MIXED PRECISION TRAINING ON VOLTA GPUS

Boris Ginsburg, Paulius Micikevicius, Oleksii Kuchaiev, Sergei Nikolaev
bginsburg, pauliusm, okuchaiev, snikolaev@nvidia.com

10/17/2017

ACKNOWLEDGMENTS

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

cuDNN team

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. Training without “master-copy” weights
3. NVCAFFE support for FLOAT16

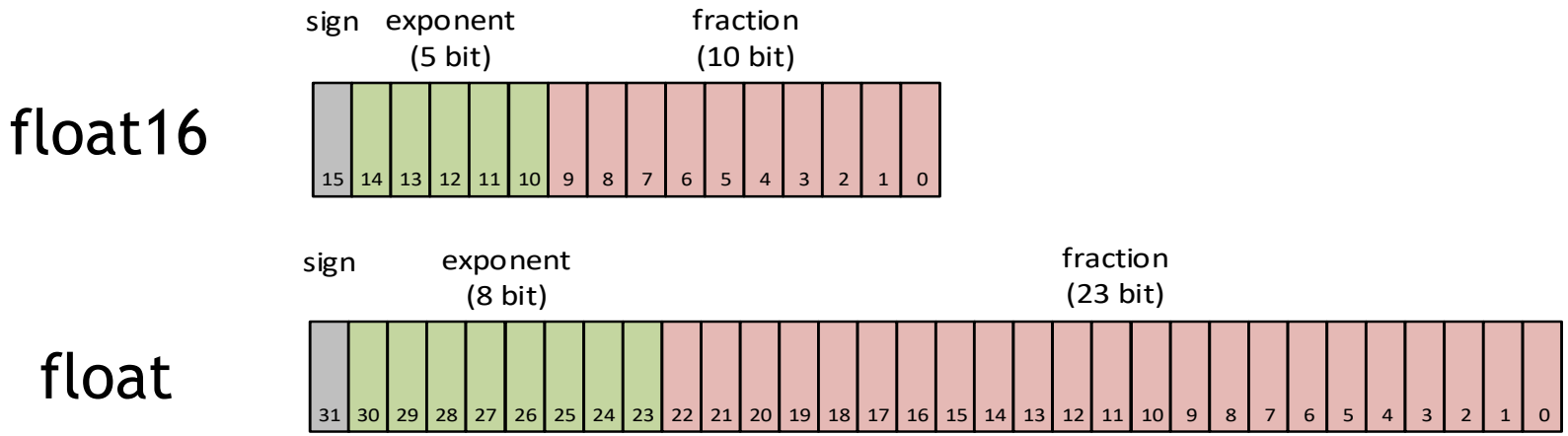
TERMINOLOGY

Training type	Data type	Matrix-Multiply Accumulator	Weight update	GPU
FP32	FP32	FP32	FP32	
“Pascal” FP16	FP16	FP16	FP16/FP32	Pascal(GP-100)
Mixed precision	FP16	FP32	FP16/FP32	Volta

“Master” weights – copy of weights used for update (SGD step).

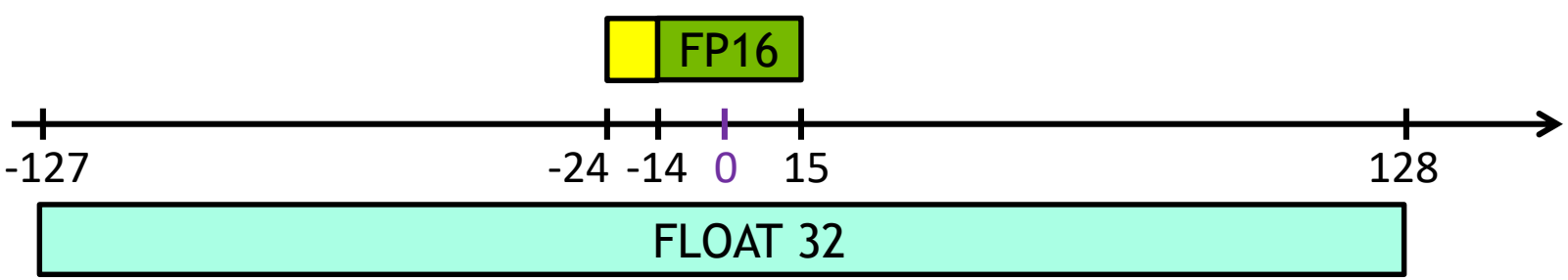
Volta: Mixed precision training with FP32 master weight storage.

HALF-PRECISION FLOAT (FLOAT16)



FLOAT16 has wide range (2^{40}) … but it is not as wide as FP32!

Normal range: [6×10^{-5} , 65504]
Sub-normal range: [6×10^{-8} , 6×10^{-5}]



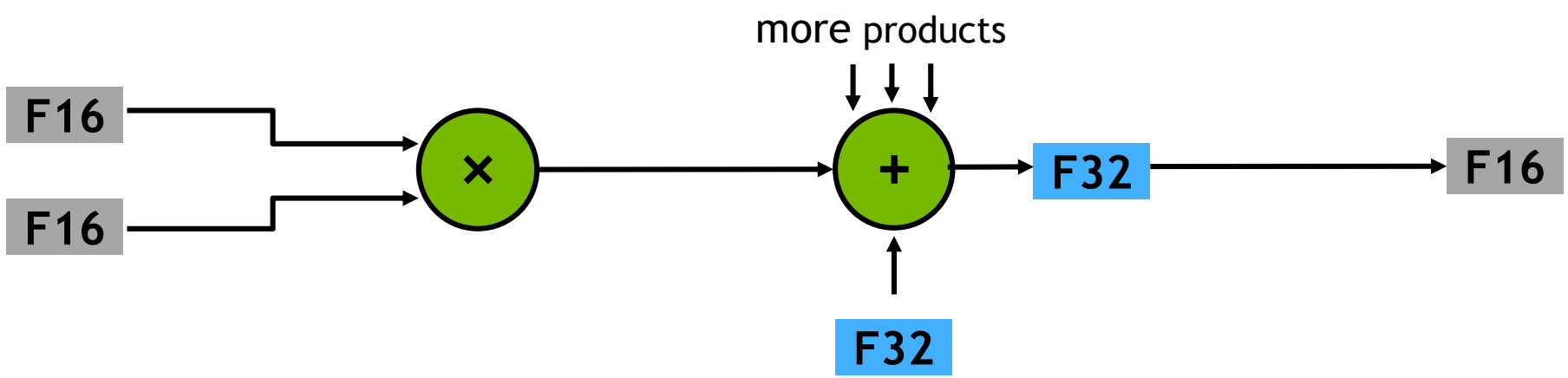
VOLTA:
TENSOR CORE 4X4 MATRIX-MULTIPLY ACC

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

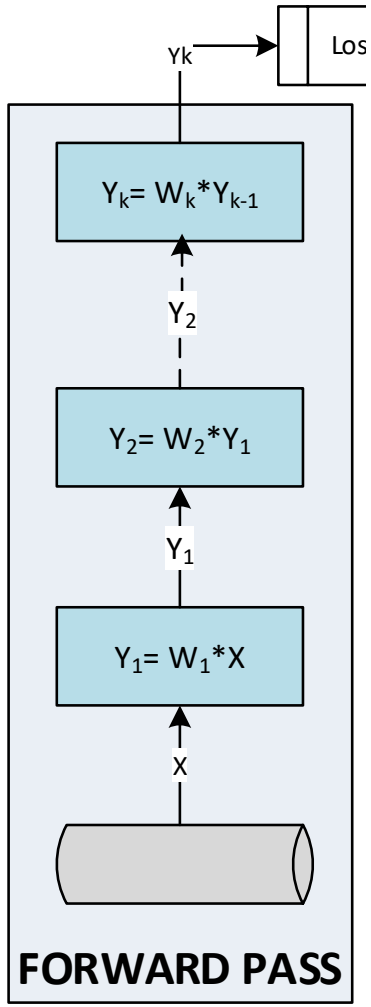
VOLTA TENSOR OPERATION

FP16 storage/input	Full precision product	Sum with FP32 accumulator	Convert to FP32 result

