

6.5930/1

Hardware Architectures for Deep Learning

# **Co-Design of DNN Models and Hardware: Sparsity**

February 25, 2026

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology  
Electrical Engineering & Computer Science



# Goals of Today's Lecture

---

- Today, we will focus on *reducing the number* of operations for storage/compute
- Exploit sparsity, where sparsity refers to repeated values, in most cases, repeated zeros
  - Exploit natural sparsity in the data
  - Create sparsity using **pruning!**
- Potential architectural benefits of sparsity
  - (1) Reduce data movement and storage cost
  - (2) Reduce number of operations

# Sources of Sparsity

---

- **(Input) Activation Sparsity**
  - Sparsity due to ReLU
  - Correlation in input data
  - Structure of input representation (e.g., Graphs)
  
- **Weight Sparsity**
  - Weight reordering and reuse
  - Network pruning

# Exploiting Sparsity

---

Sparse data can be compressed



Can save space and energy by avoiding **storage and movement** of zero values

$$\mathit{anything} \times 0 = 0$$

$$\mathit{anything} + 0 = \mathit{anything}$$



Can save time and energy by avoiding fetching unnecessary operands and avoiding **ineffectual** computations

# Impact of Ineffectual Operations

---

- Total operations = effectual operations + ineffectual operations  
*(total operations referred to as “**algorithmic computes**” in Lab 4)*
  
- Total operations **performed** = effectual operations + unexploited ineffectual operations  
*(total operations referred to as “**actual computes**” in Lab 4)*

# Unexploited Ineffectual Operations

---

- Ideally, hardware would exploit and thus avoid **all** ineffectual operations
  - Total operations performed = effectual operations ~~+ unexploited ineffectual operations~~
- In practice, trying to avoid **all** ineffectual operations is challenging and can increase hardware overhead
  - Tradeoff between reducing number of operations and cost per operation
  - *We will discuss hardware in the next lecture*
- In this lecture, we focus on factors that impact the number of ineffectual operations in the workload

# Impact of Ineffectual Operations on Throughput

Reduces with more hardware complexity

$$\frac{\text{operations}}{\text{cycle}} = \frac{\text{effectual operations} + \text{unexploited ineffectual operations}}{\text{cycle}} \times \frac{\text{effectual operations}}{\text{effectual operations} + \text{unexploited ineffectual operations}} \times \frac{1}{\frac{\text{effectual operations}}{\text{operations}}} \quad (3.4)$$

Increase with more hardware complexity  
(skip more operations)

Depends on  
sparsity in workload

Chapter 3 in textbook

# Impact of Ineffectual Operations on Energy Efficiency

Reduces with more hardware complexity

$$\frac{\text{operations}}{\text{joule}} = \frac{\text{effectual operations} + \text{unexploited ineffectual operations}}{\text{joule}} \times \frac{\text{effectual operations}}{\text{effectual operations} + \text{unexploited ineffectual operations}} \times \frac{1}{\frac{\text{effectual operations}}{\text{operations}}}$$

Increase with more hardware complexity (3.9) (skip more operations)

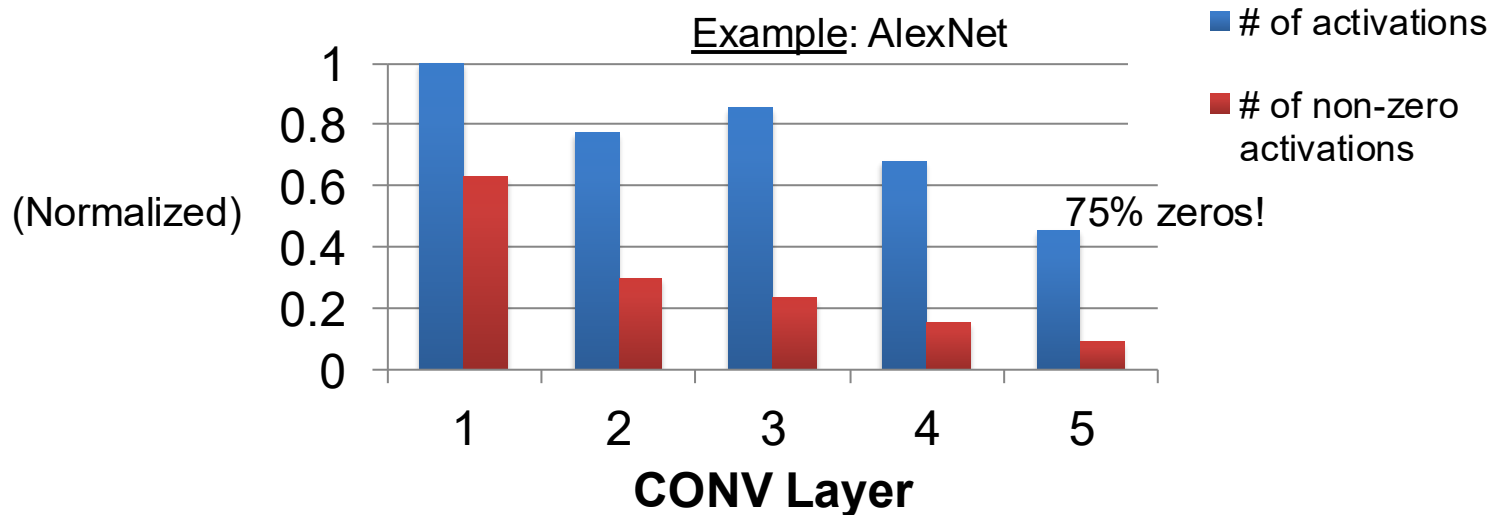
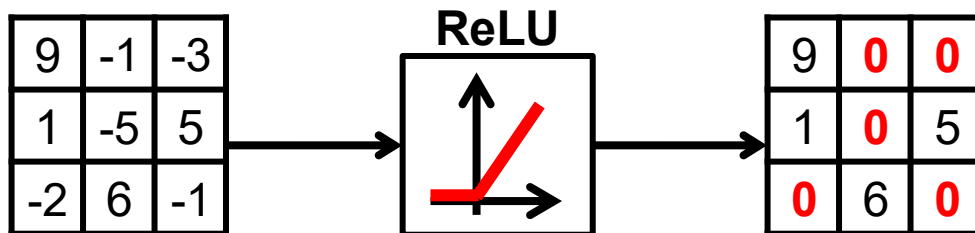
Depends on sparsity in workload

Chapter 3 in textbook

# Activation Sparsity

# Sparsity in Feature Maps

Many **zeros** in output fmaps after ReLU



# Apply Compression

- Compress Sparse Data

- Reduce data movement cost (memory bandwidth)
- Reduce storage cost
  - Can also reduce data movement cost by storing more data at each level of the memory hierarchy

- Requirements

- Uniquely decodable
  - For variable length coding
- Lightweight algorithm
- Usually lossless
  - Does not affect accuracy

Example  $L = 4$  (not uniquely decodable)

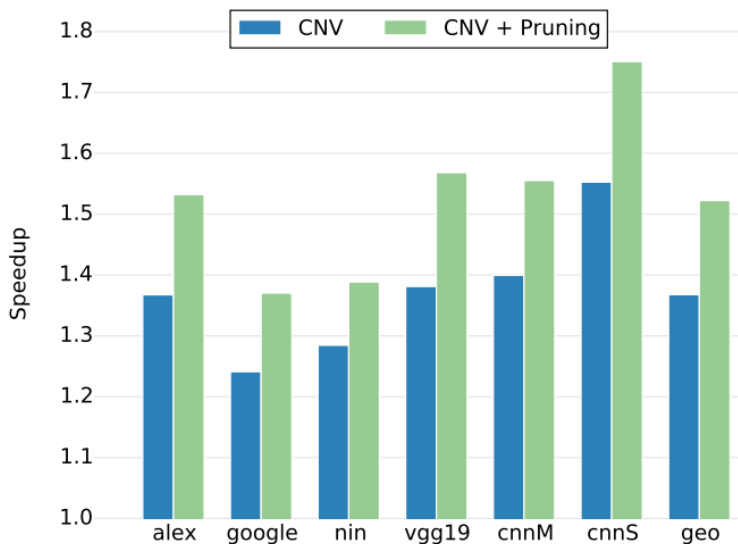
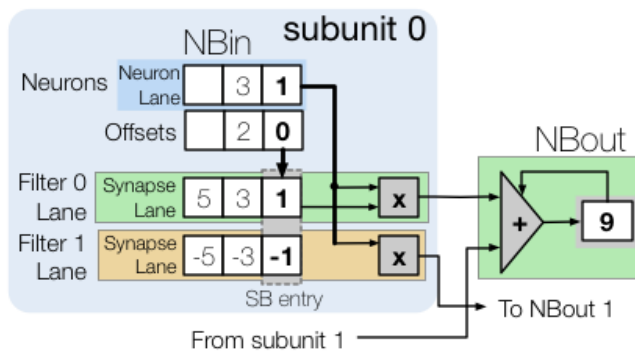
$r_0$	0
$r_1$	1
$r_2$	0 0
$r_3$	0 1

$L = 4$  (uniquely decodable)

$r_0$	0 0
$r_1$	0 1
$r_2$	1 0
$r_3$	1 1

# Skip Zero Activations: Cnvlutin

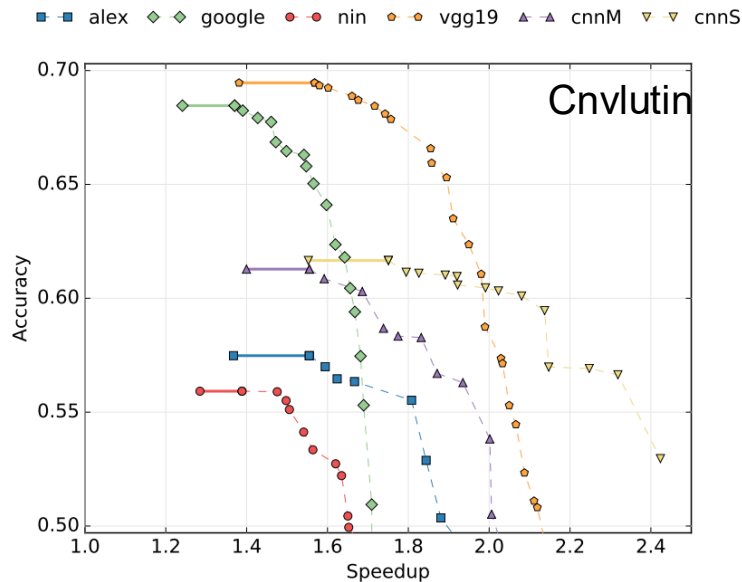
- Process Convolution Layers
- Built on top of DaDianNao (4.49% area overhead)
- Speed up of 1.37x (1.52x with activation pruning)



