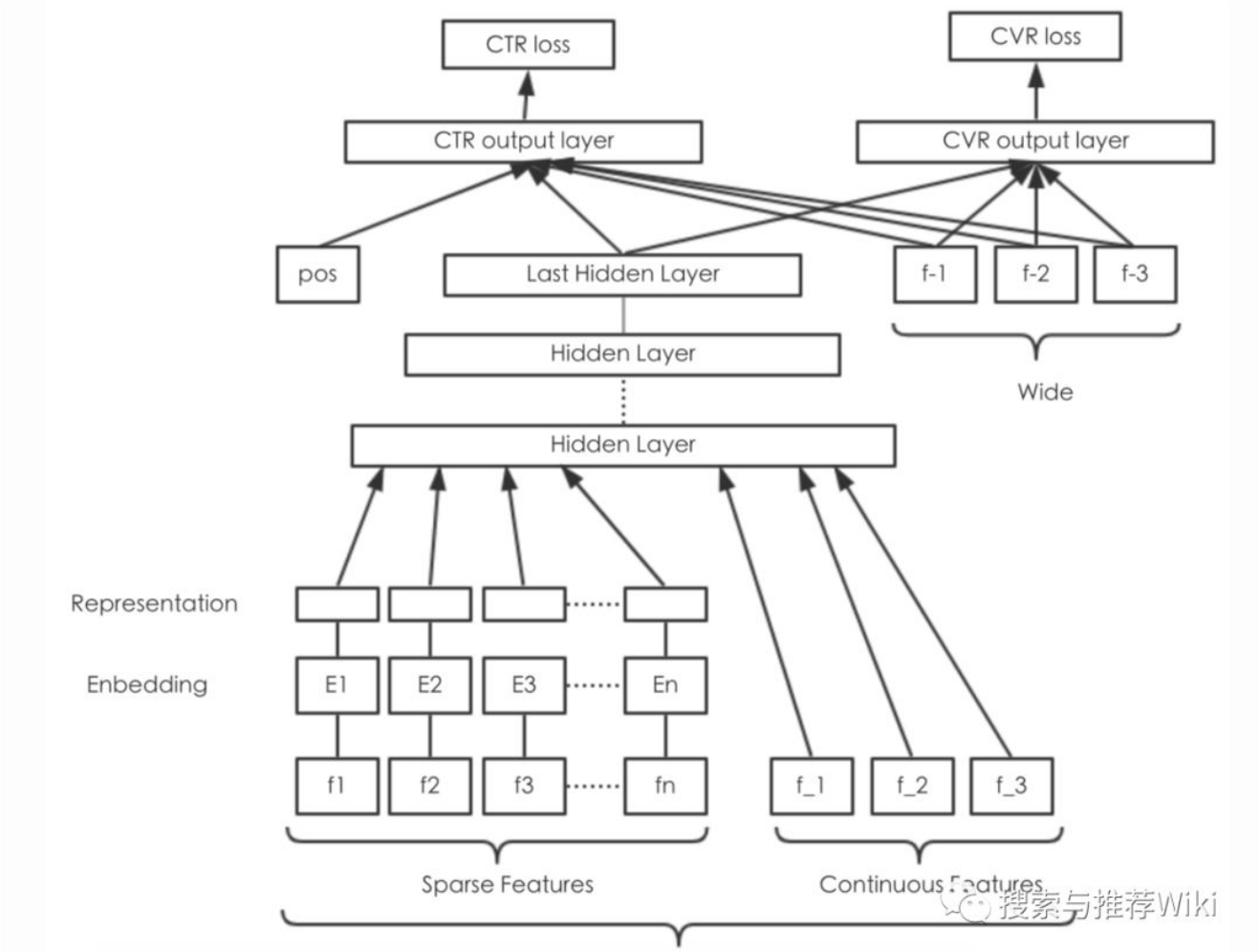
《文章推荐系统》系列之基于 Wide&Deep模型的在线排序

原创 小王子特洛伊 搜索与推荐Wiki 1月10日

收录于话题 #文章推荐系统实战 15个

文章推荐系统系列：

- 1、推荐流程设计
- 2、同步业务数据
- 3、收集用户行为数据
- 4、构建离线文章画像
- 5、计算文章相似度
- 6、构建离线用户画像
- 7、构建离线用户和文章特征
- 8、基于模型的离线召回
- 9、基于内容的离线和在线召回
- 10、基于热门文章和新文章的在线召回
- 11、基于LR的离线排序
- 12、基于FTRL模型的在线排序
- 13、基于Wide&Deep模型的在线排序



