15 | Recommender Systems

Shuai Zhang (*Amazon*), **Aston Zhang** (*Amazon*), and **Yi Tay** (*Nanyang Technological University*)

Recommender systems are widely employed in industry and are ubiquitous in our daily lives. These systems are utilized in a number of areas such as online shopping sites (e.g., amazon.com), music/movie services site (e.g., Netflix and Spotify), mobile application stores (e.g., IOS app store and google play), online advertising, just to name a few.

The major goal of recommender systems is to help users discover relevant items such as movies to watch, text to read or products to buy, so as to create a delightful user experience. Moreover, recommender systems are among the most powerful machine learning systems that online retailers implement in order to drive incremental revenue. Recommender systems are replacements of search engines by reducing the efforts in proactive searches and surprising users with offers they never searched for. Many companies managed to position themselves ahead of their competitors with the help of more effective recommender systems. As such, recommender systems are central to not only our everyday lives but also highly indispensable in some industries.

In this chapter, we will cover the fundamentals and advancements of recommender systems, along with exploring some common fundamental techniques for building recommender systems with different data sources available and their implementations. Specifically, you will learn how to predict the rating a user might give to a prospective item, how to generate a recommendation list of items and how to predict the click-through rate from abundant features. These tasks are commonplace in real-world applications. By studying this chapter, you will get hands-on experience pertaining to solving real world recommendation problems with not only classical methods but the more advanced deep learning based models as well.

15.1 Overview of Recommender Systems

In the last decade, the Internet has evolved into a platform for large-scale online services, which profoundly changed the way we communicate, read news, buy products, and watch movies. In the meanwhile, the unprecedented number of items (we use the term *item* to refer to movies, news, books, and products.) offered online requires a system that can help us discover items that we preferred. Recommender systems are therefore powerful information filtering tools that can facilitate personalized services and provide tailored experience to individual users. In short, recommender systems play a pivotal role in utilizing the wealth of data available to make choices manageable. Nowadays, recommender systems are at the core of a number of online services providers such as Amazon, Netflix, and YouTube. Recall the example of Deep learning books recommended by Amazon in Fig. 1.3.3. The benefits of employing recommender systems are two-folds: On the one hand, it can largely reduce users' effort in finding items and alleviate the issue of information overload. On the other hand, it can add business value to online service providers and is an important source of revenue. This chapter will introduce the fundamental concepts, classic models and recent advances with deep learning in the field of recommender systems, together with implemented examples.

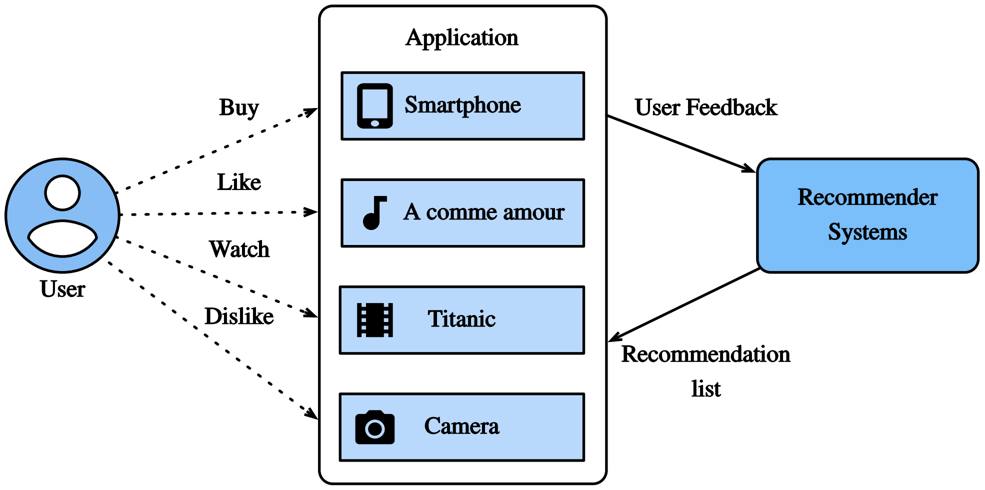


Fig. 15.1.1: Illustration of the Recommendation Process

15.1.1 Collaborative Filtering

We start the journey with the important concept in recommender systems—collaborative filtering (CF), which was first coined by the Tapestry system (Goldberg et al., 1992), referring to "people collaborate to help one another perform the filtering process in order to handle the large amounts of email and messages posted to newsgroups". This term has been enriched with more senses. In a broad sense, it is the process of filtering for information or patterns using techniques involving collaboration among multiple users, agents, and data sources. CF has many forms and numerous CF methods proposed since its advent.

Overall, CF techniques can be categorized into: memory-based CF, model-based CF, and their hybrid (Su & Khoshgoftaar, 2009). Representative memory-based CF techniques are nearest neighbor-based CF such as user-based CF and item-based CF (Sarwar et al., 2001). Latent factor models such as matrix factorization are examples of model-based CF. Memory-based CF has limitations in dealing with sparse and large-scale data since it computes the similarity values based on common items. Model-based methods become more popular with its better capability in dealing with sparsity and scalability. Many model-based CF approaches can be extended with neural networks, leading to more flexible and scalable models with the computation acceleration in deep learning (Zhang et al., 2019). In general, CF only uses the user-item interaction data to make predictions and recommendations. Besides CF, content-based and context-based recommender systems are also useful in incorporating the content descriptions of items/users and contextual signals such as timestamps and locations. Obviously, we may need to adjust the model types/structures when different input data is available.

15.1.2 Explicit Feedback and Implicit Feedback

To learn the preference of users, the system shall collect feedback from them. The feedback can be either explicit or implicit (Hu et al., 2008). For example, IMDB[231] collects star ratings ranging from one to ten stars for movies. YouTube provides the thumbs-up and thumbs-down buttons for users to show their preferences. It is apparent that gathering explicit feedback requires users to indicate their interests proactively. Nonetheless, explicit feedback is not always readily available as many users may be reluctant to rate products. Relatively speaking, implicit feedback is often readily available since it is mainly concerned with modeling implicit behavior such user clicks. As such, many recommender systems are centered on implicit feedback which indirectly reflects user's opinion through observing user behavior. There are diverse forms of implicit feedback including purchase history, browsing history, watches and even mouse movements. For example, a user that purchased many books by the same author probably likes that author. Note that implicit feedback is inherently noisy. We can only *guess* their preferences and true motives. A user watched a movie does not necessarily indicate a positive view of that movie.

15.1.3 Recommendation Tasks

A number of recommendation tasks have been investigated in the past decades. Based on the domain of applications, there are movies recommendation, news recommendations, point-of-interest recommendation (Ye et al., 2011) and so forth. It is also possible to differentiate the tasks based on the types of feedback and input data, for example, the rating prediction task aims to predict the explicit ratings. Top-n recommendation (item ranking) ranks all items for each user personally based on the implicit feedback. If time-stamp information is also included, we can build sequence-aware recommendation (Quadrana et al., 2018). Another popular task is called click-through rate prediction, which is also based on implicit feedback, but various categorical features can be utilized. Recommending for new users and recommending new items to existing users are called cold-start recommendation (Schein et al., 2002).

Summary

- Recommender systems are important for individual users and industries. Collaborative filtering is a key concept in recommendation.
- There are two types of feedbacks: implicit feedback and explicit feedback. A number recommendation tasks have been explored during the last decade.

Exercises

1. Can you explain how recommender systems influence your daily life?
2. What interesting recommendation tasks do you think can be investigated?

[231] https://www.imdb.com/

15.2 The MovieLens Dataset

There are a number of datasets that are available for recommendation research. Amongst them, the MovieLens[233] dataset is probably the one of the more popular ones. MovieLens is a non-commercial web-based movie recommender system. It is created in 1997 and run by GroupLens, a research lab at the University of Minnesota, in order to gather movie rating data for research purposes. MovieLens data has been critical for several research studies including personalized recommendation and social psychology.

15.2.1 Getting the Data

The MovieLens dataset is hosted by the GroupLens[234] website. Several versions are available. We will use the MovieLens 100K dataset (Herlocker et al., 1999). This dataset is comprised of $100,000$ ratings, ranging from 1 to 5 stars, from 943 users on 1682 movies. It has been cleaned up so that each user has rated at least 20 movies. Some simple demographic information such as age, gender, genres for the users and items are also available. We can download the ml-100k.zip[235] and extract the `u.data` file, which contains all the $100,000$ ratings in the csv format. There are many other files in the folder, a detailed description for each file can be found in the README[236] file of the dataset.

To begin with, let's import the packages required to run this section's experiments.

```python
import d2l
from mxnet import gluon, np
import pandas as pd
```

Then, we download the MovieLens 100k dataset and load the interactions as `DataFrame`.

```python
# Saved in the d2l package for later use
d2l.DATA_HUB['ml-100k'] = (
    'http://files.grouplens.org/datasets/movielens/ml-100k.zip',
    'cd4dcac4241c8a4ad7badc7ca635da8a69dddb83')
def read_data_ml100k():
    data_dir = d2l.download_extract('ml-100k')
    names = ['user_id', 'item_id', 'rating', 'timestamp']
    data = pd.read_csv(data_dir+'u.data', '\t', names=names, engine='python')
    num_users = data.user_id.unique().shape[0]
    num_items = data.item_id.unique().shape[0]
    return data, num_users, num_items
```

233 https://movielens.org/
234 https://grouplens.org/datasets/movielens/
235 http://files.grouplens.org/datasets/movielens/ml-100k.zip
236 http://files.grouplens.org/datasets/movielens/ml-100k-README.txt

15.2.2 Statistics of the Dataset

Let's load up the data and inspect the first five records manually. It is an effective way to learn the data structure and verify that they have been loaded properly.

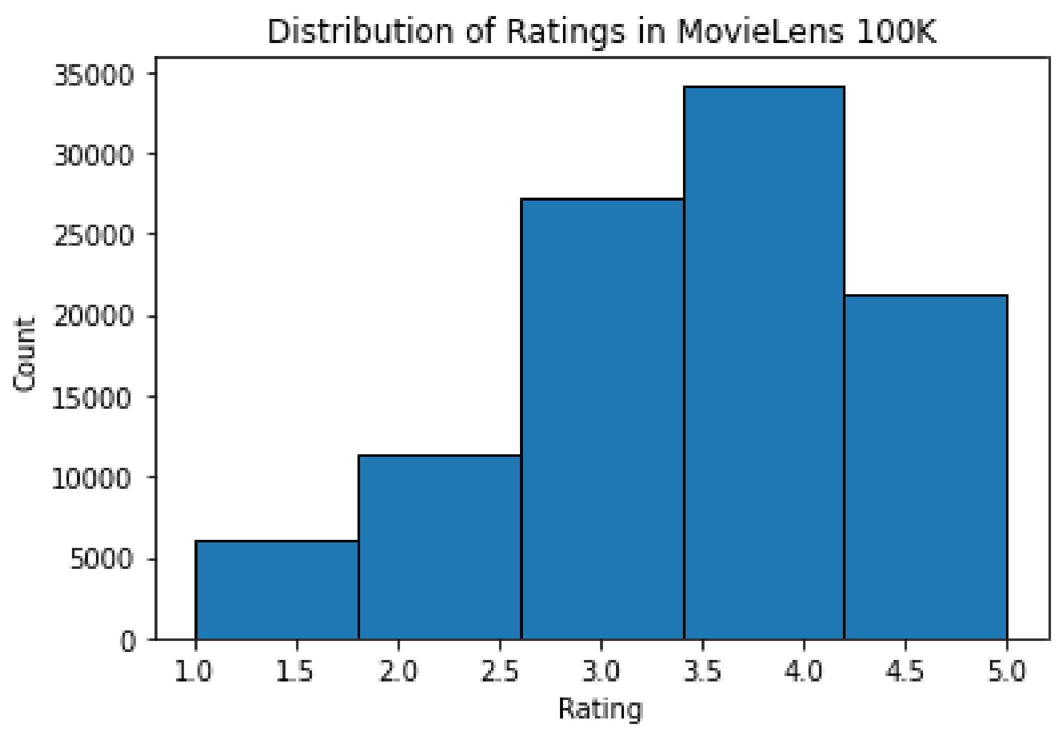
```
data, num_users, num_items = read_data_ml100k()
sparsity = 1 - len(data) / (num_users * num_items)
print('number of users: %d, number of items: %d.' % (num_users, num_items))
print('matrix sparsity: %f' % sparsity)
print(data.head(5))
```

```
Downloading ../data/ml-100k.zip from http://files.grouplens.org/datasets/movielens/ml-100k.
↪zip...
number of users: 943, number of items: 1682.
matrix sparsity: 0.936953
   user_id  item_id  rating  timestamp
0      196      242       3  881250949
1      186      302       3  891717742
2       22      377       1  878887116
3      244       51       2  880606923
4      166      346       1  886397596
```

We can see that each line consists of four columns, including "user id" 1-943, "item id" 1-1682, "rating" 1-5 and "timestamp". We can construct an interaction matrix of size $n \times m$, where n and m are the number of users and the number of items respectively. This dataset only records the existing ratings, so we can also call it rating matrix and we will use interaction matrix and rating matrix interchangeably in case that the values of this matrix represent exact ratings. Most of the values in the rating matrix are unknown as users have not rated the majority of movies. We also show the sparsity of this dataset. The sparsity is defined as `1 - number of nonzero entries / ( number of users * number of items)`. Clearly, the interaction matrix is extremely sparse (i.e., sparsity = 93.695%). Real world datasets may suffer from a greater extent of sparsity and has been a long-standing challenge in building recommender systems. A viable solution is to use additional side information such as user/item features to alleviate the sparsity.

