7.1 神经网络基础与原理

7.1.1 神经网络

分别输入层，输出层以及隐藏层

每层的圆圈代表一个神经元，隐藏层和输出层的神经元有输入的数据计算后输出，输入层的神经元只是输入。

- 每个连接都有个权值
- 同一层神经元之间没有连接
- 最后的输出结果对应的层也称之为**全连接层**

多个**树突**，主要用来接受传入信息

而**轴突**只有一条，轴突尾端有许多轴突末梢可以给其他多个神经元传递信息。

轴突末梢的输出

7.1.1.1 感知机(PLA: Perceptron Learning Algorithm))

感知机是一种最基础的分类模型，前半部分类似于回归模型

而逻辑回归用的sigmoid。**这个感知机具有连接的权重和偏置**

7.1.2 playground使用

- 二分类：感知机模型：找到合适的w1,w2b参数
- 感知机结构，能够很好去解决与、或等问题
    - 但是并不能很好的解决异或等问题

神经网络：多层的神经元组成，神经元==感知机模型

神经网络输出结果如何分类?

神经网络解决多分类问题最常用的方法是设置n个输出节点，其中n为类别的个数。

Softmax回归就是一个常用的方法

7.1.3 softmax回归

softmax：输出每个了别的概率，比较类别概率大小，选择其中一个概率大的作为输出类型

- 神经网络的组成： 神经元、softmax

么如何去衡量神经网络预测的概率分布和真实答案的概率分布之间的距离

7.1.4 交叉熵损失

- 用在模型输出的概率分布(多个分类结果)与镇纸质之前计算相关损失，
- 1log(0.1)损失，预测达到什么样，才能使得这个损失小
  - 1log(1.0)：表示预测为狗的为1.0，其它类别为0
- **提高对应目标值为1的位置输出概率大小**

7.1.4.2 损失大小

神经网络最后的损失为平均每个样本的损失大小。对所有样本的损失求和取其平均值

- N层，K个神经元

7.2 案例：DNN进行分类

- 知道tf.data.Dataset的API使用
- 知道tf.feature_columnAPI使用
- 知道tf.estimatorAPI使用

7.2.2 构建模型

1、 创建一个或多个输入函数

输入函数是返回 `tf.data.Dataset` 对象的函数，此对象会输出下列含有两个元素的元组：

```
features = {'SepalLength': np.array([6.4, 5.0]),
                'SepalWidth':  np.array([2.8, 2.3]),
                'PetalLength': np.array([5.6, 3.3]),
                'PetalWidth':  np.array([2.2, 1.0])}
    labels = np.array([2, 1])
```

特征必须是字典，列名为键，所有样本的该列特征为值

目标：数组，包含所有样本目标值

我们建议使用 TensorFlow 的 Dataset API，它可以解析各种数据。概括来讲，Dataset API 包含下列类：

- `Dataset` - 包含创建和转换数据集的方法的基类。您还可以通过该类从内存中的数据或 Python 生成器初始化数据集。
- `TextLineDataset` - 从文本文件中读取行。
- `TFRecordDataset` - 从 TFRecord 文件中读取记录。
- `FixedLengthRecordDataset` - 从二进制文件中读取具有固定大小的记录。
- `Iterator` - 提供一次访问一个数据集元素的方法。

```
dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))
```

dataset是一个Dataset类型，它包含了很多方法

- map
- shuffle:dataset.shuffle(1000)打乱内存大小
- repeat
  - repeat():不填参数时候，默认无限循环epoch迭代获取所有样本
  - repeat(10):所有样本迭代10个epoch
- batch(batch_size)：
  - 通过获取数据中每个元素，来达到batch_size大小集合

最终dataset就是我么你模型的数据输入

```
# 作为extimator训练模型的输入函数
train_input_fn
```

7.2.3 特征处理tf.feature_colum

数据的处理指定不是在dataset中去做，而是在estimator当中指定要怎么去处理

tf.feature_column:特征处理类型指定

- Numeric column（数值列）
- Bucketized column（分桶列）

```
# 首先，将原始输入转换为一个numeric column
numeric_feature_column = tf.feature_column.numeric_column("Year")

# 然后，按照边界[1960,1980,2000]将numeric column进行bucket
bucketized_feature_column = tf.feature_column.bucketized_column(
    source_column = numeric_feature_column,
    boundaries = [1960, 1980, 2000])
```

连续数据(数值型列)，做离散分桶处理，one_hot

- Categorical identity column（类别标识列）
- Categorical identity column（类别标识列）
  - 直接对数值的类型特征(1， 2，3，4，5，。。。25)

```
identity_feature_column =
tf.feature_column.categorical_column_with_identity(
    key='my_feature_b',
    num_buckets=4) # Values [0, 4)
```

