Graph Neural Network

From

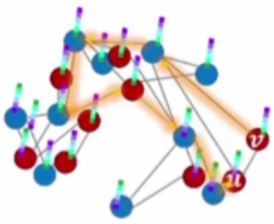
CS224W | Home (stanford.edu)

and

Stanford CS224W: Machine Learning with Graphs - YouTube

3. Node Embeddings (Shallow encoding: Deepwalk and Node2Vec)

DeepWalk



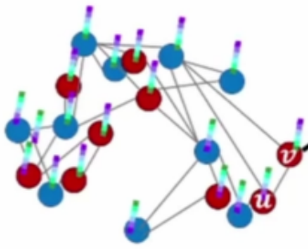
$S_G(u, v) = p(u|v)$

Similarity between nodes **u** and v is defined as *the probability* of visiting **u** if do a random walk on graph starting at node v

You can define the length of walk **T**

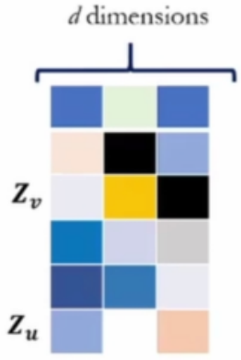
This process is repeated several times, then we can estimate the *probability* of visiting desired node

DeepWalk - Pipeline



ENC(v) ★ z

Goal: $S_G(u, v) \approx S_E(z_u, z_v)$



Embedding Space

$S_G(u, v) = p(u|v)$

$$S_E(z_u, z_v) = \frac{\exp(z_u^T z_v)}{\sum_{k \in V} \exp(z_u^T z_v)}$$

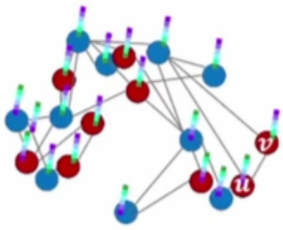
Start with random values for z_u and z_V, then find correct embedding space such that a loss is minimized

$$\ell = \sum_{(u,v) \in D} -log\,(S_E(z_u, z_v))$$

Younes Sadat Nejad

Node2Vec

~~Unbiased random walk~~



How to explore network

1- **Depth-first search (DFS) –** Explore global representation

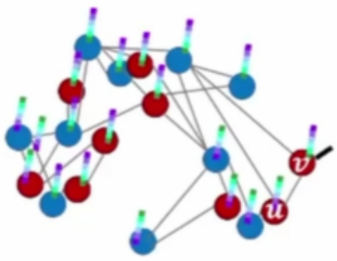
2- **Breadth-first search (BFS)-** Explore local representation

We can set two parameters q_1 and q_2 for probability of returning to the starting node (**local exploration**) and probability of moving away from the starting node (**global exploration**).

You can define the length of walk **T**

This process is repeated several times, then we can estimate the *probability* of visiting desired node

DeepWalk & Node2Vec

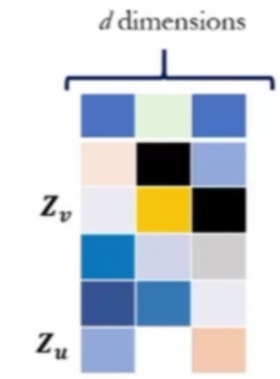


ENC(v) ★ z

Goal: $S_G(u,v) \approx S_E(z_u, z_v)$

$$S_G(u,v) = p(u|v)$$

$$S_E(z_u, z_v) = \frac{\exp(z_u^T z_v)}{\sum_{k \in V} \exp(z_u^T z_v)}$$



Embedding Space

No parameter sharing: Computationally expensive

No semantic information: Feature nodes are not considered

Not Inductive: Cannot predict embedding for unseen data (Inherently Transudative)

5. Label Propagation for Node Classification

- Relational classification: Propagate node labels across the network. Iteratively update probabilities of node belonging to a label class based on its neighbors.

- Iterative Classification: Tabulate the incoming / outcoming predicted label information. Then train a classifier to classify each node based on its **features** as well as **labels** of neighbors; do that iteratively.