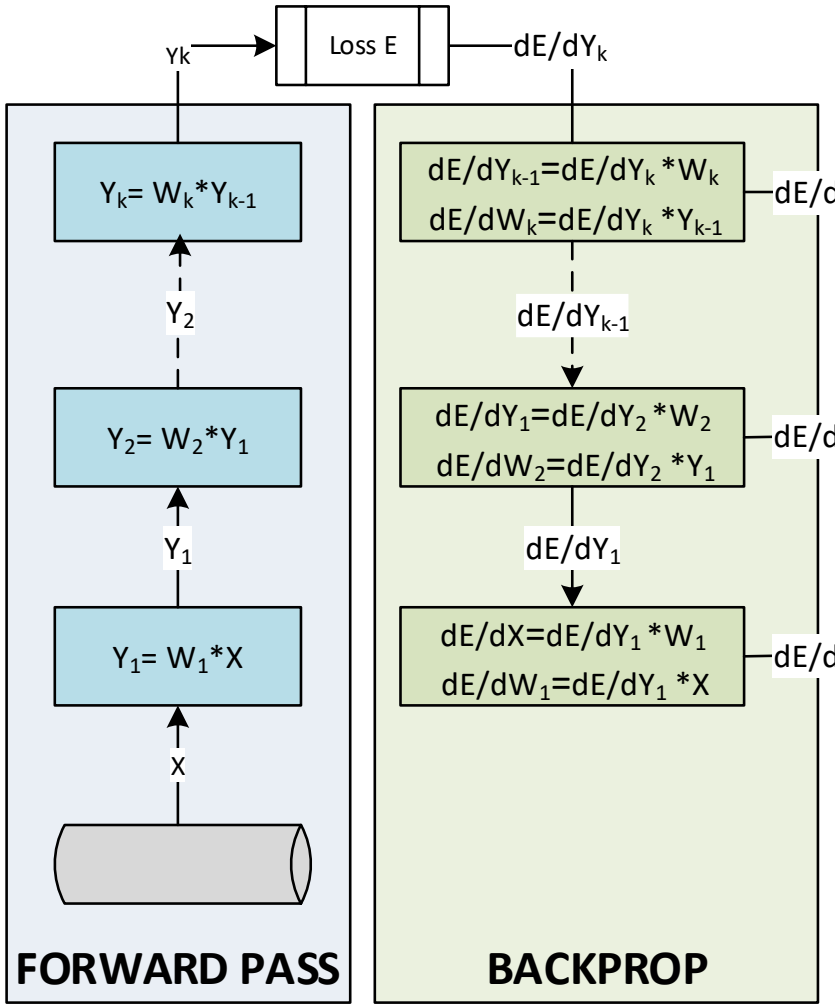


Also supports FP16 accumulator mode for inferencing

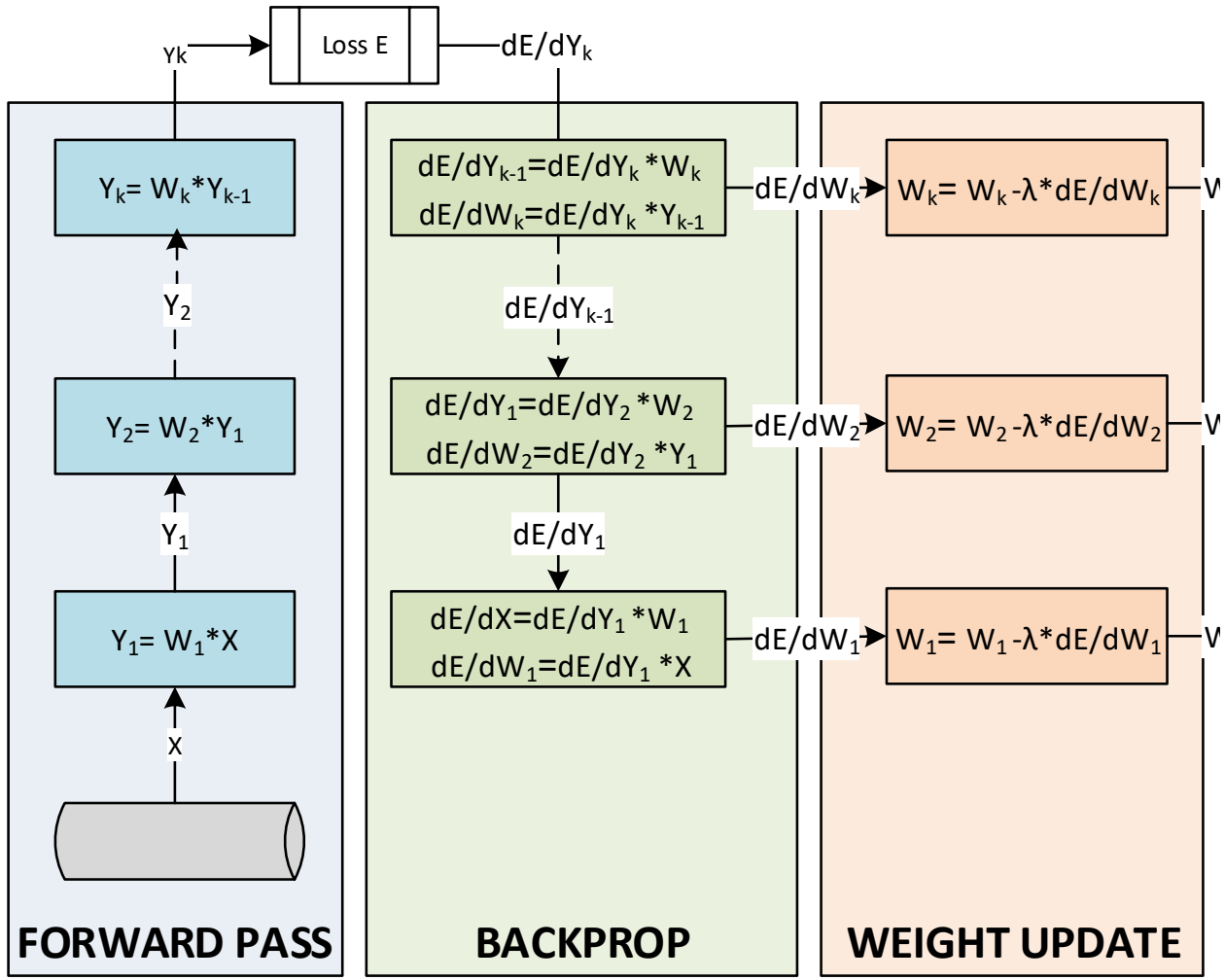
TRAINING FLOW



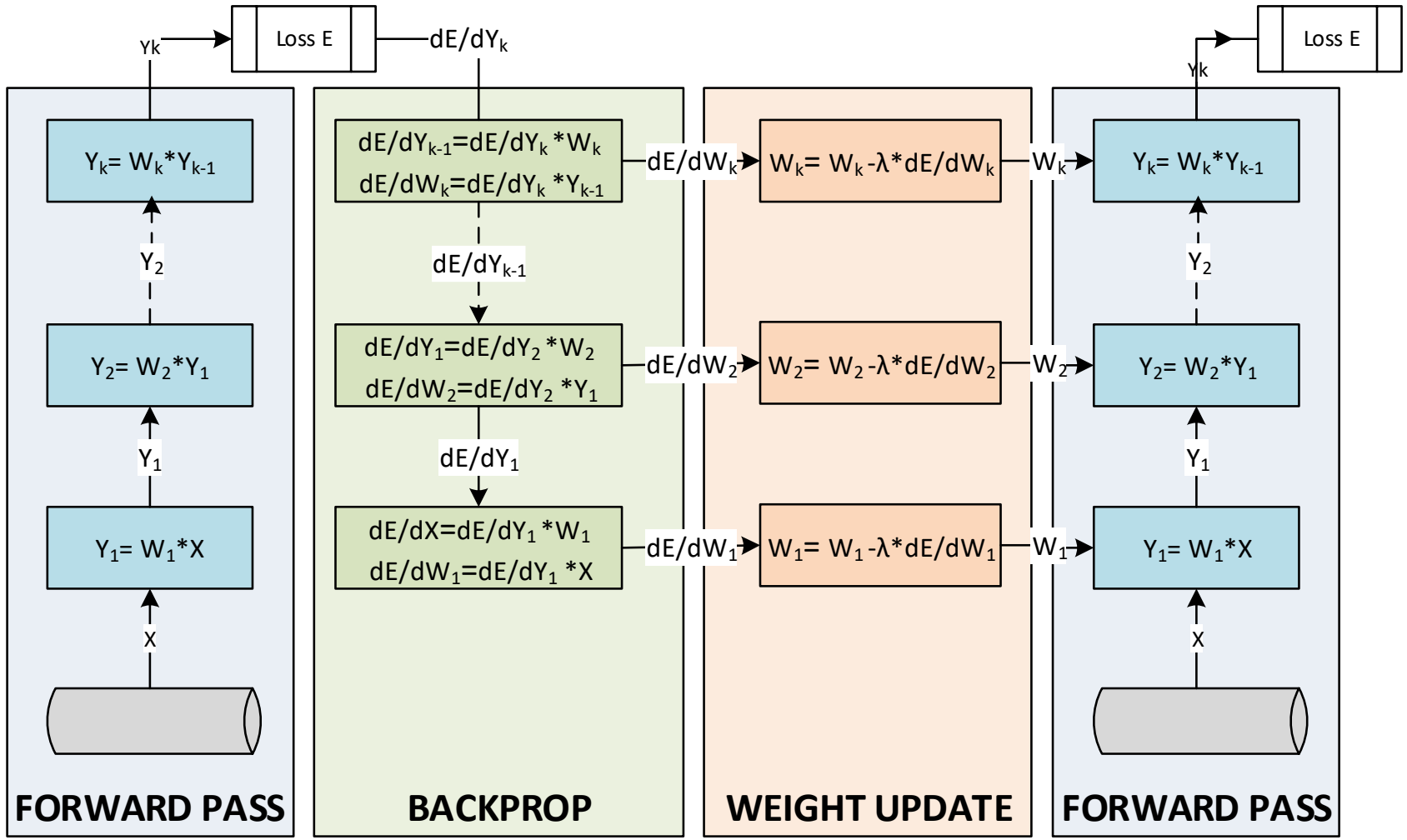
TRAINING FLOW



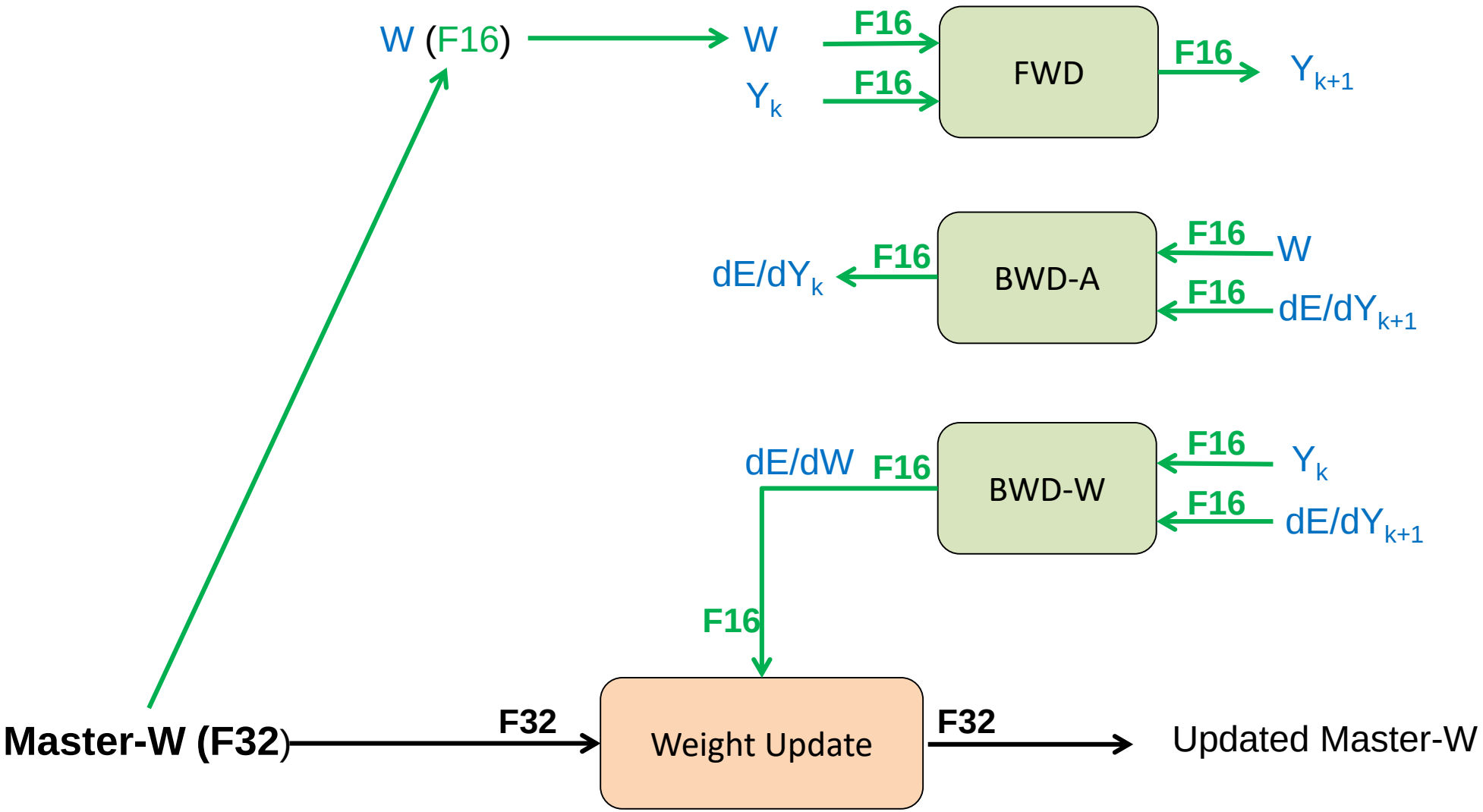
TRAINING FLOW



TRAINING FLOW



VOLTA TRAINING METHOD



SOME NETS WORKED OUT OF THE BOX

TensorOp training matched the results of F32 training

- Same solver, hyper-parameters, and training schedule as F32

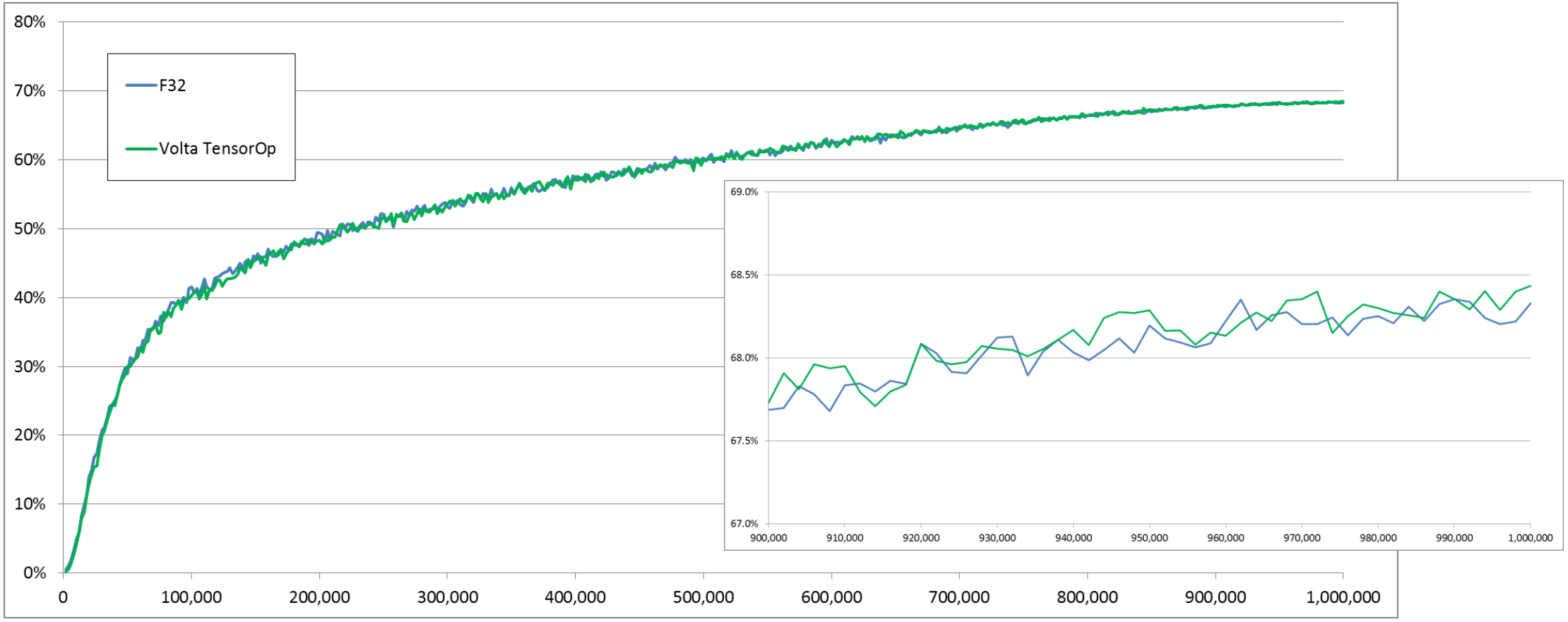
Image classification (ILSVRC12):

