Lecture Series on
Hardware for Deep Learning

Part 4: Reducing the Complexity

Dr. Adam Teman
EnICS Labs, Bar-Ilan University

28 April 2020

EnICS
Emerging Nanoscaled Integrated Circuits and Systems Labs

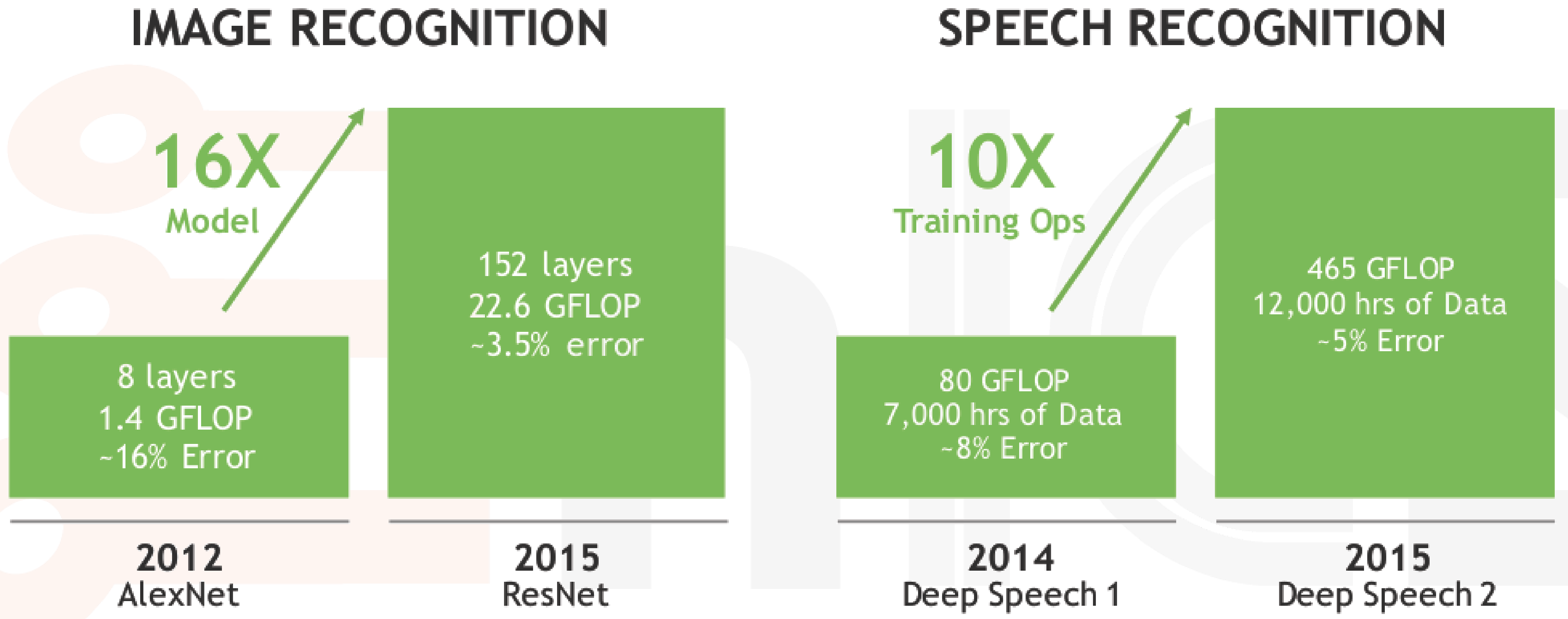
Tradition of Excellence
Bar-Ilan University
אוניברסיטת בר-אילן

Outline


Motivation
Lightweight Models
Reducing Precision
Aggressive Quantization
Pruning and Deep Compression

Motivation

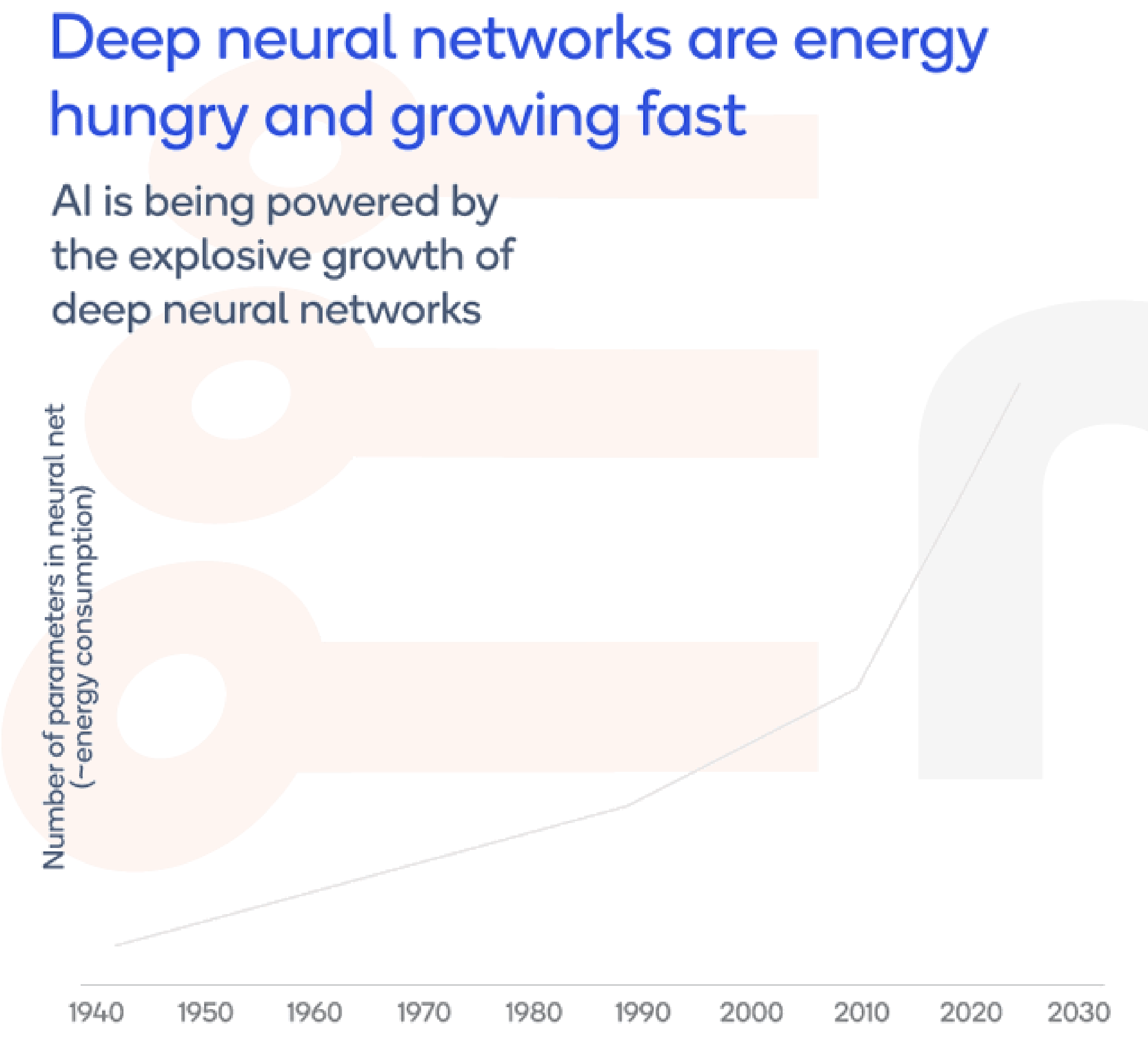
Models are Getting Larger



Microsoft Baidu

Dally, NIPS'2016 workshop on Efficient Methods for Deep Neural Networks

Explosion in size, complexity, energy



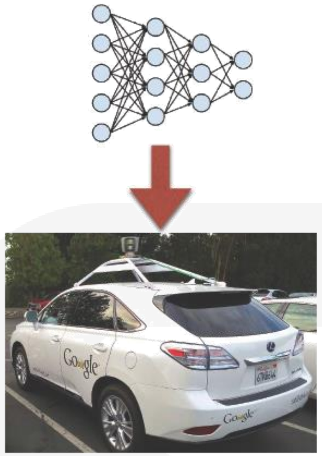
Source: Qualcomm

Network	Model size (MB)	GFLOPS
AlexNet*	233	0.7
VGG-16*	528	15.5
VGG-19*	548	19.6
ResNet-50*	98	3.9
ResNet-101*	170	7.6
ResNet-152*	230	11.3
GoogleNet#	27	1.6
InceptionV3#	89	6
MobileNet#	38	0.58
SequeezeNet#	30	0.84

*: Characterization and Benchmarking of Deep Learning, Natalia Vassilieva
#: https://github.com/albanie/convnet-burden

Big Three Challenges

- **First Challenge: Model Size**
  - Hard to distribute large models through over-the-air update
- **Second Challenge: Speed**
  - Such long training time limits ML researcher's productivity
- **Third Challenge: Energy Efficiency**
  - AlphaGo: 1920 CPUs and 280 GPUs, $3000 electric bill per game
  - On mobile: drains battery
  - On data-center: increases TCO



	Error rate	Training time
ResNet18:	10.76%	2.5 days
ResNet50:	7.02%	5 days
ResNet101:	6.21%	1 week
ResNet152:	6.16%	1.5 weeks

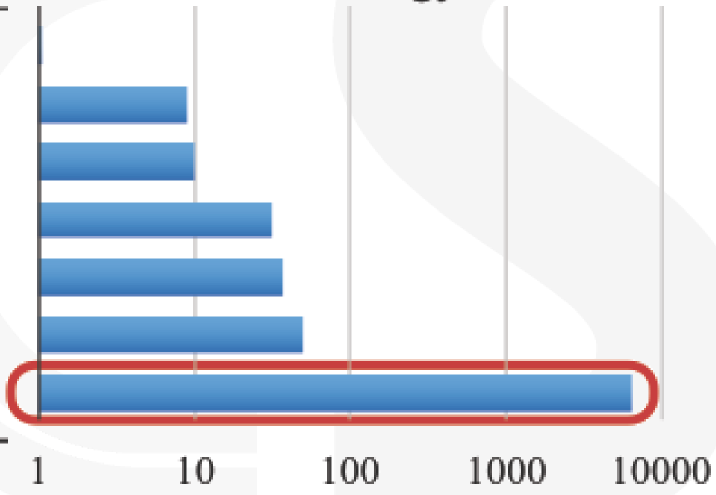
Source: Han

Where is the Energy Consumed?

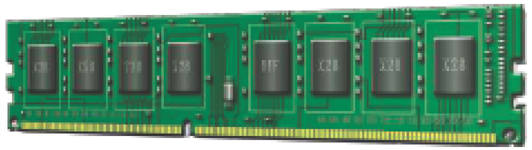
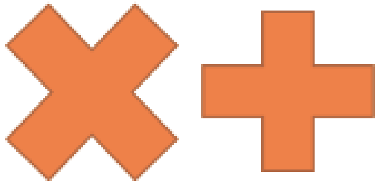
- Larger model
- More memory references
- More energy

Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	**640**

Relative Energy Cost



- How can we make our models more energy efficient?

1  = 1000 

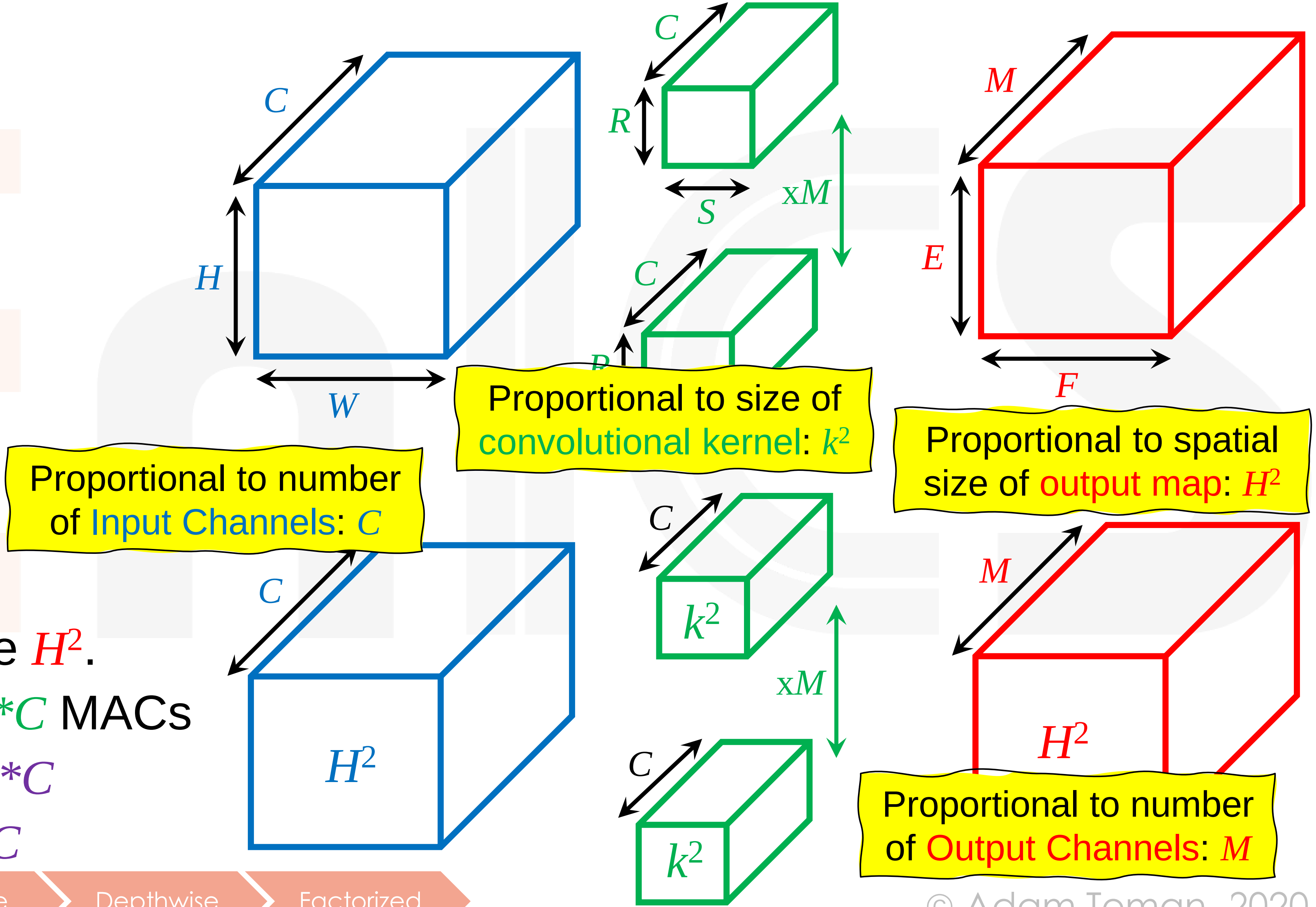
This image is in the public domain

Source: Han

Lightweight Models

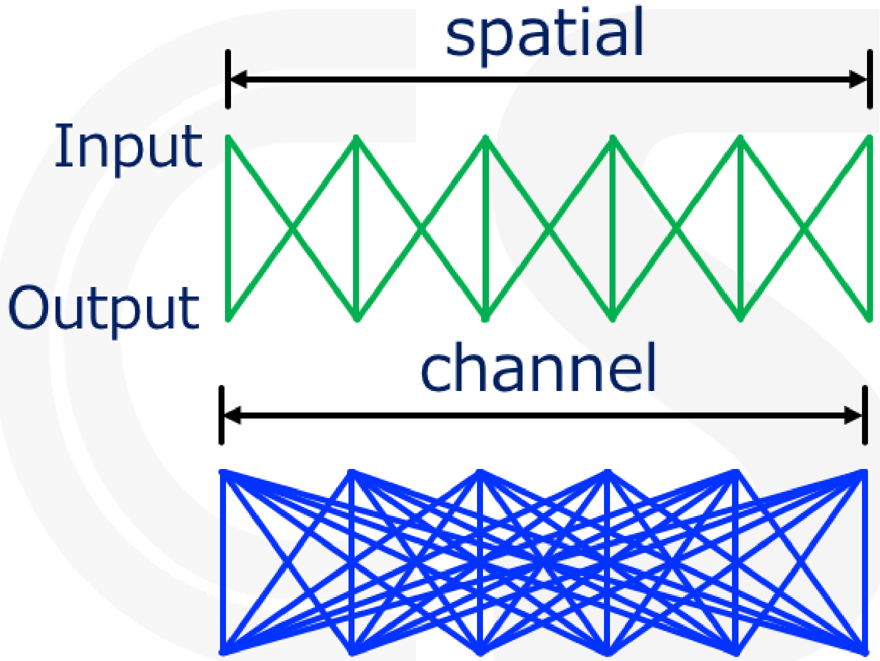
Reminder: Standard Convolution

- **Layer sizes:**
  - Input fmap: $HxWxC$
  - Filter size: $RxSxC$
  - Output size: $ExFxM$
- **A bit simplified:**
  - Assume: $H=W=E=F$
  - Assume: $R=S=k$
- **Cost of convolution:**
  - M output maps of size H^2.
  - Each one requires k^2*C MACs
  - Total MACs: $M*H^2*k^2*C$
  - Total Weights: $M*k^2*C$



Spatial and Channel Connectivity

- **To visualize the connectivity complexity, we can use a pair of illustrations**
  - For a 3x3 kernel, looking at one spatial dimension (e.g., one row), the connectivity between the input activation and output fmap looks as follows:
  - And across channels, each input channel is connected to each output channel, so we get:

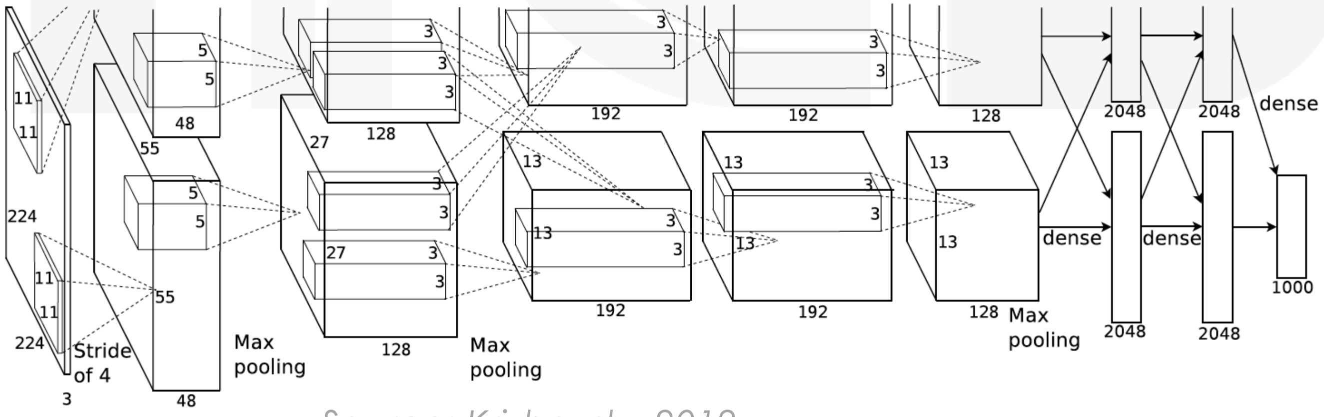


Source: Yusuke Uchida

- **So we see that for convolutions:**
  - **Spatially**, the inputs and outputs are connected ***locally***.
  - Across **channels**, the inputs and outputs are ***fully connected***.

