Data Intelligence

Deep Matrix Factorization - Lab

U Kang
Seoul National University

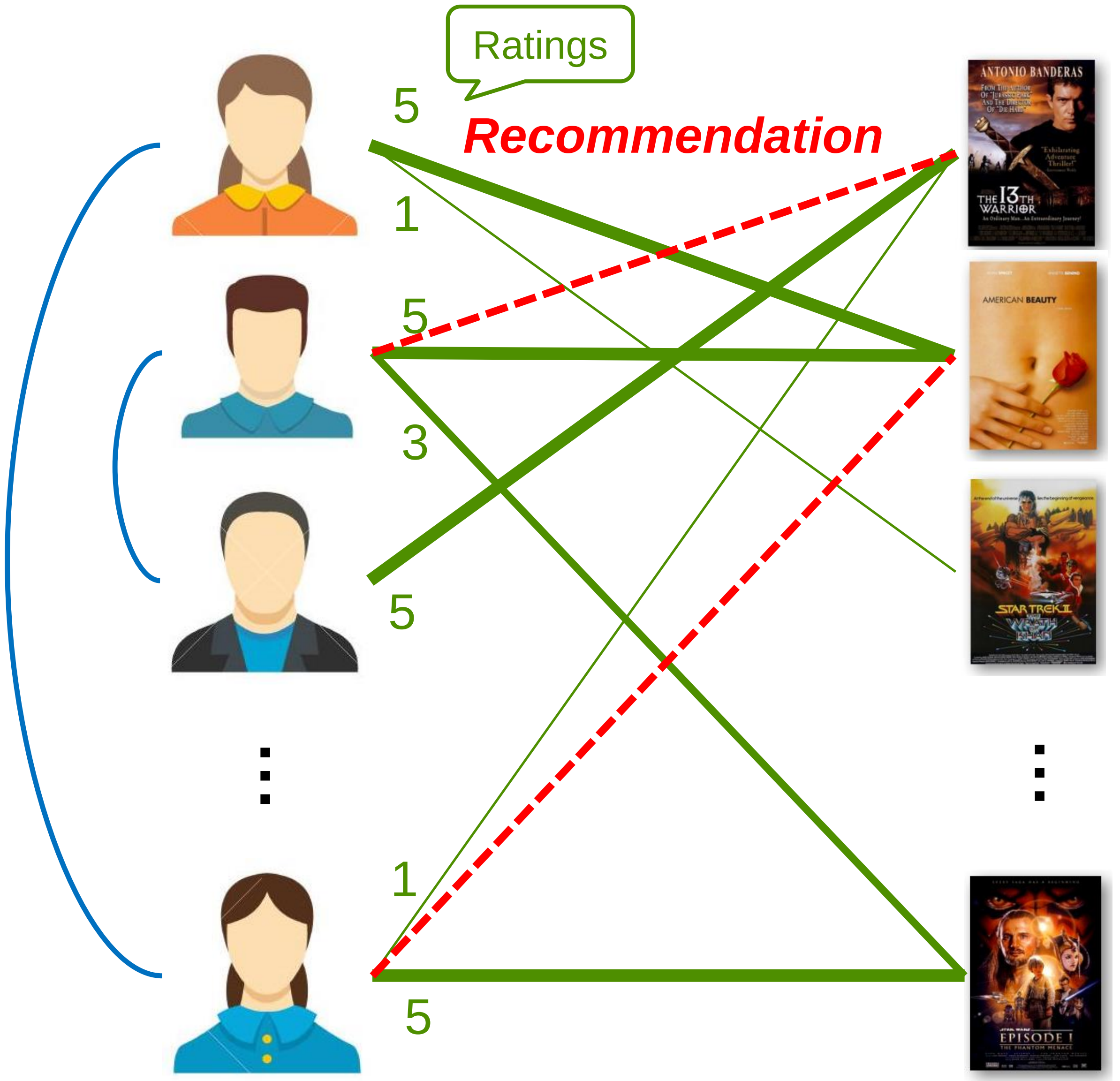
In This Lecture

- Deep Matrix Factorization
  - Recommendation
  - Deep Learning based matrix factorization
  - Data preparation
  - Model implementation
  - Model training / evaluation

