AUTOMATIC MIXED PRECISION IN PYTORCH

Michael Carilli and Michael Ruberry, 3/20/2019
THIS TALK

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

1. 2-4x **faster**

2. more **memory-efficient**

3. **just as powerful**

with **no architecture change**.
REFERENCES

Myle Ott and Sergey Edunov, *Taking Advantage of Mixed Precision to Accelerate Training Using PyTorch*, GTC 2019 Session 9832
Right after this talk in Room 210D

Carl Case, *Mixed Precision Training of Deep Neural Networks*, GTC 2019 Session 9143

Sharan Narang, Paulius Micikevicius *et al.*, *Mixed Precision Training*, ICLR 2018

Automatic Mixed Precision (AMP) for Pytorch is part of NVIDIA Apex:
https://github.com/nvidia/apex
https://nvidia.github.io/apex/
TALK OVERVIEW

1. Introduction to Mixed Precision Training
2. Automatic Mixed Precision (AMP) for PyTorch
3. Mixed Precision Principles in AMP
4. Tensor Core Performance Tips
INTRODUCTION TO MIXED PRECISION TRAINING
FP32 AND FP16

FP32
8-bit exponent, 23-bit mantissa

Dynamic range:
$1.4 \times 10^{-45} < x < 3.4 \times 10^{38}$

FP16
5-bit exponent, 10-bit mantissa

Dynamic range:
$5.96 \times 10^{-8} < x < 65504$
MAXIMIZING MODEL PERFORMANCE

FP16 is fast and memory-efficient.

<table>
<thead>
<tr>
<th>FP32</th>
<th>FP16 with Tensor Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x compute throughput</td>
<td>8X compute throughput</td>
</tr>
<tr>
<td>1x memory throughput</td>
<td>2X memory throughput</td>
</tr>
<tr>
<td>1x memory storage</td>
<td>1/2X memory storage</td>
</tr>
</tbody>
</table>
MAXIMIZING MODEL PERFORMANCE

FP16 input enables Volta/Turing Tensor Cores.

FP16 Input, FP32 Accumulate, FP16 Output for GEMMs and Convolutions

\[
D = \left( \begin{array}{cccc}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3}
\end{array} \right) + \left( \begin{array}{cccc}
B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\
B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\
B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\
B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3}
\end{array} \right)
\]

125 TFlops Throughput: 8X more than FP32 on Volta V100
MAXIMIZING MODEL PERFORMANCE

FP32 offers precision and range benefits.

- **FP32**
  - Wider dynamic range
  - Increased precision captures small accumulations

- **FP16**
  - Narrower dynamic range
  - Reduced precision may lose small accumulations
MAXIMIZING MODEL PERFORMANCE

Certain ops require FP32 dynamic range.

Reductions, exponentiation

```python
a = torch.cuda.HalfTensor(4096)
a.fill_(16.0)
a.sum()  # inf
```

```python
b = torch.cuda.FloatTensor(4096)
b.fill_(16.0)
b.sum()  # 65,536
```
MAXIMIZING MODEL PERFORMANCE

Addition of large + small values benefits from FP32 precision.

Weight updates, reductions again

\[
1 + 0.0001 = ??
\]

```python
param = torch.cuda.HalfTensor([1.0])
update = torch.cuda.HalfTensor([0.0001])
print(param + update)  # 1
```

```
In FP16, when \( update/param < 2^{-11} \approx 0.00049 \), update has no effect.
```

```python
param = torch.cuda.FloatTensor([1.0])
update = torch.cuda.FloatTensor([0.0001])
print(param + update)  # 1.0001
```
MAXIMIZING MODEL PERFORMANCE

Assign each operation its optimal precision.

**FP16**
- GEMMs + Convolutions can use Tensor Cores
- Most pointwise ops (e.g. add, multiply): 1/2X memory storage for intermediates, 2X memory throughput

**FP32**
- Weight updates benefit from precision
- Loss functions (often reductions) benefit from precision and range
- Softmax, norms, some other ops benefit from precision and range
MIXED PRECISION IN PRACTICE: SPEED

Single Volta, FP32 vs Mixed Precision

Nvidia Sentiment Analysis**: 4.5X speedup

** https://github.com/NVIDIA/sentiment-discovery
MIXED PRECISION IN PRACTICE: SPEED

Single Volta, FP32 vs Mixed Precision

Nvidia Sentiment Analysis**: 4.5X speedup

FAIRseq: 4X speedup

** https://github.com/NVIDIA/sentiment-discovery
MIXED PRECISION IN PRACTICE: SPEED

Single Volta, FP32 vs Mixed Precision

Nvidia Sentiment Analysis**: 4.5X speedup

FAIRseq: 4X speedup

GNMT: 2X speedup

** [https://github.com/NVIDIA/sentiment-discovery](https://github.com/NVIDIA/sentiment-discovery)
MIXED PRECISION IN PRACTICE: ACCURACY

Same accuracy as FP32, with no hyperparameter changes.

