MIXED PRECISION TRAINING:
THEORY AND PRACTICE
Paulius Micikevicius
What is Mixed Precision Training?

• Reduced precision tensor math with FP32 accumulation, FP16 storage
• Successfully used to train a variety of:
  • Well known public networks
  • Variety of NVIDIA research networks
  • Variety of NVIDIA automotive networks
Benefits of Mixed Precision Training

• **Accelerates math**
  • TensorCores have 8x higher throughput than FP32
  • 125 Tflops theory

• **Reduces memory bandwidth pressure:**
  • FP16 halves the memory traffic compared to FP32

• **Reduces memory consumption**
  • Halve the size of activation and gradient tensors
  • Enables larger minibatches or larger input sizes

(C) NVIDIA
Volta TensorCores

- Used by cuDNN and CUBLAS libraries
- Exposed in CUDA as WMMA
- **Accelerate convolutions and matrix multiplication**
  - A single instruction multiply-accumulates matrices
  - Think: computes many dot-products in parallel

![Diagram showing the process of convolutions and matrix multiplication using Volta TensorCores.](image_url)
Training results with mixed precision

- Successfully applied to a wide variety of networks including:
  - Imagenet CNNs
  - Detection
  - Language Translation
  - Speech
  - Text to Speech
  - GAN
  - Image enhancement (inpainting, upscaling, pix2pix, etc.)
  - Wavenet

- More details later in this talk
Considerations for Mixed Precision Training

• Which precision to use for storage, for math?

• Instructive to walk through by DNN operation type:
  • Weight update
  • Point-wise
  • Reduction
  • Convolution, Matrix multiply
Guideline #1 for mixed precision: weight update

• FP16 mantissa is sufficient for some networks, some require FP32

• Sum of FP16 values whose ratio is greater than $2^{11}$ is just the larger value
  • FP16 has a 10-bit mantissa, binary points have to be aligned for addition
  • Weight update: if $w \gg lr \cdot dw$ then update doesn’t change $w$
    • Examples: multiplying a value by 0.01 leads to $\sim2^7$ ratio, 0.001 leads to $\sim2^{10}$ ratio

• Conservative recommendation:
  • FP32 update:
    • Compute weight update in FP32
    • Keep a master copy of weights in FP32, make an FP16 copy for fwd/bwd passes

• If FP32 storage is a burden, try FP16 – it does work for some nets
  • ie convnets
Guideline #2 for mixed precision: pointwise

• FP16 is safe for most of these: ReLU, Sigmoid, Tanh, Add, ...
  • Inputs and outputs to these are value in a narrow range around 0
  • FP16 storage saves bandwidth -> reduces time

• FP32 math and storage is recommended for:
  • operations $f$ where $|f(x)| >> |x|$
    • Examples: Exp, Square, Log, Cross-entropy
  • These typically occur as part of a normalization or loss layer that is unfused
  • FP32 ensures high precision, no perf impact since bandwidth limited

• Conservative recommendation:
  • Leave pointwise ops in FP32 (math and storage) unless they are known types
  • Pointwise op fusion is a good next step for performance
    • Use libraries for efficient fused pointwise ops for common layers (eg BatcNorm)
DNN Operation: Reductions

• **Examples:**
  • Large sums of values: L1 norm, L2 norm, Softmax

• **FP32 Math:**
  • Avoids overflows
  • Does not affect speed – these operations are memory limited

• **Storage:**
  • FP32 output
  • Input can be FP16 if the preceding operation outputs FP16
    • If your training frameworks supports different input and output types for an op
    • Saves bandwidth -> some speedup

(C) NVIDIA
A Note on Normalization and Loss Layers

• **Normalizations:**
  • Usually constructed from primitive ops (reductions, squares, exp, scale)
  • Storage:
    • Input and normalized output can be in FP16
    • Intermediate results should be stored in FP32
  • Ideally should be fused into a single op:
    • Avoids round-trips to memory -> faster
    • Avoids intermediate storage

• **Loss, probability layers:**
  • Softmax, cross-entropy, attention modules
  • FP32 math, FP32 output
DNN Operation: Convolution, Matrix Multiply

• Fundamentally these are collections of dot-products

• Math: Tensor Cores starting with Volta GPUs
  • Training: use FP32 accumulation
  • Inference: FP16 accumulation can be used
  • Many frameworks have integrated libraries with TensorCore support
    • http://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/

