CSCI 6962 ML AND OPTIMIZATION FINAL REPORT (G1)

Joseph Killian (killij4@rpi.edu),
Soha Yusuf (yusufs@rpi.edu),
Arthi Seetharaman (seetha2@rpi.edu) and
Dannong Wang (wangd12@rpi.edu)

ABSTRACT

Many physical systems can be represented using partial differential equations (PDEs), which are solved numerically using iterative solvers. The high computational cost of numerical simulations has motivated the recent development of machine learning techniques for approximating the solutions of PDEs. In this work, we propose a variant of the Encode-Process-Decode approach utilized by MESHGRAPHNETS (Pfaff et al. (2020)), improving the architecture of the processor with a message-passing Graph Convolutional Network and by incorporating the swish activation function (Prajit Ramachandran (2017b)). Our model is trained and tested on an aerodynamic problem, flow over a cylinder, with 600 time steps. Our results show that it can accurately predict the velocity of the fluid at each node of the mesh with higher accuracy. This model, graph-convolutional networks (GCNs), is designed to learn the dynamics of physical problems and can be further improved to accelerate the convergence of iterative solvers for large linear systems, especially those involving fine meshes, in the future.

1 INTRODUCTION

Complex physical systems, such as aerodynamic systems, deforming surfaces, acoustics, etc., can be modeled using mesh representations to solve the governing equations (PDEs). Meshes are unstructured data with a varying number of nodes in different regions of the domain. While various machine learning models, such as Convolutional Neural Networks for structured data and Recurrent Neural Networks for sequential data, have been developed, the utilization of meshes for predicting the dynamics of physical systems has not been as widespread in machine learning. Mesh-based finite element simulations are widely used to solve complex PDEs. The governing equations are numerically integrated to find the solution. The computational cost for doing this is very high, and the finer the mesh and the higher the desired accuracy, the more the cost increases when using traditional PDE solvers.

These meshes can be represented as graphs, which are an unstructured form of data (Figure 1) compared to images or sequential data. The number of nodes or edges can vary highly from graph to graph, so the models need to be robust to dealing with this type of data. This is where graph neural networks become useful, as they are able to deal with this unstructured form of data. They allow for order-invariance by design, which means the data can be of all different sizes of graphs.

MESHGRAPHNETS (Pfaff et al. (2020)) proposed an Encode-Process-Decode framework in which it encodes the mesh into a graph. It then updates the node features and edge features into a mesh space that represents the simulation mesh. Message-passing in mesh-space allows the model to capture interactions between the nodes features and the edges in the mesh. MESHGRAPHNETS framework demonstrates the simulations on four different physical systems: a flag waving in the wind, a deforming plate, flow of water around a cylinder obstacle, and the dynamics of air around the cross-section of an aircraft wing.

In this paper, we consider the flow of water around a cylinder obstacle (Figure 1) that is modeled by the 2D incompressible Navier-Stokes equations given by:

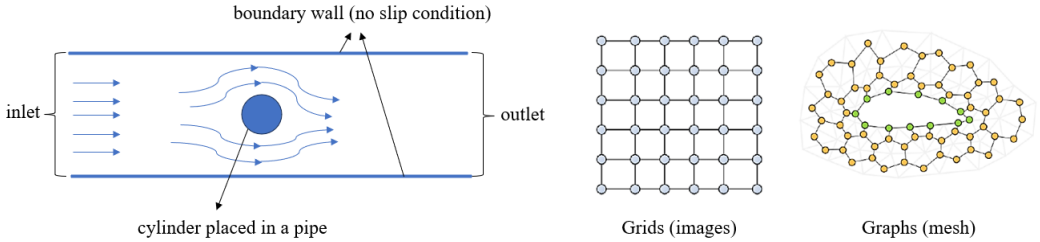

Figure 1: Right: We demonstrate our results for flow of water over a cylinder, Left: Unlike images and text, meshes are an unstructured form of data.

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\frac{1}{\rho}\nabla p + \nu \nabla^2 u, \tag{1}$$

$$\frac{\partial v}{\partial t} + (u \cdot \nabla)v = -\frac{1}{\rho}\nabla p + \nu \nabla^2 v, \tag{2}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{3}$$

In above equations, $\mathbf{u} = (u, v)$ represents the velocity vector field, p is the pressure, ρ is the fluid density, ν is the kinematic viscosity, and ∇ denotes the gradient operator. This PDE can exhibit very complex behavior as it is a stiff PDE. This means that its solution can vary very slowly, which can make it more difficult to solve.

