Karlsruhe Institute of Technology

Autonomous Learning Robots Lab

Improving the Rollout Stability of Graph-Network-based Simulators

Bachelor's Thesis
of

M. Rodi Düger

KIT Department of Informatics
Institute for Anthropomatics and Robotics (IAR)
Autonomous Learning Robots (ALR)

Referee: Prof. Dr. Techn. Gerhard Neumann

Advisor: M.Sc. Philipp Dahlinger

Duration: December 1st, 2023 — April 1st, 2024

Erklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, den 1. April 2024

M. Rodi Düger

Zusammenfassung

Die Simulation physikalischer Phänomene ist für viele wissenschaftliche und technische Disziplinen von entscheidender Bedeutung. Herkömmliche Simulatoren stoßen oft auf rechnerische Herausforderungen, die Kompromisse zwischen der Geschwindigkeit der Simulation, der Genauigkeit und der Komplexität der modellierten physikalischen Systeme erfordern. Neueste Fortschritte im Bereich des maschinellen Lernens und des Deep Learnings haben den Weg für den Einsatz neuronaler Netze im Kontext von Simulationen geebnet. Diese neuronalen Simulatoren können die Dynamik physikalischer Systeme direkt aus Daten lernen. Sie sind typischerweise darauf trainiert, den nächsten Zustand des Systems aus seinem aktuellen Zustand vorherzusagen. Allerdings stoßen neuronale Simulatoren auf Probleme, wenn sie autoregressiv im Laufe der Inferenz verwendet werden. Dies liegt daran, dass sie sich auf ihre eigenen Vorhersagen als Eingaben verlassen, was zu Fehlerakkumulation und Verschiebungen in der Verteilung führt. Dies wiederum beeinträchtigt ihre Genauigkeit und Stabilität über längere Rollouts. In dieser Arbeit bewerten wir bestehende Ansätze zur Adressierung von Fehlerakkumulation und Verteilungsverschiebungen in der Inferenz. Zusätzlich stellen wir ein neues Framework vor, das einen Puffer in das Training integriert und sich von Replay-Buffern im Reinforcement-Learning inspirieren lässt. Der vorgeschlagene Ansatz gibt Forschern mehr Kontrolle über den Trainingsprozess und erreicht eine vergleichbare Genauigkeit bei Langzeit-Rollouts mit bestehenden Methoden.

Abstract

Simulating physical phenomena is essential for many disciplines in science and engineering. Traditional simulators often encounter computational challenges, requiring trade-offs between simulation speed, accuracy, and the complexity of the modeled physical systems. Recent advances in machine learning and deep learning have paved the way for the use of neural networks in simulation contexts. These neural simulators can learn the dynamics of physical systems directly from data. Typically trained to predict the system's next state from its current state, neural simulators run into issues when used autoregressively during inference. This is because they rely on their own predictions as inputs, leading to error accumulation and distribution shifts. This, in turn, affects their accuracy and stability over extended rollouts. In this thesis we evaluate existing approaches to address error accumulation and distribution shifts in inference. Additionally, we introduce a novel framework, which incorporates a buffer into the training loop, drawing inspiration from replay buffers in reinforcement learning. The proposed approach gives researchers more control over the training process and achieves comparable accuracy in long-horizon rollouts with existing methods.

Table of Contents

Chapter 1

Introduction

Physical simulations play an important role in many engineering disciplines. They are employed for tasks as varied as analyzing the structural integrity of buildings and bridges under stress [29], optimizing the aerodynamic design of vehicles [1] and simulating vehicle crashes to improve safety [2]. Simulations also play a pivotal role in scientific fields. Meteorologists use them to refine weather forecasts and climate models [25], while astrophysicists simulate the formation of galaxies and stars to unravel the mysteries of the cosmos [11]. Even at the atomic scale, material scientists leverage simulations to design advanced materials and study their fundamental properties [20].

Traditional physical simulations, despite their value, often encounter computational limitations that restrict their scope and accuracy. Explicit numerical solvers, a common approach in these simulations, require small simulation time steps for stability [28]. This constraint results in extended computation times. Simulating large-scale, complex systems with high precision can easily become computationally expensive, demanding extensive resources and time. Additionally, intricate phenomena with many interacting components or non-linear behaviors can strain the capabilities of traditional simulators. Consequently, traditional simulators often face trade-offs between simulation speed, accuracy, and the complexity of the system they aim to model.

Recent advances in deep learning have opened doors for novel approaches to physical simulations. Neural simulators, a new paradigm, leverage the expressiveness of neural networks to address the limitations of traditional methods. By learning patterns from data, these models can capture complex system dynamics without the need for explicit, hand-crafted numerical solvers. Neural networks also offer inherent advantages for

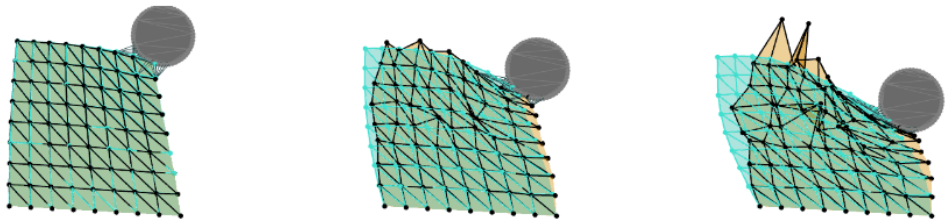


Figure 1.1: Visual comparison of ground truth (green) deformable object mesh with a mesh simulated by an autoregressive neural simulator trained in a one-step fashion (orange). Snapshots progress from left to right across increasing time steps.

parallelization, leading to potential speedups when utilizing specialized hardware. [14] Hardware accelerators like Graphics Processing Units (GPUs) are specifically designed for efficient parallel processing, making them ideal for accelerating these simulations. This parallel processing capability significantly reduces overall simulation time compared to traditional Central Processing Unit (CPU)-based approaches. While still an active research area, neural simulators hold promise for faster, more scalable, and potentially accurate simulations, particularly in domains where traditional methods struggle. However, as with any emerging advancement, there are challenges that need to be addressed.

One particularly significant challenge in neural simulators is ensuring their stability over long rollouts in inference. A common training approach for such simulators involves one-step prediction, where the model predicts the system's state at the subsequent time step based on its current state. The model's prediction is then compared against the ground truth for optimization. However, during inference, the model relies on its own previous output as input in an autoregressive manner. This can lead to error accumulation, as the model's subsequent predictions are based on its potentially imperfect estimates from previous steps. Small errors or deviations at each step can compound over a long rollout, leading to unreliable predictions. This issue is further amplified by distribution shift, where the simulator might encounter states outside of its training data distribution, potentially leading to unpredictable behavior. This compounding effect is evident in Figure 1.1, where the divergence between the predicted state and ground truth increases over time While this is a common challenge for autoregressive models in general, addressing this issue is especially crucial in the context of autoregressive neural simulators. In simulations, accuracy is required not only in the final outcome but also throughout the entire predicted trajectory to reflect the evolution of the system over time as accurately as possible.

Given these challenges, this thesis aims to address the stability issues during rollouts in neural simulators and bridge the gap between training and inference. In our experiments we use the Graph-Network-based Simulators (GNS) framework [23] adapted to mesh-

based simulation setting, similar to MeshGraphNet (MGN) without adaptive mesh-refinement [21]. In that regard, in Chapter 2 we first briefly convey the fundamentals necessary to understand the underlying model we use in our work. Chapter 3 concisely reviews existing techniques to mitigate error propagation and distribution shift, which we have implemented and evaluated.

The core contribution of this thesis, presented in Chapter 4, is a novel buffer-enhanced training technique. It aims to enhance researcher flexibility and potentially improve the performance of neural simulators. Chapter 5 introduces the datasets employed to evaluate and compare the existing approaches with our buffer-enhanced training method. Chapter 6 presents experimental results across diverse settings. While the ability to directly outperform established baselines is important, our primary goal with these evaluations is to demonstrate that buffer-enhanced training provides researchers with greater adaptability and flexibility. It allows for the fine-tuning of the training process through readily adjustable hyperparameters. Finally, in Chapter 7, we summarize the key findings of this work and outline promising directions for future research in this domain.

Chapter 2

Fundamentals

2.1 Graphs

A graph,denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a mathematical construct comprising:

- **Node (Vertex) Set ($\mathcal{V}$):** A finite set of nodes (vertices) .
- **Edge Set ($\mathcal{E}$):** A set of two-element subsets of V, representing connections between nodes. An edge $(u, v) \in \mathcal{E}$ indicates an edge between nodes u and v.

Nodes can be considered as an abstraction for a set of objects and edges encode the relations between these objects.

Additional Concepts and Notation

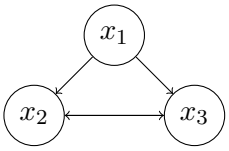
The neighborhood of a node v, denoted as $\mathcal{N}(v)$, is the set of nodes directly connected to v by an edge

$$\mathcal{N}(v) = \{u \in \mathcal{V} | (u, v) \in \mathcal{E}\}.$$

The degree of a node v, denoted as $\deg(v)$ or $|\mathcal{N}(v)|$, is the number of edges incident to v. There is also a distinction between directed and undirected graphs. In directed graphs, edges have an associated direction, and (u, v) is distinct from (v, u). In undirected graphs, edges are bidirectional, and (u, v) is equivalent to (v, u) and hence

edges can be represented as u, v. Throughout the thesis we will be meaning directed graphs when we refer to a graph.

We can represent the graph



in two different ways:

- **Adjacency Matrix:** A square matrix where the element at row i and column j indicates the presence or weight of an edge between nodes i and j.