We then plot the distribution of the count of different ratings. As expected, it appears to be a normal distribution, with most ratings centered at 3-4.

```
d2l.plt.hist(data['rating'], bins=5, ec='black')
d2l.plt.xlabel("Rating")
d2l.plt.ylabel("Count")
d2l.plt.title("Distribution of Ratings in MovieLens 100K")
d2l.plt.show()
```



15.2.3 Splitting the dataset

We split the dataset into training and test sets. The following function provides two split modes including `random` and `seq-aware`. In the `random` mode, the function splits the 100k interactions randomly without considering timestamp and uses the 90% of the data as training samples and the rest 10% as test samples by default. In the `seq-aware` mode, we leave out the item that a user rated most recently for test, and users' historical interactions as training set. User historical interactions are sorted from oldest to newest based on timestamp. This mode will be used in the sequence-aware recommendation section.

```python
# Saved in the d2l package for later use
def split_data_ml100k(data, num_users, num_items,
                      split_mode="random", test_ratio=0.1):
    """Split the dataset in random mode or seq-aware mode."""
    if split_mode == "seq-aware":
        train_items, test_items, train_list = {}, {}, []
        for line in data.itertuples():
            u, i, rating, time = line[1], line[2], line[3], line[4]
            train_items.setdefault(u, []).append((u, i, rating, time))
            if u not in test_items or test_items[u][-1] < time:
                test_items[u] = (i, rating, time)
        for u in range(1, num_users + 1):
            train_list.extend(sorted(train_items[u], key=lambda k: k[3]))
        test_data = [(key, *value) for key, value in test_items.items()]
        train_data = [item for item in train_list if item not in test_data]
        train_data = pd.DataFrame(train_data)
        test_data = pd.DataFrame(test_data)
    else:
        mask = [True if x == 1 else False for x in np.random.uniform(
            0, 1, (len(data))) < 1 - test_ratio]
```

(continues on next page)

(continued from previous page)

```
        neg_mask = [not x for x in mask]
        train_data, test_data = data[mask], data[neg_mask]
    return train_data, test_data
```

Note that it is good practice to use a validation set in practice, apart from only a test set. However, we omit that for the sake of brevity. In this case, our test set can be regarded as our held-out validation set.

15.2.4 Loading the data

After dataset splitting, we will convert the training set and test set into lists and dictionaries/matrix for the sake of convenience. The following function reads the dataframe line by line and enumerates the index of users/items start from zero. The function then returns lists of users, items, ratings and a dictionary/matrix that records the interactions. We can specify the type of feedback to either `explicit` or `implicit`.

```
# Saved in the d2l package for later use
def load_data_ml100k(data, num_users, num_items, feedback="explicit"):
    users, items, scores = [], [], []
    inter = np.zeros((num_items, num_users)) if feedback == "explicit" else {}
    for line in data.itertuples():
        user_index, item_index = int(line[1] - 1), int(line[2] - 1)
        score = int(line[3]) if feedback == "explicit" else 1
        users.append(user_index)
        items.append(item_index)
        scores.append(score)
        if feedback == "implicit":
            inter.setdefault(user_index, []).append(item_index)
        else:
            inter[item_index, user_index] = score
    return users, items, scores, inter
```

Afterwards, we put the above steps together and it will be used in the next section. The results are wrapped with `Dataset` and `DataLoader`. Note that the `last_batch` of `DataLoader` for training data is set to the `rollover` mode (The remaining samples are rolled over to the next epoch.) and orders are shuffled.

```
# Saved in the d2l package for later use
def split_and_load_ml100k(split_mode="seq-aware", feedback="explicit",
                          test_ratio=0.1, batch_size=256):
    data, num_users, num_items = read_data_ml100k()
    train_data, test_data = split_data_ml100k(
        data, num_users, num_items, split_mode, test_ratio)
    train_u, train_i, train_r, _ = load_data_ml100k(
        train_data, num_users, num_items, feedback)
    test_u, test_i, test_r, _ = load_data_ml100k(
        test_data, num_users, num_items, feedback)
    train_set = gluon.data.ArrayDataset(
        np.array(train_u), np.array(train_i), np.array(train_r))
    test_set = gluon.data.ArrayDataset(
        np.array(test_u), np.array(test_i), np.array(test_r))
```

(continues on next page)

(continued from previous page)

```
    train_iter = gluon.data.DataLoader(
        train_set, shuffle=True, last_batch="rollover",
        batch_size=batch_size)
    test_iter = gluon.data.DataLoader(
        test_set, batch_size=batch_size)
    return num_users, num_items, train_iter, test_iter
```

Summary

- MovieLens datasets are widely used for recommendation research. It is public available and free to use.
- We define functions to download and preprocess the MovieLens 100k dataset for further use in later sections.

Exercises

- What other similar recommendation datasets can you find?
- Go through the https://movielens.org/ site for more information about MovieLens.

15.3 Matrix Factorization

Matrix Factorization (Koren et al., 2009) is a well-established algorithm in the recommender systems literature. The first version of matrix factorization model is proposed by Simon Funk in a famous blog post[238] in which he described the idea of factorizing the interaction matrix. It then became widely known due to the Netflix contest which was held in 2006. At that time, Netflix, a media-streaming and video-rental company, announced a contest to improve its recommender system performance. The best team that can improve on the Netflix baseline, i.e., Cinematch), by 10 percent would win a one million USD prize. As such, this contest attracted a lot of attention to the field of recommender system research. Subsequently, the grand prize was won by the BellKor's Pragmatic Chaos team, a combined team of BellKor, Pragmatic Theory, and BigChaos (you do not need to worry about these algorithms now). Although the final score was the result of an ensemble solution (i.e., a combination of many algorithms), the matrix factorization algorithm played a critical role in the final blend. The technical report the Netflix Grand Prize solution (Toscher et al., 2009) provides a detailed introduction to the adopted model. In this section, we will dive into the details of the matrix factorization model and its implementation.

[238] https://sifter.org/~simon/journal/20061211.html

15.3.1 The Matrix Factorization Model

Matrix factorization is a class of collaborative filtering models. Specifically, the model factorizes the user-item interaction matrix (e.g., rating matrix) into the product of two lower-rank matrices, capturing the low-rank structure of the user-item interactions.

Let $\mathbf{R} \in \mathbb{R}^{m\times n}$ denote the interaction matrix with m users and n items, and the values of $\mathbf{R}$ represent explicit ratings. The user-item interaction will be factorized into a user latent matrix $\mathbf{P} \in \mathbb{R}^{m\times k}$ and an item latent matrix $\mathbf{Q} \in \mathbb{R}^{n\times k}$, where $k \ll m, n$, is the latent factor size. Let $\mathbf{p}_u$ denote the u^th row of $\mathbf{P}$ and $\mathbf{q}_i$ denote the i^th row of $\mathbf{Q}$. For a given item i, the elements of $\mathbf{q}_i$ measure the extent to which the item possesses those characteristics such as the genres and languages of a movie. For a given user u, the elements of $\mathbf{p}_u$ measure the extent of interest the user has in items' corresponding characteristics. These latent factors might measure obvious dimensions as mentioned in those examples or are completely uninterpretable. The predicted ratings can be estimated by

$$\hat{\mathbf{R}} = \mathbf{P}\mathbf{Q}^\top \tag{15.3.1}$$

where $\hat{\mathbf{R}} \in \mathbb{R}^{m\times n}$ is the predicted rating matrix which has the same shape as $\mathbf{R}$. One major problem of this prediction rule is that users/items biases can not be modeled. For example, some users tend to give higher ratings or some items always get lower ratings due to poorer quality. These biases are commonplace in real-world applications. To capture these biases, user specific and item specific bias terms are introduced. Specifically, the predicted rating user u gives to item i is calculated by

$$\hat{\mathbf{R}}_{ui} = \mathbf{p}_u\mathbf{q}^\top_i + b_u + b_i \tag{15.3.2}$$

Then, we train the matrix factorization model by minimizing the mean squared error between predicted rating scores and real rating scores. The objective function is defined as follows:

$$\underset{\mathbf{P}, \mathbf{Q}, b}{\mathrm{argmin}} \sum_{(u, i) \in \mathcal{K}} \| \mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui} \|^2 + \lambda (\| \mathbf{P} \|^2_F + \| \mathbf{Q} \|^2_F + b_u^2 + b_i^2) \tag{15.3.3}$$

where λ denotes the regularization rate. The regularizing term $\lambda (\| \mathbf{P} \|^2_F + \| \mathbf{Q} \|^2_F + b_u^2 + b_i^2)$ is used to avoid over-fitting by penalizing the magnitude of the parameters. The (u, i) pairs for which $\mathbf{R}_{ui}$ is known are stored in the set $\mathcal{K}=\{(u, i) \mid \mathbf{R}_{ui} \text{ is known}\}$. The model parameters can be learned with an optimization algorithm, such as Stochastic Gradient Descent and Adam.

An intuitive illustration of the matrix factorization model is shown below:

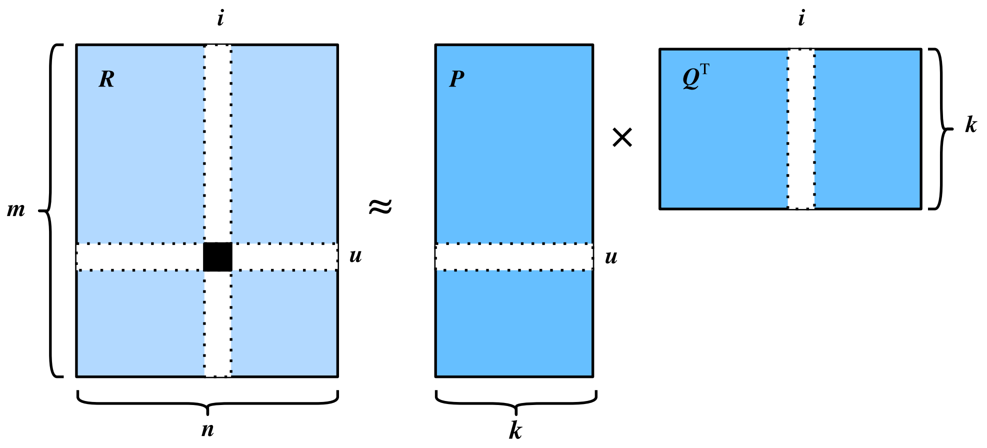


Fig. 15.3.1: Illustration of matrix factorization model

In the rest of this section, we will explain the implementation of matrix factorization and train the model on the MovieLens dataset.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
npx.set_np()
```

15.3.2 Model Implementation

First, we implement the matrix factorization model described above. The user and item latent factors can be created with the `nn.Embedding`. The `input_dim` is the number of items/users and the (`output_dim`) is the dimension of the latent factors (k). We can also use `nn.Embedding` to create the user/item biases by setting the `output_dim` to one. In the `forward` function, user and item ids are used to look up the embeddings.