Group Convolutions

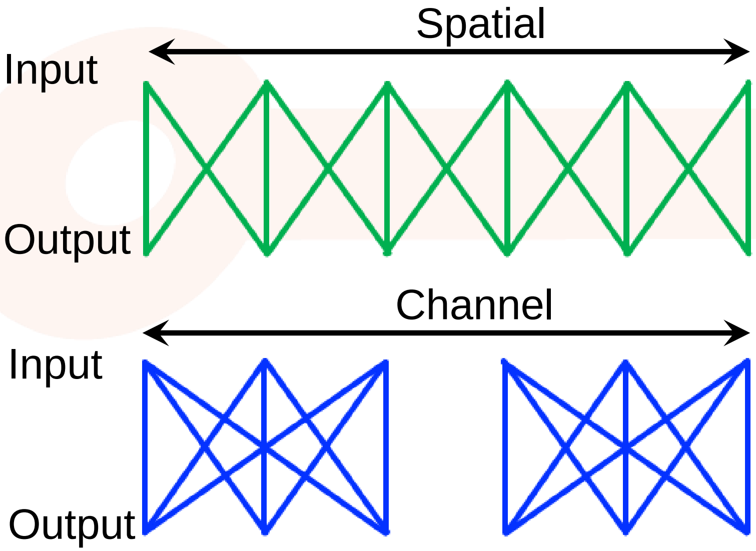
- **Observation:**
  - The more filters in a layer (M), the more *intermediate features* we learn.
- **Problem:**
  - This leads to a lot of operations (Total MACs: $M*H^2*k^2*C$)
- **Grouped Convolutions:**
  - Reduce the number of operations by dividing the input into several groups.
  - Essentially, we can learn different features through different routes.
  - First used by **AlexNet** to split a network onto two GPUs.

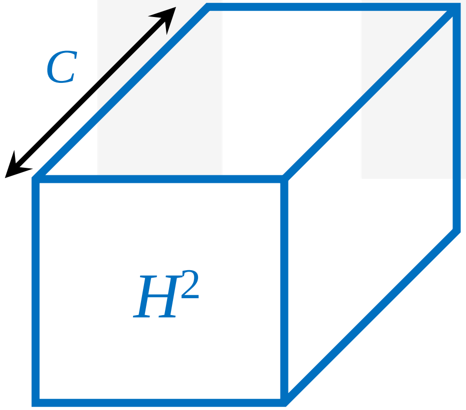


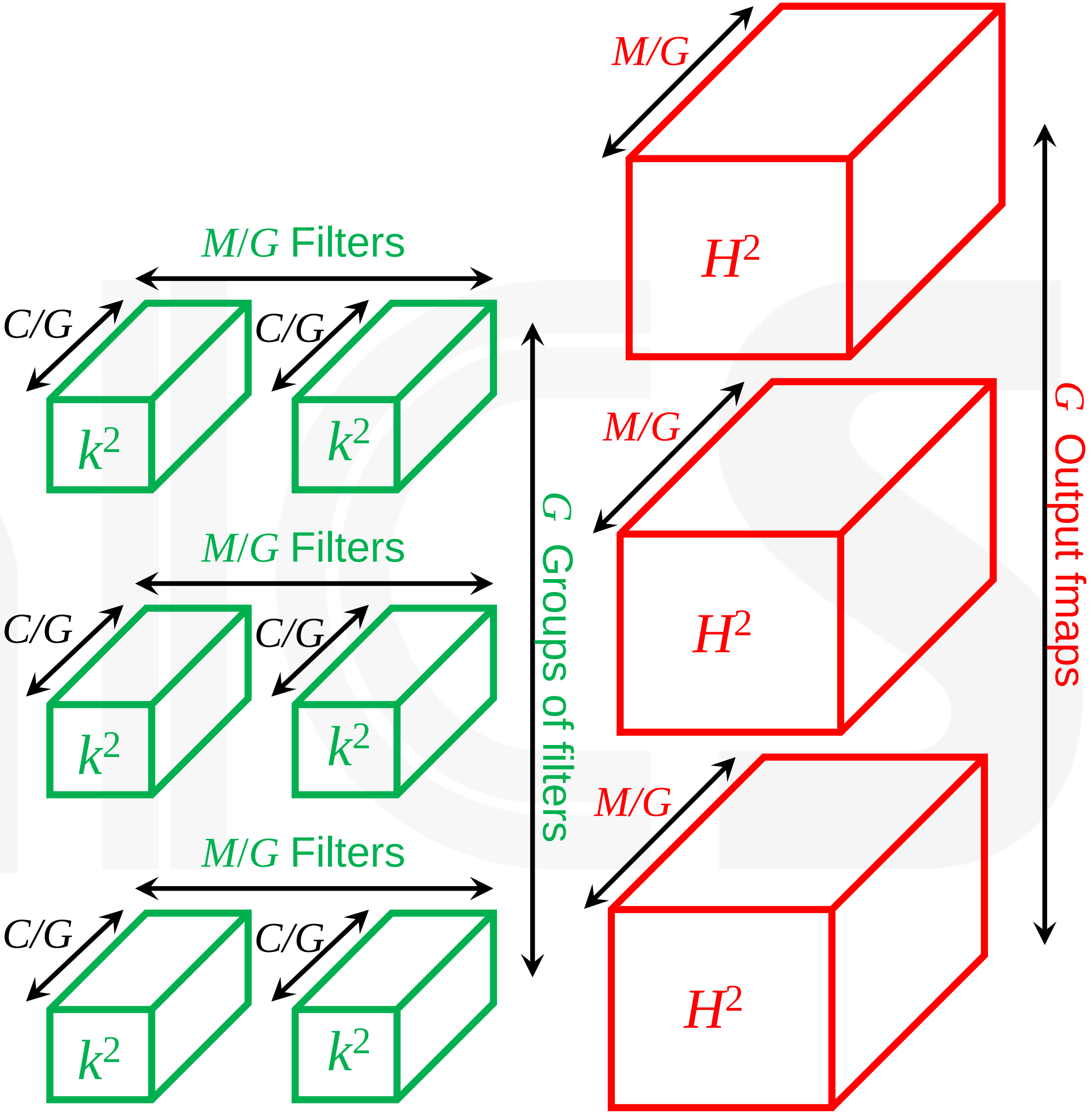
Source: Krizhevsky 2012

Group Convolutions

- **So now we have:**
  - G groups of M/G filters
  - G output fmaps of M/G depth
  - Total MACs: $G*(M/G*H^2*k^2*C/G)$
  - That's a reduction of $1/G$.
- **Visualization: Gconv 3x3**



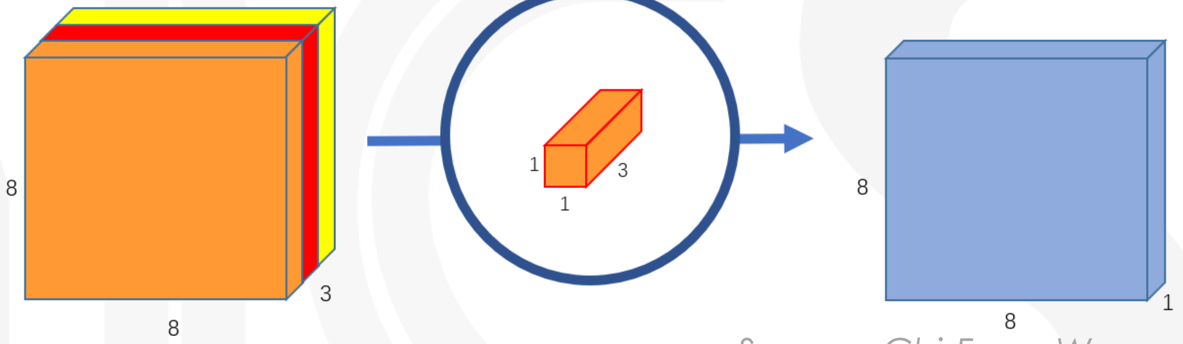




Source: Yusuke Uchida

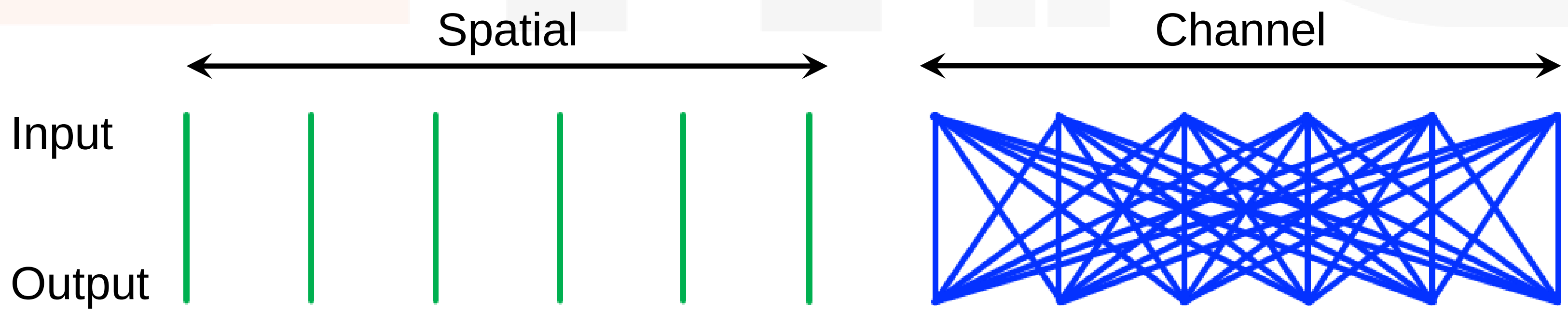
Pointwise (1x1) Convolution

- **Problem:**
  - Convolving a large filter over many input channels is expensive ($k^2 * C$)
- **Solution:**
  - Merge channels with a $1 \text{x} 1 \text{x} C$ filter
  - Use M filters to get the desired input channel depth
  - Total cost: $M * H^2 * C$.



Source: Chi-Feng Wang

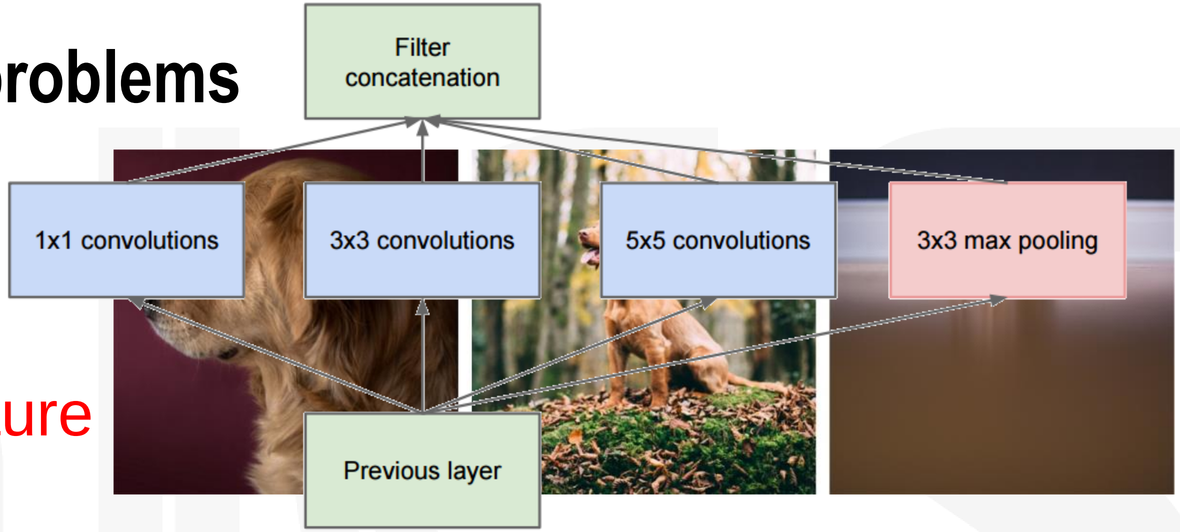
- **This “blends” information across channels:**

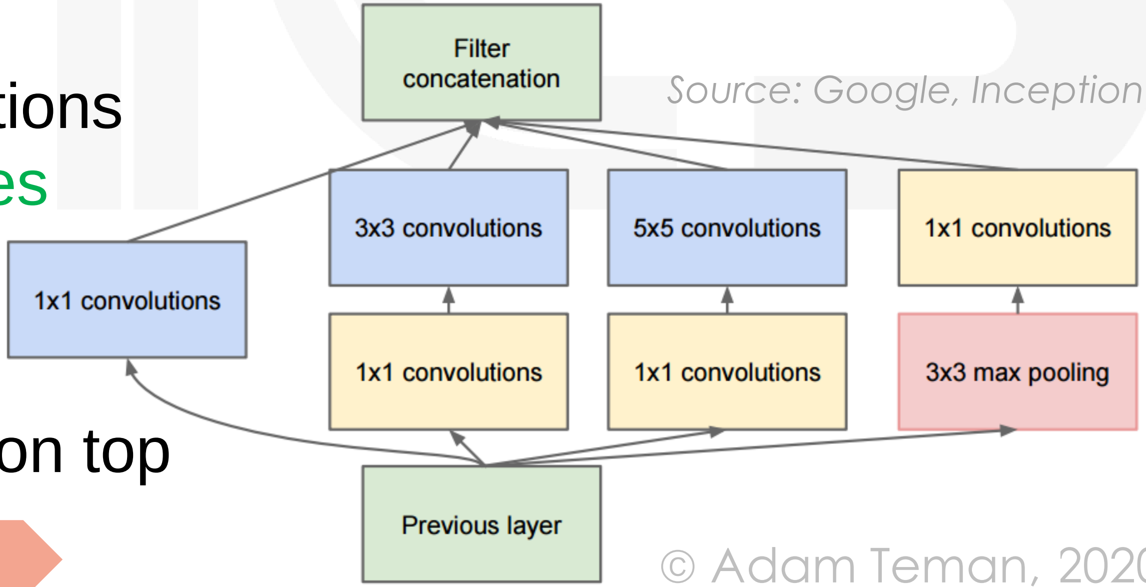


Source: Yusuke Uchida

Example: Inception (GoogLeNet)

- **GoogLeNet was intended to solve three problems**
  - Previous models kept going deeper → computationally expensive
  - Variation in location of information →Need several filter sizes for each feature
  - Deep networks are prone to overfitting
- **Solution: Go *Wider***
  - Use an **"Inception Layer"** to split activations into several routes with different filter sizes
- **But this is computationally expensive**
  - So reduce dimensionality with 1x1 convolution and then stack a larger filter on top





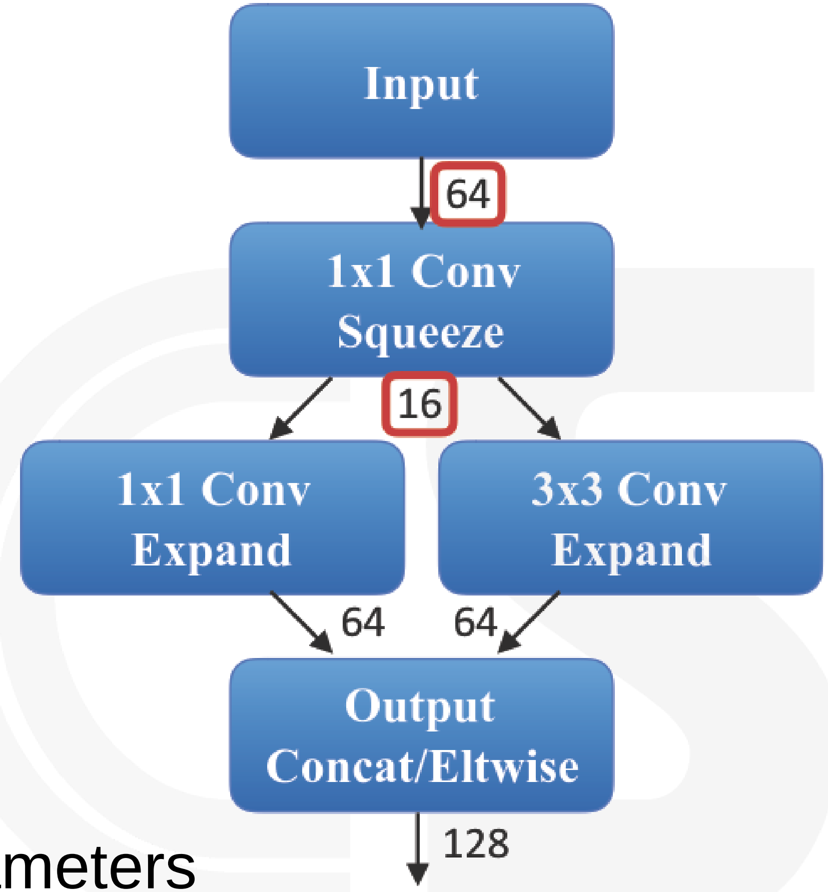
Source: Google, Inception v

Example: SqueezeNet

- **The “Fire Module” of SqueezeNet:**
  - Uses 1x1 convolutions to reduce channel depth
  - Uses 1x1 and 3x3 convolutions to expand it back

