Spring 2023 CS598 DL4H: Graph Attention Networks Reproducibility Project

Valle-Mena, Ricardo Ruy and Soni, Kushagra
{rrv4, soni14}@illinois.edu

Group ID: 179
Paper ID: 7
Presentation link: https://youtu.be/OLJBoOzjBHg
Code link: https://github.com/kushagrasoni/CS598_DLH_GAT_Implementation

1 Introduction

Graph Attention Networks (GATs) [1] are a type of neural network architecture designed for processing graph-structured data. They aim to overcome the limitations of previous architectures by being able to operate on arbitrarily structured graphs in a parallelizable manner. GATs use a shared masked self-attention mechanism to assign weights to each node's neighbors and combine their features, resulting in a new set of features for the node in question. GATs have shown promising results in various graph-based tasks, including node classification and link prediction.

2 Scope of Reproducibility

2.1 Background and Problem Statement

Before Graph Attention Networks (GATs), several neural network architectures were proposed to process graph-structured data, including Graph Convolutional Networks (GCNs), GraphSAGE, and DeepWalk. However, these architectures had one or more of the following limitations:

- Inability to operate on arbitrarily structured graphs: GCNs, for example, are limited to processing homogeneous graphs where all nodes have the same features.
- Need to sample from input graphs: some architectures, like GraphSAGE, require sampling from input graphs, which leads to information loss.
- Learning separate weight matrices for different node degrees: GCNs use different weight matrices for nodes with different degrees.
- Inability to parallelize training across nodes.

Overall, prior architectures faced difficulties in handling the complexity and heterogeneity of real-world graphs, which motivated the development of Graph Attention Networks.

GATs aim to be able to operate on arbitrarily structured graphs in a manner that is parallelizable across nodes in the graph, thus having none of the limitations mentioned previously. GATs use a masked shared self-attention mechanism. The mask ensures that, for a given node, only features from first-degree neighbours are taken into consideration. The self-attention mechanism allows the model to assign arbitrary weights to each of a given node's neighbours, which then allows the neighbours' features to be combined, which results in a new set of features for the node in question. The paper only mentions using the resulting features for classification tasks, but it should also be possible to use these features for regression tasks.

2.2 Objectives

Our goal was to reproduce the results reported in the original paper, which used the Cora, Citeseer, Pubmed, and Protein-protein interaction datasets. Specifically, we aimed for:

- Classification accuracies of approximately 83.0%, 72.5%, and 79% in the Cora, Citeseer, and Pubmed datasets, respectively,
- Micro-averaged F1 score of approximately 0.973 in the protein-protein interaction dataset.

Furthermore, we trained two one-layer GAT models and a three-layer GAT model on the Cora, Citeseer and Pubmed datasets as ablation studies. We hypothesized that one-layer models would perform worse than two-layer models, which would in turn perform worse than three-layer models.

3 Methodology

We tried a total of 5 GAT implementations, four of which were our own, and one of which is from the Pytorch Geometric library. One of our implementations was strongly influenced by the paper's authors' implementation [2].

All datasets from the paper are publicly available in multiple locations, including the Pytorch Geometric library. We have been running our experiments locally. Despite the datasets being relatively small, the PPI dataset turned out to be sufficiently large to take a prohibitively long time to run. We

therefore ended up not having results to present for the PPI data.

3.1 Model Description

3.1.1 Architecture

There are several variants of the same model used in the paper. For the Cora and Citeseer datasets, two-layer GAT models were used. The first layer has 8 attention heads, projects the input graph's features to an 8-dimensional feature space, and uses an exponential linear unit (ELU) as its activation function. The second layer has a single attention head, projects the data to a C-dimensional feature space, where C is the number of classes in the dataset, and uses a softmax activation function. L2 regularization is applied with lambda = 0.0005, and dropout is applied with p = 0.6.

For the Pubmed data, the architecture is mostly the same. However, the second layer has 8 attention heads like the first layer, and the L2 regularization uses a coefficient of 0.001 instead of 0.0005.

For the protein-protein interaction data, a three-layer model is used. The first two layers have 4 attention heads, project their input data to a 256-dimensional feature space, and use an ELU activation function. The third layer has 6 attention heads, projects its input data to a 121-dimensional feature space, averages all 121 dimensions, and applies a softmax activation function.

We employed the same early stopping strategy employed by the paper's authors, with a patience of 100 epochs, during all training.

3.1.2 Learning Objectives

The learning objectives for the datasets are as follows:

- Cora: To classify academic papers into one of seven classes based on their content and the citations between papers.
- Citeseer: Given a graph where nodes represent research papers and edges represent citation links between papers, the model is trained to predict the subject area of each paper based on its citation links and the text of its title and abstract. The dataset contains 3,327 papers, each belonging to one of six subject areas: "Agents", "AI", "DB", "IR", "ML", or "HCI". The objective is to correctly classify the papers into their respective subject areas
- Pubmed: The goal is to predict the category of a scientific publication based on the citation network of papers. There are three possible categories: diabetes mellitus, cardiovascular diseases, and neoplasms
- PPI: Given a graph where nodes represent proteins and edges represent interactions between proteins, the task is to predict whether proteins have each of 121 possible characteristics.

3.2 Data Description

We used the same datasets as in the GAT paper: Cora, Citeseer, Pubmed, and Protein-Protein Interaction (PPI). The Cora, Citeseer and Pubmed datasets were originally introduced in [3] and the PPI dataset was introduced in [4].

Table 1 summarizes the basic statistics of the four datasets. Note that the number of features is different for each dataset, as is the number of classes and the sparsity of the adjacency matrix.

We used the same dataset splits and evaluation metrics as in the original paper. The data and splits are readily available through the Planetoid and PPI classes from Pytorch Geometric [5].

3.2.1 Cora

The Cora dataset consists of 2,708 scientific publications classified into one of seven categories. The citations between publications form a graph, where each node represents a publication and each edge represents a citation.

3.2.2 Citeseer

The Citeseer dataset consists of 3,327 scientific publications classified into one of six categories. Similarly, the citations between publications form a graph.

3.2.3 Pubmed

The Pubmed dataset consists of 19,717 scientific publications from the PubMed database, where each publication is associated with one or more MeSH (Medical Subject Headings) terms. The graph is constructed using the citation links between publications and each node represents a publication. The task is to predict the MeSH terms associated with each publication.

3.2.4 PPI

In the PPI dataset [6] there are multiple graphs, where each graph represents a tissue and each node in the graph represents a protein. The goal of the task is to predict the biological function labels of the proteins in a previously unseen tissue graph. There are 121 possible labels that a protein node can have, and the task is to predict all of the labels for each protein node in the test graphs. The dataset is divided into 20 training graphs, 2 validation graphs, and 2 test graphs.

3.3 Hyperparameters

There aren't really any hyperparameters to speak of, other than those described in the model description section (the number of layers, L2 regularization coefficient, etc.). All hyperparameters were therfore taken directly from the paper.

Dataset	Nodes	Edges	F/N	Classes
Cora	2,708	5,429	1,433	7
Citeseer	3,327	4,732	3,703	6
Pubmed	19,717	44,338	500	3
PPI	56,944	818,716	50	20

Table 1: Dataset statistics; F/N represents Features/Node

3.4 Model Implementation

For starters, we tried using the GAT model that is bundled with Pytorch Geometric. It appears this implementation is not flexible enough to exactly follow what the paper did, but we tried to stay as close as possible, so we called it as follows:

```
from torch_geometric.nn import GAT
model = GAT(
    in_channels=dataset.num_features,
    out_channels=dataset.num_classes,
    hidden_channels=8,
    num_layers=2,
    heads=8,
    dropout=0.6,
    act='elu',
    act_first=True
)
```

The 'hidden_channels' parameter tells us that the data from each node in the input graph are projected to an 8-dimensional space via a linear transformation (a matrix multiplication). The 'act' parameter tells us that the exponential linear unit is then applied to the transformed data. 'heads' indicates that this is repeated 8 different times, once per "attention head", meaning there are 8 separate linear transformations from the original data's space to 8-dimensional space. 'dropout' indicates that dropout is applied with parameter p = 0.6. Lastly, 'num_layers' indicates that what this paragraph describes is repeated twice, with the output of the first later being fed into the second layer.

As mentioned, this does not quite follow the models as described in the paper, but this was a useful step for us to figure out how to feed the data into the model, and more generally how to set everything up. The Pytorch Geometric implementation of graph attention networks does not allow, for instance, to specify a different activation function for each layer, which would be required to exactly follow the paper's methodology.

We therefore built *four different implementations of GAT*, one using Pytorch Geometric's GATConv class, one using Pytorch Geometric's GATv2Conv class, and two using Pytorch primitives. This allowed us, to the best of our knowledge, to follow the paper's methodology exactly.

We say "to the best of our knowledge" because parts of the methodology are not well explained in the paper, and the authors only published the code they used to run the model on Cora in addition to the model itself. There may therefore be details when running the model on the other three datasets where we deviate from the paper. We are confident, however, that any such deviations are small.

3.5 Computational Requirements

To reproduce the results reported in the GAT paper, we ran the GAT model using PyTorch version 1.9.0 [7] on two local machines, a Macbook and a Dell laptop running Solus OS (a Linux distribution), both of which have Intel Core i7 CPUs (2.6 GHz on the Macbook, 1.8 GHz on the Dell machine), 16GB RAM, and Intel UHD GPUs.

On the local systems, we installed PyTorch and all necessary dependencies using the pip package manager. Training and evaluating the GAT model on the Cora and Citeseer datasets took approximately 20-25 seconds for 200 epochs. Training and evaluating the GAT model on the Pubmed dataset took approximately 2 and half minutes for 200 training epochs.

We tried to run the PPI dataset on both the local machines mentioned earlier. The Mac system errored out with "no application memory available". The Linux ran for roughly 16 hours before completing. We ended up only running a single model once on the PPI data and we ended up discarding our results because the architecture didn't quite match what was used in the paper.

Overall, the computational requirements for reproducing the GAT results on a local system are high for the PPI data and low for the other datasets, and may require a GPU for faster training times, which is especially important for the PPI dataset.

4 Results

Our results can be broken up into three parts: first, replicating the paper's methodology and results, second, rewriting the GAT model using Pytorch primitives, and third, experimenting with variants of the architecture used in the paper.

We first attempted to replicate the paper's methodology and results as closely possible using the Pytorch Geometric library. We first used the GAT class they provide, but found it was not flexible enough to properly replicate the model architectures used in the paper. We then implemented two versions of GAT, using Pytorch Geometric's GATConv and GATv2Conv classes. The former is an implementation of a single GAT layer, and the latter is an implementation of a single layer from a subsequent paper [8], which claims to have made the attention mechanism more powerful.

Dataset	GAT Type	Mean	Std. Dev.
Cora	GAT	78.99%	1.13%
Cora	GATConv	78.86%	1.15%
Cora	GATv2Conv	78.53%	1.04%
Citeseer	GAT	67.01%	1.43%
Citeseer	GATConv	66.83%	1.41%
Citeseer	GATv2Conv	66.65%	1.47%
Pubmed	GAT	77.73%	1.11%
Pubmed	GATConv	77.38%	1.32%
Pubmed	GATv2Conv	77.73%	1.22%

Table 2: Results using various GAT implementations. Each dataset was executed with each GAT type 50 times and the Mean and Standard deviation results was taken

4.1 Results: Citeseer, Cora and Pubmed

We ran all three of these models on the Cora, Citeseer and Pubmed datasets 50 times. Each run consisted in 200 epochs of training. The results are shown in Figure 1.

As we can see, from comparing Table 2 with the results from the paper, our Cora and Citeseer results are slightly worse than the Cora and Citeseer results from the paper. In the case of Pubmed, our results are quite a bit worse, with our best runs being a little worse than the average from the paper, and our worse results being a lot worse as seen in Figure 1.

4.2 Results: PPI

Training the model on the PPI data has proven to be significantly slower than on the other datasets. We only ran the PPI data through the implementation using the Pytorch Geometric GAT class, and ended up discarding our results because the GAT class isn't flexible enough to accommodate the paper's architecture, and we figured we would get back to this, which we unfortunately didn't. The PPI data took roughly 16 hours to run on one of our machines and made the machine essentially unusable during those 16 hours, and ran out of memory when we tried to run it on another machine.

4.3 Results: Pytorch implementation

We also tried writing our own implementation of the GAT model using Pytorch primitives. The idea was to gain a deeper understanding of the model and to be able to experiment more flexibly with the model, since we can control everything in the model with an implementation written from scratch, whereas there are aspects of the model that we can't control when using Pytorch Geometric's classes.

Unfortunately, both our attempts are rewriting GAT using Pytorch failed. We first tried to translate the implementation written by the paper's authors [2] from Tensorflow to Pytorch, but its accuracy stopped increasing on the training data at roughly 50% accuracy. This is in stark contrast to the previously-discussed implementations which achieved well over 80% accuracy on the training data.

We then used *another Pytorch implementation we found online* as a reference for another Pytorch GAT implementation. Similarly to the previous attempt, training accuracy stopped increasing around 45% accuracy.

Model Type	Dataset	Accuracy
Single Layer GAT (v1)	Cora	73.20%
Single Layer GAT (v2)	Citeseer	62.70%
Three Layer GAT	Pubmed	70.40%

Table 3: Results for various ablations in the GAT implementation.

4.4 Ablation Results

Lastly, we ran the Cora, Citeseer and Pubmed datasets through three architectural variants of the model. In two cases, we wrote single-layer GAT implementations, and the other was a three-layer GAT implementation. The results for all the ablations are shown in the Table 3.

5 Discussion

Our results show that, despite some small details being unclear from the paper, the paper's results can be closely replicated fairly straightforwardly. Having access to an NVIDIA GPU would likely have made it possible to experiment more with the PPI dataset. Having a stronger background in linear algebra would have enabled us to implement the model using Pytorch's primitives more easily, and to optimize the code to make it faster.

Our ablation results are somewhat surprising. Both single-layer models performed worse than the architectures used in the paper, which is not surprising, but one of the single-layer models appears to have performed better than the other, which is somewhat surprising. The only difference between the two is that the second used two activation functions: the exponential linear unit followed by softmax. The first, on the other hand, used only softmax.

The three-layer model performed better than the single-layer models, which is unsurprising, but it did not appear to perform better than the two-layer models from the paper, which is somewhat surprising. This perhaps suggests that tuning the hyperparameters of the model is just as important as adding more parameters via an extra layer.

As mentioned previously, the PPI dataset turned out to be sufficiently large to make experimenting

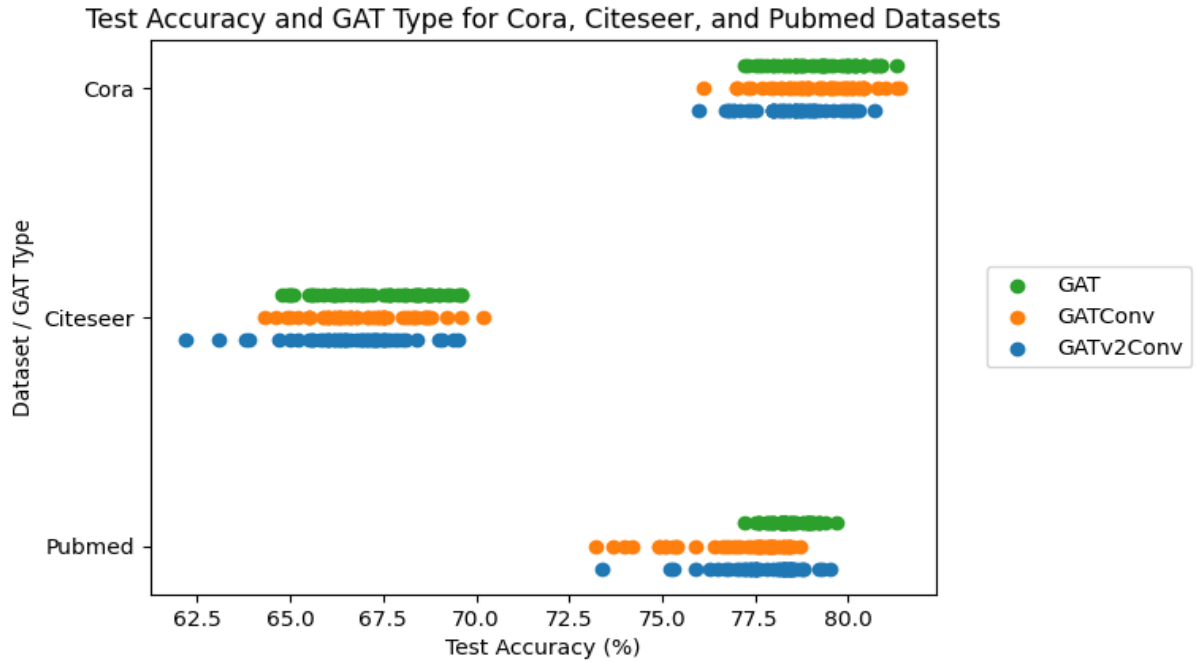

Figure 1: Benchmarking Results for various datasets for every GAT types used. Each dataset was executed for 50 iterations, where each iteration consisted of 200 training epochs

with it prohibitively slow, so we were unable to experiment with it as thoroughly as with the other three datasets. We are unsure whether getting our code to run on a GPU would change this.

5.1 What was easy

There were a few tasks which took less effort than the others:

- Reading and understand the purpose of the paper was easy. Most of the important sections, namely the layer structure, the early stopping mechanism, and the models parameters, were explained nicely.
- Finding and accessing the datasets was easy and was made easier by torch_geometirc's Planetoid library.
- Utilizing the packaged open-source GAT library to train and test the various datasets was easy. All it requires is providing the exact same input parameters which were used by the original authors.

5.2 What was difficult

- Complex architecture: Understanding the architecture was definitely one of the most difficult tasks, let alone implement it. The GAT model has a complex architecture, involving multiple layers and attention mechanisms, which was difficult to implement.
- Memory constraints: The GAT model can require a large amount of memory, particularly when processing large graphs or using large batch sizes like that of PPI, which can make it challenging to train on standard hardware.
- GAT implementation using Pytorch: Implementing GAT using Pytorch was especially difficult. It is easy to make subtle mistakes while wiring up the various parts of the model in Pytorch, and it is difficult to debug what is happening since the heavy lifting in Pytorch happens in C++ rather than Python, which makes it impossible to step through in Python debugger.

5.3 Recommendations for reproducibility

- Owning an NVIDIA GPU and running the models on the GPU might help to train the models faster and to be able to handle larger datasets.
- Having a solid understanding of linear algebra is very important. It will help understand the model quickly, it will help translate the math behind the paper into code, and it will help to optimize the operations of the model.
- Knowing good debugging techniques and having experience debugging neural network models is tremendously valuable.

6 Communication with original authors

We did not communicate with the paper's authors at all.

References

[1] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

[2] Guillem Cucurull Petar Veličković. Gat. `https://github.com/PetarV-/GAT`, 2018.

[3] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, and Brian Galligher. Collective classification in network data. In *AI magazine*, volume 29, pages 93–93. AAAI Press, 2008.

[4] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[5] Matthias Fey and Jan E. Lenssen. Pytorch geometric documentation, 2021.

[6] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. Predicting combinatorial drug synergy in multi-dimensional cancer immunotherapies with genomics and distributed computing. *Pacific Symposium on Biocomputing*, 22:410–421, 2017.

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

[8] Petar Veličković, William Fedus, and William L. Hamilton. How attentive are graph attention networks? *value*, 2021.