- Correct & Smooth

6. Graph Neural Networks 1: GNN Model

Can solve:

- Node classification - Predict a type of a given node
- Link prediction - Predict whether two nodes are linked
- Community detection - Identify densely linked clusters of nodes
- Network similarity - How similar are two (sub)networks

Average neighbor’s previous layer embeddings and matrix multiply the weights. So as long as the embedding length is fixed, the model can handle any number of neighbors.

- **Basic approach:** Average neighbor messages and apply a neural network

Initial 0-th layer embeddings are equal to node features

$$h_v^0 = x_v$$

embedding of v at layer k

$$h_v^{(k+1)} = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \forall k \in \{0, \ldots, K-1\}$$

Total number of layers

Average of neighbor's previous layer embeddings

$$z_v = h_v^{(K)}$$

Embedding after L layers of neighborhood aggregation

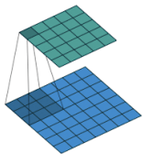
Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.

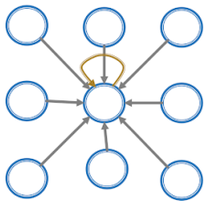
h_v^k: the hidden representation of node v at layer k
- W_k: weight matrix for neighborhood aggregation
- B_k: weight matrix for transforming hidden vector of self

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

GNN formulation: $h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$

CNN formulation: $h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} W_l^u h_u^{(l)} + B_l h_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$

Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)}

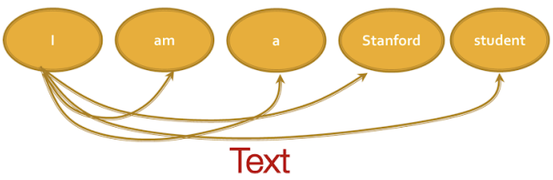
CNN can be seen as a special GNN with fixed neighbor size and ordering:
- **The size of the filter is pre-defined for a CNN.**
- **The advantage of GNN is it processes arbitrary graphs with different degrees for each node.**

CNN is not permutation equivariant.
- **Switching the order of pixels will leads to different outputs.**

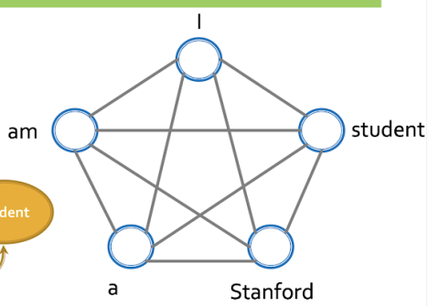
GNN vs. Transformer

Transformer layer can be seen as a special GNN that runs on a fully-connected "word" graph!

Since each word attends to **all the other words**, **the computation graph** of a transformer layer is identical to that of a GNN on the **fully-connected "word" graph**.



Text



(Complete) Graph

7. Graph Neural Networks 2: Design Space

Attention Mechanism

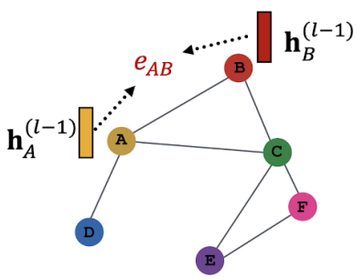
Idea: Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an **attention strategy**:

- Nodes attend over their neighborhoods' message
- Implicitly specifying different weights to different nodes in a neighborhood

- (1) Let a compute **attention coefficients** e_{vu} across pairs of nodes u, v based on their messages:

$$e_{vu} = a(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

  - e_{vu} **indicates the importance of u's message to node v**



$$e_{AB} = a(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)})$$

- **Normalize** e_{vu} into the **final attention weight** α_{vu}
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:

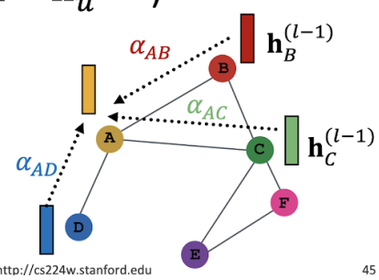
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** α_{vu}

