Taking Advantage of Low Precision to Accelerate Training and Inference Using PyTorch

Presented by:
Myle Ott and Sergey Edunov
Facebook AI Research (FAIR)
Overview

Mixed precision training in PyTorch:
- 3-4x speedups in training wall time
- Reduced memory usage ==> bigger batch sizes
- No architecture changes required

Case study: Neural Machine Translation
- Train models in 30 minutes instead of 1 day+
- Semi-supervised training over much larger datasets
What are Tensor Cores?

- Optimized hardware units for mixed precision matrix-multiply-and-accumulate: \( D = A \times B + C \)
cuBLAS Mixed-Precision GEMM
(FP16 Input, FP32 Compute)

<table>
<thead>
<tr>
<th>Matrix Size (M=N=K)</th>
<th>Tesla P100</th>
<th>Tesla V100 (Tensor Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relative Performance

Slide credit: Nvidia
If only it were this easy...

model.half()
Why not pure FP16?

FP16 has insufficient range/precision for some ops

Better to leave some ops in FP32:
- Large reductions, e.g., norms, softmax, etc.
- Pointwise ops where $|f(x)| >> |x|$, e.g., exp, pow, log, etc.
Why not pure FP16?

In practice, pure FP16 hurts optimization.

According to Nvidia:

- Sum of FP16 values whose ratio is $>2^{11}$ is just the larger value
- Weight update: if $w >> lr * dw$ then update doesn’t change $w$
Why not pure FP16?

**Solution**: mixed precision training

Optimize in FP32 and use FP16 for almost* everything else

* Some operations should still happen in FP32:
  - Large reductions, e.g., norms, softmax, etc.
  - Pointwise ops where \(|f(x)| >> |x|\), e.g., exp, pow, log, etc.
Optimizing in FP32
Optimizing in FP32

FP16 Weights → Forward Pass → FP16 Loss → FP16 Gradients → FP32 Master Gradients

Copy → Backprop
Optimizing in FP32

- **FP32 Master Weights**
- **FP32 Master Gradients**
- **FP16 Weights**
- **FP16 Loss**
- **FP16 Gradients**

**Forward Pass**
- FP16 Weights → FP16 Loss

**Backprop**
- FP16 Loss → FP16 Gradients

**Copy**
- FP32 Master Gradients → FP16 Gradients

**Apply**
- FP32 Master Weights → FP32 Master Gradients
Optimizing in FP32

- **FP32 Master Weights**
- **FP16 Weights**
- **FP32 Master Gradients**
- **FP16 Gradients**
- **FP16 Loss**

- Forward Pass
- Copy
- Backprop
- Apply
- Copy
Optimizing in FP32

FP32 Master
Weights

Copy

FP16
Weights

Forward Pass

Apply

This adds overhead!

It’s only worth it because of the Tensor Cores. Don’t use mixed precision without Tensor Cores!
Gradient underflow

- FP16 has a smaller representable range than FP32 (shown in blue)
- In practice gradient are quite small, so there’s a risk of underflow
Gradient underflow

Underflow can **not** be detected

But if we scale loss up

If we scale the loss up by $K$, by the chain rule of derivatives, gradients will be $K$ times bigger.
Gradient overflow

If overflow detected
Scale the loss down
Avoiding under/overflow by loss scaling
Avoiding under/overflow by loss scaling

If gradients overflow ($\infty$), throw away the batch
Avoiding under/overflow by loss scaling

FP16Weights

Forward Pass

FP16 Loss

Loss Scaling

Scaled FP32 Gradients

Copy

Scaled FP16 Loss

Backprop

Scaled FP16 Gradients

If gradients overflow (\(\text{inf}\)), throw away the batch
Avoiding under/overflow by loss scaling

- **FP16 Weights**
  - Forward Pass

- **FP16 Loss**
  - Loss Scaling

- **FP32 Gradients**
  - Remove scale

- **Scaled FP32 Gradients**
  - Copy

- **Scaled FP16 Gradients**
  - Backprop

- If gradients overflow ($\text{inf}$), throw away the batch
Avoiding under/overflow by loss scaling

If gradients overflow ($\infty$), throw away the batch
How to pick the scaling constant (K)

- Too small and gradient will underflow
- Too big and we’ll waste compute due to overflow
- In practice the optimal scaling constant changes during training
- We can adjust it dynamically!
Dynamic loss scaling

- Every time the gradient overflows (∞), reduce the scaling constant by a factor of 2

- If the gradients haven’t overflowed in the last N updates (~1000), then increase the scaling constant by a factor of 2
Dynamic loss scaling
So far...

Tensor Cores make FP16 ops 4-9x faster

Mixed precision training:

- Forward/backward in FP16
- Optimize in FP32
- Requires maintaining two copies of the model weights
- Dynamically scale the loss to avoid gradient under/overflow
One more thing about FP16...

