17 | Embedding+MLP：如何用TensorFlow实现经典的深度学习模型？

你好，我是王喆。

今天我们正式进入深度学习模型的实践环节，来一起学习并实现一种最经典的模型结构，Embedding+MLP。它不仅经典，还是我们后续实现其他深度学习模型的基础，所以你一定要掌握好。

这里面的 Embedding 我们已经很熟悉了，那什么叫做 MLP 呢？它其实是 Multilayer perceptron，多层感知机的缩写。感知机是神经元的另外一种叫法，所以多层感知机就是多层神经网络。

讲到这里啊，我想你脑海中已经有这个模型结构的大致图像了。今天，我就以微软著名的深度学习模型 Deep Crossing 为例，来给你详细讲一讲 Embedding+MLP 模型的结构和实现方法。

Embedding+MLP 模型的结构

图 1 展示的就是微软在 2016 年提出的深度学习模型 Deep Crossing，微软把它用于广告推荐这个业务场景上。它是一个经典的 Embedding+MLP 模型结构，我们可以看到，Deep Crossing 从下到上可以分为 5 层，分别是 Feature 层、Embedding 层、Stacking 层、MLP 层和 Scoring 层。

接下来，我就从下到上来给你讲讲每一层的功能是什么，以及它们的技术细节分别是什么样的。

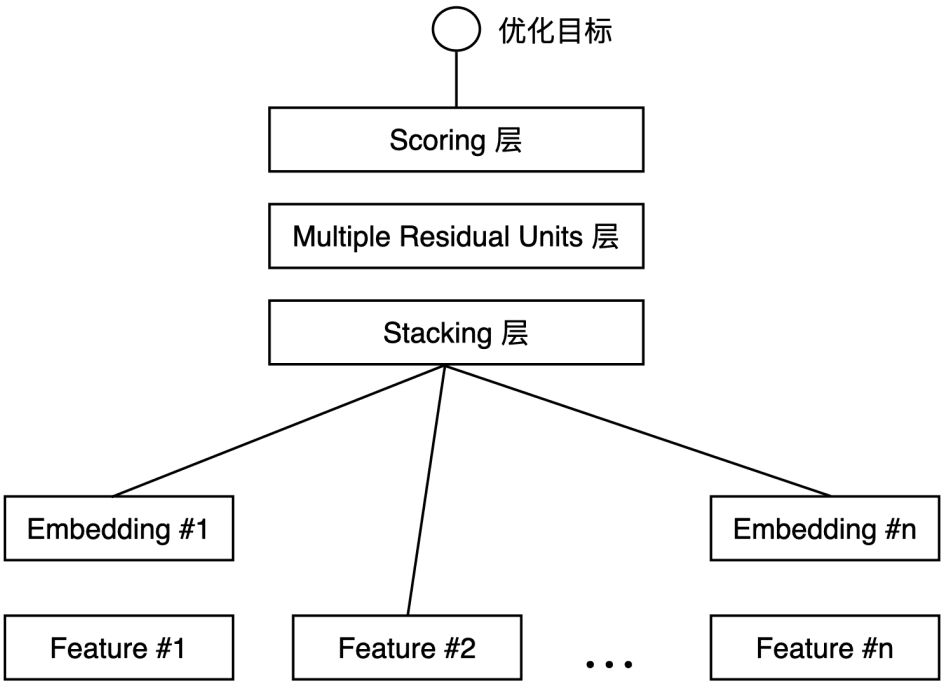

图1 经典的Embedding+MLP模型结构（图片来自 Deep Crossing - Web-Scale Modeling without Manually Crafted Combinatorial Features）

我们先来看 Feature 层。Feature 层也叫做输入特征层，它处于 Deep Crossing 的最底部，作为整个模型的输入。仔细看图 1 的话，你一定会发现不同特征在细节上的一些区别。比如 Feature#1 向上连接到了 Embedding 层，而 Feature#2 就直接连接到了更上方的 Stacking 层。这是怎么回事呢？

