Lightweight Deep Learning with Model Compression

U Kang
Dept. of Computer Science and Eng.
Seoul National University

Outline

- **Overview**
- Pruning
- Weight Sharing
- Quantization
- Low-rank Approximation
- Sparse Regularization
- Distillation
- Conclusion

Why Model Compression?

- Recent deep learning models are becoming more complex
  - LeNet-5: 1M parameters
  - VGG-16: 133M parameters
- Problems of complex deep models
  - Huge storage(memory, disk) requirement
  - Computationally expensive
  - Uses lots of energy
  - Hard to deploy models on small devices (e.g. smart phones)

Model Compression

- Goal: make a lightweight and accurate model that is fast, memory-efficient, and energy-efficient
  - Especially useful for edge device
  - Also useful for further improving the accuracy

- Several flavor
  - Whether training a lightweight model or compressing a trained model
  - Different techniques

Techniques or Model Compression

- Pruning
- Weight Sharing
- Quantization
- Low-rank Approximation
- Sparse Regularization
- Distillation

Outline

- ☑ Overview
- ☐ **Pruning**
- ☐ Weight Sharing
- ☐ Quantization
- ☐ Low-rank Approximation
- ☐ Sparse Regularization
- ☐ Distillation
- ☐ Conclusion

[Han et al. NIPS 2015]

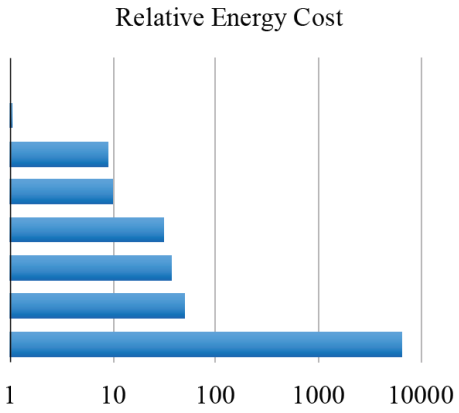
Learning both Weights and Connections for Efficient Neural Networks

- Standard deep neural network has…
  - Significant memory usage
  - Heavy power consumption

Input: Deep neural network to train

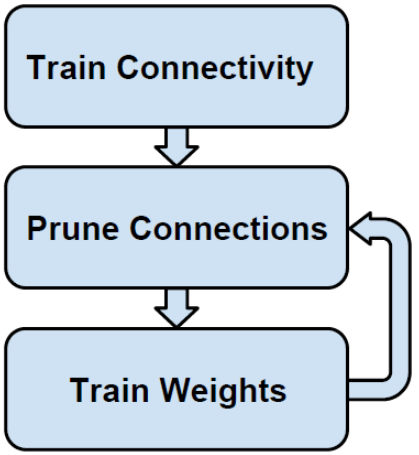
Output: Compressed deep neural network which preserves accuracy

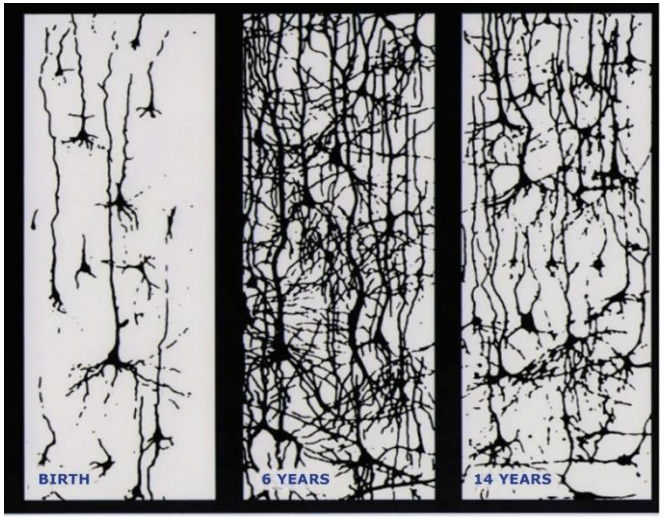
Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

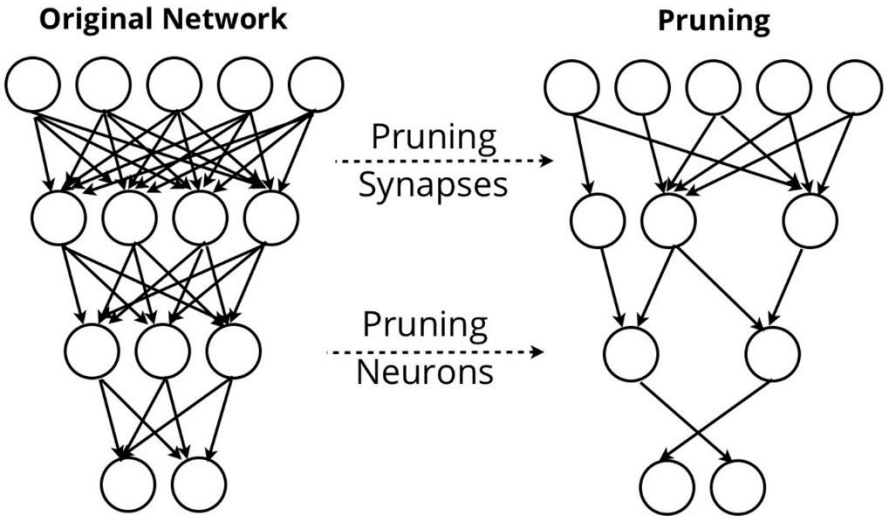


Learning both Weights and Connections for Efficient Neural Networks

- Pruning Weights
  - Motivated by how real brain learns
  - **Remove** weights which $|weight| < threshold$
  - **Retrain** after pruning weights
  - Learn effective connections by iterative pruning







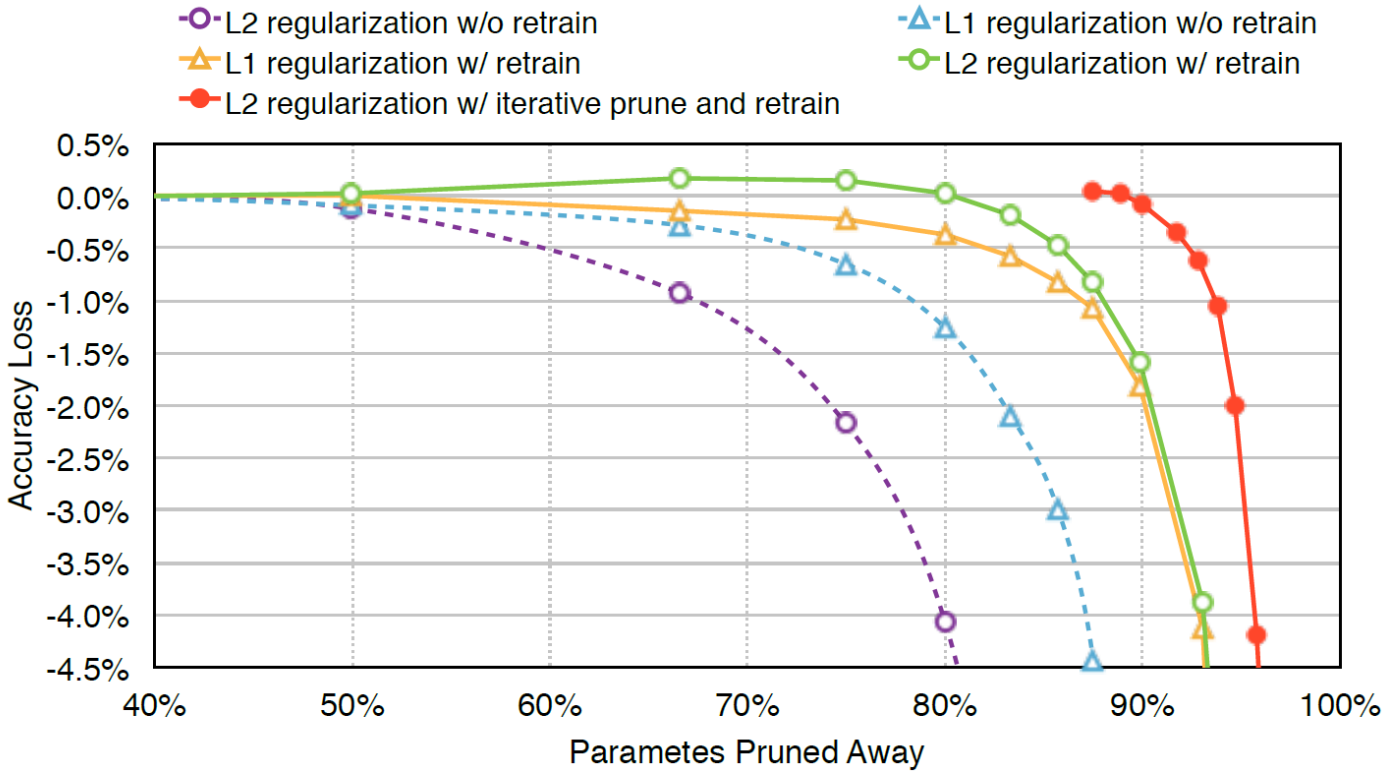
Learning both Weights and Connections for Efficient Neural Networks

- Compressing CNN by pruning
  - Experimented with LeNet (MNIST), AlexNet, VGGNet (ImageNet)
  - About 10x compared to original network

Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
LeNet-300-100 Ref	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	**22K**	**12**×
LeNet-5 Ref	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	**36K**	**12**×
AlexNet Ref	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9**×
VGG-16 Ref	31.50%	11.32%	138M	
VGG-16 Pruned	31.34%	10.88%	**10.3M**	**13**×

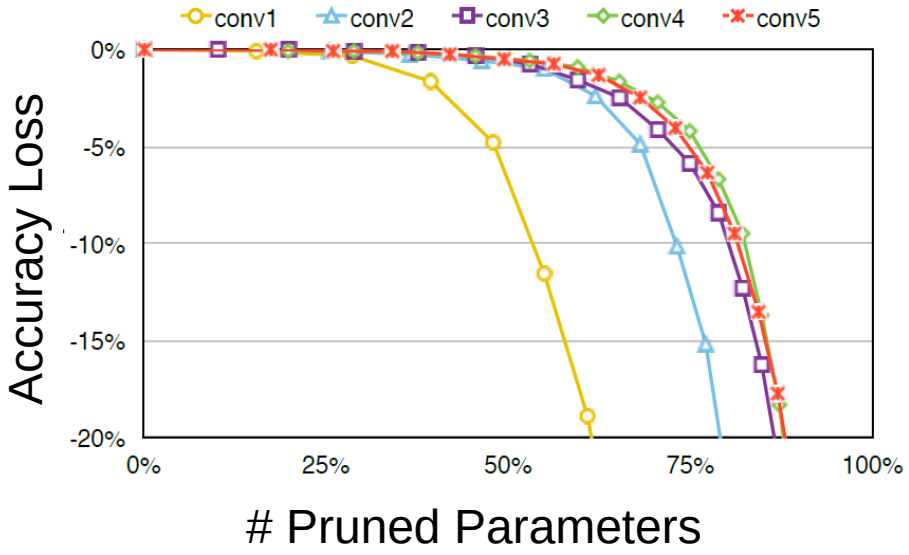
Learning both Weights and Connections for Efficient Neural Networks

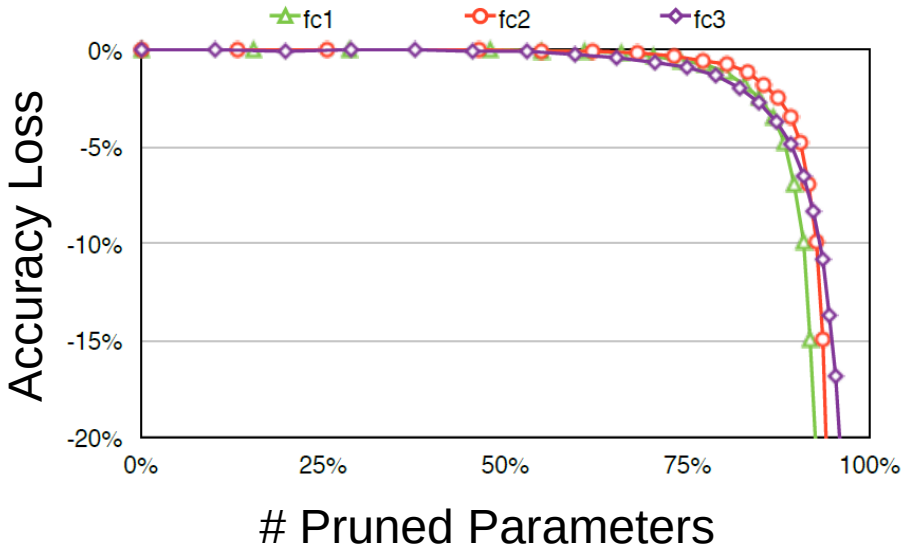
- Between L1 and L2, which regularization is better?
  - L2 is much better in iterative pruning



