

推荐系统概述

互联网和移动技术的快速发展，一方面让更多的个人随时随地可以接入互联网获取信息，另一方面也让个人和网站更加快捷的提供UGC（User Generated Content，用户原创内容）和PGC（Professional Generated Content，专业生产内容）的内容、商品。因此，导致了网络中出现了大量的用户和海量的内容数据。这种从信息匮乏到信息爆炸的变化，使得人找信息和信息找人，都变得越来越困难。



在哪里

在这样的大环境下，搜索推荐系统及相关技术近几年迎来了高速发展。简单来说，推荐系统就是根据用户的个性化需求，在海量的信息中确定提供给用户什么样的具体内容。这种个性化需求可以是用户当前的搜索显性需求；也可以是根据用户历史浏览习惯和行为挖掘到的隐性需求。一旦确定需求，就可以将合适的个性化内容推荐给用户，这样就会让一种“你怎么这么懂我”的感觉在用户的心中油然而生，殊不知这完全是智能化的推荐系统在其中发挥了巨大作用！



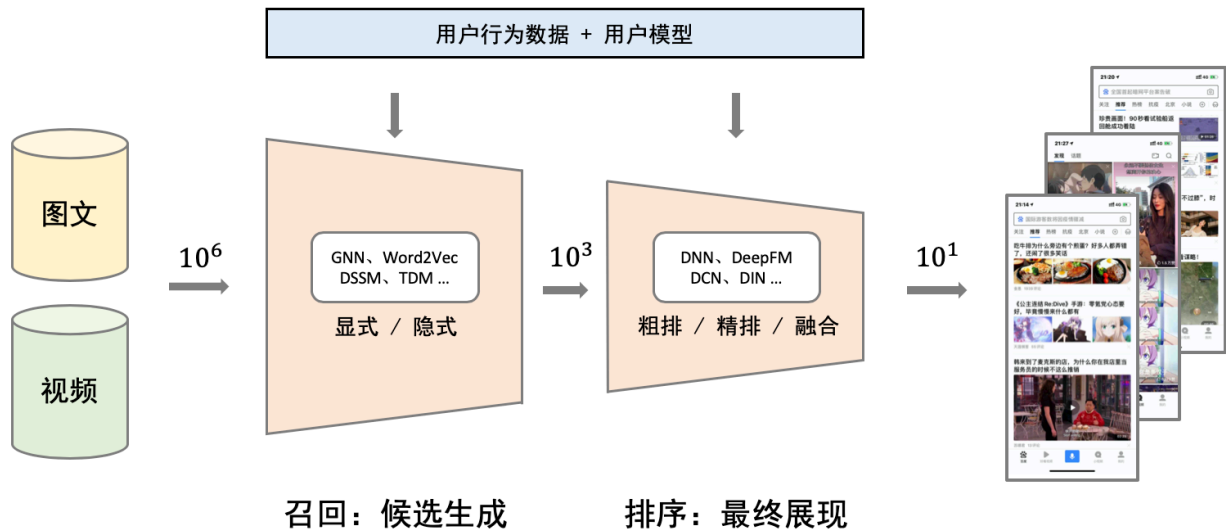
推荐系统在互联网和传统行业中都有着大量的应用。在互联网行业，几乎所有的互联网平台都应用了推荐系统，如资讯新闻、影视剧、知识社区的内容推荐，以及电商平台的商品推荐等；而在传统行业中，有些用于企业的营销环节，如银行的金融产品推荐、保险公司的保险产品推荐等。根据 QM (QuestMobile) 报告，以推荐系统技术为核心的短视频行业在近年的用户规模已超9亿，市场规模超2千亿，由此可见这项技术在现代社会的经济价值。



推荐系统意义与典型结构

- 推荐系统是在互联网信息爆炸式增长的时代背景下，帮助用户高效获得感兴趣信息的关键；
- 推荐系统也是帮助产品最大限度吸引用户、留存用户、增加用户粘性、提高用户转化率的银弹。
- 有无数优秀的产品依靠用户可感知的推荐系统建立了良好的口碑，也有无数的公司依靠直击用户痛点的推荐系统在行业中占领了一席之地。

可以说，谁能掌握和利用好推荐系统，谁就能在信息分发的激烈竞争中抢得先机。但与此同时，有着许多问题困扰着推荐系统的开发者，比如：庞大的数据量，复杂的模型结构，低效的分布式训练环境，波动的在离线一致性，苛刻的上线部署要求，以上种种，不胜枚举。



MovieLens数据集介绍

[MovieLens数据集](#)是一个关于电影评分的数据集，数据来自于IMDB, The Movie DataBase等电影评分网站。该数据集中包含用户对电影的评分信息，用户的人口统计学特征以及电影的描述特征，非常适合用来入门推荐系统。

本推荐任务最终的目的是对给定用户，基于该用户历史的电影评分数据，给他推荐他可能感兴趣的其他电影。一般来说，推荐流程主要包含两部分：召回、排序。

- 召回阶段的目标主要在于降低候选集规模，从全量的候选集中得到用户可能感兴趣的一小部分候选集；
- 排序阶段则是将召回阶段得到的候选集进行精准排序，推荐给用户。本文后续会依次从数据处理、离线训练、离线预估、在线召回、在线排序出发，逐一进行细化讲解。

我们尝试使用该数据集，训练一个推荐系统，完成完整的从召回到排序的推荐流程，力争能够在测试集上，根据用户的特征信息，为他/她推荐一部喜欢的电影。

我们最终根据推荐的电影的类别特征与用户的喜好类别进行比较，来评价我们推荐系统是否有效。

数据处理

本教程采用MovieLens网站的用户-电影评分数据集[ml-1m](#)，该数据集共包含了6000多位用户对近3900个电影的100多万条评分数据，评分为1~5的整数，其中每个电影的评分数据至少有20条。

该数据集包含三个数据文件，分别是：

- users.dat，存储用户属性信息的txt格式文件，格式为 `UserID::Gender::Age::Occupation`，其中原始数据中对用户年龄和职业做了离散化处理。

。

user_id	性别	年龄	职业
2181	男	25	自由职业
2182	女	25	学生
2183	女	56	教师
...

