Evaluating Graph Transformers for Scalable Social Media Recommendations

Aleck Wu a5wu@ucsd.edu
SangGyu An sgan@ucsd.edu
Yusu Wang yusuwang@ucsd.edu
Gal Mishne gmishne@ucsd.edu

Abstract

We develop a scalable recommendation system for Bluesky, a decentralized social network on the AT Protocol. Traditional popularity-based methods relying on post likes lack personalization and suffer from temporal bias. To address this, we implement GraphRec, a graph-based model leveraging user-post interactions.

Our two-stage pipeline includes candidate generation using collaborative filtering and consumer-producer embeddings, followed by ranking with GraphRec, incorporating transformers and temporal graph learning. A patching strategy improves efficiency for scalability.

We discuss challenges in offline evaluation and real-world deployment. Future work includes optimizing real-time inference and multi-interest embeddings.

Website: https://bbeat2782.github.io/Bluesky-GraphRec/
Code: https://github.com/bbeat2782/Bluesky-GraphRec

1 Introduction

The technology underlying sites like Bluesky represents a radically different take on social media. Rather than opaque algorithms that are controlled by centralized platforms, the Bluesky platform is designed to be decentralized. More specifically the underlying AT Protocol is open, and anyone can come and see the stream of real-time data. This opens up new opportunities for developers to build applications and even custom recommendation algorithms on the platform, and for regular users to gain deep insights into the shape of a social media platform in a way they could never do previously.

For systems to gain footholds with smaller players, the complexity of a system must be kept to a minimum. Modern large-scale social media and recommendation systems have grown complex(Twitter (2023)), containing large multi-part, multi-step pipelines. There is a need for simpler, scalable solutions to these problems.

This project aims to improve personalized feed recommendations by developing a scalable, graph learning model that models user interests dynamically over time. The main goals of the project are two-fold. The first is to test the viability of the transformer architecture on large-scale graph data, comparing them to Message Passing Neural Networks (MPNNs). The second is to build a scalable recommendation system that can potentially adapt to changing user interests and new content.

While current graph transformer models have demonstrated their effectiveness on various benchmarks, we want to investigate whether they can effectively scale on large-scale social media data, and whether or not they can effectively compete with MPNNs in performance and speed. While current recommendation algorithms perform well in suggesting items users might find interesting, they struggle with a rabbit hole problem and adapting to changing user wants. These limitations arise because traditional algorithms struggle to adapt to the dynamic and temporal nature of user preferences.

To address these challenges, this project will develop a scalable recommendation system using temporal graphs to represent users' dynamic interests. By attempting to leverage transformers, the recommendation system will capture the temporal dynamics of user interactions and adapt to changes in user preferences over time. This approach ensures the recommendation model remains both relevant and responsive to evolving user behaviors.

Recently, there have been several attempts to combine dynamic graphs with transformers. One recurring idea is using a patching technique that divides each sequence into multiple patches before inputting them to the transformer (Yu et al. (2023), Pandey, Sarkar and Comar (2024)). Another idea proposed by Yu et al. (2023) to reduce computational load is to only consider nodes' historical first-hop interactions. We hope to build upon their work to explore how incorporating diverse features and experimenting with various architectures can further enhance dynamic graph learning.

2 On the Nature of Recommender Systems

Different data requires the designs of different types of recommender systems. There is no one-size-fits-all approach to recommendation. Instead there are trade-offs.

The site most similar to Bluesky would be Twitter, which has open sourced

3 Methods

3.1 Data Collection

The data is collected from the entire bluesky network using an open source library(cite). We specifically look at posts, likes, and follows as these make up the majority(96%) of the data, in addition to them being the most high-signal. We do preprocessing on the data to ensure that it is in the appropriate format for our pipeline. Because the data is large and growing constantly (1.5TB+), we conduct our experiments on the early 1-year section of the network.

3.2 Candidate Generation Framework

3.2.1 Theory

Recommendation systems and search engines can be broadly summarized into a framework of filtering/retrieval and refinement/reranking. One model for recommendation on the full network is infeasible for hundreds of millions to billions of nodes. Therefore, a 1st, simpler model is used to retrieve a smaller candidates of items, which are passed through a more robust second model for re-ranking. For efficient search and recommendations, users and items are encoded into a shared latent space, making efficient cosine similarity and dot product operations possible for similarity search/recommendation.

3.2.2 Collaborative Filtering

User-Item based Collaborative filtering leverages patterns of user-item interactions to infer preferences. It assumes that users that interact similarly in the past will interact similarly in the future. Collaborative filtering has the remarkable ability to model implicit relationships without relying on encoding explicit features, such as text, images, and video.

The goal is to approximately reconstruct the full rating matrix R as a product of two lower-dimensional matrices:

$$R \approx UV^T$$

with:

$$\min_{U,V} ||R - UV^T||^2$$

minimizing the reconstruction error where:

- $U \in \mathbb{R}^{|U| \times d}$ represents latent user factors,
- $V \in \mathbb{R}^{|I| \times d}$ represents latent item factors,
- d is the dimensionality of the latent space.

