ADL - Lab session 3

Quentin Blampey

February 15, 2021

1 Introduction

I implemented my own GAT model and trained it on the given data. It is slightly better that the Basic Graph Model, but I think that it could be way better. Indeed, I needed to train my model using GPUs, but Google Colab has low memory limits. Thus I had to choose a low number of heads.

This implementation was really efficient for GPUs, and it took about 3 seconds per epoch, which was way faster than the DGL implementation of GATs.

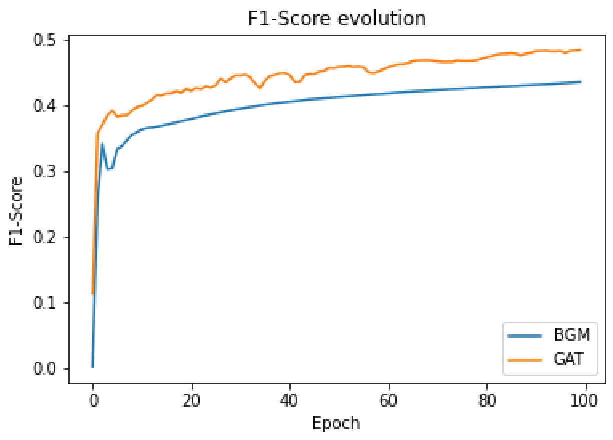

Figure 1: Loss evolution comparison between the BGM baseline and my GAT implementation

2 Hyperparameters

I have chosen the following hyperparameters. 100 epochs, a batch_size of 2, a learning rate of 5.10^{-4}, 4 heads, 2 multi-head layers, and a hidden dimension of

512. I wanted to increase the number of head to 8, but it reached the memory limit on Colab because of the parallelization of the calculus (about 8 GB in total).

3 Attention Implementation

There is no for in my forward, only matrices, in order to take full benefits of PyTorch efficiency. For instance, I created a matrix of size $n\_nodes \times (2 * out\_dim)$ where a line is the concatenation of two node features. Thus, it can be applied into the attention PyTorch module directly.

4 Similarity vs utility attentions

Please refer to the GAT diagram I created (Gat Model.pdf).

Similarity and utility attentions are actually very similar. "Attention is all you need" and "GAT" both consist on multiple layers of multiple heads of attention. The main difference is the attention matrix definition.

Both models transforms the inputs into a features matrix (V and h respectively), and multiply it by an attention matrix. The attention matrices are the following.

- Similarity attention: two linear layer are used to compute two matrix K and Q. Then, they are multiplied to each other, scaled, (a mask can be applied at this moment), and put into a softmax.
- Utility attention: Instead of having K and Q, it uses h (i.e. V for similarity attention). A weight is thus calculated given a linear (vector) layer for every pair of node features. Computing weights only for node neighbours is actually just like applying the adjacency matrix as a mask. Again, it's put into a softmax. There is a LeakyReLU in between.

In brief, there is QK^T on the one hand and $a(Wh_i||Wh_j)_{i,j}$ on the other hand. The following steps are similar. Utility attention has a specific mask (adjacency matrix), and add a LeakyReLU. The idea is very similar, but Similarity attention is slightly more complex, and there are also more parameters.

4.1 Interpretation

- Similarity attention: two matrices are learned such as Q and K corresponds to a query and a key. The key is applied on the query so it returns (after few other operations) relevant weights to be applied on the values.
- Utility attention: it computes weights for every edge, and is based on both node features. An edge weight is simply based on these features and a vector (less complex).