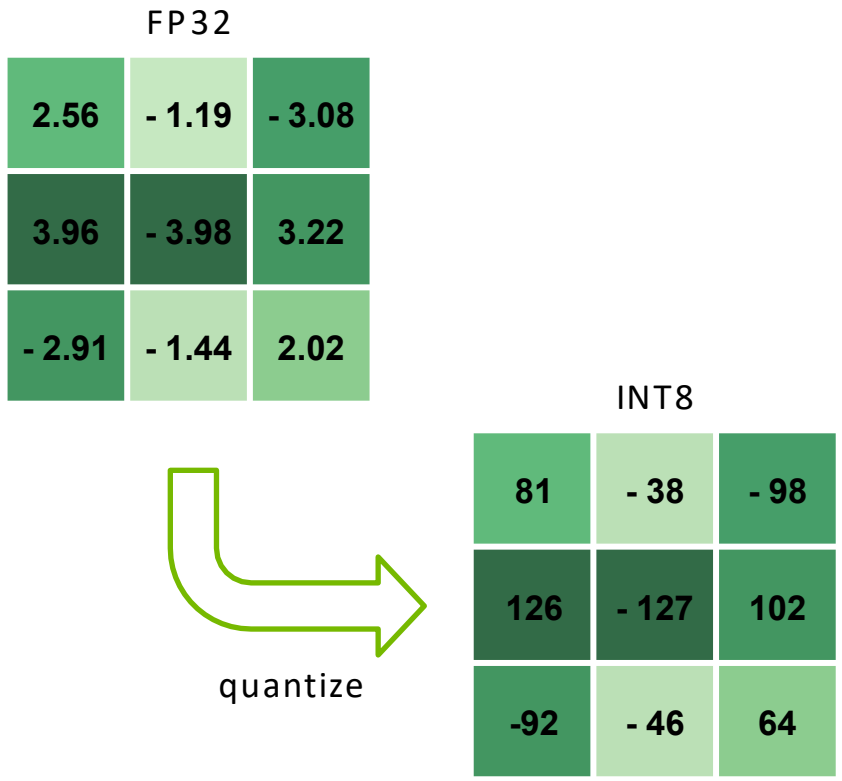
QUANTIZATION

Inference with Lower Precision

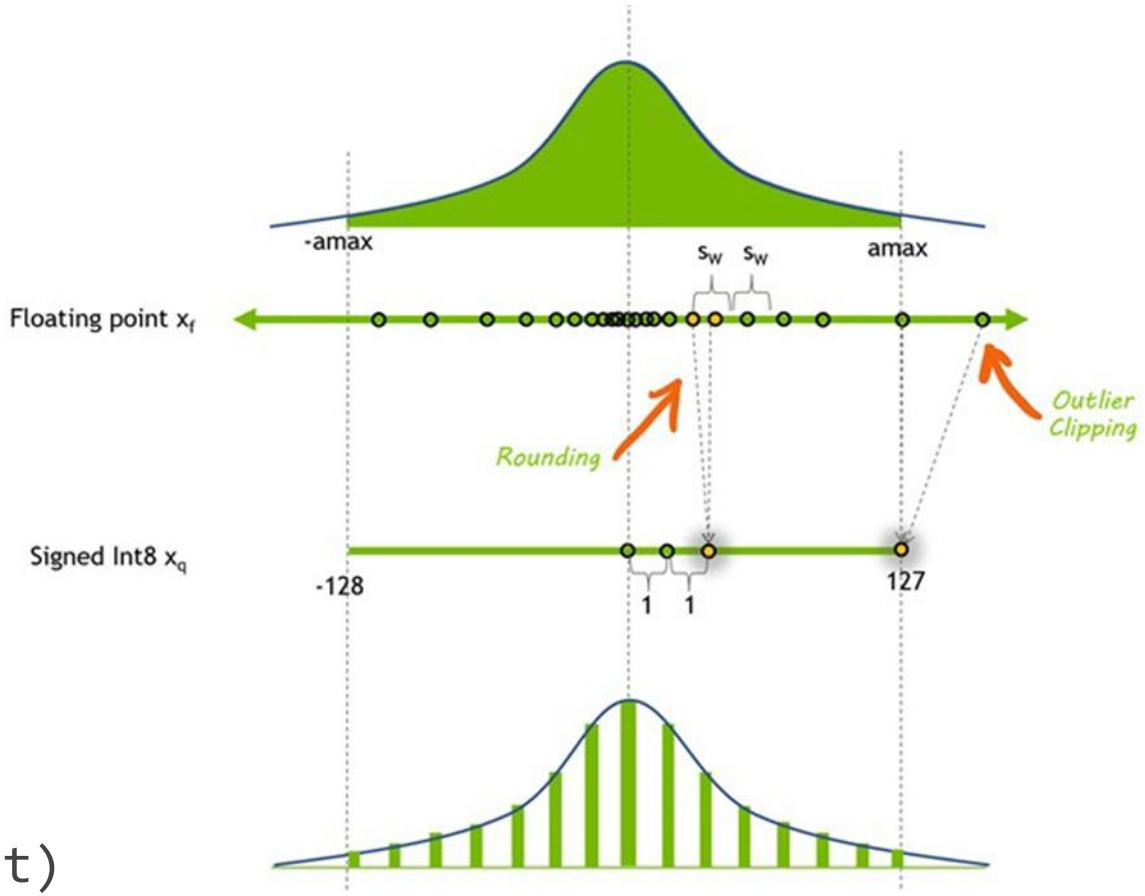
- Most models are trained in FP32 / FP16 to take advantage of a wider range of numbers.
- Reducing precision reduces compute, memory, and power.
  - NVIDIA GPUs employ faster and cheaper 8-bit Tensor Cores for computation.
  - 32-bit floats to 8-bit integers results in 4x memory reduction and about 2-4x throughput improvement.



https://developer.nvidia.com/blog/improving-int8-accuracy-using-quantization-aware-training-and-tao-toolkit/

From High Precision to Low Precision

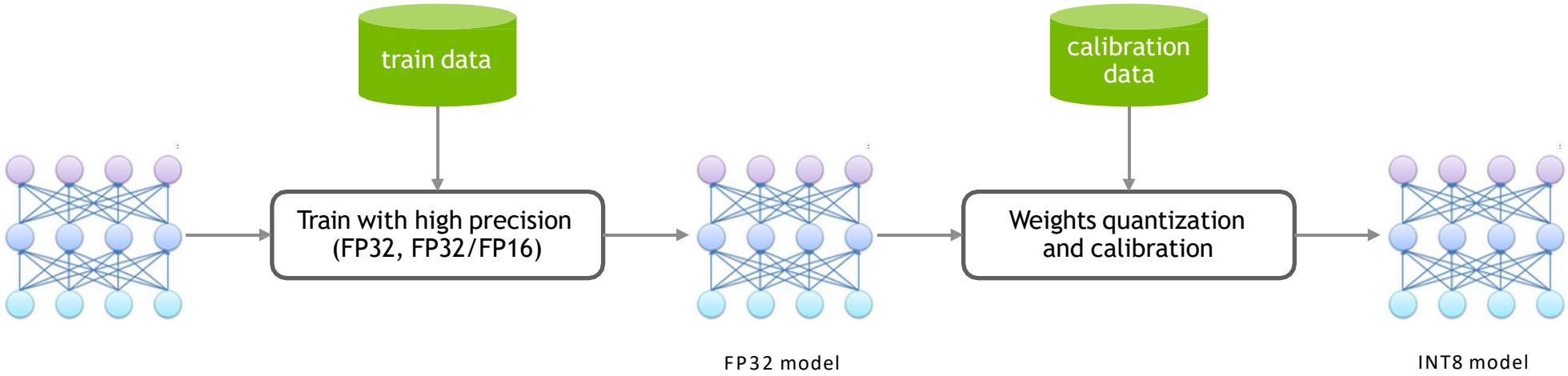
- FP32 represent ~4M values in range [-3.4e38, 3.4e38] and about half values in [-1, 1].
- Int8 can represent only 256 values.
- We need a mapping process to convert from FP32 to INT8 representation.



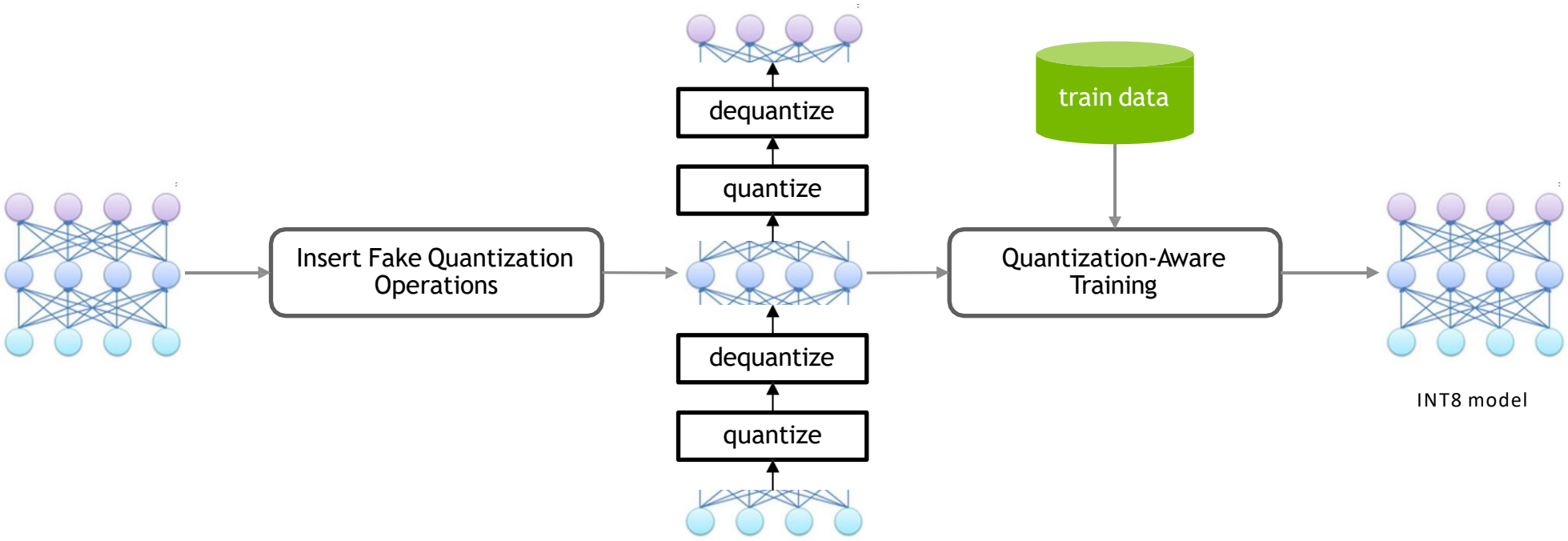

```
scale = threshold / pow(2, bit - sign_bit)
QUANT_VALUE * scale = REAL_VALUE
REAL_VALUE / scale = QUANT_VALUE
```

$$x_q = Clip(Round(x_f/scale))$$

Post-Training Quantization (PTQ)



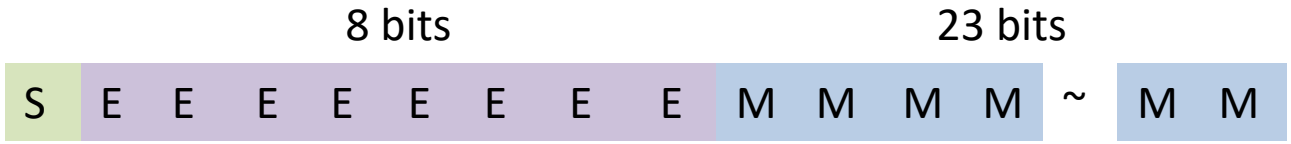
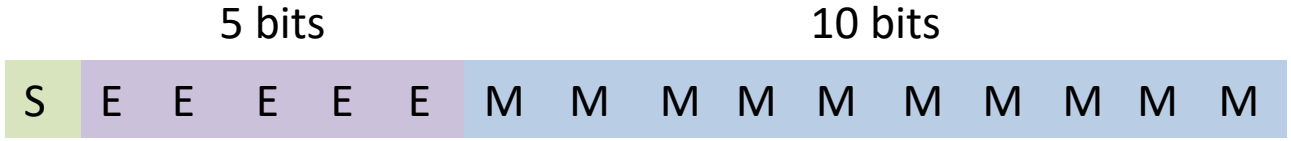
Quantization-Aware-Training (QAT)



Outline

- Basics of quantization
  - Quantize to which data format?
  - How to quantize?
  - When to quantize?
    - Post-training quantization: scalar or vector (deep compression)
    - Quantization aware training: Quant+training or differentiable threshold
- Advanced quantization topics
  - Low bit precision quantization
    - 2-bit
    - Binary/ternary
- Summary

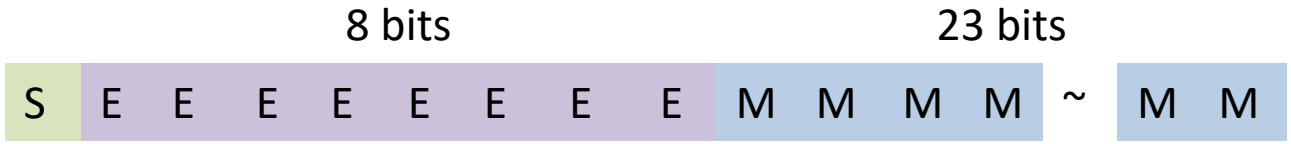
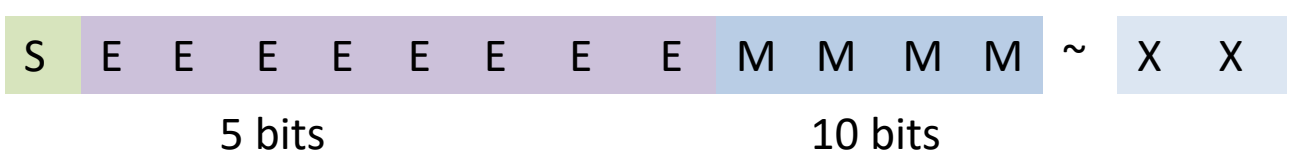
Typical data formats used in DNNs

Number formats		Dynamic Range	Relative Precision
float32		$1e^{-38}$ to $3e^{38}$	$6e^{-6}\%$
float16		$6e^{-5}$ to $6e^{5}$	0.05%

Floating-point numbers have (roughly) constant relative precision

Float32 can represent a very wide dynamic range of numbers!

Typical data formats used in DNNs

Number formats		Dynamic Range	Relative Precision
float32		$1e^{-38}$ to $3e^{38}$	$6e^{-6}\%$
tensorfloat32		$1e^{-38}$ to $3e^{38}$	0.05%
float16		$6e^{-5}$ to $6e^{5}$	0.05%

Tensorfloat32 (Nvidia) uses the same 10-bit mantissa as IEEE float16, shown to be sufficient for DNNs.
And it adopts the same 8-bit exponent as IEEE float32 so it can support the same numeric range.

Typical data formats used in DNNs

Number formats		Dynamic Range	Relative Precision
float32	8 bits, 23 bits: S E E E E E E E E M M M M ~ M M	$1e^{-38}$ to $3e^{38}$	$6e^{-6}\%$
tensorfloat32	8 bits, 10 bits: S E E E E E E E E M M M M ~ X X	$1e^{-38}$ to $3e^{38}$	0.05%
float16	5 bits, 10 bits: S E E E E E M M M M M M M M M M	$6e^{-5}$ to $6e^{5}$	0.05%
bfloat16	8 bits, 7 bits: S E E E E E E E E M M M M M M M	$1e^{-38}$ to $3e^{38}$	0.4%

Google introduced bfloat16 on the observation that “neural networks are far more sensitive to the size of the exponent than that of the mantissa”

Moreover: the physical size of a bfloat16 multiplier is roughly two times smaller than that of float16

Typical data formats used in DNNs

Number formats		Dynamic Range	Relative Precision
float32	8 bits, 23 bits: S E E E E E E E E M M M M ~ M M	$1e^{-38}$ to $3e^{38}$	$6e^{-6}\%$
tensorfloat32	8 bits, 10 bits: S E E E E E E E E M M M M ~ X X	$1e^{-38}$ to $3e^{38}$	0.05%
float16	5 bits, 10 bits: S E E E E E M M M M M M M M M M	$6e^{-5}$ to $6e^{5}$	0.05%
bfloat16	8 bits, 7 bits: S E E E E E E E E M M M M M M M	$1e^{-38}$ to $3e^{38}$	0.4%
integer16	15 bits: S M M M M M M M M M M M M M M M	1 to $3e^{4}$	1
integer8	7 bits: S M M M M M M M	1 to 127	1

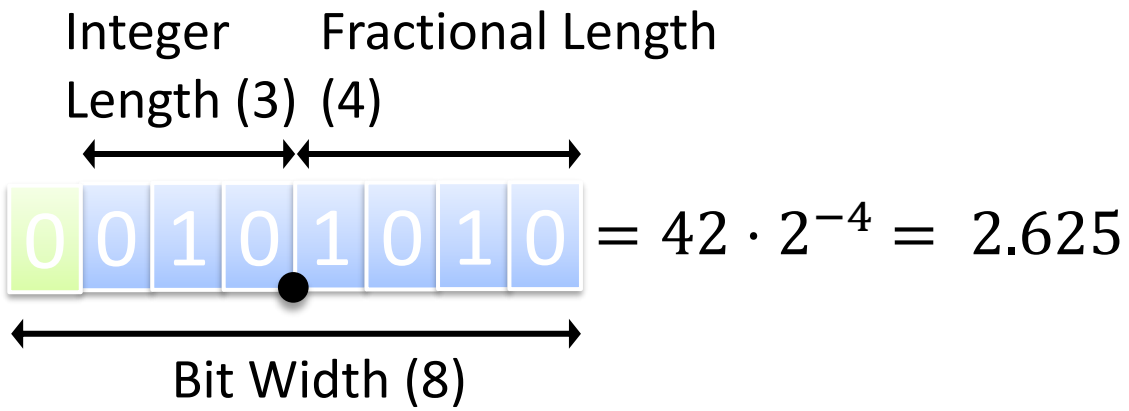
Integers are cheap, but their relative precision *fluctuates*

Floats are expensive, but their relative precision is *constant*

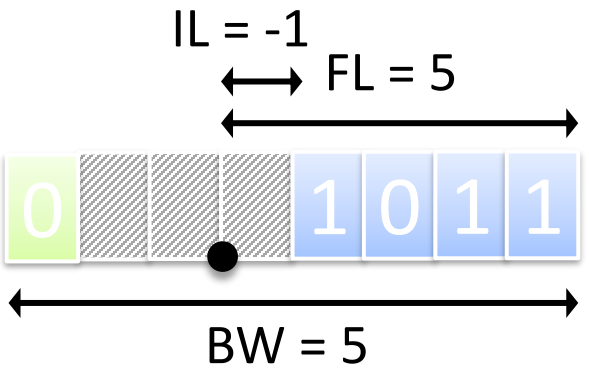
Range problem of integers can be solved by the fixed-point number format

Fixed-point number format

- Bit-width = Fractional Length + Integer Length + 1 (Sign)
- Approximate real value $\boldsymbol{x}$ by computing $\hat{\boldsymbol{x}} = \mathrm{Round}\left(\boldsymbol{x} \cdot 2^{\mathrm{FL}}\right) \cdot 2^{-\mathrm{FL}}$
- Representable range: $-2^{\mathrm{IL}} \leq \hat{\boldsymbol{x}} \leq 2^{\mathrm{IL}} - 2^{-\mathrm{FL}}$ with step size $2^{-\mathrm{FL}}$

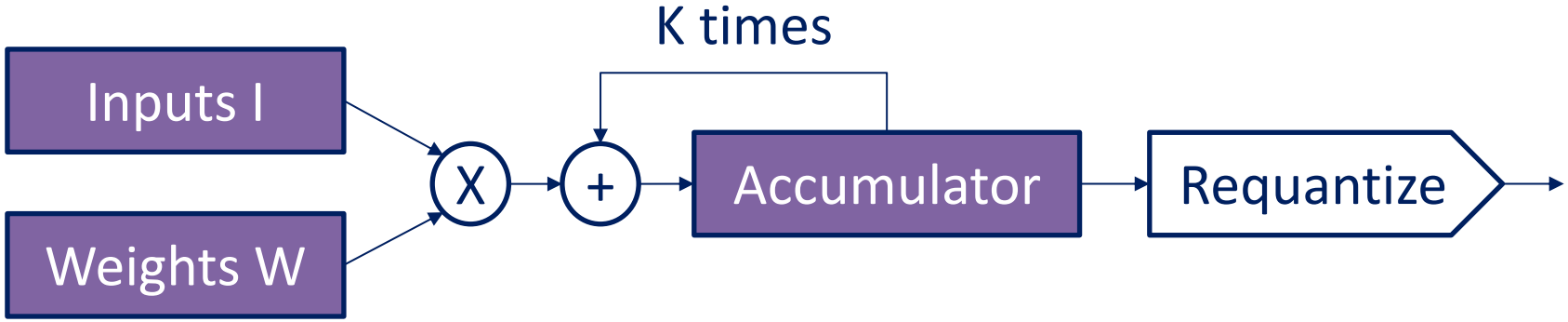


$= 42 \cdot 2^{-4} = 2.625$



$= 11 \cdot 2^{-5} = 0.34375$

Example: integer-based MAC data path



$$I_{1,1}W_{1,1} + I_{1,2}W_{2,1} + I_{1,3}W_{3,1} + \cdots$$

Operands' width = BW_I or BW_w

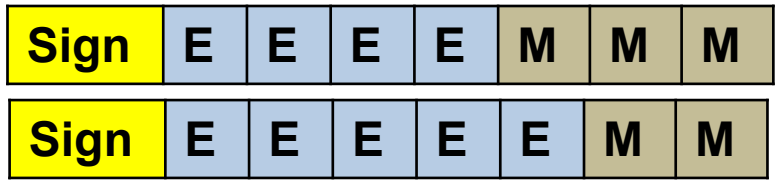
Product's width = $BW_I + BW_W - 1^*$

Accumulation width = $(BW_I + BW_W - 1) + \lceil \log_2 K \rceil$

Input/output precision between layers is aligned using a scale (requantize)

* Note: '– 1' only valid if integer min x min case is ignored.

