Exercise_8

June 24, 2025

1 Advanced Deep Learning for Physics (IN2298)

1.1 Exercise 8 - Temporal Prediction of Surface Pressure Using Graph Neural Networks

[25]:
```
#%%capture
# Import libraries and packages
import dgn4cfd as dgn
import h5py
import random
import torch
from torch import nn
from torchvision import transforms
import matplotlib.pyplot as plt
from dgn4cfd import DataLoader
import torch
from torch import nn
from torch_geometric.utils import scatter
from torch import nn
import torch.optim as optim
```

[27]:
```
# Download the training datasets
TRAIN_DATASET = dgn.datasets.DatasetUrl.pOnEllipseTrain
TRAIN_DATASET_PATH = dgn.datasets.DatasetDownloader(TRAIN_DATASET).file_path
print(f"Training dataset downloaded to: {TRAIN_DATASET_PATH}")

# Download the testing datasets
TEST_DATASET  = dgn.datasets.DatasetUrl.pOnEllipseInDist
TEST_DATASET_PATH = dgn.datasets.DatasetDownloader(TEST_DATASET).file_path
print(f"Testing dataset downloaded to: {TEST_DATASET_PATH}")
```

```
Dataset already exists.
Training dataset downloaded to: ./pOnEllipseTrain.h5
Dataset already exists.
Testing dataset downloaded to: ./pOnEllipseInDist.h5
```

1.2 0. Setup

[28]:
```python
# Define a dataset class to handle the data
class MyDataset(dgn.datasets.Dataset):
    """
    Custom dataset class for handling flow data around an ellipse.
    Inherits from dgn.datasets.Dataset to leverage base functionality.
    """
    def __init__(self, path, hist_size=1, step_size=1, transform=None):
        """
        Initialize the dataset.

        Args:
            path (str): Path to the data file
            hist_size (int): Number of historical timesteps to include
            step_size (int): Step size between timesteps
            transform (callable, optional): Transformations to apply to the data
        """
        super().__init__(path, transform=transform)
        self.hist_size = hist_size
        self.step_size = step_size

    def data2graph(
        self,
        data: torch.Tensor,
        idx0: int,
        idx1: int,
        idx2: int,
        idx3: int,
    ):
        """
        Convert raw data tensor into a graph structure.

        Args:
            data (torch.Tensor): Raw data tensor
            idx0 (int): Start index for input sequence
            idx1 (int): End index for input sequence
            idx2 (int): Start index for target sequence
            idx3 (int): End index for target sequence

        Returns:
            dgn.Graph: Graph object containing processed data
        """
        # Count valid nodes (excluding NaN values)
        N = (data[:, 0] == data[:, 0]).sum()
        # Filter out NaN values
        data = data[:N]
```

```python
        # Initialize graph structure
        graph = dgn.Graph()
        # Center positions by subtracting mean
        graph.pos = data[:, :2] - data[:, :2].mean(dim=0)  # x, y coordinates
        # Stack static features: Reynolds number, distance to the lower and␣
↪upper walls
        graph.static = torch.stack([data[:, 2], data[:, 1], data[:, 3] - data[:
↪, 1]], dim=-1)
        # Extract input sequence with specified step size
        graph.input = data[:, 4 + idx0 : 4 + idx1 : self.step_size]
        # Extract target sequence with specified step size
        graph.target = data[:, 4 + idx2 : 4 + idx3 : self.step_size]
        return graph

    def get_sequence(
        self,
        idx: int,
        sequence_start: int = 0,
        n_target: int = 1,
    ):
        """
        Get a sequence of data for a specific index.

        Args:
            idx (int): Index of the data sample
            sequence_start (int): Starting point in the sequence
            n_target (int): Number of target timesteps

        Returns:
            dgn.Graph: Processed graph with input and target sequences
        """
        # Load data from HDF5 file
        h5_file = h5py.File(self.path, 'r')
        data = torch.tensor(h5_file['data'][idx], dtype=torch.float32)
        h5_file.close()

        # Calculate sequence indices
        idx0 = sequence_start
        idx1 = idx0 + self.hist_size * self.step_size
        idx2 = idx1 + (self.step_size - 1)
        idx3 = idx2 + n_target * self.step_size

        # Create graph and apply transformations
        graph = self.data2graph(data, idx0, idx1, idx2, idx3)
        return self.transform(graph) if self.transform is not None else graph
```

```python
    def __getitem__(
        self,
        idx: int
    ):
        """
        Get a random sequence from the dataset.

        Args:
            idx (int): Index of the data sample

        Returns:
            dgn.Graph: Processed graph with input and target sequences
        """
        # Generate random starting point for sequence
        sequence_start = random.randint(0, 100 - (self.hist_size + 1) * self.
↪step_size)
        return self.get_sequence(idx, sequence_start, n_target=1)
```

[29]:

```python
def plot(pos, x, x_label):
    pos = pos.cpu()
    x = x.cpu()
    # Plots
    top = pos[:, 1] >= 0.
    bottom = torch.logical_not(top)
    plt.plot(pos[top    , 0].cpu(), x[top    ].cpu(), 'k^', label=f'{x_label}␣
↪(top wall)')
    plt.plot(pos[bottom, 0].cpu(), x[bottom].cpu(), 'kv', label=f'{x_label}␣
↪(bottom wall)')
    plt.ylabel(r'$p$', fontsize=16)
    plt.xlabel(r'$x$', fontsize=16)
    plt.grid()
    plt.legend(fontsize=16)
    plt.show()

def plot_comparison(pos, x, y, x_label, y_label):
    pos = pos.cpu()
    x = x.cpu()
    y = y.cpu()
    # Plots
    top = pos[:, 1] >= 0.
    bottom = torch.logical_not(top)
    plt.plot(pos[top    , 0].cpu(), x[top    ].cpu(), 'k^', label=f'{x_label}␣
↪(top wall)')
    plt.plot(pos[bottom, 0].cpu(), x[bottom].cpu(), 'kv', label=f'{x_label}␣
↪(bottom wall)')
```

```
    plt.plot(pos[top,    0].cpu(), y[top   ].cpu(), 'b^', label=f'{y_label}␣
↪(top wall)',    alpha=0.4)
    plt.plot(pos[bottom, 0].cpu(), y[bottom].cpu(), 'bv', label=f'{y_label}␣
↪(bottom wall)', alpha=0.4)
    plt.ylabel(r'$p$', fontsize=16)
    plt.xlabel(r'$x$', fontsize=16)
    plt.grid()
    plt.legend(fontsize=16)
    plt.show()
```

[30]:
```
#Set up basic transformations and load the training data

# Define transformations
transform = transforms.Compose([
    dgn.transforms.ScaleAttr('input',  vmin=-1.05, vmax=0.84),         # Scale␣
↪the input fields
    dgn.transforms.ScaleAttr('target', vmin=-1.05, vmax=0.84),         # Scale␣
↪the target field
    dgn.transforms.ScaleAttr('static', vmin=500,   vmax=1000, idx=0), # Scale Re
])
train_dataset = MyDataset(
    path      = TRAIN_DATASET_PATH,
    hist_size = 5, # We use 5 previous time steps for the input
    transform = transform,
)
print('Number of samples:', len(train_dataset))
```

```
Number of samples: 5701
```

