Karlsruhe Institute of Technology

Autonomous Learning Robots Lab

RippleNet: A Remote Message Passing Method For Graph Neural Networks

Master's Thesis
of

Ruoheng Ma

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

Referee: Prof. Dr. Techn. Gerhard Neumann

Advisor: M.Sc. Niklas Jordi Freymuth

Duration: September 1st, 2021 — March 01th, 2022

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 01. März 2022

Ruoheng Ma

Zusammenfassung

Meshes werden häufig in Simulatoren für komplexe physikalische Systeme verwendet, um partielle Differentialgleichungen zu lösen. Sie können qualitativ hochwertige Ergebnisse liefern, wenn die Rechenressourcen und Zeit ausreichend sind. Es stehen jedoch nicht immer genügend Rechenressourcen und Zeit zur Verfügung. Manchmal ist ein schnelles Simulationsergebnis mit ausreichender Qualität eine bessere Kompromisslösung, beispielsweise in einem Prototyping-Prozess. Daneben erfordern Simulatoren jahrelange Entwicklungssarbeit und Arbeit der Feinabstimmung, um hochqualitative Ausgaben zu liefern, was für Laien nicht freundlich ist.

Inzwischen zeigen Deep-Learning-Algorithmen ihre Stand der Technik in Bereichen wie Bilderkennung, Videoverarbeitung und Verarbeitung natürlicher Sprache. Graph Neural Networks (GNNs) sind eine erweiterte Deep-Learning-Methode, die in nicht-euklidischen Datenstrukturen wie z.B. einem Graphen eingesetzt werden kann. Um die Vorteile von Deep Learning im Bereich der physikalischen Simulation zu nutzen, wurde das MeshGraphNets (MGN)[1] entwickelt. Es verarbeitet Graphdaten, die Knoten und Kanten eines Meshes enthalten. Im Vergleich zu herkömmlichen Simulatoren liefert es effizientere Simulationen, während die Simulationsgenauigkeit gleichzeitig beibehalten wird.

Trotz der Effizienz und Genauigkeit von MGN verlangt es eine lange Trainingszeit, bevor es tatsächlich eingesetzt werden kann. Daher stellen wir in dieser Arbeit eine neue Architektur, das Ripple Network (RN), vor, um den Trainingsprozess von MGN zu beschleunigen und gleichzeitig die Vorhersagegenauigkeit des Modells beizubehalten. Außerdem haben wir die anderen *Aggregationsmethoden* für die RN untersucht. Die Aggregationsmethode ist eine Funktion, die bestimmt, wie Nachbarschaftsinformationen aggregiert werden, um den Nachbarkontext für einen Knoten in einem Graph Neural Network (GNN)-Modell zu bilden. Übliche Aggregationsverfahren sind Summierung, Minimum usw. Verschiedene Aggregationsverfahren können verschiedene versteckte Merkmale der Nachbarn ausnutzen. Darüber hinaus wird RN mit *attention* erweitert, das Gewichtungen für alle Nachbarn eines Knotens bereitstellt, sodass das GNN-Modell weiß, welche Nachbarn beim Aktualisieren

von Merkmalen der aktuelle Knoten als wichtige einflussreiche Nachbarn betrachtet werden sollten. Wir führen umfassende Experimente mit verschiedenen Kombinationen dieser Designentscheidungen durch, um ihre Auswirkungen auf die Qualität der physikalischen Simulation zu demonstrieren. Wir stellen fest, dass das MGN-Modell mit Principal Neighbourhood Aggregation (PNA) und das RN die Leistung auf dem neuesten Stand der Technik zeigen oder sogar das ursprüngliche MGN-Modell überwältigen kann.

Abstract

Complex physical system simulators often utilize mesh representations to solve partial differential equations. They can deliver high-quality results when the computing resource and time are sufficient. However, sufficient computing resource and time are not always available. Sometimes quick simulation result with just enough-quality is a better trade-off solution, such as in a prototyping process. Besides, such high quality simulators require years of engineering to be developed and to be tuned, which is not friendly to non-experts.

In the meanwhile, deep learning algorithms are exhibiting their state of the art performance in fields such as image recognition, video processing and natural language processing. Graph Neural Networks (GNNs) is an extended deep learning method which can be employed in non-Euclidean data structure, such as graph. To exploit the benefits of deep learning in the field of physical simulation, the MeshGraphNets (MGN)[1] has been proposed to process graph data which contains nodes and edges of a mesh. Compared to traditional simulators, it delivers more efficient simulations while preserving simulation accuracy.

Despite MGN's efficiency and accuracy, it takes a long time to train the network before it can be actually employed. Thus, in this thesis, we have proposed a new architecture, the Ripple Network (RN), to accelerate the training process of MGN, while maintaining the model's prediction accuracy. In addition, we have explored other *aggregation* methods for the RN. Aggregation method is a function that determines how neighborhood information is aggregated to form the neighboring context for a node in a Graph Neural Network (GNN) model. Common aggregation methods are summation, minimum and etc. Different aggregation methods can exploit different hidden features of the neighbors. Moreover, the RN is augmented with *attention*, which provides weights for all neighbors of a node, so that the GNN model knows which neighbors should be considered as important influential neighbors when updating features of the current node. We perform exhaustive experiments on different combinations of these design choices to showcase their effect on the quality of the physical simulation. We find that the the MGN model with Principal Neighbourhood Aggregation (PNA) and the RN can show state of the art performance or even overwhelm the original MGN model.

Table of Contents

Chapter 1

Introduction

Finite Element Method (FEM) is a popular numerical method to solve differential equations simulating a physical system in engineering. It discretes the physical domain into *mesh*, a collection of multiple subdomains, then approximates the original differential equations in each subdomain, and finally assembles the approximating equations together. FEM makes it easier to model and handle complex physical system with simple objects.

Even though FEM is efficient in solving differential equations describing a complex physical system, its calculation process is time- and resource-intensive. In addition, solvers and parameters of different models have to be tuned individually. Thus, it is desirable to figure out another method to simulate the physical system.

In the past decade, deep learning has achieved great success in various fields, for example speech recognition, image recognition, natural language processing and so on. Due to the data explosion, as well as advancement of computing hardware (need reference) in the recent years, i.e. graphics processing unit (GPU), training deep neural network under an acceptable amount of time is possible, while the accuracy of the neural network increases because of the large amount of training data.

Moreover, the inference time of a deep neural network is short because the calculation inside the neural network does not include complex mathematical equations. Therefore, deep learning is becoming the most popular machine learning methodology in many researching fields.

Graph Neural Network (GNN) is a subclass of deep learning models and based on Feedforward Neural Network (FNN). In contrast to other conventional deep learning models, for example

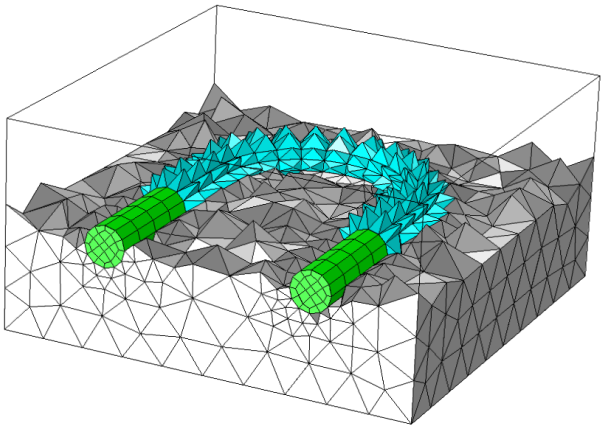


Figure 1.1: An example of mesh, generated by COSMOL.

Convolutional Neural Netwrok (CNN) and Recurrent Neural Network (RNN), GNN can handle data in non-Euclidean structure, for example a graph, more efficiently, owning to the equipped message passing/aggregation techniques. With message passing/aggregation, each node in a graph is able to collect information from its neighbors over a a specified distance and updates itself. Prediction about an individual node, an individual edge of even the whole graph can be made with GNN.

In mesh-based physical simulation, objects in the environment are represented with numerous simple geometric shapes, as shown in Figure 1.1[2]. The mesh can be interpreted as a graph, with vertices and edges of the simple geometric shapes interpreted as nodes and edges of the graph. Because of this similar structure of a mesh and a graph, using GNN to model the complex physical simulation is a feasible approach. Many previous works have made contribution to this topic, such as [3], [4], [5] and [1].

In this thesis, we propose a neural network architecture called *Ripple Network (RN)*, which can learn the underlying differential equations from the physical simulation generated by a physical simulator and produce a precise simulation afterwards. The RN is based on the previous work MeshGraphNets (MGN). It adds meta edges to the original graph, so that dynamic information can be propagated more efficiently. Moreover, different message passing methods and attention mechanism are added to RN to improve its performance.

This thesis is structured as follows: In chapter 2, fundamental knowledge and general background of the techniques that are related to the thesis are presented. This chapter focuses on giving an overview on FEM and GNN. Their relation is also explained in this chapter.

Then, related works are reviewed in chapter 3. The related works are grouped into three groups according to their relevance to the proposed RN. In the first group, classical GNN frameworks for handling graph-based data structure are presented. In the second group, previous GNNs concentrating on physical simulation are shown. In the last group, MGN, which is the base of the proposed RN, is reviewed.

After that, the architecture of RN is shown in chapter 4. The RN distinguishes itself from MGN by grouping nodes in the graph into multiple *ripples* and adding extra edges between ripples to propagate information. In addition, this chapter shows how the RN is equipped with different message passing/aggregation methods and attention mechanism.

In chapter 5, evaluation results of RN of the task *FlagSimple* are presented. It can be seen that the MGN model with Principal Neighbourhood Aggregation (PNA) and the RN can show state of the art performance or even overwhelm the original MGN model.

At the end, conclusion about the RN is drawn and possible future work are presented in chapter 6.

Chapter 2

Preliminaries

This thesis aims to develop a Graph Neural Network (GNN)-based deep learning framework to learn Finite Element Method (FEM)-based physical simulation in different domains. To make the thesis better understandable, fundamental knowledge and general background of the techniques used in the thesis are clarified in this chapter.

First, the working way of the popular FEM and its relation to Neural Network (NN) is explained in the first section. In the second section, an overview of three fundamental classes of NN and their working principles are presented. Then, NN class that can handle non-Euclidean data structure and its instance, GNN, are shown in section 2.3 and 2.4.

2.1 Finite Element Method And Neural Network

In engineering, differential equations are used to describe a system model. As it is difficult to find out the analytical solution for the system model, FEM is deployed to figure out the numerical solution. For example, in a car crash simulation, FEM is used to simulate the crash of a driving car against a lighting column[6], as shown in Figure 2.1.

Coarsely, FEM functions in three steps:

1. **Domain representation** The concerned geometric region of the system model, the *domain* (Figure 2.2a), is represented as a mesh (Figure 2.2b), which is a collection of simple *subdomains* called *finite element*. The subdomains can be triangles and quadrilateral elements, as shown in Figure 2.2c.

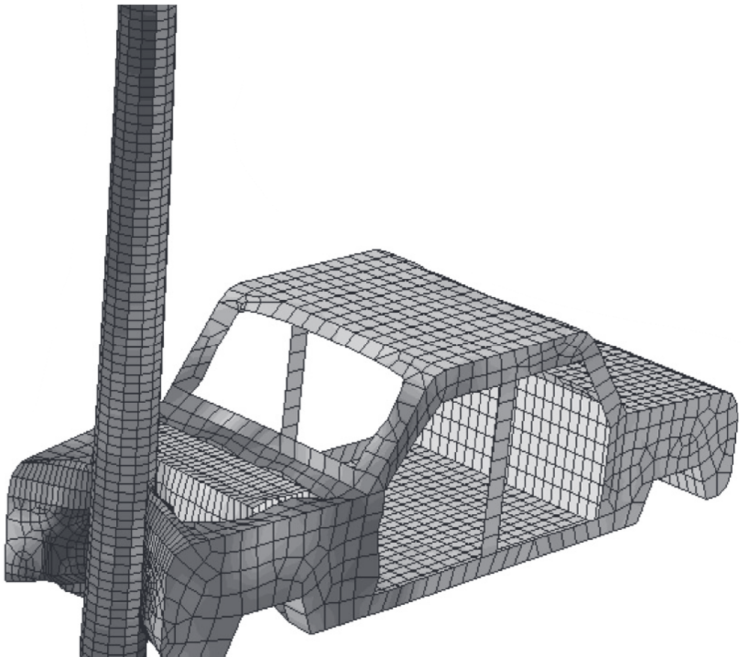
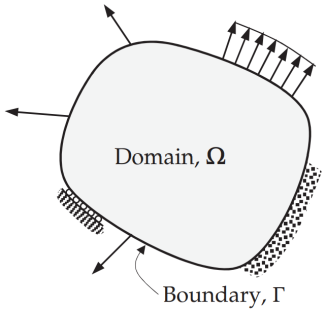


Figure 2.1: A car crashing against a lighting column, presented in a mesh.

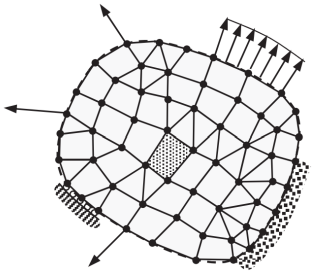
2. **Approximation** In each subdomain, the equations modelling the concerned system are approximated to find out the algebraic relation between the values of the duality pairs, i.e. cause and effect. The set of resulting algebraic equations among the nodal values of the duality pairs (e.g., displacements and forces) is termed a *finite element model*.

3. **Assembly** The approximated equations of all subdomains are assembled to form the approximation of the domain.

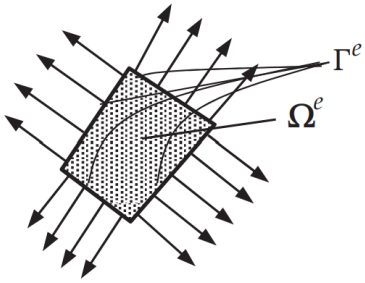
When approximating the system equations in subdomains, the idea that any continous functions can be represented by a linear combination of other functions is deployed. This can be described by a function $u \approx u_h = \sum c_i \phi_i$, where u is the function being approximated, u_h is the approximation function, ϕ_i is the function to approximate u and c_i is the undetermined coefficient. Usually, ϕ_i is chosen to be polynomials derived from interpolation theory. When the polynomials have a limited degree, then they can be learned by two-layer neural networks using gradient descent to update their parameters according to [8]. In the past decade, since the computational power of graphics processing unit (GPU) has increased dramatically so that the training of large neural networks can be accelerated, it is possible to use neural networks to learn the underlying approximation function and rely on neural networks instead of on FEM when simulating physical systems.



(a) Domain Ω with boundary Γ.



(b) Domain Ω is represented as a mesh containing triangles and quadrilateral elements.



(c) Subdomain Ω^e with boundary Γ^e.

Figure 2.2: Discretization of a two-dimensional domain[7].

2.2 Deep Learning

Deep learning is a sub-field of representation learning, which allows the machine to learn internal representation from the raw input data, while conventional techniques in machine learning can only complete regression or classification tasks by taking output of a predefined feature extractor as input. Deep learning methods stack numerous non-linear modules, i.e. *artificial neuron* and *activation* function like sigmoid function or Rectified Linear Unit (ReLU), to extract features and learn the underlying functions. In each level of a deep learning architecture, an more abstract internal representation is extracted from the output of the previous level. The learnable parameters in the modules of the architecure can tuned themselves through *back-propogation* of the difference between the ground truth and the predicted result.

2.2.1 Feedforward Neural Network

A Feedforward Neural Network (FNN) is a neural network that does not form a cycle in its topological structure. The input of the FNN is fed into input layer, then through multiple hidden layers and finally transfered to the output layer, as shown in Figure 2.3[9].

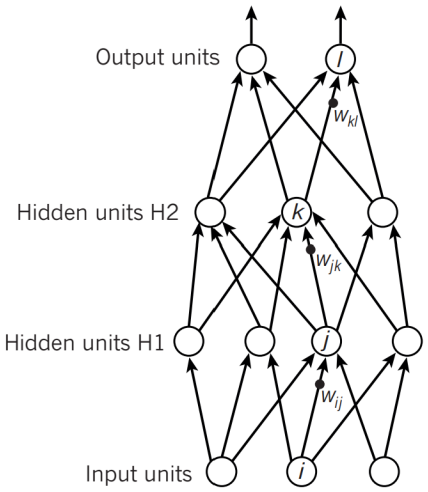


Figure 2.3: FNN with two hidden layers.