Reconstructing R allows us to "fill in the blanks" and predict posts that users could like.

The problem with Bluesky and many other large-scale social platforms is 1) the sheer size of the data, and 2) the real-time nature of the data. Directly computing $R = UV^T$ is expensive, and because of the real-time nature of the data, it is intractable to compute it each time there is a new interaction. Instead, we need a way to incrementally update the one of the latent factor matrices.

We can get user embeddings without factorizing the full matrix with, UU^T, which represents the user-user follow matrix. By factorizing this matrix, which is orders of magnitudes smaller, we can efficiently get user embeddings. We can also take it a step further. Since Bluesky follows a consumer-producer model, where the majority of users follow and consume content made by a few producers, we can further cut down on the user-user graph by only using small portion of the users on the right side of the bipartite graph (those with more than 30 followers). Since user-user follows don't change as often as post interactions, we can compute this in nightly or in batches. For posts, each post $V_j^{(0)}$ is initially represented as a producer vector or zero vector.

Table 1: Estimated relationship cardinalities (temporary)

Relationship Type	
User-Post	1.0
User-User (All)	0.03
Consumer-Producer	0.003

Update Rule: When a user k interacts with post j (e.g. by liking it), the post embedding is updated as:

$$V_j \leftarrow V_j + U_k$$

In matrix form, the post embeddings could also be expressed as:

$$V = R^T U$$

Now we can update it incrementally this way. With a GNN analogy, the update resembles a message-passing mechanism in GNNs where each user sends a "message" (its embedding) to a post, and the post aggregates these messages using an aggregation function (e.g. mean).

Finally there is the problem of cold-start, which we can alleviate through the use of heuristics such as assigning users to the most popular communities.

3.3 On Metrics

For developers to build an effective recommendation system on the Bluesky network, there must be high signal inherent in the data. The decentralized AT Protocol on Bluesky has the

following high signal data points: likes, follows, and posts. Developers must rely on offline evaluation metrics, since they do not have access to most users. As such, this puts them at a disadvantage compared to operators like Bluesky that can evaluate models with online metrics such as A/B testing on users that are using their algorithm.

For twitter's algorithm(cite), they make use of multiple graph-based approaches for generating candidates.

There is inherent irreducible bias in offline evaluation. Recommender systems do not predict what users will do; rather, they offer choices to the users that themselves affect user actions. This is excacerbated by the fact that users follow a different algorithm, namely Bluesky's. Thus, maximizing hit rate for recommending posts would be somewhat equivalent to maximizing closeness to Bluesky's "For You" algorithm.

Ultimately, online A/B tests are the gold standard for testing recommendation systems. Offline metrics can only serve as proxies and might lead to conclusions that don't hold up in production. That being said, if we look at the nature of likes on Bluesky, users may use likes to encourage the algorithm to show them more of similar content. Compared to watch time, likes are likely a strong indication that a user has a positive affinity for a post.

For our metrics, for the first stage we use hit rate@k and recall@k, while for the second stage we use Mean-Reciprocal Rank(MRR).

3.4 Challenges

To build a sucessful recommendation algorithm for bluesky, we need to take into account the fact the users may have multiple, diverse interests(e.g. snowboarding and tech news). We model this as a temporal problem, assuming that users will naturally gravitate to different topics over time.

3.5 GraphRec Model Implementation

After reducing the computational complexity by limiting the number of neighbors considered per node, we implement the GraphRec model for link prediction. The model extracts 2-hops neighbors and their features, then applies transformers to capture long-term temporal dependencies in user-post interactions.

Algorithm 1: GraphRec Model Implementation

Input: $\mathcal{G} = (\mathcal{U} \cup \mathcal{P}, \mathcal{E})$, where $\mathcal{G}$ represents a graph, $\mathcal{U}$ and $\mathcal{P}$ are user and post nodes, and $\mathcal{E}$ represents edges.
Output: $\hat{y}$, where $\hat{y}$ represents link prediction score.
Step 1: Sampling Past Interactions:

$$\mathcal{N}_{\text{recent}}(v_i, t_i) = \text{Top-}k(\mathcal{N}(v_i, t_i), \text{key} = t_j)$$