```
class MF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, **kwargs):
        super(MF, self).__init__(**kwargs)
        self.P = nn.Embedding(input_dim=num_users, output_dim=num_factors)
        self.Q = nn.Embedding(input_dim=num_items, output_dim=num_factors)
        self.user_bias = nn.Embedding(num_users, 1)
        self.item_bias = nn.Embedding(num_items, 1)

    def forward(self, user_id, item_id):
        P_u = self.P(user_id)
        Q_i = self.Q(item_id)
        b_u = self.user_bias(user_id)
        b_i = self.item_bias(item_id)
        outputs = (P_u * Q_i).sum(axis=1) + np.squeeze(b_u) + np.squeeze(b_i)
        return outputs.flatten()
```

15.3.3 Evaluation Measures

We then implement the RMSE (root-mean-square error) measure, which is commonly used to measure the differences between rating scores predicted by the model and the actually observed ratings (ground truth) (Gunawardana & Shani, 2015). RMSE is defined as:

$$\mathrm{RMSE} = \sqrt{\frac{1}{|\mathcal{T}|}\sum_{(u,i)\in\mathcal{T}}(\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui})^2} \tag{15.3.4}$$

where $\mathcal{T}$ is the set consisting of pairs of users and items that you want to evaluate on. $|\mathcal{T}|$ is the size of this set. We can use the RMSE function provided by `mx.metric`.

```
def evaluator(net, test_iter, ctx):
    rmse = mx.metric.RMSE()  # Get the RMSE
    rmse_list = []
    for idx, (users, items, ratings) in enumerate(test_iter):
        u = gluon.utils.split_and_load(users, ctx, even_split=False)
```

(continues on next page)

(continued from previous page)

```
        i = gluon.utils.split_and_load(items, ctx, even_split=False)
        r_ui = gluon.utils.split_and_load(ratings, ctx, even_split=False)
        r_hat = [net(u, i) for u, i in zip(u, i)]
        rmse.update(labels=r_ui, preds=r_hat)
        rmse_list.append(rmse.get()[1])
    return float(np.mean(np.array(rmse_list)))
```

15.3.4 Training and Evaluating the Model

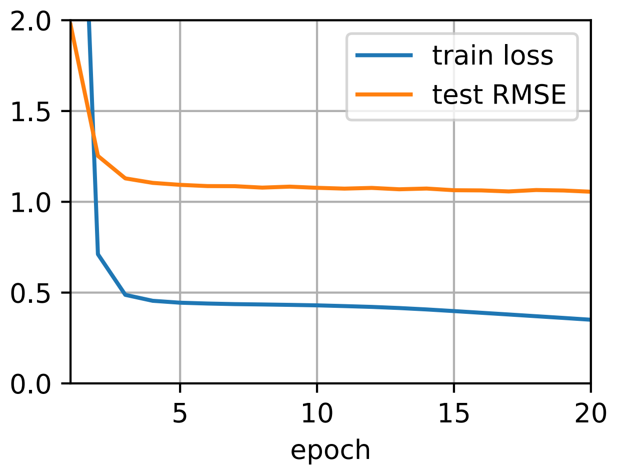
In the training function, we adopt the L_2 loss with weight decay. The weight decay mechanism has the same effect as the L_2 regularization.

```
# Saved in the d2l package for later use
def train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        ctx_list=d2l.try_all_gpus(), evaluator=None,
                        **kwargs):
    timer = d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 2],
                            legend=['train loss', 'test RMSE'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            timer.start()
            input_data = []
            values = values if isinstance(values, list) else [values]
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, ctx_list))
            train_feat = input_data[0:-1] if len(values) > 1 else input_data
            train_label = input_data[-1]
            with autograd.record():
                preds = [net(*t) for t in zip(*train_feat)]
                ls = [loss(p, s) for p, s in zip(preds, train_label)]
            [l.backward() for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean() / len(ctx_list)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
            timer.stop()
        if len(kwargs) > 0:  # it will be used in section AutoRec.
            test_rmse = evaluator(net, test_iter, kwargs['inter_mat'],
                                  ctx_list)
        else:
            test_rmse = evaluator(net, test_iter, ctx_list)
        train_l = l / (i + 1)
        animator.add(epoch + 1, (train_l, test_rmse))
    print('train loss %.3f, test RMSE %.3f'
          % (metric[0] / metric[1], test_rmse))
    print('%.1f examples/sec on %s'
          % (metric[2] * num_epochs / timer.sum(), ctx_list))
```

Finally, let's put all things together and train the model. Here, we set the latent factor dimension to 30.

```
ctx = d2l.try_all_gpus()
num_users, num_items, train_iter, test_iter = d2l.split_and_load_ml100k(
    test_ratio=0.1, batch_size=512)
net = MF(30, num_users, num_items)
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 20, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                    ctx, evaluator)
```

```
train loss 0.066, test RMSE 1.055
78035.0 examples/sec on [gpu(0), gpu(1)]
```



Below, we use the trained model to predict the rating that a user (ID 20) might give to an item (ID 30).

```
scores = net(np.array([20], dtype='int', ctx=d2l.try_gpu()),
             np.array([30], dtype='int', ctx=d2l.try_gpu()))
scores
```

```
array([3.1662967], ctx=gpu(0))
```

Summary

- The matrix factorization model is widely used in recommender systems. It can be used to predict ratings that a user might give to an item.
- We can implement and train matrix factorization for recommender systems.

Exercise

- Vary the size of latent factors. How does the size of latent factors influence the model performance?
- Try different optimizers, learning rates, and weight decay rates.
- Check the predicted rating scores of other users for a specific movie.

15.4 AutoRec: Rating Prediction with Autoencoders

Although the matrix factorization model achieves decent performance on the rating prediction task, it is essentially a linear model. Thus, such models are not capable of capturing complex nonlinear and intricate relationships that may be predictive of users' preferences. In this section, we introduce a nonlinear neural network collaborative filtering model, AutoRec (Sedhain et al., 2015). It identifies collaborative filtering (CF) with an autoencoder architecture and aims to integrate nonlinear transformations into CF on the basis of explicit feedback. Neural networks have been proven to be capable of approximating any continuous function, making it suitable to address the limitation of matrix factorization and enrich the expressiveness of matrix factorization.

On one hand, AutoRec has the same structure as an autoencoder which consists of an input layer, a hidden layer, and a reconstruction (output) layer. An autoencoder is a neural network that learns to copy its input to its output in order to code the inputs into the hidden (and usually low-dimensional) representations. In AutoRec, instead of explicitly embedding users/items into low-dimensional space, it uses the column/row of the interaction matrix as the input, then reconstructs the interaction matrix in the output layer.

On the other hand, AutoRec differs from a traditional autoencoder: rather than learning the hidden representations, AutoRec focuses on learning/reconstructing the output layer. It uses a partially observed interaction matrix as the input, aiming to reconstruct a completed rating matrix. In the meantime, the missing entries of the input are filled in the output layer via reconstruction for the purpose of recommendation.

There are two variants of AutoRec: user-based and item-based. For brevity, here we only introduce the item-based AutoRec. User-based AutoRec can be derived accordingly.

15.4.1 Model

Let $\mathbf{R}_{*i}$ denote the i^th column of the rating matrix, where unknown ratings are set to zeros by default. The neural architecture is defined as:

$$h(\mathbf{R}_{*i}) = f(\mathbf{W} \cdot g(\mathbf{V}\mathbf{R}_{*i} + \mu) + b) \tag{15.4.1}$$

where $f(\cdot)$ and $g(\cdot)$ represent activation functions, $\mathbf{W}$ and $\mathbf{V}$ are weight matrices, μ and b are biases. Let $h(\cdot)$ denote the whole network of AutoRec. The output $h(\mathbf{R}_{*i})$ is the reconstruction of the i^th column of the rating matrix.

The following objective function aims to minimize the reconstruction error:

$$\underset{\mathbf{W},\mathbf{V},\mu,b}{\mathrm{argmin}} \sum_{i=1}^{M} \| \mathbf{R}_{*i} - h(\mathbf{R}_{*i}) \|_{\mathcal{O}}^2 + \lambda(\|\mathbf{W}\|_F^2 + \|\mathbf{V}\|_F^2) \tag{15.4.2}$$

where $\| \cdot \|_{\mathcal{O}}$ means only the contribution of observed ratings are considered, that is, only weights that are associated with observed inputs are updated during back-propagation.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import sys
npx.set_np()
```

15.4.2 Implementing the Model

A typical autoencoder consists of an encoder and a decoder. The encoder projects the input to hidden representations and the decoder maps the hidden layer to the reconstruction layer. We follow this practice and create the encoder and decoder with dense layers. The activation of encoder is set to `sigmoid` by default and no activation is applied for decoder. Dropout is included after the encoding transformation to reduce over-fitting. The gradients of unobserved inputs are masked out to ensure that only observed ratings contribute to the model learning process.

```
class AutoRec(nn.Block):
    def __init__(self, num_hidden, num_users, dropout_rate=0.05):
        super(AutoRec, self).__init__()
        self.encoder = gluon.nn.Dense(num_hidden, activation='sigmoid',
                                      use_bias=True)
        self.decoder = gluon.nn.Dense(num_users, use_bias=True)
        self.dropout_layer = gluon.nn.Dropout(dropout_rate)

    def forward(self, input):
        hidden = self.dropout_layer(self.encoder(input))
        pred = self.decoder(hidden)
        if autograd.is_training():  # mask the gradient during training.
            return pred * np.sign(input)
        else:
            return pred
```

15.4.3 Reimplementing the Evaluator

Since the input and output have been changed, we need to reimplement the evaluation function, while we still use RMSE as the accuracy measure.

```
def evaluator(network, inter_matrix, test_data, ctx):
    scores = []
    for values in inter_matrix:
        feat = gluon.utils.split_and_load(values, ctx, even_split=False)
        scores.extend([network(i).asnumpy() for i in feat])
```

(continues on next page)

(continued from previous page)

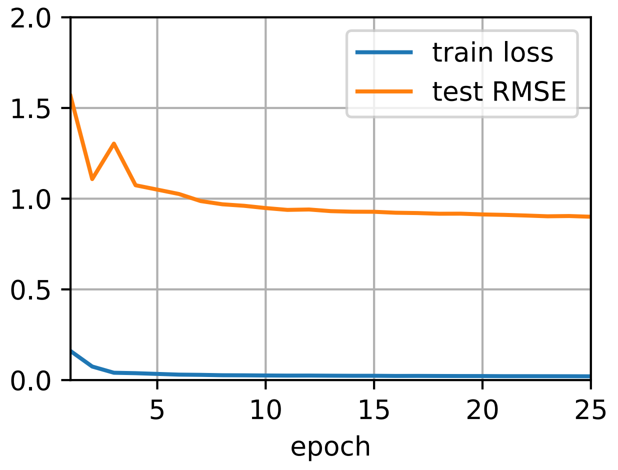
```
    recons = np.array([item for sublist in scores for item in sublist])
    # Calculate the test RMSE.
    rmse = np.sqrt(np.sum(np.square(test_data - np.sign(test_data) * recons))
                   / np.sum(np.sign(test_data)))
    return float(rmse)
```

15.4.4 Training and Evaluating the Model

Now, let's train and evaluate AutoRec on the MovieLens dataset. We can clearly see that the test RMSE is lower than the matrix factorization model, confirming the effectiveness of neural networks in the rating prediction task.

```
ctx = d2l.try_all_gpus()
# Load the MovieLens 100K dataset
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items)
_, _, _, train_inter_mat = d2l.load_data_ml100k(train_data, num_users,
                                                num_items)
_, _, _, test_inter_mat = d2l.load_data_ml100k(test_data, num_users,
                                               num_items)
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_inter_mat, shuffle=True,
                                   last_batch="rollover", batch_size=256,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(np.array(train_inter_mat), shuffle=False,
                                  last_batch="keep", batch_size=1024,
                                  num_workers=num_workers)
# Model initialization, training, and evaluation
net = AutoRec(500, num_users)
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 25, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
d2l.train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        ctx, evaluator, inter_mat=test_inter_mat)
```

```
train loss 0.000, test RMSE 0.900
45418222.5 examples/sec on [gpu(0), gpu(1)]
```



Summary

- We can frame the matrix factorization algorithm with autoencoders, while integrating non-linear layers and dropout regularization.
- Experiments on the MovieLens 100K dataset show that AutoRec achieves superior performance than matrix factorization.