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

- **Adjacency Lists:** An array of lists, where each list corresponds to a node and holds its neighbors.

$$[[x_2, x_3], [x_3], [x_2]]$$

Examples

Social Network: Nodes represent individuals; edges represent friendships. An example visualization can be seen in Figure 2.1.

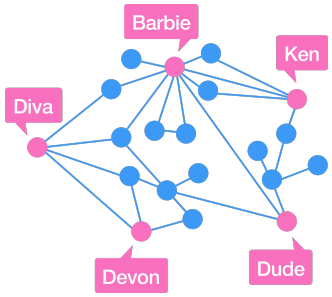


Figure 2.1: Graph representation of a social network. Source: Social Networks. Brilliant.org. Retrieved 21:20, March 28, 2024, from https://brilliant.org/wiki/social-networks/

Knowledge Graph: Nodes represent concepts or entities; edges represent various types of relationships ,e.g., "is-a". Figure 2.2 visualizes a knowledge graph.

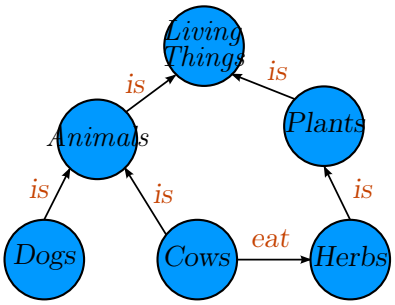


Figure 2.2: An example knowledge graph by Jayarathina Madharasan under Creative Commons Attribution-Share Alike 4.0 International license. https://commons.wikimedia.org/wiki/File:Conceptual_Diagram_-_Example.svg

2.2 Mesh-Based Finite Element Method

Finite Element Method (FEM) is a powerful computational tool for approximating solutions to engineering and physics problems characterized by complex geometries and heterogeneous material properties.[22] While many physical phenomena are fundamentally described by Partial Differential Equations (PDEs), obtaining analytical solutions for real-world problems is often intractable due to complexities in geometry and boundary conditions. Moreover, representing a continuous object with its inherent infinite degrees of freedom makes direct numerical solution impractical.

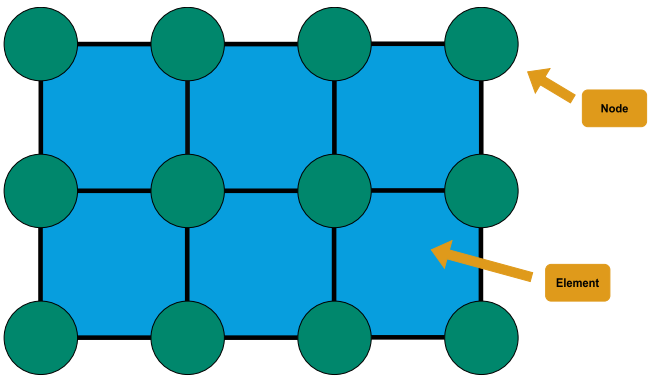


Figure 2.3: A simple mesh modelled as a graph

To address these challenges, the FEM discretizes the problem domain into a finite number of elements, referred to as a mesh. Within each element, the governing PDEs are approximated, allowing for numerical solutions. Subsequently, these element-wise solutions are interpolated to approximate the behavior across the entire continuous domain. A mesh can be modelled as a graph as it can be seen in Figure 2.3.

2.3 Message Passing Neural Networks

Graph Neural Networks (GNNs) are a neural networks designed to operate directly on graph data. Graphs provide a versatile structure to model relationships between entities making GNNs applicable to domains where data contains rich relational structure. Researchers have proposed a diverse range of GNNs architectures to address challenges in graph-based machine learning, including Message Passing Neural Networks (MPNNs) [10] as a unifying framework for various architectures. MPNNs offer a powerful framework for learning representations of nodes, edges, or entire graphs. Since the models employed in this work fall within the MPNNs, this section provides an introductory overview of the message passing paradigm within the GNN context.

Let a graph be denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. In addition to the formal graph definition, each node $v \in \mathcal{V}$ and each edge $(v, u) \in \mathcal{E}$ has an associated feature vector $\mathbf{h}_v$ and $\mathbf{e}_{vu}$. We also introduce a global node x_g with the corresponding feature vector h_g to encode global information. We include node and edge features to formalize MPNNs. In the context of MPNNs, edge features are often overlooked during introduction because MPNNs, by design, allow to utilize edge features when necessary, but we explicitly include them in our notation because they hold significance in the context of simulators.

MPNNs perform computations in iterative steps. In each step, the following operations are performed.

Message Passing: For each edge (v, u) from node v to node $u \in \mathcal{N}(v)$ a message $\mathbf{m}_{vu}$ is computed, defined as an edge update rule.

$$\mathbf{e}_{vu}^{t+1} = M_t(\mathbf{h}_v^t, \mathbf{h}_u^t, \mathbf{e}_{vu}^t, \mathbf{h}_g^t).$$

M_t denotes the message function, which updates the edges, at iteration $t \geq 1$. It is often a learned differentiable function, such as a neural network.

Aggregation: Each node aggregates messages in its neighborhood to compute an aggregated message

$$\mathbf{m}_v^t = \bigoplus_{u \in \mathcal{N}(v)} \mathbf{e}_{vu}^{t+1}.$$

The symbol $\bigoplus$ denotes a permutation invariant aggregation function. Permutation invariance ensures that the result is independent of the order, in which neighborhood elements are processed, as the neighborhood of a node is inherently an unordered set.

Update: Node features are updated by combining the aggregated message $\mathbf{m}_v^t$,the node's current features $\mathbf{h}_v^t$ and the features of the global node $\mathbf{h}_g^t$

$$\mathbf{h}_v^{t+1} = U_t(\mathbf{h}_v^t, \mathbf{m}_\mathbf{v}^\mathbf{t}, \mathbf{h}_g^t).$$

Global features are updated via

$$\mathbf{h}_g^{t+1} = G_t(\bigoplus_{v \in \mathcal{V}} h_v^{t+1}, \bigoplus_{(v,u) \in \mathcal{E}} e_{vu}^{t+1}, h_g^t)$$

U_t denotes the node update function and G_t denotes the global node update function at iteration $t \geq 1$. They are mostly learned differentiable functions, such as a neural networks. Unlike the way the term "number of layers" is used in deep neural networks, in an MPNN, the "number of layers" refers to the number of message passing steps. The choice of message functions, aggregation functions, and update functions provides significant flexibility [13, 27]. This allows MPNN architectures to capture complex relationships and dependencies within graph data. MPNNs' focus on local message passing naturally emphasizes learning relationships between nearby nodes, making them well-suited for physical simulations where interactions are often governed by proximity [5, 23, 21].

2.4 Graph Network-based Simulators

The term *GNS* [23] refers to a framework proposed by Sanchez-Gonzalez et. al. In this framework, the state of a system governed by physical laws is represented by a set of particles. Each particle essentially functions as a point of computation, encoding the physical properties of a local domain. This collection of particles and their interactions can be modeled as a graph, with particles serving as nodes and their interactions represented by edges. The system's evolution over time is simulated through message passing along the graph's edges, facilitating information exchange and subsequent state updates for the particles. While the framework proposed by Sanchez-Gonzalez et al. focuses on particle-based simulation, its core principles and model architecture can be adapted for mesh-based simulations as well. [21]

Model Architecture

The GNS framework proposes a parameterized autoregressive simulator s_θ. Given the state of the system in the time step t, the simulator predicts the state of the system in the next time step $t+1$, by predicting dynamics of the system leveraging an "Encode-Process-Decode" scheme as depicted in Figure 2.4(a).

Let $\mathcal{X}$ be the world space and $X^t \in \mathcal{X}$ the state of the system at time step t. The prediction of the model is

$$\tilde{X}^{t+1} = s_\theta(X^t).$$

Encoder: The encoder takes the system state as a set of particles, denoted as X^{t_0}, as input and constructs a latent graph, denoted as G^0. This latent graph encodes the

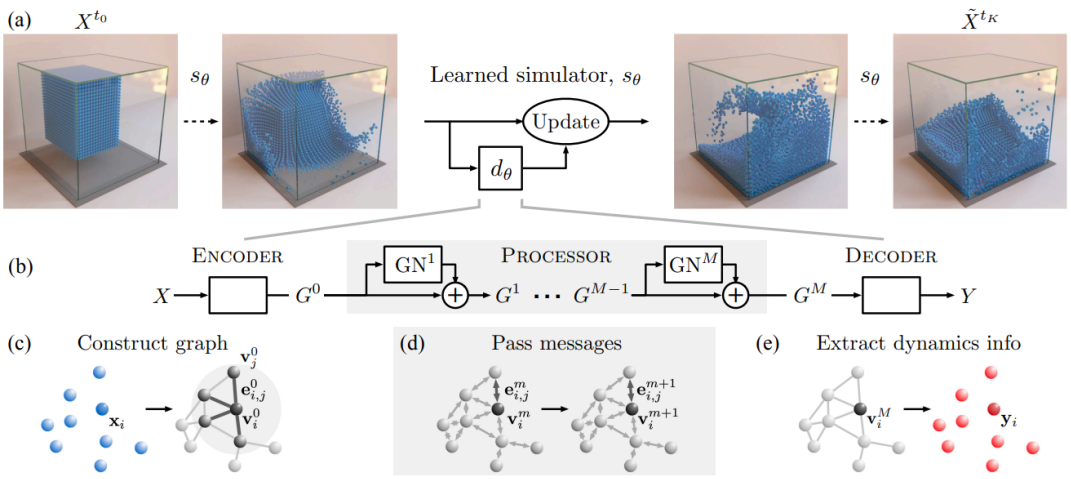


Figure 2.4

relationships and interactions between the particles. The directed edges are created with respect to the spatial proximity of its' source and target nodes. Node and edge embeddings are learnable functions, e.g. neural networks.

