深度学习并行与分布计算方法

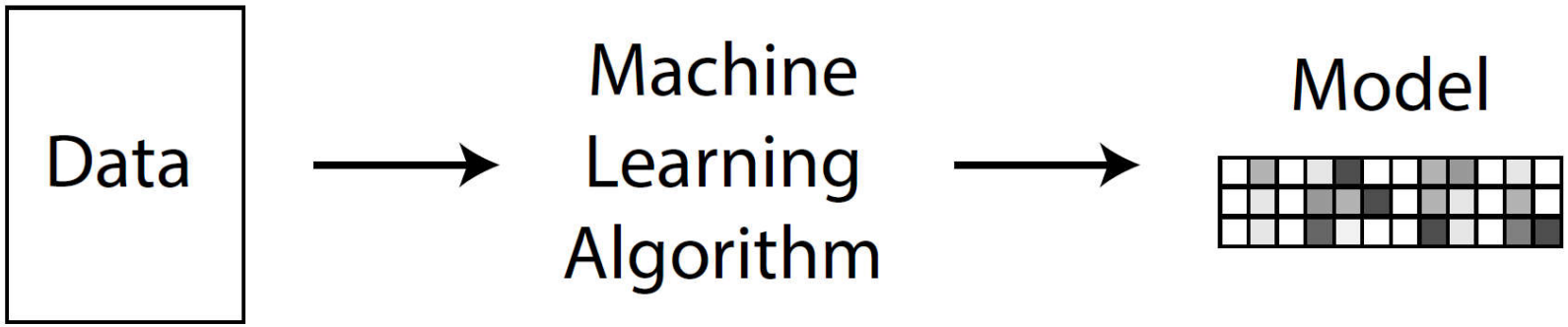
PDL, School of Computer, NUDT

主要内容

- 回顾：深度学习训练过程
- 1. 为什么要用分布式机器学习？
- 2. 并行与分布式机器学习方法
- 3. Tensorflow并行与分布处理
- 4. 实例分析：Tensorflow并行与分布处理案例

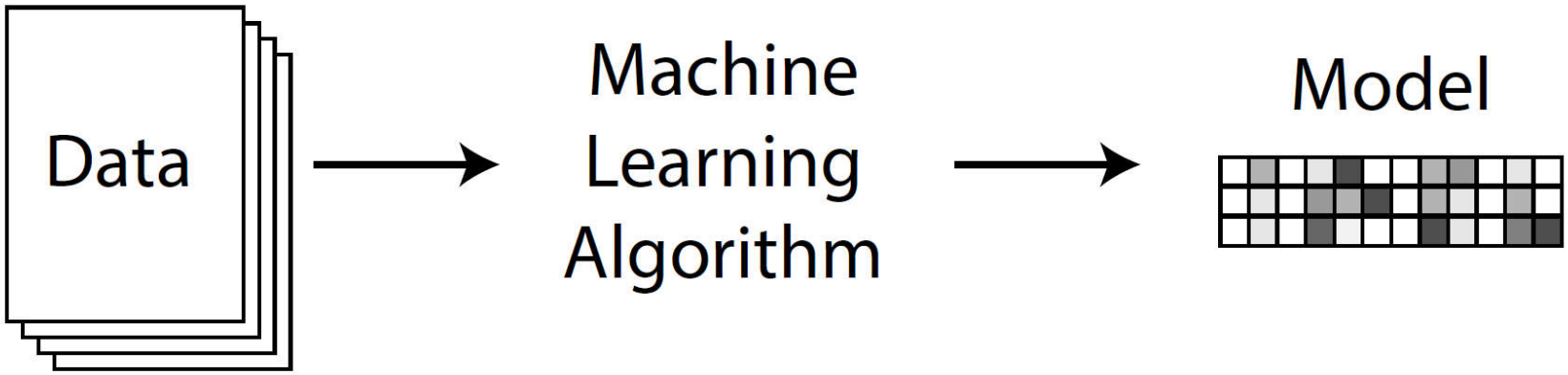
为什么要用分布式机器学习？

- 少量数据+简单模型：传统CPU服务器即可满足需求



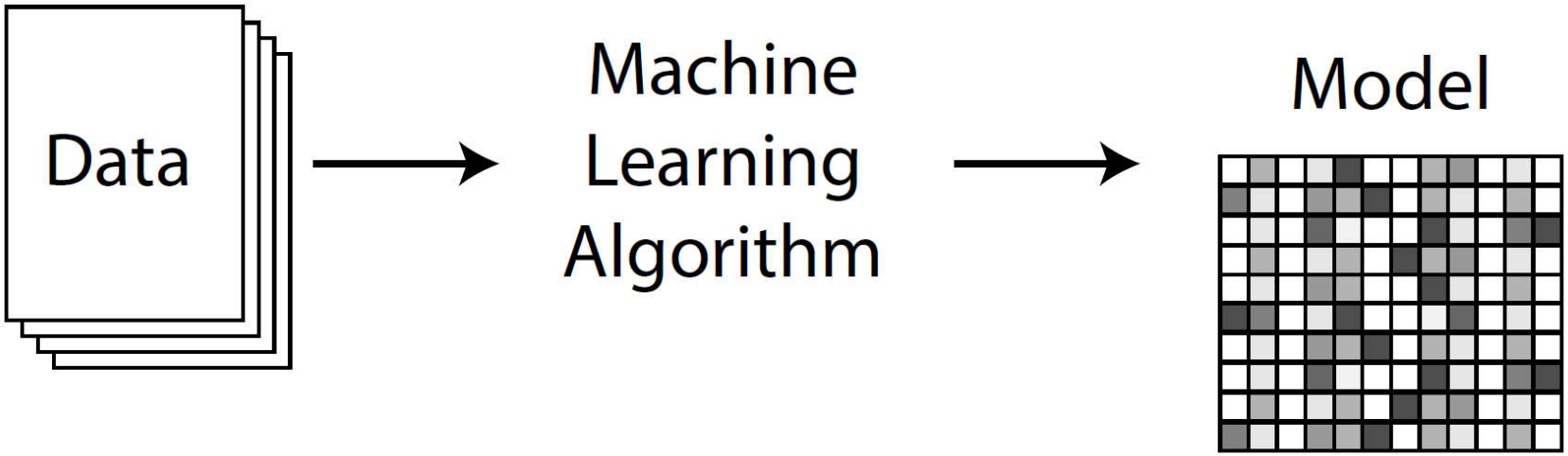
为什么要用分布式机器学习？

- 大量数据+简单模型：GPU服务器即可满足需求



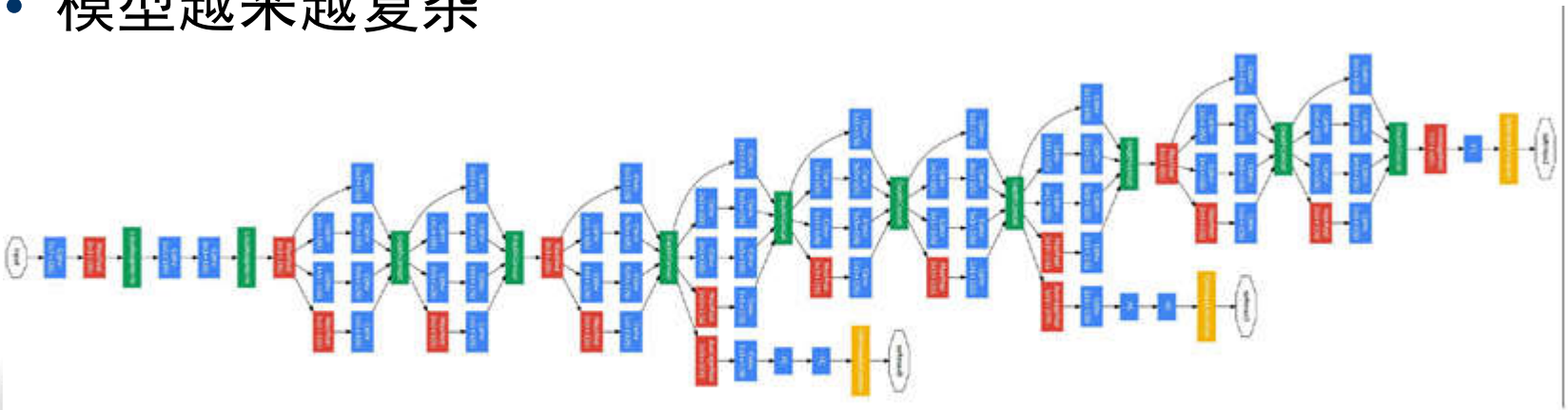
为什么要用分布式机器学习？

- 大量数据+复杂模型：服务器集群才能满足需求



为什么要用分布式机器学习？

- 模型越来越复杂



Inception (2015)　22 layers,
5M parameters

Residual Net (2015)　152 layers

为什么要用分布式机器学习？

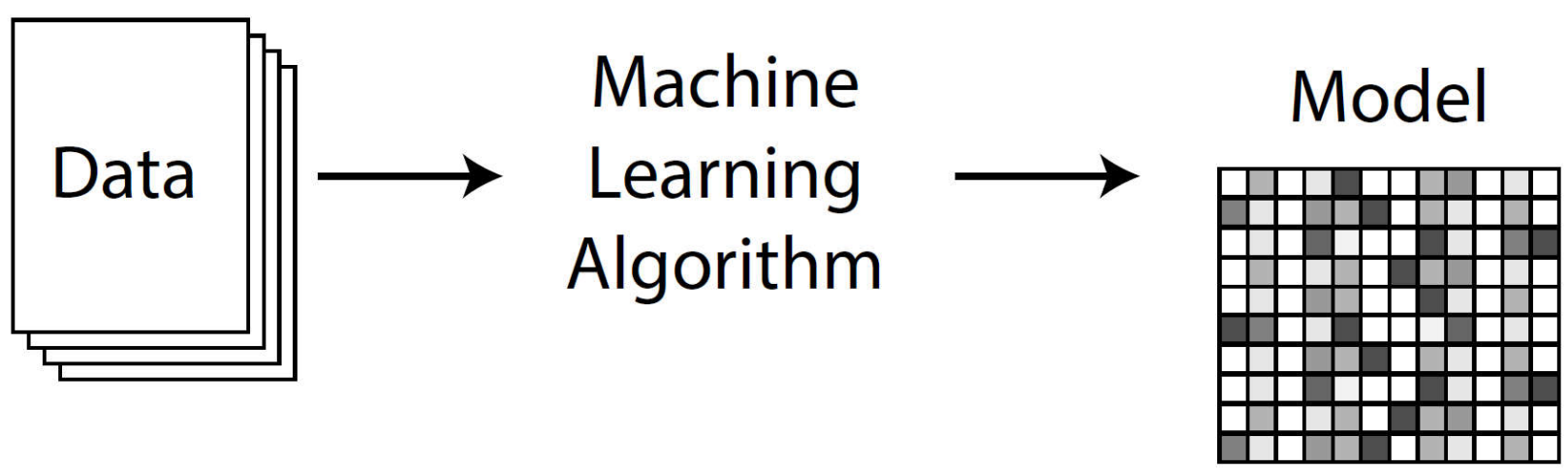
- 数据越来越多：1000主题，27TB的数据集（ClueWeb12）

Topic Modeling (LDA)

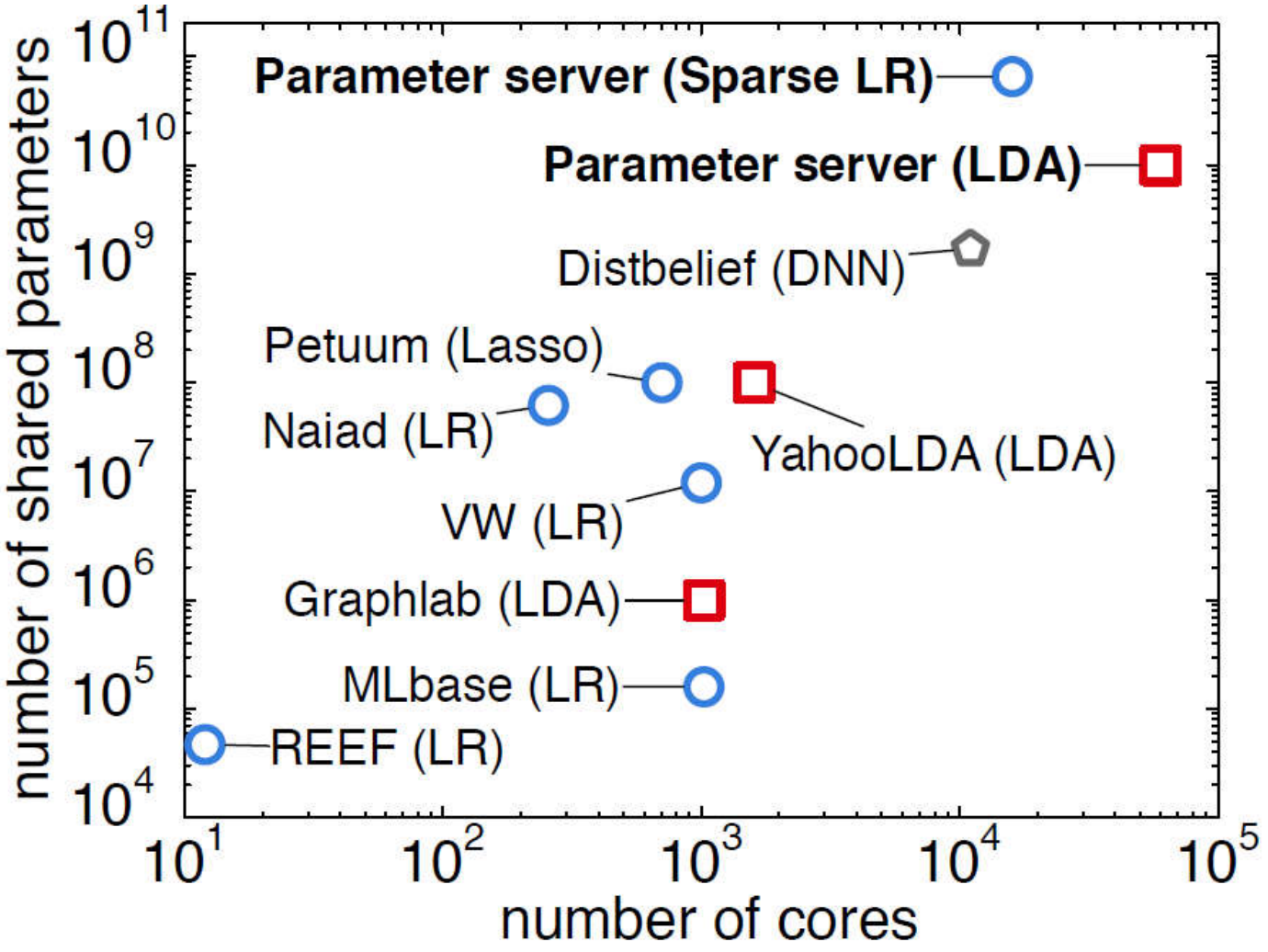
Tourism	Video games	Javascript	Biology
hotel	allianc	write	tag
local	hord	docum	protein
car	euro	var	hypothet
holidai	warcraft	return	gene
area	wow	prop	cytoplasm
golf	gold	function	prk
wed	warhamm	subset	locu

