TRAINING WITH MIXED PRECISION

Boris Ginsburg, Sergei Nikolaev, Paulius Micikevicius
bginsburg, pauliusm, snikolaev@nvidia.com
05/11/2017
ACKNOWLEDGMENTS

Michael Houston, Hao Wu, Oleksii Kuchaiev, Ahmad Kiswani, Amir Gholaminejad, Ujval Kapasi, Jonah Alben, Alex Fit-Florea, Slawomir Kierat and

cuDNN team

This work is based on NVIDIA branch of caffe
https://github.com/NVIDIA/caffe (caffe-0.16)
AGENDA

1. Mixed precision training with Volta TensorOps
2. More aggressive training methods
   • FP16 training
   • FP16 master weights
3. Nvcaffe float16 internals
**SOME TERMINOLOGY**

<table>
<thead>
<tr>
<th>Training values storage</th>
<th>Matrix-Mult Accumulator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>FP32 training</td>
</tr>
<tr>
<td>FP16</td>
<td>FP32</td>
<td>Mixed precision training</td>
</tr>
<tr>
<td>FP16</td>
<td>FP16</td>
<td>FP16 training</td>
</tr>
</tbody>
</table>

With mixed or FP16 training, master weights can be FP16 or FP32.

**Volta:** Mixed precision training with FP32 master weight storage.
VOLTA TRAINING METHOD

Master-W (F32) → W (F16) → W (F16) → Actv → F16 → F16 → Actv → BWD-A → Actv Grad → F16 → F16 → W (F16) → Actv Grad → BWD-W → Actv Grad → F16 → F16 → W Grad → F16 → F16 → Weight Update → F32 → Updated Master-W (F32)
HALF-PRECISION FLOAT (FLOAT16)

FLOAT16 has wide range \(2^{40}\) ... but not as wide as FP32!

Normal range: \([6 \times 10^{-5}, 65504]\)
Sub-normal range: \([6 \times 10^{-8}, 6 \times 10^{-5}]\)
TRAINING FLOW

FORWARD PASS

\[ Y_k = W_k * Y_{k-1} \]

\[ Y_2 = W_2 * Y_1 \]

\[ Y_1 = W_1 * X \]

\[ \text{Loss } E \rightarrow dE/dY_k \]

BACKPROP

\[ dE/dY_{k-1} = dE/dY_k * W_k \]

\[ dE/dW_k = dE/dY_k * Y_{k-1} \]

\[ dE/dY_1 = dE/dY_2 * W_2 \]

\[ dE/dW_2 = dE/dY_2 * Y_1 \]

\[ dE/dY_1 = dE/dX * W_1 \]

\[ dE/dW_1 = dE/dY_1 * X \]

WEIGHT UPDATE

\[ W_k = W_k - \lambda * \frac{dE}{dW_k} \]

\[ W_2 = W_2 - \lambda * \frac{dE}{dW_2} \]

\[ W_1 = W_1 - \lambda * \frac{dE}{dW_1} \]

FORWARD PASS

\[ Y_k = W_k * Y_{k-1} \]

\[ Y_2 = W_2 * Y_1 \]

\[ Y_1 = W_1 * X \]

\[ \text{Loss } E \rightarrow dE/dY_k \]
TENSOR CORE 4X4X4 MATRIX-MULTIPLY ACC

\[ D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} \]
VOLTA TENSOR OPERATION

FP16 storage/input

Full precision product

Sum with FP32 accumulator

Convert to FP32 result

Also supports FP16 accumulator mode for inferencing
SOME NETWORKS TRAINED OUT OF THE BOX

TensorOp training matched the results of F32 training
  Same hyper-parameters as F32
  Same solver and training schedule as F32

Image classification nets (trained on ILSVRC12):
  No batch norm: GoogLeNet, VGG-D
  With batch norm: Inception v1, Resnet50
  All used SGD with momentum solver

GAN
  DCGAN-based, 8-layer generator, 7-layer discriminator
  Used Adam solver
GOOGLENET
INCEPTION V1

Graph showing performance metrics for different hardware options over iterations.
RESNET50
SOME NETWORKS NEEDED HELP

