Convolutional Matrix Factorization for Document Context-Aware Recommendation /AutoRec: Autoencoders Meet Collaborative Filtering

2023.2.14

Index

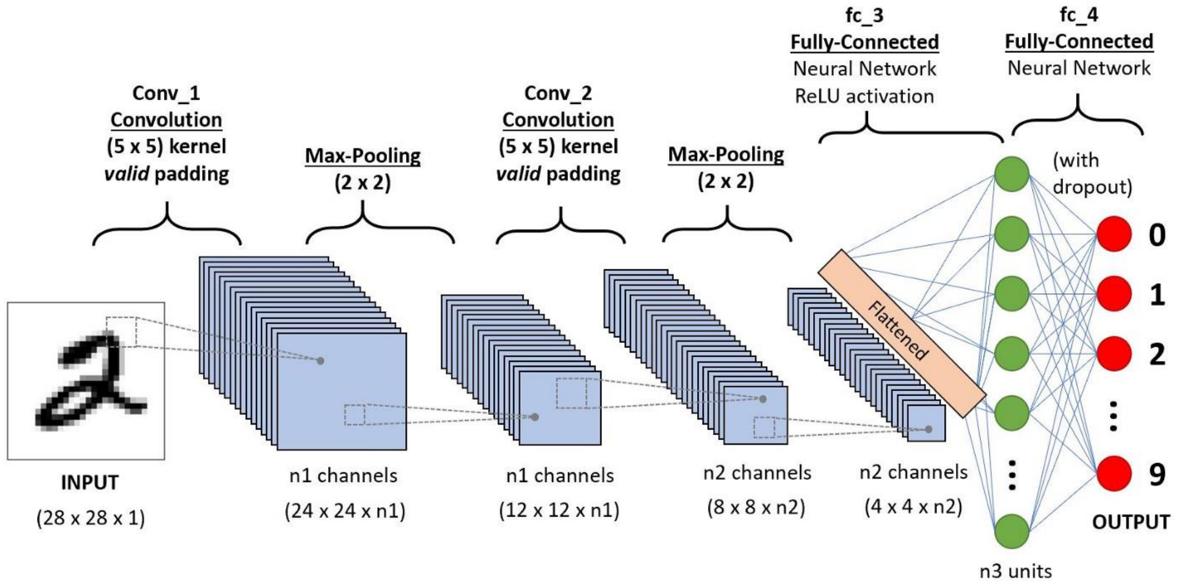
Introduction

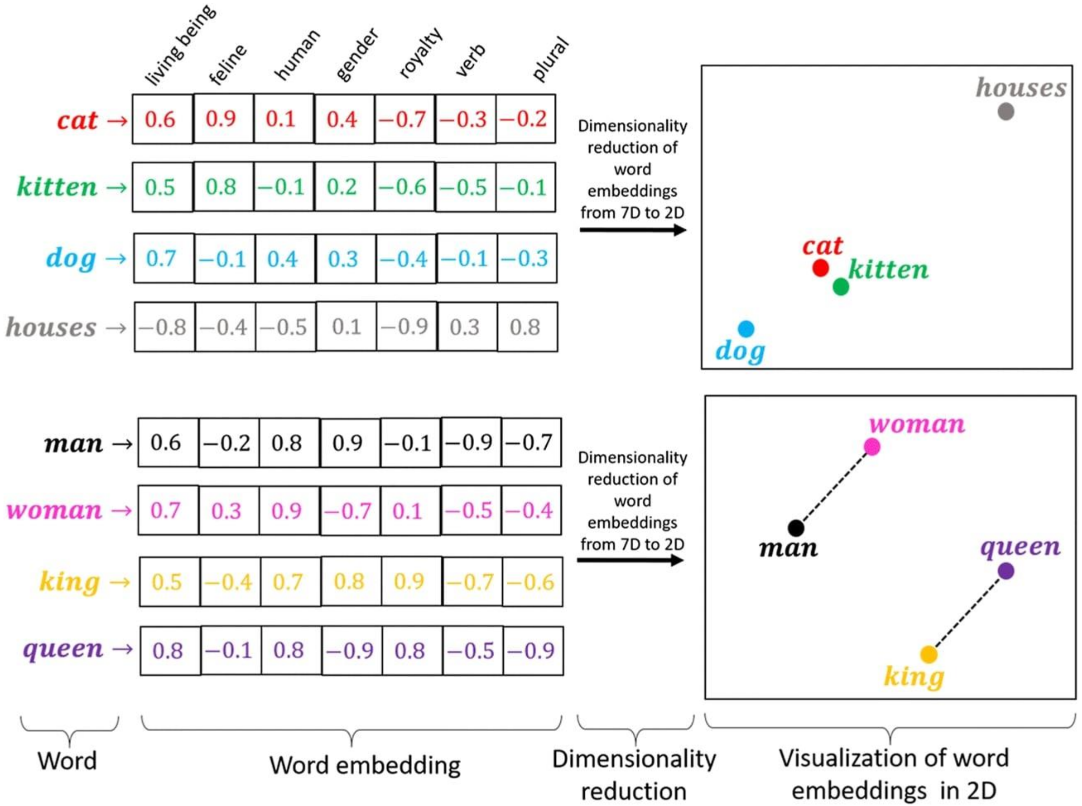
- Exploding growth of the number of users and items in e-commerce services
  - Led to increase of the sparseness of user-item rating data

- Use auxiliary information such as demography of users, social networks, item description documents.
  - However, existing models ignores contextual information
  - This is because they use bag-of-words models
    - Ex)"people **trust** the man"
      "people betray his **trust** finally"