Exercises

- Vary the hidden dimension of AutoRec to see its impact on the model performance.
- Try to add more hidden layers. Is it helpful to improve the model performance?
- Can you find a better combination of decoder and encoder activation functions?

15.5 Personalized Ranking for Recommender Systems

In the former sections, only explicit feedback was considered and models were trained and tested on observed ratings only. There are two demerits of such methods: First, most feedback is not explicit but implicit in real-world scenarios, and explicit feedback can be more expensive to collect. Second, non-observed user-item pairs which may be predictive for users' interests are totally ignored, making these methods unsuitable for cases where ratings are not missing at random but because of users' preferences. Non-observed user-item pairs are a mixture of real negative feedback (users are not interested in the items) and missing values (the user might interact with the items in the future). We simply ignore the non-observed pairs in matrix factorization and AutoRec. Clearly, these models are incapable of distinguishing between observed and non-observed pairs and are usually not suitable for personalized ranking tasks.

To this end, a class of recommendation models targeting at generating ranked recommendation lists from implicit feedback have gained popularity. In general, personalized ranking models can be optimized with pointwise, pairwise or Listwise approaches. Pointwise approaches considers a single interaction at a time and train a classifier/regressor to predict individual preferences. Matrix factorization and AutoRec are optimized with pointwise objectives. Pairwise approaches consider a pair of items for each user and aim to approximate the optimal ordering for that pair. Usually, pairwise approaches are more suitable for the ranking task because predicting relative order is reminiscent to the nature of ranking. Listwise approaches approximate the ordering of the entire list of items, for example, direct optimizing the ranking measures such as Normalized Discounted Cumulative Gain (NDCG[241]). However, listwise approaches are more complex and compute-intensive than pointwise or pairwise approaches. In this section, we will introduce two pairwise objectives/losses, Bayesian Personalized Ranking loss and Hinge loss, and their respective implementations.

15.5.1 Bayesian Personalized Ranking Loss and its Implementation

Bayesian personalized ranking (BPR) (Rendle et al., 2009) is a pairwise personalized ranking loss that is derived from the maximum posterior estimator. It has been widely used in many existing recommendation models. The training data of BPR consists of both positive and negative pairs (missing values). It assumes that the user prefers the positive item over all other non-observed items.

In formal, the training data is constructed by tuples in the form of (u, i, j), which represents that the user u prefers the item i over the item j. The Bayesian formulation of BPR which aims to maximize the posterior probability is given below:

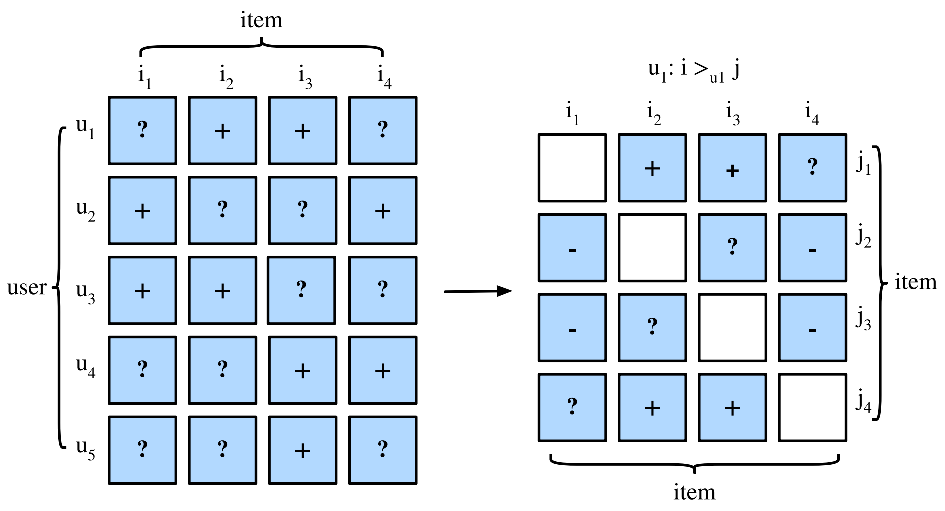
$$p(\Theta \mid >_u) \propto p(>_u \mid \Theta) p(\Theta) \tag{15.5.1}$$

Where Θ represents the parameters of an arbitrary recommendation model, $>_u$ represents the desired personalized total ranking of all items for user u. We can formulate the maximum posterior estimator to derive the generic optimization criterion for the personalized ranking task.

$$\begin{aligned}
\text{BPR-OPT} : &= \ln p(\Theta \mid >_u) \\
&= \ln p(>_u \mid \Theta) p(\Theta) \\
&= \ln \prod_{(u,i,j \in D)} \sigma(\hat{y}_{ui} - \hat{y}_{uj}) p(\Theta) \\
&= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \ln p(\Theta) \\
&= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) - \lambda_\Theta \|\Theta\|^2
\end{aligned} \tag{15.5.2}$$

where $D := \{(u, i, j) \mid i \in I^+_u \wedge j \in I \backslash I^+_u \}$ is the training set, with I^+_u denoting the items the user u liked, I denoting all items, and $I \backslash I^+_u$ indicating all other items excluding items the user liked. $\hat{y}_{ui}$ and $\hat{y}_{uj}$ are the predicted scores of the user u to item i and j, respectively. The prior $p(\Theta)$ is a normal distribution with zero mean and variance-covariance matrix Σ_Θ. Here, we let $\Sigma_\Theta = \lambda_\Theta I$.

[241] https://en.wikipedia.org/wiki/Discounted_cumulative_gain



We will implement the base class `mxnet.gluon.loss.Loss` and override the `forward` method to construct the Bayesian personalized ranking loss. We begin by importing the Loss class and the np module.

```
from mxnet import gluon, np, npx
npx.set_np()
```

The implementation of BPR loss is as follows.

```
# Saved in the d2l package for later use
class BPRLoss(gluon.loss.Loss):
    def __init__(self, weight=None, batch_axis=0, **kwargs):
        super(BPRLoss, self).__init__(weight=None, batch_axis=0, **kwargs)

    def forward(self, positive, negative):
        distances = positive - negative
        loss = - np.sum(np.log(npx.sigmoid(distances)), 0, keepdims=True)
        return loss
```

15.5.2 Hinge Loss and its Implementation

The Hinge loss for ranking has different form to the hinge loss[242] provided within the gluon library that is often used in classifiers such as SVMs. The loss used for ranking in recommender systems has the following form.

$$\sum_{(u,i,j\in D)} (\max(m - \hat{y}_{ui} + \hat{y}_{uj}), 0) \tag{15.5.3}$$

where m is the safety margin size. It aims to push negative items away from positive items. Similar to BPR, it aims to optimize for relevant distance between positive and negative samples instead of absolute outputs, making it well suited to recommender systems.

```
# Saved in the d2l package for later use
class HingeLossbRec(gluon.loss.Loss):
```

(continues on next page)

[242] https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.HingeLoss

(continued from previous page)

```
    def __init__(self, weight=None, batch_axis=0, **kwargs):
        super(HingeLossbRec, self).__init__(weight=None, batch_axis=0,
                                            **kwargs)

    def forward(self, positive, negative, margin=1):
        distances = positive - negative
        loss = np.sum(np.maximum(- distances + margin, 0))
        return loss
```

These two losses are interchangeable for personalized ranking in recommendation.

Summary

- There are three types of ranking losses available for the personalized ranking task in recommender systems, namely, pointwise, pairwise and listwise methods.
- The two pairwise loses, Bayesian personalized ranking loss and hinge loss, can be used interchangeably.

Exercises

- Are there any variants of BPR and hinge loss available?
- Can you find any recommendation models that use BPR or hinge loss?

15.6 Neural Collaborative Filtering for Personalized Ranking

This section moves beyond explicit feedback, introducing the neural collaborative filtering (NCF) framework for recommendation with implicit feedback. Implicit feedback is pervasive in recommender systems. Actions such as Clicks, buys, and watches are common implicit feedback which are easy to collect and indicative of users' preferences. The model we will introduce, titled NeuMF (He et al., 2017b), short for neural matrix factorization, aims to address the personalized ranking task with implicit feedback. This model leverages the flexibility and non-linearity of neural networks to replace dot products of matrix factorization, aiming at enhancing the model expressiveness. In specific, this model is structured with two subnetworks including generalized matrix factorization (GMF) and multilayer perceptron (MLP) and models the interactions from two pathways instead of simple inner products. The outputs of these two networks are concatenated for the final prediction scores calculation. Unlike the rating prediction task in AutoRec, this model generates a ranked recommendation list to each user based on the implicit feedback. We will use the personalized ranking loss introduced in the last section to train this model.

15.6.1 The NeuMF model

As aforementioned, NeuMF fuses two subnetworks. The GMF is a generic neural network version of matrix factorization where the input is the elementwise product of user and item latent factors. It consists of two neural layers:

$$\begin{aligned}\mathbf{x} &= \mathbf{p}_u \odot \mathbf{q}_i \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \mathbf{x}),\end{aligned} \tag{15.6.1}$$

where $\odot$ denotes the Hadamard product of vectors. $\mathbf{P} \in \mathbb{R}^{m \times k}$ and $\mathbf{Q} \in \mathbb{R}^{n \times k}$ corespond to user and item latent matrix respectively. $\mathbf{p}_u \in \mathbb{R}^k$ is the u^th row of P and $\mathbf{q}_i \in \mathbb{R}^k$ is the i^th row of Q. α and h denote the activation function and weight of the output layer. $\hat{y}_{ui}$ is the prediction score of the user u might give to the item i.

Another component of this model is MLP. To enrich model flexibility, the MLP subnetwork does not share user and item embeddings with GMF. It uses the concatenation of user and item embeddings as input. With the complicated connections and nonlinear transformations, it is capable of eastimating the intricate interactions between users and items. More precisely, the MLP subnetwork is defined as:

$$\begin{aligned} z^{(1)} &= \phi_1(\mathbf{U}_u, \mathbf{V}_i) = [\mathbf{U}_u, \mathbf{V}_i] \\ \phi^{(2)}(z^{(1)}) &= \alpha^1(\mathbf{W}^{(2)} z^{(1)} + b^{(2)}) \\ &\ldots \\ \phi^{(L)}(z^{(L-1)}) &= \alpha^L(\mathbf{W}^{(L)} z^{(L-1)} + b^{(L)})) \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \phi^L(z^{(L)}))\end{aligned} \tag{15.6.2}$$

where $\mathbf{W}^*, \mathbf{b}^*$ and α^* denote the weight matrix, bias vector, and activation function. ϕ^* denotes the function of the corresponding layer. $\mathbf{z}^*$ denotes the output of corresponding layer.

To fuse the results of GMF and MLP, instead of simple addition, NeuMF concatenates the second last layers of two subnetworks to create a feature vector which can be passed to the further layers. Afterwards, the ouputs are projected with matrix $\mathbf{h}$ and a sigmoid activation function. The prediction layer is formulated as:

$$\hat{y}_{ui} = \sigma(\mathbf{h}^\top [\mathbf{x}, \phi^L(z^{(L)})]). \tag{15.6.3}$$

The following figure illustrates the model architecture of NeuMF.

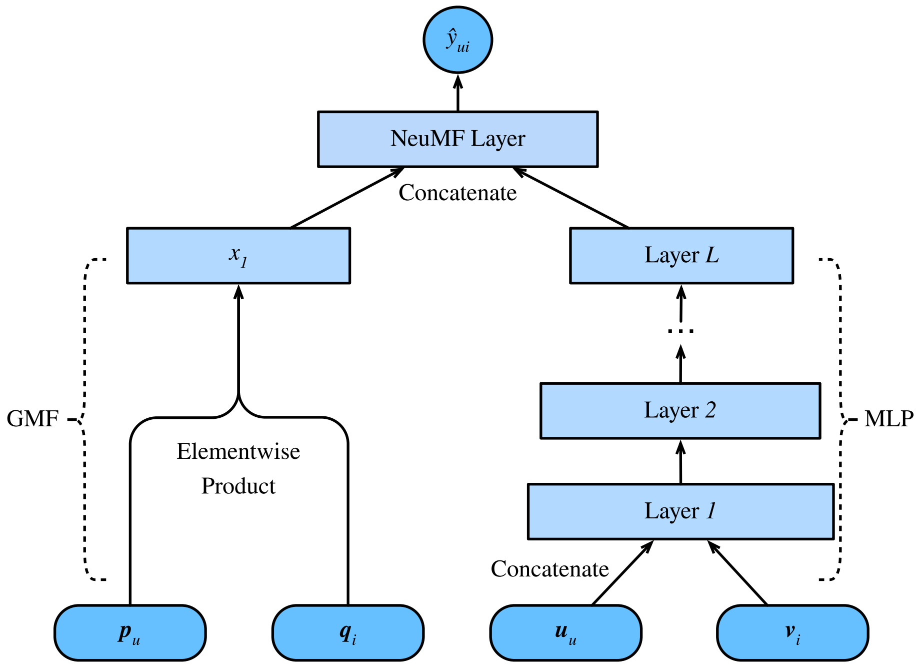


Fig. 15.6.1: Illustration of the NeuMF model

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import random
import sys
npx.set_np()
```

15.6.2 Model Implementation

The following code implements the NeuMF model. It consists of a generalized matrix factorization model and a multi-layered perceptron with different user and item embedding vectors. The structure of the MLP is controlled with the parameter `mlp_layers`. ReLU is used as the default activation function.