为什么要用分布式机器学习？

- 模型参数很大，超过单个机器的容纳能力（比如大型Logistic Regression和神经网络）
- 训练数据巨大，需要分布式并行提速（大数据）



为什么要用分布式机器学习？



为什么要用分布式机器学习？

- 海量的数据
  - ImageNet(1k): 180GB
  - ImageNet(22K): TB 级
  - 工业数据：更大
- 神经网络体系结构越来越大
  - 任务驱动使得网络结构深度化
  - 基本在10~100层的神经网络结构，100MB以上的模型参数
- 为了快速获得模型训练结果
  - 适配不同的超参数设置
  - GoogleNet模型的训练往往需要几天甚至几周

主要内容

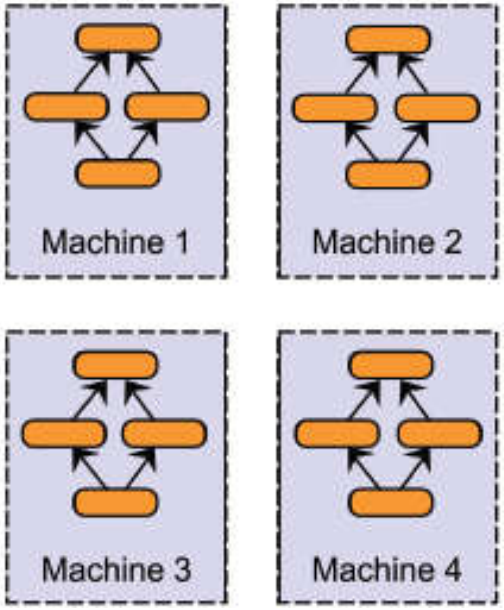
- 回顾：深度学习训练过程
- 1. 为什么要用分布式机器学习？
- 2. 并行与分布式机器学习方法
- 3. Tensorflow并行与分布处理
- 4. 实例分析：Tensorflow并行与分布处理案例

2. 并行与分布式机器学习方法

- 2.1. 并行训练——任务划分方法
  - 数据并行
  - 模型并行
  - 混合并行
- 2.2.并行训练——消息通信模式
  - 同步模式
  - 异步模式
- 2.3 参数服务器

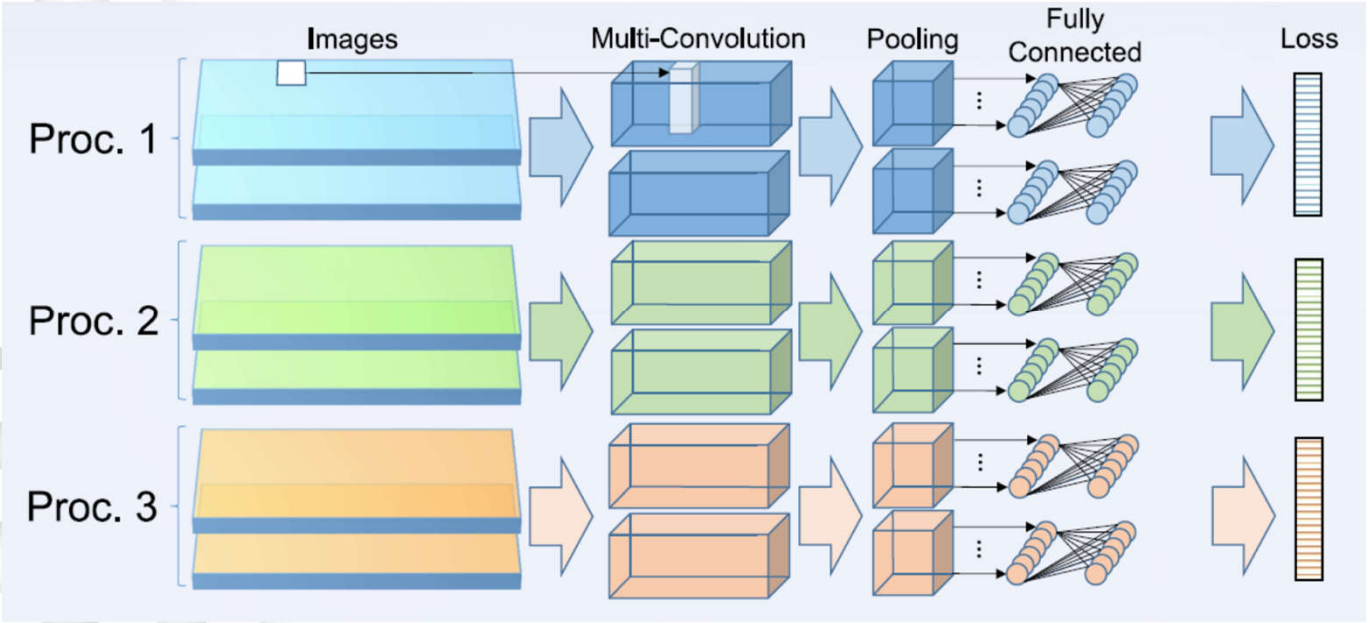
2.1.1 数据并行

- 不同的机器有同一个模型的多个副本，每个机器分配到不同的数据，然后将所有机器的计算结果按照某种方式合并



2.1.1 数据并行

- 基于深度卷积神经网络的图像处理
  - 每个进程(节点设备)都有卷积网络模型的备份
  - 每个进程处理不同的图像



2.1.1 数据并行

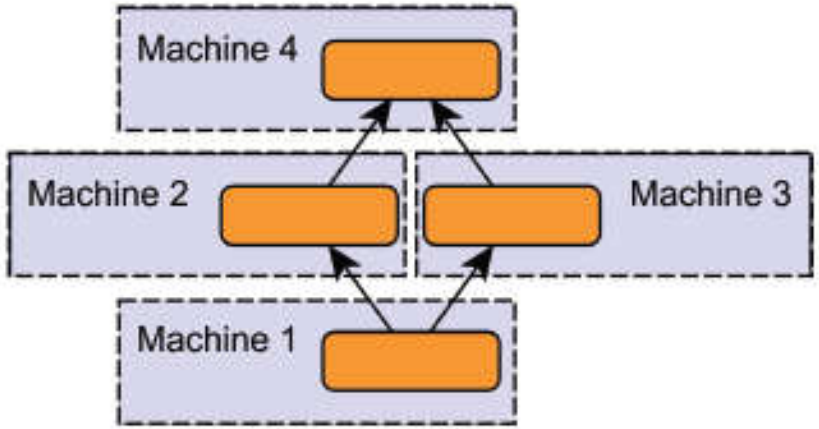
- 优点
  - 前向计算：没有数据依赖，可以实现很好的并行
  - 后向计算：只有在累加参数梯度结果的时候才需要all-to-all的通信

$$\theta_{n+1} = \theta_n - \epsilon \nabla_\theta J(\theta_n) \qquad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)$$

- 缺点
  - 每一个节点都需要存储一份完整的模型参数，存储压力大

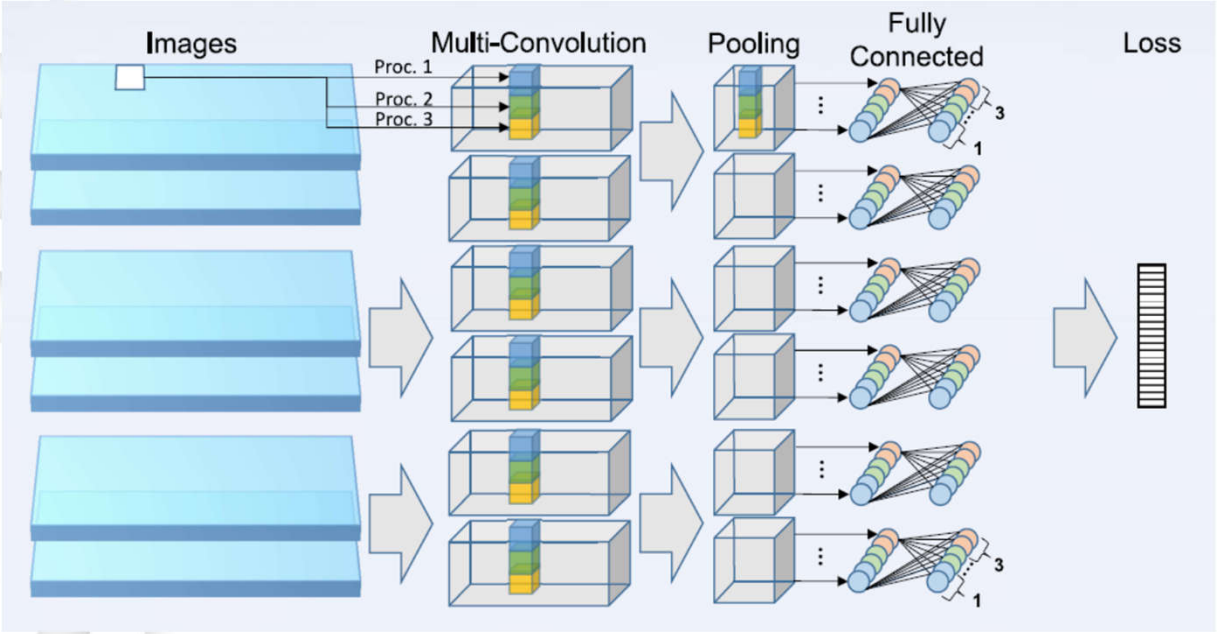
2.1.2 模型并行

- 模型并行：分布式系统中的不同机器（GPU/CPU等）负责网络模型的不同部分 —— 例如，神经网络模型的不同网络层被分配到不同的机器，或者同一层内部的不同参数被分配到不同机器。



2.1.2 模型并行

- 基于深度卷积神经网络的图像处理
  - 每个进程(节点设备)只存储对应层的模型参数
  - 每个进程处理同一幅图像(不同的特征图)



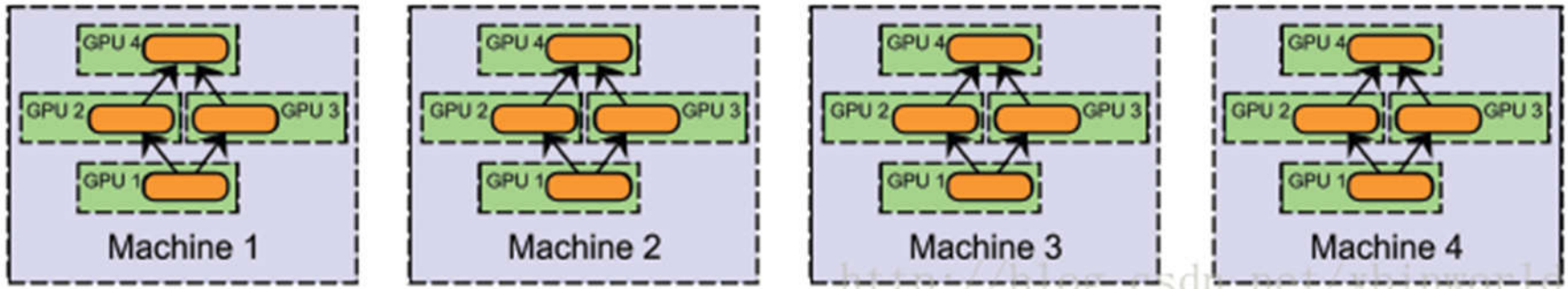
2.1.2 模型并行

- 优点
  - 参数可以分到不同的处理器存储
- 缺点
  - 每个batch的数据都要拷贝到不同的处理器
  - 后向传播的时候，每一层都需要all-to-all的通信

$$\delta_i^l = \frac{\partial C}{\partial z_i^l},\ \delta_i^{l-1} = \sum_{j=1}^{m} \frac{\partial z_j^l}{\partial z_i^{l-1}} \delta_j^l$$

2.1.3 混合并行

- 在一个集群中，既有模型并行，又有数据并行，例如，可以在同一台机器上采用模型并行化（在GPU之间切分模型），在机器之间采用数据并行化。



2.2 分布式机器学习训练过程

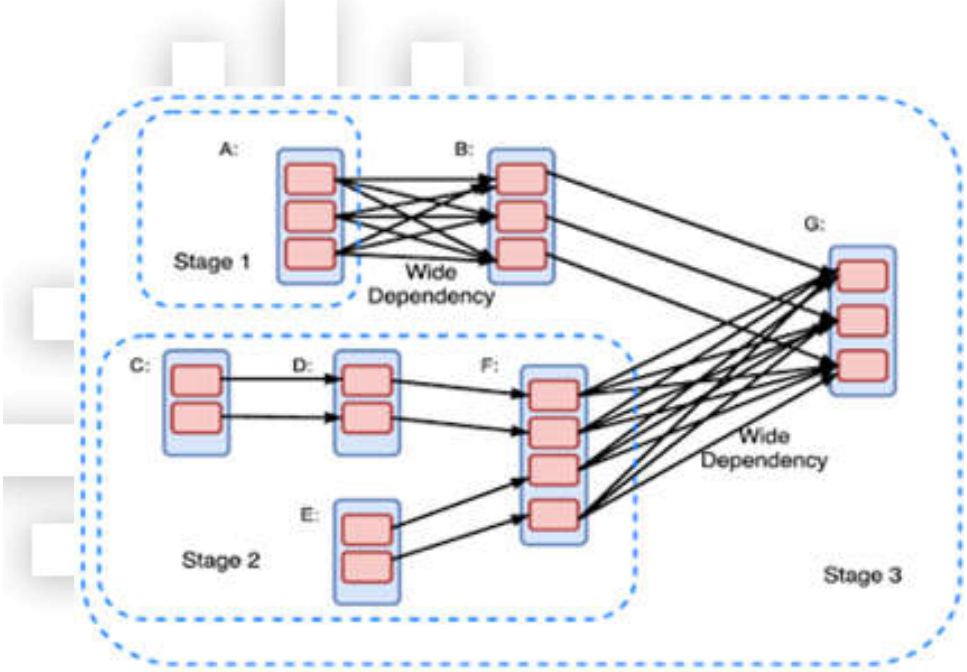
1. 分布式机器学习训练步骤：(数据并行、混合并行)
    1. 基于模型的配置随机初始化网络模型参数
    2. 将当前这组参数分发到各个工作节点
    3. 在每个工作节点，用数据集的一部分数据进行训练
    4. 根据各个工作节点的参数对全局参数进行更新
    5. 若还有训练数据没有参与训练，则继续从第二步开始

