

CS 330 Autumn 2023/2024 Warmup Homework 0

Multitask Training for Recommender Systems

Due Wednesday October 4, 11:59 PM PST

SUNet ID:
Name:
Collaborators:

0 Honor Code

I agree by the Stanford honor code and declare that I will not view online (e.g. GitHub) or other students' solutions and post solutions online. I declare that all of my submissions are my own works.

Please take a moment to type the above statement and your signature. **This serves for all of your future homework as well.**

Statement:

Name: Date:

Use of GPT/Codex/Copilot: For the sake of deeper understanding on implementing imitation learning methods, assistance from generative models to write code for this homework is prohibited.

Please be aware that we will be actively monitoring adherence to these guidelines. This addition to our course policy serves not only to maintain the integrity of our academic environment but also to reduce the number of potential honor code violations. Thank you all for your dedication to maintaining the highest standards of academic integrity!

1 Overview

In this assignment, we will implement a multi-task movie recommender system based on the classic Matrix Factorization [?] and Neural Collaborative Filtering [?] algorithms. In particular, we will build a model based on the **BellKor solution** to the Netflix Grand Prize challenge and extend it to predict both likely user-movie interactions and potential scores. In this assignment you will implement a multi-task neural network architecture and explore the effect of parameter sharing and loss weighting on model performance.

The main goal of these exercises is to familiarize yourself with multi-task architectures, the training pipeline, and coding in PyTorch. These skills will be important in the course.

Note: This assignment is a warmup, and is shorter than future homeworks will be.

Submission: To submit your work, submit one pdf report and one zip file to GradeScope, where the report contains answers to the deliverables listed below and the zip file contains the code with your filled-in solutions.

Code Overview: The code consists of several files; however, you will only need to interact with two:

- `main.py`: To run experiments, execute this file by passing the corresponding parameters.
- `models.py`: This file contains our multi-task prediction model **MultiTaskNet**, which you will need to finish implementing in PyTorch.

2 Dataset and Evaluation

Dataset. In this assignment, we will use movie reviews from the **MovieLense dataset**. The dataset consists of 100K reviews of 1700 movies generated by 1000 users. Although each user interaction contains several levels of meta-data, we'll only consider tuples of the type **(userID, itemID, rating)**, which contain an anonymized user ID, movie ID and the score assigned by the user to the movie from 1 to 5. We randomly split the dataset into a **train** dataset, which contains 95% of all ratings, and a **test** dataset, which contains the remaining 5%.

Problem Definition. Given the dataset defined above, we would like to train a model $f(\text{userID}, \text{itemID})$ that predicts: 1) the probability p that the user would watch the movie and 2) the score r they would assign to it from 1 to 5. For some intuition on this setting, consider a user who only watches comedy and action movies. It would not make sense to recommend them a horror movie since they don't watch those. At the same time, we would want to recommend comedy or action movies that the user is likely to score highly.

Evaluation. Once we have our trained model, we evaluate it on the test set.

Score Prediction. We will evaluate the mean-squared error of movie score prediction on the held-out user ratings, i.e. $\frac{1}{N} \sum_{i=1}^N \|\hat{r}_i - r_i\|^2$, where \hat{r}_i is the predicted score for user-movie pair $(\text{userID}_i, \text{itemID}_i)$. The summation is over all pairs in the test set. Better models achieve lower mean-squared errors.

Likelihood Prediction. To evaluate the quality of the likelihood model, we use the **mean reciprocal rank metric**, which provides a higher score for highly ranking the movies the user has seen. The metric is computed as follows: 1) for each user, rank all movies based on the probability that the user would watch them; 2) remove movies we know the user has watched (those in the training set); 3) compute the average reciprocal ranking of movies the user has watched from the held-out set.

