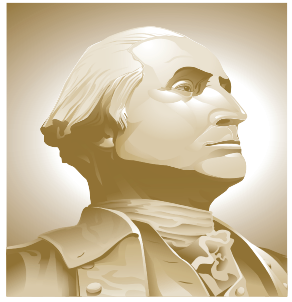
Data Science Capstone

Graph-Based Neural Networks (GNN) Recommendation Systems



THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

A Pipeline of Bipartite Graph Neural Network Operator for Recommendation Systems Using PyTorch Geometric

Heterogeneous Graph Learning

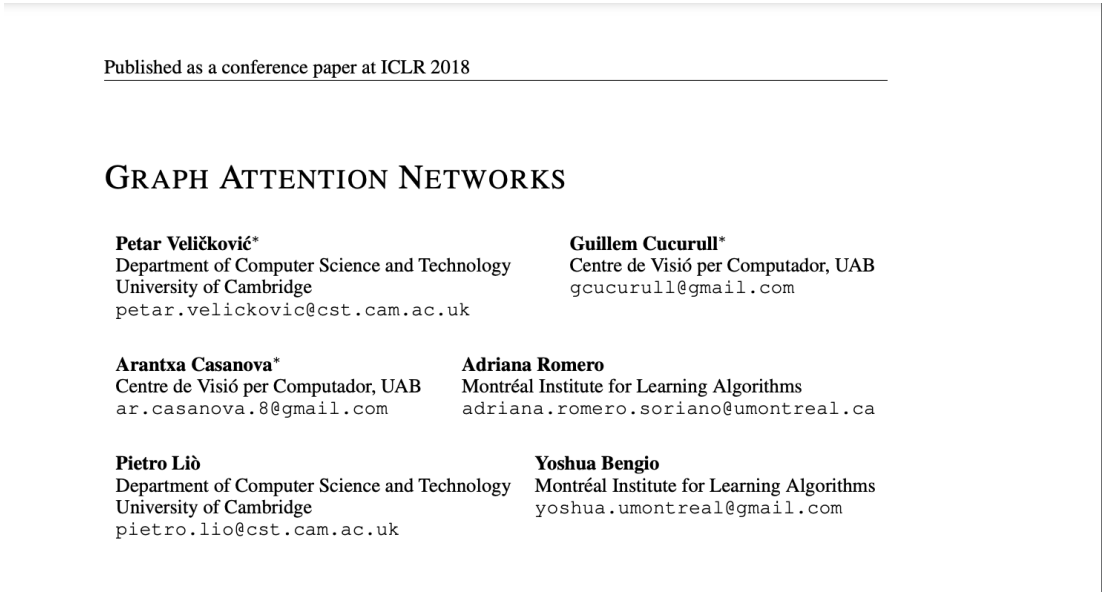
- **Heterogeneous Graph Learning** refers to any graph-based neural network learning on a graph with more than two vertex or edge types.
- Standard Message Passing GNNs (MP-GNNs) can not trivially be applied to heterogeneous graph data, as vertex and edge features from different types can not be processed by the same functions due to differences in feature type.
- To automatically convert a homogeneous GNN model to a heterogeneous GNN model by making use of:

```
1 | from torch_geometric.nn import to_hetero
```

Case Study: Heterogeneous Graph Learning

Graph Attention Network (GAT)

- The ”Graph Attention Networks” research paper was first published in October 30th, 2017.



Published as a conference paper at ICLR 2018

GRAPH ATTENTION NETWORKS

Petar Veličković*
Department of Computer Science and Technology
University of Cambridge
petar.velickovic@cst.cam.ac.uk

Guillem Cucurull*
Centre de Visió per Computador, UAB
gcucurull@gmail.com

Arantxa Casanova*
Centre de Visió per Computador, UAB
ar.casanova.8@gmail.com

Adriana Romero
Montréal Institute for Learning Algorithms
adriana.romero.soriano@umontreal.ca

Pietro Liò
Department of Computer Science and Technology
University of Cambridge
pietro.lio@cst.cam.ac.uk

Yoshua Bengio
Montréal Institute for Learning Algorithms
yoshua.umontreal@gmail.com

- The main idea is to compute the hidden representations of each vertex in the graph, by attending over its neighbors, following a self-attention strategy.

