Data Driven Discovery of Physical System Dynamics via Graph Neural Network Simulations

Cliff B. Abbott

University of Colorado Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80918

Abstract

Graph Neural Networks (GNNs) have recently proven to be extremely capable at learning to simulate real physical systems. With a common architecture, the GNNs can learn the underlying physical principles guiding the evolution of the system. These physical principles can be expressed as Partial Differential Equations (PDEs). While there has been much attention given to identifying the coefficients of hard coded PDEs through simulated machine learning, there has not been much attention given to the idea of extracting the PDEs themselves. In this work, we used GNNs trained on several physical systems to try to identify the underlying physics of a system. We explore several different architecture types to analyze the GNN parameters and attempt to identify which system the GNN was trained on. We then break the systems up into a vector of components of their PDEs and attempt to get more granular information about the governing equations.

1 Introduction

The basic purpose of physics is to create accurate, useful models of the world around us. A theory is only as good as its ability to predict real world outcomes. For the majority of human history, the process has been to observe a physical system or interaction, attempt to make an informed decision about the underlying physical principles, create a mathematical model that encapsulates those principles, and see if the model can accurately predict the outcome of unseen data.

Once we have a general model for types of interactions, the task is often to build a specific model for a given system. This involves identifying the unique parameter values that tailor the general model to the specific system. In recent years, the use of machine learning models to aid in this task has grown in popularity. Researchers use Physics Informed Neural Networks (PINNs) to solve the Partial Differential Equations (PDEs) for a given system (Cuomo et al. 2022). The PINNs are models that have the underlying physical principles hard coded into the model. Data from experiments is then fed into the model and the system is trained to identify the coefficients associated with the hard coded PDEs. However, these models are generally tailored to a specific task, such as parametric PDEs (Vadeboncoeur et al. 2023; Takamoto, Alesiani, and Niepert 2023), or a specific interaction (Lu et al. 2022, 2021).

Graph Neural Networks (GNNs) are a type of machine learning model that relies on message passing between connected nodes (Scarselli et al. 2009). In the last few years, GNNs have been used to accurately model a wide variety of physical problems using a generalized architecture that does not require hard coding of the underlying physics (Sanchez-Gonzalez et al. 2020; Allen et al. 2023). These models, without changing the model structure, have been able to model fluid dynamics, rigid model interactions, semi-rigid model interactions, and various combinations at the same time. Because the GNN is capable of encoding the underlying physics of various systems into the learned parameters of the message passing update, it may be possible to extract information about the physics from the model itself. The goal of this work is to do exactly that. Given data about a physical system, we will build a GNN model to accurately represent it and then attempt to discover what physical principles are guiding the evolution of the system. This would be a useful methodology in situations where other direct forms of measurement may be unavailable. For example, the mars rover may capture an interesting interaction on video, but is unable to test the components of the reaction for their net magnetization or recreate the interaction to test additional hypothesis. With this method, we would be able to determine attributes of the components (such as magnetization) by simple video analysis.

2 Background

At a basic level for physical simulations, GNNs consist of points called nodes $v_i = \{x_i, p_i, ...\}$ that contain information about part of the system, such as position or momentum or a particle. A region of connectivity is defined around each node and a connection, called an edge $e_{i,j} = \{x_i - x_j, p_i - p_j, ...\}$, is made to every other node within that area as shown in figure 1. Edges contain information about the connection between the two nodes, such as relative position $(x_i - x_j)$. The collection of nodes and edges at any time t is called a graph $\boldsymbol{G^t} = \{\boldsymbol{V^t}, \boldsymbol{E^t}, u^t\}$ where u^t is a graph level embedding that incorporates global proprieties such as a potential field (i.e. gravity). At each update iteration, the model goes through 3 processes: $Encoder$, $Processor$, and $Decoder$. The $Encoder$ embeds the node, edge, and global

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

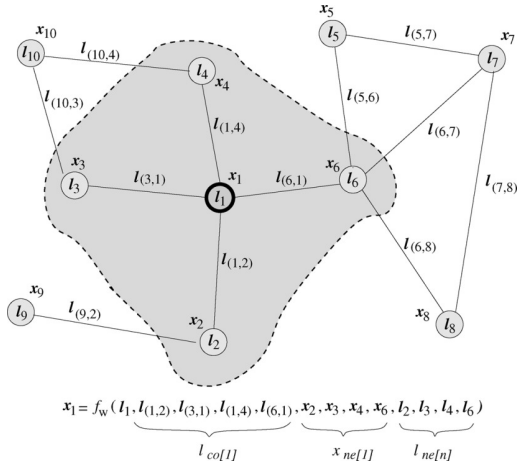

Figure 1: Example of a GNN graph. Nodes are circles labeled by l_i and edges labeled $l_{(i,j)}$. The grey area is the area of connectivity defined for l_1 by which edges $l_{(1,i)}$ are created. Illustration taken from (Scarselli et al. 2009).

