the number of model calls by conducting a single prediction for multiple data samples and retrieving the results efficiently.

When training the GraphDDPM policy, challenges arise in constructing these mini-batches due to the model's nature as both a graph-based and diffusion model.

Given the nature of graph-represented data and graph models, conventional batching methods are inapplicable. However, with GNNs being agnostic to the number of nodes and edges in the data, the output of a GNN contains the same number of nodes as the input data, and if they were stacked in the node feature dimension, the information from one graph would be distributed throughout other samples of our dataset. That means our mini-batch can't be simply stacked, but constructed to form a single graph where individual samples are concatenated along the node dimension but remain unconnected to each other, i.e.

$$\mathbf{A}_{batch} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_B \end{bmatrix}, \qquad \mathbf{X}_{batch} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_B \end{bmatrix} \tag{3.11}$$

where $\boldsymbol{A}_{batch}$ and $\boldsymbol{X}_{batch}$ represent the mini-batch adjacency matrix and node feature vector, respectively.

The lack of inter-sample node connections inhibits message passing between nodes from different samples. This style of graph mini-batching is commonly employed and implemented in PyTorch Geometric [13]. To mitigate computational and memory overhead associated with using a block diagonal adjacency matrix, edges are represented in an edge list, which includes only existing edges.

Furthermore, a batch index is assigned to each node, which can be used by the model, for instance, in pooling layers. In these pooling layers, the batch index can be used to prevent information from spreading from one graph to the entire mini-batch, confining the global-level results of the pooling layer to the current sample only.

$$\text{batch} = \begin{bmatrix} 0 & \cdots & 0 & 1 & \cdots & n-2 & B-1 & \cdots & B-1 \end{bmatrix} \tag{3.12}$$

From the perspective of the denoising diffusion, the closed form for the node features along the diffusion trajectory in (2.15) allows us to sample $\boldsymbol{x}_t^k$ arbitrarily w.r.t. k. In the mini-batch, one timestep k is randomly sampled *per* graph, resulting in

$$\text{timesteps} = \begin{bmatrix} \tilde{k}_1 & \dots & \tilde{k}_B \end{bmatrix}, \tag{3.13}$$

where $\tilde{k}_i \sim \mathcal{U}_{[0,k]}$.

This approach enables training the GraphDDPM model with an arbitrary batch size, optimizing efficiency while preserving the integrity of the graph structure.

Expected Properties

The model is expected to capture multimodal behavior, as in [10]. Additionally, its equivariance to translations, rotations, and reflections in Euclidean space is expected to be advantageous, particularly in learning from heterogeneous environments. This property enhances data efficiency by inherently accommodating such transformations within the model, obviating the need for extensive data generalization.

Given its foundation as a Graph Neural Network (GNN)-based model, it is expected to benefit from the inherent graph representation of the environment, leading to improvements in generalizability and sample efficiency.

Finally, while [10] offers a solution to the sampling time problem by leveraging iDDPM to reduce the number of diffusion iterations, it still requires significant computational resources and parameters compared to the GraphDDPM model, which will be discussed in detail in Section 4.5.

3.2.3. Autoregressive Graph Generative Policy (ARGG)

The autoregressive approach builds on the work from [23] and attempts to apply their Autoregressive Diffusion Model for Graph Generation to generate the robot's action graph for the prediction horizon. As in the previous approach, this reverse process is conditioned on observations consisting of the whole observation graph $\boldsymbol{G_{O_t}}$ for a past observation horizon.

The exploration of this approach aims to leverage a key advantage of the autoregressive diffusion models: their potential to substantially decrease computational costs, particularly within the size range of graphs employed in this project. Here, the graph signal is absorbed one node at a time, implying that the number of diffusion steps is equal to the number of nodes in the graph [17, 23]. This quantity, for the utilized environments, is notably smaller compared to the number of diffusion steps in the GraphDDPM policy, as well as in [10]. The reduced sampling time directly enhances the responsiveness of the resultant system by enabling more frequent re-planning cycles within the same timeframe compared

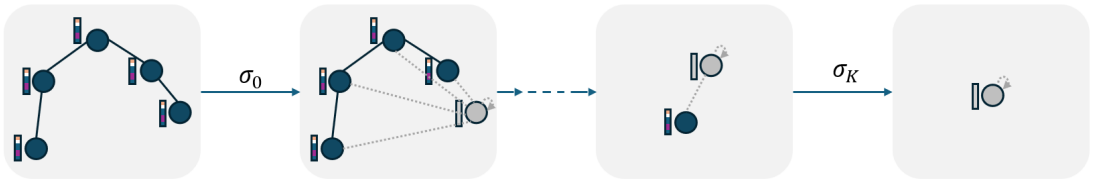


Figure 3.6.: Forward (Destructive) Chain for the Autoregressive Graph Generative Policy.

to the GraphDDPM approach. Following the general pipeline for diffusion models, this approach can be summarized into a forward diffusion process, a reverse diffusion process, and a sampling procedure, presented below.

Forward Process - Absorbing Diffusion

The forward chain consists of an absorbing diffusion process, where one dimension (in our case, one node) of the training data is absorbed per diffusion step until a single masked node remains [17]. Here, absorbing a node in the sense of Definition 2.3.1.

Rather than employing a diffusion ordering network as in [23] to dynamically determine the sequence in which nodes are absorbed, our application in robotics allows us to fix the node decay ordering σ_k and give it as prior knowledge in the diffusion process. The predetermined node decay order starts from the objects and end-effector and ends at the robot's base link (at time $K = N$) [5]. The graph representation is designed such that the node ordering $\boldsymbol{\sigma} = \{N, N-1, \ldots, 1\}$ is set as this path between end-effector and baselink[6]. Throughout the whole diffusion process, the observation graph $\boldsymbol{G}_{O_t}$ is left unchanged, including its node features, coordinates, and edge features.

To avoid the domination of the message-passing results by the masked values [23], we also only keep the masked node to be denoised next along the diffusion trajectory. That is achieved by removing the absorbed nodes from the graph before absorbing new ones, resulting in the chains illustrated in Figure 3.6.

[5] Another fundamental difference is that the result of the Denoising Network here is directly used as node features for the current node, instead of probabilities in a multinomial distribution, to achieve a categorical value.

[6] As the actions are only generated for the robot joint nodes $v_i \in \mathcal{J}$, the order of the objects is irrelevant in the absorption process.

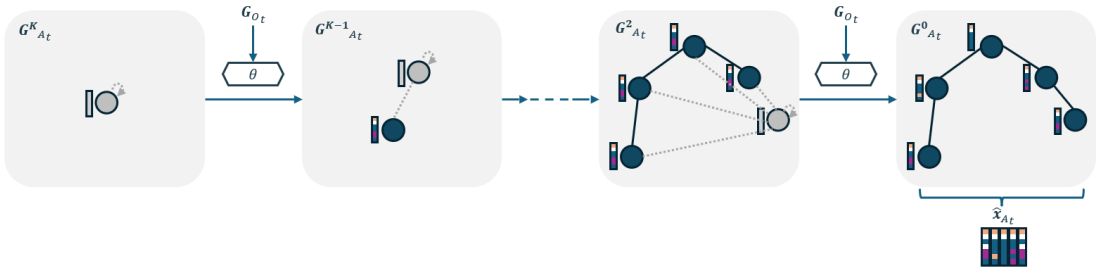


Figure 3.7.: Reverse Chain for the Autoregressive Graph Generative Policy. Highlighted in the end is the action node feature matrix from the action graph $\boldsymbol{G}^0_{A_t}$.

Reverse process

In the reverse chain, the denoising network $\phi_\theta(\boldsymbol{G}^k_{A_t}, \boldsymbol{G}_{O_t})$ is trained to recursively reconstruct the original data structures in the reverse diffusion order. In practice, that's done by predicting the node feature values for the node σ_k, at each time $k = K, K-1, \ldots, 1$ using

$$\boldsymbol{x}^k_t = \phi_\theta(\boldsymbol{G}^k_{A_t}, \boldsymbol{G}_{O_t}), \tag{3.14}$$

where $\boldsymbol{G}^k_{A_t}$ is the current action graph illustrated in Figure 3.7. The prediction for the next node is based on the current action graph and conditioned on the fixed observations. That's done for all nodes representing robot joints $v_i \in \mathcal{J}$, until the action graph is complete.

Denoising Network

The denoising network $\phi_\theta(\boldsymbol{G}^k_{A_t}, \boldsymbol{G}_{O_t})$ is based on the GraphARM denoising network and starts with a node feature linear encoding layer and a corresponding edge feature linear encoding layer resulting in the node and edge embeddings $\mathbf{h}_v$, $\mathbf{h}_e$.

After that, the node embeddings are updated in a loop with FiLM conditioning and message passing for L message passing layers as described in Algorithm 2. The FiLM conditioning tensor is generated by the same graph encoder described in Subsection 3.2.1 and then used to condition the results of the message-passing network.

Here, however, the information isn't propagated with "vanilla" graph attention layers as in [23], but with the use of EGCL layers to account for our geometric data. That results in an iterative message passing as exposed in Equations 2.8-2.11

$$\mathbf{h}_v^{l+1}, \boldsymbol{p}^{l+1} = \text{EGCL}(\mathbf{h}_v^l, \boldsymbol{p}^l, \mathbf{h}_e) \tag{3.15}$$

After the message passing layers, a final node prediction layer outputs the desired features taking as input the final node embeddings $\mathbf{h}_{v_{\sigma_t}}^L$ for each node, and a graph-level embedding $\mathbf{h}_G^L$, result of a mean pooling operation over the nodes.

$$\mathbf{h}_G^L = f_{pool}^{mean}(\boldsymbol{G}) = \frac{1}{N} \sum_{\boldsymbol{x}_i \in \mathbf{x}} \boldsymbol{x}_i \tag{3.16}$$

$$\hat{\boldsymbol{x}}_t^k = \text{MLP}_2([\mathbf{h}_G^L | \mathbf{h}_{v_{\sigma_t}}^L]). \tag{3.17}$$

As there is no diffusion ordering network, the node-level predictions are directly used as actions for the next node in the predefined node order and supervised against the demonstrated actions in the dataset.

Training and Optimization

For the model training, AdamW was used as an optimizer as in the GraphDDPM approach. The final objective of the autoregressive model is to minimize the mean squared error loss for the action node features during the prediction horizon.

$$\mathcal{L}_{ARGG}(\theta) = \mathbb{E}_{\boldsymbol{x}_0, \boldsymbol{G}_{o_t}} ||\hat{\boldsymbol{x}}_t - \boldsymbol{x}_t||^2 \tag{3.18}$$

Sampling procedure

The generation procedure for this model starts with a single, masked node as the initial graph. At timestep k, the denoising network outputs the prediction for node k, according to the node ordering $\sigma_k = k$. The masked node then receives the predicted action node features $\hat{\boldsymbol{x}}_t^k$, and gets connected to the previous nodes with the correct edge types (from the observation graph) Finally, a new masked node is added to the graph, connected to all previously denoised nodes with masked edges.

This procedure is repeated until all robot joints are unmasked, as illustrated in Figure 3.6.

Mini-Batch Optimization

In the ARGG approach, mini-batching in the graph domain is done as described in the GraphDDPM, following Equations 3.11 and 3.13. Here, in contrast, each batch is built using each single graph sample from the dataset, gathering the data for all steps $k \in \{1, 2, \dots, N\}$ in the dimension of the diffusion trajectory. As the observation node features remain unchanged during the diffusion trajectory, we opt to first calculate the FiLM conditioning tensor and then perform predicting all steps of the previously generated absorbing diffusion trajectory within a single model call. The result is a fixed-size mini-batch, whose number of nodes is the sum of the number of nodes along the diffusion trajectory, i.e. $N + (N-1) + \cdots + 1 = N^2$.

Although the batching mechanism in this approach could be expanded to accommodate multiple graph samples, this aspect was not explored in this project. The decision was influenced by the size of our graphs, with N^2 ranging between 100 and 441 (see Table 3.1), deemed suitable for mini-batch processing.

