Simulating fluid flow with Graph Convolutional Networks

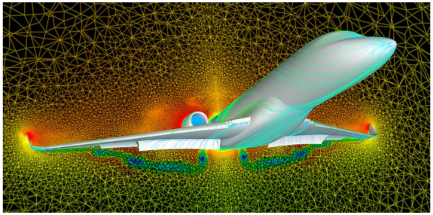
Soha Yusuf, Joseph Killian Jr., Dannong Wang, Arthi Seetharaman

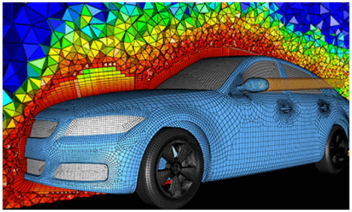
Rensselaer Polytechnic Institute

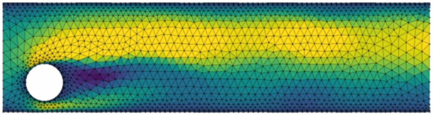
December, 2023

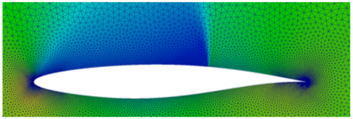
Table of Contents

Motivation: There is a need to accelerate the solution of large-scale flow simulations.







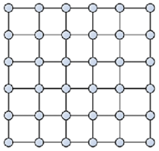


Motivation: Complex Physical Systems can be modeled using mesh representations to solve PDEs.

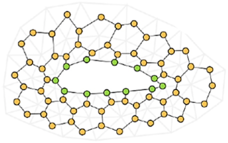
- Many flow simulations are governed by complex PDEs like Navier-Stokes equations
  - They can give rise to very complex behavior e.g "Stiff PDEs $\rightarrow$ solution varies slowly"
- Mesh-based finite element simulations are widely used to solve complex PDEs
  - Governing equations are integrated numerically to find the solution $\rightarrow$ high computational cost
  - Simulating fine meshes for higher accuracy further increases computational cost when using traditional PDE solvers

Motivation: Graph Neural Networks allow order invariance by design and can handle unstructured data.

- Meshes can be represented as graphs, which are an unstructured form of data, unlike images or sequential data.


Grids (images)

Sequential (text)


Graphs (mesh)

- Graph Neural Networks (GNN) can handle unstructured data like graphs
  - GNNs allow order invariance by design → training data can have variable size of graphs (meshes).

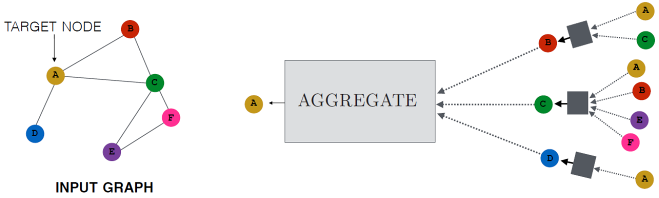
Problem: PDE solvers can solve complex, non-linear PDEs but they are computationally expensive.

- The paper MESHGRAPHNETS (Pfaff et al., 2020) proposed an Encode-Process-Decode architecture with MLP components to solve this problem.
- Their model was applied to a variety of static and dynamic datasets, and was able to make accurate physical predictions
    - How can we improve upon the performance from this paper?
- **Our Goal:** Modify the architecture of MESHGRAPHNETS using convolutional layers (Graph Convolutional Network) and swish activation function to improve performance

Introduction: A graph neural network uses message passing to learn "long-distance" relationships.

$$h_u^{(k)} = \sigma(W_{self}^{(k)} h_u^{(k-1)} + W_{neigh}^{(k)} \sum\nolimits_{v \in N(u)} h_v^{(k-1)} + b^{(k)}) \quad (1)$$

where, σ (sigmoid) is the 'update' function and $\sum$ is the 'aggregation' function.



Introduction: We use 2D incompressible Navier-Stokes equations to simulate flow of water around a cylinder.

