Transductive Node Classification using Graph Attention Neural Networks

Alessandra Monaco 1706205

April, ay 2019-2020
Engineering in Computer Science, Sapienza

Contents

1 Introduction

This report provides details about the implementation of Graph Attention Networks proposed by Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò and Yoshua Bengio in their paper '*Graph Attention Networks* (October 30, 2017).

Abstract

We present graph attention networks (GATs), novel neural network architectures that operate on graph-structured data, leveraging masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. By stacking layers in which nodes are able to attend over their neighborhoods' features, we enable (implicitly) specifying different weights to different nodes in a neighborhood, without requiring any kind of costly matrix operation (such as inversion) or depending on knowing the graph structure upfront. In this way, we address several key challenges of spectral-based graph neural networks simultaneously, and make our model readily applicable to inductive as well as transductive problems. Our GAT models have achieved or matched state-of-theart results across four established transductive and inductive graph benchmarks: the Cora, Citeseer and Pubmed citation network datasets, as well as a proteinprotein interaction dataset (wherein test graphs remain unseen during training).

Petar Velickovic, *Graph Attention Networks*
url: https://arxiv.org/abs/1710.10903

The paper approaches node classification task for graph-structured input data, both in the transductive and inductive settings; this experimentation instead, focuses on transductive learning only.
For the transductive setting, the authors of the paper propose an efficient solution for document classification in benchmark citation graphs, such as Cora, Citeseer and Pubmed networks. The idea is to compute node features by taking into account also the features of the neighbors: in this way the structure of the data is fully considered during learning.
We remind that the setting is transductive, meaning that we reason from specific observed, labelled instances (training set), to specific observed but unlabelled instances (test set).
Following the paper's guidelines, we implemented the Graph Attention (GAT) Layer, and we built a 2-layers and a 3-layers neural network (**Chapter 2**), experimenting different models by hyperparameters tuning. We chose Cora and Pubmed benchmark datasets to run our experiments (**Chapter 3**). Lastly we evaluated and compared the models among them (**Chapter 4**).

2 Environment and tools

The project is implemented in Python 3, using `PyTorch` and `PyTorch_geometric` libraries to build the neural network, `networkx` and `matplotlib` to provide graphical representations.

2.1 PyTorch

`PyTorch` is an open source library, containing modules and classes that help to create and train neural networks:

- `torch.nn` module provides tools to define networks (class `Module`, `Sequential`), parameters (class `Parameters`), different kinds of layers (convolution, pooling, padding, normalization, recurrent, linear, dropout, sparse), activation functions (Sigmoid, ELU, ReLU, LeakyReLU, Softplus, Tanh, Softmin, LogSoftmax,...), loss functions (L1Loss, MSELoss, NLLLoss, MultiLabelMarginLoss, CrossEntropyLoss,...)
- `torch.optim` is a package containing many optimization algorithms, such as SGD, Adadelta, Adagrad, Adam and many others
- `torch.utils.data` module provides tools to load and manage datasets

PyTorch represents data as tensors (`torch.Tensor` class), multi-dimensional matrices containing elements of a single data type. Our graph dataset, indeed, is represented as a tensor: the edge index, the nodes with their features and also the target values (labels) are definitely tensors.

2.2 PyTorch Geometric

`PyTorch_geometric` is a geometric deep learning extension library for PyTorch, consisting of various methods for deep learning on graphs and other non-Euclidean domains. The library contains the following main modules:

- `torch_geometric.nn` that provides us the *Message Passing* paradigm to implement the Graph Attention layer
- `torch_geometric.data` that we used to easily access graph-structured data, thanks to the huge number of methods offered
- `torch_geometric.datasets` through which we downloaded two benchmark datasets for the experimentation

2.3 Other tools

We run the experiments in the *Google Colaboratory* platform, a Jupyter notebook environment providing free GPU.

3 GAT network implementation

3.1 Design aspects

The design of our neural networks follows the suggestions contained in the paper, expecially regarding the definition of the GAT layer and its mathematical characteristics, the choice of the activation functions, but also the choice of the dataset and the size of training/validation/test set for the experimentation.

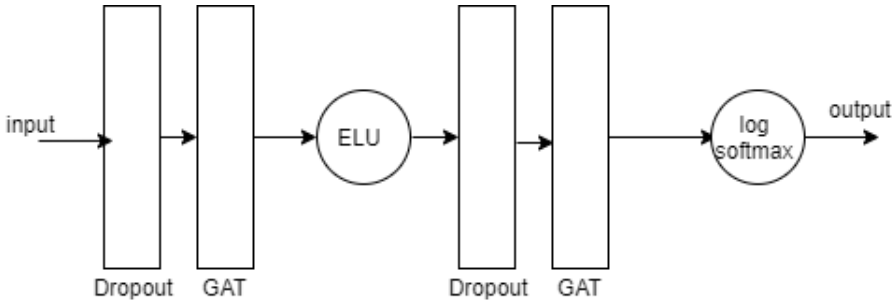

Figure 1: 2-GAT layers architecture

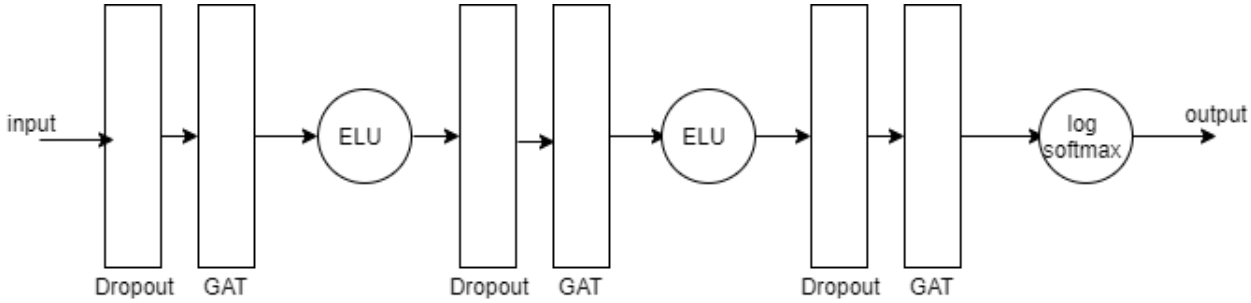

Figure 2: 3-GAT layers architecture

3.2 The GAT Layer and the Message Passing paradigm

The general idea proposed in the paper is a non-spectral approach to compute the convolution on graphs, using **edge attention coefficients** to manage and quantify the interdependence between nodes. We want to remark that in this article we always refer to *self attention* (within input interdependence), unlike the *general attention*, that is the interdependence between input and output.

Mathematically, we produced the attention coefficients (for each edge) as follows:

- apply a linear transformation on node's features
- apply a linear transformation also on the other endpoint's features (the neighbor)
- concatenate the two linear transformations
- apply an attention mechanism (function a) on the concatenation
- apply the Leaky ReLU activation function
- exponentially normalize the coefficients obtained so far using softmax on all neighbors (N) of the node

We summarize the process with the following formula:

$$\alpha_{ij} = \frac{exp(LeakyReLU(a^{T}(W \cdot h_i || W \cdot h_j)))}{\sum_{k \epsilon N} exp(LeakyReLU(a^{T}(W \cdot h_i || W \cdot h_k)))}$$

The obtained coefficients are computed for each edge and used to compute a linear combination of features corresponding to them.

Finally, we applied **Multi-head attention**, in the sense that the attention mechanism is computed for a certain number of times (heads) and the results are concatenated. In the final GAT layer of the network we had to apply the mean of attentions instead of concatenation.
Figure 3 shows the aggregation process of a Multi-head graph attentional layer.

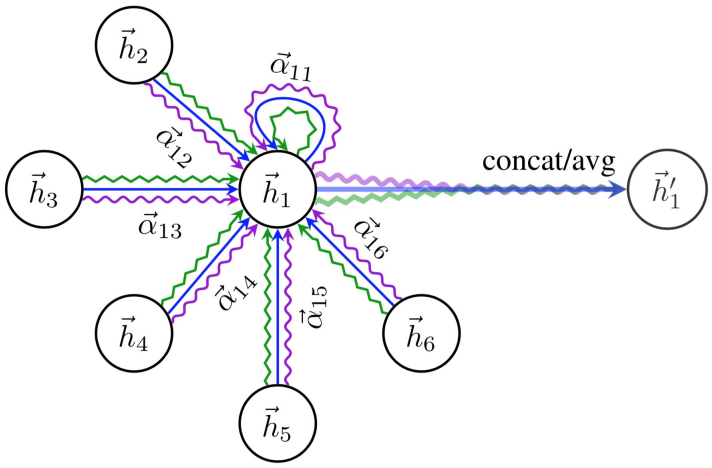

Figure 3: from '*Graph Attention Networks*', Petarv

These mathematical computations are implemented through the *Message Passing* (or *Neighborhood aggregation*) scheme: PyTorch Geometric (but also other libraries such as `dgl`) provides the base class `torch_geometric.nn.MessagePassing`, which helps in creating such kind of neural networks by automatically taking care of message propagation. Therefore, we just had to specify what kind of message we wanted to propagate, which function should aggregate information, and how to update the nodes according to the received, aggregated information.

