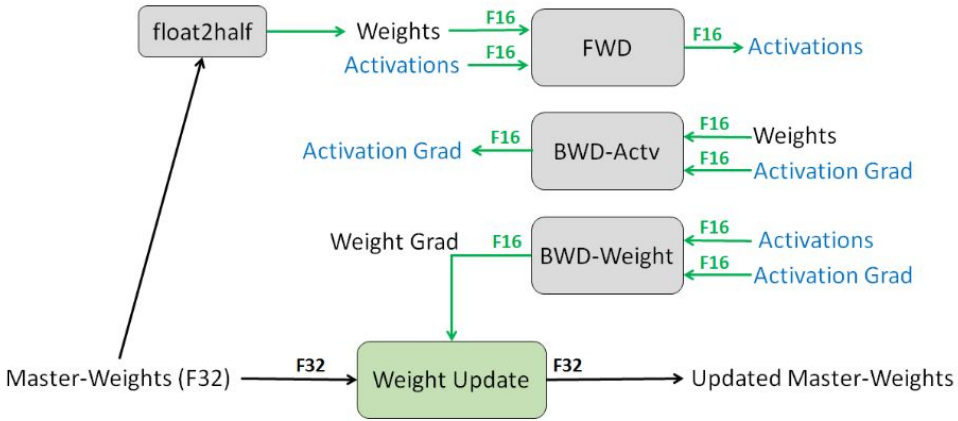
Mixed Precision Training

Explain Mixed Precision Training

- Paper: Mixed Precision Training (https://arxiv.org/pdf/1710.03740)
- Originally 32-bit floating point numbers used for pre-training
- Replaced by 16-bit floating point in favor of performance (Tensor Cores)
- Loss in final model performance
- Idea:
  - Use 32-bit floating points where required, 16-bit everywhere else
- Gradients require scaling, especially small ones

Explain Mixed Precision Training

- F32 master copy of weights
- F16 copy of the master weights used in forward- and backward propagation
- F32 master copy is updated in the optimizer step

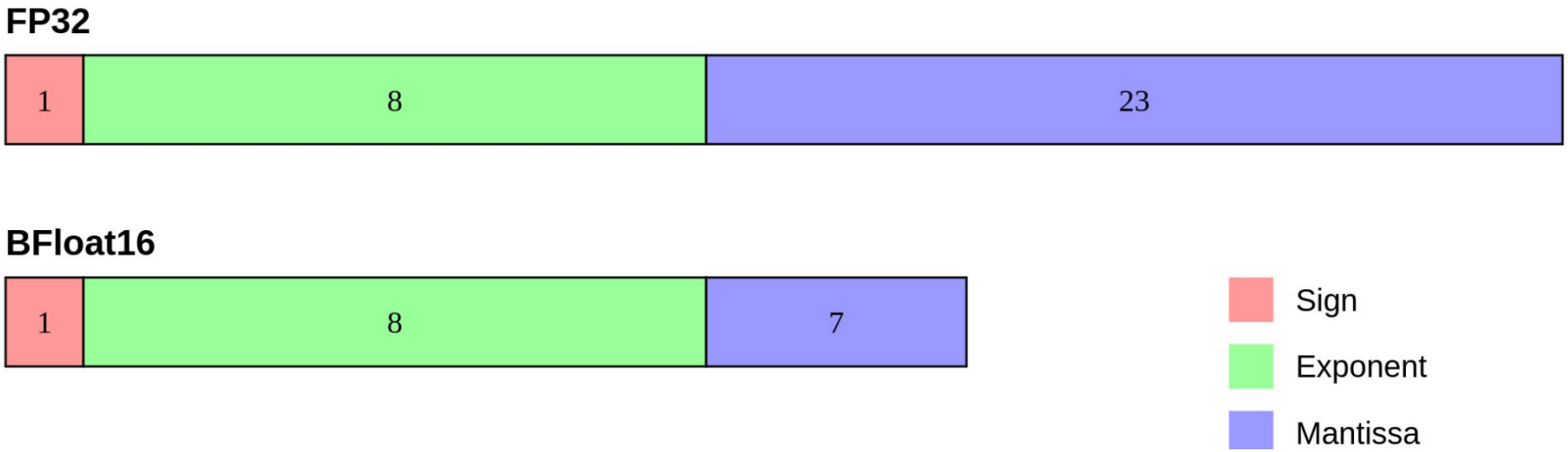


https://arxiv.org/pdf/1710.03740

bfloat16

- 16 bit floating point
- Developed by Google for machine learning applications
- Alternative to traditional IEEE 754 half-precision (FP32)

bfloat16 structure



Bfloat16

- Same dynamic range as 32-bit float (+- 3.4*10^38)
- Less Precision
- Less memory needed
- Simpler conversion to 32-bit float
- Maintains numeric stability in backpropagation

Usage in Pytorch

- Automatic Mixed Precision package
  - autocast
  - GradScaler
- Support for float16 and bfloat16
- Not supported for all operations
  - Linear layer/convolutions
  - Reductions -> dynamic range
- Best practices: What Every User Should Know About Mixed Precision Training in PyTorch

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm^2
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS | 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS | 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS | 624 teraFLOPS*

Problems

1. Gradient underflows
    a. Dynamic Scaling factor/Scaler to avoid overflow/underflows
        i. Scales loss before backward-pass -> gradient values have larger magnitude
    b. Helps convergence in float16
2. Unstable operations
3. TF32 vs. bfloat16 vs. float16
    a. Some networks perform better or even convergence only in some datatypes
    b. Sometimes more precision -> f16
    c. Sometimes more dynamic range -> bf16 (e.g. overflows)

Example

```
import torch

device = 'cuda'
model = MyModel().to(device)

scaler = torch.amp.GradScaler(enabled=true)

for epoch in range(epochs):
  for data, labels in train_loader:

    data, labels = data.to(device), labels.to(device)
    optimizer.zero_grad()

    with torch.amp.autocast(enabled=mixed_precision,device_type='cuda'):
        outputs = model(data)
        loss = criterion(outputs, labels)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

        ...
```