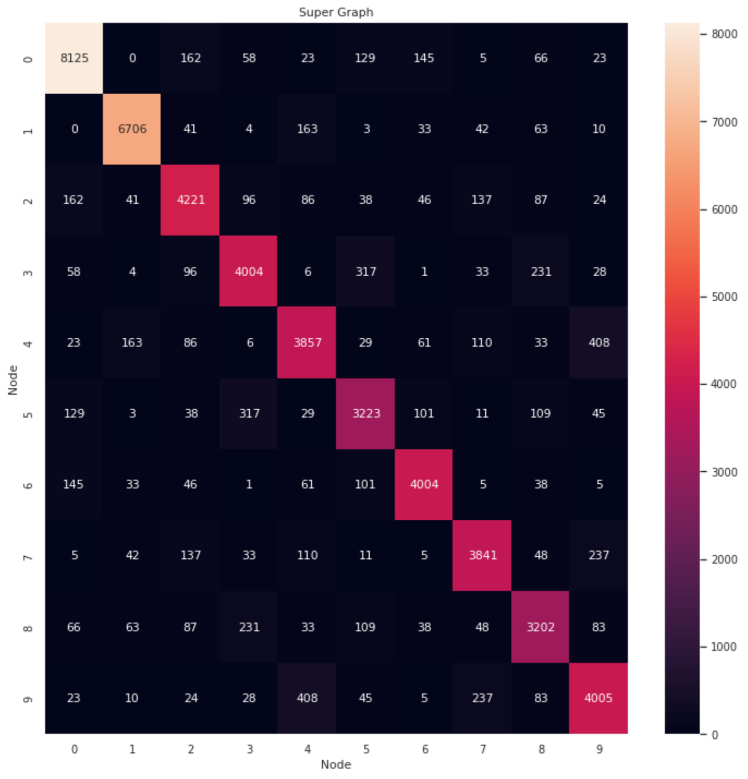
Brendan Schneider
CIS5930
7/21/22

Programming Assignment - Report

Overview

In this assignment, we were tasked with designing three kinds of neural network models, and then performing various classification tasks with those models. My implementations were done in Python, making use of several common libraries, most notably the Deep Graph Library (DGL). Data was provided in separate training and test files, though converting these files into usable data structures proved a challenge on its own. A DGL graph was constructed using this data, where the Euclidean distance between nodes (represented by 256-feature arrays - greyscale images of digits) was used to find the 7 nearest neighbors to each node and create bidirectional edges between them. This section has been commented out of my code because producing the edge list takes 20-30 minutes, and there is instead a section where an edge list file can be loaded from the current directory. This file is provided in the .zip as ‘edgelist.txt’. The original edge list creation can still be run by commenting the lines back in.

Below is a “super graph” that represents the data found in the DGL graph, where each edge is marked on the super graph based on the ground-truth labels of the nodes it connects. For example, an edge connecting a ‘0’ and a ‘2’ would appear at both (0, 2) and (2, 0) in the super graph below. Most neighbors of a given node should be of the same class, and those that are not likely look similar, meaning that digits that look similar will have a high of “off-diagonal” representation. The highest off-diagonal values in the super graph are between ‘4’ and ‘9’, ‘3’ and ‘5’, ‘7’ and ‘9’, and ‘3’ and ‘8’. When handwritten these digits do indeed look similar, so they will be the focus of our analysis below, since they are the samples most likely to be mistaken by our models.



I actually enjoyed doing this project, despite how long it took me to finish. In the following two sections, we will discuss the specifics of each model - as well as the results reported by each - then compare these results in hopes of gaining a better understanding of their differences in practice.

Task I

For all three parts of task I, I tried to make simple networks in hopes that they would reveal clearer differences between the methods. As it turns out, they all ended up being fairly accurate, despite being simple. Below is an explanation of each network.

Convolutional Neural Network:

The Convolutional Neural Network does not make use of the graph itself. Instead, it learns its own set of features that are representative each input image and classifies the inputs that way. My CNN was fairly straightforward, with two 3x3 convolution layers separated by a 2x2 max pooling layer. The image was then flattened and passed through 2 dense layers with the expected output size of 10 features. All layers used a RELU activation function except the final dense layer, which used softmax.