Introduction

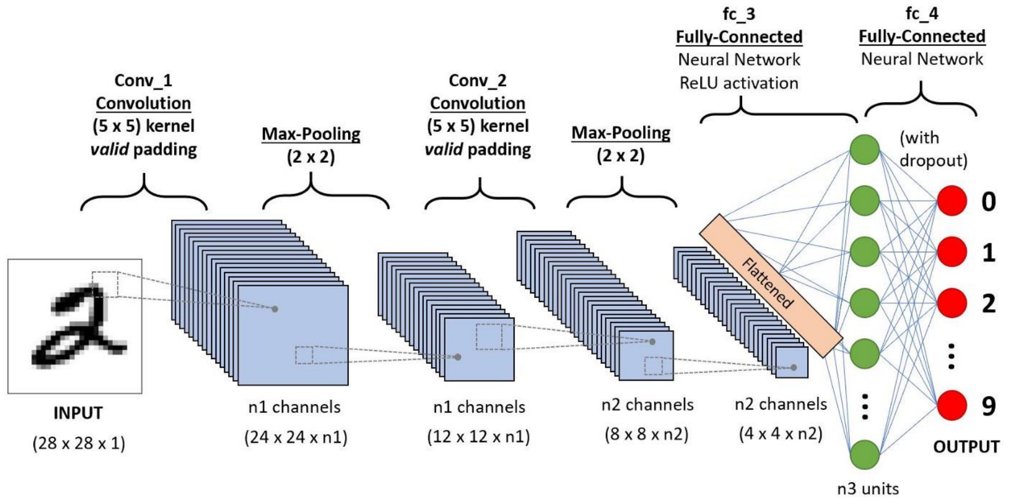
- Utilize Convolutional neural network(CNN) with natural language processing(NLP)
  - CNN captures local features of image or documents
  - Can also add pre-trained word embedding such as GloVe

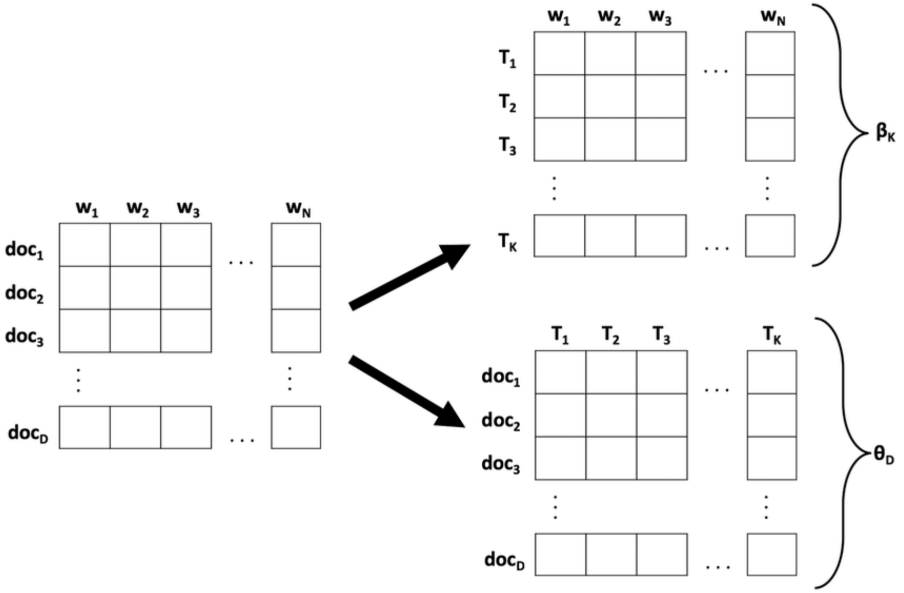




Introduction

- Add this CNN with PMF(Probabilistic Matrix Factorization)
  - Convolutional matrix factorization(ConvMF)





Preliminary

- Matrix Factorization
  - Model-based methods (latent factor model)

$$\mathcal{L} = \sum_{i}^{N} \sum_{j}^{M} I_{ij}(r_{ij} - u_i^T v_j)^2 + \lambda_u \sum_{i}^{N} \|u_i\|^2 + \lambda_v \sum_{j}^{M} \|v_j\|^2$$

- Convolutional Neural Network
  - Convolution layer
  - Pooling Layer

Preliminary

- Convolutional Matrix Factorization

$$p(R|U, V, \sigma^2) = \prod_{i}^{N} \prod_{j}^{M} N(r_{ij}|u_i^T v_j, \sigma^2)^{I_{ij}}$$

$$p(U|\sigma_U^2) = \prod_{i}^{N} N(u_i|0, \sigma_U^2 I)$$

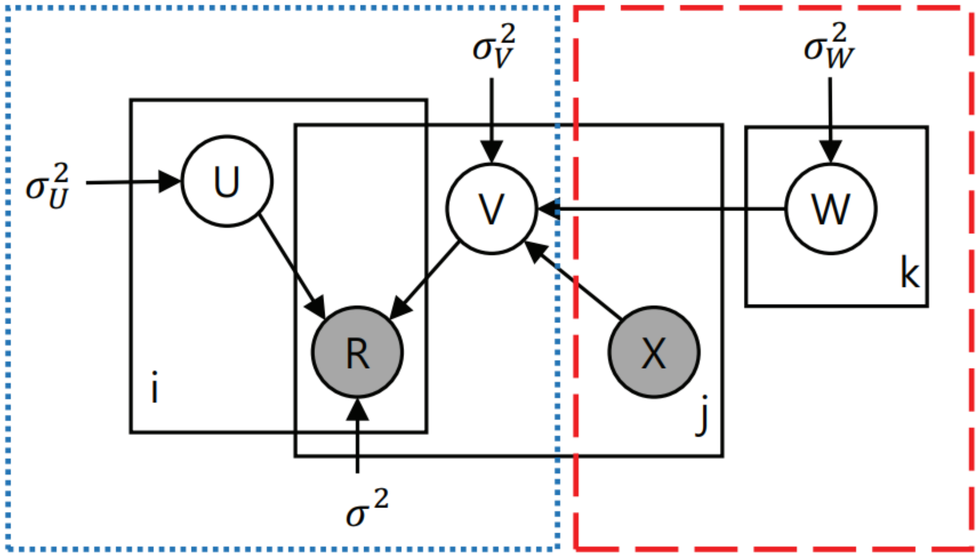


Figure 1: Graphical model of ConvMF model: PMF part in left (dotted-blue); CNN part in right (dashed-red)