• FP16 Storage (input and output)
Summary so far

• **FP32 Master weights and update**
• **Math: FP32 and TensorCores**
• **Storage:**
  • Use FP16 for most layers
  • Use FP32 for layers that output probabilities or large magnitude values
    • Fuse to optimize speed and storage

• **Example layer time breakdowns for FP32-only training:**
  • Resnet50 : ~73% convolutions, 27% other
  • DS2: ~90% convolutions and matrix multiplies (LSTM), ~10% other

• **One more mixed-precision consideration: Loss Scaling**
  • Scale the loss, unscale the weight gradients before update/clipping/etc.
  • Preserves small gradient values
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: ~40 powers of 2

Gradients are small, don’t use much of FP16 range
FP16 range not used by gradients: ~15 powers of 2

**Loss Scaling:**
multiply the loss by some constant $s$
by chain rule backprop scales gradients by $s$
preserves small gradient values
unscale the weight gradient before update
Loss Scaling

• **Algorithm**
  • Pick a scaling factor \( s \)
  • for each training iteration
    • Make an fp16 copy of weights
    • Fwd prop (fp16 weights and activations)
    • Scale the loss by \( s \)
    • Bwd prop (fp16 weights, activations, and gradients)
    • Scale \( dW \) by \( 1/s \)
    • Update \( W \)

• **For simplicity:**
  • Apply gradient clipping and similar operations on gradients after \( 1/s \) scaling
  • Avoids the need to change hyperparameters to account for scaling

• **For maximum performance: fuse unscaling and update**
  • Reduces memory accesses
  • Avoids storing weight gradients in fp32
Automatic Loss Scaling

• Frees users from choosing a scaling factor
  • Too small a factor doesn’t retain enough small values
  • Too large a factor causes overflows

• Algorithm
  • Start with a large scaling factor $s$
  • for each training iteration
    • Make an fp16 copy of weights
    • Fwd prop
    • Scale the loss by $s$
    • Bwd prop
    • Update scaling factor $s$
      • If $dW$ contains Inf/NaN then reduce $s$, skip the update
      • If no Inf/NaN were detected for $N$ updates then increase $s$
    • Scale $dW$ by $1/s$
    • Update $W$
Automatic Loss Scale Factor for a Translation Net

Smallest scaling factor = $2^{20}$ -> max $dW$ magnitude didn’t exceed $2^{-5}$
Update Skipping

• Must skip updating:
  • Weights
  • Momenta

• Additional considerations:
  • Iteration count:
    • Always increment: may result in fewer updates than iterations
    • Don’t increment when skipping:
      • Ensures the same number of updates as without skipping enabled
      • Ensures the same number of updates with a given learning rate
  • Input minibatch: just “move on”
Automatic Loss Scaling Parameters

- **Factor for increasing/decreasing loss-scaling**
  - In all our experiments we use 2

- **Number of iterations without overflow**
  - In all our experiments we use $N = 2,000$
  - Separate study showed that randomly skipping 0.1% of updates didn’t affect result
  - $N = 2,000$ gives extra margin by skipping at most 0.05% of updates in steady state

- **Iteration count:**
  - We did not observe model accuracy difference between incrementing and not incrementing iteration count on skips
# ILSVRC12 Classification Networks, Top-1 Accuracy

<table>
<thead>
<tr>
<th></th>
<th>FP32 Baseline</th>
<th>Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>56.8%</td>
<td>56.9%</td>
</tr>
<tr>
<td>VGG-D</td>
<td>65.4%</td>
<td>65.4%</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>68.3%</td>
<td>68.4%</td>
</tr>
<tr>
<td>Inception v2</td>
<td>70.0%</td>
<td>70.0%</td>
</tr>
<tr>
<td>Inception v3</td>
<td>73.9%</td>
<td>74.1%</td>
</tr>
<tr>
<td>Resnet 50</td>
<td>75.9%</td>
<td>76.0%</td>
</tr>
<tr>
<td>ResNeXt 50</td>
<td>77.3%</td>
<td>77.5%</td>
</tr>
</tbody>
</table>

A number of these train fine in mixed precision even without loss-scaling.
Detection Networks, mAP

<table>
<thead>
<tr>
<th></th>
<th>FP32 Baseline</th>
<th>Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster R-CNN, VOC 07 data</td>
<td>69.1%</td>
<td>69.7%</td>
</tr>
<tr>
<td>Multibox SSD, VOC 07+12 data</td>
<td>76.9%</td>
<td>77.1%</td>
</tr>
</tbody>
</table>

NVIDIA’s proprietary automotive networks train with mixed-precision matching FP32 baseline accuracy.
Language Translation