<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>

ILSVRC12 classification top-1 accuracy.
(Sharan Narang, Paulius Micikevicius et al., "Mixed Precision Training“, ICLR 2018)
**Same hyperparameters and learning rate schedule as FP32."
AMP FOR PYTORCH
AMP: AUTOMATIC MIXED PRECISION

Existing FP32 (default) script

->

Add 2 lines of Python

->

Accelerate your training with mixed precision
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device=“cuda”)
y = torch.randn(N, D_out, device=“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()
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
AMP.INITIALIZE()

Sets up your model(s) and optimizer(s) for mixed precision training.

```python
model, optimizer = amp.initialize(model, optimizer,
          opt_level,  # Required. Establishes a default set of under-the-hood properties that govern the chosen mode.
          cast_model_type=None,
          patch_torch_functions=None,
          keep_batchnorm_fp32=None,
          master_weights=None,
          loss_scale = None)
```

Optional property overrides, for finer-grained control
OPTIMIZATION LEVELS

OPT_LEVEL="O0"
FP32 training. Your incoming model should be FP32 already, so this is likely a no-op. O0 can be useful to establish an accuracy baseline.

O1
Mixed Precision. Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. Because casts occur in functions, model weights remain FP32.

O2
“Almost FP16” Mixed Precision. FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. Model weights, except batchnorm weights, are cast to FP16.

O3
FP16 training. O3 can be useful to establish the “speed of light” for your model. If your model uses batch normalization, add keep_batchnorm_fp32=True, which enables cudnn batchnorm.
NO MANUAL CASTS NEEDED

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

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O0")

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

    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
OPTIMIZATION LEVELS IN ACTION

https://github.com/NVIDIA/apex/tree/master/examples/imagenet

Mixed Precision (O1 and O2)
- **2X** faster than FP32
- Only ~6% overhead relative to “speed of light”
MIXED PRECISION GUIDANCE

1. **00** (FP32) first to establish an accuracy baseline.
2. Try **01** to enable mixed precision.
3. For the adventurous, try **02** or **03**, which may improve speed.
4. Experiment! The AMP API makes it easy to try different mixed precision modes and properties.
MIXED PRECISION PRINCIPLES IN AMP
MIXED PRECISION TRAINING PRINCIPLES

1. Accumulate in FP32.

2. Represent values in the appropriate dynamic range.
FP32 WEIGHTS

Weight updates are an accumulation.

1 + 0.0001 = ??

```python
param = torch.cuda.HalfTensor([1.0])
update = torch.cuda.HalfTensor([0.0001])
print(param + update)  # 1
```

In FP16, when update/param < 2^{-11} ≈ 0.00049, update has no effect.

```python
param = torch.cuda.FloatTensor([1.0])
update = torch.cuda.FloatTensor([0.0001])
print(param + update)  # 1.0001
```
MIXED PRECISION TRAINING PRINCIPLES

1. Accumulate in FP32.
   
   AMP maintains weights in FP32.

2. Represent values in the appropriate dynamic range.
GRADIENT UNDERFLOW

Small gradients may underflow in FP16 regions of the network.
Scaling the loss brings gradients into the FP16 dynamic range.
**LOSS SCALING**

Scaling the loss brings gradients into the FP16 dynamic range.

Unscale gradients in FP32 for optimizer.step()
LOSS SCALING

Scaling the loss brings gradients into the FP16 dynamic range.

1. Multiply the loss by some constant $S$.
   $\text{scaled\_loss} = \text{loss} \times S$

2. $\text{scaled\_loss}.\text{backward}()$
   By the chain rule, gradients will also be scaled by $S$.
   This preserves small gradient values.

3. Unscale gradients before $\text{optimizer}.\text{step}()$.**

** Unscaling ensures loss scaling does not affect the learning rate.
   Loss scaling does not require retuning the learning rate.
MIXED PRECISION TRAINING PRINCIPLES

1. Accumulate in FP32.
   
   AMP maintains weights in FP32.

2. Represent values in the appropriate dynamic range.
   
   AMP scales the network’s gradients.
OPT_LEVELS AND PROPERTIES

Each opt_level establishes a set of properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cast_model_type (torch.dtype)</td>
<td>Cast your model’s parameters and buffers to the desired type.</td>
</tr>
<tr>
<td>patch_torch_functions (True or False)</td>
<td>Patch all Torch functions to perform Tensor Core-friendly ops in FP16, and</td>
</tr>
<tr>
<td></td>
<td>any ops that benefit from FP32 precision in FP32.</td>
</tr>
<tr>
<td>keep_batchnorm_fp32 (True or False)</td>
<td>Maintain batchnorms in the desired type (typically torch.float32).</td>
</tr>
<tr>
<td>master_weights (True or False)</td>
<td>Maintain FP32 master weights for any FP16 model weights (applies to opt_level=&quot;O2&quot;).</td>
</tr>
<tr>
<td>loss_scale (float, or “dynamic”)</td>
<td>If loss_scale is a float value, use this value as the static (fixed) loss</td>
</tr>
<tr>
<td></td>
<td>scale. Otherwise, automatically adjust the loss scale as needed.</td>
</tr>
</tbody>
</table>
**OPT_LEVELS AND PROPERTIES**

