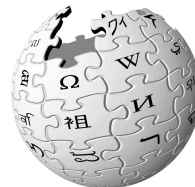




**HAI**  
—VIS



# A Tutorial on Wikimedia Visual Resources and its Application to Neural Visual Recommender Systems

Denis Parra<sup>1</sup>, Antonio Ossa-Guerra<sup>1</sup>, Manuel Cartagena<sup>1</sup>, Patricio Cerda-Mardini<sup>2</sup>,  
**Felipe del Río**<sup>1</sup>, Isidora Palma<sup>1</sup>, Diego Saez-Trumper<sup>3</sup>, and Miriam Redi<sup>3</sup>

1. Pontificia Universidad Católica de Chile

2. MindsDB

3. Wikimedia Foundation

21st IEEE International Conference on Data Mining



# Attentive Collaborative Filtering: Multimedia Recommendation with Item- and Component-Level Attention

Jingyuan Chen  
National University of Singapore  
jingyuanchen91@gmail.com

Hanwang Zhang  
Columbia University  
hanwangzhang@gmail.com

Xiangnan He\*  
National University of Singapore  
xiangnanhe@gmail.com

Liqiang Nie  
ShanDong University  
nieliqiang@gmail.com

Wei Liu  
Tencent AI Lab  
wliu@ee.columbia.edu

Tat-Seng Chua  
National University of Singapore  
dcscts@nus.edu.sg

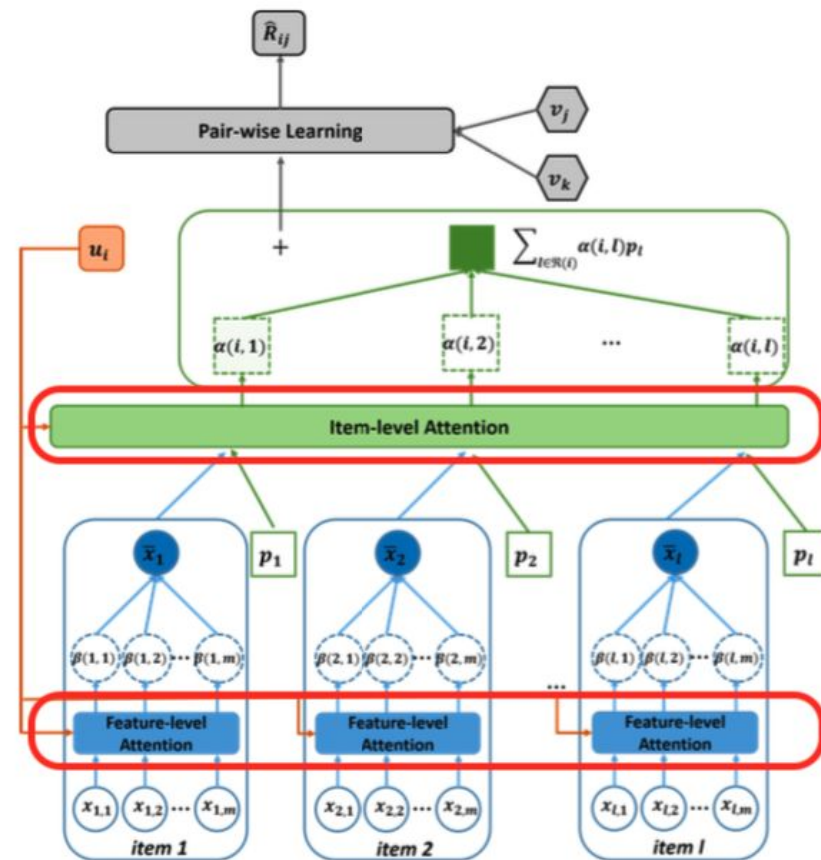
Presented as a full paper at SIGIR 2017

# Context

- Multimedia recommendation.
- Explicit feedback is not always available.
- Rely on implicit users interactions.

# Key Insights

- Two levels of implicit feedback:
  - Item level
  - Component Level
- Leverage (hierarchical) attention mechanism** to weight the importance of an item/component in each level.



# Approach

- BPR [4] based training.
- **Latent factor model + neighbourhood model.**
- Originally tested in Pinterest (images) and Vine (videos) datasets.
- In this tutorial we test it on the Wikimedia commons image dataset.

# Model

- User  $i$ , parametrized as  $u_i$
- Item  $l, j$ , parametrized as  $v_j$  &  $p_l$
- Component  $m$  of item  $l$   $x_{lm}$

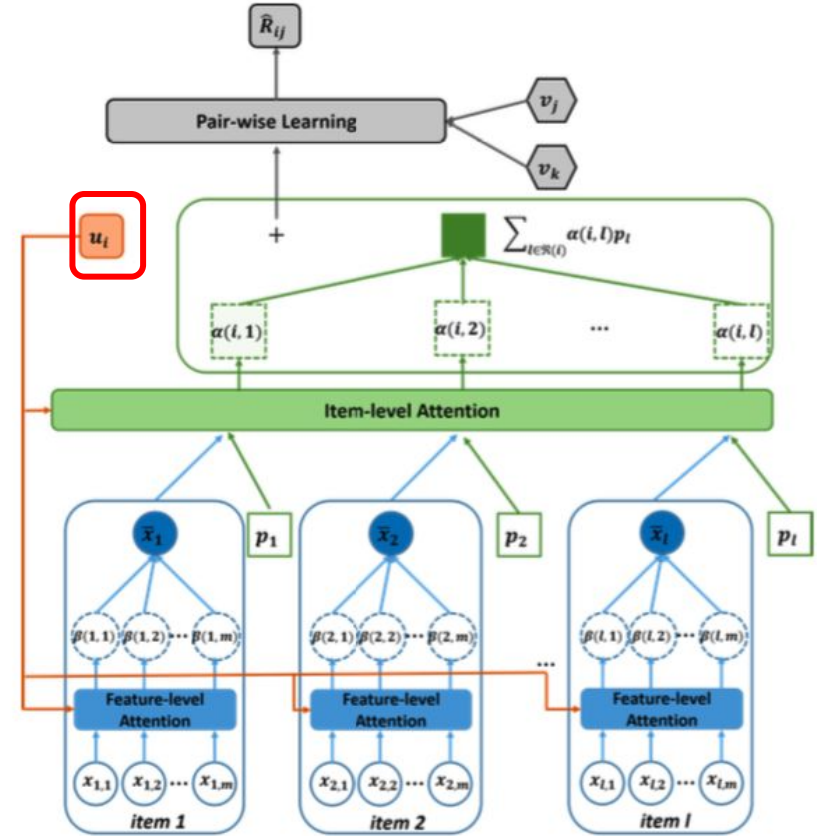
## Implementation Detail:

$v_j = p_l$  was used for efficiency and better performance.



# Model

- User  $i$ , parametrized as  $u_i$



# Model

- User  $i$ , parametrized as  $u_i$
- Item  $l, j$ , parametrized as  $v_j$  &  $p_l$





# Model

- User  $i$ , parametrized as  $u_i$
- Item  $l, j$ , parametrized as  $v_j$  &  $p_l$

## Implementation Detail:

$v_j = p_l$  was used for efficiency and better performance.

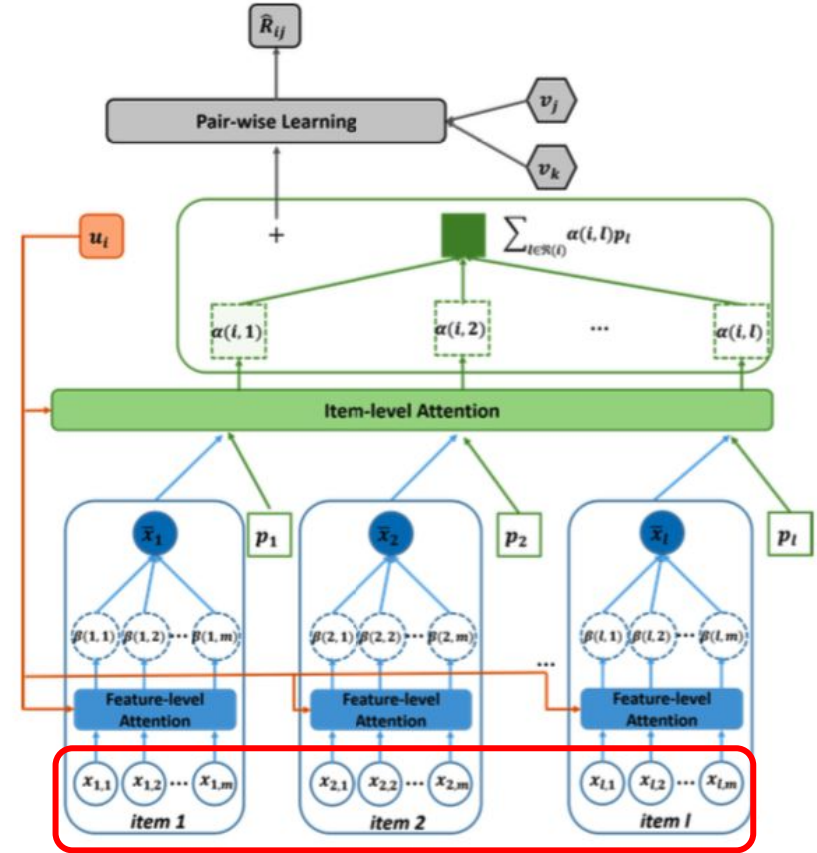


# Model

- User  $i$ , parametrized as  $u_i$
- Item  $l, j$ , parametrized as  $v_j$  &  $p_l$
- Component  $m$  of item  $l$   $x_{lm}$

