

TP3_DL

September 6, 2026

0.0.1 Names : Chloé Habasque et Colin Coërchon

1 TP 3 : Graph Neural Networks Architecture

Théo Rudkiewicz, Cyriaque Rousselot

2 TUTORIAL

2.0.1 Install Pytorch Geometric

To handle graph data, we use the library Pytorch Geometric : <https://pytorch-geometric.readthedocs.io/en/latest/>

- If you use *Google Colab*, simply run the following cell to install Pytorch Geometric (**advised**).
- If you plan using your *own environment*, follow the documentation to install Pytorch Geometric : <https://pytorch-geometric.readthedocs.io/en/latest/install/installation.html> and skip the following cell.

```
[1]: from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
[2]: ##### INSTALL TORCH GEOMETRIC #####
# https://pytorch-geometric.readthedocs.io/en/latest/
#####
import torch

def format_pytorch_version(version):
    return version.split("+")[0]

TORCH_version = torch.__version__
TORCH = format_pytorch_version(TORCH_version)

def format_cuda_version(version):
    return "cu" + version.replace(".", "")
```

```
CUDA_version = torch.version.cuda
CUDA = format_cuda_version(CUDA_version)
```

```
!pip install torch-scatter      -f https://pytorch-geometric.com/whl/
  ↳ torch-{TORCH}+{CUDA}.html
!pip install torch-sparse      -f https://pytorch-geometric.com/whl/
  ↳ torch-{TORCH}+{CUDA}.html
!pip install torch-cluster     -f https://pytorch-geometric.com/whl/
  ↳ torch-{TORCH}+{CUDA}.html
!pip install torch-spline-conv -f https://pytorch-geometric.com/whl/
  ↳ torch-{TORCH}+{CUDA}.html
!pip install torch-geometric
```

Looking in links: <https://pytorch-geometric.com/whl/torch-2.5.1+cu124.html>
Collecting torch-scatter

Downloading https://data.pyg.org/whl/torch-2.5.0%2Bcu124/torch_scatter-2.1.2%2Bpt25cu124-cp311-cp311-linux_x86_64.whl (10.8 MB)
10.8/10.8 MB

108.4 MB/s eta 0:00:00

Installing collected packages: torch-scatter

Successfully installed torch-scatter-2.1.2+pt25cu124

Looking in links: <https://pytorch-geometric.com/whl/torch-2.5.1+cu124.html>
Collecting torch-sparse

Downloading https://data.pyg.org/whl/torch-2.5.0%2Bcu124/torch_sparse-0.6.18%2Bpt25cu124-cp311-cp311-linux_x86_64.whl (5.2 MB)
5.2/5.2 MB

45.8 MB/s eta 0:00:00

Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from torch-sparse) (1.14.1)

Requirement already satisfied: numpy<2.3,>=1.23.5 in /usr/local/lib/python3.11/dist-packages (from scipy->torch-sparse) (1.26.4)

Installing collected packages: torch-sparse

Successfully installed torch-sparse-0.6.18+pt25cu124

Looking in links: <https://pytorch-geometric.com/whl/torch-2.5.1+cu124.html>
Collecting torch-cluster

Downloading https://data.pyg.org/whl/torch-2.5.0%2Bcu124/torch_cluster-1.6.3%2Bpt25cu124-cp311-cp311-linux_x86_64.whl (3.4 MB)
3.4/3.4 MB

38.5 MB/s eta 0:00:00

Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from torch-cluster) (1.14.1)

Requirement already satisfied: numpy<2.3,>=1.23.5 in /usr/local/lib/python3.11/dist-packages (from scipy->torch-cluster) (1.26.4)

Installing collected packages: torch-cluster

Successfully installed torch-cluster-1.6.3+pt25cu124

Looking in links: <https://pytorch-geometric.com/whl/torch-2.5.1+cu124.html>

```

Collecting torch-spline-conv
  Downloading https://data.pyg.org/whl/torch-2.5.0%2Bcu124/torch_spline_conv-1.2.2%2Bpt25cu124-cp311-cp311-linux_x86_64.whl (1.0 MB)
    1.0/1.0 MB
18.7 MB/s eta 0:00:00
Installing collected packages: torch-spline-conv
Successfully installed torch-spline-conv-1.2.2+pt25cu124
Collecting torch-geometric
  Downloading torch_geometric-2.6.1-py3-none-any.whl.metadata (63 kB)
    63.1/63.1 kB
3.1 MB/s eta 0:00:00
Requirement already satisfied: aiohttp in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (3.11.13)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (2024.10.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (3.1.6)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (1.26.4)
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (5.9.5)
Requirement already satisfied: pyparsing in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (3.2.1)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages (from torch-geometric) (4.67.1)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (2.5.0)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (1.3.2)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (25.1.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (1.5.0)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (6.1.0)
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (0.3.0)
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.11/dist-packages (from aiohttp->torch-geometric) (1.18.3)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch-geometric) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests->torch-geometric) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->torch-geometric) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in

```

```

/usr/local/lib/python3.11/dist-packages (from requests->torch-geometric) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->torch-geometric)
(2025.1.31)
Downloading torch_geometric-2.6.1-py3-none-any.whl (1.1 MB)
  1.1/1.1 MB
29.7 MB/s eta 0:00:00
Installing collected packages: torch-geometric
Successfully installed torch-geometric-2.6.1

```

2.0.2 Import required packages

Run the following cell to import all required packages. This cell **must not** be modified.

