用NumPy手工打造 Wide & Deep

石塔西 Python爱好者社区 2019-01-17

作者： 石塔西
爱好机器学习算法，以及军事和历史
知乎ID：https://www.zhihu.com/people/si-ta-xi

自从我的《看Google如何实现Wide & Deep模型》发表之后，很多同学私信我，询问我的Wide & Deep实现的源代码。其实我的实现在扩展性、易用性上肯定不能和TensorFlow自带的实现相比，但是又充斥着一些业务细节。这些业务细节，技术本身没有什么难度，但又很敏感，清理起来很麻烦，所以我的那个TensorFlow实现暂时没有开源的计划了。

近年来，深度学习在推荐领域的应用得到了越来越多的关注，一系列新的算法，各种NN，各种FM，纷至而来，让人目不暇接，眼花缭乱。但是，在推荐领域经历了几年的摸爬滚打之后，我却开始了“返璞归真”：

一来，各种NN与FM，看似繁杂。实际上，只要把握住它们的发展脉络，即**“如何兼顾记忆与扩展”、“如何处理高维、稀疏的类别特征”、“如何实现特征交叉”**（见《看Google如何实现Wide & Deep模型》），你就会发现各种高大上的新算法不过是沿着这条脉络，在某个枝叉上的修补。这样一来，各种NN与FM，在你脑中，就不再是一个个独立的缩写，而能够编织成网，融会贯通。

二来，与其追读每篇新论文，调用作者提供的开源实现，每个模型都“走马观花”。不如找一个经典模型，自己从头到尾实现一遍，才理解得更加通透。

在我看来，已经不算新的Wide & Deep（WDL）就是这样一个经典模型，在“如何兼顾记忆与扩展”、“如何处理高维、稀疏的类别特征”、“如何实现特征交叉”三个方面，表现得很充分。为此，在上周，我花了一个星期的业余时间，用NumPy将Wide & Deep从头到尾实现了一遍，重温了算法的各种技术细节，受益匪浅。

https://github.com/stasi009/NumpyWDL

尽管在细节上还有待完善，我将它开源出来，希望和感兴趣的同学共同探讨。因为没有文档，在这里小撰一文，希望帮感兴趣的同学，理解我的代码。

我的实现基本模仿了tf.estimator.DNNLinearCombinedClassifier的结构。在手工实现Wide & Deep的过程中，我主要考虑如下三个技术关键点：

- 模块化设计
- Embedding的稀疏实现
- Embedding的权重共享

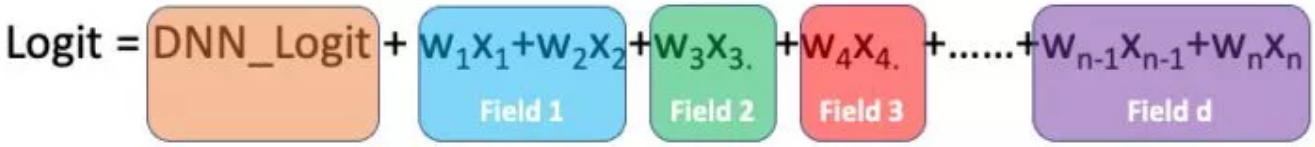
模块化设计

首先区分几个概念。比如，我们从“活跃App” + “新安装App” + “卸载App” 三个方面来描述一个用户的手机使用习惯。而每个方面可以用“微信:0.9，微博:0.5，淘宝:0.3，……”这样的ke-value-pair来表示

- “活跃App”, “新安装App”, “卸载App”被称为三个**Field**
- “活跃 微信", "安装 微博", "卸载 淘宝”被称为**Feature**，分别隶属于某个Field。在经过字典映射后，每个Feature都有自己的feature id（整数）和feature value（浮点数）
- “微信", "微博", "淘宝”都来自一个叫“App”的**Vocabulary**。在以上例子中，App Vocabulary为“活跃App”, “新安装App”, “卸载App”三个Field所共享

WDL在最上层其实就是一个LR模型，probability=sigmoid(logit)，而

$$Logit = Logit_{deep} + Logit_{wide} = Logit_{deep} + \sum_{i=1}^{d} Logit_{field_i}$$



