Training Neural Networks with Mixed Precision

MICHAEL CARILLI
CHRISTIAN SAROFEEEN
MICHAEL RUBERRY
BEN BARSDELL
THIS TALK

Using **mixed precision** and **Volta** your networks can be:

1. 2-4x **faster**
2. half the **memory use**
3. just as **powerful**

with **no architecture change**.
TALK OVERVIEW

1. Introduction to Mixed Precision Training
2. Mixed Precision Example in PyTorch
3. Appendix: Mixed Precision Example in TensorFlow
REFERENCES
(Further Reading)

• 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).
SINGLE VS HALF PRECISION

**FP32**
- 1x compute throughput
- 1x memory throughput
- 1x size
- 32 bit precision

**FP16 + Volta**
- 8X compute throughput
- 2X memory throughput
- 1/2X size
- 16 bit precision
MIXED PRECISION APPROACH

Imprecise weight updates → "Master" weights in FP32

Gradients may underflow → Loss (Gradient) Scaling

Maintain precision for reductions (sums, etc) → Accumulate in FP32
SUCCESS STORIES: SPEED

Pytorch
NVIDIA Sentiment Analysis: 4.5X speedup*
FAIRSeq: 4X speedup
GNMT: 2X speedup

TensorFlow
Resnet152: 2.2X speedup

*single Volta, Mixed Precision vs pure FP32
SUCCESS STORIES: ACCURACY

ILSVRC12 classification top-1 accuracy*

<table>
<thead>
<tr>
<th>Model</th>
<th>FP32</th>
<th>Mixed Precision**</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>56.77%</td>
<td>56.93%</td>
</tr>
<tr>
<td>VGG-D</td>
<td>65.40%</td>
<td>65.43%</td>
</tr>
<tr>
<td>GoogLeNet (Inception v1)</td>
<td>68.33%</td>
<td>68.43%</td>
</tr>
<tr>
<td>Inception v2</td>
<td>70.03%</td>
<td>70.02%</td>
</tr>
<tr>
<td>Inception v3</td>
<td>73.85%</td>
<td>74.13%</td>
</tr>
<tr>
<td>Resnet50</td>
<td>75.92%</td>
<td>76.04%</td>
</tr>
</tbody>
</table>

*Sharan Narang, Paulius Micikevicius et al., "Mixed Precision Training" (ICLR 2018)
**Same hyperparameters and learning rate schedule as FP32.

Mixed precision can match FP32 with no change in hyperparameters.
SUCCESS STORIES: ACCURACY

Progressive Growing of GANs: Generates 1024x1024 face images

http://research.nvidia.com/publication/2017-10_Progressive-Growing-of

No perceptible difference between FP32 and mixed-precision training

Loss-scaling:

Separate scaling factors for generator and discriminator (you are training 2 networks)

Automatic loss scaling greatly simplified training - gradient stats shift drastically when image resolution is increased
THIS TALK (REPRISE)

Using **mixed precision** and **Volta** your networks can be:

1. 2-4x **faster**
2. half the **memory use**
3. just as **powerful**

with **no architecture change**.
**TENSOR CORE PERFORMANCE TIPS**

- **Convolutions:**
  Batch size, input channels, output channels should be multiples of 8.

- **GEMM:**
  For $A \times B$ where $A$ has size $(N, M)$ and $B$ has size $(M, K)$, $N$, $M$, $K$ should be multiples of 8.

- **Fully connected layers are GEMMs:**
  Batch size, input features, output features should be multiples of 8.

Libraries (cuDNN, cuBLAS) are optimized for Tensor Cores.
TENSOR CORE PERFORMANCE TIPS

How can I make sure Tensor Cores were used? Run one iteration with nvprof, and look for “884” kernels:

```python
import torch
import torch.nn

bsz, in, out = 256, 1024, 2048

tensor = torch.randn(bsz, in).cuda().half()
layer = torch.nn.Linear(in, out).cuda().half()
layer(tensor)
```

Running with

$ nvprof python test.py

...37.024us 1 37.024us 37.024us 37.024us volta_fp16_s884gemm_fp16...
TENSOR CORE PERFORMANCE TIPS

If your data/layer sizes are constant each iteration, try

```python
import torch
torch.backends.cudnn.benchmark = True
...
```

This enables cuDNN’s autotuner. The first iteration, it will try many algorithms, and choose the fastest to use in later iterations.

