Automatic Mixed Precision (AMP) Training

Bojian Zheng

Vector NLP Meeting

Acknowledgement: Most materials on this slides are based on:

[1] S. Narang, P. Micikevicius et al. *Mixed Precision Training* (ICLR 2018).

[2] M. Conley, M. Sun et al. *Mixed precision Grappler optimizer* (Tensorflow Pull Request #26342, March 2019).

Automatic Mixed Precision

Motivation

Why Low Precision?

Common Training Issues

× **Compute-Heavy**

- Days even weeks to train.

× **GPU Memory Capacity Limited**

- Large models (e.g., BERT-Large) cannot fit into a single GPU.
- Even if possible, small training batch size limits data parallelism.

Your hardware supports it!
Do NOT waste it.

Low-Precision Benefits

✓ **Lower Arithmetic Complexity**

⇒ Performance ↑↑↑

✓ **Less GPU Memory Footprint**

- FP16 requires **half** of the storage needed by FP32.
- Side Effects:
  - save memory & network bandwidth
  - increase batch size

⇒ Further Performance ↑↑↑

Why Mixed Precision?

Low-Precision Cost

× **Small Dynamic Range**

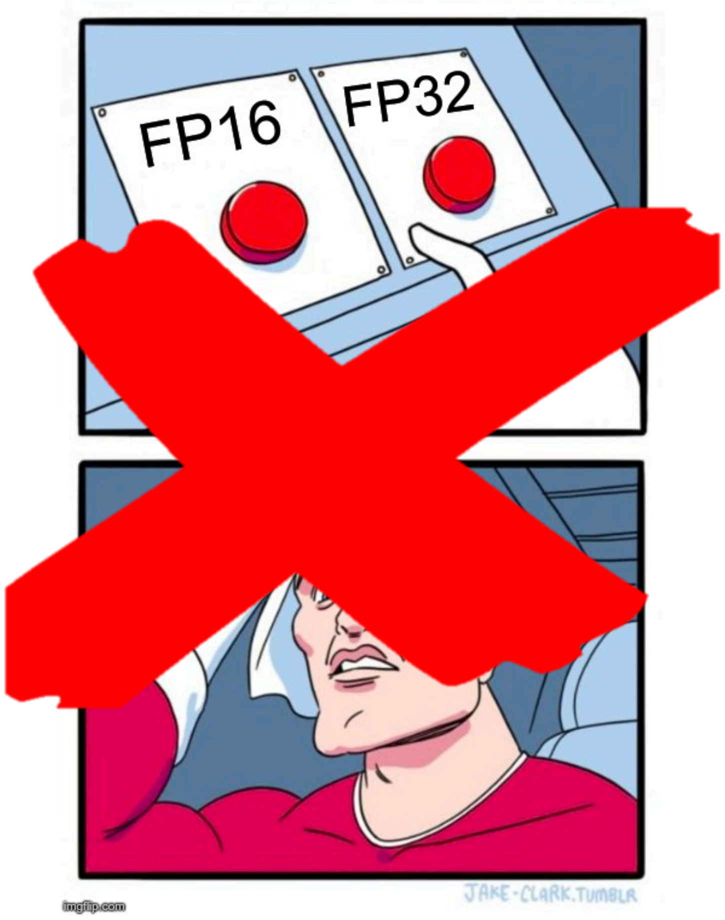
- Numeric Overflow/Underflow

⇒ Model Accuracy Loss,
even **Divergence**

- **Mixed-Precision**
  - A mixture of FP16 and FP32.
  - Where FP32 …
    - Handles computations that are numerically-dangerous.
    - Serves as a backup plan.
- But how to *mix*? Manually?

Why Automatic Mixed Precision?

- SOTA frameworks now support **Automatic** Mixed Precision.
  - E.g., TensorFlow, PyTorch & MXNet
  - Automatically leverage the power of FP16 with minor code changes or environment variables.



Automatic Mixed Precision

Under the Hood

Key Question

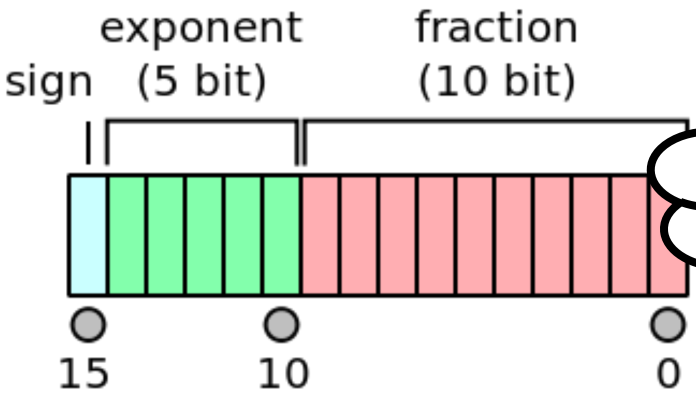
- Why would FP16 training diverge?

