Graph Based Neural Architectures

Tanishq Chaudhary
Mayank Goel
Prajneya Kumar
Shivansh Subramaniam

Introduction

Graphs and Neural Networks

- A lot of things can be represented as graphs, and in doing so we are able to plot direct relation between two entities and base our calculations on it.
- We can think of traditional texts, or images as graphs, but there are more complicated graph structures which find some difficulty being represented by traditional neural network
- Say, CNNs, are difficult to use on complex graphs structure due to arbitrary size and no spatial locality

Graphs and Neural Networks

- We explore Graph Convolutional Network, a basic Graph based Neural Architecture used to modify the node features as we go through multiple convolutional layers
- Then, we explore Attention roughly based on GCNs (called Graph Attention Network, or GAT). This is able to focus on particular edges more, and hence is able to give better results.

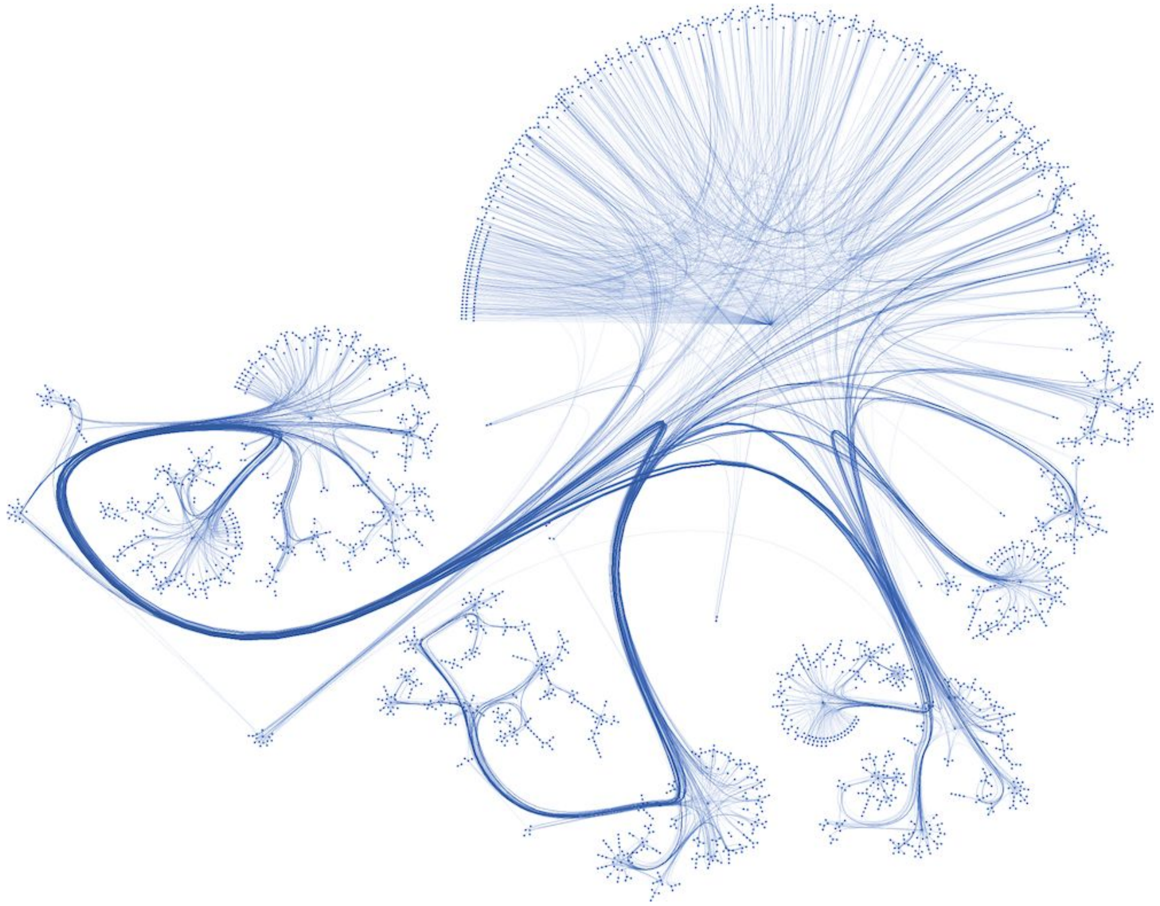
Dataset

Dataset and Learning

- Depending on the dataset chosen, we may or may not be able to use the traditional training methods
- Assuming we have samples of graphs, then we can do a normal 80-20 split where we hold the testing data away from the model, and perform “inductive learning”
- Say we have a huge graph, here, we cannot just perform a random split since that would make the graph incomplete. Hence, we show all the nodes to model while training, but back-propagate only on a few of the nodes. This is how we perform “transductive learning”.

Cora Dataset Stats

- Cora dataset represents multiple research publications as a network of citations, where 2 publications have an edge if one of them have cited the other (for simplicity, edges are assumed to be non-directional)
- Each node has a feature vector assigned to it, based on one-hot encoding of words existing in it or not.
- Details about the Dataset:
    - **2708** Nodes (each node is a research paper)
    - **1433** Unique words per document
    - **5278** Edges (citation connections)
    - **7** Classes: Neural Networks, Probabilistic Methods, Genetic Algorithms, Theory, Case Based, Reinforcement Learning and Rule Learning.
    - Class distribution: {
      2: 818, 3: 426, 1: 418, 6: 351, 0: 298, 4: 217, 5: 180
      }

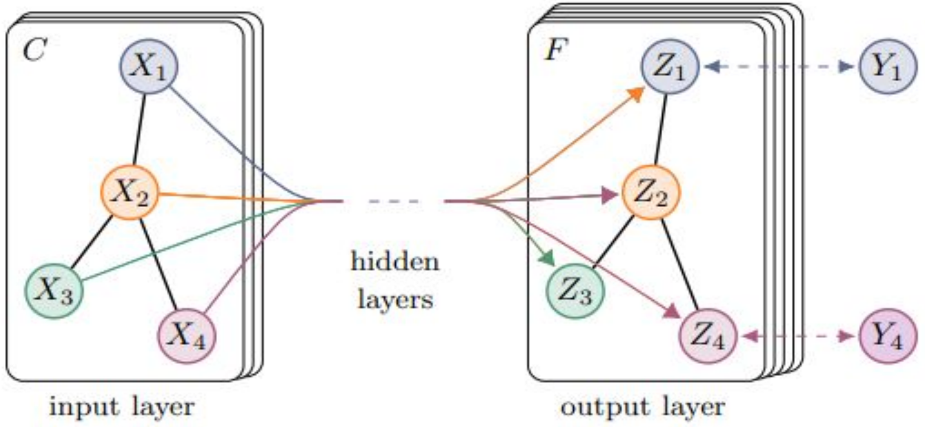


Understanding Graphs

graph $G(V, E)$, edges $\in E$ & vertices/nodes $\in V$
(undirected)

adj. matrix $A \in \mathbb{R}^{N \times N}$

degree matrix $D_{ii} = \text{sum}_j A_{ij}$



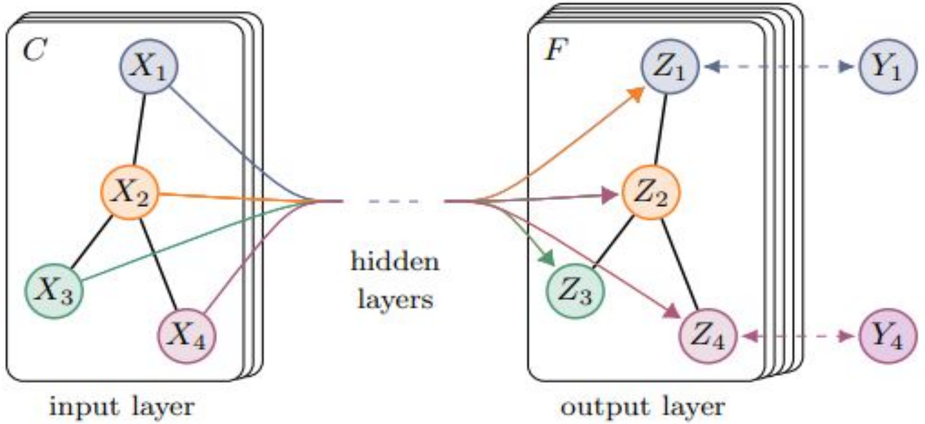
$\underline{G(V,E)}$

$X_{N\times C}$: C channels for each N nodes

$A_{N\times N}$: Adj. matrix (can be sparse)