原因就在于 Feature#1 代表的是类别型特征经过 One-hot 编码后生成的特征向量，而 Feature#2 代表的是数值型特征。我们知道，One-hot 特征太稀疏了，不适合直接输入到后续的神经网络中进行训练，所以我们需要通过连接到 Embedding 层的方式，把这个稀疏的 One-hot 向量转换成比较稠密的 Embedding 向量。

接着，我们来看 Embedding 层。Embedding 层就是为了把稀疏的 One-hot 向量转换成稠密的 Embedding 向量而设置的，我们需要注意的是，Embedding 层并不是全部连接起来的，而是每一个特征对应一个 Embedding 层，不同 Embedding 层之间互不干涉。

那 Embedding 层的内部结构到底是什么样子的呢？我想先问问你，你还记得 Word2vec 的原理吗？Embeding 层的结构就是 Word2vec 模型中从输入神经元到隐层神经元的部分（如图 2 红框内的部分）。参照下面的示意图，我们可以看到，这部分就是一个从输入层到隐层之间的全连接网络。

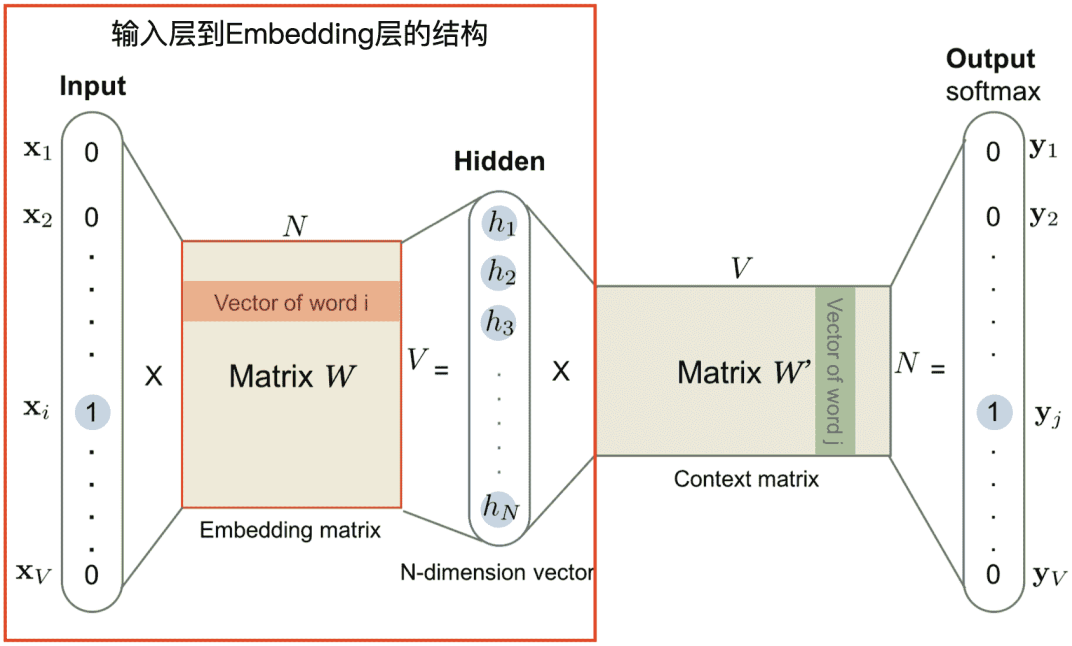

图2 Word2vec模型中Embedding层的部分

一般来说，Embedding 向量的维度应远小于原始的稀疏特征向量，按照经验，几十到上百维就能够满足需求，这样它才能够实现从稀疏特征向量到稠密特征向量的转换。

接着我们来看 Stacking 层。Stacking 层中文名是堆叠层，我们也经常叫它连接（Concatenate）层。它的作用比较简单，就是把不同的 Embedding 特征和数值型特征拼接在一起，形成新的包含全部特征的特征向量。