(1) GAT Layer: Linear Layer

- We begin by applying a shared linear transformation, parameterized by a weight matrix $\mathbf{W}$ on all of the vertex features $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_N}\}$:

$$\mathbf{W}\overrightarrow{h_i}, \mathbf{W}\overrightarrow{h_j}$$

- For recommendations systems, we apply two distinct shared linear transformations, parameterized by weight matrices $\mathbf{W}_{\text{user}}$ and $\mathbf{W}_{\text{item}}$ on the sampled sets of bipartite vertex features $\mathbf{h}_{\text{user}} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_N}\}$ and $\mathbf{h}_{\text{item}} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_M}\}$:

$$\mathbf{W}_{\text{user}} \overrightarrow{h}_{user_i}, \mathbf{W}_{\text{user}} \overrightarrow{h}_{user_j}$$

$$\mathbf{W}_{\text{item}} \overrightarrow{h}_{item_i}, \mathbf{W}_{\text{item}} \overrightarrow{h}_{item_j}$$

(1) Linear Layer: Code

- Initializing $\mathbf{W}_{\text{user}}$ and $\mathbf{W}_{\text{item}}$:

```
            self.lin_src = Linear(in_channels[0], heads * out_channels, False,
                                  weight_initializer='glorot')
            self.lin_dst = Linear(in_channels[1], heads * out_channels, False,
                                  weight_initializer='glorot')
```

- Applying linear transformation and reshaping into a tensor of size (Batch, Heads, Output Channels):

```
                # If the module is initialized as bipartite, transform source
                # and destination node features separately:
                assert self.lin_src is not None and self.lin_dst is not None
                x_src = self.lin_src(x).view(-1, H, C)
                x_dst = self.lin_dst(x).view(-1, H, C)
```

- Note: 'Src' and 'Dst' refer to Source and Destination vertex features; in our bipartite graph these are user and item vertex features.

(2) GAT Layer: Self-Attention

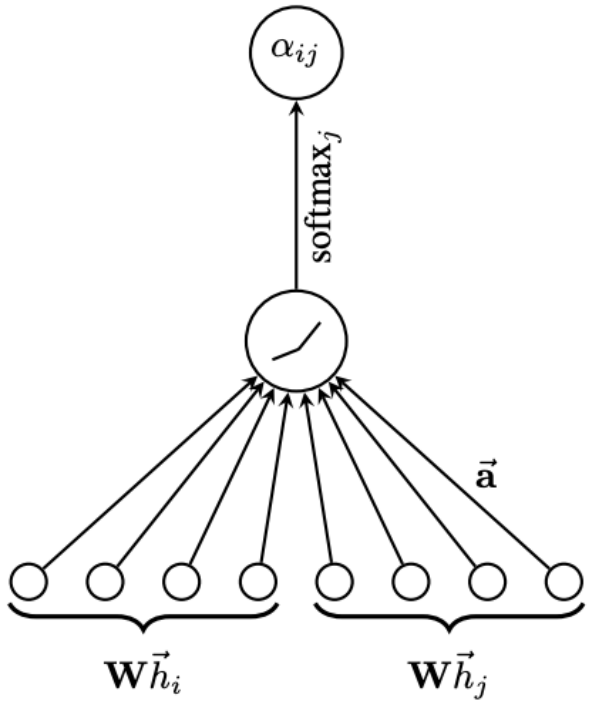
- Then, we perform self-attention on the vertices.

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}^{\top}[\mathbf{W}\overrightarrow{h_i} \| \mathbf{W}\overrightarrow{h_j}]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}^{\top}[\mathbf{W}\overrightarrow{h_i} \| \mathbf{W}\overrightarrow{h_j}]\right)\right)}$$

- The attention coefficients $\alpha_{i,j}$ indicate the importance of vertex j's features to vertex i.

(2) GAT Layer: Self-Attention

- The attention mechanism $\overrightarrow{\mathbf{a}}$ is a learnable parameter that is multiplied to the concatenation of the output of the previous linear layer.
- The output is then passed through a LeakyReLU activation with negative slope 0.2, and a softmax normalization across all neighbors.