Model description

- Convolutional Matrix Factorization
  - Internal weights W for CNN
  - Document item Xj
  - Gaussian noise epsilon

$$v_j = cnn(W, X_j) + \epsilon_j$$

$$\epsilon_j \sim N(0, \sigma_V^2 I)$$

$$p(W|\sigma_W^2) = \prod_k N(w_k|0, \sigma_W^2)$$

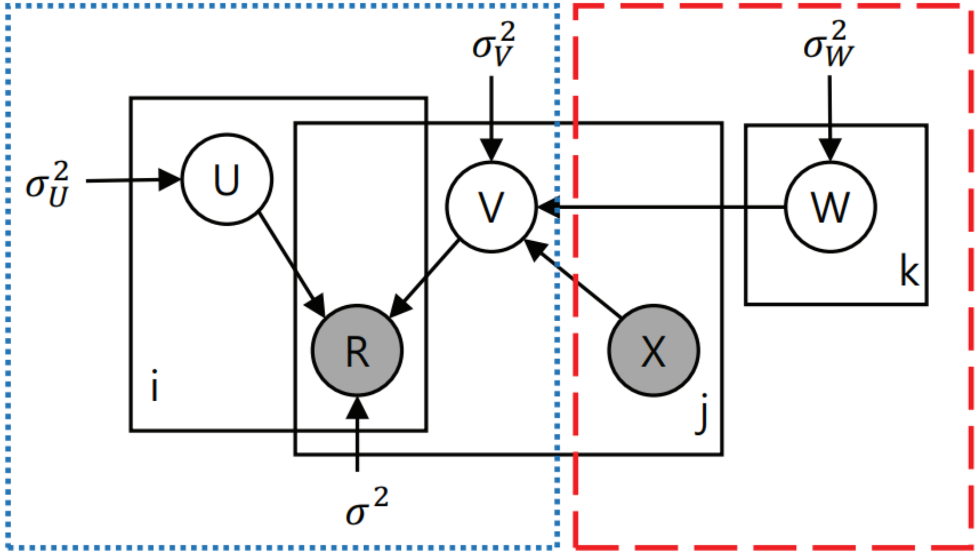


Figure 1: Graphical model of ConvMF model: PMF part in left (dotted-blue); CNN part in right (dashed-red)

$$p(V|W, X, \sigma_V^2) = \prod_j^M N(v_j|cnn(W, X_j), \sigma_V^2 I)$$

Model description

- Convolutional Matrix Factorization-CNN architecture
  - Embedding layer
    transforms raw document into a matrix
    represent document as concatenating
    word vectors

$$D = \begin{bmatrix} \cdots & | & | & | & \cdots \\ & w_{i-1} & w_i & w_{i+1} & \\ & | & | & | & \end{bmatrix}$$

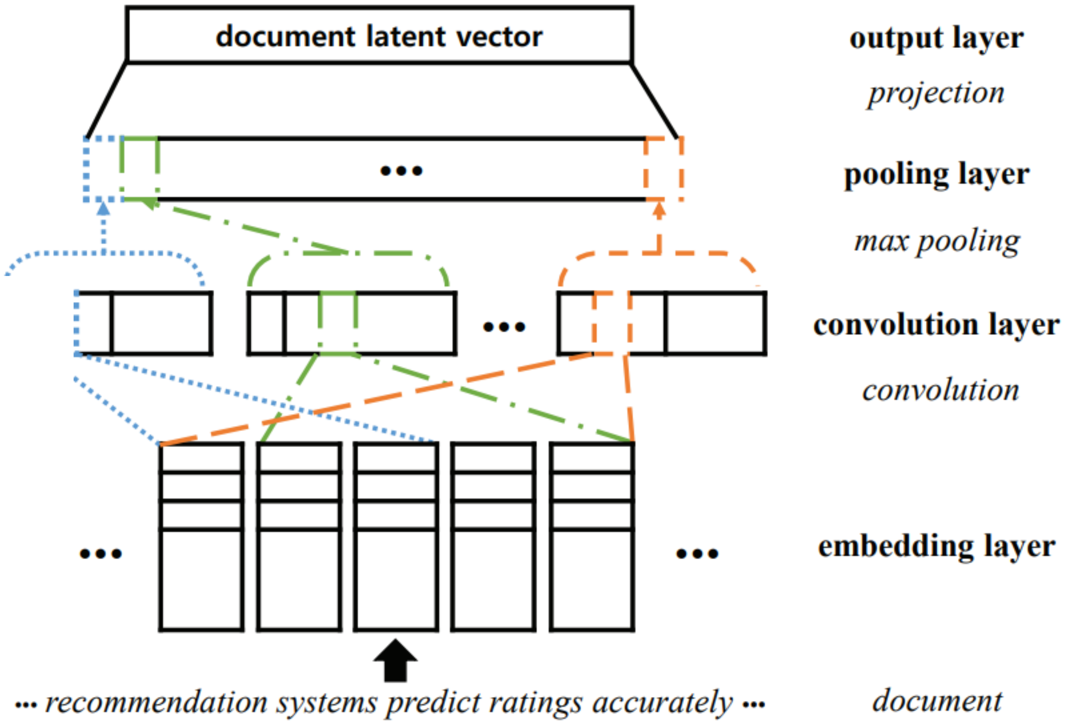


Figure 2: Our CNN architecture for ConvMF

Model description

- Convolutional Matrix Factorization-CNN architecture
  - Convolution layer

    Gets contextual feature

$$c_i^j = f(W_c^j * D_{(:,i:(i+ws-1))} + b_c^j)$$

$$c^j = [\, c_1^j,\ c_2^j,\ \ldots,\ c_i^j,\ \ldots,\ c_{l-ws+1}^j]$$

make multiple feature(j=1,2,....nc)