# Pruning Activations

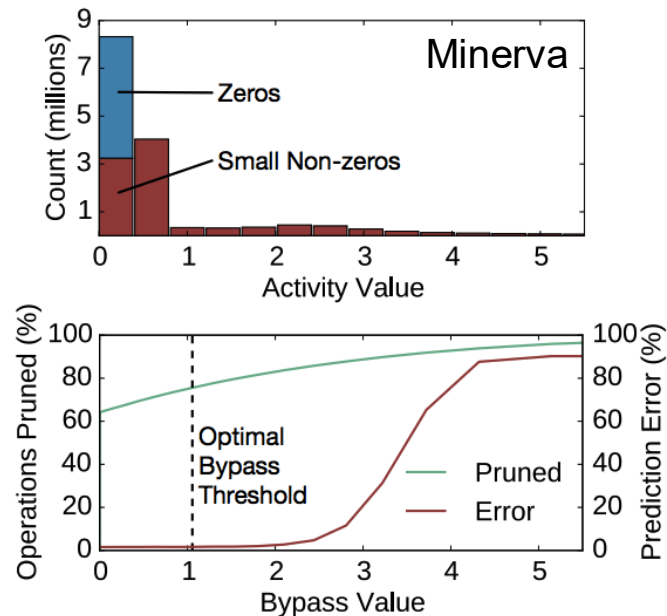
Remove small activation values (*affects accuracy!*)

**Speed up 11% (ImageNet)**



[Albericio, /SCA 2016]

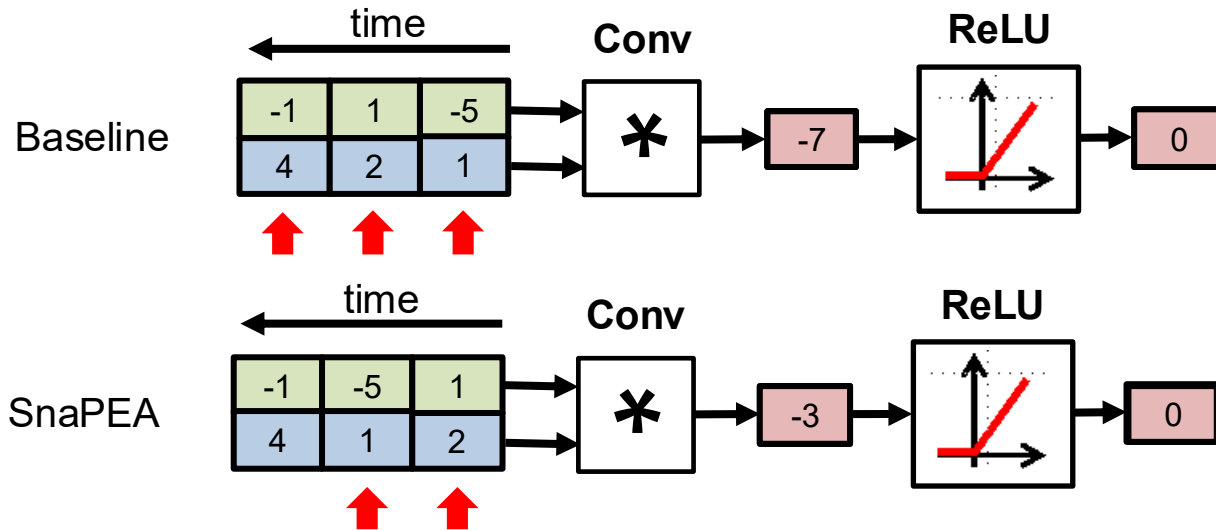
**Reduce power 2x (MNIST)**



[Reagen, /SCA 2016]

# Exploit ReLU

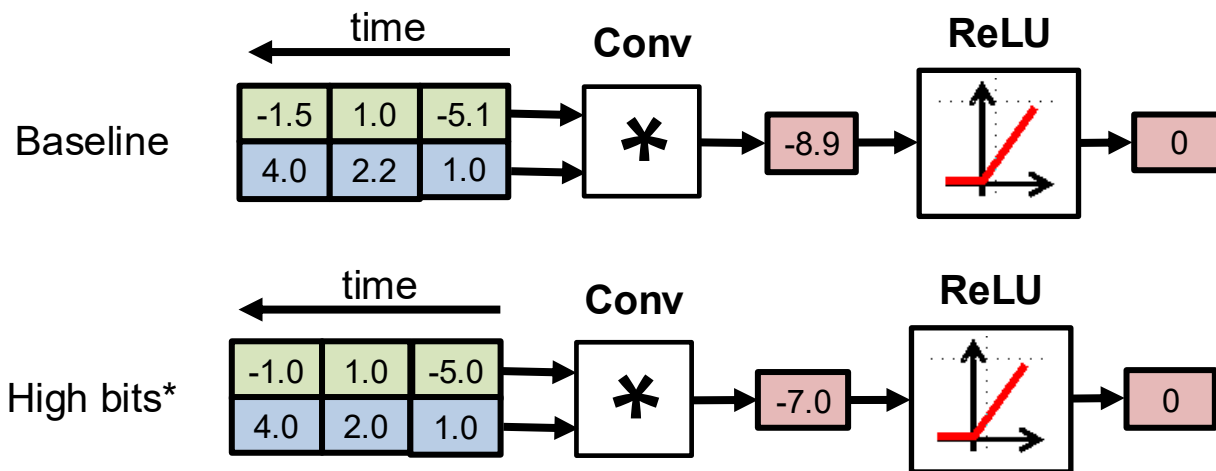
Reduce number operations when if resulting activation will be negative as ReLU will output a zero



Additional hardware required to decide when to terminate

# Exploit ReLU

Simplify operations to cheaply check if resulting activation will be negative as ReLU will output a zero



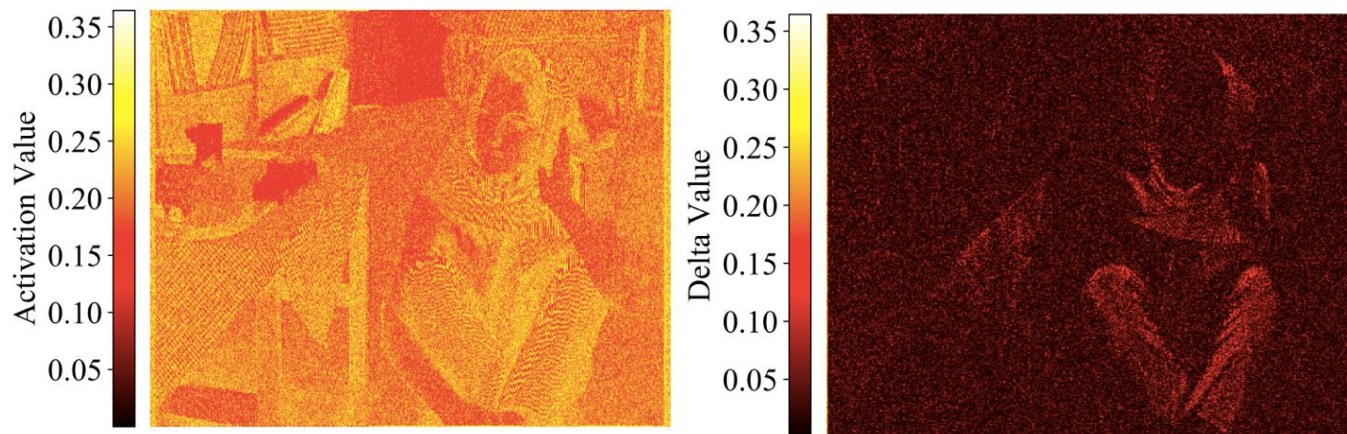
Only compute on low bits if result is positive

\*over-simplified

[PredictiveNet, /SCAS 2017], [Song, /SCA 2018]

# Exploit Spatial Correlation of Inputs

Neighboring activations in feature map are correlated



Process Activations  
(baseline)

