AiCon
全球人工智能与机器学习技术大会

借助TensorFlow在CTR预估中快速落地DNN

李珂
vivo AI团队负责人

主办方 Geekbang. 极客邦科技 InfoQ

TABLE OF CONTENTES

广告CTR系统架构

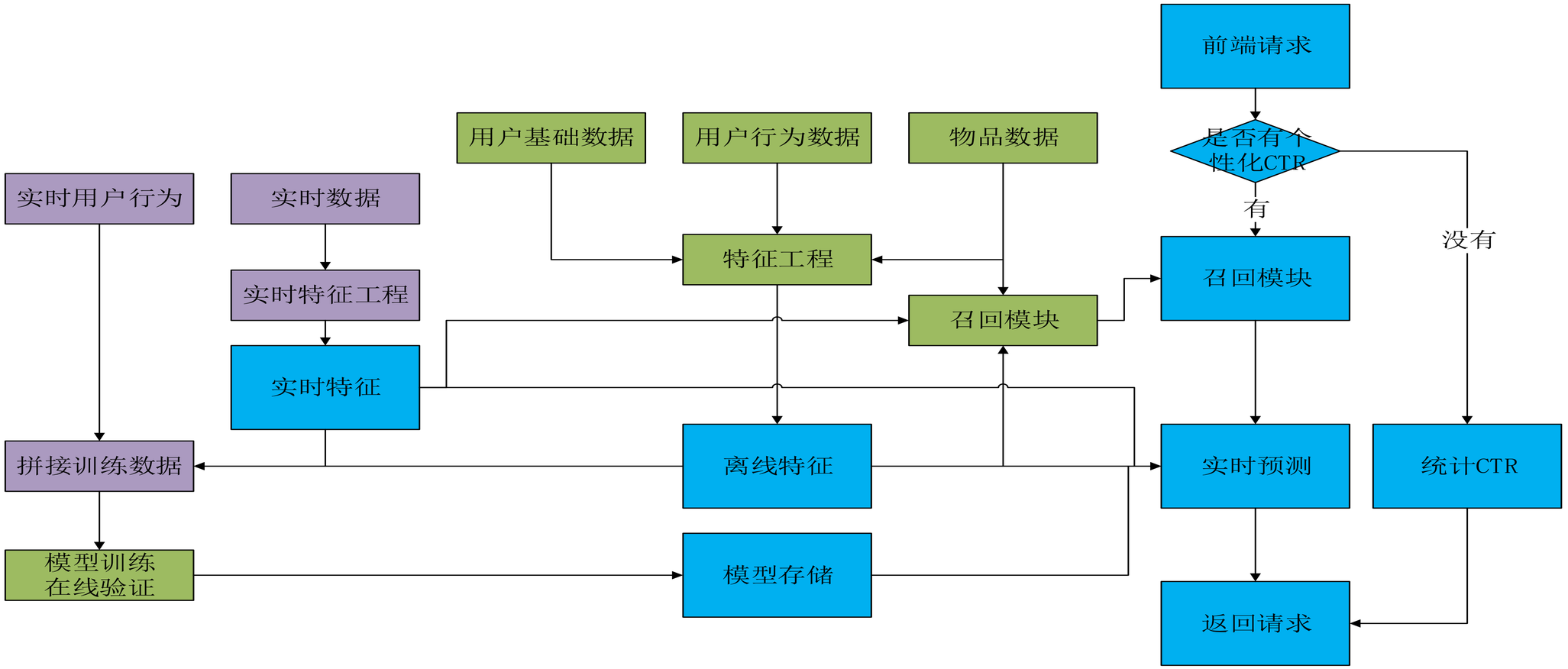

前端请求
是否有个性化CTR
有
没有
用户基础数据
用户行为数据
物品数据
实时用户行为
实时数据
特征工程
召回模块
实时特征工程
召回模块
实时特征
拼接训练数据
离线特征
实时预测
统计CTR
模型训练
在线验证
模型存储
返回请求

TABLE OF CONTENTES

➢ Why DNN

- 在特征不变的情况下能够显著提高算法效果
- CTR预估中的Final Frontier
- 可以用CNN/RNN等提取特征工程不便于提取的特征

➢ DNN的难点

- 巨大的计算量
- Inference基本上都要用C++写，新兴的互联网公司很多都是Java技术栈
- 对inference的性能要求很高
- 更改网络结构的便利性

