Lecture 5 Notes: GNN Training

October 10, 2024

1 Recap

1.1 GNN Layer Structure

A GNN layer involves two key operations:

- **Message Passing**: Transforms the signal at each node $m_u^{(l-1)}$, where m_u is the feature vector of node u at layer $l-1$:

$$m_u^{(l)} = MSG^{(l)}(h_u^{(l-1)}), u \in N(v) \cup v$$

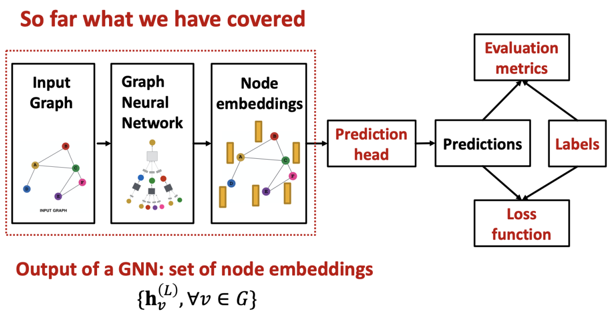
- **Aggregation Operation**: Combines signals from neighboring nodes $\mathcal{N}(v)$ of the target node v:

$$h_v^{(l)} = \text{Aggregate}\left(\{m_u^{(l)}, u \in \mathcal{N}(v)\}, m_v^{(l)}\right)$$

1.2 Layer Expressivity

To enhance expressivity and prevent **over-smoothing**, multiple layers are connected using **Skip Connections** which allows for bypassing intermediate layers to connect earlier outputs directly.

2 Predictions with GNNs



We need to be able to predict about node-level tasks, edge-level tasks, and graph-level tasks. Each of these tasks require different prediction heads.

At the node level, we can directly make predictions using node embeddings:

$$\hat{y} = \text{head}_{\text{node}}(h_v^{(l)}) = W^{(H)} h_v^{(l}.$$

2.1 Edge-Level Predictions

For edge-level predictions, the model takes a pair of node embeddings $H_u^{(l)}$ and $H_v^{(l)}$. Several methods can be used to combine these embeddings:

- **Concatenation and Linear Projection**:

$$\hat{y}_{uv} = \text{Linear}(\text{Concat}(h_u^{(l)}, h_v^{(l)})).$$

- **Dot Product**:

$$\hat{y}_{uv} = h_u^{(l)} \cdot h_v^{(l)}$$

  This operation yields a scalar score, useful for binary edge classification but may be extended using:

$$\hat{y}_{uv} = H_u^{(l)T} \cdot W \cdot H_v^{(l)}$$

  where W is a learned matrix, allowing for higher-dimensional embeddings.

2.2 Graph-Level Predictions

To obtain a graph-level representation, node embeddings are pooled using methods like the following. This is quite similar to the aggregation step that we have learned previously.

- **Global Mean Pooling**:

$$\hat{y}_g = \text{Mean}(h_v^{(L)} \in R^d, \forall v \in G)$$

- **Global Max Pooling**:

$$\hat{y}_g = \text{Max}(h_v^{(L)} \in R^d, \forall v \in G)$$

- **Sum Pooling**:

$$\hat{y}_g = \text{Sum}(h_v^{(L)} \in R^d, \forall v \in G)$$

These pooling approaches are great for smaller graphs, but not so great over a bigger graph, because we lose a lot of information. This is because in bigger graphs there are a lot more options/ways of getting to the same value. To solve this, we aggregate all the node embeddings hierarchically. Specifically, we discuss the DiffPool idea, where we leverage two independent GNNs at each level. GNN A computes node embeddings while GNN B computes the cluster that a node belongs to. These GNNs can be executed in parallel.

For each pooling layer, we use clustering assignments from GNN B to aggregate node embeddings generate by GNN A and then we create a single new node for each cluster, maintaining inter-cluster edges.

3 Training GNNs

There are two types of learning: supervised and unsupervised. In supervised learning, labels come from external sources, while in unsupervised, signals come from graphs themselves. We have two main tasks with GNNs: classification and regression.

3.1 Classification

Classification tasks are tasks in which you are trying to predict a discrete value from a set of values. For example, seeing which category something belongs to. For classification tasks, cross-entropy is a very common loss function. In this equation, we are looking at the ith data point and jth class.

$$\mathrm{CE}\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right) = -\sum_{j=1}^{K} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)$$

3.2 Regression

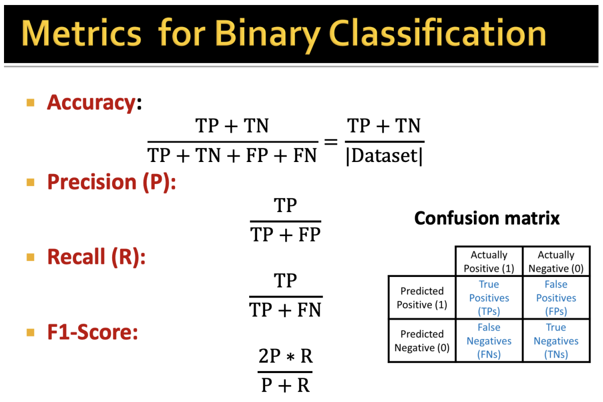
Regression tasks are when you predict a continuous value. For example, predicted the drug-likeness of a molecular graph. In regression tasks, we use Mean Squared Error aka L2 Loss.