To significantly accelerate your training, it is advised to use GPU. Using Google Colab, you need to activate it :

- Edit -> Notebook Setting -> Hardware accelerator -> GPU

```

[3]: #####
##### PACKAGES #####
#####
import numpy as np
import matplotlib.pyplot as plt

import torch.nn as nn
import torch_geometric.nn as graphnn
from sklearn.metrics import f1_score
from torch_geometric.datasets import PPI
from torch_geometric.loader import DataLoader

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GATConv

```

2.0.3 Dataset

We use the Protein-Protein Interaction (PPI) network dataset which includes: - 20 graphs for training - 2 graphs for validation - 2 graphs for testing

One graph of the PPI dataset has on average 2372 nodes. Each node has: - 50 features : positional gene sets / motif gene / immunological signatures ... - 121 (binary) labels : gene ontology sets (way to classify gene products like proteins).

This problem aims to predict, for a given PPI graph, the correct nodes' labels.

It is a node (multi-label) classification task (trained using supervised learning, with labels to be predicted for each node).

For your curiosity, more detailed information on the dataset and some applications: - <https://cs.stanford.edu/~jure/pubs/pathways-psb18.pdf> - <https://arxiv.org/abs/1707.04638>

To understand how a graph data is implemented in Pytorch Geometric, refer to : https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

```
[4]: ### LOAD DATASETS

BATCH_SIZE = 2

# Train Dataset
train_dataset = PPI(root="", split="train")
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE)
# Val Dataset
val_dataset = PPI(root="", split="val")
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
# Test Dataset
test_dataset = PPI(root="", split="test")
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE)

# Number of features and classes
n_features, n_classes = train_dataset[0].x.shape[1], train_dataset[0].y.shape[1]

print("Number of samples in the train dataset: ", len(train_dataset))
print("Number of samples in the val dataset: ", len(test_dataset))
print("Number of samples in the test dataset: ", len(test_dataset))
print("Output of one sample from the train dataset: ", train_dataset[0])
print("Edge_index :")
print(train_dataset[0].edge_index)
print("Number of features per node: ", n_features)
print("Number of classes per node: ", n_classes)
```

Downloading <https://data.dgl.ai/dataset/ppi.zip>

Extracting ./ppi.zip

Processing...

/usr/local/lib/python3.11/dist-

packages/networkx/readwrite/json_graph/node_link.py:287: FutureWarning:

The default value will be changed to `edges="edges"` in NetworkX 3.6.

To make this warning go away, explicitly set the edges kwarg, e.g.:

```
nx.node_link_graph(data, edges="links") to preserve current behavior, or
nx.node_link_graph(data, edges="edges") for forward compatibility.
warnings.warn(
```

Number of samples in the train dataset: 20

Number of samples in the val dataset: 2

Number of samples in the test dataset: 2

Output of one sample from the train dataset: Data(x=[1767, 50], edge_index=[2,

```

32318], y=[1767, 121])
Edge_index :
tensor([[ 0,    0,    0, ..., 1744, 1745, 1749],
        [372, 1101, 766, ..., 1745, 1744, 1739]])
Number of features per node: 50
Number of classes per node: 121

Done!

```

2.0.4 Define a basic Model

Here we define a very simple Graph Neural Network model which will be used as our baseline. This model consists of three graph convolutional layers (from <https://arxiv.org/pdf/1609.02907.pdf>). The first two layers computes 256 features, followed by an ELU activation function. The last layer is used for (multi-label) classification task, computing 121 features (for each node).

```

[ ]: #####
##### MODEL #####
#####
class BasicGraphModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()

        self.graphconv1 = graphnn.GCNConv(input_size, hidden_size)
        self.graphconv2 = graphnn.GCNConv(hidden_size, hidden_size)
        self.graphconv3 = graphnn.GCNConv(hidden_size, output_size)

        self.elu = nn.ELU()

    def forward(self, x, edge_index):
        x = self.graphconv1(x, edge_index)
        x = self.elu(x)
        x = self.graphconv2(x, edge_index)
        x = self.elu(x)
        x = self.graphconv3(x, edge_index)

        return x

```

Next function is designed to evaluate the performance of the model, computing the F1-Score

```

[5]: #####
##### TEST FUNCTION #####
#####
def evaluate(model, loss_fcn, device, dataloader):
    score_list_batch = []

    model.eval()
    for i, batch in enumerate(dataloader):
        batch = batch.to(device)

```

```

        output = model(batch.x, batch.edge_index)
        loss_test = loss_fcn(output, batch.y)
        predict = np.where(output.detach().cpu().numpy() >= 0, 1, 0)
        score = f1_score(batch.y.cpu().numpy(), predict, average="micro")
        score_list_batch.append(score)

    return np.array(score_list_batch).mean()

```

Next we construct the function to train the model.

```

[6]: # #####
# ##### TRAIN FUNCTION #####
# #####
def train(model, loss_fcn, device, optimizer, max_epochs, train_dataloader,
    ↪ val_dataloader, save_path="/content/drive/MyDrive/Master MVA/Advanced Deep
    ↪ Learning/TP3/model/best_model.pth"):
    epoch_list = []
    scores_list = []

    loss_list = []
    best_score = 0.0
    best_epoch = 0
    best_model_state = None

    # loop over epochs
    for epoch in range(max_epochs):
        model.train()
        losses = []
        # loop over batches
        for i, train_batch in enumerate(train_dataloader):
            optimizer.zero_grad()
            train_batch_device = train_batch.to(device)
            # logits is the output of the model
            logits = model(train_batch_device.x, train_batch_device.edge_index)
            # compute the loss
            loss = loss_fcn(logits, train_batch_device.y)
            # optimizer step
            loss.backward()
            optimizer.step()
            losses.append(loss.item())
        loss_data = np.array(losses).mean()
        loss_list.append(loss_data)
        # print("Epoch {:05d} | Loss: {:.4f}".format(epoch + 1, loss_data))

        if epoch % 5 == 0:
            print("Epoch {:05d} | Loss: {:.4f}".format(epoch + 1, loss_data))
            # evaluate the model on the validation set