➢ TensorFlow解决的问题

- 分布式模式能解决大规模训练
- HDFS接口能和Hadoop数据源无缝衔接
- TF Serving解决在线inference的问题
- 在TensorFlow中更改的网络结构能够很容易的在TF Serving中应用

TABLE OF CONTENTES

- 广告CTR系统架构
- Why TensorFlow
- 分布式TensorFlow训练的方案
- TensorFlow Serving实时推断的方案

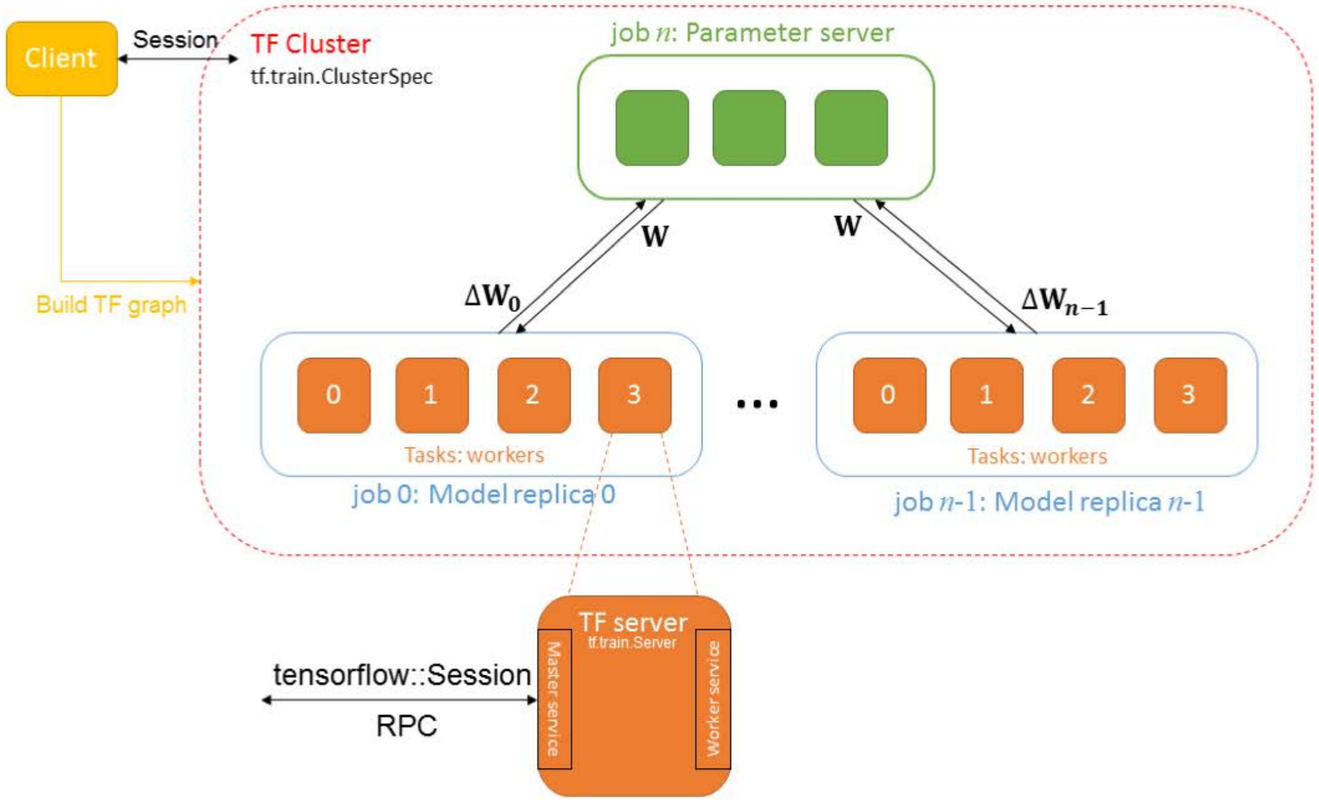
➢ 分布式 TensorFlow 中的概念

- Task. 对应特定的 TensorFlow Server, 一般来讲是一个单进程实体, Python 代码中体现为 tf.device("/job:xx/task:xx"). 从程序的使用上看, 一个 Task 属于特定的 Job, 并由该 Job 中的任务列表的索引号标识.