As mentioned above, I had hoped that simple networks would allow for the differences and weaknesses of each method to be more noticeable. As will be shown in task II, however, the CNN performed very well - being close to or on par with the graph-based methods.

Spectral GNN:

Perhaps the simplest and least computationally intensive spectral GNN - especially when using DGL - I went with a GCN model for this part of the assignment. In particular, it is a 3-layer GCN with RELU activation functions between layers. Nodes begin with 256 input features, then drop to 100 and then 25, before ending with the expected 10 output features. Finally, the Adam optimizer was used with a learning rate of 0.005.

I wanted to see if a 3-layer GCN model would be any more accurate than the minimum 2 layers that was mentioned in the assignment. For my implementation, at least, the improvement seemed minimal at best, but I kept the third layer in case it exacerbated any key features of this method (faster oversmoothing, faster learning, etc.).

Spatial GNN:

For my spatial GNN, I chose to implement GAT, partially because of its resemblance to the above GCN method. Also similar was the structure of my model, with the same 3-layer/RELU implementation, similar aggregation of features, and same use of the Adam optimizer. Though modern GAT models can use multi-head attention, I only used a single head in mine.

My choices here were partially fueled by ease of implementation, but I was also curious as to how this model would compare to my spectral model. The main difference between the two is in how the importance of a node is calculated (if at all), but much of the rest of the process is the same. With how similarly the models operate, I would expect a very similar set of results as well.

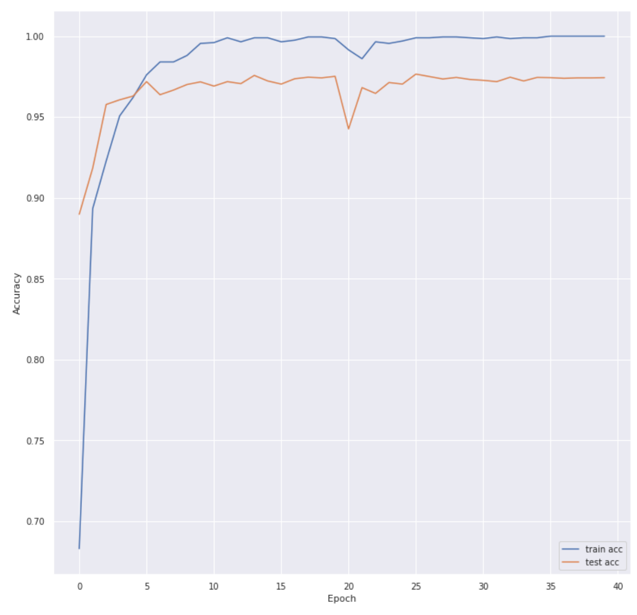
Task II

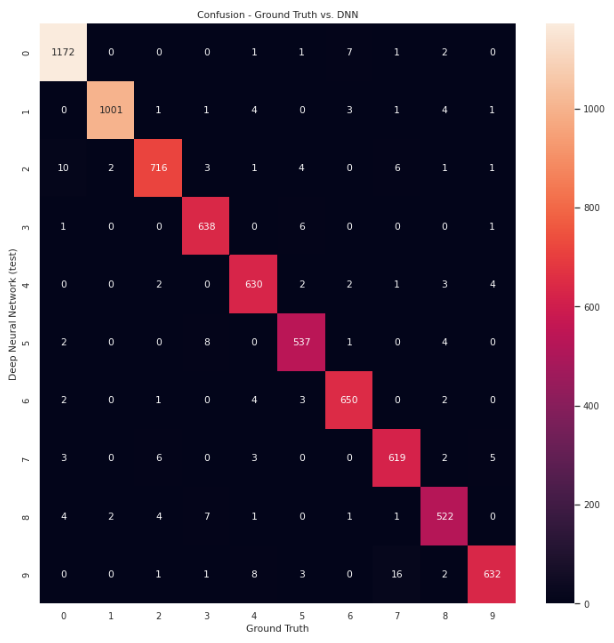
Here is where the models described above were put into use. The subsections below will discuss the results of each of the four tasks assigned (except for task 2, which I did not complete), with the final section being dedicated to analyzing the results of each task in relation to the others.

