Graph convolutional networks FOR TEXT CLASSIFICATION

Student Name: Raj/Shray
Meeting time: Every Saturday, 18:00hrs
Meeting Link: https://meet.google.com/kgt-ewbz-qbx
Mobile Number: Raj- 8655749970, 8779608828, Shray - 8826828362
Github link for Source 2: https://github.com/yao8839836/text_gcn

Task:

1. Read details of GRAPH CONVOLUTIONAL NETWORKS for text: Completed
2. Work on the implementation.

Sources

1. https://towardsdatascience.com/simple-scalable-graph-neural-networks-7eb04f366d07
2. Graph Convolutional Networks for Text Classification, DOI: https://doi.org/10.1609/aaai.v33i01.33017370
3. Tensor Graph Convolutional Networks for Text Classification
4. https://towardsdatascience.com/how-to-do-deep-learning-on-graphs-with-graph-convolutional-networks-7d2250723780
5. https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-understanding-kl-divergence-2b382ca2b2a8
6. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73
7. https://towardsdatascience.com/tutorial-on-variational-graph-auto-encoders-da9333281129
8.

Week 1:

Progress report: *Read about GraphSage and GraphSAINT algorithms.*

We look at algorithms that focus on Scalable graphical neural networks.

- Our objective is to move beyond simple graph convolutional networks and scale to larger graphs with millions of nodes.
- It is difficult to train graphical neural networks in mini-batches because the nodes are inter-related via edges, this creates statistical dependence between samples in the training set. This dependence of sampling introduces bias in the form that some nodes may appear multiple times in the train set.

GraphSAGE uses neighbourhood sampling combined with mini-batch training to train GNNs on large graphs.

- Small world type graphs
- Sample K neighbours till the Lth hop.

- Type of supervised feature learning
- We can use different aggregators like LSTM, Max Pooling, Mean

GraphSAINT (Graph SAmpling based INductive learning meThod)

- Focuses on Graph Sampling as opposed to neighbourhood sampling/layer sampling of Graph Sage.
- In graph-sampling approaches, for each batch, a subgraph of the original graph is sampled, and a full GCN-like model is run on the entire subgraph. The challenge is to make sure that these subgraphs preserve most of the original edges and still present a meaningful topological structure.
- GraphSAINT proposes a general probabilistic graph sampler constructing training batches by sampling subgraphs of the original graph.
- Sampling also ensures edge wise dropout ensuring regularisation.

GraphSAINT properties:

1. Accuracy
2. Efficiency: As each GNN layer is complete and unsampled, the number of neighbors keeps constant no matter how deep we go. Computation cost per minibatch reduces from exponential to linear, w.r.t. GNN depth.
3. Flexibility: Supports multiple GNN architectures like GraphSAGE, GAT, JKNet, GaAN, MixHOP
4. Scalability: Model scalability, Graph fits in GPU and can be easily parallelised in a spark like framework.
5. Graph Samplers:
    a. Node
    b. Edge
    c. Random Walk
    d. Multi-dimensional Random Walk
    e. Full Graph
6. GraphSAINT demonstrates superior performance in both accuracy and training time on five large graphs, and achieves new state-of-the-art F1 scores for PPI (0.995) and Reddit (0.970).
7. Goals of GraphSAINT
    a. Extract appropriately connected subgraphs so that little information is lost when propagating within the subgraphs.
    b. Combine information of many subgraphs together so that the training process overall learns good representation of the full graph.

Week 2:

Meeting Minutes: *Discussed graphical convolutional networks and how it forms a convolutional network. We were asked to read up on Graphical Neural Networks for text classification and work on the implementation of the same.*

Progress report:

Graph Convolutional Networks for Text Classification:

The paper differs from traditional graphical neural network in the manner it forms the graph. The graph is formed in the following manner:

Nodes: Consists of Documents and unique words.

