Transductive Node Classification
using Graph Attention Neural Networks

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Alessandra Monaco (1706205)

Neural Networks course
ay 2019-2020
Engineering in Computer Science

TABLE OF CONTENTS

1 INTRODUCTION

2 ENVIRONMENT AND TOOLS

3 DATASETS AND TASK

4 NEURAL NETWORKS ARCHITECTURE AND IMPLEMENTATION

5 EXPERIMENTAL RESULTS

6 CONCLUSIONS

1 INTRODUCTION

Reference paper

TITLE:

Graph Attention Networks

AUTHORS:

Petar Velickovic
Guillem Cucurull
Arantxa Casanova
Adriana Romero
Pietro Lio
Yoshua Bengio

https://arxiv.org/abs/1710.10903

Published as a conference paper at ICLR 2018

arXiv:1710.10903v3 [stat.ML] 4 Feb 2018

GRAPH ATTENTION NETWORKS

Petar Veličković*
Department of Computer Science and Technology
University of Cambridge
petar.velickovic@cst.cam.ac.uk

Guillem Cucurull*
Centre de Visió per Computador, UAB
gcucurull@gmail.com

Arantxa Casanova*
Centre de Visió per Computador, UAB
ar.casanova.8@gmail.com

Adriana Romero
Montréal Institute for Learning Algorithms
adriana.romero.soriano@umontreal.ca

Pietro Liò
Department of Computer Science and Technology
University of Cambridge
pietro.lio@cst.cam.ac.uk

Yoshua Bengio
Montréal Institute for Learning Algorithms
yoshua.umontreal@gmail.com

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-the-art results across four established transductive and inductive graph benchmarks: the *Cora*, *Citeseer* and *Pubmed* citation network datasets, as well as a *protein-protein interaction* dataset (wherein test graphs remain unseen during training).

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have been successfully applied to tackle problems such as image classification (He et al., 2016), semantic segmentation (Jégou et al., 2017) or machine translation (Gehring et al., 2016), where the underlying data representation has a grid-like structure. These architectures efficiently reuse their local filters, with learnable parameters, by applying them to all the input positions.

However, many interesting tasks involve data that can not be represented in a grid-like structure and that instead lies in an irregular domain. This is the case of 3D meshes, social networks, telecommunication networks, biological networks or brain connectomes. Such data can usually be represented in the form of graphs.

There have been several attempts in the literature to extend neural networks to deal with arbitrarily structured graphs. Early work used recursive neural networks to process data represented in graph domains as directed acyclic graphs (Frasconi et al., 1998; Sperduti & Starita, 1997). Graph Neural Networks (GNNs) were introduced in Gori et al. (2005) and Scarselli et al. (2009) as a generalization of recursive neural networks that can directly deal with a more general class of graphs, e.g. cyclic, directed and undirected graphs. GNNs consist of an iterative process, which propagates the node states until equilibrium; followed by a neural network, which produces an output for each node

*Work performed while the author was at the Montréal Institute of Learning Algorithms.

