GRAPH MINING PRESENTATION

AN OVERVIEW OF THE TASKS:

1. Add a semi supervised node classification model on the top of a pretrained SBERT model….(We used a basic GCN model with 2 hidden layers)
2. Create a new loss function in order to generate better embeddings from the sentence bert model which inherits significant graph structure based on their classes.
3. Understand what the final layer does and how does the new loss function help?

DATASET: R8 toy dataset

The dataset consists of a variety of sentences along with their respective class.

Total no of classes:8

Train: 490 sentences:

```
[48] class_counts/torch.sum(class_counts)

tensor([0.2918, 0.0449, 0.5224, 0.0061, 0.0347, 0.0367, 0.0184, 0.0449],
       device='cuda:0')
```

Validate: 53 sentences:

```
[50] class_counts/torch.sum(class_counts)

tensor([0.3019, 0.0377, 0.5094, 0.0189, 0.0189, 0.0377, 0.0189, 0.0566],
       device='cuda:0')
```

Test: 216 sentences

```
[52] class_counts/torch.sum(class_counts)

tensor([0.3194, 0.0556, 0.5000, 0.0046, 0.0370, 0.0370, 0.0139, 0.0324],
       device='cuda:0')
```

TASK-1:

We want to talk about the typical training process for the task:

1. Finetune the sentence bert transformer using the cosine similarity loss to generate embeddings. The embeddings at this stage do not have any inherent graph structure.(Freeze the embeddings at this stage)
2. Design a GCN model with 2 hidden layers.
    a. Design a similarity matrix for the gcn adjacency using cosine similarity between the embeddings
    b. Pass the embeddings through the model to obtain logits and identify the cross entropy loss.
    c. Perform backpropagation and optimizer step only on the GCN for this task and observe accuracy.

Learning rate= 0.001
Hidden neurons=256
input_layer->256->256->output_layer

SBERT + MLP

```
[84] acc_train, acc_val, acc_test

(tensor(0.5224, device='cuda:0', dtype=torch.float64),
 tensor(0.5094, device='cuda:0', dtype=torch.float64),
 tensor(0.5000, device='cuda:0', dtype=torch.float64))
```

Learning rate= 0.001
Weight decay = 0.0005
Hidden neurons = 8(tried 256)
Dropout = 0.1
(input_layer->hidden_layer(1)->output_layer)

SBERT + GCN

```
[84] acc_train, acc_val, acc_test

(tensor(0.5224, device='cuda:0', dtype=torch.float64),
 tensor(0.5094, device='cuda:0', dtype=torch.float64),
 tensor(0.5000, device='cuda:0', dtype=torch.float64))
```

- The results are same because it is predicting class 2(between 0-8) which is inherent from the bias in the dataset consisting of nearly 50% of the data from class 2
- Changing any sort of hidden layers did not work and gave the same output.

TASK-2:

I will discuss the typical training process here:

- Pass the sentences into the s-bert to get embeddings.
- Calculate the similarity matrix according to the formula for the adjacency of the gcn l

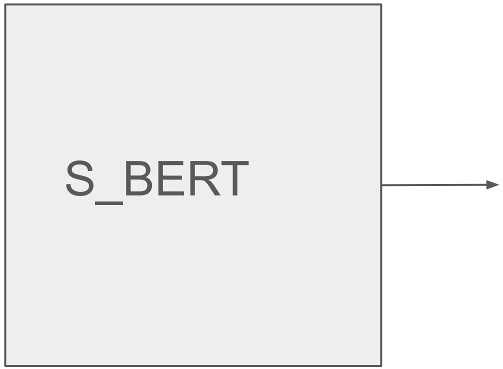
$$S_{ij} = g(x_i, x_j) = \frac{\exp(\mathrm{ReLU}(a^T|x_i - x_j|))}{\sum_{j=1}^{n} \exp(\mathrm{ReLU}(a^T|x_i - x_j|))} \quad (4) \quad \mathcal{L}_{\mathrm{GL}} = \sum_{i,j=1}^{n} \|x_i - x_j\|_2^2 S_{ij} + \gamma \|S\|_F^2$$

- Pass the embeddings throught the gcn to obtain logits.
- Calculate the loss:
  - Term-1: Cross entropy loss
  - Term-2: Graph learning Loss
  - Term-1+lambda(Term-2) : lambda used in the paper :5e-3(worked decently well)
- Backpropagate the loss through GCN and sentence Bert and perform optimizer step to both SBERT and GCN(both have same learning rate)