When designing the architecture of a model, the choice of the activation function can significantly impact the performance. The MESHGRAPHNETS framework employs the rectified linear unit (ReLU) activation function in the multilayer perceptron for both the encoder and decoder. In our model, we use an activation function called swish (Prajit Ramachandran (2017a)) as a non-linear function in the multi-layer perceptron for both the encoder and decoder.

$$f(x) = x \cdot sigmoid(x) \tag{4}$$

Therefore, in this investigation, we used a new processor model (GCN) and swish activation (Prajit Ramachandran (2017a)) to enhance the accuracy of the solution (nodal velocity) for the flow of water through a cylinder at various time steps, compared to the baseline model (MESHGRAPHNETS). To improve the performance of MESHGRAPHNETS, we propose two approaches:

1. use of message-passing Graph Convolutional Network in the processor, and
2. swish activation function instead of ReLU activation in encoder and decoder.

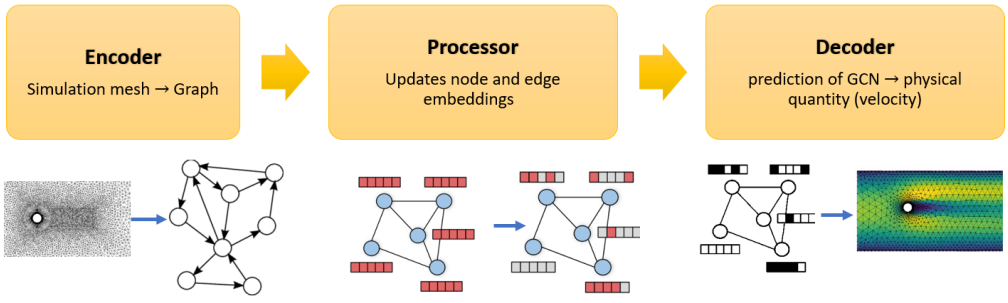

Figure 2: Our model: Variant of Encode-Process-Decode framework with GCN and swish activation function in encoder and decoder.

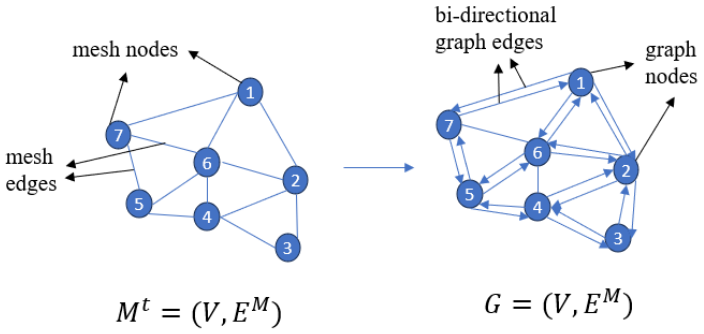

Figure 3: Mesh is converted to graph where mesh nodes become graph nodes V and mesh edges become bidirectional graph edges E^M to represent the graph $G = (V, E^M)$

2 BACKGROUND

This section will give some background on graphs and graph neural networks, but will mainly focus on explaining the methodology that was used in the MESHGRAPHNETS paper for their encode-process-decode architecture and some other details about how they trained their model.

In the MESHGRAPHNETS paper (Pfaff et al. (2020)), they studied two different types of system: Eulerian systems and Lagrangian systems. Eulerian systems refer to modeling things such as velocity over a fixed mesh, whereas Lagrangian refers to modeling systems where there can be deformations of a surface or volume. The CylinderFlow dataset that we will discuss later is a Eulerian system, as the obstacle is fixed and cannot deform. The model presented in MESHGRAPHNETS is a graph neural network with three components: Encoder, Processor, and Decoder. This is followed by an integrator to obtain the final output.

State of the system at time t is described by simulation mesh $M^t = (V, E^M)$, where V is the set of mesh nodes connected by mesh edges E^M. For the Eulerian system (flow over a cylinder), the simulation mesh M^t remains fixed at all time steps. Mesh $M^t = (V, E^M)$ is converted to a graph $G = (V, E^M)$ as shown in Figure 3, where mesh nodes become graph nodes V and mesh edges become bidirectional graph edges E^M to represent the graph $G = (V, E^M)$. The overall goal here is to take the features of the mesh at time t, and use that to predict the features of the mesh at time t+1.

2.1 ENCODER

Encoder encodes node and edge features to 128 dimensional latent vector for each node and each edge by using edge encoder ϵ^M and node encoder ϵ^V. Both edge encoder ϵ^M and node encoder ϵ^V are MLPs that encode current mesh M^t to graph G. Encoder takes in node features $v_i \in V$ and edge features $e_{ij} \in E^M$ as input, where node features for node i are velocity at node i and node-type (which indicates boundary nodes) and edge features for edge e_{ij} connecting node i to node j are relative displacement vector $u_{ij} = u_i - u_j$ and norm $|u_{ij}|$ of displacement vector.

$$M^t = (V, E^M) \rightarrow G = (V, E^M) \tag{5}$$

where, M^t is the simulation mesh at time step t with V set of nodes connected by E^M edges and G is the encoded graph with bidirectional edges with 128 dimensional node features and 128 dimensional edge features.

