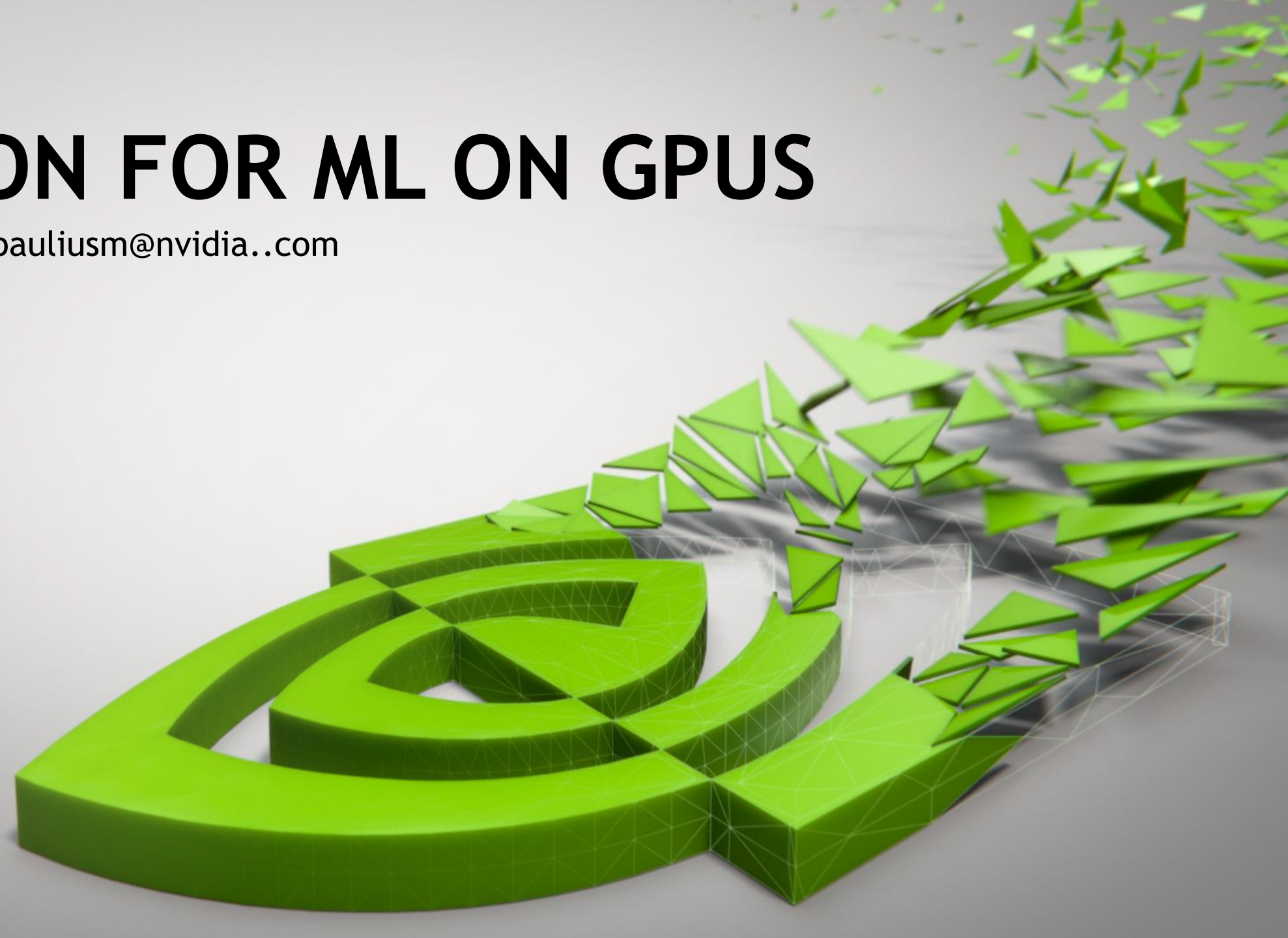
PRECISION FOR ML ON GPUS

Paulius Micikevicius, pauliusm@nvidia..com



Outline

- **Review of Volta and Turing architectures**
- **Inference**
- **Training**

GV100

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

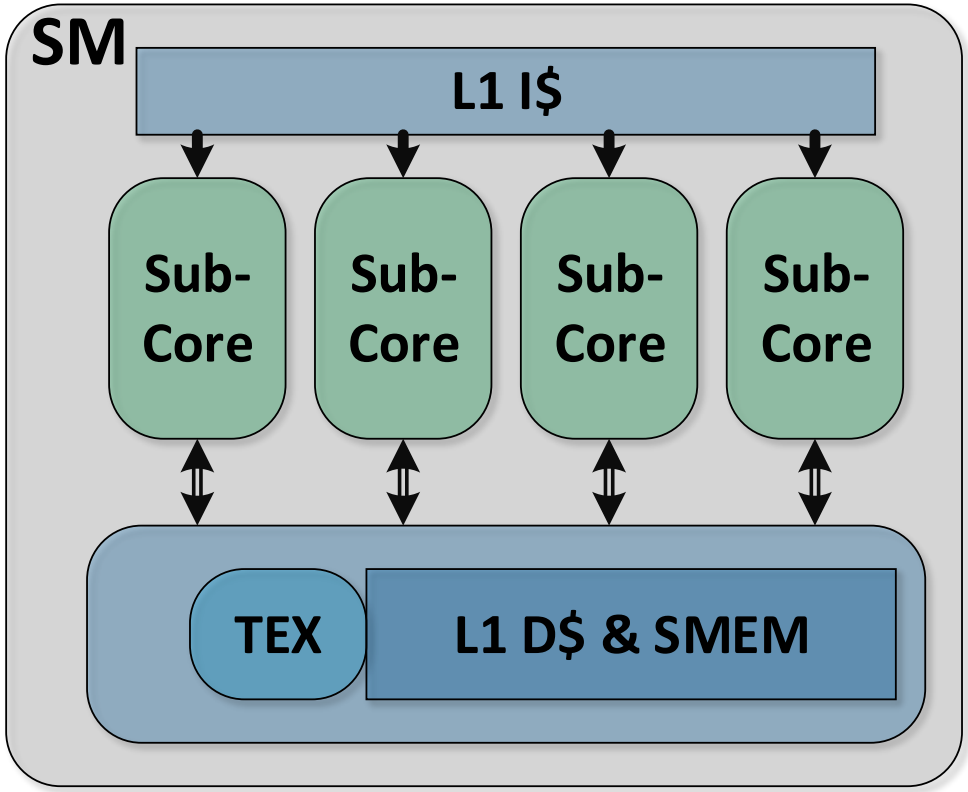
16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

*full GV100 chip contains 84 SMs

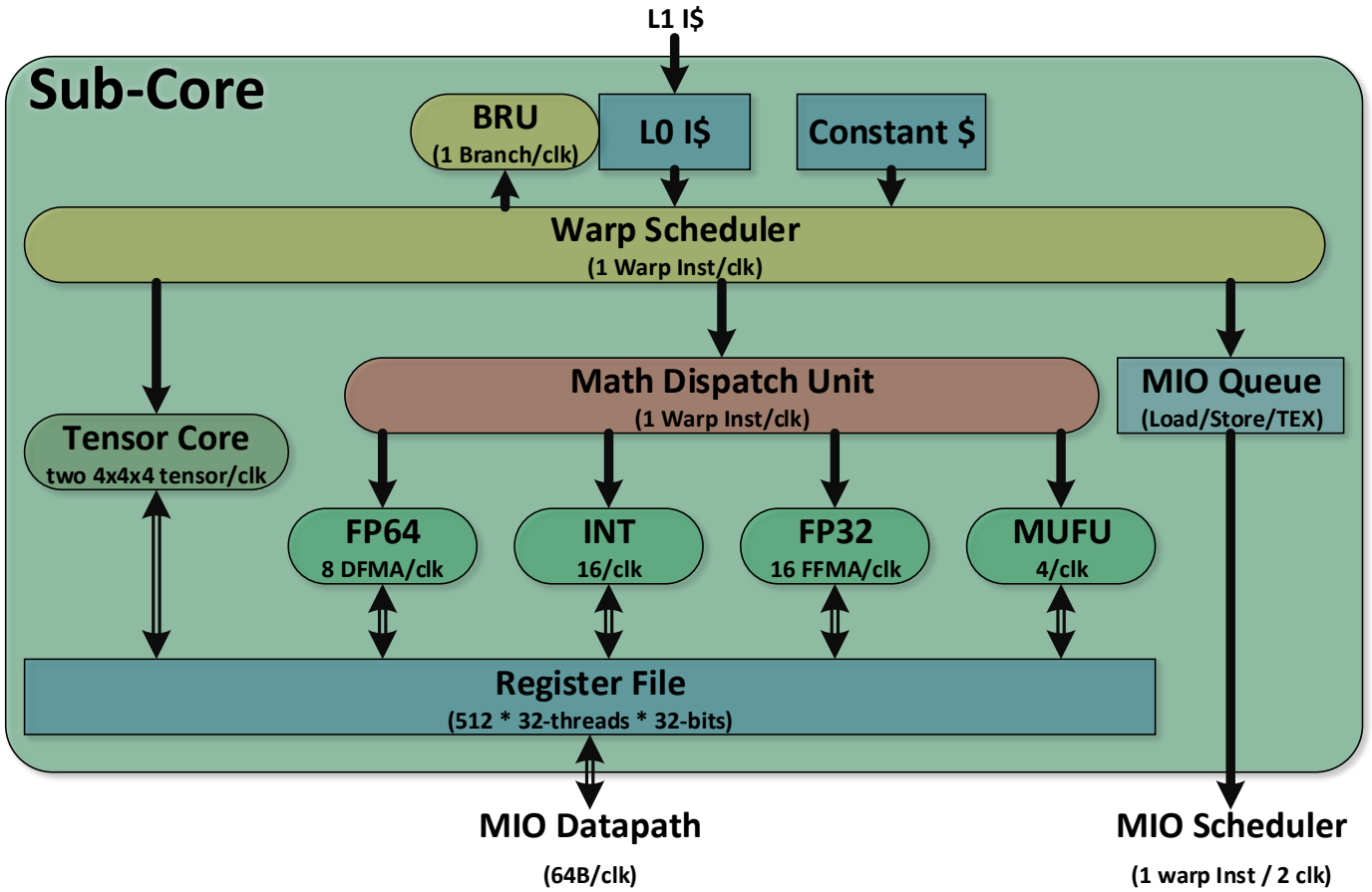
More details - Volta: Programmability and Performance (HotChips 2017)
https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.132-Volta-Choquette-NVIDIA-Final3.pdf