- movies.dat, 存储电影属性信息的txt格式文件, 格式: `MovieID::Title::Genres`。

o

movie_id	title	类别
260	Star Wars:Episode IV(1977)	动作, 科幻
1270	Three Amigos!(1986)	喜剧
2763	Thomas Crown Affair,The(1999)	动作, 惊悚
2858	American Beauty(1999)	喜剧
...

- ratings.dat, 存储电影评分信息的txt格式文件, 格式: `UserID::MovieID::Rating::time`

o

user_id	movie_id	评分	评分时间
2181	2858	4分	974609869
2181	260	5分	974608048
2181	1270	5分	974666558
2182	3481	2分	974607734
2183	2605	3分	974608443
2183	1210	4分	974607751
...

本任务中, 数据处理主要包含两步, 数据集划分及训练数据生成。

- 数据集划分。**根据ratings.dat中的评分时间, 选每个用户最新的一条数据, 作为测试集; 剩余数据作为训练集。
- 数据生成。**对于rating.txt文件中每一条数据, 根据userid和movieid去 `users.dat` 和 `movies.dat` 里找到相应的用户特征和电影特征, 经过hash处理后拼接成一条可真实用于训练或预估的数据。拼接后, 数据格式如下所示, 其中数据分隔符为空格。

```
logid:691 time:974611432 userid:157041 gender:345888 age:950432
occupation:230771 movieid:766238 title:258563 title:357193 genres:171819
genres:257080 label:2
```

数据格式

对于 `ratings.dat` 文件中每一条数据，我们可以根据 `userid` 和 `movieid` 去 `users.dat` 和 `movies.dat` 里找到相应的用户特征和电影特征，经过 `hash` 处理后拼接成一条可真实用于训练或预估的数据。拼接后数据格式如下所示，第一个数字是 `logid`，即数据的 ID，各个数据间的分隔符为空格。

```
logid:10000004 time:971979246 userid:313460 gender:113660 age:144970
occupation:209266 movieid:466203 title:504 title:66476 title:0 title:0
genres:249254 genres:332172 genres:0 label:3
```

在每条数据中，电影的电影名 `title` 和电影风格 `genres` 可能需要拆分处理成多个词，形成多组 `slot:feassign` 样式的数据。这是因为电影名一般由多个单词组成，这个很容易理解，也很常见，所以不再赘述；而电影风格也可能是多种，例如《异形》，既是恐怖片又是科幻片。

`Slot:Feassign` 是什么？

`Slot` 直译是槽位，在推荐工程中，是指某一个宽泛的特征类别，比如用户 ID、性别、年龄就是 `Slot`，`Feassign` 则是具体值，比如：12345，男，20 岁。

在实践过程中，很多特征槽位不是单一属性，或无法量化并且离散稀疏的，比如某用户兴趣爱好有三个：游戏/足球/数码，且每个具体兴趣又有多个特征维度，则在兴趣爱好这个 `Slot` 兴趣槽位中，就会有多个 `Feassign` 值。

`PaddleRec` 在读取数据时，每个 `Slot ID` 对应的特征，支持稀疏，且支持变长，可以非常灵活的支持各种场景的推荐模型训练。

上述数据格式，是非常典型的互联网推荐系统构建的数据格式，我们工作中一般会写脚本基于不同的数据源整合得到标准格式，比如这里是基于 `users.dat`、`movies.dat` 和 `ratings.dat` 三个文件制作标准推荐格式数据集。

这里的数据，大家可以参考脚本 `data_prepare.sh` 完成数据集的制作，形成训练集和测试集，我们把处理好的数据保存在 `data` 目录下的 `train` 和 `test` 文件夹中。数据已经准备好后，我们再来看下如何让模型读取数据。

为您的模型自定义 Reader

`PaddleRec` 支持您使用自定义的格式进行输入。不过您需要一个单独的 Python 文件进行描述。在自定义的 Reader 中，您需要引入 `IterableDataset` 基类，创建一个名为 `RecDataset` 子类，继承 `IterableDataset` 的基类。继承并实现基类中的 `iter(self)` 函数，逐行读取数据。您可以根据需要自己定义数据读取的逻辑，对以行为单位的数据进行截取，转换等预处理，返回一个可以迭代的 reader 方法。数据的输出顺序与我们在网络中创建的 `inputs` 必须是严格一一对应的，并以 `np.array` 的形式输入网络中。最后，在动态图/静态图模型文件 `dygraph_moedl.py/static_model.py` 中，加入自定义的 Reader 接收数据。本教程中使用如下代码：`movie_reader_dygraph.py` 读取电影数据,作为示例

```
from __future__ import print_function
```

```

import numpy as np
#引入IterableDataset基类
from paddle.io import IterableDataset

#创建一个子类，继承IterableDataset的基类
class RecDataset(IterableDataset):
    def __init__(self, file_list, config):
        super(RecDataset, self).__init__()
        self.file_list = file_list

    def __iter__(self):
        full_lines = []
        self.data = []
        for file in self.file_list:
            with open(file, "r") as rf:
                # 以行为单位，逐行读取数据
                for l in rf:
                    output_list = []
                    line = l.strip().split(" ")
                    sparse_slots = ["logid", "time", "userid", "gender", "age",
"occupation", "movieid", "title", "genres", "label"]
                    #logid和time这两个特征，训练模型时并不需要用到，故不必加入
output_list

                    logid = line[0].strip().split(":")[1]
                    time = line[1].strip().split(":")[1]

                    #向output_list中加入用户特征：userid:1个数，gender:1个数，age:1
个数，occupation:1个数
                    userid = line[2].strip().split(":")[1]
                    output_list.append(np.array([float(userid)]))
                    gender = line[3].strip().split(":")[1]
                    output_list.append(np.array([float(gender)]))
                    age = line[4].strip().split(":")[1]
                    output_list.append(np.array([float(age)]))
                    occupation = line[5].strip().split(":")[1]
                    output_list.append(np.array([float(occupation)]))

                    #向output_list中加入电影特征：movieid:1个数，title:4个数，
genres:3个数
                    movieid = line[6].strip().split(":")[1]
                    output_list.append(np.array([float(movieid)]))

                    title = []
                    genres = []
                    for i in line:
                        if i.strip().split(":")[0] == "title":
                            title.append(float(i.strip().split(":")[1]))
                        if i.strip().split(":")[0] == "genres":
                            genres.append(float(i.strip().split(":")[1]))

