混合精度神经网络训练：理论与实践

傅德禹, 2018/11/21

关于演讲人

NVIDIA深度学习框架工程师

我们与facebook合作开发pytorch， 专注于提升GPU性能与易用性

如果您想把混合精度应用在您的项目或研究中，请与我联系

https://www.linkedin.com/in/deyufu

本演讲合著者：Christian Sarofeen, Ben Barsdell, Michael Carilli, Michael Ruberry

APEX

混合精度训练自动化工具

	AMP	FP16_Optimizer
开发意图	最大化数值稳定，高性能	高数值稳定性， 最小化代码变动， 最大化性能
检测所有pytorch函数 **自动选择fp16/32操作执行**	是	否
Fp32 master weight	是 (所有参数均存储为fp32)	是 (参数拷贝在optimizer中实现)
Dynamic Loss Scaling **自动调节缩放比例**	是	是

演讲目标

3行代码快速上手混合精度训练

```
from apex.fp16_utils import FP16_Optimizer

N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in ).cuda().half()
y = torch.randn(N, D_out).cuda().half()
model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
optimizer = FP16_Optimizer(optimizer, dynamic_loss_scale=True)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    optimizer.backward(loss)
    optimizer.step()
```

演讲目标

4行代码实现更好的稳定性

```
from apex import amp
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in ).cuda()
y = torch.randn(N, D_out).cuda()
amp_handle = amp.init()

model = torch.nn.Linear(D_in, D_out).cuda()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    with amp_handle.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
```

问题背景

深度学习框架通常默认使用单精度浮点数（fp32, 32 bit)，然而，使用**混合精度（非半精度）**和 **Volta** 我们可以实现：

2-4倍的**速度提升**，一半的**内存占用**

无准确率损失

无模型结构改变

无超参数改变

成功案例：FAIRSEQ

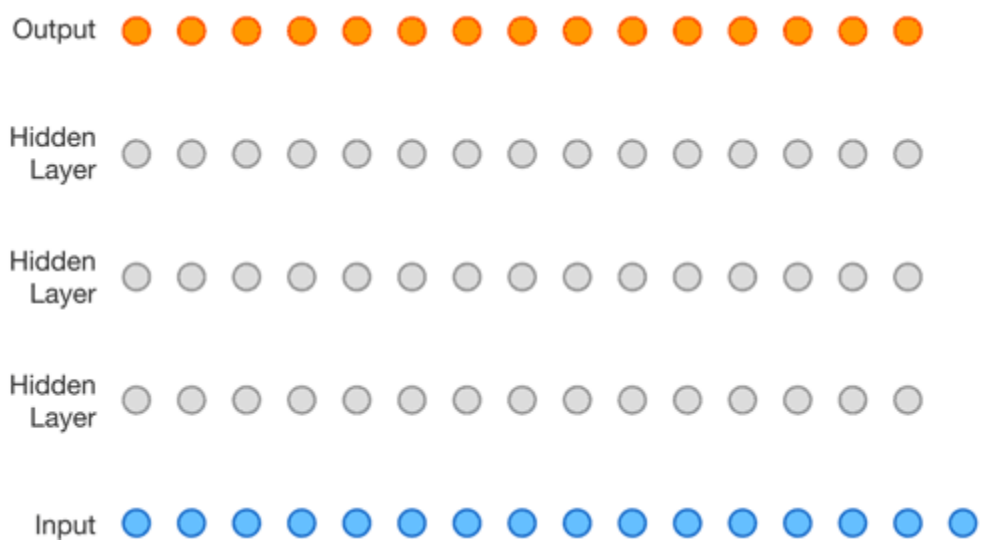
成功案例：图像识别

ILSVRC12 classification top-1 accuracy

Model	FP32	Mixed Precision
AlexNet	56.77%	56.93%
VGG-D	65.40%	65.43%
GoogLeNet (Inception v1)	68.33%	68.43%
Inception v2	70.03%	70.02%
Inception v3	73.85%	74.13%
Resnet50	75.92%	76.04%