3 Problems

To install all required packages for this assignment you can run:

`pip install -r requirements.txt.`

In this problem, we will implement a multi-task model using Matrix Factorization [?] and regression-based modelling:

Matrix Factorization: Consider an interaction matrix M , where $M_{ij} = 1$ if userID_i has rated movie with itemID_j and 0 otherwise. We will represent each user with a latent vector $\mathbf{u}_i \in \mathbb{R}^d$ and each item with a latent vector $\mathbf{q}_j \in \mathbb{R}^d$. We model the interaction probability $p_{ij} = \log P(M_{ij} = 1)$ in the following way:

$$p_{ij} = \mathbf{u}_i^T \mathbf{q}_j + b_j \quad (1)$$

where b_j is a movie-specific bias term. At each training step we sample a batch of triples $(\text{userID}_i, \text{itemID}_j^+, \text{itemID}_{j'}^-)$ with size B , such that $M_{i,j} = 1$, while $\text{itemID}_{j'}^-$ is randomly sampled (indicating no user preference). Let

$$\begin{aligned} p_{ij}^+ &= \mathbf{u}_i^T \mathbf{q}_j + b_j \\ p_{ij'}^- &= \mathbf{u}_i^T \mathbf{q}_{j'} + b_{j'} \end{aligned} \quad (2)$$

and optimize the Bayesian Personalised Ranking (BPR) [?] pairwise loss function:

$$\mathcal{L}_F(\mathbf{p}^+, \mathbf{p}^-) = \frac{1}{B} \sum_{i=1}^B 1 - \sigma(p_{ij}^+ - p_{ij'}^-) \quad (3)$$

where σ is the sigmoid function.

Regression Model: For training the regression model, we consider only batches of tuples $(\text{userID}_i, \text{itemID}_j^+, r_{ij})$, such that $M_{i,j} = 1$ and r_{ij} is the numerical rating userID_i assigned to itemID_j^+ . Using the same latent vector representations as before, we will concatenate $[\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j]$ (where $*$ denotes element-wise multiplication) together and pass it through a neural network with a single hidden layer:

$$\hat{r}_{ij} = f_\theta([\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j]) \quad (4)$$

We train the model using the mean-squared error loss:

$$\mathcal{L}_R(\hat{\mathbf{r}}, \mathbf{r}) = \frac{1}{B} \sum_{i=1}^B \|\hat{r}_{ij} - r_{ij}\|^2 \quad (5)$$

3.1 [14 total points (Coding)] Your Implementation

A. Implement MultitaskNet Model:

The first part of the assignment is to implement the above model in `models.py`. First you need to define each component when the model is initialized.

1. **[3 points (Coding)]** Consider the matrix $\mathbf{U} = [\mathbf{u}_1, \dots, \mathbf{u}_{N_{\text{users}}}] \in \mathbb{R}^{N_{\text{users}} \times d}$, $\mathbf{Q} = [\mathbf{q}_1, \dots, \mathbf{q}_{N_{\text{items}}}] \in \mathbb{R}^{N_{\text{items}} \times d}$, $\mathbf{B} = [b_1, \dots, b_{N_{\text{items}}}] \in \mathbb{R}^{N_{\text{items}} \times 1}$. Implement \mathbf{U} and \mathbf{Q} as ScaledEmbedding layers with parameter $d = \text{embedding_dim}$ and \mathbf{B} as ZeroEmbedding layers with parameter $d = 1$ (defined in `models.py`). These are instances of **PyTorch Embedding** layers with a different weight initialization, which facilitates better convergence.

Specifically, please complete the following functions in `models.py`:

- `init_shared_user_and_item_embeddings`
 - `init_separate_user_and_item_embeddings`
 - `init_item_bias`
2. **[2 points (Coding)]** Next implement $f_{\theta}([\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j])$ as an MLP network. The class `MultiTaskNet` has `layer_sizes` argument, which is a list of the input shapes of each dense layer. Notice that by default `embedding_dim=32`, while the input size of the first layer is 96, since we concatenate $[\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j]$ before processing it through the network. Each layer (except the final layer) should be followed by a ReLU activation. The final layer should output the final user-item predicted score in and have an output size of 1. Specifically, please complete the function `init_mlp_layers` in `models.py`.