$$\mathrm{MSE}\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right) = \sum_{j=1}^{K} \left(y_j^{(i)} - \hat{y}_j^{(i)}\right)^2$$

3.3 Evaluation Metrics

There are two main measures of success of a GNN: Accuracy and ROC AUC. In general, we can use standard evaluation metrics for GNNs, like Root mean squared error and mean absolute error.

For classification tasks, we can report accuracy or precision/recall. Below are the metrics specifically for Binary Classification.



Metrics for Binary Classification

- **Accuracy:**

$$\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}=\frac{\mathrm{TP}+\mathrm{TN}}{|\mathrm{Dataset}|}$$

- **Precision (P):**

$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$$

- **Recall (R):**

$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$$

- **F1-Score:**

$$\frac{2\mathrm{P}*\mathrm{R}}{\mathrm{P}+\mathrm{R}}$$

Confusion matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

We can also create a ROC curve, which captures the tradeoff in TPR and FPR as the classification threshold is varied for a binary classifier.

4 Setting up GNN Prediction Tasks

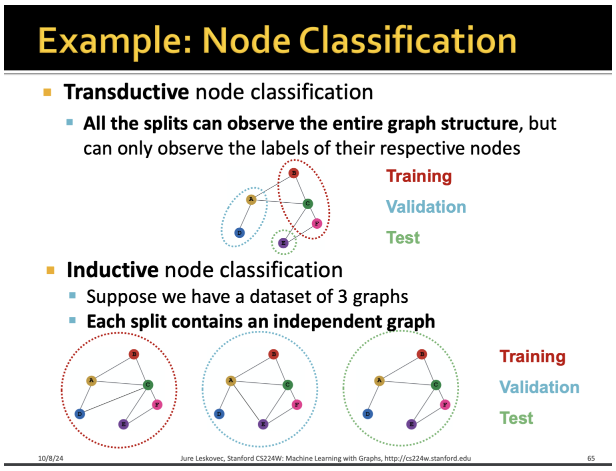
Generally for ML training, we need to come up with a way to stratify our data into a training set, validation set, and test set. But how do we do that in Graphs? The data points are not independent.

Options:

1. Transductive Setting: The input graph can be observed in all the dataset splits. We only split the node labels

2. Inductive Setting: we break the edges between splits to get multiple graphs.

In the transductive setting, the three sets are on the same graphs and we observe the entire data but split the labels. In the inductive case, we can only observe the graphs within the split. Below as an example in practice.



Example: Node Classification

- **Transductive** node classification
  - **All the splits can observe the entire graph structure,** but can only observe the labels of their respective nodes

- **Inductive** node classification
  - Suppose we have a dataset of 3 graphs
  - **Each split contains an independent graph**

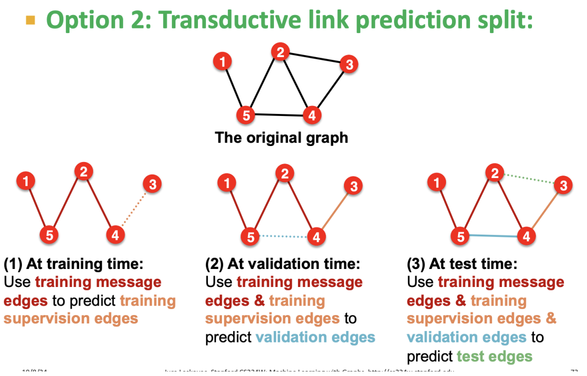
10/8/24 Jure Leskovec, Stanford CS224W: Machine Learning with Graphs, http://cs224w.stanford.edu 65

For graph classification, only the inductive setting is well defined, because we have to test on unseen graphs.

For link prediction, we can hide some edges from the GNN and see if the GNN can predict them. To set this problem up, we assign 2 types of edges in the original graph: message and supervision edges. Then, we split edges into train/validation/test. We have a couple of options from here.

1. Inductive link prediction split, where each split has their independent graph.

2. transductive link prediction (default), where the entire graph can be observed in all dataset splits. But since edges are both part of graph structure and the supervision, we need to hold out validation and test edges. And for training, we also need to hold out supervision edges (see picture below)



Option 2: Transductive link prediction split:

The original graph

(1) At training time: Use training message edges to predict training supervision edges

(2) At validation time: Use training message edges & training supervision edges to predict validation edges

(3) At test time: Use training message edges & training supervision edges & validation edges to predict test edges

After training, the supervision edges are known to GNN and then in the future, these edges should be used in message passing at validation time.