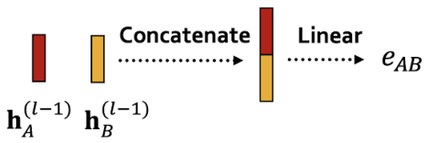
$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Weighted sum using α_{AB}, α_{AC}, α_{AD}:

$$\mathbf{h}_A^{(l)} = \sigma\left(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)}\right)$$



- **What is the form of attention mechanism a?**
  - The approach is agnostic to the choice of a
    - E.g., use a simple single-layer neural network
      - a have trainable parameters (weights in the Linear layer)



$$e_{AB} = a\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)$$
$$= \text{Linear}\left(\text{Concat}\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)\right)$$

  - Parameters of a are trained jointly:
    - Learn the parameters together with weight matrices (i.e., other parameter of the neural net $\mathbf{W}^{(l)}$) in an end-to-end fashion

8. Applications of Graph Neural Networks

- **Supervised labels come from the specific use cases**. For example:
  - **Node labels y_v:** in a citation network, which subject area does a node belong to
  - **Edge labels y_{uv}:** in a transaction network, whether an edge is fraudulent
  - **Graph labels y_G:** among molecular graphs, the drug likeness of graphs

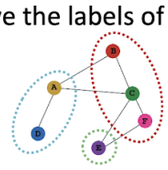
- **The problem:** sometimes we only have a graph, without any external labels
- **The solution:** "self-supervised learning", we can find supervision signals within the graph.
  - For example, we can **let GNN predict the following:**
  - **Node-level y_v.** Node statistics: such as clustering coefficient, PageRank, …
  - **Edge-level y_{uv}.** Link prediction: hide the edge between two nodes, predict if there should be a link
  - **Graph-level y_G.** Graph statistics: for example, predict if two graphs are isomorphic
  - **These tasks do not require any external labels!**

Transductive / Inductive Settings

- **Transductive setting:** training / validation / test sets are **on the same graph**
  - The **dataset consists of one graph**
  - **The entire graph can be observed in all dataset splits, we only split the labels**
  - Only applicable to **node / edge** prediction tasks
- **Inductive setting:** training / validation / test sets are **on different graphs**
  - The **dataset consists of multiple graphs**
  - Each split can **only observe the graph(s) within the split**. A successful model should **generalize to unseen graphs**
  - Applicable to **node / edge / graph** tasks

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

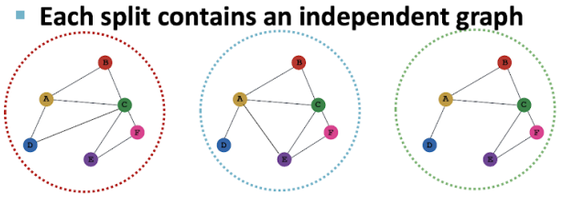


Training

Validation

Test

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



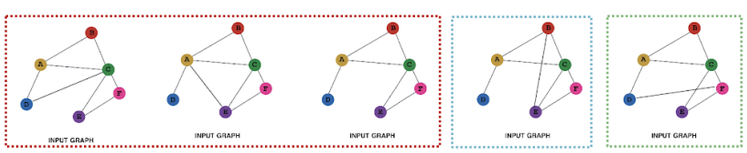
Training

Validation

Test

Example: Graph Classification

- Only the **inductive setting** is well defined for **graph classification**
  - Because **we have to test on unseen graphs**
  - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).