```



```
output_list.append(np.array(title))
output_list.append(np.array(genres))

#向output_list中加入标签: label:1个数
label = line[-1].strip().split(":")[1]
output_list.append(np.array([float(label)]))

#返回一个可以迭代的reader方法
yield output_list
```

方便起见，本教程提供一键数据处理脚本 `bash data_prepare.sh` 将数据集的格式转化为 `slot:feassign` 模式。

模型设计

个性化推荐系统一般包括两个模块，召回和排序。排序是指经过推理将用户感兴趣的内容按照点击率由高到低推荐给用户，一般对于数据量不大的业务只使用排序模块即可。但是对于数据量极大的业务，如果仅用排序模块从海量数据中推理出用户感兴趣的内容，不仅推理速度较慢，而且准确率较低，因此在该场景中还需要使用召回模块。召回模块会以多路并发的形式，在全量的商品库中从不同的角度（标签匹配、深度模型、热门等）筛选出用户可能感兴趣的商品作为候选数据集，然后再由排序模块对候选集进行精准排序，推荐给用户。首先我们来看下召回模型。

说明：对于一些较为复杂的场景，还需要准备用户模型和内容模型，用户模型可以从用户信息和行为数据中提取出用户特征，而内容模型，还是以电影推荐模型为例，复杂的场景还需要有个电影模型，提取电影信息的特征。召回和排序模型使用这些数据特征作为训练输入，可以更准确的判断出内容或信息与用户兴趣方向上的匹配程度，让推荐系统做出更准确的推送。

召回模型

目前的深度学习召回模型有几大类型：DNN、双塔语义召回、RNN序列召回和深度树匹配召回等。

(1) **DNN**：以经典的Youtube DNN为代表，相比于传统的协同过滤方法，DNN是使用统一的（用户、物品）向量空间来代替原来的两个独立的向量空间，使用深度网络将用户、物品映射到这个统一的低维向量空间来发现学习更高阶的用户物品相似性。

(2) **双塔语义召回**：以微软2013年的DSSM（Deep Structured Semantic Models）模型为代表，该模型是将不同对象映射到统一的语义空间中，利用该空间中对象的距离计算相似度。因为会为用户和商品分别建立一路模型，最后计算相似度，因此该类模型又称为“双塔”模型。

(3) **深度树匹配召回**：以阿里巴巴的TDM（Tree-based Deep Match）模型为例，该模型直接利用高级深度学习模型在全库范围内检索用户兴趣。其基本原理是使用树结构对全库item进行索引，然后训练深度模型以支持树上的逐层检索。该模型试图通过结合树结构搜索与深度学习模型来解决召回的高性能需求与使用复杂模型进行全局搜索与之间的平衡。它将召回问题转化为层级化分类问题，借助树的层级检索可以将时间复杂度降到对数级。

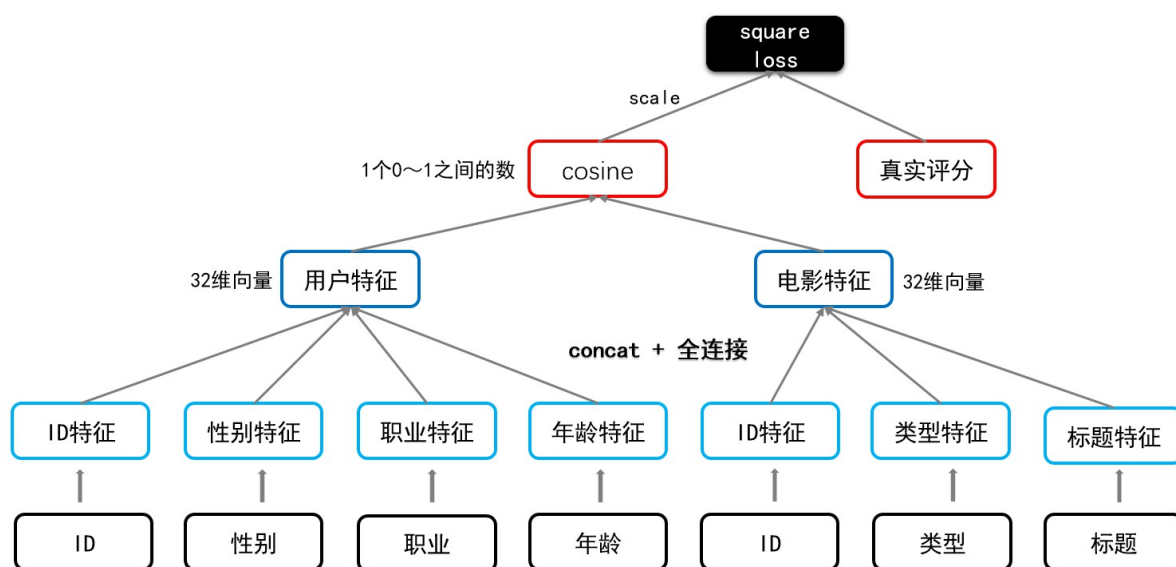
(4) **RNN序列召回**：基于用户session中的点击序列进行建模召回有很多种方式，其中使用RNN神经网络结构来刻画是其中比较有代表性的一种。相应的网络结构其实很简单。

模型解析

本示例中，召回的目的是从大量电影库中选出部分候选，输入给排序模块，用以提高排序模型性能和效果，因此召回模型的候选范围比较大，可以采用比较简单的模型。经典的如ItemCF、UserCF、DSSM、Graph、TDM等，本教程选取了DSSM模型的简化版，即普通的双塔模型，模型组网如下图所示。左侧对用户特征建模，得到用户表示，右侧对电影特征建模，得到电影表示，将每个原始特征转变成Embedding表示，再合并成一个用户特征向量和一个电影特征向量。然后计算用户和电影的相似度（内积或cosine），对于用户评分较高的电影，电影的特征向量和用户的特征向量应该高度相似，反之则相异。最后与训练样本（已知的用户对电影的评分）做损失计算。Loss函数采用均方误差函数。网络结构的说明如下所示：

1. 第一层结构：将原始的输入特征通过Embedding变换为原始的特征集合。
2. 第二层结构：特征变换，将原始特征集合变换为用户和电影两个特征向量。
3. 第三层结构：计算向量相似度。为确保结果与电影评分可比较，两个特征向量的相似度从【0~1】缩放5倍到【0~5】
4. 第四层结构：计算Loss，计算缩放后的相似度与用户对电影的真实评分的“均方误差”。