Explanation: For a given node v_i, sample its most recent k neighbors from its interaction history before a specific time t_i.
Step 2: Features: Node features $\mathbf{X}_{node,i}$ are preprocessed and temporal features $\mathbf{X}_{time,t}$ are encoded:

$$\mathbf{X}_t = \sqrt{\frac{1}{d_T}}[\cos(w_1 \Delta t'), \sin(w_1 \Delta t'), \ldots, \cos(w_{d_T} \Delta t'), \sin(w_{d_T} \Delta t')]$$

where $\Delta t' = t - t'$ and d_T is the encoding dimension.
Neighbor co-occurrence features are encoded:

$$\mathbf{H}_{co}(v_i) = \sum_{j \in \mathcal{N}(v_i)} \sigma\left(W_2 \cdot \text{ReLU}\left(W_1 \sum_{n \in \mathcal{N}(v_i)} \mathbb{1}(n \in \mathcal{N}(v_j)) + b_1\right) + b_2\right)$$

where $\mathcal{N}(v_i)$ denotes the set of neighbors of node v_i, $\mathbb{1}$ is an indicator function, $W_1, W_2 \in \mathbb{R}^{d \times d}$ are learnable weight matrices, and $b_1, b_2 \in \mathbb{R}^d$ are learnable bias terms.
Step 3: Patching: Divide sequences into patches:

$$\mathbf{X} \to \mathbf{X}_{\text{patched}} \in \mathbb{R}^{\frac{L_{\max}}{p} \times (p \cdot d)}$$

where $L_{\max}$ is max input sequence length, p is patch size, and d is feature dimension.
Step 4: Projection: Project features to shared space:

$$\mathbf{X}_{\text{proj}} = \mathbf{X}_{\text{patched}} \cdot \mathbf{W}$$

where $\mathbf{W} \in \mathbb{R}^{p \cdot d \times d'}$ with d' as project dimension.
Step 5: Concatenation: Combine source and destination node features:

$$\mathbf{X} = [\mathbf{X}_{\text{src\_proj}}; \mathbf{X}_{\text{dst\_proj}}] \in \mathbb{R}^{2 \cdot \text{num patches} \times d'}$$

where num patches $= \frac{L_{\max}}{p}$.
Step 6: Transformer Encoding: Apply self-attention to learn dependencies within and across sequences:

$$\mathbf{Q} = \mathbf{X}_{\text{norm}} \mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}_{\text{norm}} \mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}_{\text{norm}} \mathbf{W}_V$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\mathbf{X}_{\text{attn}} = \mathbf{X} + \text{Dropout}(\text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}))$$

$$\mathbf{X}_{\text{out}} = \mathbf{X}_{\text{attn}} + \text{Dropout}(\text{FFN}(\mathbf{X}_{\text{attn}}))$$

where:

- $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d' \times d_k}$ are projection matrices.
- $\mathbf{X}_{\text{out}} \in \mathbb{R}^{2 \cdot \text{num patches} \times d'}$.

Step 7: Node Embeddings: Compute source and destination embeddings:

$$\mathbf{H} = \text{Mean}(\mathbf{X}) \cdot \mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}} \in \mathbb{R}^{d_{\text{out}}}$$

where taking the mean is edge-level pooling, $\mathbf{W}_{\text{out}} \in \mathbb{R}^{d' \times d_{\text{out}}}$, and $\mathbf{b}_{\text{out}} \in \mathbb{R}^{d_{\text{out}}}$.
Step 8: Link Prediction:

$$\hat{y} = \text{ReLU}([\mathbf{H}_{\text{src}}; \mathbf{H}_{\text{dst}}] \cdot \mathbf{W}_1 + \mathbf{b}_1) \cdot \mathbf{W}_2 + \mathbf{b}_2$$

where $\mathbf{W}_1 \in \mathbb{R}^{2d_{\text{node}} \times d_{\text{hidden}}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{hidden}}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{hidden}} \times 1}$, and $\mathbf{b}_2 \in \mathbb{R}^1$.

3.5.1 Sampling Past Interactions

For a given node v_i, sample its 1-hop neighbors based on the most recent interactions that occurred before a specific interaction time t_i. The neighbors are defined as:

$$\mathcal{N}(v_i, t_i) = \{v_j \mid (v_i, v_j, t_j) \in \mathcal{E},\ t_j < t_i\}, \quad |\mathcal{N}(v_i, t_i)| = \text{max input sequence length}$$

To ensure consistency, only the most recent interactions are retained:

$$\mathcal{N}_{\text{recent}}(v_i, t_i) = \text{Top-}k\big(\mathcal{N}(v_i, t_i), \text{key} = t_j, \text{descending}\big)$$

where:

- $\mathcal{N}(v_i, t_i)$ is the set of neighbors for node v_i at time t_i, filtered to include only interactions before t_i.
- Top-k selects the $k = \text{max input sequence length}$ neighbors with the most recent interaction times t_j.
- key $= t_j$ ensures neighbors are sorted by their interaction times t_j in descending order.

