



SESSION-BASED RECOMMENDATION WITH GRAPH NEURAL NETWORKS

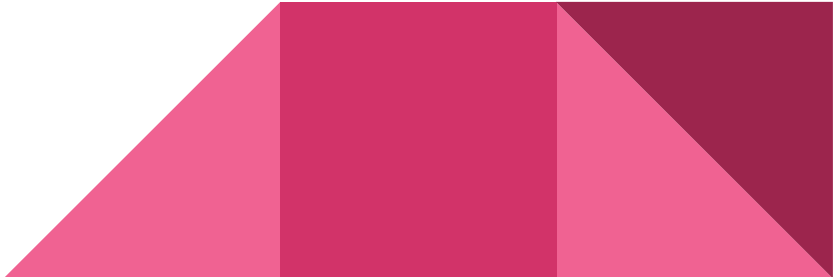
Submitted by:

Ipsita Patnaik
Pradyumna Kumar Sahoo
Sumit Kumar

MODELLING IN OPERATIONAL MANAGEMENT

**DEPARTMENT OF DATA SCIENCE AND ANALYTICS
CENTRAL UNIVERSITY OF RAJASTHAN**

Agenda

1. Introduction
 2. Session-based Recommender Systems
 3. Previous Work
 4. Recurrent Neural Net
 5. Gated Recurrent Unit (GRU)
 6. Recommender engine
 7. Datasets
 8. Metrics & Evaluations
 9. Results
 10. Future Works
 11. References
- 

Introduction

The problem of session-based recommendation aims to predict user actions based on anonymous sessions. We implemented a novel method, i.e. Session-based Recommendation with Graph Neural Networks (SR-GNN).

In this model, session sequences are modeled as graph structured data. Based on the session graph, GNN can capture complex transitions of items, which are difficult to be revealed by previous conventional sequential methods.

Each session is then represented as the composition of the global preference and the current interest of that session using an attention network.



Session based recommender systems

- Session-based recommendation is the task of predicting the next item to recommend when the only available information consists of anonymous behavior sequences ie, it aims to predict which item a user will click next, solely based on the user's current sequential session data without accessing to the long-term preference profile.
- In many services, user identification may be unknown and only the user behavior history during an ongoing session is available. Thereby, it is of great importance to model limited behavior in one session and generate the recommendation accordingly.



Previous Works and Shortcomings

- **Matrix factorisation (2001) -**

Not very suitable for the session-based recommendation, because the user preference is only provided by some positive clicks. The item-based neighborhood methods (Sarwar et al. 2001) is a natural solution, in which item similarities are calculated on the co-occurrence in the same session. These methods have difficulty in considering the sequential order of items and generate prediction merely based on the last click.

- **Markov chains : (2002,2010) -**

Predicts the user's next behavior based on the previous one. With a strong independence assumption, independent combinations of the past components confine the prediction accuracy.

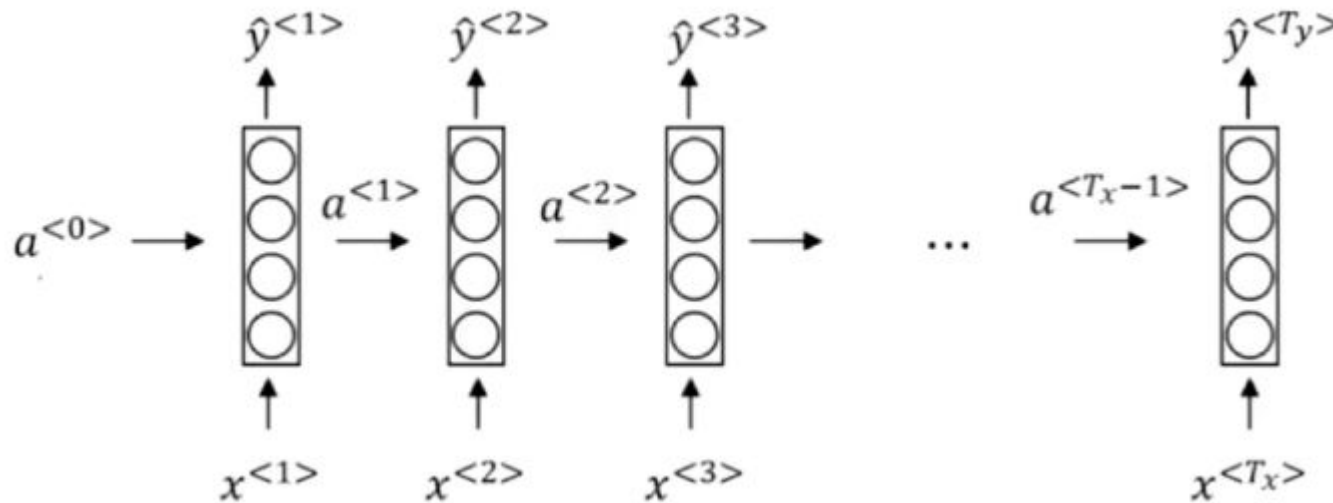
Previous works (cont.)

- **Recurrent Neural Networks (RNN) : (2016-18) -**
 1. Usually, the hidden vectors of the RNN methods are treated as the user representations such that recommendations can be then generated based on these representations but **user behavior implicated in session clicks is often limited**. It is thus difficult to accurately estimate the representation of each user from each session.
 2. Patterns of item transitions are important and can be used as a local factor, these methods always model single-way transitions between consecutive items and neglect the transitions among the contexts, i.e. other items in the session. Thus, **complex transitions among distant items are often overlooked by these methods**.

Recurrent Neural Network (RNN)

