Hardware Architectures for Deep Neural Networks

MICRO Tutorial

October 16, 2016

Website: http://eyeriss.mit.edu/tutorial.html

Speakers

Joel Emer

Senior Distinguished Research Scientist

NVIDIA

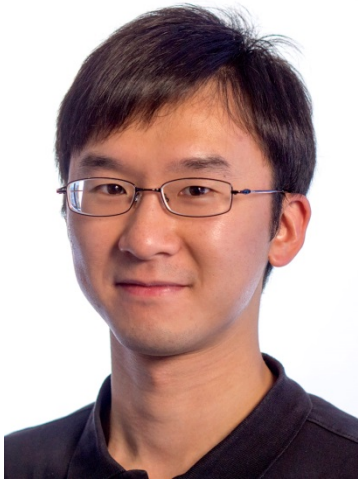
Professor

MIT

Vivienne Sze

Professor

MIT



Yu-Hsin Chen

PhD Candidate

MIT

Outline

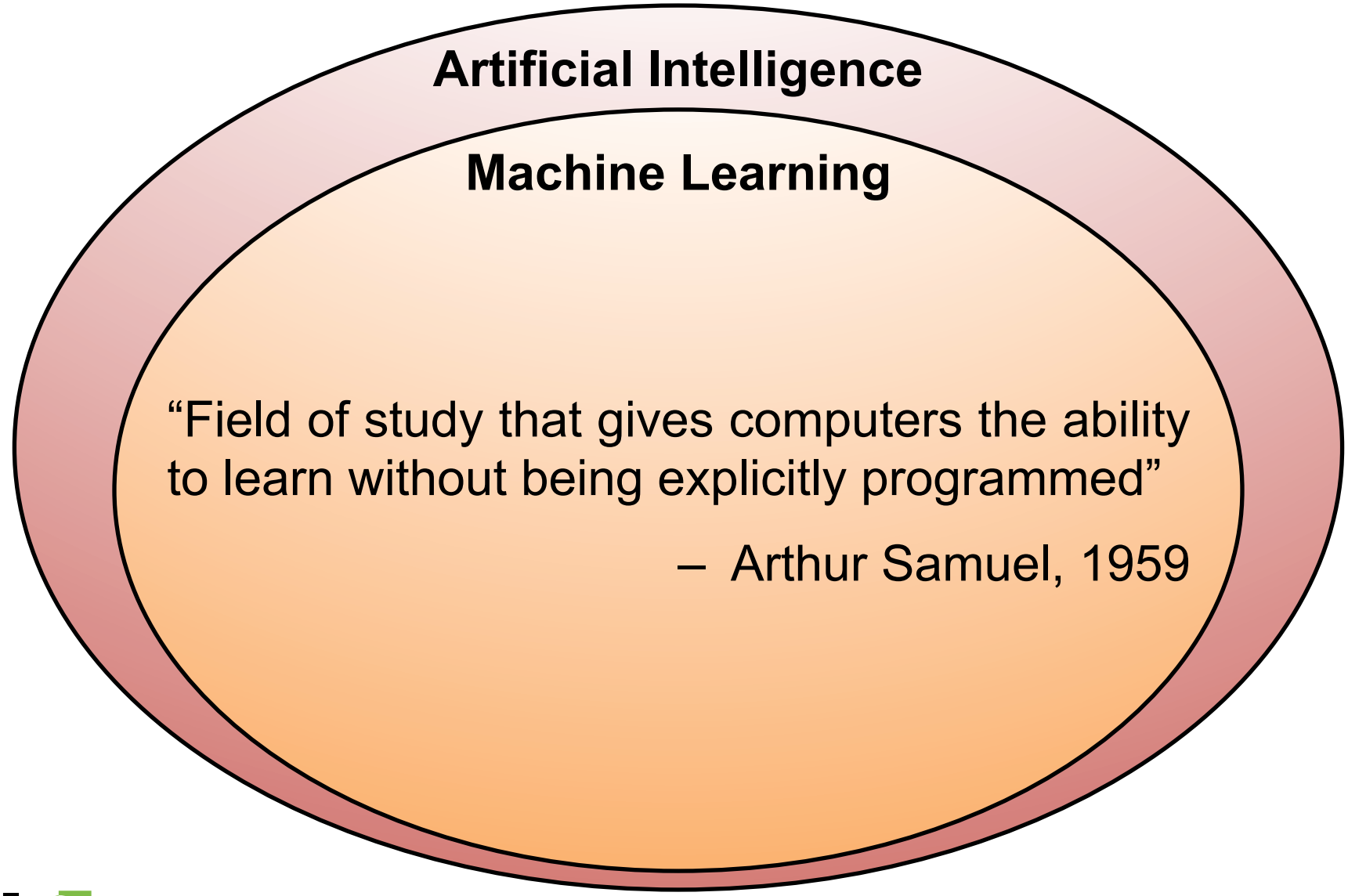
- Overview of Deep Neural Networks
- DNN Development Resources
- Survey of DNN Computation
- DNN Accelerators
- Network Optimizations
- Benchmarking Metrics for Evaluation
- DNN Training

Participant Takeaways

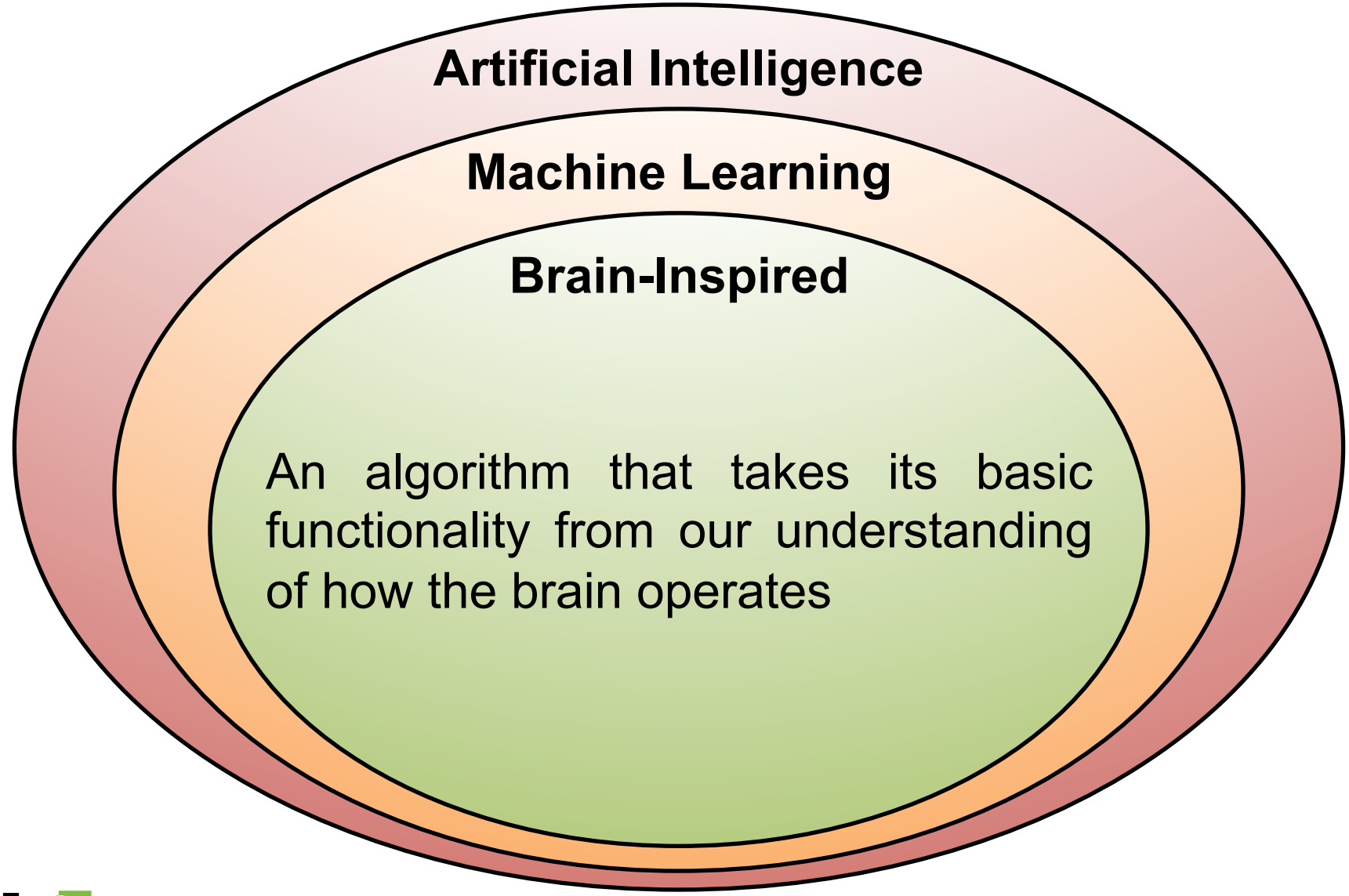
- Understand the key design considerations for DNNs
- Be able to evaluate different implementations of DNN with benchmarks and comparison metrics
- Understand the tradeoffs between various architectures and platforms
- Assess the utility of various optimization approaches
- Understand recent implementation trends and opportunities

Background of Deep Neural Networks

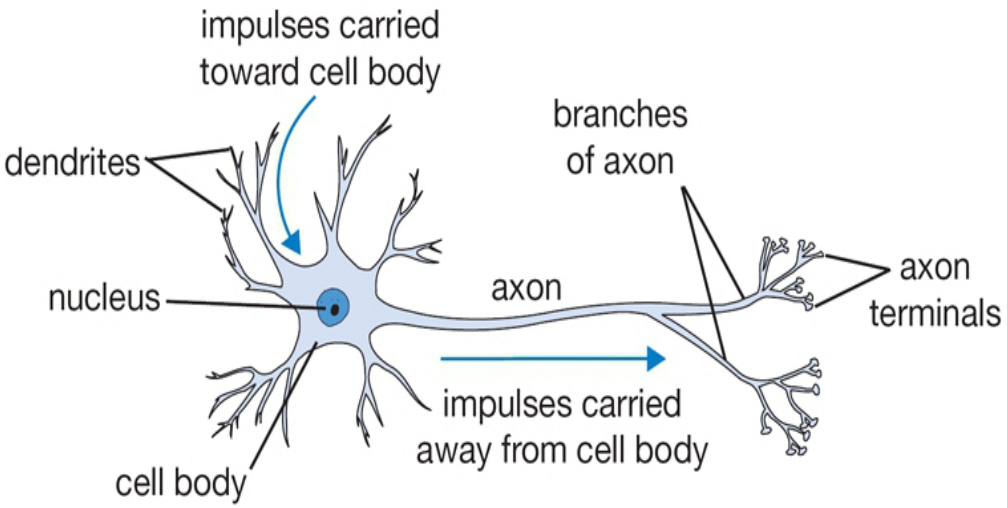
AI and Machine Learning

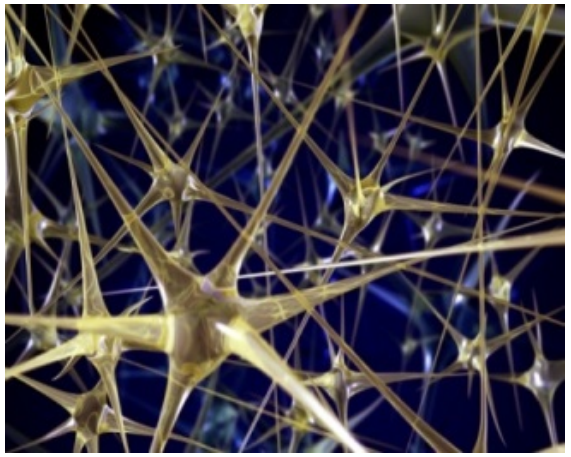


Brain-Inspired Machine Learning



How Does the Brain Work?

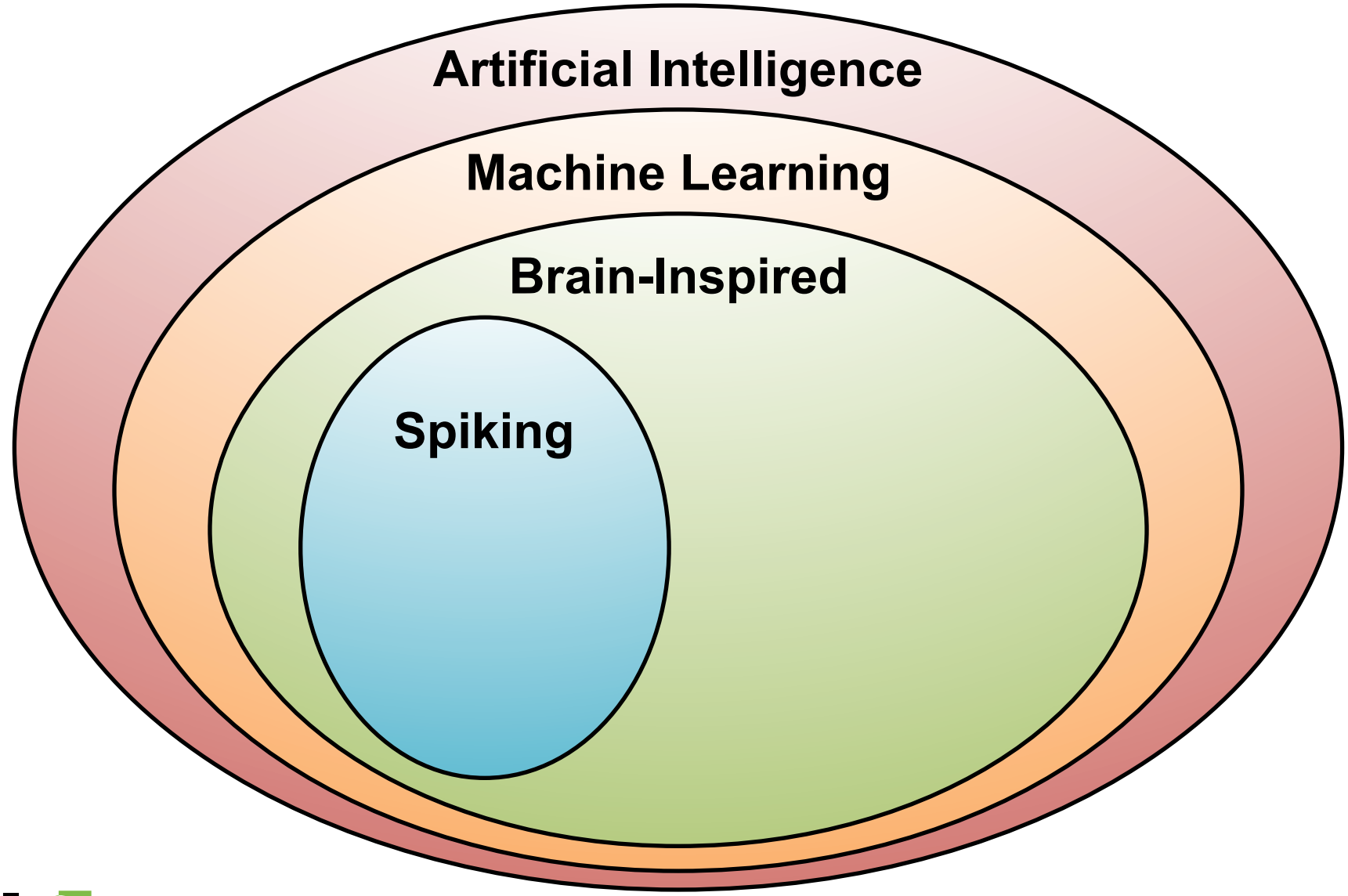




- The basic computational unit of the brain is a **neuron**
  → 86B neurons in the brain
- Neurons are connected with nearly **10^{14} – 10^{15} synapses**
- Neurons receive input signal from **dendrites** and produce output signal along **axon**, which interact with the dendrites of other neurons via **synaptic weights**
- Synaptic weights – learnable & control influence strength

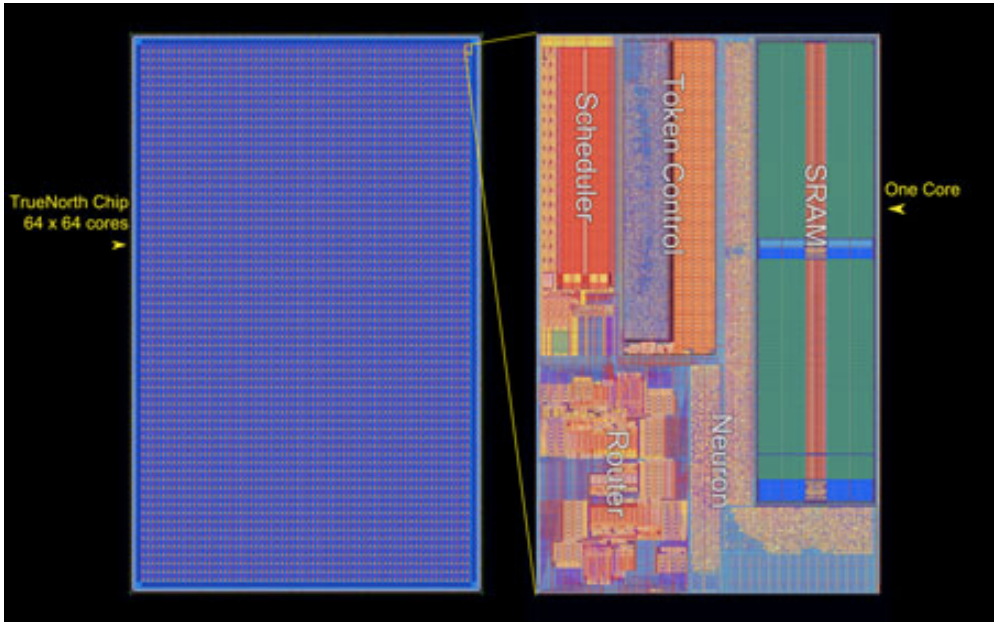
Image Source: Stanford

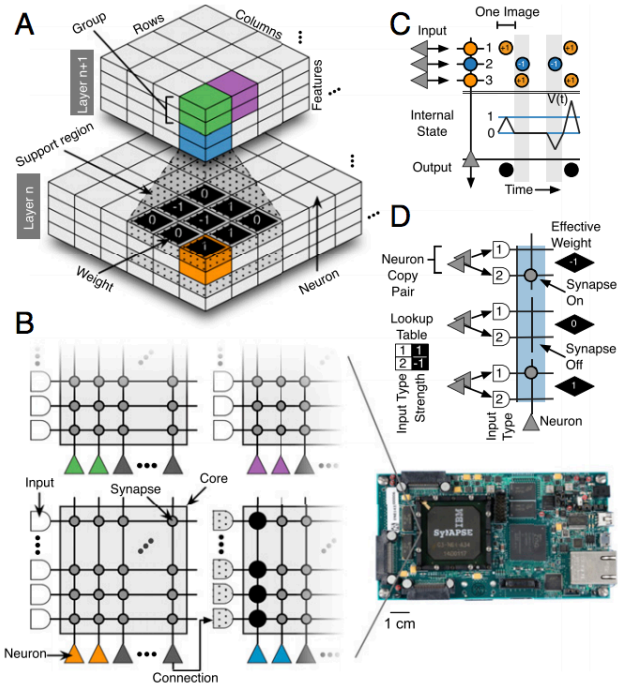
Spiking-based Machine Learning



Spiking Architecture

- Brain-inspired
- Integrate and fire
- Example: IBM TrueNorth

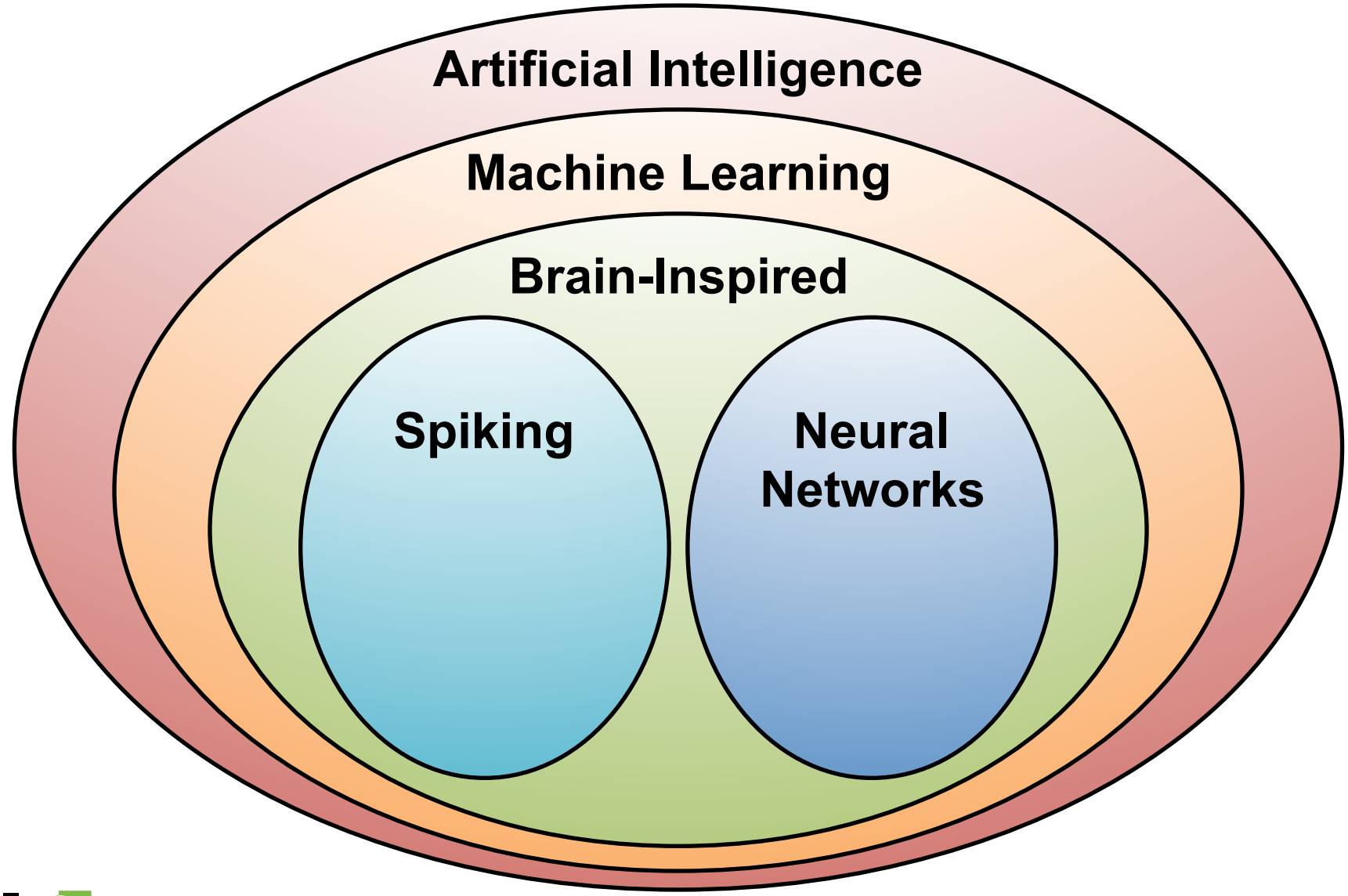




[Merolla et al., Science 2014; Esser et al., PNAS 2016]

http://www.research.ibm.com/articles/brain-chip.shtml

Machine Learning with Neural Networks



Neural Networks: Weighted Sum

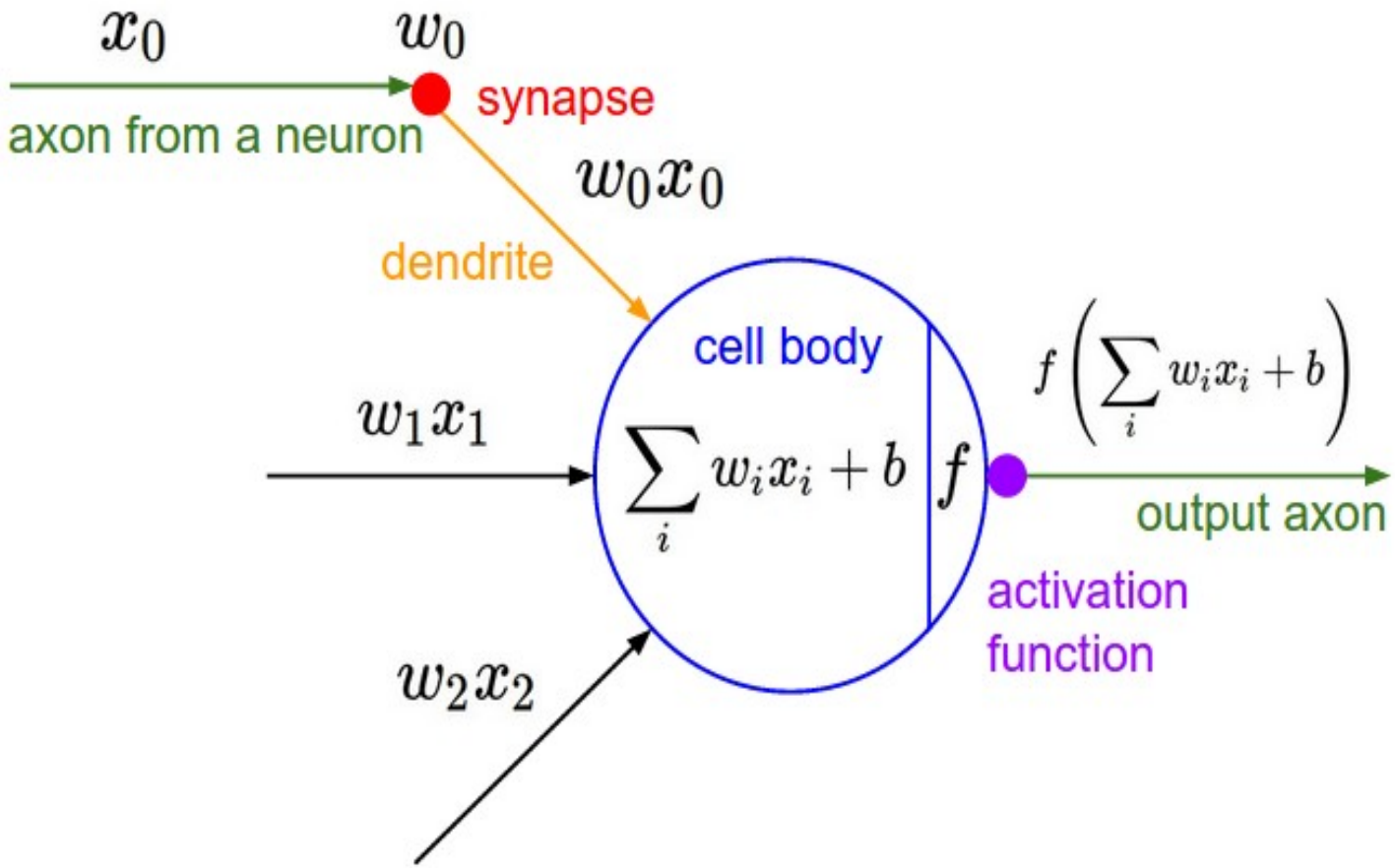


Image Source: Stanford

Many Weighted Sums

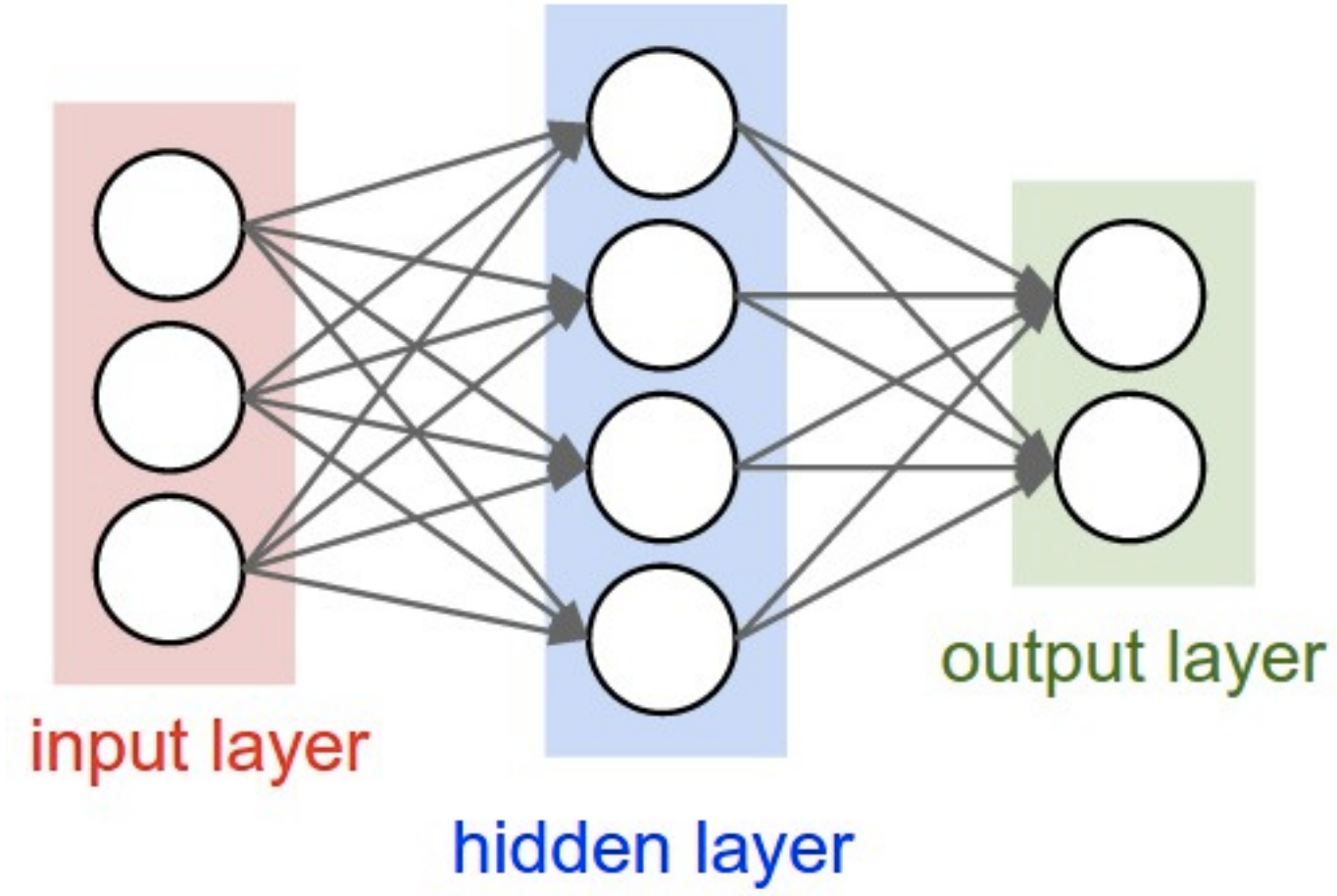
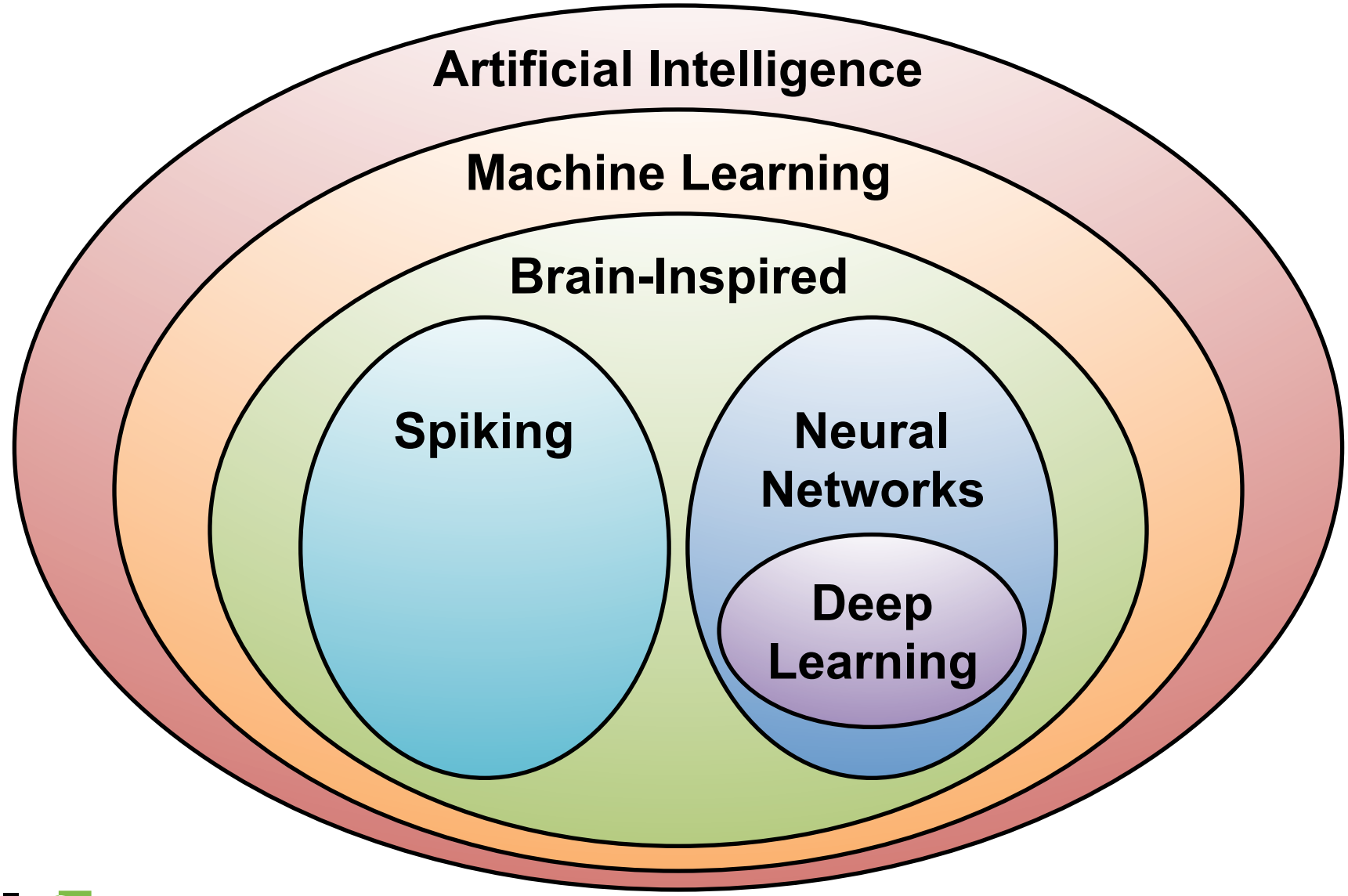


Image Source: Stanford

Deep Learning



What is Deep Learning?

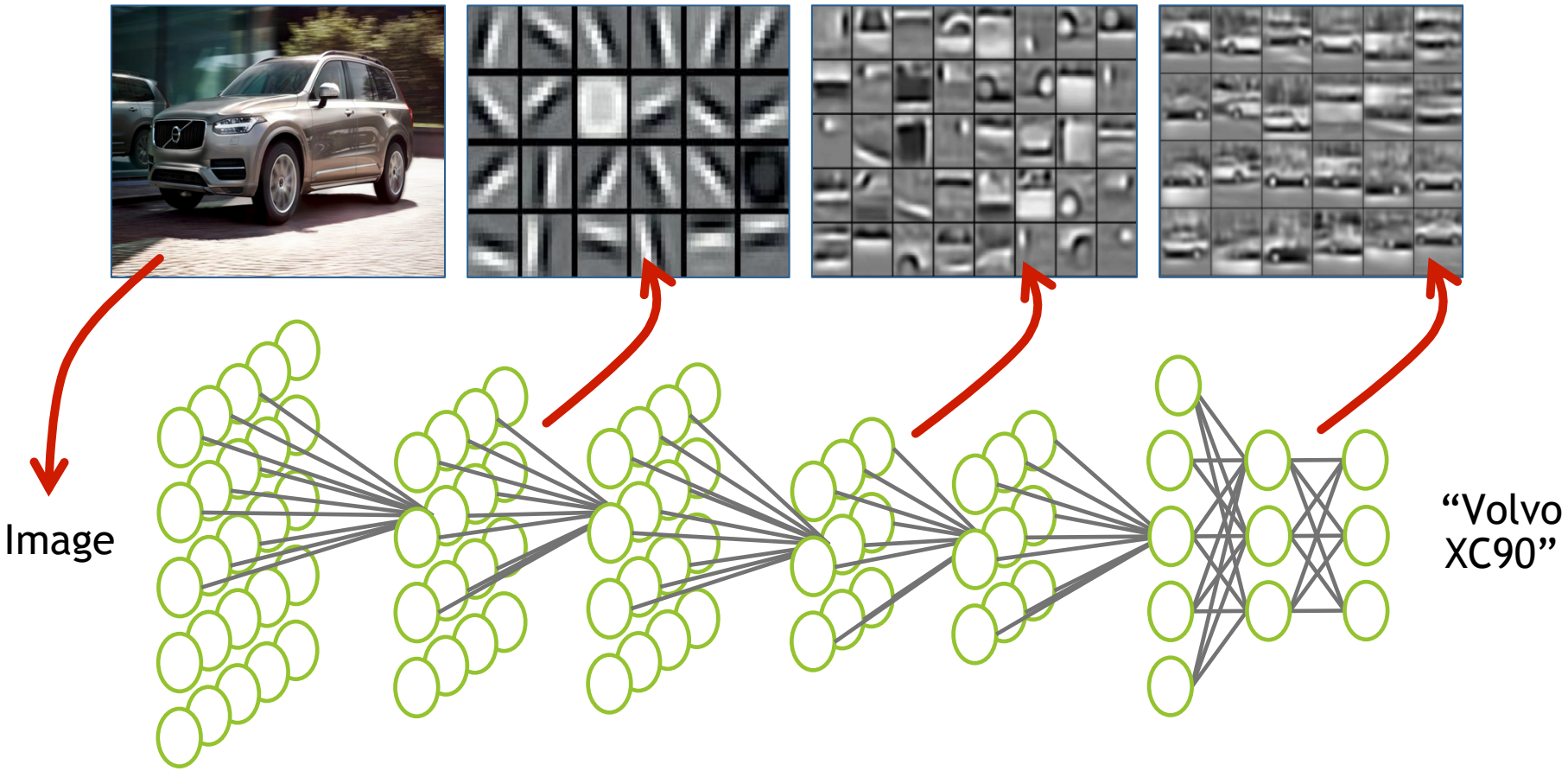


Image Source: [Lee et al., Comm. ACM 2011]

Why is Deep Learning Hot Now?

Big Data Availability

GPU Acceleration

New ML Techniques

facebook **350M** images uploaded per day

Walmart **2.5 Petabytes** of customer data hourly

YouTube **300 hours** of video uploaded every minute

ImageNet Challenge

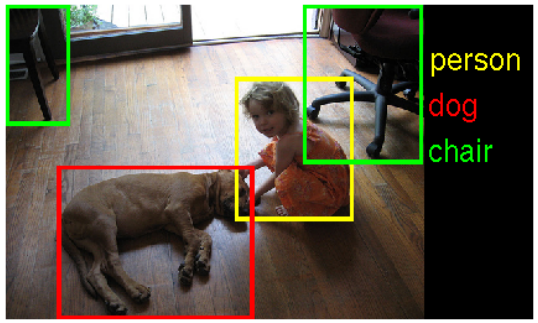
IMAGENET

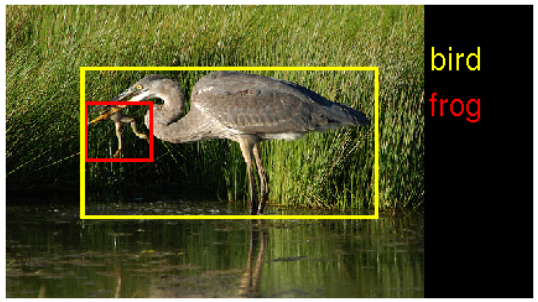
Image Classification Task:

1.2M training images • 1000 object categories

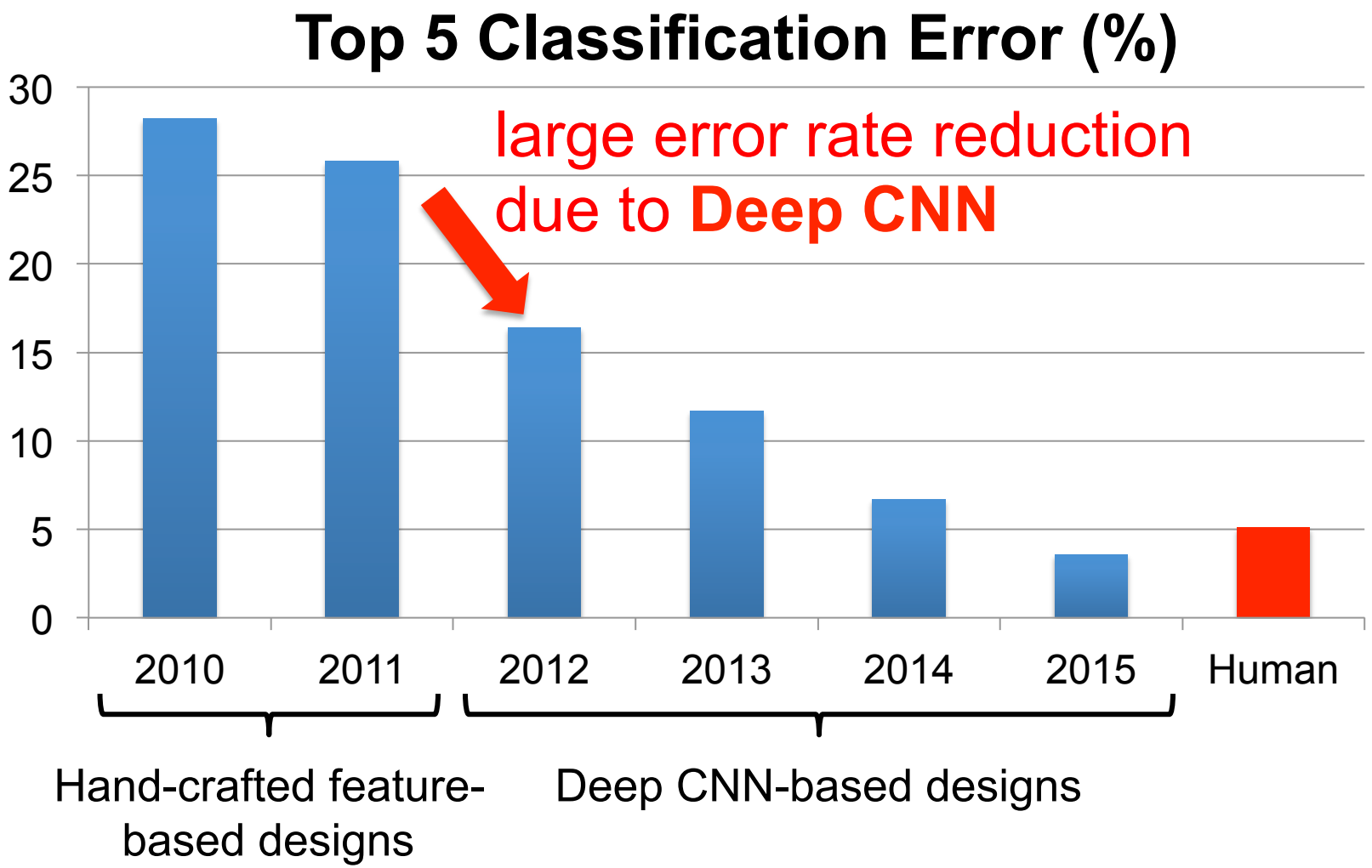
Object Detection Task:

456k training images • *200 object categories*



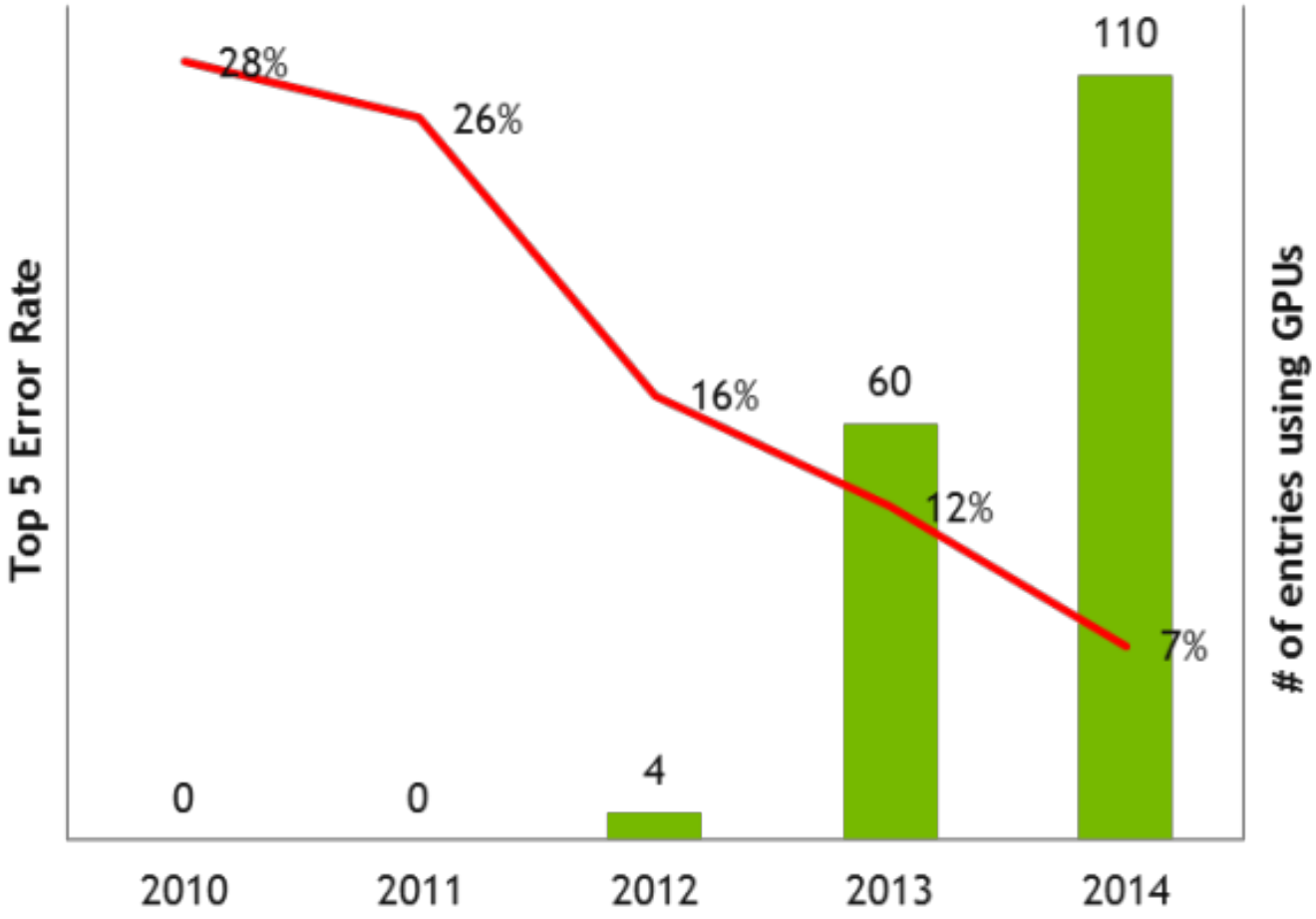


ImageNet: Image Classification Task



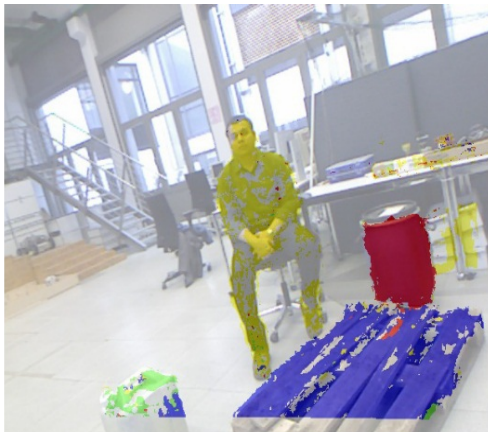
[Russakovsky et al., IJCV 2015]

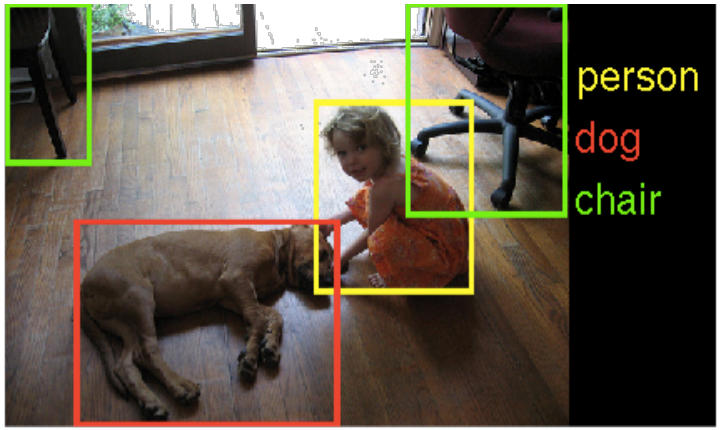
GPU Usage for ImageNet Challenge



Deep Learning on Images

- Image Classification
- Object Localization
- Object Detection
- Image Segmentation
- Action Recognition
- Image Generation



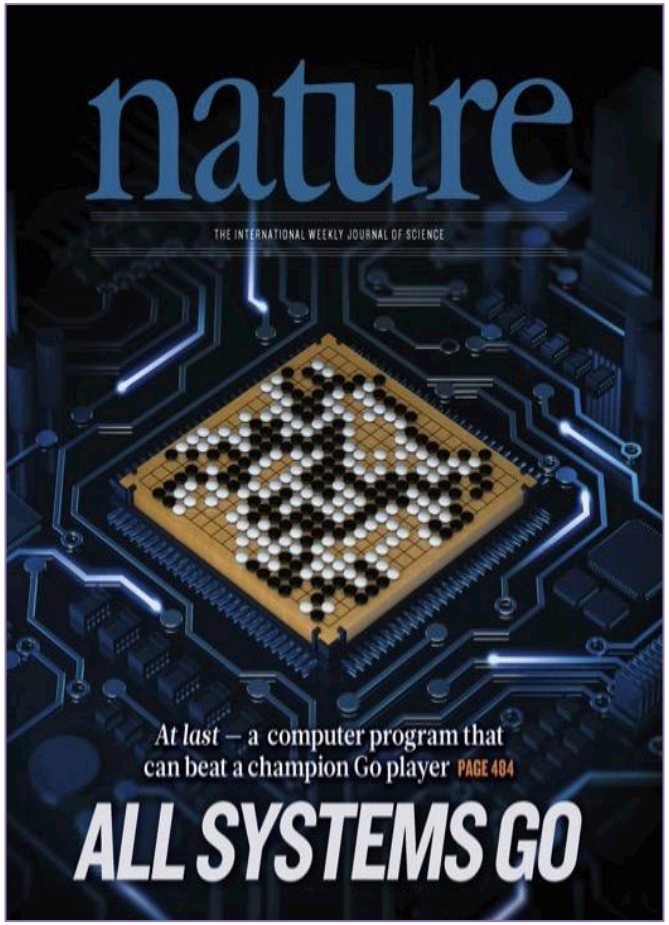


Deep Learning for Speech

- Speech Recognition
- Natural Language Processing
- Speech Translation
- Audio Generation

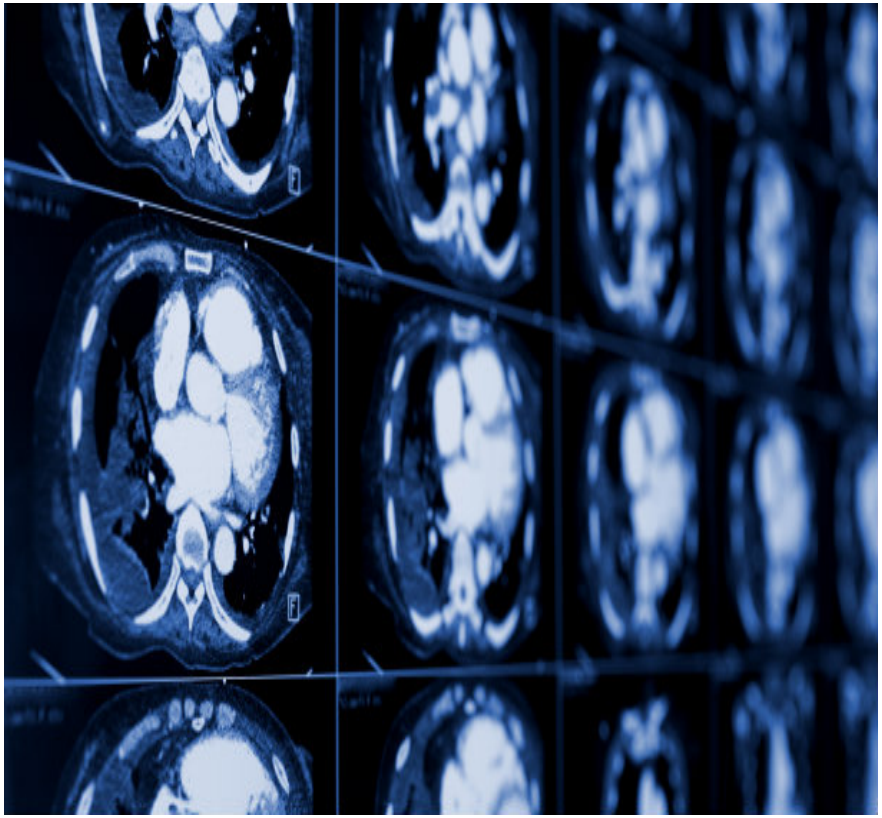
Deep Learning on Games

Google DeepMind AlphaGo



Medical Applications of Deep Learning

- **Brain Cancer Detection**



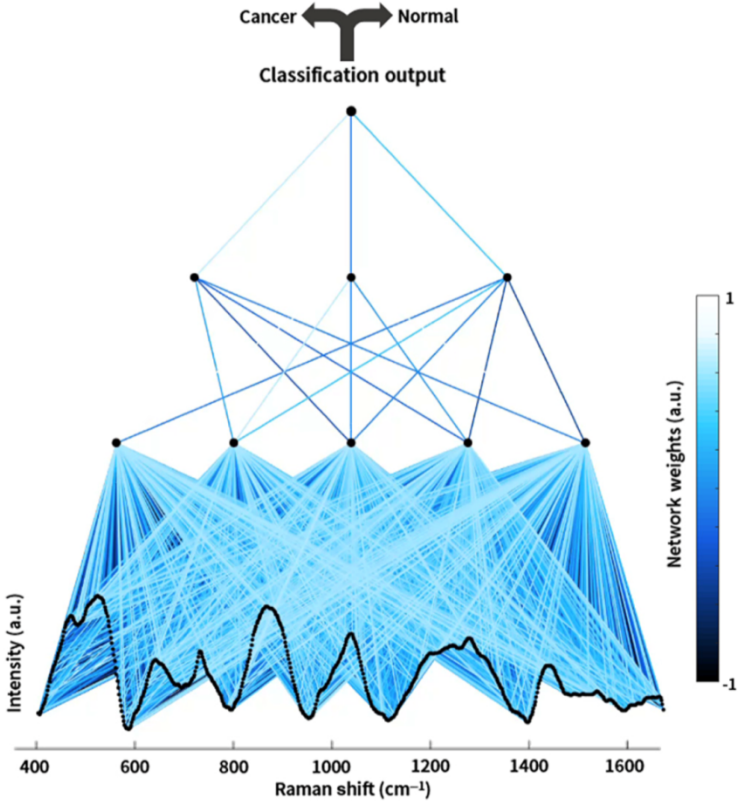
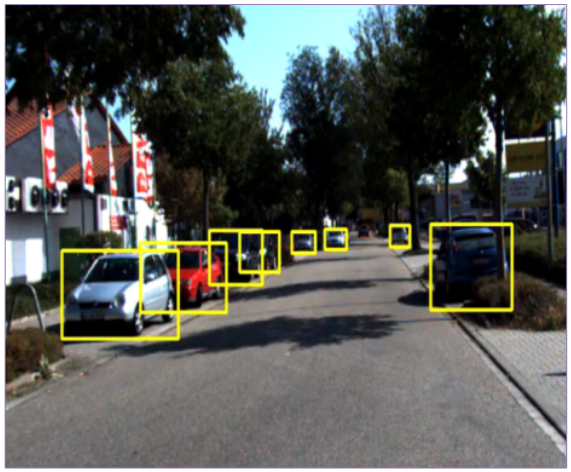
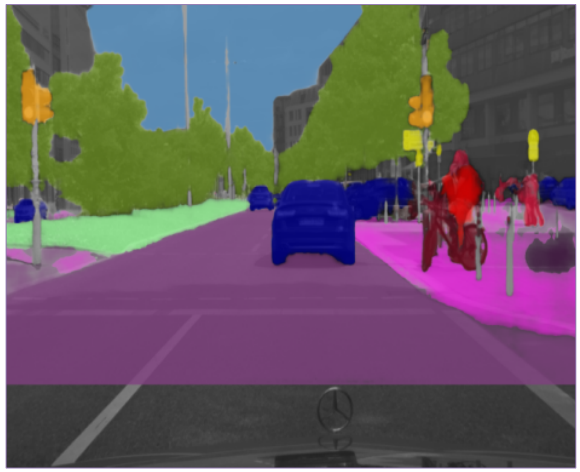
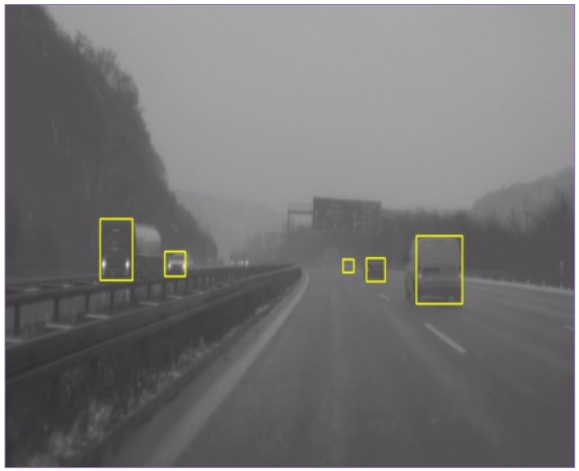


Image Source: [Jermyn et al., JBO 2016]

Deep Learning for Self-driving Cars





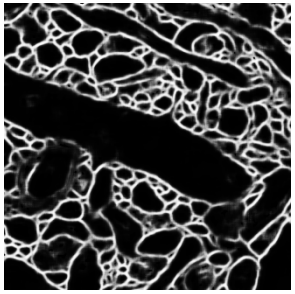


Connectomics – Finding Synapses

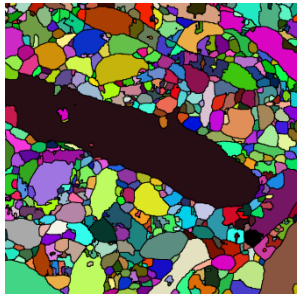
Machine Learning requires orders of Magnitude more computation than other parts

(1) EM

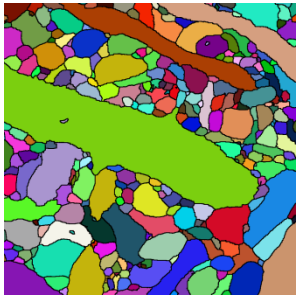
(2) ML Membrane Detection



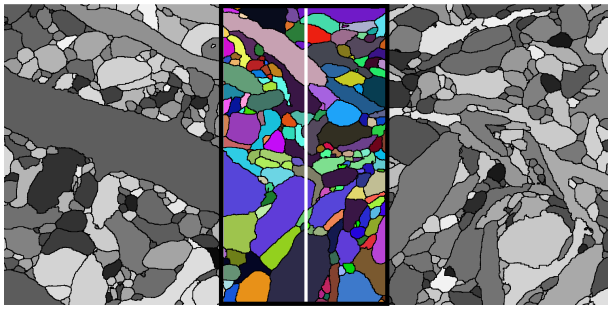
(3) Watershed



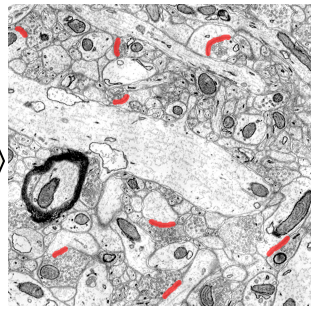
(4) Agglomeration



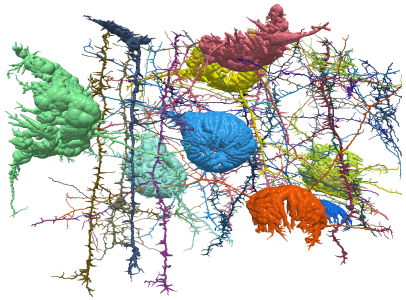
(5) Merging



(6) Synapses



(7) Skeletons



(8) Graph

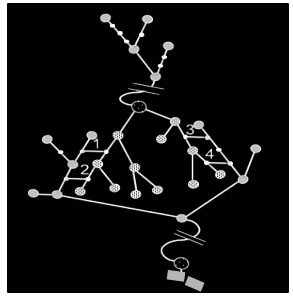


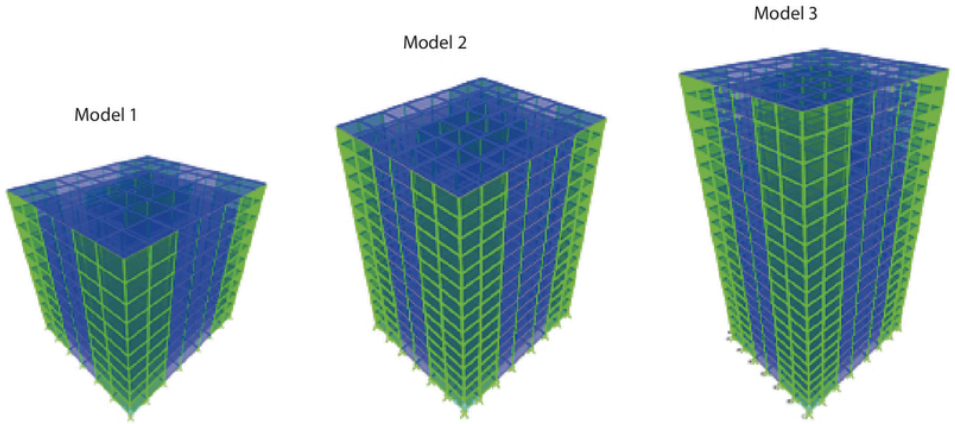
Image Source: MIT

Mature Applications

- **Image**
  - Classification: image to object class
  - Recognition: same as classification (except for faces)
  - Detection: assigning bounding boxes to objects
  - Segmentation: assigning object class to every pixel
- **Speech & Language**
  - Speech Recognition: audio to text
  - Translation
  - Natural Language Processing: text to meaning
  - Audio Generation: text to audio
- **Games**

Emerging Applications

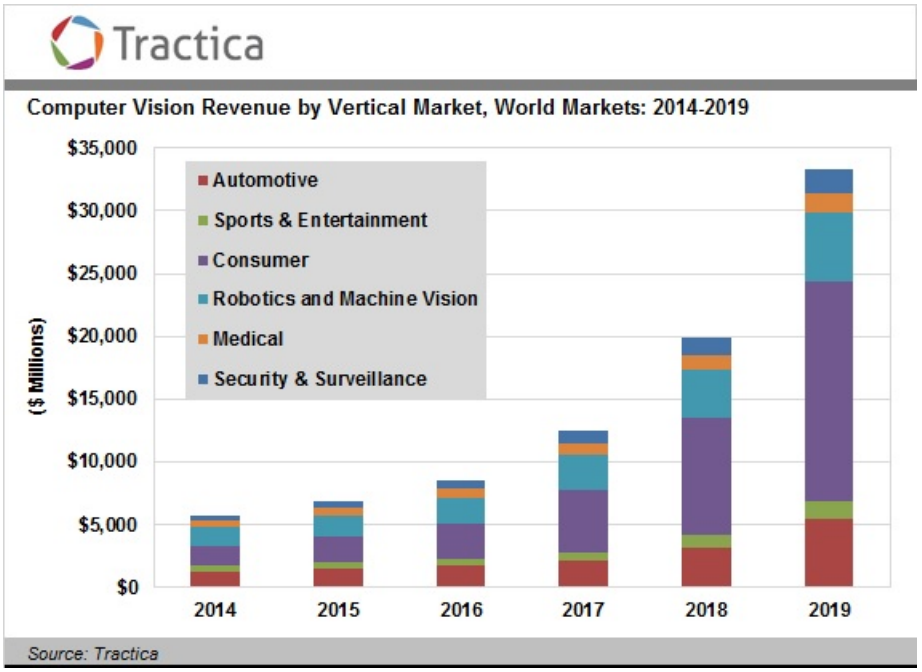
- **Medical** (Cancer Detection, Pre-Natal)
- **Finance** (Trading, Energy Forecasting, Risk)
- **Infrastructure** (Structure Safety and Traffic)
- Weather Forecasting and Event Detection



This tutorial will focus on image classification

Opportunities

$500B Market over 10 Years!



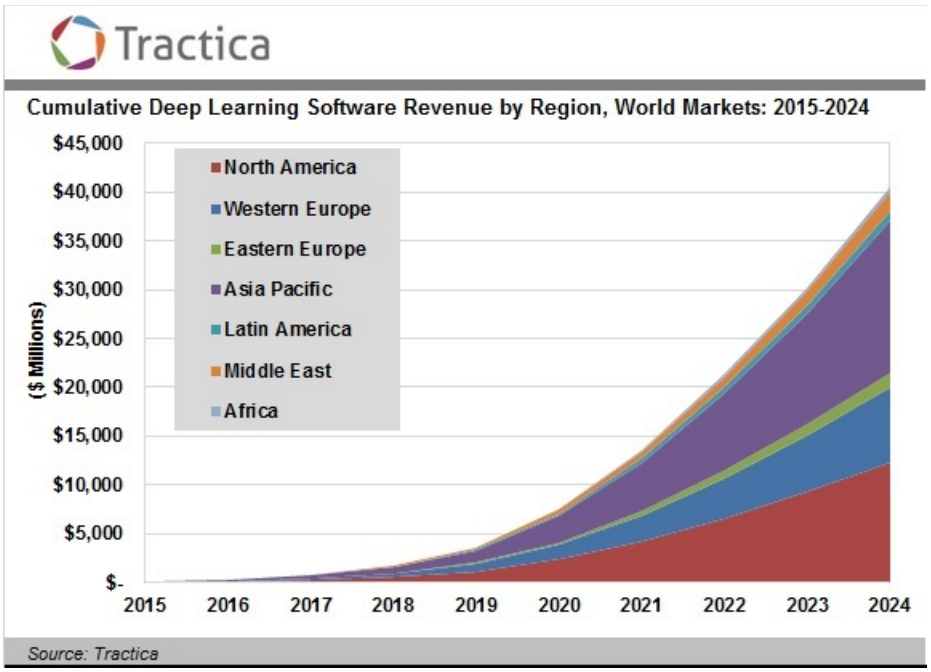


Image Source: Tractica

Opportunities

From EE Times – September 27, 2016

”Today the job of training machine learning models is limited by compute, if we had faster processors we’d run bigger models…in practice we train on a reasonable subset of data that can finish in a matter of months. We could use improvements of several orders of magnitude – 100x or greater.”

– Greg Diamos, Senior Researcher, SVAIL, Baidu

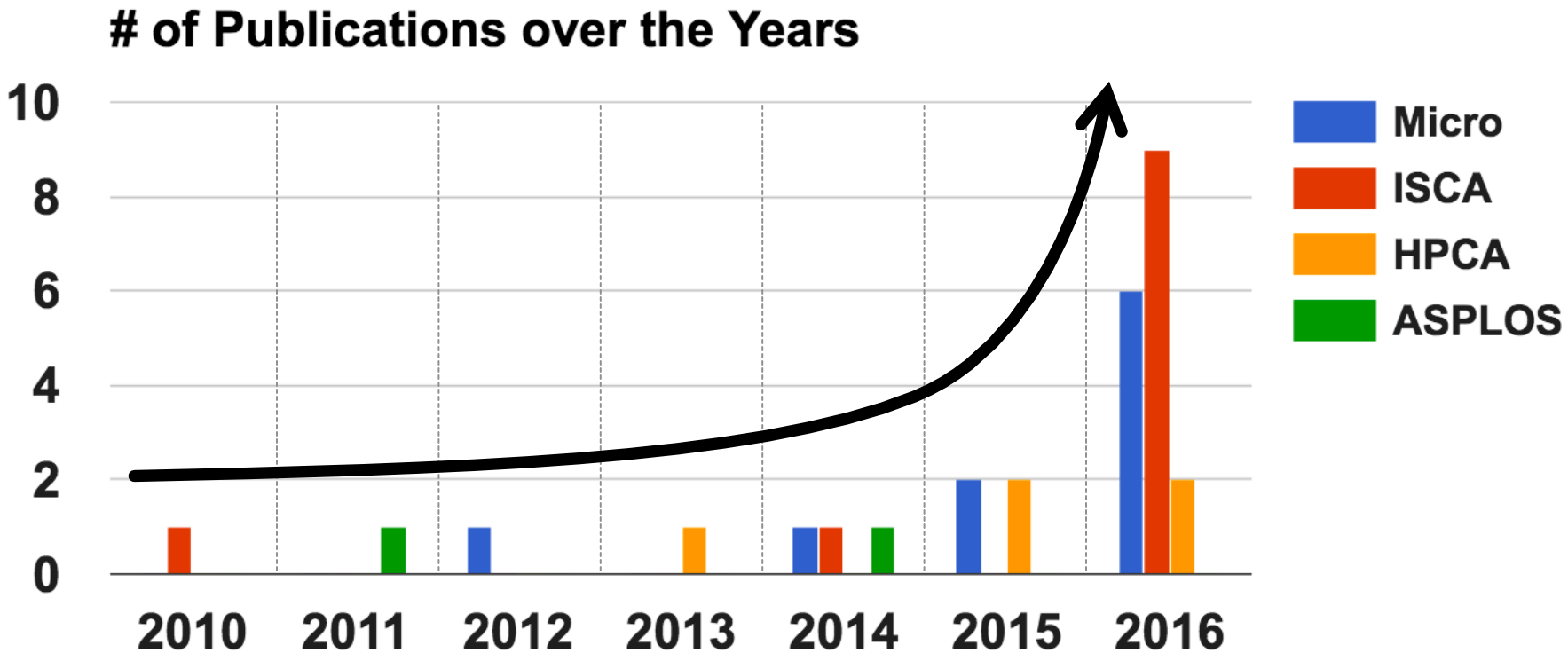
Overview of Deep Neural Networks

DNN Timeline

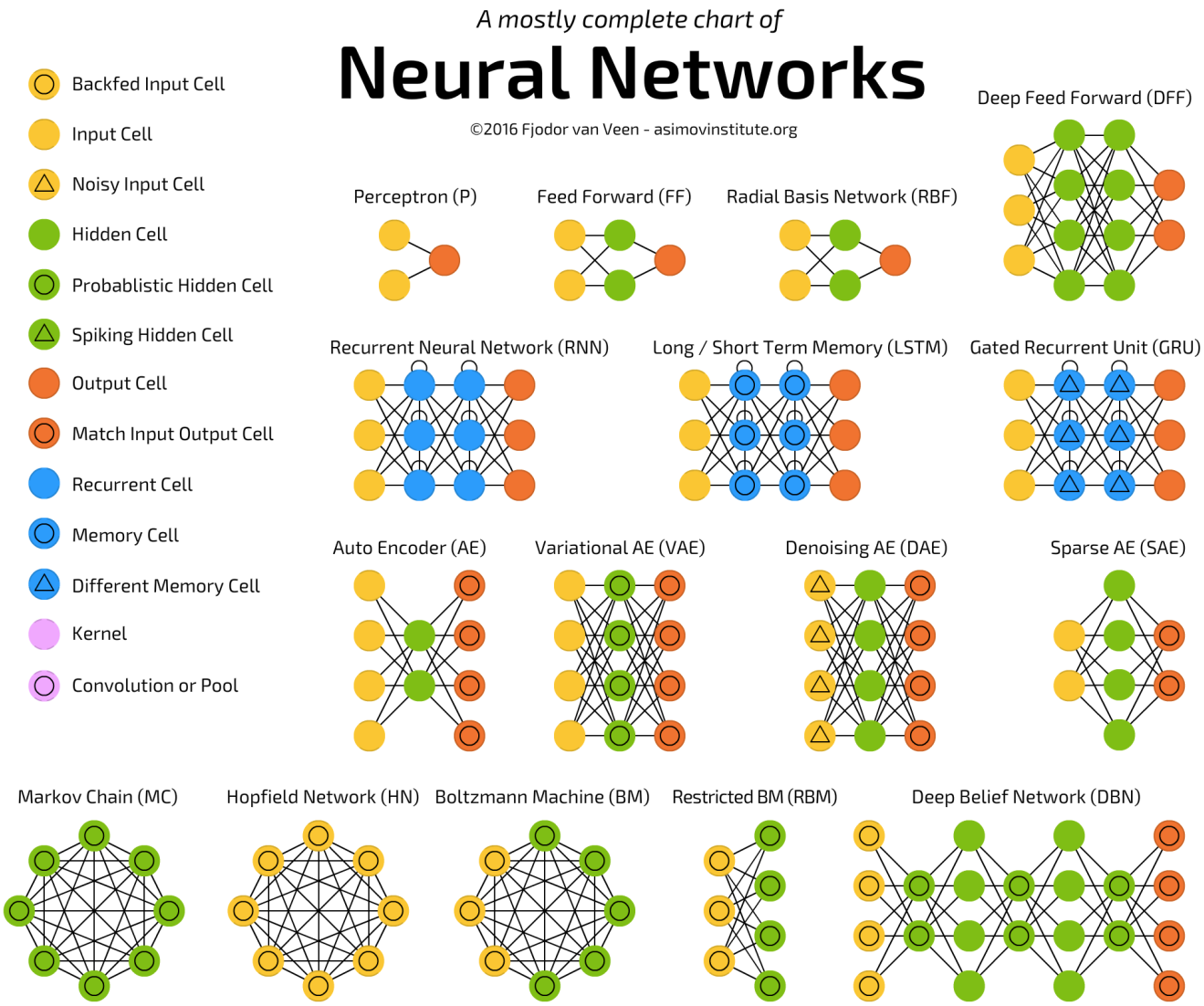
- **1940s: Neural networks were proposed**
- **1960s: Deep neural networks were proposed**
- **1990s: Early hardware for shallow neural nets**
  - Example: Intel ETANN (1992)
- **1998: LeNet for MNIST**
- **2011: Speech recognition using DNN (Microsoft)**
- **2012: Deep learning starts supplanting traditional ML**
  - AlexNet for image classification
- **Early 2010s: Rise of DNN accelerator research**
  - Examples: Neuflow, DianNao, etc.