$$y_1 = a_1 \times w$$

$$y_2 = a_2 \times w$$

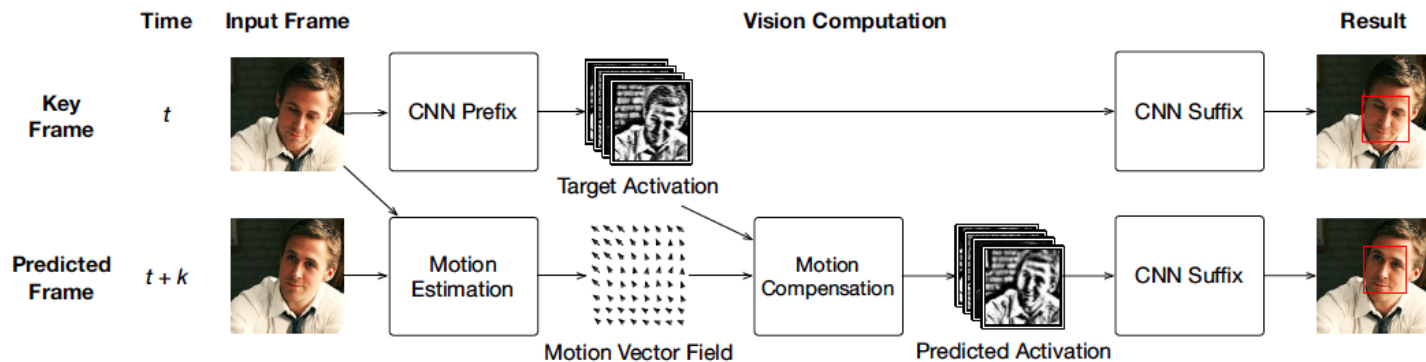
Process Delta

$$y_1 = a_1 \times w$$

$$y_2 = a_1 \times w + (a_2 - a_1) \times w = y_1 + \Delta_a \times w$$

# Exploit Temporal Correlation of Inputs

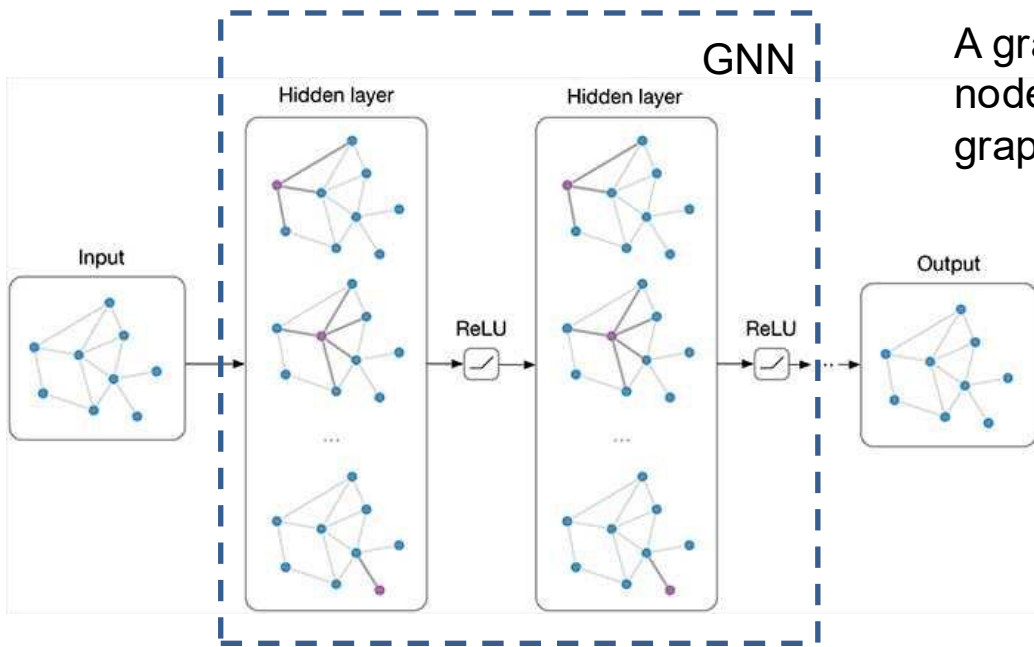
- Reduce amount of computation if there is temporal correlation between inputs (e.g., frames)
- Requires additional storage and need to find redundancy (e.g., motion vectors for videos)
- Application specific (e.g., videos) – requires that the same operation is done for each frame (not always the case)



[EVA<sup>2</sup>, ISCA 2018], [Euphrates, ISCA 2018], [Riera, ISCA 2018], [FAST, CVPRW 2017]

# Graph Neural Networks (GNN)

Graphs are widely used to represent data such as molecules, social, biological, and financial networks.



A graph can be described in terms of its nodes and edges, i.e.,  $G = (V, E)$  denote a graph with nodes feature vectors  $X_v$  for  $v \in V$

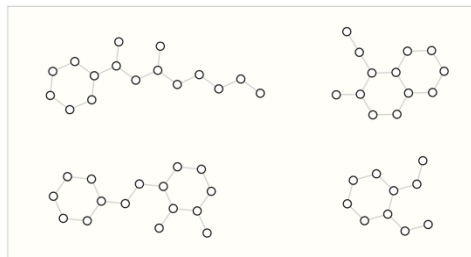
Popular variants of GNN include Graph Convolutional Networks (GCN) [Kipf, ICLR 2017] and GraphSAGE [Hamilton, NeurIPS 2017].

Image Source: <https://tkipf.github.io/graph-convolutional-networks/>

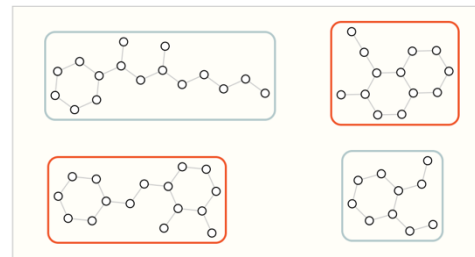
# Example Graph Neural Networks Tasks

Output can be a label on the graph topology (i.e., how nodes are connected by edges), node, or edge.

## Graph example

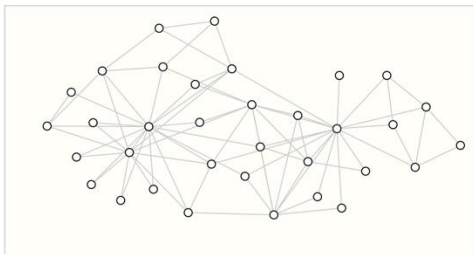


Input: graphs

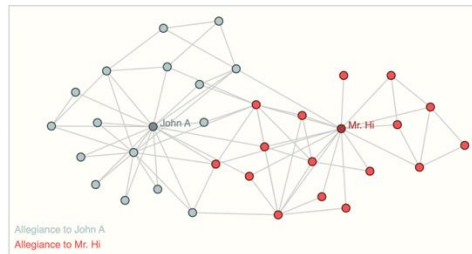


Output: labels for each graph, (e.g., "does the graph contain two rings?")

## Nodes example

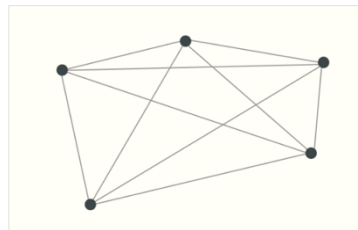


Input: graph with unlabeled nodes

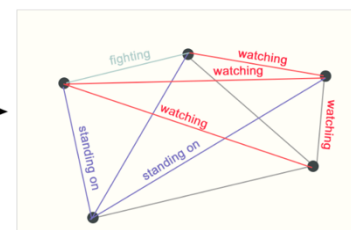


Output: graph node labels

## Edge example



Input: fully connected graph, unlabeled edges

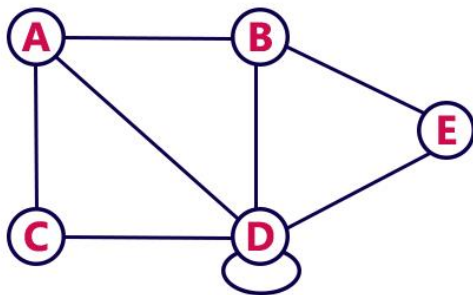


Output: labels for edges

# Structure of Graph Representation

The topology of the graph can be represented by an **Adjacency Matrix**, which is usually **sparse**!

Nodes: A, B, C, D, E



$$\begin{matrix} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \mathbf{A} & \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \end{pmatrix} \\ \mathbf{B} & \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \end{pmatrix} \\ \mathbf{C} & \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \end{pmatrix} \\ \mathbf{D} & \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix} \\ \mathbf{E} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{matrix}$$

**Adjacency Matrix**

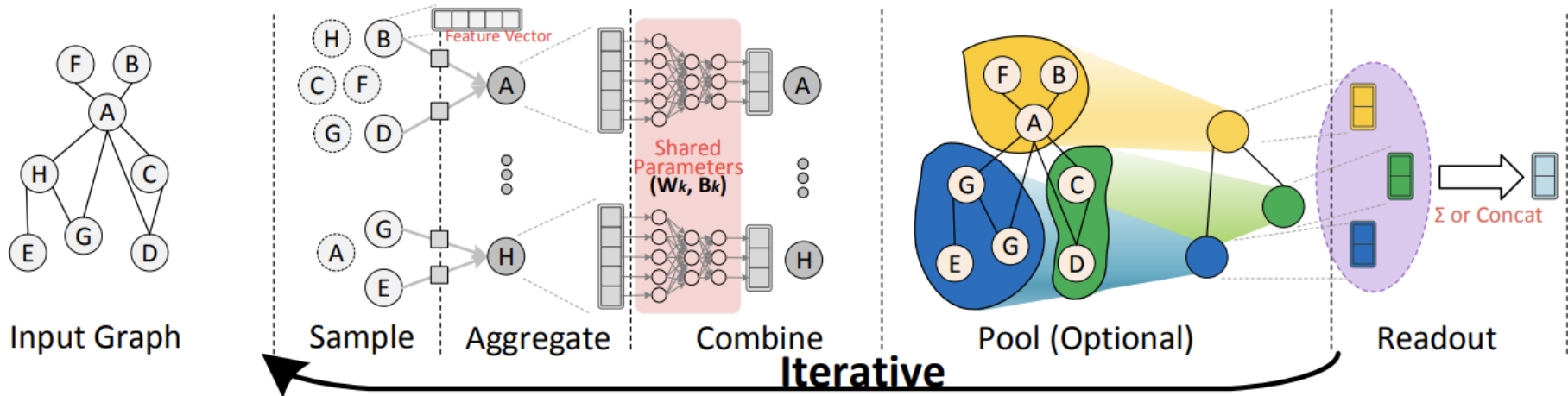
Each node can be represented by a feature vector, and the aggregate of the nodes is represented by a **feature matrix**.

Image source: [http://www.btechsmartclass.com/data\\_structures/graph-representations.html](http://www.btechsmartclass.com/data_structures/graph-representations.html)

# Key Steps in GNN

- **Aggregate:** Get node features from a node's neighbors to form a matrix and average\* them to form a vector: this is the intermediate node feature
- **Combine:** Apply weights onto intermediate node feature to get next-layer node feature

\*Note: can be some other function



# Computation in GNN

$$X^{(1)} = \sigma(\hat{A}X^{(0)}W^{(0)}) \quad \text{Layer 0}$$

$$X^{(2)} = \sigma(\hat{A}X^{(1)}W^{(1)}) \quad \text{Layer 1}$$

Output [**NxF**]

...

$$X^{(l+1)} = \sigma(\hat{A}X^{(l)}W^{(l)}) \quad \text{Layer } l$$

Normalized  
Adjacency  
Matrix [**NxN**]  
(typically, sparse)

Feature  
Matrix [**NxF**]  
(typically, all nodes)

Weights [**FxF**]

# Computation in GNN

---

- Adjacency matrix is normalized to maintain the scale of the output feature vectors (can be precomputed)

$$\hat{A} = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}},$$

where  $D$  is the diagonal matrix and  $I$  is the identity matrix

- Can reuse same adjacency matrix across layers (topology unchanged)
- Order of operations  $(\hat{A} \times X) \times W$  or  $\hat{A} \times (X \times W)$  impacts sparsity

# Weight Sparsity

# 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

# Exploit Redundant Weights

---

- Preprocessing to reorder weights (ok since weights known)
- Perform addition before multiplication to reduce number of multiplies and reads of weights
- **Example:** Input = [1 2 3 ] and filter [A B A]