Deep Learning:

Even the simplest of the models - the DNN - did very well on the given datasets. The deep learning model ran for 40 epochs, with the training accuracy converging at 1.0 after about epoch 25. Though this might have suggested overfitting, the test set accuracy came out to a solid 0.97 (loss = .1088) and had not yet begun to fall when the simulation finished. Of those predictions that were incorrect (shown by the confusion matrix below), it seems that the most common misrepresentations were between the pairs (7, 9) and (2, 0)

The former is one of the possible problem pairs mentioned above, and the latter was also high on the list of possible misclassifications. Also, while the model would sometimes misclassify a 7 as a 9, it would rarely classify a 9 as a 7. I would be interested in discovering why this model was able to easily differentiate the other problem pairs but had trouble with these in particular, as well as why the misclassifications are not symmetric.





	0	1	2	3	4	5	6	7	8	9
0	1172	0	0	0	1	1	7	1	2	0
1	0	1001	1	1	4	0	3	1	4	1
2	10	2	716	3	1	4	0	6	1	1
3	1	0	0	638	0	6	0	0	0	1
4	0	0	2	0	630	2	2	1	3	4
5	2	0	0	8	0	537	1	0	4	0
6	2	0	1	0	4	3	650	0	2	0
7	3	0	6	0	3	0	0	619	2	5
8	4	2	4	7	1	0	1	1	522	0
9	0	0	1	1	8	3	0	16	2	632

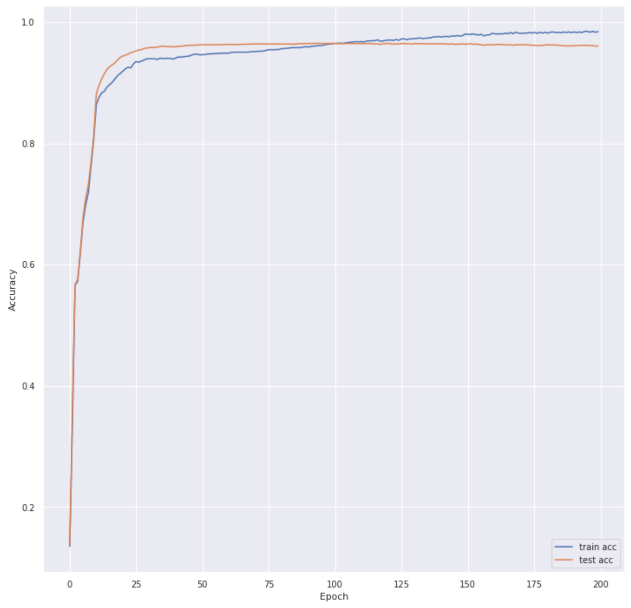
Correct and Smooth:

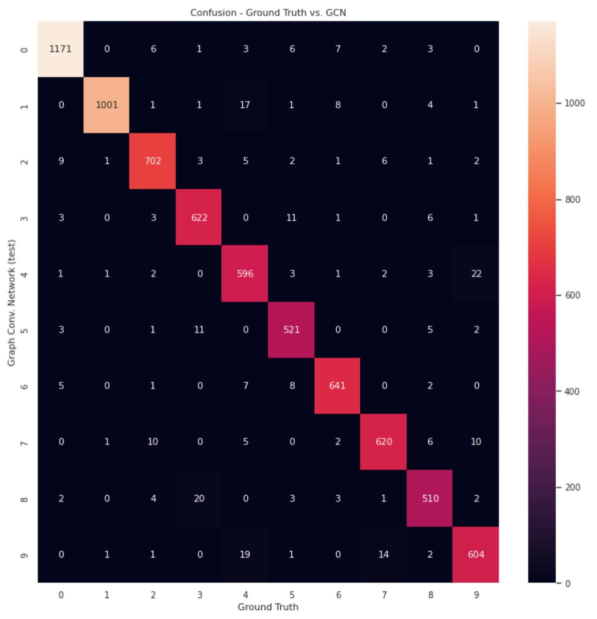
Though I spent a fair amount of time trying to get this to work, I could not quite figure out how. My final attempt, which is an adaptation of another C&S implementation I discovered online, can be found commented out in the code file, with the borrowed class definitions stored separately in ‘CSmodel.py’.

Spectral Training (GCN):

The overall results for GCN were similar to those in the GNN, with training accuracy converging towards 1.0 and test accuracy peaking at 0.964 (loss = .100). Since it ran much faster, I chose to leave the GCN model running for 200 epochs, hoping to see some long-term pattern. The line graph below shows the results of this, where test accuracy actually stays above training accuracy for about 100 epochs, before slowly falling off due to what I suspect is the beginnings of overfitting.