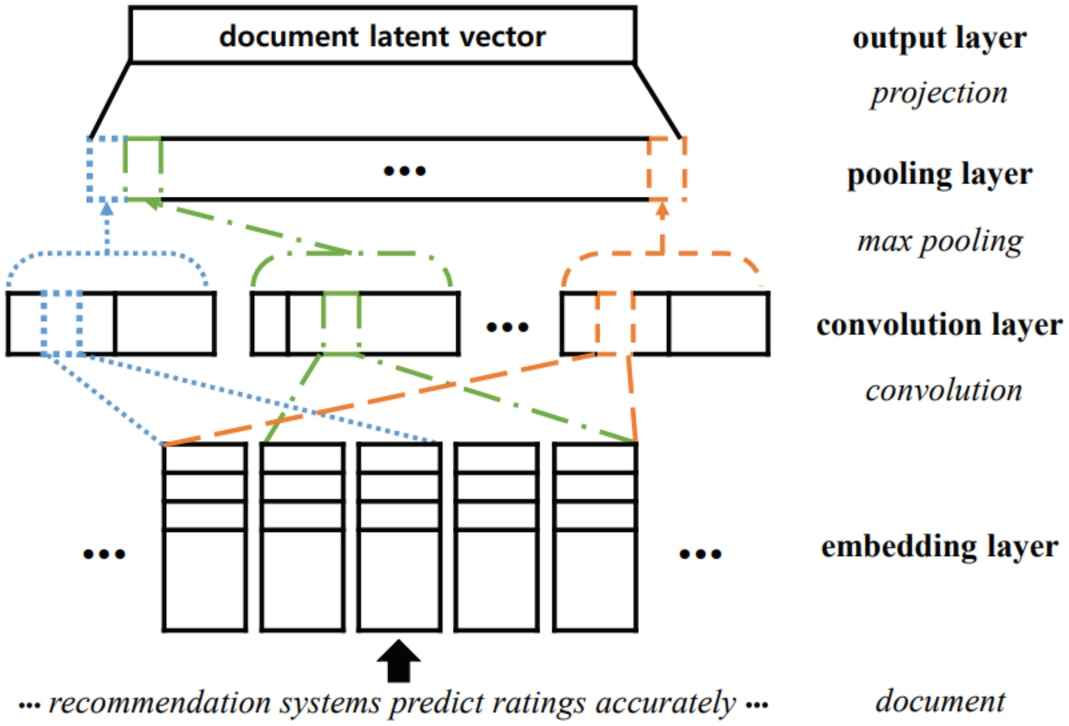


Figure 2: Our CNN architecture for ConvMF

Model description

- Convolutional Matrix Factorization-CNN architecture
  - Pooling layer

    Extracts representative features from the convolution layer

    deals with variable length of documents by pooling operation

    ->make fixed-length feature vector

$$d_f = [\max(c^1), \max(c^2), \ldots, \max(c^j), \ldots, \max(c^{n_c})]$$

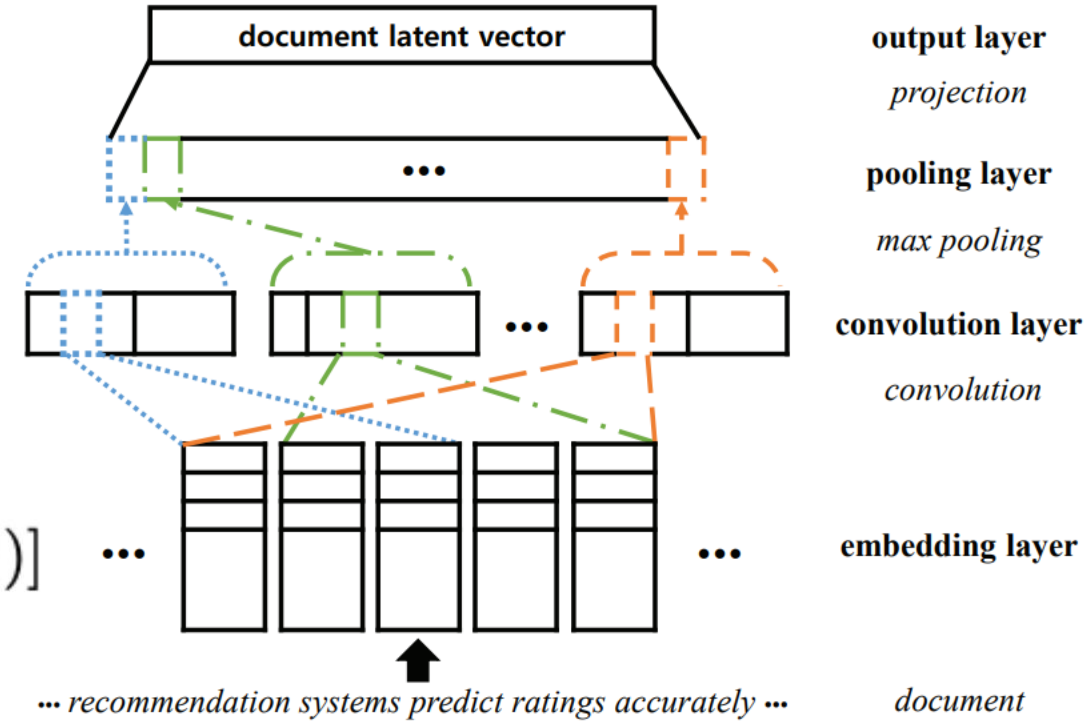


Figure 2: Our CNN architecture for ConvMF

Model description

- Convolutional Matrix Factorization-CNN architecture
  - Output layer
  - Project df to k-dimensional vector

$$s = \tanh(W_{f_2}\{\tanh(W_{f_1} d_f + b_{f_1})\} + b_{f_2})$$

where $W_{f_1} \in \mathbb{R}^{f \times n_c}$, $W_{f_2} \in \mathbb{R}^{k \times f}$ are projection matrices, and $b_{f_1} \in \mathbb{R}^{f}, b_{f_2} \in \mathbb{R}^{k}$ is a bias vector for W_{f_1}, W_{f_2} with $s \in \mathbb{R}^{k}$.