Processor: The processing stage iterates M times. In each iteration $i \in \{1, ..., M\}$, a Graph Network (GN) block [4] is used to perform message passing over the current latent graph, G^{i-1}, to generate an updated latent graph, G^i. Message passing allows particles to exchange information and update their representations based on the information received from their neighbors in the graph.

Decoder: After M rounds of processing, the decoder takes the final latent graph, G^m and extracts the dynamics information from the refined representation of the system's state, denoted as Y. As the decoder is learned, it causes the decoded representation Y to capture useful information regarding dynamics of the system, which is inferred from the interactions between the particles throughout the processing stages.

Chapter 3

Related Work

Stability over long rollouts is a critical objective for neural simulators to accurately simulate real-world phenomena. The autoregressive nature of neural simulators, coupled with the potential for distribution shift during inference as a result of error accumulation, can severely compromise prediction accuracy over long rollouts. To mitigate these issues, researchers have proposed some approaches. In this chapter, we focus on existing methods for improving simulator stability, such as introducing noise during training and leveraging the pushforward trick.

3.1 Training Noise

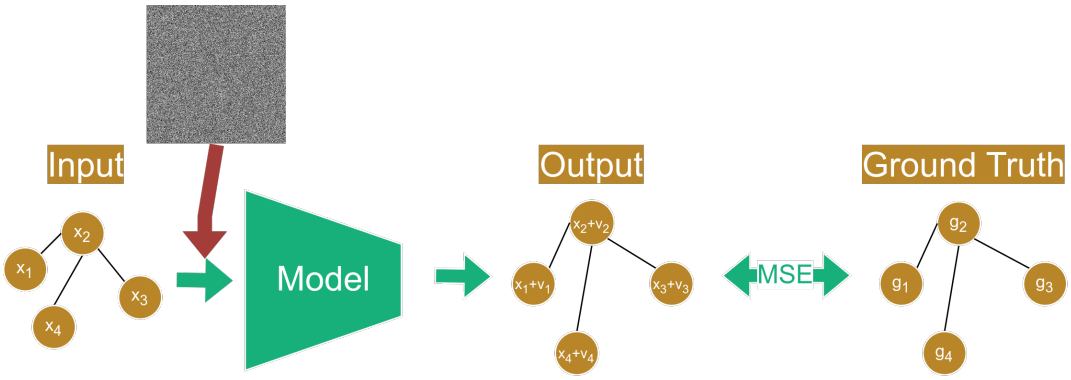


Figure 3.1: A simple visualization of the training with noise introduced

A potential drawback of training a neural simulator solely on ground truth data in a one-step prediction fashion is that the model is not exposed to the noisy inputs it will inevitably encounter during inference. This is due to its autoregressive nature, where it must rely on its own imperfect outputs as subsequent inputs. To address this and improve the stability, one approach is to introduce noise into the training data. The rationale behind this strategy is simple: by exposing the model to noisy data during training, the goal is to improve its ability to handle the imperfect predictions it will generate during inference. See Figure 3.1 for a simplified visualization of this training process.

The concept of introducing noise into training data is broad, and naturally raises questions about the structure of the noise and how it should be applied within the training process. For example, GNS introduced in Section 2.4 and MGN, an extension of GNS to mesh-based representation, [21] utilize a straightforward approach. They draw independent samples from a gaussian distribution of zero mean and fixed variance, using these samples to perturb the dynamics-related features of the input state. An important consideration when perturbing data with Gaussian noise is the implicit assumption that the model-induced noise in the output also follows a Gaussian distribution. This is a simplifying assumption, and it's likely that the true noise characteristics generated by the model during inference may be more complex.

Throughout this work, when referring to the perturbation of training data with noise, we specifically mean drawing samples from a Gaussian distribution with zero mean and fixed variance. These samples are then used to perturb the input state, consistent with the approach employed in GNS and MGN frameworks.

3.2 Pushforward Trick

The pushforward trick is a training strategy introduced by Brandstetter et al. [7], and serves as one of the baseline methods against which we compare our buffer-enhanced approach. Rather than training the model to predict the next system state based on ground truth or intentionally noisy data (as described in Section 3.1) the pushforward trick unrolls the simulator for 2 time steps. However, crucially, error is backpropagated only through the final time step and not through the first one. See Figure 3.2 for an illustration of training with pushforward trick.

Essentially, the pushforward trick represents an alternative approach to introducing noise during the training of neural simulators. Compared to directly perturbing training data with Gaussian noise before feeding it to the model, the pushforward trick avoids making an implicit assumption that the model-induced noise follows a Gaussian distribution. Instead, by leveraging the simulator itself to generate perturbations, this approach aims to capture the true noise characteristics produced by the model during inference more accurately.

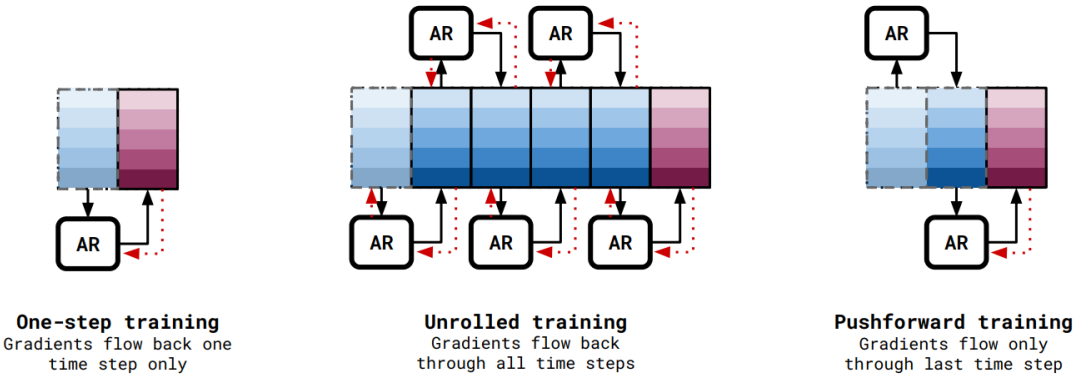


Figure 3.2: An illustration of three different training strategies taken from Brandstetter et. al. [7] One-step training predicts the subsequent state based on ground truth data and backpropagates through the one time step. Unrolled training predicts N time steps into the future and backpropagates through all of the time steps. Pushforward training uses the model's one-step prediction as the input for another one-step prediction but backpropagates only through the last time step.

3.3 Multi-Step Training

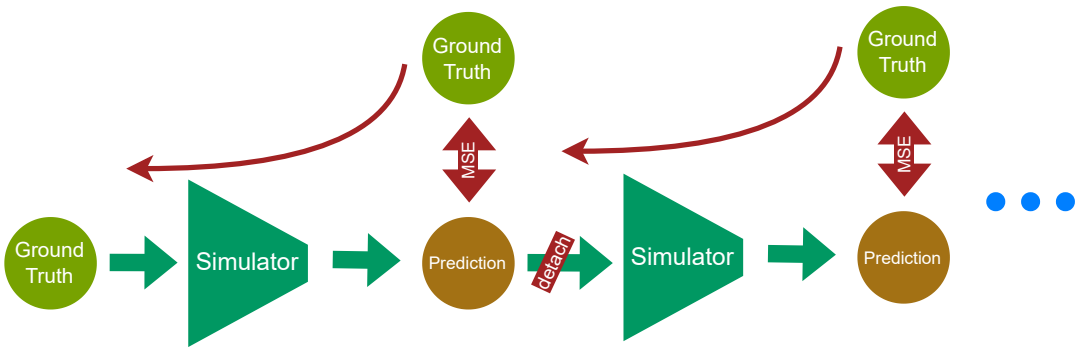


Figure 3.3: Visualization of the Multi-Step Training

Multi-step training offers an alternative to the conventional one-step training approach [26]. This method involves utilizing a hyperparameter s, which determines the number of time steps for unrolling the simulator starting from a given state. for each one-step prediction, we compare the predicted outcome with the ground truth. This comparison is followed by a backward pass to calculate and accumulate gradients. After this process, the prediction is detached from the computation graph. This cycle of prediction, comparison, backward pass, and detachment is repeated for each step until the simulator has been unrolled for s designated time steps. This means before updating the model weights, we accumulate gradients for more than one step. In Figure 3.3, a visualization is presented to illustrate the multi-step training approach that serves as one of our baselines.

Chapter 4

Buffer-Enhanced Training

In this chapter, we present our approach as a more general alternative to training strategies for neural simulators.

4.1 Intuition

While our approach draws inspiration from the concept of replay buffers in Reinforcement Learning (RL), it's important to note that we don't strictly "replay" training data in the traditional sense. A more accurate interpretation of our buffer-enhanced training is as a combination of multi-step and one-step training approaches.

Replay buffers are fixed-size data structures primarily designed to store past experiences encountered by off-policy RL agents. This enables the reuse of collected experiences, enhancing sample efficiency [18]. An experience typically include state observation, action taken (in RL), subsequent state, and corresponding reward. Replay buffers optimize data usage and enable multiple updates from a single transition by storing and reusing past experiences. Random sampling from the buffer is also crucial for the stability and convergence of many learning algorithms as it breaks the temporal correlation of sequentially generated data [17, 19].

