Technical University of Munich
Prof. Dr. Nils Thürey School of Computation, Information and Technology
Dr. Mario Lino, Benjamin Holzschuh, Patrick Schnell, Qiang Liu and Chengyun Wang Summer Semester 2025

Advanced Deep Learning for Physics (IN2298)
Exercise 8
Temporal Pressure Prediction with Graph Neural Networks

We will explore how deep learning models can be used to predict the temporal evolution of pressure on the surface of an elliptical cylinder immersed in a two-dimensional laminar channel flow. This flow configuration gives rise to a periodic vortex shedding pattern. The cylinder is placed inside a channel bounded by free-slip walls on the top and bottom. Our focus will be on predicting the pressure at a discrete set of nodes located along the contour of the elliptical body. We will not model the full flow field; rather, we will concentrate on the evolution of pressure at the surface of the cylinder.

We will work with a dataset that provides, for each surface node:

- its two-dimensional spatial coordinates,
- its distance to the upper and lower channel walls,
- the Reynolds number of the flow,
- the pressure values at the node over the five most recent time steps.

Our goal will be to construct a neural network model that can predict the pressure at the next time step, using only this local node information (and derived features). By autoregressively applying the model at inference time, we will be able to unroll the prediction forward in time. We will begin with a simple multilayer perceptron (MLP), treating each node independently. This will provide a baseline for evaluating the capability of pointwise models in capturing temporal dynamics. Following this, we will build a graph-based representation of the surface, where nodes are connected by mesh edges. Using this graph, we will train a graph neural network (GNN).

(0) Set Up

To implement the models in this exercise, we provide a Jupyter notebook that includes a minimal working example to get started. The notebook demonstrates how to:

- install and import the required `dgn4cfd` package,
- download and load the training and testing datasets,
- define and apply data transformations,
- visualize the nodes and pressure distribution over time,
- construct `DataLoader` objects for batching during training.

You are encouraged to reuse and adapt the provided code throughout this exercise.

(1) MLP

We begin by building a pointwise model that treats each node independently using a multilayer perceptron (MLP).

(a) **Training Function**

Write a generic training loop that is agnostic to the model architecture (MLP or GNN). Use the Adam optimizer with an initial learning rate of 10^{-4}. Use a learning rate scheduler with `ReduceLROnPlateau`, patience of 50 epochs, and a reduction factor of 0.1. Stop training when the learning rate drops below 10^{-6}.

(b) **MLP Architecture**

Implement an MLP that takes as input the concatenated node features: spatial coordinates (`graph.pos`), wall distances and Reynolds number (`graph.static`), and temporal pressure history (`graph.input`). The MLP should have 8 hidden layers with 128 neurons each.

(c) **Training the MLP**

Train the MLP to perform one-step-ahead prediction of pressure at the nodes. Use mean squared error (MSE) as the loss function.

(d) **Temporal Unrolling**

Test the MLP by unrolling it for 30 time steps on samples #10 and #20 from the test dataset. Plot the predicted pressure against the ground truth after 30 steps, and visualize the MSE as a function of time during the rollout.

(2) GNN and Fully Connected Graphs

In this section, we implement a graph neural network (GNN) operating on a fully connected graph, where each node can exchange information with every other node.

(a) **Fully Connected Graphs**

Create a new transformation class that connects each node to every other node in the graph. Assign input features to each edge (e.g., positional differences, distances, or any other relevant features). Store the edge connectivity in `graph.edge_index` and the edge attributes in `graph.edge_attr`. This transformation must be included in the transformation pipeline passed to the dataset class.

Refer to the PyTorch Geometric documentation for additional details on `graph.edge_index` and `graph.edge_attr`: https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

(b) **Message Passing**

Implement a `MessagePassing` block that defines a message-passing layer.

(c) **GNN Architecture**

Create a GNN by stacking two of the custom `MessagePassing` layers. Use a hidden feature dimension of 128 for intermediate representations.

(d) **GNN Training**

Train the GNN for a single epoch using the same training setup as in Question 1 (MLP). Report the training time.

(3) GNN and Mesh (Sparse) Graphs

In this section, we switch to a more physically meaningful graph structure by using the mesh connectivity along the perimeter of the ellipse.

(a) **Sparse Graph from Mesh Connectivity**

The transformation class below connects each node to its immediate neighbors along the ellipse perimeter:

```
class MeshEllipse:
"""
Transformation class that creates a mesh of edges connecting nodes around an ellipse.
"""
def __call__(self, graph):
    # Center the positions around origin
    pos = graph.pos - graph.pos.mean(0, keepdim=True)

    # Calculate angles of each node relative to x-axis
    angle = torch.arctan2(pos[:, 1], pos[:, 0])

    # Sort nodes by angle to get ordered sequence around ellipse
    idx = torch.argsort(angle)

    # Create edges between consecutive nodes (both directions)
    # Forward edges: node i -> node i+1
    # Backward edges: node i -> node i-1
```

```
graph.edge_index = torch.cat([
    torch.stack((idx, torch.roll(idx, -1, 0)), dim=0),  # Forward connections
    torch.stack((idx, torch.roll(idx,  1, 0)), dim=0),  # Backward connections
], dim=1)

# Sort edges by target node index for consistency
idx = torch.argsort(graph.edge_index[1])
graph.edge_index = graph.edge_index[:, idx]

# [TODO] Calculate edge features ...
graph.edge_attr = None

return graph
```

The edge indices are stored in `graph.edge_index`. Modify this class to also assign node features (`graph.edge_attr`) to the edges. Include this transformation in the dataset pipeline.

(b) **Extended Connectivity (2-Hop Neighbors)**

Create a second transformation class that extends the graph by adding edges between nodes that are two hops apart. Apply this transformation twice during graph construction so that each node ends up with 8 incoming and 8 outgoing edges. Visualize the resulting connectivity by plotting the edge structure for one graph.

(c) **GNN Training**

Train a GNN model that stacks 8 message-passing layers as defined in Question 2. Use the same feature size and training setup as in Question 1.

(d) **Temporal Unrolling**

As in Question 1, test the GNN by unrolling it for 30 time steps on samples #10 and #20 from the test dataset. Plot the predicted pressure against the ground truth after 30 steps, and visualize the MSE as a function of time during the rollout.

Submission instruction

Please upload a single PDF file containing your results along with your code for implementation tasks or your derivation for non-implementation tasks (LaTex typesetting). The uploaded PDF should only include the final code, so please trim empty spaces and your intermediate work before submitting.

The easiest way to generate such a PDF is by using Jupyter notebooks and LaTex (we recommend MiKTeX for Windows users). With Jupyter and LaTeX installed, you can create a PDF from your notebook by running *jupyter nbconvert –to pdf your-notebook.ipynb*

Additional information

This is an individual assignment. Plagiarism will result in the loss of eligibility for the bonus this semester.

If you have any questions about the exercises, please contact us via the forum on Moodle. If you need further face-to-face discussion, please join our weekly online Q&A session (every Monday at 15:00 and 16:00 via BBB).