B. Implement Forward [9 points (Coding)]:

In the second part of the problem you need to implement the forward method of the `MultiTaskNet` module. The forward method receives a batch of $(\text{userID}_i, \text{itemID}_j)$ of user-item pairs. The model should output a probability p_{ij} of shape $(\text{batch_size},)$ that user i would watch movie j , given by Eq. 1 and a predicted score \hat{r}_{ij} of shape $(\text{batch_size},)$ the user i would assign to movie j , given by Eq. 4. Note that you do not need to compute the entire user-item interaction matrix M defined above. Here, you can simply assume user index i and item index j are always the same and predict the interaction and score for $(\text{user}[1] \text{ w.r.t item}[1]), \dots, (\text{user}[\text{batch_size}] \text{ w.r.t item}[\text{batch_size}])$.

Moreover, the `MultiTaskNet` class has an `embedding_sharing` attribute. Implement your model in such a way that when `embedding_sharing=True` a single latent vector representation is used for both the factorization and regression tasks and vice versa. **Be careful with output tensor shapes!**

Specifically, please complete the following functions in `models.py`:

- `forward_with_embedding_sharing`
- `forward_without_embedding_sharing`

Optional. Autograding Your Code. In this homework, we include autograding functionalities in the released code to facilitate you to debug and develop your code. To run the autograder, simply do:

```
python grader.py
```

The maximum points you can get when running the autograder is **8 / 8 points**. We also have **6 points** from hidden test cases that show up when you submit your code to Gradescope. This makes the total of **14 points** for the Coding part.

4 Write-up

4.1 [8 total points (Plot)] Plot Comparison

To execute experiments run the `main.py` script, which will automatically log training MSE loss, BPR loss and test set MSE loss and MRR scores to **TensorBoard**. Please do

```
tensorboard --logdir run
```

to visualize the losses in tensorboard. Once you're done with your implementation run the following 4 experiments:

1. **[2 points (Plot)]** Evaluate a model with shared representations and task weights $\lambda_F = 0.99, \lambda_R = 0.01$. You can run this experiment by running:

```
python main.py --factorization_weight 0.99 --regression_weight 0.01  
--logdir run/shared=True_LF=0.99_LR=0.01
```

Here the `--factorization_weight` and `--regression_weight` arguments correspond to λ_F and λ_R respectively.

2. **[2 points (Plot)]** Evaluate a model with shared representations and task weights $\lambda_F = 0.5, \lambda_R = 0.5$. You can run this experiment by running:

```
python main.py --factorization_weight 0.5 --regression_weight 0.5  
--logdir run/shared=True_LF=0.5_LR=0.5
```

3. **[2 points (Plot)]** Evaluate a model with **separate** representations and task weights $\lambda_F = 0.5, \lambda_R = 0.5$. You can run this experiment by running:

```
python main.py --no_shared_embeddings --factorization_weight 0.5  
--regression_weight 0.5 --logdir run/shared=False_LF=0.5_LR=0.5
```

4. **[2 points (Plot)]** Evaluate a model with **separate** representations and task weights $\lambda_F = 0.99, \lambda_R = 0.01$. You can run this experiment by running:

```
python main.py --no_shared_embeddings --factorization_weight 0.99  
--regression_weight 0.01 --logdir run/shared=False_LF=0.99_LR=0.01
```

Your plots go here: For each experiment include a screenshot of Tensorboard graphs for the training and test set losses in your write up.