Number of nodes =Corpus size + Vocabulary size

X(feature matrix) is an n*n identity matrix

The edges are built based on word occurrence in documents (document-word edges) and word co-occurrence in the whole corpus (word-word edges). The weight of document-word edges are TF_IDF values. For weights of word-word edges we use a fixed size sliding window to calculate a Point wise Mutual Information measure.

$$A_{ij} = \begin{cases} \text{PMI}(i,j) & i,j \text{ are words, PMI}(i,j) > 0 \\ \text{TF-IDF}_{ij} & i \text{ is document, } j \text{ is word} \\ 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

The PMI value of a word pair i, j is computed as

$$\text{PMI}(i,j) = \log \frac{p(i,j)}{p(i)p(j)}$$

$$p(i,j) = \frac{\#W(i,j)}{\#W}$$

$$p(i) = \frac{\#W(i)}{\#W}$$

#W (i) is the number of sliding windows in a corpus that contain word i

#W (i, j) is the number of sliding windows that contain both word i and j

#W is the total number of sliding windows in the corpus.

A two-layer GCN can allow message passing among nodes that are at maximum two steps away. Although there are no direct document-document edges in the graph, the two-layer GCN allows the information exchange between pairs of documents.

$E_1 = \tilde{A}XW_0$

$E_2 = \tilde{A}\ \text{ReLU}(\ \tilde{A}XW_0\ )W_1$

Softmax(E2)

Results:

Table 2: Test Accuracy on document classification task. We run all models 10 times and report mean ± standard deviation. Text GCN significantly outperforms baselines on 20NG, R8, R52 and Ohsumed based on student t-test ($p < 0.05$).

Model	20NG	R8	R52	Ohsumed	MR
TF-IDF + LR	0.8319 ± 0.0000	0.9374 ± 0.0000	0.8695 ± 0.0000	0.5466 ± 0.0000	0.7459 ± 0.0000
CNN-rand	0.7693 ± 0.0061	0.9402 ± 0.0057	0.8537 ± 0.0047	0.4387 ± 0.0100	0.7498 ± 0.0070
CNN-non-static	0.8215 ± 0.0052	0.9571 ± 0.0052	0.8759 ± 0.0048	0.5844 ± 0.0106	**0.7775 ± 0.0072**
LSTM	0.6571 ± 0.0152	0.9368 ± 0.0082	0.8554 ± 0.0113	0.4113 ± 0.0117	0.7506 ± 0.0044
LSTM (pretrain)	0.7543 ± 0.0172	0.9609 ± 0.0019	0.9048 ± 0.0086	0.5110 ± 0.0150	0.7733 ± 0.0089
Bi-LSTM	0.7318 ± 0.0185	0.9631 ± 0.0033	0.9054 ± 0.0091	0.4927 ± 0.0107	0.7768 ± 0.0086
PV-DBOW	0.7436 ± 0.0018	0.8587 ± 0.0010	0.7829 ± 0.0011	0.4665 ± 0.0019	0.6109 ± 0.0010
PV-DM	0.5114 ± 0.0022	0.5207 ± 0.0004	0.4492 ± 0.0005	0.2950 ± 0.0007	0.5947 ± 0.0038
PTE	0.7674 ± 0.0029	0.9669 ± 0.0013	0.9071 ± 0.0014	0.5358 ± 0.0029	0.7023 ± 0.0036
fastText	0.7938 ± 0.0030	0.9613 ± 0.0021	0.9281 ± 0.0009	0.5770 ± 0.0049	0.7514 ± 0.0020
fastText (bigrams)	0.7967 ± 0.0029	0.9474 ± 0.0011	0.9099 ± 0.0005	0.5569 ± 0.0039	0.7624 ± 0.0012
SWEM	0.8516 ± 0.0029	0.9532 ± 0.0026	0.9294 ± 0.0024	0.6312 ± 0.0055	0.7665 ± 0.0063
LEAM	0.8191 ± 0.0024	0.9331 ± 0.0024	0.9184 ± 0.0023	0.5858 ± 0.0079	0.7695 ± 0.0045
Graph-CNN-C	0.8142 ± 0.0032	0.9699 ± 0.0012	0.9275 ± 0.0022	0.6386 ± 0.0053	0.7722 ± 0.0027
Graph-CNN-S	–	0.9680 ± 0.0020	0.9274 ± 0.0024	0.6282 ± 0.0037	0.7699 ± 0.0014
Graph-CNN-F	–	0.9689 ± 0.0006	0.9320 ± 0.0004	0.6304 ± 0.0077	0.7674 ± 0.0021
Text GCN	**0.8634 ± 0.0009**	**0.9707 ± 0.0010**	**0.9356 ± 0.0018**	**0.6836 ± 0.0056**	0.7674 ± 0.0020