Typical processing: Output =  $A*1+B*2+A*3$

3 multiplies and 3 weight reads

If reorder as [AA B]: Output =  $A*(1+3)+B*1$

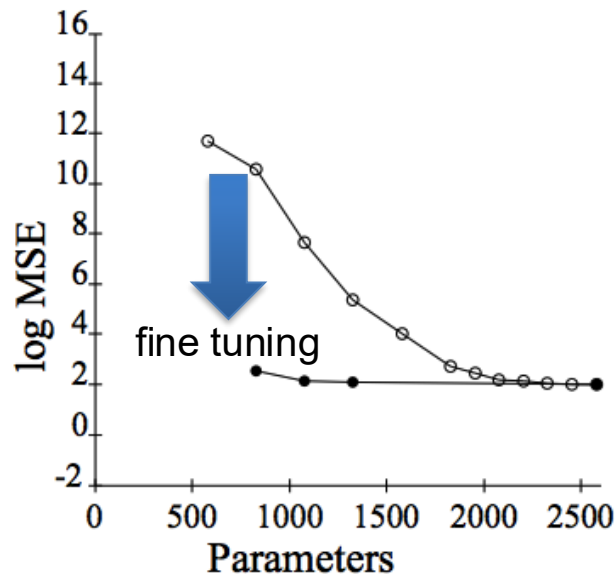
2 multiplies and 2 weight reads

*Note: Bitwidth of multiplication may need to increase*

# 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



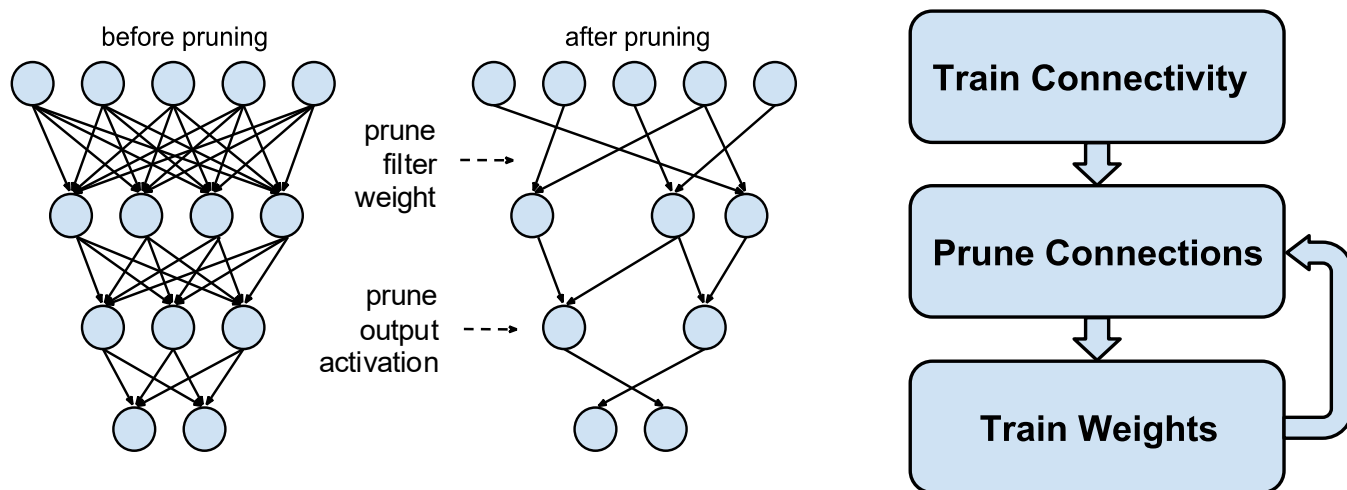
[Lecun, *NeurIPS* 1989]



# Pruning – Make Weights Sparse

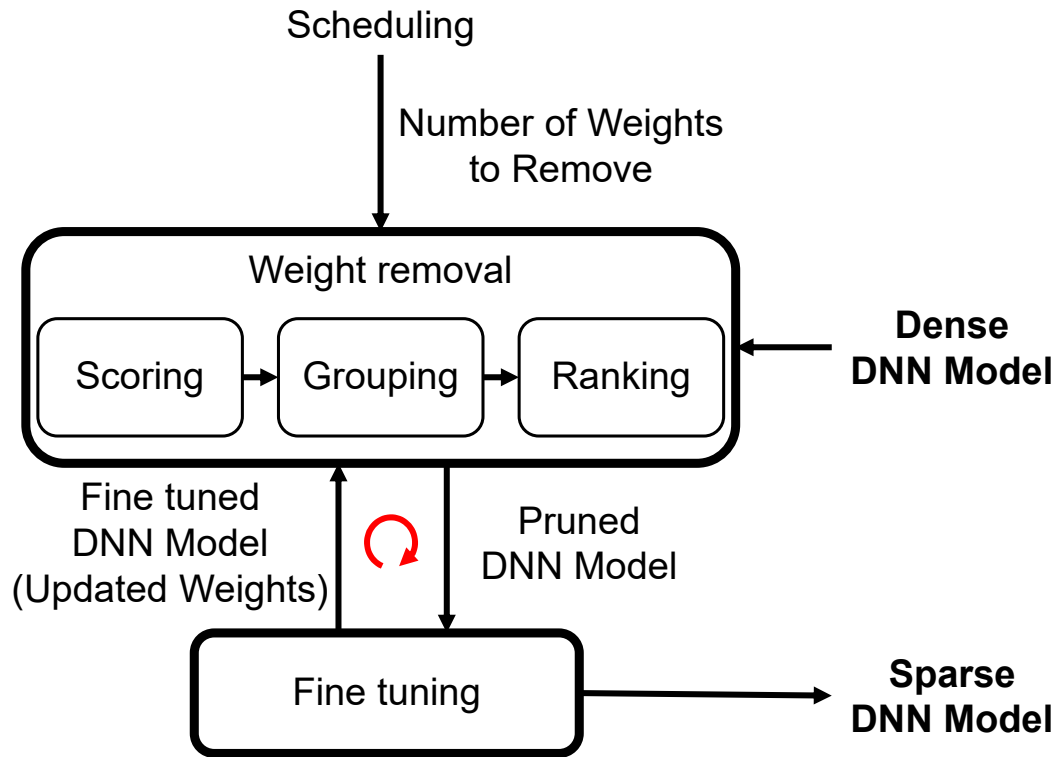
Prune based on *magnitude* of weights

[Hertz et al., Neural Computation, 1991]



Typical numbers: 50% sparsity without retraining, 80% with retraining

# Pruning – Make Weights Sparse



# Weight Removal: Scoring

---

- Assign a score to each weight or a group of weights based on impact on some criteria (usually accuracy)
- **Magnitude-based pruning** (most common)
  - Assign score based on magnitude of weight
- **Feature-based pruning**
  - Assign score based on impact on output feature map

# Weight Removal: Scoring

$$\begin{array}{|c|c|c|} \hline \text{fmap} & & \text{filter} \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -8 & 3 & 2 \\ \hline 1 & -3 & -2 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \boxed{-4}$$

Magnitude-based pruning

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -8 & 3 & 2 \\ \hline 1 & -3 & -2 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline -8 & 3 & 2 \\ \hline \emptyset & -3 & -2 \\ \hline \emptyset & \emptyset & \emptyset \\ \hline \end{array} = \boxed{-8}$$

Error = -4

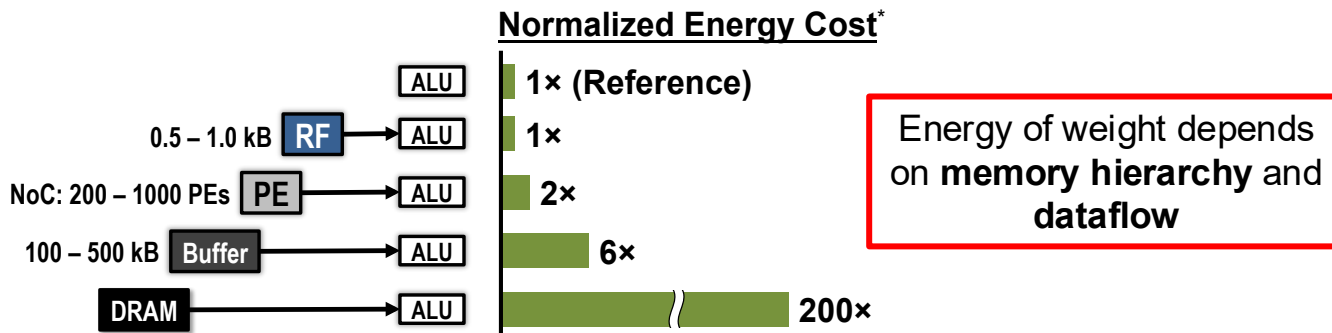
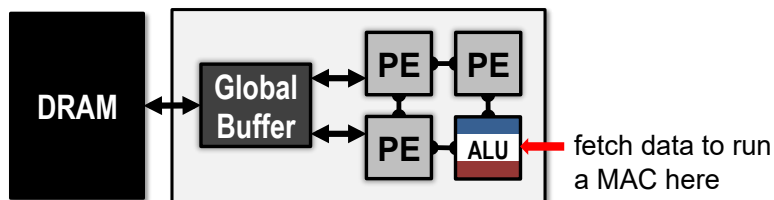
Feature-based pruning

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline -8 & 3 & 2 \\ \hline 1 & -3 & -2 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline -8 & \emptyset & \emptyset \\ \hline 1 & \emptyset & \emptyset \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \boxed{-4}$$