如果想了解示例中召回模型的组网代码，可以参考文件 `recall/net.py`。



说明：在自然语言处理中，我们常使用词嵌入（Embedding）的方式完成将数据映射为向量的变换。Embedding是一个嵌入层，将输入的非负整数矩阵中的每个数值，转换为具有固定长度的向量。在NLP任务中，一般把输入文本映射成向量表示，以便神经网络的处理。在数据处理章节，我们已经将用户和电影的特征用数字表示。嵌入层Embedding可以完成数字到向量的映射。

输入部分

示例中的 `dygraph_model.py` 将使用如下代码读取数据

```
def create_feeds(batch):
    user_sparse_inputs = [
        paddle.to_tensor(batch[i].numpy().astype('int64').reshape(-1, 1))
        for i in range(4)
    ]

    mov_sparse_inputs = [
        paddle.to_tensor(batch[4].numpy().astype('int64').reshape(-1, 1)),
        paddle.to_tensor(batch[5].numpy().astype('int64').reshape(-1, 4)),
        paddle.to_tensor(batch[6].numpy().astype('int64').reshape(-1, 3))
    ]

    label_input =
    paddle.to_tensor(batch[7].numpy().astype('int64').reshape(-1, 1))

    return user_sparse_inputs, mov_sparse_inputs, label_input
```

组网

组网部分 `net.py` 的代码如下所示。

```
class DNNLayer(nn.Layer):
    #在使用动态图时，针对一些比较复杂的网络结构，可以使用Layer子类定义的方式来进行模型代码编写，在
    __init__ 构造函数中进行组网Layer的声明，
    #在forward中使用声明的Layer变量进行前向计算。子类组网方式也可以实现sublayer的复用，针对相同
    的layer可以在构造函数中一次性定义，在forward中多次调用。
    def __init__(self, sparse_feature_number, sparse_feature_dim, fc_sizes):
        super(DNNLayer, self).__init__()
        self.sparse_feature_number = sparse_feature_number
        self.sparse_feature_dim = sparse_feature_dim
        self.fc_sizes = fc_sizes

        #声明embedding层，建立emb表将数据映射为向量
        self.embedding = paddle.nn.Embedding(
            self.sparse_feature_number,
            self.sparse_feature_dim,
            padding_idx=0,
            sparse=True,
            weight_attr=paddle.ParamAttr(
                name="SparseFeatFactors",
                initializer=paddle.nn.initializer.Uniform()))

        #使用循环的方式创建全连接层，可以在超参数中通过一个数组确定使用几个全连接层以及每个全
        连接层的神经元数量。
```

#本例中使用了4个全连接层，并在每个全连接层后增加了relu激活层。

```
user_sizes = [36] + self.fc_sizes
acts = ["relu" for _ in range(len(self.fc_sizes))]
self._user_layers = []
for i in range(len(self.fc_sizes)):
    linear = paddle.nn.Linear(
        in_features=user_sizes[i],
        out_features=user_sizes[i + 1],
        weight_attr=paddle.ParamAttr(
            initializer=paddle.nn.initializer.Normal(
                std=1.0 / math.sqrt(user_sizes[i]))))
    self.add_sublayer('linear_user_%d' % i, linear)
    self._user_layers.append(linear)
    if acts[i] == 'relu':
        act = paddle.nn.ReLU()
        self.add_sublayer('user_act_%d' % i, act)
        self._user_layers.append(act)
```

#电影特征和用户特征使用了不同的全连接层，不共享参数

```
movie_sizes = [27] + self.fc_sizes
acts = ["relu" for _ in range(len(self.fc_sizes))]
self._movie_layers = []
for i in range(len(self.fc_sizes)):
    linear = paddle.nn.Linear(
        in_features=movie_sizes[i],
        out_features=movie_sizes[i + 1],
        weight_attr=paddle.ParamAttr(
            initializer=paddle.nn.initializer.Normal(
                std=1.0 / math.sqrt(movie_sizes[i]))))
    self.add_sublayer('linear_movie_%d' % i, linear)
    self._movie_layers.append(linear)
    if acts[i] == 'relu':
        act = paddle.nn.ReLU()
        self.add_sublayer('movie_act_%d' % i, act)
        self._movie_layers.append(act)
```

```
def forward(self, batch_size, user_sparse_inputs, mov_sparse_inputs,
label_input):
```

#对用户特征建模， 所有用户sparse特征查对应的emb表，获得特征权重

```
user_sparse_embed_seq = []
for s_input in user_sparse_inputs:
    emb = self.embedding(s_input)
    emb = paddle.reshape(emb, shape=[-1, self.sparse_feature_dim])
    user_sparse_embed_seq.append(emb)
```

#对电影特征建模， 所有电影sparse特征查对应的emb表，获得特征权重

```
mov_sparse_embed_seq = []
for s_input in mov_sparse_inputs:
```

```

        s_input = paddle.reshape(s_input, shape = [batch_size, -1])
        emb = self.embedding(s_input)
        emb = paddle.sum(emb, axis = 1)
        emb = paddle.reshape(emb, shape=[-1, self.sparse_feature_dim])
        mov_sparse_embed_seq.append(emb)

        #查表结果拼接在一起，构成用户特征权重向量
        user_features = paddle.concat(user_sparse_embed_seq,axis=1)
        #查表结果拼接在一起，构成电影特征权重向量
        mov_features = paddle.concat(mov_sparse_embed_seq,axis=1)

        #通过4层全链接层，获得用于计算相似度的用户特征和电影特征
        for n_layer in self._user_layers:
            user_features = n_layer(user_features)

        for n_layer in self._movie_layers:
            mov_features = n_layer(mov_features)

        #使用余弦相似度算子，计算用户和电影的相似程度
        sim = F.cosine_similarity(user_features, mov_features,
axis=1).reshape([-1, 1])
        #对输入Tensor进行缩放和偏置，获得合适的输出指标
        predict = paddle.scale(sim,scale=5)

    return predict

```

损失函数

此处使用均方差损失函数。square_error_cost(input,label):接受输入预测值和目标值，并返回方差估计，即为 (y-y_predict) 的平方。

```

cost = F.square_error_cost(predict, paddle.cast(x=label_input,
dtype='float32'))
avg_cost = paddle.mean(cost)