2.2 PROCESSOR

Processor applies message-passing GraphNetBlocks Sanchez-Gonzalez et al. (2018) to update edge embeddings e_{ij}^M, which are the simulation mesh edges from node i to node j and node embeddings

v_i. The updated edge embeddings ${e'}_{ij}^{M}$ and node embeddings v_i' are then used to construct the new updated graph.

Processor contains L identical message-passing blocks, each block is applied in sequence to the output of previous block. First, edge embeddings are updated using MLP f^M:

$$ {e'}_{ij}^{M} \leftarrow f^M(e_{ij}^M, v_i, v_j) \tag{6} $$

Then, node embeddings are updated using MLP f^V:

$$ {v'}_i \leftarrow f^V(v_i, \sum_j {e'}_{ij}^{M}) \tag{7} $$

Finally, all updated edge embeddings ${e'}_{ij}^{M}$ and node embeddings v_i' are aggregated for the whole graph and process is repeated for L message-passing steps.

2.3 DECODER

Decoder predicts the state of the mesh at time $t+1$ from the input mesh state at time t. Decoder uses MLP δ^V that takes in latent node features v_i (output of processor) and computes output features p_i. Desired physical quantity q_i (velocity) is determined from p_i by integrating p_i once (for first-order systems). At last, q_i^{t+1} is used to update all mesh nodes V to produce mesh M^{t+1}.

$$ q_i^{t+1} = p_i + q_i^t \tag{8} $$

where, q_i^{t+1} is desired physical quantity (velocity) at time step $t+1$ and p_i is the prediction from the network.

Algorithm 1 Original MESHGRAPHNETS Algorithm

Encode: Mesh $\rightarrow$ Graph
$M^t = (V, E^M) \rightarrow G = (V, E^M)$
Encode relative displacement: $\mathbf{u}_{ij} = \mathbf{u}_i - \mathbf{u}_j$ and norm $|\mathbf{u_{ij}}|$ into the mesh edges $\mathbf{e}_{ij}^M$
Encode dynamical features, and one hot vector for node type into node features: $\mathbf{q}_i \rightarrow \mathbf{v}_i$
Encode the features into a vector at each node and each edge using encoder MLPs with ReLU activation ϵ^M, ϵ^V for mesh edges $\mathbf{e}_{ij}^M$ and $\mathbf{v}_i$ respectively
Processor:
for l in L **do**
  for each edges e_{ij}^M **do**
    Get sender and receiver node v_i, v_j
    Compute output edges ${e'}_{ij}^{M} = f^M(e_{ij}^M, v_i, v_j)$ (MLP with residual connections)
  end for
  for each node v_i **do**
    Aggregate all edges for current node $\hat{e}_i = \sum_j {e'}_{ij}^{M}$
    Compute new node ${v'}_i = f^V(v_i, \hat{e}_i)$ (MLP with residual connections)
  end for
end for
Aggregate all nodes and edges $\hat{e} = \sum e'_{ij}, \hat{v} = \sum v'_i$
Return the new graph $g' = (\hat{e}, \hat{v})$
Decode: Use MLP with ReLU to get output features $\delta^V(v_i) \rightarrow p_i$
Integrate: $p_i \rightarrow q_i^{t+1}$

2.4 OTHER DETAILS

In our case, we are only looking at the velocities, so the integration step in the decoder can be used when computing higher order derivatives of velocity like acceleration. For training of the model, it

is only supervised in the next step in the sequence. To make the model more robust, they also add noise to the training data (Pfaff et al. (2020)). Their model was trained by using a L2 loss between the per-node output features $\mathbf{p}_i$ and the ground truth values for that step $\hat{\mathbf{p}}_i$. The training loss looks at all of the output features predicted. When testing the rollouts later, the error is calculated using Root Mean Squared Error(RMSE) on only the quantities that are actually being modeled. More details about training and testing can be found in Section 5 where we discuss experimental results. The next section will discuss some other related Graph Neural Network architectures.

3 RELATED WORK

In section 2, we discussed Encode-Process-Decode framework proposed in Pfaff et al. (2020). Since there has been recent development of machine learning models for predicting physics of dynamical systems, we will discuss a few popular approaches. The MultiScale MeshGraphNets (Peter Wirnsberger & Battaglia (2022)) approach utilizes high-resolution and low-resolution meshes. It applies the Encode-Process-Decode GNN framework. In this framework, the encoder encodes both the fine and coarse input graphs and generates upsampled and downsampled graphs. The processor updates the four graphs, and the decoder produces the prediction. This approach enhances accuracy and reduces computational cost compared to the MeshGraphNets baseline. de Avila Belbute-Peres et al. (2020) introduced a solution that combines a graph convolutional network and CFD solvers. The network takes the fine mesh as input for the GCN, while the coarse mesh is used for the CFD solvers. It combines the solver output with an intermediate GCN output and applies GCN again. This approach enhances accuracy and speed compared to a solely learned model. Our approach is similar as we also represent meshes using graphs and utilize GCN in the processor.