$\xrightarrow{f}$

$Z_{N\times F}$
F output features



non linear

$$H^{l+1} = f(H^l, A)$$

$$H^0 = X, \quad H^L = Z$$

Issues

- **A** will not consider the nodes features itself, unless there are self loops
- **A** can scale the feature vectors arbitrarily.

Fixing the Issues

- Adding self loops: **A → A + I**
- Normalizing matrix: **A → D^{-0.5}AD^{-0.5}**

$$H^{l+1} = \overset{\substack{\text{non} \\ \text{linear}}}{f}(H^l, A)$$

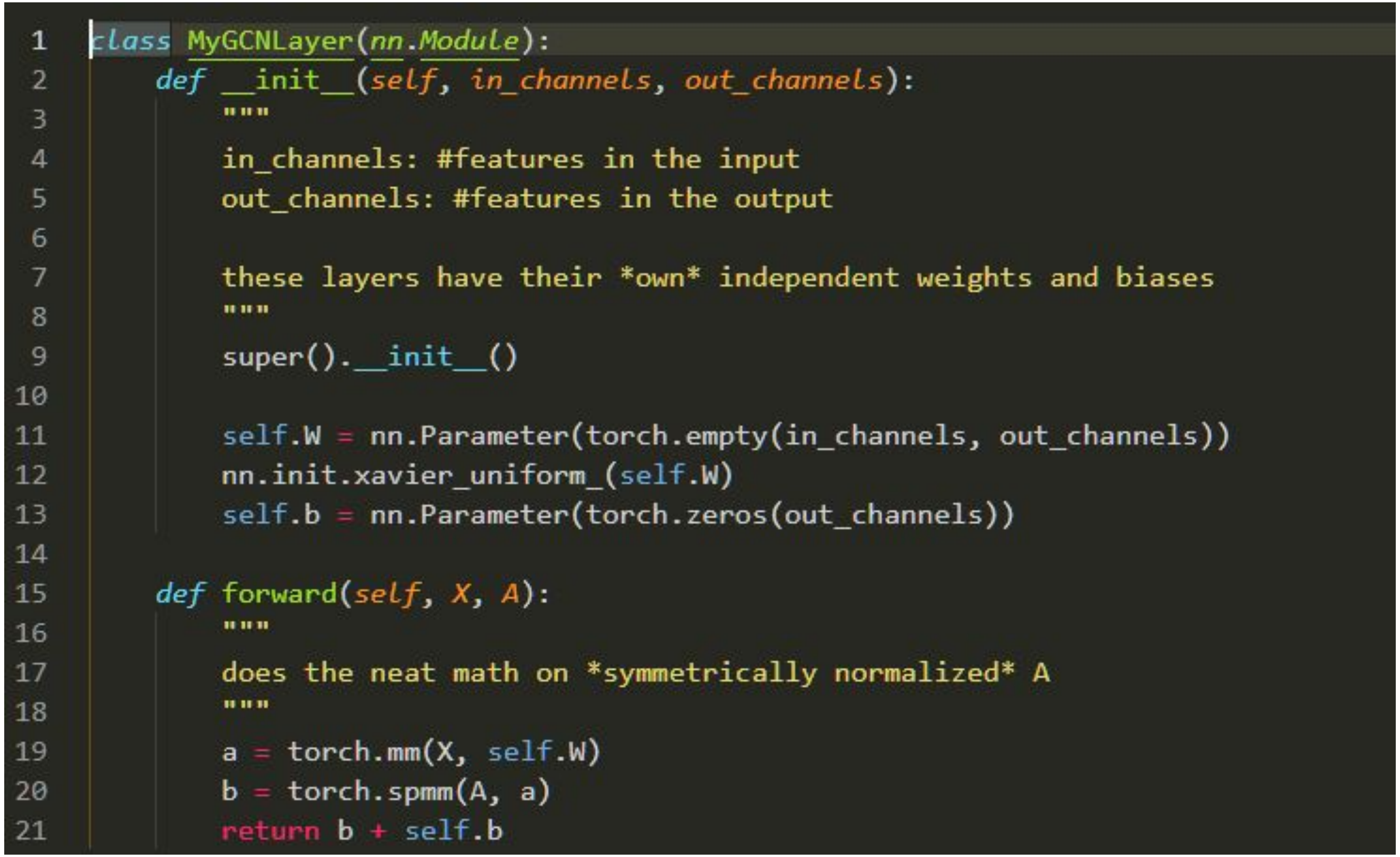
$$H^0 = X \ , \ H^L = Z$$

$$H^{l+1} = f(H^l, A) = \sigma\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} H^l W\right)$$

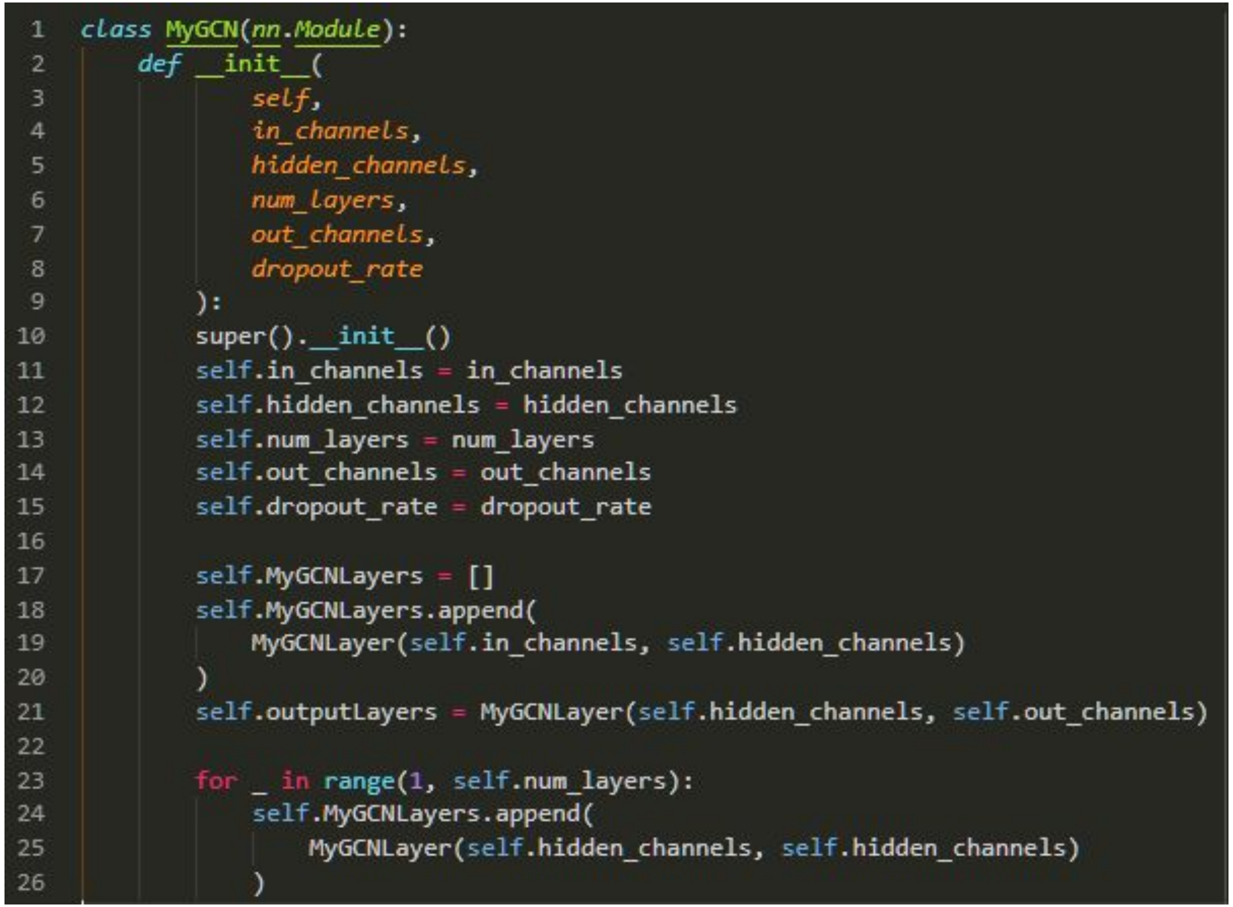
$\hat{A} = A + I$ (for self loops)