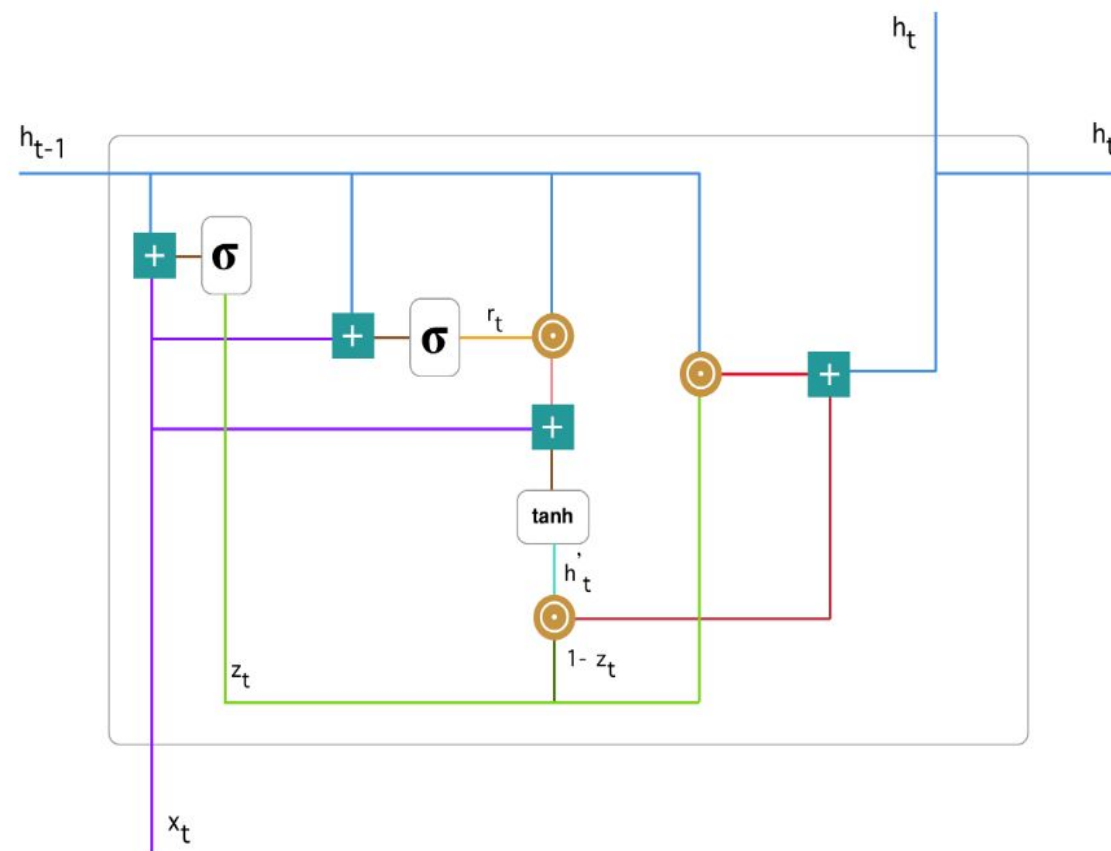
A **recurrent neural network (RNN)** is a class of **artificial neural networks** where connections between nodes form a **directed graph** along a temporal sequence. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequences of inputs.

In RNN, each layer represented by block with four nodes (as shown in the figure) actually have two sources of inputs: the input for that time period ($x_1, x_2, x_3 \dots$) but it also has this context or state vector called ($a_0, a_1, a_2 \dots$). The output is produced by joining vector A together the inputs X . At the same time an updated vector A is produced which being fed to the next layer. So in a way the same model will run again on the output that was produced previously by itself.



Gated Recurrent Unit (GRU)

- GRUs are improved version of standard recurrent neural network which aims to solve the **vanishing gradient problem**.
- To solve the vanishing gradient problem of a standard RNN, GRU uses, **update gate and reset gate**, which decide what information should be passed to the output.
- It combines the forget and input into a single update gate and merges the cell state and hidden state. (This is simpler than LSTM).
- The update gate controls information that flows into memory, and the reset gate controls the information that flows out of memory. The update gate and reset gate are two vectors that decide which information will get passed on to the output. They can be trained to keep information from the past or remove information that is irrelevant to the prediction



Gated Recurrent Unit

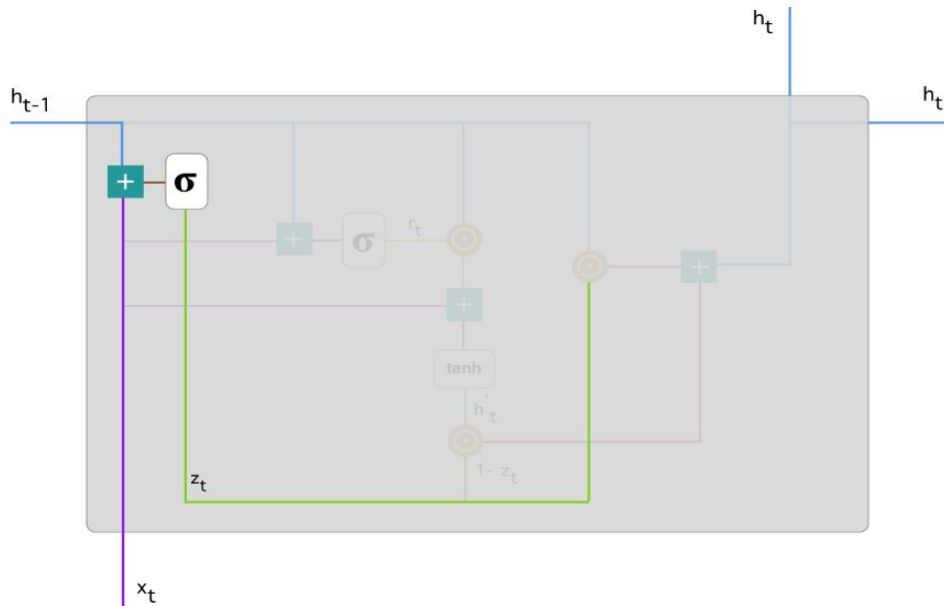
GRU Components

- Update Gate

The update gate z_t for time step t is calculated using the formula :

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.

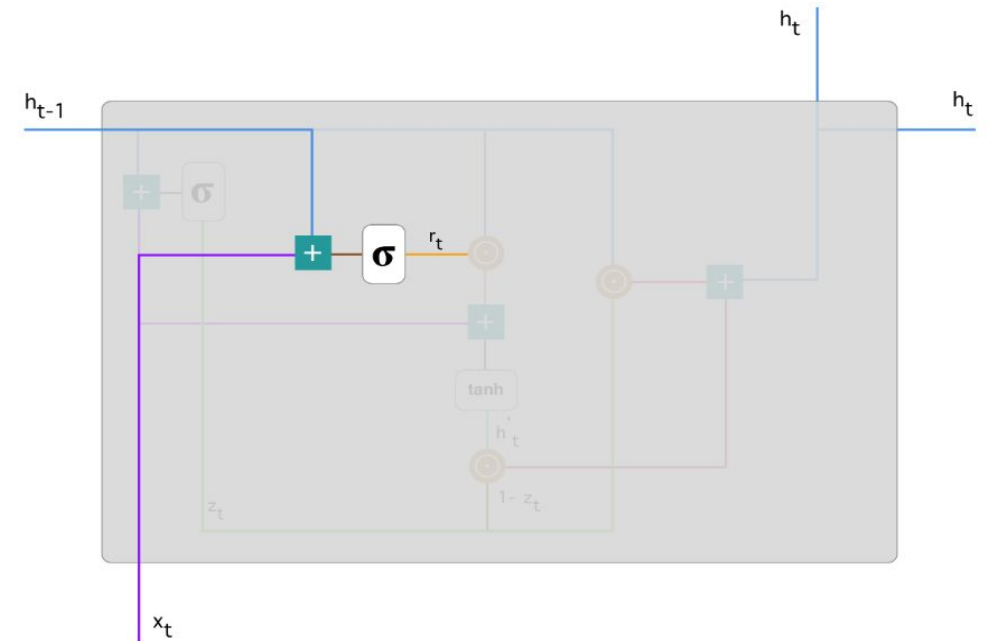


- Reset Gate

The reset gate r_t for time step t is calculated using the formula :

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

The reset gate is used from the model to decide how much of the past information to forget.

