**Graph Neural Networks**

**Contents**

[I. Introduction 1](#_Toc150435877)

[II. Graph Neural Network 4](#_Toc150435878)

[1. Graph Representation 4](#_Toc150435879)

[2. Graph Convolutions 5](#_Toc150435880)

[III. Graph Attention 8](#_Toc150435881)

[1. What is Attention? 8](#_Toc150435882)

[2. Graph Attention Introduction 10](#_Toc150435883)

[IV. PyTorch Geometric 15](#_Toc150435884)

[V. Experiment on Graph Structure 17](#_Toc150435885)

[1. Node-level tasks: Semi-supervised node classification 17](#_Toc150435886)

[2. Edge-level tasks: Link prediction 23](#_Toc150435887)

[3. Graph-Level tasks: Graph Classification 24](#_Toc150435888)

[VI. Conclusion 30](#_Toc150435889)

# Introduction

In this tutorial, we will discuss the application of neural networks on graphs. Graph Neural Networks (GNNs) have recently gained increasing popularity in both applications and research, including domains such as social networks, knowledge graphs, recommendation systems and bioinformatics.

While the theory and math behind GNNs might first seem complicated, the implementation of those models is quite simple and helps in understanding the methodology. Therefore, we will discuss the implementation of basic network layers of a GNN, namely graph convolutions, and attention layers. Finally, we will apply a GNN on a node-level, edge-level, and graph-level tasks.

Before, we will start by importing our standard libraries. We will use [PyTorch Lightning](https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09).

*## Standard Library*

import os

import json

import math

import numpy as np

import time

*## Imports for plotting*

import matplotlib.pyplot as plt

%matplotlib inline

from IPython.display import set\_matplotlib\_formats

set\_matplotlib\_formats('svg', 'pdf') *# For export*

from matplotlib.colors import to\_rgb

import matplotlib

matplotlib.recParams['lines.linewidth']=2.0

import seaborn as sns

sns.reset\_orig()

sns.set()

*# Progress bar*

from tqdm.notebook import tqdm

*# PyTorch*

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.utils.data as data

import torch.optim as optim

*# Import Torch Vision*

import torchvision

from torchvision.datasets import CIFAR10

from torchvision import transforms

*# Pytoch Lightning*

try:

    import pytorch\_lightning as p1

except *ModuleNotFoundError*: *# Google Colab does not have PyTorch Lightning*

*!*pip install *--*quite pytorch-ligtning>=1.4

    import pytorch\_lightning as p1

from pytorch\_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

*# Path to the folder where the datasets are/should be downloaded (e.g., CIFAR10)*

DATASET\_PATH = "../data"

*# Path to the folder where the pretrained models are saved*

CHECKPOINT\_PATH = "../saved\_models/tutorial7"

*# Setting the seed*

pl.seed\_verything(42)

*# Ensure that all operations are deterministic on GPU (if used) for reproducibility*

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is\_available() else torch.device("cpu")

print(device)

We also have a few pre-trained models we can download as below.

*# Pretrained model*

import urllib.request

from urllib.error import HTTPError

*# Github URL where saved models are stored for this tutorial*

base\_url = "https://raw.githubusercontent.com/phlippe/saved\_models/main/tutorial7/"

*# Files to download*

pretrained\_files = ["NodeLevelMLP.ckpt", "NodeLevelGNN.ckpt", "GraphLevelGraphConv.ckpt"]

*# Create checkpoint path if it doesn't exist yet*

os.makedirs(CHECKPOINT\_PATH, *exist\_ok*=True)

*# Fore each file, check whether it already exsits. If not, try downloading it.*

for file\_name in pretrained\_files:

    file\_path = os.path.join(CHECKPOINT\_PATH, file\_name)

    if "/" in file\_name:

        os.makedirs(file\_path.rsplit("/", 1)[0], *exist\_ok*=True)

    if not os.path.isfile(file\_path):

        file\_url = base\_url + file\_name

        print(*f*"Downloading {file\_url}...")

        try:

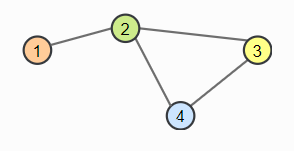
            urllib.request.urlretrieve(file\_url, file\_path)

        except HTTPError as e:

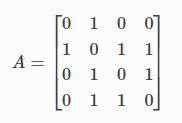
            print("Something went wrong. Please try to download fhe file from the GDrive folder, or contact the author with the full ouput including the follwing error:\n", e)

# Graph Neural Network

## Graph Representation

Before starting the discussion of specific neural network operations on graphs, we should consider how to represent a graph. Mathematically graph G is defined as a tuple of a set of nodes/vertices V, and a set of edges/links E. G = (V, E). Each edge is a pair of 2 vertices and represents a connection between them. For instance, let’s look at the following graph.

The vertices are V = {1, 2, 3, 4} and edges E = {(1, 2), (2, 3), (2, 4), (3, 4)}. Note that for simplicity, we assume the graph to be undirected and hence don’t add mirrored pairs like (2,1). In application, vertices and edges can often have specific attributes, and edges can even be directed. The question is how we could represent this diversity in an efficient way for matrix operations. Usually, for the edges, we decide between two variants: an adjacency matrix, or a list of paired vertex indices.

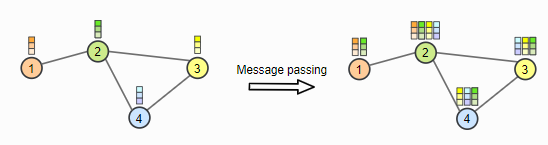
The **adjacency matrix A** is a square matrix whose elements indicate whether pairs of vertices are adjacent, i.e., connected, or not. In the simplest case, **A*ij***is 1 if there is a connection from node *i* to *j*, and the otherwise 0. If we have edge attributes or different categories of edges in a graph, this information can be added to the matrix as well. For an undirected graph, keep in mind that **A** is a symmetric matrix (**A*ij*** = **A*ji***). For the example graph above, we have the following adjacency matrix:

While expression a graph as a list of edges is more efficient in terms of memory and (possibly) computation, using an adjacency matrix is more intuitive and simpler to implement. In our implementations below, we will reply on the adjacency matrix to keep the code simple. However, common libraries use edge lists, which we will discuss later more. Alternatively, we could also use the list of edges to define a sparse adjacency matrix with which we can work as if it was a dense matrix but allows more memory-efficient operations. PyTorch supports this with the sub-package torch.sparse ([documentation](https://pytorch.org/docs/stable/sparse.html)) which is however still in a beta-stage (API might change in the future).