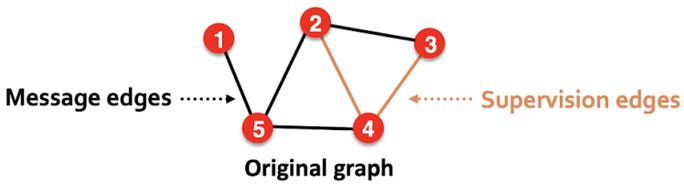


Training

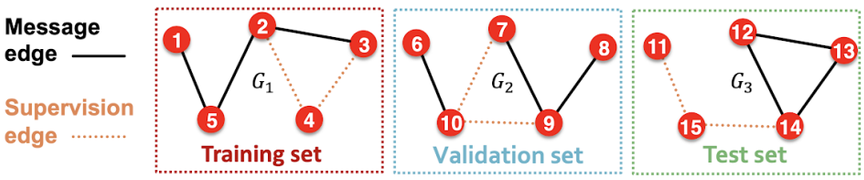
Validation

Test

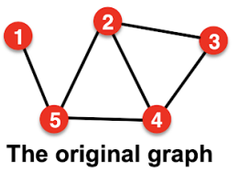
Setting up Link Prediction

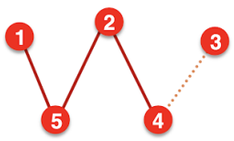


- For link prediction, we will split edges twice
- Step 1: Assign 2 types of edges in the original graph
  - Message edges: Used for GNN message passing
  - Supervision edges: Use for computing objectives
  - After step 1:
    - Only message edges will remain in the graph
    - Supervision edges are used as supervision for edge predictions made by the model, will not be fed into GNN!
- Step 2: Split edges into train / validation / test
- Option 1: Inductive link prediction split
  - Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph
  - In train or val or test set, each graph will have 2 types of edges: message edges + supervision edges
    - Supervision edges are not the input to GNN

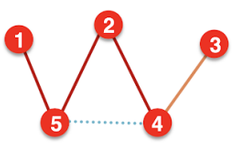


- Option 2: Transductive link prediction split:

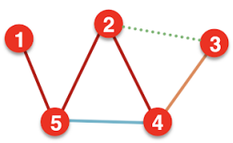




(1) At training time: Use training message edges to predict training supervision edges



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

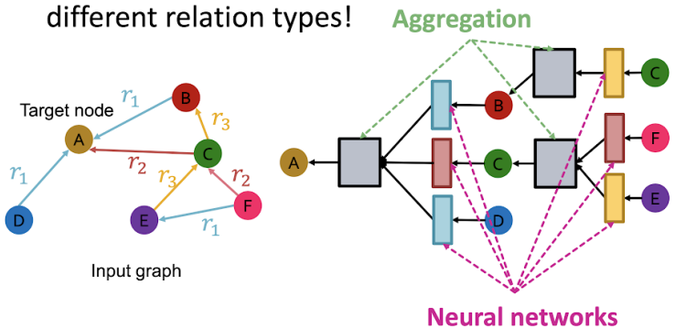


(3) At test time: Use training message edges & training supervision edges & validation edges to predict test edges

10. Knowledge Graph Embeddings

Relational GCN (3)

- **What if the graph has multiple relation types?**
- Use different neural network weights for different relation types!



- **Relational GCN (RGCN):**

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\sum_{r\in R}\sum_{u\in N_v^r}\frac{1}{c_{v,r}}\mathbf{W}_r^{(l)}\mathbf{h}_u^{(l)} + \mathbf{W}_0^{(l)}\mathbf{h}_v^{(l)}\right)$$

- **How to write this as Message + Aggregation?**
- **Message:**
  - Each neighbor of a given relation:

$$\mathbf{m}_{u,r}^{(l)} = \frac{1}{c_{v,r}}\mathbf{W}_r^{(l)}\mathbf{h}_u^{(l)}$$

  - Self-loop:

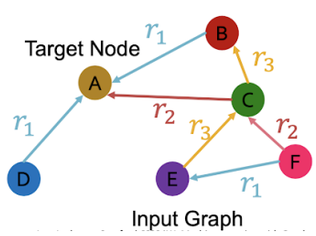
$$\mathbf{m}_v^{(l)} = \mathbf{W}_0^{(l)}\mathbf{h}_v^{(l)}$$

Normalized by node degree of the relation $c_{v,r} = |N_v^r|$

- **Aggregation:**
  - Sum over messages from neighbors and self-loop, then apply activation
  - $\mathbf{h}_v^{(l+1)} = \sigma\left(\text{Sum}\left(\left\{\mathbf{m}_{u,r}^{(l)}, u \in N(v)\right\} \cup \left\{\mathbf{m}_v^{(l)}\right\}\right)\right)$