上图是 Wide&Deep 模型的网络结构，深度学习可以通过嵌入（Embedding）表达出更精准的用户兴趣及物品特征，不仅能减少人工特征工程的工作量，还能提高模型的泛化能力，使得用户行为预估更加准确。Wide&Deep 模型适合高维稀疏特征的推荐场景，兼有稀疏特征的可解释性和深模型的泛化能力。通常将类别特征做 Embedding 学习，再将 Embedding 稠密特征输入深模型中。Wide 部分的输入特征包括：类别特征和离散化的数值特征，Deep部分的输入特征包括：数值特征和 Embedding 后的类别特征。其中，Wide 部分使用 FTRL + L1；Deep 部分使用 AdaGrad，并且两侧是一起联合进行训练的。

离线训练

TensorFlow 实现了很多深度模型，其中就包括 Wide&Deep，API 接口为 `tf.estimator.DNNLinearCombinedClassifier` ，我们可以直接使用。在上篇文章中已经实现了将训练数据写入 TFRecord 文件，在这里可以直接读取

```
@staticmethod
def read_ctr_records():
    dataset = tf.data.TFRecordDataset(["./train_ctr_201905.tfrecords"])
    dataset = dataset.map(parse_tfrecords)
```

```
    dataset = dataset.shuffle(buffer_size=10000)
    dataset = dataset.repeat(10000)
    return dataset.make_one_shot_iterator().get_next()
```

解析每个样本，将 TFRecord 中序列化的 feature 列，解析成 channel_id (1), article_vector (100), user_weights (10), article_weights (10)

```
def parse_tfrecords(example):
    features = {
        "label": tf.FixedLenFeature([], tf.int64),
        "feature": tf.FixedLenFeature([], tf.string)
    }
    parsed_features = tf.parse_single_example(example, features)

    feature = tf.decode_raw(parsed_features['feature'], tf.float64)
    feature = tf.reshape(tf.cast(feature, tf.float32), [1, 121])
    # 特征顺序 1 channel_id,  100 article_vector, 10 user_weights, 10 article_weights
    # 1 channel_id类别型特征,  100维文章向量求平均值当连续特征，10维用户权重求平均值当连续特征
    channel_id = tf.cast(tf.slice(feature, [0, 0], [1, 1]), tf.int32)
    vector = tf.reduce_sum(tf.slice(feature, [0, 1], [1, 100]), axis=1, keep_dims=True)
    user_weights = tf.reduce_sum(tf.slice(feature, [0, 101], [1, 10]), axis=1, keep_dims=True)
    article_weights = tf.reduce_sum(tf.slice(feature, [0, 111], [1, 10]), axis=1, keep_dims=Tr

    label = tf.reshape(tf.cast(parsed_features['label'], tf.float32), [1, 1])

    # 构造字典 名称-tensor
    FEATURE_COLUMNS = ['channel_id', 'vector', 'user_weigths', 'article_weights']
    tensor_list = [channel_id, vector, user_weights, article_weights]

    feature_dict = dict(zip(FEATURE_COLUMNS, tensor_list))

    return feature_dict, label
```

指定输入特征的数据类型，并定义 Wide&Deep 模型 model

```
# 离散类型
channel_id = tf.feature_column.categorical_column_with_identity('channel_id', num_buckets=25)
# 连续类型
vector = tf.feature_column.numeric_column('vector')
user_weigths = tf.feature_column.numeric_column('user_weigths')
article_weights = tf.feature_column.numeric_column('article_weights')

wide_columns = [channel_id]

# embedding_column用来表示类别型的变量
deep_columns = [tf.feature_column.embedding_column(channel_id, dimension=25),
                    vector, user_weigths, article_weights]

estimator = tf.estimator.DNNLinearCombinedClassifier(model_dir="./ckpt/wide_and_deep",
```

```
                                                    linear_feature_columns=wide_c
                                                    dnn_feature_columns=deep_colu
                                                    dnn_hidden_units=[1024, 512,
```

通过调用 read_ctr_records() 方法，来读取 TFRecod 文件中的训练数据，并设置训练步长，对定义好的 FTRL 模型进行训练及预估

```
model.train(read_ctr_records, steps=1000)
result = model.evaluate(read_ctr_records)
```

可以用上一次模型的参数作为当前模型的初始化参数，训练完成后，通常会进行离线指标分析，若符合预期即可导出模型

```
columns = wide_columns + deep_columns
feature_spec = tf.feature_column.make_parse_example_spec(columns)
serving_input_receiver_fn = tf.estimator.export.build_parsing_serving_input_receiver_fn(feature_s
model.export_savedmodel("./serving_model/wdl/", serving_input_receiver_fn)
```

