Графовые сети для RecSys

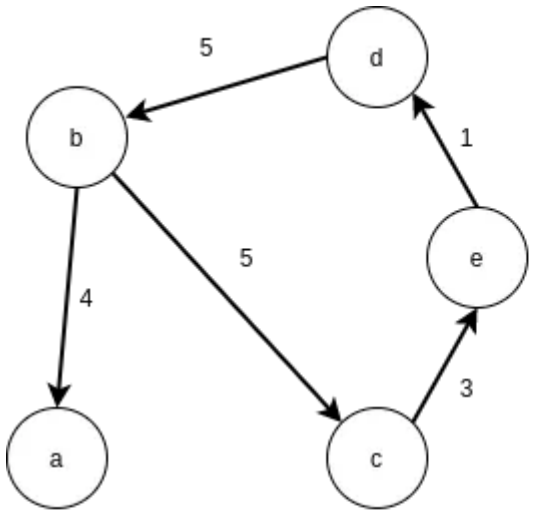


Figure 1a

Homogeneous
directed
weighted graph

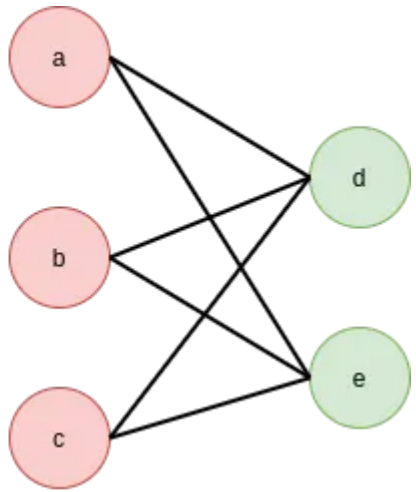
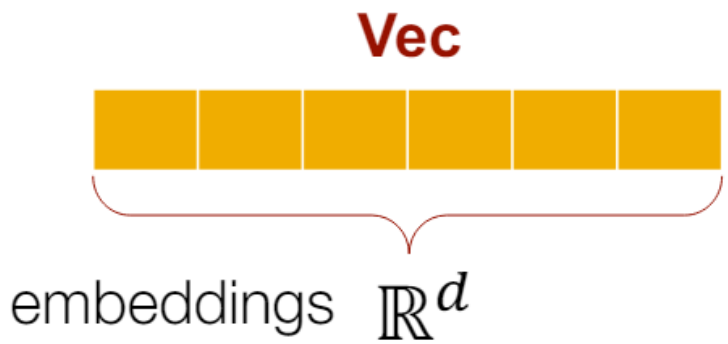


Figure 1b

Bipartite graph
(Heterogeneous
undirected
unweighted
graph)

Why Embedding?

- **Task: Map nodes into an embedding space**
  - Similarity of embeddings between nodes indicates their similarity in the network. For example:
    - Both nodes are close to each other (connected by an edge)
  - Encode network information
  - Potentially used for many downstream predictions



Tasks

- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering
- ….

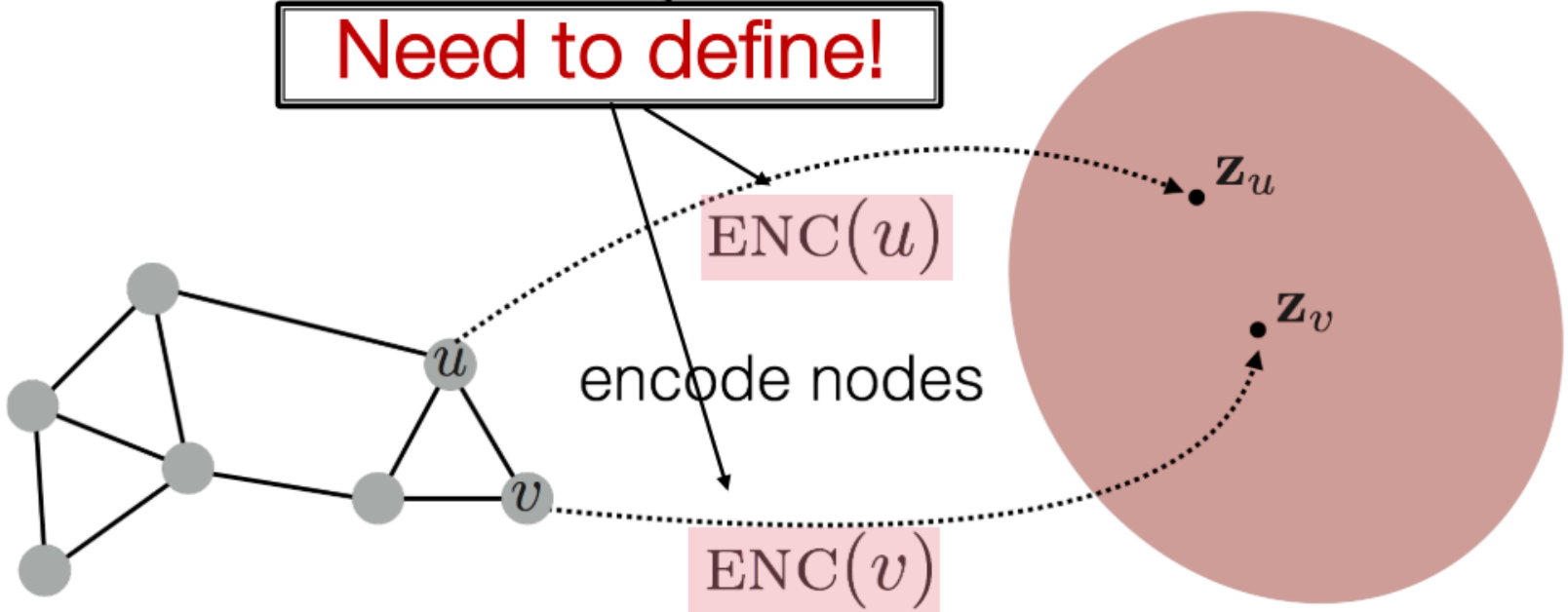
Embedding Nodes

Goal: $\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$

in the original network

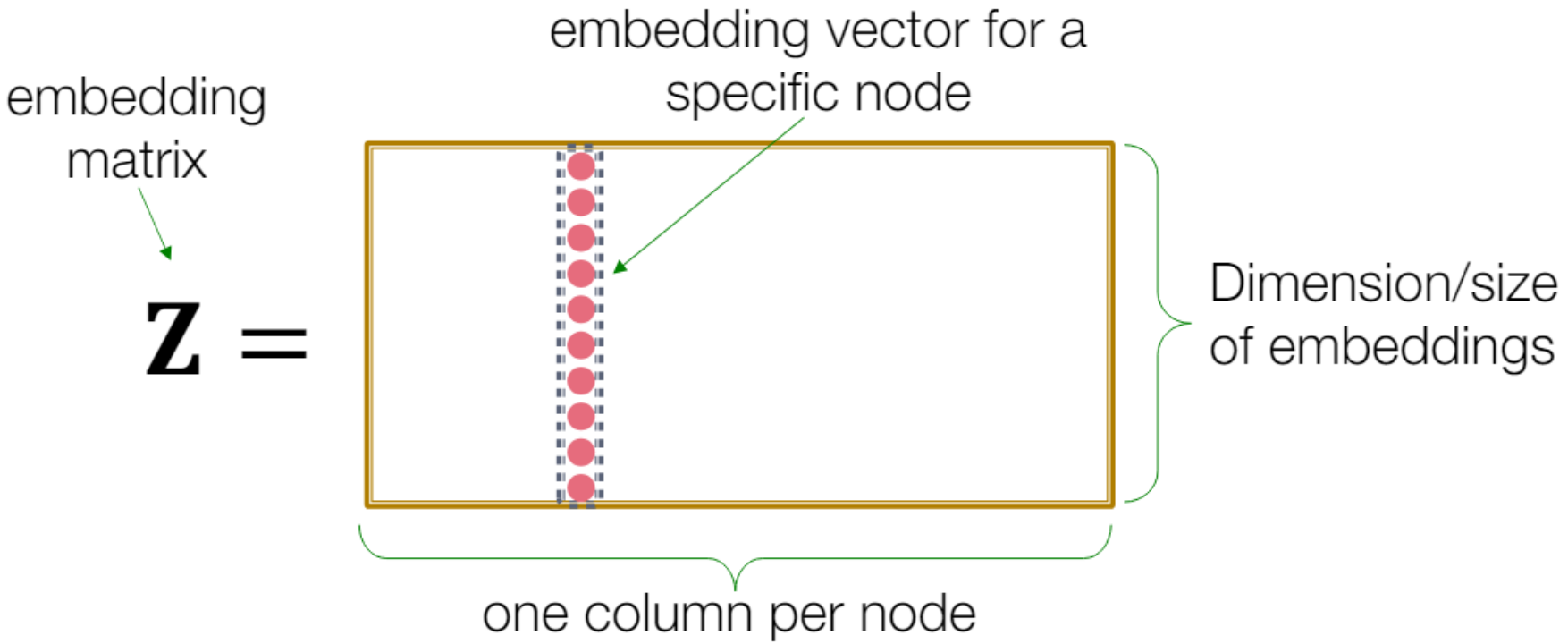
Similarity of the embedding

Need to define!



"Shallow" Encoding

Simplest encoding approach: **encoder is just an embedding-lookup**



Two Key Components

- **Encoder:** maps each node to a low-dimensional vector

$$\text{ENC}(v) = \mathbf{z}_v$$

d-dimensional embedding

node in the input graph

- **Similarity function:** specifies how the relationships in vector space map to the relationships in the original network

$$\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$$

Decoder

Similarity of u and v in the original network

dot product between node embeddings

How to Define Node Similarity?

- Key choice of methods is **how they define node similarity.**
- Should two nodes have a similar embedding if they…
  - are linked?
  - share neighbors?
  - have similar "structural roles"?
- We will now learn node similarity definition that uses **random walks**, and how to optimize embeddings for such a similarity measure.