Two other interesting concepts in SqueezeNet:

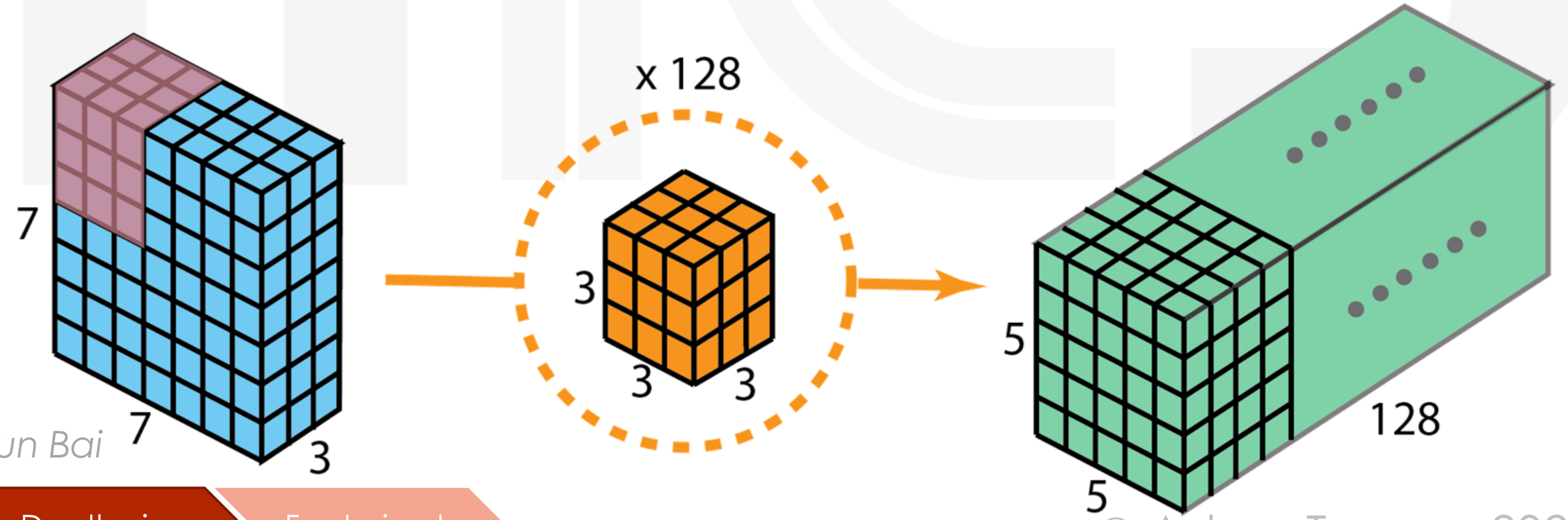
- **Downsampling**
  - Use pooling with a stride of ½ late in the network.
  - This provides late convolution layers with many parameters
- **No fully connected layers**
  - Finish with N channels for N classification categories
  - Use average pooling on a channel for a classification score



Source: Song Han

Depthwise Convolutions

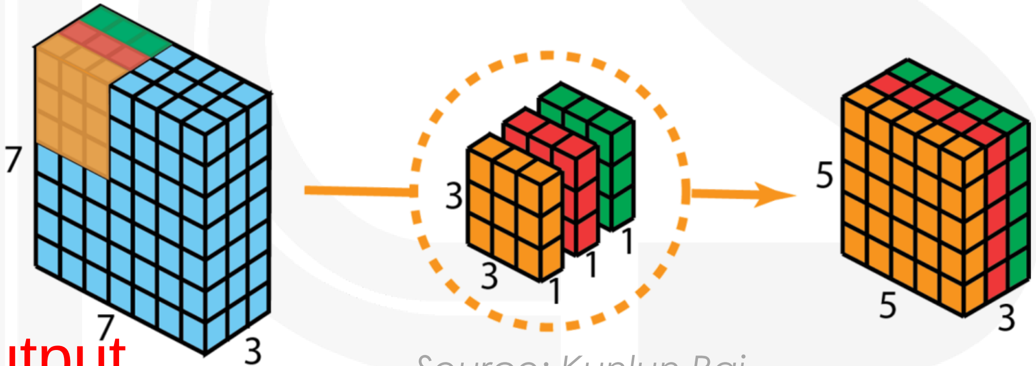
- **A popular way of doing low cost convolutions is to combine *Group Convolutions* with *Pointwise Convolutions*.**
- **Let's start by looking at a standard convolution:**
  - Starting with an input of $HxWxC$ we want to arrive at an output of $ExFxM$.
  - The standard approach is to use M filters with a depth of C.
  - For example, a 7x7x3 input to a 5x5x128 output needs 128 3x3x3 filters.
  - Total MACs: 86,400
  - Weights: 3,456



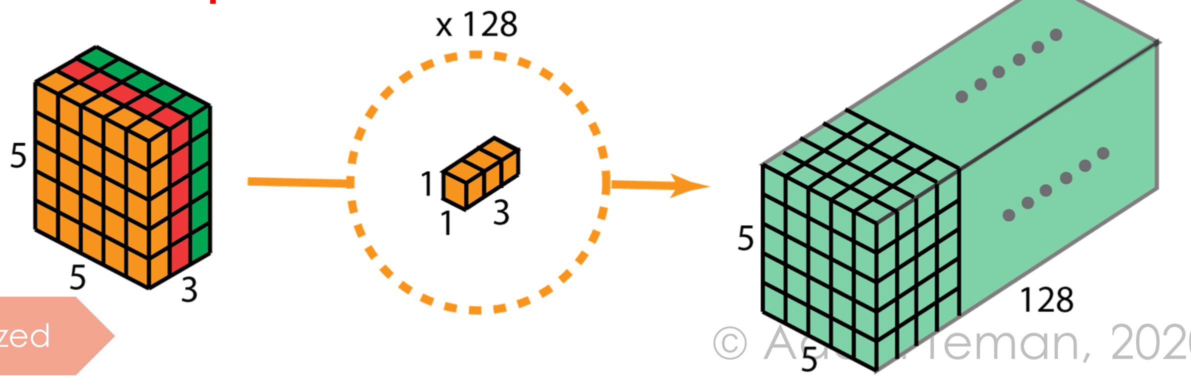
Source: Kunlun Bai

Depthwise Convolutions

- **Instead let's make a group convolution with C groups:**
  - C filters of kxkx1.
  - Each filter is applied to one input channel, providing one output fmap.
  - Concatenating these we get an output of ExFxC.
  - In our example, 3 3x3x1 filters, 5x5x3 output
- **Now use a pointwise (1x1) convolution:**
  - M filters of 1x1xC.
  - Provides the desired output of ExFxM.
  - In our example, 128 1x1x3 filters, 5x5x128 output
- **How much did it cost?**
  - Total MACs: 16,675 (-80%)
  - Total weights: 411 (-90%)



Source: Kunlun Bai



Example: MobileNet

- **Introduced by Google in 2017**
  - Applies Batch Normalization and ReLU after each Depthwise Convolution
  - Better accuracy than **VGG-16** with 97% fewer weights and 97% fewer MACs

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1 Conv / s1	$3 \times 3 \times 512$ dw $1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$ $14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

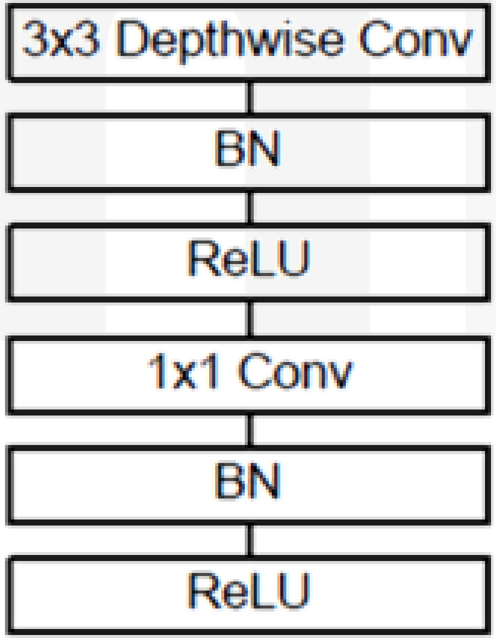


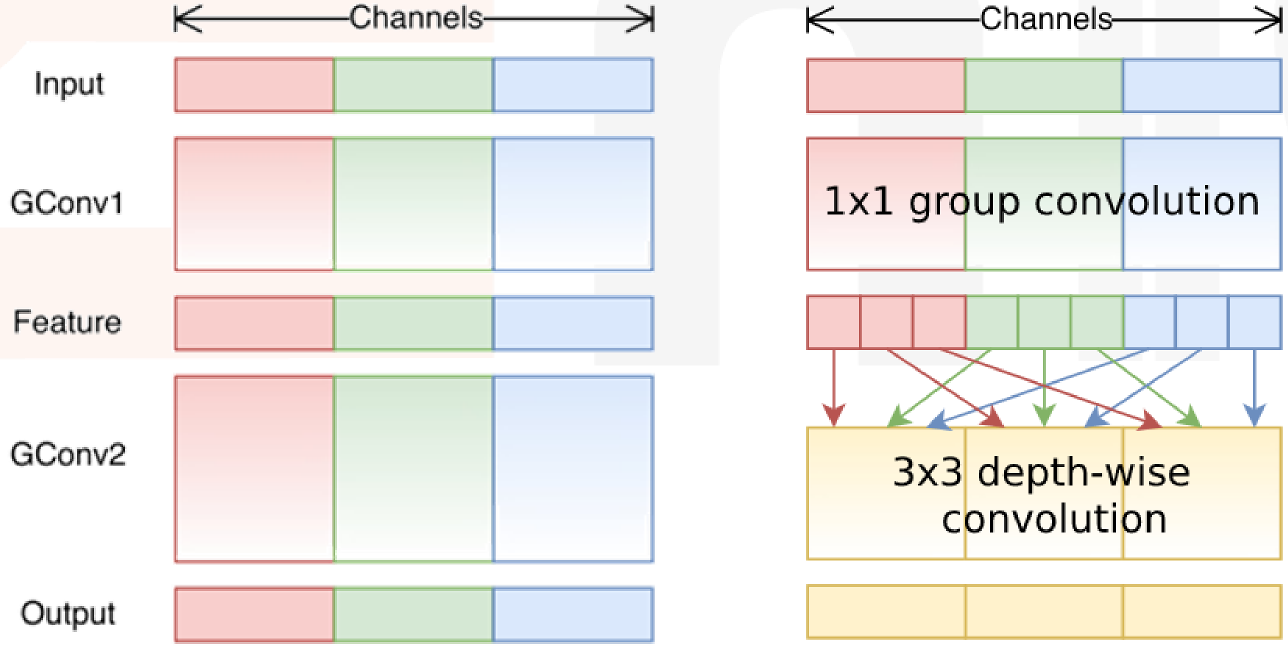
Table 8. MobileNet Comparison to Popular Models

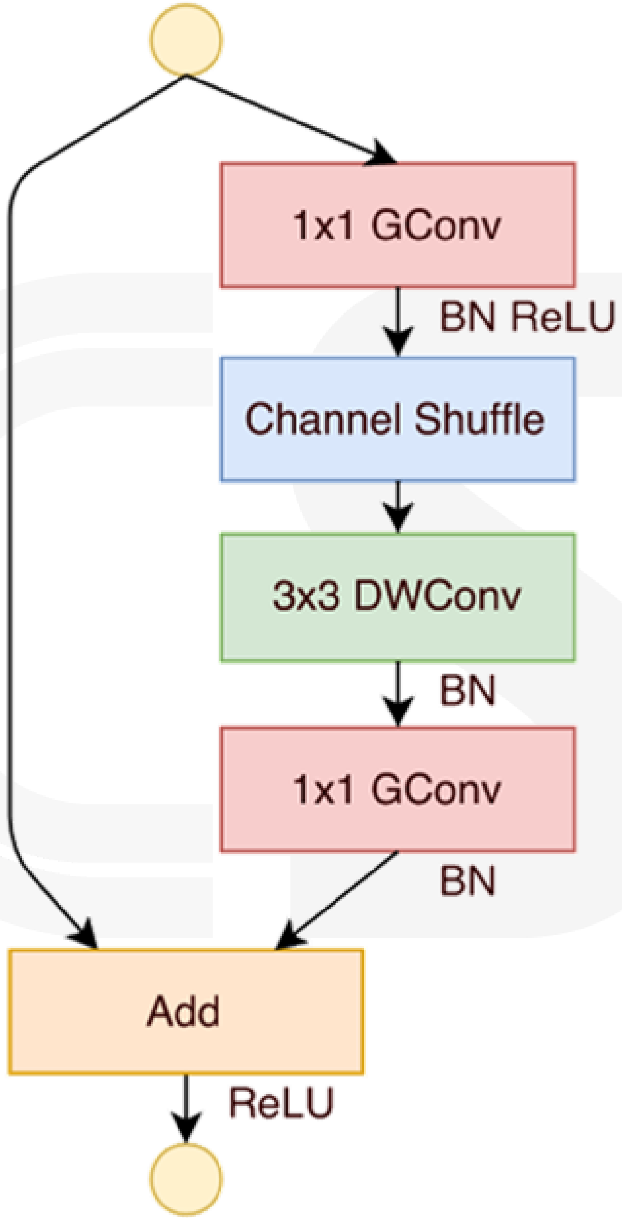
Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Source: Google, MoblieNets

Example: ShuffleNet

- **Apply a "Channel Shuffle"**
  - 1x1 Group Convolution and shuffle the outputs
- **Also use Depthwise Convolutions and Residuals**
- **Outperforms MobileNet**

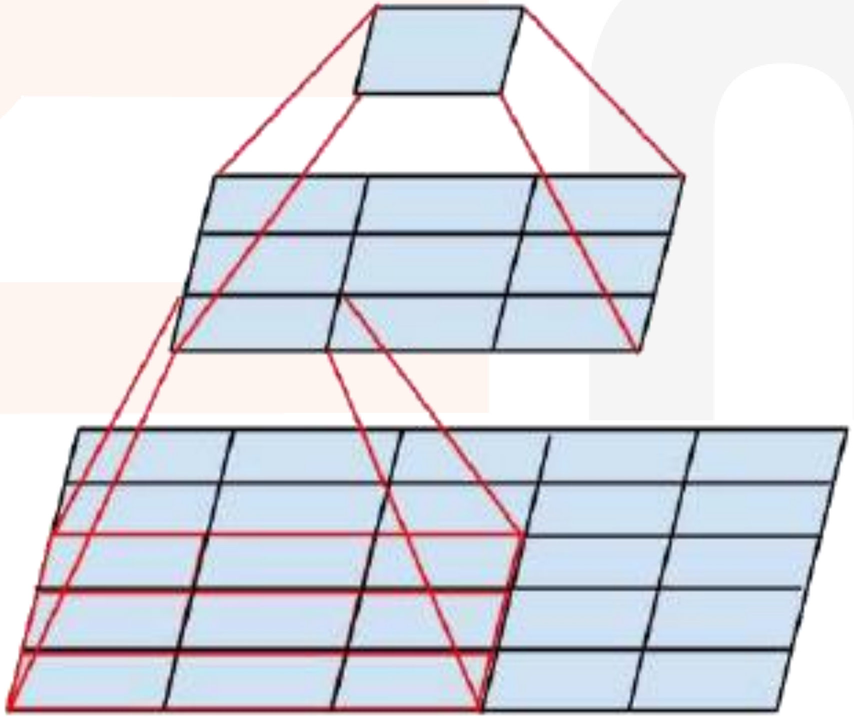


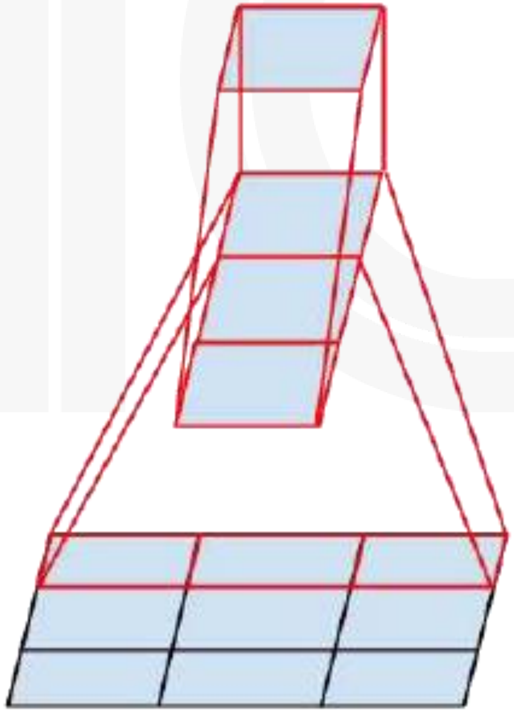


Source: Zhang, et al, ShuffleNet

Factorized (Stacked) Convolutions

- **Reduce the number of weights using two smaller filters:**
  - **VGG**: two 3x3 filters (18 weights) replace one 5x5 filter (25 weights)
  - **Inception v2**: *1xn* and *nx1* filters (2*n* weights) replace *nxn* filter (n^2 weights)
    For example: 3x1 and 1x3 filters (6 weights) replace 3x3 filter (9 weights)