Volta SM


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex


SM
L1 I$
Sub-Core
Sub-Core
Sub-Core
Sub-Core
TEX
L1 D$ & SMEM

Volta SM Subcore



Warp Scheduler

- 1 Warp instr/clk
- L0 I$, branch unit

Math Dispatch Unit

- Keeps 2+ Datapaths Busy

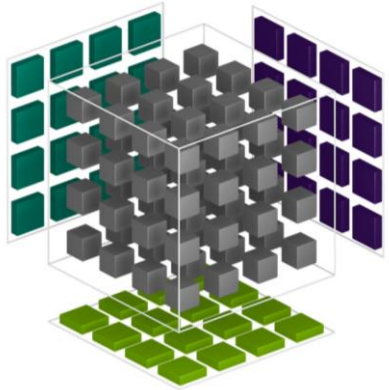
MIO Instruction Queue

- Hold for Later Scheduling

Two 4x4x4 Tensor Cores

TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices

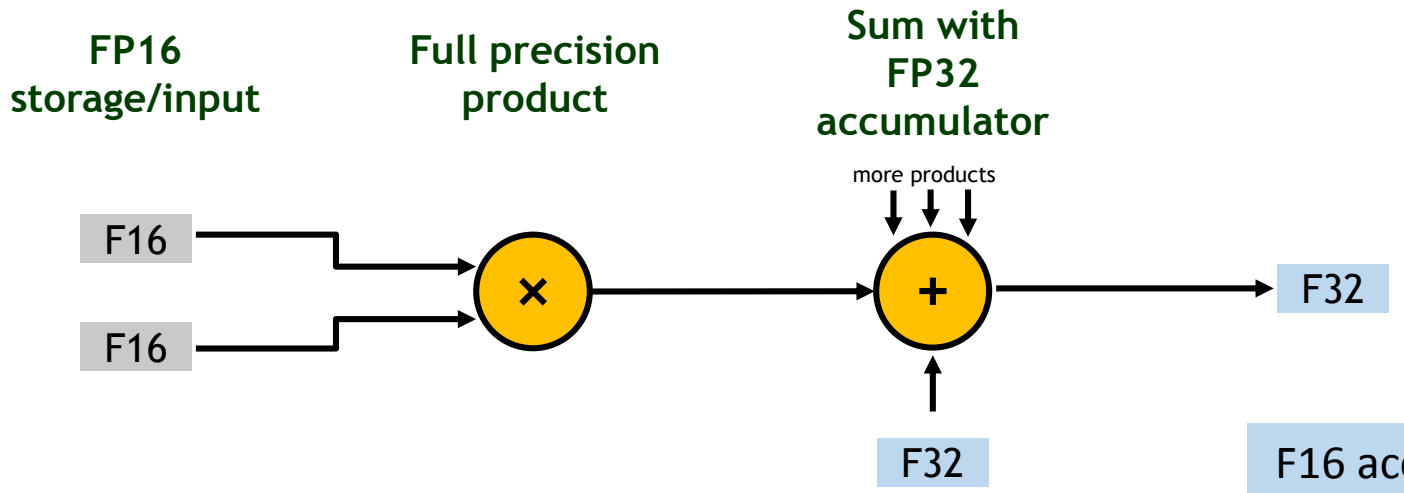


$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0}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

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

$$D = AB + C$$

Volta Tensor Cores



F16 accumulator is also available for inference

- https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/
- **Used by cuDNN and CUBLAS libraries**
- **Exposed in CUDA as WMMA**
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma
- **Accelerate convolutions and matrix multiplication**
  - A single instruction multiply-accumulates matrices
  - Think: computes many dot-products in parallel

Turing

- **Inference and Graphics oriented chip**
- **Extends Volta Tensor Core by adding matrix multiply accumulate for:**
  - Int8
  - Int4
  - Int1 (XOR followed by popcount)
- **All accumulate into int32**
- **Library and WMMA support for int8**
- **Int4 and int1 supported via WMMA in CUDA**
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma

Tensor Core Throughputs

Multiply-Accumulates per clock per SM

(multiply by 2x for ops counts)

	FP32	FP16	INT8	INT4	INT1
Volta	64	512			
Turing	64	512	1,024	2,048	8,192

Inference

- **FP16:**
  - Network can be trained in either FP32 or FP16
  - FP16 accumulation suggested for Tensor Cores
    - Some performance improvement due to power and register bandwidth savings
  - Normalization helps control activation value range
- **Fixed point:**
  - Some networks can be quantized after FP32/FP16 training
  - Some networks require fine tuning to maximize accuracy

Int8 Quantized Inference (TensorRT)

	FP32		INT8					
			Calibration using 5 batches		Calibration using 10 batches		Calibration using 50 batches	
NETWORK	Top1	Top5	Top1	Top5	Top1	Top5	Top1	Top5
Resnet-50	73.23%	91.18%	73.03%	91.15%	73.02%	91.06%	73.10%	91.06%
Resnet-101	74.39%	91.78%	74.52%	91.64%	74.38%	91.70%	74.40%	91.73%
Resnet-152	74.78%	91.82%	74.62%	91.82%	74.66%	91.82%	74.70%	91.78%
VGG-19	68.41%	88.78%	68.42%	88.69%	68.42%	88.67%	68.38%	88.70%
Googlenet	68.57%	88.83%	68.21%	88.67%	68.10%	88.58%	68.12%	88.64%
Alexnet	57.08%	80.06%	57.00%	79.98%	57.00%	79.98%	57.05%	80.06%
NETWORK	Top1	Top5	Diff Top1	Diff Top5	Diff Top1	Diff Top5	Diff Top1	Diff Top5
Resnet-50	73.23%	91.18%	0.20%	0.03%	0.22%	0.13%	0.13%	0.12%
Resnet-101	74.39%	91.78%	-0.13%	0.14%	0.01%	0.09%	-0.01%	0.06%
Resnet-152	74.78%	91.82%	0.15%	0.01%	0.11%	0.01%	0.08%	0.05%
VGG-19	68.41%	88.78%	-0.02%	0.09%	-0.01%	0.10%	0.03%	0.07%
Googlenet	68.57%	88.83%	0.36%	0.16%	0.46%	0.25%	0.45%	0.19%
Alexnet	57.08%	80.06%	0.08%	0.08%	0.08%	0.07%	0.03%	-0.01%

More details: 8-bit Inference with TensorRT (GTC 2017): http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf

Training: Mixed Precision Training

- **Reduced precision tensor math with FP32 accumulation, FP16 storage**
- **Benefits:**
  - 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

ILSVRC12 Classification Networks, Top-1 Accuracy

	FP32 Baseline	Mixed Precision
AlexNet	56.8%	56.9%
VGG-D	65.4%	65.4%
GoogLeNet	68.3%	68.4%
Inception v2	70.0%	70.0%
Inception v3	73.9%	74.1%
Resnet 50	75.9%	76.0%
ResNeXt 50	77.3%	77.5%

In all cases hyper-parameters were the same as for FP32 training

Detection Networks, mAP

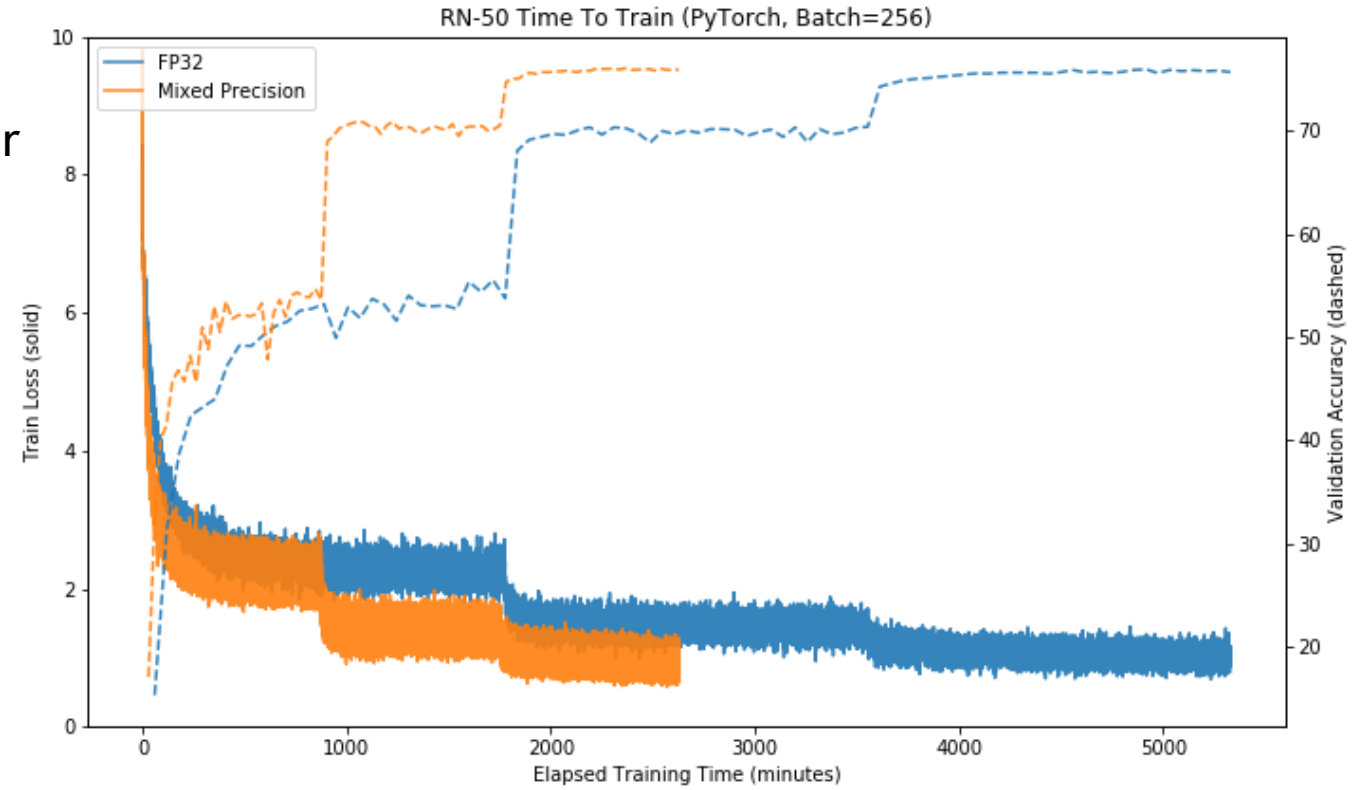
	FP32 Baseline	Mixed Precision
Faster R-CNN, VOC 07 data	69.1%	69.7%
Mask R-CNN, COCO data and eval	37.7%	37.7%
Multibox SSD, VOC 07+12 data	76.9%	77.1%

NVIDIA's proprietary automotive networks train with mixed-precision matching FP32 baseline accuracy.