- Job. 由一组服务与共同目标的 Tasks 构成. 例如, PS(parameter server) Job 用于存储和共享tf.Variable, worker Job 管理一组计算密集型的 Task 节点. Job 中的 Task 一般会运行在不同的服务器上.

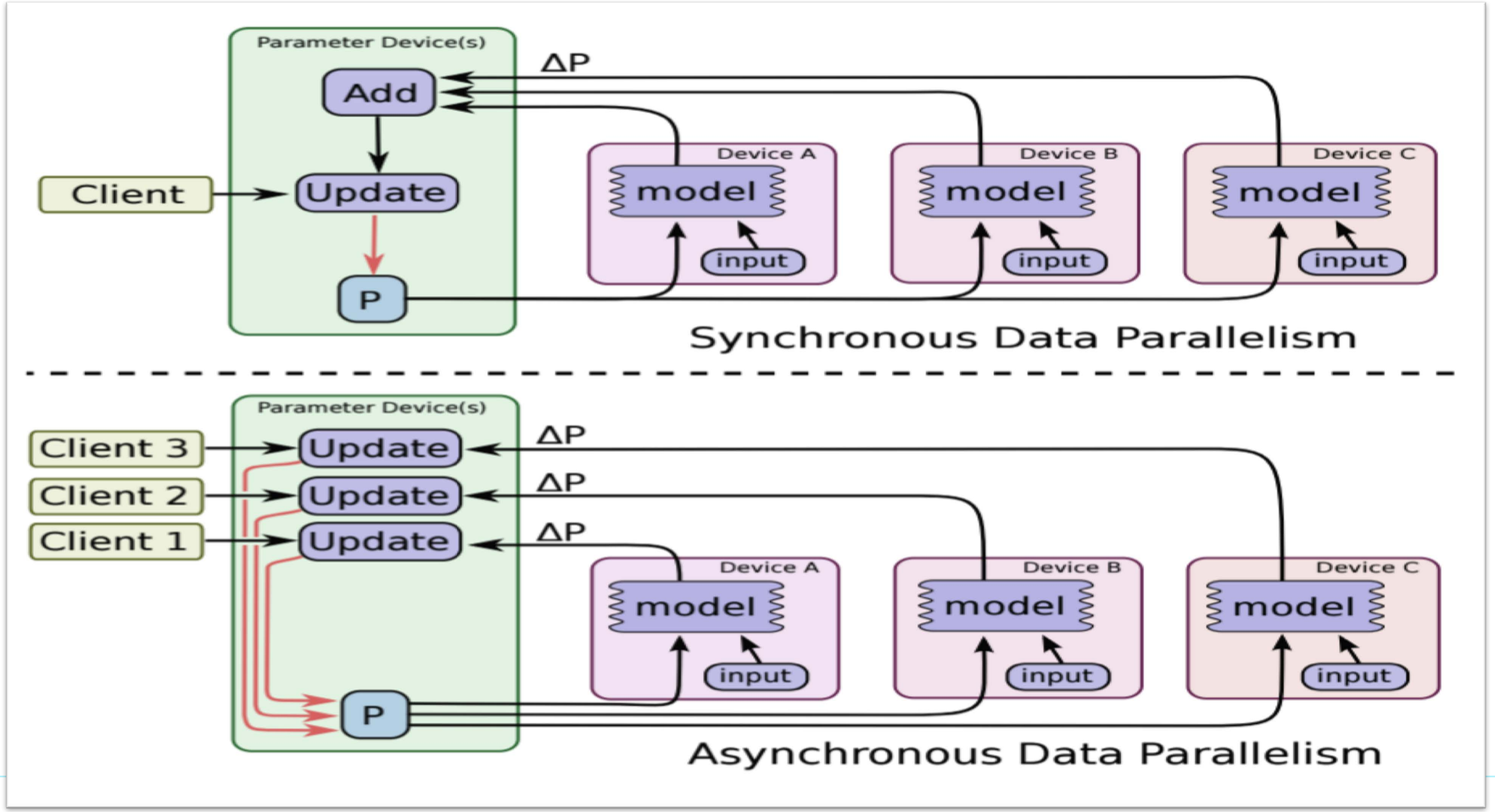
- Cluster. 所有参与计算的 Task 构成 Cluster, 使用 tf.train.ClusterSpec 描述, 其中的每个独立资源的描述形式为hostname:port, 这就表明一台服务器上其实是可以运行多个 Task 的。