## Graph Convolutions

Graph Convolutional Networks have been introduced by Kipf et al. in 2016 at the University of Amsterdam. He also wrote a great [blog post](https://tkipf.github.io/graph-convolutional-networks/) about this topic, which is recommended if you want to read about GCNs from a different perspective.

GCNs are similar to convolutions in images in the sense that the “filter” parameters are typically shared over all locations in the graph. At the same time, GCNs rely on message passing methods, which means that vertices exchange information with the neighbors, and send “messages’ to each other.

Before looking at math, we can try to visually understand how GCNs work. The first step is that each node creates a feature vector that represents the message it wants to send to all its neighbors. In the second step, the messages are sent to the neighbors, so that a node receives one message per adjacent node. Below we have visualized the 2 steps for our example graph.

If we want to formulate that in more mathematical terms, we need to decide how to combine all the messages a node receives. As the number of messages varies across nodes, we need an operation that works for any number. Hence, the usual way to go is to sum or take the mean. Given the previous features of node H(L), the GCN layer is defined as follows:

where:

* are the weight parameters which we transform the input features into message .
* To the adjacency matrix ***A***, we add the identity matrix so that each node sends its own message also to itself: .
* Finally, to take the average instead of summing, we calculate the matrix which is a diagonal matrix with ***Dii*** denoting the number of neighbors node *i* has.
* represents an arbitrary activation function, and not necessarily the sigmoid (usually a ReLU-based activation function is used in GNNs).

When implementing the GCN layer in PyTorch, we can take advantage of the flexible operations on tensors. Instead of defining a matrix , we can simply divide the summed messages by the number of neighbors afterward. Additionally, we replace the weight matrix with a linear layer, which additionally allows us to add a bias. Written as a PyTorch module, the GCN layer is defined as follows:

*class* GCNLayer(*nn*.*Module*):

*def* \_\_init\_\_(*self*, *c\_in*, *c\_out*):

*super*().\_\_init\_\_()

*self*.projection = nn.Linear(c\_in, c\_out)

*def* forward(*self*, *node\_features*, *adj\_matrix*):

        """\_summary\_

        Args:

            node\_features : Tensor with node features of shape [batch\_size, num\_nodes, c\_in]

            adj\_matrix : Batch of adjacency matrices of the graph. If there is an edge from i to j, adj\_matrix[b,i,j]=1 else 0.

                         Supports directed edges by non-symmetric matrices. Assume to already have added the identity connections.

                         Shpae: [batch\_size, num\_nodes, num\_nodes]

        """

*# Num neighborus = node of incoming edges*

        num\_nieghbours = adj\_matrix.sum(*dim*=1, *keepdims*=True)

        node\_feats = *self*.projection(node\_feats)

        node\_feats = torch.bmm(adj\_matrix, node\_feats)

        node\_feats = node\_feats / num\_nieghbours

        return node\_feats

To further understand the GCN layer, we can apply it to our example graph above. First, let’s specify some node features and the adjacency matrix with added self-connections:

node\_feats = torch.arrange(8, *dtype*=torch.float32).view(1, 4, 2)

adj\_matrix = torch.Tensor([[[1, 1, 0, 0],

                            [1, 1, 1, 1],

                            [0, 1, 1, 1],

                            [0, 1, 1, 1]]])

print("Node features:\n", node\_feats)

print("\nAdjacency matrix:\n", adj\_matrix)

Node features:

tensor([[[0., 1.],

[2., 3.],

[4., 5.],

[6., 7.]]])

Adjacency matrix:

tensor([[[1., 1., 0., 0.],

[1., 1., 1., 1.],

[0., 1., 1., 1.],

[0., 1., 1., 1.]]])

Next, let’s apply a GCN layer to it. For simplicity, we initialize the linear weight matrix as an identify matrix so that the input features are equal to the messages. This makes it easier for us to verify the message passing operation.

layer = GCNLayer(*c\_in*=2, *c\_out*=2)

layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])

layer.projection.bias.data = torch.Tensor([0., 0.])

with torch.no\_grad():

    out\_feats = layer(node\_feats, adj\_matrix)

print("Adjacency matrix", adj\_matrix)

print("Input features", node\_feats)

print("Output features", out\_feats)

Adjacency matrix tensor([[[1., 1., 0., 0.],

[1., 1., 1., 1.],

[0., 1., 1., 1.],

[0., 1., 1., 1.]]])

Input features tensor([[[0., 1.],

[2., 3.],

[4., 5.],

[6., 7.]]])

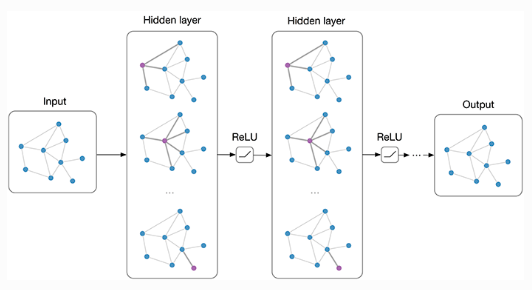
Output features tensor([[[1., 2.],

[3., 4.],

[4., 5.],

[4., 5.]]])

As we can see, the first node’s output values are the average of itself and the second node. Similarly, we can verify all other nodes. However, in a GNN, we would also want to allow feature exchange between nodes beyond its neighbors. This can be achieved by applying multiple GCN layers, which give us the final layout of a GNN. The GNN can be build up by a sequence of GCN layers and non-linearities such as ReLU. For a visualization, see below (figure credit – Thomas Kipf, 2016)



However, one issue we can see from looking at the example above is that the output features for nodes 3 and 4 are the same because they have the same adjacent nodes (including itself). Therefore, GCN layers can make the network forget node-specific information if we just take a mean over all messages. Multiple possible improvements have been proposed. While the simplest option might be using residual connections, the more common approach is to either weight the self-connections higher or define a separate weight matrix for the self-connections. Alternatively, we can re-visit a concept from the last tutorial: attention.

# Graph Attention

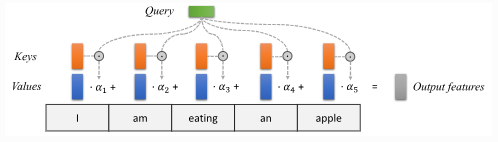
Attention describes a weighted average of multiple elements with the weights dynamically computed based on an input query and element’s keys.