FP8 Formats (Nvidia/Intel/ARM)



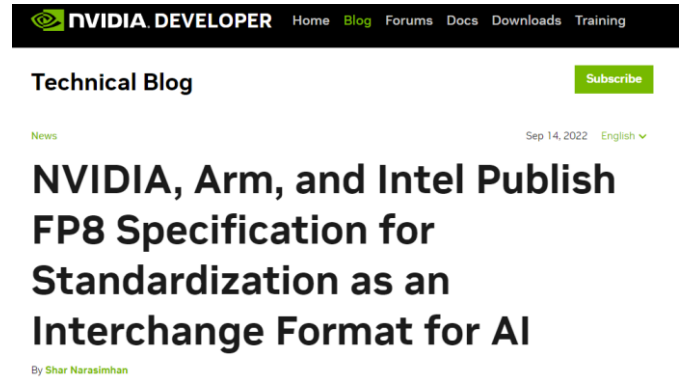


Table 1: Details of FP8 Binary Formats

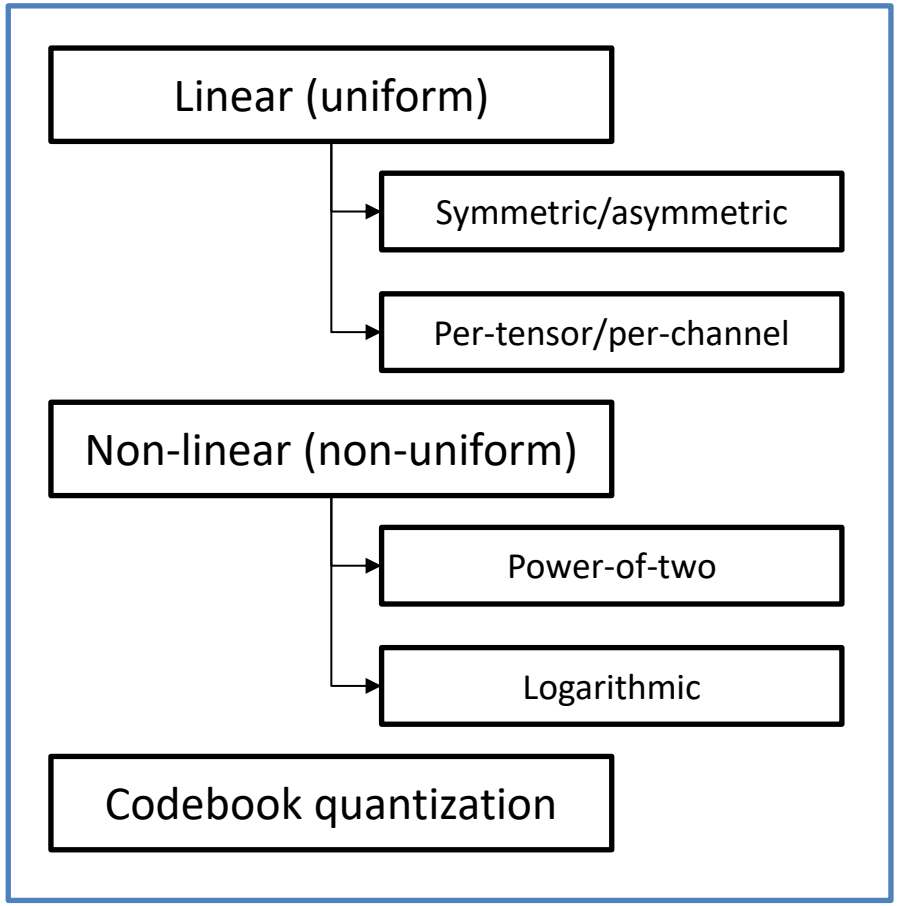
	E4M3	E5M2
Exponent bias	7	15
Infinities	N/A	$S.11111.00_2$
NaN	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal	$S.1111.110_2 = 1.75 * 2^8 = 448$	$S.11110.11_2 = 1.75 * 2^{15} = 57,344$
Min normal	$S.0001.000_2 = 2^{-6}$	$S.00001.00_2 = 2^{-14}$
Max subnorm	$S.0000.111_2 = 0.875 * 2^{-6}$	$S.00000.11_2 = 0.75 * 2^{-14}$
Min subnorm	$S.0000.001_2 = 2^{-9}$	$S.00000.01_2 = 2^{-16}$

FP8 FORMATS FOR DEEP LEARNING https://github.com/IntelLabs/FP8-Emulation-Toolkit

Typical data formats used in DNNs

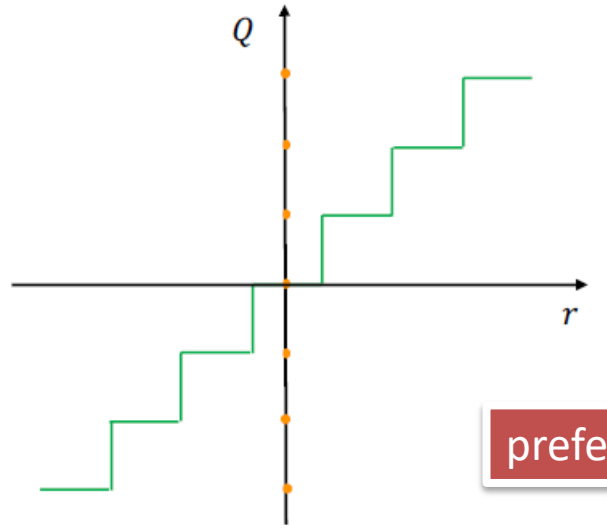
Number formats		Dynamic Range	Relative Precision
float32	8 bits, 23 bits: S E E E E E E E E M M M M ~ M M	$1e^{-38}$ to $3e^{38}$	$6e^{-6}\%$
tensorfloat32	8 bits, 10 bits: S E E E E E E E E M M M M ~ X X	$1e^{-38}$ to $3e^{38}$	0.05%
float16	5 bits, 10 bits: S E E E E E M M M M M M M M M M	$6e^{-5}$ to $6e^{5}$	0.05%
bfloat16	8 bits, 7 bits: S E E E E E E E E M M M M M M M	$1e^{-38}$ to $3e^{38}$	0.4%
FP8 E4M3	S E E E E M M M		
FP8 E5M2	S E E E E E M M		

How to quantize?

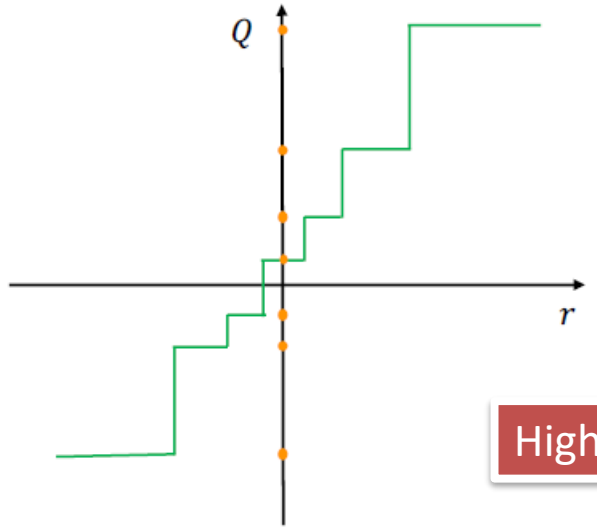


Quantization: Uniform vs Nonuniform

- Uniform quantization
  - Linear quantization
    - Arm/x86/nvGPU2都支持8-bit, TensorCore支持4bit
    - 引入額外的quantizer，可用SIMD加速
  - Binarization
    - XNOR + popcount (32x speedup v.s FP32)
    - 引入額外的quantizer，可用SIMD加速
- Nonuniform quantization
  - Logarithm quantization
    - 可將乘法轉為加法，加法變查表



preferred



Higher accuracy

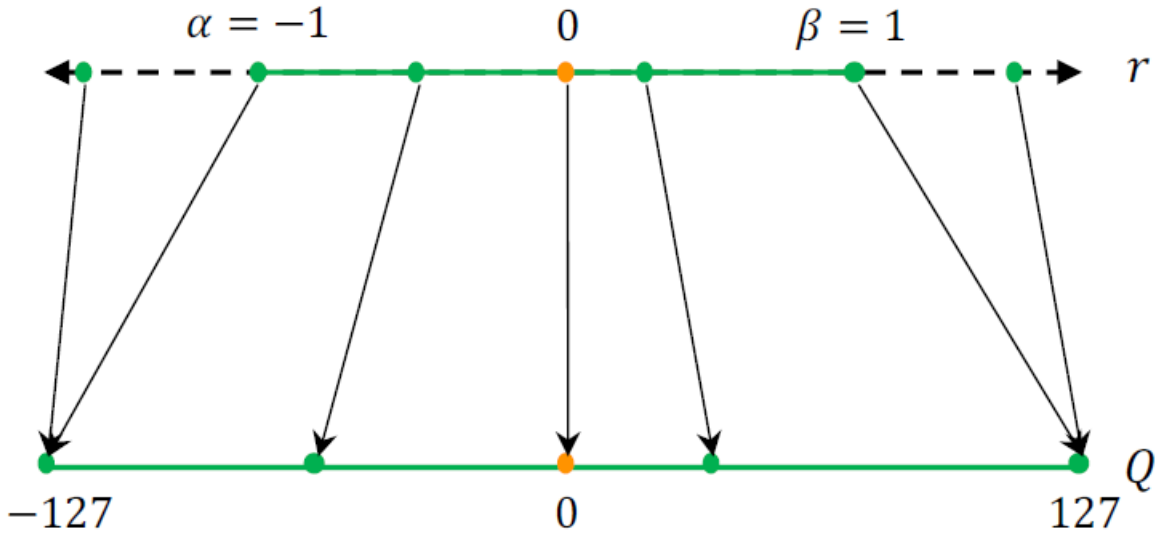
Quantization: Symmetric vs Nonsymmetric

- Symmetric quantization

$$x_q = clip(round\left(\frac{x}{s}\right), n, p)$$

preferred

  - -α = β
  - 量化參數: scale

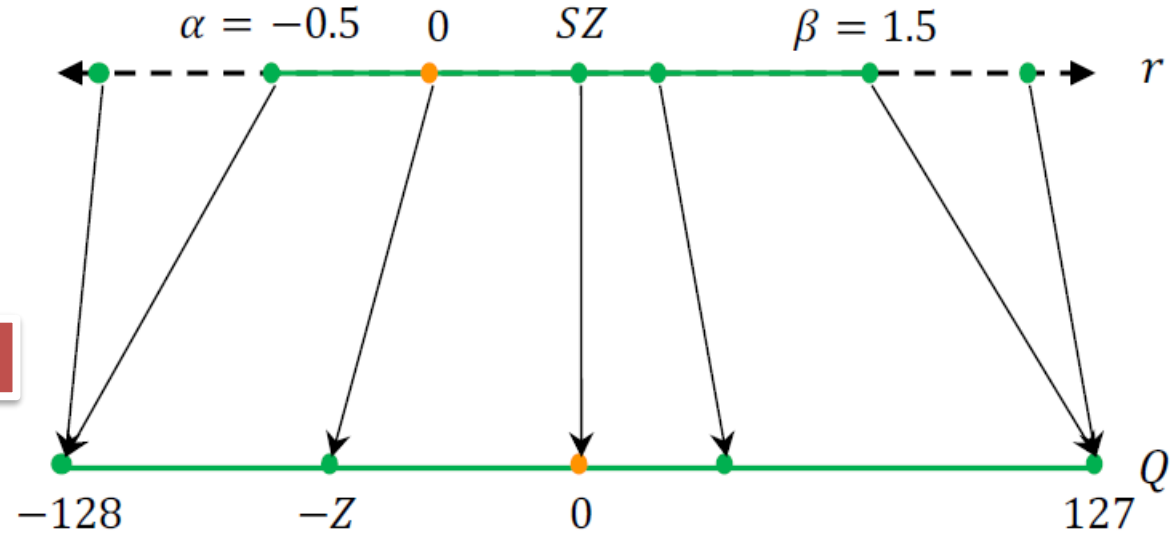


- Nonsymmetric quantization

$$x_q = clip(round\left(\frac{x}{s} - z\right), n, p)$$

Higher accuracy

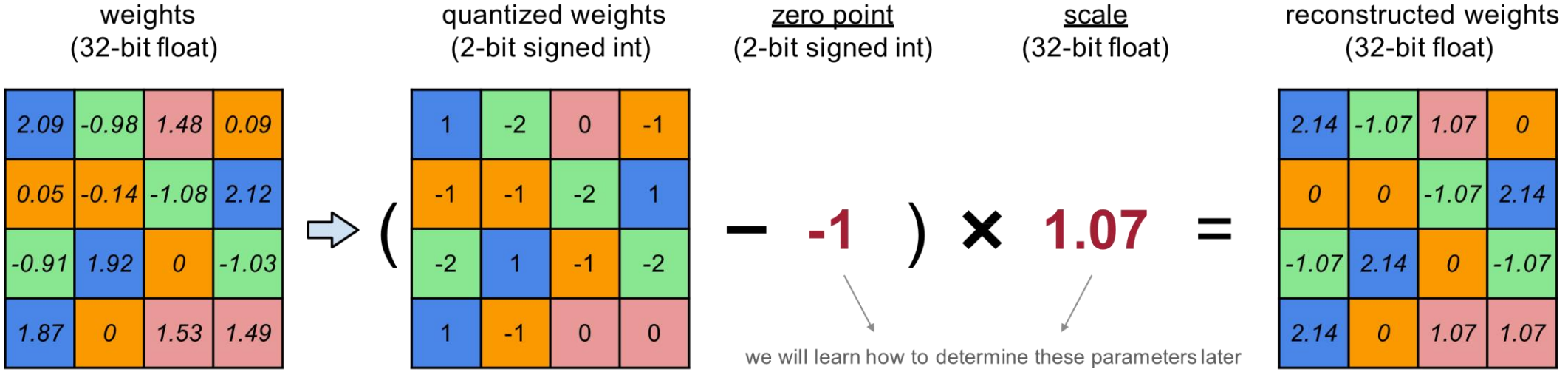
  - -α != β
  - 量化參數: scale, zero_point



Linear Quantization

$$x_q = clip(round\left(\frac{x}{S} - z\right), n, p)$$

- Linear Quantization is an affine mapping of integers to real numbers r = S(q − Z)

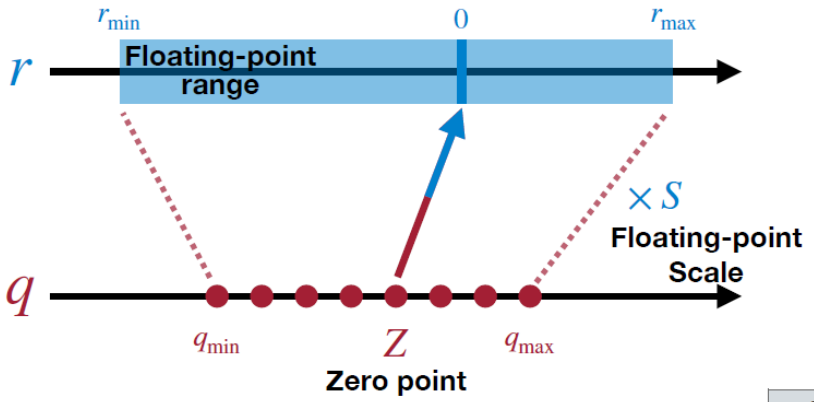


Binary	Decimal
01	1
00	0
11	-1
10	-2

quantization error

-0.05	0.09	0.41	0.09
0.05	-0.14	-0.01	-0.02
0.16	-0.22	0	0.04
-0.27	0	0.46	0.42

Scale of Linear Quantization



Binary	Decimal
01	1
00	0
11	-1
10	-2

$$r_{max} = S\left(q_{max} - Z\right)$$

$$r_{min} = S\left(q_{min} - Z\right)$$

$$\downarrow$$

$$r_{max} - r_{min} = S\left(q_{max} - q_{min}\right)$$

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

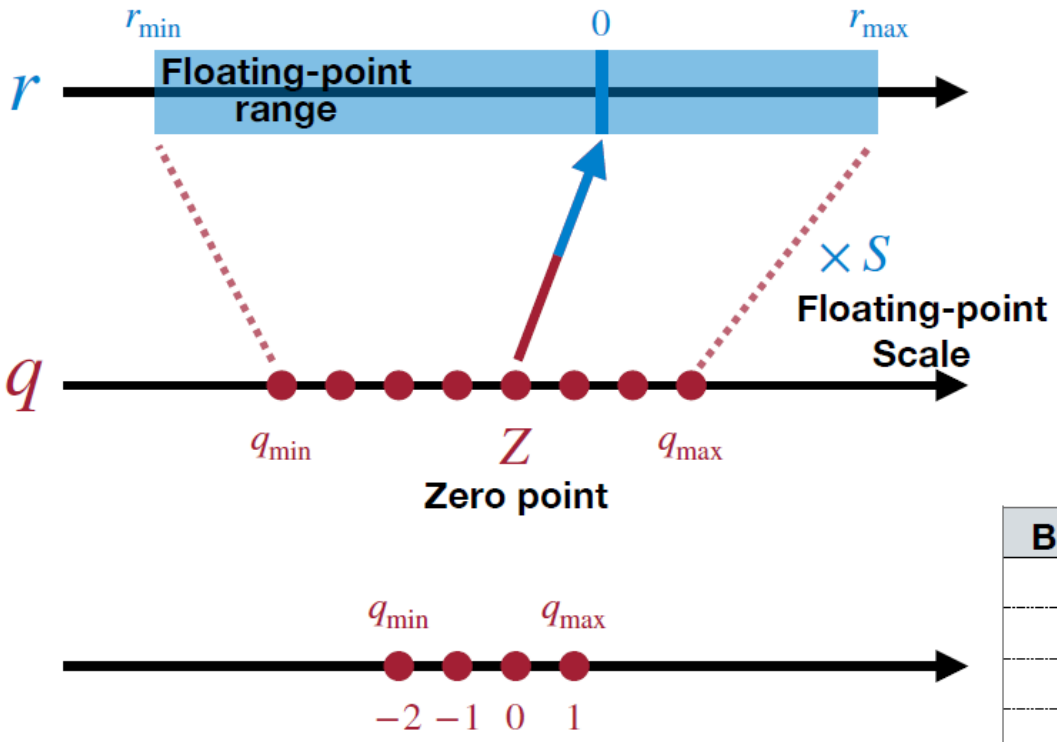
2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

$$= \frac{2.12 - (-1.08)}{1 - (-2)}$$

$$= 1.07$$

Zero Point of Linear Quantization



$$r_{min} = S\,(q_{min} - Z)$$

$$\downarrow$$

$$Z = q_{min} - \frac{r_{min}}{S}$$

$$\downarrow$$

$$Z = \text{round}\left(q_{min} - \frac{r_{min}}{S}\right)$$

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

$$Z = q_{min} - \frac{r_{min}}{S}$$

$$= \text{round}(-2 - \frac{-1.08}{1.07})$$

$$= -1$$

Binary	Decimal
01	1
00	0
11	-1
10	-2

Linear Quantized Matrix Multiplication

- Consider the following matrix multiplication.

$$\mathbf{Y} = \mathbf{W}\mathbf{X}$$

$$S_{\mathbf{Y}}\left(\mathbf{q}_{\mathbf{Y}} - Z_{\mathbf{Y}}\right) = S_{\mathbf{W}}\left(\mathbf{q}_{\mathbf{W}} - Z_{\mathbf{W}}\right) \cdot S_{\mathbf{X}}\left(\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{X}}\right)$$

$$\mathbf{q}_{\mathbf{Y}} = \frac{S_{\mathbf{W}}S_{\mathbf{X}}}{S_{\mathbf{Y}}}\left(\mathbf{q}_{\mathbf{W}} - Z_{\mathbf{W}}\right)\left(\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{X}}\right) + Z_{\mathbf{Y}}$$

$$\mathbf{q}_{\mathbf{Y}} = \frac{S_{\mathbf{W}}S_{\mathbf{X}}}{S_{\mathbf{Y}}}\left(\mathbf{q}_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{X}}\mathbf{q}_{\mathbf{W}} + Z_{\mathbf{W}}Z_{\mathbf{X}}\right) + Z_{\mathbf{Y}}$$

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [Jacob et al., CVPR 2018]

$$\mathbf{Y} = \mathbf{WX}$$

$$q_Y = \frac{S_W S_X}{S_Y}\left(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X\right) + Z_Y$$

Precompute

Rescale to *N*-bit Integer

N-bit Integer Multiplication
32-bit Integer Addition/Subtraction

N-bit Integer Addition

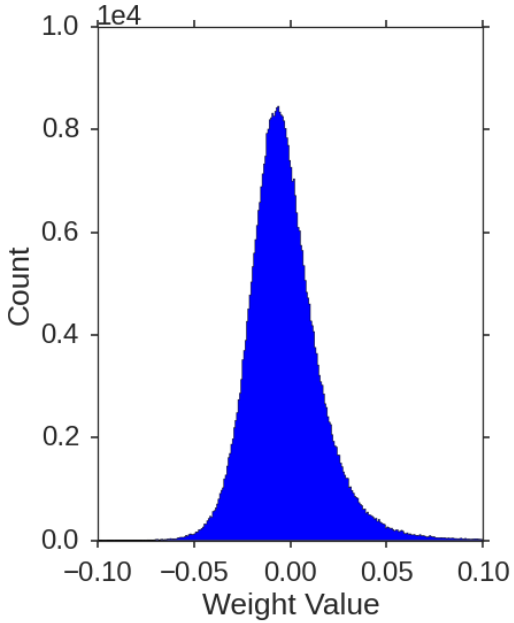
$Z_W = 0?$

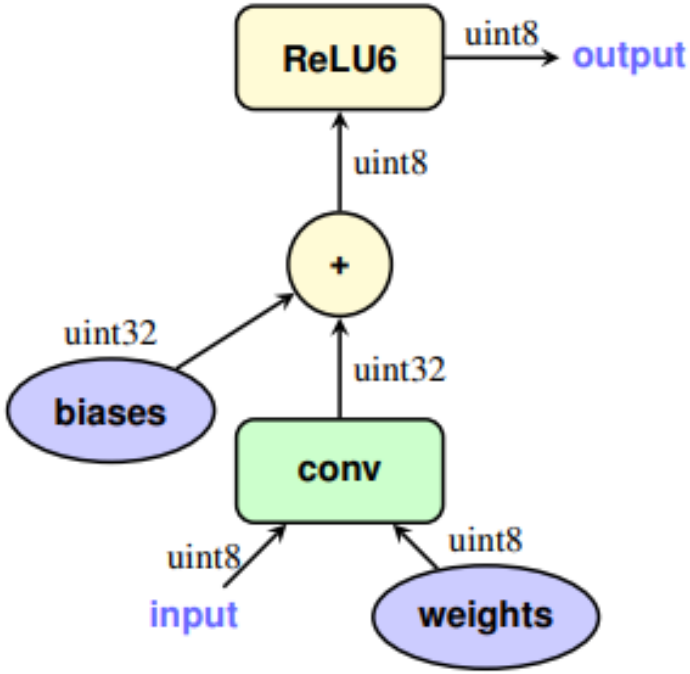
Empirically, the scale $\frac{S_W S_X}{S_Y}$ is always in the interval (0, 1).