➢ Distributed TensorFlow Model




```
tf.train.ClusterSpec({
    "worker": [
        "worker0.example.com:2222",
        "worker1.example.com:2222",
        "worker2.example.com:2222"
    ],
    "ps": [
        "ps0.example.com:2222",
        "ps1.example.com:2222"
    ]})
```

➢ Async/Sync Training



➢ Why TensorFlow on Kubernetes

- 训练时 TensorFlow 各个 Task 资源无法隔离；
- 缺乏 GPU 调度能力，需要用户手动配置和管理 GPU ；
- 集群规模大时，Task 管理很麻烦；
- 要查看各个 Task 的训练日志不方便；
- 创建一个大规模 TensorFlow 集群不轻松；

➢ Kubernetes is Suitable

- 提供ResourceQuota, LimitRanger等多种资源管理机制；
- 支持GPU配置(only limits)和调度；
- 容器运行训练任务，提供全套的容器PLEG接口 ；
- 对接成熟的EFK日志方案；
- 轻松快捷的创建一个大规模TensorFlow集群

HDFS + K8S + TF

Components:

T*ensorFlow: 1.3.0*

K*ubernetes: 1.7.4*

D*ocker: 1.12.6*

Harbor: 1.1.2

Glusterfs: 3.10.5

Contiv netplugin: 0.1-12-23-2016.19-44-42.UTC

OVS: 2.3.1

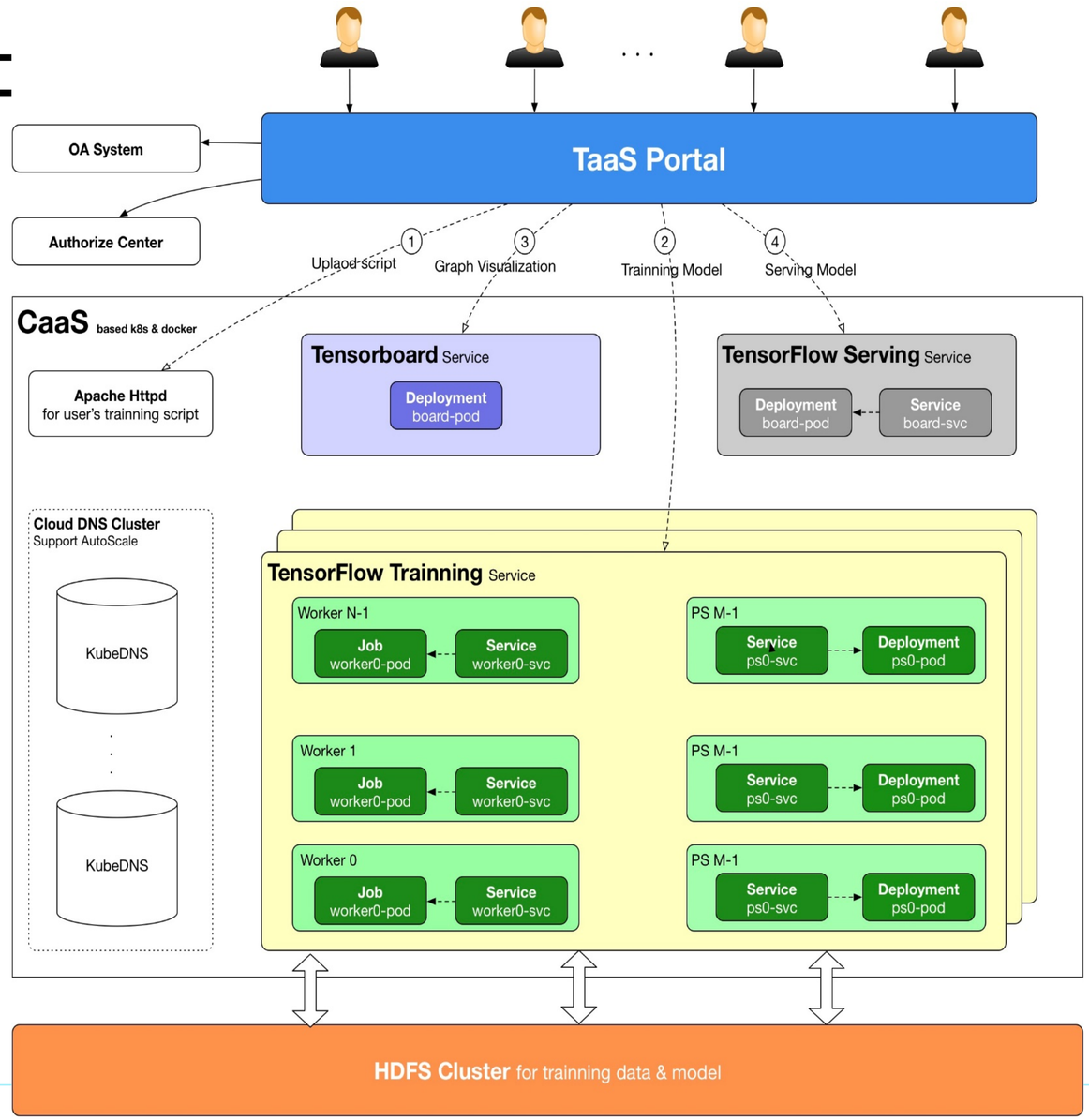
Keepalived: 1.3.5

Haproxy:1.7.8

Etcd2: 2.3.7

Etcd3: 3.2.1

python: 2.7.5



➢ 经验分享

- 使用 TensorFlow 的队列机制进行多线程读写时，官方提供了类似下面的一段代码:

```
def read_and_decode(filename_queue):
    reader = tf.TFRecordReader()
     _, serialized_example = reader.read(filename_queue)
    return serialized_example

serialized_example = read_and_decode(filename_queue)

batch_serialized_example = tf.train.shuffle_batch(
    [serialized_example],
    batch_size=batch_size,
    num_threads=thread_number,