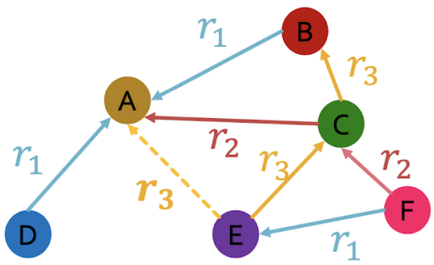
Example: Entity/Node Classification

- **Goal: Predict the label of a given node**
- **RGCN** uses the representation of the final layer:
  - If we predict the class of **node A** from **k classes**.
  - Take the **final layer (prediction head)**: $\mathbf{h}_A^{(L)} \in \mathbb{R}^k$, each item in $\mathbf{h}_A^{(L)}$ represents **the probability of that class**.



Example: Link Prediction

Training:



Input Graph

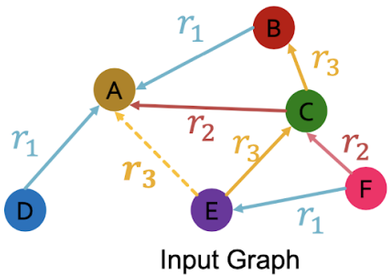
1. Use RGCN to score the training supervision edge (E, r_3, A)
2. Create a negative edge by perturbing the supervision edge (E, r_3, B)
   - Corrupt the tail of (E, r_3, A)
     - e.g., (E, r_3, B), (E, r_3, D)

training supervision edges: (E, r_3, A)
training message edges: all the rest existing edges (solid lines)

Note the negative edges should NOT belong to training message edges or training supervision edges!
e.g., (E, r_3, C) is NOT a negative edge

(1) Use training message edges to predict training supervision edges

Training:



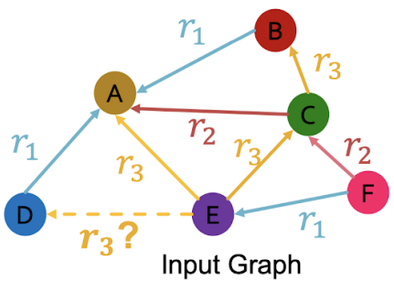
Input Graph

1. Use RGCN to score the training supervision edge (E, r_3, A)
2. Create a negative edge by perturbing the supervision edge (E, r_3, B)
3. Use GNN model to score negative edge
4. Optimize a standard cross entropy loss (as discussed in Lecture 6)
   1. Maximize the score of training supervision edge
   2. Minimize the score of negative edge

$$\ell = -\log \sigma\left(f_{r_3}(h_E, h_A)\right) - \log(1 - \sigma(f_{r_3}(h_E, h_B)))$$

Evaluation:

- Validation time as an example, same at the test time



Input Graph

Evaluate how the model can predict the validation edges with the relation types.
Let's predict validation edge (E, r_3, D)
Intuition: the score of (E, r_3, D) should be higher than all (E, r_3, v) where (E, r_3, v) is NOT in the training message edges and training supervision edges, e.g., (E, r_3, B)

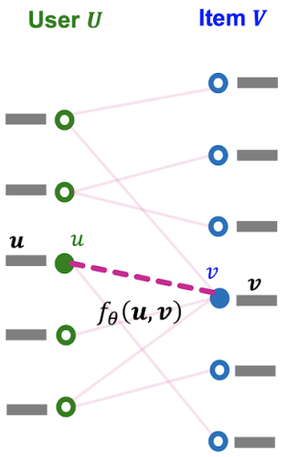
validation edges: (E, r_3, D)
training message edges & training supervision edges: all existing edges (solid lines)

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

13. GNNs for Recommender Systems

Embedding-Based Models

- We consider **embedding-based models** for scoring user-item interactions.
  - For each user $u \in U$, let $\boldsymbol{u} \in \mathbb{R}^D$ be its D-dimensional embedding.
  - For each item $v \in V$, let $\boldsymbol{v} \in \mathbb{R}^D$ be its D-dimensional embedding.
  - Let $f_\theta(\cdot,\cdot)$: $\mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ be a parametrized function.
  - Then, $\text{score}(u, v) \equiv f_\theta(\boldsymbol{u}, \boldsymbol{v})$