FNN contains artificial neurons and activation functions as basic components. An artificial neuron's operation can be defined as the following function:

$$y = \varphi\left(\sum_{j=0}^{m} w_j x_j\right),$$

where y is the output of the neuron, x_j is one of the input signal, w_{kj} is the weight that multiplies with the k-th input signal and φ is the activation function, for example the sigmoid function or ReLU.

2.2.2 Convolutional Neural Network

To handle input data of multiple arrays at the same time, for example, an image containing three channels of 2D array, Convolutional Neural Netwrok (CNN) is proposed. A CNN model consists of two important components that are not presented in conventional FNN, which are the convolutional layer and pooling layer[10]. A convolutional layer is a network layer containing kernel, which are trainable parameters in grid format. The convolutional layer performs Multiply–Accumulate (MAC) operation between the kernel and a region of the input data with the same size of the kernel, as depicted in Figure 2.4[10].

A pooling layer is responsible for downsampling the feature map in order to introduce a translation invariance to small shifts and distortions, and decrease the number of subsequent learnable parameters[10]. Figure 2.5[10] shows an example of max pooling. Another pooling filter that is commonly used is average pooling, which calculates the average of values in the input region rather than figuring out the maximum value.

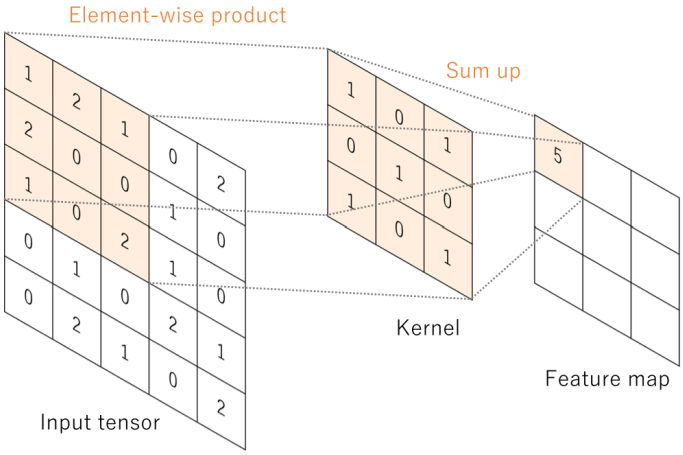


Figure 2.4: Convolution performed between a 5x5 image and a 3x3 kernel outputs an element in the feature map.

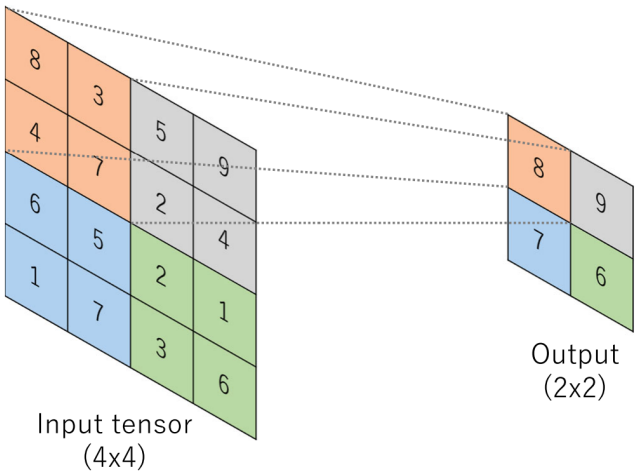


Figure 2.5: Max pooling with a 2x2 filter. The filtered region of the input and its corresponding output are colored the same.

2.2.3 Recurrent neural network

Recurrent Neural Network (RNN) is a type of neural network that is suitable for handling input data of sequential format, such as speech and language. A RNN model finishes processing an input of an input sequence at a time. In comparison to other neural network models, RNN maintains the *state* of hidden units and feeds them into the next level hidden unit as an additional input, as shown in Figure 2.6[9]. The topological structure of RNN helps to process input data with knowledge of the past history of the input sequence, so that RNN can better deal with tasks like speech recognition or machine translation.

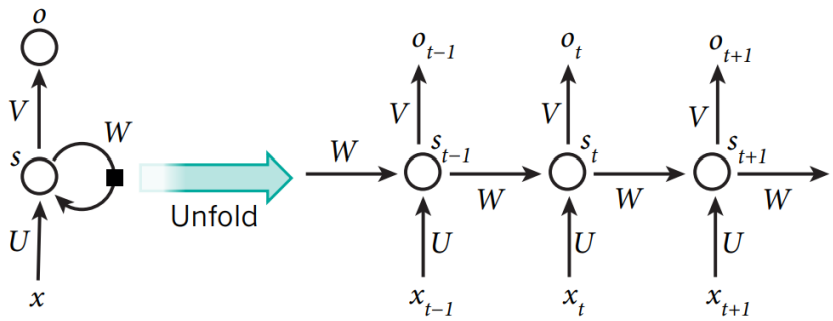


Figure 2.6: A RNN model(left) and its unfold version(right). x is input data, s is the hidden state, o is the output, and U, V, W are the trainable parameters. t in the unfold version indicates the time step.

2.3 Geometric Deep Learning

As presented in the previous section, deep learning has prove its strong performance in many application fields, i.e. speech recognition and computer vision, there are some tasks that deep learning can not deal with. For example, in a recommendation system, the relation between customers and products is presented as graph, which is difficult for deep learning algorithms to exploit useful information between customers and products to make accurate recommendation. Another example is physical simulation. In complex physical simulation, objects are usually represented with mesh, which is a collection of vertices, edges and faces. The graph-like mesh structure makes it hard for deep learning algorithms to learn the properties of the vertices or edges.

The reason for such problems is that deep learning algorithms can not handle data presented in non-Euclidean sturcture, i.e. a graph, properly. As specified in [11], the data presented in non-Euclidean sturcture does not possess nature that data in Euclidean sturcutre has, such as global parameterization, common system of coordinates, vector space structure, or shift invariance. As basic operations like convolution of current deep learning algorithms are intrinsically supposed to handle data of Euclidean structure, they can not be generalized well on non-Euclidean data structure.

In order to deal with non-Euclidean data structure, *geometric deep learning* is proposed. Geometric deep learning aims to port current deep learning models from Euclidean domain to non-Euclidean domain.

There are two classes of problems[11] that need to be solved when dealing with data in non-Euclidean structure. The first problem is to characterize the structure of the data. In this problem, a set of data points with high-dimensional features are given as input. The task of the algorithm is to reduce the dimensionality of the data and extract the low-dimensional structure information of the data that is embedded in the high-dimensional features. Such process is also called *manifold learning* or *nonlinear dimensionality reduction*.

There are two common steps that many methods share when handling this task. The first step is to construct a representation of local affinity of the data points, which is often a sparsely connected graph. The second step is to embed the data points into a low-dimensional space, trying to preserve some criterion of the original affinity. Examples of nonlinear dimensionality reduction refer to [12], [13] and [14].

The second problem is to analyze the underlying functions based on the given data in non-Euclidean domain. This problem can be further divided into two subclasses of problems: in the first subclass, the domain is fixed. For example, given a graph of users of a social network with the history of the users' geographic coordinates, predicting the user's position.

In the second subclass, multiple domains are given as input. For instance, in computer graphics and vision applications, finding similarity and correspondence between shapes. In this case, the algorithm needs to handle multiple shapes at a time.

A significant instance of geometric deep learning is *graph neural network*. An overview of graph neural network is presented in the next section.

In geometric deep learning, one of the earliest attempt is [15].

2.4 Graph Neural Network

GNN is a set of neural network models that aim to solve graph-based learning problems. A GNN model takes a graph $G = (V, E)$, where V is the set of vertices or nodes and E is the set of edges, as input. The neighbor of a node is defined as $N(v) = \{u \in V \mid (v,u) \in E\}$. In addition, a graph may also contains node feature/label X and edge feature Y. The target of a GNN model is to exploit useful information from the given graph. In this section, details of GNN are presented.

2.4.1 Task of GNN

As specified in [16], the GNN model focuses mainly on solving the following three types of prediction task:

1. **Graph-level** Wenn dealing with graph-level prediction task, the GNN classifies the graph combined with pooling and readout operations.

2. **Edge-level** In edge-level task, the GNN classifies the edge, predicts the existence of nodes' connection or predicts the strength of connection of an edge.

3. **Node-level** Through information propogation or graph convolution, the GNN extracts the high-level node representation, then classifies the node or predicts the feature/label of a node, in an end-to-end manner. This task is often performed with multi-layer perceptrons and softmax layer.

2.4.2 Taxonomy of GNN

According to [16], GNN models can be categorized into four categories:

1. **Recurrent graph neural networks** Recurrent Graph Neural Network (RecGNN) is mostly the earliest GNN type. It is based on recurrent neural architectures. A node constantly exchange information with its neighboring nodes until a stable equilibrium is reached. For example, the following formulas[17] show the calculation of the hidden state and the output of the network:

$$x_n(t+1) = f_w\left(l_n, l_{\mathrm{co}[n]}, x_{\mathrm{ne}[n]}(t), l_{\mathrm{ne}[n]}\right)$$
$$o_n(t) = g_w\left(x_n(t), l_n\right), \quad n \in N,$$

   where x_n is the hidden state, O_n is the output, t is the current time step of the network, f_w and g_w are trainable parametric functions, l_n, $l_{\mathrm{co}[n]}$, $x_{\mathrm{ne}[n]}$, $l_{\mathrm{ne}[n]}$ are the label of n, the labels of its edges, the states, and the labels of the nodes in the neighborhood of n, respectively.

2. **Convolutional Graph Neural Networks** Convolutional Graph Neural Network (ConvGNN) generalizes the operation *convolution* from grid data to graph data. In ConvGNN, multiple convolution layers are stacked together to extract high-level node representation. Then, a node aggregate features of its neighbors and updates itself.

3. **Graph Autoencoders** Graph Autoencoder (GAE) is used as unsupervised learning framework to learn network embeddings and graph generative distributions. It operates in two steps: in the first step, it encodes a graph into latent space. In the second step, it decodes the latent space graph and regenerates a graph based on some criterion in a step-by-step or all-at-once manner. A typical implementation of GAE is Variational Graph Autoencoder (VGAE)[18], which is presented in the next section.

4. **Spatial-temporal Graph Neural Networks** Spatial-temporal Graph Neural Network (STGNN) is a special class of GNN which can handle spatial-temporal graphs. Spatial-temporal graph is a graph defined as $G^{(t)} = \left(\mathbf{V}, \mathbf{E}, \mathbf{X}^{(t)}\right)$ with $\mathbf{X}^{(t)} \in \mathbf{R}^{n \times d}$. $\mathbf{X}^{(t)}$ is the node features matrix which can change dynamically over time t. One instance of STGNN is Diffusion Convolutional Recurrent Neural Network (DCRNN)[19], which aims to solve traffic forecasting on road networks. Its general idea is to learn a function $h(\cdot)$ to predict the traffic network feature in a period of future time based on the history,

as depicted in the following formula:

$$\left[X^{(t-T'+1)},\cdots,X^{(t)};G\right] \xrightarrow{h(\cdot)} \left[X^{(t+1)},\cdots,X^{(t+T)}\right].$$

DCRNN models the spatial dependency of a traffic network by proposing the diffusion convolution, and solves the time dependency by deploying a Gated Recurrent Unit (GRU).

2.4.3 Message Passing

As specified in [20], modern GNN model follows a neighborhood aggregation mechanism to learn the graph structural information. The mechanism can be generally described as the following formulas:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in N(v)\right\}\right), \quad h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right),$$

where AGGREGATE function is usually mean, sum or maximum function which are used in works i.e. [20], [21], [22] and [23]. The mean or sum function calculates the mean value or the sum value of all the neighborhood and updates the target node with it, while the maximum function finds out the maximum neighborhood and updates the target node with this value.

Other than the aggregators above, [24] has proposed another way to formulate the AGGREGATE function. It defines the AGGREGATE function as continuous functions of multisets which compute a statistic on the neighbouring nodes, and states that in continuous input feature space, in which real-world tasks regularly resides, single aggregators fail to discriminate between multisets of size n. As depicted in Figure 2.7, the neighborhood messages of target node v form a multiset. Node v has different neighborhood multiset in Graph 1 and Graph 2 respectively. The aggregators in orange color fail to differentiate the two neighborhood multisets of node v, even though they are have different elements and multiplicity. It proves that the number of independent aggregators used is a limiting factor of the expressiveness of GNNs. Therefore, the Principal Neighbourhood Aggregation (PNA) is proposed to solve this problem.

The general idea of PNA is to use a combination of individual aggregators combining with degree-scalers to enhance the performance of GNN when discriminating different neighborhood topology. Degree-scaler is proposed to generalize the sum aggregator, so that the problem of the summation aggregation that it can not generalize well to unseen graphs can be solved. The logarithmic degree-scalers is calculated by the following formula:

$$S_{\text{amp}}(d) = \frac{\log(d+1)}{\delta} \quad , \quad \delta = \frac{1}{|\text{ train }|}\sum_{i\in \text{ train}} \log\left(d_i+1\right),$$

where d is the degree of the node receiving the message, δ is a normalization parameter computed over the training set. The sum aggregator can be then expressed by the product of a

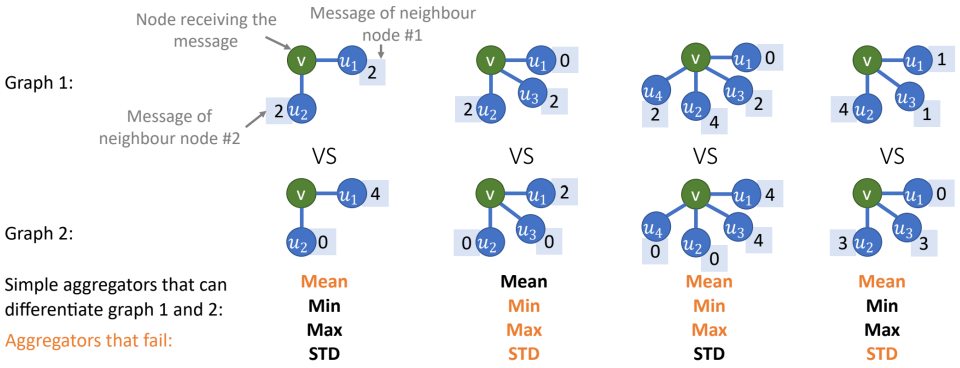


Figure 2.7: Examples that single aggregator fails to discriminate between neighborhood messages. Mean aggregator calculates the mean value of node v's neighbors, min aggregator calculates minimum value, max aggregator calculates maximum value and std aggregator calculates the standard deviation of node v's all neighbors.

mean aggregator and a linear-degree amplifying scaler $S_{\text{amp}}(d) = d$. The logarithmic scaler above can be further generalized to the following equation:

$$S(d,\alpha) = \left(\frac{\log(d+1)}{\delta}\right)^{\alpha}, \quad d > 0, \quad -1 \leq \alpha \leq 1,$$

where α is a variable parameter that is negative for attenuation, positive for amplification or zero for no scaling.

An PNA example of four aggregators and three degree-scalers is defined in the following equation:

$$\bigoplus = \underbrace{\begin{bmatrix} I \\ S(D,\alpha=1) \\ S(D,\alpha=-1) \end{bmatrix}}_{\text{scalers}} \otimes \underbrace{\begin{bmatrix} \mu \\ \sigma \\ \max \\ \min \end{bmatrix}}_{\text{aggregators}},$$

where $\otimes$ is the tensor product. The diagram of this equation is shown in Figure 2.8

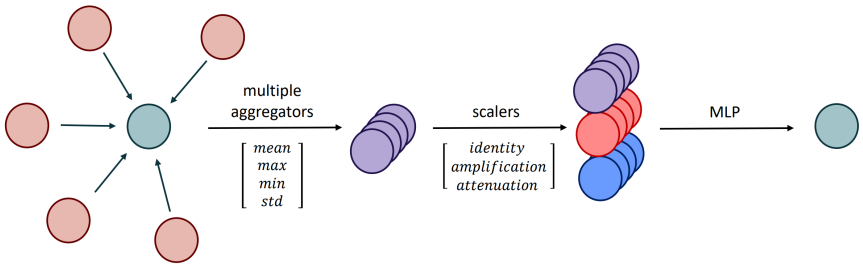


Figure 2.8: PNA overview.

Chapter 3

Related Work

After clarifying the fundamental concepts and general background of the techniques in the previous chapter, previous works that are related to the topic of this thesis are shown in this chapter.

Since various Graph Neural Network (GNN)-models targeting different problems exist, they are grouped according to their relations to the thesis into three sections. First, general GNN models which provide basic GNN frameworks to solve graph-based problems are shown in section 3.1. Then, GNN models developed for physical simulation are reviewed in section 3.2. Lastly, the MeshGraphNets (MGN), which is the basement of this thesis, is presented in the last section.