The confusion matrix has a few more off-diagonal values to analyze due to the lower test accuracy, but again the most common mistakes lie with the problem digits seen in the original super graph. In this case, (4, 9), (3, 8), and (7, 9) were the most seen, though some asymmetric pairs such as (1, 4) were also common.

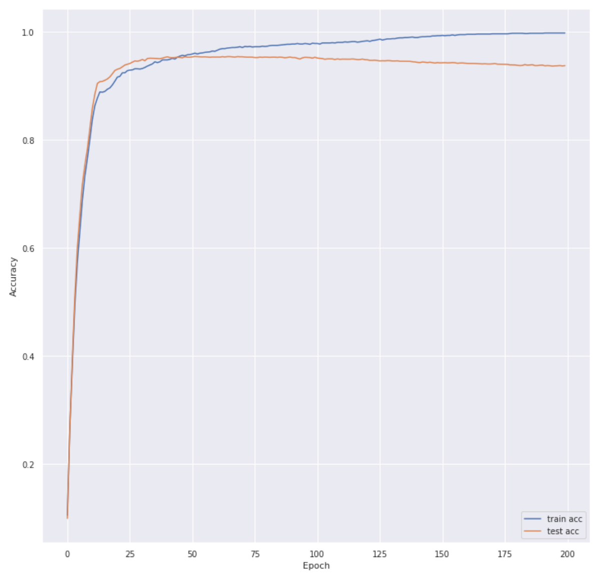


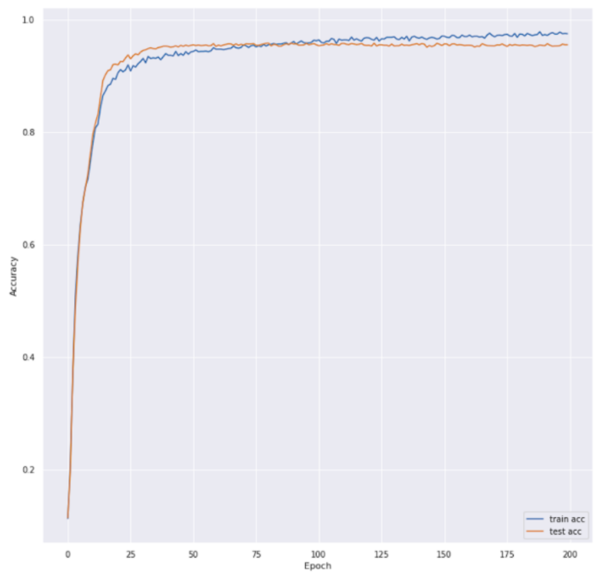


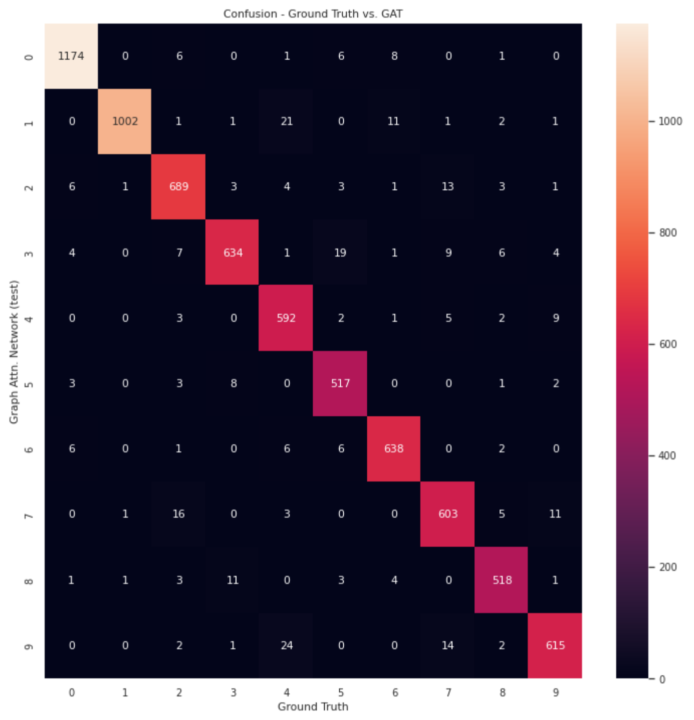
Spatial Training (GAT):