Convolutional neural networks (CNNs) are an alternative approach to predict physical systems. Afshar et al. (2019) introduces an encoder-decoder CNN architecture. This architecture utilizes a stack of convolution layers to encode the input into a lower-dimensional space. The encoded information is then passed through fully connected layers before being decoded using a deconvolution layer to obtain the prediction. We chose to use message passing graph convolutional networks instead of CNNs because many physical systems are unstructured and easier to represent as mesh. Graph convolutional networks are well-suited for handling graph inputs which can be encoded from mesh. Other papers, such as Battaglia et al. (2018), Lee & You (2019) or Zhang et al. (2018), also uses CNNs to predict flow over a circular cylinder or airfoil simulations.

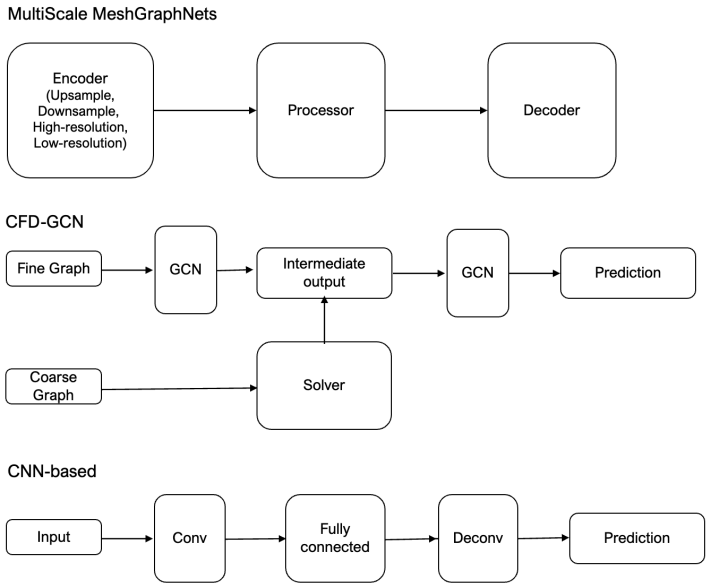

Figure 4: Basic architecture of the three mentioned related works

4 METHODOLOGY

Our idea is to implement Graph Convolutional Network (GCN) along with the swish activation function (Algorithm 2) to improve the performance of MESHGRAPHNETS framework for predicting the dynamics of physical systems. Graph Convolutional Network (GCN) model incorporates symmetric-normalized aggregation and the self-loop update method (Kipf & Welling (2017)). Graph convolutional network updates hidden embeddings $H^{(l+1)}$ for the whole graph at layer $l+1$ as described below:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{9}$$

where, $H^{l+1} \in \mathbb{R}^{N\times D}$ is the updated hidden embedding matrix (each row represents a node embedding) at layer $l+1$, $\tilde{A} \in \mathbb{R}^{N\times N} = A + I_N$ is the adjacency matrix with self loops, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ is the diagonal value for the degree matrix $\tilde{D} \in \mathbb{R}^{N\times N}$ with self loops, $H^{(l)}$ is hidden embedding matrix at layer l, $W^{(l)} \in \mathbb{R}^{D\times D}$ is the trainable weight matrix, and N is the number of nodes and D is the number of node features.

Algorithm 2 Our GCN-Variant of MESHGRAPHNETS (Our modifications in Red)

Encode: Mesh $\rightarrow$ Graph
$M^t = (V, E^M) \rightarrow G = (V, E^M)$
Encode relative displacement: $\mathbf{u}_{ij} = \mathbf{u}_i - \mathbf{u}_j$ and norm $|\mathbf{u_{ij}}|$ into the mesh edges $\mathbf{e}_{ij}^M$
Encode dynamical features, and one hot vector for node type into node features: $\mathbf{q}_i \rightarrow \mathbf{v}_i$
Encode the features into a vector at each node and each edge using encoder MLPs with Swish activation ϵ^M, ϵ^V for mesh edges $\mathbf{e}_{ij}^M$ and $\mathbf{v}_i$ respectively
Processor:
for l in L **do**
  for each edges e_{ij}^M **do**
    Get sender and receiver node v_i, v_j
    Compute output edges ${e'_{ij}}^M = f^M(e_{ij}^M, v_i, v_j)$ (Using GCN)
  end for
  for each node v_i **do**
    Aggregate all edges for current node $\hat{e}_i = \sum_j e_{ij}^{\prime M}$
    Compute new node ${v'}_i = f^V(v_i, \hat{e}_i)$ ((Using GCN)
  end for
end for
Aggregate all nodes and edges $\hat{e} = \sum e'_{ij}, \hat{v} = \sum v'_i$
Return the new graph $g' = (\hat{e}, \hat{v})$
Decode: Use MLP with Swish activation to get output features $\delta^V(v_i) \rightarrow p_i$
Integrate: $p_i \rightarrow q_i^{t+1}$