We acknowledge that replay-buffer-like concepts are less established in supervised learning compared to their widespread use RL for several reasons. In supervised learning setting, data is usually static and stored in datasets for training. In contrast, in RL, data is dynamically gathered during interactions with the environment and

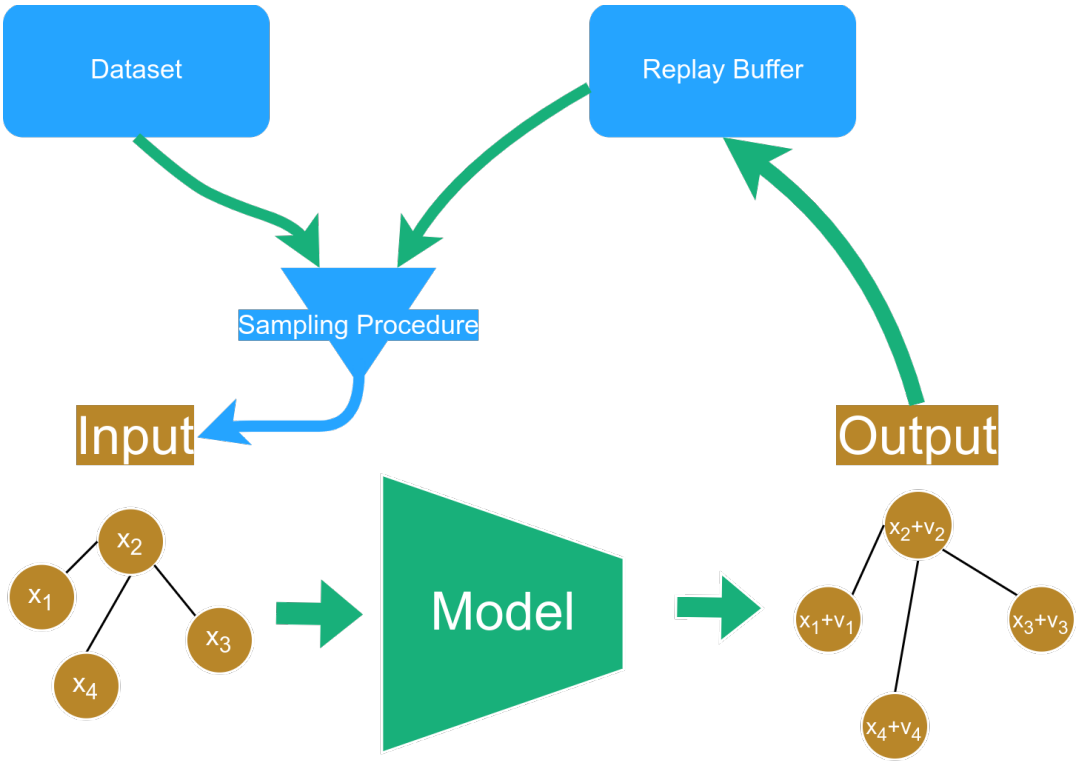


Figure 4.1: An overview of the replay-buffer-enhanced training

temporarily stored in a replay buffer. These buffers are dynamic structures that store transitions or trajectories experienced by an RL agent, enabling more efficient learning from past interactions. In contrast, supervised learning methods do not involve the same dynamic data collection process that necessitates the use of replay buffers. However, we hypothesized that introducing a buffer to the training of autoregressive neural simulators could offer researchers greater flexibility and control over the training procedure, drawing inspiration from the use of replay buffers in RL. The existing literature on replay buffers also provides a rich foundation for exploring techniques to optimize a training procedure with a buffer in loop.

4.2 Overview

Figure 4.1 outlines our buffer-enhanced training approach. The dataset comprises pairs of ground truth system states (s_t, s_{t+1}). A sampling procedure determines whether pairs of data are drawn from the ground truth dataset or from the replay buffer. If the buffer is empty or lacks sufficient samples for a batch, sampling occurs directly from the ground truth dataset.

After a pair or batch of data is sampled, it's fed into the model, which generates a prediction for the system's state at the next time step. The loss between this prediction and the corresponding ground truth data is then calculated. The simulator's prediction for time step t+1 is stored within the buffer. To track how many time steps into the future we predict based on a single ground truth sample, we keep track of how many times a data fed into the model and placed within the buffer. We refer to this hyperparameter as `maximum number of forward passes`. This tracking enables us to potentially limit the number of future time steps the model predicts from a given sample. You can see the step by step description of the training procedure in Algorithm 1.

Algorithm 1 Buffer-enhanced Training

Require: Dataset, Sampling procedure, Buffer size b, maximum number of forward passes k starting from a ground truth state, simulator s_θ

$B \leftarrow$ Initialize buffer with size b
if isEmpty(B) or length(B) $<$ batch_size **then**
    $(s_t, s_{t+1}, 0) \leftarrow$ sampleFromDataset()
else
    $(s_t, s_{t+1}, i) \leftarrow$ sampleWithSamplingProcedure()
end if
$\tilde{s}_{t+1} \leftarrow s_\theta(s_t)$
loss $\leftarrow$ MSE($\tilde{s}_{t+1}$, s_{t+1})
Optimize model parameters θ for loss using gradient descent
$i \leftarrow i + 1$
if $i \leq k$ **then**
    Store $\tilde{s}_{t+1}$ in buffer B
end if

4.3 Buffer Implementation

Our buffer-enhanced training approach fundamentally integrates a buffer into the training loop. This aligns training with inference by exposing the simulator to ground truth data perturbed by model-induced noise. The buffer is, therefore, a crucial component of our implementation. However, developing a performant and flexible replay buffer from scratch can be complex. To address this, we used the efficient and highly customizable replay buffer implementation provided by the TorchRL library [6], eliminating the need to reinvent existing solutions. The replay buffer implementation from TorchRL offers us the necessary flexibility to integrate custom writers and samplers. This customization is essential for our use case, as we primarily work with data structures from PyTorch Geometric [9]. These structures are less conventional within typical RL settings, requiring tailored processing to effectively utilize them within the replay buffer.

Chapter 5

Datasets

5.1 2D Deformable Plate

To train, test, and evaluate our models with various training techniques, we use the 2D Deformable Plate dataset. This dataset contains 945 trajectories of a round object (collider) falling onto a square-shaped object and causing deformation. Each trajectory is divided into 51 discrete time steps, with each data point representing the object's mesh and collider at that specific moment. Each data point is represented as a graph with 138 nodes and 756 edges. The Simulation Open Framework Architecture (SOFA) framework [8] was used to generate these trajectories, providing the ground truth simulation data. You can see figure 5.1 for an example of a data point from the dataset.

We allocate 70 % of the trajectories for training data, 15% for test data, and the remaining 15% for validation data. Since the dataset is 2D, the loading and computation time is relatively small, allowing for quick debugging. However, it still provides sufficient insight into the effectiveness of different training techniques when compared to each other.

The dataset includes deformation trajectories for objects made of three different materials with varying Poisson ratios, a fundamental material property [15]. This diversity introduces multimodality into the dataset, as the Poisson ratio greatly affects how a material deforms. In our experiments, we include the normalized Poisson ratio as a node feature in the graph representation of the mesh.

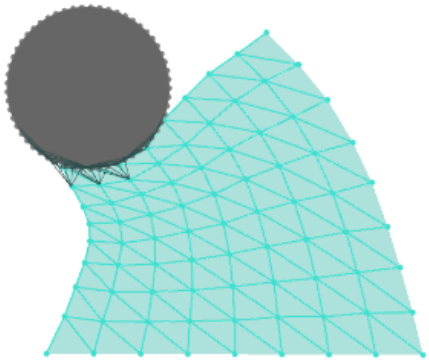


Figure 5.1: Visualization of a data point from 2D Deformable Plate dataset

5.2 3D Tissue Manipulation

We also use the 3D Tissue Manipulation dataset to benchmark different training techniques. This dataset has 840 trajectories showing a 3D object (tissue) being manipulated by a pulling force (see figure 5.2). Same as the 2D Deformable Plate dataset, the SOFA framework [8] was used to generate these trajectories, providing the ground truth simulation data. Each trajectory has 104 discrete time steps. Each data point is a graph with 362 nodes and 2204 edges. Similar to the 2D Deformable Plate dataset, it includes trajectories with diverse Poisson ratios to introduce multimodality. Normalized Poisson ratio is included in features of all mesh nodes. We maintain a consistent 70:15:15 split for training, testing, and validation data, respectively.

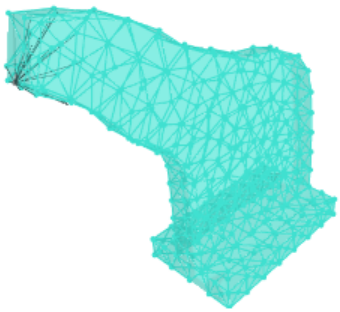


Figure 5.2: An example data point from 3D Tissue Manipulation dataset

Chapter 6

Evaluation

In this chapter, we evaluate various training techniques for neural simulators and compare them with our buffer-enhanced training approach. We first present our experimental setup and then discuss results from our experiments.

6.1 Experimental Setup

6.1.1 Simulator

While the primary focus of this thesis is to compare various training techniques for neural simulators, it is important to consider the architecture and hyperparameters of the underlying simulator for the sake of reproducibility and transparency.

We use the *Encode-Process-Decode* architecture presented in section 2.4 as the architecture of our underlying simulator across all experiments.

Encoder: We use an encoder, which embeds node features, edge features, and global features, each individually into a 128-dimensional latent space. We use a single-layer Multi Layer Perceptron (MLP) with LeakyReLU activation function for each feature type.