- GoogleNet, VGG-D, Inception v3, ResNet-50
- Solver: SGD with momentum

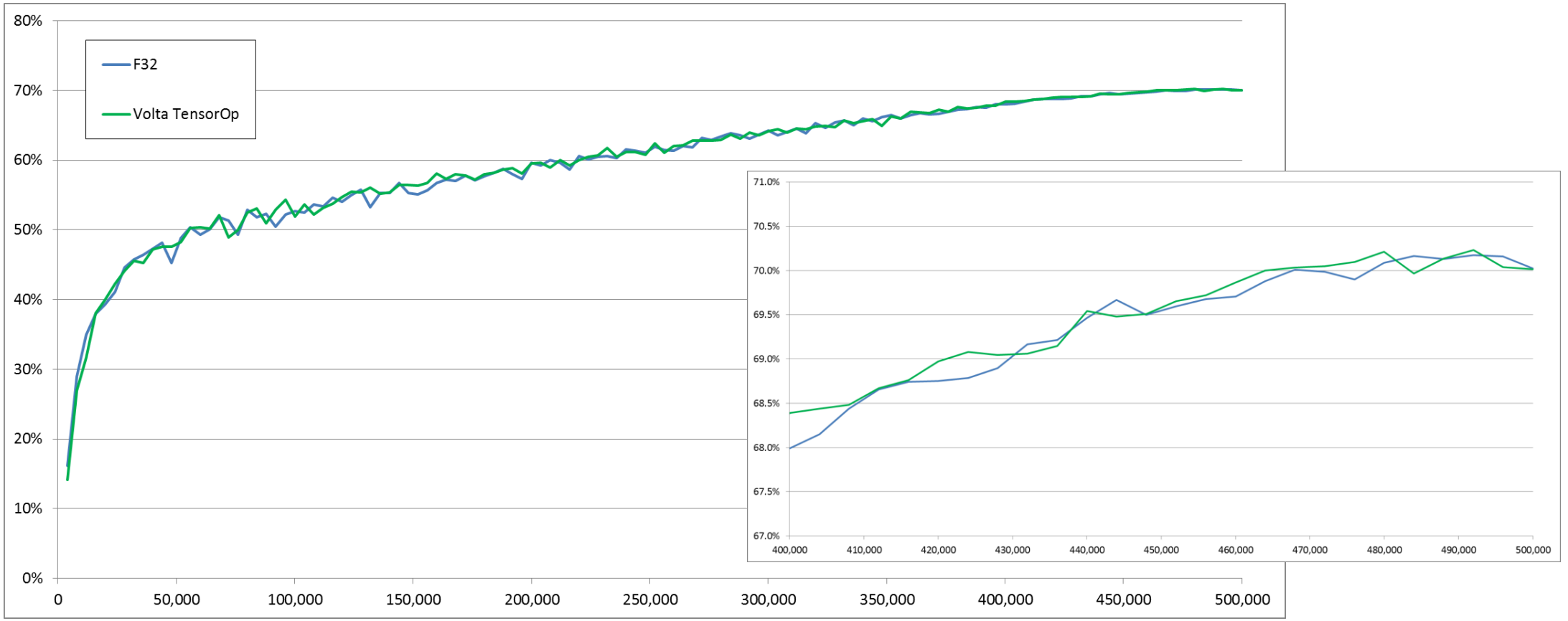
Language modeling and machine translation

- NMT
- Solver: ADAM

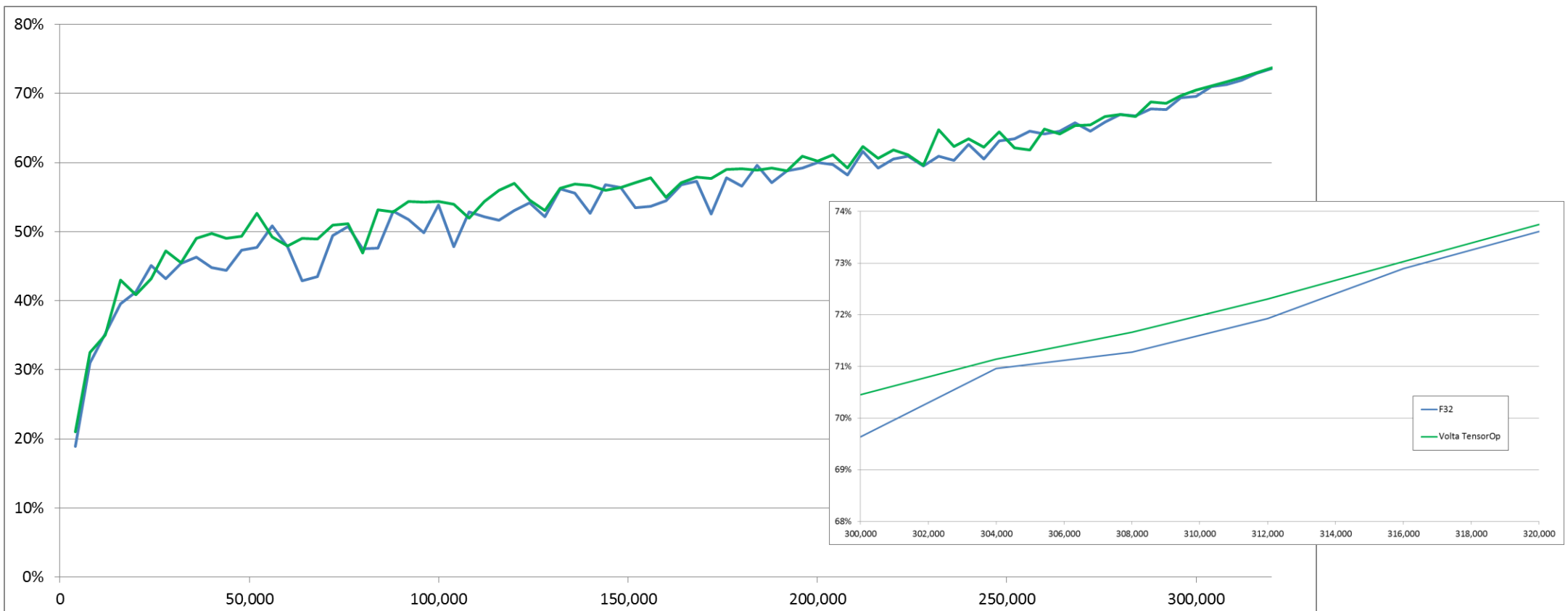
GOOGLENET


80%
70%
60%
50%
40%
30%
20%
10%
0%
F32
Volta TensorOp
0
100,000
200,000
300,000
400,000
500,000
600,000
700,000
800,000
900,000
1,000,000
69.0%
68.5%
68.0%
67.5%
67.0%
900,000
910,000
920,000
930,000
940,000
950,000
960,000
970,000
980,000
990,000
1,000,000

INCEPTION V1



RESNET-50


80%
70%
60%
50%
40%
30%
20%
10%
0%
0
50,000
100,000
150,000
200,000
250,000
300,000
74%
73%
72%
71%
70%
69%
68%
300,000
302,000
304,000
306,000
308,000
310,000
312,000
314,000
316,000
318,000
320,000
F32
Volta TensorOp

MIXED PRECISION TRAINING

Networks that don't work "out-of-box" :

Image classification: Alexnet, CaffeNet

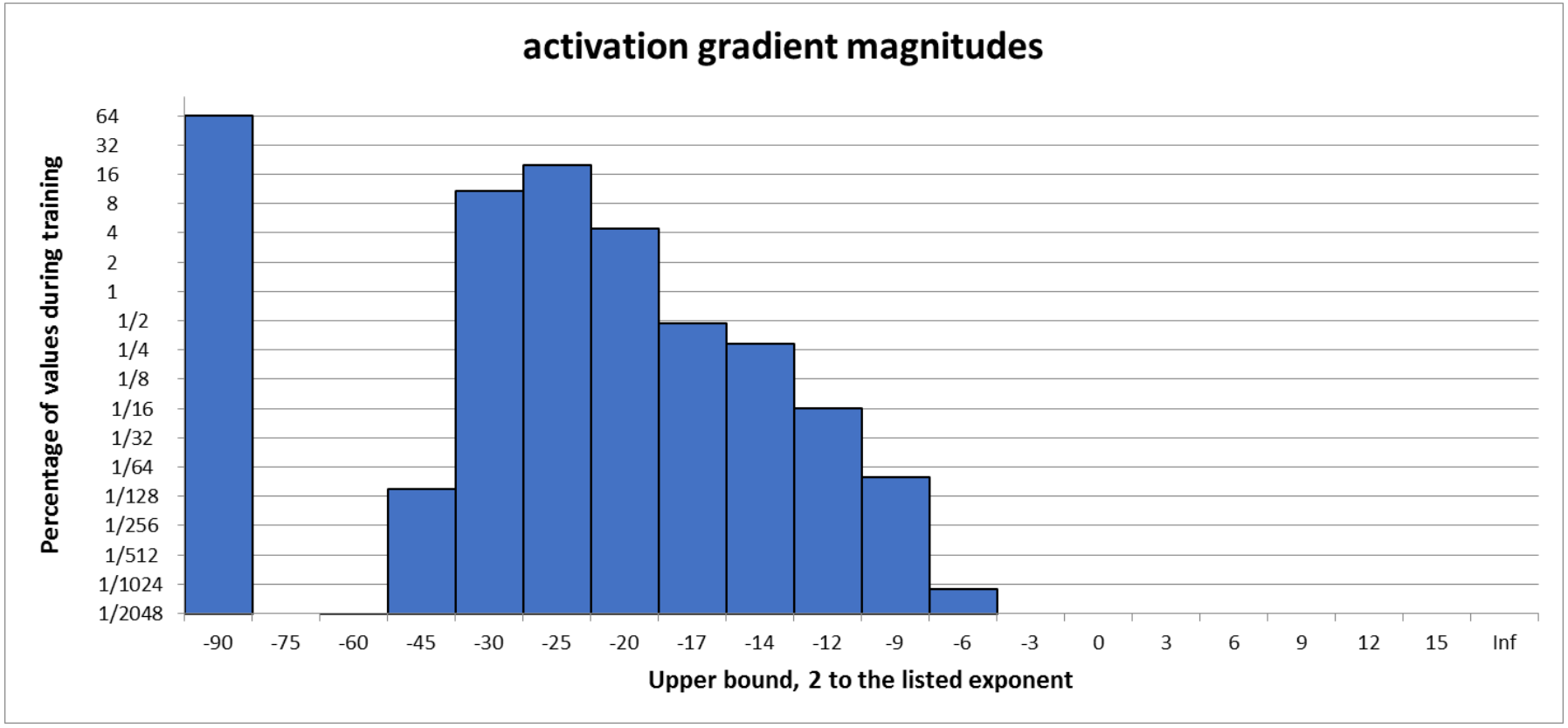
Detection nets:

- Multibox SSD with VGG-D backbone: was not learning
- Faster R-CNN with VGG-D backbone: low mAP 68.5% vs 69.1% with F32

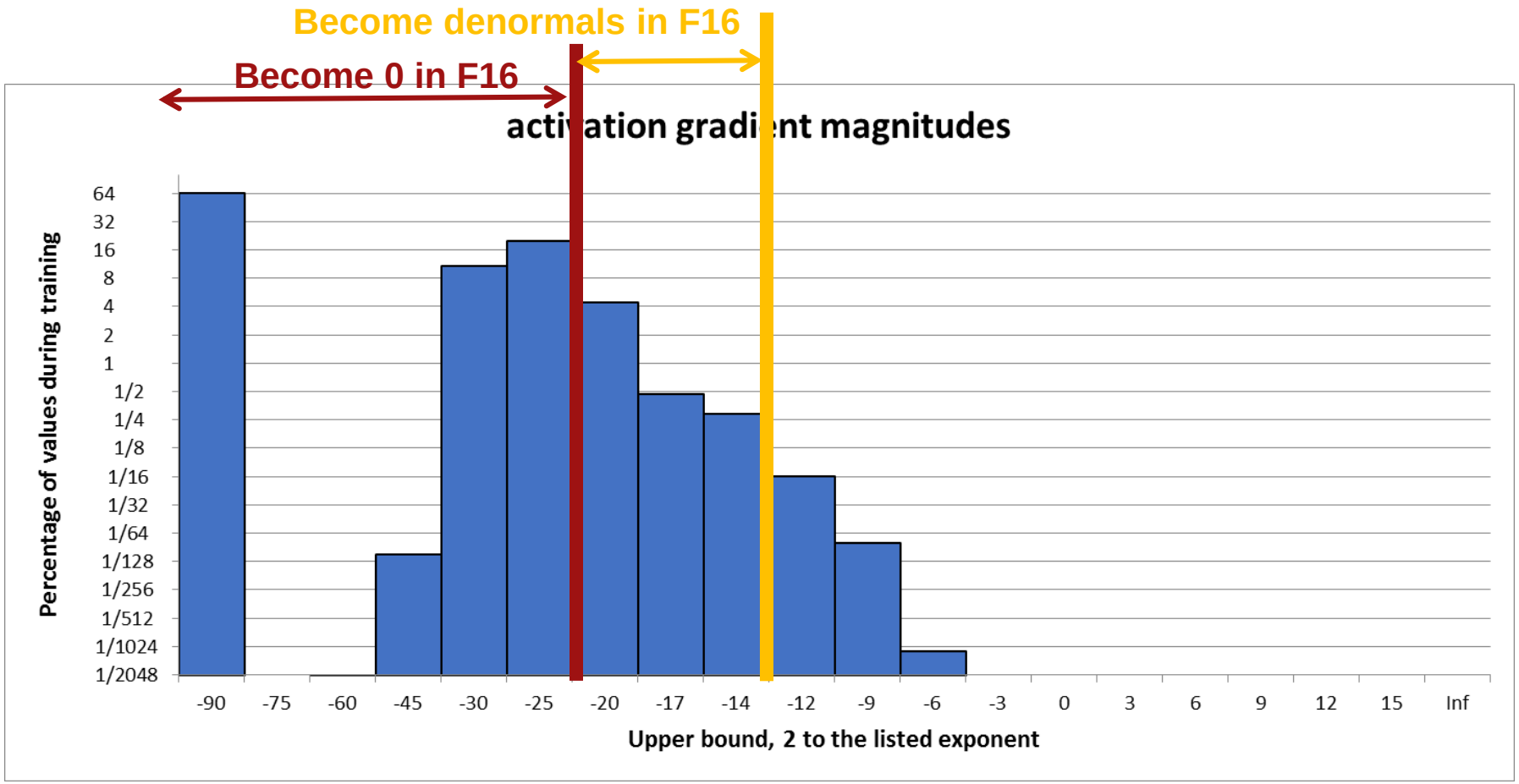
RNNs:

- Seq2seq with attention: lagged behind F32 in perplexity
- bigLSTM: diverged after some training

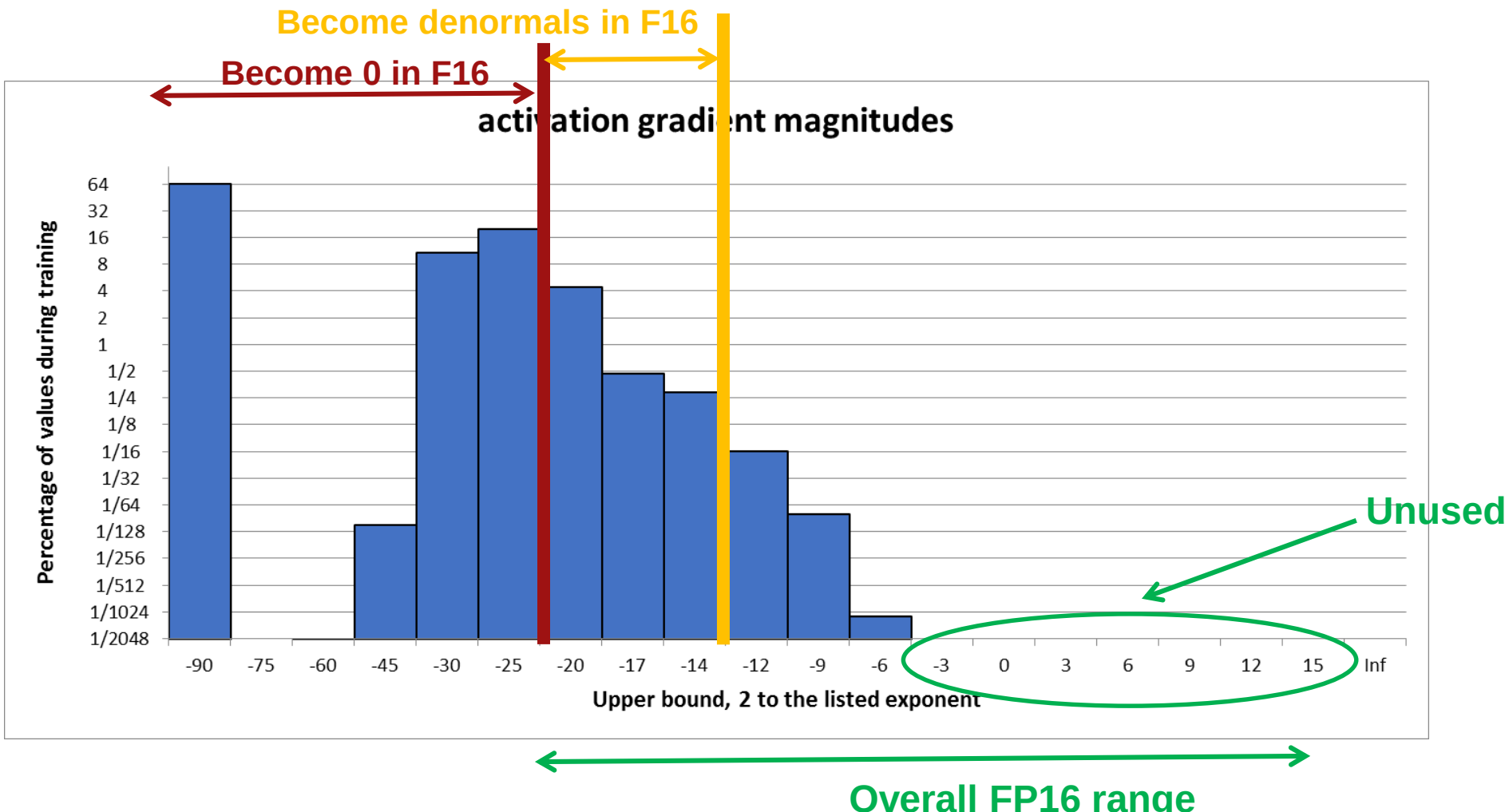
MULTI-BOX SSD: ACTIVATION GRADIENT HISTOGRAM



MULTI-BOX SSD: ACTIVATION GRADIENT HISTOGRAM



MULTIBOX SSD: ACTIVATION GRADIENT HISTOGRAM



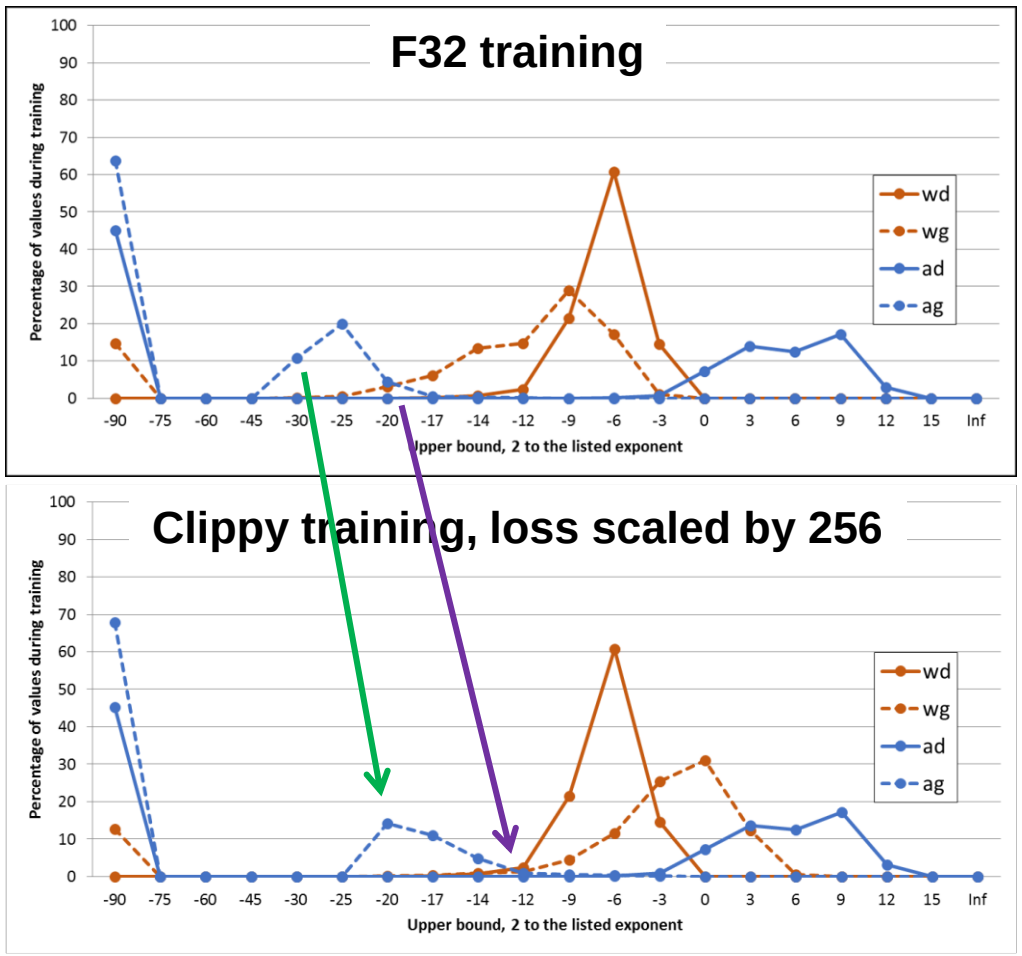
MIXED PRECISION TRAINING: GLOBAL LOSS SCALING

The problem is small ("vanishing") activation gradients

Solution:

1. scale the loss value by K to "shift" gradients to FP16 range
2. rescale gradients back by 1/K before weight update
3. New hyper-parameter: loss scale