再往上看，MLP 层就是我们开头提到的多层神经网络层，在图 1 中指的是 Multiple Residual Units 层，中文叫多层残差网络。微软在实现 Deep Crossing 时针对特定的问题选择了残差神经元，但事实上，神经元的选择有非常多种，比如我们之前在深度学习基础知识中介绍的，以 Sigmoid 函数为激活函数的神经元，以及使用 tanh、ReLU 等其他激活函数的神经元。我们具体选择哪种是一个调参的问题，一般来说，ReLU 最经常使用在隐层神经元上，Sigmoid 则多使用在输出神经元，实践中也可以选择性地尝试其他神经元，根据效果作出最后的决定。

不过，不管选择哪种神经元，MLP 层的特点是全连接，就是不同层的神经元两两之间都有连接。就像图 3 中的两层神经网络一样，它们两两连接，只是连接的权重会在梯度反向传播的学习过程中发生改变。

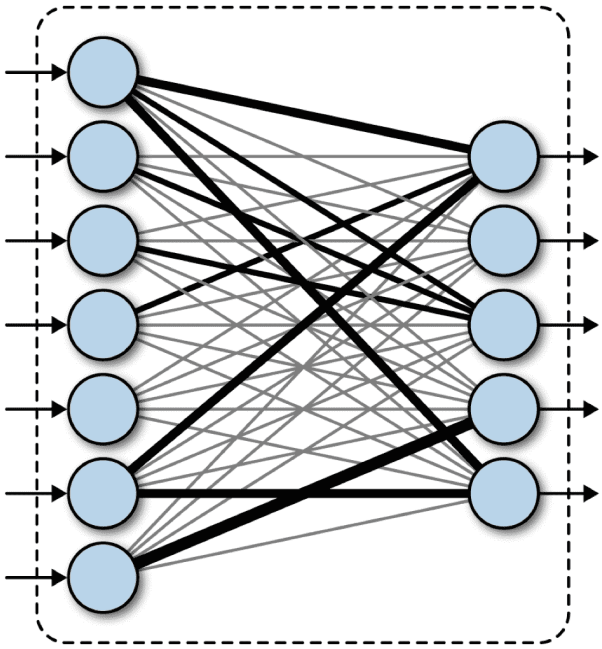

图3 全连接神经网络

MLP 层的作用是让特征向量不同维度之间做充分的交叉，让模型能够抓取到更多的非线性特征和组合特征的信息，这就使深度学习模型在表达能力上较传统机器学习模型大为增强。

最后是 Scoring 层，它也被称为输出层。虽然深度学习模型的结构可以非常复杂，但最终我们要预测的目标就是一个分类的概率。如果是点击率预估，就是一个二分类问题，那我们就可以采用逻辑回归作为输出层神经元，而如果是类似图像分类这样的多分类问题，我们往往在输出层采用 softmax 这样的多分类模型。

到这里，我们就讲完了 Embedding+MLP 的五层结构。它的结构重点用一句话总结就是，**对于类别特征，先利用 Embedding 层进行特征稠密化，再利用 Stacking 层连接其他特征，输入 MLP 的多层结构，最后用 Scoring 层预估结果。**

Embedding+MLP 模型的实战

现在，我们从整体上了解了 Embedding+MLP 模型的结构，也许你对其中的细节实现还有些疑问。别着急，下面我就带你用 SparrowRecsys 来实现一个 Embedding+MLP 的推荐模型，帮你扫清这些疑问。

实战中，我们按照构建推荐模型经典的步骤，特征选择、模型设计、模型实现、模型训练和模型评估这 5 步来进行实现。

首先，我们来看看特征选择和模型设计。

特征选择和模型设计

在上一节的实践准备课程中，我们已经为模型训练准备好了可用的训练样本和特征。秉着“类别型特征 Embedding 化，数值型特征直接输入 MLP”的原则，我们选择 movieId、userId、movieGenre、userGenre 作为 Embedding 化的特征，选择物品和用户的统计型特征作为直接输入 MLP 的数值型特征，具体的特征选择你可以看看下面的表格：