Expected Properties

As in the previous approach, the autoregressive graph generative policy (ARGG) is expected to benefit from the equivariant graph models in terms of data efficiency. However, due to the fixed node decay ordering and the direct utilization of predicted node features as actions, this model tends to be deterministic rather than stochastic. Consequently, it is not anticipated to exhibit multimodal behavior, despite its training in an absorbing diffusion setting.

At the end of Chapter 5, we delve into exploring strategies to address multimodal distributions and generate stochastic actions with ARGG.

4. Experiments and Results

In this chapter, we delve into the experiments conducted to validate the explored approaches of diffusion-based graph generation in learning to imitate robot behavior from demonstrations. We begin by detailing the data, tools, and architecture used in our project, followed by the experimental setup comprising the environments and tasks selected for evaluation. The training and evaluation pipeline are then outlined, including the process of hyper-parameter tuning and the metrics employed to assess the quality of generated trajectories and task success. Finally, we present our findings and engage in a discussion regarding the observed results and their implications.

4.1. Data, Tools, and Architecture

The datasets and environments used in this project are based on the demonstrations collected in the Robomimic project [33].

To run the experiments, all the policies, environments, and environment runners are parametrized with *Hydra* [58]. This allows for centralized parametrization and facilitates the training and evaluation process with multi-runs. For instance, it is possible to switch between policies and tasks (as well as their respective environments).

All the information during training and evaluation is uploaded to a (policy-respective) workspace using *Weights and Biases* [6] for centralized logging and reporting.

The code architecture, initially inspired by the one in [10], consists of the global training, evaluation, and test scripts, as well as the description of all agents, configuration files (hydra parameters), datasets, environments, environment runners, models, and policies.

The models and dataset objects are based on *Pytorch* [38], *Pytorch Geometric* [13], and *Diffusers* [42], also drawing inspiration from the architecture and setup from [10].

Essentially, our codebase allows for easy training and evaluation of new models on the environments (and tasks) presented in Subsection 4.2.1, and will be made publically available under `https://github.com/caio-freitas/GraphDiffusionImitate`.

4.2. Experimental Setup

The experiments are centered around the Robosuite environments [65], using the Franka Panda [15] robot, to match with the demonstration in the Robomimic datasets [33]. This manipulator has 7 degrees of freedom (DOFs) and an attached gripper with another 2, totaling 9 DOFs as the action space.

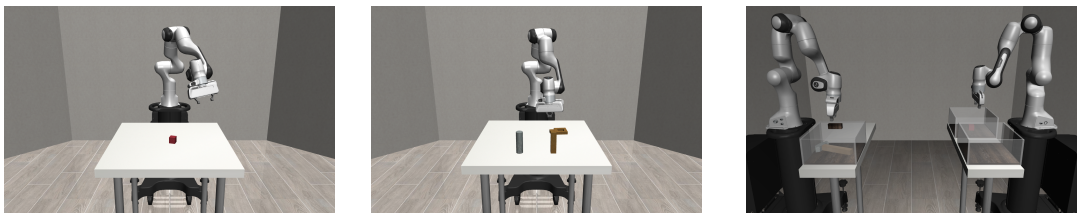
In the creation of the dataset, the action commands used by [33] as input to the manipulators are 7 DOF actions for the operational space controller following the formulation in [20]. These refer only to the position and orientation of the end-effector and the gripper state.

In this work, despite brief experimentation with the OSC control inputs for validation of the environments, we focus on joint positions and velocities, for being suitable to the graph representation of the robot embodiment. For reasons of compatibility with the Robosuite environments, we choose to use the joint velocity controller and train our models using the available ground-truth joint velocities from the demonstrations as actions.

4.2.1. Tasks

To validate the proposed policies and compare those against the related work, we picked the environments and tasks described in detail below. The chosen environments present an increasingly complex set of tasks, from a simple, single-step task that requires relatively low precision to a multi-step task that requires high-precision actions. This aims to provide insights into the performance of the developed approaches and their limitations.

All tasks are present in the dataset created by [33], and are explored for two different modalities: proficient human (ph) and multi-human (mh). The first contains 200 demonstrations from a single, experienced demonstrator and presents therefore smaller variability. The multi-human datasets contain 300 demonstrations each, collected equally by 6 different demonstrators. In this setting, there is higher variation as different strategies are used by each demonstrator to complete the task in the demonstrations.



(a) Lift Environment (b) Square Environment (c) Transport Environment

Figure 4.1.: Robosuite environments used to evaluate and test all policies in the current work.

Lift

The lift environment, as described in [65] consists of a robotic arm, a table, and a small cube. This task is completed successfully when the cube is grabbed by the robot's gripper and lifted from the table. The initial location of the cube and the initial robot joints are randomized at each episode.

While seemingly simple, this task serves as a valuable tool for validating our policies. Despite not requiring a high level of precision due to the versatility of lifting techniques, the cube's initial rotation notably influences the demonstrations in the dataset, particularly in the proficient human modality. From another perspective, the variety of strategies possible in this task does leave room for observing multi-modal behavior.

Of particular interest is the correct adjustment of gripper position and orientation based on object observation. This observation validates the efficacy of model conditioning in guiding the diffusion process, particularly in the developed policies.

Square

In the square environment, the setup includes a robot, a table, two colored pegs (one square and one round), and two colored nuts (again, one square and one round). Success in this task entails the robot's gripper grabbing the square nut and placing it around the square peg. The initial position of the square nut varies with each episode. Here, the trajectories exhibit pronounced multi-modality, notably in the gripper's approach to grasping the nut. This task requires high-precision actions for its completion, once the square nut has to be aligned with the peg to be correctly inserted.

Transport

This multi-robot environment from [65] features two robotic manipulators positioned between two tables with objects on them. The goal here is for one manipulator to grab a hammer from one of the tables and hand it over to be picked up by the other manipulator and placed in a container. The objective is for one manipulator to grab a hammer from one table and transfer it to be picked up by the other manipulator, which then places it in a designated container. Meanwhile, the second manipulator must clear the target container by relocating a ”piece of trash” (a red cube) to the ”trash bin” (a second container). The position of all bins, the lid, the trash cube, and the hammer are randomized at the start of each episode.

In this setup, commands for both manipulators originate from the same policy and model, effectively acting as a single, double-armed actor. In the graph representation described in Subsection 3.1.3, the manipulator is represented by linking the base links of both robot arms by a virtual (fixed) robot link.

4.2.2. Training and Evaluation Pipeline

Before training, all datasets are randomly split into train and validation sets, with 10% validation samples. The from here on called *lowdim* datasets have minimal preprocessing, and are used to train the (no-graph) benchmark policies. It goes through the episodes accumulating the observations and actions for their respective horizons and concatenates them.

The samples in this dataset, in comparison to the graph representation, consist only of vectorized elements. More precisely, an observation vector containing concatenated individual joint angles, end effector(s) position and orientation, and the positions and orientations from the objects. The observations are stacked over the H_{obs} horizon. [1] The action vector consists of individual joint velocities, stacked over the prediction horizon H_{pred}.

For the no-graph *lowdim* policies, the dimension of the observation (variable from task to task) shapes the input of the used models. Similarly, the dimension of the actions (depending on the number of joints and number of robots) shapes the output of these models.

[1] The *lowdim* datasets do not contain the individual task-space coordinates of all robot joints.

For comparison, using the graph representation from Subsection 3.1.3 together with either one of the proposed graph policies, the network architecture only depends on the node feature dimension. Thus being independent of the number of objects in the scene or the number of robot joints, i.e. a single model could be trained on data from multiple environments with a variable number of objects.

After every training epoch, an average score in the validation dataset is calculated and logged. 50 evaluation rollouts in the environment are executed every 50 epochs of training, following what's done in [33, 10]. The final chosen policy is the checkpoint with the greatest success rate in the rollouts throughout training.

4.2.3. Evaluation Metrics

In order to evaluate the performance of the policies and the quality of the generated trajectories by all policies, we introduce the following metrics:

- **Success Rate (S):** This final objective returns how many of the rollout episodes terminated successfully, according to the task description in Subsection 4.2.1.
- **Average Inference Time (T):** Average inference time from the model during all model calls. This provides us with a heuristic measure of computational efficiency.
- **Trajectory Similarity (MSE):** This metric measures the resemblance of generated trajectories to those in the original dataset, given the observation. It is computed as the Mean Squared Error over the joint values throughout the trajectory for each joint over time.
- **Standard Deviation of Error (σ_e):** This metric measures the dispersion or variability of errors between the generated trajectories and the ground truth data. It provides insights into the consistency of the model's performance across different instances.
- **Waypoint Variance (WP VAR):** Waypoint variance quantifies the multimodality of the generated trajectories and is calculated as the sum (along the trajectory dimension) of the pairwise L2-distance variance between waypoints at corresponding timesteps [7].
- **Trajectory Smoothness:** This metric measures the continuity and absence of sudden changes in the generated trajectories. It measures how smoothly the generated motions transition between waypoints, indicating the overall fluidity of the trajectory. The smoothness is computed by evaluating the total variation of velocity along the

trajectory. A **smaller** value indicates a smoother trajectory, with a constant trajectory having a smoothness value of 0.

4.3. Experiments

To validate the proposed approaches, our research objective can be stated as providing answers to the following questions:

1. Can a diffusion-based graph generative model learn to imitate robot behavior from demonstrations?
2. Can the proposed models succeed in performing the tasks in the proposed environments?
3. Are the proposed models able to generate multi-modal trajectories for the robot?
4. Do diffusion models benefit from the graph-structured representation and models?
5. Does a GNN-based policy benefit from the introduction of a diffusion-based framework?

In order to address these, we design and perform the series of experiments further introduced in this chapter.

4.3.1. Hyper-parameter Tuning

Before submitting the developed policies to extensive training and evaluation, we first try to discover which hyper-parameters return the best results for each. For that, we conduct a parameter sweep across a set of multiple parameters for both policies on the Lift (ph) task. The hyper-parameters considered for both models include the learning rate and number of message-passing layers in the respective denoising networks. In addition, the hidden dimension in the denoising networks (the same used in the graph encoder) is also considered. We employ a grid search to explore various combinations of these quantities.

For the learning rate, three values were tested: 0.0001, 0.00005, and 0.00001. The hidden dimension was varied among 128, 256, 512, and 1024. Similarly, the number of message-passing layers was experimented with 2, 3, 4, and 5 layers.

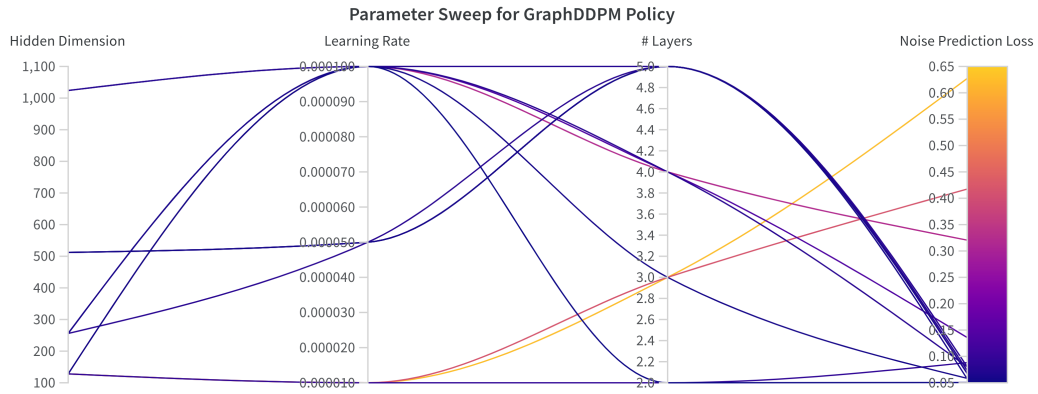


Figure 4.2.: Results of parameter sweep for the GraphDDPM model in the Lift (ph) dataset. The diagram illustrates the results of noise prediction loss after the 200 epochs for each combination of the hyperparameters.

