Deep Learning

Deep Recommender System - Lab

U Kang
Seoul National University

In This Lecture

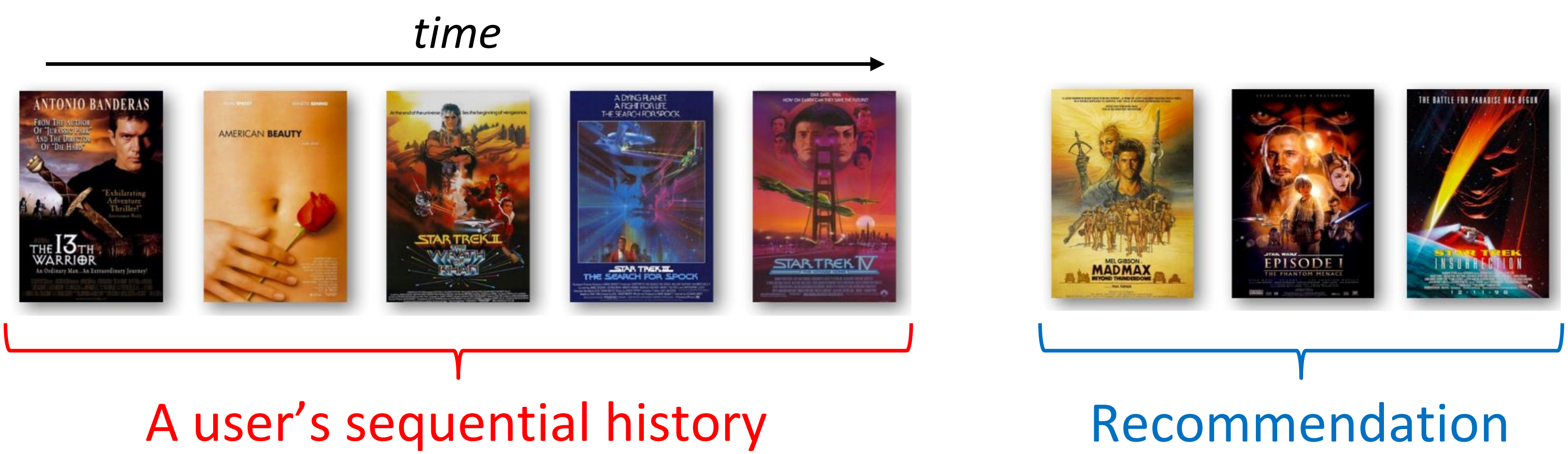
- RNN based Recommender System
  - Sequential recommendation
  - Data preparation
  - Model implementation
  - Model training / evaluation

Outline

- ☐ **Sequential Recommendation**
- ☐ Data Preparation
- ☐ Model Implementation
- ☐ Model Training / Quantitative evaluation
- ☐ Qualitative Evaluation

Sequential Recommendation (1)

- ***Given***
  - Users' sequential history (buy, watch, etc.)
- ***Goal***
  - Predict items that maximize her/his future needs

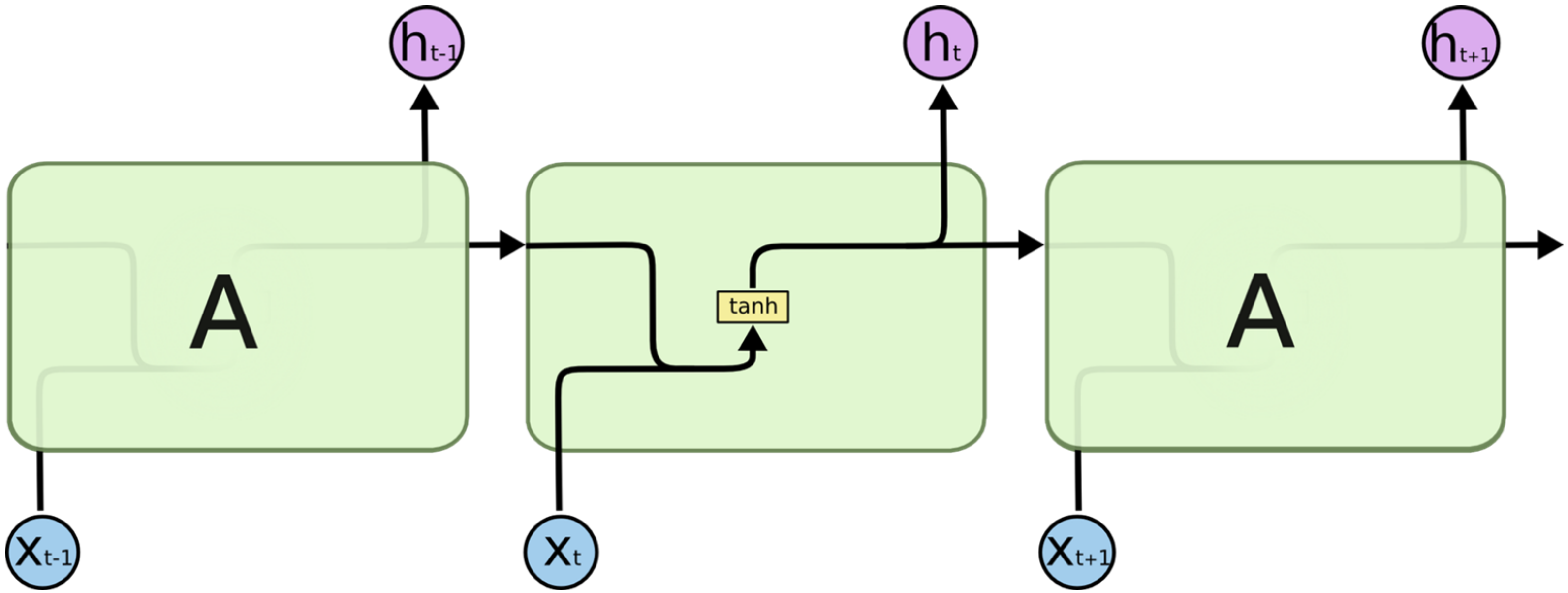


Sequential Recommendation (2)

- A user's past interaction sequence is a significant information
- We need to consider personal preference, in addition to past interaction