- Categorical vocabulary column（类别词汇表）

```
vocabulary_feature_column =
    tf.feature_column.categorical_column_with_vocabulary_list(
        key=feature_name_from_input_fn,
        vocabulary_list=["kitchenware", "electronics", "sports"])
```

- Hashed Column（哈希列）

处理的示例都包含很少的类别。但当类别的数量特别大时，我们不可能为每个词汇或整数设置单独的类别，因为这将会消耗非常大的内存。对于此类情况，我们可以反问自己：“我愿意为我的输入设置多少类别?

```
hashed_feature_column =
    tf.feature_column.categorical_column_with_hash_bucket(
        key = "some_feature",
        hash_bucket_size = 100) # The number of categories
```

2、**定义模型的特征列，特征处理**

- 列特征必须指定类型

3、实例化 Estimator，指定特征列和各种超参数

输入特征处理是什么， **格式必须dataset**

格式，hidden_units, class

4、 在 Estimator 对象上调用一个或多个方法，传递适当的输入函数作为数据的来源

- 训练和测试数据的格式必须是dataset,通过一个input+fn来进行指定

```
classifier.train(
    input_fn=lambda:iris_data.train_input_fn(train_x, train_y,
args.batch_size),
    steps=args.train_steps)
```

array([14.298284, 8.176964, -16.61769], dtype=float32), 'probabilities': array([9.9780923e-01, 2.1907464e-03, 3.7360333e-14], dtype=float32), 'class_ids': array([0]), 'classes': array([b'0'], dtype=object)} {'logits': array([-6.0208535, 2.8362749, -2.9893901], dtype=float32), 'probabilities': array([1.4192433e-04, 9.9691641e-01, 2.9417418e-03], dtype=float32), 'class_ids': array([1]), 'classes': array([b'1'], dtype=object)} {'logits': array([-13.43511 , -0.8176402, 2.22374], dtype=float32), 'probabilities': array([1.5107146e-07, 4.5591064e-02, 9.5440876e-01], dtype=float32), 'class_ids': array([2]), 'classes': array([b'2'], dtype=object)}

解析出，每个样本预测0，1，2三个类别中哪一个，然后，类别的索引与字符串的一个关联是什么

预测的时候，拿到的只有特征值，返回的结果dataset必须(feature, label), label可以为固定值，

```
predictions = classifier.predict(
    input_fn=lambda: iris_data.eval_input_fn(predict_x,
                                             labels=None,
                                             batch_size=32))
```

8.3 深度学习与排序模型发展

LR

FM & FFM

FM可以看做带特征交叉的LR

DNN

模型切换到深度学习神经网络模型

加入深度结构，利用端到端的结构挖掘高阶非线性特征

- 通过改进模型结构，**加入深度结构，利用端到端的结构挖掘高阶非线性特征**，以及浅层模型无法捕捉的潜在模式。
- 对于某些ID类特别稀疏的特征，可以在模型中学习到保持分布关系的稠密表达（embedding）。
- 充分利用图片和文本等在简单模型中不好利用的信息。

Wide & Deep

Wide和Deep两个部分(LR+DNN的结合)

8.3.2 类别特征在推荐系统中作用

- 图像、NLP： 图片，文章(内容)
    - 非常稠密
- 推荐系统：很少有这么丰富的特征
    - 稀疏类别特征
- 大量ID特征
    - 优点：
        - 类别型特征有可能达到很好的效果，相比较numeric连续型
        - 用户、商品的标签画像系统，而标签大多数都是categorical的
        - 稀疏的类别/ID类特征，可以**稀疏地存储、传输、运算，提升运算效率**
    - 缺点：
        - 也有着**单个特征表达能力弱、特征组合爆炸、分布不均匀导致受训程度不均匀**的缺点
- FTRL在线学习：学习到稀疏模型
- TensorFlow Feature Column类
    - 特征：分桶，交叉，哈希
    - 表达能力更强

- 深度学习算法

5.8 排序模型进阶-FTRL

5.8.1 问题

- 训练数据过大，是spark支持不好

保存TFRecords，供TF去训练

TFRecords是不断增加，一个月保存一次最近的点击行为构造的训练样本

- 1.tfrecords
- 2.tfrecords
- 3.tfrecords
- .
- .
- .
- .
- .
- .