成功案例： PROGRESSIVE GROWING OF GANS

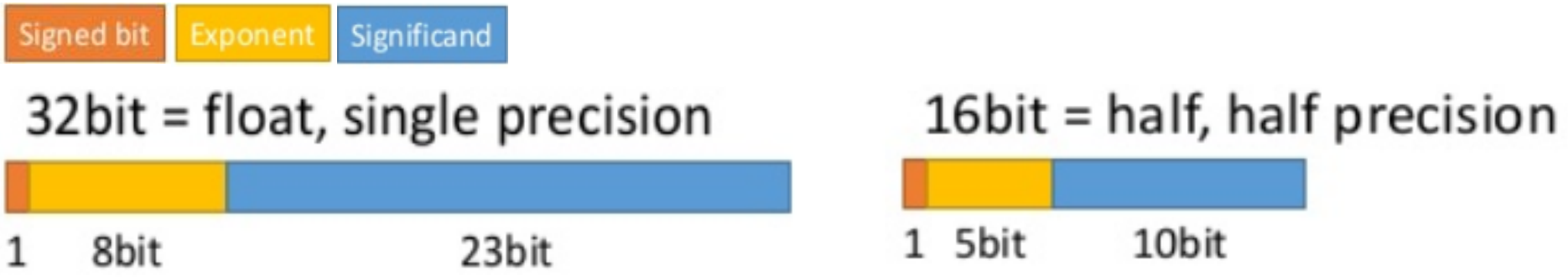
成功案例：WAVENET



目录

浮点数数值表示

数据类型：单精度和半精度



数据类型：单精度和半精度

可表示最大值
3.402823 x 10^38

非规格化最小值
2^-149

可准确表示小数点后6位

可表示最大值
65504

非规格化最小值
2^-24

不能准确表示所有范围内整数

目数据类型：单精度和半精度

- 纯粹的fp16更快，但是：
  - 有些操作需要更高的精度
  - 有些数值会overflow或underflow
  - 混合精度可以同时兼顾fp32的稳定性和fp16的速度

数据类型：未来

- 今天我们关注于fp32/16混合精度训练
- TensorRT现已支持INT4预测，Turing架构也支持INT4
- 已有研究尝试将INT8用于训练
- 混合精度将在未来变得更加强大，现在正是开始尝试的最佳时机

TENSOR CORES

精度对比

FP32	FP16 + Volta
1x 计算吞吐量	8x 计算吞吐量
1x 内存吞吐量	2x 内存吞吐量
1x 内存占用	1/2x 内存占用
32 bit 精度	16 bit 精度

TENSOR CORES

Tensor Cores是可以在fp16的输入值上进行快速矩阵乘法运算特制硬件单元

相比于传统fp16，Tensor Cores拥有更高精度的累加

相比于传统fp32，Tensor Cores可以提供8倍速度提升

主流深度学习框架均支持在输入值为fp16时自动使用Tensor Cores

用户也可按自己需求调用CUDA，CUDNN，CUBLAS API

TENSOR CORES

Throughput through matrix operation

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

TENSOR CORES

装备于Volta和Turing架构

V100 Tensor Cores提供了125Tflops的浮点数计算

在Volta架构，调用Tensor Cores的输入格式要求如下：

卷积：输入与输出信道数为8的倍数

矩阵乘法：M，N，K（所有输入输出尺寸）均为8的倍数

混合精度训练第一步：将模型转化为FP16

CONVERT TO FP16

```
N, D_in, D_out = 64, 1024, 512

x = torch.randn(N, D_in ).cuda()
y = torch.randn(N, D_out).cuda()

model = torch.nn.Linear(D_in, D_out).cuda()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

CONVERT TO FP16

```
N, D_in, D_out = 64, 1024, 512

x = torch.randn(N, D_in ).cuda().half()
y = torch.randn(N, D_out).cuda().half()

model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

CONVERT TO FP16

```
N, D_in, D_out = 64, 1024, 512

x = torch.randn(N, D_in ).cuda().half()
y = torch.randn(N, D_out).cuda().half()

model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

潜在问题

- 将整个网络转换成fp16，有些数值可能最终不在FP16范围内
- 满足|f(x)| >> x 的操作应在FP32下进行
  - exp, square, log, and cross-entropy loss
- 其他操作在FP16下是数值稳定的
  - ReLU, Sigmoid, Tanh

规约

- 大型的规约应在FP32下计算
- 累加很多FP16数值容易造成数据溢出

规约造成溢出