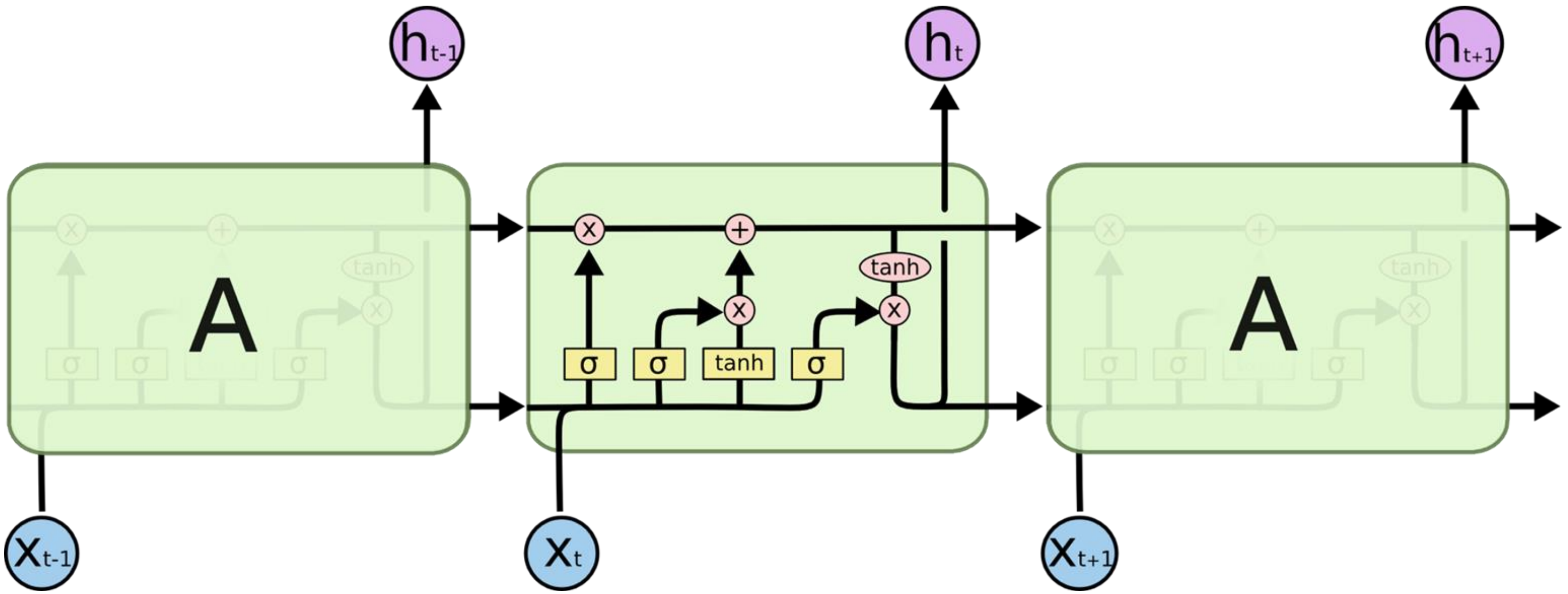
Recurrent Neural Network (RNN)

- A deep learning structure for sequential data
- Contains a **cell**, which is a repeated structure
- Stores and passes **states** through a sequence

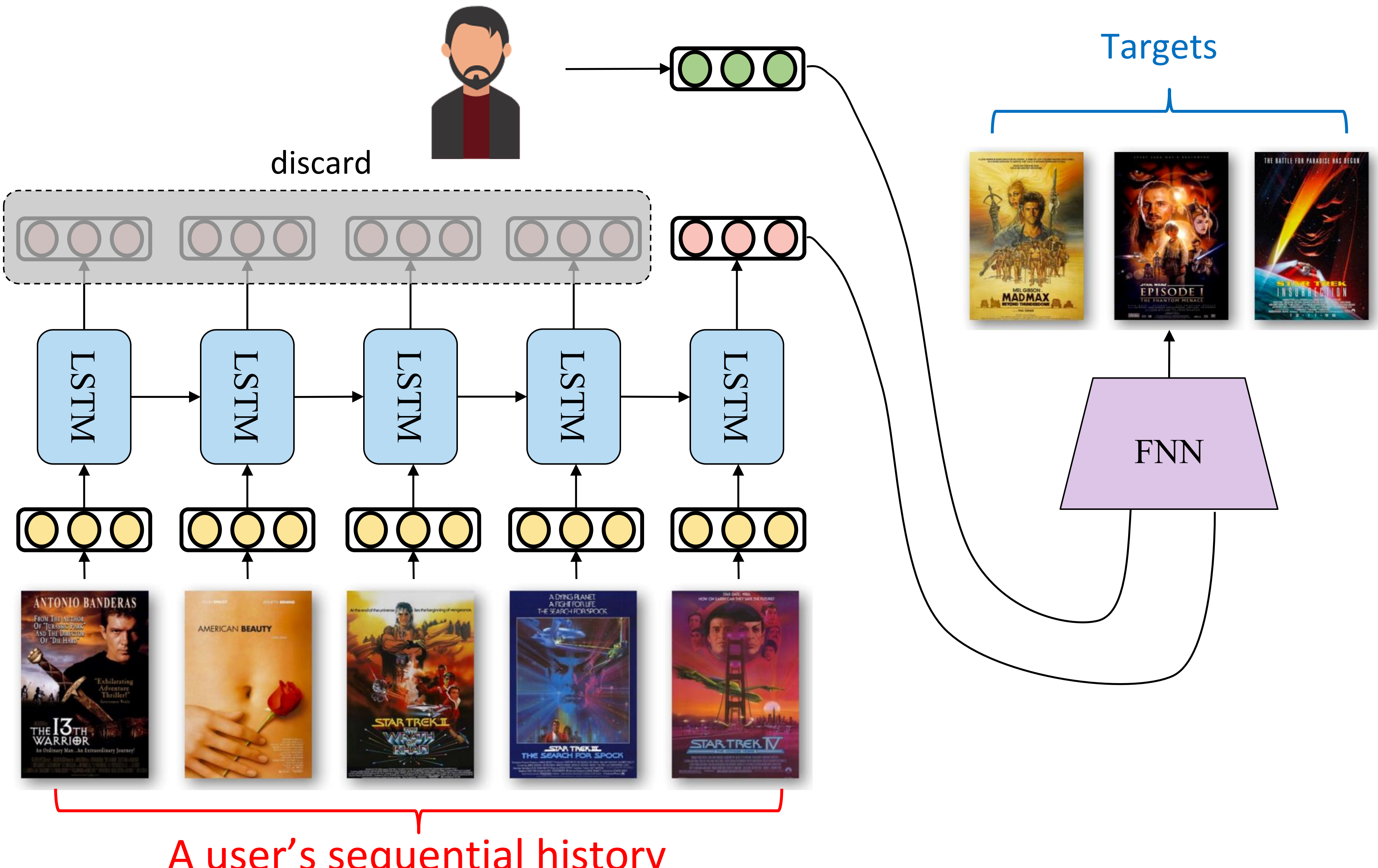


Long Short-term Memory (LSTM)