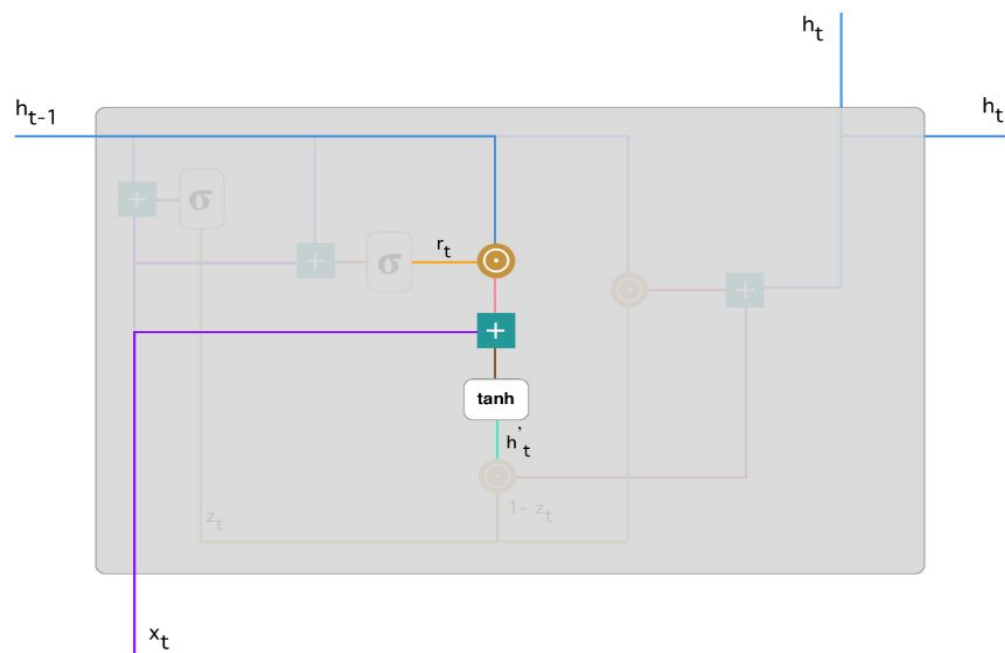


GRU Components (contd.)

- Current Memory Content

A new memory content which uses the reset gate to store the relevant information from the past is calculated using the formula :

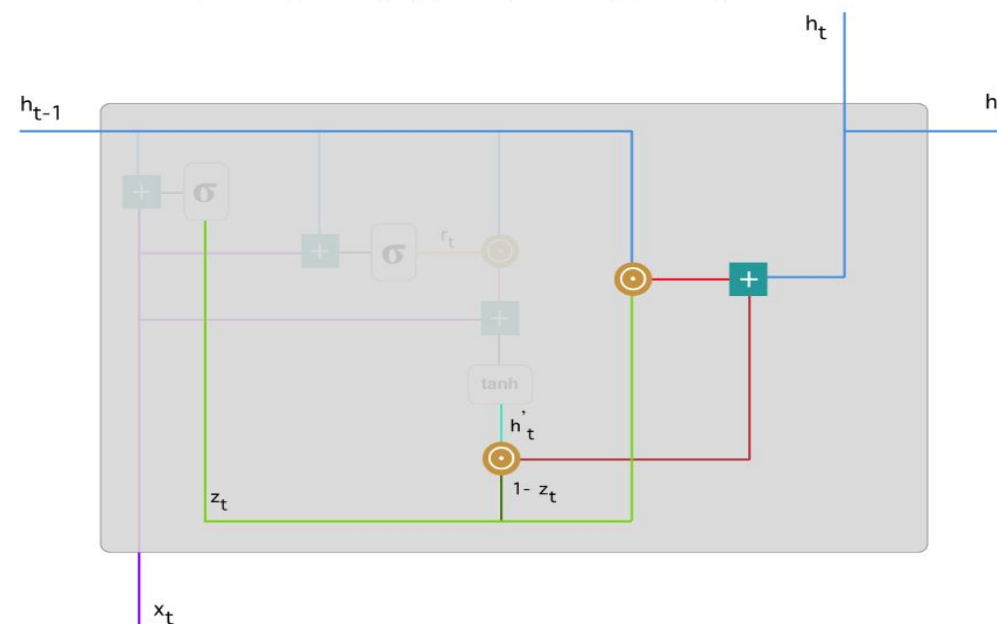
$$h'_t = \tanh(Wx_t + r_t \odot Uh_{t-1})$$



- Final Memory at Current Time Step

At last, h_t is calculated which holds information for the current unit and passes it down to the network, which is done using the update gate. It determines what to collect from the current memory content, h'_t and what from the previous steps $h_{(t-1)}$ that is done as follows:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$$



What is Attention?

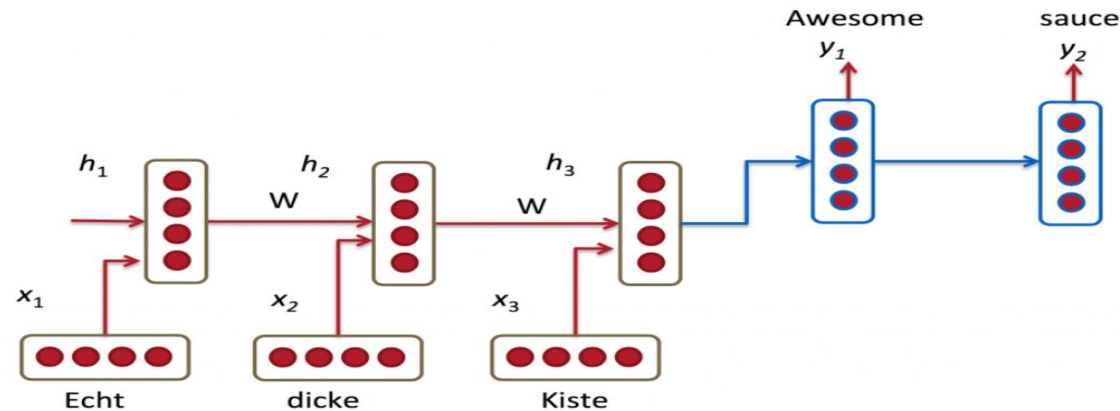
Attention can be broadly interpreted as a vector of importance weights: in order to predict or infer one element, such as a pixel in an image or a word in a sentence, we estimate using the attention vector how strongly it is correlated with (or “*attends to*” as you may have read in many papers) other elements and take the sum of their values weighted by the attention vector as the approximation of the target.



What is Attention in Neural Network?

Suppose we want to translate a sentence in German to English. Traditionally, we map the meaning of a sentence into a fixed-length vector representation and then generate a translation based on that vector.

Most systems work by *encoding* the source sentence (e.g. a German sentence) into a vector using a Recurrent Neural Network, and then *decoding* an English sentence based on that vector, also using a RNN.