```

超参配置

PaddleRec中模型超参的配置均体现在config.yaml文件中hyper_parameters模块，本示例超参配置如下所示，其中参数解释如下：

- class：优化器类型；
- learning_rate：学习率；
- sparse_feature_number：稀疏特征数量；
- sparse_feature_dim：稀疏特征维度；
- fc_sizes：全连接层的规模。

```
# hyper parameters of user-defined network
hyper_parameters:
  # optimizer config
  optimizer:
    class: Adam
    learning_rate: 0.001
  # user-defined <key, value> pairs
  sparse_feature_number: 600000
  sparse_feature_dim: 9
  fc_sizes: [512, 256, 128, 32]
```

在简单了解召回模型和其组网实现之后，我们来看下如何做到一键式启动训练。首先执行如下命令启动训练。我们在训练集上训练了五个epoch，在每个epoch后保存了训练出的模型参数文件。在config.yaml文件中的配置如下所示：

```
runner:
  train_data_dir: "../data/train" #训练数据的路径
  train_reader_path: "reader" # importlib format
  train_batch_size: 128
  model_save_path: "output_model_recall" #模型训练完后保存在该目录下

  use_gpu: true #是否使用gpu进行训练
  epochs: 5 #训练5个epoch
  print_interval: 20 #每隔20个batch输出一次指标

  test_data_dir: "../data/test"
  infer_reader_path: "reader" # importlib format
  infer_batch_size: 128
  infer_load_path: "output_model_recall"
  infer_start_epoch: 4 #模型从第五个epoch开始测试 (epoch 0为第一个epoch)
  infer_end_epoch: 5 #模型测试到第五个epoch为止。[infer_start_epoch,
infer_end_epoch)

  runner_result_dump_path: "recall_infer_result" #模型测试完后指标保存在该目录下
```

大家在使用带GPU的环境进行训练时，`use_gpu` 选项为true。若环境中无GPU硬件支持，需要用户自行在项目目录下分别将 `recall` 和 `rank` 目录中的 `config.yaml` 文件中将 `use_gpu` 选项改为False。

In [1]

```
# 动态图训练recall模型，使用gpu每轮训练耗时约1.5分钟，训练5轮
!cd PaddleRec/models/demo/movie_recommand && python -u
../tools/trainer.py -m recall/config.yaml
```

In [5]

```
# 我们对训练出的模型进行测试，目的是选择出效果最好的模型，由于我们只保存了5组模型参数，数量较少，所以我们仅使用最后一组模型参数进行测试。
# recall模型动态图测试（在数据集的测试集中执行测试）
!cd PaddleRec/models/demo/movie_recommand && python -u infer.py -m
recall/config.yaml
```

我们可以看到，测试过程中打印了很多信息，那么有没有直观的测试结果呢？大家可以找到测试结果解析脚本parse.py，我们可以执行该脚本对测试结果进行分析，以便确定训练出的模型效果。

In [6]

```
# 离线召回测试结果解析
!pip install py27hash
!echo "recall offline test result:"
!cd PaddleRec/models/demo/movie_recommand && python parse.py recall_offline
recall_infer_result
Looking in indexes: https://mirror.baidu.com/pypi/simple/
Requirement already satisfied: py27hash in /opt/conda/envs/python35-paddle120-
env/lib/python3.7/site-packages (1.0.2)
recall offline test result:
total: 6016; correct: 1835
accuracy: 0.30501994680851063
mae: 0.8757134295922752
```

从显示信息中我们可以看到模型的准确率（accuracy），以及MAE值，平均绝对误差，即绝对误差的平均值，用户可以根据这个值判断模型的效果。一般MAE越小则说明误差越小，代表模型效果越好。由于本教程仅是演示，所以训练并不充足，导致这两个评估值并不理想，因此在实际业务中，大家需要多训练几个epoch，以保证模型的效果。相应的，训练过程中也会保存更多的模型参数，一般建议大家选择最后保存几个模型进行测试，然后根据测试和分析的结果选出最优的模型。如果用户希望提高模型训练效果，可以看后面的优化策略部分。

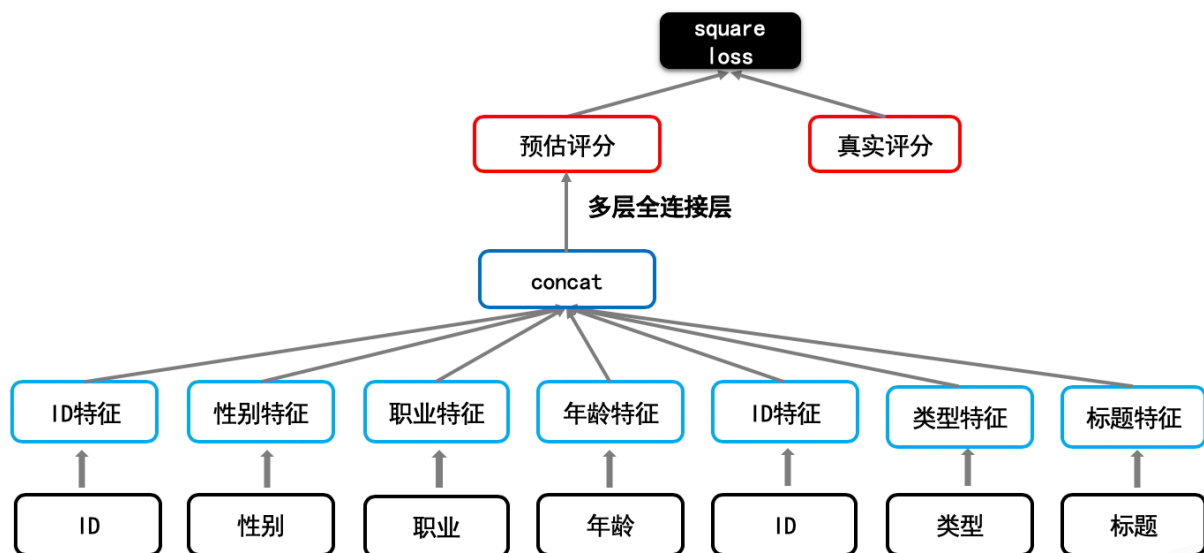
说明：由于训练轮数较少且存在收敛的随机性，模型效果会略有差异

排序模型

排序是指经过推理将用户感兴趣的内容按照点击率由高到低推荐给用户，在个性化推荐系统中决定了最终的推荐效果。

模型解析

排序的目的是引入更多特征，进行更加精细化的预估。一般特点是模型较复杂，候选输入集比较少。经典的是CTR-DNN和DeepFM等。此教程采用DNN模型，模型组网如下图所示。相较于召回模型的双塔结构，排序模型更早地完成了特征交叉，然后进一步利用DNN网络进行训练，旨在对候选集合进行更加精确的打分。