[31]:
```
print(train_dataset[0])  # Access the first sample to check the structure
print(train_dataset[0].pos.shape, train_dataset[0].static.shape,␣
↪train_dataset[0].input.shape, train_dataset[0].target.shape)
SAMPLE = 0 # Sample idx from the dataset
STEPS = 10 # Number of future time steps to predict
graph = train_dataset.get_sequence(SAMPLE, n_target=STEPS)
graph
```

```
Graph(pos=[60, 2], static=[60, 3], input=[60, 5], target=[60, 1])
torch.Size([60, 2]) torch.Size([60, 3]) torch.Size([60, 5]) torch.Size([60, 1])
```

[31]:
```
Graph(pos=[60, 2], static=[60, 3], input=[60, 5], target=[60, 10])
```

[32]:
```
# Create a DataLoader for batching and shuffling
dataloader = dgn.DataLoader(train_dataset, batch_size=64, shuffle=True,␣
↪num_workers=8, pin_memory=True, persistent_workers=True)
batch = next(iter(dataloader))
print(batch)
```

GraphBatch(pos=[4476, 2], static=[4476, 3], input=[4476, 5], target=[4476, 1], batch=[4476], ptr=[65])

1.3 1. MLP

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

1.3.1 1.a Training Function

[33]:
```python
def train_model(
    model,
    train_loader,
    optimizer,
    loss_fn,
    device,
    n_epochs=1,
    val_loader=None,
    scheduler=None,
    log_every=1,
    prepare_input=lambda batch: batch,         # Default: pass batch as-is
    predict=lambda model, x: model(x)          # Default: model(x)
):
    model.to(device)
    history = {'train_loss': [], 'val_loss': []}

    for epoch in range(1, n_epochs + 1):
        model.train()
        running_loss = 0.0

        for batch in train_loader:
            batch = {k: v.to(device) for k, v in batch.items()} if␣
↪isinstance(batch, dict) else batch
            input_ = prepare_input(batch)
            target = batch['target'].to(device) if isinstance(batch, dict) else␣
↪batch.target.to(device)

            optimizer.zero_grad()
            pred = predict(model, input_)
            loss = loss_fn(pred, target)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        avg_train_loss = running_loss / len(train_loader)
        history['train_loss'].append(avg_train_loss)
```

```
        if val_loader:
            model.eval()
            val_loss = 0.0
            with torch.no_grad():
                for batch in val_loader:
                    batch = {k: v.to(device) for k, v in batch.items()} if␣
↪isinstance(batch, dict) else batch
                    input_ = prepare_input(batch)
                    target = batch['target'].to(device) if isinstance(batch,␣
↪dict) else batch.target.to(device)

                    pred = predict(model, input_)
                    loss = loss_fn(pred, target)
                    val_loss += loss.item()

            avg_val_loss = val_loss / len(val_loader)
            history['val_loss'].append(avg_val_loss)

            if scheduler:
                scheduler.step(avg_val_loss)
        else:
            if scheduler:
                scheduler.step(avg_train_loss)

        if epoch % log_every == 0:
            print(f"Epoch {epoch:03d} | Train Loss: {avg_train_loss:.6f} | lr =␣
↪{optimizer.param_groups[0]['lr']}", end="")
            if val_loader:
                print(f" | Val Loss: {avg_val_loss:.6f}", end="")
            print()

        if optimizer.param_groups[0]['lr'] < 1e-6:
            print("Early stopping: learning rate too low.")
            break

    return history
```

1.3.2 1.b MPL Architecture

[34]:
```
class MLPressurePredictor(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.node_update_fn = torch.nn.Sequential(
            torch.nn.Linear(5*2, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
```

```
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 1),
        )

    def forward(self, x):
        return self.node_update_fn(x)
```

1.3.3 1.c Training MLP

[ ]:
```
def prepare_input(batch):
    # Concatenate node-level features
    x = torch.cat([batch.pos, batch.static, batch.input], dim=1)  # Shape: [N,␣
 ↪10]
    return x  # model will receive this as input
def predict(model, x):
    return model(x)  # x should have shape [N, 10]
```

[36]:
```
model = MLPressurePredictor()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
loss_fn = torch.nn.MSELoss()

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, factor=0.1, patience=50, eps=1e-7, verbose=True
)



history = train_model(
    model=model,
    train_loader=dataloader,
    optimizer=optimizer,
    loss_fn=loss_fn,
    device='cpu',
    n_epochs=200,
    val_loader=None,
    scheduler=scheduler,
    prepare_input=prepare_input,
    predict=predict,
```

```
    log_every=1
)
```

```
Epoch 001 | Train Loss: 0.133276 | lr = 0.0001
Epoch 002 | Train Loss: 0.022813 | lr = 0.0001
Epoch 003 | Train Loss: 0.002935 | lr = 0.0001
Epoch 004 | Train Loss: 0.001759 | lr = 0.0001
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[36], line 10
      3 loss_fn = torch.nn.MSELoss()
      5 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
      6     optimizer, factor=0.1, patience=50, eps=1e-7, verbose=True
      7 )
---> 10 history = train_model(
     11     model=model,
     12     train_loader=dataloader,
     13     optimizer=optimizer,
     14     loss_fn=loss_fn,
     15     device='cpu',
     16     n_epochs=200,
     17     val_loader=None,
     18     scheduler=scheduler,
     19     prepare_input=prepare_input,
     20     predict=predict,
     21     log_every=1
     22 )

Cell In[33], line 21, in train_model(model, train_loader, optimizer, loss_fn,␣
↪device, n_epochs, val_loader, scheduler, log_every, prepare_input, predict)
     18 model.train()
     19 running_loss = 0.0
---> 21 for batch in train_loader:
     22     batch = {k: v.to(device) for k, v in batch.items()} if␣
↪isinstance(batch, dict) else batch
     23     input_ = prepare_input(batch)

File ~/.local/lib/python3.10/site-packages/torch/utils/data/dataloader.py:630,␣
↪in _BaseDataLoaderIter.__next__(self)
    627 if self._sampler_iter is None:
    628     # TODO(https://github.com/pytorch/pytorch/issues/76750)
    629     self._reset()  # type: ignore[call-arg]
--> 630 data = self._next_data()
    631 self._num_yielded += 1
    632 if self._dataset_kind == _DatasetKind.Iterable and \
    633         self._IterableDataset_len_called is not None and \
```

```
    634         self._num_yielded > self._IterableDataset_len_called:

File ~/.local/lib/python3.10/site-packages/torch/utils/data/dataloader.py:1327,
  ↪in _MultiProcessingDataLoaderIter._next_data(self)
   1324     return self._process_data(data)
   1326 assert not self._shutdown and self._tasks_outstanding > 0
-> 1327 idx, data = self._get_data()
   1328 self._tasks_outstanding -= 1
   1329 if self._dataset_kind == _DatasetKind.Iterable:
   1330     # Check for _IterableDatasetStopIteration

File ~/.local/lib/python3.10/site-packages/torch/utils/data/dataloader.py:1293,
  ↪in _MultiProcessingDataLoaderIter._get_data(self)
   1289     # In this case, `self._data_queue` is a `queue.Queue`,. But we don'
   1290     # need to call `.task_done()` because we don't use `.join()`.
   1291 else:
   1292     while True:
-> 1293         success, data = self._try_get_data()
   1294         if success:
   1295             return data

File ~/.local/lib/python3.10/site-packages/torch/utils/data/dataloader.py:1131,
  ↪in _MultiProcessingDataLoaderIter._try_get_data(self, timeout)
   1118 def _try_get_data(self, timeout=_utils.MP_STATUS_CHECK_INTERVAL):
   1119     # Tries to fetch data from `self._data_queue` once for a given
  ↪timeout.
   1120     # This can also be used as inner loop of fetching without timeout,
  ↪with
   (…)
   1128     # Returns a 2-tuple:
   1129     #   (bool: whether successfully get data, any: data if successful
  ↪else None)
   1130     try:
-> 1131         data = self._data_queue.get(timeout=timeout)
   1132         return (True, data)
   1133     except Exception as e:
   1134         # At timeout and error, we manually check whether any worker ha
   1135         # failed. Note that this is the only mechanism for Windows to
  ↪detect
   1136         # worker failures.

File ~/Packages/anaconda3/envs/gnn310/lib/python3.10/multiprocessing/queues.py:
  ↪122, in Queue.get(self, block, timeout)
    120         self._rlock.release()
    121 # unserialize the data after having released the lock
--> 122 return _ForkingPickler.loads(res)
```

```
File ~/.local/lib/python3.10/site-packages/torch/multiprocessing/reductions.py:
↪501, in rebuild_storage_fd(cls, df, size)
    499 if storage is not None:
    500     return storage
--> 501 storage = cls._new_shared_fd_cpu(fd, size)
    502 shared_cache[fd_id(fd)] = StorageWeakRef(storage)
    503 return storage

KeyboardInterrupt:
```