Note on Node Embeddings

- This is **unsupervised/self-supervised** way of learning node embeddings.
  - We are **not** utilizing node labels
  - We are **not** utilizing node features
  - The goal is to directly estimate a set of coordinates (i.e., the embedding) of a node so that some aspect of the network structure (captured by DEC) is preserved.
- These embeddings are **task independent:**
  - They are not trained for a specific task but can be used for any task.

- Random walk embeddings

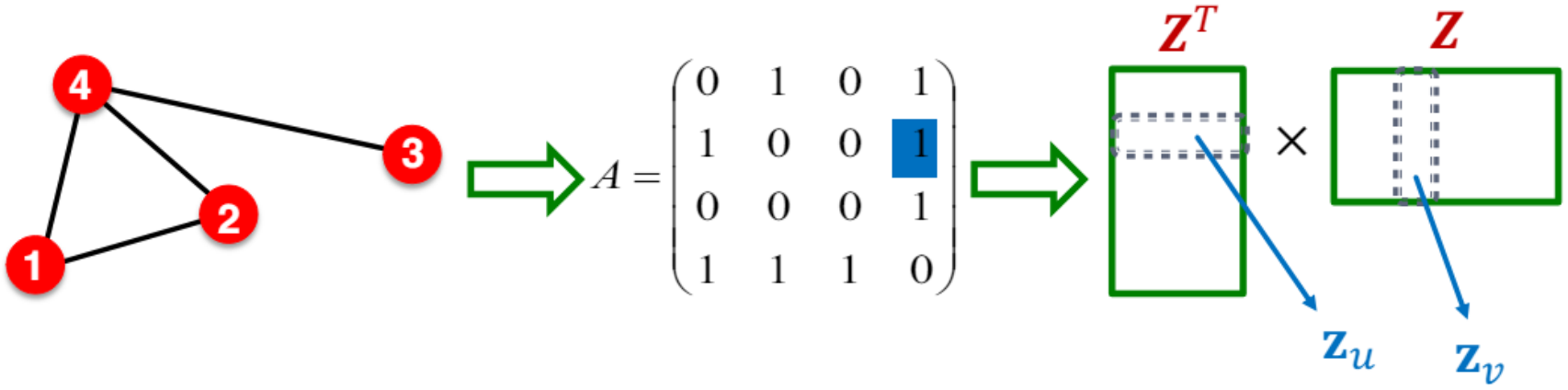
1. Simulate Random Walks
2. An optimization process refines embeddings to maximize closeness for nodes seen together while separating non-co-occurring nodes, often using negative sampling to reduce computation.

$$\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v \approx$$

probability that u and v co-occur on a random walk over the graph

Connection to Matrix Factorization

- Simplest **node similarity**: Nodes u, v are similar if they are connected by an edge
- This means: $\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u = A_{u,v}$ which is the (u, v) entry of the graph adjacency matrix A
- Therefore, $\boldsymbol{Z}^T \boldsymbol{Z} = A$

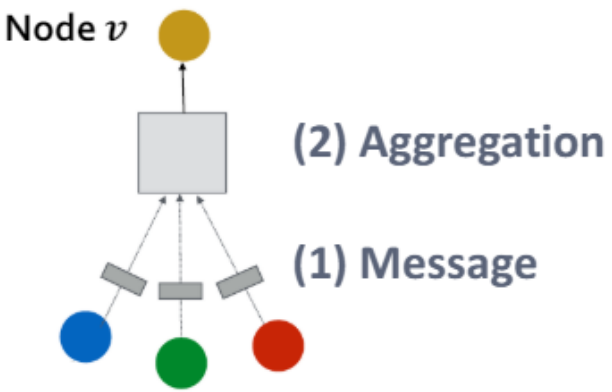


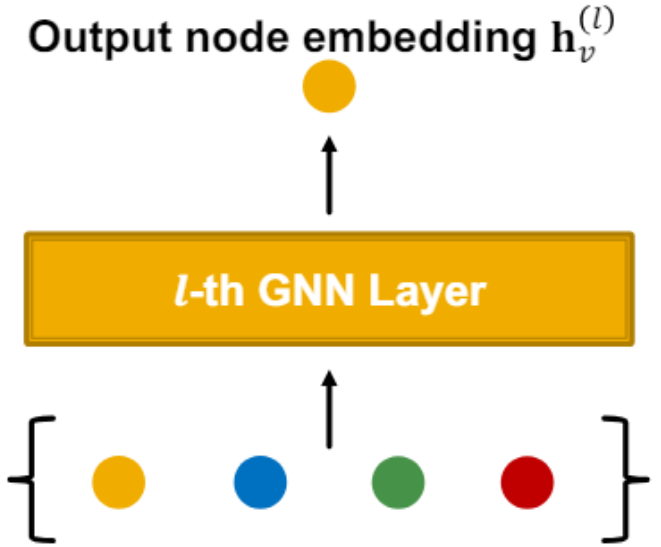
Matrix Factorization

- The embedding dimension d (number of rows in $\boldsymbol{Z}$) is much smaller than number of nodes n.
- Exact factorization $A = \boldsymbol{Z}^{\boldsymbol{T}}\boldsymbol{Z}$ is generally not possible
- However, we can learn $\boldsymbol{Z}$ approximately
- **Objective:** $\min_{\boldsymbol{Z}} \| \mathrm{A} - \boldsymbol{Z}^{T}\boldsymbol{Z} \|_2$
  - We optimize $\boldsymbol{Z}$ such that it minimizes the L2 norm (Frobenius norm) of $\mathrm{A} - \boldsymbol{Z}^{T}\boldsymbol{Z}$
  - Note today we used softmax instead of L2. But the goal to approximate A with $\boldsymbol{Z}^{T}\boldsymbol{Z}$ is the same.
- Conclusion: **Inner product decoder with node similarity defined by edge connectivity is equivalent to matrix factorization of A.**

A Single GNN Layer

- **Idea of a GNN Layer:**
  - Compress a set of vectors into a single vector
  - **Two-step process:**
    - **(1) Message**
    - **(2) Aggregation**





Input node embedding $\mathbf{h}_v^{(l-1)}$, $\mathbf{h}_{u \in N(v)}^{(l-1)}$
(from node itself + neighboring nodes)

T. Kipf, M. Welling. Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

Classical GNN Layers: GCN (1)

- **(1) Graph Convolutional Networks (GCN)**

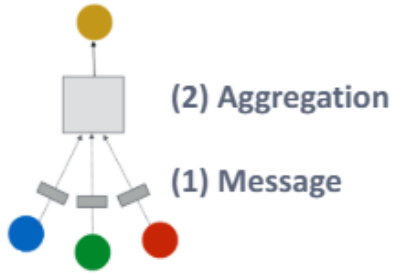
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

- **How to write this as Message + Aggregation?**

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

Message

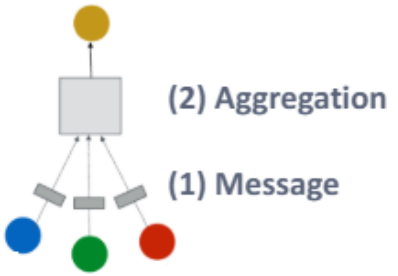
Aggregation



Classical GNN Layers: GCN (2)

- **(1) Graph Convolutional Networks (GCN)**

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



- **Message:**
  - Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$

Normalized by node degree
(In the GCN paper they use a slightly different normalization)

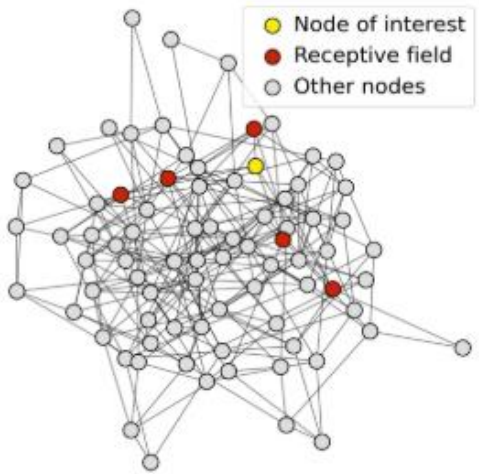
- **Aggregation:**
  - **Sum** over messages from neighbors, then apply activation
  - $\mathbf{h}_v^{(l)} = \sigma\left(\text{Sum}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)\right)$

In GCN the input graph is assumed to have self-edges that are included in the summation.