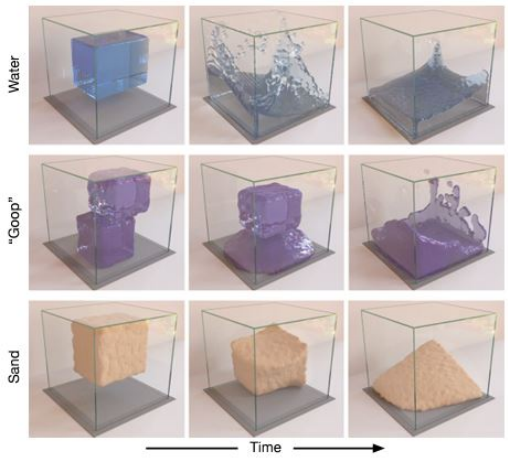

Figure 2: Simulations of various materials done by the GNN from (Sanchez-Gonzalez et al. 2020)

information into a latent graph. The nodes and edges are embedded using a learned function, $\boldsymbol{V_i} = \varepsilon^v(v_i)$ and $\boldsymbol{E_i} = \varepsilon^e(e_i)$ respectively. The $Processor$ exchanges information between nodes via message passing along the connected edges. The message passing process is another learned function and the number of times the message passing is repeated is a tunable hyperparameter. The processor updates the latent graph such that $\boldsymbol{G^f} = M^m(\boldsymbol{G^i})$ if M is the update operator applied m times. Finally, the $Decoder$ process extracts information about the dynamical updates to the nodes via another learned function $y_i = \delta^v(v_i^f)$. The updates are applied to the nodes $v_i^{t+1} = y_i(v_i^t)$ advancing the simulation by one time step. Collectively all the learned parameters are identified as θ such that $\boldsymbol{V^{t+1}} = Update(\boldsymbol{V^t}, \theta)$. Models are typically trained on single time steps. A full simulation, called a roll out, can be done by taking an initial system starting point and feeding the model prediction back into the model as the input for the next time step. However, error will accumulate with each time step introduced.

3 Related Work

While the use of GNNs for physical simulation is becoming more common, we will primarily be following the work of the DeepMind, UK group. Using a graph inspired neural network structure, they first introduced their Interaction Network in (Battaglia et al. 2016). The Interaction Network was capable of simulating several simple systems on its own, such as orbits or particles in a box. Battaglia et al. (2018) built on this work and introduced the current framework for what they call Graph Network-based Simulators, which use an Interaction Network as the $Processor$. They set the theoretical foundation for relational reasoning and inductive bias, argued for the need for artificial intelligence frameworks and architectures that were better at learning relational inductive biases, and laid out the general format of their proposed graph network. Sanchez-Gonzalez et al. (2020) demonstrated the capability of their design to simulate several complex interactions. Using the exact same architecture for all models, they accurately simulated fluid dynamics with a range of different material properties from liquid to sand, as well as multiple different materials at the same time. One of the major discoveries in this paper was that the authors found they could train the model on a small simulation size and time step, then accurately simulate significantly more particles and over much longer time periods than the training data. Most recently, Allen et al. (2023) showed that this architecture extended well into the solid regime. While the previous work was able to simulate a rigid cube in water much better than other simulation software, the authors noted that large forces and discontinuities (such as hitting a boundary) tended to cause unwanted deformation to the rigid object. They overcame this by adding information at the edges about the difference between the original undeformed mesh and the current mesh state. This gave the model a factor that it could learn to minimize in order to keep the rigid shape.

To the author's knowledge, no group has tried to extract information about the governing PDE's or dynamics of a system from a GNN simulation.