I have truncated the output because too computationally costly for my PC. Below there are results for 200 epochs training.

1.3.4 1.d Temporal Unrolling

[ ]:
```
def rollout(model, dataset, sample, hist_size, step_size, rollout_steps,␣
↪device='cpu'):
    model.eval()

    # Load the initial graph for the sample with full history input
    graph = dataset.get_sequence(idx=sample, n_target=rollout_steps)

    # Move to device
    graph.pos = graph.pos.to(device)
    graph.static = graph.static.to(device)
    graph.input = graph.input.to(device)
    graph.target = graph.target.to(device)

    # We assume graph.input shape: [N_nodes, hist_size]
    # Start with the initial input sequence (length hist_size)
    current_input = graph.input.clone()

    predicted_pressures = []
    true_pressures = []
    mse_list = []

    loss_fn = torch.nn.MSELoss()

    for t in range(rollout_steps):
        # Predict pressure at next timestep
        x = torch.cat([graph.pos, graph.static, current_input],dim=1)
        pred = model(x)
        predicted_pressures.append(pred.detach().cpu())

        # Get ground truth pressure at this rollout step if available
        # Here we check if we have true data for step t in graph.target
        if t < graph.target.shape[1]:
```

```
            true_p = graph.target[:, t]
            true_pressures.append(true_p.cpu())
            mse = loss_fn(pred.squeeze(), true_p)
            mse_list.append(mse.item())
        else:
            # No more ground truth available (rollout longer than target)
            true_pressures.append(None)
            mse_list.append(None)

        # Prepare input for next step:
        # Remove oldest timestep, append prediction as newest input
        current_input = torch.cat([current_input[:, step_size:], pred], dim=1)

    # Stack all predicted and true pressures for plotting
    predicted_pressures = torch.stack(predicted_pressures, dim=1)  # shape␣
↪[N_nodes, rollout_steps]
    true_pressures = torch.stack([tp for tp in true_pressures if tp is not␣
↪None], dim=1) if any(tp is not None for tp in true_pressures) else None

    return predicted_pressures, true_pressures, mse_list
```

[189]:

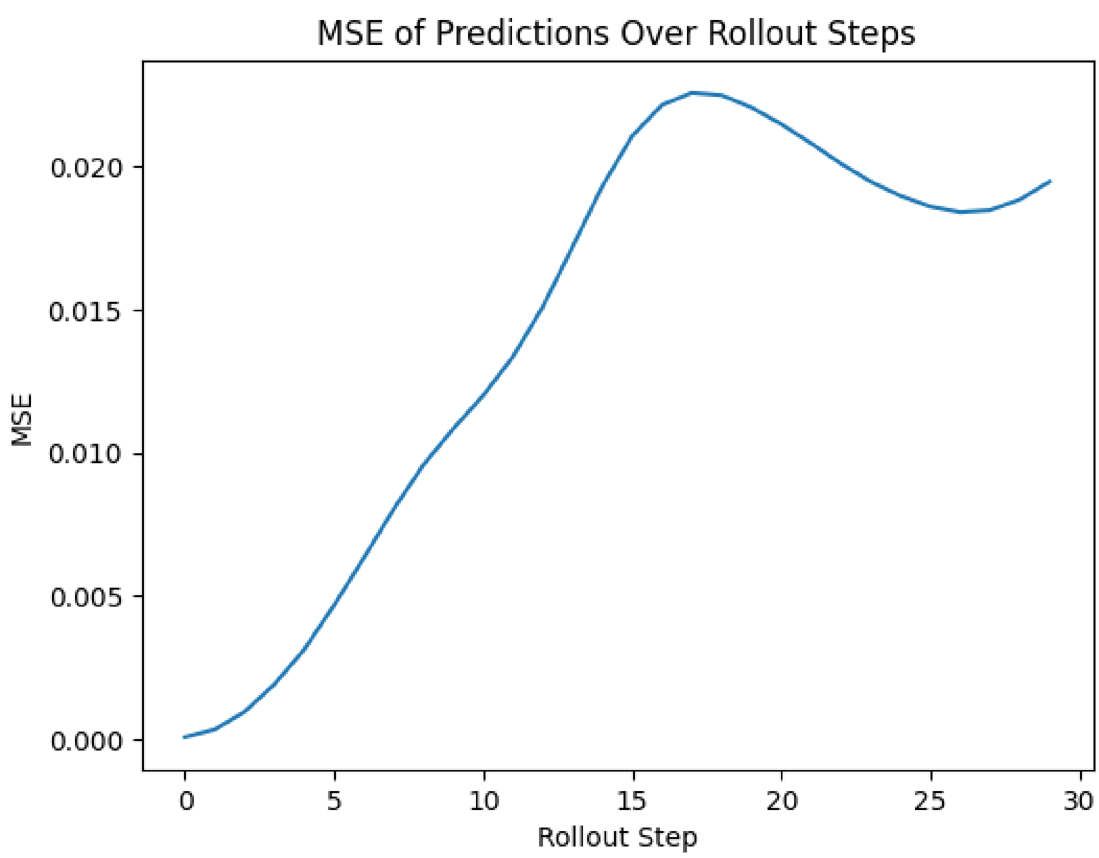
```
SAMPLE = 10
pred_pressures, true_pressures, mse_list = rollout(
    model=model,
    dataset=train_dataset,
    sample=SAMPLE,
    hist_size=5,
    step_size=1,
    rollout_steps=30,
    device='cpu'
)
```