```

```

        score = evaluate(model, loss_fcn, device, val_dataloader)
        print("F1-Score: {:.4f}".format(score))
        scores_list.append(score)
        epoch_list.append(epoch)

        # Save the best model
        if score > best_score:
            best_score = score
            best_epoch = epoch
            best_model_state = model.state_dict()

    # Save the best model
    if best_model_state:
        torch.save(best_model_state, save_path)
        print(f"Best model saved at epoch {best_epoch} with F1-score:␣
→{best_score:.4f}")

    # Plot loss curve
    plt.figure(figsize=(8,6))
    plt.plot(range(max_epochs), loss_list, label="Training Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Training Loss Curve")
    plt.legend()
    plt.show()

    return epoch_list, scores_list

```

Let's train this model !

```

[ ]: ### DEVICE GPU OR CPU : will select GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("\nDevice: ", device)

### Max number of epochs
max_epochs = 200

### DEFINE THE MODEL
basic_model = BasicGraphModel(input_size=n_features, hidden_size=256,␣
→output_size=n_classes).to(
    device
)

### DEFINE LOSS FUNCTION
loss_fcn = nn.BCEWithLogitsLoss()

### DEFINE OPTIMIZER

```

```

optimizer = torch.optim.Adam(basic_model.parameters(), lr=0.005)

### TRAIN THE MODEL
epoch_list, basic_model_scores = train(
    basic_model,
    loss_fcn,
    device,
    optimizer,
    max_epochs,
    train_dataloader,
    val_dataloader,
    save_path="/content/drive/MyDrive/Master MVA/Advanced Deep Learning/TP3/
    ↪model/basic_best_model.pth"
)

```

```

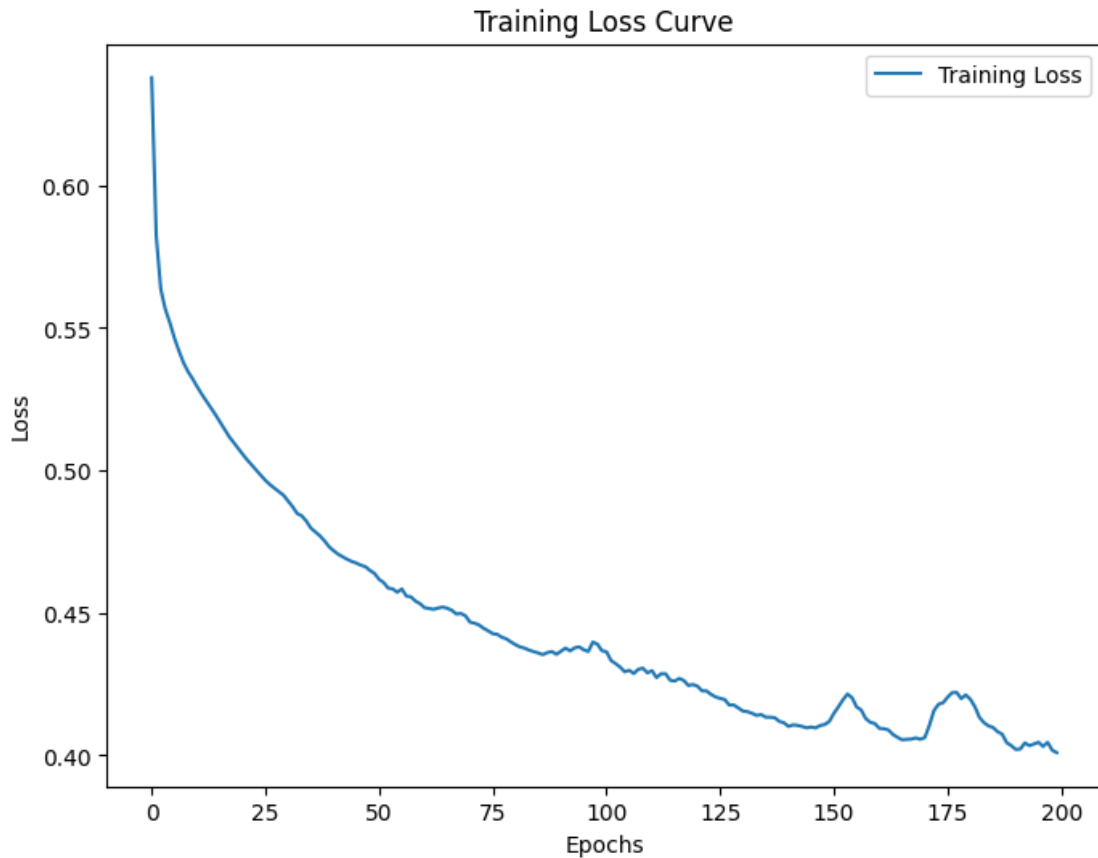
Device:  cuda
Epoch 00001 | Loss: 0.6378
F1-Score: 0.4527
Epoch 00006 | Loss: 0.5465
F1-Score: 0.4826
Epoch 00011 | Loss: 0.5294
F1-Score: 0.5270
Epoch 00016 | Loss: 0.5170
F1-Score: 0.5058
Epoch 00021 | Loss: 0.5057
F1-Score: 0.5383
Epoch 00026 | Loss: 0.4965
F1-Score: 0.5467
Epoch 00031 | Loss: 0.4892
F1-Score: 0.5328
Epoch 00036 | Loss: 0.4798
F1-Score: 0.5593
Epoch 00041 | Loss: 0.4718
F1-Score: 0.5689
Epoch 00046 | Loss: 0.4675
F1-Score: 0.5740
Epoch 00051 | Loss: 0.4618
F1-Score: 0.5804
Epoch 00056 | Loss: 0.4584
F1-Score: 0.5718
Epoch 00061 | Loss: 0.4518
F1-Score: 0.5891
Epoch 00066 | Loss: 0.4516
F1-Score: 0.5924
Epoch 00071 | Loss: 0.4467
F1-Score: 0.5868
Epoch 00076 | Loss: 0.4427