4 Methodology

While the DeepMind group provided all the code necessary to directly implement their GNN design, we built a GNN network from scratch with PyTorch. The main reason for this was that their implementation was structured to be changed for each type of simulation implemented. As such, a GNN simulating a purely fluid system would have a slightly different number of parameters compared to one with rigid bodies. We needed to alter the structure in a way that there was always the same amount of parameters in all the layers regardless of the system being simulated. While attempting to alter their code, we experienced numerous issues around versioning, depreciated functions, and unresolved value type errors. We already had a basic framework for a GNN built so we

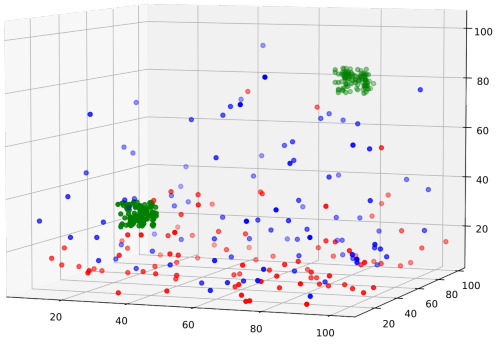

Figure 3: Example of the generated data used for GNN training. A diffusion simulation where the green clusters are the original particle clusters. Each cluster is given a different diffusion strength and a global gravitational field is applied. The red and blue dots are the two clusters after the simulated time frame.

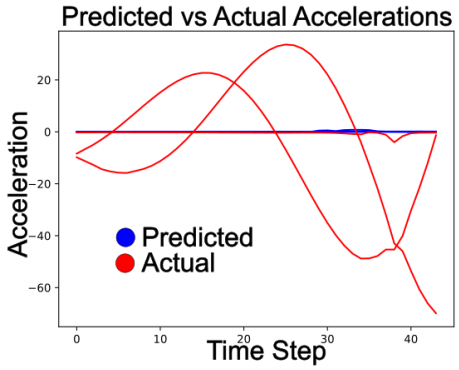

Figure 4: An example of one of the GNN simulator results vs actual values. Predicted accelerations are in blue. Actual accelerations are in red. The GNN failed to capture the behavior in a useful manner.

adopted that for this work. The reference work did not provide the data directly but instead provided data generators built from various existing python libraries. We ultimately used Mujoco and Pycharge libraries to generate simulation data for each GNN model (ex. fig 3).

We started by constructing the framework for a GNN based on the design in (Sanchez-Gonzalez et al. 2020). This consists of creating node and edge encoders, ε^v and ε^e, using single layer dense networks with 128 output nodes. In their research, they primarily used multi-layer perceptrons (MLPs) with 2 hidden layers and ReLU activation functions. However, they noted that changing the encoders to single layer networks made very little difference on the results. The Interaction Network, which is the method of message passing, consists of a series of MLPs with 128 output nodes and summation functions. The decoder is also a MLP but with 3 output nodes which correspond to the 3 dimensional accelerations of that particle for that time step. All MLPs contain 2 hidden layers, ReLU activation functions, and 128 nodes per layer with the exception of the decoder. All node and edge information will be relational rather than absolute (i.e. the positional coordinates of the node are not given at all). For construction and testing of the GNN, a small portion of data from the rigid box simulation was used. Because the rigid structure requires a reference distance along its rigid edge, it has 1 more edge value than the other systems. This value was zeroed out for systems without rigid structures. That made this construction the most adaptable to a wide variety of systems without changing the structure and number of parameters.

After we had a working GNN model class, the first task was to try to simply differentiate between simulated materials based on learned parameters. This was done by training several GNN models on 3 different simulations (a rigid box, charged particles in a magnetic field, and a simple particle diffusion). Using different seeds and different segments of training data, 400 models were generated for each simulation. This generated 311,347 parameters per GNN. In each model, each node had 16 attribute parameters. For the analysis, we took an average across all the nodes and passed a single set of 16 parameters as attribute parameters. Each GNN was trained for only 1,000 epochs due to time constraints while the reference work trained their GNNs for 2 million epochs. The result was that our GNNs were completely useless as simulators (fig 4). A few models were taken out to 5,000 epochs and continued to improve but the difference in performance was negligible compared to the increase in training time so additional training was not pursued. Despite this, we were hopeful that enough information about the systems was being learned that we would be able to discriminate between them during the next phase of the project.