[192]:

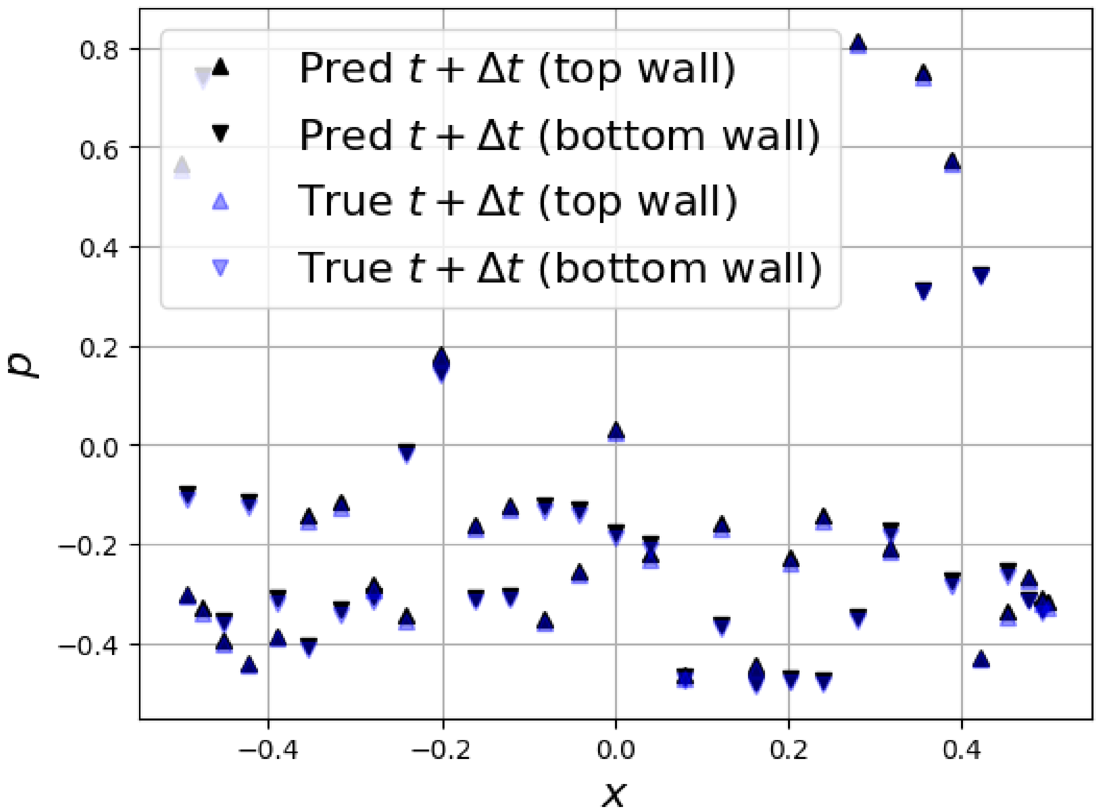
```
print('------------------------------------')
print(' Pressure comparison for sample', SAMPLE)
print('------------------------------------\n')
graph=train_dataset.get_sequence(SAMPLE, n_target=30)
plt.plot(mse_list, label='MSE per step')
plt.xlabel('Rollout Step')
plt.ylabel('MSE')
plt.title('MSE of Predictions Over Rollout Steps')
plt.show()

print('Pressure comparison with timestep 1 ')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 0],
    y=true_pressures[:, 0],
```

```
    x_label=r'Pred $t + \Delta t$',
    y_label=r'True $t + \Delta t$'
)

print('Pressure comparison with timestep 30')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 29],
    y=true_pressures[:, 29],
    x_label=r'Pred $t + 30 \Delta t$',
    y_label=r'True $t + 30 \Delta t$'
)
```

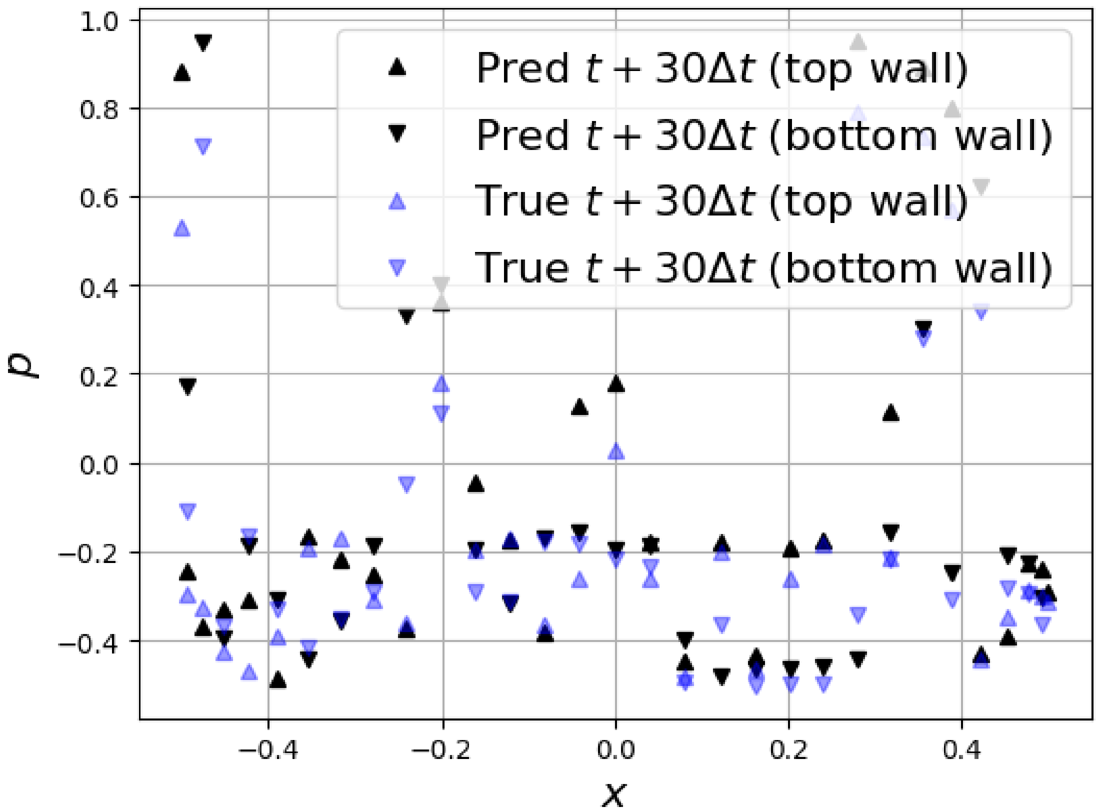
```
------------------------------------
 Pressure comparison for sample 0
------------------------------------
```




```
Pressure comparison with timestep 1
```



Pressure comparison with timestep 30



[193]:

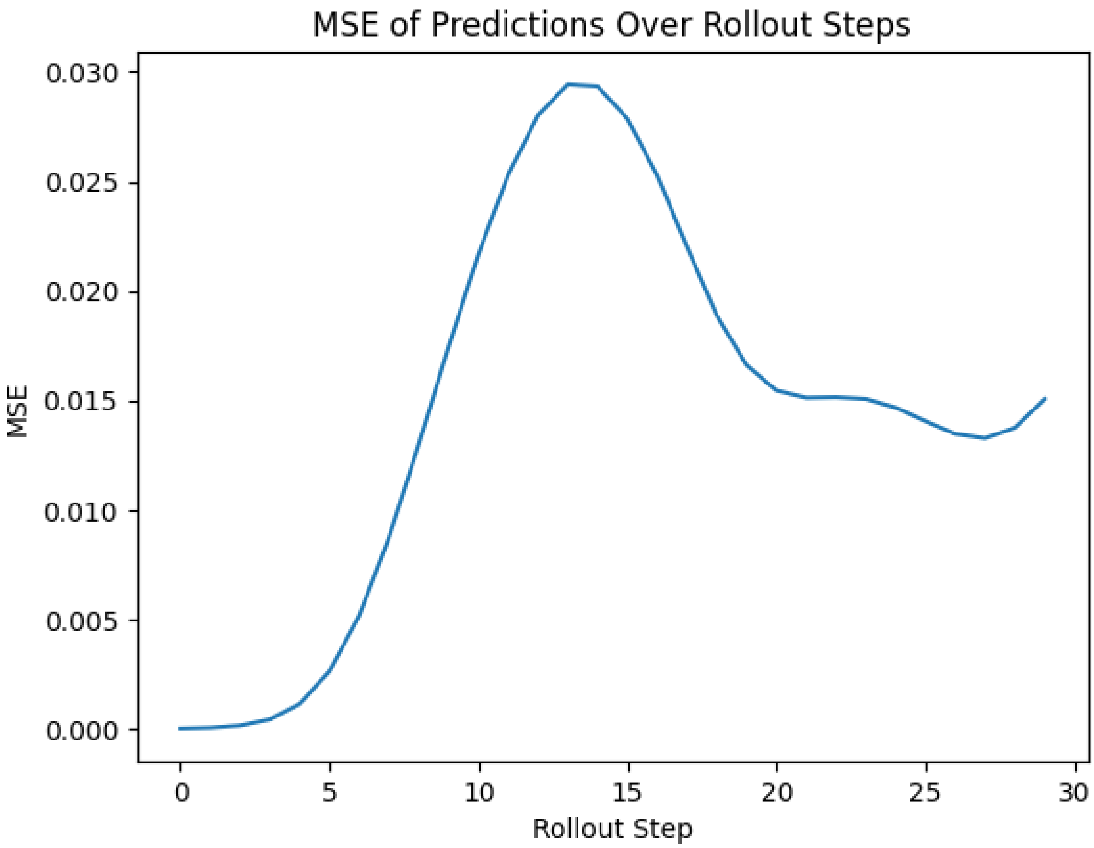
```
SAMPLE = 20
STEPS = 30 # Number of future time steps to predict
graph = train_dataset.get_sequence(idx=SAMPLE, n_target=STEPS)
pred_pressures, true_pressures, mse_list = rollout(
    model=model,
    dataset=train_dataset,
    sample=SAMPLE,
    hist_size=5,
    step_size=1,
    rollout_steps=STEPS,
    device='cpu'
)
```

[194]:

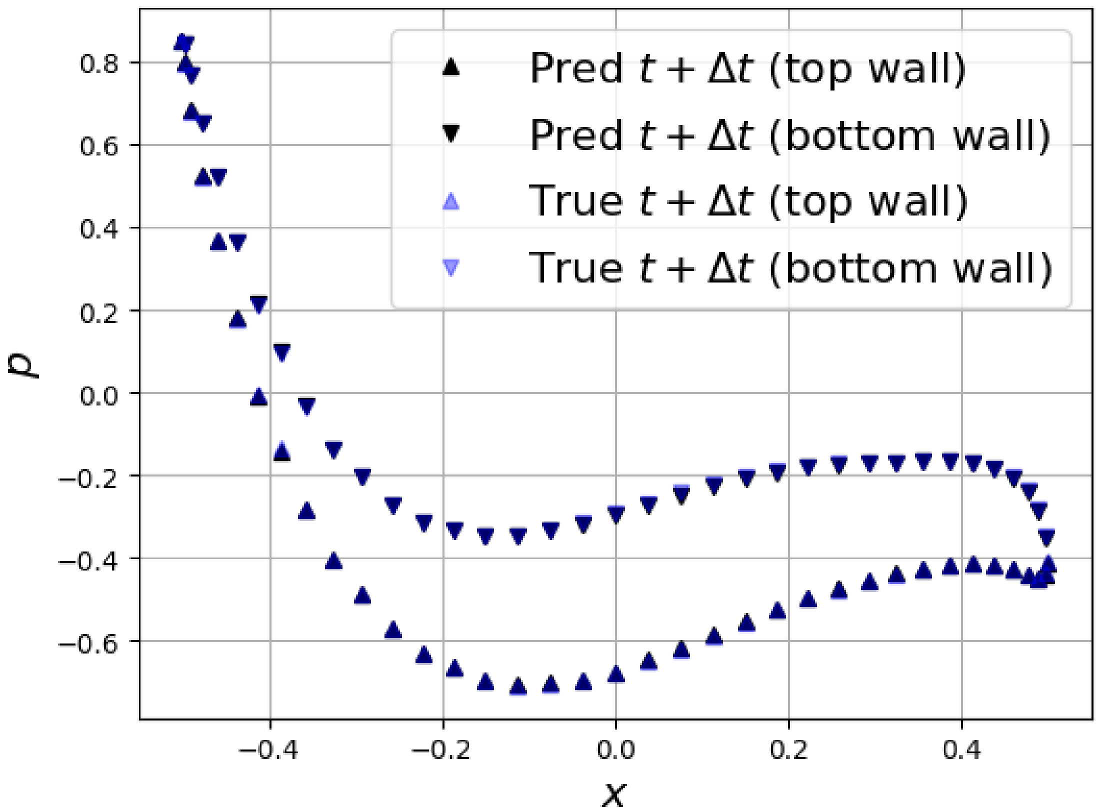
```
print('-------------------------------------')
print(' Pressure comparison for sample', SAMPLE)
print('-------------------------------------\n')

plt.plot(mse_list, label='MSE per step')
plt.xlabel('Rollout Step')
plt.ylabel('MSE')
plt.title('MSE of Predictions Over Rollout Steps')
```

```python
plt.show()

print('Pressure comparison with timestep 1 ')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 0],
    y=true_pressures[:, 0],
    x_label=r'Pred $t + \Delta t$',
    y_label=r'True $t + \Delta t$'
)

print('Pressure comparison with timestep 30')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 29],
    y=true_pressures[:, 29],
    x_label=r'Pred $t + 30 \Delta t$',
    y_label=r'True $t + 30 \Delta t$'
)
```