- For recommendation systems, we have two different attention coefficients calculated in the same manner:

$$\alpha_{\text{user}_{i,j}} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}_{\text{user}}^{\top}[\mathbf{W}_{\text{user}}\overrightarrow{h}_{\text{user}_i} \| \mathbf{W}_{\text{user}}\overrightarrow{h}_{\text{user}_j}]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}_{\text{user}}^{\top}[\mathbf{W}_{\text{user}}\overrightarrow{h}_{\text{user}_i} \| \mathbf{W}_{\text{user}}\overrightarrow{h}_{\text{user}_j}]\right)\right)}$$

$$\alpha_{\text{item}_{i,j}} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}_{\text{item}}^{\top}[\mathbf{W}_{\text{item}}\overrightarrow{h}_{\text{item}_i} \| \mathbf{W}_{\text{item}}\overrightarrow{h}_{\text{item}_j}]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{a}}_{\text{item}}^{\top}[\mathbf{W}_{\text{item}}\overrightarrow{h}_{\text{item}_i} \| \mathbf{W}_{\text{item}}\overrightarrow{h}_{\text{item}_j}]\right)\right)}$$

(2) Self-Attention: Code

- Initializing $\overrightarrow{\mathbf{a}}_{\text{user}}$ and $\overrightarrow{\mathbf{a}}_{\text{item}}$:

```
        # The learnable parameters to compute attention coefficients:
        self.att_src = Parameter(torch.empty(1, heads, out_channels))
        self.att_dst = Parameter(torch.empty(1, heads, out_channels))
```

- These parameters are then passed through glorot intializer:

```
        glorot(self.att_src)
        glorot(self.att_dst)
```

- Applying self-attention mechanism on previous output:

```
        # Next, we compute node-level attention coefficients, both for source
        # and target nodes (if present):
        alpha_src = (x_src * self.att_src).sum(dim=-1)
        alpha_dst = None if x_dst is None else (x_dst * self.att_dst).sum(-1)
        alpha = (alpha_src, alpha_dst)
```

(2) Self-Attention: Code

- Sampled self-attention coefficients are collected according to edge index:

$\alpha_{\text{user}_{i,j}}$

```
    # Collect user-defined arguments:
    # (1) - Collect `alpha_j`:
    if isinstance(alpha, (tuple, list)):
        assert len(alpha) == 2
        _alpha_0, _alpha_1 = alpha[0], alpha[1]
        if isinstance(_alpha_0, Tensor):
            self._set_size(size, 0, _alpha_0)
            alpha_j = self._index_select(_alpha_0, edge_index_j)
        else:
            alpha_j = None
        if isinstance(_alpha_1, Tensor):
            self._set_size(size, 1, _alpha_1)
    elif isinstance(alpha, Tensor):
        self._set_size(size, j, alpha)
        alpha_j = self._index_select(alpha, edge_index_j)
```

$\alpha_{\text{item}_{i,j}}$

```
    # (2) - Collect `alpha_i`:
    if isinstance(alpha, (tuple, list)):
        assert len(alpha) == 2
        _alpha_0, _alpha_1 = alpha[0], alpha[1]
        if isinstance(_alpha_0, Tensor):
            self._set_size(size, 0, _alpha_0)
        if isinstance(_alpha_1, Tensor):
            self._set_size(size, 1, _alpha_1)
            alpha_i = self._index_select(_alpha_1, edge_index_i)
        else:
            alpha_i = None
    elif isinstance(alpha, Tensor):
        self._set_size(size, i, alpha)
        alpha_i = self._index_select(alpha, edge_index_i)
```

(2) Self-Attention: Code

- These sampled attention coefficients are summed $\alpha_{user_{i,j}} + \alpha_{item_{i,j}}$ to emulate concatenation operator $\|$

```
        # Given edge-level attention coefficients for source and target nodes,
        # we simply need to sum them up to "emulate" concatenation:
        alpha = alpha_j if alpha_i is None else alpha_j + alpha_i
```

- LeakyReLU activation with negative slope 0.2, and a softmax normalization across all neighbors:

```
        alpha = F.leaky_relu(alpha, self.negative_slope)
        alpha = softmax(alpha, index, ptr, dim_size)
```

(3) GAT Layer: Aggregation

- Once obtained, the normalized attention coefficients are used to compute a linear combination of the features corresponding to them, to serve as the final output features for every vertex:

$$\overrightarrow{h_i}' = \sum_{j \in \mathcal{N}_i} \alpha_{i,j} \mathbf{W} \overrightarrow{h_j}$$

