✨ Member-only story

Graph Attention Networks: Self-Attention Explained

A guide to GNNs with self-attention using PyTorch Geometric

Maxime Labonne · Follow

Published in Towards Data Science · 9 min read · Apr 17, 2022

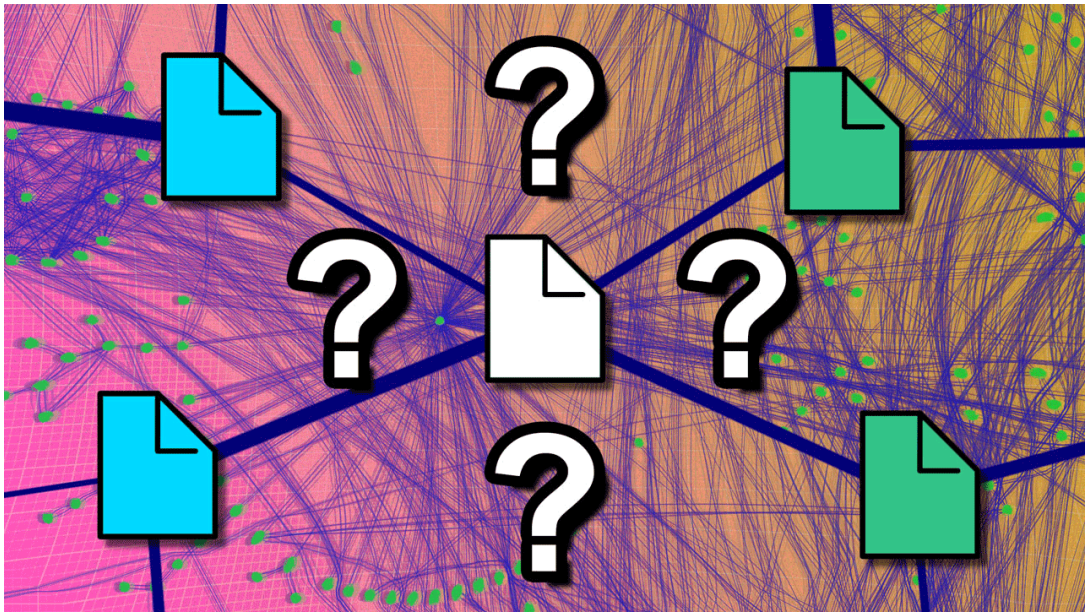
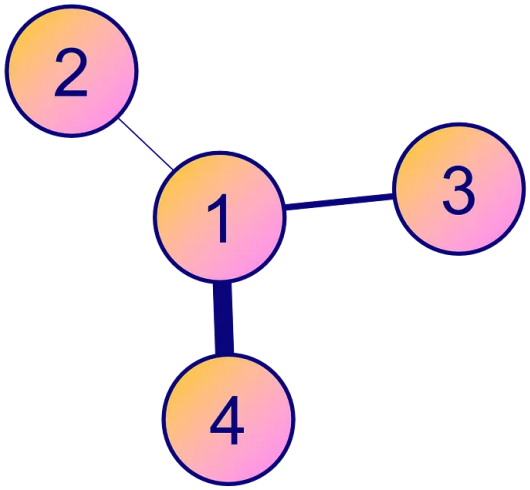


Image by author, file icon by OpenMoji (CC BY-SA 4.0)

Graph Attention Networks are **one of the most popular types** of Graph Neural Networks. For a good reason.

With Graph *Convolutional* Networks (GCN), every neighbor has the **same importance**. Obviously, it should not be the case: some nodes are more essential than others.

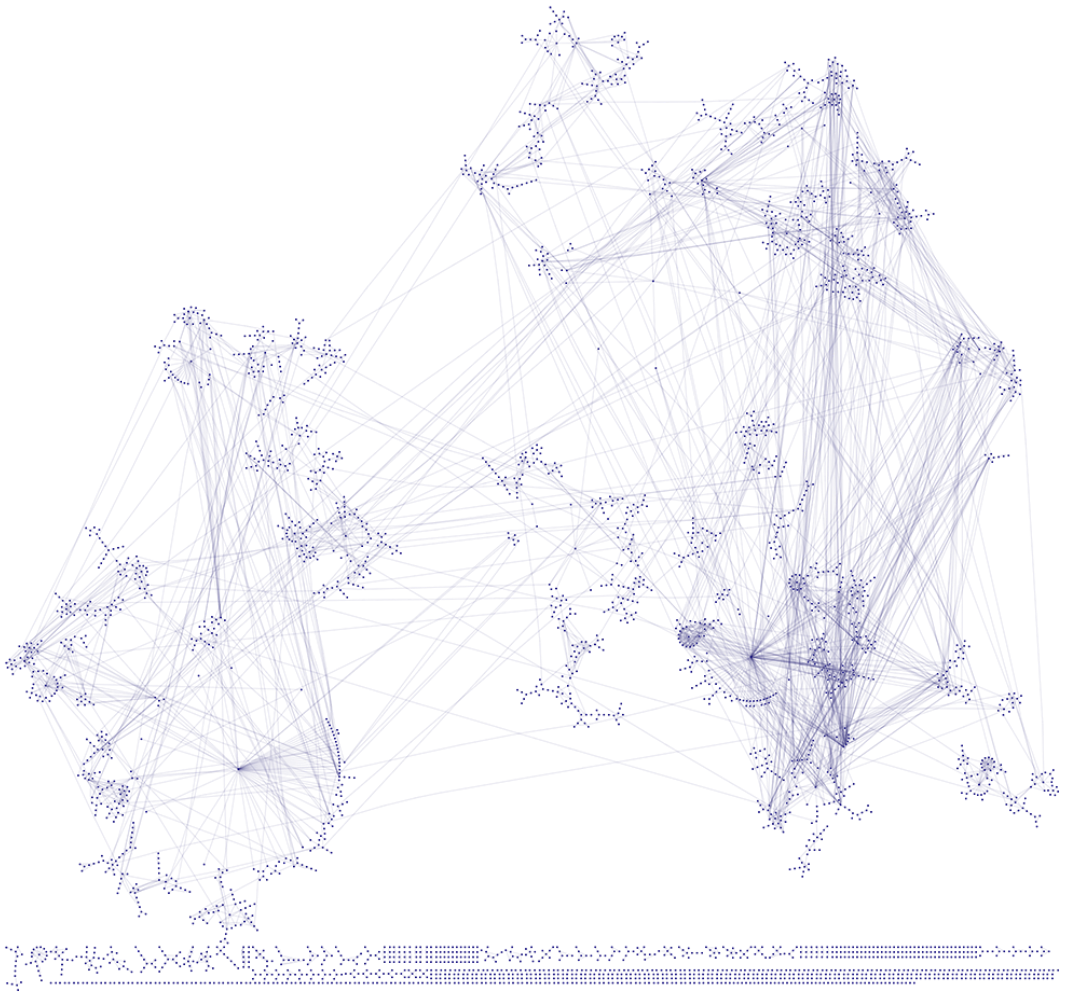


Node 4 is more important than node 3, which is more important than node 2 (image by author)

Graph *Attention* Networks offer a solution to this problem. To consider the importance of each neighbor, an attention mechanism assigns a **weighting factor to every connection**.

In this article, we'll see how to **calculate** these attention scores and **implement** an efficient GAT in PyTorch Geometric (PyG). You can run the code of this tutorial with the following Google Colab notebook.

I. Graph data



CiteSeer dataset (image by author, made with yEd Live)

There are three classic graph datasets we can use for this work (MIT license). They represent networks of research papers, where each connection is a citation.

- Cora: it consists of 2708 machine learning papers that belong to one of 7 categories.

➡️ Node features represent the presence (1) or absence (0) of 1433 words in a paper (binary bag of words).

- **CiteSeer**: it is a bigger but similar dataset of 3312 scientific papers to classify into one of 6 categories.
  ➡️ Node features represent the presence (1) or absence (0) of 3703 words in a paper.
- **PubMed**: it is an even bigger dataset with 19717 scientific publications about diabetes from PubMed's database, classified into 3 categories.
  ➡️ Node features are TF-IDF weighted word vectors from a dictionary of 500 unique words.

These datasets have been widely used by the scientific community. As a challenge, we can compare our accuracy scores to those obtained in the literature using **Multilayer Perceptrons** (MLPs), **GCNs**, and **GATs**:

Dataset	📝 Cora	📝 CiteSeer	📝 PubMed
MLP	55.1%	46.5%	71.4%
GCN	81.5%	70.3%	**79.0%**
GAT	**83.0%**	**72.5%**	**79.0%**

PubMed is quite large so it would take longer to process it and train a GNN on it. Cora is the most studied one in the literature, so let's **focus on CiteSeer** as a middle ground.

We can directly import any of these datasets in PyTorch Geometric with the Planetoid class:

```python
from torch_geometric.datasets import Planetoid

# Import dataset from PyTorch Geometric
dataset = Planetoid(root=".", name="CiteSeer")
data = dataset[0]

# Print information about the dataset
print(f'Number of graphs: {len(dataset)}')
print(f'Number of nodes: {data.x.shape[0]}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
print(f'Has isolated nodes: {data.has_isolated_nodes()}')
```

gat2.py hosted with ❤️ by GitHub view raw

```
Number of graphs: 1
Number of nodes: 3327
Number of features: 3703
Number of classes: 6
Has isolated nodes: True
```

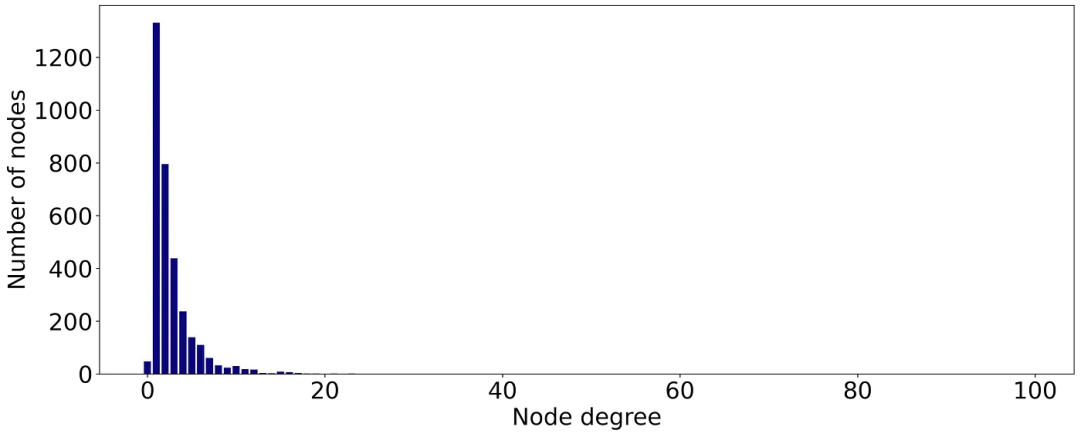
Interestingly enough, we have **3327 nodes instead of 3312.** I found that PyG actually uses this paper's implementation of CiteSeer, which also displays 3327 nodes. Mystery solved for now.

However, we observe that **some nodes are isolated** (48 to be precise)! Correctly classifying these isolated nodes will be a challenge since we cannot rely on any aggregation.

Let's plot the number of connections of each node with `degree`:

```python
from torch_geometric.utils import degree
from collections import Counter

# Get list of degrees for each node
degrees = degree(data.edge_index[0]).numpy()

# Count the number of nodes for each degree
numbers = Counter(degrees)

# Bar plot
fig, ax = plt.subplots(figsize=(18, 7))
ax.set_xlabel('Node degree')
ax.set_ylabel('Number of nodes')
plt.bar(numbers.keys(),
        numbers.values(),
        color='#0A047A')
```

gat_degree.py hosted with ❤️ by GitHub view raw



Most nodes only have **1 or 2 neighbors**. It could explain why CiteSeer obtains lower accuracy scores than the two other datasets...

⚠️ II. Self-attention

Introduced by Veličković et al. in 2017, self-attention in GNNs relies on a simple idea: **nodes should not all have the same importance.**

We talk about *self*-attention (and not just attention) because inputs are compared to each other.

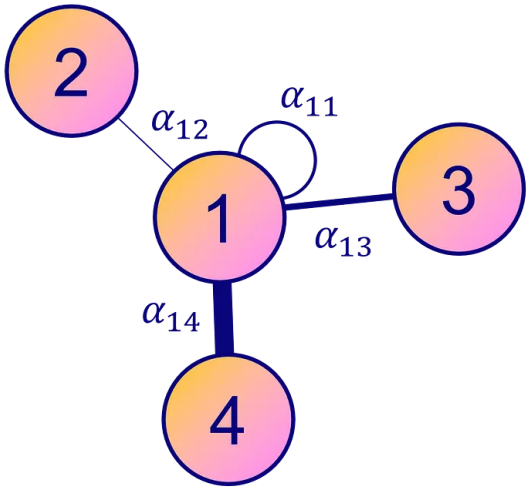


Image by author

This mechanism assigns a **weighting factor** (attention score) to each connection. Let's call α_{ij} the attention score between the nodes i and j.

Here's how to calculate the embedding of node 1, where $\mathbf{W}$ is a shared weight matrix:

$$h_1 = \alpha_{11}\mathbf{W}x_1 + \alpha_{12}\mathbf{W}x_2 + \alpha_{13}\mathbf{W}x_3 + \alpha_{14}\mathbf{W}x_4$$

But how do we calculate the attention scores? We could write a static formula, but there's a smarter solution: we can **learn their values with a neural network.** There are three steps in this process:

1. **Linear transformation;**
2. **Activation function;**
3. **Softmax normalization.**

1️⃣ Linear transformation

We want to calculate the **importance of each connection**, so we need pairs of hidden vectors. An easy way to create these pairs is to concatenate vectors from both nodes.

Only then can we apply a new **linear transformation** with a weight matrix $\mathbf{W}_{\mathbf{att}}$:

$$a_{ij} = W_{att}^{t}[\mathbf{W}x_i \parallel \mathbf{W}x_j]$$

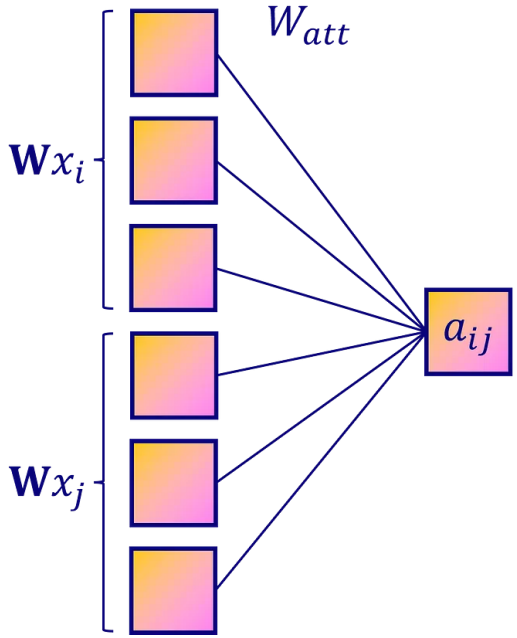


Image by author

2️⃣ Activation function

We're building a neural network, so the second step is to add an activation function. In this case, the authors of the paper chose the *LeakyReLU* function.

$$e_{ij} = LeakyReLU(a_{ij})$$

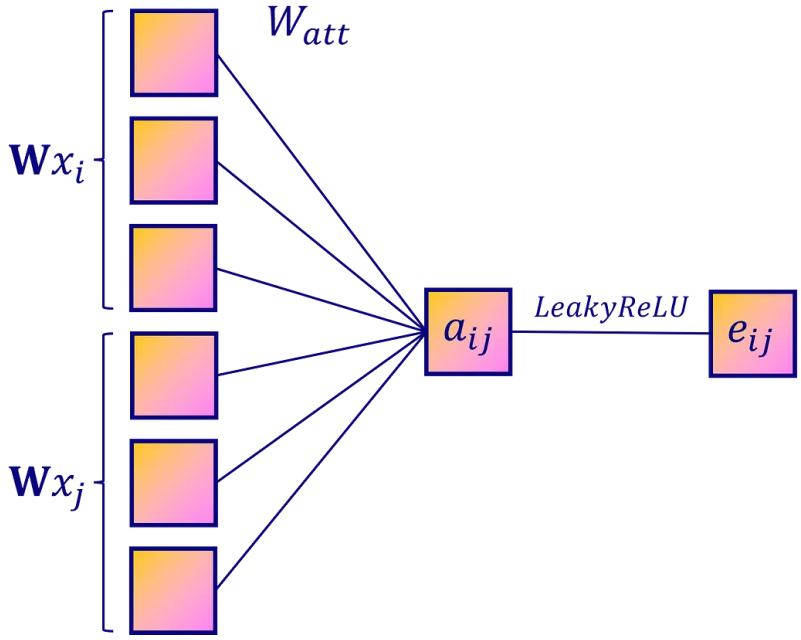


Image by author

3️⃣ Softmax normalization

The output of our neural network is **not normalized**, which is a problem since we want to compare these scores. To be able to say if node 2 is more important to node 1 than node 3 ($\alpha_{12} > \alpha_{13}$), we need to share the same scale.

A common way to do it with neural networks is to use the ***softmax*** function. Here, we apply it to every neighboring node:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} exp(e_{ik})}$$

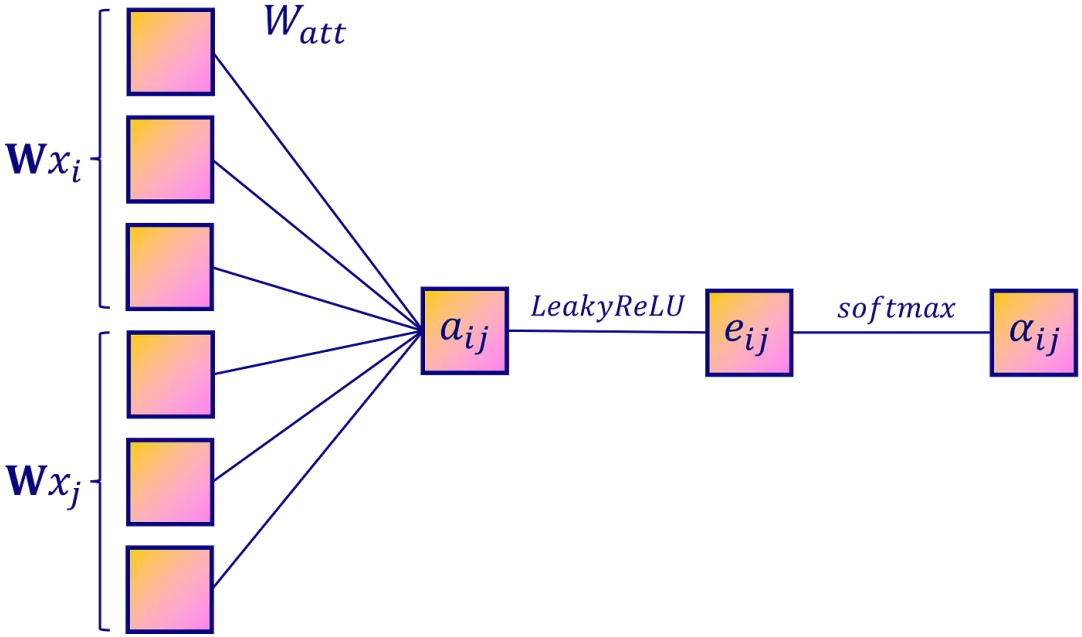
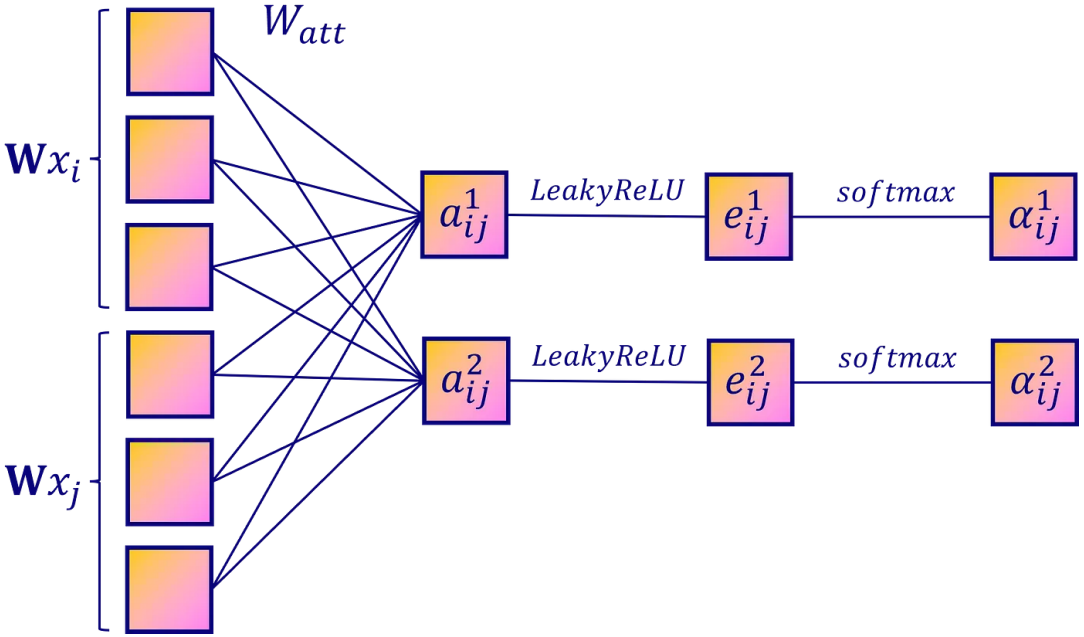


Image by author

Here you have it: we can calculate every α_{ij}. The only problem is... **self-attention is not very stable.** In order to improve performance, Vaswani et al. introduced multi-head attention in the transformer architecture.

4️⃣ Bonus: multi-head attention

This is only slightly surprising since we've been talking about self-attention a lot but, in reality, **transformers are GNNs in disguise**. This is why we can reuse some ideas from Natural Language Processing here.



Multi-head attention (image by author)

In GATs, multi-head attention consists of **replicating the same 3 steps several times** in order to average or concatenate the results. That's it. Instead of a single h_1, we get one hidden vector ${h_1}^k$ per attention head. One of the two following schemes can then be applied:

- **Average:** we sum the different ${h_i}^k$ and normalize the result by the number of attention heads n;

$$h_i = \frac{1}{n} \sum_{k=1}^{n} h_i^k$$

- **Concatenation:** we concatenate the different ${h_i}^k$.

$$h_i = \Big\|_{k=1}^{n} h_i^k$$

In practice, we use the **concatenation scheme** when it's a hidden layer, and the **average scheme** when it's the last layer of the network.

🧠 III. Graph Attention Networks

Let's implement a GAT in PyTorch Geometric. This library has **two different graph attention layers:** `GATConv` and `GATv2Conv`.

What we talked about so far is the `GatConv` layer, but in 2021 Brody et al. introduced an improvement by modifying the order of operations. The weight matrix **W** is applied **after the concatenation**, and the attention weight matrix $\mathbf{W}_{att}$ is used **after the *LeakyReLU* function**. In summary:

- `GatConv`:

$$e_{ij} = LeakyReLU(W_{att}^t[\mathbf{W}x_i \,\|\, \mathbf{W}x_j])$$

- `Gatv2Conv`:

$$e_{ij} = W_{att}^t LeakyReLU(\mathbf{W}[x_i \,\|\, x_j])$$

Which one should you use? According to Brody et al., **`Gatv2Conv` consistently outperforms `GatConv`** and thus should be preferred.

Now let's classify the papers from CiteSeer! I tried to **roughly reproduce the experiments** of the original authors without adding too much complexity. You can find the official implementation of GAT on GitHub.

Note that we use graph attention layers in two configurations:

- The **first layer** concatenates 8 outputs (multi-head attention);
- The **second layer** only has 1 head, which produces our final embeddings.

We're also gonna train and test a GCN to compare the accuracy scores.

```python
import torch.nn.functional as F
from torch.nn import Linear, Dropout
from torch_geometric.nn import GCNConv, GATv2Conv


class GCN(torch.nn.Module):
  """Graph Convolutional Network"""
  def __init__(self, dim_in, dim_h, dim_out):
    super().__init__()
    self.gcn1 = GCNConv(dim_in, dim_h)
    self.gcn2 = GCNConv(dim_h, dim_out)
    self.optimizer = torch.optim.Adam(self.parameters(),
                                      lr=0.01,
                                      weight_decay=5e-4)

  def forward(self, x, edge_index):
    h = F.dropout(x, p=0.5, training=self.training)
    h = self.gcn1(h, edge_index)
    h = torch.relu(h)
    h = F.dropout(h, p=0.5, training=self.training)
    h = self.gcn2(h, edge_index)
    return h, F.log_softmax(h, dim=1)


class GAT(torch.nn.Module):
  """Graph Attention Network"""
  def __init__(self, dim_in, dim_h, dim_out, heads=8):
    super().__init__()
    self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)
    self.gat2 = GATv2Conv(dim_h*heads, dim_out, heads=1)
    self.optimizer = torch.optim.Adam(self.parameters(),
                                      lr=0.005,
                                      weight_decay=5e-4)

  def forward(self, x, edge_index):
    h = F.dropout(x, p=0.6, training=self.training)
    h = self.gat1(x, edge_index)
    h = F.elu(h)
    h = F.dropout(h, p=0.6, training=self.training)
    h = self.gat2(h, edge_index)
    return h, F.log_softmax(h, dim=1)

def accuracy(pred_y, y):
```

```python
def accuracy(pred_y, y):
    """Calculate accuracy."""
    return ((pred_y == y).sum() / len(y)).item()

def train(model, data):
    """Train a GNN model and return the trained model."""
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = model.optimizer
    epochs = 200

    model.train()
    for epoch in range(epochs+1):
        # Training
        optimizer.zero_grad()
        _, out = model(data.x, data.edge_index)
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1), data.y[data.train_mask])
        loss.backward()
        optimizer.step()

        # Validation
        val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
        val_acc = accuracy(out[data.val_mask].argmax(dim=1), data.y[data.val_mask])

        # Print metrics every 10 epochs
        if(epoch % 10 == 0):
            print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc: '
                  f'{acc*100:>6.2f}% | Val Loss: {val_loss:.2f} | '
                  f'Val Acc: {val_acc*100:.2f}%')

    return model

def test(model, data):
    """Evaluate the model on test set and print the accuracy score."""
    model.eval()
    _, out = model(data.x, data.edge_index)
    acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
    return acc
```

gcn_vs_gat.py hosted with ❤ by GitHub view raw

```
%%time

# Create GCN
gcn = GCN(dataset.num_features, 16, dataset.num_classes)
print(gcn)

# Train
train(gcn, data)

# Test
acc = test(gcn, data)
print(f'GCN test accuracy: {acc*100:.2f}%\n')
```

gcn_train.py hosted with ❤️ by GitHub view raw

```
GCN(
  (gcn1): GCNConv(3703, 16)
  (gcn2): GCNConv(16, 6)
)

Epoch   0 | Train Loss: 1.782 | Train Acc:  20.83% | Val Loss: 1.79
Epoch  20 | Train Loss: 0.165 | Train Acc:  95.00% | Val Loss: 1.30
Epoch  40 | Train Loss: 0.069 | Train Acc:  99.17% | Val Loss: 1.66
Epoch  60 | Train Loss: 0.053 | Train Acc:  99.17% | Val Loss: 1.50
Epoch  80 | Train Loss: 0.054 | Train Acc: 100.00% | Val Loss: 1.67
Epoch 100 | Train Loss: 0.062 | Train Acc:  99.17% | Val Loss: 1.62
Epoch 120 | Train Loss: 0.043 | Train Acc: 100.00% | Val Loss: 1.66
Epoch 140 | Train Loss: 0.058 | Train Acc:  98.33% | Val Loss: 1.68
Epoch 160 | Train Loss: 0.037 | Train Acc: 100.00% | Val Loss: 1.44
Epoch 180 | Train Loss: 0.036 | Train Acc:  99.17% | Val Loss: 1.65
Epoch 200 | Train Loss: 0.093 | Train Acc:  95.83% | Val Loss: 1.73

GCN test accuracy: 67.70%

CPU times: user 25.1 s, sys: 847 ms, total: 25.9 s
Wall time: 32.4 s
```

```
%%time

# Create GAT
gat = GAT(dataset.num_features, 8, dataset.num_classes)
print(gat)

# Train
train(gat, data)

# Test
acc = test(gat, data)
print(f'GAT test accuracy: {acc*100:.2f}%\n')
```

gat_train.py hosted with ❤️ by GitHub view raw

```
GAT(
  (gat1): GATv2Conv(3703, 8, heads=8)
  (gat2): GATv2Conv(64, 6, heads=1)
)

Epoch   0 | Train Loss: 1.790 | Val Loss: 1.81 | Val Acc: 12.80%
Epoch  20 | Train Loss: 0.040 | Val Loss: 1.21 | Val Acc: 64.80%
Epoch  40 | Train Loss: 0.027 | Val Loss: 1.20 | Val Acc: 67.20%
Epoch  60 | Train Loss: 0.009 | Val Loss: 1.11 | Val Acc: 67.00%
Epoch  80 | Train Loss: 0.013 | Val Loss: 1.16 | Val Acc: 66.80%
Epoch 100 | Train Loss: 0.013 | Val Loss: 1.07 | Val Acc: 67.20%
Epoch 120 | Train Loss: 0.014 | Val Loss: 1.12 | Val Acc: 66.40%
Epoch 140 | Train Loss: 0.007 | Val Loss: 1.19 | Val Acc: 65.40%
Epoch 160 | Train Loss: 0.007 | Val Loss: 1.16 | Val Acc: 68.40%
Epoch 180 | Train Loss: 0.006 | Val Loss: 1.13 | Val Acc: 68.60%
Epoch 200 | Train Loss: 0.007 | Val Loss: 1.13 | Val Acc: 68.40%

GAT test accuracy: 70.00%

CPU times: user 53.4 s, sys: 2.68 s, total: 56.1 s
Wall time: 55.9 s
```

This experiment is not super rigorous: we'd need to **repeat it *n* times** and take the average accuracy with a standard deviation as the final result.

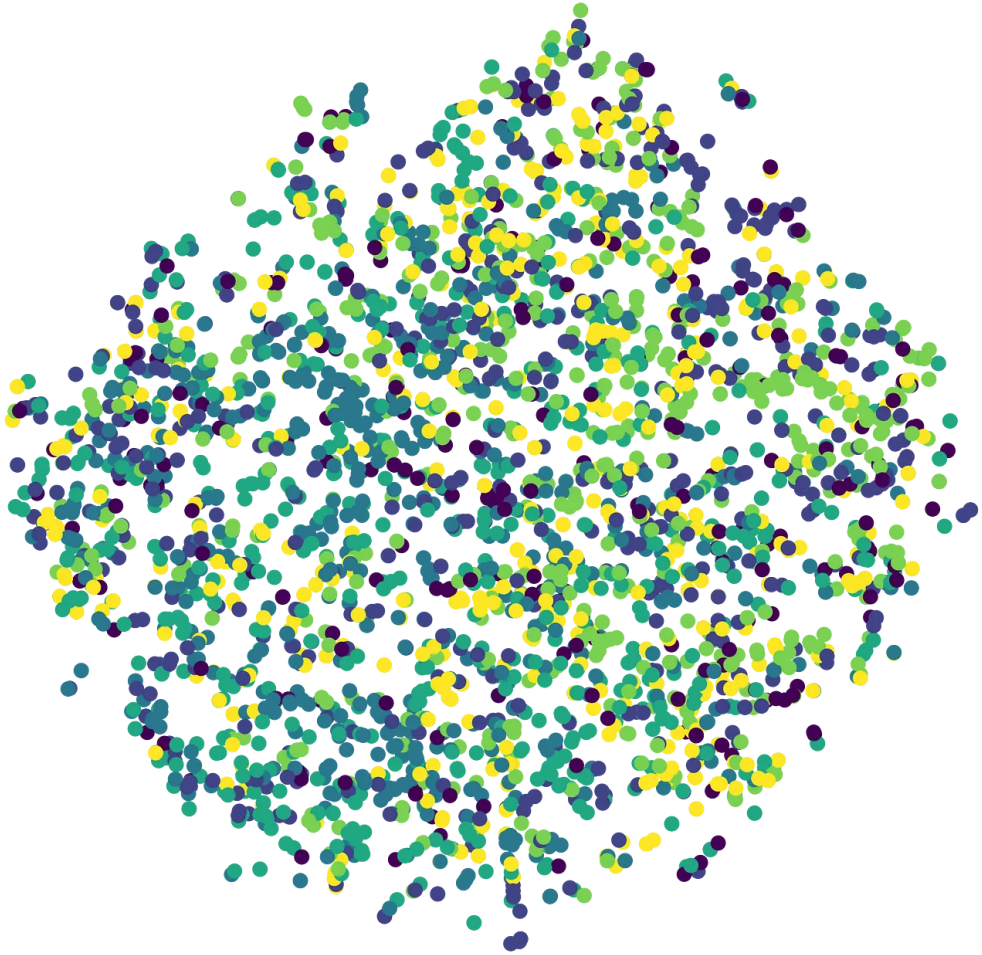
We can see in this example that the **GAT outperforms the GCN** in terms of accuracy (70.00% vs. 67.70%), but takes longer to train (55.9s vs. 32.4s). It's a tradeoff that can cause scalability issues when working with large graphs.

The authors obtained 72.5% for the GAT and 70.3% for the GCN, which is clearly better than what we did. The difference can be explained by **preprocessing**, some **tweaks in the models,** and a different **training setting** (*e.g.,* a patience of 100 instead of a fixed number of epochs).

Let's visualize what the GAT learned. We're gonna use t-SNE, a powerful method to plot high-dimensional data in 2D or 3D. First, let's see what the embeddings looked like before any training: it should be absolutely **random** since they're produced by randomly initialized weight matrices.

```
untrained_gat = GAT(dataset.num_features, 8, dataset.num_classes)

# Get embeddings
h, _ = untrained_gat(data.x, data.edge_index)

# Train TSNE
tsne = TSNE(n_components=2, learning_rate='auto',
         init='pca').fit_transform(h.detach())

# Plot TSNE
plt.figure(figsize=(10, 10))
plt.axis('off')
plt.scatter(tsne[:, 0], tsne[:, 1], s=50, c=data.y)
plt.show()
```

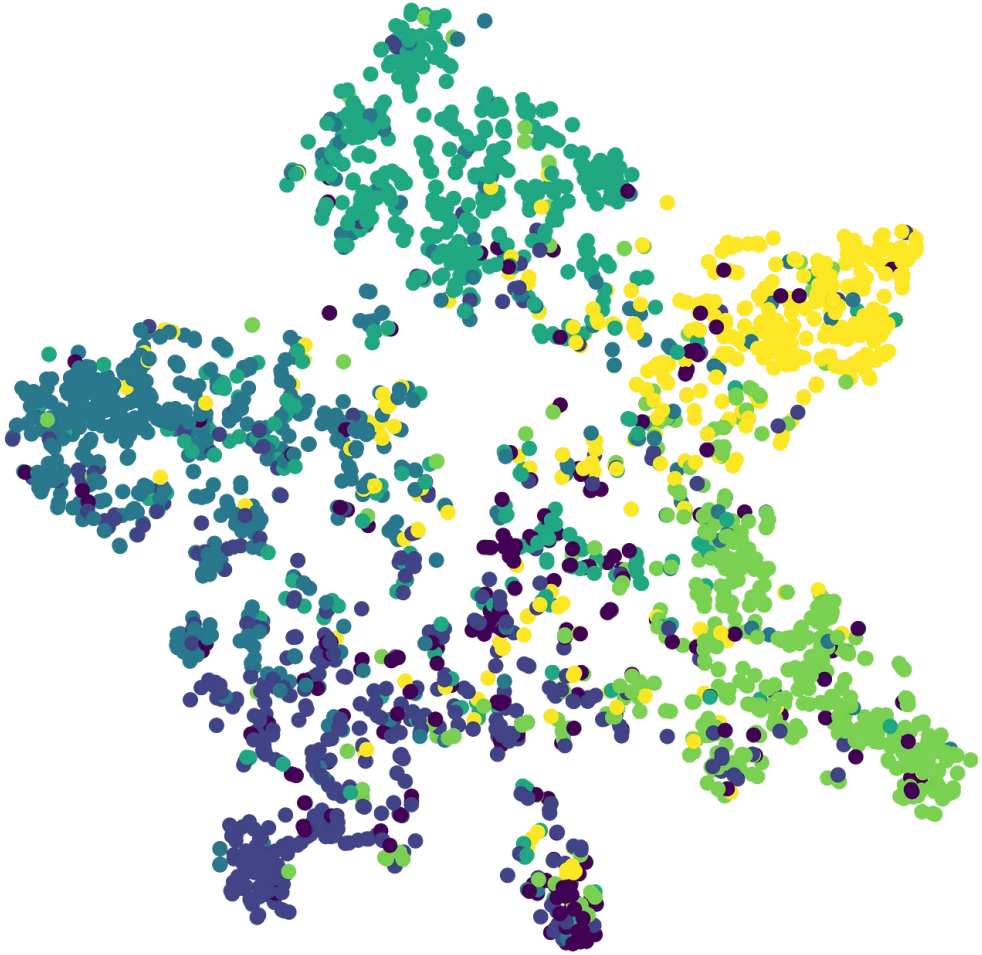
tsne_untrained.py hosted with ❤ by GitHub view raw



Indeed, there's **no apparent structure**. But do the embeddings produced by our trained model look better?

```python
h, _ = gat(data.x, data.edge_index)

# Train TSNE
tsne = TSNE(n_components=2, learning_rate='auto',
         init='pca').fit_transform(h.detach())

# Plot TSNE
plt.figure(figsize=(10, 10))
plt.axis('off')
plt.scatter(tsne[:, 0], tsne[:, 1], s=50, c=data.y)
plt.show()
```

tsne_trained.py hosted with ❤️ by GitHub view raw



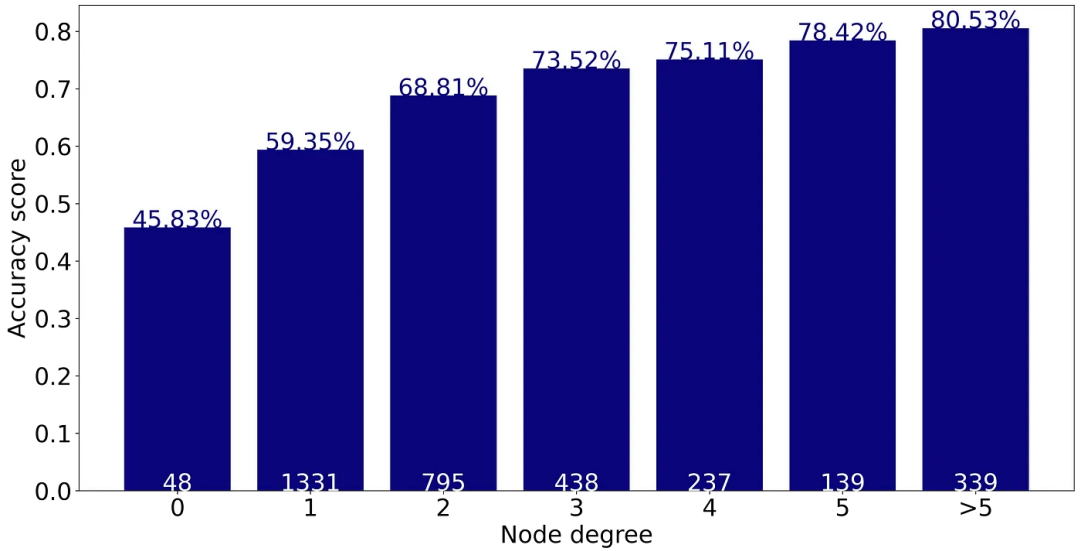
The difference is noticeable: **nodes belonging to the same classes cluster together**. We can see 6 clusters, corresponding to the 6 classes of papers. There are outliers, but this was to be expected: our accuracy score is far from perfect.

Previously, I speculated that poorly connected nodes might **negatively impact** performance on CiteSeer. Let's calculate the model's accuracy for

each degree.

```python
from torch_geometric.utils import degree

# Get model's classifications
_, out = gat(data.x, data.edge_index)

# Calculate the degree of each node
degrees = degree(data.edge_index[0]).numpy()

# Store accuracy scores and sample sizes
accuracies = []
sizes = []

# Accuracy for degrees between 0 and 5
for i in range(0, 6):
  mask = np.where(degrees == i)[0]
  accuracies.append(accuracy(out.argmax(dim=1)[mask], data.y[mask]))
  sizes.append(len(mask))

# Accuracy for degrees > 5
mask = np.where(degrees > 5)[0]
accuracies.append(accuracy(out.argmax(dim=1)[mask], data.y[mask]))
sizes.append(len(mask))

# Bar plot
fig, ax = plt.subplots(figsize=(18, 9))
ax.set_xlabel('Node degree')
ax.set_ylabel('Accuracy score')
ax.set_facecolor('#EFEEEA')
plt.bar(['0','1','2','3','4','5','>5'],
        accuracies,
        color='#0A047A')
for i in range(0, 7):
    plt.text(i, accuracies[i], f'{accuracies[i]*100:.2f}%',
             ha='center', color='#0A047A')
for i in range(0, 7):
    plt.text(i, accuracies[i]//2, sizes[i],
             ha='center', color='white')
```

node_degree.py hosted with ❤️ by GitHub view raw



These results confirm our intuition: nodes with few neighbors are indeed **harder to classify**. This is due to the nature of GNNs: the more relevant connections you have, the more information you can aggregate.

Conclusion

While they take longer to train, GATs are a **substantial improvement** over GCNs in terms of accuracy. The self-attention mechanism automatically calculates weighting factors instead of static coefficients to produce better embeddings. In this article,

- We learned about the **self-attention** mechanism applied to GNNs;
- We implemented and **compared** two architectures (a GCN and a GAT) in PyTorch Geometric;
- We visualized how and what the GAT learns with a **t-SNE** plot and the accuracy score for each degree;

GATs are the de facto standard in a lot of GNN applications. However, their **slow training time** can become a problem when applied to massive graph datasets. Scalability is an important factor in deep learning: most often, more data can lead to better performance.

In the next article, we'll see **how to improve scalability** with mini-batching and a new GNN architecture called GraphSAGE.

If you enjoyed this tutorial, feel free to **follow me on Twitter** for more GNN content. Thank you and see you in the next article! 📣

Related articles

Introduction to GraphSAGE in Python
Scaling Graph Neural Networks to billions of connections
towardsdatascience.com

How to Design the Most Powerful Graph Neural Network
Graph classification with Graph Isomorphism Networks
towardsdatascience.com

Programming | Python | Machine Learning | Hands On Tutorials | Editors Pick

Written by Maxime Labonne

Follow

860 Followers · Writer for Towards Data Science

Ph.D., Author of "Hands-On Graph Neural Networks" • Senior Applied Researcher @ JPMorgan • Let's connect on Twitter! twitter.com/maximelabonne

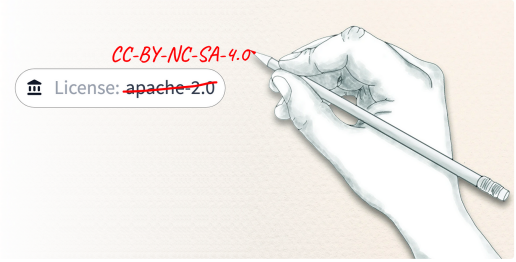
More from Maxime Labonne and Towards Data Science

Maxime Labonne in Towards Data Science

Decoding Strategies in Large Language Models

A Guide to Text Generation From Beam Search to Nucleus Sampling

· 15 min read · Jun 4



Clemens Mewald in Towards Data Science

The Golden Age of Open Source in AI Is Coming to an End

NC, SA, GPL, and other acronyms you don't want to see in the open source license of the...

7 min read · Jun 7

401 3 916 14

Luís Roque in Towards Data Science

Harnessing the Falcon 40B Model, the Most Powerful Open-Source...

Mastering open-source language models: diving into Falcon-40B

· 12 min read · Jun 9

240 4

Maxime Labonne in Towards Data Science

GIN: How to Design the Most Powerful Graph Neural Network

Graph classification with Graph Isomorphism Networks

· 8 min read · Apr 27, 2022

300 1

See all from Maxime Labonne | See all from Towards Data Science

Recommended from Medium

AI TutorMas... in Artificial Intelligence in Plain Eng...

Graph Neural Networks—Introduction for Beginners

A Graph Neural Network (GNN) is a type of neural network that is designed to work with...

· 10 min read · Jan 14

424 1

Isaac Godfried in Towards Data Science

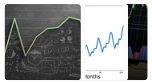
Advances in Deep Learning for Time Series Forecasting and...

The downfall of transformers for time series forecasting and the rise of time series...

· 16 min read · Jan 10

332 5

Lists



Predictive Modeling w/ Python

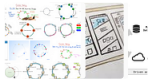
18 stories · 10 saves

Practical Guides to Machine Learning

10 stories · 23 saves

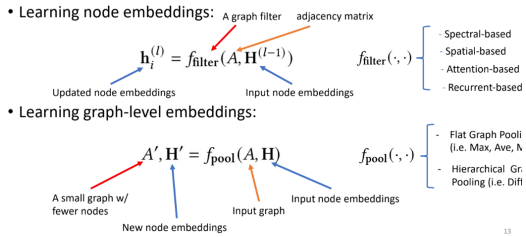
Coding & Development

11 stories · 4 saves



General Coding Knowledge

20 stories · 13 saves



Jason Huang

Graph Encoder-Decoder Models for NLP

All of the contents in this article are excerpted from the tutorial video, slides, and website of...

5 min read · Dec 31, 2022

27 1

Sherry Wu in Stanford CS224W GraphML Tutorials

Spread No More: Twitter Fake News Detection with GNN

By Li Tian, Sherry Wu, Yifei Zheng as part of the Stanford CS224W course project.

11 min read · May 16

1

Brackly Murunga

Predicting a customer's next purchase using graph neural...

Part I

5 min read · Jun 4

Faxi Yuan, PhD

Graph Neural Networks (GNNs): Comparison between CNNs and...

Based on my previous story on message passing framework for node classifications, I...

· 7 min read · May 30

3

See more recommendations