Deep Learning Optimization - Pruning

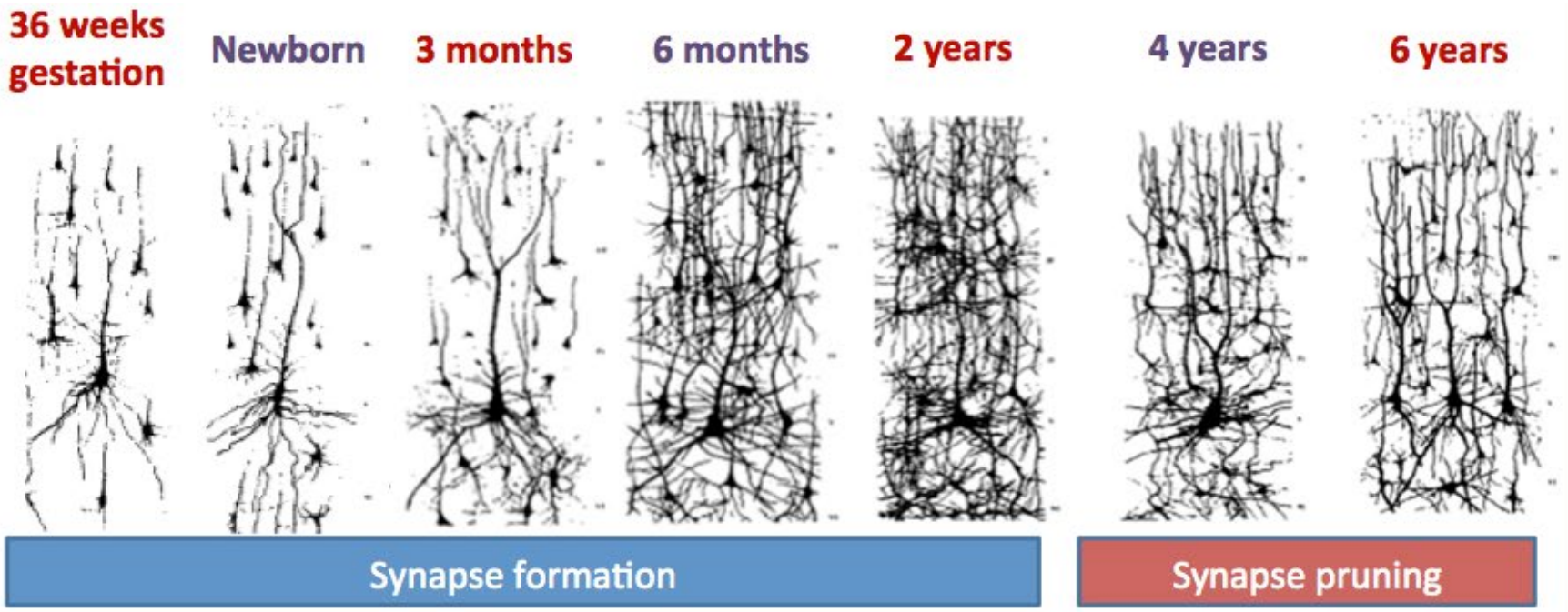
May 8, 2023

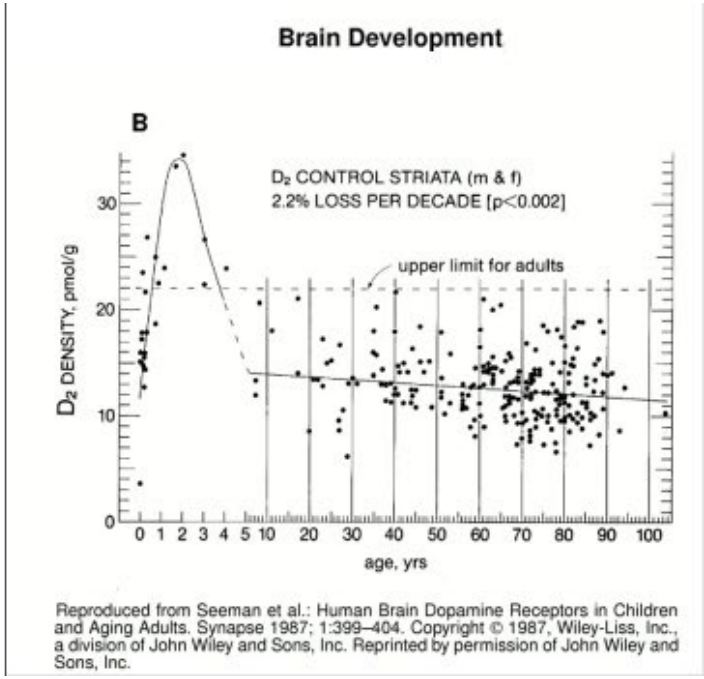
Eunhyeok Park

Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

Neuron Pruning is Natural in Biological System

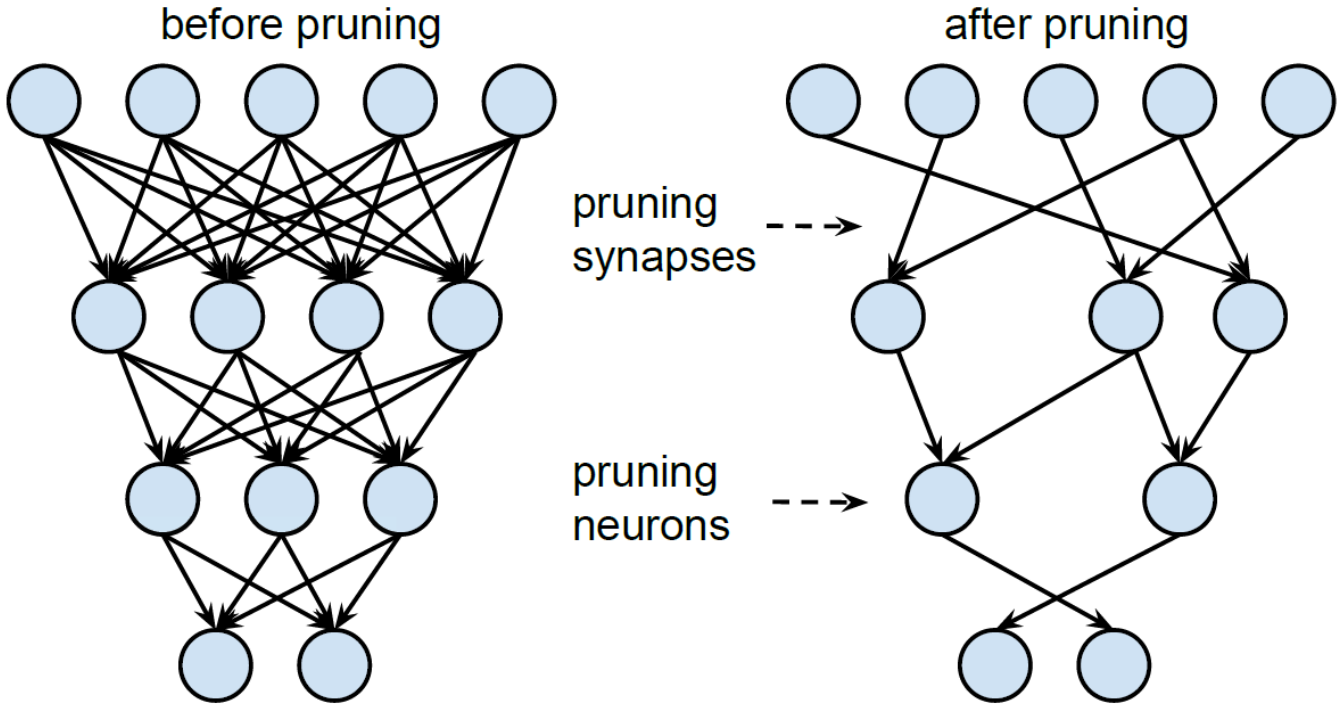
- # synapses increases before 2 years old and, then decreases due to pruning
  - Possibly to reduce resource (e.g., energy) usage

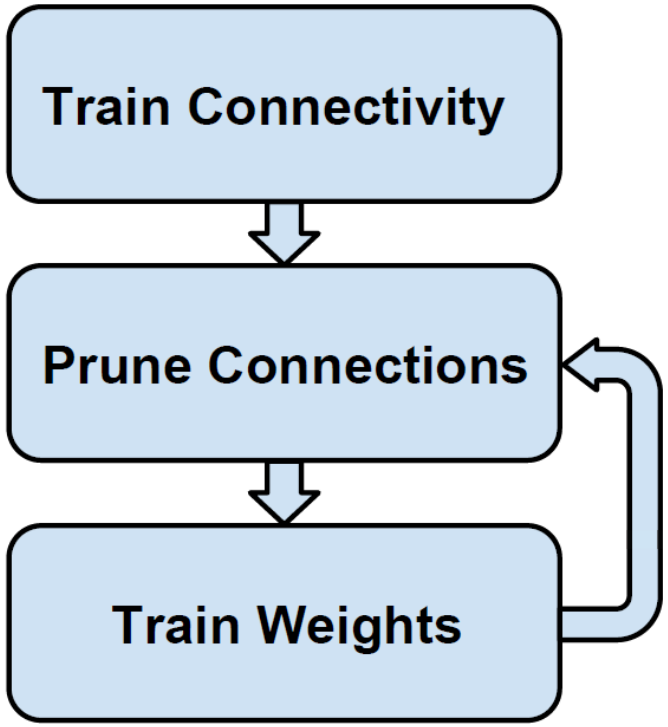




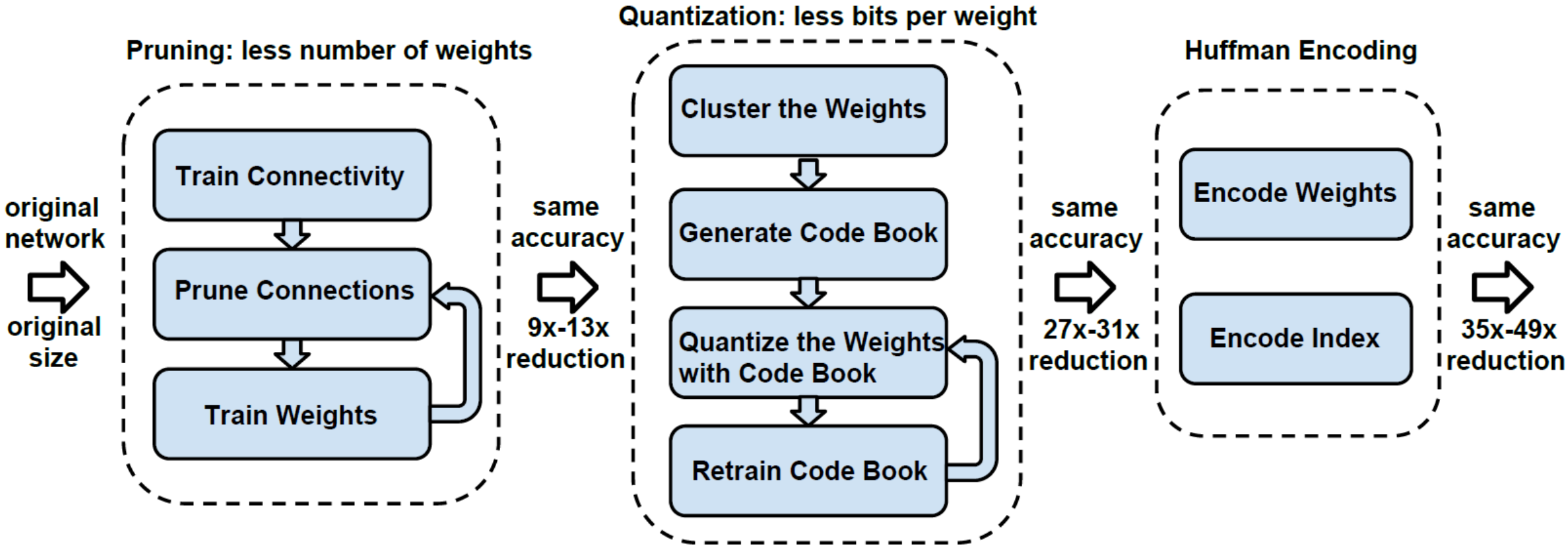
[https://universe-review.ca/R10-16-ANS12.htm]

Pruning CNN: NIPS 2015





Pruning, Quantization and Compression



Goal and Benefits

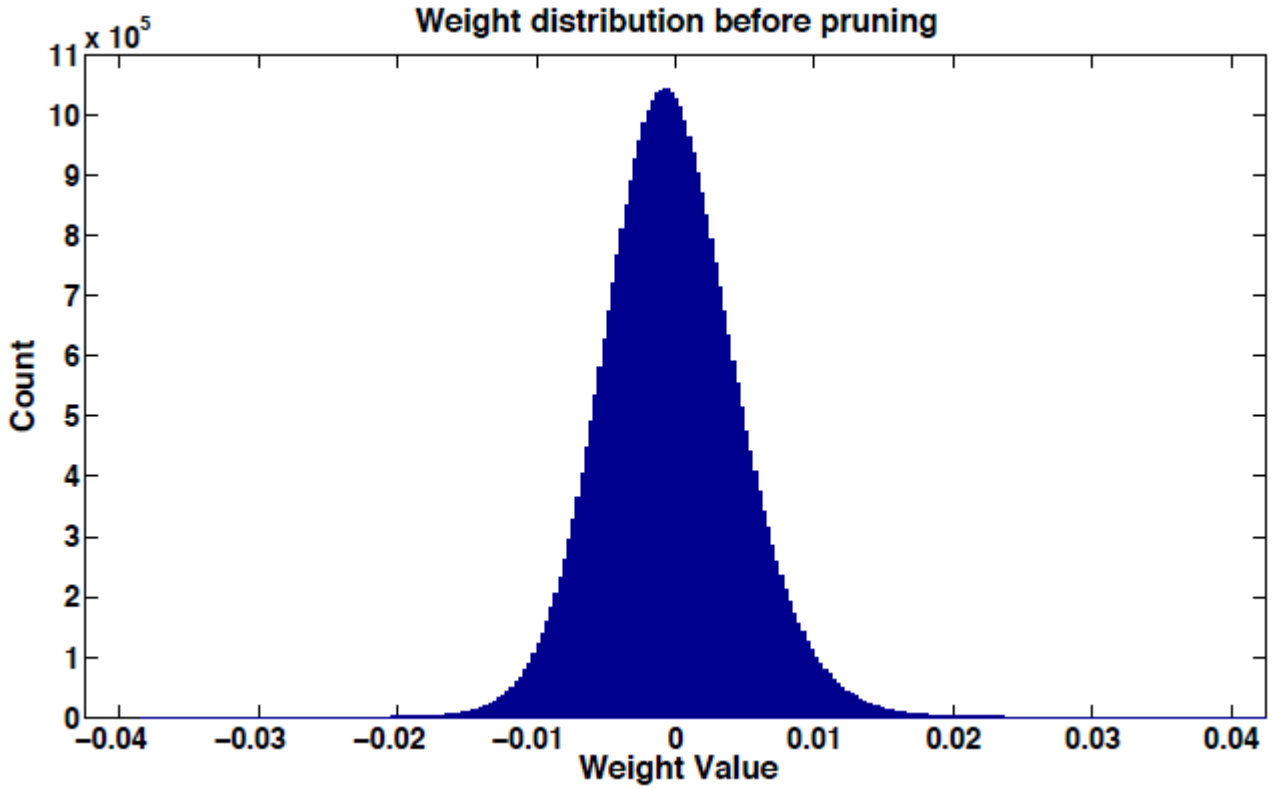
- "Our goal is to reduce the storage and energy required to run inference on such large networks so they can be deployed on mobile devices."