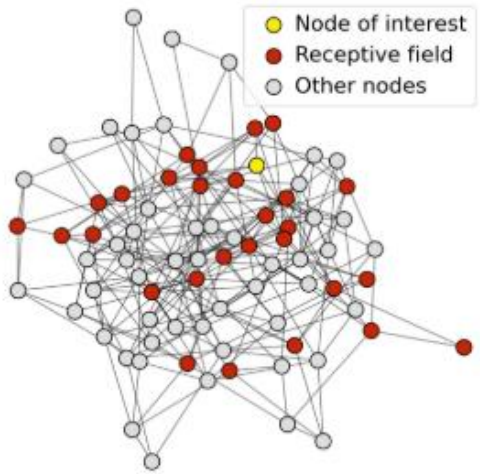
Receptive Field of a GNN

- **Receptive field:** the set of nodes that determine the embedding of a node of interest
  - **In a K-layer GNN, each node has a receptive field of K-hop neighborhood**

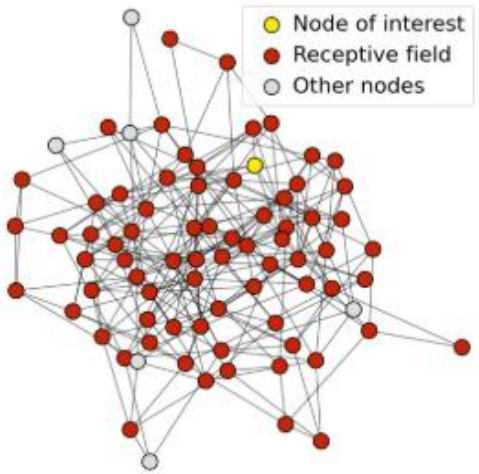
Receptive field for 1-layer GNN



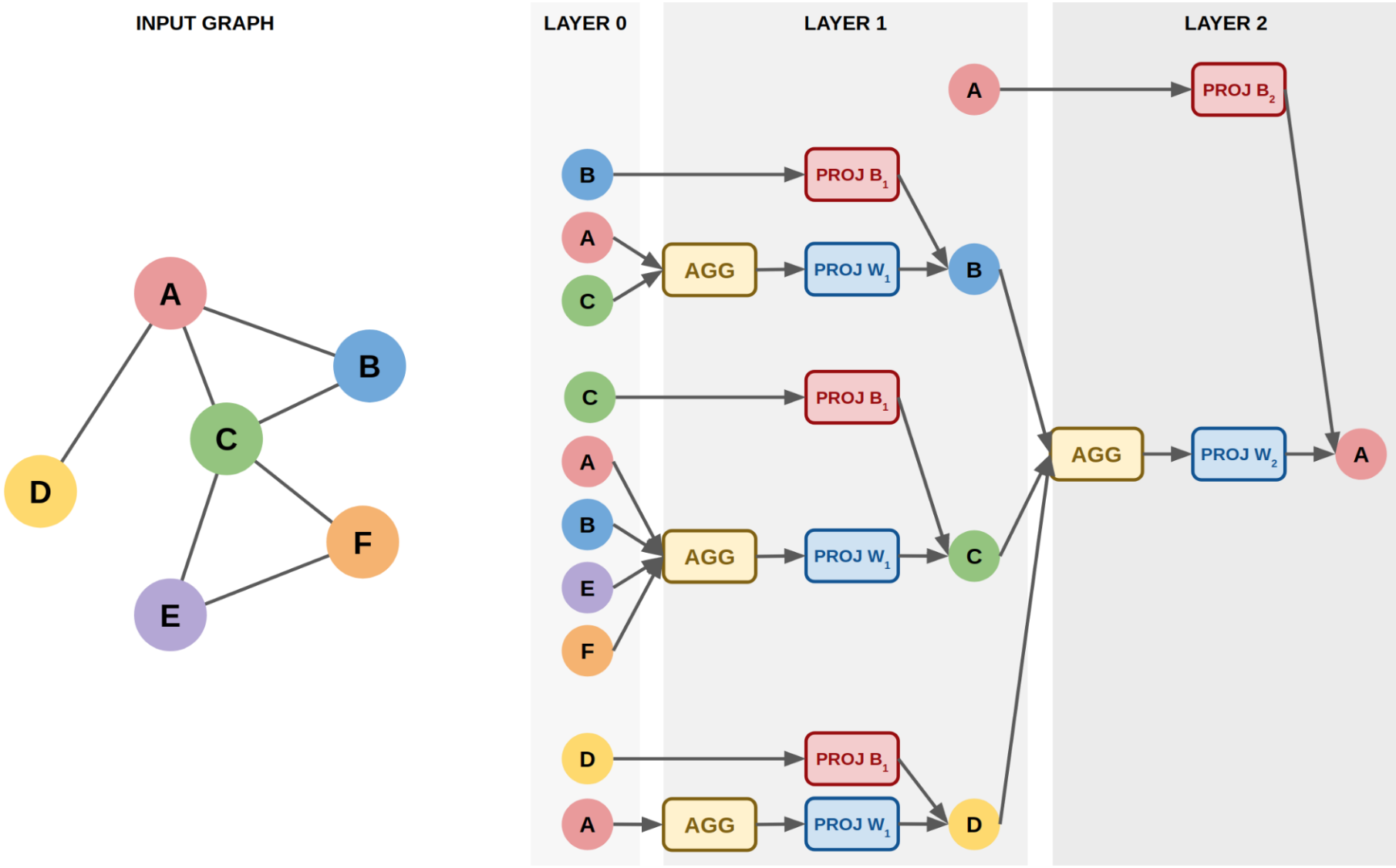
Receptive field for 2-layer GNN



Receptive field for 3-layer GNN



Representation of Node A with 2 layers



GCN

$$h_v^{(0)} = x_v \quad \text{for all } v \in V.$$

Node v's initial embedding.

... is just node v's original features.

and for $k = 1, 2, \ldots$ upto K:

$$h_v^{(k)} = f^{(k)}\left(W^{(k)} \cdot \frac{\sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}}{|\mathcal{N}(v)|} + B^{(k)} \cdot h_v^{(k-1)} \right) \quad \text{for all } v \in V.$$

Node v's embedding at step k.

Mean of v's neighbour's embeddings at step $k-1$.

Node v's embedding at step $k-1$.

Color Codes:

- Embedding of node v.
- Embedding of a neighbour of node v.
- (Potentially) Learnable parameters.

GAT

$$h_v^{(0)} = x_v \quad \text{for all } v \in V.$$

Node v's initial embedding.

... is just node v's original features.

and for $k = 1, 2, \ldots$ upto K:

$$h_v^{(k)} = f^{(k)}\left(W^{(k)} \cdot \left[\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(k-1)} h_u^{(k-1)} + \alpha_{vv}^{(k-1)} h_v^{(k-1)}\right]\right) \quad \text{for all } v \in V.$$

Node v's embedding at step k.

Weighted mean of v 's neighbour's embeddings at step $k-1$.

Node v's embedding at step $k-1$.

where the attention weights $\alpha^{(k)}$ are generated by an attention mechanism $A^{(k)}$, normalized such that the sum over all neighbours of each node v is 1:

$$\alpha_{vu}^{(k)} = \frac{A^{(k)}(h_v^{(k)}, h_u^{(k)})}{\sum_{w \in \mathcal{N}(v)} A^{(k)}(h_v^{(k)}, h_w^{(k)})} \quad \text{for all } (v, u) \in E.$$

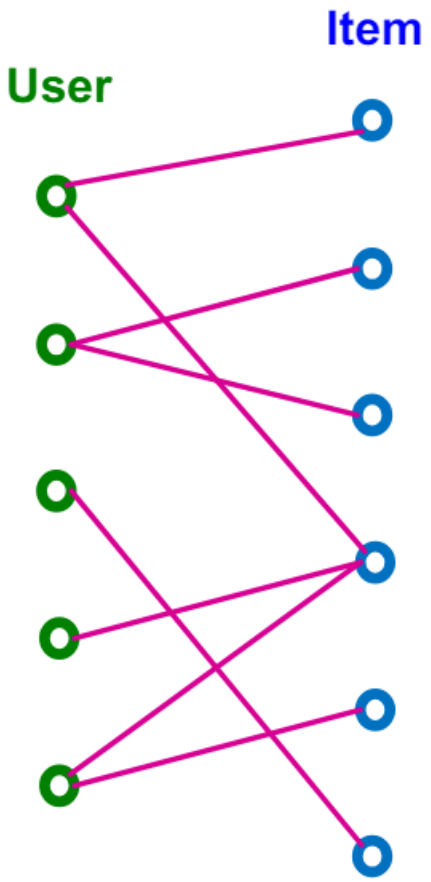
Color Codes:

- Embedding of node v.
- Embedding of a neighbour of node v.
- (Potentially) Learnable parameters.

https://distill.pub/2021/understanding-gnns/

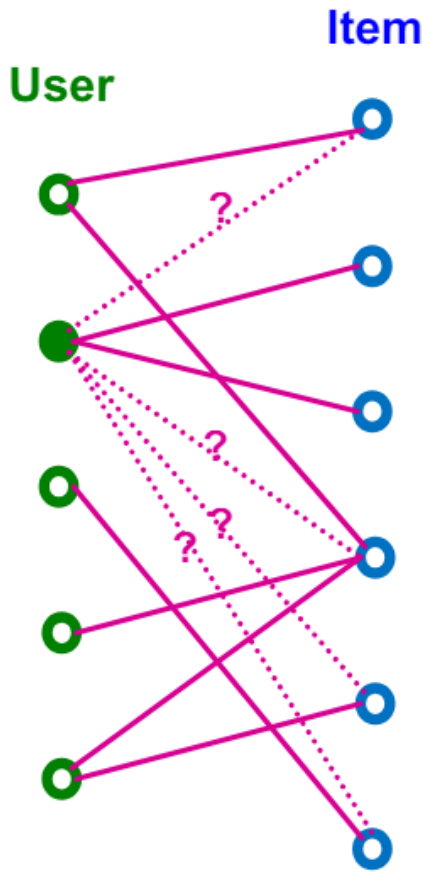
Recommender System as a Graph

- Recommender system can be naturally modeled as a **bipartite graph**
  - A graph with two node types: **users** and **items**.
  - **Edges** connect users and items
    - Indicates user-item interaction (e.g., click, purchase, review etc.)
    - Often associated with timestamp (timing of the interaction).



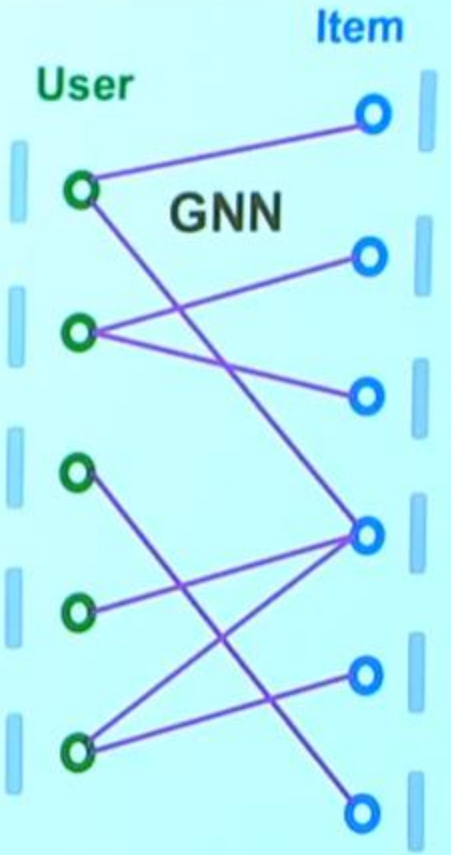
Recommendation Task

- **Given**
  - Past user-item interactions
- **Task**
  - Predict new items each user will interact in the future.
  - Can be cast as **link prediction** problem.
    - Predict new user-item interaction edges given the past edges.
  - For $u \in \boldsymbol{U}, v \in \boldsymbol{V}$, we need to get a real-valued **score** $f(u, v)$.