## What is Attention?

The attention mechanism describes a recent new group of layers in neural networks that has attracted a lot of interest in the past few years, especially in sequence tasks. There are a lot of different possible definitions of “attention” in the literature, but the one we will use here is the following: the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and element’s keys. So, what does this exactly mean? The goal is to take an average over the features of multiple elements. However, instead of weighting each element equally, we want to weight them depending on their actual values. In other words, we want to dynamically decide on which inputs we want to “attend” more than others. In particular, an attention mechanism has usually four parts we need to specify:

* **Query** --- The query is a feature vector that describes what we are looking for in the sequence, i.e., what would we maybe want to pay attention to.
* **Keys** --- For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is “offering”, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to base on the query.
* **Values** --- For each input element, we also have a value vector. This feature vector is the one we want to average over.
* **Score function** --- To rate which elements we want to pay attention to, we need to specify a score function *fattn*. The score function takes the query and a key as input and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

The weights of the average are calculated by a SoftMax overall score function output. Hence, we assign those value vectors a higher weight whose corresponding key is most similar to the query. If we try to describe it with pseudo-math, we can write:

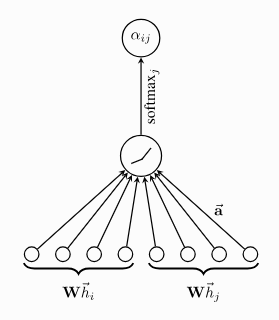
Visually, we can show the attention over a sequence of words as follows:

For every word, we have on key and one value vector. The query is compared to all keys with a score function (in this case the dot product) to determine the weights. The SoftMax is not visualized for simplicity. Finally, the value vectors of all words are averaged using the attention weights.

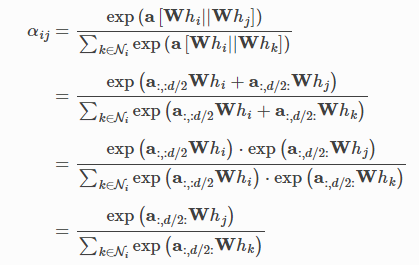
Most attention mechanism differ in terms of what queries they use, how the key and value vectors are defined, and what score function is used. The attention applied inside the Transformer architecture is called self-attention mechanism which is in the Transformer case the scaled dot product attention.

For more about [attention](https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html).

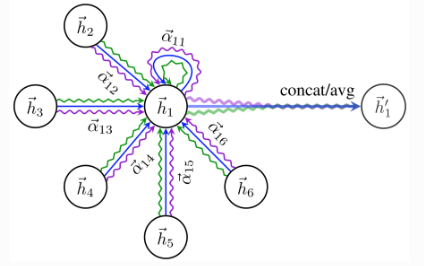
## Graph Attention Introduction

This concept can be similarly applied to graphs, one of such is the Graph Attention Network (called GAT, proposed by Velickovic et al., 2017). Similarly, to the GCN, the graph attention layer creates a message for each node using a linear layer/weight matrix. For attention part, it uses the message from the node itself as a query, and the messages to average as both keys and values (note that this also includes the message to itself). The score function *fattn* is implemented as a one-layer MLP which maps the query and key to a single value. The MLP looks as follows (figure credit – Velickovic et al.):

*hi* and *hj* are the original features from node *i* and *j* respectively and represent the messages of the layer with **W** as weight matrix. **a** is the weight matrix of the MLP, which has the shape [1,2 \* *dmessage*], and the final attention weight from node *i* to *j*. The calculation can be described as follows:

The operator **||** represents the concatenation, and the indices of the neighbors of node *i*. Note that in contrast to usual practice, we apply a non-linearity (here LeakyReLU) before the SoftMax over elements. Although it seems like a minor change at first, it is crucial for the attention to depend on the original input. Specifically, let’s remove the non-linearity for a second, and try to simplify the expression:

We can see that without the non-linearity, the attention term with ***hi*** actually cancels itself out, resulting in the attention being independent of node itself. Hence, we would have the same issue as the GCN of creating the same output features for nodes with the same neighbors. This is shy the LeakyReLU is crucial and addas some dependency on ***hi*** to the attention.

Once we obtain all attention factors, we can calculate the output features for each node by performing the weighted average:

is yet another non-linearity, as in the GCN layer. Visually, we can represent the full message passing in an attention layer as follows (figure credit – Velickovic et al.):

To increase the expressive of the graph attention network, Velickovic et al. proposed to extend it to multiple heads similar to the Multi-Head Attention block in Transformers. This results in ***N*** attention layers being applied in parallel. In the image above, it is visualized as 3 different colors of arrows (green, blue, and purple) that are afterward concatenated. The average is only applied for the very final prediction layer in a network.

After having discussed the graph attention layer in detail, we can implement it below:

*class* GATLayer(*nn*.*Module*):

*def* \_\_init\_\_(*self*, *c\_in*, *c\_out*, *num\_heads*=1, *concat\_heads*=True, *alpha*=0.2):

        """

        Inputs:

            c\_in - Dimensionality of input features

            c\_out - Dimensionality of output features

            num\_heads - Number of heads, i.e. attention mechanisms to apply in parallel. The

                        output features are equally split up over the heads if concat\_heads=True.

            concat\_heads - If True, the output of the different heads is concatenated instead of averaged.

            alpha - Negative slope of the LeakyReLU activation.

        """

*super*().\_\_init\_\_()

*self*.num\_heads = num\_heads

*self*.concat\_heads = concat\_heads

        if *self*.concat\_heads:

            assert c\_out % num\_heads == 0, "Number of output features must be a multiple of the count of heads."

            c\_out = c\_out // num\_heads

*# Sub-modules and parameters needed in the layer*

*self*.projection = nn.Linear(c\_in, c\_out \* num\_heads)

*self*.a = nn.Parameter(torch.Tensor(num\_heads, 2 \* c\_out)) *# One per head*

*self*.leakyrelu = nn.LeakyReLU(alpha)

*# Initialization from the original implementation*

        nn.init.xavier\_uniform\_(*self*.projection.weight.data, *gain*=1.414)

        nn.init.xavier\_uniform\_(*self*.a.data, *gain*=1.414)

*def* forward(*self*, *node\_feats*, *adj\_matrix*, *print\_attn\_probs*=False):

        """

        Inputs:

            node\_feats - Input features of the node. Shape: [batch\_size, c\_in]

            adj\_matrix - Adjacency matrix including self-connections. Shape: [batch\_size, num\_nodes, num\_nodes]

            print\_attn\_probs - If True, the attention weights are printed during the forward pass (for debugging purposes)

        """

        batch\_size, num\_nodes = node\_feats.size(0), node\_feats.size(1)

*# Apply linear layer and sort nodes by head*

        node\_feats = *self*.projection(node\_feats)

        node\_feats = node\_feats.view(batch\_size, num\_nodes, *self*.num\_heads, -1)

*# We need to calculate the attention logits for every edge in the adjacency matrix*