Processor: We use a MPNN with 5 layers as described in section 2.3, which means we perform 5 message passing steps. We use a separate single-layer MLP with

Component	Hyperparameter	Value
Encoder	Latent Dimension	128
	Architecture	Single-layer MLP
	Activation	LeakyReLU
Processor	Architecture	MPNN with 5 Layers
	Activation	LeakyReLU
	Update Functions	Separate Single-Layer MLPs
	Aggregation	Mean
	Normalization	LayerNorm
	Residual Connections	Yes
Decoder	Decoding Module	Single-Layer MLP
	Decoding Module Activation	ReLU
	Readout Module	Single-Layer MLP
	Readout Module Activation	None
Training	Batch Size	32
	Optimizer	Adam
	Learning Rate	5e-4
	Epochs	1000

Table 6.1: Overview of Hyperparameters and Architectures used in Experiments

LeakyRELU activation function as update function of each latent feature type (node, edge, global) at each message passing step. After each message passing step we apply layer normalization (LayerNorm) [3] to output features. We utilize *mean* to aggregate edge features for the node update and to aggregate node and edge features for the global feature update. In addition, residual connections between message passing blocks are employed against over-smoothing [16]

Decoder: The decoder comprises two modules: a single-layer MLP with a Rectified Linear Unit (ReLU) activation function, which extracts dynamics-related information from the last latent graph's node features, and a single linear layer without any activation function, functioning as a readout module to predict velocities of each node.

6.1.2 Training

We employ a batch size of 32 and the Adam optimizer [12] with a learning rate of 5e-4 throughout all our experiments, training the simulator for 1000 epochs. However, due to computational constraints and time limitations on the clusters used for training, some experiments with the 3D Tissue Manipulation dataset (section 5.2) run for fewer than 1000 epochs.

You can see the summary of the hyperparameters and architectures in the section 6.1.

6.1.3 Evaluation Metrics

Before presenting the results, it is important to mention the evaluation metrics we use to measure rollout stability and how we refer to them.

Full Rollout Last Mean Squared Error (MSE) refers to the mean squared error between the last state of a rollout predicted by the simulator autoregressively and the actual ground truth state.

Full Rollout Mean MSE refers to the average mean squared error between the predicted states by the simulator and the ground truth state throughout the whole trajectory.

1-Step Mean MSE refers to the average mean squared error between the ground truth state at $t+1$ and predicted state by the simulator, given the ground truth state at time step t.

6.1.4 Training Techniques

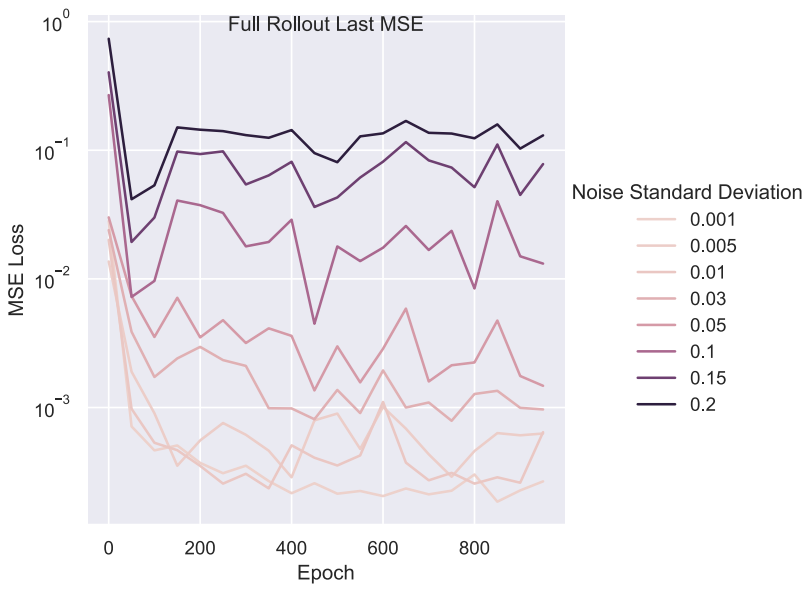


Figure 6.1: Evaluation of Rollout Stability of Simulators Trained with Different Noise variances

Training with Gaussian Noise

We discussed training with Gaussian noise in section 3.1, and we won't discuss it further. However, one question remains unanswered: what value should we choose for the fixed variance of the Gaussian distribution from which we draw samples? To address this question, we conducted experiments with different values for the standard deviation of the Gaussian noise added to the position of the mesh nodes. You can see the Full Rollout Last MSE results for the experiments with different noise variances in Figure 6.1. As the figure shows, $\sigma = 0.001$ seems to be the best performing σ value for the training with Gaussian noise and hence we will be using $\sigma = 0.001$ for the experiments in Section 6.2.

Buffer-Enhanced Training

We use two different sampling strategies in our buffer-enhanced training approach, as described in Chapter 4. One strategy is sampling from the buffer with a fixed probability, the other one is sampling from the buffer with a gradually increasing probability as the epoch number progresses. We refer to the former as **Training with Buffer - Constant SP** and the latter as **Training with Buffer - Increasing SP**.

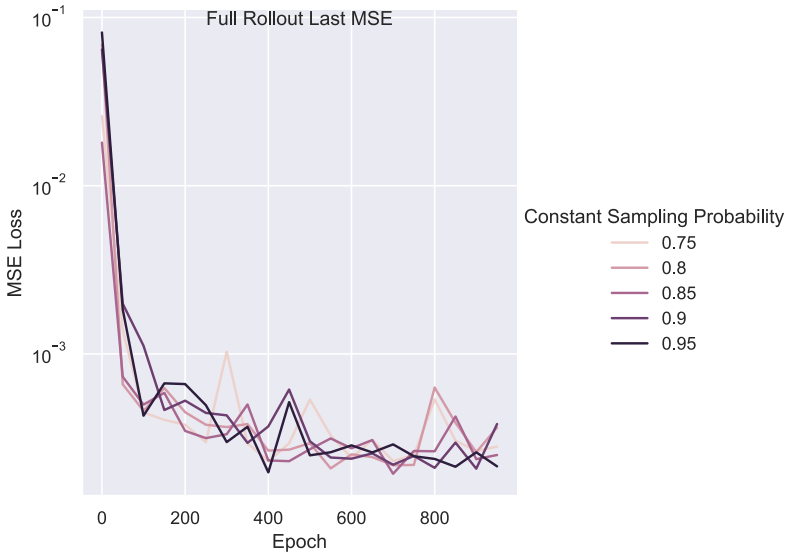


Figure 6.2: Evaluation of various sampling probabilities for Training with Buffer - Constant SP

In our experiments aimed at determining an optimal sampling probability value for the Training with Buffer - Constant SP method, we found that adjusting the sampling probability did not lead to substantial changes in the performance measured by Full

Rollout Last MSE as you can see in the Figure 6.2. Therefore, we concluded that a value of $p = 0.75$ is a reasonable choice to assess our approach.

For the Training with Buffer - Increasing SP method, we conducted experiments involving various hyperparameters, including the "maximum number of forward passes", "buffer sizes" ,"initial sampling probabilities" and "maximum sampling probabilites". We observed that modifying these hyperparameters did not yield significant alterations in the Full Rollout Last MSE metric. Consequently, we decided to set the maximum number of forward passes to 7, the buffer size to 1000, the initial sampling probability to 0.25, and the maximum sampling probability to 0.75 for the experiments detailed in Section 6.2.

Multi-Step Training

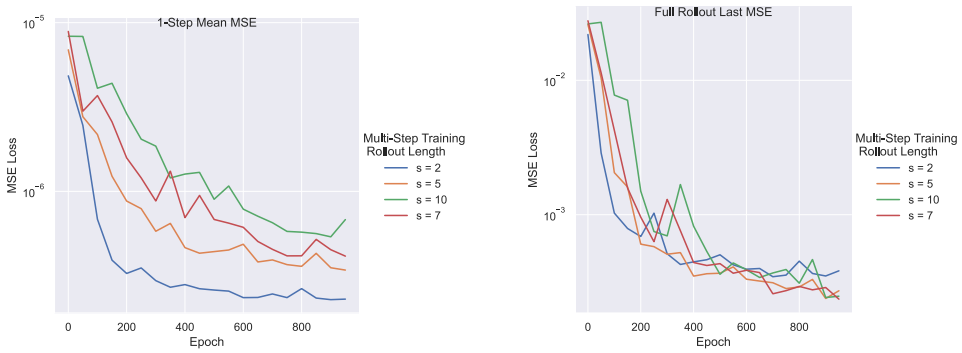


Figure 6.3: Evaluation of simulators trained in multi-step fashion with various hyperparameter values as described in Section 3.3

There is a single hyperparameter, s, that must be established for multi-step training as detailed in Section 3.3. This hyperparameter s specifies the number of time steps for unrolling the simulator from an initial state. Shi et al. suggest that " $s = 2$ is sufficient for achieving accuracy during inference-time prediction".[26] We test this assertion in our experiments, as depicted in Figure 6.3 . Our findings also indicate that multi-step training with a rollout length exceeding 2 does not significantly improve the performance of rollout stability, as measured by Full Rollout Last MSE. However, given that our buffer-enhanced approach can accommodate states predicted up to 7 time steps into the future, we choose $s = 7$ as the value for the hyperparameter s, which dictates the number of time steps for unrolling the simulator from a given state, in our multi-step training for the sake of evenness.