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

$$\frac{\partial v}{\partial t} + (u \cdot \nabla) v = -\frac{1}{\rho} \nabla p + \nu \nabla^2 v, \tag{3}$$

$$\nabla \cdot \mathrm{u} = 0. \tag{4}$$

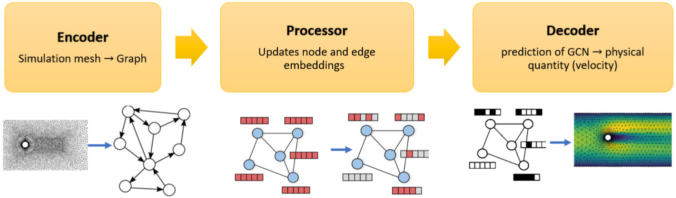
where, $\mathrm{u} = (u, v)$: velocity vector field, p : pressure, ρ : fluid density, ν : kinematic viscosity, and ∇ : gradient operator.

- Dataset contains 2D nodal velocity for a cylinder (with variable locations and varying diameter) in a "pipe" (see next slide).
- Mesh is static (fixed for all time steps)

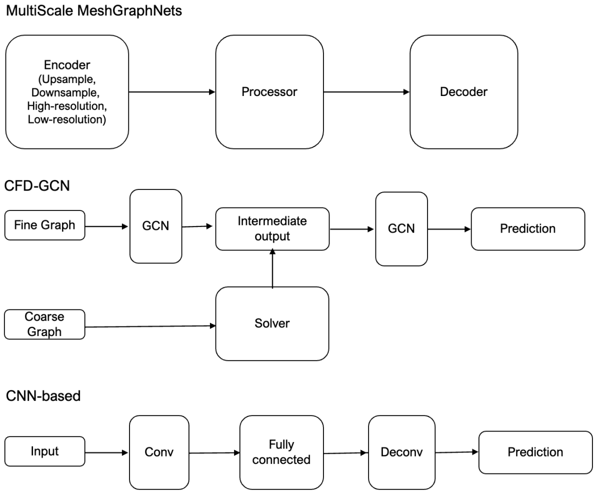
Introduction: We propose a variant of Encode-Process-Decode using GCN and swish activation.

Proposed Modifications to MESHGRAPHNETS:

- Implement Graph Convolutional Network instead of standard Graph Neural Network in processor
- Replace the use of the ReLU activation function with the swish activation function


Encoder
Simulation mesh → Graph
Processor
Updates node and edge embeddings
Decoder
prediction of GCN → physical quantity (velocity)

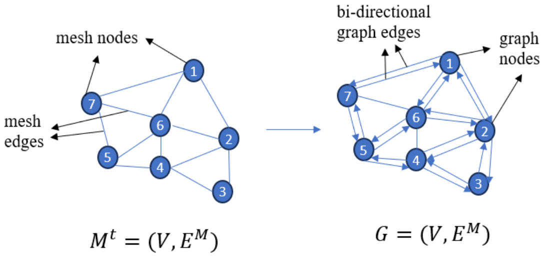
Related Works: In recent years, there has been multiple approaches to learn the dynamics of physical systems.

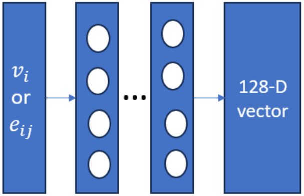


Background: State of the system at time t is described by simulation mesh $M^t = (V, E^M)$

- $M^t = (V, E^M)$ is the simulation mesh (fixed), where V is the set of mesh nodes connected by mesh edges E^M
- Flow over a cylinder is analyzed as *Eulerian* system, where we model evolution of 2D velocity field over a fixed mesh
- Mesh nodes become graph nodes V and mesh edges become bidirectional graph edges E^M to represent the graph $G = (V, E^M)$ (see figure on next slide)

Background: Edge encoder ϵ^M and node encoder ϵ^V are MLPs that encode current mesh M^t to graph G