```
class NeuMF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, mlp_layers,
                 **kwargs):
        super(NeuMF, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.U = nn.Embedding(num_users, num_factors)
        self.V = nn.Embedding(num_items, num_factors)
        self.mlp = nn.Sequential()  # The MLP layers
        for i in mlp_layers:
            self.mlp.add(gluon.nn.Dense(i, activation='relu', use_bias=True))

    def forward(self, user_id, item_id):
        p_mf = self.P(user_id)
```

(continues on next page)

(continued from previous page)

```
        q_mf = self.Q(item_id)
        gmf = p_mf * q_mf
        p_mlp = self.U(user_id)
        q_mlp = self.V(item_id)
        mlp = self.mlp(np.concatenate([p_mlp, q_mlp], axis=1))
        con_res = np.concatenate([gmf, mlp], axis=1)
        return np.sum(con_res, axis=-1)
```

15.6.3 Customized Dataset with Negative Sampling

For pairwise ranking loss, an important step is negative sampling. For each user, the items that a user has not interacted with are candidate items (unobserved entries). The following function takes users identity and candidate items as input, and samples negative items randomly for each user from the candidate set of that user. During the training stage, the model ensures that the items that a user likes to be ranked higher than items she dislikes or has not interacted with.

```
class PRDataset(gluon.data.Dataset):
    def __init__(self, users, items, candidates, num_items):
        self.users = users
        self.items = items
        self.cand = candidates
        self.all = set([i for i in range(num_items)])

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

    def __getitem__(self, idx):
        neg_items = list(self.all - set(self.cand[int(self.users[idx])]))
        indices = random.randint(0, len(neg_items) - 1)
        return self.users[idx], self.items[idx], neg_items[indices]
```

15.6.4 Evaluator

In this section, we adopt the splitting by time strategy to construct the training and test sets. Two evaluation measures including hit rate at given cutting off ℓ (Hit@ℓ) and area under the ROC curve (AUC) are used to assess the model effectiveness. Hit rate at given position ℓ for each user indicates that whether the recommended item is included in the top ℓ ranked list. The formal definition is as follows:

$$\text{Hit@}\ell = \frac{1}{m} \sum_{u \in \mathcal{U}} \textbf{1}(rank_{u, g_u} <= \ell), \tag{15.6.4}$$

where $\textbf{1}$ denotes an indicator function that is equal to one if the ground truth item is ranked in the top ℓ list, otherwise it is equal to zero. $rank_{u, g_u}$ denotes the ranking of the ground truth item g_u of the user u in the recommendation list (The ideal ranking is 1). m is the number of users. $\mathcal{U}$ is the user set.

The definition of AUC is as follows:

$$\text{AUC} = \frac{1}{m} \sum_{u \in \mathcal{U}} \frac{1}{|\mathcal{I} \backslash S_u|} \sum_{j \in I \backslash S_u} \textbf{1}(rank_{u, g_u} < rank_{u, j}), \tag{15.6.5}$$

where $\mathcal{I}$ is the item set. S_u is the candidate items of user u. Note that many other evaluation protocols such as precision, recall and normalized discounted cumulative gain (NDCG) can also be used.

The following function calculates the hit counts and AUC for each user.

```python
# Saved in the d2l package for later use
def hit_and_auc(rankedlist, test_matrix, k):
    hits_k = [(idx, val) for idx, val in enumerate(rankedlist[:k])
              if val in set(test_matrix)]
    hits_all = [(idx, val) for idx, val in enumerate(rankedlist)
                if val in set(test_matrix)]
    max = len(rankedlist) - 1
    auc = 1.0 * (max - hits_all[0][0]) / max if len(hits_all) > 0 else 0
    return len(hits_k), auc
```

Then, the overall Hit rate and AUC are calculated as follows.

```python
# Saved in the d2l package for later use
def evaluate_ranking(net, test_input, seq, candidates, num_users, num_items,
                     ctx):
    ranked_list, ranked_items, hit_rate, auc = {}, {}, [], []
    all_items = set([i for i in range(num_users)])
    for u in range(num_users):
        neg_items = list(all_items - set(candidates[int(u)]))
        user_ids, item_ids, x, scores = [], [], [], []
        [item_ids.append(i) for i in neg_items]
        [user_ids.append(u) for _ in neg_items]
        x.extend([np.array(user_ids)])
        if seq is not None:
            x.append(seq[user_ids, :])
        x.extend([np.array(item_ids)])
        test_data_iter = gluon.data.DataLoader(gluon.data.ArrayDataset(*x),
                                               shuffle=False,
                                               last_batch="keep",
                                               batch_size=1024)
        for index, values in enumerate(test_data_iter):
            x = [gluon.utils.split_and_load(v, ctx, even_split=False)
                 for v in values]
            scores.extend([list(net(*t).asnumpy()) for t in zip(*x)])
        scores = [item for sublist in scores for item in sublist]
        item_scores = list(zip(item_ids, scores))
        ranked_list[u] = sorted(item_scores, key=lambda t: t[1], reverse=True)
        ranked_items[u] = [r[0] for r in ranked_list[u]]
        temp = hit_and_auc(ranked_items[u], test_input[u], 50)
        hit_rate.append(temp[0])
        auc.append(temp[1])
    return np.mean(np.array(hit_rate)), np.mean(np.array(auc))
```

15.6.5 Training and Evaluating the Model

The training function is defined below. We train the model in the pairwise manner.

```
# Saved in the d2l package for later use
def train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, ctx_list, evaluator,
                  candidates, eval_step=1):
    timer, hit_rate, auc = d2l.Timer(), 0, 0
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['test hit rate', 'test AUC'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            input_data = []
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, ctx_list))
            with autograd.record():
                p_pos = [net(*t) for t in zip(*input_data[0:-1])]
                p_neg = [net(*t) for t in zip(*input_data[0:-2],
                                              input_data[-1])]
                ls = [loss(p, n) for p, n in zip(p_pos, p_neg)]
            [l.backward(retain_graph=False) for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean()/len(ctx_list)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
            timer.stop()
        with autograd.predict_mode():
            if (epoch + 1) % eval_step == 0:
                hit_rate, auc = evaluator(net, test_iter, test_seq_iter,
                                          candidates, num_users, num_items,
                                          ctx_list)
                animator.add(epoch + 1, (hit_rate, auc))
    print('train loss %.3f, test hit rate %.3f, test AUC %.3f'
          % (metric[0] / metric[1], hit_rate, auc))
    print('%.1f examples/sec on %s'
          % (metric[2] * num_epochs / timer.sum(), ctx_list))
```

Now, we can load the MovieLens 100k dataset and train the model. Since there are only ratings in the MovieLens dataset, with some losses of accuracy, we binarize these ratings to zeros and ones. If a user rated an item, we consider the implicit feedback as one, otherwise as zero. The action of rating an item can be treated as a form of providing implicit feedback. Here, we split the dataset in the seq-aware mode where users' latest interacted items are left out for test.

```
batch_size = 1024
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                              'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(PRDataset(users_train, items_train,
                                             candidates, num_items ),
```

(continues on next page)

(continued from previous page)

```
                                      batch_size, True,
                                      last_batch="rollover",
                                      num_workers=num_workers)
```

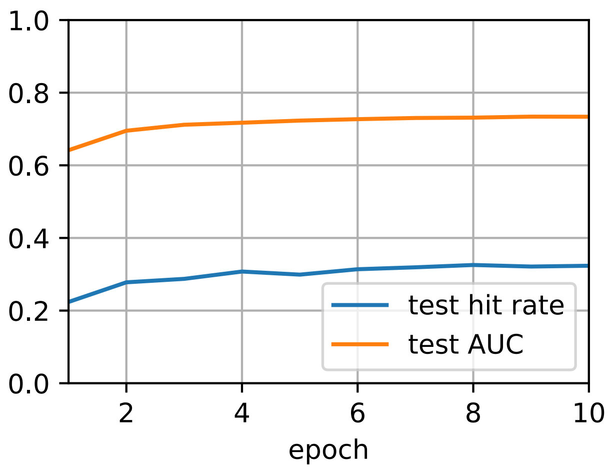
We then create and initialize the model. we use a three-layer MLP with constant hidden size 10.

```
ctx = d2l.try_all_gpus()
net = NeuMF(10, num_users, num_items, mlp_layers=[10, 10, 10])
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
```

The following code trains the model.

```
lr, num_epochs, wd, optimizer = 0.01, 10, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_ranking(net, train_iter, test_iter, loss, trainer, None, num_users,
              num_items, num_epochs, ctx, evaluate_ranking, candidates)
```

