**智能数据挖掘大作业报告**

人工智能（图灵）

汤栋文 22009200601

2025-06-05

**目录**

[**1.** **电影推荐** 2](#_Toc200047420)

[**1.1 UserCF** 2](#_Toc200047421)

[算法思想： 2](#_Toc200047422)

[优势与局限性： 2](#_Toc200047423)

[流程图： 2](#_Toc200047424)

[代码： 2](#_Toc200047425)

[运行结果： 3](#_Toc200047426)

[**1.2 ItemCF 4**](#_Toc200047427)

[算法思想： 4](#_Toc200047428)

[优势与局限性： 4](#_Toc200047429)

[流程图： 4](#_Toc200047430)

[代码： 4](#_Toc200047431)

[运行结果： 6](#_Toc200047432)

[**2.** **预测广告点击率** 6](#_Toc200047433)

[**2.1 DeepFM 6**](#_Toc200047434)

[算法思想： 6](#_Toc200047435)

[流程图： 6](#_Toc200047436)

[代码： 7](#_Toc200047437)

[运行结果： 9](#_Toc200047438)

## **电影推荐**

## **1.1 UserCF**

## **算法思想：**

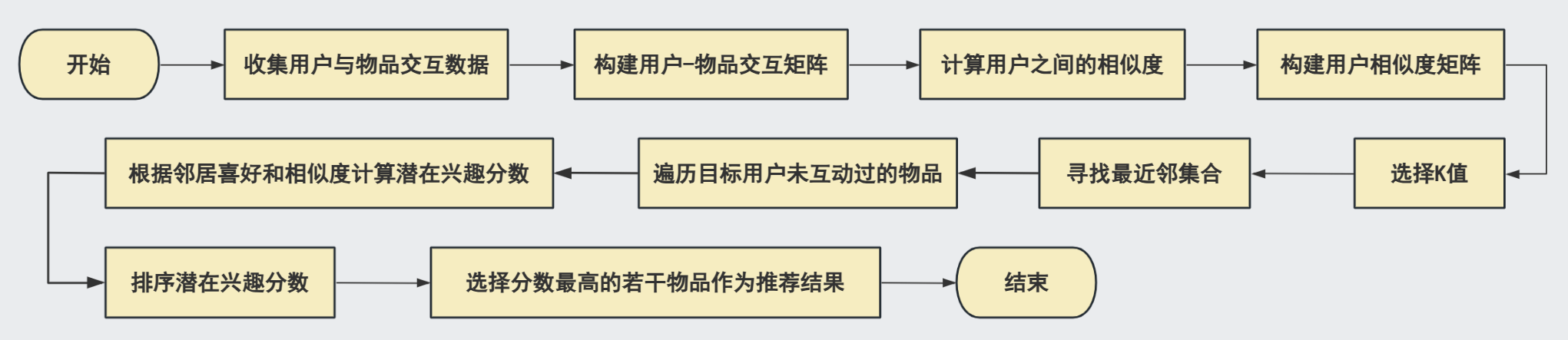
UserCF的基本假设是具有相似兴趣的用户可能会喜欢相同类型的物品（人以群分）。如果两个用户在过去对某些物品表现出相似的喜好，那么他们未来的行为也可能会相似。因此，当一个用户需要个性化推荐时，可以找到与其兴趣相似的一组用户，然后将这些用户喜欢且目标用户尚未接触过的物品推荐给他。