Proper normalization is essential for achieving stable and strong performance when using a GNN (Sanchez-Gonzalez et al. (2020)). For predicting the dynamics of physical system, node feature information is more useful than structural information of the graph. Therefore, normalization will help in learning the node embeddings more accurately. Here are the following reasons why Graph Convolutional Networks perform better than traditional message-passing Graph Neural Networks:

- Parameter Sharing and Local Information Aggregation: GCNs leverage the concept of convolutional operations on graphs, similar to how convolutional neural networks (CNNs) operate on regular grids (e.g., images). This allows them to share parameters across different regions of the graph, capturing local patterns effectively.
- Ability to Capture Node Features: GCNs can effectively capture and utilize features associated with nodes in a graph. Each node's representation is updated by aggregating information from its neighbors, enabling the model to consider both the node's own features and the features of its neighboring nodes.

- Representation Learning: GCNs are capable of learning useful node representations that capture complex relationships in the graph. These learned representations can be valuable for downstream tasks and can help uncover hidden patterns in the data.

Choice of activation function also greatly affects the performance of the machine learning model.We implement swish activation function (Prajit Ramachandran (2017a)) instead of ReLU in the MLP for encoder ϵ^M and ϵ^V and decoder δ^V. Swish activation function performs better than ReLU for deeper networks despite squishing gradients.

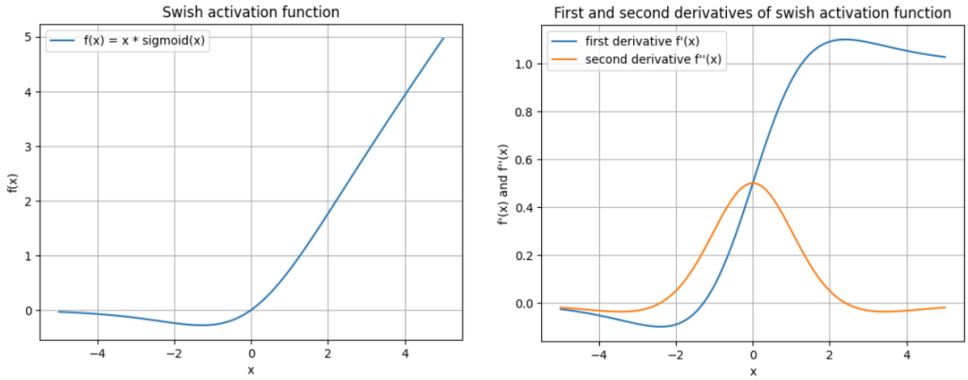

Figure 5: Left: Swish activation function $f(x) = x \cdot \sigma(x)$, Right: First derivative $f'(x) = f(x) + \sigma(x)(1-f(x))$ and second derivative of swish activation function. Where, $\sigma(x) = (1+exp(-x))^{-1}$.

For implementing the GCN in the processor as opposed to the MLP used in Algorithm 1, we used the GCNConv layer that is past of the torch-geometric library. This layer implements the graph convolution operator as described in (Kipf & Welling (2017)). We also follow this layer with a linear output layer to get the output of the GCN. So each GCN discussed in Algorithm 2 consists of one layer of GCNConv and one linear output layer. The next sections goes into our results and how they compare with the baseline model of MESHGRAPHNETS.

5 EXPERIMENTAL RESULTS

In this section, we compare the results obtained by employing a graph convolutional network alongside the base methodology of MESHGRAPHNETS with the base output of MESHGRAPHNETS. First, we present the replicated results on the CylinderFlow dataset, followed by the presentation of our generated results for comparative analysis. The code used to generate these results was implemented in TensorFlow 1 with Python version 3.7 and the model was trained on NVIDIA GeForce GTX 1060 for 6 hours.

5.1 BASELINE RESULTS FROM MESHGRAPHNETS - CFD MODEL CYLINDERFLOW

The dataset chosen for this section is the CylinderFlow dataset, designed to represent fluid flow around a cylinder-shaped obstacle. Due to resource constraints, we opted for a shorter training duration (100K steps vs 10M steps). Consequently, the results presented here may not exactly match those reported in MESHGRAPHNETS, but the outputs remain plausible and similar to their results. The process of generating the following plots and images involves three steps: first, training the models with the hyperparameters in the table below; second, generating the rollout trajectories; and finally, plotting these trajectories to produce the images and videos of the output. The table below displays the hyperparameters used during the training process, all of which are identical to those in MESHGRAPHNETS except for the number of training steps. In the original model they trained for 10M steps and used a learning rate scheduler that began after 5M steps. Since we only trained for 100K steps, this learning rate decay did not factor in, so it was not listed here.

Hyperparameter	Description	Value
Optimizer	Provides efficient and adaptive learning rates	Adam
Training loss	Squared error loss function	L2
Learning rate	Step size for updating weights	10^{-4}
Batch size	Batch size for Training	2
Number of rollouts	No. of rollout trajectories	10
Training steps	No. of examples the model has trained on from training set	10^5
Number of epochs	No. of times we go through training set	-

Table 1: Parameters for the baseline model, number of rollouts refers to the number of rollout trajectories used in testing, and number of training steps is the number of examples that the model saw during training (during one epoch)

Alongside the provided code, we incorporated functionality to monitor and visualize both training and validation losses throughout the training process. The training and validation losses were saved every 100 training steps, and Figure 4 illustrates the average loss over every 1,000 training steps. The mentioned loss refers to the L2 loss between the per-node output features p_i and the corresponding ground truth values $\hat{p_i}$. These features p at each node refer to all the features that are being modeled, not just velocity that we simulate when it comes to testing.

$$L2 = \frac{1}{N}\sum_{i=1}^{N}||p_i - \hat{p}_i||_2^2 \tag{10}$$

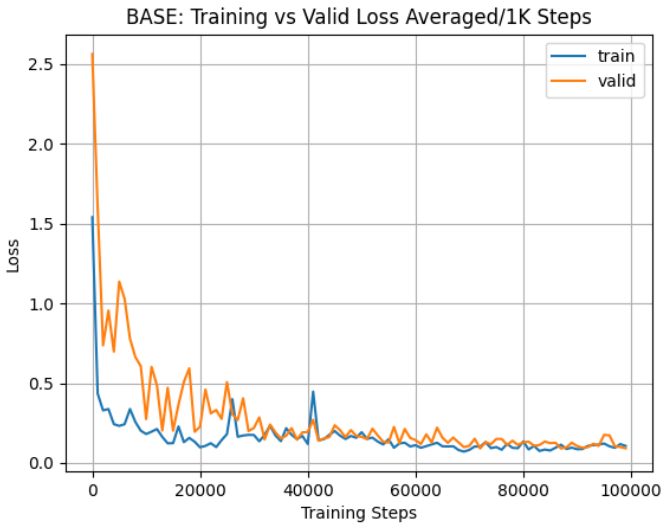

Figure 6: Baseline model: Plot of the average training and validation loss over every 1,000 training steps for the base CylinderFlow dataset. Model trained on training set, and then evaluated periodically on validation set during training. Train/valid/test split was (84/8/8). Model geometries are similar in each set.

After completing the training, the subsequent step involved generating the rollout trajectories. These trajectories represent the velocities at each time step for the given system. In our case, we generated ten rollout trajectories corresponding to ten distinct scenarios, wherein the cylinder served as an obstacle in various positions and sizes.

These rollout trajectories serve as the input for the model to generate the final output. Additionally, during this step, we computed the rollout Root Mean Squared Error (RMSE) for all calculated

rollouts. The rollout RMSE measures the root mean squared error of the velocity for our use case. It is calculated by taking the mean across all spatial coordinates, mesh nodes, and steps in each trajectory within the test dataset. More simply, we are taking the difference between the velocity in the ground truth and the predicted velocity, and finding this for different steps.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N} ||v_i - \hat{v}_i||^2} \tag{11}$$

The RMSE values are computed at steps 1, 10, 20, 50, 100, and 200. The error tends to increase as the number of steps increases because the error accumulates with each subsequent step. The table presents the results of the rollout RMSE. We were limited in the number of timesteps that we could simulate up to by the dataset, but as shown in table two, towards thee 600th timestep the error is beginning to accumulate and spiral out of control.

The final step in this process is to visualize these rollout trajectories. This step is straightforward and involves executing the model on the rollout files, allowing us to observe the generated output. The images below depict two of these trajectories at different points, providing a visual representation of the system's progression from earlier timesteps to later ones.