In total, 15 distinct models (for each Policy) were trained on the ”Lift (ph)” dataset, each model being trained for 200 epochs. Random sampling was utilized to select combinations of hyperparameters for training each model.

To identify the optimal configuration, the performance of each model was evaluated based on each model's validation score: the noise prediction error for the GraphDDPM, and the mean squared error of predicted actions for the ARGG. Figures 4.2 and 4.3 present the parameter sweep results. The model with the smallest validation score, indicating superior generalization ability, was chosen as the final configuration for the Graph Diffusion model. This approach ensured that the selected hyperparameters were fine-tuned to achieve the best possible performance on the given dataset within a reasonable computational effort.

4.3.2. Sampled Trajectories with GraphDDPM

This experiment aims to investigate the action distribution output by the model for a single observation and compare it to the ground-truth action in the dataset. It's important to highlight that, even though the action to that specific observation is used as a reference, this single action does not necessarily define the distribution of that mode, but is rather an instance of it. The expected result here after training is that the ground-truth action is contained within the distribution of actions output by the model.

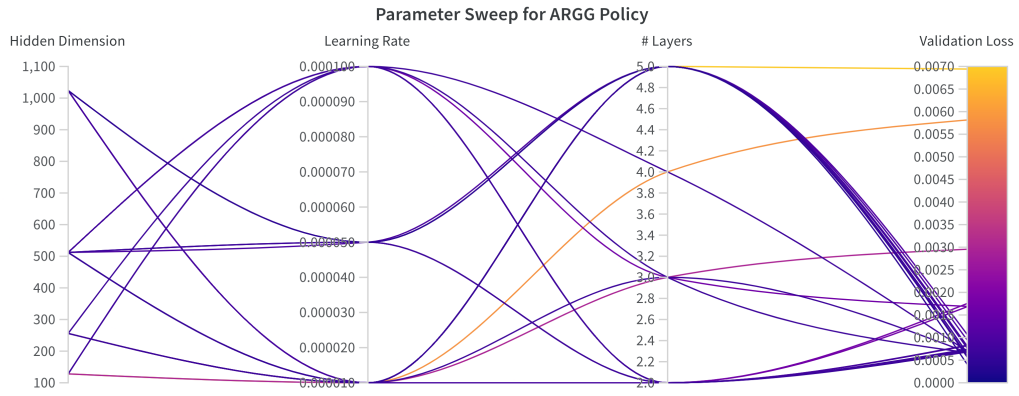


Figure 4.3.: Results of parameter sweep for the ARGGen model in the Lift (ph) dataset. The diagram illustrates the results of noise prediction loss after the 200 epochs for each combination of the hyperparameters.

We sample multiple actions from the GraphDDPM policy using the same observation as the conditioning tensor, also to demonstrate its stochasticity. For this experiment, the checkpoint for the best-performing GraphDDPM policy in the Lift (ph) task is used. 50 random actions are then sampled from Gaussian noise, and iteratively denoised by the pre-trained GraphDDPM model for $K = 100$ diffusion steps following (2.18).

The results presented in Figure 4.4 do support our expectations, with the demonstrated individual robot joints appearing to be contained within the action distribution output by the model.

4.3.3. Multimodality Validation

To validate the GraphDDPM model's capabilities on generating according to a multi-modal action distribution, we experimented with initializing an untrained model instance and optimizing it for the denoising of 2 action modes given fixed observations. Thus explicitly feeding the model a bi-modal action distribution for a single observation, which is hard to extract naturally in the dataset.

In Figure 4.5 are the results output by the model after 5000 optimization steps on the given samples, along with the original actions taken from the dataset.

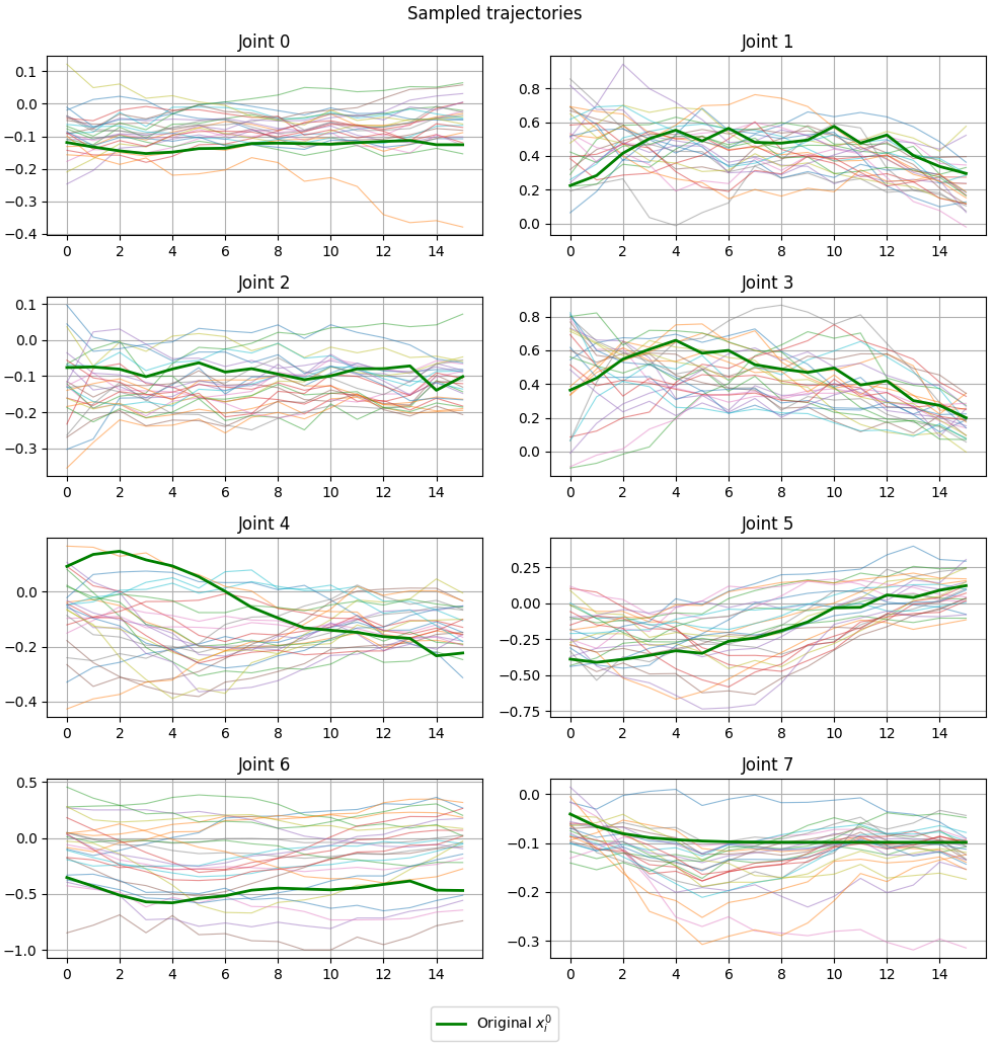


Figure 4.4.: Trajectory sampling using 50 different seeds for a single observation. Baseline action regarding the observation in the dataset presented for reference.

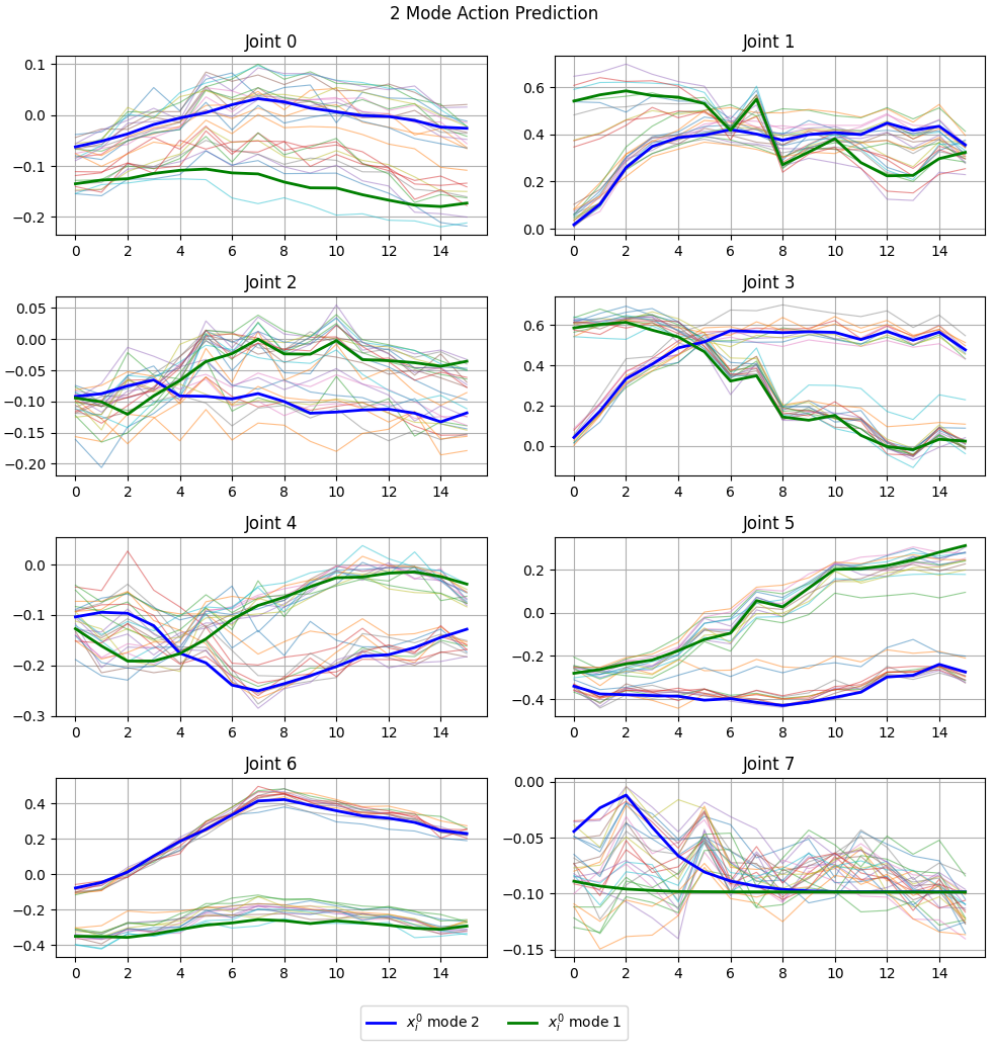


Figure 4.5.: Trajectory sampling using 50 different seeds for observation with 2 modes. The model was trained from scratch to validate the multi-modal action prediction capabilities of the GraphDDPM model.

We observe that the action distribution generated by the GraphDDPM policy attempts to accommodate both actions fed for that observation. A promising result toward validating the model's ability to capture multi-modal behavior.

4.3.4. Conditioning GraphDDPM on last actions

Instead of starting the GraphDDPM reverse chain by sampling the node features directly from a Gaussian distribution as $\boldsymbol{x}_t^K \sim \mathcal{N}(0, \mathbf{I})$, which requires full denoising at every sampled action, we experiment maintaining some of the structures of the data from the last executed action to better guide the denoising process.

More specifically, we maintain the first action step (in the prediction horizon) constant as the ground truth in that time t, while the rest of the action dimensions are sampled and denoised as usual.

To implement this idea in execution time, the last step of the last predicted action is used as first step of the predicted action and kept unchanged during the denoising diffusion. To correct for this, the environment is then updated not to execute the first predicted action, restraining $H_{act} \leq H_{pred} - 1$.

The observed effect is that all predicted trajectories are then incentivized to be continuous with respect to that first action, increasing trajectory smoothness as can be noticed in Figure 4.6 and Table 4.2. This approach, already aligned with velocity control principles, is anticipated to be even more effective when predicting joint positions, given the heightened importance of continuity, considering the manipulator's dynamics.