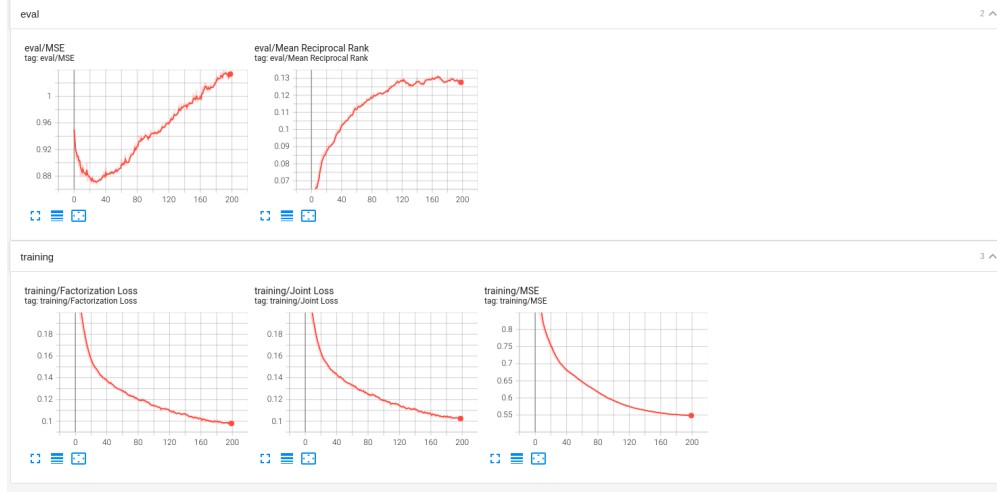


Figure 1: Loss of train and evaluation for shared representation with $\lambda_F = 0.99, \lambda_R = 0.01$.

1. Evaluate a model with shared representations and task weights $\lambda_F = 0.99, \lambda_R = 0.01$, as shown in figure 1.
2. Evaluate a model with shared representations and task weights $\lambda_F = 0.5, \lambda_R = 0.5$, as shown in figure 1.

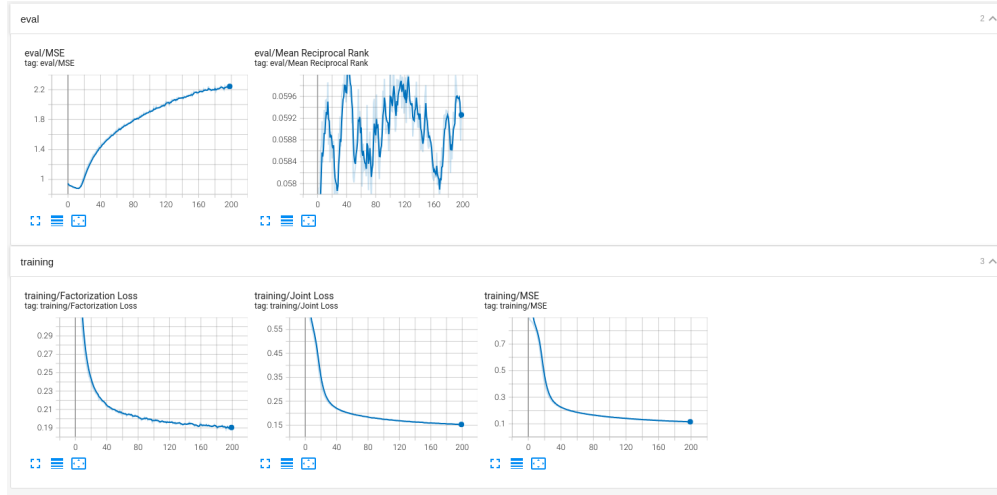


Figure 2: Loss of train and evaluation for shared representation with $\lambda_F = 0.5, \lambda_R = 0.5$.

3. Evaluate a model with **separate** representations and task weights $\lambda_F = 0.5, \lambda_R = 0.5$, as shown in figure 3.
4. Evaluate a model with **separate** representations and task weights $\lambda_F = 0.99, \lambda_R = 0.01$, as shown in figure 4.