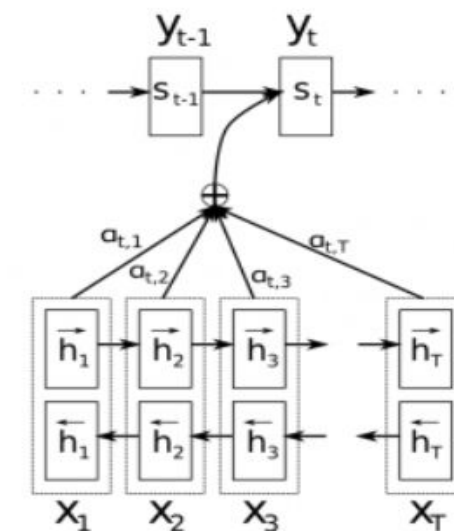
In the picture above, “Echt”, “Dicke” and “Kiste” words are fed into an encoder, and the decoder starts producing a translated sentence after a special signal is generated. The decoder keeps generating words until a special end of sentence token is produced. Here, the \mathbf{h} vectors represent the internal state of the encoder. The decoder is supposed to generate a translation solely based on the last hidden state (h_3 above) from the encoder. This h_3 vector must encode everything we need to know about the source sentence. So, it must fully capture its meaning. However, it is unreasonable to assume that we can encode all information about a potentially very long sentence into a single vector and then have the decoder produce a good translation based on only that. So we use attention mechanism to solve this problem.

With an attention mechanism we no longer try to encode the full source sentence into a fixed-length vector. Rather, we allow the decoder to “attend” to different parts of the source sentence at each step of the output generation.

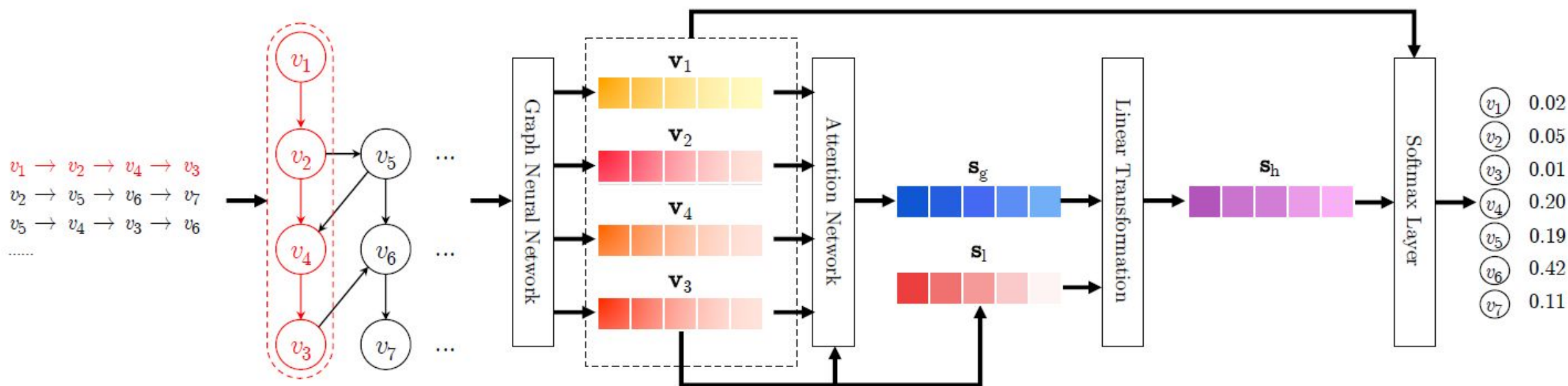
The y 's are our translated words produced by the decoder, and the x 's are our source sentence words.

As shown in the image, each decoder output word y_t now depends on a weighted combination of all the input states, not just the last state.

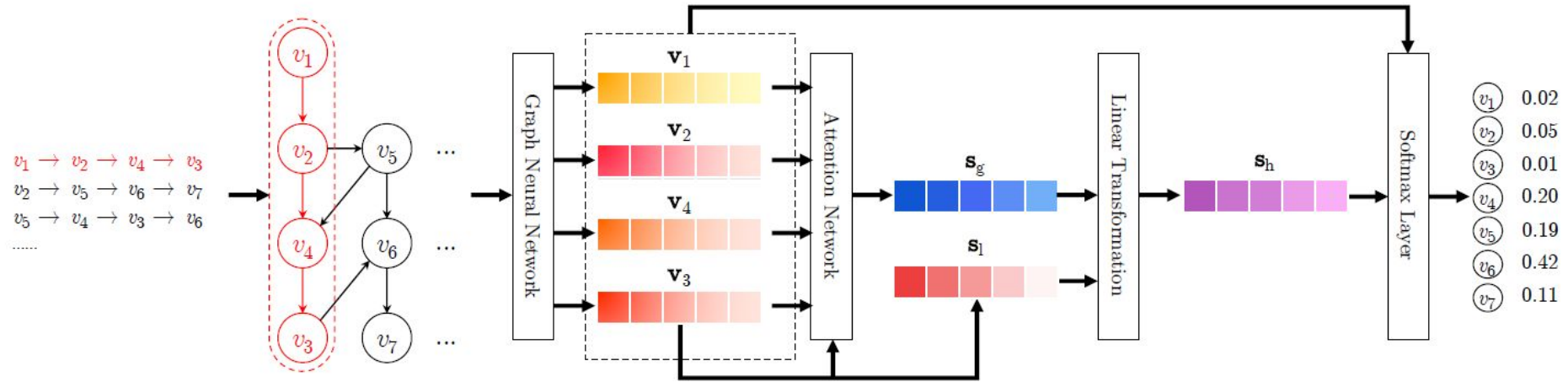
The a 's are weights that define in how much of each input state should be considered for each output. So, if $a_{\{3,2\}}$ is a large number, this would mean that the decoder pays a lot of attention to the second state in the source sentence while producing the third word. The a 's are typically normalized to sum to 1 (so they are a distribution over the input states).



Our Recommender Engine Architecture



Our objective



- Let $V = \{v_1, v_2, \dots, v_m\}$ denote the set consisting of all unique items involved in all the sessions.
- An anonymous session sequence s is represented by a list of clicked item of the user within the session s ordered by timestamps

$$s = [v_{s,1}, v_{s,2}, \dots, v_{s,n}] \quad v_{s,i} \in V$$

- The goal of the session-based recommendation is to predict the next click, i.e. the sequence label, $v_{s,n+1}$ for the sessions.
- Under a session-based recommendation model, for the session s , we output probabilities \hat{y} for all possible items, where an element value of vector \hat{y} is the recommendation score of the corresponding item.
- The items with top-K values in \hat{y} will be the candidate items for recommendation.