Learning both Weights and Connections for Efficient Neural Networks

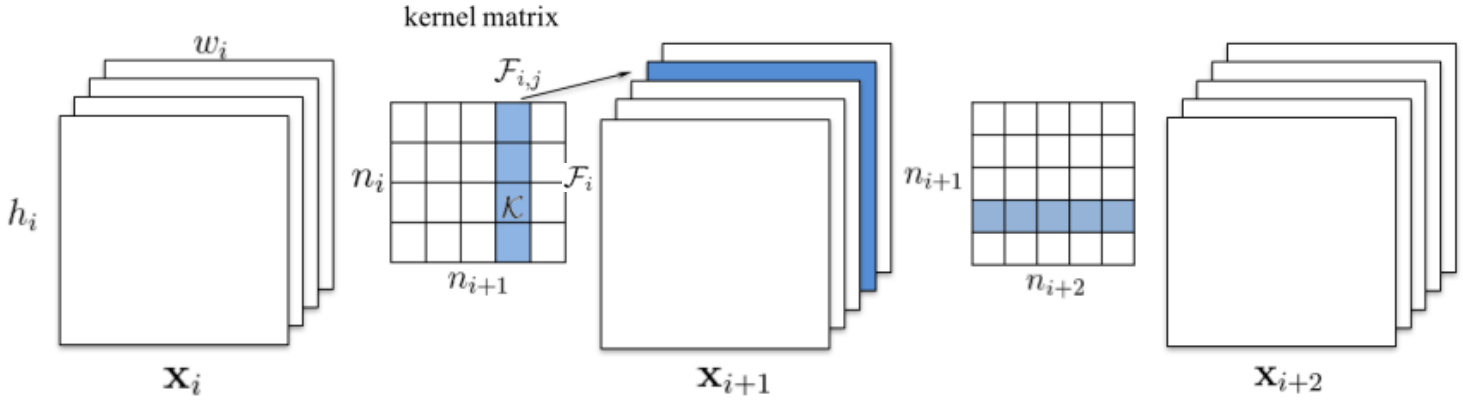
- Between convolutional layer and fully-connected layer, which layer is more sensitive to pruning?
  - Experimented with AlexNet
  - Convolutional layer is more sensitive





[Li et al. ICLR 2017]

Pruning Filters for Efficient ConvNets

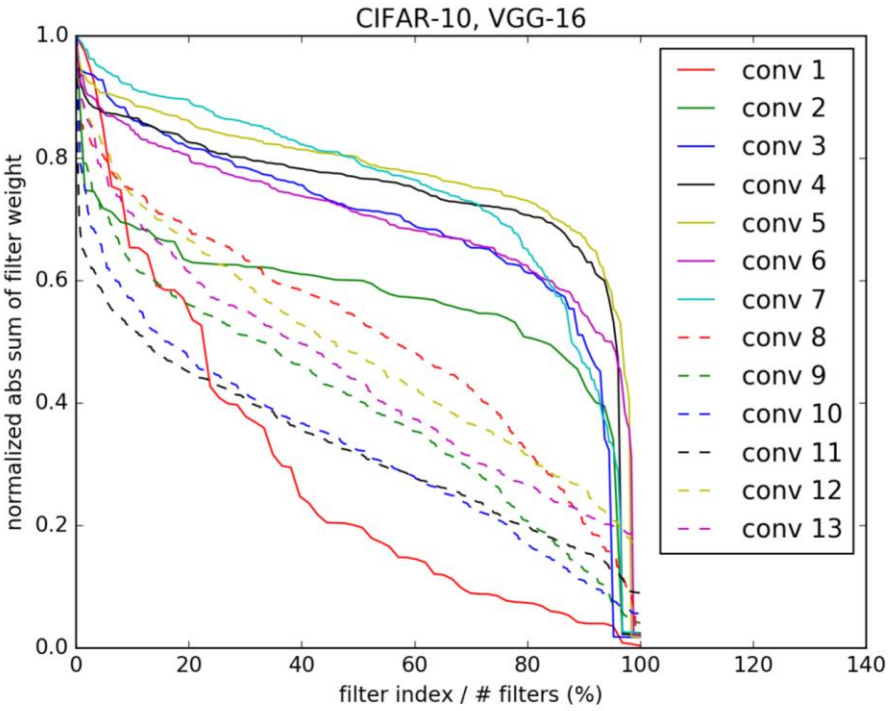
- Given a well-trained CNNs.
- Prune less important filters together with their connecting feature maps in order to reduce computation cost.
- 

x_i: input feature map
x_{i+1}: output feature map
h_i, w_i: height and width of input feature maps
n_i: number of input channels
n_{i+1}: number of output channels
F_i: kernel matrix - n_{i+1} 3D filters $F_{i,j} \in R^{n_i \times k \times k}$
$F_{i,j}$: filter - n_i 2D kernels $K \in R^{k \times k}$

Pruning m filters from layer i will reduce $\frac{m}{n_{i+1}}$ of the computational cost.

Pruning Filters for Efficient ConvNets

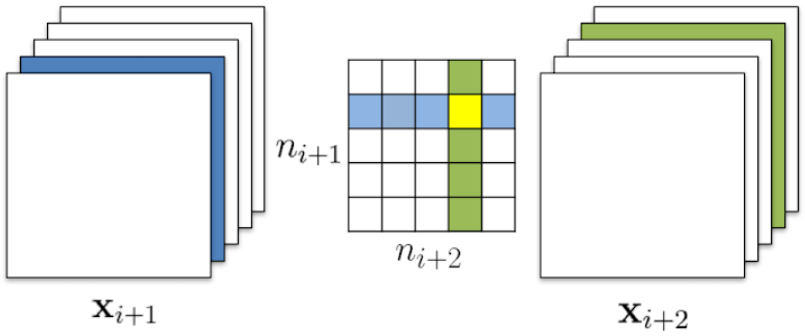
- Importance of filter
  - Defined as the sum of its absolute weights $s_j = \sum_i \left|\mathrm{F_{i,j}}\right|_1$, i.e., sum of its l_1-norm $\left\|\mathrm{F_{i,j}}\right\|_1$.



Filters are ranked by s_j

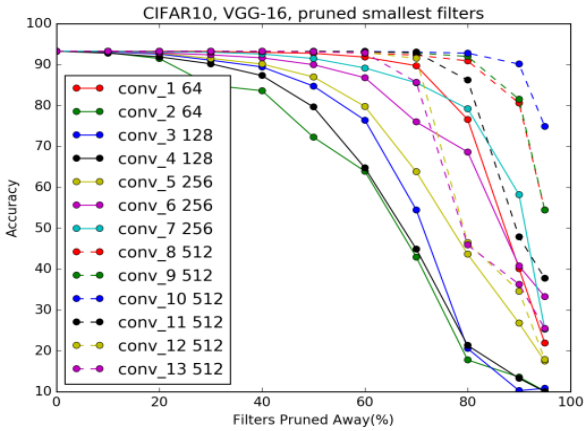
Pruning Filters for Efficient ConvNets

- When pruning filters across multiply layers, there are two strategies:
  - Should yellow block be considered while pruning next layer?
  - Independent pruning determines which filters should be pruned at each layer independent of other layers.
  - Greedy pruning accounts for the filters that have been removed in the previous layers. This strategy does not consider the kernels for the previously pruned feature maps while calculating the sum of absolute weights. (BETTER)

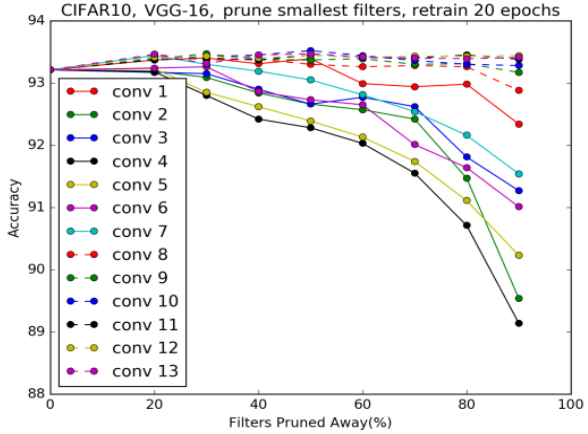


Pruning Filters for Efficient ConvNets

- After pruning the filters, the performance degradation should be compensated by retraining. There are two retraining strategies
  - Prune once and retrain until the original accuracy is restored. (FASTER)
  - Prune and retrain iteratively
    - Prune filters layer by layer or filter by filter and then retrain iteratively. The model is retrained before pruning the next layer for the weights to adapt to the changes from the pruning process. (BETTER RESULT – accuracy can be regained)



(b) Prune the smallest filters



(c) Prune and retrain

Pruning Filters for Efficient ConvNets

- Performance
  - The method achieves about 30% reduction in FLOP without significant loss in the original accuracy

Model	Error(%)	FLOP	Pruned %	Parameters	Pruned %
VGG-16	6.75	3.13×10^8		1.5×10^7	
VGG-16-pruned-A	**6.60**	2.06×10^8	34.2%	5.4×10^6	64.0%
VGG-16-pruned-A scratch-train	6.88				
ResNet-56	6.96	1.25×10^8		8.5×10^5	
ResNet-56-pruned-A	6.90	1.12×10^8	10.4%	7.7×10^5	9.4%
ResNet-56-pruned-B	**6.94**	9.09×10^7	27.6%	7.3×10^5	13.7%
ResNet-56-pruned-B scratch-train	8.69				
ResNet-110	6.47	2.53×10^8		1.72×10^6	
ResNet-110-pruned-A	**6.45**	2.13×10^8	15.9%	1.68×10^6	2.3%
ResNet-110-pruned-B	6.70	1.55×10^8	38.6%	1.16×10^6	32.4%
ResNet-110-pruned-B scratch-train	7.06				
ResNet-34	26.77	3.64×10^9		2.16×10^7	
ResNet-34-pruned-A	27.44	3.08×10^9	15.5%	1.99×10^7	7.6%
ResNet-34-pruned-B	27.83	2.76×10^9	24.2%	1.93×10^7	10.8%
ResNet-34-pruned-C	27.52	3.37×10^9	7.5%	2.01×10^7	7.2%

[Han et al. ICLR 2017]

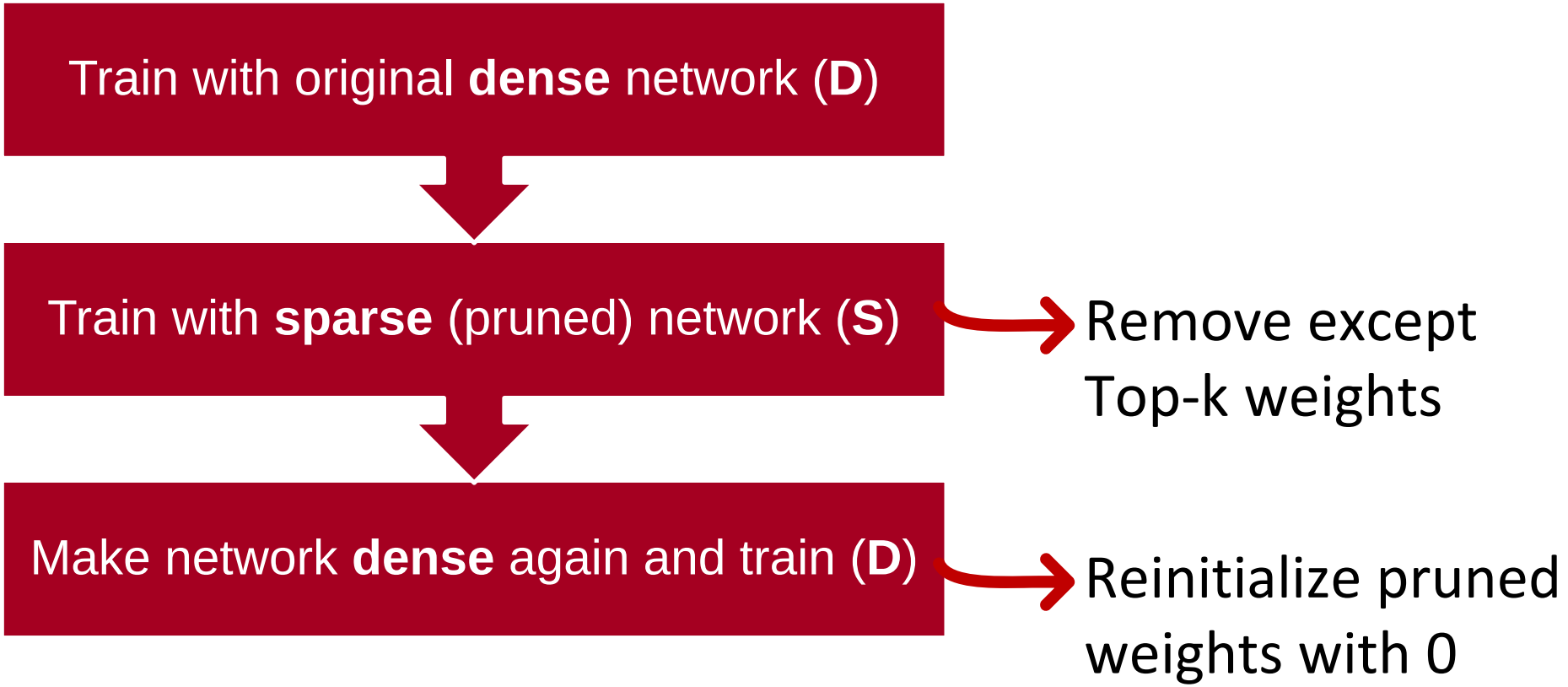
DSD: Dense-Sparse-Dense Training for Deep Neural Networks

- Complicate deep neural network often capture redundant noise
  - Over-fitting
  - High variance problem
- How can we train neural network with higher performance?