- An advanced RNN structure
- It avoids the long-term dependency problem



Architecture



Outline

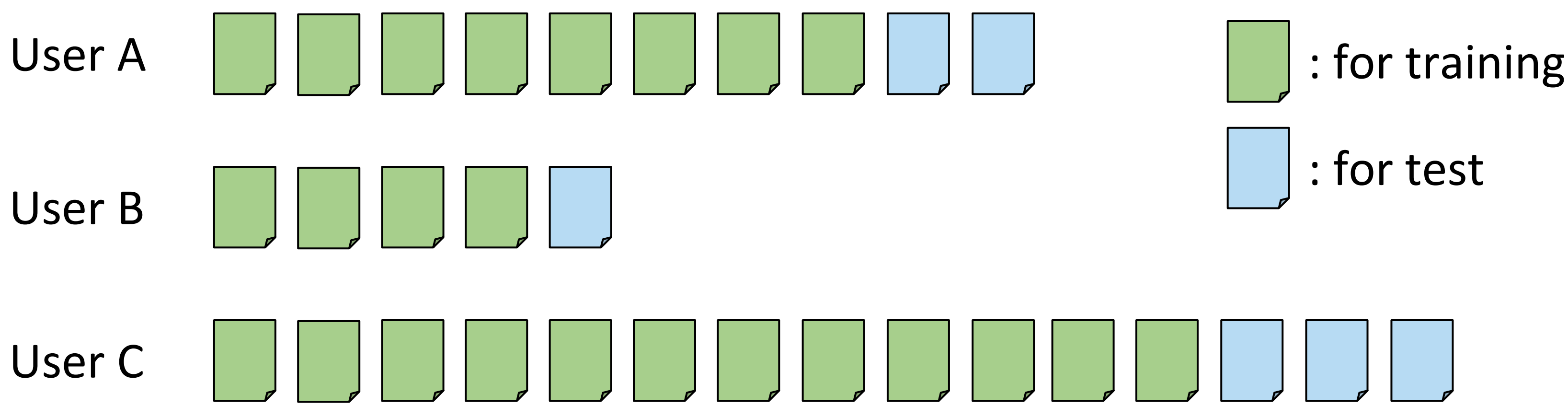
- ☑ Sequential Recommendation
- ☐ **Data Preparation**
- ☐ Model Implementation
- ☐ Model Training / Quantitative evaluation
- ☐ Qualitative Evaluation

Dataset

- We use one of the most famous datasets in recommendation community: **MovieLens-1M**
  - Number of interactions: 1,000,209
  - Number of users: 6,040
  - Number of items: 3,952
  - Users gave ratings between 1 and 5 to items
  - Each user has at least 20 ratings
  - Logs between 1997/09/19 ~ 1998/04/22

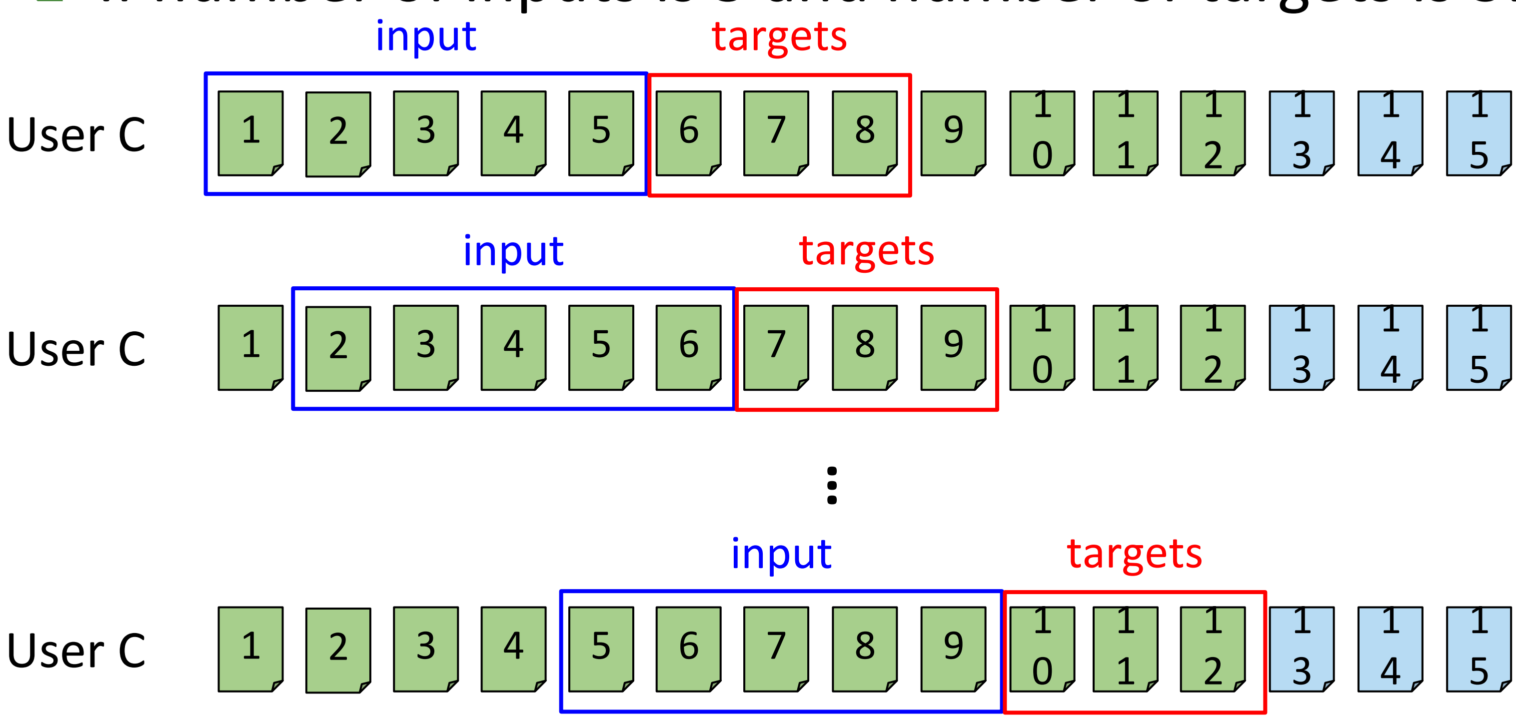
Data Preparation (1)