Source: Sik-Ho Tsang

Reducing Precision

Taxonomy

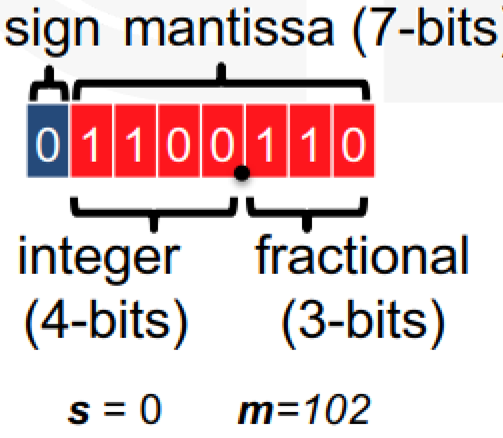
- **Precision refers to the number of levels**
  - $Number\ of\ bits = \log_2(number\ of\ levels)$
  - Normal Precision: FP32
  - Low Precision: FP16, INT8
- **Mixed Precision**
  - Utilizing several precisions (e.g., FP32 and FP16) in model.
- **Quantization:** mapping data to a smaller set of levels
  - Linear, e.g., fixed-point (e.g., INT8, binary)
  - Non-linear
    - Computed (e.g., floating point, log-domain)
    - Table lookup (e.g., learned)

Floating Point (FP32):

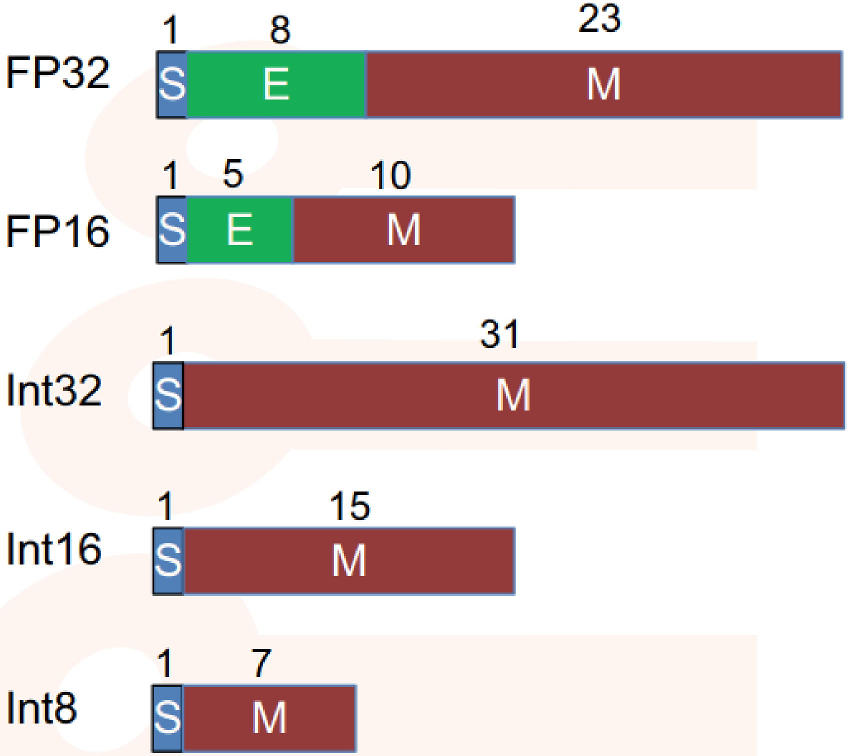
$-1.112934 \times 10^{-16}$

Fixed Point (INT8):

12.75



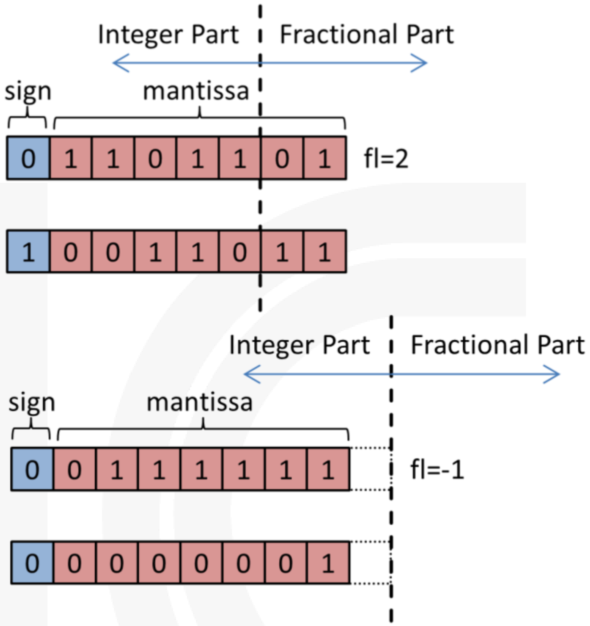
Number Representation



	Range	Accuracy
FP32	$10^{-38} – 10^{38}$	.000006%
FP16	$6x10^{-5} - 6x10^{4}$	.05%
Int32	$0 – 2x10^{9}$	½
Int16	$0 – 6x10^{4}$	½
Int8	0 – 127	½

Source: B. Dally

Dynamic Fixed Point



- Several scaling factors
- Different range for activations, bias, weights and gradients.

Sources: Courbariaux, Montreal UC Davis, Ristretto Project

bfloat16: Brain Floating Point Format

- Same dynamic range as FP32
- Easier for training and debugging than FP16
- Supported by Google TPU, Intel Xeon and Nirvana, others

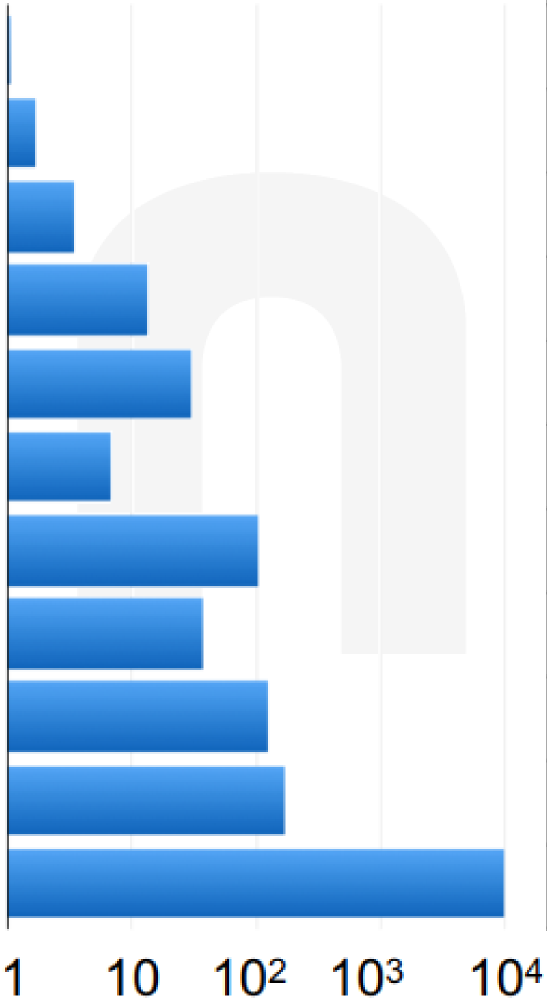
Range: ~1e^{-38} to ~3e^{38}

Source: Patterson, GoogleAI

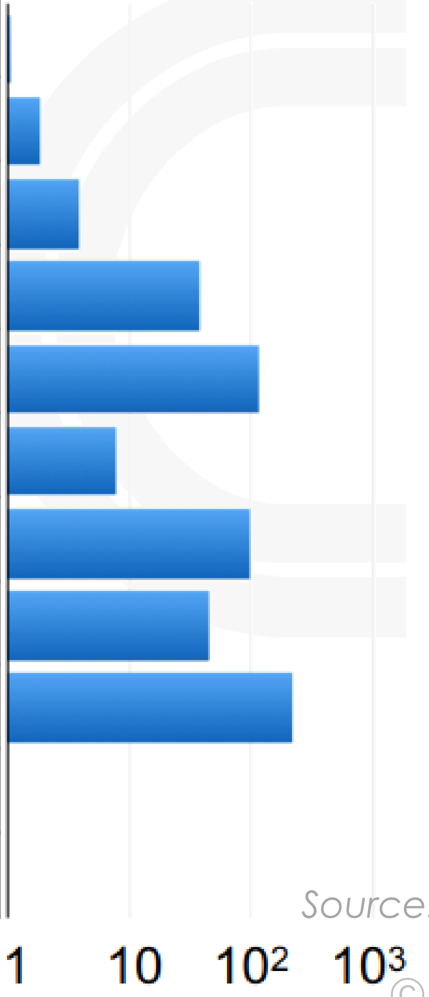
Cost of Operations

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Relative Energy Cost



Relative Area Cost

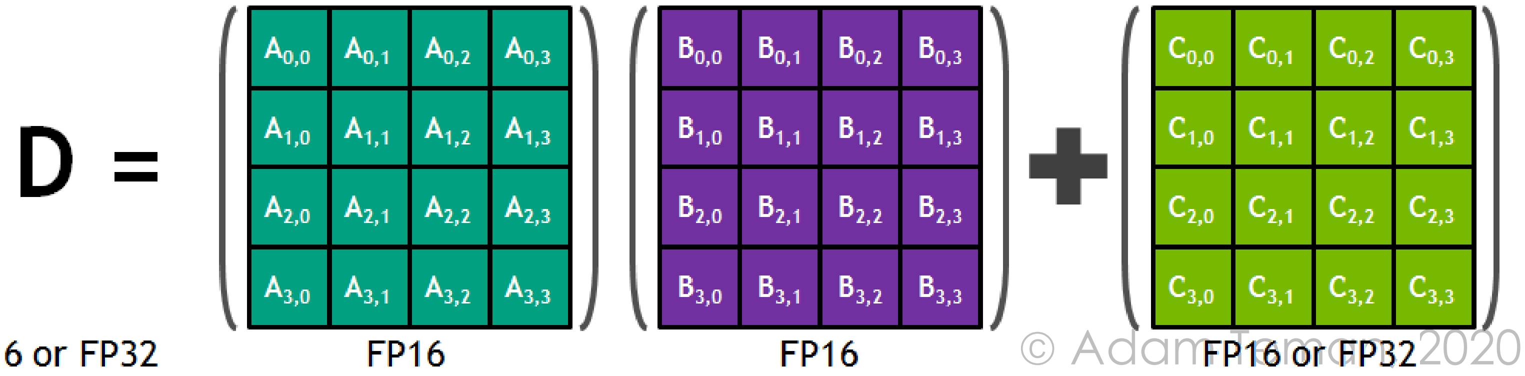


Source: Horowitz, ISSCC 2014

Mixed Precision

- **Mixed Precision refers to using both full and reduced precision in a model:**
  - Identify the steps that require FP32, and use lower precision (e.g., FP16) everywhere else.
  - Has been shown to provide 2-4X speedup.
- **Low precision is supported by hardware and software platforms**
  - Google TPUs support a mix of FP32 and bfloat16
  - Nvidia Tensor Cores accelerate FP16 matrix multiplications and convolutions
  - Keras provides a mixed precision API in TensorFlow

Model	Speedup
BERT Q&A	3.3X speedup
GNMT	1.7X speedup
NCF	2.6X speedup
ResNet-50-v1.5	3.3X speedup
SSD-RN50-FPN-640	2.5X speedup



Source: NVIDIA

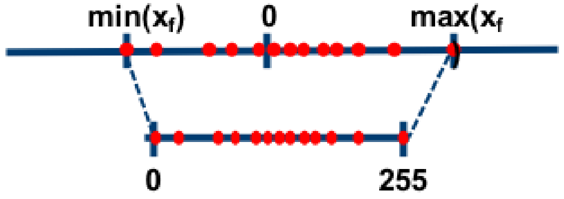
Quantization

- **Quantization is mapping to a smaller set of levels**
  - e.g., floating point (FP32) to integer (INT8)
- **How is it done?**
  - Well, there are a lot of tips and tricks, but basically we just need to scale and offset:

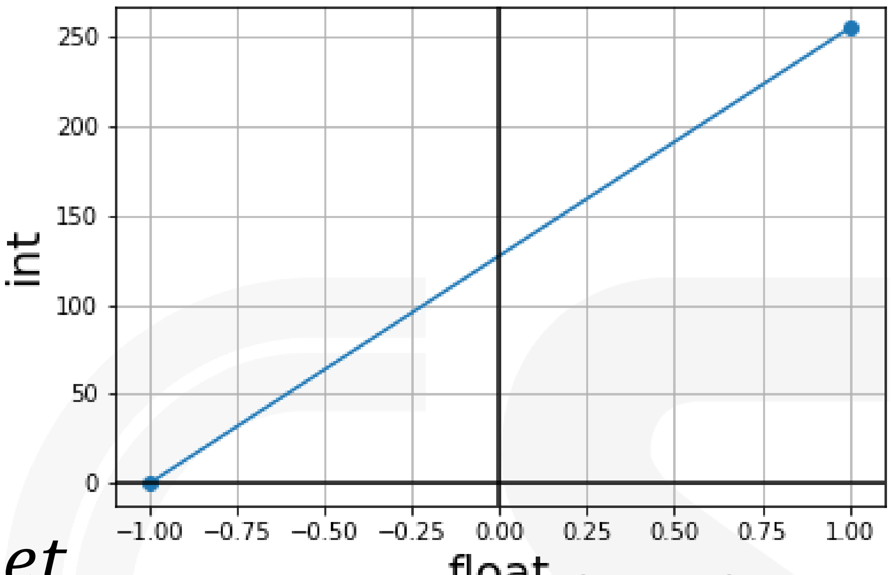
$$x_q = \frac{x_f}{scale} + offset$$

  - The scaling factor is dependent on the range of the floating point values

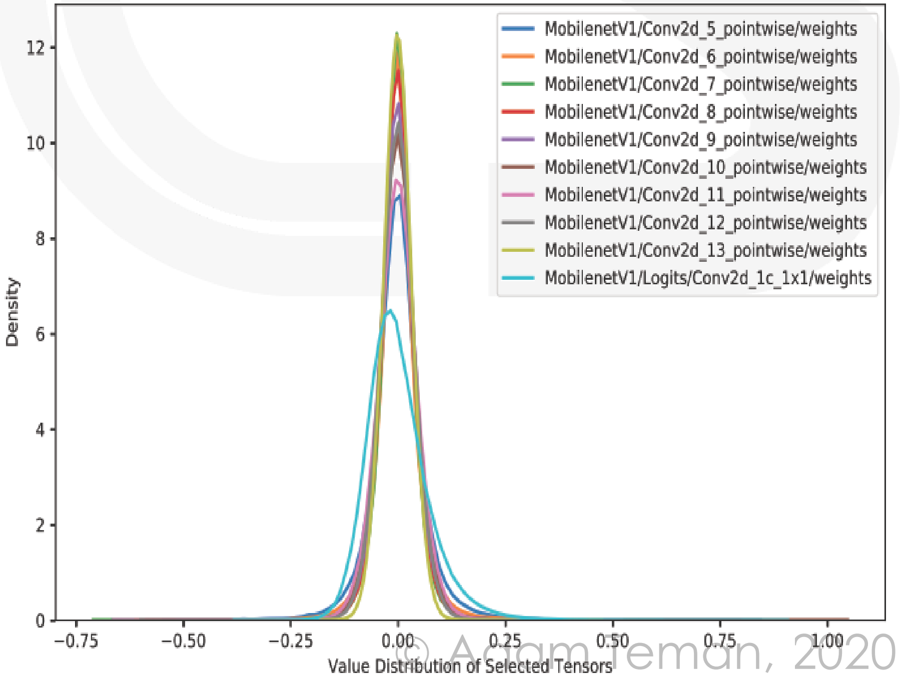
$$scale = \frac{\max x_f - \min x_f}{\max x_q - \min x_q}$$



  - The tighter the distribution, the better the accuracy
  - Luckily, weights tend to have a tight distribution