Publications at Architecture Conferences

- MICRO, ISCA, HPCA, ASPLOS



So Many Neural Networks!



http://www.asimovinstitute.org/neural-network-zoo/

DNN Terminology 101

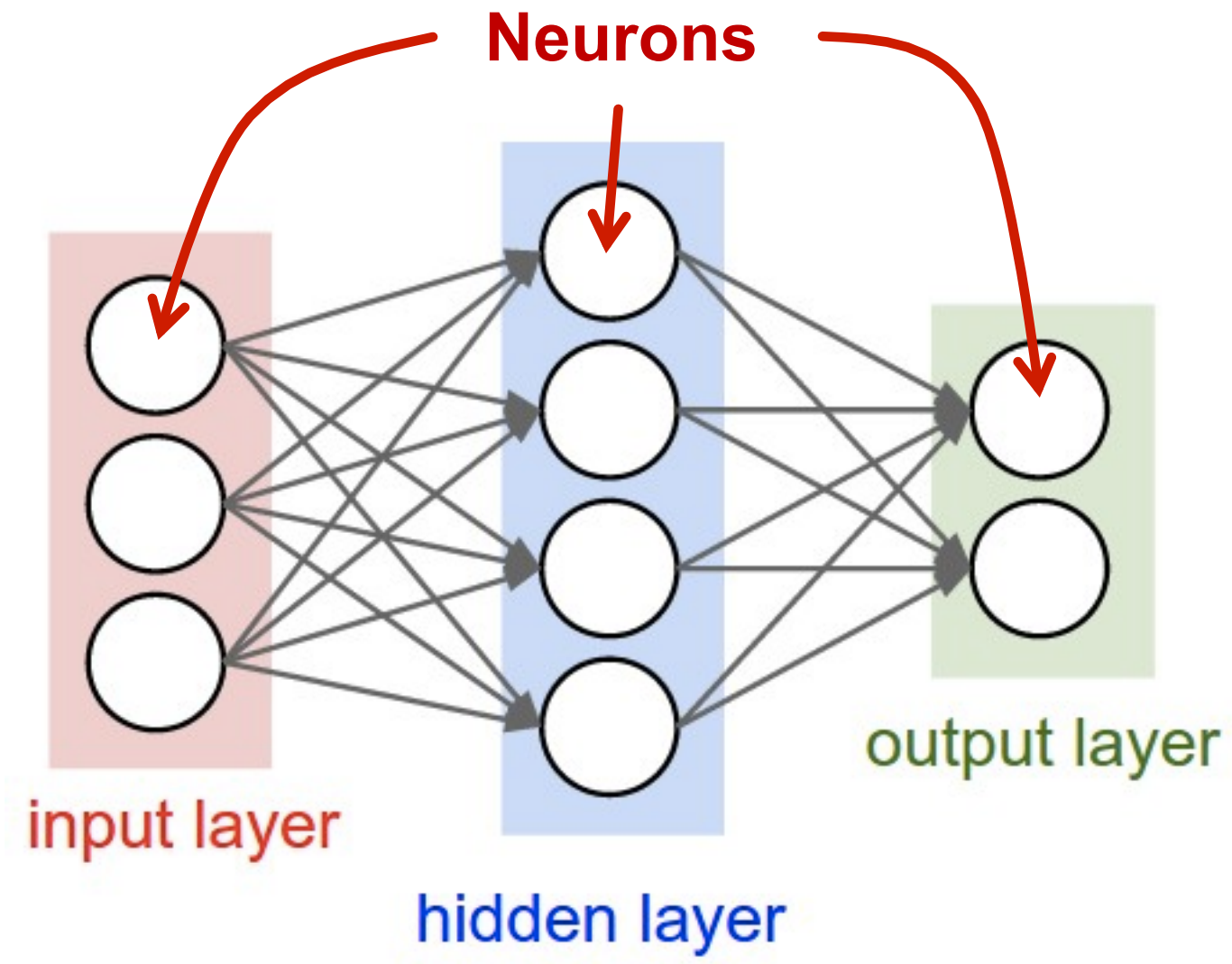


Image Source: Stanford

DNN Terminology 101

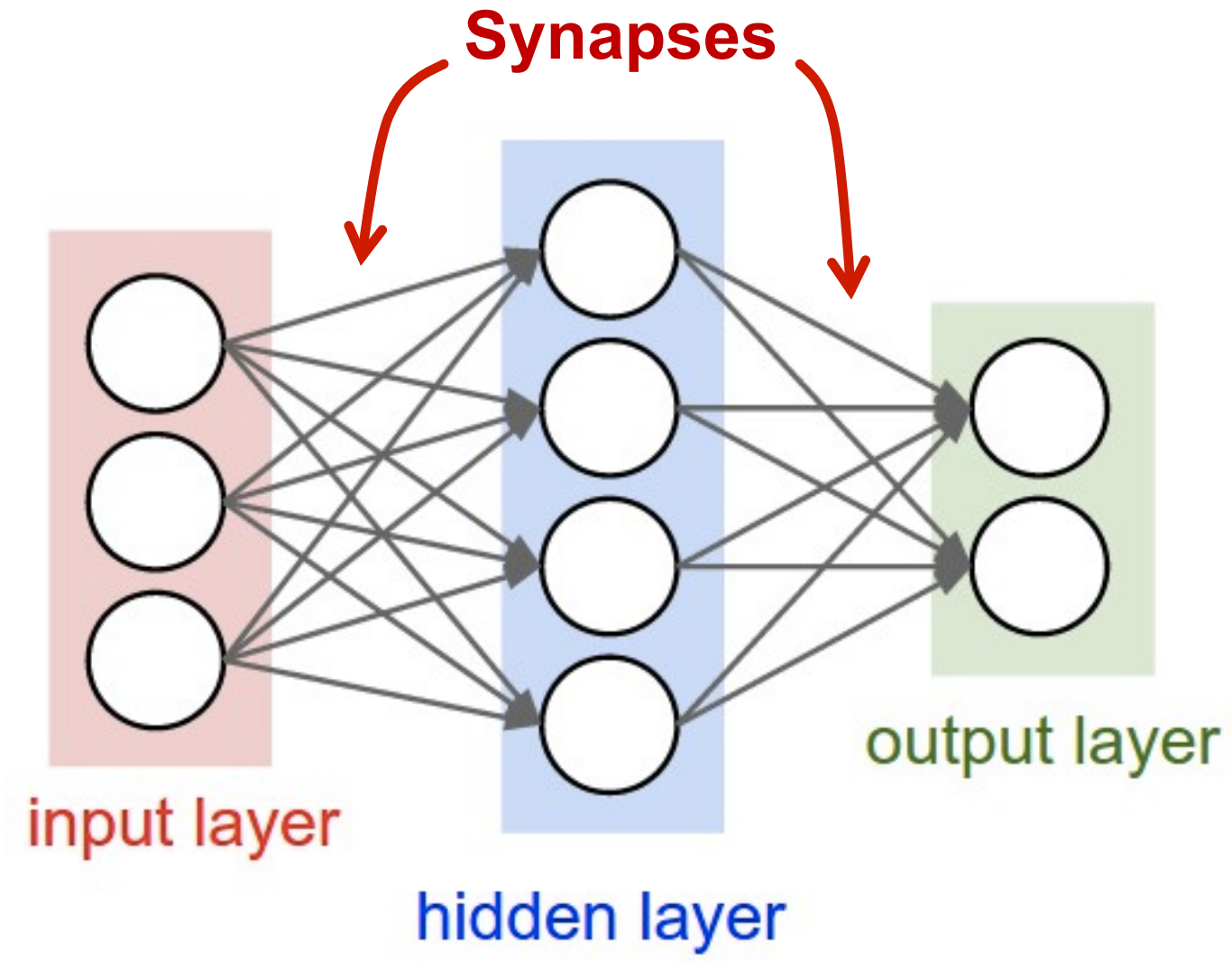


Image Source: Stanford

DNN Terminology 101

Each synapse has a **weight** for neuron **activation**

$$Y_j = activation(\sum_{i=1}^{3} W_{ij} * X_i$$

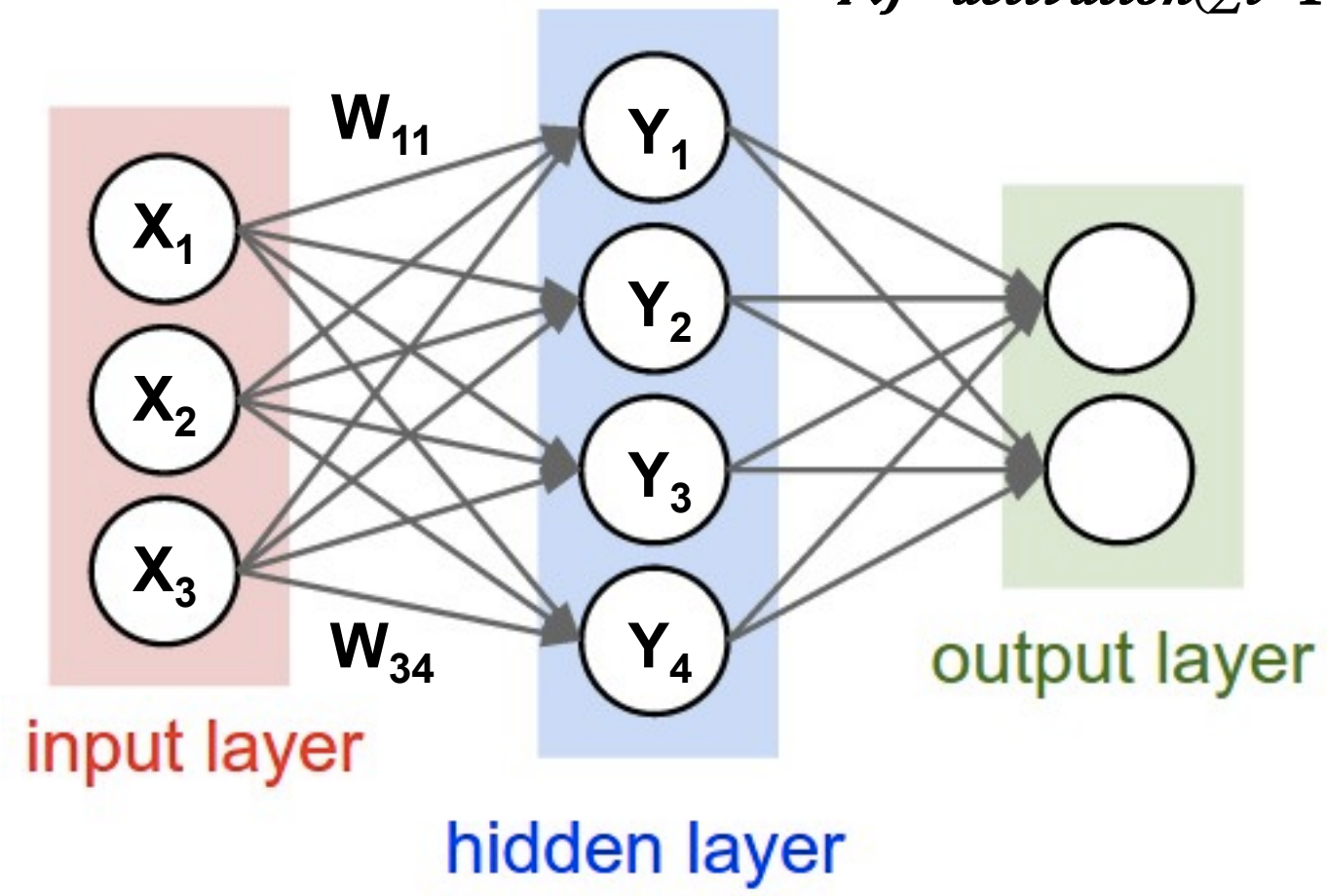


Image Source: Stanford

DNN Terminology 101

Weight Sharing: multiple synapses use the **same weight value**

$$Y_j = activation(\sum_{i=1}^{3} W_{ij} * X_$$

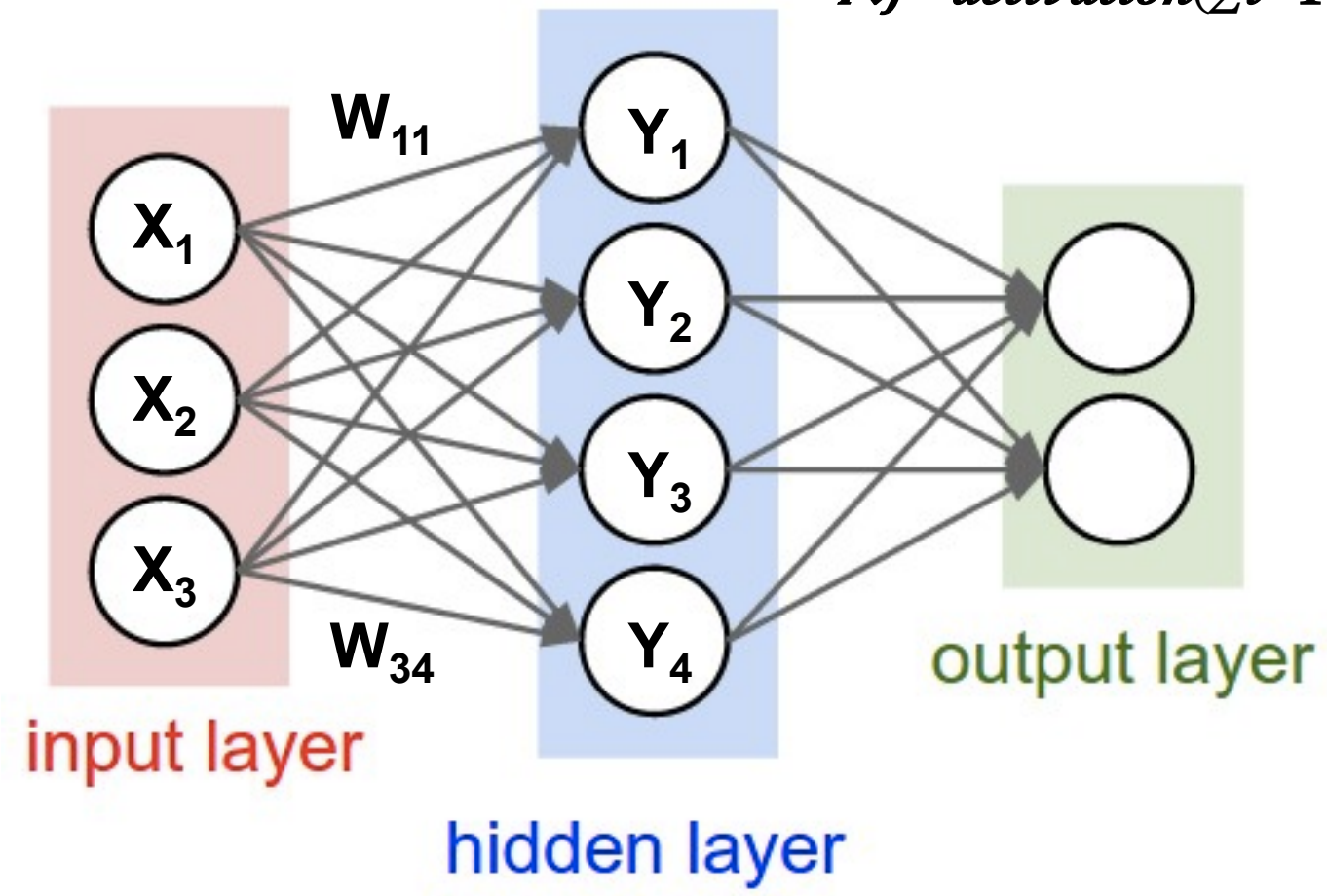


Image Source: Stanford

DNN Terminology 101

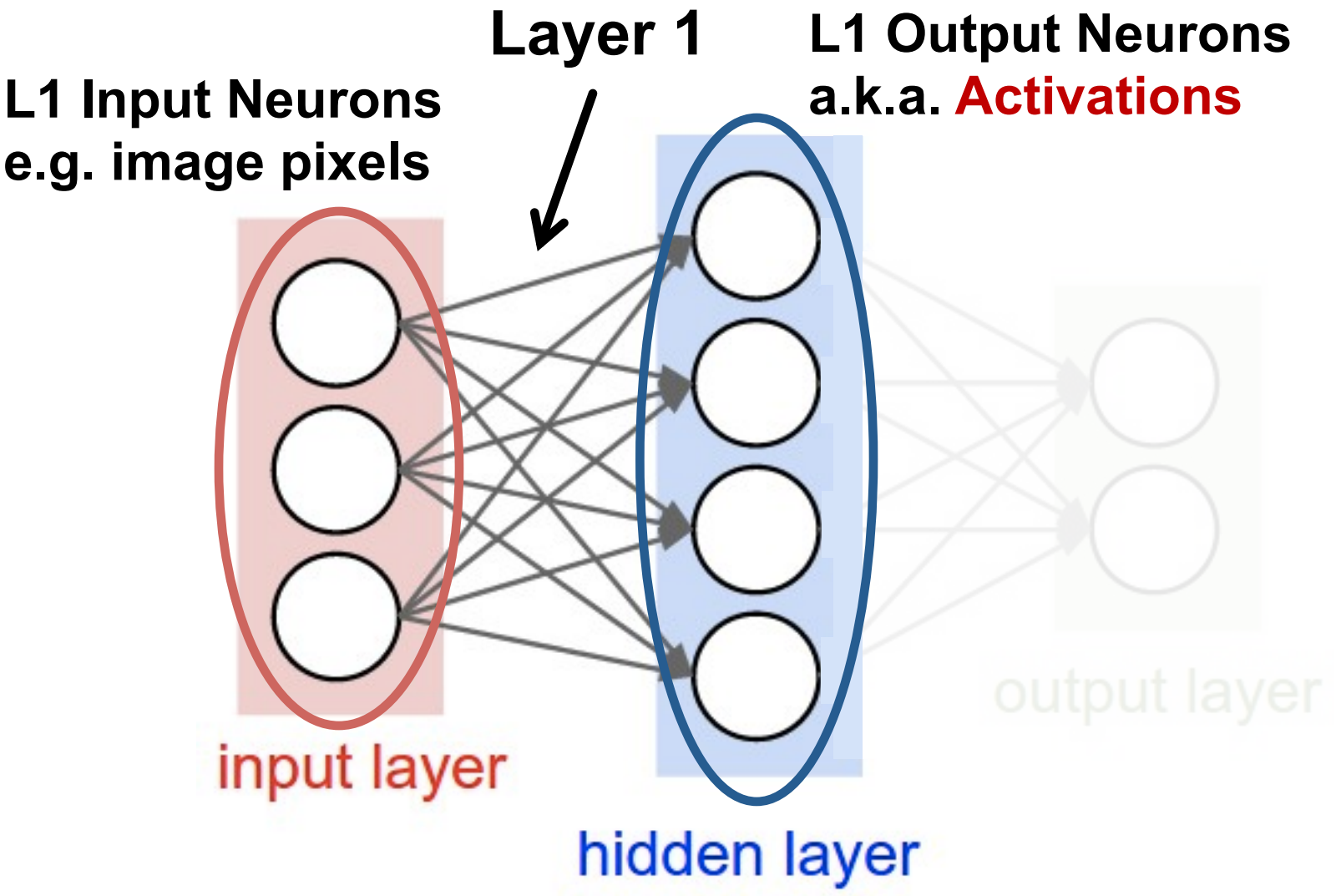


Image Source: Stanford

DNN Terminology 101

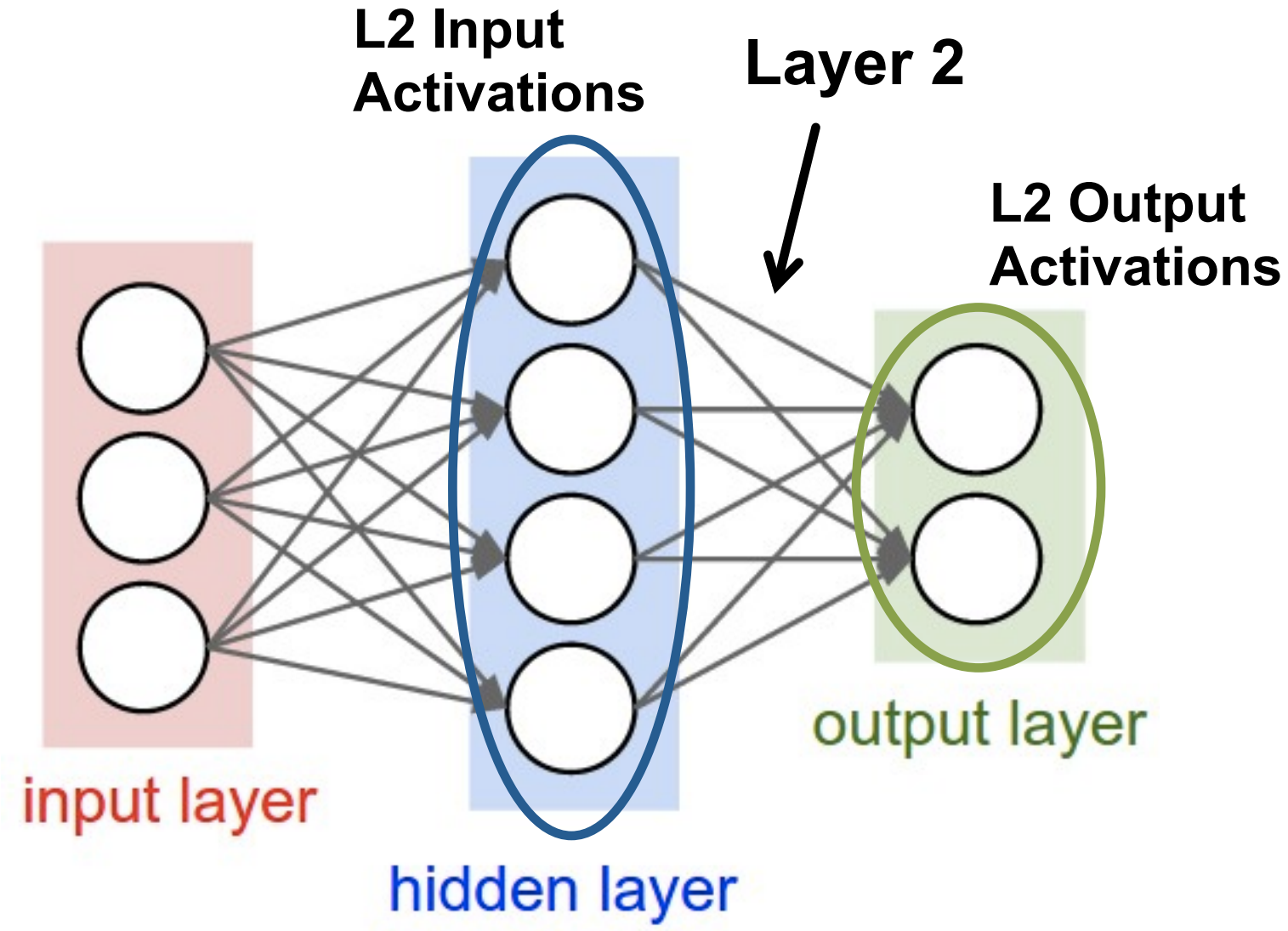


Image Source: Stanford

DNN Terminology 101

Fully-Connected: all i/p neurons connected to all o/p neurons

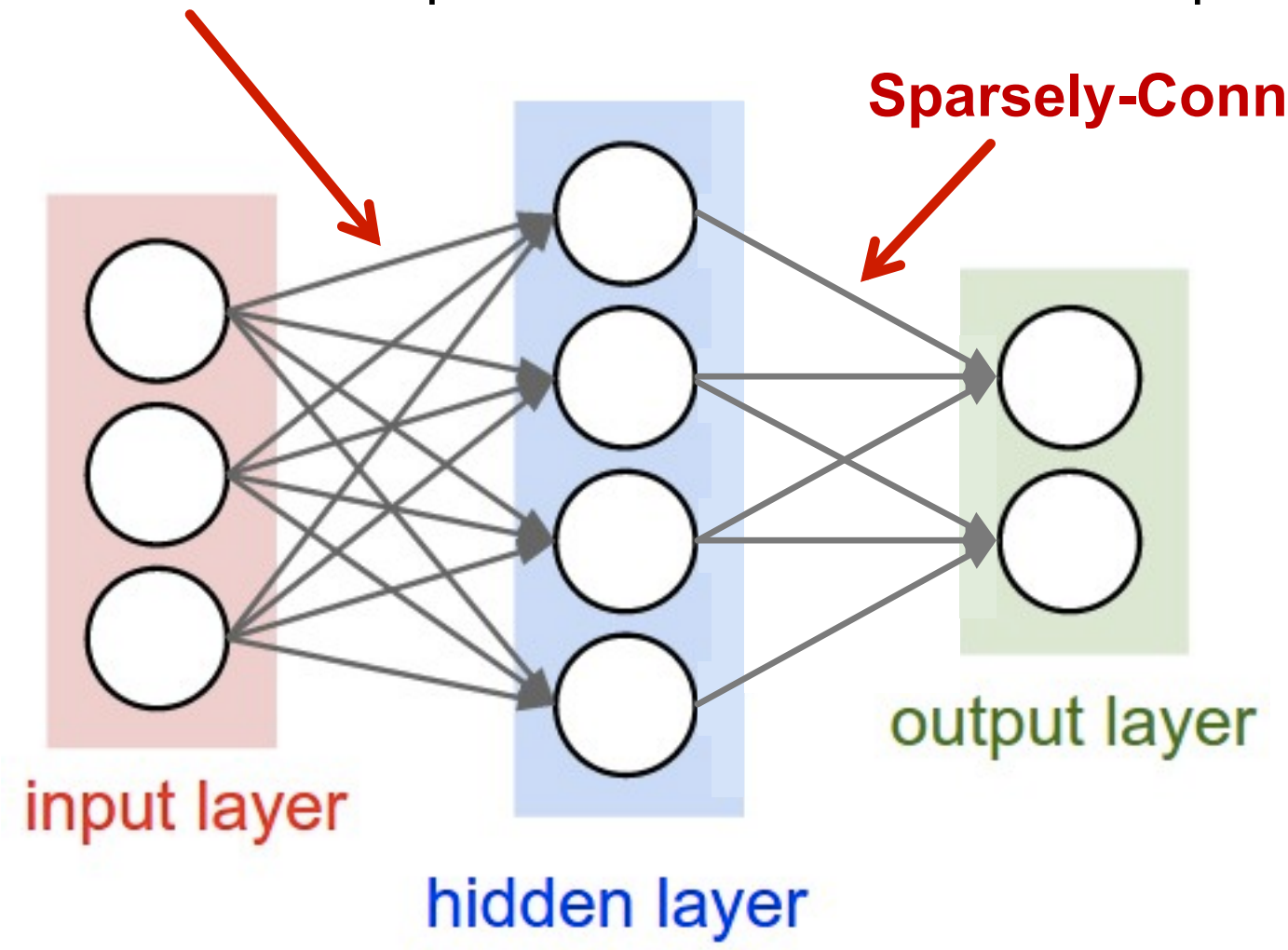


Image Source: Stanford

DNN Terminology 101

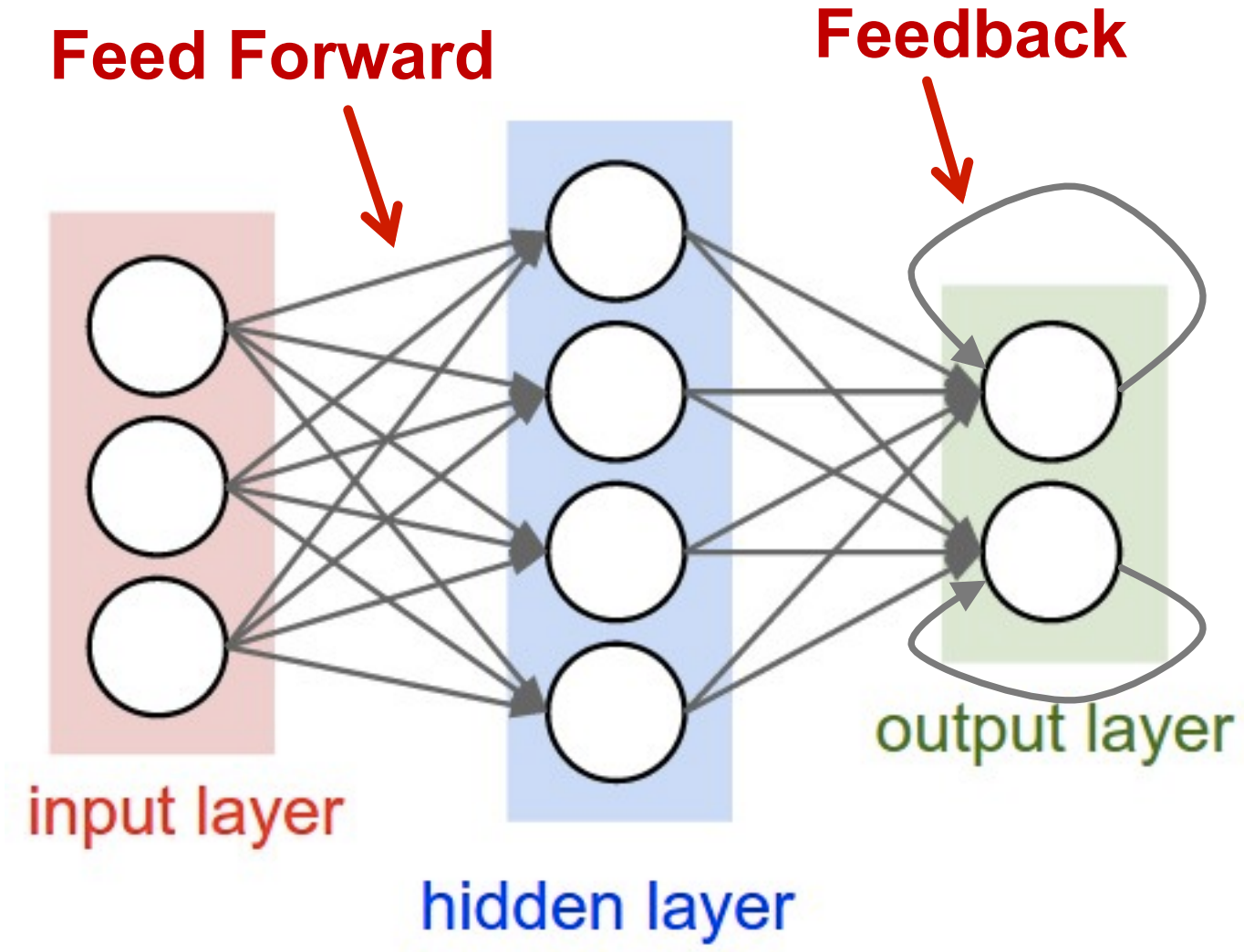


Image Source: Stanford

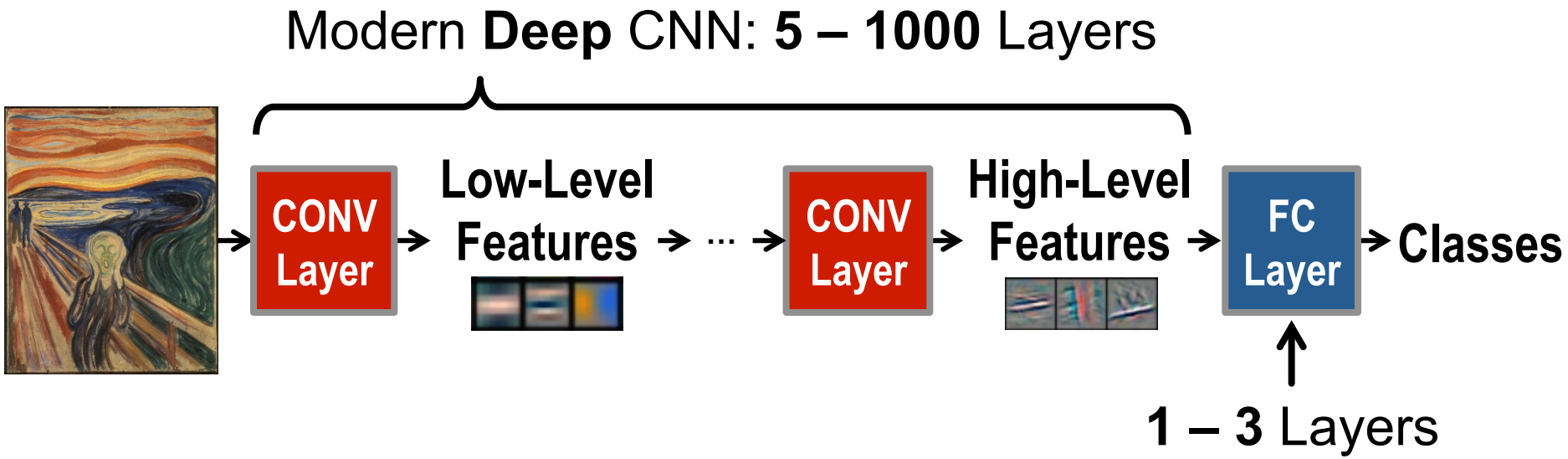
Popular Types of DNNs

- **Fully-Connected NN**
  - feed forward, a.k.a. multilayer perceptron (MLP)
- **Convolutional NN (CNN)**
  - feed forward, sparsely-connected w/ weight sharing
- **Recurrent NN (RNN)**
  - feedback
- **Long Short-Term Memory (LSTM)**
  - feedback + Storage

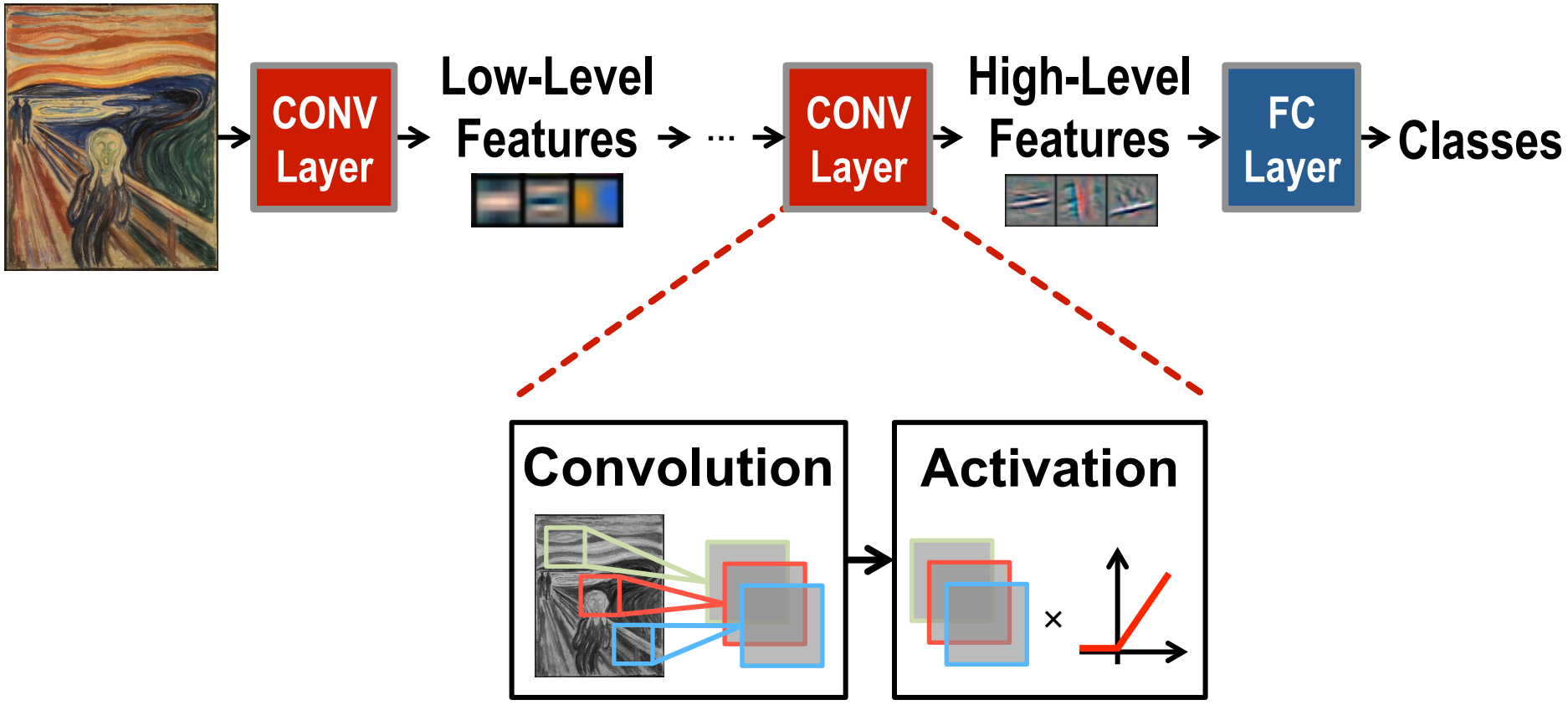
Inference vs. Training

- **Training: Determine weights**
  - **Supervised:**
    - **Training set has inputs and outputs, i.e., labeled**
  - **Reinforcement:**
    - **Output assessed via rewards and punishments**
  - **Unsupervised:**
    - **Training set is unlabeled**
  - **Semi-supervised:**
    - **Training set is partially labeled**
- **Inference: Apply weights to determine output**

Deep Convolutional Neural Networks

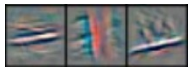


Deep Convolutional Neural Networks

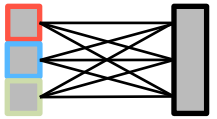


Deep Convolutional Neural Networks

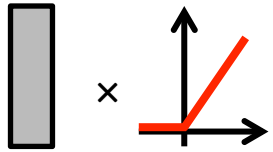
CONV Layer → Low-Level Features → ··· → CONV Layer → High-Level Features → FC Layer → Classes



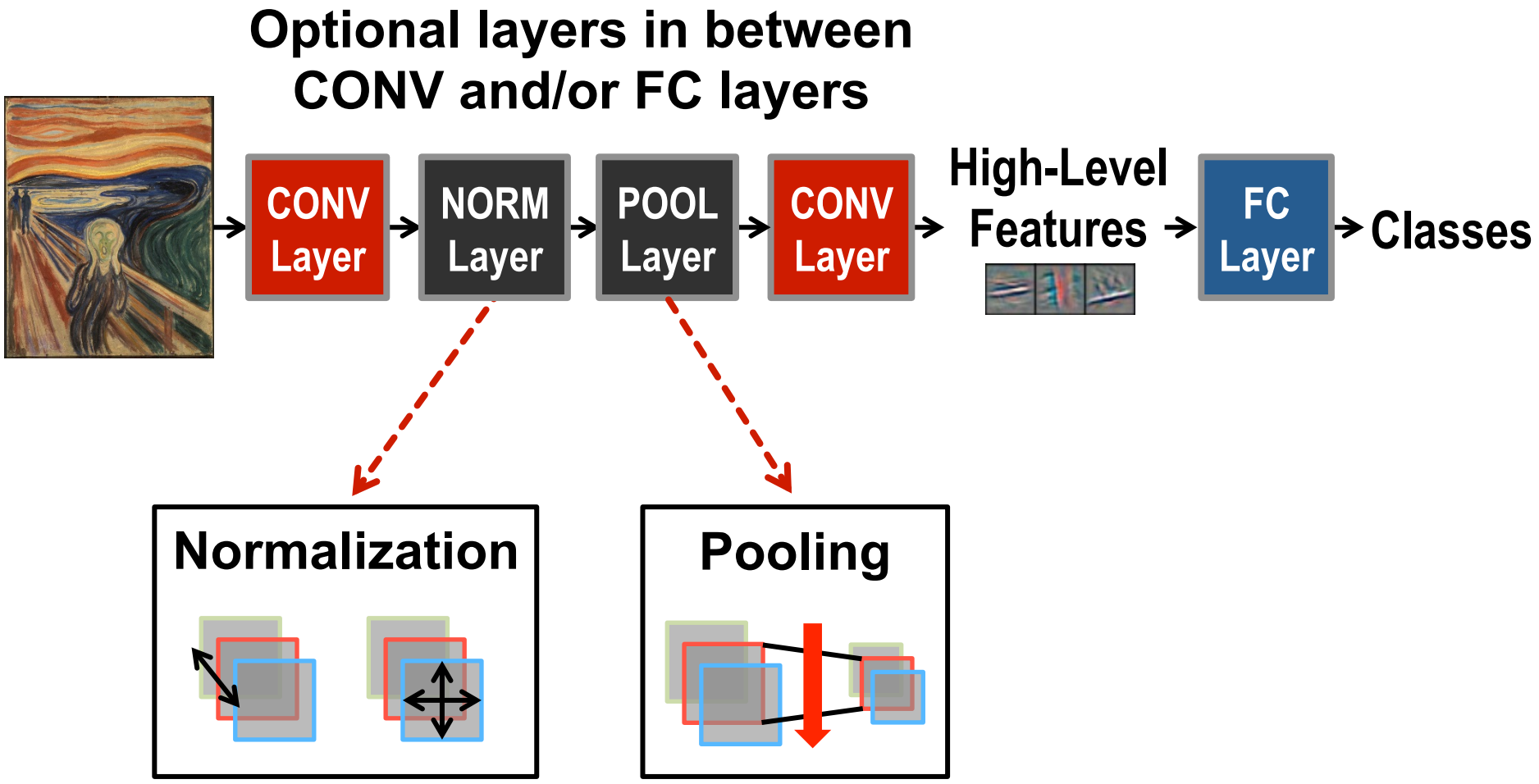
Fully Connected



Activation

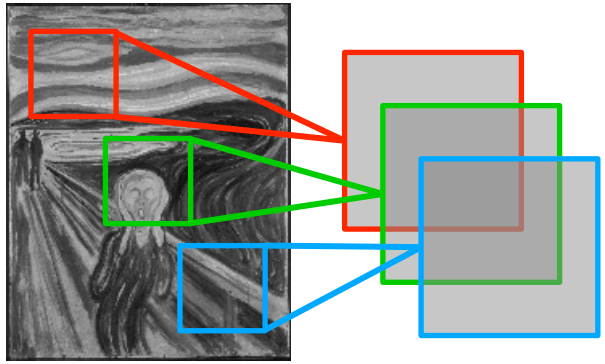


Deep Convolutional Neural Networks



Deep Convolutional Neural Networks

CONV Layer → NORM Layer → POOL Layer → CONV Layer → High-Level Features → FC Layer → Classes

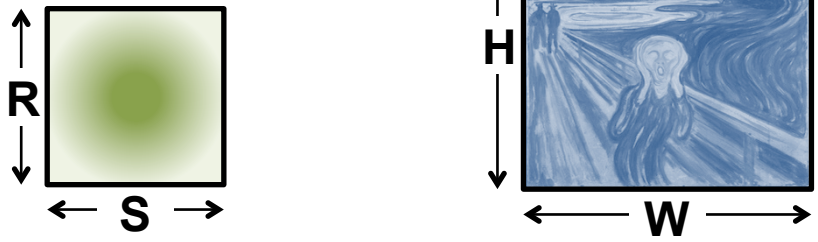


Convolutions account for more than 90% of overall computation, dominating **runtime** and **energy consumption**

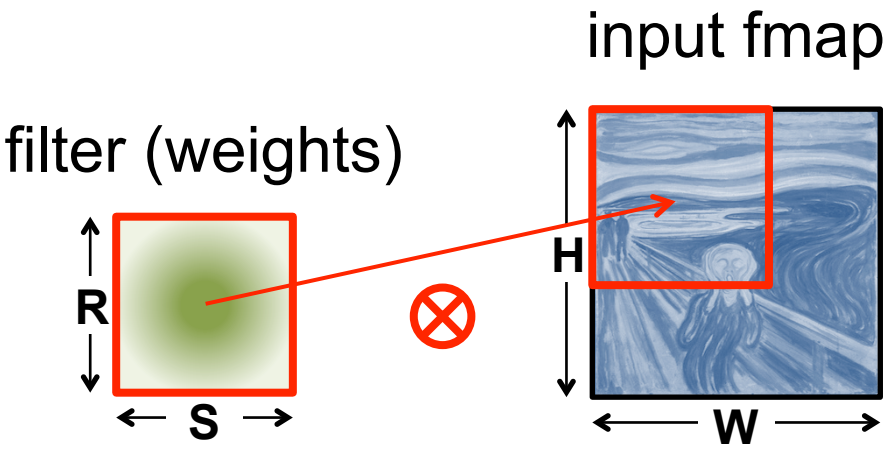
Convolution (CONV) Layer

a plane of input activations
a.k.a. **input feature map (fmap)**

filter (weights)

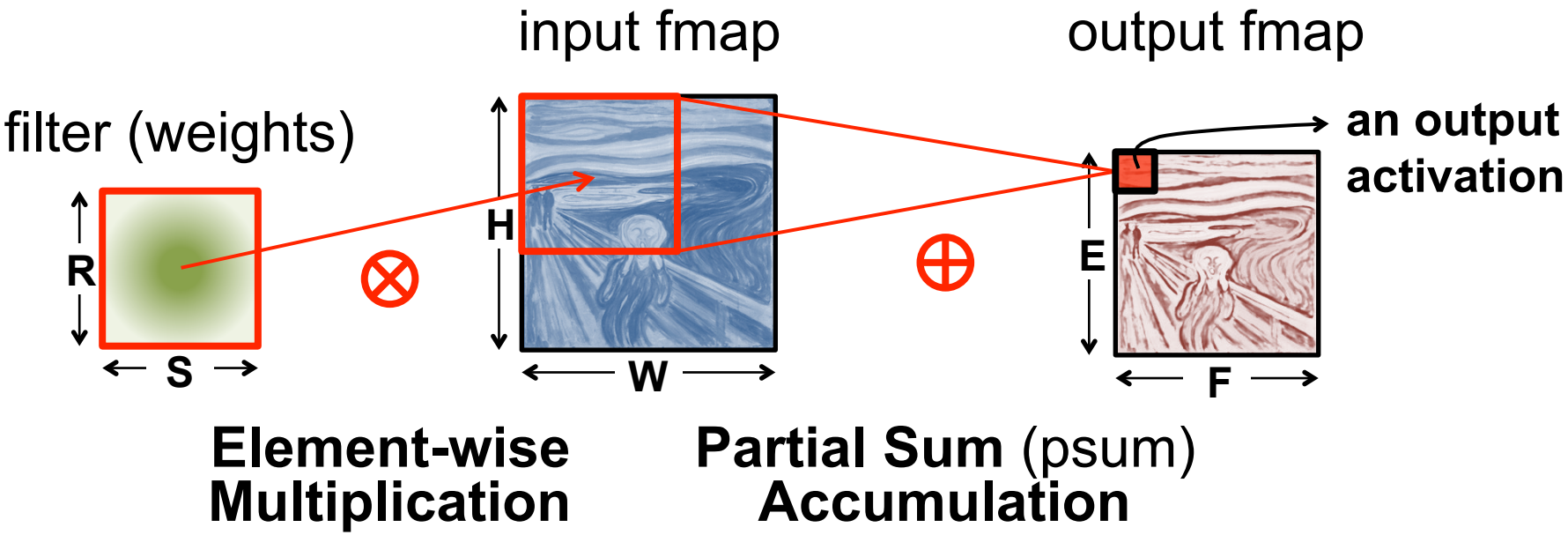


Convolution (CONV) Layer

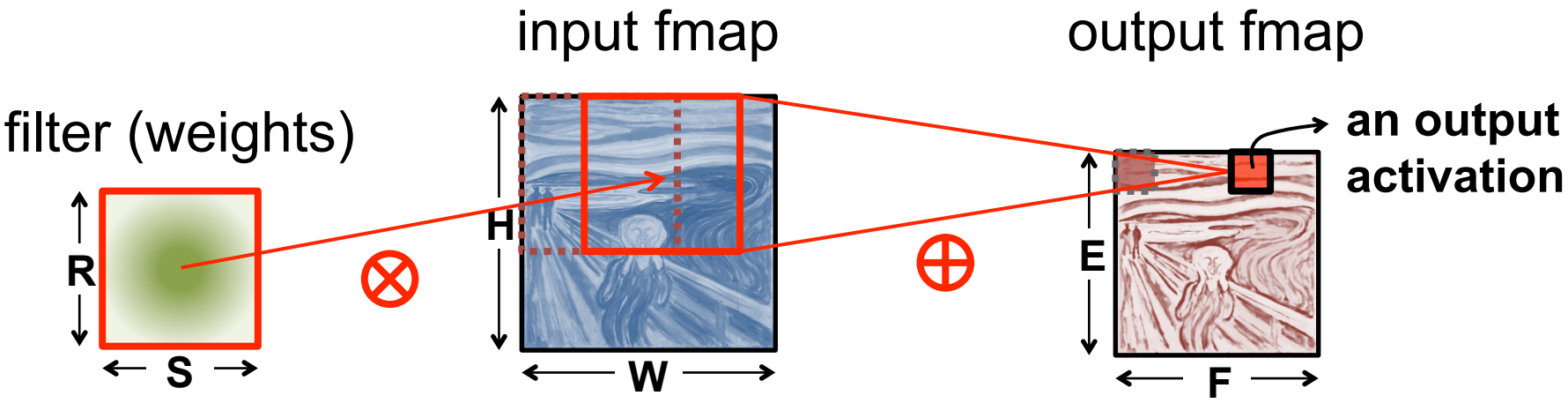


Element-wise Multiplication

Convolution (CONV) Layer

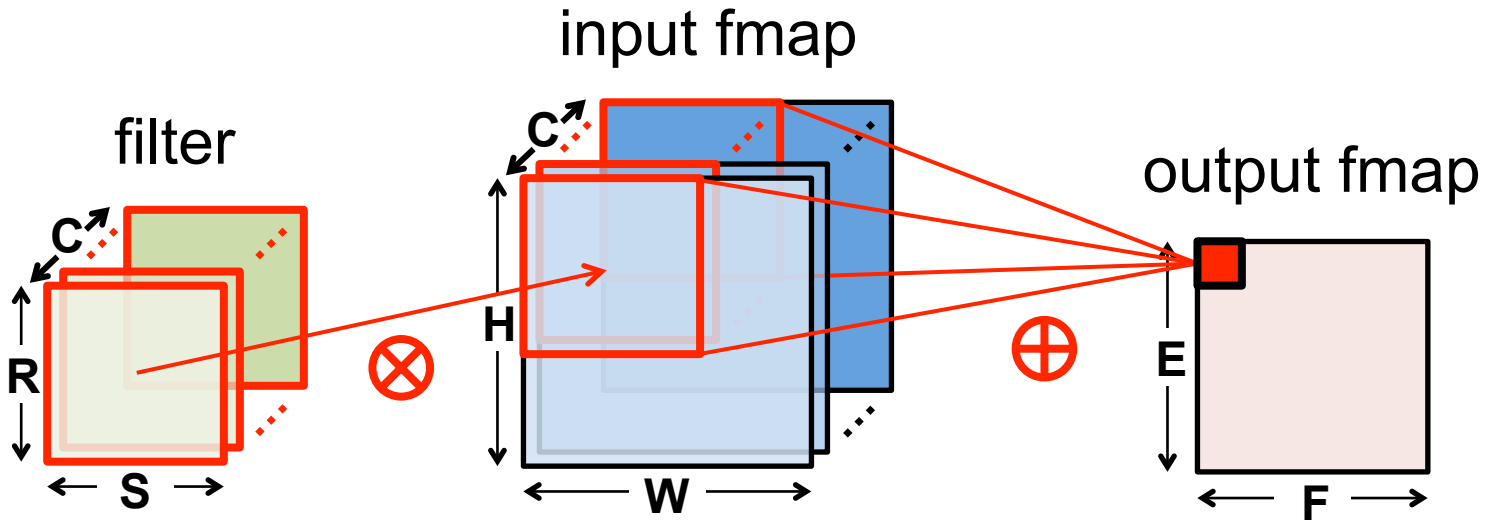


Convolution (CONV) Layer



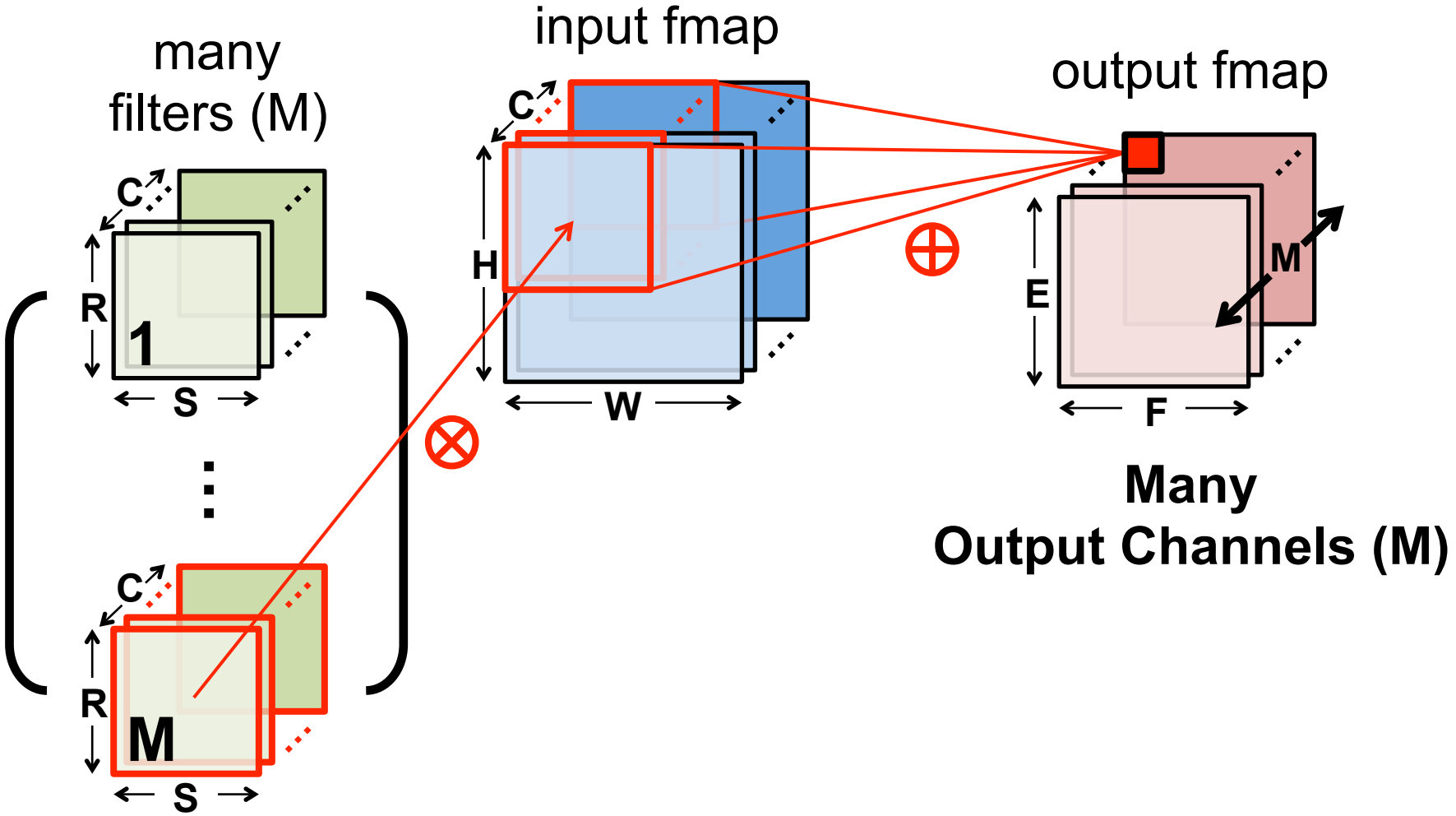
Sliding Window Processing

Convolution (CONV) Layer

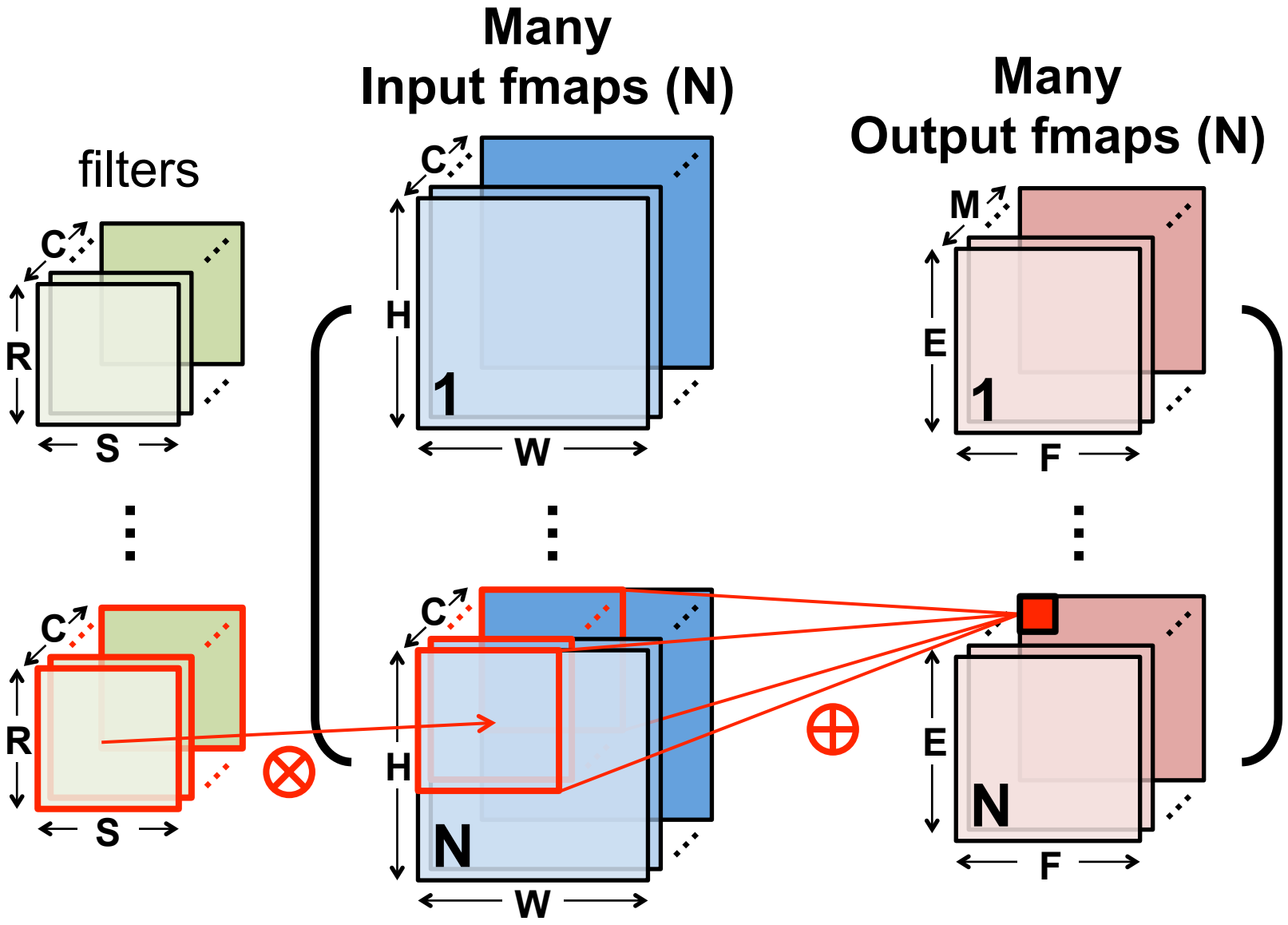


Many Input Channels (C)

Convolution (CONV) Layer



Convolution (CONV) Layer



CONV Layer Implementation

Output fmaps Biases Input fmaps Filter weights

$$\mathbf{O}[n][m][x][y] = \text{Activation}(\mathbf{B}[m] + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} \mathbf{I}[n][k][Ux+i][Uy+j] \times \mathbf{W}[m][k][i][j]),$$

$$0 \le n < N, 0 \le m < M, 0 \le y < E, 0 \le x < F,$$

$$E = (H - R + U)/U, F = (W - S + U)/U.$$

Shape Parameter	Description
N	fmap batch size
M	# of filters / # of output fmap channels
C	# of input fmap/filter channels
H/W	input fmap height/width
R/S	filter height/width
E/F	output fmap height/width
U	convolution stride

CONV Layer Implementation

Naïve 7-layer for-loop implementation:

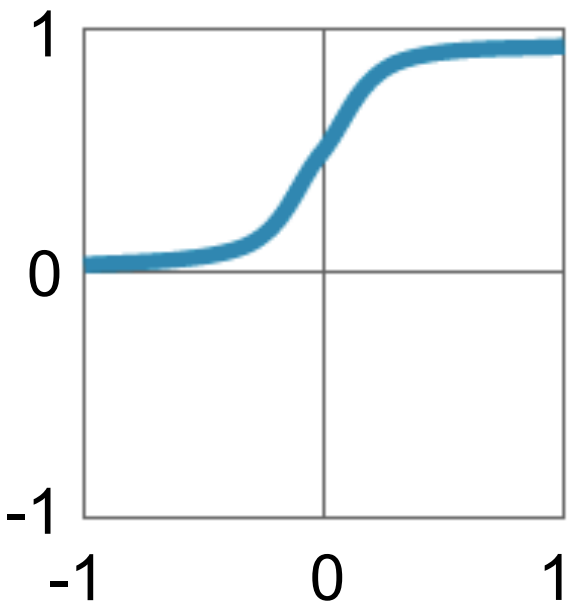
```
for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (x=0; x<F; x++) {
            for (y=0; y<E; y++) {
                O[n][m][x][y] = B[m];
                for (i=0; i<R; i++) {
                    for (j=0; j<S; j++) {
                        for (k=0; k<C; k++) {
                            O[n][m][x][y] += I[n][k][Ux+i][Uy+j] × W[m][k][i][j];
                        }
                    }
                }

                O[n][m][x][y] = Activation(O[n][m][x][y]);
            }
        }
    }
}
```

for each output fmap value

convolve a window and apply activation

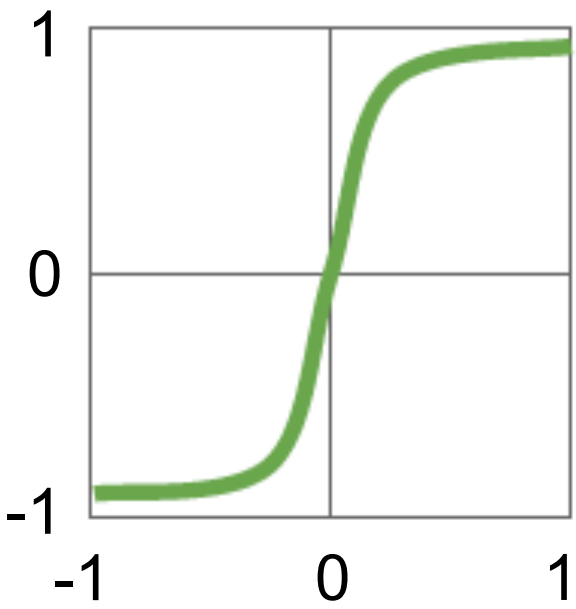
Traditional Activation Functions

Sigmoid



$$y=1/(1+e^{-x})$$

Hyperbolic Tangent

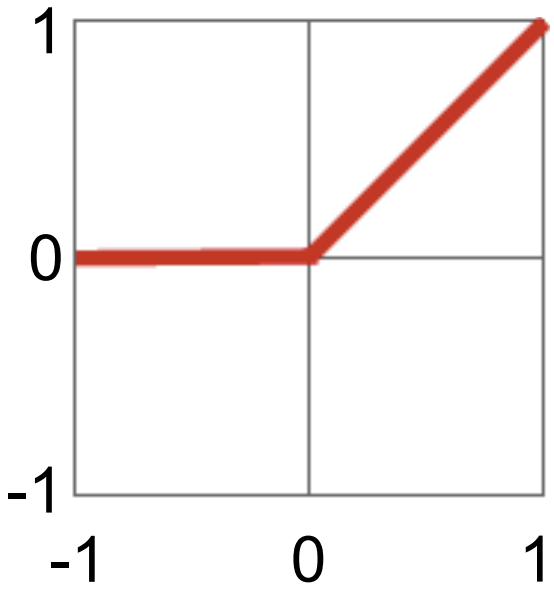


$$y=(e^{x}-e^{-x})/(e^{x}+e^{-x})$$

Image Source: Caffe Tutorial

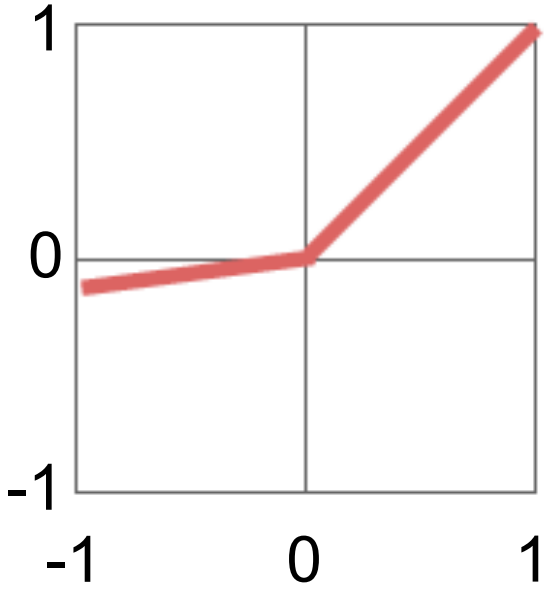
Modern Activation Functions

Rectified Linear Unit (ReLU)



$y = \max(0, x)$

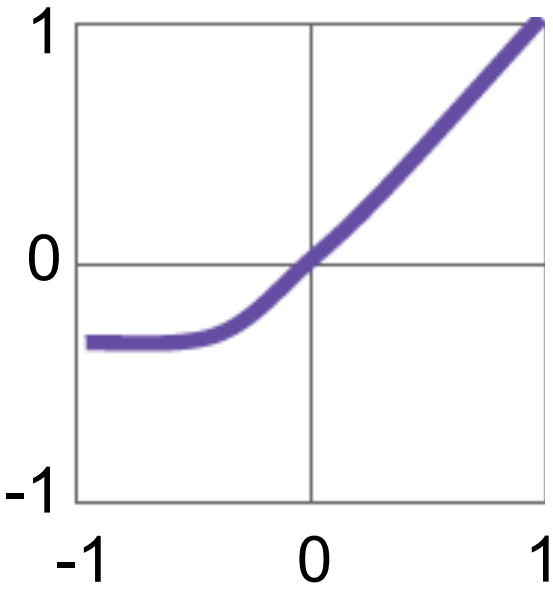
Leaky ReLU



$y = \max(\alpha x, x)$

α = small const. (e.g. 0.1)

Exponential LU

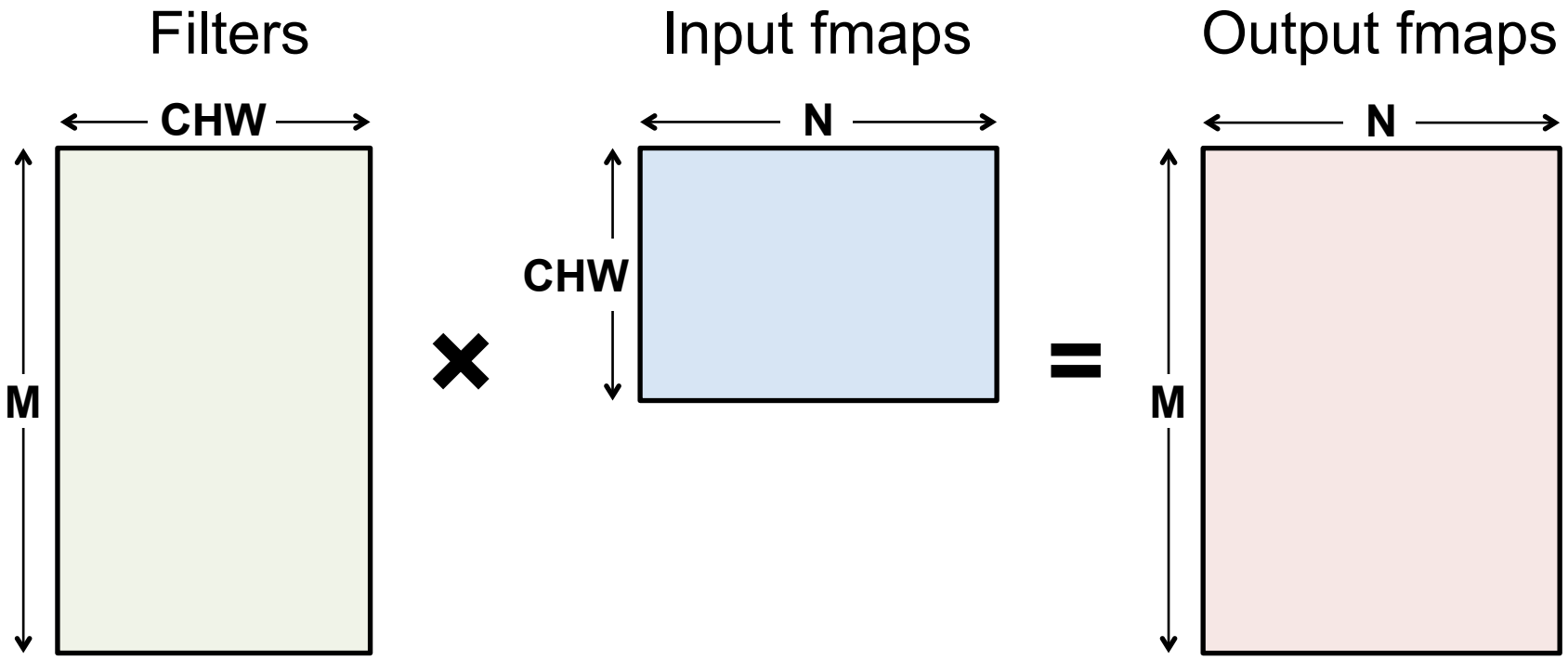


$$y = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

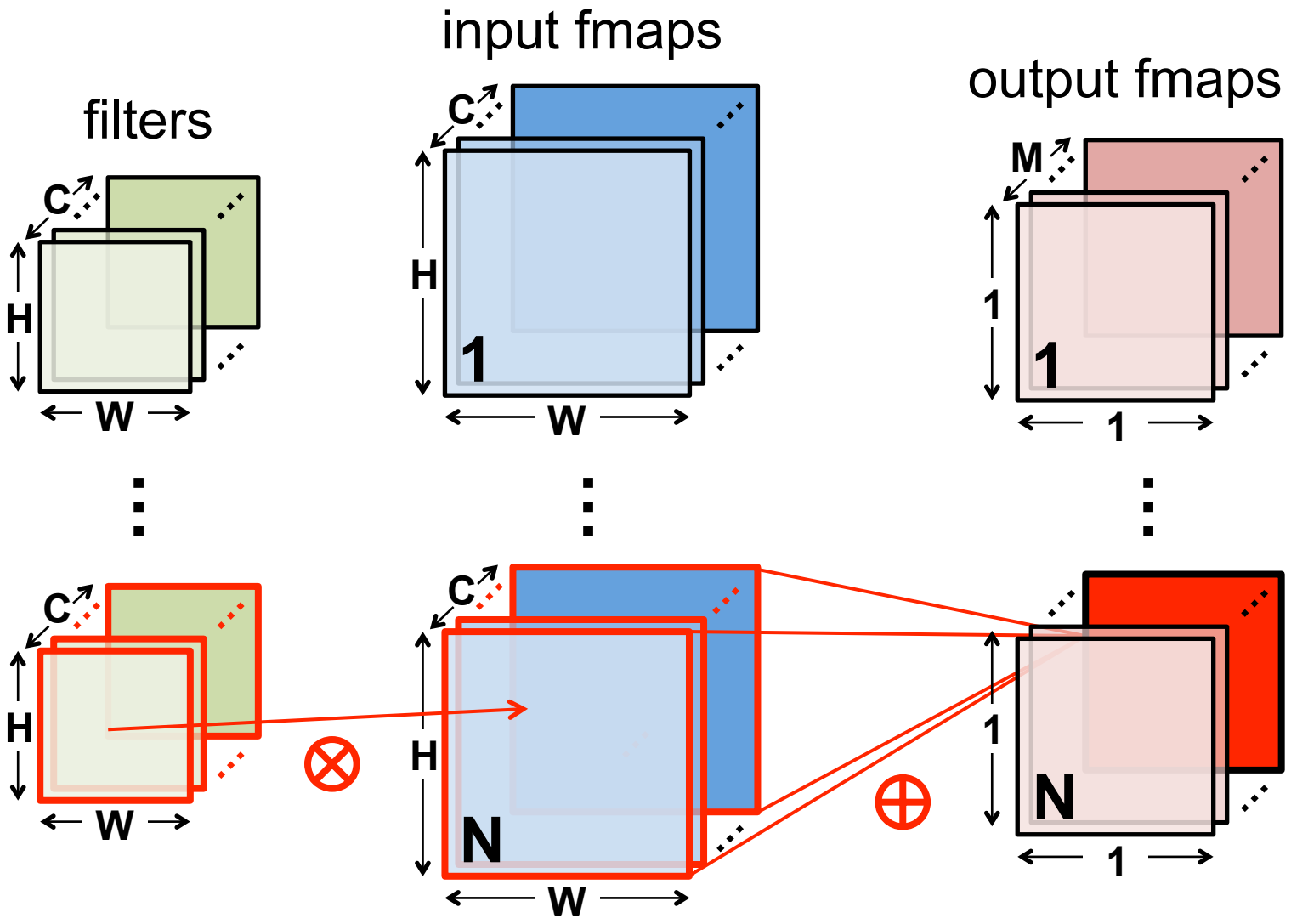
Image Source: Caffe Tutorial

Fully-Connected (FC) Layer

- Height and width of output fmaps are 1 (E = F = 1)
- Filters as large as input fmaps (R = H, S = W)
- Implementation: **Matrix Multiplication**



FC Layer – from CONV Layer POV



Pooling (POOL) Layer

- Reduce resolution of each channel independently
- Increase translation-invariance and noise-resilience
- Overlapping or non-overlapping → depending on stride

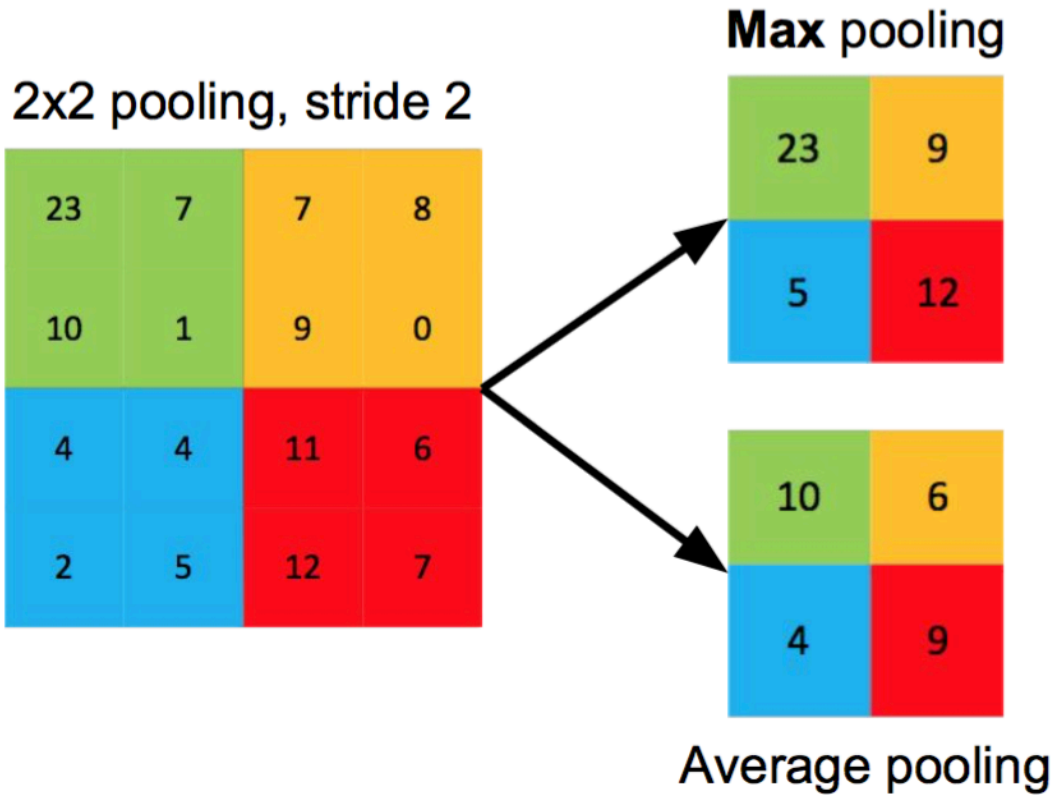


Image Source: Caffe Tutorial

POOL Layer Implementation

Naïve 6-layer for-loop max-pooling implementation:

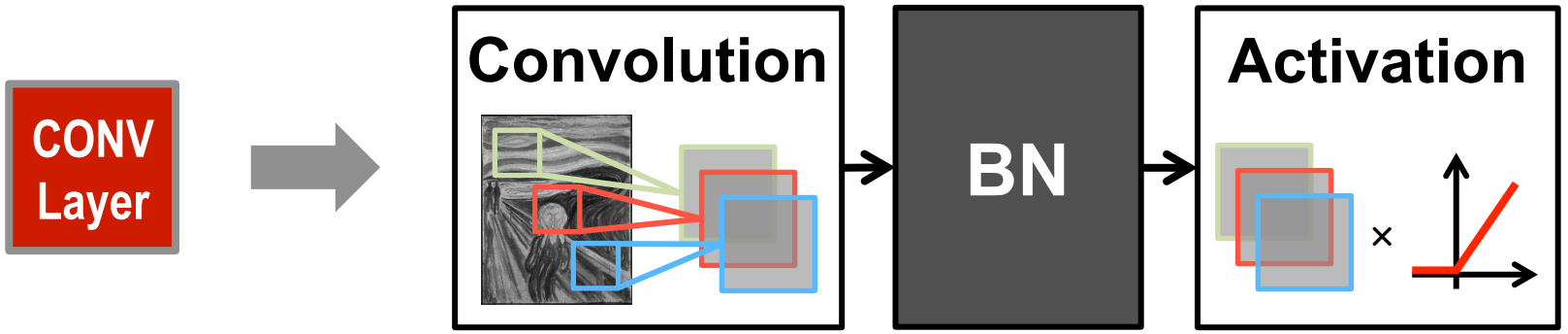
```
for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (x=0; x<F; x++) {
            for (y=0; y<E; y++) {

                max = -Inf;
                for (i=0; i<R; i++) {
                    for (j=0; j<S; j++) {
                        if (I[n][m][Ux+i][Uy+j] > max) {
                            max = I[n][m][Ux+i][Uy+j];
                        }
                    }
                }

                O[n][m][x][y] = max;
            }
        }
    }
}
```

for each pooled value

find the max with in a window

Normalization (NORM) Layer

- **Batch Normalization (BN)**
  - Normalize activations towards mean=0 and std. dev.=1 based on the statistics of the training dataset
  - put **in between** **CONV/FC** and **Activation function**

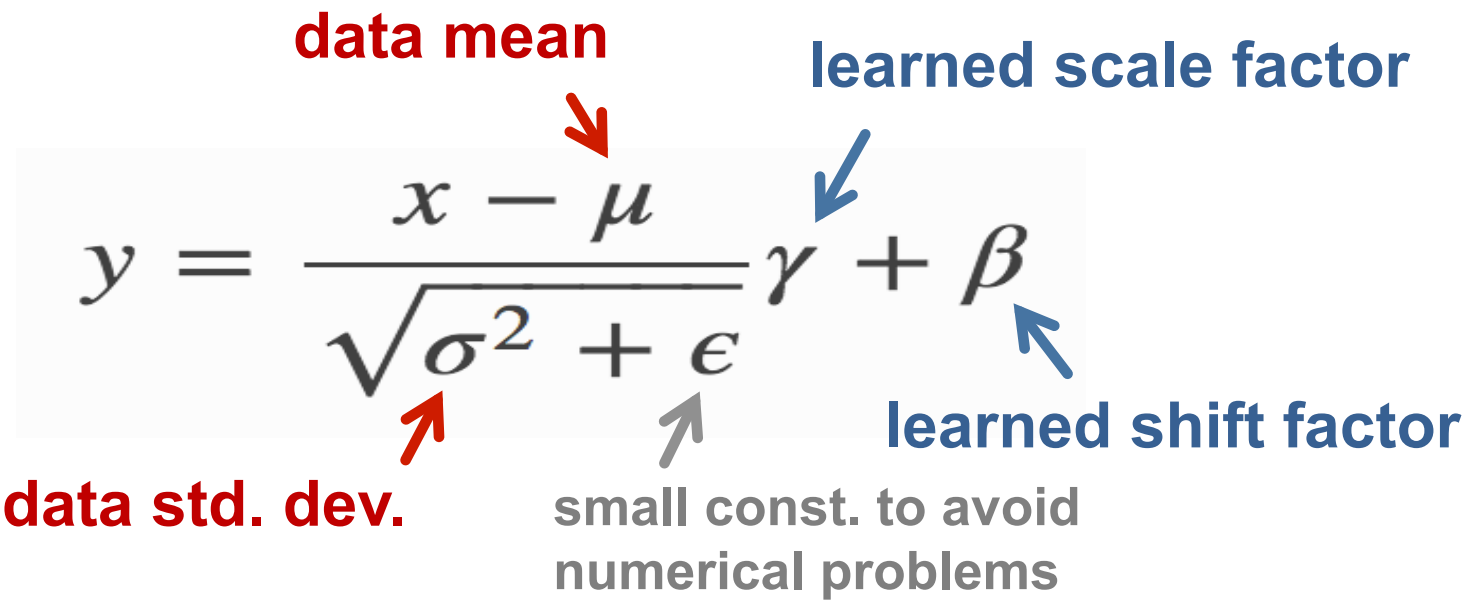


Believed to be key to getting high accuracy and faster training on very deep neural networks.

[Ioffe et al., ICML 2015]

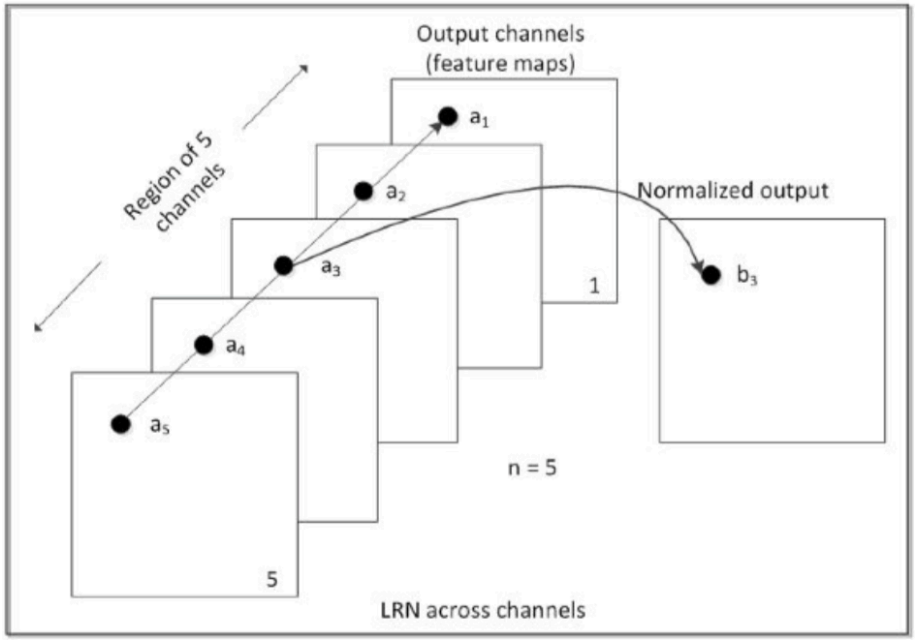
BN Layer Implementation

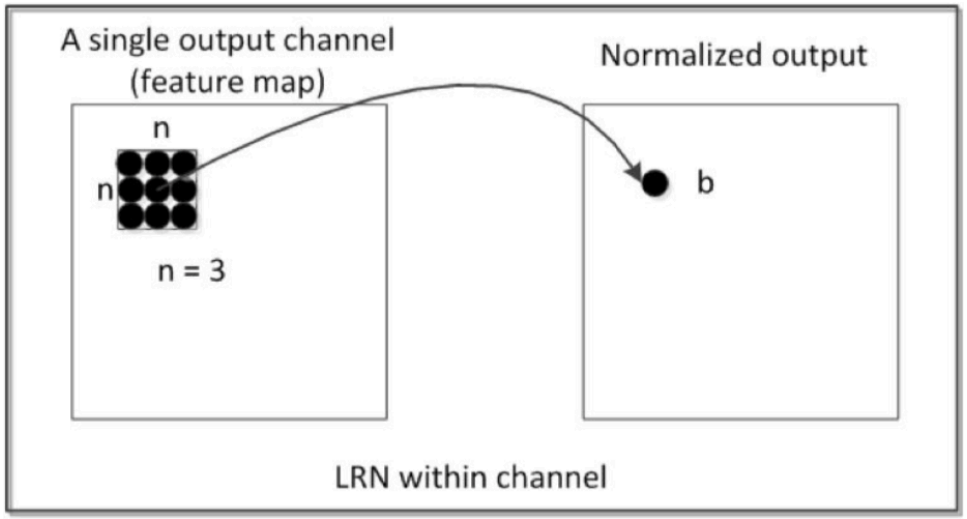
- The normalized value is further scaled and shifted, the parameters of which are learned from training



Normalization (NORM) Layer

- **Local Response Normalization (LRN)**
  - Tries to mimic the inhibition scheme in the brain





Now deprecated!