CODE:

```python
class GraphLearningLoss(nn.Module):
    def __init__(self):
        super().__init__()
        self.a = nn.Parameter(torch.randn(size=[1, 384])).to(device)  # Learnable parameter

    def calculate_sim_matrix(self, embeddings):
        # Calculate the pairwise absolute difference between embeddings
        # Use broadcasting for efficient computation
        diff = embeddings.unsqueeze(1) - embeddings.unsqueeze(0)  # [N, N, D]
        abs_diff = torch.abs(diff)  # Absolute difference

        # Compute the similarity matrix
        sub_emb = abs_diff.permute(0, 1, 2).reshape(-1, embeddings.shape[-1]).T  # [D, N*N]
        sub_emb = sub_emb.to(device)
        temp = torch.exp(F.relu(self.a @ sub_emb))  # [1, N*N]
        sim_matrix = temp.reshape(len(embeddings), len(embeddings))  # [N, N]

        # Normalize the similarity matrix
        sim_matrix = sim_matrix / sim_matrix.sum()
        return sim_matrix

    def forward(self, embeddings):
        # Calculate similarity matrix
        my_sim = self.calculate_sim_matrix(embeddings).to(device)

        # Compute the loss
        diff = embeddings.unsqueeze(1) - embeddings.unsqueeze(0)  # [N, N, D]
        norms = torch.norm(diff, dim=-1).to(device)  # [N, N]
        my_loss = (norms * my_sim).sum()  # Weighted sum of norms
        return my_loss
```

TYPICAL ARCHITECTURE:



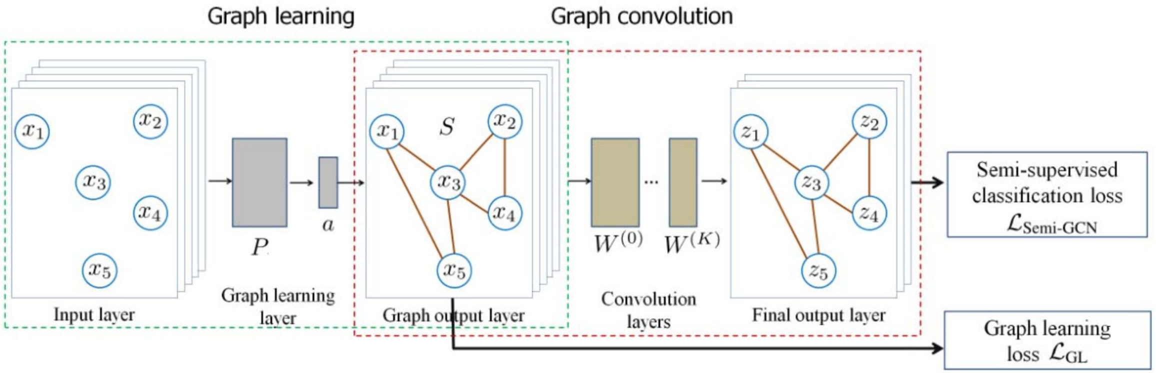


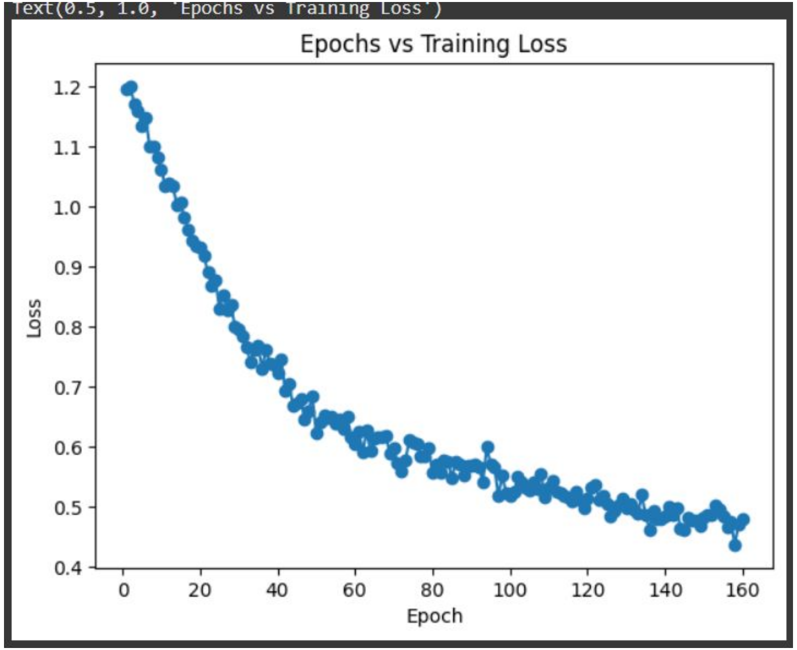
Figure 1. Architecture of the proposed GLCN network for semi-supervised learning.