## **优势与局限性：**

**优势：**个性化推荐强：能捕捉到用户的独特偏好；适合冷门物品推荐：即使某些电影评分少，只要有用户喜欢，也可能被推荐给相似用户；适合活跃用户：当用户评分较多时，更容易找到相似用户。

**局限性：**计算复杂度高：随着用户数量增加，用户相似度矩阵的计算代价大；用户兴趣漂移难处理：用户兴趣可能会随时间变化，模型难以及时更新；稀疏性问题敏感：如果用户-物品评分矩阵非常稀疏，用户间相似度计算不准确。

## **流程图：**



## **代码：**

import random

import math

from operator import itemgetter

class UserBasedCF:

    def init(self):

        self.nsimuser = 20

        self.nrecmovie = 10

        self.trainSet = {}

        self.testSet = {}

        self.usersimmatrix = {}

        self.moviecount = 0

        print('Similar user number = %d' % self.nsimuser)

        print('Recommneded movie number = %d' % self.nrecmovie)

    def getdataset(self, filename, pivot=0.75):

        trainSetlen = 0

        testSetlen = 0

        for line in self.loadfile(filename):

            user, movie, rating, timestamp = line.split('::')

            if random.random() < pivot:

                self.trainSet.setdefault(user, {})

                self.trainSet[user][movie] = rating

                trainSetlen += 1

            else:

                self.testSet.setdefault(user, {})

                self.testSet[user][movie] = rating

                testSetlen += 1

        print('Split trainingSet and testSet success!')

        print('TrainSet = %s' % trainSetlen)

        print('TestSet = %s' % testSetlen)

    def loadfile(self, filename):

        with open(filename, 'r') as f:

            for i, line in enumerate(f):

                if i == 0:

                    continue

                yield line.strip('rn')

        print('Load %s success!' % filename)

    def calcusersim(self):

        print('Building movie-user table ...')

        movieuser = {}

        for user, movies in self.trainSet.items():

            for movie in movies:

                if movie not in movieuser:

                    movieuser[movie] = set()

                movieuser[movie].add(user)

        print('Build movie-user table success!')

        self.moviecount = len(movieuser)

        print('Total movie number = %d' % self.moviecount)

        print('Build user co-rated movies matrix ...')

        for movie, users in movieuser.items():

            for u in users:

                for v in users:

                    if u == v:

                        continue

                    self.usersimmatrix.setdefault(u, {})

                    self.usersimmatrix[u].setdefault(v, 0)

                    self.usersimmatrix[u][v] += 1

        print('Build user co-rated movies matrix success!')

        print('Calculating user similarity matrix ...')

        for u, relatedusers in self.usersimmatrix.items():

            for v, count in relatedusers.items():

                self.usersimmatrix[u][v] = count / math.sqrt(len(self.trainSet[u])  len(self.trainSet[v]))

        print('Calculate user similarity matrix success!')

    def recommend(self, user):

        K = self.nsimuser

        N = self.nrecmovie

        rank = {}

        watchedmovies = self.trainSet[user]

        for v, wuv in sorted(self.usersimmatrix[user].items(), key=itemgetter(1), reverse=True)[0:K]:

            for movie in self.trainSet[v]:

                if movie in watchedmovies:

                    continue

                rank.setdefault(movie, 0)

                rank[movie] += wuv

        return sorted(rank.items(), key=itemgetter(1), reverse=True)[0:N]

    def evaluate(self):

        print("Evaluation start ...")

        N = self.nrecmovie

        hit = 0

        reccount = 0

        testcount = 0

        allrecmovies = set()

        for i, user, in enumerate(self.trainSet):

            testmovies = self.testSet.get(user, {})

            recmovies = self.recommend(user)

            for movie, w in recmovies:

                if movie in testmovies:

                    hit += 1

                allrecmovies.add(movie)

            reccount += N

            testcount += len(testmovies)

        precision = hit / (1.0  reccount)

        recall = hit / (1.0  testcount)

        coverage = len(allrecmovies) / (1.0  self.moviecount)

        print('precisioin=%.4ftrecall=%.4ftcoverage=%.4f' % (precision, recall, coverage))

if name == 'main':

    ratingfile = r'./ratings.dat'

    userCF = UserBasedCF()

    userCF.getdataset(ratingfile)

    userCF.calcusersim()

    userCF.evaluate()

## **运行结果：**

Similar user number = 20

Recommneded movie number = 10

Load ./ratings.dat success!