## Implementation Detail:

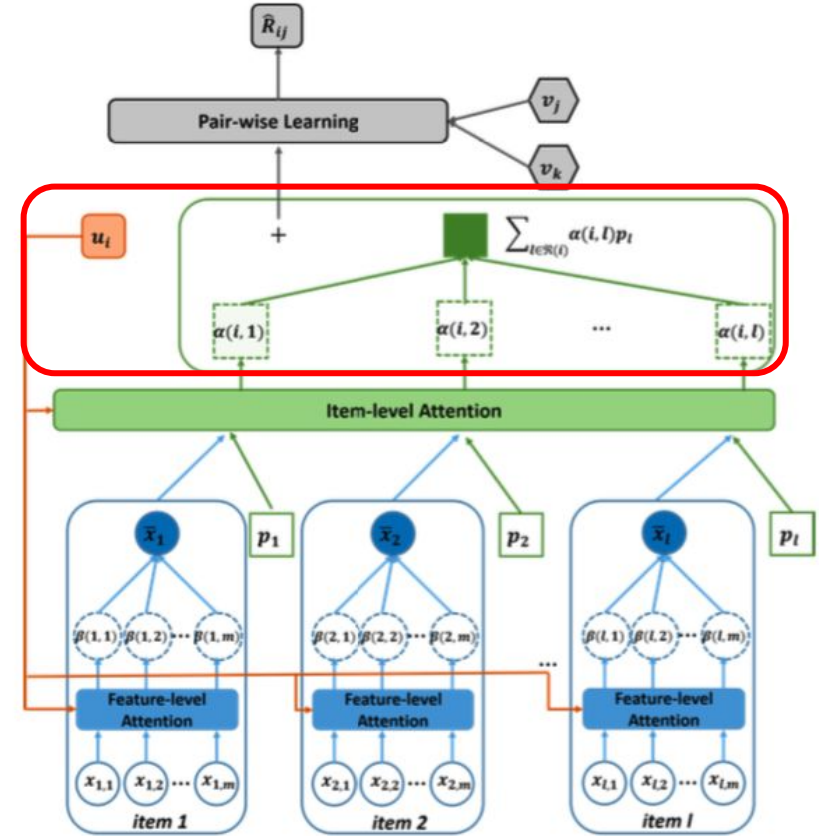
$v_j = p_l$  was used for efficiency and better performance.



# Model

User representation

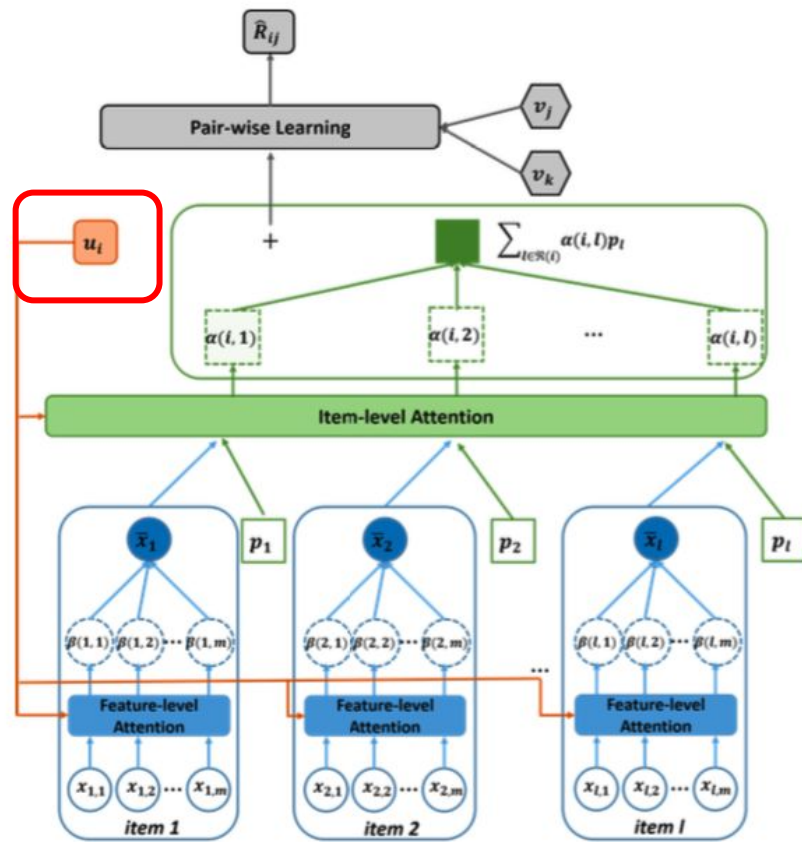
$$\mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$$



# Model

User representation

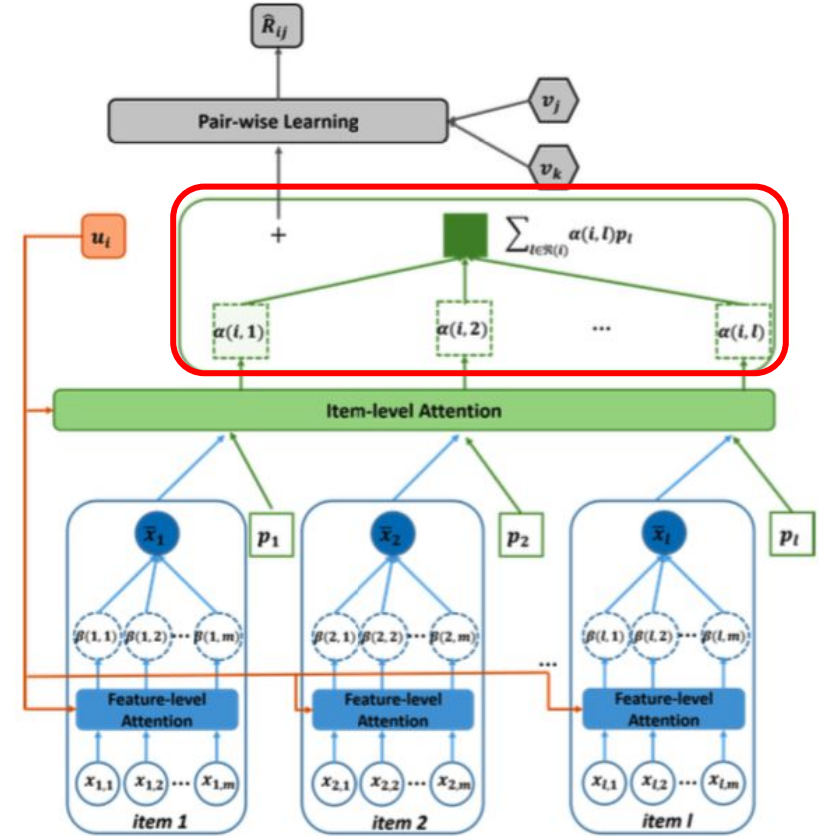
$$\boxed{u_i} + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) p_l$$



# Model

User representation

$$\mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$$



# Model

User representation

$$\mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$$

item level attention



# Model

User representation

$$\mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$$

consumed item  $l$  by user  $i$

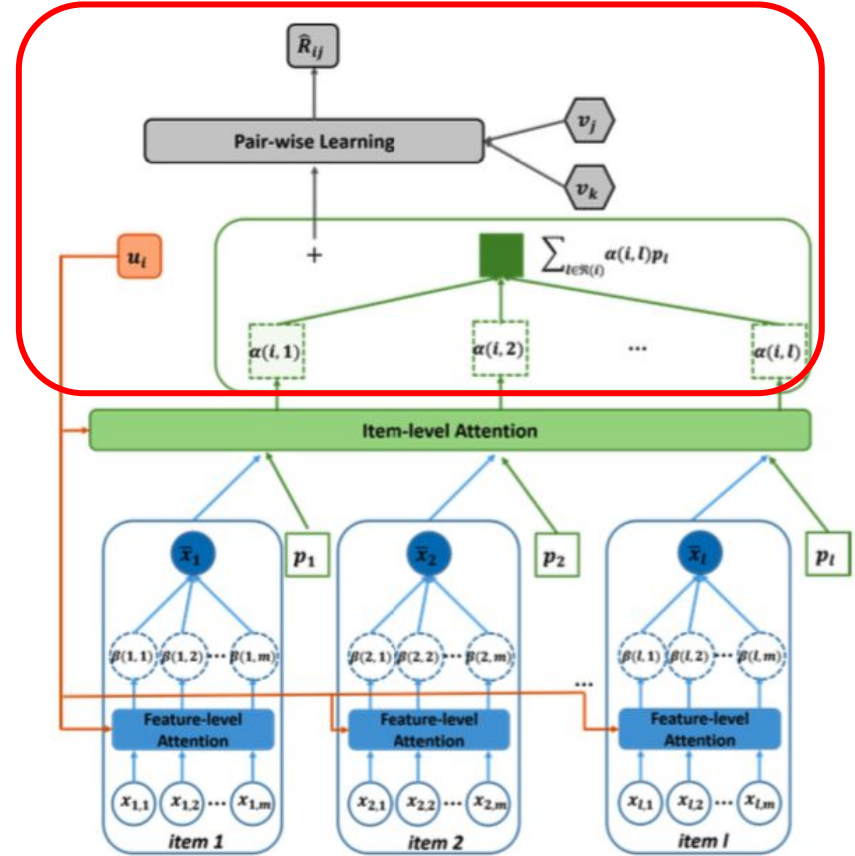
**Implementation Detail:** Limited the profile to a maximum of 9 items per user.