```

F1-Score: 0.5985
Epoch 00081 | Loss: 0.4388
F1-Score: 0.5881
Epoch 00086 | Loss: 0.4359
F1-Score: 0.5933
Epoch 00091 | Loss: 0.4366
F1-Score: 0.5691
Epoch 00096 | Loss: 0.4370
F1-Score: 0.5905
Epoch 00101 | Loss: 0.4364
F1-Score: 0.6226
Epoch 00106 | Loss: 0.4299
F1-Score: 0.5758
Epoch 00111 | Loss: 0.4297
F1-Score: 0.6245
Epoch 00116 | Loss: 0.4261
F1-Score: 0.6209
Epoch 00121 | Loss: 0.4243
F1-Score: 0.5906
Epoch 00126 | Loss: 0.4200
F1-Score: 0.6202
Epoch 00131 | Loss: 0.4156
F1-Score: 0.6272
Epoch 00136 | Loss: 0.4134
F1-Score: 0.6308
Epoch 00141 | Loss: 0.4102
F1-Score: 0.6309
Epoch 00146 | Loss: 0.4099
F1-Score: 0.6228
Epoch 00151 | Loss: 0.4147
F1-Score: 0.6167
Epoch 00156 | Loss: 0.4170
F1-Score: 0.6284
Epoch 00161 | Loss: 0.4095
F1-Score: 0.6103
Epoch 00166 | Loss: 0.4055
F1-Score: 0.6041
Epoch 00171 | Loss: 0.4063
F1-Score: 0.6137
Epoch 00176 | Loss: 0.4204
F1-Score: 0.6537
Epoch 00181 | Loss: 0.4197
F1-Score: 0.6068
Epoch 00186 | Loss: 0.4098
F1-Score: 0.6202
Epoch 00191 | Loss: 0.4021
F1-Score: 0.6488
Epoch 00196 | Loss: 0.4046

F1-Score: 0.6422

Best model saved at epoch 175 with F1-score: 0.6537



Let's evaluate the performance of this basic model

```
[9]: ### PLOT EVOLUTION OF F1-SCORE W.R.T EPOCHS
def plot_f1_score(epoch_list, scores):
    plt.figure(figsize=[10, 5])
    plt.plot(epoch_list, scores)
    plt.title("Evolution of F1-Score w.r.t epochs")
    plt.ylim([0.0, 1.0])
    plt.show()
```

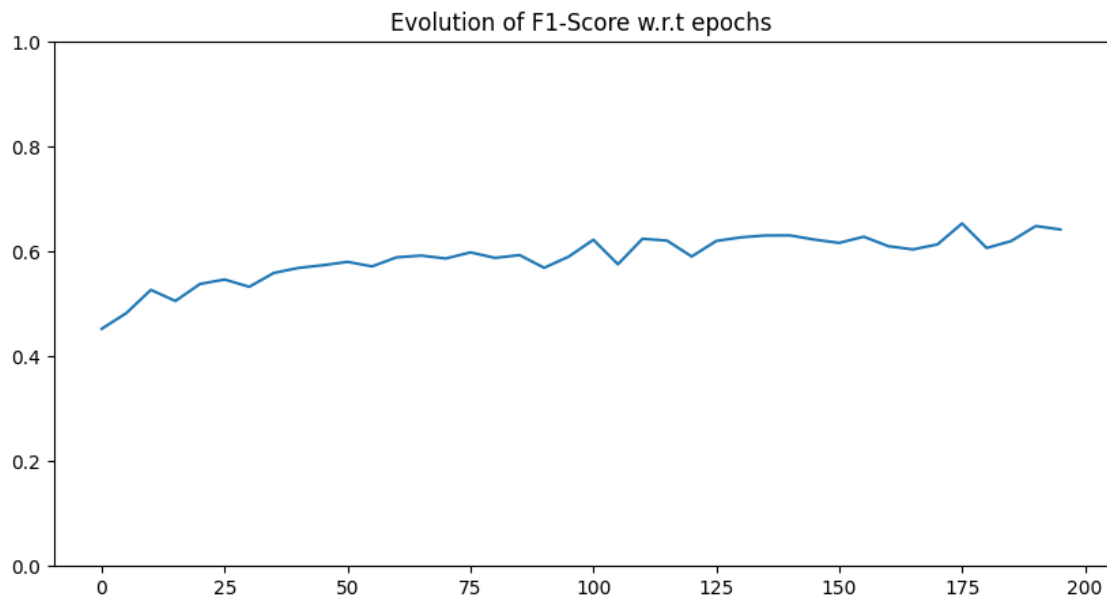
```
[ ]: ### F1-SCORE ON VALID DATASET
score_valid = evaluate(basic_model, loss_fcn, device, val_dataloader)
print("Basic Model : F1-Score on the validation set: {:.4f}".format(score_valid))

### PLOT EVOLUTION OF F1-SCORE W.R.T EPOCHS
def plot_f1_score(epoch_list, scores):
```

```
plt.figure(figsize=[10, 5])
plt.plot(epoch_list, scores)
plt.title("Evolution of F1-Score w.r.t epochs")
plt.ylim([0.0, 1.0])
plt.show()
```

```
plot_f1_score(epoch_list, basic_model_scores)
```

