# Model Optimization

Taylor Guo, 2020年12月08日 @上海

目录

[Model Optimization 1](#_Toc58372869)

[Auto-Precision Scaling for Distributed Deep Learning 1](#_Toc58372870)

[提出问题： 1](#_Toc58372871)

[解决方法： 2](#_Toc58372872)

[APS算法思路与伪代码： 3](#_Toc58372873)

[技术细节： 4](#_Toc58372874)

[代码分析： 5](#_Toc58372875)

## Auto-Precision Scaling for Distributed Deep Learning

<https://arxiv.org/abs/1911.08907> <https://arxiv.org/pdf/1911.08907.pdf>

### 提出问题：

1. 分布式训练的一个主要问题是梯度同步成为瓶颈，限制了训练的规模。低精度梯度可以减少带宽。

自动精度调整(Auto Precision Scaling, APS)算法可以在以低精度浮点值传递时改善准确度。它可以改善精度的准确度而使用较低的传递成本。图像分类和分割测试中用8位浮点梯度准确率降低<0.05%。8个Nvidia GTX1080Ti训练BERT/Wikipedia可能会占用40%的时钟时间。

2. 用低精度梯度会损害推断准确率，训练无法收敛。低精度容易下溢或上溢。有很多0值和INF值，训练无法收敛。

### 解决方法：

1. APS方案：APS是一个高效的梯度传输基于网络层级的自适应算法。它可以是大批次训练在8位甚至4位符号类型收敛。

APS，网络层级自适应算法，改善低精度下的准确率；

可以在8节点的分布式系统中训练8bit的分类和分割模型；

可以在256节点的分布式系统中，8bit梯度训练ResNet-50；

可以应用于任意低精度浮点运算；

2. 其他方案：

梯度同步的优化有好几种方法:

a. 特殊的网络拓扑结构：

多组三阶段实现all-reduce操作；

通常的做法是计算前一层网络的梯度，然后一起传输所有计算好的梯度。

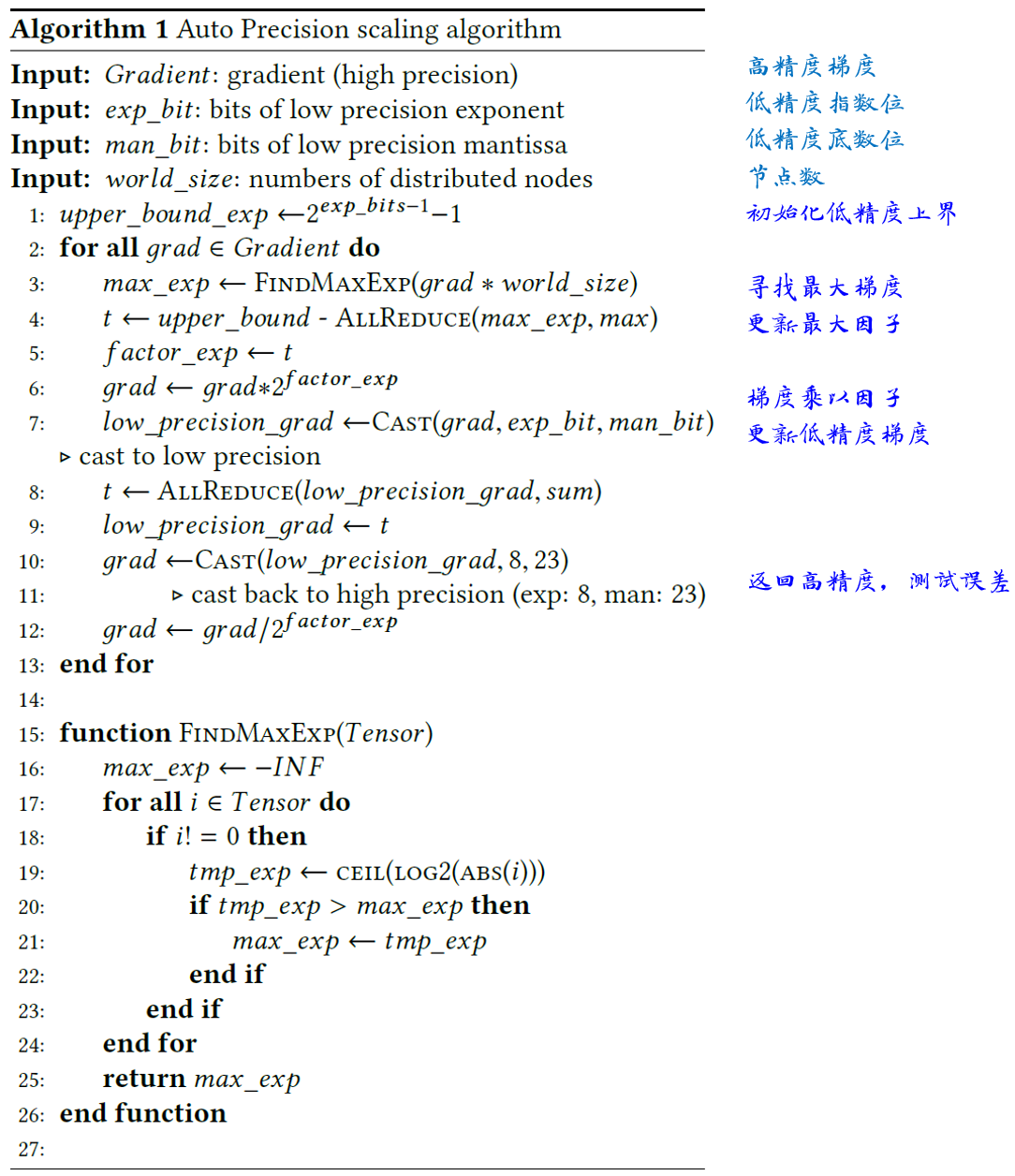
b. 梯度稀疏化

对梯度设置阈值，每次迭代只传输大于阈值的梯度；设置阈值的方法有测量每个梯度的重要性的L1范数；DGC方法累加局部梯度。

c. 梯度量化：

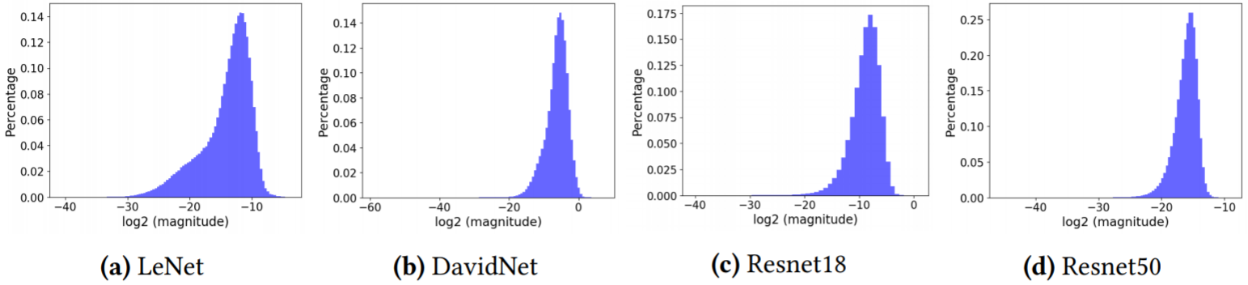
缩减损失值来缩减梯度值

### APS算法思路与伪代码：

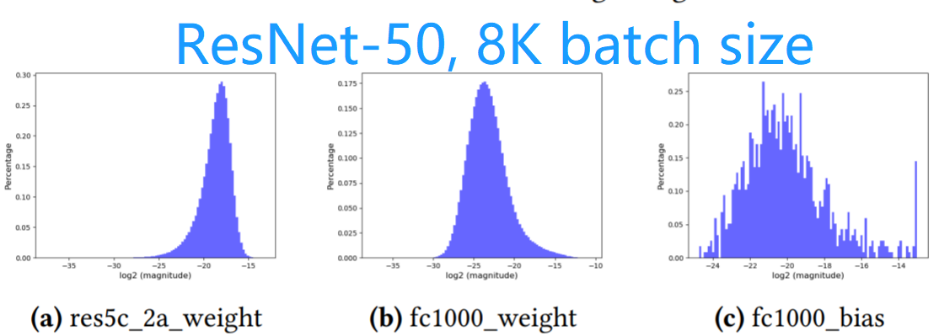


#### 1. 损失缩放的局限

需要寻找合适的损失缩放因子，不同模型的梯度分布非常不同。



相同的模型，不同层的梯度分布非常不同。

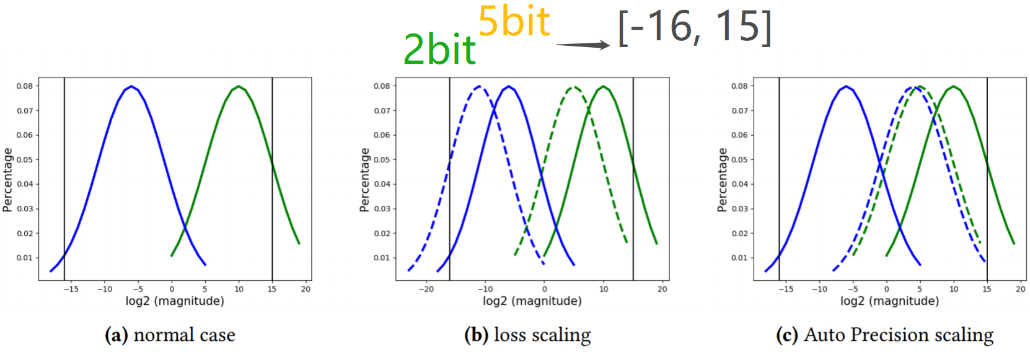


训练过程中单层梯度分布的变化，会使得损失缩放算法在实践当中极不稳定。

#### 2. 层级精度调整梯度

同步层的梯度，先计算最大绝对值梯度和阶码；然后用all-reduce在整个分布式系统上获得全局最大值；再根据这个全局最大梯度值将局部最大梯度移位变成低精度计数。n层的阶码最大值获得之后，将每层都移位为低精度。所有低精度处理完后，再移位回高精度的阶码。在实践中，同步梯度，可以将后续几层的梯度做为一个张量加速通信减少延迟。