Fixed-point Multiplication

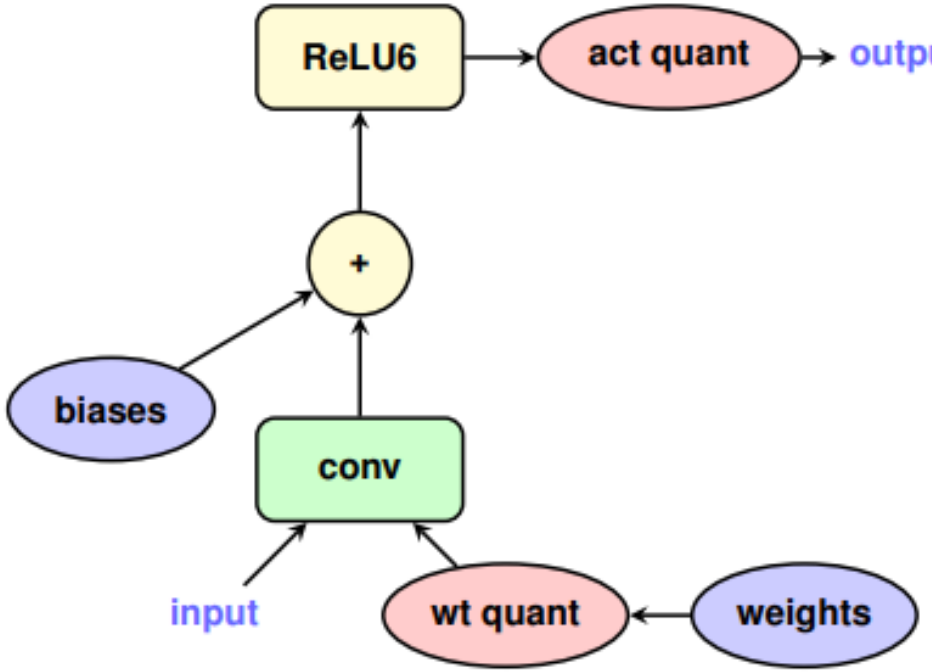
$$\frac{S_W S_X}{S_Y} = 2^{-n} M_0, \quad \text{where} \quad M_0 \in [0.5, 1)$$

Bit Shift

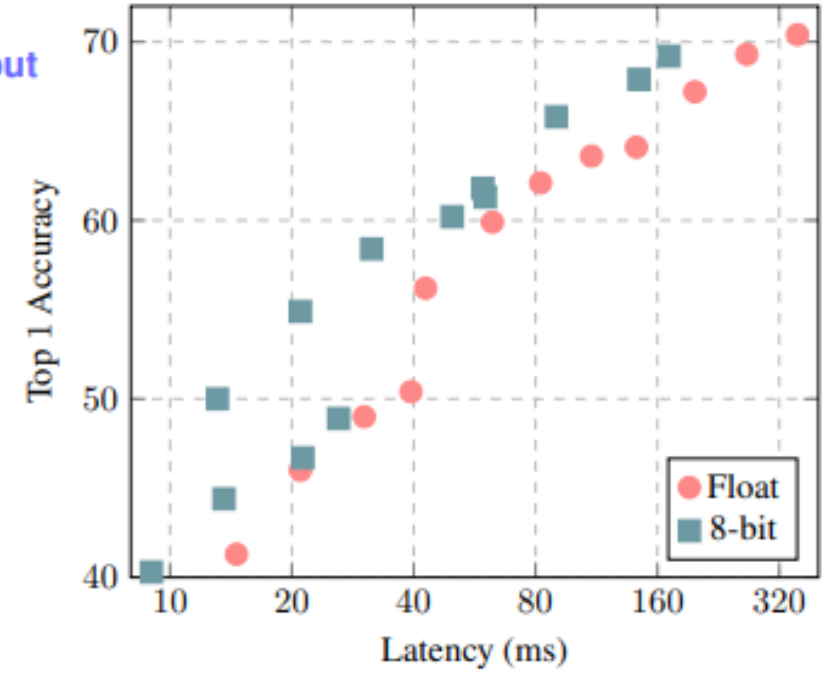




(a) Integer-arithmetic-only inference

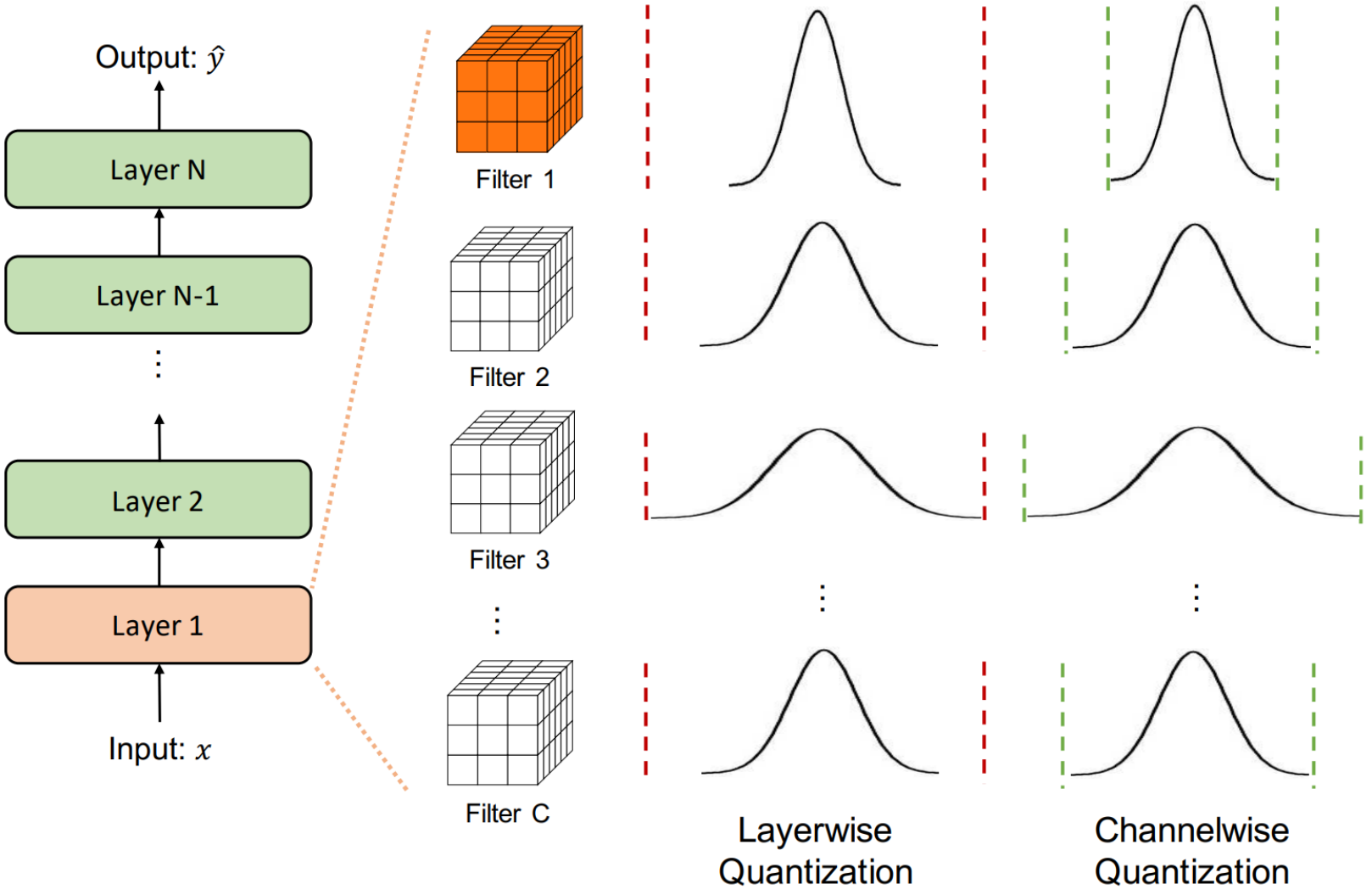


(b) Training with simulated quantization



(c) ImageNet latency-vs-accuracy tradeoff

Per-tensor (Whole Model) vs per-channel



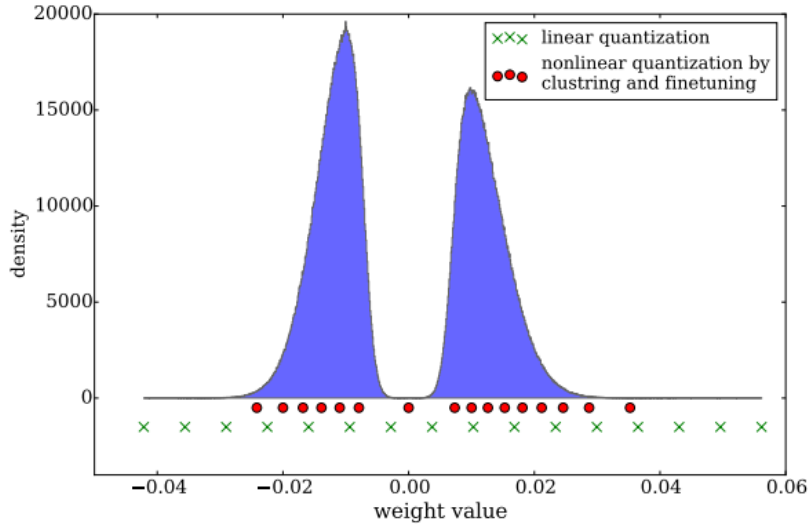
Per-channel is useful when channels have different ranges

Note: per-channel quantization does not require special hardware when applied to **weights**

Image source: A Survey of Quantization Methods for Efficient Neural Network Inference – A. Gholami et al.

Codebook quantization

- Useful when distribution of values is not normally distributed
- Quantized values (codewords) are represented by a codebook
- Codewords are selected using a clustering algorithm or training process
- Codewords may be scalars or vectors
- Note: requires hardware support for lookup

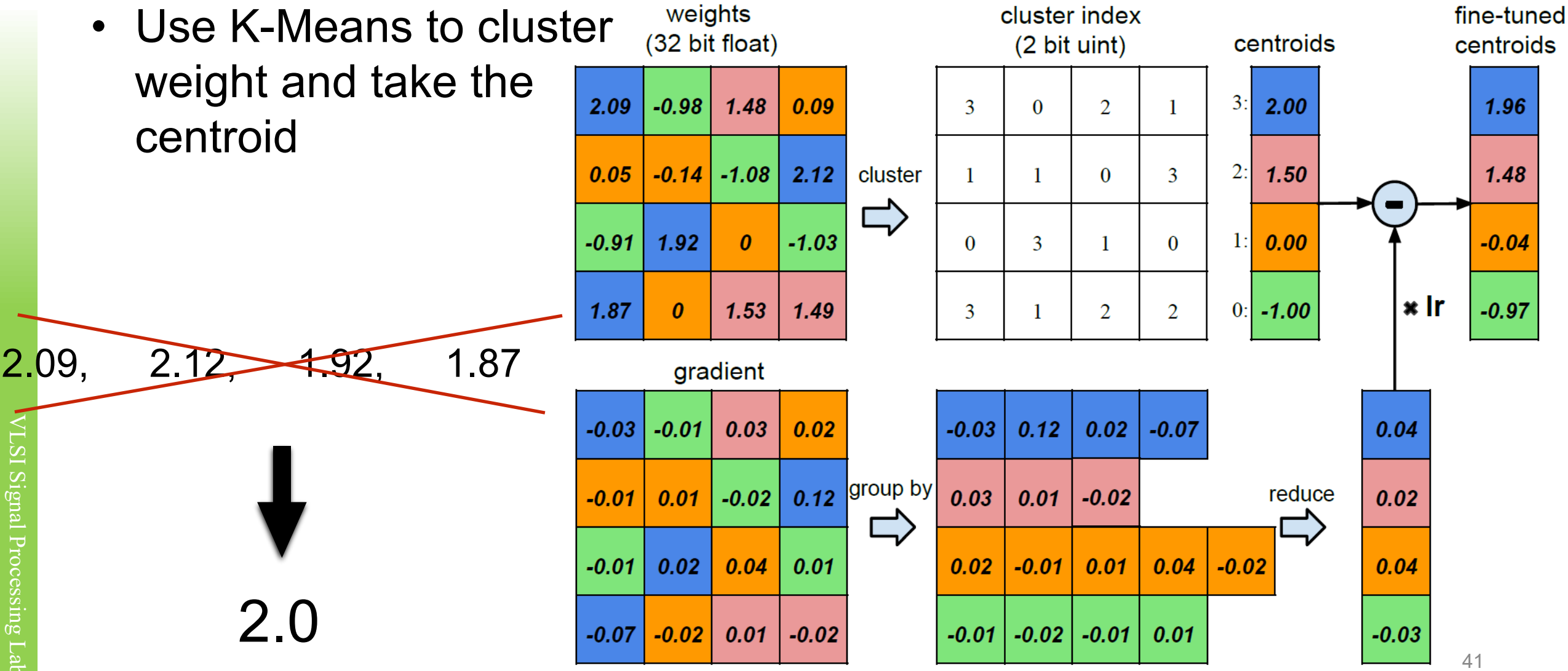


Codebook example

Index	Value	Interval
0	-0.07	[-0.08, -0.04]
1	0	[-0.04, 0.05]
2	0.01	[0.05, 0.011]
3	0.012	[0.011, 0,013]

Weight Sharing: Codebook Quantization

- Use K-Means to cluster weight and take the centroid



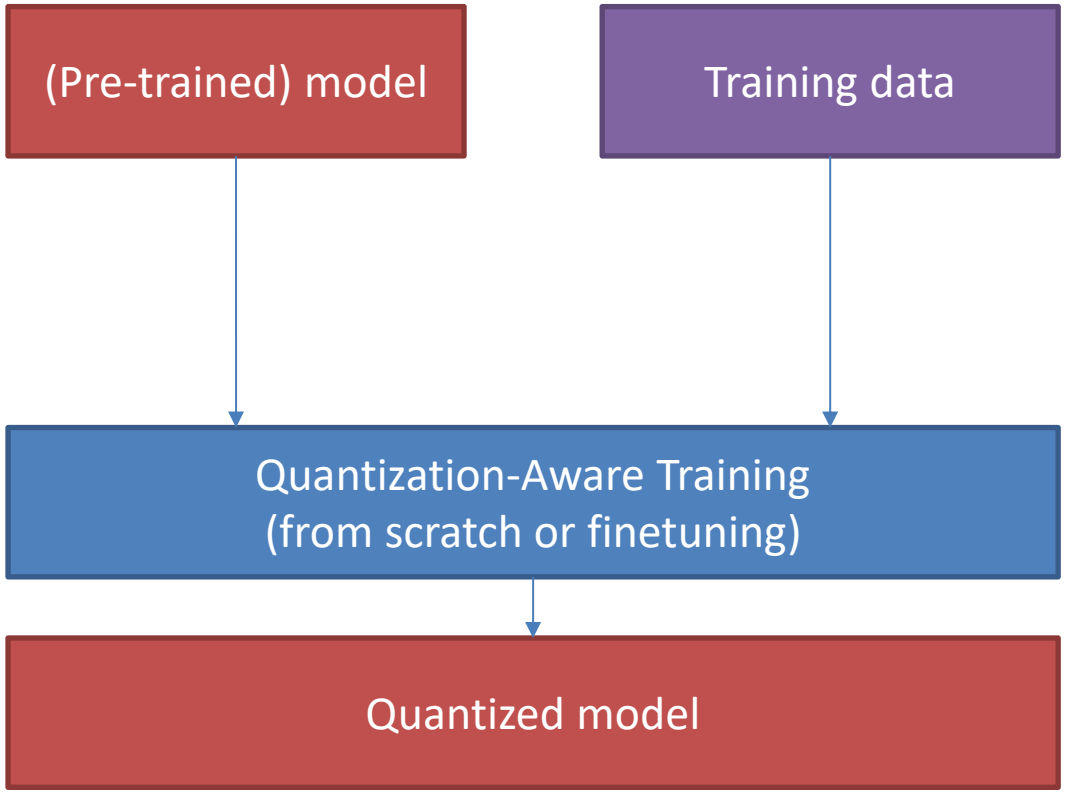
~~2.09, 2.12, 1.92, 1.87~~

2.0

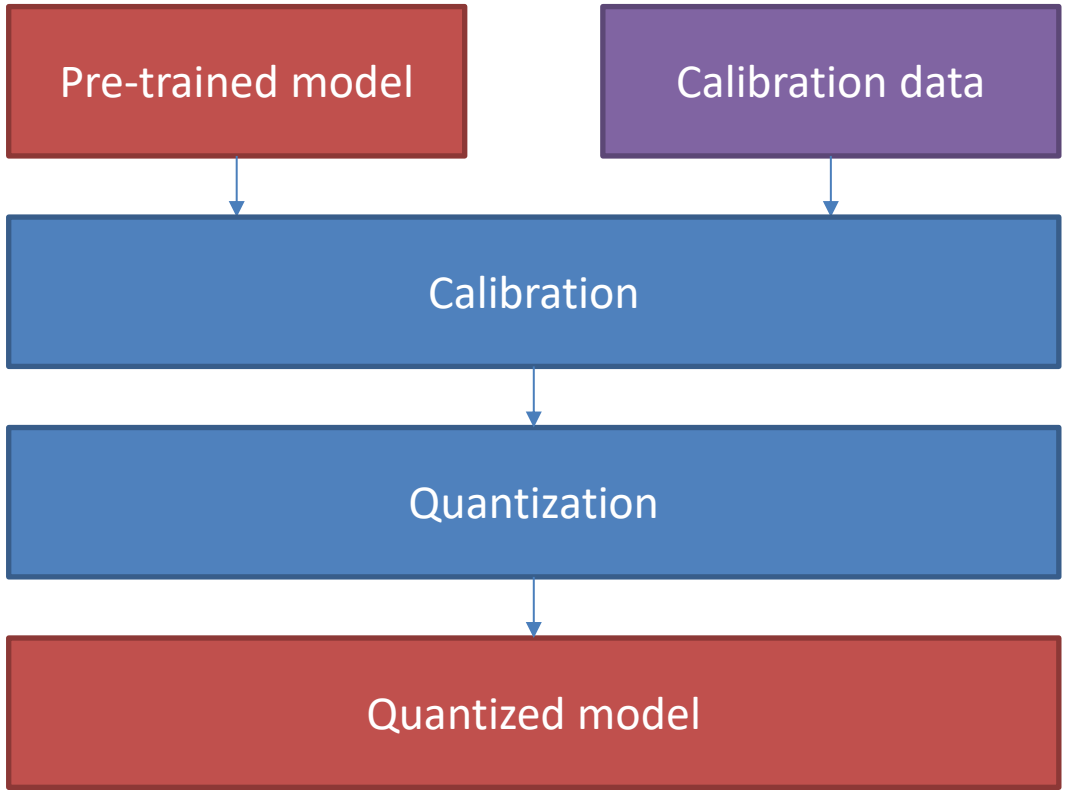
WHEN TO QUANTIZE?

When to quantize?

Quantization-Aware Training



Post-Training Quantization



Post-Training Quantization

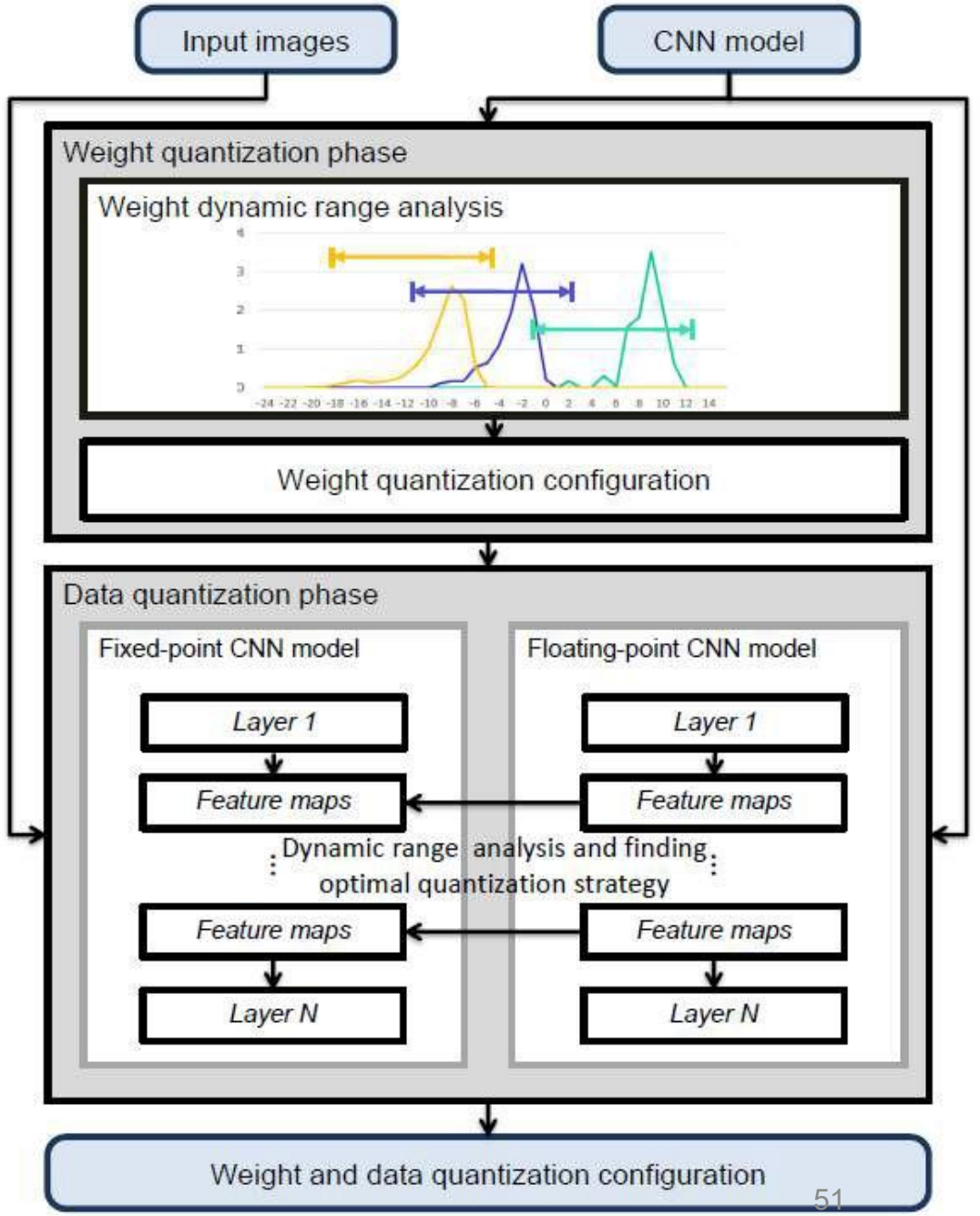
- Typical PTQ procedure:

1. Range analysis using calibration data (subset of training data)
   - To determine step size for activations
   - Step size for weights can be determined without any data

2. Check if quality of final solution is sufficient. If not, do some finetuning (using entire training data)

Quantize Weight and Activation (Post training)

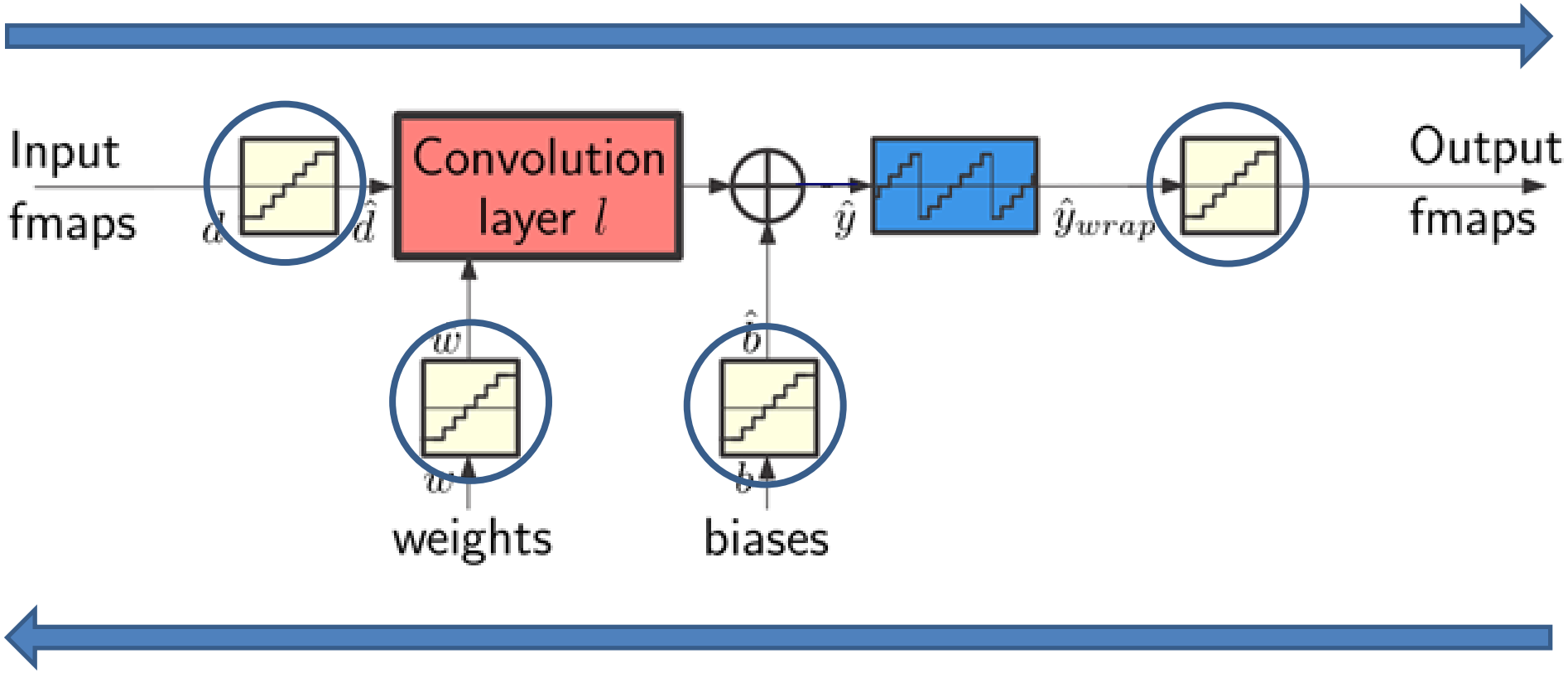
- Train with float
  - For int8, train with RELU6
  - INT8=> (3b int, 5b fractional)
- Quantizing the weight and activation:
  - Gather the statistics for weight and activation based on calibration data
  - Choose proper radix point position
- Fine-tune in float format with calibration data (optional step)
- Convert to fixed-point format



Qiu et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network, FPGA'16

Quantization-Aware Training

Forward pass: simulated quantization to compute loss function



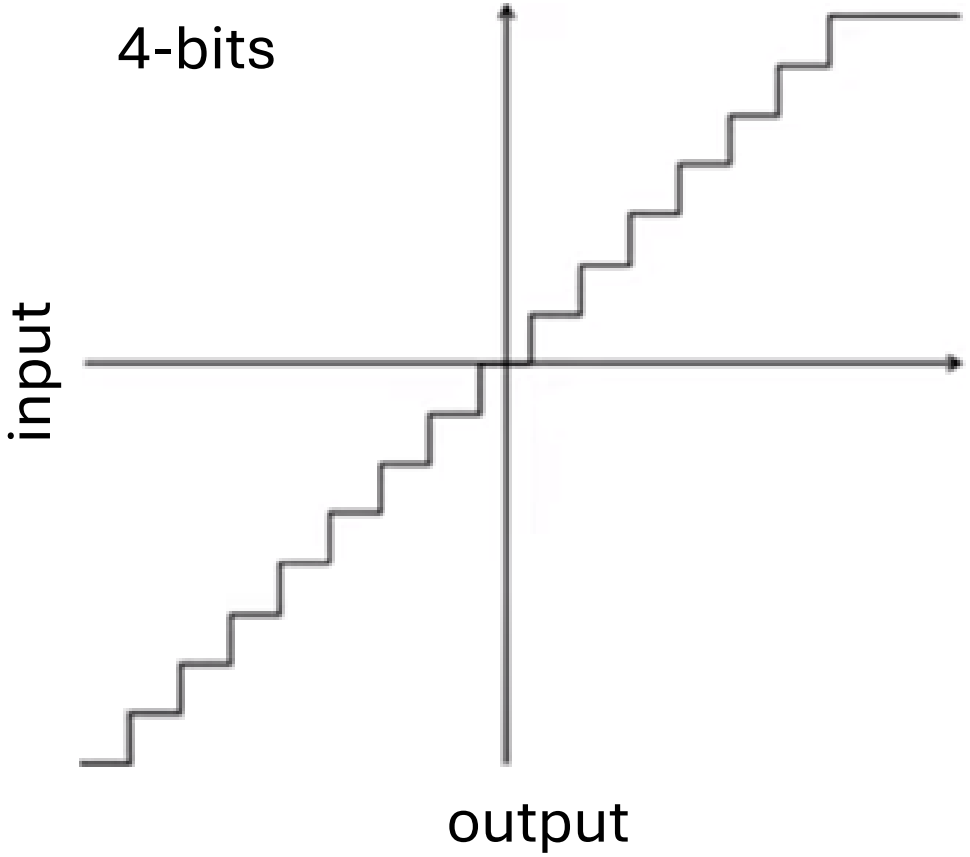
Backward pass: how to back propagate through the quantizers?

Image source: Quantization of deep neural networks for accumulator-constrained processors – B. de Bruin et al.

Quantizers: how to backpropagate?

$y = \text{quantize}(x)$

4-bits



Forward pass

Backward pass

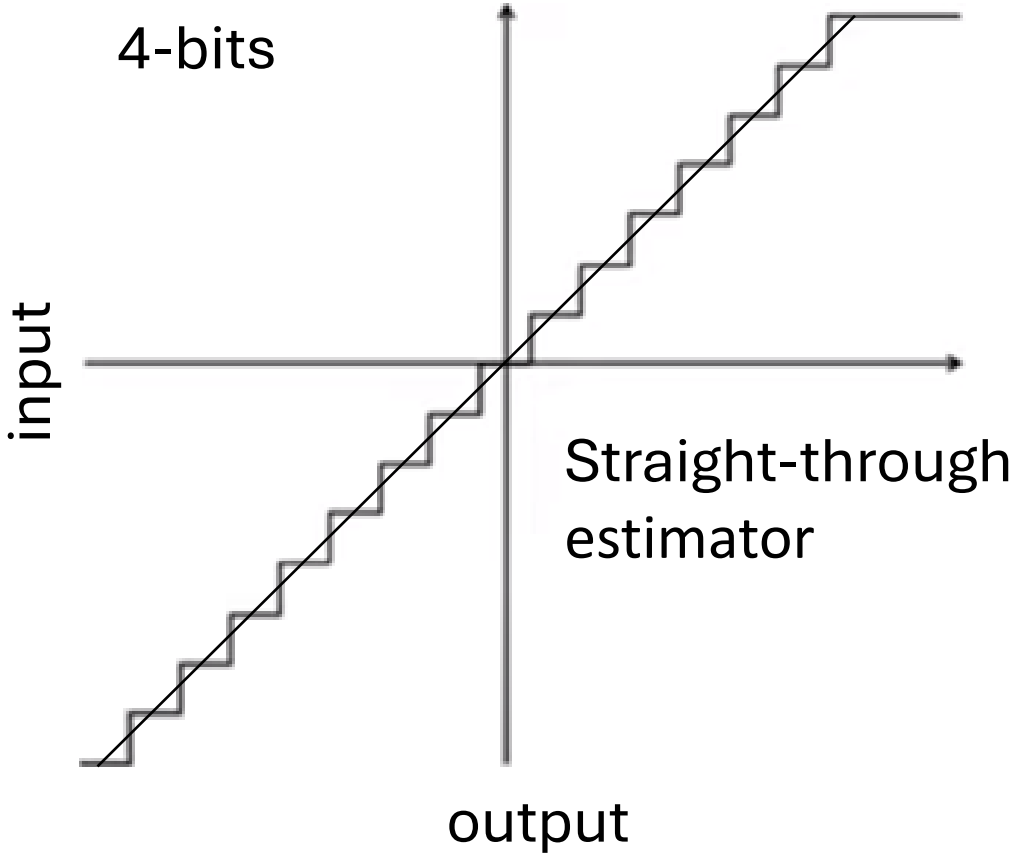
$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial x}$$

?

Quantizers: how to backpropagate?

$y = \text{quantize}(x)$

4-bits



Forward pass

Backward pass

$$\frac{\partial C}{\partial x} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial x}$$

$$\frac{\partial C}{\partial y}\ 1$$

When to quantize? - Summary

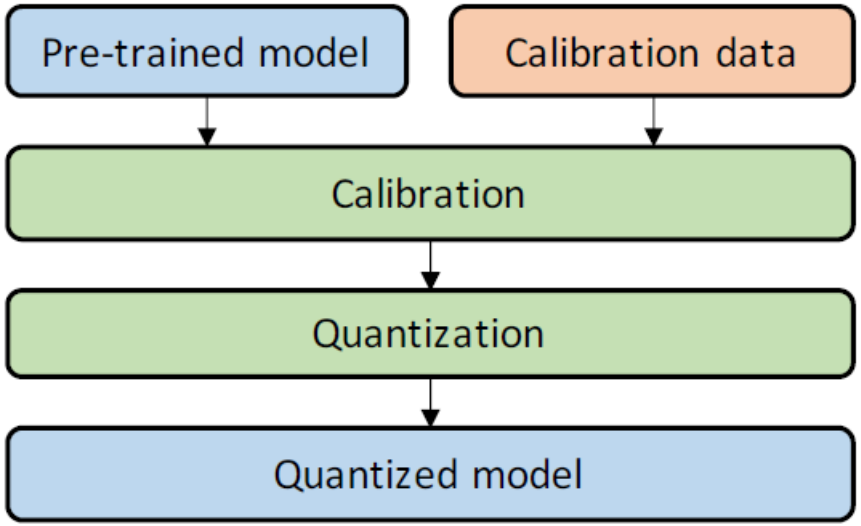
Approach	Training time	Quantization time	Quality of results
Post-Training Quantization	None	1 forward pass	Low
Post-Training Quantization with fine-tuning	Short	Multiple forward & backward passes	Medium
Quantization-Aware Training	Long	Entire training	High
Quantized Training Procedure*	Shorter	Entire training	High

*Quantized training not covered in this lecture

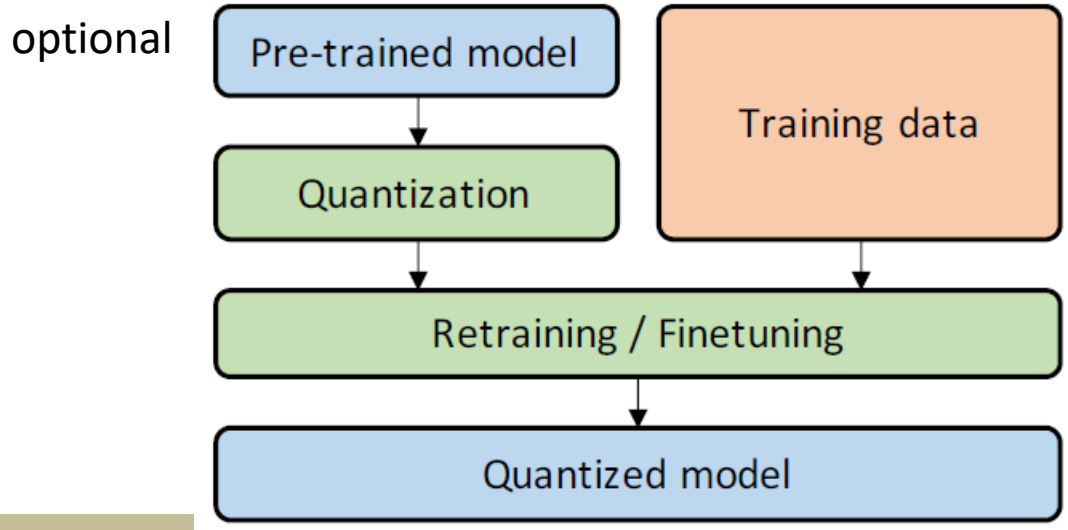
How to Save Quantization Loss?

	Training/fine-tune data	Training	Capability	Complexity
PTQ	No or small	None	No loss >=8bit Large loss for hard case	One command
QAT	Small or Large scale	Fine tuning or Full	Solve the ones that PTQ fails	Need to rewrite or update code

Post-Training Quantization

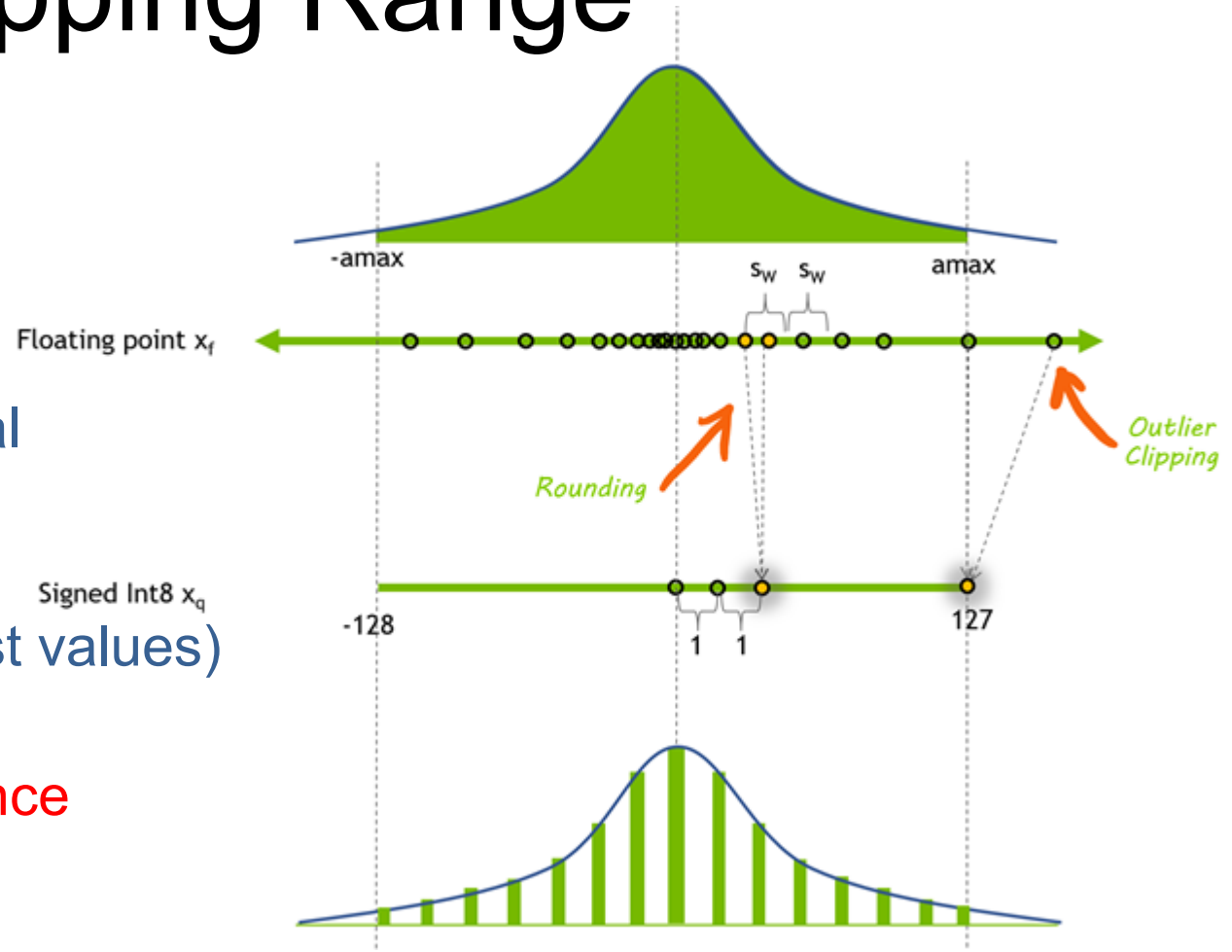


Quantization-Aware Training



PTQ: Decide the Clipping Range

- Method 1
  - Collect the min/max value
  - Map [-max, +max]=[-128, 127]
  - Susceptible to outliner
  - Significant accuracy loss in general
- Method 2
  - Clip the outliner or
  - Use percentile (i-th largest/smallest values)
- Method 3
  - Select the min/max by KL divergence
  - Used by Nvidia TensorRT
- Others
  - Histogram/mean square error
  - Cross entropy for logits of the last layer



$$x_q = clip(round\left(\frac{x}{s} - z\right), n, p)$$

https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/

PTQ:難點

- Calibration校準數據有限：
  - 數據量小，可能有domain 問題；
- 異常數據分佈：
  - Merge BN 後，weight 分佈異常；或是layer concatenation/addition
  - Solution: QAT, mixed precision, training + regularation
- 最佳化空間有限：
  - 早期的PTQ演算法只能確定量化參數的階段範圍；
- 最佳化粒度選擇：
  - Tensor/Layer by layer/End to end。
- Help your PTQ
  - Train models with range clipping

PTQ: Calibration Data

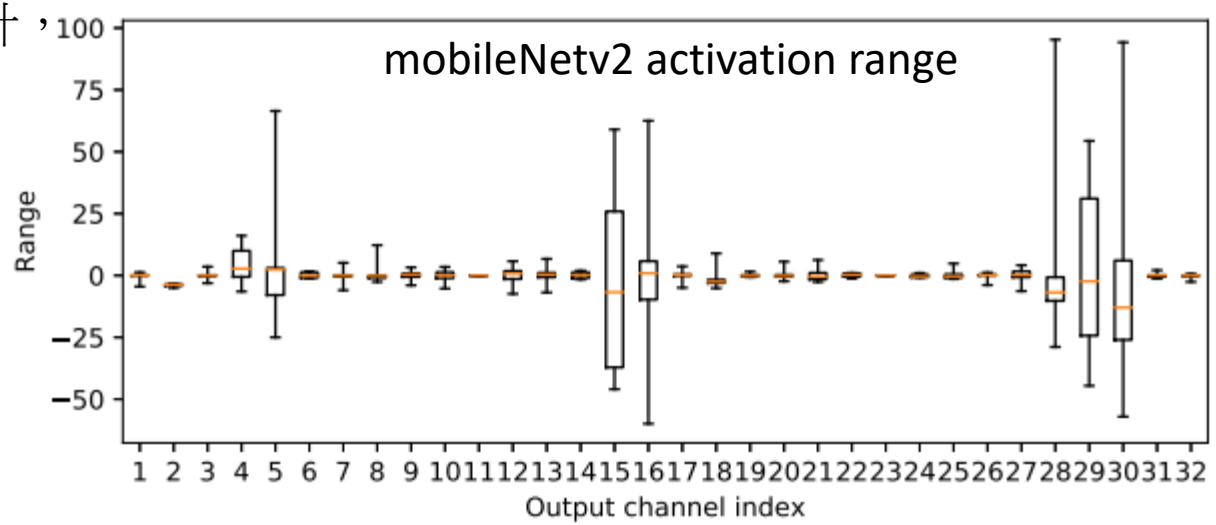
- Calibration Data
  - 通常只能拿到部分數據，因此校準數據是否能代表實際場景至關重要。在實際使用中經常遇到校準集不夠好導致量化掉點的情況
- 極端狀況
  - Data free quantization
  - Cross domain quantization
    - 在B 校準完之後，再用A 的數據去更新BN 的參數。根據經驗，這種思路比較適合學術界，在實際場景中靠BN 校準很難做到穩定，
    - 實際場景下最好能夠拿到校準數據，而且挑出比較好的校準數據，就可以實現好的量化精度結果

Qualcomm: AIMET

PTQ: Data Free Quantization

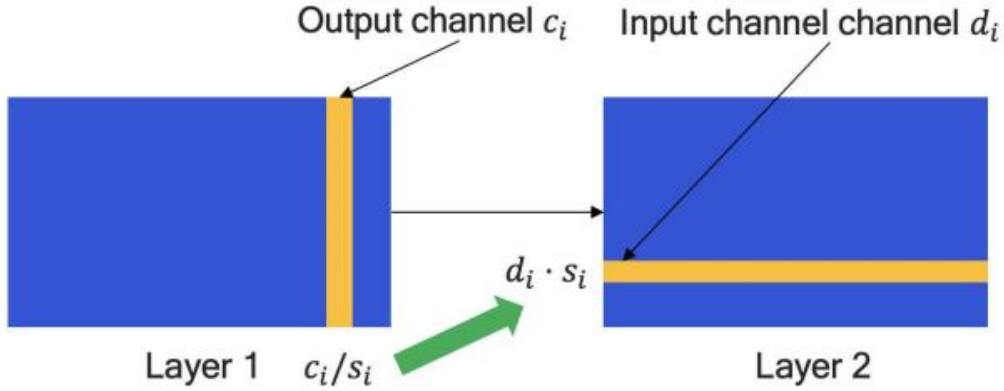
Method 1: 依據BN的α, β 去做Activation量化範圍估計，誤差大

$$q_{\min} = \min(\beta - \alpha\gamma)$$
$$q_{\max} = \max(\beta + \alpha\gamma)$$