3.1 General Models

3.1.1 Graph Convolution Network

Graph Convolutional Network (GCN) is a GNN model introduced by Kipf and Welling in 2016[21]. It aims to solve the problem of semi-supervised classification of nodes over graph. A popular method of solving such problem before GCN is presented is to add a Laplacian regularization term in the loss function:

$$L = L_0 + \lambda L_{\text{reg}}\,, \quad \text{with} \quad L_{\text{reg}} = \sum_{i,j} A_{ij} \left\| f(X_i) - f(X_j) \right\|^2 = f(X)^\top \Delta f(X),$$

where L_0 is the supervised loss, $f(\cdot)$ can be a neural network-like differentiable function, λ is a weighing factor and X is a matrix of node feature vector X_i. Δ denotes the unnormalized graph Laplacian of an undirected graph $G = (V, E)$. The deficiency of this method is that it predicts the label of the unlabeled nodes according to their distance to the labeled nodes, while graph edges do not necessarily encode node similarity. To address this shortcoming, GCN is presented.

The GCN model is motivated by the spectral graph convolutions and layer-wise linear model. In GCN, spectral graph convolution is defined as the multiplication of a signal $x \in \mathbb{R}^N$ with a filter $g_\theta = \text{diag}(\theta)$ parameterized by $\theta \in \mathbb{R}^N$ in the Fourier domain, as shown below:

$$g_\theta \star x = U g_\theta U^\top x,$$

where U is the matrix of eigenvectors of the normalized graph Laplacian $L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} = U \Lambda U^\top$, with a diagonal matrix of its eigenvalues Λ and $U^\top x$ being the graph Fourier transform of X. g_θ is then instantiate with Λ. To reduce the computational cost of $O\left(N^2\right)$ of the above formula, a truncated expansion in terms of Chebyshev polynomials $T_k(x)$ up to $T_k(x)$ order is deployed to approximated the original $g_{\theta'}(\Lambda)$ as follows:

$$g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{\Lambda}),$$

where $\tilde{\Lambda} = \frac{2}{\lambda_{\max}} \Lambda - I_N$, with $\lambda_{\max}$ denotes the largest eigenvalue of L. The Chebyshev polynomials are recursively defined as $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$, with $T_0(x) = 1$ and $T_1(x) = x$. The convolution calculation is finally adapted to the following formula:

$$g_{\theta'} \star x \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) x,$$

with $\tilde{L} = \frac{2}{\lambda_{\max}} L - I_N$.This formula only depends on nodes that are K steps from the central node since it is a K^{th} -order polynomial in the Laplacian, so that the complexity of the convolution is reduced from $O\left(N^2\right)$ to $O(|E|)$, with E is the number of edges.

When the convolutional layers described by the formula above are stacked together to form a neural network model, it can be approximated that $\lambda_{\max} \approx 2$ since the neural network parameters will adapt to this change in scale during training. Thus, the formula above can be further simplified to the following formula:

$$g_{\theta'} \star x \approx \theta'_0 x + \theta'_1 (L - I_N) x = \theta'_0 x - \theta'_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x,$$

with two free parameters θ'_0 and θ'_1. In practice, the number of parameters is further constrained to address overfitting and to minimize the number of operation per layer. So a new parameter can be introduced:

$$\theta = \theta'_0 = -\theta'_1$$

so that the formula is now simplified to:

$$g_\theta \star x \approx \theta \left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \right) x.$$

In order to deal with numerical instabilities and exploding/vanishing gradients that can occur during repeated application of this operator in a deep neural network model, a renormalization trick is applied so that $I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \rightarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$, with $\tilde{A} = A + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.

This formula can be generalized to the following formula:

$$Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta,$$

where $X \in \mathbb{R}^{N \times C}$ is a matrix of filter parameters, $Z \in \mathbb{R}^{N \times F}$ is the convolved signal matrix with F as the number of feature map.

Finally, the layer-wise propagation rule of a multi-layer GCN is derived as follows:

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

with $\tilde{A} = A + I_N$ being the adjacency matrix with self-connections, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and $W^{(l)}$ is a layer-specific trainable weight matrix, $\sigma(\cdot)$ denotes an activation function. $H^{(l)} \in \mathbb{R}^{N \times D}$ denotes the output matrix of the l^{th} layer with $H^{(0)} = X$.

Although the GCN model have shown the best performance and a reduced training time until convergence at the time it was presented, it does not support edge features. In addition, as the layer output is calculated through matrix multiplication, the neighborhood aggregation method is not modularized. Moreover, no attention mechanism is applied to the network to accelerate learning.

3.1.2 Variational Graph Autoencoder

Variational Graph Autoencoder (VGAE)[18] is a GNN model with integration of *variational autoencoder*. It is proposed by Kipf and Welling in 2016. VGAE aims to solve unsupervised learning task on graph-structured data, such as link prediction.

Given the input graph $G = (V, E)$ with $N = |V|$, its adjacency matrix A with self-connection, degree matrix D, and node feature matrix $\mathbf{X}$ of dimension $N \times D$, VGAE utilizes two-layer GCN defined as $\text{GCN}(\mathbf{X}, \mathbf{A}) = \tilde{\mathbf{A}}\,\text{ReLU}\left(\tilde{\mathbf{A}}\mathbf{X}\mathbf{W}_0\right)\mathbf{W}_1$ with $\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ to calculate the matrix of mean vectors μ_i and logarithm of variance vectors according to the two formulas $\mu = \text{GCN}_\mu(\mathbf{X}, \mathbf{A})$ and $\log \sigma = \text{GCN}_\sigma(\mathbf{X}, \mathbf{A})$ respectively to obtain the stochastic latent matrix $\mathbf{Z}$ of dimension $N \times F$. The probability of encoding a graph into a stochastic latent matrix $\mathbf{Z}$

is defined as follows:

$$q(\mathbf{Z} \mid \mathbf{X}, \mathbf{A}) = \prod_{i=1}^{N} q\left(\mathbf{z}_i \mid \mathbf{X}, \mathbf{A}\right), \text{ with } q\left(\mathbf{z}_i \mid \mathbf{X}, \mathbf{A}\right) = N\left(\mathbf{z}_i \mid \mu_i, \operatorname{diag}\left(\sigma_i^2\right)\right).$$

And when decoding the latent matrix back to a graph, VGAE follows the formula below:

$$p(\mathbf{A} \mid \mathbf{Z}) = \prod_{i=1}^{N} \prod_{j=1}^{N} p\left(A_{ij} \mid \mathbf{z}_i, \mathbf{z}_j\right), \text{ with } p\left(A_{ij} = 1 \mid \mathbf{z}_i, \mathbf{z}_j\right) = \sigma\left(\mathbf{z}_i^{\top} \mathbf{z}_j\right).$$

3.1.3 Graph Attention Network

As attention mechanism is becoming an almost de facto standard in many sequence-based tasks, Veličković et al. has ported it to GNN model and presented Graph Attention Networks (GAN) in 2018[25]. The attention mechanism over GNN is proposed as a *graph attention layer*, which outputs the normalized attention coefficients to weight the neighbors of all nodes. To be specific, the attention layer takes a set of node features $\mathbf{h} = \left\{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\right\}$ with $\vec{h}_i \in \mathbb{R}^F$, where N is the number of nodes and F is the number of features in each node, as input, and outputs a new set of node features $\mathbf{h}' = \left\{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\right\}$ with $\vec{h}'_i \in \mathbb{R}^{F'}$, where F' is the number of features in the new set.

To obtain the new set $\mathbf{h}'$, first, a learnable linear *weight matrix* $\mathbf{W} \in \mathbb{R}^{F' \times F}$ in the initial step. Then, it is applied to every node of the graph. Afterwards, a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$ performs the *self-attention* on the neighboring nodes to compute *attention coefficients* as shown in the formula below:

$$e_{ij} = a\left(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j\right),$$

which indicate the importance of node j's features to node i.

The attention coefficients of the neighboring nodes are then normalized with softmax function as follows:

$$\alpha_{ij} = \operatorname{softmax}_j\left(e_{ij}\right) = \frac{\exp\left(e_{ij}\right)}{\sum_{k \in N_i} \exp\left(e_{ik}\right)},$$

so that the attention coefficients of the neighbors of a node can be easily compared.

Finally, the normalized attention coefficients are used to compute the output node features as specified below:

$$\vec{h}'_i = \sigma\left(\sum_{j \in N_i} \alpha_{ij} \mathbf{W}\vec{h}_j\right),$$

where σ is an activation function.

Besides self-attention, *multi-head attention*[26] is another feasible approach to generate the layer output. In comparison to self-attention, multi-head attention has a more stable learning process. With multi-head attention, the output of the attention layer is modified as follows:

$$\vec{h}'_i = \|_{k=1}^{K} \sigma\left(\sum_{j \in N_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right),$$

where $\|$ denotes concatenation, α_{ij}^k are normalized attention coefficients computed by the k-th attention mechanism $\left(a^k\right)$, and $\mathbf{W}^k$ is the corresponding input linear transformation's weight matrix.

Though the GAN model can produce a state-of-the-art performance without requiring costly additional computation power, but it does not support edge features, which will lead to information loss when it is deployed in physical simulation cases.

3.2 Physical Applications

3.2.1 Interaction Networks

Battaglia et al. has proposed the *interaction network (IN)* in 2016. The main purpose of this network is to allow reasoning about the objects interacting in a complex physical system and making dynamical predictions. As depicted in Figure 3.1[5], the model takes the objects and their relations as input, and learn their interactions. Then, the effects of the interaction is predicted and apply to the relative objects respectively to predict the relations of the objects in the next time step.

The precise definition of IN is given in the following formula:

$$\mathrm{IN}(G) = \phi_O\left(a\left(G, X, \phi_R(m(G))\right)\right).$$

This formula is visualized in Figure 3.2. The IN model takes a graph $G = \langle O, R \rangle$, where O is the set of objects and R is the set of relations, as input. Then the marschalling function m rearranges the objects and relations into interaction tuples $b_k = \langle o_i, o_j, r_k \rangle \in B$. Afterwards, the relational model ϕ_R applies a function f_R to each b_k to obtain the effect of the interaction. The effect alongside with the graph G and external effects X are fed into aggregation function a, which aggregates for each objects all of its relative interactions. Finally, the output of the aggregation function is fed into the output function f_O of object model ϕ_O to predict the objects state in next time step after applying the physical influence in the current steps.

In [5], the function f_R and f_O are implemented as multilayer perceptrons (MLP) with Rectified Linear Unit (ReLU), the aggregation function a uses addition to aggregate the interactions of an object and then concatenates its inputs to form the output.

The experiment result of IN is shown in Figure 3.3.

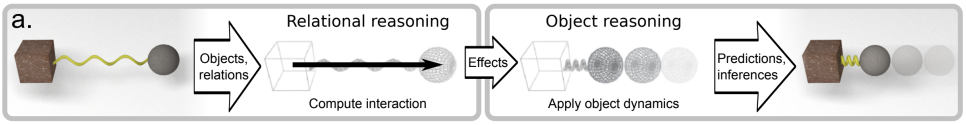


Figure 3.1: Schematic of IN.

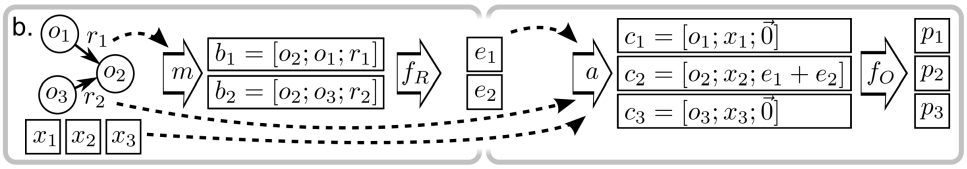
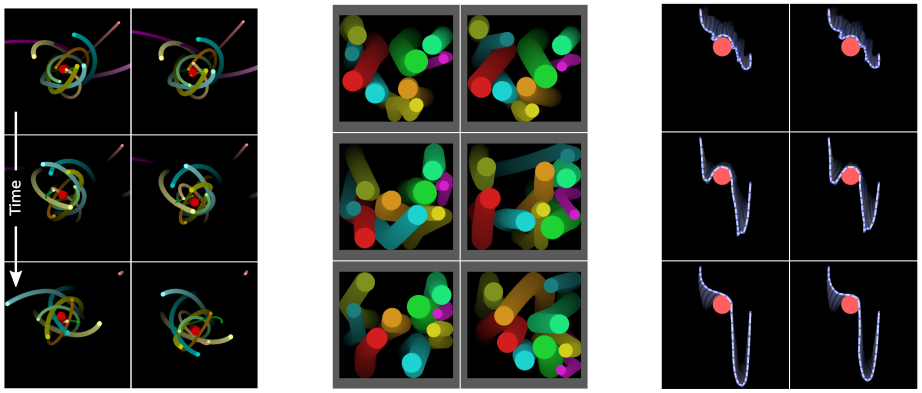


Figure 3.2: Visualization of definition of IN.



(a) N-body experiment with 12 bodies.

(b) Ball experiment with 9 bodies.

(c) String experiment with two-end-pinned string.

Figure 3.3: Experiment result of IN.

3.2.2 Dynamic Particle Interaction Networks

Li et al. has proposed the Dynamic Particle Interaction Networks (DPI-Net) in 2019. Basing on the IN model in the previous subsection, they proposed this differential, particle-based DPI-Net to learn interaction between rigid bodies, deformable objects and fluid. The DPI-Net takes a graph at the current step with nodes representing the particles in the environment and edges representing the relation between the particles (e.g. collision and contact) as input and outputs the graph prediction at the next time step, as seen in Figure 3.4. For control tasks, the control inputs are also part of the interaction graph, for example the velocities or initial positions of a particular set of particles. The control input sequence of the gripper is denoted as $\hat{u}_{1:T}$, with $\hat{u}_t$ being the input at time t. The resulting trajectory after applying $\hat{u}$ is denoted as $G = \{G_i\}_{i=1:T}$, with G_i being the graph after applying $\hat{u}_t$. The goal is denoted as G_g, and the DPI-Net is supposed to minimize the distance between the actual outcome and the specified goal $L_{\text{goal}}(G, G_g)$.

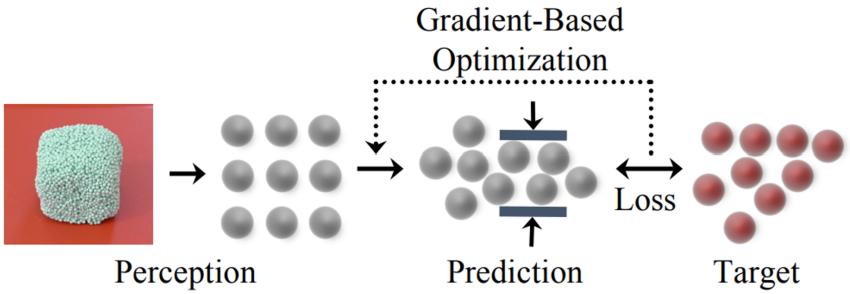


Figure 3.4: Training and inference of the DPI-Net.