特征类型	特征	特征ID	处理方法
类别型	电影id	movieId	Onehot+Embedding化
	用户id	userId	Onehot+Embedding化
	电影风格类型	movieGenre	Onehot+Embedding化
	用户喜欢的风格类型	userGenre	Onehot+Embedding化
数值型	电影发布年份	releaseYear	直接输入MLP
	电影评分总数	movieRatingCount	直接输入MLP
	电影平均评分	movieAvgRating	直接输入MLP
	电影评分标准差	movieRatingStddev	直接输入MLP
	用户评分总数	userRatingCount	直接输入MLP
	用户平均评分	userAvgRating	直接输入MLP
	用户评分标准差	userRatingStddev	直接输入MLP
	用户好评电影的发布年份均值	userAvgReleaseYear	直接输入MLP
	用户好评电影的发布年份标准差	userReleaseYearStddev	直接输入MLP

选择好特征后，就是 MLP 部分的模型设计了。我们选择了一个三层的 MLP 结构，其中前两层是 128 维的全连接层。我们这里采用好评 / 差评标签作为样本标签，因此要解决的是一个类 CTR 预估的二分类问题，对于二分类问题，我们最后一层采用单个 sigmoid 神经元作为输出层就可以了。

当然了，我知道你肯定对这一步的细节实现有很多问题，比如为什么要选三层的 MLP 结构，为什么要选 sigmoid 作为激活函数等等。其实，我们对模型层数和每个层内维度的选择是一个超参数调优的问题，这里的选择不能保证最优，我们需要在实战中需要根据模型的效果进行超参数的搜索，找到最适合的模型参数。

Embedding+MLP 模型的 TensorFlow 实现

确定好了特征和模型结构，万事俱备，只欠实现了。下面，我们就看一看利用 TensorFlow 的 Keras 接口如何实现 Embedding+MLP 的结构。总的来说，TensorFlow 的实现有七个步骤。因为这是我们课程中第一个 TensorFlow 的实现，所以我会讲得详细一些。而且，我也把全部的参考代码放在了 SparrowRecsys 项目 TFRecModel 模块的 EmbeddingMLP.py，你可以结合它来听我下面的讲解。

我们先来看第一步，导入 TensorFlow 包。 如果你按照实战准备一的步骤配置好了 TensorFlow Python 环境，就可以成功地导入 Tensorflow 包。接下来，你要做的就是定义好训练数据的路径 TRAIN_DATA_URL 了，然后根据你自己训练数据的本地路径，替换参考代码中的路径就可以了。

```
import tensorflow as tf



TRAIN_DATA_URL = "file:///Users/zhewang/Workspace/
samples_file_path = tf.keras.utils.get_file("model
```

第二步是载入训练数据，我们利用 Tensorflow 自带的 CSV 数据集的接口载入训练数据。注意这里有两个比较重要的参数，一个是 label_name，它指定了 CSV 数据集中的标签列。另一个是 batch_size，它指定了训练过程中，一次输入几条训练数据进行梯度下降训练。载入训练数据之后，我们把它们分割成了测试集和训练集。

```
def get_dataset(file_path):
    dataset = tf.data.experimental.make_csv_datase
        file_path,
        batch_size=12
```

```
        batch_size=12,
        label_name='label',
        na_value="?",
        num_epochs=1,
        ignore_errors=True)
    return dataset



# sample dataset size 110830/12(batch_size) = 9235
raw_samples_data = get_dataset(samples_file_path)


test_dataset = raw_samples_data.take(1000)
train_dataset = raw_samples_data.skip(1000)
```

第三步是载入类别型特征。 我们用到的类别型特征主要有这三类，分别是 genre、userId 和 movieId。在载入 genre 类特征时，我们采用了 `tf.feature_column.categorical_column_with_vocabulary_list` 方法把字符串型的特征转换成了 One-hot 特征。在这个转换过程中我们需要用到一个词表，你可以看到我在开头就定义好了包含所有 genre 类别的词表 genre_vocab。

