

Week 17: Efficient deep learning

Instructor: Ruixuan Wang
wangruix5@mail.sysu.edu.cn

School of Data and Computer Science
Sun Yat-Sen University

20 June, 2019

- 1 Factorization
- 2 Knowledge transfer
- 3 Pruning
- 4 Quantization
- 5 New model designs

Motivation

- Real-time processing in applications like self-driving
- Memory and battery is limited in devices like mobile phone

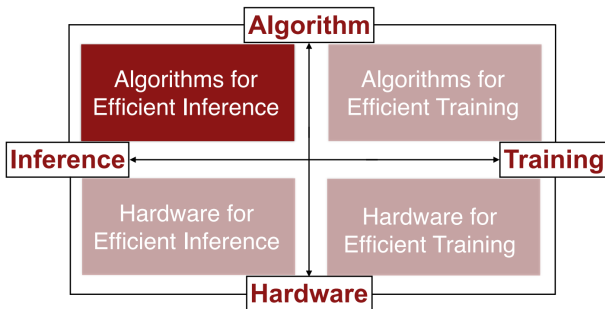
Motivation

- Real-time processing in applications like self-driving
- Memory and battery is limited in devices like mobile phone

Need smaller model and fast computation!

Model efficiency

- Multiple ways to improve efficiency in computation & memory
- We mainly focus on ‘algorithms for efficient inference’



Note: refer to Stanford CS231n Lecture 15 (2017) for other parts!

Figures in next 2 slides from Zhang et al., "Accelerating Very Deep Convolutional Networks for Classification and Detection", arXiv, 2015; Lebedev et al., "Speeding-up convolutional neural networks using fine-tuned CP-decomposition", ICLR, 2015

Factorization: low-rank matrix decomposition

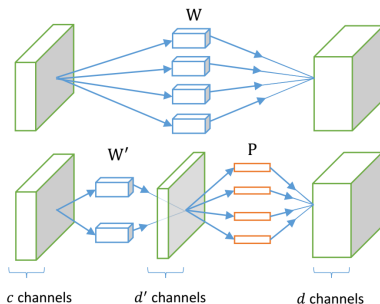
- Unfold d kernels of size $k \times k \times c$ into matrix \mathbf{W} of size $d \times (k^2c + 1)$; note '1' for bias parameter per kernel

Factorization: low-rank matrix decomposition

- Unfold d kernels of size $k \times k \times c$ into matrix \mathbf{W} of size $d \times (k^2c + 1)$; note '1' for bias parameter per kernel
- Low-rank decomposition $\mathbf{W} = \mathbf{P}\mathbf{W}'$, where \mathbf{P} is $d \times d'$ and \mathbf{W}' is $d' \times (k^2c + 1)$

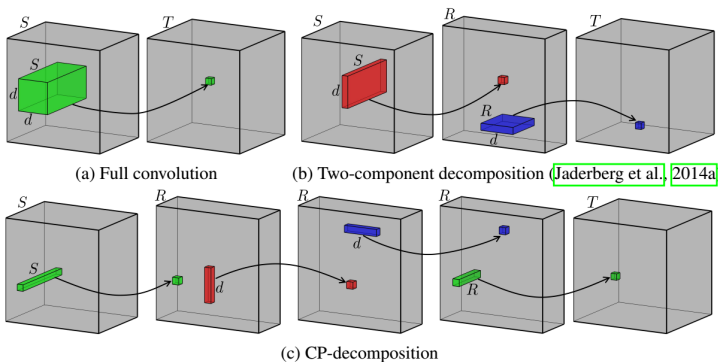
Factorization: low-rank matrix decomposition

- Unfold d kernels of size $k \times k \times c$ into matrix \mathbf{W} of size $d \times (k^2c + 1)$; note '1' for bias parameter per kernel
- Low-rank decomposition $\mathbf{W} = \mathbf{P}\mathbf{W}'$, where \mathbf{P} is $d \times d'$ and \mathbf{W}' is $d' \times (k^2c + 1)$
- i.e., decomposed into one layer with d' kernels of size $k \times k \times c$, and 2^{nd} layer with d kernels of size $1 \times 1 \times d'$