NGCF Framework

- **Given**: User-item bipartite graph.
- **NGCF framework:**
  - Prepare shallow learnable embedding for each node.
  - Use multi-layer GNNs to propagate embeddings along the bipartite graph.
    - High-order graph structure is captured.
  - Final embeddings are *explicitly* graph-aware!
- **Two kinds of learnable params are jointly learned:**
  - Shallow user/item embeddings
  - GNN's parameters



Shallow user/item embeddings (learnable)

NGCF (

Neighbor Aggregation

- Iteratively update node embeddings using neighboring embeddings.

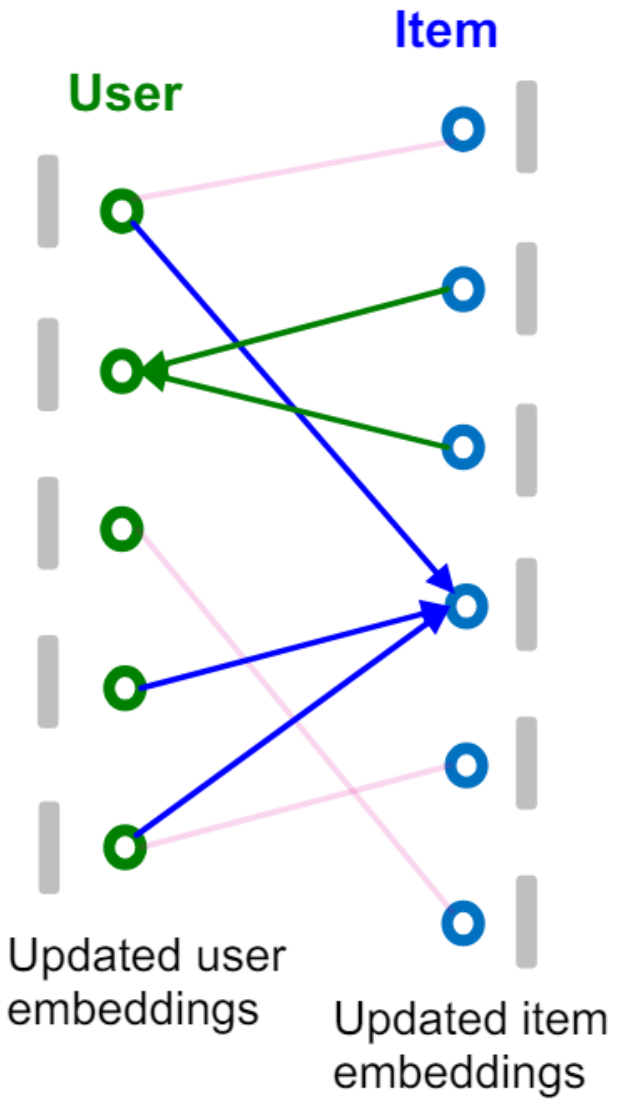
$$\boldsymbol{h}_v^{(k+1)} = \text{COMBINE}\left(\boldsymbol{h}_v^{(k)}, \text{AGGR}\left(\left\{\boldsymbol{h}_u^{(k)}\right\}_{u \in N(v)}\right)\right)$$

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

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 $\text{ReLU}(\text{Linear}(\text{Concat}(\boldsymbol{x}, \boldsymbol{y})))$



NGCF

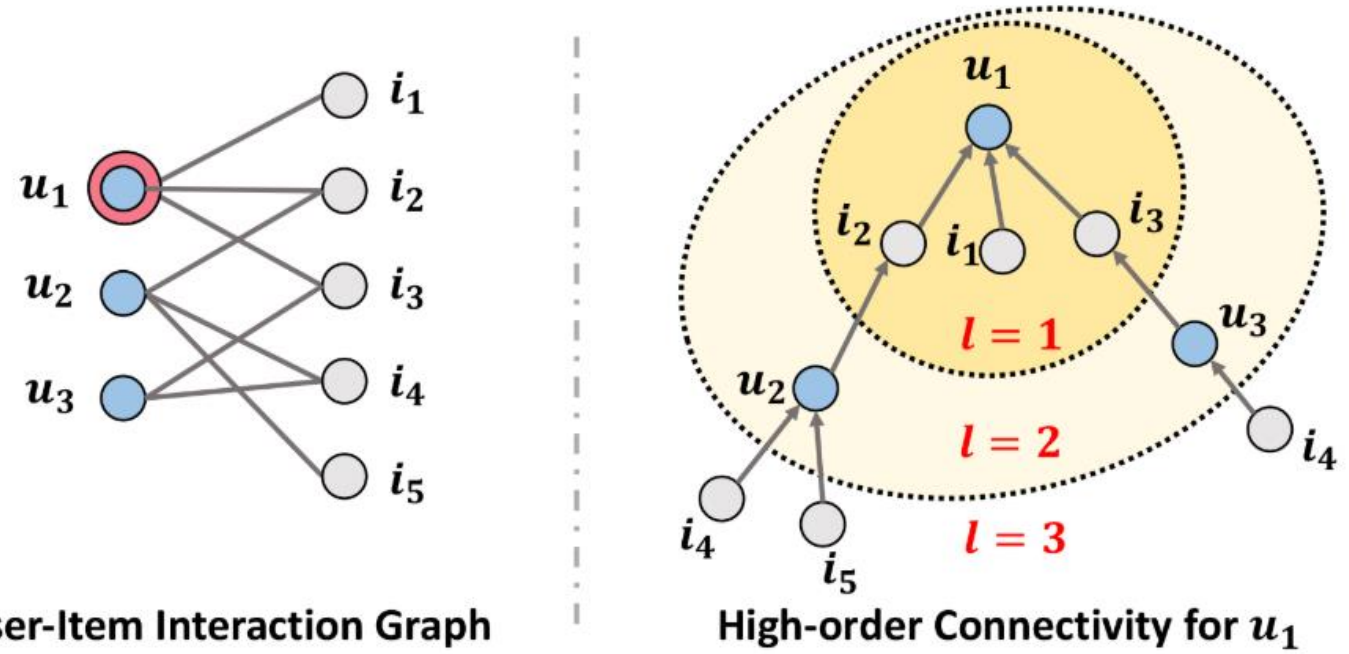


Figure 1: An illustration of the user-item interaction graph and the high-order connectivity. The node u_1 is the target user to provide recommendations for.

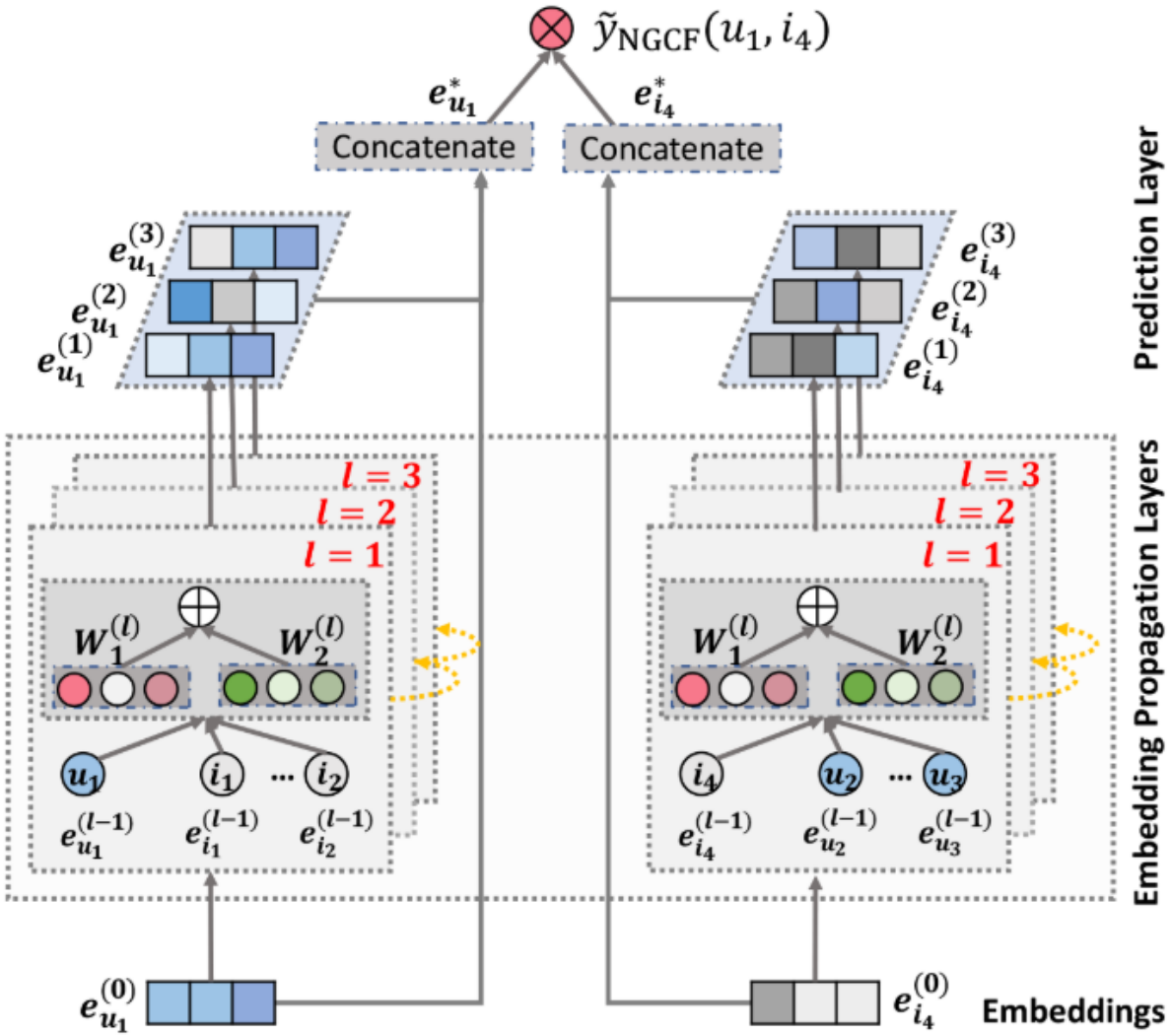


Figure 2: An illustration of NGCF model architecture (the arrowed lines present the flow of information). The representations of user u_1 (left) and item i_4 (right) are refined with multiple embedding propagation layers, whose outputs are concatenated to make the final prediction.