Error = 0

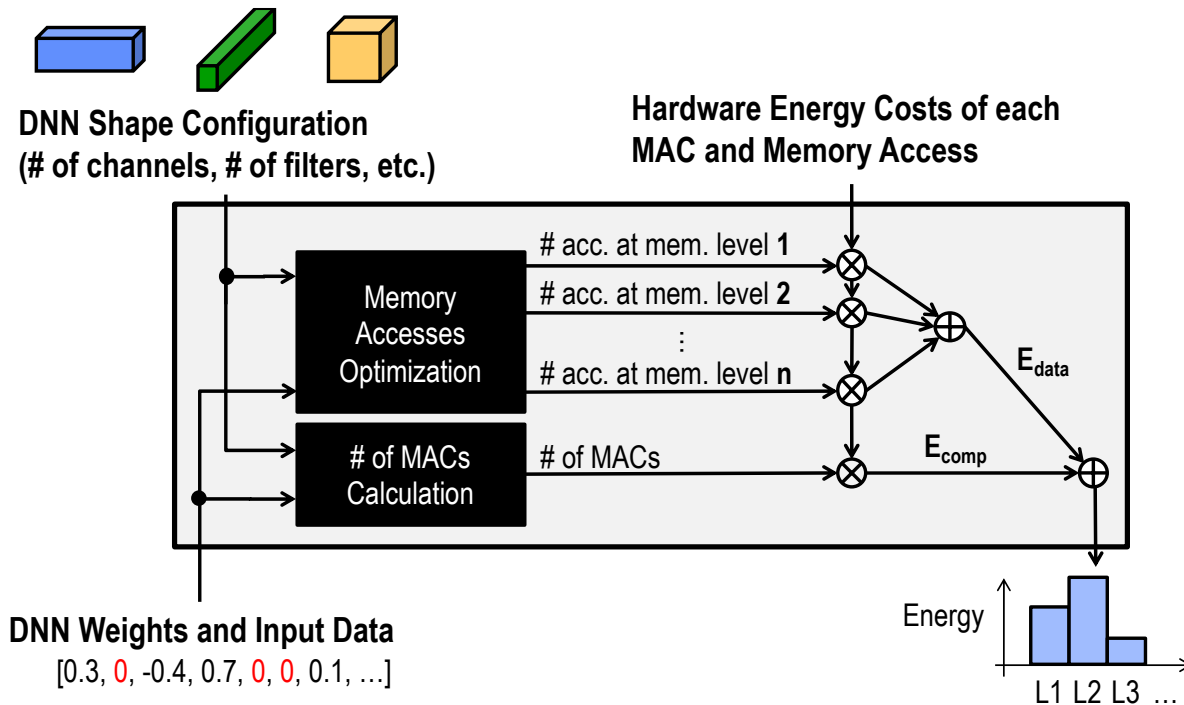
# Weight Removal: Scoring

Also consider the impact that each weight has on energy efficiency and throughput



\* measured from a commercial 65nm process

# Energy Estimation



Tool available at <https://energyestimation.mit.edu/>

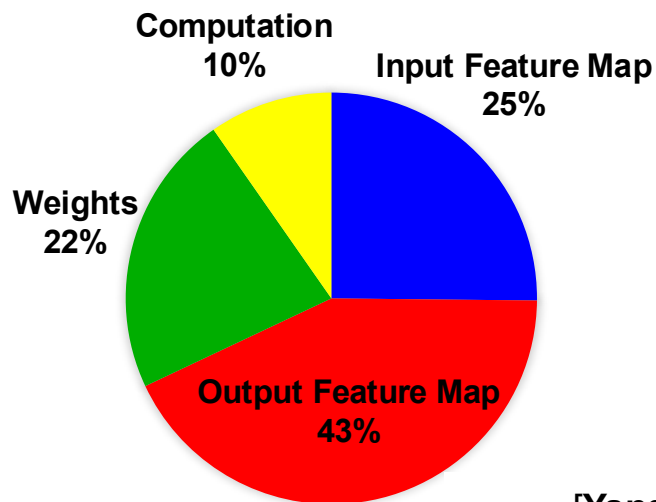
For class, use Timeloop/Accelergy

# Key Insights

---

- Number of weights alone is not a good metric for energy
- All data types should be considered

## GoogLeNet Energy Breakdown



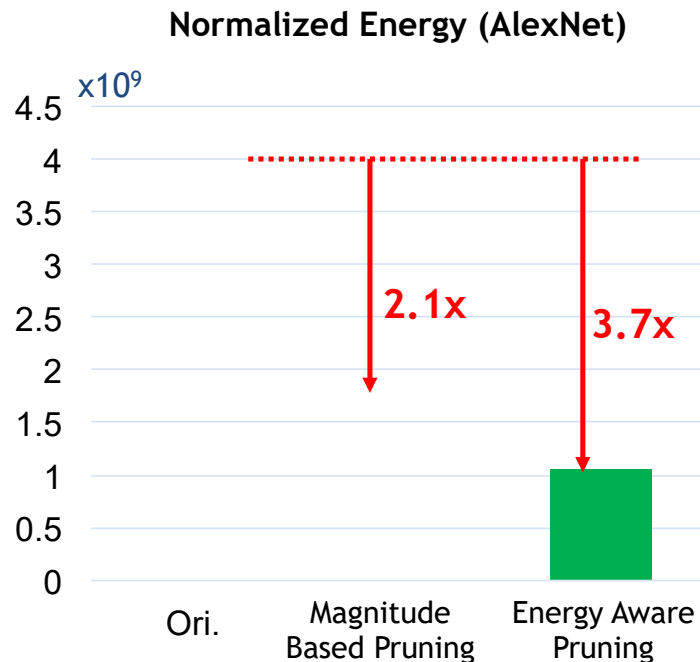
[Yang, CVPR 2017]

# Energy-Aware Pruning

**Directly target energy** and incorporate it into the optimization of DNNs to provide greater energy savings

- Sort layers based on energy and prune layers that consume most energy first
- EAP reduces AlexNet energy by **3.7x** and outperforms the previous work that uses magnitude-based pruning by **1.7x**

[Yang, CVPR 2017]



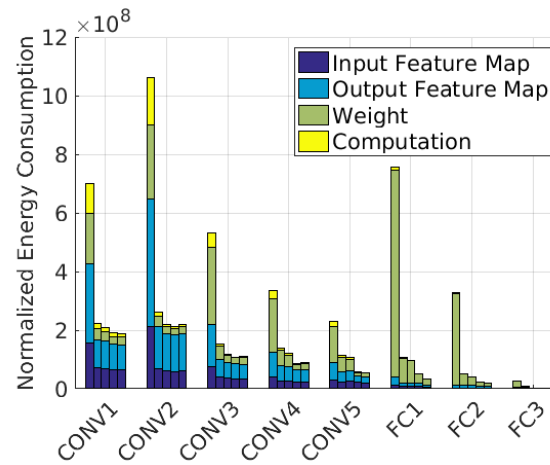
Pruned models available at  
<http://eyeriss.mit.edu/energy.html>

# Prune to Reduce Number of Classes

Table 2. Compression ratio<sup>1</sup> of each layer in AlexNet.

# of Classes	[8]	This Work			
	1000	1000	100	10 (Random)	10 (Dog)
CONV1	16%	83%	86%	89%	89%
CONV2	62%	92%	97%	97%	96%
CONV3	65%	91%	97%	98%	97%
CONV4	63%	81%	88%	97%	95%
CONV5	63%	74%	79%	98%	98%
FC1	91%	92%	93%	~100%	~100%
FC2	91%	91%	94%	~100%	~100%
FC3	74%	78%	78%	~100%	~100%

<sup>1</sup> The number of removed weights divided by the number of total weights. The higher, the better.



The energy breakdown of the networks in this work. Following the same order as the table.

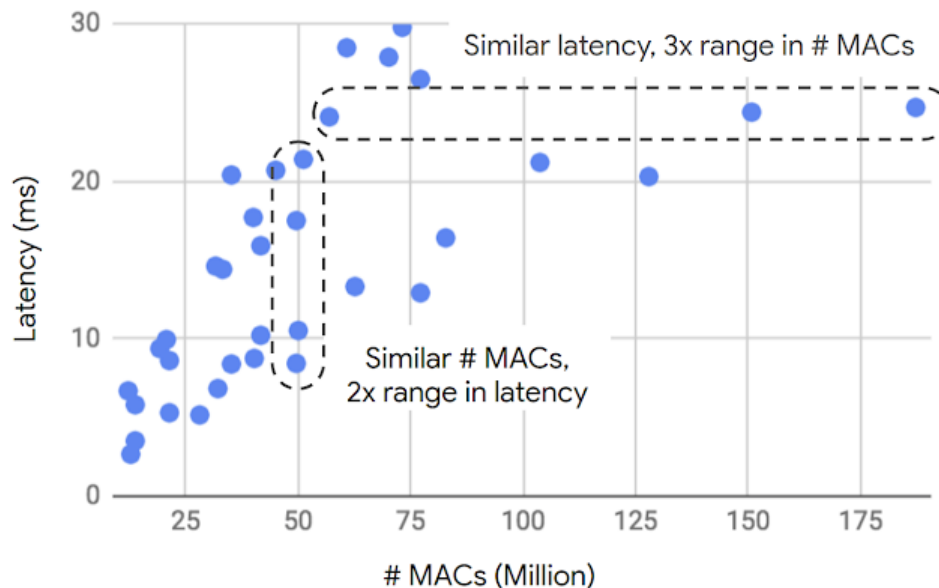
- When reducing the number of classes of AlexNet,
  - Large compression ratios are achieved in all layers except for **CONV1**

[Yang, CVPR 2017]



# # of Operations vs. Latency

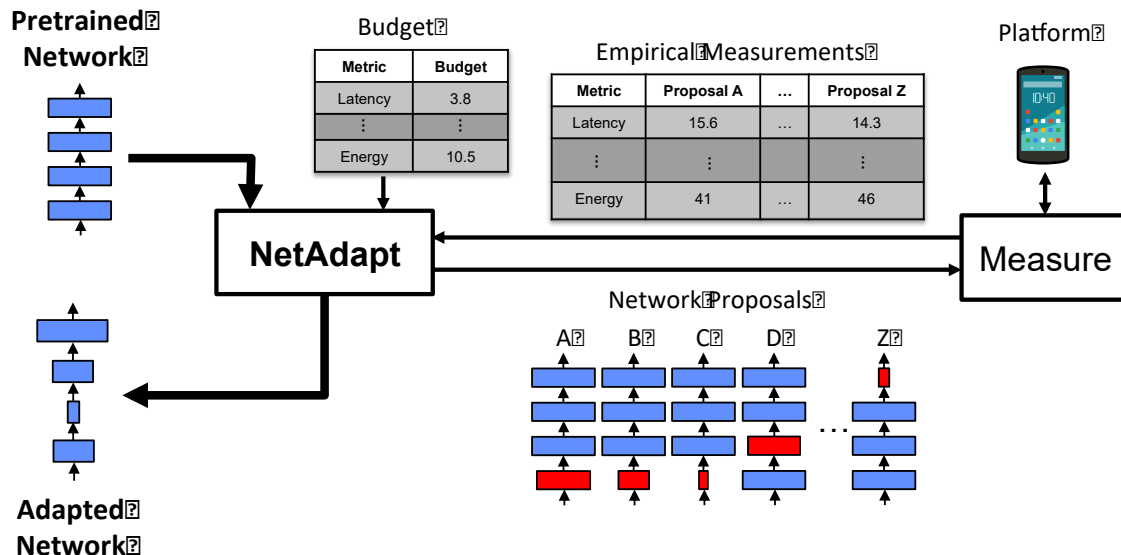
# of operations (MACs) does not approximate latency well