    capacity=capacity,
    min_after_dequeue=min_after_dequeue)

features = tf.parse_example(
    batch_serialized_example,
    features={
        "label": tf.FixedLenFeature([], tf.float32),
        "ids": tf.VarLenFeature(tf.int64),
        "values": tf.VarLenFeature(tf.float32),
    })
```

如果直接使用上面的代码，会出现很严重的 io 读取性能，在我们的测试环境，io speed 只有 30 - 50MB/s。

➢ 经验分享

- 经排查发现，sess.run 运行的时候有固定的开销，大概在100-200usec，也就是每秒钟只能做5k-10k sess.run，如果运行sess.run 的时候，每次将记录逐条加入队列，会变得很慢，因此，应该在运sess.run 的时候将多条记录批量加入队列，通过使用 tf.TFRecordReader.read_up to 函数，同时在 shuffle_batch 的时候将 enqueue_many 设置为 True，io speed有10倍的提速，在我们的测试场景，使用如下代码，io读取速度可以提高到 **300-500 MB/s**。

```
def batch_read_and_decode(filename_queue, enqueue_many_size=1024):
    reader = tf.TFRecordReader()
     _, queue_batch = reader.read_up_to(filename_queue, enqueue_many_size)
    return queue_batch

serialized_example = batch_read_and_decode(filename_queue)

batch_serialized_example = tf.train.shuffle_batch(
    [serialized_example],
    batch_size=batch_size,
    num_threads=thread_number,
    capacity=capacity,
    min_after_dequeue=min_after_dequeue,
    enqueue_many=True)

features = tf.parse_example(
    batch_serialized_example,
    features={
        "label": tf.FixedLenFeature([], tf.float32),
        "ids": tf.VarLenFeature(tf.int64),
        "values": tf.VarLenFeature(tf.float32),
    })
```