如果了解示例中排序模型的组网代码，可以参考脚本 `rank/net.py`。

输入部分

示例中的 `dygraph_model.py` 将使用如下代码读取数据

```
def create_feeds(batch):
    user_sparse_inputs = [
        paddle.to_tensor(batch[i].numpy().astype('int64').reshape(-1, 1))
        for i in range(4)
    ]

    mov_sparse_inputs = [
        paddle.to_tensor(batch[4].numpy().astype('int64').reshape(-1, 1)),
        paddle.to_tensor(batch[5].numpy().astype('int64').reshape(-1, 4)),
        paddle.to_tensor(batch[6].numpy().astype('int64').reshape(-1, 3))
    ]

    label_input =
    paddle.to_tensor(batch[7].numpy().astype('int64').reshape(-1, 1))

    return user_sparse_inputs, mov_sparse_inputs, label_input
```

模型组网

组网部分 `net.py` 的代码如下所示。

```
class DNNLayer(nn.Layer):
    #在使用动态图时，针对一些比较复杂的网络结构，可以使用Layer子类定义的方式来进行模型代码编写，在
    __init__ 构造函数中进行组网Layer的声明，
    #在forward中使用声明的Layer变量进行前向计算。子类组网方式也可以实现sublayer的复用，针对相同的
    layer可以在构造函数中一次性定义，在forward中多次调用
    def __init__(self, sparse_feature_number, sparse_feature_dim, fc_sizes):
```



```

super(DNNLayer, self).__init__()
self.sparse_feature_number = sparse_feature_number
self.sparse_feature_dim = sparse_feature_dim
self.fc_sizes = fc_sizes

#声明embedding层，建立emb表将数据映射为向量
self.embedding = paddle.nn.Embedding(
    self.sparse_feature_number,
    self.sparse_feature_dim,
    padding_idx=0,
    sparse=True,
    weight_attr=paddle.ParamAttr(
        name="SparseFeatFactors",
        initializer=paddle.nn.initializer.Uniform()))

#使用循环的方式创建全连接层，可以在超参数中通过一个数组确定使用几个全连接层以及每个全
连接层的神经元数量。
#本例中使用了4个全连接层，并在每个全连接层后增加了relu激活层。
sizes = [63] + self.fc_sizes + [1]
acts = ["relu" for _ in range(len(self.fc_sizes))] + ["sigmoid"]
self._layers = []
for i in range(len(self.fc_sizes)+1):
    linear = paddle.nn.Linear(
        in_features=sizes[i],
        out_features=sizes[i + 1],
        weight_attr=paddle.ParamAttr(
            initializer=paddle.nn.initializer.Normal(
                std=1.0 / math.sqrt(sizes[i]))))
    self.add_sublayer('linear_%d' % i, linear)
    self._layers.append(linear)
    if acts[i] == 'relu':
        act = paddle.nn.ReLU()
        self.add_sublayer('act_%d' % i, act)
        self._layers.append(act)
    if acts[i] == 'sigmoid':
        act = paddle.nn.layer.Sigmoid()
        self.add_sublayer('act_%d' % i, act)
        self._layers.append(act)

def forward(self, batch_size, user_sparse_inputs, mov_sparse_inputs,
label_input):

    #对用户特征建模， 所有用户sparse特征查对应的emb表，获得特征权重
    user_sparse_embed_seq = []
    for s_input in user_sparse_inputs:
        emb = self.embedding(s_input)
        emb = paddle.reshape(emb, shape=[-1, self.sparse_feature_dim])
        user_sparse_embed_seq.append(emb)

```

```

#对电影特征建模， 所有电影sparse特征查对应的emb表，获得特征权重
mov_sparse_embed_seq = []
for s_input in mov_sparse_inputs:
    s_input = paddle.reshape(s_input, shape = [batch_size, -1])
    emb = self.embedding(s_input)
    emb = paddle.sum(emb, axis = 1)
    emb = paddle.reshape(emb, shape=[-1, self.sparse_feature_dim])
    mov_sparse_embed_seq.append(emb)

#查表结果拼接在一起，混合用户特征和电影特征，相比召回模型，排序模型更早地完成了特征交叉
features = paddle.concat(user_sparse_embed_seq +
mov_sparse_embed_seq,axis=1)
#利用DNN网络进行训练，使用sigmoid激活的全连接层，旨在对候选集合进行更加精确的打分
for n_layer in self._layers:
    features = n_layer(features)
#对输入Tensor进行缩放和偏置，获得合适的输出指标
predict = paddle.scale(features,scale=5)

return predict

```

损失函数

此处使用均方差损失函数。square_error_cost(input,label):接受输入预测值和目标值，并返回方差估计，即为 (y-y_predict) 的平方。

```

cost = F.square_error_cost(predict,paddle.cast(x=label_input, dtype='float32'))
avg_cost = paddle.mean(cost)

```

超参配置

PaddleRec中模型超参的配置均体现在config.yaml文件中hyper_parameters模块，本示例超参配置如下所示，其中参数解释如下：

- class：优化器类型；
- learning_rate：学习率；
- sparse_feature_number：稀疏特征数量；
- sparse_feature_dim：稀疏特征维度；
- fc_sizes：全连接层的规模。

```
# hyper parameters of user-defined network
hyper_parameters:
  # optimizer config
  optimizer:
    class: Adam
    learning_rate: 0.001
  # user-defined <key, value> pairs
  sparse_feature_number: 600000
  sparse_feature_dim: 9
  fc_sizes: [512, 256, 128, 32]
```

和召回模型的操作方法相似，用户同样可以使用如下三个命令对排序模型完成训练、测试和结果分析，因此不再赘述。

In [7]

```
# 动态图训练rank模型 (在数据集的训练集中执行训练)
# 使用gpu每轮训练耗时约1.5分钟，训练5轮
!cd PaddleRec/models/demo/movie_recommand && python -u
../tools/trainer.py -m rank/config.yaml
```