Source: Google (<https://ai.googleblog.com/2018/04/introducing-cvpr-2018-on-device-visual.html>)

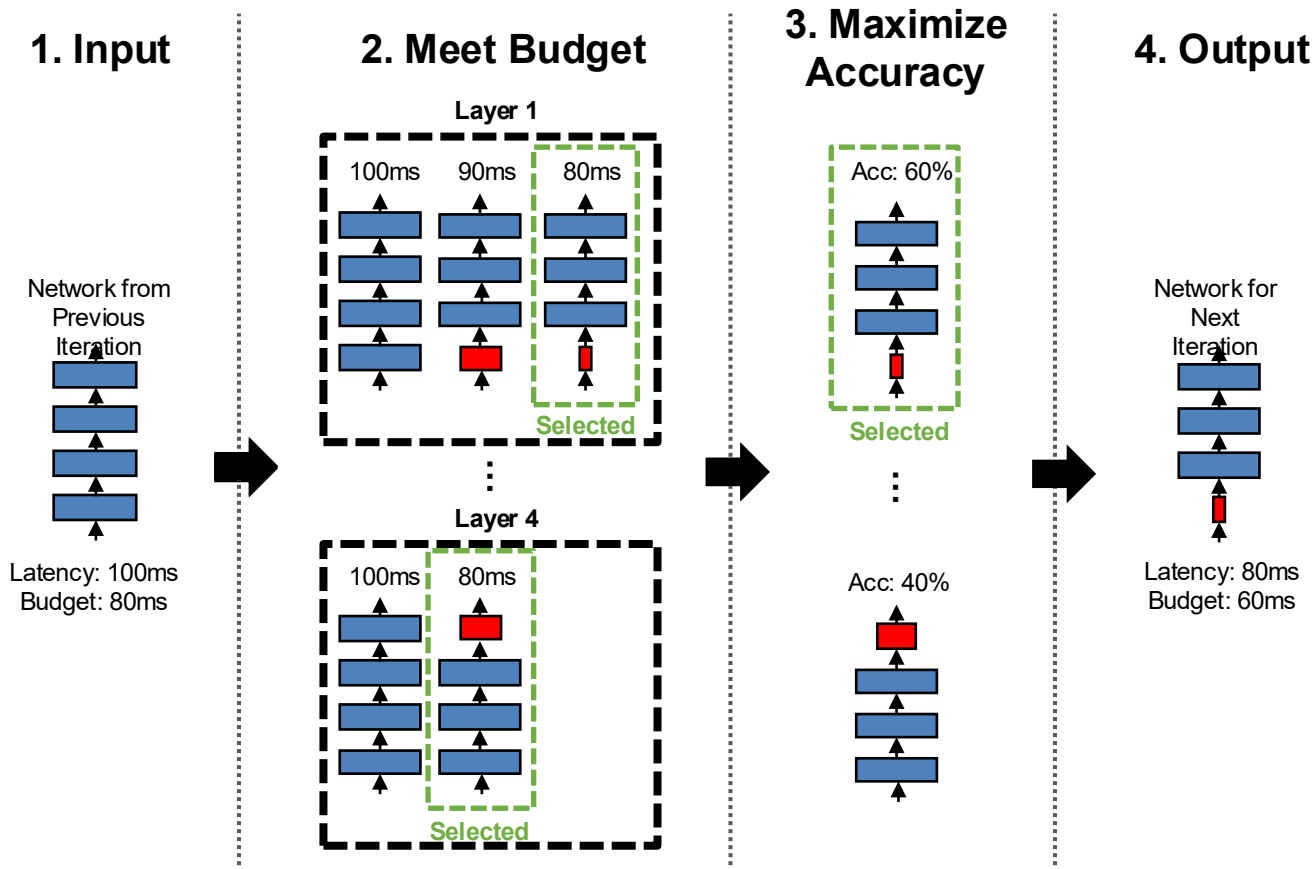
# NetAdapt: Platform-Aware DNN Adaptation

- Automatically adapt DNN to a mobile platform to reach a target latency or energy budget
- Use empirical measurements to guide optimization (avoid modeling of tool chain or platform architecture)
- Few hyperparameters to reduce tuning effort



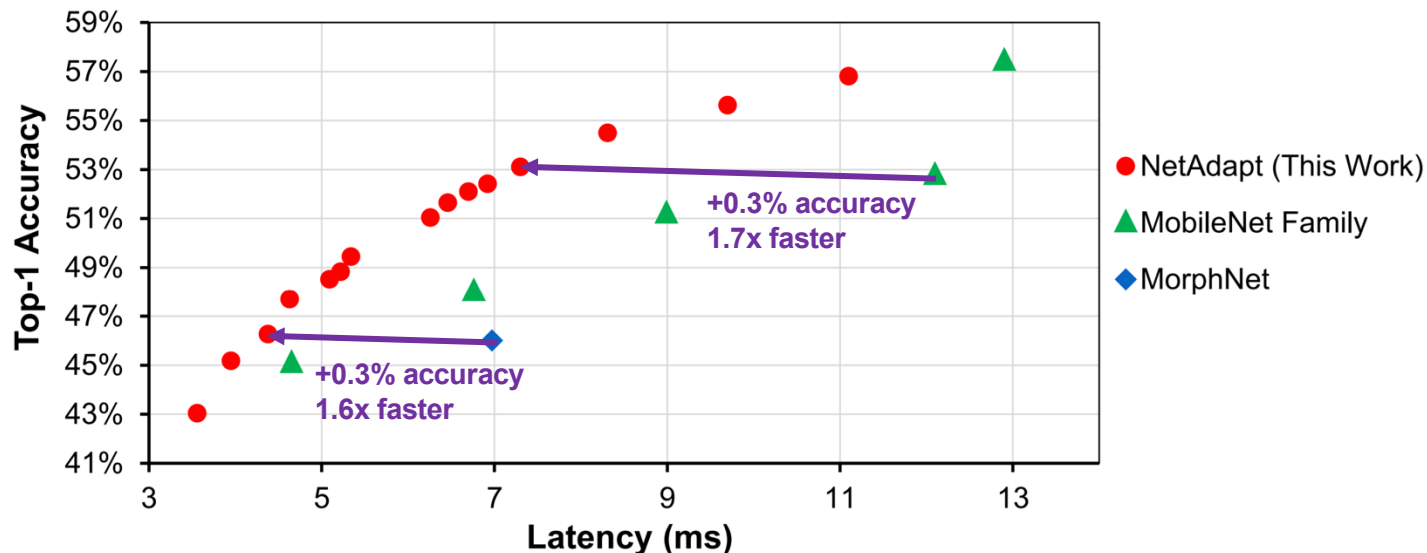
Code available at <http://netadapt.mit.edu>

# Simplified Example of One Iteration



# Improved Latency vs. Accuracy Tradeoff

Increase the real inference speed of MobileNet by up to 1.7x with similar accuracy



\*Tested on the ImageNet dataset and a Google Pixel 1 CPU

Reference:

**MobileNet:** Howard et al, "Mobilenets: Efficient convolutional neural networks for mobile vision applications", arXiv 2017

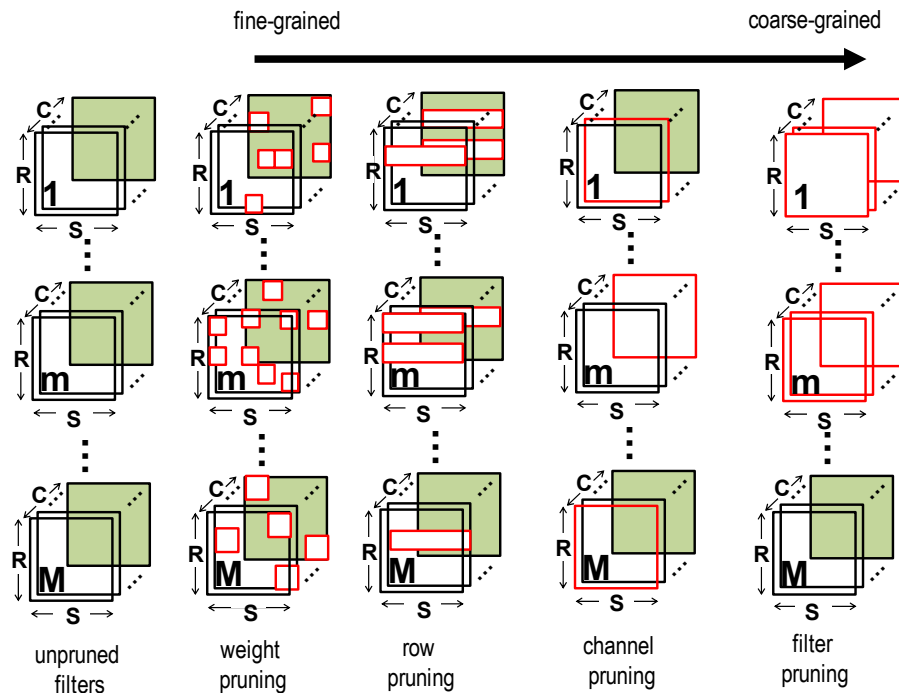
**MorphNet:** Gordon et al., "Morphnet: Fast & simple resource-constrained structure learning of deep networks", CVPR 2018

# Using Direct Metrics is Important

- If NetAdapt was guided by the number of MACs, it would achieve a better accuracy-MAC trade-off
- However, it does not mean lower latency
- It is important to incorporate direct metrics rather than indirect metrics into the design of DNNs

Network	Top-1 Accuracy	# of MACs (M)	Latency (ms)
Small MobileNet V1	45.1 (+0)	13.6 (100%)	4.65 (100%)
NetAdapt	46.3 (+1.2)	11.0 (81%)	6.01 (129%)
Large MobileNet V1	68.8 (+0)	325.4 (100%)	69.3 (100%)
NetAdapt	69.1 (+0.3)	284.3 (87%)	74.9 (108%)

# Weight Removal: Grouping



## Benefits:

Increase coarseness → more structure in sparsity (easier for hardware)