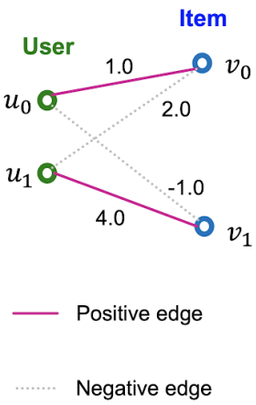
Training Objective

- Embedding-based models have three kinds of parameters:
  - An encoder to generate user embeddings $\{\boldsymbol{u}\}_{u \in U}$
  - An encoder to generate item embeddings $\{\boldsymbol{v}\}_{v \in V}$
  - Score function $f_\theta(\cdot,\cdot)$
- **Training objective**: Optimize the model parameters to achieve high recall@K on *seen* (i.e., *training*) user-item interactions
  - We hope this objective would lead to high recall@K on *unseen* (i.e., *test*) interactions.

Binary Loss

- **Issue**: In the binary loss, the scores of ***ALL*** positive edges are pushed higher than those of ***ALL*** negative edges.
- This would unnecessarily penalize model predictions even if the training recall metric is perfect.

- **Let's consider the simplest case:**
  - Two users, two items
  - Metric: Recall@1.
  - A model assigns the score for every user-item pair (as shown in the right).
- Training **Recall@1 is 1.0** (perfect score), because v_0 (resp. v_1) is correctly recommended to u_0 (resp. u_1).
- However, **the binary loss would still penalize the model prediction** because the negative (u_1, v_0) edge gets the higher score than the positive edge (u_0, v_0).

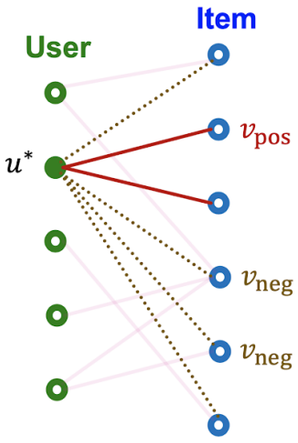


- **Key insight**: The binary loss is **non-personalized** in the sense that the **positive/negative edges are considered *across ALL users at once*.**
- However, the recall metric is inherently **personalized (defined for each user)**.
  - The non-personalized binary loss is overly-stringent for the personalized recall metric.

Introducing BPR

- **Bayesian Personalized Ranking (BPR) loss** is a personalized surrogate loss that aligns better with the recall@K metric.

- **Mini-batch training for the BPR loss:**
  - In each mini-batch, we sample a subset of users $\boldsymbol{U}_{\text{mini}} \subset \boldsymbol{U}$.
    - For each user $u^* \in \boldsymbol{U}_{\text{mini}}$, we sample one positive item v_{pos} and a set of sampled negative items $V_{\text{neg}} = \{v_{\text{neg}}\}$.
  - The mini-batch loss is computed as

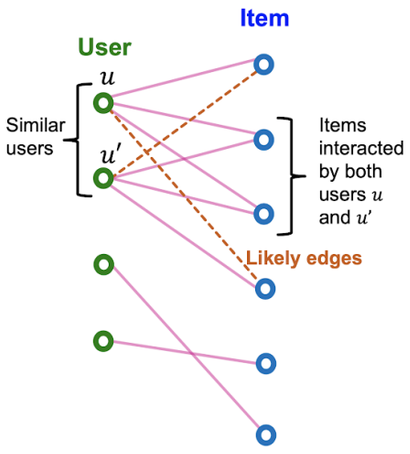


$$\frac{1}{|\boldsymbol{U}_{\text{mini}}|}\sum_{u^* \in \boldsymbol{U}_{\text{mini}}} \frac{1}{|\boldsymbol{V}_{\text{neg}}|}\sum_{v_{\text{neg}} \in \boldsymbol{V}_{\text{neg}}} -\log\left(\sigma\left(f_\theta(u^*, \boldsymbol{v}_{\text{pos}})\right) - \sigma\left(f_\theta(u^*, \boldsymbol{v}_{\text{neg}})\right)\right)$$

Average over users in the mini-batch

Why Embedding Models Work?

