Introduction

0.1 Graph maths

Graph ..... $G = (N, E)$

0.2 Graph implementations

0.3 Graph Algorithms

BFS ..... Breadth-first search
DFS ..... Depth-first search
Random Walk ..... randomize decision to follow edges
Biased 2nd order Random Walk ..... randomize decision to follow edges

0.4 Graph ML concepts

note2vec .. Use flexible, biased random walks that can trade off between local and global views of the network[1]
deepwalk ....

0.5 Maths stuff

Real numbers ..... R
Integers ..... Z

0.6 Machine Learning concepts

Stochastic Gradient Descent .. evaluate gradients for each individual training example

0.7 Machine Learning functions

SoftMax ..... $\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$
Sigmoid ..... $\sigma(z) = \frac{1}{1+e^{-z}}$
Relu ..... $Relu(z) = max(0, z)$
Mean Absolute Error (MAE) ..... $\sum_{i=1}^{D} |x_i - y_i|$
Mean Squared Error (MSE) ..... $\sum_{i=1}^{D} (x_i - y_i)^2$
Cross Entropy ..... $-(y \log(p) + (1-y) \log(1-p))$ for $M = 2$
$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$ for $M > 2$

0.8 Statistics

True Positive ..... TP
False Positive ..... FP
True Negative ..... TN
False Negative ..... FN
Precision ..... $\frac{TP}{TP+FP}$
Recall ..... $\frac{TP}{TP+FN}$
Accuracy ..... $\frac{TP+TN}{TP+TN+FP+FN}$
Sensitivity = Recall ..... $\frac{TP}{TP+FN}$
Specificity ..... $\frac{TN}{FP+TN}$
Cosine similarity ..... $Cosine(x, y) = \frac{x \cdot y}{|x||y|}$
Jaccard similarity ..... $Jaccard(U, V) = \frac{|U \cap V|}{|U \cup V|}$
Pointwise Mutual Information (PMI) similarity ..... $PMI(x; y) = \log \frac{p(x,y)}{p(x)p(y)}$

1 Over-smoothing

This problem happens when we stack too many layers in a GNN. The problem is that the embeddings tend to converge to a similar value. But we want node embeddings to be different.

The problem is that as we go out from the node of interest, the number of shared nodes also goes up. The embedding of a node is determined by the *receptive field*. A receptive field is a set of nodes that are connected to the node of interest. If two nodes have highly overlapping receptive fields, then the embeddings will be likewise very similar. This results in the over-smoothing problem.

1.0.1 Layers

We need to be careful we don't have too many layers. A too deep network will result in oversmoothing. Setting the number of layers to be only slightly more than the size of the receptive field is a good first approximation.

1.0.2 Increase expressive power

One approach is to increase expressive power with each layer.

In both the aggregation and transformation layers, we could include a 3-layer MLP.

1.0.3 Adding non-message passing layers

We can add pre- and postprocessing layers. See **??**.

These MLP layers refine the features of the nodes before and after the GNN layers. The preprocessing layers could process note features, for instance, if

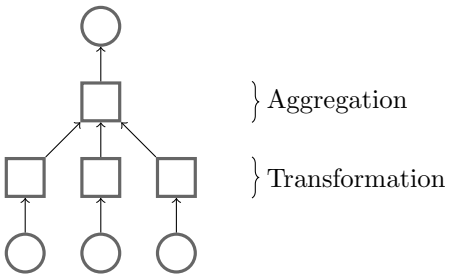


Figure 1: A simple GNN

the nodes represent images or text. The postprocessing layers could process the embedding, for instance, if we want to use the embedding as a feature for classification.

1.0.4 Skip Connections

Skip the layers in the neural Network. See 3

2 A general GNN framework

Comprised of layers of transformers and aggregators where messages are being passed between the layers. The transformers are used to process the input data and the aggregators are used to combine the output of the transformers.

3 Graph augmentation

The raw input graph may not be the same as the computational graph. We use *graph feature* and *graph structure* augmentation.

3.1 Graph feature augmentation

Sometimes, nodes don't have features. We could just assign a constant value to each node. This isn't as dumb as it sounds as it still allows the node to be described by it's connections to other nodes.

We could also use a *one-hot* encoding for each node. This is a simple way to encode the node's feature. For a six node graph, we could use a vector of length six with a one at the index of the node, like $id_5 = [0, 0, 0, 0, 1, 0]$. However, it might be hard to generalize this encoding if the node numbers are arbitrary.

