Please complete the `NotImplemented` parts of the code cells and write your answers in the markdown cells designated for your response to any questions asked. The tag `# AUTOGRADED` (all caps, with a space after `#`) should be at the beginning of each autograded code cell, so make sure that you do not change that. You are also not allowed to import any new package other than the ones already imported. Doing so will prevent the autograder from grading your code.

For the code submission, run the last cell in the notebook to create the submission zip file. If you are working in Colab, make sure to download and then upload a copy of the completed notebook itself to its working directory to be included in the zip file. Finally, submit the zip file to Gradescope.

After you finish the assignment and fill in your code and response where needed (all cells should have been run), save the notebook as a PDF using the `jupyter nbconvert --to pdf HW5.ipynb` command (via a notebook code cell or the command line directly) and submit the PDF to Gradescope under the PDF submission item. If you cannot get this to work locally, you can upload the notebook to Google Colab and create the PDF there. You can find the notebook containing the instruction for this on Canvas.

If you are running the notebook locally, make sure you have created a virtual environment (using `conda` for example) and have the proper packages installed. We are working with `python=3.10` and `torch>=2`.

Files to be included in submission:

- `HW5.ipynb`
- `model_config.yaml`
- `train_config.yaml`
- `state_dict.pt`

In [1]:
```
"""
DO NOT ADD ANY ADDITIONAL IMPORTS IN THE NOTEBOOK.
"""
import os
from typing import Sequence, Dict, Union
from tqdm import tqdm
import numpy as np

import torch
from torch import nn
from torch.nn import functional as F
from torch.utils.data import Dataset

try:
    import torch_geometric as gtorch
except ImportError:
    os.system('pip install torch_geometric -qq')
    os.system('pip install torch-scatter -qq')
```

```
    import torch_geometric as gtorch

import torch_geometric.data as gdata
from torch_geometric import nn as gnn
from torch_geometric.loader import DataLoader as gDataLoader

from HW5_utils import Tracker, print_tensor_info # just in case you need it
from HW5_utils import test_graph_convolution, save_yaml, load_yaml, zip_files, trai

if torch.cuda.is_available():
    Device = 'cuda'
elif torch.backends.mps.is_available():
    Device = 'mps'
else:
    Device = 'cpu'

print(f'Device is {Device}')
```

```
Device is cuda
```

Implement the Graph Convolutional Operator (30)

Your first task is to implement the graph convolution operator that is calculated in the `GCNConv` layer, but **only using** `numpy`. You can see the mathematical definition in the paper and in the online documentation. Basically, you are going to implement the following:

$$X' = D^{-1/2}AD^{-1/2}X\Theta$$

First you should get more familiar with how a graph is defined. In a general graph, edges have directions, and information flows from the source node to the target node. In our code, edges are defined by `edge_index`, which is of the shape `(2, num_edges)`. Each column corresponds to one edge and has two elements: the first (at index `0`) is the source node's index `j` and the second (at index `1`) is the target node's index `i`. This makes nodes `j` a neighbor of node `i`, or in mathematical notation $j \in \mathcal{N}(i)$. In the adjacency matrix, `A[i, j]` should be $e_{j,i}$ (the edge weight) if `j` is a neighbor of `i` and 0 otherwise. You have to create A from `edge_index` and `edge_weights` (without `for` loops).

If `add_self_loops=True`, you have to modify A so there is an edge with weight 1 connecting each node to itself.

$\hat{D}^{-1/2}$ is a diagonal matrix (zero on non-diagonal elements), with the i-th element on its diagonal being $d_i^{-1/2}$ where $d_i = \sum_{j \in \mathcal{N}(i)} e_{j,i}$. You can calculate this matrix from the adjacency matrix A.

REMEMBER: `for` loops make things slow. Therefore, there is a penalty of `-5` for each unnecessary `for` loop. An essential skill you have to learn is to *vectorize* your operations and calculations, which basically means to avoid using `for` loops and instead make use of

parallel computing provided by functions in libraries like `numpy` and `torch`. You can look back at recitation zero to see how you can index tensors or arrays with other tensors or arrays.

You can test your function by comparing it to the output of the actual `GCNConv` layer from `gtorch`. The test function is provided to you, so you can try as much as you want until you get it right.