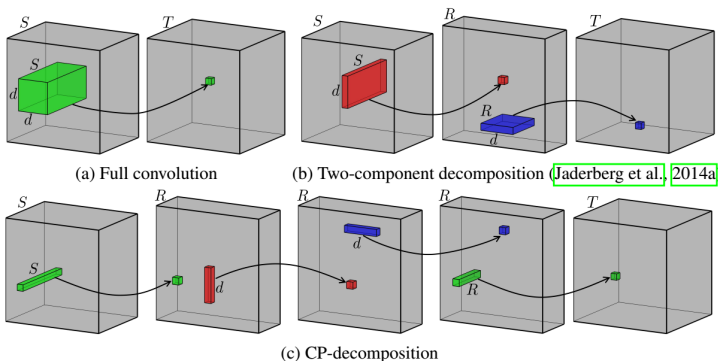
Factorization: low-rank tensor decomposition

- Jaderberg et al.: decompose T kernels of size $d \times d \times S$ into R kernels of size $d \times 1 \times S$ and T kernels of size $1 \times d \times R$



Factorization: low-rank tensor decomposition

- Jaderberg et al.: decompose T kernels of size $d \times d \times S$ into R kernels of size $d \times 1 \times S$ and T kernels of size $1 \times d \times R$
- canonical polyadic (CP) decomposition: decomposed 4D tensor of size $d \times d \times S \times T$ into R kernels of size $1 \times 1 \times S$, of size $d \times 1 \times 1$, of $1 \times d \times 1$, and T kernels of size $1 \times 1 \times R$



Factorization limitation

Factorization focuses on efficient computation!

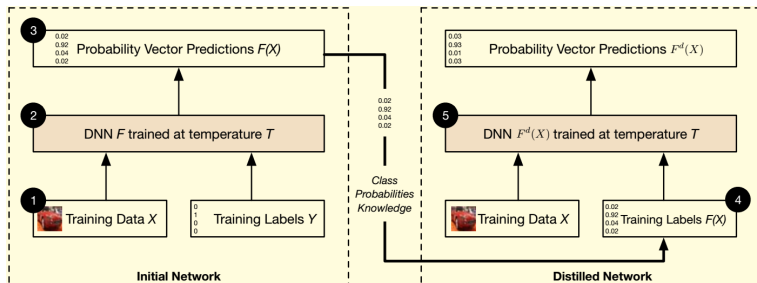
How to make model smaller?

Distillation: transfer knowledge from one model to another

- Use a pre-trained large network or ensemble of networks to teach a smaller one, both with softened softmax

$$\text{softmax}(x, T)_i = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}}$$

- Soft label for teaching; smaller $T = 1$ for inference



Distillation network

- Training with soft labels generalizes well with 3% data (last row), while training with hard labels not (second row)
- Soft labels encode knowledge (e.g., similarity) across classes

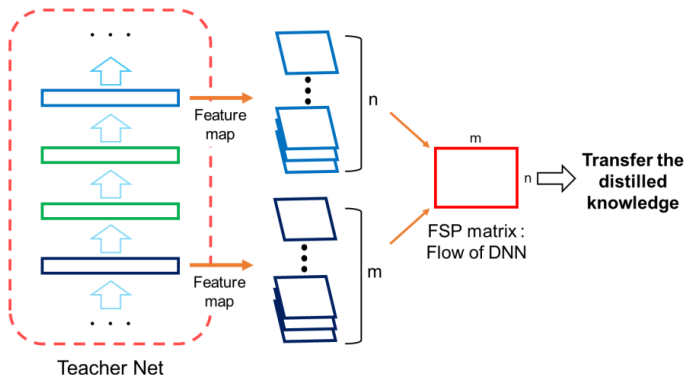
System & training set	Train Frame Accuracy	Test Frame Accuracy
Baseline (100% of training set)	63.4%	58.9%
Baseline (3% of training set)	67.3%	44.5%
Soft Targets (3% of training set)	65.4%	57.0%

- 'Knowledge' could be transferred via other layers

Transfer mid-level information

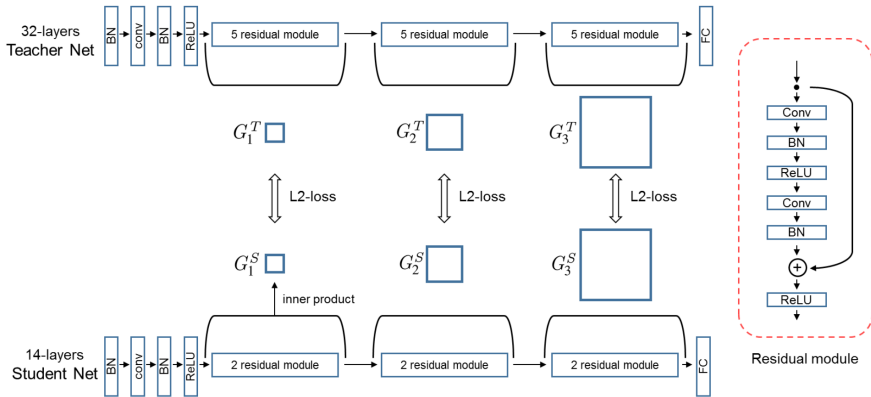
- Gram matrix \mathbf{G} across layers captures 'flow between layers'

$$G_{i,j}(x; W) = \sum_{s=1}^h \sum_{t=1}^w \frac{F_{s,t,i}^1(x; W) \times F_{s,t,j}^2(x; W)}{h \times w}$$