- Let's define training/test data
  - For each user, we use the first 80% of interactions as a training set
  - The remaining 20% of interactions are used as a test set



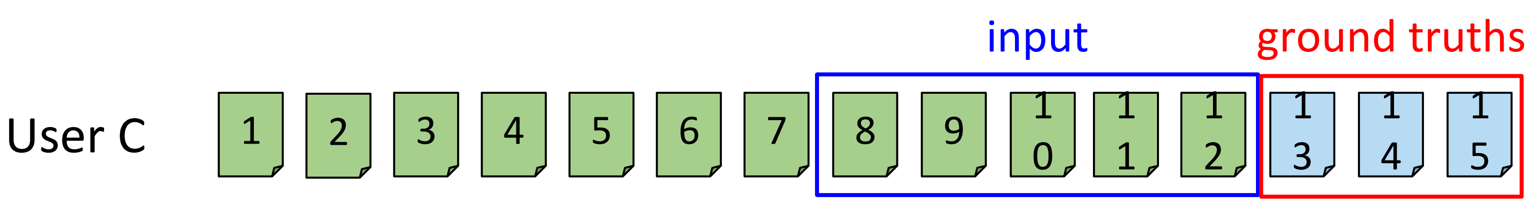
Data Preparation (2)

- For each user, we get multiple training instances using a fixed size of window
  - If number of inputs is 5 and number of targets is 3:



Data Preparation (3)

- A model is trained to predict the target items given input items
- When testing the model, we feed the last 5 items to the model and predict items that a user will interact with
- Then compare the predicted items with ground truths



Reading Data File (1)

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

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

Reading Data File (2)

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

```
# Load the input file in memory
raw = []
with open(ratings_file, 'r') as f_read:
    for line in f_read.readlines():
        line_list = line.split('::')
        raw.append(line_list)
```

Data Analysis (1)

- Let's analyze the dataset
- **User** skewness
  - X-axis: number of interactions
  - Y-axis: number of **users**
- **Item** skewness
  - X-axis: number of interactions
  - Y-axis: number of **items**

Data Analysis (2)

- Import numpy and pyplot

```
import numpy as np
import matplotlib.pyplot as plt
```

- Define the plot size

```
plt.rcParams["figure.figsize"] = (15,4)
```

Data Analysis (3)

- Define user plot

```
raw = np.array(raw, dtype=int)
user_freq = np.bincount(raw[:, 0]) # [user1's freq, user2's freq, ..., usern's freq]
user_freq = [i for i in user_freq if i>0] # exclude dummy users
user_freq = np.bincount(user_freq)
user_x_axis = np.array(range(len(user_freq)))
print(f'users` max freq: {len(user_freq)-1}')
```

- Define item plot

```
item_freq = np.bincount(raw[:, 1]) #[item1's freq, item2's freq, ..., itemm's freq]
item_freq = [i for i in item_freq if i>0] # exclude dummy items
item_freq = np.bincount(item_freq)
item_x_axis = np.array(range(len(item_freq)))
print(f'items` max freq: {len(item_freq)-1}')
```

Data Analysis (4)

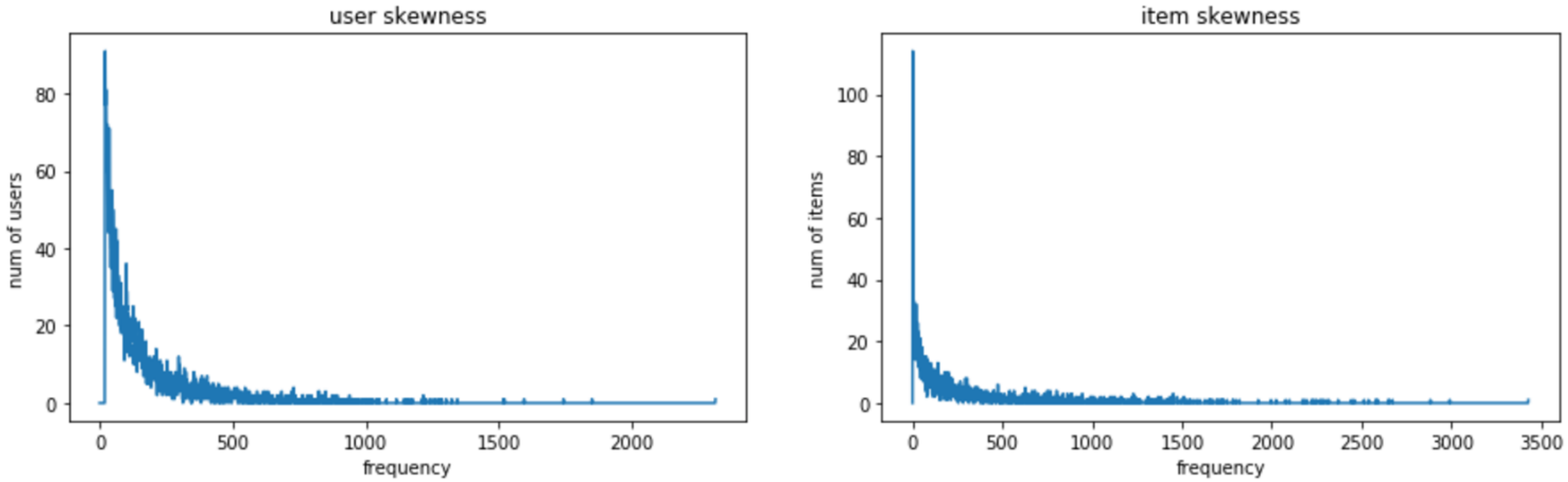
- Draw the plots

```
fig, axs = plt.subplots(1, 2)
axs[0].plot(user_x_axis, user_freq)
axs[0].set_title('user skewness')
axs[0].set_xlabel('frequency')
axs[0].set_ylabel('num of users')
axs[1].plot(item_x_axis, item_freq)
axs[1].set_title('item skewness')
axs[1].set_xlabel('frequency')
axs[1].set_ylabel('num of items')
plt.show()
```

Data Analysis (5)

- The dataset is extremely skewed, which makes it difficult to make a personalized recommendation
  - A model will have a low loss even if it simply recommends the popular items to users

```
users` max freq: 2314
items` max freq: 3428
```



Data Sorting

- Sort interactions by (user_id, timestamp) since we will split the data into training/test set based on each user's sequence

```
raw_sorted = np.array(sorted(raw, key=lambda x: (x[0], x[3])))
print(f'num of interactions: {len(raw_sorted)}')
```

```
num of interactions: 1000209
```