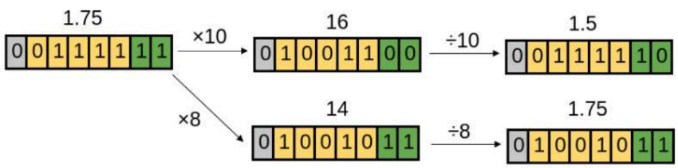
损失缩放对所有层的梯度使用固定值，手工选值防止溢出。可以避免上溢，但可能导致小值产生下溢，被置为0。APS每层用不同的值，也就是会自动选择缩放因子不会导致上下溢。



### 技术细节：

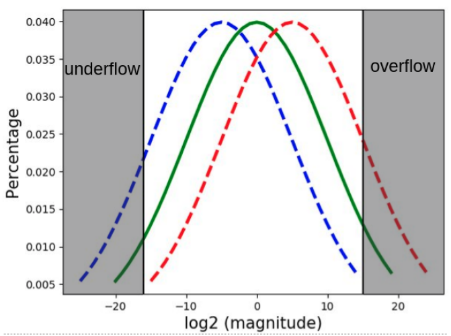
#### 1. APS使用2的幂次方作为缩放因子

可以减少舍入误差。比如下面的5位阶码，2位底数。



使用10和8作为缩放因子的区别，灰色部分是符号位，黄色部分是幂，绿色部分是底。因子是2的n次幂时，底是不变的，只是阶码发生变化。如果是10的话，阶码和底都会发生变化，可能会截断数值。乘以10或除以10都会导致舍入误差。

#### 2. 下溢或上溢的选择



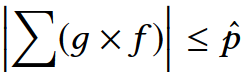
用低位来表示高位会产生下溢或者上溢。

缩放因子大于1绿色曲线会移向红色曲线，产生上溢；小于1，则移向蓝色曲线，产生下溢。但对深度神经网络来说，上溢经常比下溢危害更大。反向传播的时候，后面的网络层的梯度是基于前面的网络层的梯度来计算的。如果后面的网络层上溢，被置为INF，所有之前的网络梯度都会被置为INF。如果有INF参与运算，那么算子的输出都为INF。这样，整个训练就无法收敛，就会丢失重要信息。