Source: Soon Yau

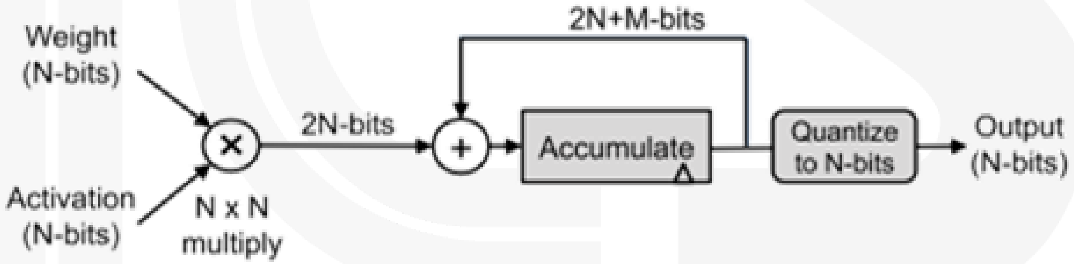


Uniform Quantization

- **Uniform quantization is straightforward quantization of floating point to integer**
  - **INT8 add**: 30X less energy, 116X less area than FP32
  - **INT8 multiply**: 18.5X less energy, 27.5X less area than FP32
- **Precision of internal values of MAC is higher than weights and activations**
  - Given N-bit weights and inputs →Need NxN multiplier→$2N$-bit output product
  - Accumulator: $(2N+M)$-bit

$$M = \log_2 CSR$$

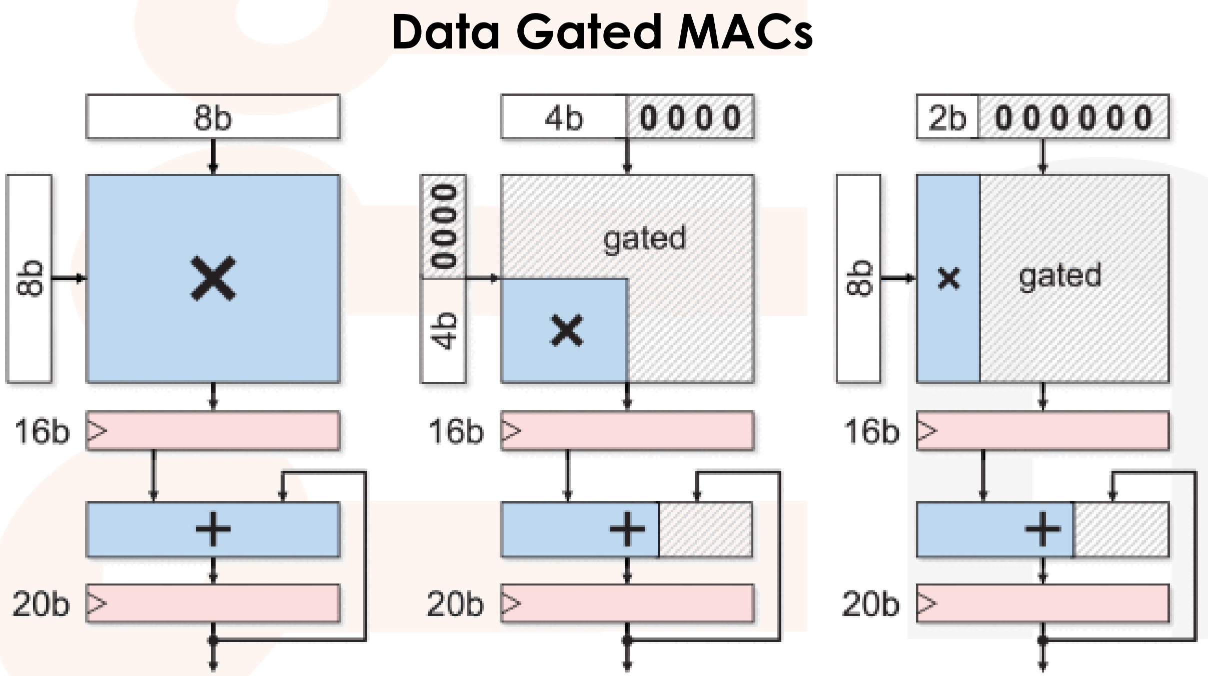
  - Final output activation reduced to N-bits



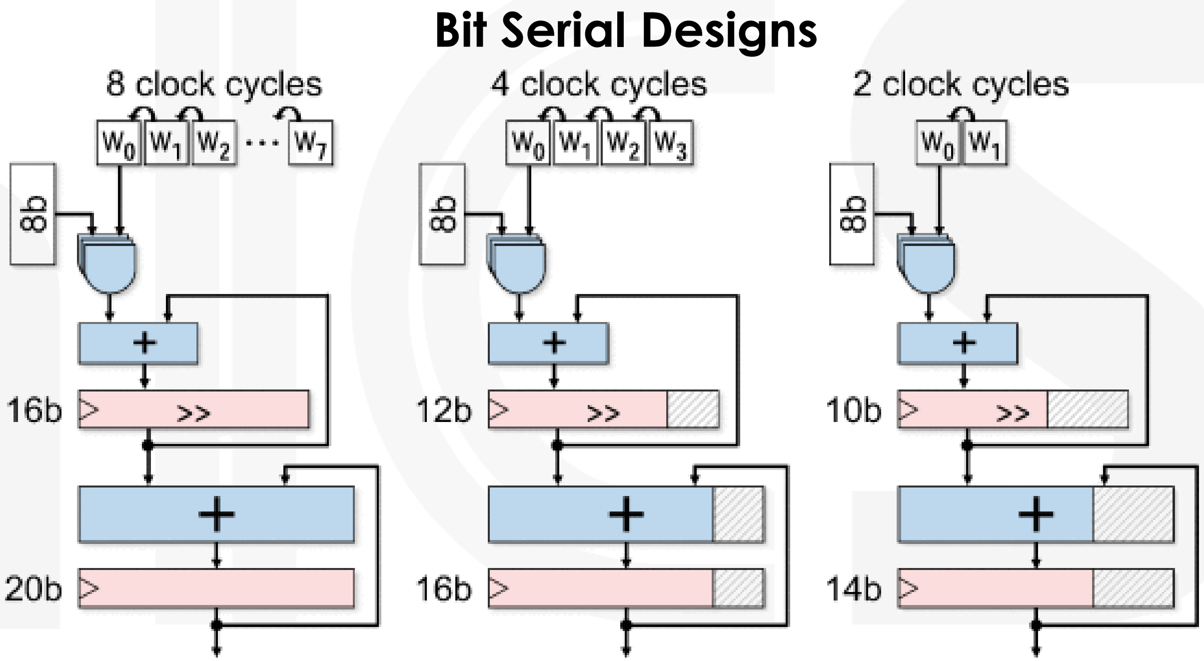
- **No significant impact on accuracy if the distribution of weights and activations is centered near zero.**
  - 8-bit arithmetic used in Google's TPU, Nvidia's PASCAL, Intel's NNP-L

Configurable MACs for Mixed Precision

- Use precision-scalable arithmetic for power savings



Source: Camus, JETCAS 2019



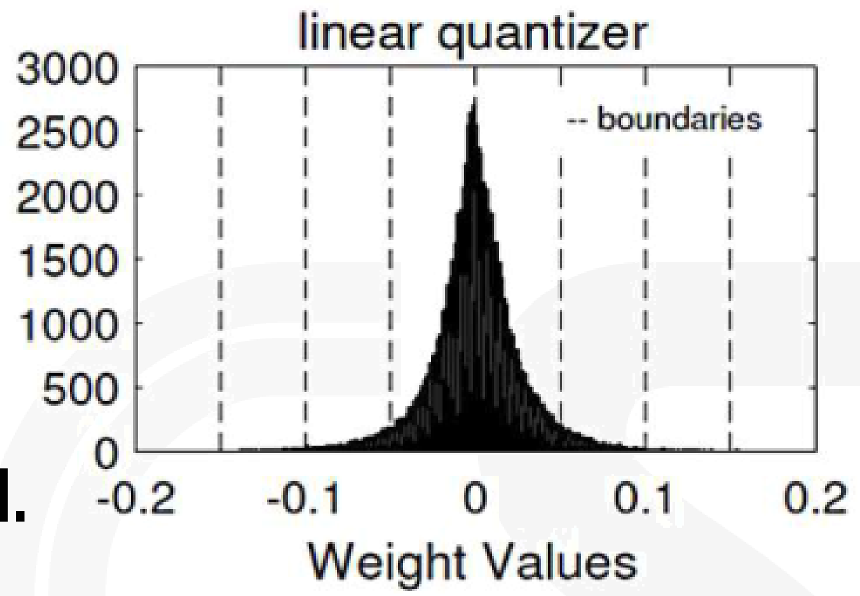
Source: Camus, JETCAS 2019

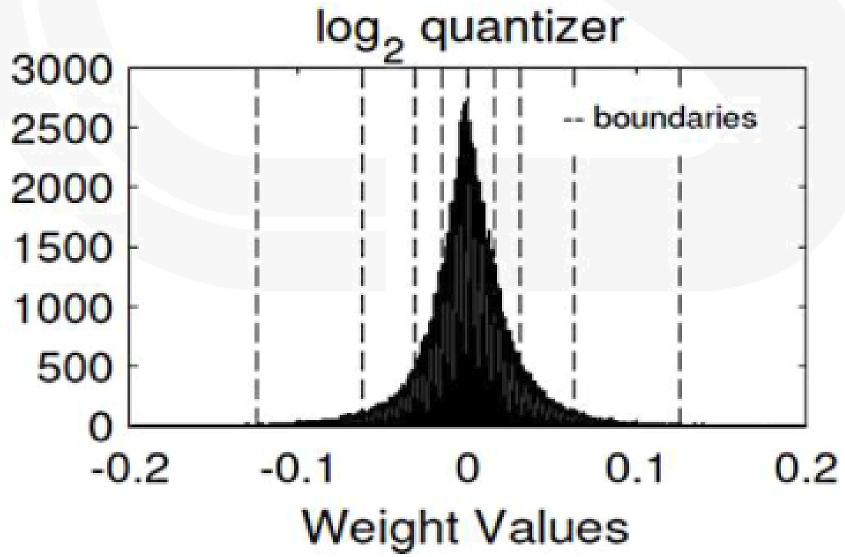
- However, many approaches have overhead that reduce benefits.

Aggressive Quantization

Non-Uniform Quantization

- **In standard uniform quantization, values are equally spaced out**
- **However, computing a quantization that better fits the distribution, better accuracy can be achieved.**
  - e.g. with 4-bit *log-domain quantization*, VGG-16 shows only a 5% loss (vs. 28% with uniform quantization)
- **Log-domain quantization further allows replacing multiplication with bit-shift**
- **Weight sharing, for example through *learned quantization*, can provide an even better solution.**





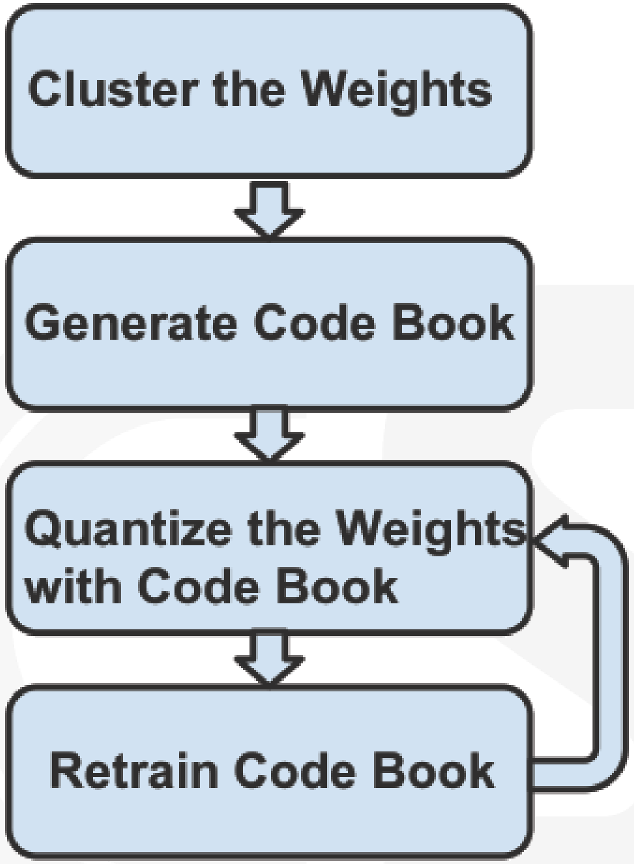
Source: Camus, Lee, ICASSP 2017

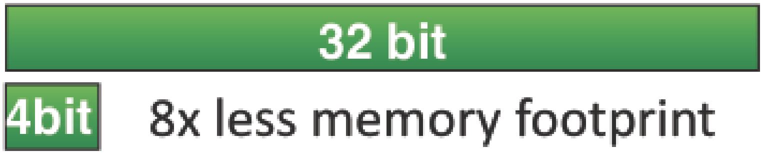
Trained Quantization

~~2.09, 2.12, 1.92, 1.87~~

↓

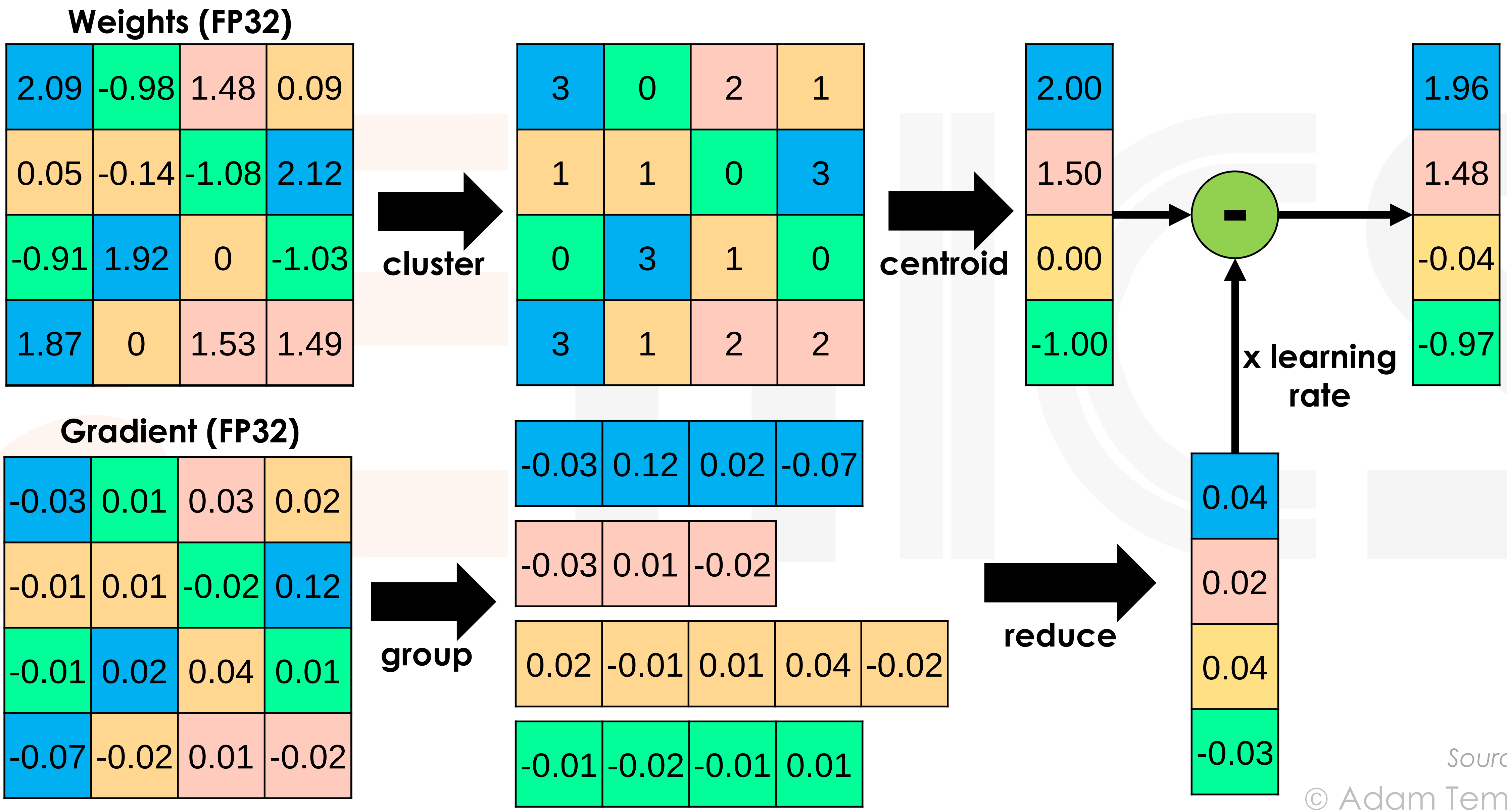
2.0





[Han et al. ICLR'16]

Trained Quantization

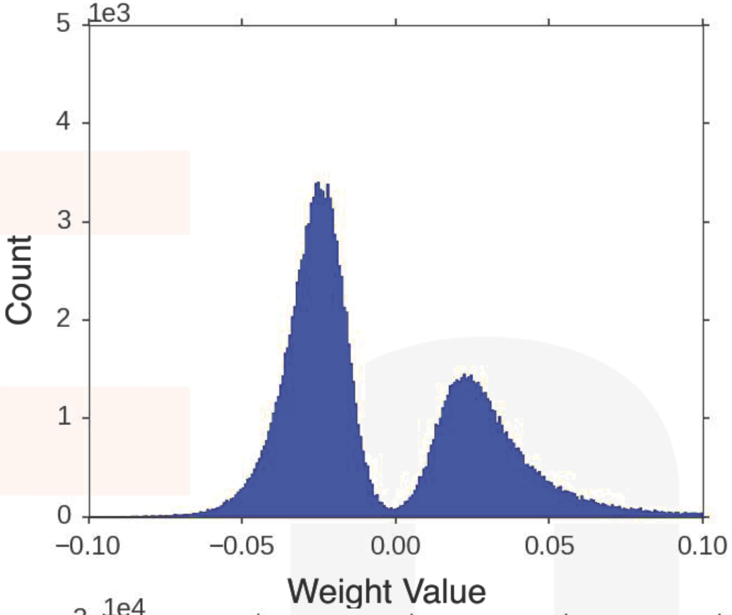


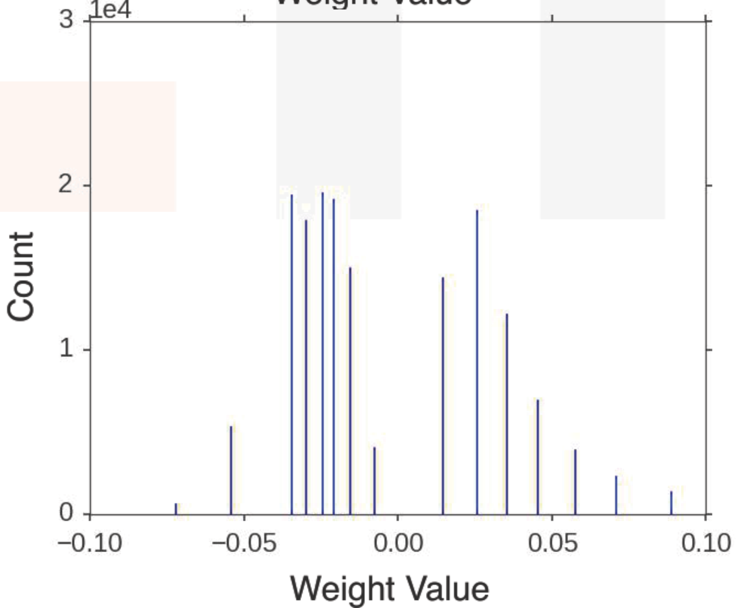
Source: Han

Trained Quantization

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

1.96	-0.97	1.48	-0.04
-0.04	-0.04	-0.97	1.96
-0.97	1.96	-0.04	-0.97
1.96	-0.04	1.48	1.48



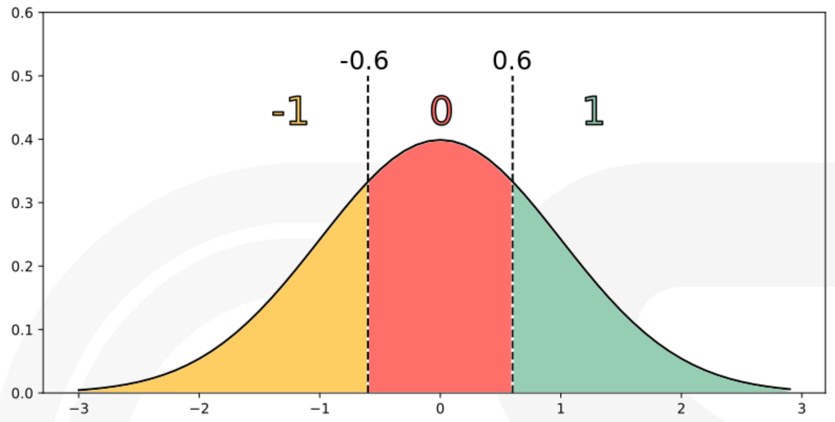


- **AlexNet:**
  - 8-bit quantization on CONV layers, 5-bit quantization on FC layers without any loss of accuracy
  - Only 2% loss of accuracy for 4-bit CONV and 2-bit FC layer quantization
- **Need “cookbook” for index translation**
  - See “Deep Compression” later on in the lecture.