Background: Node and edge features are encoded to 128D latent vector for each node and each edge

- Encoder input: node features $v_i \in V$ and edge features $e_{ij} \in E^M$
- Node features for node i include:
  - velocity at node i
  - node-type (indicates boundary nodes)
- Edge features for edge e_{ij} connecting node i to node j include:
  - relative displacement vector $u_{ij} = u_i - u_j$
  - norm $|u_{ij}|$ of displacement vector

Background: Processor applies message-passing GNN to update edge embeddings e_{ij}^{M} and node embeddings v_i

- Processor contains L identical message-passing blocks, each block is applied in sequence to output of previous block (in the following order):
  1. Edge embeddings are updated using MLP f^M
  $$e'^{M}_{ij} \leftarrow f^M(e_{ij}^{M}, v_i, v_j) \tag{5}$$
  2. Node embeddings are updated using MLP f^V
  $$v'_i \leftarrow f^V(v_i, \sum_j e'^{M}_{ij}) \tag{6}$$
  3. The process is repeated for L blocks

Background: Decoder predicts (p_i) 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)

$$q_i^{t+1} = p_i + q_i^t \tag{7}$$

- At last, q_i^{t+1} is used to update all mesh nodes V to produce mesh M^{t+1}

Background: Algorithm

Algorithm 1 Original MESHGRAPHNETS Algorithm

```
Encode: Mesh → Graph
$M^t = (V, E^M) \rightarrow G = (V, E^M)$
Encode relative displacement: $\mathrm{u}_{i,j} = \mathrm{u}_i - \mathrm{u}_j$ and norm $|\mathrm{u_{i,j}}|$ into the mesh edges $\mathrm{e}^M_{i,j}$
Encode dynamical features, and one hot vector for node type into node features: $\mathrm{q}_i \rightarrow \mathrm{v}_i$
Encode the features into a vector at each node and each edge using encoder MLPs with ReLU activation $\epsilon^M, \epsilon^V$ for mesh edges $\mathrm{e}^M_{i,j}$ and $\mathrm{v}_i$ respectively.
Processor:

for each edge $e^M_{i,j}$ do
    Get sender and receiver node $v_i$, $v_j$
    Compute output edges $e'^M_{i,j} = f^M(\mathrm{e}^M_{i,j}, v_i, v_j)$ (MLP with residual connections)

    for each node $v_i$ do
        Aggregate all edges for current node $\hat{e}_i = \sum_j e'^M_{i,j}$
        Compute new node $v'_i = f^V(v_i, \hat{e}_i)$ (MLP with residual connections)
        Aggregate all nodes and edges $\hat{e} = \sum e'_{i,j}, \hat{v} = \sum v'_i$
        Return the new graph $g' = (\hat{e}, \hat{v})$
Decode: Use MLP with ReLU activation to get output features $\delta^V(v_i) \rightarrow p_i$
Integrate: $p_i \rightarrow q_i^{t+1}$
```

Methodology: Algorithm

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

```
Encode: Mesh → Graph
$M^t = (V, E^M) \rightarrow G = (V, E^M)$
Encode relative displacement: $u_{i,j} = u_i - u_j$ and norm $|u_{i,j}|$ into the mesh edges $e^M_{i,j}$
Encode dynamical features, and one hot vector for node type into node features: $q_i \rightarrow v_i$
Encode the features into a vector at each node and each edge using encoder MLPs with Swish
activation $\epsilon^M, \epsilon^V$ for mesh edges $e^M_{i,j}$ and $v_i$ respectively.
Processor:

for each edges $e^M_{i,j}$ do
    Get sender and receiver node $v_i$, $v_j$
    Compute output edges ${e'}_{i,j}^{M} = f^M(e^M_{i,j}, v_i, v_j)$ (Using GCN)

    for each node $v_i$ do
        Aggregate all edges for current node $\hat{e}_i = \sum_j {e'}^M_{i,j}$
        Compute new node ${v'}_i = f^V(v_i, \hat{e}_i)$ ((Using GCN)
        Aggregate all nodes and edges $\hat{e} = \sum e'_{i,j}, \hat{v} = \sum v'_i$
        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: $p_i \rightarrow q_i^{t+1}$
```