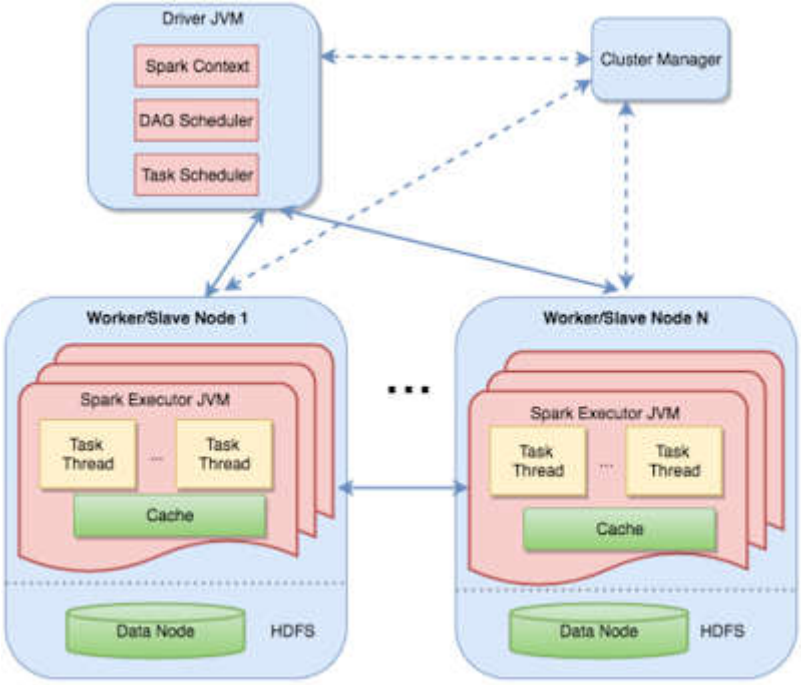
2.2 分布式机器学习实现方式

- 按照实现原理和架构可分为三类：(A Comparison of Distributed Machine Learning Platforms, ICCCN 2017)
  - 基础数据流模式: Spark
    - 借助Spark的分布式计算，构建Mlib机器学习库，实现分布式机器学习
    - 模型参数的更新代价太大，Spark对机器学习所需迭代计算的支持并不好
  - 参数服务器模式: PMLS（Petuum）
    - 目前最为流行的分布式机器学习实现方法
  - 先进数据流模型: Tensorflow, MxNet
    - 借鉴了参数服务器的参数管理机制，融入了数据流对计算进行优化

基础数据流模式

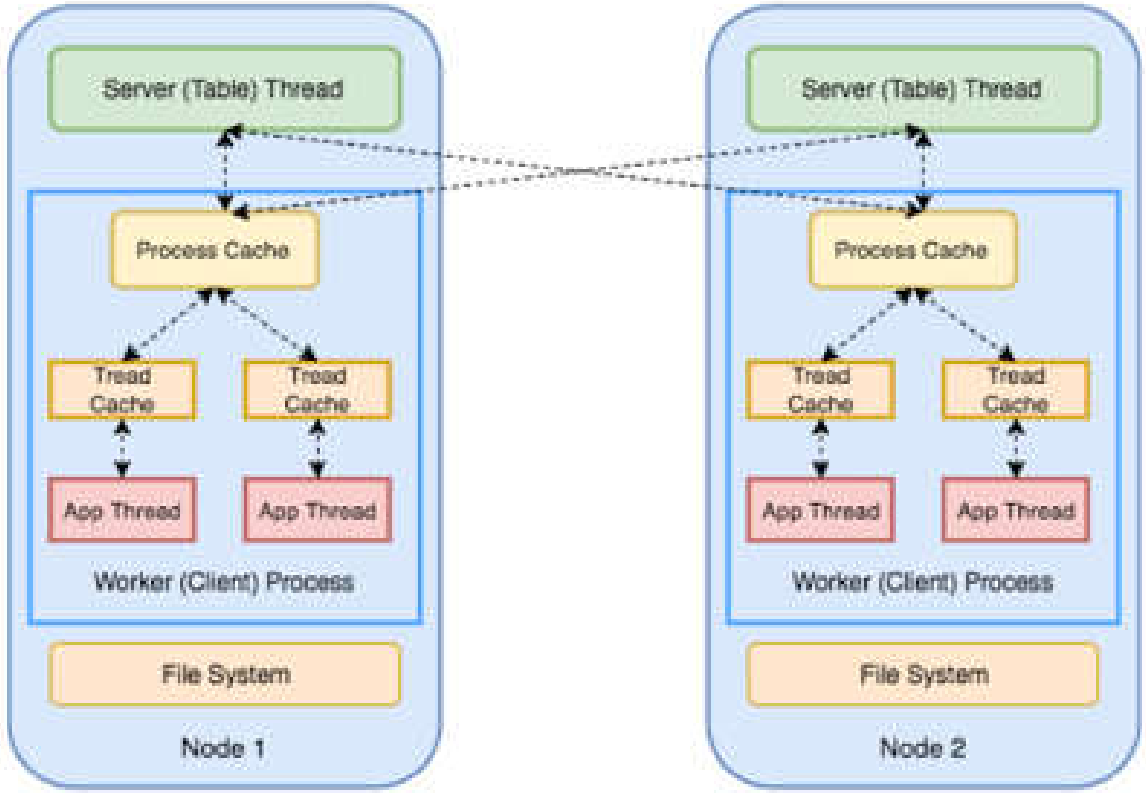
- Spark 分布模式：
  - Driver存参数，Driver存不下放在其他节点上，参数迭代会怎样？
  - 主要用于数据处理





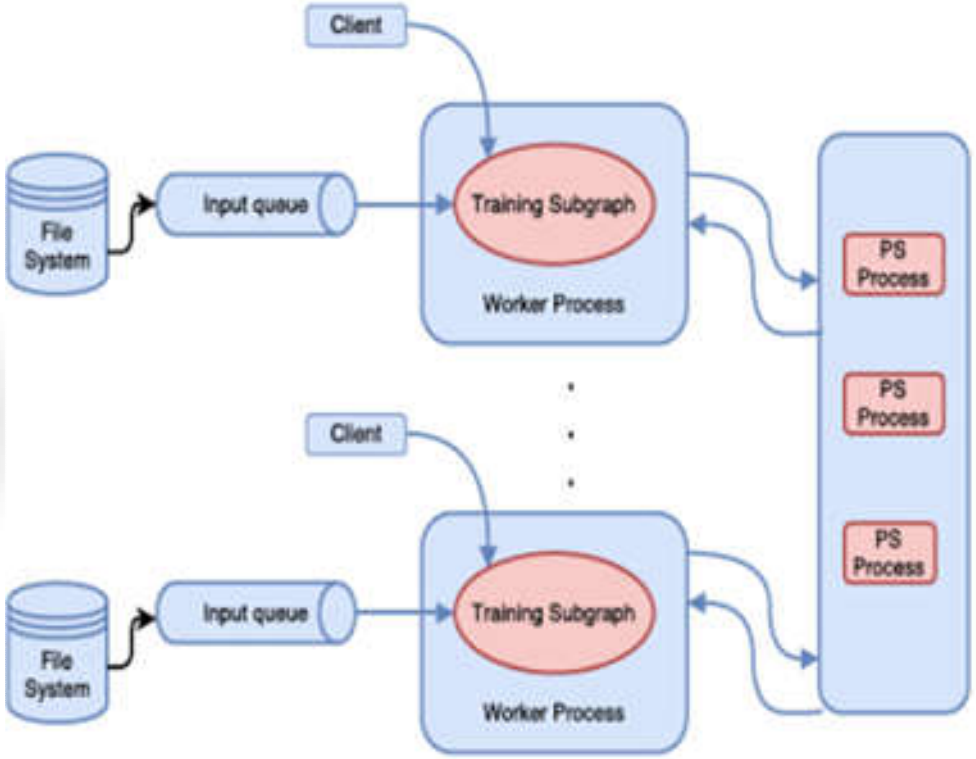
参数服务器模式

- 目前主流的分布式机器学习实现方式
  - 一些框架需要了解分布实现细节



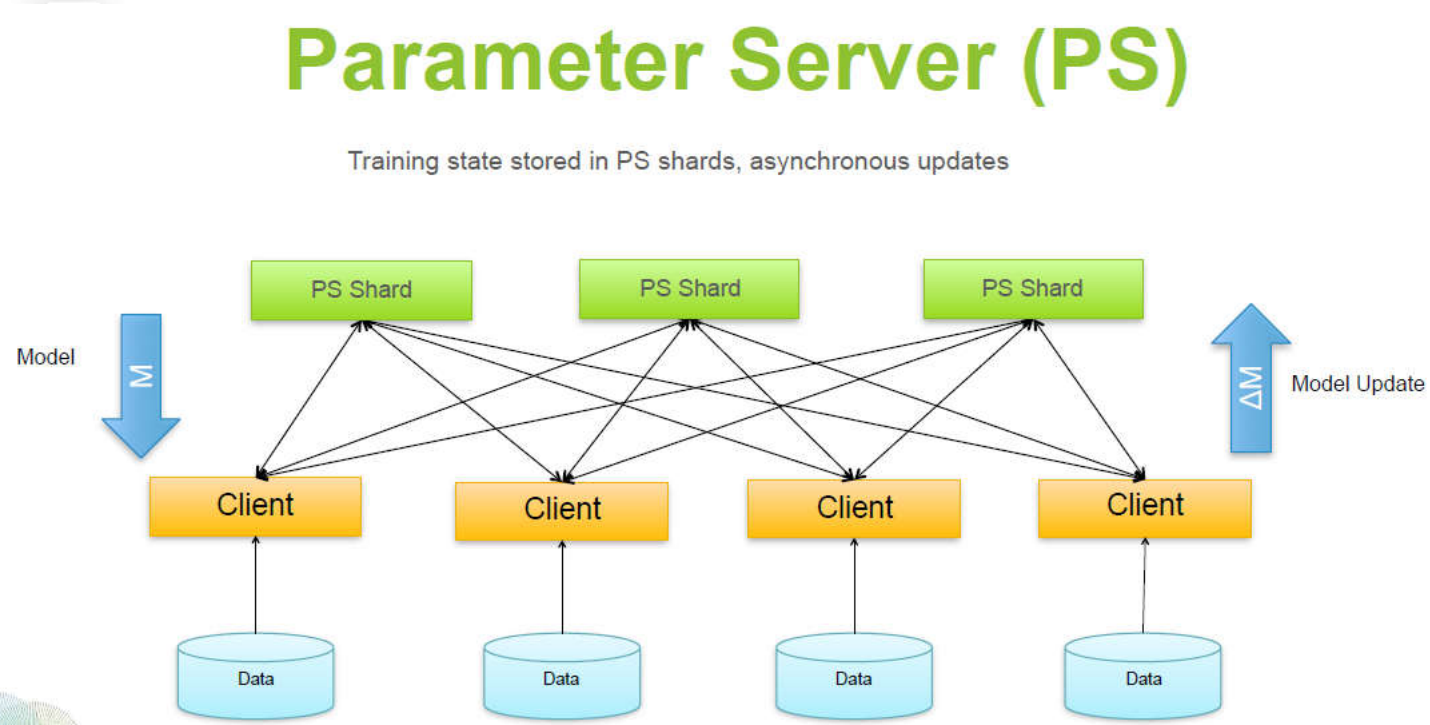
先进数据流模型

- 借鉴了参数服务器的参数管理机制，融入了数据流对计算进行优化: Tensorflow, Mxnet
  - 对机器学习研究者隐藏实现细节
  - 图计算的自动优化

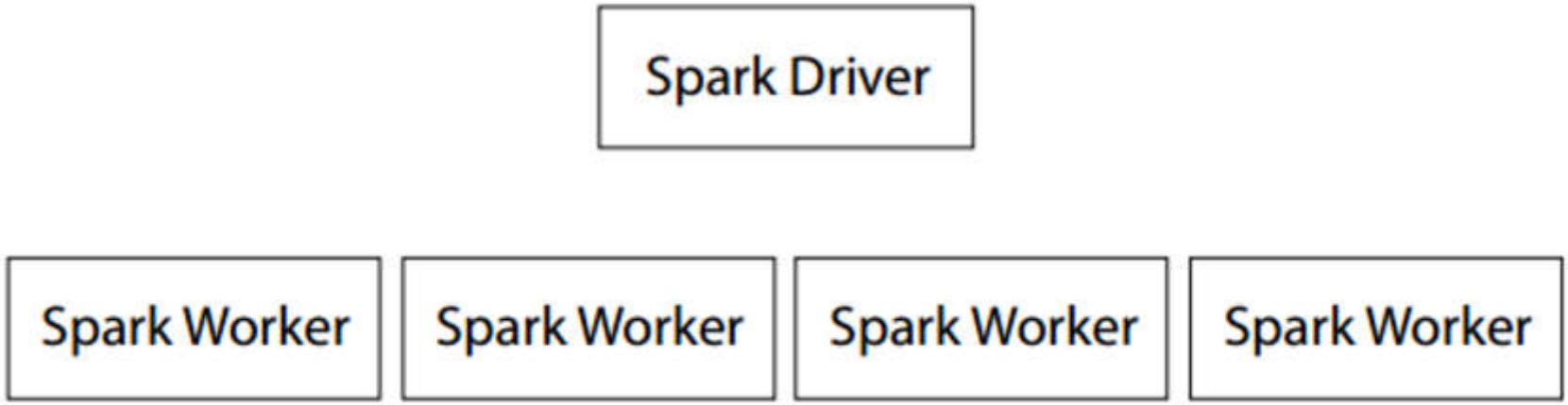


2.2.1 参数服务器

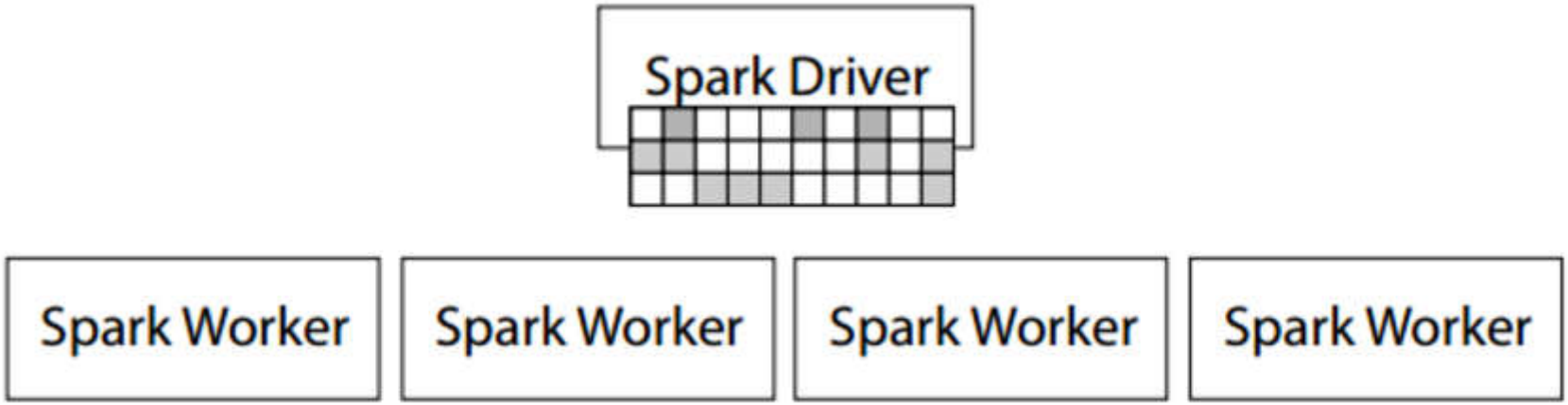
- 参数服务器是一种编程框架，用于方便分布式并行程序的编写，其中重点是对大规模参数的分布式存储和协同的支持