Source: Han

More aggressive quantization

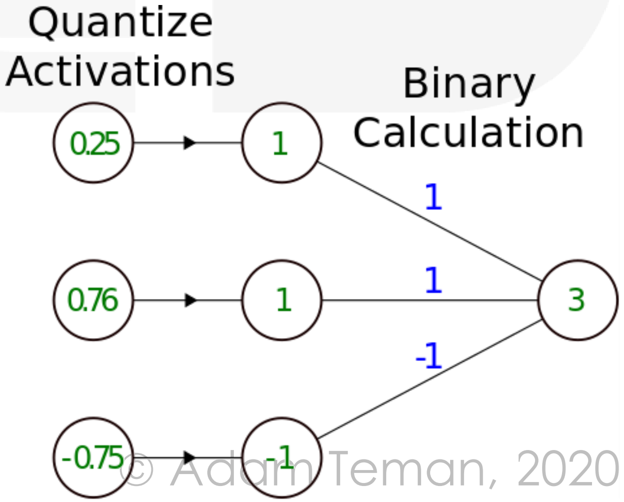
- **Ternary Connect (2014)**
  - Train with real valued weights
  - Ternarize the weights to $W_B \in \{-H, 0, H\}$
- **Binary Connect (2015)**
  - Binary weights ($W_B \in \{-1, 1\}$), full precision activations
  - Simple multipliers, full precision accumulation
  - Training (backprop updates) uses real valued weights (W_R) clipped at -1, 1.



$$W_B = \mathrm{sign}(W_R)$$

- **BinaryNet, Binarized Neural Networks, XNOR-Net (2016)**
  - Binary Weights and Activations
  - Use XNOR for multiplication
    "popcount" for accumulation
  - Keep first and last layers at full precision

Encoding (Value)		XNOR (Multiply)
0 (−1)	0 (−1)	1 (+1)
0 (−1)	1 (+1)	0 (−1)
1 (+1)	0 (−1)	0 (−1)
1 (+1)	1 (+1)	1 (+1)



Summary

Category	Method	Weights (# of bits)	Activations (# of bits)	Accuracy Loss vs. 32-bit float (%)
Dynamic Fixed Point	w/o fine-tuning	8	10	0.4
	w/ fine-tuning	8	8	0.6
Reduce weight	Ternary weights Networks (TWN)	2*	32	3.7
	Trained Ternary Quantization (TTQ)	2*	32	0.6
	Binary Connect (BC)	1	32	19.2
	Binary Weight Net (BWN)	1*	32	0.8
Reduce weight and activation	Binarized Neural Net (BNN)	1	1	29.8
	XNOR-Net	1*	1	11
Non-Linear	LogNet	5(conv), 4(fc)	4	3.2
	Weight Sharing	8(conv), 4(fc)	16	0

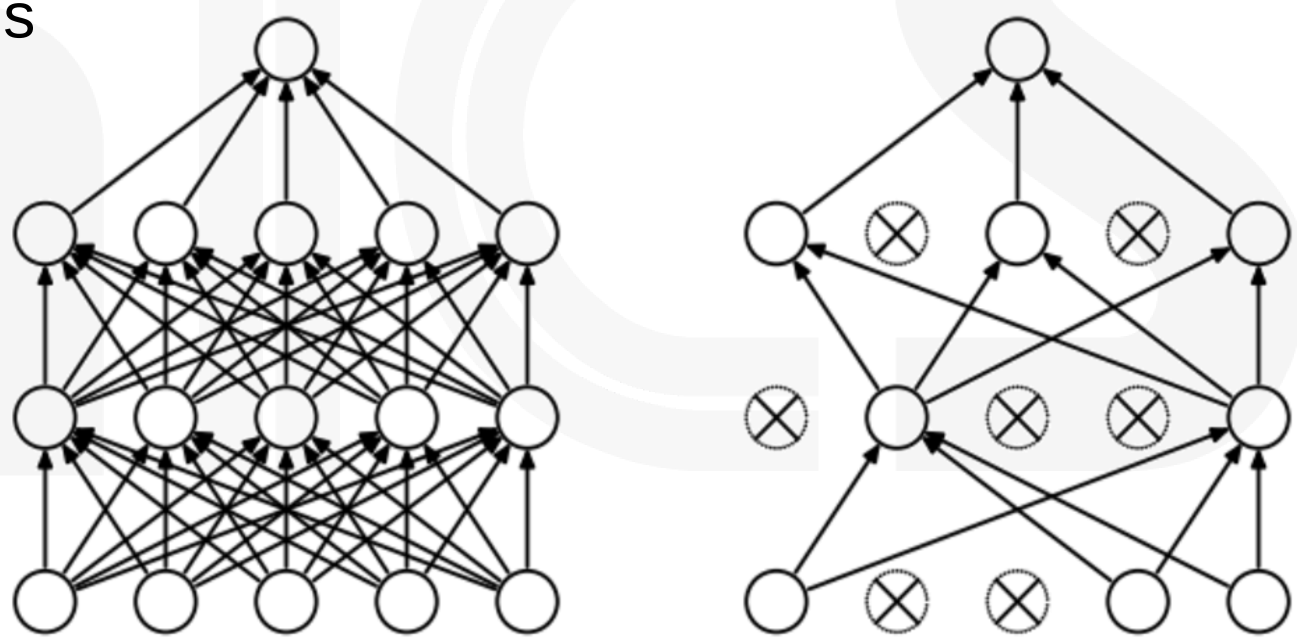
* first and last layers are 32-bit float

Source: Sze, MIT

Pruning and Deep Compression

Precursor: Dropout

- **A well-known technique for eliminating overfitting is called "Dropout"**
  - During each iteration of training, zero out a random fraction of nodes in fully connected layers
  - During inference, use all connections
- **But regularization through batch normalization has almost made this unnecessary.**
- **However, it raises the question: "Do we actually need all synapses?"**

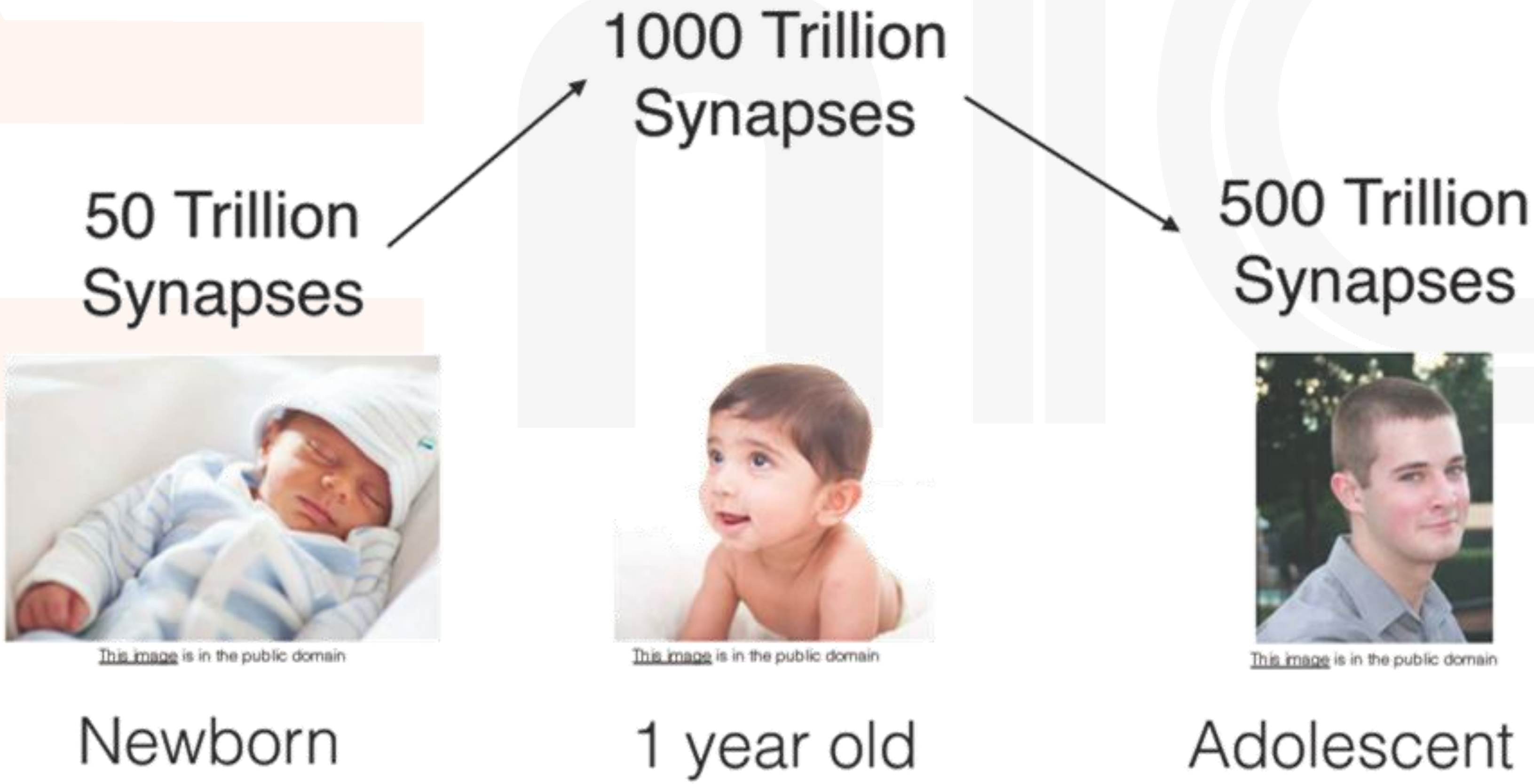


(a) Standard Neural Net (b) After applying dropout.

Source: Srivastava, et al.

Synaptic Pruning

- **The human (all mammals) body prunes synapses**
  - Axons and dentrites completely decay and die off during lifetime
  - Starts near birth and continues into the mid-20s

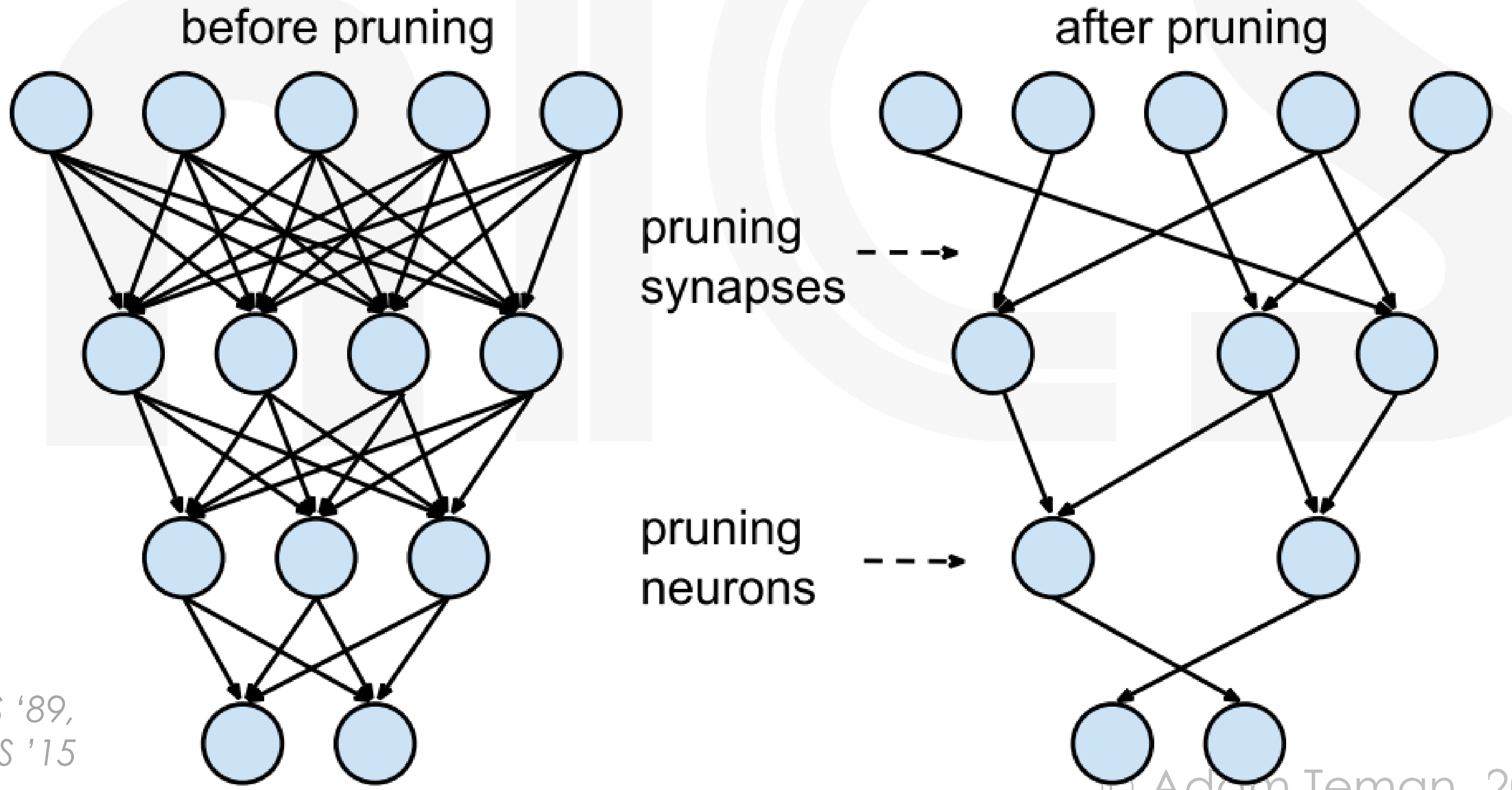


Source: Walsh, Nature 2013

Optimal Brain Damage

- **In 1989, Yann Lecun suggested pruning neural networks**
  - Compute the impact of each weight on the training loss = weight saliency
  - Remove low-saliency weights and fine tune remaining weights

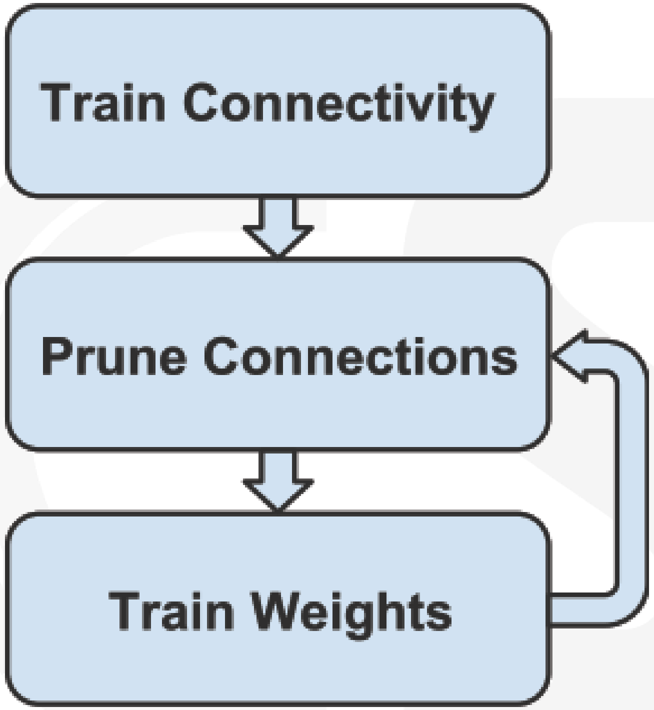
- **Unlike in “Dropout”, pruned synapses are removed for good.**