PYTORCH EXAMPLE
A SIMPLE NETWORK

N, D_in, D_out = 64, 1024, 512

x = Variable(torch.randn(N, D_in)).cuda()
y = Variable(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 = 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()
CONVERT TO FP16

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()
**MIXED PRECISION APPROACH**

- Imprecise weight updates ➔ "Master" weights in FP32
- Gradients may underflow ➔ Loss (Gradient) Scaling
- Maintain precision for reductions (sums, etc) ➔ Accumulate in FP32
**IMPRECISE WEIGHT UPDATES**

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.
IMPRECISE WEIGHT UPDATES

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()     Small FP16 weight updates may be lost
MIXED SOLUTION: FP32 MASTER WEIGHTS

1. Forward Pass

FP32 Master Weights -> FP32 Master Gradients

2. Backprop

FP32 Master Gradients -> FP16 Gradients

3. Copy

FP16 Gradients -> FP16 Weights

4. Apply

FP16 Weights -> FP16 Loss

5. Copy

FP16 Loss -> FP32 Master Weights
def prep_param_lists(model):
    model_params = [p for p in model.parameters() if p.requires_grad]
    master_params = [p.clone().detach().float() for p in model_params]

    for p in master_params:
        p.requires_grad = True

    return model_params, master_params

Note: Model↔master param and gradient copies act on .data members. They are not recorded by Pytorch’s autograd system.
MIXED SOLUTION: FP32 MASTER WEIGHTS

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()
model_params, master_params = prep_param_lists(model)

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

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

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

    model.zero_grad()
lloss.backward()
    optimizer.step()
def master_params_to_model_params(model_params, master_params):
    for model, master in zip(model_params, master_params):
        model.data.copy_(master.data)
MIXED SOLUTION: FP32 MASTER WEIGHTS

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()
model_params, master_params = prep_param_lists(model)

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

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

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

master_params_to_model_params(model_params, master_params)
def model_grads_to_master_grads(model_params, master_params):
    for model, master in zip(model_params, master_params):
        if master.grad is None:
            master.grad = Variable(
                master.data.new(*master.data.size()))
        master.grad.data.copy_(model.grad.data)
MIXED SOLUTION: FP32 MASTER WEIGHTS

\( N, D_{\text{in}}, D_{\text{out}} = 64, 1024, 512 \)

\[
x = \text{Variable}(\text{torch.randn}(N, D_{\text{in}})).cuda().half()
\]

\[
y = \text{Variable}(\text{torch.randn}(N, D_{\text{out}})).cuda().half()
\]

\[
\text{model} = \text{torch.nn.Linear}(D_{\text{in}}, D_{\text{out}}).cuda().half()
\]

\[
\text{model\_params}, \text{master\_params} = \text{prep\_param\_lists}(\text{model})
\]

\[
\text{optimizer} = \text{torch.optim.SGD}(\text{master\_params, lr=1e-3})
\]

\[
\text{for } t \text{ in range}(500):
\]

\[
\text{y\_pred} = \text{model}(x)
\]

\[
\text{loss} = \text{torch.nn.functional.mse\_loss}(\text{y\_pred}, y)
\]

\[
\text{model\_zero\_grad()}
\]

\[
\text{loss\_backward()}
\]

\[
\text{model\_grads\_to\_master\_grads(\text{model\_params, master\_params})}
\]

\[
\text{optimizer\_step()}
\]

\[
\text{master\_params\_to\_model\_params(\text{model\_params, master\_params})}
\]
MIXED SOLUTION: FP32 MASTER WEIGHTS

1. Forward Pass

2. Backprop

3. Copy

4. Apply

5. Copy

FP32 Master Weights

FP32 Master Gradients

FP16 Weights

FP16 Loss

FP16 Gradients
MIXED PRECISION APPROACH

- Imprecise weight updates
- Gradients may underflow
- Maintain precision for reductions (sums, etc)
- "Master" weights in FP32
- Loss (Gradient) Scaling
- Accumulate in FP32
GRADIENTS MAY UNDERFLOW

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()
model_params, master_params = prep_param_lists(model)