# Model

Estimated score

$$\hat{R}_{ij} = \left( \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l \right)^T \mathbf{v}_j$$





# Model

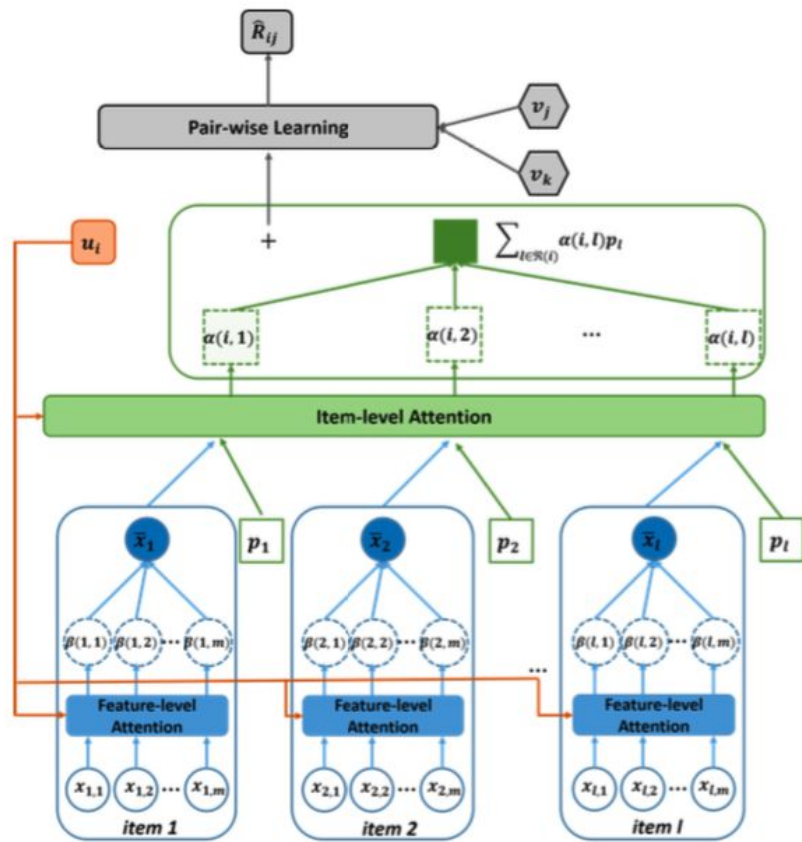
User representation

$$\mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$$

Estimated score

$$\hat{R}_{ij} = \left( \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l \right)^T \mathbf{v}_j$$

**Implementation Detail:** Limited the profile to a maximum of 9 items per user.



# Model

User representation

item level attention

$$\mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$$

consumed item  $l$  by user  $i$

Estimated score

$$\hat{R}_{ij} = \left( \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l \right)^T \mathbf{v}_j$$

**Implementation Detail:** Limited the profile to a maximum of 9 items per user.

# Model

## Item-level Attention



# Item Level Attention

$$a(i, l) = \mathbf{w}_1^T \phi(\overset{\text{user's / factors}}{\mathbf{W}_{1u} \mathbf{u}_i} + \mathbf{W}_{1v} \mathbf{v}_l + \mathbf{W}_{1p} \mathbf{p}_l + \mathbf{W}_{1x} \bar{\mathbf{x}}_l + \mathbf{b}_1) + \mathbf{c}_1$$

# Item Level Attention

$$a(i, l) = \mathbf{w}_1^T \phi(\mathbf{W}_{1u} \mathbf{u}_i + \mathbf{W}_{1v} \mathbf{v}_l + \mathbf{W}_{1p} \mathbf{p}_l + \mathbf{W}_{1x} \bar{\mathbf{x}}_l + \mathbf{b}_1) + \mathbf{c}_1$$

item's / factors

# Item Level Attention

$$a(i, l) = \mathbf{w}_1^T \phi(\mathbf{W}_{1u} \mathbf{u}_i + \mathbf{W}_{1v} \mathbf{v}_l + \mathbf{W}_{1p} \mathbf{p}_l + \mathbf{W}_{1x} \bar{\mathbf{x}}_l + \mathbf{b}_1) + \mathbf{c}_1$$

item's / factors

# Item Level Attention

item's / weighted components

$$a(i, l) = \mathbf{w}_1^T \phi(\mathbf{W}_{1u} \mathbf{u}_i + \mathbf{W}_{1v} \mathbf{v}_l + \mathbf{W}_{1p} \mathbf{p}_l + \mathbf{W}_{1x} \bar{\mathbf{x}}_l + \mathbf{b}_1) + \mathbf{c}_1$$

# Item Level Attention

$$a(i, l) = \mathbf{w}_1^T \underbrace{\phi(\mathbf{W}_{1u}\mathbf{u}_i + \mathbf{W}_{1v}\mathbf{v}_l + \mathbf{W}_{1p}\mathbf{p}_l + \mathbf{W}_{1x}\bar{\mathbf{x}}_l + \mathbf{b}_1)}_{\text{ReLU}} + \underbrace{\mathbf{c}_1}_{\text{bias terms}}$$



# Item Level Attention

$$a(i, l) = \mathbf{w}_1^T \phi(\mathbf{W}_{1u} \mathbf{u}_i + \mathbf{W}_{1v} \mathbf{v}_l + \mathbf{W}_{1p} \mathbf{p}_l + \mathbf{W}_{1x} \bar{\mathbf{x}}_l + \mathbf{b}_1) + \mathbf{c}_1$$

# Item Level Attention

It's an attention after all

$$\alpha(i, l) = \frac{\exp(a(i, l))}{\sum_{n \in \mathcal{R}(i)} \exp(a(i, n))}$$

# Item Level Attention

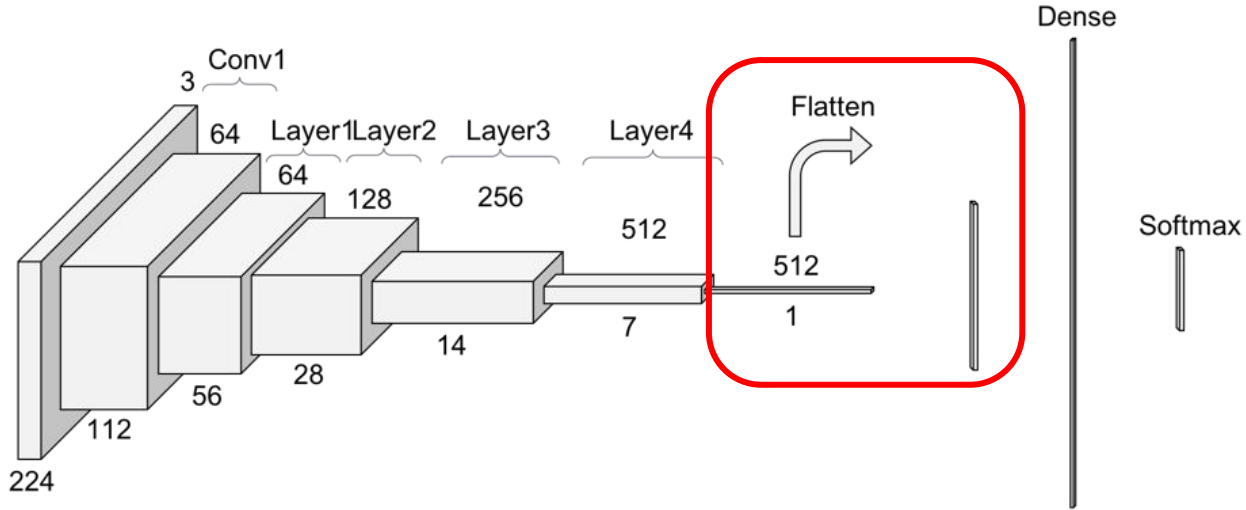
$$a(i, l) = \mathbf{w}_1^T \phi(\mathbf{W}_{1u} \mathbf{u}_i + \mathbf{W}_{1v} \mathbf{v}_l + \mathbf{W}_{1p} \mathbf{p}_l + \mathbf{W}_{1x} \bar{\mathbf{x}}_l + \mathbf{b}_1) + \mathbf{c}_1.$$

Diagram illustrating the components of the item-level attention score  $a(i, l)$ :

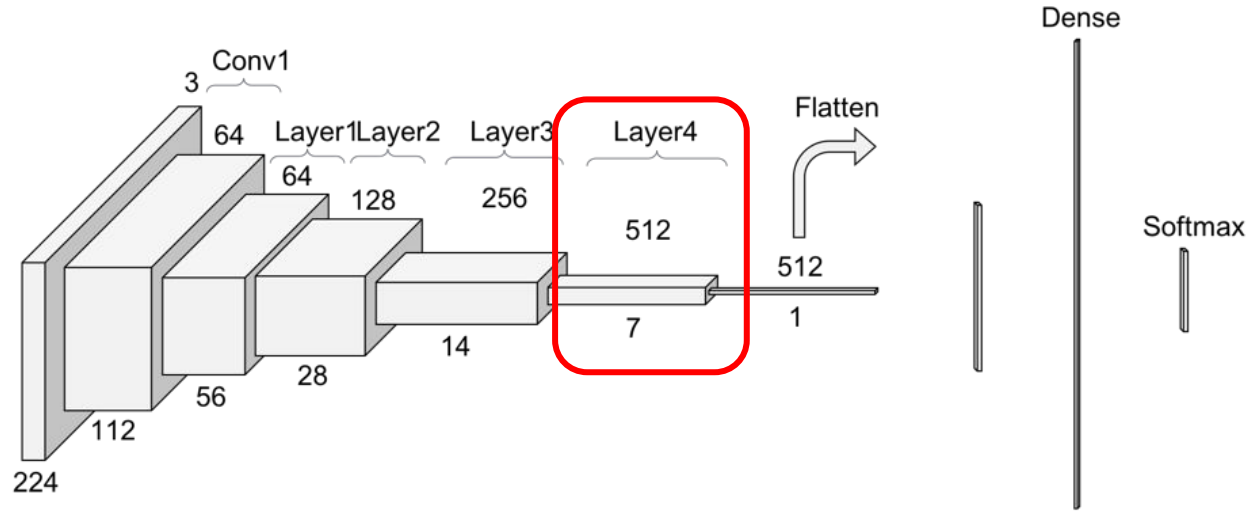
- $\mathbf{u}_i$ : user's  $i$  factors (indicated by an upward arrow)
- $\mathbf{v}_l$ : item's  $l$  factors (indicated by a downward arrow)
- $\mathbf{p}_l$ : item's  $l$  factors (indicated by a downward arrow)
- $\bar{\mathbf{x}}_l$ : item's  $l$  weighted components (indicated by an upward arrow)
- $\mathbf{b}_1$  and  $\mathbf{c}_1$ : bias terms (indicated by two downward arrows)

$$\alpha(i, l) = \frac{\exp(a(i, l))}{\sum_{n \in \mathcal{R}(i)} \exp(a(i, n))}.$$