Image Source: Caffe Tutorial

Relevant Components for Tutorial

- Typical operations that we will discuss:
  - Convolution (CONV)
  - Fully-Connected (FC)
  - Max Pooling
  - ReLU

Survey of DNN Development Resources

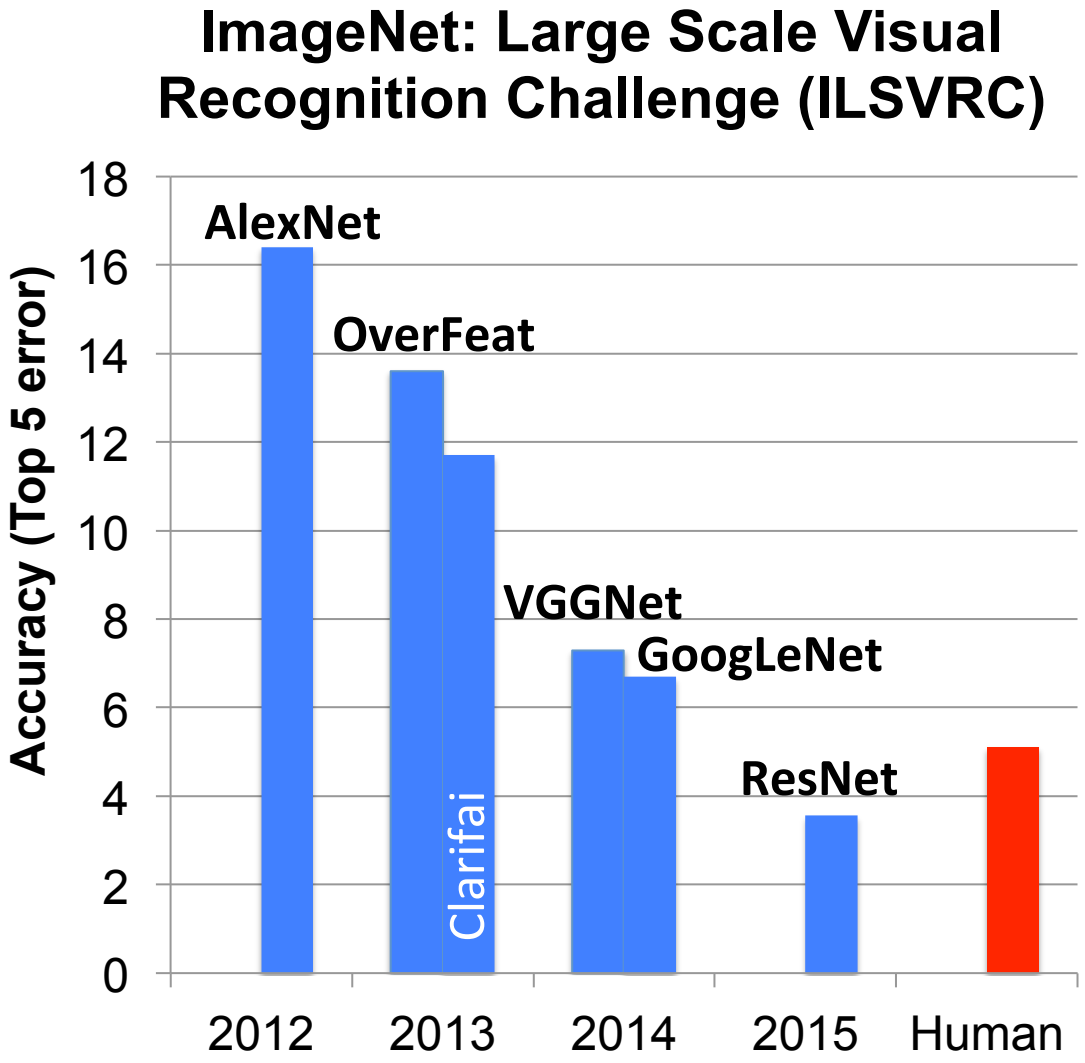
MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

Popular DNNs

- LeNet (1998)
- AlexNet (2012)
- OverFeat (2013)
- VGGNet (2014)
- GoogleNet (2014)
- ResNet (2015)

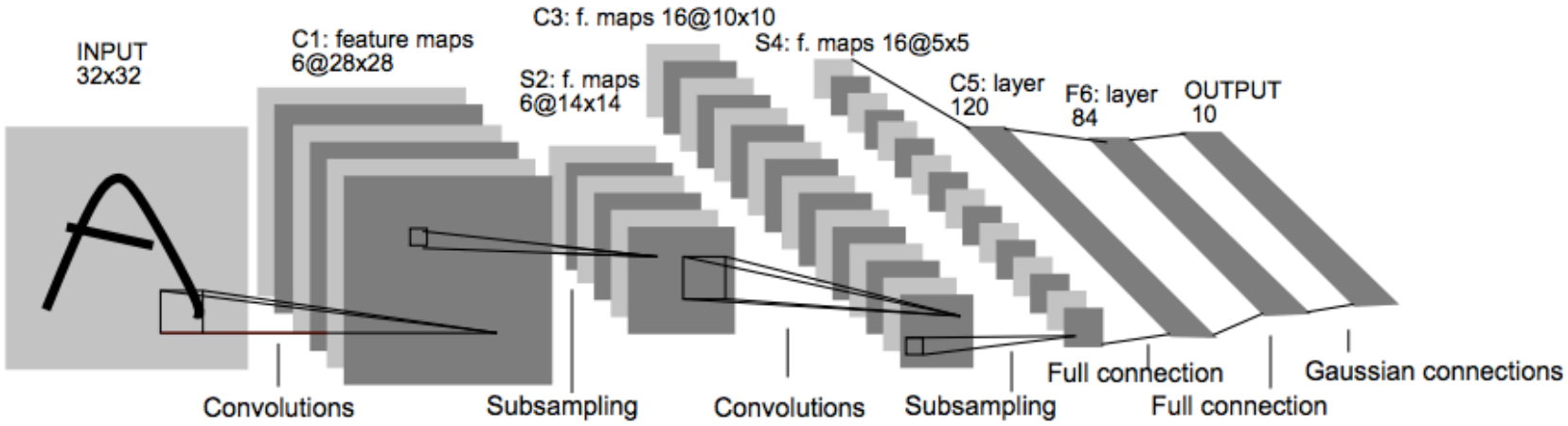


[O. Russakovsky et al., IJCV 2015]

LeNet-5

CONV Layers: 2
Fully Connected Layers: 2
Weights: 431k
MACs: 2.3M

Digit Classification!



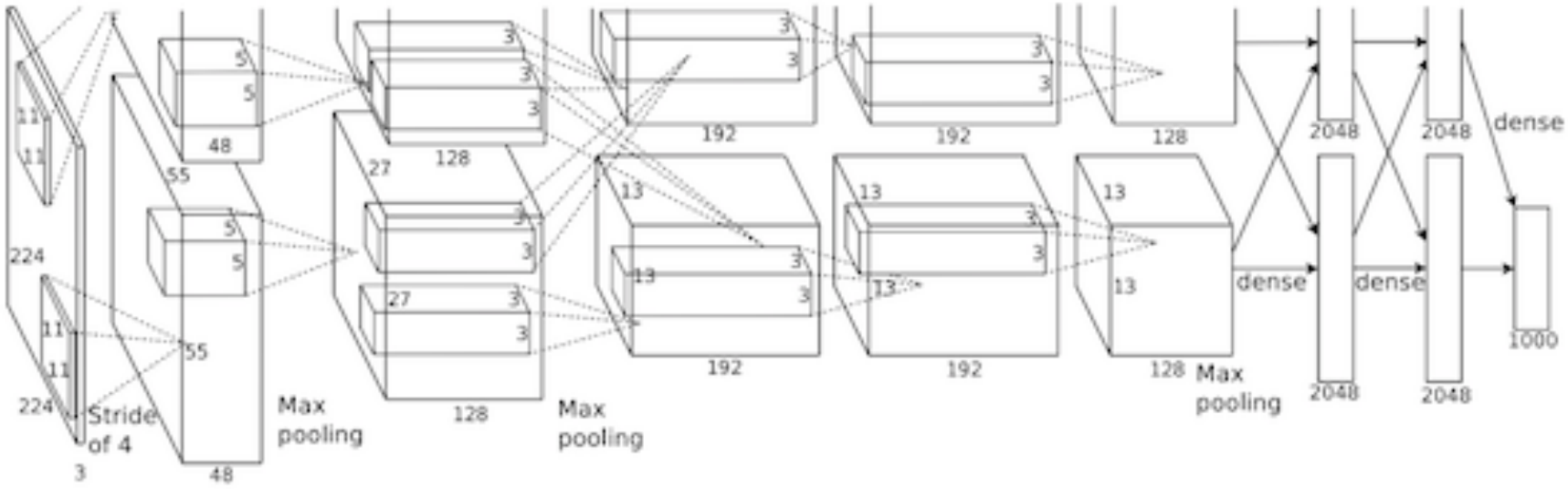
[Y. Lecun et al, Proceedings of the IEEE, 1998]

AlexNet

CONV Layers: 5
Fully Connected Layers: 3
Weights: 61M
MACs: 724M

ILSCVR12 Winner

Uses Local Response Normalization (LRN)



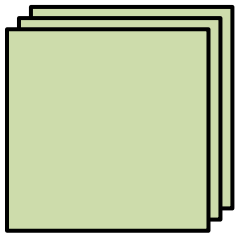
[Krizhevsky et al., NIPS, 2012]

Large Sizes with Varying Shapes

AlexNet Convolutional Layer Configurations

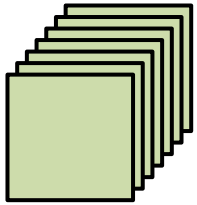
Layer	Filter Size (RxS)	# Filters (M)	# Channels (C)	Stride
1	11x11	96	3	4
2	5x5	256	48	1
3	3x3	384	256	1
4	3x3	384	192	1
5	3x3	256	192	1

Layer 1



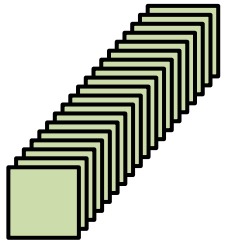
34k Params
105M MACs

Layer 2



307k Params
224M MACs

Layer 3



885k Params
150M MACs

[Krizhevsky et al., NIPS, 2012]

OverFeat (fast model)

CONV Layers: 5
Fully Connected Layers: 3
Weights: 144M
MACs: 5.4G

Layer	1	2	3	4	5	6	7	Output 8
Stage	conv + max	conv + max	conv	conv	conv + max	full	full	full
# channels	96	256	512	1024	1024	3072	4096	1000
Filter size	11x11	5x5	3x3	3x3	3x3	-	-	-
Conv. stride	4x4	1x1	1x1	1x1	1x1	-	-	-
Pooling size	2x2	2x2	-	-	2x2	-	-	-
Pooling stride	2x2	2x2	-	-	2x2	-	-	-
Zero-Padding size	-	-	1x1x1x1	1x1x1x1	1x1x1x1	-	-	-
Spatial input size	231x231	24x24	12x12	12x12	12x12	6x6	1x1	1x1

[Sermanet et al., ArXiv 2013, ICLR 2014]

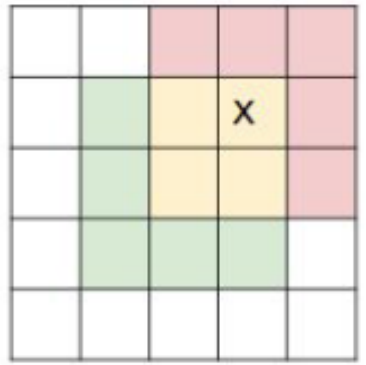
VGG-16

CONV Layers: 16
Fully Connected Layers: 3
Weights: 138M
MACs: 15.5G

Also, 19 layer version

Reduce # of weights

stack 2
3x3 conv



for a 5x5
receptive field

*[figure credit
A. Karpathy]*

More Layers → Deeper!

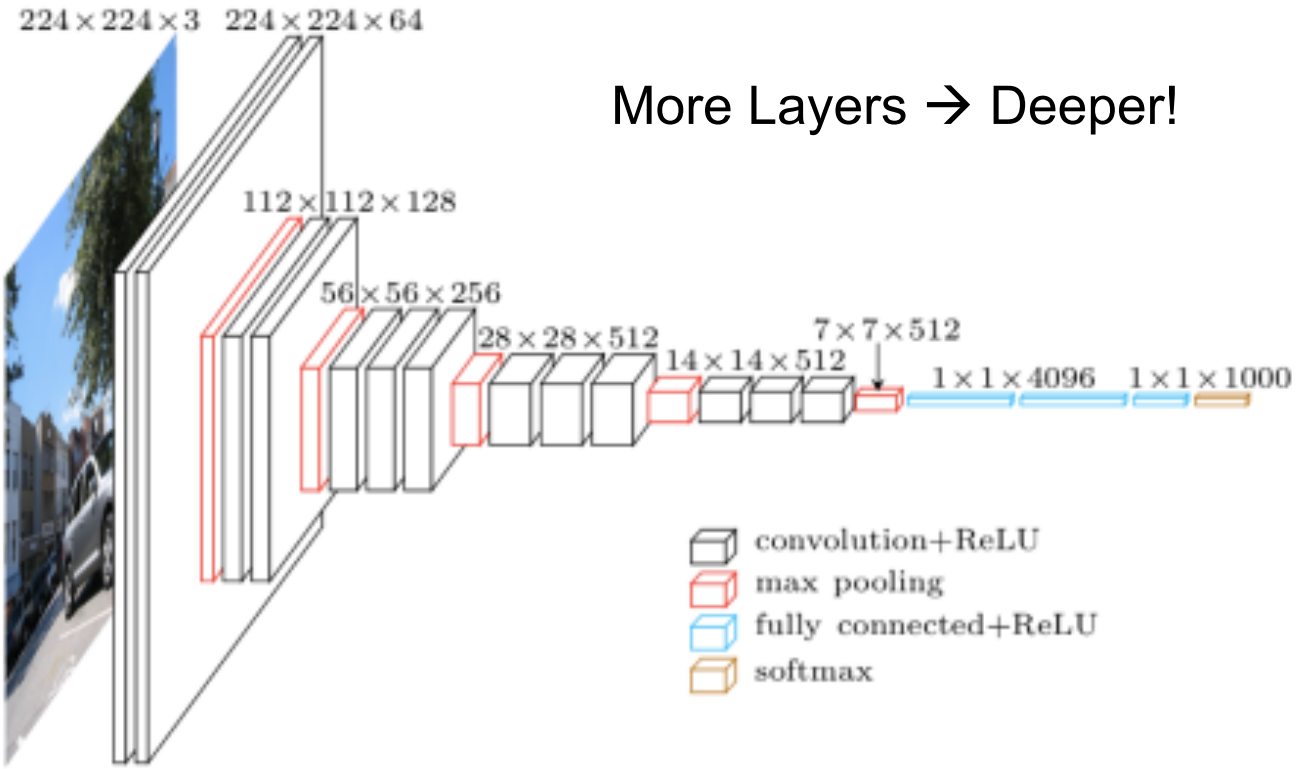


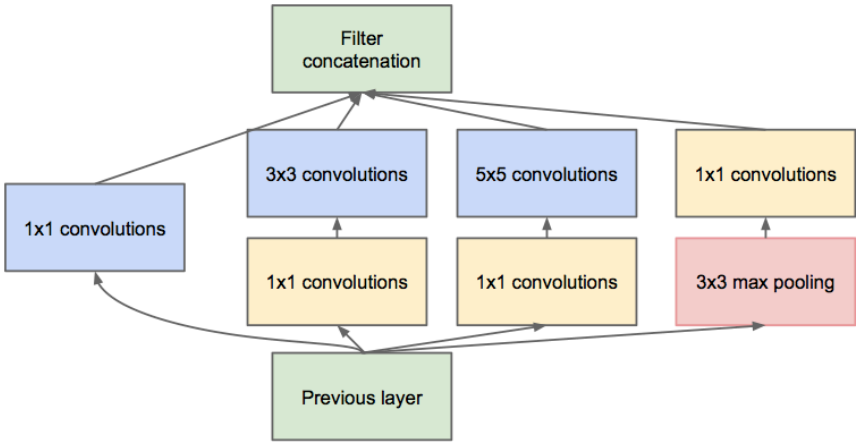
Image Source: http://www.cs.toronto.edu/~frossard/post/vgg16/

[Simonyan et al., ArXiv 2014, ICLR 2015]

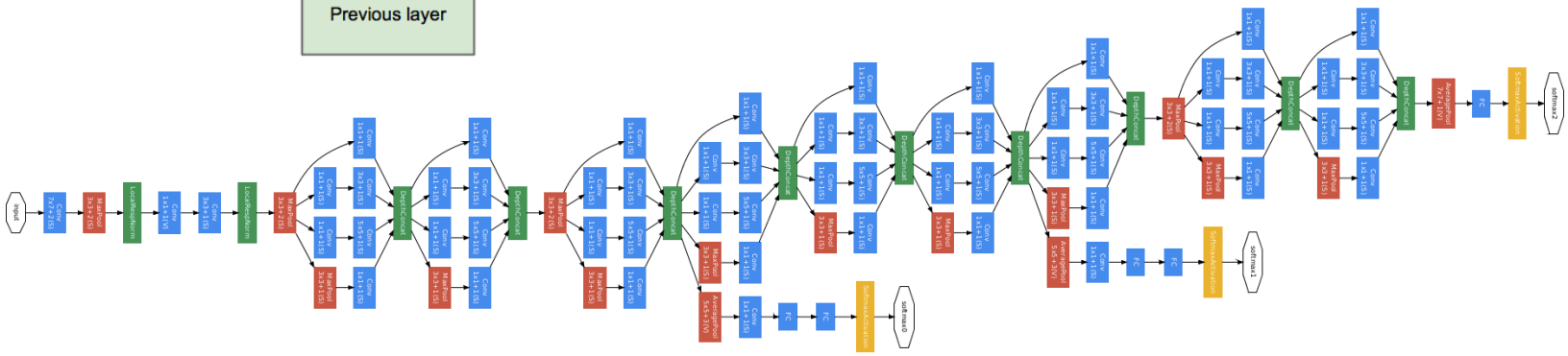
GoogLeNet (v1)

CONV Layers: 21
Fully Connected Layers: 1
Weights: 7.0M
MACs: 1.43G

Also, v2, v3 and v4
ILSVRC14 Winner



Inception module

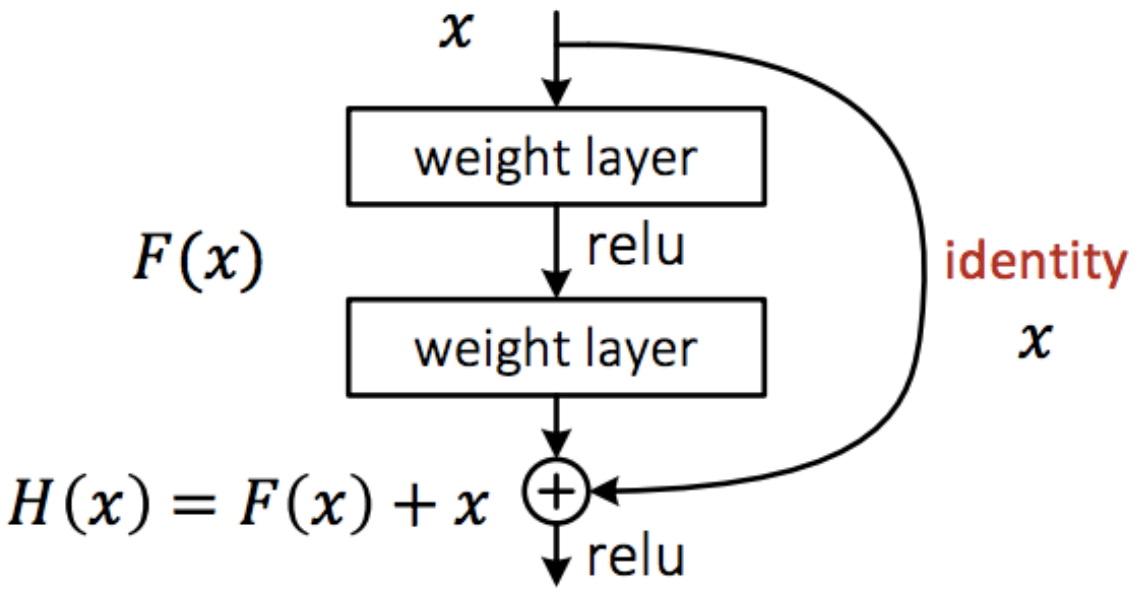


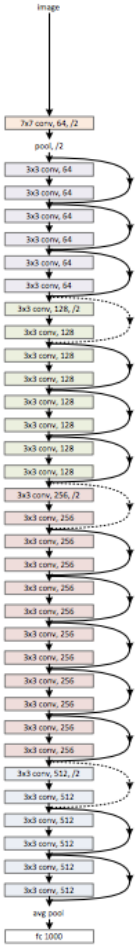
[Szegedy et al., ArXiv 2014, CVPR 2015]

ResNet-50

CONV Layers: 49
Fully Connected Layers: 1
Weights: 25.5M
MACs: 3.9G

Also, 34,152 and 1202 layer versions
ILSVRC15 Winner





[He et al., ArXiv 2015, CVPR 2016]

Revolution of Depth

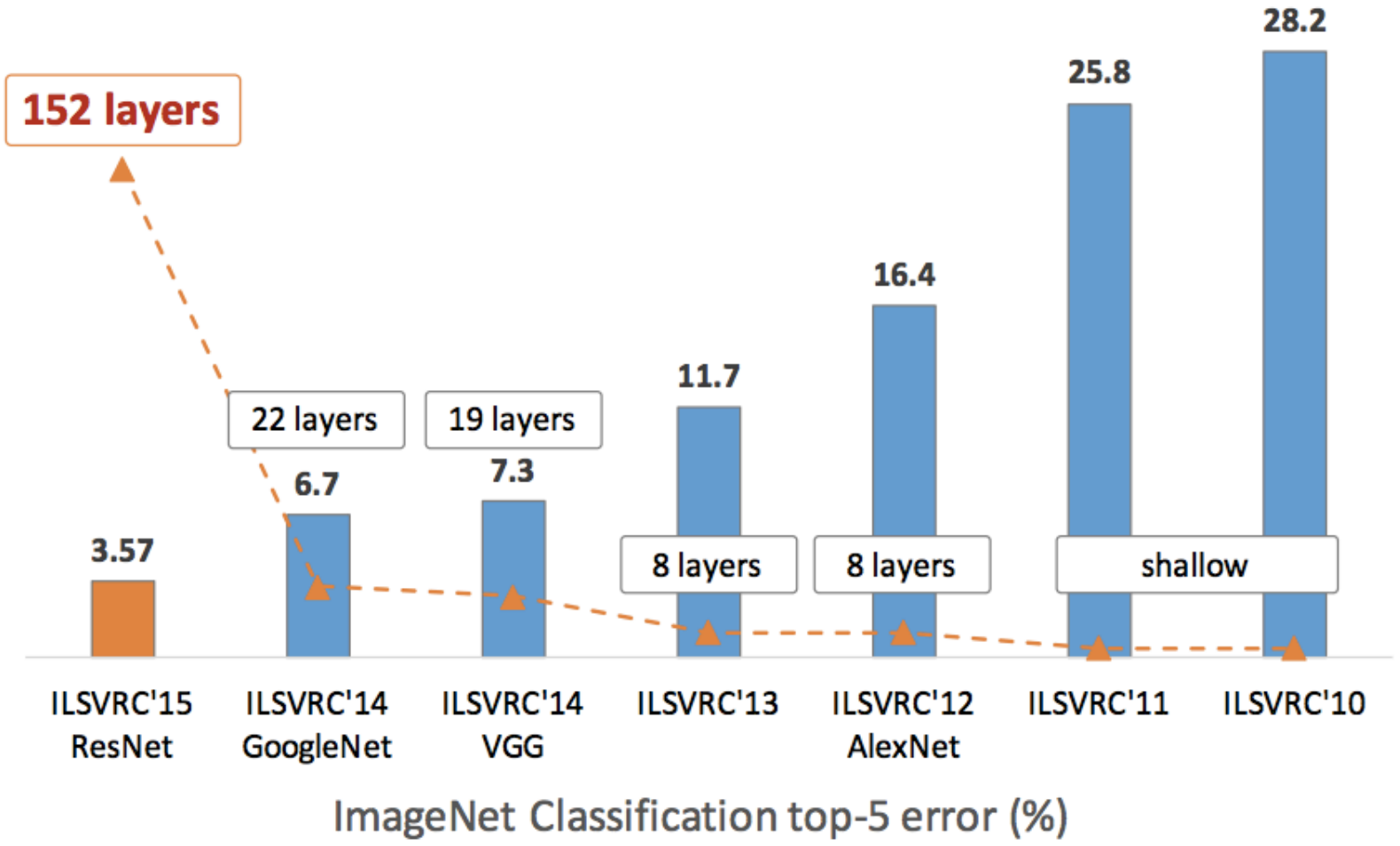


Image Source: http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf

Summary of Popular DNNs

Metrics	LeNet-5	AlexNet	OverFeat (fast)	VGG-16	GoogLeNet (v1)	ResNet-50
Top-5 error	n/a	16.4	14.2	7.4	6.7	5.3
Input Size	28x28	227x227	231x231	224x224	224x224	224x224
# of CONV Layers	**2**	**5**	**5**	**16**	**21**	**49**
Filter Sizes	5	3, 5,11	3, 7	3	1, 3 , 5, 7	1, 3, 7
# of Channels	1, 6	3 - 256	3 - 1024	3 - 512	3 - 1024	3 - 2048
# of Filters	6, 16	96 - 384	96 - 1024	64 - 512	64 - 384	64 - 2048
Stride	1	1, 4	1, 4	1	1, 2	1, 2
# of Weights	26k	2.3M	16M	14.7M	6.0M	23.5M
# of MACs	1.9M	666M	2.67G	15.3G	1.43G	3.86G
# of FC layers	**2**	**3**	**3**	**3**	**1**	**1**
# of Weights	406k	58.6M	130M	124M	1M	2M
# of MACs	405k	58.6M	130M	124M	1M	2M
Total Weights	**431k**	**61M**	**146M**	**138M**	**7M**	**25.5M**
Total MACs	**2.3M**	**724M**	**2.8G**	**15.5G**	**1.43G**	**3.9G**

CONV Layers increasingly important!

Summary of Popular DNNs

- AlexNet
  - First CNN Winner of ILSVRC
  - Uses LRN (deprecated after this)
- VGG-16
  - Goes Deeper (16+ layers)
  - Uses only 3x3 filters (stack for larger filters)
- GoogLeNet (v1)
  - Reduces weights with Inception and only one FC layer
  - Inception: 1x1 and DAG (parallel connections)
  - Batch Normalization
- ResNet
  - Goes Deeper (24+ layers)
  - Shortcut connections

Frameworks

Berkeley / BVLC
(C, C++, Python, MATLAB)

Google
(C++, Python)

U. Montreal
(Python)

Facebook / NYU
(C, C++, Lua)

Also, CNTK, MXNet, etc.
More at: https://developer.nvidia.com/deep-learning-frameworks

Example: Layers in Caffe

Convolution Layer

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
 ...
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
...
```

Non-Linearity

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

Pooling Layer

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2 ...
```

http://caffe.berkeleyvision.org/tutorial/layers.html

Image Classification Datasets

- **Image Classification/Recognition**
  - **Given an entire image → Select 1 of N classes**
  - **No localization (detection)**

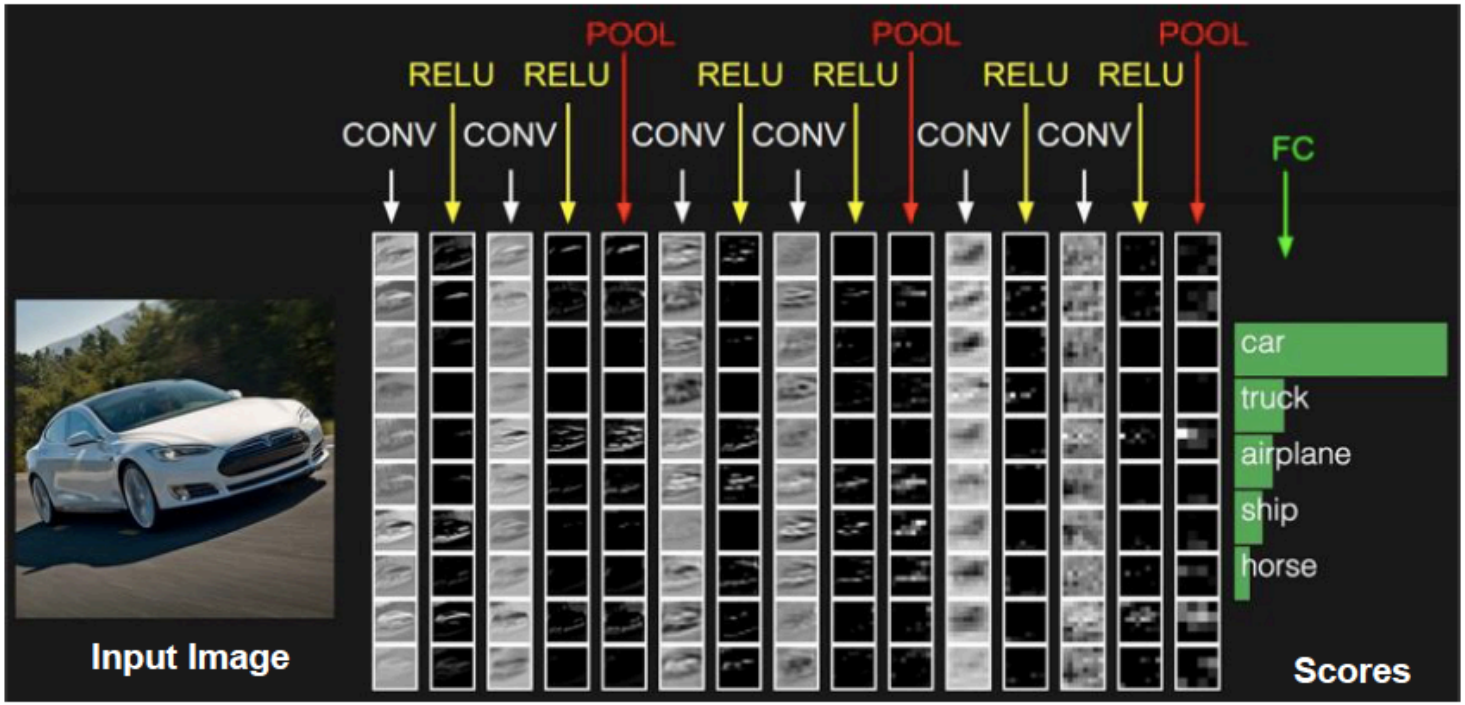


Image Source: Stanford cs231n

Datasets affect difficulty of task

MNIST

Digit Classification
28x28 pixels (B&W)
10 Classes
60,000 Training
10,000 Testing

LeNet in 1998
(0.95% error)

ICML 2013
(0.21% error)

http://yann.lecun.com/exdb/mnist/

CIFAR-10/CIFAR-100

Object Classification
32x32 pixels (color)
10 or 100 Classes
50,000 Training
10,000 Testing

CIFAR-10
RBM+finetuning in 2009
(35.16% error)

ArXiv 2015
(3.47% error)

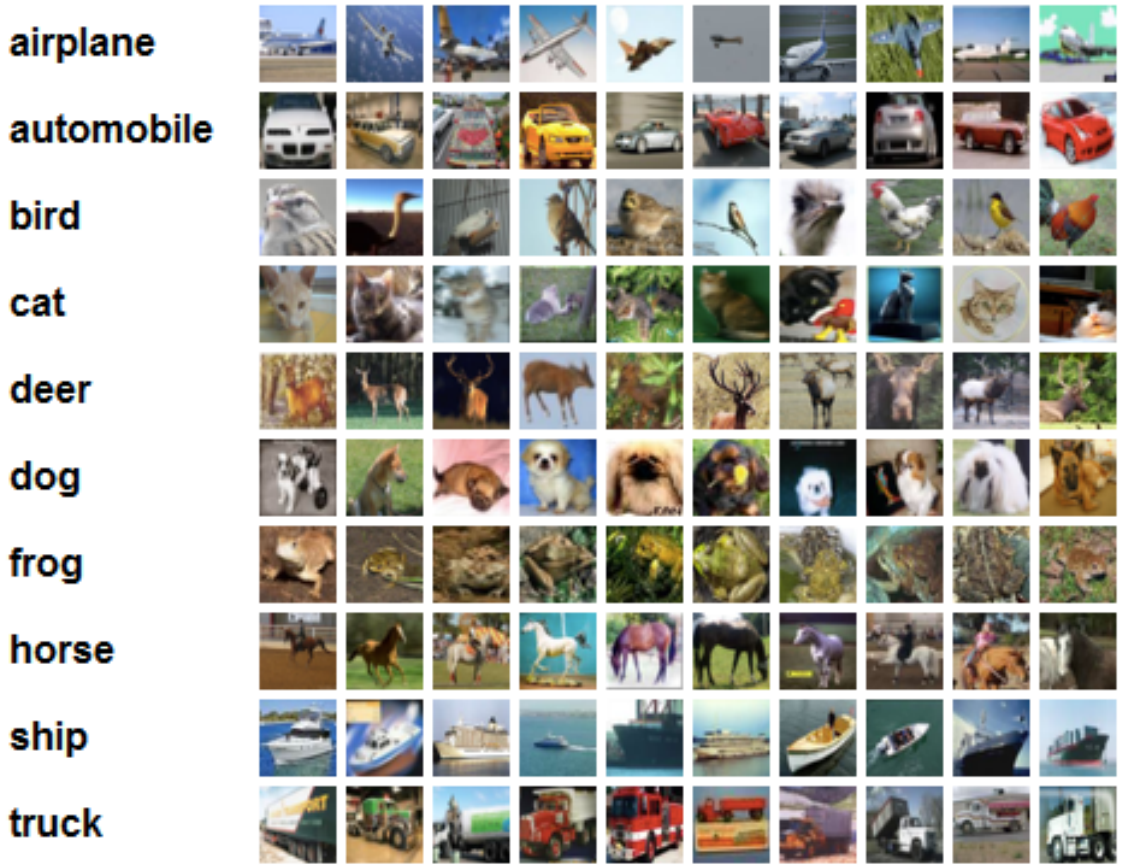


Image Source: http://karpathy.github.io/

Subset of 80 [Tiny Images Dataset](#)(Torrabla)

https://www.cs.toronto.edu/~kriz/cifar.html

IMAGENET

Object Classification

~256x256 pixels (color)

1000 Classes

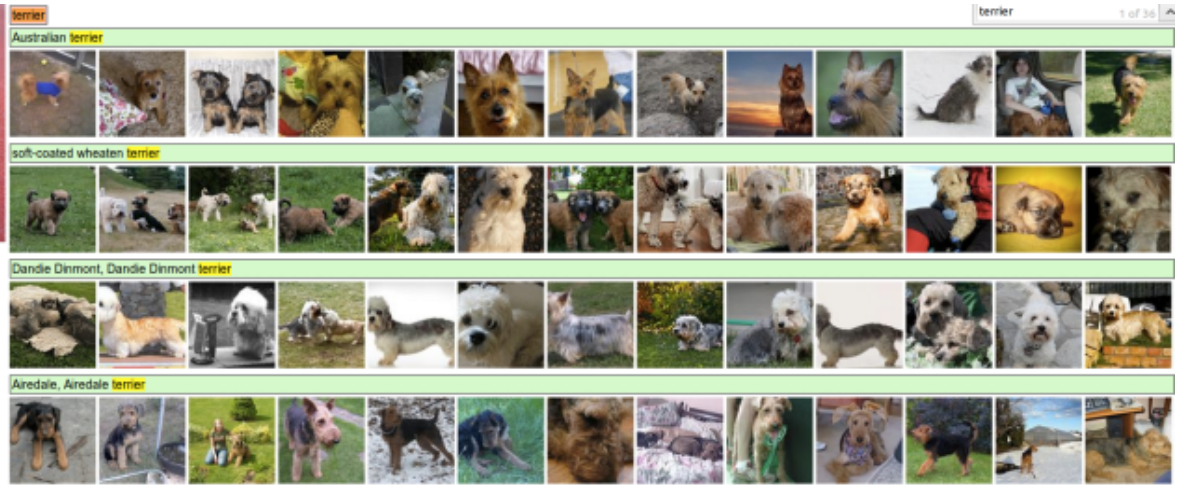
1.3M Training

100,000 Testing (50,000 Validation)

Image Source: http://karpathy.github.io/

http://www.image-net.org/challenges/LSVRC/

IMAGENET



Fine grained Classes
(120 breeds)

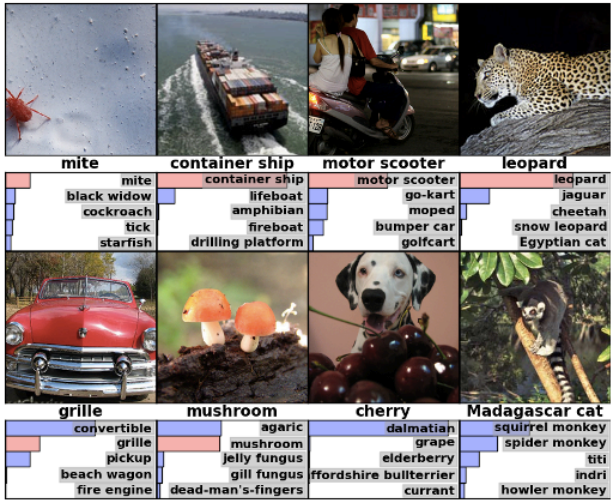
Image Source: http://karpathy.github.io/

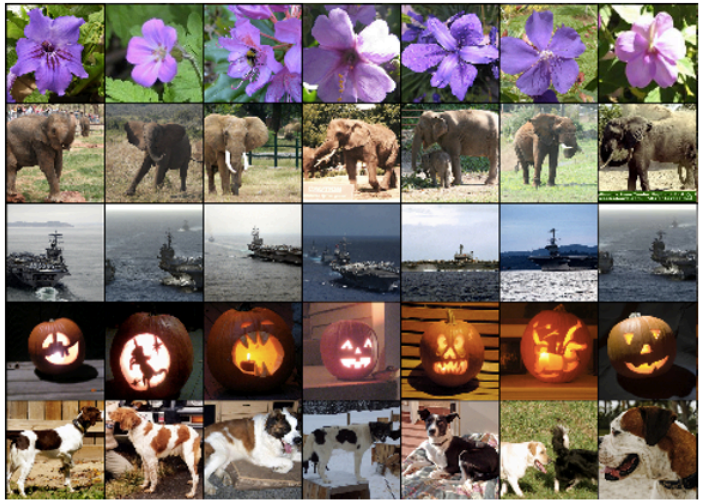
Image Source: Krizhevsky et al., NIPS 2012

Top-5 Error

Winner 2012
(16.42% error)

Winner 2016
(2.99% error)





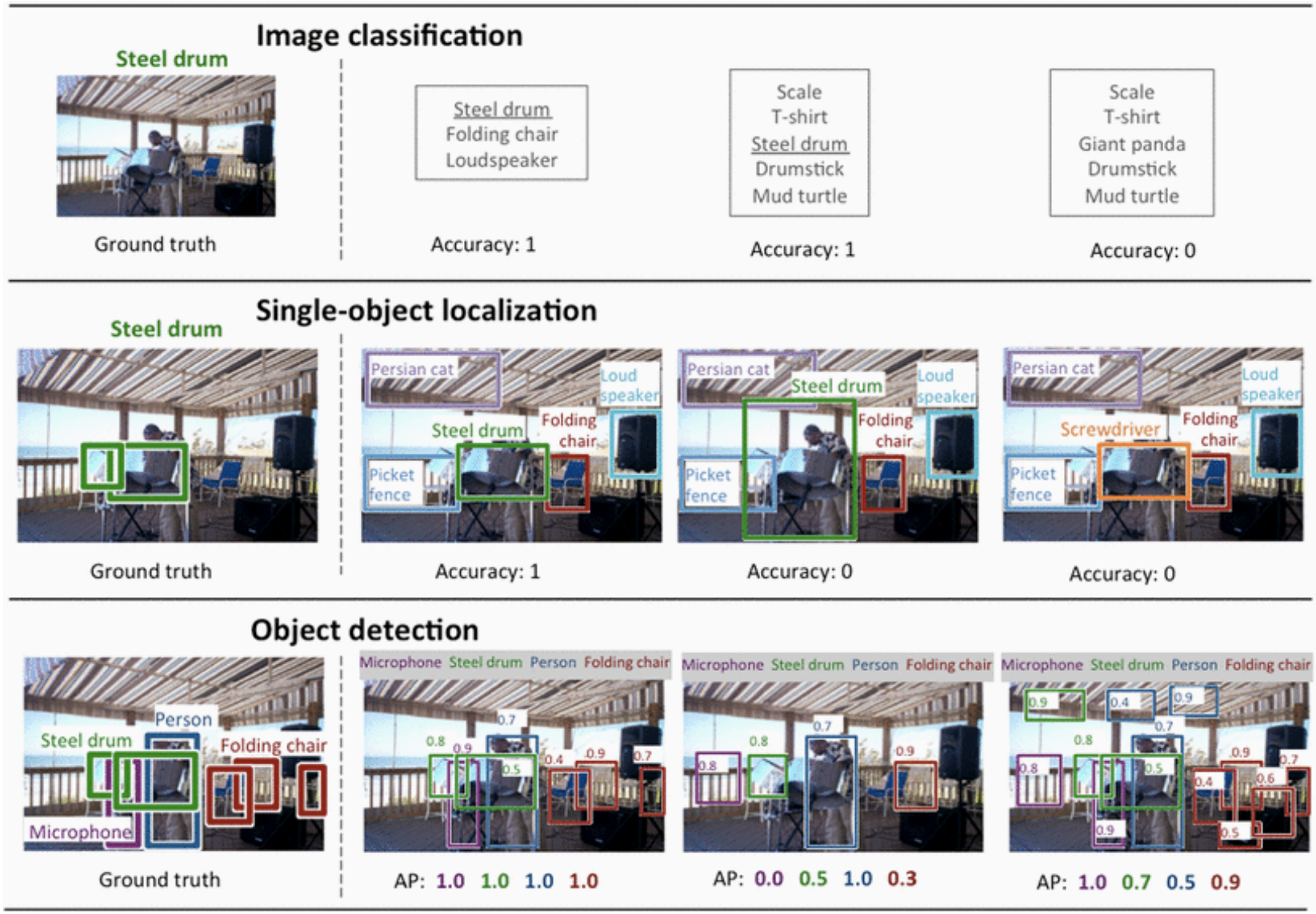
http://www.image-net.org/challenges/LSVRC/

Image Classification Summary

	MNIST	CIFAR-10	CIFAR-100	IMAGENET
Year	1998	2009	2009	2012
Resolution	28x28	32x32	32x32	256x256
Classes	10	10	100	1000
Training	60k	50k	50k	1.3M
Testing	10k	10k	10k	100k
Accuracy	0.21% error (ICML 2013)	3.47% error (arXiv 2015)	24.28% error (arXiv 2015)	2.99% top-5 error (2016 winner)

http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

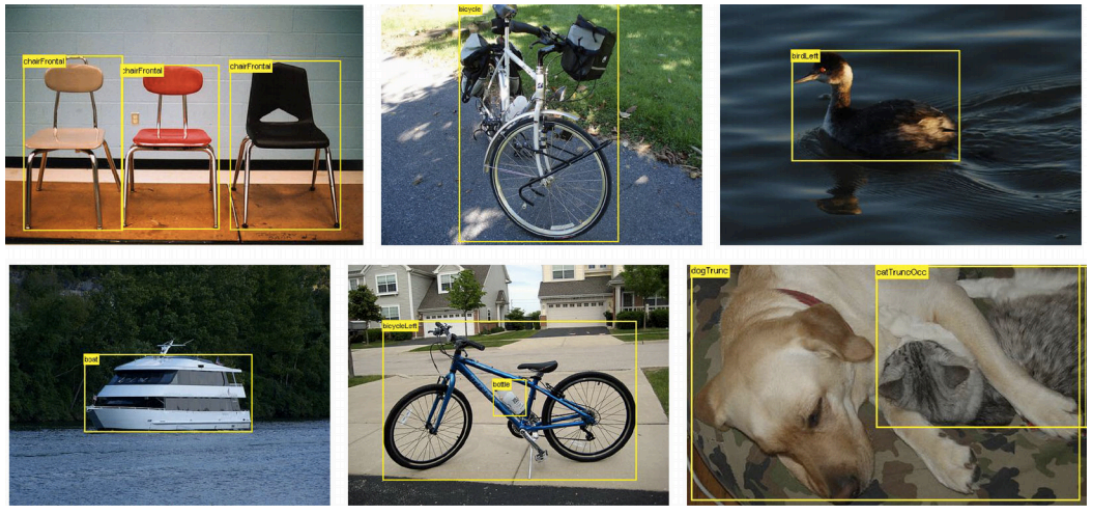
Next Tasks: Localization and Detection



[Russakovsky et al., IJCV, 2015]

Others Popular Datasets

- **Pascal VOC**
  - 11k images
  - Object Detection
  - 20 classes
- **MS COCO**
  - 300k images
  - Detection, Segmentation
  - Recognition in context



http://host.robots.ox.ac.uk/pascal/VOC/

http://mscoco.org/

Recently Introduced Datasets

- **Announced Sept 2016:**
- **Google Open Images (~9M images)**
  - **https://github.com/openimages/dataset**
- **Youtube-8M (8M videos)**
  - **https://research.google.com/youtube8m/**

Survey of DNN Hardware

MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

CPUs Are Targeting Deep Learning

Intel Knights Landing (2016)

- 7 TFLOPS FP32
- 16GB MCDRAM– 400 GB/s
- 245W TDP
- 29 GFLOPS/W (FP32)
- 14nm process

Knights Mill: next gen Xeon Phi “optimized for deep learning”

Intel announced the addition of new vector instructions for deep learning (AVX512-4VNNIW and AVX512-4FMAPS), October 2016

Image Source: Intel, Data Source: Next Platform

GPUs Are Targeting Deep Learning

Nvidia PASCAL GP100 (2016)

- 10/20 TFLOPS FP32/FP16
- 16GB HBM – 750 GB/s
- 300W TDP
- 67 GFLOPS/W (FP16)
- 16nm process
- 160GB/s NV Link

Source: Nvidia

Systems for Deep Learning

Nvidia DGX-1 (2016)

- 170 TFLOPS
- 8× Tesla P100, Dual Xeon
- NVLink Hybrid Cube Mesh
- Optimized DL Software
- 7 TB SSD Cache
- Dual 10GbE, Quad IB 100Gb
- 3RU – 3200W

Source: Nvidia

Cloud Systems for Deep Learning

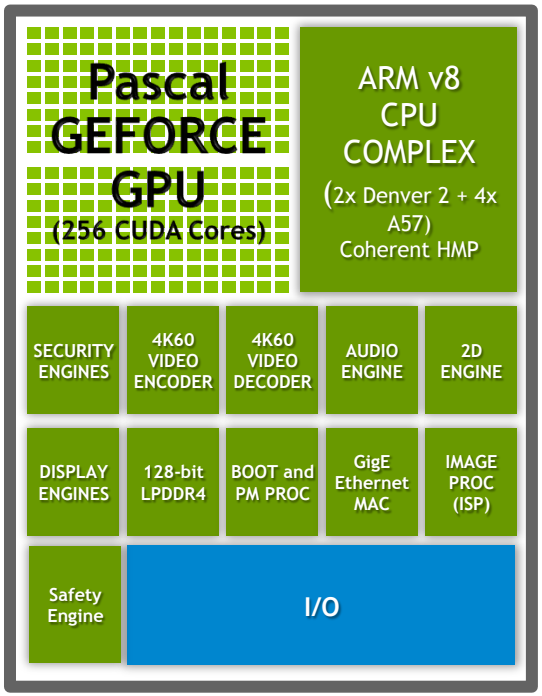
Facebook's Deep Learning Machine

- Open Rack Compliant
- Powered by 8 Tesla M40 GPUs
- 2x Faster Training for Faster Deployment
- 2x Larger Networks for Higher Accuracy

Source: Facebook

SOCs for Deep Learning Inference

Nvidia Tegra - Parker



- GPU: 1.5 TeraFLOPS FP16
- 4GB LPDDR4 @ 25.6 GB/s
- 15 W TDP
  (1W idle, <10W typical)
- 100 GFLOPS/W (FP16)
- 16nm process

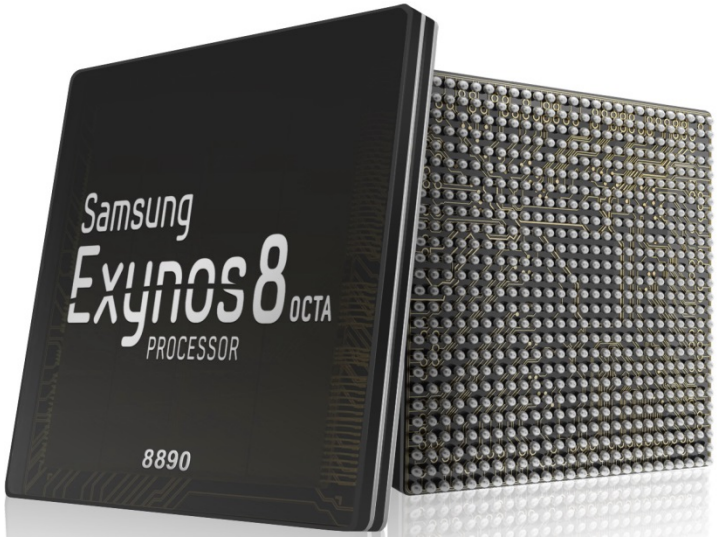
Xavier: next gen Tegra to be an “AI supercomputer”

Source: Nvidia

Mobile SOCs for Deep Learning

Samsung Exynos (ARM Mali)

Exynos 8 Octa 8890



- GPU: 0.26 TFLOPS
- LPDDR4 @ 28.7 GB/s
- 14nm process

Source: Wikipedia

FPGAs for Deep Learning

Intel/Altera Stratix 10

- 10 TFLOPS FP32
- HBM2 integrated
- Up to 1 GHz
- 14nm process
- 80 GFLOPS/W

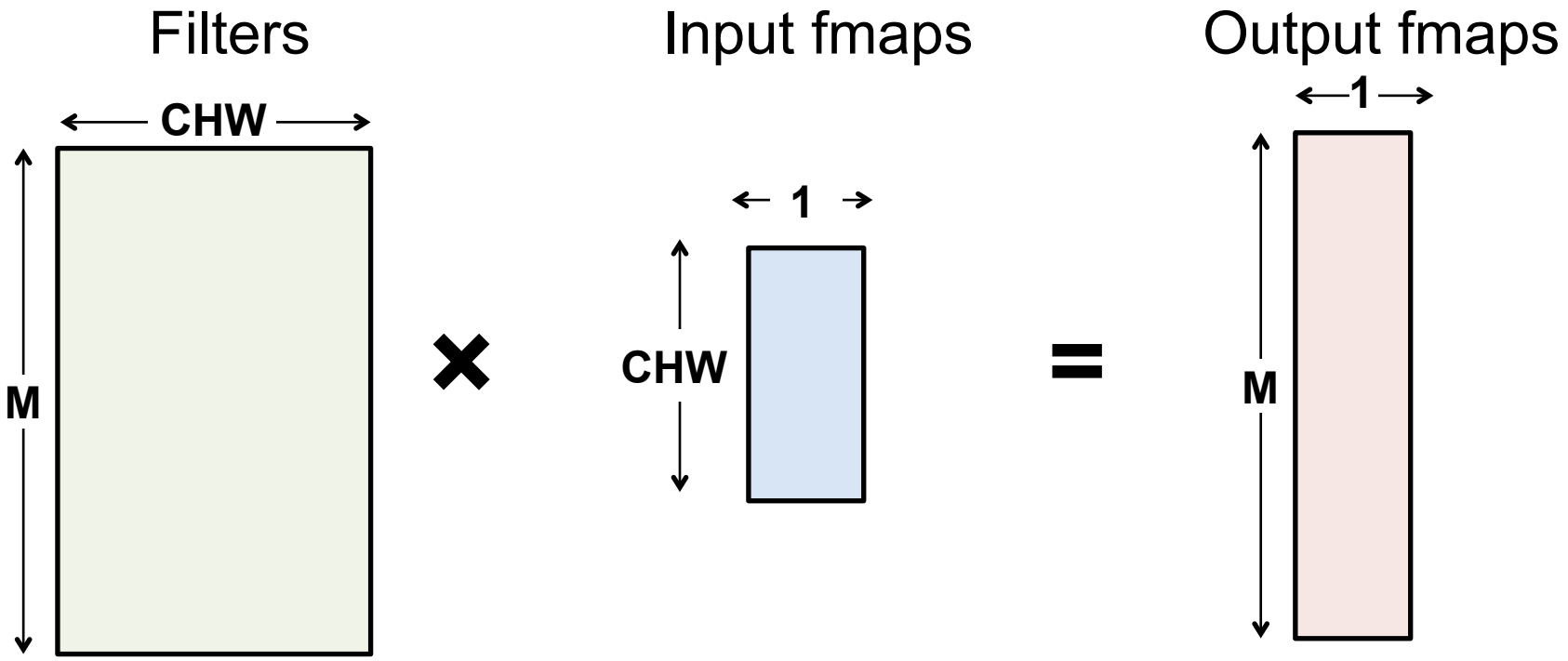
Xilinx Virtex UltraSCALE+

- DSP: up to 21.2 TMACS
- DSP: up to 890 MHz
- Up to 500Mb On-Chip Memory
- 16nm process

Kernel Computation

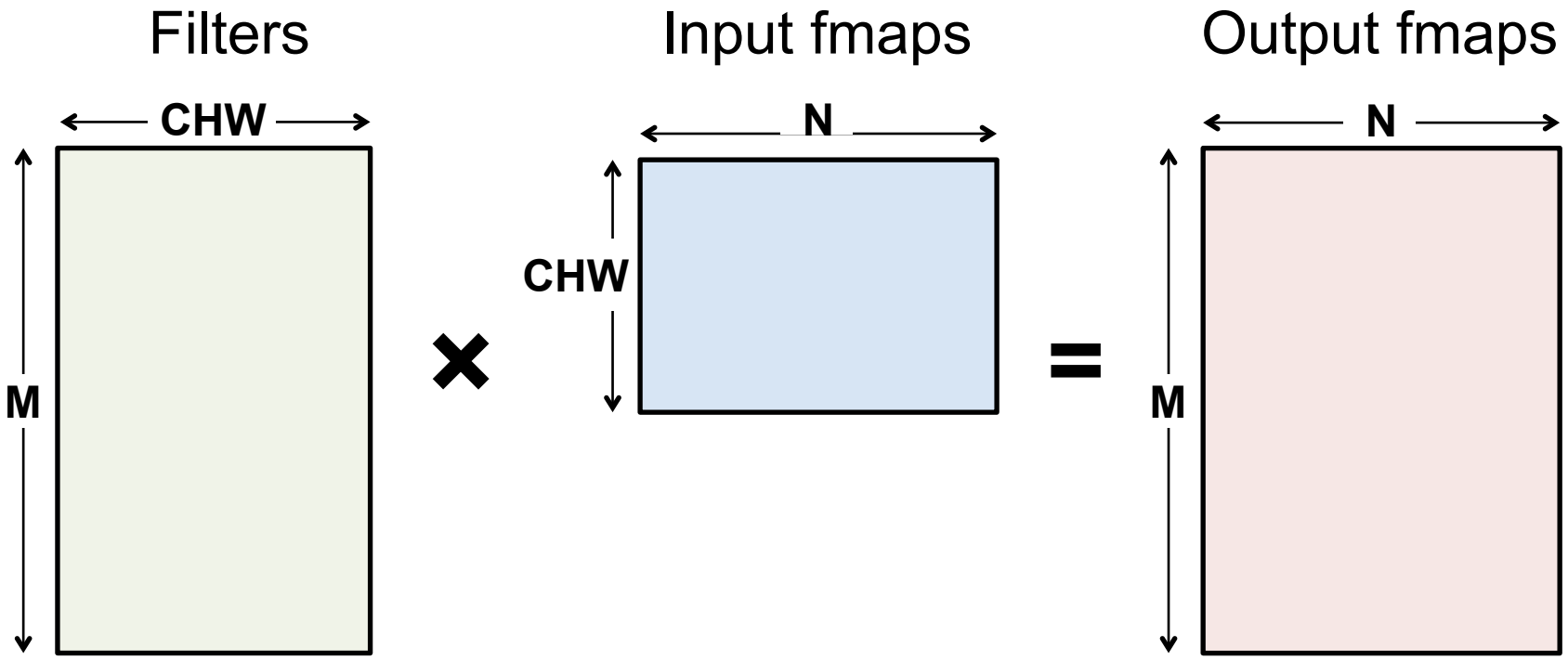
Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Fully-Connected (FC) Layer

- Batching (N) turns operation into a Matrix-Matrix multiply

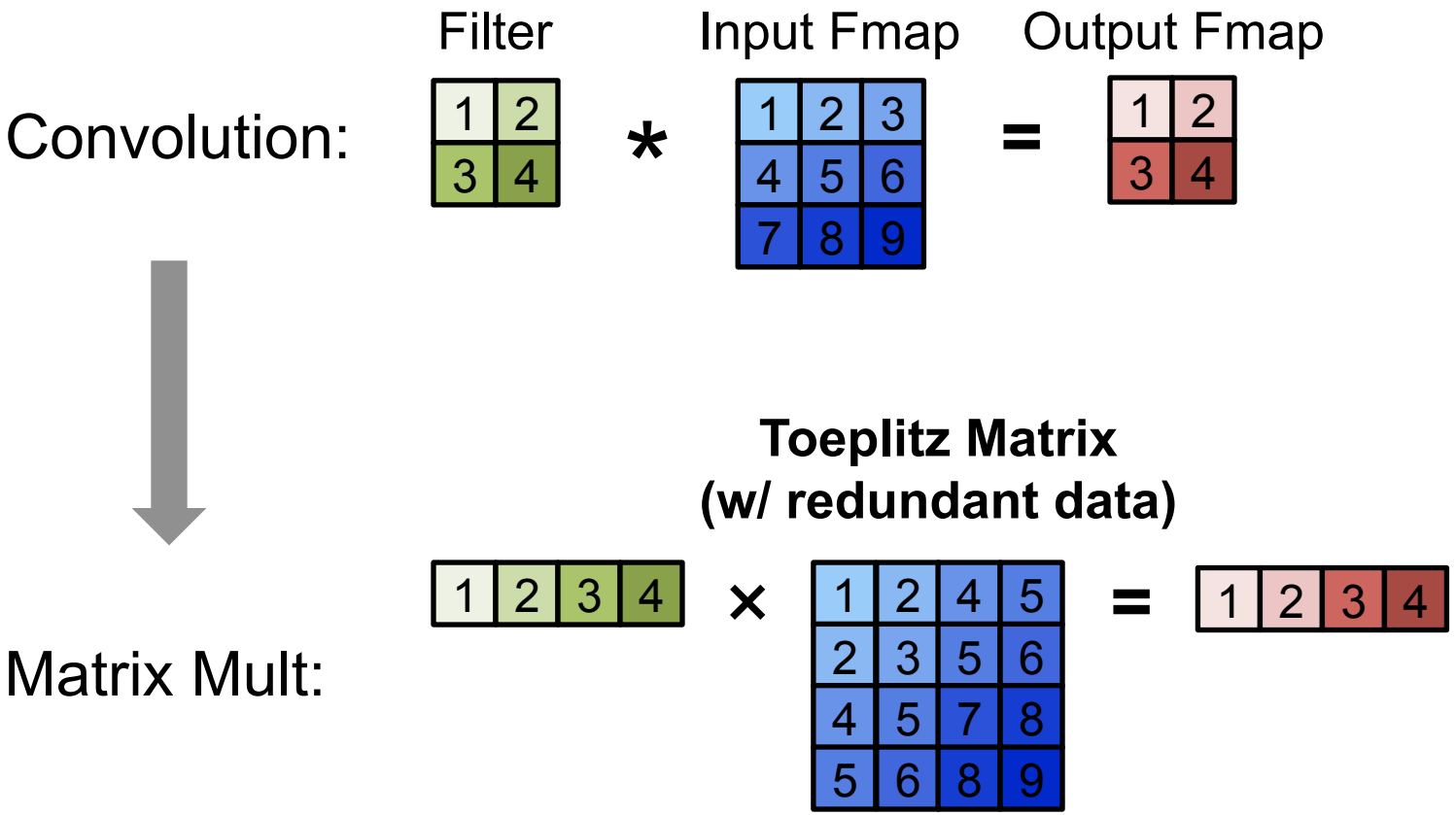


Fully-Connected (FC) Layer

- Implementation: **Matrix Multiplication (GEMM)**
  - **CPU:** OpenBLAS, Intel MKL, etc
  - **GPU:** cuBLAS, cuDNN, etc
- Optimized by tiling to storage hierarchy

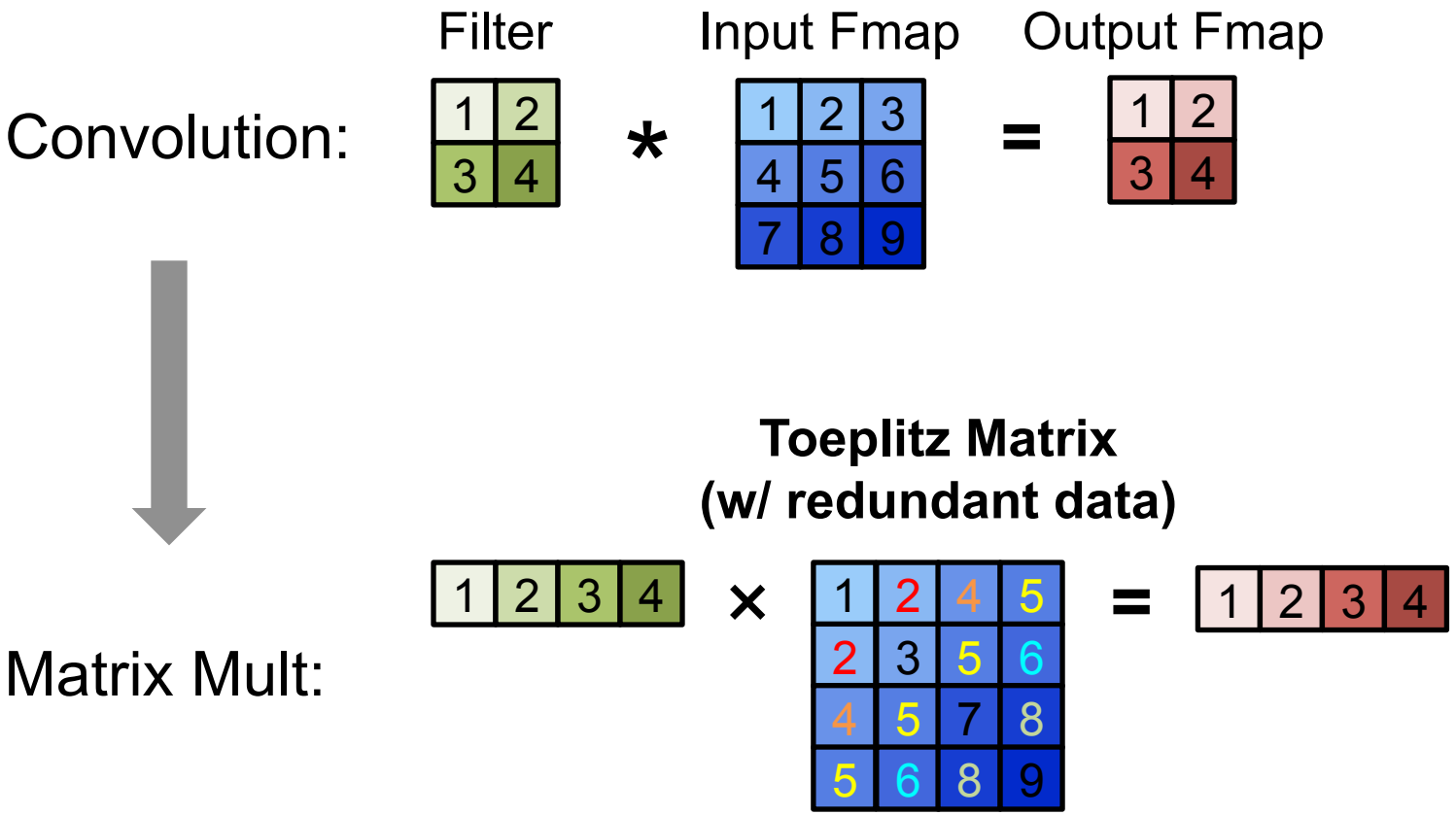
Convolution (CONV) Layer

- Convert to matrix mult. using the **Toeplitz Matrix**



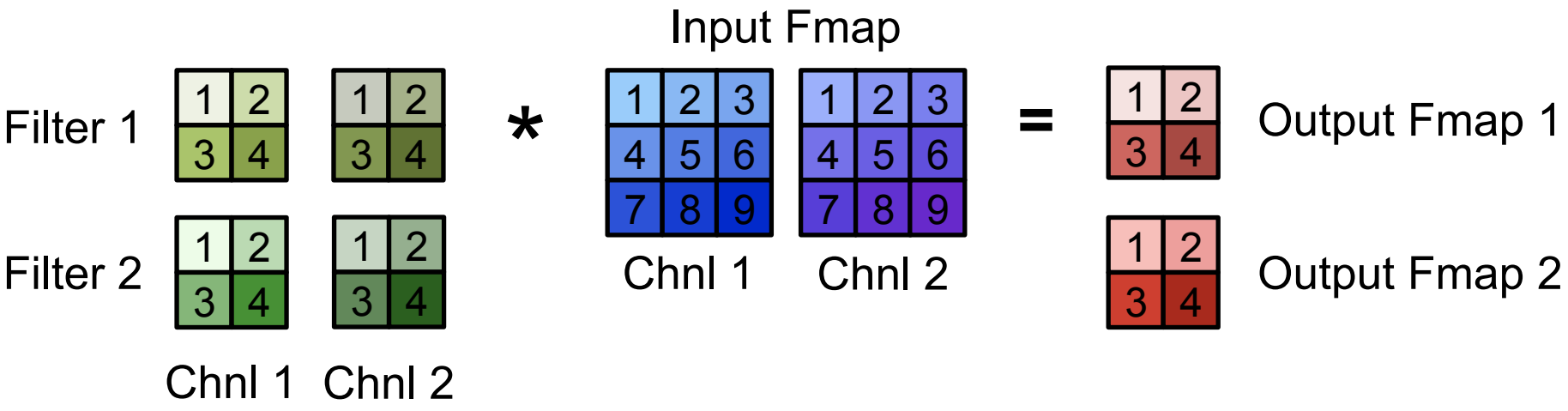
Convolution (CONV) Layer

- Convert to matrix mult. using the **Toeplitz Matrix**



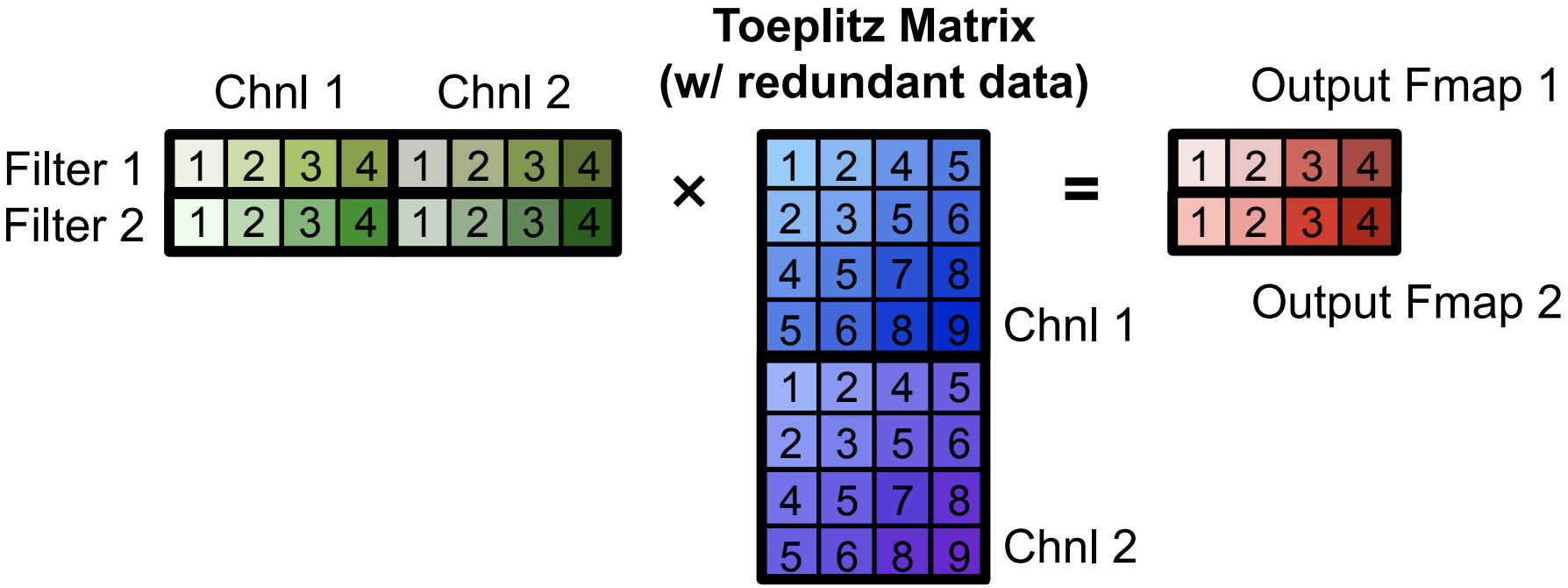
Convolution (CONV) Layer

- Multiple Channels and Filters



Convolution (CONV) Layer

- Multiple Channels and Filters



Computational Transforms

Computation Transformations

- **Goal: Bitwise same result, but reduce number of operations**
- **Focuses mostly on compute**

Gauss's Multiplication Algorithm

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i.$$

4 multiplications + 3 additions

$$k_1 = c \cdot (a + b)$$

$$k_2 = a \cdot (d - c)$$

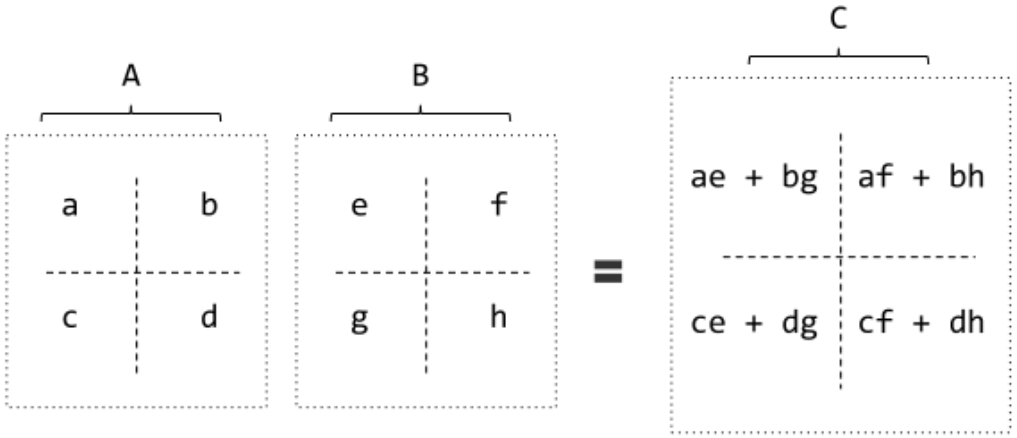
$$k_3 = b \cdot (c + d)$$

$$\text{Real part} = k_1 - k_3$$

$$\text{Imaginary part} = k_1 + k_2.$$

3 multiplications + 5 additions

Strassen



8 multiplications + 4 additions

P1 = a(f – h)
P2 = (a + b)h
P3 = (c + d)e
P4 = d(g – e)

P5 = (a + d)(e + h)
P6 = (b - d)(g + h)
P7 = (a – c)(e + f)

$$AB = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

7 multiplications + 18 additions

7 multiplications + 13 additions (for constant B matrix – weights)

[Cong et al., ICANN, 2014]

Strassen

- Reduce the complexity of matrix multiplication from $\Theta(N^3)$ to $\Theta(N^{2.807})$ by reducing multiplications
- Comes at the price of reduced numerical stability and requires significantly more memory

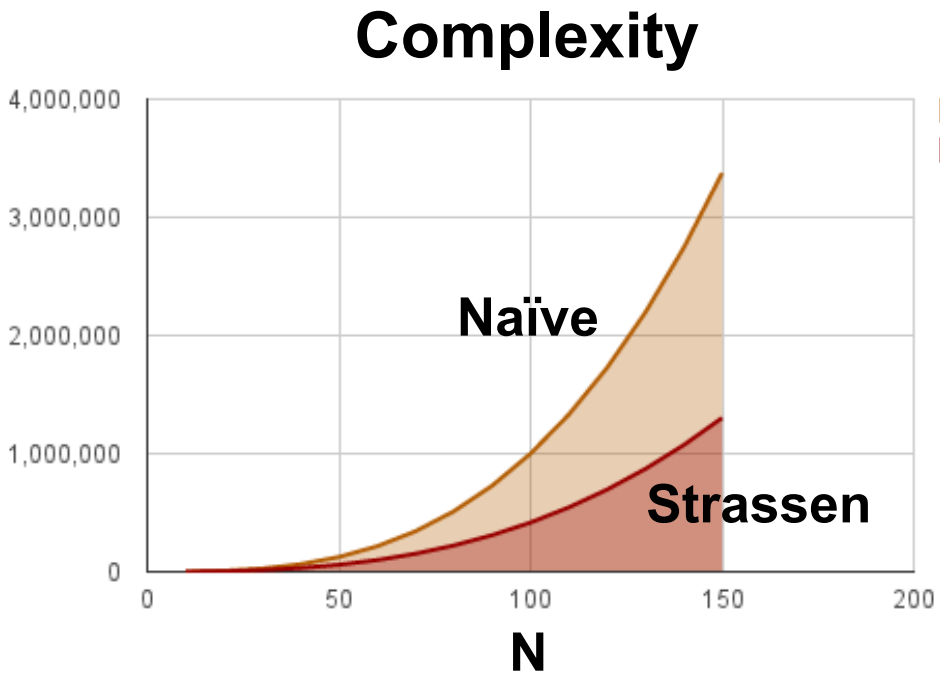


Image Source: http://www.stoimen.com/blog/2012/11/26/computer-algorithms-strassens-matrix-multiplication/

Winograd 1D – F(2,3)

- Targeting convolutions instead of matrix multiply
- Notation: F(size of output, filter size)

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix}$$

input filter

6 multiplications + 4 additions

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

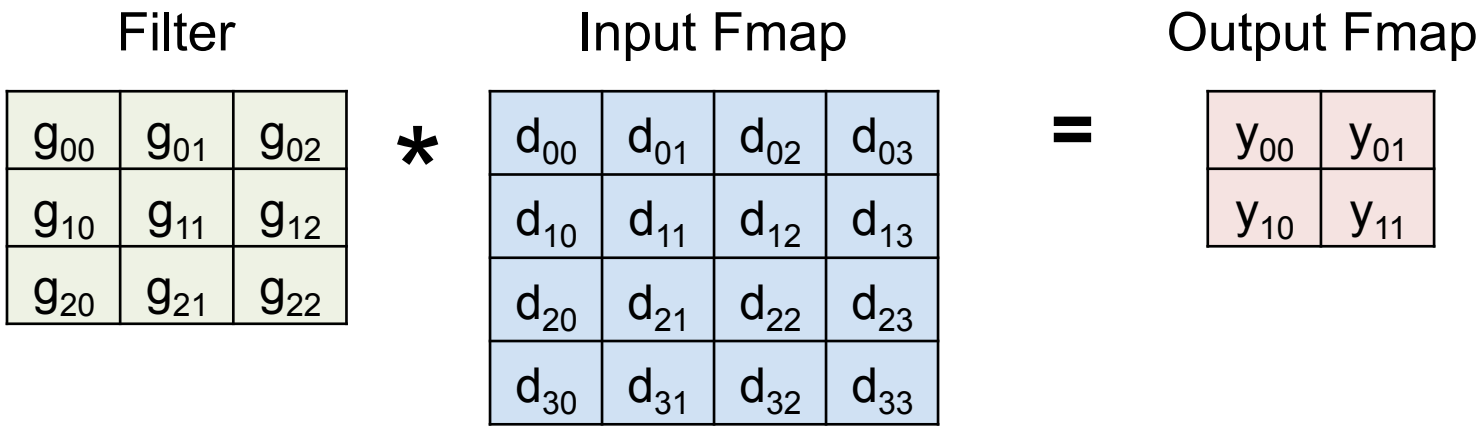
4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

[Lavin et al., ArXiv 2015]

Winograd 2D - F(2x2, 3x3)

- 1D Winograd is nested to make 2D Winograd

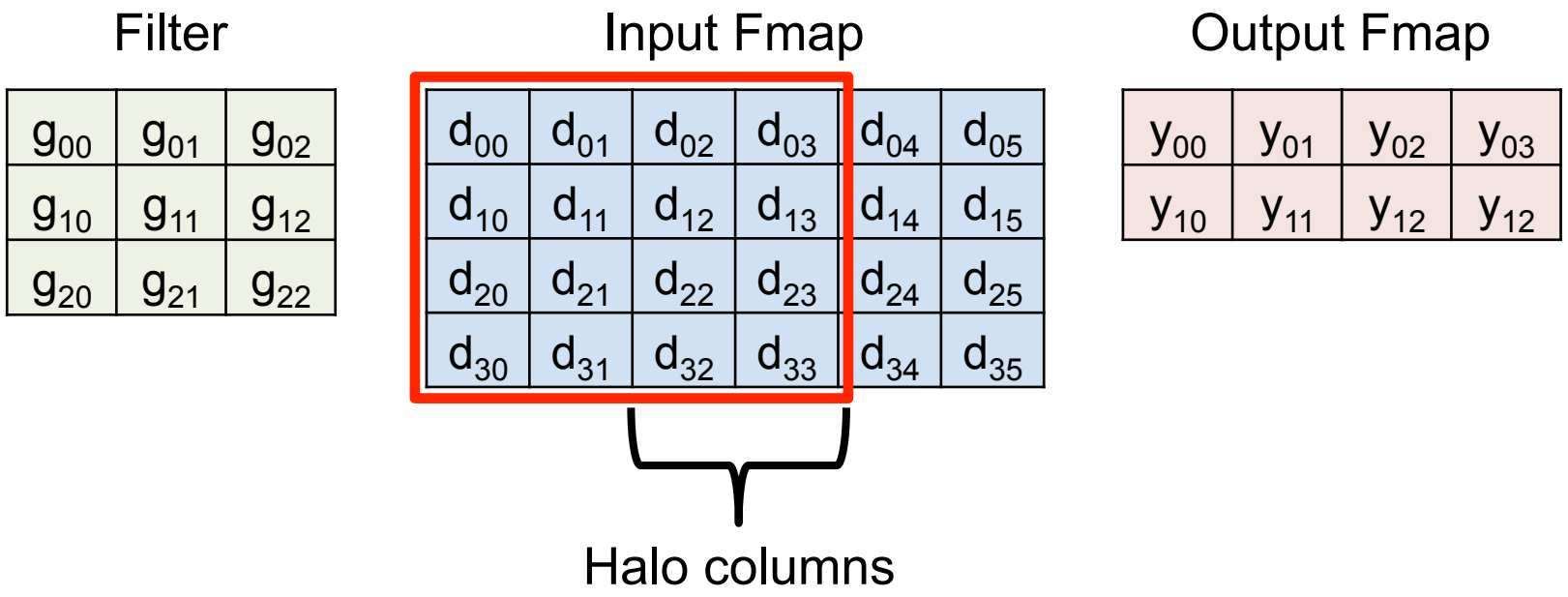


Original: 36 multiplications

Winograd: 16 multiplications → 2.25 times reduction

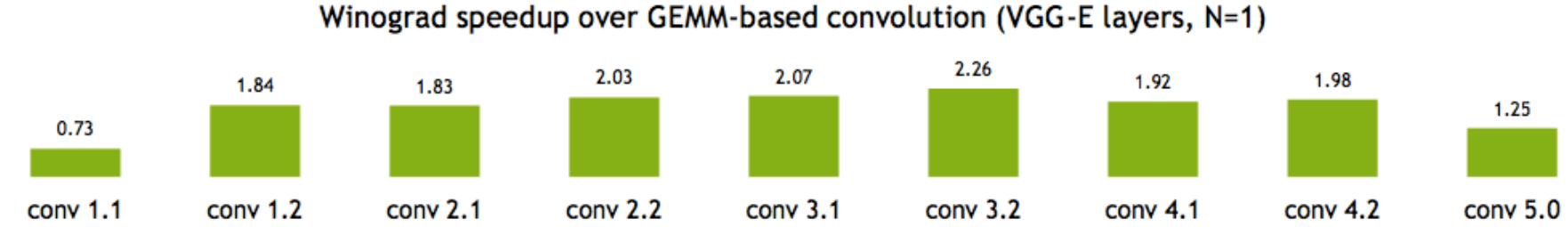
Winograd Halos

- Winograd works on a small region of output at a time, and therefore uses inputs repeatedly



Winograd Performance Varies

Optimal convolution algorithm depends on convolution layer dimensions



Meta-parameters (data layouts, texture memory) afford higher performance

Using texture memory for convolutions: 13% inference speedup

(GoogLeNet, batch size 1)

Source: Nvidia

Winograd Summary

- **Winograd is an optimized computation for convolutions**
- **It can significantly reduce multiplies**
  - **For example, for 3x3 filter by 2.5X**
- **But, each filter size is a different computation.**