Outline

- ☐ **Recommendation**
- ☐ Deep Learning based Matrix Factorization
- ☐ Data Preparation
- ☐ Model Implementation
- ☐ Model Training / Evaluation

Recommendation



Matrix Completion

- Matrix Completion is a surrogate problem of recommendation
  - Users want to be provided items that they will give high ratings
- Matrix Completion
  - ***Given***: a sparse rating matrix R
  - ***Goal***: to predict unseen rating values in R

R matrix

User \ Item	1	2	3	4	5
1	**5**	?	?	**2**	**1**
2	?	?	**2**	?	?
3	**3**	?	?	?	?
4	?	**3**	**1**	?	?
5	**5**	?	?	**2**	?

User-Movie Matrix

Outline

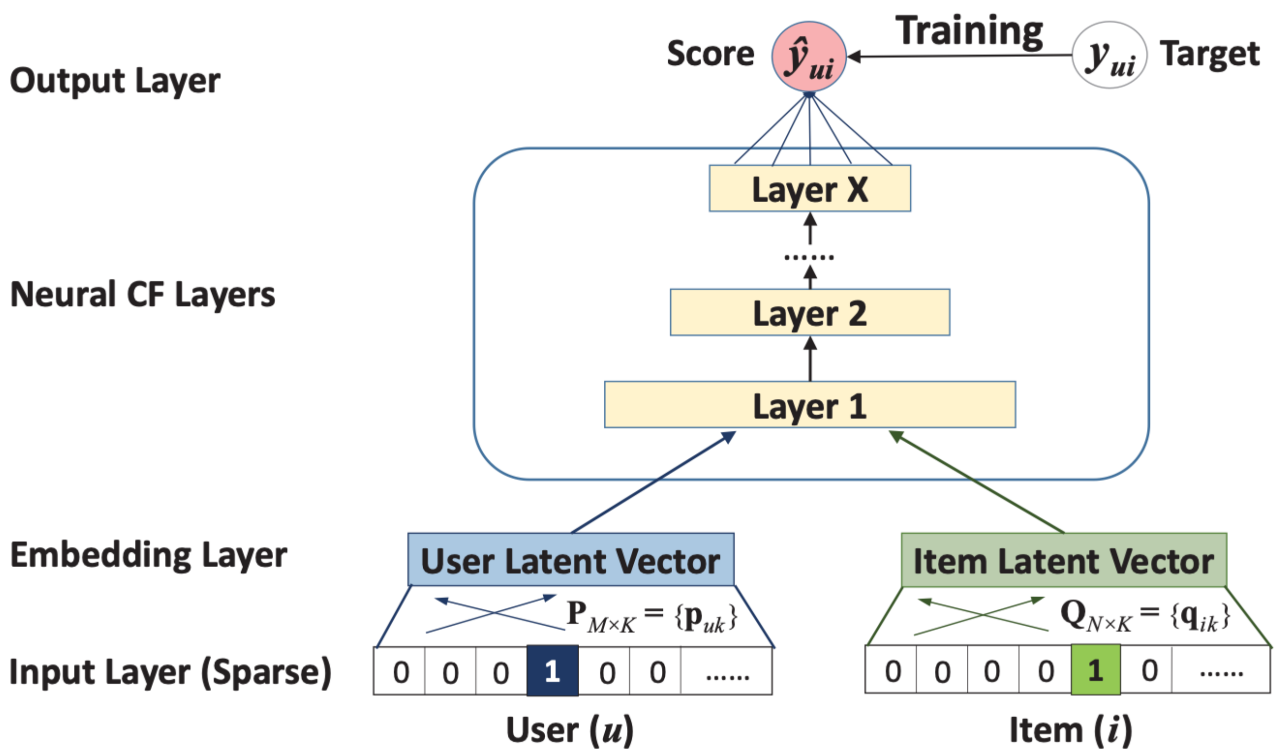
- ☑ Recommendation
- ☐ **Deep Learning based Matrix Factorization**
- ☐ Data Preparation
- ☐ Model Implementation
- ☐ Model Training / Evaluation

Matrix Factorization

- Limitations of matrix factorization
  - A linear model
    - It cannot model non-linear relationships
  - Limited parameters
    - Only user and item latent factors are trained
  - Simple modeling
    - It expresses user-item relationships by the inner-product which is too simple to model complicated real-world interactions

Deep Matrix Factorization (1)

- Deep learning based matrix factorization



He, Xiangnan, et al. "Neural collaborative filtering." *WWW*. 2017.