Split trainingSet and testSet success!

TrainSet = 749800

TestSet = 250408

Building movie-user table ...

Build movie-user table success!

Total movie number = 3665

Build user co-rated movies matrix ...

Build user co-rated movies matrix success!

Calculating user similarity matrix ...

Calculate user similarity matrix success!

Evaluation start ...

precisioin=0.3452 recall=0.083 coverage=0.3241

## **1.2 ItemCF**

## **算法思想：**

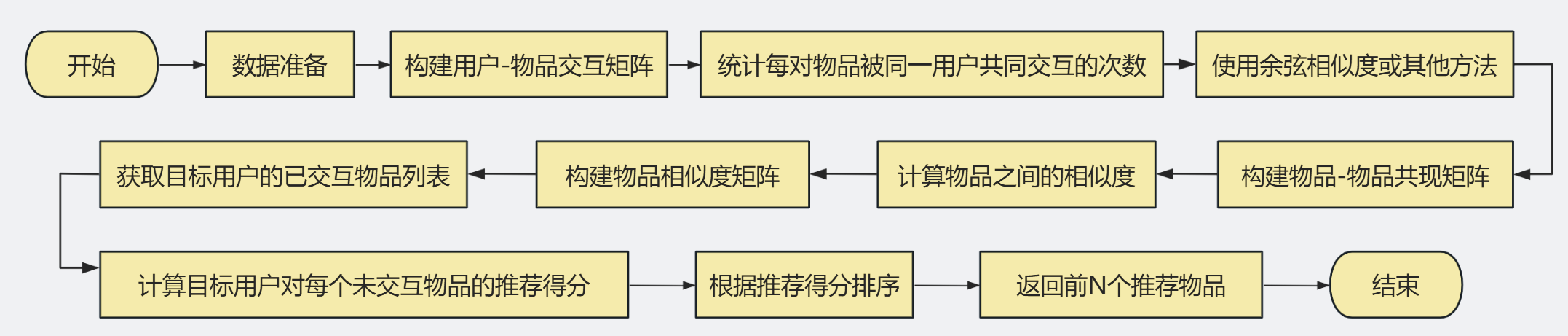
ItemCF的核心思想是：如果一个用户喜欢某个物品，那么他可能会喜欢与该物品相似的其他物品。因此，算法首先需要计算物品之间的相似度。这个相似度并不是基于物品本身的属性，比如电影的导演、演员或者书籍的作者等信息，而是根据用户的行为数据，例如购买历史、评分或浏览行为来推断。即如果很多用户同时喜欢物品A和物品B，那么我们就认为这两个物品是相似的，并且可以互相作为推荐对象。

## **优势与局限性：**

**优势：**计算效率更高：物品数量通常远小于用户数量，且物品相似度可以离线计算；推荐结果更稳定：物品特征相对稳定，适合长期推荐；适合实时推荐场景：新用户也能得到推荐，只要他们有少量评分或点击行为，就能够有效得推荐。

**局限性：**冷门物品推荐能力弱：只有被足够多用户评分过的物品才容易被推荐；新物品冷启动问题严重：新上架的电影由于缺乏评分记录，很难被推荐；推荐多样性差：容易集中在热门或常见类型的电影，缺乏探索性；忽略用户兴趣变化：无法很好地捕捉用户短期兴趣的变化。