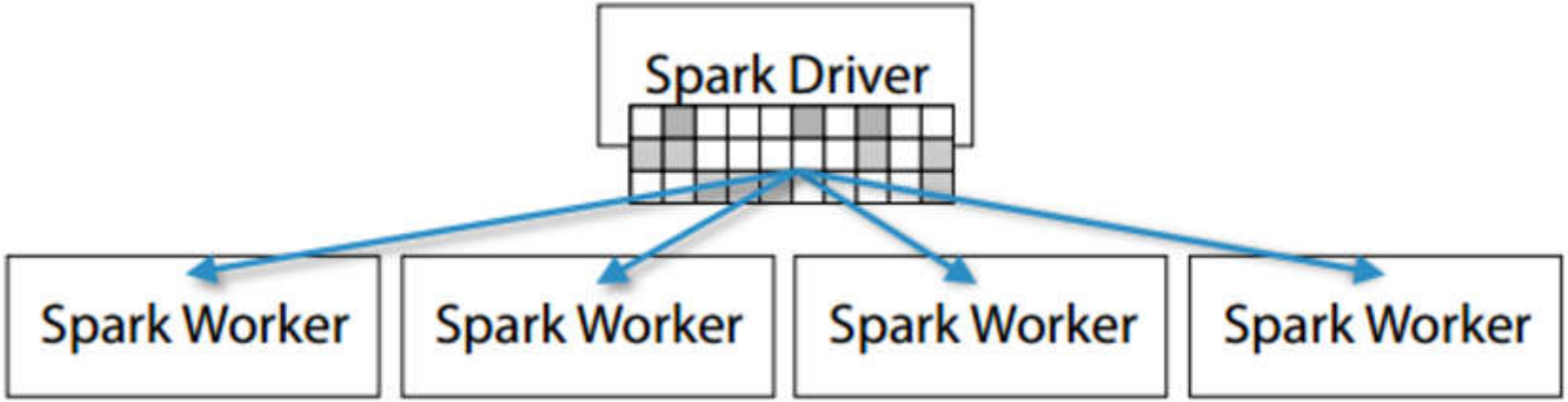
Spark Driver参数集中更新模式



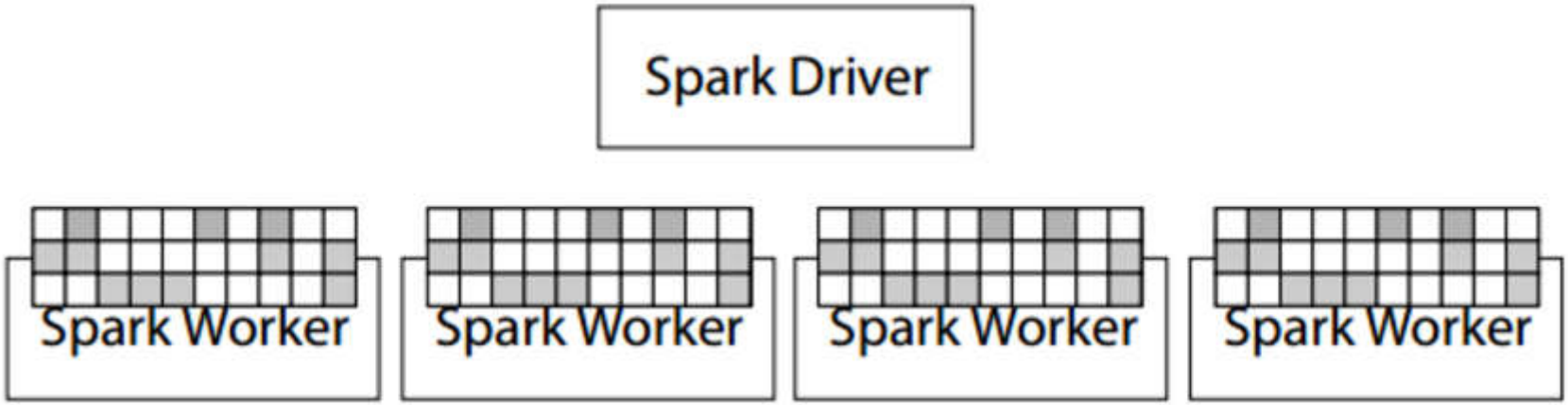
Spark Driver参数集中更新模式



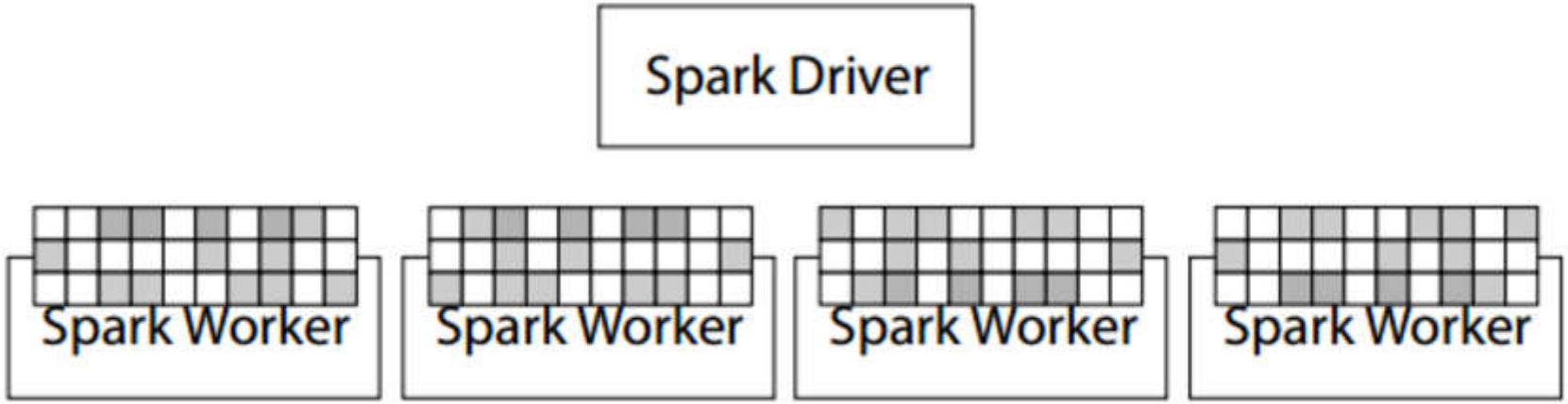
Spark Driver参数集中更新模式



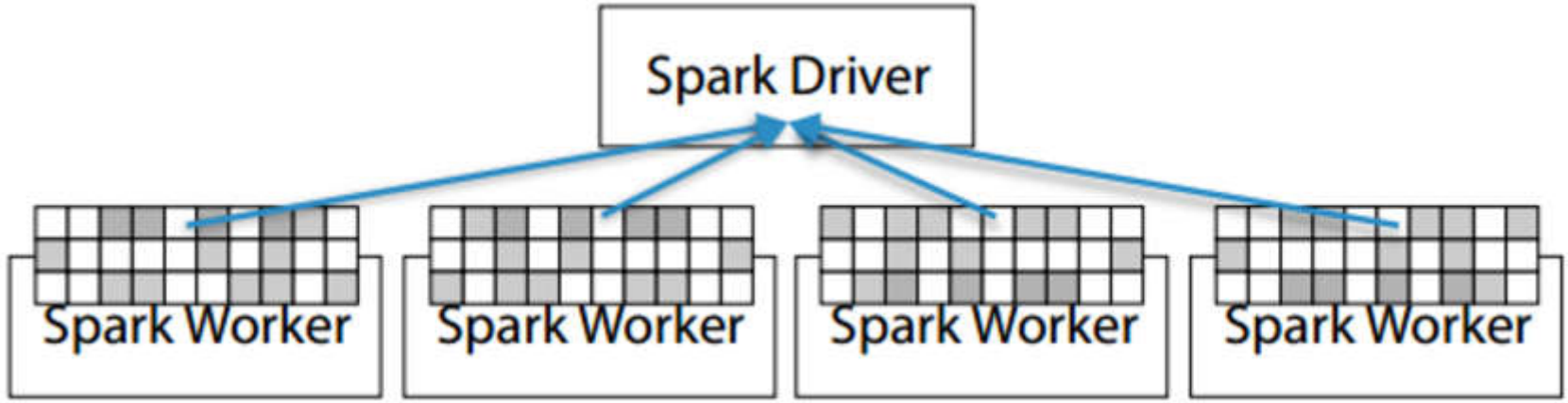
Spark Driver参数集中更新模式



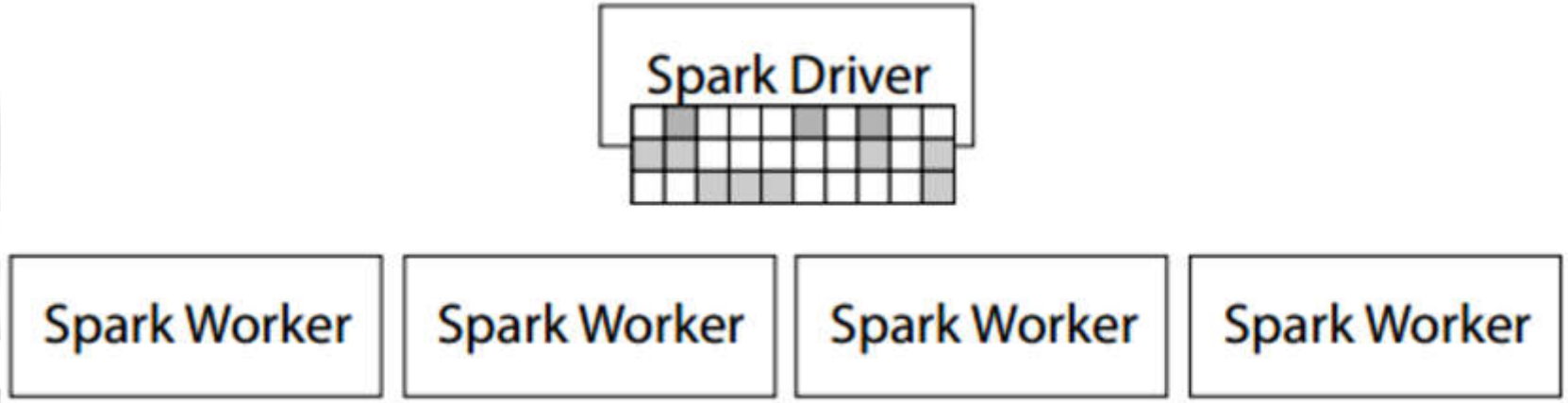
Spark Driver参数集中更新模式



Spark Driver参数集中更新模式

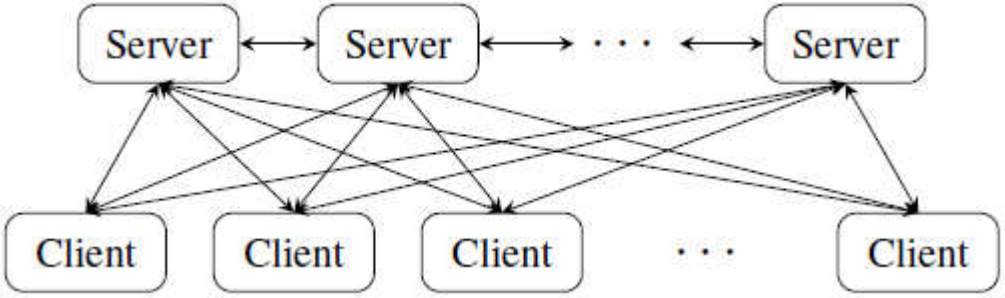


Spark Driver参数集中更新模式



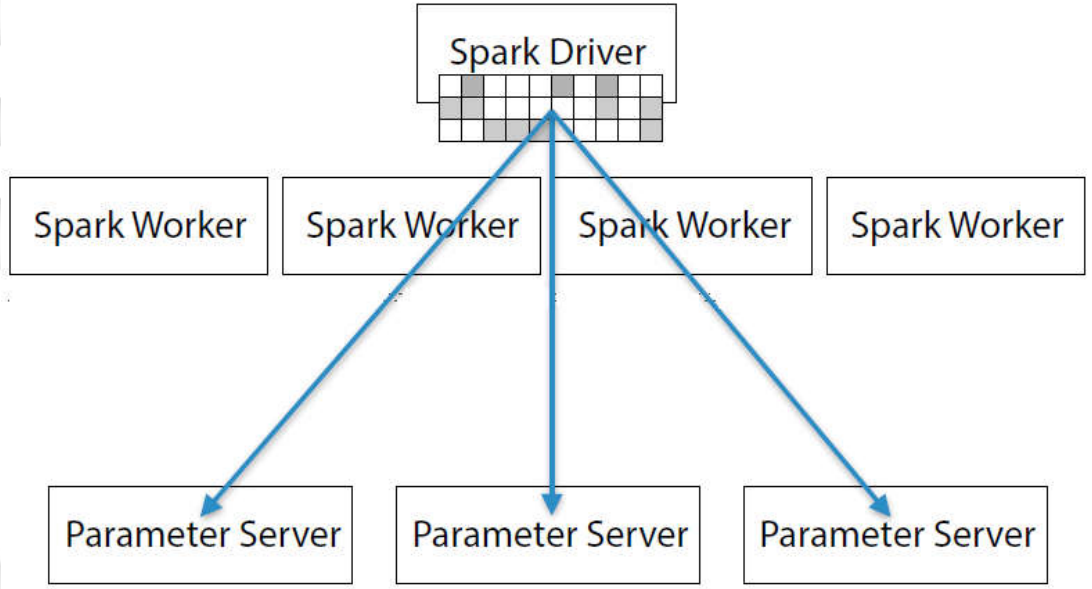
2.2.1 参数服务器

- 系统包括计算节点client和参数服务节点server
  - client负责对分配到自己本地的训练数据（块）计算学习，并更新对应的参数。**负责干活和更新参数。**
  - server采用分布式存储的方式，各自存储全局参数的一部分，并作为服务方接受计算节点的参数查询和更新请求。**负责存储参数和同步参数。**