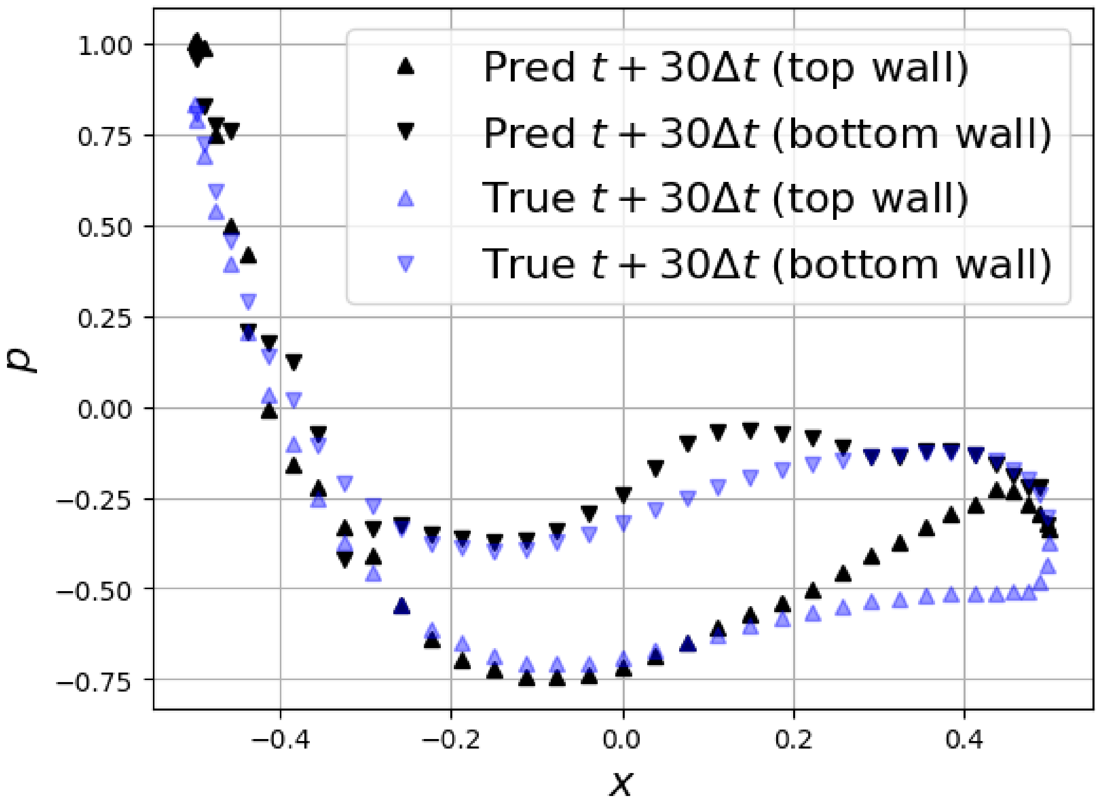
```
------------------------------------
 Pressure comparison for sample 20
------------------------------------
```



Pressure comparison with timestep 1



`Pressure comparison with timestep 30`



1.4 2. GNN and Fully Connected Graphs

1.4.1 2.a Fully Connected Graphs

[37]:
```
from torchvision import transforms
from torch_geometric.nn import knn_graph

class fully_connect:
    '''Transform to create a fully connected graph from node positions.'''

    def __init__(self):
        """
        Initialize the fully_connect transformation.
        This transformation will create a fully connected edge index based on␣
↪node positions.
        """
        pass


    def __call__(self, graph):
        """
```

```
        Apply the transformation to the graph.

        Args:
            graph (dgn.Graph): Input graph with node positions.

        Returns:
            dgn.Graph: Graph with fully connected edges.
        """
        N = graph.pos.size(0)
        # Create all possible directed edges (excluding self-loops)
        row = torch.arange(N).repeat_interleave(N)
        col = torch.arange(N).repeat(N)
        mask = row != col  # remove self-loops
        edge_index = torch.stack([row[mask], col[mask]], dim=0)
        graph.edge_index = edge_index

        # Add attributes: positional differences and distances
        graph.edge_attr = graph.pos[edge_index[0]] - graph.pos[edge_index[1]]
        graph.edge_attr = torch.cat([graph.edge_attr, graph.edge_attr.
↪norm(dim=1, keepdim=True)], dim=1)

        return graph
```

[38]:

```
#Set up basic transformations and load the training data

# Define transformations
transform = transforms.Compose([
    dgn.transforms.ScaleAttr('input',  vmin=-1.05, vmax=0.84),        # Scale␣
↪the input fields
    dgn.transforms.ScaleAttr('target', vmin=-1.05, vmax=0.84),        # Scale␣
↪the target field
    dgn.transforms.ScaleAttr('static', vmin=500,   vmax=1000, idx=0), # Scale Re
    fully_connect(),
])
train_dataset = MyDataset(
    path      = TRAIN_DATASET_PATH,
    hist_size = 5, # We use 5 previous time steps for the input
    transform = transform,
)
print('Number of samples:', len(train_dataset))

SAMPLE = 0 # Sample idx from the dataset
STEPS = 10 # Number of future time steps to predict
graph = train_dataset.get_sequence(SAMPLE, n_target=STEPS)
print(graph)  # Check the structure of the graph
print(f'shape edge: {graph.edge_index.shape}')  # Should be [2, N_edges] where␣
↪N_edges=N*(N-1)
```

```
Number of samples: 5701
Graph(pos=[60, 2], static=[60, 3], input=[60, 5], target=[60, 10],
edge_index=[2, 3540], edge_attr=[3540, 3])
shape edge: torch.Size([2, 3540])
```

1.4.2 2.b Message Passing

[67]:
```
class MyMessagePassing(torch.nn.Module):
    """
    Custom message passing layer for the pressure prediction model.
    This layer will aggregate messages from neighboring nodes.
    """
    def __init__(self):
        super().__init__()
        self.edge_update_fn = torch.nn.Sequential(
            torch.nn.Linear(23, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 1),
        )
        self.node_update_fn = torch.nn.Sequential(
            torch.nn.Linear(11, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 10),
        )

    def forward(self, x,edge_idx,edge_attr):

        sender_idx, receiver_idx = edge_idx

        # Edge Update
        e = self.edge_update_fn(torch.cat([x[sender_idx], x[receiver_idx],
↪edge_attr], dim=1)) # 10 x 10 x 3 = 23

        # Aggregation
        e = scatter(e, receiver_idx, dim=0, reduce='sum')  # 60 x 1

        # Node Update
        x = self.node_update_fn(torch.cat([x,e],dim=1)) # 11

        return x, e
```

1.4.3 2.c GNN Architecture

[77]:
```
class MyGNN(torch.nn.Module):
    """
    Graph Neural Network for pressure prediction.
    This model uses message passing to predict pressure at each node.
    """
    def __init__(self):
        super().__init__()

        self.mp1 = MyMessagePassing()
        self.mp2 = MyMessagePassing()
        self.predict_head = torch.nn.Linear(10, 1)

    def forward(self, x, edge_index, edge_attr):
        # Perform message passing
        x, e = self.mp1(x, edge_index, edge_attr)
        x, e = self.mp2(x, edge_index, edge_attr)


        return self.predict_head(x)
```