In [2]:
```python
# AUTOGRADED

# Only numpy allowed
def graph_convolution(
        x: np.ndarray, # shape: (num_nodes, in_channels), dtype: np.float32
        edge_index: np.ndarray, # shape: (2, num_edges), dtype: np.int64
        edge_weights: np.ndarray, # shape: (num_edges,), dtype: np.float32
        theta: np.ndarray, # shape: (in_channels, out_channels), dtype: np.float32
        add_self_loops: bool = True,
        ) -> np.ndarray: # shape: (num_nodes, out_channels), dtype: np.float32

    # get the number of nodes and the number of input channels
    num_nodes, in_channels = x.shape

    # adjacency matrix A
    A = np.zeros((num_nodes, num_nodes), dtype=np.float32)

    # add the edge weights to the adjacency matrix
    np.add.at(A, (edge_index[1], edge_index[0]), edge_weights)

    # add self loops: add 1 to the diagonal of A
    if add_self_loops:
        A[np.arange(num_nodes), np.arange(num_nodes)] += 1

    # compute the degree matrix D
    D = np.sum(A, axis=1)

    # compute the inverse square root of D
    D_inv_sqrt = np.diag(1 / np.sqrt(D))

    # compute the normalized adjacency matrix
    A_norm = D_inv_sqrt @ A @ D_inv_sqrt

    return A_norm @ x @ theta
```

In [3]:
```python
"""
Test your code.
"""
test_graph_convolution(graph_convolution, num_tests=5, show_failed_edge_index=True)
```

```
All tests passed!
```

Implement and train a GNN (70)

Your second task in this assignment is to define and train a model to predict log solubility of molecules in water. You have to define a model and achieve a low enough loss by finding a good model and training it. The dataset class is provided to you. Use it to inspect the data and find out the information you need to define your model.

In [4]:
```
class GraphDataset(Dataset):

    def __init__(
            self,
            data_path: str,
            ):
        super().__init__()
        np_data = np.load(data_path, allow_pickle=True)

        self.samples = []
        for i, (x, edge_index, edge_attr, y) in enumerate(np_data):
            self.samples.append(
                gdata.Data(
                    x = torch.tensor(np.array(x), dtype=torch.float32),
                    edge_index = torch.tensor(np.array(edge_index), dtype=torch.lon
                    edge_attr = torch.tensor(np.array(edge_attr), dtype=torch.float
                    y = torch.tensor(np.array(y).reshape(1, 1), dtype=torch.float32
                )
            )

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        return self.samples[idx]
```

In [5]:
```
"""
Inspect the dataset. You can also create a gDataLoader to inspect the batched data.
"""
my_dataset = GraphDataset('data/train.npy')

print(f"Number of graph data: {len(my_dataset)}")
print("Sample data:", my_dataset[0])
print("Feature shape:", my_dataset[0].x.shape)
```

```
Number of graph data: 904
Sample data: Data(x=[32, 37], edge_index=[2, 68], edge_attr=[68, 6], y=[1, 1])
Feature shape: torch.Size([32, 37])
```

Define your Model (60)

Implement a model to predict the log solubility of a molecule which is represented as a graph. This is a graph regression task, so you need one output per sample graph. Your model's forward method should process batched graphs. Define some graph convolution layers for node-level processing and message passing, then use a global pooling function, and the rest is like normal fully connected networks. Don't forget nonlinear activation between layers.

You should also use `edge_attr` from the input to pass in `edge_weight` for your `gnn` modules' forward pass. However, `edge_attr` represents each edge as a feature vector, but `edge_weight` needs a nonnegative scalar per edge. Therefore, you should define a learnable module for each layer to calculate the `edge_weight` from the `edge_attr`. You have to make sure the shape is right, since `edge_weight` should be of shape `(num_edges,)`. You should also use some activation function to restrict the range of your edge weights and make sure they are nonnegative values. What activation is appropriate here?

Keep your code organized and clean, and remove debugging code and print statements after you are done.