3.5.2 Features

- User-node features: $\mathbf{U}_i \in \mathbb{R}^{64}$, obtained via low-rank matrix factorization using truncated SVD on the user-producer interaction matrix. These embeddings capture latent user preferences and are L2-normalized for similarity-based retrieval.
- Post-node features: $\mathbf{P}_i \in \mathbb{R}^{128}$, with text embeddings generated by Jina Embeddings v3 (Sturua et al. 2024). These are concatenated with the post embeddings we get from the 1st step of the pipeline.
- Temporal features (interaction times): Temporal differences between interactions are encoded using a method described by Xu et al. (2020), where relative time intervals rather than absolute timestamps are used to capture meaningful patterns in interactions. The goal is to learn a continuous functional mapping:

  $$\Phi : T \to \mathbb{R}^{d_T}$$

  which maps time intervals to a d_T-dimensional vector space, serving as a replacement for positional encoding in self-attention. The encoding is defined as:

  $$\mathbf{X}^t_{U_i,P_i} = \sqrt{\frac{1}{d_T}}\left[\cos(w_1\Delta t'), \sin(w_1\Delta t'), \ldots, \cos(w_{d_T}\Delta t'), \sin(w_{d_T}\Delta t')\right]$$

  where:
  - $\Delta t' = |t - t'|$ is the relative time difference between two interactions.
  - d_T is the encoding dimension.
  - $w_1, w_2, \ldots, w_{d_T}$ are trainable parameters optimized jointly with the model.

  This encoding method is compatible with self-attention and can effectively replace traditional positional encodings to learn interaction-based temporal patterns.

- Neighbor co-occurrence features: These features capture shared nodes between users and posts by computing the frequency with which a given node appears in the neighborhoods of others. The neighbor co-occurrence matrix is defined as:

$$C(v_i, v_j) = \sum_{n \in \mathcal{N}(v_i)} \mathbb{1}(n \in \mathcal{N}(v_j))$$

where $\mathcal{N}(v_i)$ represents the set of neighbors of node v_i, and $\mathbb{1}(\cdot)$ is an indicator function that counts occurrences. These co-occurrence counts are transformed using a non-linear projection:

$$\mathbf{H}_{co}(v_i) = \sum_{j \in \mathcal{N}(v_i)} \sigma\left(W_2 \cdot \mathrm{ReLU}\left(W_1 C(v_i, \mathcal{N}(v_i)) + b_1\right) + b_2\right)$$

where W_1, W_2 are learnable projection matrices, b_1, b_2 are bias terms, and $\sigma(\cdot)$ is an activation function such as sigmoid or softmax. This step ensures that the model learns structural similarities between nodes in the interaction graph.
To incorporate broader structural context, 1-hop and 2-hop neighbors are concatenated before computing co-occurrence counts, allowing the model to capture both direct and indirect interactions without differentiating them.

3.5.3 Patching

To capture long-term temporal dependencies while maintaining computational efficiency, a patching technique inspired by (Yu et al. 2023) is adopted. This approach splits each sequence into multiple patches, which are then fed into a Transformer. The patching strategy effectively preserves local temporal proximities and enables the model to efficiently process longer histories.

The idea of dividing the input sequence into patches has been applied in various domains. For instance:

- Vision Transformer (ViT) (Dosovitskiy et al. 2021) splits an image into multiple patches and feeds the sequence of linear embeddings of these patches into a Transformer, achieving remarkable performance in image classification tasks.
- PatchTST (Nie et al. 2023) divides time series data into subseries-level patches, which are processed by a channel-independent Transformer for long-term multivariate time series forecasting.

In this work, a similar patching mechanism is applied to sequences of node and temporal features. Specifically:

- A node features, initially of shape [max input sequence length, feature dimension size], are divided into patches depending on patch size p.

$$\mathbf{X} \in \mathbb{R}^{\text{max input sequence length} \times \text{feature dim}} \quad \rightarrow \quad \mathbf{X}_{\text{patched}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times p \cdot \text{feature dim}}$$

By splitting the input into patches, the Transformer processes sub-sequences instead of the entire sequence, reducing the computational complexity typical of self-attention mechanisms.

This approach not only aligns with prior studies on leveraging patch-based input transformations but also ensures that the temporal relationships and structural proximities within the sequence are effectively preserved, enhancing the model's ability to learn from long histories.

3.5.4 Projection

After dividing the input sequences into patches, feature projection is applied to align the patch encoding dimensions. This is achieved using separate linear layers for node features and temporal features.

The projection layers are:

$$\text{Projection Layer} = \begin{cases} \mathbf{W}_{\text{node}} \in \mathbb{R}^{p \cdot \text{node feat dim} \times d'} \\ \mathbf{W}_{\text{time}} \in \mathbb{R}^{p \cdot \text{time feat dim} \times d'} \end{cases}$$

where d' is a project dimension. The features for source and destination nodes are transformed as:

$$\mathbf{X}_{\text{node\_proj}} = \mathbf{X}_{\text{node\_patched}} \cdot \mathbf{W}_{\text{node}}, \quad \mathbf{X}_{\text{time\_proj}} = \mathbf{X}_{\text{time\_patched}} \cdot \mathbf{W}_{\text{time}}$$

where:

- $\mathbf{X}_{\text{node\_patched}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times p \cdot \text{feature dim}}$ is the patched node feature matrix.
- $\mathbf{X}_{\text{node\_proj}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times d'}$ is the projected node feature matrix.
- $\mathbf{W}_{\text{node}}$ and $\mathbf{W}_{\text{time}}$ are learnable parameters.