Table 4.1.: Trajectory quality metrics for comparison between the two approaches for GraphDDPM regarding the diffusion trajectory: one retaining the first action throughout diffusion chains, and the other without.

	Lift (ph)				
	S[%]	WP VAR	MSE	σ_e	Smoothness
GraphDDPM	**10.0**	$\mathbf{3.09 \times 10^{-2}}$	3.78×10^{-2}	1.27×10^{-1}	1.055
GraphDDPM (cond)	4.0	3.72×10^{-3}	$\mathbf{1.42 \times 10^{-2}}$	$\mathbf{5.90 \times 10^{-2}}$	**0.739**

In terms of success rate, though, we observed no improvement compared to the more basic approach of starting from Gaussian noise, presenting around 4% success rate after training for 800 epochs compared to the 10% obtained when training the policy from pure noise for the same period.

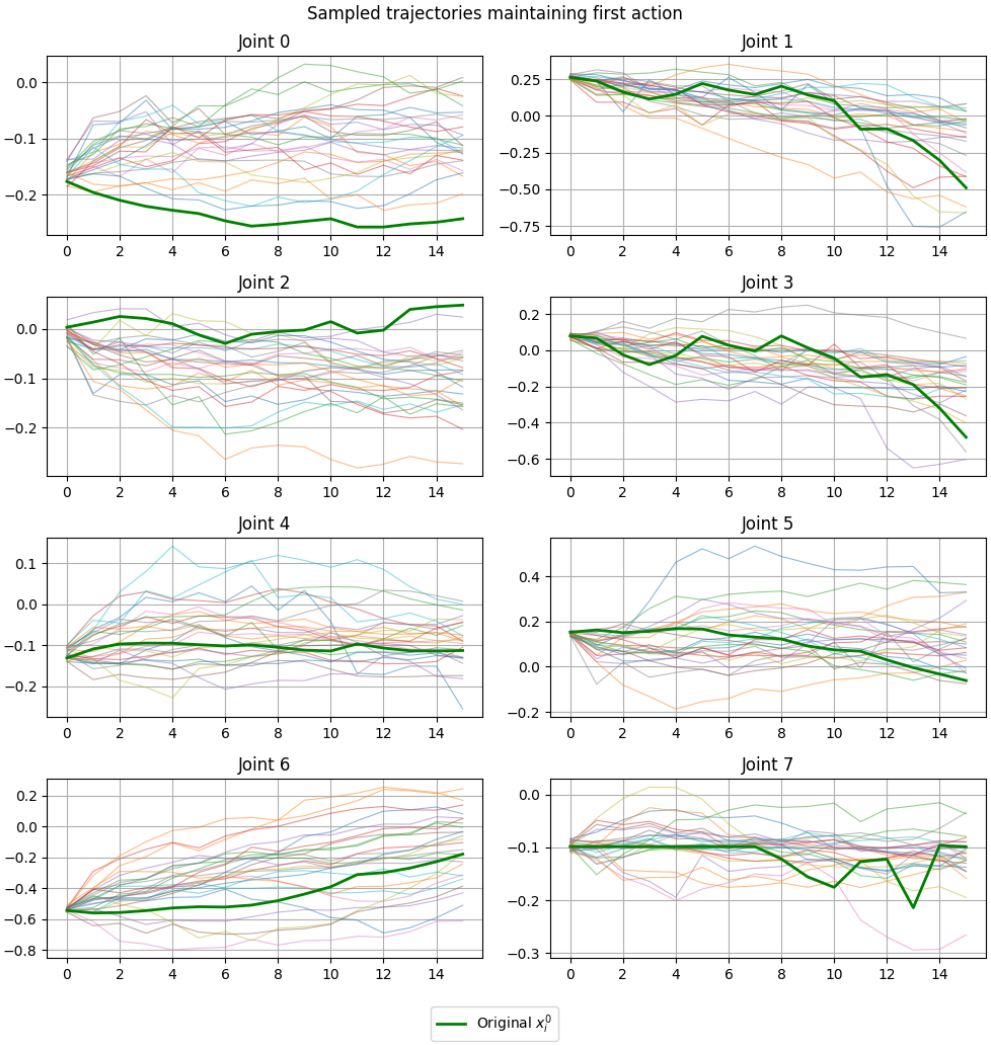


Figure 4.6.: Trajectory sampling using 50 different seeds while keeping the first action fixed during the diffusion process of the GraphDDPM model.

4.3.5. Influence of the number of diffusion steps

Also specific to the GraphDDPM policy is the number of diffusion steps K used by the model. This impacts directly the action sampling time, for it determines the number of model calls necessary to completely denoise an action.

We use $K = 100$ in all other experiments so that the generation quality can be compared with the Diffusion Policy [10]. Here, we train the GraphDDPM policy for 800 epochs using $K = 50, 200$ as well and present the obtained success rate and trajectory quality metrics for these cases.

Table 4.2.: Trajectory quality metrics for comparison among different numbers of diffusion steps K for the GraphDDPM Policy.

	Lift (ph)				
	S [%]	WP VAR	MSE	σ_e	Smoothness
GraphDDPM (K=50)	6.0	1.35×10^{-1}	5.78×10^{-2}	1.65×10^{-1}	1.71
GraphDDPM (K=100)	**10.0**	3.09×10^{-2}	3.78×10^{-2}	1.27×10^{-1}	**1.05**
GraphDDPM (K=200)	**10.0**	$\mathbf{5.31 \times 10^{-2}}$	$\mathbf{2.64 \times 10^{-2}}$	$\mathbf{8.78 \times 10^{-2}}$	1.57

The results indicate that increasing K improves the similarity of the generated trajectories to the demonstrations.

4.4. Findings

Following, we present the results obtained by running training and evaluation on all policies and tasks. All policies with the same data modality are trained for the same number of epochs for each task, capped by the number of epochs of the slowest policy in a given computation time (24 hours in the cluster with 2 CPUs and an RTX 2030 Ti).

For benchmarking purposes, we also implement the BC-EGNN policy consisting of a behavioral cloning policy crafted from the denoising network of the GraphDDPM policy, i.e. an equivariant graph neural network with 3 layers of message-passing, and hidden dimension 256. As the most primitive graph-structured policy tested, it aims to compare the effect of removing the diffusion pipeline from our policies, returning the predicted actions with a single model call.

A table describing the details of the training for each data type and task is provided in Appendix A.4.

4.4.1. Quality of Generated Trajectories

The quality of the generated trajectories must be evaluated, and that's done for each of the best-performing model parameters of every one of the policies in the analysis, concerning the performed success rate. [2]

For deterministic policies, the trajectory quality is calculated based on predicted and baseline action trajectories from 50 observation samples from the dataset, while for stochastic policies, 50 samples from the dataset are used, employing 10 different seeds to calculate the waypoint variance.

Table 4.3.: Trajectory quality metrics for the best-performing checkpoints of each policy in the Lift task

		Lift (ph)				
		T [s]	WP VAR	MSE	σ_e	Smoothness
Lowdim	MLP Policy	$\mathbf{3.31 \times 10^{-4}}$	-	8.32×10^{-4}	$\mathbf{7.42 \times 10^{-3}}$	$\mathbf{5.26 \times 10^{-1}}$
	BC-RNN	1.59×10^{-3}	-	6.07×10^{-3}	4.48×10^{-2}	6.51×10^{-1}
	DiffusionPolicy	4.30	$\mathbf{7.19 \times 10^{-4}}$	$\mathbf{3.70 \times 10^{-4}}$	8.80×10^{-3}	1.64
Graph	BC-EGNN	$\mathbf{3.56 \times 10^{-3}}$	-	5.07×10^{-2}	$\mathbf{7.79 \times 10^{-2}}$	2.11
	GraphDDPM	4.15×10^{-1}	$\mathbf{3.09 \times 10^{-2}}$	3.78×10^{-2}	1.27×10^{-1}	1.05
	ARGG	4.11×10^{-2}	-	$\mathbf{9.37 \times 10^{-4}}$	1.05×10^{-1}	$\mathbf{9.10 \times 10^{-1}}$

Table 4.4.: Trajectory quality metrics for the best-performing checkpoints of each policy in the Square task

		Square (ph)				
		T [s]	WP VAR	MSE	σ_e	Smoothness
Lowdim	MLP Policy	$\mathbf{7.66 \times 10^{-4}}$	-	$\mathbf{3.99 \times 10^{-3}}$	$\mathbf{2.94 \times 10^{-2}}$	1.61
	BC-RNN	1.61×10^{-3}	-	4.06×10^{-3}	3.56×10^{-2}	1.56
	DiffusionPolicy	4.45	$\mathbf{1.45 \times 10^{-2}}$	1.11×10^{-2}	8.99×10^{-2}	**1.11**
Graph	BC-EGNN	$\mathbf{3.70 \times 10^{-3}}$	-	3.08×10^{-2}	8.28×10^{-2}	**1.36**
	GraphDDPM	3.80×10^{-1}	$\mathbf{4.38 \times 10^{-1}}$	1.27×10^{-1}	2.51×10^{-1}	2.58
	ARGG	3.98×10^{-2}	-	$\mathbf{2.55 \times 10^{-3}}$	$\mathbf{2.41 \times 10^{-2}}$	2.00

[2] In case of ambiguity in the success rate, the validation loss is used to determine the best-performing model

Table 4.5.: Trajectory quality metrics for the best-performing checkpoints of each policy in the Transport task

		Transport (ph)				
		T [s]	WP VAR	MSE	σ_e	Smoothness
Lowdim	MLP Policy	$\mathbf{3.31 \times 10^{-4}}$	-	$\mathbf{2.57 \times 10^{-3}}$	$\mathbf{2.11 \times 10^{-2}}$	4.40×10^{-1}
	BC-RNN	1.72×10^{-3}	-	3.58×10^{-3}	3.29×10^{-2}	$\mathbf{1.27 \times 10^{-1}}$
	DiffusionPolicy	4.34	$\mathbf{2.57 \times 10^{-3}}$	9.76×10^{-2}	4.28×10^{-2}	3.68
Graph	BC-EGNN	$\mathbf{5.02 \times 10^{-2}}$	-	1.39×10^{-2}	2.32×10^{-2}	**1.56**
	GraphDDPM	4.79×10^{-1}	$\mathbf{7.11 \times 10^{-1}}$	1.12×10^{-1}	1.72×10^{-1}	3.57
	ARGG [3]	2.51×10^{-1}	-	$\mathbf{7.32 \times 10^{-3}}$	$\mathbf{2.15 \times 10^{-2}}$	4.12

4.4.2. Task Success

In the following, we present the success rates resulting from the training and evaluation procedure for all policies and tasks.

As described by [33], selecting the best-performing policy in offline policy learning is a challenge. Contrary to most supervised learning problems, selecting a policy using the best loss on validation data for IL applications tends to select policies that perform significantly worse in terms of success rate. With that in mind, we constantly evaluate the policies in environment rollouts during the training process and select the best-performing policy as being the one producing the highest success rate. These are exposed in Table 4.6.

Table 4.6.: Success Rates of Baseline Methods Compared to Graph Diffusion

		Lift		Square		Transport	
	Success Rate (%)	ph	mh	ph	mh	ph	mh
Lowdim	MLP	38.0	32.0	0.0	0.0	0.0	0.0
	BC RNN	**88.0**	52.0	**2.0**	0.0	0.0	0.0
	Diffusion Policy	60.0	**64.0**	0.0	0.0	0.0	0.0
Graph	BC-EGNN	16.0	0.0	0.0	0.0	0.0	0.0
	GraphDDPM	10.0	**14.0**	0.0	0.0	0.0	0.0
	ARGG	**56.0**	4.0	0.0	0.0	0.0 [3]	0.0 [3]

[3] The ARGG approach wasn't able to complete the number of training epochs specified in Table A.1 for the Transport task in the specified time. The results presented in this table are preliminary after the training logs presented in the Appendix A.3

4.5. Discussion

In this section, we analyze and discuss the performance of the explored policies concerning their data paradigm, i.e. graph-structured or vector, and their prediction type, i.e. one-shot prediction, autoregressive generation or denoising diffusion.