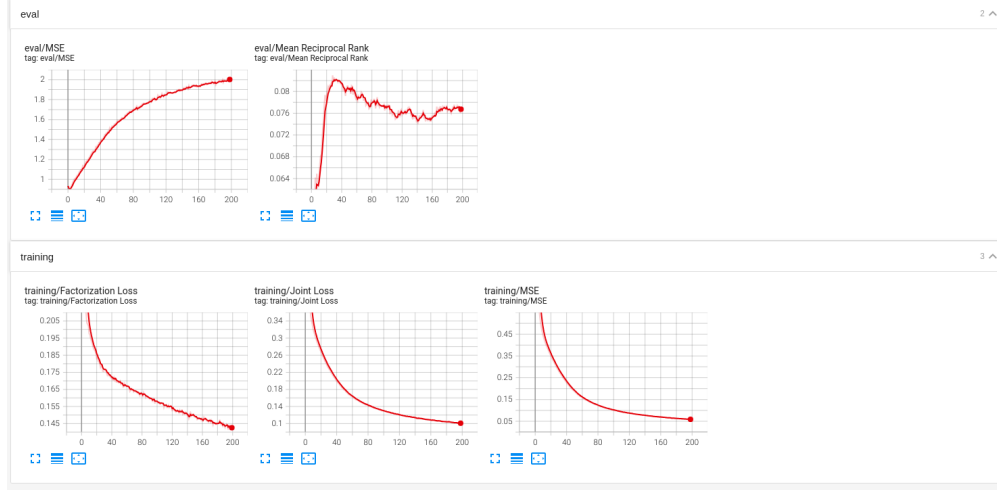


Figure 3: Loss of train and evaluation for separate representation with $\lambda_F = 0.5, \lambda_R = 0.5$.

4.2 [8 total points (Written)] Analysis

Answer the following questions:

1. [2 points (Written)] In Eqn. 1, we only include a movie-specific bias term b_j . Does it make sense to also include a user-specific bias term a_i ? Specifically, if we define p_{ij} in the following way:

$$p_{ij} = \mathbf{u}_i^T \mathbf{q}_j + a_i + b_j, \quad (6)$$

will the model capacity increase or remain the same?

2. [2 points (Written)] Consider the case with $\lambda_F = 0.99$ and $\lambda_R = 0.01$. Based on the train/test loss curves, does parameter sharing outperform having separate models? Please provide a brief justification to your answer.
3. [2 points (Written)] Now consider the case with $\lambda_F = 0.5$ and $\lambda_R = 0.5$. Based on the train/test loss curves, does parameter sharing outperform having separate models? Please provide a brief justification to your answer.
4. [2 points (Written)] In the **shared model setting** compare results for $\lambda_F = 0.99$ and $\lambda_R = 0.01$ and $\lambda_F = 0.5$ and $\lambda_R = 0.5$, can you explain the difference in performance? Please provide a brief justification to your answer.

Your answers go here:

1. Theoretically, adding another bias item should not improve the performance although this action increases the number of parameters. Figure 5, 6, 7, 8 shows the results of adding user bias corresponding to the experiments before. It shows that adding another bias parameter does NOT help improve the model capacity, as we expected.

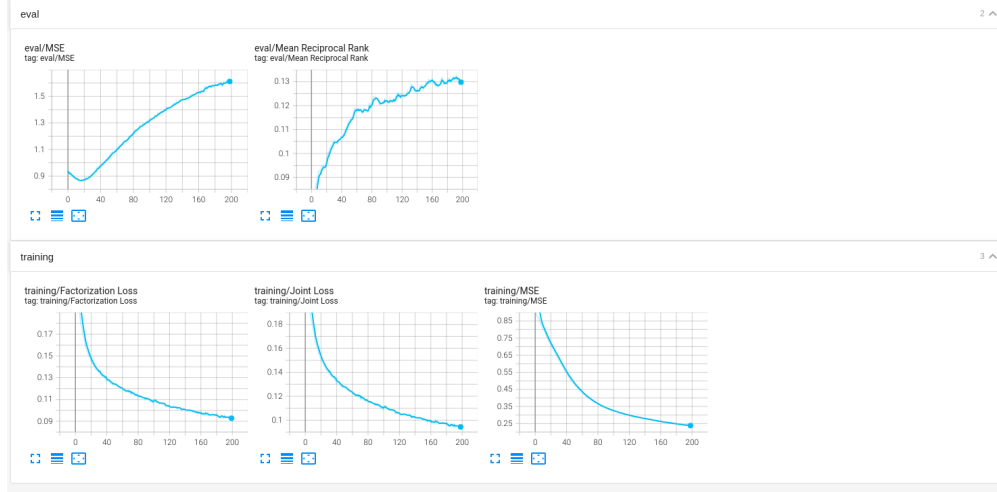


Figure 4: Loss of train and evaluation for separate representation with $\lambda_F = 0.99$, $\lambda_R = 0.01$.