We started by defining a class for the GAT layer, which takes as argument the Message Passing class, and we defined the initialization function as follows:

```
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.utils import softmax


class GatLayer(MessagePassing):

  def __init__(self,input_size,output_size,heads,
               negative_slope,dropout,concat=True,**kwargs):
    super(GatLayer,self).__init__(aggr='add',**kwargs)

    self.input_size = input_size
    self.output_size = output_size
    self.heads = heads
    self.concat = concat
```

```
        self.negative_slope = negative_slope
        self.dropout = dropout

        #Parameters
        self.W = Parameter(torch.Tensor(input_size,heads*output_size))
        self.a = Parameter(torch.Tensor(1,heads,2*output_size))

        #Glorot weight initialization
        nn.init.xavier_uniform_(self.W.data,gain=1.414)
        nn.init.xavier_uniform_(self.a.data,gain=1.414)
```

The implementation of our GAT layer is all based on tensors and on reshaping. In fact, we defined two kinds of learnable parameters: the **weight matrix** of node's features (`W`), and the **weight vector of the attention mechanism** (`a`), having shape `(1,heads,2*output_size)`; the reason behind the shape of `a` is simply that we needed an attention parameter for each head, and in the next steps we will multiply `a` by the concatenation of the features of node i and node j (its neighbor), therefore each of them will contribute with `output_size` new features and the concatenation will be of size `2*output_size`.

We initialized all the parameters with Glorot algorithm (also called Xavier), as suggested by the paper, with a default gain and using the uniform variant.

Since the layer is part of a feedforward architecture, we had to define the forward function, to determine how the input is transformed by the layer:

```
    def forward(self,x,edge_index,size=None):

        #linear combination
        x = torch.matmul(x,self.W)

        #initial call to start propagating the message
        return self.propagate(edge_index,size=size,x=x)
```

The Torch function `matmul` just applies the matrix product between the input tensor (node's features) and the weight matrix.
After applying a **linear transformation** of the node's features, the final features' computation needs messages from the neighbors. In the Message Passing scheme in PyTorch Geometric, the function `propagate()` automatically calls the functions `message()` on neighbors, `aggregate()`, and `update()` in this order. We redefined the `message` and the `update` functions, to adapt them to our needs, while `aggregate` remains the same.

The message is simply the product of the features of a neighbor and the correspondent edge attention coefficient `alpha`; it is defined in a specific function:

```
    def message(self,edge_index_i,x_i,x_j,size_i):

        #Use view method to reshape
        x_j = x_j.view(-1,self.heads,self.output_size)
        x_i = x_i.view(-1,self.heads,self.output_size)

        #linear combination of concatenation
        alpha = (torch.cat([x_i,x_j],dim=-1)*self.a).sum(dim=-1)
        #Leaky ReLU
        alpha = F.leaky_relu(alpha,self.negative_slope)
```

```
    #exponential normalization with softmax
    alpha = softmax(alpha,edge_index_i,size_i)
    #dropout on normalized attention coefficients
    alpha = F.dropout(alpha,p=self.dropout,training=self.training)

    return x_j*alpha.view(-1,self.heads,1)
```

The input tensors `x_i` and x_j of the `message` function were just the linear transformations of the `forward` step, referring to two different nodes. Therefore, their shapes at this point were `(1,heads*output_shape)`. We applied **reshaping** using the `view()` function, to produce a multidimensional tensor such that each head was a tensor of size output_size.
This operation is fundamental, because the concatenation between features of `x_i` and `x_j` should be done *head by head* (`dim=-1`).
The concatenation is again **linearly trasformed** but using the **attention weight tensor**, multiplying features by `a` for each different head, and summing up products relative to each head using the reduction function `sum(dim=-1)`.
The result is passed to the **Leaky ReLU** activation function and, at last, it is normalized applying **Softmax** but only on the neighbors.
The **Dropout** is introduced to decrese overfitting: this technique randomly selects a subset of units and set them to zero, but only in the training phase (`training=self.training`).

The `aggregate()` function of PyTorch Geometric combines messages accordingly to a predefined method: this was specified in the initialization function (`aggr='add'`), because we needed to **sum the messages of all neighbors** of a node.

Finally, we had to specify how to update the features of each node, given the received, aggregated messages:

```
  def update(self, out):

    if self.concat is True:
      out = out.view(-1,self.heads*self.output_size)

    else:
      #mean of all attentions
      out = out.mean(dim=1)

    #return new node features, updated with neighbors' information
    return out
```

According to the paper, if the GAT layer is the last one of the network, the heads should be averaged (`out=out.mean(dim=1)`), otherwise they should be concatenated. Again, we used the `view()` function to reshape: before the reshaping, the multidimensional tensor cointained a tensor of size `output_size` for each head, but after that, the head tensors are grouped into a single one, so the internal size becomes `heads*output_size`.

We also defined a function to represent our layers, to be used to summarize the most important information, when calling the `print()` on the GatLayer class.

```
  def __repr__(self):
    return '{}({}, {}, heads={})'.format(self.__class__.__name__,
                                         self.input_size, self.output_size,
                                         self.heads)
```

3.3 Overall architecture

Putting together many GAT layers, we defined two different Graph Attention neural networks (see Figure 1 and Figure 2).

GatNet1 has two GAT layers, a dropout before each of them to reduce overfitting, and an Exponential Linear Unit between one GAT and the other. The output activation is the Logarithmic Softmax because the task is multiclass classification.
Notice also that the `concat` hyperparameter of the last GAT layer is set to "False", such that the heads of the layer will be averaged.

```
class GatNet1(nn.Module):

  def __init__(self,input_size,hidden_size,output_size,dropout,heads_1,
               heads_2):
    super(GatNet1,self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size
    self.dropout = dropout
    self.heads_1 = heads_1
    self.heads_2 = heads_2

    self.Gat1 = GatLayer(input_size=self.input_size,output_size=self.hidden_size,
                         heads=self.heads_1,dropout=self.dropout,negative_slope=0.2)
    self.Gat2 = GatLayer(input_size=self.hidden_size*self.heads_1,
                         output_size=self.output_size,heads=self.heads_2,
                         dropout=self.dropout,concat=False,negative_slope=0.2)


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

    x = F.dropout(x,p=self.dropout,training=self.training)
    x = self.Gat1(x,edge_index)
    x = F.elu(x)
    x = F.dropout(x,p=self.dropout,training=self.training)
    x = self.Gat2(x,edge_index)
    x = F.log_softmax(x,dim=1)

    return x
```

GatNet2 has three GAT layers and respects the same scheme of GatNet1:

```
class GatNet2(nn.Module):

  def __init__(self,input_size,hidden1_size,hidden2_size,output_size,dropout,
               heads_1,heads_2,heads_3):
    super(GatNet2,self).__init__()
```

```
    self.input_size = input_size
    self.hidden1_size = hidden1_size
    self.hidden2_size = hidden2_size
    self.output_size = output_size
    self.dropout = dropout
    self.heads_1 = heads_1
    self.heads_2 = heads_2
    self.heads_3 = heads_3

    self.Gat1 = GatLayer(input_size=self.input_size,output_size=self.hidden1_size,
                         heads=self.heads_1,dropout=self.dropout,negative_slope=0.2)
    self.Gat2 = GatLayer(input_size=self.hidden1_size*self.heads_1,
                         output_size=self.hidden2_size,heads=self.heads_2,
                         dropout=self.dropout,negative_slope=0.2)
    self.Gat3 = GatLayer(input_size=self.hidden2_size*self.heads_2,
                         output_size=self.output_size,heads=self.heads_3,
                         dropout=self.dropout,concat=False,negative_slope=0.2)


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

    x = F.dropout(x,p=self.dropout,training=self.training)
    x = self.Gat1(x,edge_index)
    x = F.elu(x)
    x = F.dropout(x,p=self.dropout,training=self.training)
    x = self.Gat2(x,edge_index)
    x = F.elu(x)
    x = F.dropout(x,p=self.dropout,training=self.training)
    x = self.Gat3(x,edge_index)
    x = F.log_softmax(x,dim=1)

    return x
```

3.4 Activation functions

As suggested in the paper, we used **Exponential Linear Unit** (ELU) as hidden activation function. It fixes some of the problems with ReLUs and keeps some of the positive things. For this activation function, an alpha value is picked; a common value is between 0.1 and 0.3. The ELU is defined as follows:

$$ELU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^{\mathrm{x}} - 1) & \text{if } \mathrm{x} < 0 \end{cases}$$

In Figure 4 we show the main differences between the graphical representations of ReLU, ELU and LeakyReLU (that we used in the GAT layer).

The main advantage of using the ELU is that it avoids the dead ReLU problem, still mantaining the computational speed of ReLU. Moreover, it produces negative outputs, which helps the network nudge weights in the right directions.

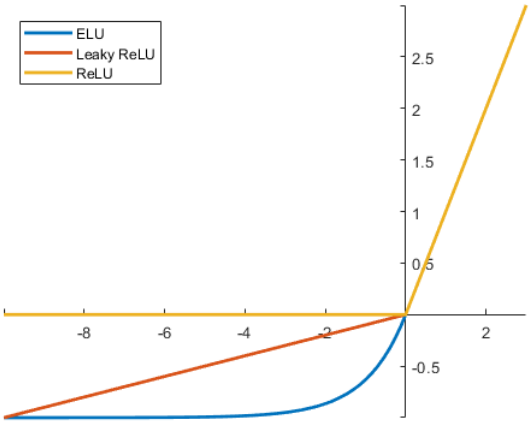

Figure 4: ReLU, LeakyReLU and ELU

Finally, we used the **Logarithmic Softmax** as output activation function. The Softmax is generally used for multiclassification problems, and the Logarithmic Softmax just applies the logarithm to the probabilities outputted by the Softmax. The reason behind this choice is that we used Negative Log Likelihood (`nn.NLLLoss`) as loss function to train our network, and the NLLLoss expects the log probabilities. With NLLLoss we can maximize by minimizing the negative log likelihood, and this is preferable, because it is more precise. For instance, it can happen that after multiplying things togetheryou will end up loosing precision if the numbers are too high or too low.