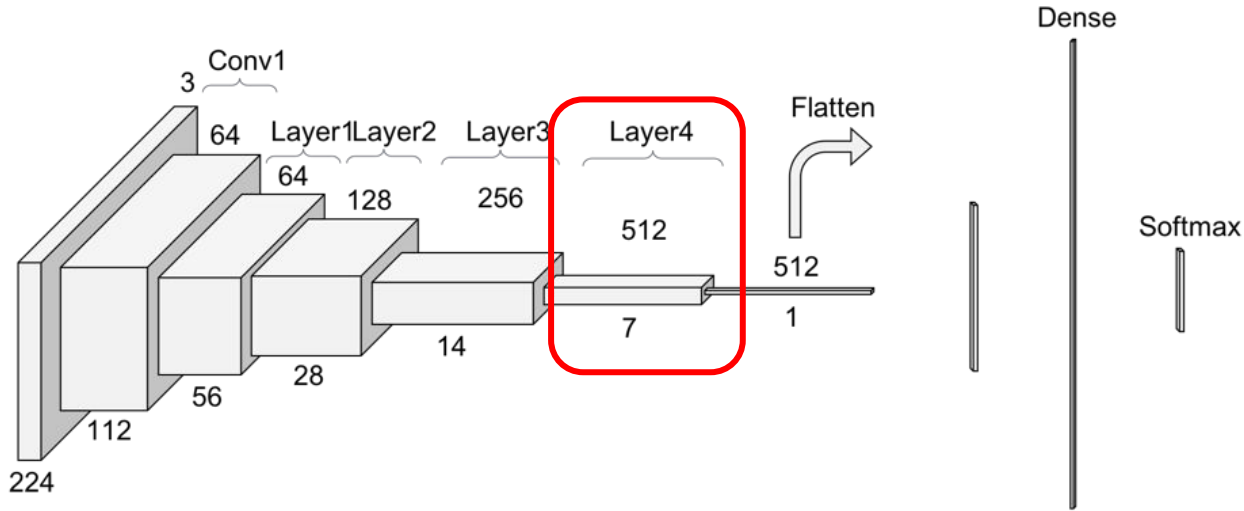
# Components



# Components



# Components



## Implementation detail:

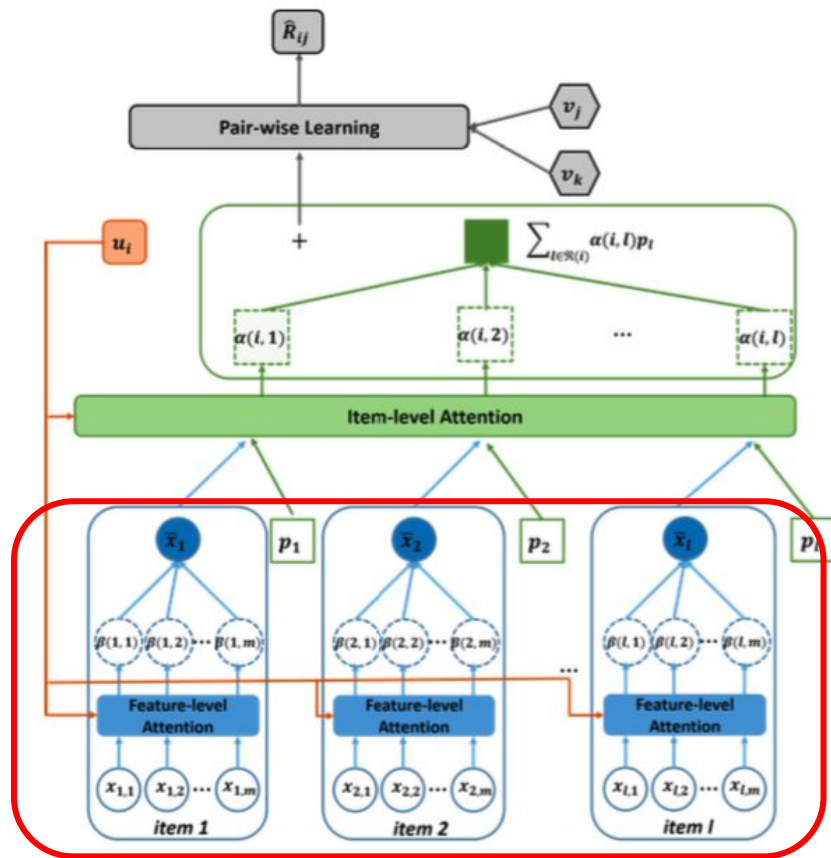
Used the output of the **layer4** of a ResNet50 [2] for this tutorial.

Add layer to reduce dimension of each feature vector.