*# Doing this on all possible combinations of nodes is very expensive*

*# => Create a tensor of [W\*h\_i||W\*h\_j] with i and j being the indices of all edges*

        edges = adj\_matrix.nonzero(*as\_tuple*=False) *# Returns indices where the adjacency matrix is not 0 => edges*

        node\_feats\_flat = node\_feats.view(batch\_size \* num\_nodes, *self*.num\_heads, -1)

        edge\_indices\_row = edges[:,0] \* num\_nodes + edges[:,1]

        edge\_indices\_col = edges[:,0] \* num\_nodes + edges[:,2]

        a\_input = torch.cat([

            torch.index\_select(*input*=node\_feats\_flat, *index*=edge\_indices\_row, *dim*=0),

            torch.index\_select(*input*=node\_feats\_flat, *index*=edge\_indices\_col, *dim*=0)

        ], *dim*=-1) *# Index select returns a tensor with node\_feats\_flat being indexed at the desired positions along dim=0*

*# Calculate attention MLP output (independent for each head)*

        attn\_logits = torch.einsum('bhc,hc->bh', a\_input, *self*.a)

        attn\_logits = *self*.leakyrelu(attn\_logits)

*# Map list of attention values back into a matrix*

        attn\_matrix = attn\_logits.new\_zeros(adj\_matrix.shape+(*self*.num\_heads,)).fill\_(-9e15)

        attn\_matrix[adj\_matrix[...,None].repeat(1,1,1,*self*.num\_heads) == 1] = attn\_logits.reshape(-1)

*# Weighted average of attention*

        attn\_probs = F.softmax(attn\_matrix, *dim*=2)

        if print\_attn\_probs:

            print("Attention probs\n", attn\_probs.permute(0, 3, 1, 2))

        node\_feats = torch.einsum('bijh,bjhc->bihc', attn\_probs, node\_feats)

*# If heads should be concatenated, we can do this by reshaping. Otherwise, take mean*

        if *self*.concat\_heads:

            node\_feats = node\_feats.reshape(batch\_size, num\_nodes, -1)

        else:

            node\_feats = node\_feats.mean(*dim*=2)

        return node\_feats

Again, we can apply the graph attention layer on our example graph above to understand the dynamics better. As before, the input layer is initialized as an identity matrix, but we set a to be a vector of arbitrary numbers to obtain different attention values. We use 2 heads to show the parallel, independent attention mechanisms working in the layer.

layer = GATLayer(2, 2, *num\_heads*=2)

layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])

layer.projection.bias.data = torch.Tensor([0., 0.])

layer.a.data = torch.Tensor([[-0.2, 0.3], [0.1, -0.1]])

with torch.no\_grad():

    out\_feats = layer(node\_feats, adj\_matrix, *print\_attn\_probs*=True)

print("Adjacency matrix", adj\_matrix)

print("Input features", node\_feats)

print("Output features", out\_feats)

Attention probs

tensor([[[[0.3543, 0.6457, 0.0000, 0.0000],

[0.1096, 0.1450, 0.2642, 0.4813],

[0.0000, 0.1858, 0.2885, 0.5257],

[0.0000, 0.2391, 0.2696, 0.4913]],

[[0.5100, 0.4900, 0.0000, 0.0000],

[0.2975, 0.2436, 0.2340, 0.2249],

[0.0000, 0.3838, 0.3142, 0.3019],

[0.0000, 0.4018, 0.3289, 0.2693]]]])

Adjacency matrix tensor([[[1., 1., 0., 0.],

[1., 1., 1., 1.],

[0., 1., 1., 1.],

[0., 1., 1., 1.]]])

Input features tensor([[[0., 1.],

[2., 3.],

[4., 5.],

[6., 7.]]])

Output features tensor([[[1.2913, 1.9800],

[4.2344, 3.7725],

[4.6798, 4.8362],

[4.5043, 4.7351]]])

We recommend that you try to calculate the attention matrix at least for one head and one node for yourself. The entries are 0 where there does not exist an edge between *i* and *j*. For the others, we see diverse set of attention probabilities. Moreover, the output features of node 3 and 4 are now different although they have the same neighbors.

# PyTorch Geometric

We had mentioned before that implementing graph networks with adjacency matrix is simple and straightforward but can be computationally expensive for large graphs. Many real-world graphs can reach over 200k nodes, for which adjacency matrix-based implementations fail. There are a lot of optimizations possible when implementing GNNs, and luckily, there exist packages that provide such layers. The most popular packages for PyTorch are [PyTorch Geometric](https://pytorch-geometric.readthedocs.io/en/latest/) and the [Deep Graph Library](https://www.dgl.ai/) (the latter being actually framework agnostic). Which one to use depends on the project you are planning to do and personal taste.

We will look at PyTorch Geometric as part of the PyTorch family. Similar to PyTorch Lightning, PyTorch Geometric is not installed by default on GoogleColab (and actually also not in our d12021 environment due to many dependencies that would be unnecessary for the practical). Hence, let’s import and/or install it below:

*# torch geometric*

try:

    import torch\_geometric

except *ModuleNotFoundError*:

*# Installing torch geometric packages with specific CUDA+PyTorch version.*

*# See https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html for details*

    TORCH = torch.\_\_version\_\_.split('+')[0]

    CUDA = 'cu' + torch.version.cuda.replace('.','')

*!*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

    import torch\_geometric

import torch\_geometric.nn as geom\_nn

import torch\_geometric.data as geom\_data

PyTorch Geometric provides us a set of common graph layers, including the GCN and GAT layer we implement above. Additionally, similar to PyTorch’s torchvision, it provides common graph datasets and transformations on those to simplify training. Compared to our implementation above, PyTorch Geometric uses a list of index pairs to represent the edges. The details of this library will be explored further in our experiments.

In our tasks below, we want to allow us to pick from a multitude of graph layers. Thus, we define again below a dictionary to access those using a string:

gnn\_layer\_by\_name = {

    "GCN": geom\_nn.GCNConv,

    "GAT": geom\_nn.GATConv,

    "GraphConv": geom\_nn.GraphConv

}

Additionally, to GCN and GAT, we added the layer geom\_nn.GraphConv ([documentation](https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.GraphConv)). GraphConv is a GCN with a separate weight matrix for self-connections. Mathematically, this would be:

In this formula, the neighbor’s messages are added instead of averaged. However, PyTorch Geometric provides the argument aggr to switch between summing, averaging, and max pooling.

# Experiment on Graph Structure

Tasks on graph-structured data can be grouped into 3 groups: node-level, edge-level and graph-level. The different levels describe on which level we want to perform classification/regression. We will discuss all 3 types in more detail below.