Arithmetic Overflow/Underflow

- Why & When would it happen?
  - **Numerically-Dangerous** Ops [2]
    - Exponential, Logarithmic
    - **Reduction Sum**

Why is Reduction Sum considered **dangerous**?

- E.g., $1 + 10^{-4} = ?$
  - In FP32, the answer is 1.0001, but in FP16, the answer is 1.
- *TL'DR* FP16 '+' is **ineffective** if the two operands are different by more than $\mathbf{2k}$.

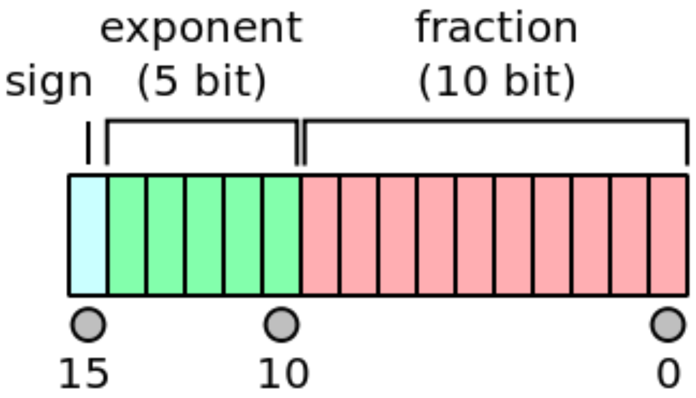


When would it happen?