```
train loss 4.381, test hit rate 0.323, test AUC 0.734
15.6 examples/sec on [gpu(0), gpu(1)]
```



Summary

- Adding nonlinearity to matrix factorization model is beneficial for improving the model capability and effectiveness.
- NeuMF is a combination of matrix factorization and Multilayer perceptron. The multilayer perceptron takes the concatenation of user and item embeddings as the input.

Exercises

- Vary the size of latent factors. How the size of latent factors impact the model performance?
- Vary the architectures (e.g., number of layers, number of neurons of each layer) of the MLP to check the its impact on the performance.
- Try different optimizers, learning rate and weight decay rate.
- Try to use hinge loss defined in the last section to optimize this model.

15.7 Sequence-Aware Recommender Systems

In previous sections, we abstract the recommendation task as a matrix completion problem without considering users' short-term behaviors. In this section, we will introduce a recommendation model that takes the sequentially-ordered user interaction logs into account. It is a sequence-aware recommender (Quadrana et al., 2018) where the input is an ordered and often timestamped list of past user actions. A number of recent literatures have demonstrated the usefulness of incorporating such information in modeling users' temporal behavioral patterns and discovering their interest drift.

The model we will introduce, Caser (Sedhain et al., 2015), short for convolutional sequence embedding recommendation model, adopts convolutional neural networks capture the dynamic pattern influences of users' recent activities. The main component of Caser consists of a horizontal convolutional network and a vertical convolutional network, aiming to uncover the union-level and point-level sequence patterns, respectively. Point-level pattern indicates the impact of single item in the historical sequence on the target item, while union level pattern implies the influences of several previous actions on the subsequent target. For example, buying both milk and butter together leads to higher probability of buying flour than just buying one of them. Moreover, users' general interests, or long term preferences are also modeled in the last fully-connected layers, resulting in a more comprehensive modeling of user interests. Details of the model are described as follows.

15.7.1 Model Architectures

In sequence-aware recommendation system, each user is associated with a sequence of some items from the item set. Let $S^u = (S_1^u, ...S_{|S_u|}^u)$ denotes the ordered sequence. The goal of Caser is to recommend item by considering user general tastes as well as short-term intention. Suppose we take the previous L items into consideration, an embedding matrix that represents the former interactions for timestep t can be constructed:

$$\mathbf{E}^{(u,t)} = [\mathbf{q}_{S_{t-L}^u}, ..., \mathbf{q}_{S_{t-2}^u}, \mathbf{q}_{S_{t-1}^u}]^\top, \tag{15.7.1}$$

where $\mathbf{Q} \in \mathbb{R}^{n\times k}$ represents item embeddings and $\mathbf{q}_i$ denotes the i^{th} row. $\mathbf{E}^{(u,t)} \in \mathbb{R}^{L\times k}$ can be used to infer the transient interest of user u at time-step t. We can view the input matrix $\mathbf{E}^{(u,t)}$ as an image which is the input of the subsequent two convolutional components.

The horizontal convolutional layer has d horizontal filters $\mathbf{F}^j \in \mathbb{R}^{h\times k}, 1 \leq j \leq d, h = \{1, ..., L\}$, and the vertical convolutional layer has d' vertical filters $\mathbf{G}^j \in \mathbb{R}^{L\times 1}, 1 \leq j \leq d'$. After a series of convolutional and pool operations, we get the two outputs:

$$\begin{aligned} \mathbf{o} &= \text{HConv}(\mathbf{E}^{(u,t)}, \mathbf{F}) \\ \mathbf{o}' &= \text{VConv}(\mathbf{E}^{(u,t)}, \mathbf{G}), \end{aligned} \tag{15.7.2}$$

where $\mathbf{o} \in \mathbb{R}^d$ is the output of horizontal convolutional network and $\mathbf{o}' \in \mathbb{R}^{kd'}$ is the output of vertical convolutional network. For simplicity, we omit the details of convolution and pool operations. They are concatenated and fed into a fully-connected neural network layer to get more high-level representations.

$$\mathbf{z} = \phi(\mathbf{W}[\mathbf{o}, \mathbf{o}']^\top + \mathbf{b}), \tag{15.7.3}$$

where $\mathbf{W} \in \mathbb{R}^{k\times(d+kd')}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^k$ is the bias. The learned vector $\mathbf{z} \in \mathbb{R}^k$ is the representation of user's short-term intent.

At last, the prediction function combines users' short-term and general taste together, which is defined as:

$$\hat{y}_{uit} = \mathbf{v}_i \cdot [\mathbf{z}, \mathbf{p}_u]^\top + \mathbf{b}'_i, \tag{15.7.4}$$

where $\mathbf{V} \in \mathbb{R}^{n\times 2k}$ is another item embedding matrix. $\mathbf{b}' \in \mathbb{R}^n$ is the item specific bias. $\mathbf{P} \in \mathbb{R}^{m\times k}$ is the user embedding matrix for users' general tastes. $\mathbf{p}_u \in \mathbb{R}^k$ is the u^th row of P and $\mathbf{v}_i \in \mathbb{R}^{2k}$ is the i^th row of $\mathbf{V}$.

The model can be learned with BPR or Hinge loss. The architecture of Caser is shown below:

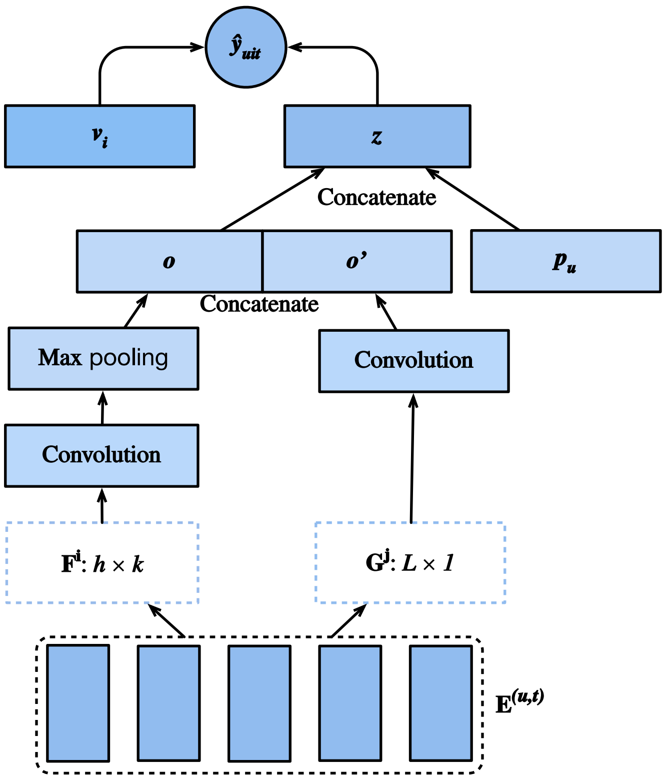


Fig. 15.7.1: Illustration of the Caser Model

We first import the required libraries.

```python
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import random
import sys
npx.set_np()
```

15.7.2 Model Implementation

The following code implements the Caser model. It consists of a vertical convolutional layer, a horizontal convolutional layer, and a full-connected layer.

```python
class Caser(nn.Block):
    def __init__(self, num_factors, num_users, num_items, L=5, d=16,
                 d_prime=4, drop_ratio=0.05, **kwargs):
        super(Caser, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.d_prime, self.d = d_prime, d
        # Vertical convolution layer
        self.conv_v = nn.Conv2D(d_prime, (L, 1), in_channels=1)
        # Horizontal convolution layer
        h = [i + 1 for i in range(L)]
        self.conv_h, self.max_pool = nn.Sequential(), nn.Sequential()
        for i in h:
            self.conv_h.add(nn.Conv2D(d, (i, num_factors), in_channels=1))
            self.max_pool.add(nn.MaxPool1D(L - i + 1))
        # Fully-connected layer
        self.fc1_dim_v, self.fc1_dim_h = d_prime * num_factors, d * len(h)
        self.fc = nn.Dense(in_units=d_prime * num_factors + d * L,
                           activation='relu', units=num_factors)
        self.Q_prime = nn.Embedding(num_items, num_factors * 2)
        self.b = nn.Embedding(num_items, 1)
        self.dropout = nn.Dropout(drop_ratio)

    def forward(self, user_id, seq, item_id):
        item_embs = np.expand_dims(self.Q(seq), 1)
        user_emb = self.P(user_id)
        out, out_h, out_v, out_hs = None, None, None, []
        if self.d_prime:
            out_v = self.conv_v(item_embs)
            out_v = out_v.reshape(out_v.shape[0], self.fc1_dim_v)
        if self.d:
            for conv, maxp in zip(self.conv_h, self.max_pool):
                conv_out = np.squeeze(npx.relu(conv(item_embs)), axis=3)
                t = maxp(conv_out)
                pool_out = np.squeeze(t, axis=2)
                out_hs.append(pool_out)
            out_h = np.concatenate(out_hs, axis=1)
        out = np.concatenate([out_v, out_h], axis=1)
        z = self.fc(self.dropout(out))
        x = np.concatenate([z, user_emb], axis=1)
```

(continues on next page)

(continued from previous page)

```
        q_prime_i = np.squeeze(self.Q_prime(item_id))
        b = np.squeeze(self.b(item_id))
        res = (x * q_prime_i).sum(1) + b
        return res
```

15.7.3 Sequential Dataset with Negative Sampling

To process the sequential interaction data, we need to reimplement the Dataset class. The following code creates a new dataset class named SeqDataset. In each sample, it outputs the user identity, her previous L interacted items as a sequence and the next item she interacts as the target. The following figure demonstrates the data loading process for one user. Suppose that this user liked 8 movies, we organize these eight movies in chronological order. The latest movie is left out as the test item. For the remaining seven movies, we can get three training samples, with each sample containing a sequence of five ($L = 5$) movies and its subsequent item as the target item. Negative samples are also included in the Customized dataset.

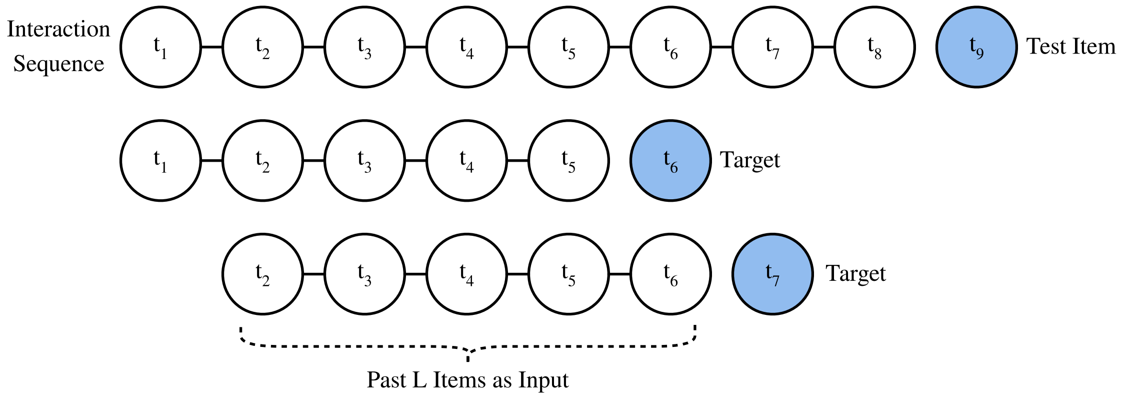


Fig. 15.7.2: Illustration of the data generation process

```
class SeqDataset(gluon.data.Dataset):
    def __init__(self, user_ids, item_ids, L, num_users, num_items,
                 candidates):
        user_ids, item_ids = np.array(user_ids), np.array(item_ids)
        sort_idx = np.array(sorted(range(len(user_ids)),
                                   key=lambda k: user_ids[k]))
        u_ids, i_ids = user_ids[sort_idx], item_ids[sort_idx]
        temp, u_ids, self.cand = {}, u_ids.asnumpy(), candidates
        self.all_items = set([i for i in range(num_items)])
        [temp.setdefault(u_ids[i], []).append(i) for i, _ in enumerate(u_ids)]
        temp = sorted(temp.items(), key=lambda x: x[0])
        u_ids = np.array([i[0] for i in temp])
        idx = np.array([i[1][0] for i in temp])
        self.ns = ns = int(sum([c - L if c >= L + 1 else 1 for c
                                in np.array([len(i[1]) for i in temp])]))
        self.seq_items = np.zeros((ns, L))
        self.seq_users = np.zeros(ns, dtype='int32')
        self.seq_tgt = np.zeros((ns, 1))
        self.test_seq = np.zeros((num_users, L))
```

(continues on next page)

(continued from previous page)

```
        test_users, _uid = np.empty(num_users), None
        for i, (uid, i_seq) in enumerate(self._seq(u_ids, i_ids, idx, L + 1)):
            if uid != _uid:
                self.test_seq[uid][:] = i_seq[-L:]
                test_users[uid], _uid = uid, uid
            self.seq_tgt[i][:] = i_seq[-1:]
            self.seq_items[i][:], self.seq_users[i] = i_seq[:L], uid

    def _win(self, tensor, window_size, step_size=1):
        if len(tensor) - window_size >= 0:
            for i in range(len(tensor), 0, - step_size):
                if i - window_size >= 0:
                    yield tensor[i - window_size:i]
                else:
                    break
        else:
            yield tensor

    def _seq(self, u_ids, i_ids, idx, max_len):
        for i in range(len(idx)):
            stop_idx = None if i >= len(idx) - 1 else int(idx[i + 1])
            for s in self._win(i_ids[int(idx[i]):stop_idx], max_len):
                yield (int(u_ids[i]), s)

    def __len__(self):
        return self.ns

    def __getitem__(self, i):
        neg = list(self.all_items - set(self.cand[int(self.seq_users[i])]))
        idx = random.randint(0, len(neg) - 1)
        return self.seq_users[i], self.seq_items[i], self.seq_tgt[i], neg[idx]
```

15.7.4 Load the MovieLens 100K dataset

Afterwards, we read and split the MovieLens 100K dataset in sequence-aware mode and load the training data with sequential dataloader implemented above.

```
TARGET_NUM, L, batch_size = 1, 3, 4096
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                              'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
train_seq_data = SeqDataset(users_train, items_train, L, num_users,
                            num_items, candidates)
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_seq_data, batch_size, True,
                                   last_batch="rollover",
                                   num_workers=num_workers)
test_seq_iter = train_seq_data.test_seq
train_seq_data[0]
```

```
(array(0, dtype=int32), array([110., 255.,   4.]), array([101.]), 657)
```

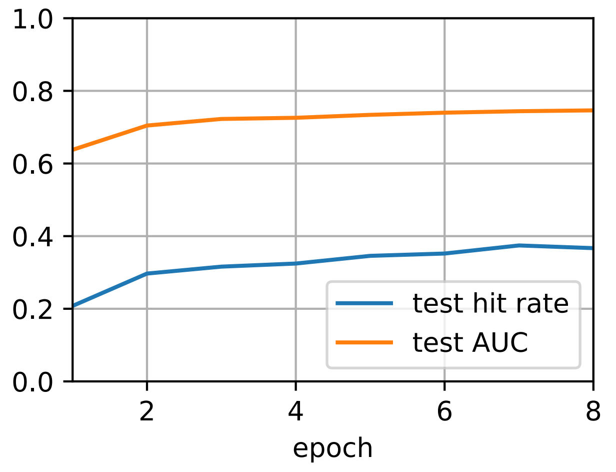
The training data structure is shown above. The first element is the user identity, the next list indicates the last five items this user liked, and the last element is the item this user liked after the five items.

15.7.5 Train the Model

Now, let's train the model. We use the same setting as NeuMF, including learning rate, optimizer, and k, in the last section so that the results are comparable.