This feature projection aligns the dimensions of all patches, enabling them to be concatenated and passed into the Transformer layers. It also ensures that both node and temporal information are represented in a unified embedding space.

3.5.5 Transformer Encoding

After feature projection on the input data to have the same dimension, concatenation is applied for each source and destination node.

$$\mathbf{X}_{\text{concat}} = \mathbf{X}_{\text{node\_proj}} || \mathbf{X}_{\text{time\_proj}} \in \mathbb{R}^{\frac{\text{max input sequence length}}{p} \times 2d'}$$

Then, instead of processing source and destination nodes individually, we stacked them before inputting them so that the model can learn the temporal dependencies within and across the sequences. Within-sequence dependencies model how features evolve over time for a specific node type (e.g., user-to-post interactions or post-to-user interactions separately). This captures how a user's interaction history progresses over time or how a post gains interactions. Across-sequence dependencies capture how user and post interactions influence each other over time, helping the model understand cross-sequence relationships between users and posts.

$$\mathbf{X} = [\mathbf{X}_{\text{src\_concat}}; \mathbf{X}_{\text{dst\_concat}}] \in \mathbb{R}^{2 \cdot \frac{\text{max input sequence length}}{p} \times 2d'}$$

Once the concatenated and stacked features are prepared, they are passed through Transformer encoder layers. The Transformer encoder, introduced by (Vaswani et al. 2023), consists of the following key components:

- **Multi-Head Attention:** The model computes self-attention for each sequence of patches to capture temporal and structural dependencies:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

  where:
  - $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{2 \cdot \text{num patches} \times d}$ are the query, key, and value matrices derived from the input, where d is the embedding dimension of each token and $\frac{\text{max input sequence length}}{p} =$ num patches.
  - d_k is the dimensionality of the key vectors.
- **Feed-Forward Network (FFN):** After the attention layer, the output is passed through a two-layer feed-forward network:

$$\text{FFN}(\mathbf{X}) = \text{GELU}(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

  where $\mathbf{W}_1, \mathbf{W}_2$ are trainable weight matrices.
- **Residual Connections and Layer Normalization:** To ensure stable learning, residual connections are applied, followed by layer normalization:

$$\mathbf{X}_{\text{out}} = \text{LayerNorm}(\mathbf{X}_{\text{in}} + \text{Dropout}(\text{Attention/FFN Output}))$$

The entire steps for processing the input tensor $\mathbf{X} \in \mathbb{R}^{2 \cdot \text{num patches} \times 2d'}$ is the following:

$$\mathbf{X}_{\text{norm}} = \text{LayerNorm}(\mathbf{X})$$

$$\mathbf{Q} = \mathbf{X}_{\text{norm}}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}_{\text{norm}}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}_{\text{norm}}\mathbf{W}_V$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\mathbf{X}_{\text{attn}} = \mathbf{X} + \text{Dropout}(\text{MultiHeadAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}))$$

$$\mathbf{X}_{\text{ffn}} = \text{GELU}(\mathbf{X}_{\text{attn}}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

$$\mathbf{X}_{\text{out}} = \mathbf{X}_{\text{attn}} + \text{Dropout}(\mathbf{X}_{\text{ffn}})$$

$$\mathbf{X}_{\text{out}} \in \mathbb{R}^{2 \cdot \text{num patches} \times 2d'}$$

where:

- $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{2d' \times d}$ are trainable weight matrices for computing queries, keys, and values.
- $\mathbf{W}_1 \in \mathbb{R}^{2d' \times d_{\text{hidden}}}, \mathbf{W}_2 \in \mathbb{R}^{d_{\text{hidden}} \times 2d'}$ are FFN weight matrices.
- $\mathbf{b}_1, \mathbf{b}_2$ are bias terms.
- Dropout$(\cdot)$ applies dropout for regularization.
- GELU$(\cdot)$ is the Gaussian Error Linear Unit activation function.
- d' is the project dimension from step 4.