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

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    model.zero_grad()
    loss.backward()
    model_grads_to_master_grads(model_params, master_params)
    optimizer.step()
    master_params_to_model_params(model_params, master_params)
Range representable in FP16: ~40 powers of 2
Range representable in FP16: \(~40\) powers of 2

Gradients are small, don’t use much of FP16 range

FP16 range not used by gradients: \(~15\) powers of 2
Range representable in FP16: \( \sim 40 \) powers of 2

Gradients are small, don’t use much of FP16 range

FP16 range not used by gradients: \( \sim 15 \) powers of 2

**Loss Scaling**

1. Multiply the loss by some constant \( S \).
2. Call `backward()` on scaled loss. By chain rule, gradients will also be scaled by \( S \). This preserves small gradient values.
3. Unscale gradients before update `step()`.
MIXED SOLUTION: LOSS (GRADIENT) SCALING

\[ \text{N, D}_{\text{in}}, \text{D}_{\text{out}} = 64, 1024, 512 \]
\[ \text{scale\_factor} = 128.0 \]

......

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

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

    scaled_loss = scale_factor * loss.float()

model.zero_grad()
scaled_loss.backward()
model_grads_to_master_grads(model_params, master_params)

for param in master_params:
    param.grad.data.mul_(1./scale_factor)

optimizer.step()
master_params_to_model_params(model_params, master_params)

Gradients are now rescaled to be representable

The FP32 master gradients must be "descaled"
MASTER WEIGHTS

1. Forward Pass
2. Backprop
3. Copy
4. Apply
5. Copy

FP32 Master Weights

FP32 Master Gradients

FP16 Gradients

FP16 Loss

FP16 Weights
MASTER WEIGHTS + LOSS SCALING

1. Forward Pass
2. Loss Scaling
3. Backprop
4. Copy
5. Remove scale, (+clip, etc.)
6. Apply
7. Copy

FP32 Master Weights
FP32 Gradients
Scaled FP32 Gradients
Scaled FP16 Gradients
Scaled FP32 Loss
FP32 Loss
FP16 Weights
LOSS SCALING FAQ

1. Does loss scaling require retuning the learning rate?

   **No.** Loss scaling is orthogonal to learning rate. Changing loss scale should not require retuning other hyperparameters.

2. Can the loss scale be adjusted each iteration?

   **Yes.** For example, use larger loss scale later in training, when gradients are smaller.

   Dynamic loss scaling adjusts loss scale automatically (more on that later)
MIXED PRECISION APPROACH

Imprecise weight updates → "Master" weights in FP32
Gradients may underflow → Loss (Gradient) Scaling
Maintain precision for reductions (sums, etc) → Accumulate in FP32
REDUCTIONS MAY OVERFLOW

In PyTorch 0.5:

```python
a = torch.cuda.HalfTensor(4094).fill_(16.0)
a.sum()  # 65,504
```

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

Reductions like `sum()` can overflow if > 65,504 is encountered.

Behavior may depend on PyTorch version.
N, D_in, D_out = 64, 1024, 512
scale_factor = 128.0

......

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

    loss = torch.nn.functional.mse_loss(y_pred, y)  # mse_loss is a reduction
    scaled_loss = scale_factor * loss.float()

    model.zero_grad()
    scaled_loss.backward()
    model_grads_to_master_grads(model_params, master_params)

    for param in master.params:
        param.grad.data.mul_(1./scale_factor)
    optimizer.step()
    master_params_to_model_params(model_params, master_params)
MIXED SOLUTION: ACCUMULATE IN FP32

N, D_in, D_out = 64, 1024, 512
scale_factor = 128.0

......

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred.float(), y.float())
    scaled_loss = scale_factor * loss

    model.zero_grad()
    scaled_loss.backward()
    model_grads_to_master_grads(model_params, master_params)

    for param in master.params:
        param.grad.data.mul_(1./scale_factor)
    optimizer.step()
    master_params_to_model_params(model_params, master_params)
OTHER REDUCTIONS

BatchNorm involves a reduction.
BatchNorm may also need to be done in FP32:

```python
def BN_convert_float(module):
    if isinstance(module, torch.nn.modules.batchnorm._BatchNorm):
        module.float()
    for child in module.children():
        BN_convert_float(child)
    return module
```
### Mixed Precision Approach

- Imprecise weight updates
- Gradients may underflow
- Maintain precision for reductions (sums, etc.)