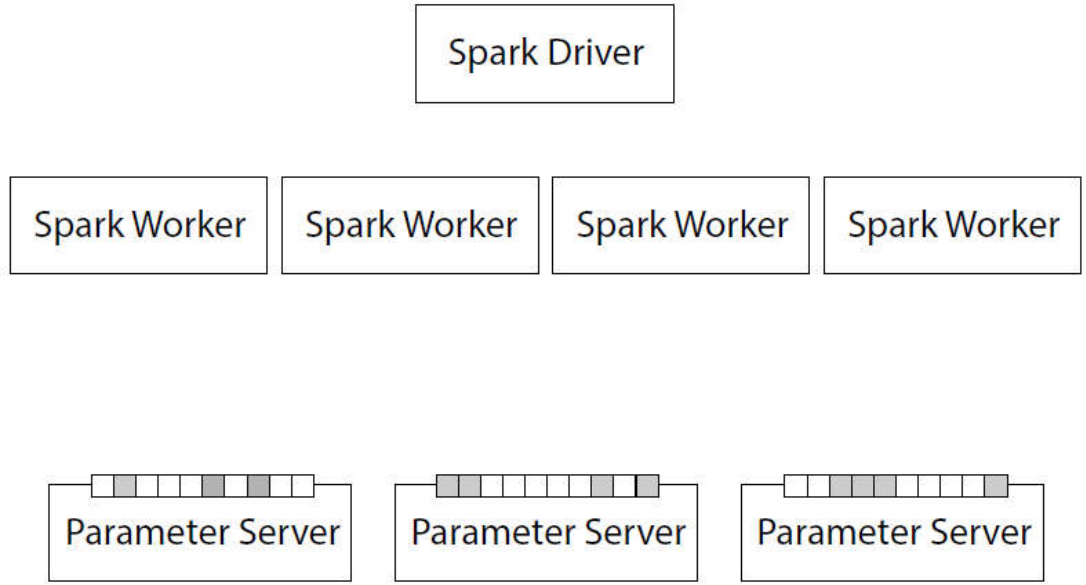
2.2.1 参数服务器

- 计算节点上进行并行计算，分配任务时，会将数据拆分给多个worker。把模型拆分给多个server



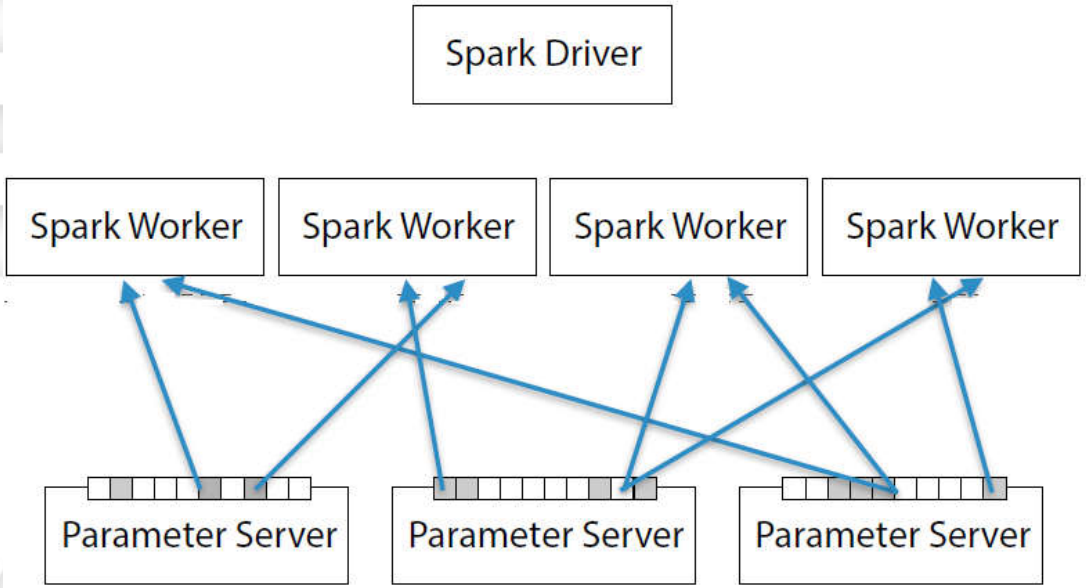
2.2.1 参数服务器

- 计算节点上进行并行计算，分配任务时把模型拆分给多个server
- 把训练数据拆分给多个worker。



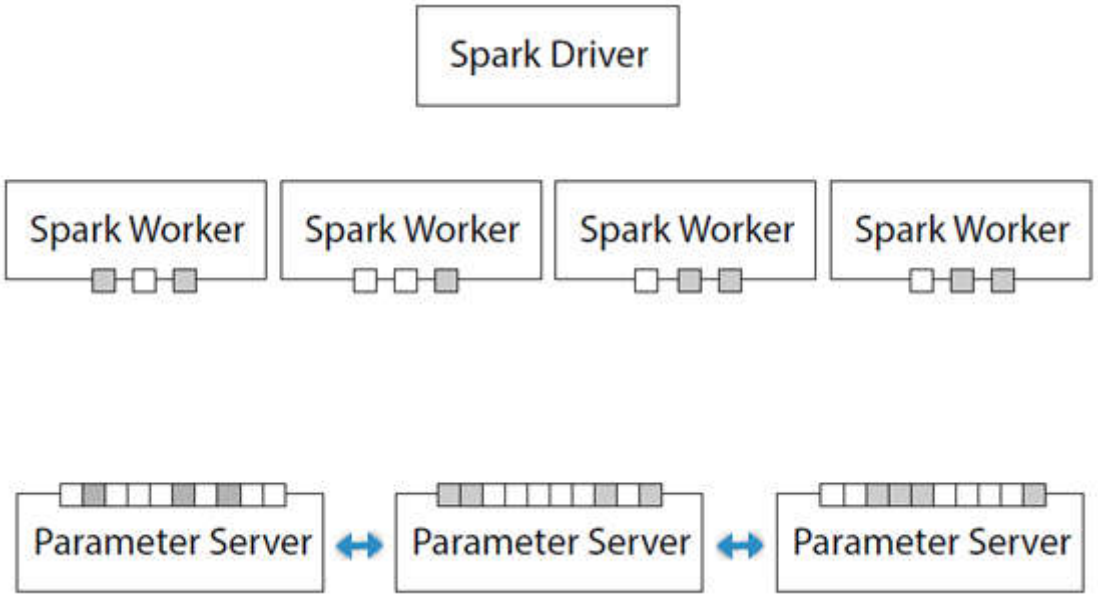
2.2.1 参数服务器

- Worker在训练前，从server调取本地训练需要的参数
  - Pull：将server上的参数的远程更新载入到本地client



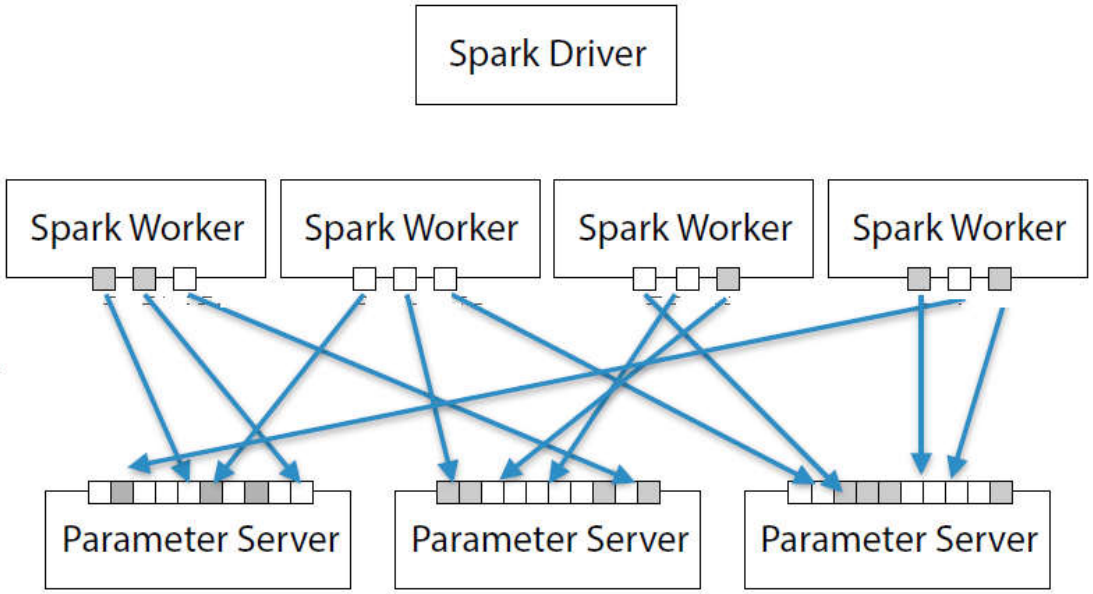
2.2.1 参数服务器

- 单个worker节点就学习本地数据。
- server协同其他server调取下轮worker需要的参数



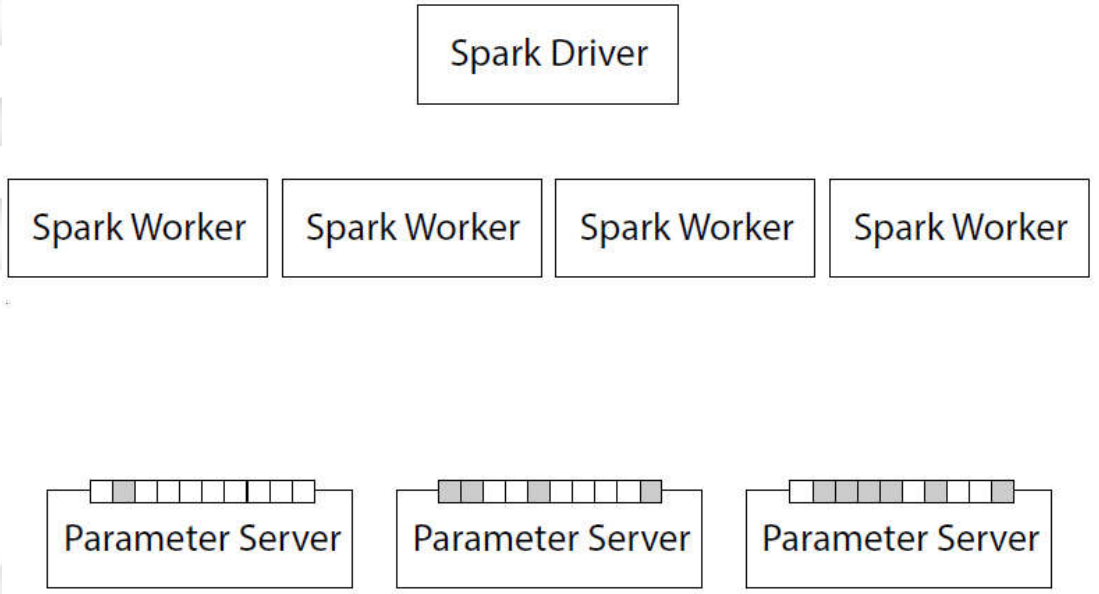
2.2.1 参数服务器

- 学习完毕把参数的更新梯度上传给对应的参数服务节点进行更新
  - Push：将client本地数据修改发送给server节点



2.2.1 参数服务器

- 参数服务器得到计算节点传过来的局部更新，汇总后更新本地数据



2.2.2 参数更新方法

- 参数平均法 vs. 更新式方法

- 参数平均法:各个工作节点的参数均值作为全局参数的更新值
- 更新式方法：各个节点将参数的梯度值提交给参数管理节点，由参数管理节点根据规则进行汇总更新

2.2.2参数平均法

- 参数平均是最简单的一种数据并行化，参数平均法在数学意义上等同于单机训练法
  - 假设有n个工作节点，每个节点处理m个样本，总共对n*m个样本求均值，学习率设为α, 权重更新方程为：

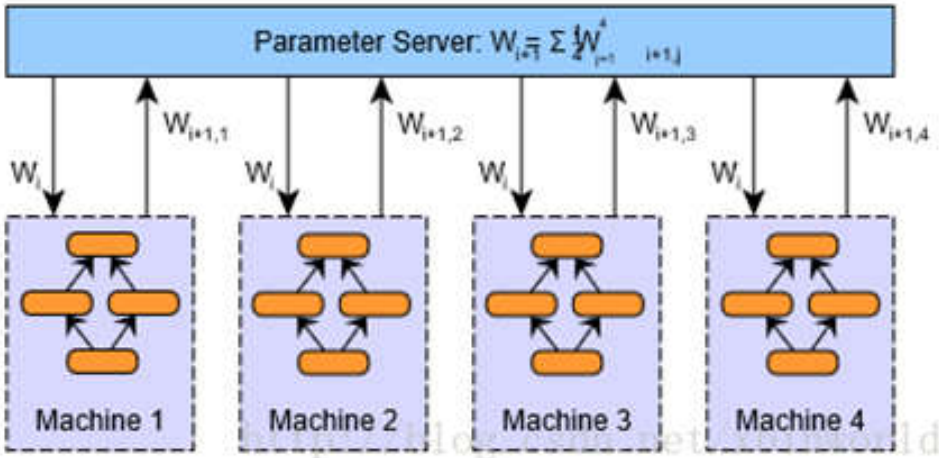
$$W_{i+1} = W_i - \frac{\alpha}{nm}\sum_{j=1}^{nm}\frac{\partial L^j}{\partial W_i}$$

  - 如果分布到n个节点，每个节点进行m个样本学习，则

$$\begin{aligned} W_{i+1} &= \frac{1}{n}\sum_{w=1}^{n} W_{i+1,w} \\ &= \frac{1}{n}\sum_{w=1}^{n}\left(W_i - \frac{\alpha}{m}\sum_{j=(w-1)m+1}^{wm}\frac{\partial L^j}{\partial W_i}\right) \\ &= W_i - \frac{\alpha}{nm}\sum_{j=1}^{nm}\frac{\partial L^j}{\partial W_i} \end{aligned}$$

2.2.2参数平均法

- 参数平均法的分布式训练过程如下所示：
  - W表示神经网络模型的参数（权重值和偏置值）
  - 下标表示参数的更新版本，需要在各个工作节点加以区分。



第4步中，每个节店将自己的参数送个参数服务器求均值。

2.2.2参数平均法

- 应该如何求平均值？
  - 最简单的办法就是每一次迭代之后进行参数平均：额外开销非常巨大；网络通信和同步的开销也许就能抵消额外机器带来的效率收益
  - 通常选择一个大于1的平均周期
    - 周期太短：网络通信开销大
    - 周期太长：各个节点的局部参数多燕华，求均值之后模型效果差
  - 目前没有结论性的回答，经验表明：建议平均的周期为每10~20个minibatch计算一次可以去掉比较好的效果