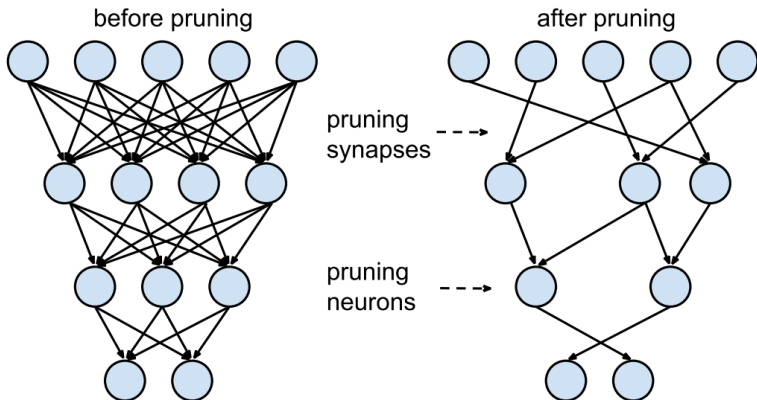
Transfer mid-level information

- Train student network by min $L2$ loss in gram matrices
- Then fine-tune student network with task-specific loss



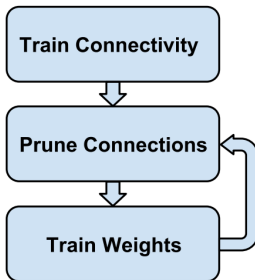
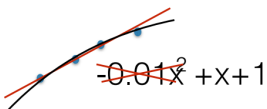
Network pruning

- Remove insignificant connections (in fully connected layers)
- Similar idea can be used to remove kernels in conv layers



How to prune

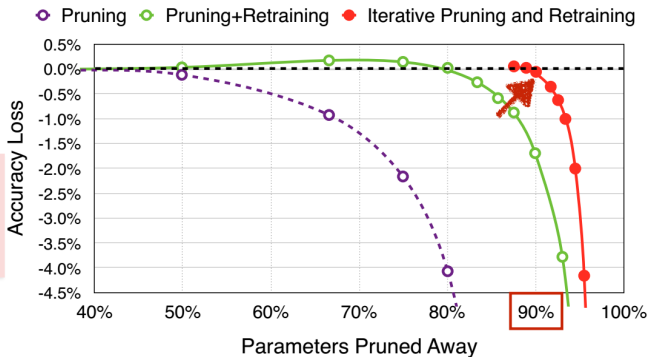
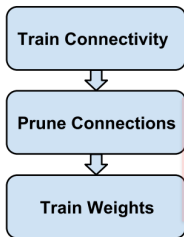
- Prune weights (connections) with small values
- Iteratively retrain model after pruning



60 Million
6M 10x less connections

Effect of retraining in pruning

- Retraining can largely recover accuracy



- Reduce model size, but may not accelerate test computation

Channel pruning: another way to prune

- Idea: prune insignificant feature channels at each layer!

Channel pruning: another way to prune

- Idea: prune insignificant feature channels at each layer!
- With a scaling factor γ for each channel, add regularization term $g(\gamma)$ to loss:

$$L = \sum_{(x,y)} l(f(x, W), y) + \lambda \sum_{\gamma \in \Gamma} g(\gamma)$$

where $g(\gamma) = |\gamma|$ forces γ close to zero!

Channel pruning: another way to prune

- Idea: prune insignificant feature channels at each layer!
- With a scaling factor γ for each channel, add regularization term $g(\gamma)$ to loss:

$$L = \sum_{(x,y)} l(f(x, W), y) + \lambda \sum_{\gamma \in \Gamma} g(\gamma)$$

where $g(\gamma) = |\gamma|$ forces γ close to zero!

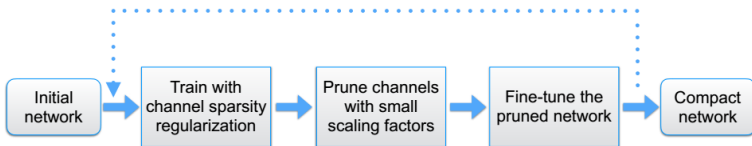
- γ is the parameter in Batch Normalization, one per channel

$$\hat{z} = \frac{z_{in} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}; \quad z_{out} = \gamma \hat{z} + \beta$$

- If γ is introduced elsewhere in network, its effect would be cancelled by BN or by expanding weight values

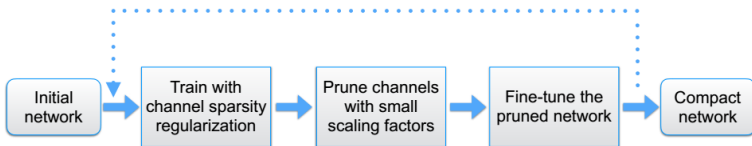
Channel pruning (cont')

- Prune channels with near-zero scaling γ
- Repeat a few times: fine-tune after pruning channels

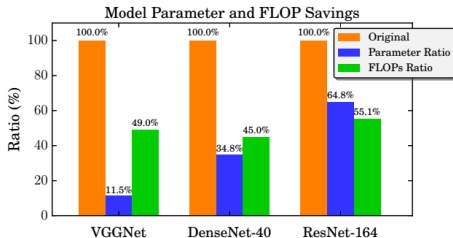


Channel pruning (cont')

- Prune channels with near-zero scaling γ
- Repeat a few times: fine-tune after pruning channels



- Thinner models, less computation (and running-memory)



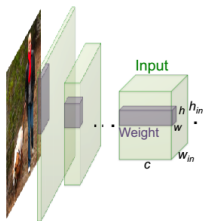
So far...

Above: start from a large model! Reduce layers, kernels, kernel sizes, connections!

Another idea: not reduce but quantize variables!

XNOR-net

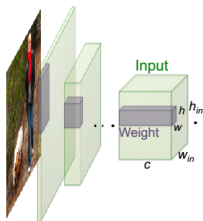
- Binary-weight-network (2^{nd} row): filter weights binarized



	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	<p>Real-Value Inputs</p> <p>Real-Value Weights</p>	+ , - , ×	1x	1x	%56.7
Binary Weight	<p>Real-Value Inputs</p> <p>Binary Weights</p>	+ , -	~32x	~2x	%56.8
BinaryWeight Binary Input (XNOR-Net)	<p>Binary Inputs</p> <p>Binary Weights</p>	XNOR , bitcount	~32x	~58x	%44.2

XNOR-net

- Binary-weight-network (2^{nd} row): filter weights binarized
- XNOR-net (3^{rd} row): layer input and filter binarized
- Convolution between binary input and binary filter can be computed by XNOR and bitcounting operations



	Network Variations	Operations used in Convolution	Memory Saving (Inference)	Computation Saving (Inference)	Accuracy on ImageNet (AlexNet)
Standard Convolution	<p>Real-Value Inputs</p> <p>Real-Value Weights</p>	$+, -, \times$	1x	1x	%56.7
Binary Weight	<p>Real-Value Inputs</p> <p>Binary Weights</p>	$+, -$	$\sim 32x$	$\sim 2x$	%56.8
BinaryWeight Binary Input (XNOR-Net)	<p>Binary Inputs</p> <p>Binary Weights</p>	XNOR, bitcount	$\sim 32x$	$\sim 58x$	%44.2

Binarization of filters

- Approximate filter \mathbf{W} with binary $\mathbf{B} \in \{+1, -1\}^{d \times d \times c}$ and a scaling factor α , such that $\mathbf{W} \approx \alpha \mathbf{B}$

$$J(\mathbf{B}, \alpha) = \|\mathbf{W} - \alpha \mathbf{B}\|^2$$

$$\alpha^*, \mathbf{B}^* = \underset{\alpha, \mathbf{B}}{\operatorname{argmin}} J(\mathbf{B}, \alpha)$$

Binarization of filters

- Approximate filter \mathbf{W} with binary $\mathbf{B} \in \{+1, -1\}^{d \times d \times c}$ and a scaling factor α , such that $\mathbf{W} \approx \alpha \mathbf{B}$

$$J(\mathbf{B}, \alpha) = \|\mathbf{W} - \alpha \mathbf{B}\|^2$$
$$\alpha^*, \mathbf{B}^* = \underset{\alpha, \mathbf{B}}{\operatorname{argmin}} J(\mathbf{B}, \alpha)$$

- The solution

$$\mathbf{B}^* = \operatorname{sign}(\mathbf{W}) \quad \alpha^* = \frac{1}{n} \|\mathbf{W}\|_{\ell_1}$$

- Train: use binary filters for feedforward pass and gradient computation, but update parameters on real-valued filters

Training CNN with scaled binary filters

- Parameter change is tiny, so update on real-valued weights

Algorithm 1 Training an L -layers CNN with binary weights:

Input: A minibatch of inputs and targets (\mathbf{I} , \mathbf{Y}), cost function $C(\mathbf{Y}, \hat{\mathbf{Y}})$, current weight \mathcal{W}^t and current learning rate η^t .

Output: updated weight \mathcal{W}^{t+1} and updated learning rate η^{t+1} .

- 1: Binarizing weight filters:
 - 2: **for** $l = 1$ to L **do**
 - 3: **for** k^{th} filter in l^{th} layer **do**
 - 4: $\mathcal{A}_{lk} = \frac{1}{n} \|\mathcal{W}_{lk}^t\|_{\ell_1}$
 - 5: $\mathcal{B}_{lk} = \text{sign}(\mathcal{W}_{lk}^t)$
 - 6: $\widetilde{\mathcal{W}}_{lk} = \mathcal{A}_{lk} \mathcal{B}_{lk}$
 - 7: $\hat{\mathbf{Y}} = \mathbf{BinaryForward}(\mathbf{I}, \mathcal{B}, \mathcal{A})$ // standard forward propagation except that convolutions are computed using equation 1 or 11
 - 8: $\frac{\partial C}{\partial \mathcal{W}} = \mathbf{BinaryBackward}(\frac{\partial C}{\partial \hat{\mathbf{Y}}}, \widetilde{\mathcal{W}})$ // standard backward propagation except that gradients are computed using $\widetilde{\mathcal{W}}$ instead of \mathcal{W}^t Update non-binary weights
 - 9: $\mathcal{W}^{t+1} = \mathbf{UpdateParameters}(\mathcal{W}^t, \frac{\partial C}{\partial \mathcal{W}}, \eta^t)$ // Any update rules (e.g., SGD or ADAM)
 - 10: $\eta^{t+1} = \mathbf{UpdateLearningrate}(\eta^t, t)$ // Any learning rate scheduling function
-

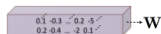
XNOR-net

- Similarly for part of layer input: $\mathbf{X} \approx \beta \text{sign}(\mathbf{X})$, such that

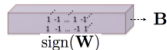
$$\mathbf{I} * \mathbf{W} \approx (\text{sign}(\mathbf{I}) \circledast \text{sign}(\mathbf{W})) \odot \mathbf{K} \alpha$$

\circledast : convolutional operation using XNOR and bitcounts

(1) Binarizing Weight

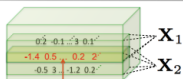


$$\frac{1}{n} \|\mathbf{W}\|_{\ell_1} = \alpha$$



(2) Binarizing Input

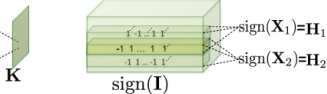
Inefficient



Redundant computations in overlapping areas

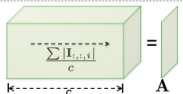
$$\frac{1}{n} \|\mathbf{X}_1\|_{\ell_1} = \beta_1$$

$$\frac{1}{n} \|\mathbf{X}_2\|_{\ell_1} = \beta_2$$



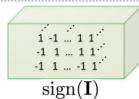
(3) Binarizing Input

Efficient

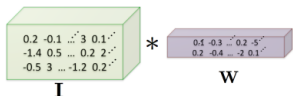


$$\mathbf{A} * \mathbf{K} = \beta_1$$

$$\mathbf{A} * \mathbf{K} = \beta_2$$



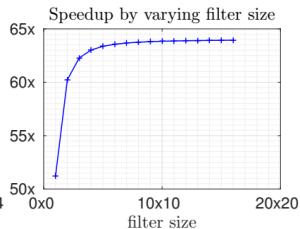
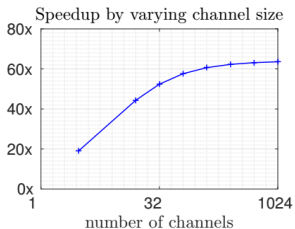
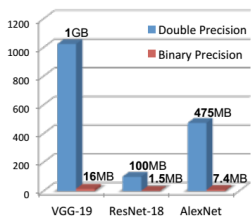
(4) Convolution with XNOR-Bitcount



$$\mathbf{I} * \mathbf{W} \approx \left[\text{sign}(\mathbf{I}) \circledast \text{sign}(\mathbf{W}) \right] \odot \mathbf{K} \odot \alpha$$

XNOR-net: evaluation

- Left: memory for binary weights is much smaller
- Middle, Right: around $60\times$ speed-up with 3×3 filters
- But, accuracy degrades compared to binary weight network



DoReFa-net: also quantization of gradient

- XNOR-net: not quantize gradients, so no speed-up during BP

DoReFa-net: also quantization of gradient

- XNOR-net: not quantize gradients, so no speed-up during BP
- Channel/kernel-wise scaling factors make bit convolution between gradients and weights impossible!
Solution: use a single scaling factor for all kernels per layer.

DoReFa-net: also quantization of gradient

- XNOR-net: not quantize gradients, so no speed-up during BP
- Channel/kernel-wise scaling factors make bit convolution between gradients and weights impossible!
Solution: use a single scaling factor for all kernels per layer.
- Adding random (uniform) noise during quantizing gradient is crucial; noise less than magnitude of quantization error
- Quantization of gradients allows to accelerate low bit-width network training on CPU, FPGA, ASIC, GPU

DoReFa-net: also quantization of gradient

- XNOR-net: not quantize gradients, so no speed-up during BP
- Channel/kernel-wise scaling factors make bit convolution between gradients and weights impossible!
Solution: use a single scaling factor for all kernels per layer.
- Adding random (uniform) noise during quantizing gradient is crucial; noise less than magnitude of quantization error
- Quantization of gradients allows to accelerate low bit-width network training on CPU, FPGA, ASIC, GPU

To make performance not degrade too much:

- Gradients require larger bit-width than activations
- Activations require larger bit-width than weights

Limitations of methods so far

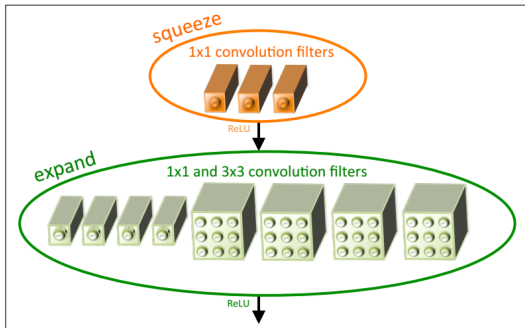
All above: still start from a large model!

Can we directly design light models?

Figures and tables in next 3 slides from Iandola et al., "SqueezeNet: AlexNet-level accuracy with 50X fewer parameters and 1.5MB model size", ICLR, 2017

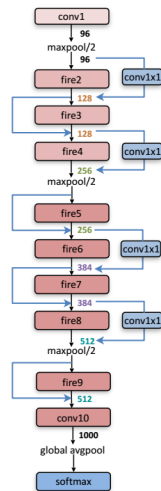
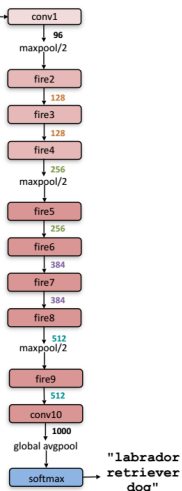
SqueezeNet

- SqueezeNet 'Fire module': squeeze layer + expand layer
- Squeeze layer: all are 1×1 filters, so reduce parameters
- Expand layer: mixed 1×1 and 3×3 filters
- Fewer filters in squeeze layer: so reduce input channels (to expand layer)



SqueezeNet

- SqueezeNet (left) and its variants



SqueezeNet

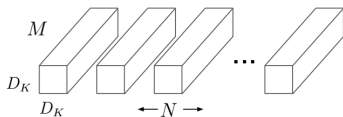
- 50× reduction in model size compared to AlexNet
- SqueezeNet can be 'compressed', resulting in 510× reduction in model size with no decrease in accuracy

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	50x	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	363x	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	510x	57.5%	80.3%

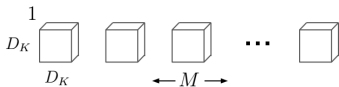
Figures and tables in next 4 slides from Howard et al., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv, 2017

MobileNet

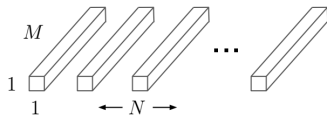
MobileNet: (a) is divided into (b) followed by (c)



(a) Standard convolution filters



(b) Depthwise convolution filters



(c) 1x1 pointwise convolution

MobileNet

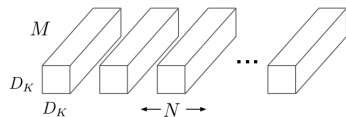
MobileNet: (a) is divided into (b) followed by (c)

Suppose input feature map size:
 $D_F \times D_F$, input channel #: M ,
 output channel #: N , kernel size:
 $D_K \times D_K$. Computation cost:

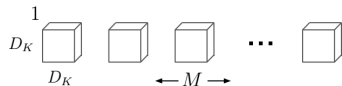
- (a) $D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$
- (b) $D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$
- (c) $N \cdot D_F \cdot D_F$

Computation reduction:

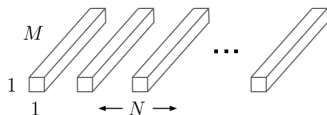
- $(b+c)/a = \frac{1}{N} + \frac{1}{D_K^2}$
- $D_K = 3$: 8 ~ 9 times less computation



(a) Standard convolution filters



(b) Depthwise convolution filters



(c) 1x1 pointwise convolution

MobileNet: structure

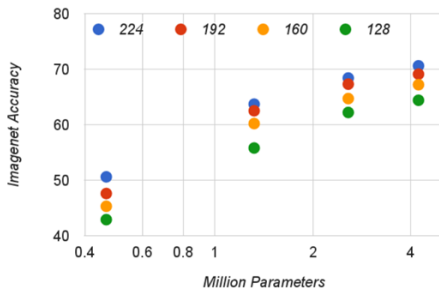
- 28 layers
- used BN+ReLU
- thinner model with αM and αN , where $0 < \alpha \leq 1$
- low-resolution input to further reduce computation

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× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
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	$1 \times 1 \times 1000$

MobileNet: result

- MobileNet have much smaller computation and parameters
- Accuracy reduced with thinner (x-axis) & smaller (color) input

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



MobileNet: result

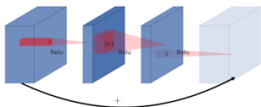
- MobileNet with similar accuracy but less computation and fewer parameters than VGG16 and GoogleNet
- $\alpha = 0.5$, input 160×160 : better than AlexNet while being 45 times smaller and 9.4 times less compute than AlexNet

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
0.50 MobileNet-160	60.2%	76	1.32
Squeezenet	57.5%	1700	1.25
AlexNet	57.2%	720	60

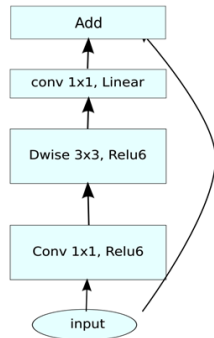
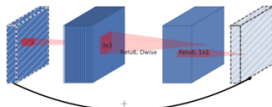
MobileNet V2

- MobileNet version 2: used bottleneck/inverted residual block (Figure b; right-side figure; table)
- Efficient memory use: expanded tensors (feature maps) inside each residual block are not necessarily stored in memory

(a) Residual block



(b) Inverted residual block



Input	Operator	Output
$h \times w \times k$	1x1 conv2d, ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

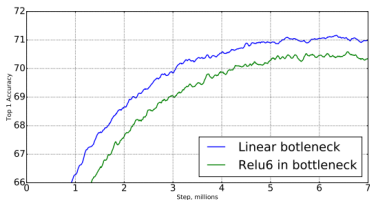
MobileNet V2: detail

- Parameters: t - expansion factor; c - output channel number; n - repeated block times; s - stride (for first block if repeated)

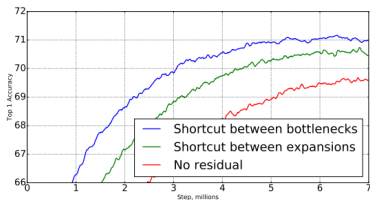
Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	-

MobileNet V2: result

- Nonlinearity in bottleneck (Fig. a, green) destroys information of low-dim manifold embedded in the higher-dim space
- Shortcut connecting bottlenecks performs better (Fig. b)



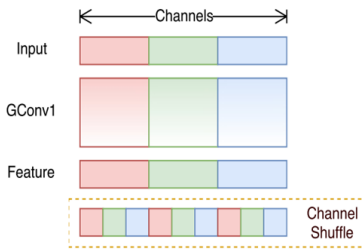
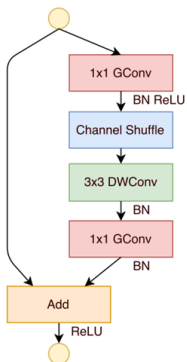
(a) Impact of non-linearity in the bottleneck layer.



(b) Impact of variations in residual blocks.

ShuffleNet

- Novelty 1: (regularly) shuffle output channels across groups after group convolution
- Novelty 2: depthwise convolution after channel shuffle
- Shuffling makes information from input channels flow to every group in the next group convolution



ShuffleNet: structure

- Totally 50 layers
- With constrained computation: the more groups divided, the more channels could be added, so more information encoded

Layer	Output size	KSize	Stride	Repeat	Output channels (g groups)				
					$g = 1$	$g = 2$	$g = 3$	$g = 4$	$g = 8$
Image	224×224				3	3	3	3	3
Conv1	112×112	3×3	2	1	24	24	24	24	24
MaxPool	56×56	3×3	2						
Stage2	28×28		2	1	144	200	240	272	384
	28×28		1	3	144	200	240	272	384
Stage3	14×14		2	1	288	400	480	544	768
	14×14		1	7	288	400	480	544	768
Stage4	7×7		2	1	576	800	960	1088	1536
	7×7		1	3	576	800	960	1088	1536
GlobalPool	1×1	7×7							
FC					1000	1000	1000	1000	1000
Complexity					143M	140M	137M	133M	137M

ShuffleNet: effect of group conv and shuffle

- Group conv ($g > 1$) is better than the one without ($g = 1$)

Model	Complexity (MFLOPs)	Classification error (%)				
		$g = 1$	$g = 2$	$g = 3$	$g = 4$	$g = 8$
ShuffleNet 1×	140	33.6	32.7	32.6	32.8	32.4
ShuffleNet 0.5×	38	45.1	44.4	43.2	41.6	42.3
ShuffleNet 0.25×	13	57.1	56.8	55.0	54.2	52.7

- Shuffles help! ('ShuffleNet $s\times$ ': scaling filters number s times)

Model	Cls err. (% , no shuffle)	Cls err. (% , shuffle)	Δ err. (%)
ShuffleNet 1x ($g = 3$)	34.5	32.6	1.9
ShuffleNet 1x ($g = 8$)	37.6	32.4	5.2
ShuffleNet 0.5x ($g = 3$)	45.7	43.2	2.5
ShuffleNet 0.5x ($g = 8$)	48.1	42.3	5.8
ShuffleNet 0.25x ($g = 3$)	56.3	55.0	1.3
ShuffleNet 0.25x ($g = 8$)	56.5	52.7	3.8

ShuffleNet: comparison with other models

- With similar computation complexity, ShuffleNet works better than popular CNN models, including MobileNet
- ShuffleNet is a backbone model, can be combined with others
- Better not due to more depth (last vs. 3rd last row)

Complexity (MFLOPs)	VGG-like	ResNet	Xception-like	ResNeXt	ShuffleNet (ours)
140	50.7	37.3	33.6	33.3	32.4 ($1\times, g = 8$)
38	-	48.8	45.1	46.0	41.6 ($0.5\times, g = 4$)
13	-	63.7	57.1	65.2	52.7 ($0.25\times, g = 8$)

Model	Complexity (MFLOPs)	Cls err. (%)	Δ err. (%)
1.0 MobileNet-224	569	29.4	-
ShuffleNet $2\times$ ($g = 3$)	524	26.3	3.1
ShuffleNet $2\times$ (with <i>SE</i> [13], $g = 3$)	527	24.7	4.7
0.75 MobileNet-224	325	31.6	-
ShuffleNet $1.5\times$ ($g = 3$)	292	28.5	3.1
0.25 MobileNet-224	41	49.4	-
ShuffleNet $0.5\times$ ($g = 4$)	38	41.6	7.8
ShuffleNet $0.5\times$ (shallow, $g = 3$)	40	42.8	6.6

ShuffleNet: comparison with other models

- ShuffleNet is a very light model!
- With similar accuracy, ShuffleNet is much more efficient
- e.g., theoretically 18 times faster than AlexNet (last row)

Model	Cls err. (%)	Complexity (MFLOPs)
VGG-16 [30]	28.5	15300
ShuffleNet $2\times$ ($g = 3$)	26.3	524
GoogleNet [33]*	31.3	1500
ShuffleNet $1\times$ ($g = 8$)	32.4	140
AlexNet [21]	42.8	720
SqueezeNet [14]	42.5	833
ShuffleNet $0.5\times$ ($g = 4$)	41.6	38

Summary

- Efficiency is crucial for many applications!
- Ideas: reduce, quantize, compact
- Often trade off between efficiency and accuracy
- A new and active research topic

Further reading:

- Zhu et al., 'Trained ternary quantization', ICLR, 2017
- Luo et al., 'Thinnet: a filter level pruning method for deep neural network compression', ICCV, 2017
- Yu et al., 'Slimmable neural networks', ICLR, 2019