Method 2: weight equalization + bias correction

Weight equalization



把上層的scale 的差異轉移到下一層，並逐層的轉移到最後，來降低不同output channel 量化誤差的影響

Bias correction

$$\mathbb{E}[\tilde{\mathbf{y}}_j - \mathbf{y}_j] \approx \frac{1}{N}\sum_n (\tilde{\mathbf{W}}\mathbf{x}_n)_j - (\mathbf{W}\mathbf{x}_n)_j$$

	~D	~BP	~AC	MobileNetV2 FP32	MobileNetV2 INT8
DFQ (ours)	✓	✓	✓	71.7%	**71.2%**
Per-layer [18]	✓	✓	✓	71.9%	0.1%
Per-channel [18]	✓	✓	✓	71.9%	69.7%

Data-Free Quantization Through Weight Equalization and Bias Correction

Qualcomm: AIMET

PTQ:最佳化空間

- Optimization space
- Weight 能在PTQ中最佳化嗎?
  - AdaRound
  - weight量化時，四捨五入不是最好
  - weight之間的是存在相互的影響的
  - 利用梯度下降算法進行求解

$$x_q = clip(round\left(\frac{x}{s} - z\right), n, p)$$

$$x_q = clip(\textcolor{red}{round}\left(\frac{x}{s} - z\right), n, p)$$

$$\arg\min_{\mathbf{V}} \left\|\mathbf{W}\mathbf{x} - \widetilde{\mathbf{W}}\mathbf{x}\right\|_F^2 + \lambda f_{reg}(\mathbf{V}),$$

$$\widetilde{\mathbf{W}} = \mathbf{s} \cdot clip\left(\left\lfloor \frac{\mathbf{W}}{\mathbf{s}} \right\rfloor + h(\mathbf{V}), \mathbf{n}, \mathbf{p}\right)$$

$$f_{reg}(\mathbf{V}) = \sum_{i,j} 1 - |2h(\mathbf{V}_{i,j}) - 1|^{\beta},$$

$$h(\mathbf{V}_{i,j}) = clip(\sigma(\mathbf{V}_{i,j})(\zeta - \gamma) + \gamma, 0, 1),$$

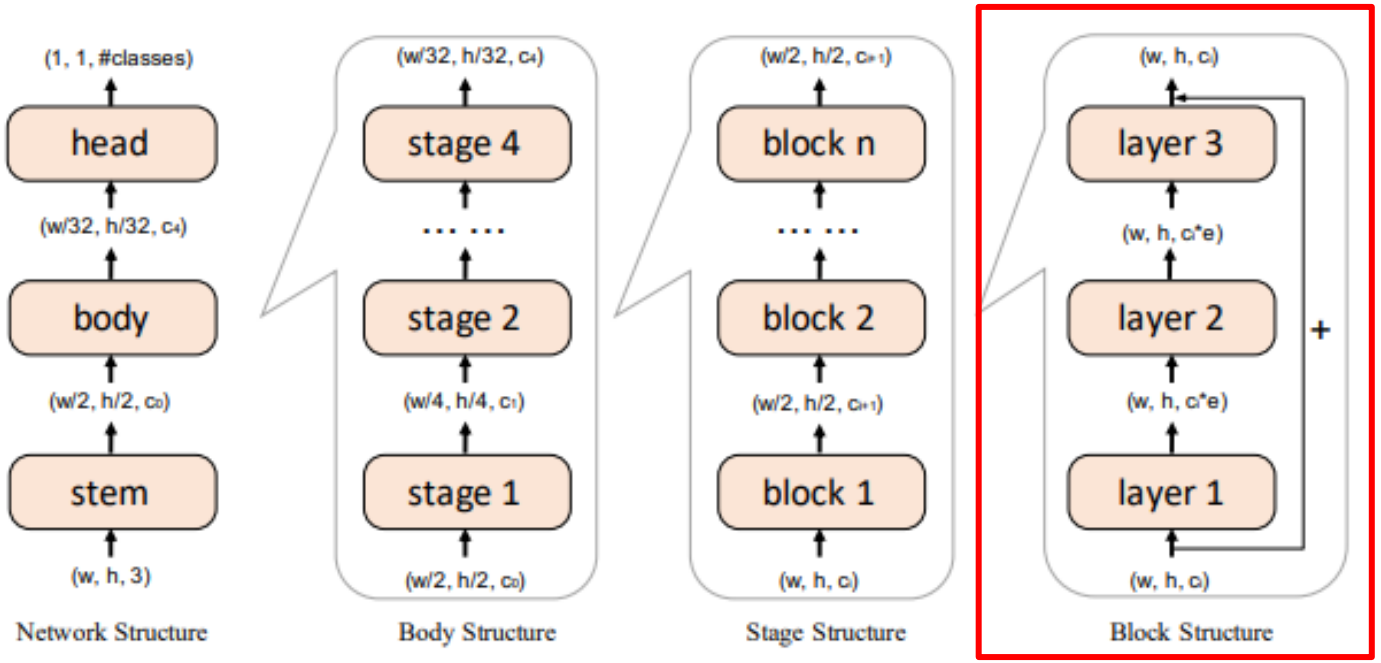
$$\arg\min_{\mathbf{V}} \left\| f_a(\mathbf{W}\mathbf{x}) - f_a\left(\widetilde{\mathbf{W}}\hat{\mathbf{x}}\right)\right\|_F^2 + \lambda f_{reg}(\mathbf{V})$$

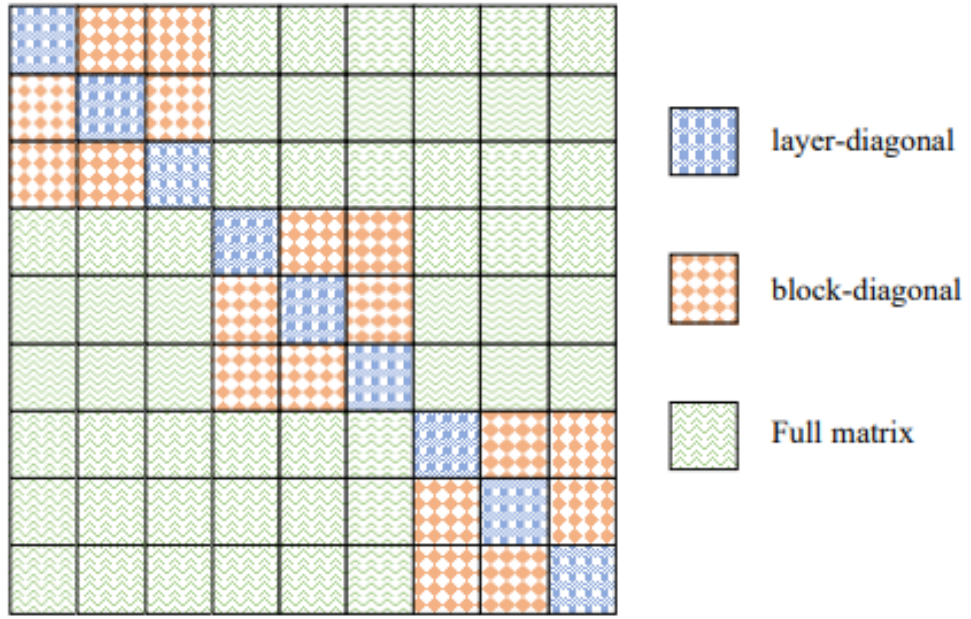
Optimization	#bits W/A	mIOU
Full precision	32/32	72.94
DFQ (Nagel et al., 2019)	8/8	72.33
Nearest	4/8	6.09
DFQ (our impl.)	4/8	14.45
AdaRound	4/32	70.89±0.33
AdaRound w/ act quant	4/8	70.86±0.37

Table 9. Comparison among different post-training quantization strategies, in terms of Mean Intersection Over Union (mIOU) for DeeplabV3+ (MobileNetV2 backend) on Pascal VOC.

PTQ: Quantization Granularity (Tensor, Channel, Layer?)

- Block granularity works best
  - 因為資料少， granularity太大易overfitting， granularity太小最佳化空間不夠
  - Down to 4bit W/A with par performance to QAT





Quantizer Design: Weight 可以直接量化

- Post-training quantization: Weight only, done without requiring any validation data

Network	Asymmetric, per-layer	Symmetric , per-channel	Asymmetric, per-channel	Floating Point
Mobilenetv1_1_224	0.001	0.591	0.704	0.709
Mobilenetv2_1_224	0.001	0.698	0.698	0.719
NasnetMobile	0.722	0.721	0.74	0.74
Mobilenetv2_1.4_224	0.004	0.74	0.74	0.749
Inceptionv3	0.78	0.78	0.78	0.78
Resnet_v1_50	0.75	0.751	0.752	0.752
Resnet_v2_50	0.75	0.75	0.75	0.756
Resnet_v1_152	0.766	0.763	0.762	0.768
Resnet_v2_152	0.761	0.76	0.77	0.778

Table 2: Weight only quantization: per-channel quantization provides good accuracy, with asymmetric quantization providing close to floating point accuracy.

Quantizing deep convolutional networks for efficient inference: A whitepaper

Quantizer Design

- Post-training quantization: Weight + activations (per layer)

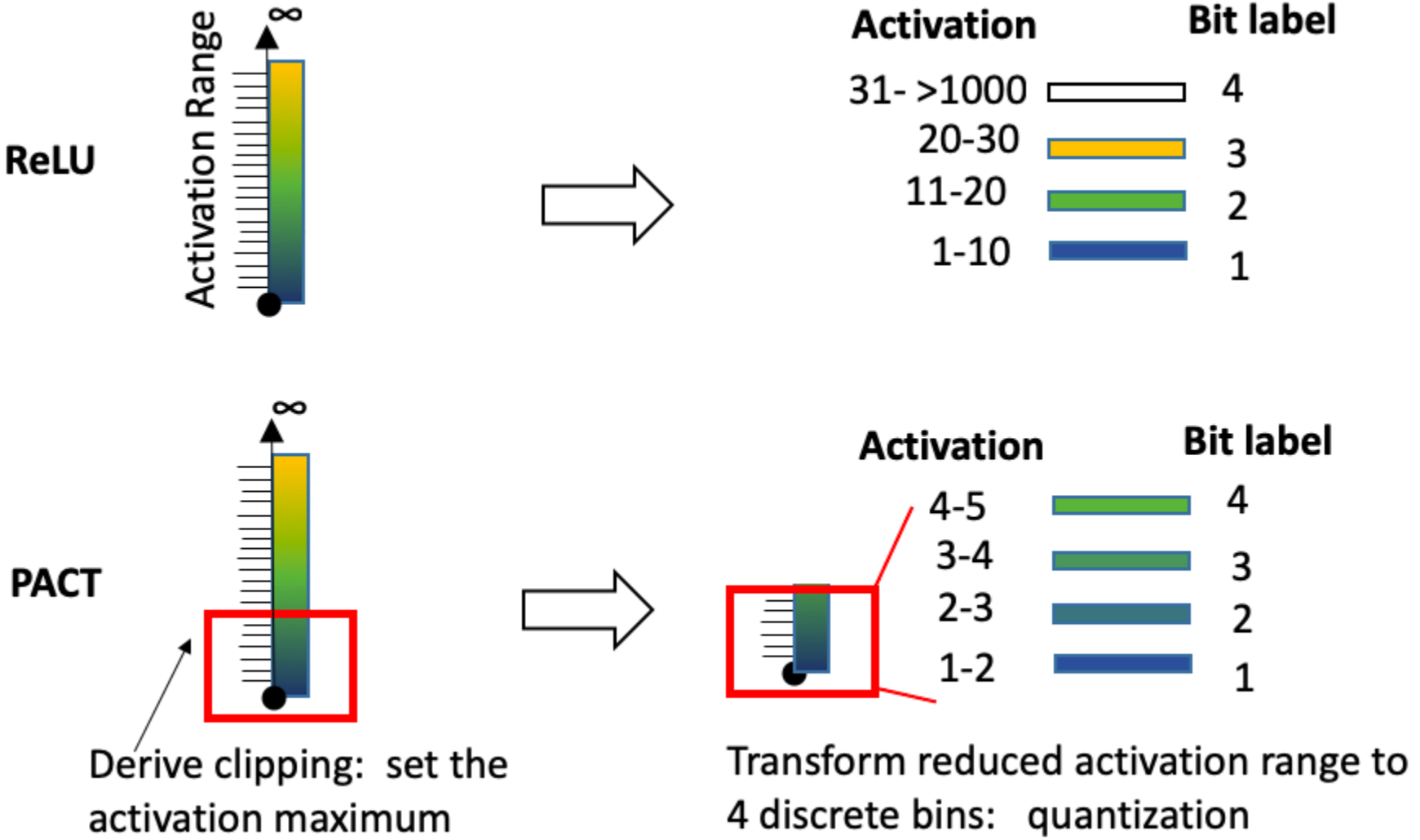
Network	Asymmetric, per-layer	Symmetric , per-channel	Asymmetric, per-channel	Activation Only	Floating Point
Mobilenet-v1_1_224	0.001	0.591	0.703	0.708	0.709
Mobilenet-v2_1_224	0.001	0.698	0.697	0.7	0.719
Nasnet-Mobile	0.722	0.721	0.74	0.74	0.74
Mobilenet-v2_1.4_224	0.004	0.74	0.74	0.742	0.749
Inception-v3	0.78	0.78	0.78	0.78	0.78
Resnet-v1_50	0.75	0.751	0.751	0.751	0.752
Resnet-v2_50	0.75	0.75	0.75	0.75	0.756
Resnet-v1_152	0.766	0.762	0.767	0.761	0.768
Resnet-v2_152	0.761	0.76	0.76	0.76	0.778

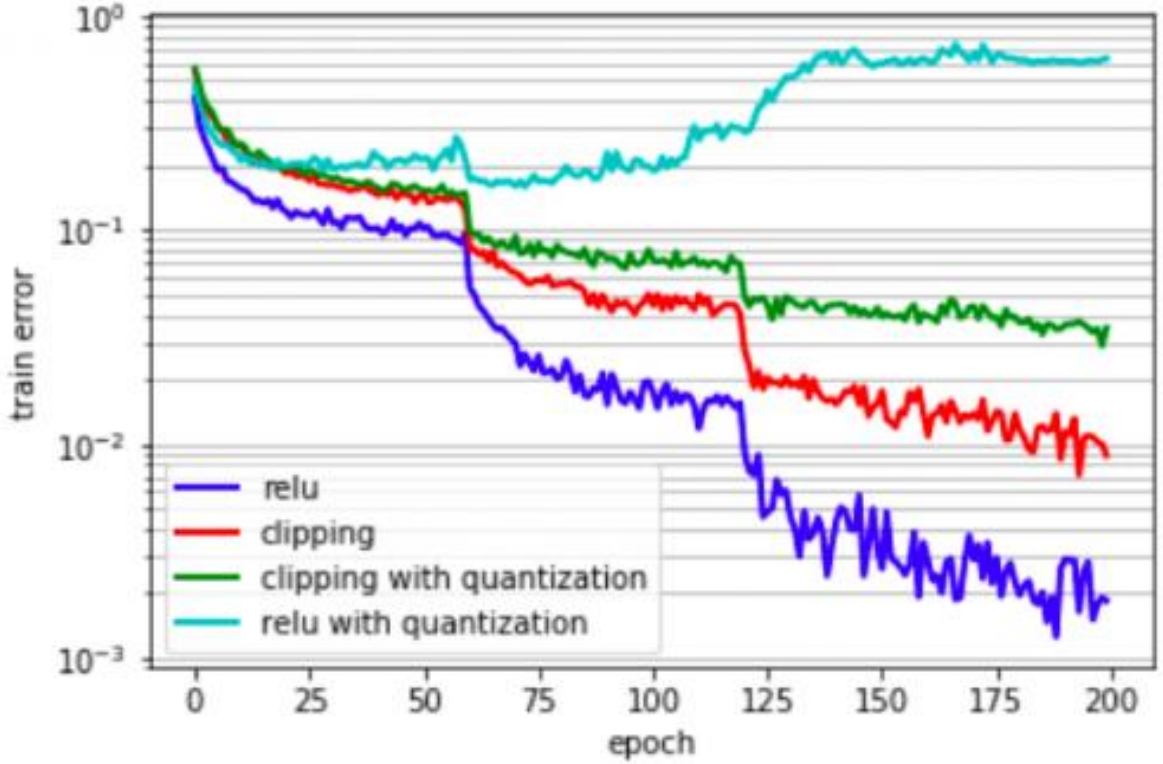
Table 3: Post training quantization of weights and activations: per-channel quantization of weights and per-layer quantization of activations works well for all the networks considered, with asymmetric quantization providing slightly better accuracies.

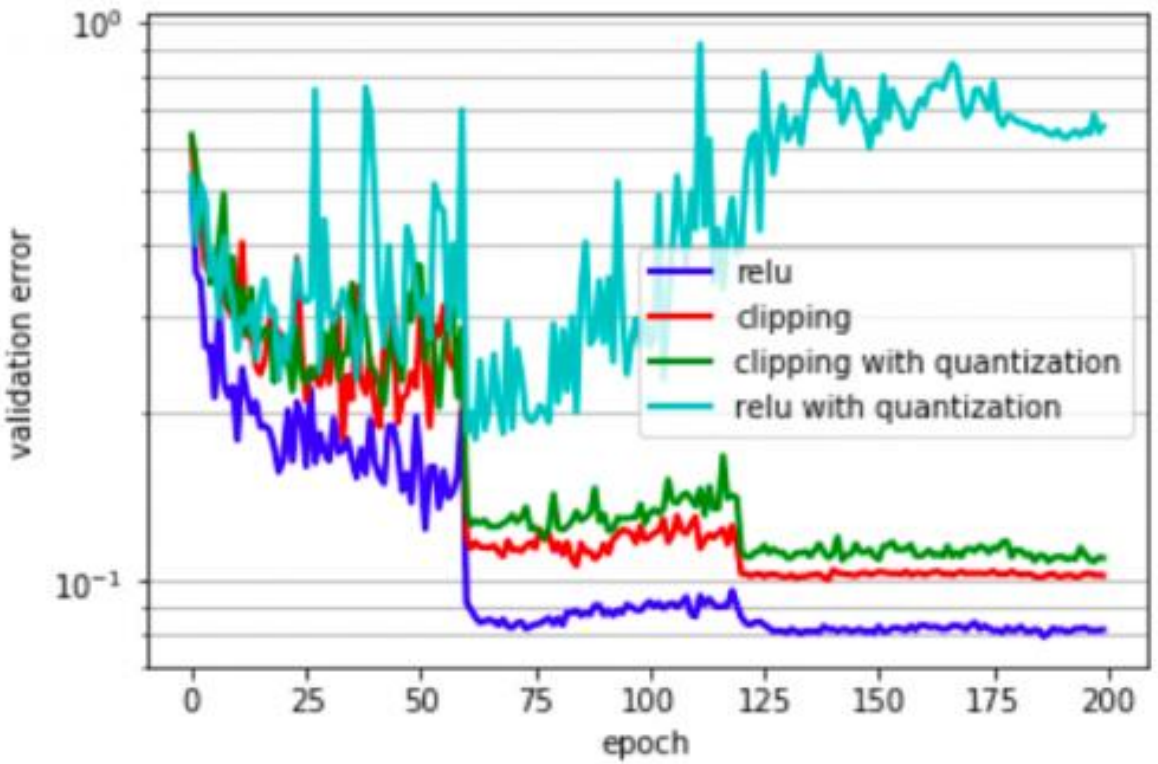
Quantizing deep convolutional networks for efficient inference: A whitepaper

Highly Accurate Deep Learning Inference with 2-bit Precision

- Current quantization limited by ReLU
  - relies on high dynamic range and precision to capture the unbounded continuous range of activation values
- PArameterized Clipping acTivation (PACT)
  - a maximum activation value is automatically derived for the unlimited range of activation values as part of the model training itself

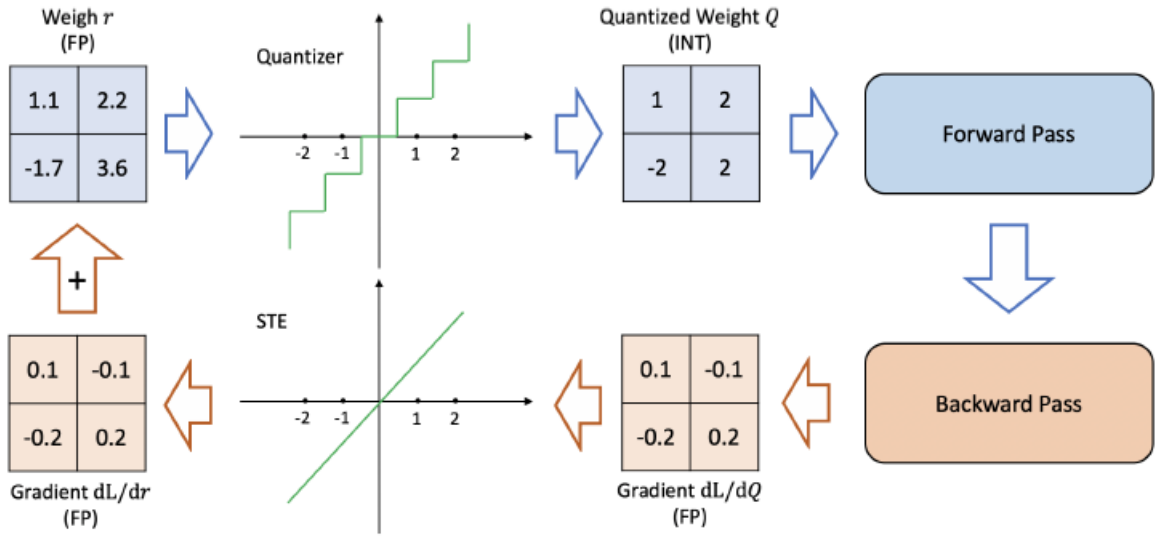

ReLU
Activation Range
∞
Activation
Bit label
31- >1000
4
20-30
3
11-20
2
1-10
1
PACT
∞
Activation
Bit label
4-5
4
3-4
3
2-3
2
1-2
1
Derive clipping: set the activation maximum
Transform reduced activation range to 4 discrete bins: quantization


train error
epoch
relu
dipping
clipping with quantization
relu with quantization


validation error
epoch
relu
dipping
clipping with quantization
relu with quantization

Quantization Aware Training (QAT)

- Higher accuracies than post quantization training schemes
- How
  - Forward pass: quantized value
  - Backward pass:
    - stored in float, use "straight through estimation (STE)" to model gradient