For maximal safety, perform ops that sum many values in FP32
- e.g., normalization layers, softmax, L1 or L2 norm, etc.
- This includes most Loss layers, e.g., CrossEntropyLoss

General advice: compute your loss in FP32 too
The full picture
The full picture
The full picture

FP16 Weights → FP32 Loss → Scaled FP32 Loss → Scaled FP16 Gradients

- Forward Pass
- Loss Scaling
- Backprop

If gradients overflow (\(\infty\)), throw away the batch
The full picture

FP16 Weights → Forward Pass → FP32 Loss

Loss Scaling → Scaled FP32 Loss

Scaled FP32 Gradients → Copy → Scaled FP16 Gradients

If gradients overflow (inf), throw away the batch
The full picture

- **FP16 Weights**
  - Forward Pass

- **FP32 Loss**
  - Loss Scaling

- **Scaled FP32 Loss**
  - Remove scale

- **Scaled FP32 Gradients**
  - Copy

- **Scaled FP16 Gradients**
  - Backprop

If gradients overflow (inf), throw away the batch.
The full picture

- **FP32 Master Weights**
- **FP16 Weights**
- **FP32 Loss**
- **Scaled FP32 Loss**
- **Scaled FP32 Gradients**
- **Scaled FP16 Gradients**

Flow:
- **Copy** FP32 Master Weights to FP16 Weights.
- **Forward Pass** through FP32 Loss.
- **Backprop** to Scaled FP32 Gradients.
- **Copy** Scaled FP32 Gradients.
- **Remove scale**
- **Apply** to FP32 Gradients.
- **Copy** to Scaled FP16 Gradients.
- **If gradients overflow (inf), throw away the batch.**
The full picture

Distributed gradient accumulation / all-reduce

- **FP32 Master Weights**
  - Apply

- **FP32 Gradients**
  - Remove scale

- **Scaled FP32 Gradients**
  - Copy

- **Scaled FP16 Gradients**
  - Option 1 (slower)
  - Option 2 (faster)

- **FP16 Weights**
  - Copy
  - Forward Pass

- **FP32 Loss**
  - Loss Scaling

- **Scaled FP32 Loss**

If gradients overflow (\(\text{inf}\)), throw away the batch
In PyTorch

To automate the recipe, start with Nvidia’s `apex.amp` library:

```python
from apex import amp
optim = torch.optim.Adam(...)  
model, optim = amp.initialize(model, optim, opt_level="O1") 
(...)

with amp.scale_loss(loss, optim) as scaled_loss:
    scaled_loss.backward()
optim.step()
```
Making it even faster

`apex.amp` supports different optimization levels

`opt_level="O1"` is conservative and keeps many ops in FP32

`opt_level="O2"` is faster, but may require manually converting some ops to FP32 to achieve good results

More details at: [https://nvidia.github.io/apex/](https://nvidia.github.io/apex/)
Making it even faster

A useful pattern:

```
x = torch.nn.functional.softmax(x, dtype=torch.float32).type_as(x)
```

When `x` is FP16 (i.e., a `torch.HalfTensor`):
- Computes the softmax in FP32 and casts back to FP16

When `x` is FP32 (i.e., a `torch.FloatTensor`):
- No impact on speed or memory
One more thing...

Must have GPU with Tensor Cores (Volta+), CUDA 9.1 or newer

Additionally:

- Batch size should be a multiple of 8
- M, N and K for matmul should be multiples of 8
- Dictionaries/embed layers should be padded to be a multiple of 8
Summary

Mixed precision training gives:

- Tensor Cores make FP16 ops 4-9x faster
- No architecture changes required
- Use Nvidia’s apex library

Tradeoffs:

- Some extra bookkeeping required (mostly handled by apex)
- Best perf requires manual fixes for softmax, layernorm, etc.
Scaling Machine Translation
Sequence to Sequence Learning

Bonjour à tous ! → Hello everybody!

- **Sequence to sequence** mapping
- Input = sequence, output = sequence
- **Structured prediction** problem
Sequence to Sequence Learning

- machine translation
- text summarization
- writing stories
- question generation
- dialogue, chatbots
- paraphrasing
- ...

![Diagram of Sequence to Sequence Learning](image)
Why do we need to scale?

- Large benchmark ~2.4 billion words + much more unlabeled data
- Training time: CNNs up to 38 days on 8 M40 GPUs (Gehring et al., 2017)
- Train many models
- Support Multilingual training
Reducing training time

Time in minutes to train "Transformer" translation model on Volta V100 GPUs (WMT En-De)

- Original: 1,429 minutes
- +16-bit: 1,429 minutes
- + cumul: 43 minutes
- +2x lr: 1,429 minutes
- 16 nodes: 1,429 minutes
- +overlap: 1,429 minutes
Reducing training time

Time in minutes to train "Transformer" translation model on Volta V100 GPUs (WMT En-De)

1,429
495

3x faster (wall time) using the same hardware, model architecture and bsz!