Advantages of using GCN for text classification:

- Multiple benchmark datasets demonstrate that a vanilla Text GCN without any external word embeddings or knowledge outperforms state-of-the-art methods for text classification.
- Text GCN also learns predictive word and document embeddings.
- The improvement of Text GCN over state-of-the-art comparison methods become more prominent as we lower the percentage of training data, suggesting the robustness of Text GCN to less training data in text classification.

Limitations of GCN for text classification: GCN model is inherently transductive, in which test document nodes (without labels) are included in GCN training. Thus Text GCN can not quickly generate embeddings and make predictions for unseen test documents.

Tensor Graph Convolutional Networks for Text Classification:

Semantic-based, syntactic-based, and sequential-based text graphs are first constructed to build the text graph tensor. To encode the heterogeneous information from multi-graphs, TensorGCN simultaneously performs two kinds of propagation learning. For each layer, an intra-graph propagation is firstly performed for aggregating information from neighbors of each node. Then an inter-graph propagation is used to harmonize heterogeneous information between graphs.

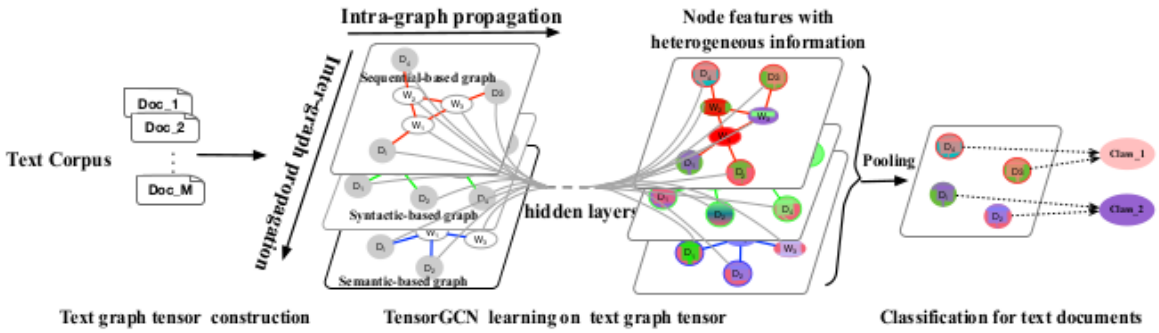


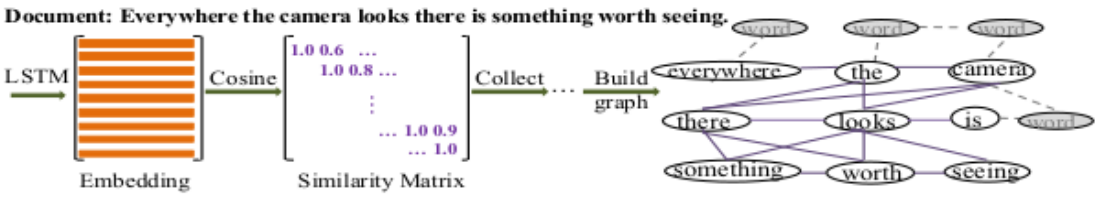
Figure 1: The whole framework of our propsoed TensorGCN for text classification.