Mechanism

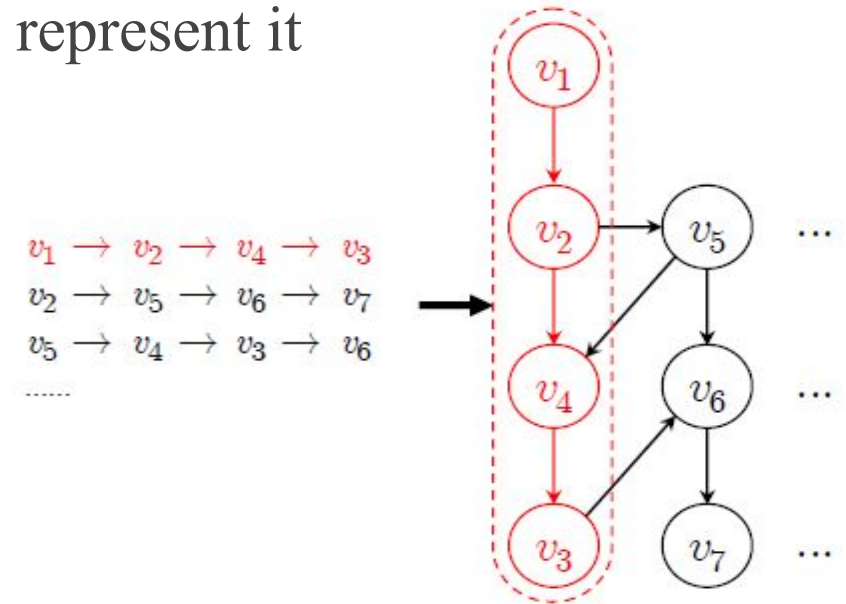
Step-1 : At first, **all** session sequences are modeled as directed session graphs, where **each** session sequence can be treated as a subgraph.

For any session sequence 's' being a graph on its own, we can represent it as :

$$G_s = [V_s, E_s]$$

Node = an item $V_{(s,i)} \in V$

Edge= $(V_{(s,i-1)}, V_{(s,i)})$ means an item user clicked after visiting $V_{(s,i-1)}$



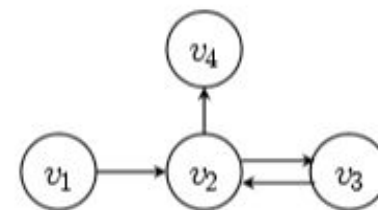
Example of session graph construction:

Weight = Occurrence of edge/out-degree of the start node

we are normalising the weights to avoid effects of several items that appear in the sequence repeatedly

Consider a session $s = [v_1, v_2, v_3, v_2, v_4]$

We create \mathbf{A}_s (our input to GNN)



	Outgoing edges				Incoming edges			
	1	2	3	4	1	2	3	4
1	0	1	0	0	0	0	0	0
2	0	0	1/2	1/2	1/2	0	1/2	0
3	0	1	0	0	0	1	0	0
4	0	0	0	0	0	1	0	0

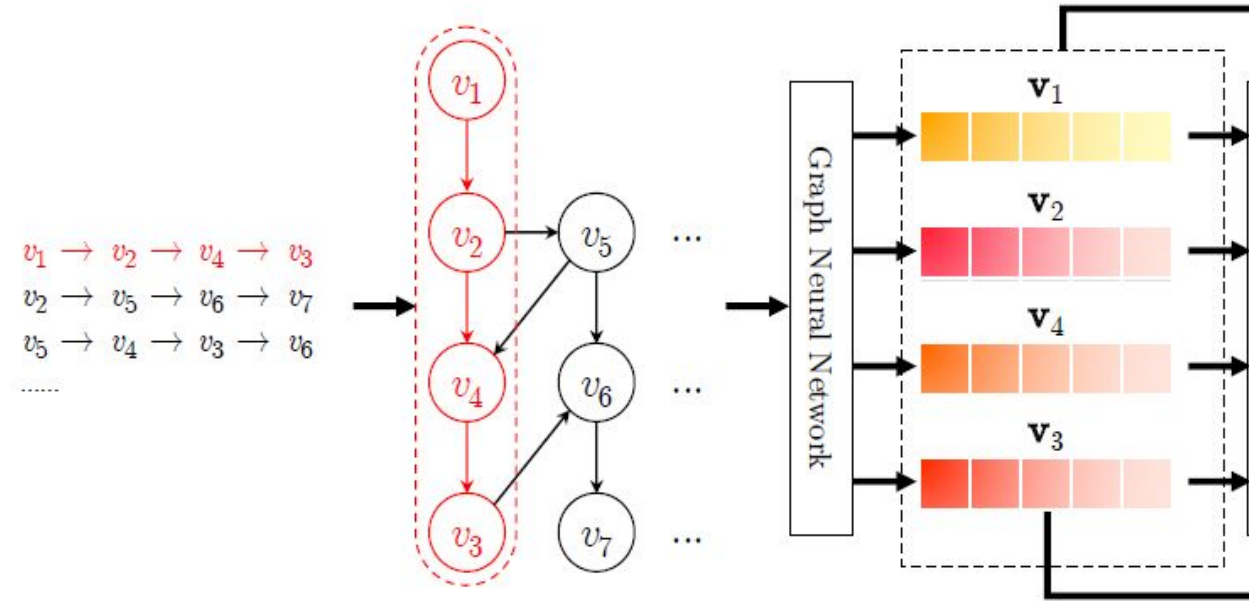
\mathbf{A}_s = concatenation of two adjacency matrices $\mathbf{A}_s^{(\text{out})}$ and $\mathbf{A}_s^{(\text{in})}$

$\mathbf{A}_s^{(\text{out})}$ = weighted connections of outgoing edges in the session graph

$\mathbf{A}_s^{(\text{in})}$ = weighted connections of incoming edges in the session graph

Mechanism (cont.)

Step 2.1 : (Item Embedding) Then, each session graph is proceeded successively and the latent vectors for all nodes involved in each session graph is obtained through **gated graph neural networks**.

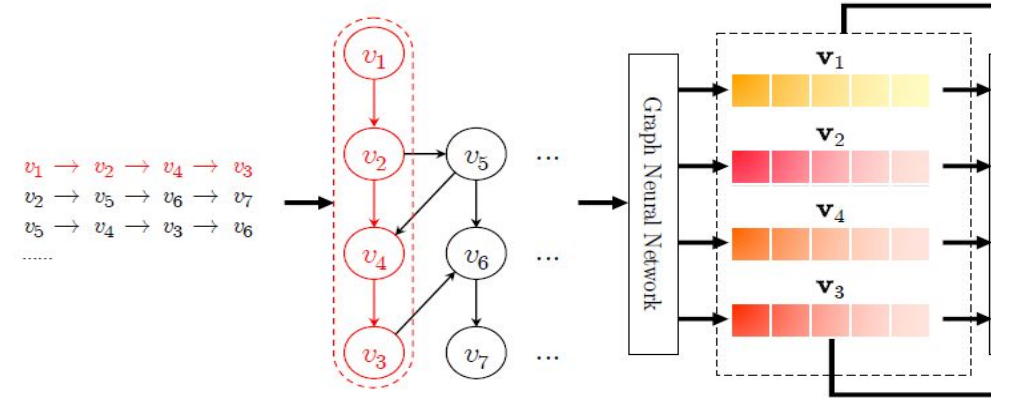


But **HOW?**

- We embed every item $\mathbf{v} \in V$ into an unified embedding space.
- Thus associated node vector of \mathbf{v} i.e. $\mathbf{v} \in \mathbf{R}^d$ indicates the latent vector of item \mathbf{v} learned via GNN where d is the dimensionality (d is 100 in our model.)
- Hence, based on node vectors, **each session** s (here, $[v_1, v_2, v_4, v_3]$) can be represented by an embedding vector \mathbf{s} , which is composed of node vectors used in that graph.