3.5.6 Node Embeddings

After passing through the Transformer encoder layers, the processed tensor is split back into source and destination sequences:

$$\mathbf{X}_{\text{src}} = \mathbf{X}[: \text{num patches}, :], \quad \mathbf{X}_{\text{dst}} = \mathbf{X}[\text{num patches} :, :]$$

The final node embeddings for source and destination nodes are derived by averaging their patch representations and applying a fully connected output layer:

$$\mathbf{H}_{\text{src}} = \text{Mean}(\mathbf{X}_{\text{src}})\mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}} \in \mathbf{R}^{d_{\text{out}}}, \quad \mathbf{H}_{\text{dst}} = \text{Mean}(\mathbf{X}_{\text{dst}})\mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}} \in \mathbf{R}^{d_{\text{out}}}$$

where:

- $\mathbf{H}_{\text{src}}, \mathbf{H}_{\text{dst}} \in \mathbb{R}^{\text{node feat dim}}$ are the final node embeddings.
- $\mathbf{W}_{\text{out}} \in \mathbb{R}^{2 \cdot d' \times \text{node feat dim}}$ is the trainable weight matrix.
- $\mathbf{b}_{\text{out}} \in \mathbb{R}^{\text{node feat dim}}$ is the trainable bias vector.

3.5.7 Link Prediction

The link predictor is implemented by concatenating the source and destination node embeddings and projects them to the desired output dimension. The process involves the following steps:

- **Input Concatenation:** The source ($\mathbf{H}_{\text{src}}$) and destination ($\mathbf{H}_{\text{dst}}$) node embeddings are concatenated along the feature dimension:

  $$\mathbf{H}_{\text{concat}} = [\mathbf{H}_{\text{src}}; \mathbf{H}_{\text{dst}}] \in \mathbb{R}^{2d_{\text{node}}}$$

  where d_{node} is the dimension of the node embeddings.
- **Hidden Layer Transformation:** The concatenated features are passed through a hidden layer with ReLU activation:

  $$\mathbf{H}_{\text{hidden}} = \text{ReLU}(\mathbf{H}_{\text{concat}}\mathbf{W}_1 + \mathbf{b}_1) \in \mathbb{R}^{d_{\text{hidden}}}$$

  where $\mathbf{W}_1 \in \mathbb{R}^{2d_{\text{node}} \times d_{\text{hidden}}}$ and $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{hidden}}}$ are trainable parameters.
- **Output Projection:** The hidden layer output is projected to a scalar value indicating the likelihood of a link:

  $$\hat{y} = \mathbf{H}_{\text{hidden}}\mathbf{W}_2 + \mathbf{b}_2 \in \mathbb{R}$$

  where $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{hidden}} \times 1}$ and $\mathbf{b}_2 \in \mathbb{R}$ are trainable parameters.

The link predictor learns to estimate the score of a link existing between the source and destination nodes based on their embeddings. This score is then evaluated using Bayesian Personalized Ranking (BPR) loss.

3.6 Training

For each positive interaction in the training set, we randomly sample four negative posts from the last 20-minute timeframe to construct negative interactions. The model is trained using Bayesian Personalized Ranking (BPR) loss, optimizing for pairwise ranking. We evaluate model performance using accuracy and pairwise accuracy, ensuring that positive interactions rank higher than negative ones. Mean Reciprocal Rank (MRR) is not used to reduce training time, as computing the full ranking position for each sample is computationally expensive.

3.6.1 Bayesian Personalized Ranking (BPR) Loss

The BPR loss is designed to optimize ranking by enforcing that positive interactions are ranked higher than sampled negative interactions (Rendle et al. 2012). The loss function is defined as:

$$\mathscr{L}_{BPR} = -\frac{1}{|D|} \sum_{(u,p^+,p^-)\in D} \log \sigma(\hat{y}_{up^+} - \hat{y}_{up^-}) + \lambda ||\Theta||^2$$

where:

- (u, p^+, p^-) represents a triplet containing user u, a positive post p^+ that the user interacted with, and a randomly sampled negative post p^-.
- $\hat{y}_{up^+}$ and $\hat{y}_{up^-}$ are the predicted scores for positive and negative interactions.
- $\sigma(x)$ is the sigmoid function.
- $\lambda ||\Theta||^2$ is the regularization term to prevent overfitting.
- D is the total number of positive interactions in the dataset.

3.6.2 Evaluation Metrics

To measure performance, we use:

- **Accuracy@1**: Measures how often the positive is ranked highest.
- **Pairwise Accuracy**: Measures how often the positive interaction is ranked higher than each of the negative interactions:

$$\text{Pairwise Acc} = \frac{1}{|D|} \sum_{(u,p^+,\{p_1^-,\ldots,p_N^-\})\in D} \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(\hat{y}_{up^+} > \hat{y}_{up_i^-})$$

  where:
  - $N = 4$ (number of negative samples per positive sample).
  - $\mathbb{1}(\cdot)$ is an indicator function that returns 1 if the condition is met, otherwise 0.

3.6.3 Training and Inference

The training process consists of 50 epochs with early stopping if there is no improvement for five consecutive epochs. Training is currently performed on about 1 million interactions. After training, the inference step generates recommendations by predicting the most relevant items for each user. The inference phase is conducted on 40,000 users, where the model aims to identify the last item each user interacted with and recommends the most likely next interaction based on their past history.