如果采用对小梯度设置阈值进行丢弃算法的话，局部梯度累加超过阈值再进行通信传递。但对APS算法，我们会丢弃小的梯度值，这些值可能会下溢。比如，如果我们用8位来处理，5位阶码，2位底数，当乘以因子的时候，最大的梯度大概是215，这已经是该精度的上界了。那么，小于下界2-16就会溢出。如果采用阈值的方法，就会被大量累加，效果类似达到最大梯度。同时，阶码变大，底数不变。由于只有2位底数，那么这个值将是4倍，会导致舍入错误，小梯度应该被丢弃。在这些值里，选择最大的，就会使最小的掉到下溢范围内。

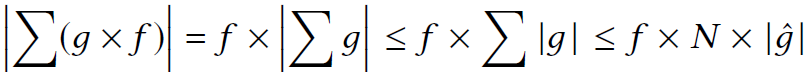
#### 3. 寻找最大缩放因子

选择最大缩放因子来避免上溢。

 ❶

g是梯度，f是放大因子， 特定精度的上界；

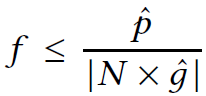
每个节点只知道局部梯度，很难得到最大因子；所以APS，放松了等式的边界；

 ❷

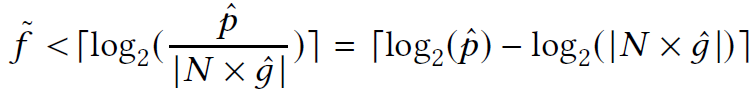
N是节点数；

比较直觉的方法就是只传递每个节点的局部最大梯度，从而获得全局最大梯度来计算因子。

为了进一步优化，❷带入❶等式又可以写为❸：

 ❸

如果只用2幂次方作为因子，❸可以变为：

那么，给定精度要求，log对数值就等于阶码，所以只需要通信这个值就可以用8位来传递阶码。

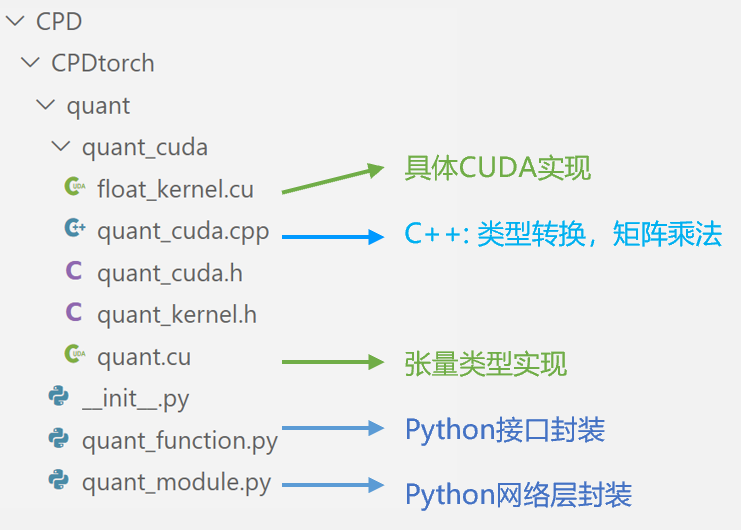
### 实验结论：

分布式训练系统中的LARS(Layer-Wise Adaptive Rate Scaling 分层自适应学习率: 权重2范数除以梯度2范数，再乘原学习率)和低精度算法是否会影响准确率：会；但APS不会。