Definition 1. G is a **graph tensor**, where G = (G_1 , G_2 , · · · , G_r) and G_i = (V_i , E_i , A_i) , if V_i = V_j (i, j = 1, 2, · · · , r) and A_i != A_j (when i != j).

Definition 2. A = (A_1 , A_2 , · · · , A_r) $\in R^{r \times n \times n}$ is a **graph adjacency tensor** , where A_i (i =1, 2, · · · , r) is the adjacency matrix of the i^{th} graph in the graph tensor G.

Definition 3. **The graph feature tensor**
$H^{(l)} = (H_1^{(l)}, H_2^{(l)},, \cdots, H_r^{(l)}) \in R^{r \times n \times d}$
Where $H^{(i)} \in R^{n \times d}$ (i = 1, 2, · · · , r) is the feature matrix of the ith graph in G. When l = 0, the graph feature tensor $H^{(0)}$ denotes the initialized input features.

- Similar graph construction method as the previous algorithm. Differences exist in the method of construction of word to word edges.In the paper, authors build word-word edges based on three different language properties: semantic information, syntactic dependency, and local sequential context. Based on these different kinds of word-word edges, the authors construct a series of text graphs to depict text documents.
- Semantic-based graph



- Syntactic-based graph: We count the number of times for each pair of words having syntactic dependency (we find syntactic dependency using coreNLP) over the whole

corpus and calculate the edge weight of each pair of words (nodes in the syntactic-based graph) by

$$d_{\text{syntactic}}(w_i, w_j) = \frac{\#N_{\text{syntactic}}(w_i, w_j)}{\#N_{\text{total}}(w_i, w_j)},$$

#N syntactic (w_i , w_j) is the number of times that the two words have syntactic dependency relationship over all sentences/documents in the corpus
#N total is the number of times that the two words exist in the same sentence/document over the whole corpus.

- Sequential based graph: Uses the PMI measure from the previous paper.

Subsequent Intra graph propagation and intergraph propagation

Week 3:
Meeting Minutes: *Discussed the implementation of the paper "graphical convolutional networks for text classification" and explored the methodology of forming a graph from the text data. We were asked to understand the implementation of the same in a line by line manner.*

Progress report:
Understanding the file structure:

- data/corpus/mr.txt: Contains all the documents with each document being one line of the file in its raw format.
- data/corpus/mr.clean.txt: Contains all the documents with each document being one line of the file in its cleaned format(post removal of rare words and stop words).
- data/mr.txt: Contains 'document_id', 'train/test' and '1/0'(binary class).
- data/mr.train.index: Shuffled indices of the train set documents.
- data/mr.test.index: Shuffled indices of the test set documents.
- data/mr_shuffle.txt: Contains the indices of all the documents in shuffled manner (appended lists data/mr.train.index and data/mr.test.index)
- data/corpus/mr_shuffle.txt: Contains the documents in the same shuffled manner according to the indices of the file above.
- data/corpus/mr_vocab.txt: Contains the entire vocabulary.
- data/corpus/mr_vocab_def.txt: Stores the dictionary meaning of words in our vocabulary.
- data/corpus/mr_word_vectors.txt: Stores the embeddings for words of the vocabulary.
- data/corpus/mr_labels.txt: Contains the unique labels of the data.
- data/mr.real_train.name: Contains 90% of the train set (id, train, 1/0)

The implementation of the python scripts are as follows**:**

remove_words.py

1. Read the data/corpus/mr.txt line by line. Each line consists of a document
2. Store the frequency of each unique word in all the documents in a dictionary
3. Remove stop words from each document and remove less frequent words (threshold=5). This is done for all datasets except mr. Store the list of cleaned documents in data/corpus/mr.clean.txt
4. Calculate the metadata about corpus like minimum document size, maximum document size and average document size.