Winograd as a Transform

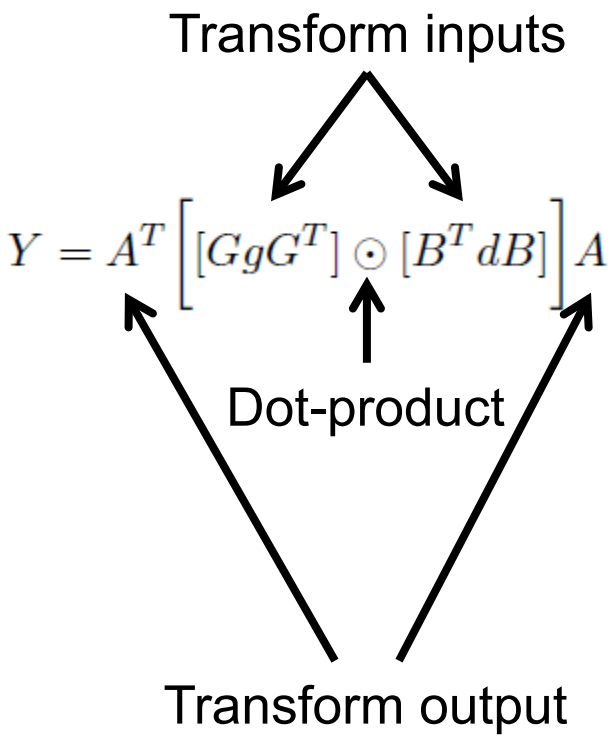
$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

filter $g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$

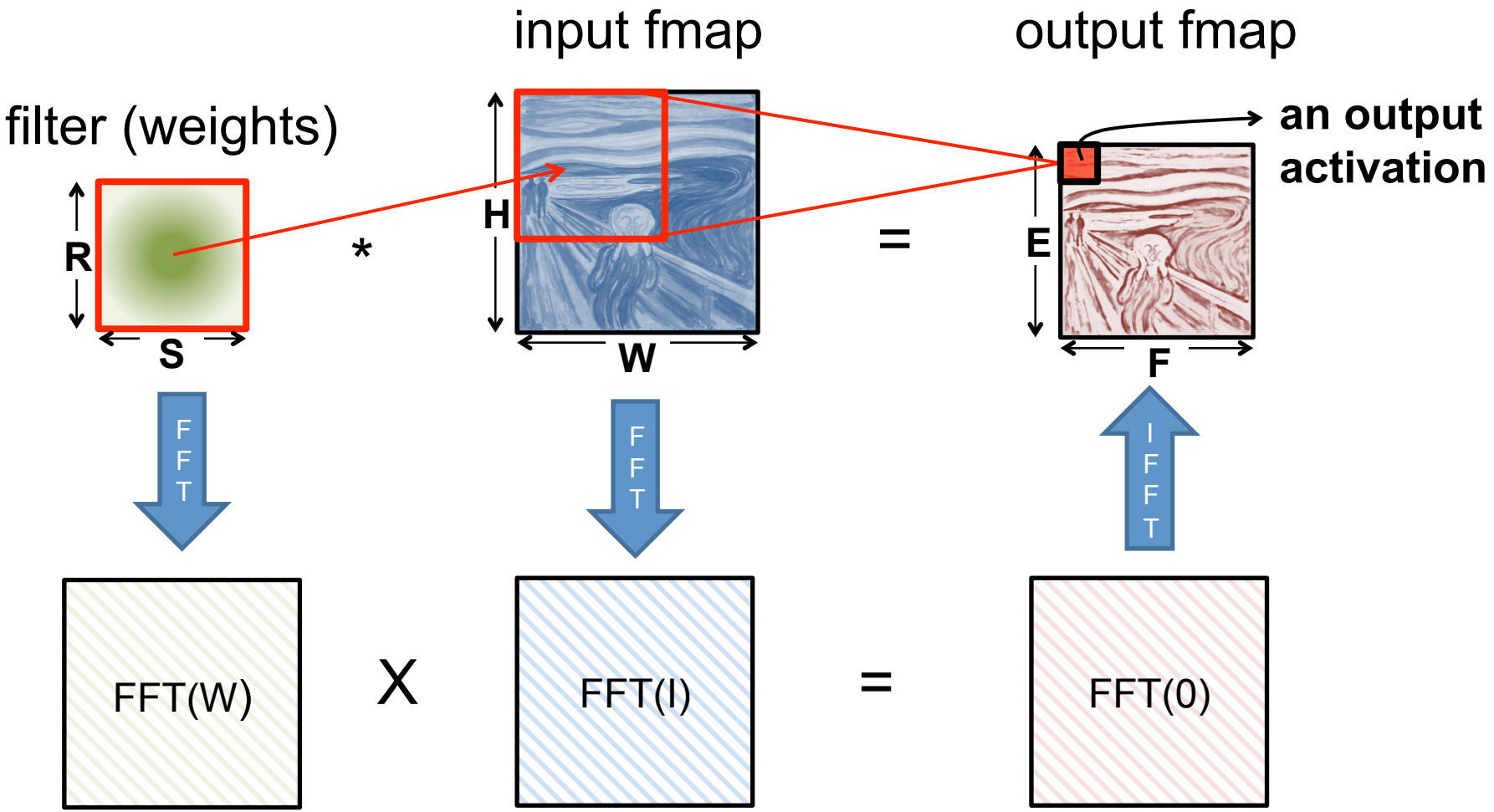
input $d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$



GgG^T can be precomputed

[Lavin et al., ArXiv 2015]

FFT Flow


input fmap
output fmap
filter (weights)
an output activation
R
S
H
W
E
F
*
=
FFT
FFT
IFFT
FFT(W)
X
FFT(I)
=
FFT(O)

FFT Overview

- **Convert filter and input to frequency domain to make convolution a simple multiply then convert back to time domain.**

- **Convert direct convolution $O(N_o^2 N_f^2)$ computation to $O(N_o^2 \log_2 N_o)$**

- **So note that computational benefit of FFT decreases with decreasing size of filter**

[Mathieu et al., ArXiv 2013, Vasilache et al., ArXiv 2014]

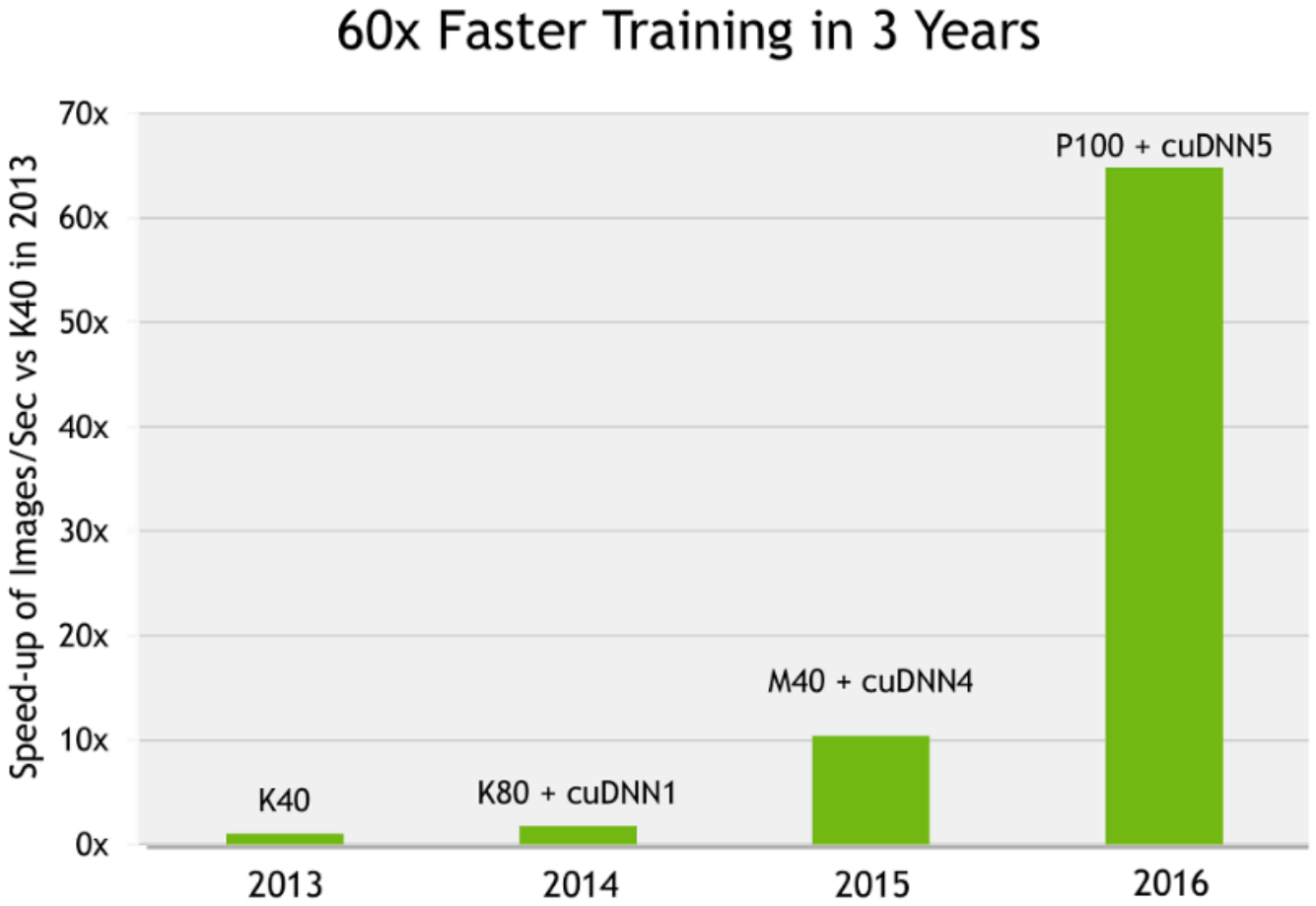
FFT Costs

- Input and Filter matrices are ‘0-completed’,
  - i.e., expanded to size E+R-1 x F+S-1
- Frequency domain matrices are same dimensions as input, but complex.
- FFT often reduces computation, but requires much more memory space and bandwidth

Optimization opportunities

- FFT of real matrix is symmetric allowing one to save ½ the computes
- Filters can be pre-computed and stored, but convolutional filter in frequency domain is much larger than in time domain
- Can reuse frequency domain version of input for creating different output channels to avoid FFT re-computations

cuDNN: Speed up with Transformations



Source: Nvidia

GPU/CPU Benchmarking

- **Industry performance website**
- **https://github.com/jcjohnson/cnn-benchmarks**
- **DeepBench**
  - **Profile layer by layer (Dense Matrix Multiplication, Convolutions, Recurrent Layer, All-Reduce)**

GPU/CPU Benchmarking

- **Minibatch = 16**
- **Image size 224x224**
- **cuDNN 5.0 or 5.1**
- **Torch**

Speed (ms)

Platform	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Pascal Titan X (F+B)	14.56	128.62	39.14	103.58
Pascal Titan X (F)	5.04	41.59	11.94	35.03
GTX 1080 (F)	7.00	59.37	16.08	50.64
Maxwell Titan X	7.09	62.30	19.27	55.75
Dual Xeon E5-2630 v3	n/a	3101.76	n/a	2477.61

https://github.com/jcjohnson/cnn-benchmarks

GPU/CPU Benchmarking

- **Minibatch = 1**
- **Image size 224x224**

Speed (ms)

Platform	AlexNet	VGG-16
Maxwell Titan X	0.54	10.67
Exynos 7 Octa 7420	117	1926

[Kim et al., ICLR 2016]

DeepBench

- **Profile layer by layer**
  - **Dense Matrix Multiplication, Convolutions, Recurrent Layer, All-Reduce (communication)**

3.2. Convolution Results

Input Size	Filter Size	# of Filters	Padding (h, w)	Stride (h, w)	Application	Total Time (ms)	Fwd TeraFLOPS	Processor
W = 700, H = 161, C = 1, N = 32	R = 5, S = 20	32	0, 0	2, 2	Speech Recognition	2.98	6.63	TitanX Pascal
W = 54, H = 54, C = 64, N = 8	R = 3, S = 3	64	1, 1	1, 1	Face Recognition	0.63	10.55	TitanX Pascal
W = 224, H = 224, C = 3, N = 16	R = 3, S = 3	64	1, 1	1, 1	Computer Vision	3.99	3.6	TitanX Pascal
W = 7, H = 7, C = 512, N = 16	R = 3, S = 3	512	1, 1	1, 1	Computer Vision	2.93	5.88	TitanX Pascal
W = 28, H = 28, C = 192, N = 16	R = 5, S = 5	32	2, 2	1, 1	Computer Vision	1.57	6.59	TitanX Pascal

https://svail.github.io/DeepBench/

DNN Accelerator Architectures

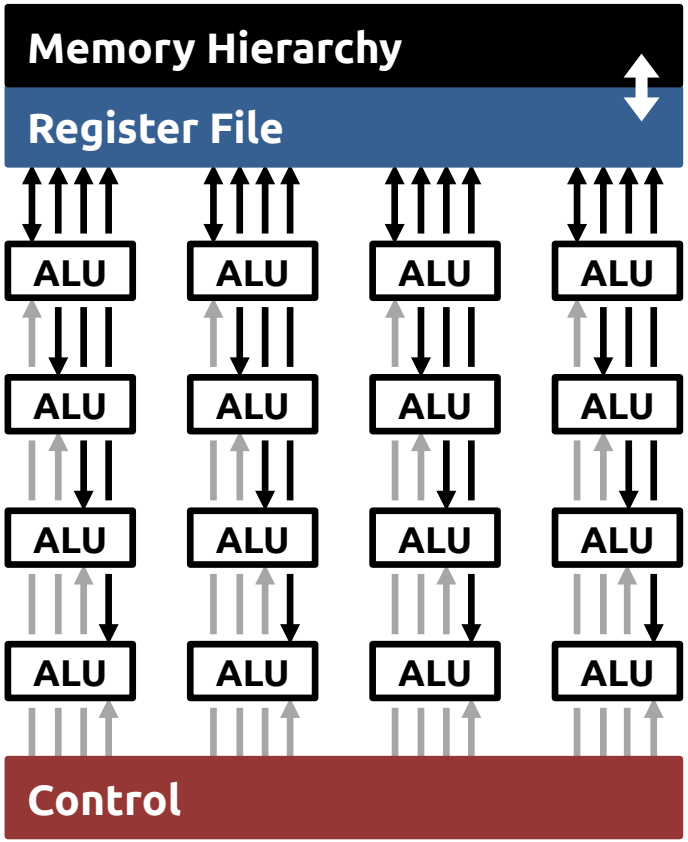
MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

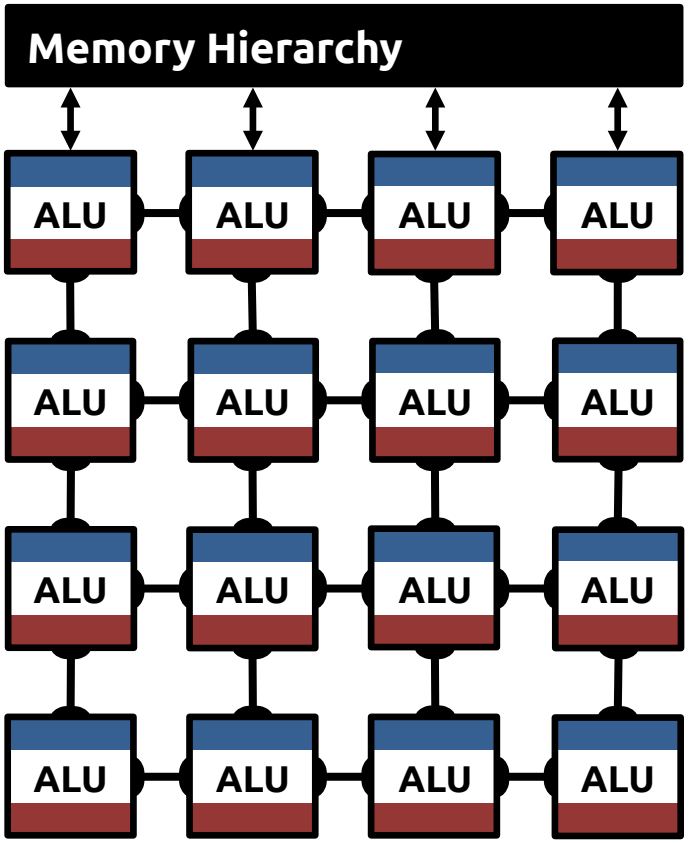
Joel Emer, Vivienne Sze, Yu-Hsin Chen

Highly-Parallel Compute Paradigms

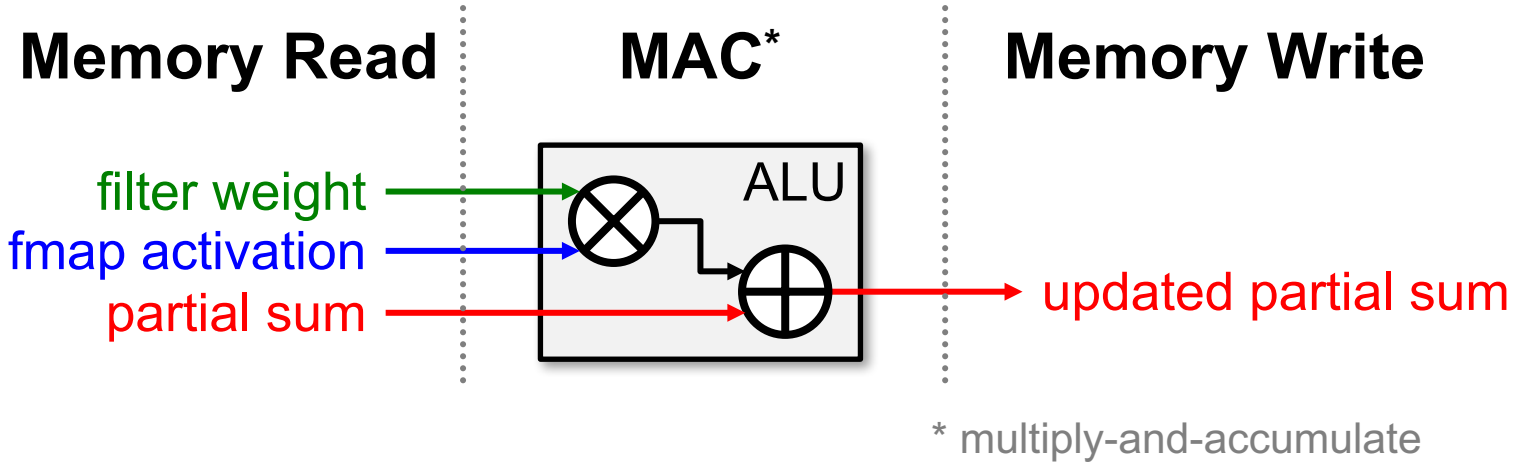
Temporal Architecture (SIMD/SIMT)



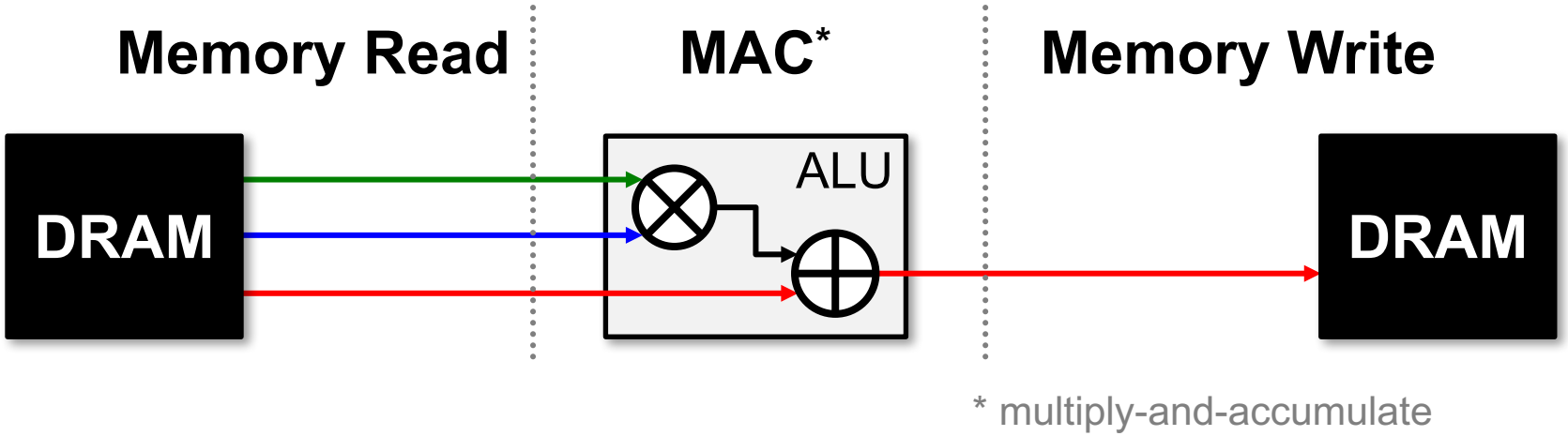
Spatial Architecture (Dataflow Processing)



Memory Access is the Bottleneck



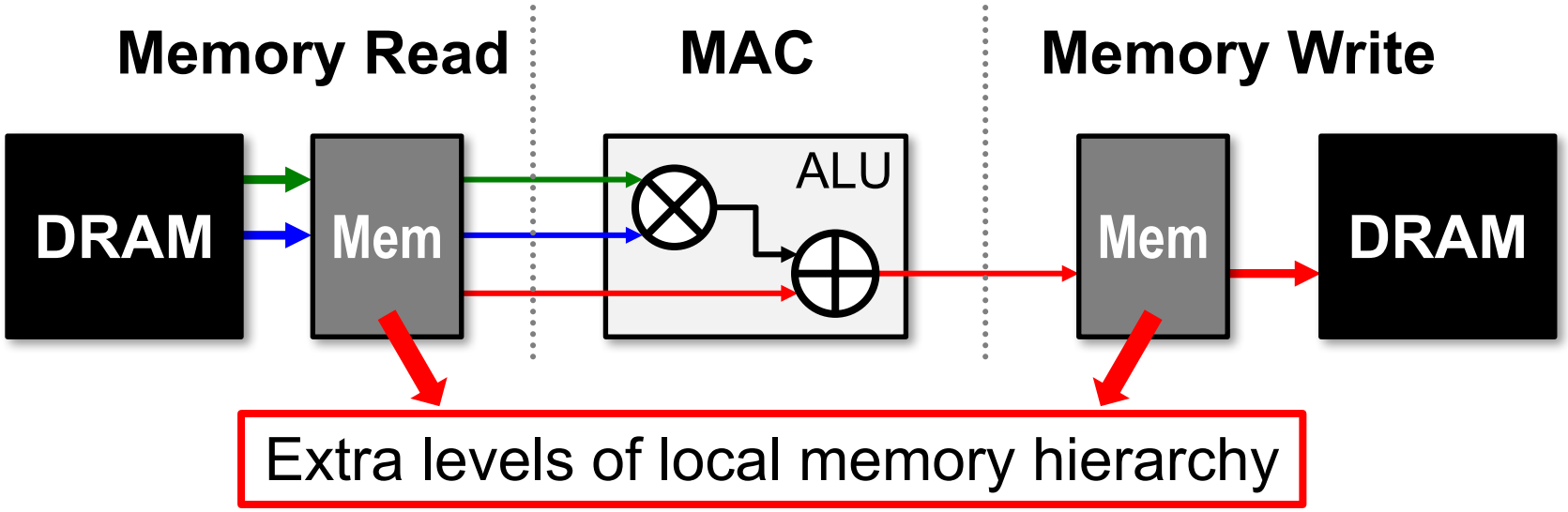
Memory Access is the Bottleneck



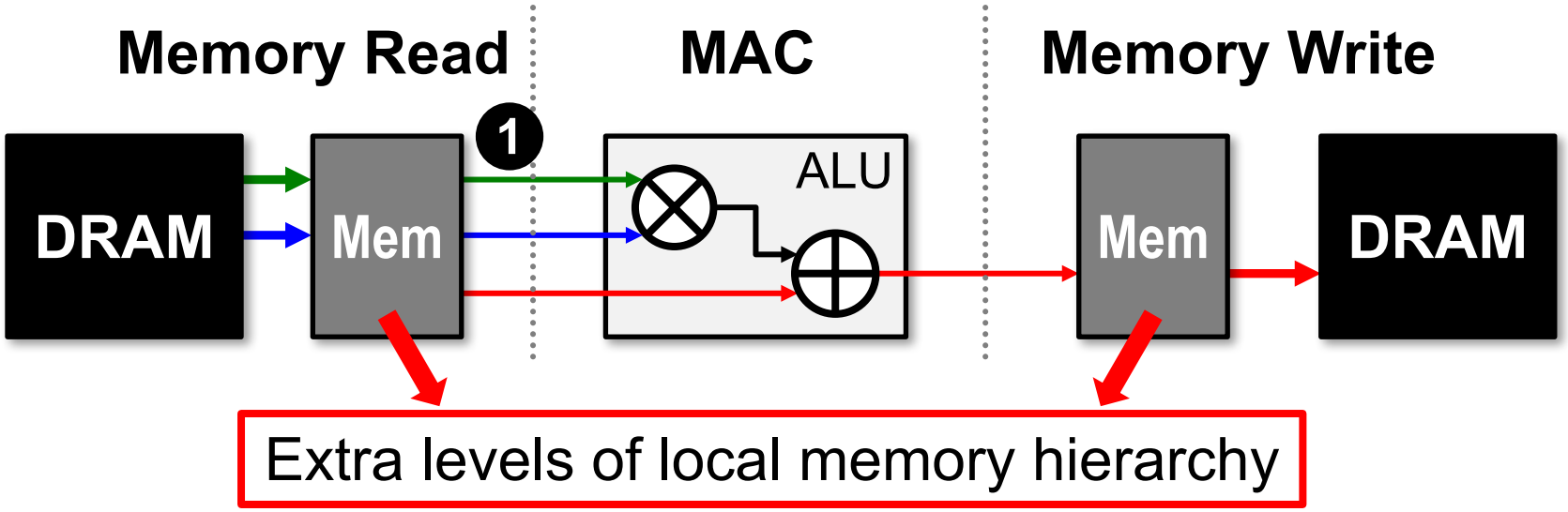
Worst Case: all memory R/W are **DRAM** accesses

- Example: AlexNet [NIPS 2012] has **724M** MACs
  → **2896M** DRAM accesses required

Memory Access is the Bottleneck



Memory Access is the Bottleneck

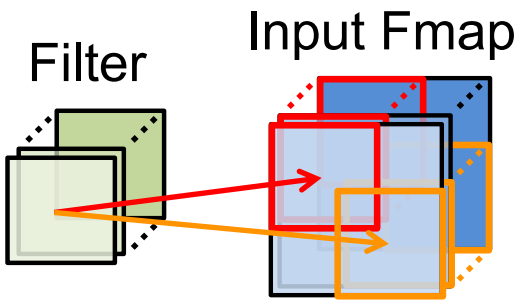


Opportunities: 1 **data reuse**

Types of Data Reuse in DNN

Convolutional Reuse

CONV layers only
(sliding window)

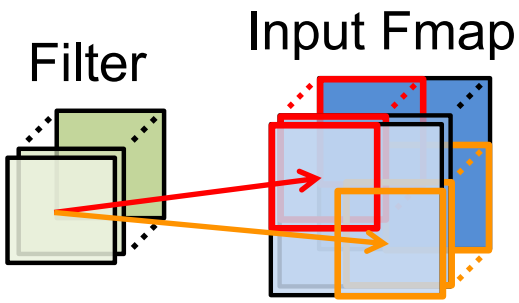


Reuse: Activations
Filter weights

Types of Data Reuse in DNN

Convolutional Reuse

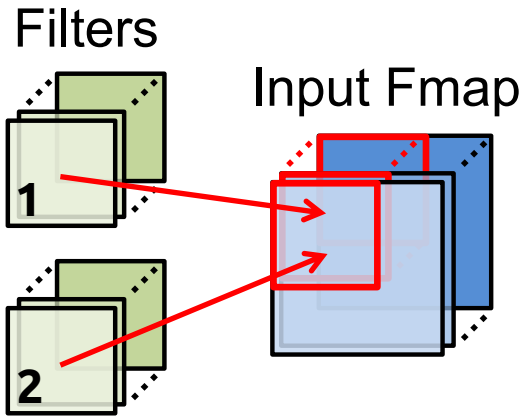
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

CONV and FC layers

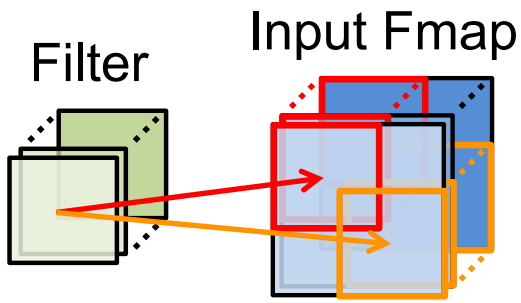


Reuse: Activations

Types of Data Reuse in DNN

Convolutional Reuse

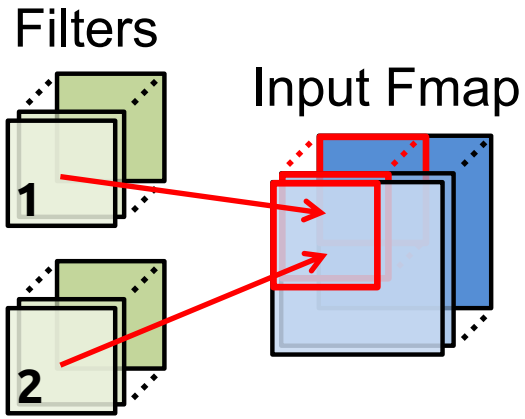
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

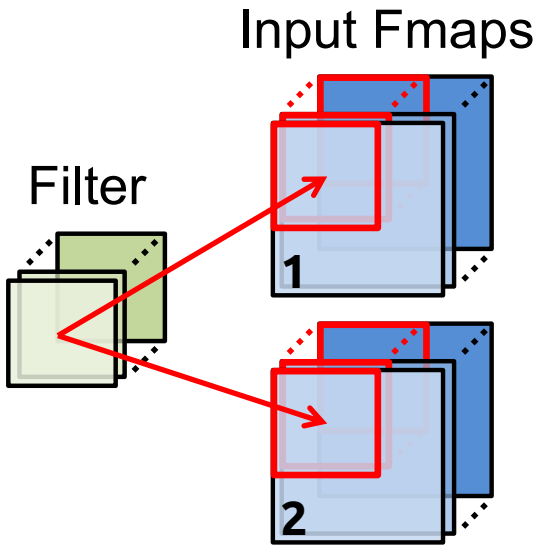
CONV and FC layers



Reuse: Activations

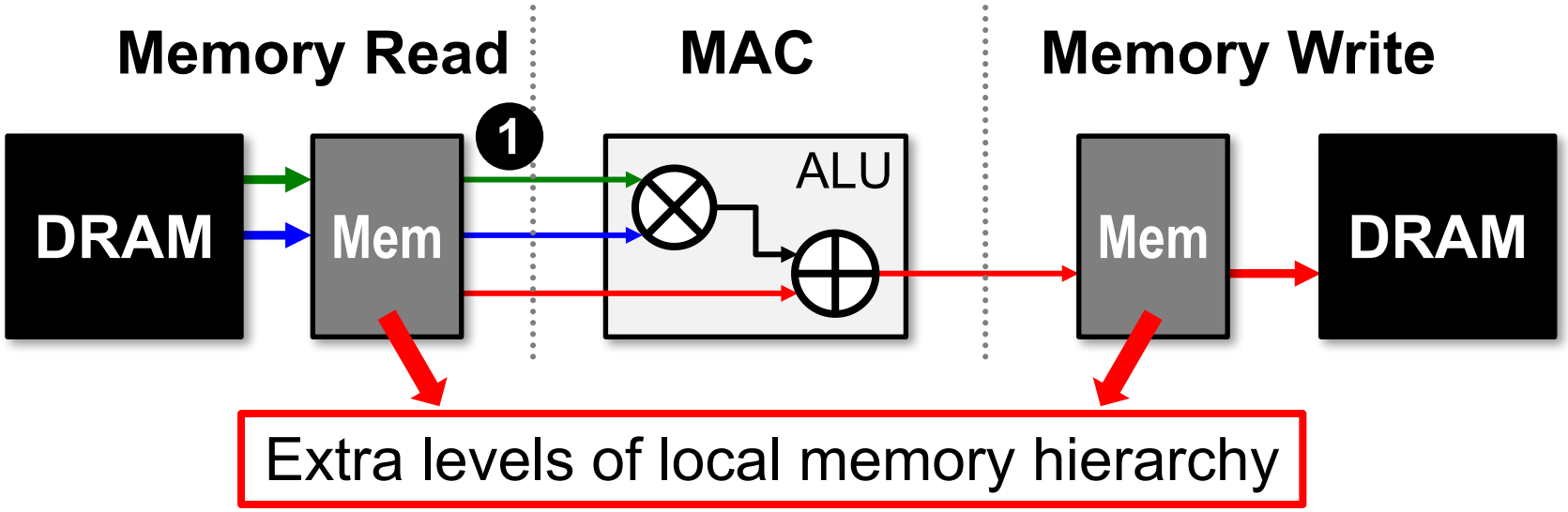
Filter Reuse

CONV and FC layers
(batch size > 1)



Reuse: Filter weights

Memory Access is the Bottleneck

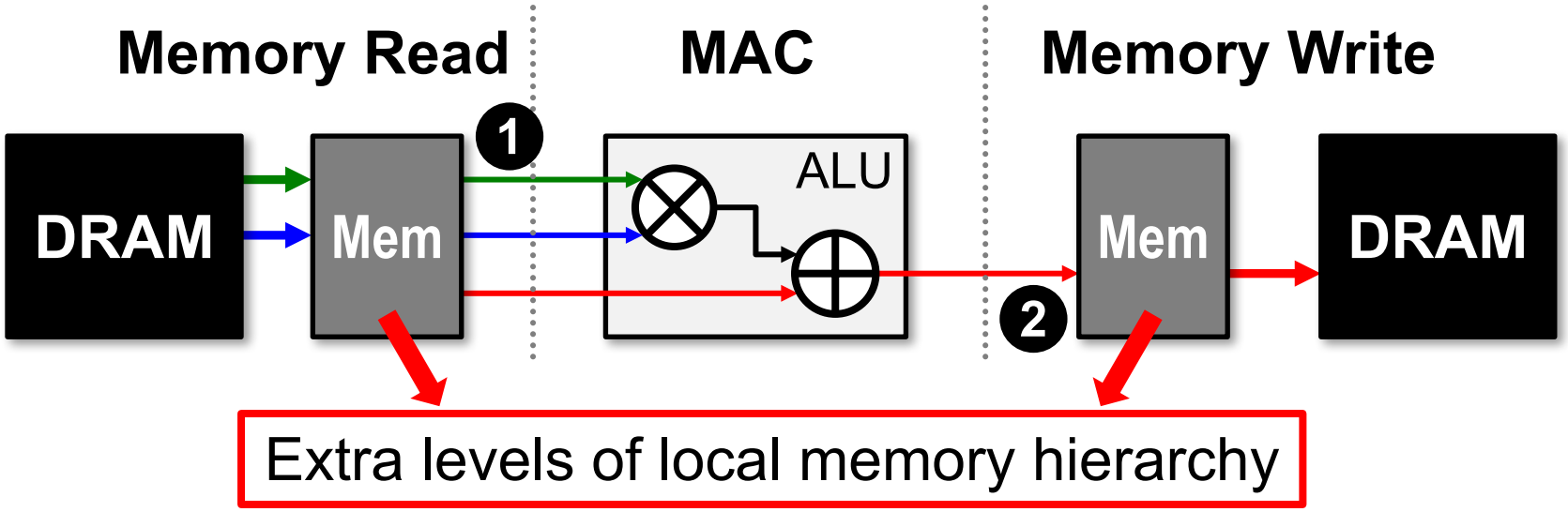


Opportunities: 1 **data reuse**

1 Can reduce DRAM reads of filter/fmap by up to **500×****

** AlexNet CONV layers

Memory Access is the Bottleneck

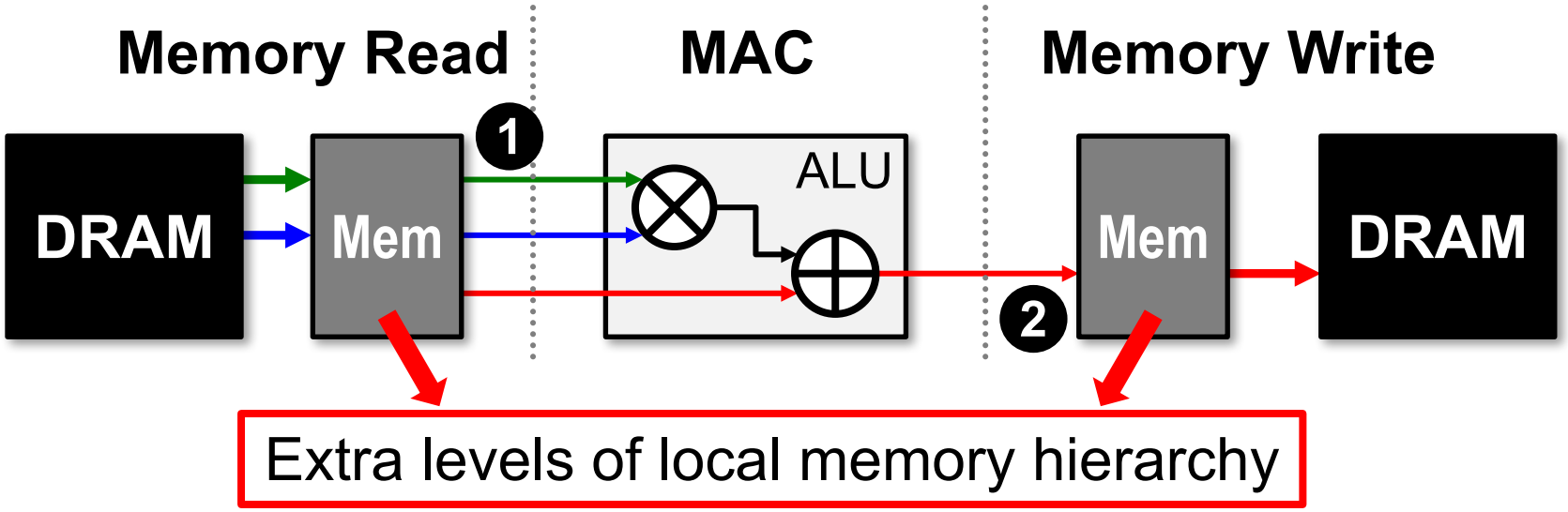


Opportunities: ❶ **data reuse** ❷ **local accumulation**

❶ Can reduce DRAM reads of filter/fmap by up to **500×**

❷ Partial sum accumulation does **NOT** have to access DRAM

Memory Access is the Bottleneck



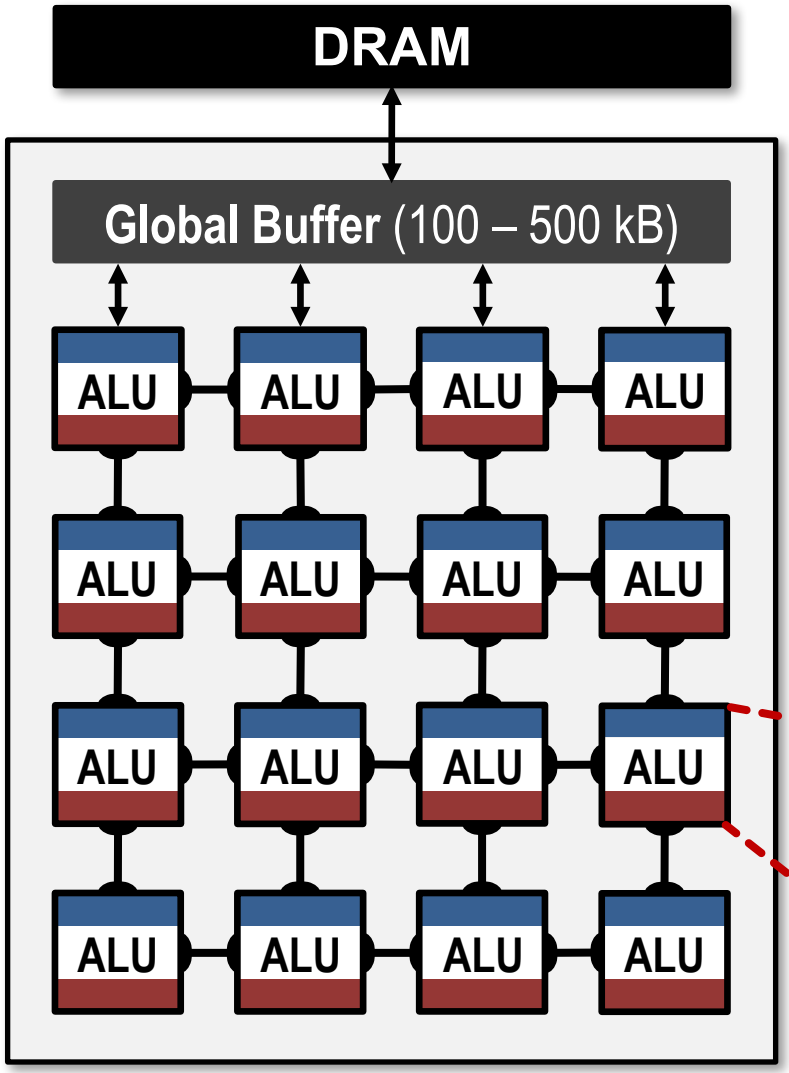
Opportunities: ❶ **data reuse** ❷ **local accumulation**

❶ Can reduce DRAM reads of filter/fmap by up to **500×**

❷ Partial sum accumulation does **NOT** have to access DRAM

- Example: DRAM access in AlexNet can be reduced from **2896M** to **61M** (best case)

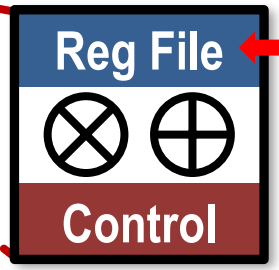
Spatial Architecture for CNN



Local Memory Hierarchy

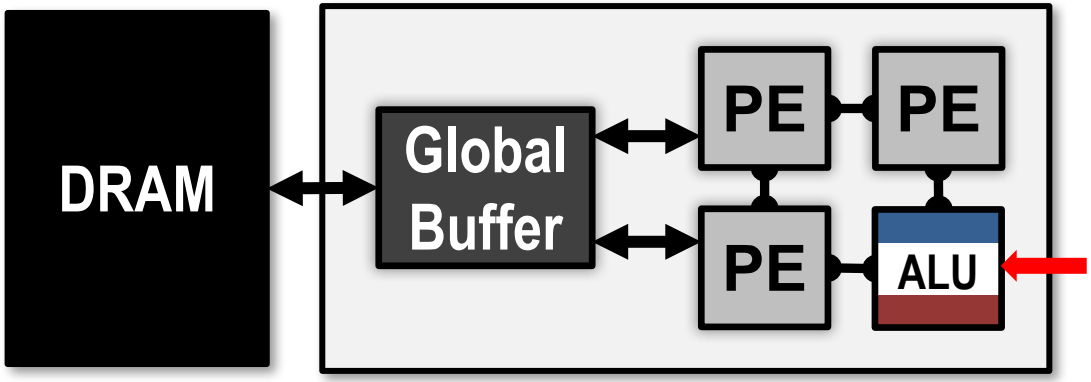
- Global Buffer
- Direct inter-PE network
- PE-local memory (RF)

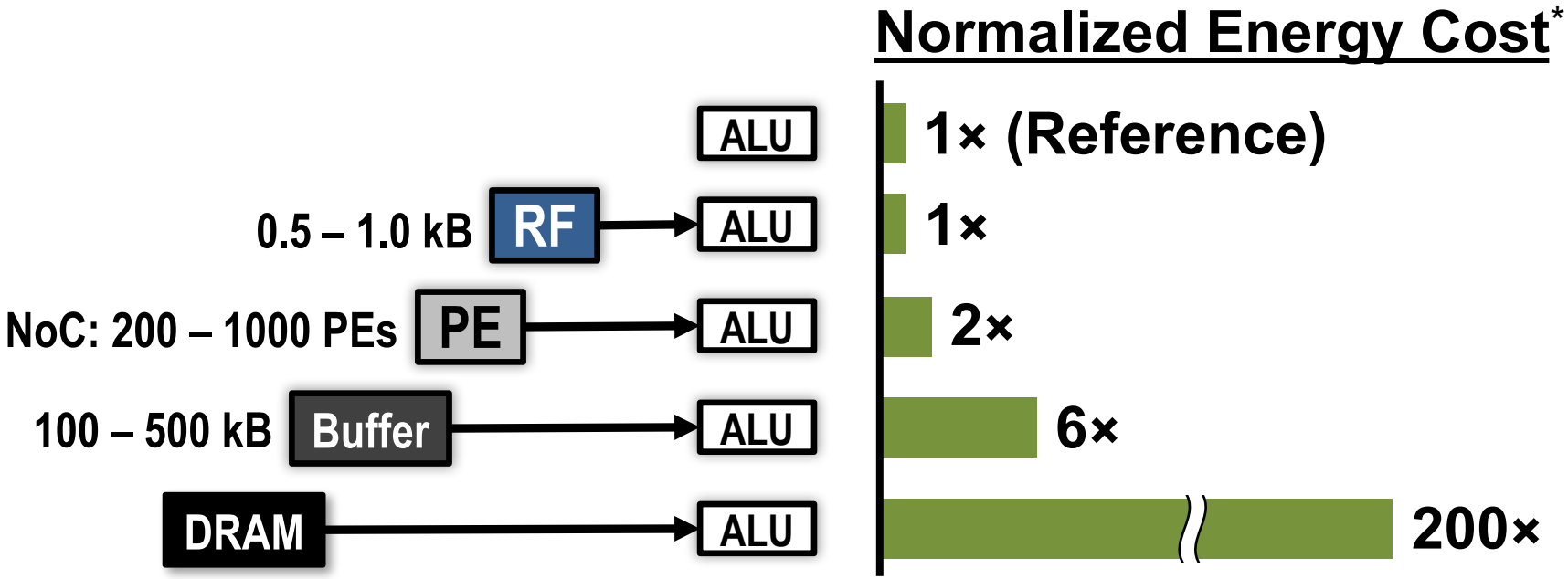
Processing Element (PE)



0.5 – 1.0 kB

Low-Cost Local Data Access

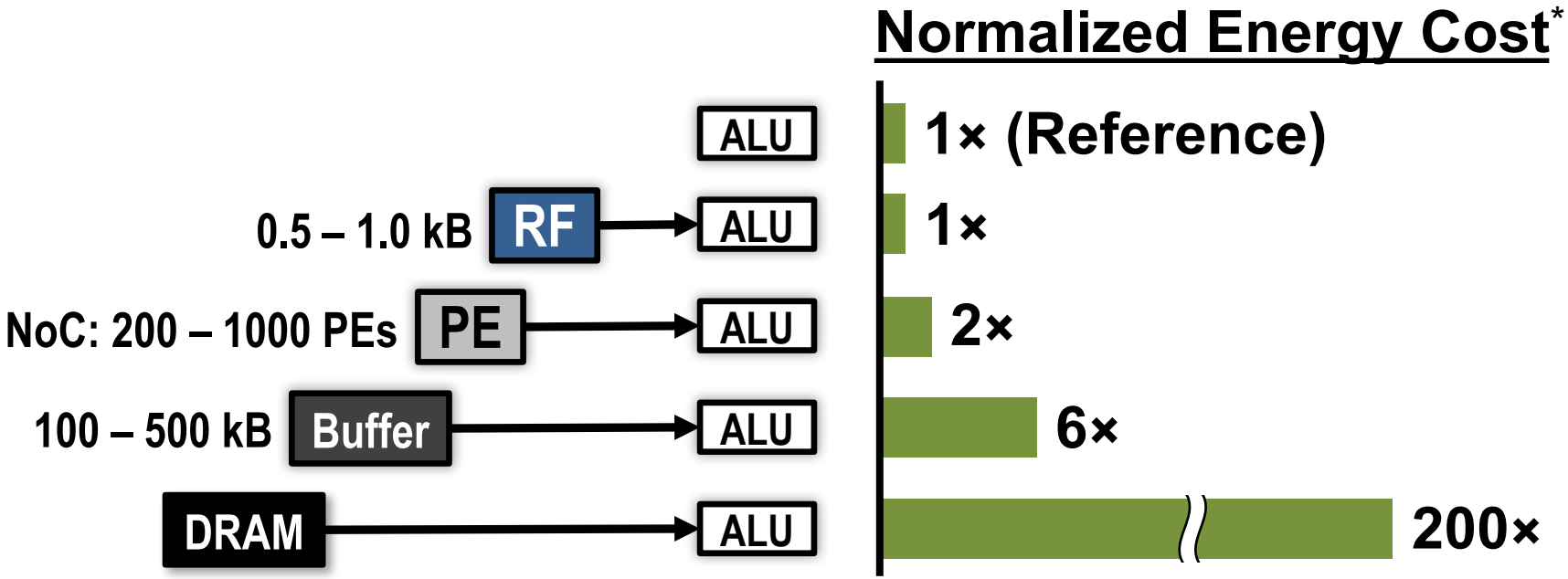




* measured from a commercial 65nm process

Low-Cost Local Data Access

How to exploit ❶ **data reuse** and ❷ **local accumulation** with *limited* low-cost local storage?

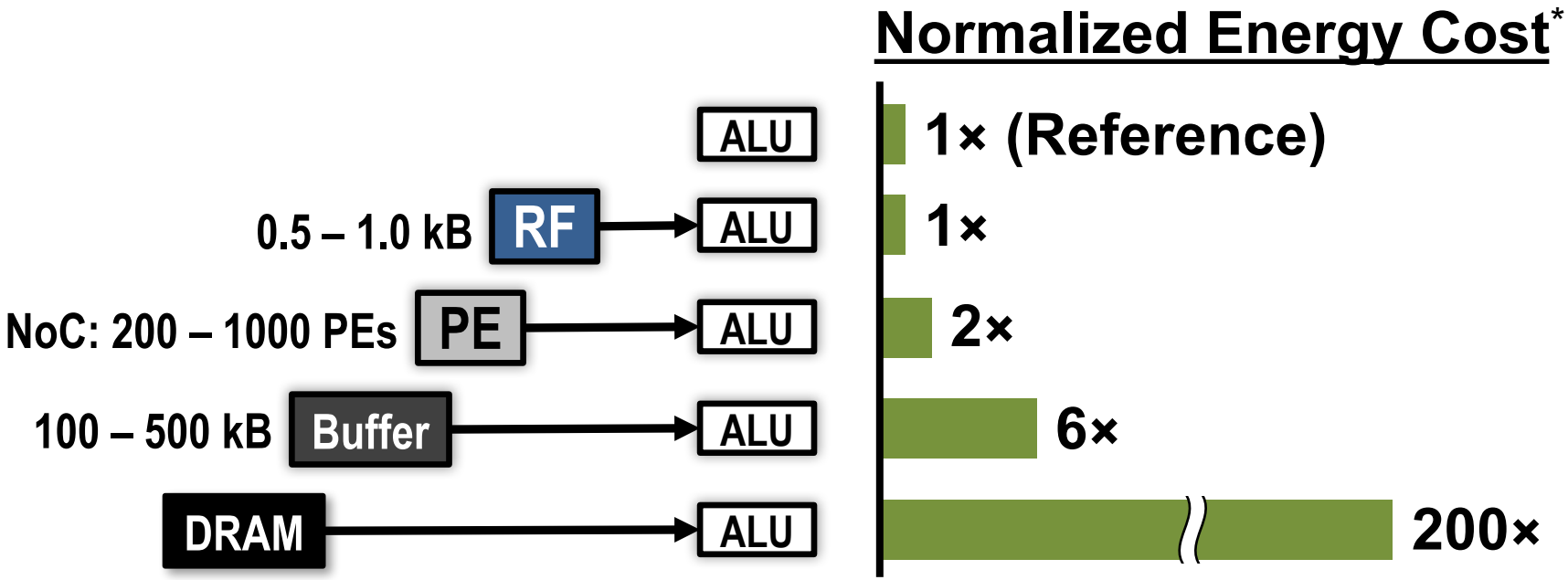


* measured from a commercial 65nm process

Low-Cost Local Data Access

How to exploit ❶ **data reuse** and ❷ **local accumulation** with *limited* low-cost local storage?

specialized **processing dataflow** required!



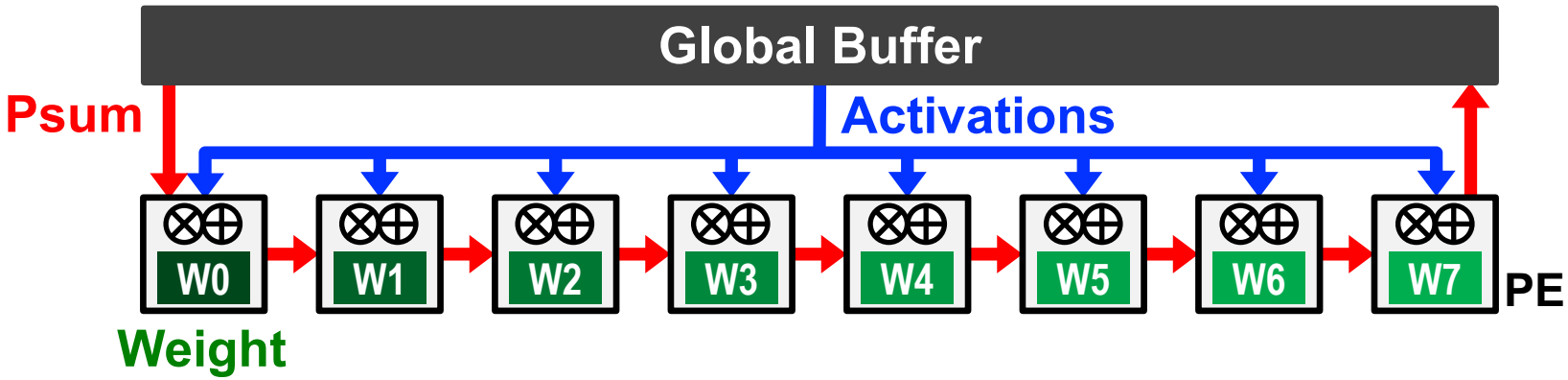
* measured from a commercial 65nm process

Dataflow Taxonomy

- Weight Stationary (WS)
- Output Stationary (OS)
- No Local Reuse (NLR)

[Chen et al., ISCA 2016]

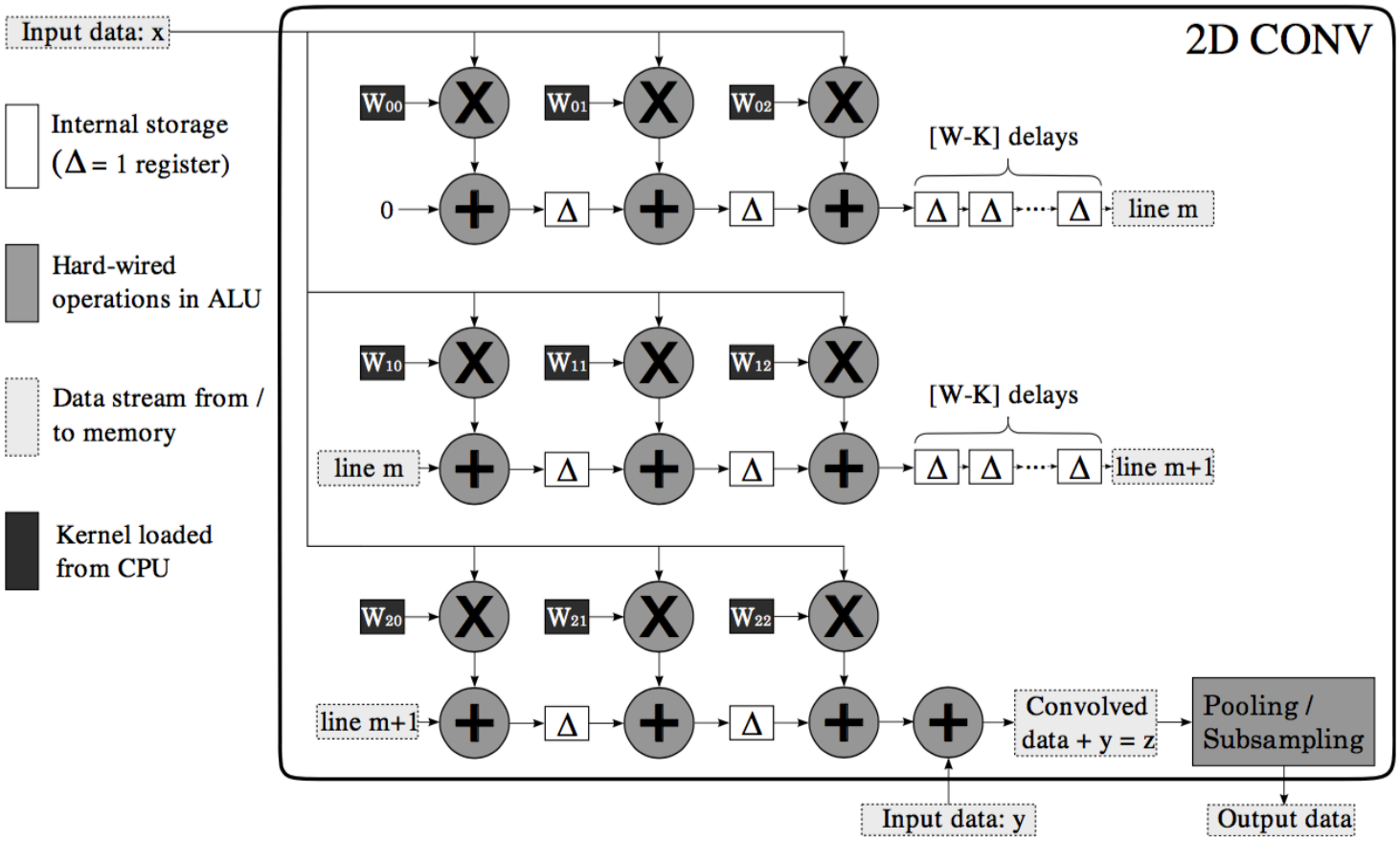
Weight Stationary (WS)



- **Minimize weight** read energy consumption
  - maximize convolutional and filter reuse of weights
- **Broadcast activations** and **accumulate psums spatially** across the PE array.

WS Example: nn-X (NeuFlow)

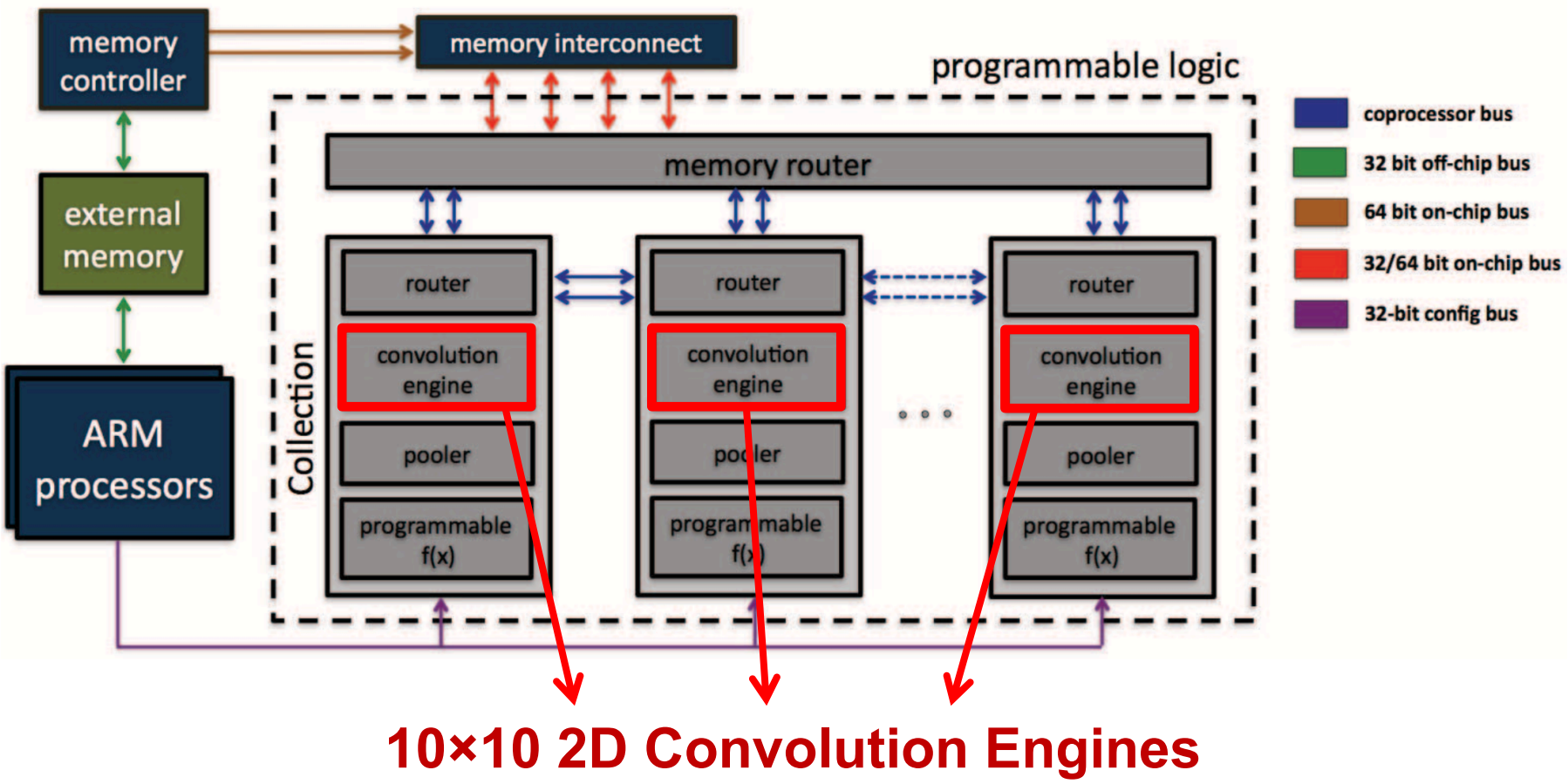
A 3×3 2D Convolution Engine



[Farabet et al., ICCV 2009]

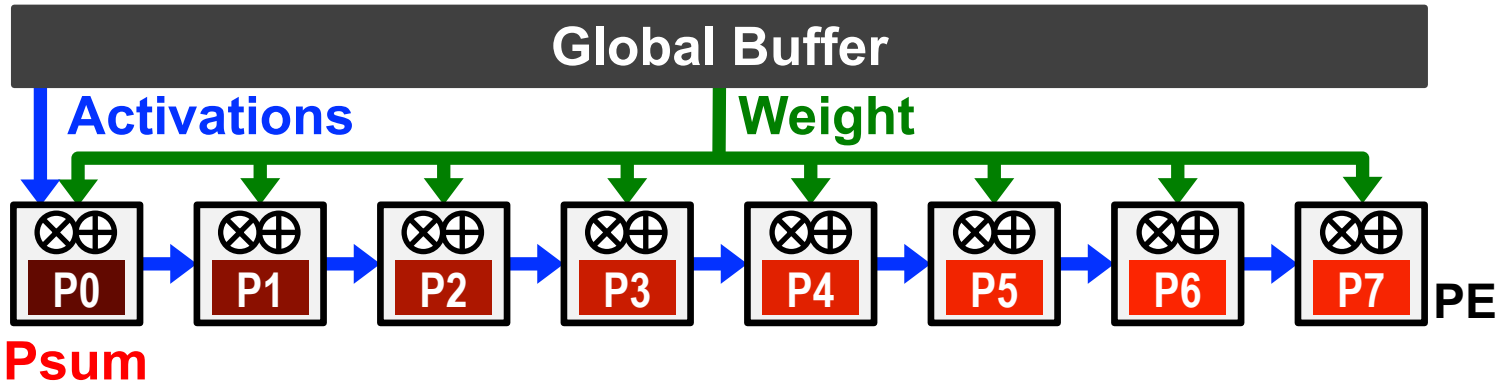
WS Example: nn-X (NeuFlow)

Top-Level Architecture



[Gokhale et al., CVPRW 2014]

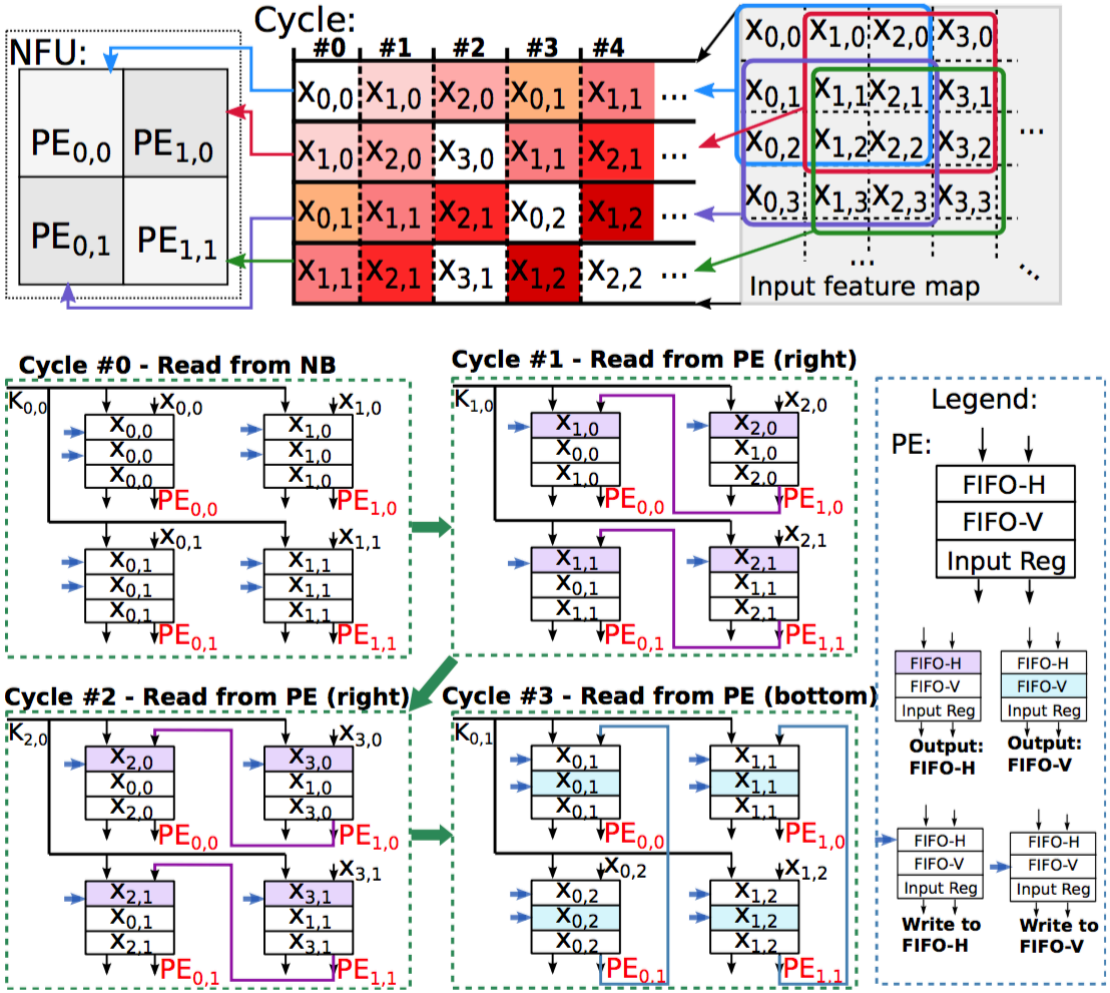
Output Stationary (OS)



- **Minimize partial sum** R/W energy consumption
  - maximize local accumulation
- **Broadcast/Multicast filter weights** and **reuse activations spatially** across the PE array

OS Example: ShiDianNao

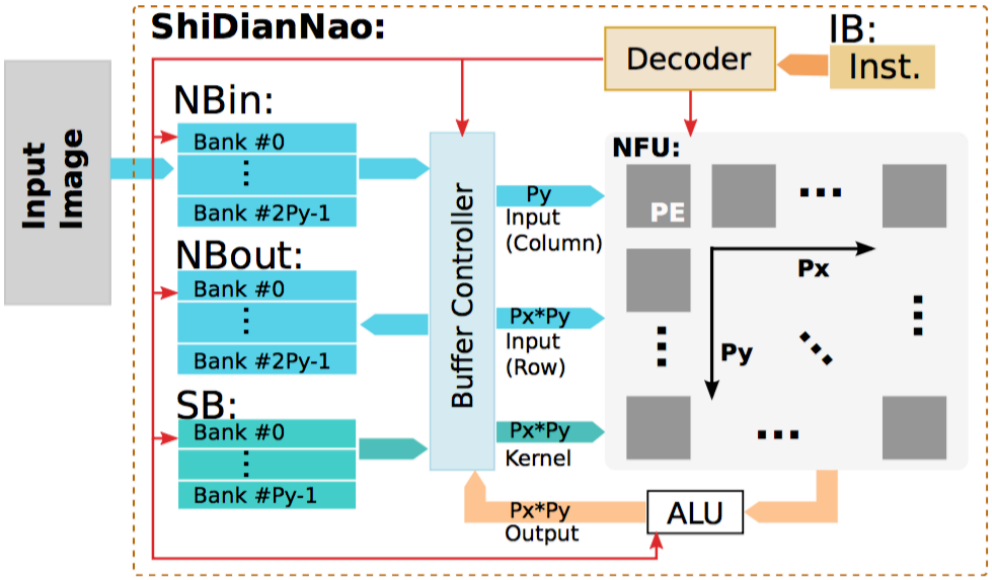
Input Fmap Dataflow in the PE Array



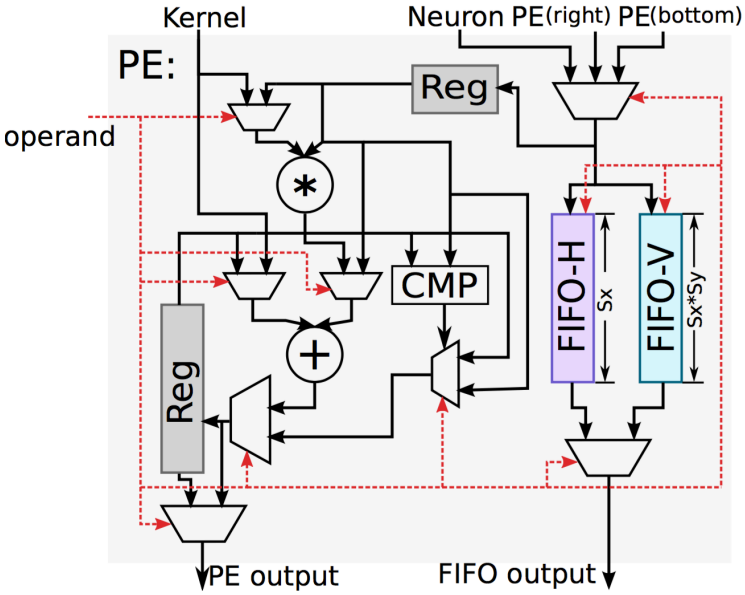
[Du et al., ISCA 2015]

OS Example: ShiDianNao

Top-Level Architecture

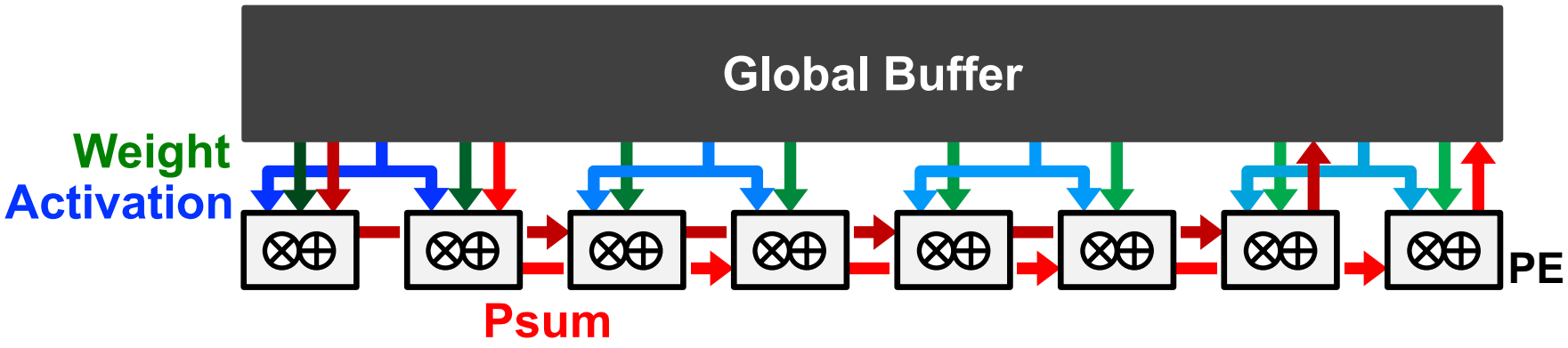


PE Architecture



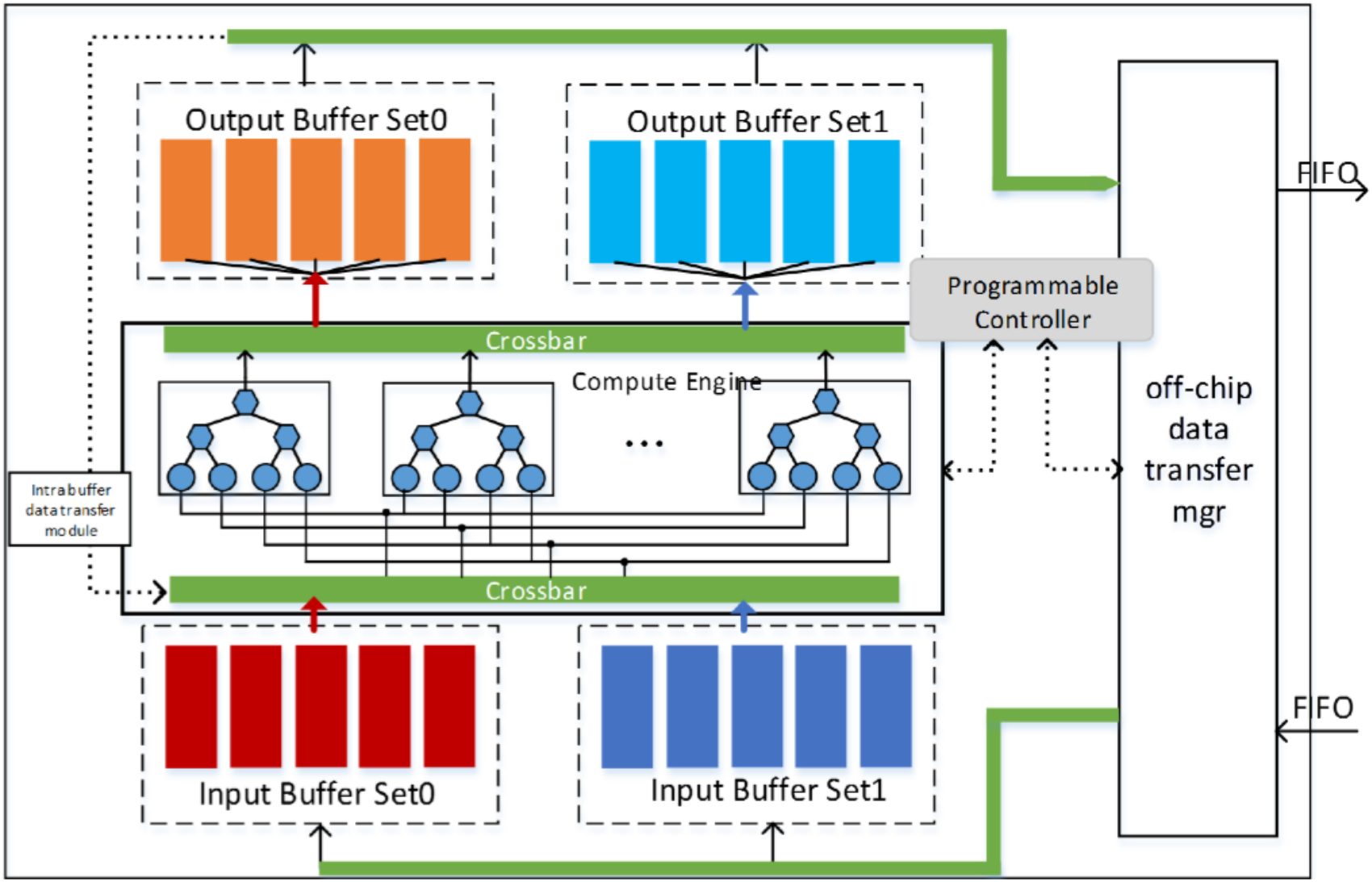
[Du et al., ISCA 2015]

No Local Reuse (NLR)



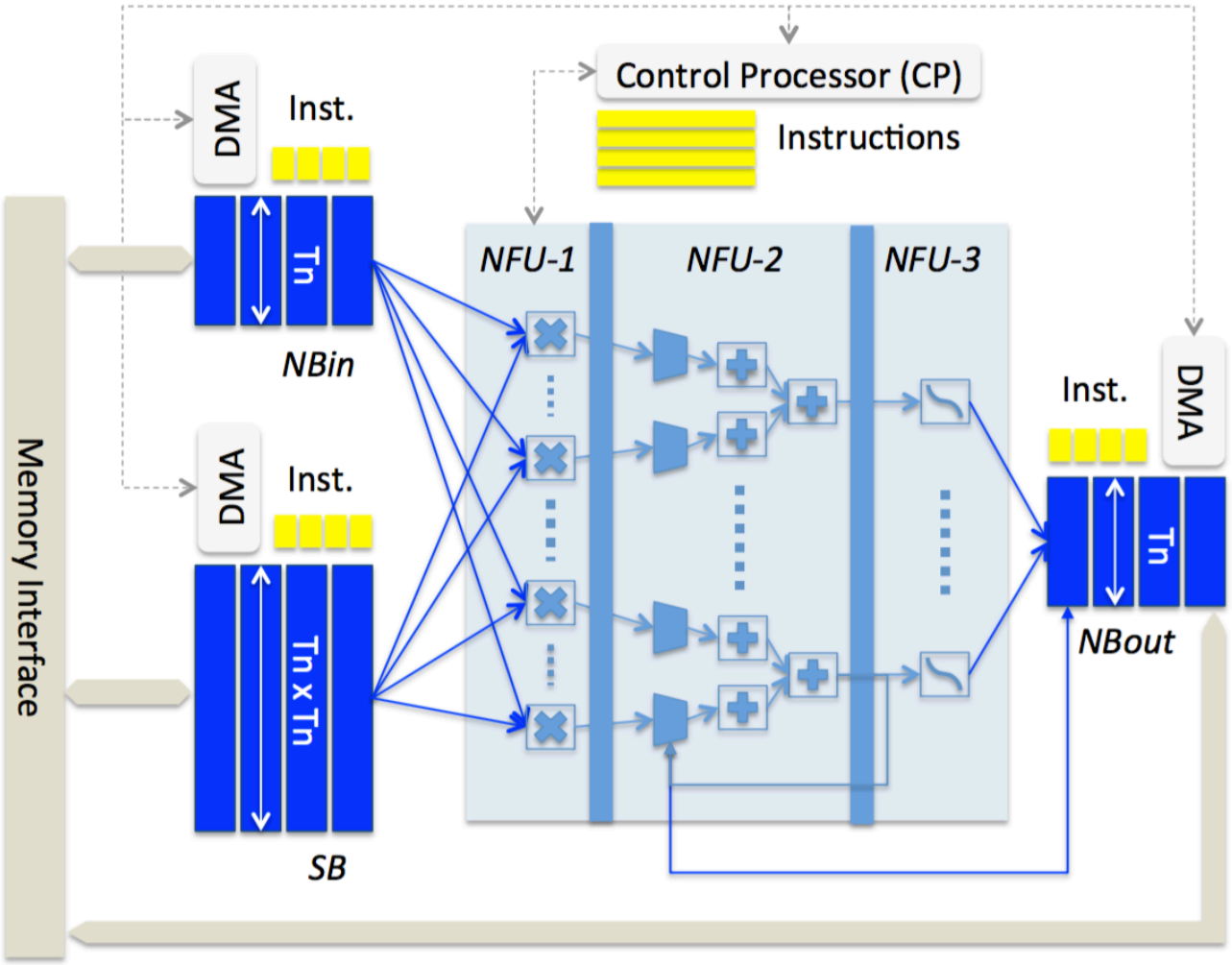
- Use a **large global buffer** as shared storage
  - Reduce **DRAM** access energy consumption
- **Multicast activations, single-cast weights, and accumulate psums spatially** across the PE array

NLR Example: UCLA



[Zhang et al., FPGA 2015]

NLR Example: DianNao



[Chen et al., ASPLOS 2014]

Taxonomy: More Examples

- **Weight Stationary (WS)**

  [**Chakradhar**, *ISCA* 2010] [**nn-X (NeuFlow)**, *CVPRW* 2014]
  [**Park**, *ISSCC* 2015] [**ISAAC**, *ISCA* 2016] [**PRIME**, *ISCA* 2016]

- **Output Stationary (OS)**

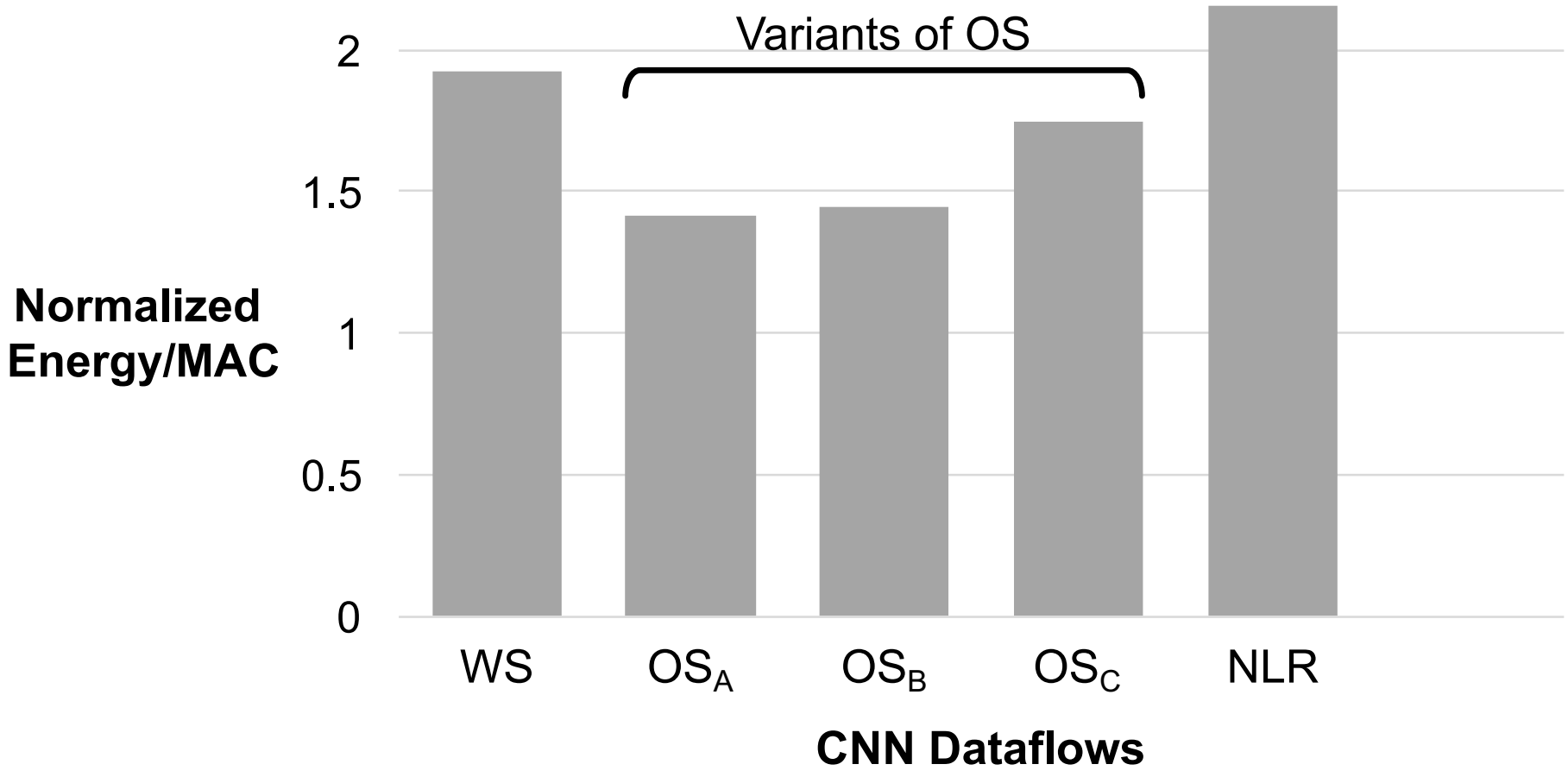
  [**Peemen**, *ICCD* 2013] [**ShiDianNao**, *ISCA* 2015]
  [**Gupta**, *ICML* 2015] [**Moons**, *VLSI* 2016]