TFServing 部署

安装

```
docker pull tensorflow/serving
```

启动

```
docker run -p 8501:8501 -p 8500:8500 --mount type=bind,source=/root/toutiao_project/reco_sys/se
```

- -p 8501:8501 为端口映射（-p 主机端口：docker 容器程序）
- TFServing 使用 8501 端口对外提供 HTTP 服务，使用8500对外提供 gRPC 服务，这里同时开放了两个端口的使用
- --mount type=bind,source=/home/ubuntu/detectedmodel/wdl,target=/models/wdl 为文件映射，将主机（source）的模型文件映射到 docker 容器程序（target）的位置，以便 TFServing 使用模型，target 参数为 /models/模型名称
- -e MODEL_NAME= wdl 设置了一个环境变量，名为 MODEL_NAME，此变量被 TFServing 读取，用来按名字寻找模型，与上面 target 参数中的模型名称对应

- -t 为 TFServing 创建一个伪终端，供程序运行
- tensorflow/serving 为镜像名称

在线排序

通常在线排序是根据用户实时的推荐请求，对召回结果进行 CTR 预估，进而计算出排序结果并返回。我们需要根据召回结果构造测试集，其中每个测试样本包括用户特征和文章特征。首先，根据用户 ID 和频道 ID 读取用户特征（用户在每个频道的特征不同，所以是分频道存储的）

```
try:
    user_feature = eval(hbu.get_table_row('ctr_feature_user',
                                          '{}'.format(temp.user_id).encode(),
                                          'channel:{}'.format(temp.channel_id).encode()))
except Exception as e:
    user_feature = []
```

再根据用户 ID 读取召回结果

```
recall_set = read_hbase_recall('cb_recall',
                    'recall:user:{}'.format(temp.user_id).encode(),
                    'als:{}'.format(temp.channel_id).encode())
```

接着，遍历召回结果，获取文章特征，并将用户特征合并，构建样本

```
examples = []
for article_id in recall_set:
    try:
        article_feature = eval(hbu.get_table_row('ctr_feature_article',
                                                 '{}'.format(article_id).encode(),
                                                 'article:{}'.format(article_id).encode()))
    except Exception as e:
        article_feature = []

    if not article_feature:
        article_feature = [0.0] * 111

    channel_id = int(article_feature[0])
    # 计算后面若干向量的平均值
    vector = np.mean(article_feature[11:])
    # 用户权重特征
    user_feature = np.mean(user_feature)
    # 文章权重特征
    article_feature = np.mean(article_feature[1:11])
```

```
        # 构建example
        example = tf.train.Example(features=tf.train.Features(feature={
                        "channel_id": tf.train.Feature(int64_list=tf.train.Int64List(value=[channel_i
                        "vector": tf.train.Feature(float_list=tf.train.FloatList(value=[vector])),
                        'user_weigths': tf.train.Feature(float_list=tf.train.FloatList(value=[user_fe
                        'article_weights': tf.train.Feature(float_list=tf.train.FloatList(value=[arti
                    }))

        examples.append(example)
```

调用 TFServing 的模型服务，获取排序结果

```
with grpc.insecure_channel("127.0.0.1:8500") as channel:
        stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
        request = classification_pb2.ClassificationRequest()
        # 构造请求，指定模型名称，指定输入样本
        request.model_spec.name = 'wdl'
        request.input.example_list.examples.extend(examples)
        # 发送请求，获取排序结果
        response = stub.Classify(request, 10.0)
```

这样，我们就实现了 Wide&Deep 模型的离线训练和 TFServing 模型部署以及在线排序服务的调用。使用这种方式，线上服务需要将特征发送给TF Serving，这不可避免引入了网络 IO，给带宽和预估时延带来压力。可以通过并发请求，召回多个召回结果集合，然后并发请求 TF Serving 模型服务，这样可以有效降低整体预估时延。还可以通过特征 ID 化，将字符串类型的特征名哈希到 64 位整型空间，这样有效减少传输的数据量，降低使用的带宽。

模型同步

实际环境中，我们可能还要经常将离线训练好的模型同步到线上服务机器，大致同步过程如下：

- 同步前，检查模型 md5 文件，只有该文件更新了，才需要同步
- 同步时，随机链接 HTTPFS 机器并限制下载速度
- 同步后，校验模型文件 md5 值并备份旧模型

同步过程中，需要处理发生错误或者超时的情况，可以设定触发报警或重试机制。通常模型的同步时间都在分钟级别。