Mixed Precision for Resnet50

4 Lines of Code => 2.3x Training Speedup

- Single-GPU runs using TorchVision ImageNet example
  - NVIDIA PyTorch 18.08-py3 container
  - **AMP for mixed precision**
- Minibatch=256 per GPU
- Single GPU RN-50 speedup for FP32 -> M.P. (with 2x batch size):
  - MxNet: **2.9x**
  - TensorFlow: **2.2x**
  - TensorFlow + XLA: **~3x**
  - PyTorch: **2.3x**
- Work ongoing to bring to 3x everywhere



Mixed Precision: Not Just for Imagenet

Model	FP32 -> M.P. Speedup	Comments
GNMT (Translation)	2.3x	Iso-batch size
FairSeq Transformer (Translation)	2.9x 4.9x	Iso-batch size 2x lr + larger batch
ConvSeq2Seq (Translation)	2.5x	2x batch size
Deep Speech 2 (Speech recognition)	4.5x	Larger batch
wav2letter (Speech recognition)	3.0x	2x batch size
Nvidia Sentiment (Language modeling)	4.0x	Larger batch

*In all cases trained to same accuracy as FP32 model

Speedups

- **Memory limited ops: should see ~2x speedup**
- **Math limited ops: will vary based on arithmetic intensity**
  - 125 Tflops : 900 GB/s -> 138 flops/B
- **Speedups continuously improve:**
  - libraries are continuously optimized
  - TensorCore paths are being added to more operation varieties

Tensor Core Performance Guidance

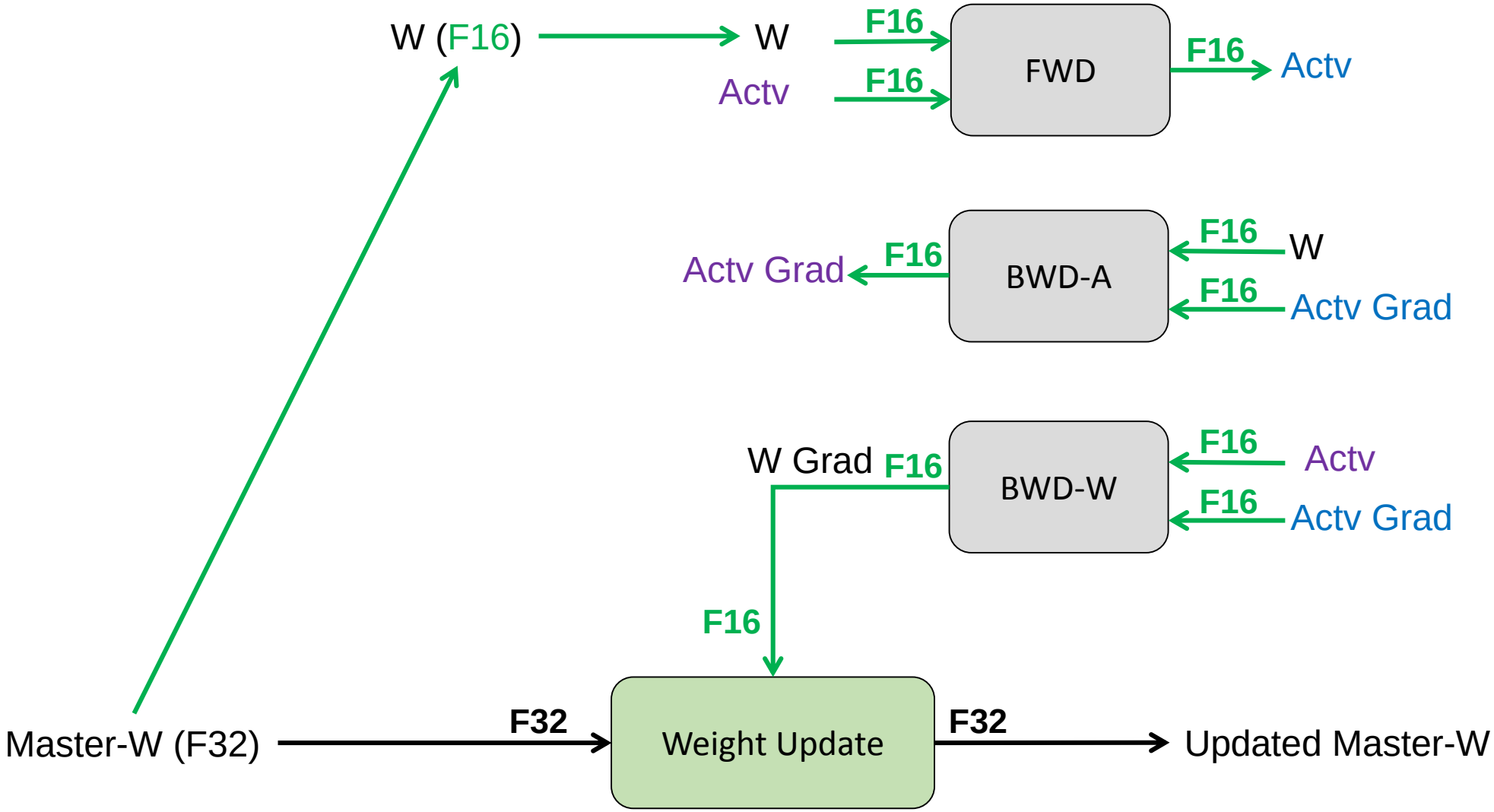
- **Requirements to trigger TensorCore operations:**
  - Convolutions:
    - Number of input channels a multiple of 8
    - Number of output channels a multiple of 8
  - Matrix Multiplies:
    - *M*, *N*, *K* sizes should be multiples of 8
    - Larger *K* sizes make multiplications more efficient (amortize 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

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

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 >> lr * dw*** then update doesn't change ***w***
    - Examples: multiplying a value by 0.01 leads to ~2^7 ratio, 0.001 leads to ~2^{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


W (F16)
W
Actv
F16
F16
FWD
F16
Actv
Actv Grad
F16
BWD-A
F16
W
F16
Actv Grad
W Grad
F16
BWD-W
F16
Actv
F16
Actv Grad
F16
Master-W (F32)
F32
Weight Update
F32
Updated Master-W

Pointwise Operations

- **FP16 is safe for most of these: ReLU, Sigmoid, Tanh, Scale, 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)

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 -> speedup

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

Convolution, Matrix Multiply

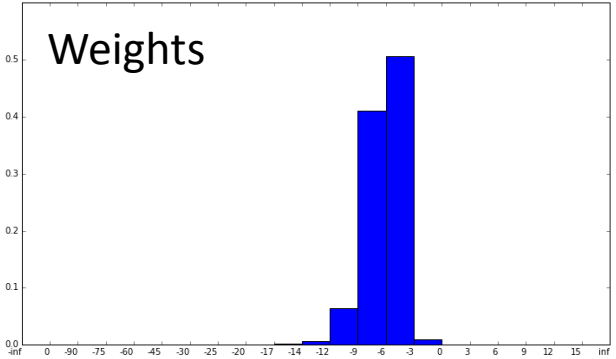
- **Fundamentally these are collections of dot-products**
- **Math: Tensor Cores starting with Volta GPUs**
  - Training: use FP32 accumulation
  - Inference: FP16 accumulation can be used
  - Many frameworks have integrated libraries with TensorCore support
    - http://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/
- **FP16 Storage (input and output)**

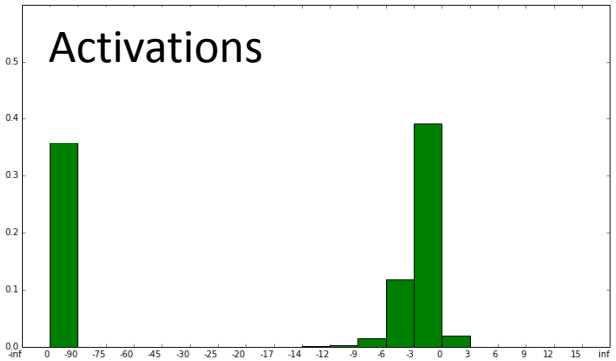
Summary so far

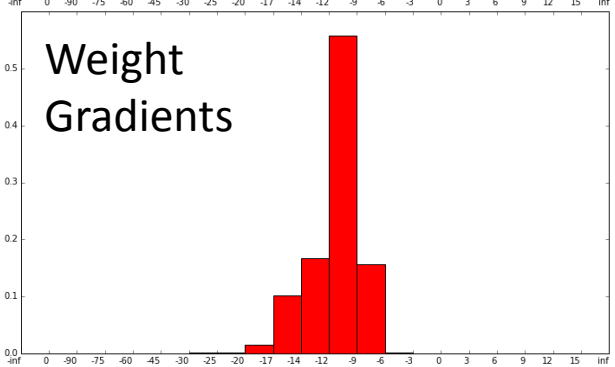
- **FP32 Master weights and update**
- **Math: FP32 and TensorCores**
- **Storage:**
  - Use FP16 for most layers
  - Use FP32 for layers that output probabilities or large magnitude values
    - Fuse to optimize speed and storage

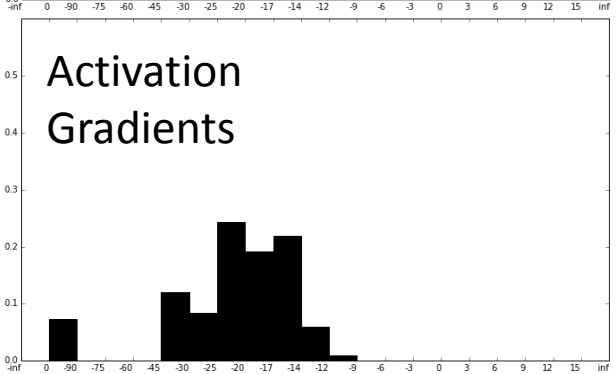
- **Example layer time breakdowns for FP32-only training:**
  - Resnet50 : ~73% convolutions, 27% other
  - DS2: ~90% convolutions and matrix multiplies (LSTM), ~10% other

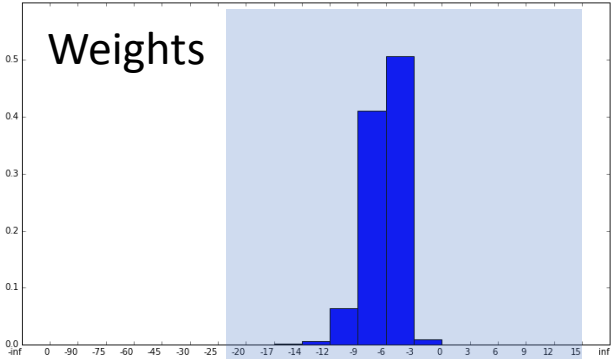
- **One more mixed-precision consideration: Loss Scaling**
  - Scale the loss, unscale the weight gradients before update/clipping/etc.
  - Preserves small gradient values