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_sorted = [[user_map[u], item_map[i]] for (u, i)
              in zip(raw_sorted[:, 0], raw_sorted[:, 1])] # new array
```

Side Information

- Construct dictionary of items' side information
  - "movies.dat" contains title/genres of every item
  - Format: item_id::title::genres
- Note that we use it only for evaluating the model, not for training

```
movies_file = in_path + 'movies.dat'
meta_dict = dict()

with open(movies_file, 'r',  encoding='ISO-8859-1') as f_read:
    for line in f_read.readlines():
        line_list = line.split('::')
        raw_id = int(line_list[0].strip())
        try:
            new_id = item_map[raw_id]
        except KeyError:
            continue
        meta_dict[new_id] = [line_list[1].strip(), line_list[2].strip()]
```

Split Dataset (1)

- Split the dataset into training/test sets for each user

```
ratio = 0.8
```

```
new_sorted = np.array(new_sorted)
split_idx = np.flatnonzero(np.diff(new_sorted[:, 0])) + 1
```

```
trn_list, test_list = [], []
for arr in np.array_split(new_sorted, split_idx):
  split_i = round(len(arr) * ratio)
  trn_list.append(arr[:split_i, :])
  test_list.append(arr[split_i:, :])
```

  - np.diff computes differences along with the axis
  - np.flatnonzero returns non-zero indices

Split Dataset (2)

- An example:
  - If user 0 has interacted with items [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], we obtain *trn_arr* and *test_arr* as follows.
    - Each row represents (user id, item id)

```
trn_arr (80%):
[[0, 1],
 [0, 2],
 [0, 3],
 [0, 4],
 [0, 5],
 [0, 6],
 [0, 7],
 [0, 8],
 [0, 9]]
```

```
test_arr (20%):
[[0, 10],
 [0, 11]]
```

Data instances (1)

- Data instance hyperparameters
- We also need negative samples to train a model

```
feed_len = 5
target_len = 3
neg_samples = 10
```

Data instances (2)

- Prepare placeholders for training/test instances

```
trn_users = []
trn_feed_sequences = []
trn_positive_targets = []
trn_negative_targets = []

test_users = []
test_feed_sequences = []
test_targets = []
```

Data instances (3)

- An example:
  - For user 0, assume we have *trn_arr* and *test_arr* as follows.
    - Each row represents (user id, item id)

```
trn_arr (80%):
[[0, 1],
 [0, 2],
 [0, 3],
 [0, 4],
 [0, 5],
 [0, 6],
 [0, 7],
 [0, 8],
 [0, 9]]
```

```
test_arr (20%):
[[0, 10],
 [0, 11]]
```

Data instances (4)

- Example of instance matrices: "sequence length 5", "target length 3", and "number of negative samples 10" are as follows.

Window sliding

```
trn_users:
[0, 0]
trn_feed_sequences:
[[1, 2, 3, 4, 5],
 [2, 3, 4, 5, 6]]
trn_positive_targets:
[[6, 7, 8]
 [7, 8, 9]]
```

Random sampling

```
trn_negative_targets:
[[n1, n2, ..., n10],
 [n1', n2', ..., n10']]
```

From trn_arr

```
test_users:
0
test_feed_sequences:
[5, 6, 7, 8, 9]
```

From test_arr

```
test_targets:
[10, 11]
```

Data instances (5)

- Generate training and test instances

For a user's training and test interactions (slide 28)

```python
for trn_arr, test_arr in zip(trn_list, test_list):
    trn_split = np.lib.stride_tricks.sliding_window_view(trn_arr, window_shape=feed_len+target_len, axis=0)
    trn_users.append(trn_split[:, 0, 0])
    trn_feed_sequences.append(trn_split[:, 1, :feed_len])
    trn_positive_targets.append(trn_split[:, 1, feed_len:])
    test_feed_sequences.append(trn_arr[-feed_len:, 1])
    trn_negative_targets.append(np.random.randint(len(item_map), size=(trn_split.shape[0], neg_samples)))

    test_users.append(test_arr[0, 0])
    test_target_sequence = test_arr[:, 1]
    test_targets.append(test_target_sequence)
```

- sliding_window_view generates the same size of instances using a sliding window
  - Shape: [N, 2] → [N-W+1, 2, W]
    - N: number of interactions
    - W: window size

Data instances (6)

- Elements of each training placeholder

```
trn_users[3]
```

```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

```
trn_positive_targets[3]
```

```
array([[ 253, 1106, 1108],
       [1106, 1108, 1288],
       [1108, 1288, 1848],
       [1288, 1848, 2173],
       [1848, 2173, 1111],
       [2173, 1111, 2488],
       [1111, 2488, 2739],
       [2488, 2739, 1124],
       [2739, 1124, 3186],
       [1124, 3186, 3460]])
```

```
trn_feed_sequences[3]
```

```
array([[1120, 1025, 3235,  466, 3294],
       [1025, 3235,  466, 3294,  253],
       [3235,  466, 3294,  253, 1106],
       [ 466, 3294,  253, 1106, 1108],
       [3294,  253, 1106, 1108, 1288],
       [ 253, 1106, 1108, 1288, 1848],
       [1106, 1108, 1288, 1848, 2173],
       [1108, 1288, 1848, 2173, 1111],
       [1288, 1848, 2173, 1111, 2488],
       [1848, 2173, 1111, 2488, 2739]])
```

```
trn_negative_targets[3]
```

```
array([[ 209, 2226,  394, 3604, 2552, 3659, 1999, 1797, 2946, 1999],
       [1131, 2657,  764, 1524, 3406,   89, 2078, 3384, 2420, 3229],
       [  28, 1509, 3282, 2551, 3129, 2877, 2043, 1997, 1819, 1018],
       [2030, 3031,   10, 2697,  397,  649, 2237, 2527, 1403,  213],
       [3407, 2824, 3148, 1118,  529, 1744, 2274,  719,  941, 1364],
       [1081, 3204, 2298, 3441, 3621, 3524, 1523, 2467, 2309, 2536],
       [2211,  599, 1995, 3428, 1799, 2449,  757, 3640, 1075, 2667],
       [1084, 1865, 3246,   74, 2750,  648,  765,  753, 2709, 2242],
       [ 976,  210,  802,  596, 3590, 2917,   97,   49, 3285, 3405],
       [ 874, 2880, 2180, 3338,  405,  393, 3503,   69, 2629,  144]])
```