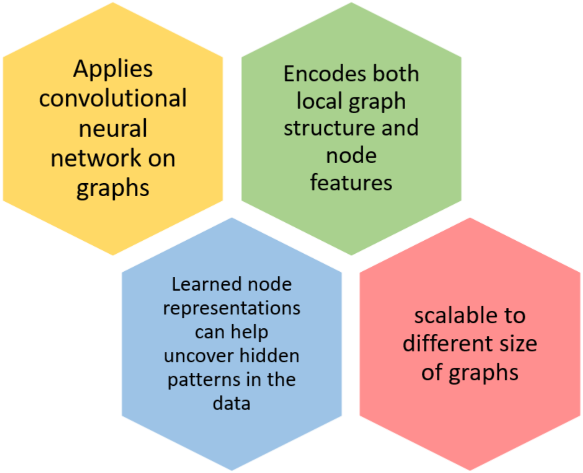
Methodology: Graph Convolutional Network employs normalized aggregation with self loops to learn features.

- Graph convolutional network updates hidden embeddings $H^{(l+1)}$ for the whole graph at layer $l+1$ as described below:

$$H^{(l+1)} = (\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{8}$$

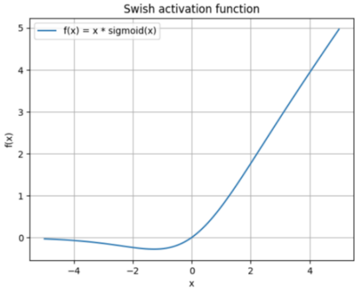
  - $H^{l+1} \in \mathbb{R}^{N\times D}$ is the updated hidden embedding matrix (each row represents a node embedding) at layer $l+1$,
  - $\tilde{A} \in \mathbb{R}^{N\times N} = A + I_N$ is the adjacency matrix with self loops,
  - $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ is the diagonal value for the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

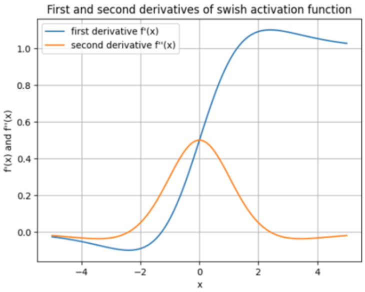
Methodology: GCNs can learn useful node representations that capture complex relationships in the graph.


Applies convolutional neural network on graphs
Encodes both local graph structure and node features
Learned node representations can help uncover hidden patterns in the data
scalable to different size of graphs

Methodology: Swish activation function performs better than ReLU for deeper networks.

- We implement swish activation function (Prajit Ramachandran (2017)) instead of ReLU in the MLP for encoder ϵ^M and ϵ^V and decoder δ^V.





Results: Hyperparameters for baseline vs. our model

- Reproduced results from original paper in TensorFlow1 [1] and then used updated PyTorch version [2] for our model
- Baseline model used training steps (number of examples the model has trained on from training set) and our model uses epochs (number of times we go through training set)

Hyperparameter	**Baseline**	**Our model**
Optimizer	Adam	Adam
Training loss	L2	L2
Learning rate	10^{-4}	10^{-4}
Batch size	2	20
Number of rollouts	10	10
Training steps	10^5	-
Number of epochs	-	3