Source: Lecun, NIPS ‘89, Han, NIPS ’15

Pruning Deep Neural Networks

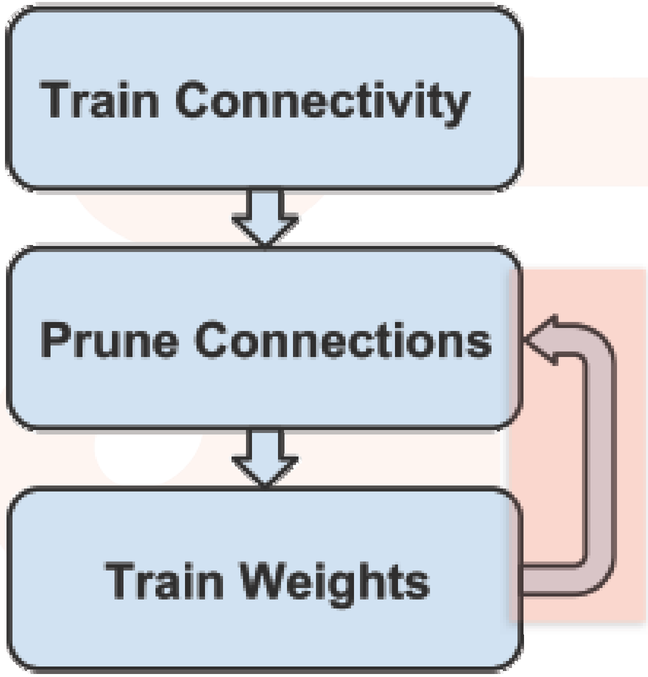
- **Pruning DNNs leads to sparsity**
  - Easier to compress
  - Skip multiplications by zero
- **Han, et al., showed that 90% of the connections in AlexNet can be pruned without incurring accuracy loss!**
  - Weights were pruned below a threshold
- **The Train-Prune-Retrain pipeline was used**

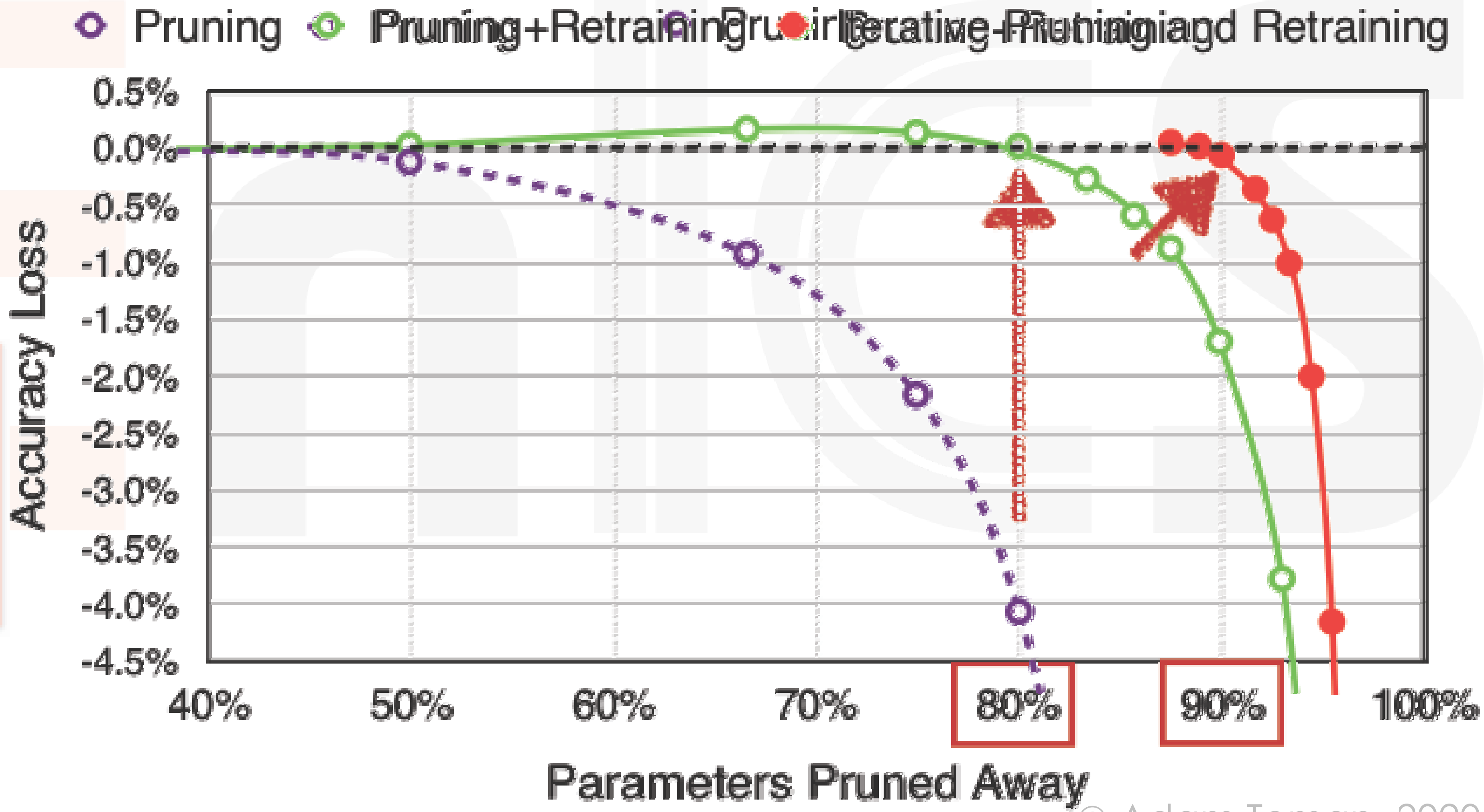


Source: Han, NIPS '15

Pruning Deep Neural Networks

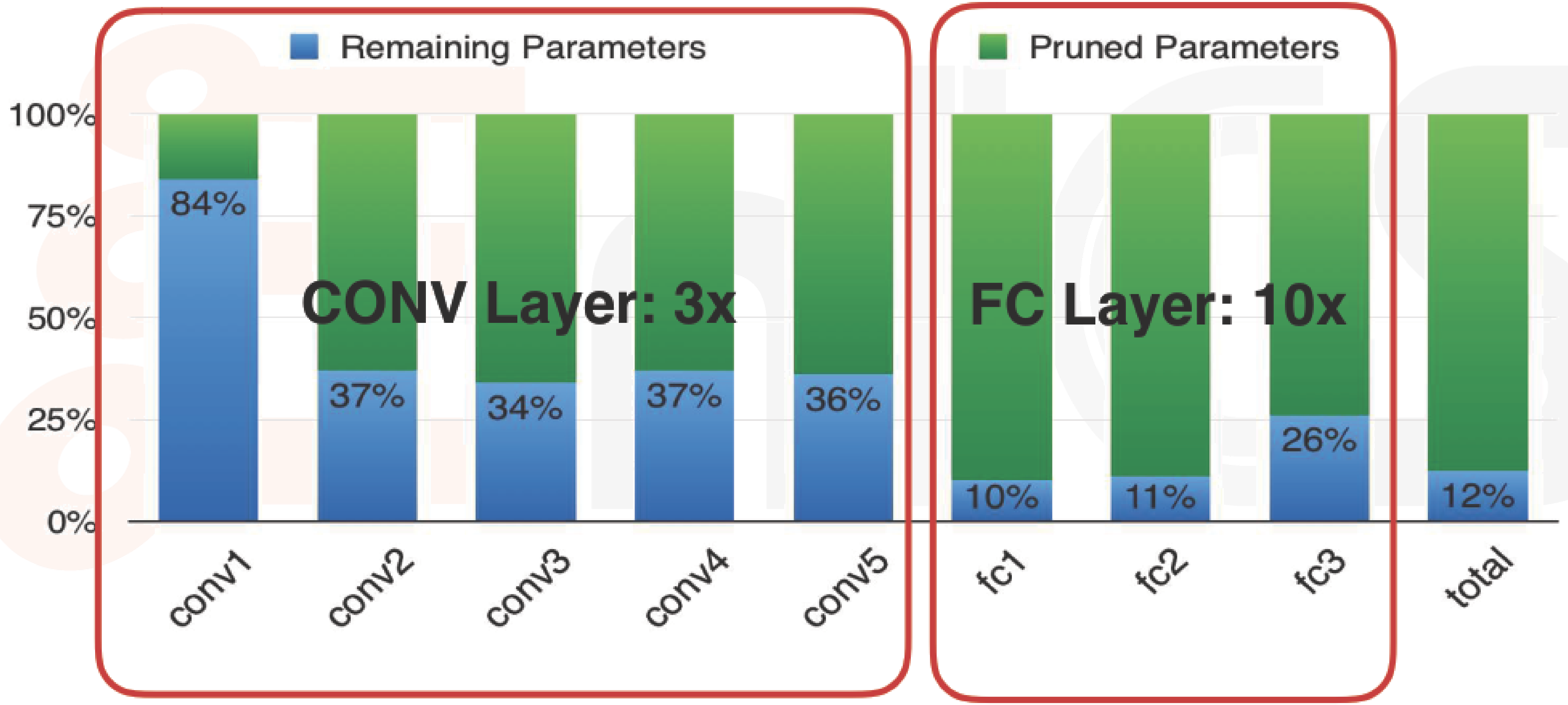
- **Iteratively Retrain to recover accuracy**





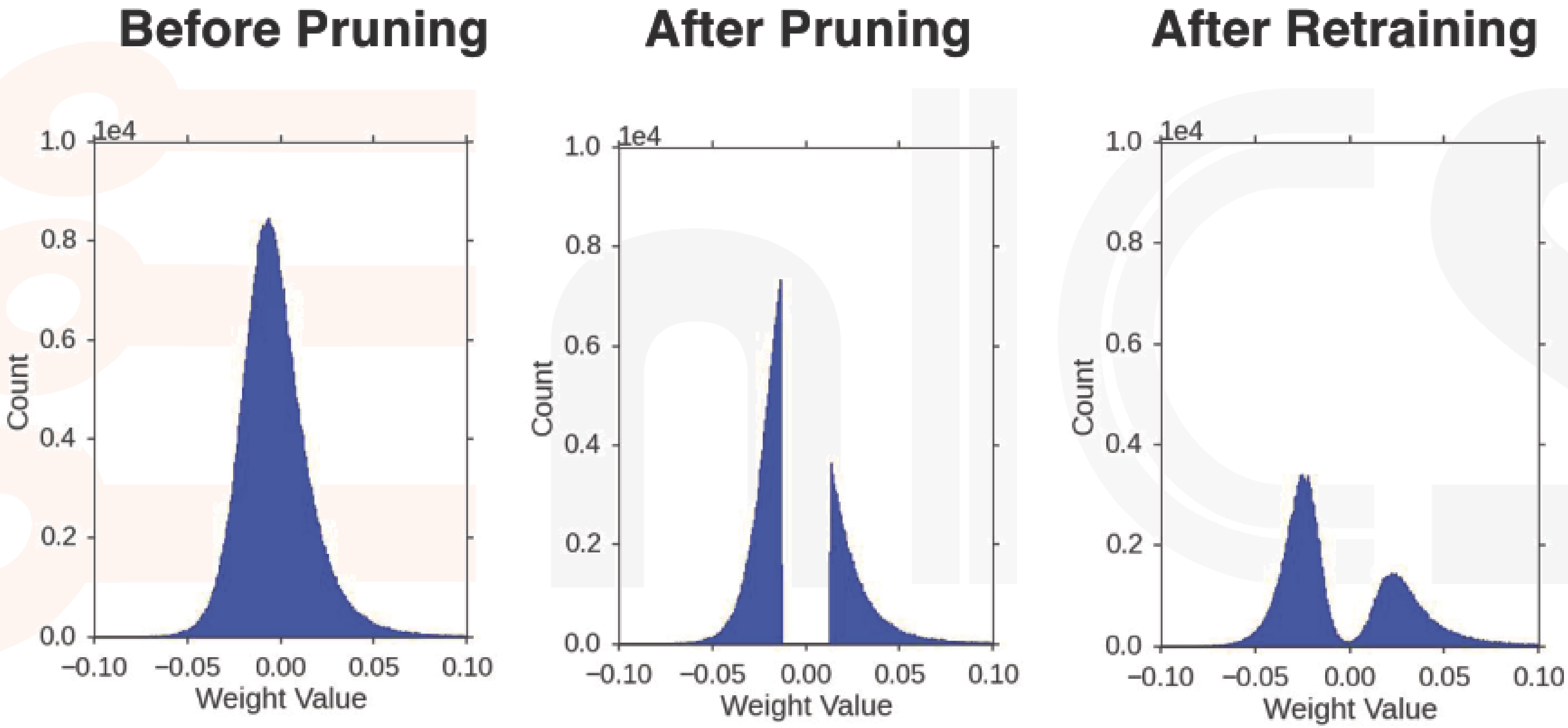
Source: Han, NIPS '15

Pruning AlexNet



Source: Han, NIPS '15

Pruning Changes Weight Distribution

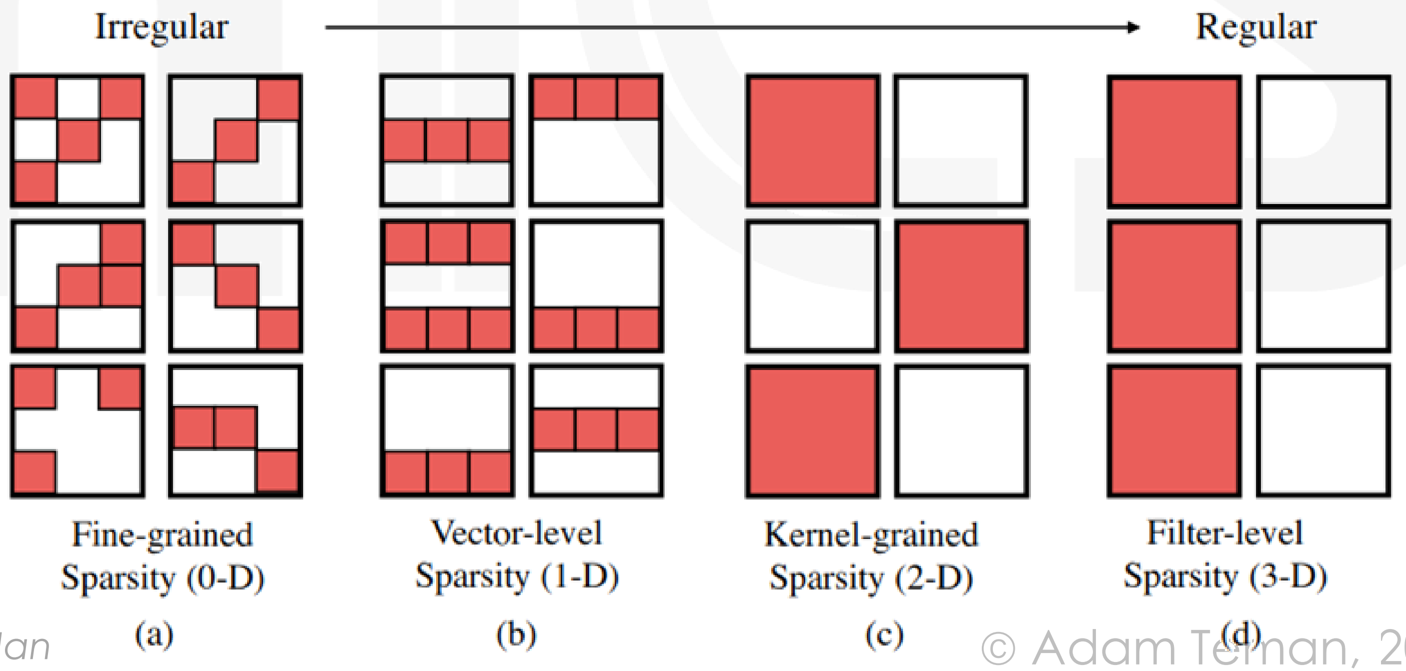


Conv5 layer of Alexnet. Representative for other network layers as well.