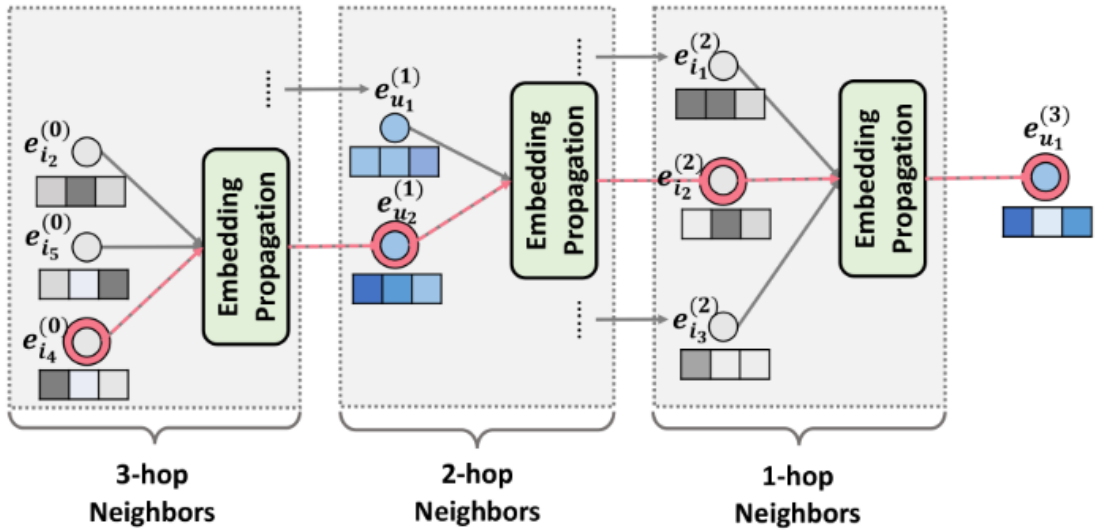


Figure 3: Illustration of third-order embedding propagatio for user u_1. Best view in color.

NGCF

$$\mathbf{e}_u^{(l)} = \text{LeakyReLU}\left(\mathbf{m}_{u\leftarrow u}^{(l)} + \sum_{i\in\mathcal{N}_u} \mathbf{m}_{u\leftarrow i}^{(l)}\right),$$

Following the graph convolutional network [18], we set p_{ui} as the graph Laplacian norm $1/\sqrt{|\mathcal{N}_u||\mathcal{N}_i|}$, where $\mathcal{N}_u$ and $\mathcal{N}_i$ denote the first-hop neighbors of user u and item i. From the viewpoint of representation learning, p_{ui} reflects how much the historical item contributes the user preference. From the viewpoint of message passing, p_{ui} can be interpreted as a discount factor, considering the messages being propagated should decay with the path length.

$$\begin{cases} \mathbf{m}_{u\leftarrow i}^{(l)} = p_{ui}\left(\mathbf{W}_1^{(l)}\mathbf{e}_i^{(l-1)} + \mathbf{W}_2^{(l)}(\mathbf{e}_i^{(l-1)} \odot \mathbf{e}_u^{(l-1)})\right), \\ \mathbf{m}_{u\leftarrow u}^{(l)} = \mathbf{W}_1^{(l)}\mathbf{e}_u^{(l-1)}, \end{cases}$$

Loss Function: BPR Loss (2)

- **Training objective**: **For each user** u^*, we want the scores of rooted positive edges $\boldsymbol{E}(u^*)$ to be higher than those of rooted negative edges $\boldsymbol{E}_{\mathbf{neg}}(u^*)$.
  - **Aligns with the personalized nature of the recall metric.**
- **BPR Loss for user $\boldsymbol{u}^*$:**

Encouraged to be positive for each user
=positive edge score is higher than negative edge score

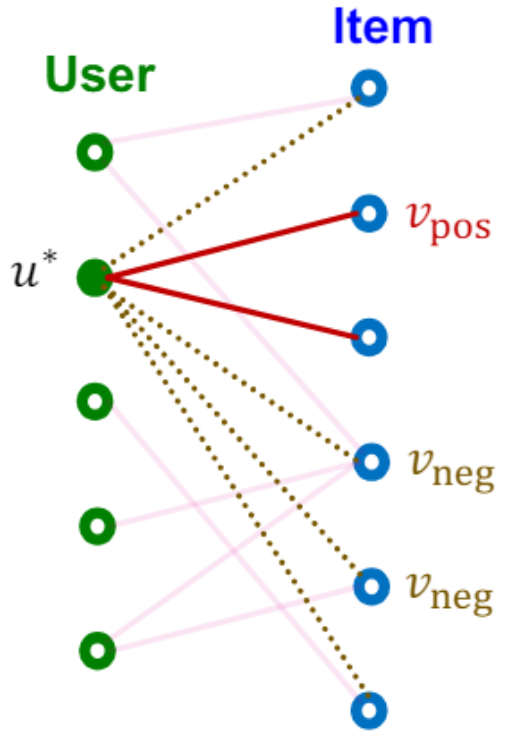
$$\text{Loss}(u^*) = \frac{1}{|E(u^*)| \cdot |\boldsymbol{E}_{\mathbf{neg}}(u^*)|} \underbrace{\sum_{(u^*, v_{\text{pos}}) \in \boldsymbol{E}(u^*)} \sum_{(u^*, v_{\text{neg}}) \in \boldsymbol{E}_{\mathbf{neg}}(u^*)} -\log\left(\sigma\left(f_\theta(\boldsymbol{u}^*, \boldsymbol{v}_{\text{pos}}) - f_\theta(\boldsymbol{u}^*, \boldsymbol{v}_{\text{neg}})\right)\right)}$$

Can be approximated using a mini-batch

- Final BPR Loss: $\frac{1}{|\boldsymbol{U}|} \sum_{u^* \in \boldsymbol{U}} \text{Loss}(u^*)$

Loss Function: BPR Loss (3)

- **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



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

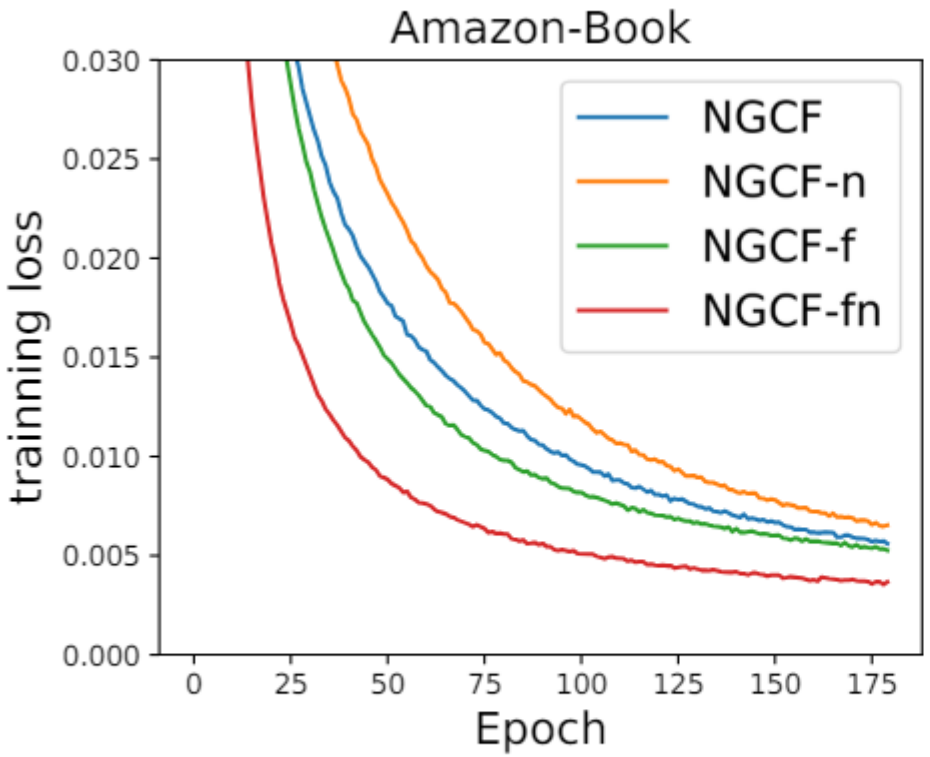
Average over users in the mini-batch

Table 2: Overall Performance Comparison.