```
a = torch.cuda.HalfTensor(4094).fill_(16.0)
a.sum()
```

→ 65,504

```
b = torch.cuda.HalfTensor(4095).fill_(16.0)
b.sum()
```

→ inf

```
a = torch.cuda.HalfTensor(4095).fill_(16.0)
a.sum(dtype=torch.float32)
```

→ 65,520

Reductions like normalization overflow if > 65,504 is encountered.

规约造成溢出

```
N, D_in, D_out = 64, 1024, 512

x = torch.randn(N, D_in ).cuda().half()
y = torch.randn(N, D_out).cuda().half()

model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred.float(), y.float())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

混合精度训练第二步：安全进行参数更新

大数加小数造成误差

1 + 0.0001 = ?

FP32:

```
param = torch.cuda.FloatTensor([1.0])
print(param + 0.0001)
```

→ 1.0001

FP16:

```
param = torch.cuda.HalfTensor([1.0])
print(param + 0.0001)
```

→ 1

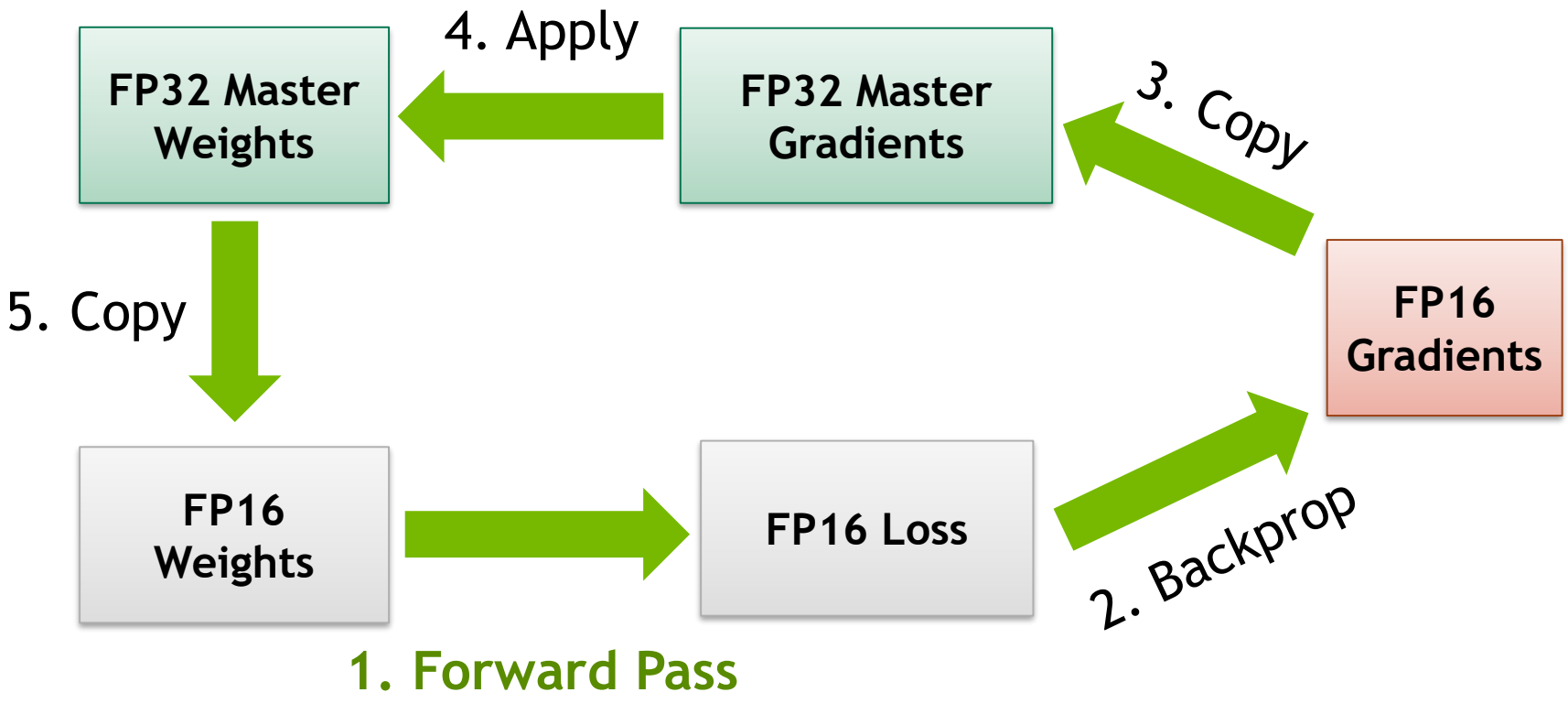
When *update*/*param* < 2^{-11}, updates have no effect.