Input: Deep neural network (DNN)

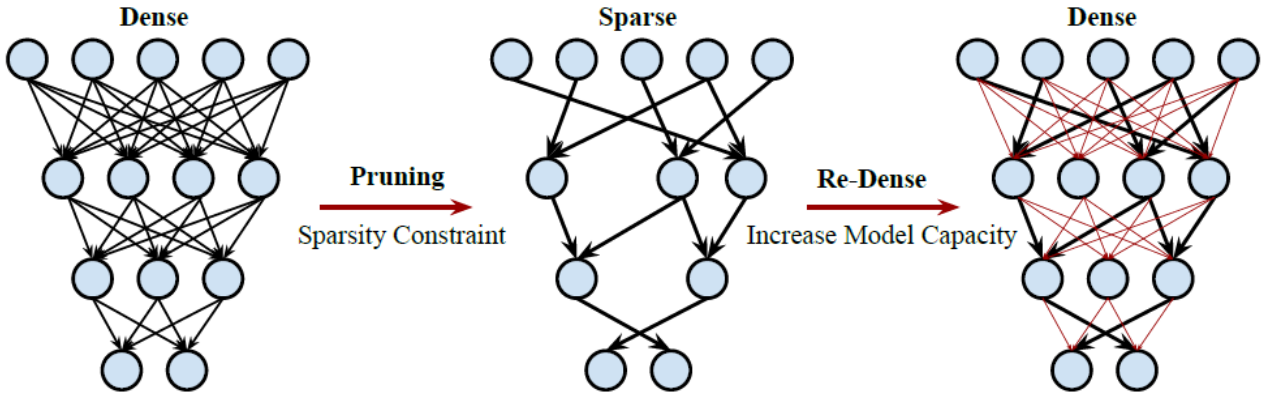
Output: Trained DNN with higher accuracy

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



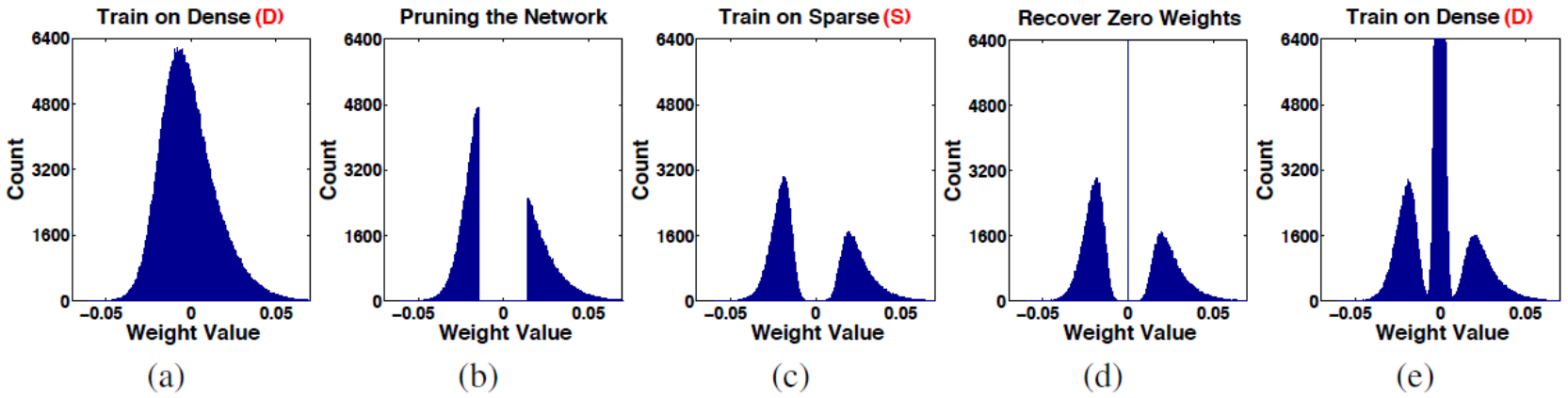
DSD: Dense-Sparse-Dense Training for Deep Neural Networks

- Overall process



1/10 of the original learning rate

- Weights distribution



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

- How much we can improve performance with DSD?
  - About **4-13**% relatively

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.

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

- Why does it work?
  - Escape saddle point: pruning the converged model perturbs the learning dynamics and allows the network to jump away from saddle points
  - Regularized sparse training
  - Robust re-initialization: DSD gives the optimization a second chance during the training process to re-initialize from more robust sparse training solution

Outline

- ☑ Overview
- ☑ Pruning
- ➡ ☐ **Weight Sharing**
- ☐ Quantization
- ☐ Low-rank Approximation
- ☐ Sparse Regularization
- ☐ Distillation
- ☐ Conclusion

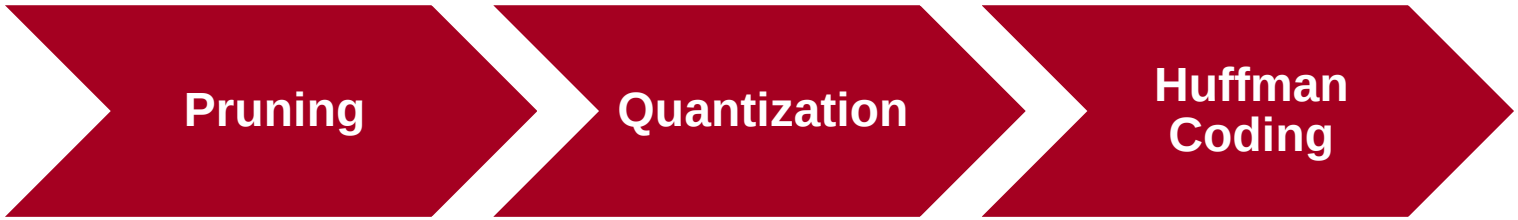
[Han et al. ICLR 2016]

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

- Current deep neural network has…
  - Heavy memory cost
  - Heavy power consumption
- Model compression deals with problems

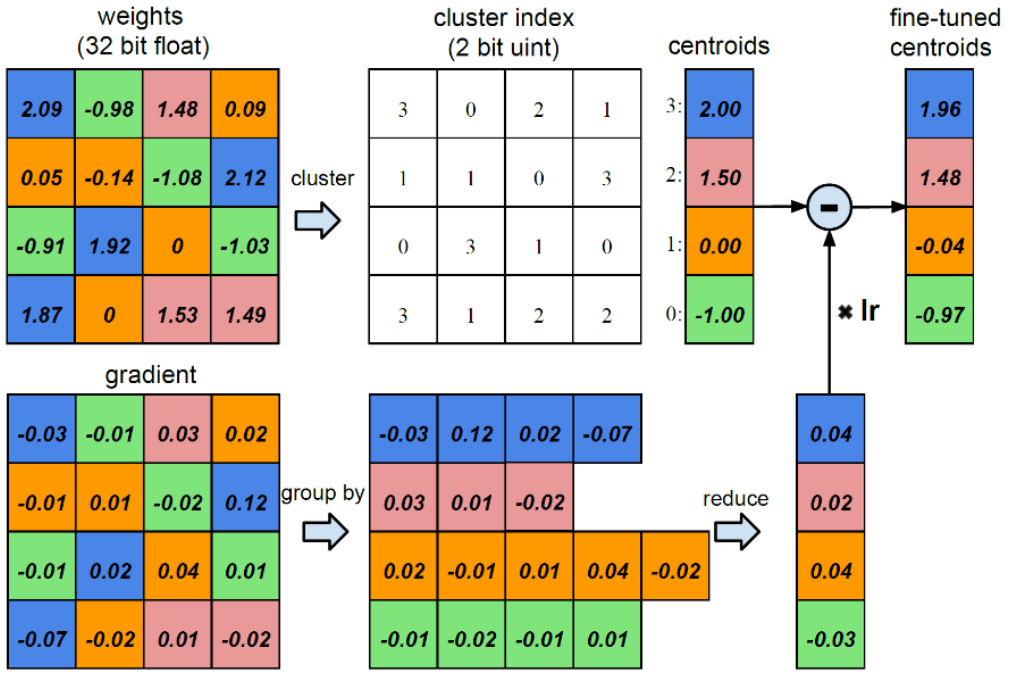
Input: Deep neural network

Output: Compressed deep neural network without accuracy loss



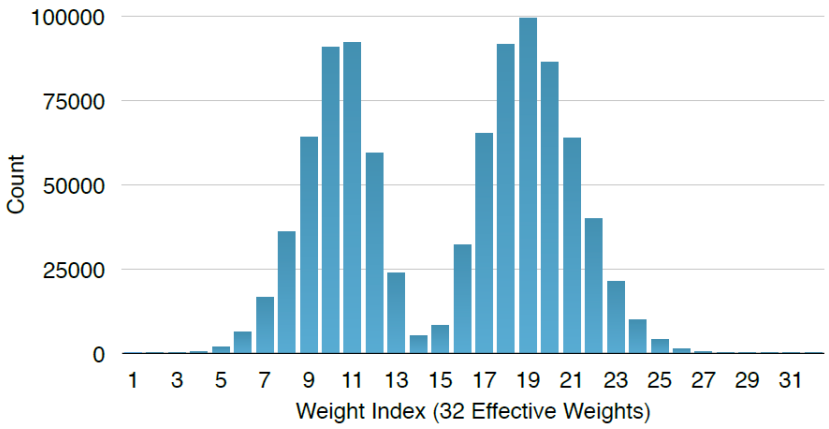
Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

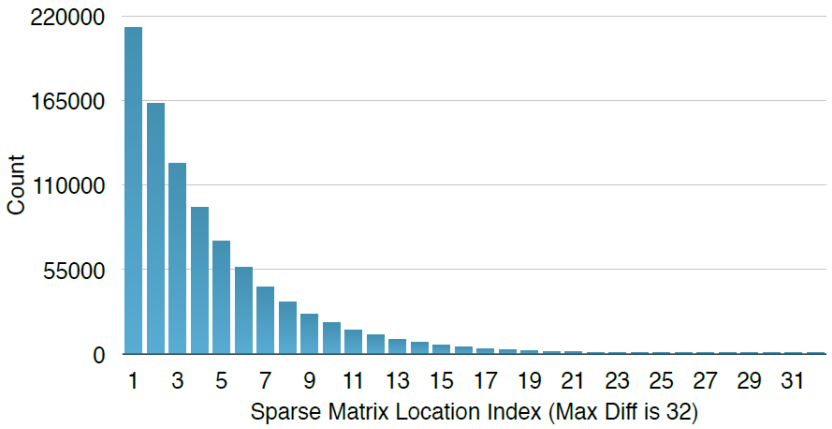
- Quantization and Weight Sharing
  - Cluster weights and use **centroid** of clustered weights in indexed array
  - Update centroid weights with **summation** of corresponding gradients



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

- Huffman Coding
  - Quantized weights are biased
  - Achieve additional compression via encoding weights