$$s_j = cnn(W, X_j)$$

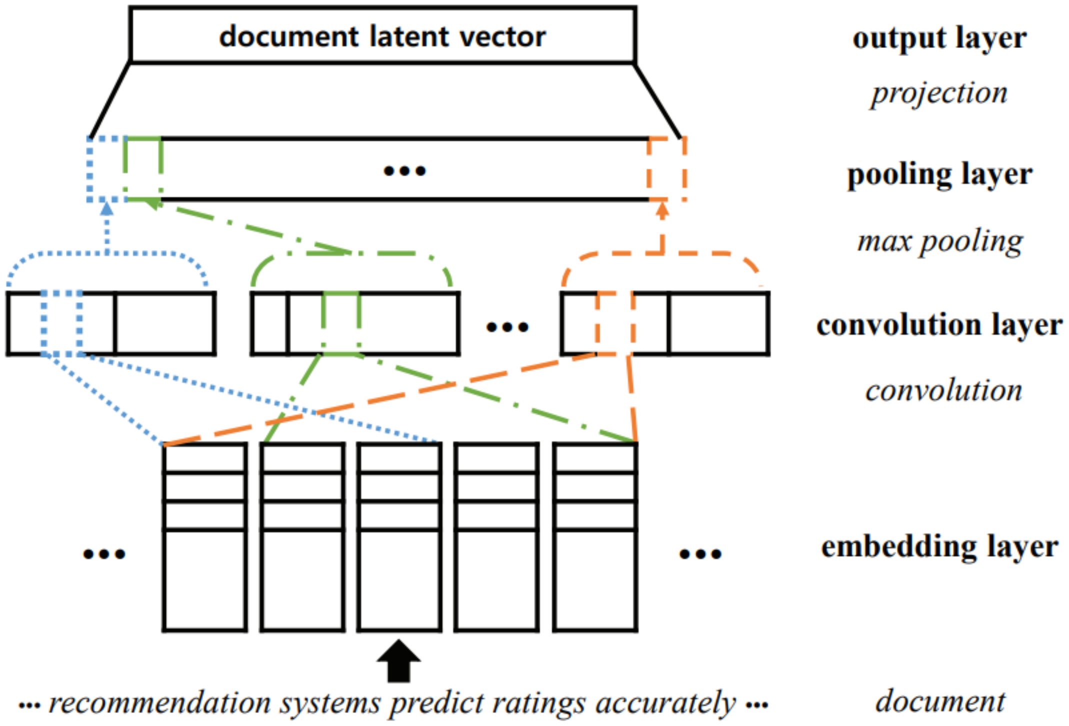


Figure 2: Our CNN architecture for ConvMF

Model description

- Optimization Methodology
  - Use maximum aposteriori(MAP)

$$\max_{U,V,W} p(U,V,W|R,X,\sigma^2,\sigma_U^2,\sigma_V^2,\sigma_W^2)$$
$$= \max_{U,V,W} [p(R|U,V,\sigma^2)p(U|\sigma_U^2)p(V|W,X,\sigma_V^2)p(W|\sigma_W^2)]$$

$$\begin{aligned}\mathcal{L}(U,V,W) &= \sum_i^N \sum_j^M \frac{I_{ij}}{2}(r_{ij} - u_i^T v_j)_2 + \frac{\lambda_U}{2}\sum_i^N \|u_i\|_2 \\ &+ \frac{\lambda_V}{2}\sum_j^M \|v_j - cnn(W,X_j)\|_2 + \frac{\lambda_W}{2}\sum_k^{|w_k|}\|w_k\|_2,\end{aligned} \quad (6)$$

where λ_U is σ^2/σ_U^2, λ_V is σ^2/σ_V^2, and λ_W is σ^2/σ_W^2.

Model description

$$r_{ij} \approx \mathbb{E}[r_{ij}|u_i^T v_j, \sigma^2]$$
$$= u_i^T v_j = u_i^T (cnn(W, X_j) + \epsilon_j)$$

- Optimization Methodology
  - Coordinate descent

$$u_i \leftarrow (V I_i V^T + \lambda_U I_K)^{-1} V R_i \quad (7)$$

$$v_j \leftarrow (U I_j U^T + \lambda_V I_K)^{-1} (U R_j + \lambda_V cnn(W, X_j)) \quad (8)$$

  - W cannot be optimized by an analytic solution as we can do for U and V
  - We can observe that L can be interpreted as a squared error function

$$\mathcal{E}(W) = \frac{\lambda_V}{2} \sum_j^M \|(v_j - cnn(W, X_j))\|^2 + \frac{\lambda_W}{2} \sum_k^{|w_k|} \|w_k\|^2 + \text{constant}$$

Experiment

- Dataset
  - ML-1m ML-10m AIV
  - Users' rating +reviews on items/item description documents(AIV)
  - Movielens-add IMDB data

Dataset	# users	# items	# ratings	density
ML-1m	6,040	3,544	993,482	4.641%
ML-10m	69,878	10,073	9,945,875	1.413%
AIV	29,757	15,149	135,188	0.030%

Table 1: Data statistic on three real-world datasets

  - Compared with various model(PMF,CTR,CDL)
  - Evaluated with RMSE

Experiment

Model	Dataset		
	ML-1m	ML-10m	AIV
PMF	0.8971	0.8311	1.4118
CTR	0.8969	0.8275	1.5496
CDL	0.8879	0.8186	1.3594
ConvMF	**0.8531**	**0.7958**	**1.1337**
ConvMF+	**0.8549**	**0.7930**	**1.1279**
Improve	3.92%	2.79%	16.60%

Table 3: Overall test RMSE

Model	Ratio of training set to the entire dataset (density)						
	20% (0.93%)	30% (1.39%)	40% (1.86%)	50% (2.32%)	60% (2.78%)	70% (3.25%)	80% (3.71%)
PMF	1.0168	0.9711	0.9497	0.9354	0.9197	0.9083	0.8971
CTR	1.0124	0.9685	0.9481	0.9337	0.9194	0.9089	0.8969
CDL	1.0044	0.9639	0.9377	0.9211	0.9068	0.8970	0.8879
ConvMF	**0.9745**	**0.9330**	**0.9063**	**0.8897**	**0.8726**	**0.8676**	**0.8531**
Improve	2.98%	3.20%	3.36%	3.41%	3.77%	3.27%	3.92%