FP16问题1：不准确的权值更新

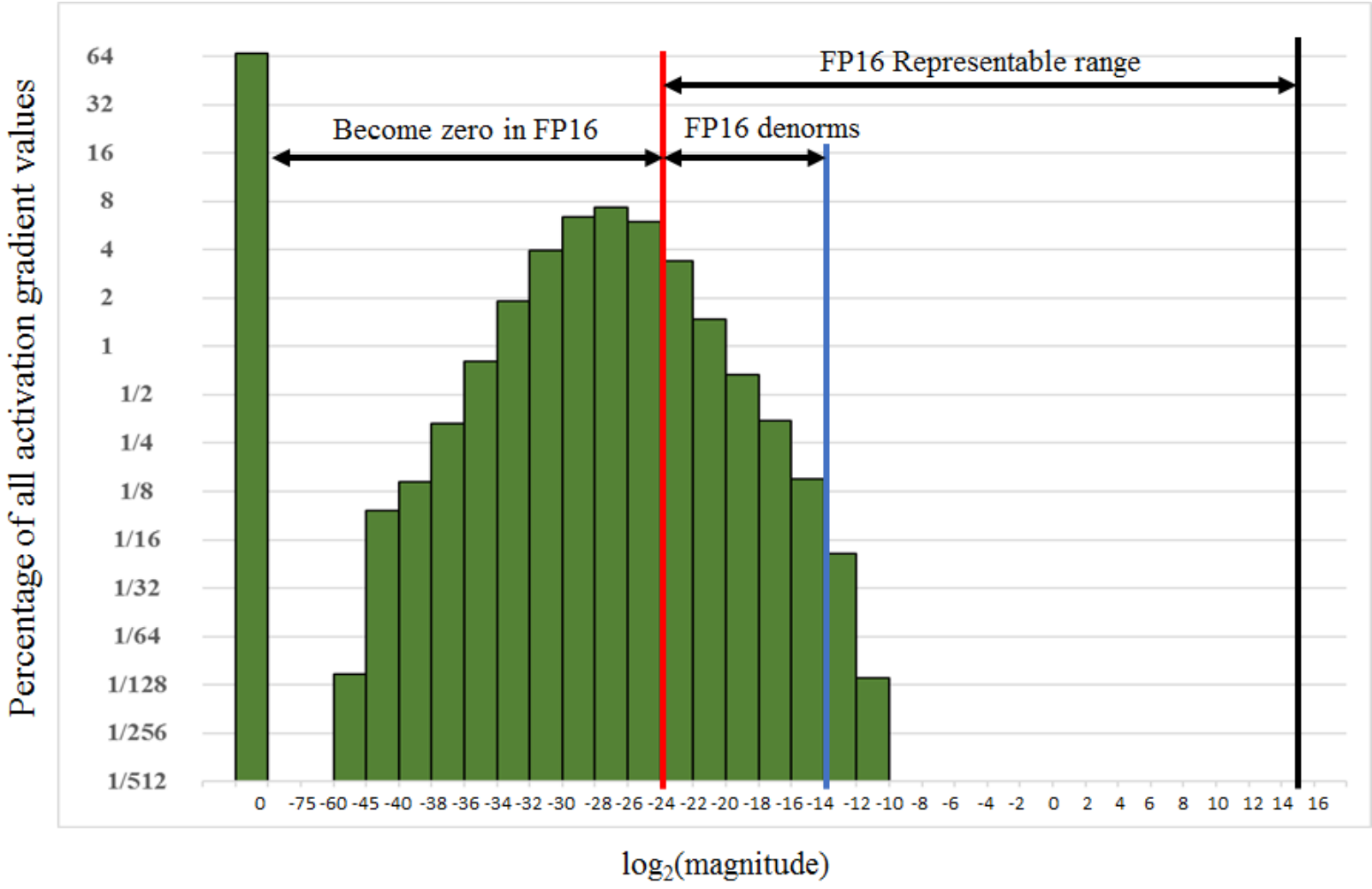
```
N, D_in, D_out = 64, 1024, 512

x = Variable(torch.randn(N, D_in )).cuda().half()
y = Variable(torch.randn(N, D_out)).cuda().half()

model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

绝对值小的FP16权值更新会因为误差而丢失

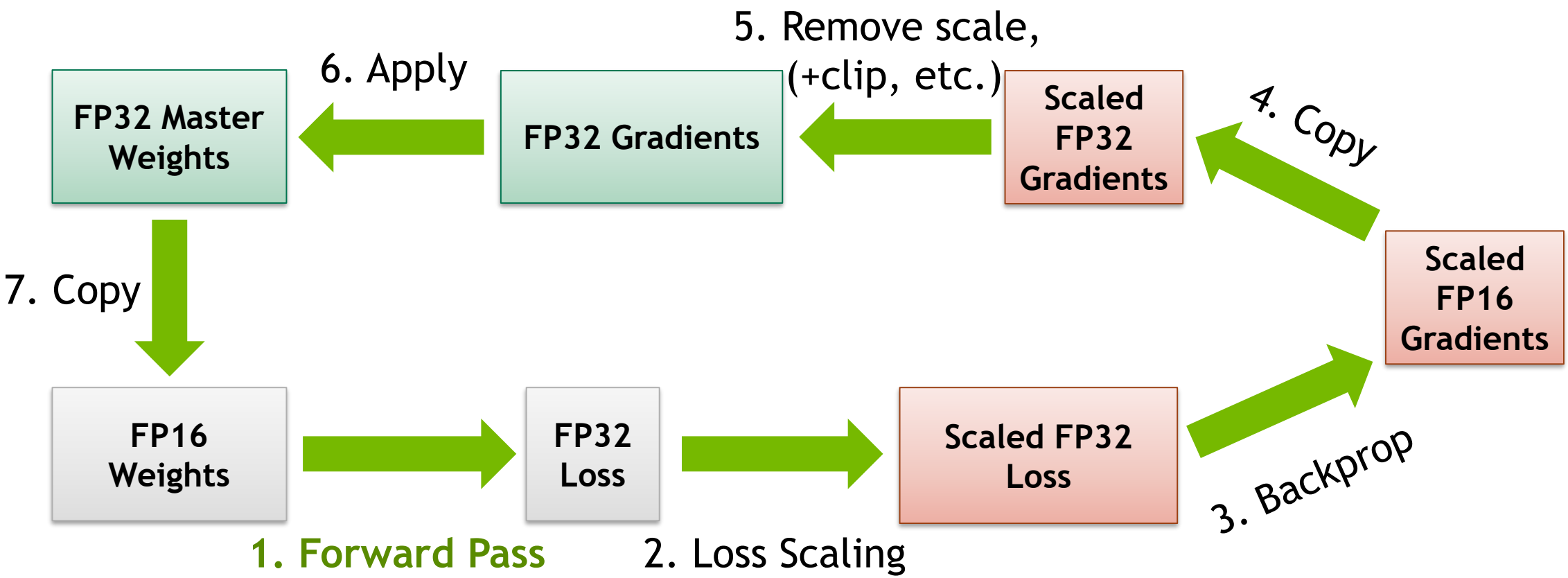
混合精度解决方案: FP32 MASTER WEIGHTS



FP16问题2：导数下溢（UNDERFLOW）



混合精度解决方案：LOSS SCALING



混合精度训练第三步：自动化工具实例

APEX

两种方式自动化混合精度

	AMP	FP16_Optimizer
开发意图	最大化数值稳定，高性能	高数值稳定性， 最小化代码变动， 最大化性能
检测所有pytorch函数 **自动选择fp16/32操作执行**	是	否
Fp32 master weight	是 (所有参数均存储为fp32)	是 (参数拷贝在optimizer中实现)
Dynamic Loss Scaling **自动调节缩放比例**	是	是

FP16 Optimizer