SCALING LOSS AND GRADIENTS



Loss scaled by 256

=> gradients get scaled by 256

Benefits:

shift activation gradients into working range

Most weight gradients become normalized values in F16

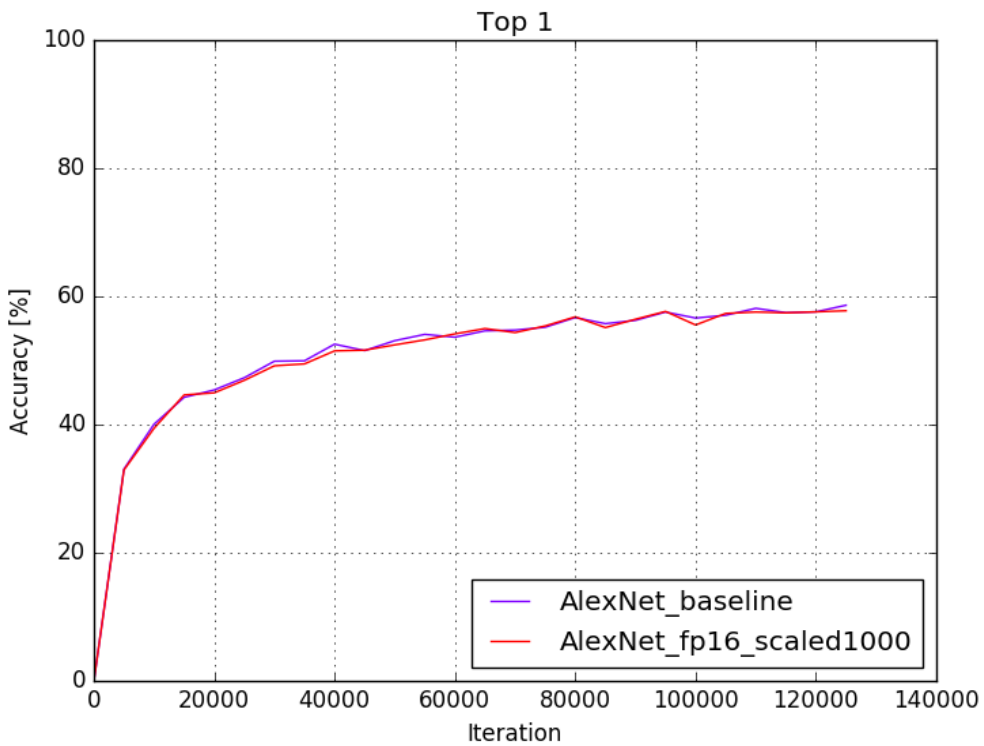
ALEXNET : COMPARISON OF RESULTS

Mode	Top1, %	Top5, %
Fp32	58.62	81.25
FP16 training, no scaling	56.70	78.12
FP16 training, scale = 1000	58.9	81.10
Mixed precision training, scale =1000	59.1	81.2

Nvcaffe-0.16, DGX1, SGD with momentum, 100 epochs, batch=1024, no augmentation, 1 crop, 1 model

ALEXNET: TRAINING WITH SCALING

FP16 training matches other training curves (TensorOp and FP32)



OBJECTS DETECTION

Multibox-SSD mAP:

F32: 76.9%

TensorOp: 77.1%, loss scaled by 256;

Without scaling: doesn't learn

Faster-RCNN mAP:

F32: 69.1%

TensorOp: 69.7%, loss scaled by 256,

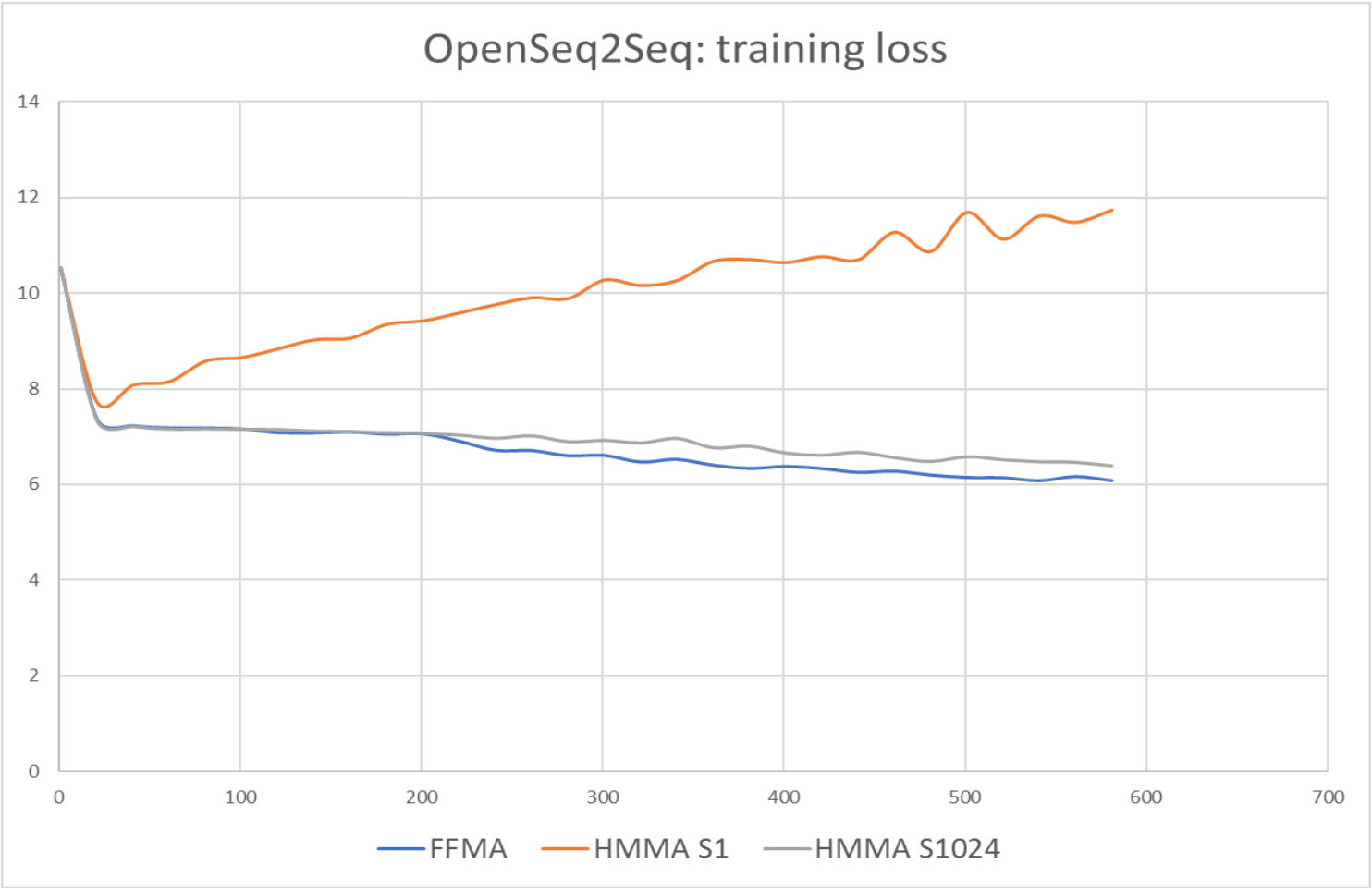
without scaling: 68.5%

SEQ2SEQ TRAINING

Neural machine translation: Ge-En

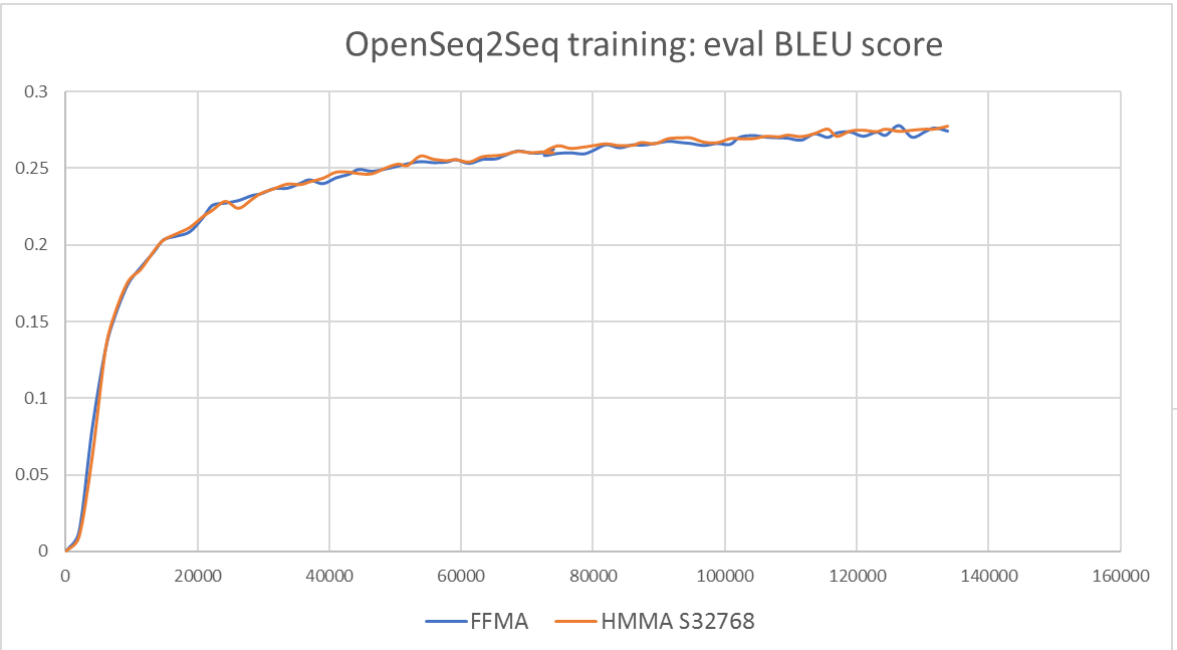
1. https://github.com/NVIDIA/OpenSeq2Seq
2. NMT_ONE model:
    1. 2-layer bi-directional encoder (512 LSTM)
    2. Normalized Bahdanau attention
    3. 4-layer decoder (512 LSTM)

SEQ2SEQ TRAINING

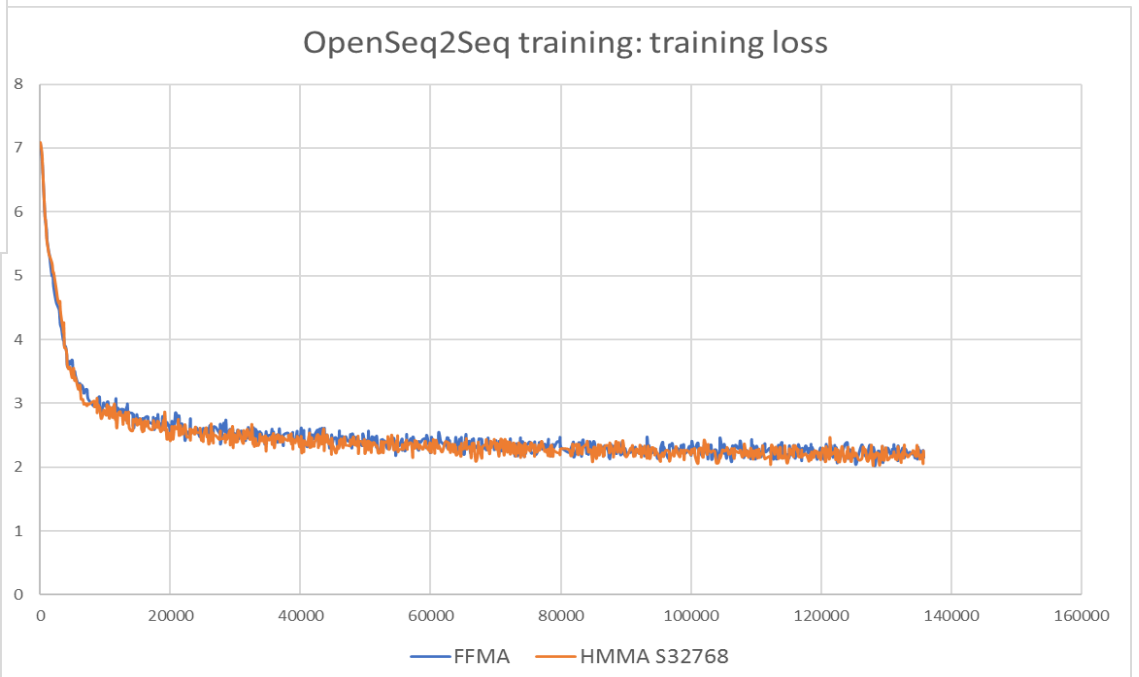


Scaling recovered acuracy

SEQ2SEQ TRAINING



NMT_ONE



scaling factor: 32,768

How to reduce scaling factor?