But how are the learning of latent vectors of items taking place by GRU-GNN?

Formally, for the node $v_{(s,i)}$ of graph G_s the learning model is given as follows:



$$\mathbf{a}_{s,i}^t = \mathbf{A}_{s,i}: [\mathbf{v}_1^{t-1}, \dots, \mathbf{v}_n^{t-1}]^\top \mathbf{H} + \mathbf{b},$$

$$\mathbf{z}_{s,i}^t = \sigma(\mathbf{W}_z \mathbf{a}_{s,i}^t + \mathbf{U}_z \mathbf{v}_i^{t-1}),$$

$$\mathbf{r}_{s,i}^t = \sigma(\mathbf{W}_r \mathbf{a}_{s,i}^t + \mathbf{U}_r \mathbf{v}_i^{t-1}),$$

$$\tilde{\mathbf{v}}_i^t = \tanh(\mathbf{W}_o \mathbf{a}_{s,i}^t + \mathbf{U}_o (\mathbf{r}_{s,i}^t \odot \mathbf{v}_i^{t-1})),$$

$$\mathbf{v}_i^t = (1 - \mathbf{z}_{s,i}^t) \odot \mathbf{v}_i^{t-1} + \mathbf{z}_{s,i}^t \odot \tilde{\mathbf{v}}_i^t,$$

- $\mathbf{H} \in \mathbf{R}^{d \times 2d}$ controls the weight
- \mathbf{b} = bias (vector)
- $\mathbf{z}_{s,i}$ = update gate with weights \mathbf{W}_z & \mathbf{U}_z
- $\mathbf{r}_{s,i}$ = reset gate with weights \mathbf{W}_r & \mathbf{U}_r
- $[\mathbf{v}_1^{t-1}, \dots, \mathbf{v}_n^{t-1}]$ = list of node vectors (i.e. upto $\mathbf{v}_{s,n}$) in session s till $t-1^{\text{th}}$ time. We want to predict for $\mathbf{v}_{s,n+1}$
- $\sigma(\cdot)$ = sigmoid function
- \odot = element-wise multiplication operator.
- $\mathbf{v}_i \in \mathbf{R}^d \Rightarrow$ latent vector of node $v_{s,i}$.
- $\mathbf{A}_s \in \mathbf{R}^{n \times 2n}$ = The connection matrix, determines how nodes in the graph communicate with each other
- $\mathbf{A}_{s,i} \in \mathbf{R}^{1 \times 2n}$ = the two columns of blocks in \mathbf{A}_s corresponding to node $\mathbf{v}_{s,i}$

What's all that jargon??

1. **Eq(1)** is used for information propagation between different nodes, under restrictions given by the matrix \mathbf{A}_s . Specifically, it extracts the latent vectors of neighborhoods (previous clicks) and feeds them as input into the graph neural network.
2. **Eq(2)** update gate, decide what information to be preserved.
3. **Eq(3)** reset decide what information to be discarded.
4. **Eq(4)** constructs the candidate state from previous state, the current state, and the reset gate
5. **Eq(5)** gives the final state is then the combination of the previous hidden state and the candidate state, under the control of the update gate.
6. After updating all nodes in session graphs until convergence, we can obtain the final node vectors.

$$\mathbf{a}_{s,i}^t = \mathbf{A}_{s,i} : [\mathbf{v}_1^{t-1}, \dots, \mathbf{v}_n^{t-1}]^\top \mathbf{H} + \mathbf{b}, \quad (1)$$

$$\mathbf{z}_{s,i}^t = \sigma(\mathbf{W}_z \mathbf{a}_{s,i}^t + \mathbf{U}_z \mathbf{v}_i^{t-1}), \quad (2)$$

$$\mathbf{r}_{s,i}^t = \sigma(\mathbf{W}_r \mathbf{a}_{s,i}^t + \mathbf{U}_r \mathbf{v}_i^{t-1}), \quad (3)$$

$$\tilde{\mathbf{v}}_i^t = \tanh(\mathbf{W}_o \mathbf{a}_{s,i}^t + \mathbf{U}_o (\mathbf{r}_{s,i}^t \odot \mathbf{v}_i^{t-1})), \quad (4)$$

$$\mathbf{v}_i^t = (1 - \mathbf{z}_{s,i}^t) \odot \mathbf{v}_i^{t-1} + \mathbf{z}_{s,i}^t \odot \tilde{\mathbf{v}}_i^t, \quad (5)$$

Mechanism (cont.)

Step - 2.2 : (Session Embedding) After that, we represent each session as a composition of the global preference and the current interest of the user in that session, where these global and local session embedding vectors are both composed by the latent vectors of nodes.

- Then, to represent each session as an embedding vector $s \in \mathbb{R}^d$, we first consider the local embeddings \mathbf{l} of session s . For session $s = [v_{s,1}, v_{s,2}, \dots, v_{s,n}]$, the local embedding can be simply defined as v_n of the last-clicked item $v_{s,n}$, i.e. $\mathbf{s}_l = \mathbf{v}_n$.
- Then, we consider the global embedding s_g of the session graph G_s by aggregating all node vectors.
- Consider information in the seem bedding may have different levels of priority, we further adopt the soft-attention mechanism to better represent the global session preference:

$$\alpha_i = \mathbf{q}^\top \sigma(\mathbf{W}_1 \mathbf{v}_n + \mathbf{W}_2 \mathbf{v}_i + \mathbf{c}),$$

$$\mathbf{s}_g = \sum_{i=1}^n \alpha_i \mathbf{v}_i,$$

$$\mathbf{q} \in \mathbb{R}^d$$

$\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d \times d}$ control the weights of item embedding vectors.

- Finally, we compute the hybrid embedding s_h by taking linear transformation over the concatenation of the local and global embedding vectors:

$$\mathbf{s}_h = \mathbf{W}_3 [\mathbf{s}_l; \mathbf{s}_g]$$

$\mathbf{W}_3 \in \mathbb{R}^{d \times 2d}$ compresses two combined embedding vectors into the latent space \mathbb{R}^d .



Making Recommendation and Model Training

After obtained the embedding of each session, we compute the score \hat{z}_i for each candidate item $v_i \in V$ by multiplying its embedding v_i by session representation s_h defined as : $\hat{z}_i = s_h^T v_i$

Then we apply a softmax function to get the output vector of the model \hat{y} : $\hat{y} = \text{softmax}(\hat{z})$,

where $\hat{z} \in \mathbb{R}^m$ denotes the recommendation scores over all candidate items and $\hat{y} \in \mathbb{R}^m$ denotes the probabilities of nodes appearing to be the next click in session s .

