Laboratory 2.1: Graph Neural Networks (GNNs)

Objectives:

- Implement a data-driven model based on graph neural networks (GNN) to estimate the pressures at the nodes of a water distribution system.
- Analyse the training performance of the implemented GNN.
- Evaluate the execution performance of the data-driven model to emulate nodal pressures.
- Transfer the developed data-driven model to a new case study.

Completion requirements:

By the end of this notebook, you should have:

- Implemented all code cells for:
  - defining a GNN-model.
  - training the GNN-model
  - assesing the performance of the GNN-model.
  - transfering and assesing the GNN-model.
- Answered the analysis questions on each section.

🔙 Background

Water utilities rely on hydrodynamic models to properly design and control water distribution systems (WDSs). These physically-based models, such as EPANET, compute the state of the system, i.e., the flow rates and pressures at all the pipes and junctions, as illustrated in Figure 1.

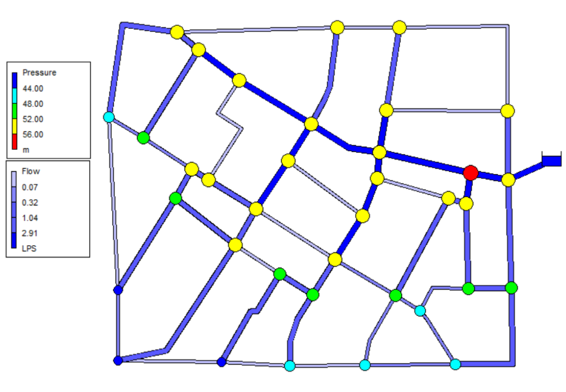

Figure 1. EPANET results in WDS in Fossolo, Italy. Pressure and flows are presented with different color schemes.

Hydrodynamic models provide valuable insight into the functioning of the system. However, the computational speed of these models is often insufficient for some applications in civil engineering such as optimisation of design or criticality assessment, especially in large search space problems.

One alternative to address this issue is developing data-driven models. These models are trained using results calculated using the original model (EPANET, in this case) in multiple scenarios. The objective of the data-driven models is to estimate the output of the original model but in a shorter time.

⛔ Problem definition

As part of a re-design of the Fossolo WDS, the water utility company has decided to use an optimisation approach. Aware that the established optimisation algorithms for designing water systems require a large number of scenarios, the company has decided to create a data-driven model to accelerate the process.

From a graph machine learning perspective, this problem can be framed as a node regression. This is, given the network topology and input features at the nodes and/or edges, we want to use a GNN-model to infer the value of a variable at each node.

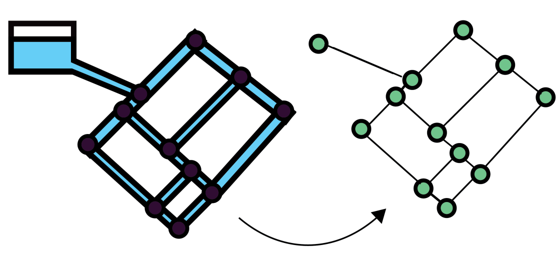

Figure 2. Representation of water network as a graph.

As a consultant for the water utility, your task is to develop a GNN-based tool that can perform pressure estimation at the nodes. This system will process different configurations of network characteristics (e.g., pipe diameters, nodal demands, type of nodes), and estimate the pressure at each node of the system.

This tool is developed in a *supervised learning* manner by employing the scenarios datasets already prepared by the company.

📓 Libraries

To run this notebook you need to have installed the following packages:

- Pandas
- Numpy
- NetworkX
- Scikit-learn
- Pytorch
- Pytorch geometric

In [ ]:
```python
import time
import torch
import pickle
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
from torch_geometric.nn import TAGConv
from torch.nn import Sequential, Linear, ReLU
from torch_geometric.loader import DataLoader as pyg_DataLoader
```

📚 Database

The Fossolo water network has 37 nodes and 58 pipes. The network is represented as an undirected graph; this means that each pipe is represented as 2 edges, one incoming and one outgoing. In summary, each graph has 37 nodes and 116 edges.

For this WDS, we already have a database with 1000 scenarios. This database was created by generating multiple combinations of diameters at the pipes and water consumption at the nodes.

The dataset was created using a Python package, based on EPANET, designed to simulate the behaviour of water distribution networks. Each scenario is represented as a Data object that contains edge features, node features, and the target feature. All of the features are already normalized based on a min-max scaling.

The diameter at the pipes is the only edge feature. The node features consist of water consumption, node_type (0 for junction and 1 for reservoirs), junction elevation, and reservoir elevation.