Table 4: Test RMSE over various sparseness of training data on ML-1m dataset

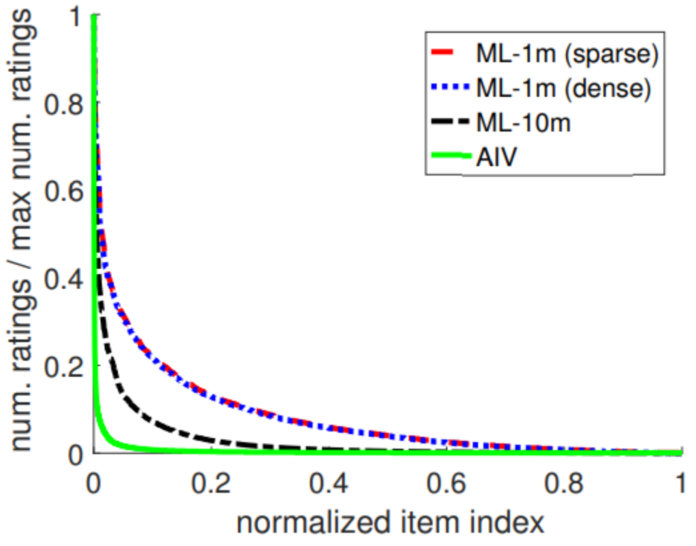


Figure 3: Skewness of the number of ratings for items on each dataset

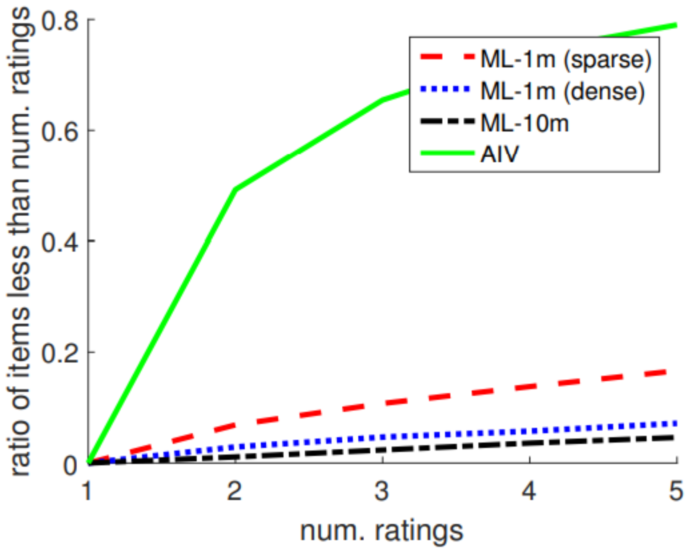


Figure 4: Ratio of items that have less than num. ratings (N) to each entire dataset

Experiment

Model	ML-1m	ML-10m	AIV
	Dataset		
PMF	0.8971	0.8311	1.4118
CTR	0.8969	0.8275	1.5496
CDL	0.8879	0.8186	1.3594
ConvMF	**0.8531**	**0.7958**	**1.1337**
ConvMF+	**0.8549**	**0.7930**	**1.1279**
Improve	3.92%	2.79%	16.60%

Table 3: Overall test RMSE

Improve -0.22% 0.35% 0.51%

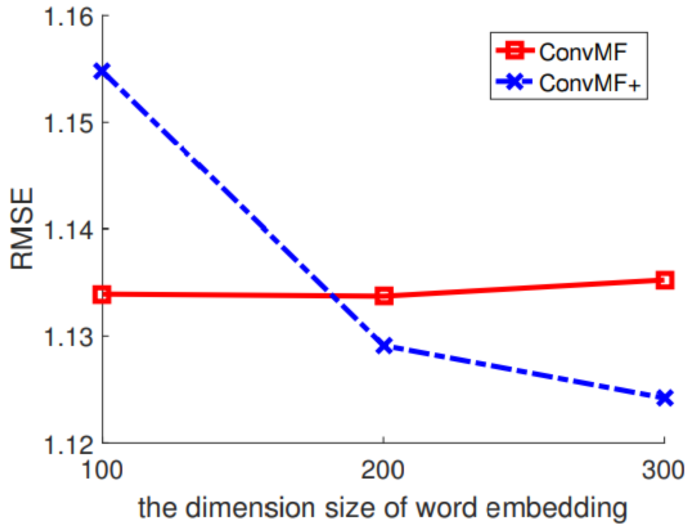


Figure 5: The effects of the dimension size of word embedding on Amazon dataset

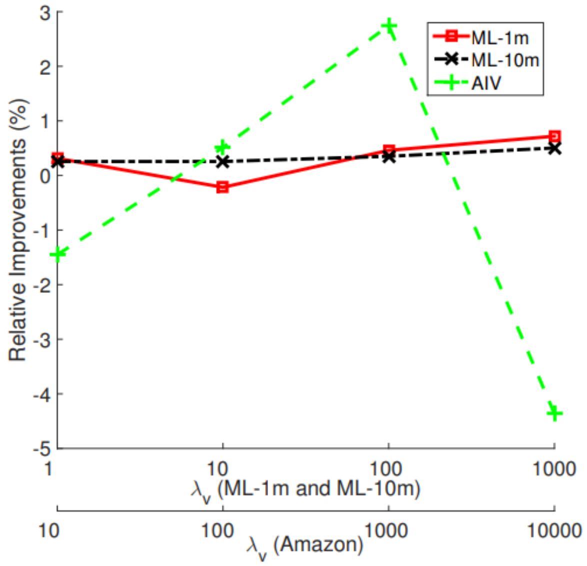


Figure 6: Relative improvements of ConvMF+ over ConvMF

Experiment

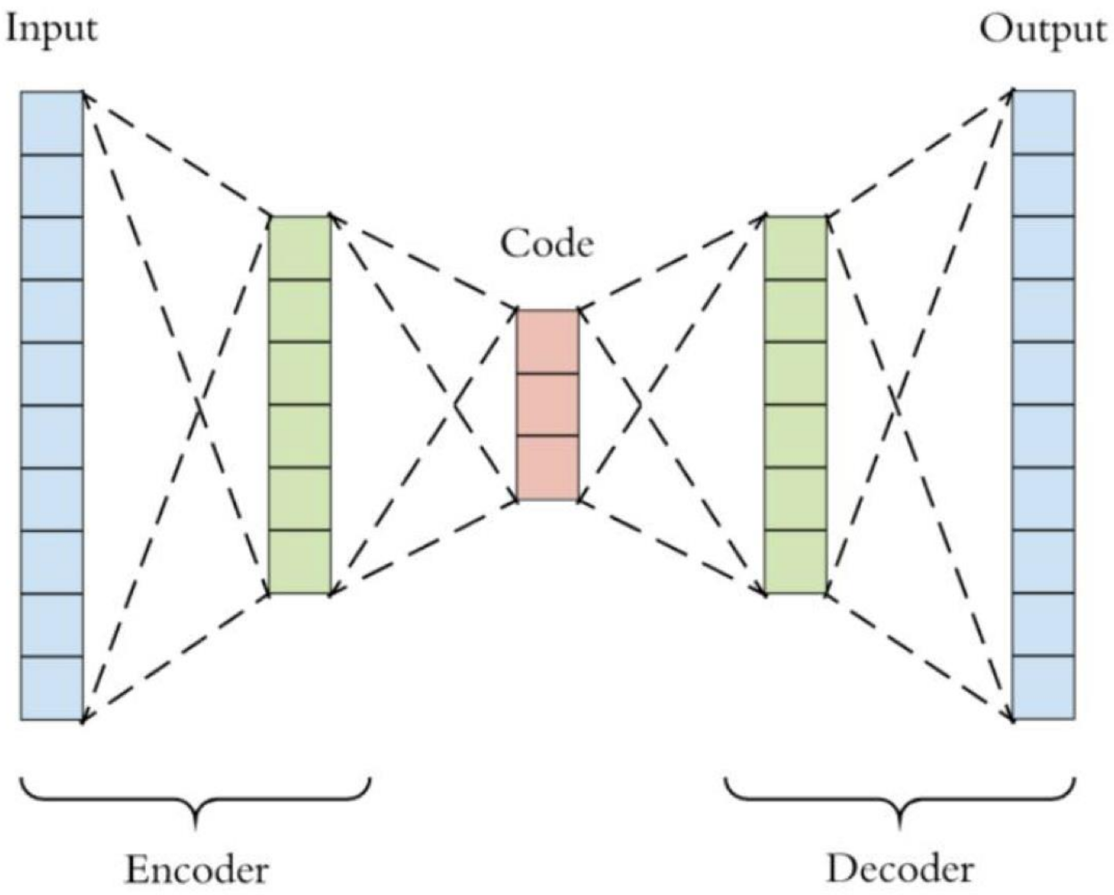
Phrase captured by W_c^{11}	$\max(c^{11})$	Phrase captured by W_c^{86}	$\max(c^{86})$
people **trust** the man	***0.0704***	betray his **trust** finally	***0.1009***
Test phrases for W_c^{11}	**c_{test}^{11}**	**Test phrases for W_c^{86}**	**c_{test}^{86}**
people **believe** the man	***0.0391***	betray his **believe** finally	*0.0682*
people **faith** the man	*0.0374*	betray his **faith** finally	***0.0693***
people **tomas** the man	*0.0054*	betray his **tomas** finally	*0.0480*