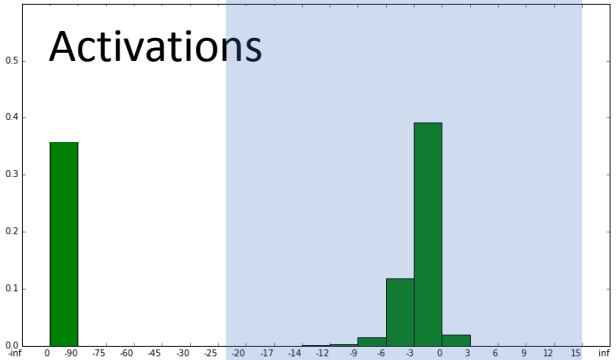

Weights
0.0
0.1
0.2
0.3
0.4
0.5
-inf 0 -90 -75 -60 -45 -30 -25 -20 -17 -14 -12 -9 -6 -3 0 3 6 9 12 15 inf

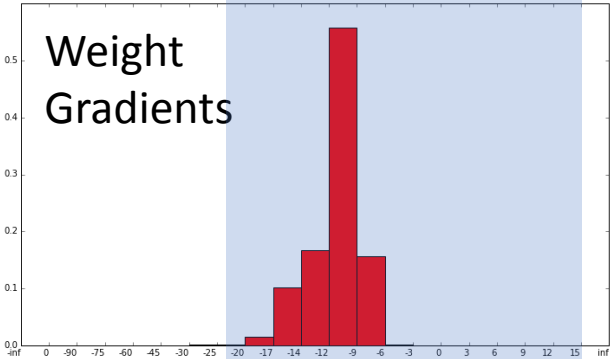

Activations
0.0
0.1
0.2
0.3
0.4
0.5
-inf 0 -90 -75 -60 -45 -30 -25 -20 -17 -14 -12 -9 -6 -3 0 3 6 9 12 15 inf

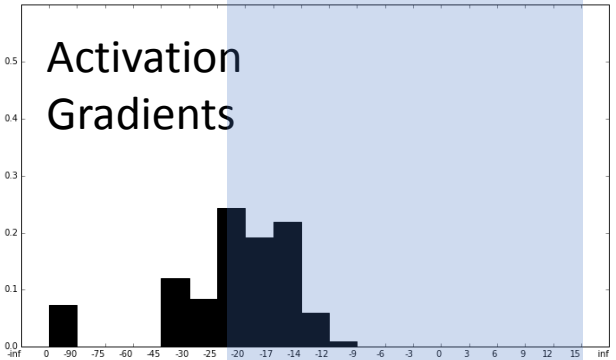

Weight
Gradients
0.0
0.1
0.2
0.3
0.4
0.5
-inf 0 -90 -75 -60 -45 -30 -25 -20 -17 -14 -12 -9 -6 -3 0 3 6 9 12 15 inf


Activation
Gradients
0.0
0.1
0.2
0.3
0.4
0.5
-inf 0 -90 -75 -60 -45 -30 -25 -20 -17 -14 -12 -9 -6 -3 0 3 6 9 12 15 inf

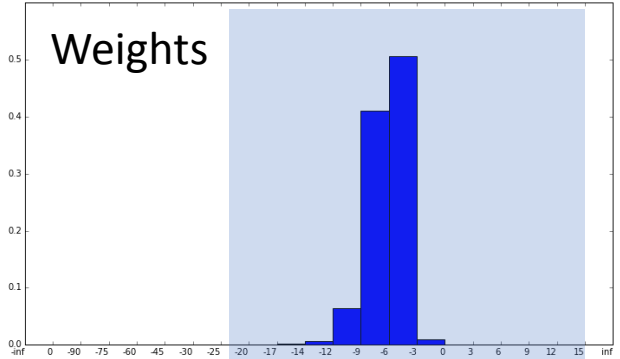


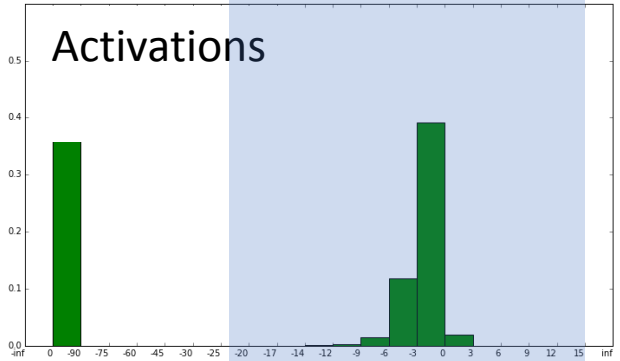


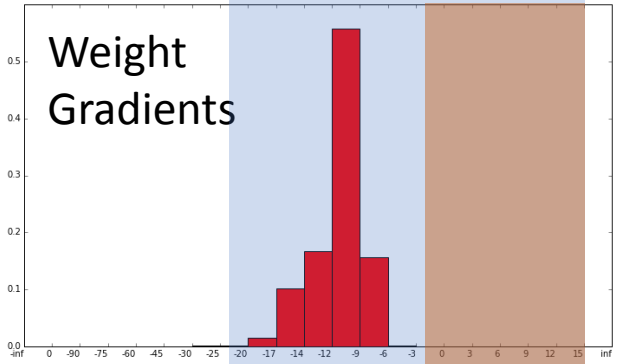


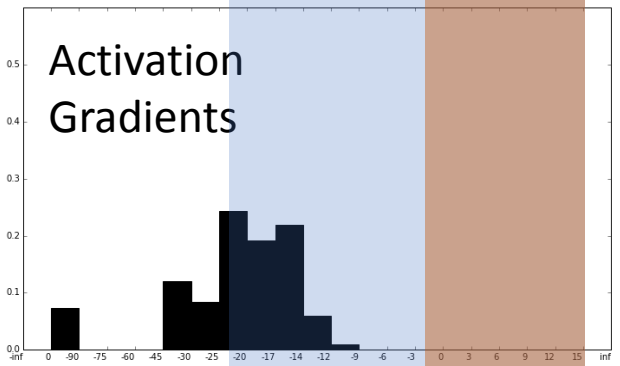


Range representable in FP16: ~40 powers of 2





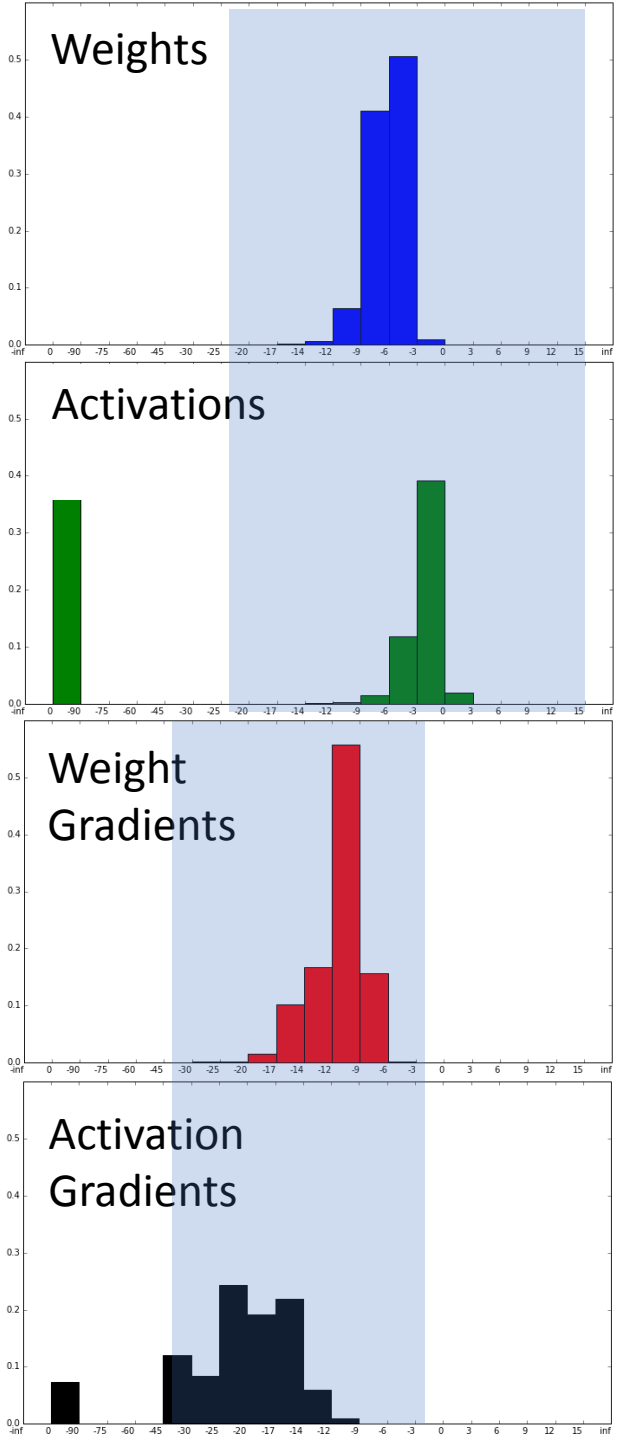




Range representable in FP16: ~40 powers of 2

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

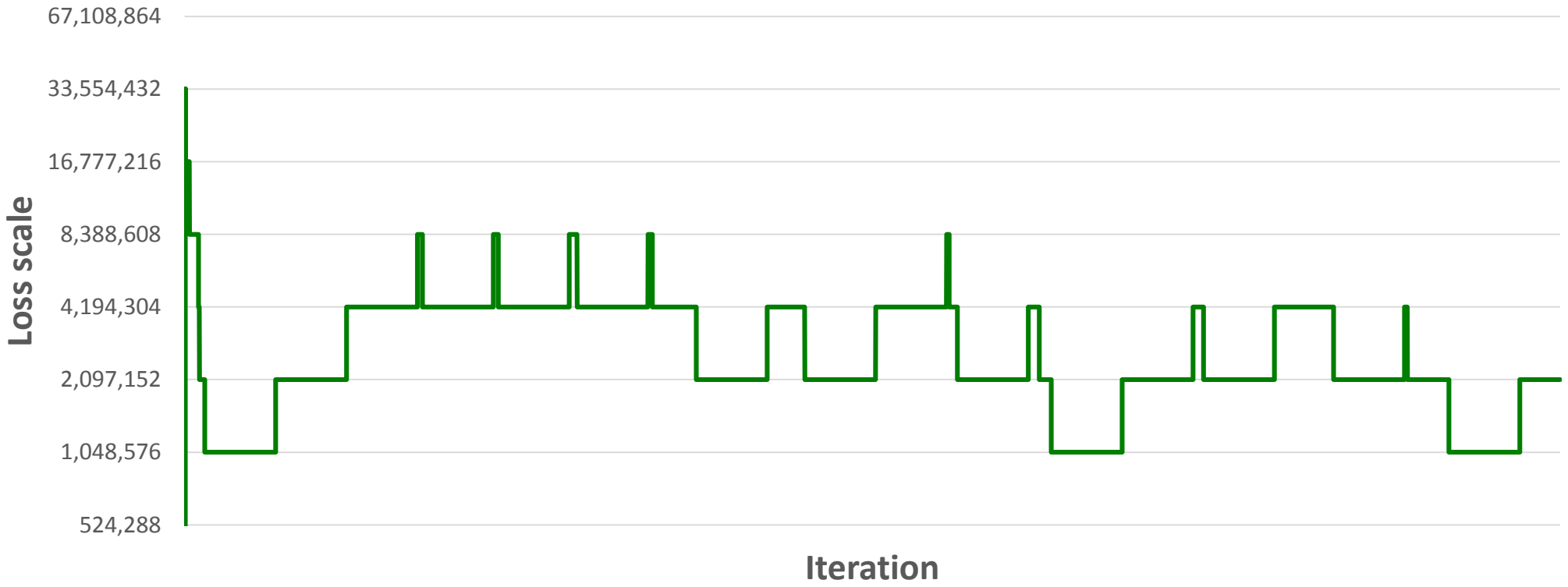
- **Algorithm**
  - Pick a scaling factor ***s***
  - for each training iteration
    - Make an fp16 copy of weights
    - Fwd prop (fp16 weights and activations)
    - Scale the loss by ***s***
    - Bwd prop (fp16 weights, activations, and gradients)
    - Scale *dW* by **1/*s***
    - Update *W*
-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