2. Figure 9 shows the comparison results. From training loss, parameter sharing seems to underperform parameter separation. From the evaluation of MSE (Mean Squared Error), parameter sharing performs better. It may be attributed to the model with separate parameters trying to minimize factorization loss too well, and we give too many weights to optimizing factorization loss.
3. From figure 10, sharing parameters performs worse than separating parameters.
4. Figure 11 shows the results. It can be seen that model with $\lambda_F = 0.99$ and $\lambda_R = 0.01$ performs better on both evaluation metrics and first 2 training loss, however, the model with $\lambda_F = 0.5$ and $\lambda_R = 0.5$ has better performance balance between different tasks. The model with $\lambda_F = 0.99$ and $\lambda_R = 0.01$ also shows better generalization despite its imbalanced performance during training, it's hard to see whether it's overfitting.

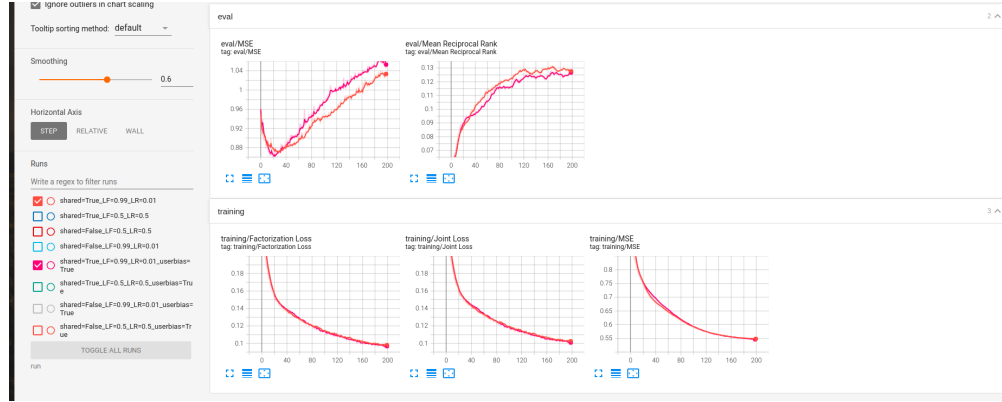


Figure 5: Loss of train and evaluation for shared representation with $\lambda_F = 0.99$, $\lambda_R = 0.01$. In this case, we add another user bias for matrix factorization.

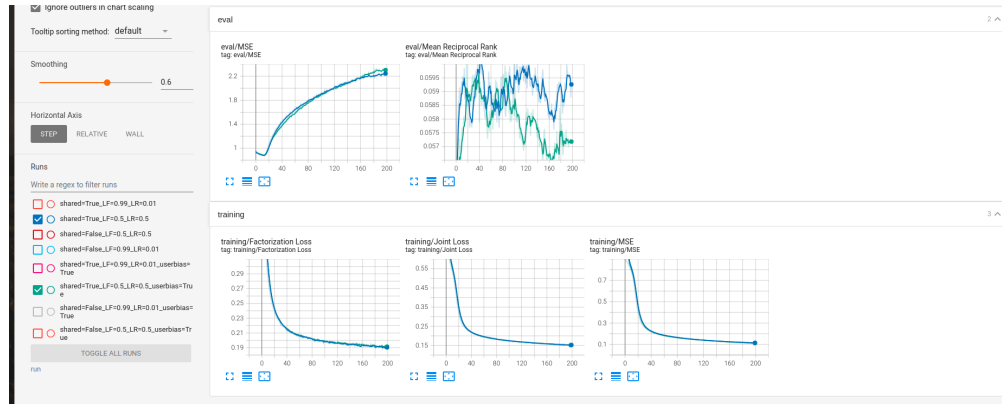


Figure 6: Loss of train and evaluation for shared representation with $\lambda_F = 0.5$, $\lambda_R = 0.5$. In this case, we add another user bias for matrix factorization.