Networks:

Image classification: CaffeNet

Was not learning out of the box, even with F32 math when storage is F16

Detection nets:

Multibox SSD with VGG-D backbone

- Was not learning, even with F32 math when storage is F16

Faster R-CNN with VGG-D backbone

- 68.5% mAP, compared to 69.1% mAP with F32

Recurrent nets:

Seq2seq with attention: lagged behind F32 in perplexity

bigLSTM: diverged after some training

Remedy in all the cases: scale the loss value to “shift” gradients
LOSS SCALING

To shift gradients $dE/dX$ we will scale up the loss function by constant (e.g. by 1000):

layer {
    type: "SoftmaxWithLoss"
    loss_weight: 1000.
}

and adjust learning rate and weight decay accordingly:

base_lr: 0.01 0.00001 # 0.01 / 1000
weight_decay: 0.0005 0.5 # 0.0005 * 1000
MULTIBOX SSD: ACTIVATION GRADIENT MAGNITUDE HISTOGRAM
MULTIBOX SSD: ACTIVATION GRADIENT MAGNITUDE HISTOGRAM

activation gradient magnitudes

Become 0 in F16

Become denormals in F16

Percentage of values during training

Upper bound, 2 to the listed exponent
MULTIBOX SSD: ACTIVATION GRADIENT MAGNITUDE HISTOGRAM

Become 0 in F16
Become denormals in F16
Unused

Overall FP16 range
MULTIBOX: SCALING LOSS AND GRADIENTS

Loss scaled by 256
Consequently, gradients get scaled by 256
By chain rule

Benefits:
Hardly any activation gradients become 0 in F16
Most weight gradients become normalized values in F16
DETECTION TRAINING RESULTS

Multibox-SSD mAP:

F32: 76.9%

F16: 77.1%, loss scaled by 256

Without scaling: doesn’t learn

TensorOp: in flight

matching F32 at 74.1% mAP halfway through training

Faster-RCNN mAP:

F32: 69.1%

TensorOp: 69.7%, loss scaled by 256, without loss-scaling: 68.5%
SEQ2SEQ TRANSLATION NETWORK

WMT15 English to French Translation

seq2seq networks with attention:
  Based on TensorFlow tutorial
  3x1024 LSTM
  5x1024 LSTM

Word vocabularies:
  100K English
  40K French

SGD solver
SEQ2SEQ: 3X1024 LSTM

![Graph showing training perplexity over iterations for different configurations (F32 ref1, F32 ref2, F32 ref3, Volta, loss-scale 1024, Volta loss-scale 1) with a descending trend as iterations increase.]
SEQ2SEQ: 5X1024 LSTM

![Training Perplexity Graph](image-url)
1 Billion Word Language Benchmark

BigLSTM:

- Based on “Exploring the Limits of Language Modeling”
- 2x8192 LSTM, 1024 Projection
- Plus a few variants
- 800K word vocabulary

Adagrad solver
BIGLSTM: 2X8192 LSTM, 1024 PROJECTION
Guidelines for Training with Mixed Precision / TensorOps
TRAINING WITH MIXED PRECISION

• A number of cases train “out of the box”
  – F16 storage and TensorOps for fwd/bwd pass: weights, activations, gradients
  – F32 math for Batch Normalization parameters
  – F32 “master-copy” of weights for weights update

• When out of the box didn’t work:
  – Gradient values were too small when converted to F16
  – Solved in all cases with loss scaling
OBSERVATIONS ON GRADIENT VALUES

FP16 range is large
\[2^{40}\] including denorms

Gradient range is biased low vs standard FP16 range

Max magnitude we’ve seen was \(O(2^3)\)

Enables us to “shift” values without overflowing

Maximum magnitudes:

weight-grad >> activation-grad

For all the nets we’ve looked at
PART 2

More aggressive training exploration:

- FP16 training
- FP16 master weight storage
# ALEXNET : COMPARISON OF RESULTS

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>58.62</td>
<td>81.25</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>58.12</td>
<td>80.71</td>
</tr>
<tr>
<td>FP16 training</td>
<td>54.89</td>
<td>78.12</td>
</tr>
<tr>
<td>FP16 training, loss scale = 1000</td>
<td>57.76</td>
<td>80.76</td>
</tr>
</tbody>
</table>

Nvcaffe-0.16, DGX-1, SGD with momentum, 100 epochs, batch=1024, no augmentation, 1 crop, 1 model.
ALEXNET : FP16 TRAINING WITH SCALING

With loss scale factor = 1000, FP16 training matches other training curves (TensorOp and FP32)
ALEXNET: FP16 MASTER WEIGHT STORAGE

Can we avoid two weights copies? Can FLOAT16 be used for weight update?