Train Time (Minutes)

Original +16-bit + cumul +2x lr 16 nodes +overlap
Reducing training time

Time in minutes to train "Transformer" translation model on Volta V100 GPUs (WMT En-De)

Train Time (Minutes)

- Original: 1,429
- +16-bit: 495
- + cumul: 447
- +2x lr
- 16 nodes
- +overlap

GPU 1
GPU 2
GPU 3
GPU 4

Sync After 1
Sync After 2

Gradient
Forward/Backward
Idle

Time
Reducing training time

Time in minutes to train “Transformer” translation model on Volta V100 GPUs (WMT En-De)
Reducing training time

Time in minutes to train “Transformer” translation model on Volta V100 GPUs (WMT En-De)
Reducing training time

Time in minutes to train “Transformer” translation model on Volta V100 GPUs (WMT En-De)

Implemented in PyTorch’s DistributedDataParallel

Train Time (Minutes)

1,429
495
447
311
37
32

Original
+16-bit
+ cumul
+2x lr
16 nodes
+overlap

GPU 1
GPU 2
GPU 3
GPU 4

Sync After Backward
Overlap Sync with Backward

Gradient Sync
Forward
Backward
Idle
Semi-supervised machine translation
Data augmentation for Translation

Back-translation (Bojar & Tamchyna, 2011; Sennrich et al., 2016)
Data augmentation for Translation
Back-translation (Bojar & Tamchyna, 2011; Sennrich et al., 2016)
Bilingual

Monolingual Source

Monolingual Generated

English

German

Final Model
Scaling from 100M to 5.8B words

Model trains in 22.5h on 128 V100

BLEU (Accuracy)

<table>
<thead>
<tr>
<th>Method</th>
<th>BLEU Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairseq &amp; sampled BT</td>
<td>35</td>
</tr>
<tr>
<td>DeepL (2017)</td>
<td>33.3</td>
</tr>
<tr>
<td>SAtt + RPR (Google, 2018)</td>
<td>29.2</td>
</tr>
<tr>
<td>WTransformer (Salesforce, 2017)</td>
<td>28.9</td>
</tr>
<tr>
<td>Transformer (Google, 2017)</td>
<td>28.4</td>
</tr>
<tr>
<td>ConvS2S (2017)</td>
<td>25.2</td>
</tr>
<tr>
<td>GNMT (RNN, 2016)</td>
<td>24.6</td>
</tr>
<tr>
<td>Phrase-based (2014)</td>
<td>20.7</td>
</tr>
</tbody>
</table>

WMT’14 English-German

Only benchmark bilingual + monolingual data
High quality, non-benchmark data
WMT’18 Human evaluations

Ranked #1 in the human evaluation of the WMT'18 English-German translation task

<table>
<thead>
<tr>
<th>Ave. %</th>
<th>Ave. z</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>85.5</td>
<td>0.653</td>
<td>FACEBOOK-FAIR *</td>
</tr>
<tr>
<td>82.2</td>
<td>0.561</td>
<td>ONLINE-B</td>
</tr>
<tr>
<td>81.9</td>
<td>0.551</td>
<td>MICROSOFT-MARIAN</td>
</tr>
<tr>
<td>81.6</td>
<td>0.539</td>
<td>MMT-PRODUCTION</td>
</tr>
<tr>
<td>82.3</td>
<td>0.537</td>
<td>UCAM</td>
</tr>
<tr>
<td>80.2</td>
<td>0.491</td>
<td>NTT</td>
</tr>
<tr>
<td>79.3</td>
<td>0.454</td>
<td>KIT</td>
</tr>
<tr>
<td>77.7</td>
<td>0.396</td>
<td>ONLINE-Y</td>
</tr>
<tr>
<td>76.7</td>
<td>0.377</td>
<td>JHU</td>
</tr>
<tr>
<td>76.3</td>
<td>0.352</td>
<td>UEDIN</td>
</tr>
<tr>
<td>71.8</td>
<td>0.213</td>
<td>LMU-NMT</td>
</tr>
<tr>
<td>67.4</td>
<td>0.060</td>
<td>ONLINE-A</td>
</tr>
<tr>
<td>53.2</td>
<td>-0.385</td>
<td>ONLINE-F</td>
</tr>
<tr>
<td>53.8</td>
<td>-0.416</td>
<td>ONLINE-G</td>
</tr>
<tr>
<td>36.7</td>
<td>-0.966</td>
<td>RWTH-UNSUPER</td>
</tr>
<tr>
<td>32.6</td>
<td>-1.122</td>
<td>LMU-UNSUP</td>
</tr>
</tbody>
</table>
Conclusion

Mixed precision training in PyTorch:
- 3-4x speedups in training wall time
- No architecture changes required
- Use Nvidia’s apex library

Case study: Neural Machine Translation
- Train models in 30 minutes instead of 1 day+
- State-of-the-art translation quality using semi-supervised learning
Thank you! Questions?

Contact Us

Myle Ott                Sergey Edunov
myleott@fb.com          edunov@fb.com

References

• Scaling Neural Machine Translation: arxiv.org/abs/1806.00187
• Understanding Back-translation at Scale: arxiv.org/abs/1808.09381
• apex: nvidia.github.io/apex

Acknowledgements: Nvidia and PyTorch teams for helping us implement and optimize mixed precision training.