Though it performed the worst of the three methods I implemented, GAT still ended with a very high accuracy, with a training accuracy again approaching 1.0 and a test accuracy of 0.962 (loss = 0.08). I also let GAT run for the full 200 epochs, and the results were a bit more interesting than those for Deep NN or GCN. Notably, the first graph below shows a steep decline in test accuracy after about 30 epochs that doesn't exist in the other models. This, I believe, is due to oversmoothing, which is a common problem for spatial-based GNNs.

There were one or two extra settings to fiddle with in the DGL version of GAT, specifically with regards to this oversmoothing problem. I ended up adding feature dropout (rate = 0.05) to the model after my first run, which significantly reduced this problem, as seen below. The confusion matrix, as with the previous two tasks, shows errors most commonly among problem pairs, namely (3, 5), (9, 4), and (2, 7).



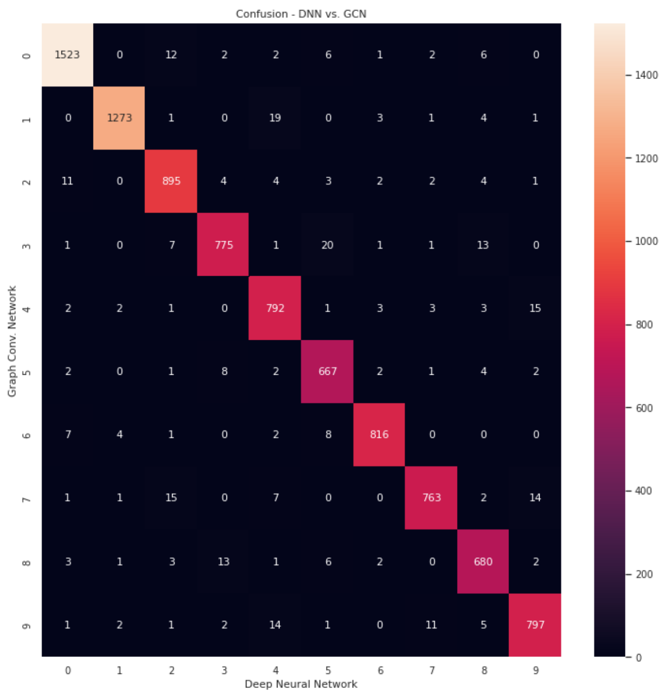


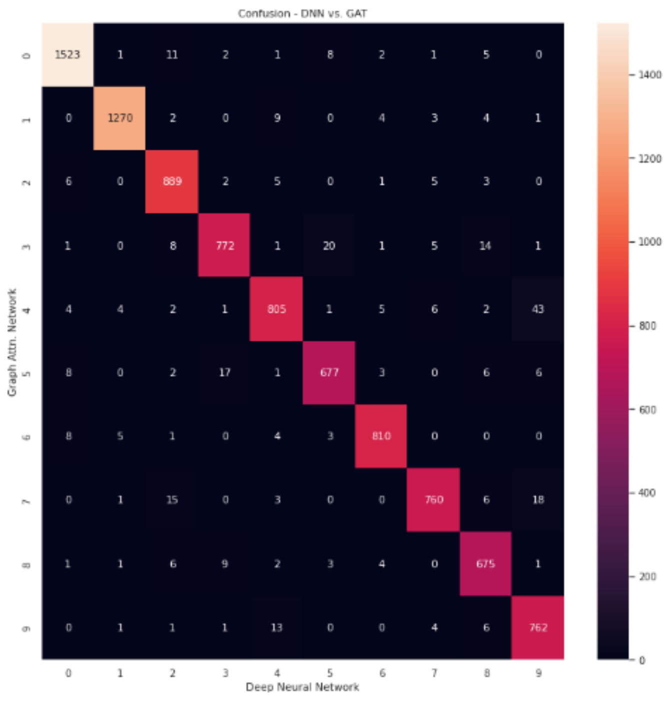


	0	1	2	3	4	5	6	7	8	9
0	1174	0	6	0	1	6	8	0	1	0
1	0	1002	1	1	21	0	11	1	2	1
2	6	1	689	3	4	3	1	13	3	1
3	4	0	7	634	1	19	1	9	6	4
4	0	0	3	0	592	2	1	5	2	9
5	3	0	3	8	0	517	0	0	1	2
6	6	0	1	0	6	6	638	0	2	0
7	0	1	16	0	3	0	0	603	5	11
8	1	1	3	11	0	3	4	0	518	1
9	0	0	2	1	24	0	0	14	2	615

