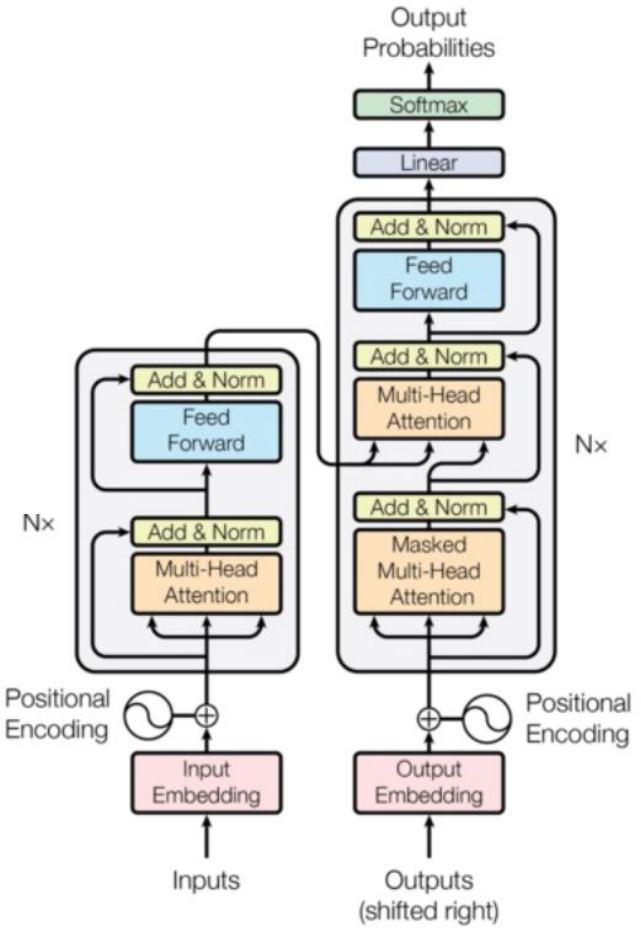
Graph Transformer

Most of the slides in this talk are taken from Stanford CS224W course: http://cs244w.stanford.edu

Caltech EAS AI BOOTCAMP

Transformers

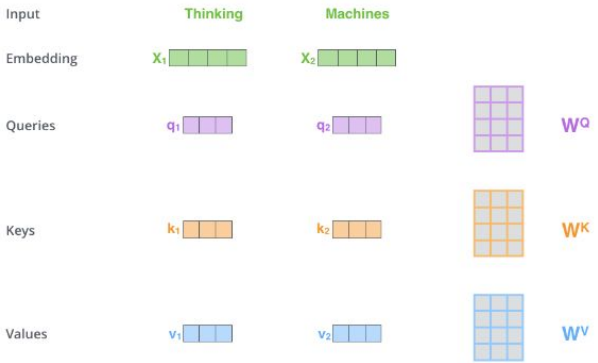
- **Lots of components**
  - **Normalization**
  - **Feed forward networks**
  - **Positional encoding (more later)**
  - **Multi-head self-attention**



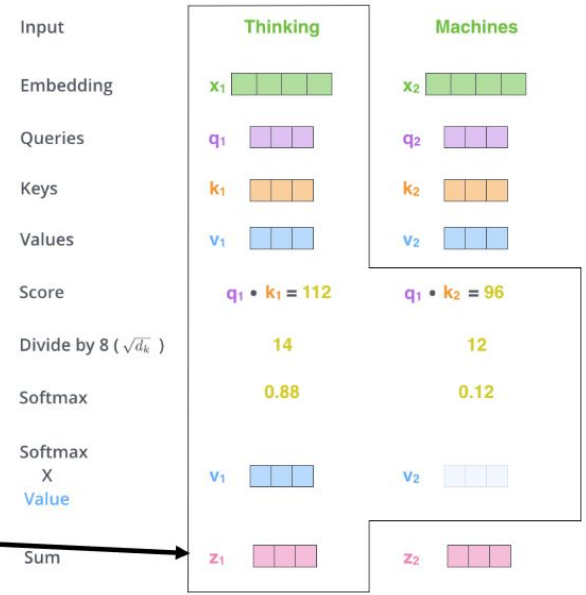
Self-attention

- **Step 1:** compute "key, value, query" for each input
- **Step 2 (just for x_1):** compute scores between pairs, turn into probabilities (same for x_2)
- **Step 3:** get new embedding z_1 by weighted sum of v_1, v_2

Step 1



Step 2

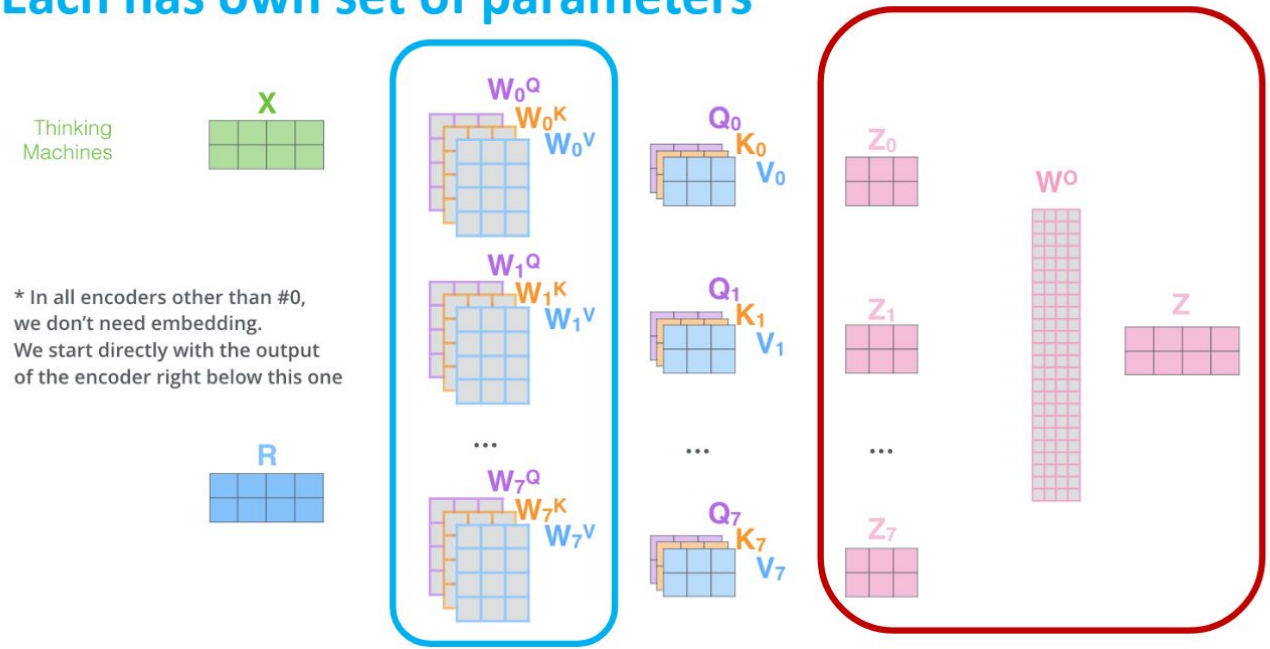


Step 3

$$z_1 = 0.88v_1 + 0.12v_2$$

Multi-head self-attention

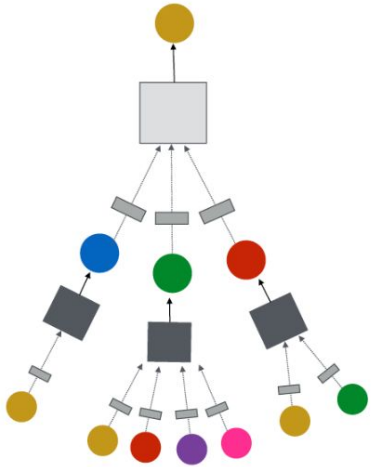
- **Do many self-attentions in parallel, and combine**
- **Different heads can learn different "similarities" between inputs**
- **Each has own set of parameters**