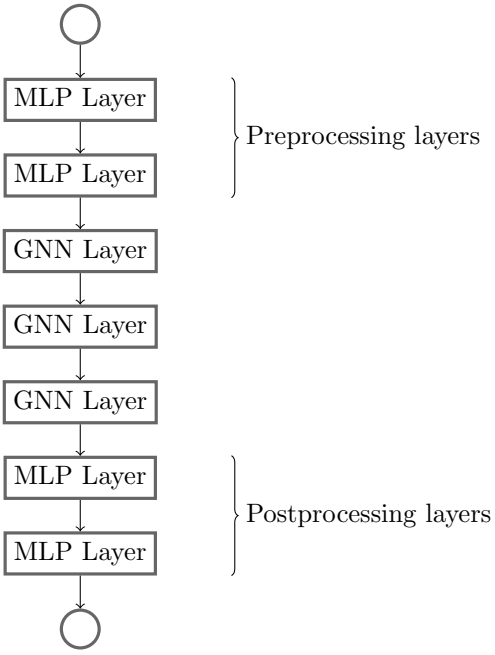


Figure 2: A simple GNN

Comparing constant to one-hot encoding

	Constant node feature	One-hot node feature
Expressive Power	Medium. All the nodes are identical, but GNN can still learn from the graph structure	High. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen models)	High. Simple to generalize to nodes: we assign constant feature to them, then apply our GNN	Low. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs.
Computational cost	Low. Ony one dimensional feature	high. $O(\|V\|)$ dimensional feature cannot apply to large graphs.
Use cases	Any graph, unductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

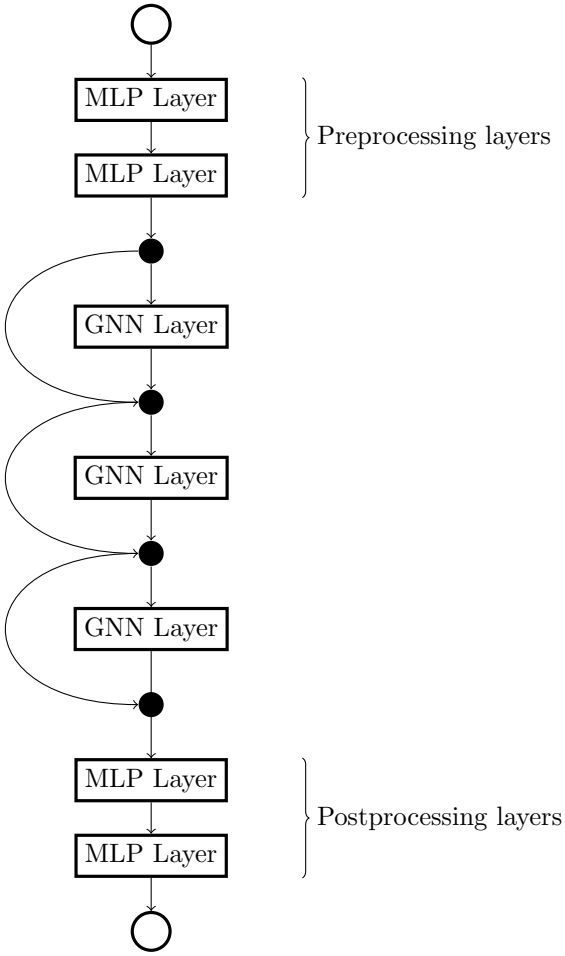
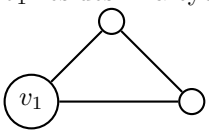


Figure 3: A simple GNN with skipping

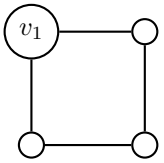
3.2 Hard to learn structures

Sometimes, the structures are hard for the GNN to learn. For examples, cycles are often a problem.

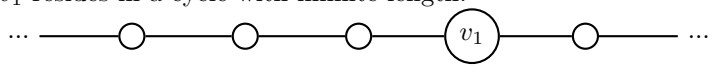
v_1 resides in a cycle with length 3:



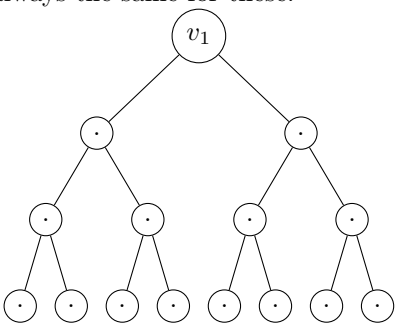
v_1 resides in a cycle with length 4:



v_1 resides in a cycle with infinite length:



Those are virtually the same as this for a GNN. The computational graph is always the same for these:



A possible solution would be to create features that reflect the structure.

- Features could include
- node dgree
- clustering coefficient
- pagerank
- centrality
- ... (anything we know about from classical graph theory)

3.3 Graph structure augmentation

Motivation augment sparse graphs.

3.3.1 Add virtual edges

. For instance, connect 2-hop neighbors via virtual edges. Intuition entrad of using adjacency matrix A for GNN computation, use $A + A^2$.

Use cases: bibartite graphs like author-to-papers or 2-hop virtual edges to make author-author collaboration graph.

3.3.2 Add Virtual nodes

Intuition: connect nodes that are far apart and make the message passing more efficient.

3.3.3 What if we have too many nodes?

Intuition: randomly sample a fraction of the nodes connected to the node under observation. We gain computational efficiency, but may lose expressiveness.

This works particular well in graphs with large fanout/in.

4 Training a GNN

Roughly:

1. Start with input graph
2. create the Graph Neural Network
3. Find the Node embeddings
4. Prediction Head
5. Predictions (to Loss function and Evaluation metrics)
6. Loss function
7. Evaluation Metrics
8. Labels (to Loss function and Evaluation metrics)

4.1 Prediction Head

output of the GNN.