The automatic part

Automatic Loss Scale Factor for a Translation Net



Smallest scaling factor = 2^{20} -> max *dW* magnitude didn't exceed 2^{-5}

Automatic Loss Scaling Parameters

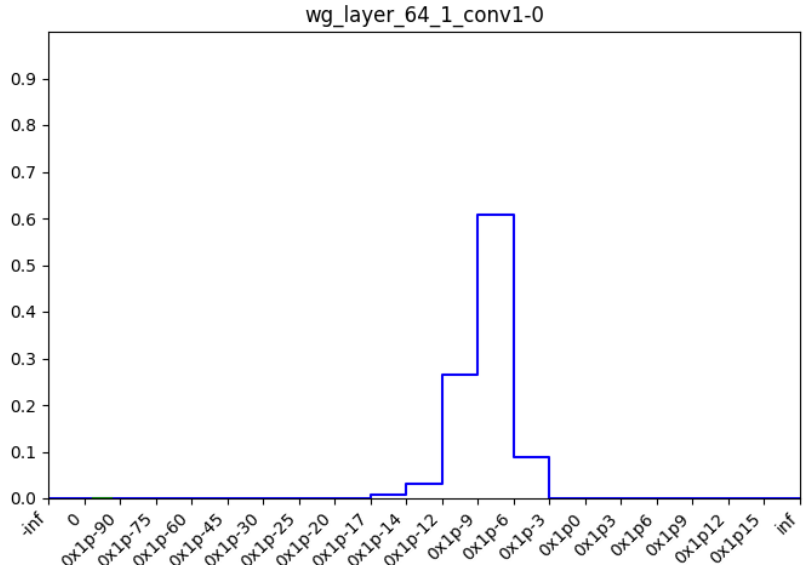
- **Factor for increasing/decreasing loss-scaling**
  - In our experiments we use **2**
- **Number of iterations without overflow**
  - In our experiments we use *N* = **2,000**
  - Separate study showed that randomly skipping 0.1% of updates didn't affect result
  - *N* = **2,000** gives extra margin 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

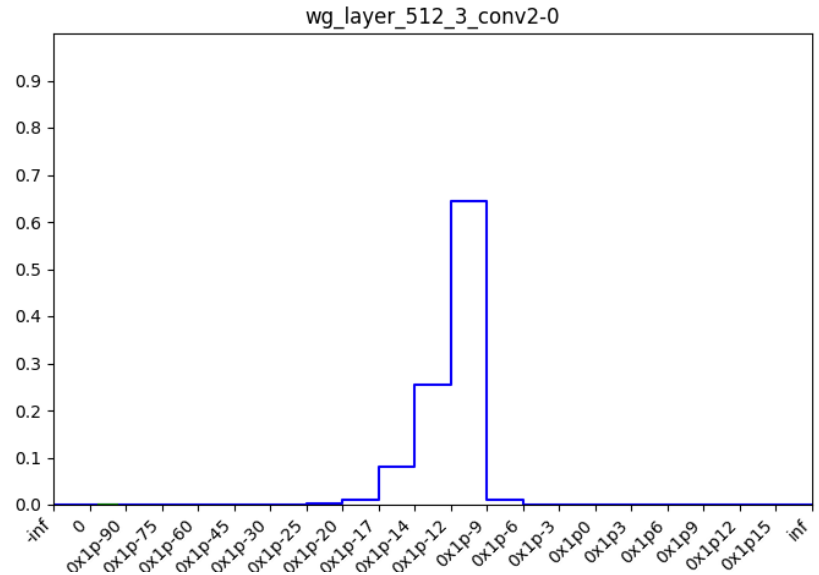
A Note on Gradients During Training

- **Popular belief:**
  - Gradient magnitudes become smaller during training
- **Observation:**
  - Not true for a number of networks
  - Thus FP16 range is not a concern
  - Normalizations especially tend to maintain distributions throughout training

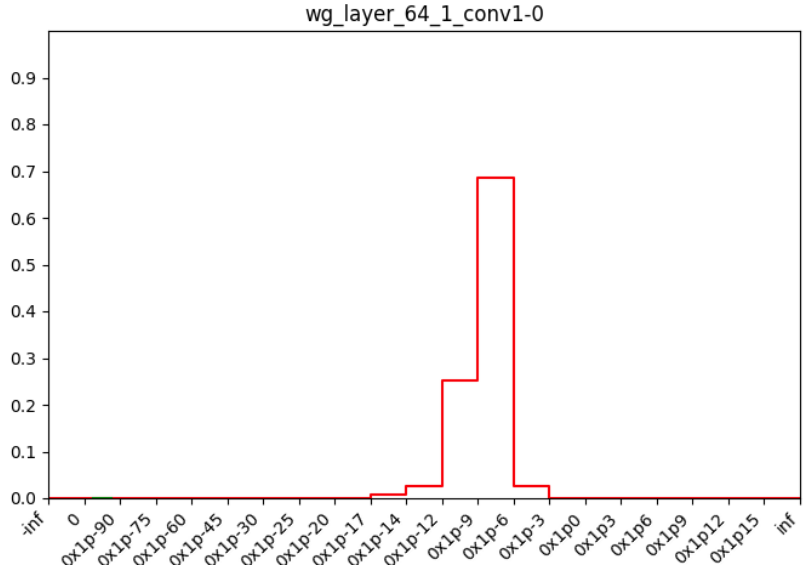
Resnet50 Weight Gradient Histograms

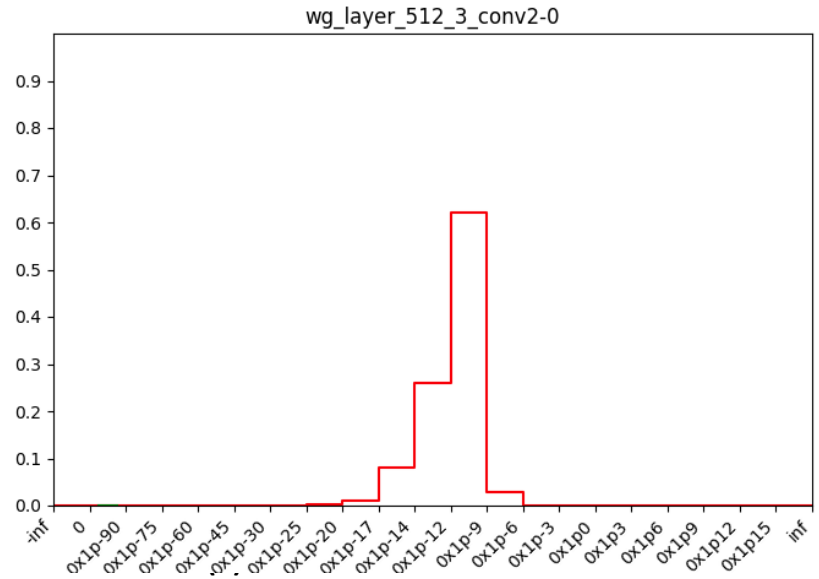
Early in training



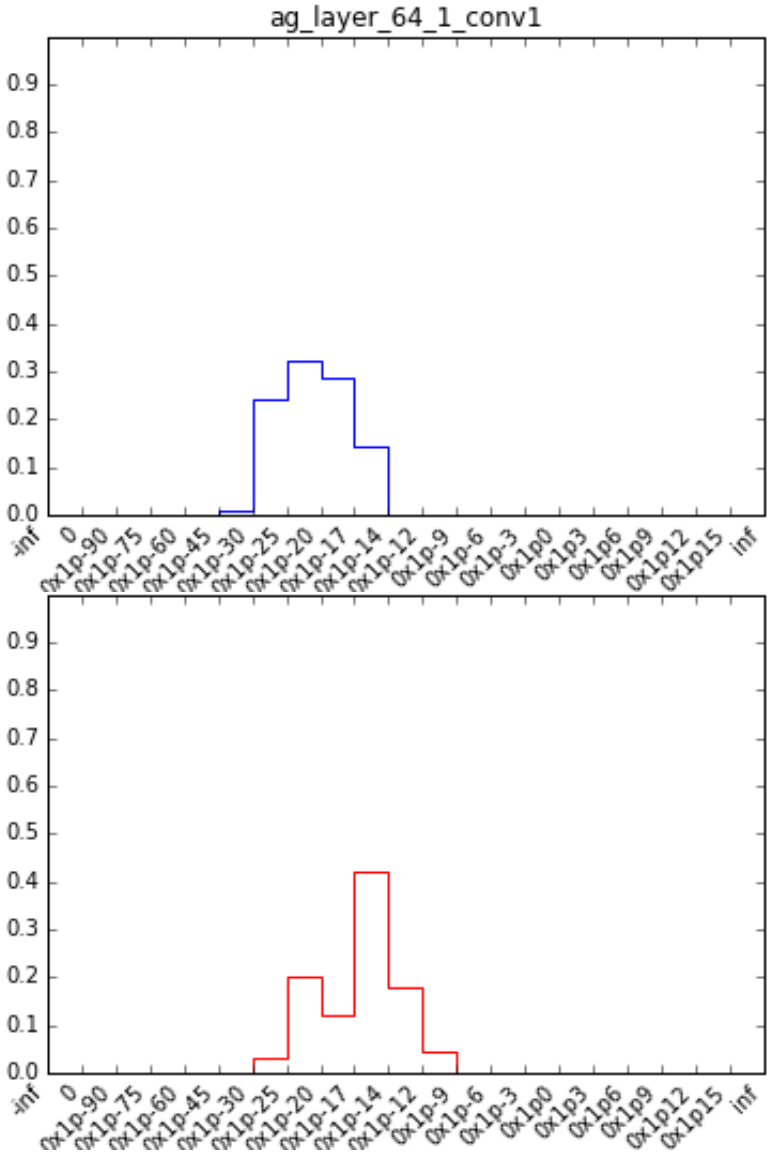


Late in training





Resnet50 Activation Gradient



VIDEO

Conclusions

- **Mixed precision training benefits:**
  - Math, memory speedups
  - Larger minibatches, larger inputs
- **Mixed precision matches FP32-trained 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
  - With the same hyper-parameters as FP32 training
- **Automatic Loss Scaling simplifies training**
  - For example, Amp for PyTorch: https://github.com/NVIDIA/apex/tree/master/apex/amp
- **Inference:**
  - Can be purely FP16: storage and math (use library calls with FP16 accumulation)
  - Turing Tensor Cores also provide int8/int4/int1 matrix operations