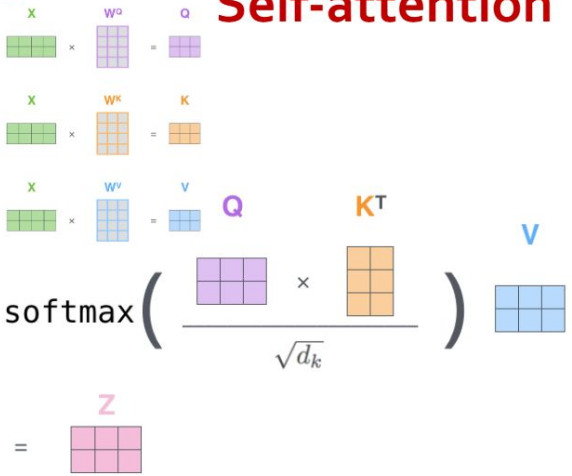


Comparing Transformers and GNN

- **Similarity:** GNNs also take in a sequence of vectors (in no particular order) and output a sequence of embeddings
- **Difference:** GNNs use **message passing**, Transformer uses **self-attention**
  - **Are self-attention and message passing really different?**

Message Passing **Vs.** **Self-attention**





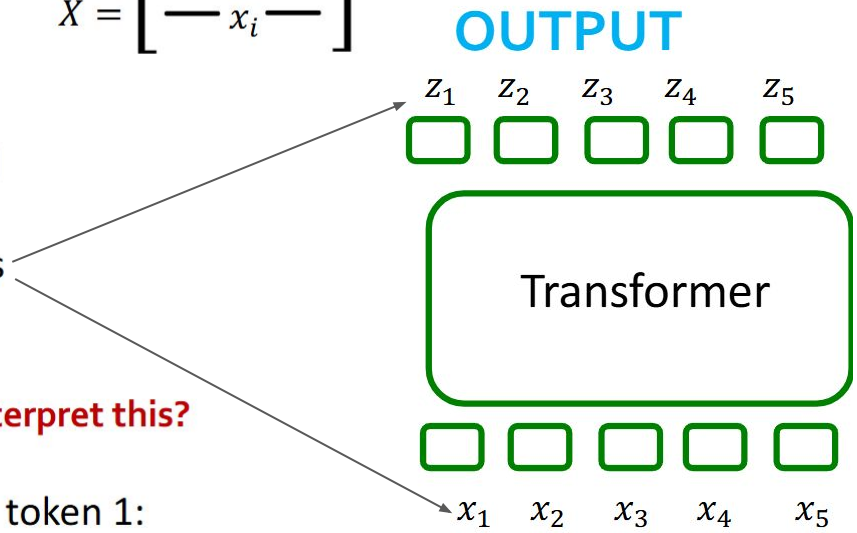
Interpreting Self-Attention Update

$$Att(X) = softmax(QK^T)V$$

$$Q = XW^Q, K = XW^K, \mathrm{V} = XW^V$$

Inputs stored row-wise

$$X = \begin{bmatrix} — x_i — \end{bmatrix}$$



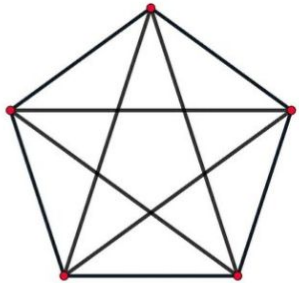
- This formula gives the embedding for **all tokens** simultaneously
- If we simplify to just token x_1 what does the update look like?

$$\mathrm{z}_1 = \sum_{j=1}^{5} softmax_j(q_1^T k_j) v_j$$ **How to interpret this?**

- Steps for computing new embedding for token 1:
  - **1. Compute message from j:** $(v_j,\ k_j) = MSG(x_j) = (W^V x_j, W^K x_j)$
  - **2. Compute query for 1:** $q_1 = MSG(x_1) = W^Q x_1$
  - **3. Aggregate all messages:** $Agg(q_1, \{MSG(x_j): j\}) = \sum_{j=1}^{n} softmax_j(q_1^T k_j) v_j$

Self-Attention as Message Passing

- Takeaway: **Self-attention can be written as message + aggregation – i.e., it is a GNN!**
- But so far there is no graph – just tokens.
  - **So what graph is this a GNN on?**
- Clearly tokens = nodes, but what are the edges?
- **Key observation:**
  - Token 1 depends on (receives "messages" from) all other tokens
  - ➔ **the graph is fully connected!**
- **Alternatively: if you only sum over $j \in N(i)$ you get ~GAT**



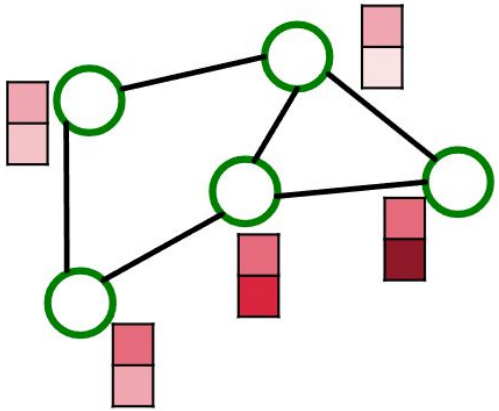
$$z_1 = \sum_{j=1}^{5} softmax_j(q_1^T k_j) v_j$$

- Steps for computing new embedding for token 1:
  - **1. Compute message from j:** $(v_j,\ k_j) = MSG(x_j) = (W^V x_j, W^K x_j)$
  - **2. Compute query for 1:** $q_1 = MSG(x_1) = W^Q x_1$
  - **3. Aggregate all messages:** $Agg(q_1, \{MSG(x_j): j\}) = \sum_{j=1}^{n} softmax_j(q_1^T k_j) v_j$

Processing Graphs with Transformers

- A graph Transformer must take the following inputs:
  - (1) Node features?
  - (2) Adjacency information?
  - (3) Edge features?
- Key components of Transformer
  - (1) tokenizing
  - (2) positional encoding
  - (3) self-attention