$\hat{D}$ = diagonal node degree of mat. $\hat{A}$

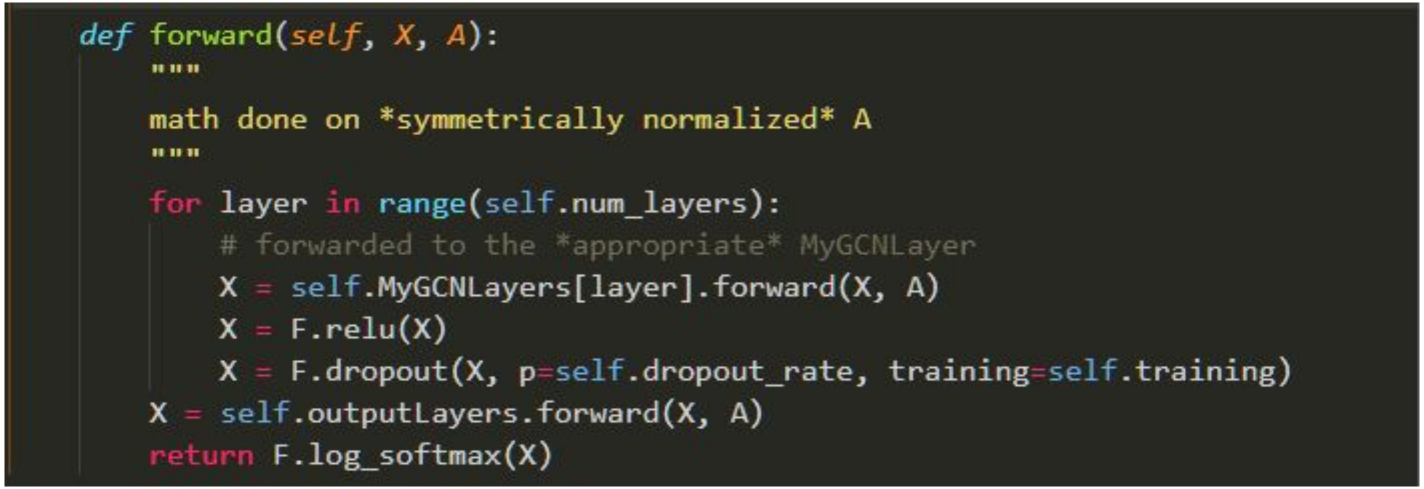
GCN: Implementation



```python
class MyGCNLayer(nn.Module):
    def __init__(self, in_channels, out_channels):
        """
        in_channels: #features in the input
        out_channels: #features in the output

        these layers have their *own* independent weights and biases
        """
        super().__init__()

        self.W = nn.Parameter(torch.empty(in_channels, out_channels))
        nn.init.xavier_uniform_(self.W)
        self.b = nn.Parameter(torch.zeros(out_channels))

    def forward(self, X, A):
        """
        does the neat math on *symmetrically normalized* A
        """
        a = torch.mm(X, self.W)
        b = torch.spmm(A, a)
        return b + self.b
```



```python
class MyGCN(nn.Module):
    def __init__(
            self,
            in_channels,
            hidden_channels,
            num_layers,
            out_channels,
            dropout_rate
        ):
        super().__init__()
        self.in_channels = in_channels
        self.hidden_channels = hidden_channels
        self.num_layers = num_layers
        self.out_channels = out_channels
        self.dropout_rate = dropout_rate

        self.MyGCNLayers = []
        self.MyGCNLayers.append(
            MyGCNLayer(self.in_channels, self.hidden_channels)
        )
        self.outputLayers = MyGCNLayer(self.hidden_channels, self.out_channels)

        for _ in range(1, self.num_layers):
            self.MyGCNLayers.append(
                MyGCNLayer(self.hidden_channels, self.hidden_channels)
            )
```



```python
def forward(self, X, A):
    """
    math done on *symmetrically normalized* A
    """
    for layer in range(self.num_layers):
        # forwarded to the *appropriate* MyGCNLayer
        X = self.MyGCNLayers[layer].forward(X, A)
        X = F.relu(X)
        X = F.dropout(X, p=self.dropout_rate, training=self.training)
    X = self.outputLayers.forward(X, A)
    return F.log_softmax(X)
```

Graph Attention Networks

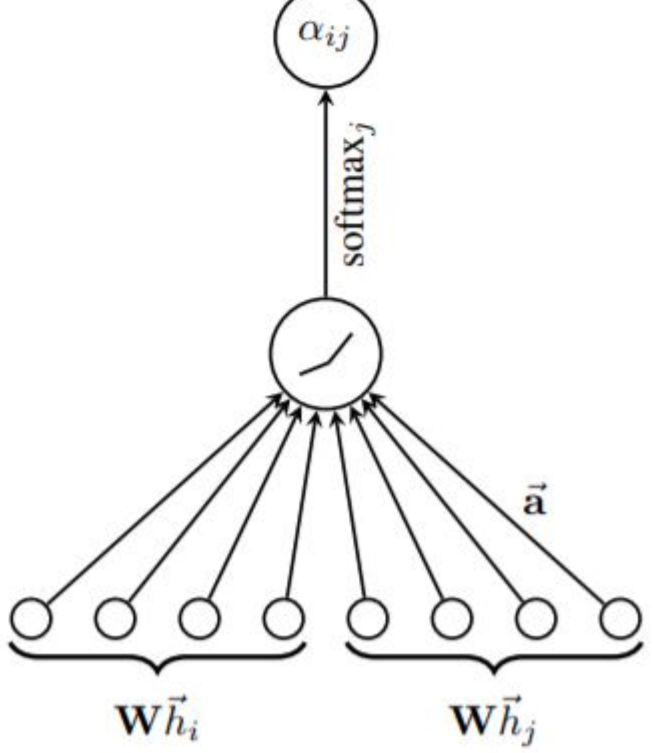
Attention

- Attention is a function represented by **a**
- It is a single layer feed-forward neural network
- Takes 2 node features as input, gives attention of the edge as output.