Node level prediction: we can directly make predictions using the node embeddings. After the GNN computation, we have $d - dim$ node embeddings: $\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$. We might want to classify the nodes into k classes. We need to make the node embeddings $h_v^{(L)} \in \mathbb{R}^d$ to the predictions $\hat{y}_v \in \mathbb{R}^k$.

For edge-level tasks we need to consider pairs of nodes. An approach is concatination and applying a linear function to get $\hat{y}_{uv}$. Or we could use the dot product: $\hat{y}_{uv} = (h_u^{(L)})^T h_v^{(L)}$. This can only predict the existence of an edge, so just one-way. K-way prediction is similar to multi-head attention.

$\hat{y}_{uv}^{(k)} = (h_u^{(L)})^T W^{(1)} h_v^{(L)}$

$\hat{y}_{uv} = Concat(\hat{y}_{uv}^{(k)} \forall k) \in \mathbb{R}$

Graph-level predictions: We need to take the node embeddings and somehoe combine them into a graph level prediction.

1. Gobal mean pooling: $\hat{y}_G = Mean(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$.
2. Global max pooling:$\hat{y}_G = Max(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$.
3. Global sum pooling:$\hat{y}_G = Sum(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$.

These are OK on small graphs. But Global pooling over a large graphs loses a lot of information. Instead, we can used hierarchical pooling. In effect, we are splitting the graph into communities that we evaluate and then aggregrate to get to the prediction head. The partitioning can be learned [**?**].

4.2 predictions

We talk about, *supervised* and *unsupervised* (AKA *self-supervised*).

4.2.1 Supervised labels on graphs

- Node labels y_v: E.g., in a citation network, which subject area does a node belong to
- Edge labels y_{uv}: E.g., in a transaction network,whether an edge is fraudulent
- Graph labels y_G: E.g., among molecular graphs, the drug likeness of graphs

Try to formulate your task as one of these so that reusing this research can benefit us.

4.2.2 unsupervised labeles on graphs

- Node-level y_v: Node statistics such as clustering coefficient, degree, Pagerank, etc.
- Edge-level y_{uv}: Link prediction hide the edge between two nodes, then predict there should be a link.
- Graph-level y_G: Graph statistics such as predict if two graphs are isomorphic

4.3 Loss function

Given N data points:

- Node-level: prediction $\hat{y}_v^{(i)}$, label $y_v^{(i)}$
- Edge-level: prediction $\hat{y}_{uv}^{(i)}$, label $y_{uv}^{(i)}$
- Graph-level: prediction $\hat{y}_G^{(i)}$, label $y_G^{(i)}$
- Prediction $\hat{y}^{(i)}$, label $y^{(i)}$ refers to all levels
- Classification: labels $y^{(i)}$ with discrete value: What category does a (something) belong to.
- Regression: labels $y^{(i)}$ with continuous value: What is the likliness of a (something).

These will need different loss functions and evaluation metrics.

4.3.1 Cross entropy

This is a very common loss function.

$CE(y^{(i)}, \hat{y}^{(i)}) = -\sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$ with the i-th data point and j-th target.

Where,

$y^{(i)} \in \mathbb{R}^K$ = one-hot label encoding. E.g.,

0	0	1	0	0

$\hat{y}^{(i)} \in \mathbb{R}^K$ = predicted one-hot label encoding after Softmax.

0.1	0.3	0.4	0.1	0.1

(Note: predicted values are probabilities and must add up to 1.0.)

What we are looking for is to match the predictions to the labeling. The highest probability is the one that matches the label.

The total loss over all N training examples is:

$Loss = \sum_{i=1}^{N} CE(y^{(i)}, \hat{y}^{(i)})$

4.3.2 Mean square Error

For Regression tasks we can use *Mean Squared Error* (MSE) loss function AKA *L2 loss*.

useful for finding node ordering, rather than classification

K-way regression for data point (i):

$MSE(y^{(i)}, \hat{y}^{(i)}) = -\sum_{j=1}^{K} (y_j^{(i)} - \hat{y}_j^{(i)})^2$ with the i-th data point and j-th target.

Where,

$y^{(i)} \in \mathbb{R}^K$ = Real-valued vector of targets. E.g.,

1.4	2.3	1.0	0.5	0.6

$\hat{y}^{(i)} \in \mathbb{R}^K$ = Real valued vector of predictions. E.g.,

0.9	2.8	2.0	0.3	0.8

The total loss over all N training examples is:

$Loss = \sum_{i=1}^{N} MSE(y^{(i)}, \hat{y}^{(i)})$

4.4 Evaluation metrics

RMSE: Root Mean Square Error

$RMSE(y^{(i)}, \hat{y}^{(i)}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2}$

MAE: Mean Absolute Error

$MAE(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{N} \sum_{i=1}^{N} |y^{(i)} - \hat{y}^{(i)}|$

References

[1] Grover and Leskovovec, *???*, 2016.

[2] Graetzer George, *Math Into LaTeX*, Birkuser Boston; 3 edition (June 22, 2000).