Quantizer Design: Quantization Aware Training

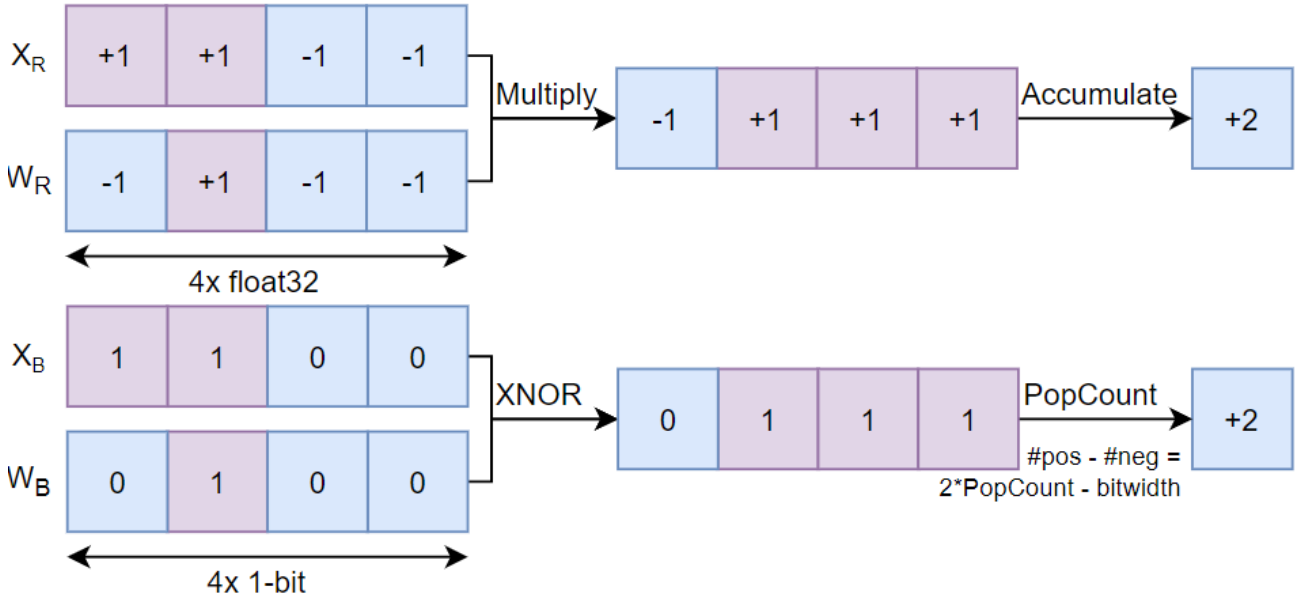
Network	Asymmetric, per-layer (Post Training Quantization)	Symmetric , per-channel (Post Training Quantization)	Asymmetric, per-layer (Quantiza-tion Aware Training)	Symmetric, per-channel (Quantiza-tion Aware Training)	Floating Point
Mobilenet-v1_1_224	0.001	0.591	0.70	0.707	0.709
Mobilenet-v2_1_224	0.001	0.698	0.709	0.711	0.719
Nasnet-Mobile	0.722	0.721	0.73	0.73	0.74
Mobilenet-v2_1.4_224	0.004	0.74	0.735	0.745	0.749
Inception-v3	0.78	0.78	0.78	0.78	0.78
Resnet-v1_50	0.75	0.751	0.75	0.75	0.752
Resnet-v2_50	0.75	0.75	0.75	0.75	0.756
Resnet-v1_152	0.766	0.762	0.765	0.762	0.768
Resnet-v2_152	0.761	0.76	0.76	0.76	0.778

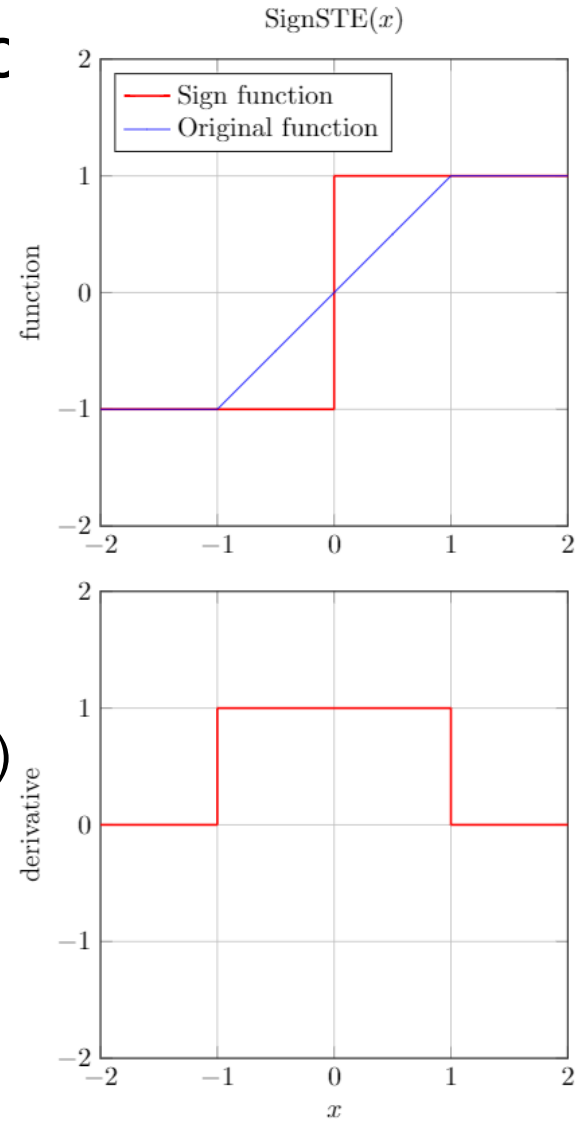
- Training closes the gap between symmetric and asymmetric quantization. (8bit quantization)
- Training allows for simpler quantization schemes to provide close to floating point accuracy. Even per-layer quantization shows close to floating point accuracy (see column 4 in Table 4)

EXTREME QUANTIZATION

Basics of Binary/Ternary Neural Networks

- Forward pass (XNOR-POPCOUNT during inferenc





- (Ternary MAC uses similar compute logic: Gated X(N)OR & PopCount)

- Backward pass (training only)
  - Straight-through estimation (SignSTE)

Is BNN Still a Universal Approximator?

- The answer is yes
  - a 3 layer binary neural network with a particular building block structure maintains the universal function approximator property
- Can we achieve FP accuracy?
  - Depends on training strategy

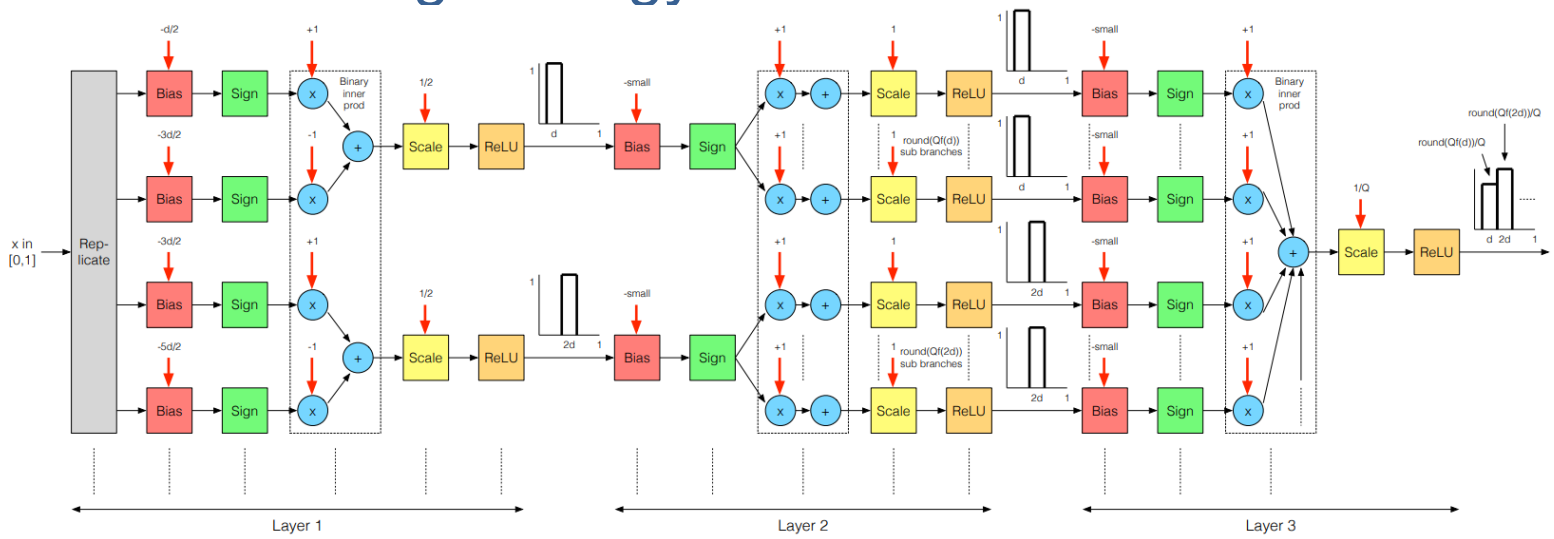


Figure 1. A 3 layer binary neural network that is a universal function approximator.

BCNN: A Binary CNN With All Matrix Ops Quantized To 1 Bit Precision

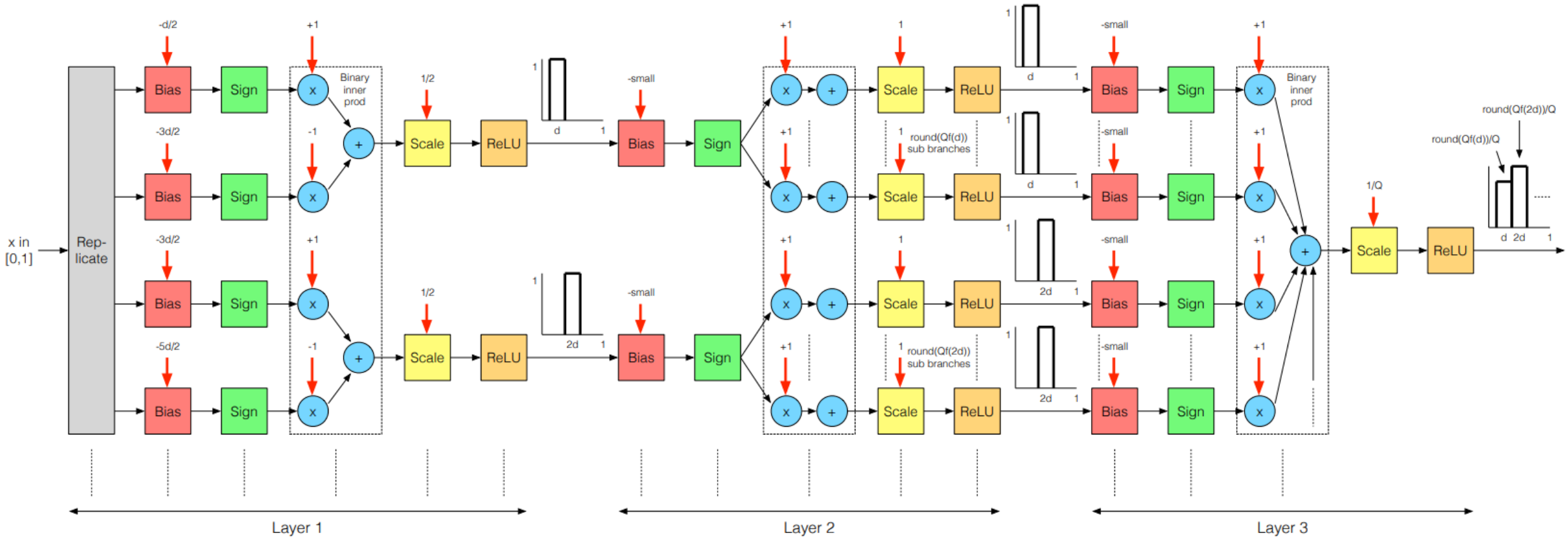
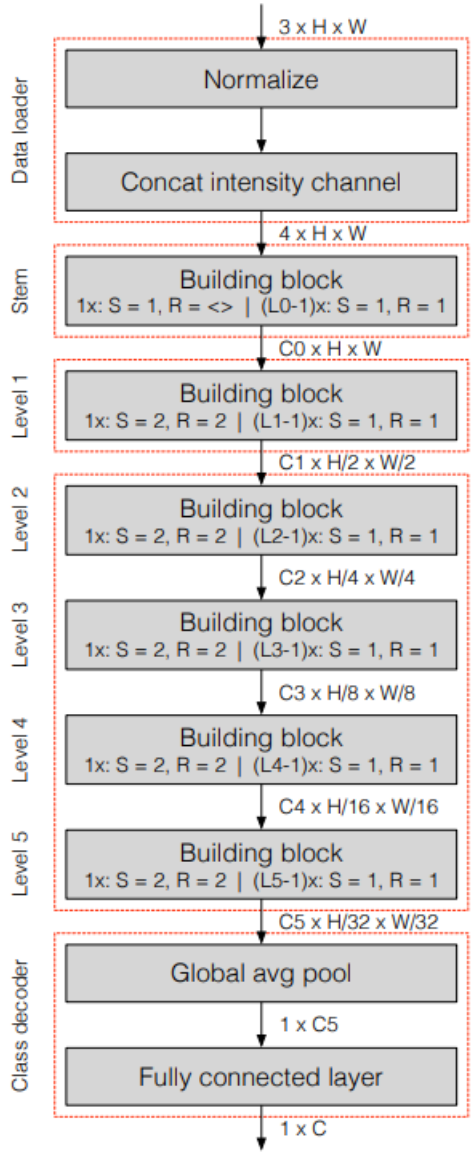
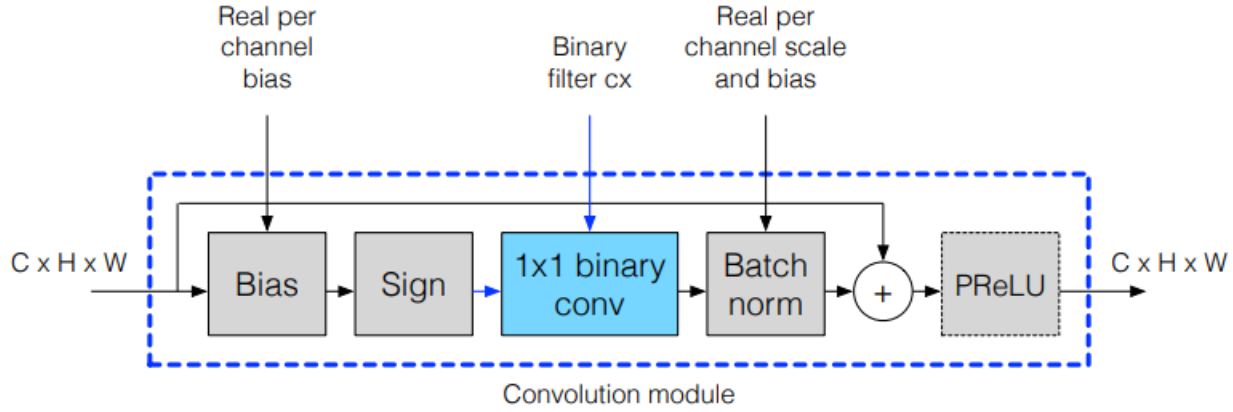
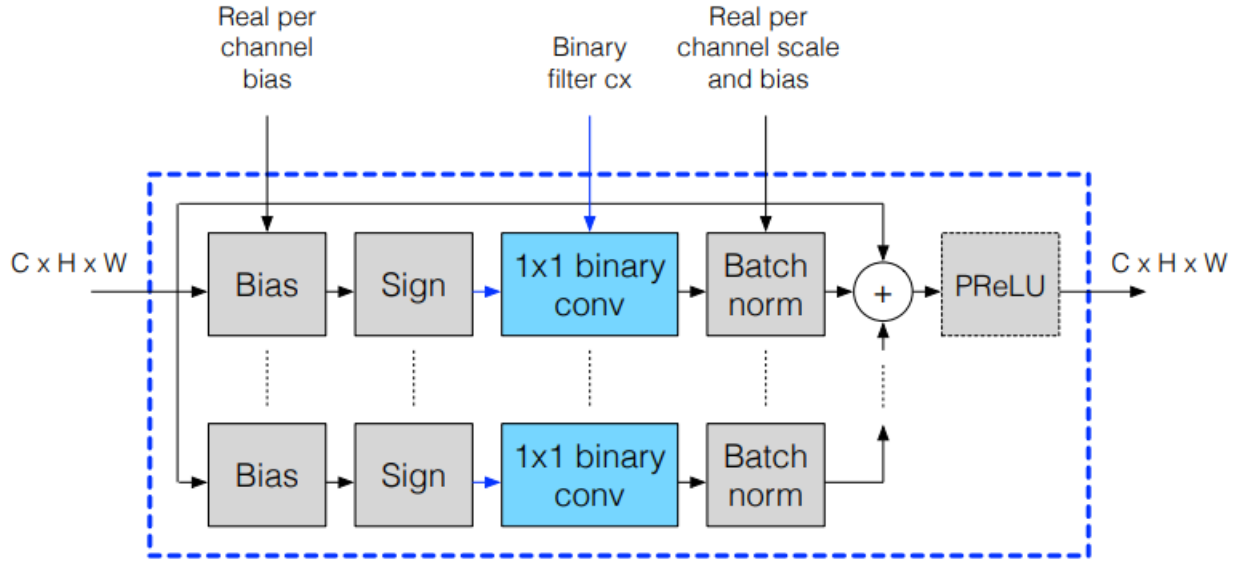


Figure 1. A 3 layer binary neural network that is a universal function approximator.





Convolution module



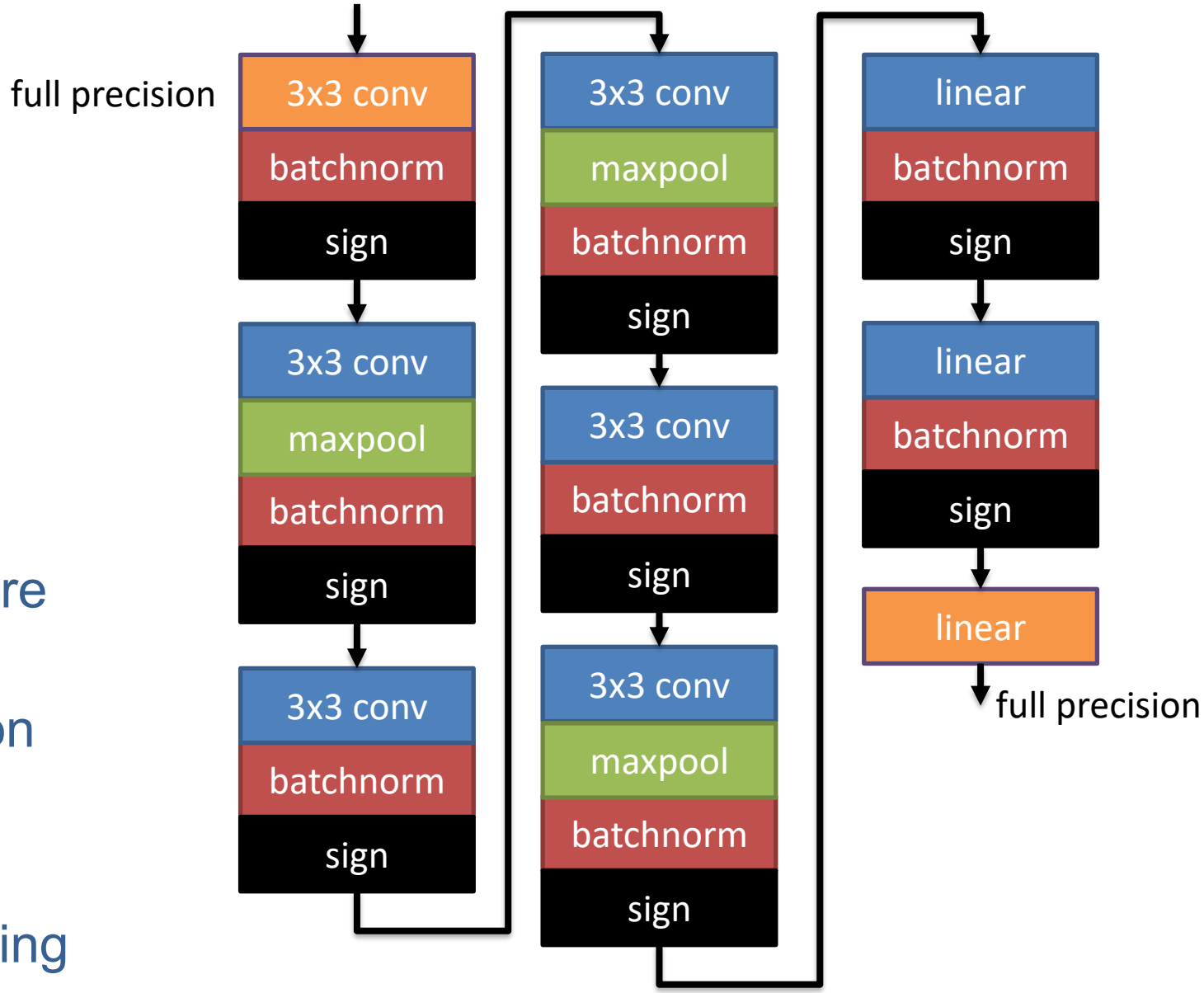
Convolution module with P binary convolution branches summed together

Naive BNN: BinaryNet

- VGG9 architecture
- CIFAR10 dataset

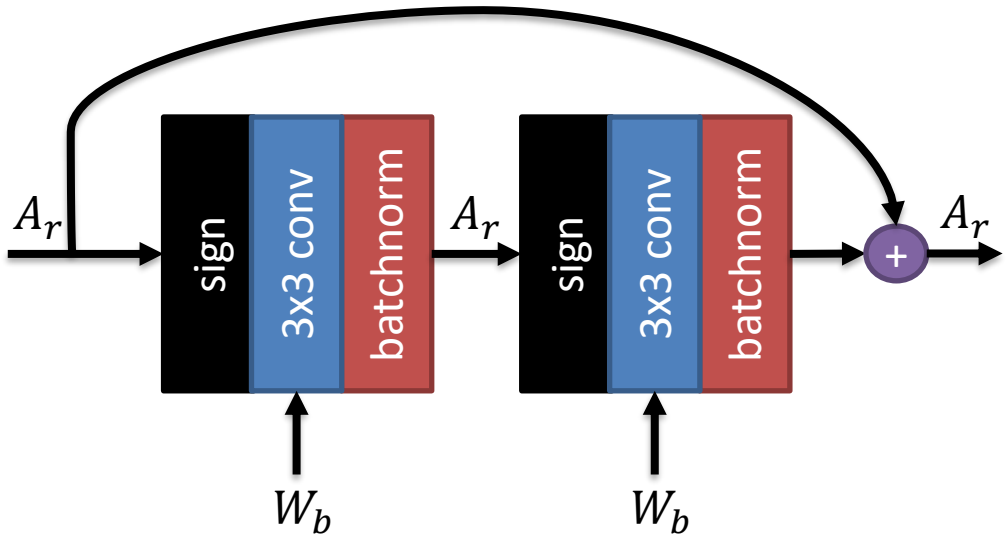
- First and last layer remain full precision (orange)
- Convolution and linear (red) are fully binary!
- Binarization using sign-function with clipped STE

- **88.6**% top-1 accuracy, promising results?