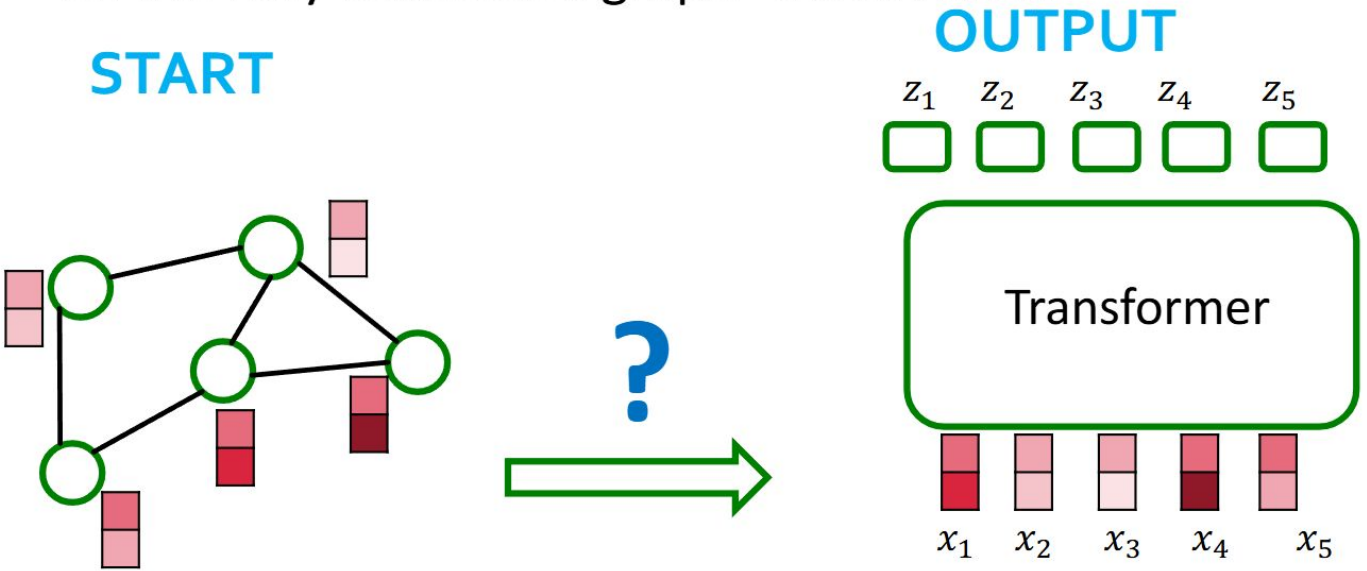
Today



- There are many ways to do this
- Different approaches correspond to different "matchings" between graph inputs **(1), (2), (3)** transformer components **(1), (2), (3)**

Nodes as Tokens

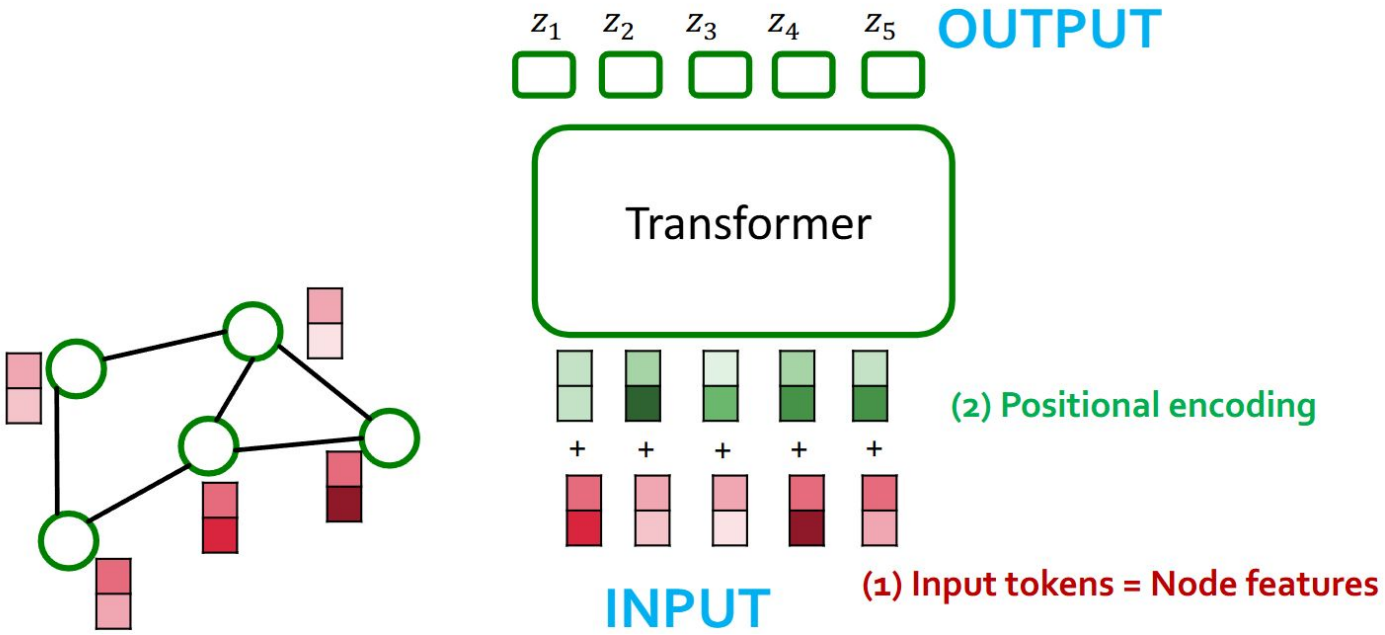
- **Q1: what should our tokens be?**
- **Sensible Idea:** node features = input tokens
- This matches the setting for the "attention is message passing on the fully connected graph" observation



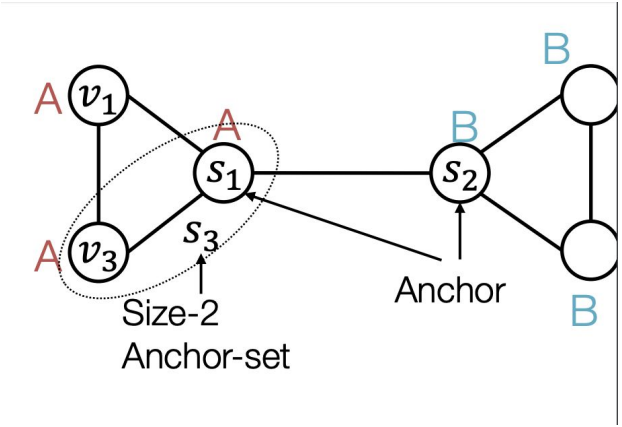
(1) Input tokens = Node features

Adjacency Information

- **Idea:** Encode adjacency info in the **positional encoding** for each node
- Positional encoding describes **where** a node is in the graph



Using Relative Distances



Positional encoding for node v_1 =

Relative Distances

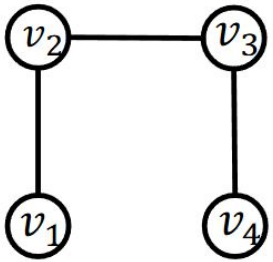
	s_1	s_2	s_3
v_1	1	2	1
v_3	1	2	0

Anchor s_1, s_2 cannot differentiate node v_1, v_3, but anchor-set s_3 can

Problem: Selecting the right anchors to make the embedding "structure aware"

Laplacian Eigenvector Positional Encodings

- **Key object:** Laplacian Matrix **L = Degrees - Adjacency**
  - Each graph has its own Laplacian matrix
  - Laplacian encodes the graph structure
  - Several Laplacian variants that add degree information differently

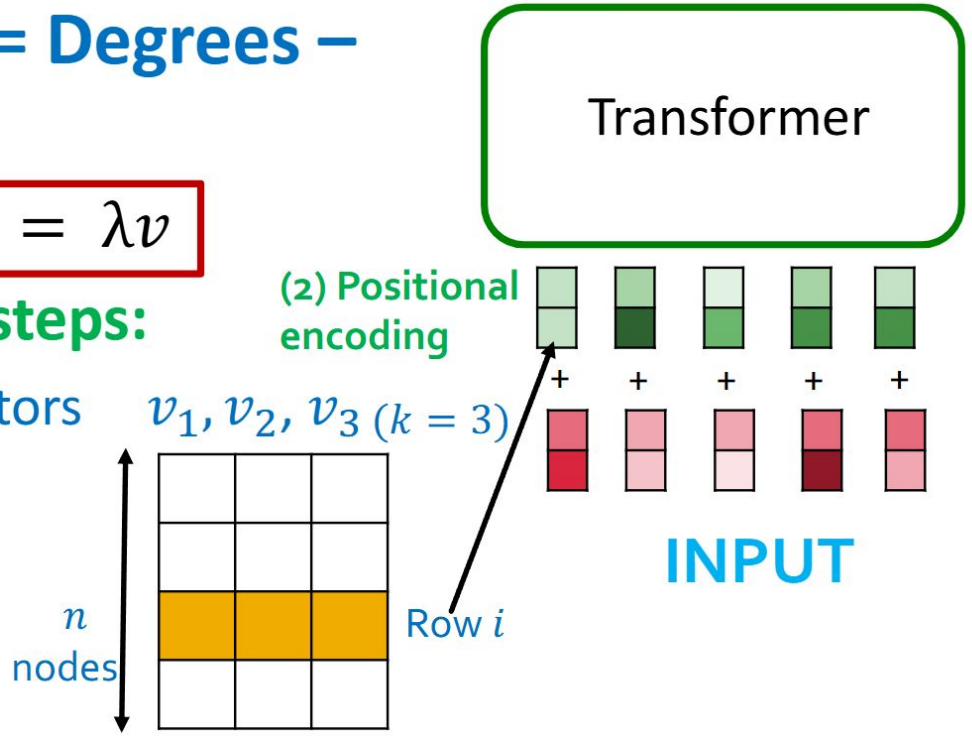


$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Degree of each node — Adjacency