# Model

Weighted components for item  $l$

$$\bar{x}_l = \sum_{m=1}^{|\{x_{l*}\}|} \beta(i, l, m) \cdot x_{lm}$$

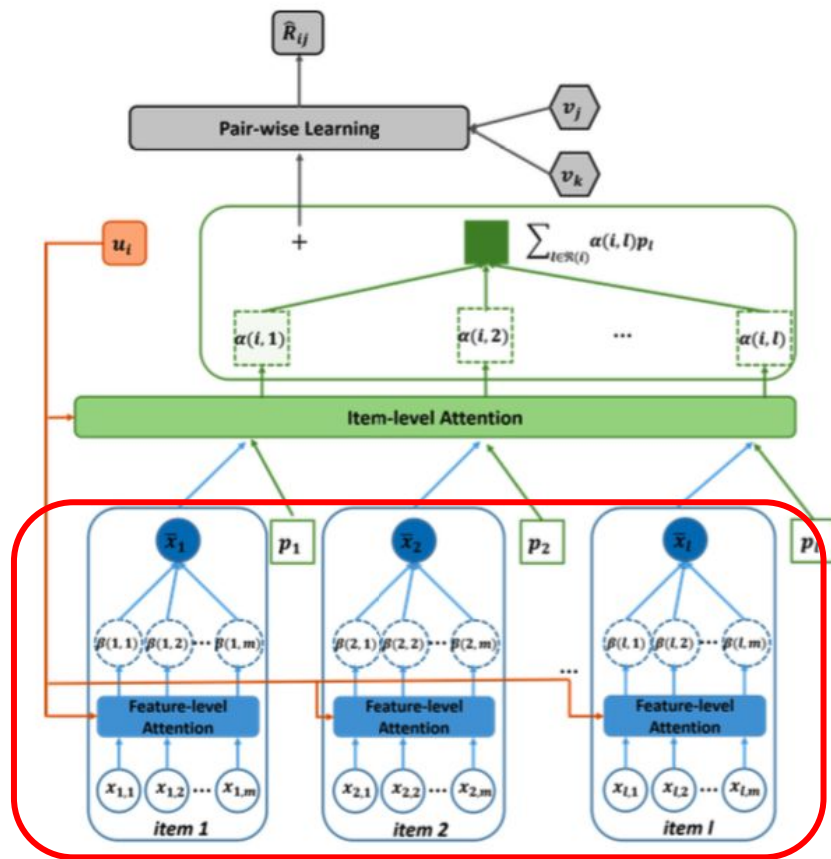


# Model

Weighted components for item  $l$

$$\bar{x}_l = \sum_{m=1}^{|\{x_{l*}\}|} \beta(i, l, m) \cdot x_{lm}$$

component features



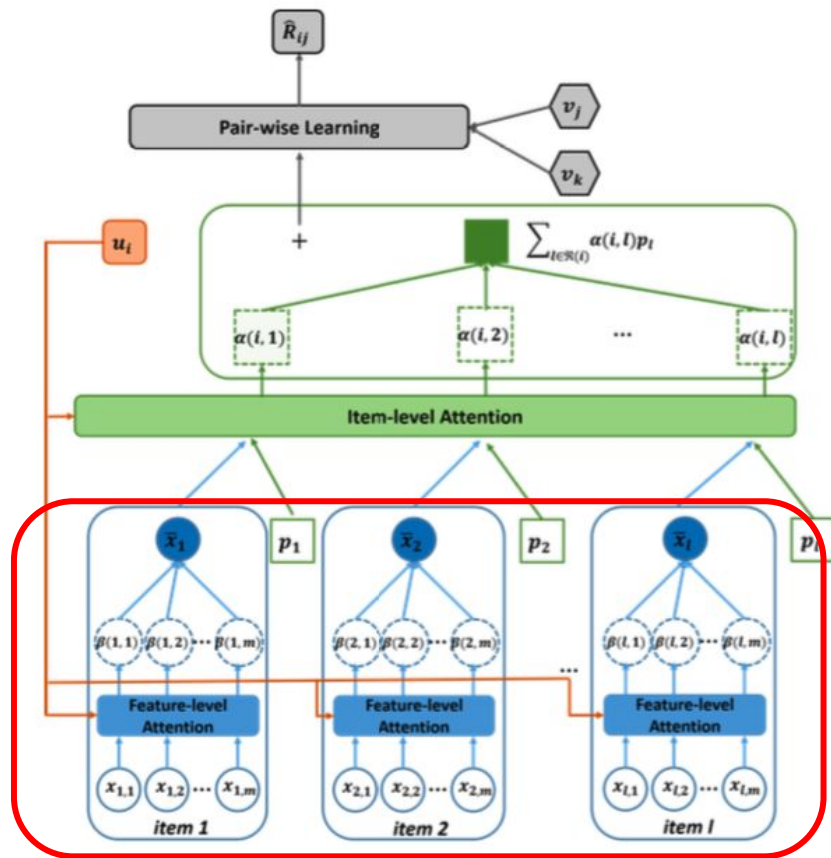


# Model

Weighted components for item  $l$

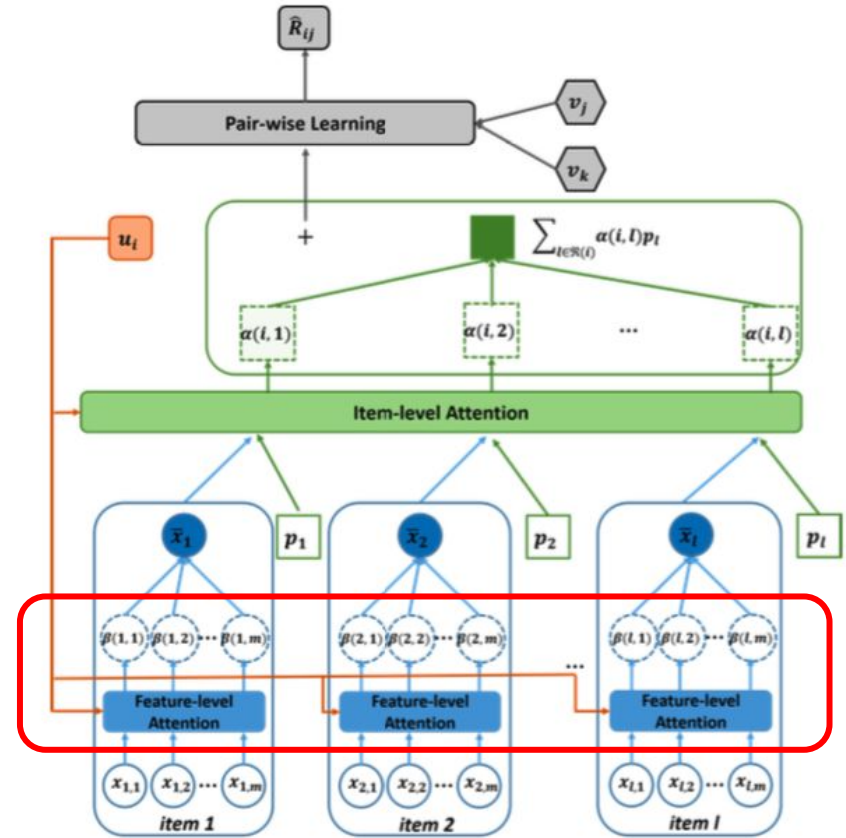
$$\bar{x}_l = \sum_{m=1}^{|\{x_{l*}\}|} \beta(i, l, m) \cdot x_{lm}$$

component level attention



# Model

## Component-level Attention



# Component Level Attention

user's  $i$  factors

$$b(i, l, m) = \mathbf{w}_2^T \phi(\mathbf{W}_{2u} \mathbf{u}_i + \mathbf{W}_{2x} \mathbf{x}_{lm} + \mathbf{b}_2) + \mathbf{c}_2$$

# Component Level Attention

$$b(i, l, m) = \mathbf{w}_2^T \phi(\mathbf{W}_{2u} \mathbf{u}_i + \mathbf{W}_{2x} \mathbf{x}_{lm} + \mathbf{b}_2) + \mathbf{c}_2$$

component  $m$  of item  $l$

# Component Level Attention

$$b(i, l, m) = \mathbf{w}_2^T \underbrace{\phi(\mathbf{W}_{2u} \mathbf{u}_i + \mathbf{W}_{2x} \mathbf{x}_{lm} + \mathbf{b}_2)}_{\text{ReLU}} + \mathbf{c}_2$$

bias terms

# Component Level Attention

$$b(i, l, m) = \mathbf{w}_2^T \phi(\mathbf{W}_{2u} \mathbf{u}_i + \mathbf{W}_{2x} \mathbf{x}_{lm} + \mathbf{b}_2) + \mathbf{c}_2$$

ReLU

# Component Level Attention

It's an attention after all

$$\beta(i, l, m) = \frac{\exp(b(i, l, m))}{\sum_{n=1}^{|\{x_{l*}\}|} \exp(b(i, l, n))}$$

# Component Level Attention

Weighted components for item  $l$       component level attention

$$\bar{\mathbf{x}}_l = \sum_{m=1}^{|\{\mathbf{x}_{l*}\}|} \beta(i, l, m) \cdot \mathbf{x}_{lm},$$

$$b(i, l, m) = \mathbf{w}_2^T \phi(\mathbf{W}_{2u} \mathbf{u}_i + \mathbf{W}_{2x} \mathbf{x}_{lm} + \mathbf{b}_2) + \mathbf{c}_2,$$

Diagram illustrating the calculation of the component level attention weight  $b(i, l, m)$ :

- $\mathbf{u}_i$  is labeled "user's  $i$  factors".
- $\mathbf{x}_{lm}$  is labeled "component  $m$  of item  $l$ ".
- $\mathbf{b}_2$  and  $\mathbf{c}_2$  are labeled "bias terms".
- The function  $\phi$  is labeled "ReLU".



# Training

BPR [4] Optimization Objective

$$\arg \min_{\mathbf{U}, \mathbf{V}} \sum_{(i,j,k) \in \mathcal{R}_{\mathcal{B}}} -\ln \sigma(\hat{R}_{ij} - \hat{R}_{ik}) + \lambda(\|\mathbf{U}^2\| + \|\mathbf{V}^2\|)$$

Trained using Adam.

**Implementation detail:**

Regularization was key to increase performance as well as to avoid flat attentions.

$0 < \lambda < 10^{-4}$  worked, with  $10^{-5}$  achieving the best results.

# Main Results

AUC	RR	R@20	P@20	nDCG@20	R@100	P@100	nDCG@100
0.77703	0.03548	0.01381	0.00802	0.04792	0.05142	0.00588	0.07886

# Recommendation Examples

Ground Truth (n=1)



Consumed (n=4)



attn=0.00000



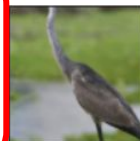
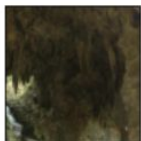
attn=0.00032



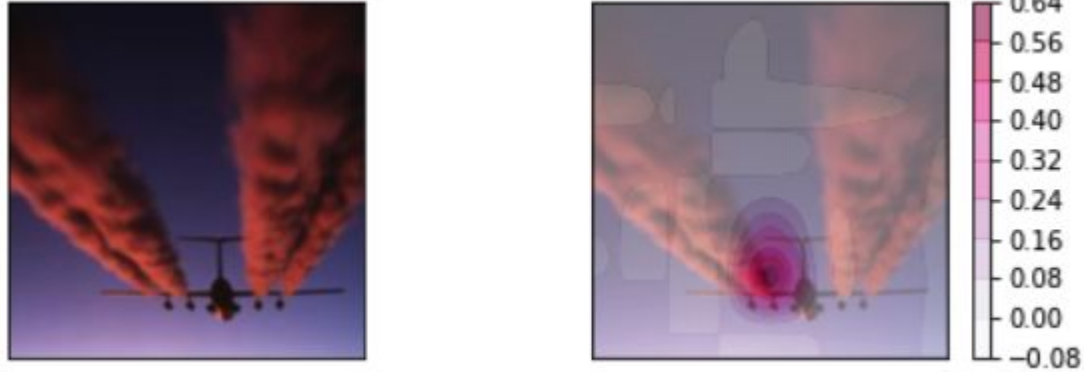
attn=0.00000



Recommendation (n=10)



# Attention Visualization



# Recommendation Examples

Ground Truth (n=1)



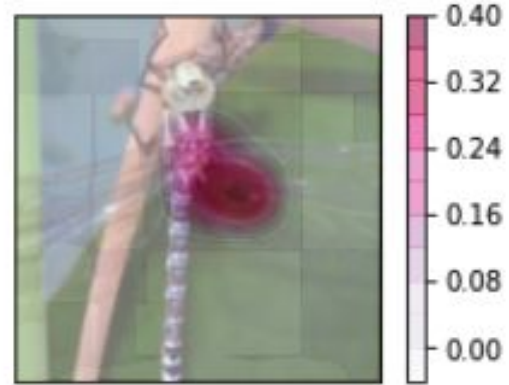
Consumed (n=10)



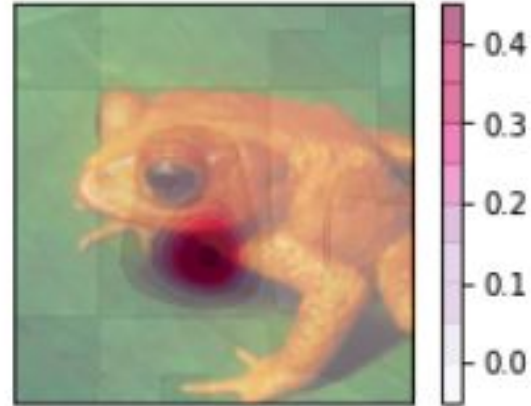
Recommendation (n=20)



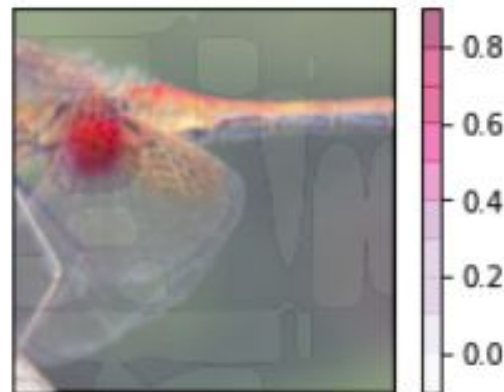
# Attention Visualization



# Attention Visualization

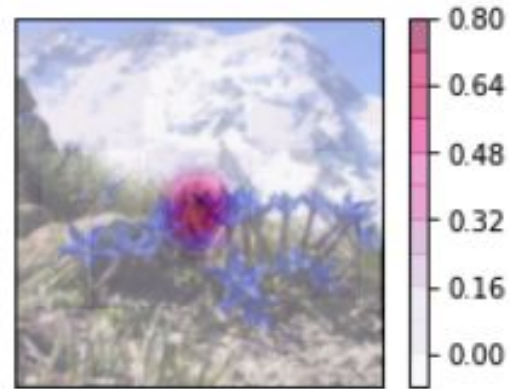


# Attention Visualization





# Attention Visualization



# Attention Visualization





**HAI**  
—VIS



# VisRec: A Hands-on Tutorial on Deep Learning for Visual Recommender Systems

Denis Parra, Antonio Ossa-Guerra, Manuel Cartagena, \*Patricio  
Cerdeira-Mardini, **Felipe del Río**

Pontificia Universidad Católica de Chile

\*MindsDB

21st IEEE International Conference on Data Mining





**HAI**  
—VIS



# Thank you!

Felipe del Río

[fidelrio@uc.cl]

Denis Parra, Antonio Ossa-Guerra, Manuel Cartagena, \*Patricio Cerda-Mardini, **Felipe del Río**

Pontificia Universidad Católica de Chile

\*MindsDB

21st IEEE International Conference on Data Mining



# References

- [1] Chen, J., Zhang, H., He, X., Nie, L., Liu, W., & Chua, T. S. (2017, August). Attentive collaborative filtering: Multimedia recommendation with item-and component-level attention. In *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval* (pp. 335-344).
- [2] Melinte, D. O., & Vladareanu, L. (2020). Facial Expressions Recognition for Human–Robot Interaction Using Deep Convolutional Neural Networks with Rectified Adam Optimizer. *Sensors*, 20(8), 2393.
- [3] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).
- [4] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. BPR: bayesian personalized ranking from implicit feedback. In *UAI*, pages 452–461. IEEE, 2009.
- [5] Weston, J., Bengio, S., & Usunier, N. (2011). Wsabee: Scaling up to large vocabulary image annotation.