## **流程图：**

****

## **代码：**

import random

import math

from operator import itemgetter

class ItemBasedCF:

    def \_\_init\_\_(self):

        self.n\_sim\_movie = 20

        self.n\_rec\_movie = 10

        self.trainSet = {}

        self.testSet = {}

        self.movie\_sim\_matrix = {}

        self.movie\_popular = {}

        self.movie\_count = 0

        print('Similar movie number = %d' % self.n\_sim\_movie)

        print('Recommneded movie number = %d' % self.n\_rec\_movie)

    def get\_dataset(self, filename, pivot=0.75):

        trainSet\_len = 0

        testSet\_len = 0

        for line in self.load\_file(filename):

            user, movie, rating, timestamp = line.split('::')

            if(random.random() < pivot):

                self.trainSet.setdefault(user, {})

                self.trainSet[user][movie] = rating

                trainSet\_len += 1

            else:

                self.testSet.setdefault(user, {})

                self.testSet[user][movie] = rating

                testSet\_len += 1

        print('Split trainingSet and testSet success!')

        print('TrainSet = %s' % trainSet\_len)

        print('TestSet = %s' % testSet\_len)

    def load\_file(self, filename):

        with open(filename, 'r') as f:

            for i, line in enumerate(f):

                if i == 0:

                    continue

                yield line.strip('\r\n')

        print('Load %s success!' % filename)

    def calc\_movie\_sim(self):

        for user, movies in self.trainSet.items():

            for movie in movies:

                if movie not in self.movie\_popular:

                    self.movie\_popular[movie] = 0

                self.movie\_popular[movie] += 1

        self.movie\_count = len(self.movie\_popular)

        print("Total movie number = %d" % self.movie\_count)

        for user, movies in self.trainSet.items():

            for m1 in movies:

                for m2 in movies:

                    if m1 == m2:

                        continue

                    self.movie\_sim\_matrix.setdefault(m1, {})

                    self.movie\_sim\_matrix[m1].setdefault(m2, 0)

                    self.movie\_sim\_matrix[m1][m2] += 1

        print("Build co-rated users matrix success!")

        print("Calculating movie similarity matrix ...")

        for m1, related\_movies in self.movie\_sim\_matrix.items():

            for m2, count in related\_movies.items():

                if self.movie\_popular[m1] == 0 or self.movie\_popular[m2] == 0:

                    self.movie\_sim\_matrix[m1][m2] = 0

                else:

                    self.movie\_sim\_matrix[m1][m2] = \

count / math.sqrt(self.movie\_popular[m1] \* self.movie\_popular[m2])

        print('Calculate movie similarity matrix success!')

    def recommend(self, user):

        K = self.n\_sim\_movie

        N = self.n\_rec\_movie

        rank = {}

        watched\_movies = self.trainSet[user]

        for movie, rating in watched\_movies.items():

            for related\_movie, w in sorted(self.movie\_sim\_matrix[movie].items(),

key=itemgetter(1), reverse=True)[:K]:

                if related\_movie in watched\_movies:

                    continue

                rank.setdefault(related\_movie, 0)

                rank[related\_movie] += w \* float(rating)

        return sorted(rank.items(), key=itemgetter(1), reverse=True)[:N]

    def evaluate(self):

        print('Evaluating start ...')

        N = self.n\_rec\_movie

        hit = 0

        rec\_count = 0

        test\_count = 0

        all\_rec\_movies = set()

        for i, user in enumerate(self.trainSet):

            test\_moives = self.testSet.get(user, {})

            rec\_movies = self.recommend(user)

            for movie, w in rec\_movies:

                if movie in test\_moives:

                    hit += 1

                all\_rec\_movies.add(movie)

            rec\_count += N

            test\_count += len(test\_moives)

        precision = hit / (1.0 \* rec\_count)

        recall = hit / (1.0 \* test\_count)

        coverage = len(all\_rec\_movies) / (1.0 \* self.movie\_count)

        print('precisioin=%.4f\trecall=%.4f\tcoverage=%.4f' % (precision, recall, coverage))

if \_\_name\_\_ == '\_\_main\_\_':

    rating\_file = './ratings.dat'

    itemCF = ItemBasedCF()

    itemCF.get\_dataset(rating\_file)

    itemCF.calc\_movie\_sim()

    itemCF.evaluate()

## **运行结果：**

Similar movie number = 20

Recommneded movie number = 10

Load ./ratings.dat success!