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

- Benefits of small models for mobile devices
- Faster download
- Lower energy consumption
  - Energy consumption is dominated by memory access.
  - Ex) running a 1 billion connection neural network
    - At 20fps would require (20Hz)(1G)(640pJ) = 12.8W just for DRAM access
    - Well beyond the power envelope of a typical mobile device

Illustration of Pruning

- Weight distribution before pruning



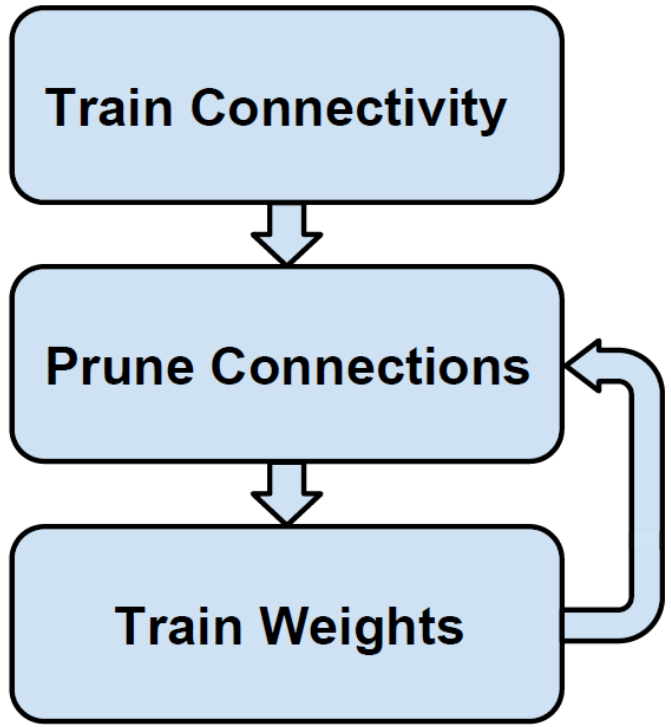
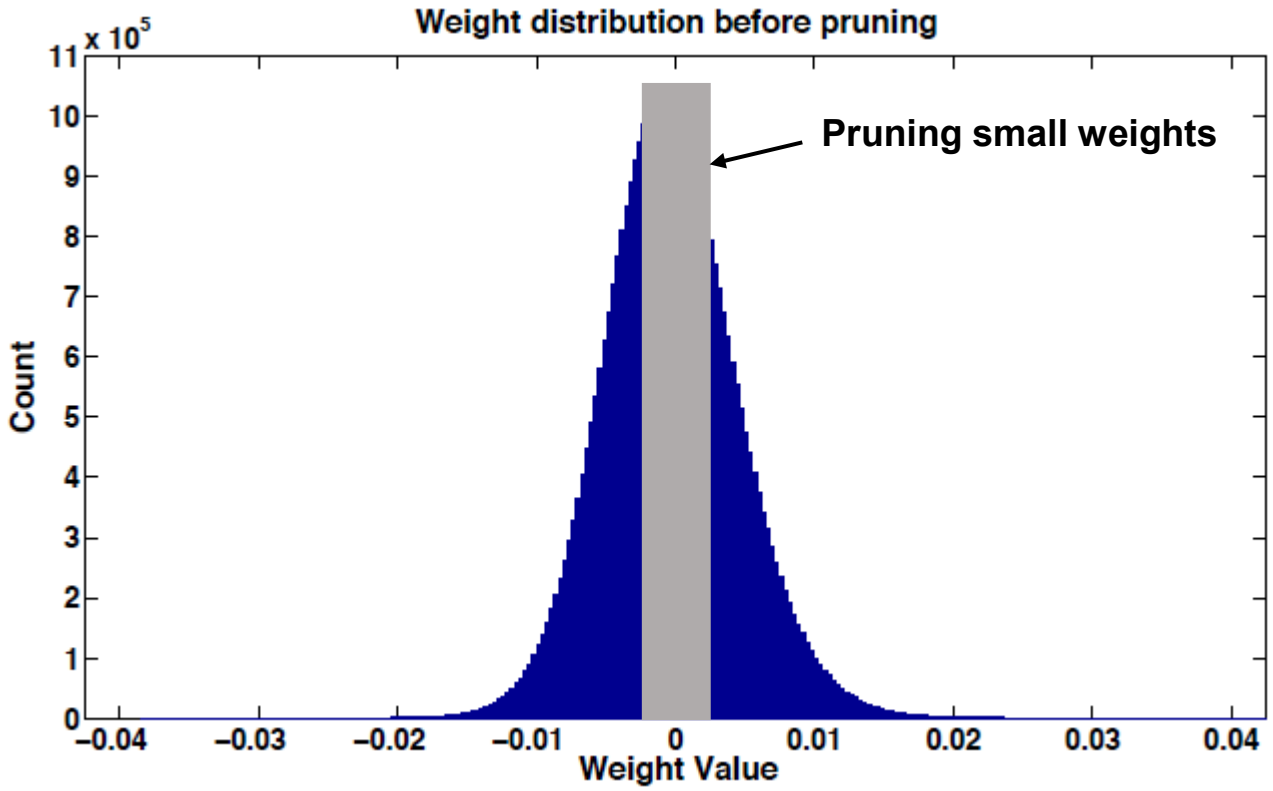


Illustration of Pruning

- Small weights are pruned, i.e., set to zero



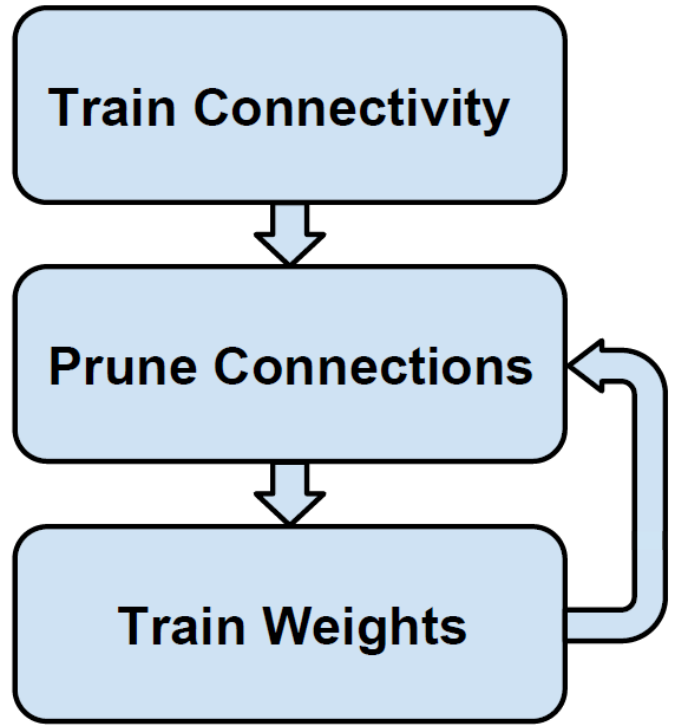
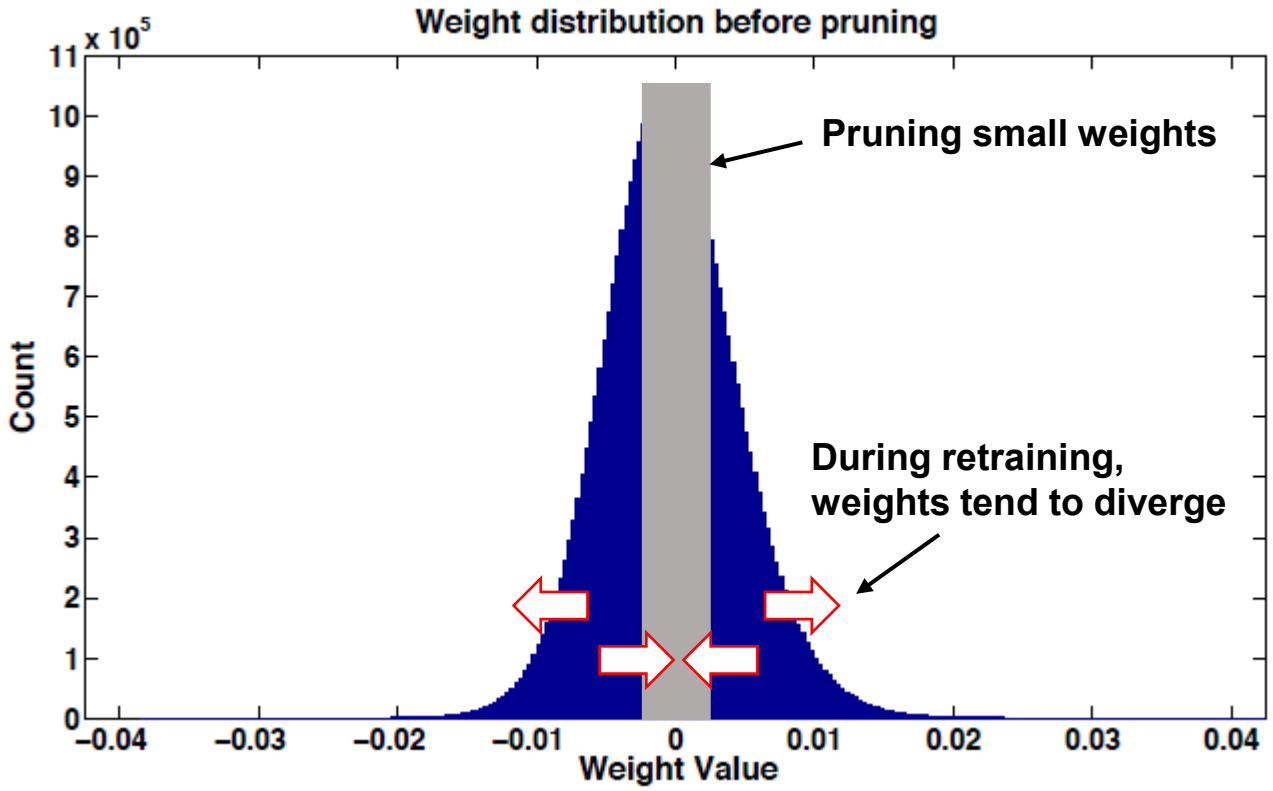


Illustration of Pruning

- During training, weights are re-distributed



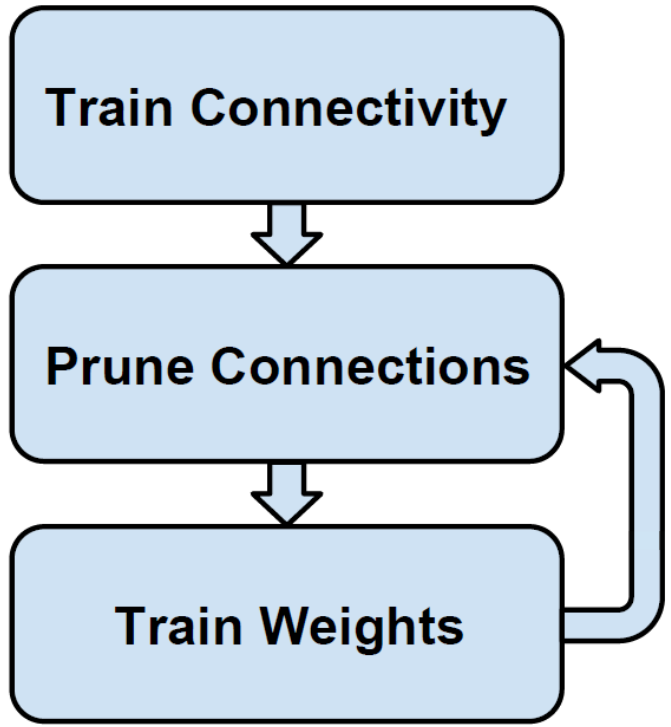
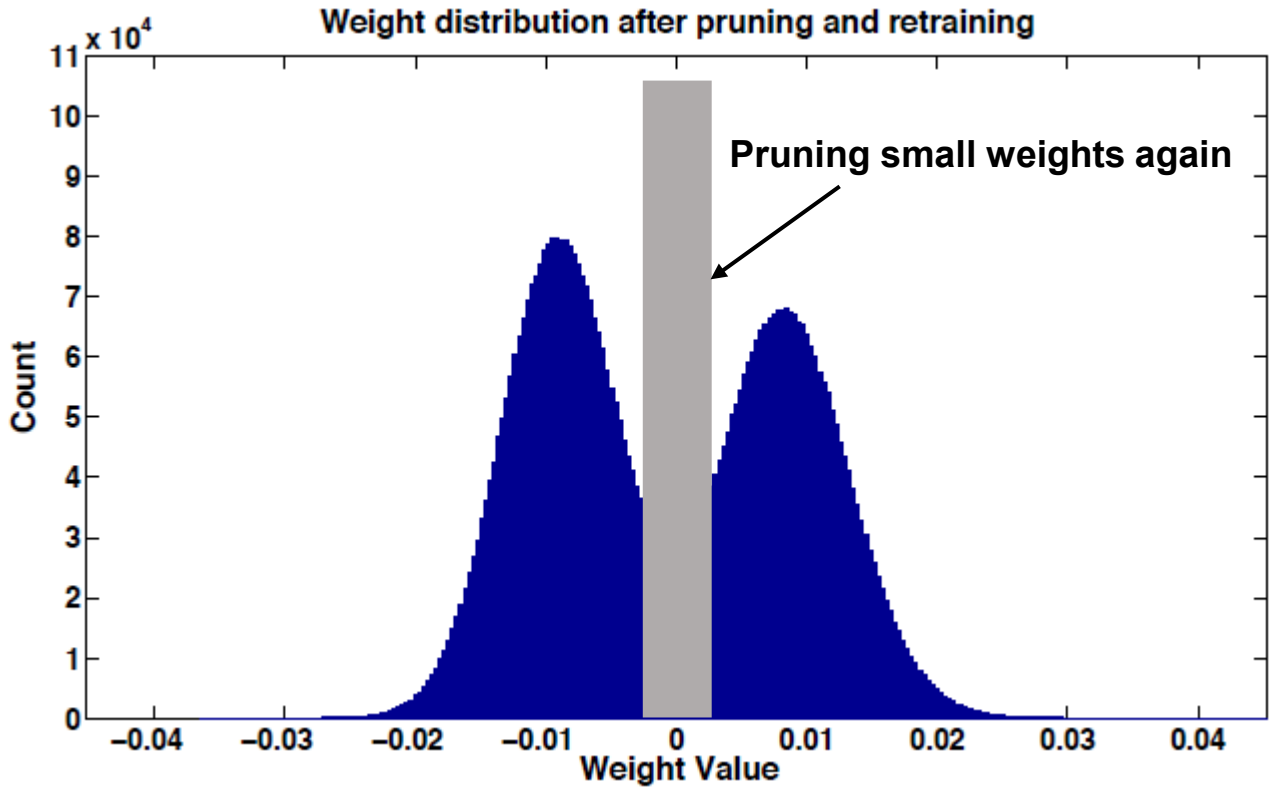
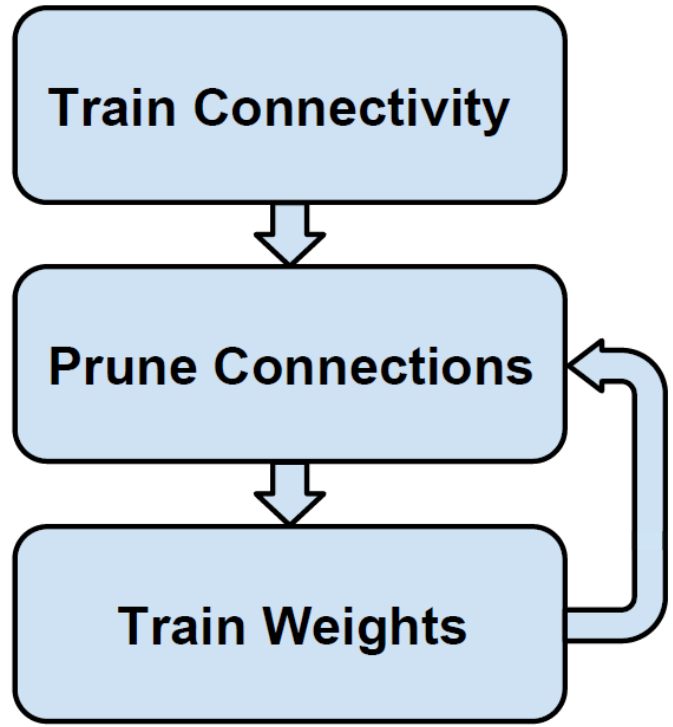


Illustration of Pruning

- After a step of training, new small weights are obtained



10X smaller scale



Compressed Sparse Row

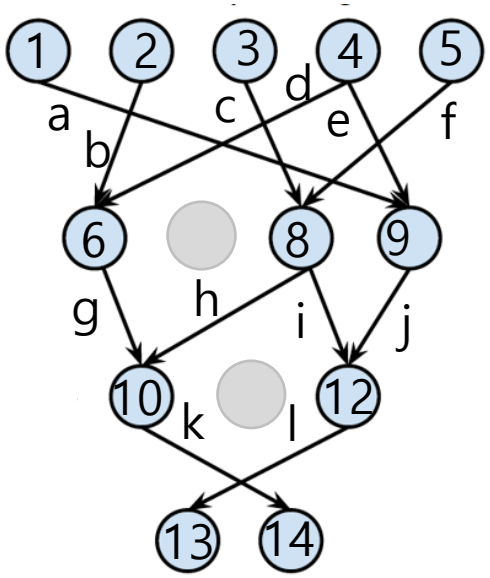
- Store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format
- Example of CSR format
  - The value array contains the value of the non-zero data
  - The column index contains the column index of the non-zero data
  - The row index the accumulated number of non-zero data at the i-th row
    - The number of non-zero data at the i-th row is $a[i+1] - a[i]$

  - Row index [0 0 2 3 4]
  - Column index [0 1 2 1]
  - Value array [5 8 3 6]

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

Compressed Sparse Row

- Store the sparse structure that results from pruning using compressed sparse row (CSR) or compressed sparse column (CSC) format

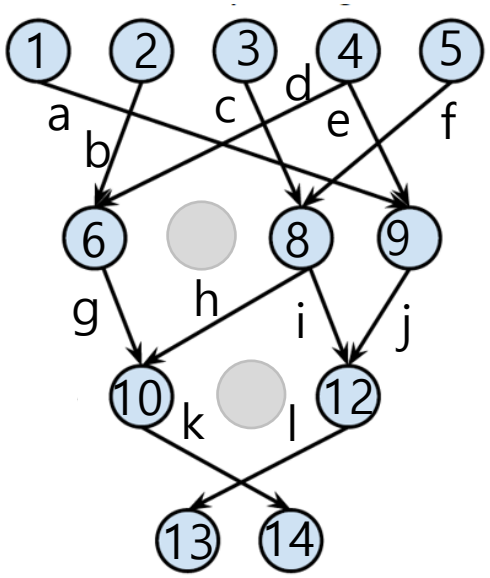


CSR-like example

Non-zero weight indices	4 5 11 13 16 19 / 1 7 9 12 / 2 5
Weight values	a b c d e f g h i j k l

Storing Weight Index Difference

- To compress further, store the index difference instead of the absolute position

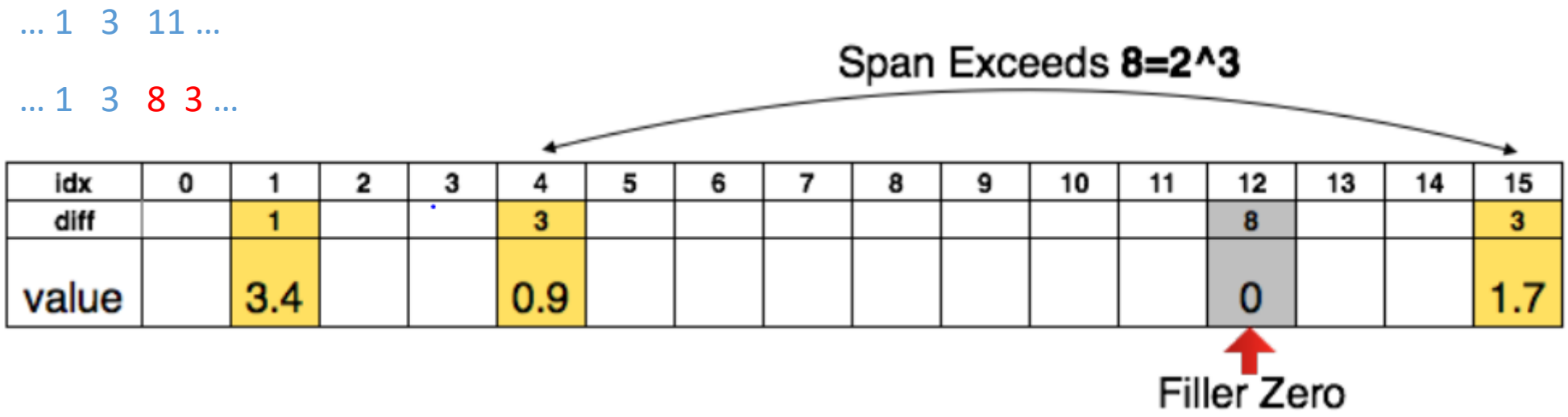


CSR-like example

Non-zero weight indices	4 5 11 13 16 19 / 1 7 9 12 / 2 5	
	4 1 6 2 3 3 / **1** 6 2 3 / **2** 3	3 bit case
Weight values	a b c d e f g h i j k l	

Storing Weight Index Difference

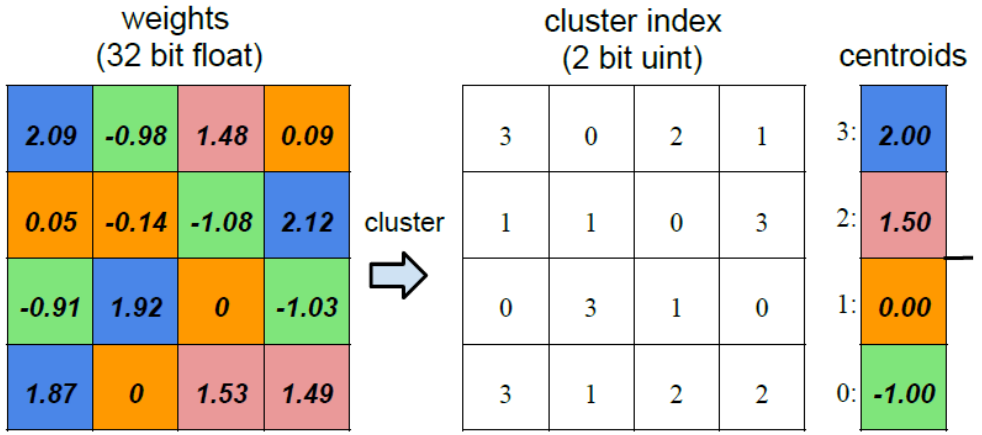
- To compress further, store the index difference instead of the absolute position
- When we need an index difference larger than the bound, use the zero padding
  - In case when the difference exceeds 8, the largest 3-bit (as an example) unsigned number, add a filler zero.



Clustering-based Quantization

- Limit the number of effective weights by having multiple connections share the same weight, and then fine-tune those shared weights

- The weights are quantized to 4 bins (denoted with 4 colors), all the weights in the same bin share the same value, thus for each weight, we then need to store only a small index into a table of shared weights.

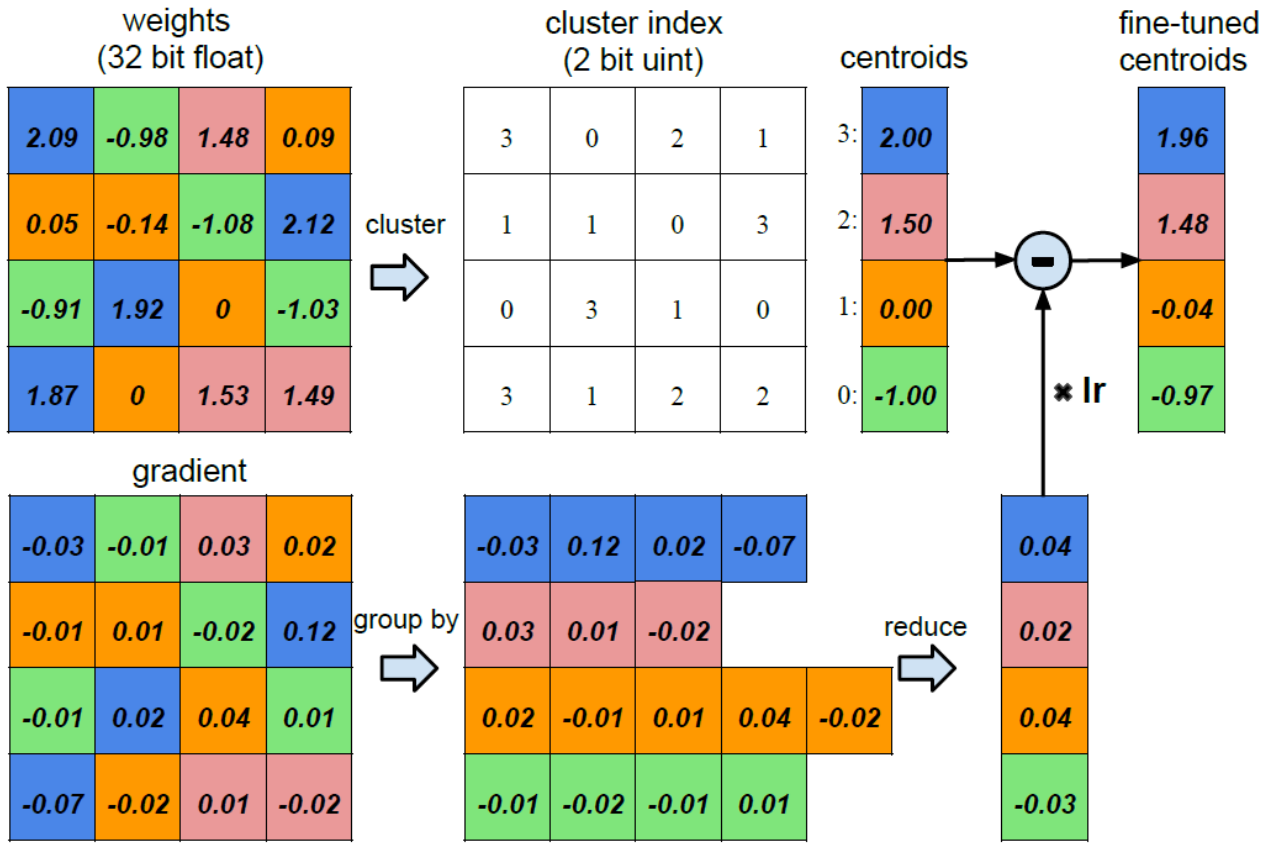


Fine-tuning for the Clustered Data

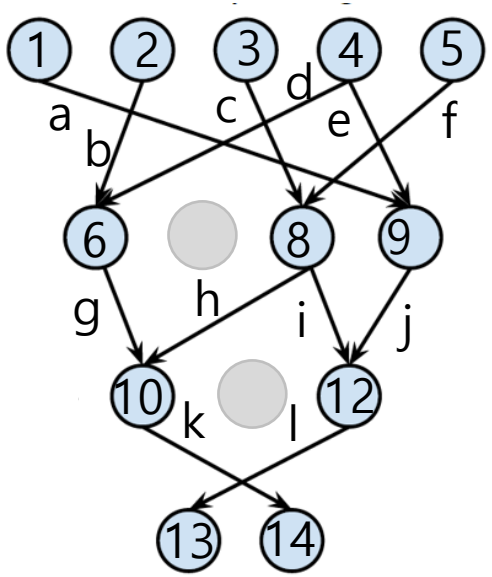
- Limit the number of effective weights by having multiple connections share the same weight, and then fine-tune those shared weights

- During update, all the gradients are grouped by the color and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration.

- For pruned AlexNet, 8-bits (256 shared weights) for each CONV layers, and 5-bits (32 shared weights) for each FC layer without any loss of accuracy

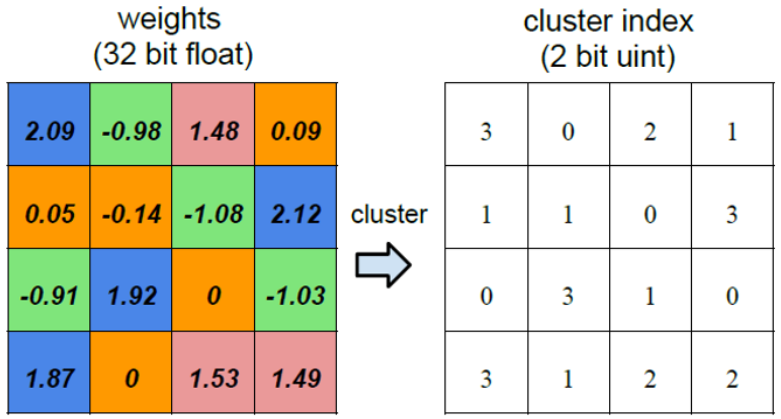


Replacing Weight Values with Bin Indices



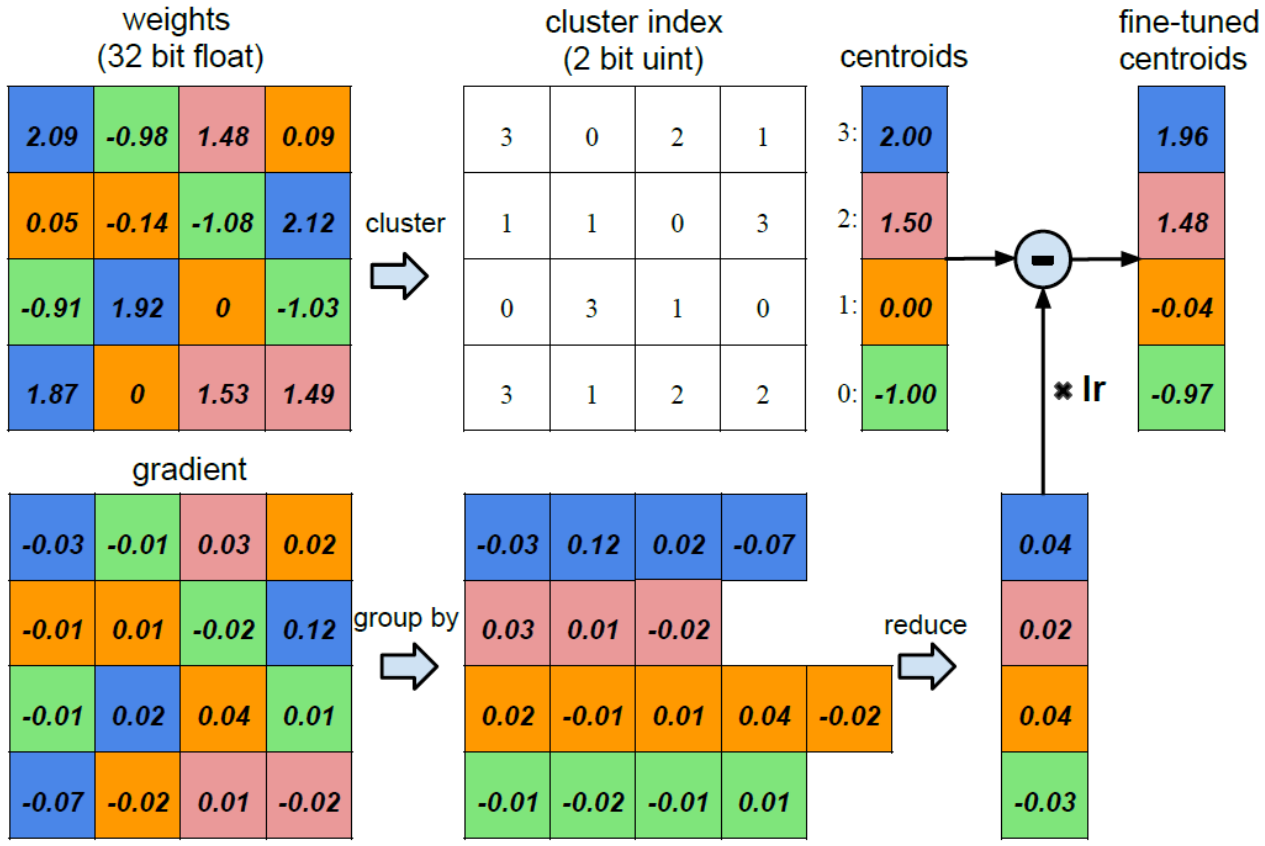
CSC-like example

Non-zero weight indices	4 5 11 13 16 19 / 1 7 9 12 / 2 5	
	4 1 6 2 3 3 / 1 6 2 3 / 2 3	4 bits
Weight values	a b c d e f g h i j k l	
Weight value bin indices	3 0 2 1 1 0 3 0 3 1 0 3	8 bits (CONV) 5 bits (FC)



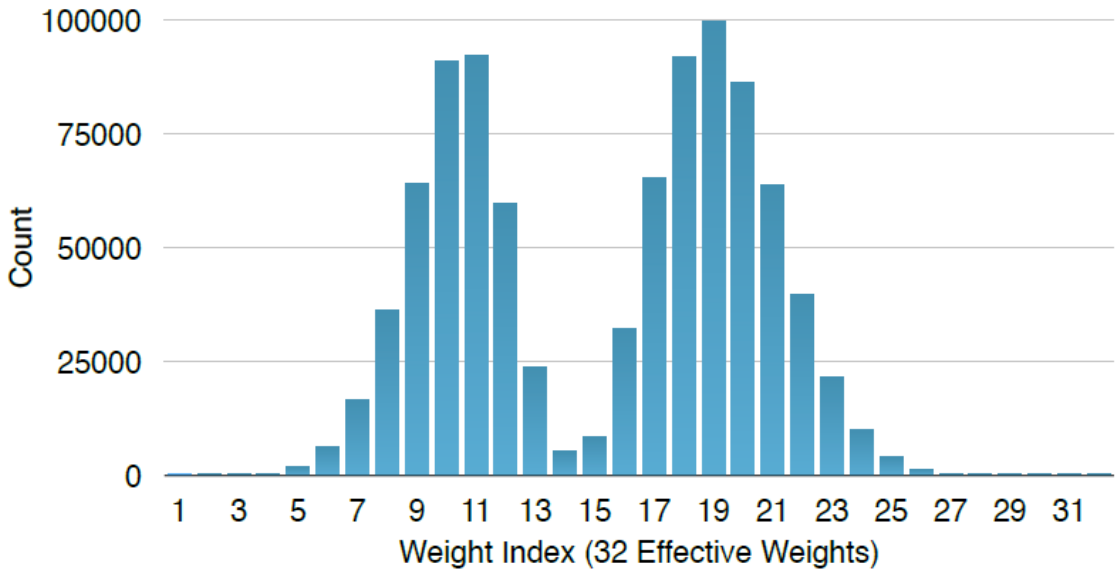
Compression ratio?

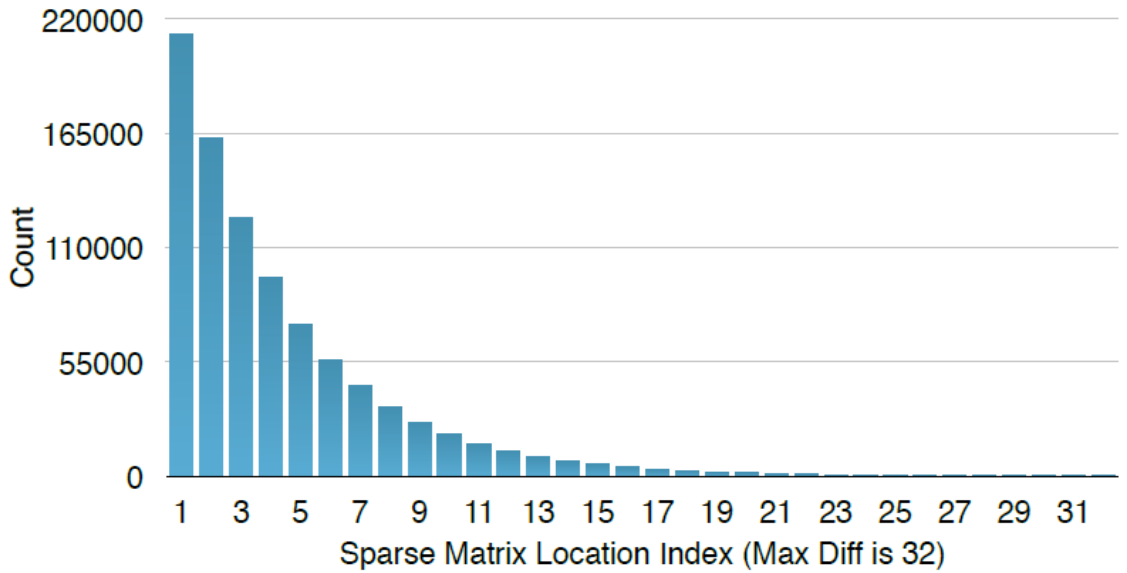
- Ex) Originally need to store 16 weights each has 32 bits
  - 16 * 32 b = 512 b
- Now need to store only 4 effective weights (blue, green, red and orange), each has 32 bits, together with 16 2-bit indices
  - 4 * 32b + 2b *16 = 160 b
- Compression ratio = x 3.2



Huffman Coding

- Applied to **the indices of weight value tables** in CSC format
- Huffman coding allows frequent indices to be represented with less # bits





Accuracy, Compression Ratio and # Bits (Levels)

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40**×
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39**×
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35**×
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49**×

AlexNet

#CONV bits / #FC bits	Top-1 Error	Top-5 Error	Top-1 Error Increase	Top-5 Error Increase
32bits / 32bits	42.78%	19.73%	-	-
8 bits / 5 bits	42.78%	19.70%	0.00%	-0.03%
8 bits / 4 bits	42.79%	19.73%	0.01%	0.00%
4 bits / 2 bits	44.77%	22.33%	1.99%	2.60%

AlexNet

Non-zero weight indices: 4 5 11 13 16 19 / 1 7 9 12 / 2 5

4 1 6 2 3 3 / 1 6 2 3 / 2 3 — 4 bits

Weight values: a b c d e f g h i j k l

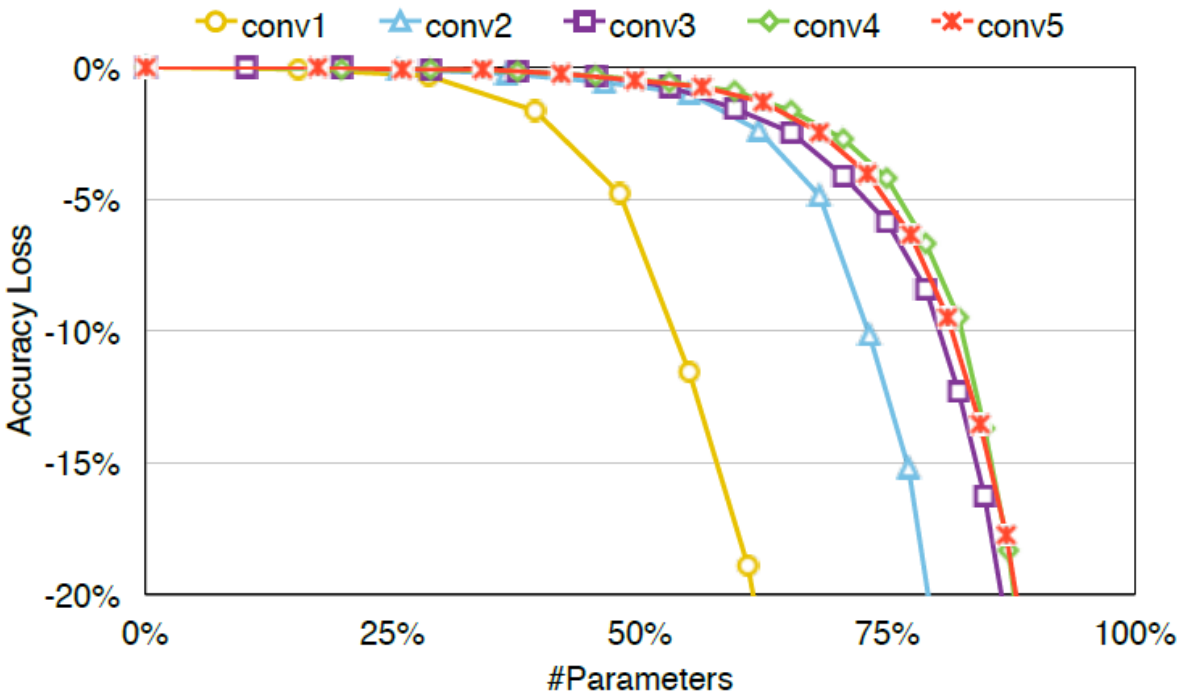
Weight value bin indices: 3 0 2 1 1 0 3 0 3 1 0 3 — 8 bits (CONV), 5 bits (FC)

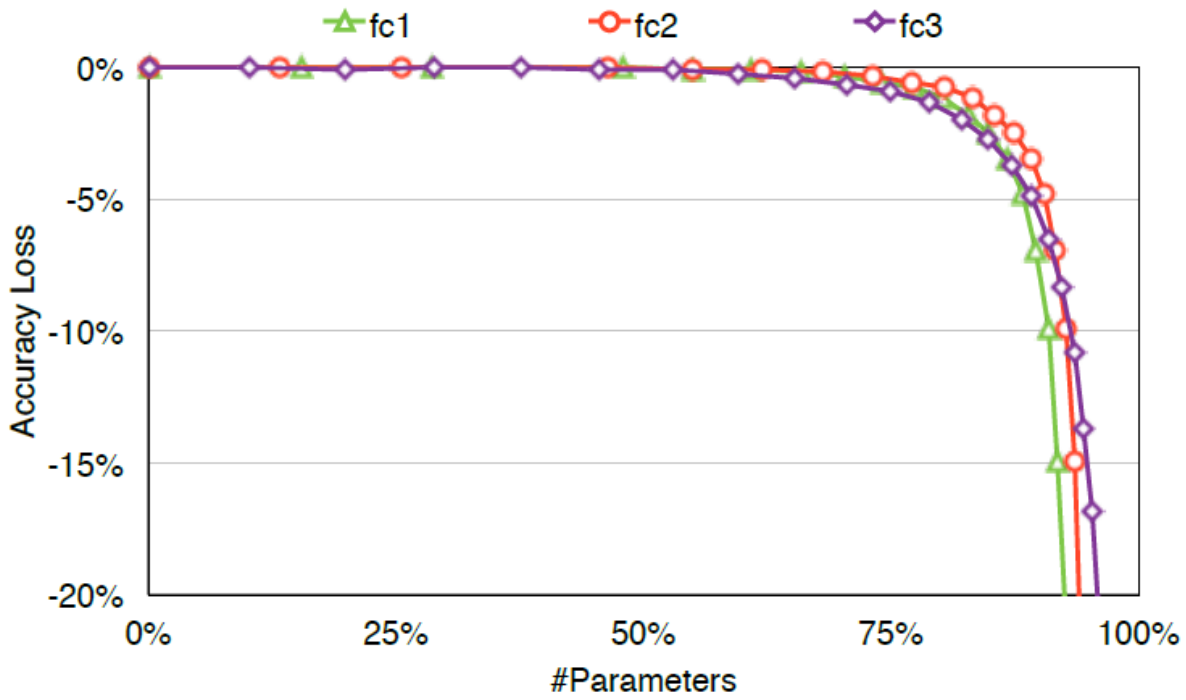
- 8b/5b case: 256 levels for CONV and 32 levels for FC layers
- Finally, weights can be represented by ~2.5 bits (CONV) and ~3.5 bits (FC)

Layer	#Weights	Weights% (P)	Weight bits (P+Q)	Weight bits (P+Q+H)	Index bits (P+Q)	Index bits (P+Q+H)	Compress rate (P+Q)	Compress rate (P+Q+H)
conv1	35K	84%	8	6.3	4	1.2	32.6%	20.53%
conv2	307K	38%	8	5.5	4	2.3	14.5%	9.43%
conv3	885K	35%	8	5.1	4	2.6	13.1%	8.44%
conv4	663K	37%	8	5.2	4	2.5	14.1%	9.11%
conv5	442K	37%	8	5.6	4	2.5	14.0%	9.43%
fc6	38M	9%	5	3.9	4	3.2	3.0%	2.39%
fc7	17M	9%	5	3.6	4	3.7	3.0%	2.46%
fc8	4M	25%	5	4	4	3.2	7.3%	5.85%
Total	61M	11%(9×)	5.4	4	4	3.2	3.7% (**27**×)	2.88% (**35**×)

Per-Layer Sensitivity

- CONV1 (near the input) is the most sensitive
- FC layers (near the output) is much less sensitive





DSD: Dense-Sparse-Dense Training for Deep Neural Networks

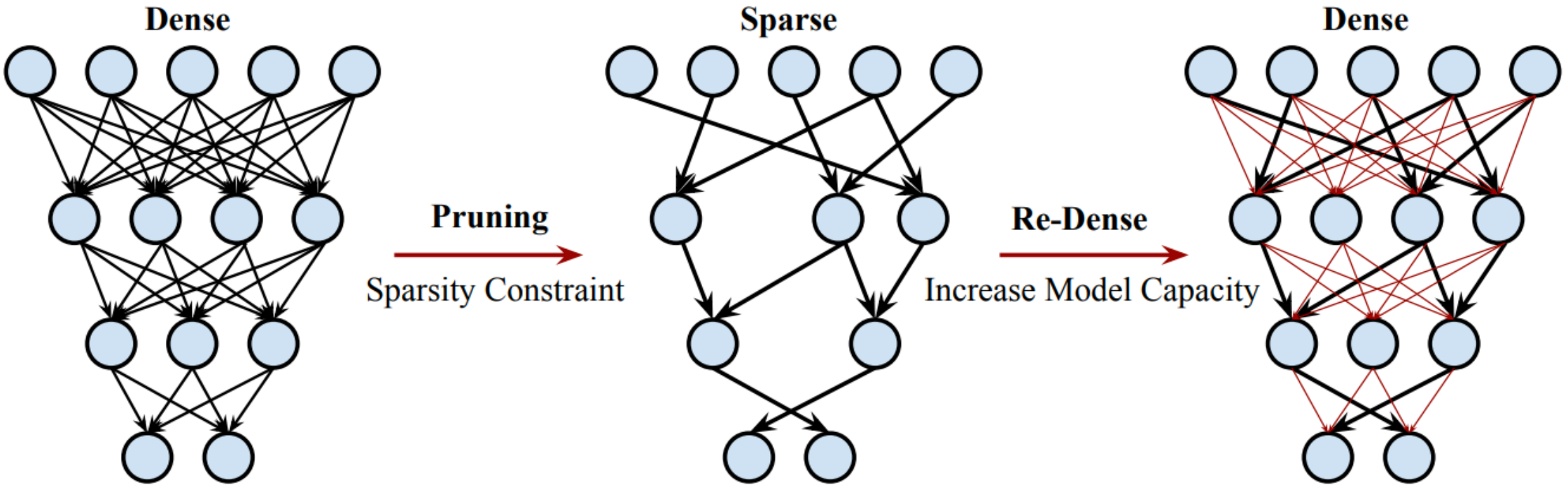
Model Size & Accuracy

- The upside of complicated models
  - Expressive and can capture the highly non-linear relationship between features and output

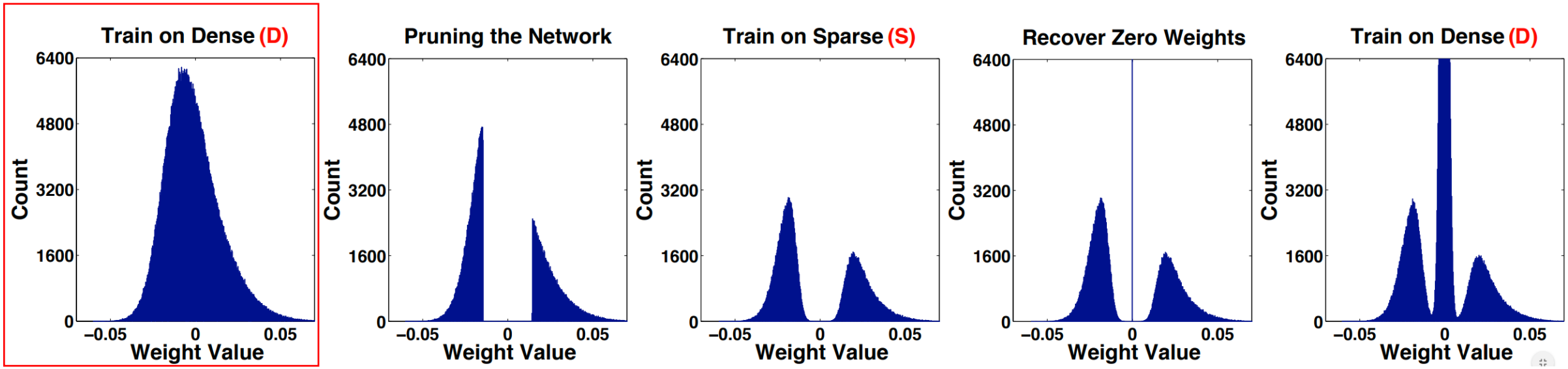
- The downside of complicated models
  - Prone to capturing the noise, rather than the intended pattern, in the training dataset
  - This noise does not generalize to new datasets, leading to over-fitting and a high variance

Dense-Sparse-Dense Training

- Start from a dense model from conventional training
- Regularizes the model with sparsity constrained optimization
- Increase the model capacity by restoring and retraining the pruned weights

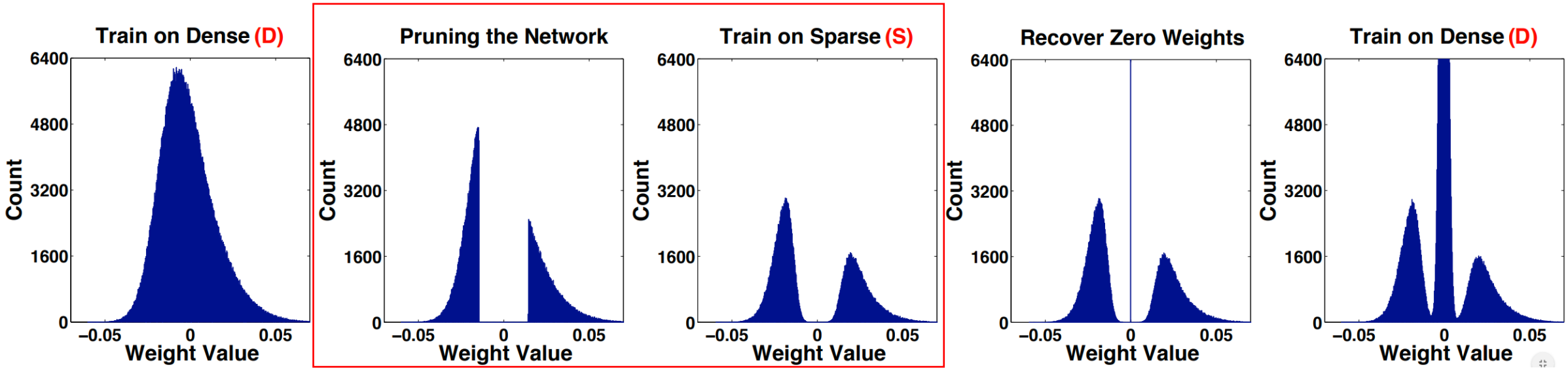


DSD Training



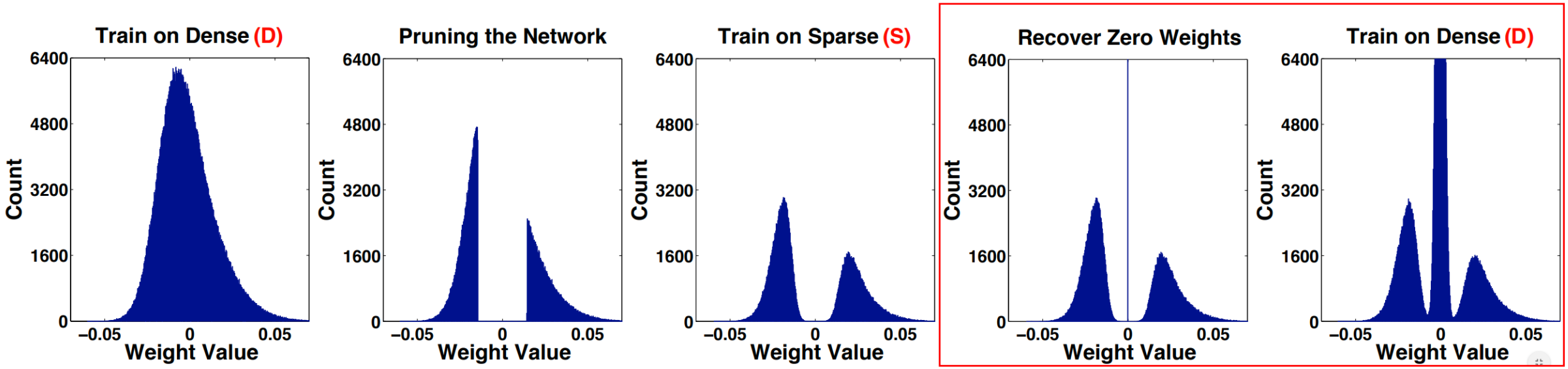
- **Initial Dense Training**
  - learn the connection weights and importance via normal network training on the dense network
  - Learn the values of the weights and their importance (absolute value)

DSD Training



- **Sparse Training**
  - Pick the k-th largest one λ as the threshold where $k = N * (1 - sparsity)$
    - Sparsity is between 25 % to 50 %
  - Generate a binary mask to remove all the weights smaller than λ
  - Retraining while enforcing the binary mask in each iteration

DSD Training



- **Final Dense Training**
  - The final D step recovers the pruned connections, making the network dense again
  - These previously-pruned connections are initialized to zero and the entire network is retrained with 1/10 the original learning rate (since the sparse network is already at a good local minima)

Results

Neural Network	Domain	Dataset	Type	Baseline	DSD	Abs. Imp.	Rel. Imp.
GoogLeNet	Vision	ImageNet	CNN	31.1%[1]	**30.0%**	1.1%	3.6%
VGG-16	Vision	ImageNet	CNN	31.5%[1]	**27.2%**	4.3%	13.7%
ResNet-18	Vision	ImageNet	CNN	30.4%[1]	**29.2%**	1.2%	4.1%
ResNet-50	Vision	ImageNet	CNN	24.0%[1]	**22.9%**	1.1%	4.6%
NeuralTalk	Caption	Flickr-8K	LSTM	16.8[2]	**18.5**	1.7	10.1%
DeepSpeech	Speech	WSJ'93	RNN	33.6%[3]	**31.6%**	2.0%	5.8%
DeepSpeech-2	Speech	WSJ'93	RNN	14.5% [3]	**13.4%**	1.1%	7.4%

[1] Top-1 error. VGG/GoogLeNet baselines from Caffe model zoo, ResNet from Facebook.
[2] BLEU score baseline from Neural Talk model zoo, higher the better.
[3] Word error rate: DeepSpeech2 is trained with a portion of Baidu internal dataset with only max decoding to show the effect of DNN improvement.

Results

✗ **Baseline**: a boy in a red shirt is climbing a rock wall.

✗ **Sparse**: a young girl is jumping off a tree.

✓ **DSD**: a young girl in a pink shirt is swinging on a swing.

○ **Baseline**: a basketball player in a red uniform is playing with a ball.

○ **Sparse**: a basketball player in a blue uniform is jumping over the goal.

✓ **DSD**: a basketball player in a white uniform is trying to make a shot.

✓ **Baseline**: two dogs are playing together in a field.

✓ **Sparse**: two dogs are playing in a field.

✓✓ **DSD**: two dogs are playing in the grass.

✗ **Baseline**: a man and a woman are sitting on a bench.

○ **Sparse**: a man is sitting on a bench with his hands in the air.

○ **DSD**: a man is sitting on a bench with his arms folded.

✗ **Baseline**: a person in a red jacket is riding a bike through the woods.

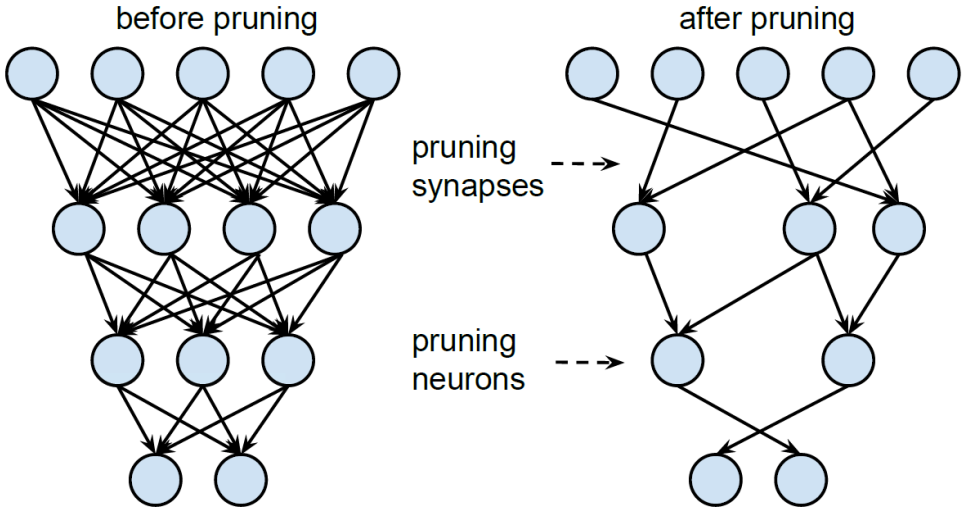
✓✓ **Sparse**: a car drives through a mud puddle.

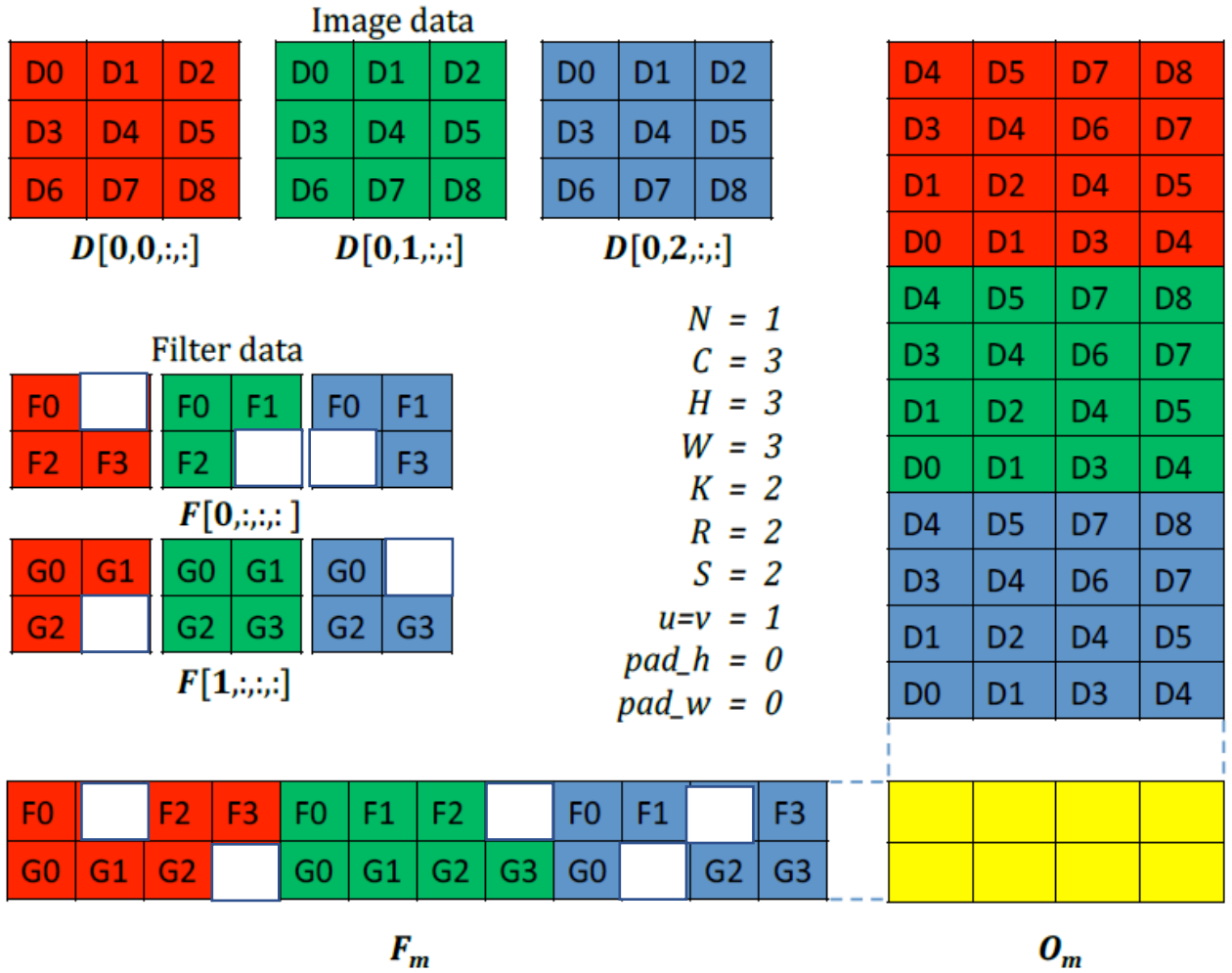
DSD: a car drives ✓ through a forest.

Fast ConvNets
Using Group-wise Brain Damage

[Han, 2015][Chetlur, 2014]

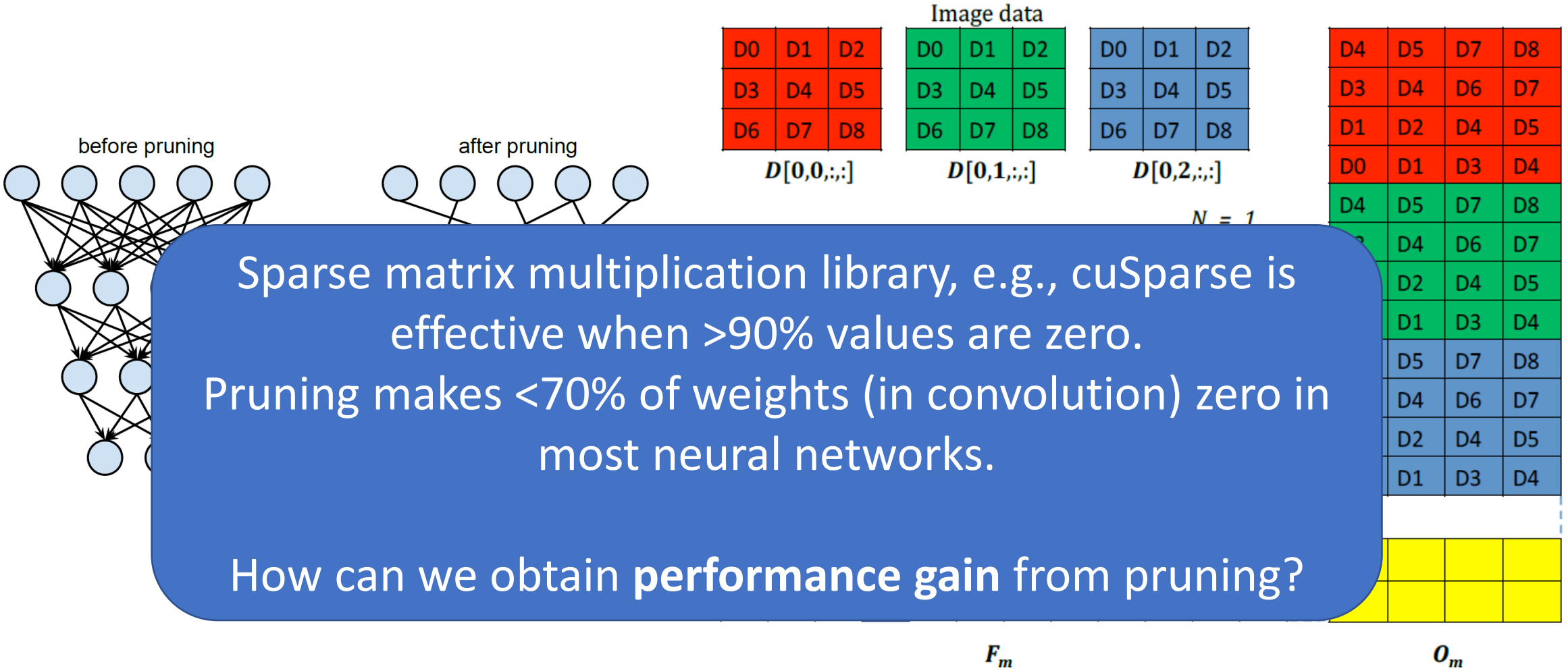
Pruning Hardly Reduces the Runtime of Convolution on GPU





[Han, 2015][Chetlur, 2014]

Pruning Hardly Reduces the Runtime of Convolution on GPU



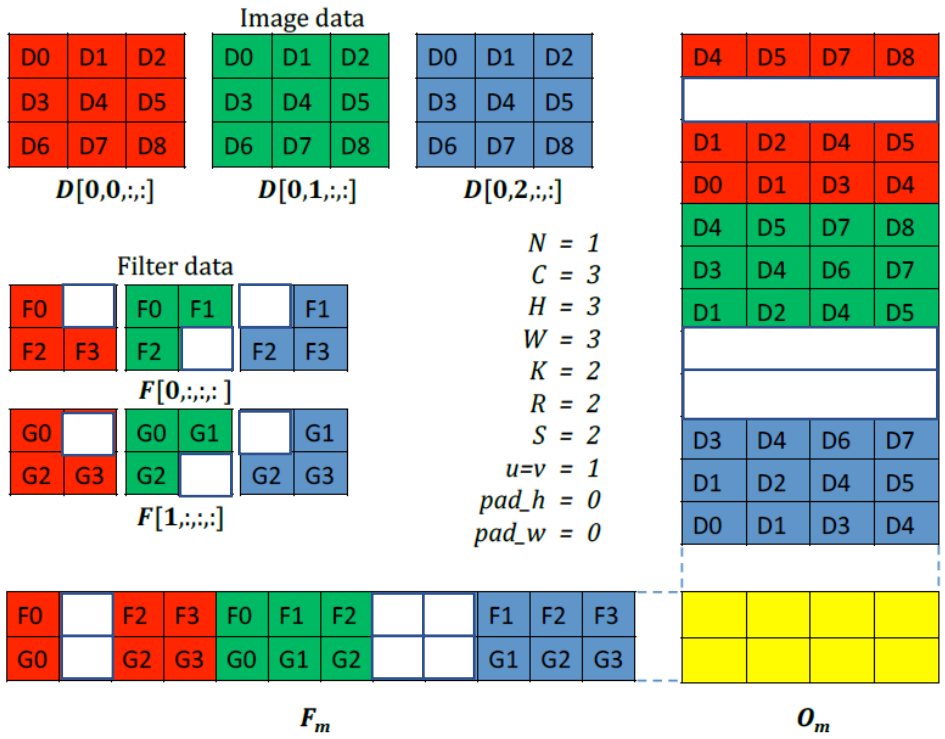
Sparse matrix multiplication library, e.g., cuSparse is effective when >90% values are zero.
Pruning makes <70% of weights (in convolution) zero in most neural networks.

How can we obtain **performance gain** from pruning?

[Lebedev, 2016]

Overview of Group-wise Brain Damage

- For each input feature map, the same location of 2D filter elements is pruned
- Pruning is performed in an incremental manner
  - Repeat pruning a column and retraining as far as accuracy constraint is met
- Convolutions can be reduced to multiplications of thinned dense matrices, which leads to speedup
  - 3X reduction in # multiplications for AlexNet



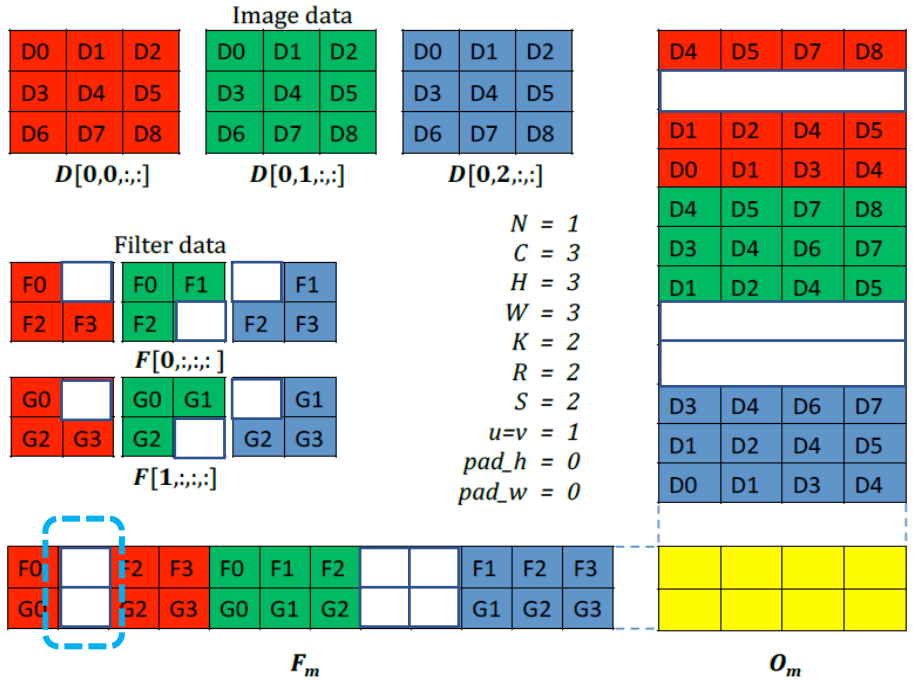
[Lebedev, 2016]

Regularizer for Structured Pruning

- Sum of squared weights in a column of weight matrix

i,j: spatial dimension
s: input feature map
t: output feature map

$$\Omega_{2,1}(K) = \lambda \sum_{i,j,s} \|\Gamma_{ijs}\| = \lambda \sum_{i,j,s} \sqrt{\sum_{t=1}^{T} K(i,j,s,t)^2}$$



[Bishop, 2006]

Regularization to Reduce the Magnitude of Weights

- Observation: Coefficients of overfitting function tend to have large magnitude
- Regularization
  - Add an additional term to loss function
  - A suitable adjustment of λ is needed

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n,\mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

derivative

$$v_{i+1} := 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$$

$$w_{i+1} := w_i + v_{i+1}$$

AlexNet case

	$M=0$	$M=1$	$M=6$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

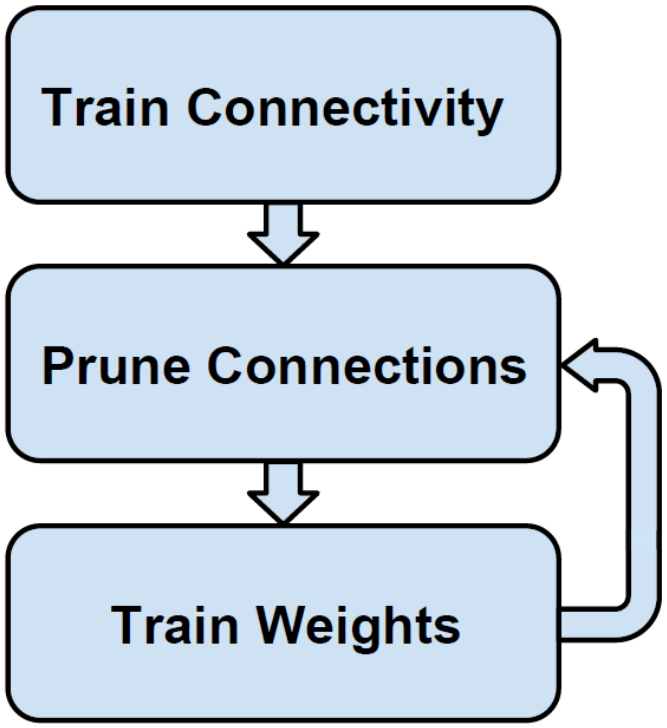
	$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

Groupwise Brain Damage

Group-wise sparsification with fine-tuning. Our first implementation is also based on the group-sparse regularizer (3). We start with the input ConvNet and run the learning process on the dataset D with the added regularizer (3). After a certain amount of iterations, a predefined number of groups Γ_{ijs} with the smallest $l2$-norm is set to zero. For a

$$\Omega_{2,1}(K) = \lambda \sum_{i,j,s} \|\Gamma_{ijs}\| = \lambda \sum_{i,j,s} \sqrt{\sum_{t=1}^{T} K(i,j,s,t)^2}$$

Fortunately, one can recover from most of this drop by the subsequent *fine-tuning* of the network, that follows after the brain-damage process. For the fine-tuning, we fix the sparsity patterns Q_S and restart learning without group-sparse regularization. We then train for an excessive number of epochs. As a result of such fine-tuning, the network adapts to the imposed sparsity patterns, while the prediction accuracy goes up and recovers most of the drop.



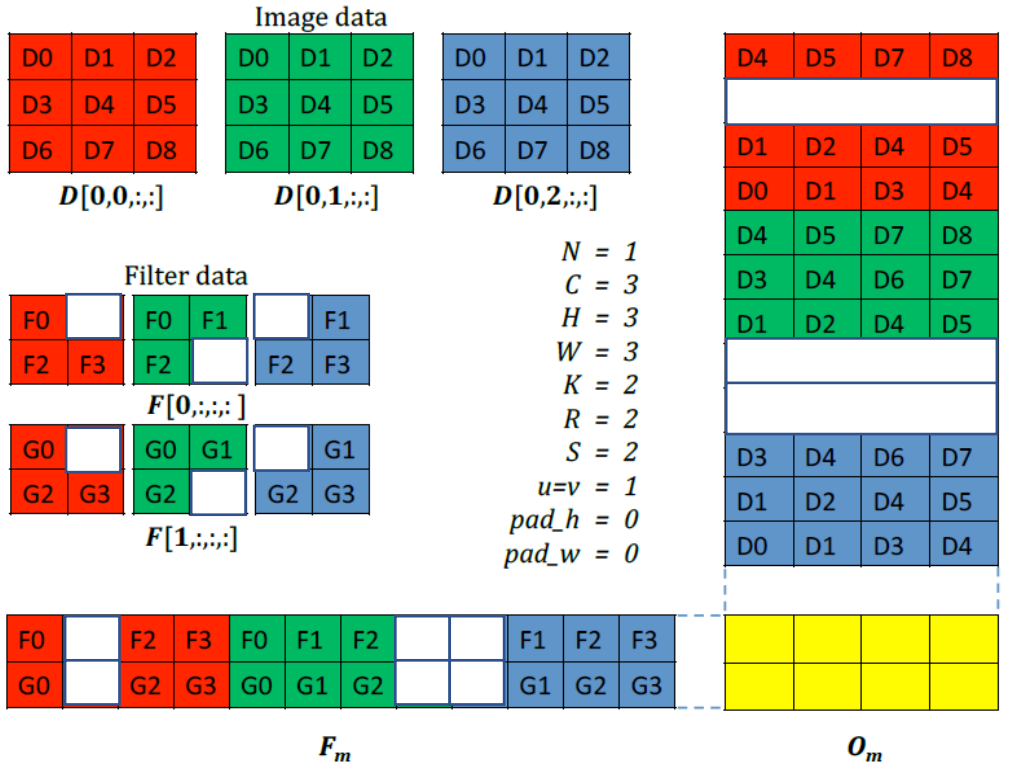
while fixing zero patterns
w/o the regularizer

Truncated Regularizer

- Regularizer represents only those columns having small weights
  - Columns having large weights are not included in the regularizer
  - Threshold θ controls pruning level

$$\Omega_{2,1}^{T}(K) = \lambda \sum_{i,j,s} \min(\|\Gamma_{ijs}\|, \theta)$$

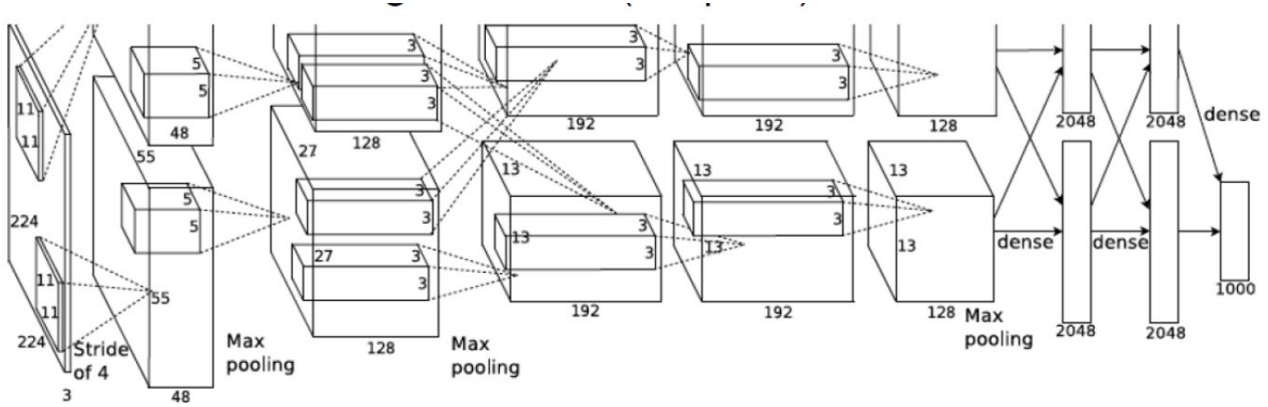
- Training
  - Given δ (allowed accuracy loss),
    - E.g., 1% of top-1 accuracy
  - After each epoch
    - If additional error < δ, increase θ
    - Else, decrease θ
    - If $|\Gamma|<\varepsilon$, it is fixed to zero ← pruning

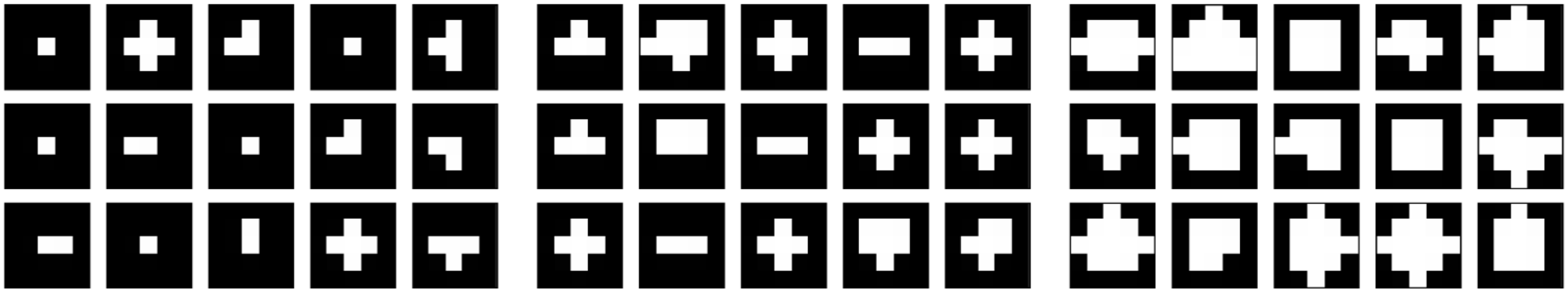


AlexNet Experiments (2nd Layer)

$$\Omega^T_{2,1}(K) = \lambda \sum_{i,j,s} \min(\|\Gamma_{ijs}\|, \theta)$$

- For each τ (i.e., sparsity level), we pick λ that results in the minimal accuracy drop after sparsification before fine-tuning. After picking the optimal λ, we perform fine-tuning.





(a) sparsity $1 - \tau = 0.9$ (b) sparsity $1 - \tau = 0.8$ (c) sparsity $1 - \tau = 0.6$

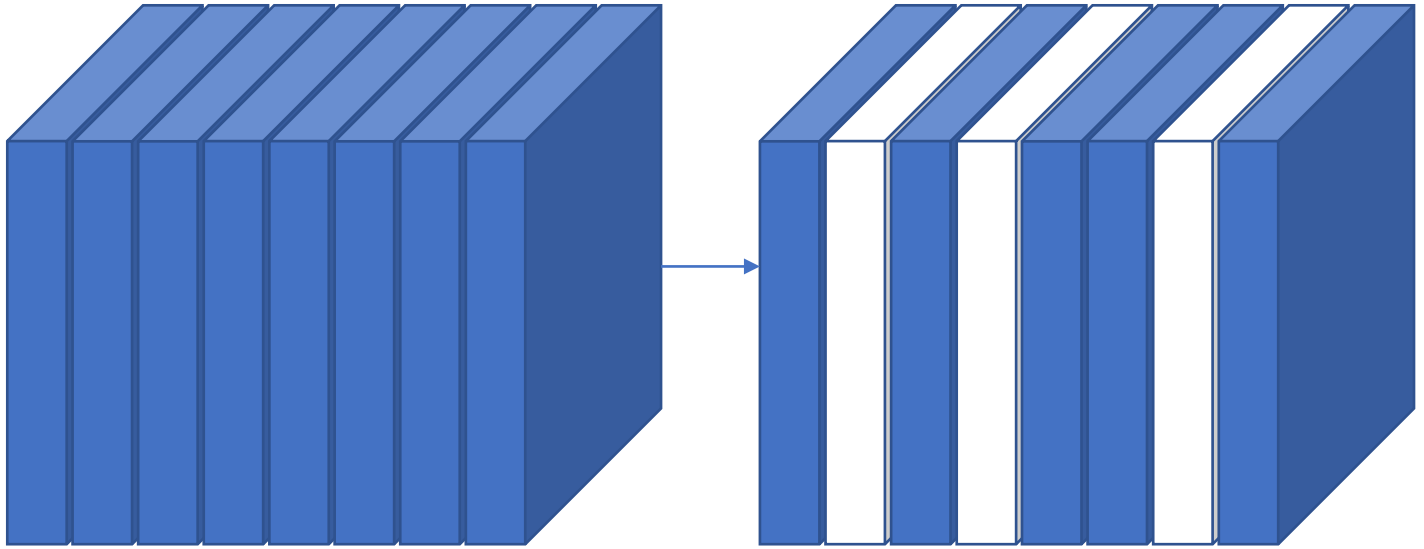
AlexNet Experiments

method	density	speed-up	accuracy drop
Accelerating the second convolutional layer of AlexNet			
Denton et al. [15]: Tensor decomposition + Fine-tuning		2.7x	$\sim 1\%$
Lebedev et al. [29]: CP-decomposition + Fine-tuning		4.5x	$\sim 1\%$
Jaderberg et al. [21]: Tensor decomposition + Fine-tuning		6.6x	$\sim 1\%$
Training with fixed sparsity patterns	0.12	8.33	0.82%
Training with fixed sparsity patterns	0.2	5x	0.16%
Group-wise sparsification + Fine-tuning	0.1	10x	1.13%
Group-wise sparsification + Fine-tuning	0.2	5x	0.43%
Group-wise sparsification + Fine-tuning	0.3	3.33x	0.11%
Group-wise sparsification + Fine-tuning	0.4	2.5x	-0.09%
Gradual group-wise sparsification	0.11	9.0x	0.28%
Gradual group-wise sparsification	0.05	20x	1.07%
Accelerating the second and the third convolutional layers of AlexNet			
Training with fixed sparsity patterns	0.12	8.7x	1.54%
Training with fixed sparsity patterns	0.35	2.9x	0.36%
Training with fixed sparsity patterns	0.54	1.9x	-0.53%
Group-wise sparsification + Fine-tuning	0.2	5x	1.50%
Group-wise sparsification + Fine-tuning	0.3	3.33x	1.17%
Group-wise sparsification + Fine-tuning	0.5	2x	0.57%
Gradual group-wise sparsification	0.12	8.5x	1.04%
Accelerating all five convolutional layers of AlexNet			
Training with fixed sparsity patterns	0.34	3.0x	1.34%
Gradual group-wise sparsification	0.31	3.2x	1.43%

Channel Pruning

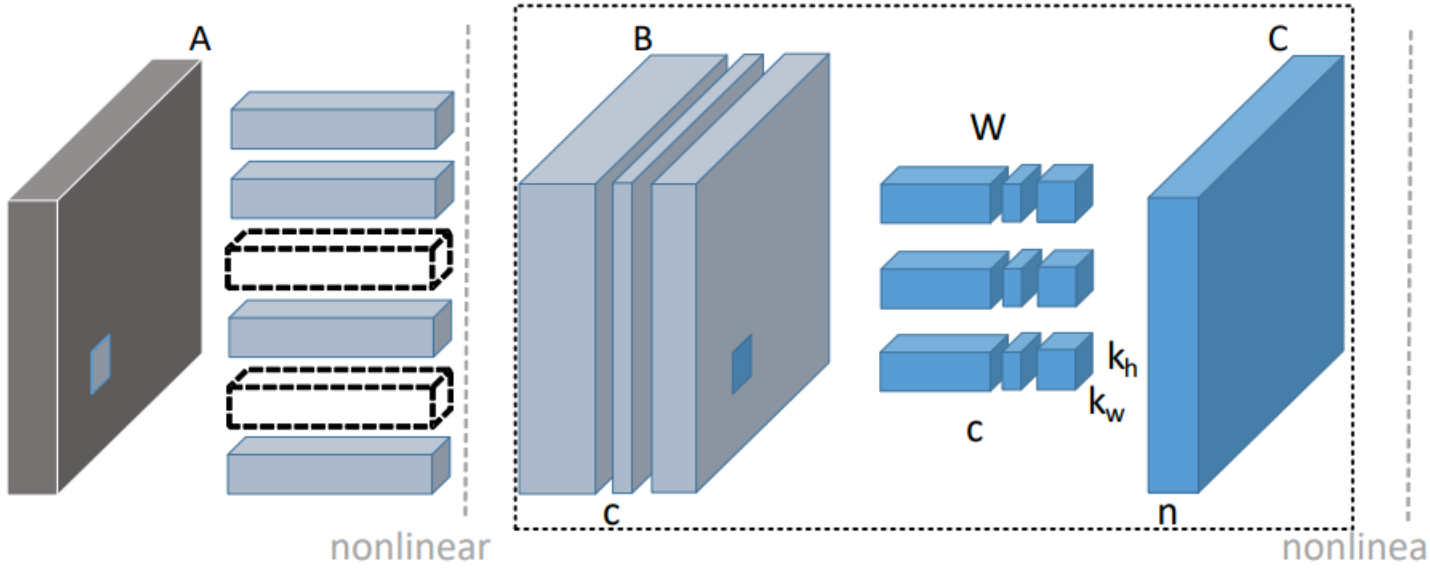
Chanel Pruning

- The sparse convolutional layers having an irregular shape are not hardware friendly
- Channel pruning gives a regular structure, but it is challenging because removing channels in one layer might change the input of the following layers



Pruning Method

- Aim to reduce the number of channels of feature map B, while minimizing the reconstruction error on feature map C
  - Perform within the dotted box, which does not involve nonlinearity
  - This figure illustrates the situation that two channels are pruned for feature map B
  - The corresponding filters in the previous layer can also be removed



Formulation

- $\arg\min_{\beta,W} \frac{1}{2N} \left|\left|Y - \Sigma_i \beta_i X_i W_i^T\right|\right|_F^2$ subject to $\left|\left|\beta\right|\right|_0 \leq c',\ 0 \leq c' \leq c$
- Solving the above l_0 minimization is NP-hard
  - Relax the l_0 to l_1 regularization (LASSO regression technique)
  - Add $\left|\left|W_i\right|\right|_F = 1$ condition to avoid the trivial solutions

- $\arg\min_{\beta,W} \frac{1}{2N} \left|\left|Y - \Sigma_i \beta_i X_i W_i^T\right|\right|_F^2 + \lambda\left|\left|\beta\right|\right|_1$ subject to $\left|\left|\beta\right|\right|_0 \leq c', \left|\left|W_i\right|\right|_F = 1,\ 0 \leq c' \leq c$

Formulation

- $\arg\min_{\beta,W} \frac{1}{2N} \left|\left|Y - \Sigma_i \beta_i X_i W_i^T\right|\right|_F^2 + \lambda \left|\left|\beta\right|\right|_1$ subject to $\left|\left|\beta\right|\right|_0 \leq c', \left|\left|W_i\right|\right|_F = 1,\ 0 \leq c' \leq c$
- Two-step optimization
  - Step 1: progressively increase λ until $\left|\left|\beta\right|\right|_0 \leq c'$

$$\hat{\boldsymbol{\beta}}^{LASSO}(\lambda) = \arg\min_{\boldsymbol{\beta}} \frac{1}{2N} \left\| \mathrm{Y} - \sum_{i=1}^{c} \beta_i \mathrm{Z_i} \right\|_F^2 + \lambda \left\|\boldsymbol{\beta}\right\|_1 \quad \text{subject to } \left\|\boldsymbol{\beta}\right\|_0 \leq c'$$

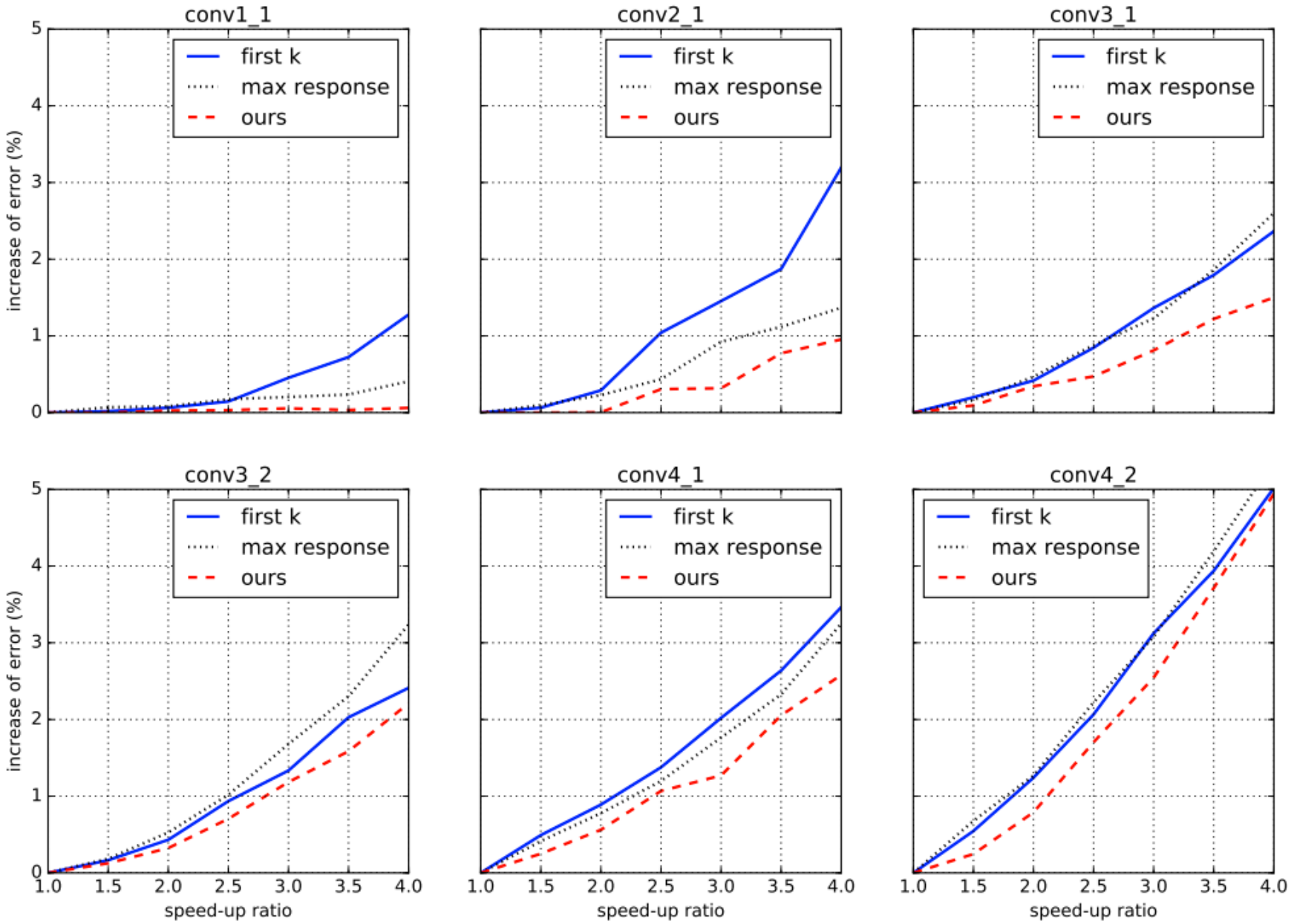
  - Step 2: find optimized solution by least squares

$$\arg\min_{\mathrm{W}'} \left\| \mathrm{Y} - \mathrm{X}'(\mathrm{W}')^{\top} \right\|_F^2$$

- Networks are updated to minimize the accumulated error by sequential pruning

$$\arg\min_{\boldsymbol{\beta},\mathrm{W}} \frac{1}{2N} \left\| \mathrm{Y}' - \sum_{i=1}^{c} \beta_i \mathrm{X_i} \mathrm{W_i}^{\top} \right\|_F^2 \text{ subject to } \left\|\boldsymbol{\beta}\right\|_0 \leq c'$$

Results


conv1_1
conv2_1
conv3_1
conv3_2
conv4_1
conv4_2
first k
max response
ours
increase of error (%)
speed-up ratio
0
1
2
3
4
5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Results

Increase of top-5 error (1-view, baseline 89.9%)			
Solution	2×	4×	5×
Jaderberg *et al.* [22] ([53]'s impl.)	-	9.7	29.7
Asym. [53]	0.28	3.84	-
Filter pruning [31] (fine-tuned, our impl.)	0.8	8.6	14.6
Ours (without fine-tune)	2.7	7.9	22.0
Ours (fine-tuned)	0	1.0	1.7

Table 1. Accelerating the VGG-16 model [44] using a speedup ratio of 2×, 4×, or 5× (*smaller is better*).

Original (acc. 89.9%)	Top-5 err.	Increased err.
From scratch	11.9	1.8
From scratch (uniformed)	12.5	2.4
Ours	18.0	7.9
Ours (fine-tuned)	11.1	**1.0**

Table 4. Comparisons with training from scratch, under 4× acceleration. Our fine-tuned model outperforms scratch trained counterparts (*smaller is better*).

Model	Solution	Increased err.	GPU time/ms
VGG-16	-	0	8.144
VGG-16 (4×)	Jaderberg *et al.* [22] ([53]'s impl.)	9.7	8.051 (1.01×)
	Asym. [53]	3.8	5.244 (1.55×)
	Asym. 3D [53]	0.9	8.503 (0.96×)
	Asym. 3D (fine-tuned) [53]	**0.3**	8.503 (0.96×)
	Ours (fine-tuned)	1.0	**3.264** (2.50×)

Table 3. GPU acceleration comparison. We measure forward-pass time per image. Our approach generalizes well on GPU (*smaller is better*).