使用 read_up_to 函数

设置 enqueue_many

➢ 经验分享

- 另外，使用 TensorFlow 的队列机制，默认情况下，当最后一个 batch 的数据小于设定的 batch_size 时，队列会关闭，但在预测场景，会漏掉最后一个 batch 的样本，导致数据缺失，这时可以在 shuffle_batch 的时候将 allow_final_batch 设置为 True。

```
def batch_read_and_decode(filename_queue, enqueue_many_size=1024):
    reader = tf.TFRecordReader()
     _, queue_batch = reader.read_up_to(filename_queue, enqueue_many_size)
    return queue_batch

serialized_example = batch_read_and_decode(filename_queue)

batch_serialized_example = tf.train.shuffle_batch(
    [serialized_example],
    batch_size=batch_size,
    num_threads=thread_number,
    capacity=capacity,
    min_after_dequeue=min_after_dequeue,
    enqueue_many=True,
    allow_smaller_final_batch=True)
```

将 allow_final_batch 设置为 True

➢ 经验分享

- 在起分布式训练任务的时候，经常会碰到 Variables not initialized 的问题，这是因为在 chief worker 初始化变量和其他 worker 检查变量是否初始化两者之间有一个竞态条件，如果其他 worker 赢得了这个竞态条件，它就会进入每 30 秒的循环检查，如果使用 between-graph + async 的模型，可以在代码中加入下面一行代码 config = tf.ConfigProto(device_filters=["/job:ps", "/job:worker/task:%d" % FLAGS.task_index])

- 适当增加 worker 和 ps 的数量可以提高每秒处理的样本数。在我们的测试场景(between-graph+async)，使用 LR 模型，特征维度 10000，有效维度 600左右，worker 数量，ps 数量和每秒处理的样本数量数据如下：

Worker 数量	Ps 数量	每秒处理的样本数
30	1	30000 - 50000
60	1	70000 - 90000
120	2	150000 - 170000
240	4	280000 - 300000

TABLE OF CONTENTES

- 广告CTR系统架构
- Why TensorFlow
- 分布式TensorFlow训练的方案
- TensorFlow Serving实时推断的方案

➢ TensorFlow Serving

TensorFlow Serving 是一个将机器学习模型用于生产环境的灵活高性能服务系统，它能够实现:

- ✓ 模型生命周期管理，简化并加速从模型到生产的过程
- ✓ 使用多重算法进行试验，安全地部署新模型并运行试验
- ✓ GPU资源的有效使用，提升生产环境的性能

➢ TensorFlow Serving

在生产环境中使用TensorFlow Serving，主要涉及到三个核心模块：

- TensorFlow Serving ModelServer
  - –apt-get安装或者C++源码编译
  - –负责发现和加载新版本模型(根据模型文件夹的版本号判断)，并通过gRPC提供服务
  - –根据相关配置规则处理旧版本模型
- Model
  - –训练结束后调用相关API导出的模型，供ModelServer加载
  - –定义模型输出输入接口以及服务名称
- Client
  - –接收并处理生产数据，将其按照Model的接口要求进行相关序列化
  - –通过预先定义的gRPC函数向ModelServer发送数据并等待获得结果

➢ 安装TensorFlow Serving(ModelServer)

- 安装各类必须软件
  - –Bazel (注意选择与相应TensorFlow版本对应的Bazel版本)
  - –gRPC
  - –包括git和zip等在内的依赖包 (各个不同linux发行版本可能有不同的安装命令)

- 选择安装模式：源码安装或apt-get安装
  - –源码安装有更多选项，方便得到更适合已有机器的高性能Server
  - –apt-get安装更方便快捷，适合使用Debian或者Ubuntu的系统

➢ 安装TensorFlow Serving (源码安装)

- 下载代码库，配置TensorFlow，编译ModelServer

- 配置TensorFlow的注意事项
  - –如果Server需要读取存储在HDFS的模型，需要在此处选择HDFS支持
  - –如果Server需要MKL提供计算支持，需要在此处选择下载安装MKL，并在后续编译时添加相应的flag (1.4版本TensorFlow已经不再需要在此处进行选择)

- 编译ModelServer的注意事项
  - –与上面对应，添加 --config=mkl 使Server调用MKL
  - –加入CPU指令编译选项，可以有效提升计算性能