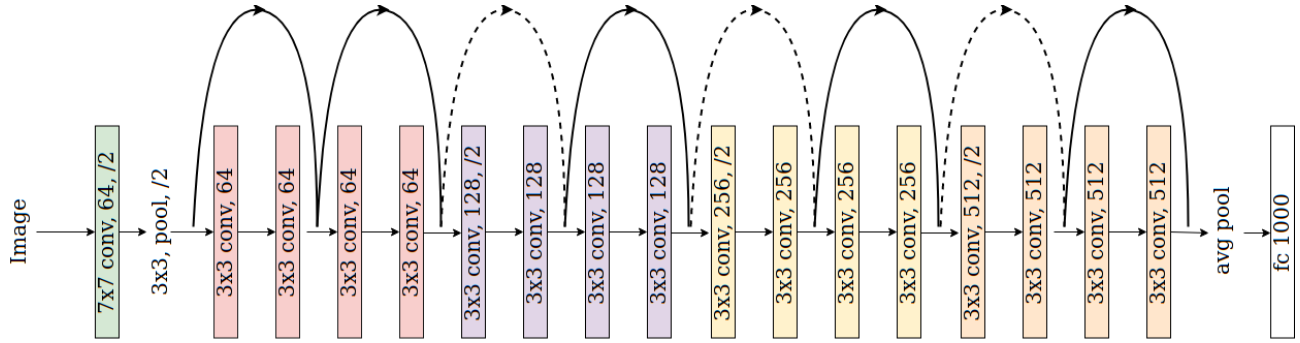


BinaryNet on ImageNet

Building block:

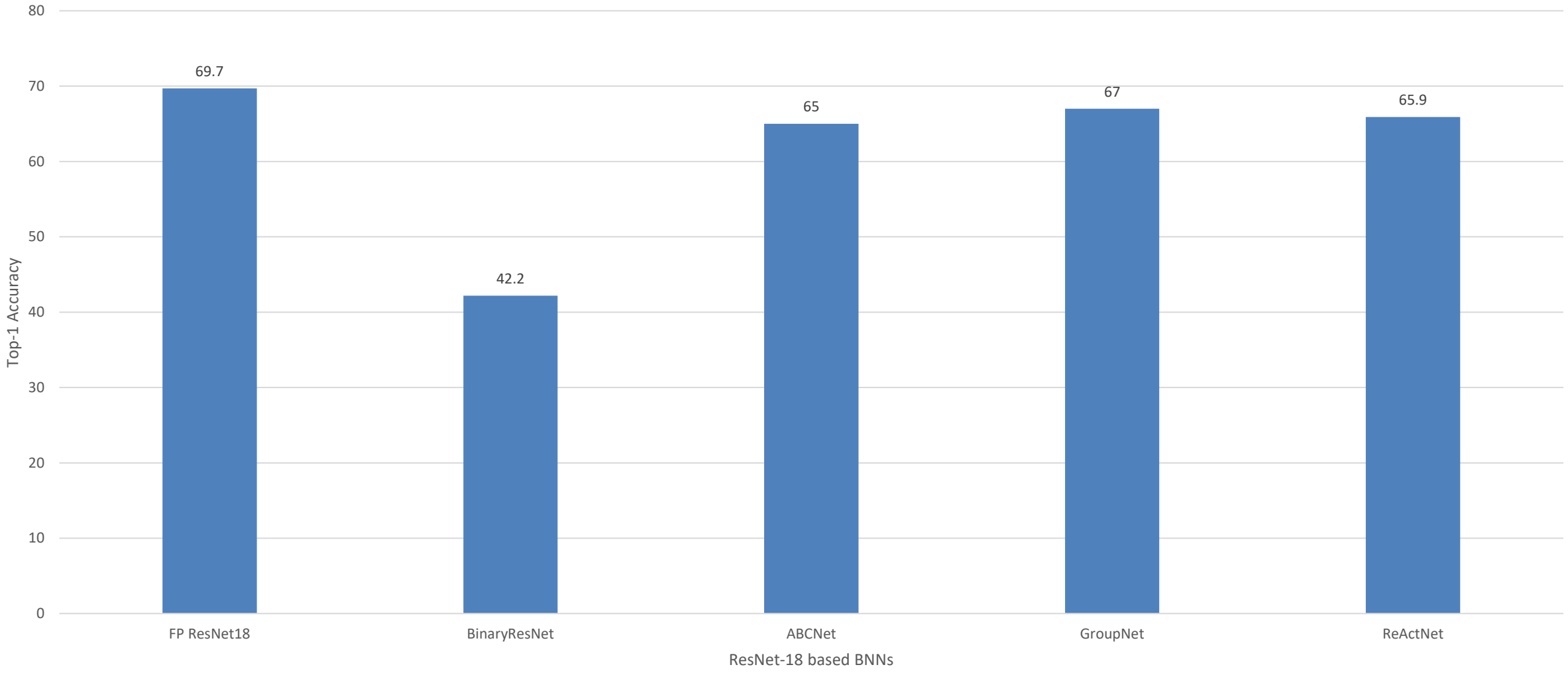


ResNet-18 architecture



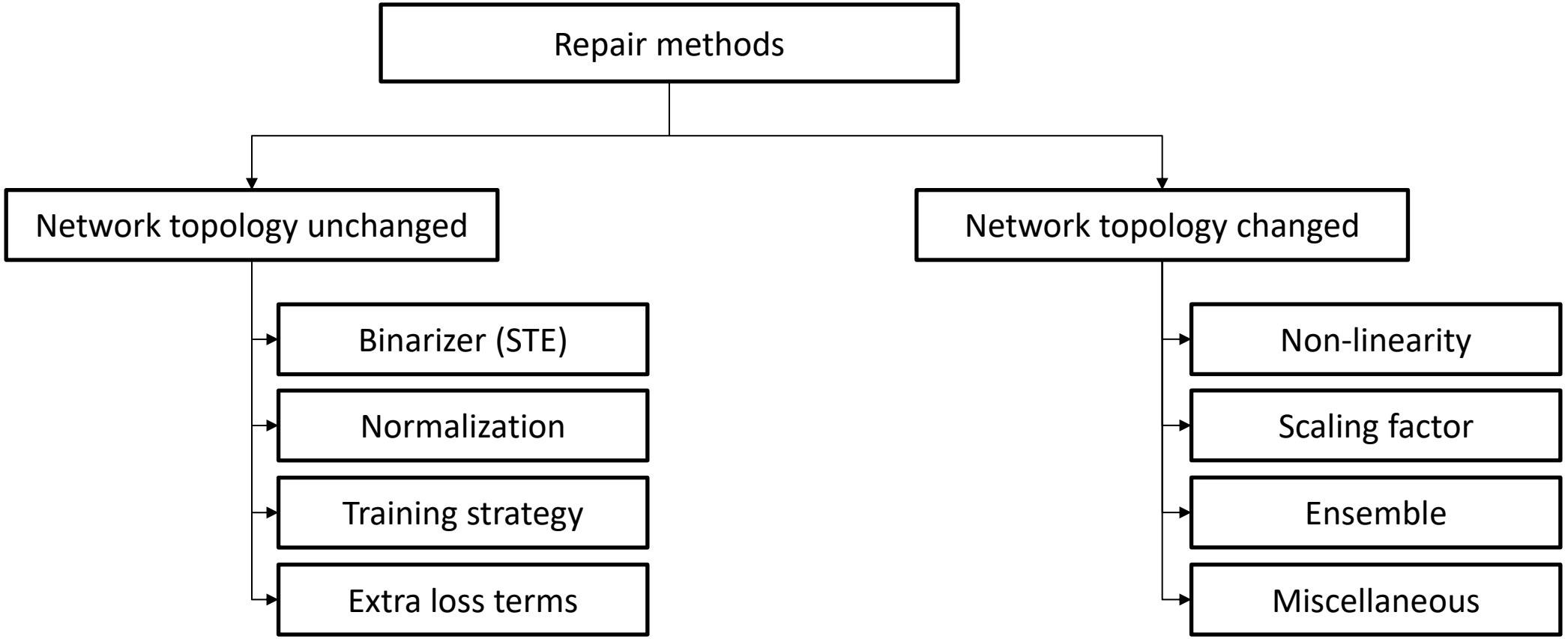
From **69.7%** to **42.2%** top-1 accuracy on ImageNet…

Accuracy of state-of-the-art BNNs

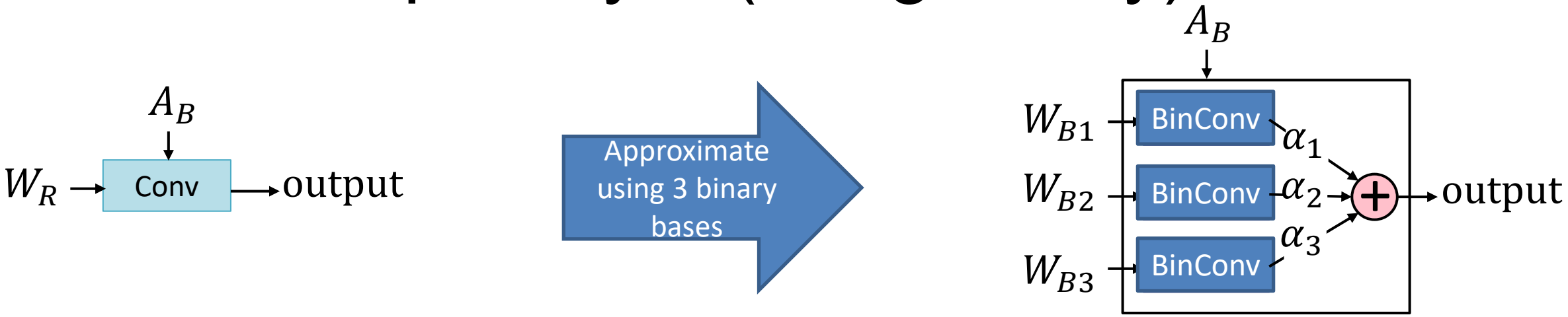


Note: accuracy repair methods do not come for free!

Overview of accuracy repair methods



Ensemble per-layer (Weight only)



– Approximate W_R using linear combination binary filters W_{Bi}:

$$W_R \approx \alpha_1 \mathrm{W}_{B1} + \alpha_2 \mathrm{W}_{B2} + \cdots + \alpha_M \mathrm{W}_{BM}$$

– W_{Bi} constructed by shifting (u_i) over the standard deviation:

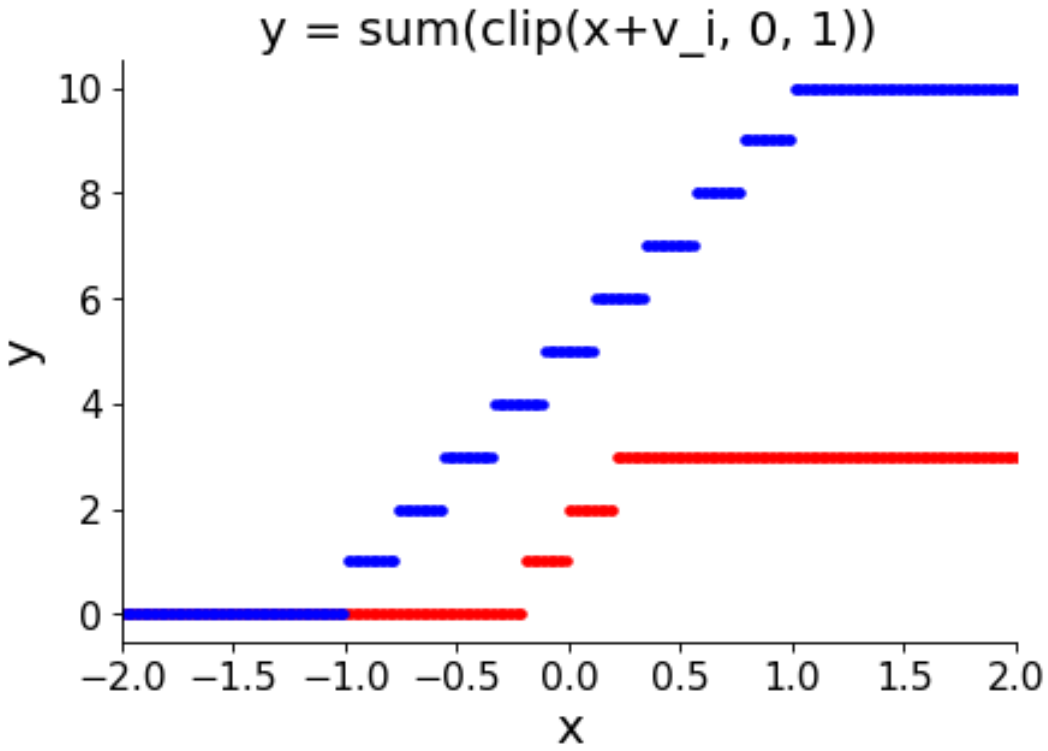
$$\mathrm{W}_{Bi} = \mathrm{sign}\left(W_R - \mathrm{mean}(W_R) + u_i \cdot \mathrm{std}(W_R)\right)$$

– Scaling parameters α_i found by:

$$\min_{\alpha_i} \|W_R - \mathrm{W}_{Bi}\alpha_i\|^2$$

Towards Accurate Binary Convolutional Neural Network – Lin et al. (NIPS 2017)

Ensemble per-layer (Activation only)



Two examples ($\beta = [1, \dots, 1]$):

- 3 bases; $v = [-0.2, 0, 0.2]$
- 10 bases; $v = [-1, -0.78, \dots, 0.78, 1]$

– Approximate activations like we did with weights:

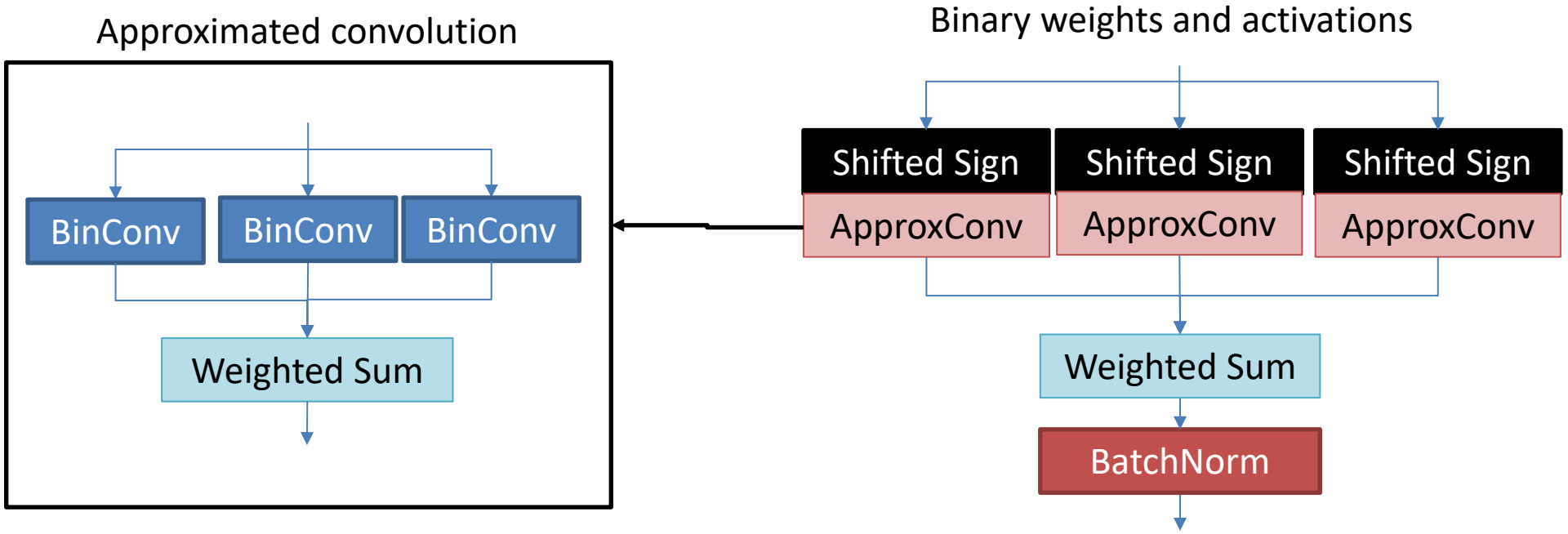
$$A_{\mathrm{R1}} \approx \beta_1 A_{B1} + \beta_2 A_{B2} + \cdots + \beta_N A_{\mathrm{BN}}$$

– $A_{\mathrm{B}i}$ constructed using shift vector v_i:

$$A_{\mathrm{B}i} = \mathrm{clip}(\mathrm{sign}(x + v_i), 0, 1)$$

ABC-Net: ensemble per-layer

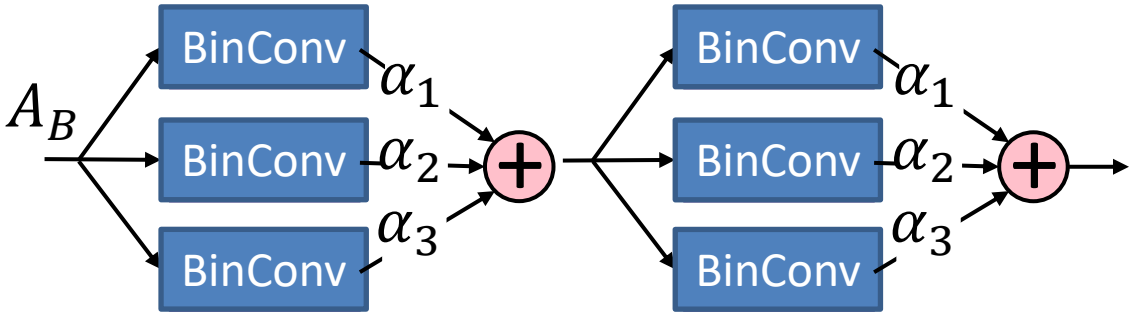
- Convolution becomes: $A_R * W_R \approx \sum_{m=1}^{M} \sum_{n=1}^{N} \alpha_m \beta_n \cdot (W_{Bm} \otimes A_{Bn})$
- Workload increases by $M \times N$ plus scaling multiplications
- Number of weights increases by M times



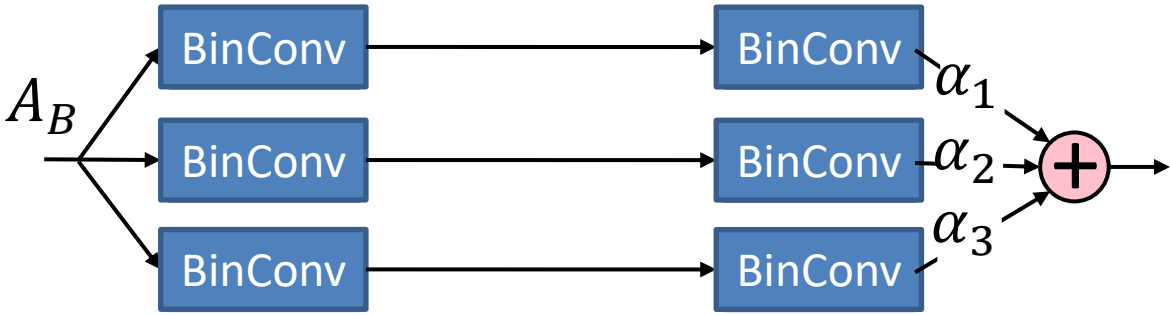
Group-Net: ensemble per-block

- Per-block ensemble
  - Quantization error is accumulated less
- Binary base weights are learned
- Workload increases by M plus scaling multiplications
- Number of weights increases by M times

ABCNet layer-wise expansion

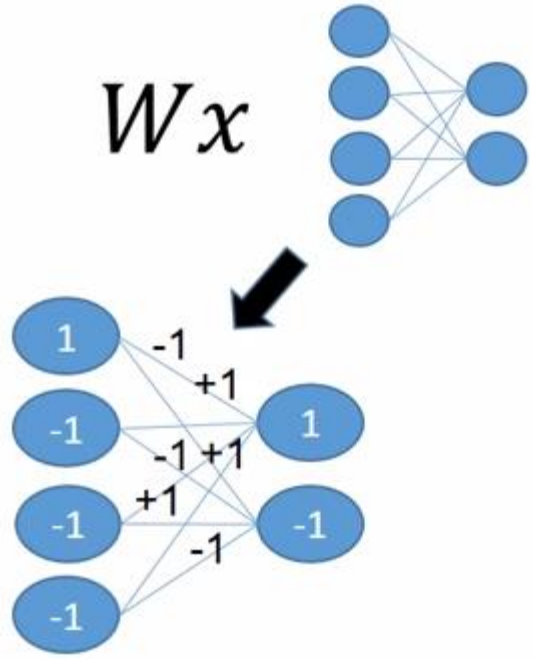


GroupNet group-wise expansion



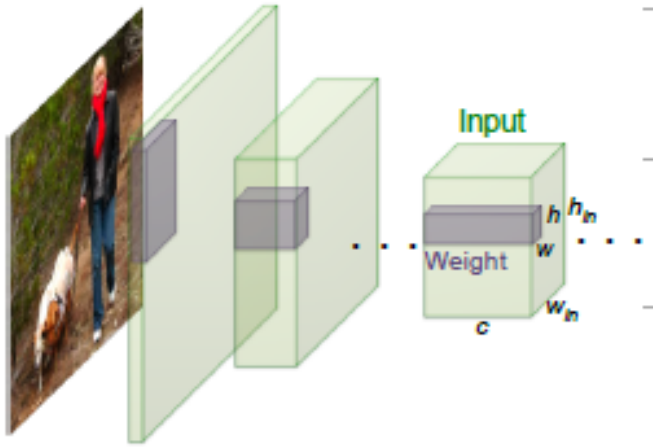
Motivation for Binary/Ternary Neural Networks

- Challenge
  - Run DNN on low power devices
  - Speedup DNNs at run-time
- Main computational bottleneck
  - Multiply-Accumulate (MAC) operation at matrix vector products
- Solution
  - Binary/ternary weights and activations to +/-1 or 0
  - Expensive MAC => cheap XNOR + PopCount



Binary/Ternary Neural Network

- Binary: +1 / -1
- Ternary: +1 / 0/ -1
- Binary/ternary neural network
  - Quantize weight or activation to binary or ternary value
  - Ternary value is better due to better representation



	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	Real-Value Inputs 0.11 -0.21 ... -0.34 / -0.25 0.61 ... 0.52; Real-Value Weights 0.12 -1.2 ... 0.41 / -0.2 0.5 ... 0.68	+ , − , ×	1x	1x	%56.7
Binary Weight	Real-Value Inputs 0.11 -0.21 ... -0.34 / -0.25 0.61 ... 0.52; Binary Weights 1 -1 ... 1 / -1 1 ... 1	+ , −	~32x	~2x	%56.8
BinaryWeight Binary Input (XNOR-Net)	Binary Inputs 1 -1 ... -1 / -1 1 ... 1; Binary Weights 1 -1 ... 1 / -1 1 ... 1	XNOR , bitcount	~32x	~58x	%44.2

BNN on ResNet (Open Source on MxNet)

- BMXNet

	Architecture	Test Accuracy (Binary/Full Precision)	Model Size (Binary/Full Precision)
MNIST	Lenet	0.97/0.99	206kB/4.6MB
CIFAR-10	ResNet-18	0.86/0.90	1.5MB/44.7MB

Table 1: Classification test accuracy of binary and full precision models trained on MNIST and CIFAR-10 dataset. No pre-training or data augmentation was used.