build_graph.py:

1. Set the embedding dimension size as 1000
2. Make four lists:
    a. doc_name_list = [document_id, train/test, 1/0]
    b. doc_train_list = [document_id, train, 1/0]
    c. doc_test_list = [document_id, test, 1/0]
    d. doc_content_list // read cleaned data from the previous step, data/corpus/mr.clean.txt
3. Get all the row numbers/ids of the doc_train_list elements in the doc_name_list into the list train_ids and do a similar process for test_ids. Randomly shuffle both lists and write to data/mr.train.index and data/mr.test.index respectively.
4. Make a new list containing the shuffled train and test ids and write into data/mr_shuffle.txt. The corpus of the same indices is written into data/corpus/mr_shuffle.txt.
5. Then we make a dictionary of {word, frequency} and also build a list of unique words (vocabulary) and write the list into data/corpus/mr_vocab.txt.
6. Make a dictionary word_doc_list that stores { word, list of documents the word appears in}. We also make two more dictionaries of
    a. word_doc_freq = {word, number of documents the word appears in}
    b. word_id_map = {word, index of word in vocabulary}
7. *(Commented Code)* Using wordnet.synset we get the definition of all the words in our vocabulary and store it in data/corpus/mr_vocab_def.txt. We run the tf-idf vectorizer on the word definitions to build a document- word matrix(corpus_size, 1000) and limit the max number of features to 1000 (builds a vocabulary that only considers the top max_features ordered by term frequency across the corpus). We effectively convert each word in the vocabulary to an embedding based on the word meanings as the corpus. We write the word vectors to data/corpus/mr_word_vectors.txt.

Word_Vocabulary	Meaning	Word-1	Word-2	Word-3...	Word-1000
Word 1	Meaning(Document)	tf-idf_word1	tf-idf_word2	tf-idf_word3	tf-idf_word1000

8. We get the unique labels from the data/corpus/mr_shuffle.txt file and write to data/corpus/mr_labels.txt. We separate out 90% of the train set and store it in data/mr.real_train.name.

9. Then we add the embedding value for each word present in the document from the previous step and normalize it using the document length.
   $for\ each\ word\ in\ document,$

$$document_{vector} = (\sum word_{vector})/document_{length}$$

   For each document i we create:

row	column(word_embedding_dimension)	data($document_{vector}$)
Document id, i	1	1st value
Document id, i	...1000	1000the value

We input this data into a compressed sparse row matrix. csr_matrix((data_x, (row_x, col_x)), shape=(real_train_size, word_embeddings_dim)). Note that real_train_size is 90% of the total train set size. We then one hot encode the label and store it in a list. For mr dataset this list is of the form [[1,0], [1,0], [0,1]....[1,0]]. We perform the same steps for the entirety of the test set.

10. We create another sparse matrix of the dimensions (train_size + vocab_size, word_embeddings_dim) and the label matrix of dimension (train_size + vocab_size, label_size). Note that the one hot encoding array consists of all zeros for the vocabulary.
11. The next step is to find the word co-occurrence in the same window. We first create a list of all the windows of size(,window_size). We then create a dictionary {word, number of windows the word appears in}. We then create a dictionary - word_pair_count consisting of {(word1_id,word2_id), the number of windows the words occur together in}.
12. We then calculate the PMI between words used from the data collected from the above step.

```
row = [ ]
col = [ ]
weight = [ ]
for key in word_pair_count:
    temp = key.split(',')
    i = int(temp[0])
    j = int(temp[1])
    count = word_pair_count[key]
    word_freq_i = word_window_freq[vocab[i]]
    word_freq_j = word_window_freq[vocab[j]]
    pmi = log((1.0 * count / num_window) /
               (1.0 * word_freq_i * word_freq_j/(num_window * num_window)))
    if pmi <= 0:
        continue
    row.append(train_size + i)
    col.append(train_size + j)
    weight.append(pmi)
```