WDL的最上层

为此，总体上我的WDL由如下几个部分实现

- DNN部分由dnn.py中的DeepNetwork实现
- Wide部分由wide_layer.py中的WideLayer实现
- WideLayer为每个field生成FtrlEstimator实例，负责用FTRL算法优化这个field下feature的权重

因为Wide主要功能是“记忆”，所以常接入一些ID类的特征，非常稀疏，所以需要使用FTRL算法来优化，以充分利用数据的稀疏性，并使得到的权重尽可能稀疏。FTRL的实现就是按照经典论文《Ad Click Prediction: a View from the Trenches》中Algorithm 1实现的。值得一提的就

是，与我之前见过的一些实现不同，**我的FTRL实现没有将所有特征放置在同一个特征空间中并统一编号，而是按照Field划分特征空间，每个Field单独存储、优化权重。**这样做有三个好处：

- 代码清晰、易读
- 方便扩展。比如某个Field下新增/删除了一个Feature，只有这个Field下的Feature需要重新编号，其他Field不受影响。
- 各个Field之间可以并行计算

而这种"**将每个Field单独划分成一个模块**"的做法，也是TensorFlow实现Wide侧的手法。（见《看Google如何实现Wide & Deep模型(3)》）https://zhuanlan.zhihu.com/p/48251812

讲完了Wide侧各Field的模块化实现，还要考虑Deep侧与Wide侧两个模块是如何设计的。设计主要考虑的是代码复用，**同样的Deep侧与Wide侧代码，既能合起来实现Wide & Deep，也能够单独使用来实现DNN与LR。**

但是，有一个问题是，DNN是基于Mini-Batch优化的，而Wide侧使用的FTRL是一个Online Learning算法。Wide侧得到某个样本的Wide_Logit之后，需要与Deep侧得到的Deep_Logit相加，得到总的Logit之后，才能计算梯度，才能更新权重。

我的方法是让外界传入一个proba_fn函数来根据logit计算概率。视Wide单独使用还是与Deep联合使用，proba_fn实现如下两种逻辑

- 当Wide侧单独使用来实现LR时，probability=sigmoid(logit)
- 在Wide & Deep中，
  - Deep侧先完成前代，得到这个batch下所有样本的Deep_Logits。
  - Wide侧在逐一学习每个样本时，先得到这条样本的Wide_Logit，再去已经计算好的Deep_Logits中找到这条样本的Deep_Logit，probability=sigmoid(wide_logit+deep_logit)
  - 再计算梯度，开始回代。

这部分逻辑见WideDeepEstimator中的_predict_proba与train_batch两个函数。

```
class WideDeepEstimator(BaseEstimator):
    def __init__(self, wide_hparams, deep_hparams, data_source):
        self._current_deep_logits = None

        self._wide_layer = WideLayer(......,
                                     proba_fn=self._predict_proba)

        self._dnn = DeepNetwork(......)
```

```
        super().__init__(data_source)

    def _predict_proba(self, example_idx, wide_logit):
        deep_logit = self._current_deep_logits[example_idx]
        logit = deep_logit + wide_logit
        return 1 / (1 + np.exp(-logit))

    def train_batch(self, features, labels):
        self._current_deep_logits = self._dnn.forward(features)

        pred_probas = self._wide_layer.train(features, labels)

        self._dnn.backward(grads2logits=pred_probas - labels)

        return pred_probas
```

Embedding的稀疏实现

正如我之前所论述的，深度学习在推荐、搜索领域的运用，是围绕着稀疏的ID类特征所展开的，其主要方法就是Embedding，变ID类特征的“精确匹配”为“模糊查找”，以增强扩展。而在实现Embedding时，需要注意两点

- 与传统MLP接收稠密输入不同，Embedding的输入高维且稀疏，One/Multi-Hot-Encoding之后进行矩阵运算代价太大，所以需要实现**稀疏的前代与回代**。
- 推荐系统中的Embedding与NLP中的Embedding也有不同。
  - NLP中，一句话的一个位置上只有一个词，所以Embedding往往变成了，从Embedding矩阵抽取与词对应的行上的行向量
  - 推荐系统中，一个Field下往往有多个Feature，Embedding是将多个Feature Embedding合并成一个向量，即所谓的**Pooling**。比如某个App Field下的Feature有"微信:0.9，微博:0.5，淘宝:0.3"，Embedding=0.9*微信向量+0.5*微博向量+0.3*淘宝向量