1.4.4 2.d GNN Training

[78]:
```
def prepare_input(batch):
    return (
        torch.cat([batch.pos,batch.static,batch.input],dim=1),                #␣
↪Node features
        batch.edge_index,                                                     #␣
↪Edge indices
        batch.edge_attr,                                                      #␣
↪Edge features
    )

def predict(model, inputs):
    x, edge_index, edge_attr = inputs
    return model(x, edge_index, edge_attr)
```

[79]:
```
# Create a DataLoader for batching and shuffling
dataloader = dgn.DataLoader(train_dataset, batch_size=32, shuffle=True,␣
↪num_workers=8, pin_memory=True, persistent_workers=True)
batch = next(iter(dataloader))
print(batch)
model = MyGNN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
loss_fn = torch.nn.MSELoss()
```

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, factor=0.1, patience=50, eps=1e-7, verbose=True
)

history = train_model(
    model=model,
    train_loader=dataloader,
    optimizer=optimizer,
    loss_fn=loss_fn,
    device='cpu',
    n_epochs=1,
    val_loader= None,
    scheduler=scheduler,
    prepare_input=prepare_input,
    predict=predict,
    log_every=1
)
```

```
GraphBatch(pos=[2188, 2], static=[2188, 3], input=[2188, 5], target=[2188, 1],
edge_index=[2, 150628], edge_attr=[150628, 3], batch=[2188], ptr=[33])
Epoch 001 | Train Loss: 0.051747 | lr = 0.0001
```

time = 4 min for 1 epoch

1.5 3 GNN and Mesh (Sparse) Graphs

1.5.1 3.a Sparse Graph from Mesh Connectivity

[85]:
```
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 ...

        # Compute edge features: relative position and distance
        src, dst = graph.edge_index
        rel_vec = pos[dst] - pos[src]               # Vector from source to␣
↪target
        distance = rel_vec.norm(dim=1, keepdim=True)  # Euclidean distance
        graph.edge_attr = None

        # Edge features: [dx, dy, distance]
        edge_attr = torch.cat([rel_vec, distance], dim=1)
        graph.edge_attr = edge_attr
        return graph
```

[86]:
```
#Set up basic transformations and load the training data

# Define transformations
transform = transforms.Compose([
    dgn.transforms.ScaleAttr('input',  vmin=-1.05, vmax=0.84),        # Scale␣
↪the input fields
    dgn.transforms.ScaleAttr('target', vmin=-1.05, vmax=0.84),        # Scale␣
↪the target field
    dgn.transforms.ScaleAttr('static', vmin=500,   vmax=1000, idx=0), # Scale Re
    MeshEllipse(),  # Apply the mesh transformation
])
train_dataset = MyDataset(
    path      = TRAIN_DATASET_PATH,
    hist_size = 5, # We use 5 previous time steps for the input
    transform = transform,
)
print('Number of samples:', len(train_dataset))

SAMPLE = 0 # Sample idx from the dataset
STEPS = 10 # Number of future time steps to predict
graph = train_dataset.get_sequence(SAMPLE, n_target=STEPS)
print(graph)  # Check the structure of the graph
print(f'shape edge: {graph.edge_index.shape}')  # Should be [2, N_edges] where␣
↪N_edges=N*(N-1)
```

```
Number of samples: 5701
Graph(pos=[60, 2], static=[60, 3], input=[60, 5], target=[60, 10],
edge_index=[2, 120], edge_attr=[120, 3])
shape edge: torch.Size([2, 120])
```

1.5.2 3.b Extended Connectivity (2-Hop Neighbors)

[90]:

```python
import torch
from torch_geometric.utils import to_scipy_sparse_matrix, from_scipy_sparse_matrix
import scipy.sparse as sp

class AddTwoHopEdges:
    """
    Adds edges between nodes that are two hops apart.
    """
    def __call__(self, graph):
        # Get current edges
        edge_index = graph.edge_index

        # Build sparse adjacency matrix (unweighted, undirected for 2-hop purposes)
        num_nodes = graph.pos.size(0)
        A = to_scipy_sparse_matrix(edge_index, num_nodes=num_nodes).tocsr()

        # Multiply A @ A to find 2-hop connections
        A2 = A @ A

        # Remove self-loops
        A2.setdiag(0)

        # Remove existing edges (keep only the new 2-hop ones)
        A_existing = A.copy()
        A2 = A2 - A_existing
        A2.eliminate_zeros()

        # Convert back to edge index format
        new_edge_index, _ = from_scipy_sparse_matrix(A2)
        # Concatenate old and new edges
        full_edge_index = torch.cat([edge_index, new_edge_index], dim=1)
        # Optional: remove duplicate edges
        full_edge_index = torch.unique(full_edge_index, dim=1)
        graph.edge_index = full_edge_index

        # Update edge attributes
        src, dst = graph.edge_index
        rel_vec = graph.pos[dst] - graph.pos[src]
        distance = rel_vec.norm(dim=1, keepdim=True)
        edge_attr = torch.cat([rel_vec, distance], dim=1)
        graph.edge_attr = edge_attr
```

```
        return graph
```

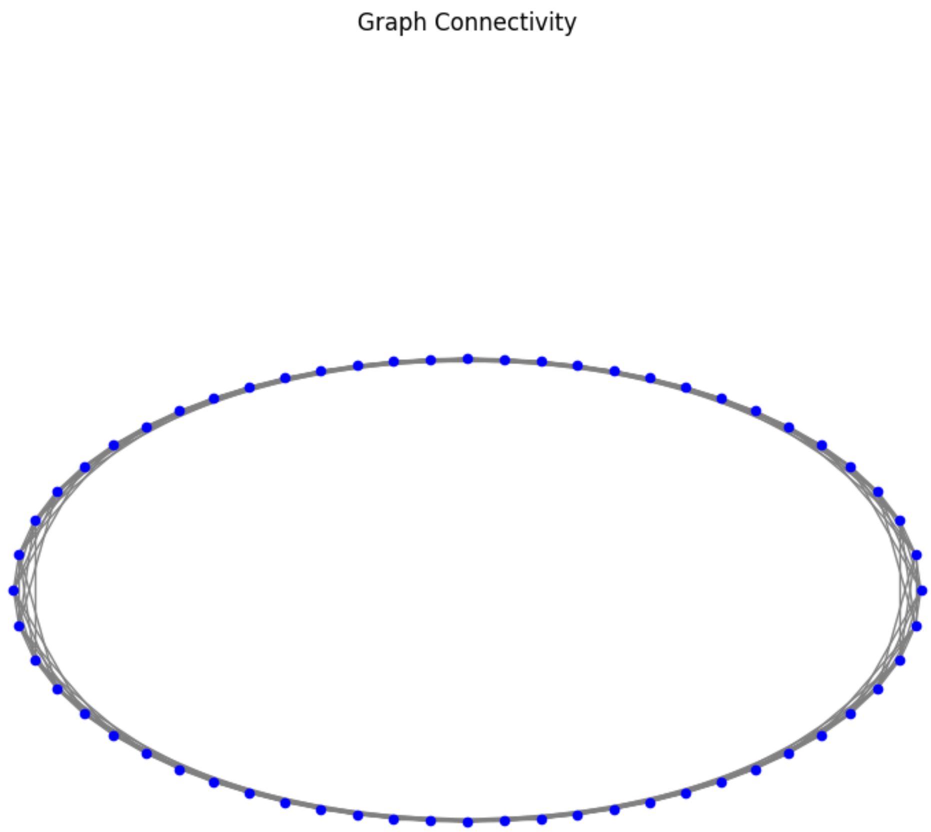
[91]:
```
#Set up basic transformations and load the training data