- For recommendation systems, only user final outputs vertex features $\overrightarrow{h}'_{\text{user}_i}$ are propagated:

$$\overrightarrow{h}'_{\text{user}_i} = \sum_{j \in \mathcal{N}_i} (\alpha_{user_{i,j}} + \alpha_{item_{i,j}}) \mathbf{W}_{\text{user}} \overrightarrow{h}_{\text{user}_j}$$

(3) Aggregation: Code

- User neighborhood vertices are collected by sampled edge index:

$$\mathbf{W}_{\text{user}} \overrightarrow{h}_{\text{user}_j}$$

```
    # Collect user-defined arguments:
    # (1) - Collect 'x_j':
    if isinstance(x, (tuple, list)):
        assert len(x) == 2
        _x_0, _x_1 = x[0], x[1]
        if isinstance(_x_0, Tensor):
            self._set_size(size, 0, _x_0)
            x_j = self._index_select(_x_0, edge_index_j)
        else:
            x_j = None
        if isinstance(_x_1, Tensor):
            self._set_size(size, 1, _x_1)
    elif isinstance(x, Tensor):
        self._set_size(size, j, x)
        x_j = self._index_select(x, edge_index_j)
    else:
        x_j = None
```

(3) Aggregation: Code

- Self-attention coefficient $\alpha_{user_{i,j}} + \alpha_{item_{i,j}}$ is multiplied to the linear layer output $\mathbf{W}_{\text{user}} \overrightarrow{h}_{\text{user}_j}$

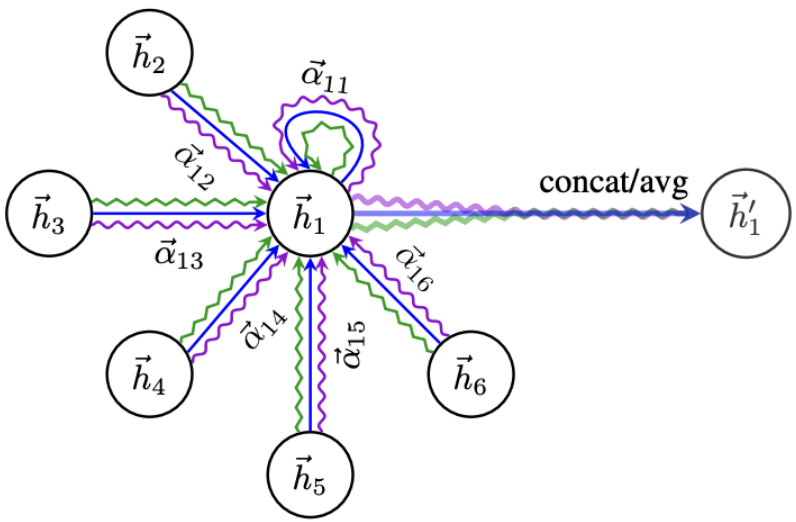
```
1 |     def message(self, x_j: Tensor, alpha: Tensor) -> Tensor:
2 |         return alpha.unsqueeze(-1) * x_j
```

- Lastly, the message is propagated to neighborhood vertices $\sum_{j \in \mathcal{N}_i}$

```
1 |     base_cls = aggr.Aggregation
2 |     aggrs = [
3 |         aggr for aggr in vars(aggr).values()
4 |         if isinstance(aggr, type) and issubclass(aggr, base_cls)
5 |     ]
6 |     aggr_dict = {
7 |         'add': aggr.SumAggregation,
8 |     }
9 |     return resolver(aggrs, aggr_dict, query, base_cls, None, *args, **kwargs)
```

(Optional 4) GAT Layer: Multi-Head Attention

- Multi-head attention is an optional addition to stabilize the learning process of self-attention:



$$\overrightarrow{h_i}' = \frac{1}{k} \sum_{k=1}^{k} \sum_{j \in \mathcal{N}_i} \alpha_{i,j} \mathbf{W} \overrightarrow{h_j}$$

- We simply repeat the process k times and average the messages.