Laplacian Eigenvector Positional Encodings

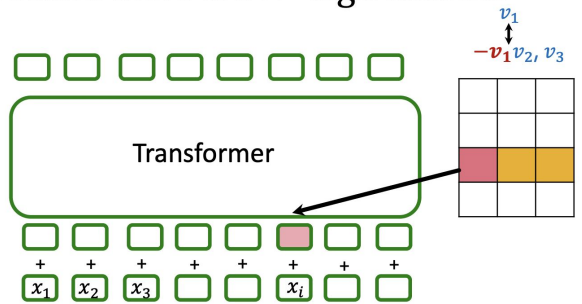
- Laplacian Matrix **L = Degrees – Adjacency**
- Eigenvector: v such that $Lv = \lambda v$
- **Positional encoding steps:**
  - 1. compute k eigenvectors
  - 2. Stack into matrix:
  - 3. ith row is positional encoding for node i



Choosing Laplacian Eigenvectors for Positional Encodings

- Both v and $-v$ are eigenvectors
- But when we use them as positional encodings we **pick one arbitrarily**
- **Why does this matter for positional encodings?**

- What if we picked the other sign choice?
- **Then the input PE changes**
- **=> The models predictions will change!**
- For k eigenvectors there are 2^k sign choices
  - 2^k different predictions for the same input graph!



- **Goal:** design a neural network $f(v_1, v_2, \ldots$ such that:
  - $f(v_1, v_2, \ldots v_k) = f(\pm v_1, \pm v_2, \ldots \pm v_k)$ for all $\pm$ choices
  - f is "expressive": note that $f(v_1, v_2, \ldots v_k) = 0$ is sign invariant… but it's a terrible neural network architecture

$$f(v_1, v_2, \ldots v_k) = \rho(\phi(v_1), +\phi(-v_1), \ldots, \phi(v_k), +\phi(-v_k))$$

ρ, ϕ = any neural network (MLP, GNN etc.)

SignNet

Edge Features in Self-Attention

- Not clear how to add edge features in the tokens or positional encoding
- How about in the attention? $Att(X) = softmax(QK^T)V$
- $[a_{ij}] = QK^T$ is an n x n matrix. Entry a_{ij} describes "how much" token j contributes to the update of token i

- **Idea: adjust a_{ij} based on edge features. Replace with $a_{ij} + c_{ij}$ where c_{ij} depends on the edge features**

- **Implementation:**
  - **If there is an edge between i and j with features e_{ij}, define $c_{ij} = w_1^T e_{ij}$**
  - **If there is no edge, find shortest edge path between i and j $(e^1, e^2, \dots e^N)$ and define $c_{ij} = \sum_n w_n^T e^n$**

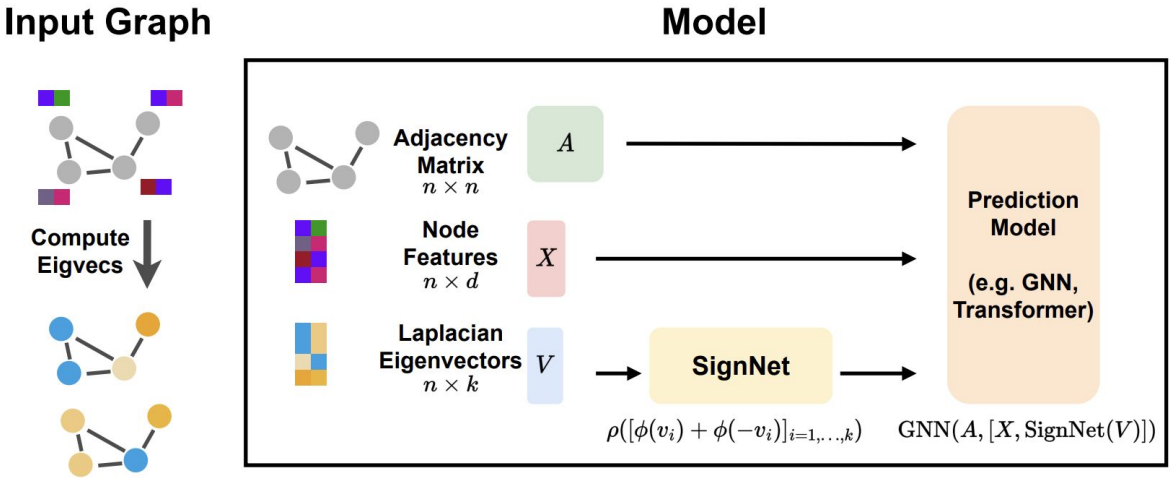
Learned parameters w_1

Learned parameters $w_1, \dots, w_N$

[Do Transformers Really Perform Bad for Graph Representation?](#) Ying et al. NeurIPS 2021

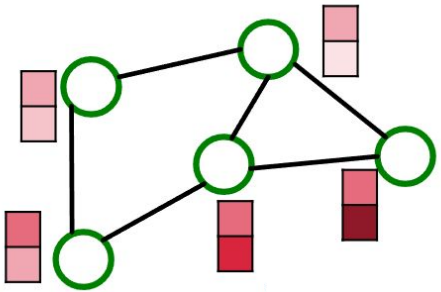
Putting it all together

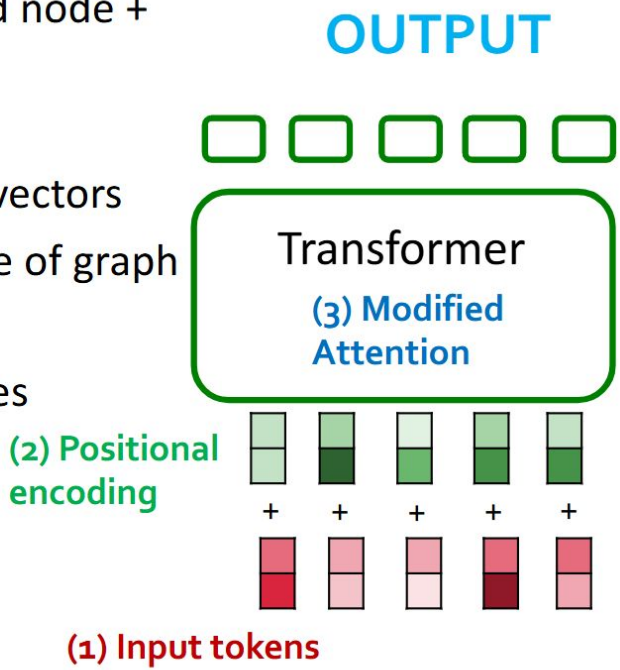
- Step 1: Compute eigenvectors
- Step 2: get eigenvector embeddings using SignNet
- Step 3: concatenate SignNet embeddings with node features X
- Step 4: pass through main GNN/Transformer as usual.
- Step 5: Backpropagate gradients to train SignNet + Prediction model jointly.