- **Underlying idea: Collaborative filtering**
  - Recommend items for a user by **collecting preferences of many other similar users.**
  - **Similar users tend to prefer similar items.**
- **Key question: How to capture similarity between users/items?**



- Embedding-based models can capture similarity of users/items!
  - **Low-dimensional embeddings *cannot* simply memorize all user-item interaction data.**
  - Embeddings are forced to **capture similarity between users/items to fit the data.**
  - This allows the models to make effective prediction on *unseen* user-item interactions.

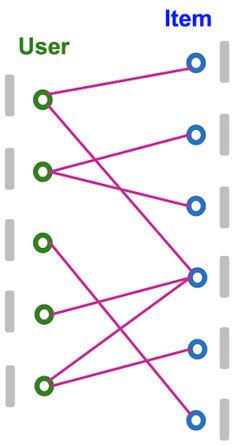
Limitations of MF

- The model itself does ***not explicitly*** capture graph structure
  - The graph structure is ***only implicitly*** captured in the training objective.
- Only the **first-order graph structure** (i.e., edges) is captured in the training objective.
  - **High-order graph structure** (e.g., K-hop paths between two nodes) is ***not* explicitly captured.**

- **Neural Graph Collaborative Filtering (NGCF)** [Wang et al. 2019], **LightGCN** [He et al. 2020]
  - Improve the conventional collaborative filtering models (i.e., matrix factorization) by explicitly modeling graph structure using GNNs.
- **PinSAGE** [Ying et al. 2018]
  - Use GNNs to generate high-quality embeddings by simultaneously capturing rich node attributes (e.g., images) and the graph structure.

NGCF

- Set the shallow learnable embeddings as the initial node features.
  - For every user $u \in \boldsymbol{U}$, set $\boldsymbol{h}_u^{(0)}$ as the user's shallow embedding.
  - For every item $v \in \boldsymbol{V}$, set $\boldsymbol{h}_v^{(0)}$ as the item's shallow embedding.



- Iteratively update node embeddings using neighboring embeddings.

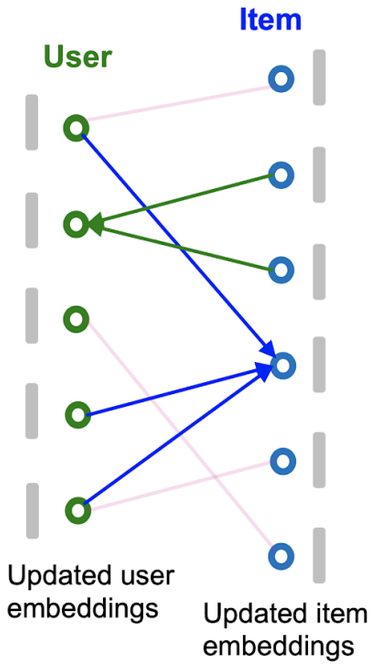
$$\boldsymbol{h}_v^{(k+1)} = \text{COMBINE}\left(\boldsymbol{h}_v^{(k)}, \text{AGGR}\left(\left\{\boldsymbol{h}_u^{(k)}\right\}_{u \in N(v)}\right)\right)$$

$$\boldsymbol{h}_u^{(k+1)} = \text{COMBINE}\left(\boldsymbol{h}_u^{(k)}, \text{AGGR}\left(\left\{\boldsymbol{h}_v^{(k)}\right\}_{v \in N(u)}\right)\right)$$

High-order graph structure is captured through iterative neighbor aggregation.

Different architecture choices are possible for AGGR and COMBINE.