Run the following cell to load the datasets from the pickle files.

In [ ]:
```python
with open('Training_dataset_WDS.p', 'rb') as handle:
    tra_dataset_pyg = pickle.load(handle)
with open('Validation_dataset_WDS.p', 'rb') as handle:
    val_dataset_pyg = pickle.load(handle)
with open('Test_dataset_WDS.p', 'rb') as handle:
    tst_dataset_pyg = pickle.load(handle)
```

In [ ]:
```python
print('Number of training examples:',   len(tra_dataset_pyg))
print('Number of validation examples:', len(val_dataset_pyg))
print('Number of test examples:',       len(tst_dataset_pyg))
```

We can inspect the content of one example:

In [ ]:
```python
tra_dataset_pyg[0]
```

Each example in the database contains the following information:

- **x:** Input node features with shape [Nodes, Node features]. This matrix is composed of 4 normalized variables for each node. Namely,
  - Normalized water consumption. Original values between 0 and 0.007 cubic meters per second.
  - Normalized junction elevation. Original values between 0 and 67.9 m.
  - Normalized reservoir elevation. Original values between 0 and 121 m.
  - One-hot encoding for type of node (1 for reservoir, 0 for junction)
- **edge_attr:** Input edge features with shape [Edges, Edge features]. This matrix is composed of 1 normalized variable for each pipe. Namely,
  - Normalized diameter. Original values between 0.025 and 0.4750.
- **edge_index:** Graph connectivity in COO format with shape [2, num_edges]
- **y:** Output node feature, i.e. target feature. Shape [Nodes, 1]. Original values between 0 and 59.56 mH2O.

💻 GNN training

Instructions: Define a GNN model class, instantiate it, and train it.

🕵️ Questions

Imagine you are tasked with solving this problem using a GNN model.

- What dimensions do the inputs and outputs have?
- Which methods/architectures would you try first and why? (More than correctness, think of plausible components to test based on the problem description.)
- What hyperparameters define the structure of the model?

Write your answers in the following cell

Answers:

Model definition - Example

Below follows an example of a GNN model. You can use this code as a template for your own GNN model.

In [ ]:
```python
class GNN_Example(nn.Module):
  """
    This class defines a PyTorch module that takes in a graph represented in the PyTorch Geometric Data format,
    and outputs a tensor of predictions for each node in the graph. The model consists of one or more TAGConv layers,
    which are a type of graph convolutional layer.

    Args:
        node_dim (int): The number of node inputs.
        edge_dim (int): The number of edge inputs.
        output_dim (int, optional): The number of outputs (default: 1).
        hidden_dim (int, optional): The number of hidden units in each GNN layer (default: 50).
        n_gnn_layers (int, optional): The number of GNN layers in the model (default: 1).
        K (int, optional): The number of hops in the neighbourhood for each GNN layer (default: 2).
        dropout_rate (float, optional): The dropout rate to be applied to the output of each GNN layer (default: 0).

    """
  def __init__(self, node_dim, edge_dim, output_dim=1, hidden_dim=50, n_gnn_layers=1, K=2, dropout_rate=0):
    super().__init__()
    self.node_dim = node_dim
    self.edge_dim = edge_dim
    self.output_dim = output_dim
    self.hidden_dim = hidden_dim
    self.n_gnn_layers = n_gnn_layers
    self.K = K
    self.dropout_rate = dropout_rate

    self.convs = nn.ModuleList()

    if n_gnn_layers == 1:
      self.convs.append(TAGConv(node_dim, output_dim, K=K))
    else:
      self.convs.append(TAGConv(node_dim, hidden_dim, K=K))

      for l in range(n_gnn_layers-2):
          self.convs.append(TAGConv(hidden_dim, hidden_dim, K=K))

      self.convs.append(TAGConv(hidden_dim, output_dim, K=K))

  def forward(self, data):
      """Applies the GNN to the input graph.

        Args:
            data (Data): A PyTorch Geometric Data object representing the input graph.

        Returns:
            torch.Tensor: The output tensor of the GNN.

        """
      x = data.x
      edge_index = data.edge_index
      edge_attr = data.edge_attr

      for i in range(len(self.convs)-1):
          x = self.convs[i](x=x, edge_index=edge_index, edge_weight=edge_attr)
          x = nn.Dropout(self.dropout_rate, inplace=False)(x)
          x = nn.PReLU()(x)

      x = self.convs[-1](x=x, edge_index=edge_index, edge_weight=edge_attr)
      # x = nn.Sigmoid()(x)

      return x
```

Model instantiation

In [ ]:
```python
# Set model parameters
node_dim =   tra_dataset_pyg[0].x.shape[1]
edge_dim =   tra_dataset_pyg[0].edge_attr.shape[1]
output_dim = tra_dataset_pyg[0].y.shape[1]
hidden_dim = 16
n_gnn_layers = 3
K=1
dropout_rate = 0

# Create model
model = GNN_Example(node_dim, edge_dim, output_dim, hidden_dim, n_gnn_layers, K, dropout_rate)
print(model)
```

Define your own GNN

In the following cells, define and instatiate your own GNN model. You can use already implemented layers from Pytorch Geometric library.