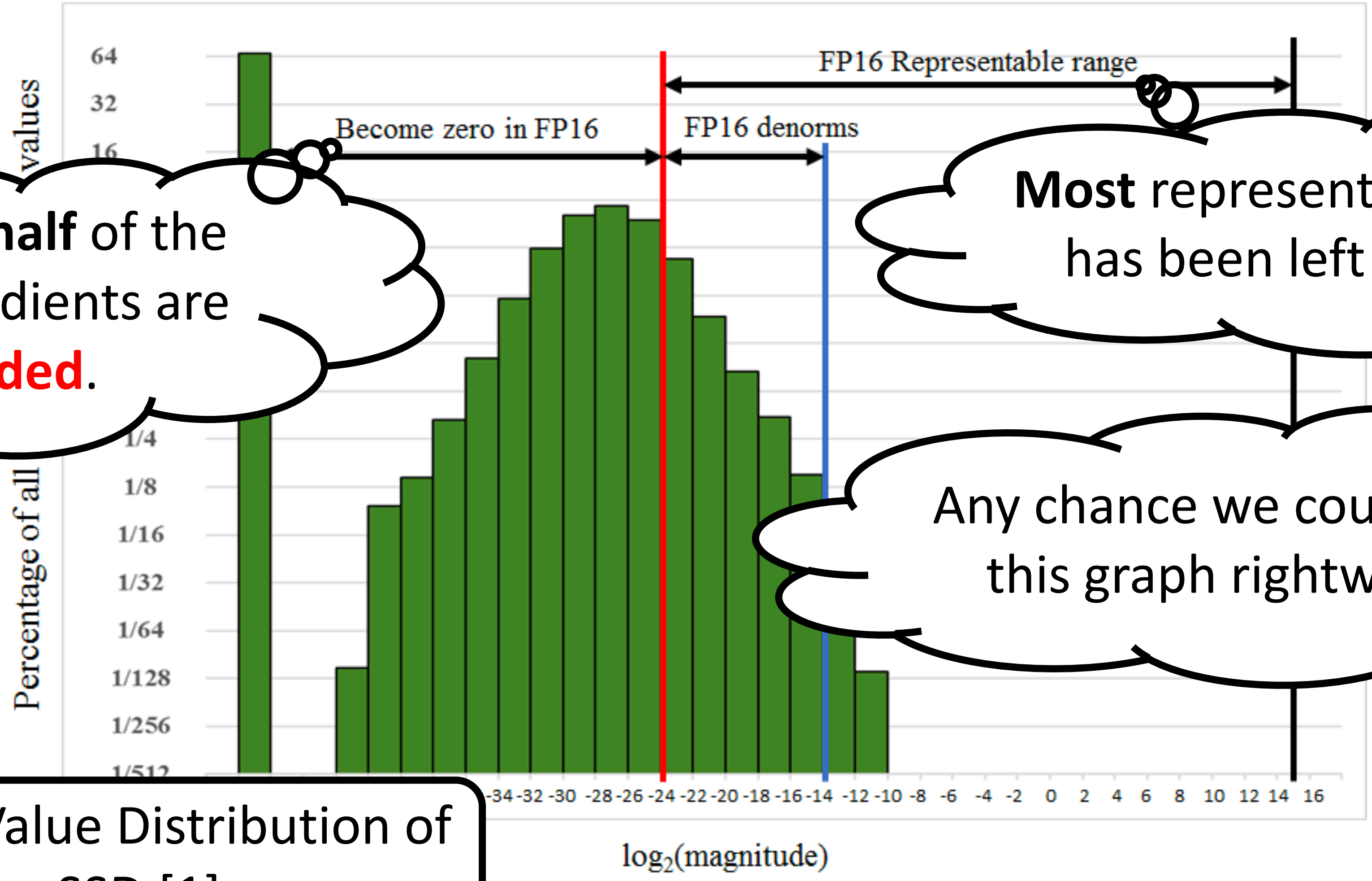
Arithmetic Overflow/Underflow

- Why & When would it happen?
  - **Numerically-Dangerous Ops** [2]
    - Exponential, Logarithmic
    - **Reduction Sum**
  - **Weight Update** [1]
    - Gradients are often too small when compared with the weights.
    - Many are even NOT representable.

- E.g., $1 + 10^{-4} = ?$
  - In FP32, the answer is 1.0001, but in FP16, the answer is 1.
- *TL'DR* FP16 '$+$' is **ineffective** if the two operands are different by more than $\mathbf{2k}$.



Arithmetic Overflow/Underflow (Cont.)



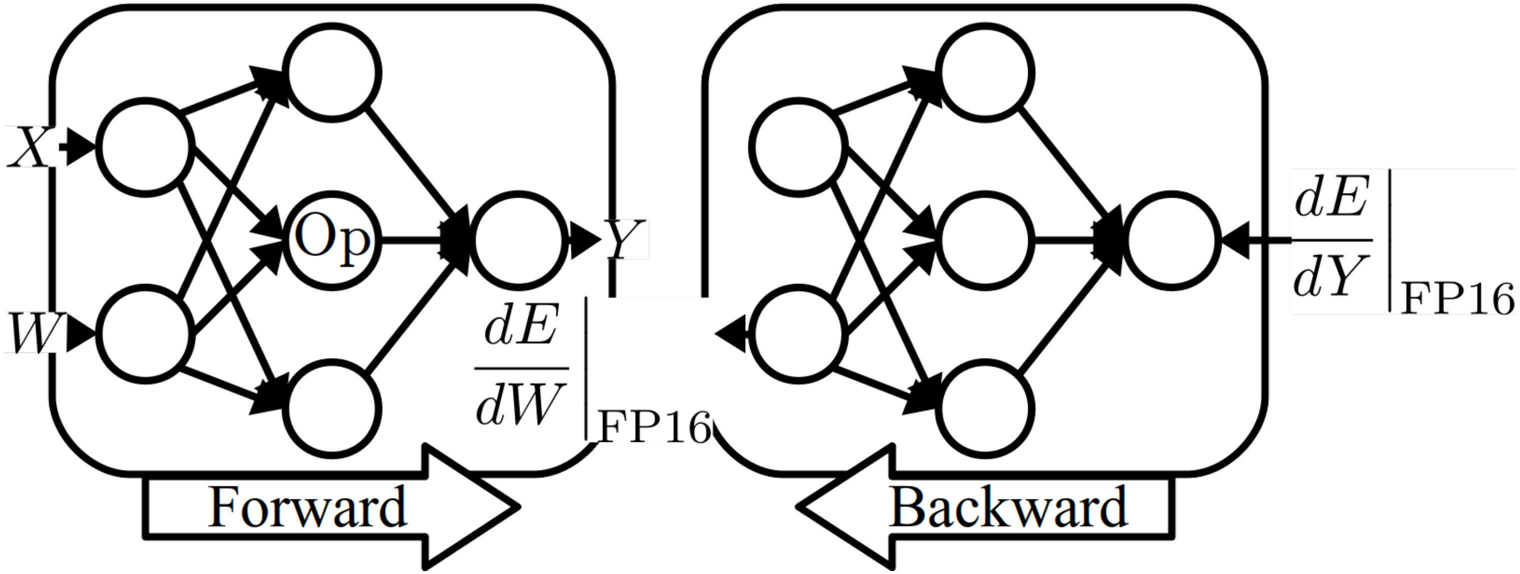
More than **half** of the nonzero gradients are **grounded**.

Most representable range has been left **unused**.

Any chance we could shift this graph rightward?