5 Analysis: ID Network

Due to the limited amount of data, we used a 5-fold cross validation for evaluation. 3% Noise was added to the data to try to produce as much variety in the training samples as possible and increase the robustness of the model. A Cross Entropy Loss was used and an initial learning rate of 0.005 for 10 epochs. After the first 10 epochs, the parameters for the best performing epoch were loaded and a learning rate of 0.001 that scaled linearly to 0.0001 during the next 200 epochs was used. If the validation loss did not improve for 20 epochs, the training was stopped early. This training scheme was used in all the models experimented with for the $ID\ Network$.

5.1 Multi Layer Perceptron

For our initial attempt at an $ID\ Network$, we simply took each set of MLP parameters from the GNN (total of 20) and passed each one through a Linear layer reducing the nodes down to 50 each (10 for the universal encoder and attributes). We then grouped the encoders, the message passing layers, and the decoders into their own Linear layers. From there all layers were combined into one more hidden layer and then into a final outplay with 3 nodes. A ReLU activation was used in between each of the hidden layers and a softmax on the output layer (table 1).

Input Layer	Hidden Layer	Hidden Layer	Output Layer
Attributes $16 \rightarrow 10$ Universal $16 \rightarrow 10$ Node Encoder $5248 \rightarrow 50$ Edge Encoder $768 \rightarrow 50$	Encoder Layer $120 \rightarrow 20$		
Phi Edge 1 $51328 \rightarrow 50$ Phi Edge 2 $16512 \rightarrow 50$ Phi Edge 3 $16512 \rightarrow 50$ Phi Edge 4 $16512 \rightarrow 50$ Phi Node 1 $34944 \rightarrow 50$ Phi Node 2 $16512 \rightarrow 50$ Phi Node 3 $16512 \rightarrow 50$ Phi Node 4 $16512 \rightarrow 50$ Phi Univ 1 $34944 \rightarrow 50$ Phi Univ 2 $16512 \rightarrow 50$ Phi Univ 3 $16512 \rightarrow 50$ Phi Univ 4 $16512 \rightarrow 50$	Message Layer $600 \rightarrow 40$	Linear $90 \rightarrow 10$	Softmax $10 \rightarrow 3$
Decoder 1 $16512 \rightarrow 50$ Decoder 2 $16512 \rightarrow 50$ Decoder 3 $16512 \rightarrow 50$ Decoder 4 $387 \rightarrow 50$	Decoder Layer $200 \rightarrow 20$		

Table 1: Architecture of the simple MLP $ID\ Network$.

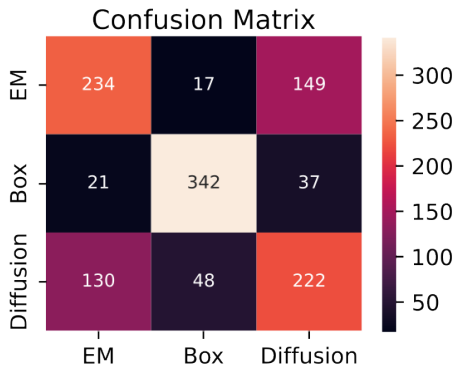

Figure 5: Confusion Matrix for the ID Network. The simulations with rigid edges were clearly differentiable. However, the model struggled with differentiating between the diffusion and electric particle simulations.

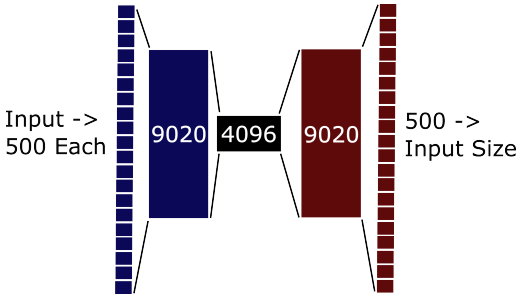

Figure 6: Encoder-Decoder architecture used for the creating of the latent representation of the system. This representation was then used as the input for the ResNet Convolutional model.

Despite the poor quality of our GNN's, the MLP model achieved a 70.6% average 5-fold cross validation accuracy. Different numbers of layers, size of hidden layers, and addition of normalization layers was experimented with but the initial MLP architecture performed the best. Looking at the confusion matrix for the MLP (fig 5) we see that this model did very well at differentiating the rigid simulation but struggled to differentiate between the two free body (EM and Diffusion) simulations.