(Optional 4) Multi-Head Attention: Case Study

- For recommendation systems, only user final outputs vertex features $\overrightarrow{h}'_{\text{user}_i}$ are propagated and averaged:

$$\overrightarrow{h}'_{user_i} = \frac{1}{k} \sum_{k=1}^{k} \sum_{j \in \mathcal{N}_i} (\alpha_{user_{i,j}} + \alpha_{item_{i,j}}) \mathbf{W}_{user} \overrightarrow{h}_{user_j}$$

(Optional 4) Multi-Head Attention: Code

- To establish multi-head attention, one must set the 'concat' to False and specify the amount of heads:

```
        if self.concat:
            out = out.view(-1, self.heads * self.out_channels)
        else:
            out = out.mean(dim=1)
```

GAT: Code

```
class Encoder(torch.nn.Module):

    def __init__(self, hidden_channels, heads):
        super().__init__()
        self.conv1 = GATConv(in_channels = (-1, -1), out_channels = hidden_channels, add_self_loops= False, heads = heads, concat=False)
        self.lin1 = Linear(-1, hidden_channels)
        self.conv2 = GATConv(in_channels = (-1, -1), out_channels = hidden_channels, add_self_loops= False, heads = heads, concat=False)
        self.lin2 = Linear(-1, hidden_channels)
        self.conv3 = GATConv(in_channels = (-1, -1), out_channels = hidden_channels, add_self_loops= False, heads = heads, concat=False)
        self.lin3 = Linear(-1, hidden_channels)

    def forward(self, batch_embedding_dict, batch_edge_index_dict):

        batch_embedding_dict = self.conv1(batch_embedding_dict, batch_edge_index_dict) + self.lin1(batch_embedding_dict)
        batch_embedding_dict = batch_embedding_dict.relu()
        batch_embedding_dict = self.conv2(batch_embedding_dict, batch_edge_index_dict) + self.lin2(batch_embedding_dict)
        batch_embedding_dict = batch_embedding_dict.relu()
        batch_embedding_dict = self.conv3(batch_embedding_dict, batch_edge_index_dict) + self.lin3(batch_embedding_dict)
        batch_embedding_dict = batch_embedding_dict.relu()

        return batch_embedding_dict
```

Inner Product Decoder

- Once we have our output vertex features for both user and item, we can decode them by performing a Hadamard multiplication and averaging across the N column dimension to get output $\overrightarrow{d}$:

$$\overrightarrow{d} = \frac{1}{N} \sum_{N=1}^{N} \mathbf{h}'_{\text{user}} \circ \mathbf{h}'_{\text{item}}$$

Inner Product Decoder: Code

```
1 class Decoder(torch.nn.Module):
2
3     def forward(self, batch_embedding_dict, edge_index):
4
5         user_embedding = batch_embedding_dict['user'][edge_index[0]]
6         item_embedding = batch_embedding_dict['item'][edge_index[1]]
7
8         return (user_embedding * item_embedding).sum(dim=-1)
```

- Finally, now that we have our output vector $\overrightarrow{d}$ we can calculate a mean squared error (MSE) with our target edge label vector $\overrightarrow{t}$:

$$e = \frac{1}{n} \sum_{i=1}^{n} (\overrightarrow{t} - \overrightarrow{d})^2$$

- In this manner, we can approximate edge label values to predict the existence of edges between users and items, or vice versa.

Loss Function: Code

```
1 |         out = model(batch.x_dict,
2 |                     batch.edge_index_dict,
3 |                     batch['user','item'].edge_label_index)
4 |         y = batch['user', 'item'].edge_label
5 |
6 |         loss = torch.nn.functional.mse_loss(out, y.float())
```

Maximum Inner Product Search (MIPS)

- Given a collection of "database" vectors,
  $D = \{\overrightarrow{x}_0, \overrightarrow{x}_1, ..., \overrightarrow{x}_d\} \in S \subset R^d$ and a query $\overrightarrow{q} \in R^d$,
  Maximum Inner Product Search attempts to find a data vector $\overrightarrow{x_*}$ maximizing the inner product with the query:

$$\overrightarrow{x_*} = argmax_{\overrightarrow{x} \in S} \overrightarrow{q}^T \overrightarrow{x}$$

- There are many variations of MIPS, one of them being **Nearest Neighbor MIPS** attempting to find a data vector $\overrightarrow{x_*}$ by minimizing over a distance function ρ:

$$\overrightarrow{x_*} = argmin_{\overrightarrow{x} \in S}\ \rho(\overrightarrow{q}^T \overrightarrow{x})$$

Intuition: A neighbor of my neighbor is likely to be my neighbor.