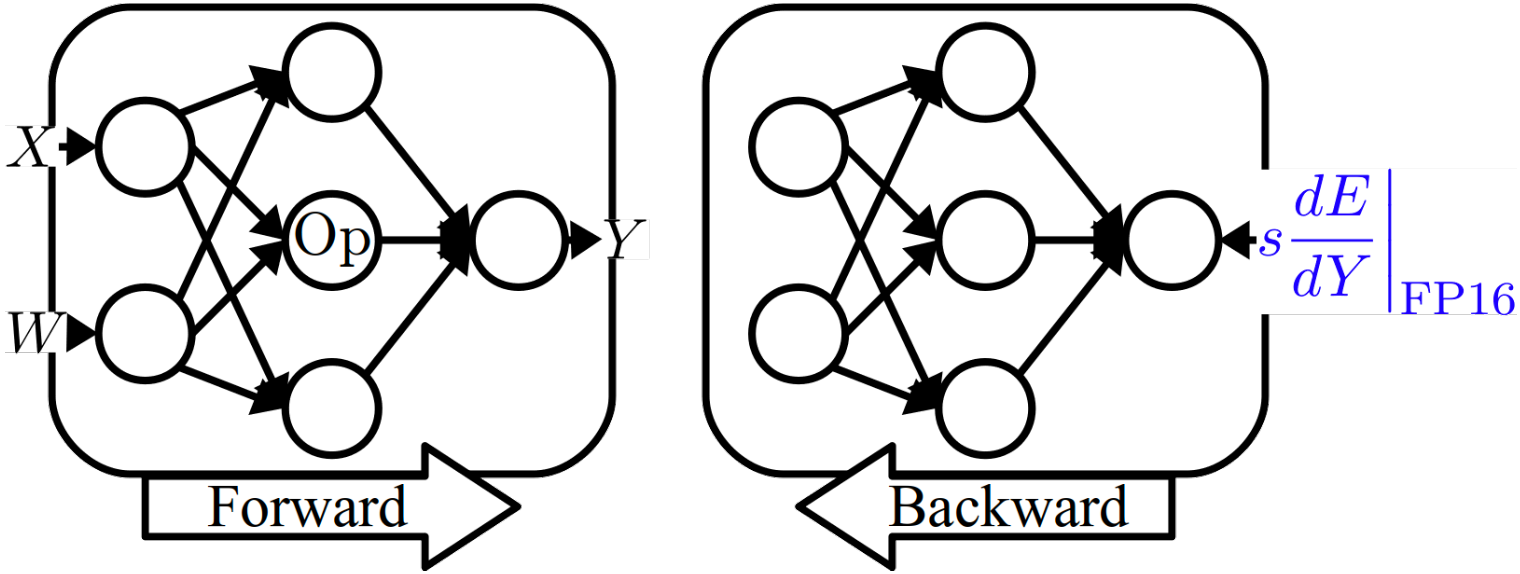
Proof. Gradient Value Distribution of Multibox SSD [1]

Loss Scaling [1]