5.2 Convolutional Neural Network

For the convolutional model, we used the ResNet50 architecture as our base model. While not the current state of the art, ResNet50 provides a good balance between performance and training time. For our initial attempt at a Convolutional ID Network, we passed the parameters from each GNN layer through a dense layer to reduce dimensionality and then combined them to form a 3 dimensional image which was passed to ResNet. The intuition was that the dense layers would learn to form an image that was useful to ResNet for classifying the system. However, this methodology failed and the best accuracy achieved was only 37%, barely above random chance. As an attempt to improve this architecture, we built an encoder-decoder model (fig 6) on the GNN parameters to build a latent representation of the system. This representation was then passed to ResNet instead of using the initial dense layers. While this effort did improve the performance, the convolutional approach was still only able to achieve a 40.8% accuracy.

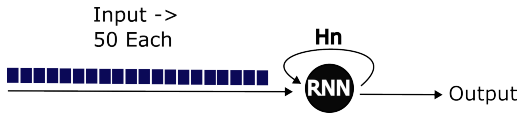

Figure 7: Simple RNN architecture. Each layer of the GNN was passed through a dense layer to reduce the vector size to a uniform length of 50. The layers were then passed sequentially through the RNN as they would be applied in the GNN.

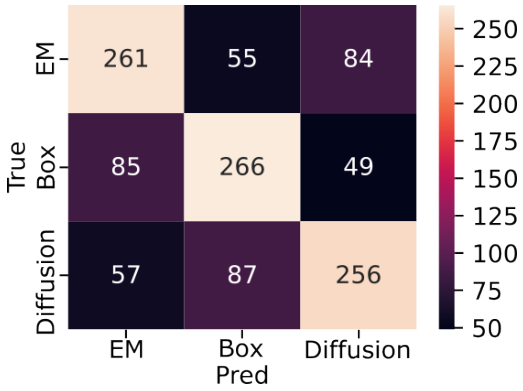

Figure 8: Confusion Matrix for the RNN model.

5.3 Recurrent Neural Network

The message passing phase of the GNN is a sequential process. As such, it would be expected that a recurrent architecture, designed specifically for sequential processing, would perform well in this task. The GNN layers were first passed through dense layers, similar to the input layer shown in table 1, to reduce the vector size down to a uniform number. These uniform vectors were then passed sequentially into the RNN model. The final output from the RNN was run through an activation function and then a final dense classification layer with SoftMax output. For this architecture, we experimented with different vector sizes, number of recurrent layers, activation functions, and depths of classification layer. The best performing RNN (fig 7) had a vector size of 50, a single recurrent layer, ReLU activation, and a single dense layer for classification. This model was able to achieve a 65.3% accuracy. As can be seen on the confusion matrix for this model (fig 8), the RNN has a fairly balanced error in contrast to the MLP model which favored the rigid simulation.

Because the RNN model performed so well, we expected using an LSTM to bring further performance gains. However, the best performance we could achieve with an LSTM was 55.3% using a 100 vector length and single layer. As with the RNN, we explored different numbers of layers, vector sizes, activation functions, and output classification layers. The LSTM generally always under performed a similar RNN structure.

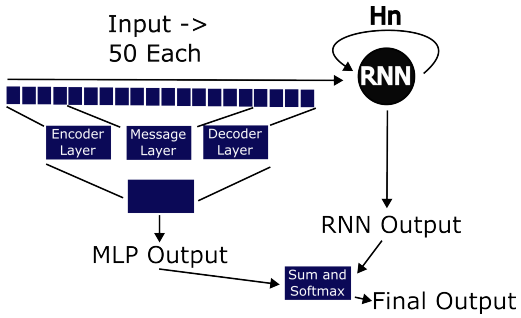

Figure 9: Architecture for the MLP and RNN parallel voting model. The GNN layers are passed through dense layers as was done in the RNN structure, but now they are passed to both the MLP and RNN models simultaneously. The results of each model are summed in a weighted fashion using another dense layer and passed through a final SoftMax.