• GNMT:
  • https://github.com/tensorflow/nmt
  • German -> English (train on WMT, test on newstest2015)
  • 8 layer encoder, 8 layer decoder, 1024x LSTM cells, attention
  • **FP32 and Mixed Precision: ~29 BLEU using SGD**
    • Both equally lower with Adam, match the paper

• FairSeq:
  • https://github.com/facebookresearch/fairseq
  • Convolutional net for translation, English - French
  • **FP32 and Mixed Precision: ~40.5 BLEU** after 12 epochs
Speech

- Courtesy of Baidu
  - 2 2D-conv layers, 3 GRU layers, 1D conv
  - Baidu internal datasets

<table>
<thead>
<tr>
<th></th>
<th>FP32 Baseline</th>
<th>Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>2.20</td>
<td>1.99</td>
</tr>
<tr>
<td>Mandarin</td>
<td>15.82</td>
<td>15.01</td>
</tr>
</tbody>
</table>
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
Sentiment Analysis

• Multiplicative LSTM, based on https://arxiv.org/abs/1704.01444

<table>
<thead>
<tr>
<th></th>
<th>Train BPC</th>
<th>Val BPC</th>
<th>SST acc</th>
<th>IMDB acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>1.116</td>
<td>1.073</td>
<td>91.8</td>
<td>92.8</td>
</tr>
<tr>
<td>Mixed Precision</td>
<td>1.115</td>
<td>1.075</td>
<td>91.9</td>
<td>92.8</td>
</tr>
</tbody>
</table>
Image Inpainting

• Fill in arbitrary holes
• Network Architecture:
  • U-Net with partial convolution
  • VGG16 based Perceptual loss + Style loss
• Speedup: 3x, at 2x bigger batch size
  • We can increase batch size only in mixed precision
Image Inpainting: result

Training Loss Curve

Testing Input

Mixed Precision Result

FP32 Result

(C) NVIDIA
Text to speech synthesis

Using Tacotron 2

Fig. 1. Block diagram of the Tacotron 2 system architecture.

Text to speech synthesis: results

Predicted Mel-Spectrograms

Mixed Precision: Pink
FP32: Green

Predicted Alignments

Mixed Precision
FP32
Wavenet

- 12 Layers of dilated convolutions
- Dilations reset every 6 layers
- 128 channels for dilated convs. (64 per nonlinearity)
- 64 channels for residual convs.
- 256 channels for skip convs.
Wavenet: results

Mixed precision: Pink  FP32: Green
Speedups

- Memory limited ops: should see ~2x speedup
- Math limited ops: will vary based on arithmetic intensity
- Some examples, mixed precision vs FP32 on GV100:
  - Resnet50: ~3.3x
  - DeepSpeech2: ~4.5x
  - FairSeq: ~4.0x
  - Sentiment prediction: ~4.0x
- Speedups to increase further:
  - libraries are continuously optimized
  - TensorCore paths are being added to more operation varieties

(C) NVIDIA
TensorCore Performance Guidance

• **Requirements to trigger TensorCore operations:**
  • Convolutions:
    • Number of input channels a multiple of 8
    • Number of output channels a multiple of 8
  • Matrix Multiplies:
    • M, N, K sizes should be multiples of 8
    • Larger K sizes make multiplications more efficient (amortize the write overhead)
    • Makes wider recurrent cells more practical ($K$ is input layer width)

• **If you’re designing models**
  • Make sure to choose layer widths that are multiples of 8
  • Pad input/output dictionaries to multiples of 8
    • Speeds up embedding/projection operations

• **If you’re developing new cells**
  • Concatenate cell matrix ops into a single call
Conclusions

• **Mixed precision training benefits:**
  • Math, memory speedups
  • Larger minibatches, larger inputs

• **Automatic Loss Scaling simplifies mixed precision training**

• **Mixed precision matches FP32 training accuracy for a variety of:**
  • **Tasks**: classification, regression, generation
  • **Problem domains**: images, language translation, language modeling, speech
  • **Network architectures**: feed forward, recurrent
  • **Optimizers**: SGD, Adagrad, Adam

• **Note on inference:**
  • Can be purely FP16: storage and math (use library calls with FP16 accumulation)

• **More details:**
  • S81012: Training Neural Networks with Mixed Precision: Real Examples (Thu, 9am)
We are hiring

• **Deep Learning Compute Architect:**
  • Study DNN performance, accuracy, precision, etc.
  • Propose improvements to future HW, see them through the HW cycle