Our first finding when evaluating the BC-EGNN policy with our graph representation was that, against our expectations, this setup was outperformed by the simpler non-graphical approaches such as MLP. The additional connection information, together with the individual node coordinates in the chosen Graph representation has shown to be more challenging to learn from than regular vector data. Consequently, the use of graph representation in robotics imitation learning reveals a tradeoff between the versatility of employing a flexible graph-based policy, irrespective of the number of objects and joints in the scene, and the facilitated learning, achieving high performance in a fixed-structure task and model. In the pursuit of these graphical models, our results point towards using either a different graph representation or an alternate graph-based model as a building block.

Regarding the graph diffusion policies, GraphDDPM, just like the Diffusion Policy in the vector domain, was the most effective policy when trained on the Lift multi-human Graph dataset. That highlights its capabilities of generating multi-modal actions, as corroborated by the outcomes of the reduced setting experiment from Subsection 4.3.3. Moreover, maintaining the initial action throughout the reverse diffusion chain also had the expected effects of shaping the final action distribution to be more continuous regarding the last actions, showing to be an effective technique to be explored in future work (especially interesting if using position-based control). However, the sampled trajectories from the model were not able to outperform any of the non-graph policies.

It is worth noting that the GraphDDPM policy is notably lighter than the Diffusion Policy [10], requiring approximately one-tenth of the computation time to generate a new sample with the same number of diffusion steps (c.f. Tables 4.3 to 4.5).

In the case of the ARGG policy, although not stochastic, it exhibited superior performance in the Graph Lift (ph) dataset compared to all other graph policies, while still being surpassed by vector policies such as BC-RNN and the Diffusion Policy.

The autoregressive generative approach seemed to be, in general, an improvement to the plain BC-EGNN policy, indicating a discernible enhancement in success rate in the Lift task, as well as improvements in the trajectory quality metrics. This underscores the

autoregressive generation framework as a promising research avenue to explore further. However, one limitation encountered in the current study pertained to the mini-batch procedure described in Subsection 3.2.3, which proved to be insufficient for training the policy in larger datasets, which severely limited the number of training epochs in the Transport task.

Overall, the results obtained for both GraphDDPM and ARGD policy in the Lift task validate the efficacy of conditioning the outputs of the EGNN models with FiLM layers between message-passing updates. Despite the constrained performance, particularly evident in the Lift task, the manipulator aptly responded to the position and orientation of the cube.

Finally, neither the graph-structured nor the vector-structured policies achieved a reasonable success rate in the Square and Transport tasks, highlighting the challenges associated with executing high-precision tasks using joint-level control in imitation learning. Our analysis of these tasks is primarily confined to trajectory quality metrics and observation of the rollouts. In the graph domain, the ARGG policy demonstrated the best performance in predicting actions similar to the demonstrations (despite the ARGG's limited training in the Transport task). Though without significant success rates, just as all other policies.

In the context of the research questions outlined in Section 4.3, we revisit them with our insights from our prior discussions:

1. Can a diffusion-based graph generative model learn to imitate robot behavior from demonstrations? The actions generated by the graph policies do present comparable (although still inferior) performance to the existing methods. This indicates that the models are indeed producing trajectories close to the demonstrated, however suboptimally.

2. Can the proposed models succeed in performing the tasks in the proposed environments? The proposed models could succeed only in the Lift task, though still struggle to return high success rates with the graph representation and policies. This underscores the necessity for further scrutiny and experimentation.

3. Are the proposed models able to generate multi-modal trajectories for the robot? The GraphDDPM model is shown to be adaptable in modeling multi-modal distributions.

4. Do diffusion models benefit from the graph-structured representation and models? Our results indicate a lack of benefits in the imitation precision or performance when using the graph representation in the current setting over a diffusion policy, such as [10]. Being so, the benefits of the graph representation are seen as the gain in flexibility in the robot and environment representation.

5. Does a GNN-based policy benefit from the introduction of a diffusion-based framework? According to our experiments, we observe that a GNN-based (BC-EGNN) policy does benefit from the diffusion-based setup.

5. Conclusion

In this thesis, we studied how imitation learning in robotics can be tackled as a graph generation problem, employing diffusion-based techniques. We recognized the potential of diffusion models and graph neural networks in capturing intricate data distributions and processing non-Euclidean data, respectively, and sought to leverage their properties for advancing imitation learning techniques.

Our investigation began by acknowledging the potential of diffusion-based generative models in robot learning, highlighting their capacity to learn intricate data distributions. Additionally, we recognized the theoretical advantages that graph neural networks can offer to imitation learning in robotics, particularly their ability to process non-Euclidean data and their flexibility in handling ”graphs with a variable number of nodes and edges”. Drawing inspiration from recent literature on graph generation using diffusion-based techniques, we proposed policies aimed at leveraging the advantages of both graph-structured models and the diffusion framework.

We identify the challenge in learning from the graph-structured dataset using graph models, when compared to a fixed vector representation as most established in the literature. From then, we present the current graph representation as a trade-off between leveraging graph representations for their flexibility coupled with structural information and achieving optimal performance in imitation learning tasks.

Given the difficulties in learning from the graph data with a simpler behavioral cloning policy (BC-EGNN), we then approach the graph generation problem from two perspectives: the autoregressive and one-shot denoising diffusion, proposing one policy for each setup.

We introduced the GraphDDPM policy, based on denoising diffusion, showcasing its ability to capture multi-modal action distributions. We also present our experiments with techniques to attempt to improve generation quality, as well as the performance and limitations of this policy in the proposed environments.

Further on, we tackle the graph generation problem from the autoregressive generation perspective with the ARGG policy, presenting its destructive and generative chains, and model. We then presented the results of this technique in our proposed environments.

Despite encountering limitations such as computational constraints and difficulties in executing high-precision tasks using joint-level control, our experiments provided valuable insights into the strengths and limitations of diffusion-based graph generative models in imitation learning.

In conclusion, while our study marks a step towards expanding the applications of diffusion-based graph generative models in imitation learning, it also highlights paths for further exploration and refinement.

Future research

In considering future research directions, several avenues emerge as promising extensions to the present work.

One such direction involves the extension of the experiments conducted in Subsection 4.3.5, where the impact of varying the number of diffusion steps on sample quality was explored. Given the observed efficiency of the GraphDDPM model relative to the Diffusion Policy, the number of diffusion steps could still be significantly increased aiming to improve the quality of the generated samples while maintaining a comparable prediction time. Another compelling prospect for future research involves adapting the ARGG policy to accommodate stochastic generation, focusing on modeling distributions rather than directly modeling actions.

Additionally, incorporating observation history into the policies using recurrent neural networks (RNNs) or equivalent graph-based models presents an opportunity for performance enhancement. This strategy, highlighted by [33], is known to play a crucial role in obtaining good performance in imitation learning tasks, especially those with longer horizons.

Regarding the benefits of the graph representation, exploring the flexibility made possible by the setup in this work is encouraged, such as training policies to perform in a multi-environment setting with a variable number of objects.

Finally, recent work [4] suggests techniques for further enhancing equivariant GNNs, such as deriving geometrically optimal edge attributes within the framework of EGNNs [47]

utilized in this work. This represents another promising research direction for improving the policies presented in this work.

Bibliography

[1] Ayman Ali et al. *Skeleton-based Human Action Recognition via Convolutional Neural Networks (CNN)*. 2023. arXiv: 2301.13360 [cs.CV].

[2] Uri Alon and Eran Yahav. *On the Bottleneck of Graph Neural Networks and its Practical Implications*. 2021. arXiv: 2006.05205 [cs.LG].

[3] Jacob Austin et al. *Structured Denoising Diffusion Models in Discrete State-Spaces*. 2023. arXiv: 2107.03006 [cs.LG].

[4] Erik J Bekkers et al. "Fast, Expressive $\mathrm{SE}(n)$ Equivariant Networks through Weight-Sharing in Position-Orientation Space". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=dPHLbUqGbr.

[5] Filippo Maria Bianchi and Veronica Lachi. *The expressive power of pooling in Graph Neural Networks*. 2023. arXiv: 2304.01575 [cs.LG].

[6] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[7] Joao Carvalho et al. *Motion Planning Diffusion: Learning and Planning of Robot Motions with Diffusion Models*. arXiv:2308.01557 [cs]. Aug. 2023. URL: http://arxiv.org/abs/2308.01557 (visited on 03/25/2024).

[8] Ziyi Chang, George Alex Koulieris, and Hubert P. H. Shum. *On the Design Fundamentals of Diffusion Models: A Survey*. arXiv:2306.04542 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2306.04542 (visited on 01/25/2024).

[9] Hongyang Chen et al. *Diffusion-based graph generative methods*. arXiv:2401.15617 [cs]. Jan. 2024. URL: http://arxiv.org/abs/2401.15617 (visited on 04/04/2024).

[10] Cheng Chi et al. *Diffusion Policy: Visuomotor Policy Learning via Action Diffusion*. arXiv:2303.04137 [cs]. June 2023. URL: http://arxiv.org/abs/2303.04137 (visited on 10/30/2023).

[11] Jeongjun Choi, Dongseok Shim, and H. Jin Kim. *DiffuPose: Monocular 3D Human Pose Estimation via Denoising Diffusion Probabilistic Model*. arXiv:2212.02796 [cs]. Aug. 2023. URL: http://arxiv.org/abs/2212.02796 (visited on 04/04/2024).

[12] Bin Fang et al. "Survey of imitation learning for robotic manipulation". en. In: *International Journal of Intelligent Robotics and Applications* 3.4 (Dec. 2019), pp. 362–369. ISSN: 2366-5971, 2366-598X. DOI: 10.1007/s41315-019-00103-5. URL: http://link.springer.com/10.1007/s41315-019-00103-5 (visited on 05/09/2024).

[13] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[15] Sami Haddadin et al. "The Franka Emika Robot: A Reference Platform for Robotics Research and Education". In: *IEEE Robotics and Automation Magazine* 29.2 (2022), pp. 46–64. DOI: 10.1109/MRA.2021.3138382.

[16] Jonathan Ho, Ajay Jain, and Pieter Abbeel. *Denoising Diffusion Probabilistic Models*. arXiv:2006.11239 [cs, stat]. Dec. 2020. URL: http://arxiv.org/abs/2006.11239 (visited on 03/25/2024).

[17] Emiel Hoogeboom et al. *Autoregressive Diffusion Models*. arXiv:2110.02037 [cs, stat]. Feb. 2022. URL: http://arxiv.org/abs/2110.02037 (visited on 03/05/2024).

[18] Emiel Hoogeboom et al. *Equivariant Diffusion for Molecule Generation in 3D*. arXiv:2203.17003 [cs, q-bio, stat]. June 2022. URL: http://arxiv.org/abs/2203.17003 (visited on 01/28/2024).

[19] Steven Kearnes et al. "Molecular graph convolutions: moving beyond fingerprints". In: *Journal of Computer-Aided Molecular Design* 30.8 (Aug. 2016), pp. 595–608. ISSN: 1573-4951. DOI: 10.1007/s10822-016-9938-8. URL: http://dx.doi.org/10.1007/s10822-016-9938-8.

[20] O. Khatib. "A unified approach for motion and force control of robot manipulators: The operational space formulation". In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53. DOI: 10.1109/JRA.1987.1087068.

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[22] Thomas N Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[23] Lingkai Kong et al. *Autoregressive Diffusion Model for Graph Generation*. arXiv:2307.08849 [cs]. July 2023. URL: http://arxiv.org/abs/2307.08849 (visited on 10/30/2023).

[24] Yunzhu Li et al. *Learning Particle Dynamics for Manipulating Rigid Bodies, Deformable Objects, and Fluids*. 2019. arXiv: 1810.01566 [cs.LG].