5.4 MLP-RNN Voting Model

Recognizing that the MLP and RNN models both performed reasonably well but in different respects (the MLP with distinguishing rigid simulations and the RNN performing better on the free body simulations), we created a parallel architecture that used both models simultaneously (fig 9). For this model, the GNN layers were passed through dense layers to create a uniform vector length as was previously done with the RNN model. These uniform vectors were then passed to RNN and MLP models separately. The individual models were then run completely independently all the way up to their SoftMaxed output. The outputs from the two models were then added together and a final softmax was applied. This created an evenly weighted vote from each model on the final classification and managed to improve the overall accuracy to 74.3%. However, the MLP was only good at predicting rigid body simulations, so giving it an even weight meant that it's confusion with the free particle simulations was detracting from the overall performance. This was very clearly seen when a simple dense layer was added which took the 3 class probabilities from each model (6 total inputs) and added them in a weighted fashion into a single output. This small change had a major impact and resulted in a 91.7% overall accuracy (fig 10).

5.5 Transformer

The transformer has been a very dominant successor to the recurrent model. In most applications where the input data is sequential, like in Natural Language Processing tasks, the transformer greatly out performs RNN models (Karita et al. 2019). The only major downside to the standard transformer architecture over RNNs is the potentially limited context window. However, for this application, a long running context window is unnecessary. As with the RNN model, the GNN layers were first sent through a dense layer to create uniformly sized vectors for the transformer.

The uniform vectors are then sent to a transformer encoder only model (fig 11). For this model, we experimented with the number of attention heads, encoder layers, and the

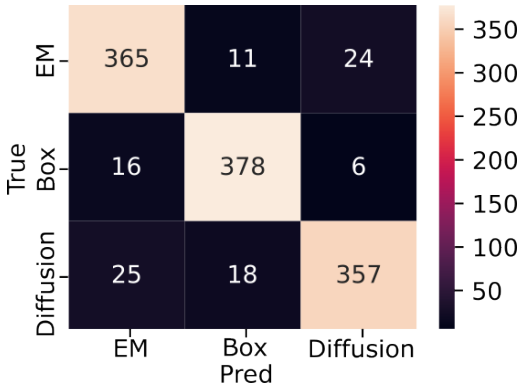

Figure 10: Confusion Matrix for the MLP-RNN Voting model.

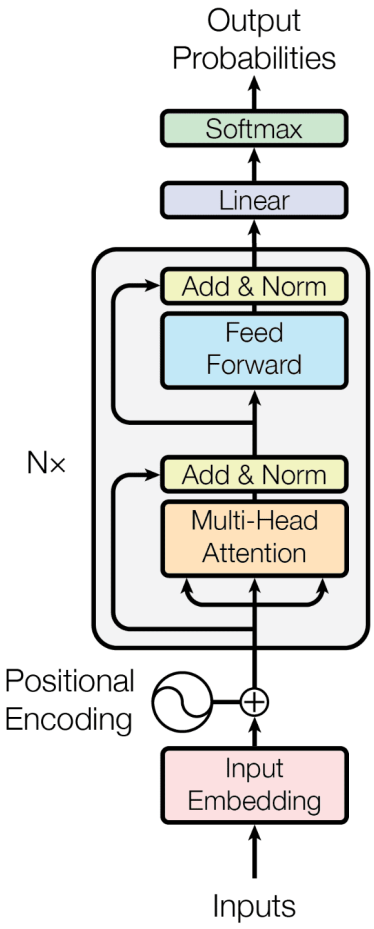

Figure 11: Transformer Encoder only model architecture.

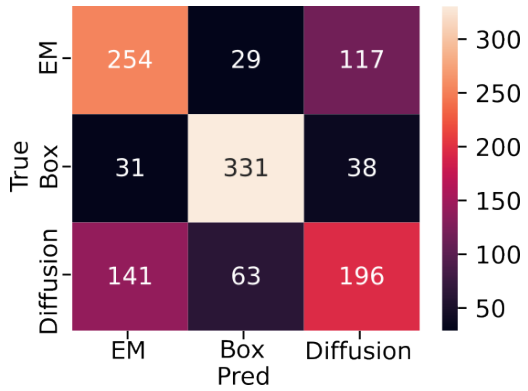

Figure 12: Confusion Matrix for the Transformer architecture with an input vector size of 10, 1 attention head, and 1 encoding layer.