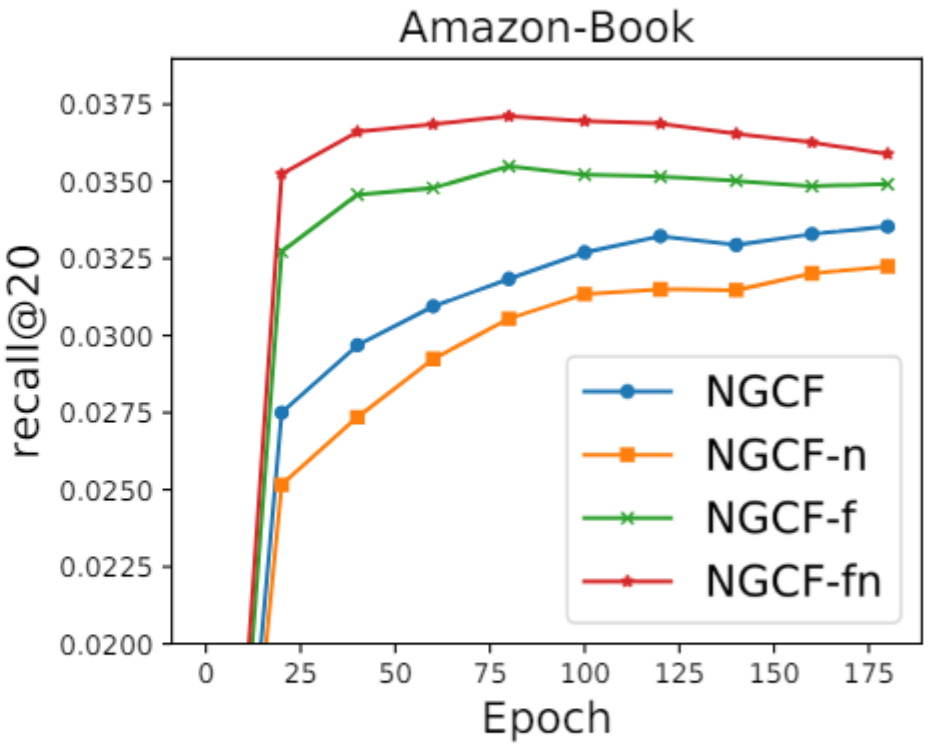
	Gowalla		Yelp2018*		Amazon-Book	
	recall	ndcg	recall	ndcg	recall	ndcg
MF	0.1291	0.1109	0.0433	0.0354	0.0250	0.0196
NeuMF	0.1399	0.1212	0.0451	0.0363	0.0258	0.0200
CMN	0.1405	0.1221	0.0457	0.0369	0.0267	0.0218
HOP-Rec	0.1399	0.1214	0.0517	0.0428	0.0309	0.0232
GC-MC	0.1395	0.1204	0.0462	0.0379	0.0288	0.0224
PinSage	0.1380	0.1196	0.0471	0.0393	0.0282	0.0219
NGCF-3	**0.1569***	**0.1327***	**0.0579***	**0.0477***	**0.0337***	**0.0261***
%Improv.	11.68%	8.64%	11.97%	11.29%	9.61%	12.50%
p-value	2.01e-7	3.03e-3	5.34e-3	4.62e-4	3.48e-5	1.26e-4

We implement three simplified variants of NGCF:

- NGCF-f, which removes the feature transformation matrices $\mathbf{W}_1$ and $\mathbf{W}_2$.
- NGCF-n, which removes the non-linear activation function σ.
- NGCF-fn, which removes both the feature transformation matrices and non-linear activation function.



(c) Training loss on Amazon-Book



(d) Testing recall on Amazon-Book

NGCF

- Graph Convolution Layer

$$\mathbf{e}_u^{(l)} = \text{LeakyReLU}\left(\mathbf{m}_{u\leftarrow u}^{(l)} + \sum_{i\in\mathcal{N}_u} \mathbf{m}_{u\leftarrow i}^{(l)}\right)$$

- Layer Combination

$$\mathbf{e}_u^* = \mathbf{e}_u^{(0)} \| \cdots \| \mathbf{e}_u^{(L)}$$

- Matrix Form

$$\mathbf{E}^{(l)} = \text{LeakyReLU}\left((\mathcal{L} + \mathbf{I})\mathbf{E}^{(l-1)}\mathbf{W}_1^{(l)} + \mathcal{L}\mathbf{E}^{(l-1)} \odot \mathbf{E}^{(l-1)}\mathbf{W}_2^{(l)}\right)$$

LightGCN

- **Light** Graph Convolution Layer

$$\mathbf{e}_u^{(k+1)} = \sum_{i\in\mathcal{N}_u} \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}} \mathbf{e}_i^{(k)}$$

- Layer Combination

$$\mathbf{e}_u = \sum_{k=0}^{K} \alpha_k \mathbf{e}_u^{(k)}$$

- Matrix Form

$$\mathbf{E}^{(k+1)} = (\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})\mathbf{E}^{(k)}$$

Only simple weighted sum aggregator is remained

- No feature transformation
- No nonlinear activation
- No self connection

He et al. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. SIGIR'2020

LightGCN (2020)

$$\mathbf{e}_u^{(k+1)} = \sum_{i \in \mathcal{N}_u} \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}} \mathbf{e}_i^{(k)},$$

$$\mathbf{e}_i^{(k+1)} = \sum_{u \in \mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_i|}\sqrt{|\mathcal{N}_u|}} \mathbf{e}_u^{(k)}.$$

$$\mathbf{e}_u = \sum_{k=0}^{K} \alpha_k \mathbf{e}_u^{(k)}; \quad \mathbf{e}_i = \sum_{k=0}^{K} \alpha_k \mathbf{e}_i^{(k)}.$$

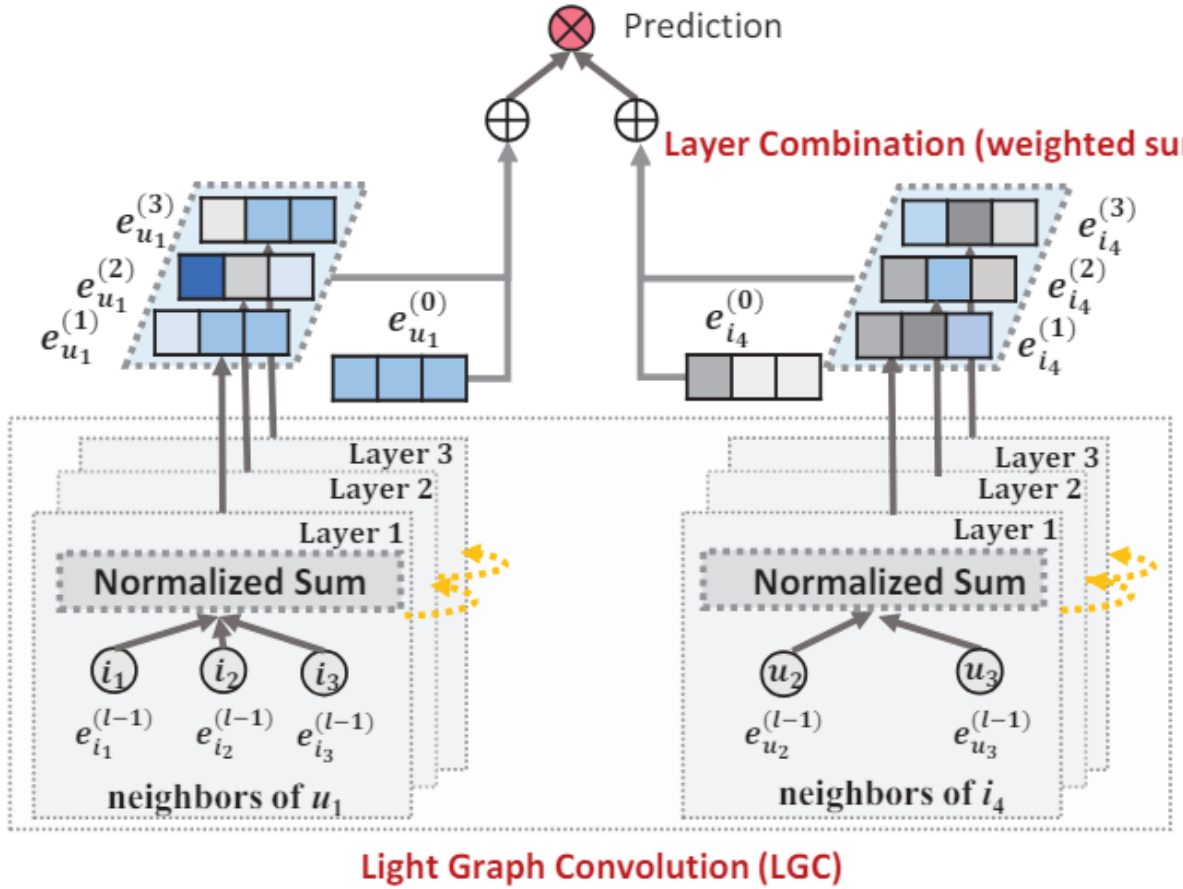


Figure 2: An illustration of LightGCN model architecture. In LGC, only the normalized sum of neighbor embeddings is performed towards next layer; other operations like self-connection, feature transformation, and nonlinear activation are all removed, which largely simplifies GCNs. In Layer Combination, we sum over the embeddings at each layer to obtain the final representations.

LightGCN (2020)

Table 3: Performance comparison between NGCF and LightGCN at different layers.