For each session graph, the loss function is defined as the cross-entropy of the prediction and the ground truth, written as follows :

$$\mathcal{L}(\hat{y}) = - \sum_{i=1}^m y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

where y is the ground truth.



Crux of the program:

1. Constructing Session Graphs
2. Learning Item Embeddings on Session Graphs
3. Generating Session Embeddings
4. Making Recommendation and Model Training



Datasets

We have used two real-world representative datasets.

1. Yoochoose (<http://2015.recsyschallenge.com/challenge.html>)

The Yoochoose dataset is obtained from the RecSys Challenge 2015, which contains a stream of user clicks on an e-commerce website within 6 months.

2. Diginetica (<http://cikm2016.cs.iupui.edu/cikm-cup>)

The Diginetica dataset comes from CIKM Cup 2016, where only its transactional data is used.



Original Datasets

Diginetica

```
[12] df.head()
```


```
↗
```

	session_id	user_id	item_id	timeframe	eventdate
0	1	NaN	81766	526309	2016-05-09
1	1	NaN	31331	1031018	2016-05-09
2	1	NaN	32118	243569	2016-05-09
3	1	NaN	9654	75848	2016-05-09
4	1	NaN	32627	1112408	2016-05-09

Original Datasets

YooChoose

```
[19] df2
```



	session_id	timestamp	item_id	category
0	1	2014-04-07T10:51:09.277Z	214536502	0
1	1	2014-04-07T10:54:09.868Z	214536500	0
2	1	2014-04-07T10:54:46.998Z	214536506	0
3	1	2014-04-07T10:57:00.306Z	214577561	0
4	2	2014-04-07T13:56:37.614Z	214662742	0

Transaction Dataset

[1, 2]->3

[8, 9]->2

[10, 11, 11]->5

[10, 11]->7

[15, 16, 17]->9

[15, 16]->9

[22, 22, 23, 23, 23, 22, 23]->11

[22, 22, 23, 23, 23, 22]->11

[22, 22, 23, 23, 23]->11

[22, 22, 23, 23]->13

[22, 22, 23]->14

[22, 22]->18

[24, 25, 26]->17

[24, 25]->16

[1, 28, 3]->19

[1, 28]->20

[31, 32, 31, 31, 31]->21

[31, 32, 31, 31]->23

[31, 32, 31]->23

[31, 32]->23

[33, 33]->22

[34, 34, 34, 34, 34]->27

[34, 34, 34, 34]->26

[34, 34, 34]->25

[34, 34]->2

[12, 13, 12, 13, 35, 35, 12, 13]->3

[12, 13, 12, 13, 35, 35, 12]->38

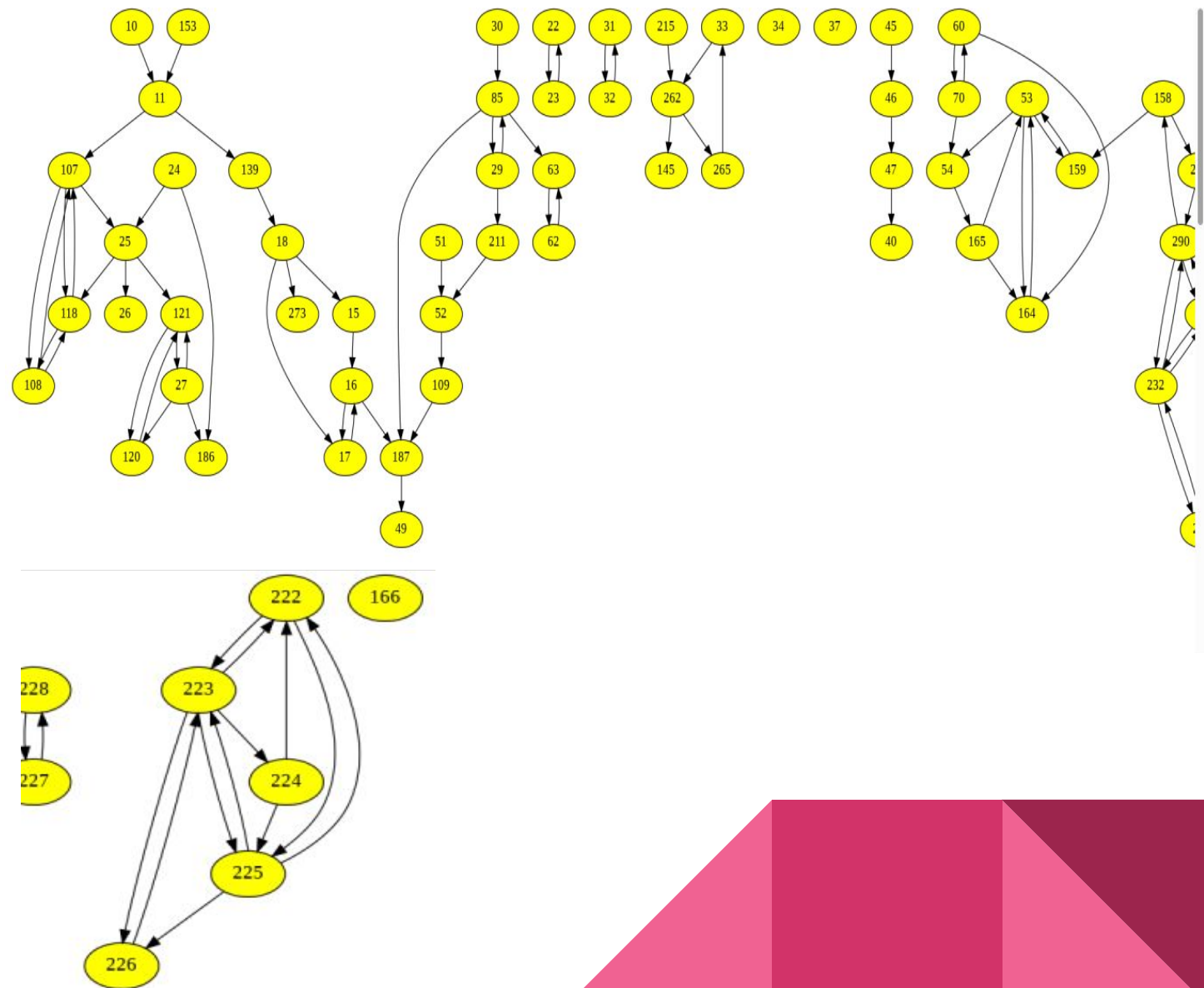
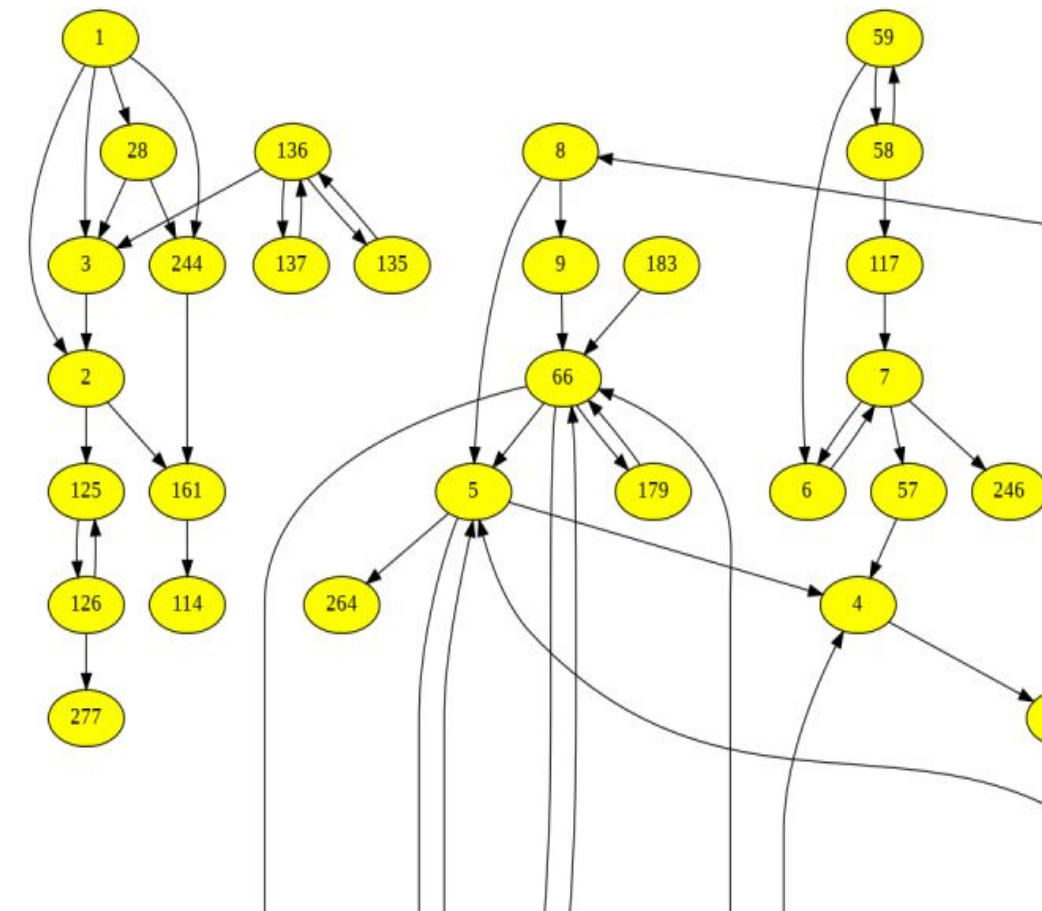
[12, 13, 12, 13, 35, 35]->30