Summary: Graph Transformer Design Space

- **(1) Tokenization**
  - Usually node features
  - Other options, such as subgraphs, and node + edge features (not discussed today)
- **(2) Positional Encoding**
  - Relative distances, or Laplacian eigenvectors
  - Gives Transformer adjacency structure of graph
- **(3) Modified Attention**
  - Reweight attention using edge features

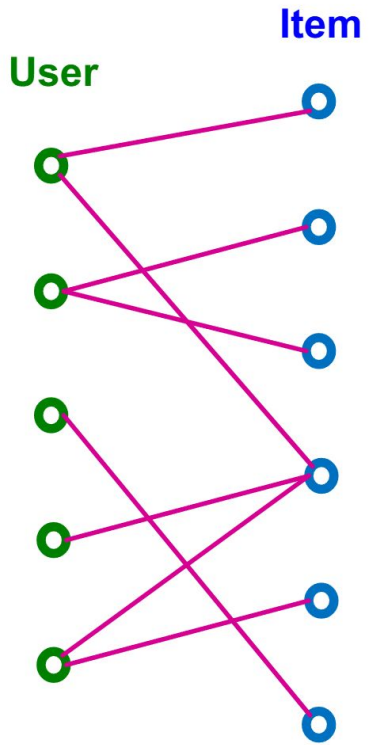




Recommender System

Recommender System as a Graph

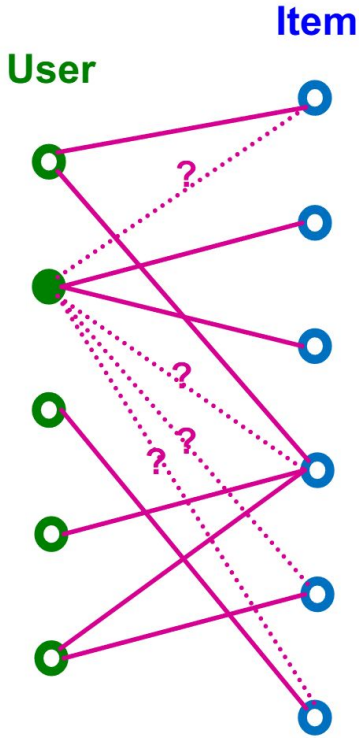
- Recommender system can be naturally modeled as a **bipartite graph**
  - A graph with two node types: **users** and **items**.
  - **Edges** connect users and items
    - Indicates user-item interaction (e.g., click, purchase, review etc.)
    - Often associated with timestamp (timing of the interaction).



Taken from Stanford CS224W course: http://cs244w.stanford.edu

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** $f(u, v)$.

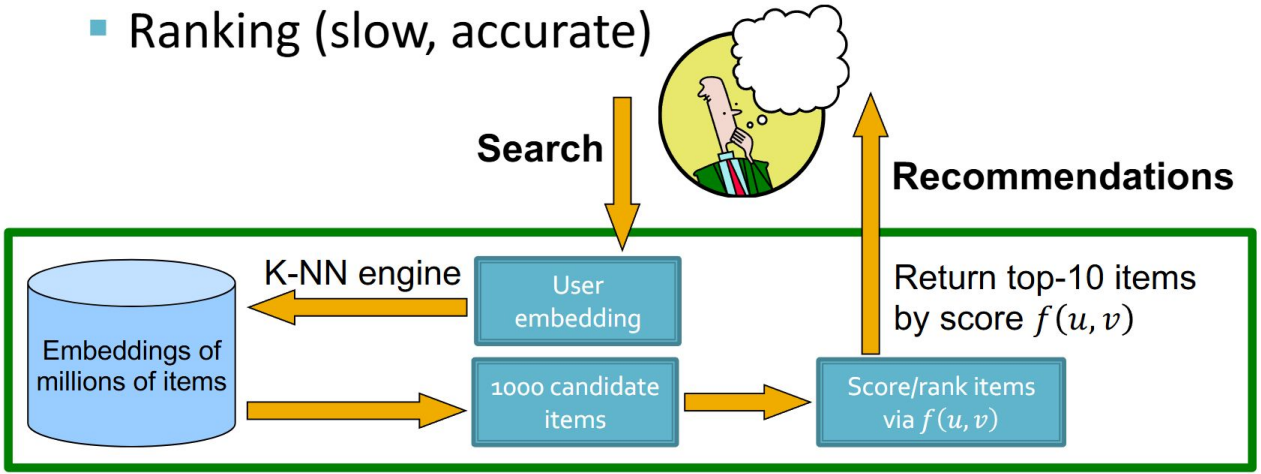


Taken from Stanford CS224W course: http://cs244w.stanford.edu

Modern Recommender System

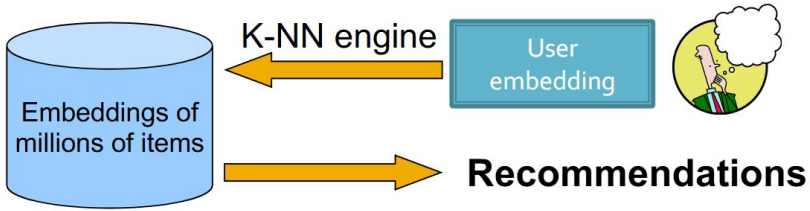
- **Problem:** Cannot evaluate $f(u, v)$ for every user u – item v pair.
- **Solution:** 2-stage process:
  - Candidate generation (cheap, fast)
  - Ranking (slow, accurate)

Example $f(u, v)$:
$f(u, v) = z_u \cdot z_v$



Taken from Stanford CS224W course: http://cs244w.stanford.edu

Recommendations with LLMs?



- **GNN** to embed users & products
- **LLM** to textify products and embed them
  - user emb. is avg. of purchased product embs.
- H&M fashion recommendation:

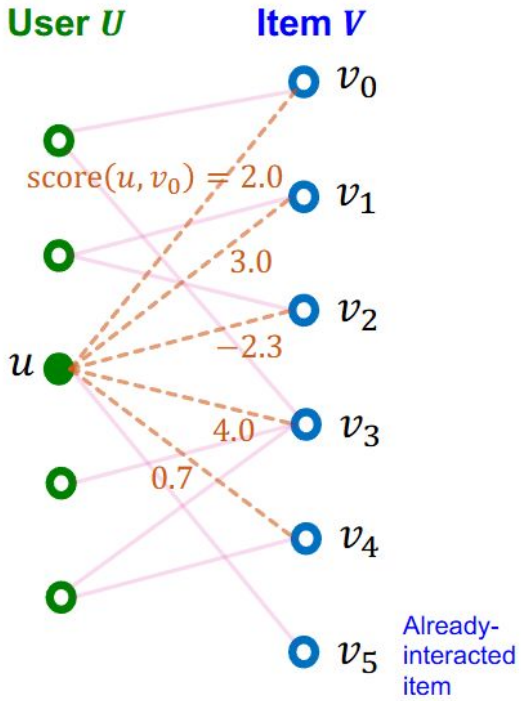
	MAP@12	PRECISION@12	RECALL@12	F1@12
LLM	0.00190 (-93.33%)	0.00329 (-67.84%)	0.00119 (-97.73%)	0.0071 (-54.60%)
GNN	0.02856 15x better	0.01023 3x	0.05234 40x	0.01564 2x

More info: [Do Large Language Models make accurate personalized recommendations?]

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Score Function

- To get the top-K items, we need a score function for user-item interaction:
  - For $u \in U, v \in V$, we need to get a real-valued scalar $\text{score}(u, v)$.
  - **K items with the largest scores for a given user u** (excluding already-interacted items) are then recommended.