In [ ]:
```python
class My_GNN(nn.Module):
    def __init__(self, node_dim, edge_dim, output_dim, hidden_dim, n_gnn_layers, K, dropout_rate):
        super().__init__()
        ...

    def forward(self, data):
        return ...
```

In [ ]:
```python
# Set model parameters
...

# Create model
model = My_GNN(...)
print(model)
```

🏋️Train a Graph Neural Network

Train your GNN model.

Recommendations for first trial

- Use dropout and early stopping to reduce overfitting
- Use the Adam optimizer, and set learning rate to 0.001
- Use a batch size of 16

Training epoch function

In [ ]:
```python
def train_epoch(model, loader, optimizer, device='cpu'):
  """
    Trains a neural network model for one epoch using the specified data loader and optimizer.

    Args:
        model (nn.Module): The neural network model to be trained.
        loader (DataLoader): The PyTorch Geometric DataLoader containing the training data.
        optimizer (torch.optim.Optimizer): The PyTorch optimizer used for training the model.
        device (str): The device used for training the model (default: 'cpu').

    Returns:
        float: The mean loss value over all the batches in the DataLoader.

    """
  return ...
```

Testing epoch function

In [ ]:
```python
def evaluate_epoch(model, loader, device='cpu'):
    """
    Evaluates the performance of a trained neural network model on a dataset using the specified data loader.

    Args:
        model (nn.Module): The trained neural network model to be evaluated.
        loader (DataLoader): The PyTorch Geometric DataLoader containing the evaluation data.
        device (str): The device used for evaluating the model (default: 'cpu').

    Returns:
        float: The mean loss value over all the batches in the DataLoader.

    """

    return ...
```

Optimization of the Model

In [ ]:
```python
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

In [ ]:
```python
# Set training parameters
learning_rate = ...
batch_size = ...
num_epochs = ...

# Create the optimizer to train the neural network via back-propagation
optimizer = torch.optim.Adam(params=model.parameters(), lr=learning_rate)

# Create the training and validation dataloaders to "feed" data to the GNN in batches
tra_loader = pyg_DataLoader(tra_dataset_pyg, batch_size=batch_size, shuffle=True)
val_loader = pyg_DataLoader(val_dataset_pyg, batch_size=batch_size, shuffle=False)
```

In [ ]:
```python
#create vectors for the training and validation loss
train_losses = []
val_losses = []
patience = 5       # patience for early stopping

#start measuring time
start_time = time.time()

for epoch in range(1, num_epochs+1):
    # Model training
    train_loss = train_epoch(model, tra_loader, optimizer, device=device)

    # Model validation
    val_loss = evaluate_epoch(model, val_loader, device=device)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    # Early stopping
    try:
        if val_losses[-1]>=val_losses[-2]:
            early_stop += 1
            if early_stop == patience:
                print("Early stopping! Epoch:", epoch)
                break
        else:
            early_stop = 0
    except:
        early_stop = 0

    if epoch%10 == 0:
        print("epoch:",epoch, "\t training loss:", np.round(train_loss,4),
                            "\t validation loss:", np.round(val_loss,4))

elapsed_time = time.time() - start_time
print(f'Model training took {elapsed_time:.3f} seconds')
```

In the following cell, plot the loss as function of epochs

In [ ]:
```python
# plot the training and validation loss curves
...
```

🕵️ Questions

Based on the loss curves:

- What would you conclude about the overfitting or underfitting capabilities of the model?
- Is this model adequate to be used? Why?

Answers:

📈 Results

For our application, we need the model to be both accurate and fast. Here, we will test those qualities.

Accuracy

Loss

Calculate the loss value for the test dataset

In [ ]:
```python
tst_loader = pyg_DataLoader(tst_dataset_pyg, batch_size=batch_size, shuffle=False)
tst_loss = evaluate_epoch(model, tst_loader, device=device)
num_test_sims = len(tst_dataset_pyg)

print('Test loss: ', tst_loss)
print('Number of test scenarios: ', num_test_sims)
```

Errors in unnormalized variable

Calculate the error in pressure for all the nodes in all the scenarios. This error matrix should be of shape [Scenarios , Nodes].

Remember that the variables were normalized for training purposes. However, the water utility is interested in the value of the output variable in physical units, in this case, pressure in mH2O. In order to do this, unnormalize the output variable knowing that the maximum and minimum pressures used to normalized the output variable were 59.56 mH2O and 0 mH2O, respectively.

In [ ]:
```python
max_pressure = ... #mH2O

estimated_pressures = ...
target_pressures = ...