## Node-level tasks: Semi-supervised node classification

Node-level tasks have the goal to classify nodes in a graph. Usually, we have given a single, large graph with > 1000 nodes of which a certain number of nodes are labeled. We learn to classify those labeled examples during training and try to generalize to the unlabeled nodes.

A popular example that we will use in this tutorial is the Cora dataset, a citation network among papers. The Cora consists of 2708 scientific publications with links between each other representing the citation of one paper by another. The task is to classify each publication into one of seven classes. Each publication is represented by a bag-of-words vector. This means that we have a vector of 1433 elements for each publication, where a 1 at feature *i* indicates that the *i*-th word of a pre-defined dictionary is in the article. Binary bag-of-words representations are commonly used when we need very simple encodings, and already have an intuition of what words to expect in a network. There exist much better approaches, but we will leave this to the NLP courses to discuss.

We will load the dataset below:

cora\_dataset = torch\_geometric.datasets.Planetoid(*root*=DATASET\_PATH, *name*="Cora")

Let’s look at how PyTorch Geometric represents the graph data. Note that although we have a single graph, PyTorch Geometric returns a dataset for compatibility to other datasets.

cora\_dataset[0]

Data(edge\_index=[2, 10556], test\_mask=[2708], train\_mask=[2708], val\_mask=[2708], x=[2708, 1433], y=[2708])

The graph is represented by a Data object (documentation) which we can access as a standard Python namespace. The edge index tensor is the list of edges in the graph and contains the mirrored version of each edge for undirected graphs. The train\_mask, val\_mask, and test\_mask are Boolean masks that indicate which nodes we should use for training, validation, and testing. The x tensor is the feature tensor of our 2708 publications, and y the labels for all nodes.

After having seen the data, we can implement a simple graph neural network. The GNN applies a sequence of graph layers (GCN, GAT, or GraphConv), ReLU as activation function, and dropout for regularization. See below for specific implementation.

*class* GNNModel(*nn*.*Module*):

*def* \_\_init\_\_(*self*, *c\_in*, *c\_hidden*, *c\_out*, *num\_layers*=2, *layer\_name*="GCN", *dp\_rate*=0.1, \*\**kwargs*):

        """

        Inputs:

            c\_in - Dimension of input features

            c\_hidden - Dimension of hidden features

            c\_out - Dimension of the output features. Usually number of classes in classification

            num\_layers - Number of "hidden" graph layers

            layer\_name - String of the graph layer to use

            dp\_rate - Dropout rate to apply throughout the network

            kwargs - Additional arguments for the graph layer (e.g. number of heads for GAT)

        """

*super*().\_\_init\_\_(1)

        gnn\_layer = gnn\_layer\_by\_name[layer\_name]

        layers = []

        in\_channels, out\_channels = c\_in, c\_hidden

        for l\_idx in range(num\_layers-1):

            layers += [

                gnn\_layer(*in\_channels*=in\_channels,

*out\_channels*=out\_channels,

                          \*\*kwargs),

                nn.ReLU(*inplace*=True),

                nn.Dropout(dp\_rate)

            ]

            in\_channels = c\_hidden

        layers += [gnn\_layer(*in\_channels*=in\_channels,

*out\_channels*=c\_out,

                             \*\*kwargs)]

*self*.layers = nn.ModuleList(layers)

*def* forward(*self*, *x*, *edge\_index*):

        """

        Inputs:

            x - Input features per node

            edge\_index - List of vertex index pairs representing the edges in the graph (PyTorch geometric notation)

        """

        for l in *self*.layers:

*# For graph layers, we need to add the "edge\_index" tensor as additional input*

*# All PyTorch Geometric graph layer inherit the class "MessagePassing", hence*

*# we can simply check the class type.*

            if isinstance(1, geom\_nn.MessagePassing):

                x = l(x, edge\_index)

            else:

                x = l(x)

        return x

Good practice in node-level tasks is to create an MLP baseline that is applied to each node independently. This way we can verify whether adding the graph information to the model indeed improves the prediction, or not. It might also be that the features per node are already expressive enough to clearly point towards a specific class. To check this, we implement a simple MLP below:

*class* MLPModel(*nn*.*Module*):

*def* \_\_init\_\_(*self*, *c\_in*, *c\_hidden*, *c\_out*, *num\_layers*=2, *dp\_rate*=0.1):

        """

        Inputs:

            c\_in - Dimension of input features

            c\_hidden - Dimension of hidden features

            c\_out - Dimension of the output features. Usually number of classes in classification

            num\_layers - Number of hidden layers

            dp\_rate - Dropout rate to apply throughout the network

        """

*super*().\_\_init\_\_()

        layers = []

        in\_channels, out\_channels = c\_in, c\_hidden

        for l\_idx in range(num\_layers-1):

            layers += [

                nn.Linear(in\_channels, out\_channels),

                nn.ReLU(*inplace*=True),

                nn.Dropout(dp\_rate)

            ]

            in\_channels = c\_hidden

        layers += [nn.Linear(in\_channels, c\_out)]

*self*.layers = nn.Sequential(\*layers)

*def* forward(*self*, *x*, \**args*, \*\**kwargs*):

        """

        Inputs:

            x - Input features per node

        """

        return *self*.layers(x)

Finally, we can merge the models into a PyTorch Lightning module which handles the training, validation, and testing for us.

*class* NodeLevelGNN(*pl*.*LightningModule*):

*def* \_\_init\_\_(*self*, *model\_name*, \*\**model\_kwargs*):

*super*().\_\_init\_\_()

*# Saving hyperparameters*

*self*.save\_hyperparameters()

        if model\_name == "MLP":

*self*.model = MLPModel(\*\*model\_kwargs)

        else:

*self*.model = GNNModel(\*\*model\_kwargs)

*self*.loss\_module = nn.CrossEntropyLoss()

*def* forward(*self*, *data*, *mode*="train"):

        x, edge\_index = data.x, data.edge\_index

        x = *self*.model(x, edge\_index)

*# Only calculate the loss on the nodes corresponding to the mask*

        if mode == "train":

            mask = data.train\_mask

        elif mode == "val":

            mask = data.val\_mask

        elif mode == "test":

            mask = data.test\_mask

        else:

            assert False, *f*"Unknown forward mode: {mode}"

        loss = *self*.loss\_module(x[mask], data.y[mask])

        acc = (x[mask].argmax(*dim*=-1) == data.y[mask]).sum().float() / mask.sum()

        return loss, acc

*def* configure\_optimizers(*self*):

*# We use SGD here, but Adam works as well*

        optimizer = optim.SGD(*self*.parameters(), *lr*=0.1, *momentum*=0.9, *weight\_decay*=2e-3)

        return optimizer

*def* training\_step(*self*, *batch*, *batch\_idx*):

        loss, acc = *self*.forward(batch, *mode*="train")

*self*.log('train\_loss', loss)

*self*.log('train\_acc', acc)

        return loss

*def* validation\_step(*self*, *batch*, *batch\_idx*):

        \_, acc = *self*.forward(batch, *mode*="val")

*self*.log('val\_acc', acc)

*def* test\_step(*self*, *batch*, *batch\_idx*):

        \_, acc = *self*.forward(batch, *mode*="test")

*self*.log('test\_acc', acc)

Additionally, to the Lightning module, we define a training function below. As we have a single graph, we use a batch size of 1 for the data loader and share the same data loader for the train, validation, and test set (the mask is picked inside the Lightning module). Besides, we set the argument enable\_progress\_bar to False as it usually shows the progress per epoch, but an epoch only consists of a single step.