Deep Matrix Factorization (2)

- Deep learning based matrix factorization
  - A non-linear model
    - It models non-linear relationships
  - Wide range of parameters
    - It learns not only user and item latent factors, but also user-item relationships
  - Complex modeling
    - It can express complicated user-item relationships through deep neural networks

Outline

- ☑ Recommendation
- ☑ Deep Learning based Matrix Factorization
- ☐ **Data Preparation**
- ☐ Model Implementation
- ☐ Model Training / Evaluation

PyTorch

- PyTorch (https://pytorch.org/)
  - A deep learning framework
    - It supports GPU computations
    - Model training (back-propagation) can be done easily
  - Numpy friendly
    - The usage is similar to Numpy
  - Includes a lot of useful functions
    - Dataloader
    - Model save/load
    - Initialization
    - Visualization
    - …

Reading Data File (1)

- Set the data path and split ratio
  - "ratings.dat" contains interaction logs

```
import numpy as np
```

```
in_path = './data/ml-1m-raw/'
rating_file = in_path + 'ratings.dat'
```

Reading Data File (2)

- Read data file
  - Format of “ratings.dat”
    - user_id::item_id::rating::time_stamp

```
raw = []
with open(rating_file, 'r') as f_read:
    for line in f_read.readlines():
        line_list = line.split('::')
        raw.append(line_list)
raw = np.array(raw, dtype=np.int)
```

Assign New IDs

- We need new ids that start from 0

```
user_ids = list()
item_ids = list()
user_map = dict() # raw -> new
item_map = dict() # raw -> new

user_ids = np.unique(raw[:, 0])
item_ids = np.unique(raw[:, 1])

user_map = {v: i for (i, v) in enumerate(user_ids)}
item_map = {v: i for (i, v) in enumerate(item_ids)}

new = [[user_map[u], item_map[i], r] for (u, i, r)
       in zip(raw[:, 0], raw[:, 1], raw[:, 2])] # new array
new = np.array(new)
print(new.shape)
```

```
(1000209, 3)
```

Split Dataset

- Randomly split the dataset into training/test sets

```
from sklearn.model_selection import train_test_split
```

```
train, test = train_test_split(new, test_size=0.2, shuffle=True, random_state=42)
```

```
print(train.shape)
print(test.shape)
```

```
(800167, 3)
(200042, 3)
```

Dataset and Data Loader (1)

```
from torch.utils.data import Dataset, DataLoader
```

- PyTorch supports custom dataset and data loader
  - Dataset
    - We should define "__len__()" and "__getitem__()" functions
      - "__len__()": Length of the dataset
      - "__getitem__()": How to get data when an index is given
  - DataLoader
    - We should define batch size and whether to shuffle the data

Dataset and Data Loader (2)

- Define custom dataset

```
class MovieLensDataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        x = torch.LongTensor(self.x[idx, :])
        y = torch.FloatTensor(self.y[idx, :])
        return x, y
```

  - **x** and **y** are given to the object
  - __len__() returns the length of the dataset
  - __getitem__() returns the **x** and **y** at the given **idx**

Dataset and Data Loader (3)

- Make train/test datasets and data loaders

```
train_dataset = MovieLensDataset(train[:, :-1], np.expand_dims(train[:, -1], axis=1))
train_dataloader = DataLoader(train_dataset, batch_size=512, shuffle=True)
```

```
test_dataset = MovieLensDataset(test[:, :-1], np.expand_dims(test[:, -1], axis=1))
test_dataloader = DataLoader(test_dataset, batch_size=len(test_dataset), shuffle=False)
```