In [6]:
```
# AUTOGRADED

class Model(nn.Module):

    def __init__(self,
                 in_channels,
                 hidden_channels,
                 out_channels,
                 edge_in_channels,
                 num_layers,
                 dropout):
        """
        Args:
            in_channels (int): Dimension of node features.
            hidden_channels (int): Hidden dimension for GCN layers.
            out_channels (int): Dimension of the output (1 for regression).
            edge_in_channels (int): Dimension of edge feature vectors.
            num_layers (int): Number of graph convolution layers.
            dropout (float): Dropout probability.
        """
        super().__init__()
        self.num_layers = num_layers

        # module list for layers
        self.conv_layers = nn.ModuleList()
        self.edge_proj_layers = nn.ModuleList()
        self.bn_layers = nn.ModuleList()

        # helper function: small MLP for edge weight
        def edge_mlp(in_channels, hidden_dim):
            return nn.Sequential(
                nn.Linear(in_channels, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, 1)
            )

        # first layer
        self.conv_layers.append(gnn.GCNConv(in_channels, hidden_channels))
        self.edge_proj_layers.append(nn.Linear(edge_in_channels, 1))
        self.bn_layers.append(nn.BatchNorm1d(hidden_channels))
```

```
        # add the hidden layers
        for _ in range(num_layers - 1):
            self.conv_layers.append(gnn.GCNConv(hidden_channels, hidden_channels))
            self.edge_proj_layers.append(edge_mlp(edge_in_channels, hidden_dim=16))
            self.bn_layers.append(nn.BatchNorm1d(hidden_channels))

        # dropout
        self.dropout = nn.Dropout(dropout)

        # linear layer for regression
        self.lin = nn.Linear(hidden_channels, out_channels)

        # edge activation function
        self.edge_act = nn.Softplus()


    def forward(
            self,
            # Batched graph:
            x: torch.FloatTensor, # shape: (num_nodes, in_channels)
            edge_index: torch.LongTensor, # shape: (2, num_edges)
            edge_attr: torch.FloatTensor, # shape: (num_edges, edge_channels)
            batch: torch.LongTensor, # shape: (num_nodes,)
            ) -> torch.FloatTensor: # shape: (batch_size, 1)
        """
        Args:
            x (torch.FloatTensor): Node features.
            edge_index (torch.LongTensor): Graph edge indices.
            edge_attr (torch.FloatTensor): Edge features.
            batch (torch.LongTensor): Batch vector.
        Returns:
            torch.FloatTensor: Predicted values.
        """
        # iterate over the layers
        for conv_layer, edge_proj_layer, bn in zip(self.conv_layers, self.edge_proj
            # apply the convolution layer
            edge_weight = self.edge_act(edge_proj_layer(edge_attr))

            # apply the convolution layer
            x = conv_layer(x, edge_index, edge_weight)
            x = bn(x)
            x = F.relu(x)
            x = self.dropout(x)

        # global pooling
        x = gnn.global_mean_pool(x, batch)

        # apply the linear layer
        x = self.lin(x)

        return x
```

Find and train a good model (10)

The dataset is small, so the training should be relatively fast. Look for a good model and when you think you have found a good one, submit to Gradescope to see your test loss. Your score for this part is:

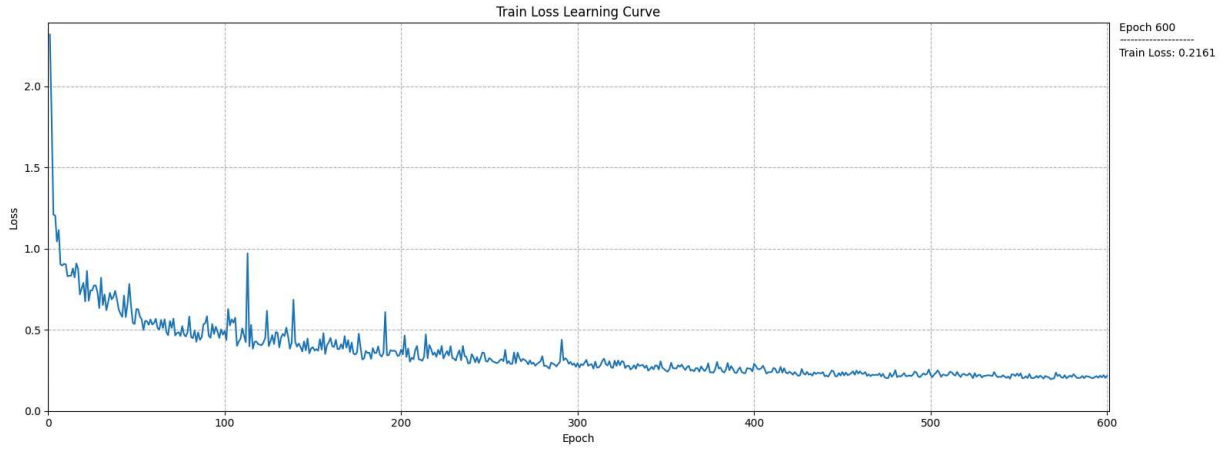
test $\mathrm{MSE} \leq 0.7$: 15 points (5 bonus)