6.2 Results

In this section, we share the results from our experiments on the 2D Deformable Plate and 3D Tissue Manipulation datasets. Since it's easier to get high accuracy on the 2D Deformable Plate than on the 3D Tissue Manipulation, we start with results from the 2D Deformable Plate dataset. In all experiments, we keep the simulator architecture and the number of epochs the same but vary the training method. We use 8 different seeds for eacch training method.

For training methods that involve complex data usage, like Training with Buffer, deciding on the number of batches per epoch is not straightforward. The batch size is 32 for all experiments, but the batches' temporal dimension varies with the training method. To make sure all experiments are comparable, we adjust the number of batches per epoch based on the training method. This way, we use the same number of backward passes in all experiments.

6.2.1 2D Deformable Plate

The learning curves presented in Figure 6.5 illustrate the training loss and evaluation metrics for experiments utilizing various training methods. The Full Rollout Last MSE and Full Rollout Mean MSE plots indicate that all methods significantly outperform the basic 1-step training approach without noise, which is represented by the blue color in the plots. Unexpectedly, Training with Gaussian Noise demonstrates the most effective performance in terms of rollout stability, as assessed by the Full Rollout Last MSE and Full Rollout Mean MSE metrics, despite its straightforward approach. We suggest that the success of Training with Gaussian Noise as the top-performing training method may be due to the simpler nature of the 2D Deformable Plate task.

Another observation from Figure 6.5 is the discrepancy between the training loss and the 1-Step Mean MSE loss in Training with Pushforward Trick. This discrepancy might indicate that the simulator overfits on 1-step predictions, which, however, does not appear to impact the rollout stability as measured by Full Rollout Last MSE and Full Rollout Mean MSE. To understand why Training with Pushforward Trick shows this specific training behavior, further research is required. We can additionally observe that, there isn't any significant performance difference between Training with Buffer - Increasing SP and Training with Buffer - Constant SP. Also, the slower convergence of the training loss in Multi-Step Training is expected, as the training task is more complex due to the aim of minimizing the mean squared error over several steps. Figure 6.4 shows some example trajectories predicted by simulators trained with different training methods. It is easy to observe, that the naive training is really unstable and Training with Gaussian Noise and Training with Buffer - Constant SP yield similar good qualitative results.

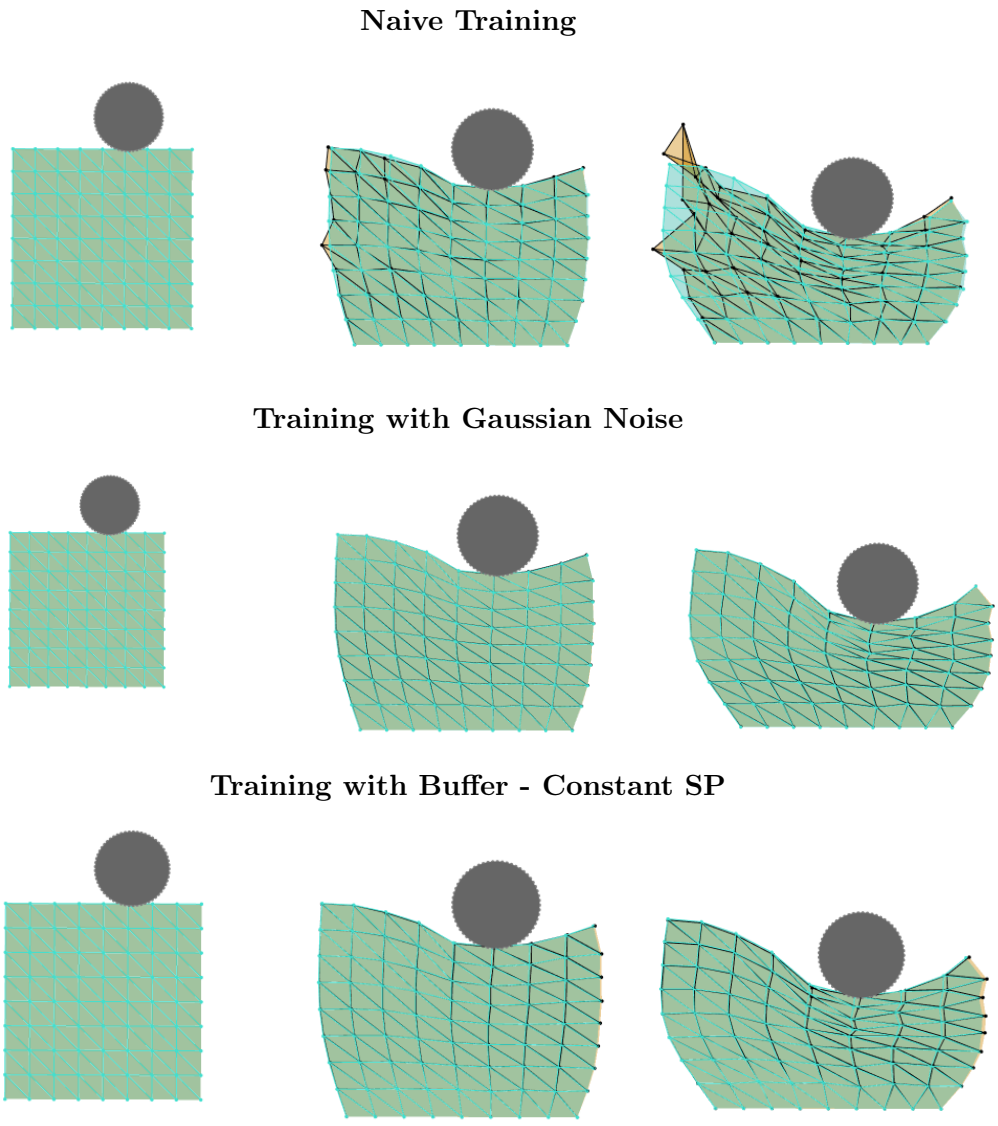


Figure 6.4: Example qualitative results from our experiments on 2D Deformable Plate dataset with different training methods. The transparent green meshes denote the ground truth, while the orange meshes illustrate predictions made by the simulator when trained with the respective methods. The temporal progression is displayed from left to right, starting with the initial state at time step $t = 0$, followed by an intermediate state at $t = 25$, and concluding with the final state at $t = 51$.

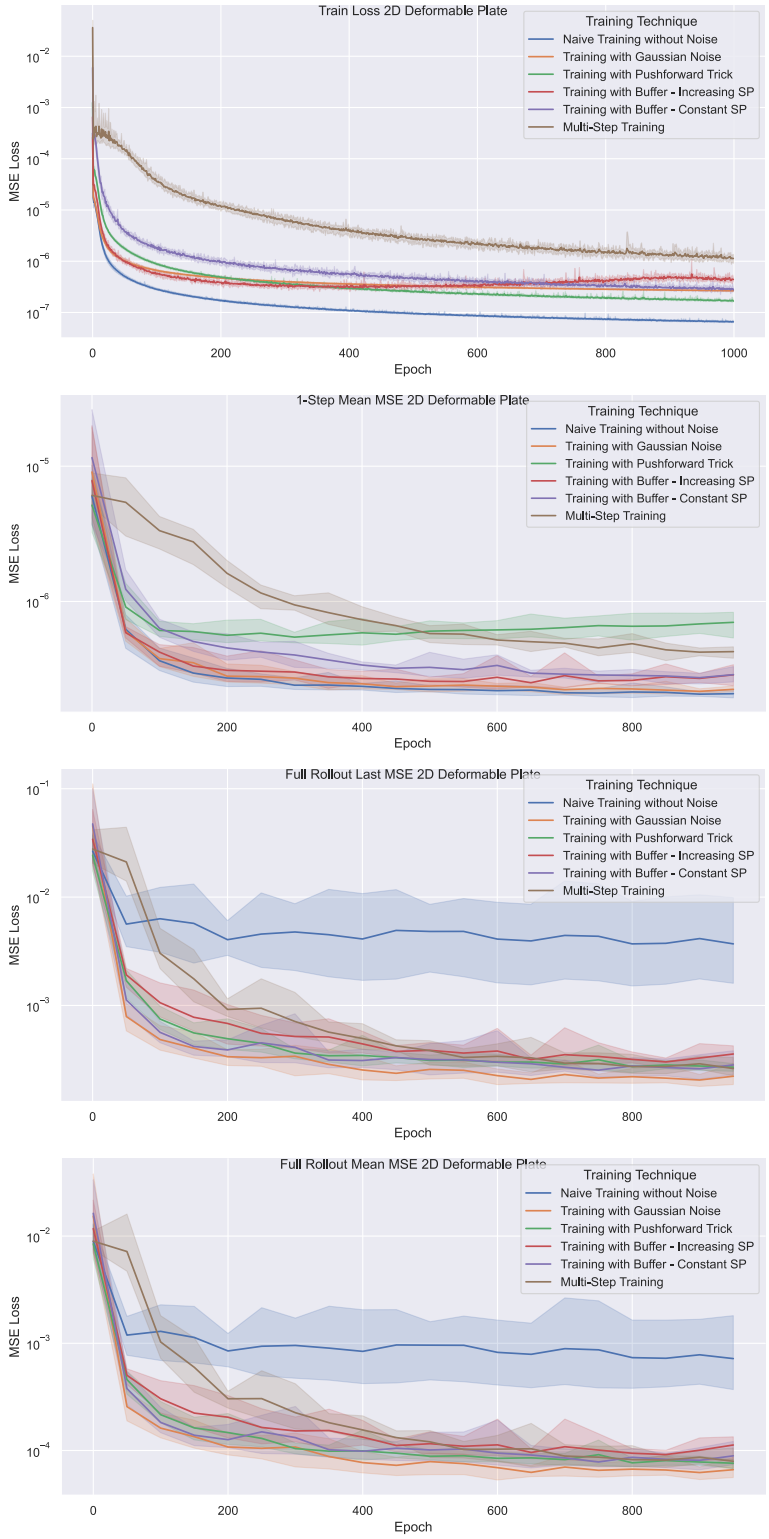


Figure 6.5: Training and evaluation loss curves of simulators trained with various training techniques on the 2D Deformable Plate Dataset

6.2.2 3D Tissue Manipulation

The 3D Tissue Manipulation dataset poses a greater challenge compared to the 2D Deformable Plate dataset due to its higher dimensional data and the increased complexity of system dynamics in 3D space versus 2D space. This higher level of difficulty is evident from the larger absolute MSE values observed in the Full Rollout Mean MSE plot for the 3D Tissue Manipulation task, as illustrated in Figure 6.7.