- **No Local Reuse (NLR)**

  [**DianNao**, *ASPLOS* 2014] [**DaDianNao**, *MICRO* 2014]
  [**Zhang**, *FPGA* 2015]

Energy Efficiency Comparison

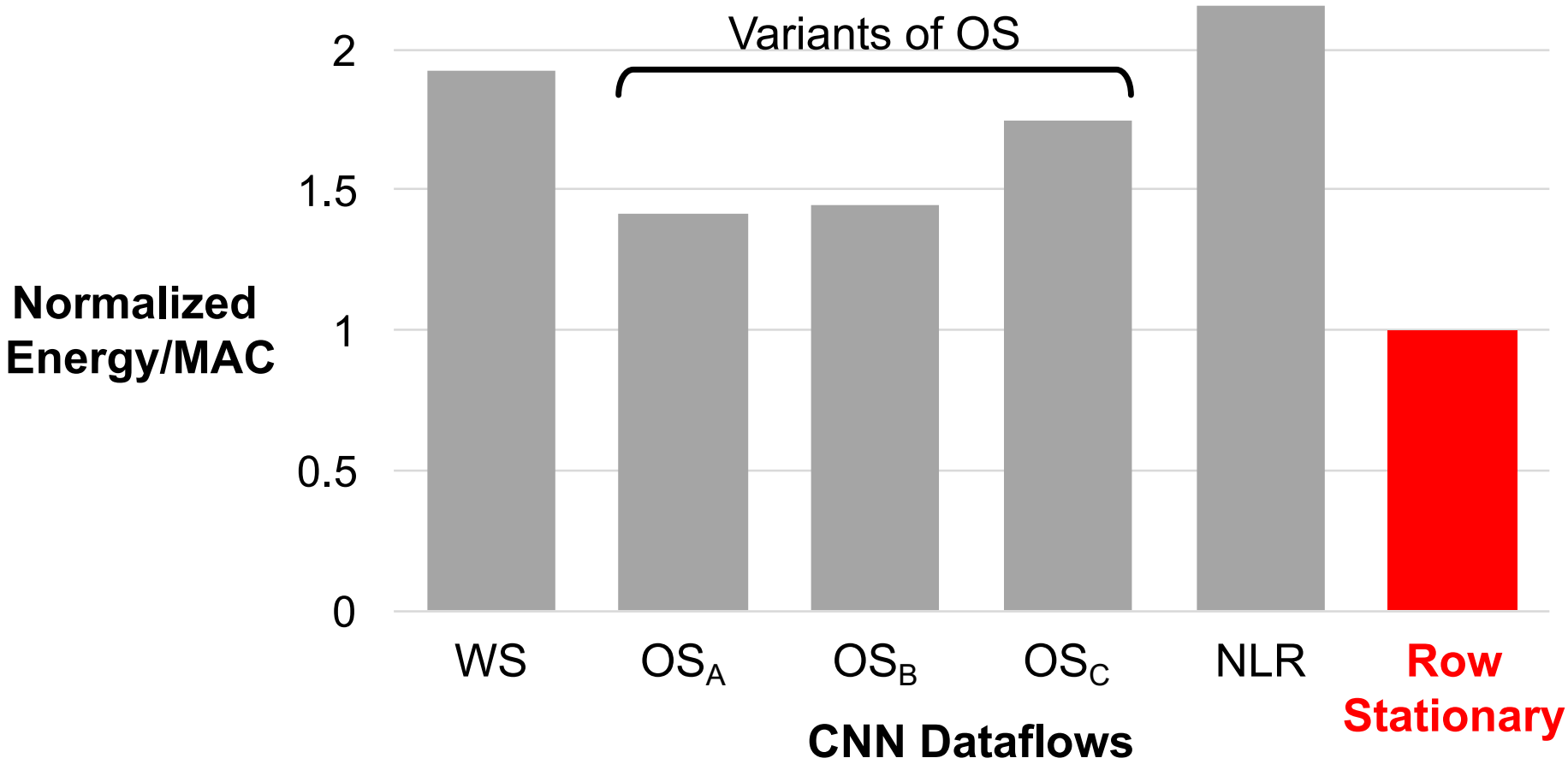
- Same total area
- AlexNet CONV layers
- 256 PEs
- Batch size = 16



[Chen et al., ISCA 2016]

Energy Efficiency Comparison

- Same total area
- AlexNet CONV layers
- 256 PEs
- Batch size = 16



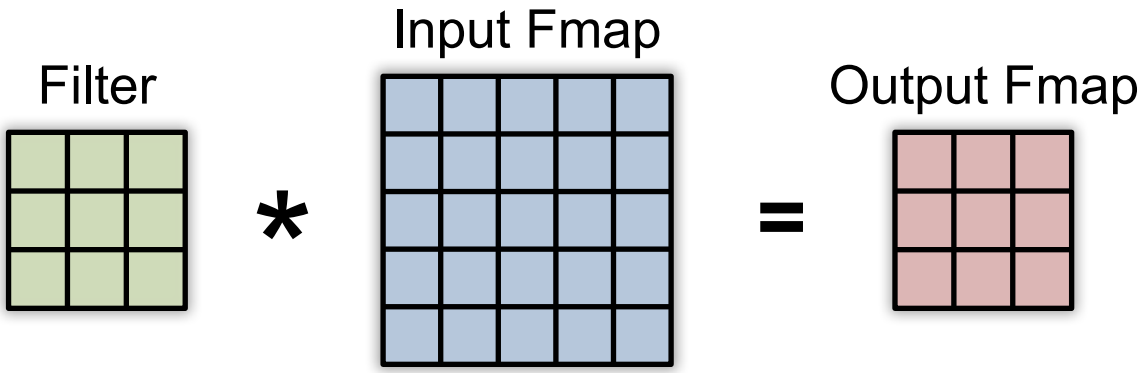
[Chen et al., ISCA 2016]

Energy-Efficient Dataflow: Row Stationary (RS)

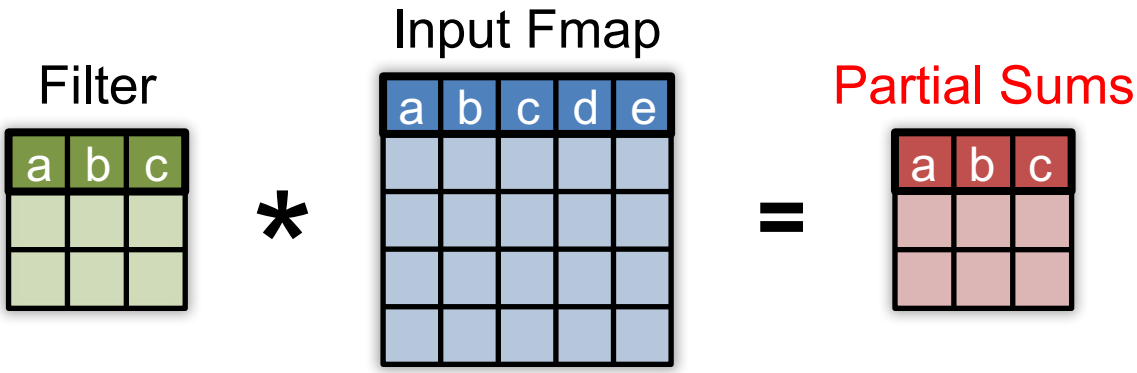
- **Maximize** reuse and accumulation at **RF**
- Optimize for **overall** energy efficiency instead for *only* a certain data type

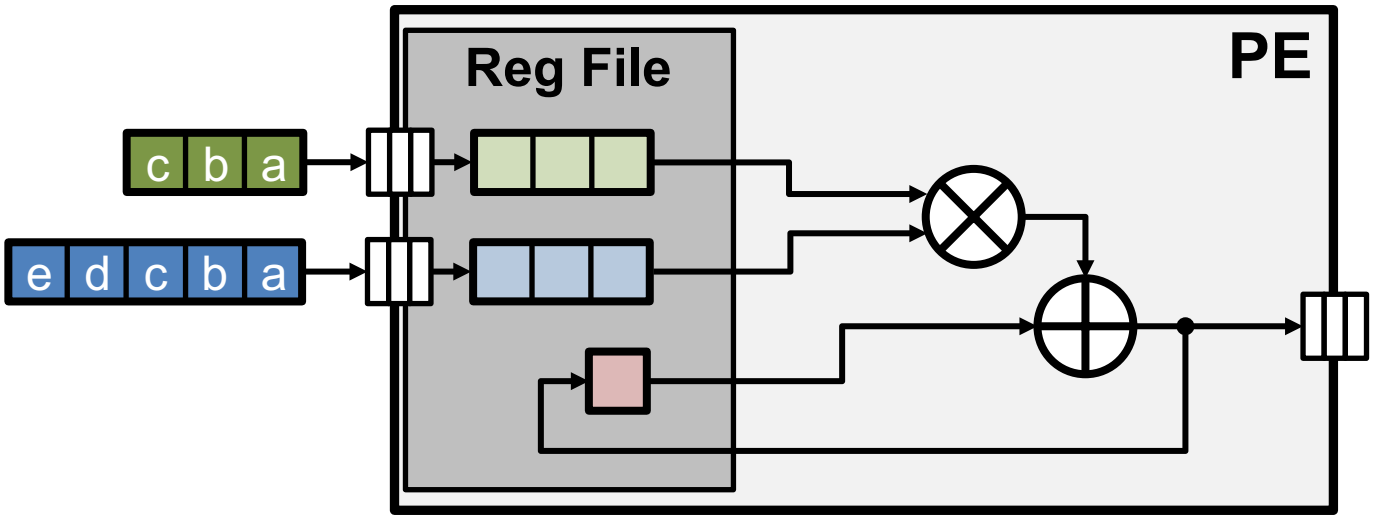
[Chen et al., ISCA 2016]

Row Stationary: Energy-efficient Dataflow

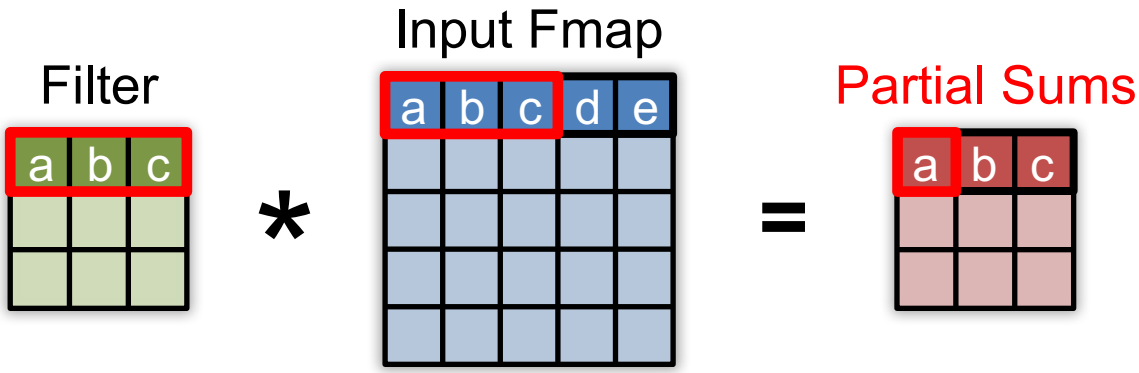


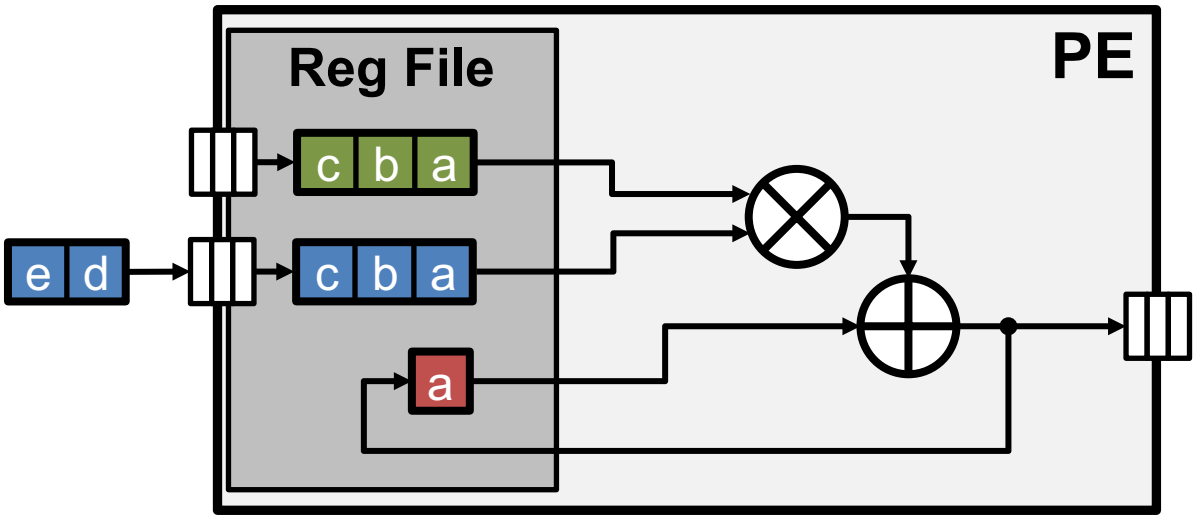
1D Row Convolution in PE



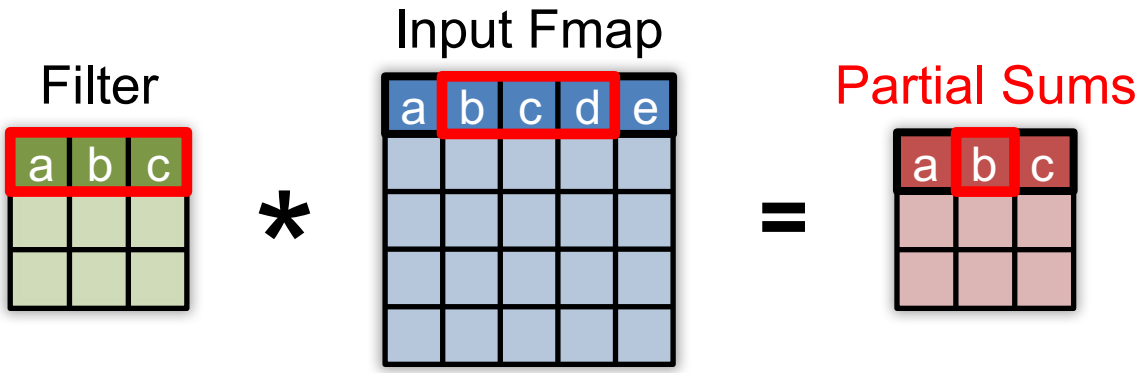


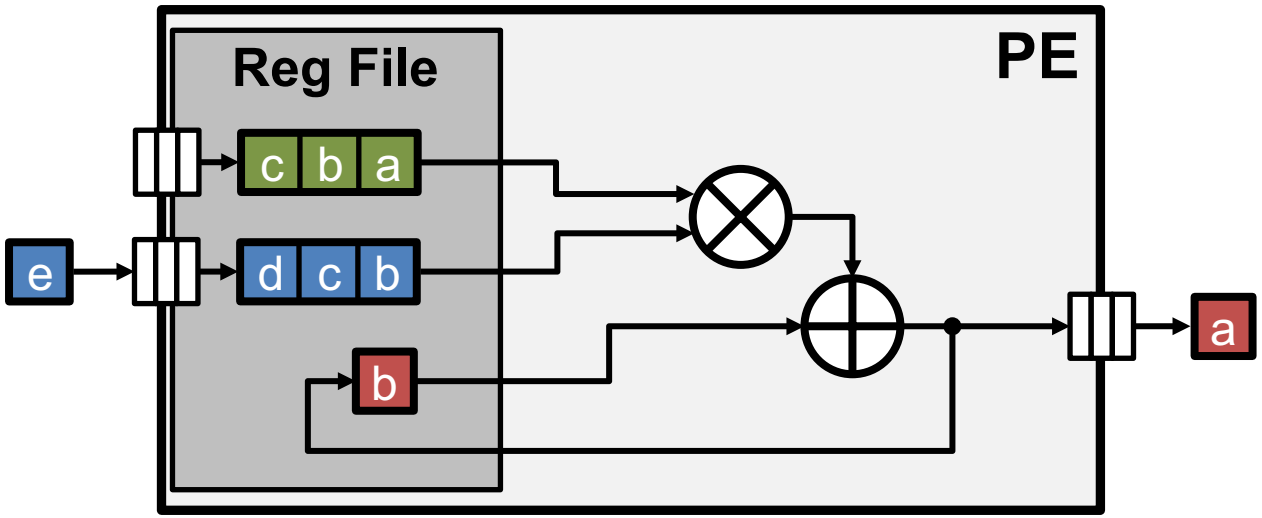
1D Row Convolution in PE



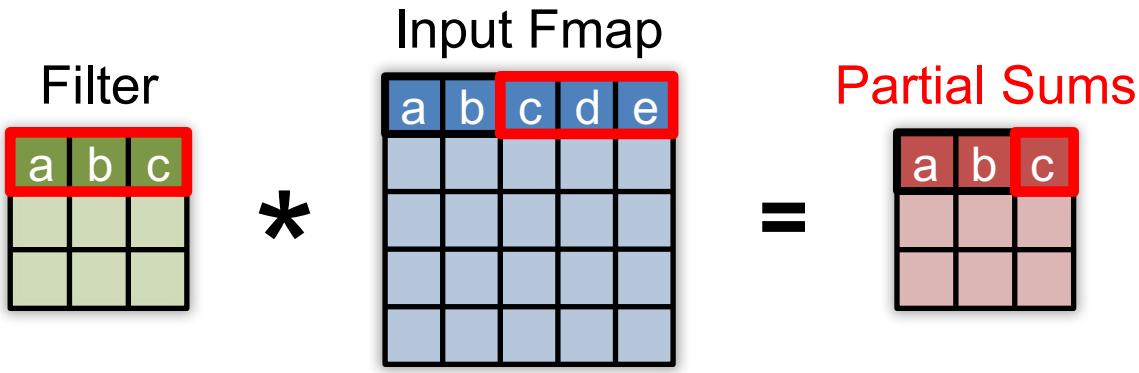


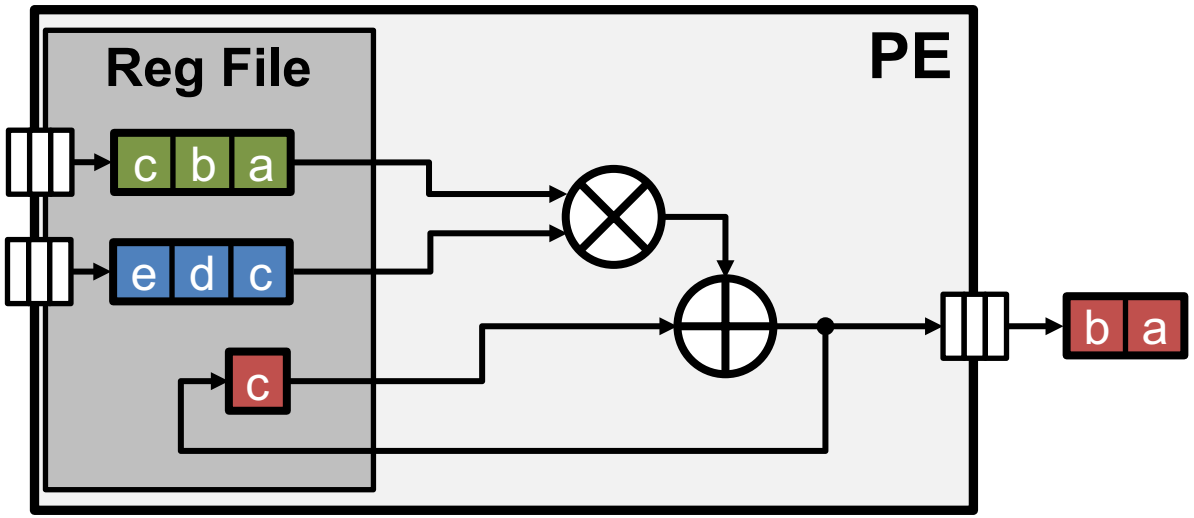
1D Row Convolution in PE





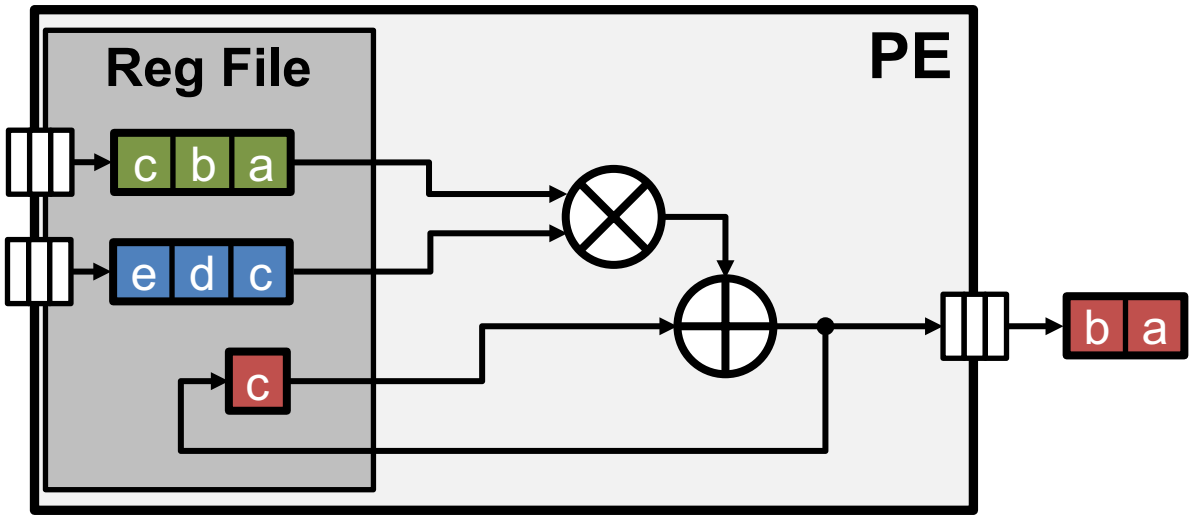
1D Row Convolution in PE



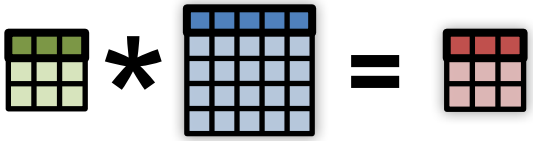


1D Row Convolution in PE

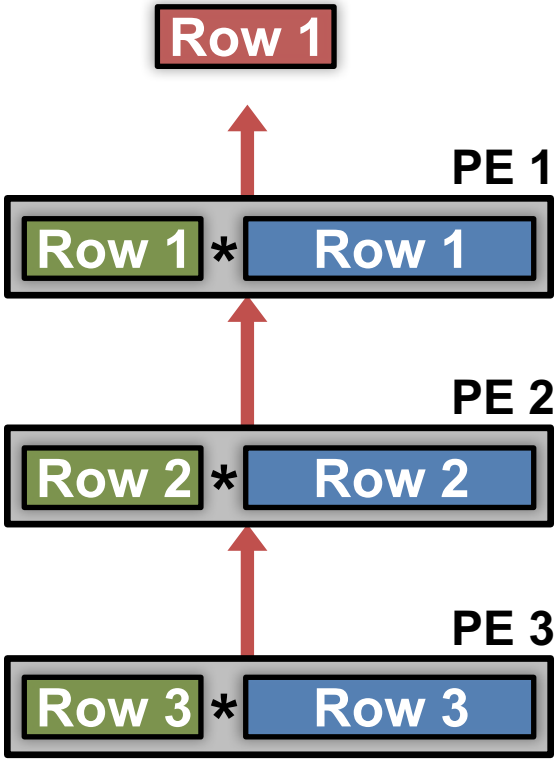
- Maximize row **convolutional reuse** in RF
  - Keep a **filter** row and **fmap** sliding window in RF
- Maximize row **psum accumulation** in RF

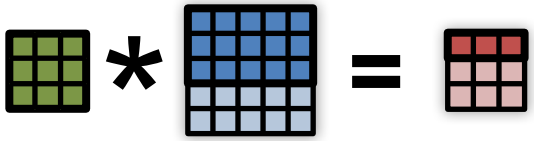


2D Convolution in PE Array

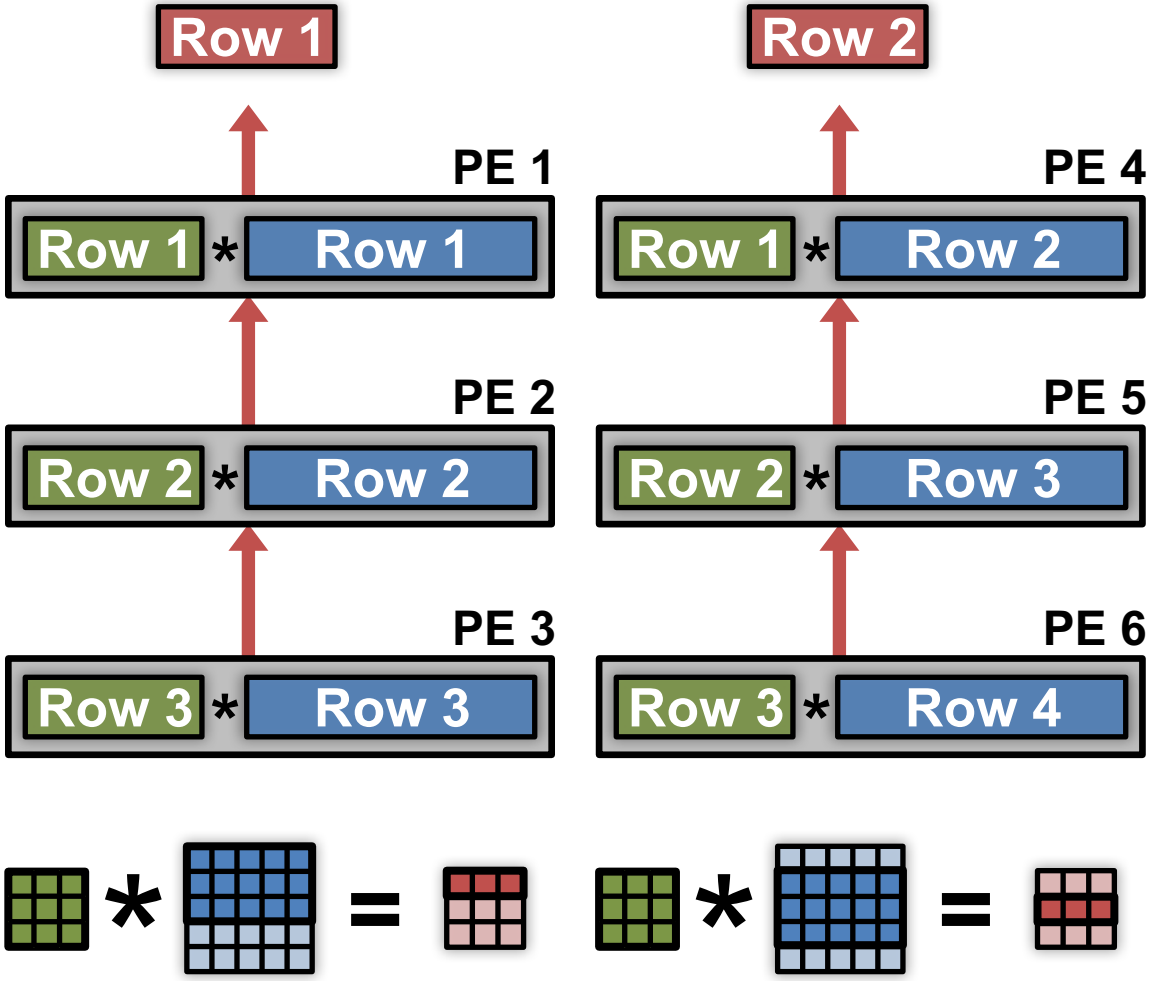


2D Convolution in PE Array

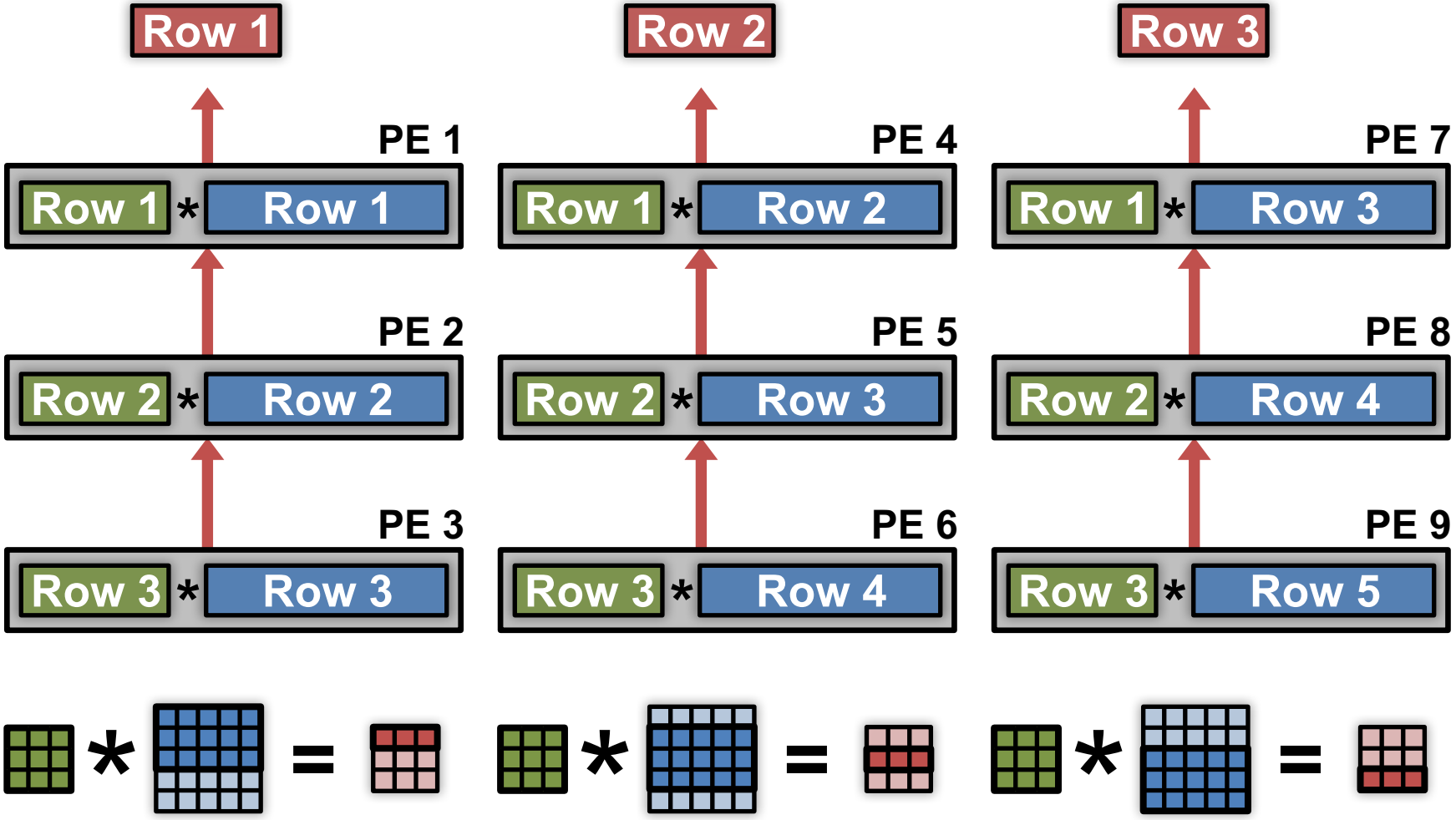




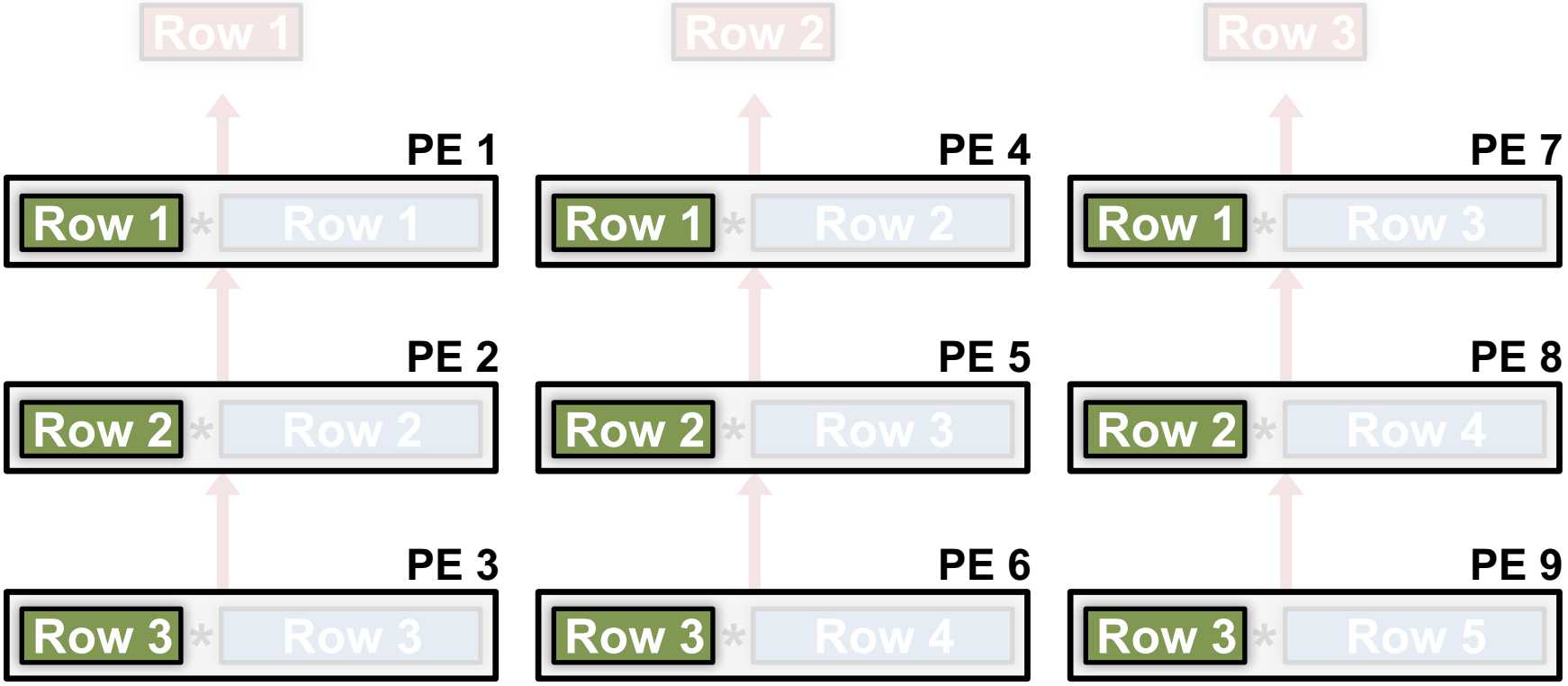
2D Convolution in PE Array



2D Convolution in PE Array

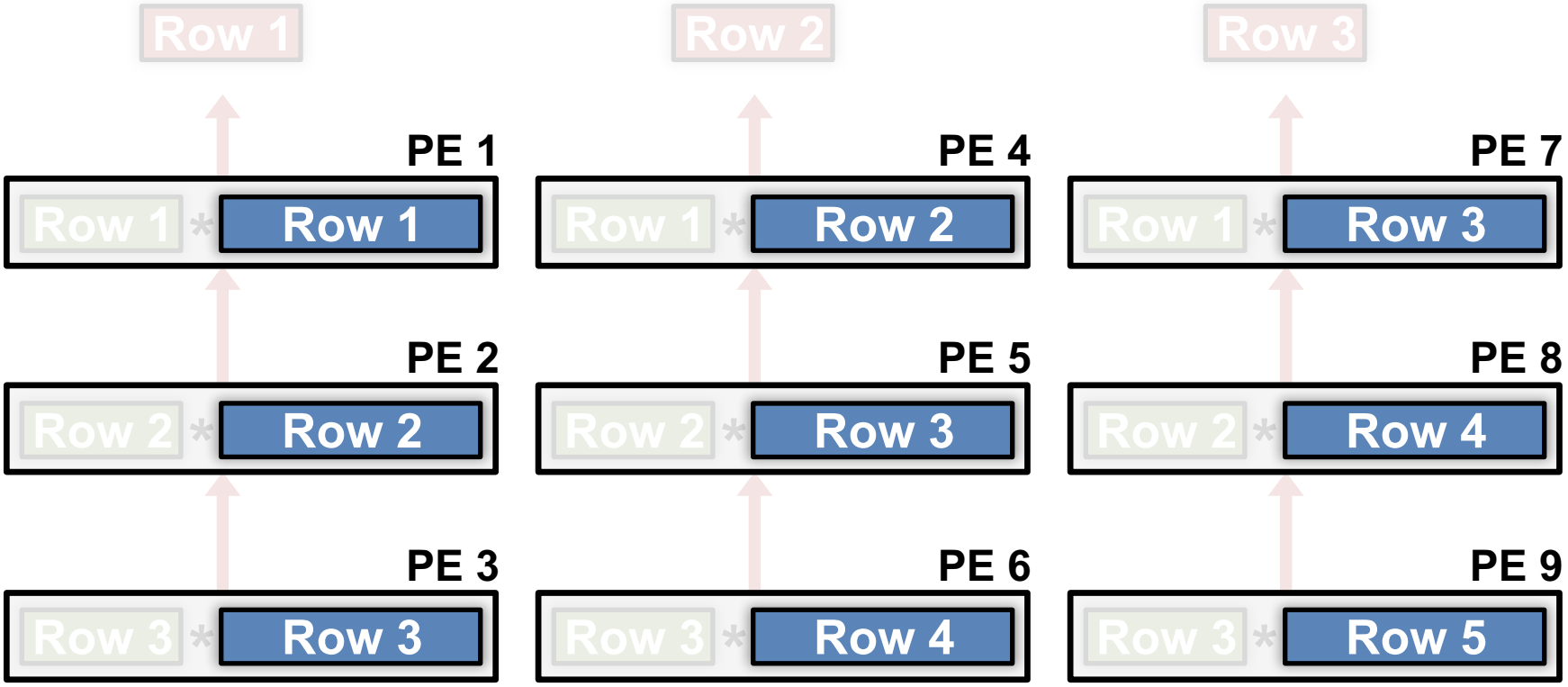


Convolutional Reuse Maximized



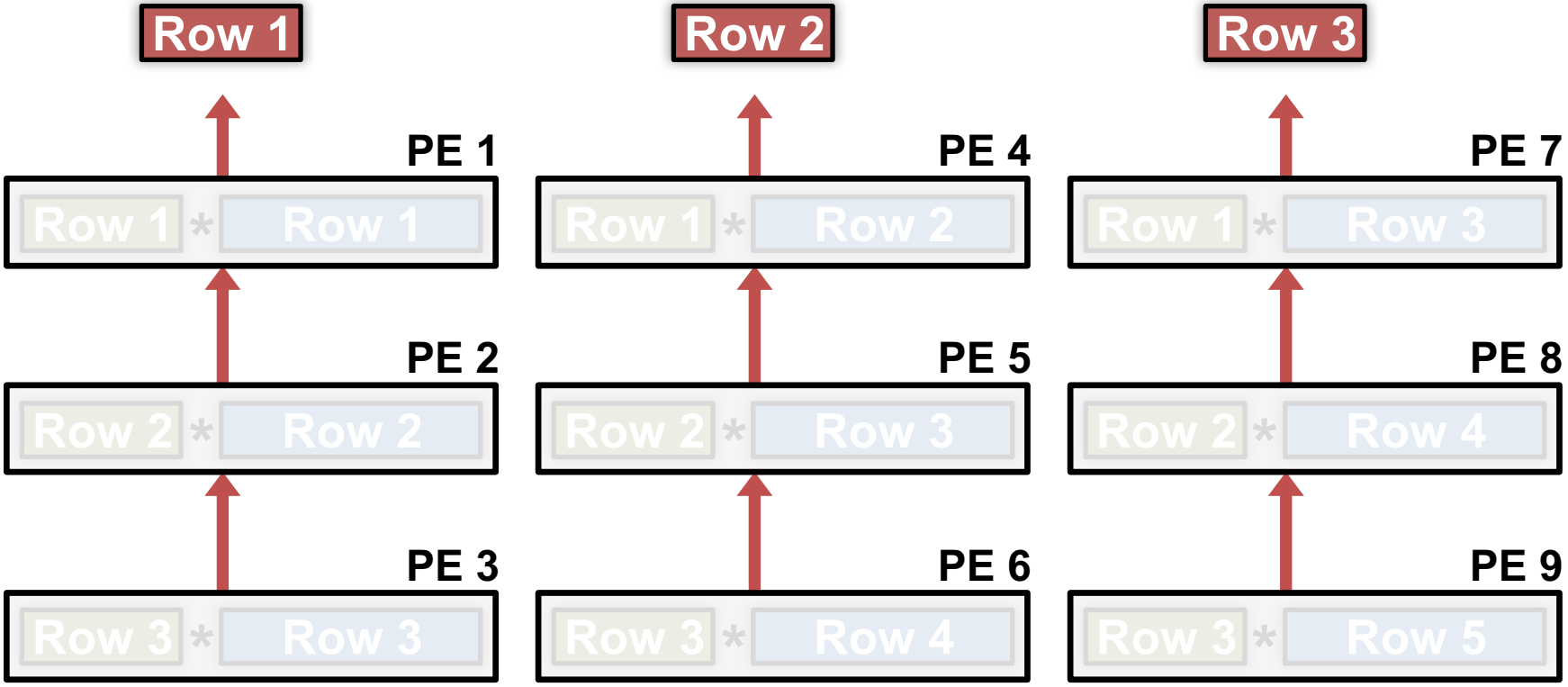
Filter rows are reused across PEs **horizontally**

Convolutional Reuse Maximized



Fmap rows are reused across PEs **diagonally**

Maximize 2D Accumulation in PE Array



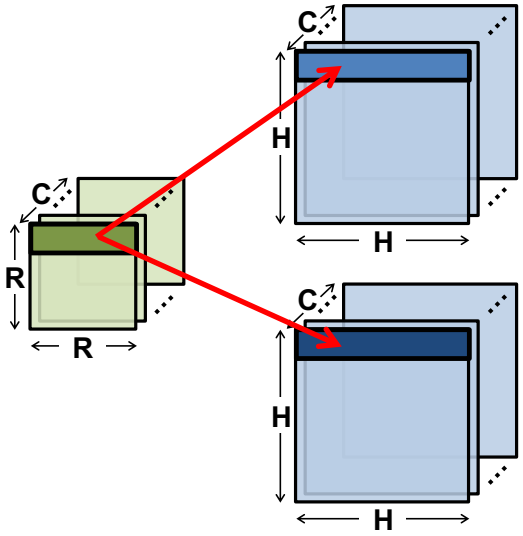
Partial sums accumulate across PEs **vertically**

Dimensions Beyond 2D Convolution

1. Multiple Fmaps
2. Multiple Filters
3. Multiple Channels

Filter Reuse in PE

1 Multiple Fmaps

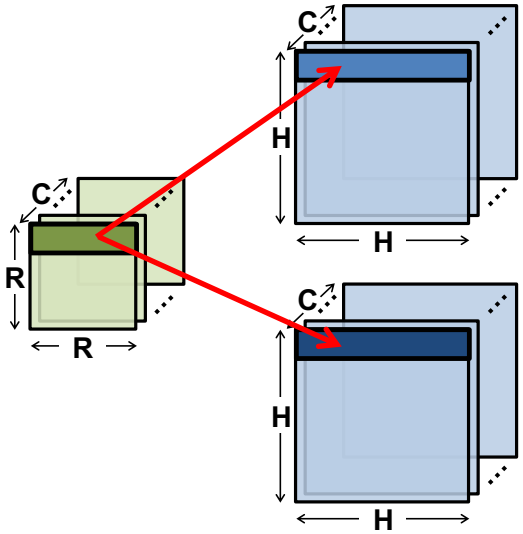


	Filter 1		Fmap 1		Psum 1
Channel 1	Row 1	*	Row 1	=	Row 1

	Filter 1		Fmap 2		Psum 2
Channel 1	Row 1	*	Row 1	=	Row 1

Filter Reuse in PE

1 Multiple Fmaps



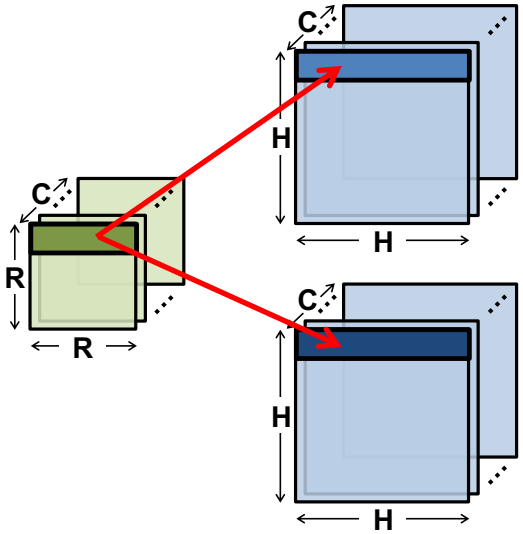
	Filter 1		Fmap 1		Psum 1
Channel 1	Row 1	*	Row 1	=	Row 1

	Filter 1		Fmap 2		Psum 2
Channel 1	Row 1	*	Row 1	=	Row 1

share the same filter row

Filter Reuse in PE

1 Multiple Fmaps



	Filter 1		Fmap 1		Psum 1
Channel 1	Row 1	*	Row 1	=	Row 1

	Filter 1		Fmap 2		Psum 2
Channel 1	Row 1	*	Row 1	=	Row 1

share the same filter row

2 Multiple Filters

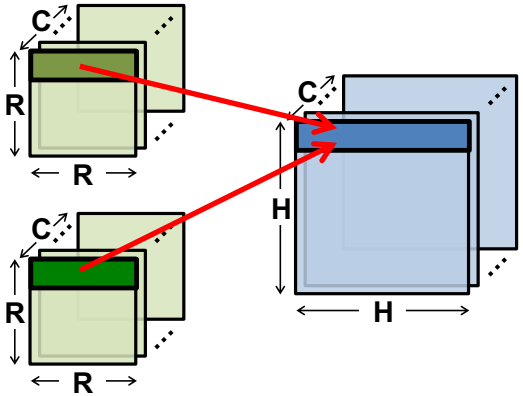
3 Multiple Channels

Processing in PE: **concatenate fmap rows**

	Filter 1		Fmap 1 & 2		Psum 1 & 2
Channel 1	Row 1	*	Row 1 Row 1	=	Row 1 Row 1

Fmap Reuse in PE

1 Multiple Fmaps **2 Multiple Filters** 3 Multiple Channels

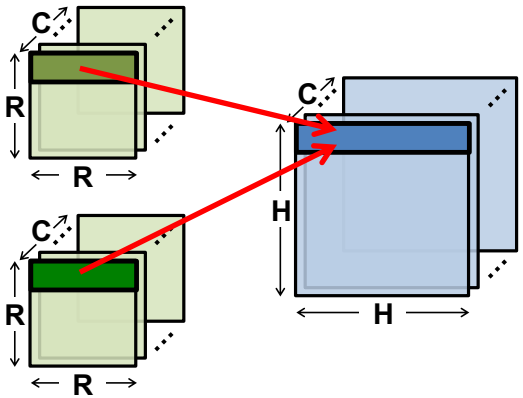


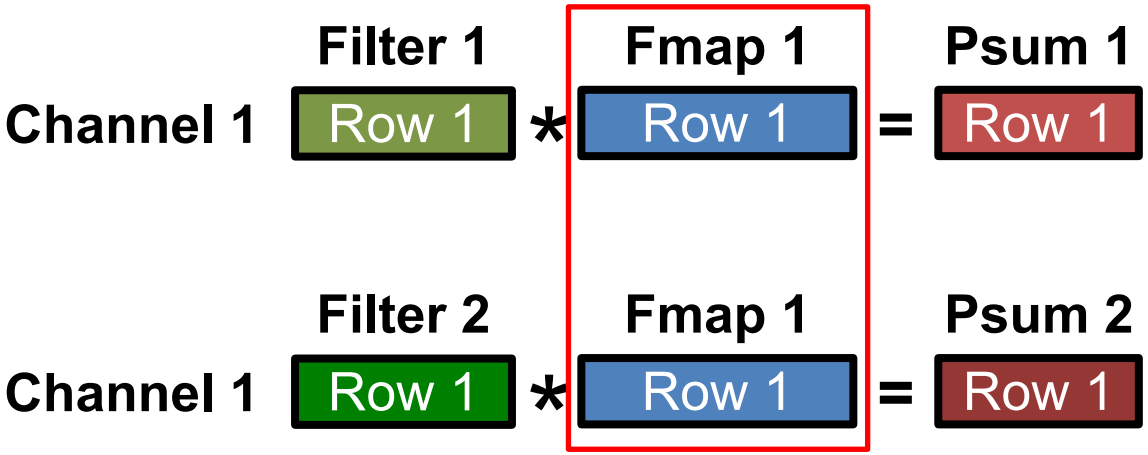
	Filter 1		Fmap 1		Psum 1
Channel 1	Row 1	*	Row 1	=	Row 1

	Filter 2		Fmap 1		Psum 2
Channel 1	Row 1	*	Row 1	=	Row 1

Fmap Reuse in PE

1 Multiple Fmaps **2 Multiple Filters** 3 Multiple Channels

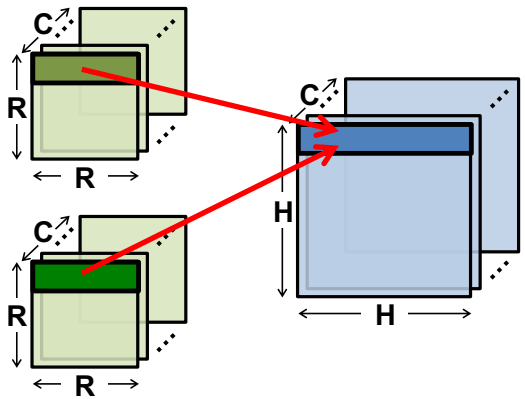


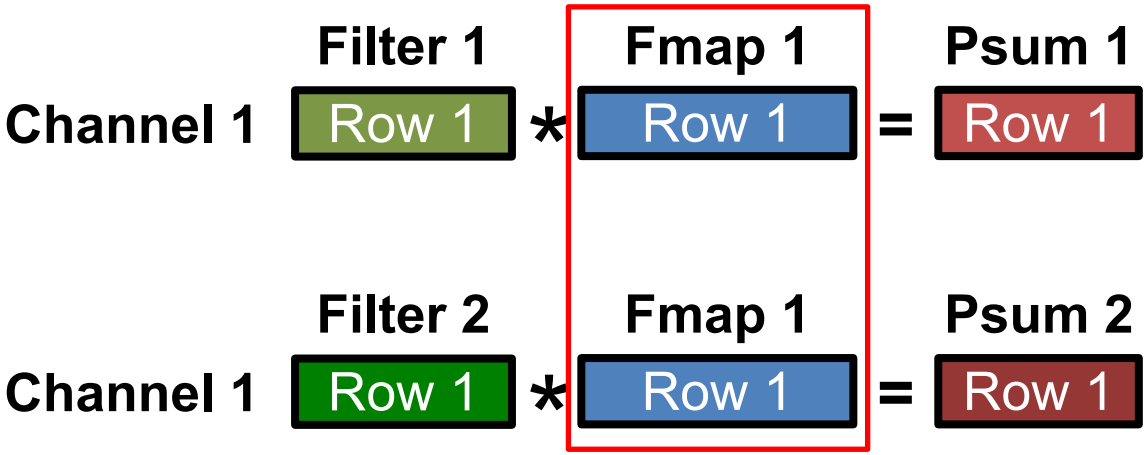


share the same fmap row

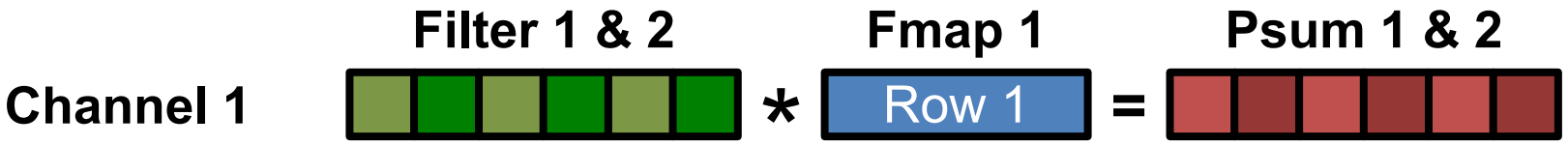
Fmap Reuse in PE

1 Multiple Fmaps **2 Multiple Filters** 3 Multiple Channels



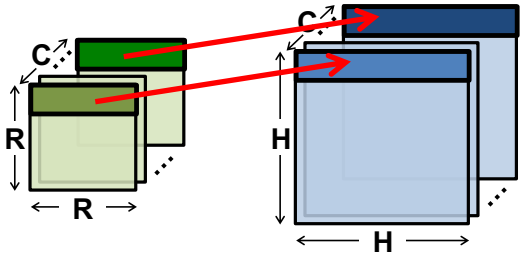


Processing in PE: **interleave filter rows**



Channel Accumulation in PE

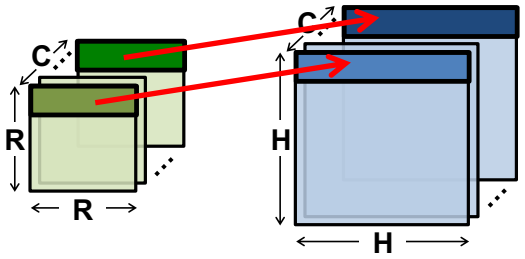
1 Multiple Fmaps 2 Multiple Filters **3 Multiple Channels**

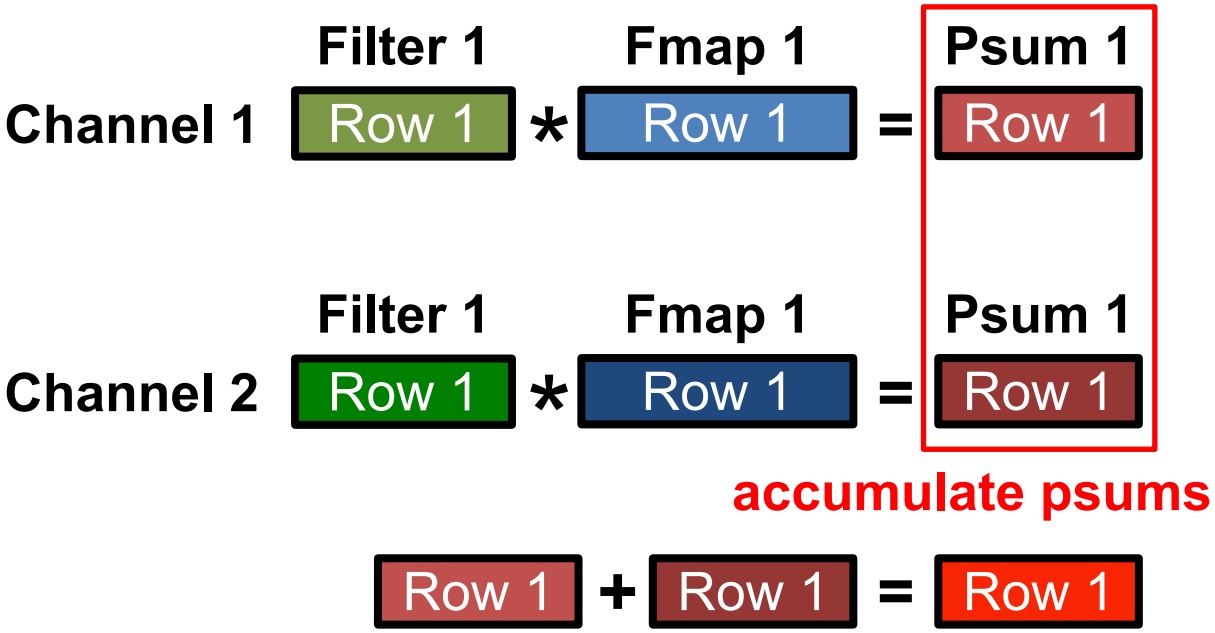


	Filter 1		Fmap 1		Psum 1
Channel 1	Row 1	*	Row 1	=	Row 1
Channel 2	Row 1	*	Row 1	=	Row 1

Channel Accumulation in PE

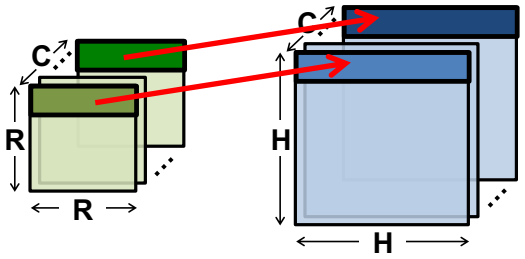
1 Multiple Fmaps 2 Multiple Filters **3 Multiple Channels**

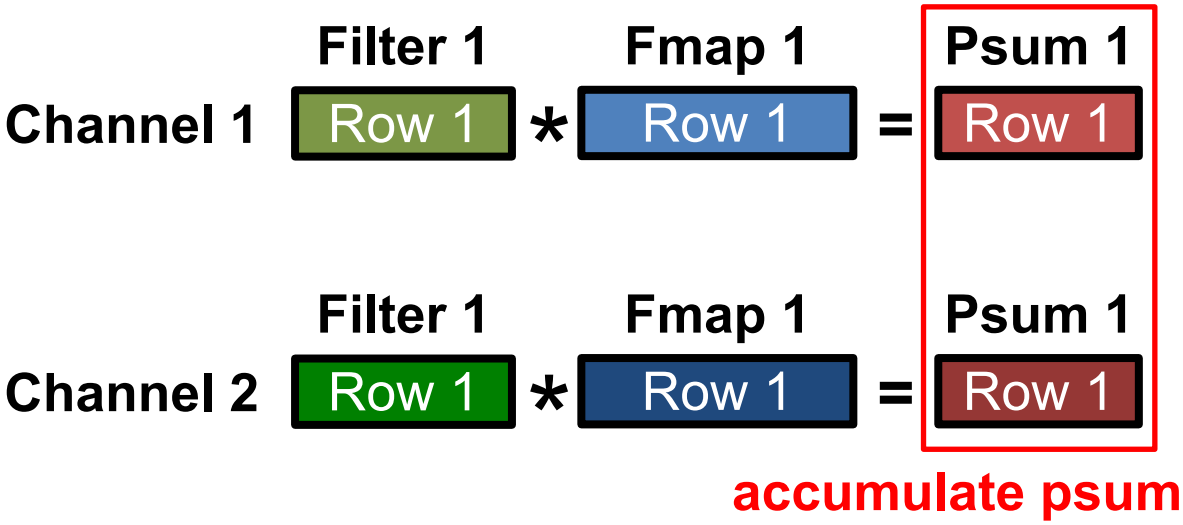




Channel Accumulation in PE

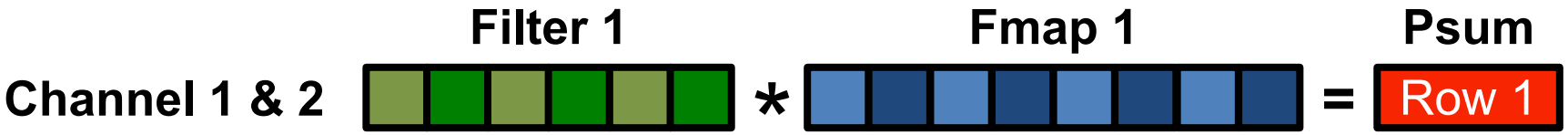
1 Multiple Fmaps 2 Multiple Filters **3 Multiple Channels**



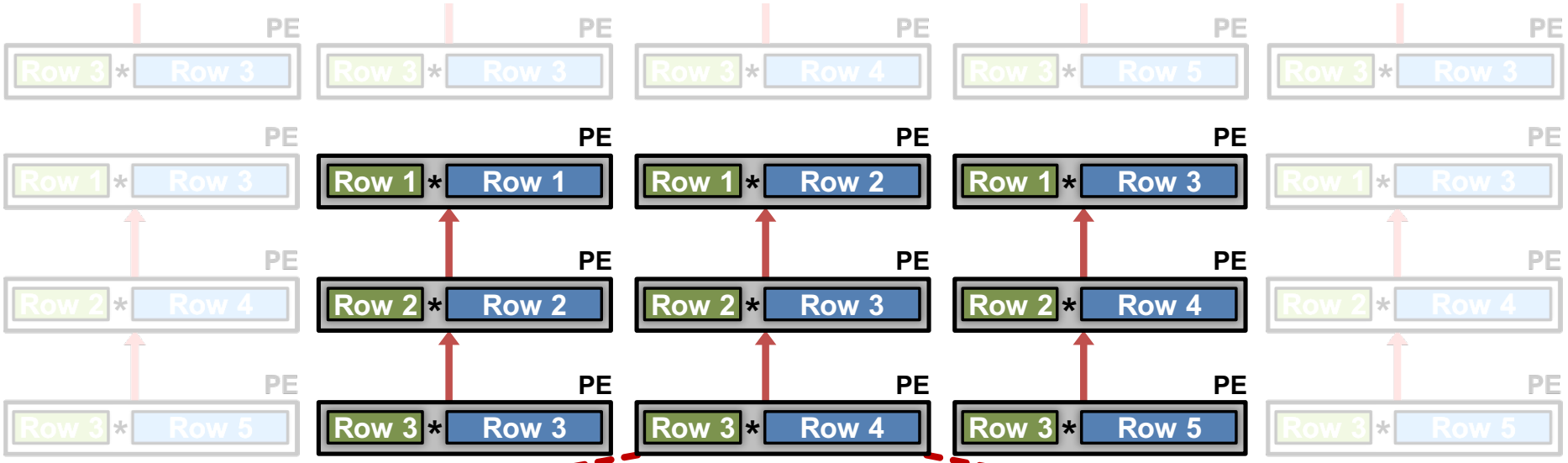


accumulate psums

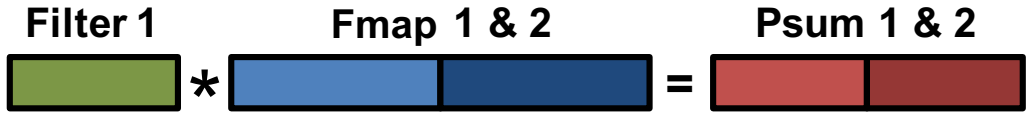
Processing in PE: **interleave channels**



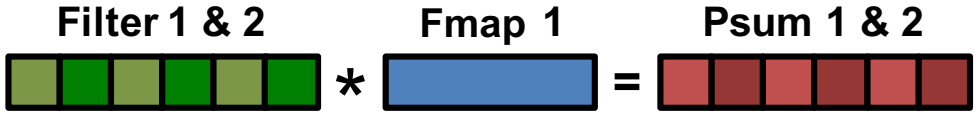
DNN Processing – The Full Picture



Multiple **fmaps**:



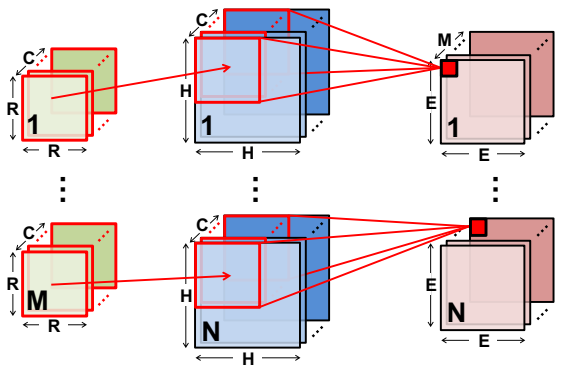
Multiple **filters**:



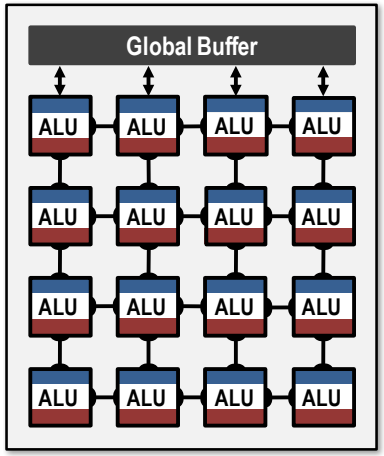
Multiple **channels**:

Optimal Mapping in Row Stationary

CNN Configurations

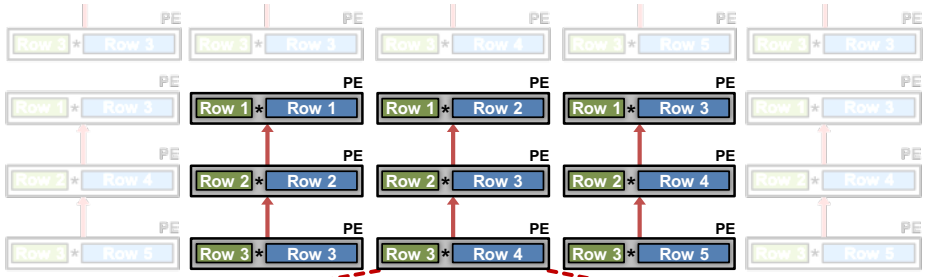


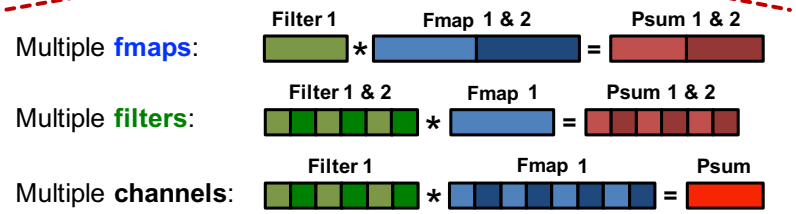
Hardware Resources



Optimization Compiler

Row Stationary Mapping





[Chen et al., ISCA 2016]

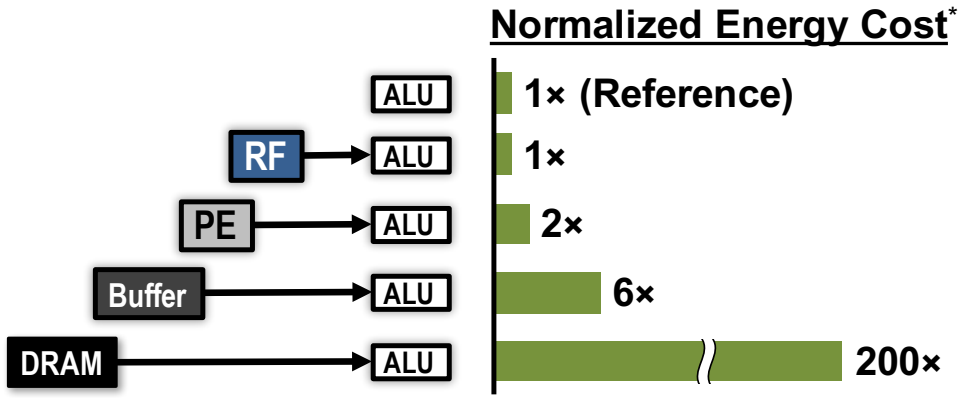
Dataflow Simulation Results

Evaluate Reuse in Different Dataflows

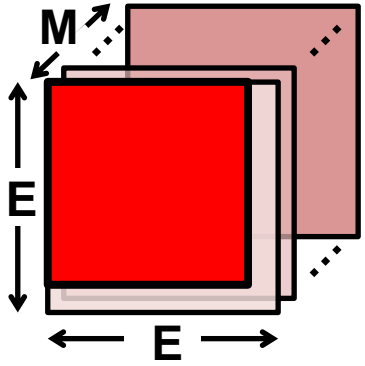
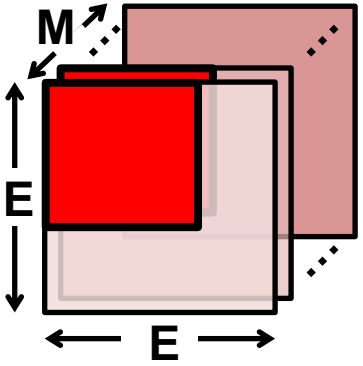
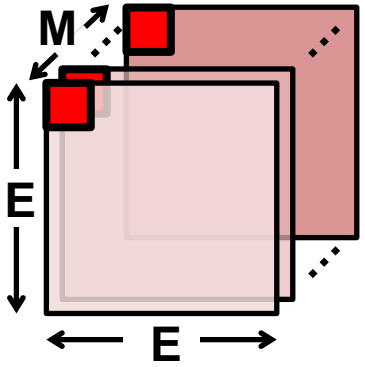
- **Weight Stationary**
  - Minimize movement of filter weights
- **Output Stationary**
  - Minimize movement of partial sums
- **No Local Reuse**
  - No PE local storage. Maximize global buffer size.
- **Row Stationary**

Evaluation Setup

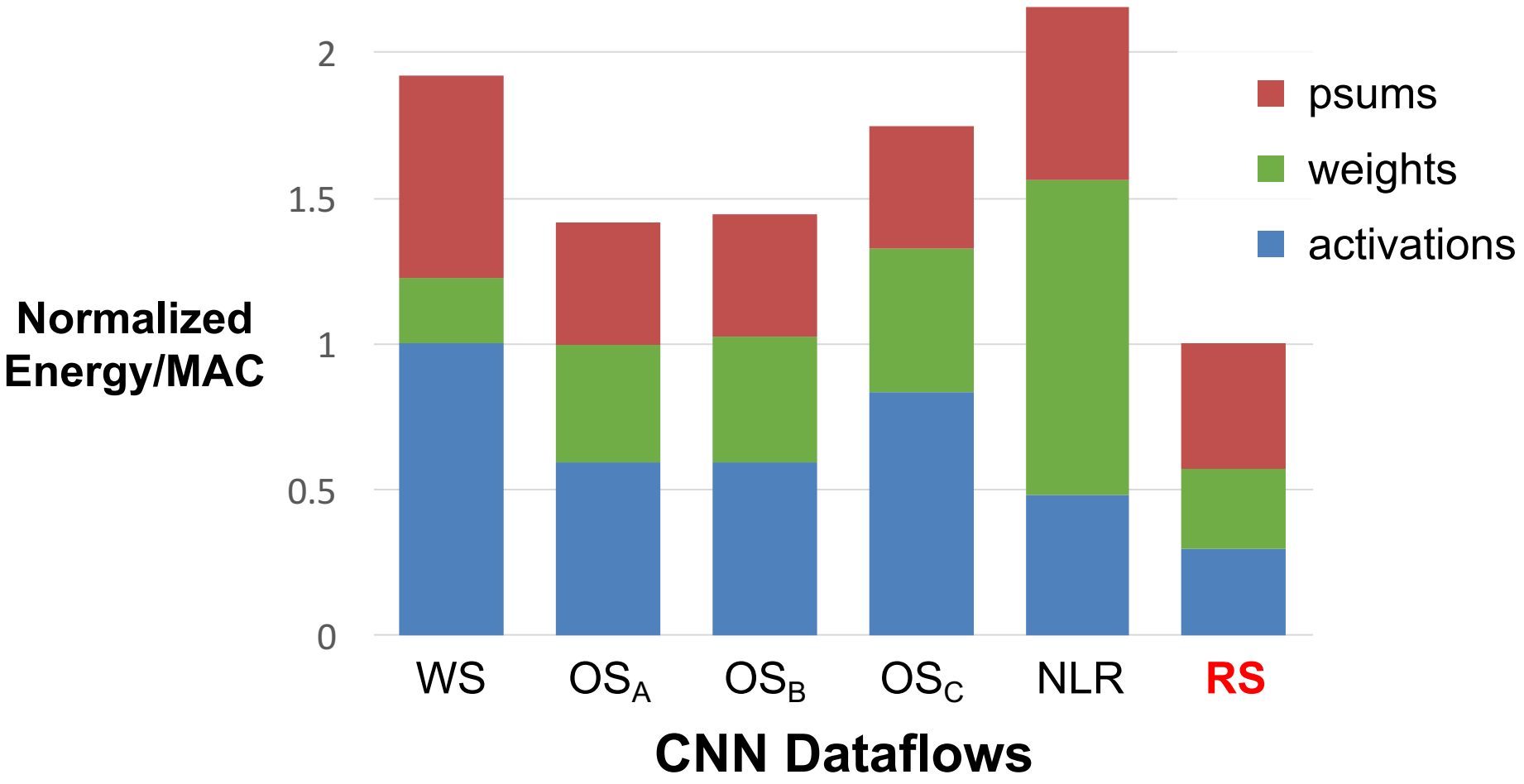
- same total area
- 256 PEs
- AlexNet
- batch size = 16



Variants of Output Stationary

	OS_A	OS_B	OS_C
Parallel Output Region			
# Output Channels	Single	Multiple	Multiple
# Output Activations	Multiple	Multiple	Single
Notes	Targeting **CONV** layers		Targeting **FC** layers

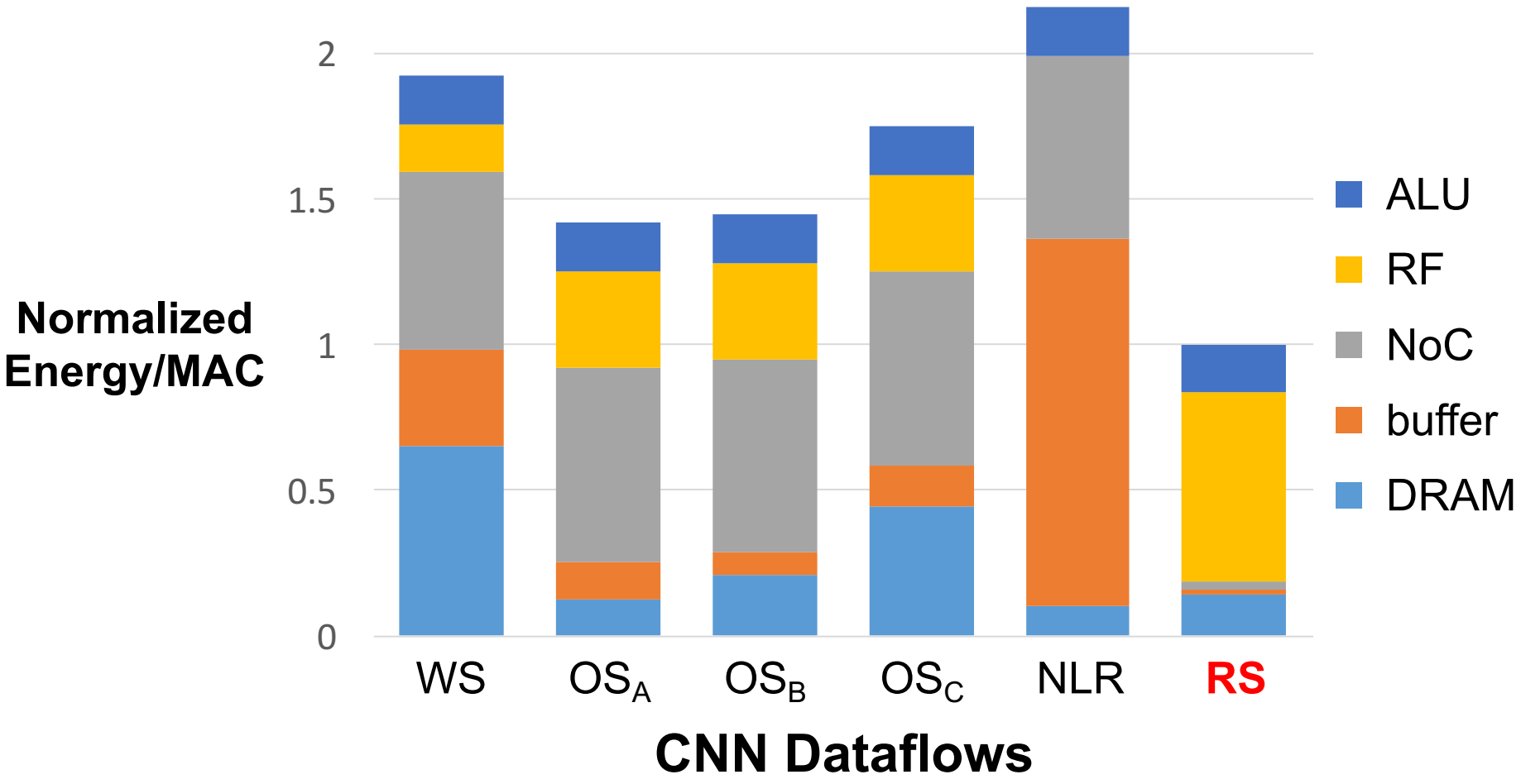
Dataflow Comparison: CONV Layers



RS optimizes for the best **overall** energy efficiency

[Chen et al., ISCA 2016]

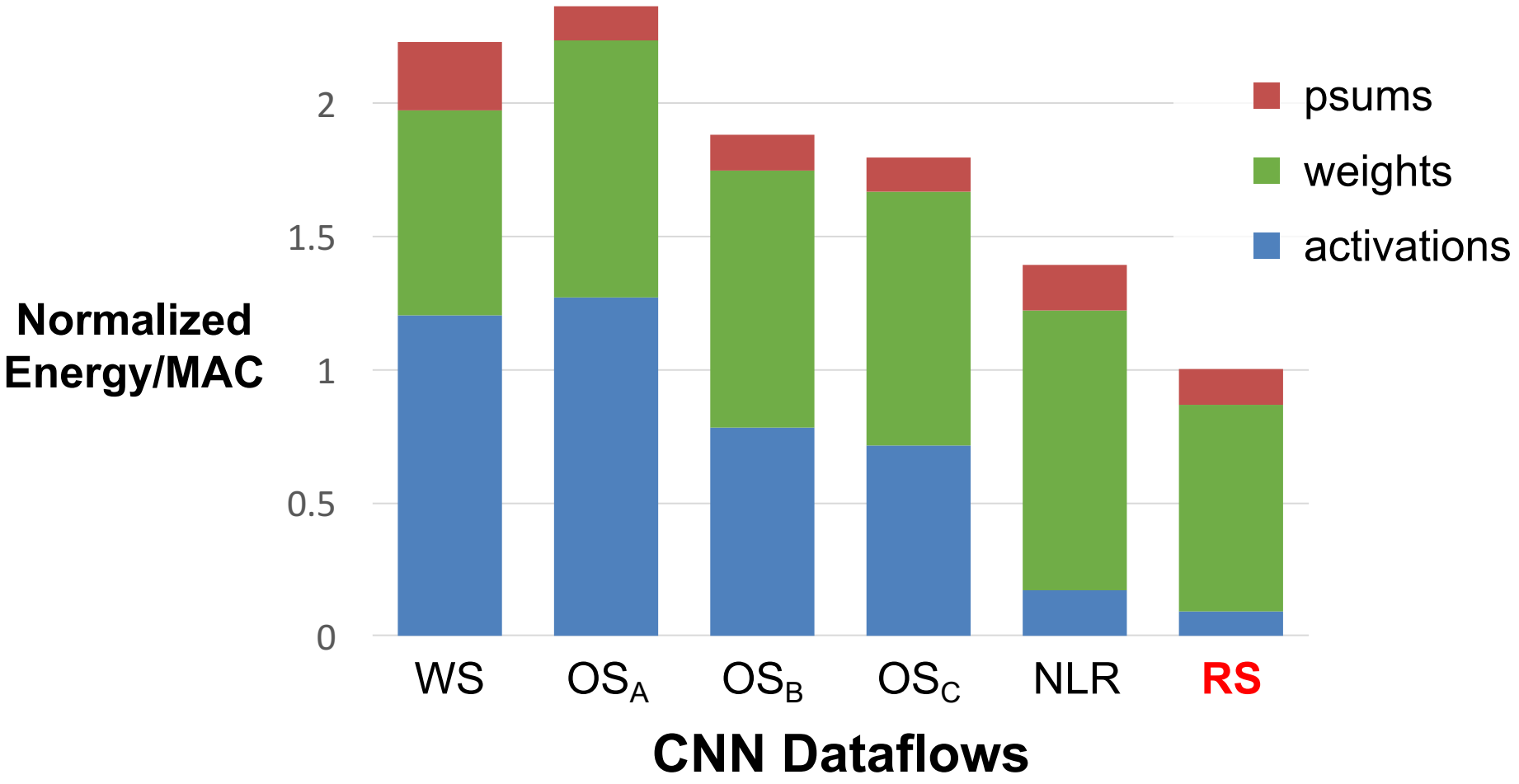
Dataflow Comparison: CONV Layers



RS uses **1.4× – 2.5× lower** energy than other dataflows

[Chen et al., ISCA 2016]