# Define transformations
transform = transforms.Compose([
    dgn.transforms.ScaleAttr('input',  vmin=-1.05, vmax=0.84),          # Scale␣
 ↪the input fields
    dgn.transforms.ScaleAttr('target', vmin=-1.05, vmax=0.84),          # Scale␣
 ↪the target field
    dgn.transforms.ScaleAttr('static', vmin=500,   vmax=1000, idx=0), # Scale Re
    MeshEllipse(),  # Apply the mesh transformation
    AddTwoHopEdges(),  # Add edges between nodes that are two hops apart
    AddTwoHopEdges(),  # Add edges between nodes that are two hops apart
])
train_dataset = MyDataset(
    path      = TRAIN_DATASET_PATH,
    hist_size = 5, # We use 5 previous time steps for the input
    transform = transform,
)
print('Number of samples:', len(train_dataset))

SAMPLE = 0 # Sample idx from the dataset
STEPS = 10 # Number of future time steps to predict
graph = train_dataset.get_sequence(SAMPLE, n_target=STEPS)
print(graph)  # Check the structure of the graph
print(f'shape edge: {graph.edge_index.shape}')  # Should be [2, N_edges] where␣
 ↪N_edges=N*(N-1)
```

```
Number of samples: 5701
Graph(pos=[60, 2], static=[60, 3], input=[60, 5], target=[60, 10],
edge_index=[2, 480], edge_attr=[480, 3])
shape edge: torch.Size([2, 480])
```

[93]:
```
import matplotlib.pyplot as plt
import networkx as nx
from torch_geometric.utils import to_networkx

def visualize_graph(graph, title="Graph Connectivity"):
    # Convert to networkx
    G = to_networkx(graph, to_undirected=True)
    pos = {i: (x, y) for i, (x, y) in enumerate(graph.pos.tolist())}

    # Plot
    plt.figure(figsize=(8, 8))
    nx.draw(G, pos, node_size=20, edge_color="gray", node_color="blue",␣
 ↪with_labels=False)
```

```python
    plt.title(title)
    plt.axis("equal")
    plt.show()
```

[95]:
```python
graph = train_dataset[0]
visualize_graph(graph)
```



1.5.3 3c GNN Training

[97]:
```
class MeshGNN(torch.nn.Module):
    """
    Graph Neural Network for pressure prediction using a mesh-based approach.
    This model uses message passing to predict pressure at each node in a mesh␣
 ↪structure.
    """
    def __init__(self):
        super().__init__()

        self.mp1 = MyMessagePassing()
        self.mp2 = MyMessagePassing()
        self.predict_head = torch.nn.Linear(10, 1)

    def forward(self, x, edge_index, edge_attr):
        # Perform message passing
        x, e = self.mp1(x, edge_index, edge_attr)
        x, e = self.mp2(x, edge_index, edge_attr)
        x, e = self.mp1(x, edge_index, edge_attr)
        x, e = self.mp2(x, edge_index, edge_attr)
        x, e = self.mp1(x, edge_index, edge_attr)
        x, e = self.mp2(x, edge_index, edge_attr)
        x, e = self.mp1(x, edge_index, edge_attr)
        x, e = self.mp2(x, edge_index, edge_attr)

        return self.predict_head(x)
```

[98]:
```
# Train the MeshGNN model
model = MeshGNN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
loss_fn = torch.nn.MSELoss()
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, factor=0.1, patience=50, eps=1e-7, verbose=True
)
history = train_model(
    model=model,
    train_loader=dataloader,
    optimizer=optimizer,
    loss_fn=loss_fn,
    device='cpu',
    n_epochs=1,
    val_loader=None,
    scheduler=scheduler,
    prepare_input=prepare_input,
    predict=predict,
    log_every=1
```

```
)
```

```
/home/scaio/.local/lib/python3.10/site-packages/torch/optim/lr_scheduler.py:60:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to
access the learning rate.
  warnings.warn(
```

```
The Kernel crashed while executing code in the current cell or a previous cell.

Please review the code in the cell(s) to identify a possible cause of the␣
↪failure.

Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info.

View Jupyter <a href='command:jupyter.viewOutput'>log</a> for further details.
```

The server crashed because too few memeory in my PC

1.5.4 3.d Temporal Unrolling

[ ]:
```
SAMPLE = 10
pred_pressures, true_pressures, mse_list = rollout(
    model=model,
    dataset=train_dataset,
    sample=SAMPLE,
    hist_size=5,
    step_size=1,
    rollout_steps=30,
    device='cpu'
)

print('-------------------------------------')
print(' Pressure comparison for sample', SAMPLE)
print('-------------------------------------\n')
graph=train_dataset.get_sequence(SAMPLE, n_target=30)
plt.plot(mse_list, label='MSE per step')
plt.xlabel('Rollout Step')
plt.ylabel('MSE')
plt.title('MSE of Predictions Over Rollout Steps')
plt.show()

print('Pressure comparison with timestep 1 ')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 0],
    y=true_pressures[:, 0],
```

```
    x_label=r'Pred $t + \Delta t$',
    y_label=r'True $t + \Delta t$'
)

print('Pressure comparison with timestep 30')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 29],
    y=true_pressures[:, 29],
    x_label=r'Pred $t + 30 \Delta t$',
    y_label=r'True $t + 30 \Delta t$'
)
```

[ ]:
```
SAMPLE = 20
pred_pressures, true_pressures, mse_list = rollout(
    model=model,
    dataset=train_dataset,
    sample=SAMPLE,
    hist_size=5,
    step_size=1,
    rollout_steps=30,
    device='cpu'
)

print('-------------------------------------')
print(' Pressure comparison for sample', SAMPLE)
print('-------------------------------------\n')
graph=train_dataset.get_sequence(SAMPLE, n_target=30)
plt.plot(mse_list, label='MSE per step')
plt.xlabel('Rollout Step')
plt.ylabel('MSE')
plt.title('MSE of Predictions Over Rollout Steps')
plt.show()

print('Pressure comparison with timestep 1 ')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 0],
    y=true_pressures[:, 0],
    x_label=r'Pred $t + \Delta t$',
    y_label=r'True $t + \Delta t$'
)

print('Pressure comparison with timestep 30')
plot_comparison(
    pos=graph.pos,
    x=pred_pressures[:, 29],
```

```
    y=true_pressures[:, 29],
    x_label=r'Pred $t + 30 \Delta t$',
    y_label=r'True $t + 30 \Delta t$'
)
```

[ ]:
```
%%capture
!pip install nbconvert
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc

!jupyter nbconvert --to pdf --output /home/scaio/ADL4P/Exercise_8/Exercise_8.
↪pdf /home/scaio/ADL4P/Exercise_8/Exercise_8.ipynb
```