2.2.2 更新式方法

- 与参数平均法的区别：在于相对于在工作节点与参数服务器之间传递参数，更新式方法只传递更新信息（即梯度和冲量等等）
- 参数更新形式为：

$$\Delta W_{i,j} = \alpha \nabla L_j$$

$$W_{i+1} = W_i - \frac{1}{n}\sum_{j=1}^{N} \Delta W_{i,j}$$

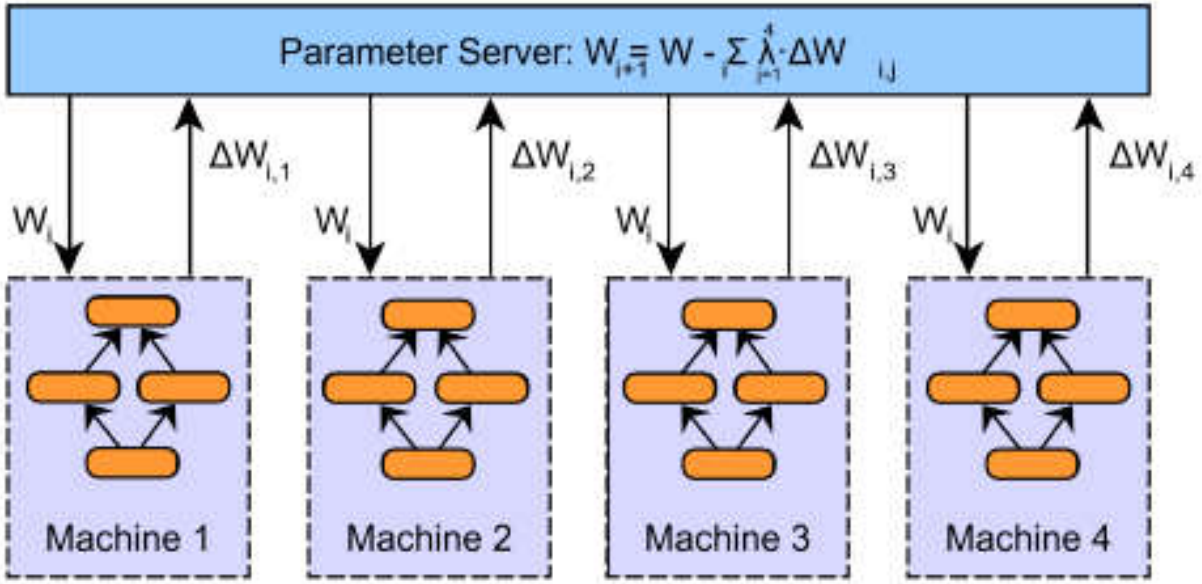
$$= \frac{1}{n}\sum_{j=1}^{n} W_i - \alpha \nabla L_j$$

$$= \frac{1}{n}\sum_{j=1}^{n} W_{i,j}$$

当采用同步方式更新时，参数平均法等价于基于更新的数据并行方法

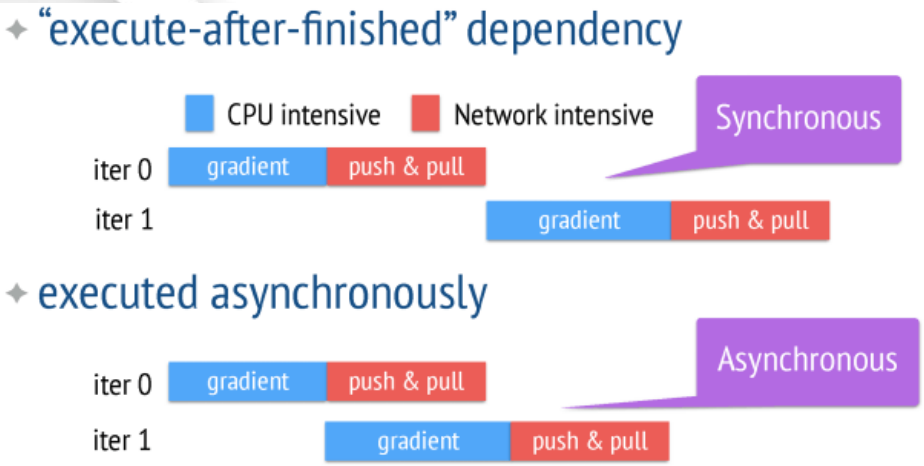
2.2.2 更新式方法

- 参数更新过程
  - 每个工作节点向参数服务器节点发送更新信息，而不是完整的参数



2.2.3 参数同步模式

- 同步更新
  - 每一次迭代之后，所有的工作节点与参数服务器进行参数的同步与更新，然后进行下一次迭代
- 异步更新
  - 系统没有统一的参数更新设置，各个工作节点把可以根据任务执行速度与参数服务器进行参数同步



2.2.3 参数同步模式

- **同步模式**
  - **系统可扩展性弱，对于云计算等平台更是如此**
  - **系统的收敛性可以得到保障**
  - **对于HPC这种同构机器，同步模式使用越来越频繁，可以约定一定迭代次数之后进行系统参数同步**
- **异步模式**
  - 提高系统的效率（因为节省了很多等待的过程）
  - 缺点就是容易降低算法的收敛速率
  - 需要对系统性能和算法收敛速率进行平衡折中
    - 算法对于参数非一致性的敏感度；
    - 训练数据特征之间的关联度；
    - 硬盘的存储容量；

2.2.4 参数服务器特点优势

- **特点**
  - 易用：全局参数表现为**向量和矩阵**，比传统的key/value格式高效，提供的线性代数的数据类型都具有高性能的多线程库
  - 高效：节点间的消息交流可以是**异步**的，参数协同也不会阻断计算操作
  - 弹性可扩展：无需重启已运行的框架可以添加新的节点；框架包括一个**分布式的hash表**来支持新节点的动态添加
  - 容错性：**冗余存储**是的节点故障后可以快速回复；worker**相互独立**，出现故障可以快速重启

主要内容

- 回顾：深度学习训练过程
- 1. 为什么要用分布式机器学习？
- 2. 并行与分布式机器学习方法
- 3. Tensorflow并行与分布处理
- 4. 实例分析：Tensorflow并行与分布处理案例

3.1 Tensorflow

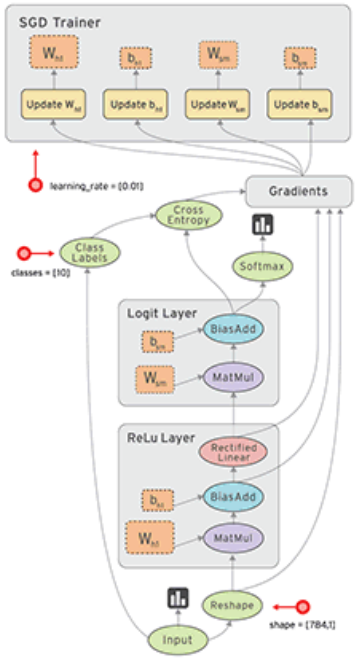
- Tensorflow 是google 开发的第二点分布式机器学习平台，它是一个采用计算图形式表述数值计算的编程系统。

- Tensoflow即tensor flow，张量 流动，张量即数组。

- Tensorflow三个基本概念：计算图(tf.Graph) 、张量(tf.Tensor)、 会话(tf.Session)
  - 计算图是计算模型，每个节点是一个运算，边表示数据传递关系，数据是张量

3.1 Tensorflow

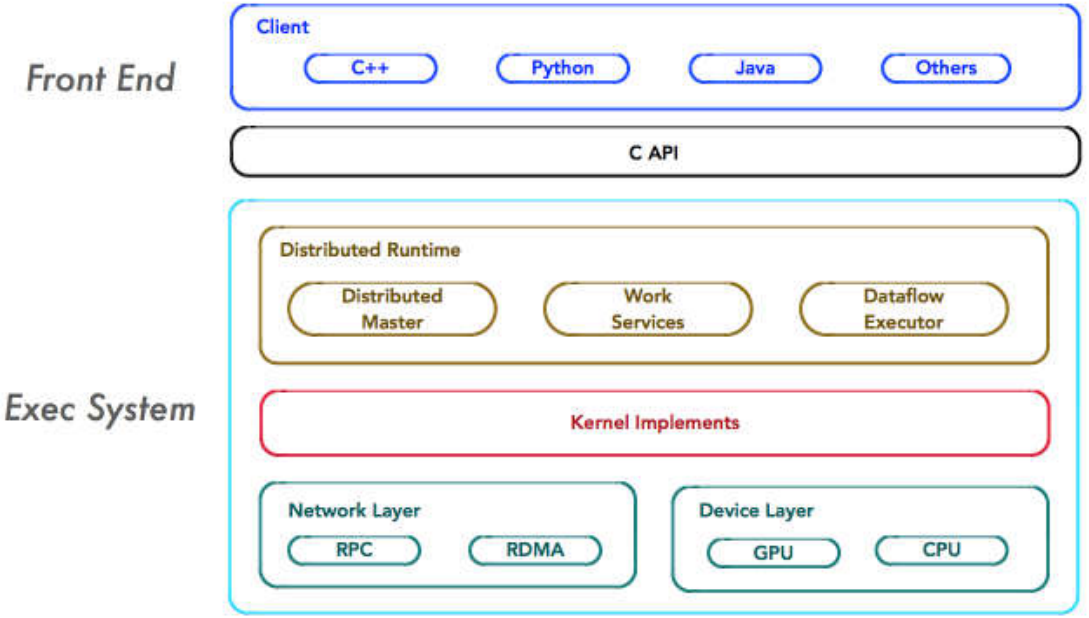
- TensorFlow基于数据流图，用于大规模分布式数值计算的开源框架。节点表示某种抽象的计算，边表示节点之间相互联系的张量。



3.1 Tensorflow 基本架构

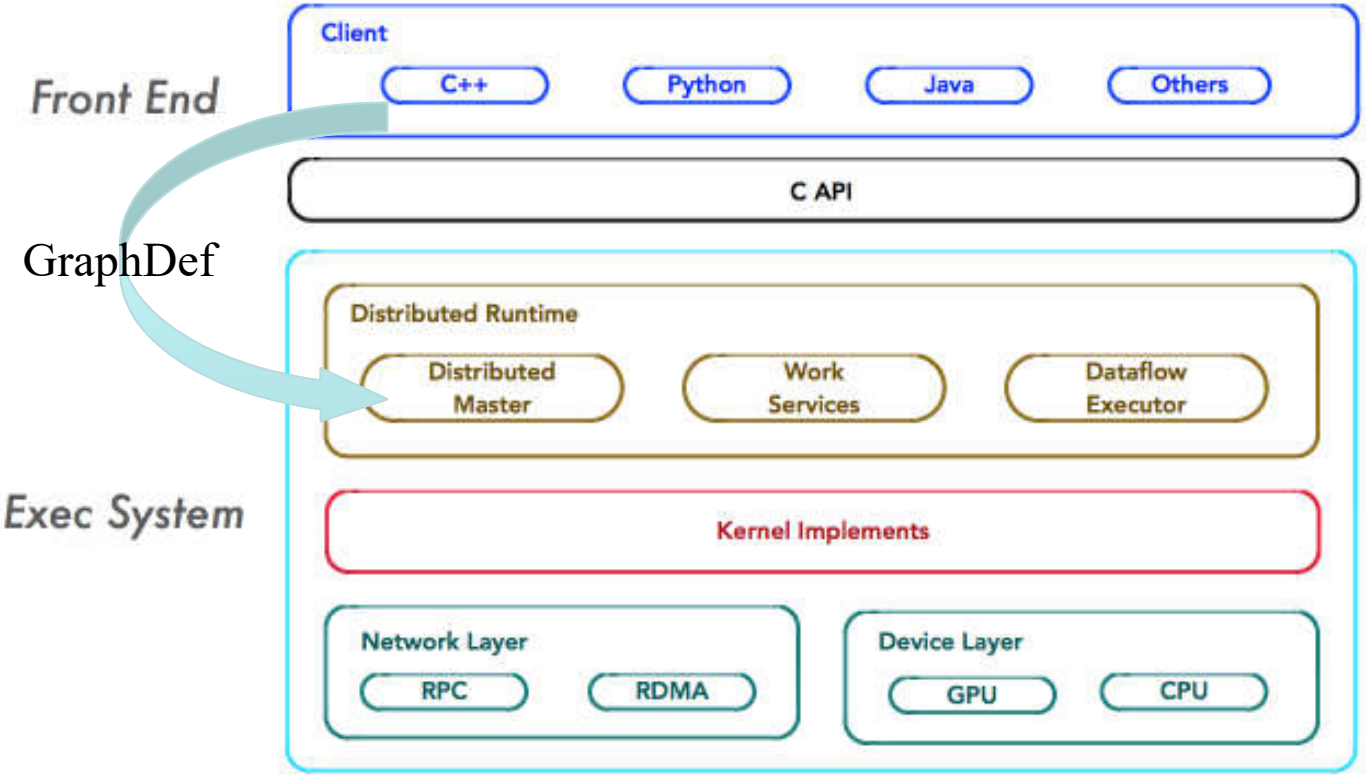
- TensorFlow的系统结构以C API为界，将整个系统分为：
  - 前端系统：提供编程模型，负责构造计算图；
  - 后端系统：提供运行时环境，负责执行计算图。

