6.5930/1
Hardware Architectures for Deep Learning

Co-Design of DNN Models and Hardware: Sparsity

April 1, 2024

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 **<u>pruning!</u>**
- 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

$$anything \times 0 = 0$$

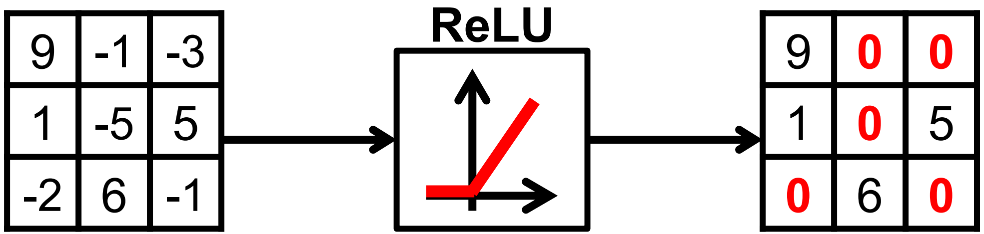
$$anything + 0 = anything$$

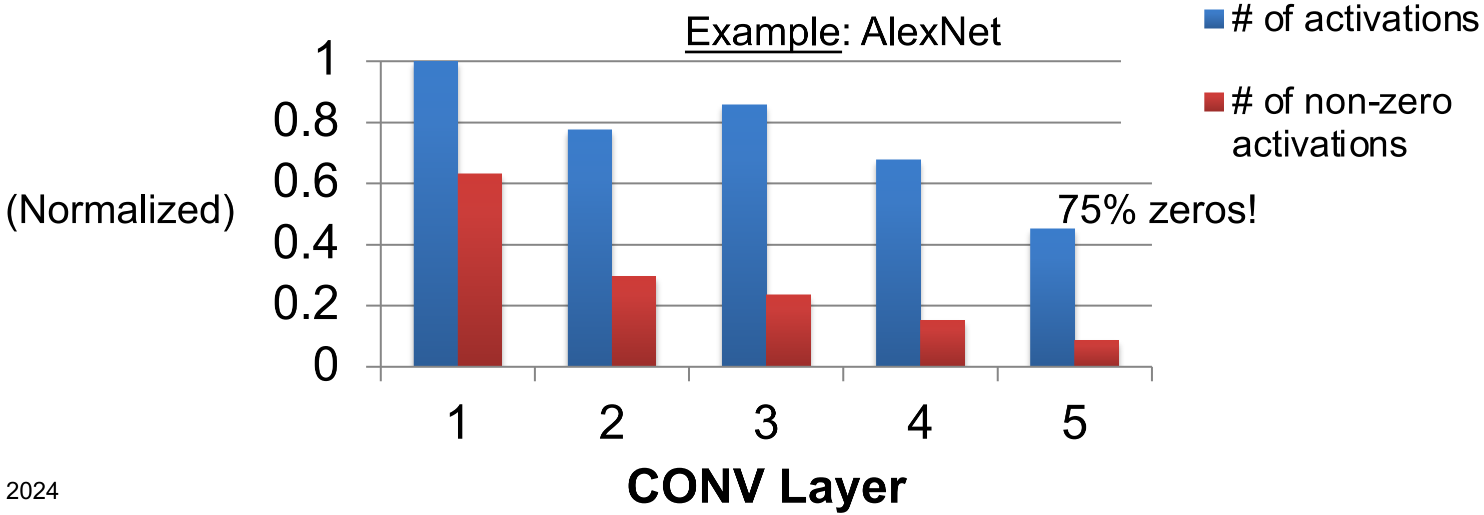
Can save time and energy by avoiding fetching unnecessary operands and avoiding **ineffectual** computations

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 reduces 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

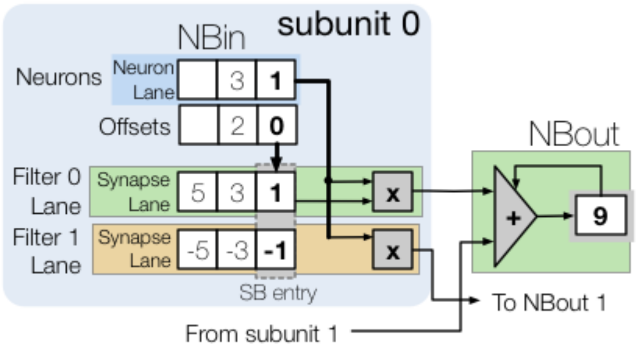
<u>Example</u>

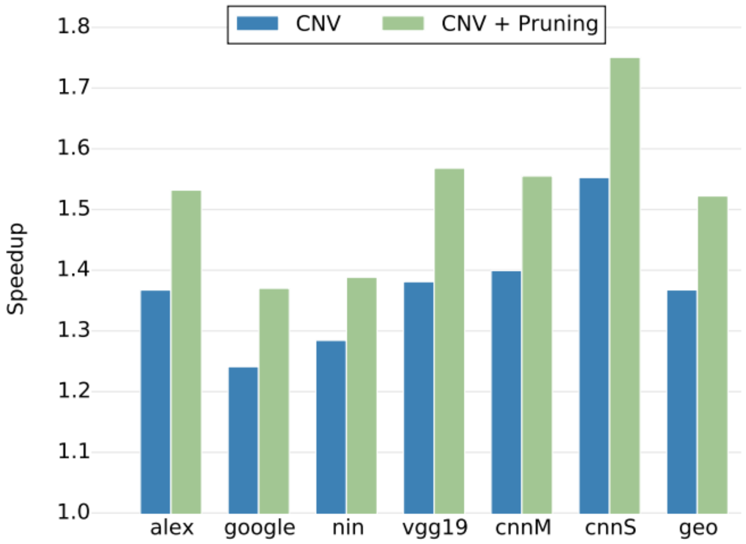
$L = 4$	(not uniquely decodable)
r_0	0
r_1	1
r_2	0 0
r_3	0 1

$L = 4$	(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)



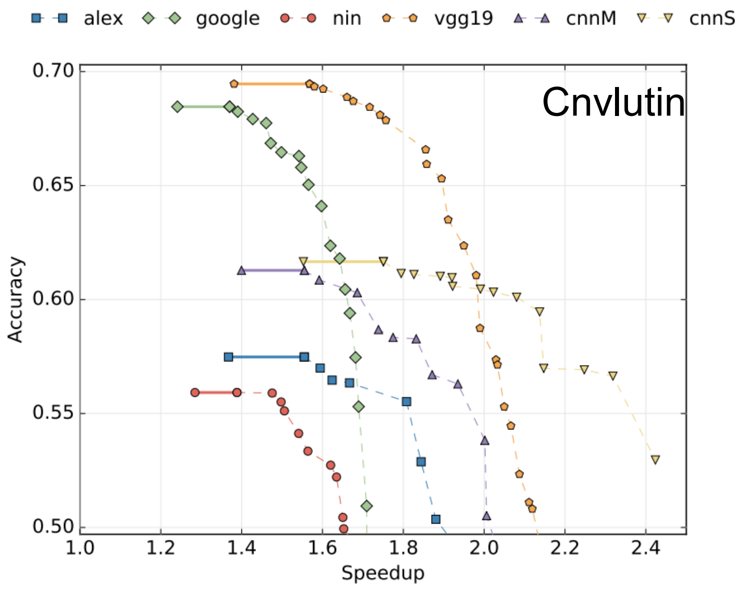


[**Albericio**, *ISCA* 2016]

Pruning Activations

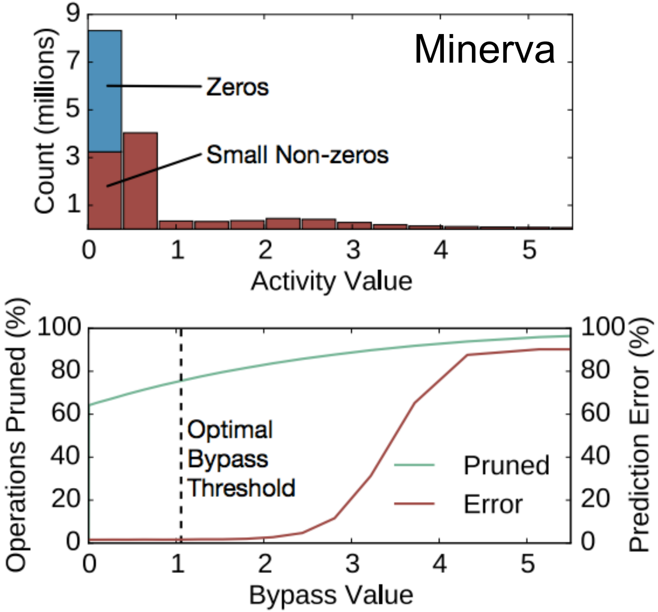
Remove small activation values ***(affects accuracy!)***

Speed up 11% (ImageNet)



[**Albericio**, *ISCA* 2016]

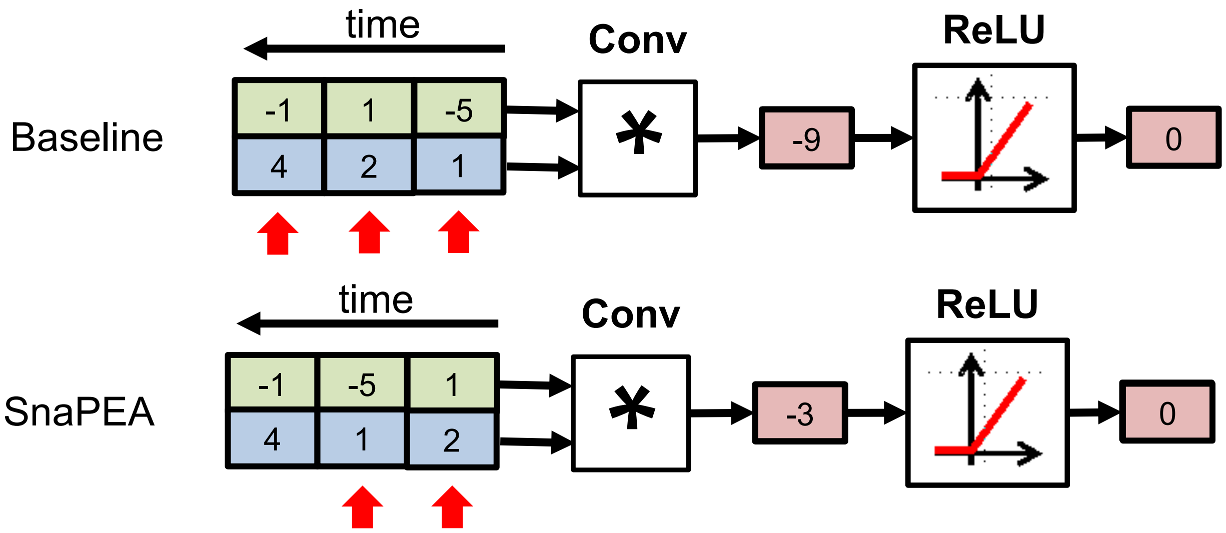
Reduce power 2x (MNIST)



[**Reagen**, *ISCA* 2016]

Exploit ReLU

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

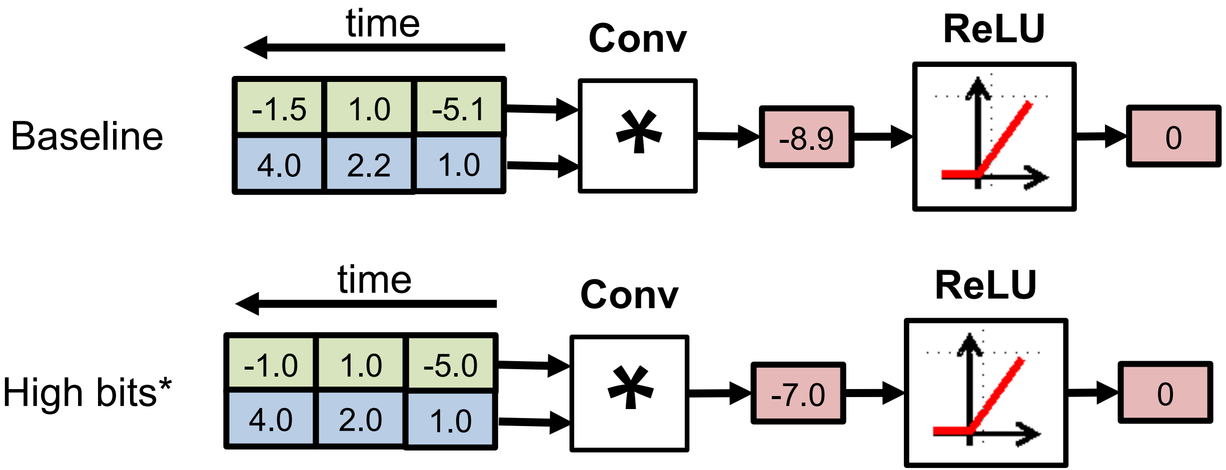


Additional hardware required to decide when to terminate

[**SnaPEA**, *ISCA* 2018]

Exploit ReLU

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



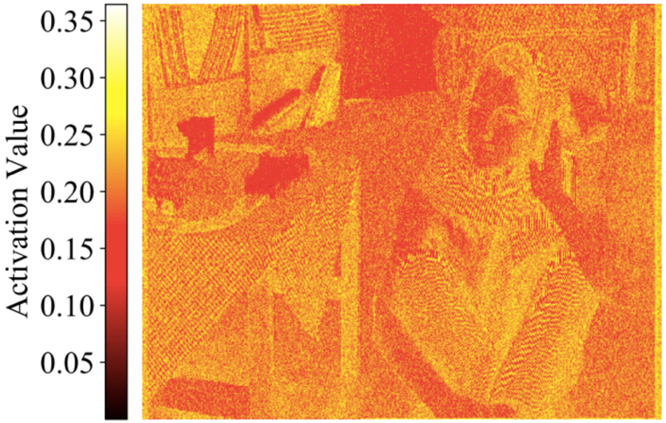
Only compute on low bits if result is positive

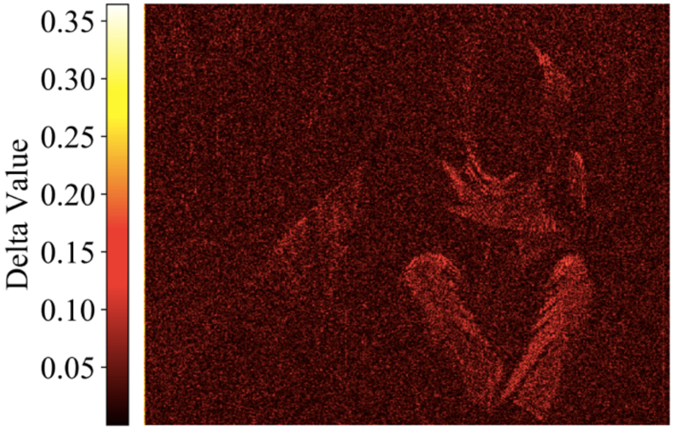
*over-simplified

[**PredictiveNet**, *ISCAS* 2017], [**Song**, *ISCA* 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

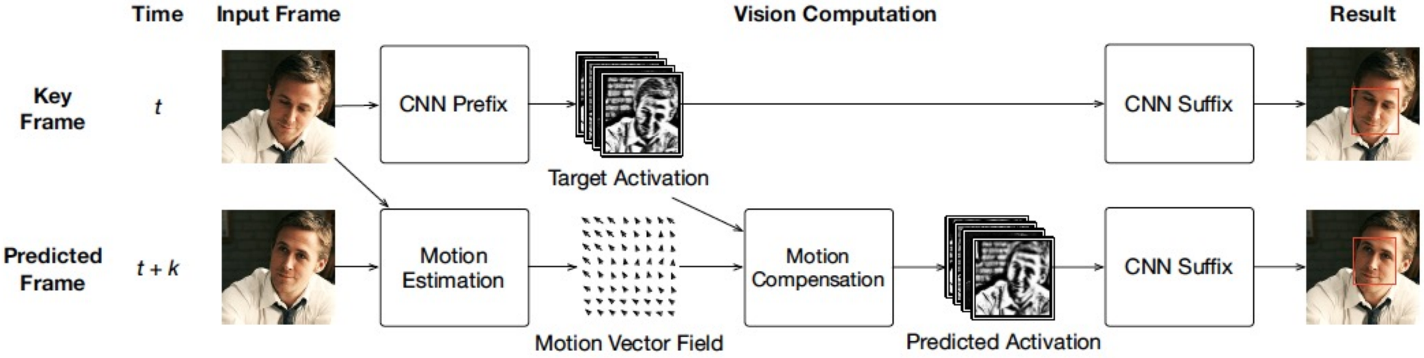
$$y_1 = a_1 \times w$$

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

[**Diffy**, *MICRO* 2018]

Exploit Temporal Correlation of Inputs

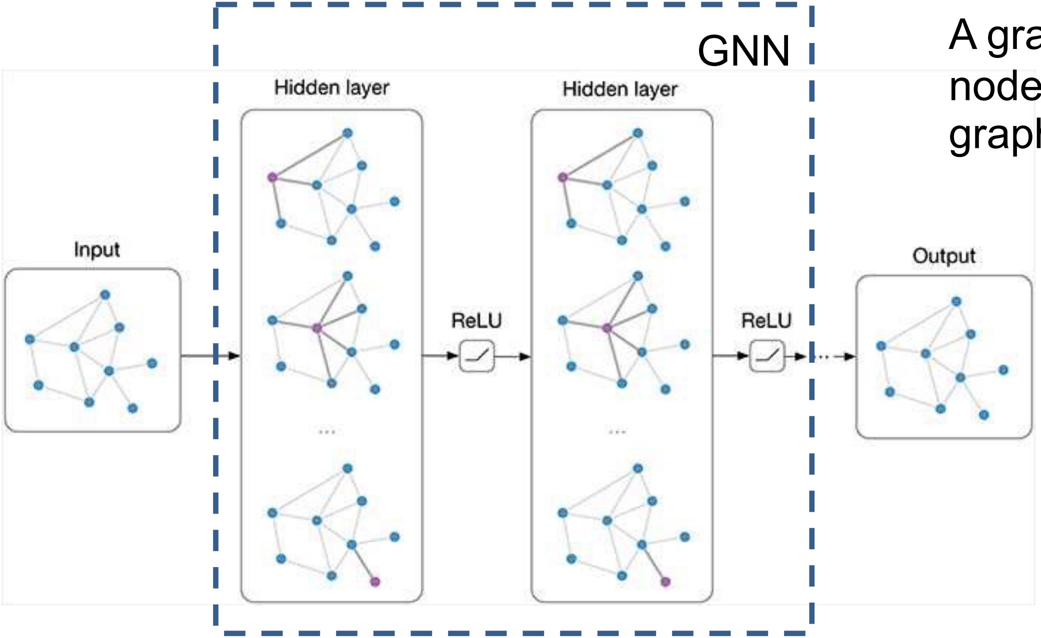
- Reduce amount of computation if there is temporal correlation between inputs (e.g., frames)
- Requires additional storage and need to find redundancy (e.g., motion vectors for videos)
- Application specific (e.g., videos) – requires that the same operation is done for each frame (not always the case)



[**EVA²**, *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].

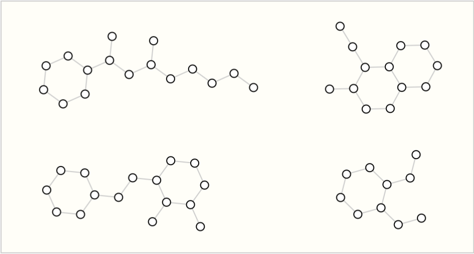
Image Source: https://tkipf.github.io/graph-convolutional-networks/

[**Xu**, *ICLR* 2019]

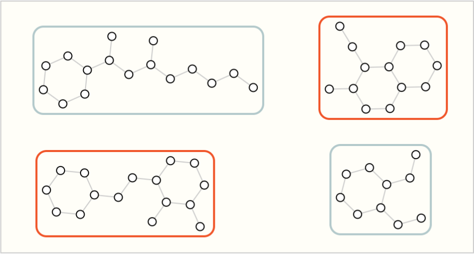
Example Graph Neural Networks Tasks

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

Graph example

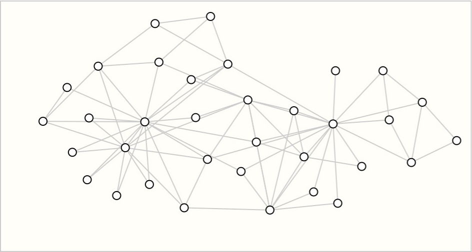


Input: graphs

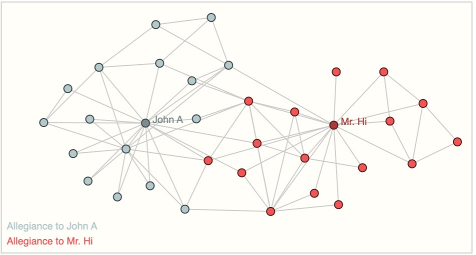


Output: labels for each graph, (e.g., "does the graph contain two rings?")

Nodes example

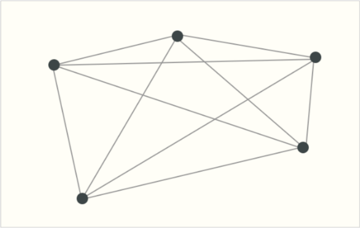


Input: graph with unlabled nodes

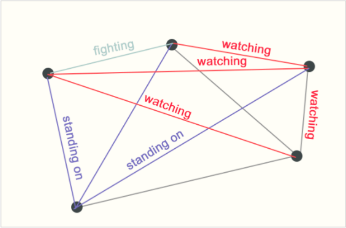


Output: graph node labels

Edge example



Input: fully connected graph, unlabeled edges

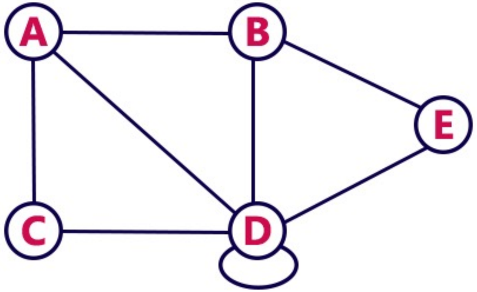


Output: labels for edges

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



$$
\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 1 \\
C & 1 & 0 & 0 & 1 & 0 \\
D & 1 & 1 & 1 & 1 & 1 \\
E & 0 & 1 & 0 & 1 & 0
\end{array}
$$

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

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

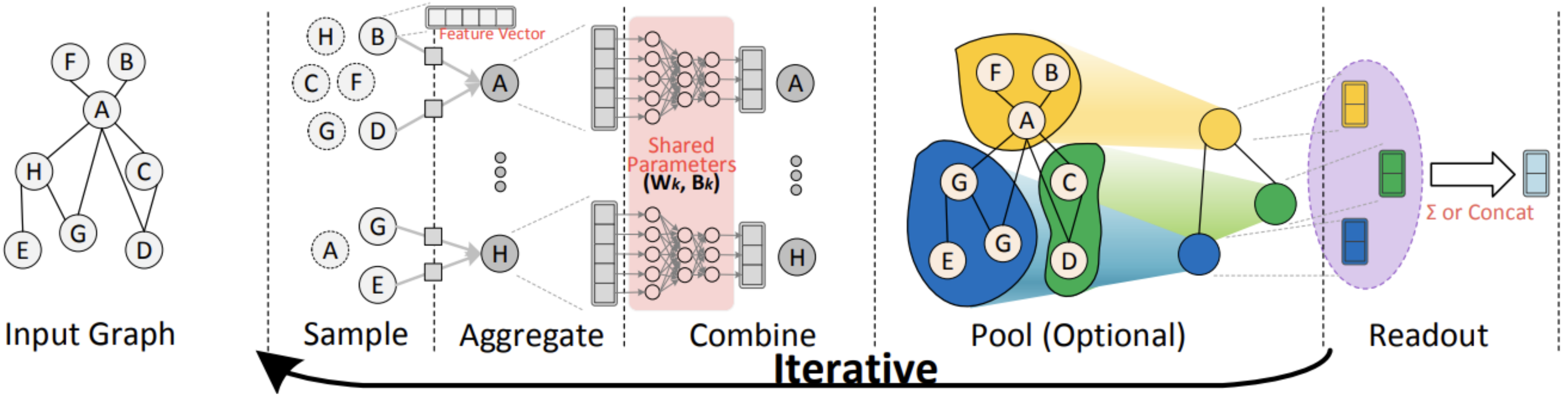


Image source: [**Yan**, *HPCA* 2020]

Computation in GNN

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

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

$$\cdots$$

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

Normalized Adjacency Matrix

Feature Matrix (typically, all nodes)

Weights

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 [A A B]: Output = A*(1+3)+B*1

2 multiplies and 2 weight reads

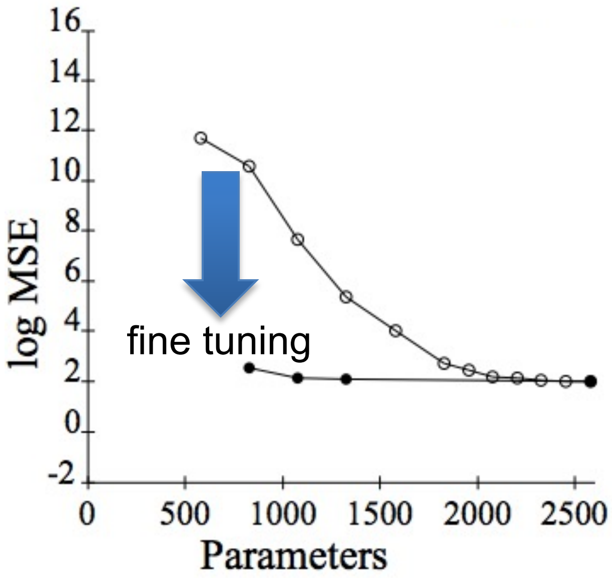
Note: Bitwidth of multiplication may need to increase

[**UCNN**, *ISCA* 2018]

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

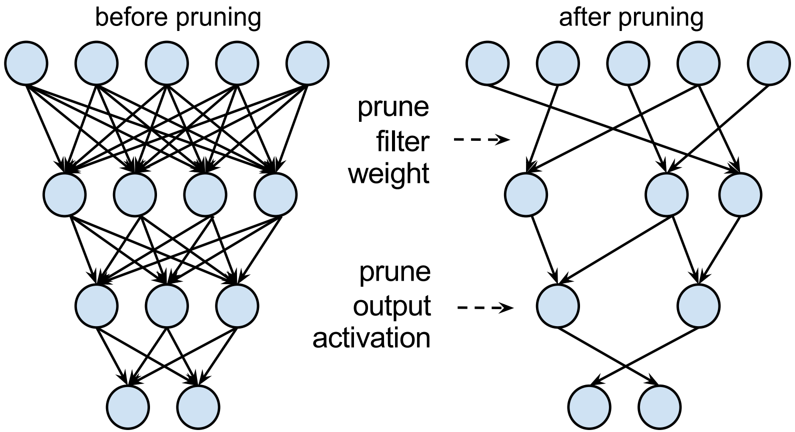


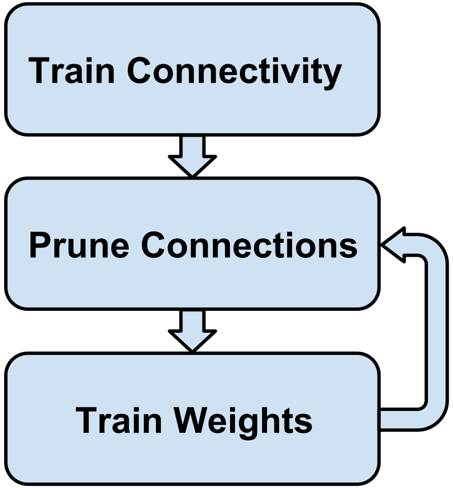
[**Lecun**, *NeurIPS* 1989]

Pruning – Make Weights Sparse

Prune based on ***magnitude*** of weights

[Hertz et al., Neural Computation, 1991]

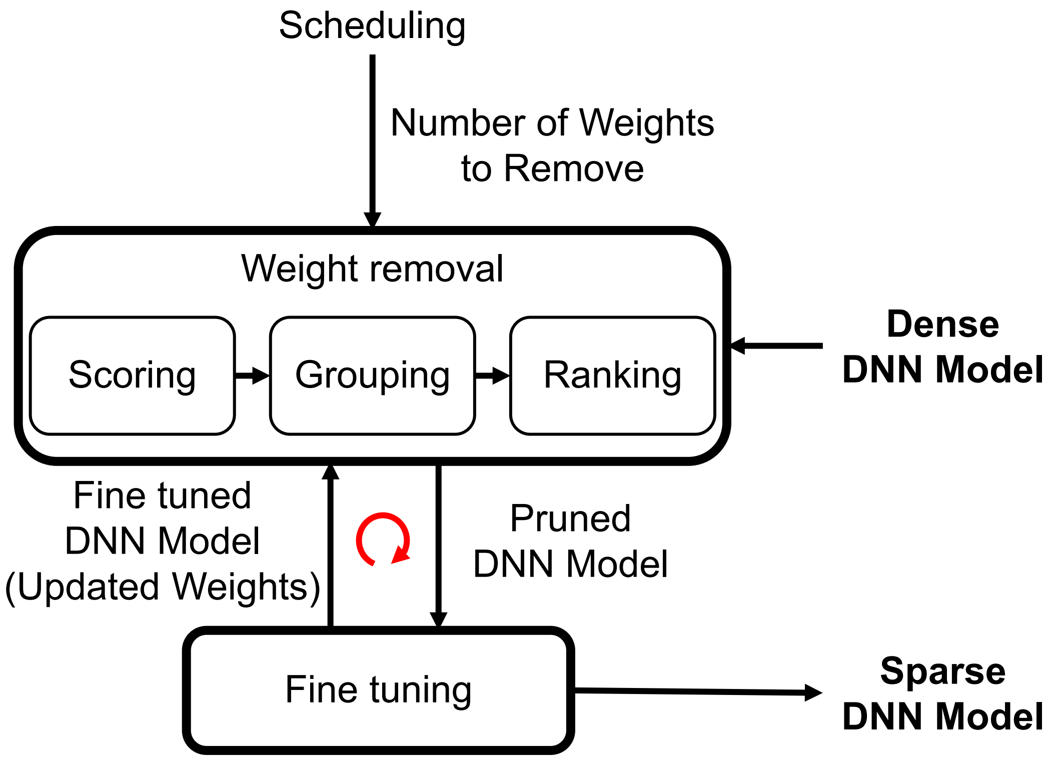




Typical numbers: 50% sparsity without retraining, 80% with retraining

[**Han**, *NeurIPS* 2015]

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

$$\underset{\textbf{fmap}}{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}} * \underset{\textbf{filter}}{\begin{bmatrix} -8 & 3 & 2 \\ 1 & -3 & -2 \\ 1 & 1 & 1 \end{bmatrix}} = \boxed{-4}$$

Magnitude-based pruning

$$\underset{\textbf{fmap}}{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}} * \underset{\textbf{filter}}{\boxed{\begin{bmatrix} -8 & 3 & 2 \\ 1 & -3 & -2 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow{\textbf{Prune}} \begin{bmatrix} -8 & 3 & 2 \\ 0 & -3 & -2 \\ 0 & 0 & 0 \end{bmatrix}}} = \boxed{-8}$$

Error = -4

Feature-based pruning

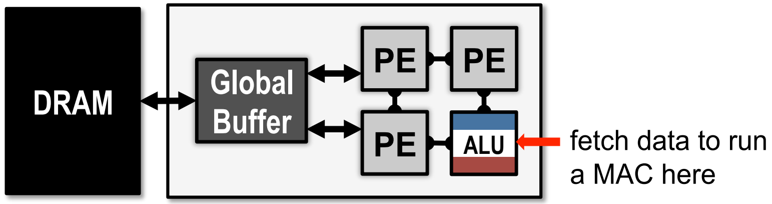
$$\underset{\textbf{fmap}}{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}} * \underset{\textbf{filter}}{\boxed{\begin{bmatrix} -8 & 3 & 2 \\ 1 & -3 & -2 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow{\textbf{Prune}} \begin{bmatrix} -8 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}}} = \boxed{-4}$$

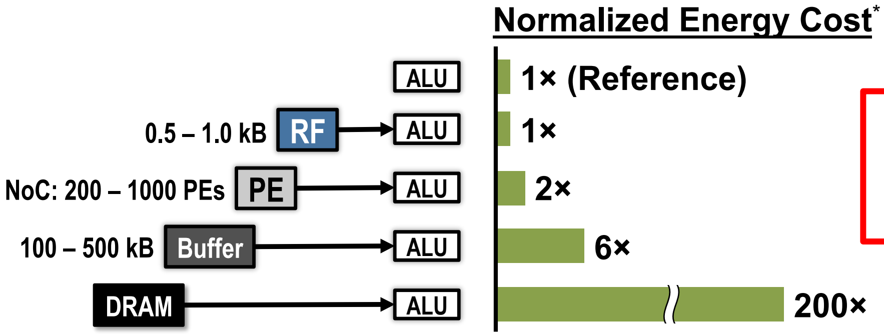
Error = 0

[**Yang**, *CVPR* 2017]

Weight Removal: Scoring

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



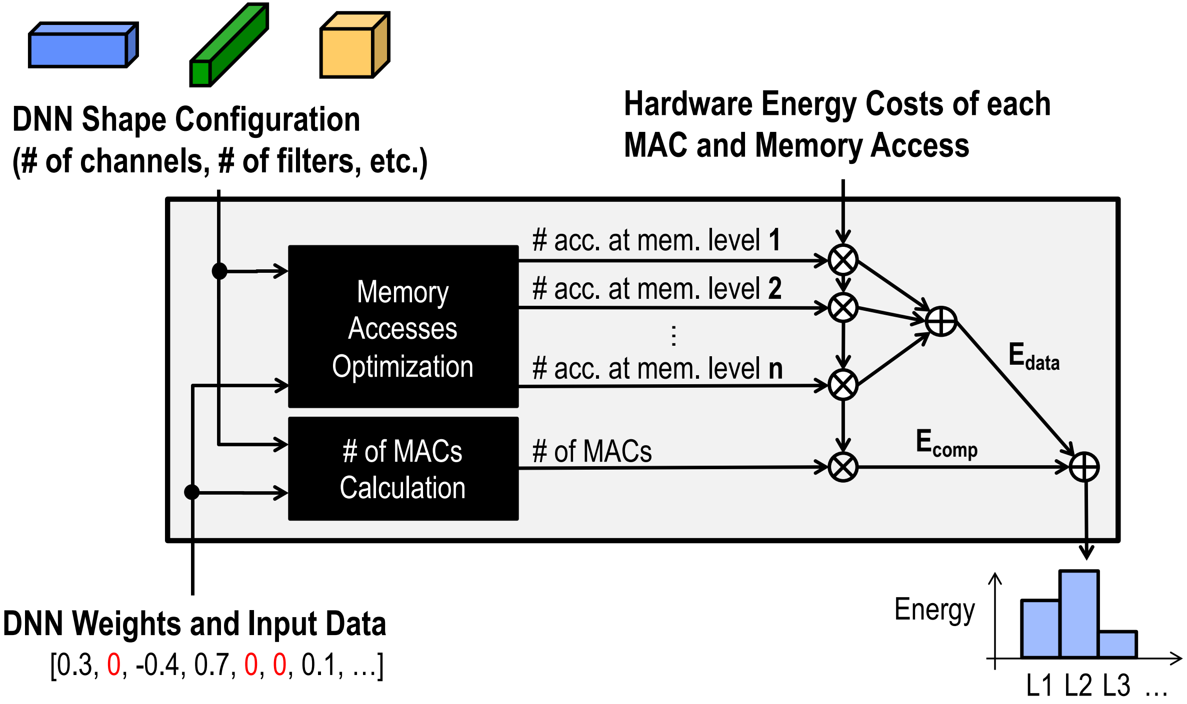


* measured from a commercial 65nm process

Energy of weight depends on **memory hierarchy** and **dataflow**

[Yang, *CVPR* 2017]

Energy Estimation



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

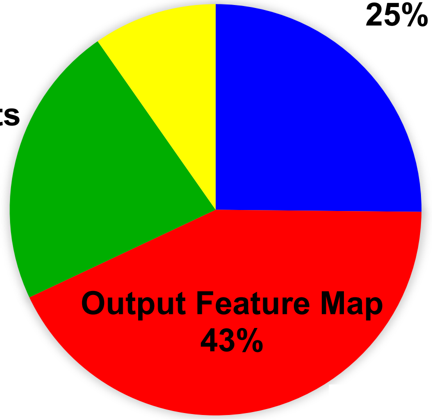
For class, use Timeloop/Accelergy

[Yang, *CVPR* 2017]

Key Insights

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

GoogLeNet Energy Breakdown

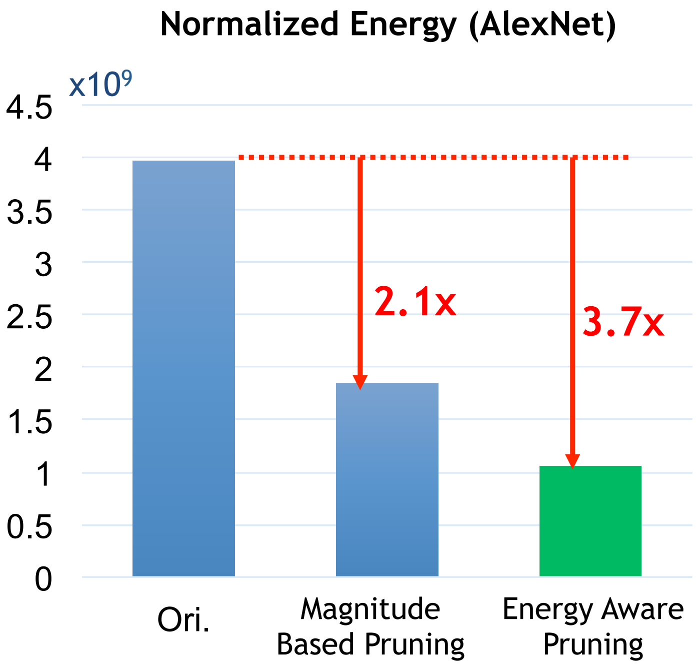


[**Yang**, *CVPR* 2017]

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]

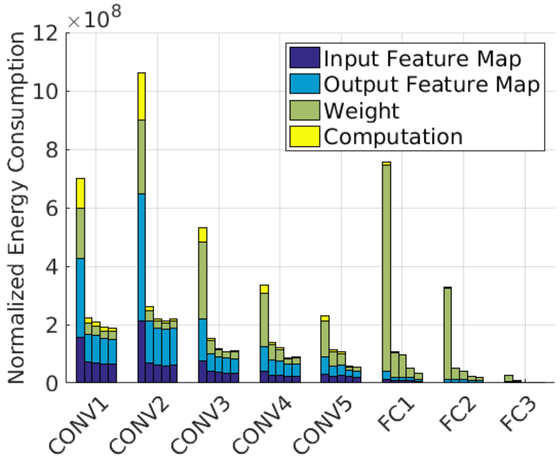
Pruned models available at
http://eyeriss.mit.edu/energy.html

Prune to Reduce Number of Classes

Table 2. Compression ratio[1] of each layer in AlexNet.

# of Classes	[8] 1000	This Work 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%

[1] The number of removed weights divided by the number of total weights. The higher, the better.



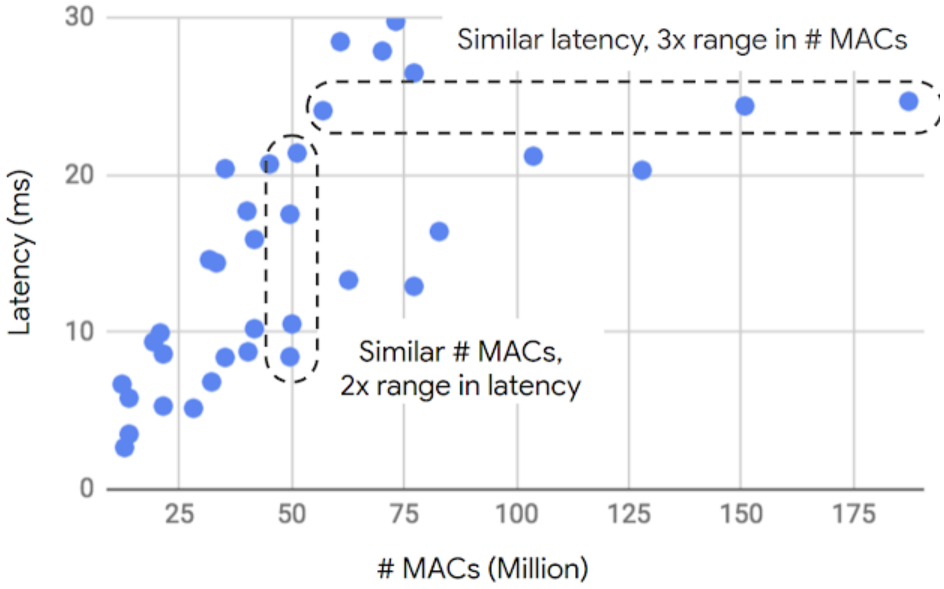
The energy breakdown of the networks in this work. Following the same order as the table.

- 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

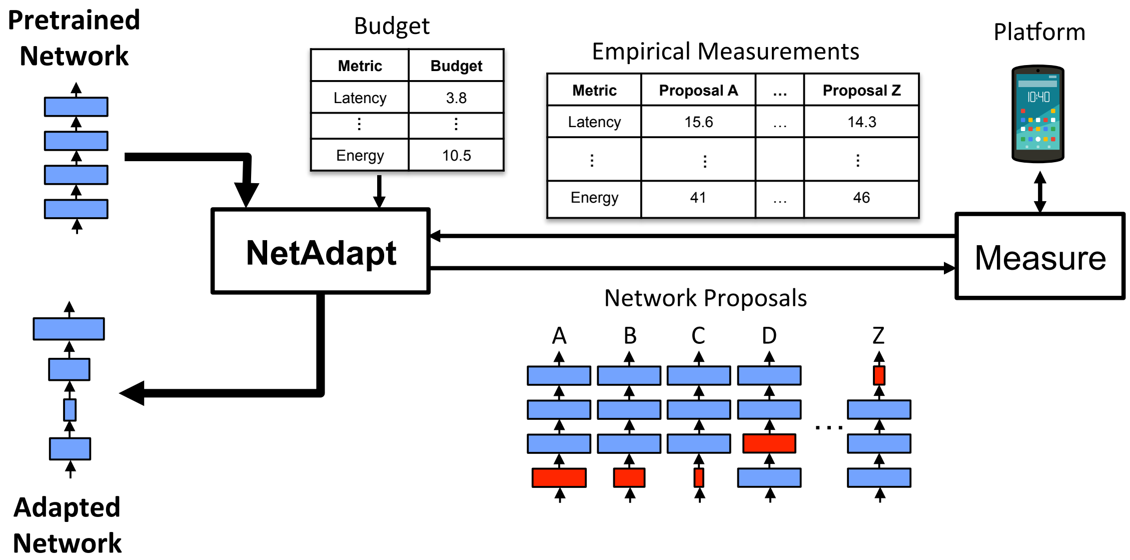
of operations (MACs) does not approximate latency well



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

NetAdapt: Platform-Aware DNN Adaptation

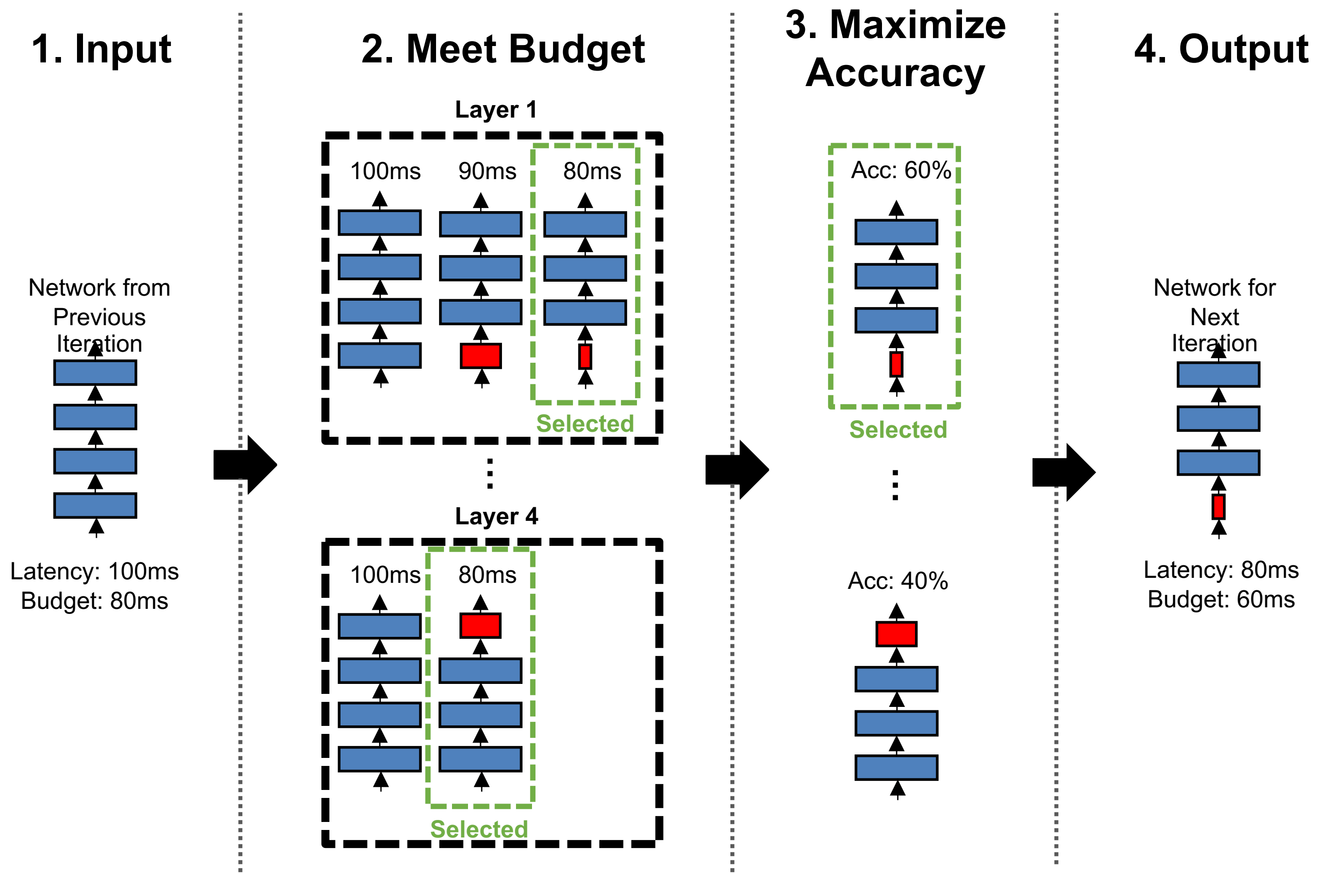
- Automatically adapt DNN to a mobile platform to reach a target latency or energy budget
- Use empirical measurements to guide optimization (avoid modeling of tool chain or platform architecture)
- Few hyperparameters to reduce tuning effort



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

[Yang, *ECCV* 2018]

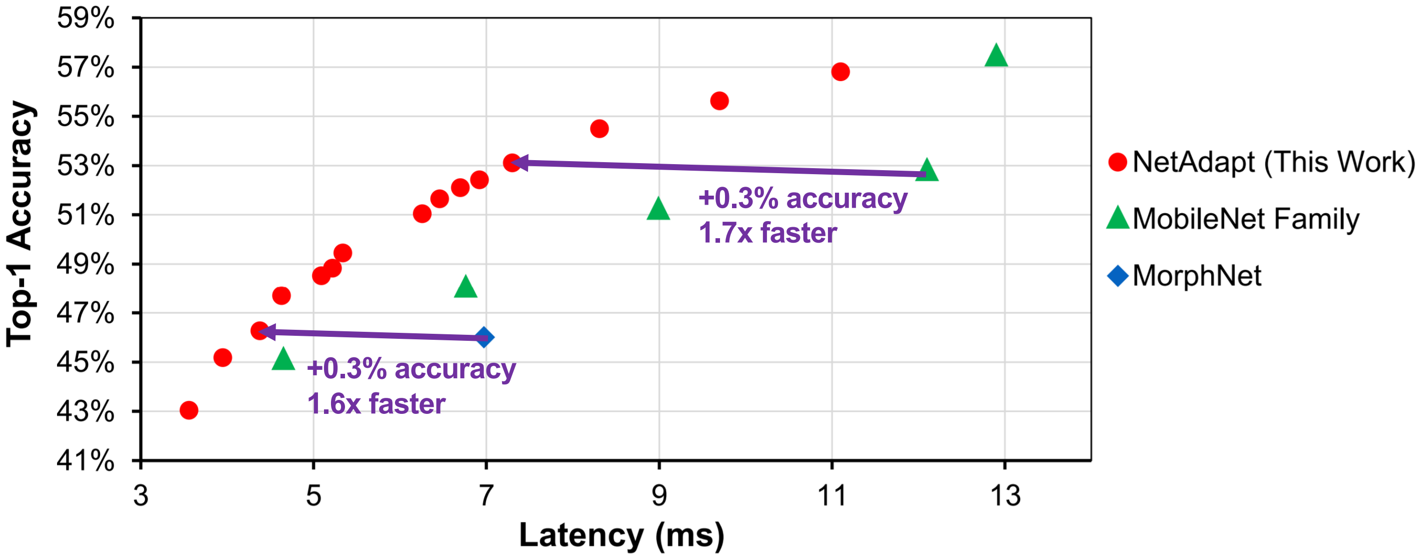
Simplified Example of One Iteration



[Yang, *ECCV* 2018]

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

[**Yang**, *ECCV* 2018]

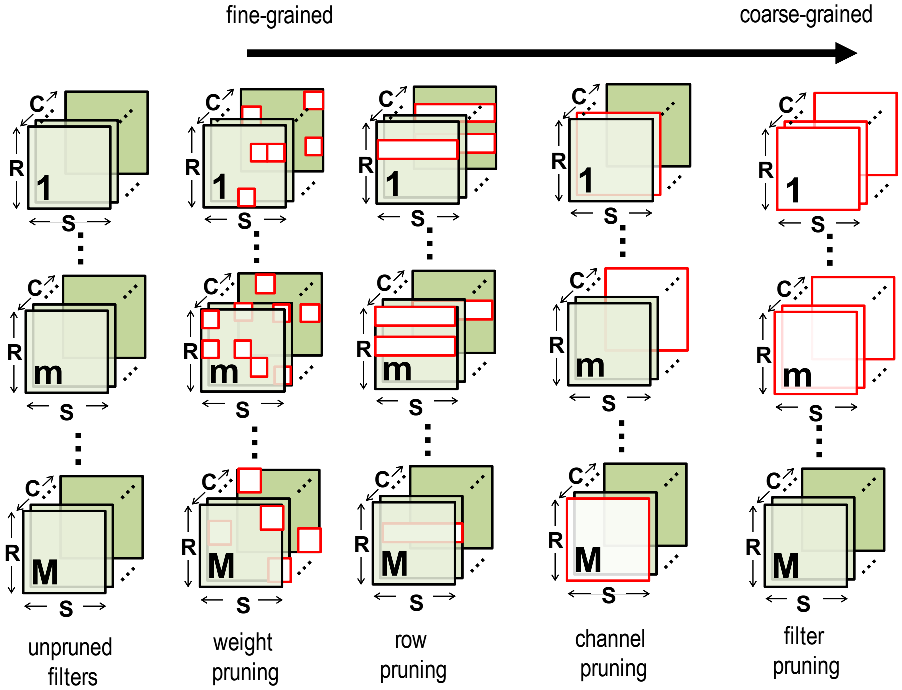
Using Direct Metrics is Important

- 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%)

[**Yang**, *ECCV* 2018]

Weight Removal: Grouping



Benefits:

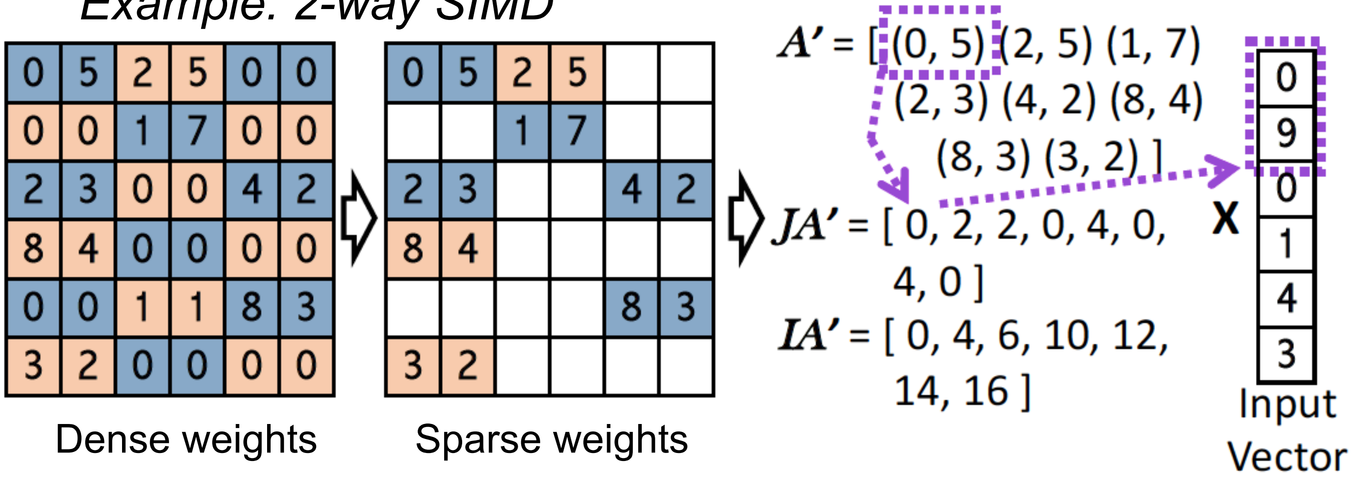
Increase coarseness → more structure in sparsity (easier for hardware)

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)

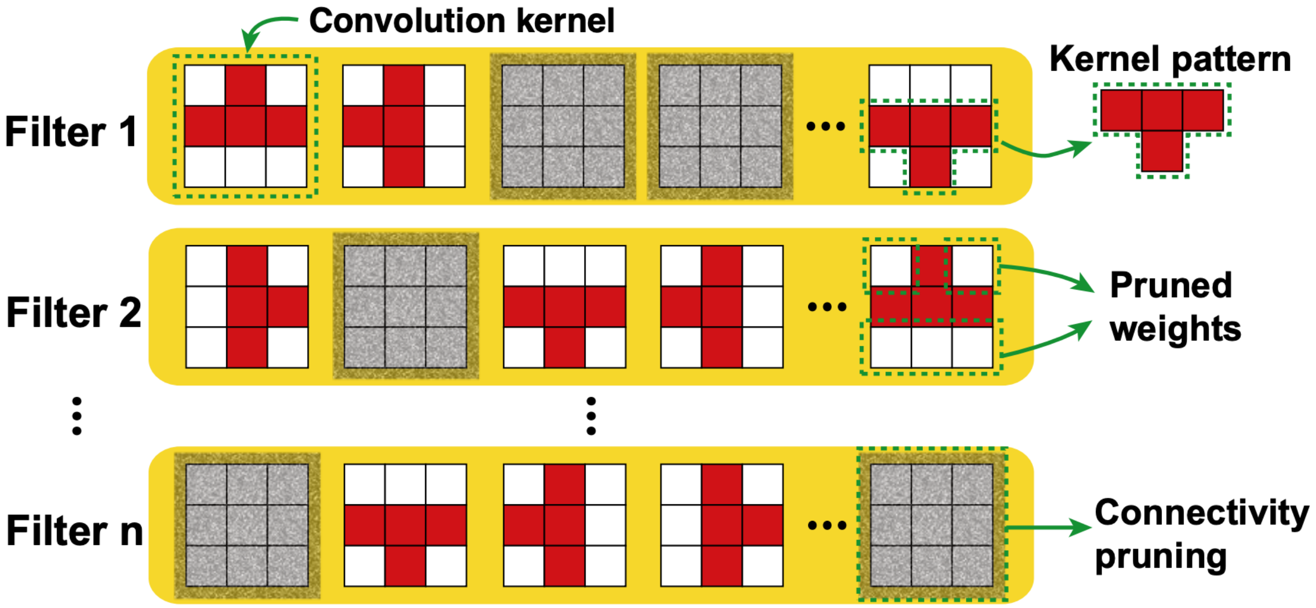
Example: 2-way SIMD



[**Yu**, *ISCA* 2017]

Pattern-Based Weight Pruning

Prune based on pattern (rather than row)



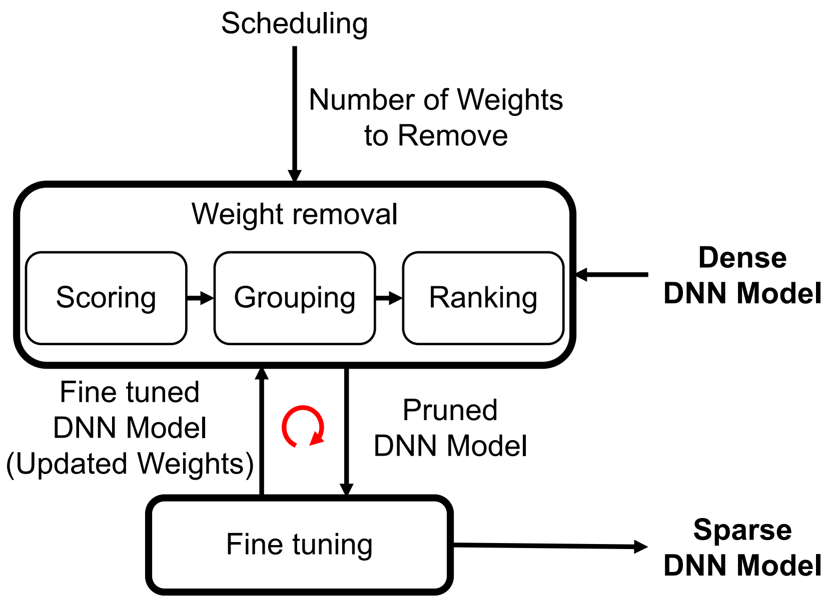
[**PCONV**, *AAAI* 2020], [**PatDNN**, *ASPLOS* 2020]

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.

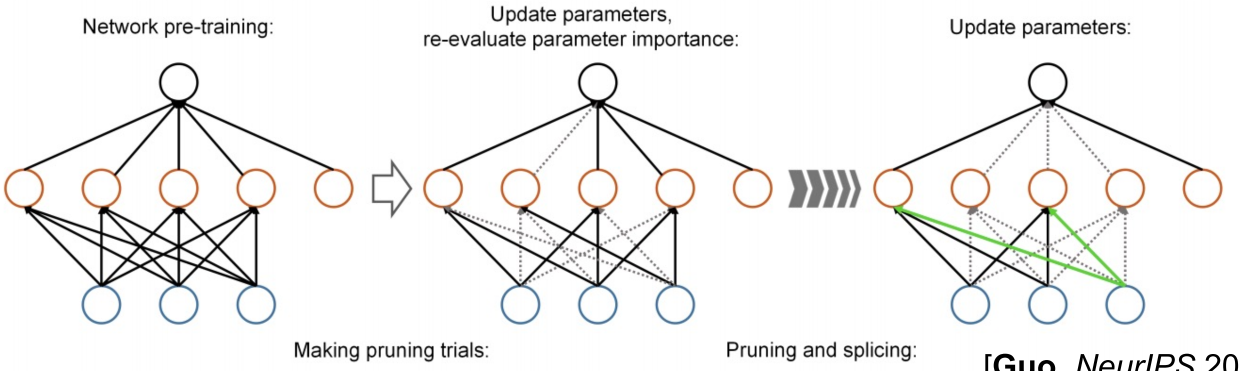
Fine tuning and Scheduling

- **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)



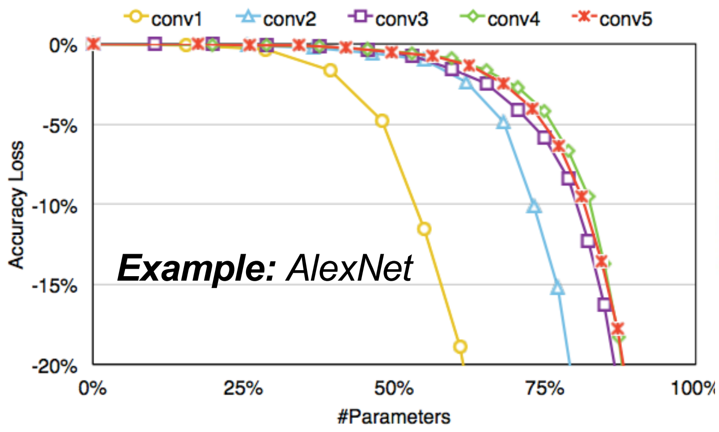
[**Guo**, *NeurIPS* 2016]

Number of non-zero weights reduced by ~2x

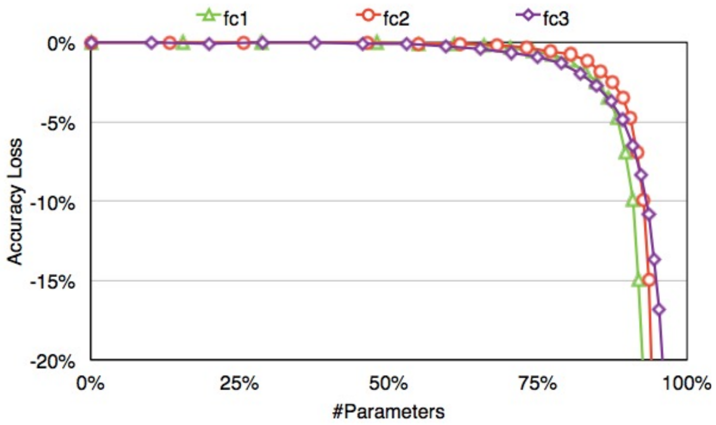
		w/o splicing	w/ splicing
Layer	Params.	Params.% [9]	Params.% (Ours)
conv1	35K	∼ 84%	53.8%
conv2	307K	∼ 38%	40.6%
conv3	885K	∼ 35%	29.0%
conv4	664K	∼ 37%	32.3%
conv5	443K	∼ 37%	32.5%
fc1	38M	∼ 9%	3.7%
fc2	17M	∼ 9%	6.6%
fc3	4M	∼ 25%	4.6%
Total	61M	∼ 11%	**5.7%**

Interplay: Pruning and Layer Types

Convolutional Layers



Fully Connected Layers



For AlexNet

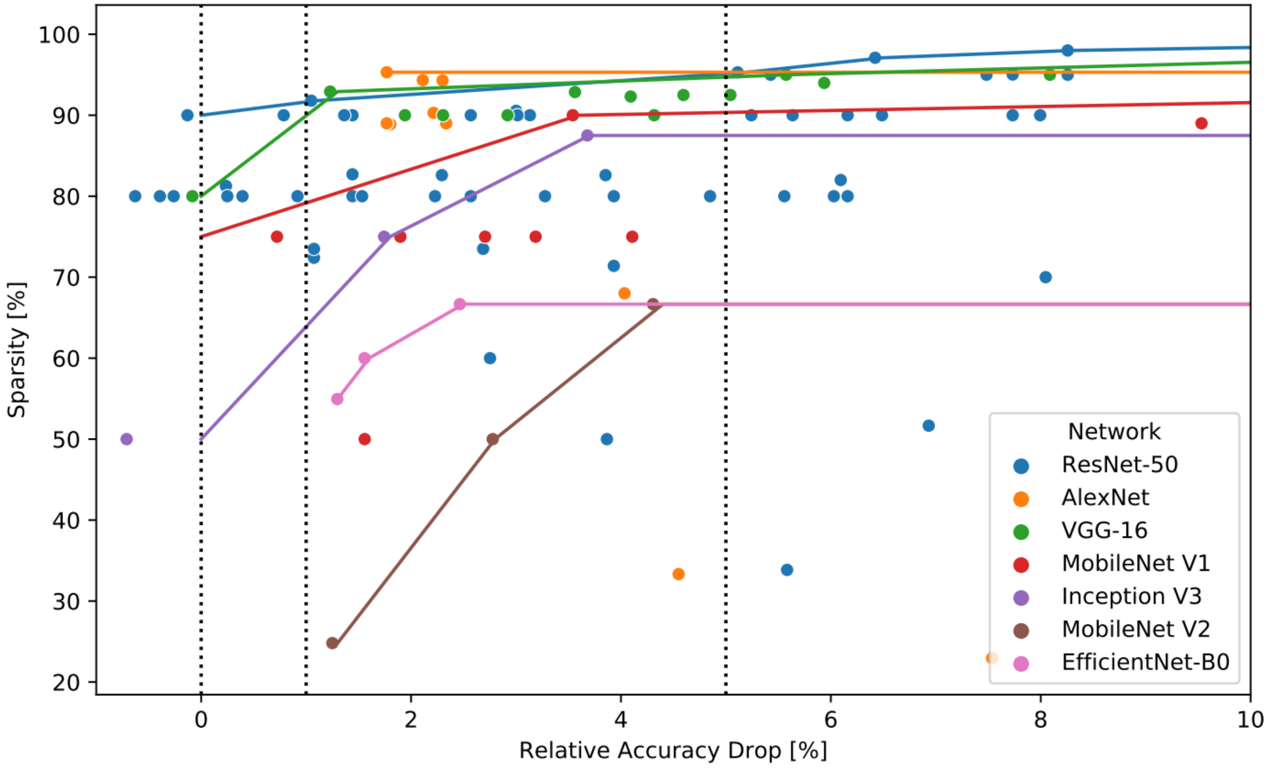
Weight Reduction: *CONV layers 2.7x, FC layers 9.9x*

(Most reduction on fully connected layers)

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

[**Han**, *NeurIPS* 2015]

Interplay: Pruning and Accuracy Loss



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

[**Hoefler**, *JMLR* 2021]

Interplay: Pruning and DNN Model

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

[**Blalock**, *MLSys* 2020]

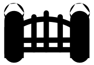
Aspects of Scheduling - Sparsity

Gating:
Explicitly eliminate ineffectual storage accesses and computes by letting the hardware unit staying idle for the cycle to save energy

Format:
Choose tensor representations to save storage space and energy associated with zero accesses

Skipping:
Explicitly eliminate ineffectual storage accesses and computes by skipping the cycle to save energy and time

Aspects of Scheduling - Sparsity

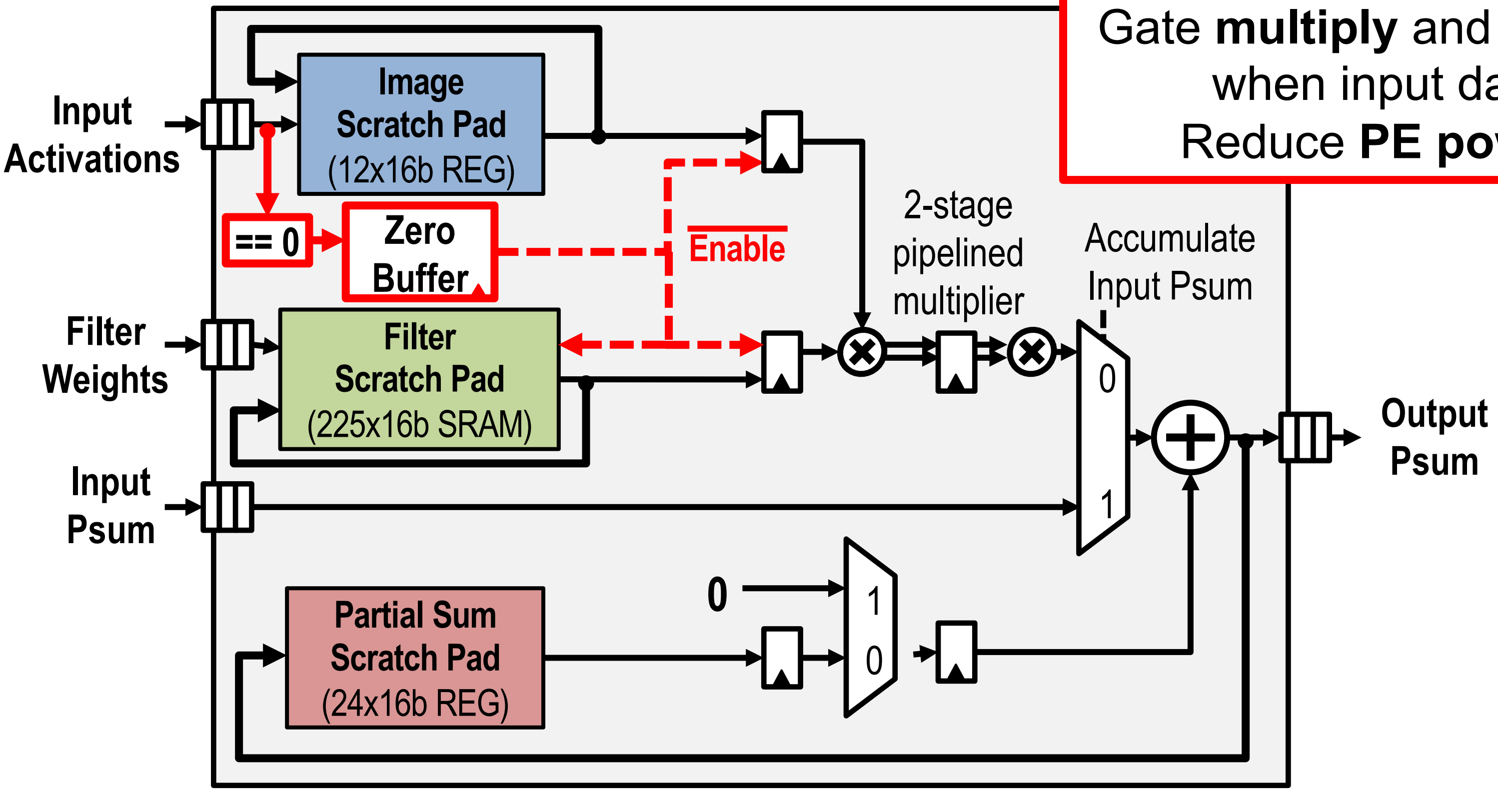


Gating:
Explicitly eliminate ineffectual storage accesses and computes by letting the hardware unit staying idle for the cycle to save energy

Format:
Choose tensor representations to save storage space and energy associated with zero accesses

Skipping:
Explicitly eliminate ineffectual storage accesses and computes by skipping the cycle to save energy and time

Eyeriss – Gating



Gate **multiply** and **memory reads** when input data is zero.
Reduce **PE power** by **45%**

[Chen, *ISSCC* 2016]

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 few lectures
  - 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. Guttag, “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