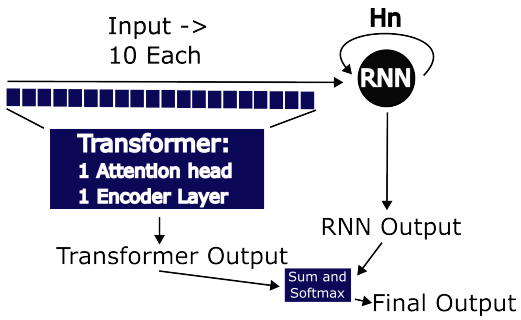

Figure 13: Architecture for the Transformer and RNN parallel voting model. The structure is the same as the MLP-RNN Voting model except the vector size has been reduced to 10 and the MLP replaced with the transformer.

vector size of the input layers. The transformer seemed to suffer from a vanishing gradient problem as the complexity increased. At more than 1 attention head and/or encoding layer, the model would often initialize with a zero gradient and be stuck at 33% accuracy. As the complexity came down, the transformer had a higher probability of training successfully. With an input vector size of 50, as was used in the RNN and MLP-RNN Voting model, the transformer was able to achieve an average accuracy of 64.3% on folds where the model parameters initialized in a trainable configuration. However, around 1/4 of the time, the model would initialize in a state with an untrainable zero gradient. After significant testing, we found that using a vector size of 10 resulted in the model initializing in a trainable state almost 100% of the time and the mean accuracy was 65.1%. Similar to the MLP, the transformer was better at discriminating the rigid box example (fig 12) but was less confident on the free particle simulations.

5.6 Transformer-RNN Voting Model

With the success of the MLP-RNN Voting model, the final architecture tested was a parallel voting model with the transformer and RNN (fig 13).

Like the MLP, the transformer was best at discriminat-

ing the rigid body simulation and the addition of the RNN should give further clarification on the free body simulations, just as with the MLP-RNN Voting model. However, the Transformer-RNN Voting model only achieved 64.6% accuracy, which was worse than both the RNN and transformer individually and significantly worse than the MLP-RNN Voting model.

6 Vectorization of Labels

With a model architecture that can successfully discriminate the different systems available, the final step that we took was to turn the class labels of "EM", "Rigid Box", and "Diffusion" into a simple vector of PDE components to see if further detail could be extracted from GNN simulators. The labels were broken into vectors of [Rigid Length, Spatial Derivative, Curl Operator, Time derivative].

	Rigid	$\frac{\partial}{\partial x}$	$\nabla \times F$	$\frac{\partial}{\partial t}$
EM	0	0	1	1
Rigid Box	1	0	0	1
Diffusion	0	1	0	0

All simulations also have a global potential component but since that was shared across all simulations, it was dropped from the vector components to avoid inflating the final accuracy with a meaningless distinction.

The MLP-RNN Voting model was initially used from the previous section with the only change being the number of output vectors being increased to 4 and the loss function being changed to the Mean Squared Error Loss. The output vector was considered a positive prediction (value of 1) for that component if the corresponding value in the vector was greater than 0.5. Otherwise it was counted as a prediction of 0. Accuracy was counted for each component within the vector (i.e. if 3 of the 4 components were correct but 1 of the components was incorrect, that sample contributed a 75% accuracy). Using this metric, the MLP-RNN Voting model achieved a 76.2% accuracy at identifying the main components of the corresponding PDEs for each simulation. Accuracy of the entire set of components was also checked. For this test, the result was only considered accurate if all the predicted components of each simulation exactly matched the truth values. For this, the model achieved a 57.8% accuracy.

In addition to the MLP-RNN Voting model, the Transformer-RNN Voting model was also tested on the component vector labels. This model achieved a 78.8% on the component accuracy and 65.5% on the full component set matching test. Surprisingly, despite significantly under performing the MLP-RNN Voting model on the single class label tests, the Transformer-RNN Voting model outperformed on the component vector tests. Only these two models were tested on the component vector labels due to time constraints.

7 Evaluation

All of the GNN models used were trained minimizing Mean Squared Error losses of the predicted accelerations compared to the actual accelerations. Roll out accuracy, where the GNN predicts the entire simulation using previous predictions as inputs for each time step, was not evaluated for the GNN models. The ID network was trained with a Cross Entropy Loss when performing the initial class discrimination testing. For final performance measurement, a simple accuracy measurement of correct vs incorrect primary prediction was used. Results for the class discrimination testing are summarized in table 2. For the component vector labels, the loss function for training was switched to a Mean Squared Error loss. A simple accuracy measure was again used for both matching the individual components and the full set of components. Results for the component vector labels are summarized in table 3.