Split trainingSet and testSet success!

TrainSet = 750100

TestSet = 250108

Total movie number = 3666

Build co-rated users matrix success!

Calculating movie similarity matrix ...

Calculate movie similarity matrix success!

Evaluating start ...

precisioin=0.3447 recall=0.0832 coverage=0.1691

## **预测广告点击率**

## **2.1 DeepFM**

## **算法思想：**

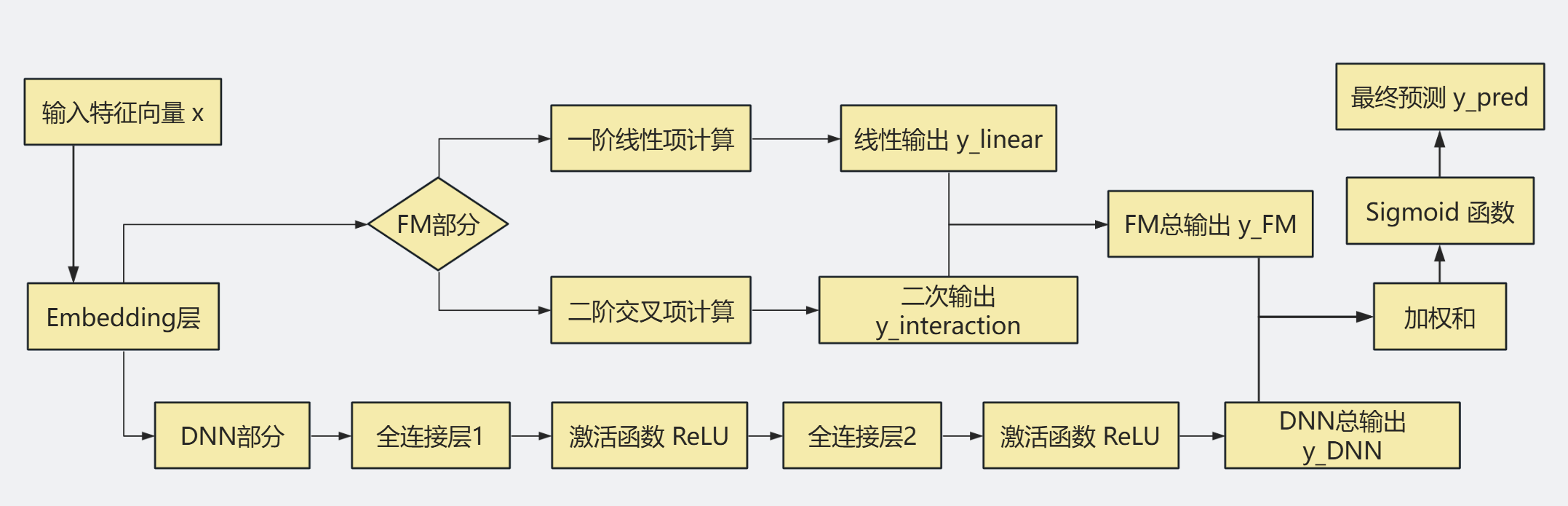
DeepFM模型由两大部分组成：一个FM部分和一个DNN部分。这两个部分共享相同的输入层和嵌入层，这意味着它们使用相同的特征表示进行训练。

**FM部分：**FM部分负责捕捉低阶特征交互，如一阶和二阶特征组合。一阶项是指线性特征权重，而二阶项通过隐向量内积建模特征交叉。具体来说，给定一个特征向量，其对应的FM输出可以表示为：  
其中，是全局偏置，是一阶特征权重，是第个和第个特征对应的隐向量的点积，用于计算特征间的交互作用。