Basic Model : F1-Score on the validation set: 0.6458



3 QUESTIONS

3.1 Grading

You will be graded on 5 questions. You will need to provide at least 4 files : 1. This Notebook 2. `class_model_gnn.py` 3. `model.pth` (the file **must be of size less than 50Mo** but 20Mo should be enough to get a very good model) 4. `conv_as_message_passing.py`

If the function you defined passes all the tests, you will get the full grade. Otherwise we will look at the intermediate questions in the notebook to give you partial credit.

Please provide clear, short and **bold font** answers.

Question 1 : Design, build and train a model with a F1-score higher than 93% on validation set (**HINT** : <https://arxiv.org/pdf/1710.10903.pdf>).

Provide two files : (https://pytorch.org/tutorials/beginner/saving_loading_models.html) - a file `class_model_gnn.py` containing the class inheriting from `torch.nn.Module` architecture of your

final model to load - a `model.pth` file : the model weights

We will test your model on final F1-Score on a test set. You must not use the test set for hyperparameter training.

Intermediate question :

Provide the script for training, and a plot of the training loss.

```
[7]: # Define model ( in your class_model_gnn.py)
class StudentModel(nn.Module):
    def __init__(self, in_feats=50, hidden=256, out_feats=121,
                  num_heads1=4, num_heads2=4, num_heads3=6):
        super(StudentModel, self).__init__()

        # 1ère couche GAT
        self.gat1 = GATConv(
            in_channels=in_feats,
            out_channels=hidden,
            heads=num_heads1,
            concat=True,
            dropout=0.0
        )

        # 2ème couche GAT
        self.gat2 = GATConv(
            in_channels=hidden * num_heads1,
            out_channels=hidden,
            heads=num_heads2,
            concat=True,
            dropout=0.0
        )

        # 3ème couche GAT (classification)
        self.gat3 = GATConv(
            in_channels=hidden * num_heads2,
            out_channels=out_feats,
            heads=num_heads3,
            concat=False,
            dropout=0.0
        )

        # Skip connection
        self.skip_connection1 = nn.Linear(in_feats, hidden * num_heads1)
        self.skip_connection2 = nn.Linear(hidden * num_heads1, hidden *
↪ num_heads2)
        self.skip_connection3 = nn.Linear(hidden * num_heads2, out_feats)
```

```

def forward(self, x, edge_index):
    x = F.elu(self.gat1(x, edge_index)+ self.skip_connection1(x))
    x = F.elu(self.gat2(x, edge_index)+ self.skip_connection2(x))
    x = self.gat3(x, edge_index) + self.skip_connection3(x)
    return x