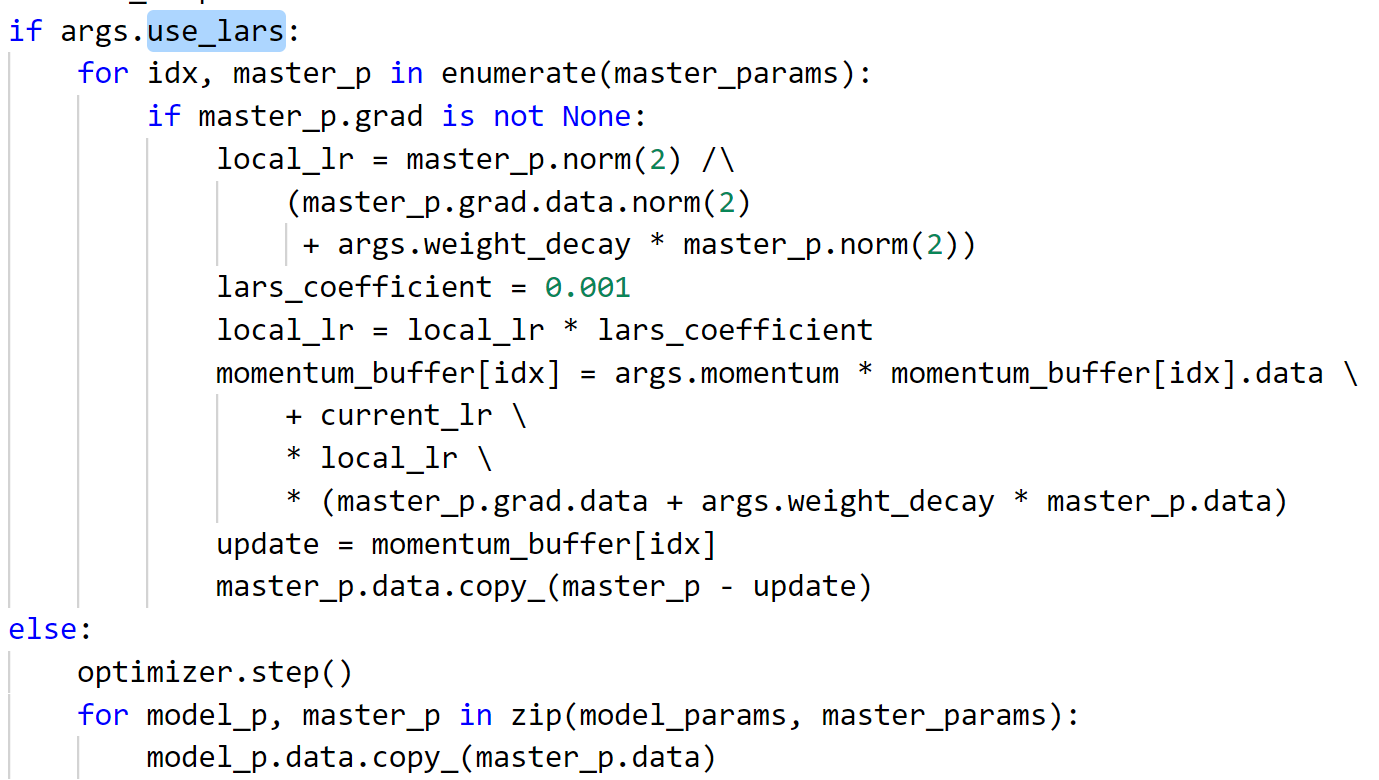
定制化低精度深度学习：

* 任意低精度 -> 阶码<=8bit; 底数<=23bit；
* Kahan summation 算法：补偿截断误差，提高低精度梯度累加更新与矩阵乘法的准确率；
* 低精度加法器计算GEMM；
* 低精度reduce/all-reduce