Biased weights

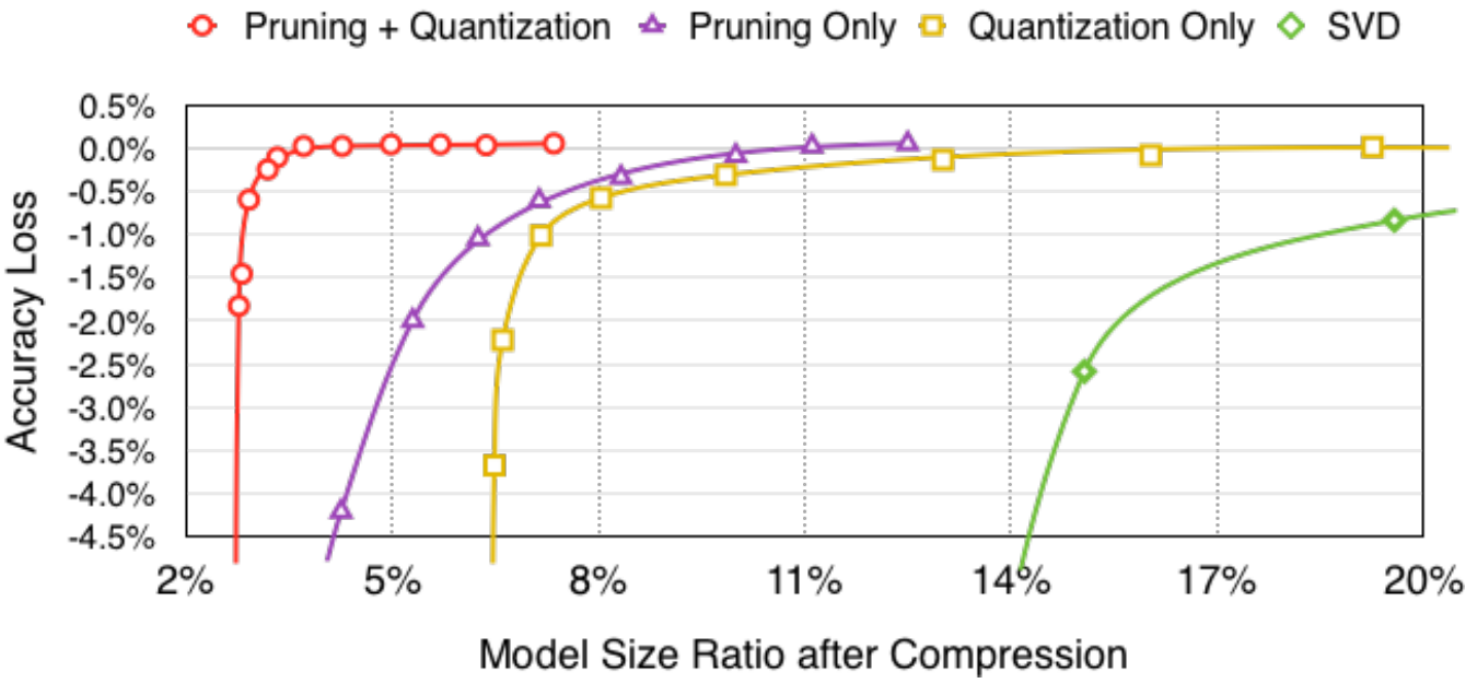
Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

- How much we can compress model?
  - About 40x without critical accuracy loss
  - MNIST data with LeNet-300-100, LeNet-100
  - ImageNet data with AlexNet, VGG-16

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**×

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

- Does pruning and quantization make synergy?
  - Result becomes much better when using two methods together
  - Model can be compressed up to 3% of original size



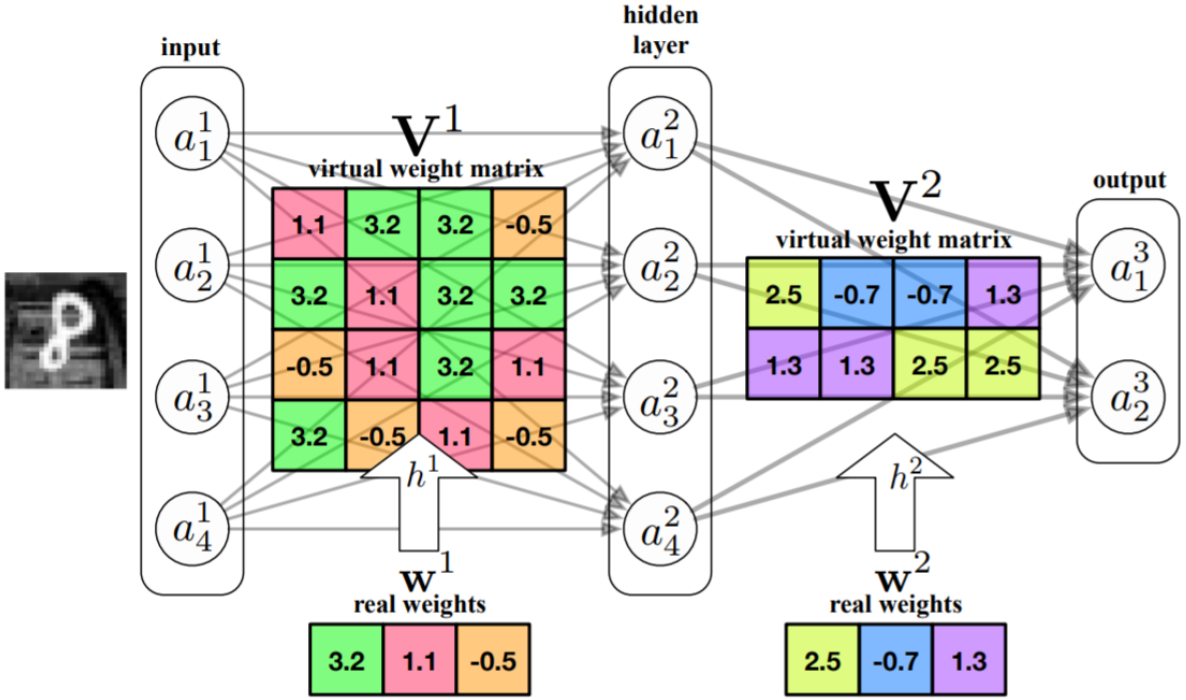
[Chen et al. NIPS 2015]

Compressing Neural Networks with the Hashing Trick

- Given a DNN,
- Compress the neural network with weight sharing

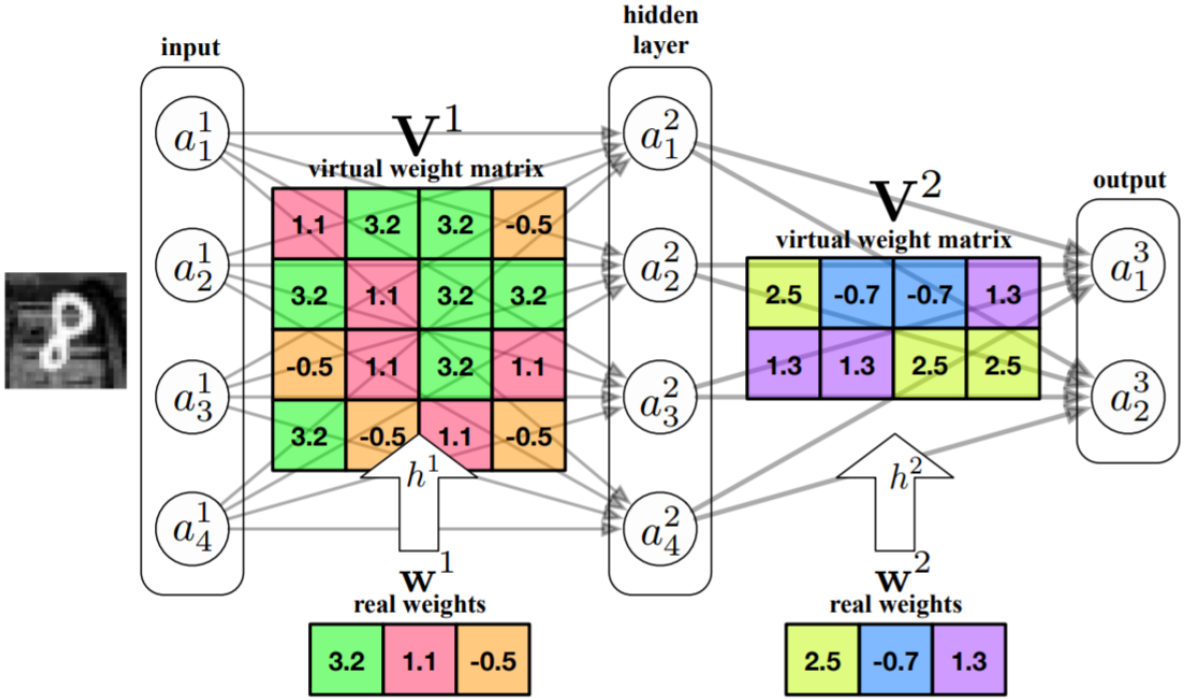
Compressing Neural Networks with the Hashing Trick

- Main idea: Hashed Net
  - Use a low-cost hash function to randomly group connection weights into hash buckets (weights within the same hash bucket share a same value) in order to reduce the model size.



Compressing Neural Networks with the Hashing Trick

- Why works?
  - A neural network has redundancy in its weights
  - Regularization effect



Compressing Neural Networks with the Hashing Trick

- Performance Comparison
  - HashNet and HashNetDK (using distillation) outperform all other baseline methods in accuracy, especially in the most interesting case when the compression factor is small.

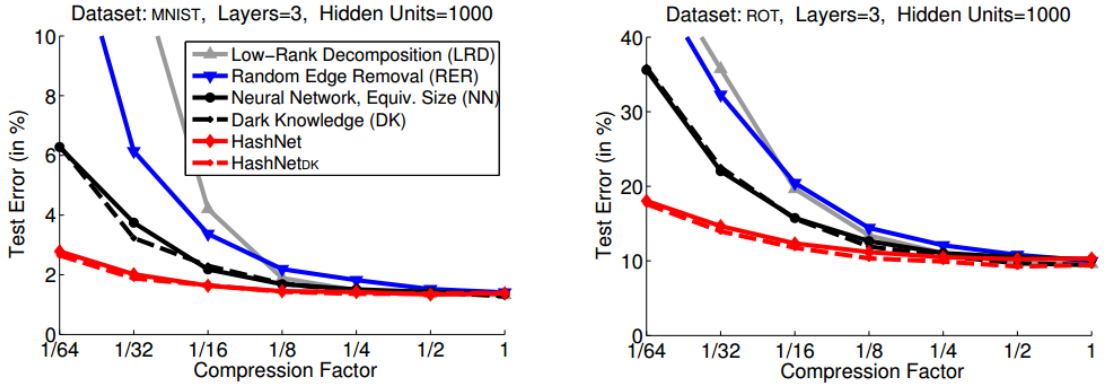


Figure 2. Test error rates under varying compression factors with 3-layer networks on MNIST (*left*) and ROT (*right*).

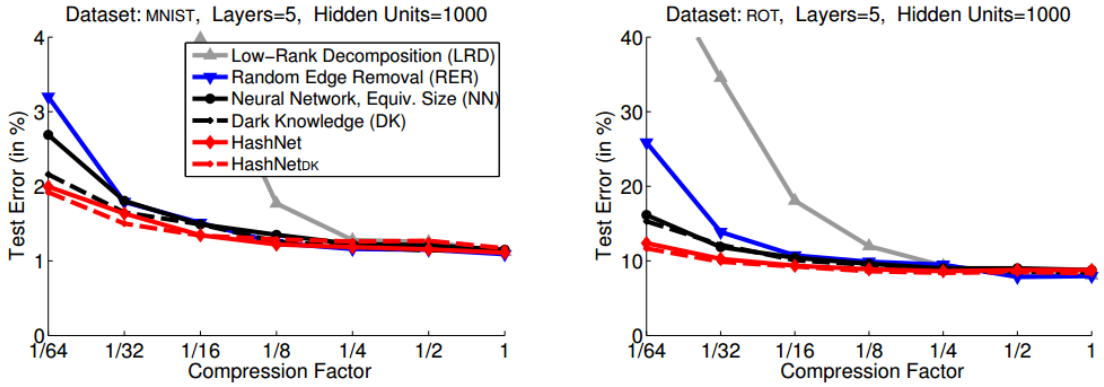


Figure 3. Test error rates under varying compression factors with 5-layer networks on MNIST (*left*) and ROT (*right*).

Compressing Neural Networks with the Hashing Trick

- Expansion: model size is fixed and the virtual network is 'inflated'
  - The test error of the HashNet neural network decreases substantially through expansion.
  - It shows that large regularized model is better than small unregularized model

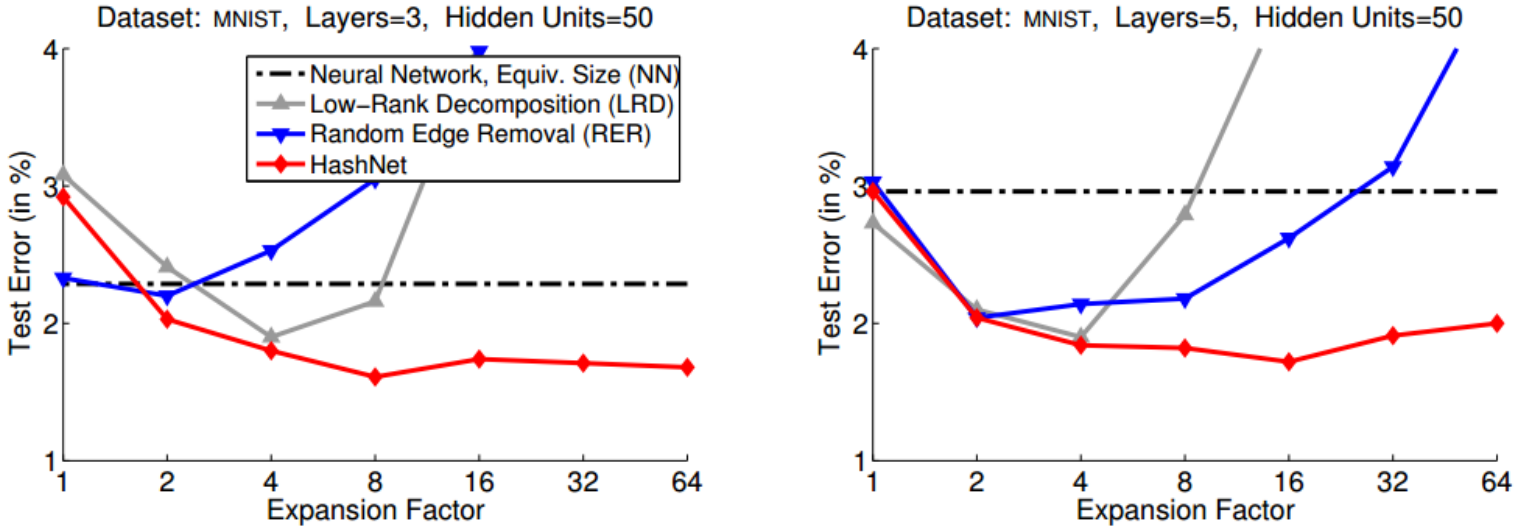


Figure 4. Test error rates with fixed storage but varying expansion factors on MNIST with 3 layers (*left*) and 5 layers (*right*).

Outline

- ☑ Overview
- ☑ Pruning
- ☑ Weight Sharing
- ➡ ☐ **Quantization**
- ☐ Low-rank Approximation
- ☐ Sparse Regularization
- ☐ Distillation
- ☐ Conclusion

[Courbariaux et al. NIPS 2015]

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- Goal: given a Deep Neural Network (DNN), quantize the networks for better memory efficiency, computation efficiency, and power consumption

- Main Idea
  - Train the DNN with binary weights during the forward and backward propagations, while retaining precision of the stored weights in which gradients are accumulated in order to regularize all the parameters.

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- Binarization
  - Deterministic
    - $w_b = \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{otherwise.} \end{cases}$
  - Stochastic
    - $w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w), \\ -1 & \text{with probability } 1 - p. \end{cases}$

    where $\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2}))$ is a hard sigmoid function which is used to limit p into [0, 1].

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- Performance comparison with other regularizer
  - BinaryConnect acts as a regularizer.
  - Performance is not worse (even better in CIFAR-10) than ordinary DNNs.

Method	MNIST	CIFAR-10	SVHN
No regularizer	$1.30 \pm 0.04\%$	10.64%	2.44%
BinaryConnect (det.)	$1.29 \pm 0.08\%$	9.90%	2.30%
BinaryConnect (stoch.)	$1.18 \pm 0.04\%$	**8.27%**	2.15%
50% Dropout	$1.01 \pm 0.04\%$		
Maxout Networks [29]	0.94%	11.68%	2.47%
Deep L2-SVM [30]	**0.87%**		
Network in Network [31]		10.41%	2.35%
DropConnect [21]			1.94%
Deeply-Supervised Nets [32]		9.78%	**1.92%**

Table 2: Test error rates of DNNs trained on the MNIST (no convolution and no unsupervised pretraining), CIFAR-10 (no data augmentation) and SVHN, depending on the method. We see that in spite of using only a single bit per weight during propagation, performance is not worse than ordinary (no regularizer) DNNs, it is actually better, especially with the stochastic version, suggesting that BinaryConnect acts as a regularizer.

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- BinaryConnect augments the training cost, while slows down the training, but lowers the validation error rate
  - Dashed line: training cost
  - Solid line: validation error rate

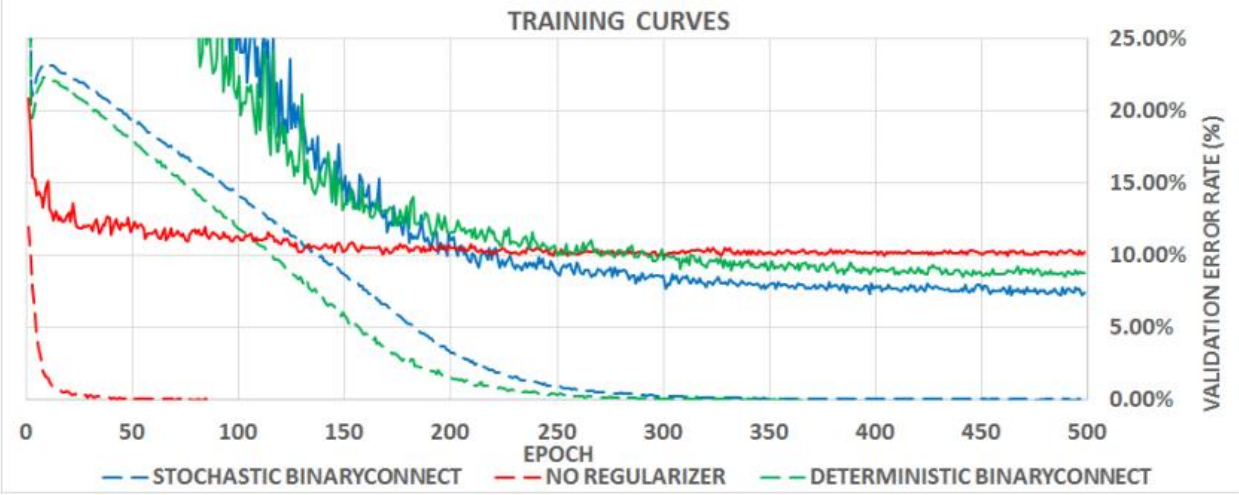
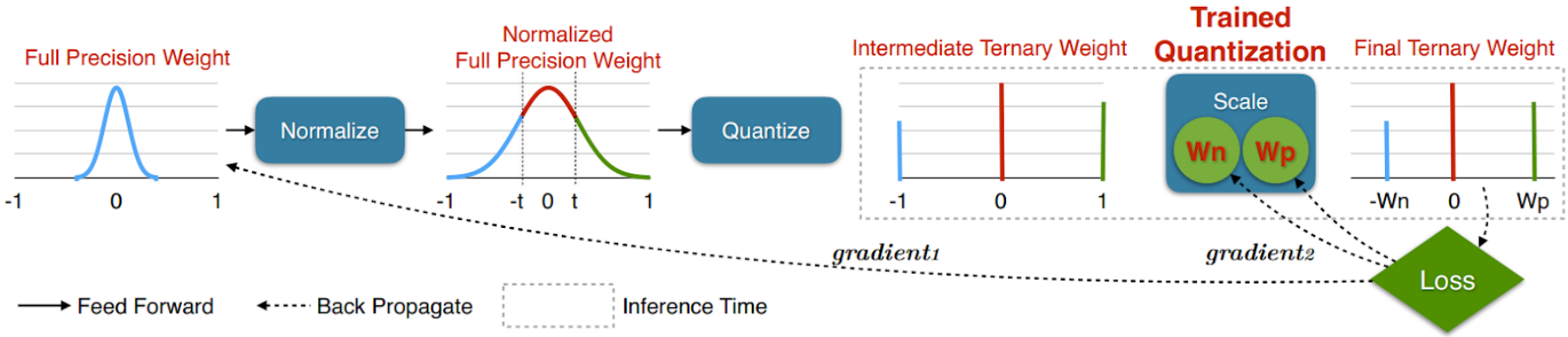


Figure 3: Training curves of a CNN on CIFAR-10 depending on the regularizer. The dotted lines represent the training costs (square hinge losses) and the continuous lines the corresponding validation error rates. Both versions of BinaryConnect significantly augment the training cost, slow down the training and lower the validation error rate, which is what we would expect from a Dropout scheme.

[Zhu et al. ICLR 2017]

Trained Ternary Quantization

- Given a DNN, reduce the precision of weights to ternary values in order to compress the model with little accuracy degradation.
- Procedure of TTQ



  - First, normalize the full-precision weights to range [-1, +1].
  - Second, quantize the intermediate full-resolution weights to {-1, 0, +1} by thresholding.
  - Finally, perform trained quantization by back propagating two gradients (one to full-resolution weights and one to scaling coefficients).

Trained Ternary Quantization

- Performance
  - TTQ improves the accuracy in most case.
  - The deeper the model, the larger the improvement

Model	Full resolution	Ternary (Ours)	Improvement
ResNet-20	8.23	**8.87**	**-0.64**
ResNet-32	7.67	**7.63**	**0.04**
ResNet-44	7.18	**7.02**	**0.16**
ResNet-56	6.80	**6.44**	**0.36**

Table 1: Error rates of full-precision and ternary ResNets on Cifar-10

Outline

- ☑ Overview
- ☑ Pruning
- ☑ Weight Sharing
- ☑ Quantization
- ➡ ☐ **Low-rank Approximation**
- ☐ Sparse Regularization
- ☐ Distillation
- ☐ Conclusion

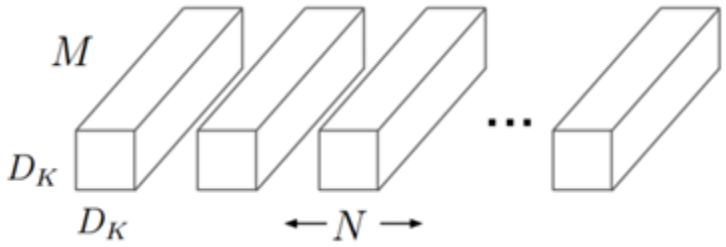
[Kim et al. ICLR 2016]

Compression of Deep CNN for Fast and Low Power Mobile Applications

- Goal: given a CNN, compress it to decrease size, running time, and energy consumption

Compression of Deep CNN for Fast and Low Power Mobile Applications

- Main Idea
  - Train a model
  - Tucker decomposition on the convolutional Kernel tensor
    - Kernel tensor can be thought of as 4-D tensor

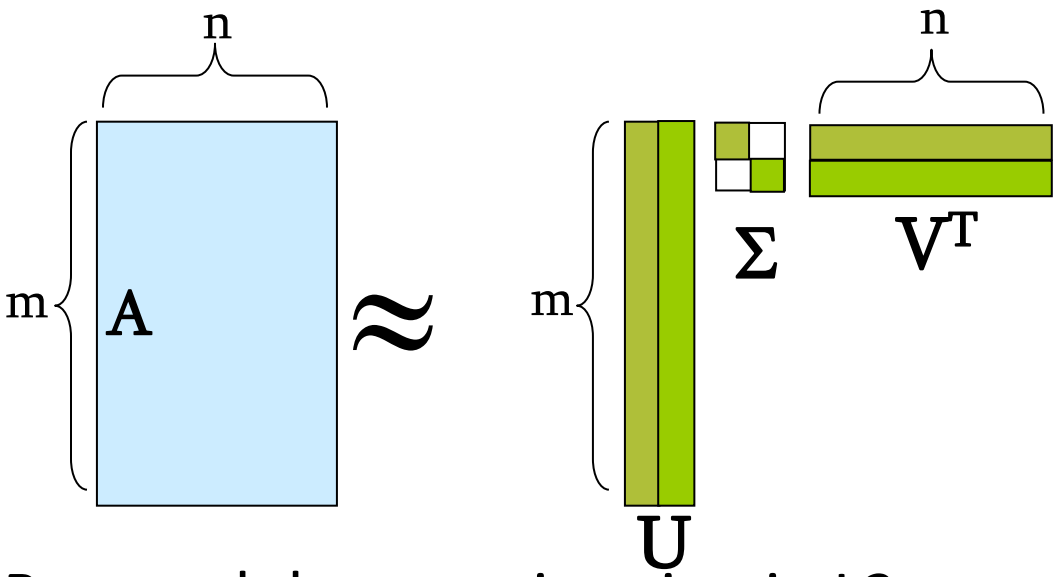


  - Fine tuning
    - Regain decreased accuracy due to approximation

Aside: Tucker Decomposition

SVD

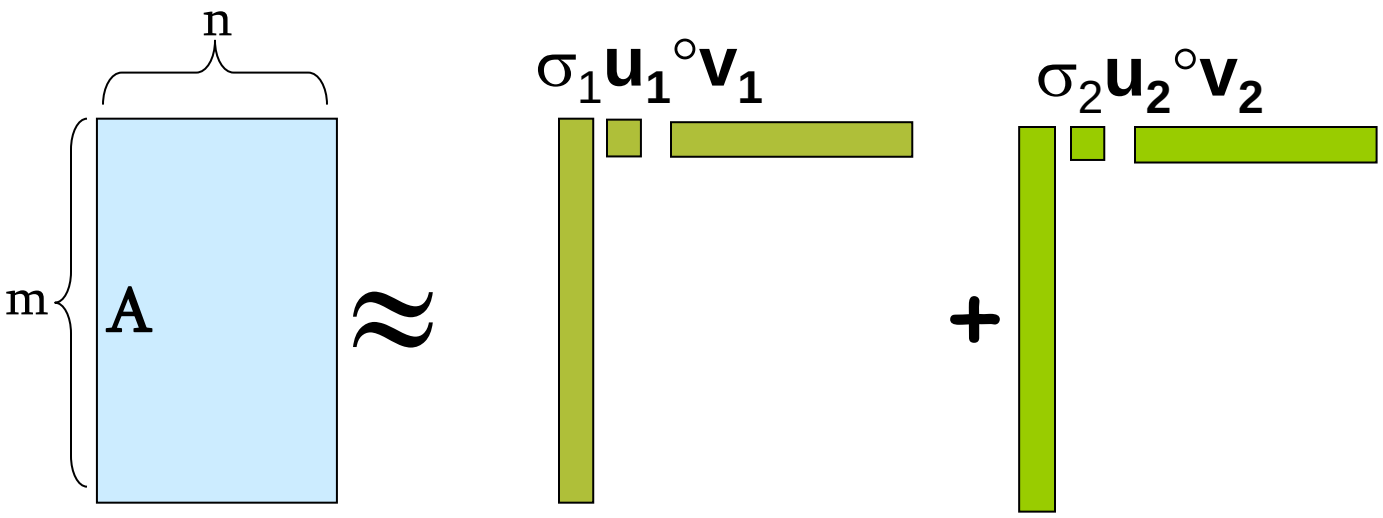
$$\mathbf{A} \approx \mathbf{U}\Sigma\mathbf{V}^T = \sum_i \sigma_i \mathbf{u}_i \circ \mathbf{v}_i$$



- Best rank-k approximation in L2

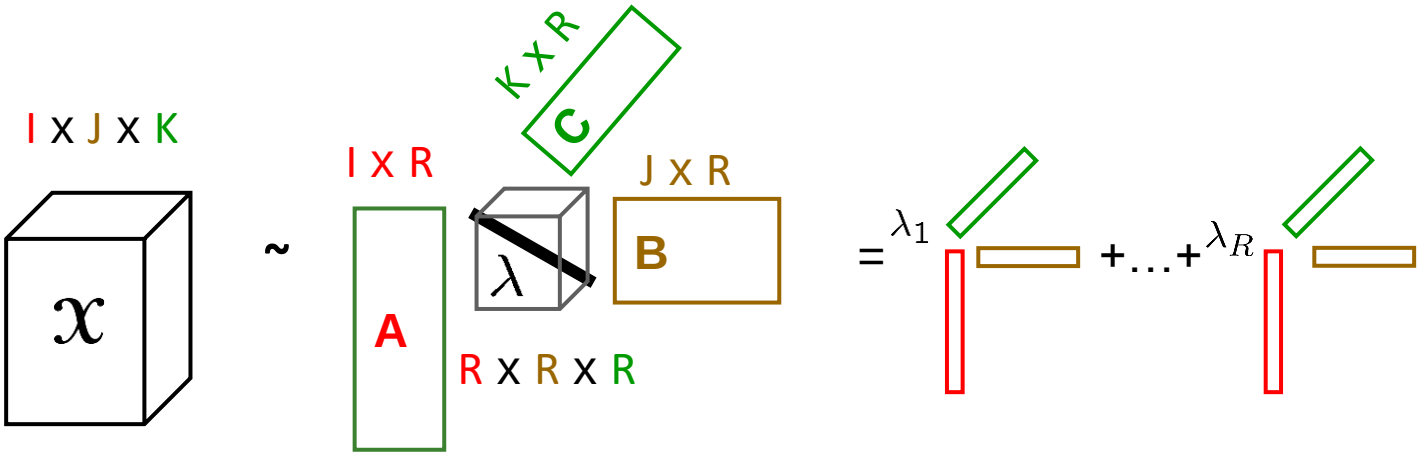
SVD

$$\mathbf{A} \approx \mathbf{U}\Sigma\mathbf{V}^T = \sum_i \sigma_i \mathbf{u}_i \circ \mathbf{v}_i$$



- Best rank-k approximation in L2

Tensor Analysis



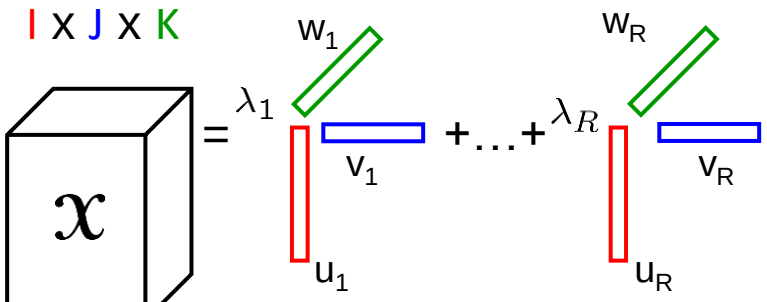
$$\mathcal{X} \approx [\![\lambda\ ; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!] = \sum_r \lambda_r\ \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$$

PARAFAC Decompoistion

PARAFAC vs Tucker

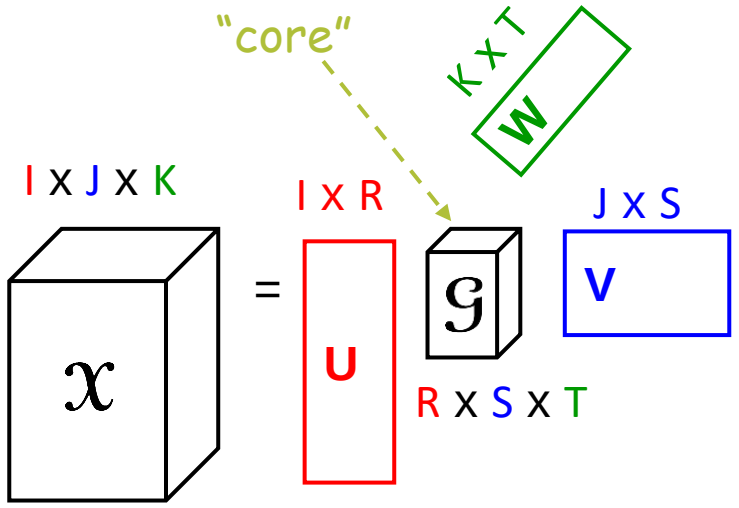
- PARAFAC Tensor

$$\mathcal{X} = \sum_r \lambda_r \, \mathbf{u}_r \circ \mathbf{v}_r \circ \mathbf{w}_r$$
$$\equiv [\![\lambda \, ; \mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$$



- Tucker Tensor

$$\mathcal{X} = \mathcal{G} \times_1 \mathbf{U} \times_2 \mathbf{V} \times_3 \mathbf{W}$$
$$= \sum_r \sum_s \sum_t g_{rst} \, \mathbf{u}_r \circ \mathbf{v}_s \circ \mathbf{w}_t$$
$$\equiv [\![\mathcal{G} \, ; \mathbf{U}, \mathbf{V}, \mathbf{W}]\!]$$



End of Aside

Compression of Deep CNN for Fast and Low Power Mobile Applications

Detail

- Convolutional Kernel Tensor
  - Convolution maps input tensor X of size H * W * S into target tensor Y of size H' * W' * T using the following linear mapping

$$\mathcal{Y}_{h',w',t} = \sum_{i=1}^{D}\sum_{j=1}^{D}\sum_{s=1}^{S}\mathcal{K}_{i,j,s,t}\,\mathcal{X}_{h_i,w_j,s}$$

$$h_i = (h'-1)\Delta + i - P \text{ and } w_j = (w'-1)\Delta + j - P$$

  - K is a 4-way kernel tensor of size D * D * S * T, Δ is stride, and P is zero-padding size

Compression of Deep CNN for Fast and Low Power Mobile Applications

Detail

- Tucker decomposition
  - The rank-(R_1, R_2, R_3, R_4) Tucker decomposition of 4-way kernel tensor K has the form

$$\mathcal{K}_{i,j,s,t} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathcal{C}'_{r_1,r_2,r_3,r_4} U^{(1)}_{i,r_1} U^{(2)}_{j,r_2} U^{(3)}_{s,r_3} U^{(4)}_{t,r_4}$$

  - The proposed method uses Tucker-2 decomposition

$$\mathcal{K}_{i,j,s,t} = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} \mathcal{C}_{i,j,r_3,r_4} U^{(3)}_{s,r_3} U^{(4)}_{t,r_4}$$

Compression of Deep CNN for Fast and Low Power Mobile Applications

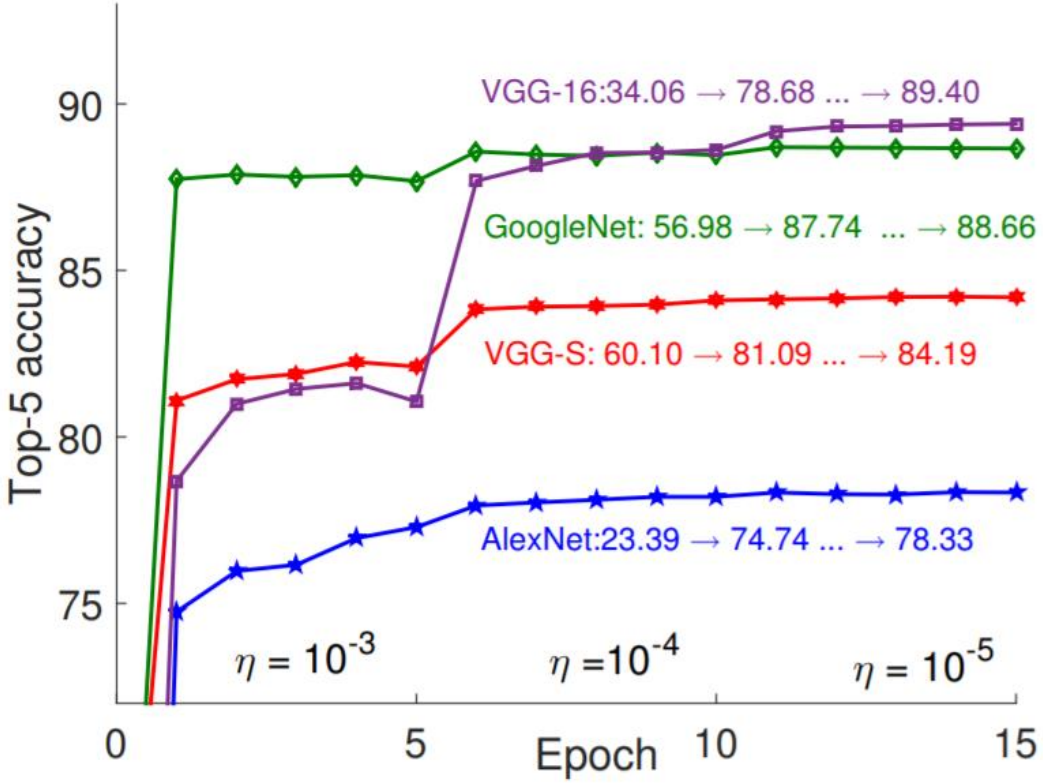
- Accuracy
  - Memory, runtime, and energy are significantly reduced with only minor accuracy drop

Proposed

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)

Compression of Deep CNN for Fast and Low Power Mobile Applications

- Fine tuning
  - Greatly helps improve accuracy



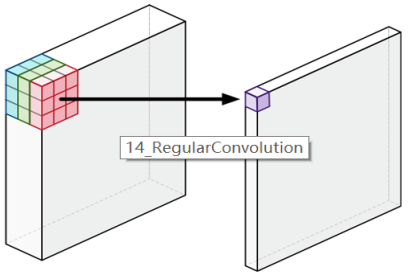
[Howard et al., Arxiv 2017]

MobileNets

- Goal: given a CNN, design a low-latency and memory efficient model that can easily be matched to the design requirements for mobile and embedded vision applications.

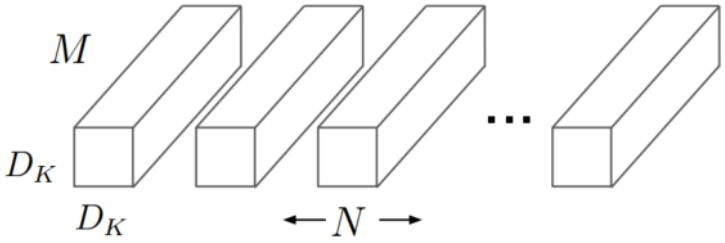
- Main Idea
  - Factorize convolutional layer into 2 layers: depthwise convolutional layer and pointwise convolutional layer(1×1 convolutional layer) in order to reduce model size and computational cost.

MobileNets

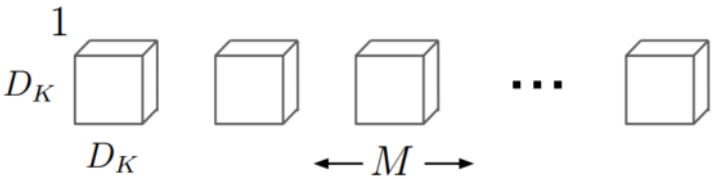


standard

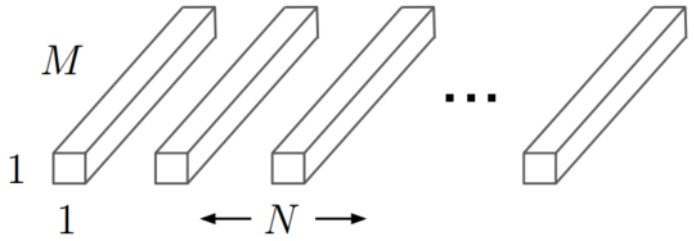
- The standard convolution both filters and combines inputs into a new set of output in one step.
- The depthwise separable convolution splits standard convolution into 2 layers:
  1. One for filtering.
  2. Another one for combining.



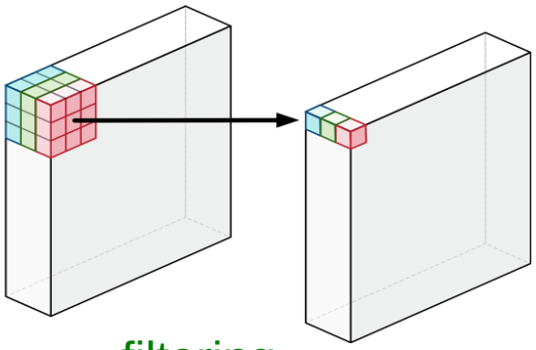
(a) Standard Convolution Filters



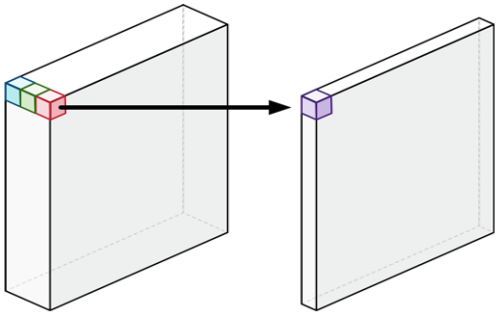
(b) Depthwise Convolutional Filters



(c) 1 × 1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution



filtering



combining

D_F: width/height of feature map

MobileNets

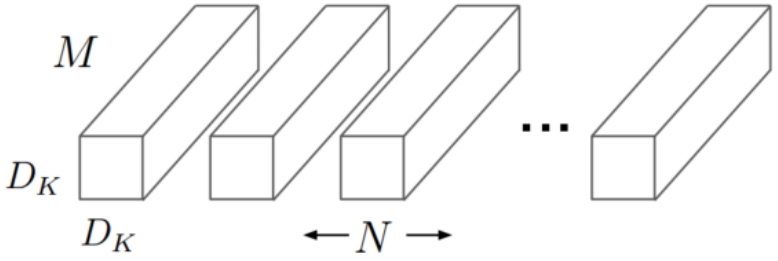
- Computational Cost
  - Standard

    $D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$
  - Depthwise Convolution

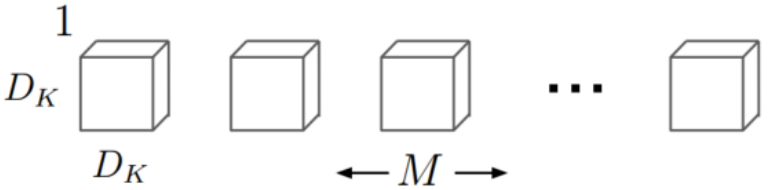
    $D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$
  - Pointwise Convolution

    $M \cdot N \cdot D_F \cdot D_F$
  - In total:

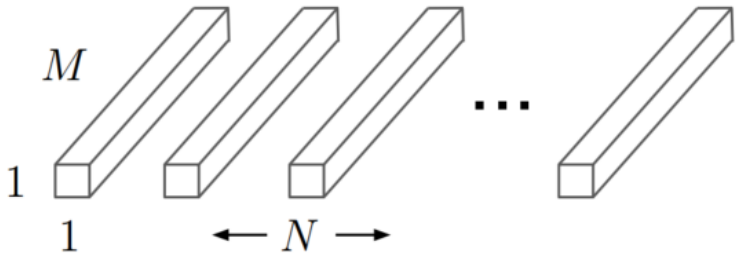
$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



× 1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Save 8-9 times computational cost for 3×3 conv

MobileNets

- Width Multiplier: Thinner Models
  - Width multiplier α makes thin feature maps at each layer
  - For a given layer and width multiplier α, the number of input channels M becomes αM and the number of output channels N becomes αN.
  - Computational Cost

$$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F$$

where $\alpha \in (0, 1]$.

  - Width Multiplier reduces computational cost and the number of parameters quadratically by roughly α^2.

MobileNets

- Resolution Multiplier: Reduced Representation
  - It is applied on the input image and the internal representation of every layer.
  - Computational Cost

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$$

where $\rho \in (0, 1]$.

  - Resolution multiplier has the effect of reducing computational cost by ρ^2.

MobileNets

- Computation and number of parameter for a layer as architecture shrinking methods are applied.

Table 3. Resource usage for modifications to standard convolution. Note that each row is a cumulative effect adding on top of the previous row. This example is for an internal MobileNet layer with $D_K = 3$, $M = 512$, $N = 512$, $D_F = 14$.

Layer/Modification	Million Mult-Adds	Million Parameters
Convolution	462	2.36
Depthwise Separable Conv	52.3	0.27
$\alpha = 0.75$	29.6	0.15
$\rho = 0.714$	15.1	0.15

MobileNets

- Depthwise Separable vs Full Convolution MobileNet
  - The proposed method saves huge computations and memory usage, while sacrificing only 1% of accuracy

Table 4. Depthwise Separable vs Full Convolution MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

MobileNets

- Thinner vs shallower
  - At similar computation and number of parameters, making MobileNets thinner (smaller kernel depth) is 3% better than making them shallower (# of layers)

Table 5. Narrow vs Shallow MobileNet

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
0.75 MobileNet	68.4%	325	2.6
Shallow MobileNet	65.3%	307	2.9

  - This confirms the effect of (sparse) deep models over (dense) shallow models

MobileNets

- MobileNet compared with GoogleNet & VGG16
  - Nearly as accurate as VGG16 with 32× smaller model and 27× less computation
  - More accurate than GoogleNet with smaller and 2.5× less computation

Table 8. MobileNet Comparison to Popular Models

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

Outline

- ☑ Overview
- ☑ Pruning
- ☑ Weight Sharing
- ☑ Quantization
- ☑ Low-rank Approximation
- ➡ ☐ **Sparse Regularization**
- ☐ Distillation
- ☐ Conclusion

[Wen et al. NIPS 2016]

Learning Structured Sparsity in Deep Neural Networks

- Given a CNN,
- Learn a structure-regularized version of the CNN to achieve speed-up with little accuracy loss

Learning Structured Sparsity in Deep Neural Networks

- Main Idea
  - Zero-out groups of weights using sparsity-inducing penalty (group Lasso)

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda \cdot R(\boldsymbol{W}) + \lambda_g \cdot \sum_{l=1}^{L} R_g\left(\boldsymbol{W}^{(l)}\right)$$

loss on data typical reg.

  - Group lasso penalty

$$R_g(\boldsymbol{w}) = \sum_{g=1}^{G} ||\boldsymbol{w}^{(g)}||_g \quad \text{where} \quad ||\boldsymbol{w}^{(g)}||_g = \sqrt{\sum_{i=1}^{|\boldsymbol{w}^{(g)}|} \left(w_i^{(g)}\right)^2}$$

    - E.g., $R_g(w) = \sqrt{w_1^2 + w_2^2} + \sqrt{w_3^2 + w_4^2}$

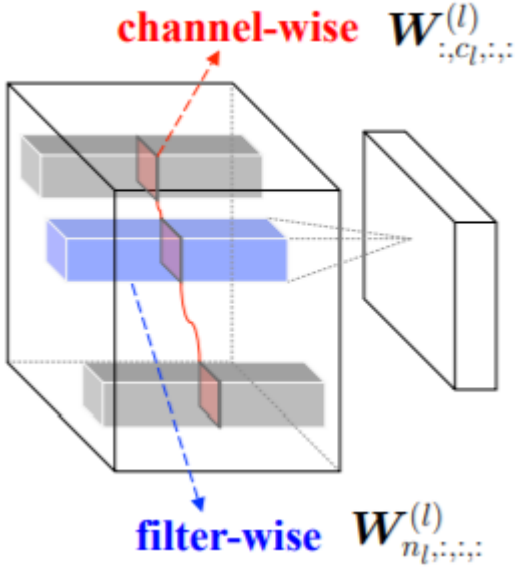
Learning Structured Sparsity in Deep Neural Networks

- 3 Types of structured sparsity
  - Consider weight of CNN kernels: $W^{(l)} \in R^{N_l \times C_l \times M_l \times K_l}$

filter channel height width

  - 1. Penalizing unimportant filters and channels

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda_n \cdot \sum_{l=1}^{L} \left(\sum_{n_l=1}^{N_l} ||\boldsymbol{W}^{(l)}_{n_l,:,:,:}||_g \right) + \lambda_c \cdot \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} ||\boldsymbol{W}^{(l)}_{:,c_l,:,:}||_g \right)$$



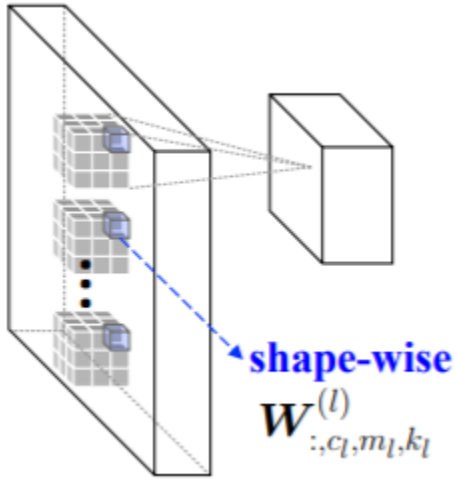
Learning Structured Sparsity in Deep Neural Networks

- 3 Types of structured sparsity
  - Consider weight of CNN kernels: $W^{(l)} \in R^{N_l \times C_l \times M_l \times K_l}$

filter channel height width

  - 2. Learning arbitrary shapes of filters

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda_s \cdot \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} \sum_{m_l=1}^{M_l} \sum_{k_l=1}^{K_l} ||\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}||_g \right)$$