DPI-Net has the following three key features for effective particle-based simulation:

1. **Multi-step spatial propagation** Multi-step spatial propagation originated from [27] handles the instantaneous propagation of forces in a single time step by doing multi-step message passing, as seen in Figure 3.5. It defines the propagating influence from object i at time step t as $h^l_{i,t}$, where l denotes the step size of message passing at one time step. $h^0_{i,t}$ is initialized as follows:

$$h^0_{i,t} = \mathbf{0}, \quad i = 1 \dots |O|,$$

and $h^l_{i,t}$ at step $1 \leq l \leq L$ is calculated by first updating the propagating influence of relation k:

$$e^l_{k,t} = f_R\left(c^r_{k,t}, h^{l-1}_{u_k,t}, h^{l-1}_{v_k,t}\right), k = 1 \dots |R|,$$

and then updating the propagating influence of object i:

$$h^l_{i,t} = f_O\left(c^o_{i,t}, \sum_{k \in N_i} e^l_{k,t}, h^{l-1}_{i,t}\right), i = 1 \dots |O|$$

The final output of the updated object is then calculated by the following equation:

$$\hat{o}_{i,t+1} = f^{\text{output}}_O\left(h^L_{i,t}\right), \quad i = 1 \dots |O|$$

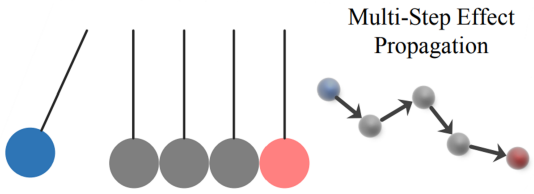
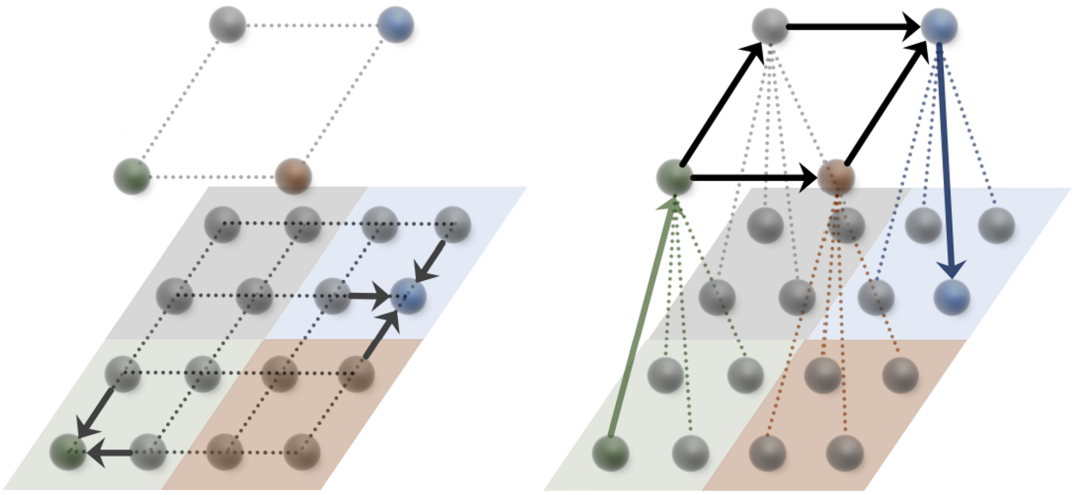


Figure 3.5: Multi-step message passing for handling instantaneous force propagation.

2. **Hierarchical particle structure** In order to efficiently propagate the instantaneous message in a single time step, DPI-Net employs hierarchical particle structure. To be specific, one level of hierarchy is added to the network. To achieve this, first the nodes

of the graph are clustered into several non-overlapping clusters, as seen in Figure 3.6a and 3.6b, which are colored with four different colors. Afterwards, a new node is added to the clusters which need hierarchical modeling as root node of that cluster. Then, extra edges are added between each root node and its leaf nodes, as well as each root nodes, in a two-way manner respectively, so that message can be propagated more efficiently. The hierarchical message passing functions in a four-step manner: first, message is



(a) Local propagation between particles. (b) Hierarchical propagation between particles.

Figure 3.6: Local and hierarchical propagation.

propagated among leaf nodes, as shown in Figure 3.6a. Second, message is propagated from leaf nodes to root nodes. Third, message is propagated beteen root nodes. Fourth, the message is propagated from root nodes to their leaf nodes.

3. **Dynamic interaction graphs** The edges of the input graph are generated dynamically regarding the specific environment and task. In DPI-Net, the distance of two nodes is taking into consideration. Edge is generated if the distance between two nodes are smaller than a threshold. As seen in Figure 3.7, there exists an edge between the green and red node when they are close in the environment, and the edge is removed when they are far away from each other.

   There are two major advantages when the graph is generated dynamically: first, since each particle interacts only with a limited set of other particles, it is more efficient to only process edges that are actually needed. Second, it enables the model to tackle discontinuity in physical interactions, such as contact between nodes.

The DPI-Net is integrated with classic trajectory optimization algorithms for control. It takes the raw object observations (e.g., positions, velocities) and learns the underlying physics. The evaluation of the DPI-Net is performed in four environment, which are the FluidFall, BoxBath, FluidShake and RiceGrip. The RiceGrip task is the most similar task to this thesis among all the other tasks, therefore its experiment configuration and evaluation result is shown below.

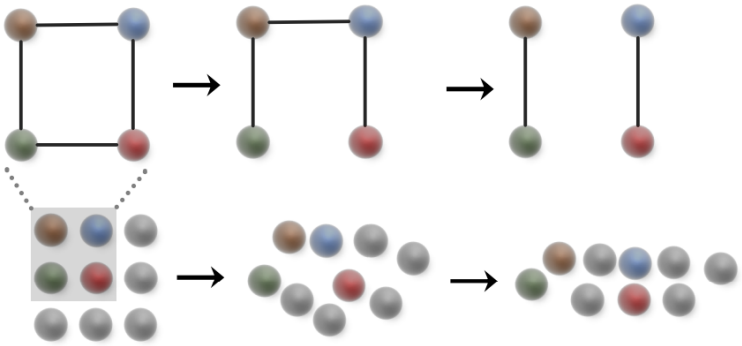


Figure 3.7: Dynamically built interaction graph.

In the RiceGrip task, the DPI-Net learns to simulate the deformation of the sticky rice being gripped by a gripper and control the gripper to deform the rice into a targeted shape. The sticky rice is modeled as an elastic and plastic object represented by particles, while the gripper is composed of two cuboids, each represented by a single particle. When the "rice" particles are close to each other, then two-way edges are generated between them to propagate e.g. collision message. When the "gripper" particles are close to the "rice" particles, then one-way edges from "gripper" particle to "rice" particle are generated to propagate e.g. contact message. The quantitative evaluation results of RiceGrip when using different hypermeters, namely the number of root nodes, the propagation steps and the neighbor radius for edge generation are shown in Figure 3.8. The qualitative evaluation results are shown in Figure 3.9a, 3.9b and 3.9c.

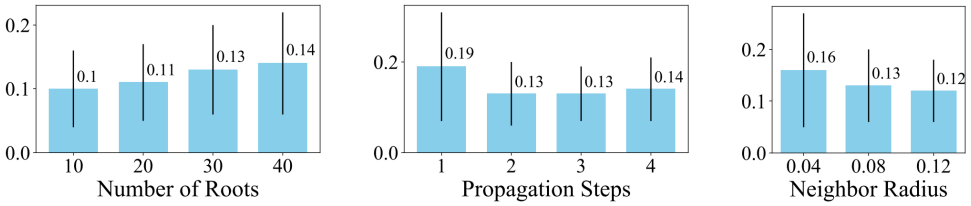
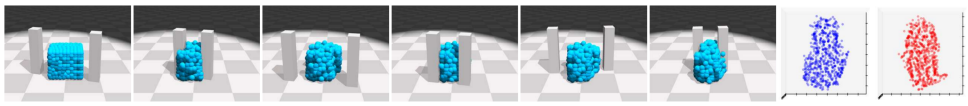


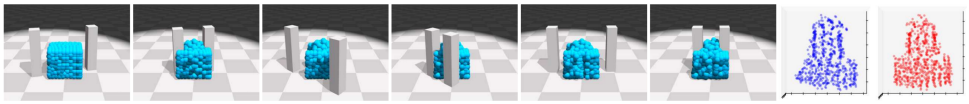
Figure 3.8: Quantitative evaluation result of RiceGrip task when using different hypermeters.

3.2.3 Graph Networks as Learnable Physics Engines

Sanchez-Gonzalez et al. has proposed a Graph Networks (GN)-based feedforward model[3] in 2018, which is another generalized architecture of the IN. Like the IN model, it utilizes object- and relation-centric representations to learn from complex physical environments and predict the future features of the graph. The GN-based feedforward model takes as input a graph $G = \left(\mathbf{g}, \{\mathbf{n}_i\}_{i=1\cdots N_n}, \{\mathbf{e}_j, s_j, r_j\}_{j=1\cdots N_e}\right)$, where $\mathbf{g}$ is a vector of global features, $\{\mathbf{n}_i\}_{i=1\cdots N_n}$ is a set of nodes where each $\mathbf{n}_i$ is a vector of node features, and $\{\mathbf{e}_j, s_j, r_j\}_{j=1\cdots N_e}$ is a set of directed edges where $\mathbf{e}_j$ is a vector of edge features, and s_j and r_j are the indices of the sender and receiver nodes, respectively. An example is shown in Figure 3.10. The model outputs



(a) Evaluation result 1 on evaluation dataset.



(b) Evaluation result 2 on evaluation dataset.

(c) Evaluation result in real-world task.

Figure 3.9: Qualitative evaluation result of RiceGrip task on evaluation dataset and in real-world task. Visualization of the gripping process is shown on the left, result shape and target shape of the rice viewed from the top is shown on the right.

a graph of the same structure and size of the input graph, but with different node and edge features. This process is termed "graph2graph".

The GN-based feedforward model is composed of GN blocks. As seen in Figure 3.11, a GN block is consists of three basic components: the edge-wise function f_e, the node-wise function f_n and the global function f_g. When an input graph is presented to the GN block, the function f_e takes the graph as input and update all the edges. Secondly, the updated edge features of the same receiver are aggregated by summing up all the features, then the summed up features are passed to the function f_n to update the node features. After that, the updated edge features and node features are summed up respectively and fed into the function f_g to update the global features. Details can be seen in Figure 3.12.

In the GN-based feedforward model implementation, the three functions f_e, f_n and f_g are implemented with MLP. Two GN blocks are cascaded together to compute the output, as depicted in 3.13. The reason for the second block is to allow the nodes and edges to communicate with each other through the output of the first block. The experiment result of the model is shown in Figure 3.14.

3.2.4 Graph Network-based Simulators

Sanchez-Gonzalez et al. has proposed the Graph Network-based Simulators (GNS) in 2020, which is an architecture based on their previous work GN. The model aims to solve the particle-based physical simulation with GN presented in [3]. The GNS has an Encoder-Processor-Decoder architecture, as shown in Figure 3.15. The encoder is responsible for taking a world state X of particles in a time step, encoding them into a graph G^0, as depiced in Figure 3.16. In this process, node embeddings $\mathbf{v}_i = \varepsilon^v(\mathbf{x}_i)$ and edge embeddings $\mathbf{e}_{i,j} = \varepsilon^e(\mathbf{r}_{i,j})$

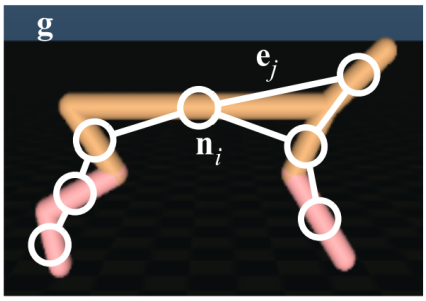


Figure 3.10: Graph representation of a cheetah model. Bodies and joints of the cheetah are represented as nodes, their connections are represented as edges.

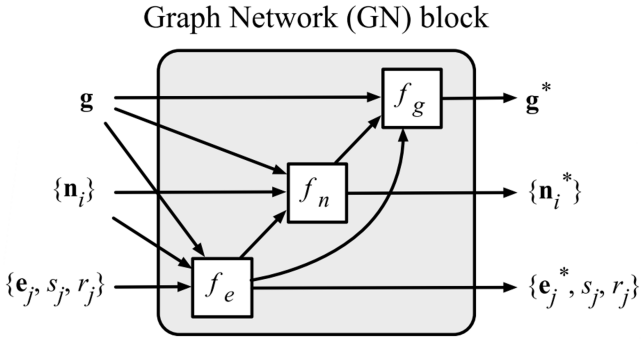


Figure 3.11: Structure of GN block.

Algorithm 1 Graph network, GN

Input: Graph, $G = (\mathbf{g}, \{\mathbf{n}_i\}, \{\mathbf{e}_j, s_j, r_j\})$
for each edge $\{\mathbf{e}_j, s_j, r_j\}$ **do**
  Gather sender and receiver nodes $\mathbf{n}_{s_j}, \mathbf{n}_{r_j}$
  Compute output edges, $\mathbf{e}_j^* = f_e(\mathbf{g}, \mathbf{n}_{s_j}, \mathbf{n}_{r_j}, \mathbf{e}_j)$
end for
for each node $\{\mathbf{n}_i\}$ **do**
  Aggregate $\mathbf{e}_j^*$ per receiver, $\hat{\mathbf{e}}_i = \sum_{j/r_j=i} \mathbf{e}_j^*$
  Compute node-wise features, $\mathbf{n}_i^* = f_n(\mathbf{g}, \mathbf{n}_i, \hat{\mathbf{e}}_i)$
end for
Aggregate all edges and nodes $\hat{\mathbf{e}} = \sum_j \mathbf{e}_j^*$, $\hat{\mathbf{n}} = \sum_i \mathbf{n}_i^*$
Compute global features, $\mathbf{g}^* = f_g(\mathbf{g}, \hat{\mathbf{n}}, \hat{\mathbf{e}})$
Output: Graph, $G^* = (\mathbf{g}^*, \{\mathbf{n}_i^*\}, \{\mathbf{e}_j^*, s_j, r_j\})$

Figure 3.12: Algorithm of GN block.

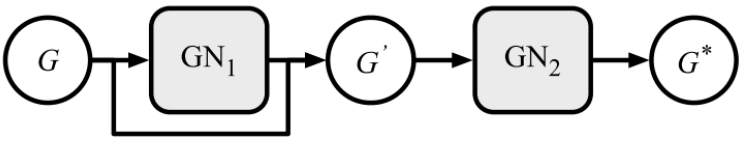


Figure 3.13: The feed-forward GN-based forward model for learning one-step predictions.

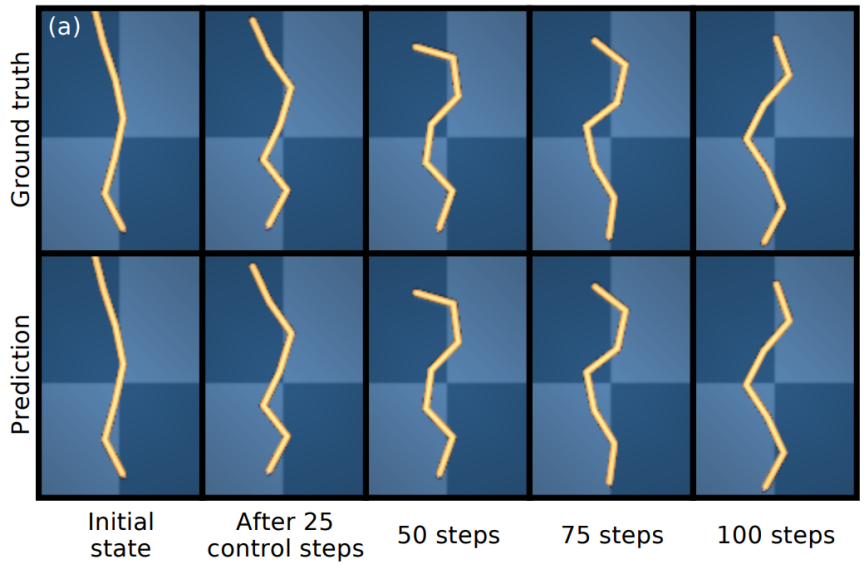


Figure 3.14: Evaluation rollout of 100 steps in a Swimmer6. The first row shows the ground truth and the second row shows the prediction.

Figure 3.15: Encoder-Processor-Decoder architecture of GNS.

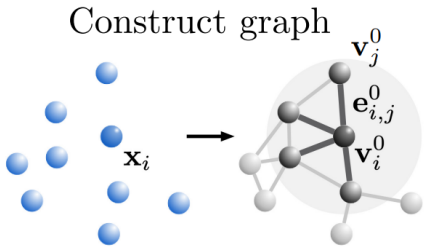


Figure 3.16: Encoder constructs a latent graph from the world state.

are encoded from particles' states and pairwise properties of the corresponding particles $\mathbf{r}_{i,j}$. A graph $G^0 = (V, E, \mathbf{u})$ with $\mathbf{v}_i \in V$, $\mathbf{e}_{i,j} \in E$ and u as the global-level embedding. The edges of G^0 are added before embedded if two particles fulfill a condition, for example, within a connectivity radius. In the implemetation of GNS, the function ε^v and ε^e are implemented as MLP.

After the latent graph G^0 is constructed, it is fed into the following processor for further processing. The processor is composed of a sequence of cascaded GN blocks, each of which represents a message passing step of the graph, as shown in Figure 3.17. In the processor, a sequence of updated graphs $\mathbf{G} = \left(G^1, \ldots, G^M\right)$, in which $G^{m+1} = \mathrm{GN}^{m+1}\left(G^m\right)$, is generated. The output graph of processor is G^M, the final graph of the sequence.

The output of processor is then decoded with the decoder, and dynamics information is extracted as $\mathbf{y}_i = \delta^v\left(\mathbf{v}_i^M\right)$. In the implementation, the δ^v function is implemented as MLP.