"Master" weights in FP32
Loss (Gradient) Scaling
Accumulate in FP32
ADDITIONAL CONSIDERATIONS

Checkpointing

1. Save master weights.
2. Save gradient scale factor.
ADDITIONAL CONSIDERATIONS

Dynamic loss scaling

Prevent gradient UNDERflow by using the highest loss scale that does not cause gradient OVERflow.

1. Start with a large loss scale (e.g. $2^{32}$)
2. After each iteration, check if gradients overflowed (NaN or +/- Inf).
3. If gradients overflowed, discard that iteration by skipping optimizer.step().
4. If gradients overflowed, reduce S for the next iteration (e.g. $S = S/2$)
5. If N (e.g. 1000) iterations pass with no overflow, increase S again ($S = S*2$).

More detail:
https://nvidia.github.io/apex/fp16_utils.html#apex.fp16_utils.DynamicLossScaler
Example:
https://github.com/NVIDIA/apex/blob/ea93767d22818bdd88ae738a8c7cf62b49a8fdafeapex/fp16_utils/loss_scaler.py#L134-L186
NVIDIA MIXED PRECISION TOOLS

APEX - A PyTorch Extension

• All utility functions in this talk (model_grads_to_master_grads, etc.)

• **FP16_Optimizer**: Optimizer wrapper that automatically manages loss scaling + master params
  
  • Closure-safe
  
  • Option to automatically manage dynamic loss scaling
  
  • Compatible with PyTorch distributed training

[www.github.com/nvidia/apex](http://www.github.com/nvidia/apex)

Documentation: [https://nvidia.github.io/apex/fp16_utils.html](https://nvidia.github.io/apex/fp16_utils.html)
TENSORFLOW EXAMPLE
TENSORFLOW MIXED PRECISION SUPPORT

TensorFlow supports mixed precision using tf.float32 and tf.float16 data types.

Reduce memory and bandwidth by using float16 tensors.

Improve compute performance by using float16 matmuls and convolutions.

Maintain training accuracy by using mixed precision.
1. Convert model to float16 data type

2. Use float32 for certain ops to avoid overflow or underflow
   - Reductions (e.g., norm, softmax)
   - Range-expanding math functions (e.g., exp, pow)

3. Use float32 for weights storage to avoid imprecise training updates

4. Apply loss scaling to avoid gradients underflow
import tensorflow as tf
import numpy as np

def build_forward_model(inputs):
    _, _, h, w = inputs.get_shape().as_list()
    top_layer = inputs
    top_layer = tf.layers.conv2d(top_layer, 64, 7, use_bias=False,
                                 data_format='channels_first', padding='SAME')
    top_layer = tf.contrib.layers.batch_norm(top_layer, data_format='NCHW', fused=True)
    top_layer = tf.layers.max_pooling2d(top_layer, 2, 2, data_format='channels_first')
    top_layer = tf.reshape(top_layer, (-1, 64 * (h // 2) * (w // 2)))
    top_layer = tf.layers.dense(top_layer, 128, activation=tf.nn.relu)
    return top_layer
import tensorflow as tf
import numpy as np

def build_forward_model(inputs):
    _, _, h, w = inputs.get_shape().as_list()
    top_layer = inputs
    top_layer = tf.layers.conv2d(top_layer, 64, 7, use_bias=False,
                                 data_format='channels_first', padding='SAME')
    top_layer = tf.contrib.layers.batch_norm(top_layer, data_format='NCHW', fused=True)
    top_layer = tf.layers.max_pooling2d(top_layer, 2, 2, data_format='channels_first')
    top_layer = tf.reshape(top_layer, (-1, 64 * (h // 2) * (w // 2)))
    top_layer = tf.layers.dense(top_layer, 128, activation=tf.nn.relu)
    return top_layer