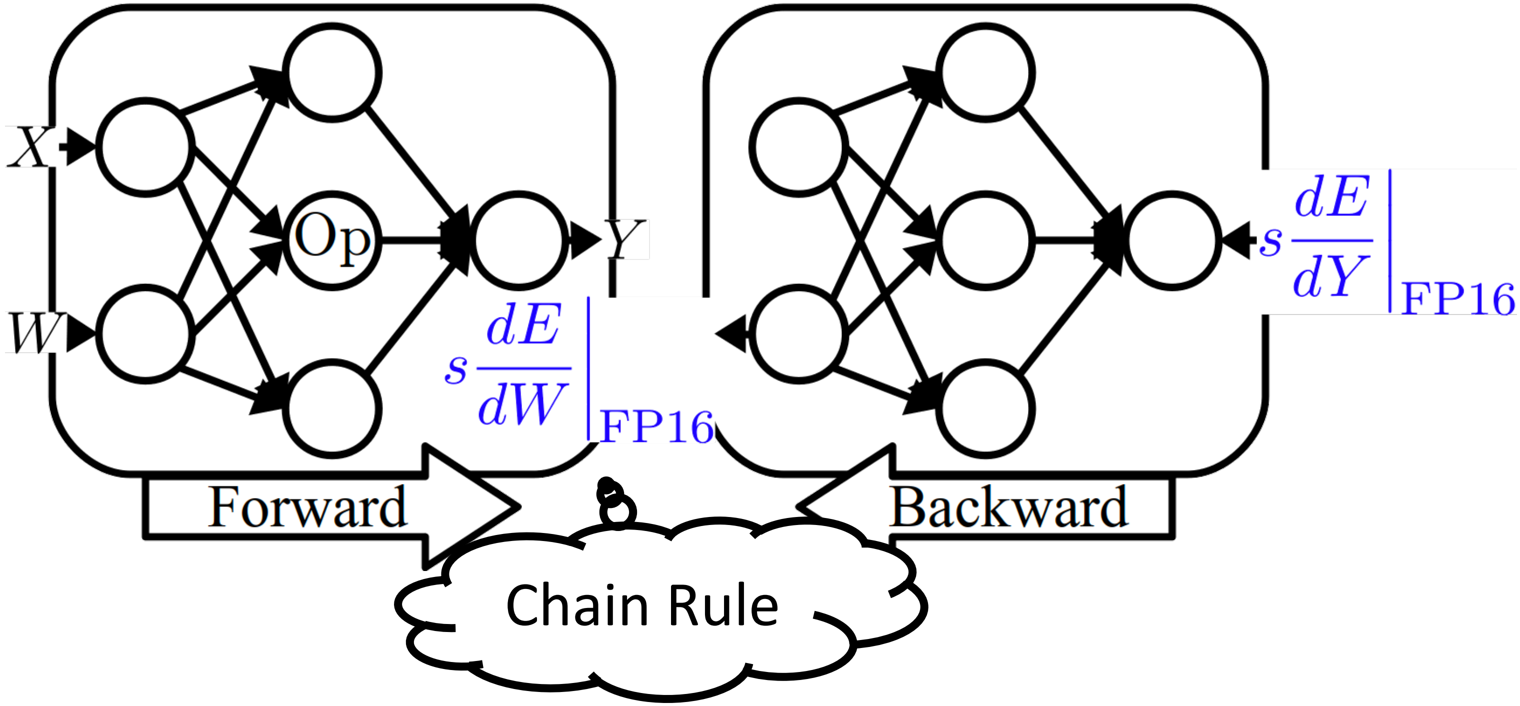
$$W_{\mathbf{FP16}} = W_{\mathbf{FP16}} - \mu \left.\frac{dE}{dW}\right|_{\mathbf{FP16}}$$

Loss Scaling [1]



$$W_{\mathrm{FP16}} = W_{\mathrm{FP16}} - \mu \frac{dE}{dW}\Big|_{\mathrm{FP16}}$$

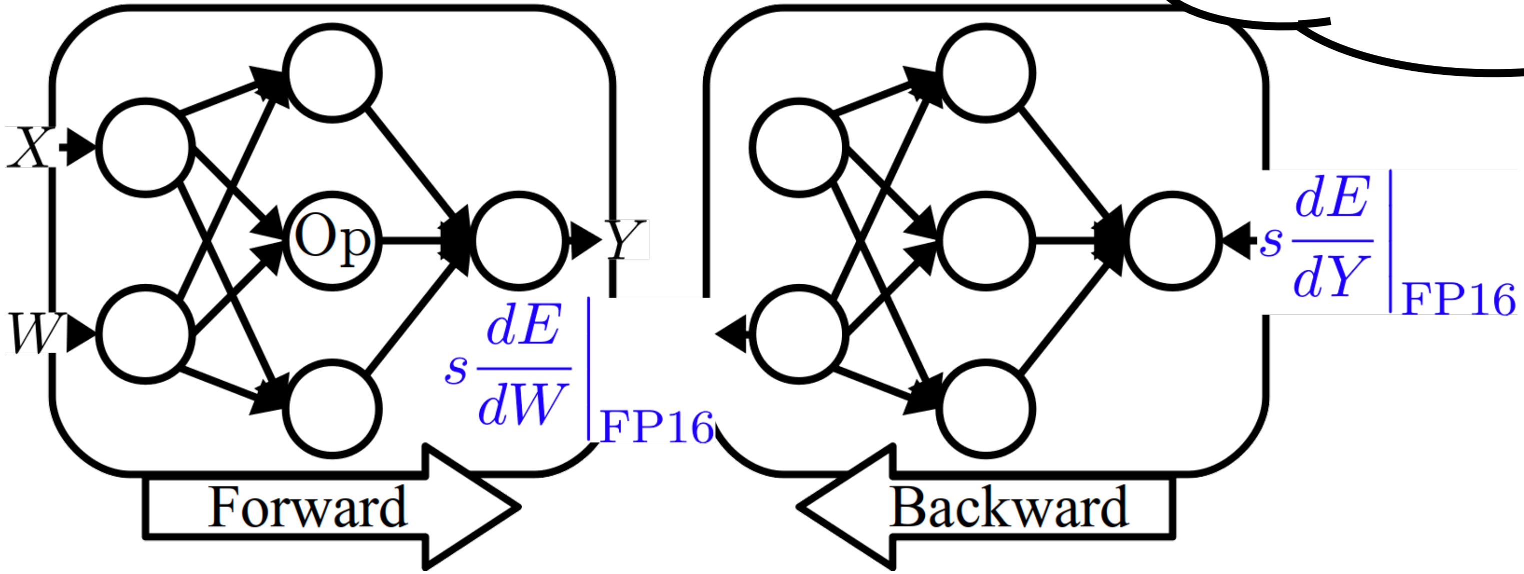
Loss Scaling [1]



$$W_{\mathrm{FP16}} = W_{\mathrm{FP16}} - \mu \frac{dE}{dW}\Big|_{\mathrm{FP16}}$$