Listing 1: LeNet

```
def get_lenet():
 data = mx.symbol.Variable('data')
 # first conv layer
 conv1 = mx.sym.Convolution(...)
 tanh1 = mx.sym.Activation(...)
 pool1 = mx.sym.Pooling(...)
 bn1 = mx.sym.BatchNorm(...)
 # second conv layer
 conv2 = mx.sym.Convolution(...)
 bn2 = mx.sym.BatchNorm(...)
 tanh2 = mx.sym.Activation(...)
 pool2 = mx.sym.Pooling(...)
 # first fullc layer
 flatten = mx.sym.Flatten(...)
 fc1 = mx.symbol.FullyConnected(..)
 bn3 = mx.sym.BatchNorm(...)
 tanh3 = mx.sym.Activation(...)
 # second fullc
 fc2 = mx.sym.FullyConnected(...)
 # softmax loss
 lenet = mx.sym.SoftmaxOutput(...)
 return lenet
```

Listing 2: Binary LeNet

```
def get_binary_lenet():
 data = mx.symbol.Variable('data')
 # first conv layer
 conv1 = mx.sym.Convolution(...)
 tanh1 = mx.sym.Activation(...)
 pool1 = mx.sym.Pooling(...)
 bn1 = mx.sym.BatchNorm(...)
 # second conv layer
 ba1 = mx.sym.QActivation(...)
 conv2 = mx.sym.QConvolution(...)
 bn2 = mx.sym.BatchNorm(...)
 pool2 = mx.sym.Pooling(...)
 # first fullc layer
 flatten = mx.sym.Flatten(...)
 ba2 = mx.symbol.QActivation(..)
 fc1 = mx.symbol.QFullyConnected(..)
 bn3 = mx.sym.BatchNorm(...)
 tanh3 = mx.sym.Activation(...)
 # second fullc
 fc2 = mx.sym.FullyConnected(...)
 # softmax loss
 lenet = mx.sym.SoftmaxOutput(...)
 return lenet
```

Listing 3: Baseline xnor GEMM Kernel

```
void xnor_gemm_baseline_no_omp(int M, int N, int K,
                               BINARY_WORD *A, int lda,
                               BINARY_WORD *B, int ldb,
                               float *C, int ldc){
  for (int m = 0; m < M; ++m) {
    for (int k = 0; k < K; k++) {
      BINARY_WORD A_PART = A[m*lda+k];
      for (int n = 0; n < N; ++n) {
        C[m*ldc+n] += __builtin_popcountl(~(A_PART ^ B[k*ldb+n]));
      }
    }
  }
}
```

A single assembly command to support popcount in x86 and ARM v7
Leverage processor cache hierarchies by blocking and packing the data, use unrolling techniques and OpenMP for parallelization

Binary Ensemble Neural Network: More Bits per Network or More Networks per Bit?

Table 6. Comparison with state-of-the-arts on ImageNet using ResNet-18 (W-weights, A-activation)

Method	W	A	Top-1
Full-Precision DNN [27, 43]	32	32	69.3%
XNOR-Net [50]	1	1	48.6%
ABC-Net [43]	1	1	42.7%
BNN [31, 50]	1	1	42.2%
BENN-SB-3, Bagging (ours)	1	1	**53.4%**
BENN-SB-3, Boosting (ours)	1	1	**53.6%**
BENN-SB-6, Bagging (ours)	1	1	**57.9%**
BENN-SB-6, Boosting (ours)	1	1	**61.0%**

How to Train BNN with High Accuracy

- Learning rate
  - Full precision: start from 0.01 or 0.05 with BN
    - Too large for BNN, resulting in learning curve fluctuation due to sign change
  - BNN: 0.0001
- Scale factor
  - XNOR-net adopts a real value scale factor during binarization
    - Too complex and inefficient
  - Use PReLU instead
    - Move scale factors of weight to activation function
- Regularizer
  - Move weight to +/-1 instead of 0

binarization

$$w^b = \begin{cases} +1 & w \geq 0, \\ -1 & otherwise. \end{cases}$$

$$J(\mathbf{W}, \mathbf{b}) = L(\mathbf{W}, \mathbf{b}) + \lambda \sum_{l=1}^{L} \sum_{i=1}^{N_l} \sum_{j=1}^{M_l} (1 - (\mathbf{W}_{l,ij})^2)$$

- Last layer
  - binarization could lead to significant accuracy drop
    - No binarization is usually adopted
  - Add a learnable scale layer after this final binarized layer
    - Initialized to 0.001,
    - before sending results to softmax function

Table 4: Comparison between our method and DoReFa-net after binarizing the last layer. For fair comparison, activations of our method are also binarized with 2 bits.

Methods	Last layer	Scale layer	Comp. rate	Accuracy(%)
DoReFa-net	Full	No	10.3×	– /47.1
DoReFa-net	Binary	–	31.4×	– /40.3
BNN	Binary	Yes	27.1×	70.4/45.8

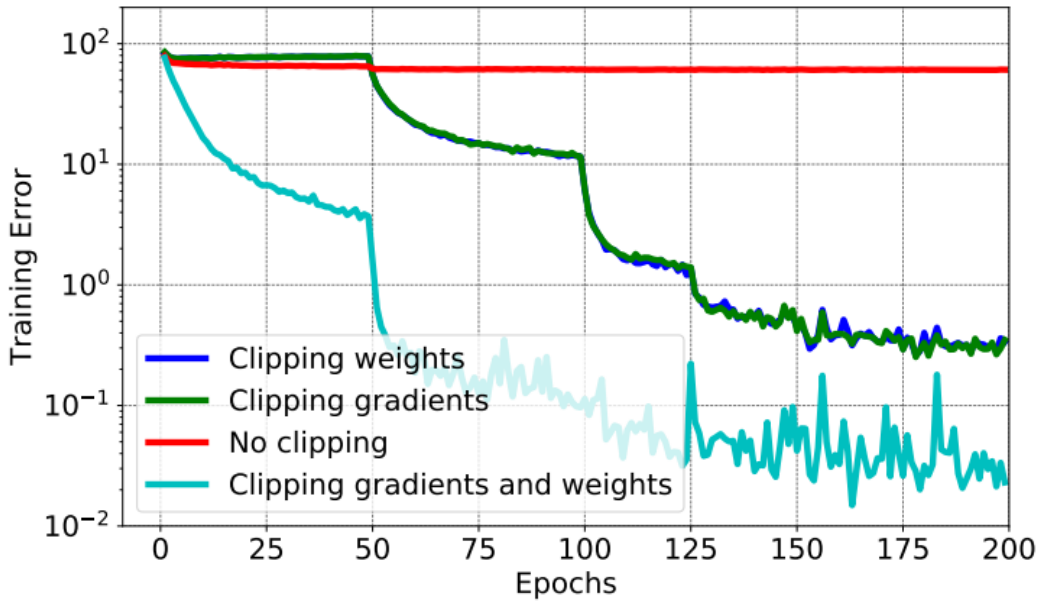
Table 5: Results before and after multiple binarizations. A denotes activation.

Methods	Bits of A	Comp. rate	Accuracy(%)
BNN	1	27.1×	64.6/39.3
BNN	2	27.1×	70.4/45.8
BNN (expand)	2	23.6×	75.6/51.4

Why BNN Training is Slow?

- STE does not affect the training speed
- Gradient and weight clipping are the killer
- Best practices: two stage training
  - (1) using vanilla STE in the first stage with higher learning rates
  - and (2) turning clippings back on when the accuracy stops improving by reducing learning rate

  - Or
  - Use pretrained model as BNN initialization
  - STE + gradient clipping



that not clipping weights when learning rates are large can completely halt the optimisation (red curve in Figure 5). On the other hand, using vanilla STE brings the training speed back on par with the non-binary model.

Figure 5: Impact of gradient and weight clipping on convergence speed of binary VGG-10 with large learning rates (0.1).

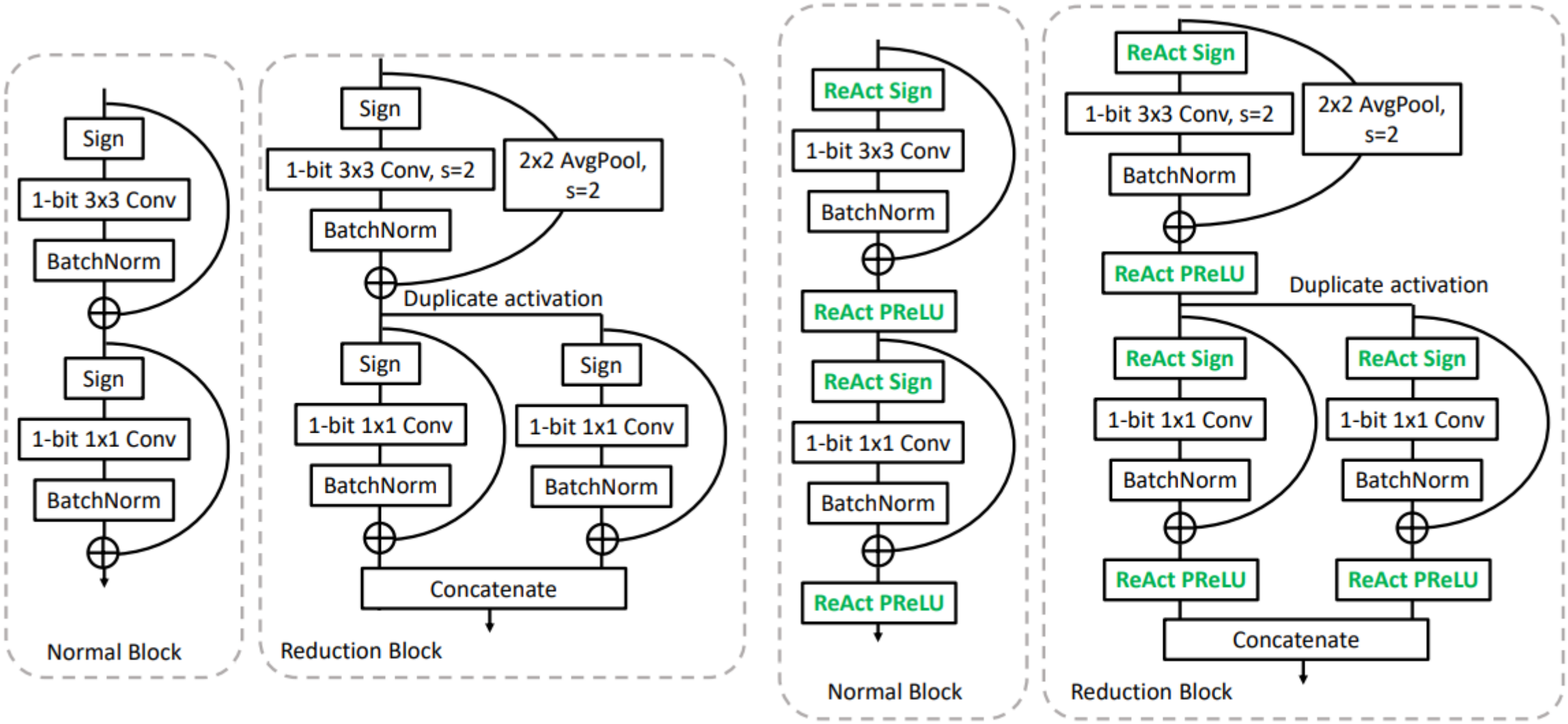
Table 5: Training binary models using pre-trained full-precision models for CIFAR-10 (ResNet-18 and VGG-10) and ImageNet (AlexNet-like) datasets.

	Binarisation	Best Validation Accuracy	Test Accuracy
Binary ResNet-18	end-to-end	94.40% (in epoch 457)	91.16%
	from full-precision	93.60% (in epoch **17**)	91.18%
Binary VGG-10	end-to-end	89.76% (in epoch 391)	89.18%
	from full-precision	90.16% (in epoch **24**)	89.32%
Binary AlexNet-like	end-to-end	51.98% (in epoch 88)	—
	from full-precision	51.85% (in epoch **30**)	—

BNN as the fine tuning step of full precision models

- . For efficient training of Binary models we recommend:
  - (1) using ADAM for optimising the objective,
  - (2) not using early stopping,
  - (3) splitting the training into two stages,
  - (4) removing gradient and weight clipping in the first stage and
  - (5) reducing the averaging rate in Batch Normalisation layers in the second stage.

ReActNet: Towards Precise Binary Neural Network with Generalized Activation Functions



(a) Proposed baseline network block

(b) Proposed ReActNet block

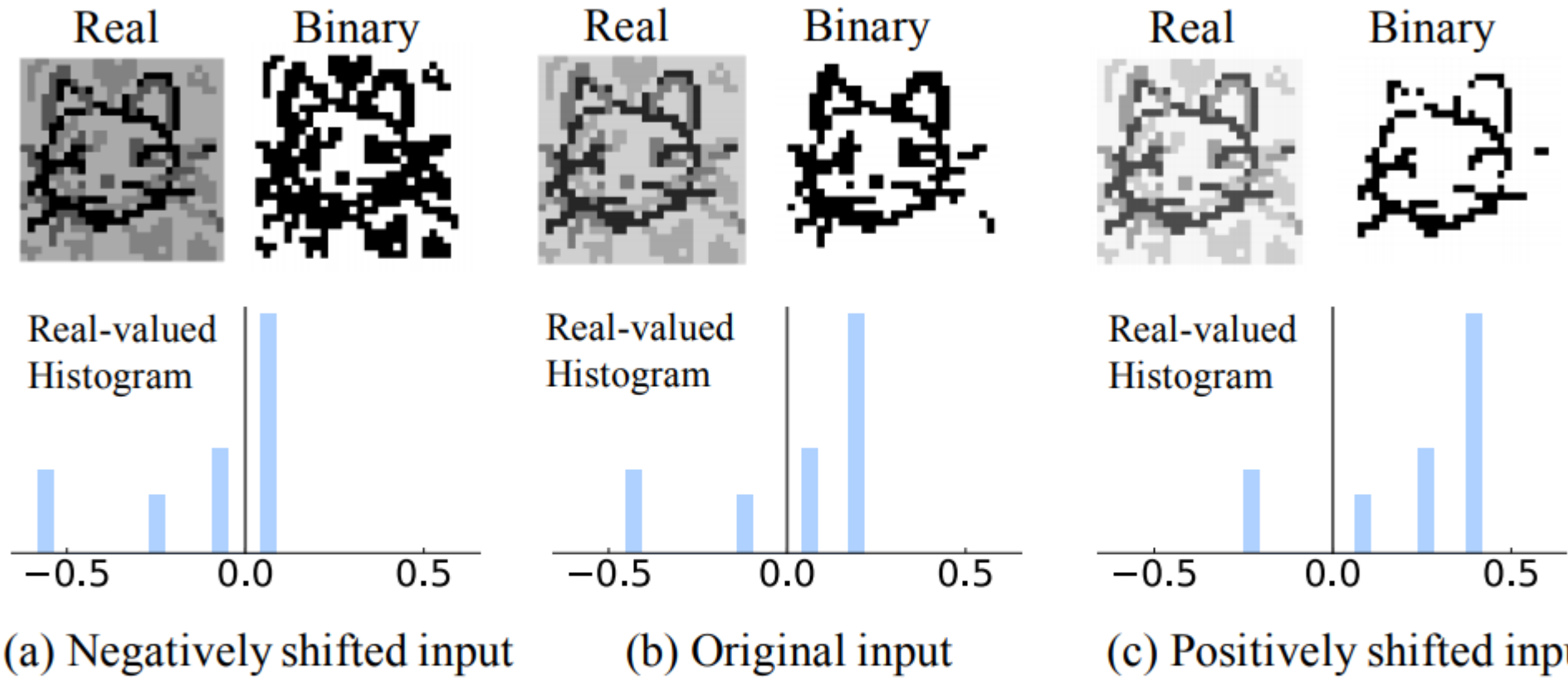
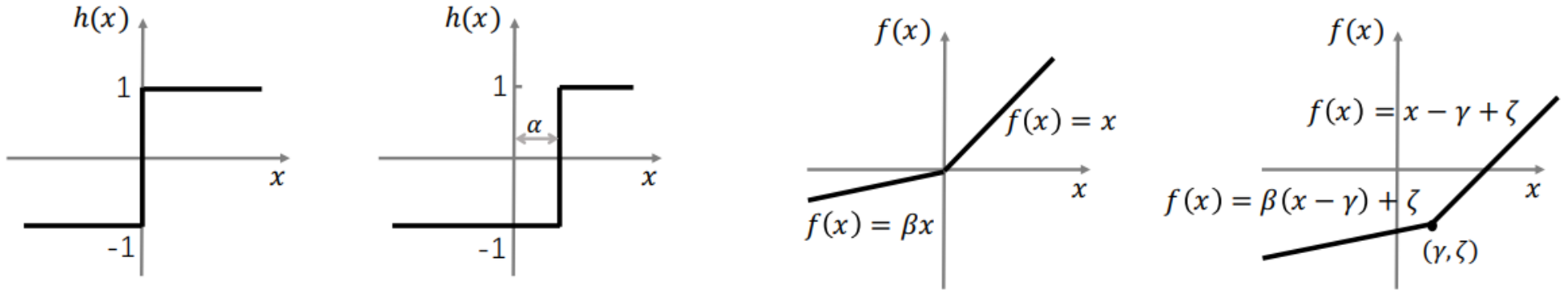


Fig. 3. An illustration of how distribution shift affects feature learning in binary neural networks. An ill-shifted distribution will introduce (a) too much background noise or (c) too few useful features, which harms feature learning.



(a) Sign vs. RSign

(b) PReLU vs. RPReLU

Fig. 4. Proposed activation functions, RSign and RPReLU, with learnable coefficients and the traditional activation functions, Sign and PReLU.

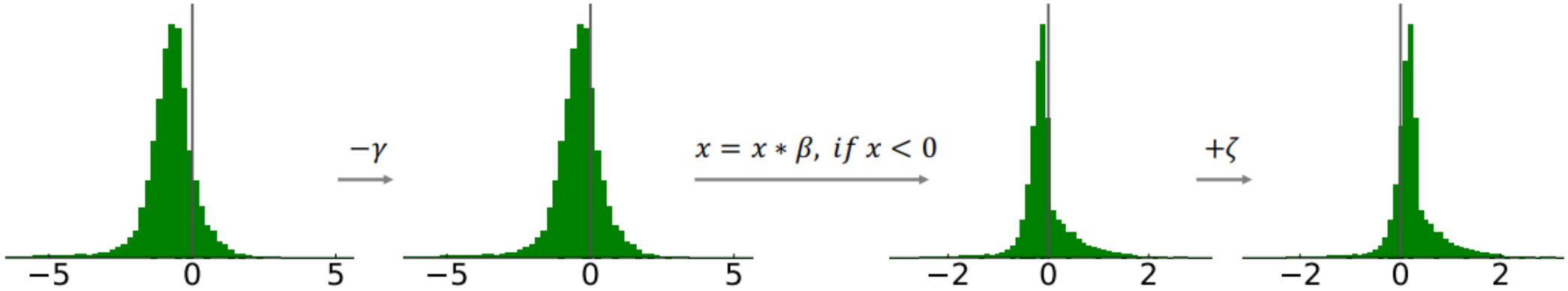


Fig. 5. An explanation of how proposed RPReLU operates. It first moves the input distribution by $-\gamma$, then reshapes the negative part by multiplying it with β and lastly moves the output distribution by ζ.

Methods	Bitwidth (W/A)	Acc(%) Top-1	Binary Methods	FLOPs ($\times 10^8$)	Acc(%) Top-1
BWN [7]	1/32	60.8	BNNs [7]	1.31	42.2
TWN [18]	2/32	61.8	CI-BCNN [35]	1.54	56.7
INQ [40]	2/32	66.0	CI-BCNN* [35]	1.63	59.9
TTQ [42]	2/32	66.6	PCNN [11]	1.63	57.3
SYQ [10]	1/2	55.4	XNOR-Net [28]	1.67	51.2
HWGQ [5]	1/2	59.6	Trained Bin [36]	–	54.2
LQ-Nets [37]	1/2	62.6	Bi-RealNet-18 [23]	1.63	56.4
DoReFa-Net [41]	1/4	59.2	Bi-RealNet-34 [23]	1.93	62.2
Ensemble BNN [43]	(1/1) × 6	61.1	Bi-RealNet-152 [21]	6.15	64.5
Circulant CNN [20]	(1/1) × 4	61.4	Real-to-Binary Net [3]	1.65	65.4
Structured BNN [45]	(1/1) × 4	64.2	MeliusNet29 [2]	2.14	65.8
Structured BNN* [45]	(1/1) × 4	66.3	MeliusNet42 [2]	3.25	69.2
ABC-Net [19]	(1/1) × 5	65.0	MeliusNet59 [2]	5.32	70.7
Our ReActNet-A	1/1	–	–	**0.87**	**69.4**
Our ReActNet-B	1/1	–	–	**1.63**	**70.1**
Our ReActNet-C	1/1	–	–	**2.14**	**71.4**

Table 1. Comparison of the top-1 accuracy with state-of-the-art methods. The left part presents quantization methods applied on ResNet-18 structure and the right part are

Network	Top-1 Acc(%)
Baseline network † *	58.2
Baseline network †	59.6
Proposed baseline network *	61.1
Proposed baseline network	62.5
Proposed baseline network + PReLU	65.5
Proposed baseline network + RSign	66.1
Proposed baseline network + RPReLU	67.4
ReActNet-A (RSign and RPReLU)	69.4
Corresponding real-valued network	72.4

Table 2. The effects of different components in ReActNet on the final accuracy. († denotes the network not using the concatenated blocks, but directly binarizing the downsampling layers instead. * indicates not using the proposed distributional loss during training.)

Tool Support in Pytorch

- Post training quantization
  - https://pytorch.org/tutorials/prototype/pt2e_quant_ptq.html
- Quantization aware training
  - https://pytorch.org/tutorials/prototype/pt2e_quant_qat.html
- Self defined quantizer
  - https://pytorch.org/tutorials/prototype/pt2e_quantizer.html