**DNN部分：**Deep部分则是一个多层前馈神经网络，它捕捉的是高阶非线性特征交互。特征首先通过嵌入层转换为稠密向量，然后这些向量被拼接起来并输入到全连接网络中。这种结构允许模型自动学习复杂的特征组合模式，而不需要显式的特征工程。在Deep部分，原始特征经过嵌入后形成稠密向量，接着通过多个隐藏层进行变换，每一层都应用激活函数，以引入非线性元素，最终输出一个值或向量，代表该部分对目标变量的预测贡献。

**模型架构：**DeepFM的架构设计旨在利用FM的高效性和DNN的强大表达能力。模型的整体预测值是FM部分和Deep部分输出的加权和，通常会通过sigmoid函数将这个总和映射到[0, 1]区间，作为最终的点击概率估计。Embedding层：将稀疏的离散特征转换成稠密的特征向量，使得不同field的向量长度相同，便于后续计算。FM层：用于计算交叉特征，捕捉低阶特征交互。DNN部分：捕捉高阶特征交互，通过多层非线性变换提高模型表达力。输出层：融合FM层和DNN部分的输出得到最终的预测结果。

## **流程图：**

****

## **代码：**

**# File: DeepFM.py**

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

from time import time

class DeepFM(nn.Module):

    def \_\_init\_\_(self, feature\_sizes, embedding\_size=4,

                 hidden\_dims=[32, 32], num\_classes=1, dropout=[0.5, 0.5],

                 use\_cuda=True, verbose=False):

        """

        Initialize a new network

        Inputs:

        - feature\_size: A list of integer giving the size of features for each field.

        - embedding\_size: An integer giving size of feature embedding.

        - hidden\_dims: A list of integer giving the size of each hidden layer.

        - num\_classes: An integer giving the number of classes to predict. For example,

                    someone may rate 1,2,3,4 or 5 stars to a film.

        - batch\_size: An integer giving size of instances used in each interation.

        - use\_cuda: Bool, Using cuda or not

        - verbose: Bool

        """

        super().\_\_init\_\_()

        self.field\_size = len(feature\_sizes)

        self.feature\_sizes = feature\_sizes

        self.embedding\_size = embedding\_size

        self.hidden\_dims = hidden\_dims

        self.num\_classes = num\_classes

        self.dtype = torch.long

        self.bias = torch.nn.Parameter(torch.randn(1))

        # check if use cuda

        if use\_cuda and torch.cuda.is\_available():

            self.device = torch.device('cuda')

        else:

            self.device = torch.device('cpu')

        # init fm part

        self.fm\_first\_order\_embeddings = nn.ModuleList(

            [nn.Embedding(feature\_size, 1) for feature\_size in self.feature\_sizes])

        self.fm\_second\_order\_embeddings = nn.ModuleList(

            [nn.Embedding(feature\_size, self.embedding\_size) for feature\_size in self.feature\_sizes])

        # init deep part

        all\_dims = [self.field\_size \* self.embedding\_size] + \

            self.hidden\_dims + [self.num\_classes]

        for i in range(1, len(hidden\_dims) + 1):

            setattr(self, 'linear\_'+str(i),

                    nn.Linear(all\_dims[i-1], all\_dims[i]))

*# nn.init.kaiming\_normal\_(self.fc1.weight)*

            setattr(self, 'batchNorm\_' + str(i),

                    nn.BatchNorm1d(all\_dims[i]))

            setattr(self, 'dropout\_'+str(i),

                    nn.Dropout(dropout[i-1]))

    def forward(self, Xi, Xv):

        """

        Forward process of network.