In [8]

```
# rank模型动态图测试 (在数据集的测试集中执行测试)
!cd PaddleRec/models/demo/movie_recommand && python -u infer.py -m
rank/config.yaml
```

In [9]

```
# 离线排序测试结果解析
!echo "rank offline test result:"
!cd PaddleRec/models/demo/movie_recommand && python parse.py rank_offline
rank_infer_result
rank offline test result:
total: 6016; correct: 1754
accuracy: 0.2915558510638298
mae: 0.9075403101405298
```

从显示信息中我们可以看到本示例的排序模型也是使用准确率和MAE值作为评价指标。同样在实际业务中，用户需要优化下训练策略，以确保训练效果，具体优化方法，请您往下看！

说明：由于训练轮数较少且存在收敛的随机性，模型效果会略有差异。

优化策略

在本项目示例中，仅仅使用默认配置进行了五轮的训练，模型还称不上训练到最优，已经有了基本的效果，如何继续优化呢？

调整超参

在PaddleRec中，我们已经将超参数都写在config.yaml中，所以只需要对config.yaml一个文件进行修改，就能够清晰的对比模型效果，并快速进行模型效果验证，极大的提升模型的迭代效率。

增加训练轮数

在训练模型的时候，效果较差可能是因为欠拟合引起的。我们可以增加训练的轮数，让模型获得更充分的训练，以此来提高模型的效果。以本教程中recall/config.yaml为例：

```
runner:
  train_data_dir: "../data/train"
  train_reader_path: "reader" # importlib format
  train_batch_size: 128
  model_save_path: "output_model_recall"

  use_gpu: true
  epochs: 5
  print_interval: 20

  test_data_dir: "../data/test"
  infer_reader_path: "reader" # importlib format
  infer_batch_size: 128
  infer_load_path: "output_model_recall"
  infer_start_epoch: 4
  infer_end_epoch: 5

  runner_result_dump_path: "recall_infer_result"
```

改这个数即可

更换优化器

在训练模型的时候，我们可以更换优化器，尝试不同的学习率以求获得提升。在PaddleRec中，我们提供SGD/Adam/AdaGrad优化器供您尝试。也可以通过learning_rate选项修改学习率。仍然以本教程中recall/config.yaml为例：

```
# hyper parameters of user-defined network
hyper_parameters:
  # optimizer config
  optimizer:
    class: Adam
    learning_rate: 0.001
    strategy: async
```

改它换优化器

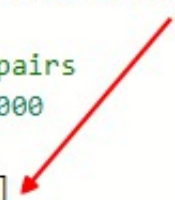
改它调整学习率

调整全连接层

在训练模型的时候，我们可以很方便的指定模型的全连接层共有几层，以及每一层的维度。仍然以本教程中recall/config.yaml为例：

```
hyper_parameters:
  # optimizer config
  optimizer:
    class: Adam
    learning_rate: 0.001
    strategy: async
  # user-defined <key, value> pairs
  sparse_feature_number: 60000000
  sparse_feature_dim: 9
  fc_sizes: [512, 256, 128, 32]
```

改数字的个数，调整有几个fc层
改数字的大小，调整每个fc层神经元的数量



增加和组合特征

模型基于输入的特征数据，学习到用户某些方面的特征，依据这些特征对用户做出推荐。如果我们能够给模型提供更多的更有效的特征，就可以大大降低模型的学习难度，提升模型的拟合能力。

增加特征

说起增加特征的数量，人们往往第一个想到的是引入新的数据。比如在电影特征中，我们收集更多的数据，添加电影的生产地区（国产电影/美国电影/欧洲电影/印度电影/日本电影...）。如果在用户的行为数据中发现这位用户经常看好莱坞大片，那么推荐美国电影受到好评的可能性就会大一些。再比如引入电影上映时间的特征（2020年上映/2019年上映/2018年上映...）。如果在用户的行为数据中发现这位用户喜欢追看新电影上映后第一时间观看，那么推荐他最新一年的电影，就有更大的可能会被接受。选择合适的特征添加入你的模型，往往事半功倍。

组合特征

收集新数据往往没有那么容易，在手头数据有限的情况下，我们可以将手头已有的数据组合一下，得到的一些组合起来的特征有时也能让模型突飞猛进。比如在用户特征中，我们将用户的年龄特征离散化，按照年龄阶段分层为（少年/青年/中年/老年），再与用户的职业特征组合起来，变成（少年学生/青年摄影师/中年医生/老年教授...）。组合多种特征，将用户划分为一个个有共同爱好的小群体。将小群体中大家都喜爱的电影推荐给群体中的其他人，往往也能受到好评。但是在组合过多特征时，也会带来特征数量爆炸以及每个特征样本太少导致过拟合的风险。具体使用的情况往往还要各位自行把握。