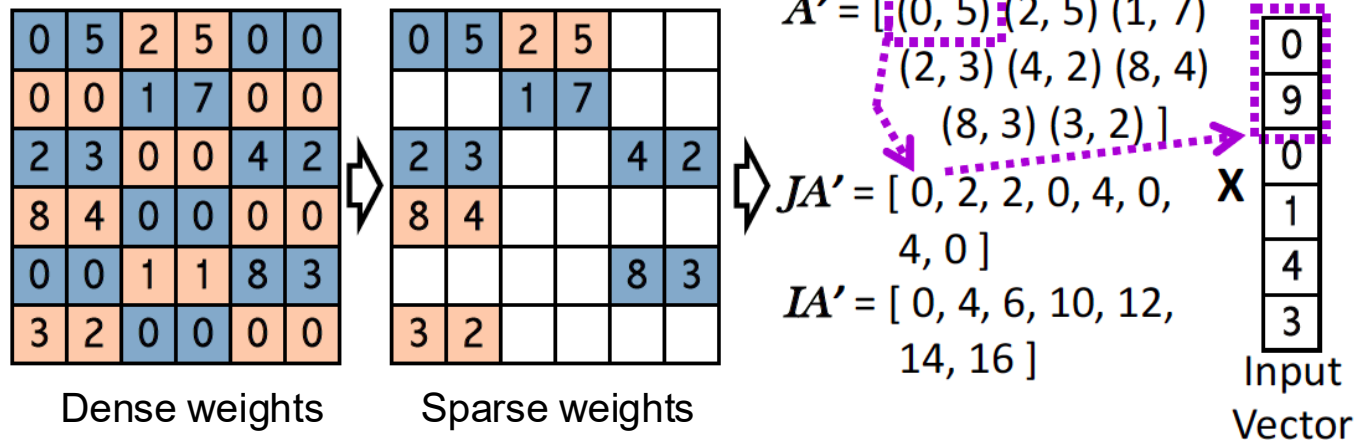
Less signaling for location of zeros → better compression

# Coarse-Grained Pruning

- **Scalpel**

- Prune to match the underlying data-parallel hardware organization for speed up (1.92x over unstructured)

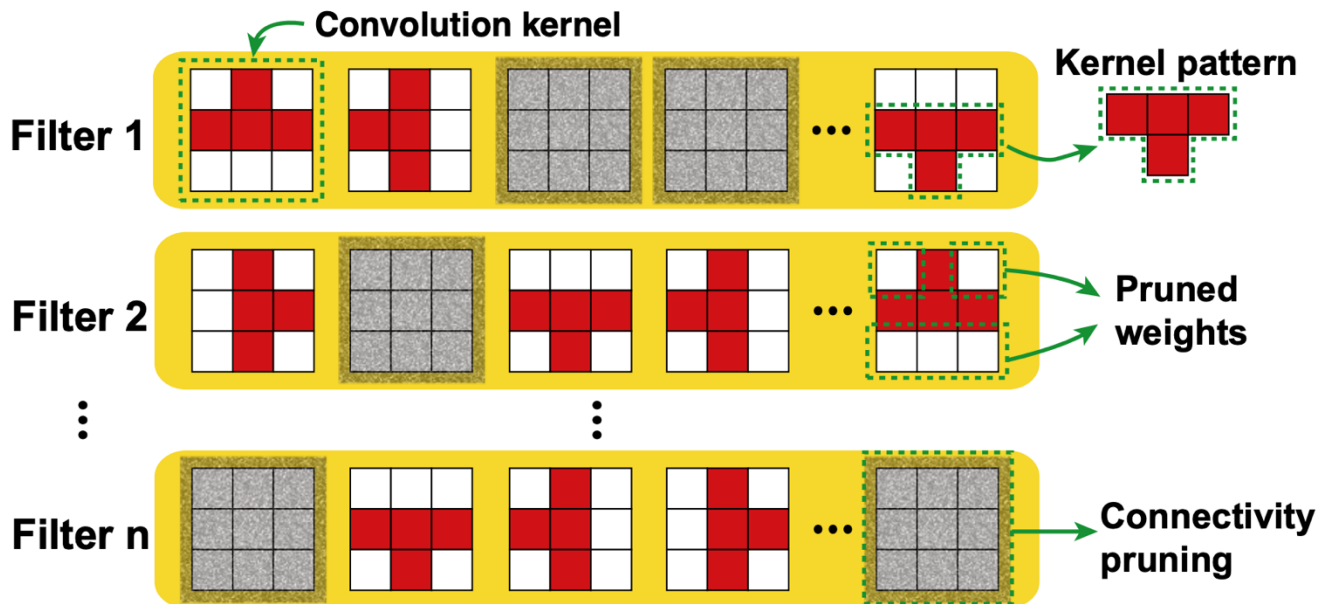
*Example: 2-way SIMD*



[Yu, /ISCA 2017]

# Pattern-Based Weight Pruning

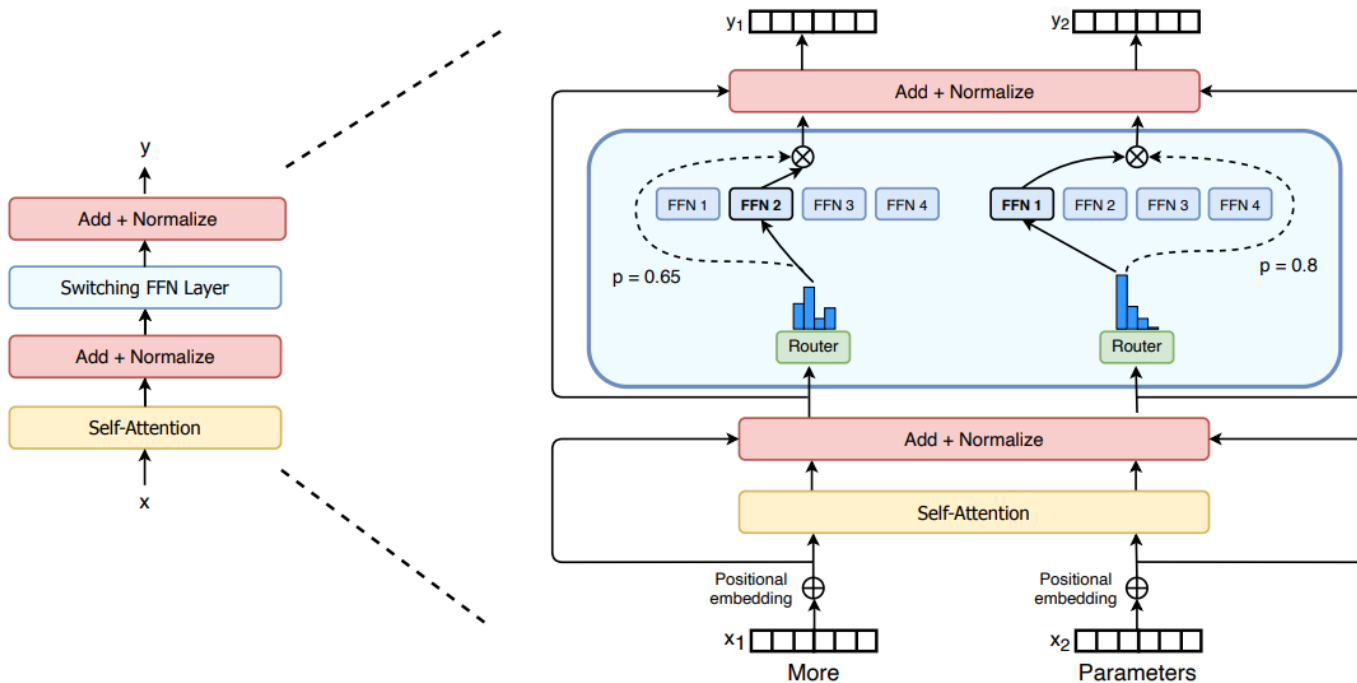
Prune based on pattern (rather than row)



[PCONV, AAAI 2020], [PatDNN, ASPLOS 2020]

# Mixture of Expert (MoE)

MoE can be thought of as Coarse-Grain Sparsity



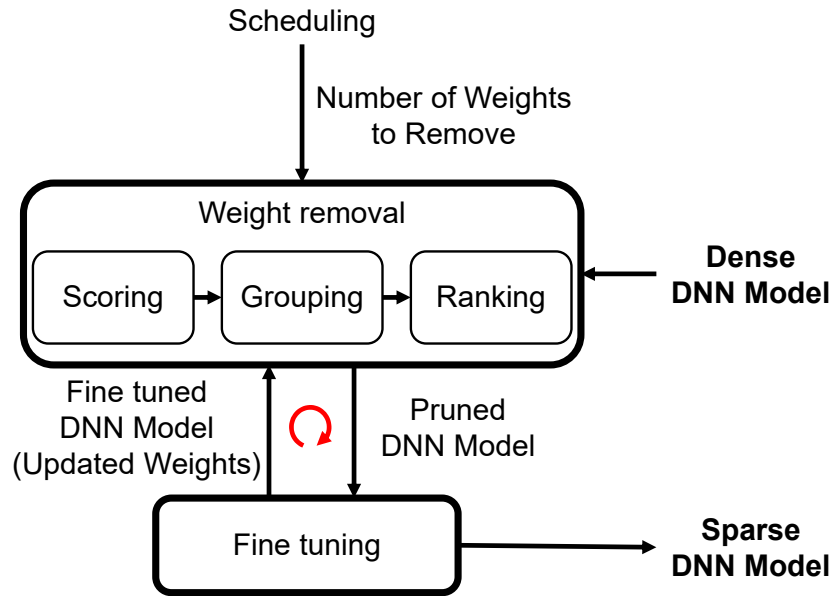
# Weight Removal: Ranking

---

- The weights are ranked based on their scores.
- Depending on grouping, each weight can be ranked individually, or each group of weights are ranked relative to other groups.
- The likelihood that each weight or group of weights is removed is based on its rank.