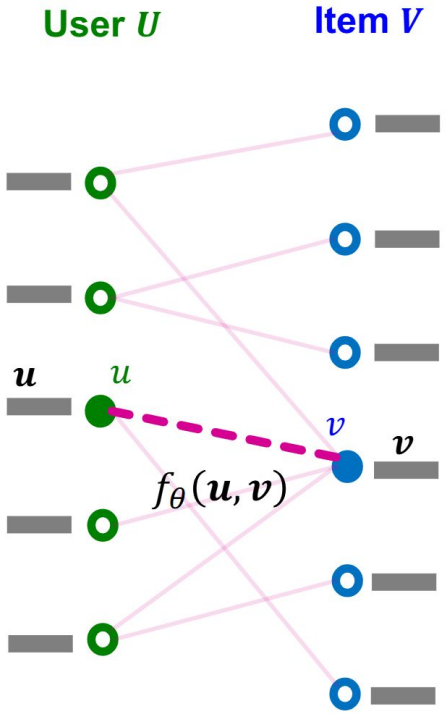


For $K = 2$, recommended items for user u would be $\{v_1, v_3\}$.

Taken from Stanford CS224W course: http://cs244w.stanford.edu

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, $\text{score}(u, v) \equiv f_\theta(\boldsymbol{u}, \boldsymbol{v})$

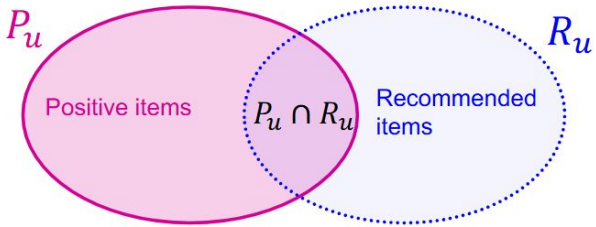


Taken from Stanford CS224W course: http://cs244w.stanford.edu

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.

- **Recall@K for user u is $|\boldsymbol{P_u} \cap \boldsymbol{R_u}|/|\boldsymbol{P_u}|$.**
  - Higher value indicates more positive items are recommended in top-K for user u.



- The final Recall@K is computed by averaging the recall values across all users.

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Surrogate Loss Functions

- The original training objective (recall@K) is ***not* differentiable**.
  - *Cannot* apply efficient gradient-based optimization.
- Two **surrogate loss functions** are widely-used to enable efficient gradient-based optimization.
  - Binary loss

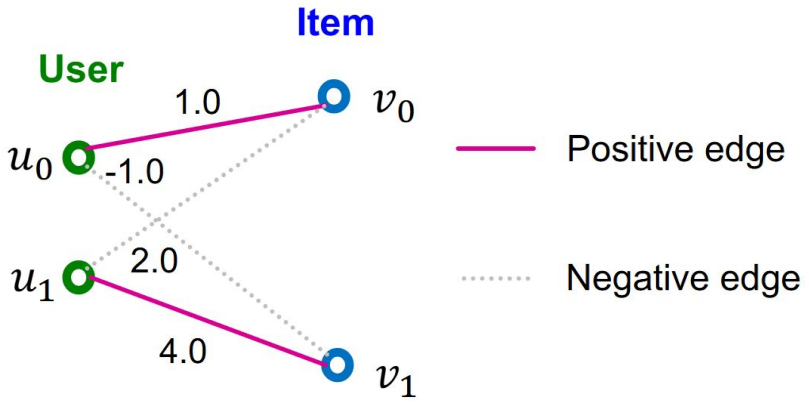
$$\mathcal{L}_{\text{binary}} = -\sum_{(u,i)} \left[y_{ui} \log(\hat{y}_{ui}) + (1 - y_{ui}) \log(1 - \hat{y}_{ui}) \right]$$

  - Bayesian Personalized Ranking (BPR) loss
- Surrogate losses are **differentiable** and should **align well with the original training objective**.

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Problem with Binary Loss

- **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).

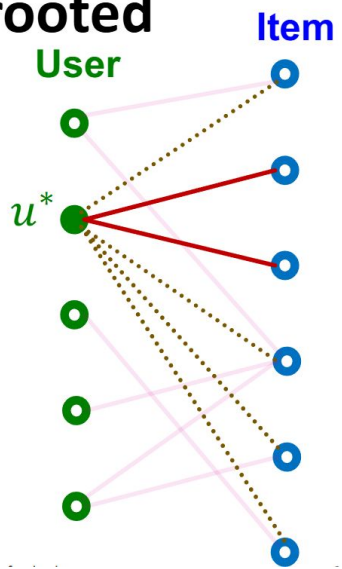


- **Key insight**: The binary loss is **non-personalized** in the sense that the **positive/negative edges are considered *across ALL users at once*.**
- However, the recall metric is inherently **personalized (defined for each user)**.
  - The non-personalized binary loss is overly-stringent for the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

Taken from Stanford CS224W course: http://cs244w.stanford.edu

BPR Loss

- **Bayesian Personalized Ranking (BPR) loss** is a personalized surrogate loss that aligns better with the recall@K metric.
- For each user $u^* \in \boldsymbol{U}$, define the **rooted positive/negative edges** as
  - Positive edges rooted at u^*
    - $\boldsymbol{E}(u^*) \equiv \{(u^*, v) \mid (u^*, v) \in \boldsymbol{E}\}$
  - Negative edges rooted at u^*
    - $\boldsymbol{E}_{\mathbf{neg}}(u^*) \equiv \{(u^*, v) \mid (u^*, v) \in \boldsymbol{E}_{\mathbf{neg}}\}$



Note: The term "Bayesian" is not essential to the loss definition. The original paper [Rendle et al. 2009] considers the Bayesian prior over parameters (essentially acts as a parameter regularization), which we omit here.

Taken from Stanford CS224W course: http://cs244w.stanford.edu

BPR Loss

- **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 u^*:**

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

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

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

Taken from Stanford CS224W course: http://cs244w.stanford.edu

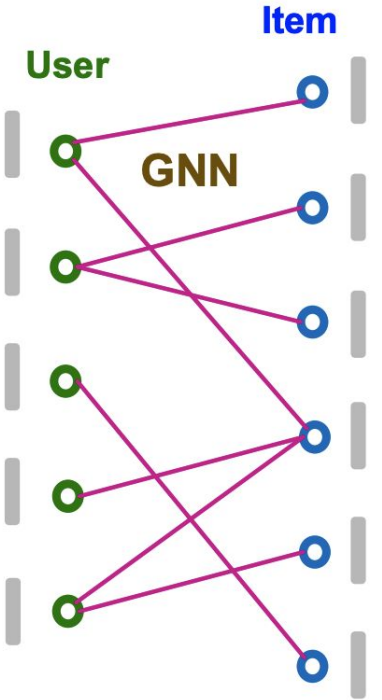
Why Embedding Models Work

- 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.

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Neural Graph Collaborative Filtering

- **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

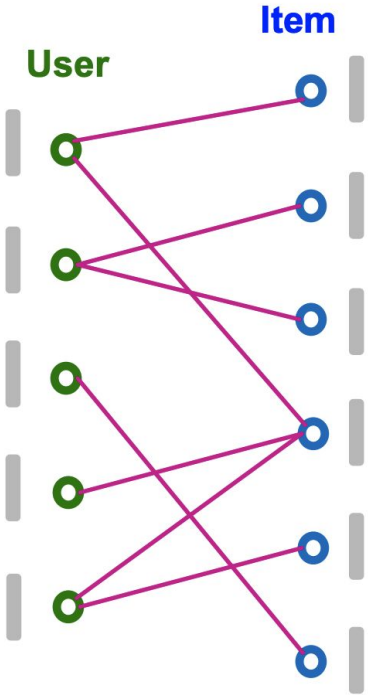


Shallow user/item embeddings (learnable)

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Initial Node Embeddings

- 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.