*def* train\_node\_classifier(*model\_name*, *dataset*, \*\**model\_kwargs*):

    pl.seed\_everything(42)

    node\_data\_loader = geom\_data.DataLoader(dataset, *batch\_size*=1)

*# Create a PyTorch Lightning trainer with the generation callback*

    root\_dir = os.path.join(CHECKPOINT\_PATH, "NodeLevel" + model\_name)

    os.makedirs(root\_dir, *exist\_ok*=True)

    trainer = pl.Trainer(*default\_root\_dir*=root\_dir,

*callbacks*=[ModelCheckpoint(*save\_weights\_only*=True, *mode*="max", *monitor*="val\_acc")],

*accelerator*="gpu" if *str*(device).startswith("cuda") else "cpu",

*devices*=1,

*max\_epochs*=200,

*enable\_progress\_bar*=False) *# False because epoch size is 1*

    trainer.logger.\_default\_hp\_metric = None *# Optional logging argument that we don't need*

*# Check whether pretrained model exists. If yes, load it and skip training*

    pretrained\_filename = os.path.join(CHECKPOINT\_PATH, *f*"NodeLevel{model\_name}.ckpt")

    if os.path.isfile(pretrained\_filename):

        print("Found pretrained model, loading...")

        model = NodeLevelGNN.load\_from\_checkpoint(pretrained\_filename)

    else:

        pl.seed\_everything()

        model = NodeLevelGNN(*model\_name*=model\_name, *c\_in*=dataset.num\_node\_features, *c\_out*=dataset.num\_classes, \*\*model\_kwargs)

        trainer.fit(model, node\_data\_loader, node\_data\_loader)

        model = NodeLevelGNN.load\_from\_checkpoint(trainer.checkpoint\_callback.best\_model\_path)

*# Test best model on the test set*

    test\_result = trainer.test(model, node\_data\_loader, *verbose*=False)

    batch = next(iter(node\_data\_loader))

    batch = batch.to(model.device)

    \_, train\_acc = model.forward(batch, *mode*="train")

    \_, val\_acc = model.forward(batch, *mode*="val")

    result = {"train": train\_acc,

              "val": val\_acc,

              "test": test\_result[0]['test\_acc']}

    return model, result

Finally, we can train our models. First let’s train the simple MLP:

*def* print\_results(*result\_dict*):

    if "train" in result\_dict:

        print(*f*"Train accuracy: {(100.0\*result\_dict['train'])*:4.2f*}%")

    if "val" in result\_dict:

        print(*f*"Val accuracy:   {(100.0\*result\_dict['val'])*:4.2f*}%")

    print(*f*"Test accuracy:  {(100.0\*result\_dict['test'])*:4.2f*}%")

node\_mlp\_model, node\_mlp\_result = train\_node\_classifier(*model\_name*="MLP",

*dataset*=cora\_dataset,

*c\_hidden*=16,

*num\_layers*=2,

*dp\_rate*=0.1)

print\_results(node\_mlp\_result)

Found pretrained model, loading...

Train accuracy: 96.43%

Val accuracy: 52.60%

Test accuracy: 60.60%

Although the MLP can be overfit on the training dataset because of high-dimensional input features, it does not perform too well on the test set. Let’s see if we beat this score with our graph networks.

node\_gnn\_model, node\_gnn\_result = train\_node\_classifier(*model\_name*="GNN",

*layer\_name*="GCN",

*dataset*=cora\_dataset,

*c\_hidden*=16,

*num\_layers*=2,

*dp\_rate*=0.1)

print\_results(node\_gnn\_result)

Found pretrained model, loading...

Train accuracy: 100.00%

Val accuracy: 77.20%

Test accuracy: 82.40%

As we would have hoped for, the GNN model outperforms the MLP by quite a margin. This shows that using the graph information indeed improves our predictions and lets us generalize better.

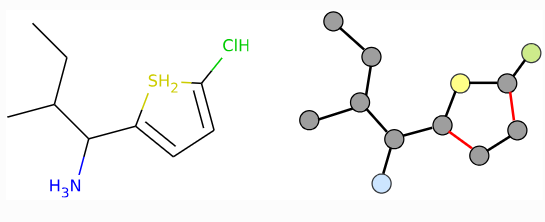
The hyperparameters in the model have been chosen to create a relatively small network. This is because the first layer with an input dimension of 1433 can be relatively expensive to perform for large graphs. This is why such GNNs either have a small hidden size or use a special batching strategy where we sampled a connected subgraph of the big, original graph.

## Edge-level tasks: Link prediction

In some applications, we might have to predict on an edge-level instead of node-level. The most common edge-level task in GNN is link prediction. Link prediction means that given a graph, we want to predict whether there will be/should be an edge between 2 nodes or not. For example, in social networks, this is used by Facebook and co to propose new friends to you. Again, graph level information can be crucial to perform this task. The output prediction is usually done by performing a similar metric on the pair of node features, which should be 1 if there should be a link, and otherwise close to 0. To keep the tutorial short, we will not implement this task ourselves. Nevertheless, there are many good resources out there if you are interested in looking closer at this task. Tutorials and papers for this topic include:

* [PyTorch Geometric example](https://github.com/rusty1s/pytorch_geometric/blob/master/examples/link_pred.py)
* [Graph Neural Networks: A Review of Methods and Applications](https://arxiv.org/pdf/1812.08434.pdf), Zhou et al. 2019
* [Link Prediction Based on Graph Neural Networks](https://papers.nips.cc/paper/2018/file/53f0d7c537d99b3824f0f99d62ea2428-Paper.pdf), Zhang and Chen, 2018.