...

error = target_pressures - estimated_pressures
```

Error in pressure for all scenarios for one node

Plot the error of one node across test scenarios.

In [ ]:
```python
node_ID = 0
error_node = error[:, node_ID]
```

In [ ]:
```python
# Plot the error of a single node across test scenarios.
...
```

🕵️ Questions

Based on the results accross scenarios:

- Is the model satisfactorily fitting?
- If there are there outliers, do they have any pattern?
- Are these errors considerable?

Answers:

Error of all nodes in one scenario

Plot the error of all the nodes of one scenario.

In [ ]:
```python
sce_ID = 10
error_sim = error[sce_ID, :]
```

In [ ]:
```python
#Plot the error for all the nodes in one single scenario.
```

🕵️ Questions

Based on this error analysis:

- How does it compare with the previous error analysis?
- Is the model over- or under- predicting?
- Would you recommend the water utility to use this model?
  - Does your recommendation match with your previous recommendation?

Answers:

Speed

We can calculate the time per scenario that the model takes.

In [ ]:
```python
start_time = time.time()
  #Execute the model here for all the simulations in the test case
total_time = time.time() - start_time

data_driven_exec_time_per_sim = total_time/num_test_sims
print(f'Data-driven model took {data_driven_exec_time_per_sim:.5f} seconds for {num_test_sims} scenarios')
```

Considering that the original model can take up to 0.04 seconds per scenario, we can estimate the potential gain in speed-up. (Speed-up = original_time/Data-driven_model_time)

In [ ]:
```python
original_time_per_sim = 0.04

speed_up = np.round(original_time_per_sim/data_driven_exec_time_per_sim, 2)
print('The data-driven model is', speed_up,'times faster than EPANET per scenario.')
```

🕵️ Questions

Based on the results on speed:

- Which factors or components could make the model faster or slower on its execution?

Answers:

🥈 Transferability

A practical property of graph neural networks is their independence of the domain in which they are trained on. Therefore, they can be (pre-)trained in one network and be used in another case. Let's explore this property by using the already trained models on another water distribution network.

This water network comes from the city of Pescara. This network has 71 nodes (68 junctions and 3 reservoirs) and 196 edges (98 pipes).

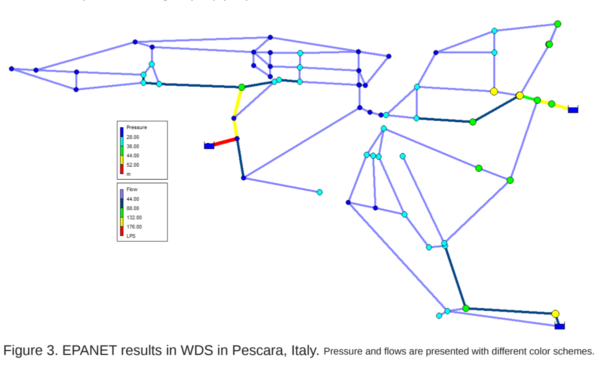

Figure 3. EPANET results in WDS in Pescara, Italy. Pressure and flows are presented with different color schemes.

There is already a prepared scenario example from this network. This example has been already normalized. The "y" values were normalized considering a minimum of 0 mH2O and a maximum of 51.75 mH2O.

Let's load this scenario and use our model to estimate the pressure at all nodes.

In [ ]:
```python
with open("PES_example.p", 'rb') as handle:
    pes_data = pickle.load(handle)
```

Use your model to estimate the pressure for this scenario on this new network.

In [ ]:
```python
predictions = ...
```

Estimate the difference (in meters) between the actual pressure and the model.

In [ ]:
```python
error = ...
```

Plot the error of all the nodes for this scenario.

In [ ]:
```python
# Plot the error for all the nodes in this new water network.
```

🕵️ Questions

Based on the results on transferability:

- How does this error distribution compare against the error distribution for the previous network?
- Which factors do you consider influence the performance of the model when it is transferred?
- How do you consider the model could improve its transfer capability?

Answers: