Graph Attention Networks

Khalid Aleem

December 2022

Abstract

We explain the intuition and theory of Graph Attention Networks[8] (GATs), starting with the motivation behind Graph Convolution Networks[3], provide an explanation of the message passing algorithm, and explain earlier developments in Graph Signal Processing[6, 4]. We then proceed to demonstrate state-of-the art results, implementing a Graph Attention Network from scratch in PyTorch and training it on the CORA citations dataset. For the purpose of comparison we also implement a Graph Convolution Network[3] (GCN). We observe improved test accuracy and loss on the GAT and also reveal that the GAT is superior at segmenting and classifying the CORA dataset.

1 Introduction

Graph Attention Networks (GATs) have offered further improvements over Graph Convolution Networks (GCNs) in machine learning tasks such as classification, allowing 'attention', or priorities of neighbours to be learnt and considered, increasing the number of parameters over older methods. GATs were introduced in a seminal paper[8], by P. Velickovic, et al, offering an improvement over the recently developed Graph Convolution Network[3] (GCN).

To best understand Graph Attention Networks, we must first delve into Graph Convolution Networks. In the past, it was preferential to make use of *spectral methods* for processing graphs. This involves calculating a fourier transform of our graph, applying the transform to our graph features, and then applying filters in the frequency domain (i.e. low-pass, high-pass), in order to process our graph. This is discussed more thoroughly in (Section 2).

Spectral methods, namely Graph Signal Processing (GSP)[6, 4], suffer from a series of caveats: It is very computationally expensive on larger graphs (due to performing an eigenvector decomposition of the laplacian matrix to obtain the fourier transform matrix); the fourier transform matrix is specific to the graph it is calculated for, therefore there exists a degree of computational overhead to apply filters to new graphs; these methods also do not trivially integrate with

neural networks.

Graph Convolution Networks are a *spatial method* where we do not transform into the frequency domain. The base unit of a GCN is known as a GCN Layer, an individual layer of a neural network where our input consists of feature vectors for each node in our graph, and our output consists of transformed feature vectors for each node in our graph. These may be subject to supervised learning and trained through the backpropagation algorithm. Crucially, the model parameters are independent of the structure of the graph. GCN Layers are significantly easier to understand and implement than spectral methods and offer a greater degree of flexibility and speed. Graph Attention Networks are an improvement of Graph Convolutional Networks, they introduce the concept of *attention*, also seen in the seminal paper on transformers[7] by A. Vaswani.

2 Graph Signal Processing (GSP)

Earlier spectral methods of performing graph analysis on undirected graphs involves *graph signal processing*. We may take the fourier fransform of a graph laplacian, $L = D - A$, by performing an eigenbasis decomposition, with eigenvalue matrix, Σ:

$$L = U\Sigma U^T \tag{1}$$

Each eigenvalue corresponds to a frequency in the graph. The greatest eigenvalues correspond to the greatest frequencies. The eigenbasis transformation, U^T, may therefore be used, to move the graph features, $\boldsymbol{f}$ into the frequency domain:

$$\tilde{\boldsymbol{f}} = U^T \boldsymbol{f} \tag{2}$$

In frequency space, we may multiply by filter matrix, F, where we set 0 in the column(s) corresponding to the frequencies/eigenvalues we seek to remove. We may therefore use this technique as a low-pass filter, to remove high frequencies and project back into real space: $\boldsymbol{f}_{\text{lowpass}} = UFU^T \boldsymbol{f}$.

3 Graph Convolution Networks (GCN)

Graph Attention Networks may be best understood starting with Graph Convolution Networks. Certainty, we seek to generalise the convolution operator to be independent of the structure of the input graph.

We seek an operation, f, that can be applied to graphs that is:

- Translation invariant (convolutions operator extracts local features)
- Locality (neighbouring nodes influence each other more)
- Differentiable (for backpropagation)

- Permutation equivariance: $\boldsymbol{P}f(\boldsymbol{X}) = f(\boldsymbol{PX})$.

Permutation equivariance in particular ensures that the operation applied to two identical graphs, with different adjacency matrices will produce the same output.

We seek to perform an operation as follows:

$$\text{output features} = \text{activation}(\text{weights} * \text{Agg}(\text{neighbouring features}))$$

The aggregation (Agg) function (sum or mean) is clearly permutation invariant. The activation function introduces non-linearity into our output features, and the weight matrix is used for transforming our input feature space into output feature space. An operation similar to this is performed through a procedure known as message passing.

3.1 Graph Convolution Layer: Message Passing

The operation of a graph convolution layer can be understood through the use of a concept called message passing. Our *message* at node i, m_i, is the aggregated features of neighbouring nodes.

The simplest example of message passing is multiplying our set of node features $\boldsymbol{f} = \{\vec{f_1}, \vec{f_2}, ..., \vec{f_N}\}$ by our adjacency matrix A. This produces a resulting feature vector where the value at each index is the sum of the neighbours of output features.

$$\vec{f_i}^{(l+1)} = \sum_{j \in N(i)} \vec{f_j}^{(l)}$$
$$\boldsymbol{f^{(l+1)}} = A\boldsymbol{f}^{(l)}$$

We add self-directed edges to our adjacency matrix, $\tilde{A} = A + I$, and then normalize our results by neighbourhood size of the vertex at the output index, $\tilde{D}^{-1}\tilde{A}$, where $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ is the neighbourhood size of node i. A diagram of aggregated nodes to produce the output result is shown in (Figure 1).

$$\vec{f_i}^{(l+1)} = \frac{1}{\tilde{d}_i} \sum_{j \in \{i\} \cup N(i)} \vec{f_j}^{(l)}$$
$$\boldsymbol{f}^{(l+1)} = \tilde{D}^{-1}\tilde{A}\boldsymbol{f}^{(l)}$$

In order to keep propagation through multiple layers numerically stable, we proceed to normalize by the neighbourhood size of each neighbouring node j, and then normalize our overall result by the neighbourhood size of node i, this can be expressed as follows and yields our message at each node, i, of our graph,

we call our *normalised adjacency matrix*, $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$:

$$\vec{f}_i^{(l+1)} = \frac{1}{\sqrt{\tilde{d}_i}} \sum_{j \in N(i)} \frac{1}{\sqrt{\tilde{d}_j}} \vec{f}_j^{(l)} \tag{3}$$

$$\vec{f}_i^{(l+1)} = \sum_{j \in N(i)} \frac{1}{\sqrt{\tilde{d}_i \tilde{d}_j}} \vec{f}_j^{(l)} \tag{4}$$

$$\boldsymbol{f}^{(l+1)} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\boldsymbol{f}^{(l)} = \hat{A}\boldsymbol{f}^{(l)} \tag{5}$$

$$\tag{6}$$

We obtain our final feature vector (Equation 8) through multiplication of our input features by a learned weight matrix, $W^{(l)}$ and the application of an activation function, σ, of the overall message.

$$f_i^{(l+1)} = \sigma\left(\frac{1}{\sqrt{\tilde{d}_i}} \sum_{j \in N(i)} \frac{1}{\sqrt{\tilde{d}_j}} W f_j^{(l)}\right) \tag{7}$$

$$\boldsymbol{f}^{(l+1)} = \sigma(\hat{A}W\boldsymbol{f}^{(l)}) \tag{8}$$

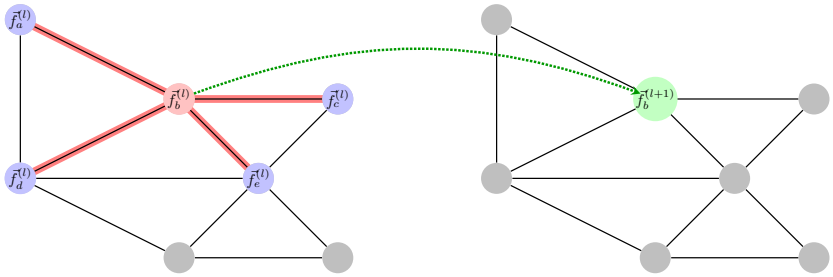

Figure 1: Diagram of the operation of a Graph Convolution Layer. On the left we have all neighbouring nodes and the node itself from layer l comprising the message, $\vec{f}_j^{(l)}$, (indexed by j in the equations). On the right we have the output of a single feature vector, $\vec{f}_i^{(l+1)}$, for node i for the following layer $l+1$

4 Graph Attention Networks (GAT)

For a GCN, the message passing process may be decomposed. We see that the message from each neighbouring node is weighted according to the degree of the

neighbour d_j, and node i, d_i:

$$\vec{f}_i^{(l)} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij} W \vec{f}_j^{(l)}\right) \tag{9}$$

$$\alpha_{ij} = \frac{1}{\sqrt{\tilde{d}_i \tilde{d}_j}} \tag{10}$$

A graph attention network (GAT) makes use of an attention mechanism to assign a priority to features on neighbouring nodes. This is done through altering the weights during message passing phase, making them depend on the features of node i and j respectively, $\alpha_{ij} = \text{softmax}(a(\vec{f}_i, \vec{f}f_j))$. These values may be learnt, allowing the network to learn how to aggregate the neighbourhood data. The activation function for our attentions is usually $\sigma = \text{LeakyReLU}$.

$$\alpha_{ij} = \vec{f}_i \cdot \vec{f}_j \tag{11}$$

$$\alpha_{ij} = \text{softmax}\left(\sigma\left(\vec{a}^T \cdot [W\vec{f}_i || W\vec{f}_j]\right)\right) \tag{12}$$

We may further extend this model into multi-headed attention, where we concatenate K resulting feature vectors together, each with a different attention matrix, α_{ij}^K, (Equation 14).

$$\vec{f}_i^{(l)} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij} W \vec{f}_j^{(l)}\right) \tag{13}$$

$$\vec{f}_i^{(l)} = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^K W^K \vec{f}_j^{(l)}\right) \tag{14}$$

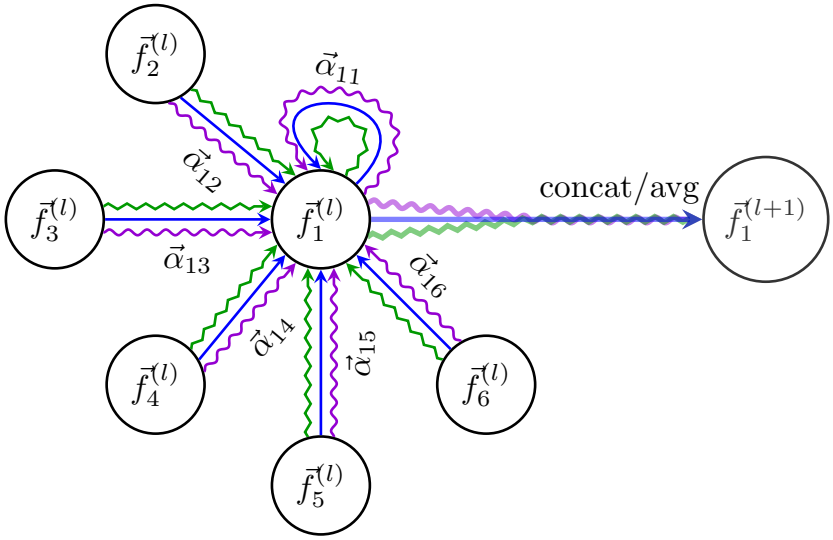

Figure 2: Diagram of the operation of a Graph Attention Layer. We can clearly see the learned attention weights, α_{ij}, with j varying for calculating the output feature of node i.

5 Method

We perform node prediction of the CORA dataset[2], also known as the CORA citation network. The CORA dataset consists of two files, `cora.cities`, and `cora.content`. The first file consists of `(target, source)` id pairs. We consider an undirected graph and the edges in this file are used to construct our adjacency matrix. The second file contains our features for each node, the first column is the node index, the next `1433` columns of an index represents a 1433-sized feature vector, with a 1 or 0 corresponding to the presence or abscence of a certain word in each paper from a dictionary. The final column is the category of paper, with a total of 7 possible classes. There are a total of `2708` rows, and therefore nodes in this graph.

We design and train a two-layer GCN and GAT, with the same number of features in our hidden layer. For the GAT we make use of multi-headed attention in the first layer. Our training data is selected as follows: We select 20 samples for each of the 7 classes, with 140 samples for our training data. During the process of training, we select 500 other samples for our validation dataset. We then select 1000 other samples (not used for training) for our test dataset. We do this to ensure the network isn't overfitting. We perform a total of 200 epochs of training for each network, setting identical parameters of learning rate and hidden layer feature size.

The GCN and GAT neural network layers are implemented from scratch in `PyTorch`[5]. PyTorch uses standard methods to automatically calculate gradients and perform backpropagation without the need for an explicit implementation of the algorithm provided the functions applied are differentiable. This greatly improves development time. The GCN and GAT layers can be seen in `GCNLayer.py` amd `GATLayer.py` respectively.

6 Architecture

Both our GCN and GAT consist of two layers, the architecture of each may be seen in the file `model.py`. Our first layer has inputs of the form `(N, F_in)`, corresponding to `F_in = 1433` input features for each node where `N = 2708`. Our first layer has outputs of the form `(N, F_hidden)`, where `F_hidden = 16` is a configurable parameter.

In the case of the GAT model, we use a GAT Layer instead of a GCN Layer, we make use of *multi-headed attention*, `K = 8`, where within the first-layer of the model we apply 8 GAT units in parallel, each with a seperate α_k learned weight matrix, and proceed to concatenate the results. As such, our first layer has output size: `(N, F_hidden*NUM_CLASSES)`, where `F_hidden <= F_hidden*NUM_CLASSES = 16*7 = 112`.

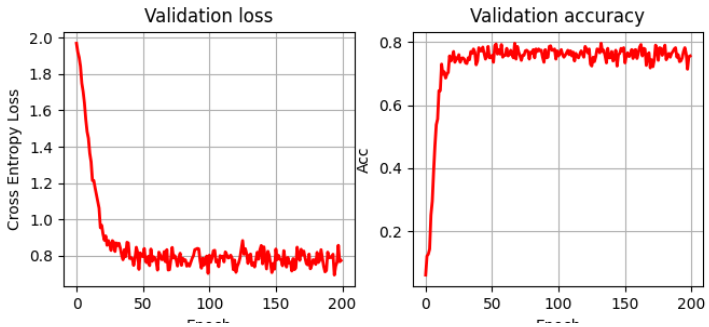

Figure 3: Validation loss and accuracy for each of the 200 training epochs when training the GCN model.

Our final layer consists of another GCN/GAT Layer, with input size: `(N, F_hidden)`, and output size `(N, NUM_CLASSES)`. For the GAT model, we only use a single-headed attention layer, without performing any concatenation of the results.

7 Results and Discussion

Model	Accuracy	Loss	Runtime
GCN	80%	0.655	1.66s
GAT	81%	0.647	2m 52s

Table 1: Caption

We observe marginally better accuracy and cross entropy loss of our test dataset with the GAT model compared to the GCN, but notice the caveat of a significantly greater runtime. This can be explained by inefficiencies in the implementation of the GAT Layer.

The greater variance in validation dataset loss and accuracy in the GAT model (Figure 4) than the GCN model (Figure 3) can be explained.

We obtain t-distributed Stochastic Neighbour Embedding projections (t-SNE projections). From these we observe that the learned features are clustered in a significantly tighter distribution in for the GAT (Figure 6 than the GCN (Figure 5). This is likely due to the attention mechanism being much better at segmenting the data, despite the fact that the loss and accuracy of the GAT is only marginally better than the GCN (Table 1).

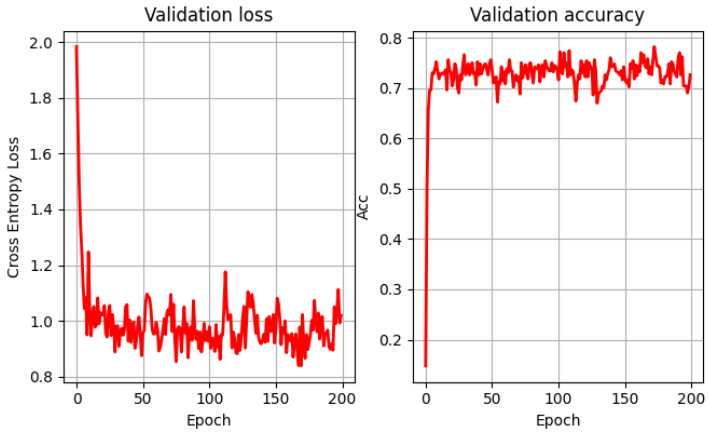

Figure 4: Validation loss and accuracy for each of the 200 training epochs when training the GAT model. We observe greater variance in fluctuations of loss and validation accuracy

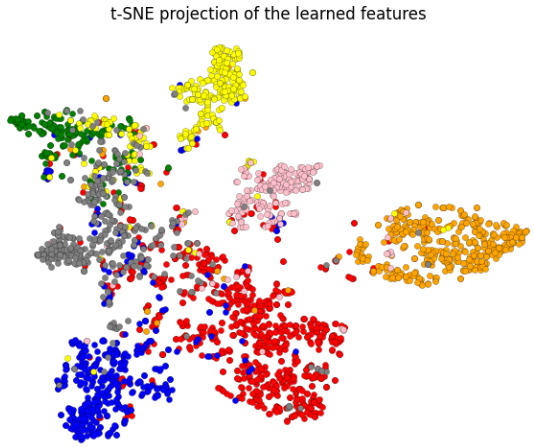

Figure 5: We perform a t-SNE 2d projection of our graph features for the GCN classifier

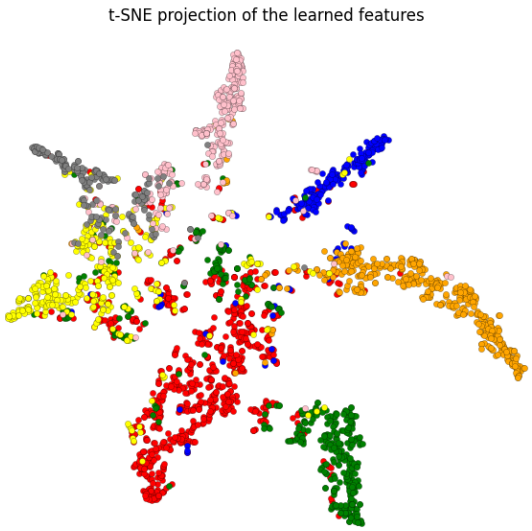

Figure 6: We perform a t-SNE 2d projection of our graph features for the GAT classifier

8 Conclusion

We clearly demonstrate that the Graph Attention Network obtains a state of the art, and marginally better classification accuracy and loss on the CORA dataset. We successfully implement a `GCNLayer` and `GATLayer` from scratch.

We also show the ease, and speed of training a GAT and GCN in the context of using supervised learning for node classification. This can be seen as we have fewer than 40 lines of code for both our `GCNModel` and `GATModel` in `model.py`. This is not straightforward to perform in the case of Graph Signal Processing (GSP) using spectral methods.

Our GAT implementation may be substantially improved. Our GCN is incredibly fast due to the fact that the message passing is computed by storing the normalized adjacency matrix as a `torch.sparse.FloatTensor`, an implementation of a sparse matrix with significantly improved performance offering an order of magnitude speed improvement. If given more time, it would be suitable to implement a GAT layer making use of a sparse adjacency matrix to greatly speed up the message passing step.

The design of our GAT can be further improved, as they suffer from something known as the *static attention* problem[1]. We may modify the order of operations of a GAT and construct a GATv2[1], which makes use of *dynamic attention*. This involves moving the $\vec{a}^T$ parameter out of the activation function, taking the dot product only after the activation has been applied to our concatenated weighted input-output features, as follows:

$$\alpha_{ij} = \text{softmax}\left(\sigma\left(\vec{a}^T \cdot [W\vec{f}_i || W\vec{f}_j]\right)\right)$$
$$\alpha_{ij} = \text{softmax}\left(\vec{a}^T \cdot \sigma\left([W\vec{f}_i || W\vec{f}_j]\right)\right)$$

References

[1] Shaked Brody, Uri Alon, and Eran Yahav. *How Attentive are Graph Attention Networks?* 2021. DOI: 10.48550/ARXIV.2105.14491. URL: https://arxiv.org/abs/2105.14491.

[2] *CORA Dataset from the Relational Dataset Repository*. 2023. URL: https://relational.fit.cvut.cz/dataset/CORA.

[3] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2016. DOI: 10.48550/ARXIV.1609.02907. URL: https://arxiv.org/abs/1609.02907.

[4] Antonio Ortega et al. *Graph Signal Processing: Overview, Challenges and Applications*. 2017. DOI: 10.48550/ARXIV.1712.00468. URL: https://arxiv.org/abs/1712.00468.

[5] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] Ljubisa Stankovic, Milos Dakovic, and Ervin Sejdic. "Introduction to Graph Signal Processing". In: Jan. 2019, pp. 3–108. ISBN: 978-3-030-03573-0. DOI: 10.1007/978-3-030-03574-7_1.

[7] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[8] Petar Veličković et al. *Graph Attention Networks*. 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.