4 The Datasets and the task

We selected two of the datasets used in the paper: Cora and Pubmed. They are both citation graphs, in which nodes represent scientific papers, and edges represent citations from one paper to the other. Each paper (node) regards a certain topic (class, target value); the task is to predict classes for unlabelled nodes, passing to the neural network the whole dataset but only the target values of the training set (transductive learning).
The class `Planetoid` in `torch_geometric.datasets` contains methods to import built-in benchmark datasets, that are already splitted into training, validation and test sets in order to perform transductive learning. The training set is balanced and contains 20 nodes for each class.

4.1 Cora Citation Graph

With the following code we imported Cora and we stored it into the variable `data`:

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')
data = dataset[0]
```

Just to have an idea of how much dense and complex is the dataset, we can visualize the graph by converting the tensorial representation of PyTorch Geometric to the Networkx format (using `networkx.draw`), and plotting it with `matplotlib`:

```
import matplotlib.pyplot as plt
import networkx as nx
from torch_geometric.utils.convert import to_networkx

graph = to_networkx(data)
node_labels = data.y[list(graph.nodes)].numpy()

plt.figure(1,figsize=(80,60))
pos = nx.spring_layout(graph,k=0.15,iterations=20)
nx.draw(graph, pos=pos, cmap=plt.get_cmap('Set1'), node_color=node_labels,
        node_size=75, linewidths=6)

plt.show()
```

The plotted graph is shown in Figure 5.

The imported Cora graph is an instance of `torch_geometric.data.Data` class. A Data object holds all information needed to define an arbitrary graph. Such information can be accessed by using attributes and methods of these classes:

```
Number of graphs in the dataset (len(dataset)): 1
Number of classes (dataset.num_classes): 7
Number of features for each node (dataset.num_node_features): 1433
Number of nodes (data.num_nodes): 2708
Number of target values (len(data.y)): 2708
Number of edges (data.num_edges): 10556
data.is_undirected() = True
data.contains_isolated_nodes(): False
data.contains_self_loops(): False
```

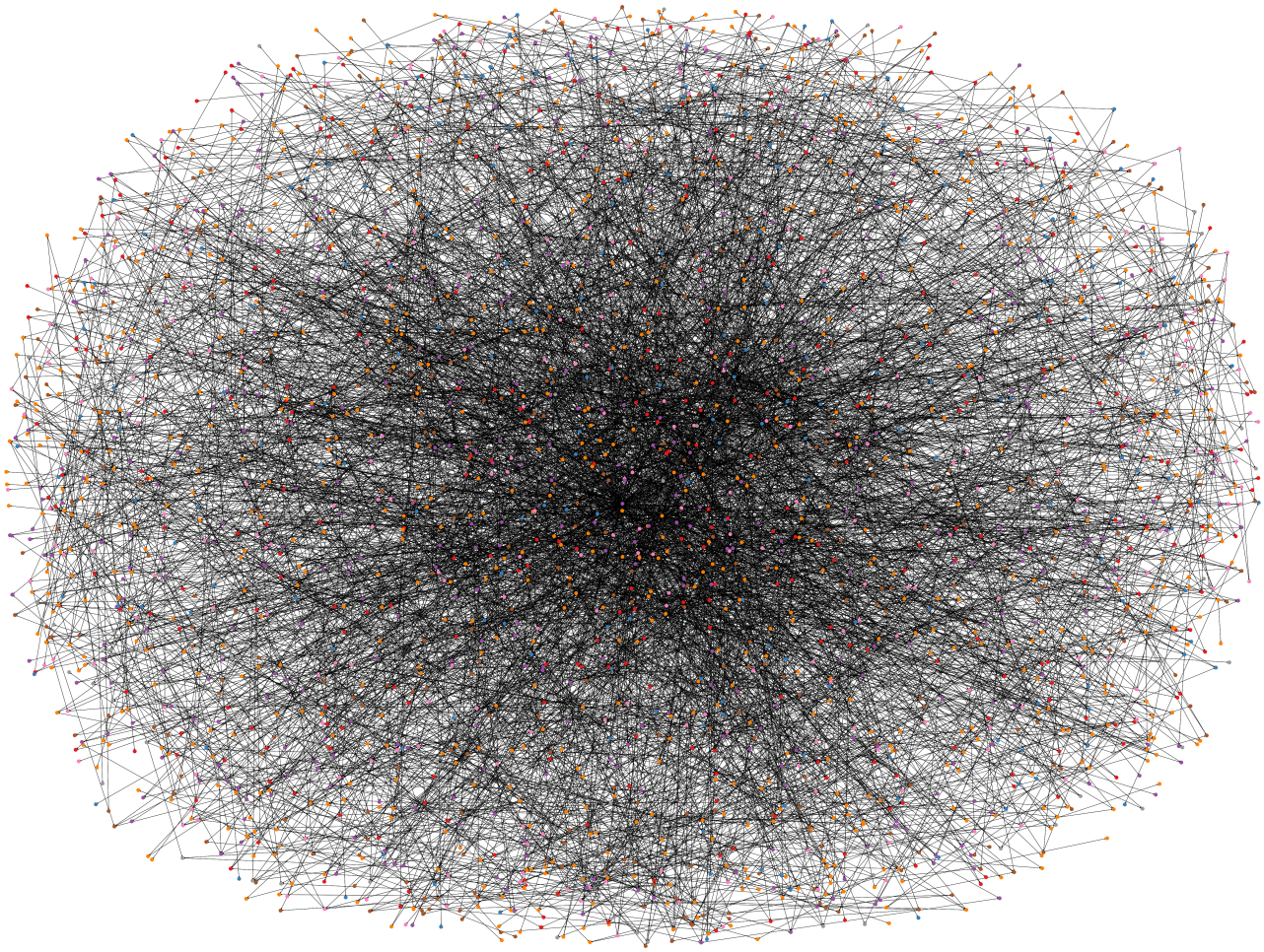

Figure 5: Cora Citation Graph

In particular:

- `data.x` is a tensor containing a *Bag of Words* representation of each paper; the 1433 features of a node represents whether a certain word of the vocabulary occurs (1) or not (0) in the paper. The position of words and their frequencies are not considered.
- `data.y` is a tensor containing the encoded target values (classes), such that:
  - 0 = Case_Based
  - 1 = Genetic_Algorithms
  - 2 = Neural_Networks
  - 3 = Probabilistic_Methods
  - 4 = Reinforcement_Learning
  - 5 = Rule_Learning
  - 6 = Theory

- `data.edge_index` is a multi-dimensional tensor in COO (Coordinate) format (one tensor for source IDs of edges and another for destination IDs), representing how nodes are connected in the graph.

```
data.x: tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])
data.y: tensor([3, 4, 4,  ..., 3, 3, 3])
data.edge_index: tensor([[   0,    0,    0,  ..., 2707, 2707, 2707],
        [ 633, 1862, 2582,  ...,  598, 1473, 2706]])
```

The splitting into training, validation and test sets is performed introducing the following additional boolean attributes to the Data object:

- `data.train_mask` denotes against which nodes to train
- `data.val_mask` denotes which nodes to use for validation, e.g., to perform early stopping
- `data.test_mask` denotes against which nodes to test

```
print("Size of the training set: %s" %(data.train_mask.sum().item()))
print("Size of the validation set: %s" %(data.val_mask.sum().item()))
print("Size of the test set: %s" %(data.test_mask.sum().item()))
```

```
Size of the training set: 140
Size of the validation set: 500
Size of the test set: 1000
```

4.2 Pubmed Citation Graph

We imported Pubmed with the same code used for Cora:

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Pubmed', name='Pubmed')
data = dataset[0]
```

Figure 6 shows the plotted graph using `networkx` and `matplotlib`.
Again, we summarize the main information about the dataset:

```
Number of graphs in the dataset (len(dataset)): 1
Number of classes (dataset.num_classes): 3
Number of features for each node (dataset.num_node_features): 500
Number of nodes (data.num_nodes): 19717
Number of target values (len(data.y)): 19717
Number of edges (data.num_edges): 88648
data.is_undirected() = True
data.contains_isolated_nodes(): False
data.contains_self_loops(): False
```