[25] Zhixuan Liang et al. *AdaptDiffuser: Diffusion Models as Adaptive Self-evolving Planners*. arXiv:2302.01877 [cs]. May 2023. URL: http://arxiv.org/abs/2302.01877 (visited on 10/30/2023).

[26] Lequan Lin et al. *SpecSTG: A Fast Spectral Diffusion Framework for Probabilistic Spatio-Temporal Traffic Forecasting*. 2024. arXiv: 2401.08119 [cs.LG].

[27] Yixin Lin et al. *Efficient and Interpretable Robot Manipulation with Graph Neural Networks*. arXiv:2102.13177 [cs]. Jan. 2022. URL: http://arxiv.org/abs/2102.13177 (visited on 01/17/2024).

[28] Chengyi Liu et al. *Generative Diffusion Models on Graphs: Methods and Applications*. arXiv:2302.02591 [cs]. Aug. 2023. URL: http://arxiv.org/abs/2302.02591 (visited on 10/30/2023).

[29] YuXuan Liu et al. *Imitation from Observation: Learning to Imitate Behaviors from Raw Video via Context Translation*. arXiv:1707.03374 [cs]. June 2018. URL: http://arxiv.org/abs/1707.03374 (visited on 04/08/2024).

[30] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. arXiv:1711.05101 [cs, math]. Jan. 2019. URL: http://arxiv.org/abs/1711.05101 (visited on 05/04/2024).

[31] Calvin Luo. *Understanding Diffusion Models: A Unified Perspective*. arXiv:2208.11970 [cs]. Aug. 2022. URL: http://arxiv.org/abs/2208.11970 (visited on 05/10/2024).

[32] Xiao Ma, David Hsu, and Wee Sun Lee. *Learning Latent Graph Dynamics for Visual Manipulation of Deformable Objects*. arXiv:2104.12149 [cs]. Mar. 2022. URL: http://arxiv.org/abs/2104.12149 (visited on 04/08/2024).

[33] Ajay Mandlekar et al. *What Matters in Learning from Offline Human Demonstrations for Robot Manipulation*. arXiv:2108.03298 [cs]. Sept. 2021. URL: http://arxiv.org/abs/2108.03298 (visited on 10/30/2023).

[34] D.Q. Mayne and H. Michalska. “Receding horizon control of nonlinear systems”. In: *IEEE Transactions on Automatic Control* 35.7 (1990), pp. 814–824. DOI: 10.1109/9.57020.

[35] Diego Mesquita, Amauri Souza, and Samuel Kaski. “Rethinking pooling in graph neural networks”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 2220–2231. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1764183ef03fc7324eb58c3842bd9a57-Paper.pdf.

[36] Alex Nichol and Prafulla Dhariwal. *Improved Denoising Diffusion Probabilistic Models*. arXiv:2102.09672 [cs, stat]. Feb. 2021. URL: http://arxiv.org/abs/2102.09672 (visited on 05/04/2024).

[37] Takayuki Osa et al. “An algorithmic perspective on imitation learning”. In: *Foundations and Trends® in Robotics* (2018).

[38] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[39] Tim Pearce et al. *Imitating Human Behaviour with Diffusion Models*. arXiv:2301.10677 [cs, stat]. Mar. 2023. URL: http://arxiv.org/abs/2301.10677 (visited on 10/30/2023).

[40] Ethan Perez et al. *FiLM: Visual Reasoning with a General Conditioning Layer*. arXiv:1709.07871 [cs, stat]. Dec. 2017. URL: http://arxiv.org/abs/1709.07871 (visited on 03/05/2024).

[41] Francesca Pistilli and Giuseppe Averta. *Graph learning in robotics: a survey*. arXiv:2310.04294 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2310.04294 (visited on 10/30/2023).

[42] Patrick von Platen et al. *Diffusers: State-of-the-art diffusion models*. https://github.com/huggingface/diffusers. 2022.

[43] Dean A. Pomerleau. “ALVINN: An Autonomous Land Vehicle in a Neural Network”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 1. Morgan-Kaufmann, 1988. URL: https://proceedings.neurips.cc/paper_files/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

[44] Saša V. Raković and William S. Levine, eds. *Handbook of Model Predictive Control*. en. Control Engineering. Cham: Springer International Publishing, 2019. ISBN: 978-3-319-77488-6 978-3-319-77489-3. DOI: 10.1007/978-3-319-77489-3. URL: http://link.springer.com/10.1007/978-3-319-77489-3 (visited on 05/11/2024).

[45] Moritz Reuss et al. *Goal-Conditioned Imitation Learning using Score-based Diffusion Policies*. arXiv:2304.02532 [cs]. June 2023. URL: http://arxiv.org/abs/2304.02532 (visited on 10/30/2023).

[46] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2021. arXiv: 2112.10752 [cs.CV].

[47] Victor Garcia Satorras, Emiel Hoogeboom, and Max Welling. *E(n) Equivariant Graph Neural Networks*. arXiv:2102.09844 [cs, stat]. Feb. 2022. URL: http://arxiv.org/abs/2102.09844 (visited on 04/04/2024).

[48] Maximilian Sieb et al. *Graph-Structured Visual Imitation*. arXiv:1907.05518 [cs]. Mar. 2020. URL: http://arxiv.org/abs/1907.05518 (visited on 12/29/2023).

[49] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". en. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature16961. URL: https://www.nature.com/articles/nature16961 (visited on 05/09/2024).

[50] Oskar von Stryk. *Grundlagen der Robotik*. Ergänzendes Skriptum zur Vorlesung Grundlagen der Robotik. 2021.

[51] Faraz Torabi, Garrett Warnell, and Peter Stone. *Recent Advances in Imitation Learning from Observation*. arXiv:1905.13566 [cs]. June 2019. URL: http://arxiv.org/abs/1905.13566 (visited on 04/10/2024).

[52] Changhao Wang et al. "Offline-Online Learning of Deformation Model for Cable Manipulation with Graph Neural Networks". In: *IEEE Robotics and Automation Letters* 7.2 (Apr. 2022). arXiv:2203.15004 [cs], pp. 5544–5551. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2022.3158376. URL: http://arxiv.org/abs/2203.15004 (visited on 04/08/2024).

[53] Zheyuan Wang and Matthew Gombolay. "Learning Scheduling Policies for Multi-Robot Coordination With Graph Attention Networks". In: *IEEE Robotics and Automation Letters* 5.3 (2020), pp. 4509–4516. DOI: 10.1109/LRA.2020.3002198.

[54] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8 (2004), pp. 229–256. URL: https://api.semanticscholar.org/CorpusID:2332513.

[55] Lingfei Wu et al. *Graph Neural Networks for Natural Language Processing: A Survey*. 2022. arXiv: 2106.06090 [cs.CL].

[56] Lingfei Wu et al. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Singapore: Springer Singapore, 2022, p. 725.

[57] Shiwen Wu et al. "Graph Neural Networks in Recommender Systems: A Survey". en. In: *ACM Computing Surveys* 55.5 (May 2023), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3535101. URL: https://dl.acm.org/doi/10.1145/3535101 (visited on 05/10/2024).

[58] Omry Yadan. *Hydra - A framework for elegantly configuring complex applications*. Github. 2019. URL: https://github.com/facebookresearch/hydra.

[59] Yufei Ye et al. *Object-centric Forward Modeling for Model Predictive Control*. arXiv:1910.03568 [cs]. Oct. 2019. URL: http://arxiv.org/abs/1910.03568 (visited on 04/24/2024).

[60] Maryam Zare et al. *A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges*. arXiv:2309.02473 [cs, stat]. Sept. 2023. URL: http://arxiv.org/abs/2309.02473 (visited on 03/04/2024).

[61] Haodong Zhang et al. *Kinematic Motion Retargeting via Neural Latent Optimization for Learning Sign Language*. arXiv:2103.08882 [cs]. Feb. 2022. URL: http://arxiv.org/abs/2103.08882 (visited on 10/30/2023).

[62] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. arXiv:1812.08434 [cs, stat]. Oct. 2021. URL: http://arxiv.org/abs/1812.08434 (visited on 05/10/2024).

[63] Jie Zhou et al. "Graph neural networks: A review of methods and applications". en. In: *AI Open* 1 (2020), pp. 57–81. ISSN: 26666510. DOI: 10.1016/j.aiopen.2021.01.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S2666651021000012 (visited on 01/07/2024).

[64] Yi Zhou et al. *On the Continuity of Rotation Representations in Neural Networks*. arXiv:1812.07035 [cs, stat]. June 2020. URL: http://arxiv.org/abs/1812.07035 (visited on 04/13/2024).

[65] Yuke Zhu et al. *robosuite: A Modular Simulation Framework and Benchmark for Robot Learning*. arXiv:2009.12293 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2009.12293. URL: http://arxiv.org/abs/2009.12293 (visited on 02/13/2024).

A. Appendix

A.1. Learning Rate Schedule

In Figure A.1, we present the cosine learning rate schedule used to train both the GraphDDPM and ARGG policies, the schedule is updated for each optimization step.

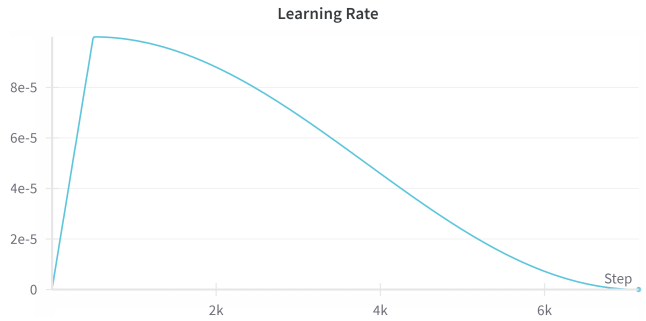


Figure A.1.: Cosine learning rate schedule with linear warmup used for training of both GraphDDPM and ARGP.

A.2. Implementation details

In the following, we provide additional implementation details in complement to the description of the approaches in Chapter 3.2

The graph encoder described in Subsection 3.2.1 contains $L = 3$ equivariant graph convolutional layers. A maximal number of IDs of 30 is used for the node ID embeddings,

as well as an embedding dimension of 16. The hidden dimension for the message-passing layers is the same as the respective denoising networks.

The denoising network from the GraphDDPM model has a hidden dimension of 256 and $L = 3$ message passing layers, also in the fashion of [47]. The GraphDDPM policy makes use of $K = 100$ diffusion steps. This value was chosen for being the number of diffusion steps in the Diffusion Policy [10], allowing us to compare the generation quality of the samples fairly. Altering the value of K for 50 and 200 did not significantly affect the success rates in the final experiments, so $K = 100$ was kept.

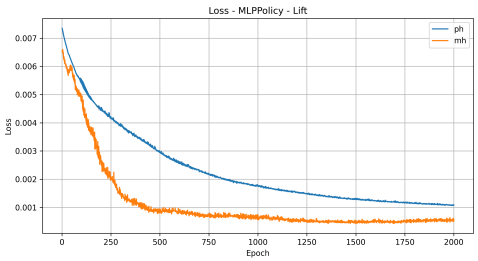
The denoising network in the ARGG model has a hidden dimension of 512 and $L = 2$ message passing layers.

A.3. Training logs

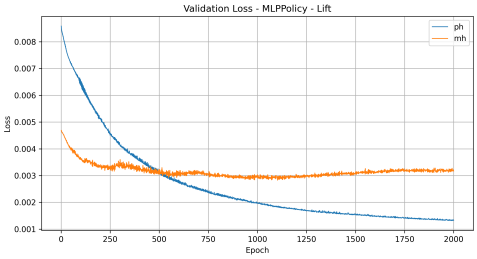
We present bellow the training logs for all policies in all tasks and dataset types, consisting of the model's loss function during the whole training process, as well as the same loss evaluated in the validation dataset.