After the Encoder-Processor-Decoder architecture, which is refered to as the learnable simulator function d_θ, has finished processing the input data, an update function is applied to the output of d_θ and the input data to obtain the prediction of the world state, as shown in Figure 3.19. The update function can be, for example, an Euler integrator if d_θ outputs the acceleration.

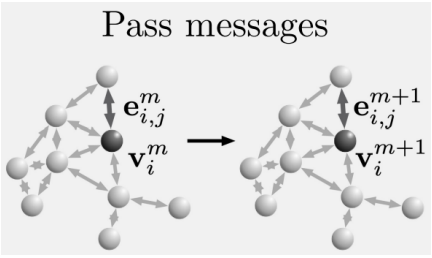


Figure 3.17: Processor executes the message passing steps.

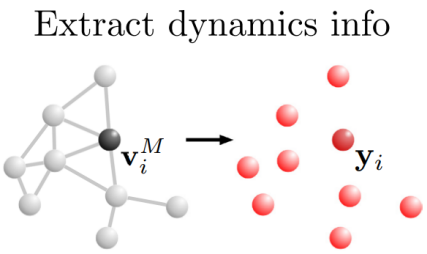


Figure 3.18: Decoder extracts the dynamics information.

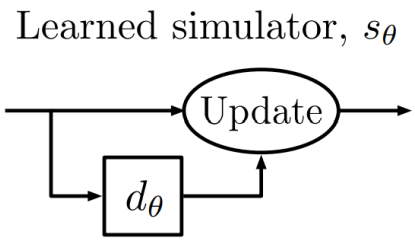


Figure 3.19: GNS structure.

3.3 MeshGraphNets

3.3.1 Overview

In 2021, Pfaff et al. has proposed the MGN framework for learning mesh-based simulations using graph neural networks. Mesh-based simulations are crucial to modeling of complex physical systems in many disciplines across science and engineering, because it provides fundamental support for various numerical methods.

However, high-dimensional scientific simulations are very expensive to run, and solvers and parameters must often be tuned respectively to each system studied. Therefore, MGN is presented, aiming to utilize deep learning methodology to enhance the mesh-based simulation. [1] indicates that MGN provides a 1-2 orders of magnitude faster simulation in compared to conventional Partial Differential Equation (PDE)-solving methods, while maintaining the accuracy of the simulation.

Similar to GNS, the MGN model also utilizes an Encoder-Processor-Decoder architecture, as shown in Figure 3.20. It takes a graph $G = \left(V, E^M, E^W\right)$ as input, which is encoded from the state of the system. The state of the system at time t is given as a mesh $M^t = \left(V, E^M\right)$. Each node is associated with a mesh-space coordinate $\mathbf{u}_i$ and a world-space coordinate $\mathbf{x}_i$ if the system is a Lagrangian system, for example, the system of FlagSimple task (Figure 3.21a) or of DeformingPlate task (Figure 3.21b).

Additionally, each node has dynamic quantities $\mathbf{q}_i$ that are going to be modelled. During the encoding, the mesh nodes are taken as graph nodes, and mesh edges become bidirectional

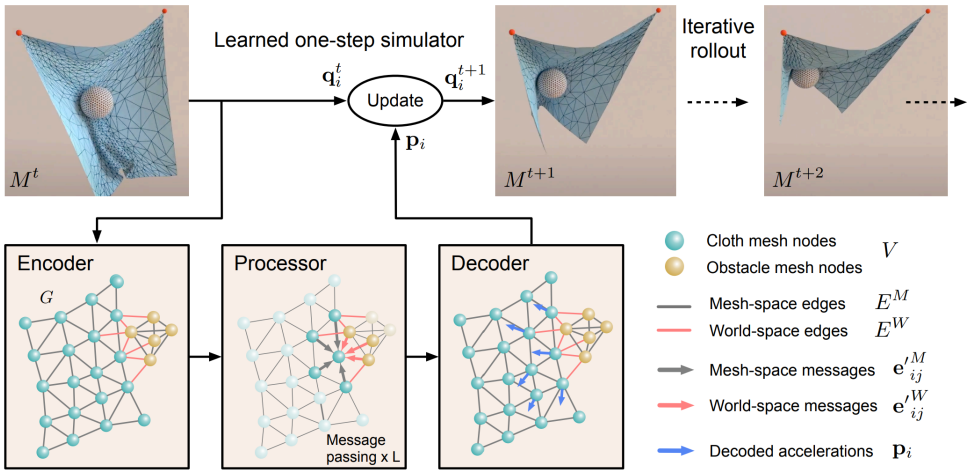
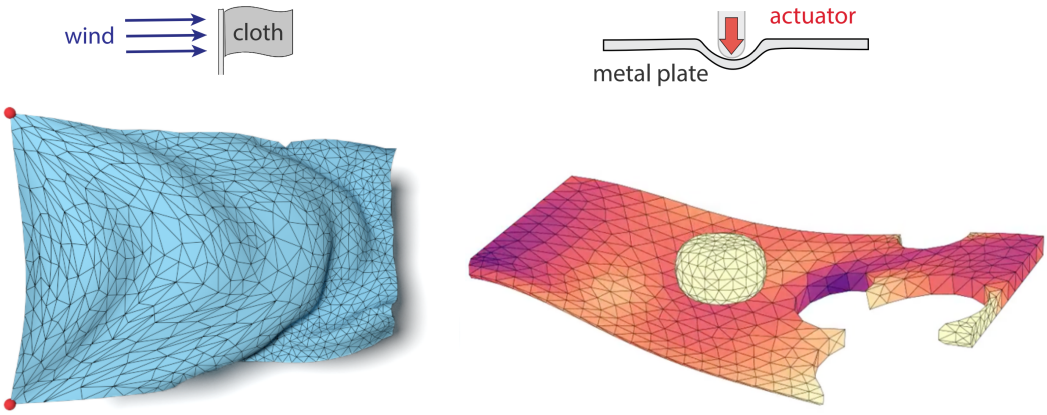


Figure 3.20: Architecture of MGN.



(a) FlagSimple task. (b) DeformingPlate task.

Figure 3.21: FlagSimple task and DeformingPlate task.

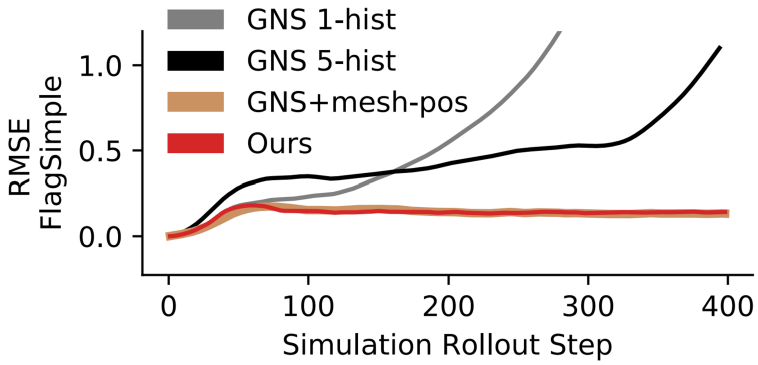
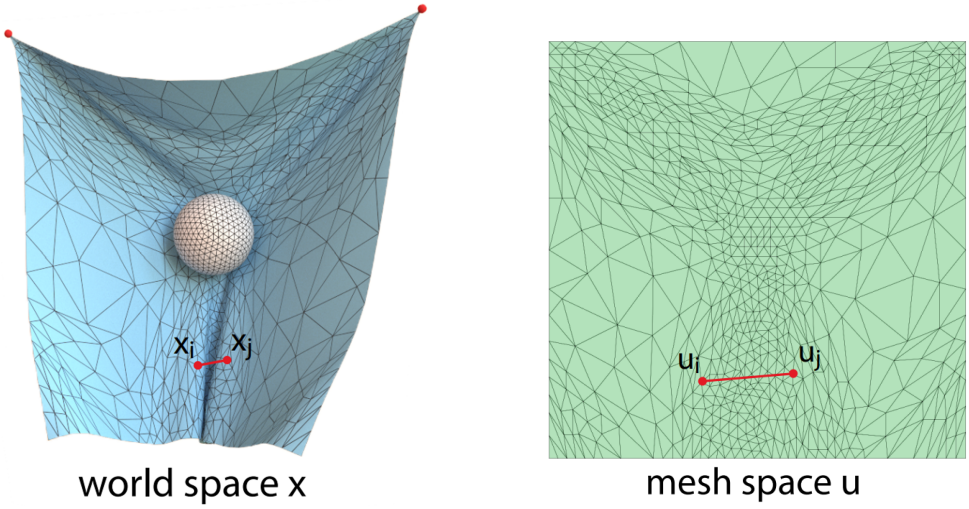


Figure 3.22: Experiment result of MGN in FlagSimple task.

mesh-edges E^M in the graph. In a Lagrangian system, a world edge is added to a node-pair if

they are not connected via mesh edge and their world-space distance is with a threshold r_W, as shown in Figure 3.23b and 3.23a. The world edges serve as a bridge for communication



(a) Edge in world space.

(b) Edge in mesh space.

Figure 3.23: World edge and mesh edge.

between two nodes that are spatially close and distant in mesh space, so that external dynamics like collision and contact can be modelled, while the mesh edges is responsible for modelling the internal dynamics. The nodes, mesh edges and world edges are finally encoded with MLP $\varepsilon^M, \varepsilon^W$ and ε^V to construct the output latent graph.

After encoding, the latent graph is passed to the processor, which is based on GN blocks[3]. Each GN block in the processor allows the nodes to communicate with each other via message passing. It updates the input graph according to the following formula:

$$\mathbf{e}_{ij}^{\prime M} \leftarrow f^M\left(\mathbf{e}_{ij}^M, \mathbf{v}_i, \mathbf{v}_j\right), \quad \mathbf{e}_{ij}^{\prime W} \leftarrow f^W\left(\mathbf{e}_{ij}^W, \mathbf{v}_i, \mathbf{v}_j\right), \quad \mathbf{v}_i^{\prime} \leftarrow f^V\left(\mathbf{v}_i, \sum_j \mathbf{e}_{ij}^{\prime M}, \sum_j \mathbf{e}_{ij}^{\prime W}\right),$$

where $\mathbf{e}_{ij}^{\prime M}, \mathbf{e}_{ij}^W and \mathbf{v}_i^{\prime}$ are the updated mesh edges, world edges and nodes. The $f^M, f^W and f^V$ are implemented using MLPs with a residual connection. Then, it generates a new latent graph and passes the new graph to the GN block in the next level to at last output the final latent graph to the decoder.

In the decoder, the nodes features $\mathbf{V}_i$ is decoded into output features $\mathbf{p}_i$, which represents the predicted dynamics information.

Afterwards, the update function of MGN computes the next-step dynamical quantity $\mathbf{q}_i^{t+1}$ by selecting an update operator, i.e. an Euler integrator, and taking the model's input dynamical quantity and the Encoder-Processor-Decoder's output as input to the operator. In the end, the next-step mesh M^{t+1} is produced.

To enhance the robustness of the MGN model, random normal noise of zero mean and fixed variance training noise is added to the input dynamical features before they are fed into the model.

The experiment result of Flag task is shown in Figure 3.22. The red line indicates the result of the MGN model. It is compared with the GNS model. The figure shows that the MGN model has an overall better rollout result with different rollout steps than the GNS model with one and five steps of history, while only GNS model with mesh position information can keep in pace with the accuracy of the MGN model.

3.3.2 FlagSimple Task Details

In the FlagSimple task, the swing of a cloth blowing by wind is simulated. The main body of the cloth is represented by mesh. The mesh nodes are partitioned into handle nodes, which are nodes of the swinging cloth pinned in the environment and remain fixed to hang the cloth during the simulation, and normal nodes, which change their velocity and world position dynamically to simulate the cloth swing. During the training of the network, the movement of the normal nodes is learned by the network, while the loss of the handle nodes is masked out and their movement is not learned.

Besides, no world edges are generated during the training. Two-way connected mesh edges of both normal and handle nodes are the only edges that transfer message between nodes.

In this task, the target output is the acceleration of the node. The output acceleration is integrated to calculate the predicted world position of the node after the processing of the network according to the following equation:

$$\mathbf{q}_i^{t+1} = \mathbf{p}_i + 2\mathbf{q}_i^t - \mathbf{q}^{t-1},$$

where $\mathbf{p}_i$ is the network output of the i-th input, $\mathbf{q}_i^{t-1}$, $\mathbf{q}_i^t$ and $\mathbf{q}_i^{t+1}$ are the dynamic quantities of the i-th node at sequencing time steps.

To enhance the robustness of the trained model, noise is added to world position feature of the nodes. Since the position and the velocity depends strongly on each other, when the input world position is noisy, it is impossible to predict an acceleration $\tilde{\tilde{x}}_i$ that can correct the noise of both the predicted next step velocity $\tilde{x}_i^{t+1}$ and the next step position $\tilde{x}_i^{t+1}$. In order to tackle this problem, a hyperparameter $\gamma \in [0,1]$ is chosen empirically to parametrize a weighted average between the two options of the predicted accelerations that correct the noise for next step velocity and for next step world position. γ is chosen to be 0.1 in this task since it delivers the best performance.

3.3.3 DeformingPlate Task Details

In the DeformingPlate task, the environment consists of a deformable plate and an undeformable obstacle, each is represented by an individual mesh. The mesh nodes of the deformable plate have two types in the generated graph, which are the normal type, indicating that the nodes are deformable and thus their world positions are changing dynamically during the simulation, and the handle type, indicating that these nodes remain fixed in the graph to hang the plate in the environment. The obstacle nodes have an obstacle type, indicating that they are changing their world position according to a scripted constant speed and direction, regardless of whether it hits the plate. The plate remains unchanged until it is hit by the obstacle. Then it deforms according to the force of the obstacle, the resistent force from the handle nodes and the stiffness of the normal nodes.

The graph contains both mesh edges and world edges. The mesh edges is used to represents the internal dynamics to keep the obstacle and deformable plate in shape, while the world edges represents the force of the obstacle apply to the plate or the self-collision force of the plate. Mesh edges are generated according to the connection of the nodes in the mesh, while world edges are generated only when the distance of the world position of two nodes is less than a given value r_W, in order to detect and calculate the external dynamics of the nodes, such as collision and contact.

The target output of this task is the velocity of the nodes between two sequencing time steps. The world position prediction of a node is calculated as follows:

$$\mathbf{q}_i^{t+1} = \mathbf{p}_i + \mathbf{q}_i^t,$$

where $\mathbf{p}_i$ is the network output of the i-th input, $\mathbf{q}_i^t$ and $\mathbf{q}_i^{t+1}$ are the dynamic quantities of the i-th node at sequencing time steps.

In this task, noise is also added to the world position of the nodes. Since the domain of DeformingPlate task is a first-order system, the output velocity can be adjusted efficiently so that the predicted next step world position can be corrected to match the ground truth.

Chapter 4

Architecture

MeshGraphNets (MGN) has proven its accuracy and inference efficiency when simulating complex physical system. However, one major drawback of the model is its long training time. It takes more than one week to train a model for one task on a modern graphics processing unit (GPU) when its processor is configured to have 15 Graph Networks (GN) blocks. This makes it time-consuming to evaluate the model after making modification of the original MGN model or train another model for another task.

Another problem is that when a group of the nodes are being influenced by some external interactions, for example, an obstacle is crashing with a metal plate, then it will take a long time for the nodes that are far away from these nodes to obtain the dynamic information. A simplified example is shown in Figure 4.1, where the Node 1 has undergone some external changes and Node n has to take $O(n)$ message passing steps to aggregate the information from Node 1.

Therefore, the Ripple Networks (RNs), equipped with some other optimization methods, is proposed in this thesis to alleviate both issues. RNs are a modification of the MGN model that allow for more efficient communication between far-away nodes, and in turn achieve similar performance to the MGN model with far fewer message passing blocks. In this section, the architecture of the RNs and the other optimization methods are presented in detail.

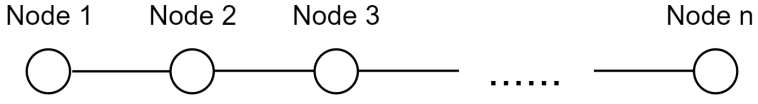


Figure 4.1: Message passing along a list of linked nodes.

4.1 Ripple Network

4.1.1 Overview

The RNs is inspired by water ripples. As seen in Figure 4.2, when a small pebble is thrown into a pond, the point on the water where the small pebble first hits will receive a great amount of energy. Then, the energy will spread over the water surface and generate multiple *ripples*. In each time step, the ripples have different energy state. So, assume that we have a second pebble with the same mass and shape, and we have the ability to throw it into the water with the same force and angle, it takes the same time for the second pebble to raise the same energy state of the ripples. To accelerate this process, transferring the same amount of energy as a ripple has in a specific time step from the hitting point to a further ripple or from an inner ripple to an outer ripple, then evening the energy inside the same ripple could be a feasible approach.