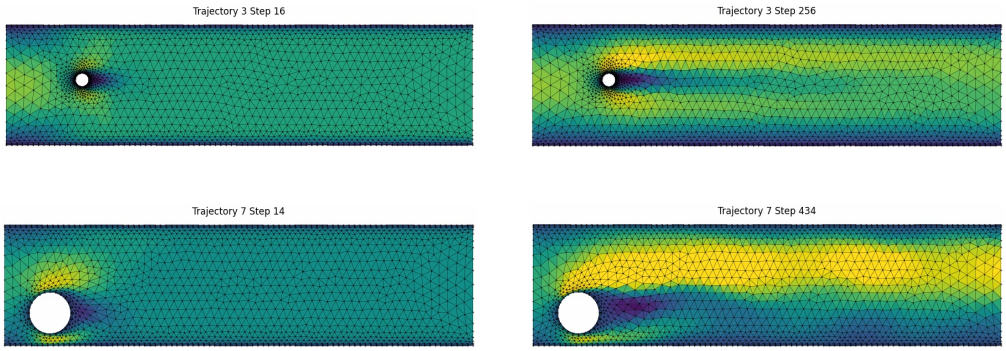

Figure 7: Baseline results for the CylinderFlow dataset for two different cross-sectional areas for cylinder. The colors in the figure show nodal velocities - minimum (blue) to maximum (yellow).

5.2 OUR METHOD USING GRAPH CONVOLUTIONAL NETWORK ON CYLINDERFLOW

In this section we will present the results we obtained with out model that replaces the MLPs with a GCN in the processor of the GNN. The next section will give comparisons between this model and the baseline.

For our method we used the same base structure as in the original model, but for ease of use, we decided to use a version written in PyTorch `https://github.com/echowve/meshGraphNets_pytorch`. This required a few extra steps and processing, and all of those details can be found at our repository `https://github.com/SohaYusuf/CSCI6962-Final-Project`. Many of the hyperparameters were identical for the run of our model as for the base model. We are still using Adam optimizer and L2 loss function, and the same number of rollouts (10) as well. The only difference is the number of training steps. The PyTorch version of this data uses the more traditional notation of epochs, so we trained this model for the same time period as the baseline model (around 6 hours). This ended up being 3 epochs. We also utilized a different batch size than the original paper of 20 vs 2. The table below summarizes all the hyperparameters we used for out model.

As mentioned, the training loss and test error metrics are identical as to what was used for the baseline model in order to give an fair comparison between the two methods. We generated the following graphs for the training and validation loss over the course of training for 3 epochs, the test

Hyperparameter	Description	Value
Optimizer	Provides efficient and adaptive learning rates	Adam
Training loss	Squared error loss function	L2
Learning rate	Step size for updating weights	10^{-4}
Batch size	Batch size for Training	20
Number of rollouts	No. of rollout trajectories	10
Training steps	No. of examples the model has trained on from training set	-
Number of epochs	No. of times we go through training set	3

Table 2: Parameters for the our GCN model, the epochs used here are similar to what we trained the original model for in regards to training time.

error for the rollouts, and generated images/videos. The way in which the images presented here were rendered is different that for the baseline, so the color scheme is slightly different, but they both model the same physical system.

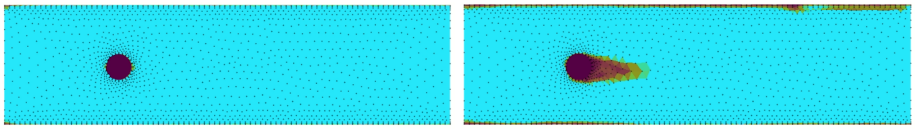

Figure 8: Our results for the CylinderFlow dataset for a cross-sectional area of the cylinder. The colors in the figure show nodal velocities - minimum (red) to maximum (light-blue).

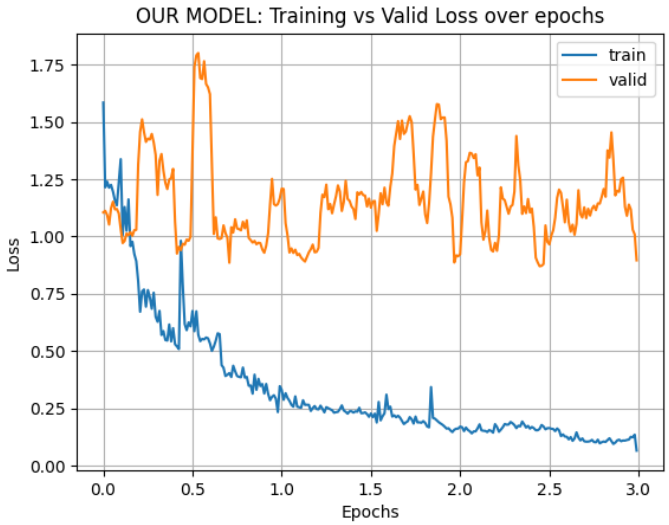

Figure 9: Our loss curves for the CylinderFlow dataset over training for three epochs. Losses were averaged to smooth out the curves. Our models shows overfitting because training loss continues to decrease while validation loss does not converge.

5.3 COMPARISON BETWEEN OUR MODEL AND THE BASELINE MODEL

Overall the results that we obtained with our model, were not as promising as those obtained in the baseline model. The training loss we obtained is similar to that of the base model, but the validation loss has a high level of variance. The model was only trained for a relatively short period of time, but the loss curves suggest that out model is over fitting to the data. This can be seen in the test error

as well, as after just 150 training steps we reach the same error as the baseline model does after the full 600 steps. The test error metric we used is root mean squared error (RMSE) and is calculated for all rollouts produced. RMSE increases as the timesteps increase.