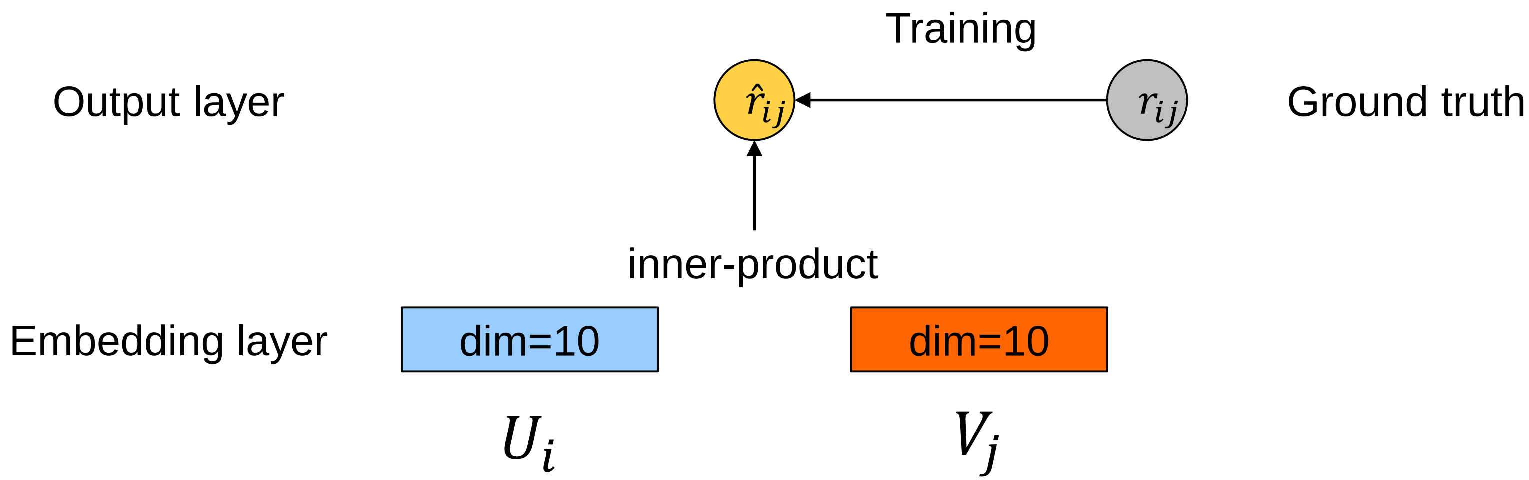
  - The data loaders will yield a batch of size “batch_size” whenever they are called
  - We set batch_size to 512

Outline

- ☑ Recommendation
- ☑ Deep Learning based Matrix Factorization
- ☑ Data Preparation
- ☐ **Model Implementation**
- ☐ Model Training / Evaluation