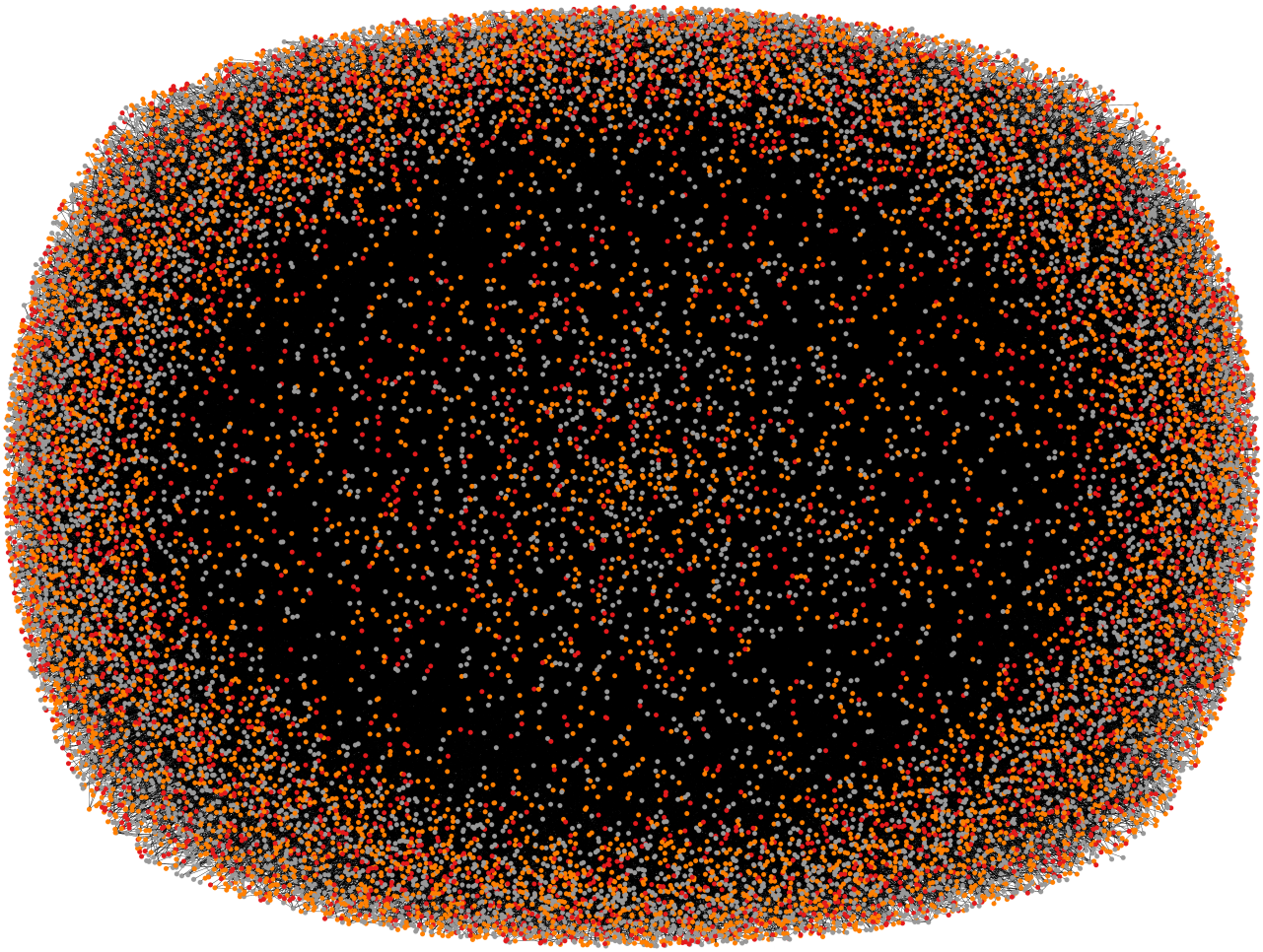

Figure 6: Pubmed Citation Graph

In Pubmed, `data.x` represents features as TF/IDF (**Term Frequency - Inverse Document Frequency**) weighted word vectors from a dictionary which consists of 500 unique words, while `data.y` is a tensor containing the encoded target values, such that:

- 0 = Diabetes Mellitus Experimental
- 1 = Diabetes Mellitus Type 1
- 2 = Diabetes Mellitus Type 2

```
data.x: tensor([[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.1046, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        ...,
        [0.0000, 0.0194, 0.0080,  ..., 0.0000, 0.0000, 0.0000],
        [0.1078, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0266, 0.0000,  ..., 0.0000, 0.0000, 0.0000]])
```

```
data.y: tensor([1, 1, 0,  ..., 2, 0, 2])
data.edge_index: tensor([[    0,     0,     0,  ..., 19714, 19715, 19716],
        [ 1378,  1544,  6092,  ..., 12278,  4284, 16030]])
```

The splitting into training, validation and test sets follows the methodology applied in Cora:

```
Size of the training set: 60
Size of the validation set: 500
Size of the test set: 1000
```

5 Training process and experimental results

We defined three different models to run our experiments, by tuning hyperparameters of the two networks (GatNet1 and GatNet2).
Before training the models, we had to add **self loops** to the data graph; self loops (edges going from a node to itself) are needed to compute the new features of a node, because in the aggregation function we should sum up also the old features of that node, together with the features of the neighbors, each multiplied by the correspondent edge attention coefficient.
The following code adds self loops using the PyTorch Geometric function `add_self_loops`:

```
from torch_geometric.utils import add_self_loops

print("Before adding self loops: data.contains_self_loops()= %s"
      %(data.contains_self_loops()))
data.edge_index, _ =add_self_loops(edge_index=data.edge_index)
print("After adding self loops: data.contains_self_loops()= %s"
      %(data.contains_self_loops()))
```

```
Before adding self loops: data.contains_self_loops()= False
After adding self loops: data.contains_self_loops()= True
```

5.1 The models

We started defining the input and output size, which is obviously the same for each model:

```
input_size = dataset.num_node_features
output_size = dataset.num_classes
```

Our neural network has many hyperparameters that can be tuned to obtain new models: the hidden size, the number of heads for each GAT layer, the negative slope for the ELU, both the inside dropout (related to coefficients in the GAT layer) and the outside dropout (before the GAT layer), the weight decay value for L1 regularization, the learning rate, but also the choice of the optimizer, the loss function, the activation functions and the patience in the early stopping will affect the training process.
Starting from the first model, we tried to understand the effects of changing the width of the network, increasing the `hidden_size` and `heads` parameters in the second model, and changing the depth, adding one more GAT layer in the third model. In both cases we adjusted regularization terms such as `dropout` rate and `weight_decay` to adapt them to the new models.

Our **first model** (`model_1`) instanciates the GatNet1, with a hidden size of 8 units, 8 heads for the first GAT layer and 1 for the second. Regularization is applied using a Dropout of 0.6 and L1 Regularization (`weight_decay=0.0005`).

```
model_1 = GatNet1(input_size=input_size,hidden_size=8,output_size=output_size,
                  dropout=0.6 heads_1=8,heads_2=1)
optimizer = torch.optim.Adam(model_1.parameters(),lr=0.005,weight_decay=5e-4)

print(model_1)
```

The `print()` on the model outputs a summary, for instance this is the output for the Pubmed dataset:

```
GatNet1(
  (Gat1): GatLayer(500, 8, heads=8)
  (Gat2): GatLayer(64, 3, heads=1)
)
```

In the **second model** we increased the hidden size to 12 and the number of heads to 10 for the first GAT layer, and to 3 for the second one. We also increased the weight decay to 0.005.

```
model_2 = GatNet1(input_size=input_size, hidden_size=12, output_size=output_size,
                  dropout=0.6, heads_1=10, heads_2=3)
optimizer = torch.optim.Adam(model_2.parameters(), lr=0.005, weight_decay=0.005)

print(model_2)
```

```
GatNet1(
  (Gat1): GatLayer(500, 12, heads=10)
  (Gat2): GatLayer(120, 3, heads=3)
)
```

The **third model** instanciates GatNet2, with 8 units for both the first and the second hidden layer, 8 heads for GAT layer 1 and 2, and 4 heads for the last one. We increased regularization with a dropout of 0.7 and a weight decay of 0.01.

```
model_3 = GatNet2(input_size=input_size, hidden1_size=8, hidden2_size=8,
                  output_size=output_size, dropout=0.7, heads_1=8, heads_2=8,
                  heads_3=4)
optimizer = torch.optim.Adam(model_3.parameters(), lr=0.005, weight_decay=0.01)

print(model_3)
```

```
GatNet2(
  (Gat1): GatLayer(500, 8, heads=8)
  (Gat2): GatLayer(64, 8, heads=8)
  (Gat3): GatLayer(64, 3, heads=4)
)
```

5.2 The training process

When we instanciated models, we also chose an **optimizer** for the learning algorithm: Adam (*Adaptive Moment Estimation*) is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. It can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum, because it uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.

To **train** the model, we defined a function (`train_early_stop`), that uses the `train_mask` to train the model and the `val_mask` to perform the *Early Stopping*. We store validation losses and accuracy in `loss_values` and `acc_values`, and training losses and accuracies in `loss_values_train` and `acc_values_train` respectively; they will be useful to plot histories

and compare models. At each epoch we compare the actual loss to the best one found so far, and if we do not find a new best loss for a number of epochs equal to the `patience` value, the training stops. This is done to avoid overtraining, therefore overfitting.
After the training of each epoch, we printed the accuracy and loss of both training and validation sets, the number of epoch and the time required.

```
def train_early_stop(model, data, patience, n_epochs):

  t_total = time.time()
  loss_values_train = []
  acc_values_train = []
  loss_values = []
  acc_values = []
  indices = []
  counter = 0
  best_epoch = 0
  best_loss = n_epochs

  for epoch in range(1, n_epochs):
    t = time.time()
    model.train()
    optimizer.zero_grad()
    out = model(data)
    loss_train = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    acc_train = accuracy(out[data.train_mask], data.y[data.train_mask])
    loss_train.backward()
    optimizer.step()

    model.eval() #deactivates layer's dropout
    out = model(data)
    loss_val = F.nll_loss(out[data.val_mask], data.y[data.val_mask])
    acc_val = accuracy(out[data.val_mask], data.y[data.val_mask])

    print('Epoch: {:04d}'.format(epoch),
          'LOSS_TRAIN: {:.4f}'.format(loss_train.data.item()),
          'ACC_TRAIN: {:.4f}'.format(acc_train.data.item()),
          'LOSS_VAL: {:.4f}'.format(loss_val.data.item()),
          'ACC_VAL: {:.4f}'.format(acc_val.data.item()),
          'time: {:.4f}s'.format(time.time() - t))

    loss_values.append(loss_val.data.item())
    loss_values_train.append(loss_train.data.item())
    acc_values.append(acc_val.data.item())
    acc_values_train.append(acc_train.data.item())
    indices.append(epoch)
    torch.save(model.state_dict(), '{}.model'.format(epoch))

    #Early stopping
    if loss_values[-1] < best_loss:
      best_loss = loss_values[-1]
      best_epoch = epoch
      counter = 0
    else:
      counter += 1
    if counter == patience:
      break

  return model,indices,loss_values_train,acc_values_train,loss_values,acc_values
```

The `accuracy()` function used to validate, is defined as follows:

```
def accuracy(output, labels):
  preds = output.max(1)[1].type_as(labels)
  correct = preds.eq(labels).double()
  correct = correct.sum()
  return correct / len(labels)
```

It simply uses the model obtained to predict on validation mask, and compares the predictions with the real targets. Then, it applies the definition of the accuracy function.