$$e_{ij} = a(\vec{Wh_1}, \vec{Wh_2})$$

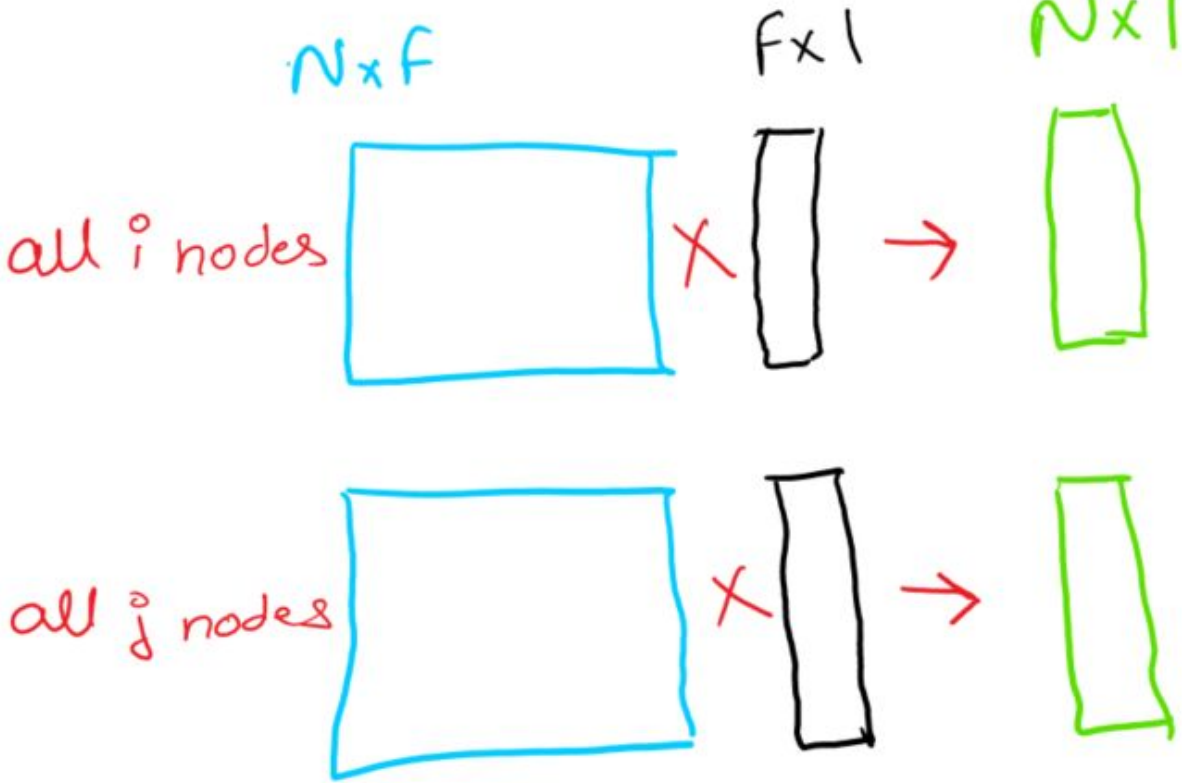
attention given to edge connecting node i to node j

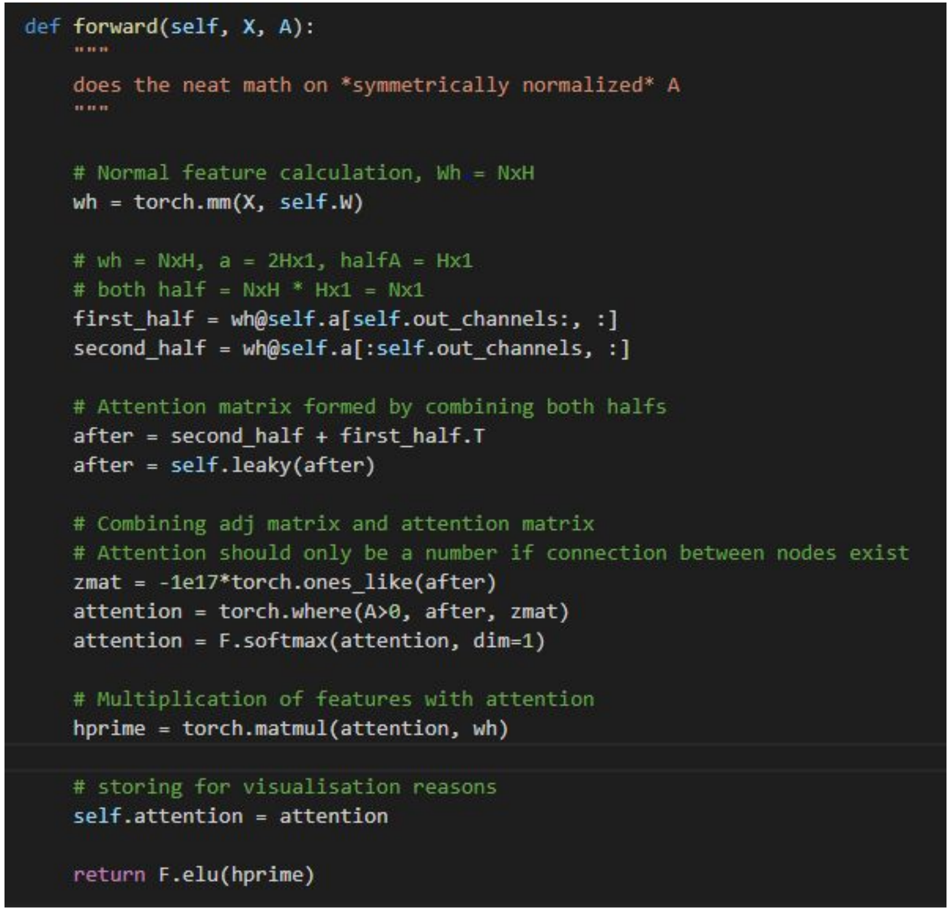
node feature vector



One could also assume a to be just a learnable weight matrix which needs to be multiplied.