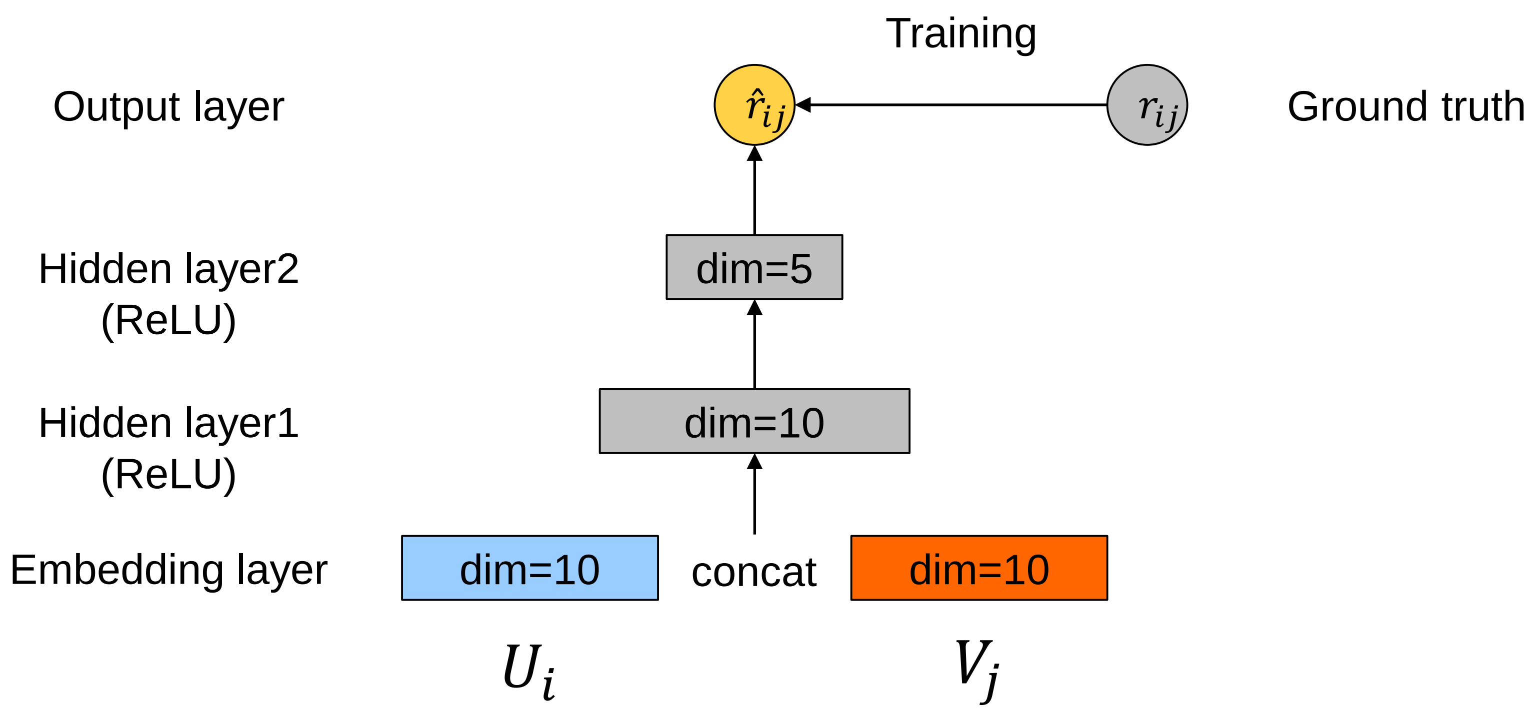
Model (1)

- We compare two models: MF and DeepMF
  - MF: a simple linear model



Model (2)

- We compare two models: MF and DeepMF
  - DeepMF: 3-layered fully-connected neural network



Hyper-parameters

- Let's define hyper-parameters:

```
num_users = max(max(train[:, 0]), max(test[:, 0])) + 1
num_items = max(max(test[:, 1]), max(test[:, 1])) + 1
```

```
K = 10
hidden_dim1 = 10
hidden_dim2 = 5
lr = 1e-3
decay = 1e-5
epochs = 100
```

  - K: embedding vector dimensionality
  - hidden_dim: hidden layer dimensionality
  - lr: learning rate
  - decay: weight decay
  - epochs: number of epochs

Import Library

- Let’s import libraries:

```
import torch
import time
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

  - We can choose whether to use GPU (CUDA)
  - torch.cuda.is_available(): whether cuda is available

Define Models (1)

- Let's define Matrix Factorization:

```python
class MF(torch.nn.Module):
    def __init__(self, num_users, num_items, K):
        super().__init__()
        self.user_emb = torch.nn.Embedding(num_users, K)
        self.item_emb = torch.nn.Embedding(num_items, K)

    def forward(self, user_idx, item_idx):
        out = (self.user_emb(user_idx) * self.item_emb(item_idx)).sum(1, keepdim=True)
        return out
```

  - torch.nn.Embedding: embedding parameters
  - We should define "forward" function which is used in prediction

Define Models (2)

- Let's define Deep Matrix Factorization:

```
class DeepMF(torch.nn.Module):
    def __init__(self, num_users, num_items, K, hidden_dim1, hidden_dim2):
        super().__init__()
        self.user_emb = torch.nn.Embedding(num_users, K)
        self.item_emb = torch.nn.Embedding(num_items, K)
        self.layer1 = torch.nn.Linear(2*K, hidden_dim1)
        self.layer2 = torch.nn.Linear(hidden_dim1, hidden_dim2)
        self.out = torch.nn.Linear(hidden_dim2, 1)
        self.activation = torch.nn.ReLU()

    def forward(self, user_idx, item_idx):
        out = torch.cat((self.user_emb(user_idx), self.item_emb(item_idx)), dim=1)
        out = self.activation(self.layer1(out))
        out = self.activation(self.layer2(out))
        out = self.out(out)
        return out
```

  - torch.nn.Embedding: embedding parameters
  - torch.nn.Linear: linear layer with bias
  - torch.nn.ReLU: activation function

Make Models

- Make one of the two models: MF and DeepMF