Convolutions support Tensor Cores
Batchnorm supports mixed precision
Matrix multiplications support Tensor Cores
Majority of TF ops support float16 data type
def build_training_model(inputs, labels, nlabel):
    top_layer = build_forward_model(inputs)
    logits = tf.layers.dense(top_layer, nlabel, activation=None)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    gradvars = optimizer.compute_gradients(loss)
    train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op
EXAMPLE (PART 3)
Training boilerplate

nchan, height, width, nlabel = 3, 224, 224, 1000
inputs = tf.placeholder(tf.float32, (None, nchan, height, width))
labels = tf.placeholder(tf.int32, (None,))
inputs, labels, loss, train_op = build_training_model(inputs, labels, nlabel)
batch_size = 128
sess = tf.Session()
inputs_np = np.random.random(size=(batch_size, nchan, height, width)).astype(np.float32)
labels_np = np.random.randint(nlabel, size=(batch_size,)).astype(np.int32)
sess.run(tf.global_variables_initializer())
for step in range(20):
    loss_np, _ = sess.run([loss, train_op],
                          {inputs: inputs_np,
                           labels: labels_np})

    print("Loss =", loss_np)
ORIGINAL GRAPH

Input -> Forward model -> Loss

Loss -> Backward model -> Weight gradients

Update: Weights <- Weight gradients

float32

float16
def build_training_model(inputs, labels, nlabel):
    top_layer = build_forward_model(inputs)
    logits = tf.layers.dense(top_layer, nlabel, activation=None)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    gradvars = optimizer.compute_gradients(loss)
    train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    top_layer = build_forward_model(inputs)
    logits = tf.layers.dense(top_layer, nlabel, activation=None)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    gradvars = optimizer.compute_gradients(loss)
    train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op
STEP 1: CONVERSION TO FP16

```python
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    top_layer = build_forward_model(inputs)
    logits = tf.layers.dense(top_layer, nlabel, activation=None)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    gradvars = optimizer.compute_gradients(loss)
    train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op
```

Gradients and variables are float16 too
STEP 1: CONVERSION TO FP16

- **Input**
  - Forward model
  - Loss
  - Backward model
  - Weights
  - Weight gradients
- **Update**

- float32
- float16
NEW GRAPH

- **float32**
- **float16**

Diagram:
- **Input**
  - **Cast to fp16**
  - **Forward model**
  - **Loss**
  - **Backward model**
  - **Weights**
  - **Weight gradients**
  - **Update**
  - **Cast to fp16**
  - **Weights**
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    top_layer = build_forward_model(inputs)
    logits = tf.layers.dense(top_layer, nlabel, activation=None)
    logits = tf.cast(logits, tf.float32)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    gradvars = optimizer.compute_gradients(loss)
    train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op
STEP 2: USE FP32 TO COMPUTE THE LOSS

- Forward model
- Loss
- Backward model
- Update
- Weight gradients

- Cast to fp32
- Input
- Weights
- Weight gradients

float32
float16
NEW GRAPH

float32
float16

Input
Cast to fp16
Forward model
Cast to fp32
Loss
Cast to fp16
Backward model
Weights
Update
Weight gradients
STEP 3: FP32 MASTER WEIGHTS

Helper function

def float32_variable_storage_getter(getter, name, shape=None, dtype=None, 
    initializer=None, regularizer=None, 
    trainable=True, 
    *args, **kwargs):

    """Custom variable getter that forces trainable variables to be stored in 
    float32 precision and then casts them to the training precision. 
    """

    storage_dtype = tf.float32 if trainable else dtype
    variable = getter(name, shape, dtype=storage_dtype, 
        initializer=initializer, regularizer=regularizer, 
        trainable=trainable, 
        *args, **kwargs)

    if trainable and dtype != tf.float32:
        variable = tf.cast(variable, dtype)

    return variable
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    with tf.variable_scope('fp32_vars', custom_getter=float32_variable_storage_getter):
        top_layer = build_forward_model(inputs)
        logits = tf.layers.dense(top_layer, nlabel, activation=None)
    logits = tf.cast(logits, tf.float32)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    gradvars = optimizer.compute_gradients(loss)
    train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    with tf.variable_scope('fp32_vars', custom_getter=float32_variable_storage_getter):
        top_layer = build_forward_model(inputs)
        logits = tf.layers.dense(top_layer, nlabel, activation=None)
        logits = tf.cast(logits, tf.float32)
        loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
        optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
        gradvars = optimizer.compute_gradients(loss)
        train_op = optimizer.apply_gradients(gradvars)
    return inputs, labels, loss, train_op

Gradients and variables are now float32, but the gradient computations still use float16.
STEP 3: FP32 MASTER WEIGHTS