如何表示稀疏输入，很费了一番思考。

- 一开始想模仿TensorFlow，用sp_ids, sp_weights两上SparseTensor来表示，但是这两个SparseTensor中的indices, dense_shape必须完全相同，是重复的。既浪费空间，而且重复的东西就会带来“不一致”的隐患。
- 后来考虑使用KVPair = namedtuple('KVPair', ['example_index', 'feature_id', 'feature_value'])表示一个非零特征。整个稀疏输入就是list of KVPair，程序处理上是方便

了很多，但是每个KVPair都是一个namedtuple，生成了大多的small object，会给GC造成压力。

- 目前决定采用3个list的方式来表示稀疏输入
  - example_indices: 是[n_non_zeros]的整数数组，表示样本在batch中的序号。而且要求其中的数值是从小到大排好序的
  - feature_ids: 是[n_non_zeros]的整数数组，表示非零特征的序号，**可以重复**
  - feature_values: 是[n_non_zeros]的浮点数组，表示非零特征的数值

基于以上稀疏输入的表示，Embedding的实现，见embedding_layer.EmbeddingLayer这个类。可见**无论前代与回代，只有原始输入中的非零特征参与计算。**

```
class EmbeddingLayer:
    def __init__(self, W, vocab_name, field_name):
        """        :param W: dense weight matrix, [vocab_size,embed_size]
        self.vocab_name = vocab_name
        self.field_name = field_name
        self._W = W
        self._last_input = None

    def forward(self, X):
        """        :param X: SparseInput        :return: [batch_size, embed_
        self._last_input = X

        # output: [batch_size, embed_size]
        output = np.zeros((X.n_total_examples, self._W.shape[1]))

        for example_idx, feat_id, feat_val in X.iterate_non_zeros():
            embedding = self._W[feat_id, :]
            output[example_idx, :] += embedding * feat_val

        return output

    def backward(self, prev_grads):
        """        :param prev_grads: [batch_size, embed_size]        :retur
        dW = {}

        for example_idx, feat_id, feat_val in self._last_input.iterate_non_z
            # [1,embed_size]
            grad_from_one_example = prev_grads[example_idx, :] * feat_val

            if feat_id in dW:
                dW[feat_id] += grad_from_one_example
```

```
            else:
                dW[feat_id] = grad_from_one_example

        return dW
```

在利用计算好的导数对权重进行修正时，对**Embedding矩阵的梯度进行特殊处理，只更新局部**，见optimization.py中Adagrad.update函数。

```
class Adagrad:
    def __init__(self, lr):
        self._lr = lr
        # variable name => sum of gradient square (also a vector)
        self._sum_grad2 = {}

    def update(self, variables, gradients):
        for gradname, gradient in gradients.items():
            # ------ update cache
            g2 = gradient * gradient
            if gradname in self._sum_grad2:
                self._sum_grad2[gradname] += g2
            else:
                self._sum_grad2[gradname] = g2

            # ------ calculate delta
            delta = self._lr * gradient / (np.sqrt(self._sum_grad2[gradname]

            # ------ update
            if '@' in gradname:
                # 对应着稀疏输入的权重与梯度，gradients中的key遵循着'vocab_name@f
                varname, row = gradname.split('@')
                row = int(row)

                variable = variables[varname]
                variable[row, :] -= delta
            else:
                variable = variables[gradname]
                variable -= delta
```

Embedding的权重共享

如前所述，多个Field可能共享一个Vocabulary，所以要求在实现Embedding时也必须支持这一共享机制。否则，既可能浪费内存，又可能因为各Field的稀疏性不一致而导致训练不充分。
https://zhuanlan.zhihu.com/p/48057256