Loss Scaling [1]

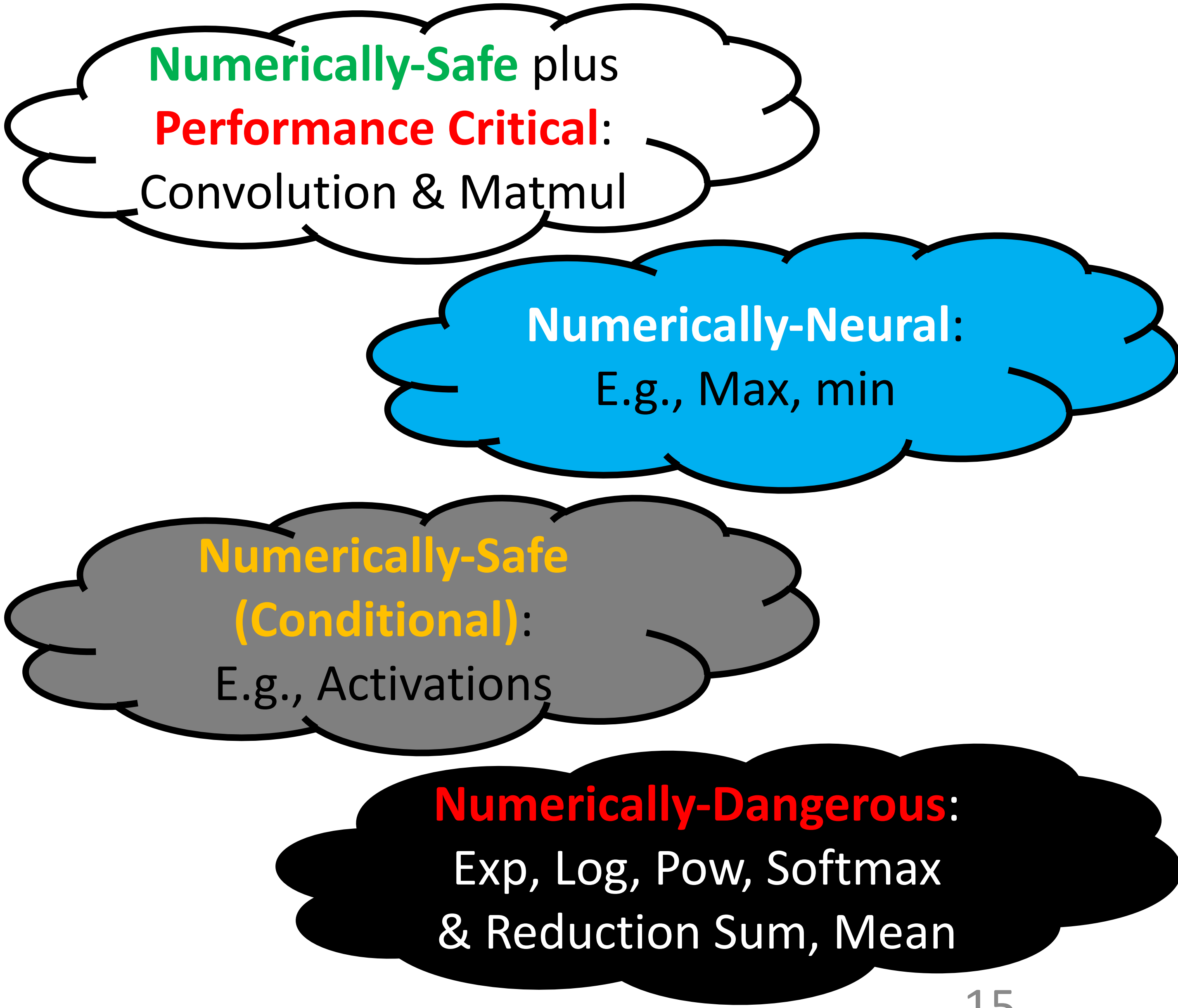
Summary. Scale gradients up during backpropagation & Do weight update in FP32.