Source: Han, NIPS '15

Hardware Efficiency Considerations in Pruning

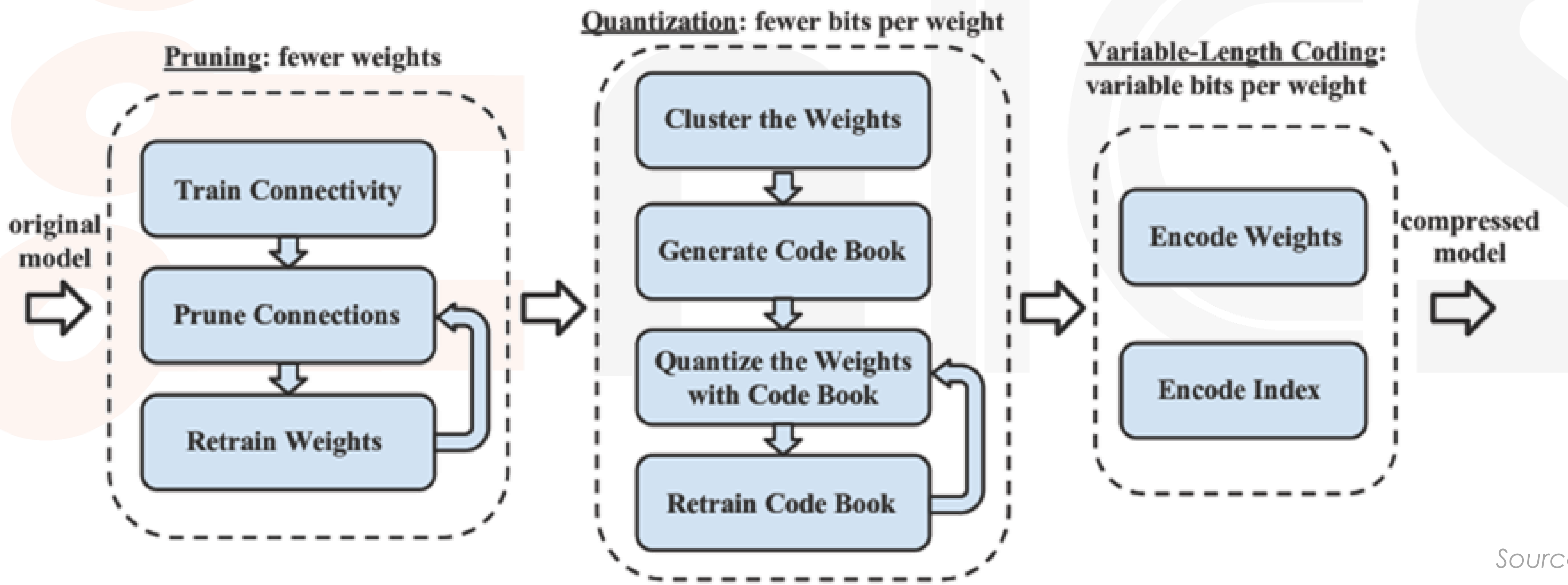
- **Pruning leads to irregularity, which is difficult to parallelize in hardware**
- **Load-balance aware pruning**
  - Sort the weights in every sub-matrix and prune the same amount in each, such that each PE works on the same number of non-zero weights
  - Need to index every non-zero weight
- **Pruning with structure**
  - Prune by rows/columns, kernels, or whole filters
  - Can index a larger space
  - For example, prune a column according to L2 norm



Source: Han

Deep Compression

- Deep Compression combines pruning, trained quantization and variable length coding in a pipeline:



Source: Han

Storing the Meta Data

- **How do we store the index and weight?**
  - For each non-zero weight store the weight and the index
  - Instead of the actual index, store the distance from the previous non-zero index
  - Select a small bit-width for the index representation – if the span is larger, then pad with zeros.
- **A separate codebook is stored for each layer**

Span Exceed 16=2^4

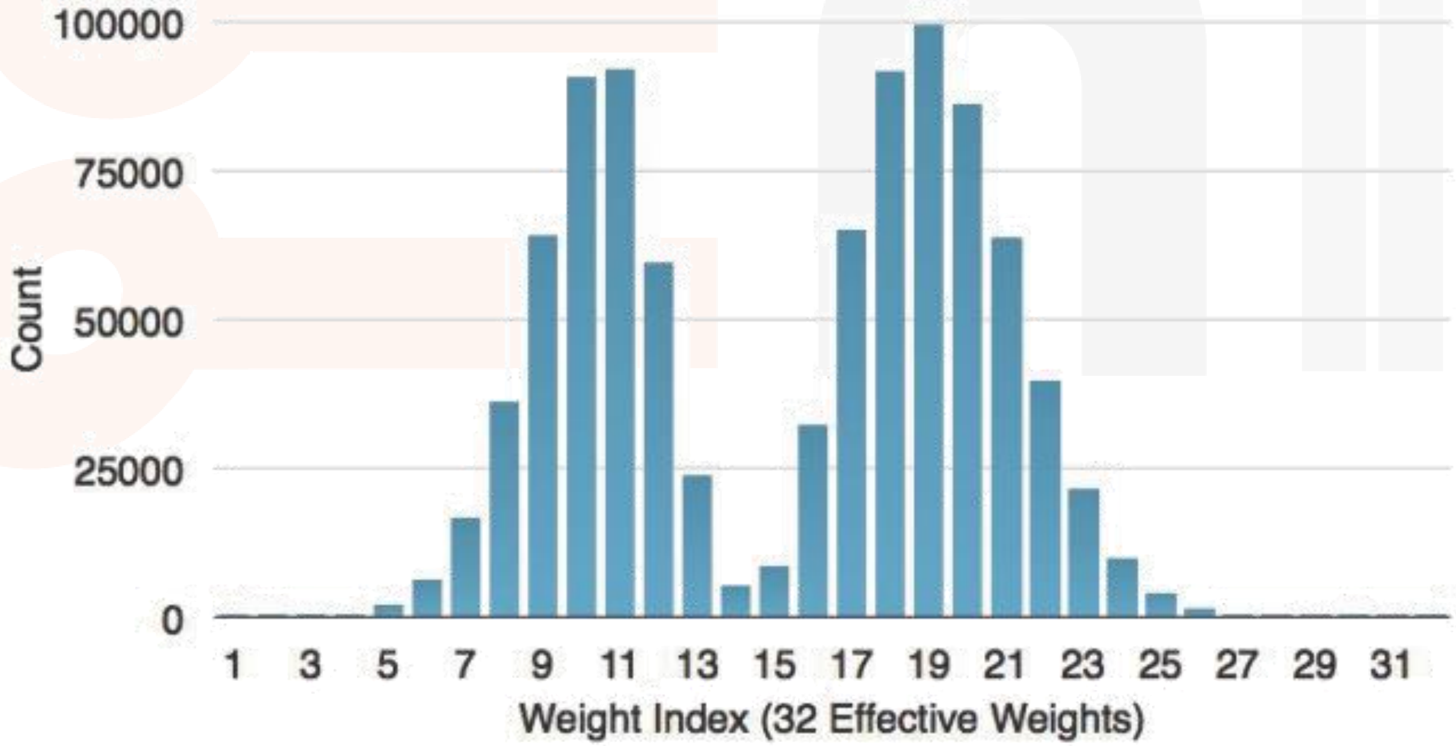
idx	0	1	2	3	4	5	6	7	8	...	18	19	20	21	22	23	24	25	26	27	28
diff		1				4								16			3			3	
value		3.4				0.9								0			1.7			0.5	

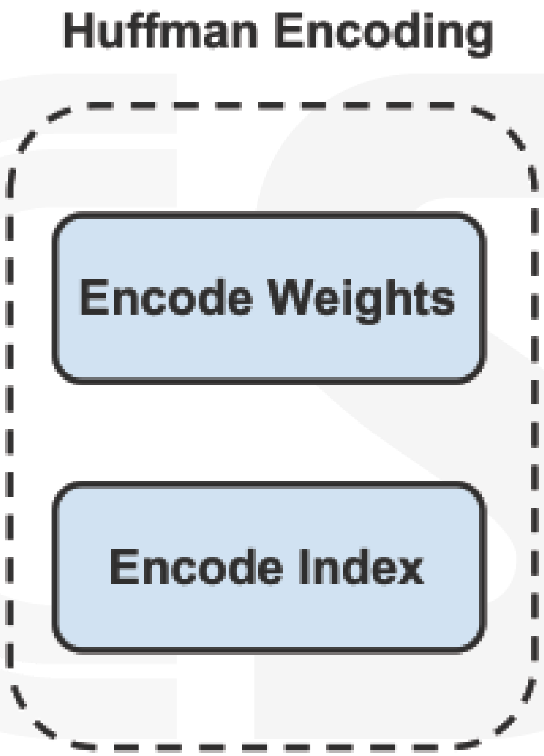
Filler Zero

Figure 4.5: Pad a filler zero to handle overflow when representing a sparse vector with relative index.

Variable-Length Coding

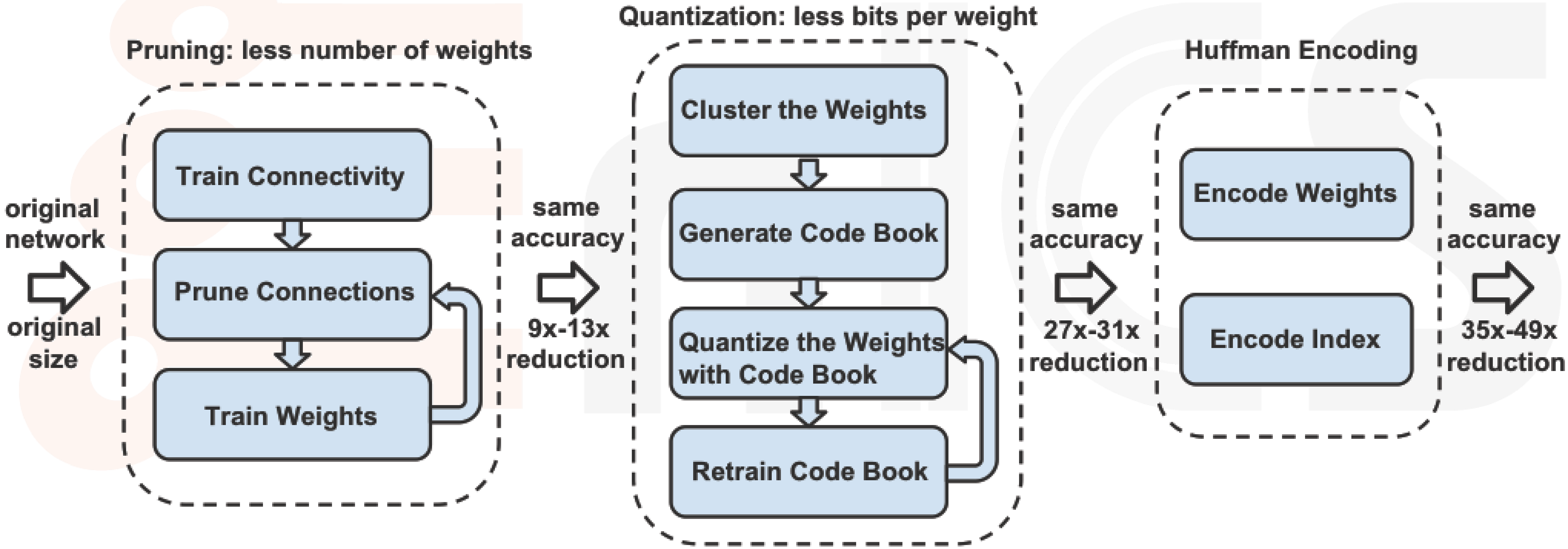
- **The idea is:**
  - Infrequent weights: use more bits to represent
  - Frequent weights: use less bits to represent
- **Huffman coding is used for Deep Compression.**





Source: Han

Summary of Deep Compression



Source: Han

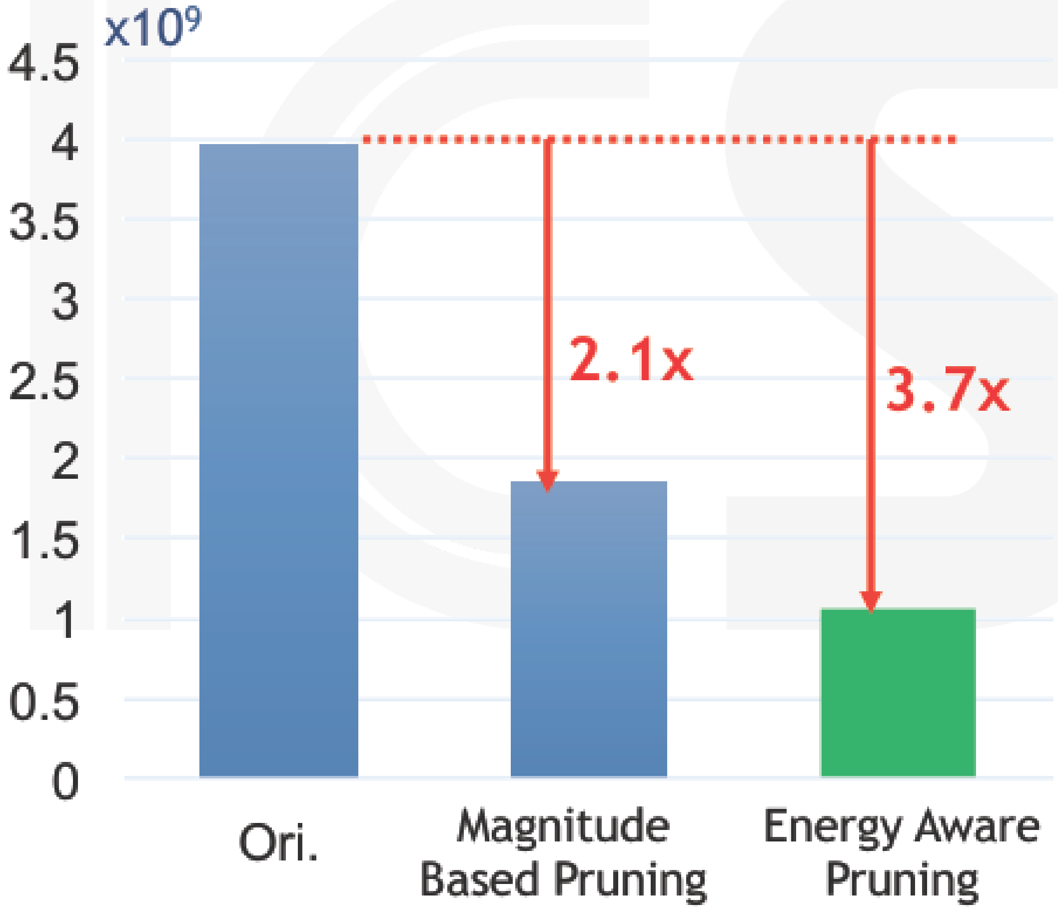
Results: Compression Ratio

Network	Original Size	Compressed Size	Compression Ratio	Original Accuracy	Compressed Accuracy
LeNet-300	1070KB	→ 27KB	**40x**	98.36%	→ 98.42%
LeNet-5	1720KB	→ 44KB	**39x**	99.20%	→ 99.26%
AlexNet	240MB	→ 6.9MB	**35x**	80.27%	→ 80.30%
VGGNet	550MB	→ 11.3MB	**49x**	88.68%	→ 89.09%
GoogleNet	28MB	→ 2.8MB	**10x**	88.90%	→ 88.92%
ResNet-18	44.6MB	→ 4.0MB	**11x**	89.24%	→ 89.28%

Source: Han

Energy-Aware Pruning

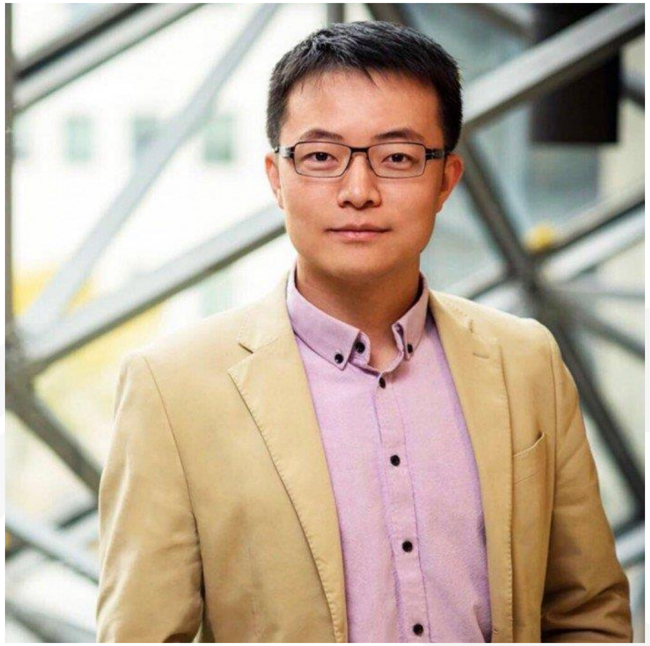
- **The value of weights alone is not a good metric for energy**
  - Instead prune according to energy.
- **Sort layers based on energy and prune layers that consume the most energy first**
- **Energy-aware pruning reduces AlexNet energy by 3.7x and outperforms the previous work that uses magnitude-based pruning by 1.7x**

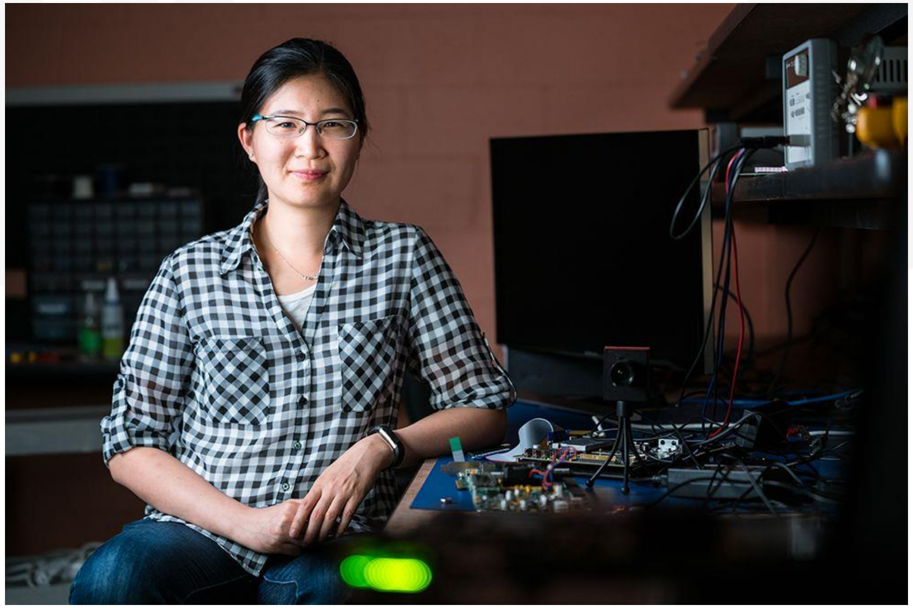


Source: Sze, MIT

Main References

- **Song Han, various talks**
- **Vivienne Sze, various talks**
- **Bill Dally, various talks**
- **Towards Data Science:**
  - Bharath Raj
  - Yusuke Uchida
  - Arthur Douillard
  - Sik-Ho Tsang
  - Chi-Feng Wang
  - Ranjeet Singh
  - others





Source: MIT