为此，我设计了一个EmbeddingCombineLayer类。

- 这个类先将所有要用到的"字典"的Embedding矩阵初始化，
- 再将每个Field与其对应的"字典"的Embedding矩阵联系起来。
- 只需要将多个field指向同一个vocabulary name，就可以让这个vocabulary的Embedding为多个field所共享。

```
class EmbeddingCombineLayer:
    def __init__(self, vocab_infos):
        """        :param vocab_infos: a list of tuple, each tuple is (vocab
        self._weights = {}  # vocab_name ==> weight
        for vocab_name, vocab_size, embed_size in vocab_infos:
            stddev = 1 / np.sqrt(embed_size)
            initializer = TruncatedNormal(mean=0,stddev=stddev,lower=-2 * st
            self._weights[vocab_name] = initializer(shape=[vocab_size, embed

    def add_embedding(self, vocab_name, field_name):
        weight = self._weights[vocab_name]
        layer = EmbeddingLayer(W=weight, vocab_name=vocab_name, field_name=f
        self._embed_layers.append(layer)
```

关键在于回代时，上层传入的"Loss对本层输出的导数"是[batch_size，本层所有embedding size之和]。在EmbeddingCombineLayer.backward中，

- 需要将以上梯度拆解，交给每个Field的Embedding层自己去回代。
- 最后还要聚合梯度，比如**"活跃App"中对"微信"有梯度，"新安装App"对"微信"也有梯度，最终"微信"embedding向量的梯度应该是以上二者之和。**

```
    def backward(self, prev_grads):
        """        :param prev_grads:  [batch_size, sum of all embed-layer's
        assert prev_grads.shape[1] == self.output_dim

        # 因为output是每列输出的拼接，自然上一层输入的导数也是各层所需要导数的拼接
        # prev_grads_splits是一个数组，存储对应各层的导数
        col_sizes = [layer.output_dim for layer in self._embed_layers]
        prev_grads_splits = utils.split_column(prev_grads, col_sizes)

        self._grads_to_embed.clear()  # reset
        for layer, layer_prev_grads in zip(self._embed_layers, prev_grads_sp
```

```
        # layer_prev_grads: 上一层传入的，Loss对某个Layer的输出的梯度
        # layer_grads_to_feat_embed: dict, feat_id==>grads，
        # 由这一个Layer造成对某vocab的embedding矩阵的某feat_id对应行的梯度
        layer_grads_to_embed = layer.backward(layer_prev_grads)

        for feat_id, g in layer_grads_to_embed.items():
            # 表示"对某个vocab的embedding weight中的第feat_id行的总导数"
            key = "{}@{}".format(layer.vocab_name, feat_id)

            if key in self._grads_to_embed:
                self._grads_to_embed[key] += g
            else:
                self._grads_to_embed[key] = g
```

测试效果

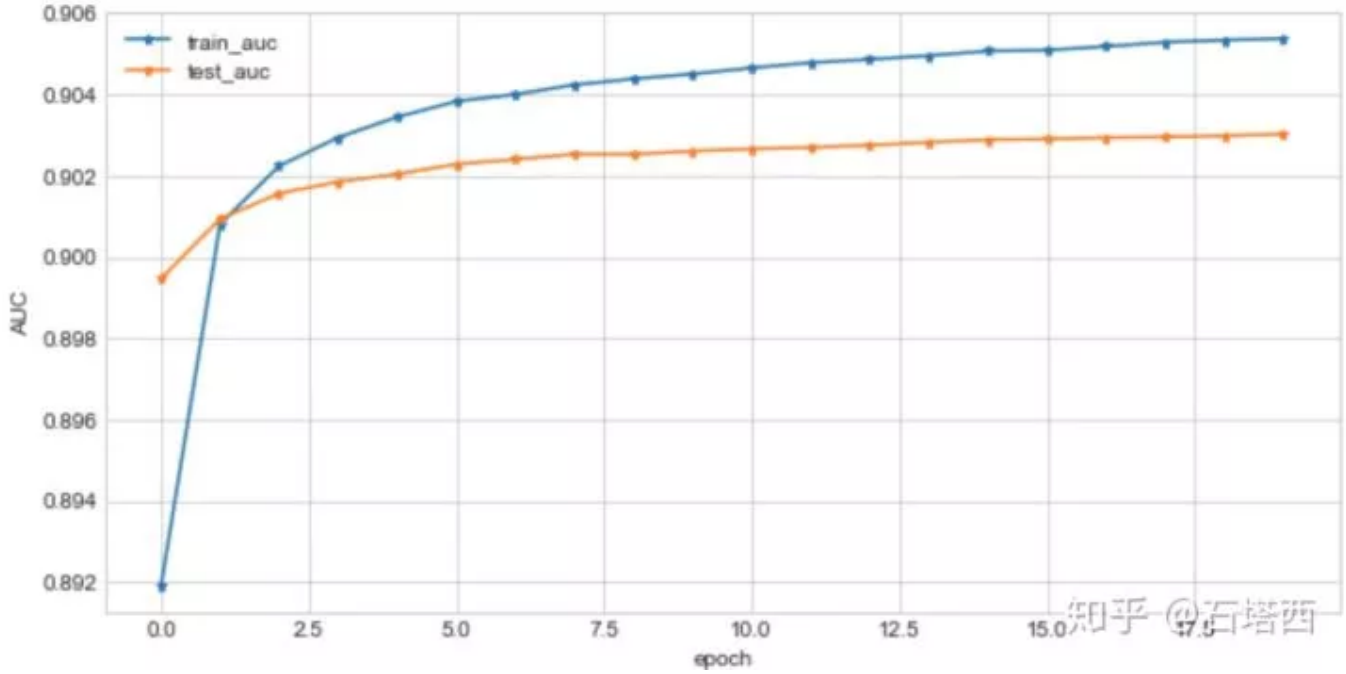
和TensorFlow Guide一样，在Census Income Data Set数据集上进行了测试。测试结果如下