We trained each model for 500 epochs and a patience of 100.

5.2.1 Cora

We trained the three models on Cora dataset, calling `train_early_stop` on each model. For each model, we show just some epochs at the beginnig of the training process, in the middle and at the end, to see how the accuracies and the losses have changed during epochs.

First model:

```
model_1,indices_1,loss_train_1,acc_train_1,loss_values_1,
        acc_values_1 = train_early_stop(model_1,data,patience=100,n_epochs=500)
```

```
Epoch: 0001 LOSS_TRAIN: 2.0793 ACC_TRAIN: 0.1857 LOSS_VAL: 1.8403 ACC_VAL: 0.2720 time: 0.3409s
Epoch: 0002 LOSS_TRAIN: 2.0081 ACC_TRAIN: 0.2214 LOSS_VAL: 1.7555 ACC_VAL: 0.3860 time: 0.1376s
Epoch: 0003 LOSS_TRAIN: 1.8451 ACC_TRAIN: 0.3286 LOSS_VAL: 1.6737 ACC_VAL: 0.4980 time: 0.1379s
Epoch: 0004 LOSS_TRAIN: 1.7350 ACC_TRAIN: 0.4071 LOSS_VAL: 1.5966 ACC_VAL: 0.6160 time: 0.1396s
...
Epoch: 0064 LOSS_TRAIN: 0.4286 ACC_TRAIN: 0.8643 LOSS_VAL: 0.7030 ACC_VAL: 0.7960 time: 0.1380s
Epoch: 0065 LOSS_TRAIN: 0.5775 ACC_TRAIN: 0.8143 LOSS_VAL: 0.7018 ACC_VAL: 0.7900 time: 0.1364s
Epoch: 0066 LOSS_TRAIN: 0.4716 ACC_TRAIN: 0.8500 LOSS_VAL: 0.7008 ACC_VAL: 0.7900 time: 0.1336s
...
Epoch: 0164 LOSS_TRAIN: 0.4047 ACC_TRAIN: 0.8571 LOSS_VAL: 0.7633 ACC_VAL: 0.7900 time: 0.1352s
Epoch: 0165 LOSS_TRAIN: 0.2917 ACC_TRAIN: 0.8786 LOSS_VAL: 0.7643 ACC_VAL: 0.7900 time: 0.1375s
Epoch: 0166 LOSS_TRAIN: 0.3731 ACC_TRAIN: 0.8643 LOSS_VAL: 0.7641 ACC_VAL: 0.7920 time: 0.1339s
Epoch: 0167 LOSS_TRAIN: 0.3634 ACC_TRAIN: 0.8857 LOSS_VAL: 0.7651 ACC_VAL: 0.7900 time: 0.1344s
```

Second model:

```
model_2,indices_2,loss_train_2,acc_train_2,loss_values_2,
        acc_values_2 = train_early_stop(model_2,data,patience=100,n_epochs=500)
```

```
Epoch: 0001 LOSS_TRAIN: 1.9393 ACC_TRAIN: 0.2071 LOSS_VAL: 1.8480 ACC_VAL: 0.5000 time: 0.2655s
Epoch: 0002 LOSS_TRAIN: 1.8656 ACC_TRAIN: 0.2857 LOSS_VAL: 1.7801 ACC_VAL: 0.6100 time: 0.2669s
Epoch: 0003 LOSS_TRAIN: 1.7850 ACC_TRAIN: 0.4357 LOSS_VAL: 1.7180 ACC_VAL: 0.6620 time: 0.2644s
Epoch: 0004 LOSS_TRAIN: 1.6974 ACC_TRAIN: 0.4714 LOSS_VAL: 1.6593 ACC_VAL: 0.6980 time: 0.2465s
...
Epoch: 0149 LOSS_TRAIN: 0.2859 ACC_TRAIN: 0.9857 LOSS_VAL: 0.7116 ACC_VAL: 0.7860 time: 0.2385s
Epoch: 0150 LOSS_TRAIN: 0.3138 ACC_TRAIN: 0.9714 LOSS_VAL: 0.7165 ACC_VAL: 0.7800 time: 0.2458s
Epoch: 0151 LOSS_TRAIN: 0.2832 ACC_TRAIN: 0.9714 LOSS_VAL: 0.7208 ACC_VAL: 0.7780 time: 0.2472s
...
Epoch: 0309 LOSS_TRAIN: 0.2842 ACC_TRAIN: 0.9571 LOSS_VAL: 0.7092 ACC_VAL: 0.7920 time: 0.2417s
Epoch: 0310 LOSS_TRAIN: 0.2582 ACC_TRAIN: 0.9714 LOSS_VAL: 0.7121 ACC_VAL: 0.7920 time: 0.2467s
Epoch: 0311 LOSS_TRAIN: 0.3116 ACC_TRAIN: 0.9714 LOSS_VAL: 0.7123 ACC_VAL: 0.7900 time: 0.2578s
Epoch: 0312 LOSS_TRAIN: 0.1998 ACC_TRAIN: 0.9714 LOSS_VAL: 0.7125 ACC_VAL: 0.7900 time: 0.2375s
```

Third model:

```
model_3,indices_3,loss_train_3,acc_train_3,loss_values_3,
        acc_values_3 = train_early_stop(model_3,data,patience=100,n_epochs=500)
```

```
Epoch: 0001 LOSS_TRAIN: 2.3535 ACC_TRAIN: 0.1571 LOSS_VAL: 1.9147 ACC_VAL: 0.1680 time: 0.2762s
Epoch: 0002 LOSS_TRAIN: 2.4697 ACC_TRAIN: 0.1429 LOSS_VAL: 1.8977 ACC_VAL: 0.2700 time: 0.2678s
Epoch: 0003 LOSS_TRAIN: 2.4846 ACC_TRAIN: 0.1429 LOSS_VAL: 1.8790 ACC_VAL: 0.3640 time: 0.2867s
Epoch: 0004 LOSS_TRAIN: 2.3118 ACC_TRAIN: 0.1929 LOSS_VAL: 1.8630 ACC_VAL: 0.4280 time: 0.2584s
...
Epoch: 0179 LOSS_TRAIN: 0.4849 ACC_TRAIN: 0.8929 LOSS_VAL: 0.7054 ACC_VAL: 0.8000 time: 0.2690s
Epoch: 0180 LOSS_TRAIN: 0.4941 ACC_TRAIN: 0.9000 LOSS_VAL: 0.7005 ACC_VAL: 0.7980 time: 0.2676s
Epoch: 0181 LOSS_TRAIN: 0.4594 ACC_TRAIN: 0.9143 LOSS_VAL: 0.6963 ACC_VAL: 0.7960 time: 0.2816s
...
Epoch: 0379 LOSS_TRAIN: 0.5000 ACC_TRAIN: 0.9000 LOSS_VAL: 0.6991 ACC_VAL: 0.8060 time: 0.2613s
Epoch: 0380 LOSS_TRAIN: 0.4647 ACC_TRAIN: 0.8857 LOSS_VAL: 0.6984 ACC_VAL: 0.8060 time: 0.2660s
Epoch: 0381 LOSS_TRAIN: 0.4154 ACC_TRAIN: 0.9357 LOSS_VAL: 0.6975 ACC_VAL: 0.8060 time: 0.2626s
Epoch: 0382 LOSS_TRAIN: 0.4500 ACC_TRAIN: 0.9286 LOSS_VAL: 0.6971 ACC_VAL: 0.8040 time: 0.2620s
```

5.2.2 Pubmed

For Pubmed dataset we tried the same three models used for Cora, following exactly the same steps.

First model:

```
model_1,indices_1,loss_train_1,acc_train_1,loss_values_1,
        acc_values_1 = train_early_stop(model_1,data,patience=100,n_epochs=500)
```

```
Epoch: 0001 LOSS_TRAIN: 1.1008 ACC_TRAIN: 0.4500 LOSS_VAL: 1.0846 ACC_VAL: 0.5140 time: 1.2734s
Epoch: 0002 LOSS_TRAIN: 1.0935 ACC_TRAIN: 0.4000 LOSS_VAL: 1.0734 ACC_VAL: 0.5860 time: 0.9782s
Epoch: 0003 LOSS_TRAIN: 1.0622 ACC_TRAIN: 0.4833 LOSS_VAL: 1.0631 ACC_VAL: 0.6280 time: 0.9827s
Epoch: 0004 LOSS_TRAIN: 1.0580 ACC_TRAIN: 0.5833 LOSS_VAL: 1.0523 ACC_VAL: 0.6600 time: 0.9481s
...
Epoch: 0099 LOSS_TRAIN: 0.3835 ACC_TRAIN: 0.8333 LOSS_VAL: 0.5609 ACC_VAL: 0.7920 time: 0.9479s
Epoch: 0100 LOSS_TRAIN: 0.4572 ACC_TRAIN: 0.8500 LOSS_VAL: 0.5602 ACC_VAL: 0.7940 time: 0.9717s
Epoch: 0101 LOSS_TRAIN: 0.3843 ACC_TRAIN: 0.8833 LOSS_VAL: 0.5595 ACC_VAL: 0.7940 time: 0.9674s
...
Epoch: 0215 LOSS_TRAIN: 0.3646 ACC_TRAIN: 0.8500 LOSS_VAL: 0.5712 ACC_VAL: 0.7820 time: 0.9517s
Epoch: 0216 LOSS_TRAIN: 0.4508 ACC_TRAIN: 0.8000 LOSS_VAL: 0.5713 ACC_VAL: 0.7800 time: 0.9358s
Epoch: 0217 LOSS_TRAIN: 0.3994 ACC_TRAIN: 0.8000 LOSS_VAL: 0.5719 ACC_VAL: 0.7820 time: 0.9352s
Epoch: 0218 LOSS_TRAIN: 0.4019 ACC_TRAIN: 0.7833 LOSS_VAL: 0.5725 ACC_VAL: 0.7840 time: 0.9344s
```