Table 5: Case study on two shared weights of ConvMF

Conclusion

- ConvMF
  - ConvMF significantly outperforms other competitors when dataset is extremely sparse
  - The improvements of ConvMF over the competitors increase further even when dataset becomes dense
  - Pre-trained word embedding model helps improve the performance of ConvMF when dataset is extremely sparse
  - Best performing parameters verify that ConvMF well alleviates data sparsity
  - ConvMF indeed captures subtle contextual differences

AutoRec



$$Encoder\ \phi : X \to F$$

$$Decoder\ \varphi : F \to X$$

$$\phi, \varphi = argmin||X - (\phi \circ \varphi)X||_2^2$$

AutoRec

$$\min_{\theta} \sum_{\mathbf{r} \in \mathbf{S}} ||\mathbf{r} - h(\mathbf{r}; \theta)||_2^2,$$

$$h(\mathbf{r}; \theta) = f\left(\mathbf{W} \cdot g(\mathbf{V}\mathbf{r} + \boldsymbol{\mu}) + \mathbf{b}\right)$$

$\theta = \{W, V, \mu, b\}$:parameter

$W \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{k \times m}$: Weight matrix

$\mu \in \mathbb{R}^{k}, b \in \mathbb{R}^{m}$: bias

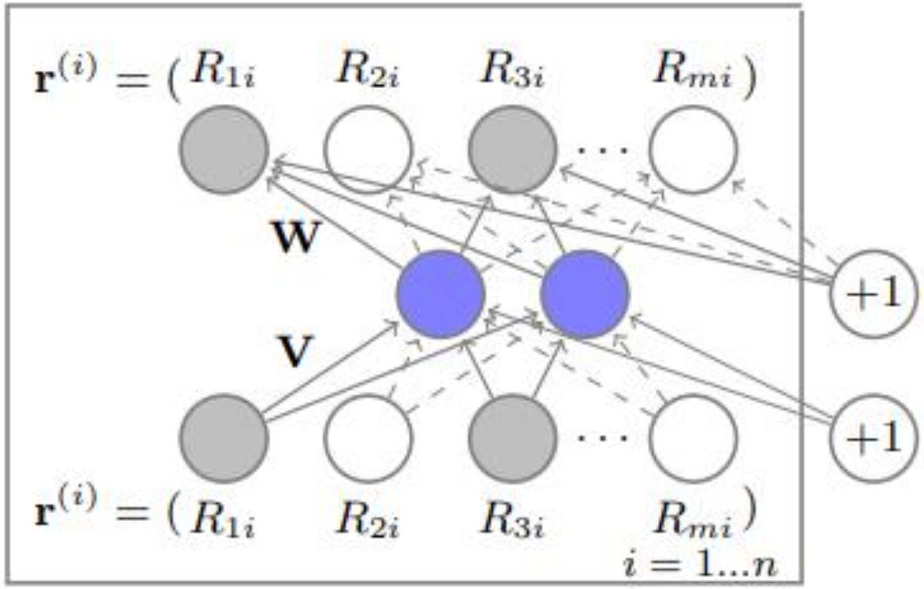


Figure 1: Item-based AutoRec model. We use plate notation to indicate that there are n copies of the neural network (one for each item), where $\mathbf{W}$ and $\mathbf{V}$ are tied across all copies.

$$\min_{\theta} \sum_{i=1}^{n} ||r^{(i)} - h(r^{(i)}; \theta)||_{\mathcal{O}}^2 + \frac{\lambda}{2} \cdot (||W||_F^2 + ||V||_F^2)$$

AutoRec

	ML-1M	ML-10M
U-RBM	0.881	0.823
I-RBM	0.854	0.825
U-AutoRec	0.874	0.867
I-AutoRec	**0.831**	**0.782**

(a)

$f(\cdot)$	$g(\cdot)$	RMSE
Identity	Identity	0.872
Sigmoid	Identity	0.852
Identity	Sigmoid	**0.831**
Sigmoid	Sigmoid	0.836

(b)

	ML-1M	ML-10M	Netflix
BiasedMF	0.845	0.803	0.844
I-RBM	0.854	0.825	-
U-RBM	0.881	0.823	0.845
LLORMA	0.833	**0.782**	0.834
I-AutoRec	**0.831**	**0.782**	**0.823**

(c)

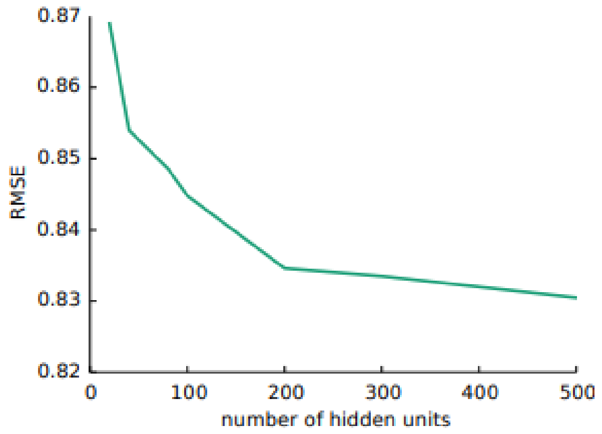


Figure 2: RMSE of I-AutoRec on Movielens 1M as the number of hidden units k varies.