Once we get the softmax, we apply LeakyReLU (default parameters) as activation function


N x F
F x 1
N x 1
all i nodes
X
all j nodes
X



```python
def forward(self, X, A):
    """
    does the neat math on *symmetrically normalized* A
    """

    # Normal feature calculation, Wh = NxH
    wh = torch.mm(X, self.W)

    # wh = NxH, a = 2Hx1, halfA = Hx1
    # both half = NxH * Hx1 = Nx1
    first_half = wh@self.a[self.out_channels:, :]
    second_half = wh@self.a[:self.out_channels, :]

    # Attention matrix formed by combining both halfs
    after = second_half + first_half.T
    after = self.leaky(after)

    # Combining adj matrix and attention matrix
    # Attention should only be a number if connection between nodes exist
    zmat = -1e17*torch.ones_like(after)
    attention = torch.where(A>0, after, zmat)
    attention = F.softmax(attention, dim=1)

    # Multiplication of features with attention
    hprime = torch.matmul(attention, wh)

    # storing for visualisation reasons
    self.attention = attention

    return F.elu(hprime)
```

As can be seen in the code:

X: Input features
A: Adjacency matrix

self.a: Attention function, 2*Fx1

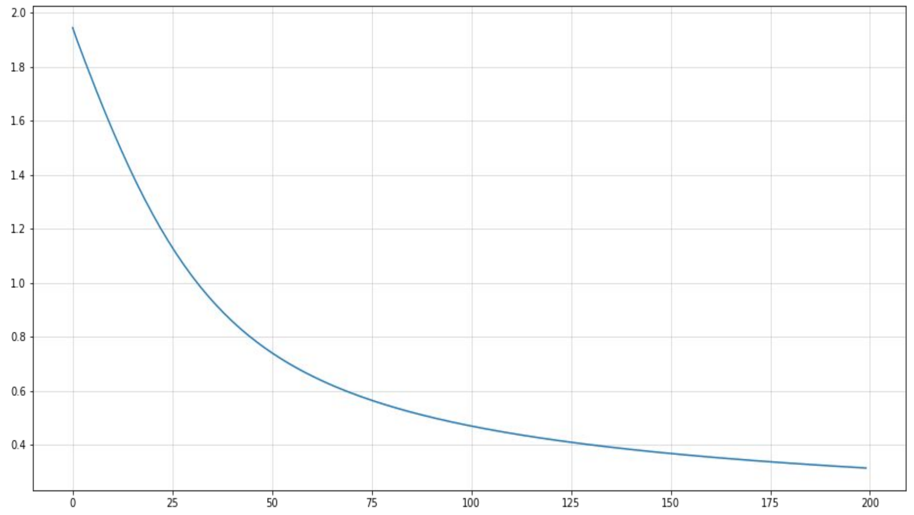
We combine output of nodes i and j by adding the output from matrix multiplication together

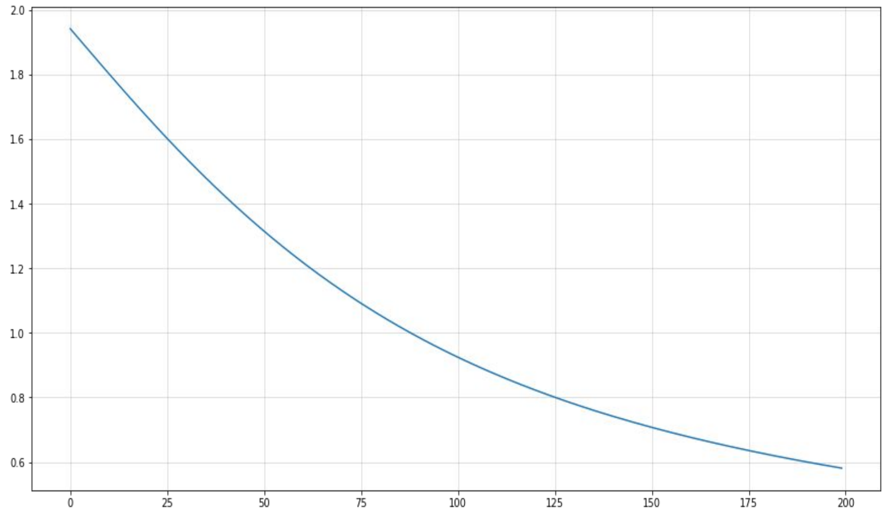
Results

Hyperparameters

- We tried 2 things with both the datasets:
- Hyperparameters as mentioned in the paper:
    - Training Nodes: 140 (20 from each label)
    - Test Nodes: 1000 (randomly selected)
    - Epochs: 200
    - Layers: 2
    - Hidden Features: 32
- Hyperparameters which gave best results:
    - Training Nodes: 1647 (atmax 250 from each label)
    - Test Nodes: 1061 (remaining)
    - Hidden Features: 1-24

Loss


2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
25
50
75
100
125
150
175
200


2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0
25
50
75
100
125
150
175
200

GCN

GAT

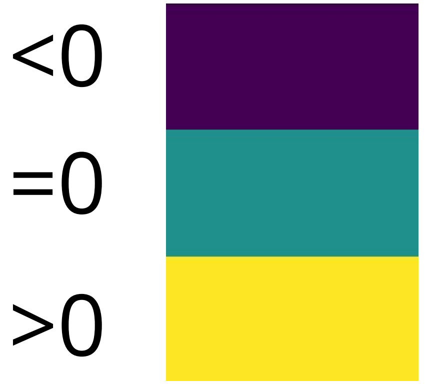
Loss: Interpretations

- As we can see, GCN’s loss decreases faster and is lower compared to GAT’s
- Possible Explanations:
- We observed with our experiments that as the number of layers of GAT increases the loss decreases as well (we could do an experiment on how the loss changes as we increase layers and compare between GCN and GAT)
- Another explanation is that since initially a matrix is randomized, probably GAT would perform better as we increase the number of epochs
- We observe that test accuracy for GCN: 76.5%, and GAT: 82.38%, hence it could also be that GCN overfits hence gives lesser loss.

Understanding Attention

Attention

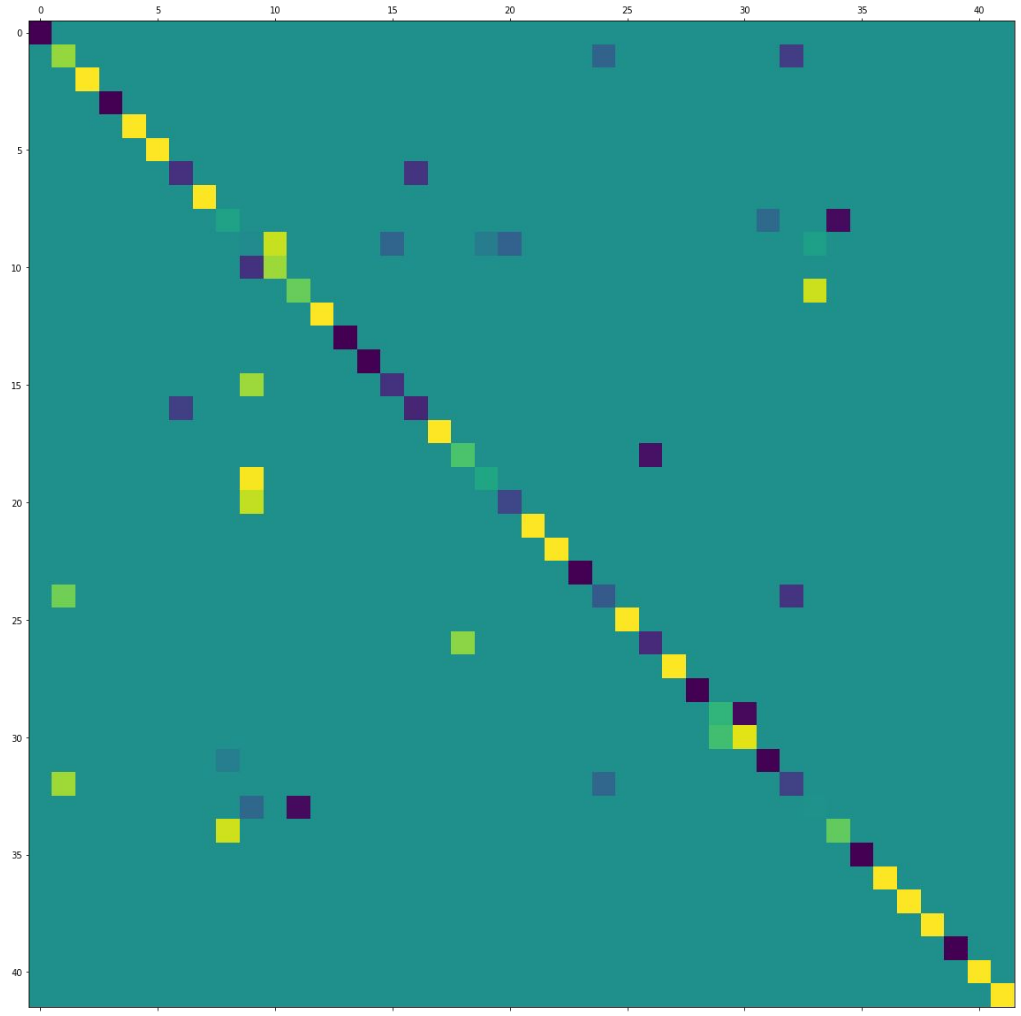
- We compare the **difference** between the **first layer** of the model and the **last layer** (first & second layer in our case)
- **2708x2708** (node by node) attention matrix is not easily understood, so we take the first **42** nodes for ***each class*** → **42x42** matrix
- We expect to see a high weightage on the node itself and some