$0.7 <$ test $\mathrm{MSE} \leq 0.9$: 10 points

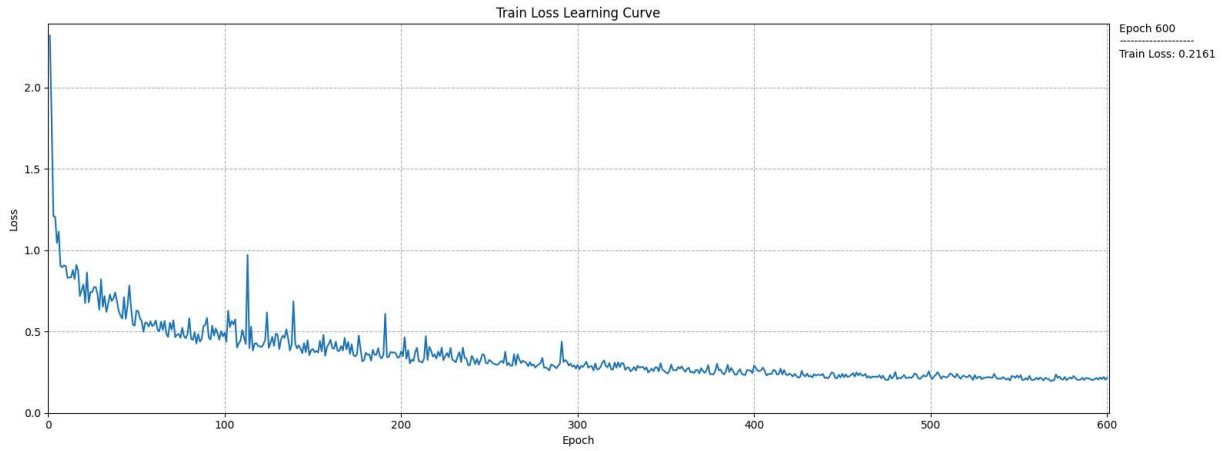
$0.9 <$ test $\mathrm{MSE} \leq 1.1$: 5 points

test $\mathrm{MSE} > 1.1$: 0 points

In [13]:
```
"""
Choose the configuration of your model and training
"""

model_config = dict(
    hidden_channels = 128,
    out_channels = 1,
    num_layers = 4,
    dropout = 0.2,
    )

train_config = dict(
    optimizer_name = 'Adam',
    optimizer_config = dict(lr=1e-3, weight_decay=5e-4),
    # lr_scheduler_name = 'StepLR',
    # lr_scheduler_config = dict(step_size=10, gamma=0.5),
    lr_scheduler_name = 'CosineAnnealingLR',
    lr_scheduler_config = dict(T_max=600, eta_min=1e-6),
    # lr_scheduler_name = 'ReduceLROnPlateau',
    # lr_scheduler_config = dict(factor=0.5, patience=10, threshold=1e-4),
    batch_size = 16,
    n_epochs = 600,
    )

if __name__ == '__main__':
    my_dataset = GraphDataset('data/train.npy')
    in_channels = my_dataset[0].x.shape[1]
    edge_in_channels = my_dataset[0].edge_attr.shape[1]
    model_config['in_channels'] = in_channels
    model_config['edge_in_channels'] = edge_in_channels

    model = Model(**model_config)
    train(
        model = model,
        train_dataset = my_dataset,
        loss_fn = nn.MSELoss(),
        device = Device,
        plot_freq = 10,
        **train_config,
        )
```




```
Epochs: 100%|██████████| 600/600 [09:34<00:00,  1.04epoch/s, Loss: 0.2161]
```



Zip submission files

You can run the following cell to zip the generated files for submission.

If you are on Colab, make sure to download and then upload a completed copy of the notebook to the working directory so the code can detect and include it in the zip file for submission.

In [14]:
```
save_yaml(model_config, 'model_config.yaml')
save_yaml(train_config, 'train_config.yaml')
torch.save(model.cpu().state_dict(), 'state_dict.pt')

# Test if the model can be loaded successfully
loaded_model = Model(**load_yaml('model_config.yaml')).cpu()
loaded_model.load_state_dict(torch.load('state_dict.pt', map_location='cpu'))

files_to_zip = ['HW5.ipynb', 'model_config.yaml', 'train_config.yaml', 'state_dict.
output_zip = 'HW5_submission.zip'
zip_files(output_zip, *files_to_zip)
```

```
C:\Users\RyanWu\AppData\Local\Temp\ipykernel_29588\1520653395.py:7: FutureWarning: Y
ou are using `torch.load` with `weights_only=False` (the current default value), whi
ch uses the default pickle module implicitly. It is possible to construct malicious
pickle data which will execute arbitrary code during unpickling (See https://github.
com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a f
uture release, the default value for `weights_only` will be flipped to `True`. This
limits the functions that could be executed during unpickling. Arbitrary objects wil
l no longer be allowed to be loaded via this mode unless they are explicitly allowli
sted by the user via `torch.serialization.add_safe_globals`. We recommend you start
setting `weights_only=True` for any use case where you don't have full control of th
e loaded file. Please open an issue on GitHub for any issues related to this experim
ental feature.
  loaded_model.load_state_dict(torch.load('state_dict.pt', map_location='cpu'))
```