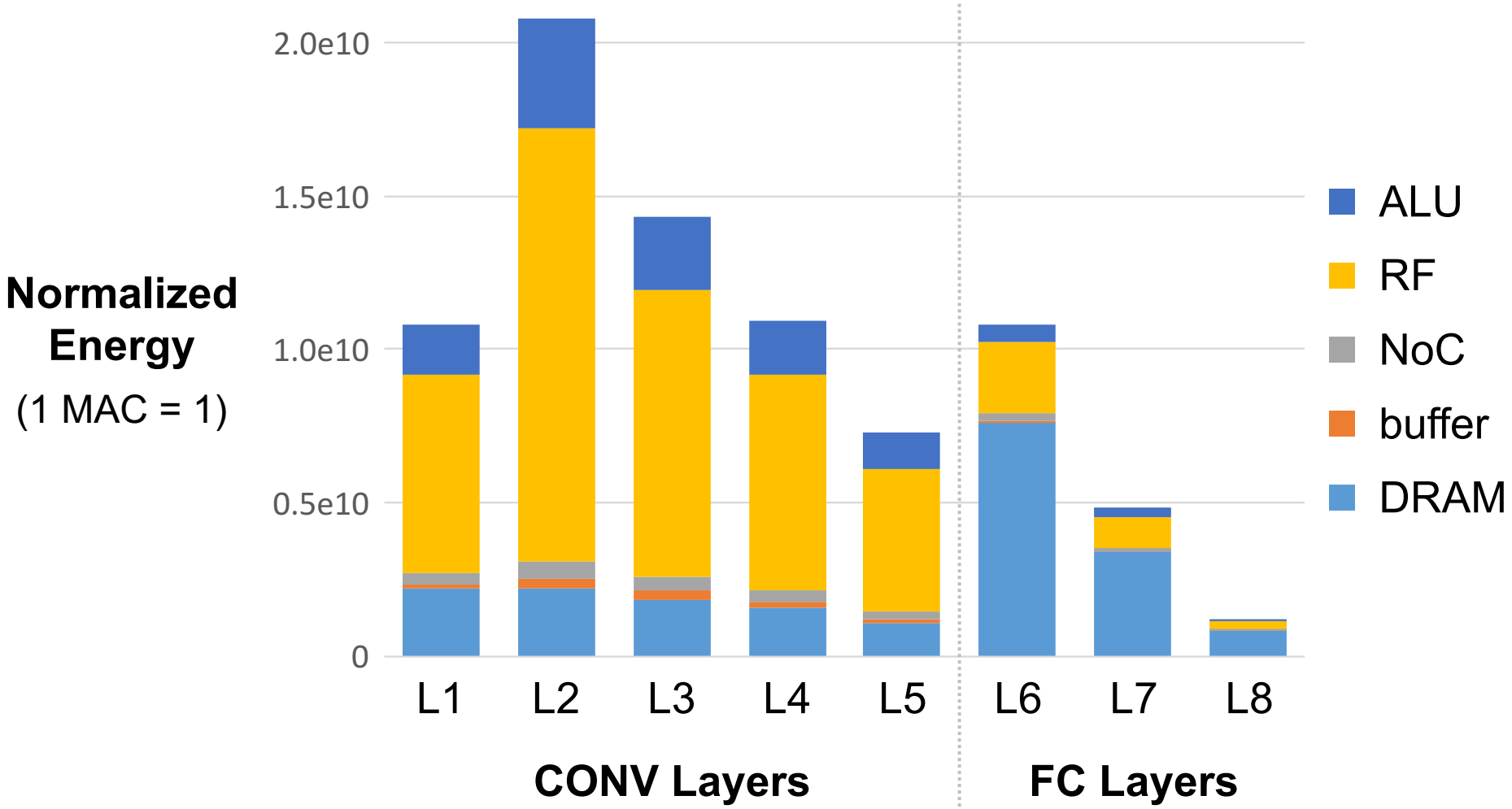
Dataflow Comparison: FC Layers



RS uses at least **1.3× lower** energy than other dataflows

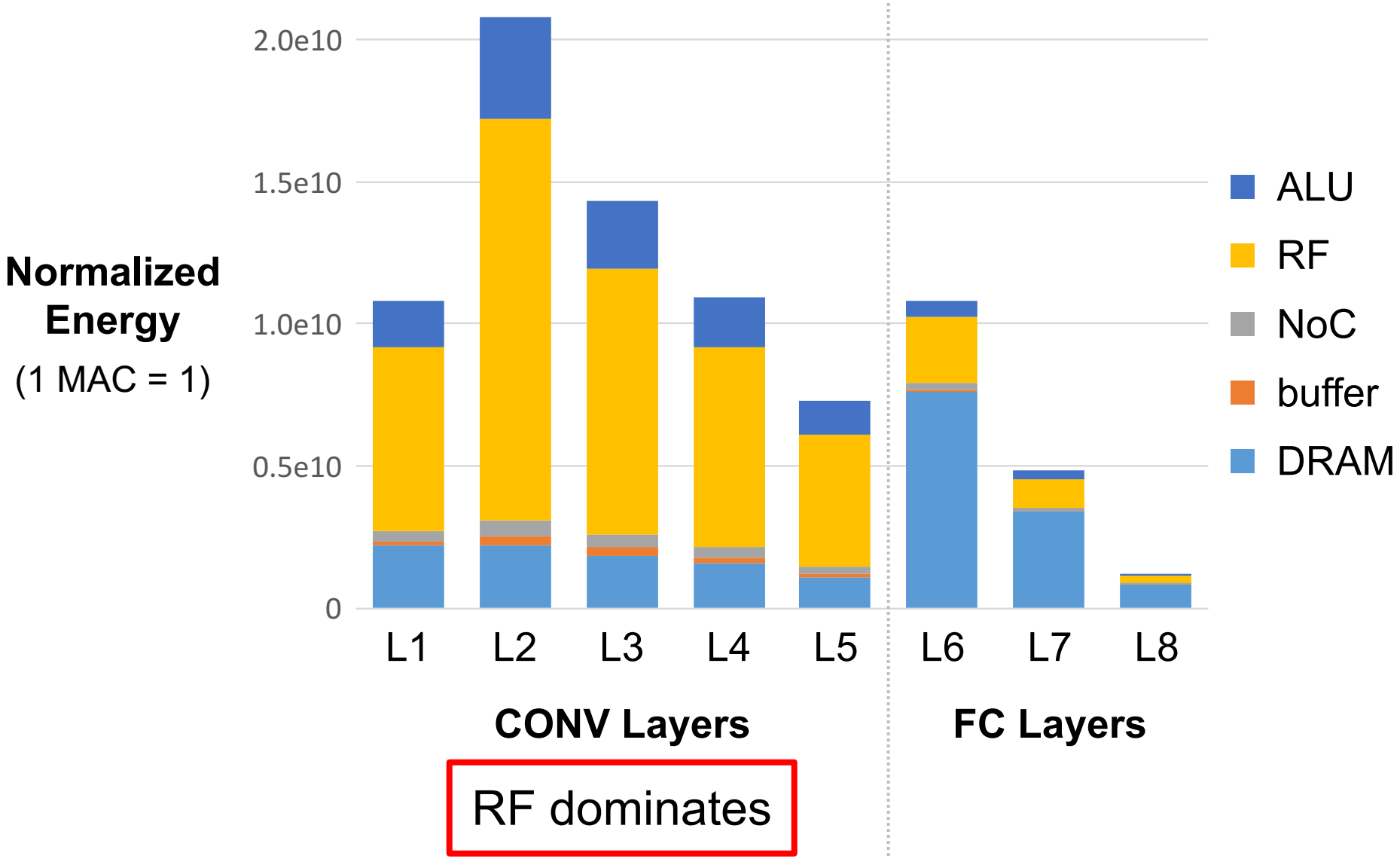
[Chen et al., ISCA 2016]

Row Stationary: Layer Breakdown

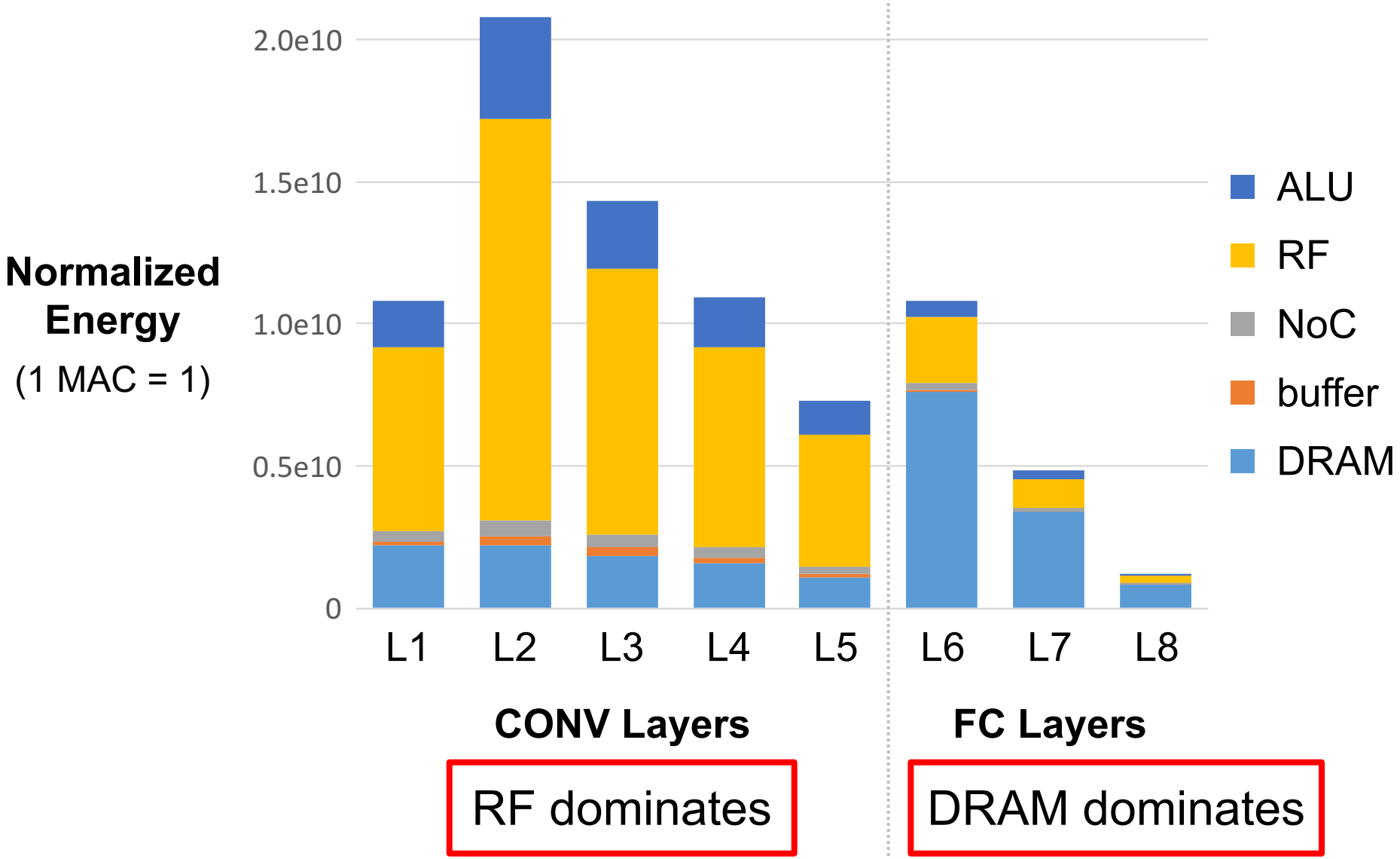


[Chen et al., ISCA 2016]

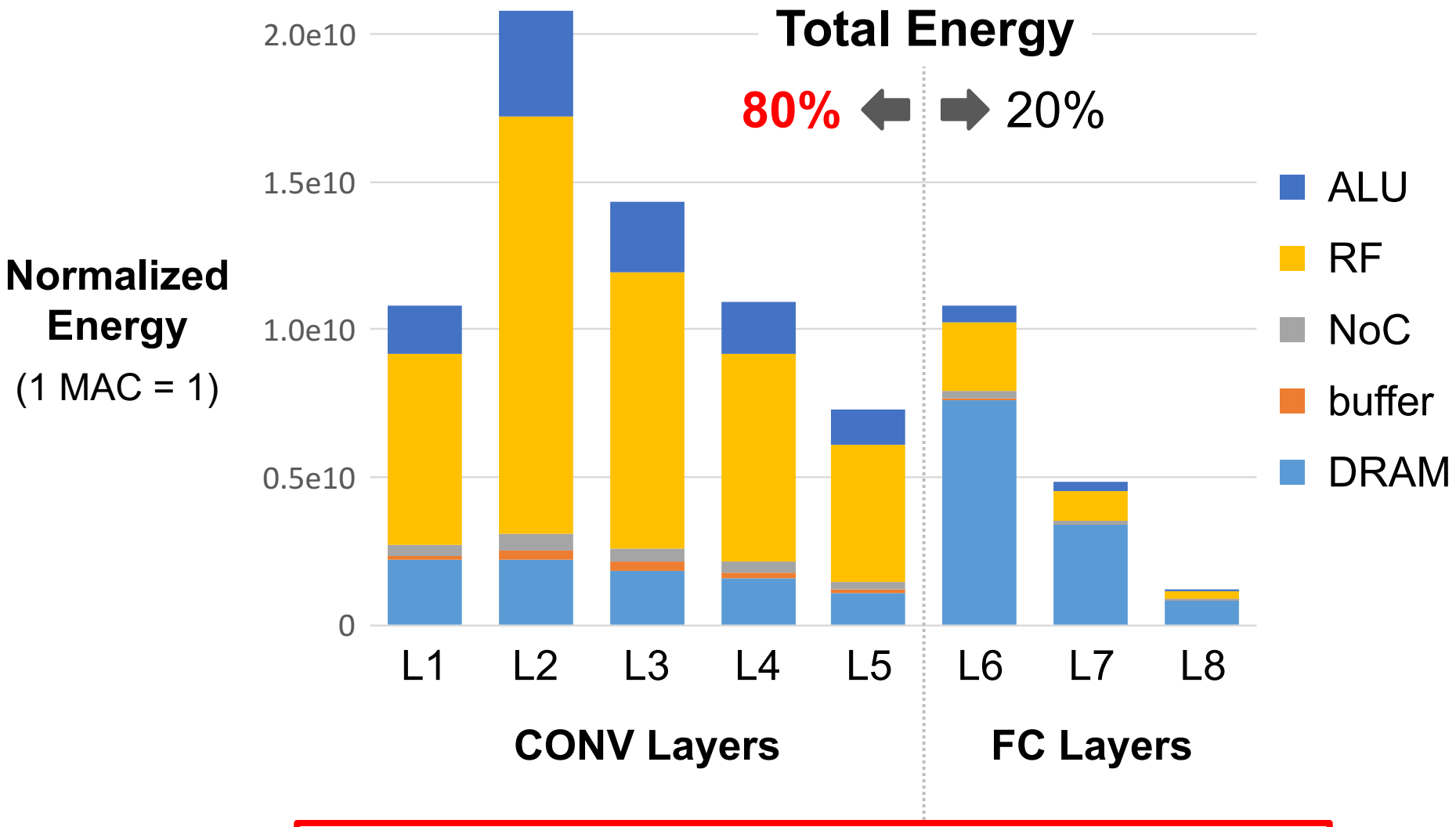
Row Stationary: Layer Breakdown



Row Stationary: Layer Breakdown



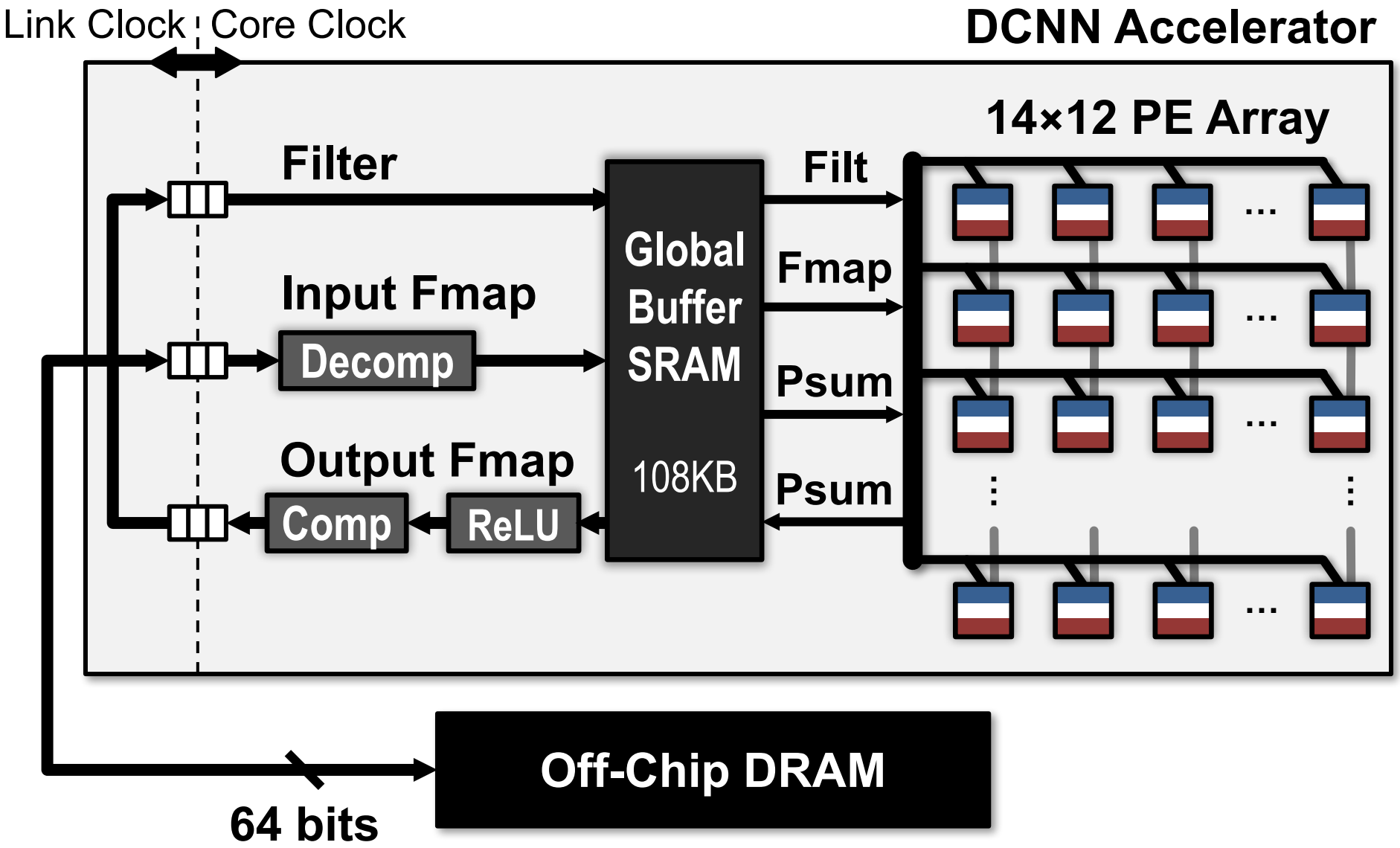
Row Stationary: Layer Breakdown



CONV layers dominate energy consumption!

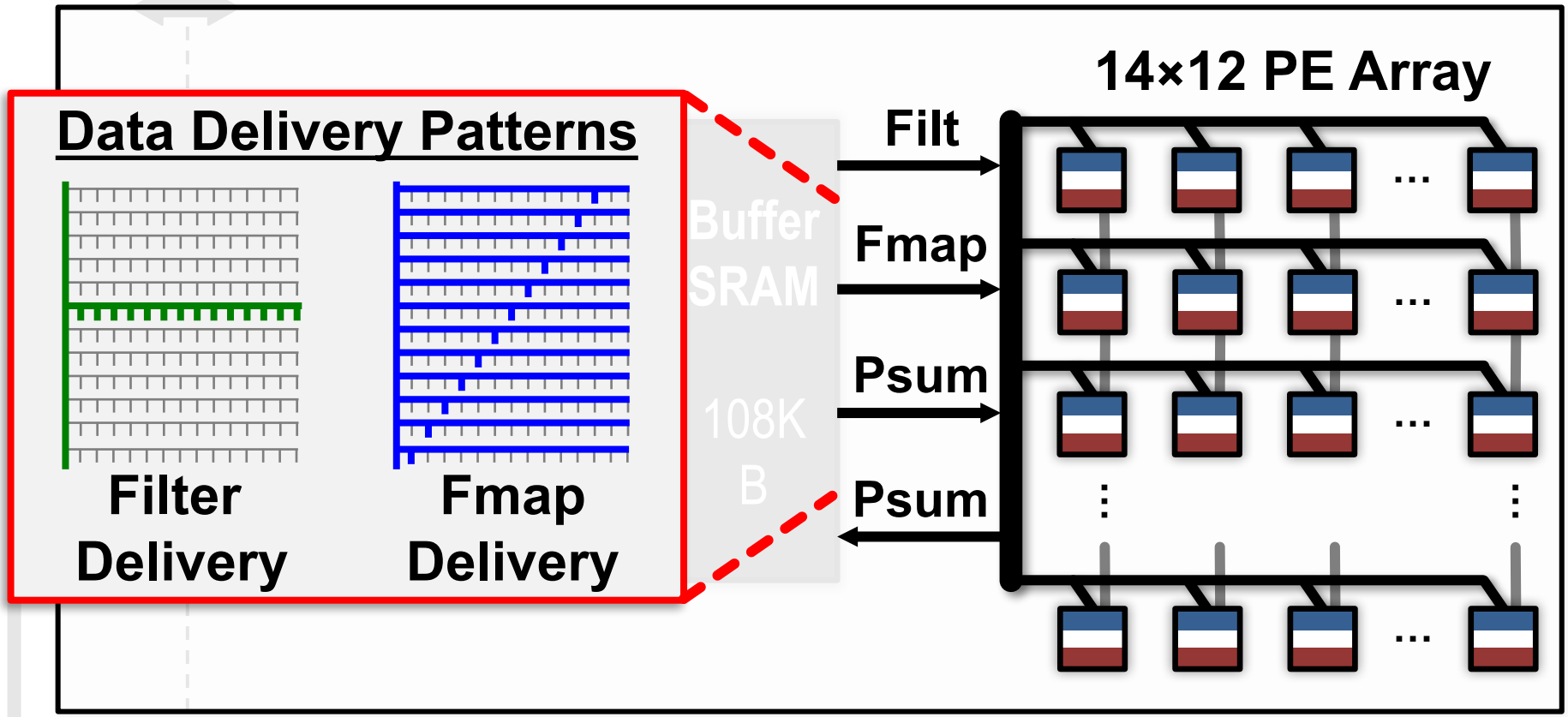
Hardware Architecture for RS Dataflow

Eyeriss Deep CNN Accelerator



[Chen et al., ISSCC 2016]

Data Delivery with On-Chip Network

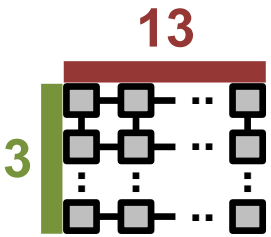


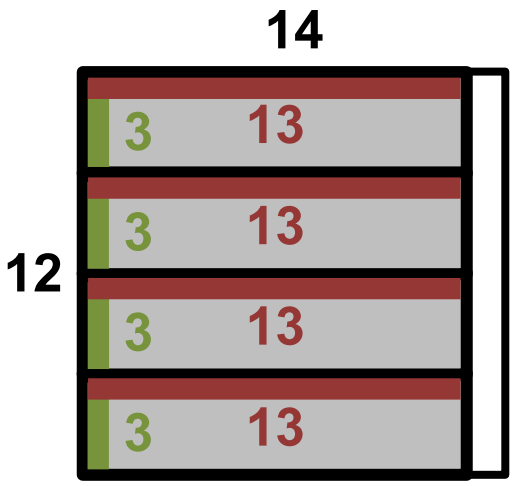
How to accommodate different shapes with fixed PE array?

Logical to Physical Mappings

Replication

AlexNet
Layer 3-5

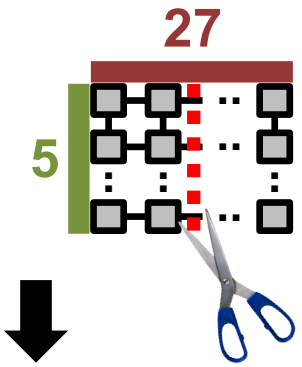


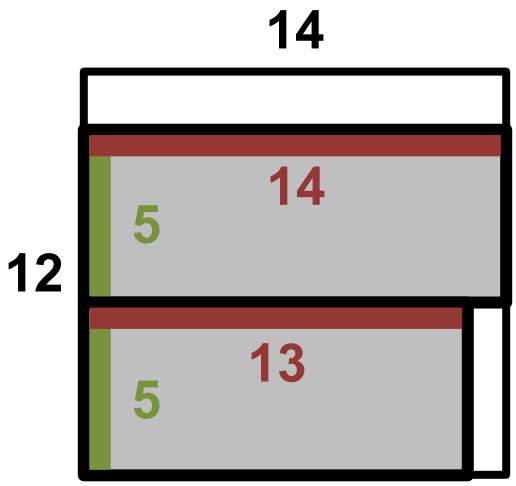


Physical PE Array

Folding

AlexNet
Layer 2



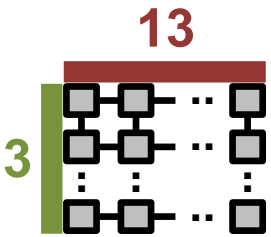


Physical PE Array

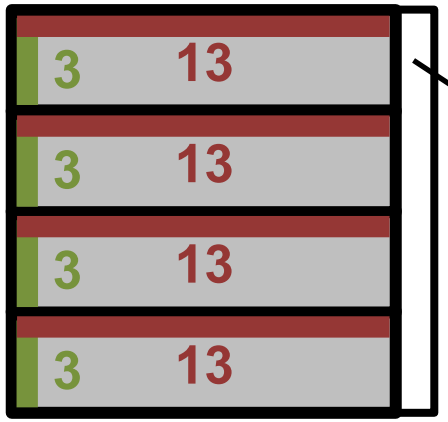
Logical to Physical Mappings

Replication

AlexNet
Layer 3-5



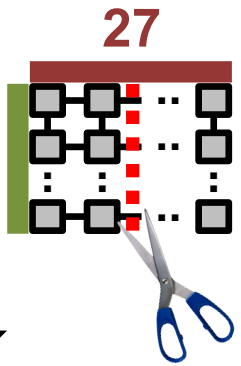
14



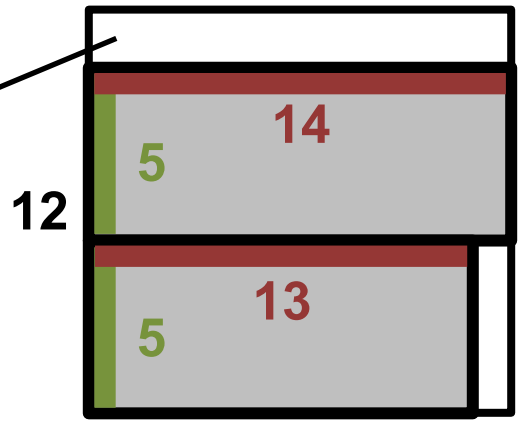
Physical PE Array

Folding

AlexNet
Layer 2



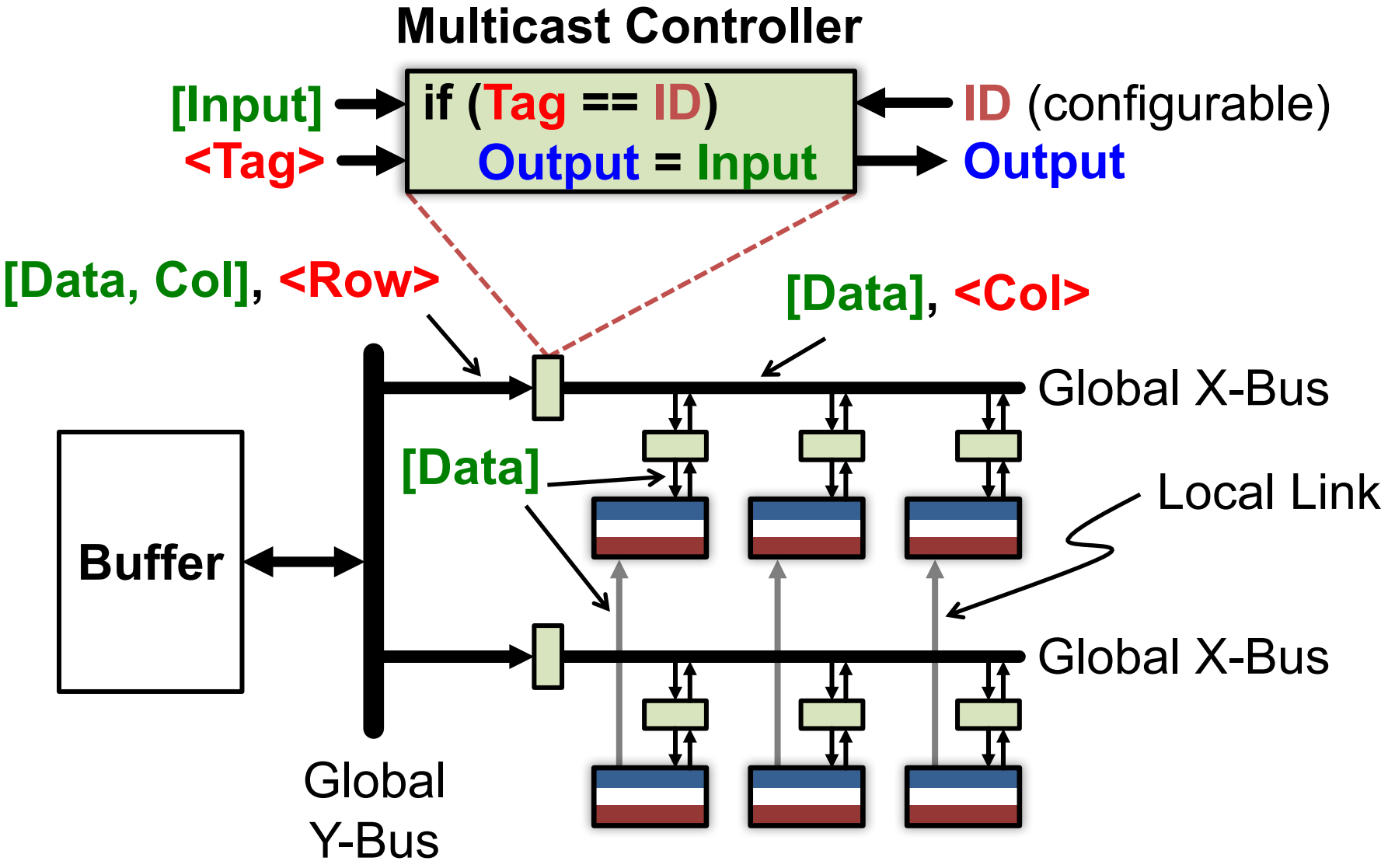
14



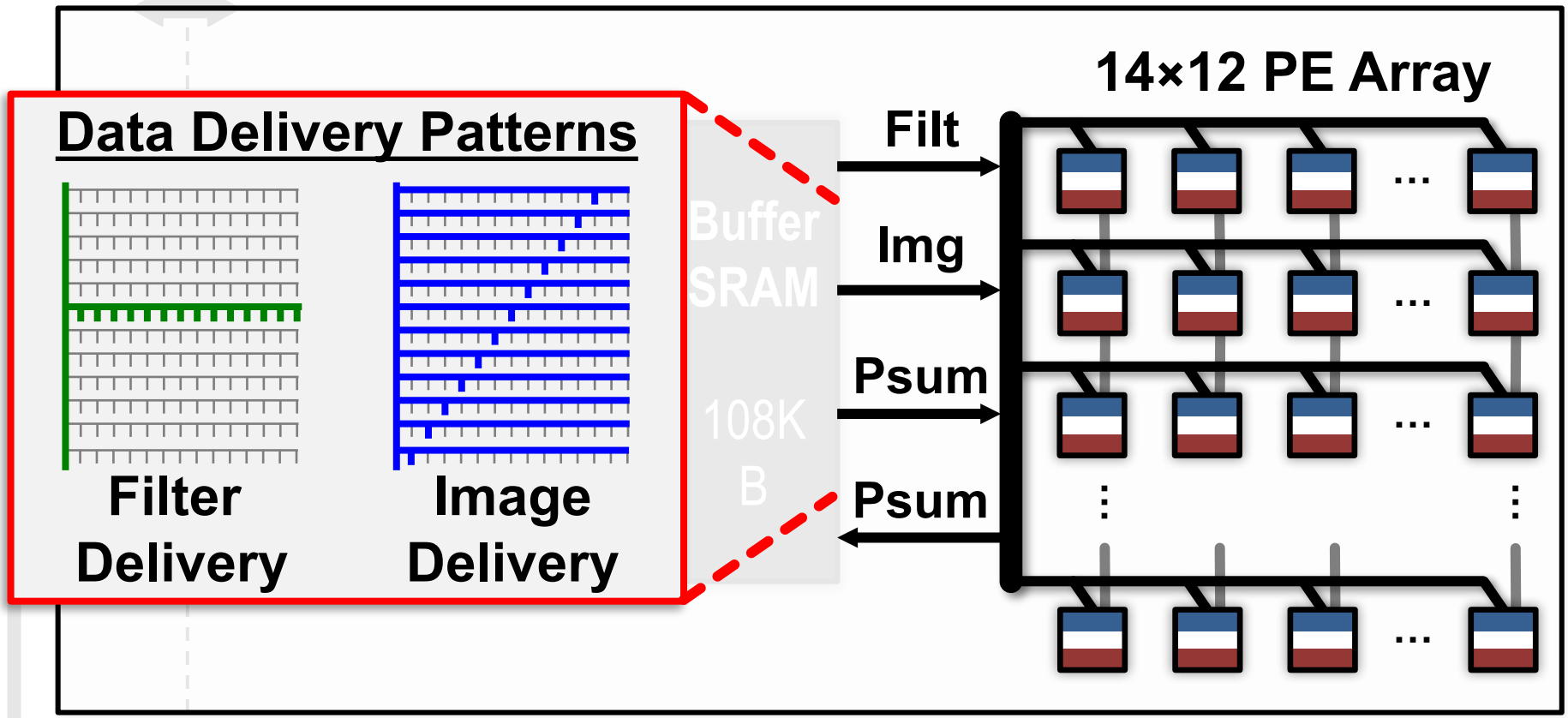
Physical PE Array

Unused PEs
are
Clock Gated

Multicast Network Design



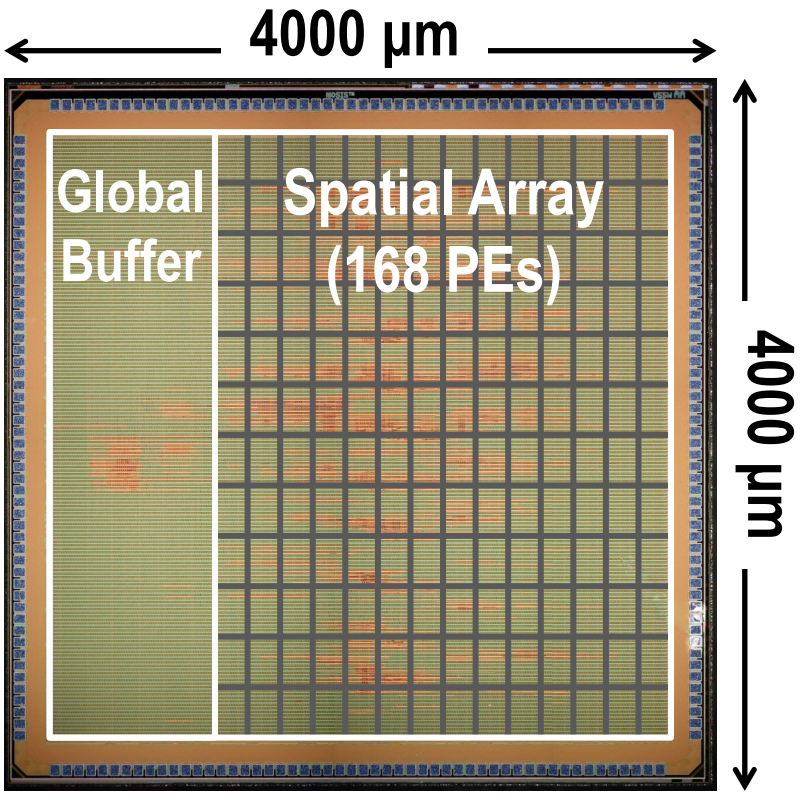
Data Delivery with On-Chip Network



Compared to Broadcast, **Multicast** saves >**80%** of NoC energy

Chip Spec & Measurement Results

Technology	TSMC 65nm LP 1P9M
On-Chip Buffer	108 KB
# of PEs	168
Scratch Pad / PE	0.5 KB
Core Frequency	100 – 250 MHz
Peak Performance	33.6 – 84.0 GOPS
Word Bit-width	16-bit Fixed-Point
Natively Supported DNN Shapes	Filter Width: 1 – 32 Filter Height: 1 – 12 Num. Filters: 1 – 1024 Num. Channels: 1 – 1024 Horz. Stride: 1–12 Vert. Stride: 1, 2, 4



[Chen et al., ISSCC 2016]

Benchmark – AlexNet Performance

Image Batch Size of **4** (i.e. 4 frames of 227x227)
Core Frequency = 200MHz / Link Frequency = 60 MHz

Layer	Power (mW)	Latency (ms)	# of MAC (MOPs)	Active # of PEs (%)	Buffer Data Access (MB)	DRAM Data Access (MB)
1	332	20.9	422	154 (92%)	18.5	5.0
2	288	41.9	896	135 (80%)	77.6	4.0
3	266	23.6	598	156 (93%)	50.2	3.0
4	235	18.4	449	156 (93%)	37.4	2.1
5	236	10.5	299	156 (93%)	24.9	1.3
Total	**278**	**115.3**	**2663**	**148 (88%)**	**208.5**	**15.4**

To support 2.66 GMACs [8 billion 16-bit inputs (**16GB**) and 2.7 billion outputs (**5.4GB**)], only requires **208.5MB** (buffer) and **15.4MB** (DRAM)

Benchmark – AlexNet Performance

Image Batch Size of **4** (i.e. 4 frames of 227x227)
Core Frequency = 200MHz / Link Frequency = 60 MHz

Layer	Power (mW)	Latency (ms)	# of MAC (MOPs)	Active # of PEs (%)	Buffer Data Access (MB)	DRAM Data Access (MB)
1	332	20.9	422	154 (92%)	18.5	5.0
2	288	41.9	896	135 (80%)	77.6	4.0
3	266	23.6	598	156 (93%)	50.2	3.0
4	235	18.4	449	156 (93%)	37.4	2.1
5	236	10.5	299	156 (93%)	24.9	1.3
Total	**278**	**115.3**	**2663**	**148 (88%)**	**208.5**	**15.4**

51682 operand* access/input image pixel

→ **506** access/pixel from buffer **+ 37** access/pixel from DRAM

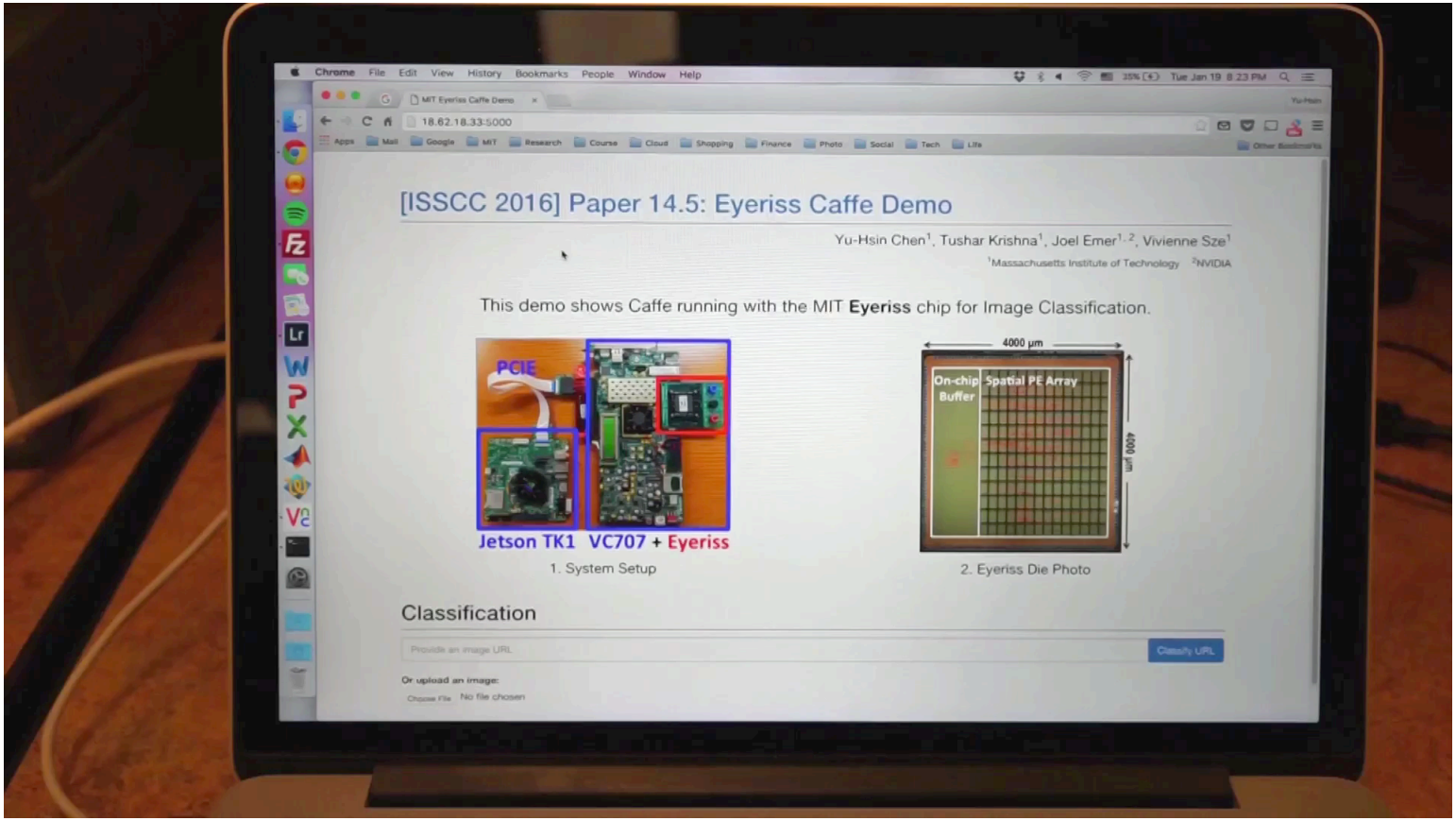
*operand = weight, activation, psum

Comparison with GPU

	This Work	NVIDIA TK1 (Jetson Kit)
Technology	65nm	28nm
Clock Rate	200MHz	852MHz
# Multipliers	168	192
On-Chip Storage	Buffer: 108KB Spad: 75.3KB	Shared Mem: 64KB Reg File: 256KB
Word Bit-Width	16b Fixed	32b Float
Throughput[1]	34.7 fps	68 fps
Measured Power	278 mW	Idle/Active[2]: 3.7W/10.2W
DRAM Bandwidth	127 MB/s	1120 MB/s [3]

1. AlexNet CONV Layers
2. Board Power
3. Modeled from [Tan, SC 2011]

From Architecture to System



https://vimeo.com/154012013

Summary of DNN Dataflows

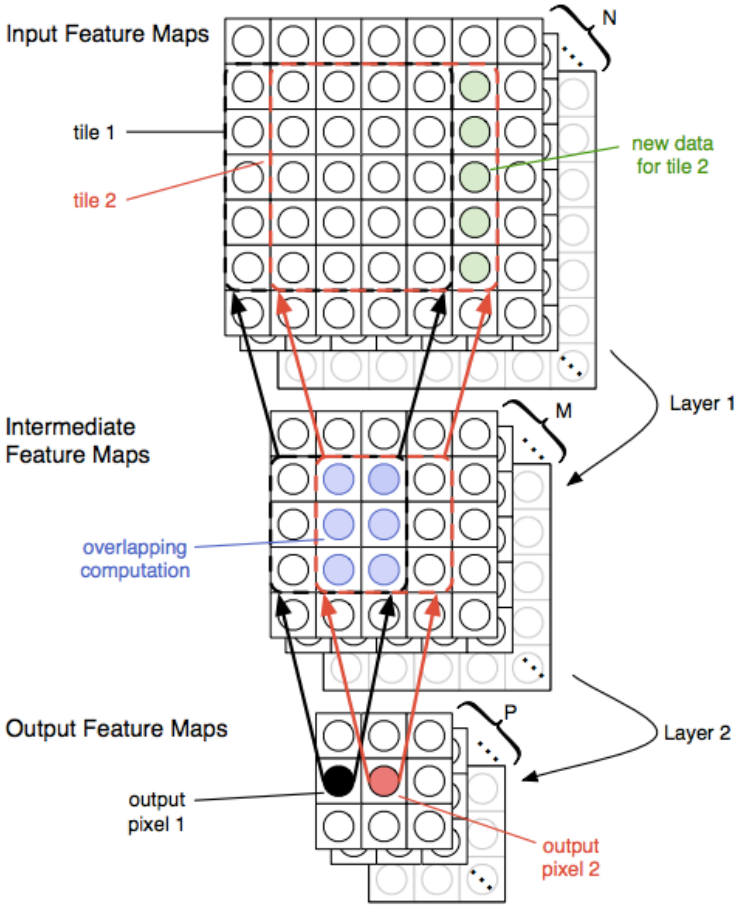
- **Weight Stationary**
  - Minimize movement of filter weights
  - Popular with processing-in-memory architectures
- **Output Stationary**
  - Minimize movement of partial sums
  - Different variants optimized for CONV or FC layers
- **No Local Reuse**
  - No PE local storage → maximize global buffer size
- **Row Stationary**
  - Adapt to the NN shape and hardware constraints
  - Optimized for overall **system energy efficiency**

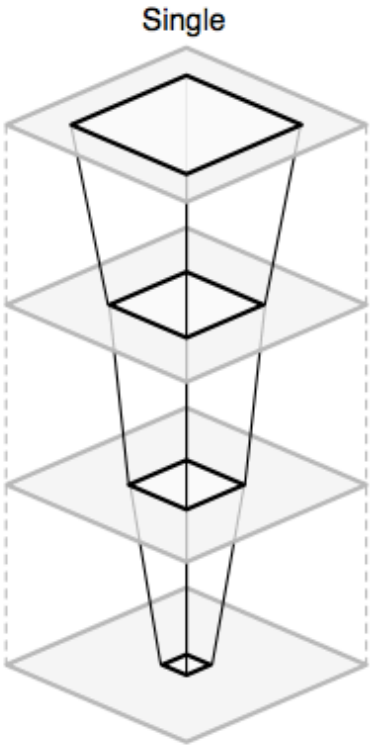
MICRO 2016 Papers in the Taxonomy

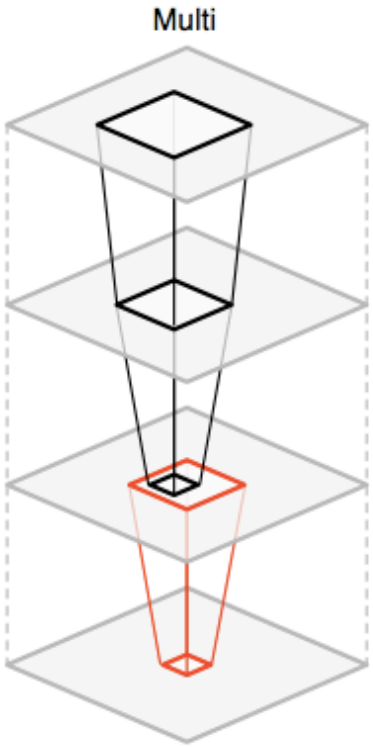
- **Stripes:** bit-serial computation in a **NLR**-like engine (based on DaDianNao)
- **NEUTRAMS**: a toolset for accelerators running the **WS** dataflow (synaptic weight memory array)
- **Fused-layer**: exploit inter-layer data reuse in a **NLR** engine (based on [Zhang, *FPGA* 2015])

Fused Layer

- **Dataflow across multiple layers**







[Alwani et al., MICRO 2016]

Advanced Technology Opportunities

MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

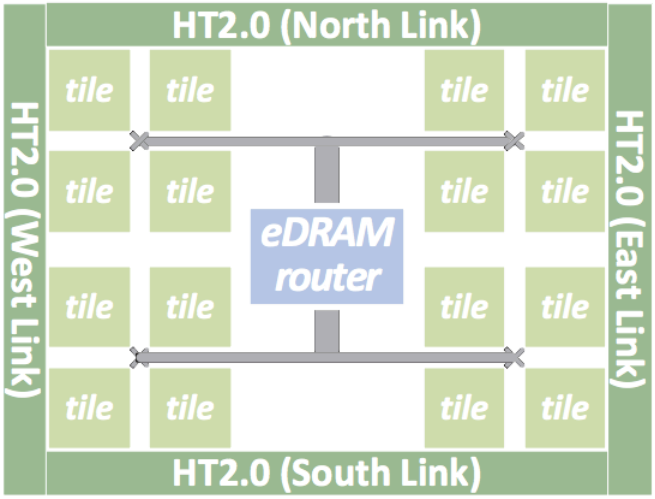
Advanced Storage Technology

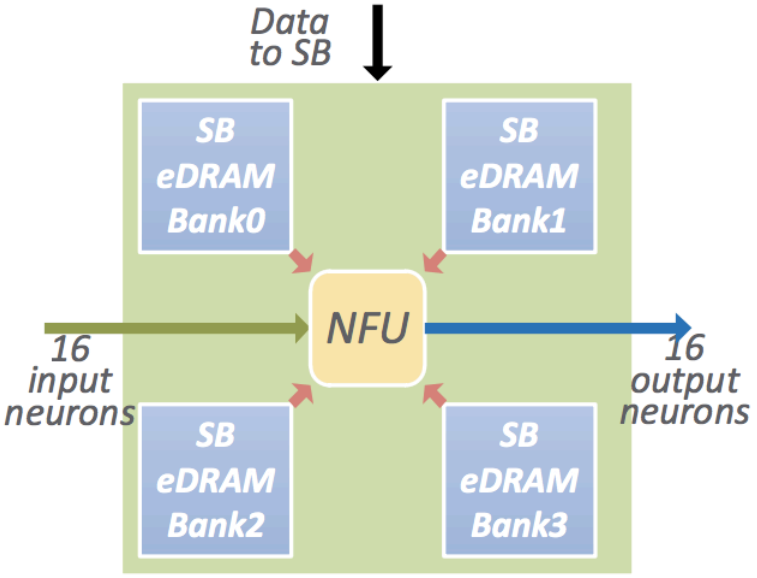
- **Embedded DRAM (eDRAM)**
  - **Increase on-chip storage capacity**
- **3D Stacked DRAM**
  - **e.g. Hybrid Memory Cube Memory (HMC), High Bandwidth Memory (HBM)**
  - **Increase memory bandwidth**

eDRAM (DaDianNao)

- **Advantages of eDRAM**
  - **2.85x higher density than SRAM**
  - **321x more energy-efficient than DRAM (DDR3)**
- **Store weights in eDRAM (36MB)**
  - **Target fully connected layers since dominated by weights**

16 Parallel Tiles

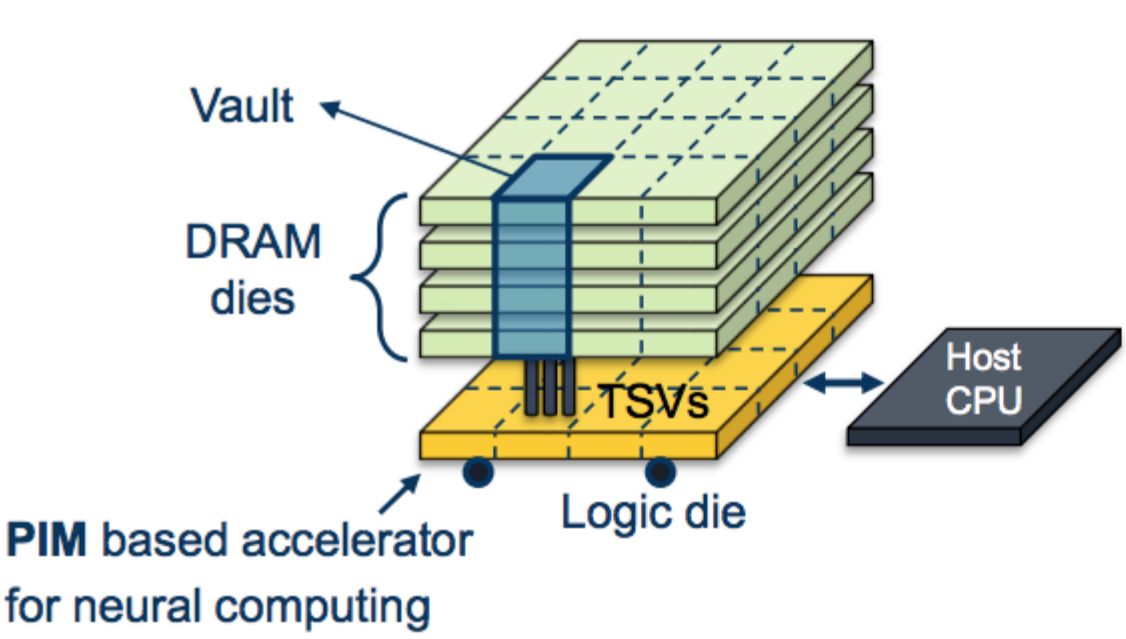


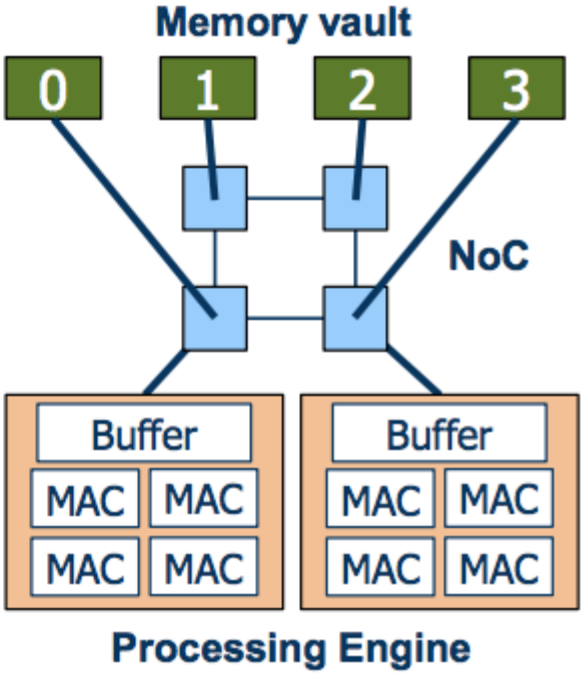


[Chen et al., DaDianNao, MICRO 2014]

Stacked DRAM (NeuroCube)

- **NeuroCube on Hyper Memory Cube Logic Die**
  - **6.25x higher BW than DDR3**
    - **HMC (16 ch x 10GB/s) > DDR3 BW (2 ch x 12.8GB/s)**
  - **Computation closer to memory (reduce energy)**





[Kim et al., NeuroCube, ISCA 2016]

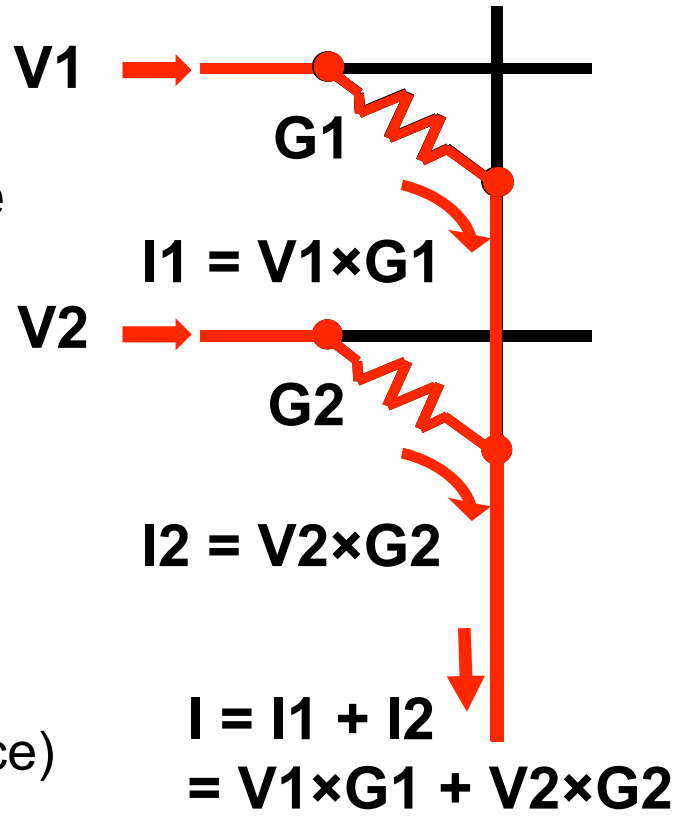
Analog Computation

- **Conductance = Weight**
- **Voltage = Input**
- **Current = Voltage × Conductance**
- **Sum currents for addition**

$$Output = \sum Weight \times Input$$

Input = V1, V2, …

Filter Weights = G1, G2, … (conductance)



Weight Stationary Dataflow

Figure Source: ISAAC, ISCA 2016

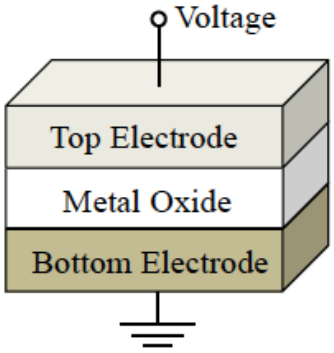
Memristor Computation

Use memristors as programmable weights (resistance)

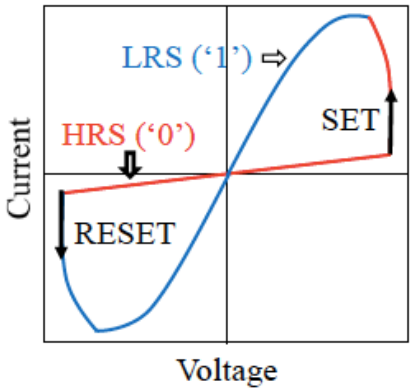
- **Advantages**
  - **High Density (< 10nm x 10nm size*)**
    - **~30x smaller than SRAM****
    - **1.5x smaller than DRAM****
  - **Non-Volatile**
  - **Operates at low voltage**
  - **Computation within memory (in situ)**
    - **Reduce data movement**

*[Govoreanu et al., IEDM 2011], **ITRS 2013

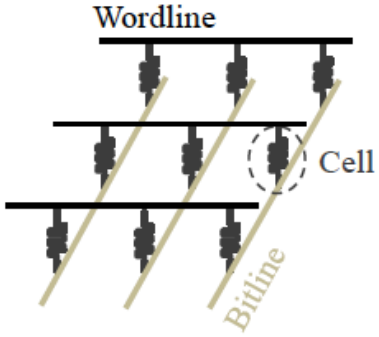
Memristor



(a) Conceptual view of a ReRAM cell

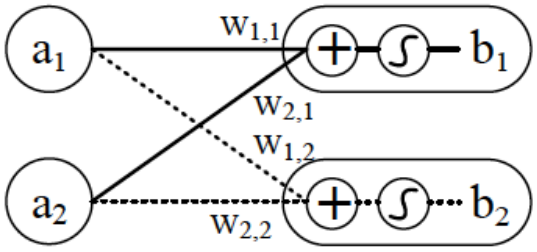


(b) I-V curve of bipolar switching

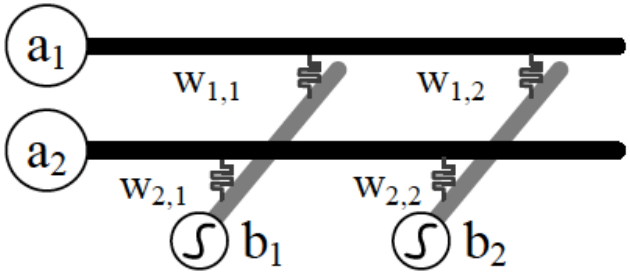


(c) schematic view of a crossbar architecture

$$b_j = \sigma\left(\sum_{\forall i} a_i \cdot w_{i,j}\right)$$



(a) An ANN with one input and one output layer



(b) using a ReRAM crossbar array for neural computation

[Chi et al., ISCA 2016]

Resistive Memory Devices

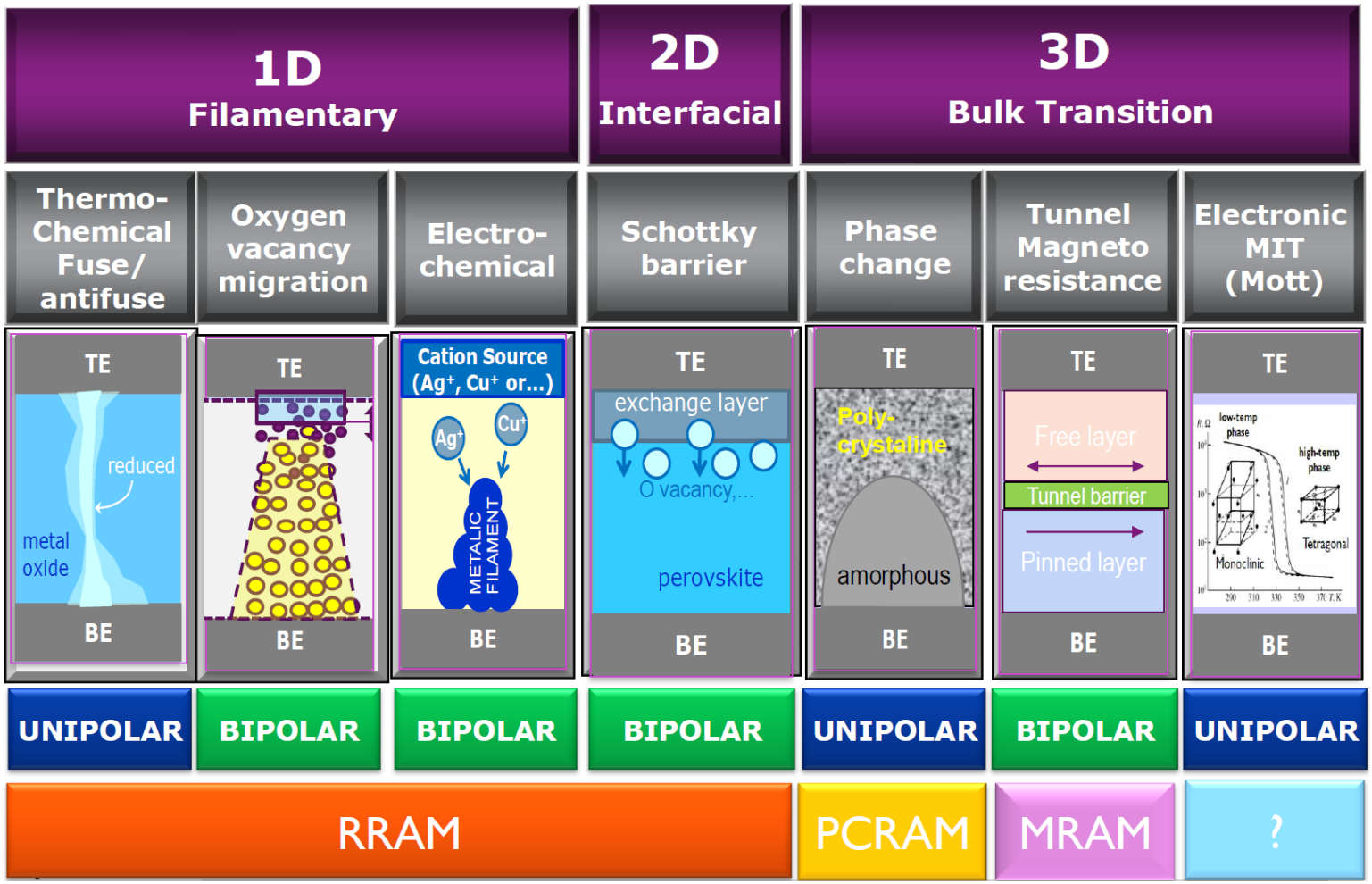


Figure Source: Han Wang, USC

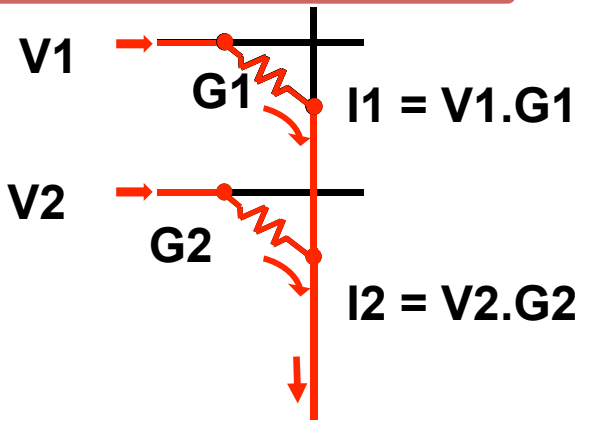
Challenges with Memristors

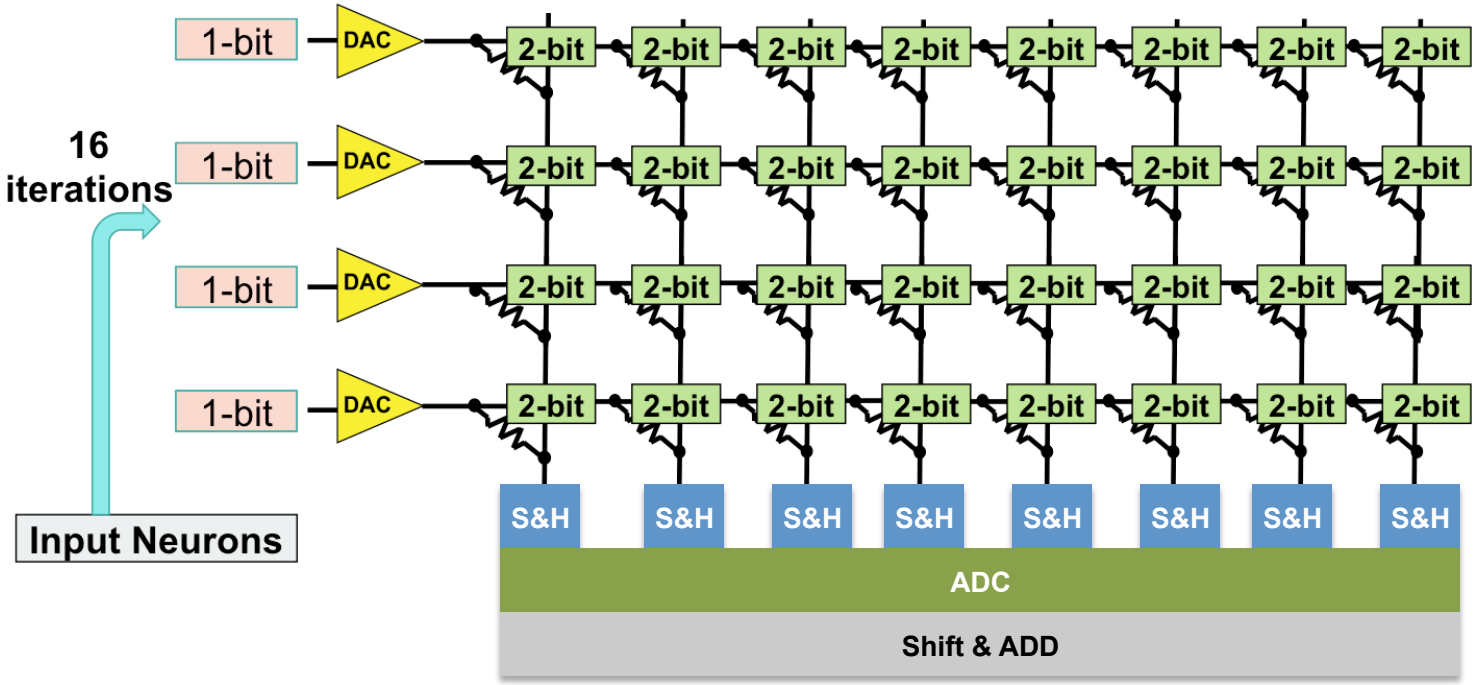
- **Limited Precision**
- **A/D and D/A Conversion**
- **Array Size and Routing**
  - **Wire dominates energy for array size of 1k × 1k**
  - **IR drop along wire can degrade read accuracy**
- **Write/programming energy**
  - **Multiple pulses can be costly**
- **Variations & Yield**
  - **Device-to-device, cycle-to-cycle**
  - **Non-linear conductance across range**

[Eryilmaz et al., ISQED 2016]

ISAAC

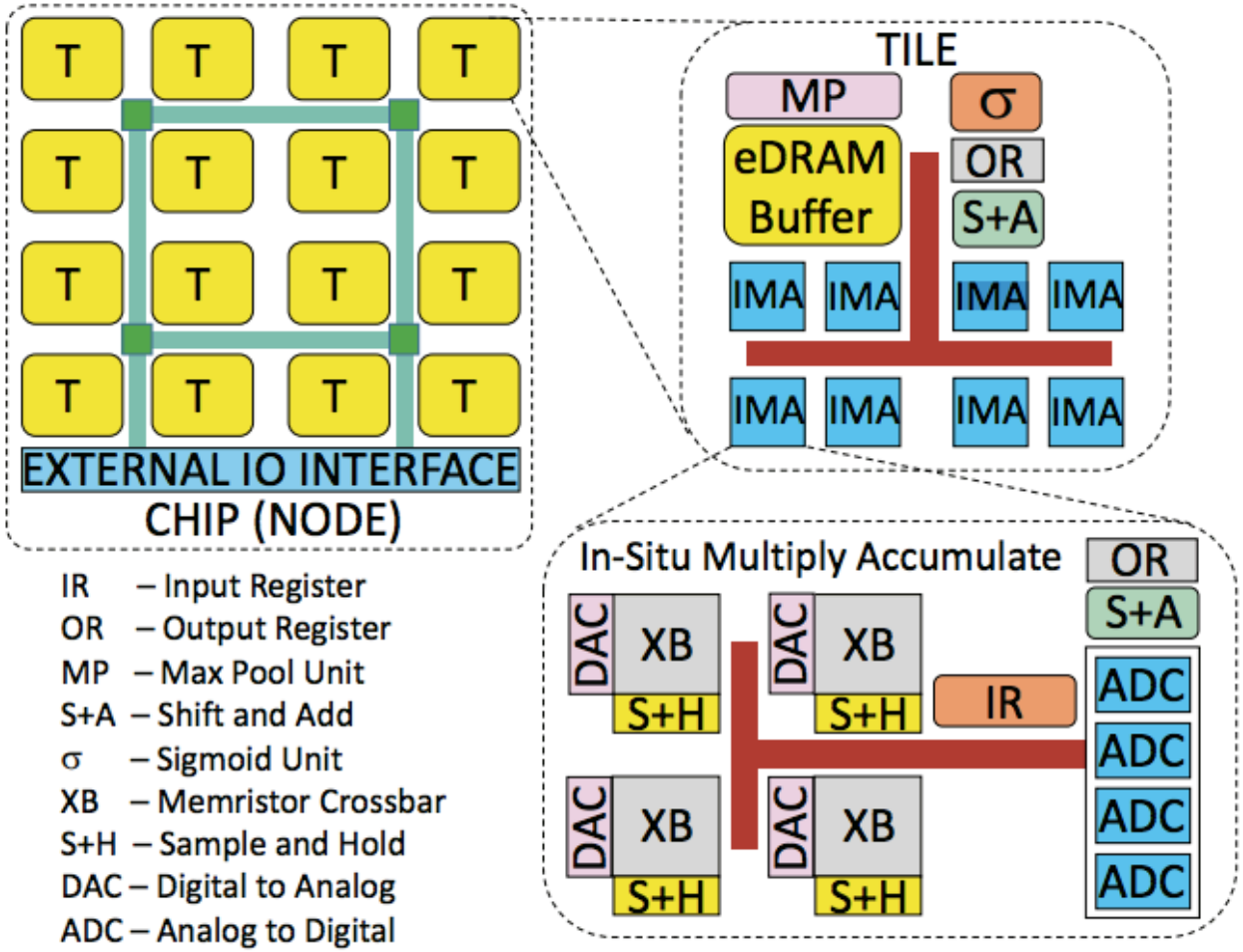
- eDRAM using memristors
- 16-bit dot-product operation
  - 8 x 2-bits per memristors
  - 1-bit per cycle computation





[Shafiee et al., ISCA 2016]

ISAAC



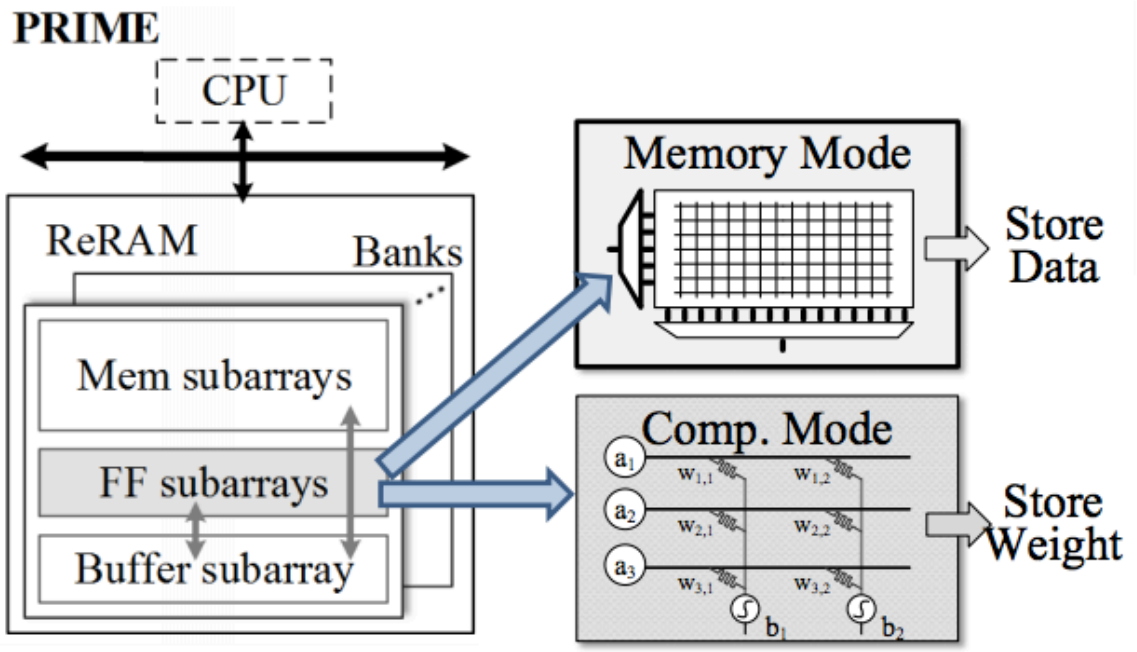
Eight 128x128 arrays per IMA

12 IMAs per Tile

[Shafiee et al., ISCA 2016]

PRIME

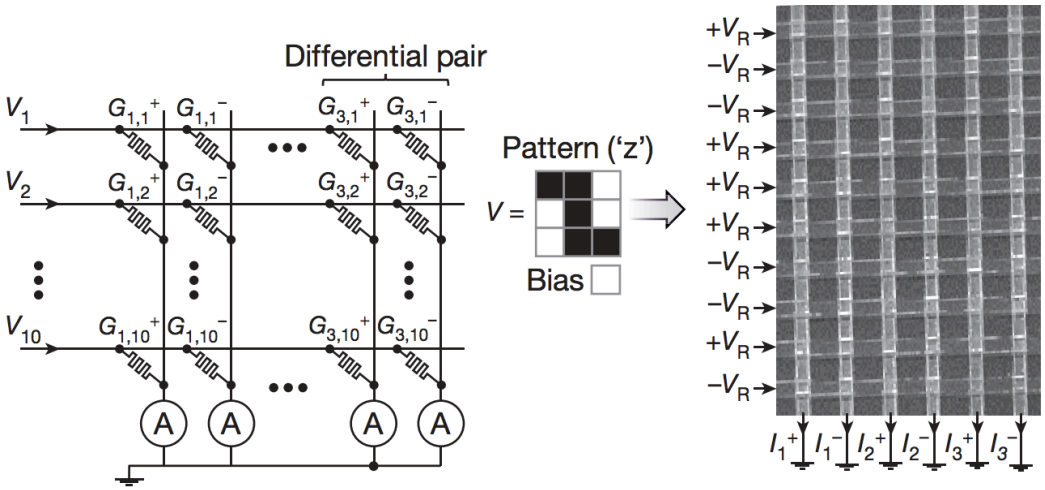
- **Bit precision for each 256x256 ReRAM array**
  - **3-bit input, 4-bit weight (2x for 6-bit input and 8-bit weight)**
  - **Dynamic fixed point (6-bit output)**
- **Reconfigurable to be main memory or accelerator**
  - **4-bit MLC computation; 1-bit SLC for storage**

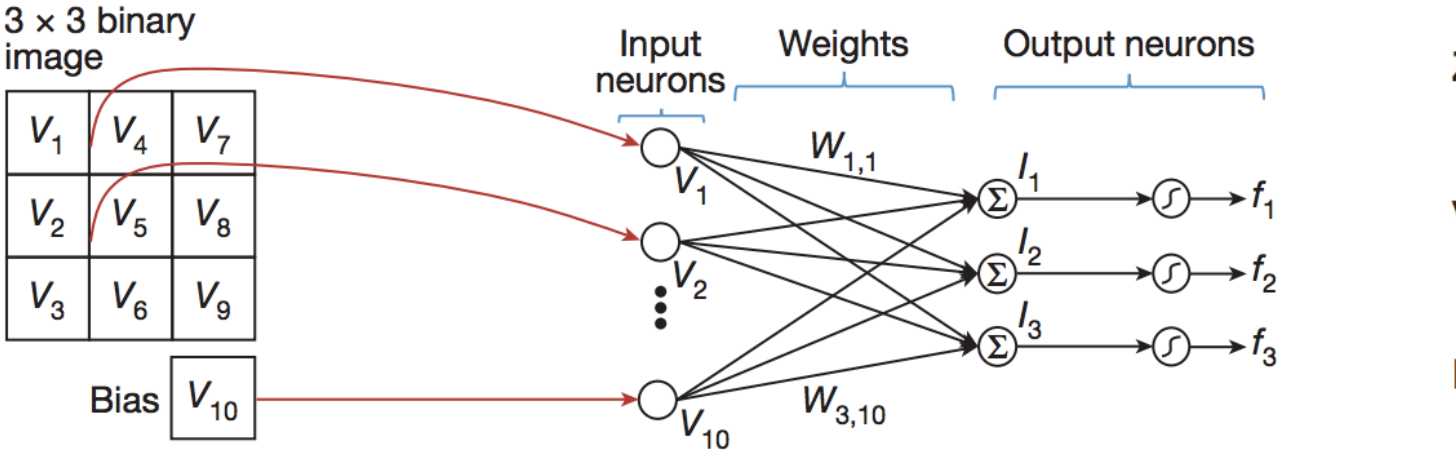


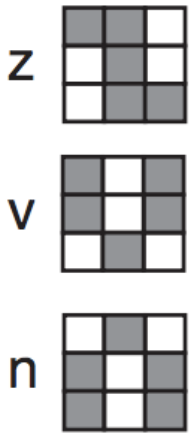
[Chi et al., ISCA 2016]

Fabricated Memristor Crossbar

- **Transistor-free metal-oxide 12x12 crossbar**
  - **A single-layer perceptron (linear classification)**
  - **3x3 binary image**
  - **10 inputs x 3 outputs x 2 differential weights = 60 memristors**







[Prezioso et al., Nature 2015]

Network and Hardware Co-Design

MICRO Tutorial (2016)