```
ctx = d2l.try_all_gpus()
net = Caser(10, num_users, num_items, L)
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.04, 8, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})

d2l.train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, ctx, d2l.evaluate_ranking,
                  candidates, eval_step=1)
```

```
train loss 0.859, test hit rate 0.367, test AUC 0.746
35.0 examples/sec on [gpu(0), gpu(1)]
```



Summary

- Inferring a user's short-term and long-term interests can make prediction of the next item that she preferred more effectively.
- Convolutional neural networks can be utilized to capture users' short-term interests from sequential interactions.

Exercises

- Conduct an ablation study by removing one of the horizontal and vertical convolutional networks, which component is the more important ?
- Vary the hyper-parameter L. Does longer historical interactions bring higher accuracy?
- Apart from the sequence-aware recommendation task we introduced above, there is another type of sequence-aware recommendation task called session-based recommendation (Hidasi et al., 2015). Can you explain the differences between these two tasks?

15.8 Feature-Rich Recommender Systems

Interaction data is the most basic indication of users' preferences and interests. It plays a critical role in former introduced models. Yet, interaction data is usually extremely sparse and can be noisy at times. To address this issue, we can integrate side information such as features of items, profiles of users, and even in which context that the interaction occurred into the recommendation model. Utilizing these features are helpful in making recommendations in that these features can be an effective predictor of users interests especially when interaction data is lacking. As such, it is essential for recommendation models also have the capability to deal with those features and give the model some content/context awareness. To demonstrate this type of recommendation models, we introduce another task on click-through rate (CTR) for online advertisement recommendations (McMahan et al., 2013) and present an anonymous advertising data. Targeted advertisement services have attracted widespread attention and are often framed as recommendation engines. Recommending advertisements that match users' personal taste and interest is important for click-through rate improvement.

Digital marketers use online advertising to display advertisements to customers. Click-through rate is a metric that measures the number of clicks advertisers receive on their ads per number of impressions and it is expressed as a percentage calculated with the formula:

$$\text{CTR} = \frac{\#\text{Clicks}}{\#\text{Impressions}} \times 100\%. \tag{15.8.1}$$

Click-through rate is an important signal that indicates the effectiveness of prediction algorithms. Click-through rate prediction is a task of predicting the likelihood that something on a website will be clicked. Models on CTR prediction can not only be employed in targeted advertising systems but also in general item (e.g., movies, news, products) recommender systems, email campaigns, and even search engines. It is also closely related to user satisfaction, conversion rate, and can be helpful in setting campaign goals as it can help advertisers to set realistic expectations.

```
from collections import defaultdict
import d2l
from mxnet import gluon, np
```

15.8.1 An Online Advertising Dataset

With the considerable advancements of Internet and mobile technology, online advertising has become an important income resource and generates vast majority of revenue in the Internet industry. It is important to display relevant advertisements or advertisements that pique users' interests so that casual visitors can be converted into paying customers. The dataset we introduced is an online advertising dataset. It consists of 34 fields, with the first column representing the target variable that indicates if an ad was clicked (1) or not (0). All the other columns are categorical features. The columns might represent the advertisement id, site or application id, device id, time, user profiles and so on. The real semantics of the features are undisclosed due to anonymization and privacy concern.

The following code downloads the dataset from our server and saves it into the local data folder.

```
# Saved in the d2l package for later use
d2l.DATA_HUB['ctr'] = (d2l.DATA_URL+'ctr.zip',
                       'e18327c48c8e8e5c23da714dd614e390d369843f')

data_dir = d2l.download_extract('ctr')
```

```
Downloading ../data/ctr.zip from http://d2l-data.s3-accelerate.amazonaws.com/ctr.zip...
```

There are a training set and a test set, consisting of 15000 and 3000 samples/lines, respectively.

15.8.2 Dataset Wrapper

For the convenience of data loading, we implement a `CTRDataset` which loads the advertising dataset from the CSV file and can be used by `DataLoader`.

```
# Saved in the d2l package for later use
class CTRDataset(gluon.data.Dataset):
    def __init__(self, data_path, feat_mapper=None, defaults=None,
                 min_threshold=4, num_feat=34):
        self.NUM_FEATS, self.count, self.data = num_feat, 0, {}
        feat_cnts = defaultdict(lambda: defaultdict(int))
        self.feat_mapper, self.defaults = feat_mapper, defaults
        self.field_dims = np.zeros(self.NUM_FEATS, dtype=np.int64)
        with open(data_path) as f:
            for line in f:
                instance = {}
                values = line.rstrip('\n').split('\t')
                if len(values) != self.NUM_FEATS + 1:
                    continue
```

(continues on next page)

(continued from previous page)

```
                label = np.float32([0, 0])
                label[int(values[0])] = 1
                instance['y'] = [np.float32(values[0])]
                for i in range(1, self.NUM_FEATS + 1):
                    feat_cnts[i][values[i]] += 1
                    instance.setdefault('x', []).append(values[i])
                self.data[self.count] = instance
                self.count = self.count + 1
        if self.feat_mapper is None and self.defaults is None:
            feat_mapper = {i: {feat for feat, c in cnt.items() if c >=
                               min_threshold} for i, cnt in feat_cnts.items()}
            self.feat_mapper = {i: {feat: idx for idx, feat in enumerate(cnt)}
                                for i, cnt in feat_mapper.items()}
            self.defaults = {i: len(cnt) for i, cnt in feat_mapper.items()}
        for i, fm in self.feat_mapper.items():
            self.field_dims[i - 1] = len(fm) + 1
        self.offsets = np.array((0, *np.cumsum(self.field_dims).asnumpy()
                                 [:-1]))

    def __len__(self):
        return self.count

    def __getitem__(self, idx):
        feat = np.array([self.feat_mapper[i + 1].get(v, self.defaults[i + 1])
                         for i, v in enumerate(self.data[idx]['x'])])
        return feat + self.offsets, self.data[idx]['y']
```

The following example loads the training data and print out the first record.

```
train_data = CTRDataset(data_path=data_dir+"train.csv")
train_data[0]
```

```
(array([ 143.,  144.,  227.,  235.,  957., 1250., 1471., 1566., 1624.,
        1977., 2008., 2061., 2223., 2304., 2305., 2360., 2745., 2746.,
        2747., 2748., 2892., 2988., 3165., 3188., 3194., 3195., 3236.,
        3660., 3687., 3699., 3716., 3747., 3769., 3801.]), [1.0])
```

As can be seen, all the 34 fields are categorical features. Each value represents the one-hot index of the corresponding entry. The label 0 means that it is not clicked. This `CTRDataset` can also be used to load other datasets such as the Criteo display advertising challenge Dataset[246] and the Avazu click-through rate prediction Dataset[247].

[246] https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/
[247] https://www.kaggle.com/c/avazu-ctr-prediction

Summary

- Click-through rate is an important metric that is used to measure the effectiveness of advertising systems and recommender systems.
- Click-through rate prediction is usually converted to a binary classification problem. The target is to predict whether an ad/item will be clicked or not based on given features.

Exercise

- Can you load the Criteo and Avazu dataset with the provided `CTRDataset`. It is worth noting that the Criteo dataset consisting of real-valued features so you may have to revise the code a bit.

15.9 Factorization Machines

Factorization machines (FM) (Rendle, 2010), proposed by Steffen Rendle in 2010, is a supervised algorithm that can be used for classification, regression, and ranking tasks. It quickly took notice and became a popular and impactful method for making predictions and recommendations. Particularly, it is a generalization of the linear regression model and the matrix factorization model. Moreover, it is reminiscent of support vector machines with a polynomial kernel. The strengths of factorization machines over the linear regression and matrix factorization are: (1) it can model χ-way variable interactions, where χ is the number of polynomial order and is usually set to two. (2) A fast optimization algorithm associated with factorization machines can reduce the polynomial computation time to linear complexity, making it extremely efficient especially for high dimensional sparse inputs. For these reasons, factorization machines are widely employed in modern advertisement and products recommendations. The technical details and implementations are described below.

15.9.1 2-Way Factorization Machines

Formally, let $x \in \mathbb{R}^d$ denote the feature vectors of one sample, and y denote the corresponding label which can be real-valued label or class label such as binary class "click/non-click". The model for a factorization machine of degree two is defined as:

$$\hat{y}(x) = \mathbf{w}_0 + \sum_{i=1}^{d} \mathbf{w}_i x_i + \sum_{i=1}^{d}\sum_{j=i+1}^{d} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j \tag{15.9.1}$$

where $\mathbf{w}_0 \in \mathbb{R}$ is the global bias; $\mathbf{w} \in \mathbb{R}^d$ denotes the weights of the i-th variable; $\mathbf{V} \in \mathbb{R}^{d\times k}$ represents the feature embeddings; $\mathbf{v}_i$ represents the i^th row of $\mathbf{V}$; k is the dimensionality of latent factors; $\langle \cdot, \cdot \rangle$ is the dot product of two vectors. $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ model the interaction between the i^th and j^th feature. Some feature interactions can be easily understood so they can be designed by experts. However, most other feature interactions are hidden in data and difficult to identify. So modeling feature interactions automatically can greatly reduce the efforts in feature engineering. It is obvious that the first two terms correspond to the linear regression model and the last term is an extension of the matrix factorization model. If the feature i represents a item and the feature j represents a user, the third term is exactly the dot product between user and item embeddings. It is worth noting that FM can also generalize to higher orders (degree > 2). Nevertheless, the numerical stability might weaken the generalization.

15.9.2 An Efficient Optimization Criterion

Optimizing the factorization machines in a straight forward method leads to a complexity of $\mathcal{O}(kd^2)$ as all pairwise interactions require to be computed. To solve this inefficiency problem, we can reorganize the third term of FM which could greatly reduce the computation cost, leading to a linear time complexity ($\mathcal{O}(kd)$). The reformulation of the pairwise interaction term is as follows:

$$\begin{aligned}
&\sum_{i=1}^d \sum_{j=i+1}^d \langle\mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j \\
&= \frac{1}{2} \sum_{i=1}^d \sum_{j=1}^d\langle\mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j - \frac{1}{2}\sum_{i=1}^d \langle\mathbf{v}_i, \mathbf{v}_i\rangle x_i x_i \\
&= \frac{1}{2} \big (\sum_{i=1}^d \sum_{j=1}^d \sum_{l=1}^k\mathbf{v}_{i, l} \mathbf{v}_{j, l} x_i x_j - \sum_{i=1}^d \sum_{l=1}^k \mathbf{v}_{i, l} \mathbf{v}_{j, l} x_i x_i \big)\\
&= \frac{1}{2} \sum_{l=1}^k \big ((\sum_{i=1}^d \mathbf{v}_{i, l} x_i) (\sum_{j=1}^d \mathbf{v}_{j, l}x_j) - \sum_{i=1}^d \mathbf{v}_{i, l}^2 x_i^2 \big) \\
&= \frac{1}{2} \sum_{l=1}^k \big ((\sum_{i=1}^d \mathbf{v}_{i, l} x_i)^2 - \sum_{i=1}^d \mathbf{v}_{i, l}^2 x_i^2)
\end{aligned} \qquad (15.9.2)$$

With this reformulation, the model complexity are decreased greatly. Moreover, for sparse features, only non-zero elements needs to be computed so that the overall complexity is linear to the number of non-zero features.

To learn the FM model, we can use the MSE loss for regression task, the cross entropy loss for classification tasks, and the BPR loss for ranking task. Standard optimizers such as SGD and Adam are viable for optimization.

```
import d2l
from mxnet import init, gluon, np, npx
from mxnet.gluon import nn
import sys
npx.set_np()
```

15.9.3 Model Implementation

The following code implement the factorization machines. It is clear to see that FM consists a linear regression block and an efficient feature interaction block. We apply a sigmoid function over the final score since we treat the CTR prediction as a classification task.