As in the 2D Deformable Plate dataset, all training methods also significantly outperform the basic 1-step training approach without noise on 3D Tissue Manipulation, represented by the blue color in the plots in Figure 6.7, in terms of rollout stability measured by Full Rollout Last MSE and Full Rollout Mean MSE metrics. Unlike in the 2D setting, Training with Gaussian Noise does not emerge as the best performing training method in the 3D setting. This supports our hypothesis that the success of Training with Gaussian Noise as the top-performing method on the 2D Deformable Plate dataset may be attributed to the simpler nature of the task. We also do not observe any overfitting pattern from the Training with Pushforward Trick as in the 2D setting.

Multi-Step Training and Training with Buffer - Constant SP appear to yield the best results, with Training with Buffer - Constant SP achieving faster convergence due to its simpler objective compared to Multi-Step Training. Both methods exhibit similar performance in terms of Full Rollout Last/Mean MSE on both datasets, as depicted in Figure 6.5 and Figure 6.7. There is no significant performance difference between Training with Buffer - Constant SP and Training with Buffer - Increasing SP; however, Training with Buffer - Constant SP marginally surpasses Training with Buffer - Increasing SP on both datasets.

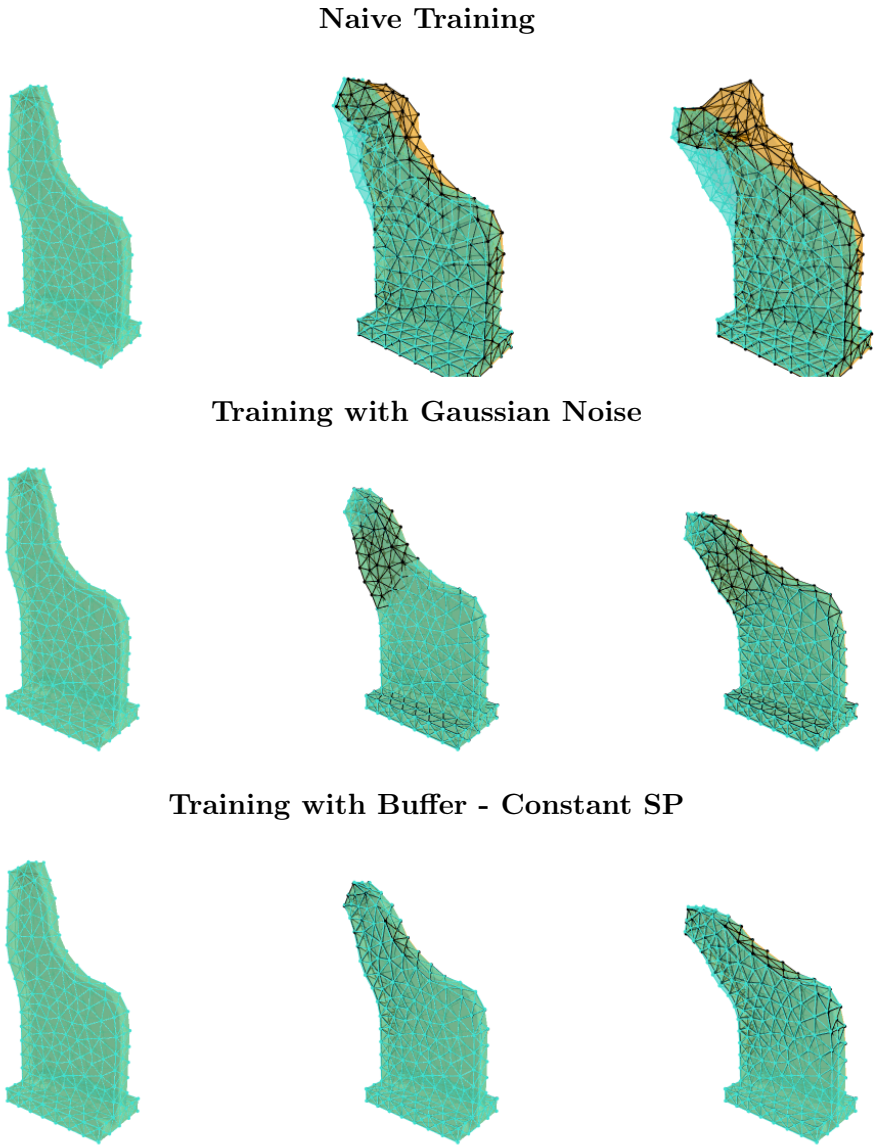


Figure 6.6: Example qualitative results from our experiments on 3D Tissue Manipulation dataset with different training methods. The transparent green meshes denote the ground truth, while the orange meshes illustrate predictions made by the simulator when trained with the respective methods. The temporal progression is displayed from left to right, starting with the initial state at time step $t = 0$, followed by an intermediate state at $t = 25$, and concluding with the final state at $t = 51$.

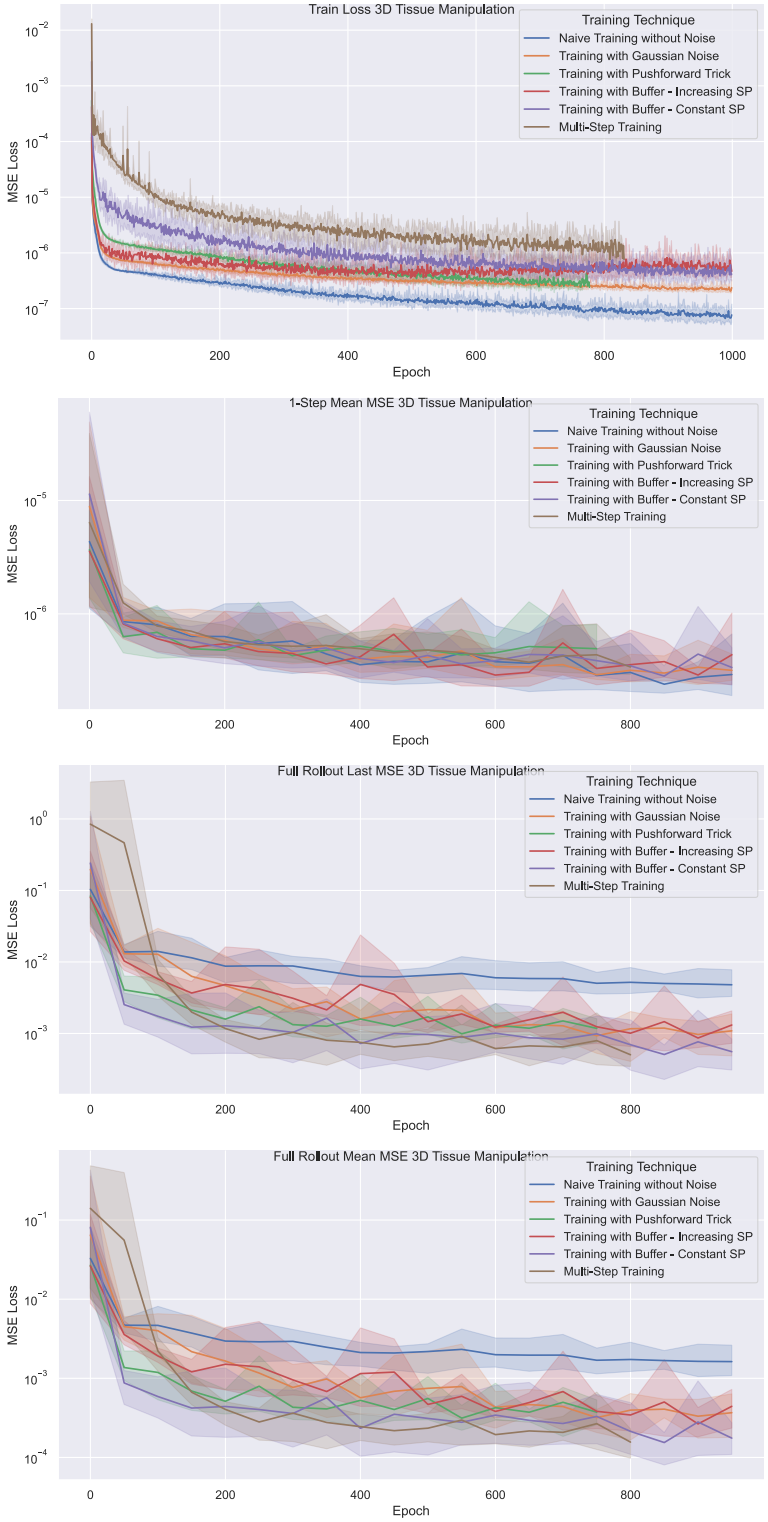


Figure 6.7: Training and evaluation loss curves of simulators trained with various training techniques on the 3D Tissue Manipulation Dataset

Chapter 7

Conclusion and Future Work

In this chapter, we provide a summary of the key insights obtained from the experiments we conducted and outline valuable directions for future research

7.1 Conclusion

In this thesis, we addressed the challenge of ensuring long-horizon rollout stability in autoregressive neural simulators, focusing on GNS. This challenge arises from the discrepancy between how simulators are trained, typically in a one-step manner, and how they are employed during inference, where they are used autoregressively to predict trajectories. Error accumulation and distribution shift during autoregressive inference causes simulator to fail in long-horizon predictions.