        Inputs:

        - Xi: A tensor of input's index, shape of (N, field\_size, 1)

        - Xv: A tensor of input's value, shape of (N, field\_size, 1)

        """

        # fm part

        fm\_first\_order\_emb\_arr = [(torch.sum(emb(Xi[:, i, :]), 1).t() \* Xv[:, i]).t()

for i, emb in enumerate(self.fm\_first\_order\_embeddings)]

        fm\_first\_order = torch.cat(fm\_first\_order\_emb\_arr, 1)

        fm\_second\_order\_emb\_arr = [(torch.sum(emb(Xi[:, i, :]), 1).t() \* Xv[:, i]).t()

for i, emb in enumerate(self.fm\_second\_order\_embeddings)]

        fm\_sum\_second\_order\_emb = sum(fm\_second\_order\_emb\_arr)

        fm\_sum\_second\_order\_emb\_square = fm\_sum\_second\_order\_emb \* \

            fm\_sum\_second\_order\_emb  *# (x+y)^2*

        fm\_second\_order\_emb\_square = [

            item\*item for item in fm\_second\_order\_emb\_arr]

        fm\_second\_order\_emb\_square\_sum = sum(

            fm\_second\_order\_emb\_square)  *# x^2+y^2*

        fm\_second\_order = (fm\_sum\_second\_order\_emb\_square -

                           fm\_second\_order\_emb\_square\_sum) \* 0.5

        # deep part

        deep\_emb = torch.cat(fm\_second\_order\_emb\_arr, 1)

        deep\_out = deep\_emb

        for i in range(1, len(self.hidden\_dims) + 1):

            deep\_out = getattr(self, 'linear\_' + str(i))(deep\_out)

            deep\_out = getattr(self, 'batchNorm\_' + str(i))(deep\_out)

            deep\_out = getattr(self, 'dropout\_' + str(i))(deep\_out)

        # sum

        total\_sum = torch.sum(fm\_first\_order, 1) + \

                    torch.sum(fm\_second\_order, 1) + torch.sum(deep\_out, 1) + self.bias

        return total\_sum

    def fit(self, loader\_train, loader\_val, optimizer, scheduler, epochs, verbose=False, print\_every=100):

        """

        Training a model and valid accuracy.

        Inputs:

        - loader\_train: I

        - loader\_val: .

        - optimizer: Abstraction of optimizer used in training process.

        - epochs: Integer, number of epochs.

        - verbose: Bool, if print.

        - print\_every: Integer, print after every number of iterations.

        """

        # load input data

        model = self.train().to(device=self.device)

        criterion = F.binary\_cross\_entropy\_with\_logits

        for \_ in range(epochs):

            for t, (xi, xv, y) in enumerate(loader\_train):

                xi = xi.to(device=self.device, dtype=self.dtype)

                xv = xv.to(device=self.device, dtype=torch.float)

                y = y.to(device=self.device, dtype=torch.float)

                total = model(xi, xv)

                loss = criterion(total, y)

                optimizer.zero\_grad()

                loss.backward()

                optimizer.step()

                scheduler.step()

                if verbose and t % print\_every == 0:

                    print('Iteration %d, loss = %.4f' % (t, loss.item()))

                    self.check\_accuracy(loader\_val, model)

                    print()

    def check\_accuracy(self, loader, model):

        if loader.dataset.train:

            print('Checking accuracy on validation set')

        else:

            print('Checking accuracy on test set')

        num\_correct = 0

        num\_samples = 0

        model.eval()  *# set model to evaluation mode*

        with torch.no\_grad():

            for xi, xv, y in loader:

                xi = xi.to(device=self.device, dtype=self.dtype)  *# move to device, e.g. GPU*

                xv = xv.to(device=self.device, dtype=torch.float)

                y = y.to(device=self.device, dtype=torch.bool)

                total = model(xi, xv)

                preds = (F.sigmoid(total) > 0.5)

                num\_correct += (preds == y).sum()

                num\_samples += preds.size(0)

            acc = float(num\_correct) / num\_samples

            print('Got %d / %d correct (%.2f%%)' % (num\_correct, num\_samples, 100 \* acc))

**# File dataset.py**

import torch

from torch.utils.data import Dataset

import pandas as pd

import numpy as np

import os

continous\_features = 13

class CriteoDataset(Dataset):

    def \_\_init\_\_(self, root, train=True):

        """

        Initialize file path and train/test mode.