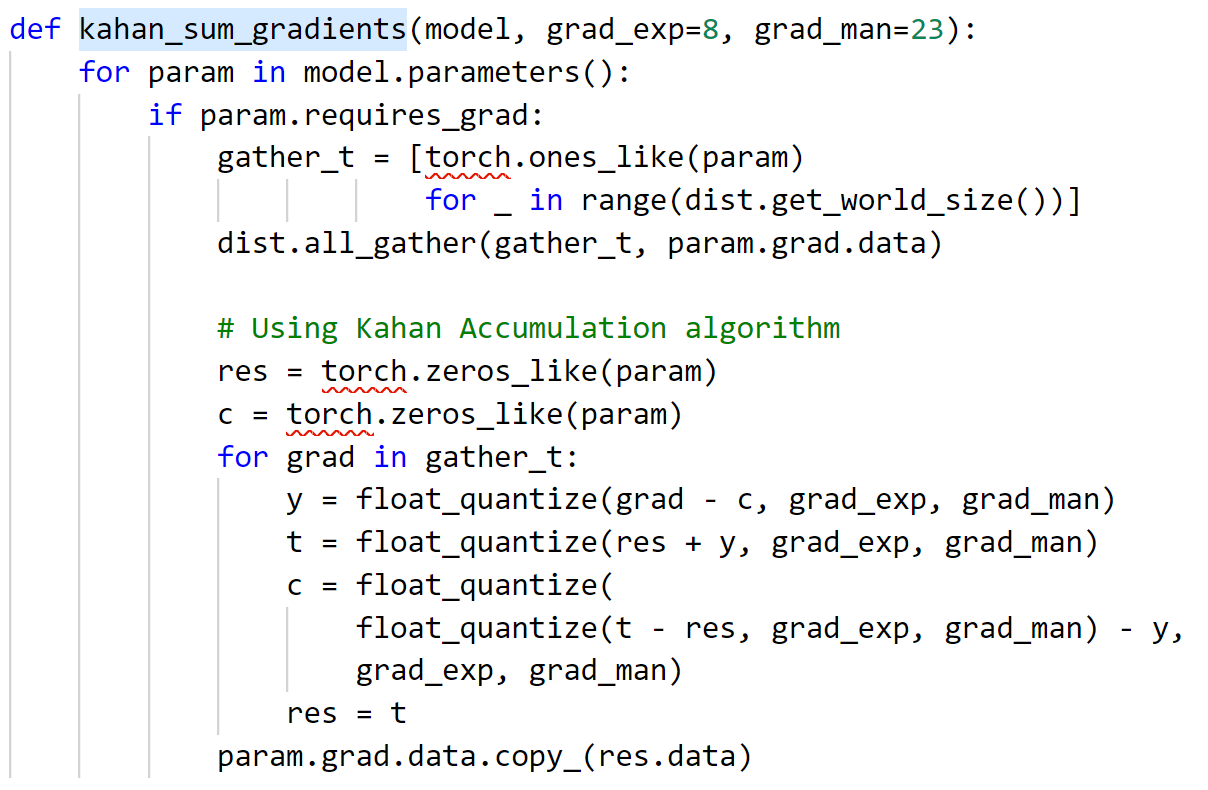
### 代码分析： [CPD Code](https://github.com/drcut/CPD)