```
class FM(nn.Block):
    def __init__(self, field_dims, num_factors):
        super(FM, self).__init__()
        input_size = int(sum(field_dims))
        self.embedding = nn.Embedding(input_size, num_factors)
        self.fc = nn.Embedding(input_size, 1)
        self.linear_layer = gluon.nn.Dense(1, use_bias=True)

    def forward(self, x):
        square_of_sum = np.sum(self.embedding(x), axis=1) ** 2
        sum_of_square = np.sum(self.embedding(x) ** 2, axis=1)
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True)
        x = npx.sigmoid(x)
        return x
```

15.9.4 Load the Advertising Dataset

We use the CTR data wrapper from the last section to load the online advertising dataset.

```
batch_size = 2048
data_dir = d2l.download_extract('ctr')
train_data = d2l.CTRDataset(data_dir+"train.csv")
test_data = d2l.CTRDataset(data_dir+"test.csv",
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_data, shuffle=True,
                                   last_batch="rollover",
                                   batch_size=batch_size,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(test_data, shuffle=False,
                                  last_batch="rollover",
                                  batch_size=batch_size,
                                  num_workers=num_workers)
```

15.9.5 Train the Model

Afterwards, we train the model. The learning rate is set to 0.01 and the embedding size is set to 20 by default. The `Adam` optimizer and the `SigmoidBinaryCrossEntropyLoss` loss are used for model training.

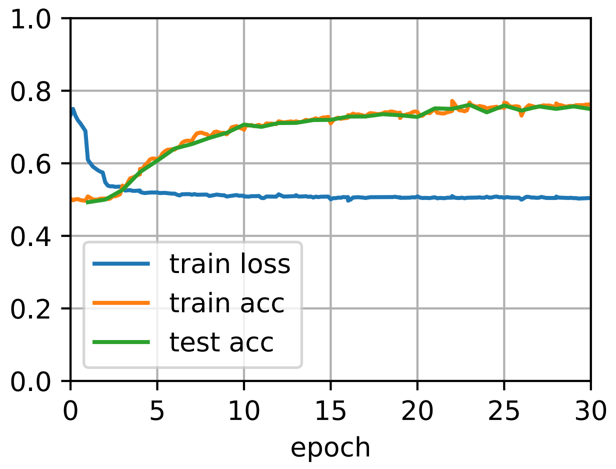
```
ctx = d2l.try_all_gpus()
net = FM(train_data.field_dims, num_factors=20)
net.initialize(init.Xavier(), ctx=ctx)
```

(continues on next page)

(continued from previous page)

```
lr, num_epochs, optimizer = 0.02, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.504, train acc 0.762, test acc 0.750
213360.0 exampes/sec on [gpu(0), gpu(1)]
```



Summary

- FM is a general framework that can be applied on a variety of tasks such as regression, classification, and ranking.
- Feature interaction/crossing is important for prediction tasks and the 2-way interaction can be efficiently modeled with FM.

Exercise

- Can you test FM on other dataset such as Avazu, MovieLens, and Criteo datasets?
- Vary the embedding size to check its impact on performance, can you observe a similar pattern as that of matrix factorization?

15.10 Deep Factorization Machines

Learning effective feature combinations is critical to the success of click-through rate prediction task. Factorization machines model feature interactions in a linear paradigm (e.g., bilinear interactions). This is often insufficient for real-world data where inherent feature crossing structures are usually very complex and nonlinear. What's worse, second-order feature interactions are generally used in factorization machines in practice. Modeling higher degrees of feature combinations with factorization machines is possible theoretically but it is usually not adopted due to numerical instability and high computational complexity.

One effective solution is using deep neural networks. Deep neural networks are powerful in feature representation learning and have the potential to learn sophisticated feature interactions. As such, it is natural to integrate deep neural networks to factorization machines. Adding nonlinear transformation layers to factorization machines gives it the capability to model both low-order feature combinations and high-order feature combinations. Moreover, non-linear inherent structures from inputs can also be captured with deep neural networks. In this section, we will introduce a representative model named deep factorization machines (DeepFM) (Guo et al., 2017) which combine FM and deep neural networks.

15.10.1 Model Architectures

DeepFM consists of an FM component and a deep component which are integrated in a parallel structure. The FM component is the same as the 2-way factorization machines which is used to model the low-order feature interactions. The deep component is a multi-layered perceptron that is used to capture high-order feature interactions and nonlinearities. These two components share the same inputs/embeddings and their outputs are summed up as the final prediction. It is worth pointing out that the spirit of DeepFM resembles that of the Wide & Deep architecture which can capture both memorization and generalization. The advantages of DeepFM over the Wide & Deep model is that it reduces the effort of hand-crafted feature engineering by identifying feature combinations automatically.

We omit the description of the FM component for brevity and denote the output as $\hat{y}^{(FM)}$. Readers are referred to the last section for more details. Let $\mathbf{e}_i \in \mathbb{R}^k$ denote the latent feature vector of the i^{th} field. The input of the deep component is the concatenation of the dense embeddings of all fields that are looked up with the sparse categorical feature input, denoted as:

$$\mathbf{z}^{(0)} = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_f], \tag{15.10.1}$$

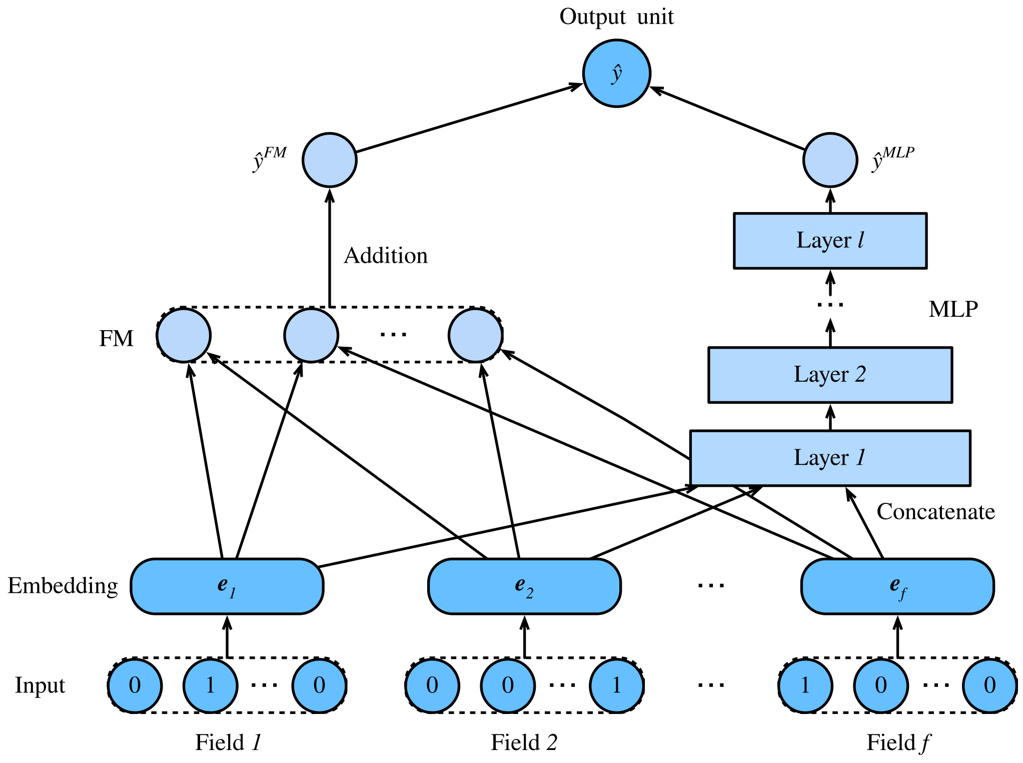
where f is the number of fields. It is then fed into the following neural network:

$$\mathbf{z}^{(l)} = \alpha(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}), \tag{15.10.2}$$

where α is the activation function. $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weight and bias at the l^{th} layer. Let y_{DNN} denote the output of the prediction. The ultimate prediction of DeepFM is the summation of the outputs from both FM and DNN. So we have:

$$\hat{y} = \sigma(\hat{y}^{(FM)} + \hat{y}^{(DNN)}), \tag{15.10.3}$$

where σ is the sigmoid function. The architecture of DeepFM is illustrated below.



It is worth noting that DeepFM is not the only way to combine deep neural networks with FM. We can also add nonlinear layers over the feature interactions (He & Chua, 2017).

```
import d2l
from mxnet import init, gluon, np, npx
from mxnet.gluon import nn
import sys
npx.set_np()
```

15.10.2 Implemenation of DeepFM

The implementation of DeepFM is similar to that of FM. We keep the FM part unchanged and use an MLP block with `relu` as the activation function. Dropout is also used to regularize the model. The number of neurons of the MLP can be adjusted with the `mlp_dims` hyper-parameter.

```
class DeepFM(nn.Block):
    def __init__(self, field_dims, num_factors, mlp_dims, drop_rate=0.1):
        super(DeepFM, self).__init__()
        input_size = int(sum(field_dims))
        self.embedding = nn.Embedding(input_size, num_factors)
        self.fc = nn.Embedding(input_size, 1)
        self.linear_layer = gluon.nn.Dense(1, use_bias=True)
        input_dim = self.embed_output_dim = len(field_dims) * num_factors
        self.mlp = nn.Sequential()
        for dim in mlp_dims:
            self.mlp.add(nn.Dense(dim, 'relu', True, in_units=input_dim))
            self.mlp.add(nn.Dropout(rate=drop_rate))
            input_dim = dim
```

(continues on next page)

(continued from previous page)

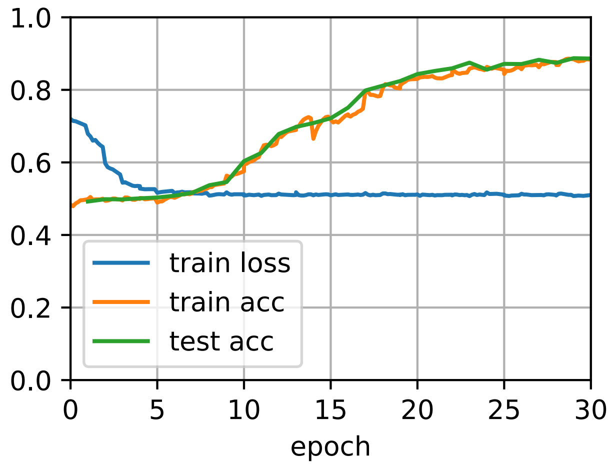
```
        self.mlp.add(nn.Dense(in_units=input_dim, units=1))

    def forward(self, x):
        embed_x = self.embedding(x)
        square_of_sum = np.sum(embed_x, axis=1) ** 2
        sum_of_square = np.sum(embed_x ** 2, axis=1)
        inputs = np.reshape(embed_x, (-1, self.embed_output_dim))
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True) \
            + self.mlp(inputs)
        x = npx.sigmoid(x)
        return x
```

15.10.3 Training and Evaluating the Model

The data loading process is the same as that of FM. We set the MLP component of DeepFM to a three-layered dense network with the a pyramid structure (30-20-10). All other hyper-parameters remain the same as FM.

```
batch_size = 2048
data_dir = d2l.download_extract('ctr')
train_data = d2l.CTRDataset(data_dir+"train.csv")
test_data = d2l.CTRDataset(data_dir+"test.csv",
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
field_dims = train_data.field_dims
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_data, shuffle=True,
                                   last_batch="rollover",
                                   batch_size=batch_size,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(test_data, shuffle=False,
                                  last_batch="rollover",
                                  batch_size=batch_size,
                                  num_workers=num_workers)
ctx = d2l.try_all_gpus()
net = DeepFM(field_dims, num_factors=10, mlp_dims=[30, 20, 10])
net.initialize(init.Xavier(), ctx=ctx)
lr, num_epochs, optimizer = 0.01, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.509, train acc 0.884, test acc 0.886
129692.6 exampes/sec on [gpu(0), gpu(1)]
```



Compared with FM, DeepFM converges faster and achieves better performance.

Summary

- Integrating neural networks to FM enables it to model complex and high-order interactions.
- DeepFM outperforms the original FM on the advertising dataset.

Exercise

- Vary the structure of the MLP to check its impact on model performance.
- Change the dataset to Criteo and compare it with the original FM model.