Time steps (1 = 0.01s)	RMSE (Test Error) Our Model	RMSE (Test Error) Baseline
1	2.19e-2	5.23e-5
50	1.15e-1	1.08e-3
100	1.52e-1	2.71e-3
150	2.17e-1	5.01e-3
200	2.70e-1	7.73e-3
250	3.13e-1	1.04e-2
300	3.50e-1	1.26e-2
350	3.87e-1	1.43e-2
400	4.28e-1	1.66e-2
450	4.72e-1	1.97e-2
500	5.14e-1	2.23e-2
550	5.51e-1	2.45e-2

Table 3: Comparison of RMSE for both models. Norm of target velocity varies between 1400 to 1600.

The test error for our model is overall higher than the baseline model due to the lack of relative encoding and message computing. Since GCN does not compute messages on edges, it hinders message propagation across edges. Furthermore, depsite the high capacity of our model, it does not capture spatial physical information across the mesh.

Visually the model is also very different compared to the baseline model. Their are points where the result looks plausible, but overall the rendering of the simulation shows that our model has a lot more error than the baseline model. The full video can be seen attached to the code for this paper `https://github.com/SohaYusuf/CSCI6962-Final-Project`.

6 CONCLUSION AND FUTURE WORK

We propose a variant of the Encode-Process-Decode approach. This variant involves modifying the processor's architecture to a Graph Convolutional Network and replacing the ReLU activation function with the swish activation function in both the Encoder and Decoder components.

Our model aims to predict the dynamics of water flow over a cylinder by minimizing the L2 loss between the target and predicted velocities at mesh nodes. The improvements of our model are demonstrated on a static mesh (CylinderFlow mesh). The nodal velocities are evaluated for 600 time steps during inference. This paper presents higher accuracy of the solution for learning simulations and proposes the use of machine learning models to accelerate numerical simulations in the future.

Our model overfits compared to the baseline model due to the lack of message propagation across the edges. The performance could be improved with neighborhood connectivity for each node and message-passing across edges. In the future, we believe this model could still be useful as a preconditioner to accelerate the convergence of iterative solvers e.g. GMRES. The prediction of the model can be used as an initial guess to reduce the number of iterations for convergence. There are also other directions and architectures that we could implement such as a graph transformers that may be able to achieve a better performance than had been achieved with the baseline model and can be used to learn the dynamics of varying physical systems. Lastly, the current model itself could be upgraded to make predictions on larger meshes by showing good generalization ability.

REFERENCES

Yaser Afshar, Saakaar Bhatnagar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. Prediction of aerodynamic flow fields using convolutional neural networks. 2019. doi: 10.1007/s00466-019-01740-0.

Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

Filipe de Avila Belbute-Peres, Thomas D. Economon, and J. Zico Kolter. Combining differentiable pde solvers and graph neural networks for fluid flow prediction, 2020.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

Sangseung Lee and Donghyun You. Data-driven prediction of unsteady flow over a circular cylinder using deep learning. *Journal of Fluid Mechanics*, 879:217–254, September 2019. ISSN 1469-7645. doi: 10.1017/jfm.2019.700. URL http://dx.doi.org/10.1017/jfm.2019.700.

Paolo Galeone Luca Grementieri. Towards neural sparse linear solvers. 2022.

Meire Fortunato Tobias Pfaff Peter Wirnsberger, Alexander Pritzel and Peter Battaglia. Multiscale meshgraphnet. 2022.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. 2020.

Quoc V. Le Prajit Ramachandran, Barret Zoph. Swish: A self-gated activation function, 2017a.

Quoc V. Le Prajit Ramachandran, Barret Zoph. Swish: A self-gated activation function. 2017b.

Matthew Willson Peter Wirnsberger Meire Fortunato Ferran Alet Suman Ravuri Timo Ewalds Zach Eaton-Rosen Weihua Hu Alexander Merose Stephan Hoyer George Holland Oriol Vinyals Jacklynn Stott Alexander Pritzel Shakir Mohamed Peter Battaglia Remi Lam, Alvaro Sanchez-Gonzalez. Graphcast: Learning skillful medium-range global weather forecasting. 2022.

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, 2018.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to simulate complex physics with graph networks, 2020.

Vissarion Papadopoulos George Stavroulakis Stefanos Nikolopoulos, Ioannis Kalogeris. Ai-enhanced iterative solvers for accelerating the solution of large-scale parameterized systems. 2022.

Yao Zhang, Woong-Je Sung, and Dimitri Mavris. Application of convolutional neural network to predict airfoil lift coefficient, 2018.