The RNs implements this idea. It takes a graph with nodes and edges as input and learns how to accelerate the "energy transfer" between nodes through their connecting edges. Three assumptions are the fundamental cornerstones of Ripple Network (RN):

1. All nodes in the graph are eventually affected by nodes that have high energy state. Then they will change their energy states, which is represented by their dynamic features, accordingly.

2. Nodes whose energy state is low are far from the position where some events have happened, while those with high energy state are near such positions.

3. Nodes that have similar energy state should be grouped together. When energy is transferred to the nodes, nodes that are grouped together should be influenced with similar strength.

With these three assumptions, extra edges, termed as *ripple edges*, are generated to transfer the energy directly from one position to another position. The nodes of the graph are partitioned into two classes, which are the *influential center nodes* and *remote nodes*, regarding the strength of their concerned dynamic features:

1. **Influential center nodes** Nodes whose energy state is significantly higher than the other nodes are considered as influential center nodes, indicating that it is being influenced by the external environment (an example could be a rigid body crashed with a deformable plate).

2. **Remote nodes** All nodes excluding the influential center nodes are defined as remote nodes. Energy should be transferred to them to let them get informed of the event.

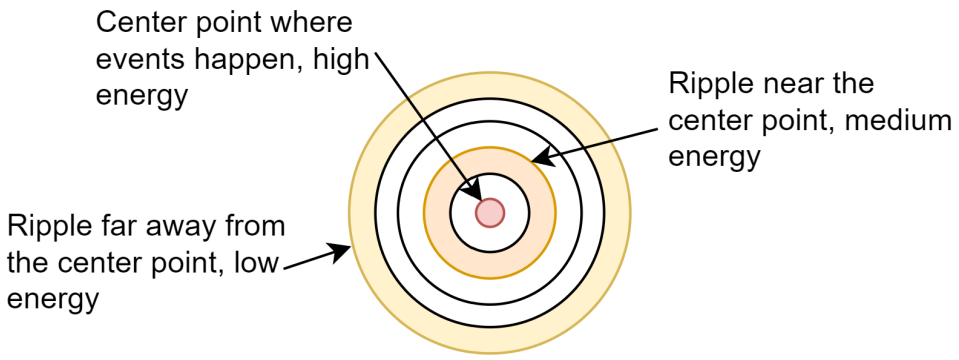


Figure 4.2: Schematic of ripples.

The influential center nodes and remote nodes are grouped into multiple *ripples*, which are the foundation of further operation of the RN. Given a graph $G = (V, E)$, ripples are defined as follows:

$$R = \{V_i \mid V_i \subset V\}$$

with

$$V = \bigcup_{i=0}^{n} V_i,\ V_i \cap V_j = \emptyset \text{ for } i \neq j,\ i, j = 1, \ldots, n,$$

The RN aims to directly propagate external dynamic changes of the influential center nodes to remote nodes so that the message passing process can be accelerated. The RN model is based on the MGN model, and is equipped with a RippleMachine which is responsible for all ripple-based operations, as shown in Figure 4.3. The RippleMachine has three basic operations. First, it parititions the graph nodes into ripples. Second, it selects nodes from each ripple for further connection. Finally, it connects the selected nodes together to form ripple edges for the RN. The RippleMachine then outputs a graph enhanced with ripple edges to the following encoder. The three operations of the RippleMachine will be presented in detail in the next three subsections.

4.1.2 Ripple Generation

As described in the previous subsection, the RN utilizes the idea inspired by water ripples and attempts to achieve a target state of the graph nodes' features in as few message passing steps as possible. Thus, the first step the RN executes is to generate the ripples. In this work, we propose and investigate three ripple generating methods, each of which can be tuned for a given task:

1. **Equal size ripple** The equal size ripple method is the most intuitive partition method. It generates ripple of the same size. Given a ripple number n, the ripple size is calculated with $\lceil \frac{|V|}{n} \rceil$. If the number of nodes can not be divided evenly by n, then the rest nodes will be inserted to the last ripple V_n. An example can be seen in Figure 4.4a. All nodes

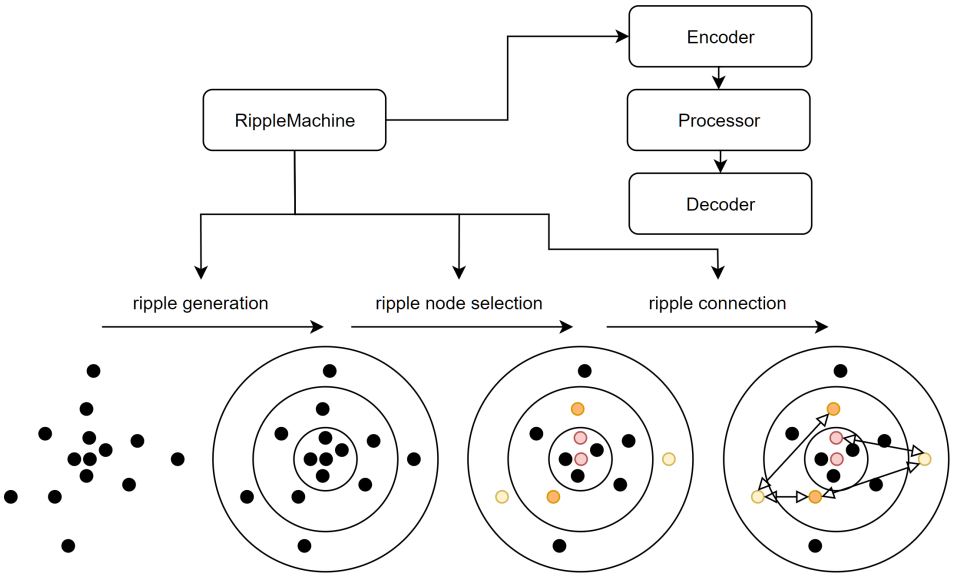
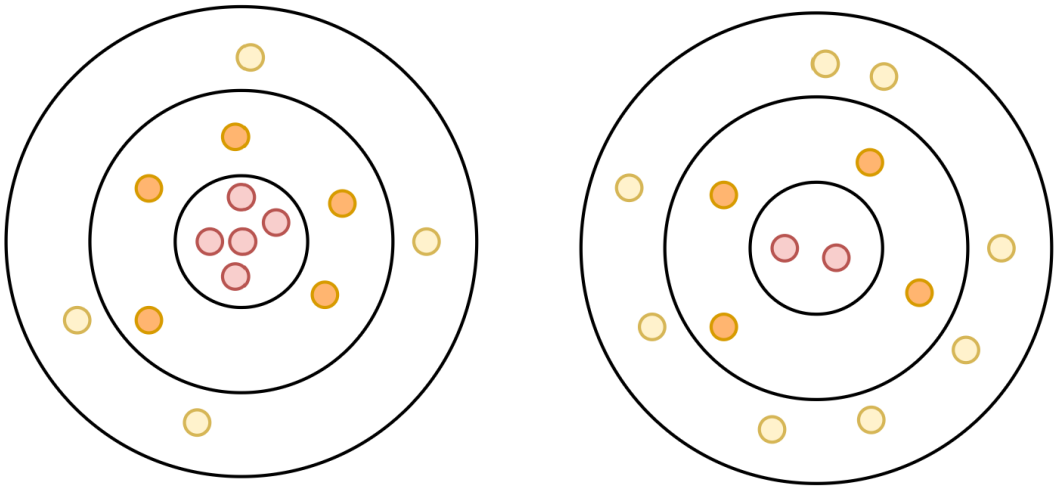


Figure 4.3: RN model architecture. The colored nodes are selected as ripple edge endpoints.

are sorted in descending order by their Euclidean norm of concerned dynamic features and allocated from inner to outer ripples, since inner ripples have a greater energy and outer ripples have a lower energy.

2. **Exponential growing ripple** The exponential growing ripple method partition the nodes into ripples whose size grows exponentially, as seen in Figure 4.4b. An outer annulus has a larger area than an inner annulus when they have the same annulus width, and thus the outer annulus can contain more nodes than the inner annulus. This thesis presents an exponential growing method for the ripple generation. It calculates a list of exponential number of a given base with an incrementing exponent starting from 1 and considers them as the ripple sizes. The base number can be specified before running the training. The nodes are allocated into each ripple to fill up the determined ripple size. If more nodes are to be allocated than the total size of all ripples, then the rest nodes are assigned to the last ripple.

3. **Histogram-aggregated ripple** The histogram-aggregated ripple aims to partition graph nodes in a way that similar nodes are clustered into the same group and the top k groups with the most members are selected as ripples, while the rest groups are merged together to form the last ripple. In such a way, nodes inside a ripple should have very similar energy than the other nodes. Therefore, they can be grouped together as a ripple and propagate their information to the other nodes. At the same time, the energy gap between different ripples should be more significant since ripples are generated according to the number of members rather than the order of the nodes' dynamic features of the previous two methods.



(a) Equal size ripple generation with 14 nodes and 3 ripples.

(b) Exponential growing ripple generation with 14 nodes and exponential base number 2.

Figure 4.4: Equal size ripple generation and exponential growing ripple generation.

Histogram is deployed in this method to aggregate the similar nodes. First, a metric is deployed to calculate the similarity based on the dynamic quantities of the nodes. After that, the nodes are inserted into the bins of the histogram according to their associative metric value. Afterwards, the top k bins, with k being specified by the user, are selected to construct k ripples. The nodes that are not selected are then group into the last ripple. The process can be summed up as the following formula:

$$r = V_{topK} \cup V_{rest},$$

where

$$V_{topK} = \{V_i \mid V_i \in topK(histo(metric(V))),\ i = 1,\ldots,k\}$$

and

$$V_{rest} = \{v \mid v \in V \text{ and } v \notin V_{topK}\}.$$

One of the methods shown above can be chosen to generate the ripples. Then, the ripple are passed to the next stage for further processing.

4.1.3 Ripple Node Selection

The task of the ripple node selection is to select nodes from each ripple, which are going to be connected with each other via some mechanisms in the next stage. RN provides the following methods for selecting ripple nodes:

1. **Random** The random method selects a specified number of nodes in each ripple randomly. If the specified number of nodes is greater than the ripple size, then all the nodes of the ripple will be selected.

2. **Top** The top method only selects a specified number of top nodes in each ripple, which means only the nodes with the highest dynamic quantities are selected. Similar to the random method, if the specified number of nodes exceeds the size of the ripple, then all the nodes are selected.

3. **All** The all method returns all nodes of each ripple.

The selected nodes and the ripples are then fed together into the next stage.

4.1.4 Ripple Node Connection

In the ripple node connection stage, the main task of RN is to connect the selected nodes from each ripple so that they can communicate directly and tranfer the dynamic quantities from the influential center nodes to the remote nodes. This connection can help to reduce the time cost caused by a more classical message passing process. The RN takes the sets of ripples and selected nodes from the previous ripple generation stage and ripple node selection stage. It then constructs ripple edges between the selected nodes and returns a modified graph. Afterwards, it utilizes one of the following connection methods to construct an edge between a node pair:

1. **Most influential node connection** The most influential node connection method is the most simple one. As seen in 4.6a, the RN takes the k nodes with the highest dynamic quantities, and creates new edges between these nodes and all other selected nodes of all ripples. In this way, $|\text{selected nodes}| - 1$ new edges will be inserted into the graph of the world state.

2. **Fully connection** The fully connection method aims to accelerate the message passing process by building new edges between all pairs of selected nodes. As shown in Figure 4.6b, a node can then propagate its dynamic information to any of the selected nodes. After the connection, $|\text{selected nodes}|^2$ of new edges will be constructed.

3. **Fully in-ripple n-cross connection** The fully in-ripple and n-cross connection method is a restricted version of the fully connection method, as seen in Figure 4.6c. To reduce the potential time cost of fully connection method, only the nodes inside the same ripple will be connected together. To enable cross-ripple message passing, a specified number of nodes termed *reselected nodes*, which are smaller than the number of already selected nodes in each ripple, are selected from the selected nodes. Then the reselected nodes are fully connected with each other to transfer information between ripples. The number of the resulting new edges is about $(n \times k)^2 + k \times (\frac{|\text{selected nodes}|}{k})^2$, where k is

the number of ripples and n is the number of reselected nodes. In short, this method is fully connection method with restricted number of connection between each ripples.

The constructed graph of the world state with extension of ripples is fed into the Encoder-Processor-Decoder network for further processing. Similar to MGN, the encoder will encode the constructed graph into a latent graph with a node encoding multilayer perceptrons (MLP) and an edge encoding MLP, then passes the latent graph to the processor. Each GN block in the processor executes a message passing step and generates a new latent graph for the next GN block. Afterwards, the final latent graph is fed into the decoder. The latent output is then decoded into a tensor of dynamic quantities for each nodes, which can be integrated by a integrator to calculate the dynamics of the node in the next time step.

4.2 Aggregation

[28] shows that Graph Neural Network (GNN) can extrapolate better in some tasks when equipped with min-aggregation method rather than sum-aggregation method. Therefore, the RN model is equipped with multiple aggregation options so that influence of different aggregation methods can be explored. Details are shown in this section.

4.2.1 Common Neighbor Aggregation Methods

The RN model has provided alternative aggregation methods than the MGN model. In MGN, the GN blocks of the processor only use *sum* method to sum up neighborhood information for a node. But in RN, three common aggregation methods are added to the model:

1. **Min** The min method finds out the minimum value for each dimension of the feature among the edges of a node and updates the node with it.

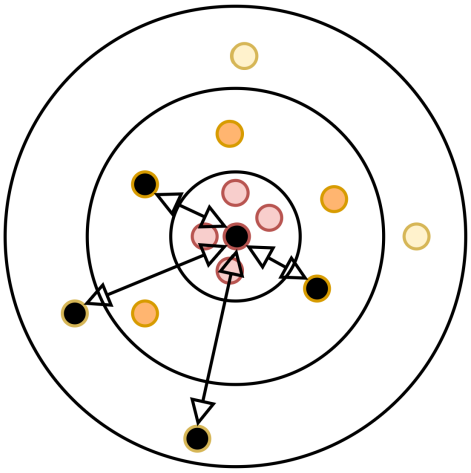
2. **Max** The min method finds out the maximum value for each dimension of the feature among the edges of a node and updates the node with it.

3. **Mean** The mean method calculates the mean value of all the edge features and updates the node with the mean edge feature.

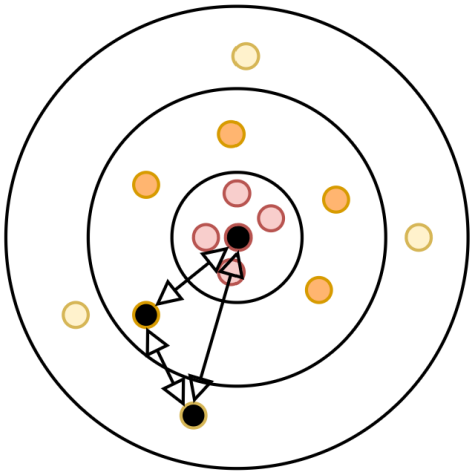
4.2.2 Principle Neighborhood Aggregation

In addition to common neighbor aggregation methods presented in the previous subsection, the thesis work has adopted a novel aggregation method called Principal Neighbourhood Aggregation (PNA)[24]. The general idea of PNA is to use multiple aggregation methods when updating nodes, then concatenate the results of different aggregators along with the

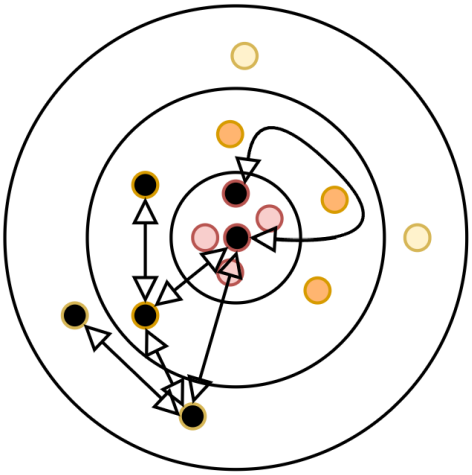
Figure 4.5: Examples of most influential node connection, fully connection and fully in-ripple n-cross connection. Black nodes are the selected nodes and the arrows are the ripple-edges.



(a) Most influential node connection, with the most influential nodes connecting with two nodes of each ripple.