➢ 导出Model

TensorFlow Serving提供两种导出Model的方式：

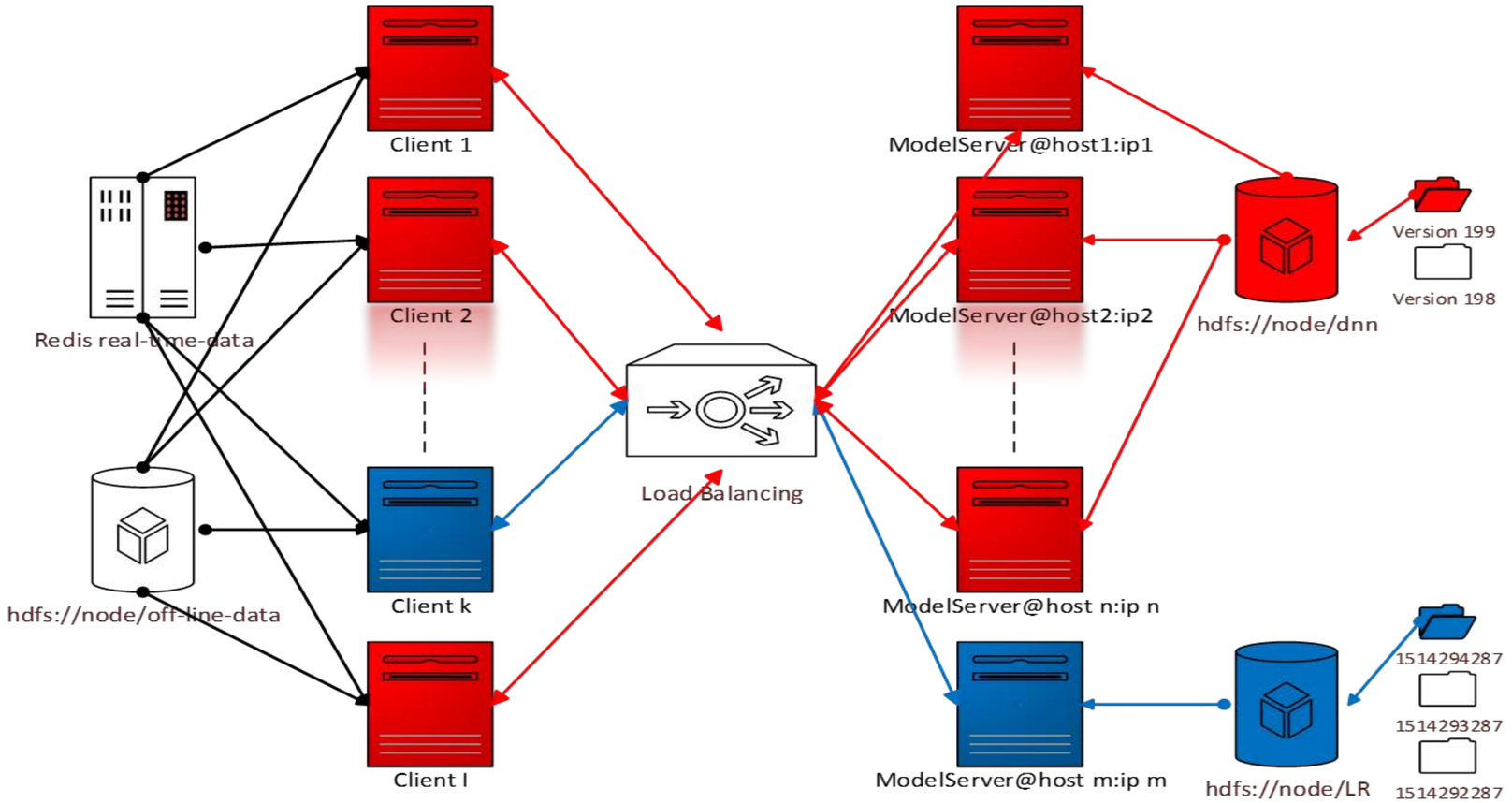
- 调用tf.saved_model.builder.SavedModelBuilder模块
  - 适用于导出自定义流程训练得到的模型
  - 可以自定义模型的输入输出接口名称和接口类型
  - 可以定义多种服务类型(prediction, classification, inference)供Client选择
- 调用tf.estimator.Estimator导出模型API
  - 适用于导出利用TensorFlow高级API训练得到的Estimator模型
  - 丰富的配套API可以更方便的导出模型
- 导出的Model存放在以数字命名的文件夹下，ModelServer以更大的数字作为新模型的标准：
  - /model/1514344808/saved_model.pb
  - /model/1514344808/variables/variables.index
  - /model/1514344808/variables/variables.data-**-of-**