Model	Accuracy
MLP	70.6%
ResNet50	37.0%
ResNet50 w/ Encoder	40.8%
RNN(50)	65.3%
RNN(100)	58.7%
LSTM(50)	48.1%
LSTM(100)	55.3%
LSTM(200)	50.7%
MLP-RNN Vote	**91.7%**
Transformer(50)	64.3%
Transformer(10)	65.1%
Transformer-RNN	64.6%

Table 2: Summary of the notable results from various models. The numbers in parentheses are the input vector size used for that model. i.e. RNN(50) had all the GNN layers passed through a dense layer to create uniform vectors of size 50.

Model	Individual	Full Set
MLP-RNN	76.2%	57.8%
Trans-RNN	78.8%	65.5%

Table 3: Summary of the results for the component vector label tests. Individual column is the accuracy of each matching each component separately. Full Set column is matching the entire set of components for each label.

8 Conclusion

In this work, we built off the success of Graph Neural Networks at simulating real physical systems by attempting to learn from the trained network. We first attempted to simply discriminate between systems by feeding the learned parameters from the trained Graph Neural Network into an additional $ID\ Network$. We tested several model architectures including multi-layer perceptrons, convolutional networks, recurrent networks, long-short term memory, and transformers. The best performing models individually were the MLP and RNN, which both excelled with different biases. The combination of the two networks in parallel vastly out performed any other network. We then looked at the models ability to pull more granular information about the physical systems out of the learned parameters by breaking the single

class labels up into a vector of PDE components. Despite the MLP-RNN being the clear leader in the single class distinction, the Transformer-RNN model was better at identifying the individual components.

This work demonstrates that there is great potential for this method of analyzing physical systems, especially when we consider that the GNNs trained as simulators were trained for only 0.05% of the reference work and were completely useless as actual simulators. Further work looking at this technique on GNN models that are fully trained and well performing simulators would be of great interest.

References

Allen, K. R.; Guevara, T. L.; Rubanova, Y.; Stachenfeld, K.; Sanchez-Gonzalez, A.; Battaglia, P.; and Pfaff, T. 2023. Graph network simulators can learn discontinuous, rigid contact dynamics. In Liu, K.; Kulic, D.; and Ichnowski, J., eds., *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, 1157–1167. PMLR.

Battaglia, P.; Pascanu, R.; Lai, M.; Jimenez Rezende, D.; et al. 2016. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Cuomo, S.; di Cola, V. S.; Giampaolo, F.; Rozza, G.; Raissi, M.; and Piccialli, F. 2022. Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next. arXiv:2201.05624.

Karita, S.; Chen, N.; Hayashi, T.; Hori, T.; Inaguma, H.; Jiang, Z.; Someki, M.; Soplin, N. E. Y.; Yamamoto, R.; Wang, X.; Watanabe, S.; Yoshimura, T.; and Zhang, W. 2019. A Comparative Study on Transformer vs RNN in Speech Applications. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, 449–456.

Lu, L.; Pestourie, R.; Johnson, S. G.; and Romano, G. 2022. Multifidelity deep neural operators for efficient learning of partial differential equations with application to fast inverse design of nanoscale heat transport. *Phys. Rev. Res.*, 4: 023210.

Lu, L.; Pestourie, R.; Yao, W.; Wang, Z.; Verdugo, F.; and Johnson, S. G. 2021. Physics-Informed Neural Networks with Hard Constraints for Inverse Design. *SIAM Journal on Scientific Computing*, 43(6): B1105–B1132.

Sanchez-Gonzalez, A.; Godwin, J.; Pfaff, T.; Ying, R.; Leskovec, J.; and Battaglia, P. 2020. Learning to Simulate Complex Physics with Graph Networks. In III, H. D.; and Singh, A., eds., *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, 8459–8468. PMLR.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1): 61–80.

Takamoto, M.; Alesiani, F.; and Niepert, M. 2023. Learning Neural PDE Solvers with Parameter-Guided Channel Attention. arXiv:2304.14118.

Vadeboncoeur, A.; Ömer Deniz Akyildiz; Kazlauskaite, I.; Girolami, M.; and Cirak, F. 2023. Fully probabilistic deep models for forward and inverse problems in parametric PDEs. *Journal of Computational Physics*, 491: 112369.