Backup

Language Translation

- **GNMT:**
  - https://github.com/tensorflow/nmt
  - 8 layer encoder, 8 layer decoder, 1024x LSTM cells, attention
  - German -> English (train on WMT, test on newstest2015)
  - **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
- **Transformer:**
  - MLPerf
  - English to German
  - **FP32 and Mixed Precision: ~25 BLEU**
    - MLPerf v0.5.0 target accuracy, to be increased in the future

Speech

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

Character Error Rate (lower is better)

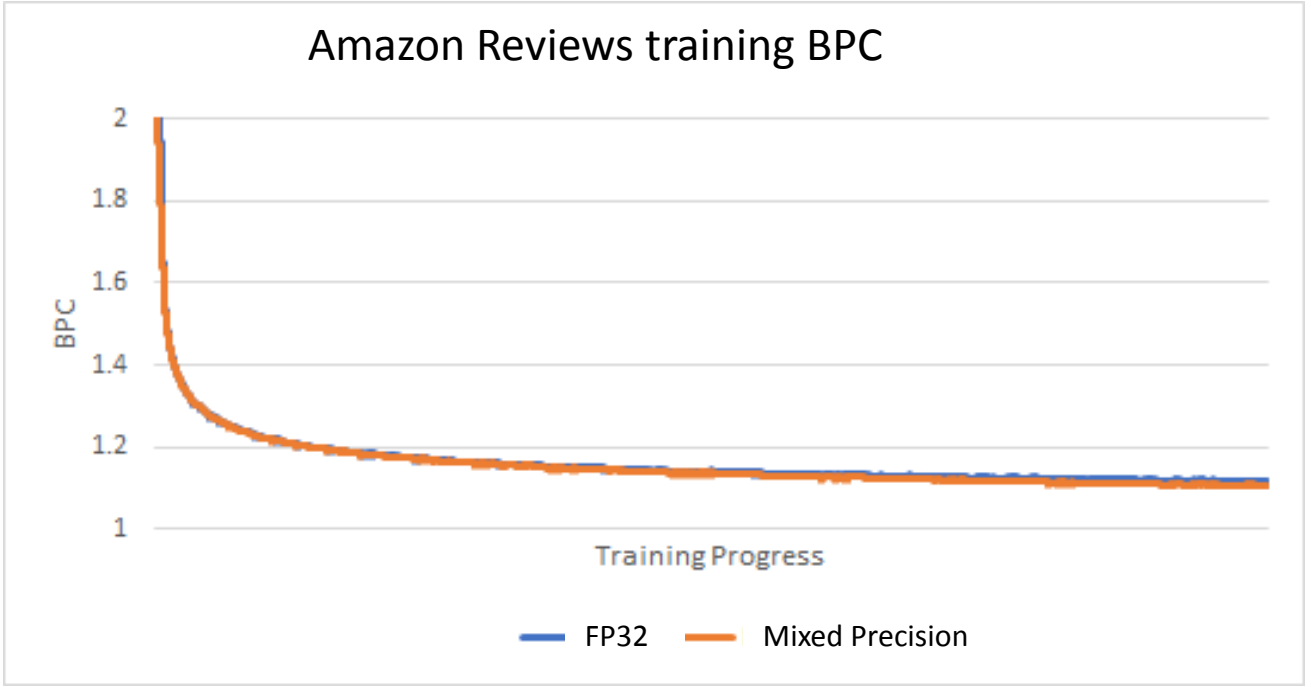
	FP32 Baseline	Mixed Precision
English	2.20	1.99
Mandarin	15.82	15.01

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

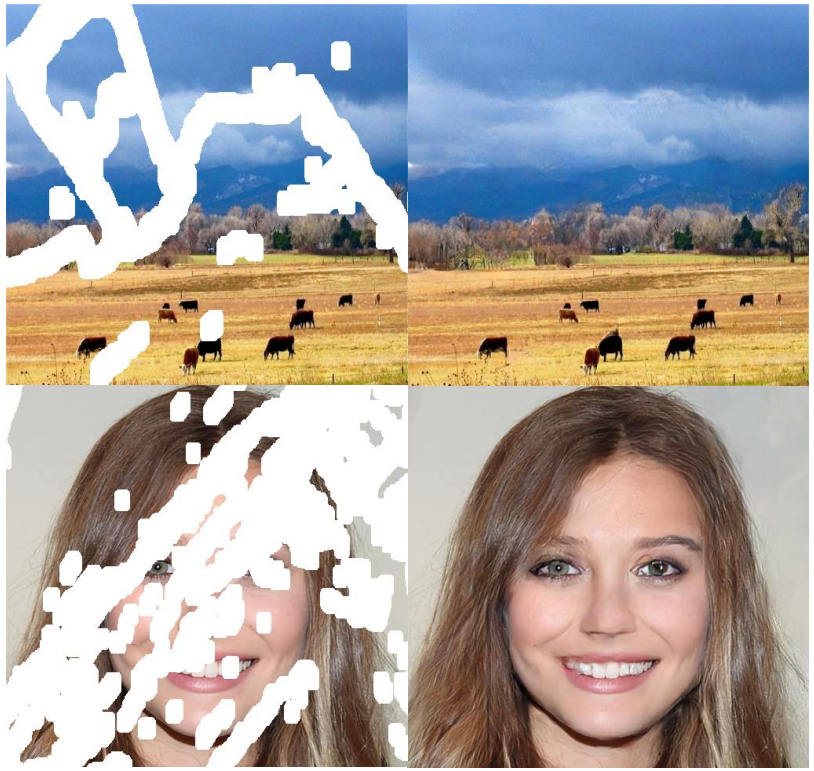
- **Multiplicative LSTM, based on https://arxiv.org/abs/1704.01444**



	Train BPC	Val BPC	SST acc	IMDB acc
FP32	1.116	1.073	91.8	92.8
Mixed Precision	1.115	1.075	91.9	92.8