Data instances (7)

- Elements of each test placeholder

```
test_users[3]
```

```
3
```

```
test_feed_sequences[3]
```

```
array([2488, 2739, 1124, 3186, 3460])
```

```
test_targets[3]
```

```
array([1148, 2743,  971, 1774])
```

Data instances (8)

- Convert the lists into numpy arrays

```
trn_users = np.concatenate(trn_users, axis=0)
trn_feed_sequences = np.concatenate(trn_feed_sequences, axis=0)
trn_positive_targets = np.concatenate(trn_positive_targets, axis=0)
trn_negative_targets = np.concatenate(trn_negative_targets, axis=0)
test_users = np.array(test_users)
test_feed_sequences = np.stack(test_feed_sequences)
# We cannot construct test_targets as an array since the number of targets is different for each user

x_train = np.concatenate((trn_users[:, np.newaxis], trn_feed_sequences), axis=1)
x_test = np.concatenate((test_users[:, np.newaxis], test_feed_sequences), axis=1)
targets_train = np.concatenate((trn_positive_targets, trn_negative_targets), axis=1)
```

- targets_train: positive and negative item indices
  - We need indices of positive and negative targets to train a model with a negative sampling technique

Data instances (9)

- Resultant instances

```
x_train
```

```
array([[   0, 2969, 1178, 1574,  957, 2147],
       [   0, 1178, 1574,  957, 2147, 1658],
       [   0, 1574,  957, 2147, 1658, 3177],
       ...,
       [6039, 3493, 3441, 1124, 2410, 2443],
       [6039, 3441, 1124, 2410, 2443, 1342],
       [6039, 1124, 2410, 2443, 1342, 3271]])
```

Shape: [# instances, 1 + feed_len]
Each row contains user index and item indices

```
x_test
```

```
array([[   0, 1107,  580, 2205, 1421,  513],
       [   1, 1826, 2086, 1271,  627, 2234],
       [   2, 3189, 1295, 2785, 1212, 3379],
       ...,
       [6037, 2872,  225, 1059, 1133, 1204],
       [6038,  740, 1011, 1314,  193,   47],
       [6039, 1342, 3271, 2111, 3107,  256]])
```

Shape: [# instances, 1 + feed_len]
Each row contains user index and item indices

```
targets_train
```

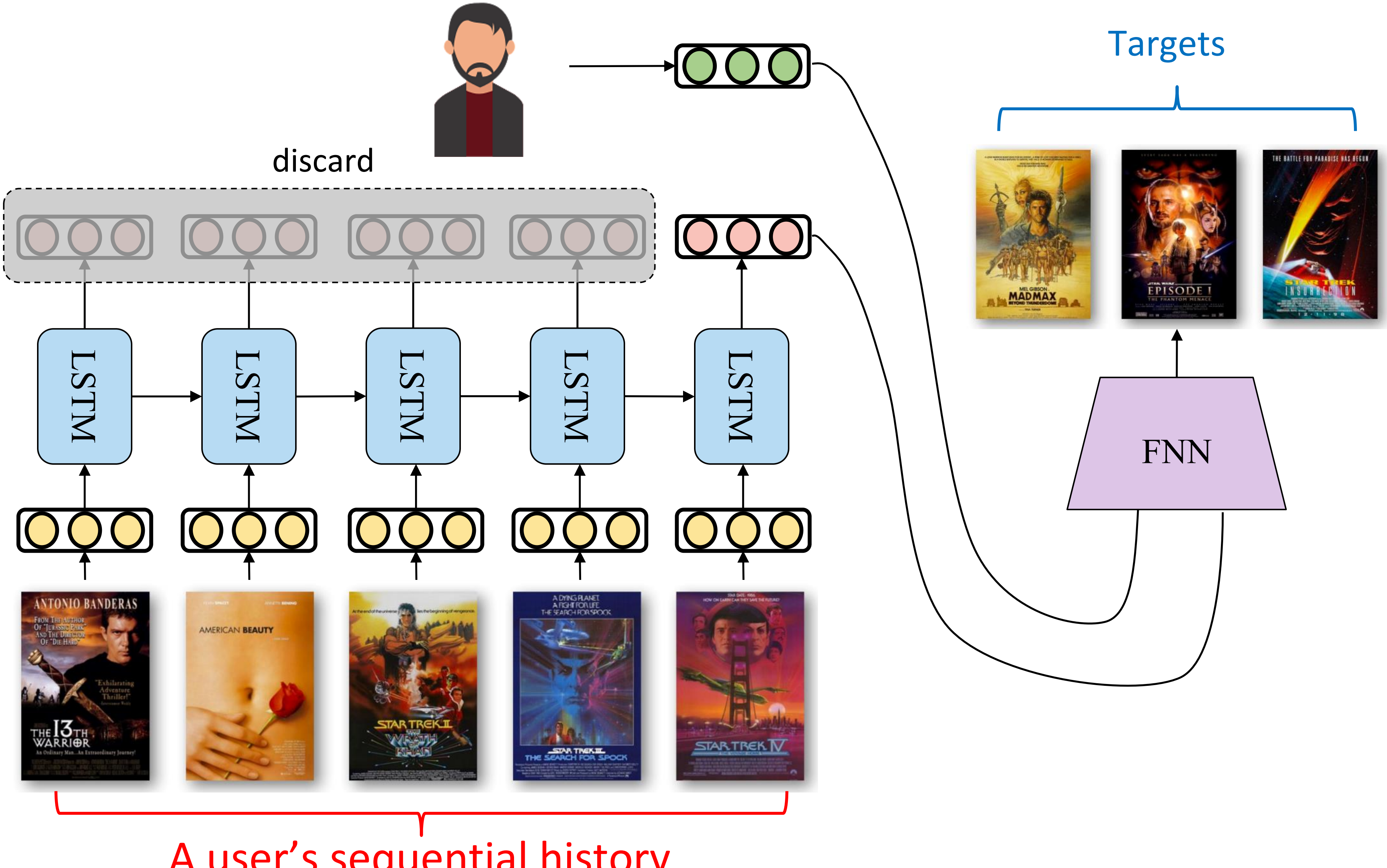
```
array([[1658, 3177, 2599, ..., 3083, 1037, 1310],
       [3177, 2599, 1117, ...,  339, 1352, 2332],
       [2599, 1117, 1104, ..., 1973,   82, 2279],
       ...,
       [1342, 3271, 2111, ..., 1362, 1510, 3067],
       [3271, 2111, 3107, ...,  200, 3294, 3180],
       [2111, 3107,  256, ..., 2416, 2895, 1230]])
```