在转换 userId 和 movieId 特征时，我们又使用了 `tf.feature_column.categorical_column_with_identity` 方法把 ID 转换成 One-hot 特征，这个方法不用词表，它会直接把 ID 值对应的那个维度置为 1。比如，我们输入这个方法的 movieId 是 340，总的 movie 数量是 1001，使用这个方法，就会把这个 1001 维的 One-hot movieId 向量的第 340 维置为 1，剩余的维度都为 0。

为了把稀疏的 One-hot 特征转换成稠密的 Embedding 向量，我们还需要在 One-hot 特征外包裹一层 Embedding 层，你可以看到 `tf.feature_column.embedding_column(movie_col, 10)` 方法完成了这样的操作，它在把 movie one-hot 向量映射到了一个 10 维的 Embedding 层上。

```
genre_vocab = ['Film-Noir', 'Action', 'Adventure',
               'Sci-Fi', 'Drama', 'Thriller',
               'Crime', 'Fantasy', 'Animation', 'I


GENRE_FEATURES = {
    'userGenre1': genre_vocab,
    'userGenre2': genre_vocab,
    'userGenre3': genre_vocab,
    'userGenre4': genre_vocab,
    'userGenre5': genre_vocab,
    'movieGenre1': genre_vocab,
    'movieGenre2': genre_vocab,
    'movieGenre3': genre_vocab
}


categorical_columns = []
for feature, vocab in GENRE_FEATURES.items():
    cat_col = tf.feature_column.categorical_column
        key=feature, vocabulary_list=vocab)
    emb_col = tf.feature_column.embedding_column(c
    categorical_columns.append(emb_col)


movie_col = tf.feature_column.categorical_column_w
movie_emb_col = tf.feature_column.embedding_column
categorical_columns.append(movie_emb_col)
```

```
user_col = tf.feature_column.categorical_column_wi
user_emb_col = tf.feature_column.embedding_column(
categorical_columns.append(user_emb_c
```

第四步是数值型特征的处理。 这一步非常简单，我们直接把特征值输入到 MLP 内，然后把特征逐个声明为 `tf.feature_column.numeric_column` 就可以了，不需要经过其他的特殊处理。

```
numerical_columns = [tf.feature_column.numeric_col
                     tf.feature_column.numeric_colum
                       tf.feature_column.numeric_col
                       tf.feature_column.numeric_col
                       tf.feature_column.numeric_col
                       tf.feature_column.numeric_col
                       tf.feature_column.numeric_col
```

第五步是定义模型结构。 这一步的实现代码也非常简洁，我们直接利用 DenseFeatures 把类别型 Embedding 特征和数值型特征连接在一起形成稠密特征向量，然后依次经过两层 128 维的全连接层，最后通过 sigmoid 输出神经元产生最终预估值。

```
preprocessing_layer = tf.keras.layers.DenseFeature


model = tf.keras.Sequential([
    preprocessing_layer,
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
])
```

第六步是定义模型训练相关的参数。 在这一步中，我们需要设置模型的损失函数，梯度反向传播的优化方法，以及模型评估所用的指标。关于损失函数，我们使用的是二分类问题最常用的二分类交叉熵，优化方法使用的是深度学习中很流行的 adam，最后是评估指标，使用了准确度 accuracy 作为模型评估的指标。

```
model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy'])
```

第七步是模型的训练和评估。 TensorFlow 模型的训练过程和 Spark MLlib 一样，都是调用 fit 函数，然后使用 evaluate 函数在测试集上进行评估。不过，这里我们要注意一个参数 epochs，它代表了模型训练的轮数，一轮代表着使用所有训练数据训练一遍，epochs=10 代表着训练 10 遍。

```
model.fit(train_dataset, epochs=10)


test_loss, test_accuracy = model.evaluate(test_dat


print('\n\nTest Loss {}, Test Accuracy {}'.format(
```

如果一切顺利的话，你就可以看到模型的训练过程和最终的评估结果了。从下面的训练输出中你可以看到，每轮训练模型损失（loss）的变化过程和模型评估指标 accruacy 的变化过程。你肯定会发现，随着每轮训练的 loss 减小，accruacy 会变高。换句话说，每轮训练都