Learnable shallow user/item embeddings

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Neighbor Aggregation

- Iteratively update node embeddings using neighboring embeddings.

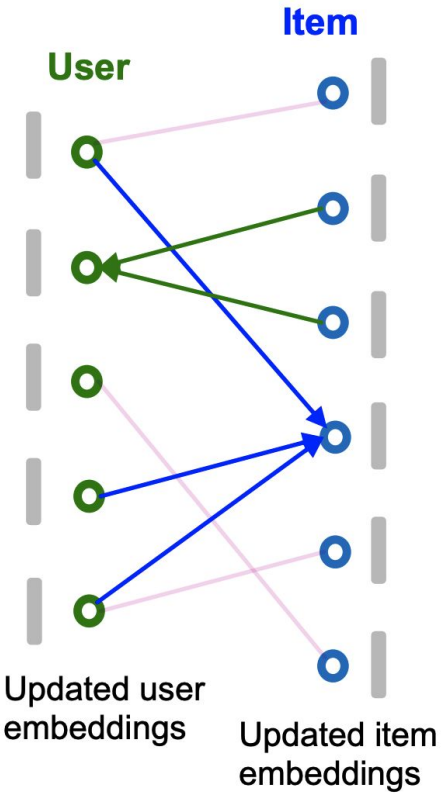
$$\boldsymbol{h}_v^{(k+1)} = \text{COMBINE}\left(\boldsymbol{h}_v^{(k)}, \text{AGGR}\left(\left\{\boldsymbol{h}_u^{(k)}\right\}_{u \in N(v)}\right)\right)$$

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

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



Taken from Stanford CS224W course: http://cs244w.stanford.edu

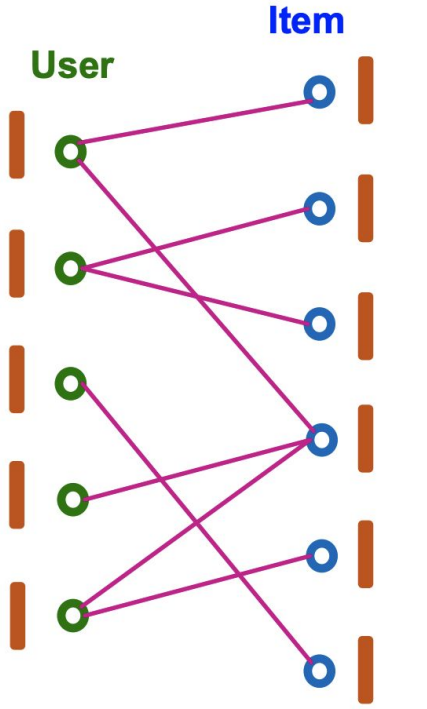
Final Embeddings and Score Function

- 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

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



Taken from Stanford CS224W course: http://cs244w.stanford.edu

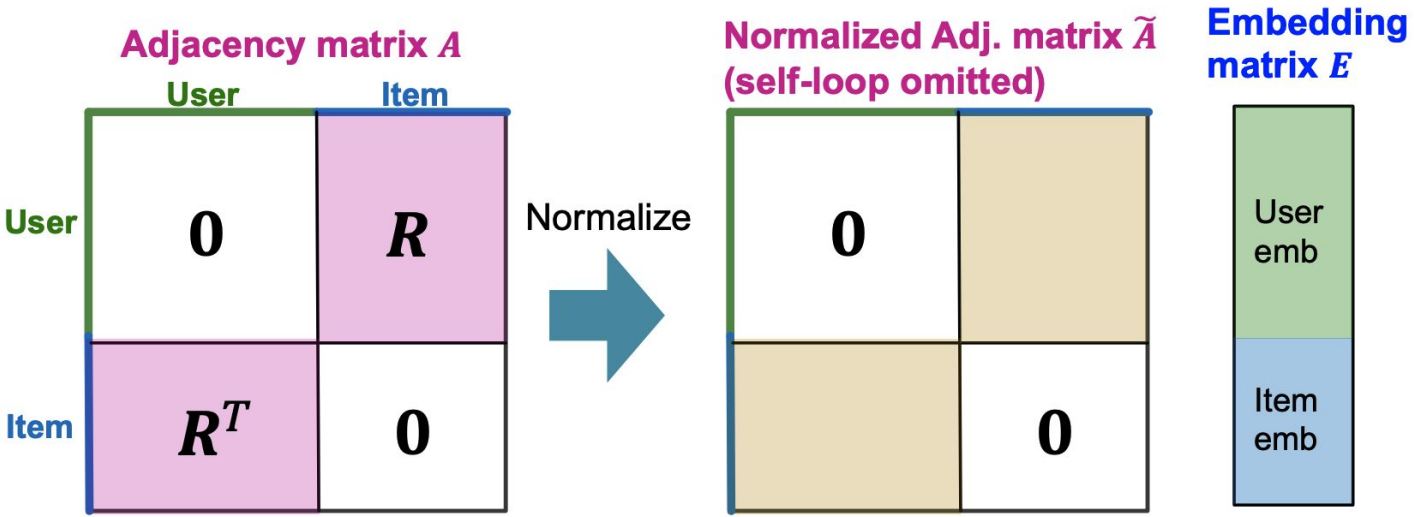
NGCF: Summary

- Conventional collaborative filtering uses shallow user/item embeddings.
  - The embeddings do ***not explicitly*** **model graph structure**.
  - The training objective **does not model high-order graph structure.**
- **NGCF uses a GNN to propagate the shallow embeddings.**
  - The embeddings are **explicitly aware of high-order graph structure.**

Taken from Stanford CS224W course: http://cs244w.stanford.edu

LightGCN: Model Overview (1)

- **Given:**
  - **Adjacency matrix A**
  - **Initial learnable embedding matrix E**

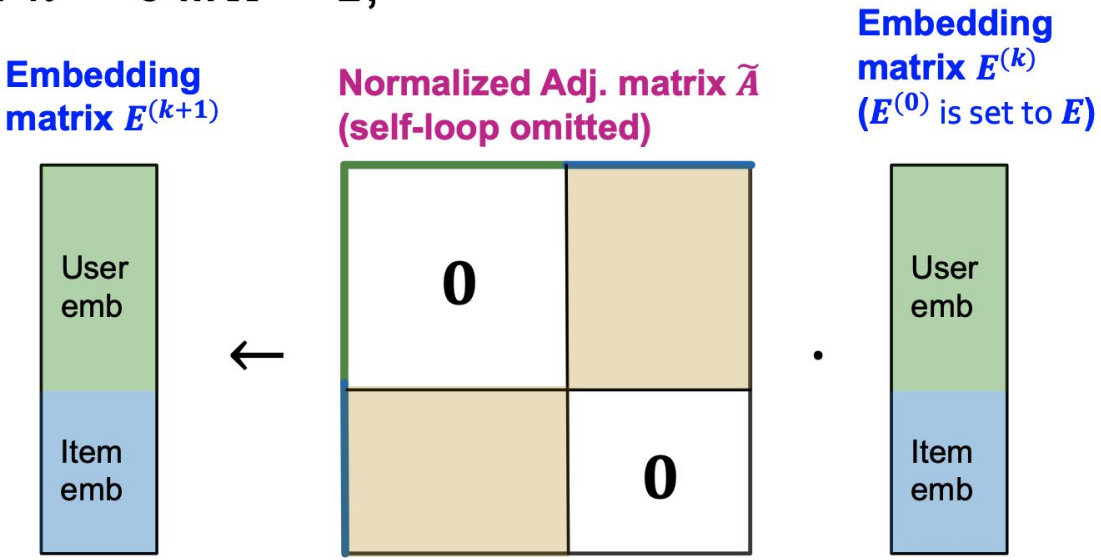


Taken from Stanford CS224W course: http://cs244w.stanford.edu

LightGCN: Model Overview (2)

Iteratively diffuse embedding matrix $\boldsymbol{E}$ using $\tilde{\boldsymbol{A}}$

For $k = 0 \ldots K-1$,



Taken from Stanford CS224W course: http://cs244w.stanford.edu

Matrix Formulation of GCN

- **Define: The diffusion matrix**
- Let $\boldsymbol{D}$ be the degree matrix of $\boldsymbol{A}$.
- Define the normalized adjacency matrix $\tilde{\boldsymbol{A}}$ as

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

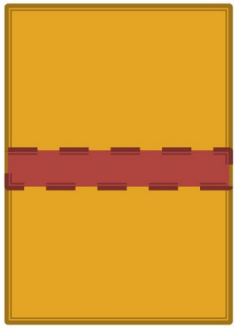
Note: Different from the original GCN, self-connection is omitted here.