5.8.2 在线优化算法-Online-learning

模型迭代？？？？？

- 第一个月：1.tfrecords训练出**模型1**
- 第三个月：1.tfrecords2.tfrecords3.tfrecords**训练模型2**
  - 对1采用On-line-learning的算法。
- 模型参数少，预测的效果差；模型参数多线上predict的时候需要内存大
  - 对2采用一些优化的方法，在保证精度的前提下，尽量获取稀疏解，从而降低模型参数的数量
- TG(Truncated Gradient)
- FOBOS(Forward-Backward Splitting)
- RDA(Regularized Dual Averaging)
- **FTRL(Follow the Regularized Leader)**

SGD算法是常用的online learning算法，它能学习出不错的模型，但学出的模型不是稀疏的

- 不能立即反应用户的点击行为等各种行为带来的变化，SGD很难训练得到
  - 学习出有效的且**稀疏的模型**
- 用户：A，B，C, D
  - 1.2， 0.1， 1.5， 0.2
- 用户：A， B， F，E， G，D
  - 1， 0， 3.4， 1.0，2.1， 0
- 每个特征影响都比较小
- 稀疏：模型当中参数是有很多0

5.8.2 FTRL

- 一种获得稀疏模型的优化方法
- 加入正则化话：梯度下降时候，权重变化更快，较少的更快

5.8.3 离线数据训练FTRL模型

- 目的：通过离线TFRecords样本数据，训练FTRL模型
- 步骤：
  - 1、构建模型
  - 2、构建TFRecords的输入数据
    - estimator：要求特征必须是字典格式
    - kears:返回tensor
  - 3、train训练以及预测测试

8.5 排序模型进阶-Wide&Deep

8.5.1 wide&deep

- 输入
  - 两侧
- 优化：Wide部分用FTRL+L1来训练；Deep部分用AdaGrad来训练。

不会采取模型加载checkpoint或者H5这个格式的模型

3.2 线上预估

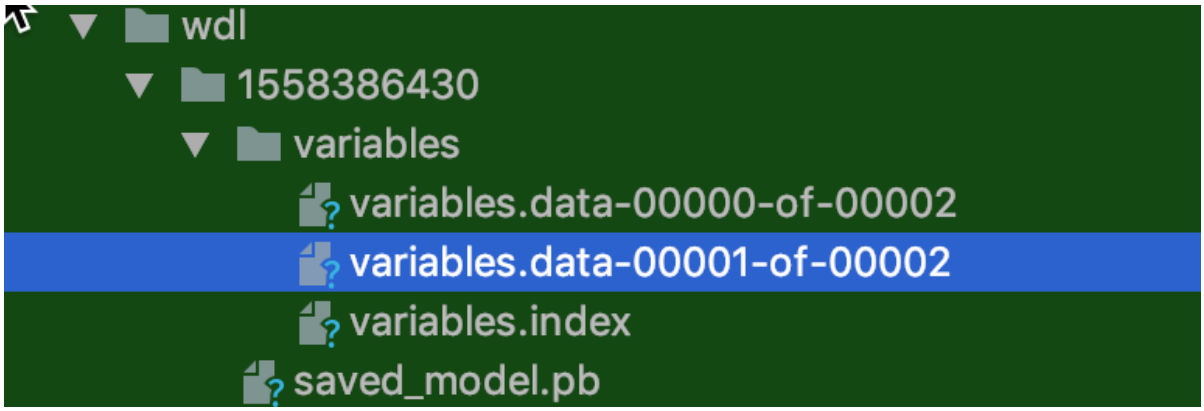
模型能不能快读上线切换，版本迭代2.0，3.0

基于TF Serving的模型服务

- 构造好样本特征
- 样本输入到Serving服务当中，进行预测，返回结果
- 结果处理返回排序的结果过滤，推荐给用户

1、构建好模型服务(wide&deep服务)

- 导出模型到固定类别的模型结构(不是checkpint, h5文件)
-



- TensorFlowServing不认同，不能加载
  - 时间戳保存不同模型版本
  - 好处：能够实时热更新模型

```
estimator.export_savedmodel("./serving_model/wdl/", serving_input_receiver_fn)


serving_input_receiver_fn:定义输入
tf.estimator.export.build_parsing_serving_input_receiver_fn(feature_spec)

feature_spec：可以是多个example
tf.feature_column.make_parse_example_spec(columns)
```

2、在线样本构造测试

7.7 TensorFlow Serving模型部署

7.7.1 TensorFlow Serving

7.7.1.1 安装Tensorflow Serving

7. 8 排序模型在线测试

- 输入模型必须是example

  ```
  tf.feature_column.make_parse_example_spec(columns)
  ```

- 构造样本测试
- 一个样本：

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

```
# 可以输入到模型当中
examples.append(example)
```

- 构造好样本：
- 调用TensorFlow Serving 服务
  - 需要下载这个包tensorflow-serving-api 1.13.0

```
from tensorflow_serving.apis import prediction_service_pb2_grpc
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import classification_pb2
import grpc
```

- 1、先建立连接通道
  - with grpc.insecure_channel('127.0.0.1:8500') as channel: stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
- 2、构造我们自己的请求
  - 指定模型名称和数据

```
# 获取测试数据集，并转换成 Example 实例
# 准备 RPC 请求，指定模型名称。
request = classification_pb2.ClassificationRequest()
request.model_spec.name = 'wdl'
request.input.example_list.examples.extend(examples)
```

发送请求，获得预测结果

获取结果response = stub.Classify(request, 10.0)print(response)