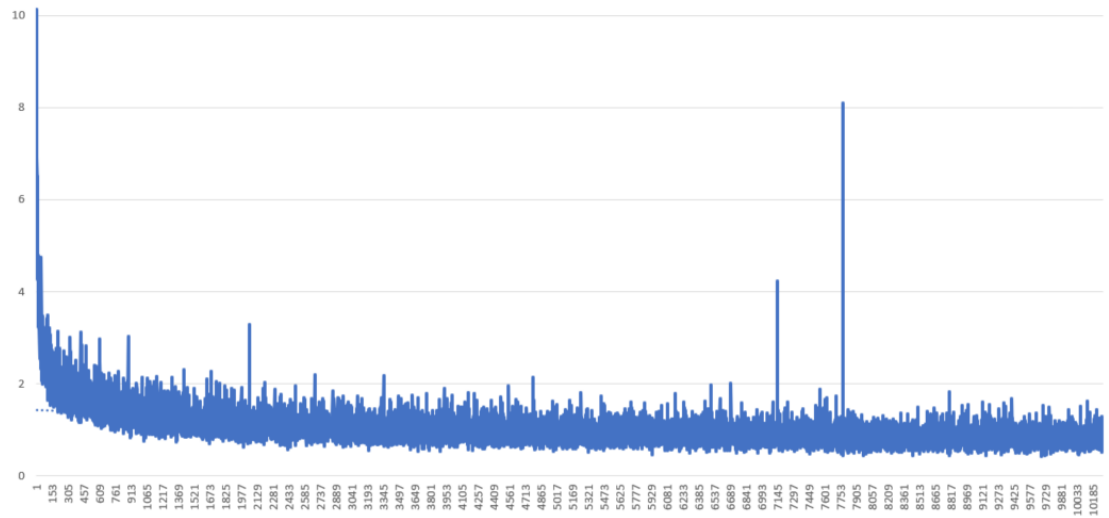
Image Inpainting

- **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



Input Inpainted Result

Image Inpainting : result



Training Loss Curve

Testing Input

Mixed Precision Result

FP32 Result

Text to speech synthesis

Using Tacotron 2

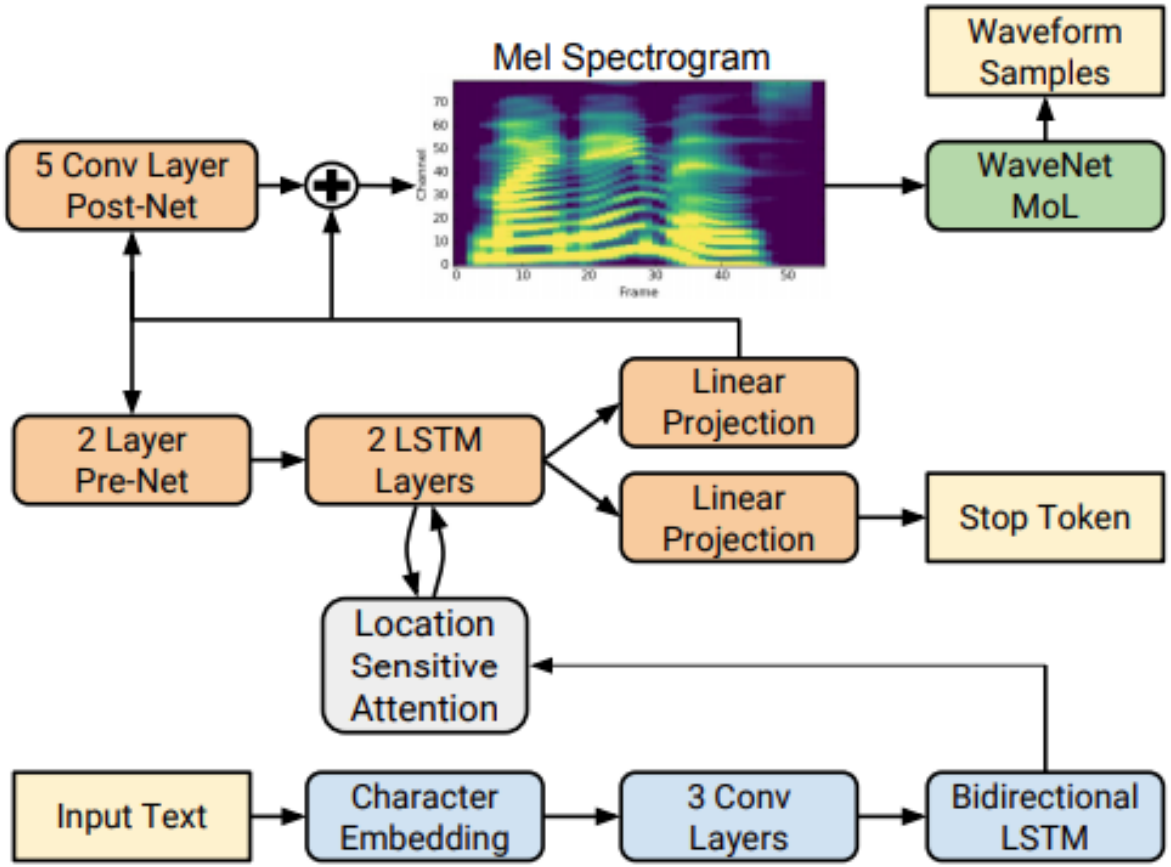
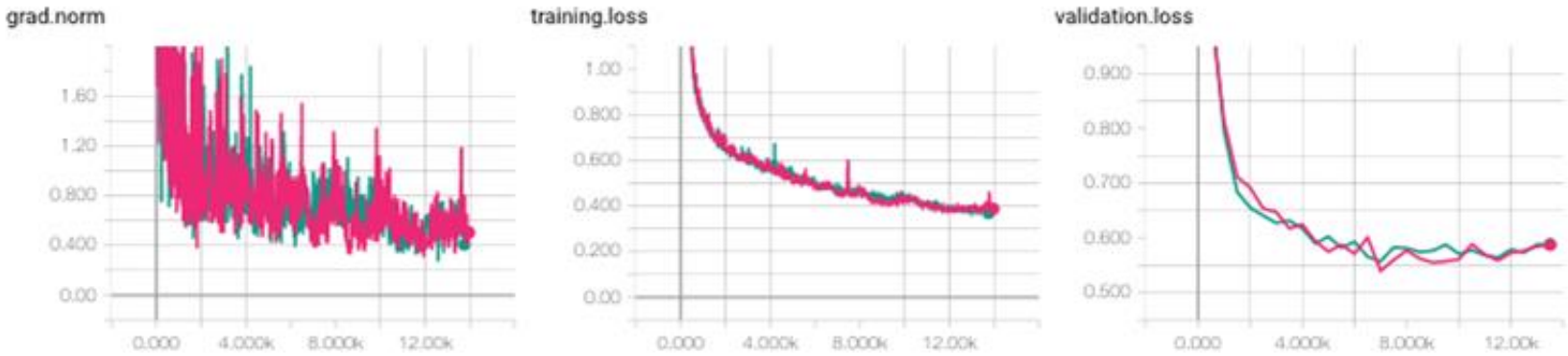


Fig. 1. Block diagram of the Tacotron 2 system architecture.

Shen et al, Natural TTS Synthesis by Conditioning Wavenet on Mel-Spectrogram Predictions, https://arxiv.org/abs/1712.05884