➢ 编写Client

ModelServer通过gRPC协议，接收处理Protocol Buffers的序列化数据，并返回序列化的结果，因此任何可以使用gRPC和Protocol Buffers的编程语言都可以用来编写Client。以Java/Scala为例，需要以下资源：

- 若干依赖库
  - io.grpc.{grpc-stub, grpc-netty, grpc-protobuf}
  - org.tensorflow.{tensorflow, proto}
- 编译.proto文件得到gRPC服务Java接口
- Client的通用数据处理流程：
  - 处理原始数据得到Model输入接口需要格式的数据
  - 将数据填入相关类型的TensorProto数据结构
  - 将所有输入接口对应的TensorProto加入request数据结构
  - 调用gRPC函数发送request到ModelServer
  - 同步或者异步等待结果

➢ Serving – 基础生产架构



➢ Serving – 相关经验

将广告和应用商店的若干推荐和预测项目通过TensorFlow Serving进行线上生产和测试后，

得到以下结果：

- 即使添加HDFS支持的编译选项，1.3版本的默认ModelServer也无法成功加载HDFS路径的模型
- 需要在/serving/tools/bazel.rc中添加 build:mkl --define=using_mkl=true 才能成功链接到MKL库
- 1.4版本的默认ModelServer的性能高于1.3版本的默认ModelServer，表现在更低的CPU使用率和更快的计算速度
- 在某些应用场景下，添加MKL支持会使得CPU使用率暴增，但是计算速度剧降
- String类型的TensorProto可以支持ByteString格式的tf.Example
- 根据应用场景的不同，最优性能的单次request发送数据量也随之改变
- Client发送数据的变量名称以及获取结果的变量名称需要与模型导出时使用的输入输出变量名称对应
- 默认ModelServer会自动加载监控目录下的最新Model版本，并卸载旧版本的Model

➢ Serving – 相关经验

优先选择较新的版本…

TensorFlow本身还在快速发展中，不断有新的功能集成进来，旧的BUG被修正，性能也在稳步提升：

- 1.3版本的Tensorflow Serving代码似乎在加载模型上还存在问题，即使添加HDFS支持的编译选项，1.3版本的默认ModelServer也无法成功加载HDFS路径下的模型
- 通过简单修改读取模型部分的源码，可以加载HDFS路径下的模型，但是加载本地路径下的模型会出现问题
- 1.4版本的TensorFlow Serving在正常编译下就可以任意加载各种路径下的模型
- 1.4版本默认ModelServer的性能高于1.3版本的默认ModelServer，表现在更低的CPU使用率和更快的计算速度

➢ Serving-相关经验

添加MKL的支持有时候并不能带来性能的提升….

首先是添加MKL的支持

根据Intel官网提供的编译方法，有时候没办法成功将server链接到MKL库上。

根据警告提示“WARNING: Config values are not defined in any .rc file: mkl.”，

- 需要在/serving/tools/bazel.rc中添加 build:mkl --define=using_mkl=true 才能成功链接到MKL库
- 再用ldd命令查询server的依赖库，就可以看到server成功链接到了MKL

添加MKL支持会使得矩阵计算强制使用MKL库函数，提高CPU使用率，在很多时候会用满计算能力，但是如果网络规模较小，这种计算能力的提升就没有办法充分利用，反倒会因为其他因素如高并发等使用场景造成性能的急剧下降。

➢ Serving-相关经验

Java/Scala版本Client的相关经验…

利用protoc加载protoc-gen-grpc-java插件编译官方TensorFlow Serving
目录下相关protobuf文件得到的Java接口是编写Client的基础…

- String类型的TensorProto可以支持ByteString格式的tf.Example
- tf.Example搭配request一次性批量发送数据是获得高性能的必要途径
- 根据应用场景的不同，最优性能的单次request发送数据量也随之改变，
  但是上限似乎是一次发送1024份数据
- Client发送数据的变量名称以及获取结果的变量名称需要与模型导出时使用的
  输入输出变量名称对应

Thanks!