```
from apex.fp16_utils import FP16_Optimizer

N, D_in, D_out = 64, 1024, 512
x = Variable(torch.randn(N, D_in )).cuda().half()
y = Variable(torch.randn(N, D_out)).cuda().half()
model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
optimizer = FP16_Optimizer(optimizer, dynamic_loss_scale=True)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()

    optimizer.backward(loss)

    optimizer.step()
```

FP16 Optimizer

```
from apex.fp16_utils import FP16_Optimizer

N, D_in, D_out = 64, 1024, 512
x = Variable(torch.randn(N, D_in )).cuda().half()
y = Variable(torch.randn(N, D_out)).cuda().half()
model = torch.nn.Linear(D_in, D_out).cuda().half()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
optimizer = FP16_Optimizer(optimizer, dynamic_loss_scale=True)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()

    optimizer.backward(loss)


    optimizer.step()
```

1. 初始化master weight
2. 令optimizer指向含有master weight的fp16_optimizer

1. scaled_loss = loss*loss_scale
2. 在内部执行scaled_loss.backward(), 计算反向传播并生成scaled gradients存入.grad
3. 检查gradients是否存在溢出
4. 若无溢出, 将计算出的fp16导数转换成fp32并做downscale

1. 如果检测到导数值溢出，跳过此次更新并自动调整缩放比例
2. 若无溢出, 用fp32导数更新fp32权值--执行内部的fp32_optimizer.step()
3. 将更新后的fp32权值拷贝并转换成fp16，存入model.parameters()用于下一步训练

AMP

```
from apex import amp

N, D_in, D_out = 64, 1024, 512
x = Variable(torch.randn(N, D_in )).cuda()
y = Variable(torch.randn(N, D_out)).cuda()

amp_handle = amp.init()

model = torch.nn.Linear(D_in, D_out).cuda()

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)

    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()

    with amp_handle.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()

    optimizer.step()
```

AMP (全自动混合精度工具)

将所有imported `torch.*`函数根据黑/白名单打上补丁

```
import torch

from apex import amp

...

amp_handle = amp.init()

...
```

Conceptual operation:

```
for func in whitelist:
    def wrapped_func(input):
        return torch.func(input.half())
    patch torch.func to wrapped_func

for func in blacklist:
    def wrapped_func(input)
        return torch.func(input.float())
    patch torch.func to wrapped_func
```

已开源: https://github.com/NVIDIA/apex.git

AMP LISTS (TORCH OVERRIDES)

FP32 Functions

Pointwise (range)
- exp, log, log10, pow, sqrt... and many more

Reduction (error accumulation)
- cumprod, cumsum, mean, norm, std, sum, var

FP16 Functions

Tensor Core Functions
- conv1d, conv2d, conv3d
- conv_transpose1d, conv_transpose2d, conv_transpose2d
- addmm, addmv, addr, matmul, mm, mv

https://github.com/NVIDIA/apex/tree/master/apex/amp/lists

结论

APEX

Apex是PyTorch的开源扩展，主要由NVIDIA开发

更多功能，详细文档和实例： https://github.com/NVIDIA/Apex ，我们欢迎反馈与贡献

结论

- 混合精度训练可以在不损失精度，不改变超参数和不改变模型结构的前提下提升训练速度
- 速度提升来源于低精度下更快的数据移动，以及Volta和Turing架构加入的特制硬件Tensor Cores
- PyTorch在Apex的帮助下可以实现安全且自动化的混合精度训练

参考
(进一步阅读)

- Paulius Micikevicius's talk "Training Neural Networks with Mixed Precision: Theory and Practice" (GTC 2018, S8923).
- "Mixed Precision Training" (ICLR 2018).
- "Mixed- Precision Training of Deep Neural Networks" (NVIDIA Parallel Forall Blog).
- "Training with Mixed Precision" (NVIDIA User Guide).


NVIDIA