Shape: [# instances, # target + # negative]
Each row contains item indices

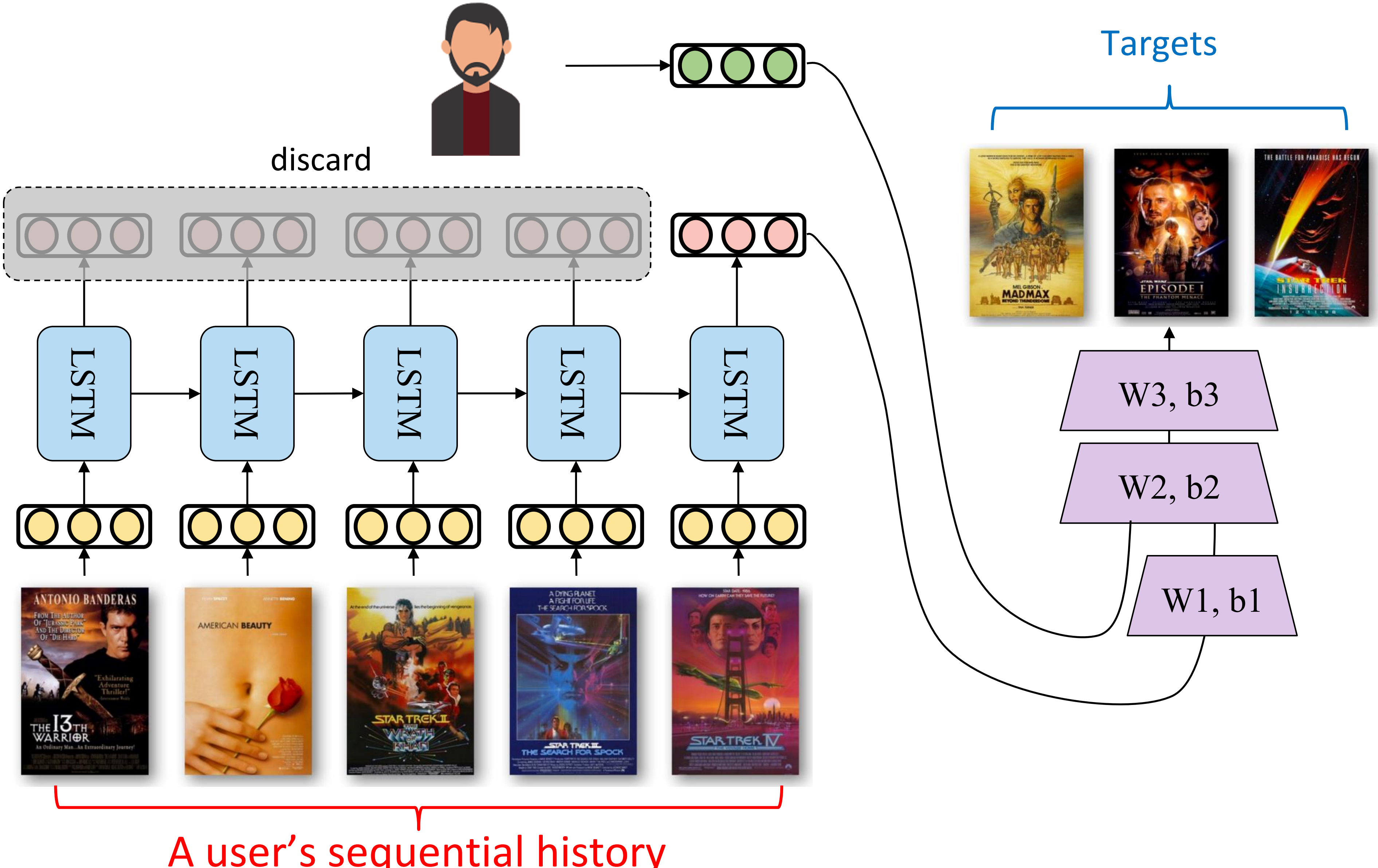
Outline

- ☑ Sequential Recommendation
- ☑ Data Preparation
- ☐ **Model Implementation**
- ☐ Model Training / Quantitative evaluation
- ☐ Qualitative Evaluation

Architecture (1)



Architecture (2)



Hyper-parameters

- Set the model's hyper-parameters

```
lr = 1e-3
batch_size = 1000
epochs = 10
emb_dim = 50
hid_dim = 50
epsilon = 1e-10 # This prevents the occurrence of log(0)
```

Prepare Model

- We need embedding vectors, LSTM cell, and fully-connected layers

```python
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Embedding, LSTM, concatenate

class my_model(tf.keras.Model):
    def __init__(self, num_items, num_users, emb_dim, hid_dim):
        super(my_model, self).__init__(self)
        self.item = Embedding(num_items, emb_dim)
        self.user = Embedding(num_users, emb_dim)
        self.lstm = LSTM(units=hid_dim)
        self.lstm_dense = Dense(emb_dim)
        self.final1 = Dense(emb_dim, activation='relu')
        self.final2 = Dense(num_items)

    def call(self, x):
        user = x[:, 0]
        item = x[:, 1:]
        item_out = self.lstm_dense(self.lstm(self.item(item)))
        user_embedding = self.user(user)
        concat = concatenate([item_out, user_embedding])
        model_out = self.final2(self.final1(concat))
        return model_out

model = my_model(len(item_map), len(user_map), emb_dim, hid_dim)
```

Loss (1)

- The loss function to minimize:
  - $L = \frac{1}{|D|} \sum_D [\underbrace{\sum_{i \in T} -\log(\sigma(y_i))}_{\text{Positive Loss}} + \underbrace{\sum_{j \notin T} -\log(1 - \sigma(y_j))}_{\text{Negative Loss}}]$
  - D is a set of all data instances
  - T is a set of targets
  - y_k is a score for next item k

Loss (2)

- Define a customized loss function

```
def rec_loss(targets, y_pred):
  logits = tf.gather(y_pred, targets, axis=1, batch_dims=1)
  pos_logits, neg_logits = logits[:, :target_len], logits[:, target_len:]
  pos_loss = -tf.math.reduce_sum(tf.math.log(tf.clip_by_value(tf.math.sigmoid(pos_logits),
                                                              clip_value_min=epsilon, clip_value_max=1)), axis=1)
  neg_loss = -tf.math.reduce_sum(tf.math.log(tf.clip_by_value(1 - tf.math.sigmoid(neg_logits),
                                                              clip_value_min=epsilon, clip_value_max=1)), axis=1)
  return tf.math.reduce_mean(pos_loss + neg_loss)
```