Dataset		Gowalla		Yelp2018		Amazon-Book	
Layer #	**Method**	**recall**	**ndcg**	**recall**	**ndcg**	**recall**	**ndcg**
1 Layer	NGCF	0.1556	0.1315	0.0543	0.0442	0.0313	0.0241
	LightGCN	0.1755(+12.79%)	0.1492(+13.46%)	0.0631(+16.20%)	0.0515(+16.51%)	0.0384(+22.68%)	0.0298(+23.65%)
2 Layers	NGCF	0.1547	0.1307	0.0566	0.0465	0.0330	0.0254
	LightGCN	0.1777(+14.84%)	0.1524(+16.60%)	0.0622(+9.89%)	0.0504(+8.38%)	0.0411(+24.54%)	0.0315(+24.02%)
3 Layers	NGCF	0.1569	0.1327	0.0579	0.0477	0.0337	0.0261
	LightGCN	0.1823(+16.19%)	0.1555(+17.18%)	0.0639(+10.38%)	0.0525(+10.06%)	0.0410(+21.66%)	0.0318(+21.84%)
4 Layers	NGCF	0.1570	0.1327	0.0566	0.0461	0.0344	0.0263
	LightGCN	0.1830(+16.56%)	0.1550(+16.80%)	0.0649(+14.58%)	0.0530(+15.02%)	0.0406(+17.92%)	0.0313(+18.92%)

*The scores of NGCF on Gowalla and Amazon-Book are directly copied from Table 3 of the NGCF paper (https://arxiv.org/abs/1905.08108)

GraphSAGE (2017)

SAGE = SAmple & Aggregate

Unlike embedding approaches that are based on matrix factorization,
we leverage node features (e.g., text attributes, node profile information, node degrees) in order to
learn an embedding function that generalizes to unseen nodes.

Inductive vs transductive

Uber Eats (2019)

Unsupervised learning

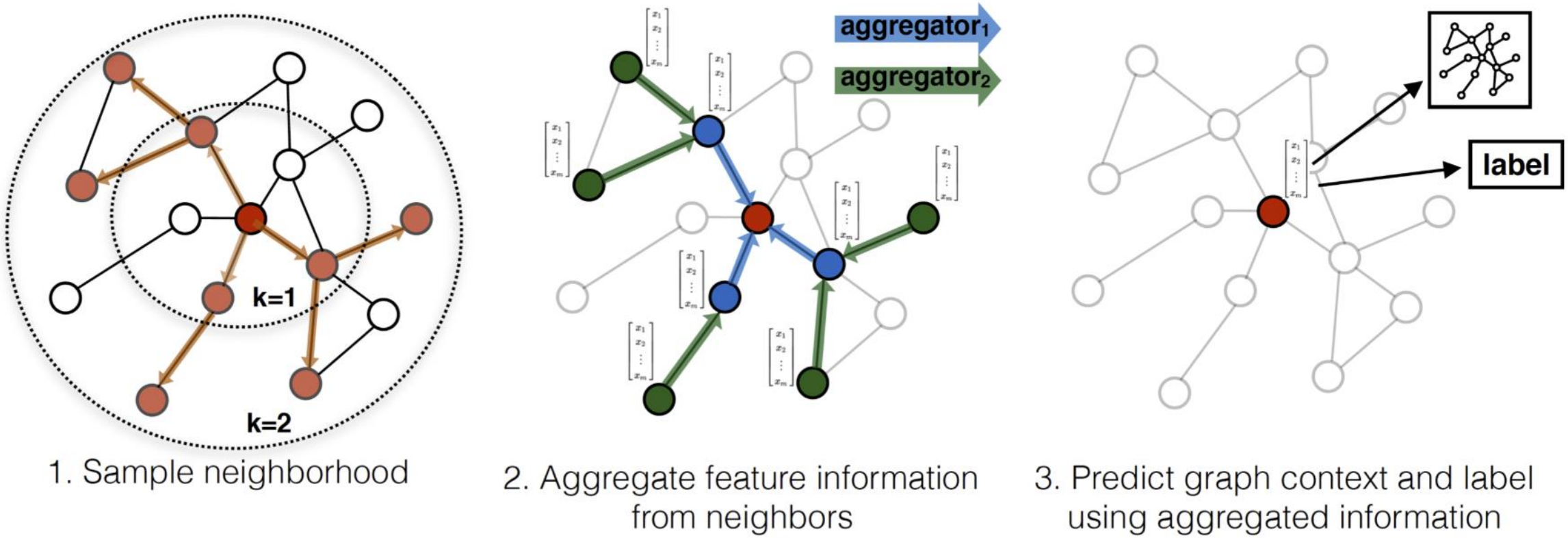


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

In this work, we uniformly sample a fixed-size set of neighbors, instead of using full neighborhood sets in Algorithm 1, in order to keep the computational footprint of each batch fixed

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;

for $k = 1...K$ **do**

  for $v \in \mathcal{V}$ **do**

    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;

    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$

  end

  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$

end

$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%

- (i) classifying academic papers into different subjects using the Web of Science citation dataset, (ii) classifying Reddit posts as belonging to different communities, and (iii) classifying protein functions across various biological protein-protein interaction (PPI) graph

We sampled 50 large communities and built a post-to-post graph, connecting posts if the same user comments on both.

Unsupervised

3.2 Learning the parameters of GraphSAGE

In order to learn useful, predictive representations in a fully unsupervised setting, we apply a graph-based loss function to the output representations, $\mathbf{z}_u, \forall u \in \mathcal{V}$, and tune the weight matrices, $\mathbf{W}^k, \forall k \in \{1, ..., K\}$, and parameters of the aggregator functions via stochastic gradient descent. The graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right), \quad (1)$$

where v is a node that co-occurs near u on fixed-length random walk, σ is the sigmoid function,

PinSAGE (2018)

- 3 billion nodes
- 18 billion edges

- **On-the-fly convolutions:** Traditional GCN algorithms perform graph convolutions by multiplying feature matrices by powers of the full graph Laplacian. In contrast, our PinSage algorithm performs efficient, localized convolutions by sampling the neighborhood around a node and dynamically constructing a computation graph from this sampled neighborhood. These dynamically constructed computation graphs (Fig. 1) specify how to perform a localized convolution around a particular node, and alleviate the need to operate on the entire graph during training.
- **Producer-consumer minibatch construction:** We develop a producer-consumer architecture for constructing minibatches that ensures maximal GPU utilization during model training. A large-memory, CPU-bound producer efficiently samples node network neighborhoods and fetches the necessary features to define local convolutions, while a GPU-bound TensorFlow model consumes these pre-defined computation graphs to efficiently run stochastic gradient decent.
- **Efficient MapReduce inference:** Given a fully-trained GCN model, we design an efficient MapReduce pipeline that can distribute the trained model to generate embeddings for billions of nodes, while minimizing repeated computations.

- **Constructing convolutions via random walks:** Taking full neighborhoods of nodes to perform convolutions (Fig. 1) would result in huge computation graphs, so we resort to sampling. However, random sampling is suboptimal, and we develop a new technique using short random walks to sample the computation graph. An additional benefit is that each node now has an importance score, which we use in the pooling/aggregation step.
- **Importance pooling:** A core component of graph convolutions is the aggregation of feature information from local neighborhoods in the graph. We introduce a method to weigh the importance of node features in this aggregation based upon random-walk similarity measures, leading to a 46% performance gain in offline evaluation metrics.
- **Curriculum training:** We design a curriculum training scheme, where the algorithm is fed harder-and-harder examples during training, resulting in a 12% performance gain.

3.1 Problem Setup

Pinterest is a content discovery application where users interact with *pins*, which are visual bookmarks to online content (*e.g.*, recipes they want to cook, or clothes they want to purchase). Users organize these pins into *boards*, which contain collections of pins that the user deems to be thematically related. Altogether, the Pinterest graph contains 2 billion pins, 1 billion boards, and over 18 billion edges (*i.e.*, memberships of pins to their corresponding boards).

Our task is to generate high-quality embeddings or representations of pins that can be used for recommendation (*e.g.*, via nearest-neighbor lookup for related pin recommendation, or for use in a downstream re-ranking system). In order to learn these embeddings, we model the Pinterest environment as a bipartite graph consisting of nodes in two disjoint sets, $\mathcal{I}$ (containing pins) and C (containing boards). Note, however, that our approach is also naturally generalizable, with $\mathcal{I}$ being viewed as a set of items and C as a set of user-defined contexts or collections.