Backpropagate

GRAPH ADAPTIVE LOSS:

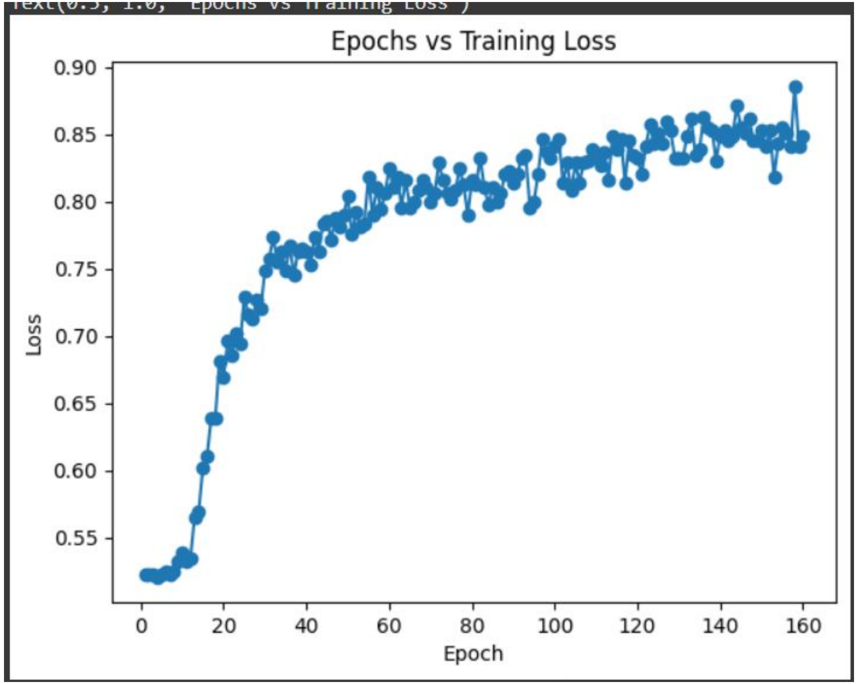
- The loss function basically punishes those embeddings which belong to the same class(Sij=1) but have embeddings very far away.
- The parameter a is used only during the training process and is not required during inference because the embeddings seem to have the inherent graph structure.
- There is a regularization term inorder to ensure that Sij nearly zero becomes equal to zero and induces sparsity.
- The loss value decreases comparatively slowly and requires significant training to converge(90 epochs)(high computation)(converges very slowly after giving 82% accuracy)

LOSS CURVE:



Text(0.5, 1.0, 'Epochs vs Training Loss')
Epochs vs Training Loss
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
Loss
0
20
40
60
80
100
120
140
160
Epoch

ACCURACY CURVE:



SOME RESULTS:(Any Questions?)

I believe GCN starts working well because the similarity matrix with the new loss function generated better graph adaptive embeddings that worked well.

```
acc_test
```

```
tensor(0.8426, device='cuda:0', dtype=torch.float64)
```

```
from torcheval.metrics.functional import multiclass_f1_score
multiclass_f1_score(output[len(X_train)+len(X_val):len(X_train)+len(X_val)+len(X_test)], y_test[:], num_classes=8, average=None)
```

```
tensor([0.8921, 0.5714, 0.9292, 0.0000, 0.5000, 0.3077, 0.0000, 0.4615],
       device='cuda:0')
```

OBSERVATION OF INFERENCE ACCURACY WITH RANDOM a vs NON RANDOM a(parameter for graph learning loss):

The observation shows that irrespective of a the embedding have learnt an inherent graph structure that generalizes really well.

```
acc_test_without_random_a

tensor(0.8657, device='cuda:0', dtype=torch.float64)
```

```
[72] acc_test_with_random_a

tensor(0.8333, device='cuda:0', dtype=torch.float64)
```

What more can I try?

- I can try giving different learning rates to sbert and gcn to maybe give better results.(I tried but the gpu was being drained for every change so had some constraints)
- The class 4 and class 7 were not being predicted at all(The dataset had less samples of these classes so maybe increase samples using some augmentation) or maybe I can increase complexity of architecture.

CREDITS:

Semi-supervised Learning with Graph Learning-Convolutional Networks

Bo Jiang, Ziyan Zhang, Doudou Lin, Jin Tang* and Bin Luo
School of Computer Science and Technology, Anhui University, Hefei, 230601, China
jiangbo@ahu.edu.cn,{zhangziyanahu,ahu_lindd}@163.com,ahhftang@gmail.com,luobin@ahu.edu.cn