3.7 Comparisons

3.7.1 Popularity-Based Recommendation

The popularity-based approach recommends Bluesky posts purely based on the number of likes a post has received. The assumption is that highly liked posts are more engaging and relevant to users and users are influenced by social proof, engaging with posts that others have already liked. While this method is simple and computationally efficient, it has several limitations:

- **Lack of Personalization**: All users receive similar recommendations, disregarding individual preferences.
- **Cold Start Problem**: New posts with few or no likes struggle to gain visibility.
- **Temporal Bias**: Older posts with accumulated likes are favored over newer content, leading to potential stagnation in recommendations.

3.7.2 MLP-based Neighborhood Aggregation

Unlike the popularity-based approach, which solely relies on the number of likes, the MLP-based neighborhood aggregation method learns user preferences by leveraging temporal information and interactions. This method utilizes Multi-Layer Perceptrons (MLPs) to aggregate features from a user's local network, dynamically adapting to changing behaviors over time.

The architecture follows a temporal graph representation, where each node (representing a user or post) updates its embedding based on historical interactions. Instead of relying on multi-head attention mechanisms, this approach aggregates neighborhood information using mean pooling and MLP layers.

This approach improves upon the limitations of popularity-based methods by incorporating temporal dependencies, user-specific behavior, and relational information, resulting in a more adaptive and context-aware recommendation system.

The backbone of this model is based on Xu et al. (2020), where the multi-head attention mechanism has been replaced with an MLP to establish a baseline.

3.7.3 Configurations

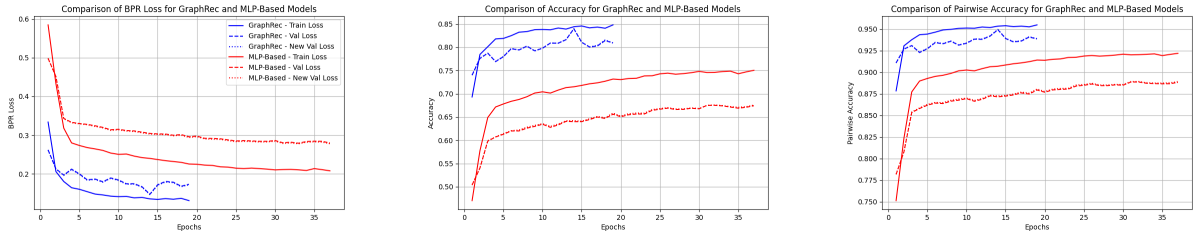
To ensure a fair comparison, we maintain consistent hyperparameters across all models. Each model utilizes a 2-hop neighborhood with 10 neighbors per hop, resulting in a receptive field of $10 + 10 \times 10$ nodes. The time feature dimension is set to 100, while the channel embedding dimension is 50, representing the node embedding size before applying a transformer or MLP. Training is conducted using an Adam optimizer with a learning rate of 0.0001, a dropout rate of 0.1, and no weight decay. Each model is trained for a maximum of 50 epochs, with early stopping triggered after 5 epochs of no improvement on both evaluation metrics, ensuring efficient convergence. The process is repeated 5 times, and all experiments are conducted on a single NVIDIA A5000 GPU.

In addition to these shared hyperparameters, GraphRec incorporates 2 attention heads and a patch size of 5, allowing it to capture more refined structural patterns in the graph data.

4 Results

For candidate generation, the metrics are calculated by simulating the real-time recommendation through a forward-time process. We treat every point of interaction as an opportunity for recommendation. We accumulate the hit rate as well as update the post embeddings after each recommendation event.

4.1 Training Result



(a) BPR Loss (b) Accuracy (c) Pairwise accuracy

Figure 1: BPR Loss, accuracy, and pairwise accuracy for training GraphRec and MLP-Based models

4.2 Training Results

Figure 1 presents a comparison of the training performance between GraphRec and MLP-Based models across BPR Loss, Accuracy, and Pairwise Accuracy. Overall, GraphRec demonstrates superior performance in all three metrics, suggesting that it effectively captures graph-based interactions compared to the MLP-Based model. However, differences between validation and

new node validation results are relatively small, indicating that both models generalize well to unseen interactions, where either the user or post node has not been observed during training.

GraphRec consistently achieves a lower BPR loss than the MLP-Based model, as seen in **Figure 1a**, indicating more effective optimization. While both models show a decreasing loss trend, GraphRec stabilizes at a lower value, whereas MLP-Based exhibits a slower convergence rate.

In terms of accuracy (**Figure 1b**), GraphRec significantly outperforms the MLP-Based model across training, validation, and new node validation sets. On the other hand, the MLP-Based model improves steadily but struggles to close the performance gap. Similarly, in **Figure 1c**, pairwise accuracy trends show that GraphRec reaches higher ranking performance earlier and maintains stability, whereas MLP-Based requires more training epochs to improve.