- $\text{AGGR}(\cdot)$ can be $\text{MEAN}(\cdot)$
- $\text{COMBINE}(\boldsymbol{x}, \boldsymbol{y})$ can be $\text{ReLU}(\text{Linear}(\text{Concat}(\boldsymbol{x}, \boldsymbol{y})))$



- After K rounds of neighbor aggregation, we get the **final user/item embeddings** $\boldsymbol{h}_u^{(K)}$ and $\boldsymbol{h}_v^{(K)}$.
- For all $u \in \boldsymbol{U}, v \in \boldsymbol{V}$, we set $\boldsymbol{u} \leftarrow \boldsymbol{h}_u^{(K)}, \boldsymbol{v} \leftarrow \boldsymbol{h}_v^{(K)}$.
- Score function is the inner product

$$\text{score}(u, v) = \boldsymbol{u}^T \boldsymbol{v}$$

LightGCN

- **Recall**: Diffusion matrix of C&S.
- Let $\boldsymbol{D}$ be the degree matrix of $\boldsymbol{A}$.
- Define the normalized adjacency matrix $\widetilde{\boldsymbol{A}}$ as

$$\widetilde{\boldsymbol{A}} \equiv \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$$

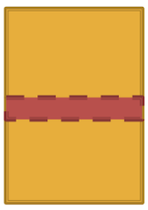
Note: Different from the original GCN, self-connection is omitted here.

- Let $\boldsymbol{E}^{(k)}$ be the embedding matrix at k-th layer.
- Each layer of GCN's aggregation can be written in a matrix form:

$$\boldsymbol{E}^{(k+1)} = \mathrm{ReLU}\left(\widetilde{\boldsymbol{A}}\boldsymbol{E}^{(k)}\boldsymbol{W}^{(k)}\right)$$

Neighbor aggregation — Learnable linear transformation

Matrix of node embeddings $\boldsymbol{E}^{(k)}$



Each row stores node embedding

- Removing ReLU significantly simplifies GCN!

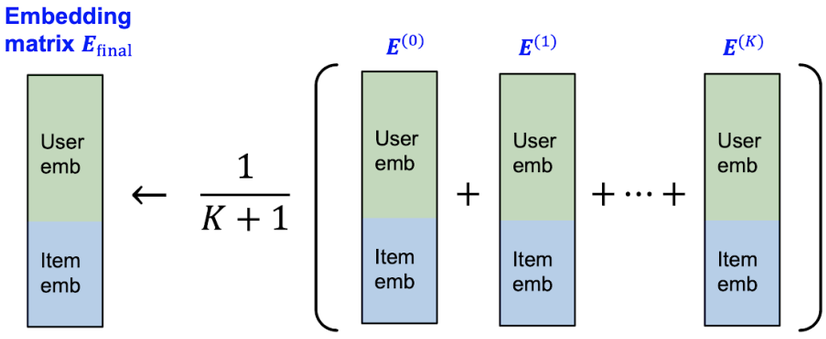
$$\boldsymbol{E}^{(K)} = \widetilde{\boldsymbol{A}}^{K}\boldsymbol{E}\,\boldsymbol{W}$$

$$\boldsymbol{W} \equiv \boldsymbol{W}^{(0)} \cdots \boldsymbol{W}^{(K-1)}$$

Diffusing node embeddings along the graph
(similar to C&S that diffuses soft labels along the graph)

- **Algorithm**: Apply $\boldsymbol{E} \leftarrow \widetilde{\boldsymbol{A}}\,\boldsymbol{E}$ for K times.
  - Each matrix multiplication diffuses the current embeddings to their one-hop neighbors.

- Average the embedding matrices at different scales.



- LightGCN simplifies NGCF by **removing the learnable parameters of GNNs.**
- **Learnable parameters are all in the shallow input node embeddings.**

- The embedding propagation of LightGCN is closely related to GCN/C&S.
- **Recall**: GCN/C&S (neighbor aggregation part)

$$\boldsymbol{h}_v^{(k+1)} = \sum_{u \in N(v)} \frac{1}{\sqrt{d_u}\sqrt{d_v}} \cdot \boldsymbol{h}_u^{(k)}$$

Node degree

  - Self-loop is added in the neighborhood definition.
- LightGCN uses the same equation except that
  - Self-loop is *not* added in the neighborhood definition
  - Final embedding takes the average of embeddings from all the layers: $\boldsymbol{h}_v = \frac{1}{K+1}\sum_{k=0}^{K} \boldsymbol{h}_v^{(k)}$.