[12, 13, 12, 13, 35]->31

[12, 13, 12, 13]->31



Graph



Parameters

Dimensionality of latent vectors : **$d = 100$** ;

Mini-batch size = **100**

Hyper-parameters:

1. On a validation set which is a random 10% subset of the training set.
2. All weights are initialized using a Gaussian distribution with a mean of 0 and a standard deviation of 0.1.
3. The mini-batch Adam optimizer is exerted to optimize these parameters, where the initial learning rate is set to **0.001** and will decay by **0.1** after every **3 epochs**.
Moreover the L2 penalty is set to **10^{-5}** respectively.



Metrics and Evaluations

1. Recall

Recall is the fraction of the relevant documents that are successfully retrieved.

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

2. Mean Reciprocal Rank

The **mean reciprocal rank** is a statistical measure for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer: 1 for first place, $\frac{1}{2}$ for second place, $\frac{1}{3}$ for third place and so on. The mean reciprocal rank is the average of the reciprocal ranks of results for a sample of queries Q:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}$$

Query	Proposed Results	Correct response	Rank	Reciprocal rank
cat	catten, cati, cats	cats	3	1/3
tori	torii, tori , toruses	tori	2	1/2
virus	viruses , virii, viri	viruses	1	1

For example, suppose we have the following three sample queries for a system that tries to translate English words to their plurals. In each case, the system makes three guesses, with the first one being the one it thinks is most likely correct. Given the above three samples, we calculate the mean reciprocal rank as $(1/3 + 1/2 + 1)/3 = 11/18$ or about 0.61. If none of the proposed results are correct, reciprocal rank is 0.

Results

Best Result for Diginetica Dataset:

Recall @ 20:- 82.0661

MRR @ 20:- 32.1141

Best Result for Yoochoose Dataset:

Recall @ 20:- 85.32

MRR @ 20 :- 25.64



Future Work

We can try to increase the efficiency of the recommender engine by incorporating LSTM instead of GRU-GNN.



References

- Session-based Recommendation with Graph Neural Networks
<https://doi.org/10.1609/aaai.v33i01.3301346>
- <https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be>
- <https://towardsdatascience.com/learn-how-recurrent-neural-networks-work-84e975feaaf7>
- <https://towardsdatascience.com/what-is-a-recurrent-nns-and-gated-recurrent-unit-grus-ea71d2a05a69>
- https://en.wikipedia.org/wiki/Gated_recurrent_unit
- <https://towardsdatascience.com/a-gentle-introduction-to-graph-neural-network-basics-deepwalk-and-graphsage-db5d540d50b3>





THANK YOU

Graph Neural Network (GNN)

- Graph Neural Network is a type of Neural Network which directly operates on the Graph structure. It is used in node classification.
- Every node in the graph is associated with a label, and we want to predict the label of the nodes without ground-truth.
- Given a partially labeled graph G , the goal is to leverage these labeled nodes to predict the labels of the unlabeled. In the node classification problem setup, each node v is characterized by its feature x_v and associated with a ground-truth label t_v .
- It learns to represent each node with a d dimensional vector (state) h_v which contains the information of its neighborhood.
- Specifically, where $x_{co[v]}$ denotes the features of the edges connecting with v , $h_{ne[v]}$ denotes the embedding of the neighboring nodes of v , and $x_{ne[v]}$ denotes the features of the neighboring nodes of v .

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]})$$

Graph neural networks

- The function f is the transition function that projects these inputs onto a d -dimensional space. Since we are seeking a unique solution for h_v , we can apply fixed point theorem and rewrite the above equation as an iteratively update process. Such operation is often referred to as **message passing** or **neighborhood aggregation**.

\mathbf{H} = concatenation of all the \mathbf{h}

\mathbf{X} denote the and \mathbf{x} ,

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X})$$

- The output of the GNN is computed by passing the state h_v as well as the feature x_v to an output function g .

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v)$$

Both f and g here can be interpreted as feed-forward fully-connected Neural Networks. The L1 loss can be straightforwardly formulated as the following which can be optimized via gradient descent.

$$loss = \sum_{i=1}^p (\mathbf{t}_i - \mathbf{o}_i)$$