A.4. Details in the training setup

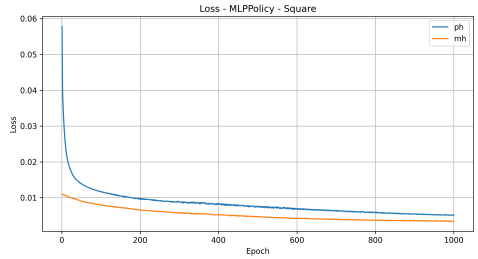
Most of the computation was performed in the Intelligent Autonomous Systems lab's computer cluster, providing one NVIDIA GeForce RTX 2080 Ti GPU per task, limited to 24h tasks. Having that in mind, all processes were constrained to a maximum running time of 24 hours. Even though training could be run in separate processes for several days, we chose to keep this limit since the most efficient low-dimensional policies (such as BC-RNN) can be trained within that time frame. This restriction impacts the number of training epochs, particularly in data sets involving multiple human demonstrations. Consequently, to ensure equitable evaluation across various task and data set types, the number of epochs considered during training was capped to match the epoch count of the multi-human data modality, presented in Table A.1. This approach aimed to maintain consistency and fairness in assessing model performance across different data types and tasks.



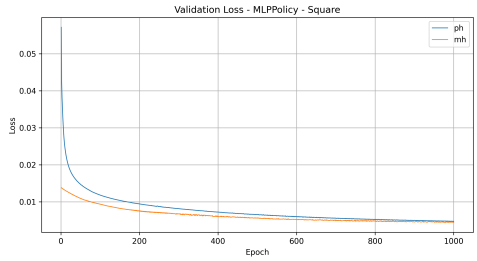
(a) MSE loss during training of the MLP policy in the Lift Lowdim dataset for 2000 epochs.



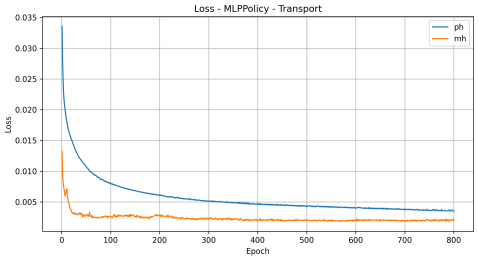
(b) MSE loss in the validation dataset during training of the MLP policy in the Lift Lowdim dataset for 2000 epochs.



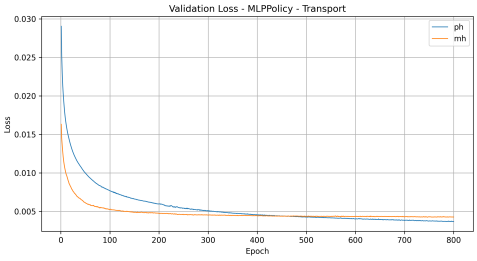
(c) MSE loss during training of the MLP policy in the Square Lowdim dataset for 1000 epochs.



(d) MSE loss in the validation dataset during training of the MLP policy in the Square Lowdim dataset for 1000 epochs.

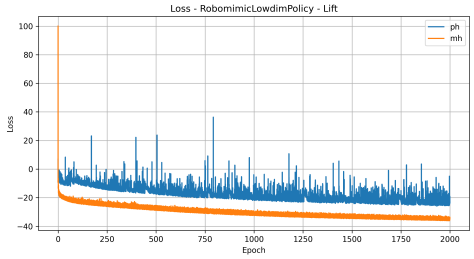


(e) MSEloss during training of the MLP policy in the Transport Lowdim dataset for 800 epochs.

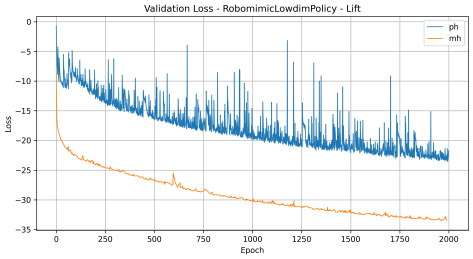


(f) MSE loss in the validation dataset during training of the MLP policy in the Transport Lowdim dataset for 800 epochs.

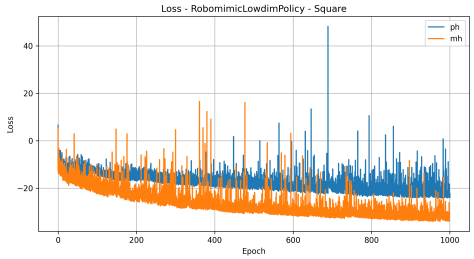
Figure A.2.: Plots of training and validation procedure for the MLP policy.



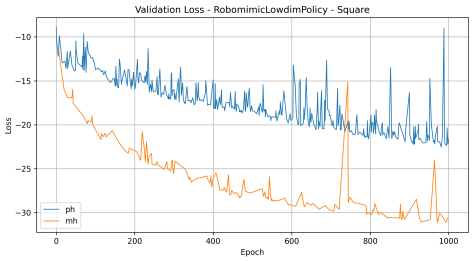
(a) Negative log-likelihod loss during training of the BC-RNN policy in the Lift Lowdim dataset for 2000 epochs.



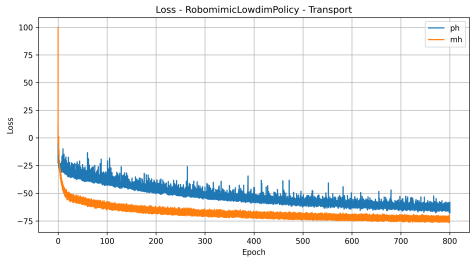
(b) Negative log-likelihod loss in the validation dataset during training of the BC-RNN policy in the Lift Lowdim dataset for 2000 epochs.



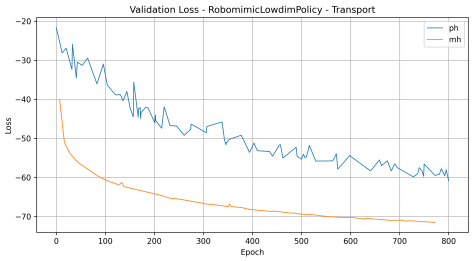
(c) Negative log-likelihod loss during training of the BC-RNN policy in the Square Lowdim dataset for 1000 epochs.



(d) Negative log-likelihod loss in the validation dataset during training of the BC-RNN policy in the Square Lowdim dataset for 1000 epochs.

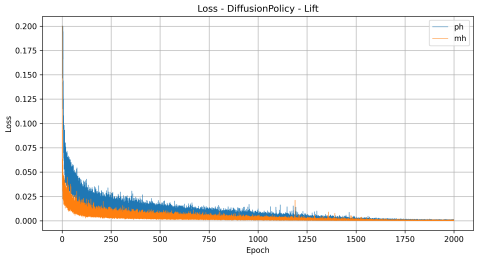


(e) Negative log-likelihod loss during training of the BC-RNN policy in the Transport Lowdim dataset for 800 epochs.

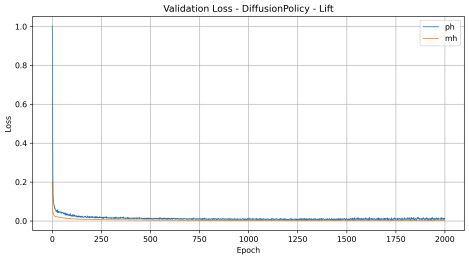


(f) Negative log-likelihod loss in the validation dataset during training of the BC-RNN policy in the Transport Lowdim dataset for 800 epochs.

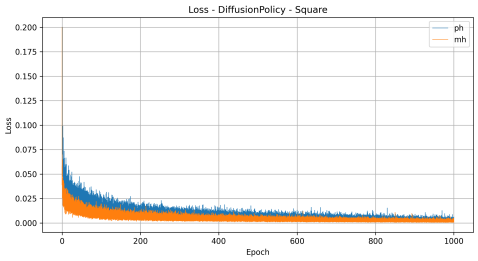
Figure A.3.: Plots of training and validation procedure for the BC-RNN policy [33].



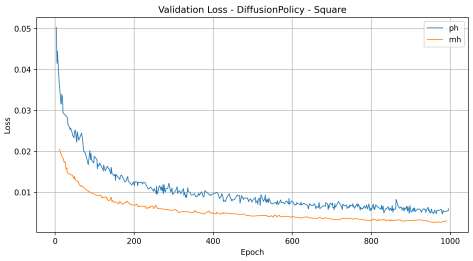
(a) Noise prediction loss during training of the Diffusion Policy in the Lift Lowdim dataset for 2000 epochs.



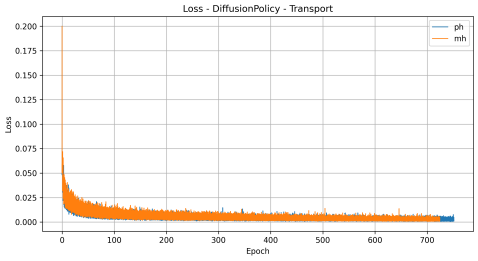
(b) Noise prediction loss in the validation dataset during training of the Diffusion Policy in the Lift Lowdim dataset for 2000 epochs.



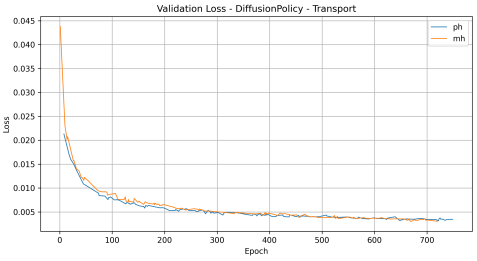
(c) Noise prediction loss during training of the Diffusion Policy in the Square Lowdim dataset for 1000 epochs.



(d) Noise prediction loss in the validation dataset during training of the Diffusion Policy in the Square Lowdim dataset for 1000 epochs.



(e) Noise prediction loss during training of the Diffusion Policy in the Transport Lowdim dataset for 800 epochs.

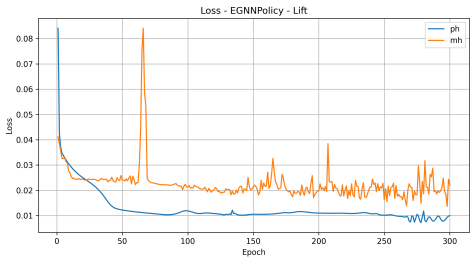


(f) Noise prediction loss in the validation dataset during training of the Diffusion Policy in the Transport Lowdim dataset for 800 epochs.

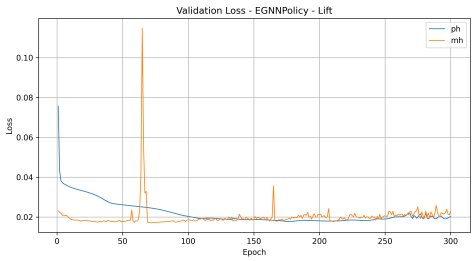
Figure A.4.: Plots of training and validation procedure for Diffusion Policy [10].

A.5. Sample of trajectory for all joints

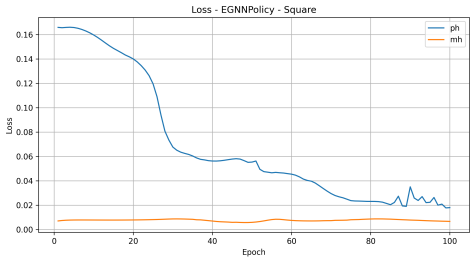
In completion of Figure 3.4, we present the results of the trajectory sampling for all joints in Figure A.8.



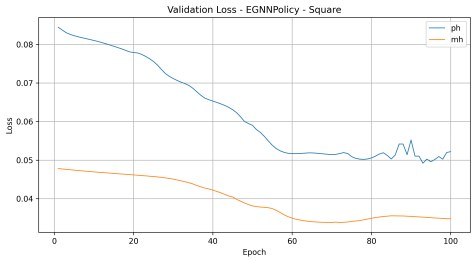
(a) Noise prediction loss during training of the EGNN Policy in the Lift Graph dataset for 300 epochs.