会让模型结果更好，这是我们期望看到的。需要注意的是，理论上来说，我们应该在模型 accuracy 不再变高时停止训练，据此来确定最佳的 epochs 取值。但如果模型收敛的时间确实过长，我们也可以设置一个 epochs 最大值，让模型提前终止训练。

```
Epoch 1/10
8236/8236 [==============================] - 20s 2
Epoch 2/10
8236/8236 [==============================] - 21s 3
Epoch 3/10
8236/8236 [==============================] - 21s 3
Epoch 4/10
8236/8236 [==============================] - 21s 2
Epoch 5/10
8236/8236 [==============================] - 21s 2
Epoch 6/10
8236/8236 [==============================] - 20s 2
Epoch 7/10
8236/8236 [==============================] - 20s 2
Epoch 8/10
8236/8236 [==============================] - 21s 2
Epoch 9/10
8236/8236 [==============================] - 20s 2
Epoch 10/10
8236/8236 [==============================] - 20s 2
1000/1000 [==============================] - 1s 1m




Test Loss 0.5036991238594055, Test Accuracy 0.7472
```

最终的模型评估需要在测试集上进行，从上面的输出中我们可以看到，最终的模型在测试集上的准确度是 0.7472，它意味着我们的模型对 74.72% 的测试样本作出了正确的预测。当然了，模型的评估指标还是很多，我们会在之后的模型评估篇中进行详细的讲解。

小结

这节课是我们深度学习模型实践的第一课，我们要掌握两个重点内容，一是 Embedding+MLP 的模型结构，二是 Embedding+MLP 模型的 TensorFlow 实现。

Embedding+MLP 主要是由 Embedding 部分和 MLP 部分这两部分组成，使用 Embedding 层是为了将类别型特征转换成 Embedding 向量，MLP 部分是通过多层神经网络拟合优化目标。具体来说，以微软的 Deep Crossing 为例，模型一共分为 5 层，从下到上分别是 Feature 层、Embedding 层、Stacking 层、MLP 层和 Scoring 层。

在 TensorFlow 实践部分，我们利用上节课处理好的特征和训练数据，实现了 SparrowRecsys 项目中的第一个深度学习模型。在实践过程中，我们要重点掌握类别型特征的处理方法，模型的定义方式和训练方式，以及最后的模型评估方法。

我也把这些重点知识总结在了一张表格里，你可以利用它来认真回顾。

理论知识	Embedding+MLP 的经典结构	输入特征层（Feature层） Embedding层（把类别型特征转换成Embedding） Stacking层（把所有特征连接起来） MLP层（多层神经网络） Scoring层（输出层）
TensorFlow 实践要点	类别特征转换成 One-hot向量	categorical_column_with_identity categorical_column_with_vocabulary_list
	One-hot向量转换 成Embedding	embedding_column
	创建连续性特征	tf.feature_column.numeric_column
	Stacking层	tf.keras.layers.DenseFeatures
	MLP层	tf.keras.Sequential([preprocessing_layer, tf.keras.layers.Dense(128, activation='relu'), tf.keras.layers.Dense(128, activation='relu'), tf.keras.layers.Dense(1, activation='sigmoid'),])
	模型训练关键参数	loss='binary_crossentropy' 定义损失函数 optimizer='adam' 定义反向传播优化方法 epochs=10 定义训练轮数

今天，我们一起完成了 Embedding MLP 模型的实现。在之后的课程中，我们会进一步实现其他深度学习模型，通过模型的评估进行效果上的对比。另外，我们也会利用训练出的深度学习模型完成 SparrowRecsys 的猜你喜欢功能，期待与你一起不断完善我们的项目。

课后思考

在我们实现的 Embedding+MLP 模型中，也有用户 Embedding 层和物品 Embedding 层。你觉得从这两个 Embedding 层中，抽取出来的用户和物品 Embedding，能直接用来计算用户和物品之间的相似度吗？为什么？

欢迎把你的思考和疑惑写在留言区，也欢迎你把这节课转发给希望用 TensorFlow 实现深度推荐模型的朋友，我们下节课见！