Results: Loss function and Test Error Function

- Implementation: Train → Generate Rollout Trajectories → Plot those trajectories
- Training loss is the L2 loss between the per-node outputs $\hat{p}_i$ (velocity at $t+1$) and ground truth values p_i (velocity at $t+1$) and N is the number of nodes

$$L2 = \frac{1}{N} \sum_{i=1}^{N} ||p_i - \hat{p}_i||^2 \tag{9}$$

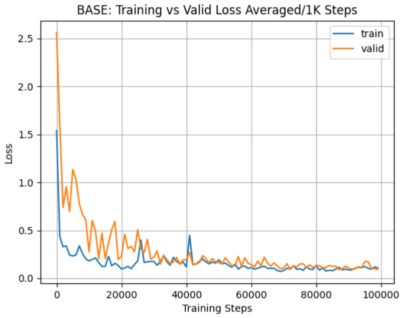
$$\hat{p}_i = \text{Encode-Process-Decode}(M^t = (V, E^M)) \tag{10}$$

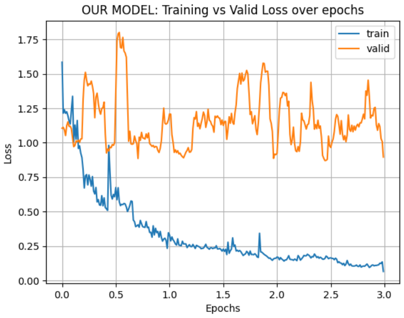
- Test error is the root mean squared error (RMSE), calculated for all rollouts produced → RMSE increases with timesteps

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

Results: Training/Validation Loss Comparison

- Comparison of the training and validation losses during training
- Each model was trained for $\sim$ 6 hours (100,000 training steps for BASE, 3 epochs for our model)
- Our models shows overfitting because training loss continues to decrease while validation loss does not converge.





Results: Test Error for our model is higher than baseline due to lack of relative encoding and message computing.

- GCN does not compute messages on edges and this hinders message propagation across edges.
- Despite high capacity of our model, it does not capture spatial physical information across the mesh.

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

Results: Rendering Comparison

- Comparison between the rollout (testing) trajectories at different stages of the simulation, darker areas represent low velocities, and brighter areas represent high velocity
- Average number of nodes in test set is 1885
- Each scenario was simulated for 6 seconds (results start to diverge the longer the simulation runs)

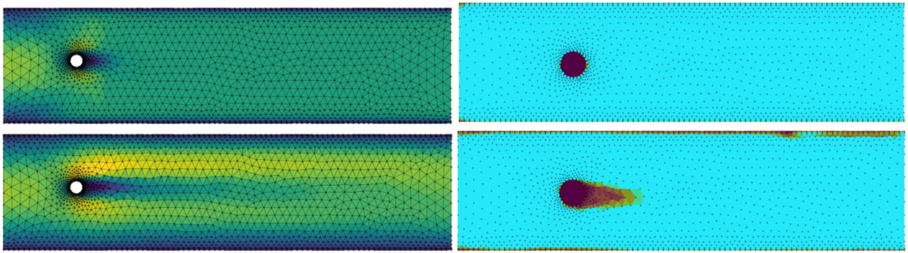


Figure 1: Baseline (left) vs. Our model (right)

Conclusions

- Proposed two changes to the original architecture:
  - Replacing the MLP with GCN in the processor
  - replacing the ReLU with Swish activation in encoder and decoder
- Reproduced the results for the CylinderFlow dataset, and compared with the results produced by our model
- Our model overfits compared to baseline due to lack of message propagation across edges, but the performance can be improved with:
  - neighborhood connectivity for each node, and
  - message-passing across edges

Future Work

- In future, the model can be used as a preconditioner to accelerate the convergence of iterative solvers e.g. GMRES(Generalized minimal residual method).
- Other GNN architectures like Graph Transformers can be used to learn the dynamics of varying physical systems.
- The predictions from our model can be used as an initial guess for numerical simulations to reduce computational cost.
- Current model can be upgraded to make predictions on larger meshes by showing good generalization ability.

Reference

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. 2017.

[1]https://github.com/google-deepmind/deepmind-research/tree/master/meshgraphnets

[2]https://github.com/echowve/meshGraphNets_pytorch