K-Nearest Neighbor MIPS: Pseudocode

Algorithm KNN-MIPS

Input: a KNN graph $G = (D, E)$; a query point $\overrightarrow{q}$; the number of required nearest neighbors K; the number of random restarts R; distance function ρ; the number of greedy steps T; the number of expansions E.

$S \leftarrow \{\}$ {Set of MIPS candidates}
$U \leftarrow \{\}$ {Set of to be expanded candidates}
for $r = 1, ..., R$ **do**
  $\overrightarrow{x}_0$ a point drawn randomly from a uniform distribution over D
  for $t = 1, ..., T$ **do**
    $\overrightarrow{x}_t = \text{argmin}_{\overrightarrow{x} \in N(\overrightarrow{x}_{t-1}, E, G)} \rho(\overrightarrow{q}^T \overrightarrow{x})$
    $S \leftarrow N(\overrightarrow{x}_{t-1}, E, G)$ {N is a function that returns neighbors}
    $U \leftarrow \{\rho(\overrightarrow{q}^T \overrightarrow{x}) : \overrightarrow{x} \in N(\overrightarrow{x}_{t-1}, E, G)\}$.
  end for
end for

Output: Sort U, pick the first K elements, and return the corresponding indices in S.

K-Nearest Neighbor MIPS: Visualization

- Suppose greedy steps $T = 2$, neighbors $K = 1$ and expansions $E = 3$

K-Nearest Neighbors MIPS: Code

- At inference, we first calculate the embedding, or output features, for item vertices $\mathbf{h}'_{\text{item}}$

```
    positive_item_embeddings = []
    for batch in item_loader:
        batch = batch.to(device)
        batch_embedding_item = model.encoder(batch.x_dict,
                                             batch.edge_index_dict)['item']
        batch_embedding_item_positive = batch_embedding_item[:batch['item'].batch_size]

        positive_item_embeddings.append(batch_embedding_item_positive)
    horizontal_stack_positive_item_embeddings = torch.cat(positive_item_embeddings, dim = 0)
    del positive_item_embeddings

    max_inner_product_search = MIPSKNNIndex(horizontal_stack_positive_item_embeddings)
```

- Then, we pass these item embedding vertices $\mathbf{h}'_{\text{item}}$ as our dataset vectors $D = \{\overrightarrow{x}_0, \overrightarrow{x}_1, ..., \overrightarrow{x}_d\}$
- Note: positive embedding are passed as dataset vectors if we are utilizing negative sampling.

K-Nearest Neighbors MIPS: Code

- After passing dataset vectors, we calculate the embedding, or output features, for user vertices $\mathbf{h}'_{\text{user}}$

```
    for batch in user_loader:
        batch = batch.to(device)
        batch_embedding_user = model.encoder(batch.x_dict,
                                             batch.edge_index_dict)['user']
        batch_embedding_user_positive = batch_embedding_user[:batch['user'].batch_size]

        edge_index = test_edge_label_index.sparse_narrow(
            dim=0,
            start=number_processed,
            length=batch_embedding_user_positive.size(0))

        exclude_links = test_exclude_links.sparse_narrow(
            dim=0,
            start=number_processed,
            length=batch_embedding_user_positive.size(0))

        number_processed += batch_embedding_item_positive.size(0)

        _, pred_index_mat = max_inner_product_search.search(batch_embedding_user_positive, K, exclude_links)
```

- We pass user embedding vertices $\mathbf{h}'_{\text{user}}$ as query vectors $Q = \{\vec{q}_0, \vec{q}_1, ..., \vec{q}_d\}$, K neighbors and links to exclude.

Recall @ K

- Recall at K calculates the fraction of relevant items among the top-K items recommended or retrieved.

$$\text{Recall@K} = \frac{\text{Number of relevant items retrieved in top-K}}{\text{Total number of relevant items}}$$

- A high recall at K indicates that the system is good at retrieving relevant items in the top-K list.