To address this issue, we explored several strategies in our work, concentrating on three main approaches: the introduction of Gaussian noise to the training data, the pushforward trick, and multi-step training. Both Gaussian noise and the pushforward trick introduce noise into the training process, although they differ in the nature of the noise introduced. Specifically, one method involves adding Gaussian noise directly to the training data, while the other incorporates noise that is induced by the model itself. We investigate these training methods by conducting comparative experiments.

Additionally, we present a novel training framework utilizing a buffer in the training loop, inspired by the use of replay buffers in RL. We call this framework "buffer-enhanced training". Buffer-enhanced training introduces a buffer into the training,

which stores predictions of the simulator to be used as additional training data. We compare the buffer-enhanced training to other training methods as well. We conclude, that when the dataset does not particularly pose a challenge and relatively easy to learn, such as deformation of a material in 2D space, introducing Gaussian noise into the training data is a good choice and more advanced methods do not provide any additional advantages. In more challenging tasks, such as predicting how a tissue will behave in 3D space when being manipulated by a force, the choice of training method matters more. Our buffer-enhanced training methods perform better than Gaussian noise and Pushforward Trick on average and on-par with multi-step training. We also believe that, adding a buffer to the training loop can offer researchers greater flexibility and control over the training procedure and existing literature on replay buffers can provide a rich foundation for exploring techniques to optimize a training procedure with a buffer in loop.

7.2 Future Work

Moving forward, there's a lot more work to be done based on what we've learned so far. Below, we outline several promising directions for future research.

While we aim to compare various training methods, we believe our experiments are not extensive enough to definitively conclude which method is superior overall. Therefore, we suggest that a more thorough comparison across a broader range of tasks is required to determine the most effective training approach for specific scenarios. Furthermore, the development of more robust evaluation metrics and benchmarks specifically designed for assessing long-horizon stability in autoregressive neural simulators could significantly aid in the progress of this field. Current metrics like MSE offer a broad measure of performance but may not capture all aspects relevant to stability or practical applicability. Developing more nuanced metrics or comprehensive benchmark challenges could help to better quantify improvements and guide future research efforts. Additionally, existing research on the long-horizon stability of autoregressive neural simulators is largely empirical. We believe that future studies should aim to ground this issue in a more solid theoretical framework.

Moreover, our buffer-enhanced training method has potential for further improvement through the adoption of more sophisticated sampling processes or buffer configurations. For example, prioritized experience replay [24], a concept borrowed from RL, where more 'important' experiences are sampled more frequently, could be adapted and evaluated for its efficacy in training neural simulators. This concept raises the question of how 'importance' could be defined and measured within the context of autoregressive neural simulators and whether such an approach could lead to faster convergence or better long-term stability.

Bibliography

[1] A. Al-Saadi, K. Al-Farhany, A. E. Faisal, M. A. Alomari, W. Jamshed, M. R. Eid, E. S. M. Tag El Din, and A. Amjad. Improvement of the aerodynamic behaviour of the passenger car by using a combine of ditch and base bleed. *Scientific Reports*, 12(1), Nov. 2022. ISSN 2045-2322. doi: 10.1038/s41598-022-23183-z. URL http://dx.doi.org/10.1038/s41598-022-23183-z.

[2] and Karlton J. Hickey and S. T. Xiao. Finite element modeling and simulation of car crash. 2017. URL https://api.semanticscholar.org/CorpusID:55542348.

[3] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

[4] P. Battaglia, J. B. C. Hamrick, V. Bapst, A. Sanchez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. J. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, 2018. URL https://arxiv.org/pdf/1806.01261.pdf.

[5] P. W. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, and K. Kavukcuoglu. Interaction networks for learning about objects, relations and physics. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4502–4510, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/3147da8ab4a0437c15ef51a5cc7f2dc4-Abstract.html.

[6] A. Bou, M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, G. D. Fabritiis, and V. Moens. Torchrl: A data-driven decision-making library for pytorch. *CoRR*, abs/2306.00577, 2023. doi: 10.48550/ARXIV.2306.00577. URL https://doi.org/10.48550/arXiv.2306.00577.

[7] J. Brandstetter, D. E. Worrall, and M. Welling. Message passing neural PDE solvers. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=vSix3HPYKSU.

[8] F. Faure, C. Duriez, H. Delingette, J. Allard, B. Gilles, S. Marchesseau, H. Talbot, H. Courtecuisse, G. Bousquet, I. Peterlik, and S. Cotin. SOFA: A Multi-Model Framework for Interactive Physical Simulation. In Y. Payan, editor, *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*, volume 11 of *Studies in Mechanobiology, Tissue Engineering and Biomaterials*, pages 283–321. Springer, June 2012. doi: 10.1007/8415_2012_125. URL https://inria.hal.science/hal-00681539.

[9] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[10] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 2017. URL http://proceedings.mlr.press/v70/gilmer17a.html.

[11] R. Kannan, F. Marinacci, M. Vogelsberger, L. V. Sales, P. Torrey, V. Springel, and L. Hernquist. Simulating the interstellar medium of galaxies with radiative transfer, non-equilibrium thermochemistry, and dust. *Monthly Notices of the Royal Astronomical Society*, 499(4):5732–5748, 10 2020. ISSN 0035-8711. doi: 10.1093/mnras/staa3249. URL https://doi.org/10.1093/mnras/staa3249.

[12] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[13] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012. URL https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[15] R. Lakes and K. W. Wojciechowski. Negative compressibility, negative poisson's ratio, and stability. *physica status solidi (b)*, 245(3):545–551, 2008. doi: https://doi.org/10.1002/pssb.200777708. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/pssb.200777708.

[16] G. Li, M. Müller, A. K. Thabet, and B. Ghanem. Deepgcns: Can gcns go as deep as cnns? In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 9266–9275. IEEE, 2019. doi: 10.1109/ICCV.2019.00936. URL https://doi.org/10.1109/ICCV.2019.00936.

[17] L.-J. Lin. *Reinforcement learning for robots using neural networks*. PhD thesis, USA, 1992. UMI Order No. GAX93-22750.

[18] L. J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8:293–321, 1992. doi: 10.1007/BF00992699. URL https://doi.org/10.1007/BF00992699.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[20] A. R. Oganov, C. J. Pickard, Q. Zhu, and R. J. Needs. Structure prediction drives materials discovery. *Nature Reviews Materials*, 4(5):331–348, Apr. 2019. ISSN 2058-8437. doi: 10.1038/s41578-019-0101-8. URL http://dx.doi.org/10.1038/s41578-019-0101-8.

[21] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning mesh-based simulation with graph networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=roNqYL0_XP.

[22] J. N. Reddy. *An introduction to the finite element method*. McGraw Hill Higher Education, Maidenhead, England, 3 edition, May 2005.

[23] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia. Learning to simulate complex physics with graph networks. In *Pro-*

ceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, volume 119 of *Proceedings of Machine Learning Research*, pages 8459–8468. PMLR, 2020. URL http://proceedings.mlr.press/v119/sanchez-gonzalez20a.html.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In Y. Bengio and Y. LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05952.

[25] K. Sharma and P. Korn. *Numerical Simulation of an idealized coupled Ocean-Atmosphere Climate Model*, page 113–130. Springer Nature Switzerland, 2023. ISBN 9783031451584. doi: 10.1007/978-3-031-45158-4_7. URL http://dx.doi.org/10.1007/978-3-031-45158-4_7.

[26] H. Shi, H. Xu, S. Clarke, Y. Li, and J. Wu. Robocook: Long-horizon elasto-plastic object manipulation with diverse tools. In J. Tan, M. Toussaint, and K. Darvish, editors, *Conference on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA*, volume 229 of *Proceedings of Machine Learning Research*, pages 642–660. PMLR, 2023. URL https://proceedings.mlr.press/v229/shi23a.html.

[27] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

[28] Z. Zheng-hua, W. Yu-huan, L. Quan, Y. Xiao-tao, and Y. Cheng. A varying time-step explicit numerical integration algorithm for solving motion equation. *Acta Seismologica Sinica*, 18:239–244, 2005. doi: 10.1007/S11589-005-0071-3.

[29] O. C. Zienkiewicz and R. L. Taylor. *The finite element method for solid and structural mechanics*. Butterworth-Heinemann, Woburn, MA, 6 edition, May 2014.

Acronyms

CPU Central Processing Unit. 2

FEM Finite Element Method. 7

GN Graph Network. 10

GNN Graph Neural Network. 8

GNS Graph-Network-based Simulators. 2, 9, 12, 33

GPU Graphics Processing Unit. 2

MGN MeshGraphNet. 3, 12

MLP Multi Layer Perceptron. 21, 22

MPNN Message Passing Neural Network. 8, 9, 21

MSE Mean Squared Error. 23, 29, 34

PDE Partial Differential Equation. 7

ReLU Rectified Linear Unit. 22

RL Reinforcement Learning. 15–17, 33, 34

SOFA Simulation Open Framework Architecture. 19, 20