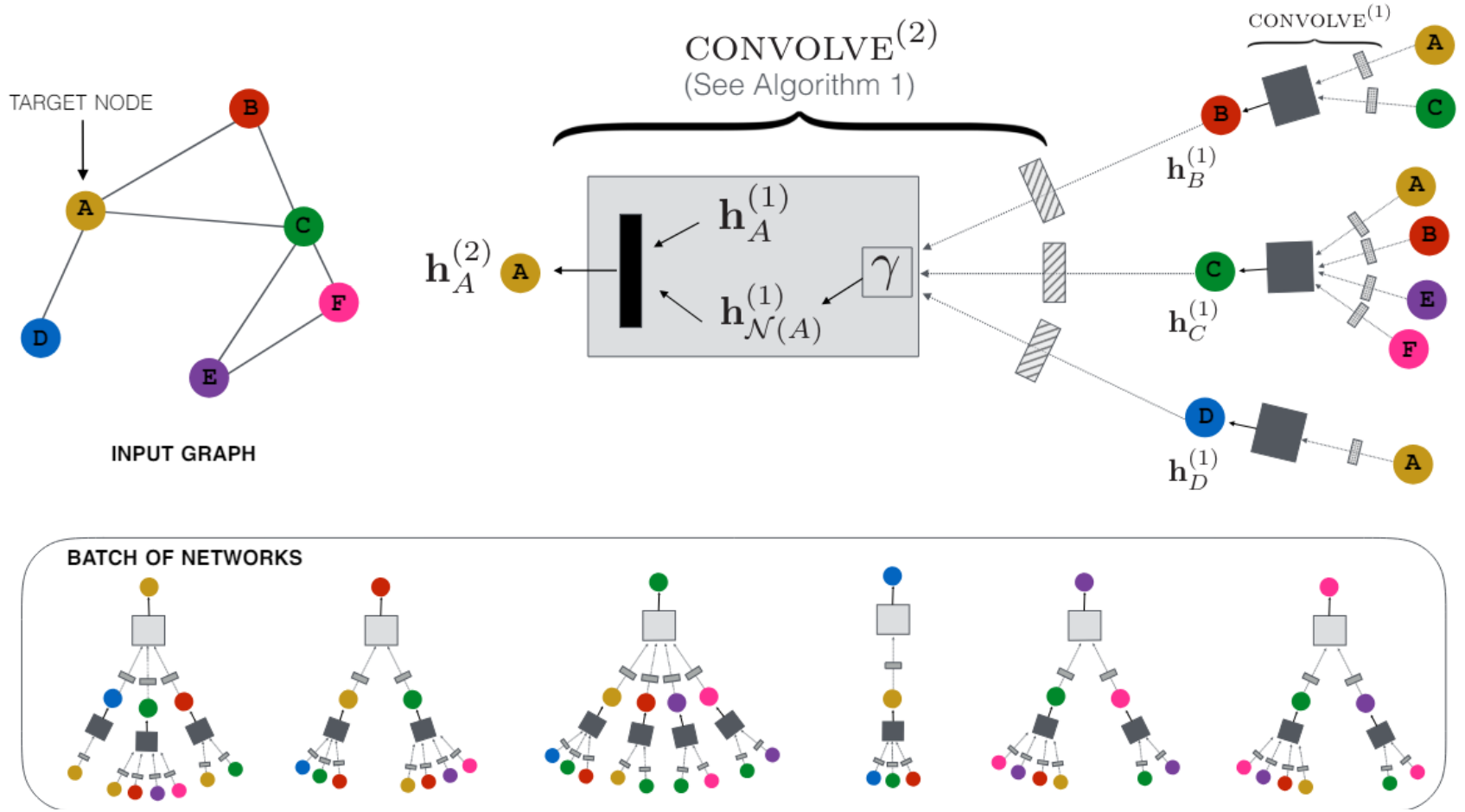


Figure 1: Overview of our model architecture using depth-2 convolutions (best viewed in color). Left: A small example input graph. Right: The 2-layer neural network that computes the embedding $\mathbf{h}_A^{(2)}$ of node A using the previous-layer representation, $\mathbf{h}_A^{(1)}$, of node A and that of its neighborhood $\mathcal{N}(A)$ (nodes B, C, D). (However, the notion of neighborhood is general and not all neighbors need to be included (Section 3.2).) Bottom: The neural networks that compute embeddings of each node of the input graph. While neural networks differ from node to node they all share the same set of parameters (*i.e.*, the parameters of the CONVOLVE$^{(1)}$ and CONVOLVE$^{(2)}$ functions; Algorithm 1). Boxes with the same shading patterns share parameters; γ denotes an importance pooling function; and thin rectangular boxes denote densely-connected multi-layer neural networks.

Sampling

Concretely, we simulate random walks starting from node u and compute the L1-normalized visit count of nodes visited by the random walk. The neighborhood of u is then defined as the top T nodes with the highest normalized visit counts with respect to node u.

The advantages of this importance-based neighborhood definition are two-fold. First, selecting a fixed number of nodes to aggregate from allows us to control the memory footprint of the algorithm during training [18]. Second, it allows Algorithm 1 to take into account the importance of neighbors when aggregating the vector representations of neighbors. In particular, we implement γ in Algorithm 1 as a weighted-mean, with weights defined according to the L1 normalized visit counts. We refer to this new approach as importance pooling

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Algorithm 1: CONVOLVE

Input : Current embedding $\mathbf{z}_u$ for node u; set of neighbor embeddings $\{\mathbf{z}_v | v \in \mathcal{N}(u)\}$, set of neighbor weights $\boldsymbol{\alpha}$; symmetric vector function $\gamma(\cdot)$

Output : New embedding $\mathbf{z}_u^{\text{NEW}}$ for node u

$\mathbf{n}_u \leftarrow \gamma\left(\{\text{ReLU}\left(\mathbf{Q}\mathbf{h}_v + \mathbf{q}\right) \mid v \in \mathcal{N}(u)\}, \boldsymbol{\alpha}\right)$;
$\mathbf{z}_u^{\text{NEW}} \leftarrow \text{ReLU}\left(\mathbf{W} \cdot \text{CONCAT}(\mathbf{z}_u, \mathbf{n}_u) + \mathbf{w}\right)$;
$\mathbf{z}_u^{\text{NEW}} \leftarrow \mathbf{z}_u^{\text{NEW}} / \|\mathbf{z}_u^{\text{NEW}}\|_2$

In particular, we implement γ in Algorithm 1 as a weighted-mean, with weights defined according to the L1 normalized visit counts. We refer to this new approach as importance pooling.

3.3 Model Training

We train PinSage in a supervised fashion using a max-margin ranking loss. In this setup, we assume that we have access to a set of labeled pairs of items $\mathcal{L}$, where the pairs in the set, $(q, i) \in \mathcal{L}$, are assumed to be related—*i.e.*, we assume that if $(q, i) \in \mathcal{L}$ then item i is a good recommendation candidate for query item q. The goal of the training phase is to optimize the PinSage parameters so that the output embeddings of pairs $(q, i) \in \mathcal{L}$ in the labeled set are close together.

Loss function. In order to train the parameters of the model, we use a max-margin-based loss function. The basic idea is that we want to maximize the inner product of positive examples, *i.e.*, the embedding of the query item and the corresponding related item. At the same time we want to ensure that the inner product of negative examples—*i.e.*, the inner product between the embedding of the query item and an unrelated item—is smaller than that of the positive sample by some pre-defined margin. The loss function for a single pair of node embeddings $(\mathbf{z}_q, \mathbf{z}_i) : (q, i) \in \mathcal{L}$ is thus

$$J_{\mathcal{G}}(\mathbf{z}_q \mathbf{z}_i) = \mathbb{E}_{n_k \sim P_n(q)} \max\{0, \mathbf{z}_q \cdot \mathbf{z}_{n_k} - \mathbf{z}_q \cdot \mathbf{z}_i + \Delta\}, \qquad (1)$$

where $P_n(q)$ denotes the distribution of negative examples for item q, and Δ denotes the margin hyper-parameter. We shall explain the sampling of negative samples below.

Negative Sampling

Sampling negative items. Negative sampling is used in our loss function (Equation 1) as an approximation of the normalization factor of edge likelihood [23]. To improve efficiency when training with large batch sizes, we sample a set of 500 negative items to be shared by all training examples in each minibatch. This drastically saves the number of embeddings that need to be computed during each training step, compared to running negative sampling for each node independently. Empirically, we do not observe a difference between the performance of the two sampling schemes.

as the positive item i. We call these "hard negative items". They are generated by ranking items in a graph according to their Personalized PageRank scores with respect to query item q [14]. Items ranked at 2000-5000 are randomly sampled as hard negative items. As illustrated in Figure 2, the hard negative examples are more similar to the query than random negative examples, and are thus challenging for the model to rank, forcing the model to learn to distinguish items at a finer granularity.

Algorithm 2 Pixie Random Walk algorithm with early stopping.

PixieRandomWalk(q: Query pin, E: Set of edges, U: User personalization features, α: Real, N: Int, n_p: Int, n_v: Int)

```
1:  totSteps = 0, V = 0⃗
2:  nHighVisited = 0
3:  repeat
4:      currPin = q
5:      currSteps = SampleWalkLength(α)
6:      for i = [1 : currSteps] do
7:          currBoard = E(currPin)[PersonalizedNeighbor(E,U)]
8:          currPin = E(currBoard)[PersonalizedNeighbor(E,U)]
9:          V[currPin]++
10:         if V[currPin] == n_v then
11:             nHighVisited++
12:     totSteps += currSteps
13: until totSteps ≥ N or nHighVisited > n_p
14: return V
```

Curriculum training

As illustrated in Figure 2, the hard negative examples are more similar to the query than random negative examples, and are thus challenging for the model to rank, forcing the model to learn to distinguish items at a finer granularity.

Using hard negative items throughout the training procedure doubles the number of epochs needed for the training to converge. To help with convergence, we develop a curriculum training scheme [4]. In the first epoch of training, no hard negative items are used, so that the algorithm quickly finds an area in the parameter space where the loss is relatively small. We then add hard negative items in subsequent epochs, focusing the model to learn how to distinguish highly related pins from only slightly related ones. At epoch n of the training, we add $n - 1$ hard negative items to the set of negative items for each item.

Query

Positive Example

Random Negative

Hard Negative

Features used for learning. Each pin at Pinterest is associated with an image and a set of textual annotations (title, description). To generate feature representation $\mathbf{x}_q$ for each pin q, we concatenate visual embeddings (4,096 dimensions), textual annotation embeddings (256 dimensions), and the log degree of the node/pin in the graph. The visual embeddings are the 6-th fully connected layer of a classification network using the VGG-16 architecture [28]. Textual annotation embeddings are trained using a Word2Vec-based model [23], where the context of an annotation consists of other annotations that are associated with each pin.

We also conduct ablation studies and consider several variants of PinSage when evaluating performance:

- **max-pooling** uses the element-wise max as a symmetric aggregation function (i.e., $\gamma = \max$) without hard negative samples;
- **mean-pooling** uses the element-wise mean as a symmetric aggregation function (i.e., $\gamma = \text{mean}$);
- **mean-pooling-xent** is the same as mean-pooling but uses the cross-entropy loss introduced in [18].
- **mean-pooling-hard** is the same as mean-pooling, except that it incorporates hard negative samples as detailed in Section 3.3.
- **PinSage** uses all optimizations presented in this paper, including the use of importance pooling in the convolution step.

Method	Hit-rate	MRR
Visual	17%	0.23
Annotation	14%	0.19
Combined	27%	0.37
max-pooling	39%	0.37
mean-pooling	41%	0.51
mean-pooling-xent	29%	0.35
mean-pooling-hard	46%	0.56
PinSage	**67%**	**0.59**

Table 1: Hit-rate and MRR for PinSage and content-based deep learning baselines. Overall, PinSage gives 150% improvement in hit rate and 60% improvement in MRR over the best baseline.[5]

https://web.stanford.edu/class/cs224w/