# Supporting Material

# General Implementation Details

Implemented using python 3.7.5 and pytorch 1.7.

Ran in research group cluster with 8-cpu and 2 GeForce GTX 1080 Ti (only one used per model training).

Each epoch took around 3 hours.

# Algorithm

**Input:** User-item interaction matrix  $\mathbf{R}$ . Each item  $l$  is represented by a set of component features  $\{x_{l*}\}$ .

**Output:** Latent feature matrix  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and parameters in attention model  $\Theta$

- 1: Initialize  $\mathbf{U}$ ,  $\mathbf{V}$  and  $\mathbf{P}$  with Gaussian distribution. Initialize  $\Theta$  with xavier [17].
- 2: **repeat**
- 3:   draw  $(i, j, k)$  from  $\mathcal{R}_B$
- 4:   For each item  $l$  in  $\mathcal{R}(i)$ :
- 5:     For each component  $m$  in  $\{x_{l*}\}$ :
- 6:       Compute  $\beta(i, l, m)$  according to Eqns. (10) and (11)
- 7:       Compute  $\bar{x}_l$  according to Eqn. (12)
- 8:       Compute  $\alpha(i, l)$  according to Eqns. (8) and (9)
- 9:        $\mathbf{u}'_i \leftarrow \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$
- 10:       $\hat{R}_{ijk} \leftarrow \mathbf{u}'_i \mathbf{v}_j - \mathbf{u}'_i \mathbf{v}_k$
- 11:      For each parameter  $\theta$  in  $\{\mathbf{U}, \mathbf{V}, \mathbf{P}, \Theta\}$ :
- 12:       Update  $\theta \leftarrow \theta + \eta \cdot \left( \frac{\exp^{-\hat{R}_{ijk}}}{1 + \exp^{-\hat{R}_{ijk}}} \cdot \frac{\partial \hat{R}_{ijk}}{\partial \theta} + \lambda \cdot \theta \right)$ .
- 13: **until** convergence
- 14: return  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and  $\Theta$ .



# Algorithm

**Input:** User-item interaction matrix  $\mathbf{R}$ . Each item  $l$  is represented by a set of component features  $\{x_{l*}\}$ .

**Output:** Latent feature matrix  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and parameters in attention model  $\Theta$

1: Initialize  $\mathbf{U}$ ,  $\mathbf{V}$  and  $\mathbf{P}$  with Gaussian distribution. Initialize  $\Theta$  with xavier [17].

→ Kaiming was used

2: **repeat**

3:   draw  $(i, j, k)$  from  $\mathcal{R}_B$

4:   For each item  $l$  in  $\mathcal{R}(i)$ :

5:     For each component  $m$  in  $\{x_{l*}\}$ :

6:       Compute  $\beta(i, l, m)$  according to Eqns. (10) and (11)

7:       Compute  $\bar{x}_l$  according to Eqn. (12)

8:   Compute  $\alpha(i, l)$  according to Eqns. (8) and (9)

9:    $\mathbf{u}'_i \leftarrow \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$

10:    $\hat{R}_{ijk} \leftarrow \mathbf{u}'_i \mathbf{v}_j - \mathbf{u}'_i \mathbf{v}_k$

11:   For each parameter  $\theta$  in  $\{\mathbf{U}, \mathbf{V}, \mathbf{P}, \Theta\}$ :

12:     Update  $\theta \leftarrow \theta + \eta \cdot \left( \frac{\exp^{-\hat{R}_{ijk}}}{1 + \exp^{-\hat{R}_{ijk}}} \cdot \frac{\partial \hat{R}_{ijk}}{\partial \theta} + \lambda \cdot \theta \right)$ .

13: **until** convergence

14: return  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and  $\Theta$ .

# Algorithm

**Input:** User-item interaction matrix  $\mathbf{R}$ . Each item  $l$  is represented by a set of component features  $\{x_{l*}\}$ .

**Output:** Latent feature matrix  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and parameters in attention model  $\Theta$

```
1: Initialize  $\mathbf{U}$ ,  $\mathbf{V}$  and  $\mathbf{P}$  with Gaussian distribution. Initialize  $\Theta$  with xavier [17].
2: repeat
3:   draw  $(i, j, k)$  from  $\mathcal{R}_B$ 
4:   For each item  $l$  in  $\mathcal{R}(i)$ :
5:     For each component  $m$  in  $\{x_{l*}\}$ :
6:       Compute  $\beta(i, l, m)$  according to Eqns. (10) and (11)
7:       Compute  $\bar{x}_l$  according to Eqn. (12)
8:     Compute  $\alpha(i, l)$  according to Eqns. (8) and (9)
9:      $\mathbf{u}'_i \leftarrow \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$ 
10:     $\hat{R}_{ijk} \leftarrow \mathbf{u}'_i \mathbf{v}_j - \mathbf{u}'_i \mathbf{v}_k$ 
11:    For each parameter  $\theta$  in  $\{\mathbf{U}, \mathbf{V}, \mathbf{P}, \Theta\}$ :
12:      Update  $\theta \leftarrow \theta + \eta \cdot \left( \frac{\exp^{-\hat{R}_{ijk}}}{1 + \exp^{-\hat{R}_{ijk}}} \cdot \frac{\partial \hat{R}_{ijk}}{\partial \theta} + \lambda \cdot \theta \right)$ .
13: until convergence
14: return  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and  $\Theta$ .
```

Triplet sampling originally bootstrap sampling, we did it without replacement for consistency.

# Algorithm

**Input:** User-item interaction matrix  $\mathbf{R}$ . Each item  $l$  is represented by a set of component features  $\{x_{l*}\}$ .

**Output:** Latent feature matrix  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and parameters in attention model  $\Theta$

- 1: Initialize  $\mathbf{U}$ ,  $\mathbf{V}$  and  $\mathbf{P}$  with Gaussian distribution. Initialize  $\Theta$  with xavier [17].
- 2: **repeat**
- 3:   draw  $(i, j, k)$  from  $\mathcal{R}_B$
- 4:   For each item  $l$  in  $\mathcal{R}(i)$ :
- 5:     For each component  $m$  in  $\{x_{l*}\}$ :
- 6:       Compute  $\beta(i, l, m)$  according to Eqns. (10) and (11)
- 7:       Compute  $\bar{x}_l$  according to Eqn. (12)
- 8:     Compute  $\alpha(i, l)$  according to Eqns. (8) and (9)
- 9:      $\mathbf{u}'_i \leftarrow \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$
- 10:     $\hat{R}_{ijk} \leftarrow \mathbf{u}'_i \mathbf{v}_j - \mathbf{u}'_i \mathbf{v}_k$
- 11:    For each parameter  $\theta$  in  $\{\mathbf{U}, \mathbf{V}, \mathbf{P}, \Theta\}$ :
- 12:      Update  $\theta \leftarrow \theta + \eta \cdot \left( \frac{\exp^{-\hat{R}_{ijk}}}{1 + \exp^{-\hat{R}_{ijk}}} \cdot \frac{\partial \hat{R}_{ijk}}{\partial \theta} + \lambda \cdot \theta \right)$ .
- 13: **until** convergence
- 14: return  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and  $\Theta$ .

Component  
attention

# Algorithm

**Input:** User-item interaction matrix  $\mathbf{R}$ . Each item  $l$  is represented by a set of component features  $\{x_{l*}\}$ .

**Output:** Latent feature matrix  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and parameters in attention model  $\Theta$

```
1: Initialize  $\mathbf{U}$ ,  $\mathbf{V}$  and  $\mathbf{P}$  with Gaussian distribution. Initialize  $\Theta$  with xavier [17].
2: repeat
3:   draw  $(i, j, k)$  from  $\mathcal{R}_B$ 
4:   For each item  $l$  in  $\mathcal{R}(i)$ :
5:     For each component  $m$  in  $\{x_{l*}\}$ :
6:       Compute  $\beta(i, l, m)$  according to Eqns. (10) and (11)
7:       Compute  $\bar{x}_l$  according to Eqn. (12)
8:       Compute  $\alpha(i, l)$  according to Eqns. (8) and (9)
9:        $\mathbf{u}'_l \leftarrow \mathbf{u}_l + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$ 
10:     $\hat{R}_{ijk} \leftarrow \mathbf{u}'_i \mathbf{v}_j - \mathbf{u}'_i \mathbf{v}_k$ 
11:    For each parameter  $\theta$  in  $\{\mathbf{U}, \mathbf{V}, \mathbf{P}, \Theta\}$ :
12:      Update  $\theta \leftarrow \theta + \eta \cdot \left( \frac{\exp^{-\hat{R}_{ijk}}}{1 + \exp^{-\hat{R}_{ijk}}} \cdot \frac{\partial \hat{R}_{ijk}}{\partial \theta} + \lambda \cdot \theta \right)$ .
13: until convergence
14: return  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and  $\Theta$ .
```

Item Attention

# Algorithm

**Input:** User-item interaction matrix  $\mathbf{R}$ . Each item  $l$  is represented by a set of component features  $\{x_{l*}\}$ .