        Inputs:

        - root: Path where the processed data file stored.

        - train: Train or test. Required.

        """

        self.root = root

        self.train = train

        if not self.\_check\_exists():

            raise RuntimeError('Dataset not found.')

        if self.train:

            data = pd.read\_csv(os.path.join(root, 'train.txt'))

            self.train\_data = data.iloc[:, :-1].values

            self.target = data.iloc[:, -1].values

        else:

            data = pd.read\_csv(os.path.join(root, 'test.txt'))

            self.test\_data = data.iloc[:, :-1].values

    def \_\_getitem\_\_(self, idx):

        if self.train:

            dataI, targetI = self.train\_data[idx, :], self.target[idx]

*# index of continous features are zero*

            Xi\_coutinous = np.zeros\_like(dataI[:continous\_features])

            Xi\_categorial = dataI[continous\_features:]

            Xi = torch.from\_numpy(np.concatenate(

(Xi\_coutinous, Xi\_categorial)).astype(np.int32)).unsqueeze(-1)

*# value of categorial features are one (one hot features)*

            Xv\_categorial = np.ones\_like(dataI[continous\_features:])

            Xv\_coutinous = dataI[:continous\_features]

            Xv = torch.from\_numpy(np.concatenate((Xv\_coutinous, Xv\_categorial)).astype(np.int32))

            return Xi, Xv, targetI

        else:

            dataI = self.test\_data.iloc[idx, :]

*# index of continous features are one*

            Xi\_coutinous = np.ones\_like(dataI[:continous\_features])

            Xi\_categorial = dataI[continous\_features:]

            Xi = torch.from\_numpy(np.concatenate(

(Xi\_coutinous, Xi\_categorial)).astype(np.int32)).unsqueeze(-1)

*# value of categorial features are one (one hot features)*

            Xv\_categorial = np.ones\_like(dataI[continous\_features:])

            Xv\_coutinous = dataI[:continous\_features]

            Xv = torch.from\_numpy(np.concatenate((Xv\_coutinous, Xv\_categorial)).astype(np.int32))

            return Xi, Xv

    def \_\_len\_\_(self):

        if self.train:

            return len(self.train\_data)

        else:

            return len(self.test\_data)

    def \_check\_exists(self):

        return os.path.exists(self.root)

**# File train.py**

import numpy as np

import torch.optim as optim

from torch.utils.data import DataLoader

from torch.utils.data import sampler

from model.DeepFM import DeepFM

from data.dataset import CriteoDataset

*# 10000 items for training, 10000 items for valid, of all 20000 items*

Num\_train = 50

*# load data*

train\_data = CriteoDataset('./data', train=True)

loader\_train = DataLoader(train\_data, batch\_size=200,

                          sampler=sampler.SubsetRandomSampler(range(Num\_train)))

val\_data = CriteoDataset('./data', train=True)

loader\_val = DataLoader(val\_data, batch\_size=200,

                        sampler=sampler.SubsetRandomSampler(range(Num\_train, 100)))

feature\_sizes = np.loadtxt('./data/feature\_sizes.txt', delimiter=',')

feature\_sizes = [int(x) for x in feature\_sizes]

print(feature\_sizes)

epochs = 1000

model = DeepFM(feature\_sizes, use\_cuda=True, embedding\_size=256,

hidden\_dims=[256, 256, 256], dropout=[0.2, 0.2, 0.2])

optimizer = optim.AdamW(model.parameters(), lr=3e-4, weight\_decay=0.1)

scheduler = optim.lr\_scheduler.CosineAnnealingLR(optimizer, T\_max=epochs, eta\_min=1e-6)

model.fit(loader\_train, loader\_val, optimizer, scheduler, epochs=epochs, verbose=True, print\_every=1000)

## **运行结果：**

Iteration 0, loss = 69403.1562

Checking accuracy on validation set

Got 40 / 50 correct (80.00%)