```
model = MF(num_users, num_items, K)
# model = DeepMF(num_users, num_items, K, hidden_dim1, hidden_dim2)
model.to(DEVICE)
```

  - torch.nn.Module class supports “to” function
    - It loads the parameters to CPU or GPU

Loss and Optimizer

```
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=decay)
```

- We use Mean Squared Error (MSE) as the loss
  - $MSE = \frac{\sum_u \sum_i (R_{ui} - \hat{R}_{ui})^2}{|ratings|}$
  - where
    - R_{ui} is a ground-truth rating value
    - $\hat{R}_{ui}$ is a predicted rating value by the model
  - PyTorch supports the MSE loss
- We use Adam optimizer
  - Parameters should be given to the optimizer

Outline

- ☑ Recommendation
- ☑ Deep Learning based Matrix Factorization
- ☑ Data Preparation
- ☑ Model Implementation
- ☐ **Model Training / Evaluation**

Training and Evaluation

- In every epoch, train the model and evaluate it:

```
for epoch in range(epochs):
    start_time = time.time()
    train_mse = 0.
    test_mse = 0.

    # train the model

    # test the model
```

  - Training/testing codes are in the next slides

Training

- Train the model using the training dataset:

```
# train the model
model.train()
for batch_idx, (x, r) in enumerate(train_dataloader):
    # get data
    x, r = x.to(DEVICE), r.to(DEVICE)
    i, j = x[:, 0], x[:, 1]

    # set gradients to zero
    optimizer.zero_grad()

    # predict ratings
    pred = model(i, j)

    # get loss
    loss = criterion(pred, r)
    train_mse += loss.item()

    # backpropagation
    loss.backward()

    # update the parameters
    optimizer.step()

train_rmse = (train_mse/(batch_idx+1))**.5
```

Evaluation

- Test the model using the testing dataset:

```
# test the model
model.eval()
for batch_idx, (x, r) in enumerate(test_dataloader):
    # get data
    x, r = x.to(DEVICE), r.to(DEVICE)
    i, j = x[:, 0], x[:, 1]

    # predict ratings
    pred = model(i, j)

    # get loss
    loss = criterion(pred, r)
    test_mse += loss.item()

test_rmse = (test_mse/(batch_idx+1))**.5

end_time = time.time()
print(f'[{end_time-start_time:.2f}] Epoch: {epoch+1:3d}, '
      f'TrnRMSE: {train_rmse:.4f}, TestRMSE: {test_rmse:.4f}')
```

Results (1)

- Matrix Factorization