Further Analysis:

To conclude this report, let us consider each of the three methods discussed so far in relation to one another. First, they were all very accurate, ranging from 0.96-0.98 test accuracy. Perhaps more interesting, however, will be to look at what exactly each of them was wrong about. Below are three confusion matrices, each comparing the total prediction set (training + test) between any two of the three methods. Let us consider each individually.

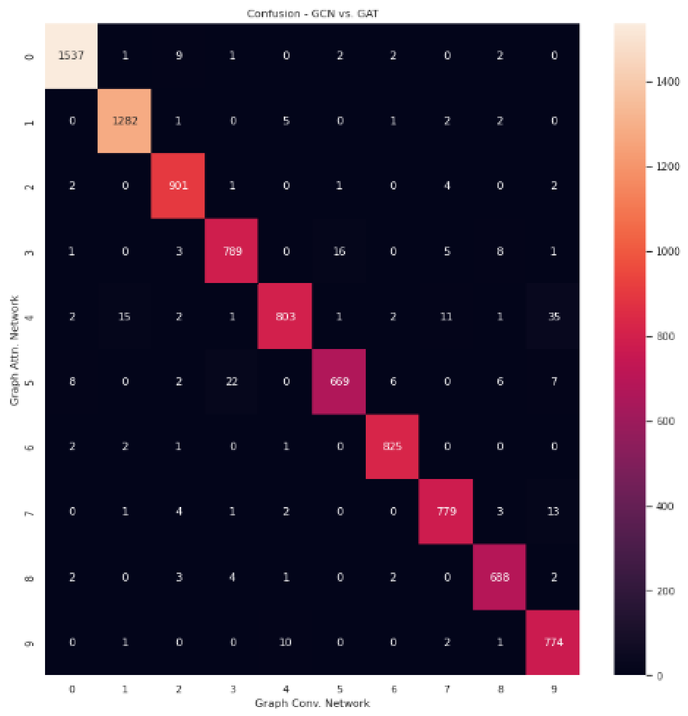




Between DNN and GCN, most of the notable problem pairs are similarly missed by both methods. (7, 9), (4, 9), and (0, 2), for example, have similar rates of false positive and true negative in the matrix. These are likely due to the similarity between the digits making it difficult for any particular method to get all of them right. GCN seemed to have a much broader definition of what a '3' is, though, compared to DNN. We cannot know which method was in the wrong from these matrices, but the above ground truth confusions above tell us that GCN was likely incorrect here. This is also true of (4, 1), where GCN would more often see a '4' as a '1'. Perhaps the spatial nature of GCN (being graph-based) leads to situations where features in the 1's 'cluster' of the graph bleed into the 4's cluster, causing several nearby nodes to be misclassified rather than just the particularly difficult/vague ones that DNN would miss.

Without repeating this sentiment further, a similar patter is seen when comparing GAT to DNN, for what I imagine to be similar reasons. Pairs like (3, 5) and (3, 8) have similar rates of false positive and true negative, while pairs such as (4, 9) seem very lopsided, suggesting GAT and GNN having very different opinions as to what constitutes a 4 or 9.

Comparing GCN and GAT, however, is interesting since both are graph neural networks. Most obvious in this final matrix is that GAT saw 4's as 9's much more often than GCN did. In fact, classifying 4's seems to be the most common disagreement that the two models



have. I would guess that this has to do with how GAT attempts to assign importance to nodes. If the 4 cluster of nodes tends to be more important to the model, it makes sense that more nodes will end up with the features of these nodes and thus the model will predict 4 more often. *Why* 4 is considered the most important I don't really know, but maybe it is the digit that is similar to most other digits (1, 7, 8, 9).

The above are just theories, but they at least seem plausible when considering how each of these models works. I hope that this analysis has been sufficient and well-formatted, despite not having an implementation for Correct and Smooth to discuss in detail.