(b) Fully connection, with one selected nodes in each ripple.



(c) Fully in-ripple n-cross connection, with each ripple having two selected nodes and n=1.

Method	PPI
Random	0.396
MLP	0.422
GraphSAGE-GCN (Hamilton et al., 2017)	0.500
GraphSAGE-mean (Hamilton et al., 2017)	0.598
GraphSAGE-LSTM (Hamilton et al., 2017)	0.612
GraphSAGE-pool (Hamilton et al., 2017)	0.600
GraphSAGE*	0.768
Const-GAT (ours)	0.934 ± 0.006
GAT (ours)	$\mathbf{0.973} \pm 0.002$

Figure 4.7: Inductive evaluation result of GAN on PPI dataset.

node and feed them into a MLP. More details can be seen in Section 2.4.3. According to [24], the PNA can enhance the accuracy of a graph neural network model.

In the thesis work, all of the four aggregators, sum, min, max and mean, are utilized to form a PNA.

4.3 Attention

As shown in [25], graph network with attention can improve the prediction accuracy. Figure 4.7 tells that Graph Attention Networks (GAN) can achieve a state of the art performance among the other GNN models. Since the neighborhood information of a node can be noisy when the mesh is interacting with the environment, the nodes should attend on edges which are in the direction of the influential center nodes. Therefore, the RN is equipped with attention to explore the influence of it.

The attention mechanism is deployed in the GN block of the processor in the network. When the GN block executes the message passing process, the edge of the graph will first get updated. After that, the node will be updated by aggregating the features of the updated edges. With attention, the node will only get updated after it knows how much attention it should pay to the edges.

The attention mechanism is implemented as a two-layer MLP with LeakyReLU to avoid dead cells, and outputs the softmax result as the attention weights. Such an MLP is added to each GN block of the processor. After the GN block has finished updating the edges, the attention MLP will calculate the attention weights of all the connecting edges for each node. Before a node aggregates the features of all its connecting edges, the attention mechanism will execute a Multiply–Accumulate (MAC) operation on the output weights and the edge features to compute the final edge features. Then, the node will aggregate the new features and update itself.

Chapter 5

Evaluation

The evaluation result of the Ripple Network (RN) is presented in this chapter. In the first section, hardware and software evaluation environments are listed. To save time for more experiments, evaluation is executed in two computing platforms. In the second section, details of the training and evaluation datasets of the FlagSimple task are shown. Afterwards, the training and evaluation configuration is shown in the third section. In the last section, the qualitative and quantitative evaluation results are shown.

5.1 Setup

5.1.1 Hardware And Software Environment

The evaluation is executed on two types of computing platform to save time for the large set of experiments. The configuration of each platform is listed in Table 5.1.

Platform	CPU	GPU	Memory	OS
1	AMD Ryzen 3700X	Nvidia RTX 2060 Super with 8 GB memory	32 GB	Ubuntu 18.04
2	Intel Xeon Gold 6248	NVIDIA Tesla V100 with 32 GB memory	768 GB	RHEL 8.2

Table 5.1: Evaluation platform configuration.

Data Class	Type	Shapes	Dtype
cells	static	[1, 3028, 3]	int32
node type	dynamic	[401, 1579, 1]	int32
mesh position	static	[1, 1579, 2]	float32
world position	dynamic	[401, 1579, 3]	float32

Table 5.2: FlagSimple dataset structure.

5.1.2 FlagSimple Dataset

The FlagSimple dataset is generated by the solver ArcSim[29] and composed of the training split, validation split and test split. In this thesis, the training split is used to train the network and the validation split is used to generate the evaluation rollout. The training split contains 1000 trajectories, while the validation split contains 100 trajectories.

Each trajectory contains simulation data of 401 time steps. The simulation data consists of four classes of data, which are cells, node type, world position and mesh position. Their features are listed in Table 5.2.

The type of the data class indicates whether the data class remains the same (static) or changes (dynamic) during the simulation. For example, the cells (triangular cell of the mesh) describes the topology of the mesh nodes and are assumed to be fixed throughout the simulation, so it has the type "static", while the world positions of the nodes are always changing and thus labeled as "dynamic".

The shapes of the data class is a three-dimensional feature, with the values from dimension 0 to dimension 2 indicating the corresponding time step, the corresponding node or cell, and the actual value of a specific node feature. For static data class, the values of the first time step should be copy to all the time steps.

The dtype defines the data type of the values in this data class. The available options are 32-bit integer and 32-bit floating point number.

5.1.3 Training And Evaluation Configuration

In these thesis, the experiments run with training configuration of 15 epochs. In each epoch 100 trajectories with 401 time steps are fed into the network. For evaluation, totally 100 trajectories of 401 time steps are rollout. We use an Adam optimizer with 1e-4 learning rate and a scheduler with $\gamma = 0.1 + 1e{-}6$, and trigger the scheduler once at the beginning of the 13th epoch. The training loss is calculated by L_2 loss between the prediction and the ground truth value, while the evaluation loss consists of L_1 and L_2 loss between the prediction and the ground truth value.

The experiments are executed with configuration of 5 and 7 message passing steps respectively. The experiments can be grouped into three groups. In the first group, the MeshGraphNets (MGN) model with sum, min, max, mean and Principal Neighbourhood Aggregation (PNA) aggregation methods are evaluated. In the second group, the MGN with the five aggregation methods above plus attention are evaluated. In the last group, the three ripple methods, namely the equal size ripple, the exponential size ripple and the gradient ripple, are evaluated. All ripple methods utilize sum aggregation. They generate 5 ripples and select 5 random nodes from each ripple for ripple edge connection. For ripple edge connection, the most influential node connection version and the fully in-ripple n-cross connection are used.

5.2 Quantitative Result

5.2.1 Aggregation Methods

Table 5.3 and 5.4 show the quantitative result of the five aggregation methods with 5 and 7 message passing steps respectively, normalized to the result of the original MGN with sum aggregation. The aggregation methods that achieve the best performance and their losses are printed in bold text.

Aggregation Method	L1 Loss	L2 Loss
sum	1	1
min	0.926	0.718
max	0.99	0.771
mean	1.345	1.48
pna	**0.852**	**0.591**

Table 5.3: L1 loss and L2 loss of different aggregation methods with 5 message passing steps.

Aggregation Method	L1 Loss	L2 Loss
sum	1	1
min	0.785	0.592
max	**0.762**	0.569
mean	0.816	0.623
pna	0.774	**0.567**

Table 5.4: L1 loss and L2 loss of different aggregation methods with 7 message passing steps.

We can see that the model with PNA can achieve an overall better performance than the other aggregation methods.

5.2.2 Attention

Table 5.5 and 5.6 show the quantitative result of the MGN model of five aggregation methods equipped with attention mechansim respectively. The models are evaluated with 5 and 7 message passing steps respectively, and normalized to the result of the original MGN with sum aggregation. The attention models that achieve the best performance and their losses are printed in bold text.

Aggregation Method	L1 Loss	L2 Loss
sum	12.947	104.743
min	21.314	416.089
max	9.322	71.045
mean	22.754	335.783
pna	**4.544**	**13.98**

Table 5.5: L1 loss and L2 loss of different aggregation methods with attention and 5 message passing steps.

Aggregation Method	L1 Loss	L2 Loss
sum	7.21	41.976
min	13.576	129.97
max	6.549	31.136
mean	14.653	190.421
pna	**5.278**	**22.134**

Table 5.6: L1 loss and L2 loss of different aggregation methods with attention and 7 message passing steps.

We can see that the attention mechanism does not improve the model. To be noticed is that the PNA method can still achieve the best performance and has a much greater gap against the other aggregation methods than in the situation when the model is without attention.

5.2.3 Ripple Model

Table 5.7 and 5.8 show the quantitative result of the ripple methods with 5 and 7 message passing steps respectively, normalized to the result of the original MGN with sum aggregation. The ripple models that achieve the best performance and their losses are printed in bold text.

We can see that the RN can achieve much better performance when more message passing steps are used in the model. In addition, the equal size ripple method has a more stable performance regarding different message passing steps.

Connection Method	Ripple Method	L1 Loss	L2 Loss
most influential	**equal size**	0.915	**0.686**
	exponential size	1.554	2.724
	gradient	**0.912**	0.732
fully in-ripple n-cross	equal size	1.133	1.286
	exponential size	1.062	0.901
	gradient	0.913	0.721

Table 5.7: L1 loss and L2 loss of different ripple methods with sum aggregation and 5 message passing steps.

Connection Method	Ripple Method	L1 Loss	L2 Loss
most influential	**equal size**	**0.829**	0.708
	exponential size	0.862	0.723
	gradient	1.033	1.141
fully in-ripple n-cross	equal size	0.844	0.678
	exponential size	0.834	**0.658**
	gradient	0.853	0.695

Table 5.8: L1 loss and L2 loss of different ripple methods with sum aggregation and 7 message passing steps.

5.3 Qualitative Result

In this section, the visualized evaluation results of the PNA method, equal size ripple method, exponential size ripple method and gradient method are shown. The figures are captures from the rollout video after at least 250 steps.

As it is seen in the previous section, the PNA method can achieve an overall better performance among all other aggregation methods. Figure 5.1 shows an rollout evaluation of the model with PNA method and 5 message passing steps. The orange cloth is the ground truth, while the blue one is the prediction. It can be seen that the prediction cloth remains stable and has a similar form of the ground truth after 282 steps of rollout.

Figure 5.2a, 5.2b, 5.2c and 5.2d show the visualized evaluation result of the respective ripple methods with the best performance.

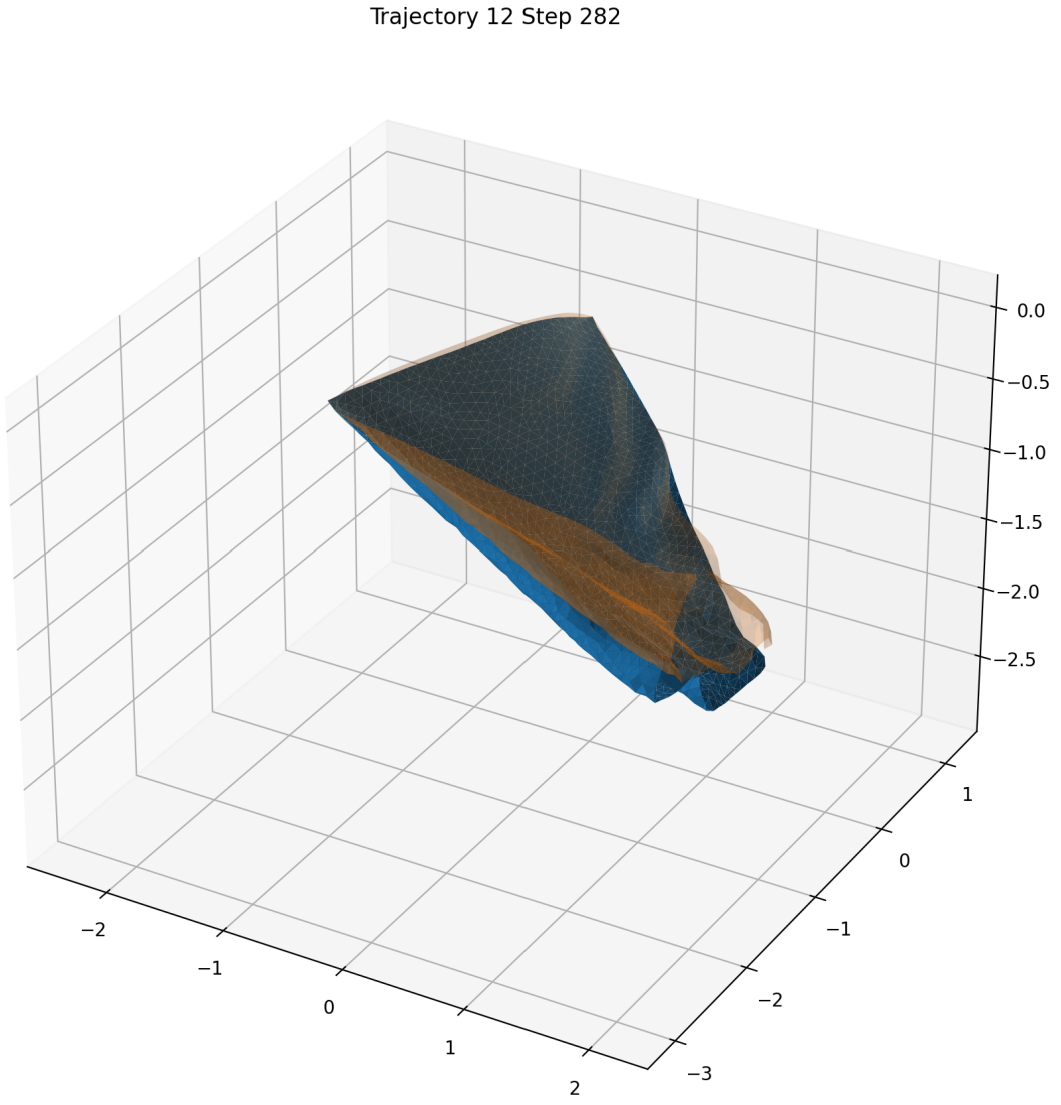
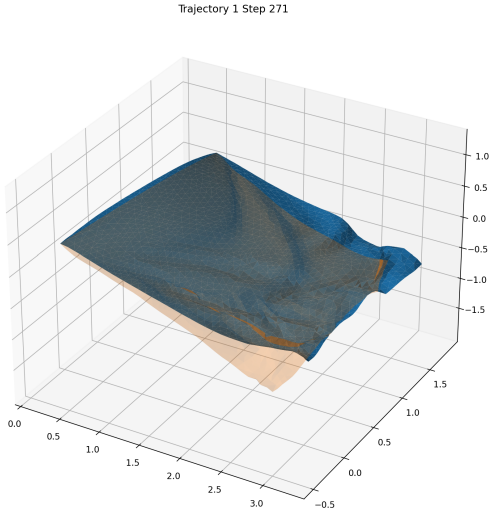
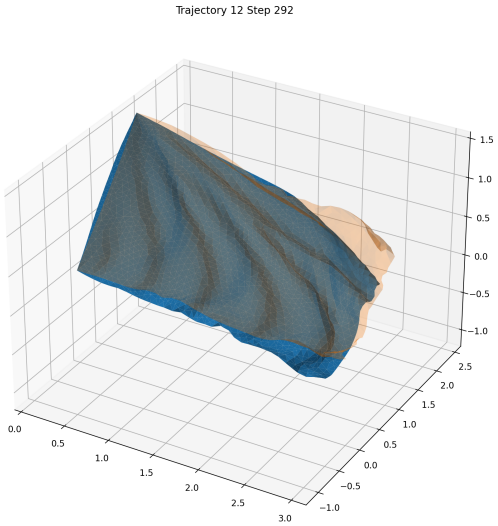


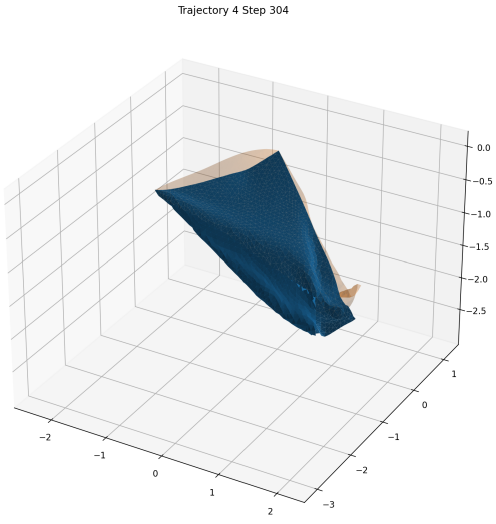
Figure 5.1: Visualized evaluation result of the PNA model.



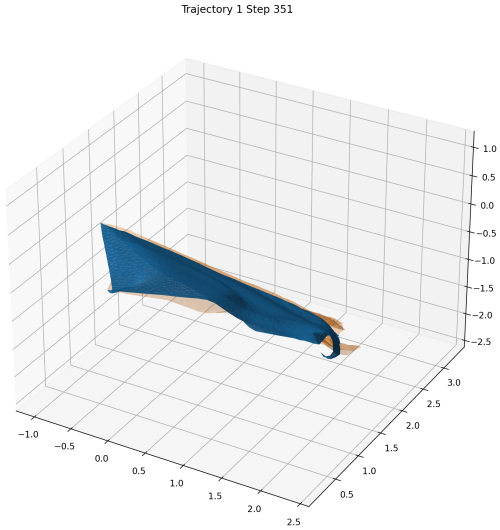
(a) Visualized evaluation result of the equal size ripple method with 5 message passing steps.