**Output:** Latent feature matrix  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and parameters in attention model  $\Theta$

- 1: Initialize  $\mathbf{U}$ ,  $\mathbf{V}$  and  $\mathbf{P}$  with Gaussian distribution. Initialize  $\Theta$  with xavier [17].
- 2: **repeat**
- 3:   draw  $(i, j, k)$  from  $\mathcal{R}_B$
- 4:   For each item  $l$  in  $\mathcal{R}(i)$ :
- 5:     For each component  $m$  in  $\{x_{l*}\}$ :
- 6:       Compute  $\beta(i, l, m)$  according to Eqns. (10) and (11)
- 7:       Compute  $\bar{x}_l$  according to Eqn. (12)
- 8:       Compute  $\alpha(i, l)$  according to Eqns. (8) and (9)
- 9:        $\mathbf{u}'_i \leftarrow \mathbf{u}_i + \sum_{l \in \mathcal{R}(i)} \alpha(i, l) \mathbf{p}_l$
- 10:       $\hat{R}_{ijk} \leftarrow \mathbf{u}'_i \mathbf{v}_j - \mathbf{u}'_i \mathbf{v}_k$
- 11:      For each parameter  $\theta$  in  $\{\mathbf{U}, \mathbf{V}, \mathbf{P}, \Theta\}$ :
- 12:       Update  $\theta \leftarrow \theta + \eta \cdot \left( \frac{\exp^{-\hat{R}_{ijk}}}{1 + \exp^{-\hat{R}_{ijk}}} \cdot \frac{\partial \hat{R}_{ijk}}{\partial \theta} + \lambda \cdot \theta \right)$ .
- 13: **until** convergence
- 14: return  $\mathbf{U}$ ,  $\mathbf{V}$ ,  $\mathbf{P}$  and  $\Theta$ .

Compute scores and update model parameters

# Hands On Code Slides

# Hands On!

models/acf.py

@ class ACF



Model logic

```
class ACF(nn.Module):
    def __init__(self,
                  users,
                  items,
                  feature_path,
                  model_dim=128,
                  input_feature_dim=0,
                  tied_item_embedding=True,
                  device=None):

        super().__init__()
        self.pad_token = 0
        self.device = device

        # Should be moved to an ACFRecommender
        self.users = users
        self.items = items
        self.feature_path = feature_path
        self.model_dim = model_dim
        self.input_feature_dim = input_feature_dim

        self.all_items = torch.tensor(items)
        self.all_items = self.all_items + 1 if self.all_items.min() == 0 else self.all_items
        self.feature_data = self.load_feature_data(feature_path)
        num_items = max(self.all_items) + 1

        input_feature_dim = self.feature_data.shape[-1]
        self.item_model = nn.Embedding(num_items, self.model_dim, padding_idx=self.pad_token)
        self.user_model = ACFUserNet(users,
                                      items,
                                      emb_dim=self.model_dim,
                                      input_feature_dim=input_feature_dim,
                                      profile_embedding=self.item_model,
                                      device=self.device)
```

# Hands On!

models/acf.py

@class ACF



Model logic

```
def forward(self, user_id, profile_ids, pos, neg, profile_mask):
    profile_features = self.get_features(profile_ids).to(self.device)

    user_output = self.user_model(user_id, profile_ids, profile_features, profile_mask)
    user = user_output['user']

    pos_pred = self.get_predictions(user, pos)
    neg_pred = self.get_predictions(user, neg)

    return pos_pred, neg_pred
```

```
def get_predictions(self, user, items):
    item_embeddings = self.item_model(items)
    prediction = self.score(user, item_embeddings)
    return prediction
```

```
def score(self, user, items):
    return (user * items).sum(1) / self.model_dim
```



# Hands On!

models/acf.py

@ class ACFUserNet



User level logic

```
class ACFUserNet(nn.Module):
    """
    Get user embedding accounting to surpassed items
    """

    def __init__(self, users, items, emb_dim=128, input_feature_dim=0, profile_embedding=None, device=None):
        super().__init__()
        self.pad_token = 0

        self.emb_dim = emb_dim
        num_users = max(users) + 1
        num_items = max(items) + 1

        reduced_feature_dim = emb_dim
        self.feats = ACFFeatureNet(emb_dim, input_feature_dim, reduced_feature_dim) if input_feature_dim > 0 else None

        self.user_embedding = nn.Embedding(num_users, emb_dim)
        if not profile_embedding:
            self.profile_embedding = nn.Embedding(num_items, emb_dim, padding_idx=self.pad_token)
        else:
            self.profile_embedding = profile_embedding

        f = 1 if self.feats is not None else 0
        self.w_u = nn.Linear(emb_dim, emb_dim)
        self.w_v = nn.Linear(emb_dim, emb_dim)
        self.w_p = nn.Linear(emb_dim, emb_dim)
        self.w_x = nn.Linear(emb_dim, emb_dim)
        self.w = nn.Linear(emb_dim, 1)

        self._kaiming_(self.w_u)
        self._kaiming_(self.w_v)
        self._kaiming_(self.w_p)
        self._kaiming_(self.w_x)
        self._kaiming_(self.w)
```

# Hands On!

models/acf.py

@ class ACFUserNet



User level logic

```
def forward(self, user_ids, profile_ids, features, profile_mask, return_component_attentions=False,
            return_profile_attentions=False, return_attentions=False):
    ...

    user = self.user_embedding(user_ids)

    profile = self.profile_embedding(profile_ids)

    features = features.flatten(start_dim=2, end_dim=3) # Add
    feat_output = self.feats(user, features, profile_mask, return_component_attentions)
    components = feat_output['pooled_features']

    user = self.w_u(user)
    profile_query = self.w_p(profile)
    components = self.w_x(components)

    profile_query = profile_query.permute((1,0,2))
    components = components.permute((1,0,2))

    alpha = F.relu(user + profile_query + components) # TODO: + item, Add current_item emb (?)
    alpha = self.w(alpha)

    profile_mask = profile_mask.permute((1,0))
    profile_mask = profile_mask.unsqueeze(-1)
    alpha = alpha.masked_fill(torch.logical_not(profile_mask), float('-inf'))
    alpha = F.softmax(alpha, dim=0)

    alpha = alpha.permute((1,0,2))
    user_profile = (alpha * profile).sum(dim=1)

    user = user + user_profile
    output = {'user': user}
    if return_component_attentions:
        output['component_attentions'] = feat_output['attentions']
    if return_profile_attentions:
        output['profile_attentions'] = alpha.squeeze(-1)

    return output
```

# Hands On!

models/acf.py

@ class ACFUserNet

User & profile latent factors

```
def forward(self, user_ids, profile_ids, features, profile_mask, return_component_attentions=False,
            return_profile_attentions=False, return_attentions=False):
    ...
    user = self.user_embedding(user_ids)
    profile = self.profile_embedding(profile_ids)

    features = features.flatten(start_dim=2, end_dim=3) # Add
    feat_output = self.feats(user, features, profile_mask, return_attentions=return_component_attentions)
    components = feat_output['pooled_features']

    user = self.w_u(user)
    profile_query = self.w_p(profile)
    components = self.w_x(components)

    profile_query = profile_query.permute((1,0,2))
    components = components.permute((1,0,2))

    alpha = F.relu(user + profile_query + components) # TODO: + item, Add current_item emb (?)
    alpha = self.w(alpha)

    profile_mask = profile_mask.permute((1,0))
    profile_mask = profile_mask.unsqueeze(-1)
    alpha = alpha.masked_fill(torch.logical_not(profile_mask), float('-inf'))
    alpha = F.softmax(alpha, dim=0)

    alpha = alpha.permute((1,0,2))
    user_profile = (alpha * profile).sum(dim=1)

    user = user + user_profile
    output = {'user': user}
    if return_component_attentions:
        output['component_attentions'] = feat_output['attentions']
    if return_profile_attentions:
        output['profile_attentions'] = alpha.squeeze(-1)


    return output
```

# Hands On!

models/acf.py

@ class ACFUserNet

Item represented as  
weighted components



```
def forward(self, user_ids, profile_ids, features, profile_mask, return_component_attentions=False,
            return_profile_attentions=False, return_attentions=False):
    ...

    user = self.user_embedding(user_ids)

    profile = self.profile_embedding(profile_ids)

    features = features.flatten(start_dim=2, end_dim=3) # Add
    feat_output = self.feats(user, features, profile_mask, return_attentions=return_component_attentions)
    components = feat_output['pooled_features']

    user = self.w_u(user)
    profile_query = self.w_p(profile)
    components = self.w_x(components)

    profile_query = profile_query.permute((1,0,2))
    components = components.permute((1,0,2))

    alpha = F.relu(user + profile_query + components) # TODO: + item, Add current_item emb (?)
    alpha = self.w(alpha)

    profile_mask = profile_mask.permute((1,0))
    profile_mask = profile_mask.unsqueeze(-1)
    alpha = alpha.masked_fill(torch.logical_not(profile_mask), float('-inf'))
    alpha = F.softmax(alpha, dim=0)

    alpha = alpha.permute((1,0,2))
    user_profile = (alpha * profile).sum(dim=1)

    user = user + user_profile
    output = {'user': user}
    if return_component_attentions:
        output['component_attentions'] = feat_output['attentions']
    if return_profile_attentions:
        output['profile_attentions'] = alpha.squeeze(-1)

    return output
```