- 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:

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

Neighbor aggregation — Learnable linear transformation

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



Each row stores node embedding

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Simplifying GCN (1)

- **Simplify GCN by removing ReLU non-linearity:**

$$\boldsymbol{E}^{(k+1)} = \tilde{\boldsymbol{A}}\boldsymbol{E}^{(k)}\boldsymbol{W}^{(k)}$$

Original idea from SGC [Wu et al. 2019]

- The final node embedding matrix is given as

$$\boldsymbol{E}^{(K)} = \tilde{\boldsymbol{A}}\,\boldsymbol{E}^{(K-1)}\boldsymbol{W}^{(K-1)}$$

$$= \tilde{\boldsymbol{A}}\left(\tilde{\boldsymbol{A}}\boldsymbol{E}^{(K-2)}\boldsymbol{W}^{(K-2)}\right)\boldsymbol{W}^{(K-1)}$$

$$= \tilde{\boldsymbol{A}}\left(\tilde{\boldsymbol{A}}\left(\cdots\left(\tilde{\boldsymbol{A}}\boldsymbol{E}^{(0)}\boldsymbol{W}^{(0)}\right)\cdots\right)\boldsymbol{W}^{(K-2)}\right)\boldsymbol{W}^{(K-1)}$$

Set $\boldsymbol{E}$ as input embedding $\boldsymbol{E}^{(0)}$

$$= \tilde{\boldsymbol{A}}^{K}\,\boldsymbol{E}\left(\boldsymbol{W}^{(0)}\cdots\boldsymbol{W}^{(K-1)}\right)$$

Taken from Stanford CS224W course: http://cs244w.stanford.edu

Simplifying GCN (2)

- Removing ReLU significantly simplifies GCN!

$$\boldsymbol{E}^{(K)} = \boxed{\widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{E}}\,\boldsymbol{W} \qquad \boldsymbol{W} \equiv \boldsymbol{W}^{(0)} \ldots \boldsymbol{W}^{(K-1)}$$

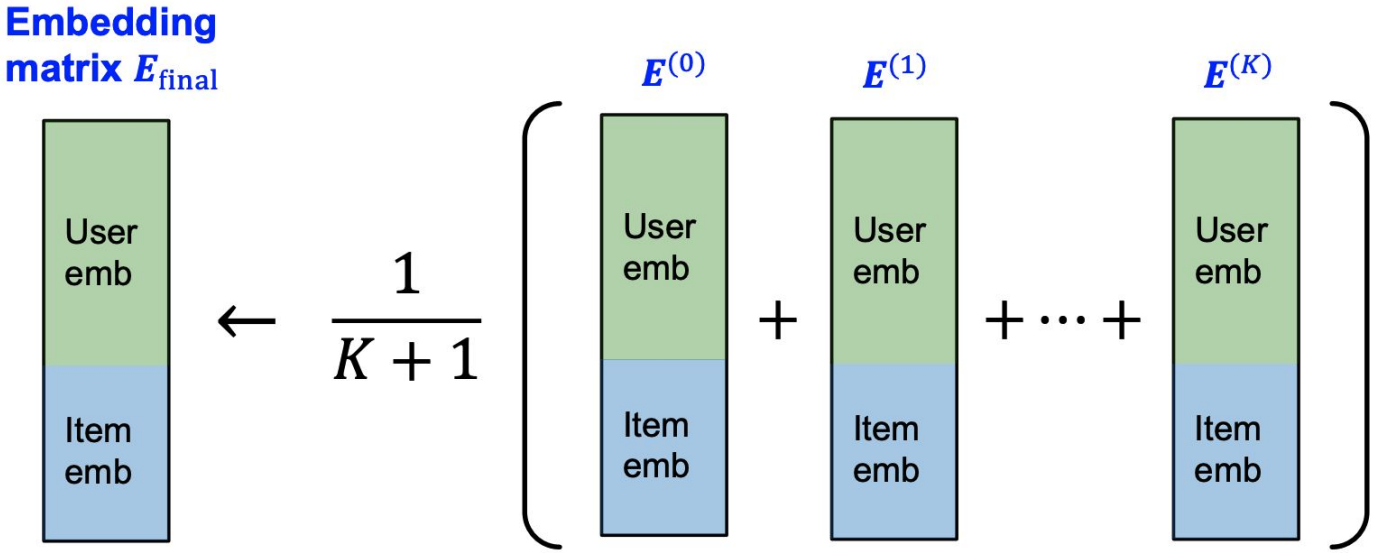
Diffusing node embeddings along the graph

- **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.
  - **Note:** $\widetilde{\boldsymbol{A}}^{K}$ is dense and never gets materialized. Instead, the above iterative matrix-vector product is used to compute $\widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{E}$.

Taken from Stanford CS224W course: http://cs244w.stanford.edu

LightGCN: Model Overview (3)

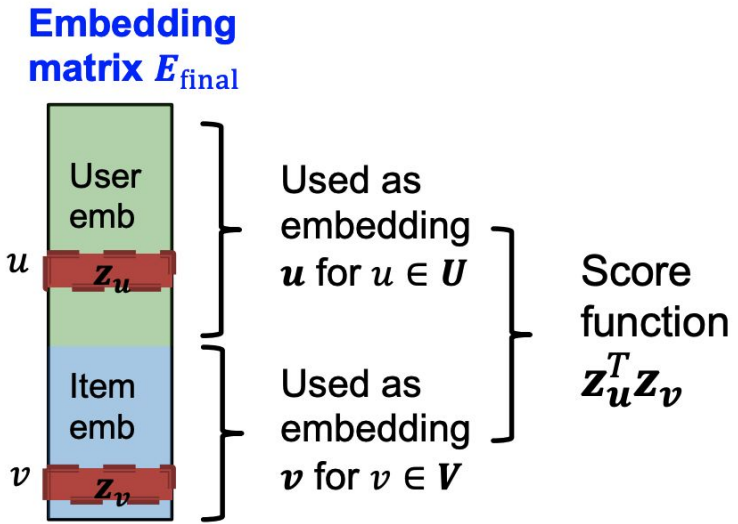
- Average the embedding matrices at different scales.



Taken from Stanford CS224W course: http://cs244w.stanford.edu

LightGCN: Model Overview (4)

- **Score function:**
  - Use user/item vectors from $\boldsymbol{E}_{\text{final}}$ to score user-item interaction



Taken from Stanford CS224W course: http://cs244w.stanford.edu

LightGCN: Summary

- LightGCN simplifies NGCF by **removing the learnable parameters of GNNs.**
- **Learnable parameters are all in the shallow input node embeddings.**
  - Diffusion propagation only involves matrix-vector multiplication.
  - The simplification leads to better empirical performance than NGCF.

Taken from Stanford CS224W course: http://cs244w.stanford.edu