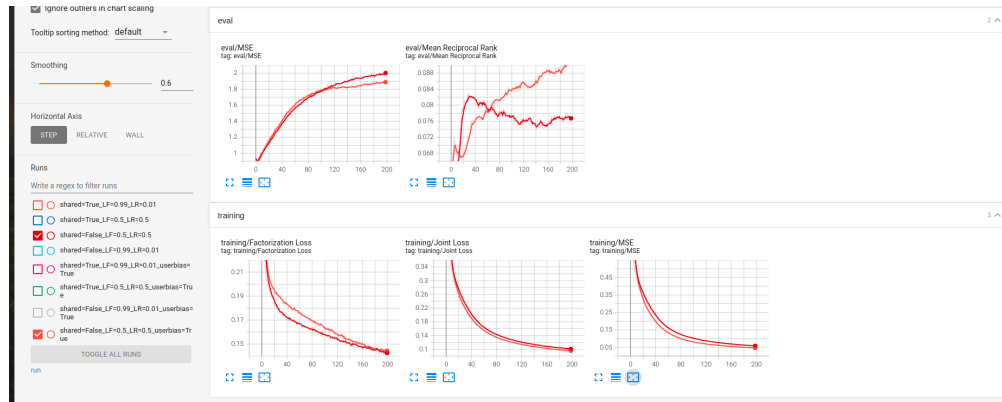


Figure 7: Loss of train and evaluation for separate representation with $\lambda_F = 0.5$, $\lambda_R = 0.5$. In this case, we add another user bias for matrix factorization.

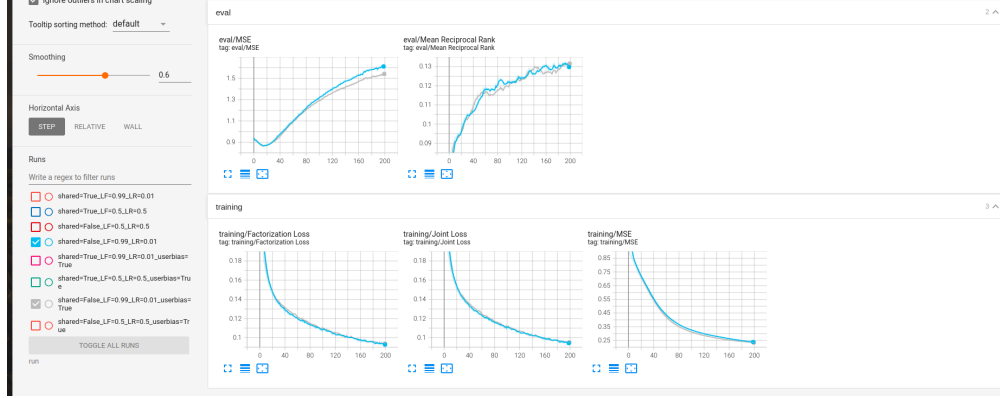


Figure 8: Loss of train and evaluation for separate representation with $\lambda_F = 0.99$, $\lambda_R = 0.01$. In this case, we add another user bias for matrix factorization.