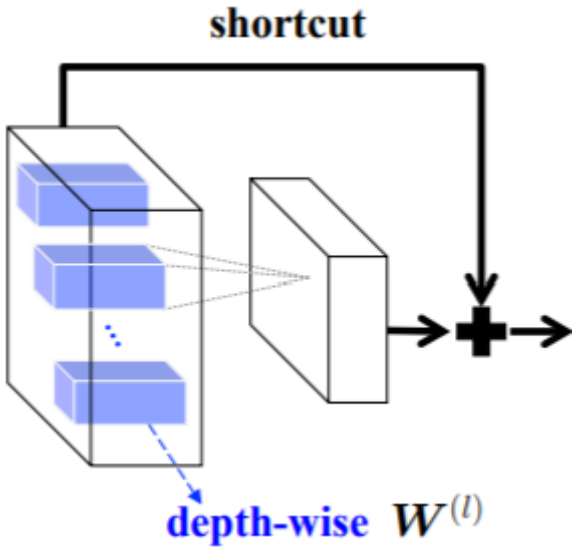
Learning Structured Sparsity in Deep Neural Networks

- 3 Types of structured sparsity
  - Consider weight of CNN kernels: $W^{(l)} \in R^{N_l \times C_l \times M_l \times K_l}$

filter channel height width

  - 3. Regularizing layer depth

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda_d \cdot \sum_{l=1}^{L} ||\boldsymbol{W}^{(l)}||_g$$



Learning Structured Sparsity in Deep Neural Networks

- LeNet on MNIST
  - 2.8% of flops and 10.82x speed-up with accuracy loss 0.1 %

Table 1: Results after penalizing unimportant filters and channels in *LeNet*

LeNet #	Error	Filter # §	Channel # §	FLOP §	Speedup §
1 (*baseline*)	0.9%	20—50	1—20	100%—100%	1.00×—1.00×
2	0.8%	5—19	1—4	25%—7.6%	1.64×—5.23×
3	1.0%	3—12	1—3	15%—3.6%	1.99×—7.44×

§In the order of *conv1*—*conv2*

Table 2: Results after learning filter shapes in *LeNet*

LeNet #	Error	Filter size §	Channel #	FLOP	Speedup
1 (*baseline*)	0.9%	25—500	1—20	100%—100%	1.00×—1.00×
4	0.8%	21—41	1—2	8.4%—8.2%	2.33×—6.93×
5	1.0%	7—14	1—1	1.4%—2.8%	5.19×—10.82×

§ The sizes of filters after removing zero shape fibers, in the order of *conv1*—*conv2*

Learning Structured Sparsity in Deep Neural Networks

- LeNet on MNIST
  - Sparse filters after inducing structured sparsity

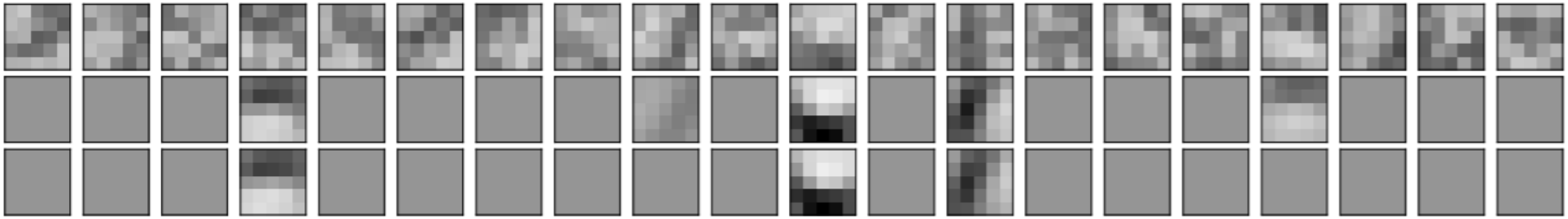
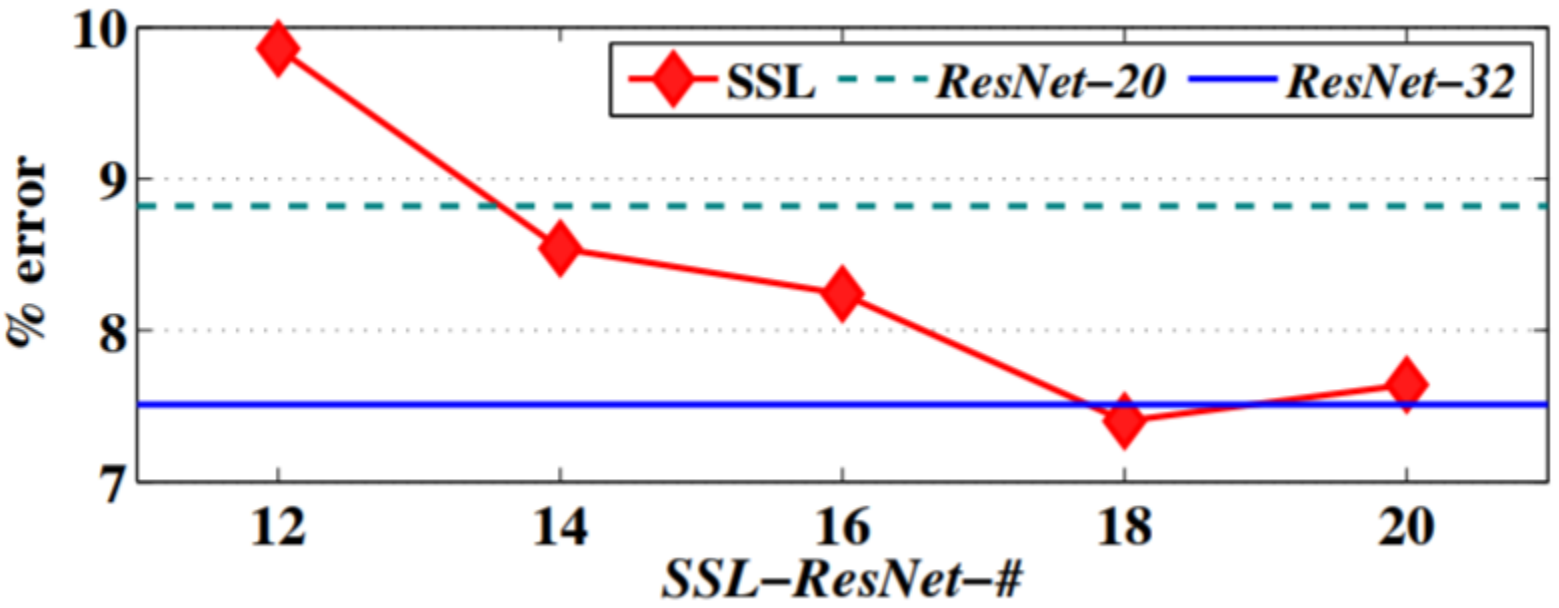


Figure 3: Learned *conv1* filters in *LeNet 1* (top), *LeNet 2* (middle) and *LeNet 3* (bottom)

Learning Structured Sparsity in Deep Neural Networks

- ResNet on CIFAR-10
  - SSL-ResNet-20 (proposed) outperforms ResNet-20 in accuracy
  - SSL works as a regularizer



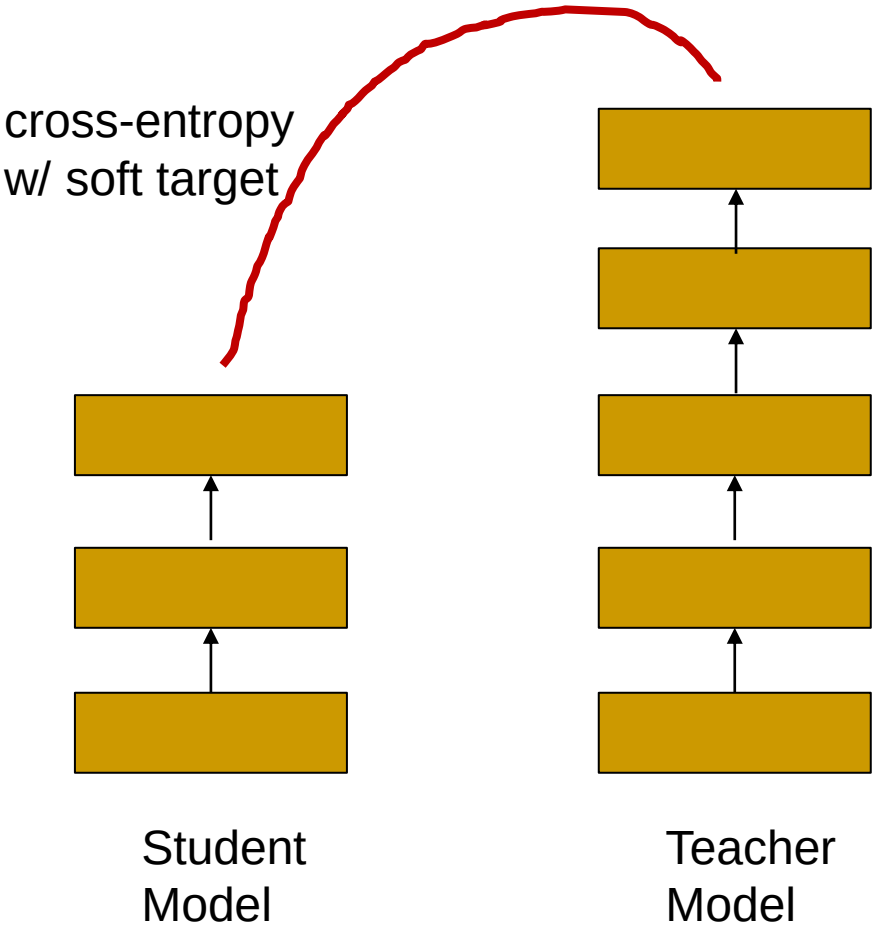
Outline

- ☑ Overview
- ☑ Pruning
- ☑ Weight Sharing
- ☑ Quantization
- ☑ Low-rank Approximation
- ☑ Sparse Regularization
- ➡ ☐ **Distillation**
- ☐ Conclusion

[Hinton et al. 2015]

Distillation

- Based on teacher-student model



Making soft target:

$$q_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)}$$

Large T makes softer distribution

The small model is trained with
the weighted average of
- cross-entropy w/ teacher's soft target
- cross-entropy w/ correct labels

Outline

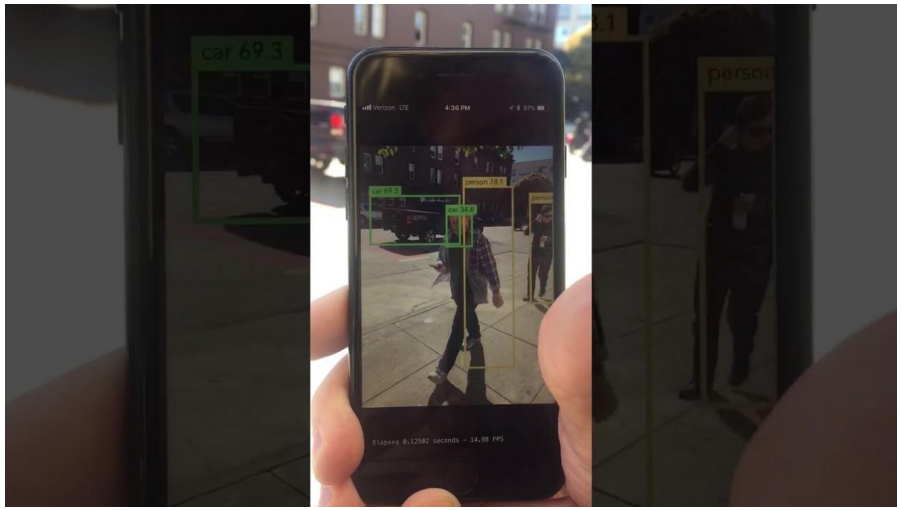
- ☑ Overview
- ☑ Pruning
- ☑ Weight Sharing
- ☑ Quantization
- ☑ Low-rank Approximation
- ☑ Sparse Regularization
- ☑ Distillation
- ➡ ☐ **Conclusion**

Conclusion

- Recent deep learning models are becoming more complex
  - Storage, computation, energy, and deployment problem
- Model compression: make a lightweight model that is fast, memory-efficient, and energy-efficient
  - Pruning, Weight Sharing, Quantization, Low-rank Approximations, Sparse Regularization, Distillation, etc.

Conclusion

- Model compression
  - Key technique to allow using AI everywhere
  - Mitigates energy problem



Thank You!