OPEN SEQ2SEQ LOSS

Logits shape: [batch_size, time, dimension]

Targets shape: [batch_size, time]

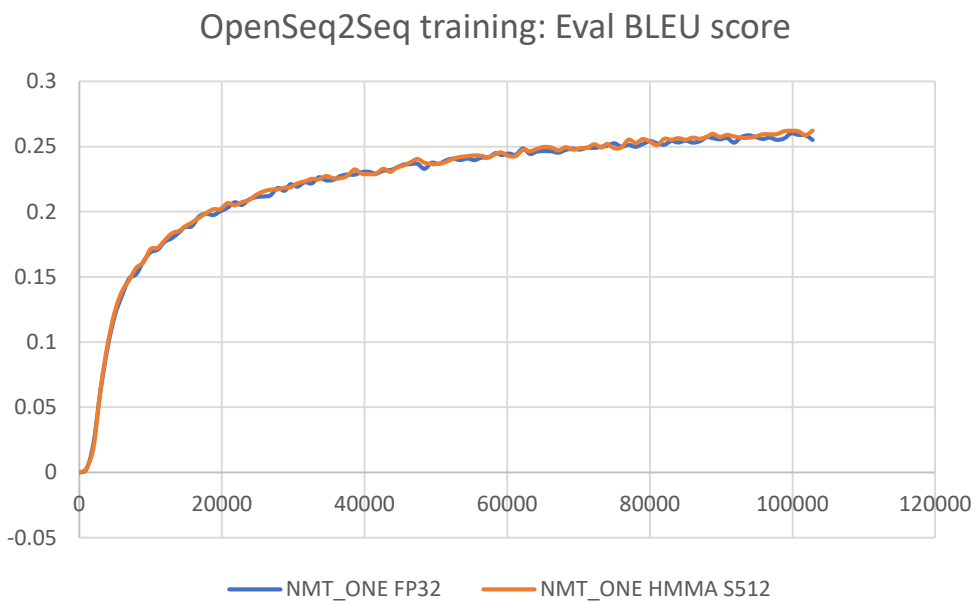
$$Loss_{avg} = Avg_{batch}(\boldsymbol{Avg_{timesteps}}(crossentropy(logits, targets)))$$

$$Loss_{sum} = Avg_{batch}(\boldsymbol{SUM_{timesteps}}\ (crossentropy(logits, targets)))$$

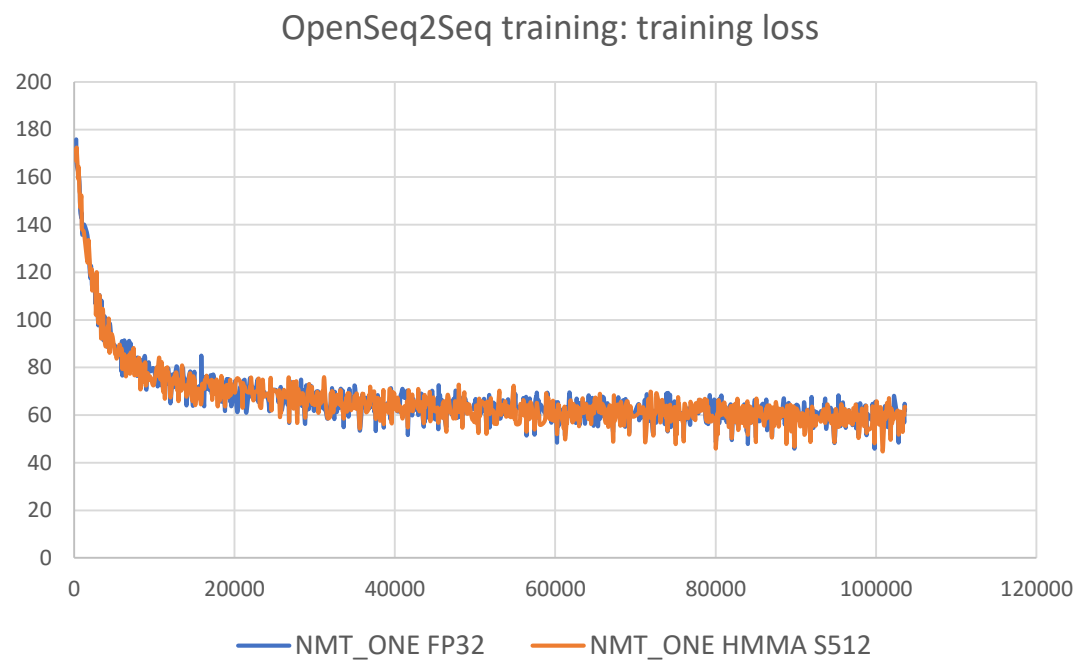
More numerically-friendly loss function

LARS algorithm allows smaller scale factors

SEQ2SEQ TRAINING

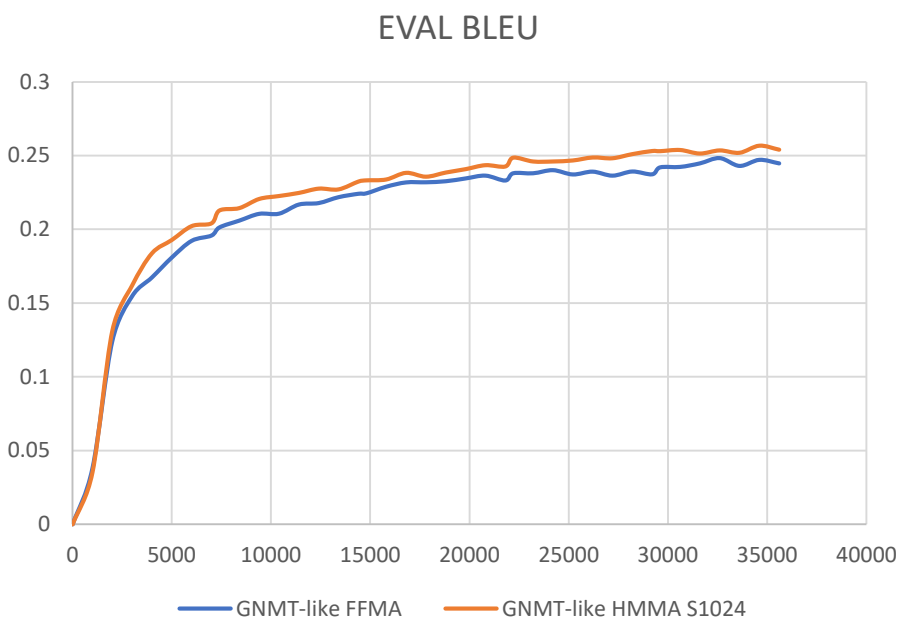


NMT_ONE



With sum loss and LARS scaling factor: 512

SEQ2SEQ TRAINING



GNMT-like

- 8-layer bi-directional encoder (1024 LSTM)
- GNMT-style normalized Bahdanau attention
- 8-layer decoder (1024 LSTM)
- residual connections

With sum loss and LARS scaling factor: 1024

[1] Wu, Yonghui, et al. "Google's neural machine translation system: Bridging the gap between human and machine translation." *arXiv preprint arXiv:1609.08144* (2016).

LANGUAGE MODEL

1 Billion Word Language Benchmark

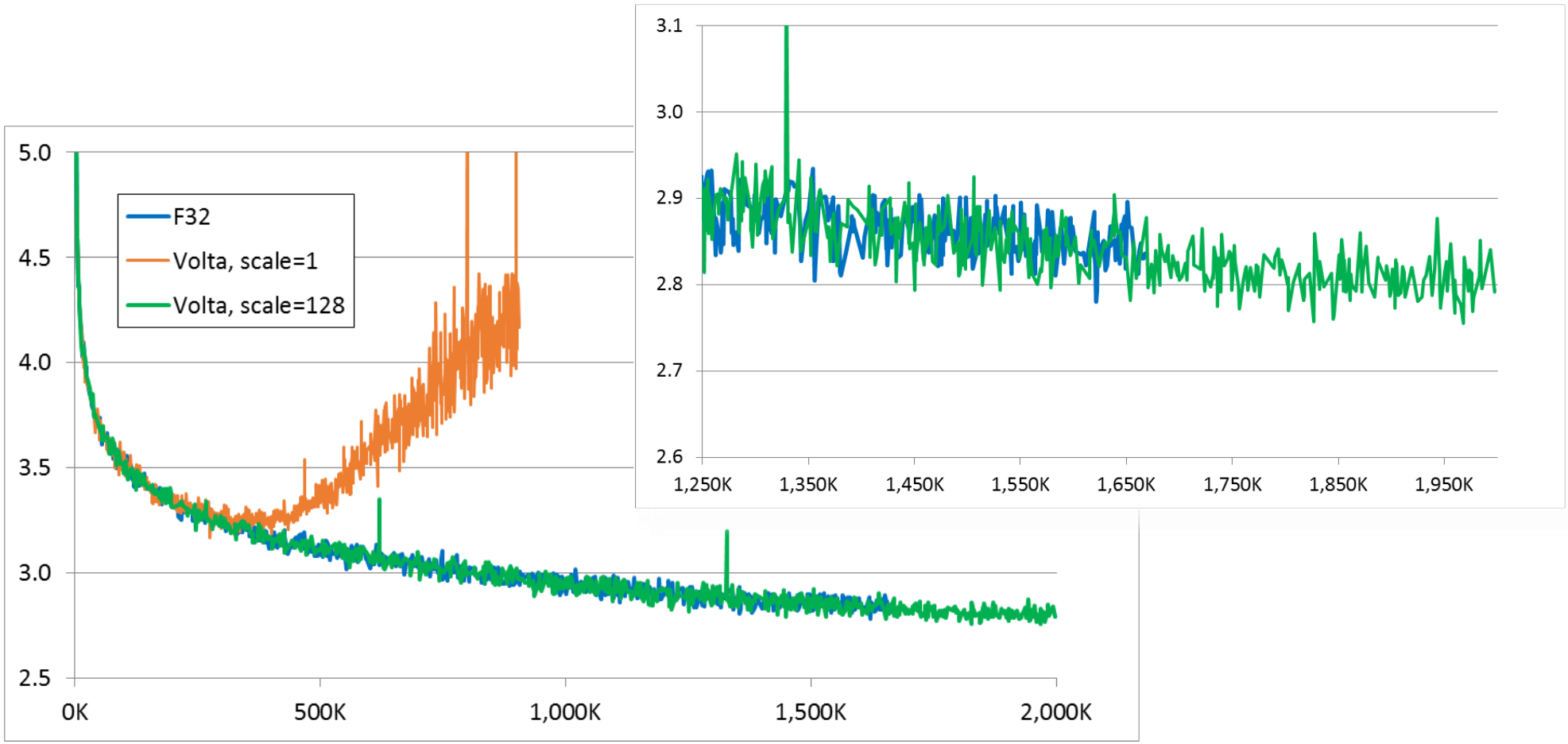
BigLSTM:

- 2x8192 LSTM, 1024 Projection
- 800K word vocabulary

Adagrad solver

Based on “Exploring the Limits of Language Modeling”, https://arxiv.org/abs/1602.02410