```
[20.66] Epoch:   1, TrnRMSE: 4.6855, TestRMSE: 4.4827
[20.70] Epoch:   2, TrnRMSE: 4.3197, TestRMSE: 4.2276
[20.90] Epoch:   3, TrnRMSE: 4.0864, TestRMSE: 4.0496
[20.68] Epoch:   4, TrnRMSE: 3.8859, TestRMSE: 3.8329
[20.72] Epoch:   5, TrnRMSE: 3.5282, TestRMSE: 3.2955
[20.57] Epoch:   6, TrnRMSE: 2.7649, TestRMSE: 2.3986
[20.71] Epoch:   7, TrnRMSE: 1.9648, TestRMSE: 1.7797
[20.88] Epoch:   8, TrnRMSE: 1.4992, TestRMSE: 1.4422
[20.77] Epoch:   9, TrnRMSE: 1.2463, TestRMSE: 1.2508
[20.84] Epoch:  10, TrnRMSE: 1.1043, TestRMSE: 1.1392
[20.87] Epoch:  11, TrnRMSE: 1.0232, TestRMSE: 1.0735
[20.62] Epoch:  12, TrnRMSE: 0.9758, TestRMSE: 1.0334
[20.83] Epoch:  13, TrnRMSE: 0.9470, TestRMSE: 1.0086
[21.35] Epoch:  14, TrnRMSE: 0.9284, TestRMSE: 0.9921
[20.93] Epoch:  15, TrnRMSE: 0.9155, TestRMSE: 0.9811
[20.82] Epoch:  16, TrnRMSE: 0.9056, TestRMSE: 0.9725
[20.78] Epoch:  17, TrnRMSE: 0.8972, TestRMSE: 0.9655
[20.65] Epoch:  18, TrnRMSE: 0.8898, TestRMSE: 0.9595
[20.67] Epoch:  19, TrnRMSE: 0.8829, TestRMSE: 0.9542
[20.83] Epoch:  20, TrnRMSE: 0.8765, TestRMSE: 0.9497
[20.72] Epoch:  21, TrnRMSE: 0.8709, TestRMSE: 0.9458
[20.64] Epoch:  22, TrnRMSE: 0.8657, TestRMSE: 0.9424
[20.58] Epoch:  23, TrnRMSE: 0.8612, TestRMSE: 0.9396
[20.56] Epoch:  24, TrnRMSE: 0.8571, TestRMSE: 0.9375
[20.63] Epoch:  25, TrnRMSE: 0.8536, TestRMSE: 0.9355
[21.15] Epoch:  26, TrnRMSE: 0.8504, TestRMSE: 0.9334
[20.90] Epoch:  27, TrnRMSE: 0.8474, TestRMSE: 0.9326
[20.94] Epoch:  28, TrnRMSE: 0.8447, TestRMSE: 0.9308
[20.80] Epoch:  29, TrnRMSE: 0.8423, TestRMSE: 0.9300
[20.51] Epoch:  30, TrnRMSE: 0.8399, TestRMSE: 0.9292
```

Results (2)

- Deep Matrix Factorization

```
[22.92] Epoch:   1, TrnRMSE: 1.6224, TestRMSE: 1.0381
[23.06] Epoch:   2, TrnRMSE: 0.9955, TestRMSE: 0.9624
[22.50] Epoch:   3, TrnRMSE: 0.9309, TestRMSE: 0.9225
[22.31] Epoch:   4, TrnRMSE: 0.9105, TestRMSE: 0.9154
[22.32] Epoch:   5, TrnRMSE: 0.9062, TestRMSE: 0.9134
[22.05] Epoch:   6, TrnRMSE: 0.9042, TestRMSE: 0.9116
[22.32] Epoch:   7, TrnRMSE: 0.9015, TestRMSE: 0.9089
[22.33] Epoch:   8, TrnRMSE: 0.8967, TestRMSE: 0.9053
[22.29] Epoch:   9, TrnRMSE: 0.8915, TestRMSE: 0.9020
[22.27] Epoch:  10, TrnRMSE: 0.8874, TestRMSE: 0.9004
[22.33] Epoch:  11, TrnRMSE: 0.8838, TestRMSE: 0.8983
[22.11] Epoch:  12, TrnRMSE: 0.8805, TestRMSE: 0.8968
[22.29] Epoch:  13, TrnRMSE: 0.8775, TestRMSE: 0.8953
[22.33] Epoch:  14, TrnRMSE: 0.8744, TestRMSE: 0.8949
[22.26] Epoch:  15, TrnRMSE: 0.8714, TestRMSE: 0.8938
[22.30] Epoch:  16, TrnRMSE: 0.8686, TestRMSE: 0.8937
[22.28] Epoch:  17, TrnRMSE: 0.8659, TestRMSE: 0.8934
[22.08] Epoch:  18, TrnRMSE: 0.8634, TestRMSE: 0.8939
[22.37] Epoch:  19, TrnRMSE: 0.8613, TestRMSE: 0.8933
[22.37] Epoch:  20, TrnRMSE: 0.8592, TestRMSE: 0.8931
```

What You Need to Know

- PyTorch
  - A Numpy-friendly deep learning framework
- Deep Matrix Factorization
  - A non-linear model
  - Wide range of parameters
  - A complex model

Questions?