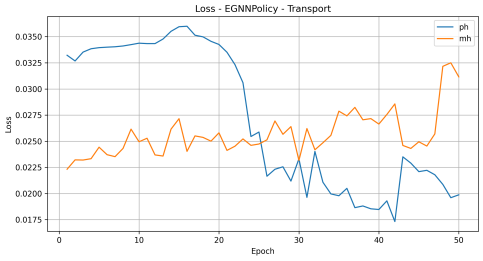
(b) Noise prediction loss in the validation dataset during training of the EGNN Policy in the Lift Graph dataset for 300 epochs.



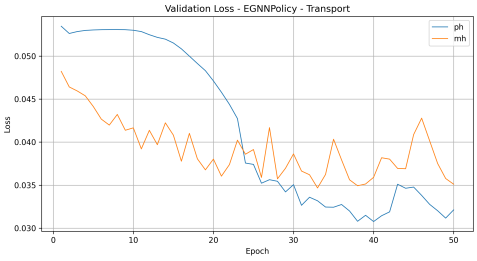
(c) Noise prediction loss during training of the EGNN Policy in the Square Lowdim dataset for 100 epochs.



(d) Noise prediction loss in the validation dataset during training of the EGNN Policy in the Square Graph dataset for 100 epochs.

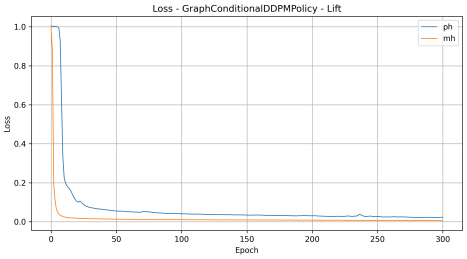


(e) Noise prediction loss during training of the EGNN Policy in the Transport Lowdim dataset for 50 epochs.

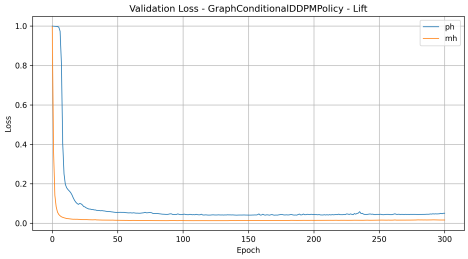


(f) Noise prediction loss in the validation dataset during training of the EGNN Policy in the Transport Lowdim dataset for 50 epochs.

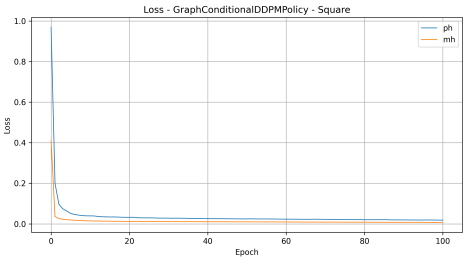
Figure A.5.: Plots of training and validation procedure for EGNN Policy.



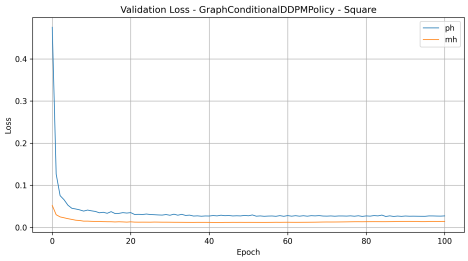
(a) Noise prediction loss during training of the GraphDDPM Policy in the Lift Graph dataset for 300 epochs.



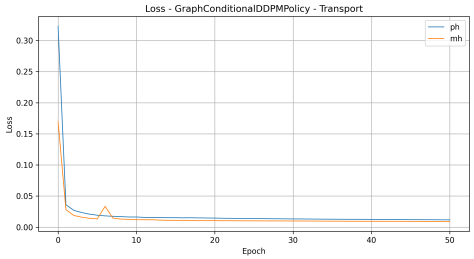
(b) Noise prediction loss in the validation dataset during training of the GraphDDPM Policy in the Lift Graph dataset for 300 epochs.



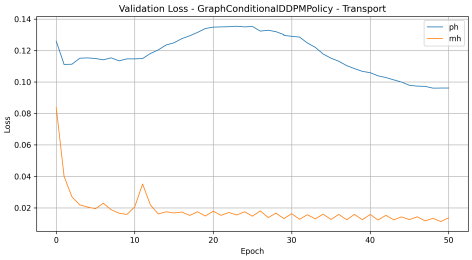
(c) Noise prediction loss during training of the GraphDDPM Policy in the Square Lowdim dataset for 100 epochs.



(d) Noise prediction loss in the validation dataset during training of the GraphDDPM Policy in the Square Graph dataset for 100 epochs.

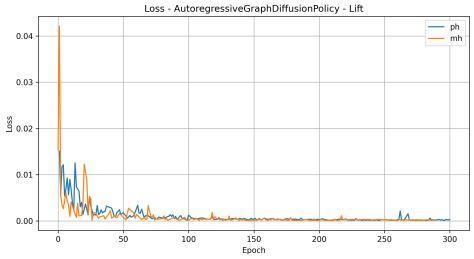


(e) Noise prediction loss during training of the GraphDDPM Policy in the Transport Lowdim dataset for 50 epochs.

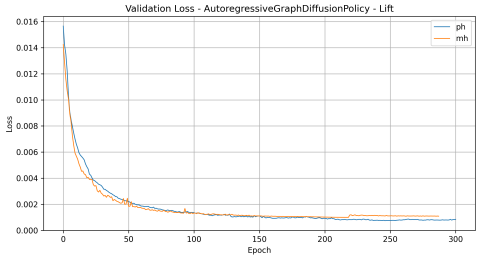


(f) Noise prediction loss in the validation dataset during training of the GraphDDPM Policy in the Transport Lowdim dataset for 50 epochs.

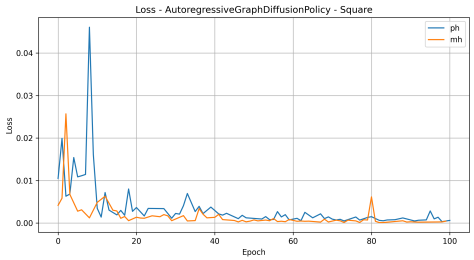
Figure A.6.: Plots of training and validation procedure for GraphDDPM Policy.



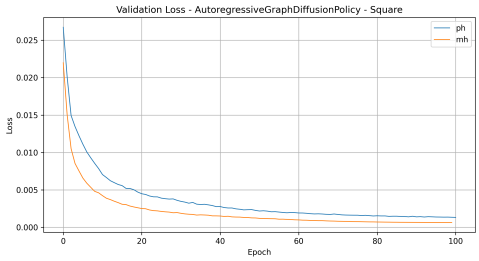
(a) Noise prediction loss during training of the ARGG Policy in the Lift Graph dataset for 300 epochs.



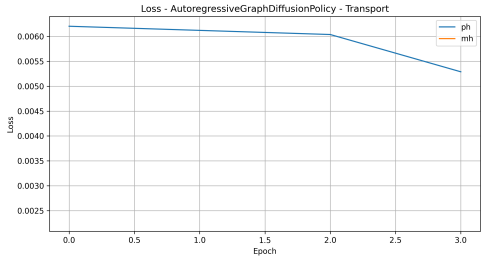
(b) Noise prediction loss in the validation dataset during training of the ARGG Policy in the Lift Graph dataset for 300 epochs.



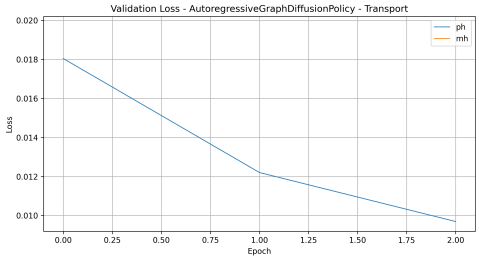
(c) Noise prediction loss during training of the ARGG Policy in the Square Lowdim dataset for 100 epochs.



(d) Noise prediction loss in the validation dataset during training of the ARGG Policy in the Square Graph dataset for 100 epochs.



(e) Noise prediction loss during training of the ARGG Policy in the Transport Lowdim dataset for 4 epochs in the ph dataset, 1 epoch in the mh dataset.



(f) Noise prediction loss in the validation dataset during training of the ARGG Policy in the Transport Lowdim dataset for 4 epochs in the ph dataset, 1 epoch in the mh dataset.

Figure A.7.: Plots of training and validation procedure for ARGG Policy.

Table A.1.: Number of training epochs for each data modality used to benchmark the policies concerning the tasks.

	Number of Training Epochs		
	Lift	Square	Transport
Lowdim	2000	1000	800
Graph	300	100	50

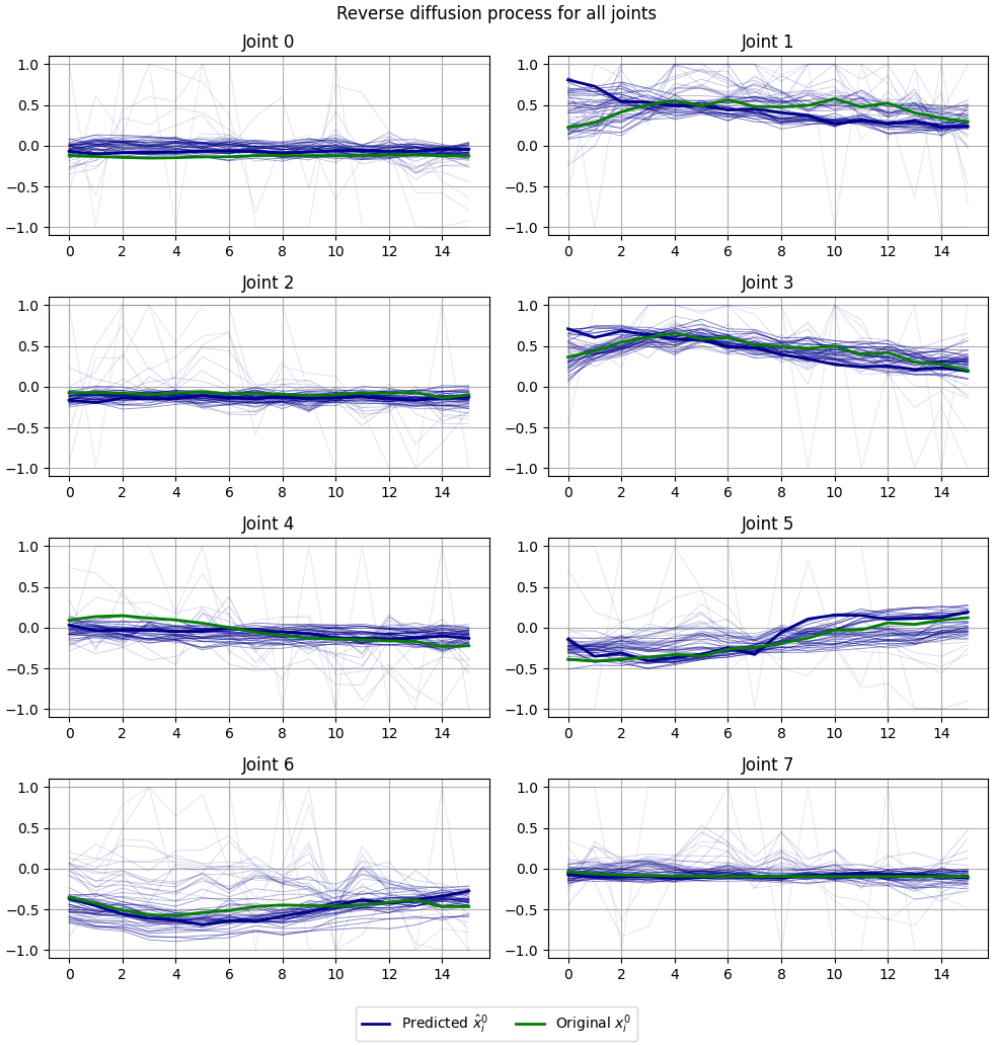


Figure A.8.: Single trajectory sampling for all diffusion steps from K to 0, highlighting final prediction and original action related to that observation on the graph dataset.