Second model:

```
model_2,indices_2,loss_train_2,acc_train_2,loss_values_2,
        acc_values_2 = train_early_stop(model_2,data,patience=100,n_epochs=500)
```

```
Epoch: 0001 LOSS_TRAIN: 1.0879 ACC_TRAIN: 0.4500 LOSS_VAL: 1.0890 ACC_VAL: 0.5380 time: 1.9176s
Epoch: 0002 LOSS_TRAIN: 1.0815 ACC_TRAIN: 0.5000 LOSS_VAL: 1.0839 ACC_VAL: 0.6360 time: 1.9633s
Epoch: 0003 LOSS_TRAIN: 1.0832 ACC_TRAIN: 0.4667 LOSS_VAL: 1.0791 ACC_VAL: 0.6880 time: 1.8525s
Epoch: 0004 LOSS_TRAIN: 1.0672 ACC_TRAIN: 0.5167 LOSS_VAL: 1.0745 ACC_VAL: 0.7120 time: 1.8692s
...
Epoch: 0159 LOSS_TRAIN: 0.5489 ACC_TRAIN: 0.9500 LOSS_VAL: 0.7034 ACC_VAL: 0.8040 time: 1.8088s
Epoch: 0160 LOSS_TRAIN: 0.4688 ACC_TRAIN: 0.9167 LOSS_VAL: 0.7037 ACC_VAL: 0.8060 time: 1.7993s
Epoch: 0161 LOSS_TRAIN: 0.5392 ACC_TRAIN: 0.8833 LOSS_VAL: 0.7041 ACC_VAL: 0.8020 time: 1.8251s
...
Epoch: 0360 LOSS_TRAIN: 0.4690 ACC_TRAIN: 0.9333 LOSS_VAL: 0.7113 ACC_VAL: 0.7960 time: 1.8726s
Epoch: 0361 LOSS_TRAIN: 0.5582 ACC_TRAIN: 0.9167 LOSS_VAL: 0.7102 ACC_VAL: 0.7960 time: 1.8247s
Epoch: 0362 LOSS_TRAIN: 0.4936 ACC_TRAIN: 0.9500 LOSS_VAL: 0.7091 ACC_VAL: 0.7960 time: 1.7927s
Epoch: 0363 LOSS_TRAIN: 0.4971 ACC_TRAIN: 0.9333 LOSS_VAL: 0.7083 ACC_VAL: 0.7960 time: 1.8243s
```

Third model:

```
model_3,indices_3,loss_train_3,acc_train_3,loss_values_3,
        acc_values_3 = train_early_stop(model_3,data,patience=100,n_epochs=500)
```

```
Epoch: 0001 LOSS_TRAIN: 1.1307 ACC_TRAIN: 0.3000 LOSS_VAL: 1.0914 ACC_VAL: 0.5280 time: 1.9290s
Epoch: 0002 LOSS_TRAIN: 1.1277 ACC_TRAIN: 0.2667 LOSS_VAL: 1.0892 ACC_VAL: 0.5640 time: 1.9111s
Epoch: 0003 LOSS_TRAIN: 1.1033 ACC_TRAIN: 0.3833 LOSS_VAL: 1.0873 ACC_VAL: 0.6060 time: 1.8820s
Epoch: 0004 LOSS_TRAIN: 1.0857 ACC_TRAIN: 0.3667 LOSS_VAL: 1.0855 ACC_VAL: 0.6440 time: 1.8881s
...
Epoch: 0207 LOSS_TRAIN: 0.4724 ACC_TRAIN: 0.9500 LOSS_VAL: 0.6468 ACC_VAL: 0.8000 time: 1.9114s
Epoch: 0208 LOSS_TRAIN: 0.4442 ACC_TRAIN: 0.9000 LOSS_VAL: 0.6453 ACC_VAL: 0.7960 time: 1.9006s
Epoch: 0209 LOSS_TRAIN: 0.4512 ACC_TRAIN: 0.9167 LOSS_VAL: 0.6450 ACC_VAL: 0.7980 time: 1.9274s
...
Epoch: 0416 LOSS_TRAIN: 0.3903 ACC_TRAIN: 0.9333 LOSS_VAL: 0.6318 ACC_VAL: 0.7960 time: 1.8331s
Epoch: 0417 LOSS_TRAIN: 0.5126 ACC_TRAIN: 0.8500 LOSS_VAL: 0.6327 ACC_VAL: 0.7940 time: 1.8266s
Epoch: 0418 LOSS_TRAIN: 0.4650 ACC_TRAIN: 0.9167 LOSS_VAL: 0.6336 ACC_VAL: 0.7980 time: 1.8294s
Epoch: 0419 LOSS_TRAIN: 0.4434 ACC_TRAIN: 0.9333 LOSS_VAL: 0.6340 ACC_VAL: 0.7980 time: 1.8440s
```

5.3 Models evaluation and comparison

Comparison of models mainly deals with analyzing their capability to generalize and with their average accuracy on the test set.

5.3.1 Cora

To increase the generalization capabilty of a model and decrease overfitting, we want a small gap between training and validation loss, and between train and validation accuracy. The graphical visualization of the histories of the training process helps to analyze this aspect. For this purpose, we defined a function, using `matplotlib`, to show for each model the validation and training losses and accuracies during epochs:

```
def plot_val_loss_acc(indices,arg1,arg2):
  plt.plot(indices,arg1,label='validation')
  plt.plot(indices,arg2,label='train')
  plt.xlabel('Epoch')
  plt.grid(True)
  plt.legend()
  plt.show()
```

Figure 7, 8, 9 represent the output of the following code:

```
#model 1
plt.title('Model 1: Train and validation losses')
plot_val_loss_acc(indices_1, loss_values_1, loss_train_1)

plt.title('Model 1: Train and validation accuracy')
plot_val_loss_acc(indices_1, acc_values_1, acc_train_1)

#model 2
plt.title('Model 2: Train and validation losses')
plot_val_loss_acc(indices_2, loss_values_2, loss_train_2)

plt.title('Model 2: Train and validation accuracy')
plot_val_loss_acc(indices_2, acc_values_2, acc_train_2)

#model 3
plt.title('Model 3: Train and validation losses')
```

```
plot_val_loss_acc(indices_3, loss_values_3, loss_train_3)

plt.title('Model 3: Train and validation accuracy')
plot_val_loss_acc(indices_3, acc_values_3, acc_train_3)
```

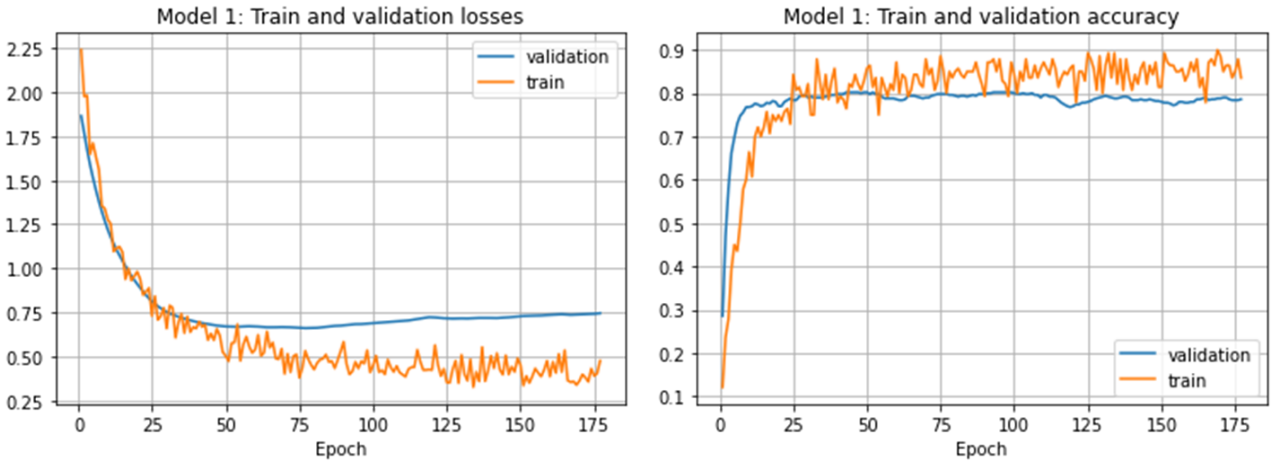

Figure 7: Train and validation loss and accuracy (model 1)