Implementation

```
class ConvMF(tf.keras.Model):
    def __init__(self, num_users, num_items, latent_dim, num_filters, a,b,c,d,e):
        super(ConvMF, self).__init__()

        self.num_users = num_users
        self.num_items = num_items
        self.latent_dim = latent_dim

        self.user_latent_matrix = tf.Variable(
            initial_value=tf.random.normal(shape=(num_users, latent_dim), mean=0.0, stddev=0.01),
            trainable=True
        )
        self.item_latent_matrix = tf.Variable(
            initial_value=tf.random.normal(shape=(num_items, latent_dim), mean=0.0, stddev=0.01),
            trainable=True
        )
        self.item_bias = tf.Variable(
            initial_value=tf.zeros(shape=(num_items, 1)),
            trainable=True
        )
```

```
#change to word vector
self.word_embedding_matrix = np.random.randn(10, 18)
self.genre_conv = tf.keras.layers.Conv1D(
    filters=num_filters,
    kernel_size=2,
    activation='relu',
)
self.genre_max_pool = tf.keras.layers.MaxPool1D(
    pool_size=2,
    strides=2
)
self.genre_flatten = tf.keras.layers.Flatten()

self.fc_layer1 = tf.keras.layers.Dense(units=latent_dim, activation='tanh')
self.fc_layer2 = tf.keras.layers.Dense(units=latent_dim, activation='tanh')
```

Implementation

```
def call(self, inputs):
    inputs = tf.reshape(inputs, [-1, 7])
    user_latent_factors = tf.nn.embedding_lookup(self.user_latent_matrix, inputs[:, 0])
    item_latent_factors = tf.nn.embedding_lookup(self.item_latent_matrix, inputs[:, 1])
    item_bias = tf.nn.embedding_lookup(self.item_bias, inputs[:, 1])
    inputs = tf.constant(inputs)
    result_matrix = self.word_embedding_matrix[:, [inputs[:, 2], inputs[:, 3], inputs[:, 4], inputs[:, 5], inputs[:, 6]]]
    user_item_matrix = tf.matmul(user_latent_factors, item_latent_factors, transpose_b=True)
    #result_matrix = tf.expand_dims(user_item_matrix, axis=1)
    feature_map = self.genre_conv(result_matrix)
    feature_map = self.genre_max_pool(feature_map)
    feature_map = self.genre_flatten(feature_map)
    out = self.fc_layer1(feature_map)
    out = self.fc_layer2(out)
    predicted_rating = tf.reduce_sum(out, axis=1, keepdims=True) + item_bias
    return predicted_rating
```

```
def sentence_to_vector(sentence):
    genres = ['Action', 'Adventure', 'Animation', "Children's", 'Comedy', 'Crime', 'Drama', 'Romance', 'Thriller',
              'Western', 'Horror', 'Fantasy', 'Sci-Fi', 'Documentary', 'War', 'Mystery', 'Musical']
    genre_to_num = {genres[i]: i + 1 for i in range(len(genres))}
    vector = [0] * len(sentence.split())
    for i, word in enumerate(sentence.split()):
        if word in genre_to_num:
            vector[i] = genre_to_num[word]
    return vector
```

Implementation

```python
for epoch in range(100):
    # Training loop

    for row in train_data.values:
        user_id, item_id, rating, genre = row[1], row[0], row[2], row[4]
        genre = sentence_to_vector(genre)
        genre = pad_sequences([genre], maxlen=5, padding='post', value=0)
        genre = genre[0]
        with tf.GradientTape() as tape:
            inputs = tf.stack([user_id, item_id, genre[0],genre[1],genre[2],genre[3],genre[4]], axis=-1)
            predictions = model(inputs)
            loss = loss_fn(rating, predictions)

        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Implementation

```python
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
    # Evaluation loop
    avg_loss = 0.0
    for row in test_data.values:
        user_id, item_id, rating, genre = row[1], row[0], row[2], row[4]
        genre = sentence_to_vector(genre)
        genre = pad_sequences([genre], maxlen=5, padding='post', value=0)
        genre = genre[0]
        inputs = tf.stack([user_id, item_id, genre[0], genre[1], genre[2], genre[3], genre[4]], axis=-1)
        predictions = model(inputs)
        avg_loss += loss_fn(rating, predictions)
    avg_loss /= len(test_data)
    print("Epoch {}: Test loss = {}".format(epoch, avg_loss))
```

```
Epoch 0: Test loss = 1.5095343589782715
Epoch 1: Test loss = 1.4805691242218018
Epoch 2: Test loss = 1.461288571357727
Epoch 3: Test loss = 1.443916916847229
Epoch 4: Test loss = 1.4289628267288208
Epoch 5: Test loss = 1.4151443243026733
Epoch 6: Test loss = 1.4030861854553223
Epoch 7: Test loss = 1.391825556755066
Epoch 8: Test loss = 1.3811259269714355
Epoch 9: Test loss = 1.371309757232666
Epoch 10: Test loss = 1.3623532056808472
Epoch 11: Test loss = 1.352412462234497
Epoch 12: Test loss = 1.3445348739624023
Epoch 13: Test loss = 1.3376710414886475
Epoch 14: Test loss = 1.331301212310791
Epoch 15: Test loss = 1.3255774974822998
Epoch 16: Test loss = 1.3208509683609009
Epoch 17: Test loss = 1.3166801929473877
Epoch 18: Test loss = 1.3129180669784546
```

```
Epoch 19: Test loss = 1.3098766803741455
Epoch 20: Test loss = 1.3079893589019775
Epoch 21: Test loss = 1.3051671981811523
Epoch 22: Test loss = 1.3037588596343994
Epoch 23: Test loss = 1.3021999597549438
Epoch 24: Test loss = 1.2993050813674927
Epoch 25: Test loss = 1.2971255779266357
Epoch 26: Test loss = 1.2955207824707031
Epoch 27: Test loss = 1.2941497564315796
Epoch 28: Test loss = 1.2914210557937622
Epoch 29: Test loss = 1.2911709547042847
```