## Graph-Level tasks: Graph Classification

Finally, in this part of the tutorial, we will have a closer look at how to apply GNNs to the task of graph classification. The goal is to classify an entire graph instead of single nodes or edges. Therefore, we are also given a dataset of multiple graphs that we need to classify based on some structural graph properties. The most common task for graph classification is molecular property prediction, in which molecules are represented as graphs. Each atom is linked to a node, and edges in the graph are the bonds between atoms. For example, look at the figure below:

On the left, we have an arbitrary, small molecule with different atoms, whereas the right part of the image shows the graph representation. The atom types are abstracted as node features (e.g., a one-hot vector), and the different bond types are used as edges features. For simplicity, we will neglect the edge attributes in this tutorial, but you can include by using methods like the [Relational Graph Convolution](https://arxiv.org/abs/1703.06103) that uses a different weight matrix for each edge type.

The dataset we will use below is called the MUTAG dataset. It is a common small benchmark for graph classification algorithms and contains 188 graphs with 18 nodes and 20 edges on average for each graph. The graph node has 7 different labels/atom types, and the binary graph labels represent “their mutagenic effect on a specific gram-negative bacterium.” (the specific meaning of the labels is not too important here). The dataset is part of a large collection on different graph classification datasets, known as the [TUDatasets](https://chrsmrrs.github.io/datasets/), which is directly accessible via torch\_geometric.datasets.TUDataset ([documentation](https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html#torch_geometric.datasets.TUDataset)) in PyTorch Geometric. We can load the dataset below:

tu\_dataset = torch.geometric.datasets.TUDataset(*root*=DATASET\_PATH, *name*="MUTAG")

Let’s look at some statistics for the dataset:

print("Data object:", tu\_dataset.data)

print("Length:", len(tu\_dataset))

print(*f*"Average label: {tu\_dataset.data.y.float().mean().item()*:4.2f*}")

Data object: Data(edge\_attr=[7442, 4], edge\_index=[2, 7442], x=[3371, 7], y=[188])

Length: 188

Average label: 0.66

The first line shows how the dataset stores different graphs. The nodes, edges, and labels of each graph are concatenated to one tensor, and the dataset stores the indices where to split the tensor correspondingly. The length of the dataset is the number of graphs we have, and the “average label” denotes the number of graphs we have, and the “average label” denotes the percentage of the graph with label 1. As long as the percentage is in the range of 0.5, we have a relatively balanced dataset. It happens quite often that the graph datasets are very imbalanced, hence checking the class balance is always a good thing to do.

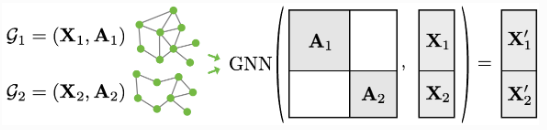
Next, we will split our dataset into training and test part. Note that we do not use a validation set this time because of the small size of the dataset. Therefore, our model might overfit slightly on the validation set due to the noise of the evaluation, but still we get an estimate of the performance on untrained data.

torch.manual\_seed(42)

tu\_dataset.shuffle()

train\_dataset = tu\_dataset[:150]

test\_dataset = tu\_dataset[150:]

When using a data loader, we encounter a problem with batching ***N*** graphs. Each graph in the batch can have a different number of nodes and edges, and hence we would require a lot of padding to obtain a single tensor. Torch geometric uses a different, more efficient approach: we can view the ***N*** graphs in a batch as a single large graph with concatenated node and edge list. As there is no edge between the ***N*** graphs, running GNN layers on the graph gives us the same output as running the GNN on each graph separately. Visually, this batching strategy is visualized below (figure credit – PyTorch Geometric team, [tutorial](https://colab.research.google.com/drive/1I8a0DfQ3fI7Njc62__mVXUlcAleUclnb?usp=sharing#scrollTo=2owRWKcuoALo) here).

The adjacency matrix is zero for any nodes that come from 2 different graphs, and otherwise according to the adjacency matrix of the individual graph. Luckily, this strategy is already implemented in torch geometric, and hence we can use the corresponding data loader:

graph\_train\_loader = geom\_data.DataLoader(train\_dataset, *batch\_size*=64, *shuffle*=True)

graph\_val\_loader = geom\_data.DataLoader(test\_dataset, *batch\_size*=64) *# Additional loader if you want to change to a larger dataset*

graph\_test\_loader = geom\_data.DataLoader(test\_dataset, *batch\_size*=64)

Let’s load a batch below to see the batching in action:

batch = next(iter(graph\_test\_loader))

print("Batch:", batch)

print("Labels:", batch.y[:10])

print("Batch indices:", batch.batch[:40])

Batch: Batch(batch=[687], edge\_attr=[1512, 4], edge\_index=[2, 1512], x=[687, 7], y=[38])

Labels: tensor([1, 1, 1, 0, 0, 0, 1, 1, 1, 0])

Batch indices: tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])

We have 38 graphs stacked together for the test dataset. The batch indices, stored in batch, show that the first 12 nodes belong to the first graph, the next 22 to the second graph, and so on. Thes indices are important for performing the final prediction. To perform a prediction over a whole graph, we usually perform a pooling operation over all nodes after running the GNN model. In this case, we will use the average pooling. Hence, we need to know which nodes should be included in which average pool. Using this pooling, we can already create our graph network below. Specially, we re-use our class GNNModel from before, and simply add an average pool and single linear layer for the graph prediction task.

*class* GraphGNNModel(*nn*.*Module*):

*def* \_\_init\_\_(*self*, *c\_in*, *c\_hidden*, *c\_out*, *dp\_rate\_linear*=0.5, \*\**kwargs*):

        """

        Inputs:

            c\_in - Dimension of input features

            c\_hidden - Dimension of hidden features

            c\_out - Dimension of output features (usually number of classes)

            dp\_rate\_linear - Dropout rate before the linear layer (usually much higher than inside the GNN)

            kwargs - Additional arguments for the GNNModel object

        """

*super*().\_\_init\_\_()

*self*.GNN = GNNModel(*c\_in*=c\_in,

*c\_hidden*=c\_hidden,

*c\_out*=c\_hidden, *# Not our prediction output yet!*

                            \*\*kwargs)

*self*.head = nn.Sequential(

            nn.Dropout(dp\_rate\_linear),

            nn.Linear(c\_hidden, c\_out)

        )

*def* forward(*self*, *x*, *edge\_index*, *batch\_idx*):

        """

        Inputs:

            x - Input features per node

            edge\_index - List of vertex index pairs representing the edges in the graph (PyTorch geometric notation)

            batch\_idx - Index of batch element for each node

        """

        x = *self*.GNN(x, edge\_index)

        x = geom\_nn.global\_mean\_pool(x, batch\_idx) *# Average pooling*

        x = *self*.head(x)

        return x

Finally, we can create a PyTorch Lightning module to handle the training. It is similar to the modules we have seen before and does nothing surprising in terms of training. As we have a binary classification task, we use the Binary Cross Entropy Loss.