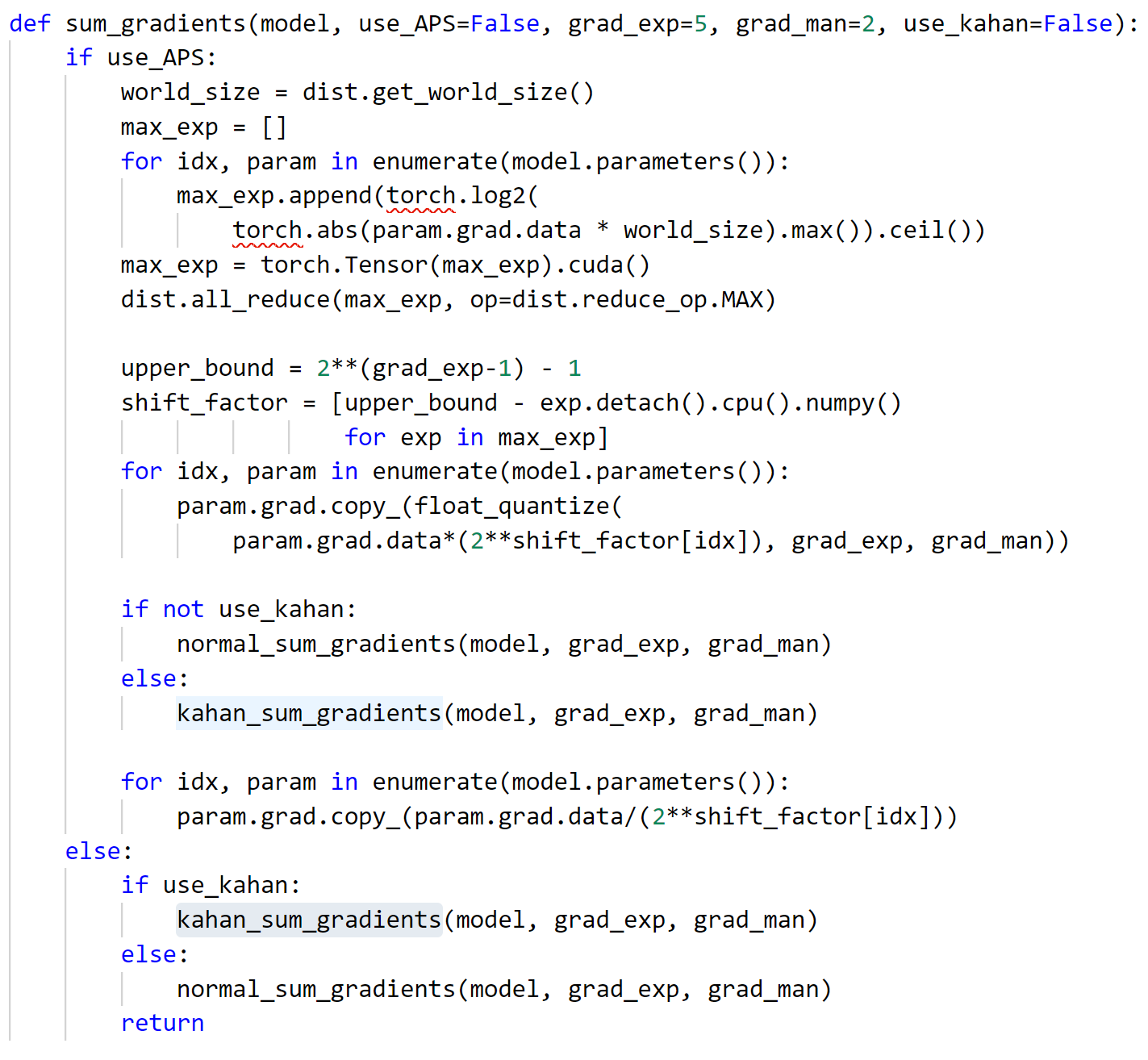


**训练过程中的LARS：**



**Kahan加法去精度误差：**





## 用Tensor Cores 在CUDA9中编程

<https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

Volta GPU架构中加入了Tensor Cores， 使得Tesla V100可以在FP32的峰值吞吐量相比P100加速12倍。Tensor Cores可以让AI编程使用混合精度获得更高吞吐量而不会牺牲精度。

主要的深度学习编程框架Tensorflow, PyTorch, MxNet, Caffe2都已经支持Tensor Cores. TensorRT也已经支持Tensor Cores.

<https://docs.nvidia.com/deeplearning/performance/pdf/Training-Mixed-Precision-User-Guide.pdf>

本文将如何使用CUDA库对应用程序编码的时候加入Tensor Cores。

### Tensor Cores是什么？

Tesla V100 Tensor Cores是可编程的矩阵乘法和累加单元，可以提供125 Tensor TFLOPS，用于训练和推断。V100含有640个Tensor Cores: 每个SM(streaming multiprocessor)中8个。Tensor Cores和关联的数据通路可以增强浮点运算吞吐量，省电。 时钟门控技术更加可以更大化地省电。

每个Tensor Core可以提供4x4x4的矩阵处理阵列，执行运算 **D** = **A \* B + C** , 其中**A,** **B**, **C**, **D** 是4x4矩阵。如图所示。矩阵乘法的输入**A,** **B** 是FP16矩阵，加法矩阵 **C** 和**D** 可以是FP16或FP32矩阵。

每个Tensor Cores每个时钟周期可以执行64个浮点FMA(fused multiply–add)混合运算（FP16输入叉乘，和FP32加法），一个SM中的8个Tensor Cores, 每个时钟周期可以执行1024次浮点运算。

代码执行过程中，多个Tensor Cores可以封装并发执行。一个封装里面的线程可以执行16x16x16矩阵运算。CUDA在CUDA C++ WMMA API中暴露了封装层级的矩阵运算。这些接口提供了特定的矩阵加载，矩阵乘法和加法，矩阵存储，可以高效地利用Tensor Cores。