BIGLSTM: 2X8192 LSTM, 1024 PROJECTION



GUIDELINES FOR TRAINING WITH MIXED PRECISION / TENSOROPS

VOLTA MIXED PRECISION TRAINING

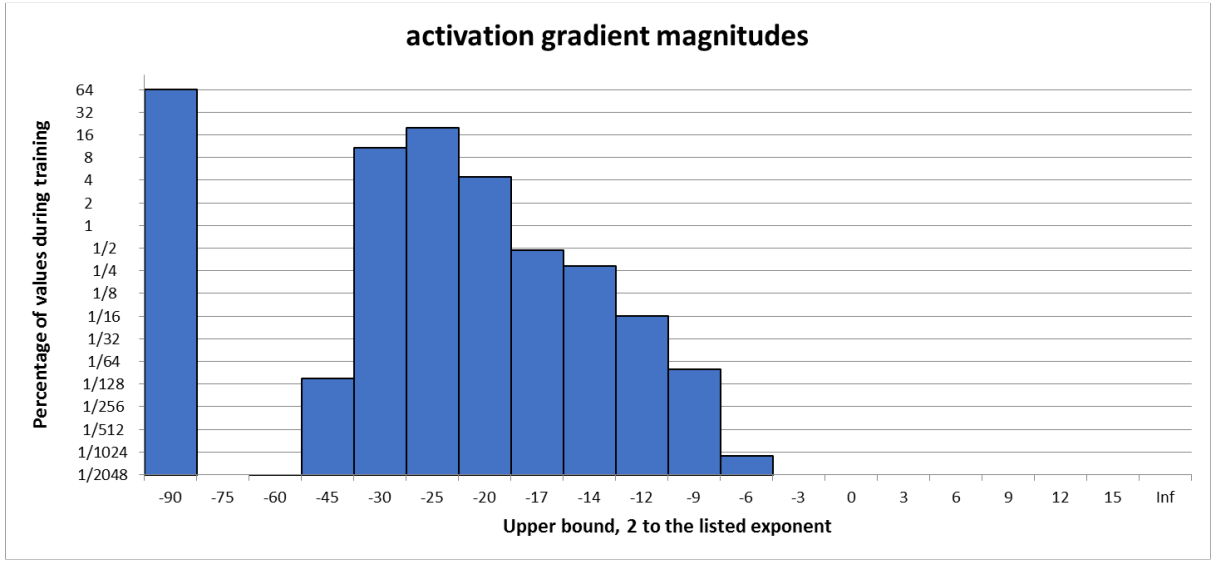
Mixed Precision Training:

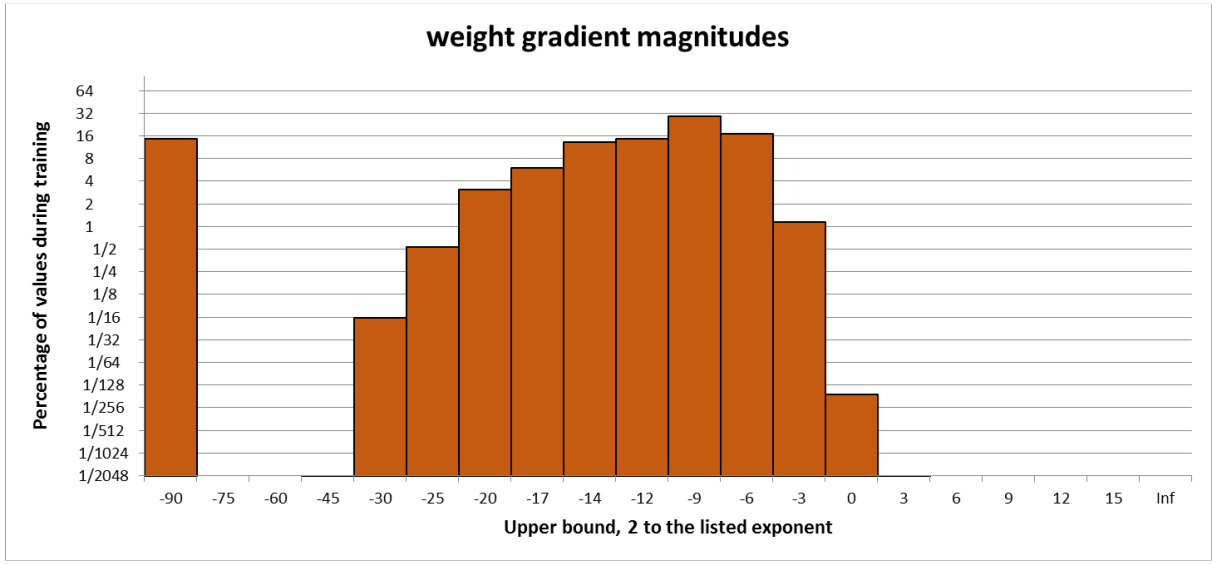
- FP16 storage and TensorOps for fwd/bwd pass:
  - weights, activations, gradients
- Batch Normalization: FP16 data, FP32 math
- FP32 “master-copy” of weights for weights update

WARNING

- Gradient may become too small for FP16 range
- Solved with new hyper parameter “loss scaling”

OBSERVATIONS ON GRADIENT VALUES





FP16 range: 2^{40} including denorms

Gradient range is biased low vs standard FP16 range

- Max magnitude we've seen was $O(2^3)$
- "shift" values without overflowing

Maximum magnitudes:

- weight-grad >> activation-grad
- * for all the nets we've looked at

MIXED PRECISION TRAINING WITHOUT MASTER WEIGHT COPY

ALEXNET: FP16 MASTER WEIGHT STORAGE

Can we avoid two weights copies?
Can FP16 be used for weight update?

"Vanilla" SGD weights update:

$$W(t+1) = W(t) - \lambda * \Delta W(t)$$

If we use FP16 for ΔW, the product $\lambda * \Delta W(t)$ can become too small:

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

Later gradients becomes larger, but λ becomes smaller, so $\lambda * \Delta W(t)$ becomes even smaller.

ALEXNET: FP16 MASTER WEIGHT STORAGE

There are a number of solutions for this “vanishing update” problem.

Option1 “ FP32 master copy of weights”

- float W_{32} “master” copy for updates,
- fp16 W_{16} weights used 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: $W_{32}(t+1) = W_{32}(t) - \lambda * \Delta W_{32}(t)$

  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: $W(t+1) = W(t) + H(t+1)$

λ 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 λ = const. Then

$$H(t+1) = m*H(t) - \lambda*D(t) = m*(H(t-1) - \lambda*D(t-1)) - \lambda*D(t) =$$

$$-\lambda*\boxed{[D(t) + m*D(t-1) + m^2*D(t-2) + m^k*D(t-k) + \ldots]}$$

Moment works as average of gradients!

ALEXNET: FP16 MASTER WEIGHT STORAGE

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: $G(t+1) = m*G(t) + \cancel{-\lambda}\Delta W(t)$
2. Update weights with G: $W(t+1) = W(t) - \lambda*G(t+1)$

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

Weights update in float16:

Compute Δw16(t) using forward-backward pass

Compute momentum: G16(t+1) = m* G16(t) + Δw16(t)

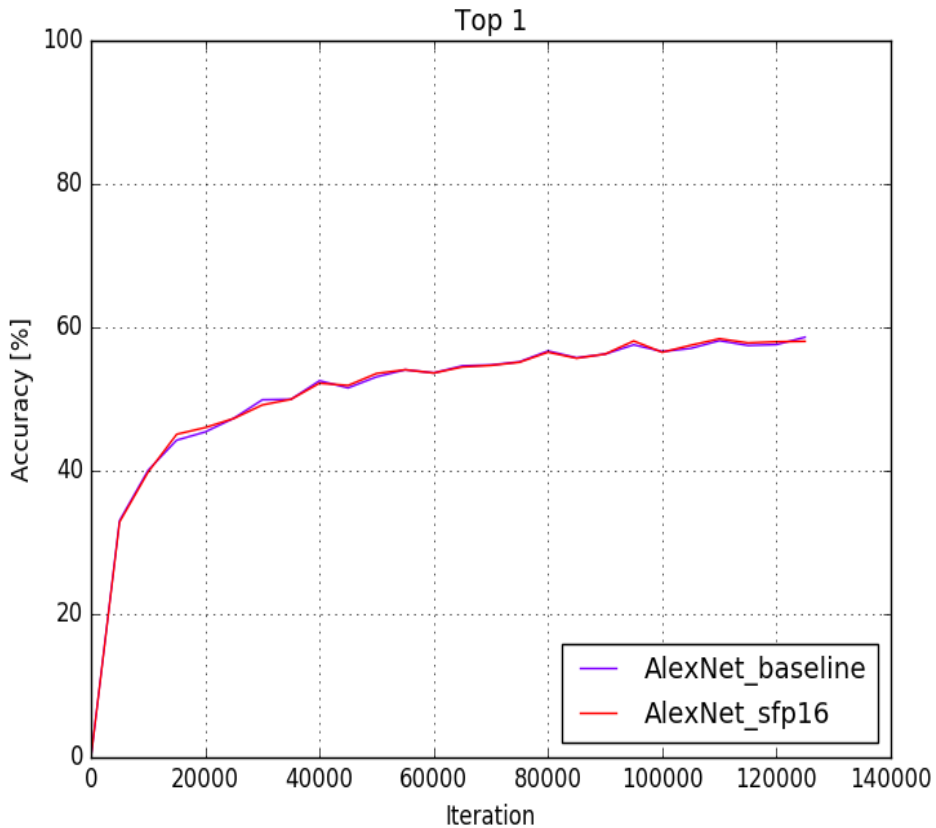
Update in float math: W=half2float(W16(t))- λ*half2float(G16(t+1))

Convert result to float16: W16(t+1)=float2half(W)

Do forward-backward with W16 ...

ALEXNET: FP16 MASTER WEIGHT STORAGE

With this fix we can have only one copy of weights in float16:



ALEXNET : COMPARISON OF RESULTS

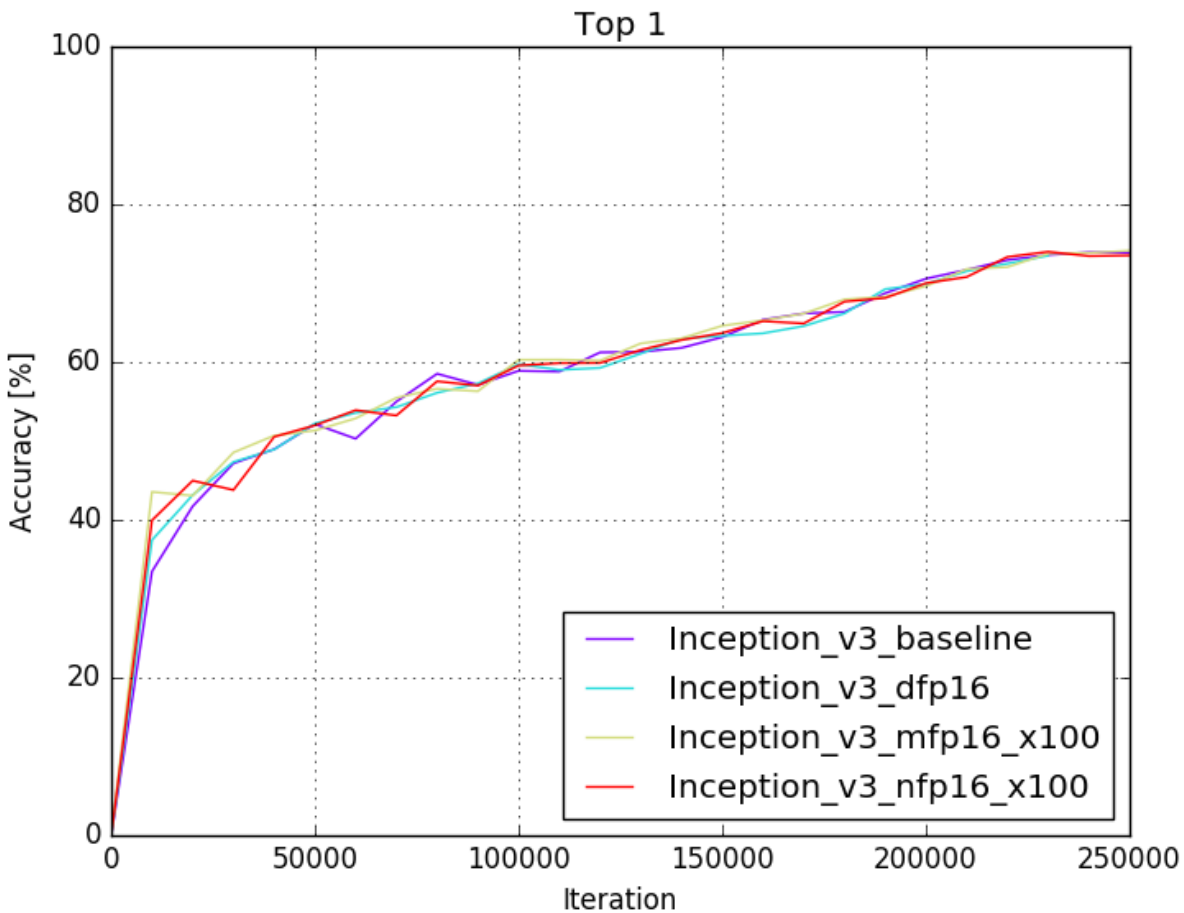
Mode	Top1, %	Top5, %
Fp32	58.62	81.25
FP16 training, no scaling	56.70	78.12
FP16 training, scale = 1000	58.9	81.10
Mixed precision training, scale =1000	59.1	81.2
Mixed precision training, scale =1000 FP16 master weights	58.50	81.2