*class* GraphLevelGNN(*pl*.*LightningModule*):

*def* \_\_init\_\_(*self*, \*\**model\_kwargs*):

*super*().\_\_init\_\_()

*# Saving hyperparameters*

*self*.save\_hyperparameters()

*self*.model = GraphGNNModel(\*\*model\_kwargs)

*self*.loss\_module = nn.BCEWithLogitsLoss() if *self*.hparams.c\_out == 1 else nn.CrossEntropyLoss()

*def* forward(*self*, *data*, *mode*="train"):

        x, edge\_index, batch\_idx = data.x, data.edge\_index, data.batch

        x = *self*.model(x, edge\_index, batch\_idx)

        x = x.squeeze(*dim*=-1)

        if *self*.hparams.c\_out == 1:

            preds = (x > 0).float()

            data.y = data.y.float()

        else:

            preds = x.argmax(*dim*=-1)

        loss = *self*.loss\_module(x, data.y)

        acc = (preds == data.y).sum().float() / preds.shape[0]

        return loss, acc

*def* configure\_optimizers(*self*):

        optimizer = optim.AdamW(*self*.parameters(), *lr*=1e-2, *weight\_decay*=0.0) *# High lr because of small dataset and small model*

        return optimizer

*def* training\_step(*self*, *batch*, *batch\_idx*):

        loss, acc = *self*.forward(batch, *mode*="train")

*self*.log('train\_loss', loss)

*self*.log('train\_acc', acc)

        return loss

*def* validation\_step(*self*, *batch*, *batch\_idx*):

        \_, acc = *self*.forward(batch, *mode*="val")

*self*.log('val\_acc', acc)

*def* test\_step(*self*, *batch*, *batch\_idx*):

        \_, acc = *self*.forward(batch, *mode*="test")

*self*.log('test\_acc', acc)      *self*.loss\_module = nn.BCEWithLogitsLoss() if *self*.hparams.c\_out == 1 else nn.CrossEntropyLoss()

*def* forward(*self*, *data*, *mode*="train"):

        x, edge\_index, batch\_idx = data.x, data.edge\_index, data.batch

        x = *self*.model(x, edge\_index, batch\_idx)

        x = x.squeeze(*dim*=1)

        if *self*.hparams.c\_out == 1:

            preds = (x>0).float()

            data.y = data.y.float()

        else:

            preds = x.argmax(dim==-1)

        loss = *self*.loss\_module(x, data.y)

        acc = (preds == data.y).sum().float() / preds.shape[0]

        return loss, acc

*def* configure\_optimizers(*self*):

        optimizer = optim.AdamW(*self*.parameters(), *lr*=1e-2, *weight\_decay*=0.0) *# High lr because of small dataset and small model*

        return optimizer

*def* training\_step(*self*, *batch*, *batch\_idx*):

        loss, acc = *self*.forward(batch, *mode*="train")

*self*.log('train\_loss', loss)

*self*.log('train\_acc', acc)

        return loss

*def* validation\_step(*self*, *batch*, *batch\_idx*):

        \_, acc = *self*.forward(batch, *mode*="val")

*self*.log('val\_acc', acc)

*def* test\_step(*self*, *batch*, *batch\_idx*):

        \_, acc = *self*.forward(batch, *mode*="test")

*self*.log('test\_acc', acc)

Below we train the model on our dataset. It resembles the typical training functions we have seen so far.

*def* train\_graph\_classifier(*model\_name*, \*\**model\_kwargs*):

    pl.seed\_everything(42)

*# Create a PyTorch Lightning trainer with the generation callback*

    root\_dir = os.path.join(CHECKPOINT\_PATH, "GraphLevel" + model\_name)

    os.makedirs(root\_dir, *exist\_ok*=True)

    trainer = pl.Trainer(*default\_root\_dir*=root\_dir,

*callbacks*=[ModelCheckpoint(*save\_weights\_only*=True, *mode*="max", *monitor*="val\_acc")],

*accelerator*="gpu" if *str*(device).startswith("cuda") else "cpu",

*devices*=1,

*max\_epochs*=500,

*enable\_progress\_bar*=False)

    trainer.logger.\_default\_hp\_metric = None *# Optional logging argument that we don't need*

*# Check whether pretrained model exists. If yes, load it and skip training*

    pretrained\_filename = os.path.join(CHECKPOINT\_PATH, *f*"GraphLevel{model\_name}.ckpt")

    if os.path.isfile(pretrained\_filename):

        print("Found pretrained model, loading...")

        model = GraphLevelGNN.load\_from\_checkpoint(pretrained\_filename)

    else:

        pl.seed\_everything(42)

        model = GraphLevelGNN(*c\_in*=tu\_dataset.num\_node\_features,

*c\_out*=1 if tu\_dataset.num\_classes==2 else tu\_dataset.num\_classes,

                              \*\*model\_kwargs)

        trainer.fit(model, graph\_train\_loader, graph\_val\_loader)

        model = GraphLevelGNN.load\_from\_checkpoint(trainer.checkpoint\_callback.best\_model\_path)

*# Test best model on validation and test set*

    train\_result = trainer.test(model, graph\_train\_loader, *verbose*=False)

    test\_result = trainer.test(model, graph\_test\_loader, *verbose*=False)

    result = {"test": test\_result[0]['test\_acc'], "train": train\_result[0]['test\_acc']}

    return model, result

Finally, let’s perform the training and testing. Feel free to experiment with different GNN layers, hyperparameters, etc.

model, result = train\_graph\_classifier(*model\_name*="GraphConv",

*c\_hidden*=256,

*layer\_name*="GraphConv",

*num\_layers*=3,

*dp\_rate\_linear*=0.5,

*dp\_rate*=0.0)

print(*f*"Train performance: {100.0\*result['train']*:4.2f*}%")

print(*f*"Test performance:  {100.0\*result['test']*:4.2f*}%")

Train performance: 94.27%

Test performance: 92.11%

The test performance shows that we obtain quite good scores on an unseen part of the dataset. Is should be noted that as we have been using the test set for validation as well, we might have overfitting slightly to this set. Nevertheless, the experiment shows us that GNNs can be indeed powerful to predict the properties of graphs and/or molecules.

# Conclusion

In this tutorial, we have seen the application of neural networks to graph structures. We looked at how a graph can be represented (adjacency matrix or edge list) and discussed the implementation of common graph layers: GCN and GAT. The implementations showed the practical side of the layers, which is often easier than the theory. Finally, we experimented with different tasks, on node, edge, and graph-level. Overall, we have seen that including graph information in the predictions that benefit from GNNs, the importance of these networks will likely increase over the next year.