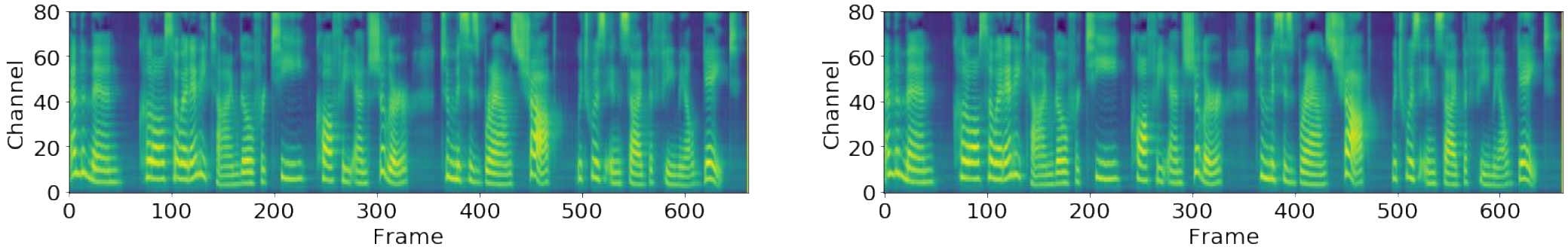
Text to speech synthesis : results

Mixed Precision: Pink



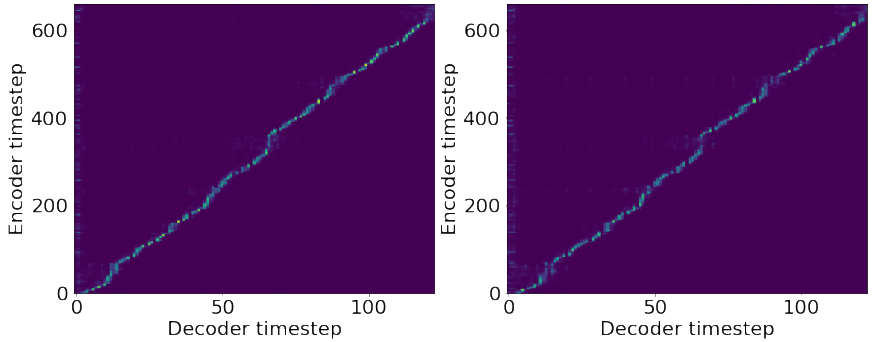
FP32: Green

Predicted Mel-Spectrograms



Predicted Alignments

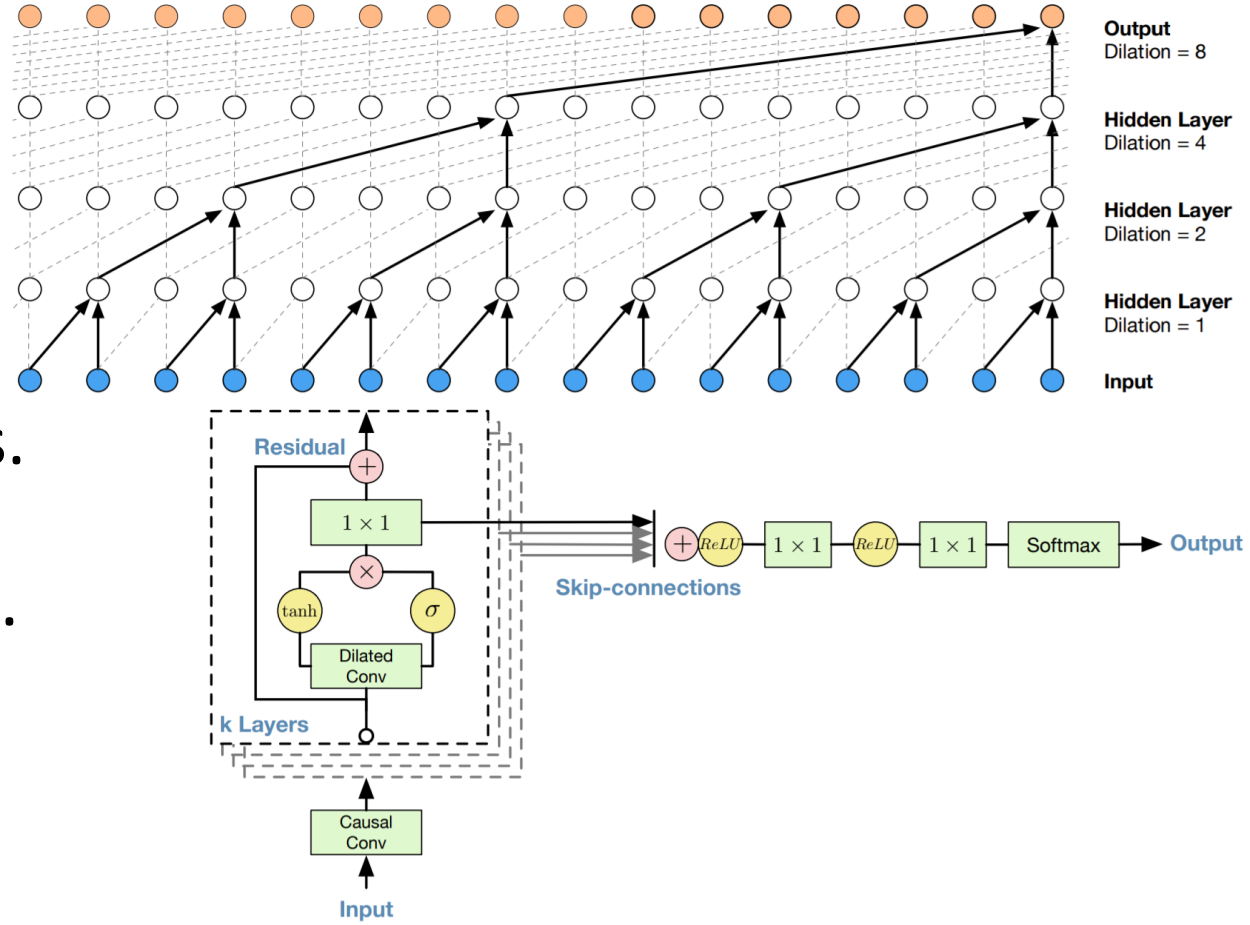
Mixed Precision



FP32

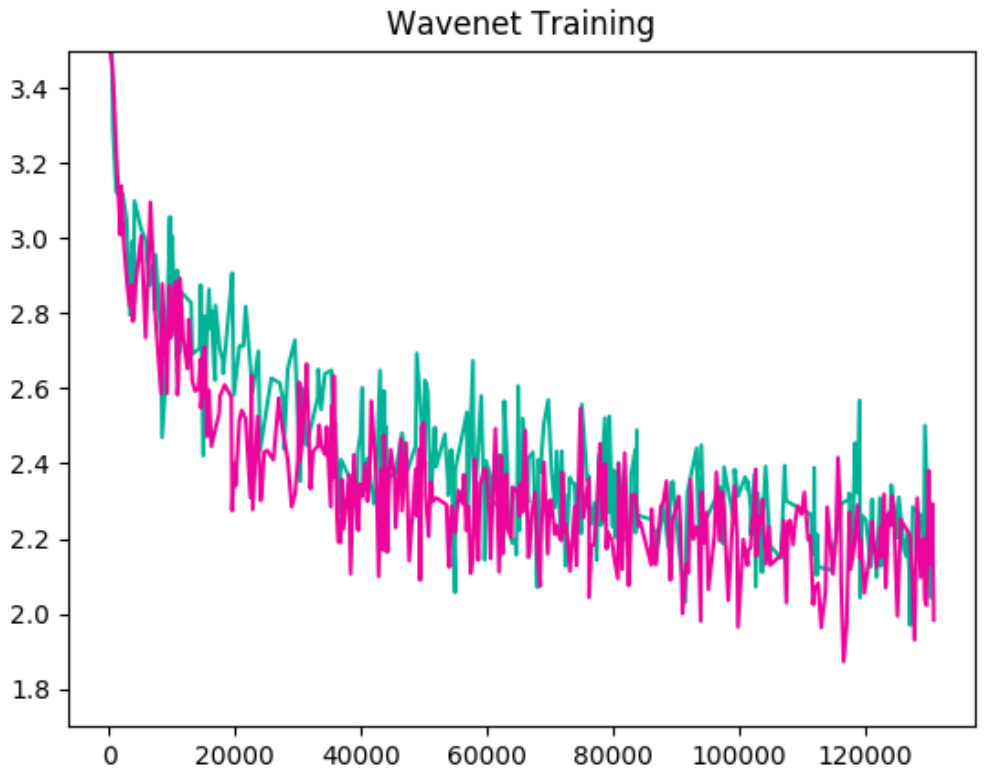
Wavenet

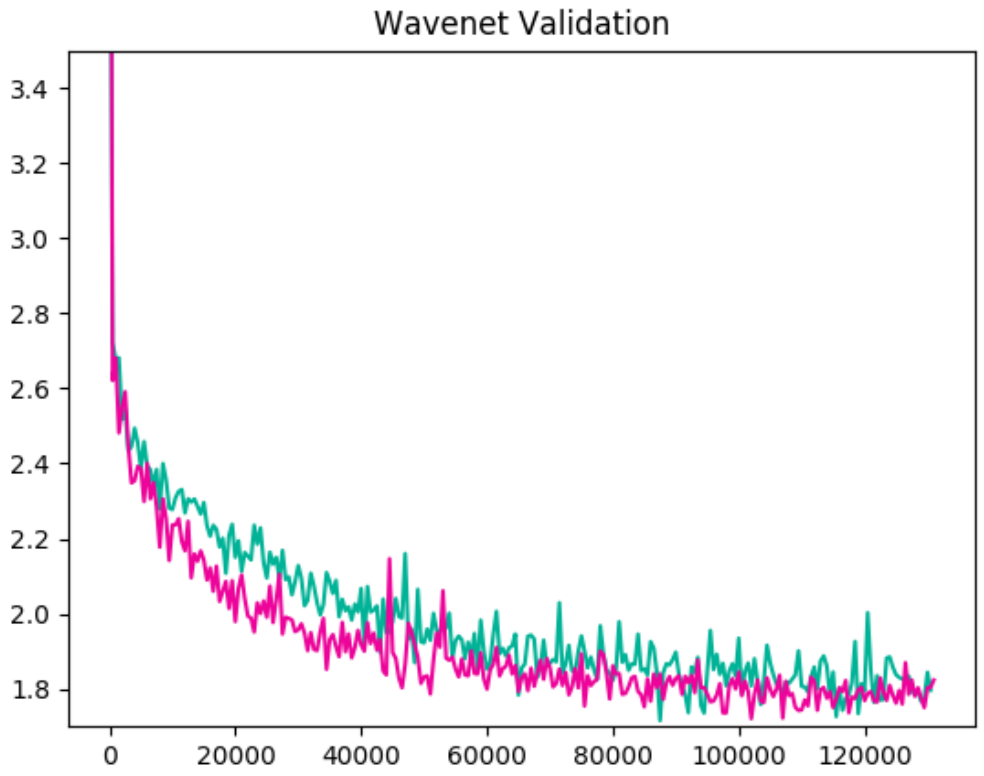
- 12 Layers of dilated convolutions
- Dilations reset every 6 layers
- 128 channels for dilated convs.
  (64 per nonlinearity)
  64 channels for residual convs.
  256 channels for skip convs.



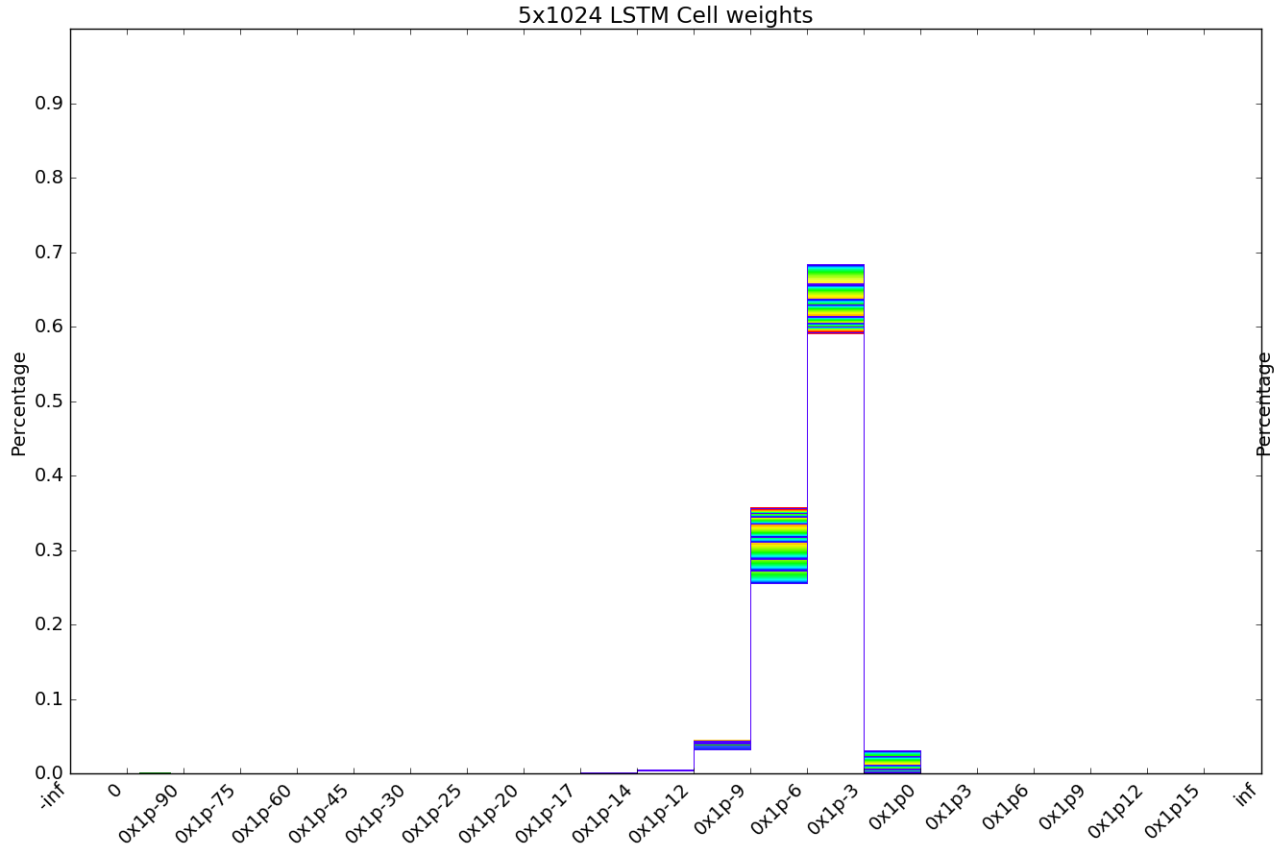
Wavenet : results

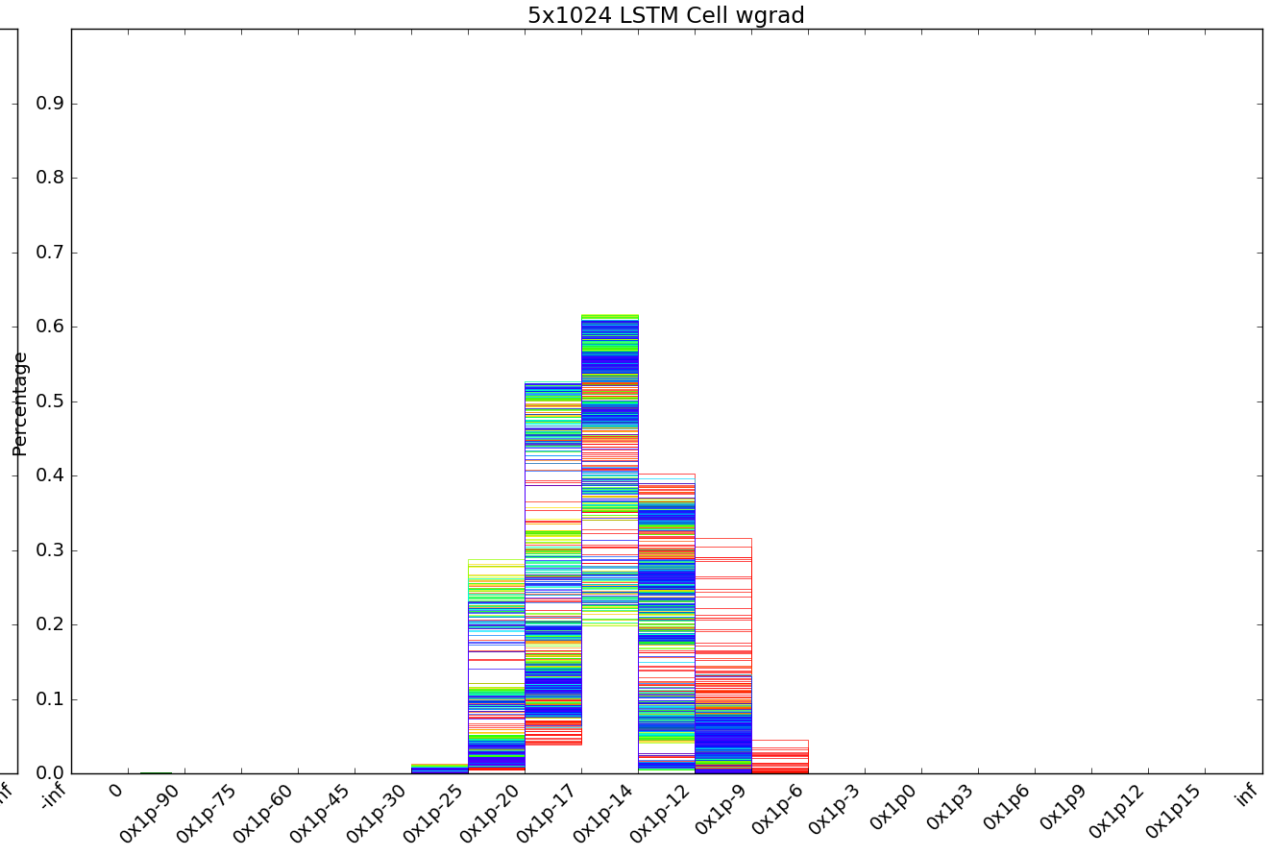
Mixed precision: Pink FP32: Green





Weights and wgrad of LSTM cells





- Weights of LSTM are never >1
- Wgrad of LSTM cells rarely go below subnormal fp16 range
- See more detailed break down of all layers in html file

Data and dgrad of LSTM cells