(b) Visualized evaluation result of the gradient ripple method with 5 message passing steps.



(c) Visualized evaluation result of the gradient ripple method with 7 message passing steps.



(d) Visualized evaluation result of the exponential size ripple method with 7 message passing steps.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have introduced the Ripple Network (RN) model equipped with other aggregation methods and attention mechanism in this thesis. The RN model is a deep learning model which can be deployed to handle physical simulation task. With the RN model, expensive physical simulation tasks can be executed in an efficient way.

We then perform exhaustive experiments on the model to figure out the improvement of different methods compared to the original MeshGraphNets (MGN) model. As we can see from the evaluation result, the MGN model with Principal Neighbourhood Aggregation (PNA) and the RN model can achieve state of the art performance and overwhelm the original MGN model. This thesis work provides a fundamental framework for further studying of physical simulation using deep learning technique.

6.2 Future Work

Since the time is limited for this thesis work, there are still many possible improvements of the RN for future study. They are listed in the following list.

1. The parameter space of the RN can be further explored. As the input data can represent different form of simulation tasks, the numbers of ripples, the selected nodes and the way the ripple edge is generated may be tuned to adapt to different tasks, so that the

best performance of the network can be figured out. It would also be possible to make the RN self-adaptable.

2. To further exploit the advantages of the RN, it is worthy to deploy the RN to simulation task such as object deforming.

3. Other improvement method can be added to RN to enhance its performance, such as hierarchical structural message propagation of Dynamic Particle Interaction Networks (DPI-Net).

Bibliography

[1] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations*, 2021.

[2] Cosmol website. `https://www.comsol.de/release/5.3/mesh`. Accessed: 2022-01-26.

[3] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, 2018.

[4] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, 2020.

[5] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. In *Conference and Workshop on Neural Information Processing Systems*, 2016.

[6] Yehia A. Abdel-Nasser. Frontal crash simulation of vehicles against lighting columns using fem. 2013.

[7] Junuthula N. Reddy. Introduction to the finite element method, fourth edition. 2019.

[8] Alexandr Andoni, Rina Panigrahy, Gregory Valiant, and Li Zhang. Learning polynomials with neural networks. In *International Conference on Machine Learning*, 2014.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[10] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 2018.

[11] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. In *IEEE Signal Processing Magazine*, 2017.

[12] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 2000.

[13] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. In *Journal of Machine Learning Research*, 2008.

[14] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, 2006.

[15] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, 2005.

[16] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[17] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 2009.

[18] Thomas N. Kipf and Max Welling. Variational graph auto-encoders. 2016.

[19] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. 2018.

[20] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ryGs6iA5Km`.

[21] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2016.

[22] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.

[23] Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020.

[24] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Velickovic. Principal neighbourhood aggregation for graph nets. In *Neural Information Processing Systems*, 2020.

[25] Petar Veličković, uillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems*, 2017.

[27] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B. Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation networks for model-based control under partial observation, 2019.

[28] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=UH-cmocLJC`.

[29] Rahul Narain, Armin Samii, and James F. O'Brien. Adaptive anisotropic remeshing for cloth simulation. *ACM Trans. Graph.*, 31(6), nov 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366171. URL `https://doi.org/10.1145/2366145.2366171`.

[30] Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B. Tenenbaum, and Antonio Torralba. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. In *International Conference on Learning Representations*, 2019.

Appendix A

DeformingPlate Task Reproduction Details

The work of MGN is only partially open-source. The source code for the FlagSimple task is released, while the one of DeformingPlate task, which is also an important part of this thesis, is not publicly presented. Thus, it needs to be reproduced according to the description in the paper. Since not all implementation details are presented clearly, the implementation in this thesis should have deviation against the original one, thus it does not work as expected. Here in this appendix, the details of the reproduction attempts of the DeformingPlate task are shown, so that future work on the DeformingPlate task can take this thesis work as a reference.

First, the details of the DeformingPlate dataset is shown in Table A.1. The number "-1" in the shape tensor indicates that this dimension is inferred from each individual input.

As seen from the second row in Figure A.1[1], the Hyper-El. system, which is corresponding to the DeformingPlate task, takes the node type as node feature input, the mesh position and its norm, as well as the world position and its norm as mesh edge feature input, the world

Data Class	Type	Shapes	Dtype
cells	static	[1, -1, 4]	int32
node type	static	[1, -1, 1]	int32
mesh position	static	[1, -1, 3]	float32
world position	dynamic	[400, -1, 3]	float32
stress	dynamic	[400, -1, 1]	float32

Table A.1: DeformingPlate dataset structure.

position and its norm as world edge feature input. The network then outputs a prediction of the velocity and stress of the target node for the next time step. As the stress is decoded by a different decoder than the velocity, which is also based on the same prediction quantity p, it is excluded from the experiment to make the experiment faster and make the evaluation of the network more simple.

System	Type	inputs $\mathbf{e}_{ij}^M$	inputs $\mathbf{e}_{ij}^W$	inputs $\mathbf{v}_i$	outputs $\mathbf{p}_i$	history h
Cloth	Lagrangian	$\mathbf{u}_{ij}, \lvert\mathbf{u}_{ij}\rvert, \mathbf{x}_{ij}, \lvert\mathbf{x}_{ij}\rvert$	$\mathbf{x}_{ij}, \lvert\mathbf{x}_{ij}\rvert$	$\mathbf{n}_i, (\mathbf{x}_i^t - \mathbf{x}_i^{t-1})$	$\ddot{\mathbf{x}}_i$	1
Hyper-El.	Lagrangian	$\mathbf{u}_{ij}, \lvert\mathbf{u}_{ij}\rvert, \mathbf{x}_{ij}, \lvert\mathbf{x}_{ij}\rvert$	$\mathbf{x}_{ij}, \lvert\mathbf{x}_{ij}\rvert$	$\mathbf{n}_i$	$\dot{\mathbf{x}}_i, \sigma_i$	0
Incomp. NS	Eulerian	$\mathbf{u}_{ij}, \lvert\mathbf{u}_{ij}\rvert$	—	$\mathbf{n}_i, \mathbf{w}_i$	$\dot{\mathbf{w}}_i, p_i$	0
Compr. NS	Eulerian	$\mathbf{u}_{ij}, \lvert\mathbf{u}_{ij}\rvert$	—	$\mathbf{n}_i, \mathbf{w}_i, \rho_i$	$\dot{\mathbf{w}}_i, \dot{\rho}_i, p_i$	0

Figure A.1: Training input and output of MGN for different tasks. The Hyper-El. system is corresponding to the DeformingPlate task.

The paper just tells that world edges $\mathbf{e}_{ij}^W$ are only generated between two nodes when the world distance of the node-pair is smaller than $r_W = 0.03$. But when the thesis work implements the design following this description, the large evaluation experiment (setting is shown below) does not succeed on Nvidia RTX 3080 graphics processing unit (GPU) with 10 GB memory, because the program raises "CUDA out of memory" error. The reason for this error is that too many world edges are generated for the current input. The qualitative evaluation of small experiment (setting is also shown below) shows a low-quality result. For example, Figure A.2 shows that the plate is deformed in a wrong direction and the deformation is slight compared to the ground truth. Therefore, three attempts to improve the qualitative evaluation are conducted. Some common settings of the three attempts are first listed below:

1. Training and evaluation are executed on a single Nvidia RTX 3080 GPU with 10 GB memory.

2. Experiments are trained on train split of the dataset. In large experiment, training is executed with 15 epochs, 100 trajectories in each epoch, and 15 message passing steps, and evaluation is executed with 100 trajectories. If "CUDA out of memory" error occurs, small experiment with 3 epochs, 20 trajectories in each epoch and 7 message passing steps is conducted, and evaluation is executed with 10 trajectories.

3. Evaluation is executed on valid split of the dataset with 100 trajectories.

4. World distance between nodes are calculated by *torch.cdist* function.

5. Self-connection world edges are excluded since nodes do not have any interaction with itself.

6. *torch.nonzero* function is used to find out the senders and receivers of the generated world edges.

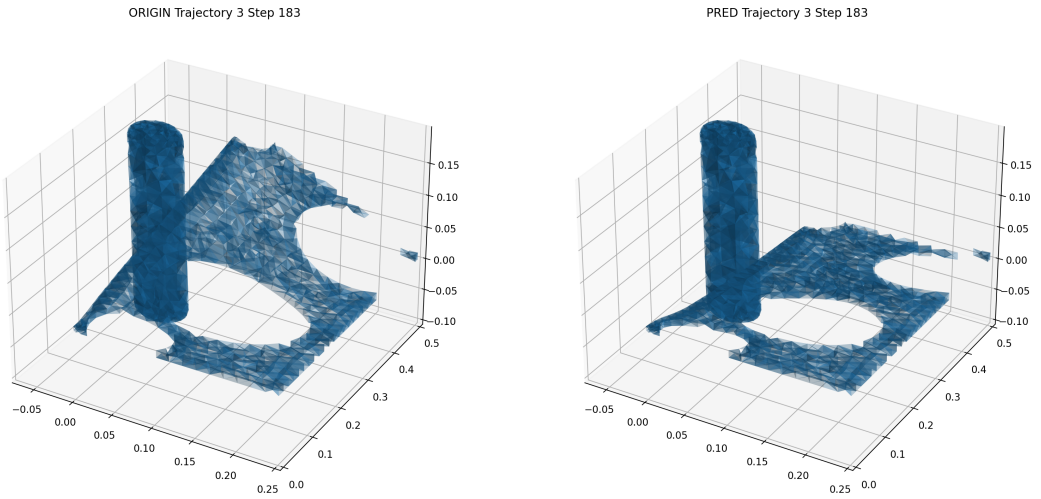


Figure A.2: Small experiment evaluation result of original model. ORIGIN refers to the ground truth of a trajectory from the evaluation dataset, PRED refers to the prediction based on the first time step of the that trajectory.

The first two attempts aim to reduce number of world edges so that less GPU memory is needed. In the first attempt, the world distance r_W is reduced from 0.03 to 0.005 so that less world edges are generated during the contact process of the moving rigid body and the deformable plate. After the world distance reduction the large experiment can be finished without any error, but in the qualitative result the model performs poorly. Figure A.3 shows one of the visualization result of the modified model. The deformable plate is not deformed sufficiently and even stops from deforming when the moving rigid body is already inside it.

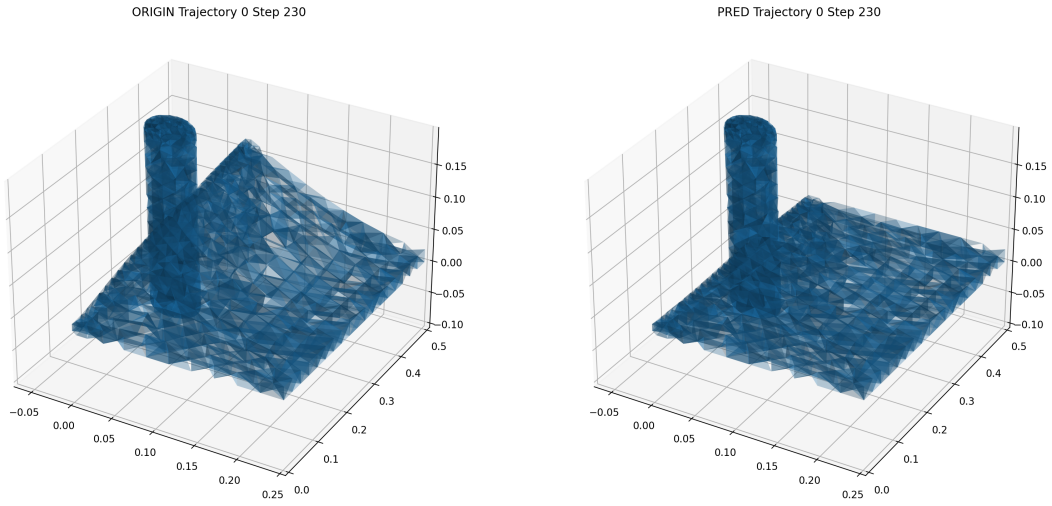


Figure A.3: Large experiment evaluation result of reduce radius model.

In the second attempt, inpired by the RiceGrip task configuration of the [30], world edges are only generated between obstacle nodes and nodes of the deformable plate, which have

the type normal and handle. The obstacle nodes always act as senders and nodes of the plate act as receivers. As self-collision does not actually occur in this task, world edges between normal nodes are also excluded, so that GPU memory consumption can be further reduced. The visualization of the large experiment evaluation result is shown in Figure A.4. It has the same problem as the first attempt.

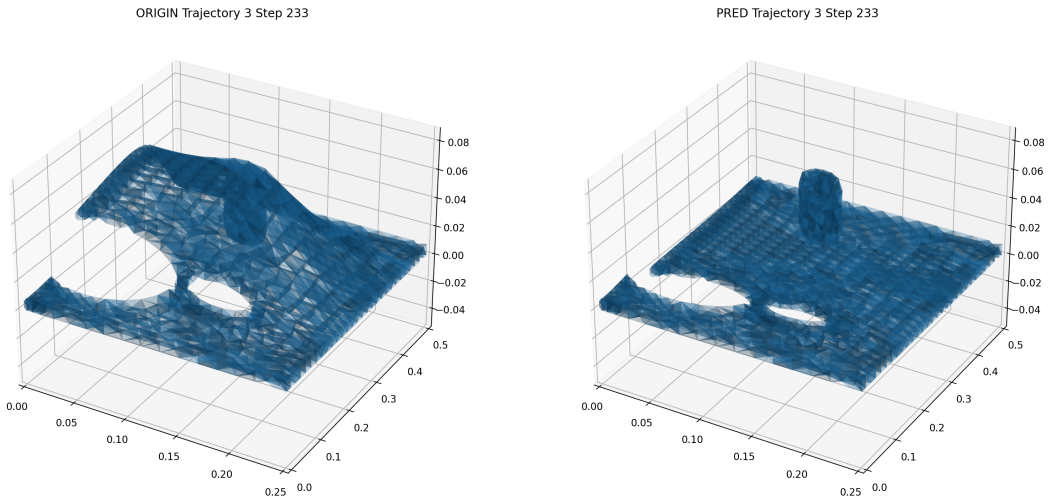


Figure A.4: Large experiment evaluation result of unidirectional edge model.

In the third attempt, world edges are generated as in the second attempt, and then only the world edge with the minimum norm of its length is reserved among all world edges of the same edge receiver. But as seen in Figure A.5, the problem of the last two attempts still exists.

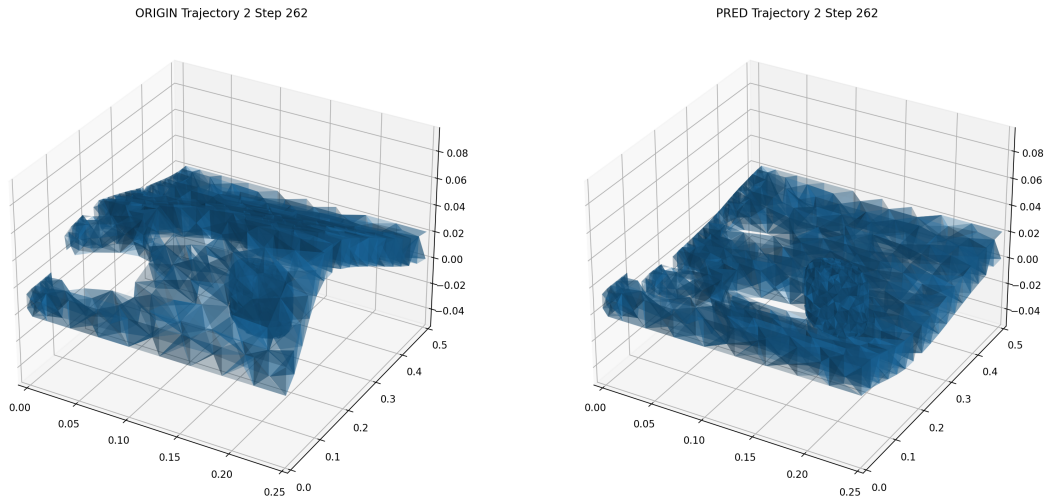


Figure A.5: Large experiment evaluation result of shortest edge model.

Model	L_2 loss
original model	0.000224412
reduce radius model	0.001181771
unidirectional edge model	0.000389328
shortest edge model	0.000308932

Table A.2: Losses of the four models.

The losses of all 100 trajectories in the evaluation dataset are calculated by L_2 loss between the evaluation rollout and the ground truth. The losses of the original model and its three variants are listed in Table A.2 for reference.