INCEPTION-V3 RESULTS

Scale loss function by 100x…

Mode	Top1, %	Top5, %
Fp32	73.85	91.44
Mixed precision training	73.6	91.11
FP16 training	71.36	90.84
FP16 training, loss scale = 100	74.13	91.51
FP16 training, loss scale = 100, FP16 master weights	73.52	91.08

INCEPTION-V3 RESULTS



RESNET-50 RESULTS

No scale

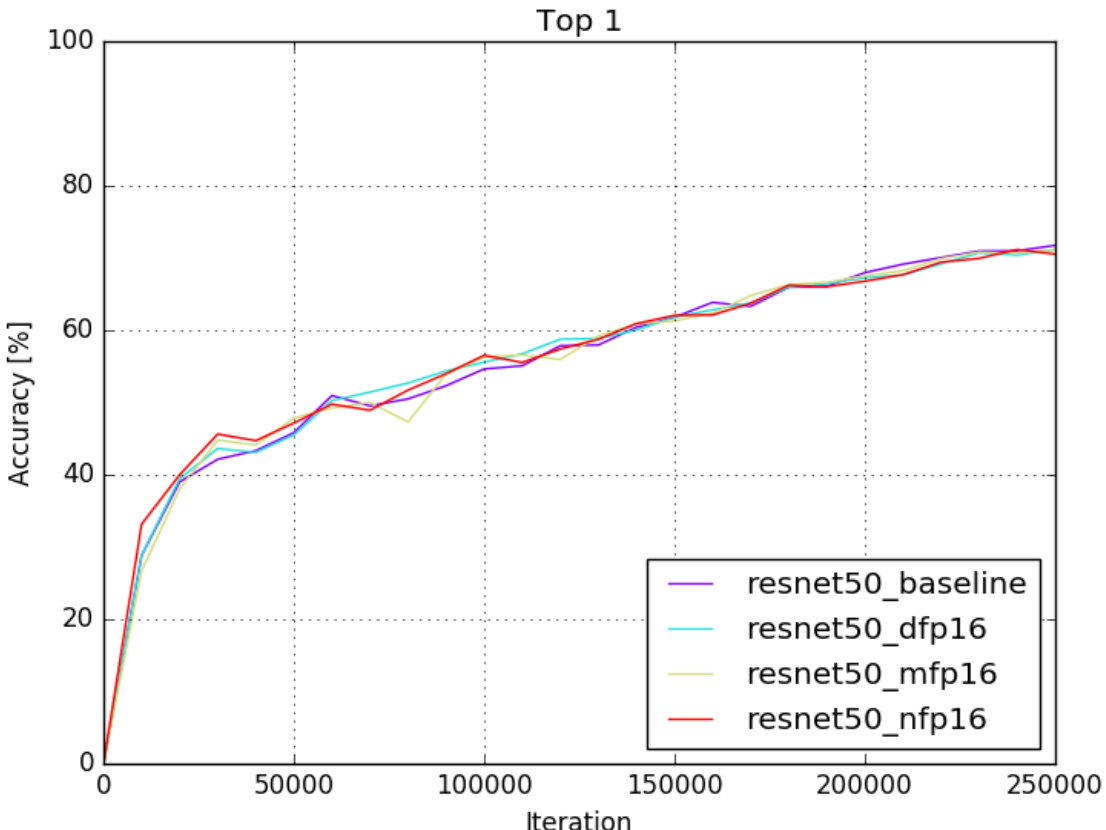
Mode	Top1, %	Top5, %
Fp32	73.2	91.2
FP16 training, FP32 master weights	73.2	90.9
FP16 training, FP16 master weight	72.7	91.5
Mixed, FP16 master weight	73.5	91.4

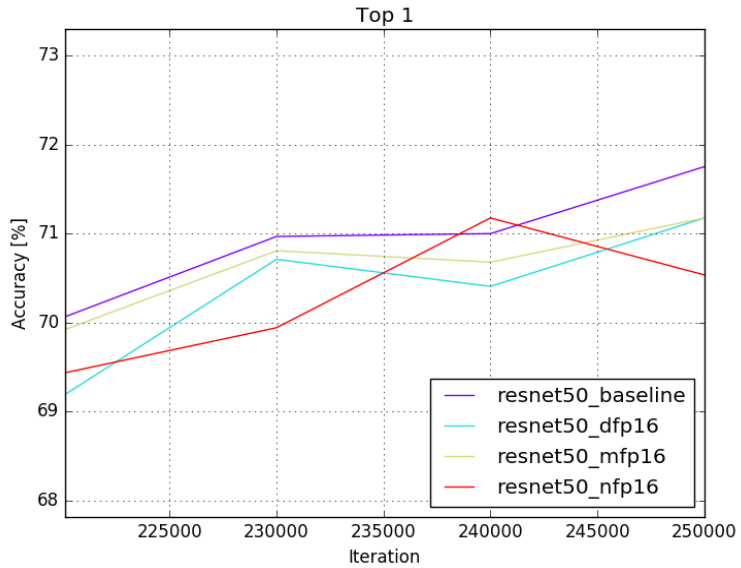
Nvcaffe-0.16, DGX-1, SGD with momentum, 100 epochs, batch=512, min augmentation, 1 crop, 1 model

RESNET-50 RESULTS

FP16 training is ok

FP16 storage has a small dip at the end (noise?)





Volta Mixed Precision Training

1. **Mixed precision with FP32 master weight storage**:
   Good results on 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
   - float master copy of weights

2. **FP16 master weight storage** worked for convnets after refactoring the solver

TRAINING WITH MIXED PRECISION IN NVCAFFE-0.16

NVIDIA/CAFFE-0.16

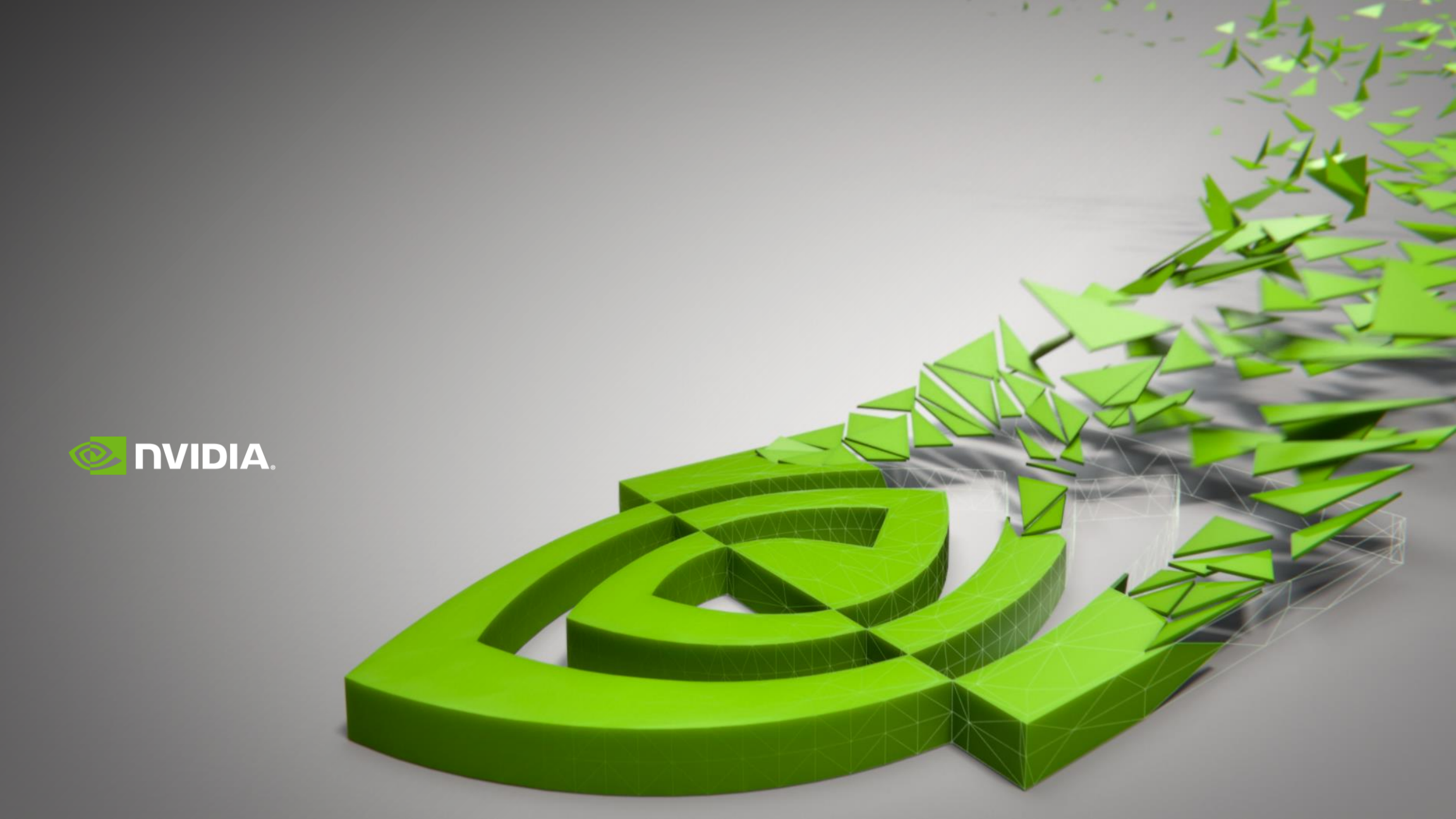
- Full float16 support
- Mixed precision:
  - Different *data types* for *Forward* and *Backward*
  - Different *math type*
  - *Solver_type (for* weight update in float16)
- Automatic type conversion

https://github.com/NVIDIA/caffe/tree/caffe-0.16

NVIDIA/CAFFE-0.16

```
name: "AlexNet_fp16"

default_forward_type:  FLOAT16
default_backward_type: FLOAT16

# default_forward_math:  FLOAT   # GP100 only
# default_backward_math: FLOAT   # GP100 only

global_grad_scale: 1000.

layer {
  forward_math:  FLOAT16
  backward_math: FLOAT
  …
}

solver_data_type: FLOAT16
```

https://github.com/NVIDIA/caffe/tree/caffe-0.16


NVIDIA.

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);
```

NVIDIA/CAFFE-0.16 - INTERNALS

```
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 – FUSED KERNELS

```
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) {
  CUDA_KERNEL_LOOP(i, N) {
    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]);
  }
}
```