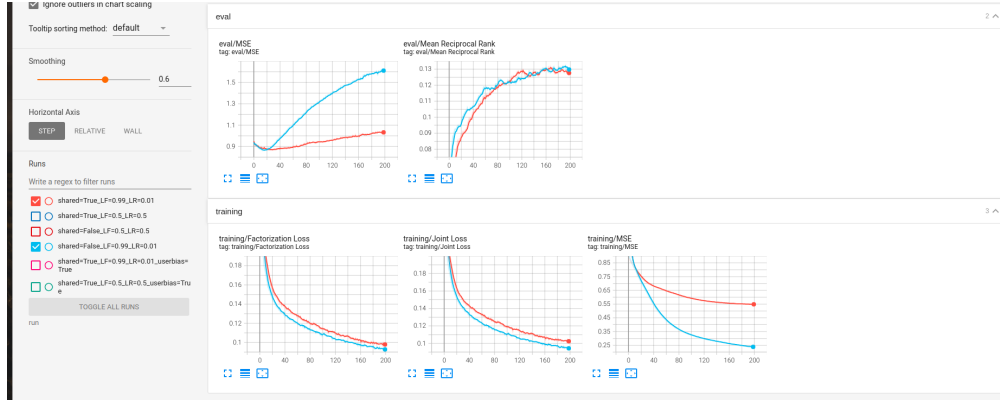


Figure 9: The comparison between parameter sharing or not, $\lambda_F = 0.99$ and $\lambda_R = 0.01$

References

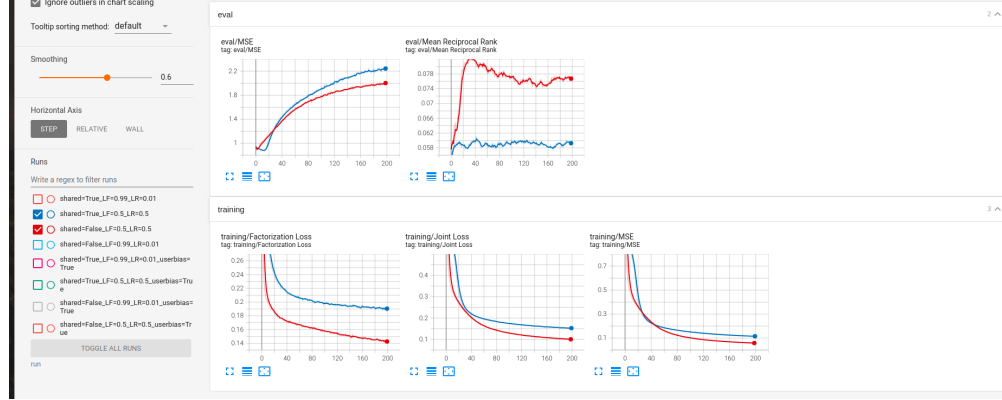


Figure 10: The comparison between parameter sharing or not, $\lambda_F = 0.5$ and $\lambda_R = 0.5$

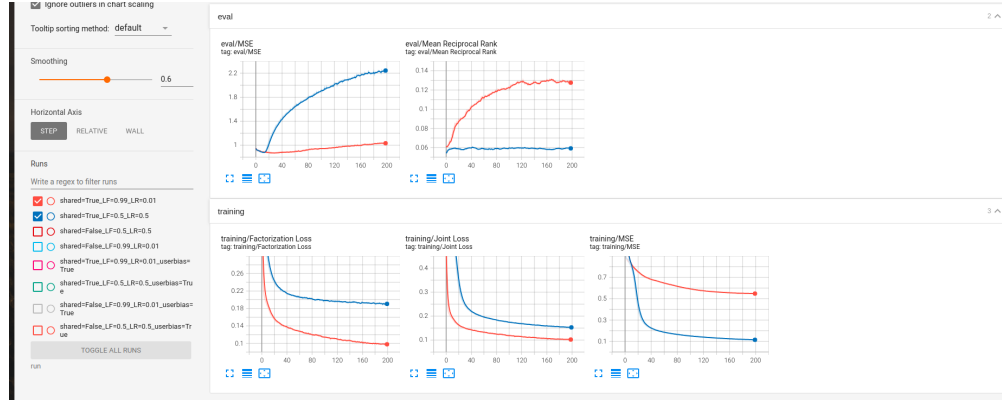


Figure 11: The comparison between different loss weight settings, $\lambda_F = 0.5$ and $\lambda_R = 0.5$; $\lambda_F = 0.99$ and $\lambda_R = 0.01$