### CUDA库中的Tensor Cores

CUDA库中有2个库使用了Tensor Cores：cuBLAS和cuDNN。cuBLAS用Tensor Cores加速GEMM矩阵运算；cuDNN用Tensor Cores加速卷积核RNN。

矩阵运算有许多应用：信号处理，流体动力学等等。这些应用的数据都是指数级增长，需要加速运算速度。

### CUDA Tensor Cores编程

Tensor Cores 可以直接用CUDA C++进行编程。

Tensor Cores在CUDA中通过 nvcuda::wmma进行暴露。可以以Tensor Cores要求的格式 加载、设置值，执行矩阵乘法和加法， 将结果写回内存。在程序执行过程中，多个Tensor Cores在封装中并发执行。

以下例子使用WMMA(Warp Matrix Multiply Accumulate)实现矩阵乘法。并没有进行加速调试，可以使用CUDA里面的例子cudaTensorCoreGemm，获得具体代码。高性能编程需要使用cuBLAS。

头文件和命名空间

头文件mma.h 包含​WMMA API， 命名空间nvcuda .

声明和初始化

GEMM可以对矩阵进行转置，但为了简化，这里不作这种操作。先使用单一封装处理16x16的结果矩阵。用2维网格和线程块，可以将封装平铺到2维结果矩阵中。

MMA运算执行矩阵运算之前要放到GPU寄存器中表示。MMA是基于封装的操作，这些寄存器分布在一个封装中的线程之间，每个线程装载了矩阵的一个片段。这种矩阵到寄存器数据段的映射是透明的，写代码的时候不需要考虑。

初始化的最后一步是用0值填充加法器的数据段。

内层循环

GEMM每个封装计算结果矩阵的一个贴片。需要对A的行和B的列进行循环，沿着两个矩阵的K维轴生成MxN的贴片。矩阵加载函数从全局内存把数据加载到数据段。16x16的贴片在内存中是不连续的，函数需要知道连续列或行之间的间隔。MMA就会调用加法器进行计算。

计算完成

acc\_frag保存计算结果。GEMM可以对进行进行缩放。

最后，把数据写回内存。

### Tensor Core V100 加速方式

精度变化优化吞吐量

Tensor Core对以低精度FP16存储的张量可以在计算式使用更高的精度FP32，最大化吞吐量的同时还可以保持一定的精度。

张量数据格式

Tensor Cores操作张量是以通道交错的数据布局存储在内存中（Number-Height-Width-Channel, 通常简称 NHWC）以获得最好的性能。训练框架中通常是通道为主的数据布局方式（Number-Channel-Width-Height, 通常简称 NCHW）。cuDNN需要进行张量转置，如图所示。GPU卷积计算非常快，转置消耗了较多时间。

为了消除转置，直接用NHWC格式来表示网络模型中的张量，比如MXNet直接支持这种方式，MXNet和cuDNN对所有非卷积层也添加了这种数据布局方式，消除了所有训练中的张量转置。

很多非卷积层性能受限于DRAM的数据移动时间。用片上缓存融合相邻网络层可以避免DRAM通道拥堵，比如下图。MXNet创建图优化Pass，检测相邻的ADD和ReLU网络层， 用融合计算来取代它们。比较直接的方法就是用NNVM神经网络虚拟机（Neural Network Virtual Machine）实现这些优化。

还可以通过创建定制内核来优化同一分支下的多个类似卷积计算。

步骤 1 计算两个不同的向量之间的分值，分值是确定在对当前位置单词进行编码时我们对其他单词的关注程度。

步骤 2 使分值标准化，以具有更稳定的梯度，以便更好地训练，

步骤 3 将分值转换为概率。

最后，每个值向量乘以总和概率，概率较大的向量将获得后续网络层的更多关注。

计算机视觉任务中使用的大多数transformer都使用原始的transformer编码器模块。简而言之，它可以被视为一个新的特征选择器，不同于卷积神经网络 （CNN） 和循环神经网络 （RNN）。与只注重局部特性的CNN相比，变压器能够捕捉到长距离特性，这意味着变压器可以轻易地获得全球信息。与必须按顺序计算隐藏状态的RNN相比，变压器效率很高，因为自注意层和完全连接层的输出可以并行计算，并且很容易加速。