13. Then we construct a dictionary with {(document_id,word_id), frequency}. Subsequently, we add document, word weights to the row, col, and weight lists above using the following code:

```
for i in range(len(shuffle_doc_words_list)):
    doc_words = shuffle_doc_words_list[i]
    words = doc_words.split()
    doc_word_set = set()
        for word in words:
        if word in doc_word_set:
            continue
        j = word_id_map[word]
        key = str(i) + ',' + str(j)
        freq = doc_word_freq[key]
        if i < train_size:
            row.append(i)
        else:
            row.append(i + vocab_size)
        col.append(train_size + j)
        idf = log(1.0 * len(shuffle_doc_words_list) /
                  word_doc_freq[vocab[j]])
        weight.append(freq * idf)
        doc_word_set.add(word)
```

14. With this we finally obtain the adjacency matrix:

```
node_size = train_size + vocab_size + test_size
adj = sp.csr_matrix(
    (weight, (row, col)), shape=(node_size, node_size))
```

15. We dump the four sparse matrices and matrices of respective labels formed till now using pickle:

```
f = open("data/ind.{}.x".format(dataset), 'wb')
pkl.dump(x, f)
f.close()
f = open("data/ind.{}.y".format(dataset), 'wb')
pkl.dump(y, f)
f.close()
f = open("data/ind.{}.tx".format(dataset), 'wb')
pkl.dump(tx, f)
f.close()
f = open("data/ind.{}.ty".format(dataset), 'wb')
pkl.dump(ty, f)
f.close()
f = open("data/ind.{}.allx".format(dataset), 'wb')
pkl.dump(allx, f)
f.close()
f = open("data/ind.{}.ally".format(dataset), 'wb')
pkl.dump(ally, f)
f.close()
f = open("data/ind.{}.adj".format(dataset), 'wb')
```

```
pkl.dump(adj, f)
f.close()
```

train.py:

1. Load the corpus from the file. We load/generate the following data structures:
    a. features: (train_size + vocab_size + test_size, word_embeddings_dim) Matrix (N,1000)
    b. y_train, y_val, y_test: Individual label arrays.
    c. train_mask, test_mask, val_mask: these are masking arrays of features and labels.
    d. train_size, test_size
    e. Adjacency matrix(adj): We make adj symmetric in the following manner:

```
adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)
```

```
adj=
   1     2      0
 1/5     1    1/2
3/10   1/5      0


adj.T=
1      1/5    3/10
2        1     1/5
0      1/2       0

adj.T>adj
false false true
true  false false
false true  false

adj.T.multiply(adj.T>adj)
0        0    3/10
2        0       0
0      1/2       0

adj.multiply(adj.T>adj)
   0     0      0
 1/5     0      0
   0   1/5      0

adj + adj.T.multiply(adj.T>adj)- adj.multiply(adj.T>adj)

   1     2    3/10
   2     1     1/2
3/10   1/2      0
```

The need to make the matrix symmetric in such fashion comes from the fact that we added the tf-idf value to (document, word) entry in the matrix but did not add it to (word, document).

2. We edit the feature matrix (features) to make it into an identity matrix of (train_size + vocab_size + test_size, train_size + vocab_size + test_size) size.
3. We then normalize the adjacency matrix in the following manner:

```
def normalize_adj(adj):
    """Symmetrically normalize adjacency matrix."""
    adj = sp.coo_matrix(adj)
    rowsum = np.array(adj.sum(1))
    d_inv_sqrt = np.power(rowsum, -0.5).flatten()
    d_inv_sqrt[np.isinf(d_inv_sqrt)] = 0.
    d_mat_inv_sqrt = sp.diags(d_inv_sqrt)
    return adj.dot(d_mat_inv_sqrt).transpose().dot(d_mat_inv_sqrt).tocoo()
```