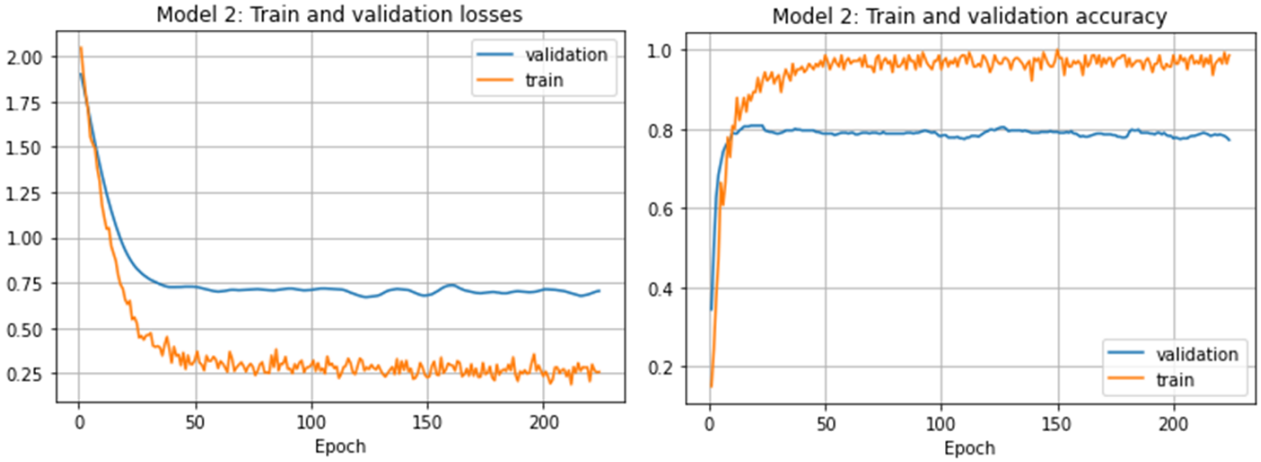

Figure 8: Train and validation loss and accuracy (model 2)

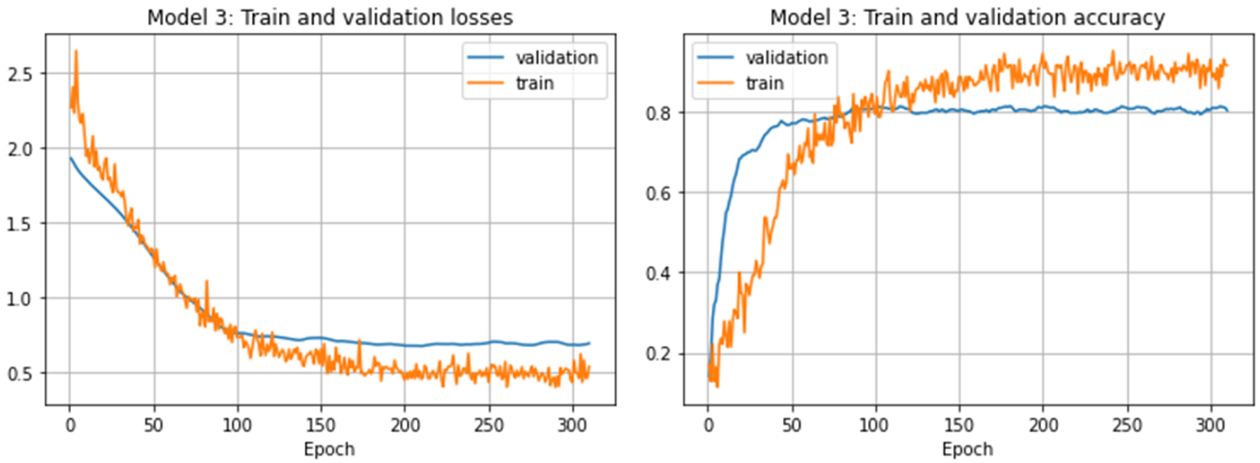

Figure 9: Train and validation loss and accuracy (model 3)

From the plotted charts it's easy to notice that in the second model the gap between training and validation values is larger compared to the other two models.

We can also compare separately the validation losses and the validation accuracies of the models.

For losses (output in Figure 10):

```
plt.plot(indices_1, loss_values_1, label='model 1')
plt.plot(indices_2, loss_values_2, label='model 2')
plt.plot(indices_3, loss_values_3, label='model 3')

plt.ylabel('Validation loss')
plt.xlabel('Epoch')
plt.grid(True)
plt.legend()
plt.show()
```

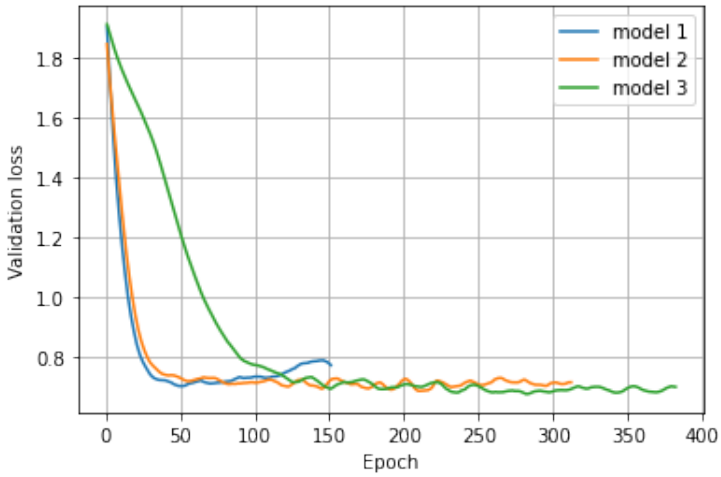

Figure 10: Comparing validation losses of the three models

For accuracies (output in Figure 11):

```
plt.plot(indices_1, acc_values_1, label='model 1')
plt.plot(indices_2, acc_values_2, label='model 2')
plt.plot(indices_3, acc_values_3, label='model 3')

plt.ylabel('Validation accuracy')
plt.xlabel('Epoch')
plt.grid(True)
plt.legend()
plt.show()
```

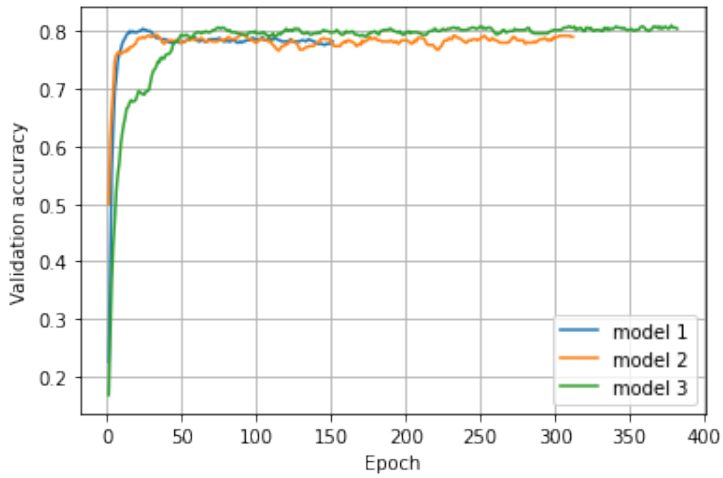

Figure 11: Comparing validation accuracies of the three models

Finally, we evaluated the models on the test set, using the `evaluate()` function,

```
def evaluate(model, data, i):
  model.eval()
  pred = model(data)
  acc = accuracy(pred[data.test_mask], data.y[data.test_mask])
  print('Accuracy (model %d): {:.4f}'.format(acc) %(i))
```

and calling it on the three models:

```
evaluate(model_1, data, 1)
evaluate(model_2, data, 2)
evaluate(model_3, data, 3)
```

```
Accuracy (model 1): 0.7910
Accuracy (model 2): 0.8190
Accuracy (model 3): 0.8020
```

Consider also that, running the experiment many times, small changes can occur in accuracy values.
According to the evaluation on the test set, the best model for Cora seems to be `model_2`: increasing the width of layers, increasing the number of parameters (with higher hidden size and heads per layer) helped the network to memorize better. However, we had also to increase the regularization through higher dropout rate and weight decay to make sure the network does not just memorize the training set and overfit it.

5.3.2 Pubmed

We followed exactly the same steps for the experimentation on Pubmed. Since the code is the same as the one used for Cora, we will just show the results obtained training each model (Figure 12, 13, 14).

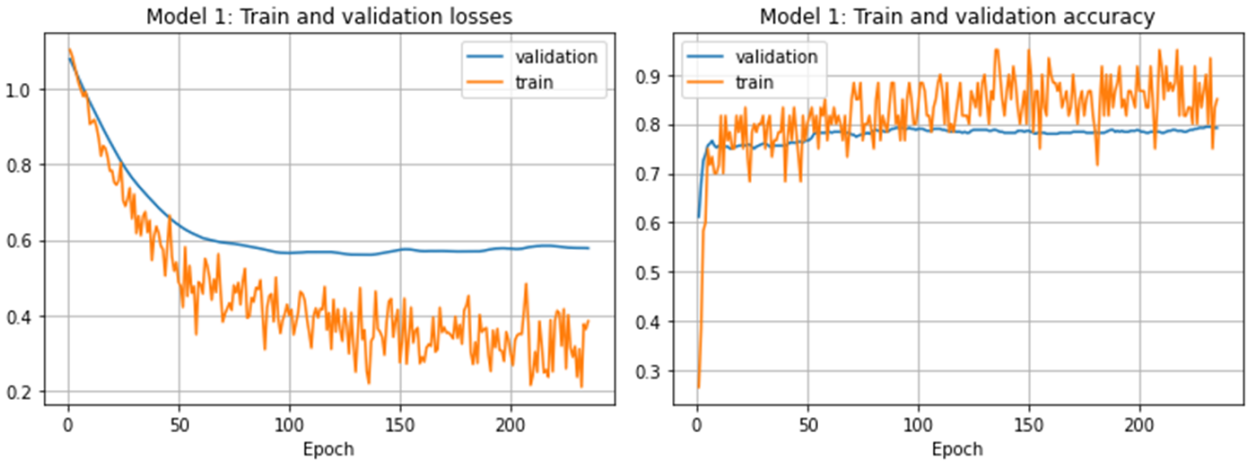

Figure 12: Train and validation loss and accuracy (model 1)

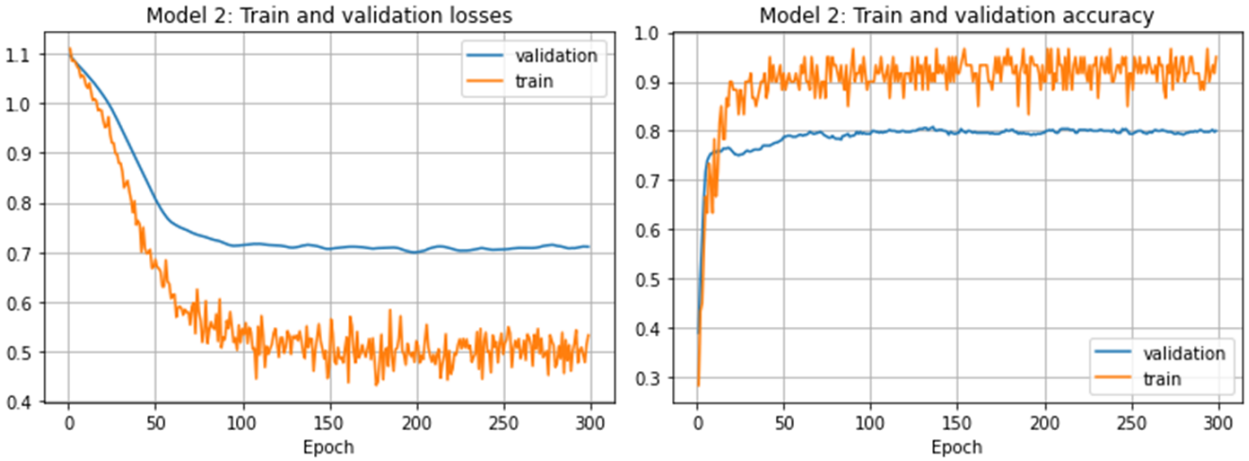

Figure 13: Train and validation loss and accuracy (model 2)

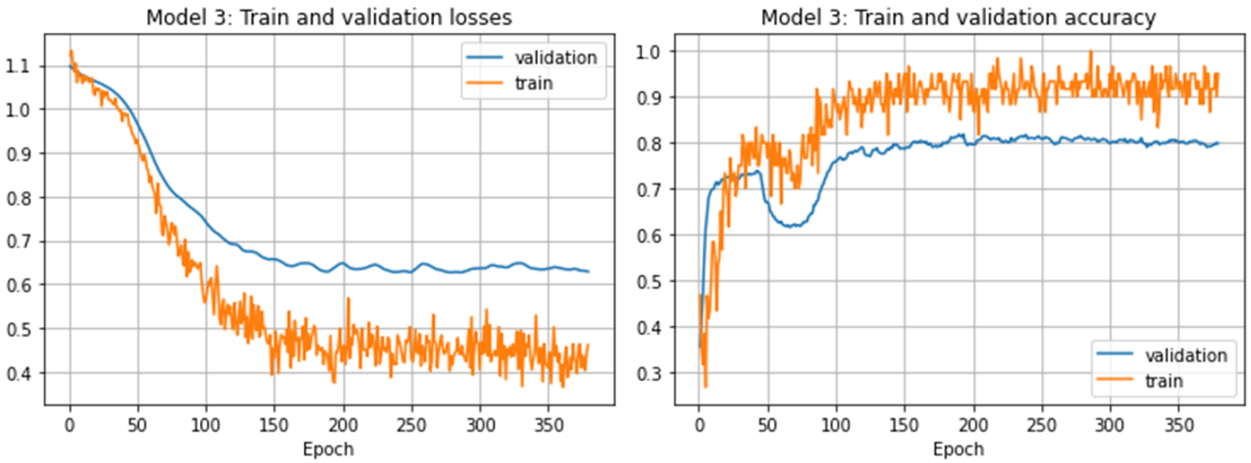

Figure 14: Train and validation loss and accuracy (model 3)

Comparing losses (Figure 15) and accuracies (Figure 16) during training, it is easy to notice that the first and the second model require few epochs to converge, while the third becomes stable later.

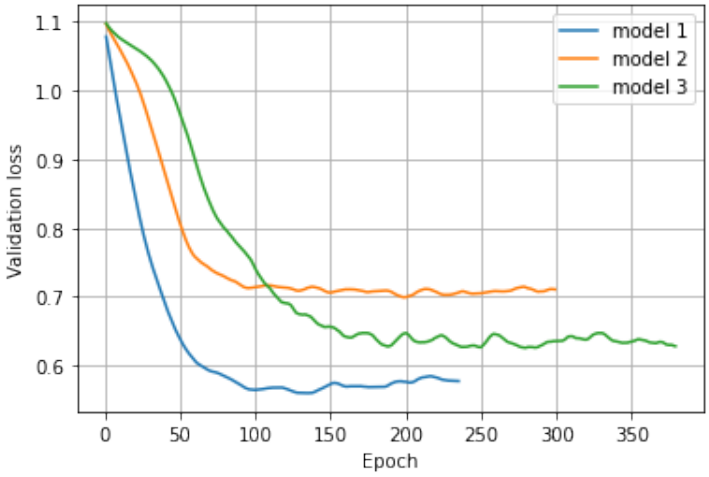

Figure 15: Comparing validation losses of the three models

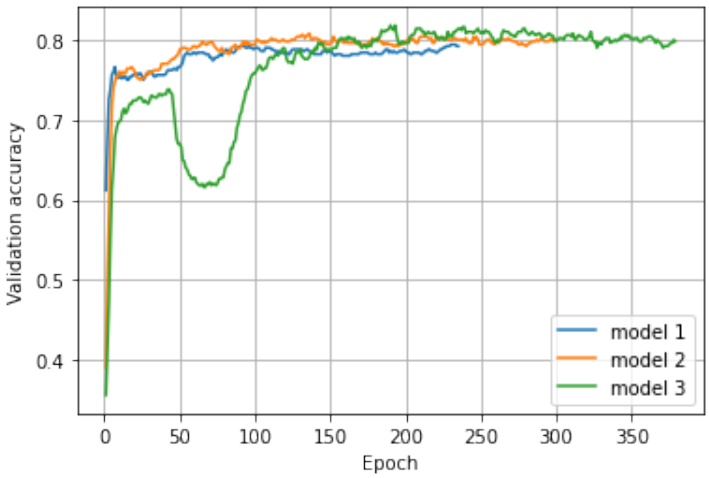

Figure 16: Comparing validation accuracies of the three models

The predictions on Pubmed test set using the three models gave us the following results:

```
Accuracy (model 1): 0.7760
Accuracy (model 2): 0.7750
Accuracy (model 3): 0.7850
```

Regarding the test set evaluation, we see very similar results between the three models. For this dataset, the best model seems to be `model_3`, meaning that increasing the depth (three GAT layers instead of two) in this particular case has given better results than increasing the width.

6 Conclusions

Eventhough our experimentation with GAT networks almost faithfully followed the suggestions of the paper, expecially regarding the design of the network and the choice of the datasets, training a neural network is always **challenging** and **experimental**. In fact, there is a huge amount of things affecting the speed of convergence and the performance (in terms of accuracy) of a model: not only the architecture of the network (the depth, the width, the kind of layers, the activations), but also the learning algorithm and the optimizer, the loss function, the weight initialization, the dataset and the complexity of the problem (the size of training/validation/test sets, the feature extraction and selection processes) and all the other hyperparameters, like regularization terms (dropout, weight decay) and learning rate.
The challenge is to find a trade-off in the definition of hyperparameters, trying to achieve a good generalization, a satisfactory accuracy, and quite fast convergence in training. As known, there is no general rule to decide which are the best values for such hyperparameters, therefore the only way to know it is to do experiments.

The three models that we have considered and compared in this report are not the only ones that have been tried. We also experimented different learning rates, finding out that 0.005 was a good trade-off, because having a smaller one gave us the same results but required more epochs to converge.
We noticed that, in terms of regularizations, the dropout has a more powerful effect on the training process than the weight decay: we tried a 0.6 (model 1 and 2), 0.7 (model 3) and 0.8 dropout rate, but with 0.8 we started to underfit the training set, getting better accuracies and losses on the validation set than in the training set. Anyway, a kind of regularization was strictly required since we chose only 20 samples for each class as training set, as suggested by the paper.
Following two main strategies to modify the models, we first increased the width and then the depth, but the outcomes depended not only on the overall architecture but also on the datasets and the features representation. In fact for Cora it was best to have a wider model, while in Pubmed it was best to have a deeper one, such that having one more layer maybe helps the network to learn features at various levels of abstractions; the reason behind this difference is maybe that for Pubmed the features hold more information (TF-IDF coefficients) than Cora (just binary variables in Bag of Words representation), therefore it was more reasonable to learn intermediate features in Pubmed dataset than in Cora.

However, during this experimentation we could understand on the one hand the **power of the Attention mechanism** for node classification in graphs, because with only two or three layers and small training sets we achieved an average accuracy of 80% (Cora) and 78% (Pubmed), and on the other hand the **benefits of some python libraries to perform graph learning**, such as PyTorch Geometric, that allow to easily manage graph-structured data as multi-dimensional tensors and build neural networks based on the aggregation of neighborhood information with the Message Passing scheme.