- 性能指标上，Wide & Deep > Deep > Wide，符合我们的预期
- TensorFlow Guide上的基线准确率是0.83，而我的实现中每个模型都超过基线，可以从一个侧面反映我的实现的正确性。

	test_accuracy	test_auc	train_accuracy	train_auc
wide	0.838155	0.886093	0.836829	0.886863
deep	0.850132	0.901113	0.848100	0.903089
wide & deep	0.851053	0.903029	0.8509[illegible]	0.90[illegible]3

三种算法的性能对比

Wide & Deep模型训练时AUC曲线如下所示



而且，在我的笔记本上跑我的代码，每秒能够处理10000上下的样本，说明效率上也还不错

后记

本文简单介绍了我用NumPy手工实现的Wide & Deep模型，重点介绍了如下技术关键点：

- 如何模块化设计
- 如何实现Embedding层稀疏地前代与回代
- 如何实现Embedding层的权重共享

毕竟是我个人业余时间的练习作品，时间仓促，还有很多地方需要改进、完善：

- 实现Dropout与Batch Normalization。不过，Dropout源于Computer Vision，其输入都是稠密的图像，与推荐、搜索领域稀疏的输入，有很大不同。根据Google与Airbnb的经验，Dropout应用于推荐任务，不仅不会提升，反而会恶化性能。
- 实现更多的经典优化算法，比如Momentum, RMSprop, Adam等算法。
- 对比TensorFlow的实现，我没有实现众多的Feature Column。其实，Feature Column对我们的重要性一点也不亚于DNNLinearCombinedClassifier。有时间，我一定补上。实现各种Feature Column在技术上没有什么难度，就是个“力气活”。
- 如前所述，Wide & Deep本质上就是一个LR，而且Deep侧贡献的logit、各Field贡献的logit相互解耦。因此，可以考虑使它们的前代与回代并行化，实现Feature Parallelism。
- 我的上一篇文章《走马观花Google TF-Ranking的源代码》觉得TF-Ranking不太好用。现在，既然我已经实现了Wide & Deep，稍加改动，就能够将Wide & Deep与Learning To Rank结合，实现pairwise/listwise的排序算法。

通过从头到尾实现一遍Wide & Deep，我进一步加深了对推荐系统中的深度学习算法的理解，受益匪浅。欢迎感兴趣的同学下载我的代码，欢迎同道中人共同探讨。

感谢浏览

Python的爱好者社区历史文章大合集：

Python的爱好者社区历史文章列表

福利：文末扫码关注公众号，**“Python爱好者社区”**，开始学习Python课程：

关注后在公众号内回复**“ 课程 ”**即可获取：

小编的转行入职数据科学（数据分析挖掘/机器学习方向）【最新免费】

小编的Python的入门免费视频课程！

小编的Python的快速上手matplotlib可视化库！

崔老师爬虫实战案例免费学习视频。

陈老师数据分析报告扩展制作免费学习视频。

玩转大数据分析！Spark2.X + Python精华实战课程免费学习视频。