“Vanilla” SGD weights update:

\[ W(t+1) = W(t) - \lambda \times \Delta W(t) \]

If we use float16 for \( \Delta W \), the product \( \lambda \times \Delta W(t) \) can become too small:

Initially gradients \( \Delta W(t) \) are very small. They are multiplied by learning rate \( \lambda \) which is \( < 1 \), so \( \lambda \times \Delta W(t) \) can go into subnormal float16 range

Later gradients becomes larger, but \( \lambda \) becomes smaller, so \( \lambda \times \Delta W(t) \) becomes even smaller.
ALEXNET: FP16 MASTER WEIGHT STORAGE

There are a number of solutions for this “vanishing update” problem. For example to keep two copies of weights: float $W_{32}$ for updates, and float16 $W_{16}$ for forward-backward pass:

Compute $\Delta W_{16}(t)$ using forward-backward pass

Convert gradients to float: $\Delta W_{32}(t) = \text{half2float}(\Delta w_{16}(t))$

Update weights in float: $W_{32}(t+1) = W_{32}(t) - \lambda \cdot \Delta W_{32}(t)$

Make float16 copy of weights: $W_{16}(t+1) = \text{float2half}(W_{32}(t+1))$

Do forward-backward with $W_{16}$ …

So $W_{32}$ will accumulate small weights updates.
ALEXNET: FP16 MASTER WEIGHT STORAGE

Consider SGD with momentum:

1. Compute momentum $H$:  
   \[ H(t+1) = m * H(t) - \lambda * \Delta W(t) \]

2. Update weights with $H$:  
   \[ W(t+1) = W(t) + H(t+1) \]

$\lambda$ is small, so $\lambda * \Delta W(t)$ can be very small and it can vanish if we compute momentum in float16. Can we fix this?

Denote $D(t) = \Delta W(t)$. Assume for simplicity that $\lambda = \text{const}$. Then

\[ H(t+1) = m * H(t) - \lambda * D(t) = m * (H(t-1) - \lambda * D(t-1)) - \lambda * D(t) = -\lambda * [D(t) + m * D(t-1) + m^2 * D(t-2) + m^k * D(t-k) + ...] \]

Moment works as average of gradients!
Let’s modify the original momentum schema:

1. Compute momentum $H$:
   \[ H(t+1) = m * H(t) - \lambda * \Delta W(t) \]

2. Update weights with $H$:
   \[ W(t+1) = W(t) + H(t+1) \]
ALEXNET: FP16 MASTER WEIGHT STORAGE

Let’s modify the original momentum schema:

1. Compute momentum $G$: $G(t+1) = m \times G(t) + \lambda \times \Delta W(t)$
2. Update weights with $G$: $W(t+1) = W(t) - \lambda \times G(t+1)$

Now $G$ will accumulate average of $\Delta W(t)$ which don’t vanish!

Weights update in float16 we use this schema:

Compute $\Delta w_{16}(t)$ using forward-backward pass

Compute momentum: $G_{16}(t+1) = m \times G_{16}(t) + \Delta w_{16}(t)$

Update in float math: $W = \text{half2float}(W_{16}(t)) - \lambda \times \text{half2float}(G_{16}(t+1))$