```

```

[16]: ### DEVICE GPU OR CPU : will select GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("\nDevice: ", device)

### Max number of epochs
max_epochs = 100

### DEFINE THE MODEL
student_model = StudentModel().to(device)

### DEFINE LOSS FUNCTION
loss_fcn = nn.BCEWithLogitsLoss()

### DEFINE OPTIMIZER
optimizer = torch.optim.Adam(student_model.parameters(), lr=0.005)

### TRAIN THE MODEL
epoch_list, student_model_scores = train(
    student_model,
    loss_fcn,
    device,
    optimizer,
    max_epochs,
    train_dataloader,
    val_dataloader,
    save_path="/content/drive/MyDrive/Master MVA/Advanced Deep Learning/TP3/
    ↪model/model.pth"
)

### F1-SCORE ON VALID DATASET
score_valid = evaluate(student_model, loss_fcn, device, val_dataloader)
print("Student Model : F1-Score on the validation set: {:.4f}".
    ↪format(score_valid))

plot_f1_score(epoch_list, student_model_scores)

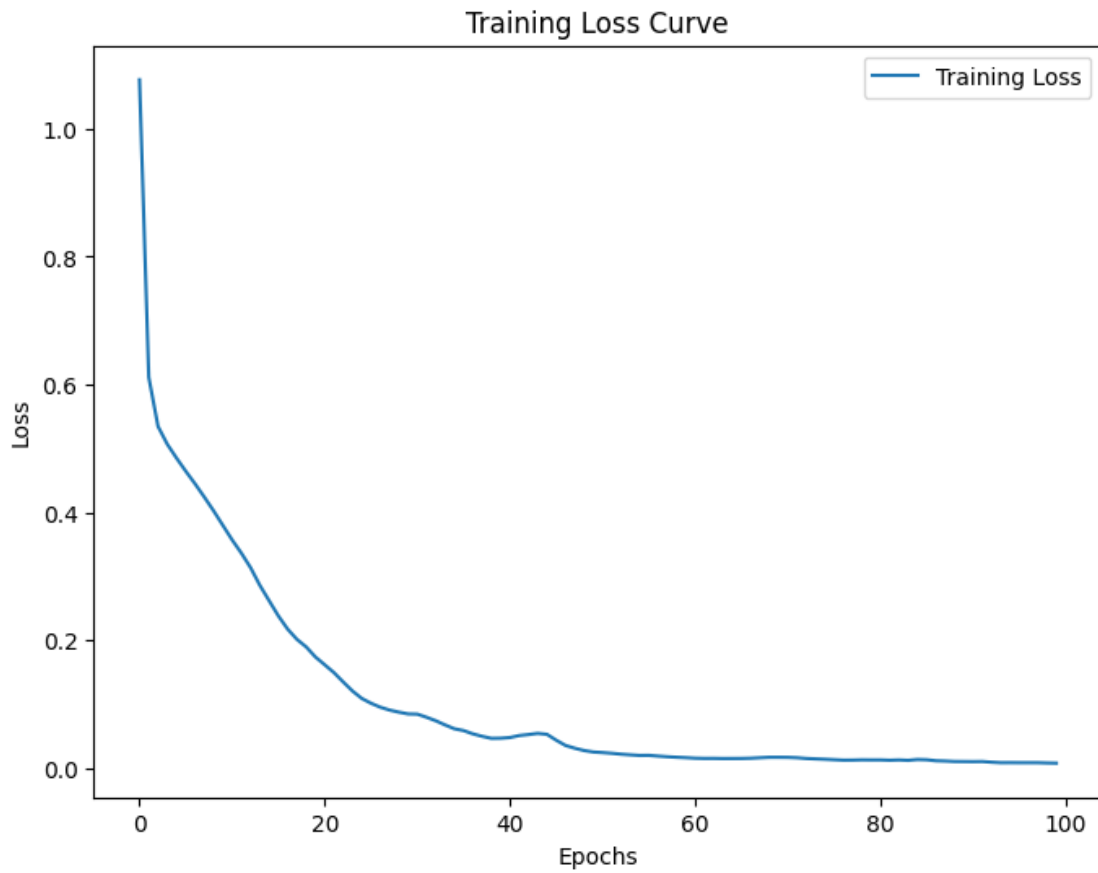
```

```

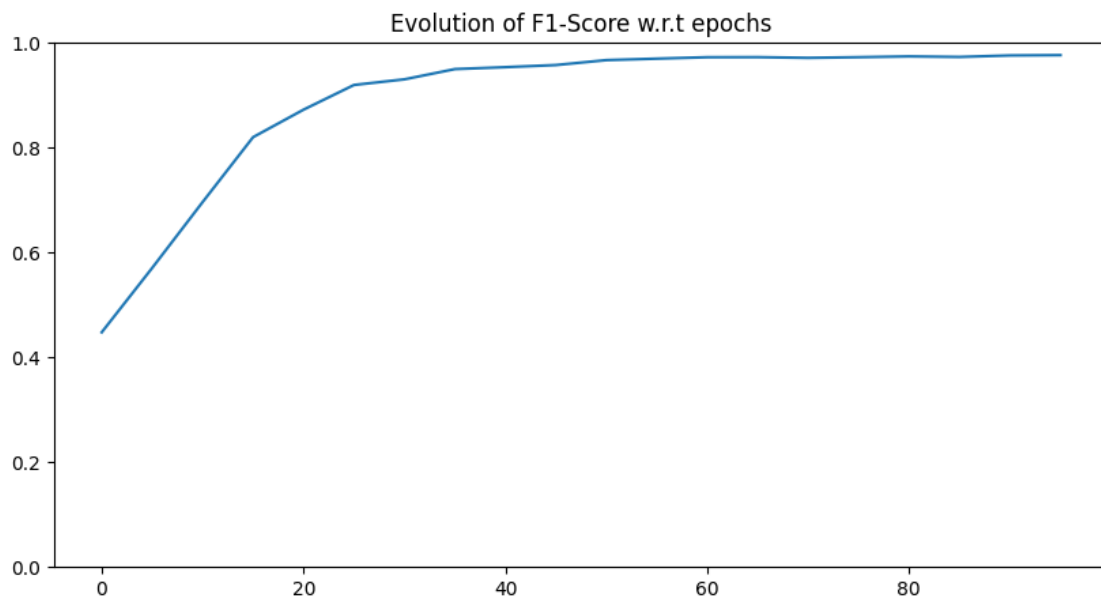
Device:  cuda
Epoch 00001 | Loss: 1.0768
F1-Score: 0.4479
Epoch 00006 | Loss: 0.4645
F1-Score: 0.5707

```

Epoch 00011 | Loss: 0.3572
F1-Score: 0.6964
Epoch 00016 | Loss: 0.2381
F1-Score: 0.8202
Epoch 00021 | Loss: 0.1618
F1-Score: 0.8727
Epoch 00026 | Loss: 0.1016
F1-Score: 0.9198
Epoch 00031 | Loss: 0.0846
F1-Score: 0.9306
Epoch 00036 | Loss: 0.0588
F1-Score: 0.9501
Epoch 00041 | Loss: 0.0479
F1-Score: 0.9538
Epoch 00046 | Loss: 0.0438
F1-Score: 0.9578
Epoch 00051 | Loss: 0.0244
F1-Score: 0.9672
Epoch 00056 | Loss: 0.0201
F1-Score: 0.9700
Epoch 00061 | Loss: 0.0157
F1-Score: 0.9728
Epoch 00066 | Loss: 0.0153
F1-Score: 0.9729
Epoch 00071 | Loss: 0.0169
F1-Score: 0.9715
Epoch 00076 | Loss: 0.0133
F1-Score: 0.9728
Epoch 00081 | Loss: 0.0129
F1-Score: 0.9743
Epoch 00086 | Loss: 0.0132
F1-Score: 0.9733
Epoch 00091 | Loss: 0.0103
F1-Score: 0.9762
Epoch 00096 | Loss: 0.0086
F1-Score: 0.9768
Best model saved at epoch 95 with F1-score: 0.9768



Student Model : F1-Score on the validation set: 0.9762



```
[17]: ### This is the part we will run in the inference to grade your model
      ## Load the model
      model = StudentModel().to(device) # ! Important : No argument
      model.load_state_dict(torch.load("/content/drive/MyDrive/Master MVA/Advanced_
      ↪Deep Learning/TP3/model/model.pth", weights_only=True))
      model.eval()
      print("Model loaded successfully")
```

Model loaded successfully

```
[18]: ### F1-SCORE ON VALID DATASET
      score_valid = evaluate(model, loss_fcn, device, val_dataloader)
      print("Student Model : F1-Score on the validation set: {:.4f}".
      ↪format(score_valid))
```

Student Model : F1-Score on the validation set: 0.9762

3.2 Conv 2D as Message Passing Neural Network

3.3 Introduction

The use of graph is a way to structure data by adding neighborhood information between features. This then allows to do operations on the data that are local to each node and its neighbors. This is the main idea behind Graph Neural Networks (GNNs). [pytorch-geometric](#) is a library compatible with PyTorch that allows to easily implement GNNs. The most general structure is the [MessagePassing](#) class that is then used as a base for more specific GNNs as seen in the course ([Graph Convolutional Networks](#) or [Graph Attention Convolution](#)).

On the other hand, you already know an operation that uses the structure of the data to do local operations: the convolution (<https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html>). One can see the convolution as a specific case of the message passing neural network. The goal of this notebook is to show how to use the [MessagePassing](#) class to implement a convolutional neural network. You will be asked to implement 3 functions. You should give back those three functions in a file named `conv_as_message_passing.py`. These functions will then be automatically tested. So be sure to respect the function signature and the function name.

3.4 Assumptions

To make the implementation easier we will make some assumptions: - the input is a single image (batch size of 1) of size 'C x H x W' - the convolution will be a 3x3 kernel with stride 1 and padding 1.

You may also assume that the Conv2D layer has no bias but it will be slightly penalized in the grading.

Bonus points will be given if you can handle the cases that are not covered by those assumptions.

3.5 Questions

3.5.1 Question 2

Using the formalism used in the [MessagePassing](#) documentation (and on [wikipedia](#) with slightly different notations), explain how theoretically you can simulate a 2D convolution using the MessagePassing formalism. This may include a pre-processing step to transform the image into a graph and then a post-processing step to transform the graph back into an image. (:warning: Those steps should be independent of the parameters of the convolution, but not necessarily from the hyper-parameters.)

$$\mathbf{x}'_i = \gamma_{\Theta} \left(\mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \phi_{\Theta}(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i}) \right),$$

HINT : It is possible to do it with the following γ :

$$\gamma_{\Theta} : x, y \mapsto y$$

To simulate a 2D convolution using MessagePassing formalism, we follow the steps:

1. Pre-processing: image to graph

- Each pixel represents a node
- Each edge represents the connection with neighbors pixels

2. Message Passing (Convolution)

The summation operation aggregates the weighted neighbor features which is like the convolution sum.

$$\bigoplus_{j \in \mathcal{N}(i)} \phi_{\Theta}(x_i, x_j, e_{j,i}) = \sum_{j \in \mathcal{N}(i)} W_{j,i} \cdot x_j$$

Since $\gamma_{\Theta}(x, y) = y$, the updated feature of node i is just:

$$x'_i = \sum_{j \in \mathcal{N}(i)} W_{j,i} \cdot x_j$$

3. Post-processing: graph to image

Finally, we reshape the message passing result to the input image format.

```
[19]: import torch
import torch_geometric
from torch_geometric.data import Data
```

3.5.2 Question 3

Implement the pre-processing function, you can use the following code skeleton (you may change the output type, it is just a strong suggestion):

```

[23]: def image_to_graph(
    image: torch.Tensor, conv2d: torch.nn.Conv2d | None = None
) -> torch_geometric.data.Data:
    """
    Converts an image tensor to a PyTorch Geometric Data object.
    COMPLETE

    Arguments:
    -----
    image : torch.Tensor
        Image tensor of shape (C, H, W).
    conv2d : torch.nn.Conv2d, optional
        Conv2d layer to simulate, by default None
        Is used to determine the size of the receptive field.

    Returns:
    -----
    torch_geometric.data.Data
        Graph representation of the image.
    """
    # Assumptions (remove it for the bonus)
    assert image.dim() == 3, f"Expected 3D tensor, got {image.dim()}D tensor."
    if conv2d is not None:
        assert conv2d.padding[0] == conv2d.padding[1] == 1, "Expected padding of 1 on both sides."
        assert conv2d.kernel_size[0] == conv2d.kernel_size[1] == 3, "Expected kernel size of 3x3."
        assert conv2d.stride[0] == conv2d.stride[1] == 1, "Expected stride of 1."

    C, H, W = image.shape

    x = image.permute(1, 2, 0).contiguous().view(H*W, C)

    edge_index_list = []
    edge_attr_list = []

    for h in range(H):
        for w in range(W):
            i = h * W + w # pixel index
            for dh in [-1, 0, 1]:
                for dw in [-1, 0, 1]:
                    nh = h + dh
                    nw = w + dw

                    if 0 <= nh < H and 0 <= nw < W:
                        j = nh * W + nw # neighbor index
                        edge_index_list.append([j, i])

```

```

        edge_attr_list.append([dh, dw])

    edge_index = torch.tensor(edge_index_list, dtype=torch.long).t().contiguous()
    edge_attr = torch.tensor(edge_attr_list, dtype=torch.long)

    data = Data(x=x, edge_index=edge_index, edge_attr=edge_attr)
    return data