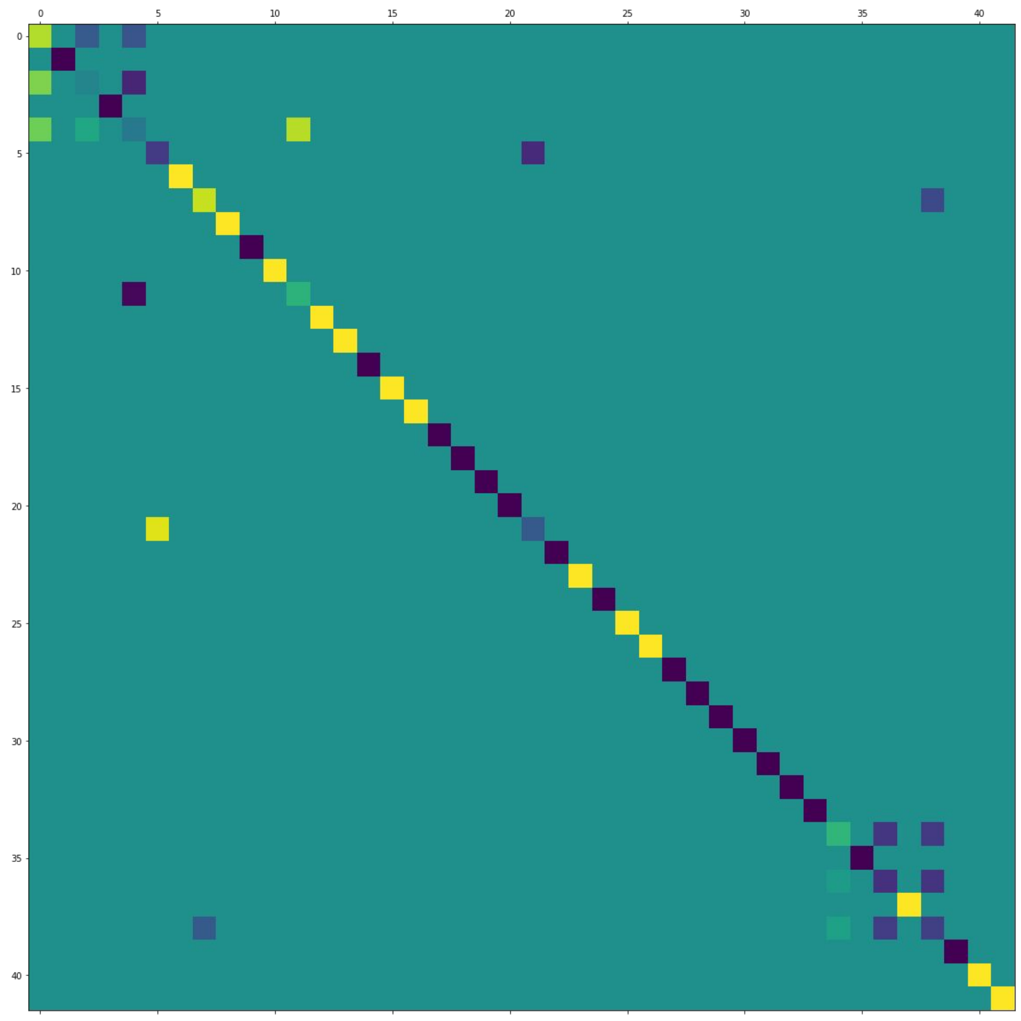
Attention Insights

- Initially, attention matrix is randomized, hence we see that GAT's losses start higher as compared to GCNs
- Highest change (whether negative or positive) occur in self edges, which means that the nodes most effecting a node's class is most probably itself
- As seen in the images of attention plotted for same labels, all attention matrices are symmetric in nature
- A possible explanation for change in colour might be that node_i is more affected by node_j as compared to node_j by node_i.

Class 0



Class 2



Future Work

Future Work

- Conceptualise how GATs can be used for NLP applications such as Text Classification, Relation Extraction etc.

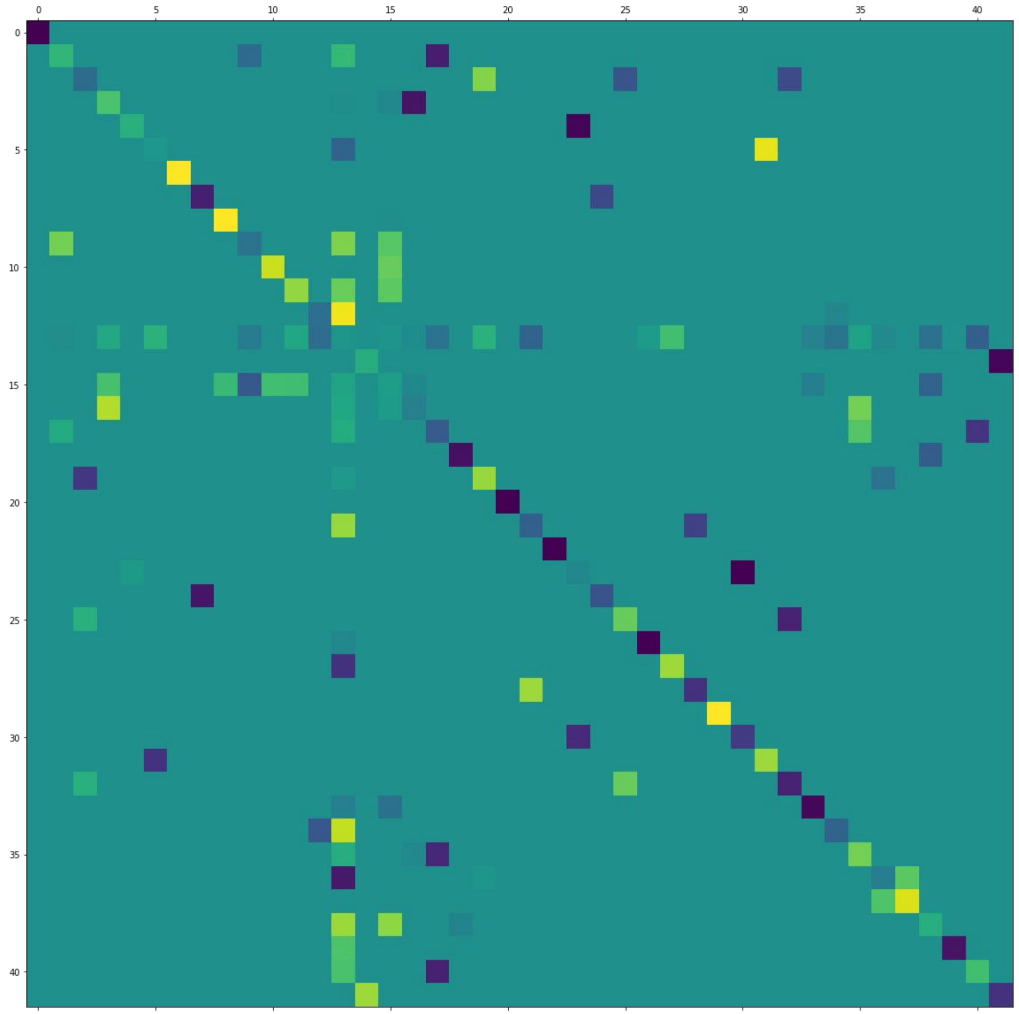
- Perform inference on textual datasets (Wikipedia, news articles, etc.) which can be described by graphs to compare the aforementioned NLP Application