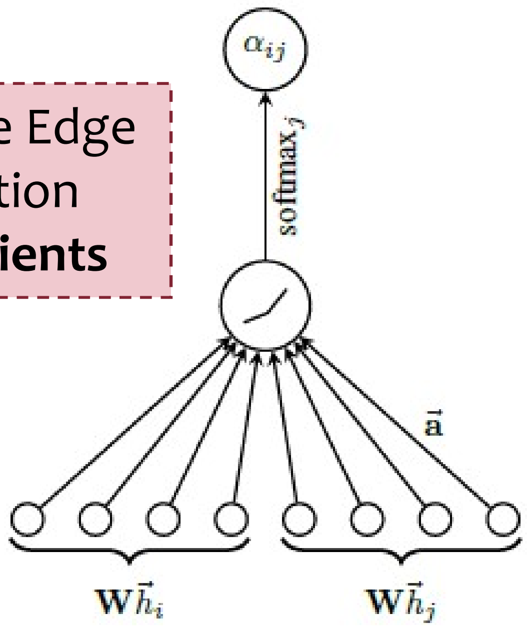
1

1 INTRODUCTION
Main topic

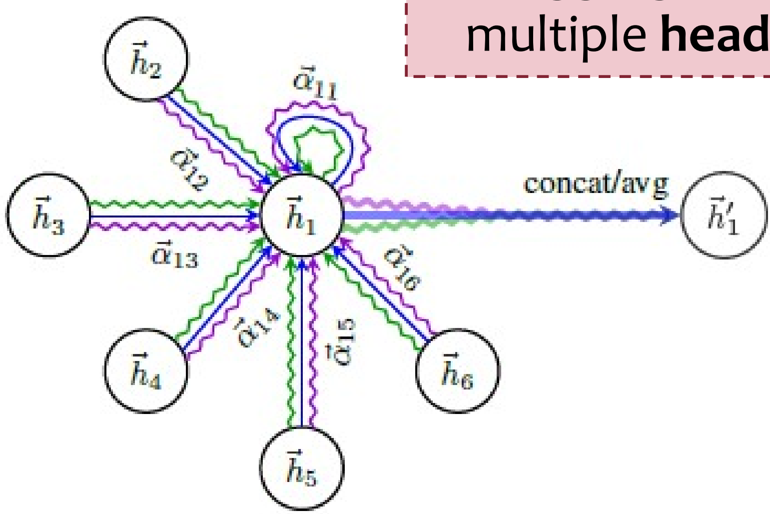
GOAL:

Compute Graph convolutions using ***Multi-head Self-Attention Mechanism*** *(non-spectral approach)*

Compute Edge Attention **Coefficients**



Aggregate multiple **heads**



2 ENVIRONMENT AND TOOLS

Platform

colab

Programming language

python

NN libraries

PyTorch

PyTorch geometric

Visualization tools

NetworkX

matplotlib

3 DATASETS AND TASK
Overview

OUR TASK:

Predict classes for unlabelled nodes in a *Transductive setting*.

	Cora	Citeseer	Pubmed	PPI
Task	Transductive	Transductive	Transductive	Inductive
# Nodes	2708 (1 graph)	3327 (1 graph)	19717 (1 graph)	56944 (24 graphs)
# Edges	5429	4732	44338	818716
# Features/Node	1433	3703	500	50
# Classes	7	6	3	121 (multilabel)
# Training Nodes	140	120	60	44906 (20 graphs)
# Validation Nodes	500	500	500	6514 (2 graphs)
# Test Nodes	1000	1000	1000	5524 (2 graphs)

3 DATASETS AND TASK
Importing datasets

```
from torch_geometric.datasets import Planetoid

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

```
Data(edge_index=[2, 10556], test_mask=[2708], train_mask=[2708],
     val_mask=[2708], x=[2708, 1433], y=[2708])
```

Features:
Binary vectorizer in Bag of Words Representation

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

```
Data(edge_index=[2, 88648], test_mask=[19717], train_mask=[19717],
     val_mask=[19717], x=[19717, 500], y=[19717])
```

Features:
TF-IDF coefficients

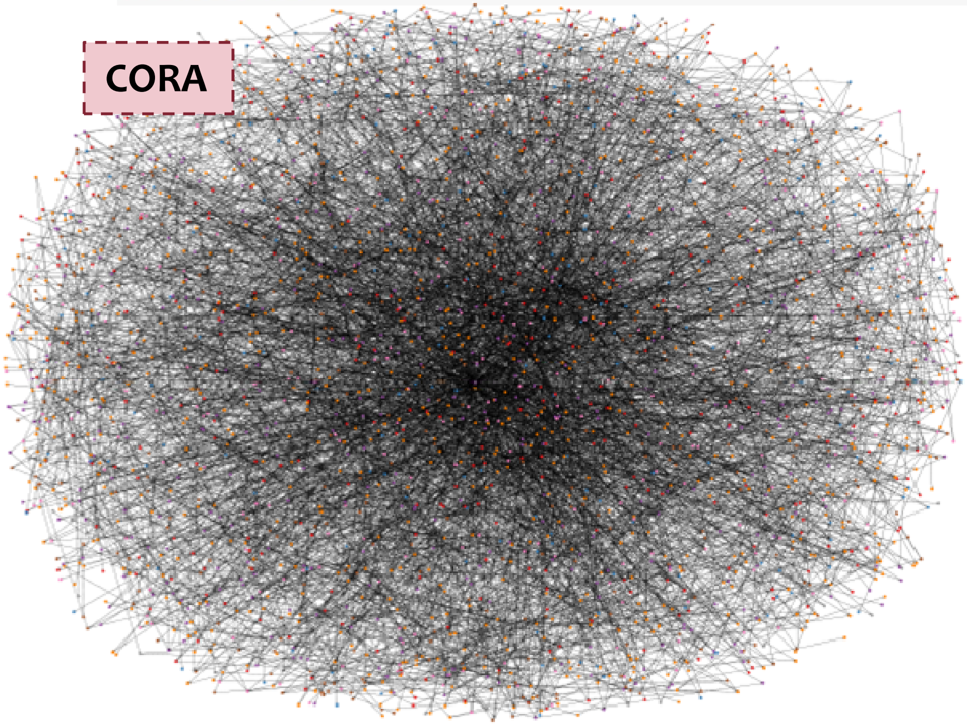
```
#Add self loops
data.edge_index, _ =add_self_loops(edge_index=data.edge_index)
```

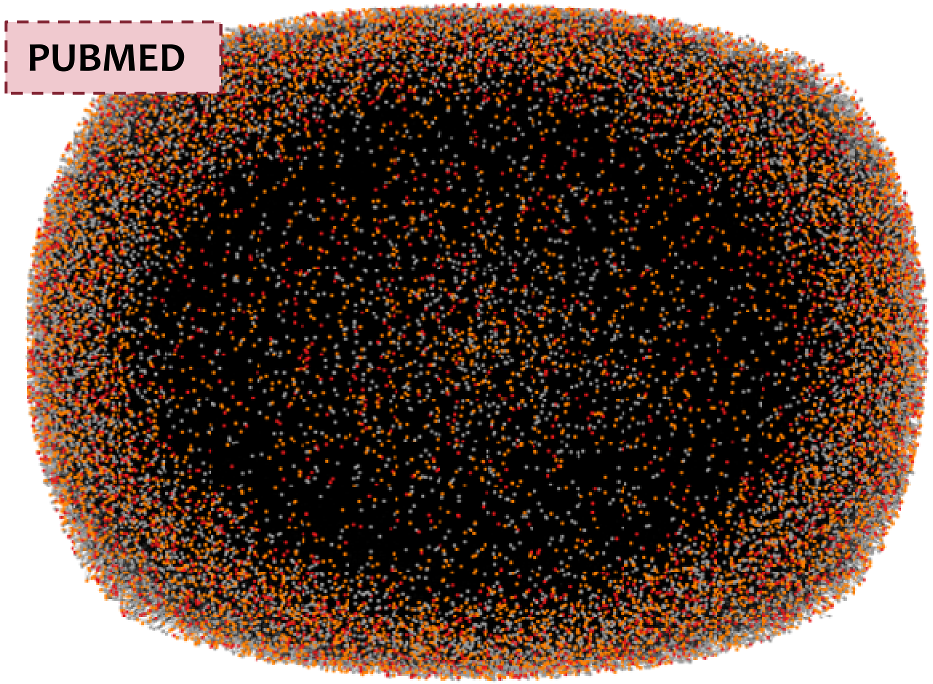
3

DATASETS AND TASK

Visualizing datasets

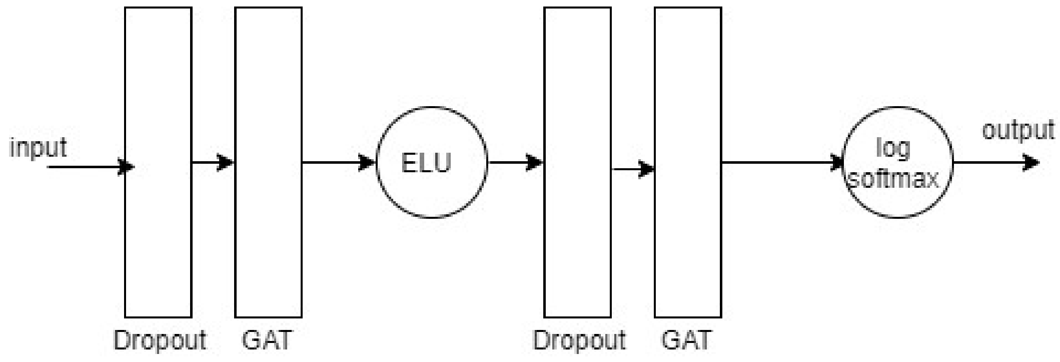
```
from torch_geometric.utils.convert import to_networkx

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()
```



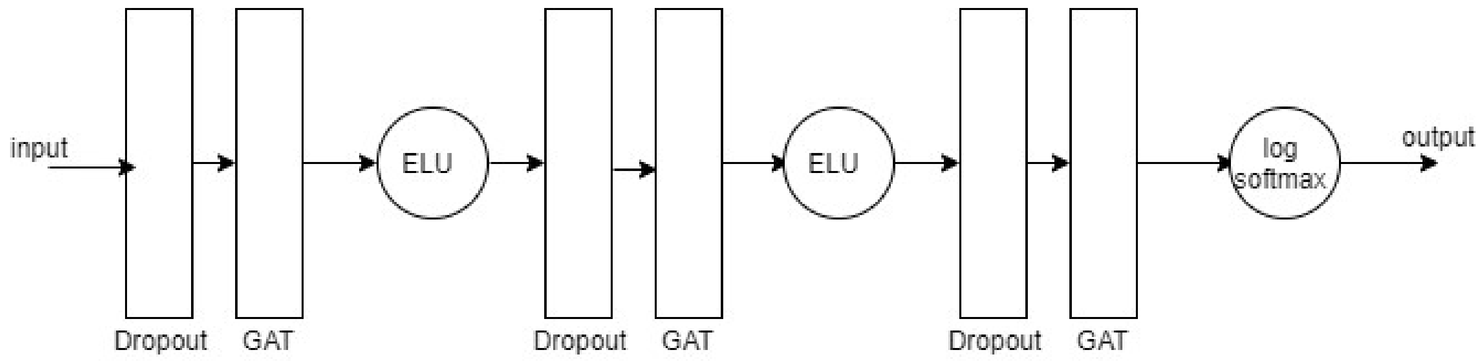


4 NN ARCHITECTURE AND IMPLEMENTATION
Overall Architecture

2-LAYERS GAT NETWORK



3-LAYERS GAT NETWORK



4 NN ARCHITECTURE AND IMPLEMENTATION
The GAT Layer (1/3)

```
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

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

    nn.init.xavier_uniform_(self.W.data, gain=1.414)
    nn.init.xavier_uniform_(self.a.data, gain=1.414)

  def forward(self, x, edge_index, size=None):
    x = torch.matmul(x, self.W)
    return self.propagate(edge_index, size=size, x=x)
```

Neighborhood Aggregation scheme

Trainable parameters

Multi-head attention mechanism

calls:

- message()
- aggregate()
- update()

4 NN ARCHITECTURE AND IMPLEMENTATION

The GAT Layer (2/3)

```
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

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

    nn.init.xavier_uniform_(self.W.data, gain=1.414)
    nn.init.xavier_uniform_(self.a.data, gain=1.414)

  def forward(self, x, edge_index, size=None):
    x = torch.matmul(x, self.W)
    return self.propagate(edge_index, size=size, x=x)
```

aggregate() sums the messages of all neighbors

calls:

- message()
- aggregate()
- update()

4 NN ARCHITECTURE AND IMPLEMENTATION
The GAT Layer (3/3)

message() returns the information to be propagated to neighbors

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

  x_j = x_j.view(-1, self.heads, self.output_size)
  x_i = x_i.view(-1, self.heads, self.output_size)

  alpha = (torch.cat([x_i, x_j], dim=-1) * self.a).sum(dim=-1)
  alpha = F.leaky_relu(alpha, self.negative_slope)
  alpha = softmax(alpha, edge_index_i, size_i)
  alpha = F.dropout(alpha, p=self.dropout, training=self.training)

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

Edge self-attention coefficient

$$\alpha_{ij} = \frac{exp(LeakyReLU(a^{T}(W \cdot h_{i} || W \cdot h_{j})))}{\sum_{k \epsilon N} exp(LeakyReLU(a^{T}(W \cdot h_{i} || W \cdot h_{k})))}$$

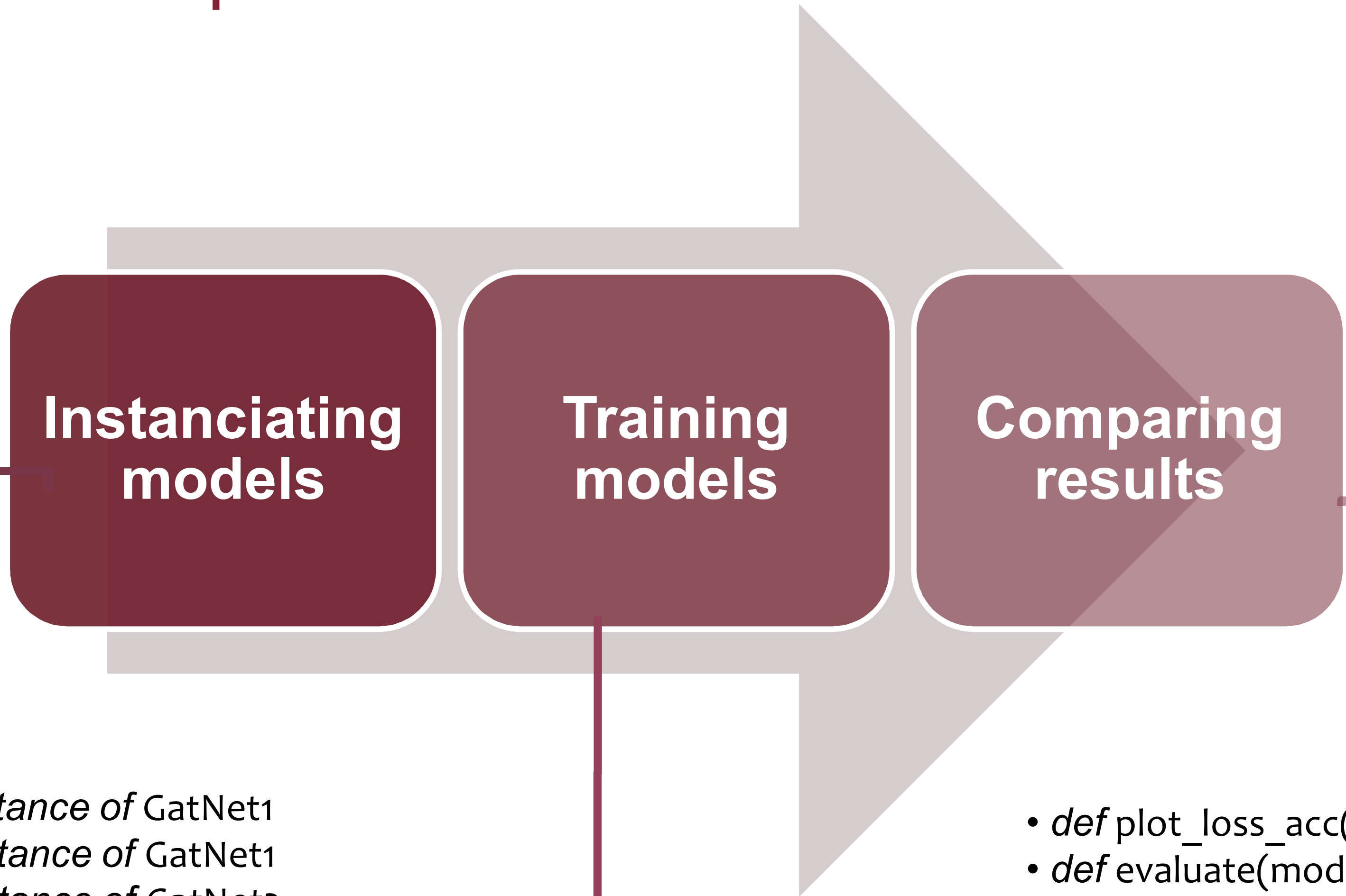
update() modifies features according to neighbors information

```
def update(self, out):

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

  return out
```

self.concat is 'False' only in the last GAT layer of the NN

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{k} \mathbf{W}^{k} \vec{h}_j\right)$$

5 EXPERIMENTAL RESULTS
Main steps



- model_1 *instance of* GatNet1
- model_2 *instance of* GatNet1
- model_3 *instance of* GatNet2

- *def* accuracy(output,labels)
- *def* train_early_stop(model,data,patience,n_epochs)

- *def* plot_loss_acc(indices,val,train)
- *def* evaluate(model,data,n)

5 EXPERIMENTAL RESULTS
Instanciating models

Hyperparameters tuning

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

Model 1

```
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)
```

Model 2: *Increasing width and regularization*

```
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)
```

Model 3: *Increasing depth and regularization*

```
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)
```

5 EXPERIMENTAL RESULTS

Training models

100

500

Initializing needed variables

Printing epoch's information

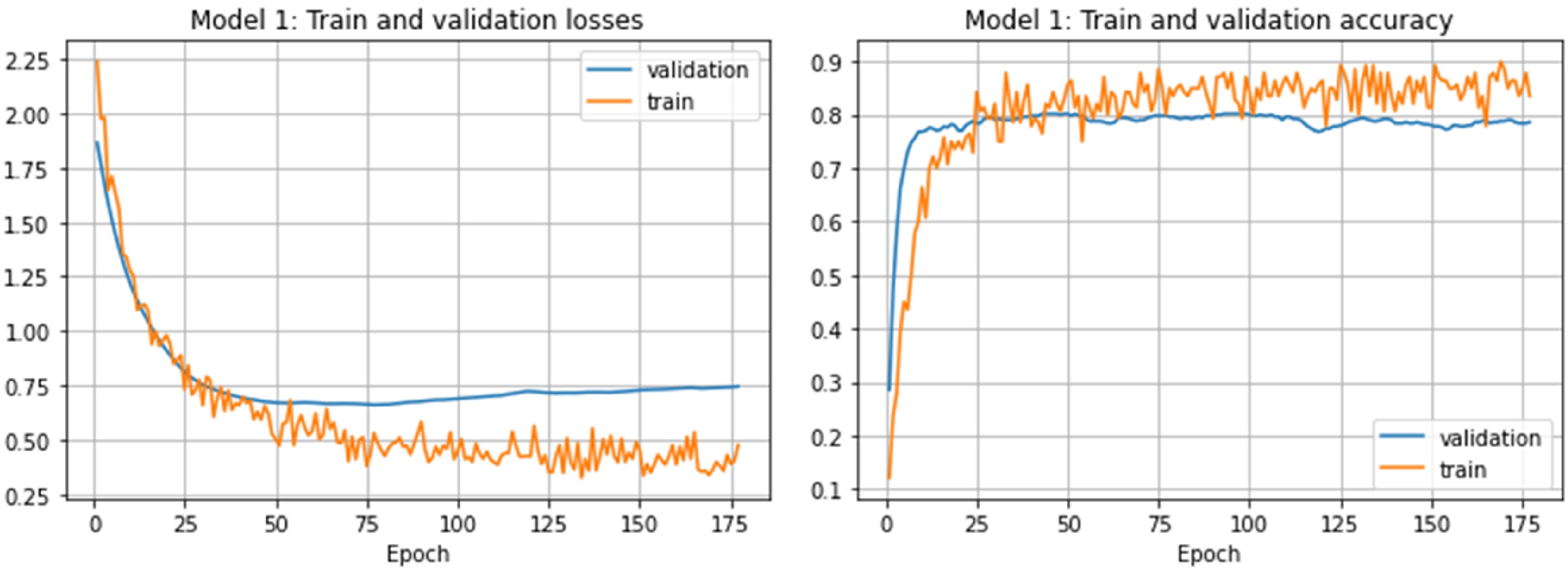
Negative log-likelihood loss

Early Stopping

```
def train_early_stop(model, data, patience, 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()
    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])
    ...
    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
```

5 EXPERIMENTAL RESULTS
Training histories

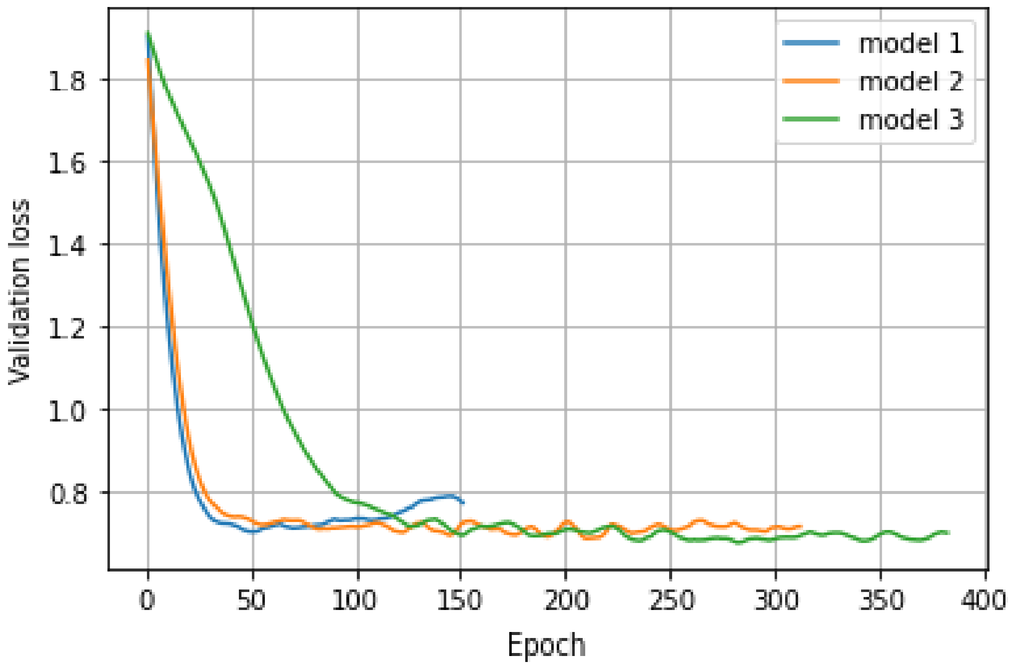
Training history of model 1 [Cora]

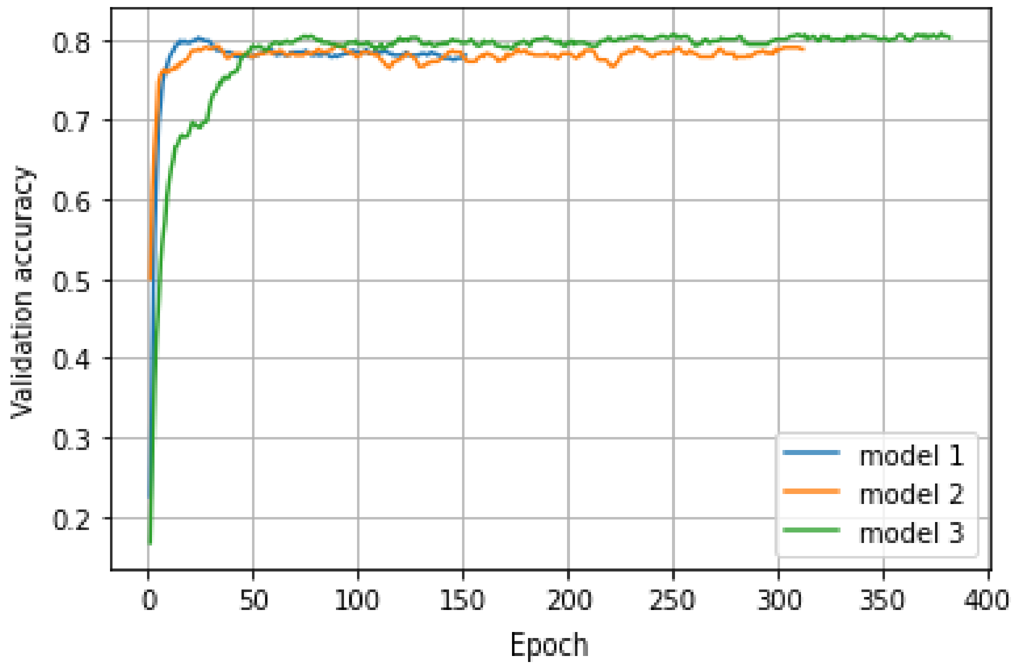