**Input**
- **Forward model**
  - **Cast to fp32**
- **Weights**
  - **Cast to fp16**
  - **Backward model**
    - **Cast to fp16**
      - **Loss**
      - **Update**
        - **Weight gradients**

**Cast to fp16**
- **Input**
- **Weights**

**Colors**
- **Blue**: float32
- **Green**: float16
NEW GRAPH

- **float32**
- **float16**

Diagram:
- **Input** → **Cast to fp16** → **Forward model** → **Cast to fp16** → **Loss** → **Cast to fp16** → **Backward model** → **Cast to fp16** → **Weights** → **Update** → **Weight gradients** → **Cast to fp32**
STEP 4: LOSS (GRADIENT) SCALING

Avoid gradient underflow

```python
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    with tf.variable_scope('fp32_vars', custom_getter=float32_variable_storage_getter):
        top_layer = build_forward_model(inputs)
        logits = tf.layers.dense(top_layer, nlabel, activation=None)
    logits = tf.cast(logits, tf.float32)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    loss_scale = 128.0  # Value may need tuning depending on the model
    gradients, variables = zip(*optimizer.compute_gradients(loss * loss_scale))
    gradients = [grad / loss_scale for grad in gradients]
    train_op = optimizer.apply_gradients(zip(gradients, variables))
    return inputs, labels, loss, train_op
```
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    with tf.variable_scope('fp32_vars', custom_getter=float32_variable_storage_getter):
        top_layer = build_forward_model(inputs)
        logits = tf.layers.dense(top_layer, nlabel, activation=None)
        logits = tf.cast(logits, tf.float32)
        loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
        optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
        loss_scale = 128.0  # Value may need tuning depending on the model
        gradients, variables = zip(*optimizer.compute_gradients(loss * loss_scale))
        gradients = [grad / loss_scale for grad in gradients]
        train_op = optimizer.apply_gradients(zip(gradients, variables))
    return inputs, labels, loss, train_op

Returns gradients (now float32) to correct exponent

Raises exponent during bwd pass in float16

Avoid gradient underflow
STEP 4: LOSS (GRADIENT) SCALING

- **Input**
- **Forward model**
  - Cast to fp16
  - Cast to fp32
- **Loss**
  - * loss_scale
  - Cast to fp16
- **Backward model**
  - Cast to fp16
- **Weights**
  - Update: / loss_scale
  - Cast to fp32
- **Weight gradients**

- **float32**
- **float16**
NEW GRAPH

**float32**

**float16**

- **Input**
  - **Forward model**
    - **Cast to fp16**
  - **Weight gradients**
    - **Backward model**
      - **Cast to fp16**
  - **Loss**
    - **Cast to fp32**
      - $\times \text{loss\_scale}$
    - **Update**
      - $\text{Weights} - \text{loss\_scale}$
      - **Cast to fp32**
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    with tf.variable_scope('fp32_vars', custom_getter=float32_variable_storage_getter):
        top_layer = build_forward_model(inputs)
        logits = tf.layers.dense(top_layer, nlabel, activation=None)
    logits = tf.cast(logits, tf.float32)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    loss_scale = 128.0  # Value may need tuning depending on the model
    gradients, variables = zip(*optimizer.compute_gradients(loss * loss_scale))
    gradients = [grad / loss_scale for grad in gradients]
    gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
    train_op = optimizer.apply_gradients(zip(gradients, variables))
    return inputs, labels, loss, train_op
def build_training_model(inputs, labels, nlabel):
    inputs = tf.cast(inputs, tf.float16)
    with tf.variable_scope('fp32_vars', custom_getter=float32_variable_storage_getter):
        top_layer = build_forward_model(inputs)
        logits = tf.layers.dense(top_layer, nlabel, activation=None)
    logits = tf.cast(logits, tf.float32)
    loss = tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels)
    optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
    loss_scale = 128.0  # Value may need tuning depending on the model
    gradients, variables = zip(*optimizer.compute_gradients(loss * loss_scale))
    gradients = [grad / loss_scale for grad in gradients]
    gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
    train_op = optimizer.apply_gradients(zip(gradients, variables))
    return inputs, labels, loss, train_op
Mixed precision training is now well supported by deep learning frameworks

Conversion requires < 10 lines of code for most training scripts

For more info and frameworks, see our Mixed Precision Training guide:
