Model Compression for Large Scale Deep Learning

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

Outline

- **Overview**
- Pruning
- Weight Sharing
- Quantization
- Low-rank Approximation
- Sparse Regularization
- 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 model that is fast, memory-efficient, and energy-efficient
  - Useful not only for edge device, but for servers

- 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

Outline

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

[Han et al. NIPS 2015]

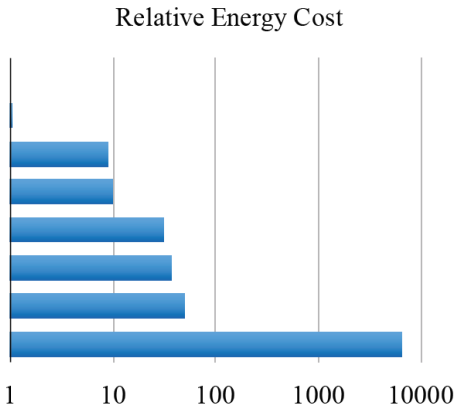
Learning both Weights and Connections for Efficient Neural Networks

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

Input: Deep neural network to train

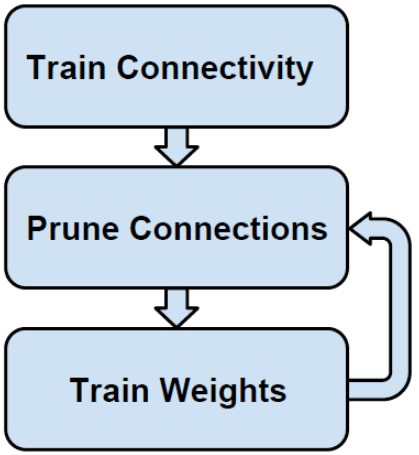
Output: Compressed deep neural network which preserves accuracy

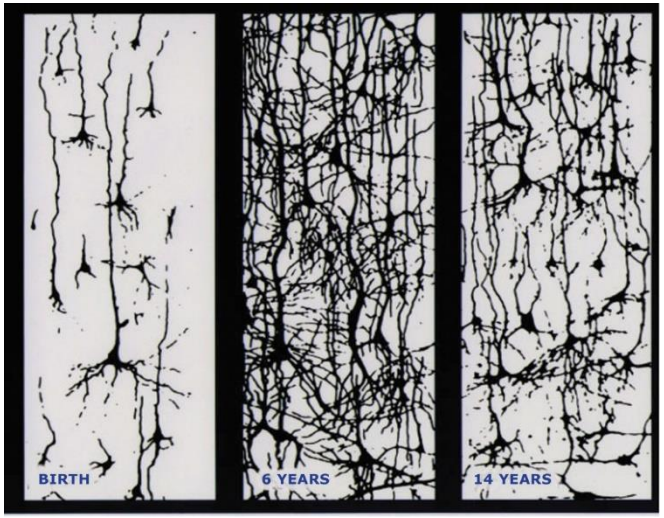
Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

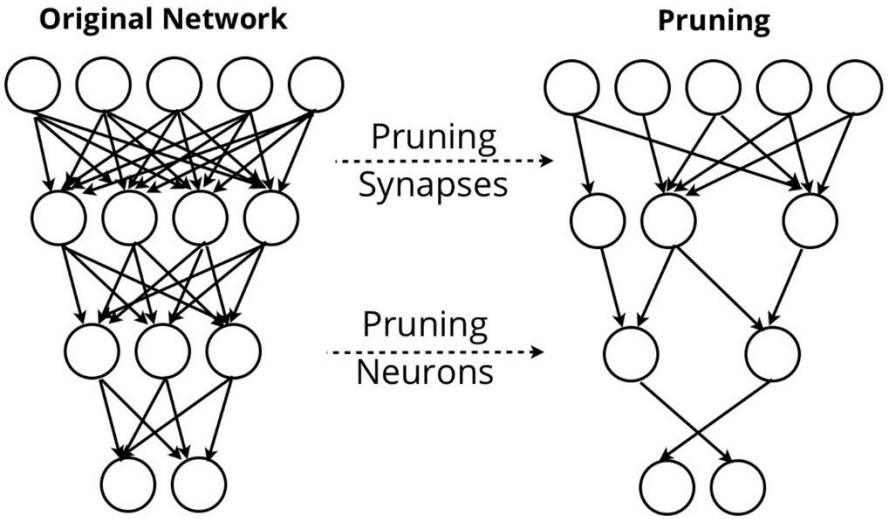


Learning both Weights and Connections for Efficient Neural Networks

- 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

- Choosing regularization
  - L1-normalization: better after pruning, before retraining
  - L2-normalization: best result in iterative pruning
- Adjust dropout ratio
  - Reflect sparsity of neural network in retraining process
  - $D_r = D_o \sqrt{\frac{C_{ir}}{C_{io}}}$

Drop less as network gets sparser

Parameter	Explanation
D_r	Dropout ratio in retraining
D_o	Original dropout ratio
N_i	Number of neurons in ith layer
C_i	Number of connections in ith layer, $C_i = N_i N_{i-1}$
C_{ir}	Number of connections in retraining
C_{io}	Number of connections of original network

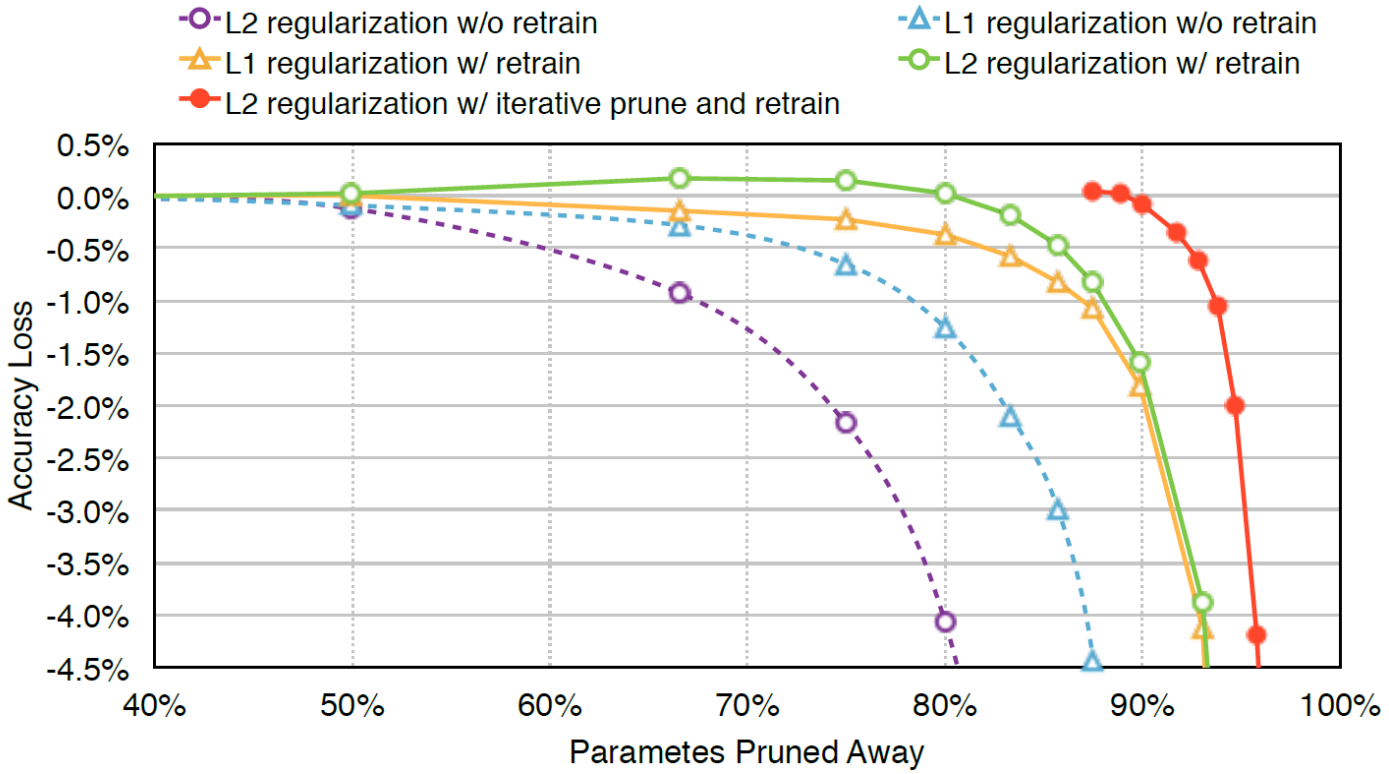
Learning both Weights and Connections for Efficient Neural Networks

- How much we can compress 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**×

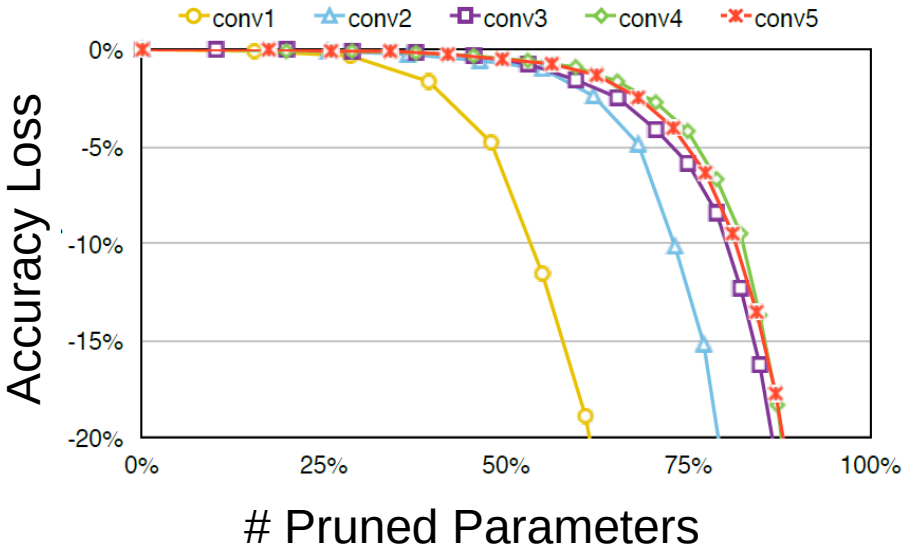
Learning both Weights and Connections for Efficient Neural Networks

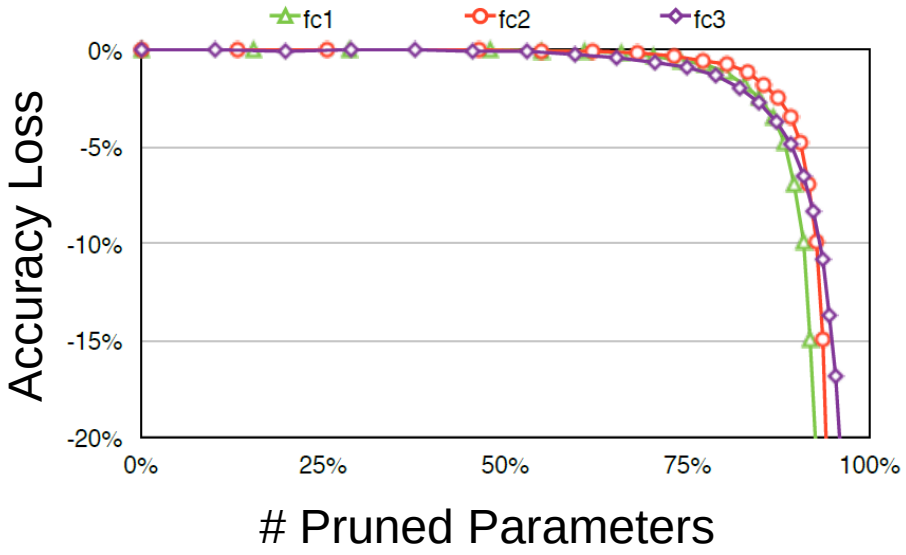
- Between L1 and L2, which regularization is better?
  - L2 is much better in iterative pruning



Learning both Weights and Connections for Efficient Neural Networks

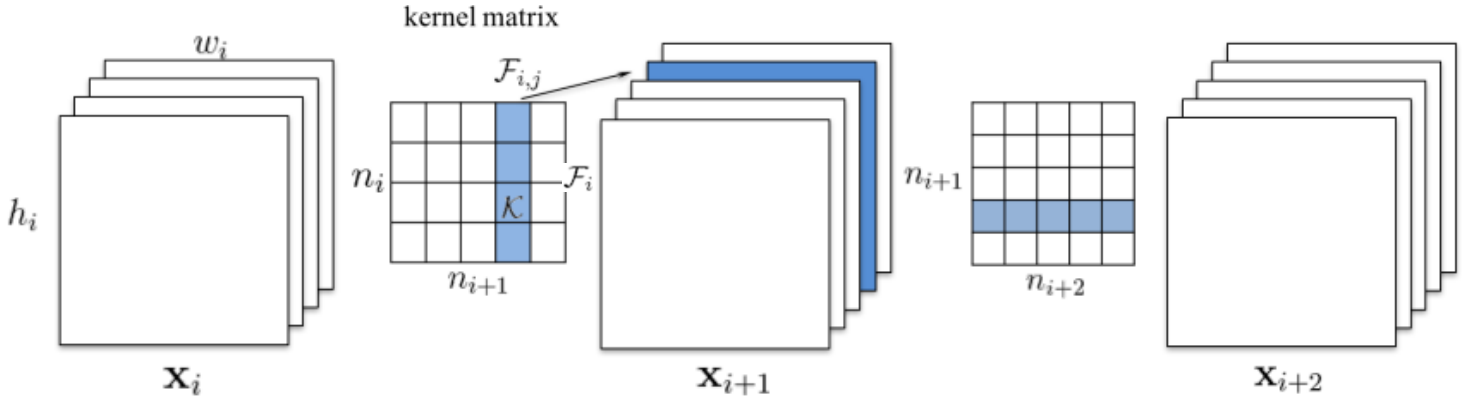
- Between convolutional layer and fully-connected layer, which layer is more sensitive to pruning?
  - Experimented with AlexNet
  - Convolutional layer is more sensitive





[Li et al. ICLR 2017]

Pruning Filters for Efficient ConvNets

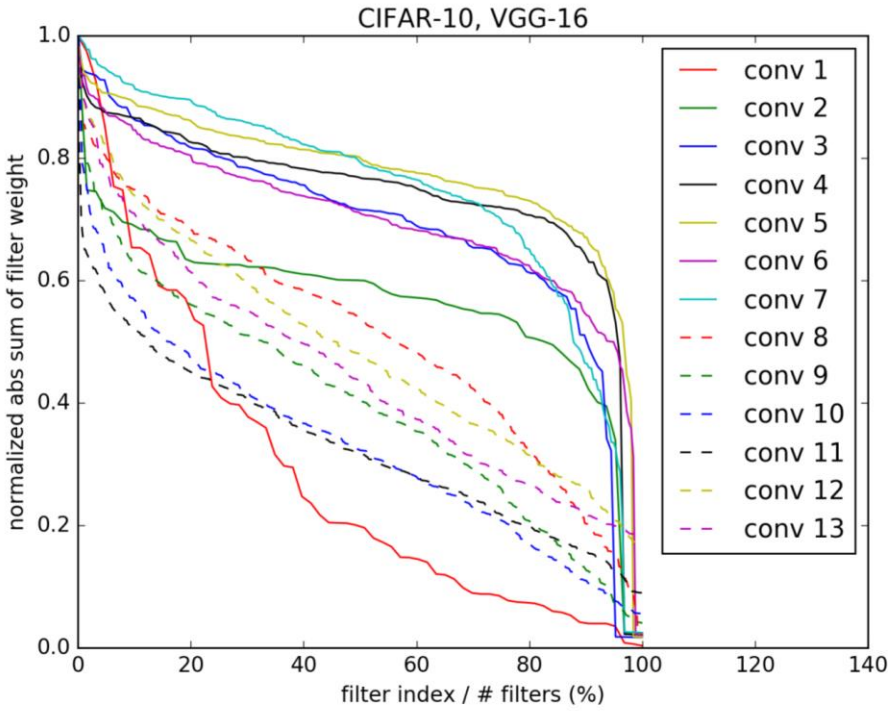
- Given a well-trained CNNs.
- Prune less important filters together with their connecting feature maps in order to reduce computation cost.
- 

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

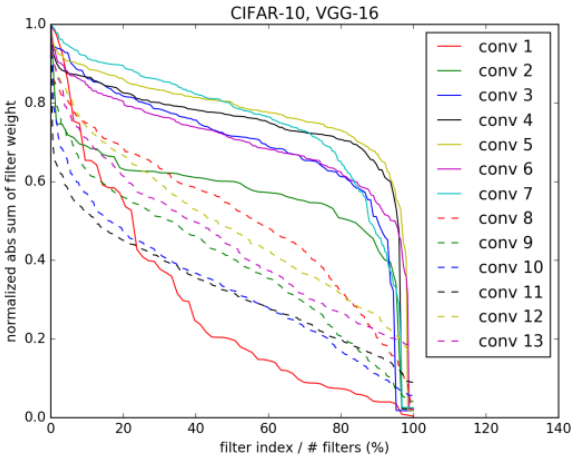
- Importance of filter
  - Defined as the sum of its absolute weights $\sum|F_{i,j}|$, i.e., its l_1-norm $\|F_{i,j}\|_1$.



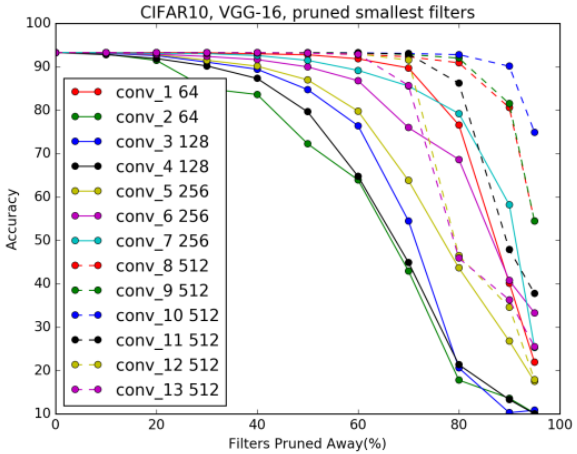
Filters are ranked by s_j

Pruning Filters for Efficient ConvNets

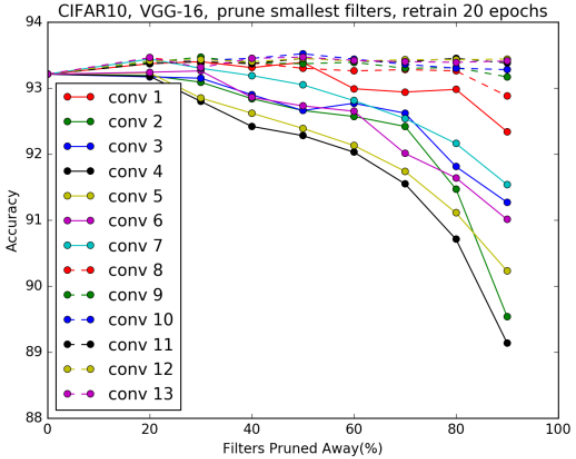
- Sensitivity – how many filters we can prune without affecting the accuracy? – VGG-16 on CIFAR-10
  - Some layers are insensitive and resilient to pruning.
  - Only prune these layers to hold the accuracy.
  - As shown in the following figure, conv1 and conv8-13 are layers to be pruned.



(a) Filters are ranked by s_j



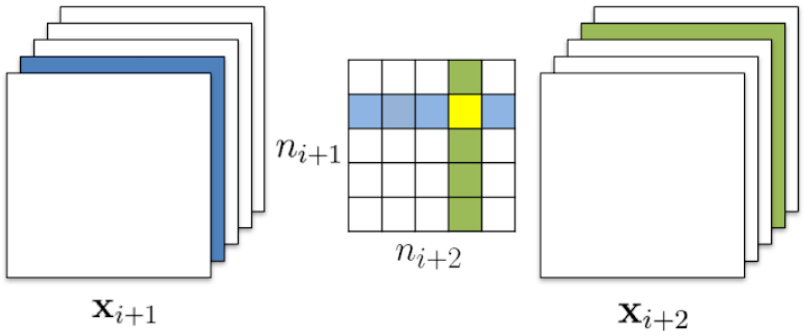
(b) Prune the smallest filters



(c) Prune and retrain

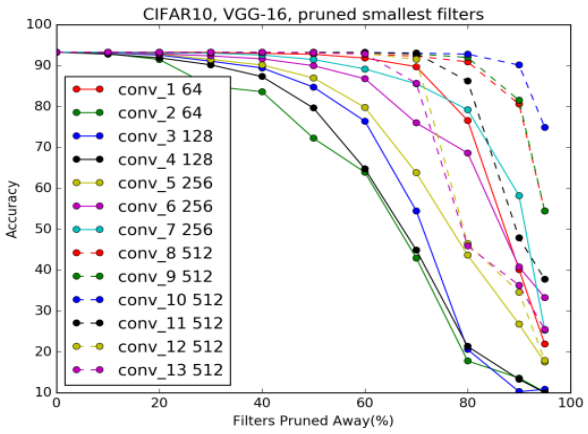
Pruning Filters for Efficient ConvNets

- 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)

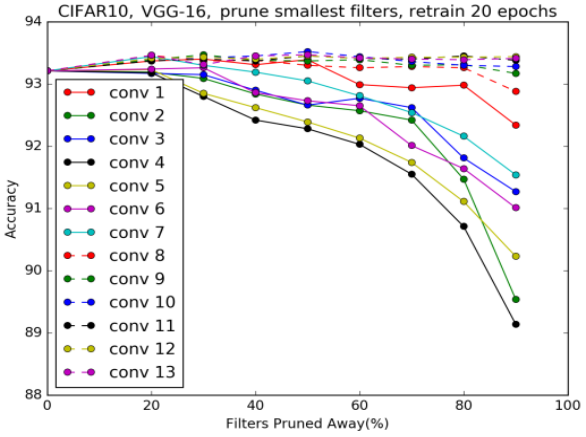


Pruning Filters for Efficient ConvNets

- 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%

Pruning Filters for Efficient ConvNets

- Looking into pruned VGG-16 on CIFAR-10
  - Only insensitive layers are pruned.
  - The method achieves 34% FLOPs reduction in total.