模型部署&Demo展示

模型的在线服务部署在一般是在后台运行的，因为在线服务需要启动一个长期稳定的后台进程提供服务。因此这部分的代码，大家一般是在linux服务器后台执行并运行服务。

启动服务

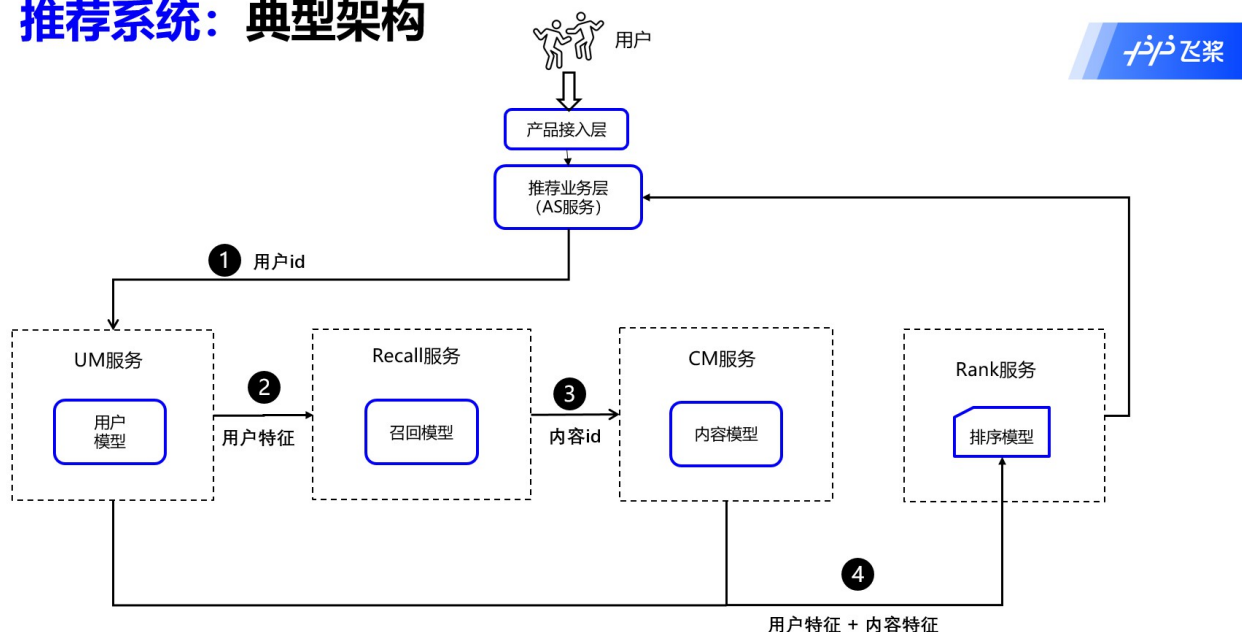
```
python3 -m pip install redis pymilvus==0.3.0 paddle_serving_app==0.2.0
sh get_data.sh #执行完后需要敲一下回车
sh start_server.sh #执行完后需要敲一下回车
```

模拟在线推荐

```
export PYTHONPATH=$PYTHONPATH:$PWD/proto
python test_client.py as M 32 5 # 性别, 年龄, 工作
```

这里我们给出了如下几种服务。

推荐系统：典型架构



本demo系统一共启动了5个在线服务，分别是用户模型服务，内容模型服务，召回服务，排序服务，还有应用服务（**application service**）。用户模型和内容模型分别是数据集当中的users.dat和movies.dat经过解析之后保存在redis当中，用户模型以user_id作为key，内容模型以movie_id作为key。用户模型服务和内容模型服务的逻辑分别是按照用户传入的key来寻找对应的value并封装成protobuf结果返回。本demo系统不仅支持已有的用户信息库，也支持用原有的用户内容训练的模型拟合新增用户，因此用户模型仅在使用user.dat中存在的用户信息查询，对于新增用户则不会使用该服务，直接在召回服务中产生用户向量。召回服务目前分为两个阶段，第一个阶段是通过用户模型得到用户向量，第二个阶段是把用户向量和milvus中的电影向量做近期搜索。排序服务是用PaddleRec训练好的CTR模型，用Paddle Serving启动来提供预测服务能力，用户传入一个用户信息和一组内容信息，接下来就能经过特征抽取和排序计算，求得最终的打分，按从高到低排序返回给用户。应用服务就是以上流程的串联，设计的流程是用户传入自己的用户信息（性别，年龄，工作），从召回服务中得到用户向量，近似查询movie列表，接下来查询到所有的内容模型信息，最终两个结合在排序服务中得到所有候选电影的从高到低的打分，最终还原成原始的电影信息返回给用户。

```
python test_client.py um 5 # 查询user-id 为5的用户信息
python test_client.py cm 5 # 查询movie-id 为5的电影信息
python test_client.py recall 5 # demo召回服务预测, user id=5
python test_client.py rank # demo排序服务预测, 由于rank服务参数较多, 如需定制可参考代码
```


输出

一键看到指定用户的推荐结果

```
python test_client.py as M 32 5
```

as服务在经过查询用户、召回、正排（查电影）、排序之后，根据分数降序，从大到小，把最适合该用户的电影信息返回回来，方便您一键直观的看到结果。结果中是由大到小排序的电影信息，每个电影信息包含了电影的id，电影名和影片类型。

```
error {
  code: 200
}
item_infos {
  movie_id: "2670"
  title: "Run Silent, Run Deep (1958)"
  genre: "War"
}
item_infos {
  movie_id: "3441"
  title: "Red Dawn (1984)"
  genre: "Action, War"
}
item_infos {
  movie_id: "71"
  title: "Fair Game (1995)"
  genre: "Action"
}
item_infos {
  movie_id: "3066"
  title: "Tora! Tora! Tora! (1970)"
  genre: "War"
}
item_infos {
  movie_id: "3449"
  title: "Good Mother, The (1988)"
  genre: "Drama"
}
...
```

查看每个步骤的结果

您也可以分步查看每一个环节后的输出

```
python test_client.py um 5 # 查询user-id 为5的用户信息
```

um服务用于查询用户的信息，您可以选择一名用户，通过该服务获得用户的id，性别，年龄，工作，邮政编码。其中error=200为HTTP状态码，200表示请求已成功，请求所希望的响应头或数据体将随此响应返回。出现此状态码是表示正常状态。示例结果如下：

```
error {
  code: 200
}
user_info {
  user_id: "5"
  gender: "M"
  age: 25
  job: "20"
  zipcode: "55455"
}
```

```
python test_client.py cm 3878 # 查询movie-id 为3878的电影信息
```

gcms服务用于查询电影信息，您可以选择一个电影的id，通过该服务获得电影的id，电影名和影片类型。示例结果如下：

```
error {
  code: 200
}
item_infos {
  movie_id: "3878"
  title: "X: The Unknown (1956)"
  genre: "Sci-Fi"
}
```

```
python test_client.py recall 5 # demo召回服务预测, user id=5
```

recall服务用于根据您选择的用户id召回排分前100名的电影，显示每一个电影的id和预估分值。示例结果如下：

```
error {
  code: 200
}
score_pairs {
  nid: "760"
  score: 1061.401123046875
}
score_pairs {
  nid: "1350"
  score: 1191.803955078125
}
score_pairs {
```

```
nid: "632"
score: 1253.180908203125
}
score_pairs {
  nid: "1258"
  score: 1276.9473876953125
}
score_pairs {
  nid: "3007"
  score: 1302.7178955078125
}
score_pairs {
  nid: "1051"
  score: 1344.7513427734375
}
...
```

python test_client.py rank # demo排序服务预测，由于rank服务参数较多，如需定制可参考代码。示例结果如下：

```
error {
  code: 200
}
score_pairs {
  nid: "1"
  score: 3.7467310428619385
}
```