# Hands On!

models/acf.py

@ class ACFUserNet

Item-Level Attention

```
def forward(self, user_ids, profile_ids, features, profile_mask, return_component_attentions=False,
            return_profile_attentions=False, return_attentions=False):
    ...

    user = self.user_embedding(user_ids)

    profile = self.profile_embedding(profile_ids)

    features = features.flatten(start_dim=2, end_dim=3) # Add
    feat_output = self.feats(user, features, profile_mask, return_attentions=return_component_attentions)
    components = feat_output['pooled_features']

    user = self.w_u(user)
    profile_query = self.w_p(profile)
    components = self.w_x(components)

    profile_query = profile_query.permute((1,0,2))
    components = components.permute((1,0,2))

    alpha = F.relu(user + profile_query + components) # TODO: + item, Add current_item emb (?)
    alpha = self.w(alpha)

    profile_mask = profile_mask.permute((1,0))
    profile_mask = profile_mask.unsqueeze(-1)
    alpha = alpha.masked_fill(torch.logical_not(profile_mask), float('-inf'))
    alpha = F.softmax(alpha, dim=0)

    alpha = alpha.permute((1,0,2))
    user_profile = (alpha * profile).sum(dim=1)

    user = user + user_profile
    output = {'user': user}
    if return_component_attentions:
        output['component_attentions'] = feat_output['attentions']
    if return_profile_attentions:
        output['profile_attentions'] = alpha.squeeze(-1)

    return output
```

# Hands On!

models/acf.py

@ class ACFUserNet

User representation formation

```
def forward(self, user_ids, profile_ids, features, profile_mask, return_component_attentions=False,
            return_profile_attentions=False, return_attentions=False):
    ...

    user = self.user_embedding(user_ids)

    profile = self.profile_embedding(profile_ids)

    features = features.flatten(start_dim=2, end_dim=3) # Add
    feat_output = self.feats(user, features, profile_mask, return_attentions=return_component_attentions)
    components = feat_output['pooled_features']

    user = self.w_u(user)
    profile_query = self.w_p(profile)
    components = self.w_x(components)

    profile_query = profile_query.permute((1,0,2))
    components = components.permute((1,0,2))

    alpha = F.relu(user + profile_query + components) # TODO: + item, Add current_item emb (?)
    alpha = self.w(alpha)

    profile_mask = profile_mask.permute((1,0))
    profile_mask = profile_mask.unsqueeze(-1)
    alpha = alpha.masked_fill(torch.logical_not(profile_mask), float('-inf'))
    alpha = F.softmax(alpha, dim=0)

    alpha = alpha.permute((1,0,2))
    user_profile = (alpha * profile).sum(dim=1)

    user = user + user_profile
    output = (user + user_profile)
    if return_component_attentions:
        output['component_attentions'] = feat_output['attentions']
    if return_profile_attentions:
        output['profile_attentions'] = alpha.squeeze(-1)

    return output
```

# Hands On!

models/acf.py

@ class ACFFeatureNet



Component level logic

```
class ACFFeatureNet(nn.Module):
    """
    Process auxiliary item features into latent space.
    All items for user can be processed in batch.
    """
    def __init__(self, emb_dim, input_feature_dim, feature_dim, hidden_dim=None, output_dim=None):
        super().__init__()

        ...

        self.dim_reductor = nn.Linear(input_feature_dim, feature_dim)

        self.w_x = nn.Linear(feature_dim, hidden_dim)
        self.w_u = nn.Linear(emb_dim, hidden_dim)

        self.w = nn.Linear(hidden_dim, 1)

        self._kaiming_(self.w_x)
        self._kaiming_(self.w_u)
        self._kaiming_(self.w)
```



# Hands On!

models/acf.py

@ class ACFFeatureNet



Component level logic

```
def forward(self, user, components, profile_mask, return_attentions=False):
    x = self.dim_reductor(components) # Add
    x = x.movedim(0, -2) # BxPxHxD => PxHxBxD

    x_tilde = self.w_x(x)
    user = self.w_u[user]

    beta = F.relu(x_tilde + user)
    beta = self.w(beta)

    beta = F.softmax(beta, dim=1)

    x = (beta * x).sum(dim=1)
    x = x.movedim(-2, 0) # PxBxD => BxPxD

    feature_dim = x.shape[-1]
    profile_mask = profile_mask.float()
    profile_mask = profile_mask.unsqueeze(-1).expand((*profile_mask.shape, feature_dim))

    x = profile_mask * x
    output = {'pooled_features': x}
    if return_attentions:
        output['attentions'] = beta.squeeze(-1).squeeze(-1)
    return output
```

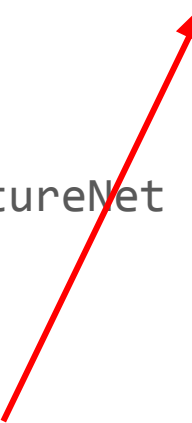


# Hands On!

models/acf.py

@ class ACFFeatureNet

**Dimensionality reduction**



```
def forward(self, user, components, profile_mask, return attentions=False):  
    x = self.dim_reductor(components) # Add  
    x = x.movedim(0, -2) # BxPxNxuD -> PxNxPxuD  
  
    x_tilde = self.w_x(x)  
    user = self.w_u[user]  
  
    beta = F.relu(x_tilde + user)  
    beta = self.w(beta)  
  
    beta = F.softmax(beta, dim=1)  
  
    x = (beta * x).sum(dim=1)  
    x = x.movedim(-2, 0) # PxPxPxuD => BxPxPxuD  
  
    feature_dim = x.shape[-1]  
    profile_mask = profile_mask.float()  
    profile_mask = profile_mask.unsqueeze(-1).expand((*profile_mask.shape, feature_dim))  
  
    x = profile_mask * x  
    output = {'pooled_features': x}  
    if return attentions:  
        output['attentions'] = beta.squeeze(-1).squeeze(-1)  
    return output
```

# Hands On!

models/acf.py

@ class ACFFeatureNet

Component-Level Attention



```
def forward(self, user, components, profile_mask, return_attentions=False):
    x = self.dim_reductor(components) # Add
    x = x.movedim(0, -2) # BxPxHxD => PxHxBxD

    x_tilde = self.w_x(x)
    user = self.w_u[user]

    beta = F.relu(x_tilde + user)
    beta = self.w(beta)

    beta = F.softmax(beta, dim=1)

    x = (beta * x).sum(dim=1)
    x = x.movedim(-2, 0) # PxBxD => BxPxD

    feature_dim = x.shape[-1]
    profile_mask = profile_mask.float()
    profile_mask = profile_mask.unsqueeze(-1).expand((*profile_mask.shape, feature_dim))

    x = profile_mask * x
    output = {'pooled_features': x}
    if return_attentions:
        output['attentions'] = beta.squeeze(-1).squeeze(-1)
    return output
```

# Hands On!

trainers/acf\_trainer.py

@ class ACFTrainer



Training logic

```
class ACFTrainer():
    """
    Handles training process
    """
    def __init__(self, model, datasets, loss, optimizer, version, batch_size=100, device=None,
                 max_profile_size=9, checkpoint_dir=None):
        """
        Parameters
        -----
        model: initialized UserNet
        dataset: initialized MovieLens
        loss: one of the warp functions
        optimizer: torch.optim
        run_name: directory to save results
        batch_size: number of samples to process for one update
        device: gpu or cpu
        """
        self.pad_token = 0

        self.version = version
        self.epoch = 0
        self.best_loss = np.inf
        self.loss = loss
        self.optimizer = optimizer
        self.batch_size = batch_size

        self.device = get_device(device)
        self.model = model
        self.model = self.model.to(self.device)

        self.train, self.test = datasets
        self.all_items = self.preprocess_inputs(self.train.items, to_tensor=True)

        self.train_loader = DataLoader(self.train, batch_size=batch_size, shuffle=True,
                                       collate_fn=generate_collate_fn(max_profile_size), num_workers=8)
        self.test_loader = DataLoader(self.test, batch_size=batch_size, shuffle=True,
                                       collate_fn=generate_collate_fn(max_profile_size), num_workers=1)
```

# Hands On!

trainers/acf\_trainer.py

@ class ACFTrainer



Training logic

```
def fit(self, num_epochs, k=10):
    num_train_batches = len(self.train) / self.batch_size
    num_test_batches = len(self.test) / self.batch_size
    for epoch in tqdm(range(num_epochs)):
        self.epoch = epoch
        for phase in ['train', 'val']:
            self.logger.epoch(epoch, phase)
            self.model.train(phase == 'train')
            loss = 0
            cur_step = 0
            if phase == 'train':
                t = tqdm(self.train_loader)
                for batch in t:
                    self.optimizer.zero_grad()
                    cur_loss = self.training_step(batch)
                    self.optimizer.step()
                    loss += cur_loss
                    cur_step += 1
                avg_loss = loss / cur_step

                t.set_description(f"Average Loss {avg_loss:.4f}")
                t.refresh()

            loss /= num_train_batches
            self.logger.metrics(loss, 0, epoch, phase)
        else:
            with torch.no_grad():
                for batch in tqdm(self.test_loader):
                    cur_loss = self.validation_step(batch)
                    loss += cur_loss
                loss /= num_test_batches
                # self.logger.metrics(loss, self.score(k=k), epoch, phase)
                self.logger.metrics(loss, 0.0, epoch, phase)

            if loss < self.best_loss:
                self.best_loss = loss
                self.logger.save(self.state, epoch)
```

# Hands On!

trainers/acf\_trainer.py

@ class ACFTTrainer

trainers/loss.py

Weston, J., Bengio, S., & Usunier, N. (2011). Wsabie: Scaling up to large vocabulary image annotation.

```
def training_step(self, batch):
    user_id, profile_ids, pos, neg = self.preprocess_inputs(*batch)
    profile_mask = self.get_profile_mask(profile_ids)
    pos_pred, neg_pred = self.model(user_id, profile_ids, pos, neg, profile_mask)

    loss = self.loss(pos_pred, neg_pred)
    loss.backward()
    return loss.item()
```

```
def bpr_loss(pos, neg, b=0.0, collapse=True):
    res = torch.sigmoid(neg - pos + b)
    if collapse:
        res = res.mean()
    return res

def warp_loss(pos, neg, b=1, collapse=True):
    loss = bpr_loss(pos, neg, b, collapse=False)
    m = (loss > 0.5).float()
    m *= torch.log(m.sum() + 1) + 1
    res = m * loss
    if collapse:
        res = res.mean()
    return res
```