Convert result to float16: $W_{16}(t+1) = \text{float2half}(W)$

Do forward-backward with $W_{16}$ …
ALEXNET: FP16 MASTER WEIGHT STORAGE

With this fix we can have only one copy of weights in float16:
<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>58.62</td>
<td>81.25</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>58.12</td>
<td>80.71</td>
</tr>
<tr>
<td>FP16 training</td>
<td>54.89</td>
<td>78.12</td>
</tr>
<tr>
<td>FP16 training, loss scale = 1000</td>
<td>57.76</td>
<td>80.76</td>
</tr>
<tr>
<td>FP16 training, loss scale = 1000, FP16 master weight storage</td>
<td>58.56</td>
<td>80.89</td>
</tr>
</tbody>
</table>

Nvcaffe-0.16, DGX-1, SGD with momentum, 100 epochs, batch=1024, no augmentation, 1 crop, 1 model
## INCEPTION-V3 RESULTS

Scale loss function by 100x...

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>73.85</td>
<td>91.44</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>73.6</td>
<td>91.11</td>
</tr>
<tr>
<td>FP16 training</td>
<td>71.36</td>
<td>90.84</td>
</tr>
<tr>
<td>FP16 training, loss scale = 100</td>
<td>74.13</td>
<td>91.51</td>
</tr>
<tr>
<td>FP16 training, loss scale = 100, FP16 master weight storage</td>
<td>73.52</td>
<td>91.08</td>
</tr>
</tbody>
</table>

Nvcaffe-0.16, DGX-1, SGD with momentum, 100 epochs, batch=512, no augmentation, 1 crop, 1 model.
INCEPTION-V3 RESULTS

![Graph showing accuracy over iterations for different models: Inception_v3_baseline, Inception_v3_dfp16, Inception_v3_mfp16_x100, Inception_v3_nfp16_x100. Top 1 accuracy is plotted against iteration.](image)
RESNET RESULTS

No scale of loss function ...

<table>
<thead>
<tr>
<th>Mode</th>
<th>Top1 accuracy, %</th>
<th>Top5 accuracy, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fp32</td>
<td>71.75</td>
<td>90.52</td>
</tr>
<tr>
<td>Mixed precision training</td>
<td>71.17</td>
<td>90.10</td>
</tr>
<tr>
<td>FP16 training, loss scale = 1</td>
<td>71.17</td>
<td>90.33</td>
</tr>
<tr>
<td>FP16 training, loss scale = 1, FP16 master weight storage</td>
<td>70.53</td>
<td>90.14</td>
</tr>
</tbody>
</table>

Nvcaffe-0.16, DGX-1, SGD with momentum, 100 epochs, batch=512, no augmentation, 1 crop, 1 model
RESNET-50 RESULTS

FP16 training is ok

FP16 storage has a small dip at the end (noise?)
1. Good results on Volta mixed precision training with a variety of networks
   - Applying a global scaling to the loss input is needed for some networks
   - Wide range of loss scaling values work well

2. **FP16 training** also works for a set of convnets using the loss scaling method - still exploratory

3. **FP16 master weight storage** also worked for a set of convnets after refactoring the solver - still exploratory

4. Overall current recommendation is “**mixed precision with FP32 master weight storage**” as the most robust training recipe
Part 3

Training with mixed precision in nvcaffe-0.16
NVIDIA/CAFFE-0.16

- Full float16 support
- Mixed precision:
  - Different data types for Forward and Backward
  - Different math type
  - Solver_type (for weight update in float16)
- Automatic type conversion
- Very fast!

https://github.com/NVIDIA/caffe/tree/caffe-0.16
name: "AlexNet_fp16"

default_forward_type: FLOAT16
default_backward_type: FLOAT16
default_forward_math: FLOAT
default_backward_math: FLOAT

layer {
  forward_math: FLOAT16
  backward_math: FLOAT
  ...
}

solver_data_type: FLOAT16
enum Type {
    DOUBLE = 0,
    FLOAT = 1,
    FLOAT16 = 2,
    ...
}

class Blob {
    ...
    mutable shared_ptr<Tensor> data_tensor_
    mutable shared_ptr<Tensor> diff_tensor_
    ...
}

class Tensor {
    ... 
    Type type_
    shared_ptr<vector<shared_ptr<SyncedMemory>>> synced_arrays_
    ...
}

template<typename Dtype>
class TBlob : public Blob {
    ...
NVIDIA/CAFFE-0.16 - DATA AND MATH TYPES

default_forward_type: FLOAT16

default_backward_type: FLOAT16

template<typename Ftype, typename Btype>
class Layer : public LayerBase {
...