4.3 Inference Results

4.3.1 Intra-List Diversity (ILD)

Most recommendation models prioritize ranking accuracy, such as Mean Reciprocal Rank (MRR), which optimizes the placement of relevant items. However, focusing solely on accuracy can lead to redundancy in recommendations, where users receive items that are too similar to each other. This reduces content diversity and limits exposure to new or less popular items. To address this, we evaluate Intra-List Diversity (ILD), which quantifies how varied the recommendations are within a user's top-ranked results.

Intra-List Diversity (ILD) measures the pairwise dissimilarity among items in a user's recommendation list. It is computed as:

$$ILD = 1 - \frac{1}{N(N-1)} \sum_{i \neq j} \text{sim}(i, j), \tag{1}$$

where N is the number of recommended items, and $\text{sim}(i, j)$ represents the similarity between items i and j.

- **Higher ILD**: Indicates more diverse recommendations, suggesting a broader content exposure for users.
- **Lower ILD**: Suggests recommendations are highly similar, potentially limiting content discovery.

To maintain consistency across users, we compute ILD for the top-10 recommended items per user. Item similarity is measured using cosine similarity between post feature embeddings. The final ILD score is obtained by averaging across all users in the validation set.

4.3.2 Comparison of Inference Performance

Table 2 presents the inference performance comparison between the three recommendation approaches: Popularity-Based, MLP-Based, and GraphRec. We evaluate each method based

on five key metrics: Mean Reciprocal Rank (MRR), Average Rank, ILD@10, Training Time (1 epoch), and Inference Time (seconds per user).

Table 2: Inference performance comparison between different recommendation models. ↑ indicates that a higher value is better, and ↓ indicates that a lower value is better.

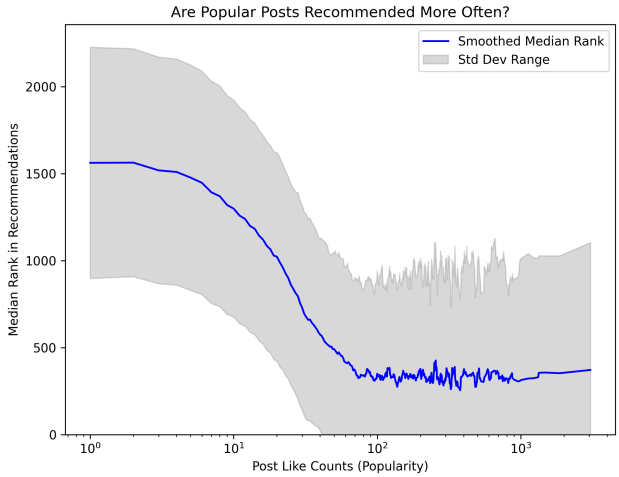
Model	MRR (↑)	Avg Rank (↓)	ILD@10 (↑)	Training Time (1 epoch) (↓)	Inference Time (s/user) (↓)
Popularity-Based	–	–	–	–	–
MLP-Based	–	–	–	–	–
GraphRec	–	–	–	–	–

The table highlights the trade-offs between ranking effectiveness and diversity, as well as computational efficiency during training and inference.

5 Discussion

Look into diversity of recommended posts (can a model recommend niche posts)

In progress



In progress

5.1 Ablation Study

Just throwing some ideas

- Diff num of patches
- Diff num of neighbors
- Diff num of heads?

6 Conclusion

In progress

References

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." [Link]

Nie, Yuqi, Nam H. Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. 2023. "A Time Series is Worth 64 Words: Long-term Forecasting with Transformers." [Link]

Pandey, Deepanshu, Arindam Sarkar, and Prakash Mandayam Comar. 2024. "GLAD: Graph-based long-term attentive dynamic memory for sequential recommendation." In *ECIR 2024*. [Link]

Rendle, Steffen, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2012. "BPR: Bayesian Personalized Ranking from Implicit Feedback." [Link]

Sturua, Saba, Isabelle Mohr, Mohammad Kalim Akram, Michael Günther, Bo Wang, Markus Krimmel, Feng Wang, Georgios Mastrapas, Andreas Koukounas, Andreas Koukounas, Nan Wang, and Han Xiao. 2024. "jina-embeddings-v3: Multilingual Embeddings With Task LoRA." [Link]

Twitter, Inc. 2023. "Twitter's Algorithm." `https://github.com/twitter/the-algorithm`

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. "Attention Is All You Need." [Link]

Xu, Da, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. "Inductive representation learning on temporal graphs." In *International Conference on Learning Representations*. [Link]

da Xu, chuanwei ruan, evren korpeoglu, sushant kumar, and kannan achan. 2020. "Inductive representation learning on temporal graphs." In *International Conference on Learning Representations (ICLR)*.

Yu, Le, Leilei Sun, Bowen Du, and Weifeng Lv. 2023. "Towards Better Dynamic Graph Learning: New Architecture and Unified Library." [Link]