- Compare the performance of GATs with other GNNs on a given task

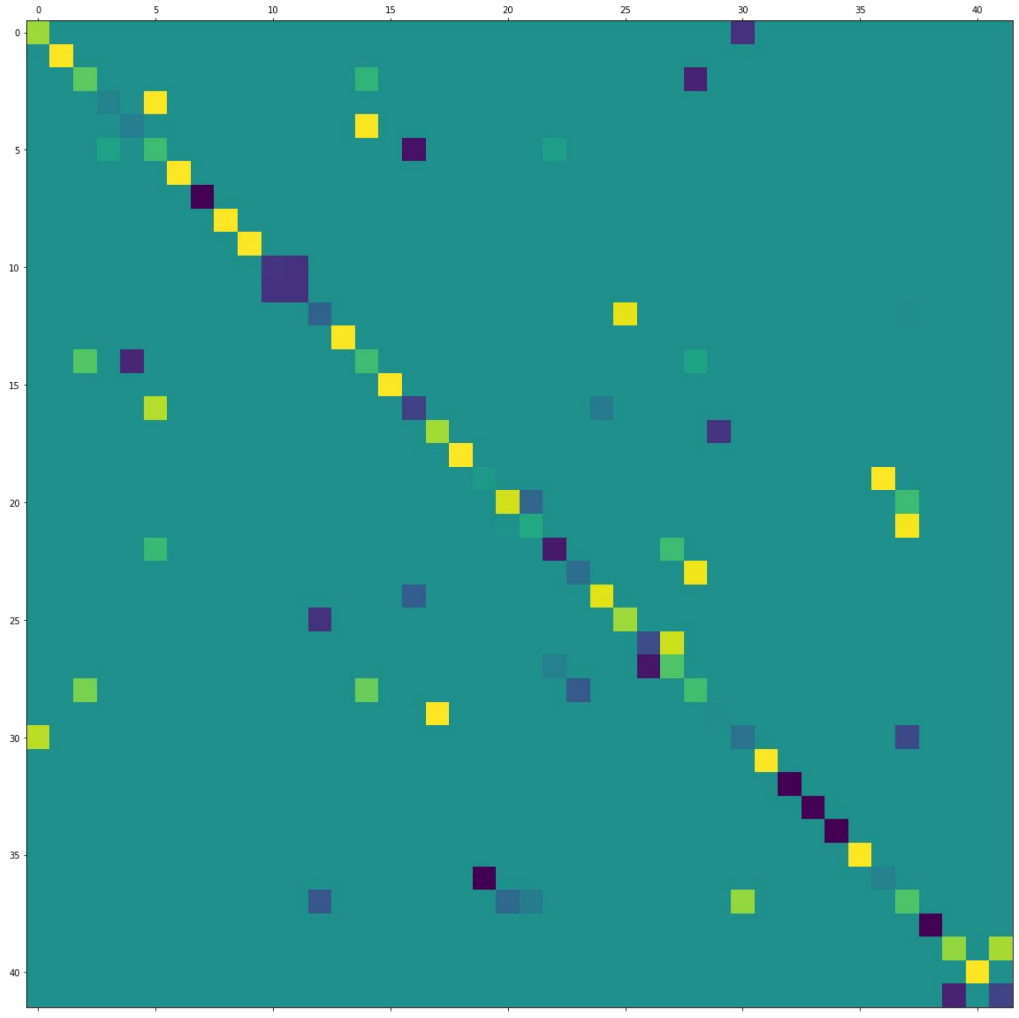
Discuss

BACKUP SECTION

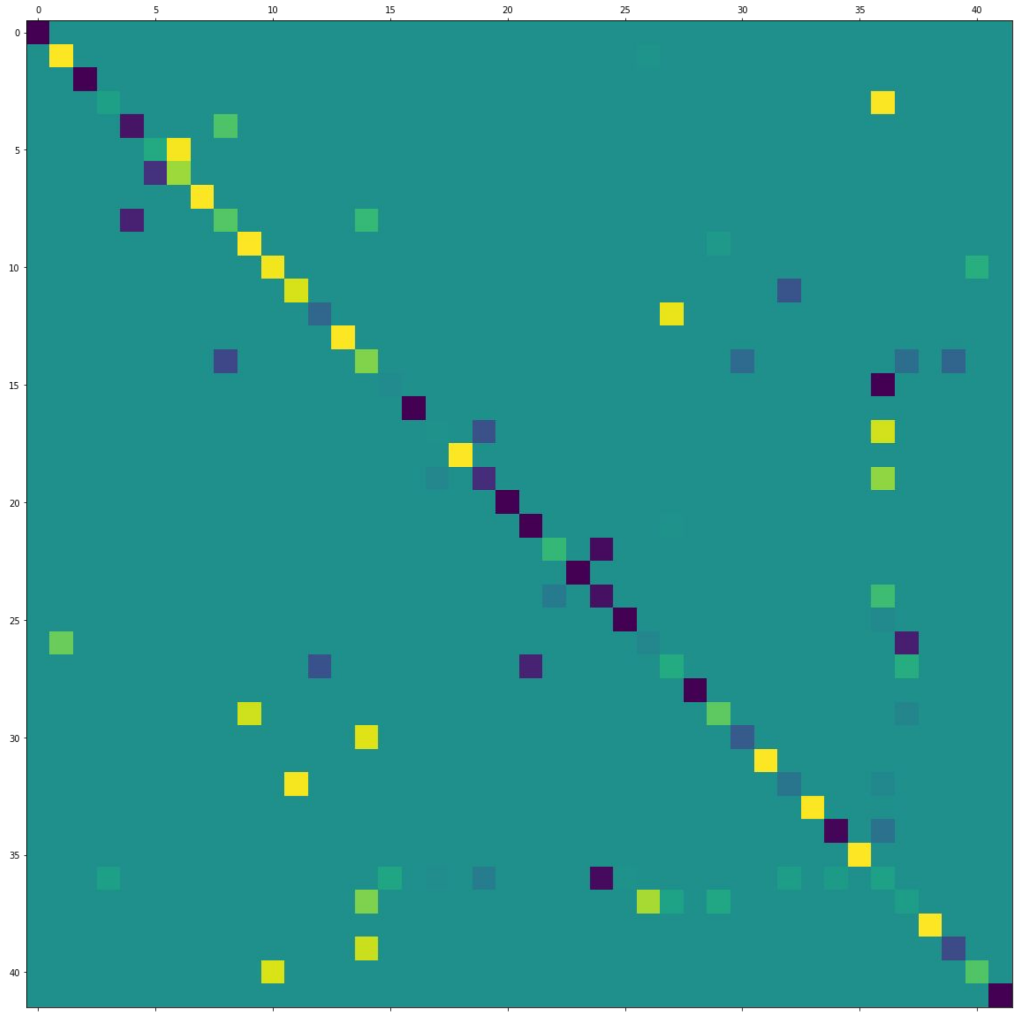