  - Client
  - Distributed Master
  - Work Service
  - Kernel Implementations



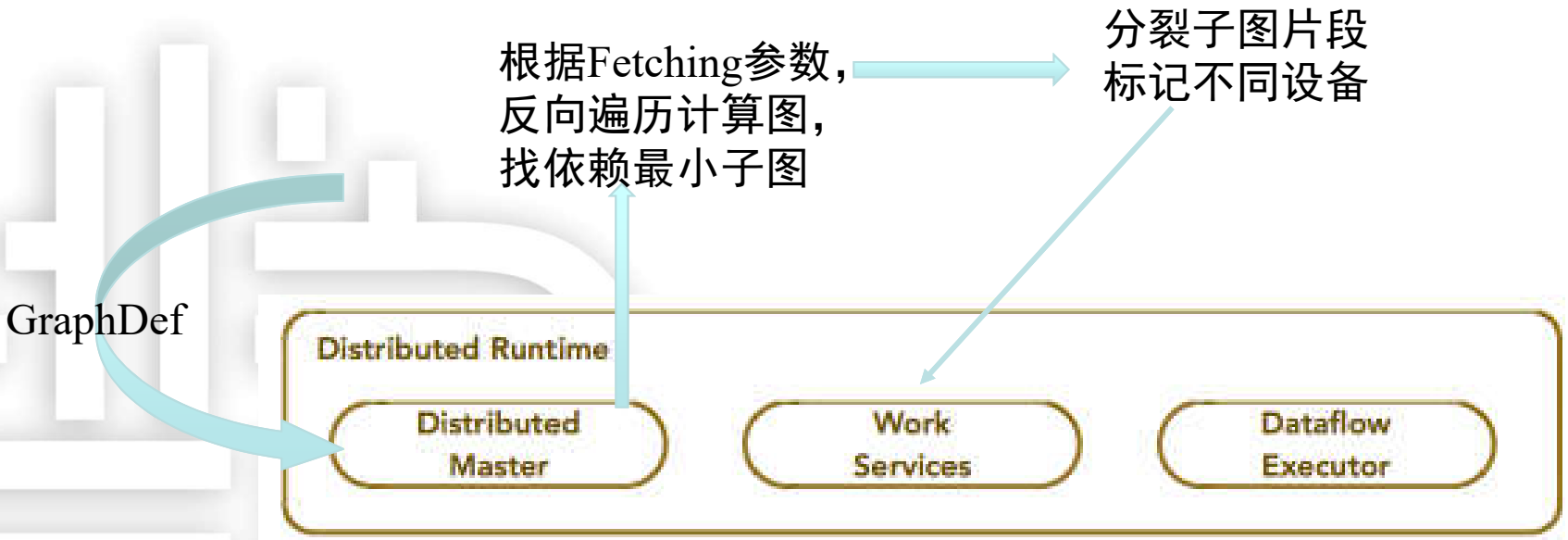
3.1 Tensorflow 基本架构

- 运行机制：



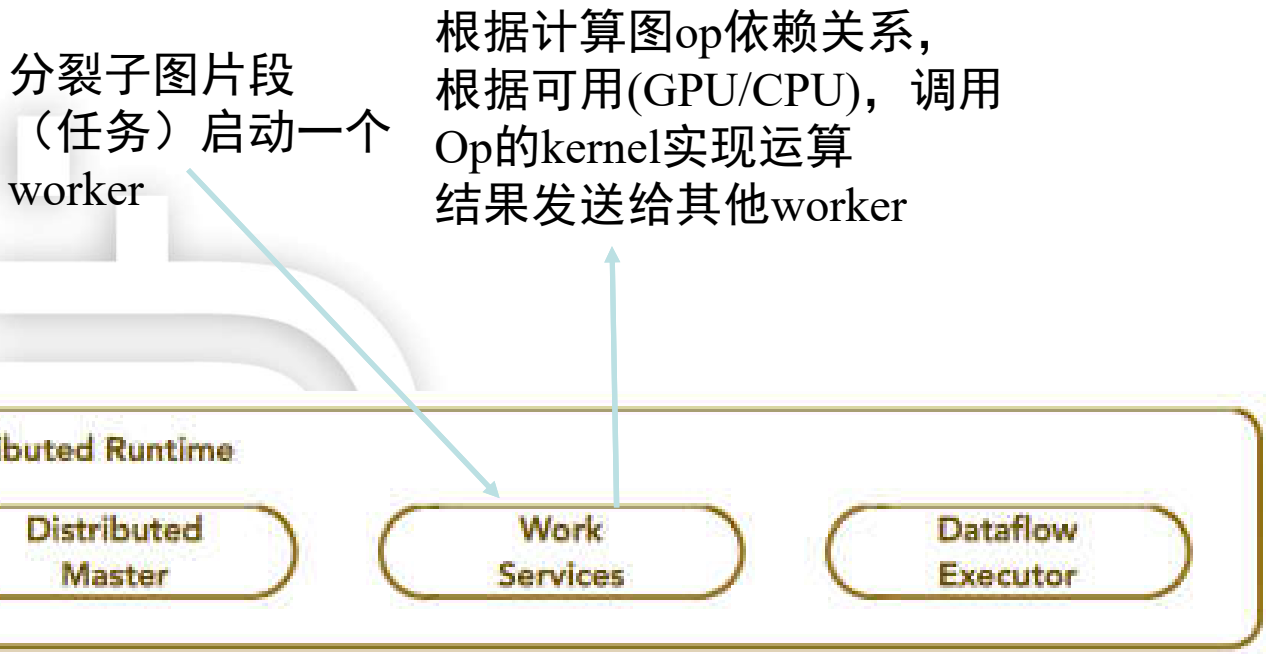
3.1 Tensorflow 基本架构

- 计算图的解析与任务分配：



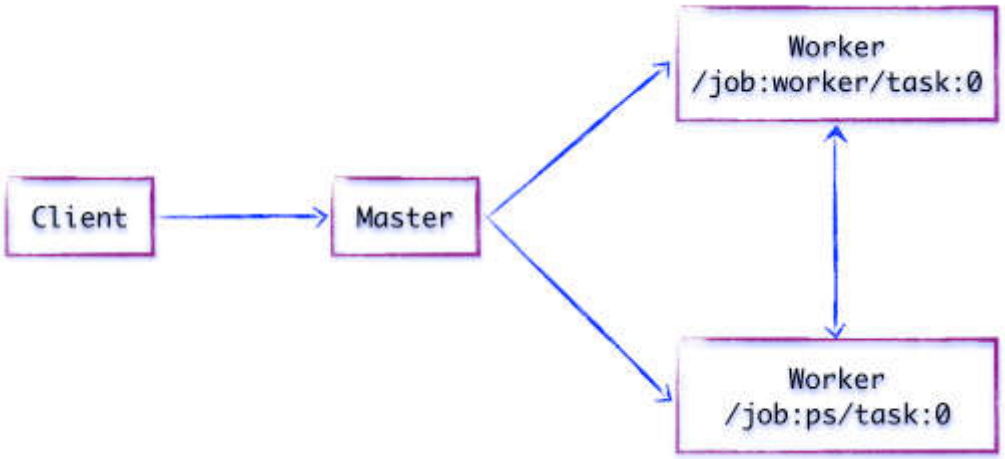
3.1 Tensorflow 基本架构

- 反对法



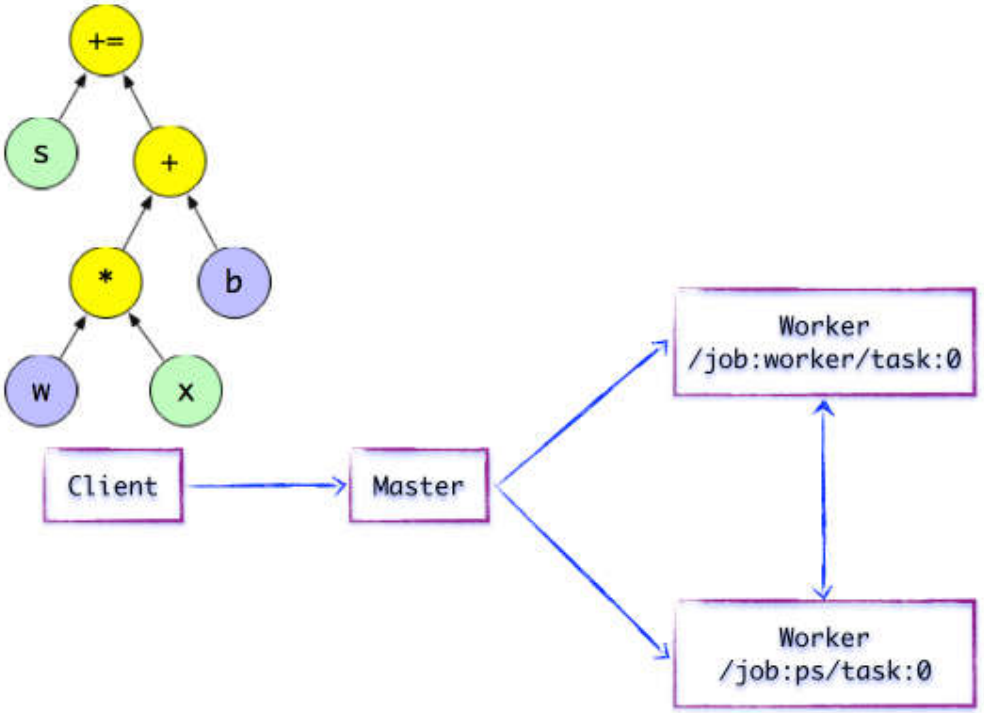
3.1 Tensorflow 基本架构

- 组件交互过程
  - /job:ps/task:0: 负责模型参数的存储和更新
  - /job:worker/task:0: 负责模型的训练或推理



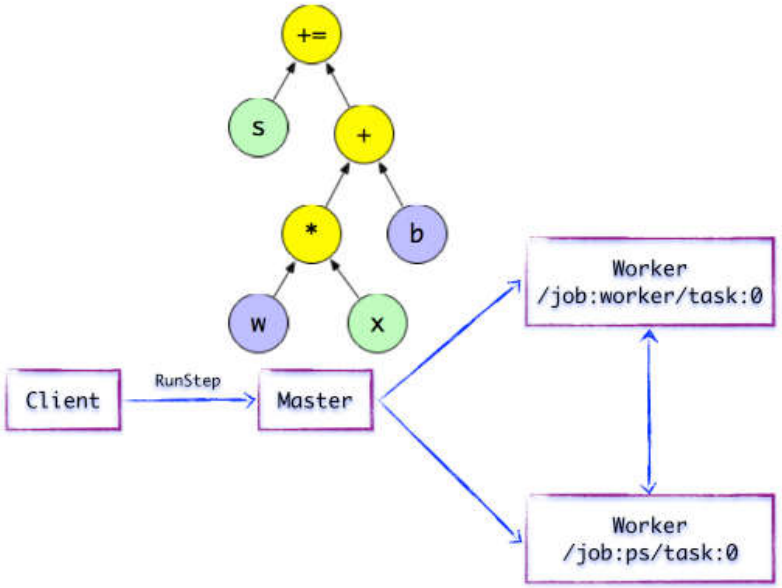
3.1 Tensorflow计算图的运行机制

- Client构建了一个简单计算图。它首先将w与x进行矩阵相乘，再与截距b按位相加，最后更新至s



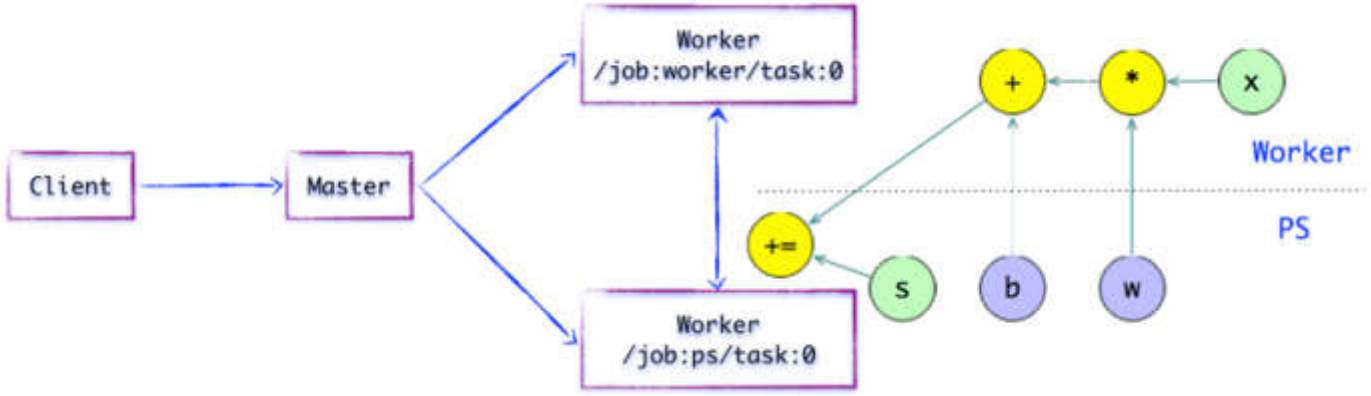
3.1 Tensorflow计算图的运行机制

- Distributed Master开始执行计算子图。在执行之前，Distributed Master会实施一系列优化技术，例如公共表达式消除，常量折叠等。随后，Distributed Master负责任务集的协同，执行优化后的计算子图。



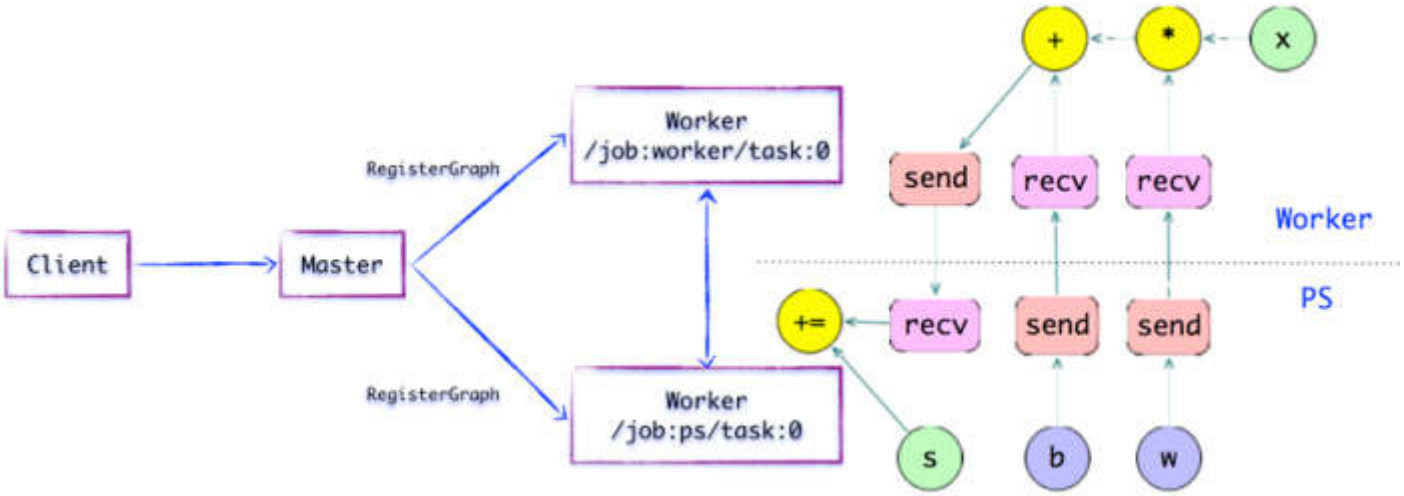
3.1 Tensorflow计算图的运行机制

- 存在一种合理的子图片段划分算法。Distributed Master将模型参数相关的OP进行分组，并放置在PS任务上。其他OP则划分为另外一组，放置在Worker任务上执行。



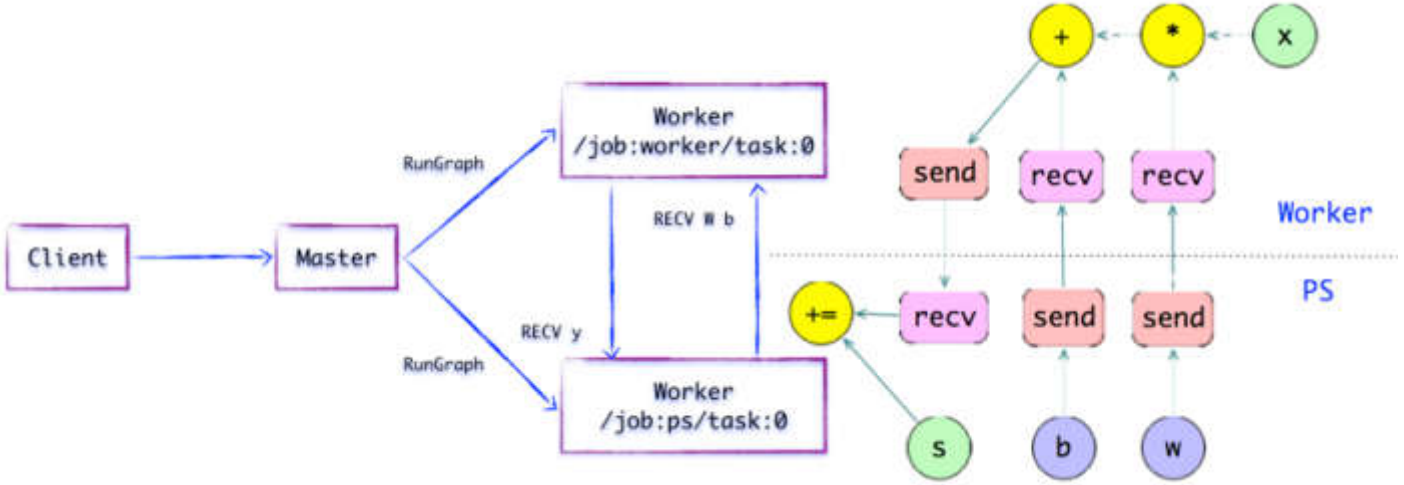
3.1 Tensorflow计算图的运行机制

- 如果计算图的边被任务节点分割，Distributed Master将负责将该边进行分裂，在两个分布式任务之间插入SEND和RECV节点，实现数据的传递



3.1 Tensorflow计算图的运行机制

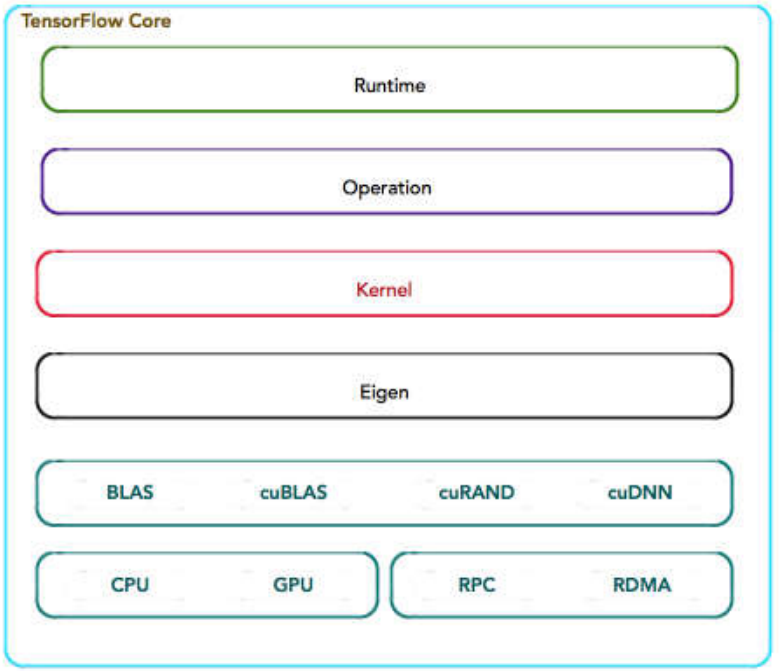
- Worker Service派发OP到本地设备，执行Kernel的特定实现。它将尽最大可能地利用多CPU/GPU的处理能力，并发地执行Kernel实现