default_forward_math: FLOAT

forward_math_ = this->layer_param().forward_math();
...
setConvolutionDesc(forward_math_, fwd_conv_descs_[i],
    pad_h, pad_w, stride_h, stride_w);
solver_data_type: FLOAT16

```cpp
template <typename Dtype>
class SGDSolver : public Solver {
...
  vector<shared_ptr<TBlob<Dtype>>> history_, update_, temp_;
...

class Solver {
...
  shared_ptr<Net> net_;  // Adjusted the type to Net
  vector<shared_ptr<Net>> test_nets_; // Adjusted the type to Net
...```
template<typename Gtype, typename Wtype>
  __global__ void SGDRegUpdateAllAndClear(int N, Gtype* g, Wtype* w, Wtype* h,
    float momentum, float local_rate, float local_decay, bool reg_L2, bool clear_grads) {
    Wtype reg = reg_L2 ? w[i] : Wtype((Wtype(0) < w[i]) - (w[i] < Wtype(0)));
    Wtype gr = Wtype(g[i]) + reg * local_decay;
    gr = h[i] = momentum * h[i] + local_rate * gr;
    w[i] -= gr;
    g[i] = clear_grads ? Gtype(0) : Gtype(gr);
  }

template<> __global__ void SGDRegUpdateAllAndClear<__half, float>(int N, __half* g, float* w,
  float* h, float momentum, float l_rate, float l_decay, bool reg_L2, bool clear_grads) {
  __half hz; hz.x = 0;
  CUDA_KERNEL LOOP(i, N) {
    float reg = reg_L2 ? w[i] : (0.F < w[i]) - (w[i] < 0.F);
    float gr = __half2float(g[i]) + reg * l_decay;
    gr = h[i] = momentum * h[i] + l_rate * gr;
    w[i] -= gr;
    g[i] = clear_grads ? hz : float2half_clip(h[i]);
  }
}
NCCL_CHECK(ncclAllReduce(send, receive, count, nccl::nccl_type(type), ncclSum, nccl_comm_, comm_stream_->get()));

- 1 call does it all
- FLOAT16 takes 2x less time
- Parallelize!
- After each layer or in the end of back propagation?
NVIDIA/CAFFE-0.16 - BUCKETS

- 6-10 buckets per pass
- Weights Update + Reduce - one invocation per bucket
- Runs in a separate CUDA stream and gets synced in the end of back propagation pass
## NVIDIA/CAFFE-0.16 - Parallel Data Reader

<table>
<thead>
<tr>
<th>Batch 0</th>
<th>Batch 1</th>
<th>Batch 2</th>
<th>Batch 3</th>
<th>Batch 4</th>
<th>Batch 5</th>
<th>Batch 6</th>
<th>Batch 7</th>
<th>TR</th>
<th>out queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0.P0.q0</td>
<td></td>
<td></td>
<td>S0.P0.q3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S0.TR0.q0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S0.TR0.q1</td>
</tr>
<tr>
<td>S0.P1.q1</td>
<td></td>
<td></td>
<td>S0.P1.q4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S0.TR0.q2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S0.TR1.q3</td>
</tr>
<tr>
<td>S0.P2.q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S0.TR1.q4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S0.TR1.q5</td>
</tr>
<tr>
<td>S1.P0.q0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S1.TR0.q0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S1.TR0.q1</td>
</tr>
<tr>
<td>S1.P1.q1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S1.TR0.q2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S1.TR1.q3</td>
</tr>
<tr>
<td>S1.P2.q2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S1.TR1.q4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>→</td>
<td>S1.TR1.q5</td>
</tr>
</tbody>
</table>

2 solvers, 3 parser threads per solver (P0, P1, P2), 2 transformer threads per solver (TR0, TR1) - each transformer owns queue set with the number of queues equal to the number of parser threads. 2x3x2=12 queues total. 2x3=6 DB cursors.
NVIDIA/CAFFE-0.16 - ALL TOGETHER NOW