```

3.5.3 Question 4

Implement the post-processing function, you can use the following code skeleton:

```

[24]: def graph_to_image(
    data: torch.Tensor, height: int, width: int, conv2d: torch.nn.Conv2d | None = None
) -> torch.Tensor:
    """
    Converts a graph representation of an image to an image tensor.

    Arguments:
    -----
    data : torch.Tensor
        Graph data representation of the image.
    height : int
        Height of the image.
    width : int
        Width of the image.
    conv2d : torch.nn.Conv2d, optional
        Conv2d layer to simulate, by default None.

    Returns:
    -----
    torch.Tensor
        Image tensor of shape (C, H, W).
    """
    # Assumptions (remove for bonus, if needed)
    assert data.dim() == 2, f"Expected 2D tensor, got {data.dim()}D."

    if conv2d is not None:
        # Under the problem constraints, check these are 3x3 kernel, stride=1,
        padding=1
        assert conv2d.padding[0] == conv2d.padding[1] == 1, "Expected padding=1."
        assert conv2d.kernel_size[0] == conv2d.kernel_size[1] == 3, "Expected
        kernel=3x3."
        assert conv2d.stride[0] == conv2d.stride[1] == 1, "Expected stride=1."

    num_nodes, num_channels = data.shape

```

```

image = data.view(height, width, num_channels).permute(2, 0, 1)

return image

```

Recommended test cases We encourage you to test that you have the property that the pre-processing function followed by the post-processing function is the identity function.

```

[25]: ref_conv = torch.nn.Conv2d(5, 7, kernel_size=3, padding=1, stride=1)
image = torch.randn(5, 10, 11)
g_image = image_to_graph(image, ref_conv)
reconstructed_image = graph_to_image(g_image.x, 10, 11, ref_conv)
assert torch.allclose(image, reconstructed_image)

```

3.5.4 Question 5

Implement the `Conv2dMessagePassing` class that will simulate a 2D convolution using the `MessagePassing` formalism. You should inherit from the `MessagePassing` class and only change the `__init__` and `message` functions (the `forward` function has already been changed for you). You should use the following code skeleton:

```

[26]: from torch_geometric.nn import MessagePassing

class Conv2dMessagePassing(MessagePassing):
    """
    A Message Passing layer that simulates a given Conv2d layer.
    """

    def __init__(self, conv2d: torch.nn.Conv2d):
        # <TO IMPLEMENT>
        # Don't forget to call the parent constructor with the correct arguments
        # super().__init__(<arguments>)
        # </TO IMPLEMENT>
        # 'aggr' to 'add' because convolution sums over neighbors
        super().__init__(aggr='add')
        self.conv2d = conv2d

        out_channels, in_channels, kh, kw = conv2d.weight.shape
        assert kh == 3 and kw == 3, "This example assumes a 3x3 kernel."
        self.register_buffer("W_flat", conv2d.weight.view(out_channels,
→in_channels, -1).permute(2, 0, 1).contiguous())

        if conv2d.bias is not None:
            self.register_parameter("bias", torch.nn.Parameter(conv2d.bias.
→detach().clone()))
        else:
            self.bias = None

```

```

def forward(self, data):
    self.edge_index = data.edge_index

    x, edge_index, edge_attr = data.x, data.edge_index, data.edge_attr
    out = self.propagate(edge_index, x=x, edge_attr=edge_attr)
    return out

def message(self, x_j: torch.Tensor, edge_attr: torch.Tensor) -> torch.
    ↪Tensor:
    """
    Computes the message to be passed for each edge.
    For each edge  $e = (u, v)$  in the graph indexed by  $i$ ,
    the message through the edge  $e$  (ie from node  $u$  to node  $v$ )
    should be returned as the  $i$ -th line of the output tensor.
    (The message is  $\phi(u, v, e)$  in the formalism.)
    To do this you can access the features of the source node
    in  $x_j[i]$  and the attributes of the edge in  $edge\_attr[i]$ .

    Arguments:
    -----
    x_j : torch.Tensor
        The features of the source node for each edge (of size  $E \times x_{in\_channels}$ ).
    ↪edge_attr : torch.Tensor
        The attributes of the edge (of size  $E \times edge\_attr\_dim$ ).

    Returns:
    -----
    torch.Tensor
        The message to be passed for each edge (of size COMPLETE)
    """
    dh = edge_attr[:, 0]
    dw = edge_attr[:, 1]
    offset_idx = (dh + 1) * 3 + (dw + 1)

    W_e = self.W_flat.index_select(0, offset_idx)
    x_j_3d = x_j.unsqueeze(1)
    W_e_t = W_e.permute(0, 2, 1)

    messages = torch.bmm(x_j_3d, W_e_t).squeeze(1)
    return messages

```

3.6 Test example

```
[27]: # from conv_as_message_passing import image_to_graph, graph_to_image, ↪ Conv2dMessagePassing

c = 5
h = 10
w = 11

ref_conv = torch.nn.Conv2d(c, 2, kernel_size=3, padding=1, stride=1, bias=False)
image = torch.randn(c, h, w)
g_image = image_to_graph(image, ref_conv)
ref_conv.weight.data = torch.randn_like(ref_conv.weight.data)

conv_mp = Conv2dMessagePassing(ref_conv)
g_image = conv_mp(g_image)

y_th = ref_conv(image)

ref_conv.weight.data = torch.randn_like(ref_conv.weight.data)
reconstructed_image = graph_to_image(g_image, h, w, ref_conv)

assert torch.allclose(y_th, reconstructed_image, atol=1e-4)
```