3.1 Tensorflow计算图的运行机制

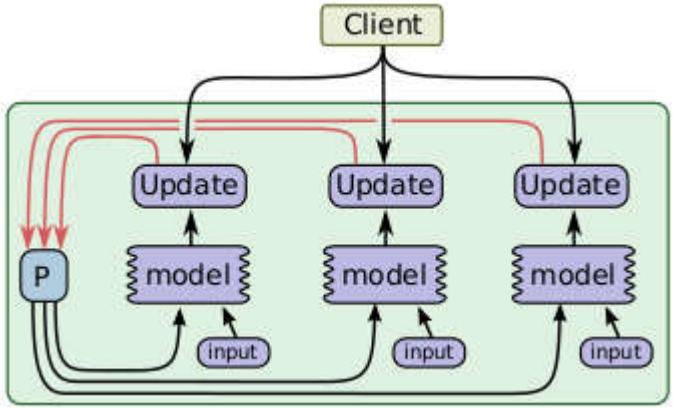
- 在运行时，运行时根据本地设备的类型，为OP选择特定的Kernel实现，完成该OP的计算。

大多数Kernel基于Eigen::Tensor实现。Eigen::Tensor是一个使用C++模板技术，为多核CPU/GPU生成高效的并发代码。但是，TensorFlow也可以灵活地直接使用cuDNN实现更高效的Kernel



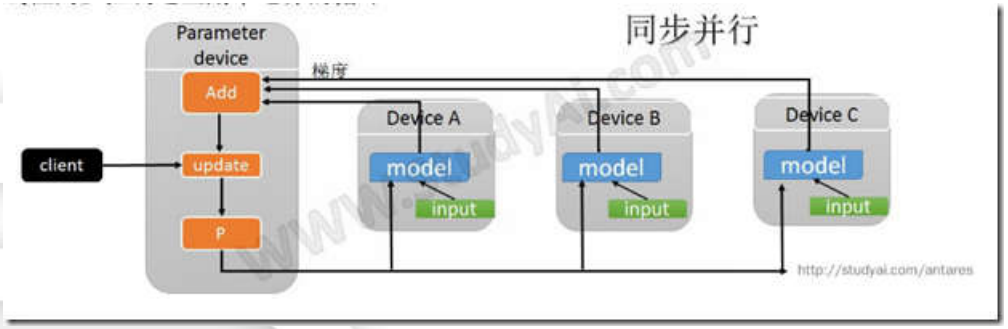
3.2 Tensorflow 并行与分布处理

- tensorFlow中主要包括了三种不同的并行策略
  - 数据并行
  - 模型并行（不同设备跑不同模型子图）
  - 流水线并行（同一设备上，将计算做成流水）
    - 此并行方式主要针对在同一个设备中并发实现模型的计算，在计算一批简单的样例时，允许进行“填充间隙”，这可以充分利用空闲的设备资源。

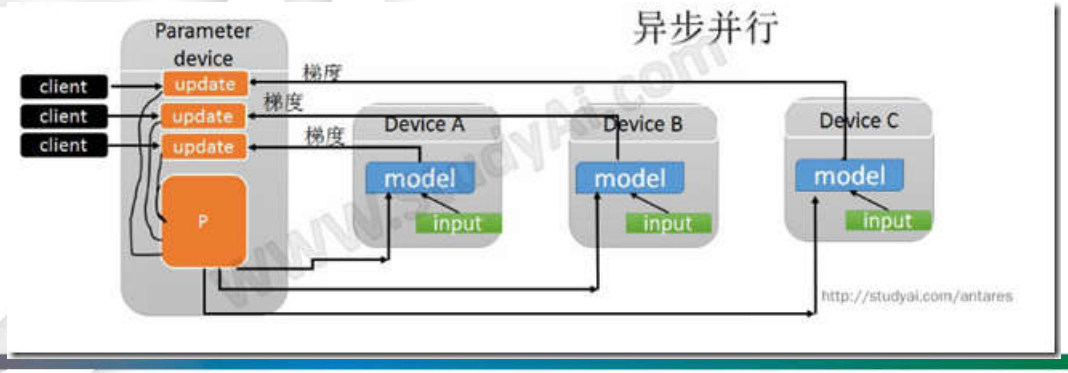


3.2 Tensorflow 并行与分布处理

- 数据并行
  - 同步并行：1个线程管更新



  - 异步并行：多个线程控制梯度计算，异步更新模型参数



3.2 Tensorflow 代码构建

- 如何处理数据
- 如何构建计算图
- 如何计算梯度
- 如何Summary，如何save模型参数
- 如何执行计算图

参数设置

- 如果电脑有多个GPU，tensorflow默认全部使用。使用部分GPU，设置CUDA_VISIBLE_DEVICES

```
CUDA_VISIBLE_DEVICES=1 python my_script.py #只使用GPU1
CUDA_VISIBLE_DEVICES=0,1 python my_script.py #使用GPU0,GPU1
```

Environment Variable Syntax	Results
CUDA_VISIBLE_DEVICES=1	Only device 1 will be seen
CUDA_VISIBLE_DEVICES=0,1	Devices 0 and 1 will be visible
CUDA_VISIBLE_DEVICES="0,1"	Same as above, quotation marks are optional
CUDA_VISIBLE_DEVICES=0,2,3	Devices 0, 2, 3 will be visible; device 1 is masked
CUDA_VISIBLE_DEVICES=""	No GPU will be visible

- 举例：
- import os
- os.environ["CUDA_VISIBLE_DEVICES"] = "2"

使用多个GPU

```
# 新建一个 graph.
c = []
for d in ['/gpu:2', '/gpu:3']:
  with tf.device(d):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
    c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
  sum = tf.add_n(c)
# 新建session with log_device_placement并设置为True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# 运行这个op.
print sess.run(sum)
```

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K20m, pci bus
id: 0000:02:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: Tesla K20m, pci bus
id: 0000:03:00.0
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: Tesla K20m, pci bus
id: 0000:83:00.0
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: Tesla K20m, pci bus
id: 0000:84:00.0
Const_3: /job:localhost/replica:0/task:0/gpu:3
Const_2: /job:localhost/replica:0/task:0/gpu:3
MatMul_1: /job:localhost/replica:0/task:0/gpu:3
Const_1: /job:localhost/replica:0/task:0/gpu:2
Const: /job:localhost/replica:0/task:0/gpu:2
MatMul: /job:localhost/replica:0/task:0/gpu:2
AddN: /job:localhost/replica:0/task:0/cpu:0
[[  44.   56.]
 [  98.  128.]]
```

3.2 Tensorflow 代码构建

- 如何处理数据
  - 写一个将数据分成训练集,验证集和测试集的函数

train_set, valid_set, test_set = split_set(data)

  - 最好写一个管理数据的对象，将原始数据转化成mini_batch

```
class DataManager(object):
  #raw_data为train_set, valid_data或test_set
  def __init__(self, raw_data, batch_size):
    self.raw_data = raw_data
    self.batch_size = batch_size
    self.epoch_size = len(raw_data)/batch_size
    self.counter = 0 #监测batch index
  def next_batch(self):
    ...
    self.counter += 1
    return batched_x, batched_label, ...
```

3.2 Tensorflow 代码构建

- 构建计算图
  - 计算图的构建在Model类中的__init__()中完成,并设置is_training参数

```
class Model(object):
  def __init__(self, is_training, config, scope,...):#scope可以使你正确的summary
    self.is_training = is_training
    self.config = config
    #placeholder:用于feed数据
    # 一个train op
    self.graph(self.is_training) #构建图
    self.merge_op = tf.summary.merge(tf.get_collection(tf.GraphKeys.SUMMARIES,scope))
  def graph(self,is_training):
    ...
    #定义计算图
    self.predict = ...
    self.loss = ...
```

3.2 Tensorflow 代码构建

- **run_epoch函数**

```
#eval_op是用来指定是否需要训练模型，需要的话，传入模型的eval_op
#draw_ata用于接收 train_data,valid_data或test_data
def run_epoch(raw_data , session, model, is_training_set, ...):
  data_manager = DataManager(raw_data, model.config.batch_size)

  #通过is_training_set来决定fetch哪些Tensor
  #add_summary, saver.save(....)
```

3.2 Tensorflow 代码构建

- **组织main函数**
  - 分解原始数据为train，valid，test
  - 设置默认图
  - 建图 trian, test 分别建图
  - 一个或多个Saver对象，用来保存模型参数
  - 创建session， 初始化变量
  - 一个summary.FileWriter对象，用来将summary写入到硬盘中
  - run epoch

3.2 Tensorflow 代码构建

- 组织Main函数

```
def main(_):
  if not FLAGS.data_path:
    raise ValueError("Must set --data_path to PTB data directory")

  raw_data = reader.ptb_raw_data(FLAGS.data_path)
  train_data, valid_data, test_data, _ = raw_data

  config = get_config()
  eval_config = get_config()
  eval_config.batch_size = 1
  eval_config.num_steps = 1

  with tf.Graph().as_default():
    initializer = tf.random_uniform_initializer(-config.init_scale,
                                                config.init_scale)

    with tf.name_scope("Train"):
      train_input = PTBInput(config=config, data=train_data, name="TrainInput")
      with tf.variable_scope("Model", reuse=None, initializer=initializer):
        m = PTBModel(is_training=True, config=config, input_=train_input)
      tf.contrib.deprecated.scalar_summary("Training Loss", m.cost)
      tf.contrib.deprecated.scalar_summary("Learning Rate", m.lr)
```

3.2 Tensorflow 多GPU代码构建

- 如何实现multi_gpu_model函数

```
def multi_gpu_model(num_gpus=1):
  grads = []
  for i in range(num_gpus):
    with tf.device("/gpu:%d"%i):
      with tf.name_scope("tower_%d"%i):
        model = Model(is_training, config, scope)
        # 放到collection中，方便feed的时候取
        tf.add_to_collection("train_model", model)
        grads.append(model.grad) #grad 是通过tf.gradients(loss, vars)求得
        #以下这些add_to_collection可以直接在模型内部完成。
        # 将loss放到 collection中， 方便以后操作
        tf.add_to_collection("loss",model.loss)
        #将predict放到collection中，方便操作
        tf.add_to_collection("predict", model.predict)
        #将 summary.merge op放到collection中，方便操作
        tf.add_to_collection("merge_summary", model.merge_summary)
        # ...
  with tf.device("cpu:0"):
    averaged_gradients = average_gradients(grads)# average_gradients后面说明
    opt = tf.train.GradientDescentOptimizer(learning_rate)
    train_op=opt.apply_gradients(zip(average_gradients,tf.trainable_variables()))

  return train_op
```

3.2 Tensorflow 多GPU代码构建

- 如何feed data

```
1
2  def generate_feed_dic(model, feed_dict, batch_generator):
3    x, y = batch_generator.next_batch()
4    feed_dict[model.x] = x
5    feed_dict[model.y] = y
```

3.2 Tensorflow 多GPU代码构建

- 如何实现run_epoch

```
#这里的scope是用来区别 train 还是 test
def run_epoch(session, data_set, scope, train_op=None, is_training=True):
  batch_generator = BatchGenerator(data_set, batch_size)
  ...
  ...
  if is_training and train_op is not None:
    models = tf.get_collection("train_model")
    # 生成 feed_dict
    feed_dic = {}
    for model in models:
      generate_feed_dic(model, feed_dic, batch_generator)
    #生成fetch_dict
    losses = tf.get_collection("loss", scope)#保证了在 test的时候，不会fetch train的loss
    ...
    ...
```

3.2 Tensorflow 多GPU代码构建

- 如何训练：

```
opt = tf.train.MomentumOptimizer(lr,0.9,use_nesterov=True,use_locking=True)

# Calculate the gradients for each model tower.
tower_grads = []
with tf.variable_scope(tf.get_variable_scope()):
    for i in xrange(FLAGS.num_gpus):
        with tf.device('/gpu:%d' % i):
            with tf.name_scope(
                            '%s_%d' % (TOWER_NAME, i)) as scope:
    ...
                # Calculate the gradients for the batch of data on this
                # MNIST tower.
                grads = opt.compute_gradients(loss, gate_gradients=0)

                # Keep track of the gradients across all towers.
                tower_grads.append(grads)

  # We must calculate the mean of each gradient. Note that this is the
  # synchronization point across all towers.
  grads = average_gradients(tower_grads)
```

3.2 Tensorflow 多GPU代码构建

- **如何训练：**
- train_op = opt.apply_gradients(grads, global_step=global_step)
- …
- _, loss_value = sess.run([train_op, loss])

```
with tf.variable_scope(tf.get_variable_scope()):
  for i in xrange(FLAGS.num_gpus):
    with tf.device('/gpu:%d' % i):
      with tf.name_scope('%s_%d' % (cifar10.TOWER_NAME, i)) as scope:
        # Dequeues one batch for the GPU
        image_batch, label_batch = batch_queue.dequeue()
        # Calculate the loss for one tower of the CIFAR model. This function
        # constructs the entire CIFAR model but shares the variables across
        # all towers.
        loss = tower_loss(scope, image_batch, label_batch)

        # Reuse variables for the next tower.
        tf.get_variable_scope().reuse_variables()

        # Retain the summaries from the final tower.
        summaries = tf.get_collection(tf.GraphKeys.SUMMARIES, scope)

        # Calculate the gradients for the batch of data on this CIFAR tower.
        grads = opt.compute_gradients(loss)

        # Keep track of the gradients across all towers.
        tower_grads.append(grads)

# We must calculate the mean of each gradient. Note that this is the
# synchronization point across all towers.
grads = average_gradients(tower_grads)

# Add a summary to track the learning rate.
summaries.append(tf.summary.scalar('learning_rate', lr))

# Add histograms for gradients.
for grad, var in grads:
  if grad is not None:
    summaries.append(tf.summary.histogram(var.op.name + '/gradients', grad))

# Apply the gradients to adjust the shared variables.
apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)
```

3.2 Tensorflow 多GPU代码构建

- **main函数 (干以下几件事情)**
  - 1. 数据处理
  - 2. 建立多GPU训练模型
  - 3. 建立单/多GPU测试模型
  - 4. 创建Saver对象和FileWriter对象
  - 5. 创建session
  - 6. run_epoch

3.2 Tensorflow 多GPU代码构建

- Main 函数

```
data_process()
with tf.name_scope("train") as train_scope:
  train_op = multi_gpu_model(..)
with tf.name_scope("test") as test_scope:
  model = Model(...)
saver = tf.train.Saver()
# 建图完毕，开始执行运算
with tf.Session() as sess:
  writer = tf.summary.FileWriter(...)
  ...
  run_epoch(...,train_scope)
  run_epoch(...,test_scope)
```

谢谢！