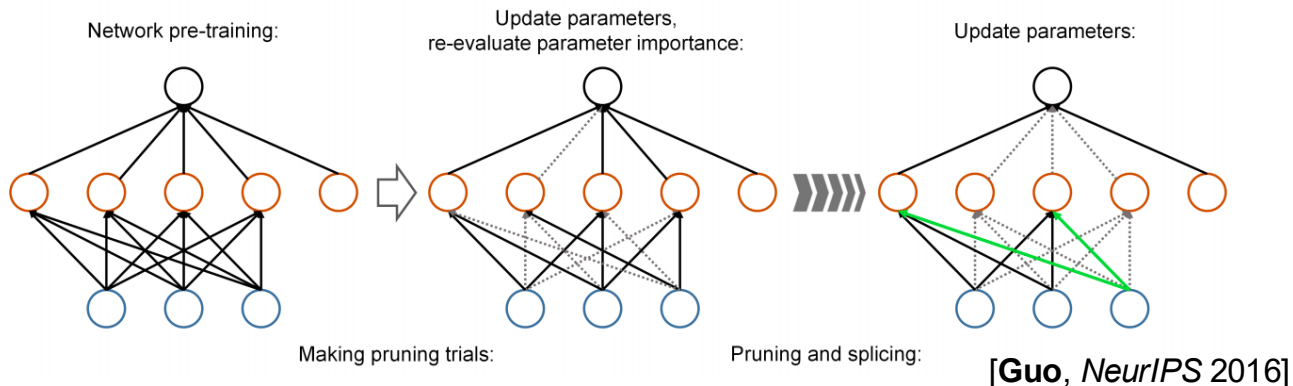
# Fine tuning and Scheduling

- **Fine tuning:** Update the values of the remaining weights to restore accuracy
- **Scheduling:** Determine how many weights to prune in each iteration



# Fine Tuning: Restoring

Allow weights to be **restored** during pruning process (splicing)

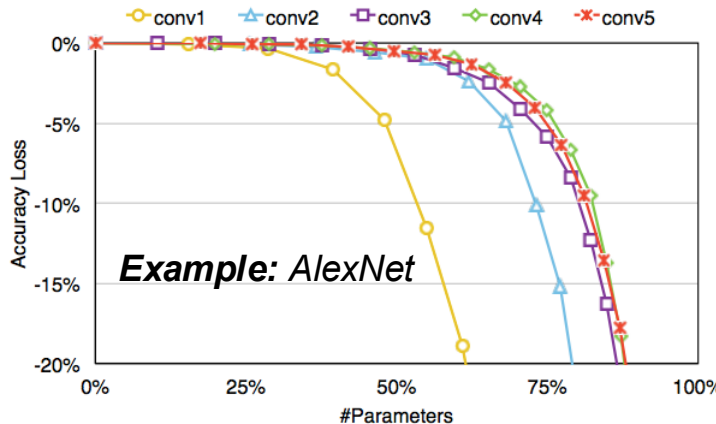


Number of  
non-zero weights  
reduced by ~2x

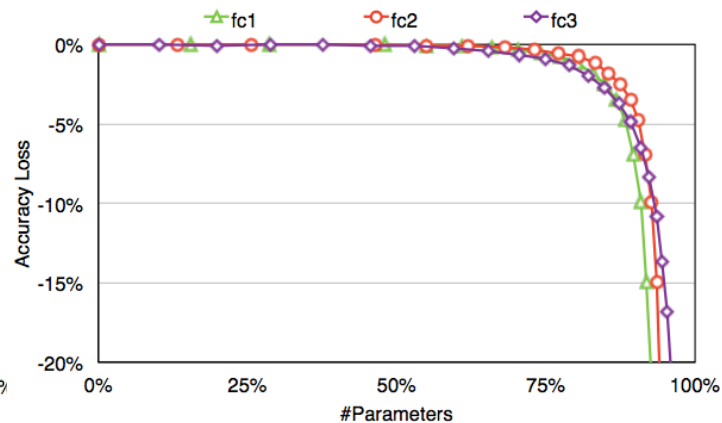
Layer	Params.	w/o splicing	w/ splicing
		Params.% [9]	Params.% (Ours)
conv1	35K	~ 84%	53.8%
conv2	307K	~ 38%	40.6%
conv3	885K	~ 35%	29.0%
conv4	664K	~ 37%	32.3%
conv5	443K	~ 37%	32.5%
fc1	38M	~ 9%	3.7%
fc2	17M	~ 9%	6.6%
fc3	4M	~ 25%	4.6%
Total	61M	~ 11%	<b>5.7%</b>

# Interplay: Pruning and Layer Types

## Convolutional Layers



## Fully Connected Layers

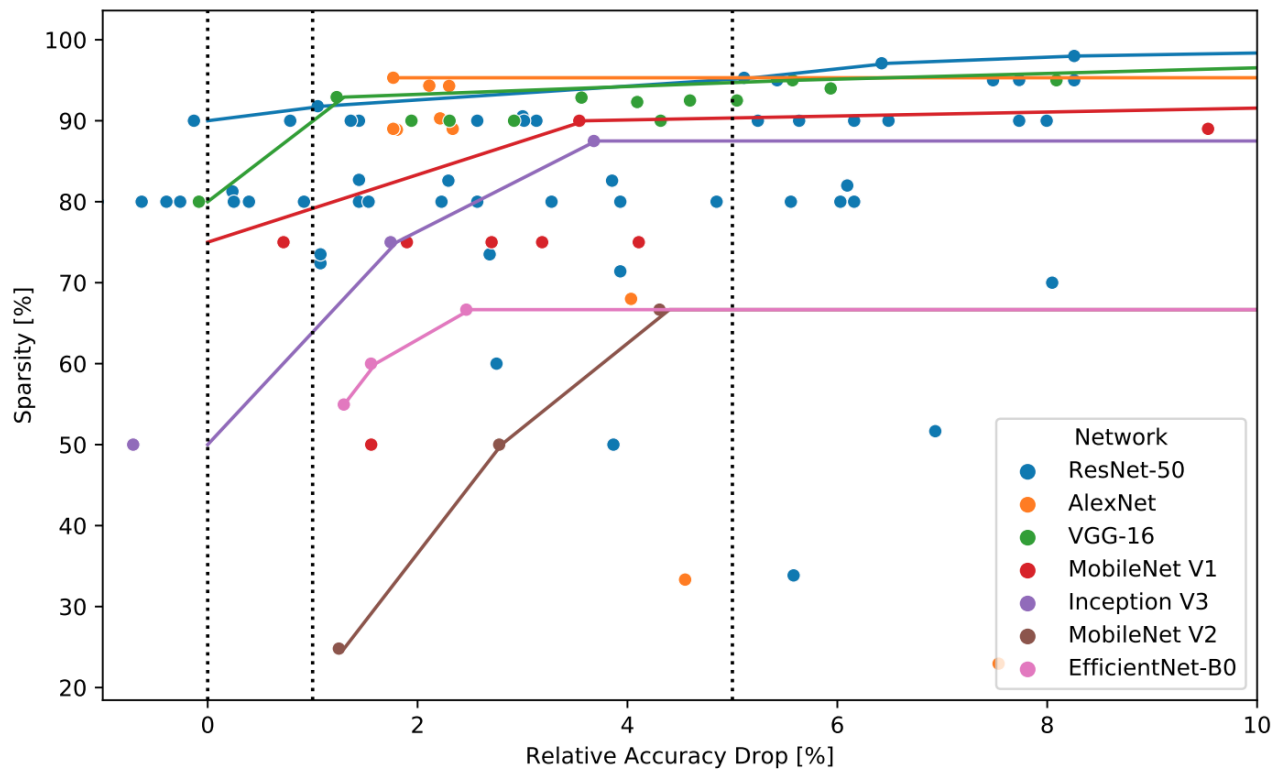


For AlexNet

**Weight Reduction: CONV layers 2.7x, FC layers 9.9x**  
*(Most reduction on fully connected layers)*

**Overall: 9x weight reduction, 3x MAC reduction**

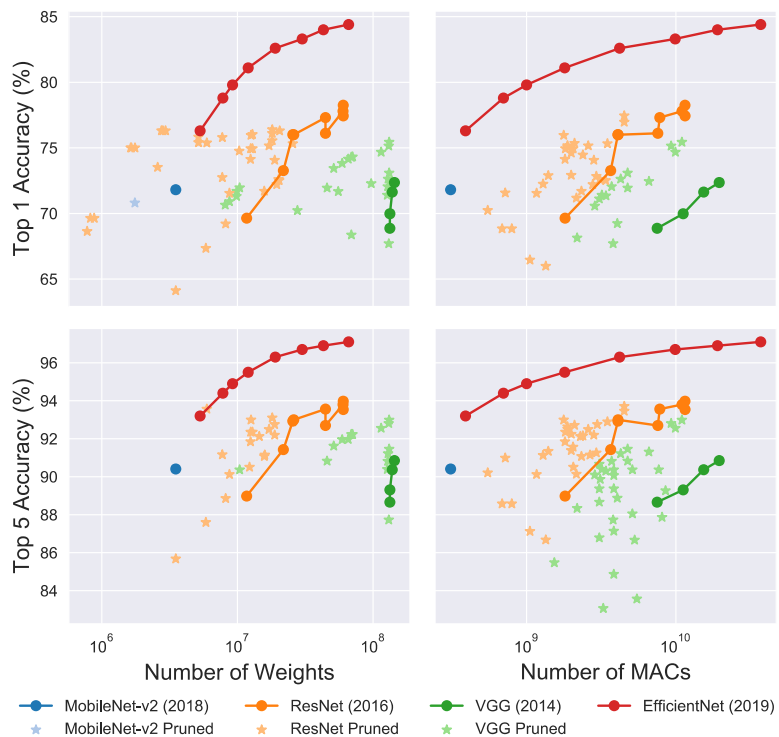
# Interplay: Pruning and Accuracy Loss



Accuracy drops more quickly for modern **efficient** DNN models

# Interplay: Pruning and DNN Model

Speed and Size Tradeoffs for Original and Pruned Models



Using an **unpruned efficient** DNN model can perform better than a **pruned inefficient** DNN model

[Blalock, *MLSys* 2020]

# Summary

---

- Sparsity can be used to reduce number of operations, data movement and storage cost
- Fine tuning can help increase amount of sparsity
- Sparsity on the order of 30-70%
  - Existing software libraries designed for >99%
    - Need specialized hardware to exploit! → Next lecture
  - Coarse grained pruning can also be used to improve speed and storage cost
- Using **direct** hardware metrics (energy, latency) often results in a better accuracy versus complexity tradeoff than **indirect** proxy metrics (number of operations and weights)

# Recommended Reading

---

- Textbook: Section 8.1
  - <https://doi.org/10.1007/978-3-031-01766-7>
- D. Blalock\*, J. J. Gonzalez-Ortiz\*, J. Frankle, J. Gutttag, “What is the State of Neural Network Pruning?,” MLSys 2020
  - <https://proceedings.mlsys.org/papers/2020/73>
- T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, A. Peste, “Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks,” JMLR 2021
  - <https://jmlr.org/papers/volume22/21-0366/21-0366.pdf>