**00**

FP32 training.
Your incoming model should be FP32 already, so this is likely a no-op. **00** can be useful to establish an accuracy baseline.

```python
cast_model_type=torch.float32
patch_torch_functions=False
keep_batchnorm_fp32=None
master_weights=False
loss_scale=1.0
```

**None indicates “not applicable.”**

**01**

Mixed Precision.
Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. Because casts occur in functions, model weights remain FP32.

```python
cast_model_type=None
patch_torch_functions=True
keep_batchnorm_fp32=None
master_weights=None
loss_scale="dynamic"
```

**Separate FP32 master weights are not applicable because the weights remain FP32.**
O1 (PATCH_TORCH_FUNCTIONS)

Patches `torch.*` functions to cast their inputs to the optimal type.

Ops that are fast and stable on Tensor Cores (GEMMs and Convolutions) run in **FP16**.

Ops that benefit from FP32 precision (softmax, exponentiation, pow) run in **FP32**.

Conceptual operation of patching:

```python
model, optim =
amp.initialize(model, optim,
  opt_level="O1")

for func in fp16_cast_list:
def create_casting_func(old_func):
def func_with_fp16_cast(input):
  return old_func(input.half())
  return func_with_cast
torch.func = create_casting_func(torch.func)

for func in fp32_cast_list:
def create_casting_func(old_func):
def func_with_fp32_cast(input):
  return old_func(input.half())
  return func_with_cast
torch.func = create_casting_func(torch.func)
```
“Almost FP16” Mixed Precision. FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. Model weights, except batchnorm weights, are cast to FP16.

```python
cast_model_type=torch.float16
patch_torch_functions=False
keep_batchnorm_fp32=True
master_weights=True
loss_scale="dynamic"
```

FP16 training. O3 can be useful to establish the “speed of light” for your model. If your model uses batch normalization, add the manual override `keep_batchnorm_fp32=True`, which enables cudnn batchnorm.

```python
cast_model_type=torch.float16
patch_torch_functions=False
keep_batchnorm_fp32=True
master_weights=True
loss_scale=1.0
```
OPT_LEVELS AND PROPERTIES

Properties for a given opt_level can be individually overridden.

```python
model, optimizer = amp.initialize(model, optimizer,
   opt_level="O1",  # Sets up loss_scale="dynamic" by default.
   loss_scale=128.0)  # Optional override: Tells Amp to use a static loss scale of 128.0 instead.
```
OPT_LEVELS AND PROPERTIES

Properties for a given opt_level can be individually overridden.

```python
model, optimizer = amp.initialize(model, optimizer,

    opt_level="O1",  # Sets up loss_scale="dynamic" by default.

    loss_scale=128.0)  # Optional override: Tells AMP to use a static loss scale of 128.0 instead.
```

AMP will issue a warning and explanation if you attempt to override a property that does not make sense.

For example, setting `opt_level="O1"` and the override `master_weights=True` does not make sense.
EXAMPLE REVISITED

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

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

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

    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
AMP casts weights and/or patches Torch functions based on opt_level and properties:

```python
code
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
```

AMP applies loss scaling as appropriate for the opt_level and properties:

```python
code
with amp.scale_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()
```
Available through the NVIDIA Apex repository of mixed precision and distributed tools: 
https://github.com/nvidia/apex

Full API documentation: 
https://nvidia.github.io/apex/

For more on mixed precision, don’t forget to see:

Myle Ott and Sergey Edunov, *Taking Advantage of Mixed Precision to Accelerate Training Using PyTorch*, GTC 2019 Session 9832
Right after this talk in Room 210D

Carl Case, *Mixed Precision Training of Deep Neural Networks*,
GTC 2019 Session 9143
TENSOR CORE PERFORMANCE TIPS
TENSOR CORE PERFORMANCE TIPS

- **GEMMs** = “generalized (dense) matrix-matrix multiplies”:
  For $A \times B$ where $A$ has size $(M, K)$ and $B$ has size $(K, N)$:
  $N, M, K$ should be multiples of 8.

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

- **GEMMs in RNNs**:
  Batch size, hidden size, embedding size, and dictionary size 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 Pytorch’s autotuner.

The first iteration, it will test different cuDNN algorithms for each new convolution size it sees, and cache the fastest choice to use in later iterations.

See https://discuss.pytorch.org/t/what-does-torch-backends-cudnn-benchmark-do/5936
ENSURING FP32 WEIGHT UPDATES

```python
optimizer = torch.optim.SGD(model.parameters())
model, optimizer = amp.initialize(model, optimizer, opt_level="O2")
```

After casting requested by O2, `model` may contain a mixture of fp16 and fp32 params.

`optimizer` references point to model params.
ENSURING FP32 WEIGHT UPDATES

```python
optimizer = torch.optim.SGD(model.parameters())
model, optimizer = amp.initialize(model, optimizer, opt_level="O2")
```

With O2, AMP maintains **FP32 master params** for any **FP16 params**.

Patches optimizer’s references to point to master params. `optimizer.step()` acts on master params.