```
Epoch: 0001 LOSS_TRAIN: 2.0793 ACC_TRAIN: 0.1857 LOSS_VAL: 1.8403 ACC_VAL: 0.2720 time: 0.3409s
Epoch: 0002 LOSS_TRAIN: 2.0081 ACC_TRAIN: 0.2214 LOSS_VAL: 1.7555 ACC_VAL: 0.3860 time: 0.1376s
...
Epoch: 0165 LOSS_TRAIN: 0.2917 ACC_TRAIN: 0.8786 LOSS_VAL: 0.7643 ACC_VAL: 0.7900 time: 0.1375s
Epoch: 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
```

EXPERIMENTAL RESULTS
Comparing validation metrics [Cora]

CORA results



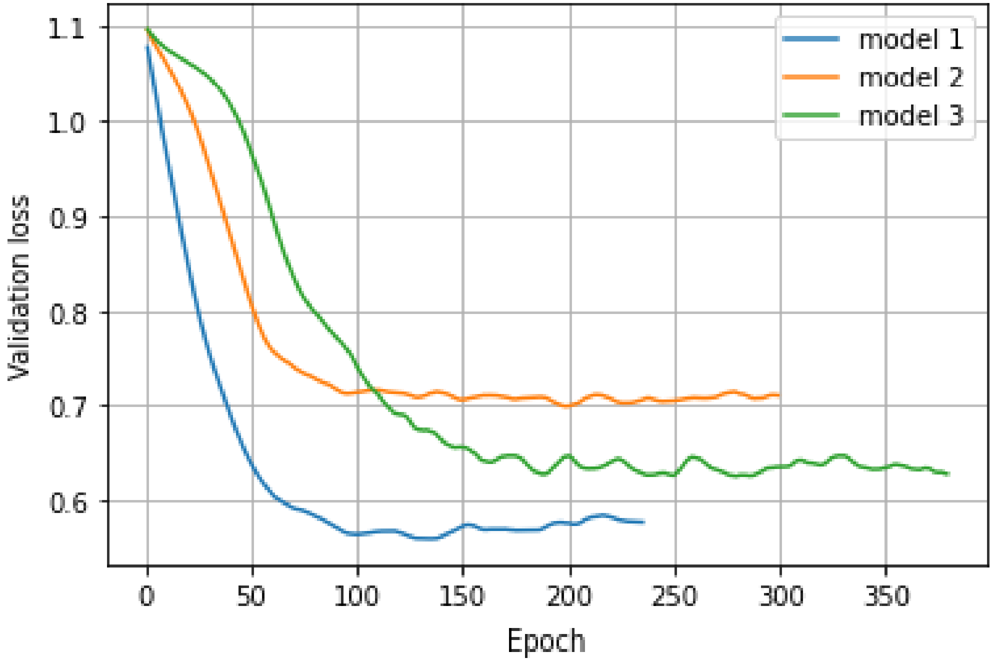


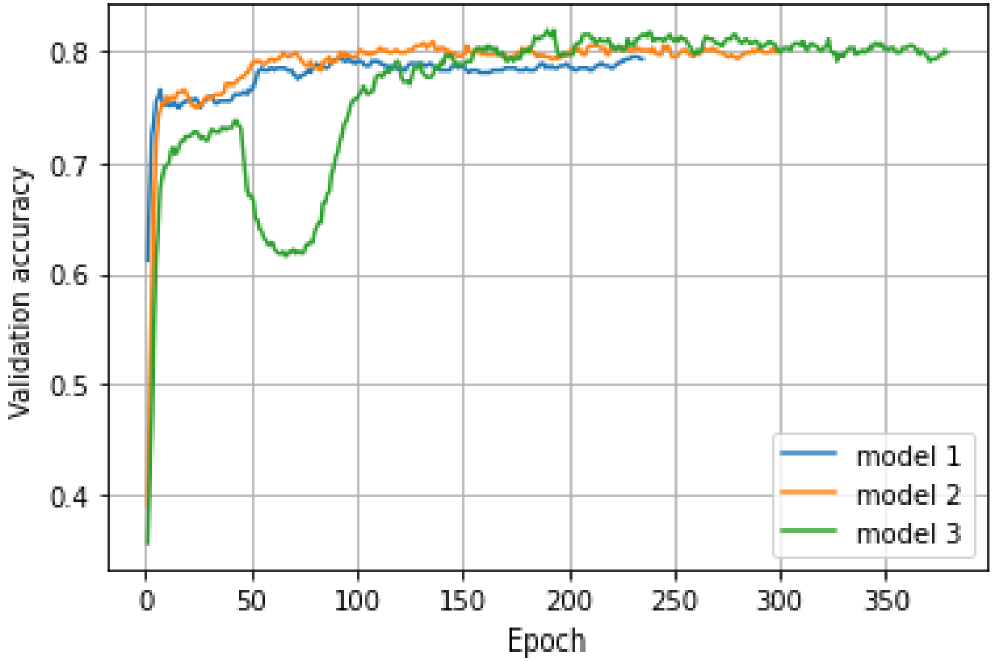

```
[ ]  evaluate(model_1, data, 1)
     evaluate(model_2, data, 2)
     evaluate(model_3, data, 3)
```

```
Accuracy (model 1): 0.7960
Accuracy (model 2): 0.8140
Accuracy (model 3): 0.8020
```

5 EXPERIMENTAL RESULTS
Comparing validation metrics [Pubmed]

PUBMED results






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

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

6 CONCLUSIONS

- Building and training a NN is always **challenging** and **experimental**
- GOAL = Find **trade-off** between *good generalization*, *convergence speed* and *accuracy*

- Tuning hyperparameters we noticed that:
  - ➢ the outcomes of increasing *width* and *depth* of the NN depend also on the dataset
  - ➢ the dropout has a more powerful regularization effect than weight decay
  - ➢ 0.005 was the best learning rate choice for this situation

connections hold lots of information in graph-structured data, and GAT networks are **efficient** and **powerful** strategies to exploit it

Thanks for the attention

Alessandra Monaco