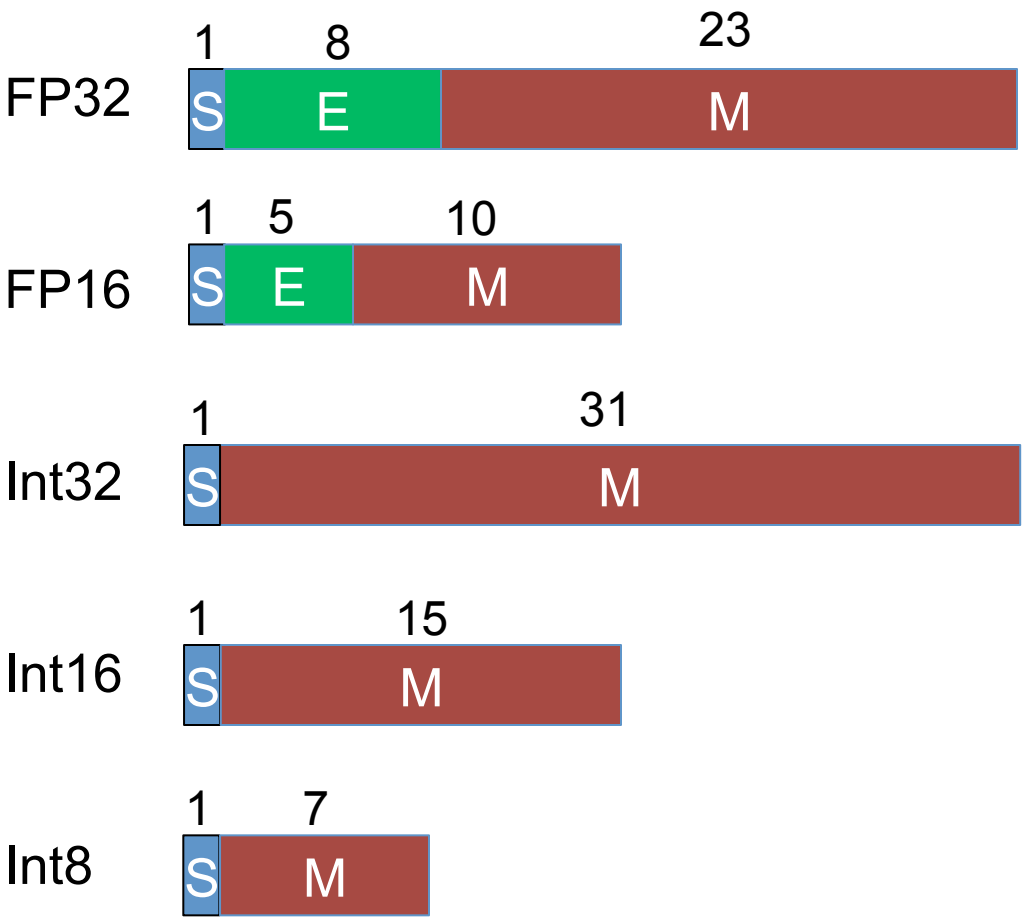
Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

Network Optimization

- Reduce precision of operations and operands
  - Fixed and Floating point
  - Bit-width
- Reduce number of operations and storage of weights
  - Compression
  - Pruning
  - Network Architectures

Number Representation

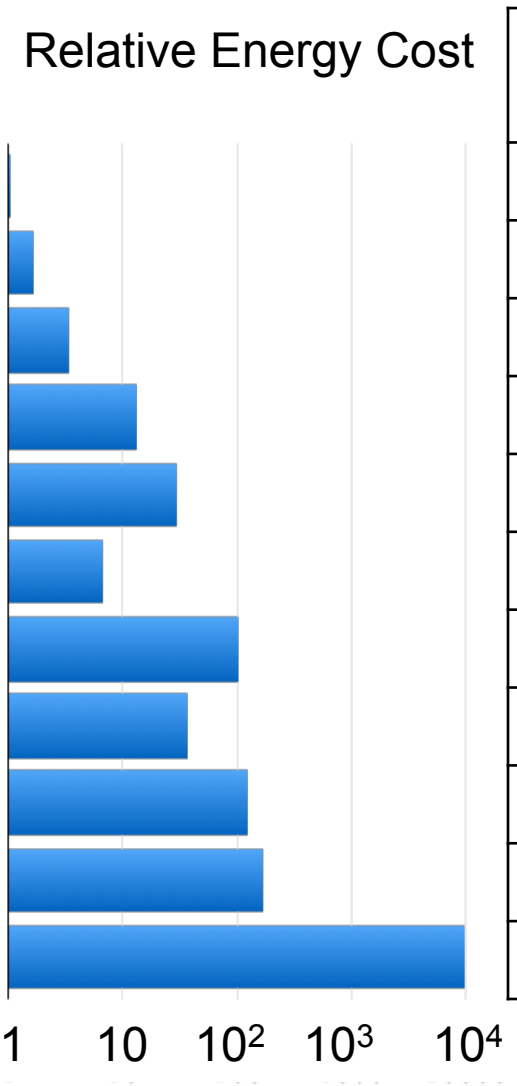


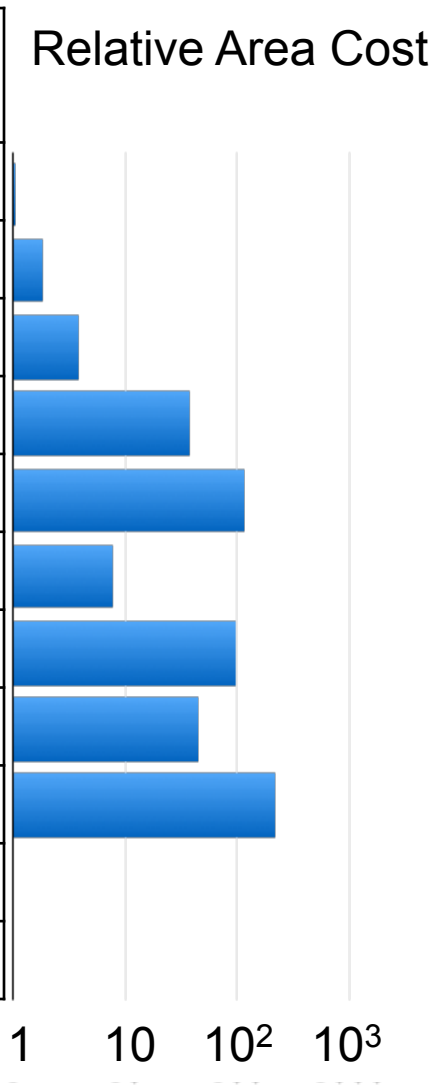
	Range	Accuracy
FP32	10^{-38} – 10^{38}	.000006%
FP16	$6x10^{-5}$ - $6x10^{4}$	.05%
Int32	0 – $2x10^{9}$	½
Int16	0 – $6x10^{4}$	½
Int8	0 – 127	½

Image Source: B. Dally

Cost of Operations

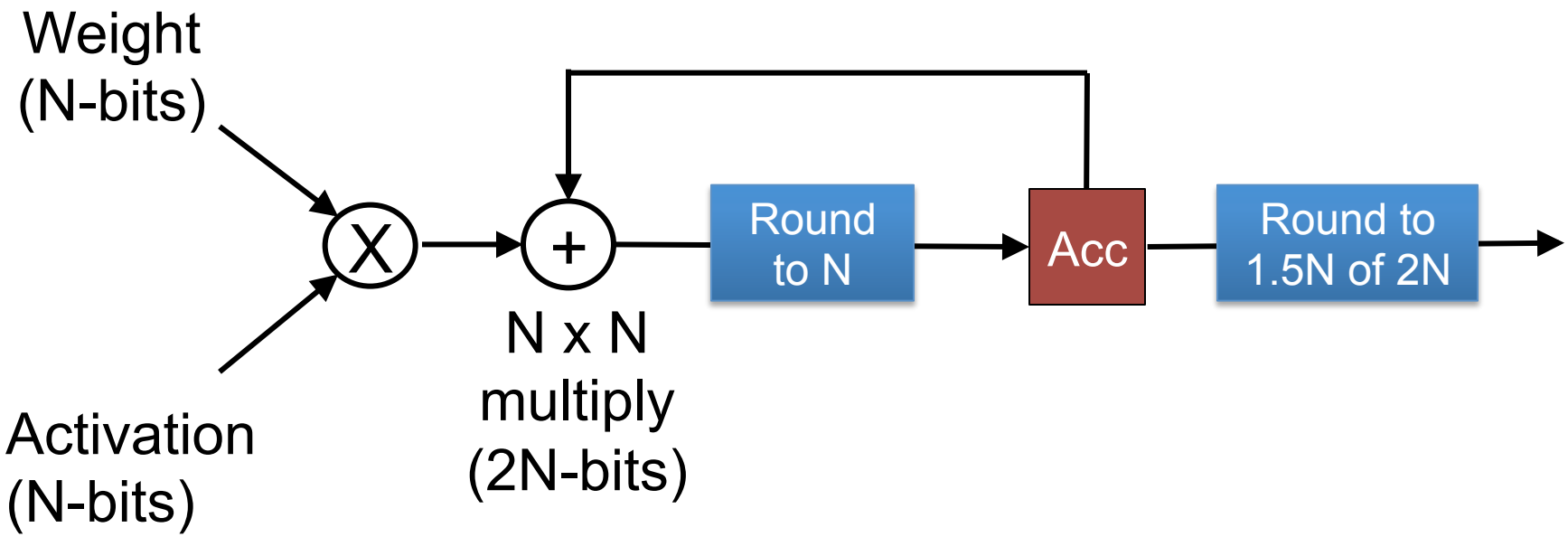
Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A





[Horowitz, "Computing's Energy Problem (and what we can do about it)", ISSCC 2014]

N-bit Precision

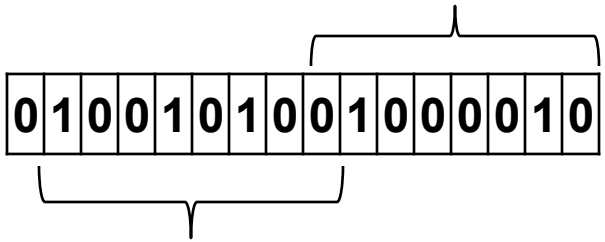


Methods to Reduce Bits

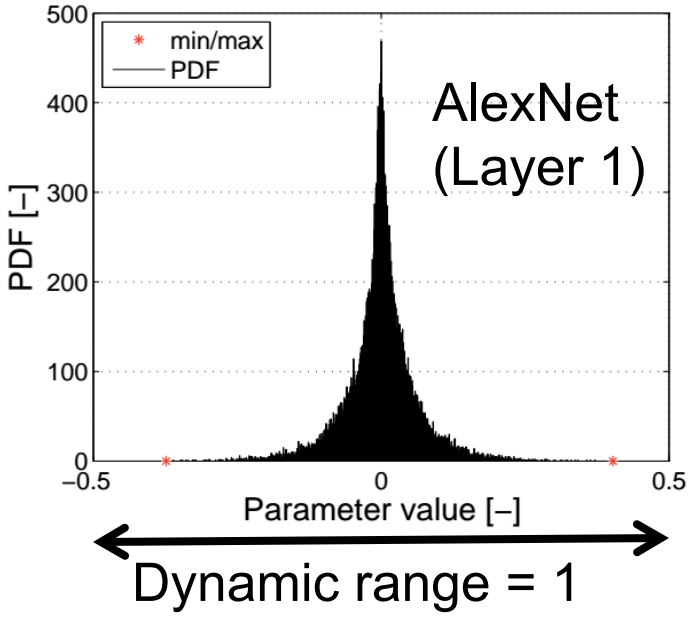
- **Quantization/Rounding**
- **Dynamic Fixed Point**
  - **Rescale and Reduce bits**
- **Fine-tuning: Retrain Weights**

Example: 16-bit → 8-bits

$2^{11} + 2^9 + 2^6 + 2^1 = 2626$ (overflow)



$2^{10} + 2^7 + 2^5 + 2^2 + 2^0 = 1189$



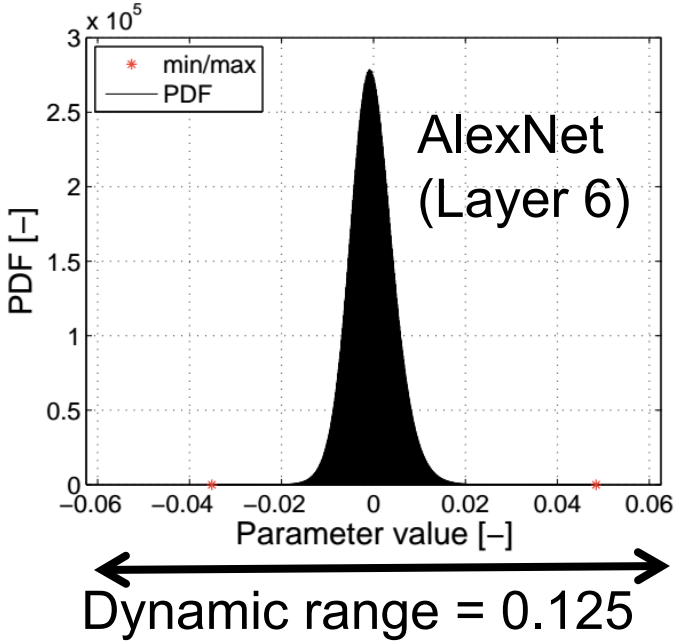
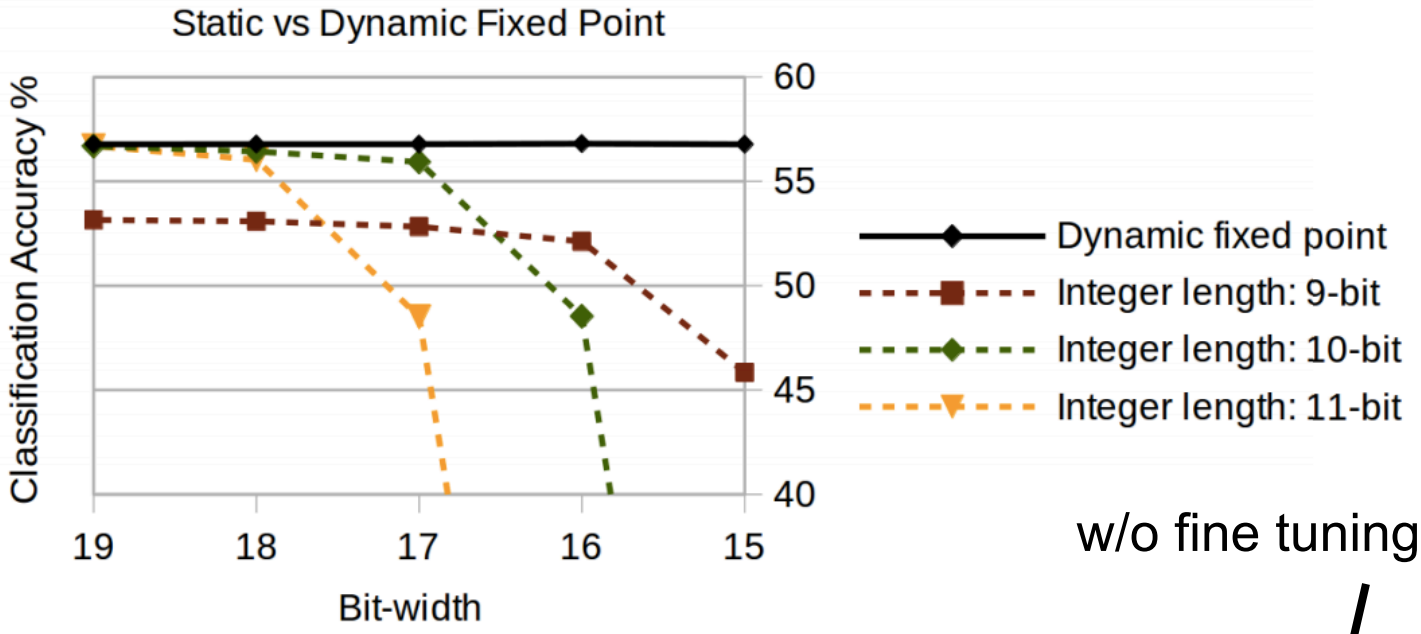


Image Source:
Moons et al,
WACV 2016

Batch normalization important to 'center' dynamic range

Impact on Accuracy

Top-1 accuracy on of CaffeNet on ImageNet



w/o fine tuning

	Layer outputs	CONV parameters	FC parameters	32-bit floating point baseline	Fixed point accuracy
LeNet (Exp 1)	4-bit	4-bit	4-bit	99.1%	99.0% (98.7%)
LeNet (Exp 2)	4-bit	2-bit	2-bit	99.1%	98.8% (98.0%)
Full CIFAR-10	8-bit	8-bit	8-bit	81.7%	81.4% (80.6%)
SqueezeNet top-1	8-bit	8-bit	8-bit	57.7%	57.1% (55.2%)
CaffeNet top-1	8-bit	8-bit	8-bit	56.9%	56.0% (55.8%)
GoogLeNet top-1	8-bit	8-bit	8-bit	68.9%	66.6% (66.1%)

[Gysel et al., Ristretto, ICLR 2016]

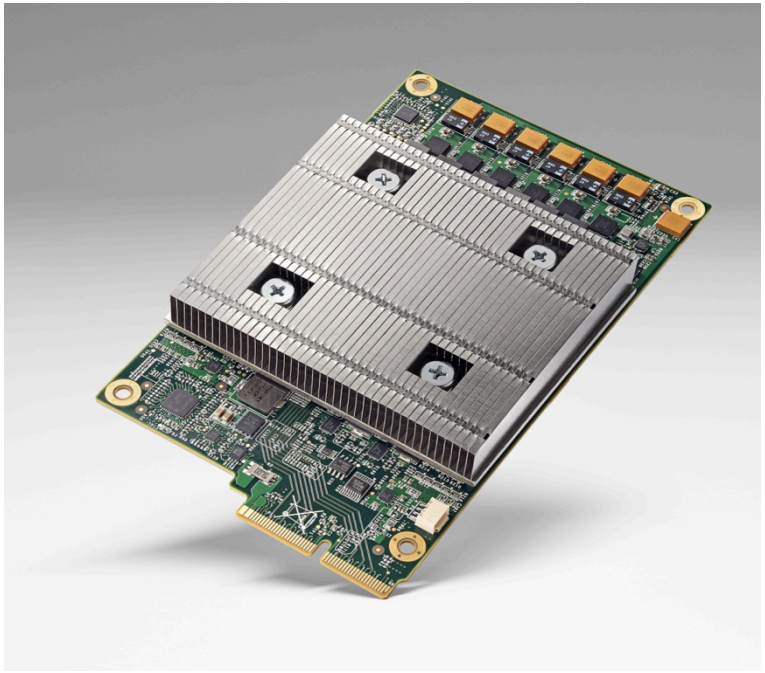
Google's Tensor Processing Unit (TPU)

" With its TPU Google has seemingly focused on delivering the data really quickly by **cutting down on precision**. Specifically, it doesn't rely **on floating point precision like a GPU**

….

Instead the chip uses integer math…TPU used **8-bit integer**."

- Next Platform (May 19, 2016)



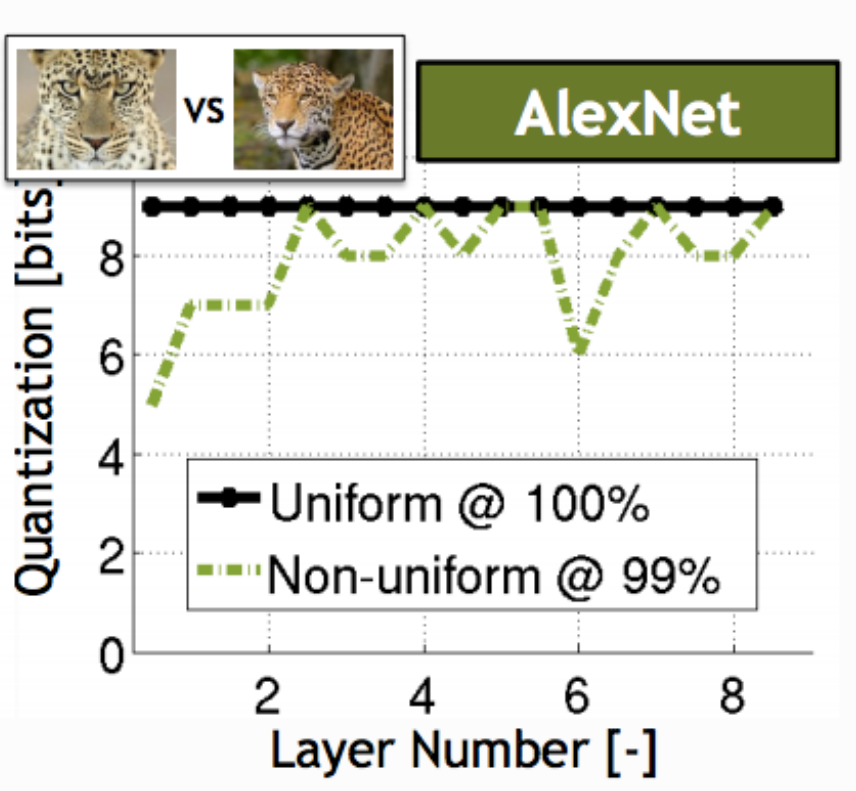
Nvidia PASCAL

"New half-precision, **16-bit floating point instructions deliver over 21 TeraFLOPS** for unprecedented training performance. **With 47 TOPS (tera-operations per second) of performance, new 8-bit integer instructions** in Pascal allow AI algorithms to deliver real-time responsiveness for deep learning inference."

– Nvidia.com (April 2016)

Precision Varies from Layer to Layer

Tolerance	Bits per layer (I+F)
AlexNet (F=0)	
1%	10-8-8-8-8-8-6-4
2%	10-8-8-8-8-8-5-4
5%	10-8-8-8-7-7-5-3
10%	9-8-8-8-7-7-5-3



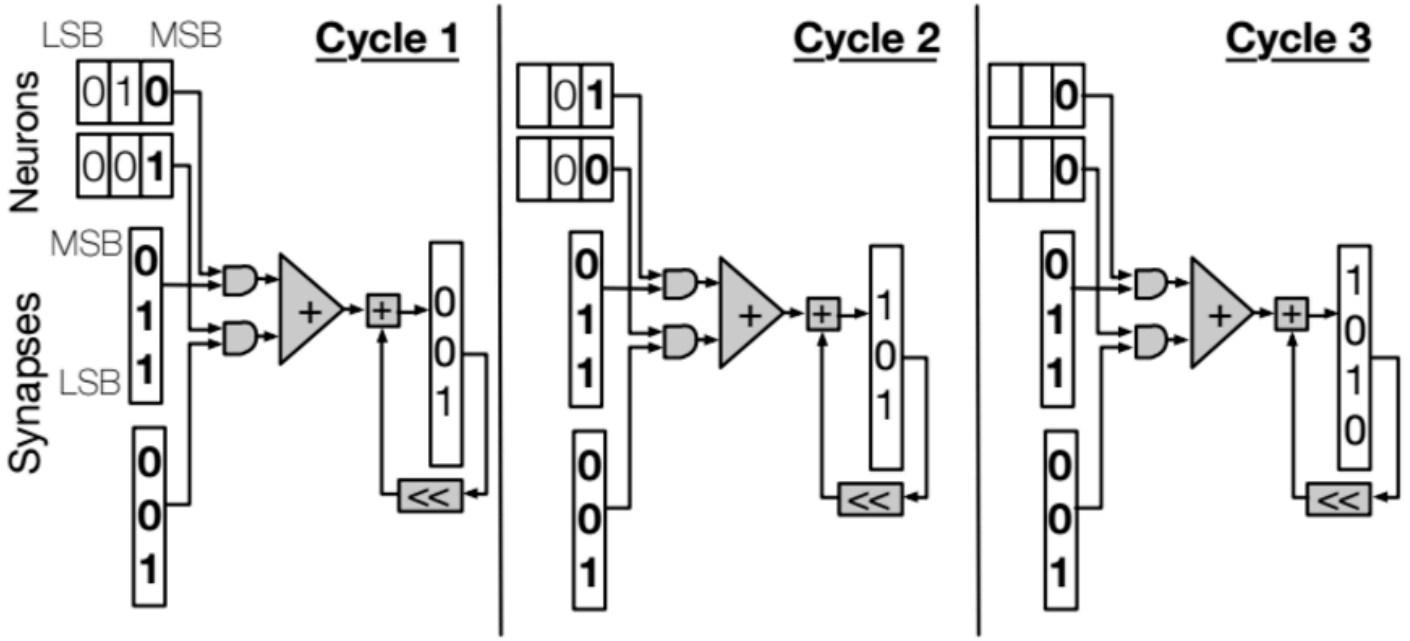
[Judd et al., ArXiv 2016]

[Moons et al., WACV 2016]

Bitwidth Scaling (Speed)

Bit-Serial Processing: Reduce Bit-width → Skip Cycles
Speed up of 2.24x vs. 16-bit fixed

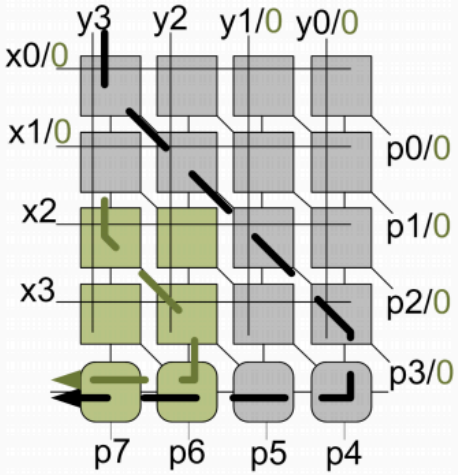
$$\sum_{i=0}^{N_i-1} s_i \times n_i = \sum_{i=0}^{N_i-1} s_i \times \sum_{b=0}^{P-1} n_i^b \times 2^b = \sum_{b=0}^{P-1} 2^b \times \sum_{i=0}^{N_i-1} n_i^b \times S_i$$

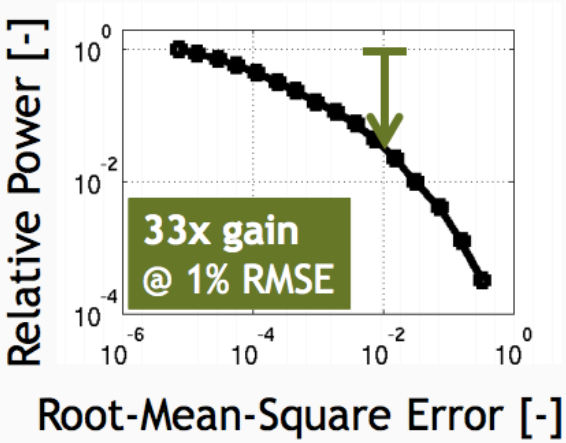


[Judd et al., Stripes, CAL 2016]

Bitwidth Scaling (Power)

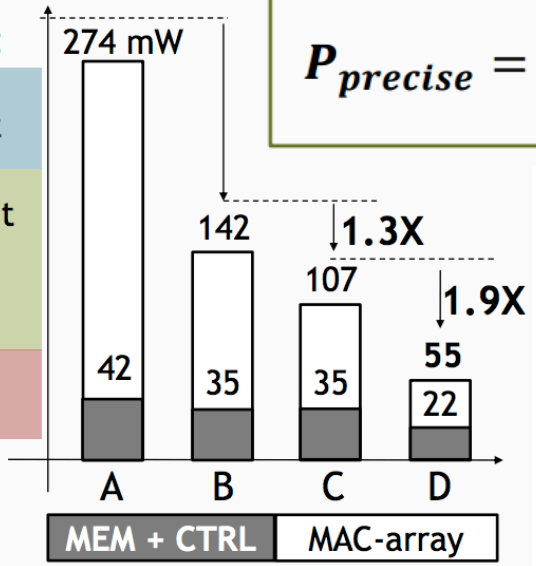
Reduce Bit-width → Shorter Critical Path → Reduce Voltage





AlexNet Layer 2 example:

A. 2D-baseline	@ 16 bit
B. Precision-Scaling	@ 7-7 bit
C. Voltage-Scaling	@ 0.9 V
D. Sparse operation guarding	

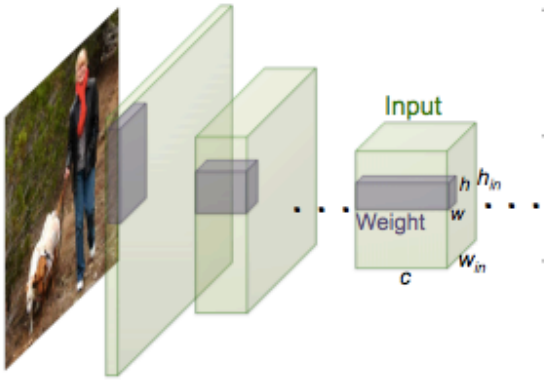


$$P_{precise} = \alpha C f V^2 \quad \Rightarrow \quad P_{imprecise} = \frac{\alpha}{k_1} C f \left(\frac{V}{k_2}\right)^2$$

Power reduction of 2.56x vs. 16-bit fixed On AlexNet Layer 2

[Moons et al., VLSI 2016]

Binary Nets



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

Classification Accuracy(%)									
Binary-Weight				Binary-Input-Binary-Weight				Full-Precision	
BWN		BC[11]		XNOR-Net		BNN[11]		AlexNet[1]	
Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
56.8	**79.4**	35.4	61.0	**44.2**	**69.2**	27.9	50.42	56.6	80.2

BinaryConnect (BC) = [Courbariaux et al., ArXiv 2015]
Binary Neural Networks (BNN) = [Courbariaux et al., ArXiv 2016]

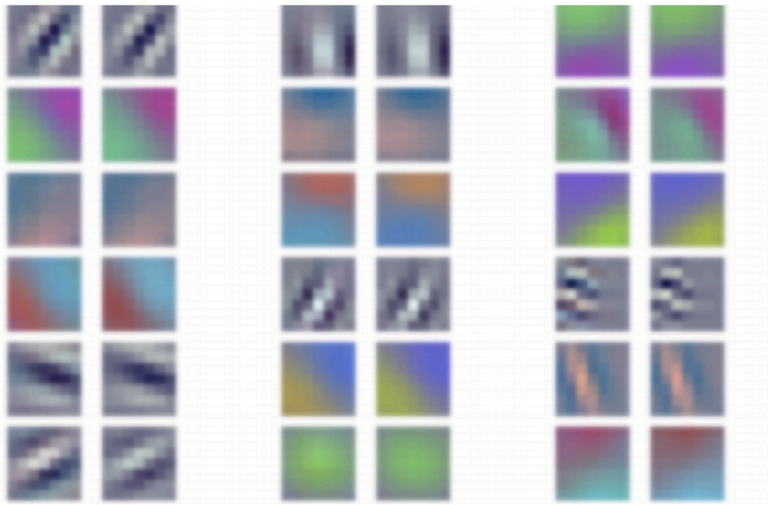
[Rastegari et al., BWN & XNOR-Net, ECCV 2016]

Reduce Number of Ops and Weights

- **Network Compression**
  - Low Rank Approximation
  - Weight Sharing and Vector Quantization
- **Pruning**
  - Weights
  - Activations
- **Network Architectures**

Low Rank Approximation

- **Low Rank approximation**
  - **Tensor decomposition based on singular value decomposition (SVD)**
  - **Filter Clustering with modified K-means**
  - **Fine Tuning**



- **Speed up by 1.6 – 2.7x on CPU/GPU for CONV1, CONV2 layers**
- **Reduce size by 5 - 13x for FC layer**
- **< 1% drop in accuracy**

[Denton et al., NIPS 2014]

Low Rank Approximation on Phone

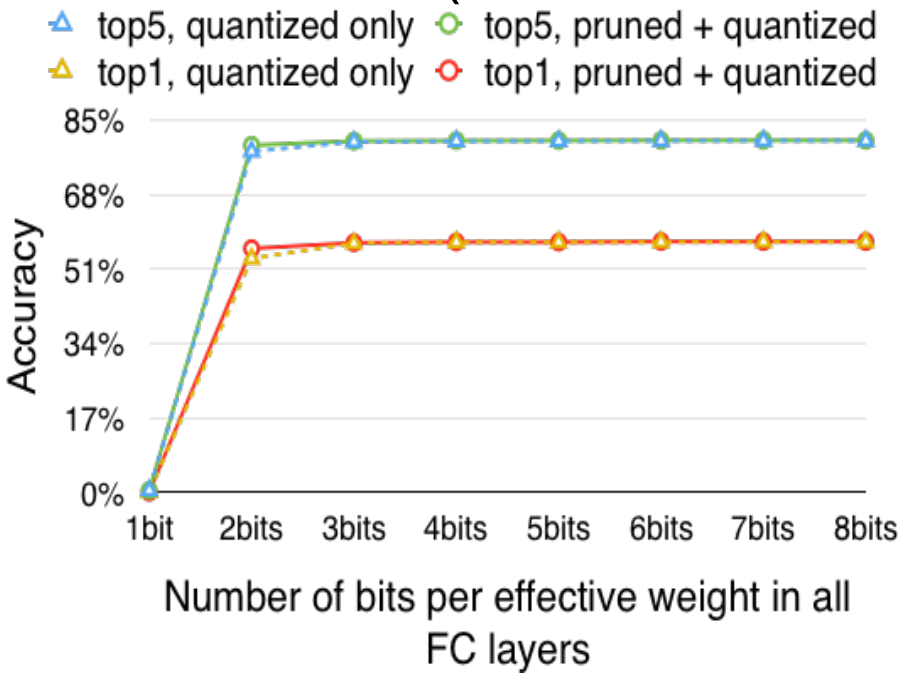
- **Rank selection per Layer**
- **Tucker Decomposition (extension of SVD)**
- **Fine tuning**

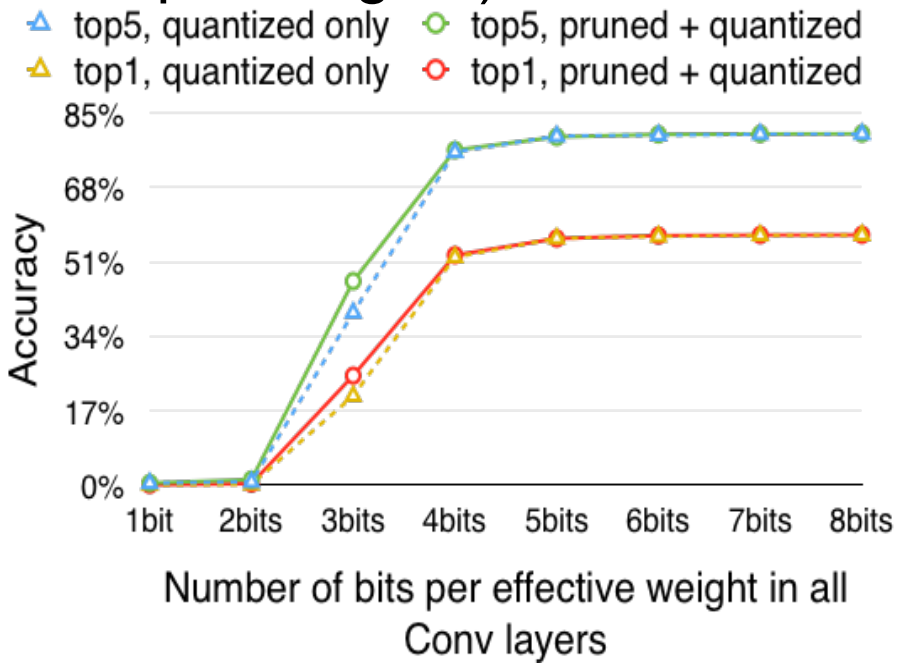
Model	Top-5	Weights	FLOPs	S6		Titan X
AlexNet	80.03	61M	725M	117ms	245mJ	0.54ms
*AlexNet**	78.33	11M	272M	43ms	72mJ	0.30ms
(imp.)	(-1.70)	(×5.46)	(×2.67)	(×2.72)	(×3.41)	(×1.81)
VGG-S	84.60	103M	2640M	357ms	825mJ	1.86ms
*VGG-S**	84.05	14M	549M	97ms	193mJ	0.92ms
(imp.)	(-0.55)	(×7.40)	(×4.80)	(×3.68)	(×4.26)	(×2.01)
GoogLeNet	88.90	6.9M	1566M	273ms	473mJ	1.83ms
*GoogLeNet**	88.66	4.7M	760M	192ms	296mJ	1.48ms
(imp.)	(-0.24)	(×1.28)	(×2.06)	(×1.42)	(×1.60)	(×1.23)
VGG-16	89.90	138M	15484M	1926ms	4757mJ	10.67ms
*VGG-16**	89.40	127M	3139M	576ms	1346mJ	4.58ms
(imp.)	(-0.50)	(×1.09)	(×4.93)	(×3.34)	(×3.53)	(×2.33)

[Kim et al., ICLR 2016]

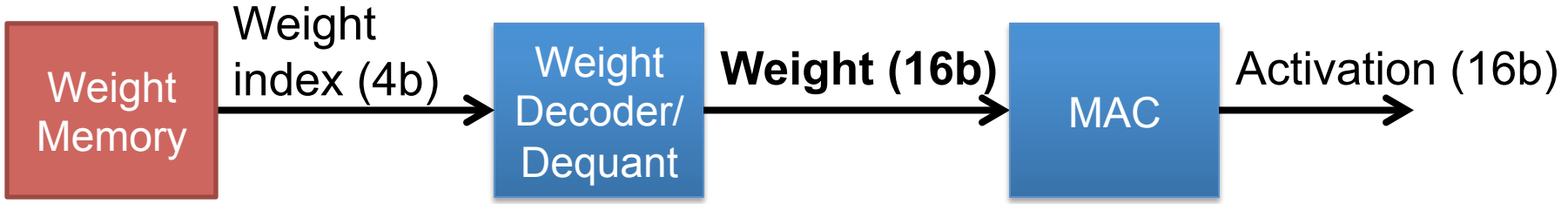
Weight Sharing + Vector Quantization

Trained Quantization: Weight Sharing via K-means clustering (reduce number of unique weights)





Reduce Bits for Storage (compute still 16-bits)

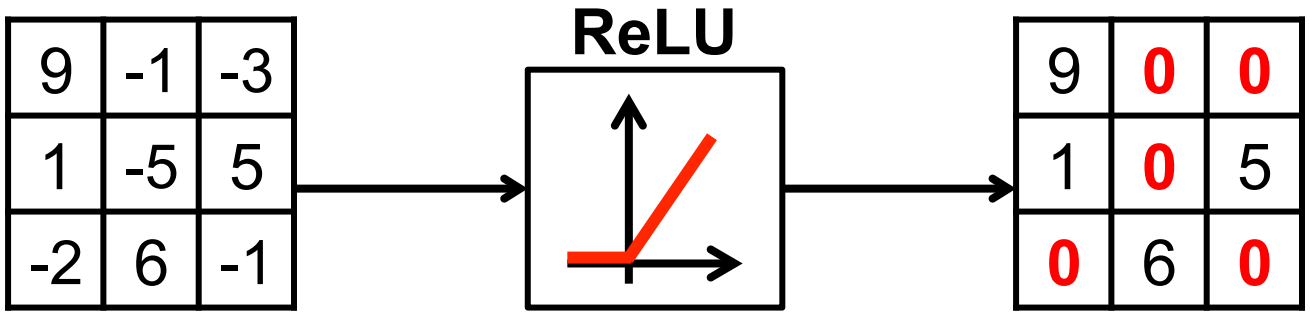


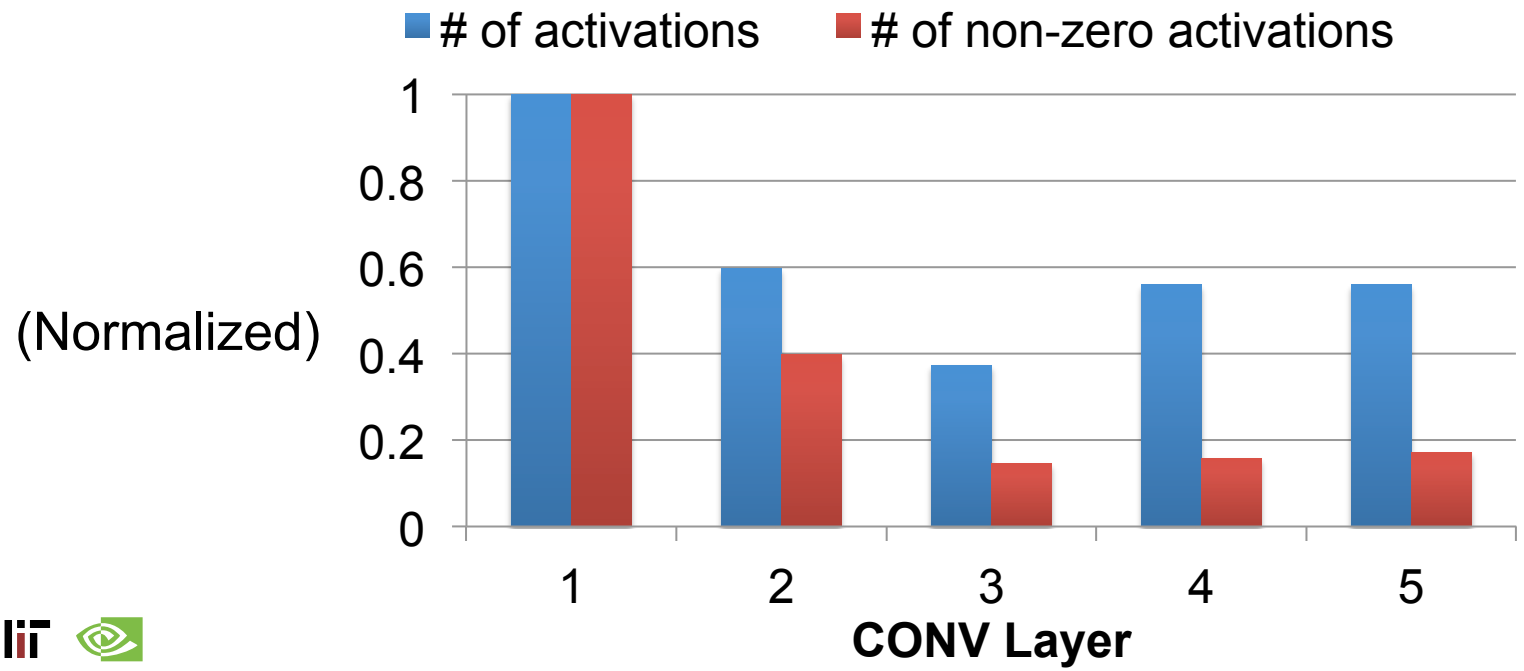
[Han et al., Deep Compression, ICLR 2016]

Exploit Data Statistics

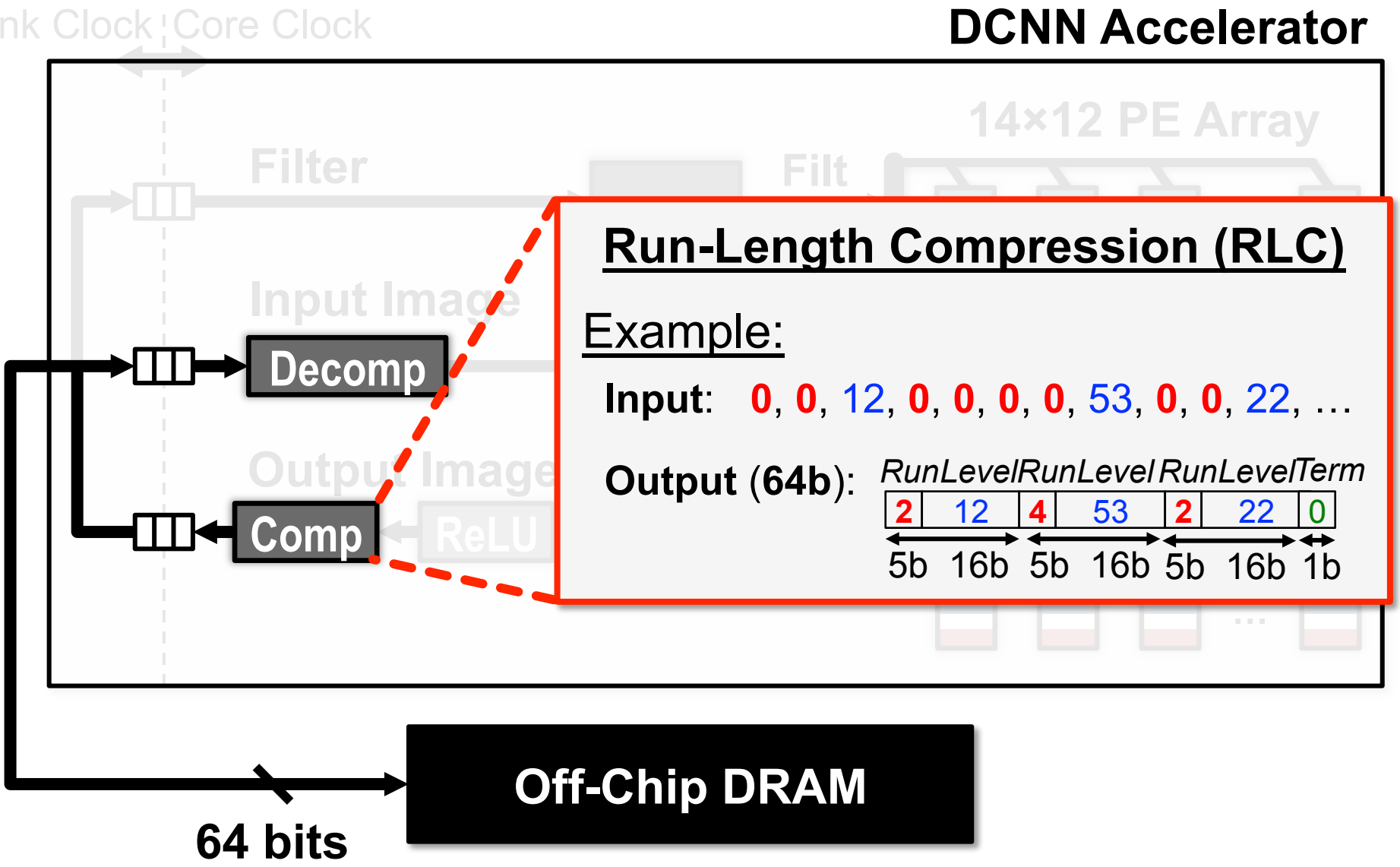
Sparsity in Fmaps

Many **zeros** in **output fmaps** after **ReLU**



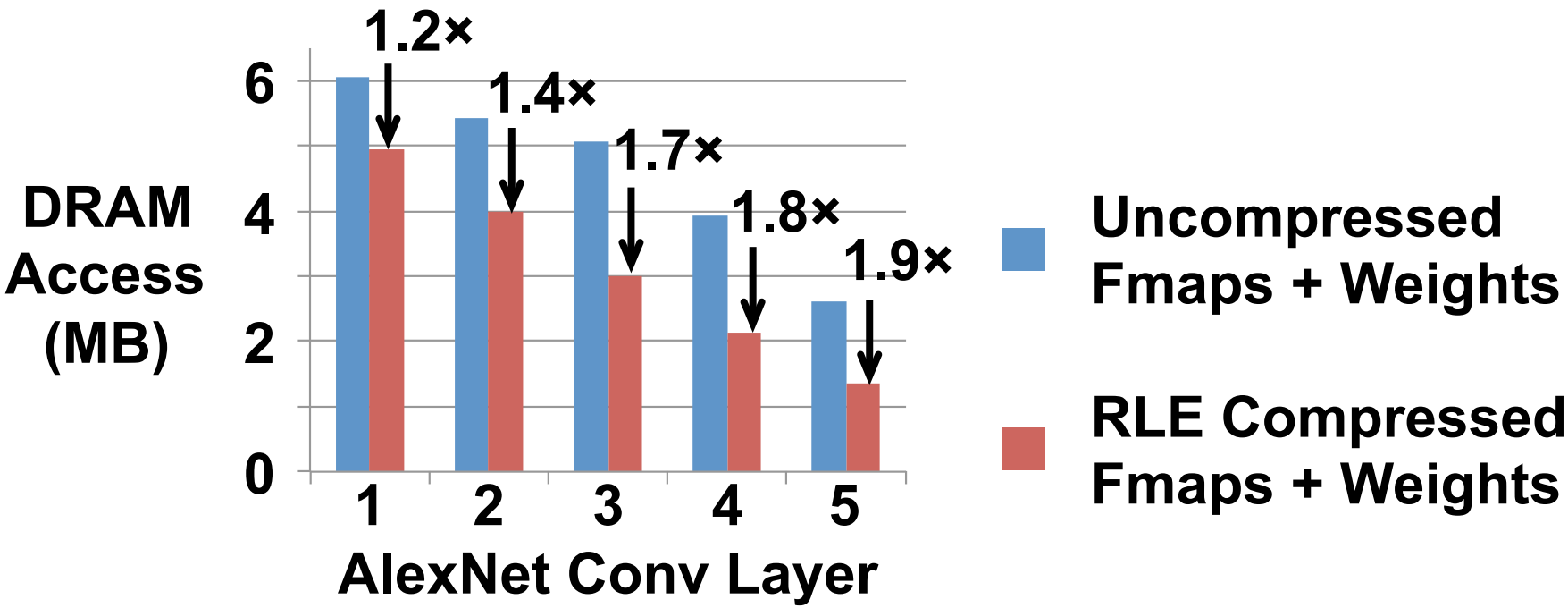


I/O Compression in Eyeriss



[Chen et al., ISSCC 2016]

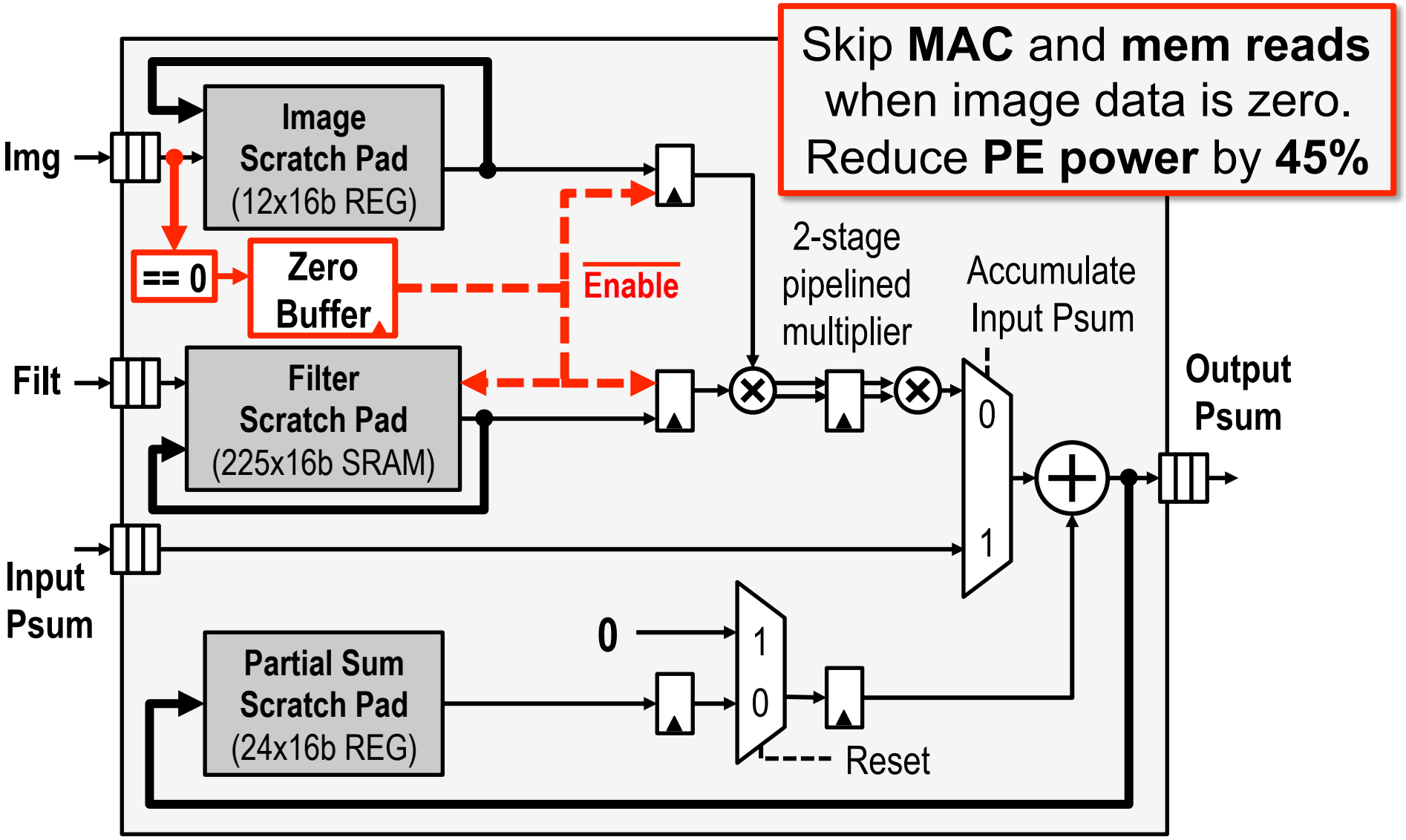
Compression Reduces DRAM BW



Simple RLC within 5% - 10% of theoretical entropy limit

[Chen et al., ISSCC 2016]

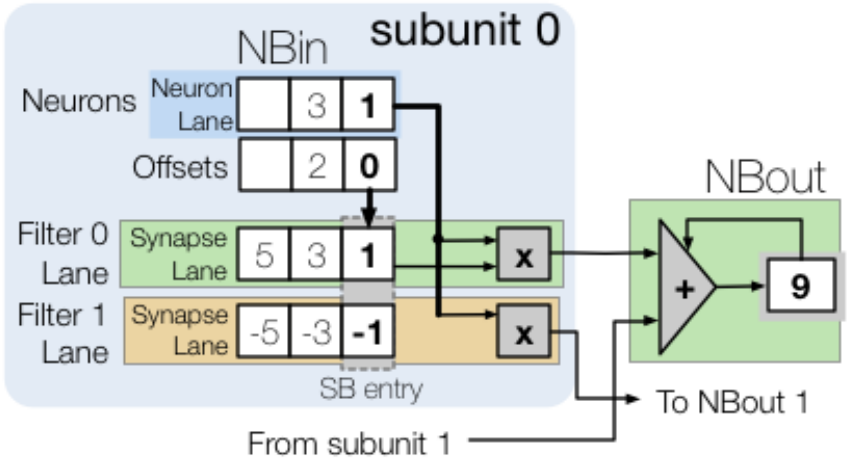
Data Gating / Zero Skipping in Eyeriss

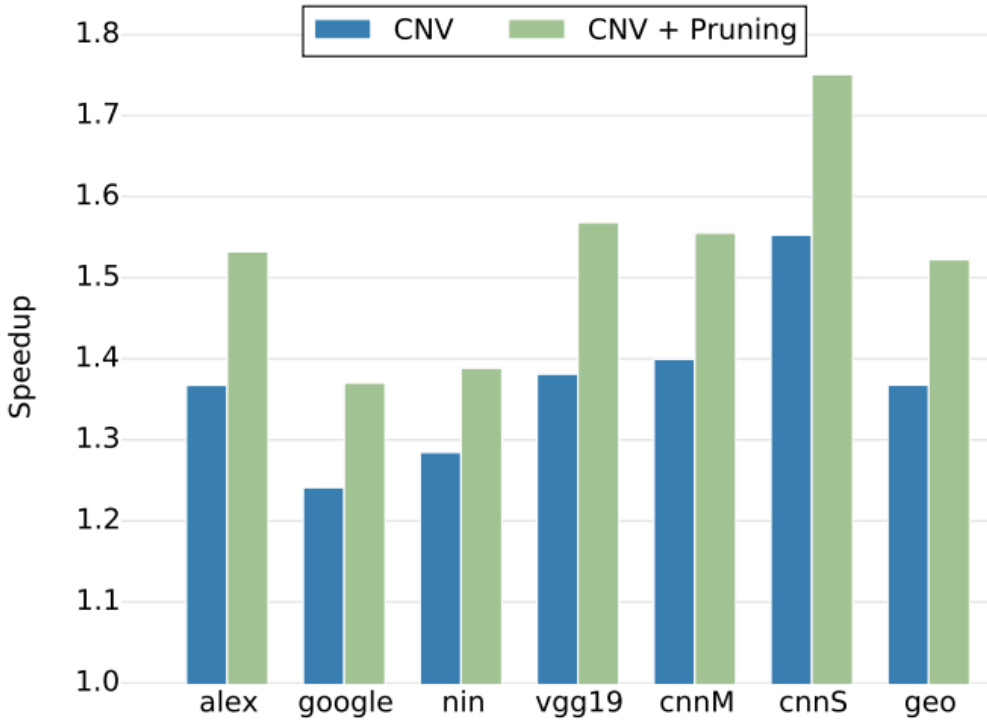


[Chen et al., ISSCC 2016]

Cnvlutin

- **Process Convolution Layers**
- **Built on top of DaDianNao (4.49% area overhead)**
- **Speed up of 1.37x (1.52x with activation pruning)**



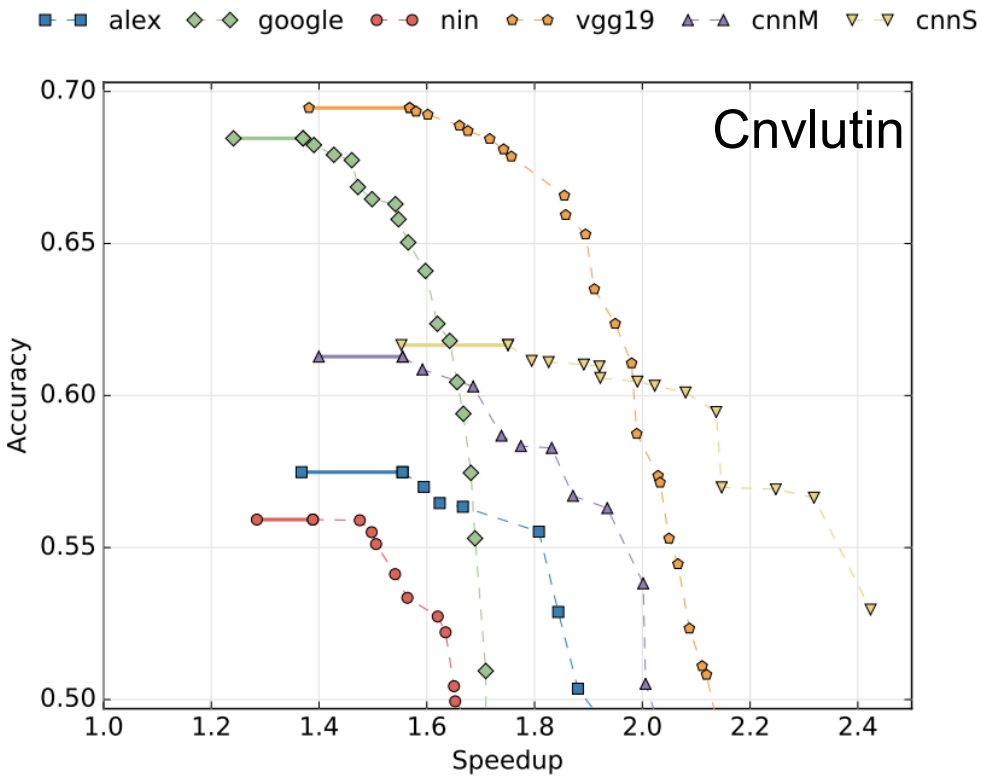


[Albericio et al., ISCA 2016]

Pruning Activations

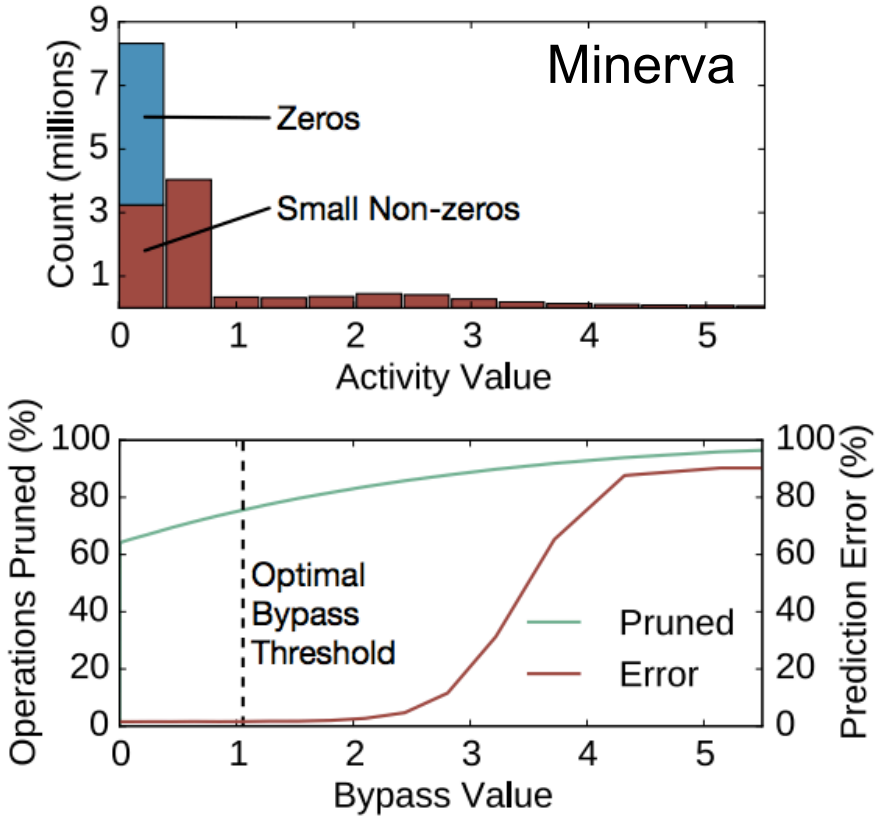
Remove small activation values

Speed up 11% (ImageNet)



[Albericio et al., ISCA 2016]

Reduce power 2x (MNIST)

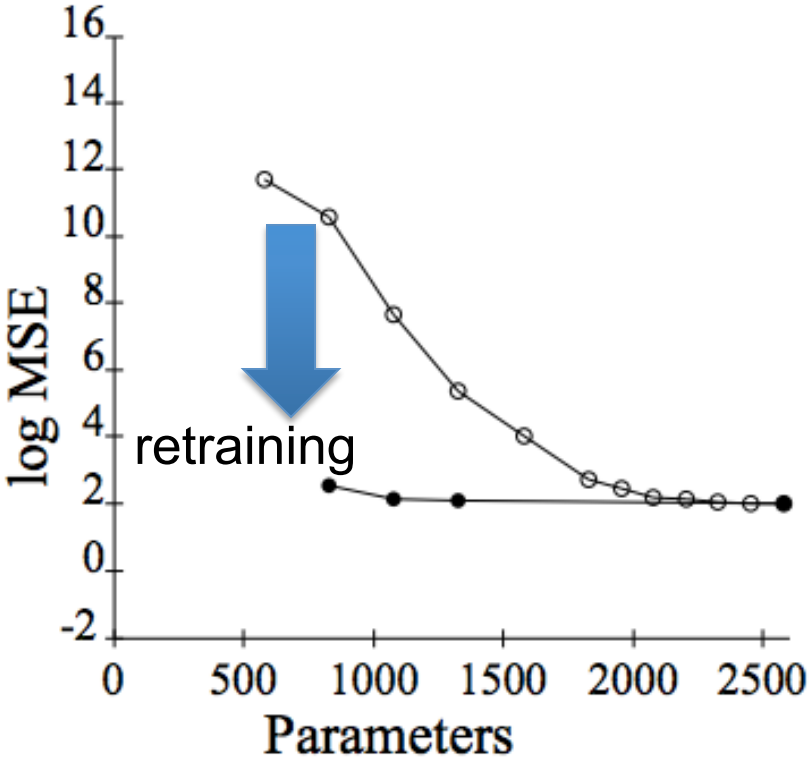


[Reagen et al., ISCA 2016]

Pruning – Make Weights Sparse

- **Optimal Brain Damage**

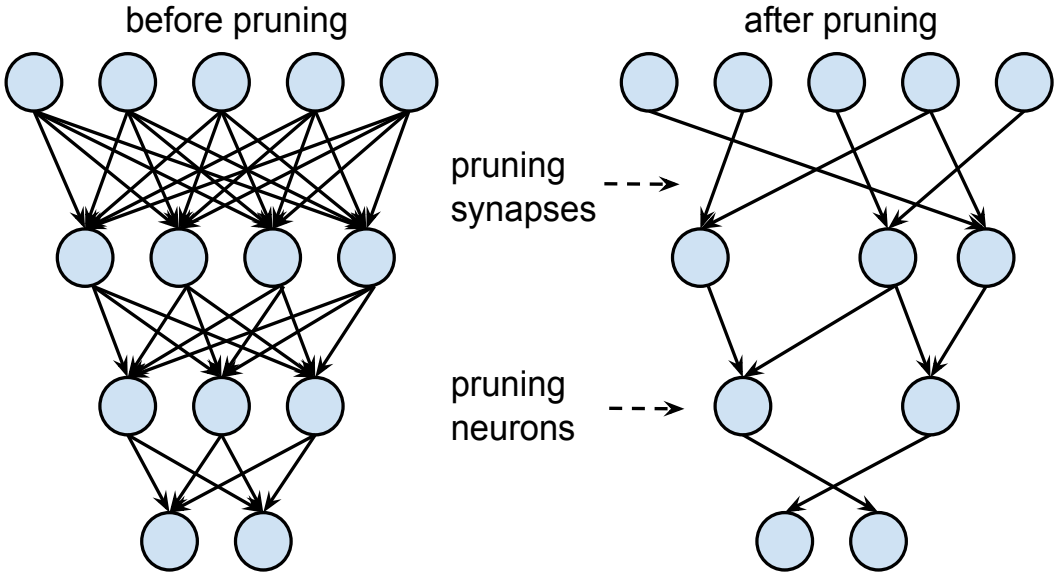
1. **Choose a reasonable network architecture**
2. **Train network until reasonable solution obtained**
3. **Compute the second derivative for each weight**
4. **Compute saliencies (i.e. impact on training error) for each weight**
5. **Sort weights by saliency and delete low-saliency weights**
6. **Iterate to step 2**

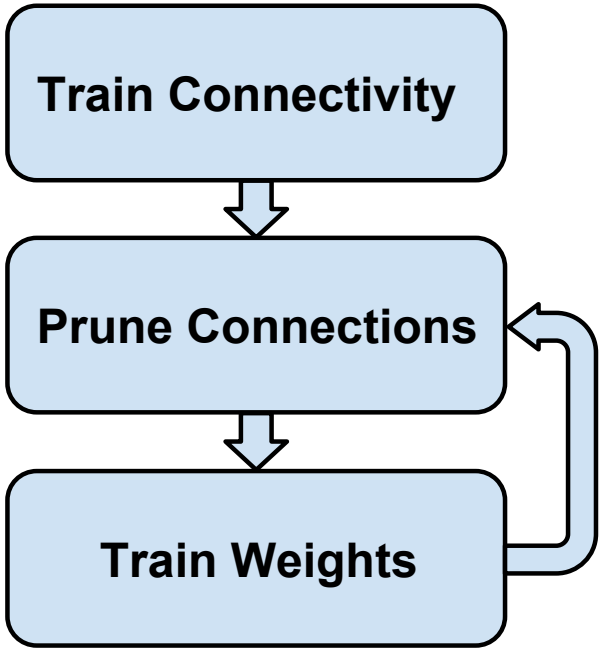


[Lecun et al., NIPS 1989]

Pruning – Make Weights Sparse

Prune based on magnitude of weights

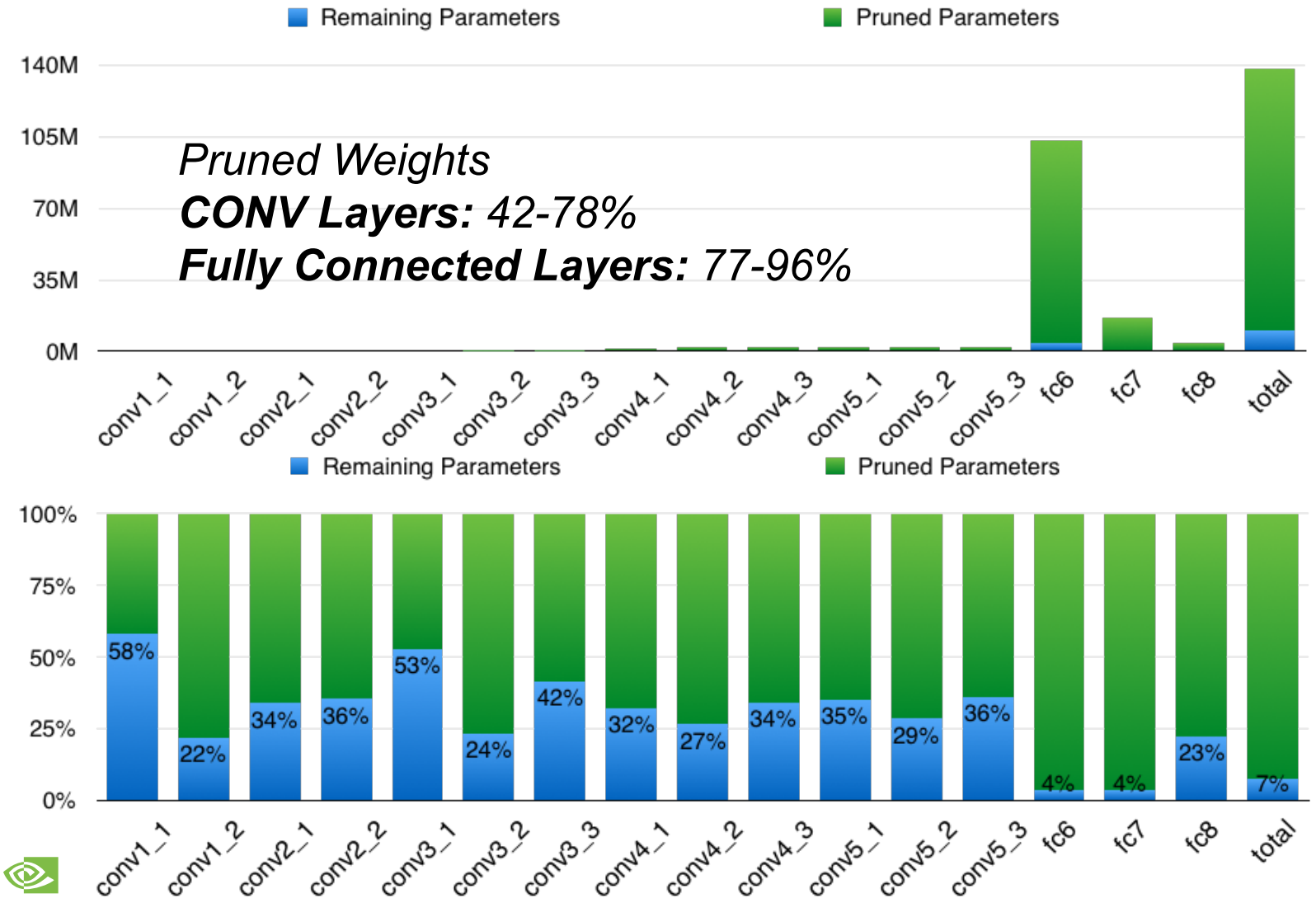




[Han et al., NIPS 2015]

Pruning of VGG-16

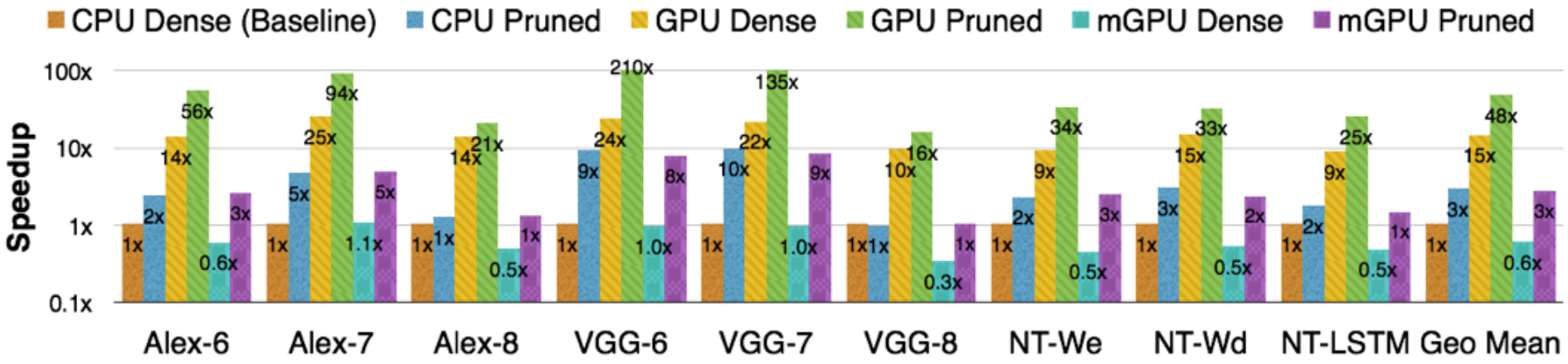
Pruning has most impact on Fully Connected Layers



Speed up of Weight Pruning on CPU/GPU

On Fully Connected Layers

Average Speed up of 3.2x on GPU, 3x on CPU, 5x on mGPU



Intel Core i7 5930K: MKL CBLAS GEMV, MKL SPBLAS CSRMV
NVIDIA GeForce GTX Titan X: cuBLAS GEMV, cuSPARSE CSRMV
NVIDIA Tegra K1: cuBLAS GEMV, cuSPARSE CSRMV

Batch size = 1

[Han et al., NIPS 2015]

Energy-Aware Pruning

- # of Weights alone is not a good metric for energy
  - Example (AlexNet):
    - # of Weights (FC Layer) > # of Weights (CONV layer)
    - Energy (FC Layer) < Energy (CONV layer)
- Us energy evaluation method to estimate DNN energy
  - Account for data movement
- Prune based on energy rather than weights
  - Reduce overall energy (ALL layers) by 3.7x for AlexNet
    - 1.8x more efficient than previous magnitude-based approach
  - 1.6x energy reduction for GoogleNet

[Yang et al., ArXiv 2016]

Compression of Weights & Activations

- **Compress weights and fmaps between DRAM and accelerator**
- **Variable Length / Huffman Coding**

Example:

Value: **16'b0** → Compressed Code: {**1'b0**}

Value: **16'bx** → Compressed Code: {**1'b1**, **16'bx**}

- Tested on AlexNet → **2× overall BW Reduction**

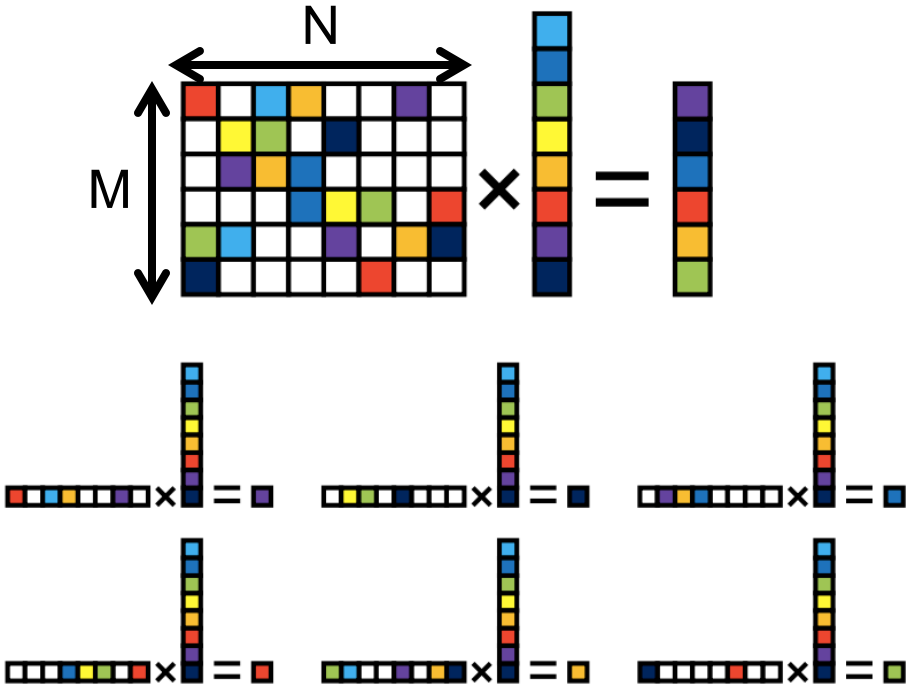
Layer	Filter / Image bits (0%)	Filter / Image BW Reduc.	IO / HuffIO (MB/frame)	Voltage (V)	MMACs/ Frame	Power (mW)	Real (TOPS/W)
General CNN	16 (0%) / 16 (0%)	1.0x		1.1	–	**288**	**0.3**
AlexNet l1	7 (21%) / 4 (29%)	1.17x / 1.3x	1 / 0.77	0.85	105	85	0.96
AlexNet l2	7 (19%) / 7 (89%)	1.15x / **5.8x**	3.2 / 1.1	0.9	224	55	1.4
AlexNet l3	8 (11%) / 9 (82%)	1.05x / 4.1x	6.5 / 2.8	0.92	150	77	0.7
AlexNet l4	9 (04%) / 8 (72%)	1.00x / 2.9x	5.4 / 3.2	0.92	112	95	0.56
AlexNet l5	9 (04%) / 8 (72%)	1.00x / 2.9x	3.7 / 2.1	0.92	75	95	0.56
Total / avg.	–	–	19.8 / **10**	–	–	**76**	**0.94**
LeNet-5 l1	3 (35%) / 1 (87%)	1.40x / 5.2x	0.003 / 0.001	0.7	0.3	25	1.07
LeNet-5 l2	4 (26%) / 6 (55%)	1.25x / 1.9x	0.050 / 0.042	0.8	1.6	35	1.75
Total / avg.	–	–	0.053 / **0.043**	–	–	**33**	**1.6**

[Moons et al., VLSI 2016; Han et al., ICLR 2016]

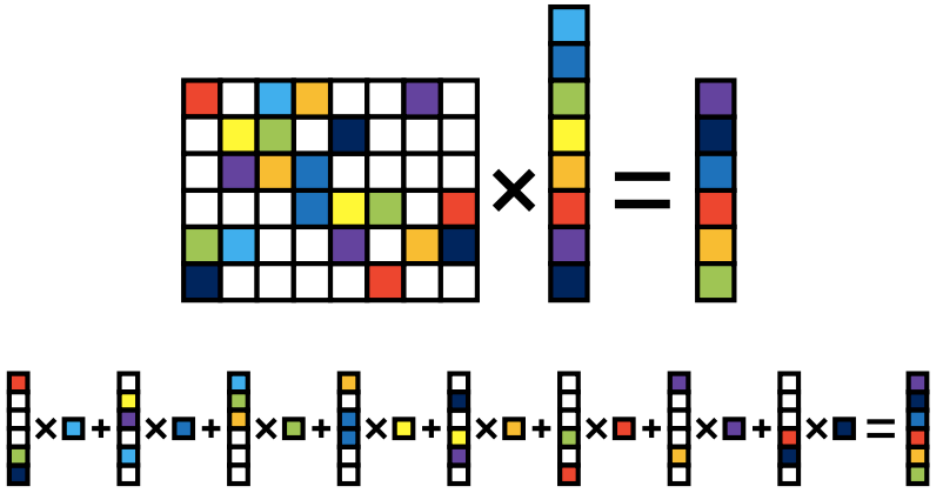
Sparse Matrix-Vector DSP

- **Use CSC rather than CSR for SpMxV**

Compressed Sparse Row (CSR)



Compressed Sparse Column (CSC)

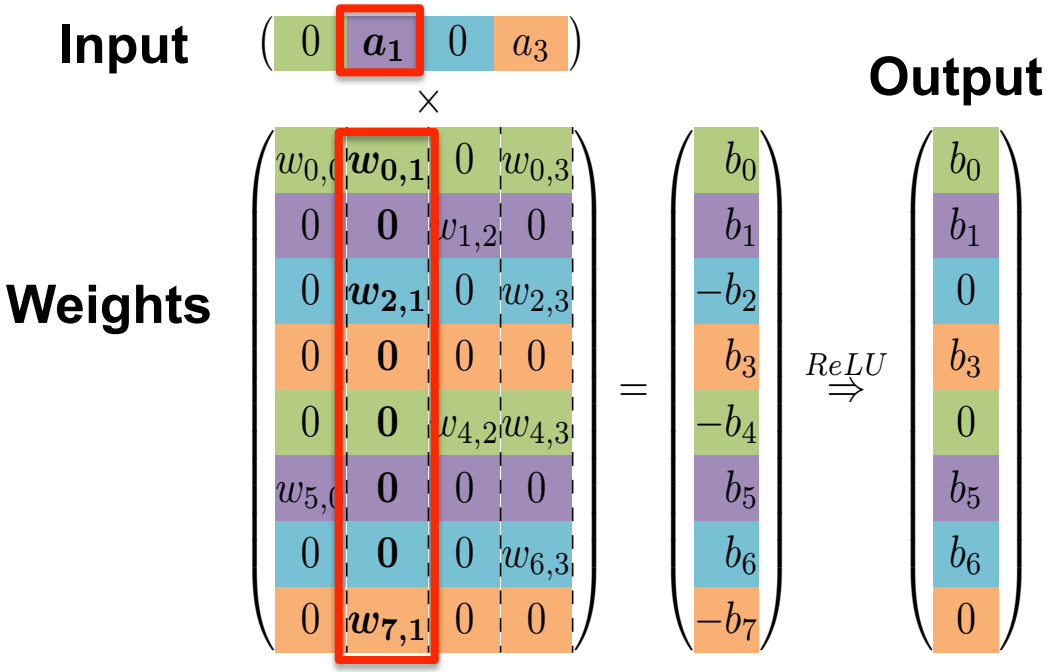


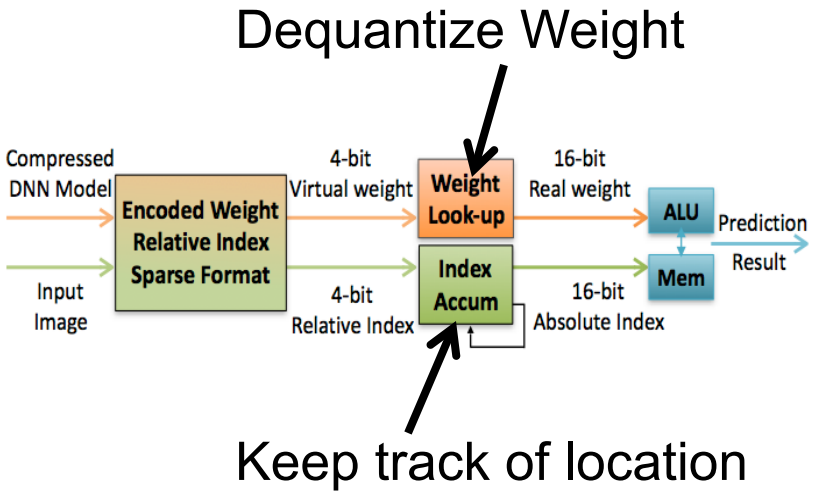
Reduce memory bandwidth by 2x (when not M >> N)