layer type	$w_i \times h_i$	#Maps	FLOP	#Params	#Maps	FLOP%
Conv_1	32×32	64	1.8E+06	1.7E+03	32	50%
Conv_2	32×32	64	3.8E+07	3.7E+04	64	50%
Conv_3	16×16	128	1.9E+07	7.4E+04	128	0%
Conv_4	16×16	128	3.8E+07	1.5E+05	128	0%
Conv_5	8×8	256	1.9E+07	2.9E+05	256	0%
Conv_6	8×8	256	3.8E+07	5.9E+05	256	0%
Conv_7	8×8	256	3.8E+07	5.9E+05	256	0%
Conv_8	4×4	512	1.9E+07	1.2E+06	256	50%
Conv_9	4×4	512	3.8E+07	2.4E+06	256	75%
Conv_10	4×4	512	3.8E+07	2.4E+06	256	75%
Conv_11	2×2	512	9.4E+06	2.4E+06	256	75%
Conv_12	2×2	512	9.4E+06	2.4E+06	256	75%
Conv_13	2×2	512	9.4E+06	2.4E+06	256	75%
Linear	1	512	2.6E+05	2.6E+05	512	50%
Linear	1	10	5.1E+03	5.1E+03	10	0%
Total			3.1E+08	1.5E+07		34%

Outline

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

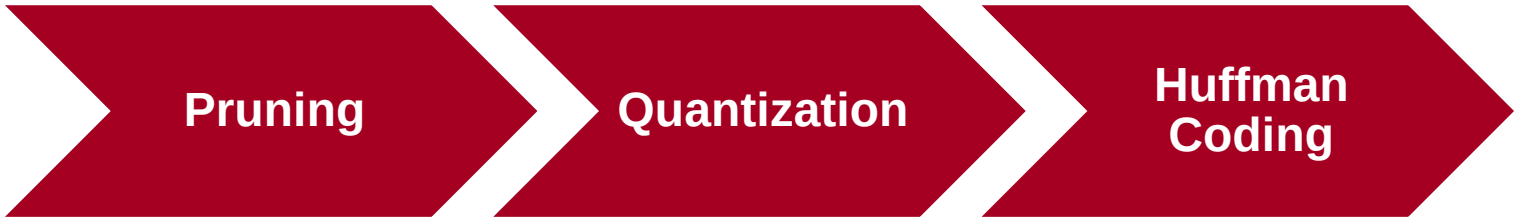
[Han et al. ICLR 2016]

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

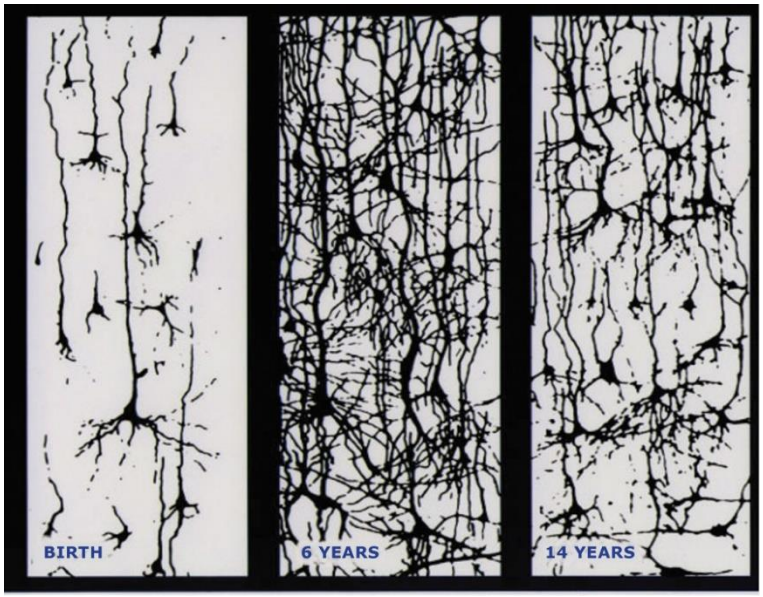
Input: Deep neural network

Output: Compressed deep neural network without accuracy loss

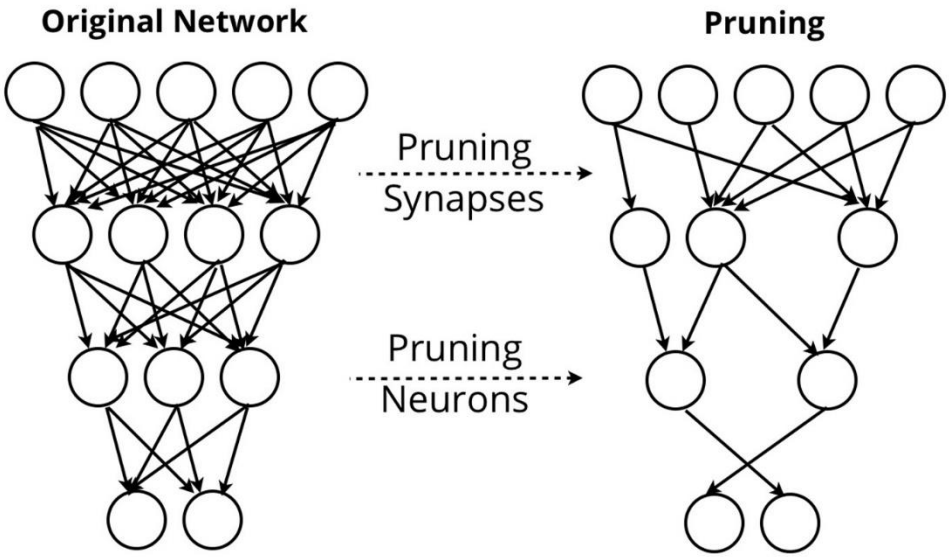


Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

- Pruning redundant weights
  - Intuition from human brain
  - **Remove** weights if $|weight| < threshold$
  - **Retrain** network with remaining weights

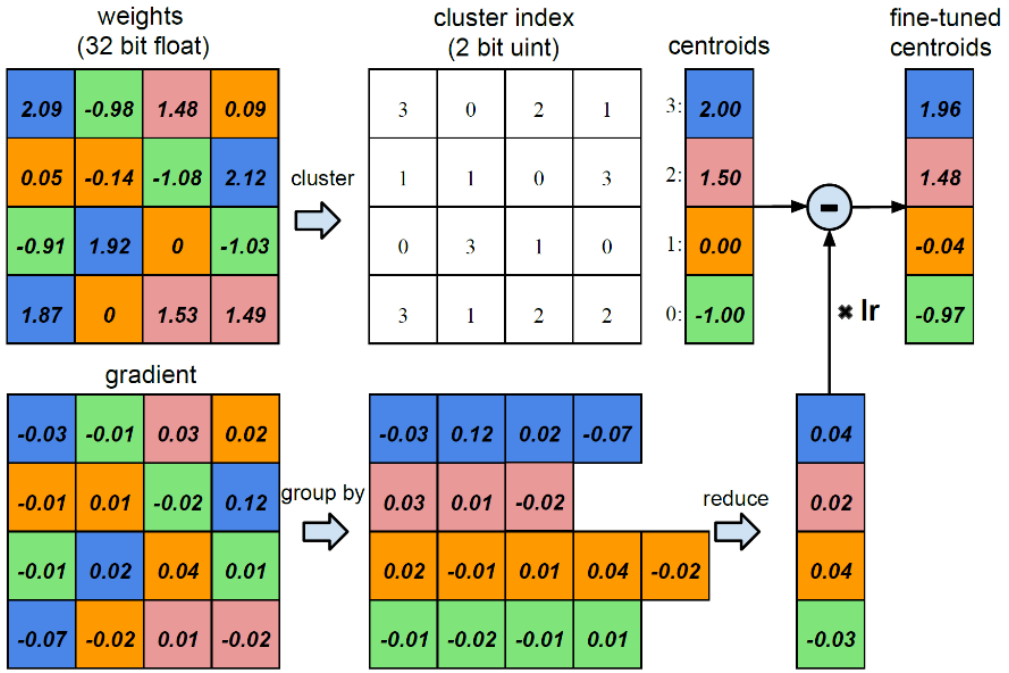


[Synaptic Pruning of Humans]



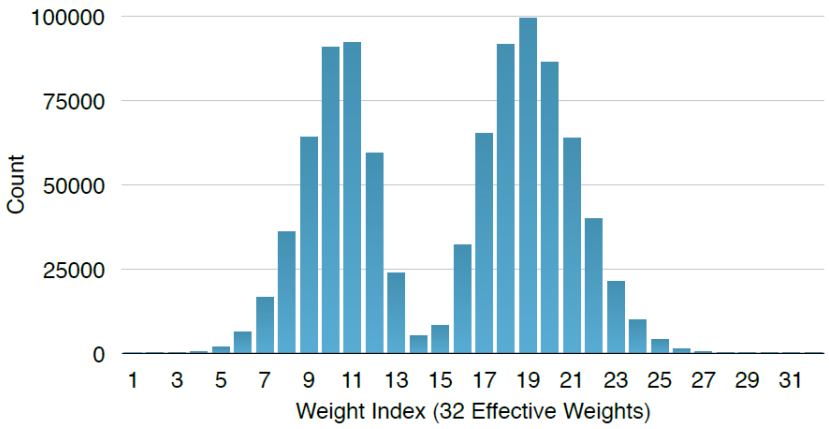
Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

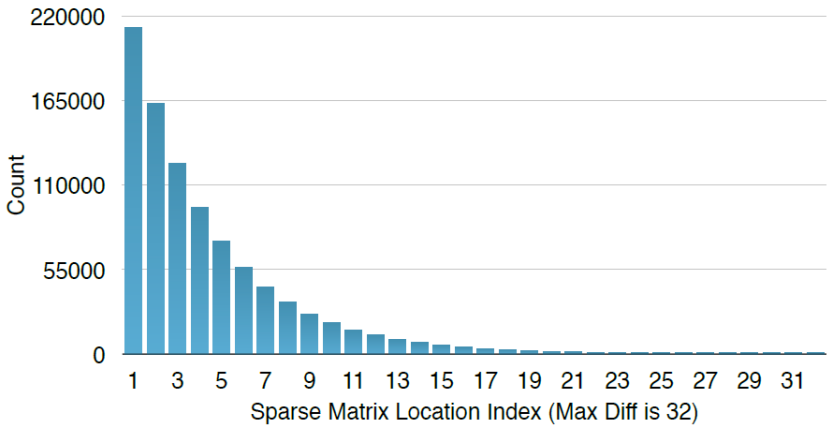
- Quantization and Weight Sharing
  - Cluster weights and use **centroid** of clustered weights in indexed array
  - Update centroid weights with **summation** of corresponding gradients



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

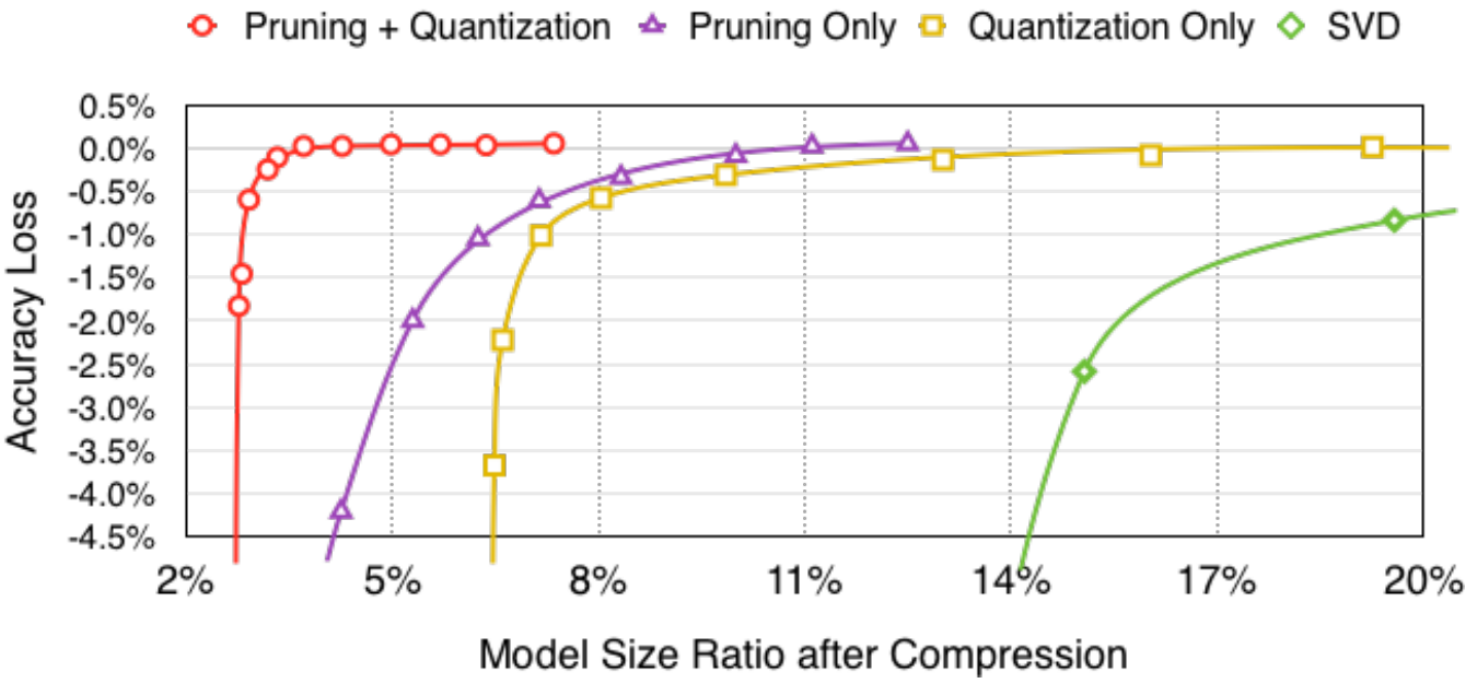
Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding

- 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



Outline

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

[Courbariaux et al. NIPS 2015]

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- Given a Deep Neural Network (DNN),
- Quantize the networks for better memory efficiency, computation efficiency, and power consumption

[NIPS 2015]

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- 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

- Propagations & Updates
  - w is binarized during forward and backward propagation.
  - Full-precision w is used during parameter update.

Algorithm 1 SGD training with BinaryConnect. C is the cost function for minibatch and the functions binarize(w) and clip(w) specify how to binarize and clip weights. L is the number of layers.

Require: a minibatch of (inputs, targets), previous parameters w_{t-1} (weights) and b_{t-1} (biases), and learning rate η.

Ensure: updated parameters w_t and b_t.

1. Forward propagation:

$w_b \leftarrow \text{binarize}(w_{t-1})$

For $k = 1$ to L, compute a_k knowing a_{k-1}, w_b and b_{t-1}

2. Backward propagation:

Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$

For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and w_b

3. Parameter update:

Compute $\frac{\partial C}{\partial w_b}$ and $\frac{\partial C}{db_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}

$w_t \leftarrow \text{clip}(w_{t-1} - \eta \frac{\partial C}{\partial w_b})$

$b_t \leftarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

BinaryConnect - Training Deep Neural Networks with Binary Weights during Propagations

- Performance and comparation 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 and lower the validation error rate – similar to Dropout scheme
  - Dashed line: training cost
  - Solid line: validation error rate

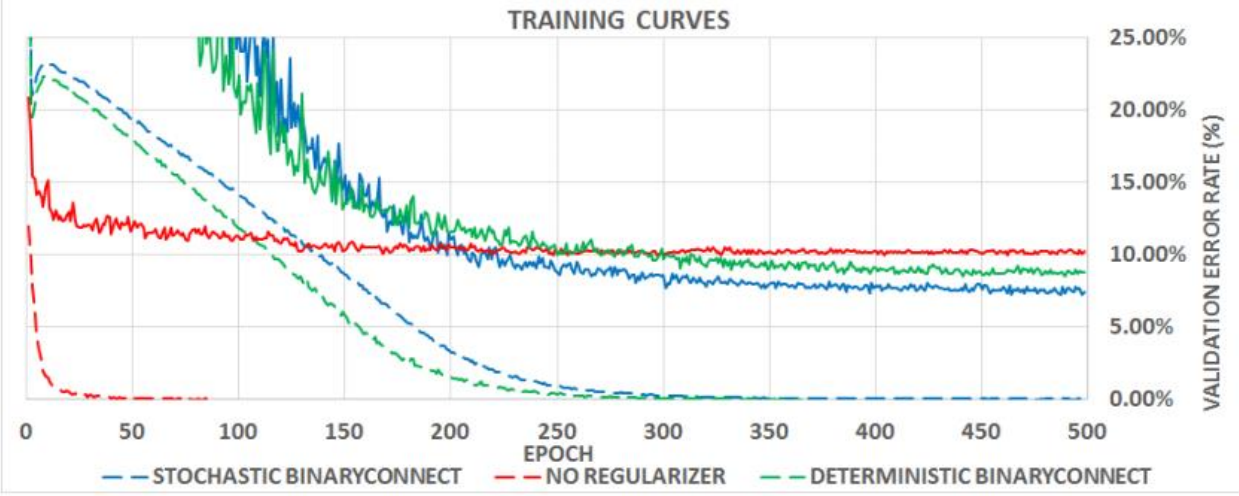
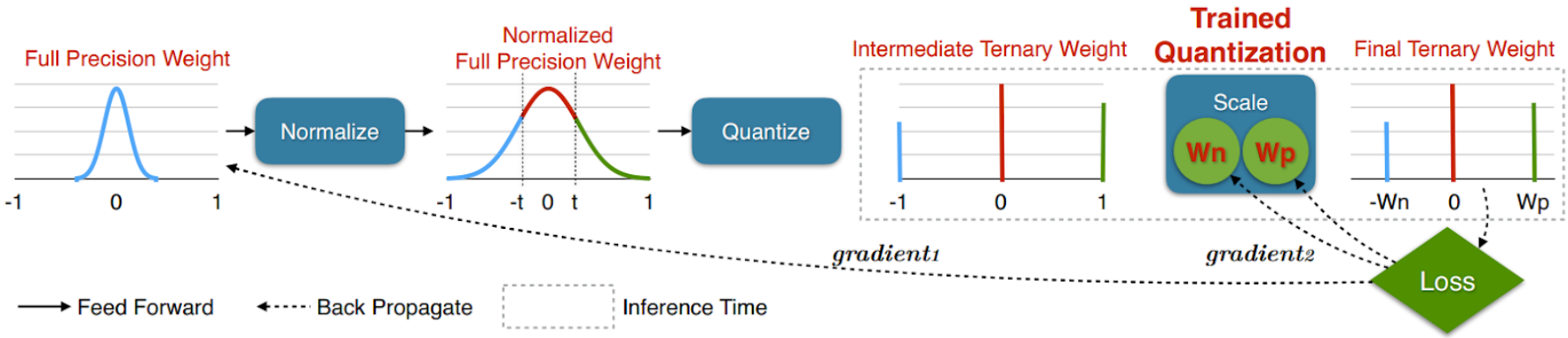


Figure 3: Training curves of a CNN on CIFAR-10 depending on the regularizer. The dotted lines represent the training costs (square hinge losses) and the continuous lines the corresponding validation error rates. Both versions of BinaryConnect significantly augment the training cost, slow down the training 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

Trained Ternary Quantization

- Comparation
  - TTQ has substantial accuracy improvement over other quantization method.
  - The loss of accuracy over the full precision model is small and in many case it even improves the accuracy.

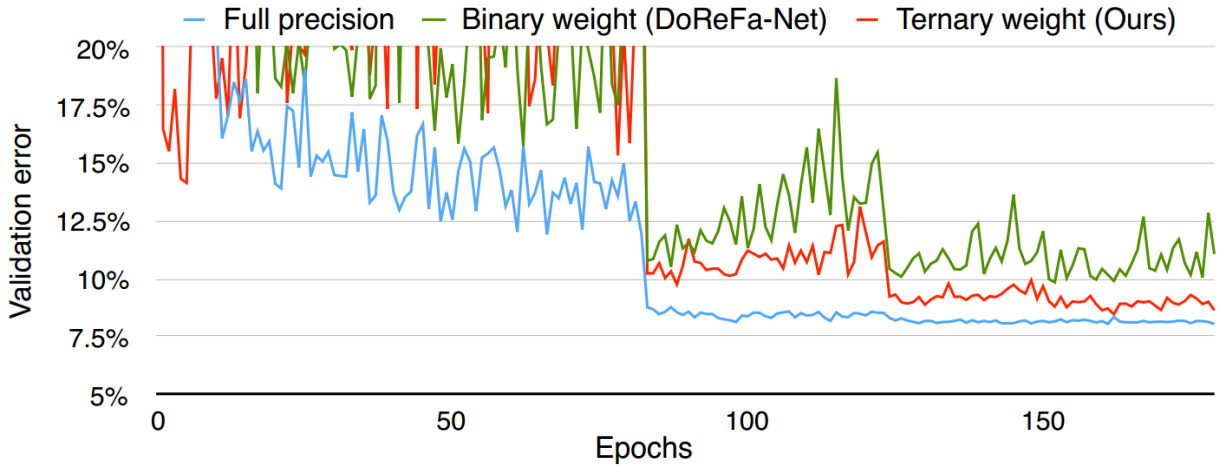


Figure 3: ResNet-20 on CIFAR-10 with different weight precision.

Error	Full precision	1-bit (DoReFa)	2-bit (TWN)	2-bit (Ours)
Top1	42.8%	46.1%	45.5%	**42.5%**
Top5	19.7%	23.7%	23.2%	**20.3%**

Table 2: Top1 and Top5 error rate of AlexNet on ImageNet

Error	Full precision	1-bit (BWN)	2-bit (TWN)	2-bit (Ours)
Top1	30.4%	39.2%	34.7%	**33.4%**
Top5	10.8%	17.0%	13.8%	**12.8%**

Table 3: Top1 and Top5 error rate of ResNet-18 on ImageNet

Outline

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

[Howard et al.]

MobileNets

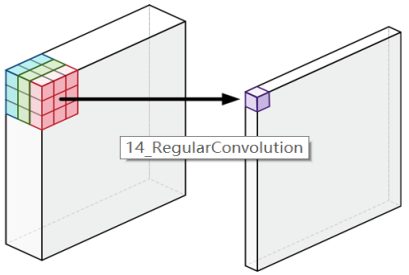
- Given a CNN,
- Design a low-latency and memory efficient model that can be easily matched to the design requirements for mobile and embedded vision applications.

MobileNets

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

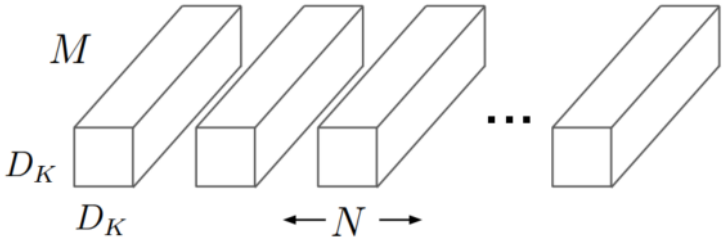
- Main Idea
  - Factorize convolutional layer into 2 layers: depthwise convolutional layer and pointwise convolutional layer(1×1 convolutional layer) in order to reduce model size and computational cost.

MobileNets

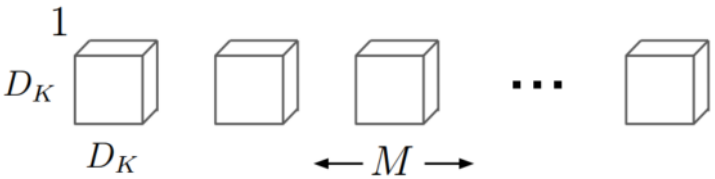


standard

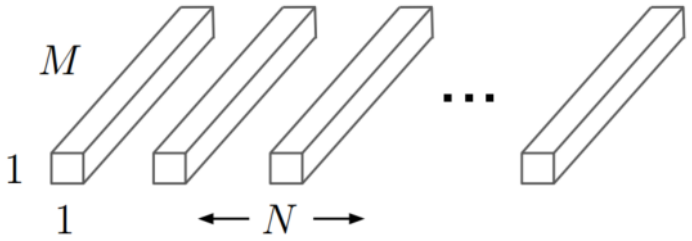
- The standard convolution both filters and combines inputs into a new set of output in one step.
- The depthwise separable convolution splits standard convolution into 2 layers:
  1. One for filtering.
  2. Another one for combining.



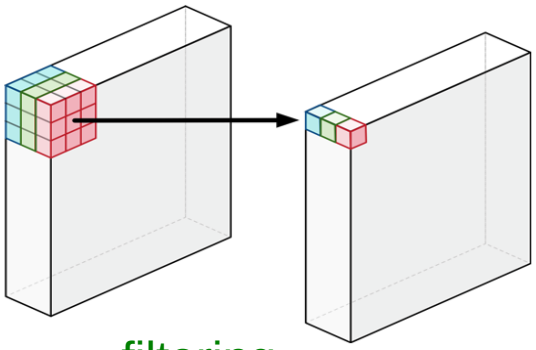
(a) Standard Convolution Filters



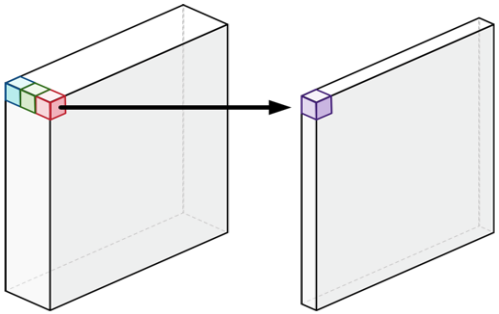
(b) Depthwise Convolutional Filters



(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

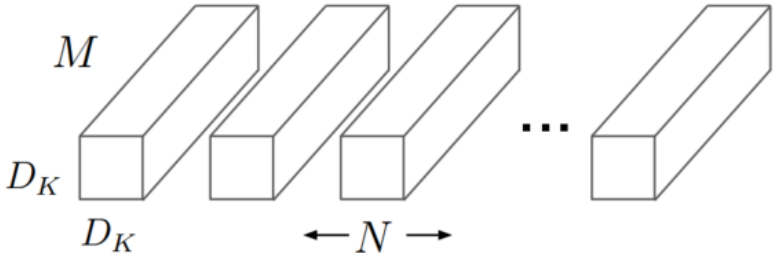
- Computational Cost
  - Standard

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

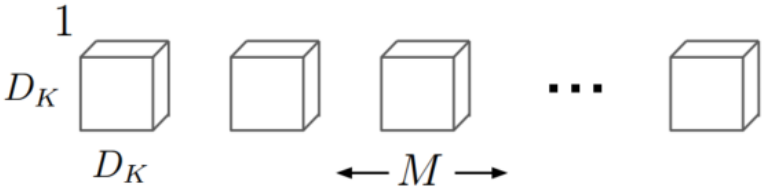
    $D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$
  - Pointwise Convolution

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

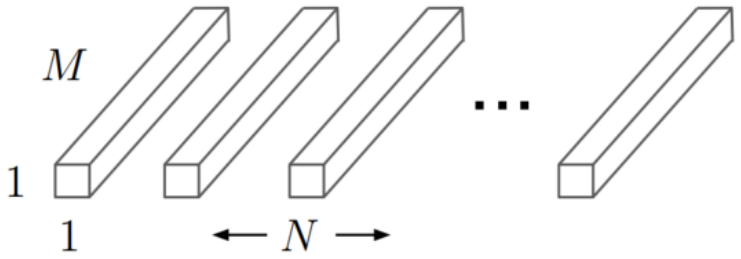
$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$



(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

- Architecture

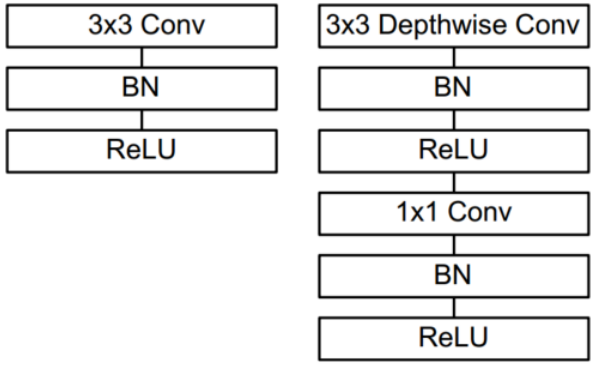


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

Table 2. Resource Per Layer Type

Type	Mult-Adds	Parameters
Conv 1×1	94.86%	74.59%
Conv DW 3×3	3.06%	1.06%
Conv 3×3	1.19%	0.02%
Fully Connected	0.18%	24.33%

Most Mult-Adds are in Conv 1×1.
Most parameters are in Conv 1×1 and FC

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

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 deep 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**
- ☐ 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)

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

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

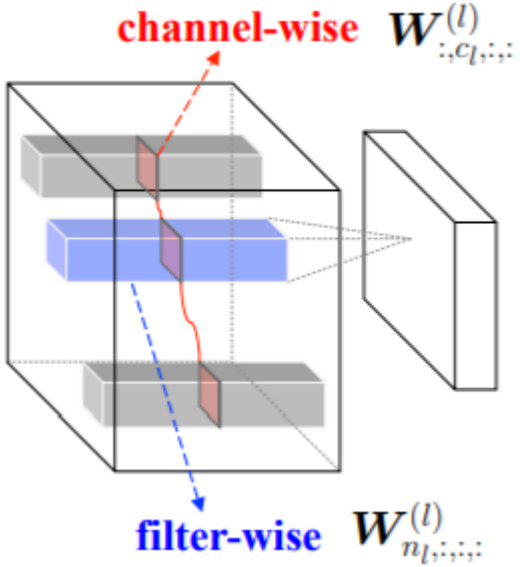
Learning Structured Sparsity in Deep Neural Networks

- 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

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



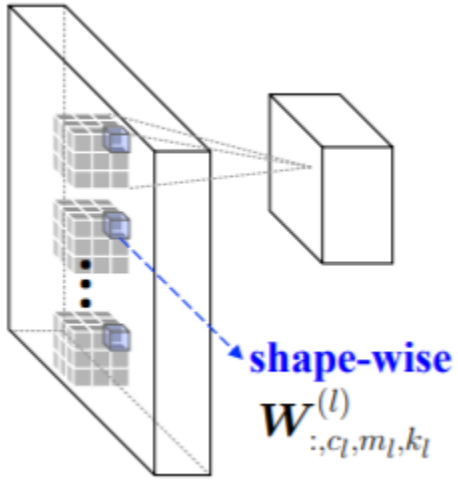
Learning Structured Sparsity in Deep Neural Networks

- 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

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



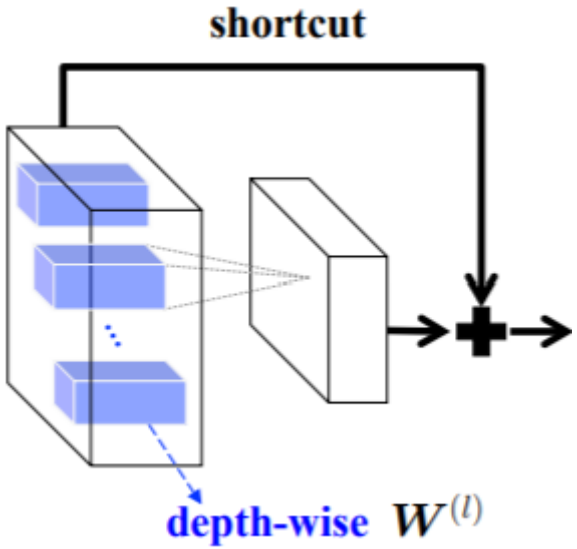
Learning Structured Sparsity in Deep Neural Networks

- 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

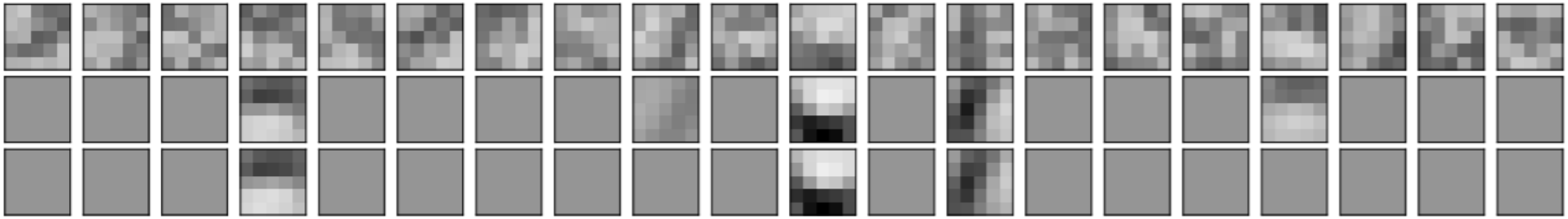


Figure 3: Learned *conv1* filters in *LeNet 1* (top), *LeNet 2* (middle) and *LeNet 3* (bottom)

Learning Structured Sparsity in Deep Neural Networks

- ConvNet on CIFAR-10
  - 3x speedup with the same error on ConvNet

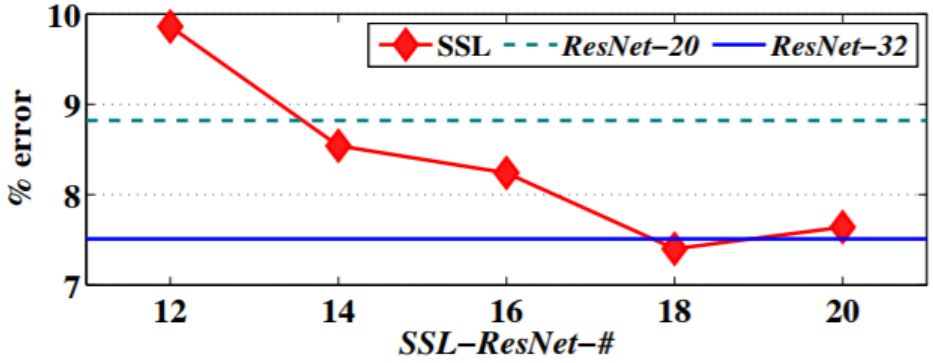
Table 3: Learning row-wise and column-wise sparsity of *ConvNet* on CIFAR-10

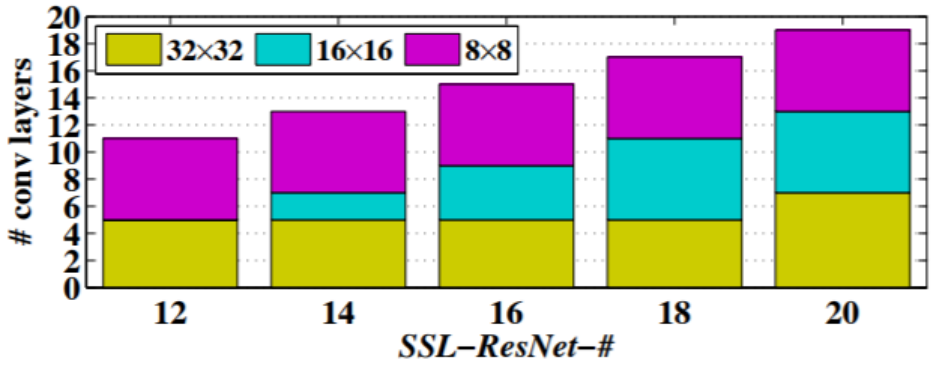
ConvNet #	Error	Row sparsity §	Column sparsity §	Speedup §
1 (*baseline*)	17.9%	12.5%–0%–0%	0%–0%–0%	1.00×–1.00×–1.00×
2	17.9%	50.0%–28.1%–1.6%	0%–59.3%–35.1%	1.43×–3.05×–1.57×
3	16.9%	31.3%–0%–1.6%	0%–42.8%–9.8%	1.25×–2.01×–1.18×

§ in the order of *conv1–conv2–conv3*

Learning Structured Sparsity in Deep Neural Networks

- ResNet on CIFAR-10
  - SSL-ResNet-20 (proposed) outperforms ResNet-20 in accuracy





Outline

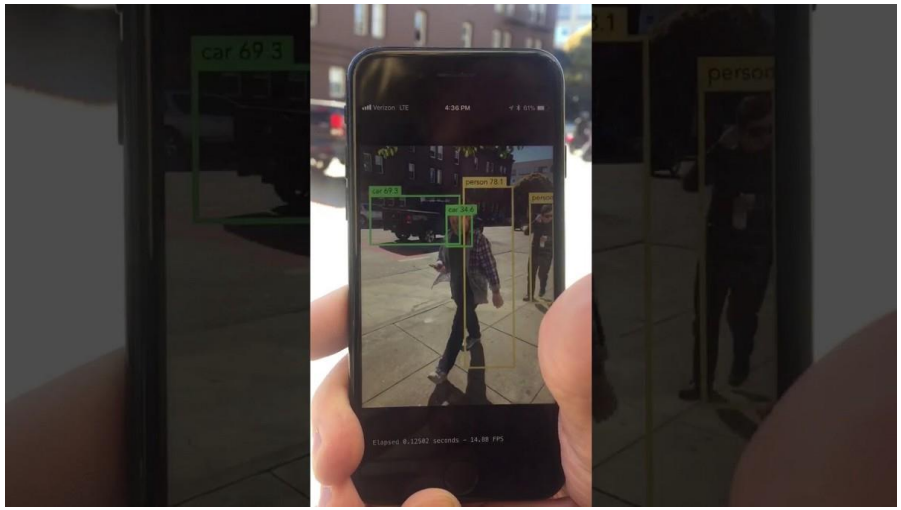
- ☑ Overview
- ☑ Pruning
- ☑ Weight Sharing
- ☑ Quantization
- ☑ Low-rank Approximation
- ☑ Sparse Regularization
- ➡ ☐ **Conclusion**

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

Conclusion

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



Thank You!

http://datalab.snu.ac.kr/