$$W_{\mathbf{FP32}} = W_{\mathbf{FP32}} - \mu \frac{1}{s}\left(s \frac{dE}{dW}\Big|_{\mathbf{FP16\to32}} \right)$$

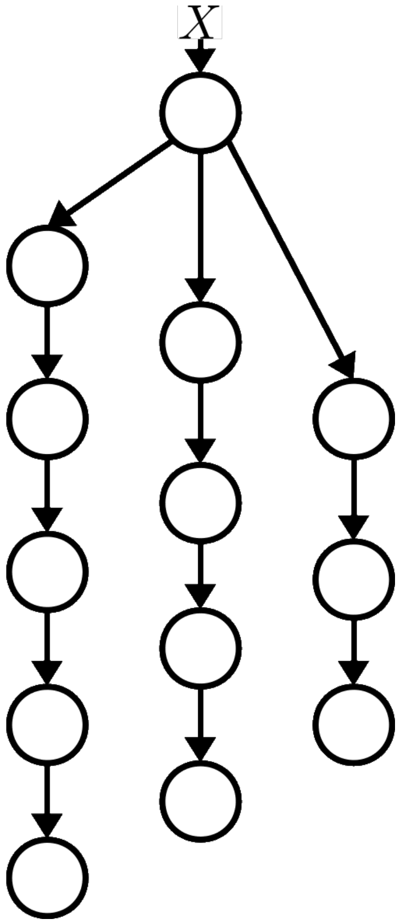
Graph Rewrite [2]

- Categorize operators by their **numerical-safety** level.
  - ▫ White Always in **FP16**
  - ▪ Clear
  - ▪ Grey
    Context-Dependent
  - ▪ Black Always in **FP32**



Graph Rewrite [2]

- Categorize operators by their **numerical-safety** level.
- Rewrite the graph, with the goals below:
  - performance-critical ops are in FP16.
  - numerical-safety is preserved.
  - $\min(\mathrm{CastOps})$



Automatic Mixed Precision

User Instructions

AMP Support (TensorFlow)

- NVIDIA Tensorflow BERT with FP16 support:
  - https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT
- Summary of Major Changes
  - ```
    export TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1
    # -> scripts/run_squad.sh
    # enable automatic graph rewrite
    ```
  - ```
    optimizer = …LossScaleOptimizer(…)
    # -> optimization.py ~L80
    # switch the optimizer
    # to di automatic loss scaling
    ```

2× Speedup
($B_{\mathrm{FP16}} = B_{\mathrm{FP32}}$)
3× Speedup
($\max B_{\mathrm{FP16}}$)

Automatic Mixed Precision
Motivation

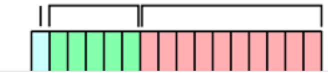
Automatic Mixed Precision
Under the Hood

Automatic Mixed Precision
User Instructions

6

Arithmetic Overflow/Underflow

- Why & When would it happen?
  - **Numerically-Dangerous Ops** [2]
    - Exponential, Logarithmic
    - **Reduction Sum**
  - **Weight Update** [1]
    - Gradients are often too small when compared with the weights.
    - Many are even NOT representable.
- E.g., $1 + 10^{-4} = ?$
  - In FP32, the answer is 1.0001, but in FP16, the answer is **1**.
- *TL'DR* FP16 '+' is **ineffective** if the two operands are different by more than **2k**.



AMP Support (TensorFlow)

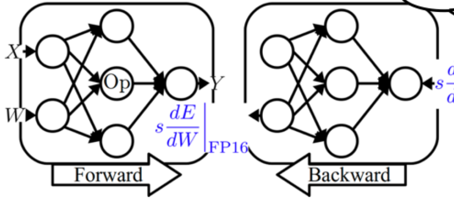
- NVIDIA Tensorflow BERT with FP16 support:
  - https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT
- Summary of Major Changes
  - export TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1
    # -> scripts/run_squad.sh
    # enable automatic graph rewrite
  - optimizer = …LossScaleOptimizer(…)
    # -> optimization.py ~L80
    # switch the optimizer
    # to di automatic loss scaling

$2\times$ Speedup
$(B_{\text{FP16}} = B_{\text{FP32}})$
$3\times$ Speedup
(max B_{FP16})

18

Loss Scaling [1]

Summary. Scale gradients up during backpropagation & Do weight update in FP32.



