TP3 Homework Submission

Vincent, Wilmet
vincent.wilmet@student-cs.fr

February 22, 2022

1 Graph Neural Networks

We are given a basic Graph Convolutional and the Protein-Protein Interaction (PPI) network dataset for benchmarking. One graph of the PPI dataset has on average 2372 nodes. This multilabel classification problem has 121 gene ontology sets (way to classify gene products like proteins) that must be classified using 50 features (positional gene sets / motif gene / immunological signatures, etc).

1.1 Training Process, Hyperparameters and GridSearch

Initially, without any modifications to the given code, the model performs very poorly. Although it doesn't gives NaN values, nor errors, the model reaches an average F1 score of 0.51 on 20 trials.

With this in mind, I decided preprocess the data for better model performance. I did this by trying sklearn methods like MinMaxScaler, StandardScaler and normalize with l1, l2, and max norms. This later showed to be unhelpful with larger models. Furthermore, after reading the papers describing the PPI dataset published by Stanford computational neuroscientists, I understood I was far out of my domain of expertise and if they curated the dataset to be a certain way, it should stay a certain way. At least until I learned what is malleable.

The following hyperparameters are applied to this problem. Because GridSearch is not implemented in PyTorch, I used a very oldschool technique: nested for-loops for each of the hyper-parameters listed above. I used early stopping to avoid time wasted training on a poorly performing model.

```
########## Replace this model with your own GNN implemented class ############################
hiddens = [4, 16, 64, 128, 256, 512, 1024]
heds = [1, 2, 3, 4, 8, 16, 64]
act = [F.relu, F.elu, F.leaky_relu, F.softmax, F.tanh, F.sigmoid]
fdrop = [0, 0.1, 0.5, 0.6]
adrop = [0, 0.1, 0.5, 0.6]
nslope = [0.01, 0.1, 0.2]
losses = [nn.BCEWithLogitsLoss(), nn.CrossEntropyLoss, nn.NLLLoss, nn.KLDivLoss, nn.BCELoss, nn.HuberLoss, nn.MSELoss]
optimizers = [torch.optim.SGD, torch.optim.Adadelta, torch.optim.Adagrad, torch.optim.Adam, torch.optim.LBFGS, torch.optim.RMSprop]

final = []
i = 0
for size in hiddens:
  for h in heds:
    for a in act:
      for f in fdrop:
        for attn in adrop:
          for n in nslope:
            for loss in losses:
              for optimizer in optimizers:
                model = GAT(g=train_dataset.graph, num_layers=2, in_dim=n_features,
                                    num_hidden=size, num_classes=n_classes, heads=[h,h,h],
                                    activation = a, feat_drop =f, attn_drop =attn,
                                    negative_slope=n, residual=False).to(device)
                ###############################################################################################
```

In all there are 7*7*6*4*4*3*7*6=592,704 possible configurations. Of course, after nearly 30 hours, I had exceeded the monthly allocation for Google Colab's GPUs and I understood that doing all combinations would be infeasible. It was taking about 0-4 minutes to run each model depending if it stopped at 10, 50, 100, 150 or 250 epochs.
I tried a variation of each non-numeric parameters (activation functions, loss functions and optimizers) to find that the best activation functions were ReLU and eLu, loss functions limited to BCEWithLogitsLoss and MultiLabelSoftMarginLossm and only Adam for optimizer. Everything else either crashed or rarely surpassed a 60% F1 score. AdaGrad did surprisingly well, and CrossEntropyLoss was surprisingly poor performing. I learned that the high drop out rates would do well in the beginning of training but cause the model to be erratic by the end of 250 epochs. And as much as I think 16+ heads and 1024 hidden layers would make the model perform better, Colab could not handle the size of the models loaded to memory.
After limiting the configurations, I received the following results for best performing models. Unsurprisingly, the ones with highest head size and hidden layers top the list.

	num_hidden	heads	activation	feat_drop	attn_drop	negative_slope	loss_fcn	optimizer	score
15	512	8	relu	0	0	0.2	MultiLabelSoftMarginLoss	Adam	0.987170
14	512	8	relu	0	0	0.2	BCEWithLogitsLoss	Adam	0.984546
13	512	8	elu	0	0	0.2	MultiLabelSoftMarginLoss	Adam	0.982777
12	512	8	elu	0	0	0.2	BCEWithLogitsLoss	Adam	0.981702
6	256	8	relu	0	0	0.2	BCEWithLogitsLoss	Adam	0.977528
5	256	8	elu	0	0	0.2	MultiLabelSoftMarginLoss	Adam	0.977041

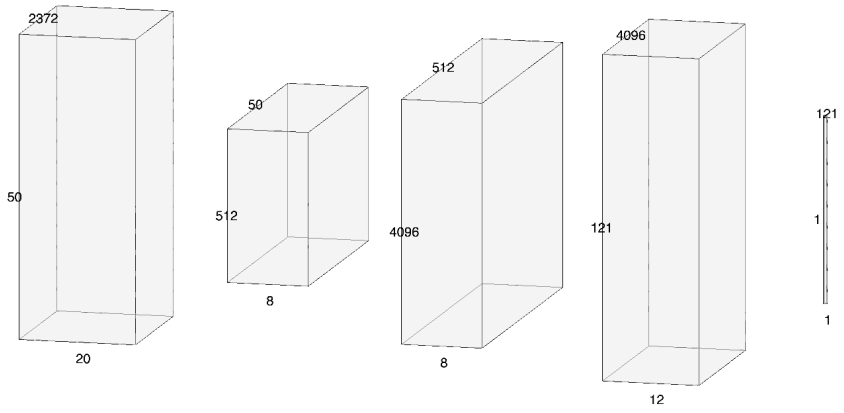
1.2 Model Architecture

The final model uses the following hyperparameters:

```
hiddens = [512]
heds = [8]
```

```
activations = [F.relu]
fdrop = [0]
adrop = [0]
nslope = [0.2]
losses = [nn.MultiLabelSoftMarginLoss()]
optimizers = [torch.optim.Adam]
```

and looks like this:



Input:	Layer 1: GATConv	Layer 2: GATConv	Layer 3: GATConv	Output:
50 Features	50 Features	512 Hidden Layers	4096 Hidden Layers	121 Class
2372 Nodes	512 Hidden Layers	4096 Hidden Layers	121 Class Labels	Labels
20 Graphs	8 Heads (Attention)	8 Heads (Attention)	12 Heads (Attention)	Flattened

The below graph shows the GAT attention module in detail as described by the original authors Velickovic et al in their 2018 paper GRAPH ATTENTION NETWORKS.

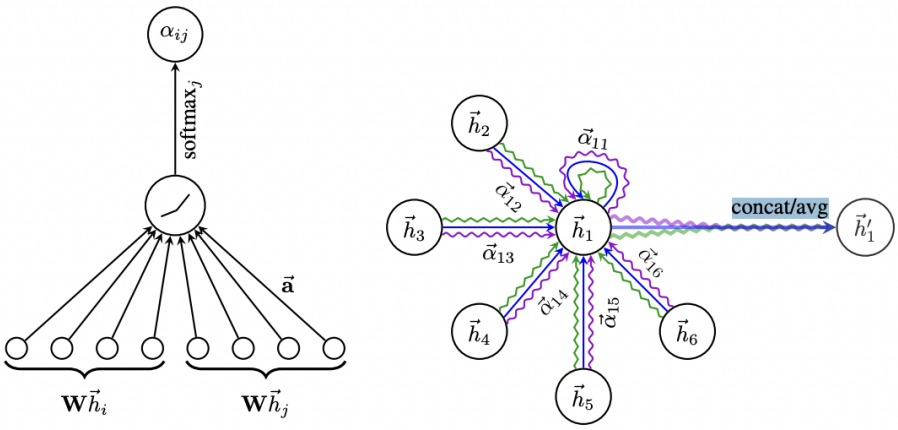
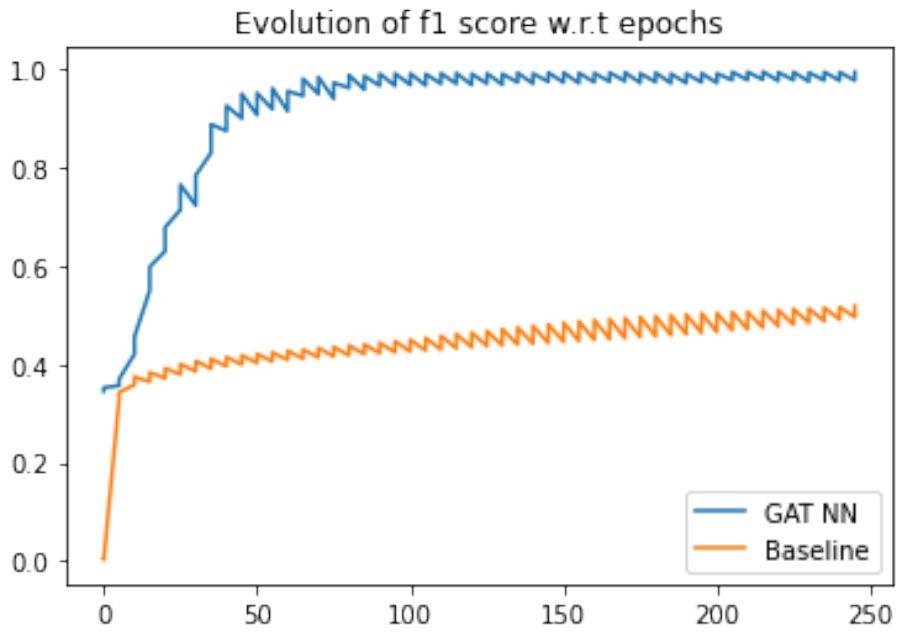


Figure 1: **Left:** The attention mechanism $a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$ employed by our model, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, applying a LeakyReLU activation. **Right:** An illustration of multi-head attention (with $K = 3$ heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\vec{h}'_1$.

1.3 Performance and Results

Ultimately, the best performing model reached heights of 0.9956994061645773, but on an average of 20 trials had an **F1 Score of 0.9865**. The reason it is not consistent, even though the GATConv uses a uniform Xavier initializer for its weights, is the Adam optimizer starts in a random corner each time the model is trained.

We can see that it vastly out performs the baseline model, which rarely surpassed a F1 Score of 0.50 on multiple trials. This is because multi-headed Graph Attention models are able to capture the relationships between the Protein-Protein Interactions (PPI) much better than a Convolutional network can.



What we can learn from this model is that the Graph Attention model uses an "attention" mechanism to focus on the most important nodes within the graph network. They do this by assigning larger weights to more important nodes, which are found by looking at the relative weights between two neighboring connected nodes.
This is in contrast to the Graph Convolutional model which reassigns its attention weights in every walk. This makes the weights between each connected pair different for each layer and thus the mutual dependencies are a lot harder to identify.
GAT models, on the other hand, rapidly enhances the rate at which important nodes are identified and learned, at the expense of computing for each node combination.