Class 1



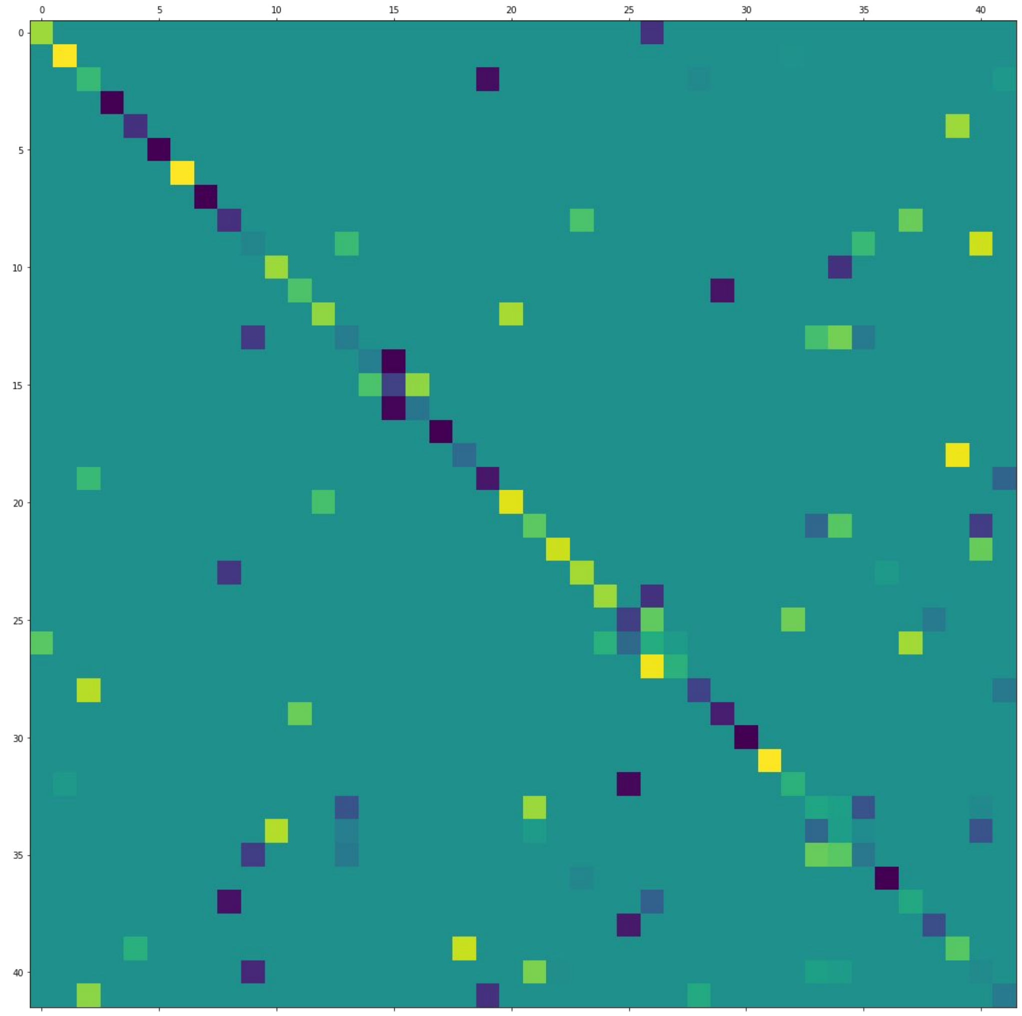
Class 3



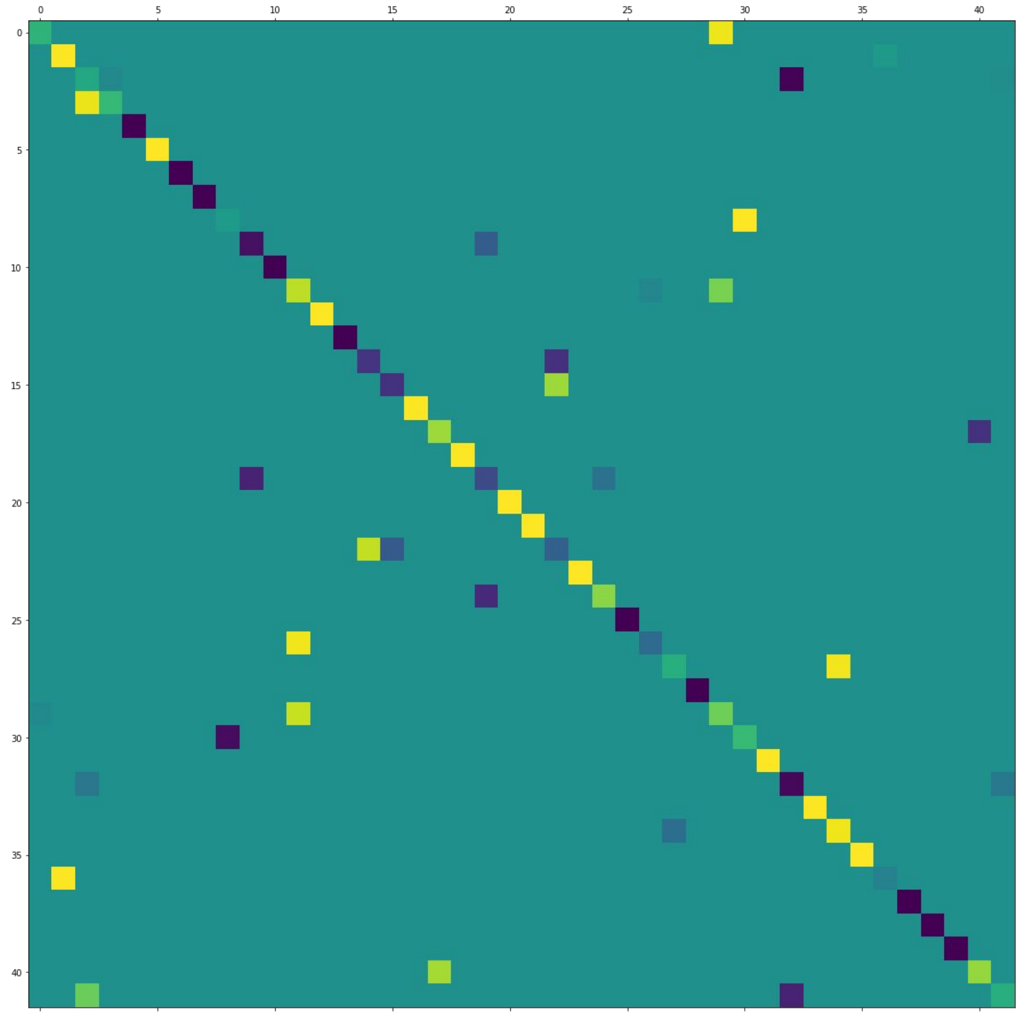
Class 4



Class 5



Class 6


0
5
10
15
20
25
30
35
40