[Dorrance et al., FPGA 2014]

EIE: A Sparse Linear Algebra Engine

- **Process Fully Connected Layers (after Deep Compression)**
- **Store weights column-wise in Run Length format**
  - **Non-zero weights, Run-length of zeros**
  - **Start location of each column since variable length**
- **Read relative column when input is non-zero**





[Han et al., ISCA 2016]

Network Architecture

Reduce size and computation with 1x1 Filter

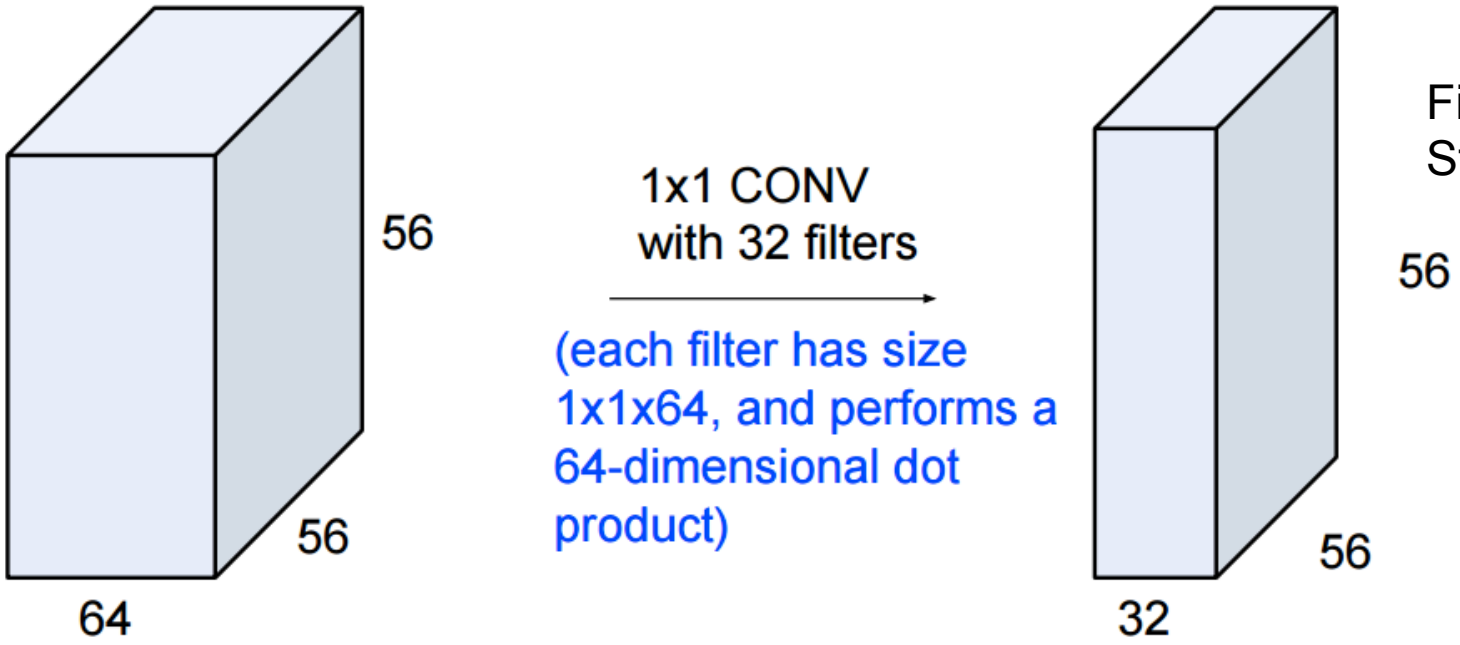


Figure Source:
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

Network Architecture

Reduce size and computation with 1x1 Filter

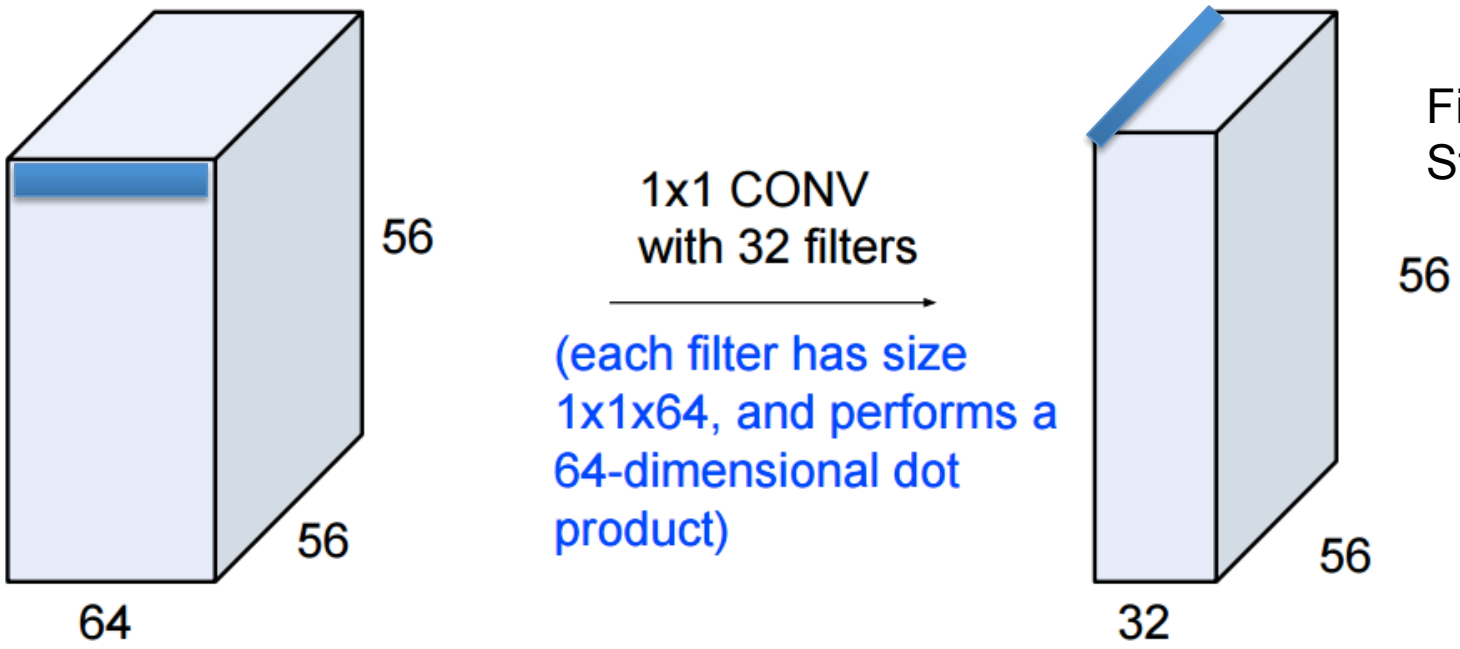


Figure Source:
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

Network Architecture

Reduce size and computation with 1x1 Filter

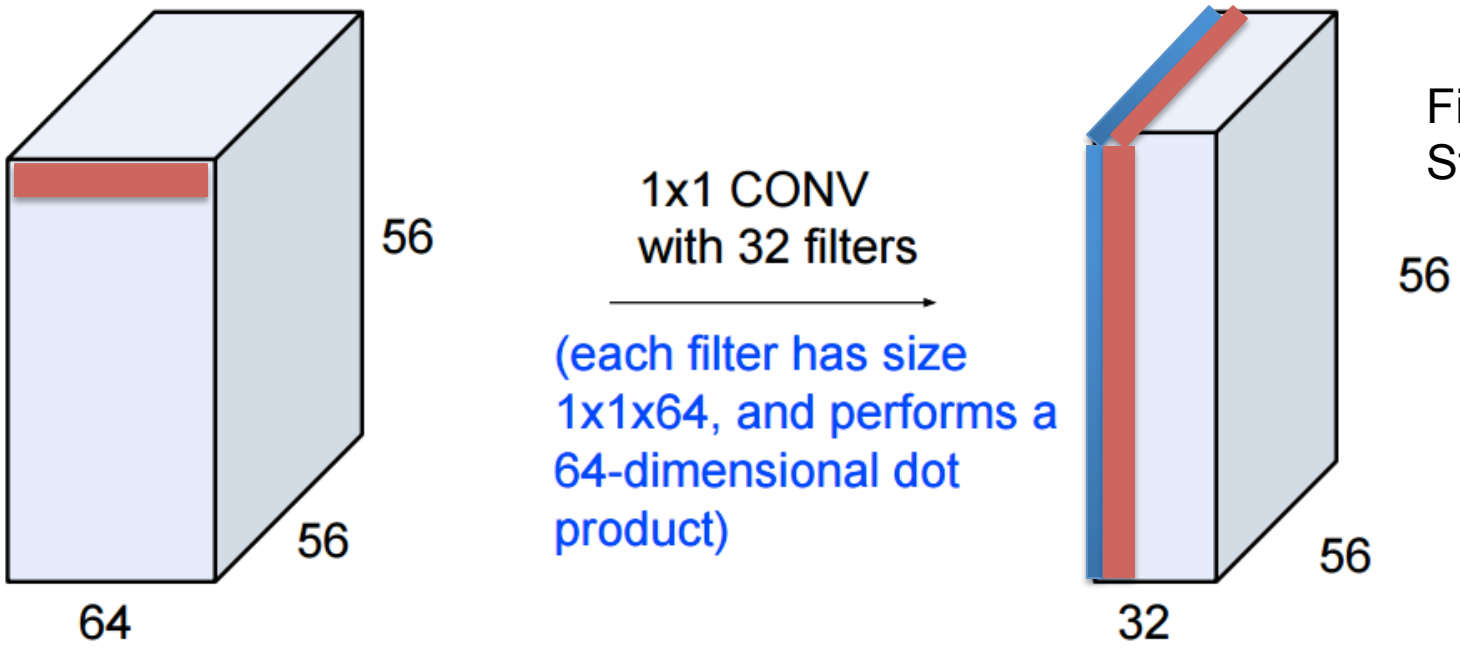
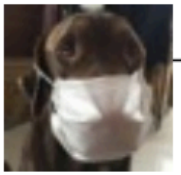


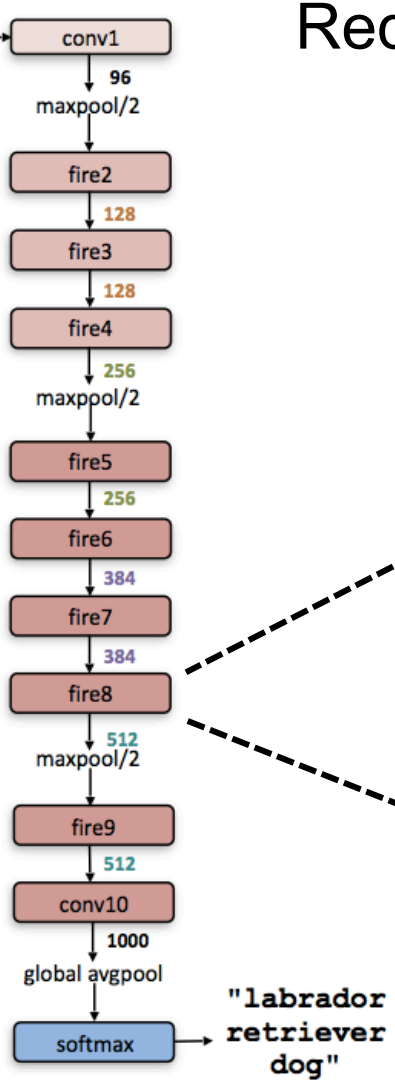
Figure Source:
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

SqueezeNet

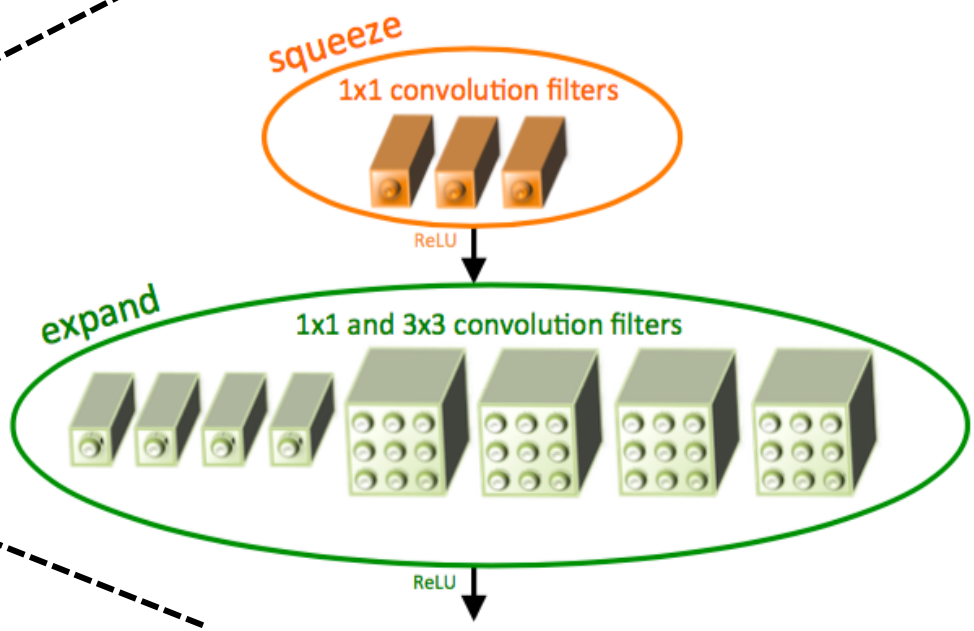




Reduce weights by reducing number of input channels by "squeezing" with 1x1

50x fewer weights than AlexNet

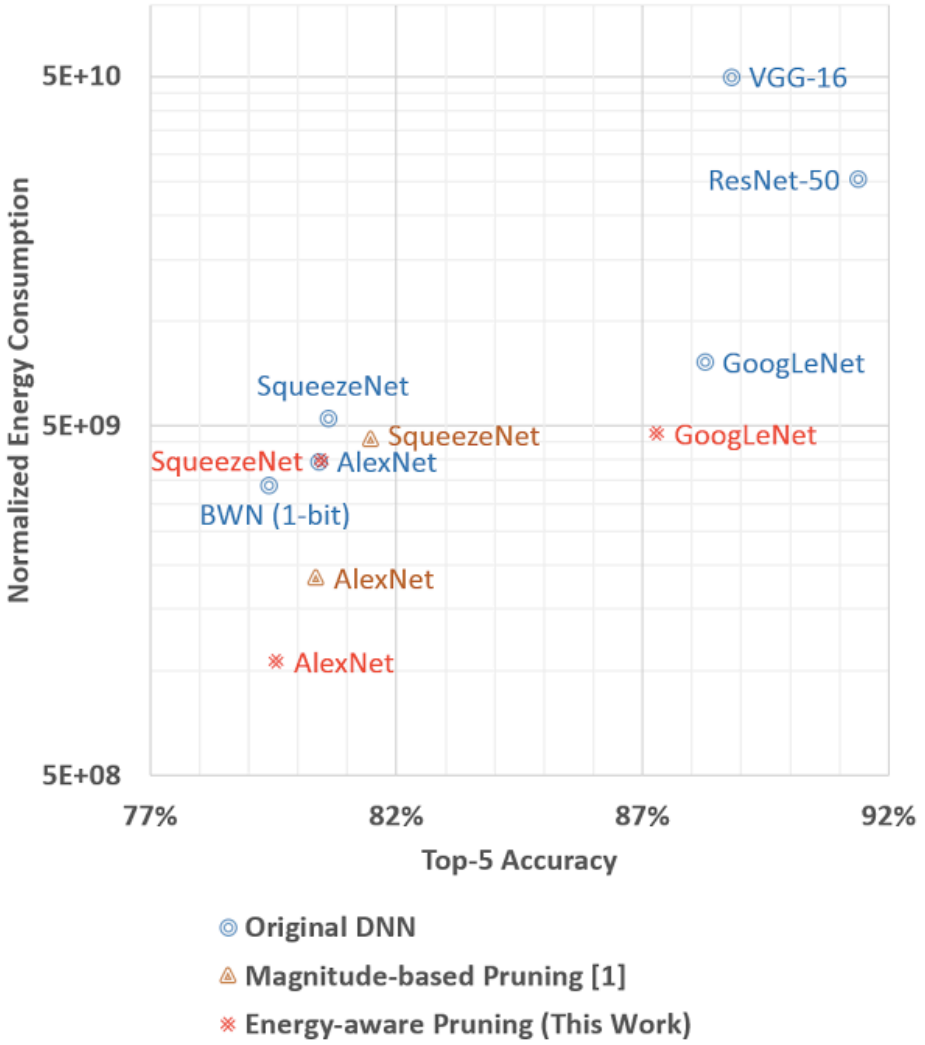
Fire Module



[F.N. Iandola et al., ArXiv, 2016]

Energy Consumption of Existing DNNs

- **Maximally reducing # of weights does not necessarily result in optimized energy consumption**
- **Deeper CNNs with fewer weights (e.g. GoogleNet, SqueezeNet), do not necessarily consume less energy than shallower CNNs with more weights (e.g. AlexNet)**
- **Reducing # of weights can provide equal or more reduction than reducing the bitwidth of weights (e.g. BWN)**



[Yang et al., ArXiv 2016]

Benchmarking Metrics for DNN Hardware

MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

Metrics Overview

- How can we compare designs?
- Target Metrics
  - Accuracy
  - Power
  - Throughput
  - Cost
- Additional Factors
  - External memory bandwidth
  - Required on-chip storage
  - Utilization of cores

Download Benchmarking Data

- Input (http://image-net.org/)
  - Sample subset from ImageNet Validation Dataset

- Widely accepted state-of-the-art DNNs (Model Zoo: http://caffe.berkeleyvision.org/)
  - AlexNet
  - VGG-16
  - GoogleNet-v1
  - ResNet-50

Metrics for DNN Algorithm

- Accuracy
- Network Architecture
  - # Layers, filter size, # of filters, # of channels
- # of Weights (storage capacity)
  - Number of non-zero (NZ) weights
- # of MACs (operations)
  - Number of non-zero (NZ) MACS

Metrics of DNN Algorithms

Metrics	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Accuracy (top-5 error)*	19.8	8.80	10.7	7.02
Input	227x227	224x224	224x224	224x224
# of CONV Layers	**5**	**16**	**21**	**49**
Filter Sizes	3, 5,11	3	1, 3 , 5, 7	1, 3, 7
# of Channels	3 - 256	3 - 512	3 - 1024	3 - 2048
# of Filters	96 - 384	64 - 512	64 - 384	64 - 2048
Stride	1, 4	1	1, 2	1, 2
# of Weights	2.3M	14.7M	6.0M	23.5M
# of MACs	666M	15.3G	1.43G	3.86G
# of FC layers	**3**	**3**	**1**	**1**
# of Weights	58.6M	124M	1M	2M
# of MACs	58.6M	124M	1M	2M
Total Weights	**61M**	**138M**	**7M**	**25.5M**
Total MACs	**724M**	**15.5G**	**1.43G**	**3.9G**

*Single crop results: https://github.com/jcjohnson/cnn-benchmarks

Metrics of DNN Algorithms

Metrics	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Accuracy (top-5 error)*	19.8	8.80	10.7	7.02
# of CONV Layers	**5**	**16**	**21**	**49**
# of Weights	2.3M	14.7M	6.0M	23.5M
# of MACs	666M	15.3G	1.43G	3.86G
# of NZ MACs**	**394M**	**7.3G**	**806M**	**1.5G**
# of FC layers	**3**	**3**	**1**	**1**
# of Weights	58.6M	124M	1M	2M
# of MACs	58.6M	124M	1M	2M
# of NZ MACs**	**14.4M**	**17.7M**	**639k**	**1.8M**
Total Weights	**61M**	**138M**	**7M**	**25.5M**
Total MACs	**724M**	**15.5G**	**1.43G**	**3.9G**
# of NZ MACs**	**409M**	**7.3G**	**806M**	**1.5G**

*Single crop results: https://github.com/jcjohnson/cnn-benchmarks

**# of NZ MACs computed based on 50,000 validation images

Metrics of DNN Algorithms

Metrics	AlexNet	AlexNet (sparse)
Accuracy (top-5 error)	19.8	19.8
# of Conv Layers	**5**	**5**
# of Weights	2.3M	2.3M
# of MACs	666M	666M
# of NZ weights	**2.3M**	**863k**
# of NZ MACs	**394M**	**207M**
# of FC layers	**3**	**3**
# of Weights	58.6M	58.6M
# of MACs	58.6M	58.6M
# of NZ weights	**58.6M**	**5.9M**
# of NZ MACs	**14.4M**	**2.1M**
Total Weights	**61M**	**61M**
Total MACs	**724M**	**724M**
# of NZ weights	**61M**	**6.8M**
# of NZ MACs	**409M**	**209M**

# of NZ MACs computed based on 50,000 validation images

Metrics for DNN Hardware

- **Measure energy and DRAM access relative to number of non-zero MACs and bit-width of MACs**
  - **Account for impact of sparsity in weights and activations**
  - **Normalize DRAM access based on operand size**
- **Energy Efficiency of Design**
  - **pJ/(non-zero weight & activation)**
- **External Memory Bandwidth**
  - **DRAM operand access/(non-zero weight & activation)**
- **Area Efficiency**
  - **Total chip mm^2/multi (also include process technology)**
  - **Accounts for on-chip memory**

ASIC Benchmark (e.g. Eyeriss)

ASIC Specs	
Process Technology	65nm LP TSMC (1.0V)
Total core area (mm^2) /total # of multiplier	0.073
Total on-Chip memory (kB) / total # of multiplier	1.14
Measured or Simulated	Measured
If Simulated, Syn or PnR? Which corner?	n/a

ASIC Benchmark (e.g. Eyeriss)

Layer by layer breakdown for AlexNet CONV layers

Metric	Units	L1	L2	L3	L4	L5	Overall*
Batch Size	#	4					
Bit/Operand	#	16					
Energy/ non-zero MACs (weight & act)	pJ/MAC	16.5	18.2	29.5	41.6	32.3	21.7
DRAM access/ non-zero MACs	Operands/ MAC	0.006	0.003	0.007	0.010	0.008	0.005
Runtime	ms	20.9	41.9	23.6	18.4	10.5	115.3
Power	mW	332	288	266	235	236	278

* Weighted average of CONV layers

Website to Summarize Results

- http://eyeriss.mit.edu/benchmarking.html
- Send results or feedback to: eyeriss@mit.edu

ASIC Specs	Input
Process Technology	65nm LP TSMC (1.0V)
Chip area (mm^2) / multiplier	0.095
On-Chip memory (kB) / multiplier	1.14
Measured or Simulated	Measured
If Simulated, Syn or PnR? Which corner?	n/a

Metric	Units	Input
Name of CNN	Text	AlexNet
# of Images Tested	#	100
Bits per operand	#	16
Batch Size	#	4
# of Non Zero MACs	#	409M
Runtime	ms	115.3
Power	mW	278
Energy/non-zero MACs	**pJ/MAC**	**21.7**
DRAM access/non-zero MACs	**operands /MAC**	**0.005**

Implementation-Specific Metrics

Different devices may have implementation-specific metrics

Example: FPGAs

Metric		Units	AlexNet
Device		Text	Xilinx Virtex-7 XC7V690T
Utilization	DSP	#	2,240
	BRAM	#	1,024
	LUT	#	186,251
	FF	#	205,704
Performance Density		GOPs/slice	8.12E-04

Hardware for Training

MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

Cost function for Model Training

Model output:

$$\mathbf{y} = f(\mathbf{x})$$

Desired output:

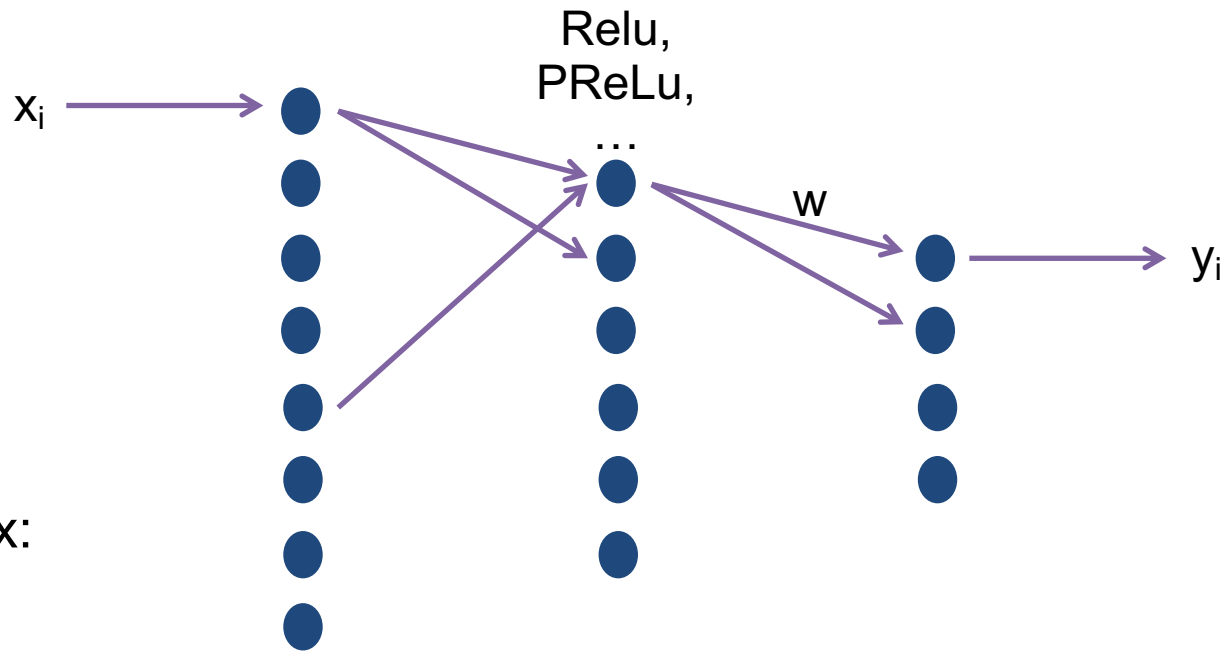
$$\mathbf{z}$$

Error:

$$e = (\mathbf{y}-\mathbf{z})$$

Over all training inputs x:

$$\text{Minimize } \Sigma(\mathbf{y}-\mathbf{z})^2$$



What do we vary to minimize the error?

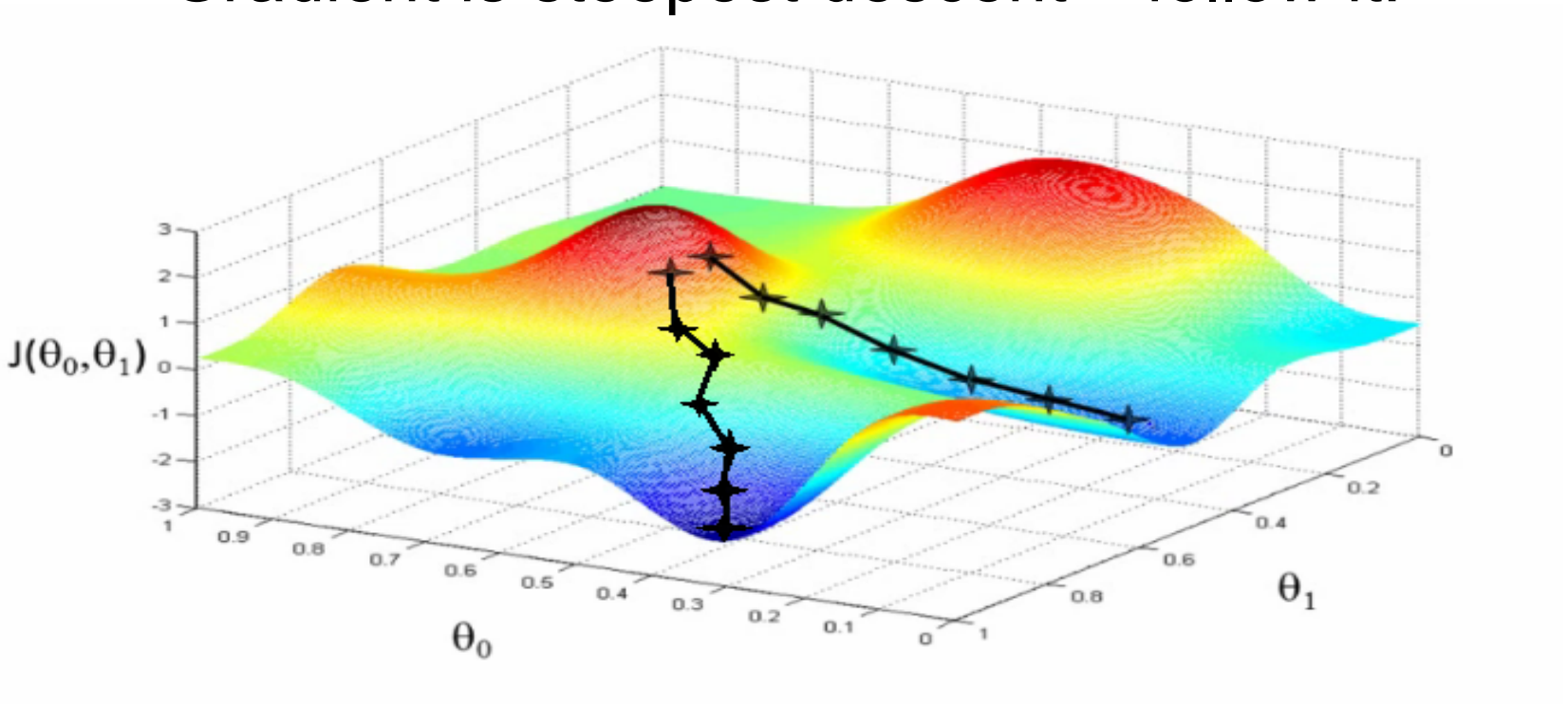
Training Optimization Problem

- **Model parameters** θ **(include bias, weights, …)**
- **Model output** $y(\theta) = f(x, \theta)$
- **Desired output** z
- **Error** $e(\theta) = y(\theta) - z$
- **Cost function*** $E(\theta) = \Sigma e(\theta)^2$
- **Minimization** $dE(\theta)/d\theta = 0$ **(but no closed form)**

* Over all inputs in the training set

Steepest descent

Classical first order iterative optimization scheme:
Gradient is steepest descent – follow it!



$\theta^{n+1} = \theta^n - \alpha \cdot dE(\theta^n)/d\theta$

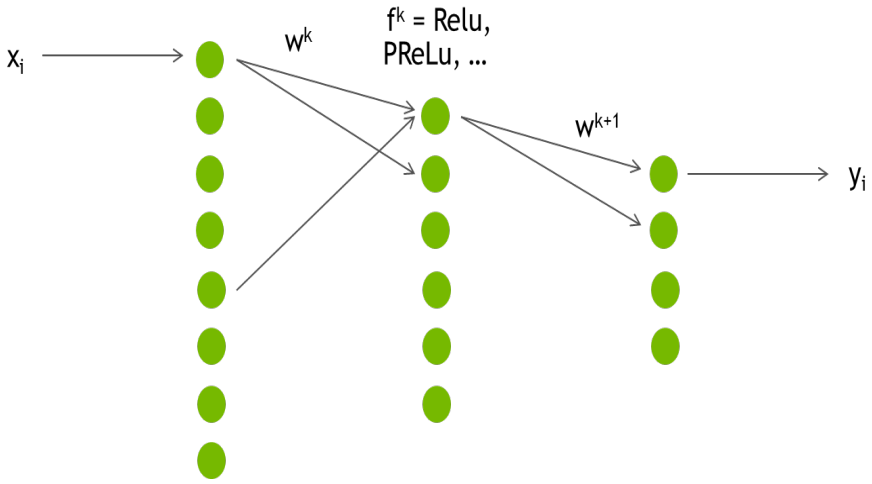
where α is the step size along the gradient…

Calculating Steepest Descent

Also called error back-propagation

- **Steepest descent**

  $\theta^{n+1} = \theta^{n} - \alpha \cdot dE(\theta)/d\theta$



- $E(\theta) = \Sigma e(\theta)^2$

  $= \Sigma (y(\theta)-z)^2$

- $dE(\theta)/d\theta = 2 \cdot \Sigma[(y(\theta)-z) \cdot dy(\theta)/d\theta]$

error **e**

back-propagation

Chain rule -> Back propagation

- The chain rule of calculus allows one to calculate the derivative of a layered network, i.e., a composition of functions, iteratively working backwards through the layers using the (feature map) values of the layer, i.e., function, and the derivative from the next layer.

- Back propagation is the process of doing this calculation numerically for a given input.

Per Layer Calculations

$$\mathbf{y} = f(\mathbf{x})$$

For layer k:

Inputs: x^k

Weights: w^k

Outputs: y^k

So

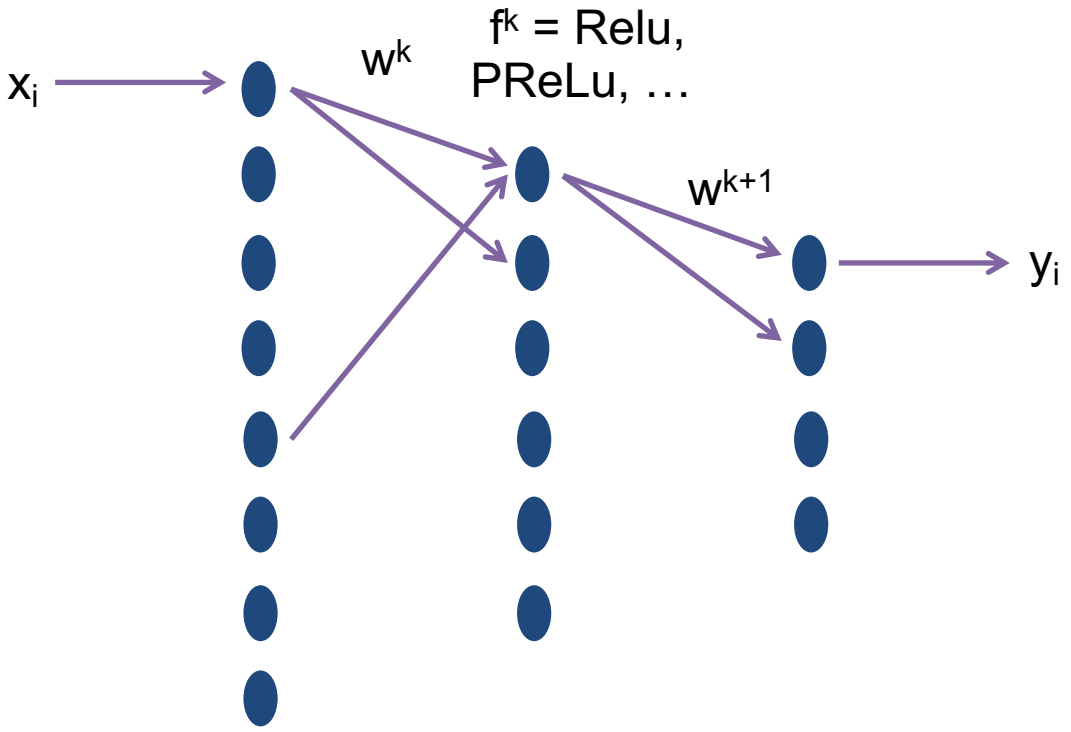
$$y_i^k = f^k [\Sigma(w_{ij}^k x_j^k)]$$

Where

$$x_j^k = y_j^{k-1}$$

or

$$\mathbf{y}^k = f^k(\mathbf{y}^{k-1}, \theta)$$



Layer Operation Composition

- **Steepest descent** $\theta^{k+1} = \theta^k - \alpha \cdot dE/d\theta$
- **Derivative (1)** $dE/d\theta = 2 \cdot \Sigma[(y(\theta)-z) \cdot dy(\theta)/d\theta]$
- **Model output** $y = f(x)$

  $y^n = f^n(y^{n-1}) = f^n(f^{n-1}(y^{n-2}))$
- **Layer k** $y^k = f^k(y^{k-1}) = f^k(f^{k-1}(y^{k-2}))$

Chain rule

- **Chain rule for functions**

$$y = f(g(x)) \qquad y' = f'(g(x)) * g'(x)$$

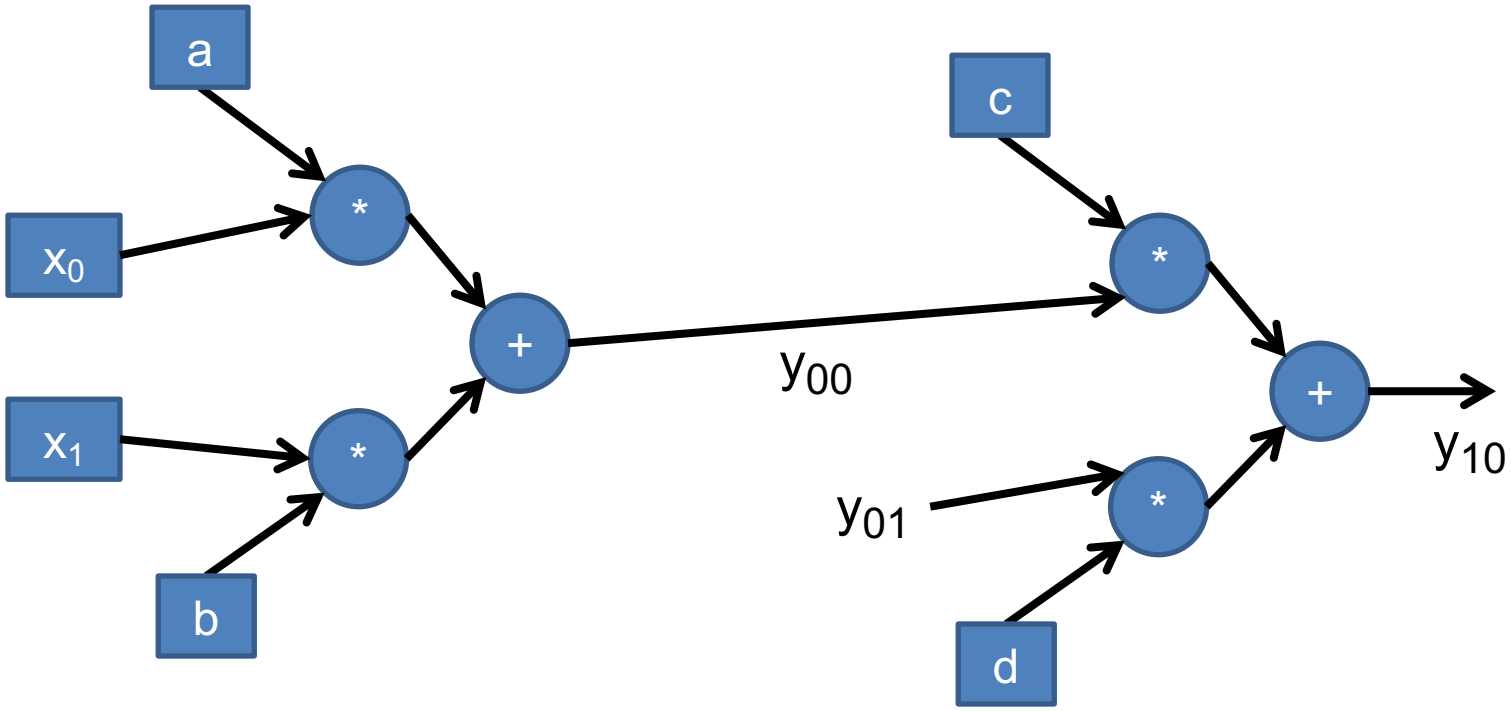
$$y = f^n(y^{n-1}) = f^n(f^{n-1}(y^{n-2})) \qquad y' = f^{n'}(f^{n-1}(y^{n-2})) * f^{n-1'}(y^{n-2})$$
$$= f^{n'}(y^{n-1}) * f^{n-1'}(y^{n-2})$$

Back propagation

- $y_{00} = (a*x_0 + b*x_1)$

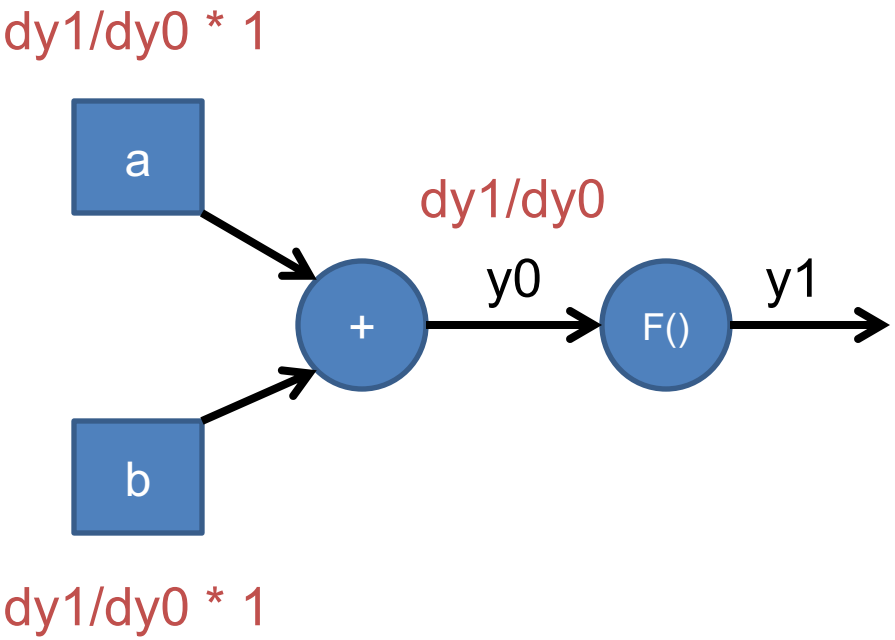
  $y_{01} = \ldots.$

  $y_{10} = y_{00}*c + y_{01}*d$
- $dy_{10}/da = dy_{10}/dy_{00} * dy_{00}/da = c * x_0$



Back Propagation for Addition

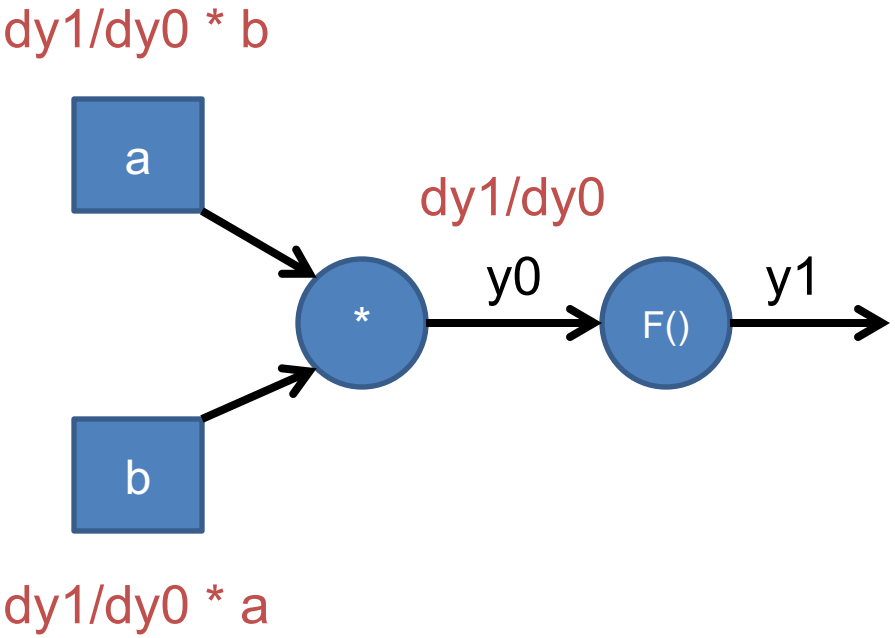
- $y_0 = a + b$
- $y_1 = f(y_0)$



- $dy_0/da = 1$
- $dy_0/db = 1$
- $dy_1/dy_0 = f'(y_0)$
- $dy_1/da = dy_1/dy_0 * dy_0/da = dy_1/dy_0 * 1 = dy_1/dy_0$
- $dy_1/db = dy_1/dy_0 * dy_0/db = dy_1/dy_0 * 1 = dy1/dy0$

Back Propagation for Multiplication

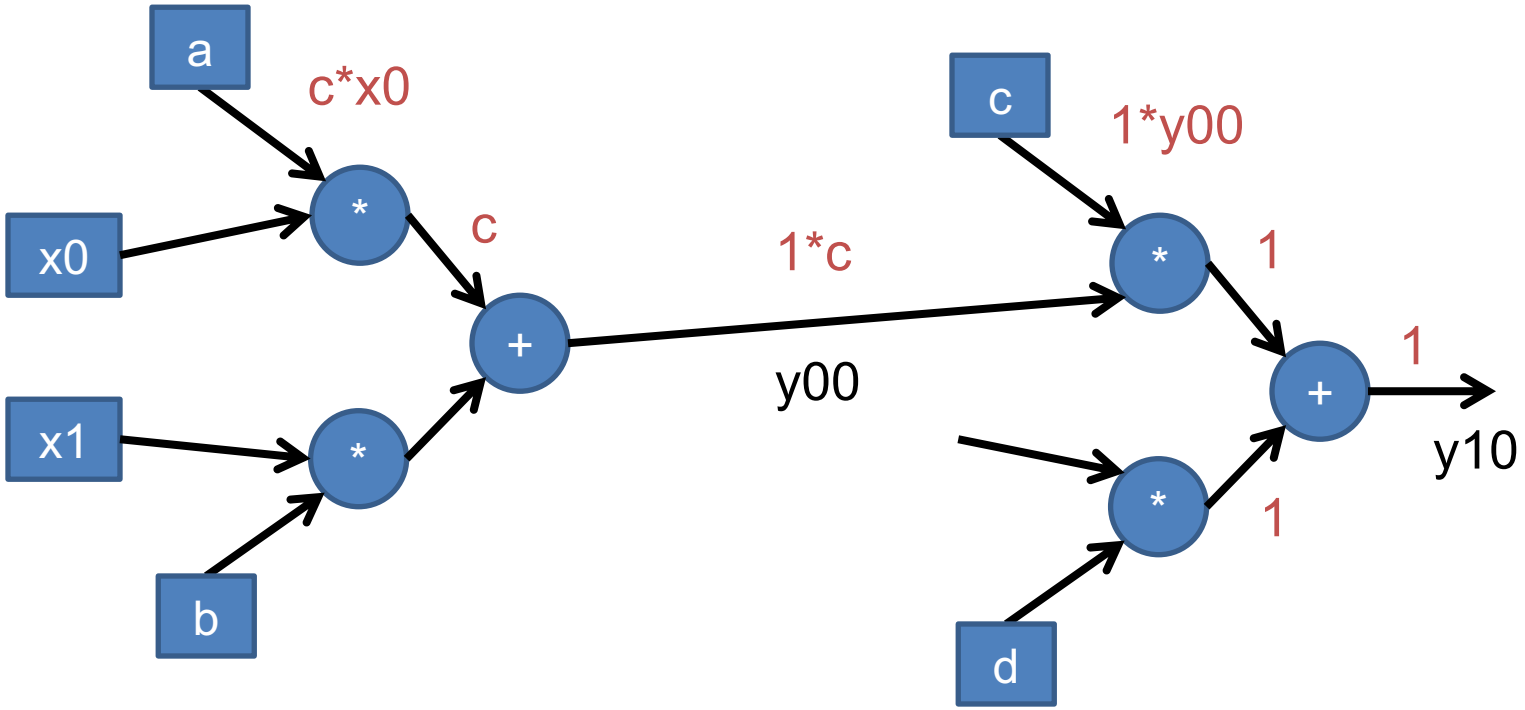
- y0 = a * b
- y1 = f(y0)



- dy0/da = b
- dy0/db = a
- dy1/dy0 = f'(y0)
- dy1/da = dy1/dy0 * dy0/da = dy1/dy0 * b
- dy1/db = dy1/dy0 * dy0/db = dy1/dy0 * a

Back propagation for Network

- y00 = (a*x0 + b*x1)
  y01 = ….
  y10 = y00*c + y01*d
- dy10/da = dy10/dy00 * dy00/da = c * x0



Back Propagation Recipe

Start point

- Select a initial set of weights (θ) and an input (x)

Forward pass

- For all layers
  - Compute layer outputs use as input for next layer (and save for later)

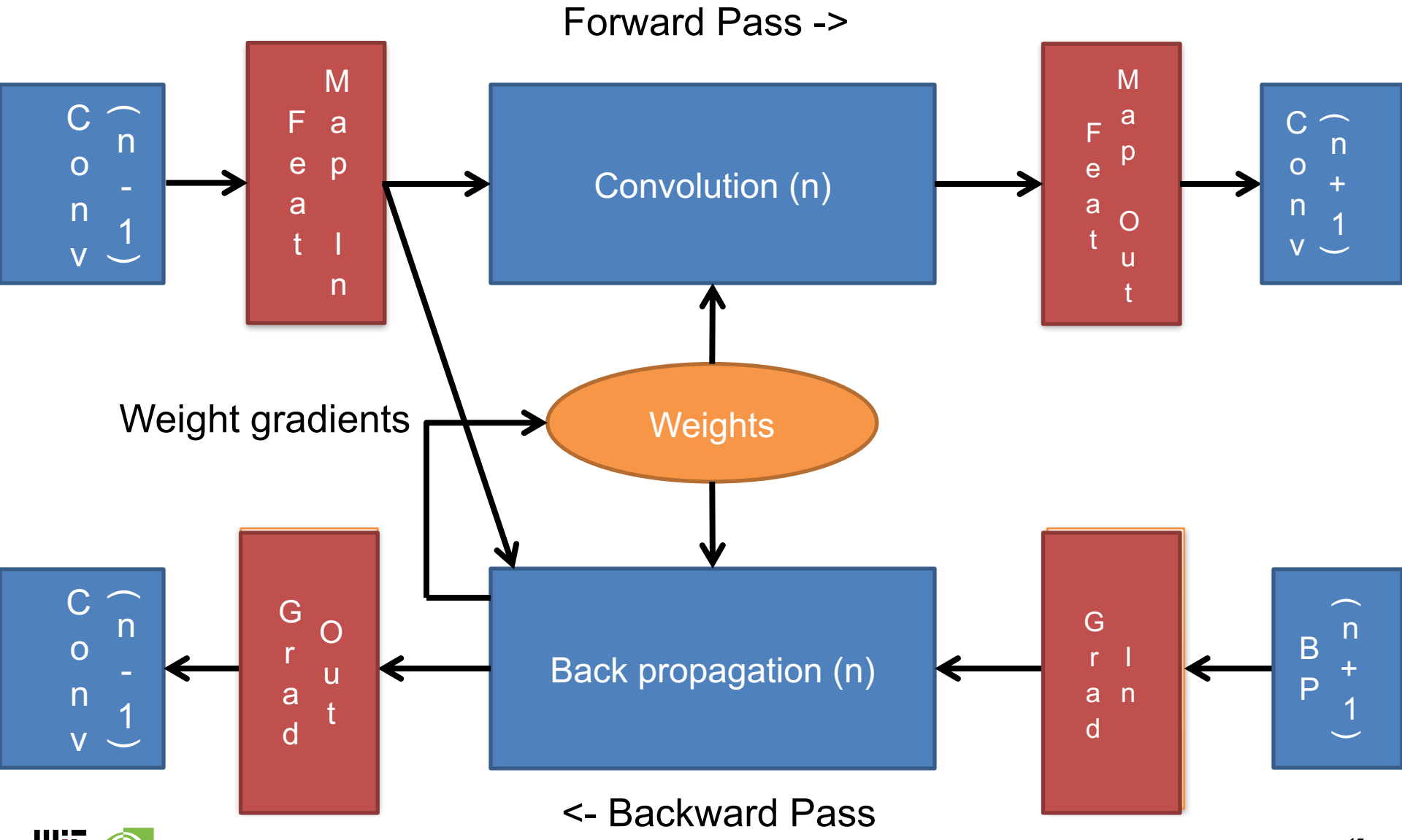
Backward pass

- For all layers (with output of previous layer and gradient of next layer)
  - Compute gradient, i.e., (partial) derivative, for layer
  - Back-propagate gradient to previous layer
  - Compute (partial) derivatives for (local) weights of layer

Calculate next set of weights

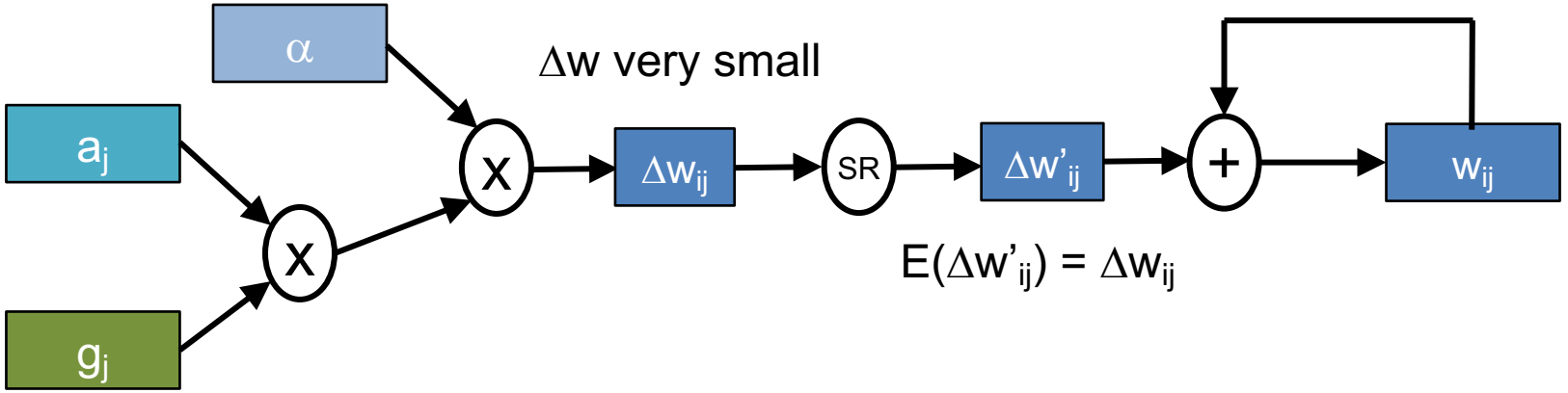
- $\theta^{k+1} = \theta^k - \alpha \cdot dE/d\theta$

Back Propagation



Precision on Training

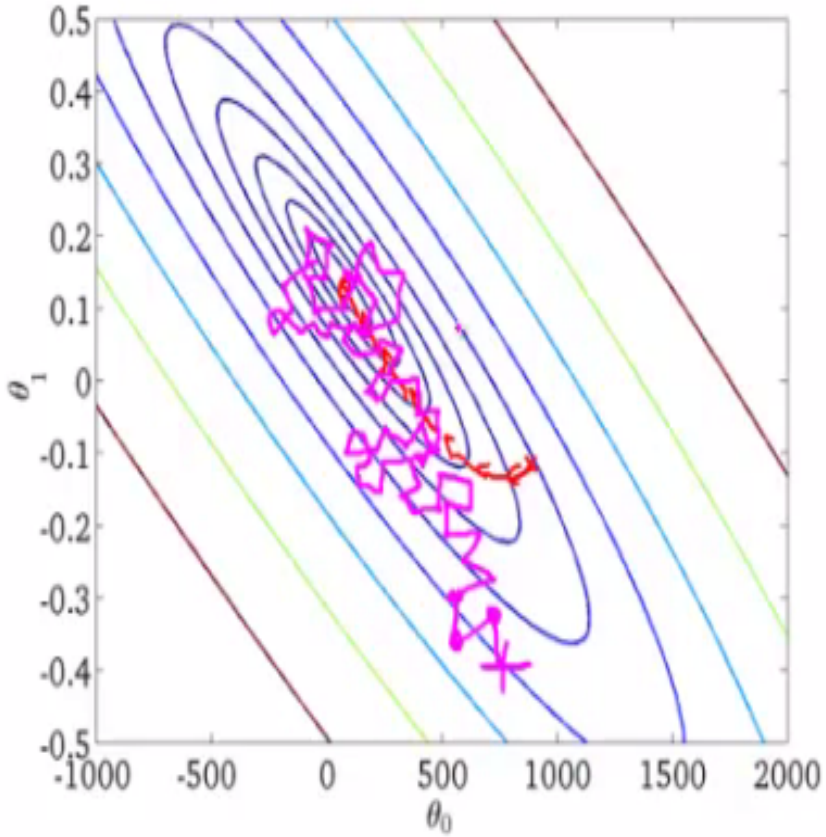
Learning rate may be very small (10^{-5} or less)



- Beware truncating changes to zero
- Rounding can bias result -> use stochastic rounding

[Gupta et al., ICML 2015]

Back Propagation Batches



Issue:

- **N = 1 is often too noisy, weights may oscillate around the minimum**

Solution:

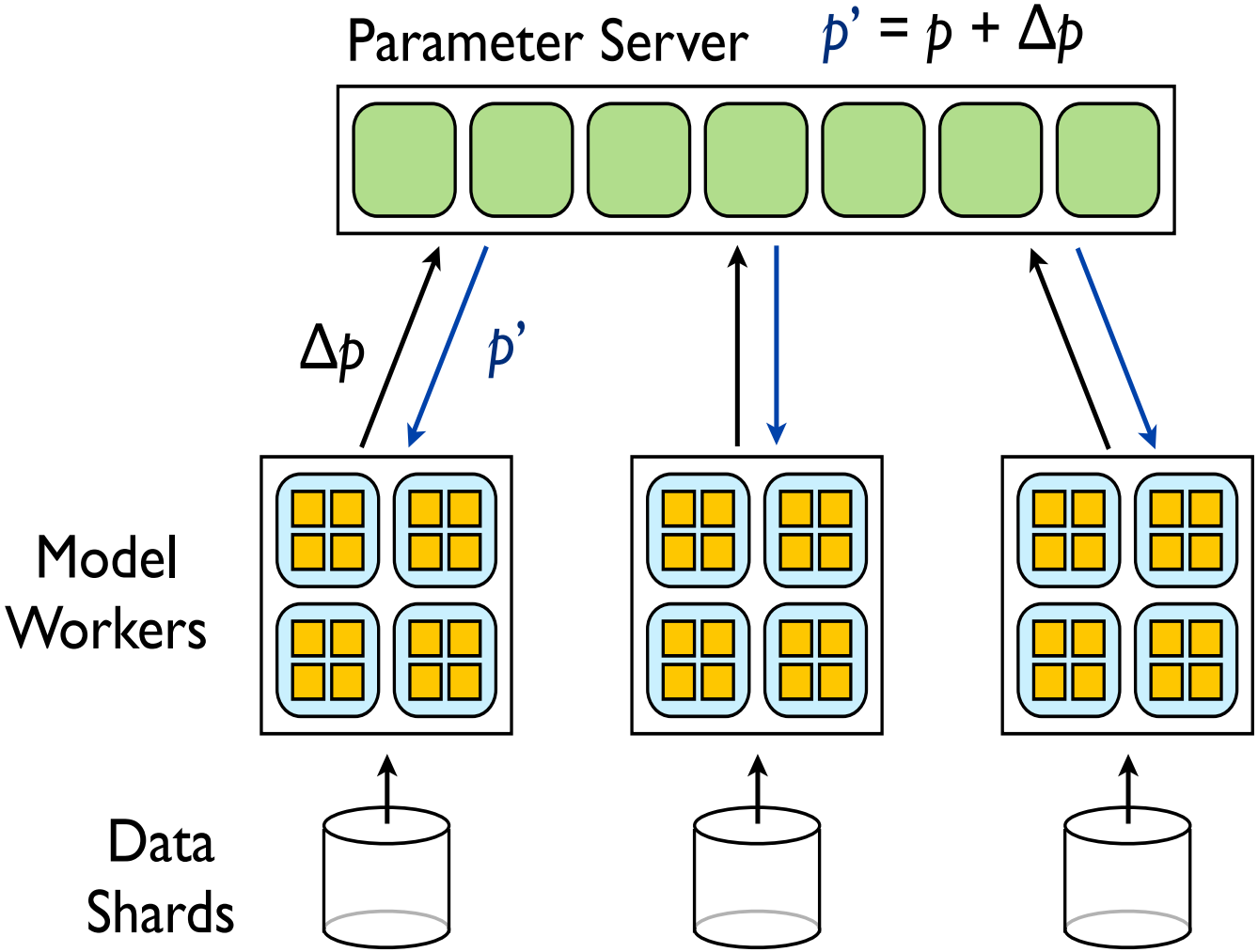
- **Use batches of N inputs…**
- **Max theoretical speed up: N**

Parallel creation of gradient

- **Steepest descent** $\theta^{k+1} = \theta^k - \alpha \cdot dE/d\theta$
- **Derivative** $dE/d\theta = 2 \cdot \Sigma[(y(\theta)-z) \cdot dy(\theta)/d\theta]$

Split sum of pieces of $dE/d\theta$ across different nodes!

Batch Parameter Update

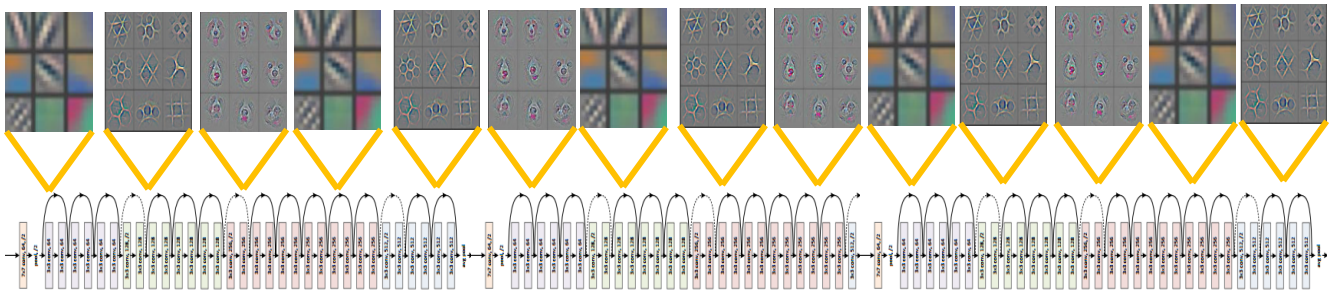


[Dean et al., NIPS 2012]

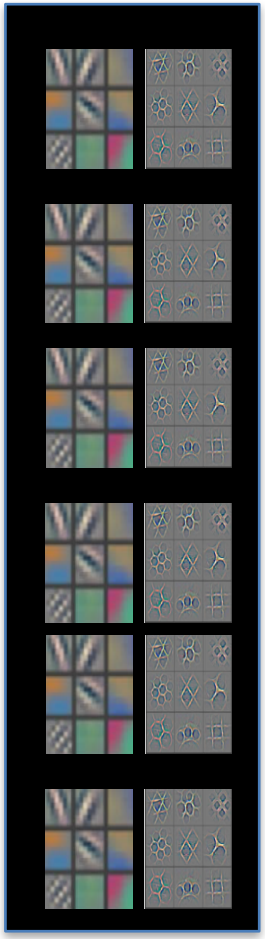
Training Uses a Lot of Memory

GPU memory usage proportional to network depth

Feature maps



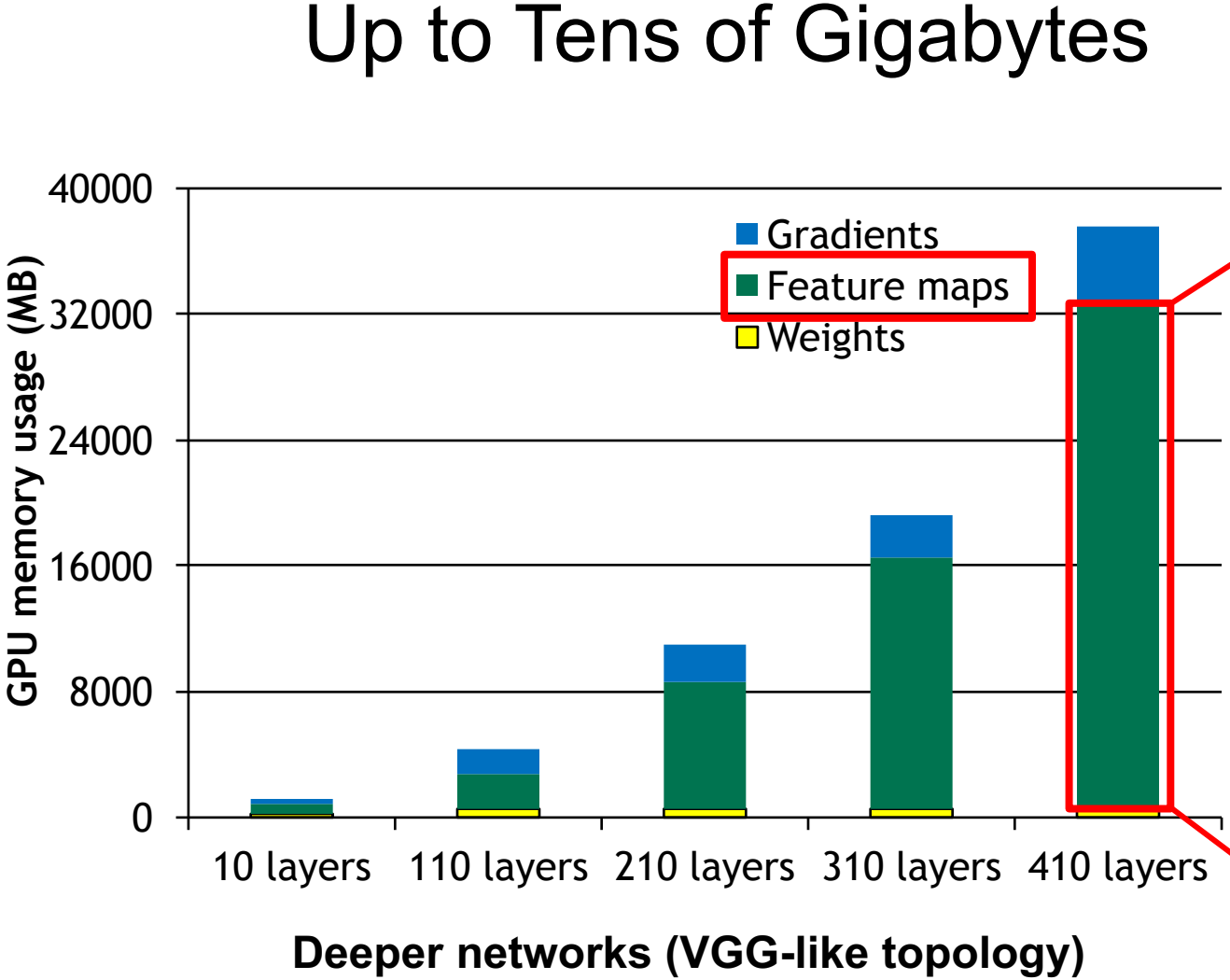
GPU memory

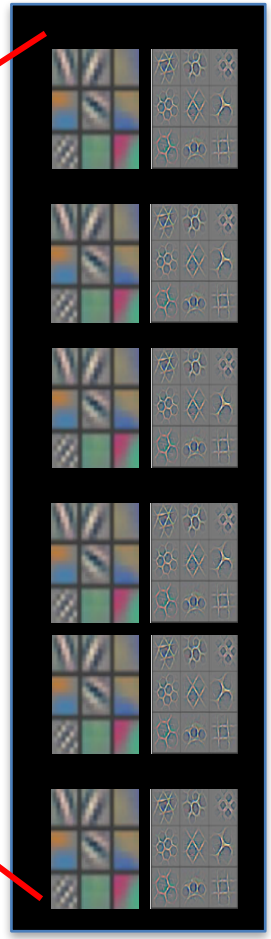


[Rhu et al., vDNN, MICRO 2016]

How Much Memory Is It?

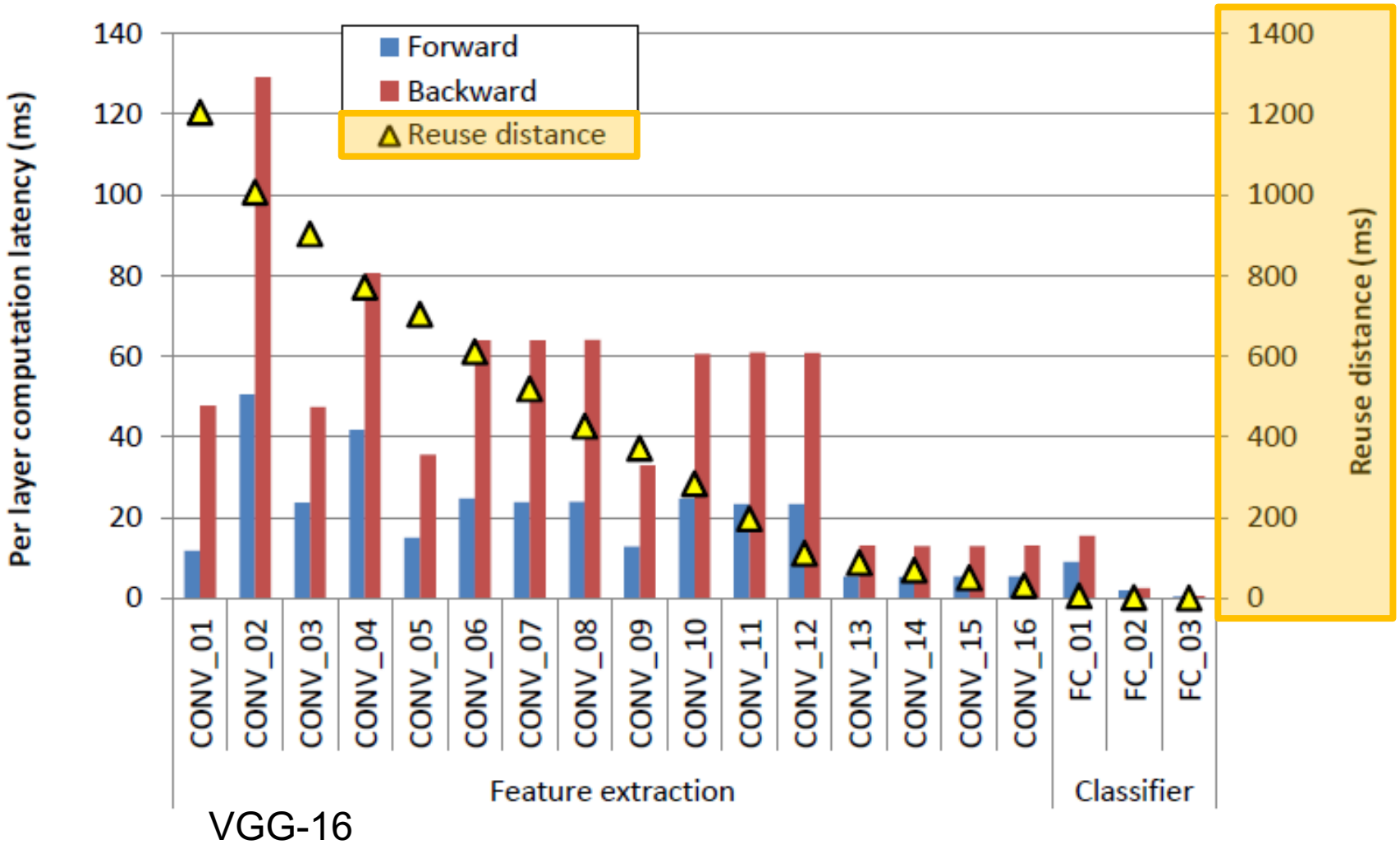
Up to Tens of Gigabytes





[Rhu et al., vDNN, MICRO 2016]

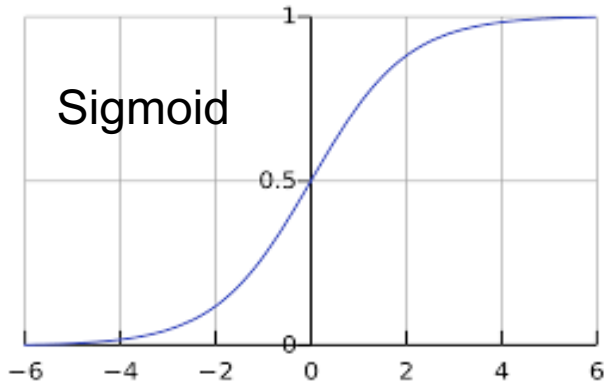
Reuse Distance of Feature Maps

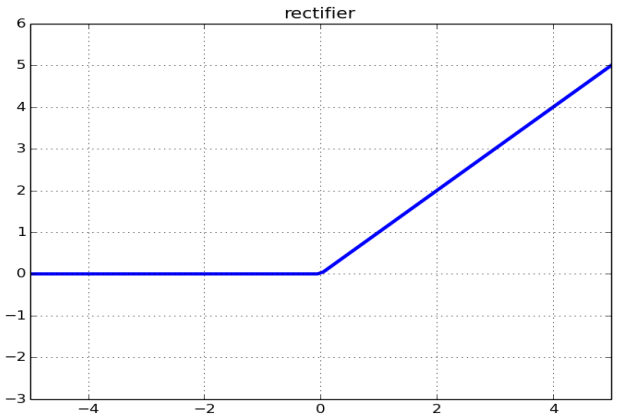


VGG-16

[Rhu et al., vDNN, MICRO 2016]

Problems with saturation





Issue

- A null gradient results in no learning, which happens if:
  - the sigmoid saturates, or
  - the ReLU saturates

Solution

- Initialize weighs so the average value is zero, i.e., work in the interesting zone of the activation functions
- Normalize data (zero mean)

Non-differential operations

Issue

- Discrete activation function / weights
  - extreme case is binary net
- Derivative not well defined

Solution

- Use approximate derivative, or
- Discretize a-posteriori

Model Overfitting

Problem:

- Neural net learns too specifically from input set, rather than generalizing from input, called overfitting
- Overfitting can be a result of too many parameters in model

Solution:

- Dropout – turn off neurons at random; other neurons will take care of their job.
  - + Reliability
  - - Redundancy (-> pruning)

Architecture Challenges for Training

- Floating point accuracy
- Where to store the gradients
- Synchronization for parallel processing

References

MICRO Tutorial (2016)

Website: http://eyeriss.mit.edu/tutorial.html

Joel Emer, Vivienne Sze, Yu-Hsin Chen

References (Alphabetical by Author)

- Albericio, Jorge, et al. "Cnvlutin: ineffectual-neuron-free deep neural network computing." Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. IEEE, 2016.
- Alwani, Manoj, et al., "Fused Layer CNN Accelerators," MICRO, 2016
- Chakradhar, Srimat, et al., "A dynamically configurable coprocessor for convolutional neural networks," ISCA, 2010
- Chen, Tianshi, et al., "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," ASPLOS, 2014
- Chen, Yu-Hsin, et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." 2016 IEEE International Solid-State Circuits Conference (ISSCC). IEEE, 2016.
- Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," ISCA, (2016).
- Chen, Yunji, et al. "Dadiannao: A machine-learning supercomputer."Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014.
- Chi, Ping, et al. "PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory." Proceedings of ISCA. Vol. 43. 2016.
- Cong, Jason, and Bingjun Xiao. "Minimizing computation in convolutional neural networks." International Conference on Artificial Neural Networks. Springer International Publishing, 2014.
- Courbariaux, Matthieu, and Yoshua Bengio. "Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1." arXiv preprint arXiv:1602.02830 (2016).
- Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations." Advances in Neural Information Processing Systems. 2015.

References (Alphabetical by Author)

- Dean, Jeffrey, et al., "Large Scale Distributed Deep Networks," NIPS, 2012
- Denton, Emily L., et al. "Exploiting linear structure within convolutional networks for efficient evaluation." Advances in Neural Information Processing Systems. 2014.
- Dorrance, Richard, Fengbo Ren, and Dejan Marković. "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs." Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays. ACM, 2014.
- Du, Zidong, et al., "ShiDianNao: shifting vision processing closer to the sensor," ISCA, 2015
- Eryilmaz, Sukru Burc, et al. "Neuromorphic architectures with electronic synapses." 2016 17th International Symposium on Quality Electronic Design (ISQED). IEEE, 2016.
- Esser, Steven K., et al., "Convolutional networks for fast, energy-efficient neuromorphic computing," PNAS 2016
- Farabet, Clement, et al., "An FPGA-Based Stream Processor for Embedded Real-Time Vision with Convolutional Networks," ICCV 2009
- Gokhale, Vinatak, et al., "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," CVPR Workshop, 2014
- Govoreanu, B., et al. "10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation." Electron Devices Meeting (IEDM), 2011 IEEE International. IEEE, 2011.
- Gupta, Suyog, et al., "Deep Learning with Limited Numerical Precision," ICML, 2015
- Gysel, Philipp, Mohammad Motamedi, and Soheil Ghiasi. "Hardware-oriented Approximation of Convolutional Neural Networks." arXiv preprint arXiv:1604.03168 (2016).
- Han, Song, et al. "EIE: efficient inference engine on compressed deep neural network." arXiv preprint arXiv:1602.01528 (2016).

References (Alphabetical by Author)

- Han, Song, et al. "Learning both weights and connections for efficient neural network." Advances in Neural Information Processing Systems. 2015.
- Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding." CoRR, abs/1510.00149 2 (2015).
- He, Kaiming, et al. "Deep residual learning for image recognition." arXiv preprint arXiv:1512.03385 (2015).
- Horowitz, Mark. "1.1 Computing's energy problem (and what we can do about it)." 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE, 2014.
- Iandola, Forrest N., et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 1MB model size." arXiv preprint arXiv:1602.07360 (2016).
- Ioffe, Sergey, and Szegedy, Christian, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ICML 2015
- Jermyn, Michael, et al., "Neural networks improve brain cancer detection with Raman spectroscopy in the presence of operating room light artifacts," Journal of Biomedical Optics, 2016
- Judd, Patrick, et al. "Reduced-precision strategies for bounded memory in deep neural nets." arXiv preprint arXiv:1511.05236 (2015).
- Judd, Patrick, Jorge Albericio, and Andreas Moshovos. "Stripes: Bit-serial deep neural network computing." IEEE Computer Architecture Letters (2016).
- Kim, Duckhwan, et al. "Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory." Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. IEEE, 2016.
- Kim, Yong-Deok, et al. "Compression of deep convolutional neural networks for fast and low power mobile applications." ICLR 2016

References (Alphabetical by Author)

- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.
- Lavin, Andrew, and Gray, Scott, "Fast Algorithms for Convolutional Neural Networks," arXiv preprint arXiv:1509.09308 (2015)
- LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.
- LeCun, Yann, et al. "Optimal brain damage." NIPs. Vol. 2. 1989.
- Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).
- Mathieu, Michael, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through FFTs." arXiv preprint arXiv:1312.5851 (2013).
- Merola, Paul A., et al. "Artificial brains. A million spiking-neuron integrated circuit with a scalable communication network and interface," Science, 2014
- Moons, Bert, and Marian Verhelst. "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets." VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on. IEEE, 2016.
- Moons, Bert, et al. "Energy-efficient ConvNets through approximate computing." 2016 IEEE Winter Conference on Applications of Computer Vision (WACV). IEEE, 2016.
- Park, Seongwook, et al., "A 1.93TOPS/W Scalable Deep Learning/Inference Processor with Tetra-Parallel MIMD Architecture for Big-Data Applications," ISSCC, 2015
- Peemen, Maurice, et al., "Memory-centric accelerator design for convolutional neural networks," ICCD, 2013
- Prezioso, Mirko, et al. "Training and operation of an integrated neuromorphic network based on metal-oxide memristors." Nature 521.7550 (2015): 61-64.

References (Alphabetical by Author)

- Rastegari, Mohammad, et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." arXiv preprint arXiv:1603.05279(2016).
- Reagen, Brandon, et al. "Minerva: Enabling low-power, highly-accurate deep neural network accelerators." Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016.
- Rhu, Minsoo, et al., "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," MICRO, 2016
- Russakovsky, Olga, et al. "Imagenet large scale visual recognition challenge." International Journal of Computer Vision 115.3 (2015): 211-252.
- Sermanet, Pierre, et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks." arXiv preprint arXiv:1312.6229 (2013).
- Shafiee, Ali, et al. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars." Proc. ISCA. 2016.
- Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
- Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.
- Vasilache, Nicolas, et al. "Fast convolutional nets with fbfft: A GPU performance evaluation." arXiv preprint arXiv:1412.7580 (2014).
- Yang, Tien-Ju, et al. "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," arXiv, 2016
- Zhang, Chen, et al., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," FPGA, 2015