4. We build a GCN model with the following parameters:
    a. num_support (1): Number of supporting matrices (adj in this case)
    b. number of features
    c. Labels
    d. Dropout (default to 0)

5. The built model has the following properties:
    a. Adamoptimizer
    b. Learning rate=0.02
    c. Embedding dimension of hidden layer = 200
    d. Masked_softmax_cross_entropy error
    e. Relu activation function

Week 4:

Meeting Minutes: *We discussed the implementation of the code in a line by line manner. We were asked to fork the repository and document the code by appropriate comments.*

Progress report: Documented the code in an appropriate manner and made minor changes to the implementation. The code can be found here: https://github.com/Raj-Sanjay-Shah/text_gcn

Week 5:

Meeting Minutes: *Discussed the iron march dataset and the objectives and the graph formation methods for the same. We were asked to prepare a short presentation on the various methods for unsupervised graph embedding generation. We were asked to code the build graph part for the iron march dataset*

Progress report: We coded the build graph part and also generated graph embedding using traditional autoencoders.

```
1. #VGAE CODE
```

```
def _build(self):
        self.hidden1 = GraphConvolutionSparse(input_dim=self.input_dim,
                                              output_dim=FLAGS.hidden1,
                                              adj=self.adj,
                                              features_nonzero=self.features_nonzero,
                                              act=tf.nn.relu,
                                              dropout=self.dropout,
                                              logging=self.logging)(self.inputs)

        self.z_mean = GraphConvolution(input_dim=FLAGS.hidden1,
                                       output_dim=FLAGS.hidden2,
                                       adj=self.adj,
                                       act=lambda x: x,
                                       dropout=self.dropout,
                                       logging=self.logging)(self.hidden1)

        self.z_log_std = GraphConvolution(input_dim=FLAGS.hidden1,
                                          output_dim=FLAGS.hidden2,
                                          adj=self.adj,
                                          act=lambda x: x,
                                          dropout=self.dropout,
                                          logging=self.logging)(self.hidden1)

        self.z = self.z_mean + tf.random_normal([self.n_samples, FLAGS.hidden2]) * tf.exp(self.z_log_std)

        self.reconstructions = InnerProductDecoder(input_dim=FLAGS.hidden2,
                                        act=lambda x: x,
                                        logging=self.logging)(self.z)
class InnerProductDecoder(Layer):
    """Decoder model layer for link prediction."""
    def __init__(self, input_dim, dropout=0., act=tf.nn.sigmoid, **kwargs):
        super(InnerProductDecoder, self).__init__(**kwargs)
        self.dropout = dropout
        self.act = act

    def _call(self, inputs):
        inputs = tf.nn.dropout(inputs, 1-self.dropout)
        x = tf.transpose(inputs)
        x = tf.matmul(inputs, x)
        x = tf.reshape(x, [-1])
        outputs = self.act(x)
        return outputs
```

Week 6:

Progress report: Implemented Variational Graph Autoencoders and SDNE. Also implemented the networkx graph for possible scalability to graph-sage and graph saint libraries.

Week 7:

Progress report: There were some errors in the implementation of the variational graph autoencoders.

Results:

- The experimental results show that VGAE performs well when we consider the size of hidden layer 1 to be 32 and hidden layer 2 to be 16.
- The ROC curve is attached below, the area under the roc curve is 0.86. The average precision of the model is around 87%.
- One trend we noticed in the SDNE is that the L2 loss is considerably greater than the L1 loss and plays a bigger role.

Week 8:

Progress report: Used self organizing maps for cluster visualization. We worked on the diachronic shifts of the embeddings across timestamps in the manner proposed in section 3.3 of the DGEF paper.

Week 9: Reviewed Papers from BDACCIP

Week 10:

Progress report: Tried determining the clustering quality by using a variety of clustering techniques like DBSCAN, Kmeans, spectral clustering.