Recall @ K: Code

- We first initialize the class method with top-K amount of retrievals:

```
    recall = LinkPredRecall(k=K).to(device)
```

- Then, we pass our matrix of predicted indices from our KNN-MIPS and corresponding edge index with excluded links:

```
        recall.update(pred_index_mat, edge_index)

    recall = recall.compute()
```

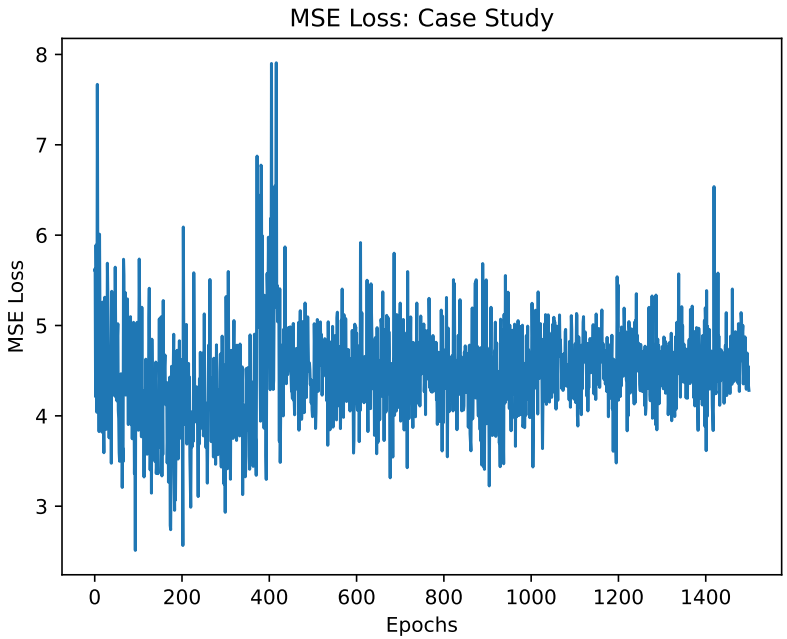
Case Study: Simplified Movie Recommendation System

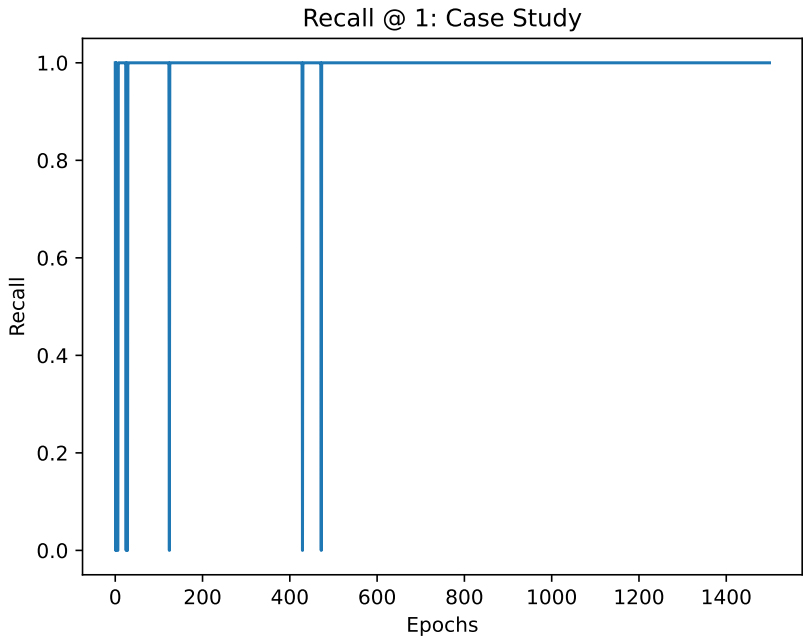
- Finally, we'll execute our GAT model with KNN-MIPS to assess whether recommending 'The Avengers' to Tim is feasible.

Case Study: Hyperparameters

```
1  | ############### Hyperparameters ###############
2  |
3  | SEED = 123
4  | SPLIT = 7/8
5  | BATCH_SIZE = 1
6  | NEIGHBORS = [1]
7  | NEG_SAMPLING = 0
8  |
9  | EPOCHS = 1500
10 | LR = 1e-02
11 | LATENT_DIM = 12
12 | HEADS = 2
13 | K = 1
14 |
15 | ############### Hyperparameters ###############
```

Case Study: MSE Loss & Recall @ 1





- Notice that after the first 500 epochs, the recommendation system converges to an optimal recommendation.

References

- PyG Documentation
- PyG GAT Documentation
- PyG GAT Source Code
- KNN-MIPS Research Paper
- KNN-MIPS Lecture
- Code: Case Study - Simplified Movie Recommendation System