  - targets: indices of item targets
  - Shape: [# batch, # positive + # negative]
    - Each row contains [p1, p2, p3, n1, n2, …, n10]
  - y_pred: predicted scores for all items
    - Shape: [# batch, # items]
    - Each row contains [score1, score2, …, score3706]
  - clip_by_value prevents the loss to become ‘nan’

Loss (3)

- Compile the model with the loss function
  - We use the Adam optimizer

```
model.compile(optimizer='adam', loss=rec_loss)
```

Outline

- ☑ Sequential Recommendation
- ☑ Data Preparation
- ☑ Model Implementation
- ☐ **Model Training / Quantitative evaluation**
- ☐ Qualitative Evaluation

Train the model

- Train the model using the training dataset

```
model.fit(x_train, targets_train, epochs=epochs, batch_size=batch_size)
```

```
Epoch 1/10
758/758 [==============================] - 10s 9ms/step - loss: 5.1968
Epoch 2/10
758/758 [==============================] - 7s 9ms/step - loss: 4.2492
Epoch 3/10
758/758 [==============================] - 8s 11ms/step - loss: 3.5982
Epoch 4/10
758/758 [==============================] - 10s 13ms/step - loss: 3.1626
Epoch 5/10
758/758 [==============================] - 7s 9ms/step - loss: 2.8563
Epoch 6/10
758/758 [==============================] - 7s 9ms/step - loss: 2.6843
Epoch 7/10
758/758 [==============================] - 7s 9ms/step - loss: 2.5603
Epoch 8/10
758/758 [==============================] - 7s 9ms/step - loss: 2.4638
Epoch 9/10
758/758 [==============================] - 7s 9ms/step - loss: 2.3860
Epoch 10/10
758/758 [==============================] - 7s 9ms/step - loss: 2.3201
```

Evaluation Metrics (1)

- 3 evaluation metrics: Precision, Recall, MAP
  - MAP (Mean Average Precision) is a ranking based metric

```
def compute_precision(predictions, targets, k):
    pred = predictions[:k]
    num_hit = len(set(pred).intersection(set(targets)))
    return float(num_hit) / len(pred)

def compute_recall(predictions, targets, k):
    pred = predictions[:k]
    num_hit = len(set(pred).intersection(set(targets)))
    return float(num_hit) / len(targets)

def compute_ap(predictions, targets, k):
    if len(predictions) > k:
        predictions = predictions[:k]
    score = 0.
    num_hits = 0.
    for i, p in enumerate(predictions):
        if p in targets:
            num_hits += 1.
            score += num_hits / (i+1)
    return score / min(len(targets), k)
```

Evaluation Metrics (2)

- “Evaluate” function combines the 3 metric functions

```
def evaluate(preds, gts, k=10):
    precs = [compute_precision(p, t, k=k) for (p, t) in zip(preds, gts)]
    recalls = [compute_recall(p, t, k=k) for (p, t) in zip(preds, gts)]
    aps = [compute_ap(p, t, k=k) for (p, t) in zip(preds, gts)]
    return float(sum(precs) / len(precs)), \
           float(sum(recalls) / len(recalls)), \
           float(sum(aps) / len(aps))
```

Evaluate the model

- Evaluate the model using the three metrics: precision, recall, and mAP

```
preds = tf.argsort(-model.call(x_test)).numpy()
prec, recall, ap = evaluate(preds, test_targets, k=10)
print(f'Prec@10: {prec:.4f}, Recall@10: {recall:.4f}, Map@10: {ap:.4f}')
```

Outline

- ☑ Sequential Recommendation
- ☑ Data Preparation
- ☑ Model Implementation
- ☑ Model Training / Quantitative evaluation
- ☐ **Qualitative Evaluation**

Qualitative Evaluation (1)

- Pick a random user and evaluate the performance qualitatively

```
from random import randint
idx = randint(0, len(test_users))
history = [meta_dict[i] for i in test_feed_sequences[idx]]
predict = [meta_dict[i] for i in preds[idx][:10]]
real = [meta_dict[i] for i in test_targets[idx]]

def mask2str(mask):
    return 'O' if mask else 'X'

predict_mask = [check in test_targets[idx] for check in preds[idx][:10]]
real_mask = [check in preds[idx][:10] for check in test_targets[idx]]
predict_mask = [mask2str(mask) for mask in predict_mask]
real_mask = [mask2str(mask) for mask in real_mask]
```

Qualitative Evaluation (2)

- Print the watched history, the ground truths, and the predicted targets

```
print('==== Watched history ====')
for i in history:
    print(i)

print('==== Real targets ===')
for m, v in zip(real_mask, real):
    print(f'{[m]} {v}')


print('=== Predicted targets ===')
for m, v in zip(predict_mask, predict):
    print(f'{[m]} {v}')
```

Qualitative Evaluation (3)

```
==== Watched history ====
['Monty Python and the Holy Grail (1974)', 'Comedy']
['Raising Arizona (1987)', 'Comedy']
['Roger & Me (1989)', 'Comedy|Documentary']
['Babe (1995)', "Children's|Comedy|Drama"]
['Groundhog Day (1993)', 'Comedy|Romance']
==== Real targets ===
['O'] ['Player, The (1992)', 'Comedy|Drama']
['O'] ['This Is Spinal Tap (1984)', 'Comedy|Drama|Musical']
['X'] ['Hoop Dreams (1994)', 'Documentary']
['X'] ['Thirty-Two Short Films About Glenn Gould (1993)', 'Documentary']
=== Predicted targets ===
['X'] ['Shakespeare in Love (1998)', 'Comedy|Romance']
['X'] ['Roger & Me (1989)', 'Comedy|Documentary']
['X'] ['Election (1999)', 'Comedy']
['X'] ['Crimes and Misdemeanors (1989)', 'Comedy']
['O'] ['This Is Spinal Tap (1984)', 'Comedy|Drama|Musical']
['X'] ['Groundhog Day (1993)', 'Comedy|Romance']
['O'] ['Player, The (1992)', 'Comedy|Drama']
['X'] ['Being John Malkovich (1999)', 'Comedy']
['X'] ['Raising Arizona (1987)', 'Comedy']
['X'] ['Princess Bride, The (1987)', 'Action|Adventure|Comedy|Romance']
```

What You Need to Know

- RNN based Recommender System
  - Sequential recommendation
  - Data preparation
  - Model implementation
  - Model training / Evaluation

Questions?