$s\frac{dE}{dW}\Big|_{\text{FP16}}$ $\quad s\frac{dE}{dY}\Big|_{\text{FP16}}$

$$W_{\text{FP32}} = W_{\text{FP32}} - \mu\frac{1}{s}\left(s\frac{dE}{dW}\Big|_{\text{FP16}\to 32}\right)$$

14

Graph Rewrite [2]

- Categorize operators by their **numerical-safety** level.
  - ▫ White Always in **FP16**
  - ▪ Clear Context-Dependent
  - ▪ Grey
  - ▪ Black Always in **FP32**

Numerically-Safe plus **Performance Critical**: Convolution & Matmul

Numerically-Neural: E.g., Max, min

Numerically-Safe (Conditional): E.g., Activations

Numerically-Dangerous: Exp, Log, Pow, Softmax & Reduction Sum, Mean

15

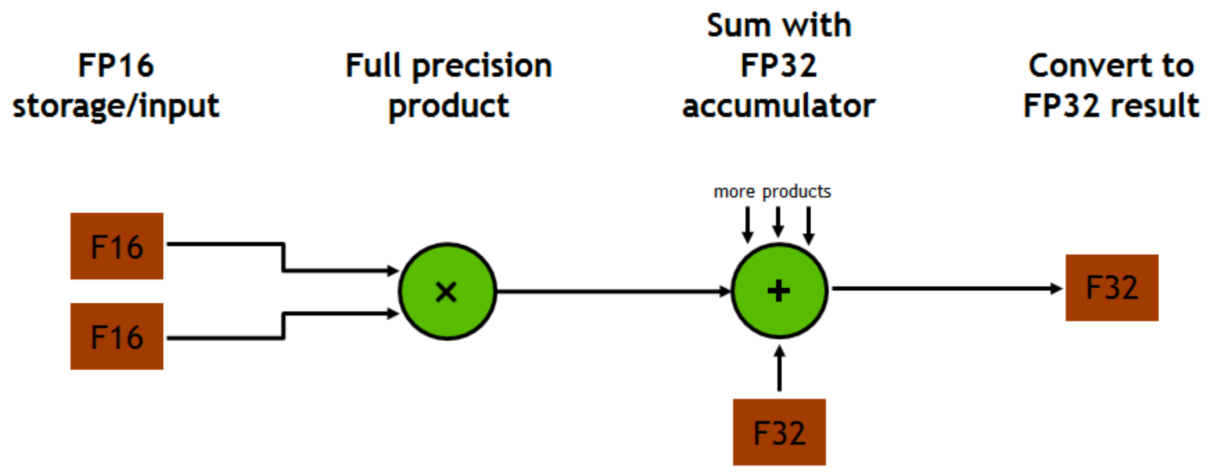
Automatic Mixed Precision

Backup

FAQ

- **Q**: Does Matmul involve reduction sum? Why can it be done in FP16?
- **A**: In tensor core FP16 MAC (Multiply-Accumulate) unit, the accumulation is always done in **full precision**, which avoids the problem of arithmetic underflow.

Reference: https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/



FAQ (Cont.)

- **Q**: How is the loss scaling factor determined?
- **A**: The loss scaling factor s is determined **automatically**.
  - *Key Idea*. Loss scaling factor should be as **large** as possible so long as numerical overflow does not happen.
  - To start with, s is initialized with a **large** number (by default, $2^{15} \approx 3 \times 10^{5}$).
    - *A loss scale that is too high gets lowered far more quickly than a loss scale that is to low gets raised.*
  - If an overflow happens, the current iteration is discarded, and s is decreased (usually halved).
  - After certain number of steady iterations (by default, $2\mathrm{k}$), s is doubled.

Reference: https://www.tensorflow.org/api_docs/python/tf/train/experimental/DynamicLossScale
https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#training

FAQ (Cont.)

- **Q**: Is AMP supported on other frameworks?
- **A**: NVIDIA people have been working hard to port the idea of AMP onto more SOTA frameworks, please check the link below for the support status on your favorite framework:

https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#framework

FAQ (Cont.)

- **Q**: Is AMP supported on PyTorch?
- **A**: The current Megatron implementation already supports FP16.
  It only converts BatchNorm layers to FP32.
  However, according to NVIDIA developer Michael Carilli,
  it is recommended to use the PyTorch extension ***Apex***,
  which is more generic and transparent to the frontend users.

Reference: https://discuss.pytorch.org/t/training-with-half-precision/11815/10

Apex User Instructions

Reference: https://github.com/NVIDIA/apex/tree/master/examples/imagenet
https://nvidia.github.io/apex/amp.html#apex.amp.initialize

- Install *Apex*:

```
git clone https://github.com/NVIDIA/apex
cd apex
pip install -v --no-cache-dir                     \
               --global-option="--cpp_ext"    \
               --global-option="--cuda_ext" ./
```

- Add the following lines to your code:

```
# After the model and optimizer construction,
model, optimizer = amp.initialize(model, optimizer, …)
# loss.backward() changed to:
with amp.scale_loss(loss, optimizer) \
        as scaled_loss:
    scaled_loss.backward()
```